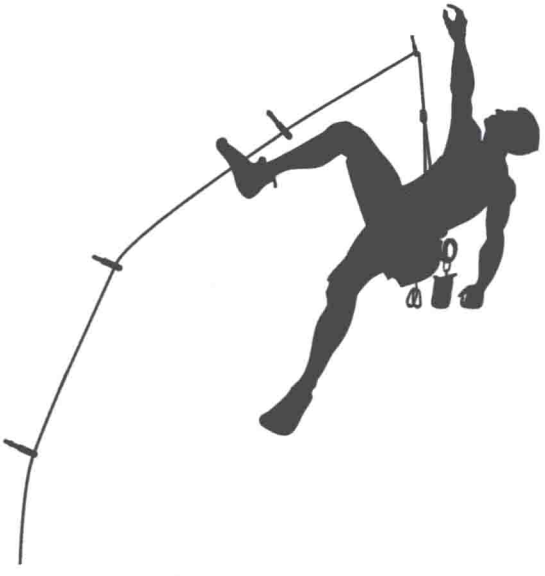
HZ BOOKS
华章科技

国内首部UEFI专著，资深UEFI专家兼布道者撰写，英特尔中国研究院院长吴甘沙及大数据和数据经济实验室研究总监周鑫联袂推荐

以实战为导向，全面介绍UEFI的使用、深入剖析UEFI的原理，为开发UEFI应用和驱动程序提供了翔实的指导

實戰



戴正华 著

UEFI 原理与编程

机械工业出版社
China Machine Press

提供各种书籍pdf下载，如有需要，请联系 QQ: 461573687

PDF制作说明:

本人可以提供各种PDF电子书资料，计算机类，文学，艺术，设计，医学，理学，经济，金融，等等。质量都很清晰，而且每本100%都带书签和目录，方便读者阅读观看，只要您提供给我书的相关信息，一般我都能找到，如果您有需求，请联系我 **QQ: 461573687**, 或者 **QQ: 2404062482**。

本人已经帮助了上万人找到了他们需要的PDF，其实网上有很多PDF,大家如果在网上不到的话，可以联系我QQ。因PDF电子书都有版权，请不要随意传播，最近pdf也越来越难做了，希望大家尊重下个人劳动，谢谢！

备用QQ:2404062482

實戰

UEFI: Principles and Programming

UEFI 原理与编程

戴正华 著

机械工业出版社
China Machine Press

图书在版编目（CIP）数据

UEFI 原理与编程 / 戴正华著 . —北京：机械工业出版社，2015.1
（实战）

ISBN 978-7-111-48729-6

I. U…　II. 戴…　III. 程序设计　IV. TP311.1

中国版本图书馆 CIP 数据核字（2014）第 282707 号

UEFI 原理与编程

出版发行：机械工业出版社（北京市西城区百万庄大街 22 号　邮政编码：100037）
责任编辑：高婧雅　　责任校对：殷　虹
印　　刷：三河市宏图印务有限公司　　版　　次：2015 年 1 月第 1 版第 1 次印刷
开　　本：186mm × 240mm　1/16　　印　　张：26
书　　号：ISBN 978-7-111-48729-6　　定　　价：89.00 元

凡购本书，如有缺页、倒页、脱页，由本社发行部调换
客服热线：（010）88378991　88361066　　投稿热线：010）88379604
购书热线：（010）68326294　88379649　68995259　　读者信箱：hzjsj@hzbook.com

版权所有 • 侵权必究
封底无防伪标均为盗版
本书法律顾问：北京大成律师事务所　韩光 / 邹晓东

Preface 序

收到书稿，看到熟悉的作者名字，立刻有种开始仔细阅读的冲动。

翻完最后一页，掩卷体会，其中充满了正华苦心钻研技术的精神与回馈社会的热情。

在 IT 行业各个细分领域里，系统程序员需要埋头于汇编语言 /C 语言的编辑器、调试器和体系结构的大部头手册，堪称最枯燥、最辛苦的工种之一。在系统领域取得成就的程序员需要有丰富的知识、扎实的技巧和踏实的态度，缺一不可。正华在我们团队开始系统程序员的职业生涯，他从一个普通的 C 程序员，以孜孜不倦的学习态度，快速成长为一个出色的编程语言和编译器系统的开发骨干。然后，他去了更广阔的天地，直到今天成为 UEFI 专家。正华的钻研精神和成长经历，是本书在专业内容以外，更值得读者学习和体会的精神食粮。

UEFI 是一个很出色的方向。尤其是对有志于学习计算机体系结构和操作系统的初学者来说，UEFI 是一个复杂度适中、内容涵盖度超高的学习平台。我们这一代的系统程序员，都是从奔腾 /586 CPU、DoS、Windows95、Linux Kernel 2.0 时代成长起来的。仔细阅读、修改、调试这些操作系统平台，是我们成长最有效的途径。但是，现代主流桌面操作系统，比如 Windows、Linux Ubuntu、Fedora，经过 20 多年的发展，功能超级丰富，系统设计超级复杂、代码量超级多，已经不适合作为体系结构和操作系统初学者的学习对象。UEFI 在恰当的时候填补了这个空缺。相比它的前辈 BIOS，UEFI 已经涵盖现代操作系统的核心内核功能和外设模块，可以视为一个最最简化版的操作系统。学通、学透 UEFI，可以帮助操作系统初学者快速入门。同时，对于安全领域和 X86 嵌入式领域来说，UEFI 是不可或缺的知识构成部分。

本书是正华热情回馈社会的成果。书里详细、具体地再现了一个学习者对 UEFI 的学习历程。相对于枯燥的手册，这种历程回顾方式的学习材料，更具有可阅读性、可学习性、可

操作性。事无巨细地列举了所有环境细节、命令参数，能极大地帮助初学者绕过障碍，专注于核心内容的学习，缩短学习过程。这种貌似简单，实则烦琐的学习总结，需要极大的精力和时间来准备与撰写，正华的回馈精神体现无疑。

希望读者们从内容中获取知识的同时，也体会学习和回馈的精神。

周鑫

英特尔中国研究院　大数据和数据经济实验室　研究总监

2014 年 11 月 23 日

Preface 前　言

BIOS 已经逐渐成为历史，UEFI 目前开始全面取代 BIOS。

UEFI 全称统一可扩展固件接口，是 UEFI 论坛发布的一种操作系统和平台固件之间的标准。它之所以能迅速取代 BIOS，源于硬件平台的发展以及 UEFI 相对于 BIOS 的巨大优势。UEFI 可编程性好，可扩展性好，性能高，安全性高。

随着 64 位 CPU 取代 32 位 CPU，UEFI 也完成了对 BIOS 的取代。市场已经完成了这种转变，对固件开发者来说，这是一种挑战，固件开发模式已经发生了巨大的改变；这也是一个机遇，UEFI 为计算机系统提供了更丰富的功能，为开发者提供了更强大的开发手段。

为什么写这本书

21 世纪什么最重要？大家都知道答案。人才是需要培养的。培养 UEFI 开发者最重要的是相关开发资料。但是目前 UEFI 相关的开发资料十分匮乏。UEFI 是一种全新的技术，是对 BIOS 的一种革命，BIOS 时代的技术积累很难转移给 UEFI。在 BIOS 时代，BIOS 开发采用汇编语言并且与硬件特性息息相关，技术被几大公司垄断，进入 BIOS 开发领域的门槛高，离开的代价大，相关的技术资料很难在广大程序员中流传。进入 UEFI 时代后，这种开发特点因为惯性的作用依然会延续一段时间。

UEFI 开发对广大 C/C++ 开发者敞开了大门，进入和离开 UEFI 的门槛降低了很多。相信会有更多的开发者进入和离开固件开发领域，这种流动性也会促进这个领域的繁荣，相关的开发资料也会越来越多，越来越精彩。

本书特色

本书是国内第一本介绍 UEFI 开发的书籍，希望这本书能改善 UEFI 开发者无处学习的困境。本书以实战为主，辅以相关理论知识，重要的章节还附有完整的应用程序。力图让读

者不仅知道如何开发 UEFI 程序，还让读者了解为什么这样编程。本书提供了丰富的源代码，让读者可以在实践中快速掌握编程要点。

读者对象

本书内容循序渐进，非常适合刚刚接触 UEFI 开发的初学者、大专院校在校的学生，也适合对 UEFI 有一定了解的专业开发者。

如何阅读本书

本书假定读者有基本的 C 和 C++ 知识，但并不要求读者有固件开发的相关知识。本书将一步一步引导读者熟悉 UEFI，随着本书的不断深入，使读者成为 UEFI 开发专家。

本书从实战的角度介绍如何开发 UEFI 应用和驱动，用生动的实例介绍如何使用 UEFI 提供的服务，同时我们将讲述所需的理论知识以及 UEFI 的内部实现。本书既可以作为 UEFI 爱好者的入门教材，也可以帮 UEFI 开发者更加深入了解 UEFI 内部实现。

本书不是 UEFI 开发的参考手册，所以读者在学习过程中最好准备 UEFI Spec 2.4 以备参考。本书提供了大量的实例程序，读者还需下载 TianoCore 的源码，边学习边实践。

本书内容编排如下：

第 1 章 UEFI 概述介绍 UEFI 发展的历史及 UEFI 理论知识，UEFI 系统启动到结束可分为 7 个过程，本章从程序开发的角度阐述了这个过程的执行流程。

第 2 章 介绍 UEFI 开发环境的搭建，EDK2 提供了 Linux、Windows、Cygwin 等多种开发环境，本章逐步讲述了 Linux 和 Windows 下 UEFI 开发环境的搭建，如何制作 OVMF 固件，以及如何制作启动盘。

第 3 章 介绍 EDK2 工程模块及包的概念，主要包括 .inf 文件、.dsc 文件和 .dec 文件的格式与用法。其中，主要的工程模块包括 UEFI 应用程序模块、驱动程序模块、库模块、Shell 应用程序模块。本章最后讲述了如何在模拟器下调试应用程序。

第 4 章 介绍 Protocol 的概念和用法。UEFI 提供的绝大多数服务都是以 Protocol 的形式提供的，由此可见其重要性。

第 5 章 介绍 UEFI 中的基础服务，包括系统表，启动服务和运行时服务。系统表是应用程序进入内核的接口，启动服务用于在启动过程中管理计算机软硬件资源，运行时服务是为操作系统访问固件而提供的服务。

第 6 章 介绍了事件及异步操作方式。除了介绍事件的使用方法，本章还介绍了事件及异步操作在 UEFI 内核的实现机制。最后本章以具体实例介绍了如何使用键盘、鼠标和定时

器事件。

第 7 章 介绍了 GPT 硬盘以及文件系统和文件的操作方法。相比 BIOS，UEFI 的一个重大进步就是开始支持文件系统，FAT32 文件系统是 UEFI 内建文件系统。本章重点讲述 FAT32 文件系统之上的文件操作。

第 8 章 以视频解码服务为例介绍如何利用 Protocol 提供服务。服务型 Protocol 在 UEFI 中占有重要地位，服务型 Protocol 的开发是 UEFI 开发的一项基本功，一定要熟练掌握。

第 9 章 介绍驱动开发模型，并以 AC97 驱动为例介绍开发设备驱动的具体步骤。

第 10 章 介绍如何支持 C++ 语言特性。UEFI 内核是 EDK2 采用 C 语言实现的，UEFI 提供的服务也是以 C 语言形式提供的，因而 C 天然适用于 UEFI 开发，但 C++ 会使得开发更有效率。本章首先讲述了如何使用 C++ 基础语法开发 UEFI 应用，然后讲述如何支持全局构造函数、析构函数，支持 new、delete 等操作符，支持标准模板库等高级特性。

第 11 章 介绍开发 GUI 的基础知识，包括在 UEFI 中如何使用字符串资源、字体和图像。通过使用字符串资源，可以实现字符串的本地化。本地化后，字符串的显示需要字体的支持，EDK2 默认支持英文和法文字符的显示，要显示其他语言的字符串还需要开发者自行开发相应字符的字体。

第 12 章 以视频播放器为例介绍 GUI 开发。目前还没有成熟的 GUI 库可以用于 UEFI 开发。本章从零开始利用第 11 章介绍的基础知识开发 GUI 系统，主要分两大部分：GUI 事件的派遣和响应；GUI 控件的绘制。

第 13 章 介绍多任务的开发。本章分为两大块：多核和多线程。首先主要介绍 MPP(MP Services Protocol)，MPP 用于管理多核，本节重点讲述了 MPP 如何在其他 CPU 核心上执行任务。UEFI 没有提供对多线程的支持，对此我们讲述了如何使用 LongJmp 技术实现简单的多线程。

第 14 章 介绍如何开发网络应用。本章首先简单介绍了 UEFI 提供的网络协议栈，然后讲述了使用 TCP 协议开发网络应用的基本框架。

第 15 章 介绍如何使用 C 标准库中的函数。

第 16 章 介绍应用程序中的 Shell 服务、Shell 脚本的语法以及常用 Shell 命令。

资源和勘误

由于笔者水平与时间有限，书中可能出现一些错误或不准确的地方，恳请读者批评、斧正。如果您对本书有任何的意见和指正，请发邮件至 djx.zhenghua@gmail.com。本书附带的

源代码，可以从如下网址获取：https://uefi-programming-guider.googlecode.com/svn/trunk/，或者从华章网站（www.hzbook.com）下载。

致谢

感谢华章公司的高婧雅与杨福川付出的巨大努力，高编辑耐心、细致地反复审阅书稿，使得本书从粗疏渐渐变得精准。

感谢宋风龙在本书写作过程中给予的技术支持。

本书最早起始于我在博客园发表的几篇博客，感谢博客园的众多网友的支持和意见，你们的建议让我受益颇深。

也要感谢工作在 CryptoMill 的同事。

特别鸣谢

最后要特别感谢我的妻子张一茁女士。从初稿到定稿近一年的时间里，我将大部分的节假日和周末用在了这本书上，这本书的完成，离不开妻子的付出和支持。

戴正华

Contents 目　录

序
前　言
第 1 章　UEFI 概述 …… 1
1.1　BIOS 的前世今生 …… 1
1.1.1　BIOS 在计算机系统中的作用 …… 1
1.1.2　BIOS 缺点 …… 2
1.2　初识 UEFI …… 2
1.2.1　UEFI 系统组成 …… 3
1.2.2　UEFI 的优点 …… 4
1.2.3　UEFI 系统的启动过程 …… 5
1.3　本章小结 …… 12
第 2 章　UEFI 开发环境搭建 …… 14
2.1　配置 Windows 开发环境 …… 14
2.1.1　安装所需开发工具 …… 15
2.1.2　配置 EDK2 开发环境 …… 15
2.1.3　编译 UEFI 模拟器和 UEFI 工程 …… 17
2.1.4　运行模拟器 …… 19
2.2　配置 Linux 开发环境 …… 21
2.2.1　安装所需开发工具 …… 22
2.2.2　配置 EDK2 开发环境 …… 22
2.2.3　编译 UEFI 模拟器和 UEFI 工程 …… 23
2.2.4　运行模拟器 …… 24
2.3　OVMF 的制作和使用 …… 25
2.4　UEFI 的启动 …… 27
2.5　本章小结 …… 28
第 3 章　UEFI 工程模块文件 …… 29
3.1　标准应用程序工程模块 …… 30
3.1.1　入口函数 …… 30
3.1.2　工程文件 …… 31
3.1.3　编译和运行 …… 37
3.1.4　标准应用程序的加载过程 …… 37
3.2　其他类型工程模块 …… 43
3.2.1　Shell 应用程序工程模块 …… 43
3.2.2　使用 main 函数的应用程序工程模块 …… 46
3.2.3　库模块 …… 47
3.2.4　UEFI 驱动模块 …… 49
3.2.5　模块工程文件小结 …… 50
3.3　包及 .dsc、.dec、.fdf 文件 …… 51
3.3.1　.dsc 文件 …… 51

3.3.2 .dec 文件 …… 56
3.4 调试 UEFI …… 59
3.5 本章小结 …… 61

第 4 章 UEFI 中的 Protocol …… 62

4.1 Protocol 在 UEFI 内核中的表示 …… 64
4.2 如何使用 Protocol 服务 …… 65
4.2.1 OpenProtocol 服务 …… 66
4.2.2 HandleProtocol 服务 …… 67
4.2.3 LocateProtocol 服务 …… 69
4.2.4 LocateHandleBuffer 服务 …… 69
4.2.5 其他一些使用 Protocol 的服务 …… 71
4.2.6 CloseProtocol 服务 …… 72
4.3 Protocol 服务示例 …… 73
4.4 本章小结 …… 75

第 5 章 UEFI 的基础服务 …… 76

5.1 系统表 …… 76
5.1.1 系统表的构成 …… 77
5.1.2 使用系统表 …… 79
5.2 启动服务 …… 82
5.2.1 启动服务的构成 …… 82
5.2.2 启动服务的生存期 …… 91
5.3 运行时服务 …… 93
5.4 本章小结 …… 102

第 6 章 事件 …… 103

6.1 事件函数 …… 104
6.1.1 等待事件的服务 WaitForEvent …… 105
6.1.2 生成事件的服务 CreateEvent …… 106
6.1.3 CreateEventEx 服务 …… 110
6.1.4 事件相关的其他函数 …… 112
6.2 定时器事件 …… 113
6.3 任务优先级 …… 114
6.3.1 提升和恢复任务优先级 …… 115
6.3.2 UEFI 中的时钟中断 …… 116
6.3.3 UEFI 事件 Notification 函数的派发 …… 126
6.4 鼠标和键盘事件示例 …… 127
6.5 本章小结 …… 128

第 7 章 硬盘和文件系统 …… 129

7.1 GPT 硬盘 …… 129
7.1.1 基于 MBR 分区的传统硬盘 …… 129
7.1.2 GPT 硬盘详解 …… 130
7.2 设备路径 …… 134
7.3 硬盘相关的 Protocol …… 139
7.3.1 BlockIo 解析 …… 140
7.3.2 BlockIo2 解析 …… 142
7.3.3 DiskIo 解析 …… 146
7.3.4 DiskIo2 解析 …… 147
7.3.5 PassThrough 解析 …… 150
7.4 文件系统 …… 152
7.5 文件操作 …… 153
7.5.1 打开文件 …… 154
7.5.2 读文件 …… 156
7.5.3 写文件 …… 159
7.5.4 关闭文件（句柄）…… 160
7.5.5 其他文件操作 …… 160

7.5.6 异步文件操作 …… 162
7.5.7 EFI_SHELL_PROTOCOL 中的文件操作 …… 166
7.6 本章小结 …… 170

第 8 章 开发 UEFI 服务 …… 171

8.1 Protocol 服务接口设计 …… 172
8.2 Protocol 服务的实现 …… 174
8.3 服务型驱动的框架 …… 178
8.4 ffmpeg 的移植与编译 …… 179
8.4.1 libavcodec 的建立和移植 …… 181
8.4.2 其他库的建立与移植 …… 182
8.4.3 在驱动型服务中使用 StdLib …… 186
8.5 使用 Protocol 服务 …… 188
8.6 本章小结 …… 190

第 9 章 开发 UEFI 驱动 …… 191

9.1 UEFI 驱动模型 …… 192
9.1.1 EFI Driver Binding Protocol 的构成 …… 192
9.1.2 EFI Component Name Protocol 的作用和构成 …… 196
9.2 编写设备驱动的步骤 …… 197
9.3 PCI 设备驱动基础 …… 199
9.4 AC97 控制器芯片的控制接口 …… 202
9.5 AC97 驱动 …… 206
9.5.1 AC97 驱动的驱动服务 EFI_AUDIO_PROTOCOL …… 206
9.5.2 AC97 驱动的框架部分 …… 213
9.5.3 AC97 驱动实验 …… 220
9.6 本章小结 …… 221

第 10 章 用 C++ 开发 UEFI 应用 …… 222

10.1 从编译器角度看 C 与 C++ 的差异 …… 222
10.2 在 EDK2 中支持 C++ …… 224
10.2.1 使 EDK2 支持 C++ 基本特性 …… 224
10.2.2 在 Windows 系统下的程序启动过程 …… 226
10.2.3 在 Windows 系统下支持全局构造和析构 …… 229
10.2.4 在 Linux 系统下的程序启动过程 …… 231
10.2.5 在 Linux 系统下支持全局构造和析构 …… 240
10.2.6 支持 new 和 delete …… 242
10.2.7 支持 STL …… 243
10.3 GcppPkg 概览 …… 243
10.4 测试 GcppPkg …… 246
10.5 本章小结 …… 248

第 11 章 GUI 基础 …… 249

11.1 字符串 …… 249
11.1.1 字符串函数 …… 249
11.1.2 字符串资源 …… 251
11.1.3 管理字符串资源 …… 255
11.2 管理语言 …… 260
11.3 包列表 …… 262
11.4 图形界面显示 …… 263
11.4.1 显示模式 …… 264
11.4.2 Block Transfer（Blt）传输图像 …… 267

11.4.3 在图形界面下显示字符串 …269
11.5 用 SimpleFont 显示中文 ……272
11.5.1 SimpleFont 格式 ……273
11.5.2 如何生成字体文件 ……275
11.5.3 如何注册字体文件 ……276
11.6 开发 SimpleFont 字库程序 ……277
11.7 字体 Font ……278
11.7.1 Font 的格式 ……279
11.7.2 字体包的格式 ……279
11.7.3 为什么 Font 性能高于 SimpleFont ……281
11.8 本章小结 ……284

第 12 章 GUI 应用程序 ……285

12.1 UEFI 事件处理 ……285
12.1.1 键盘事件 ……285
12.1.2 鼠标事件 ……292
12.1.3 定时器事件 ……293
12.1.4 UI 事件服务类 ……294
12.2 事件处理框架 ……297
12.3 鼠标与控件的绘制 ……302
12.3.1 鼠标的绘制 ……303
12.3.2 控件的绘制 ……305
12.4 控件系统包 GUIPkg ……306
12.5 简单视频播放器的实现 ……309
12.6 本章小结 ……315

第 13 章 深入了解多任务 ……317

13.1 多处理器服务 ……317
13.1.1 EFI_MP_SERVICES_PROTOCOL 功能及用法 …317
13.1.2 启动 AP 的过程 ……324
13.2 内联汇编基础和寄存器上下文的保存与恢复 ……333
13.2.1 内联汇编基础 ……333
13.2.2 寄存器上下文的保存与恢复 ……335
13.3 多线程 ……336
13.3.1 生成线程 ……337
13.3.2 调度线程 ……340
13.3.3 等待线程结束 ……341
13.3.4 SimpleThread 服务 ……341
13.4 本章小结 ……345

第 14 章 网络应用开发 ……346

14.1 在 UEFI 中使用网络 ……348
14.2 使用 EFI_TCP4_PROTOCOL …350
14.2.1 生成 Socket 对象 ……352
14.2.2 连接 ……356
14.2.3 传输数据 ……358
14.2.4 关闭 Socket ……361
14.2.5 测试 Socket ……362
14.3 本章小结 ……363

第 15 章 使用 C 标准库 ……364

15.1 为什么使用 C 标准库函数 ……364
15.2 实现简单的 Std 函数 ……365
15.2.1 简单标准库函数包 sstdPkg ……366
15.2.2 使用 sstdPkg ……368
15.3 使用 EDK2 的 StdLib ……369
15.3.1 main 函数工程 ……369

15.3.2 非 main 函数工程 …… 374
15.4 本章小结 …… 376

第 16 章 Shell 及常用 Shell 命令 …… 377

16.1 Shell 的编译与执行 …… 377
16.2 Shell 服务 …… 379
16.3 Shell 脚本 …… 385
16.3.1 Shell 脚本语法简介 …… 385
16.3.2 自动运行指定应用程序 …… 388
16.4 Shell 内置命令 …… 388
16.4.1 调试设备的相关命令 …… 388
16.4.2 驱动相关命令 …… 390
16.4.3 网络相关命令 …… 392
16.5 本章小结 …… 394

附录 A UEFI 常用术语及简略语 …… 395

附录 B RFC 4646 常用语言列表 …… 397

附录 C 状态值 …… 398

附录 D 参考资料 …… 400

第 1 章 Chapter 1

UEFI 概述

从我们按下开机键到进入操作系统，对用户来说是一个等待的过程，而对计算机来说是一个复杂的过程。在 BIOS 时代，这个过程重复了一年又一年，操作系统已经从枯燥的文本界面演化到丰富多彩的图形界面，BIOS 却一直延续着枯燥的过程，BIOS 设置也一直是单调的蓝底白字格式。BIOS 的坚持出于两个原因：外因是 BIOS 基本能满足市场需求，内因是 BIOS 的设计使得 BIOS 的升级和扩增变得非常困难。随着 64 位 CPU 逐渐取代 32 位 CPU，BIOS 越来越不能满足市场的需求，这使得 UEFI 作为 BIOS 的替代者，逐渐开始取代 BIOS 的地位。

1.1 BIOS 的前世今生

BIOS 诞生于 1975 年的 CP/M 计算机，诞生之初，也曾是一种先进的技术，并且是系统中相当重要的一个部分。随着 IBM PC 兼容机的流行，BIOS 也逐渐发展起来。它“统治”了计算机系统 20 多年的时间，在这段时间里，CPU 每 18 个月性能提升一倍。计算机软硬件都已经繁衍了无数代，BIOS 诞生之初与之配套的 8 位 CPU 和 DOS 系统都已经退出历史舞台，而 BIOS 依然顽强地存在于计算机中。

1.1.1 BIOS 在计算机系统中的作用

BIOS 全称为“基本输入 / 输出系统”，它是存储在主板 ROM 里的一组程序代码，这些代码包括：

- 加电自检程序，用于开机时对硬件的检测。
- 系统初始化代码，包括硬件设备的初始化、创建 BIOS 中断向量等。
- 基本的外围 I/O 处理的子程序代码。
- CMOS 设置程序。

BIOS 程序运行在 16 位实模式下，实模式下最大的寻址范围是 1MB，0x0C0000 ~ 0x0FFFFF 保留给 BIOS 使用。开机后，CPU 跳到 0x0FFFF0 处执行，一般这里是一条跳转指令，跳到真正的 BIOS 入口处执行。BIOS 代码首先做的是“加电自检”（Power On Self Test，POST），主要是检测关机设备是否正常工作，设备设置是否与 CMOS 中的设置一致。如果发现硬件错误，则通过喇叭报警。POST 检测通过后初始化显示设备并显示显卡信息，接着初始化其他设备。设备初始化完毕后开始检查 CPU 和内存并显示检测结果。内存检测通过以后开始检测标准设备，例如硬盘、光驱、串口设备、并口设备等。然后检测即插即用设备，并为这些设备分配中断号、I/O 端口和 DMA 通道等资源。如果硬件配置发生变化，那么这些变化的配置将更新到 CMOS 中。随后，根据配置的启动顺序从设备启动，将启动设备主引导记录的启动代码通过 BIOS 中断读入内存，然后控制权交到引导程序手中，最终引导进入操作系统。

1.1.2 BIOS 缺点

随着 CPU 及其他硬件设备的革新，BIOS 逐渐成为计算机系统发展的瓶颈，主要体现在如下几个方面：

1）**开发效率低**：大部分 BIOS 代码使用汇编开发，开发效率不言而喻。汇编开发的另一个缺点是使得代码与设备的耦合程度太高，代码受硬件变化的影响大。

2）**性能差**：BIOS 基本输入 / 输出服务需要通过中断来完成，开销大，并且 BIOS 没有提供异步工作模式，大量的时间消耗在等待上。

3）**功能扩展性差，升级缓慢**：BIOS 代码采用静态链接，增加硬件功能时，必须将 16 位代码放置在 0x0C0000 ~ 0x0DFFFF 区间，初始化时将其设置为约定的中断处理程序。而且 BIOS 没有提供动态加载设备驱动的方案。

4）**安全性**：BIOS 运行过程中对可执行代码没有安全方面的考虑。

5）**不支持从硬盘 2 TB 以上的地址引导**：受限于 BIOS 硬盘的寻址方式，BIOS 硬盘采用 32 位地址，因而引导扇区的最大逻辑块地址是 2^{32}（换算成字节地址，即 $2^{32} \times 512$=2TB）。

1.2 初识 UEFI

UEFI（Unified Extensible Firmware Interface，统一可扩展固件接口）定义了操作系统和平台固件之间的接口，它是 UEFI Forum 发布的一种标准。它只是一种标准，没有提供实现。

其实现由其他公司或开源组织提供，例如英特尔公司提供的开源 UEFI 实现 TianoCore 和 Phoenix 公司的 SecureCore Tiano。UEFI 实现一般可分为两部分：

- 平台初始化（遵循 Platform Initialization 标准，同样由 UEFI Forum 发布）。
- 固件 – 操作系统接口。

UEFI 发端于 20 世纪 90 年代中期的安腾系统。相对于当时流行的 32 位 IA32 系统，安腾是一种全新的 64 位系统，BIOS 的限制对这种 64 位系统变得不可接受（BIOS 也正是随着 32 位系统被 64 位系统取代而逐渐退出市场的）。因为 BIOS 在 64 位系统上的限制，1998 年英特尔公司发起了 Intel Boot Initiative 项目，后来更名为 EFI（Extensible Firmware Interface）。2003 年英特尔公司的安腾 CPU 计划遭到 AMD 公司的 x86_64 CPU 顽强阻击，x86_64 CPU 时代到来，市场更愿意接受渐进式的变化，英特尔公司也开始发布兼容 32 位系统的 x86_64 CPU。安腾虽然没有像预期那样独占市场，EFI 却显示出了它的价值。2005 年，英特尔公司联合微软、AMD、联想等 11 家公司成立了 Unified EFI Forum，负责制定统一的 EFI 标准。第一个 UEFI 标准——UEFI 2.0 在 2006 年 1 月发布。目前最新的 UEFI 标准是 2013 年发布的 UEFI 2.4。

1.2.1　UEFI 系统组成

UEFI 提供给操作系统的接口包括启动服务（Boot Services，BS）和运行时服务（Runtime Service，RT）以及隐藏在 BS 之后的丰富的 Protocol。BS 和 RT 以表的形式（C 语言中的结构体）存在。UEFI 驱动和服务以 Protocol 的形式通过 BS 提供给操作系统。

图 1-1 展示了基于 EFI 的计算机系统的组成。

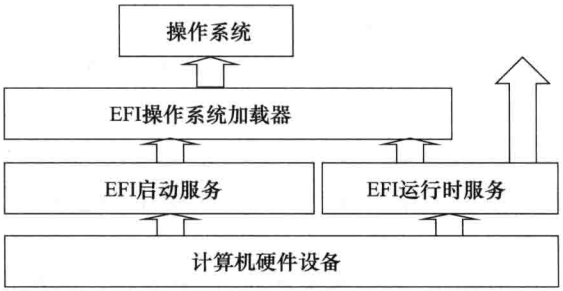


图 1-1　EFI 系统组成

从操作系统加载器（OS Loader）被加载，到 OS Loader 执行 ExitBootServices() 的这段时间，是从 UEFI 环境向操作系统过渡的过程。在这个过程中，OS Loader 可以通过 BS 和 RT 使用 UEFI 提供的服务，将计算机系统资源逐渐转移到自己手中，这个过程称为 TSL（Transient System Load）。

当 OS Loader 完全掌握了计算机系统资源时，BS 也就完成了它的使命。OS Loader 调用 ExitBootServices() 结束 BS 并回收 BS 占用的资源，之后计算机系统进入 UEFI Runtime 阶段。

在 Runtime 阶段只有运行时服务继续为 OS 提供服务，BS 已经从计算机系统中销毁。

在 TSL 阶段，系统资源通过 BS 管理，BS 提供的服务如下。

1）**事件服务**：事件是异步操作的基础。有了事件的支持，才可以在 UEFI 系统内执行并发操作。

2）**内存管理**：主要提供内存的分配与释放服务，管理系统内存映射。

3）**Protocol 管理**：提供了安装 Protocol 与卸载 Protocol 的服务，以及注册 Protocol 通知函数（该函数在 Protocol 安装时调用）的服务。

4）**Protocol 使用类服务**：包括 Protocol 的打开与关闭，查找支持 Protocol 的控制器。例如要读写某个 PCI 设备的寄存器，可以通过 OpenProtocol 服务打开这个设备上的 PciIo Protocol，用 PciIo->Io.Read() 服务可以读取这个设备上的寄存器。

5）**驱动管理**：包括用于将驱动安装到控制器的 connect 服务，以及将驱动从控制器上卸载的 disconnect 服务。例如，启动时，如果我们需要网络支持，则可以通过 loadImage 将驱动加载到内存，然后通过 connect 服务将驱动安装到设备。

6）**Image 管理**：此类服务包括加载、卸载、启动和退出 UEFI 应用程序或驱动。

7）**ExitBootServices**：用于结束启动服务。

RT 提供的服务主要包括如下几个方面。

1）**时间服务**：读取 / 设定系统时间。读取 / 设定系统从睡眠中唤醒的时间。

2）**读写 UEFI 系统变量**：读取 / 设置系统变量，例如 BootOrder 用于指定启动项顺序。通过这些系统变量可以保存系统配置。

3）**虚拟内存服务**：将物理地址转换为虚拟地址。

4）**其他服务**：包括重启系统的 ResetSystem，获取系统提供的下一个单调单增值等。

1.2.2 UEFI 的优点

UEFI 能迅速取代 BIOS，得益于 UEFI 相对 BIOS 的几大优势。

（1）UEFI 的开发效率

BIOS 开发一般采用汇编语言，代码多是硬件相关的代码。而在 UEFI 中，绝大部分代码采用 C 语言编写，UEFI 应用程序和驱动甚至可以使用 C++ 编写。UEFI 通过固件 - 操作系统接口（BS 和 RT 服务）为 OS 和 OS 加载器屏蔽了底层硬件细节，使得 UEFI 上层应用可以方便重用。

（2）UEFI 系统的可扩展性

UEFI 系统的可扩展性体现在两个方面：一是驱动的模块化设计；二是软硬件升级的兼容性。

大部分硬件的初始化通过 UEFI 驱动实现。每个驱动是一个独立的模块，可以包含在固

件中，也可以放在设备上，运行时根据需要动态加载。

UEFI 中每个表、每个 Protocol(包括驱动）都有版本号，这使得系统的平滑升级变得简单。

（3）UEFI 系统的性能

相比 BIOS，UEFI 有了很大的性能提升，从启动到进入操作系统的时间大大缩短。性能的提高源于以下几个方面：

1）UEFI 提供了异步操作。基于事件的异步操作，提高了 CPU 利用率，减少了总的等待时间。

2）UEFI 舍弃了中断这种比较耗时的操作外部设备的方式，仅仅保留了时钟中断。外部设备的操作采用“事件＋异步操作”完成。

3）可伸缩的遍历设备的方式，启动时可以仅仅遍历启动所需的设备，从而加速系统启动。

（4）UEFI 系统的安全性

UEFI 的一个重要突破就是其安全方面的考虑。当系统的安全启动功能被打开后，UEFI 在执行应用程序和驱动前会先检测程序和驱动的证书，仅当证书被信任时才会执行这个应用程序或驱动。UEFI 应用程序和驱动采用 PE/COFF 格式，其签名放在签名块中。

1.2.3　UEFI 系统的启动过程

UEFI 系统的启动遵循 UEFI 平台初始化（PlatformInitialization）标准[㊀]。UEFI 系统从加电到关机可分为 7 个阶段：

SEC（安全验证）→ PEI（EFI 前期初始化）→ DXE（驱动执行环境）
→ BDS（启动设备选择）→ TSL（操作系统加载前期）
→ RT（Run Time）
→ AL（系统灾难恢复期）

图 1-2 展示了 UEFI 系统从加电到关机的 7 个阶段[㊁]（以图中竖线为界）。

前三个阶段是 UEFI 初始化阶段，DXE 阶段结束后 UEFI 环境已经准备完毕。

BDS 和 TSL 是操作系统加载器作为 UEFI 应用程序运行的阶段。

操作系统加载器调用 ExitBootServices() 服务后进入 RT 阶段，RT 阶段包括操作系统加载器后期和操作系统运行期。

当系统硬件或操作系统出现严重错误不能继续正常运行时，固件会尝试修复错误，这时系统进入 AL 期。但 PI 规范和 UEFI 规范都没有规定 AL 期的行为。“?”号表示其行为由系统供应商自行定义。

㊀ www.uefi.org。

㊁ http://sourceforge.net/apps/mediawiki/tianocore/index.php?title=File:PI_Boot_Phases.jpg。

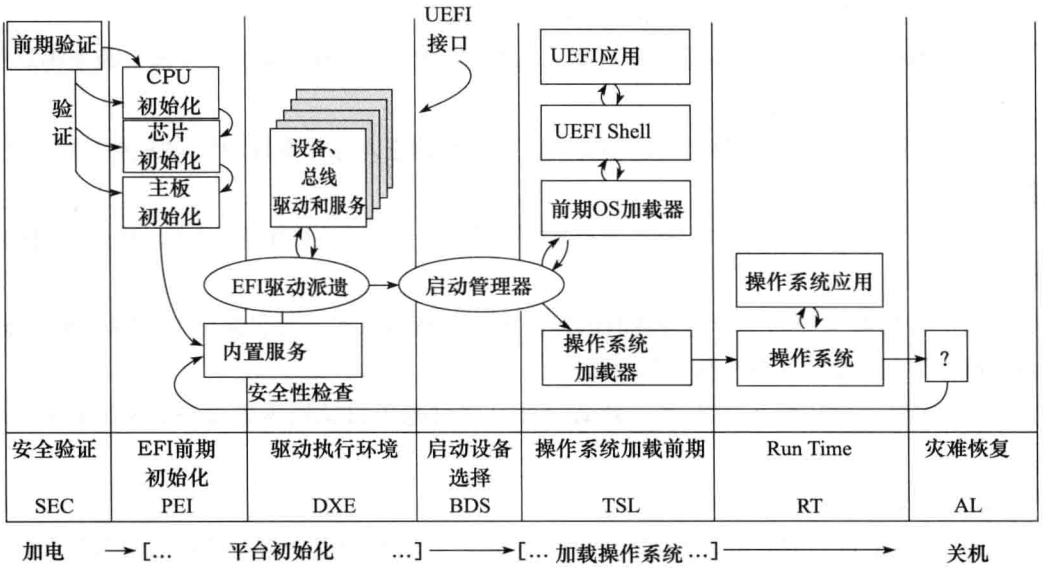


图 1-2 UEFI 系统的 7 个阶段

1. SEC 阶段

SEC（Security Phase）阶段是平台初始化的第一个阶段，计算机系统加电后进入这个阶段。

（1）SEC 阶段的功能

UEFI 系统开机或重启进入 SEC 阶段，从功能上说，它执行以下 4 种任务。

1）**接收并处理系统启动和重启信号：**系统加电信号、系统重启信号、系统运行过程中的严重异常信号。

2）**初始化临时存储区域：**系统运行在 SEC 阶段时，仅 CPU 和 CPU 内部资源被初始化，各种外部设备和内存都没有被初始化，因而系统需要一些临时 RAM 区域，用于代码和数据的存取，我们将之称为临时 RAM，以示与内存的区别。这些临时 RAM 只能位于 CPU 内部。最常用的临时 RAM 是 Cache，当 Cache 被配置为 no-eviction 模式时，可以作为内存使用，读命中时返回 Cache 中的数据，读缺失时不会向主存发出缺失事件；写命中时将数据写入 Cahce，写缺失时不会向主存发出缺失事件，这种技术称为 CAR（Cache As Ram）。

3）**作为可信系统的根：**作为取得对系统控制权的第一部分，SEC 阶段是整个可信系统的根。SEC 能被系统信任，以后的各个阶段才有被信任的基础。通常，SEC 在将控制权转移给 PEI 之前，可以验证 PEI。

4）**传递系统参数给下一阶段（即 PEI）：**SEC 阶段的一切工作都是为 PEI 阶段做准备，最终 SEC 要把控制权转交给 PEI，同时要将现阶段的成果汇报给 PEI。汇报的手段就是将如

下信息作为参数传递给 PEI 的入口函数。

- 系统当前状态，PEI 可以根据这些状态判断系统的健康状况。
- 可启动固件（Boot Firmware Volume）的地址和大小。
- 临时 RAM 区域的地址和大小。
- 栈的地址和大小。

（2）SEC 阶段执行流程

上面介绍了 SEC 的功能，下面再来看看 SEC 的执行流程，如图 1-3 所示。

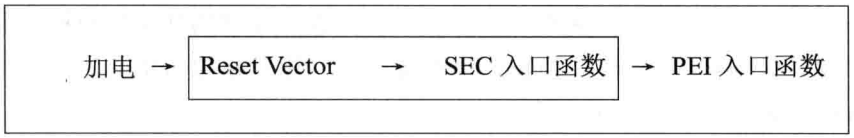


图 1-3　SEC 阶段执行流程

以临时 RAM 初始化为界，SEC 的执行又分为两大部分：临时 RAM 生效之前称为 Reset Vector 阶段，临时 RAM 生效后调用 SEC 入口函数从而进入 SEC 功能区。

其中 Reset Vector 的执行流程如下。

1）进入固件入口。

2）从实模式转换到 32 位平坦模式（包含模式）。

3）定位固件中的 BFV（Boot Firmware Volume）。

4）定位 BFV 中的 SEC 映像。

5）若是 64 位系统，从 32 位模式转换到 64 位模式。

6）调用 SEC 入口函数。

下面的代码描述了从固件入口 Reset Vector 到 SEC 入口函数的执行过程：

```
; file: UefiCpuPkg/ResetVector/Vtf0/Ia16/ResetVectorVtf0.asm
resetVector:
jmp     short EarlyBspInitReal16
```

⇩

```
; file: UefiCpuPkg/ResetVector/Vtf0/Ia16/Init16.asm EarlyBspInitReal16:
mov     di, 'BP'
jmp     short Main16
```

⇩

```
; file: UefiCpuPkg/ResetVector/Vtf0/Main.asm
Main16:
OneTimeCall EarlyInit16
OneTimeCall TransitionFromReal16To32BitFlat;    从实模式转换到 32 位平坦模式
OneTimeCall Flat32SearchForBfvBase;             定位固件中的 BFV
```

```
OneTimeCall Flat32SearchForSecEntryPoint;          定位 BFV 中的 SEC 映像
;esi 寄存器存放了 SEC 的入口地址，ebp 寄存器存放了 BFV 起始地址
%ifdef ARCH_IA32
mov eax, esp
jmp esi; 跳到 SEC 入口
%else
OneTimeCall Transition32FlatTo64Flat;              从 32 位模式转换到 64 位模式
...
jmp rsi;                                           跳到 SEC 入口
%endif
```

在 Reset Vector 部分，因为系统还没有 RAM，因而不能使用基于栈的程序设计，所有的函数调用都使用 jmp 指令模拟。OneTimeCall 是宏，用于模拟 call 指令。例如，宏调用 OneTimeCall EarlyInit16，如图 1-4 所示。

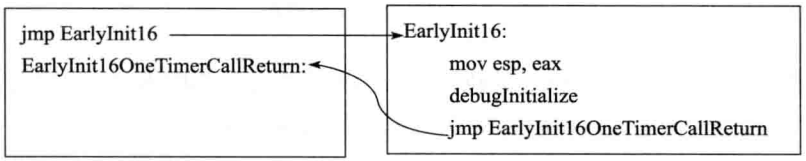


图 1-4 OneTimeCall 示例

进入 SEC 功能区后，首先利用 CAR 技术初始化栈，初始化 IDT，初始化 EFI_SEC_PEI_HAND_OFF，将控制权转交给 PEI，并将 EFI_SEC_PEI_HAND_OFF 传递给 PEI。

不同的硬件平台，SEC 代码会有不同的实现方式，但大致执行过程相似。下面以 OVMF（具体介绍参见第 2 章）为例，介绍 SEC 功能区的执行过程。

```
# file: OvmfPkg/Sec/X64/SecEntry.S
ASM_PFX(_ModuleEntryPoint):
# 临时 RAM 已经初始化，设置栈地址。PcdOvmfSecPeiTempRamBase 和 PcdOvmfSecPeiTempRamSize
    在 OvmfPkgIa32X64.fdf 中定义
# 分别为 0x010000 和 0x008000
.set SEC_TOP_OF_STACK, FixedPcdGet32 (PcdOvmfSecPeiTempRamBase) +
FixedPcdGet32 (PcdOvmfSecPeiTempRamSize)
movq $SEC_TOP_OF_STACK, %rsp
movq %rbp, %rcx#rcx: BFV 首地址，rbp 为传入参数
movq %rsp, %rdx#rdx: 栈起始地址
subq $0x20, %rsp
call ASM_PFX(SecCoreStartupWithStack)# 此时栈已可用，故可使用 call 指令
```

```
# file: OvmfPkg/Sec/X64/SecEntry.S
VOID EFI_API SecCoreStartupWithStack(
IN EFI_FIRMWARE_VOLUME_HEADER *BootFv,
```

```
IN VOID *TopOfCurrentStack
){
  EFI_SEC_PEI_HAND_OFF SecCoreData;
  //初始化浮点寄存器
  //初始化 IDT
  //初始化 SecCoreData，将临时 RAM 地址、栈地址、BFV 地址赋值给 SecCoreData
  SecStartupPhase2(&SecCoreData);
}
```

```
# file: OvmfPkg/Sec/X64/SecEntry.S
VOID EFI_API SecStartupPhase2(IN VOID *Context)
{
  EFI_SEC_PEI_HAND_OFF          SecCoreData;
  EFI_PEI_CORE_ENTRY_POINT      PeiCoreEntryPoint;
  SecCoreData = (EFI_SEC_PEI_HAND_OFF *) Context;
  //从 BFV 中找出 PEI 的入口函数
  FindAndReportEntryPoints (&SecCoreData->BootFirmwareVolumeBase,
  &PeiCoreEntryPoint);
  //调用 PEI 入口函数，SecCoreData 包含了临时 RAM、栈、BFV 地址和大小，
  //mPrivateDispatchTable 包含了 EFI_PEI_TEMPORARY_RAM_SUPPORT_PPI
  (*PeiCoreEntryPoint) (SecCoreData,
  (EFI_PEI_PPI_DESCRIPTOR *)&mPrivateDispatchTable);
}
```

2. PEI 阶段

PEI（Pre-EFI Initialization）阶段资源仍然十分有限，内存到了 PEI 后期才被初始化，其主要功能是为 DXE 准备执行环境，将需要传递到 DXE 的信息组成 HOB（Handoff Block）列表，最终将控制权转交到 DXE 手中。PEI 执行流程如图 1-5 所示。

从功能上讲，PEI 可分为以下两部分。

- PEI 内核（PEI Foundation）：负责 PEI 基础服务和流程。
- PEIM（PEI Module）派遣器：主要功能是找出系统中的所有 PEIM，并根据 PEIM 之间的依赖关系按顺序执行 PEIM。PEI 阶段对系统的初始化主要是由 PEIM 完成的。

每个 PEIM 是一个独立的模块，模块的入口函数类型定义如下所示：

```
typedef EFI_STATUS(EFIAPI *EFI_PEIM_ENTRY_POINT2)(
  IN EFI_PEI_FILE_HANDLE FileHandle,IN CONST EFI_PEI_SERVICES  **PeiServices
);
```

通过 PeiServices，PEIM 可以使用 PEI 阶段提供的系统服务，通过这些系统服务，PEIM 可以访问 PEI 内核。PEIM 之间的通信通过 PPI（PEIM-to-PEIM Interfaces）完成。

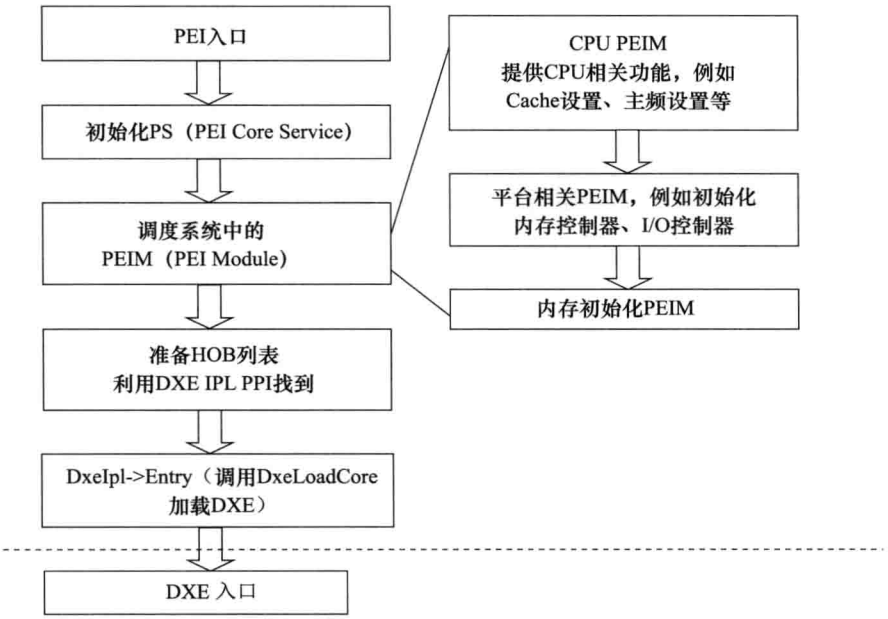


图 1-5　PEI 执行流程

PPI 与 DXE 阶段的 Protocol 类似，每个 PPI 是一个结构体，包含了函数指针和变量，例如：

```
struct _EFI_PEI_DECOMPRESS_PPI {
  EFI_PEI_DECOMPRESS_DECOMPRESS Decompress;
}
extern EFI_GUID   gEfiPeiDecompressPpiGuid;
```

每个 PPI 都有一个 GUID。根据 GUID，通过 PeiServices 的 LocatePpi 服务可以得到 GUID 对应的 PPI 实例。

UEFI 的一个重要特点是其模块化的设计。模块载入内存后生成 Image。Image 的入口函数为 _ModuleEntryPoint。PEI 也是一个模块，PEI Image 的入口函数 _ModuleEntryPoint，位于 MdePkg/Library/PeimEntryPoint/PeimEntryPoint.c。_ModuleEntryPoint 最终调用 PEI 模块的入口函数 PeiCore，位于 MdeModulePkg/Core/Pei/PeiMain/PeiMain.c。进入 PeiCore 后，首先根据从 SEC 阶段传入的信息设置 Pei Core Services，然后调用 PeiDispatcher 执行系统中的 PEIM，当内存初始化后，系统会发生栈切换并重新进入 PeiCore。重新进入 PeiCore 后使用的内存为我们所熟悉的内存。所有 PEIM 都执行完毕后，调用 PeiServices 的 LocatePpi 服务得到 DXE IPL PPI，并调用 DXE IPL PPI 的 Entry 服务，这个 Entry 服务实际上是 DxeLoadCore，它找出 DXE Image 的入口函数，执行 DXE Image 的入口函数并将 HOB 列表传递给 DXE。

3. DXE 阶段

DXE（Driver Execution Environment）阶段执行大部分系统初始化工作，进入此阶段时，内存已经可以被完全使用，因而此阶段可以进行大量的复杂工作。从程序设计的角度讲，DXE 阶段与 PEI 阶段相似，执行流程如图 1-6 所示。

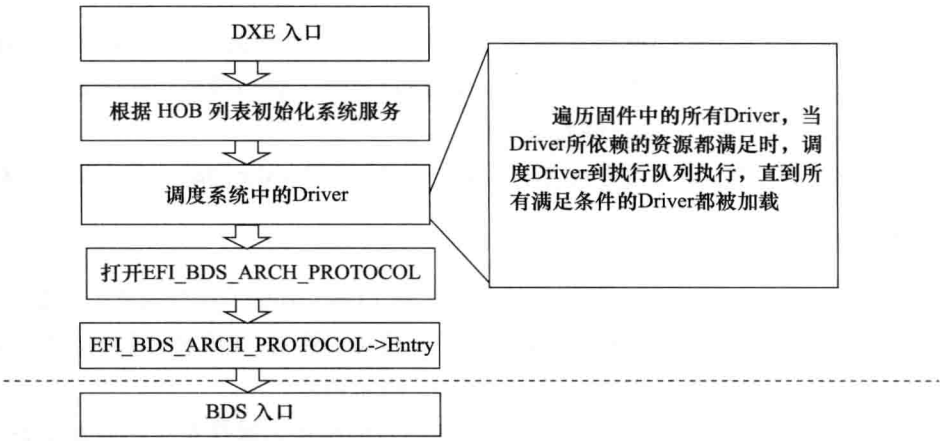


图 1-6　DXE 执行流程

与 PEI 类似，从功能上讲，DXE 可分为以下两部分。

- DXE 内核：负责 DXE 基础服务和执行流程。
- DXE 派遣器：负责调度执行 DXE 驱动，初始化系统设备。

DXE 提供的基础服务包括系统表、启动服务、Run Time Services。

每个 DXE 驱动是一个独立的模块，模块入口函数类型定义为：

```
typedef EFI_STATUS(EFIAPI *EFI_IMAGE_ENTRY_POINT)(
  IN  EFI_HANDLE ImageHandle,
  IN  EFI_SYSTEM_TABLE *SystemTable
);
```

DXE 驱动之间通过 Protocol 通信。Protocol 是一种特殊的结构体，每个 Protocol 对应一个 GUID，利用系统 BootServices 的 OpenProtocol，并根据 GUID 来打开对应的 Protocol，进而使用这个 Protocol 提供的服务。

当所有的 Driver 都执行完毕后，系统完成初始化，DXE 通过 EFI_BDS_ARCH_PROTOCOL 找到 BDS 并调用 BDS 的入口函数，从而进入 BDS 阶段。从本质上讲，BDS 是一种特殊的 DXE 阶段的应用程序。

4. BDS 阶段

BDS（Boot Device Selection）的主要功能是执行启动策略，其主要功能包括：

- 初始化控制台设备。
- 加载必要的设备驱动。
- 根据系统设置加载和执行启动项。

如果加载启动项失败，系统将重新执行 DXE dispatcher 以加载更多的驱动，然后重新尝试加载启动项。

BDS 策略通过全局 NVRAM 变量配置。这些变量可以通过运行时服务的 GetVariable() 读取，通过 SetVariable() 设置。例如，变量 BootOrder 定义了启动顺序，变量 Boot#### 定义了各个启动项（#### 为 4 个十六进制大写符号）。

用户选中某个启动项（或系统进入默认的启动项）后，OS Loader 启动，系统进入 TSL 阶段。

5. TSL 阶段

TSL（Transient System Load）是操作系统加载器（OS Loader）执行的第一阶段，在这一阶段 OS Loader 作为一个 UEFI 应用程序运行，系统资源仍然由 UEFI 内核控制。当启动服务的 ExitBootServices() 服务被调用后，系统进入 Run Time 阶段。

TSL 阶段之所以称为临时系统，在于它存在的目的就是为操作系统加载器准备执行环境。虽然是临时系统，但其功能已经很强大，已经具备了操作系统的雏形，UEFI Shell 是这个临时系统的人机交互界面。正常情况下，系统不会进入 UEFI Shell，而是直接执行操作系统加载器，只有在用户干预下或操作系统加载器遇到严重错误时才会进入 UEFI Shell。

6. RT 阶段

系统进入 RT（Run Time）阶段后，系统的控制权从 UEFI 内核转交到 OS Loader 手中，UEFI 占用的各种资源被回收到 OS Loader，仅有 UEFI 运行时服务保留给 OS Loader 和 OS 使用。随着 OS Loader 的执行，OS 最终取得对系统的控制权。

7. AL 阶段

在 RT 阶段，如果系统（硬件或软件）遇到灾难性错误，系统固件需要提供错误处理和灾难恢复机制，这种机制运行在 AL（After Life）阶段。UEFI 和 UEFI PI 标准都没有定义此阶段的行为和规范[⊖]。

1.3 本章小结

相对 BIOS，UEFI 有更好的可编程性，强大的可扩展性，出色的安全性，并且其设计更

⊖ http://sourceforge.net/apps/mediawiki/tianocore/index.php?title=PI_Boot_Flow。

能适应 64 位平台。这些优势使得 UEFI 可以迅速取代 BIOS。

UEFI 定义了操作系统和平台固件之间的接口。UEFI 接口可分为以下两个部分。

1）启动服务：启动服务的主要服务对象是操作系统加载器以及其他 UEFI 应用程序和 UEFI 驱动。操作系统加载器通过启动服务逐步取得对整个计算机系统资源的控制。当加载器完全控制计算机软硬件资源后，系统结束启动服务，进入运行时。启动服务主要包括：事件服务、内存管理、Protocol 管理、Protocol 使用类服务、驱动管理、Image 管理以及 ExitBootServices 服务。

2）运行时服务：运行时服务的主要服务对象是操作系统、操作系统加载器以及 UEFI 应用和 UEFI 驱动。运行时服务主要包括：时间服务、读写 UEFI 系统变量的服务、虚拟内存服务、重启系统的服务。

基于 UEFI 的计算机系统，从启动到关机可以分为 7 个阶段，本书分别对这 7 个阶段的功能和执行流程进行了简单介绍。启动服务和运行时服务只有在系统进入 DXE 阶段后才生成。本书重点讲述的 UEFI 应用程序和 UEFI 驱动就是运行在这一阶段。

通过本章的学习，读者应该对 UEFI 有了初步的认识。下一章主要介绍 UEFI 开发环境的搭建。

Chapter 2 第 2 章

UEFI 开发环境搭建

通过前面的学习，我们已经知道 UEFI 是一种标准，它没有给出具体的实现。软件厂商可以根据 UEFI 标准开发自己的 UEFI 实现，其中常用的开源实现是 EDK2。EDK2 是遵循 UEFI 标准和 PI 标准的跨平台固件开发环境。UEFI 的目标是完全取代 BIOS，因而它要能完全支持所有类型的 CPU[⊖]，并让所有的硬件厂商接受这种变化。来自不同厂商的开发者使用各种不同的开发环境开发自己的产品。为了让这些不同的开发者愉快地接受 EDK2 来开发自己平台上的 UEFI 固件或应用，EDK2 对每种平台都提供了对应的开发工具。EDK2 支持在多种操作系统下的开发，例如 Windows、Linux、Darwin、UNIX 等，并支持跨平台编译，如在 Windows 开发环境下可以编译出 Arm 平台上的 UEFI 应用程序。下面我们分别介绍在 Windows 和 Linux 下如何使用 EDK2 进行开发。

2.1 配置 Windows 开发环境

EDK2 目前支持 Windows 7、Windows 8、Windows 8.1。开发 UEFI 应用和驱动之前要建立开发环境，分为如下几步。

1）首先需安装 EDK2 依赖的开发工具：Windows SDK、C 编译器、IASL 编译器，然后下载 EDK2 源码，EDK2 源码包里包含了开发所需的源码和工具。

2）配置 EDK2：主要是设置开发工具的路径。然后就可以通过 EDK2 提供的源码和工

⊖ 目前 EDK2 支持 x86（32 位和 64 位）CPU、安腾 CPU 和 Arm CPU。

具开发 UEFI 应用和驱动。

3）编译 UEFI 模拟器和 UEFI 工程。

4）运行 UEFI 模拟器。

下面详细介绍动手开发之前必须进行的这几个步骤。

2.1.1　安装所需开发工具

首先需要安装 EDK2 所依赖的开发工具以及 EDK2 本身。以下是主要的安装步骤。

1）安装 Windows SDK。Windows SDK 可从如下地址下载：

http://www.microsoft.com/download/en/details.aspx?displaylang=en&id=24826

2）安装 C 编译器。推荐安装微软公司的 Visual Studio 编译器或英特尔公司的 ICC 编译器，也可以安装 Cygwin 及其 gcc 编译器。

3）安装 IASL 编译器（https://www.acpica.org/downloads/binary_tools）。

4）IASL 用于编译 .asl 文件。.asl 是高级配置与电源接口（Advanced Configuration and Power Interface）源文件。

5）下载 EDK2 开发包：可以从 TianoCore 官方网站下载 EDK2 发行版（https://sourceforge.net/projects/edk2/files/UDK2014_Releases/UDK2014/UDK2014.Complete.MyWorkSpace.zip/download），也可以用 subversion 工具或命令行获得最新源码。下面是两个简单的示例。

1）用 subversion 命令行获得 EDK2 源码：

```
svn co https://svn.code.sf.net/p/edk2/code/trunk/edk2edk2
```

2）或者使用 TortoiseSvn 下载源码，如图 2-1 所示。

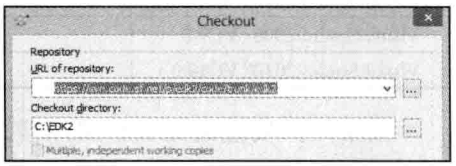


图 2-1　用 TortoiseSvn 下载 EDK2 源码

2.1.2　配置 EDK2 开发环境

下面讲解如何配置 EDK2 开发环境。

1）首先进入 EDK2 目录并运行 edksetup.bat，此步骤用于建立 Conf 目录下的 target.txt、tools_def.txt 等文件。

```
C:\>EDK2Edksetup.bat
```

2）编辑 Conf\target.txt。

根据用户的编译器环境修改编译工具 TOOL_CHAIN_TAG。此处以 x86_64 平台的 32 位的 Visual Studio 2008 为例，需设置 TOOL_CHAIN_TAG=VS2008x86。

图 2-2 是设置了 TOOL_CHAIN_TAG 为 VS2008x86 的 target.txt 文件。

64位操作系统上的Conf\target.txt

```
ACTIVE_PLATFORM              = Nt32Pkg/Nt32Pkg.dsc
TARGET                       = DEBUG
TARGET_ARCH                  = IA32
TOOL_CHAIN_CONF              = Conf/tools_def.txt
TOOL_CHAIN_TAG               = VS2008x86
MAX_CONCURRENT_THREAD_NUMBER = 1
BUILD_RULE_CONF = Conf/build_rule.txt
```

图 2-2 Conf\target.txt 配置文件

注意 TOOL_CHAIN_TAG 的设置要根据 Visual Studio 的安装路径来确定。

IA32 平台下 TOOL_CHAIN_TAG 的设置见表 2-1。

表 2-1 IA32 平台下的 TOOL_CHAIN_TAG

Visual Studio 版本	TOOL_CHAIN_TAG 的值
Visual Studio 2003（VC7）	VS2003
Visual Studio 2005（VC8）	VS2005
Visual Studio 2008（VC9.0）	VS2008
Visual Studio 2010（VC10.0）	VS2010

AMD64 平台下的 TOOL_CHAIN_TAG 的设置见表 2-2。

表 2-2 AMD64 平台下的 TOOL_CHAIN_TAG

Visual Studio 版本	TOOL_CHAIN_TAG 的值（64 位 VS）	TOOL_CHAIN_TAG 的值（32 位 VS）
Visual Studio 2003（VC7）	VS2003	VS2003x86
Visual Studio 2005（VC8）	VS2005	VS2005x86
Visual Studio 2008（VC9.0）	VS2008	VS2008x86
Visual Studio 2010（VC10.0）	VS2010	VS2010x86

如果用户安装的是 ICC 编译器，则需要将 TOOL_CHAIN_TAG 设置为 ICC 或 ICC11，具体配置需根据 ICC 版本号确定。例如，安装了 ICC11 编译器，则做如下设置：

```
TOOL_CHAIN_TAG = ICC11
```

如果用户安装的是 Cygwin，则需要将 TOOL_CHAIN_TAG 设置为 CYGWIN：

```
TOOL_CHAIN_TAG = CYGWIN
```

3）检查 Conf\tools_def.txt，确保编译器路径正确。

tools_def.txt 工具为 EDK2 自动生成的文件，里面预定义了几种常用的编译器。

例如，我们在步骤 2 中设置了 TOOL_CHAIN_TAG 为 VS2008x86，在 tools_def.txt 查找 VS2008x86_BIN，看其路径是否正确，如果路径不正确，需设置为 cl.exe 的正确路径。

图 2-3 为 tools_def.txt 文件中关于 VS2008x86 的部分，其中 DEFINE VS2008x86_BIN 定义了编译器 cl.exe 的路径。当编译 32 位 EDK 程序时，build 工具会使用 VS2008x86_BIN 下的 cl 编译器；当编译 x86_64 程序时，会使用 VS2008x86_BINX64 下的 cl 编译器。

DEFINE WIN_ASL_BIN_DIR 定义了 IASL 编译器的路径。

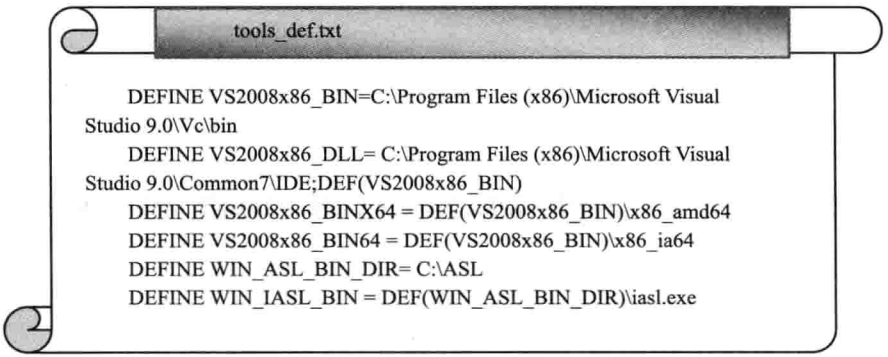


图 2-3　conf\tools_def.txt 文件 VS2008x86 部分，该文件定义了编译器的路径

2.1.3　编译 UEFI 模拟器和 UEFI 工程

下面讲解如何利用 EDK2 提供的工具链进行编译，并对其中的重要参数进行讲解。

编译 UEFI 代码是在 CMD 命令行中通过运行 EDK2 工具链命令完成的，分两种情况：一是编译 Nt32Pkg 工程，二是编译非 Nt32Pkg 的 UEFI 工程。

1. 编译 UEFI 模拟器，即 Nt32Pkg

首先需要运行 edksetup.bat--nt32 以设置 EDK2 环境变量，如下所示：

```
Setting environment for using Microsoft Visual Studio 2008 x86 tools.
C:\Program Files(x86)\Microsoft Visual Studio 9.0\VC>cd c:\EDK2
C:\EDK2>edksetup.bat --nt32
```

设置好环境变量后，就可以使用 EDK2 的工具链编译 UEFI 模拟器了。编译 UEFI 模拟器的命令非常简单，在命令行执行 build 命令即可，如下所示：

```
C:\EDK2> build
```

build 命令相当于 Visual Studio 的 nmake 命令，它分析 UEFI 的工程文件，根据分析结果

自动执行相应编译和连接命令。

不带参数的 build 命令等同于 build -a IA32 -p Nt32Pkg\Nt32Pkg.dsc，这是因为 build 命令使用了 conf\target.txt 中定义的默认参数 TARGET_ARCH（对应 -a 参数）和 ACTIVE_PLATFORM（对应 -p 参数）。

2. 编译非 Nt32Pkg 工程

首先设置环境变量，打开 Visual Studio 2008 command prompt，进入 EDK2 目录并运行 edksetup.bat，如下所示：

```
Setting environment for using Microsoft Visual Studio 2008 x86 tools.
C:\Program Files(x86)\Microsoft Visual Studio 9.0\VC>cd c:\EDK2
C:\>EDK2>edksetup.bat
```

设置好环境变量后，就可以使用 EDK2 提供的 build 工具编译 UEFI 代码了。例如，要编译 MdePkg：

```
C:\EDK2> build -a X64 -p MdePkg\MdePkg.dsc
```

3. build 命令

build 命令是编译 UEFI 工程常用的命令。它有三个重要参数：-a、-p 和 -m。

- -a 用来选择目标平台。可供选择的选项有 IA32（32 位 x86 CPU）、X64（64 位 x86_64 CPU）、IPF（ Itanium Processor Family）、ARM 和 EBC（EFI byte code）；默认的参数在 Conf/target.txt 中设置。
- -p 用来指定要编译的 package 或 Platform。-p 的参数是这个 package 或 Platform 的 .dsc 文件。默认的参数在 Conf/target.txt 中设置。
- -m 用来指定要编译的模块。如果不指定 -m 选项，build 将编译 .dsc 文件指定的所有模块。

例如，编译 32 位的 Shell 后，Shell.efi 存放于 edk2\Build\ShellPkg\DEBUG_VS2008x86\IA32\。

```
C:\EDK2> build -a IA32 -p ShellPkg/ShellPkg.dsc -m ShellPkg/Application/Shell/
    Shell.inf
```

编译 64 位的 Shell 后，Shell.efi 存放于 edk2\Build\ShellPkg\DEBUG_VS2008x86\X64\。

```
C:\EDK2> build -a X64 -p ShellPkg/ShellPkg.dsc -m ShellPkg/Application/Shell/
    Shell.inf
```

表 2-3 列出了 build 命令的常用参数及用法。

表 2-3　build 命令的常用参数

build 命令参数	参数用法
-a ARCH	选择目标平台，ARCH 可以是 IA32、X64、IPF、ARM 或 EBC，该选项将会取代 Conf\target.txt 文件中的 TARGET_ARCH
-DMACROS	定义宏，例如 -D NETWORK_ENABLE
-h	显示帮助信息
-j LOGFILE	将编译信息输出到文件
-b TARGET	选择编译成 DEBUG 还是 RELEASE 例如： -b DEBUG -b RELEASE
-t TOOLCHAIN	选择 tools_def.txt 中定义的编译工具，例如要使用 Visual Studio 2010： -t vs2010
-n ThreadNumber	编译器使用的线程数量
-p PlatformFile	通过指定 .dsc 文件指定要编译的 Package，该选项将会取代 Conf\target.txt 文件中的 ACTIVE_PLATFORM。例如，要编译 AppPkg，可以使用如下选项： -p AppPkg\AppPkg.dsc
-m ModuleFile	指定要编译的模块，build 工具将只编译此模块
-q	编译过程中只显示严重错误信息
-s	使用沉默模式执行 make 或 nmake
-u	跳过 AutoGen 这一步
-c	文件名不区分大小写

2.1.4　运行模拟器

使用以下命令运行 UEFI 模拟器：

```
C:\EDK2> build run
```

因为在 target.txt 中已经设置了 TARGET_ARCH 与 ACTIVE_PALTFORM，所以 build run 与 build -a IA32 -p Nt32Pkg\Nt32Pkg.dsc run 命令等同，可以用来运行 UEFI 模拟器。或者直接运行 Build\NT32\DEBUG_VS2008x86\IA32 目录下的 SecMain.exe。

通常模拟器最终会进入 UEFI Shell，有关 UEFI Shell 内容我们会在第 16 章详细讲述。EDK2 默认将目录 C:\edk2\Build\NT32\DEBUG_VS2008x86\IA32\ 映射为文件系统 FSNT0，在 Shell 中执行命令 FSNT0:，Shell 将会打开 FSNT0 分区并将当前目录切换到 FSNT0 分区的根目录，如下列代码所示：

```
Shell>FSNT0:
FSNT0:\>
```

执行“?”命令将会列出所有 Shell 支持的命令，如图 2-4 所示。

```
FSNT0:\>?
alias           - Displays, Creates, or deletes UEFI Shell aliases.
attrib          - Displays or changes the attributes of files or directories.
...
```

图 2-4　Shell 中的“?”命令

图 2-5 显示了 UEFI 模拟器启动时的界面。图 2-6 显示了 UEFI 模拟器进入 Shell 时的界面。

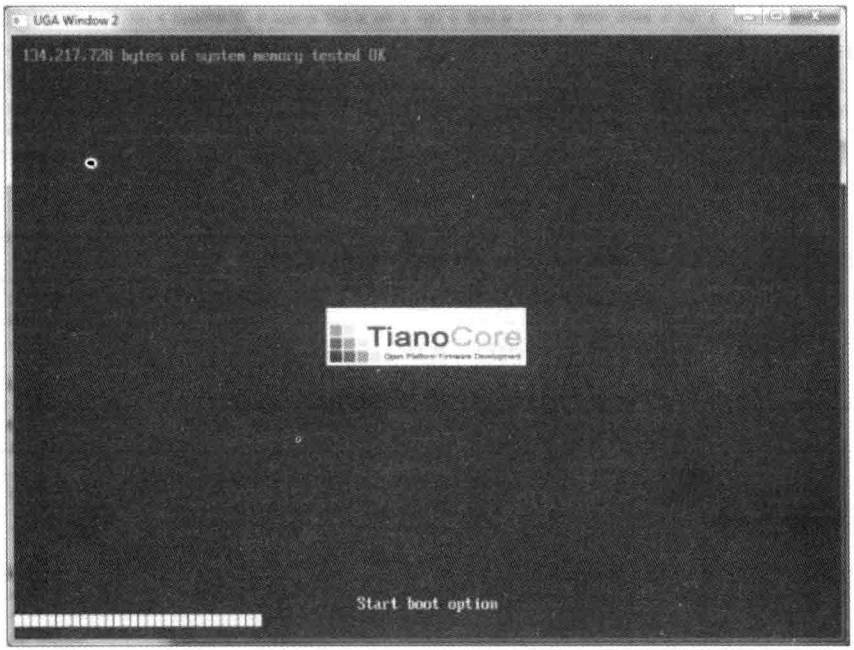


图 2-5　NT32 UEFI 模拟器启动界面图

图 2-5 中第二行显示了 Current running mode 1.1.2，这说明 UEFI Shell 是 EDK2 开发包中预先编译的 Shell。我们可以用如下方式进入刚才编译的 Shell。

1）将 Shell.efi 文件从 edk2\Build\ShellPkg\DEBUG_VS2008x86\IA32\ 复制到 Build\NT32\DEBUG_VS2008x86\IA32 目录。

2）在模拟器窗口的 Shell 中执行以下命令：

```
Shell>FSNT0:
FSNT0:\>shell
```

执行完毕后将会进入新的 UEFI Shell，新 Shell 如图 2-7 所示。

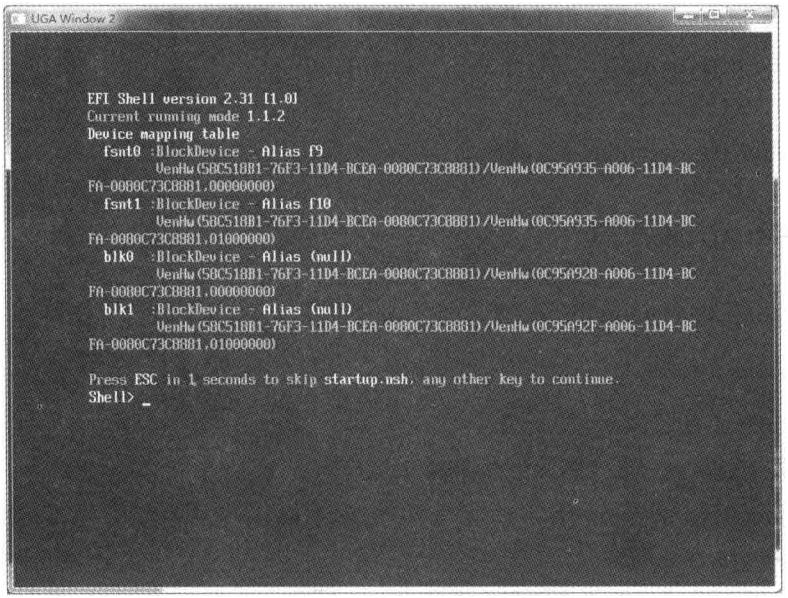


图 2-6　NT32 UEFI 模拟器 Shell 界面

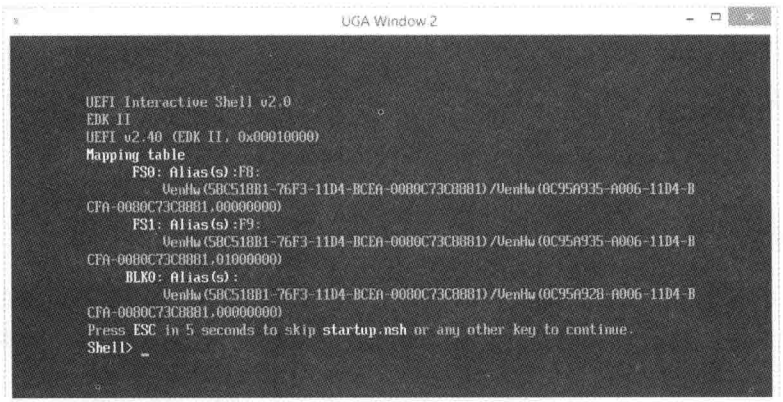


图 2-7　UEFI 2.4 标准下的 UEFI Shell

2.2　配置 Linux 开发环境

与配置 Windows 开发环境相似，配置 Linux 开发环境也包括如下步骤。

1）安装 gcc 编译器，并下载 EDK2 源码。

2）配置 EDK2，包括编译 EDK2 工具链和设置编译器路径。在配置 Windows 开发环境时，我们没有编译 EDK2 工具链，因为 EDK2 源码包中包含了 Windows 下的 EDK2 工具链可执行文件。但是，由于 Linux 发行版众多，因此，EDK2 源码包中没有包含 EDK2 工具链

可执行文件。然后就可以通过 EDK2 提供的源码和工具开发 UEFI 应用和驱动了。

3）编译 UEFI 模拟器和 UEFI 工程。

4）运行 UEFI 模拟器。

下面详细介绍配置 Linux 开发环境的这几个步骤。

2.2.1 安装所需开发工具

1）安装 gcc：在 Ubuntu 下可以使用如下命令安装 gcc。

```
EDK2:$apt-get install gcc
```

2）下载 EDK2 开发包。

① 可以从 TianoCore 官方网站下载 EDK2 发行版：

https://sourceforge.net/projects/edk2/files/UDK2014_Releases/UDK2014/UDK2014.Complete.MyWorkSpace.zip/download

② 也可以用代码管理工具获得最新源码。

3）用 subversion 命令行获得 EDK2 源码：

```
EDK2:$svn co https://svn.code.sf.net/p/edk2/code/trunk/edk2edk2
```

或用 git 命令行下载源码：

```
EDK2:$git clone https://github.com/tianocore/edk2
```

2.2.2 配置 EDK2 开发环境

1）编辑 Conf/target.txt。根据用户的编译器环境修改编译工具 TOOL_CHAIN_TAG。此处以 x86_32 Ubuntu 的 gcc 4.4 为例，需设置 TOOL_CHAIN_TAG=GCC44，如图 2-8 所示。

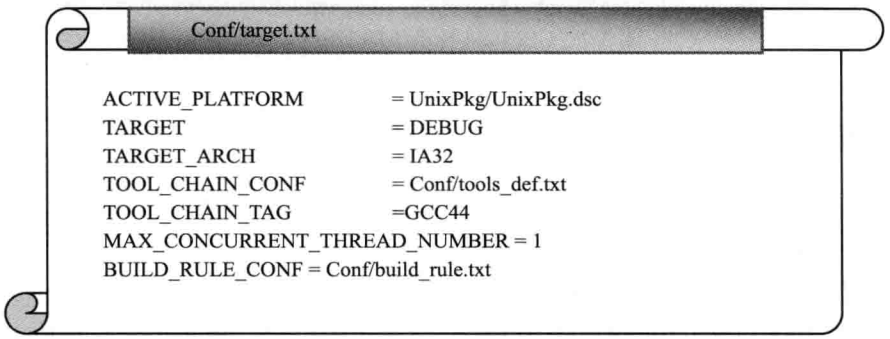


图 2-8 Conf/target.txt 文件，操作系统为 x86_32 Ubuntu，编译器为 gcc 4.4

2）检查 Conf\tools_def.txt（见图 2-9），确保编译器路径正确。

Conf/tools_def.txt

```
DEFINE GCC44_IA32_PREFIX     = /usr/bin/
DEFINE GCC44_X64_PREFIX      = /usr/bin/

DEFINE GCC45_IA32_PREFIX     = /usr/bin/
DEFINE GCC45_X64_PREFIX      = /usr/bin/

DEFINE GCC46_IA32_PREFIX     = /usr/bin/
```

图 2-9　Conf\tools_def.txt 文件 GCC44 相关部分，该文件定义了编译器的路径

3）编译 EDK2 工具链。

Linux 环境里 EDK2 工具链位于 BaseTools/BinWrappers/PosixLike/ 目录下。EDK2 工具链包括 build、GenFv、GenFw 等工具。如果该目录下没有这些工具，就要编译 BaseTools 来获得这些工具。

要编译 BaseTools，首先需保证系统中已经安装了 make、gcc 等编译工具。下面的命令用于安装编译 BaseTools 所需的编译工具：

```
EDK2$sudo apt-get install build-essential uuid-dev
```

然后进入 BaseTools 目录并使用 make 命令编译，如下所示：

```
EDK2$ cd BaseTools
EDK2/BaseTools$ make
```

编译后，BaseTools/BinWrappers/PosixLike/ 目录如图 2-10 所示。

```
EDK2$ls BaseTools/BinWrappers/PosixLike/
BootSectImage  EfiRom     GenFv            GenVtf             Split          VfrCompile
BPDG           GenCrc32   GenFw            GnuGenBootSector   TargetTool     VolInfo
build          GenDepex   GenPage          LzmaCompress       TianoCompress
Ecc            GenFds     GenPatchPcdTable LzmaF86Compress    Trim
EfiLdrImage    GenFfs     GenSec           PatchPcdValue      UPT
EDK2$
```

图 2-10　Linux 环境下 EDK2 工具链

2.2.3　编译 UEFI 模拟器和 UEFI 工程

UEFI 开发环境已经配置完毕，下面就可以编译 UEFI 模拟器和 UEFI 应用程序了。

1）首先设置环境变量。打开命令行工具，并进入 EDK2 目录，然后执行 source edksetup.sh，如下所示：

```
EDK2$ source edksetup.sh
```

2）环境变量配置好后就可以编译 UFEI 模拟器和其他 UEFI 工程了。

① 编译 UEFI 模拟器。

Linux 下的模拟器是 UnixPkg，编译模拟器就是编译 UnixPkg。可以使用 build 命令或带 -p 参数的 build 命令编译 UnixPkg，如下所示：

```
EDK2$build
```

或

```
EDK2$build -p UnixPkg/UnixPkg.dsc
```

编译后，模拟器 SecMain 将输出到 EDK2/Build/Unix/DEBUG_GCC44/IA32 目录。

build 命令的用法在 2.1 节配置 Windows 开发环境时已经做了详细说明，Linux 环境下 build 命令与之完全相同，在此不再重复说明。

② 编译其他 UEFI 工程。

编译其他 UEFI 工程也是通过 build 命令完成的。下面是使用 build 命令编译 Shell.efi 的示例。

【示例 2-1】 用如下命令可以编译 32 位的 Shell。编译后，Shell.efi 存放于 Build/Unix/DEBUG_GCC44/IA32/ 目录下。

```
EDK2$build -a IA32 -p ShellPkg/ShellPkg.dsc -m ShellPkg/Application/Shell/
    Shell.inf
```

【示例 2-2】 用如下命令可以编译 64 位的 Shell。编译后，Shell.efi 存放于 Build/Unix/DEBUG_GCC44/X64/ 目录下。

```
EDK2$build -a X64 -p ShellPkg/ShellPkg.dsc -m ShellPkg/Application/Shell/Shell.inf
```

2.2.4 运行模拟器

有两种方式可以运行 UEFI 模拟器。

第一种方式是在 Linux 图形界面的命令行中运行 build run 命令。

```
EDK2$build run
```

第二种方式是执行 EDK2/Build/Unix/DEBUG_GCC44/IA32 目录下的 SecMain 命令，如下所示。

```
EDK2$Build/Unix/DEBUG_GCC44/IA32/SecMain
```

图 2-11 是 Linux 下的 UEFI 模拟器。

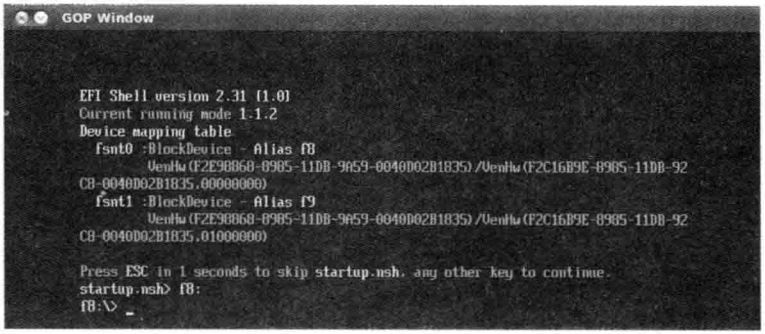


图 2-11　Linux 下的 UEFI 模拟器

2.3　OVMF 的制作和使用

OVMF(Open Virtual Machine Firmware，开放虚拟机固件）是用于虚拟机上的 UEFI 固件。在开发过程中，我们需要不断地测试所开发的产品。在模拟器中测试非常方便，但模拟器功能有限，并且模拟器只能测试 32 位程序。另外，在真实的 UEFI 环境中，测试又往往比较烦琐。在虚拟机中测试无疑是一种方便、快捷的方式，它既能较好地模拟真实环境，又可以做到快速、方便。EDK2 提供了制作虚拟机固件的方法，称为 OVMF。下面介绍如何编译和使用虚拟机固件。

1. 制作 OVMF

编译 OVMF 包，分如下两种情况。

1）用如下命令编译 64 位 OVMF 固件。

```
C:\EDK2>build -a X64 -p OvmfPkg\OvmfPkgX64.dsc
```

编译后的固件为 edk2\Build\OvmfX64\DEBUG_VS2008x86\FV\ 目录下的 OVMF.fd。

2）用如下命令编译 32 位 OVMF 固件。

```
C:\EDK2>build -a IA32 -p OvmfPkg\OvmfPkgIa32.dsc
```

编译后的固件为 edk2\Build\OvmfIa32\DEBUG_VS2008x86\FV\ 目录下的 OVMF.fd。

2. 在 QEMU 虚拟机中使用 OVMF

QEMU 是目前广泛使用的计算机仿真器和虚拟机。在 QEMU 虚拟机中，用户可以使用自定义的固件，利用这个特性我们可以测试 OVMF。首先，将存放于 edk2\Build\OvmfPkgIa32\DEBUG_VS2008x86\FV 下的文件 OVMF.fd 复制到 QEMU 的安装目录。然

后，在 QEMU 虚拟机中加载 OVMF，有两种方式：一种是使用 QEMU Manager（管理器）运行 QEMU 虚拟机，在图形界面中指定 OVMF 固件；另一种是使用 QEMU 命令行指定 OVMF 固件。

（1）使用 QEMU Manager

在 QEMU Manager 控制面板选择 Advanced → BIOS Filename，然后选择 OVMF.fd，如图 2-12 所示。

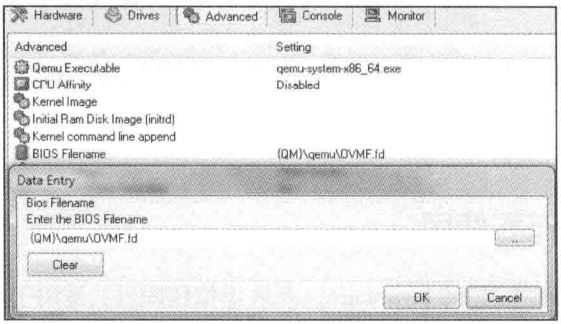


图 2-12　在 QEMU Manager 中选择固件 OVMF.fd

运行虚拟机，运行后将进入 UEFI Shell 界面，如图 2-13 所示。

（2）在 CMD 命令行运行 QEMU 命令

在 CMD 命令行运行如下 QEMU 命令。

```
C:\Program files\Qemu\>qemu-system-x86_64.exe-bios "OVMF.fd" -M "pc" -m 256 -cpu
    "qemu64" -vga cirrus -serial vc -parallel vc -name "UEFI" -boot order=dc
```

该命令运行后同样会进入图 2-13 所示的 UEFI Shell 界面。

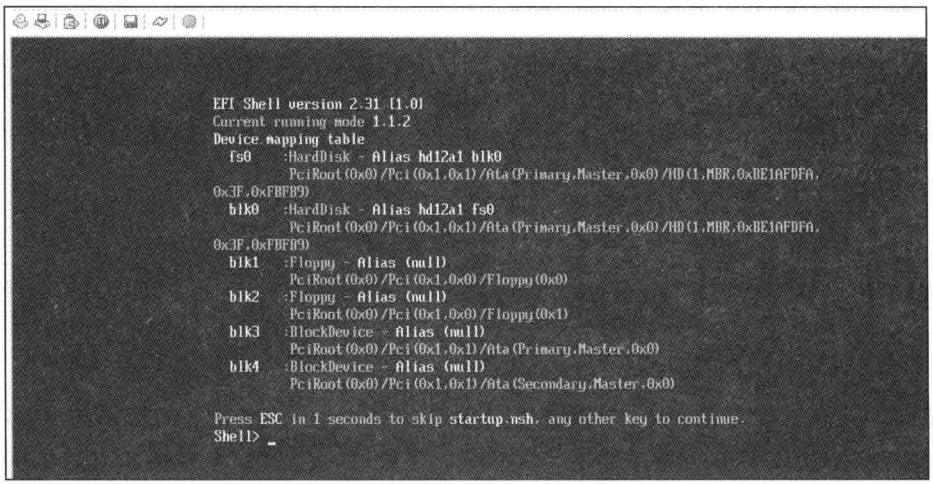


图 2-13　基于固件 OVMF.fd 的虚拟机

2.4　UEFI 的启动

虽然现在已经进入了 UEFI 时代，但是因为 UEFI 刚刚开始取代 BIOS，目前依然有很多运行着的 BIOS 系统。为了方便开发者从 BIOS 转移到 UEFI，EDK2 提供了 DUET，用于在 BIOS 系统上模拟 UEFI 执行环境。下面介绍一下 DUET。

1. DUET 简介

DUET（Developer's UEFI Emulation）是基于 Legacy BIOS 系统的 UEFI 模拟器，主要为 UEFI 开发者提供一个在传统 BIOS 系统上的 UEFI 运行环境。DUET 支持基于 MBR 的启动方式，从 MBR 启动后进入 UEFI 执行环境。

下面开始制作传统 BIOS 平台下的模拟 UEFI 启动盘。

如果目标平台是 Legacy BIOS，则需要在 U 盘中制作 MBR 和引导文件，可按如下步骤来进行。

1）编译 DuetPkg，在 CMD 命令行输入如下命令。

```
C:\EDK2>build -a IA32 -p DuetPkg\DuetPkgIa32.dsc
```

或者

```
C:\EDK2>build -a X64 -p DuetPkg\DuetPkgX64.dsc
```

2）通过如下命令生成引导文件。

```
C:\EDK2>  cd DuetPkg
C:\EDK2\DuetPkg>postbuild.bat Ia32
```

或者

```
C:\EDK2\DuetPkg>postbuild.bat X64
```

3）插入 U 盘，假设 J: 是 U 盘盘符，通过如下命令向 U 盘写入 MBR。

```
C:\EDK2\DuetPkg>  createbootdisk usb J: FAT32 IA32
```

或者

```
C:\EDK2\DuetPkg>  createbootdisk usb J: FAT32 X64
```

4）拔出并重新插入 U 盘，通过如下命令向 U 盘复制 UEFI 文件。

```
C:\EDK2\DuetPkg>   createbootdisk usb J: FAT32 IA32 step2
```

或者

```
C:\EDK2\DuetPkg>  createbootdisk usb J: FAT32 X64 step2
```

此命令向 U 盘根目录复制了 efildr20（该文件用于引导系统进入 UEFI 环境），并向 efi\boot 目录复制了引导文件 bootia32.efi（源文件位于 ShellBinPkg\UefiShell\Ia32\Shell.efi）或 bootx64.efi（源文件位于 ShellBinPkg\UefiShell\X64\Shell.efi）。

总结一下，要制作传统 BIOS 平台下的模拟 UEFI 启动盘（64 位），需在 CMD 命令行执行如下命令。

```
C:\EDK2>build -a X64 -p DuetPkg\DuetPkgX64.dsc
C:\EDK2>  cd DuetPkg
C:\EDK2\DuetPkg>postbuild.bat X64
C:\EDK2\DuetPkg>  createbootdisk usb J: FAT32 X64
```

拔出并重新插入 U 盘，继续执行如下命令。

```
C:\EDK2\DuetPkg>  createbootdisk usb J: FAT32 X64 step2
```

32 位模拟 UEFI 启动盘与 64 位相似，此处不再赘述。

有时需要使用最新的 Shell，此时要编译 ShellPkg，然后将编译好的 Shell.efi 复制到 U 盘 efi\boot 目录并重命名为引导文件（bootia32.efi 或 bootx64.efi）。

接下来就可以用 U 盘来引导进入 UEFI 世界了。

2. 制作 UEFI 平台下的 USB 启动盘

如果目标平台是 UEFI 平台，那么启动盘的制作就变得非常简单，可按如下步骤来进行。

1）格式化 U 盘为 FAT（FAT、FAT16 或 FAT32）格式。

2）在 U 盘上建立目录 efi\boot。

3）将 efi 的应用程序复制到 efi\boot 目录，并改名为 bootx64.efi 或者 bootia32.efi。

与 Legacy BIOS 需要 MBR 来引导操作系统不同，UEFI 的启动文件是 FAT 盘内 efi\boot 目录中的 bootx64.efi 或 bootia32.efi。

2.5 本章小结

本章主要是为开发 UEFI 应用和驱动准备开发环境，主要内容包括：

- Windows 和 Linux 下的 UEFI 开发环境的搭建（主要是 EDK2 的配置）。
- build 命令的用法。
- 制作和使用 OVMF 固件。
- 利用 DUET 制作传统 BIOS 系统上的 UEFI 启动盘。

第 3 章我们将开始介绍 UEFI 工程模块，主要包括 UEFI 应用程序工程、UEFI 驱动工程和 UEFI 类库工程。

第 3 章 Chapter 3

UEFI 工程模块文件

第 2 章讲述了如何编译 EDK2，下面开始介绍如何在 EDK2 环境下编程。编程之前首先要了解 EDK2 的两个概念：模块（Module）和包（Package）。

在 EDK2 根目录下，有很多以 *Pkg 命名的文件夹，每一个这样的文件夹称为一个 Package。当然，这种说法不准确，准确地说，“包”是一组模块及平台描述文件（.dsc 文件）、包声明文件（.dec 文件）组成的集合。模块是 UEFI 系统的一个特色。模块（可执行文件，即 .efi 文件）像插件一样可以动态地加载到 UEFI 内核中。对应到源文件，EDK2 中的每个工程模块由元数据文件（.inf 文件）和源文件（有些情况下也可以包含 .efi 文件）组成。

在 Linux 下编程，除了编写源代码之外，还要编写 Makefile 文件。在 Windows 下使用 VS（Visual Studio）时通常要建立工程文件和源文件。与之相似，在 EDK2 环境下，我们除了要编写源文件外，还要为工程编写元数据文件（.inf）。.inf 文件与 VS 的工程文件及 Linux 下的 Makefile 文件功能相似，用于自动编译源代码。包相当于 VS 中的项目，.dsc 文件则相当于 VS 项目的 .sln 文件；模块相当于 VS 项目中的工程，.inf 文件则相当于 VS 工程的 .proj 文件。

UEFI 模块类型众多，表 3-1 列出了 EDK2 中主要的模块类型。

表 3-1　UEFI 模块

模块类型	说　明
标准应用程序工程模块	在 DXE 阶段运行的应用程序（Shell 环境下也可以运行）
ShellAppMain 应用程序工程模块	Shell 环境下运行的应用程序
main 应用程序工程模块	Shell 环境下运行的应用程序，并且应用程序链接了 StdLib 库

（续）

模块类型	说　明
UEFI 驱动模块	符合 UEFI 驱动模型的驱动，仅在 BS 期间有效
库模块	作为静态库被其他模块调用
DXE 驱动模块	DXE 环境下运行的驱动，此类驱动不遵循 UEFI 驱动模型
DXE 运行时驱动模块	进入运行期仍然有效的驱动
DXE SAL 驱动模块	仅对安腾 CPU 有效的一种驱动
DXE SMM 驱动模块	系统管理模式驱动，模块被加载到系统管理内存区。系统进入运行期该驱动仍然有效
PEIM 模块	PEI 阶段的模块
SEC 模块	固件的 SEC 阶段
PEI_CORE 模块	固件的 PEI 阶段
DXE_CORE 模块	固件的 DXE 阶段

本章主要讲述 UEFI 编程常用的几种模块，包括 3 种应用程序工程模块、UEFI 驱动模块和库模块。

3.1 标准应用程序工程模块

标准应用程序工程模块是其他应用程序工程模块的基础，也是 UEFI 中常见的一种应用程序工程模块。每个工程模块由两部分组成：工程文件和源文件，标准应用程序工程模块也不例外。源文件包括 C/C++ 文件、.asm 汇编文件，也可以包括 .uni（字符串资源文件）和 .vfr（窗体资源文件）等资源文件。下面以一个简单的标准应用程序工程模块为例来介绍它的格式。

一个简单的标准应用程序工程模块包含一个 C 程序源文件（本例中为 Main.c）以及一个工程文件（Main.inf）。该示例程序在 infs\UefiMain 目录下。

3.1.1 入口函数

示例 3-1 就是这个简单模块的源程序，它仅有一个函数 UefiMain。UefiMain 就是这个模块的入口函数，其功能是向标准输出设备输出字符串“HelloWorld”。

【示例 3-1】 简单的标准应用程序。

```
#include <Uefi.h>
EFI_STATUS UefiMain(IN EFI_HANDLE ImageHandle, IN EFI_SYSTEM_TABLE *SystemTable)
{
   SystemTable->ConOut-> OutputString(SystemTable->ConOut, L"HelloWorld\n");
   return EFI_SUCCESS;
}
```

一般来说，标准应用程序至少要包含以下两个部分。

1）**头文件：**所有的 UEFI 程序都要包含头文件 Uefi.h。Uefi.h 定义了 UEFI 基本数据类型及核心数据结构。

2）**入口函数：**UEFI 标准应用程序的入口函数通常是 UefiMain。之所以说通常是 UefiMain 而不是说必须是 UefiMain，是因为入口函数可有开发者指定。UefiMain 只是一个约定俗成的函数名。入口函数由工程文件 UefiMain.inf 指定。虽然入口函数的函数名可以变化，但其函数签名（即返回值类型和参数列表类型）不能变化。

①入口函数的返回值类型是 EFI_STATUS。

- 在 UEFI 程序中基本所有的返回值类型都是 EFI_STATUS。它本质上是无符号长整数。
- 最高位为 1 时其值为错误代码，为 0 时表示非错误值。通过宏 EFI_ERROR(Status) 可以判断返回值 Status 是否为错误码。若 Status 为错误码，EFI_ERROR(Status) 返回真，否则返回假。
- EFI_SUCCESS 为预定义常量，其值为 0，表示没有错误的状态值或返回值。

②入口函数的参数 ImageHandle 和 SystemTable。

- .efi 文件（UEFI 应用程序或 UEFI 驱动程序）加载到内存后生成的对象称为 Image（映像）。ImageHandle 是 Image 对象的句柄，作为模块入口函数参数，它表示模块自身加载到内存后生成的 Image 对象。
- SystemTable 是程序同 UEFI 内核交互的桥梁，通过它可以获得 UEFI 提供的各种服务，如启动（BT）服务和运行时（RT）服务。SystemTable 是 UEFI 内核中的一个全局结构体。

向标准输出设备打印字符串是通过 SystemTable 的 ConOut 提供的服务完成的。ConOut 是 EFI_SIMPLE_TEXT_OUTPUT_PROTOCOL 的一个实例。而 EFI_SIMPLE_TEXT_OUTPUT_PROTOCOL 的主要功能是控制字符输出设备。向输出设备打印字符串是通过 ConOut 提供的 OutputString 服务完成的。该服务（函数）的第一个参数是 This 指针，指向 EFI_SIMPLE_TEXT_OUTPUT_PROTOCOL 实例（此处为 ConOut）本身；第二个参数是 Unicode 字符串。关于 Protocol 和 This 指针将在第 4 章详细介绍。简而言之，这条打印语句的意义就是通过 SystemTable → ConOut → OutputString 服务将字符串 L“HelloWorld”打印到 SystemTable → ConOut 所控制的字符输出设备。

3.1.2　工程文件

要想编译 Main.c，还需要编写 .inf（Module Information File）文件。.inf 文件是模块的工程文件，其作用相当于 Makefile 文件或 Visual Studio 的 .proj 文件，用于指导 EDK2 编译工具自动编译模块。

注意 在工程文件中，字符 # 后面的内容为注释。

工程文件分为很多个块，每个块以“[块名]”开头，“[块名]”必须单独占一行。有些块是所有工程文件都必需的块，这些块包括 [Defines]、[Sources]、[Packages] 和 [LibraryClasses]，见表 3-2。其他的块并不是每个模块都一定要编写的块，仅在用到的时候需要编写这些块，表 3-3 列出了一些非必需块。

表 3-2 工程文件必需块

必需块	块描述
[Defines]	定义本模块的属性变量及其他变量，这些变量可在工程文件其他块中引用
[Sources]	列出本模块的所有源文件及资源文件
[Packages]	列出本模块引用到的所有包的包声明文件。可能引用到的资源包括头文件、GUID、Protocol 等，这些资源都声明在包的包声明文件 .dec 中
[LibraryClasses]	列出本模块要链接的库模块

表 3-3 工程文件非必需块

非必需块	块描述
[Protocols]	列出本模块用到的 Protocol
[Guids]	列出本模块用到的 GUID
[BuildOptions]	指定编译和链接选项
[Pcd]	Pcd 全称为平台配置数据库（Platform Configuration Database）。[Pcd] 用于列出本模块用到的 Pcd 变量，这些 Pcd 变量可被整个 UEFI 系统访问
[PcdEx]	用于列出本模块用到的 Pcd 变量，这些 Pcd 变量可被整个 UEFI 系统访问
[FixedPcd]	用于列出本模块用到的 Pcd 编译期常量
[FeaturePcd]	用于列出本模块用到的 Pcd 常量
[PatchPcd]	列出的 Pcd 变量仅本模块可用

下面以我们编写的简单标准应用程序工程模块为例分别讲述几个必需块及其他常用块的用法。

1. [Defines] 块

[Defines] 块用于定义模块的属性和其他变量，块内定义的变量可被其他块引用。

（1）属性定义语法

```
属性名 = 属性值
```

（2）属性

块内必须定义的属性包括：

- INF_VERSION：INF 标准的版本号。EDK2 的 build 会检查 INF_VERSION 的值并根据这个值解释 .inf 文件。最新的 INF 标准版本号为 0x00010016，前半部分为主版本号，后半部分为次版本号。通常将 INF_VERSION 设置为 0x00010005 即可。
- BASE_NAME：模块名字字符串，不能包含空格。它通常也是输出文件的名字。例如，Base_Name 为 UefiMain，则最终生成的文件为 UefiMain.efi。
- FILE_GUID：每个工程文件必须有一个 8-4-4-4-12 格式的 GUID，用于生成固件。例如，FILE_GUID = 6987936E-ED34-ffdb-AE97-1FA5E4ED2117。
- VERSION_STRING：模块的版本号字符串。例如，可以设置为 VERSION_STRING= 1.0。
- MODULE_TYPE：定义模块的模块类型，可以是 SEC、PEI_CORE、PEIM、DXE_CORE、DXE_SAL_DRIVER、DXE_SMM_DRIVER、UEF_DRIVER 、DXE_DRIVER、DXE_RUNTIME_DRIVER、UEFI_APPLICATION、BASE 中的一个。对标准应用程序工程模块来说，MODULE_TYPE 的值为 UEFI_APPLICATION。
- ENTRY_POINT：定义模块的入口函数。在上文中我们提到 UefiMain 是模块的入口函数，就是在此处通过设置 ENTRY_POINT 的值为 UefiMain 得到的。

示例 3-2 是标准应用程序工程模块示例工程文件中完整的 [Defines] 块。

【示例 3-2】 标准应用程序工程模块工程文件的 [Defines]。

```
[Defines]
  INF_VERSION = 0x00010005
  BASE_NAME = UefiMain
  FILE_GUID = 6987936E-ED34-ffdb-AE97-1FA5E4ED2117
  MODULE_TYPE = UEFI_APPLICATION
  VERSION_STRING = 1.0
  ENTRY_POINT = UefiMain
```

2. [Sources] 块

[Sources] 用于列出模块的所有源文件和资源文件。

（1）语法

块内每一行表示一个文件，文件使用相对路径，根路径是工程文件所在的目录。作为示例的标准应用程序工程模块仅含有一个源文件。[Sources] 块如下所示：

```
[Sources.$(Arch)]
  UefiMain.c
```

（2）体系结构相关块

.$（Arch）是可选项，可以是 IA32、X64、IPF、EBC、ARM 中的一个，表示本块适用的体系结构。[Sources] 块适用于任何体系结构。例如，编译 32 位模块时（即在 build 命令选项中使用了 -a IA32 选项），工程将包含 [Sources] 和 [Sources.IA32] 中的源文件；编译 64 位

模块时，工程将包含 [Sources] 和 [Sources.X64] 中的源文件。

（3）示例

示例 3-3 包含了三个块：[Sources]、[Sources.IA32] 和 [Sources.X64]。在编译 32 位模块时，模块包含 [Sources.IA32] 中的文件 Cpu32.c 和 [Sources] 中的 Common.c；在编译 64 位模块时，模块包含文件 [Sources.X64] 中的 Cpu64.c 和 [Sources] 中的 Common.c。

【示例 3-3】 标准应用程序工程模块的 [Sources]。

```
[Sources]
   Common.c
[Sources.IA32]
   Cpu32.c
[Sources.X64]
   Cpu64.c
```

（4）编译工具链相关的源文件

有时文件名后面会有一个该号符号，该符号后面会跟工具链名字，如示例 3-4 所示。

【示例 3-4】 工具链相关的源文件。

```
[Sources]
   TimerWin.c   | MSFT
   TimerLinux.c   | GCC
```

这表示 TimerWin.c 仅在使用 Visual Studio 编译器时有效，TimerLinux.C 仅在使用 GCC 编译器时有效。除了 MSFT 和 GCC 外，EDK2 还定义了 INTEL 和 RVCT 两种工具链。INTEL 工具链是 ICC 编译器或 Intel EFI 字节码编译器；RVCT 是 ARM 编译器。

3. [Packages] 块

[Packages] 列出本模块引用到的所有包的包声明文件（.dec 文件）。

（1）语法

[Packages] 块内每一行列出一个文件，文件使用相对路径，相对路径的根路径为 EDK2 的根目录。若 [Sources] 块内列出了源文件，则在 [Packages] 块必须列出 MdePkg/MdePkg.dec，并将其放在本块的首行。

（2）示例

在本节介绍的这个简单标准应用程序工程模块示例中，UefiMain.c 仅仅引用了 MdePkg 中的头文件 Uefi.h，因而 [Packages] 仅列出 MdePkg/MdePkg.dec 即可，如示例 3-5 所示。

【示例 3-5】 工程文件的 [Packages] 块。

```
[Packages]
  MdePkg/MdePkg.dec
```

4. [LibraryClasses]

[LibraryClasses] 块列出本模块要链接的库模块。

（1）语法

块内每一行声明一个要链接的库（库定义在包的 .dsc 文件中，定义方法将在下文讲述）。语法如下：

```
[LibraryClasses]
   库名称
```

（2）常用库

应用程序工程模块必须链接 UefiApplicationEntryPoint 库；驱动模块必须链接 UefiDriverEntryPoint 库。

（3）示例

本示例中，UefiMain.c 文件的 UefiMain 函数没有使用其他库函数，因而在 [LibraryClasses] 块列出 UefiApplicationEntryPoint 即可，如下所示：

```
[LibraryClasses]
  UefiApplicationEntryPoint
```

5. [Protocols] 块

[Protocols] 列出模块中使用的 Protocol，严格来说，列出的是 Protocol 对应的 GUID。如果模块未使用任何 Protocol，则此块为空。

（1）语法

块内每一行声明一个在本模块中引用的 Protocol。语法如下：

```
[LibraryClasses]
   Protocol 的 GUID
```

（2）示例

如果在程序中使用了某个 Protocol 的 GUID，例如，源程序中使用了如下代码：

```
Status = gBS->LocateProtocol ( &gEfiHiiDatabaseProtocolGuid,
              NULL,  (VOID **) &HiiDatabase );
```

则在 [Protocols] 块必须列出 gEfiHiiDatabaseProtocolGuid，如示例 3-6 所示。

【示例 3-6】 工程文件的 [LibraryClasses]。

```
[LibraryClasses]
   gEfiHiiDatabaseProtocolGuid
```

6. [BuildOptions] 块

[BuildOptions] 指定本模块的编译和连接选项。

（1）语法

```
[BuildOptions]
 [编译器家族]: [$(Target)]_[TOOL_CHAIN_TAG]_[$(Arch)]_[CC|DLINK]_FLAGS[=|==] 选项
```

说明如下：

- 编译器家族可以是 MSFT（Visual Studio 编译器家族）、INTEL（Intel 编译器家族）、GCC（GCC 编译器家族）和 RVCT（ARM 编译器家族）中的一个。
- Target 是 DEBUG、RELEASE 和 * 中的一个，* 为通配符，表示对 DEBUG 和 RELEASE 都有效。
- TOOL_CHAIN_TAG 是编译器名字。编译器名字定义在 Conf\tools_def.txt 文件中，预定义的编译器名字有 VS2003、VS2005、VS2008、VS2010、GCC44、GCC45、GCC46、CYGGCC、ICC 等，* 表示对指定编译器家族内的所有编译器都有效。
- Arch 是体系结构，可以是 IA32、X64、IPF、EBC 或 ARM，* 表示对所有体系结构都有效。
- CC 表示编译选项。DLINK 表示连接选项。
- = 表示选项附加到默认选项后面。== 表示仅使用所定义的选项，弃用默认选项。
- = 或 == 号后面是编译选项或连接选项。

（2）示例

示例 3-7 表示使用 Visual Studio 编译器编译时添加 /wd4804 编译选项，连接时添加 /BASE:0 选项。

【示例 3-7】 使用“=”的 [BuildOptions]。

```
[BuildOptions]
  MSFT:*_*_*_CC_FLAGS = /wd4804
  MSFT:*_*_*_DLINK_FLAGS = /BASE:0
```

示例 3-8 表示使用 VS2010 编译 32 位 DEBUG 版本时仅使用指定的编译选项，忽略所有默认的编译选项。

【示例 3-8】 使用“==”的 [BuildOptions]。

```
[BuildOptions]
MSFT:DEBUG_VS2010_IA32_CC_FLAGS ==  /nologo /c /WX /GS- /W4 /Gs32768 /D
UNICODE /O1ib2 /GL  /EHs-c- /GR- /GF /Gy /Zi /Gm /D EFI_SPECIFICATION_VERSION=
0x0002000A /D TIANO_RELEASE_VERSION=0x00080006 /FAs /Oi-
```

最后将所有的块放在一起，组成完整的标准应用程序工程文件，如示例 3-9 所示。

【示例 3-9】 标准应用程序 HelloWorld 的完整工程文件。

```
[Defines]
  INF_VERSION = 0x00010005
```

```
  BASE_NAME = UefiMain
  FILE_GUID = 6987936E-ED34-ffdb-AE97-1FA5E4ED2117
  MODULE_TYPE = UEFI_APPLICATION
  VERSION_STRING = 1.0
  ENTRY_POINT = UefiMain
[Sources]
   main.c
[Packages]
  MdePkg/MdePkg.dec
[LibraryClasses]
  UefiApplicationEntryPoint
  UefiLib
```

3.1.3　编译和运行

源文件和工程文件都已经编写完成，下面编译和运行这个标准应用程序工程模块。将 UefiMain.inf 添加到 Nt32Pkg.dsc 或 UnixPkg.dsc 的 [Components] 部分，例如添加下面一行代码（uefi 目录在 EDK2 下）：

```
[Components]
uefi/book/infs/UefiMain.inf
```

然后就可以使用 BaseTools 下的 build 工具进行编译了。

Windows 下执行如下命令进行编译：

```
C:\EDK2>edksetup.bat --nt32
C:\EDK2>build -p Nt32PkgNt32Pkg.dsc -a IA32
```

Linux 下执行如下命令进行编译：

```
$>source ./edksetup.sh BaseTools
$>build -p UnixPkg/UnixPkg.dsc -a IA32
```

在 UEFI 模拟器中执行 UefiMain 命令，输出如图 3-1 所示。

图 3-1　标准应用程序 UefiMain.efi 的输出

3.1.4　标准应用程序的加载过程

下面深入分析一下应用程序的加载和调用过程。开始介绍之前，还要了解一下应用程序是如何被编译成 .efi 文件的，整个过程分为以下三步。

1）UefiMain.c 首先被编译成目标文件 UefiMain.obj。

2）连接器将目标文件 UefiMain.obj 和其他库连接成 UefiMain.dll。

3）GenFw 工具将 UefiMain.dll 转换成 UefiMain.efi。

整个过程由 build 命令自动完成。第 2）和 3）步的命令如图 3-2 所示。

```
link.exe /OUT:d:\edk2\Build\...\DEBUG\UefiMain.dll /NOLOGO /NODEFAULTLIB
    /IGNORE:4001 /OPT:REF /OPT:ICF=10 /MAP /ALIGN:32 /SECTION:.xdata,D
    /SECTION:.pdata,D /MACHINE:X86 /LTCG /DLL /ENTRY:_ModuleEntryPoint
    /SUBSYSTEM:EFI_BOOT_SERVICE_DRIVER /SAFESEH:NO /BASE:0 /DRIVER/DEBUG
    /EXPORT:InitializeDriver=_ModuleEntryPoint /BASE:0x10000 /ALIGN:4096
    /FILEALIGN:4096 /SUBSYSTEM:CONSOLE  @d:\edk2\Build\...\OUTPUT\static_
        library_files.lst
"GenFw" -e UEFI_APPLICATION -o d:\edk2\Build\...\DEBUG\UefiMain.efi d:\edk2\Build\
…\DEBUG\UefiMain.dll
```

图 3-2　标准应用程序连接和 GenFw 过程

说明：连接器在生成 UefiMain.dll 时使用了 /dll/entry:_ModuleEntryPoint。.efi 是遵循 PE32 格式的二进制文件，_ModuleEntryPoint 便是这个二进制文件的入口函数。

3.1.1 节讲到模块的入口函数是 UefiMain，_ModuleEntryPoint 与 UefiMain 是什么关系呢？让我们带着这个疑问来看应用程序的加载过程。

1. 将 UefiMain.efi 文件加载到内存

首先来看 UefiMain.efi 文件是如何加载到内存的。当在 Shell 中执行 UefiMain.efi 时，Shell 首先用 gBS->LoadImage() 将 UefiMain.efi 文件加载到内存生成 Image 对象，然后调用 gBS->StartImage(Image) 启动这个 Image 对象。具体加载过程如代码清单 3-1 所示。

代码清单 3-1　应用程序的加载

```
//@file ShellPkg\Application\Shell\ShellProtocol.c
EFI_STATUS EFIAPI InternalShellExecuteDevicePath(
  IN CONST EFI_HANDLE*ParentImageHandle,
  IN CONST EFI_DEVICE_PATH_PROTOCOL *DevicePath,        //UefiMain.efi 的设备路径
  IN CONST CHAR16 *CommandLine OPTIONAL,                //应用程序所需的命令行参数
  IN CONST CHAR16 **Environment OPTIONAL,               //UEFI 环境变量
  OUT EFI_STATUS *StatusCode OPTIONAL                   //程序 UefiMain.efi 的返回值
  )
{
  EFI_STATUS                    Status;
  EFI_HANDLE                    NewHandle;
  EFI_LOADED_IMAGE_PROTOCOL     *LoadedImage;
  LIST_ENTRY                    OrigEnvs;
  EFI_SHELL_PARAMETERS_PROTOCOL ShellParamsProtocol;
…
//第一步：将 UefiMain.efi 文件加载到内存，生成 Image 对象，NewHandle 是这个对象的句柄
```

```
Status = gBS->LoadImage(
    FALSE,
    *ParentImageHandle,
    (EFI_DEVICE_PATH_PROTOCOL*)DevicePath,
    NULL,
    0,
    &NewHandle);

if (EFI_ERROR(Status)) {
    if (NewHandle != NULL) {
        gBS->UnloadImage(NewHandle);
    }
    return (Status);
  }
//第二步：取得命令行参数，并将命令行参数交给 UefiMain.efi 的 Image 对象，即 NewHandle
Status = gBS->OpenProtocol(
    NewHandle,
    &gEfiLoadedImageProtocolGuid,
    (VOID**)&LoadedImage,
    gImageHandle,
    NULL,
    EFI_OPEN_PROTOCOL_GET_PROTOCOL);

if (!EFI_ERROR(Status)) {
    ASSERT(LoadedImage->LoadOptionsSize == 0);
    if (CommandLine != NULL) {
        LoadedImage->LoadOptionsSize  = (UINT32)StrSize(CommandLine);
        LoadedImage->LoadOptions      = (VOID*)CommandLine;
}
}
...
//第三步：启动所加载的 Image
if (!EFI_ERROR(Status)) {
    if (StatusCode != NULL) {
        *StatusCode = gBS->StartImage(NewHandle, NULL, NULL);
    } else {
        Status      = gBS->StartImage(NewHandle, NULL, NULL);
    }
}
...
//退出应用程序后清理资源
}
```

加载应用程序中最重要的一步，也是我们最关心的部分，就是 gBS->StartImage(NewHandle，NULL，NULL)。StartImage 的主要作用是找出可执行程序映像（Image）的入口函数并执行找到的入口函数。gBS->StartImage 是个函数指针，它实际指向 CoreStartImage 函数。

2. 进入映像的入口函数

CoreStartImage 的主要作用是调用映像的入口函数。下面来看 CoreStartImage 是如何做的，具体如代码清单 3-2 所示。

代码清单 3-2　启动应用程序

```
//@file MdeModulePkg\Core\Dxe\Image\Image.c
EFI_STATUS EFIAPI CoreStartImage (IN EFI_HANDLE  ImageHandle,
  OUT UINTN  *ExitDataSize,  OUT CHAR16 **ExitData  OPTIONAL)
{
  EFI_STATUS Status;
  LOADED_IMAGE_PRIVATE_DATA *Image;
  LOADED_IMAGE_PRIVATE_DATA *LastImage;
  UINT64 HandleDatabaseKey;
  UINTN SetJumpFlag;
  UINT64 Tick;
  EFI_HANDLE Handle;

  //设置 LongJump，用于退出此程序
  Image->JumpBuffer = AllocatePool (sizeof (BASE_LIBRARY_JUMP_BUFFER) +
      BASE_LIBRARY_JUMP_BUFFER_ALIGNMENT);
  if (Image->JumpBuffer == NULL) {
    return EFI_OUT_OF_RESOURCES;
  }
  Image->JumpContext = ALIGN_POINTER (Image->JumpBuffer, BASE_LIBRARY_JUMP_
      BUFFER_ALIGNMENT);
  SetJumpFlag = SetJump (Image->JumpContext);
  //首次调用 SetJump() 返回 0。通过 LongJump(Image->JumpContext) 跳转到此处时返回非零值
  if (SetJumpFlag == 0) {
    //调用 Image 的入口函数
    Image->Started = TRUE;
    Image->Status = Image->EntryPoint (ImageHandle, Image->Info.SystemTable);
    //设置 Image 执行后的状态，然后通过 LongJump 跳到应用程序退出点
    CoreExit (ImageHandle, Image->Status, 0, NULL);
  }
  //此处是应用程序退出点
  //程序通过 LongJump 跳转到此处，然后根据 Image->Status 进行错误处理
...
}
```

在 gBS->StartImage 中，SetJump/LongJump 为应用程序的执行提供了一种错误处理机制，执行流程如图 3-3 所示。

gBS->StartImage 的核心是 Image->EntryPoint(…)，它就是程序映像的入口函数，对应用程序来说，就是 _ModuleEntryPoint 函数。进入 _ModuleEntryPoint 后，控制权才转交给应用程序（此处就是我们的 UefiMain.efi）。代码清单 3-3 是 _ModuleEntryPoint 的代码。

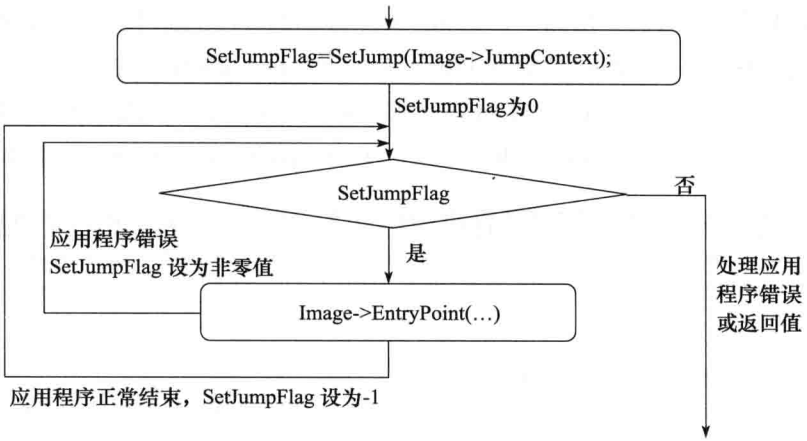


图 3-3　gBS->StartImage 执行流程

代码清单 3-3　程序映像的入口函数 _ModuleEntryPoint

```
//@file MdePkg\UefiApplicationEntryPoint\ApplicationEntryPoint.c
EFI_STATUS EFIAPI _ModuleEntryPoint ( IN EFI_HANDLE ImageHandle,
  IN EFI_SYSTEM_TABLE  *SystemTable )
{
  EFI_STATUS Status;
  if (_gUefiDriverRevision != 0) {
    //确保系统平台的 UEFI 版本号大于或等于 ImageHandle 的 UEFI 版本号
      if (SystemTable->Hdr.Revision < _gUefiDriverRevision) {
        return EFI_INCOMPATIBLE_VERSION;
      }
  }
  //所有将被使用的库的构造函数
  ProcessLibraryConstructorList (ImageHandle, SystemTable);
  //调用 Image 的入口函数
  Status = ProcessModuleEntryPointList (ImageHandle, SystemTable);
  //所有库的析构函数
  ProcessLibraryDestructorList (ImageHandle, SystemTable);
  return Status;
}
```

_ModuleEntryPoint 主要处理以下三个事情。

1）**初始化**：在初始化函数 ProcessLibraryConstructorList 中会调用一系列的构造函数。

2）**调用本模块的入口函数**：在 ProcessModuleEntryPointList 中会调用应用程序工程模块真正的入口函数（即我们在 .inf 文件中定义的入口函数 UefiMain）。

3）**析构**：在析构函数 ProcessLibraryDestructorList 中会调用一系列析构函数。

那么这三个 Process* 函数是在哪里定义的呢？在命令行执行 build 命令的时候，build 命令会解析模块的工程文件（即 .inf 文件），然后生成 AutoGen.h 和 AutoGen.c，这三个函数便

是 AutoGen.c 中的一部分。一般而言，在 .inf 文件的 [LibraryClasses] 段声明了某个库后，如果这个库有构造函数，AutoGen 便会在 ProcessLibraryConstructorList 中加入这个库的构造函数。另外，ProcessLibraryConstructorList 还会加入启动服务和运行时服务的构造函数。代码清单 3-4 是标准应用程序工程模块 HelloWorld 的 ProcessLibraryConstructorList 函数。

代码清单 3-4　标准应用程序工程模块 HelloWorld 的 ProcessLibraryConstructorList 函数

```
VOID EFIAPI ProcessLibraryConstructorList (IN EFI_HANDLE ImageHandle,
  IN EFI_SYSTEM_TABLE  *SystemTable)
{
  EFI_STATUS  Status;
  //初始化全局变量 gBS、gST 和 gImageHandle
  Status = UefiBootServicesTableLibConstructor (ImageHandle, SystemTable);
  ASSERT_EFI_ERROR (Status);
  //初始化全局变量 gRT
  Status = UefiRuntimeServicesTableLibConstructor (ImageHandle, SystemTable);
  ASSERT_EFI_ERROR (Status);
  //初始化 UefiLib，Print 函数就是在 UefiLib 中实现的
  Status = UefiLibConstructor (ImageHandle, SystemTable);
  ASSERT_EFI_ERROR (Status);
}
```

gBS 指向启动服务表，gST 指向系统表（System Table），gImageHandle 指向正在执行的驱动或应用程序，gRT 指向运行时服务表，这几个全局变量在开发应用程序和驱动时会经常用到。使用 gBS、gST、gImageHandle 前需加入 #include<include/UefiBootServicesTableLib.h>。使用 gRT 之前需加入 #include<include/UefiRuntimeServicesTableLib.h>

与构造函数相似，AutoGen 会在析构函数中调用相应 Library 的析构函数。代码清单 3-5 是 Hello World 标准应用程序工程模块的析构函数。Hello World 模块的析构函数 ProcessLibraryDestructorList 函数为空，因为 UefiBootServicesTableLib、UefiRuntimeServicesTableLib、UefiLib 这三个 Library 都没有析构函数。

代码清单 3-5　标准应用程序工程模块 HelloWorld 的析构函数 ProcessLibraryDestructorList

```
VOID EFIAPI ProcessLibraryDestructorList (IN EFI_HANDLE        ImageHandle,
  IN EFI_SYSTEM_TABLE  *SystemTable )
{
}
```

3. 进入模块入口函数

在 ProcessModuleEntryPointList 中，调用了应用程序工程模块的真正入口函数 UefiMain，如代码清单 3-6 所示。

代码清单 3-6　标准应用程序工程模块 HelloWorld 的 ProcessModuleEntryPointList 函数

```
EFI_STATUS EFIAPI ProcessModuleEntryPointList (  IN EFI_HANDLE ImageHandle,
  IN EFI_SYSTEM_TABLE  *SystemTable  )
{
  return UefiMain (ImageHandle, SystemTable);
}
```

至此，我们已经了解了标准应用程序工程模块入口函数调用的整个过程，再来简单回顾一下整个过程：StartImage → _ModuleEntryPoint → ProcessModuleEntryPointList → UefiMain。

通过本节的学习，我们还了解了标准应用程序工程模块的组成、入口函数 UefiMain 的结构，以及 .inf 文件的结构。下面继续学习其他类型工程模块的编写方法。

3.2　其他类型工程模块

常用的工程模块除了标准应用程序工程模块外，还有 Shell 应用程序工程模块、使用 main 函数的应用程序工程模块、库模块和驱动模块，下面分别介绍这几种模块。

3.2.1　Shell 应用程序工程模块

从 3.1.4 节的讲述可以看出，标准应用程序处理命令行参数很不方便。但是，能在 Shell 中执行的命令（命令也是应用程序）通常都会带有命令行参数，为了方便开发者开发能在 Shell 环境下执行的应用程序，EDK2 提供了一种特殊的应用程序工程模块，这种模块以 INTN ShellAppMain(IN UINTN Argc，IN CHAR16**Argv) 作为入口函数。我们称这种模块为 Shell 应用程序工程模块。一个简单的示例如示例 3-10 所示。

【示例 3-10】 Shell 应用程序工程模块的入口函数。

```
#include <Uefi.h>
#include <Library/UefiBootServicesTableLib.h>
INTN ShellAppMain (IN UINTN Argc,  IN CHAR16 **Argv)
{
  gST -> ConOut-> OutputString(gST -> ConOut, L"HelloWorld\n");
  return 0;
}
```

因为在入口函数的参数中没有了 SystemTable，所以要通过全局变量 gST 使用系统表。入口函数 ShellAppMain 的第一个参数 Argc 是命令行参数个数，第二个参数 Argv 是命令行参数列表，Argv 列表中的每个参数都是 Unicode 字符串（CHAR16* 类型的字符串）。在 UEFI 中使用的字符串通常都是 Unicode 字符串。

在介绍标准应用程序工程模块（参见 3.1 节）时讲过，每一种模块都由两部分组成：工

程文件和源文件，Shell 应用程序工程模块也不例外。上文介绍了源文件的格式，下面介绍一下工程文件的编写方法。

1）首先处理的是 [Defines] 块。在 [Defines] 块中，将 MODULE_TPYE 设置为 UEFI_APPLICATION，这一点与标准应用程序工程模块相同。然后将 ENTRY_POINT 设置为 ShellCEntryLib，这里可以看出与标准应用程序工程模块的不同，标准应用程序工程模块的 ENTRY_POINT 为 UefiMain，同时在源程序中开发者需实现 UefiMain 函数，也就是说，开发者需提供由 ENTRY_POINT 指定的入口函数。在 Shell 应用程序工程模块中，ENTRY_POINT 必须为 ShellCEntryLib，而在源程序中开发者必须提供 ShellAppMain。

2）在 [Packages] 块中，必须列出 MdePkg/MdePkg.dec 和 ShellPkg/ShellPkg.dec。

3）在 [LibraryClasses] 块中，必须列出 ShellCEntryLib，通常还要列出 UefiBootServicesTableLib 和 UefiLib。

完整的工程文件如示例 3-11 所示。

【示例 3-11】 Shell 应用程序工程模块的工程文件。

```
[Defines]
  INF_VERSION = 0x00010006
  BASE_NAME = Main
  FILE_GUID = 4ea97c46-7491-4dfd-b442-747010f3ce5f
  MODULE_TYPE = UEFI_APPLICATION
  VERSION_STRING = 0.1
  ENTRY_POINT = ShellCEntryLib
[Sources]
  Main.c
[Packages]
  MdePkg/MdePkg.dec
  ShellPkg/ShellPkg.dec
[LibraryClasses]
  ShellCEntryLib
  UefiBootServicesTableLib
  UefiLib
```

UEFI 的入口函数 ShellCEntryLib 究竟做了哪些工作呢？可以分析一下 ShellCEntryLib 的源码。该函数位于 ShellPkg\Library\UefiShellCEntryLib\UefiShellCEntryLib.c 中，其源码如代码清单 3-7 所示。

代码清单 3-7　ShellCEntryLib 函数

```
// ShellCEntryLib 库提供的应用程序入口函数，该函数最终会调用用户的入口函数 ShellAppMain
EFI_STATUS EFIAPI ShellCEntryLib (
  IN EFI_HANDLE ImageHandle,                    // UEFI 应用程序的 ImageHandle
  IN EFI_SYSTEM_TABLE  *SystemTable             // UEFI 系统的 EFI 系统表
  )
```

```
{
  INTN ReturnFromMain;
  EFI_SHELL_PARAMETERS_PROTOCOL *EfiShellParametersProtocol;
  EFI_SHELL_INTERFACE *EfiShellInterface;
  EFI_STATUS Status;
  ReturnFromMain = -1;
  EfiShellParametersProtocol = NULL;
  EfiShellInterface = NULL;
  //首先取得命令行参数
  Status = SystemTable->BootServices->OpenProtocol(ImageHandle,
    &gEfiShellParametersProtocolGuid,
    (VOID **)&EfiShellParametersProtocol,
    ImageHandle,
    NULL,
    EFI_OPEN_PROTOCOL_GET_PROTOCOL);
  if (!EFI_ERROR(Status)) {
    //通过 Shell 2.0 接口获得命令行参数
    //调用用户的入口函数
    ReturnFromMain = ShellAppMain ( EfiShellParametersProtocol->Argc,
                     EfiShellParametersProtocol->Argv);
  } else {
    //打开 Shell 2.0 Protocol 失败，尝试 Shell 1.0
    Status = SystemTable->BootServices->OpenProtocol(ImageHandle,
      &gEfiShellInterfaceGuid,
      (VOID **)&EfiShellInterface,
      ImageHandle,
      NULL,
      EFI_OPEN_PROTOCOL_GET_PROTOCOL
      );
    if (!EFI_ERROR(Status)) {
      //通过 Shell 1.0 接口获得命令行参数
      //调用用户的入口函数
      ReturnFromMain = ShellAppMain ( EfiShellInterface->Argc, EfiShellInterface->Argv);
    } else {
      ASSERT(FALSE);
    }
  }
  if (ReturnFromMain == 0) {
    return (EFI_SUCCESS);
  } else {
    return (EFI_UNSUPPORTED);
  }
}
```

可以看出，ShellCEntryLib 函数的输入参数与我们前面用到的 UefiMain 函数的输入参数完全相同，就是标准的 UEFI application 入口函数。在 ShellCEntryLib 中，首先打开 EFI_SHELL_PARAMETERS_PROTOCOL，通过 EFI_SHELL_PARAMETERS_PROTOCOL 可以获

得命令行参数个数 Argc 及命令行参数数组 Argv，然后调用 ShellAppMain 函数。代码清单 3-8 是 EFI_SHELL_PARAMETERS_PROTOCOL 数据结构，在可执行文件被 UEFI 通过 Load Image Protocol 载入 UEFI 系统时，会在可执行文件的 Image Handle 上安装 Shell Protocol 和 Shell Parameter Protocol。

代码清单 3-8 EFI_SHELL_PARAMETERS_PROTOCOL 数据结构

```
//@file ShellPkg\Include\Protocol\EfiShellParameters.h
typedef struct _EFI_SHELL_PARAMETERS_PROTOCOL {
  CHAR16 **Argv;                    //命令行参数数组，第一个参数是可执行文件的全路径
  UINTN Argc;                       //Argv 数组的元素个数
  SHELL_FILE_HANDLE StdIn;          //标准输入句柄
  SHELL_FILE_HANDLE StdOut;         //标准输出句柄
  SHELL_FILE_HANDLE StdErr;         //标准错误句柄
} EFI_SHELL_PARAMETERS_PROTOCOL;
```

3.2.2 使用 main 函数的应用程序工程模块

对 C 语言程序员来说，最熟悉的程序莫过于 main 函数。EDK2 也提供了使用 main 函数的应用程序工程模块，通常在此类应用程序中都会使用 C 标准库（StdLib）中的函数。示例 3-12 是一个使用 main 函数的应用程序工程模块的简单示例。

【示例 3-12】 使用 main 函数的应用程序工程模块的源文件。

```
#include <stdio.h>
int main (int argc, char **argv)
{
  printf("HelloWorld\n");
  return 0;
}
```

在工程文件中要进行如下设置。

- 在 [Defines] 块中，设置 MODULE_TPYE 为 UEFI_APPLICATION。
- 在 [Defines] 块中，设置 ENTRY_POINT 为 ShellCEntryLib。
- 在 [Packages] 块中，列出 MdePkg/MdePkg.dec、ShellPkg/ShellPkg.dec 和 StdLib/StdLib.dec。
- 在 [LibraryClasses] 块中，列出 ShellCEntryLib（提供 ShellCEntryLib 函数）、LibC（提供 ShellAppMain 函数）及 LibStdio（提供 printf 函数）库。
- 在 [Sources] 中，列出源文件 main.c。

回忆一下 3.2.1 节，Shell 应用程序工程模块使用了 ShellCEntryLib，然后我们自己实现了 ShellAppMain 函数作为程序的入口函数。

在使用 main 函数的应用程序工程模块中使用了 StdLib，而 StdLib 提供了 ShellAppMain 函数。开发者要实现 int main(int Argc，char** Argv) 作为程序的入口函数以供 ShellAppMain

调用。而真正的模块入口函数是 ShellCEntryLib，调用过程为 ShellCEntryLib->ShellAppMain->main。

完整的工程文件如示例 3-13 所示。

【示例 3-13】 使用 main 函数的应用程序工程模块的工程文件。

```
[Defines]
  INF_VERSION = 0x00010006
  BASE_NAME = Main
  FILE_GUID = 4ea97c46-7491-4dfd-b442-747010f3ce5f
  MODULE_TYPE = UEFI_APPLICATION
  VERSION_STRING = 0.1
  ENTRY_POINT = ShellCEntryLib
[Sources]
  main.c
[Packages]
  MdePkg/MdePkg.dec
  ShellPkg/ShellPkg.dec
  StdLib/StdLib.dec
[LibraryClasses]
  LibC
  LibStdio
  ShellCEntryLib
```

还要再说明一点，如果用户的程序中用到了 printf(...) 等标准 C 的库函数，那么一定要使用此种类型的应用程序工程模块。ShellCEntryLib 函数中会调用 StdLib 的 ShellAppMain(...)，这个 ShellAppMain 函数会对 StdLib 进行初始化。StdLib 的初始化完成后才可以调用 StdLib 的函数。关于 StdLib 的使用将在后面章节详细介绍。

通常，使用 main 函数的应用程序工程模块在 AppPkg 环境下才能成功编译。首先将 main.inf 添加到 AppPkg\AppPkg.dsc 文件的 [Components]。

```
##  @file  AppPkg.dsc
[Components]
  uefi\book\infs\main\main.inf
```

然后可以用如下命令编译这个工程：

```
build -p AppPkg\AppPkg.dsc -m uefi\book\infs\main\main.inf
```

3.2.3　库模块

开发大型工程的时候经常会用到库，例如我们要开发视频解码程序，会用到 zlib 库。在库模块的工程文件中，需要设置 MODULE_TYPE 为 BASE；设置 LIBRARY_CLASS 为 library 的名字，例如 zlib。同时，不要设置 ENTRY_POINT。[Packages] 块列出库引用到的包，[LibraryClasses] 列出包所依赖的其他库。示例 3-14 是 zlib 库的工程文件。

【示例 3-14】 zlib 库的工程文件。

```
[Defines]
  INF_VERSION = 0x00010005
  BASE_NAME = zlib
  FILE_GUID = 348aaa62-BFBD-4882-9ECE-C80BBbbbb736
  VERSION_STRING = 1.0
  MODULE_TYPE = BASE
  LIBRARY_CLASS = zlib
[Sources]
  adler32.c
  # 此处不一一列举 zlib 中的源文件，感兴趣的读者可以参考本书附带的源码 (book\ffmpeg\zlib\zlib.inf)
[Packages]
  MdePkg/MdePkg.dec
  MdeModulePkg/MdeModulePkg.dec
  StdLib/StdLib.dec
[LibraryClasses]
  MemoryAllocationLib
  BaseLib
  UefiBootServicesTableLib
  BaseMemoryLib
  UefiLib
```

有些库仅能被某些特定的模块调用，编写这种库时需在工程文件中声明库的适用范围，声明方法是在 [Defines] 块的 LIBRARY_CLASS 变量中定义，格式如下：

```
LIBRARY_CLASS = 库名字 | 适用模块类型 1 适用模块类型 2
```

例如，如果想使 zlib 库仅能被应用程序工程模块调用，那么 LIBRARY_CLASS 需设置为：

```
LIBRARY_CLASS  = zlib | UEFI_APPLICATION
```

编写了库之后，要使库能被其他模块调用，还要在包的 .dsc 文件中声明该库，例如要使得 AppPkg 中的模块能调用 zlib 库，需将 zlib|zlib-1.2.6/zlib.inf（假设 zlib-1.2.6 在 EKD2 的根目录下）放到 AppPkg\AppPkg.dsc 文件 [LibraryClasses] 中，如下所示：

```
[LibraryClasses]
  zlib|zlib-1.2.6/zlib.inf
```

调用 zlib 时，在需要链接 zlib 的工程模块的工程文件 [LibraryClasses] 中添加 zlib 即可。

```
[LibraryClasses]
  zlib
```

如果库使用之前需要进行初始化，在库的工程文件需指定 CONSTRUCTOR 和 DESTRUCTOR，CONSTRUCTOR 函数会加入到 ProcessLibraryConstructorList 中，这个 CONSTRUCTOR 函数会在 ENTRY_POINT 之前执行；DESTRUCTOR 函数会加入到 ProcessLibraryDestructorList 中，这个 DESTRUCTOR 就会在 ENTRY_POINT 之后执行。

例如，如果 zlib 库在被调用之前需在 InitializeLib() 中初始化，程序结束之前需调用 LibDestructor() 清理 zlib 库占用的资源，那么要在工程文件中做如下设置：

```
[Defines]
  …
  CONSTRUCTOR =InitializeLib
  DESTRUCTOR  =LibDestructor
```

然后还需在库的源文件中提供这两个函数，如示例 3-15 所示。

【示例 3-15】 zlib 库的构造函数和析构函数。

```
RETURN_STATUS EFI_API InitializeLib()
{
  EFI_STATUS Status;
    …// 初始化库
  return Status;
}

RETURN_STATUS EFI_API LibDestructor ()
{
  EFI_STATUS Status;
    …// 清理库所占资源
  return Status;
}
```

3.2.4　UEFI 驱动模块

在 UEFI 中，驱动分为两类：一类是符合 UEFI 驱动模型的驱动，模块类型为 UEFI_DRIVER，称为 UEFI 驱动；另一类是不遵循 UEFI 驱动模型的驱动，模块类型包括 DXE_DRIVER、DXE_SAL_DRIVER、DXE_SMM_DRIVER、DXE_RUNTIME_DRIVER，称为 DXE 驱动。

驱动与应用程序的模块入口函数（ENTRY_POINT）类型一样，函数原型如代码清单 3-9 所示。

代码清单 3-9　应用程序工程模块和驱动程序模块入口函数的函数原型

```
typedef EFI_STATUS API (*UEFI_ENTRYPOINT)(
  IN EFI_HANDLE ImageHandle,
  IN EFI_SYSTEM_TABLE *SystemTable);
```

驱动与应用程序的最大区别是驱动会常驻内存，而应用程序执行完毕后就会从内存中清除。

本书重点讲述第一类驱动。UEFI 驱动模型将在第 9 章详细讲述，本节主要讲述 UEFI 驱动模块工程文件的格式。

- 在 [Defines] 块，将 MODULE_TYPE 设置为 UEFI_DRIVER。其他宏变量如 INF_VERSION、BASE_NAME、FILE_GUID、VERSION_STRING 和 ENTRY_POINT，相信读者已经明白其含义和设置方法，在此不再赘述。

- 在 [Sources] 块，通常含有 ComponentName.c，在此文件中定义了驱动的名字，驱动安装后，这个名字将显示给用户。
- 在 [LibraryClasses] 块，必须包含 UefiDriverEntryPoint。

代码清单 3-10 是 DiskIo 驱动的工程文件。

代码清单 3-10　DiskIo 驱动的工程文件

```
//@file MdeModulePkg/Universal/Disk/DiskIoDxe/DiskIoDxe.inf
[Defines]
  INF_VERSION  = 0x00010005
  BASE_NAME = DiskIoDxe
  FILE_GUID  = 6B38F7B4-AD98-40e9-9093-ACA2B5A253C4
  MODULE_TYPE = UEFI_DRIVER
  VERSION_STRING = 1.0
  ENTRY_POINT = InitializeDiskIo
[Sources]
  ComponentName.c
  DiskIo.h
  DiskIo.c
[Packages]
  MdePkg/MdePkg.dec
[LibraryClasses]
  UefiDriverEntryPoint
  UefiBootServicesTableLib
  MemoryAllocationLib
  BaseMemoryLib
  BaseLib
  UefiLib
  DebugLib
[Protocols]
  gEfiDiskIoProtocolGuid                          ## BY_START
  gEfiBlockIoProtocolGuid                         ## TO_START
```

3.2.5　模块工程文件小结

前面我们讲述了模块的工程文件 .inf，现在我们知道，.inf 就像 Visual Studio 里的工程文件或者 Linux 里面的 Makefile 文件，用于帮助我们组织和编译工程。.inf 文件有 [Defines]、[Sources]、[Packages]、[LibraryClasses]、[Protocols]、[BuildOptions] 几个部分。

- [Defines] 部分定义了模块类型（MODULE_TYPE ）、模块的名字（BASE_NAME）、版本号（VERSION_STRING）、入口函数（ENTRY_POINT ）等。
- [Sources] 部分定义了本模块包含的源文件或目标文件。
- [Packages] 指明了要引用的包，包中的头文件可以在库的源文件中引用。
- [LibraryClasses] 列出了需要链接的库。

- [Protocols] 里列出了本模块用到的 Protocol。
- [BuildOptions] 列出了编译本模块中的源文件时用到的编译选项。

3.3　包及 .dsc、.dec、.fdf 文件

前面我们介绍了 .inf 文件，如果说 .inf 文件相当于 Visual studio 中的工程文件，.dsc（Platform Description File）则相当于 Visual studio 中的 solution 文件。每个包包含一个 .dec（Package Declaration File）文件、一个 .dsc 文件。如过一个包用于生成固件 Image 或 Option Rom Image，这个包还要包含 .fdf（Flash Description Files），.fdf 用于生成固件 Image、Option Rom Image 或可启动 Image。

- build 命令用于编译包，它需要一个 .dsc 文件、一个 .dec 文件以及一个或多个 .inf 文件。
- GenFW 命令用于制作固件或 Option Rom Image，它需要一个 .dec 文件和一个 .fdf 文件。

图 3-4 展示了 EDK2 文件与 EDK2 工具链命令之间的关系。

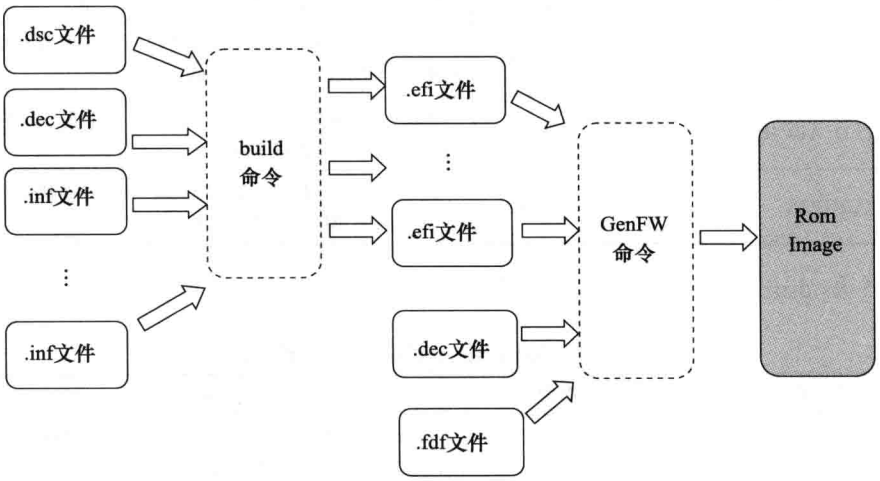


图 3-4　EDK2 文件与 EDK2 工具链关系图

下面讲述常用的两种文件：.dsc 与 .dec 文件。

3.3.1　.dsc 文件

.inf 用于编译一个模块，而 .dsc 文件用于编译一个 Package，它包含了 [Defines]、[LibraryClasses]、[Components] 几个必需部分以及 [PCD]、[BuildOptions] 等几个可选部分。

1. [Defines] 块

[Defines] 用于设置 build 相关的全局宏变量，这些变量可以被 .dsc 文件的其他模块引用。[Defines] 必须是 .dsc 文件的第一个部分，格式如下。

```
[Defines]
  宏变量名 = 值
  DEFINE 宏变量名 = 值
  EDK_GLOBAL 宏变量名 = 值
```

[Defines] 中通过 DEFINE 和 EDK_GLOBAL 定义的宏可以在 .dsc 文件和 .fdf 文件中通过 $（宏变量名）使用。表 3-4 列出了 .dsc 文件中 [Defines] 必须定义的宏变量。

表 3-4 .dsc 文件中 [Defines] 必须定义的宏变量

宏变量名	值类型	说　明
DSC_SPECIFICATION	数值	DSC Spec 1.22 对应的值为 0x00010006。目前（UEFI Spec 2.3/2.4）常用值为 0x00010005。DSC 必须保证向后兼容
PLATFORM_GUID	GUID	平台 GUID，每个 .dsc 文件必须有一个独一无二的 GUID
PLATFORM_VERSION	数值	.dsc 文件变化时，增加此数值
PLATFORM_NAME	标识符	标识符字符串中只能包含英文字符、数字、横线和下划线
SKUID_IDENTIFIER	标识符	该宏可以通过命令行在 build 时传入。可以是 Default。如果不是 Default，值必须是 [SkuIds] 中的一个
SUPPORTED_ARCHITECTURES	列表	通过“|”分隔的列表，该 .dsc 所支持的平台的体系结构。例如 IA32 或者 IA32|X64
BUILD_TARGETS	列表	通过“|”分隔的列表，该 .dsc 所支持的编译目标，例如 BUILD 或者 BUILD | RELEASE

表 3-5 是 .dsc 文件 [Defines] 可选宏变量。

表 3-5 .dsc 文件 [Defines] 可选宏变量

宏变量名	值类型	说　明
OUTPUT_DIRECTORY	路径	目标文件路径。可以是相对路径，也可以是绝对路径。默认为 $(WORKSPACE)/ Build/$(PLATFORM_NAME)
FLASH_DEFINITION	文件名	FDF 文件。可以是文件名，也可以是带路径的文件名。如果仅仅是文件名，则该文件必须在 .dsc 所在的目录
BUILD_NUMBER	最多 4 个字符	用于 Makefile 文件
FIX_LOAD_TOP_MEMORY_ADD RESS	地址	驱动、应用程序在内存中的起始地址
TIME_STAMP_FILE	文件名	时间戳文件。可以是文件名，也可以是带路径的文件名。该文件包含了一个时间戳，所有编译过程中生成的文件使用该文件戳

（续）

宏变量名	值类型	说　明
DEFINE	MACRO= PATH|Value	宏定义。定义的宏可以在 .dsc 文件中调用。例如， DEFINE DEBUG_ENABLE_OUTPUT = FALSE 在后续的 .dsc 文件中可以这样使用： !if $(DEBUG_ENABLE_OUTPUT) !endif
EDK_GLOBAL	MACRO= PATH|Value	仅用于 EDK 模块。EDK2 模块忽略该值
RFC_LANGUAGES	RFC 4646 语言代码列表	RFC 4646 语言代码字符串，各个语言代码之间用分号“;”分隔。用于在 AutoGen 阶段处理 Unicode 字符串。字符串必须用 " 包裹 "
ISO_LANGUAGES	ISO-639-2 语言代码列表	ISO-639-2 语言代码字符串，语言代码之间无任何分隔，每个语言代码为 3 字符。用于在 AutoGen 阶段处理 Unicode 字符串。字符串必须用 " 包裹 "
VPD_TOOL_GUID	GUID	在 AutoGen 阶段调用此 GUID 对应的 VPD 程序。VPD 程序定义在 Conf\tools_def.txt 文件中，默认为 BPDG
PCD_INFO_GENERATION	布尔值	TRUE 表示在 PCD 数据库中生存 PCD 信息

代码清单 3-11 是 Nt32Pkg 中 .dsc 文件的 [Defines] 部分。

代码清单 3-11　Nt32Pkg 中 .dsc 文件的 [Defines] 块

```
[Defines]
  PLATFORM_NAME                  = NT32
  PLATFORM_GUID                  = EB216561-961F-47EE-9EF9-CA426EF547C2
  PLATFORM_VERSION               = 0.4
  DSC_SPECIFICATION              = 0x00010005
  OUTPUT_DIRECTORY               = Build/NT32
  SUPPORTED_ARCHITECTURES        = IA32
  BUILD_TARGETS                  = DEBUG|RELEASE
  SKUID_IDENTIFIER               = DEFAULT
  FLASH_DEFINITION               = Nt32Pkg/Nt32Pkg.fdf
```

2. [LibraryClasses] 块

在 [LibraryClasses] 块中定义了库的名字以及库 .inf 文件的路径。这些库可以被 [Components] 块内的模块引用。

（1）语法

[LibraryClasses] 块语法如下：

```
[LibraryClasses.$(Arch).$(MODULE_TYPE)]
  LibraryName | Path/LibraryName.inf
```

或者

```
[LibraryClasses.$(Arch).$(MODULE_TYPE), LibraryClasses.$(Arch1).$(MODULE_TYPE1)]
  LibraryName | Path/LibraryName.inf
```

$(Arch) 和 $(MODULE_TYPE) 是可选项。逗号表示并列关系，块内的库对 Library Classes.$(Arch).$(MODULE_TYPE) 和 LibraryClasses.$(Arch1).$(MODULE_TYPE1) 都有效。

通常，.dsc 文件中都有 [LibraryClasses] 区块，表示块内定义的库对所有体系结构和所有类型的模块都有效。

$(Arch) 表示体系结构，可以是下列值之一：IA32、X64、IPF、EBC、ARM、common。common 表示对所有体系结构有效。

$(MODULE_TYPE) 表示模块的类别，块内列出的库只能供 $(MODULE_TYPE) 类别的模块链接。$(MODULE_TYPE) 可以是下列值：SEC、PEI_CORE、PEIM、DXE_CORE、DXE_SAL_DRIVER、BASE、DXE_SMM_DRIVER、DXE_DRIVER、DXE_RUNTIME_DRIVER、UEFI_DRIVER、UEFI_APPLICATION、USER_DEFINED。

（2）示例

例如，在 Nt32Pkg.dsc 中有：

```
[LibraryClasses.common.PEIM,LibraryClasses.common.PEI_CORE]
  HobLib|MdePkg/Library/PeiHobLib/PeiHobLib.inf
```

在 ShellPkg.dsc 中有：

```
[LibraryClasses.ARM]
  NULL|ArmPkg/Library/CompilerIntrinsicsLib/CompilerIntrinsicsLib.in
```

引用库是在 .inf 文件的 [LibraryClasses] 块中完成的。例如，本章第一个示例程序的 .inf 文件中引用了 UefiApplicationEntryPoint 和 UefiLib，如下所示：

```
[LibraryClasses]
  UefiApplicationEntryPoint
  UefiLib
```

3. [Components] 块

在该区块内定义的模块都会被 build 工具编译并生成 .efi 文件，格式如下：

```
[Components.$(Arch)]
  Path\Exectuables.inf
```

或者

```
[Components.$(Arch)]
  Path\Exectuables.inf{
<LibraryClasses> #嵌套块
   LibraryName|Path/LibraryName.inf
```

```
<BuildOptions>    # 嵌套块
    # 子块中还可以包含 <Pcds*>
  }
```

如果 Path 是相对路径，则相对路径起始于 $(WORKSPACE)，$(WORKSPACE) 通常是 EDK2 的根目录。在 Path 中可以使用通过 DEFINE 命令定义的宏。例如：

```
[Components]
  DEFINE MYSOURCE_PATH = D:/Source
  $(MYSOURCE_PATH)/Hello.inf # 相当于 D:/Source/Hello.inf
```

上述格式中的大括号内的内容仅对本模块有效，例如，示例 3-16 中 [Components] 声明的 DevicePathDxe.inf 会调用在 MdePkg/Library/UefiDevicePathLib/UefiDevicePathLib.inf 定义的 DevicePathLib 库，其他模块会使用在全局 [LibraryClasses] 块中声明的 UefiDevicePathLib DevicePathProtocol.inf 对应的 DevicePathLib 库。

【示例 3-16】 .dsc 文件中的嵌套块。

```
[LibraryClasses]
DevicePathLib|MdePkg/Library/UefiDevicePathLibDevicePathProtocol/UefiDevicePath
    LibDevicePathProtocol.inf
[Components]
  ...
  MdeModulePkg/Universal/DevicePathDxe/DevicePathDxe.inf {
    <LibraryClasses>
      DevicePathLib|MdePkg/Library/UefiDevicePathLib/UefiDevicePathLib.inf
  }
```

下面是在 MdeMoudlePkg.dsc 中的一个实例，这个块内的模块对 IA32、X64 和 IPF 体系结构有效。

```
[Components.IA32, Components.X64, Components.IPF]
  MdeModulePkg/Universal/Network/UefiPxeBcDxe/UefiPxeBcDxe.inf
  MdeModulePkg/Universal/DebugSupportDxe/DebugSupportDxe.inf
  MdeModulePkg/Universal/EbcDxe/EbcDxe.inf
```

4. [BuildOptions] 块

[BuildOptions] 格式在前面的 .inf 文件已经介绍过，.dsc 文件的 [BuildOptions] 与 .inf 文件的 [BuildOptions] 格式大致相同，区别在于 .dsc 文件的 [BuildOptions] 对 .dsc 文件内的所有模块有效。[BuildOptions] 格式如下：

```
[BuildOptions.$(Arch).$(CodeBase)]
  [编译器]:[$(Target)]_[Tool]_[$(Arch)]_[CC|DLINK]_FLAGS=
```

$(Arch) 与 $(CodeBase) 都是可选项。$(CodeBase) 是 EDK 和 EDK2 之一。$(Arch) 是体

系结构，如 IA32、X64 等。

$(Target) 是 DEBUG、RELEASE、* 中的一个。Tool 是编译工具的名字，如 VS2012、GCC44 等。CC 表示编译选项，DLINK 表示连接选项。

代码清单 3-12 是 AppPkg.dsc 中的 [BuildOptions] 块内容。

代码清单 3-12　AppPkg.dsc 中的 [BuildOptions] 块

```
[BuildOptions]
!ifndef $(EMULATE)
  INTEL:*_*_*_CC_FLAGS       = /Qfreestanding /D UEFI_C_SOURCE
   MSFT:*_*_*_CC_FLAGS       = /X /Zc:wchar_t /D UEFI_C_SOURCE
   ...
!else
  INTEL:*_*_IA32_CC_FLAGS      = /Od /D UEFI_C_SOURCE
   MSFT:*_*_IA32_CC_FLAGS      = /Od /D UEFI_C_SOURCE
   ...
!endif
```

在 .dsc 文件中可以使用 ! 命令。!include 用于加载其他文件。!if、!ifdef、!ifndef、!else、!end 是条件处理语句。

最后简单介绍一下 [PCD] 块。[PCD] 块用于定义平台配置数据。其目的是在不改动 .inf 文件和源文件的情况下完成对平台的配置。例如在 UEFI 模拟器 Nt32Pkg 的 .dsc 文件 Nt32Pkg.dsc 中，可以通过 PCD 的 PcdWinNtFileSystem 配置模拟器文件系统路径，如下所示：

```
gEfiNt32PkgTokenSpaceGuid.PcdWinNtFileSystem|L".!..\..\..\..\EdkShellBinPkg\
    Bin\Ia32\Apps"|VOID*|106
```

该项配置被竖线“|”分为 4 个部分。第一部分中 gEfiNt32PkgTokenSpaceGuid 是名字空间，PcdWinNtFileSystem 是变量名。第二部分是值。第三部分是变量类型。第四部分是变量数据的最大长度。

在源文件中可以通过 LibPcdGetPtr(_PCD_TOKEN_PcdWinNtFileSystem) 获得 gEfiNt32Pkg TokenSpaceGuid.PcdWinNtFileSystem 定义的值。

3.3.2　.dec 文件

.dec 文件定义了公开的数据和接口，供其他模块使用。它包含了必需区块 [Defines] 以及可选区块 [Includes]、[LibraryClasses]、[Guids]、[Protocols]、[Ppis] 和 [PCD] 几个部分。

1. [Defines] 块

[Defines] 区块用于提供 package 的名称、GUID、版本号等信息，格式如下：

```
[Defines]
  Name = Value
```

Name 可以是下面 4 个：DEC_SPECIFICATION、PACKAGE_NAME、PACKAGE_GUID、PACKAGE_VERSION。

例如，MdePkg.dsc 中的 [Defines] 部分如下所示：

```
[Defines]
  DEC_SPECIFICATION  = 0x00010005
  PACKAGE_NAME = MdePkg
  PACKAGE_GUID = 1E73767F-8F52-4603-AEB4-F29B510B6766
  PACKAGE_VERSION  = 1.03
```

2. [Includes] 块

[Includes] 中列出了本 Package 提供的头文件所在的目录，格式如下：

```
[Includes.$(Arch)]
  Path
```

Path 只能是相对路径，该相对路径起始于本 Package 的 .dsc 所在的目录。

例如，MdePkg.dec 文件中的 [Includes] 部分如下所示：

```
[Includes]
  Include
[Includes.IA32]                          # 编译 32 位程序时的头文件路径
  Include/Ia32
[Includes.X64]                           # 编译 x86_64 位程序时的头文件路径
  Include/X64
```

3. [LibraryClasses] 块

Package 可以通过 .dec 文件对外提供库，每个库都必须有一个头文件，放在 Include\Library 目录下。本区块用于明确库和头文件的对应关系。其格式如下：

```
[LibraryClasses.$(Arch)]
  LibraryName | Path/LibraryHeader.h
```

例如，下面的代码是 MdePkg.dec 中的 [LibraryClasses] 的一部分。

```
 [LibraryClasses]
  UefiUsbLib|Include/Library/UefiUsbLib.h
...
[LibraryClasses.IA32, LibraryClasses.X64]
  SmmLib|Include/Library/SmmLib.h
```

4. [Guids] 块

在 Package\Include\Guid 目录中有很多文件，每个文件内定义了一个或几个 GUID，例如

在 MdePkg\Include\Gpt.h 文件中定义了 Gpt 分区相关的 GUID，如下所示：

```
extern EFI_GUID gEfiPartTypeUnusedGuid;
extern EFI_GUID gEfiPartTypeSystemPartGuid;
extern EFI_GUID gEfiPartTypeLegacyMbrGuid
```

可以看出这些定义仅仅是声明。那么真正的常量定义在什么地方呢？真正的常量定义在 AutoGen.c 中，其值定义在 .dec 文件的 [Guids] 区块。其格式为：

```
[Guids.$(Arch)]
  GUIDName = GUID
```

回到前面讲的 Gpt 相关的 GUID，在 MdePkg.dec 中，这些 GUID 定义如下：

```
[Guids]
...
## Include/Guid/Gpt.h
  gEfiPartTypeLegacyMbrGuid  = { 0x024DEE41, 0x33E7, 0x11D3, { 0x9D, 0x69,
      0x00, 0x08, 0xC7, 0x81, 0xF3, 0x9F }}
gEfiPartTypeSystemPartGuid = { 0xC12A7328, 0xF81F, 0x11D2, { 0xBA, 0x4B, 0x00,
      0xA0, 0xC9, 0x3E, 0xC9, 0x3B }}
  gEfiPartTypeUnusedGuid = { 0x00000000, 0x0000, 0x0000, { 0x00, 0x00, 0x00,
      0x00, 0x00, 0x00, 0x00, 0x00 }}
```

当在模块工程文件的 [Guids] 中引用这些 Guid 时，这些值就会复制到 AutoGen.c 中。

5. [Protocols] 块

与 Guids 类似，在 Package\Include\Protocols 目录下有很多头文件，每个头文件定义了一个或多个 Protocol，这些 Protocol 的 GUID 值就定义在 .dec 文件的 [Protocols] 区块，格式如下：

```
[Protocols.$(Arch)]
  ProtocolName = GUID
```

例如，在 MdePkg\Include\Protocol 目录下的 BlockIo.h 定义了 BlockIo Protocol。

```
extern EFI_GUID gEfiBlockIoProtocolGuid;
```

gEfiBlockIoProtocolGuid 的值就定义在 MdePkg.dec 的 [Protocols] 块内。

```
[Protocols]
  gEfiBlockIoProtocolGuid = { 0x964E5B21, 0x6459, 0x11D2, { 0x8E, 0x39, 0x00,
      0xA0, 0xC9, 0x69, 0x72, 0x3B }}
```

下面简单说明 [Ppis]、[PCD] 和 [UserExtensions]。[Ppis] 用于定义源文件中用到的 PPI（回忆一下，PPI 是 PEI 阶段 PEI 模块之间通信的接口），语法类似于 [Protocols]。[PCD] 块是 .dsc 文件中 [PCD] 块的补充。

3.4　调试 UEFI

UEFI 有两种调试方式，一种是在模拟环境 Nt32Pkg 下调试，另一种是通过串口调试真实环境中的 UEFI 程序。下面介绍在 Nt32Pkg 下的调试。

在需要调试的代码前面加入“ _asm int 3”后编译，然后在模拟环境（Nt32Pkg）中运行该程序，当模拟器执行到“int 3”指令时，会弹出对话框，然后就可以调试了。

下面以一个标准应用程序工程模块为例演示如何调试。示例 3-17 是这个模块的源文件，详细工程文件和代码在本书附带的代码的 book\infs\Debug 目录中。该示例中，打印语句前加入了 _asm int 3 指令。

【示例 3-17】 在源文件中加入调试代码。

```
#include <Uefi.h>
EFI_STATUS
UefiMain (IN EFI_HANDLE ImageHandle,IN EFI_SYSTEM_TABLE  *SystemTable)
{
    _asm int 3
    SystemTable -> ConOut->OutputString(SystemTable->ConOut,L"HelloWorld\n");
    return EFI_SUCCESS;
}
```

需要注意的是，使用 _asm int 3 只能调试 32 位的应用程序或驱动。

另一个要注意的地方是，Visual C 编译时默认打开了优化选项，优化后“ _asm int 3”的位置可能发生变动，导致无法进入正确的调试位置。如果出现这种情况，需要设置 .inf 文件中的 [BuildOptions]，关闭优化选项。设置方法如下：

```
[BuildOptions]
MSFT:DEBUG_*_IA32_CC_FLAGS = /Od
```

示例 3-18 是这个模块的工程文件，在该工程文件中优化选项被关闭。

【示例 3-18】 关闭优化选项的工程文件。

```
[Defines]
  INF_VERSION = 0x00010006
  BASE_NAME = UefiMain
  FILE_GUID = 4ea97c46-7491-4dfd-b442-747010f3ce5f
  MODULE_TYPE = UEFI_APPLICATION
  VERSION_STRING  = 0.1
  ENTRY_POINT = UefiMain
[Sources]
  Main.c
[Packages]
  MdePkg/MdePkg.dec
[LibraryClasses]
```

```
  UefiApplicationEntryPoint
  UefiLib
[BuildOptions]
  MSFT:DEBUG_*_IA32_CC_FLAGS = /Od
```

编译后生成 DebugMain.efi 文件。按如下步骤进入调试 DebugMain.efi 的调试界面。

1）首先启动 UEFI 模拟器，进入 UEFI Shell，然后在 UEFI Shell 中执行 DebugMain.efi 程序，如图 3-5 所示。

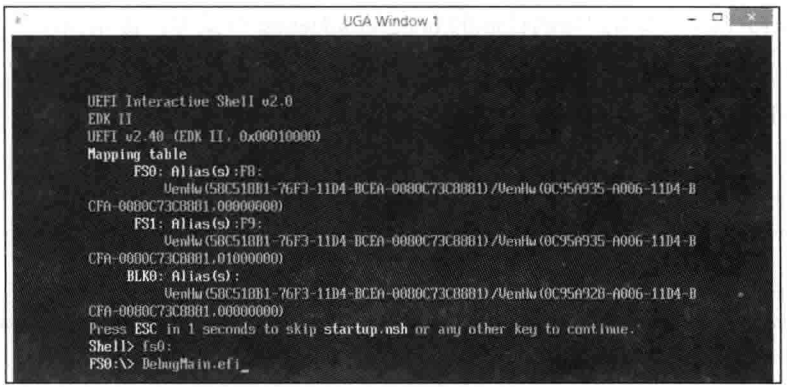


图 3-5　在 UEFI Shell 中执行 DebugMain.efi 程序

2）执行 DebugMain.efi 程序后，遇到“int 3”指令后会弹出系统调试对话框，如图 3-6 所示。

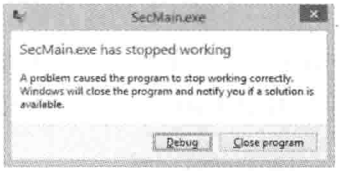


图 3-6　程序遇到“int 3”指令后弹出的系统调试对话框

3）单击调试对话框的“Debug”按钮，进入 Visual Studio 调试器，如图 3-7 所示。

4）单击 Visual Studio 调试器的“Yes”按钮，进入 Visual Studio 调试准备界面，如图 3-8 所示。

图 3-7　Visual Studio 调试器

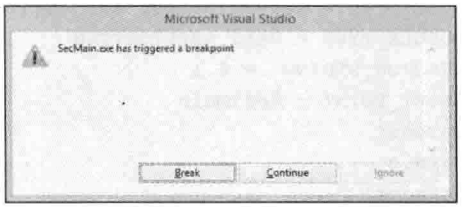


图 3-8　Visual Studio 调试准备界面

5）单击 Visual Studio 调试准备界面中的“ Break”按钮，进入 Visual Studio 调试界面，如图 3-9 所示。

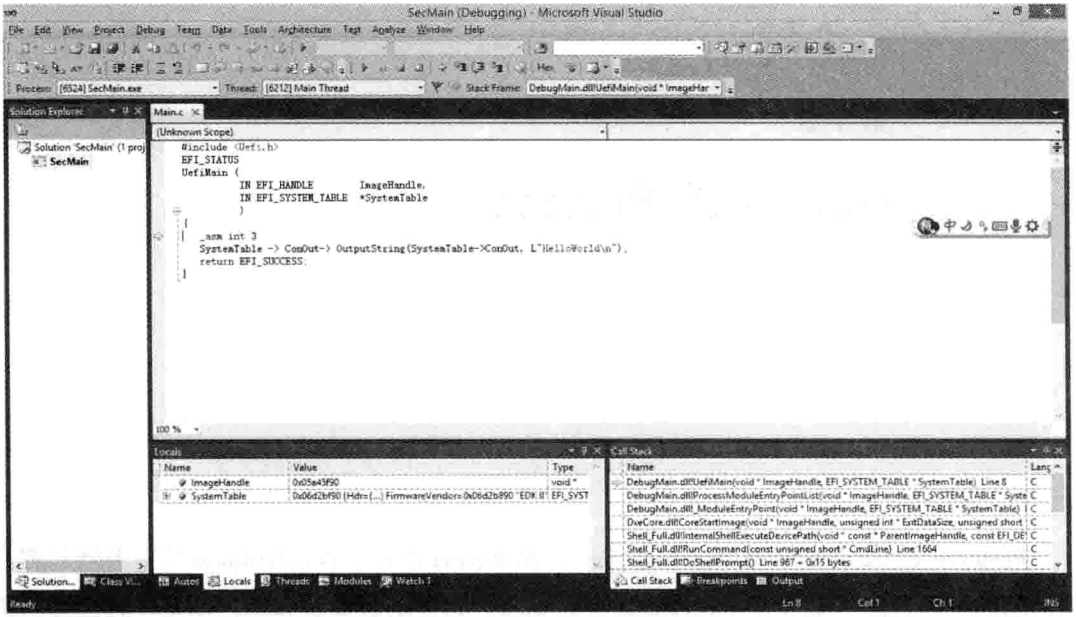


图 3-9　Visual Studio 调试界面

然后就可以利用 Visual Studio 调试器调试 DebugMain.efi 程序了。

3.5　本章小结

本章讲述了 .dsc、.dec 和 .inf 文件的格式。.dsc 文件相当于 Visual Studio 的项目文件，用于指导编译工具编译整个包；.inf 文件相当于 Visual Studio 的工程文件，用于指导编译工具编译这个模块；.dec 文件则用于向其他工程模块提供本 Package 的资源。

现在我们已经“扫除”了编译 UEFI 应用程序的所有障碍。

在下一章，将讲述 UEFI 开发中一定会遇到的系统服务。

Chapter 4 第 4 章

UEFI 中的 Protocol

Protocol 是 UEFI 中最重要的概念之一。从字面意思上理解，Protocol 是服务器和客户端之间的一种约定，双方根据这种约定互通信息。这里的服务器和客户端是一种广义的称呼，提供服务的称为服务器，使用服务的称为客户端。TCP 是一种 Protocol，客户端（应用程序）通过一组函数来压缩包和解压包，压缩包和解压包是服务器提供的服务。COM 也是一种 Protocol，客户端通过 CoCreateInstance(...) 和 GUID 获得指向 COM 对象的指针，然后使用该指针获得 COM 对象提供的服务，GUID 标示了这个 COM 对象。现在我们对 Protocol 有了概念上的理解，那么具体到 UEFI 里，Protocol 是什么样子呢？如何得到 Protocol 对应的对象？

在讲 Protocol 的结构之前，还要先介绍一下 C 语言与 C++ 语言的区别。UEFI 是用 C 语言来实现的，而 C 语言是面向过程的一种语言。完全使用面向过程的思想来管理和使用众多的 UEFI Protocol 会使程序变得非常复杂。Protocol 作为一种对象来设计管理会比较直观。因而 UEFI 中的 Protocol 引入了面向对象的思想：

- 用 struct 来模拟 class。
- 用函数指针（Protocol 的成员变量）模拟成员函数，此种函数的第一参数必须是指向 Protocol 的指针，用来模拟 this 指针。

下面以 EFI_BLOCK_IO_PROTOCOL（简称 BlockIo）为例介绍 Protocol 的结构，其数据结构如代码清单 4-1 所示。

代码清单 4-1　EFI_BLOCK_IO_PROTOCOL 数据结构

```
//@file MdePkg/Include/Protocol/BlockIo.h

//通过这个 Protocol 可以控制块设备
struct _EFI_BLOCK_IO_PROTOCOL {
//Protocol 版本号，Protocol 必须保证向后兼容
//如果没有向后兼容，必须给未来的版本定义不同的 GUID，也就是必须定义一个不同的 Protocol
  UINT64                Revision;
  EFI_BLOCK_IO_MEDIA    *Media;
  EFI_BLOCK_RESET       Reset;
  EFI_BLOCK_READ        ReadBlocks;
  EFI_BLOCK_WRITE       WriteBlocks;
  EFI_BLOCK_FLUSH       FlushBlocks;
};
extern EFI_GUID gEfiBlockIoProtocolGuid;
```

每个 Protocol 必须有一个唯一的 GUID，例如在 BlockIo.h 中定义了 BlockIo 的 GUID，如下所示：

```
#define EFI_BLOCK_IO_PROTOCOL_GUID {  0x964e5b21, 0x6459, 0x11d2, {0x8e, 0x39,
    0x0, 0xa0, 0xc9, 0x69, 0x72, 0x3b }}
typedef struct _EFI_BLOCK_IO_PROTOCOL  EFI_BLOCK_IO_PROTOCOL;
```

结构体 EFI_BLOCK_IO_PROTOCOL 有两个成员变量和 4 个成员函数（当然，从 C 语言的角度来看，“成员函数”这样的叫法不准确，它实际上也是一个成员变量，只是这个变量是函数指针而已）。gEfiBlockIoProtocolGuid(｛0x964e5b21，0x6459,0x11d2,{0x8e,0x39,0x0,0xa0,0xc9,0x69,0x72,0x3b}｝) 是一种标示符，标示了 EFI_BLOCK_IO_PROTOCOL。

以 ReadBlocks 为例来看成员函数的声明，如代码清单 4-2 所示。

代码清单 4-2　EFI_BLOCK_IO_PROTOCOL 的 ReadBlocks 服务的函数原型

```
/**  从地址 Lba 开始的块读取 BufferSize 字节到缓冲区
  @retval EFI_SUCCESS             数据从设备正确读出
  @retval EFI_DEVICE_ERROR        设备出现错误
  @retval EFI_NO_MEDIA            设备中没有介质
  @retval EFI_MEDIA_CHANGED       MediaId 与当前设备不符
  @retval EFI_BAD_BUFFER_SIZE     缓冲区大小不是块的整数倍
  @retval EFI_INVALID_PARAMETER 要读取的块中包含无效块；或缓冲区未对齐
**/
typedef EFI_STATUS(EFIAPI *EFI_BLOCK_READ)(
  IN EFI_BLOCK_IO_PROTOCOL *This,        //This 指针，指向调用上下文
  IN UINT32  MediaId,                    //media Id
  IN EFI_LBA Lba,                        //要读取的启始块逻辑地址
  IN UINTN BufferSize,                   //要读取的字节数，必须是块大小的整数倍
```

```
  OUT VOID *Buffer                    // 目的缓冲区，调用者负责该缓冲区的创建与删除
);
```

EFI_BLOCK_READ 具体用法先不讲，先来看它的第一个参数，指向 EFI_BLOCK_IO_PROTOCOL 对象自己的 This 指针，这是成员函数区别于一般函数的重要特征。通常，计算机中有许多不同的块设备，每个块设备都有一个 EFI_BLOCK_IO_PROTOCOL 的实例，This 指针就是指向这个实例，用于告诉成员函数我们正在操作哪个设备。This 指针是 Protocol 成员函数的一个重要特征，与 C++ 成员函数 this 指针的区别是，C++ 的 this 指针由编译器自动加入，而 Protocol 成员函数的 This 指针需手工添加。

4.1 Protocol 在 UEFI 内核中的表示

使用 Protocol 之前，要先弄清楚 Protocol 位于什么地方。首先我们要来认识一下 EFI_HANDLE。

```
typedef VOID *EFI_HANDLE;
```

EFI_HANDLE 是指向某种对象的指针，UEFI 用它来表示某个对象。UEFI 扫描总线后，会为每个设备建立一个 Controller 对象，用于控制设备，所有该设备的驱动以 Protocol 的形式安装到这个 Controller 中，这个 Controller 就是一个 EFI_HANDLE 对象。当我们将一个 .efi 文件加载到内存中时，UEFI 也会为该文件建立一个 Image 对象（此 Image 非图像的意思），这个 Image 对象也是一个 EFI_HANDLE 对象。在 UEFI 内部，EFI_HANDLE 被理解为 IHANDLE，IHANDLE 的数据结构如代码清单 4-3 所示。

代码清单 4-3　IHANDLE 数据结构

```
/// IHANDLE - 包含了 Protocols 链表
typedef struct {
  UINTN Signature;                    // 表明 Handle 的类别
  LIST_ENTRY AllHandles;              /// 所有 IHANDLE 组成的链表
  LIST_ENTRY Protocols;               /// 此 Handle 的 Protocols 链表
  UINTN LocateRequest;
  UINT64 Key;
} IHANDLE;
```

每个 IHANDLE 中都有一个 Protocols 链表，存放属于自己的 Protocol。所有的 IHANDLE 通过 AllHandles 链接起来。图 4-1 展示了 IHANDLE 内的 Protocol 是如何被组织起来的。IHANDLE 的 Protocols 是一个双向链表，链表中每一个元素是 PROTOCOL_INTERFACE，通过 PROTOCOL_INTERFACE 的 Protocol 指针可以得到这个 Protocol 的 GUID，通过 Interface 指针可以得到这个 Protocol 的实例。

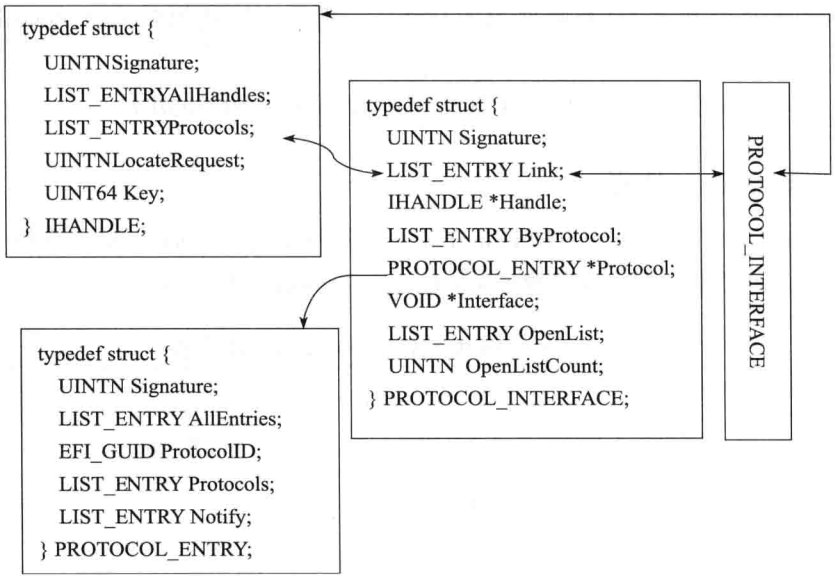


图 4-1　IHANDLE 内的 Protocol 组织图

4.2　如何使用 Protocol 服务

启动服务提供了丰富的服务供开发者操作 Protocol，操作分两种，一种是使用 Protocol，另一种是产生 Protocol。本章只讲述使用 Protocol 的操作，如何产生 Protocol 将在第 8 章和第 9 章讲述。第 8 章会讲述如何产生服务型 Protocol，第 9 章会讲述如何产生驱动型 Protocol。

要使用 Protocol，首先要根据 GUID 找到 Protocol 对象，启动服务提供了 OpenProtocol(...)、HandleProtocl(...) 和 LocateProtocol(...) 三种服务用于找出指定的 Protocol。OpenProtocol 用于打开指定句柄上的 Protocol。HandleProtocol 是 OpenProtocol 的简化版。LocateProtocol 用于找出指定的 Potocol 在系统中的第一个实例。使用完 Protocol 后还要关闭打开的 Protocol，否则可能会造成内存泄漏。

除了打开和关闭 Protocol，有时还需要找出能支持某个 Protocol 的所有设备。例如，要找出支持 BlockIo 的所有设备（即找出所有块设备），这时就要用到启动服务提供的 LocalHandleBuffer 服务。

使用 Protocol 时，一般需要经过以下三个步骤。

第一步：通过 gBS->OpenProtocol（或 HandleProtocol、LocateProtocol）找出 Protocol 对象。

第二步：使用这个 Protocol 提供的服务。

第三步：通过 gBS->CloseProtocol 关闭打开的 Protocol。

如果想知道某个 Protocol 被哪些设备打开了，那么可以使用 OpenProtocolInformation 服务。

下面详细讲述 Boot Service（BS）中这几个 Protocol 相关服务的用法。

4.2.1 OpenProtocol 服务

OpenProtocol 用于查询指定的 Handle 中是否支持指定的 Protocol，如果支持，则打开该 Protocol，否则返回错误代码。代码清单 4-4 是 OpenProtocol 服务的函数原型。

代码清单 4-4 BS 中的 OpenProtocol 函数原型

```
/** gBS->OpenProtocol
  打开 Protocol
  @retval EFI_SUCCESS              成功打开 Protocol，打开信息添加到 Protocol 的打开列表中
  @retval EFI_UNSUPPORTED          Handle 不支持 Protocol
  @retval EFI_INVALID_PARAMETER    无效参数
  @retval EFI_ACCESS_DENIED        Attribute 不被当前环境支持
  @retval EFI_ALREADY_STARTED      Protocol 已经被同一 AgentHandle 打开
**/
typedef EFI_STATUS (EFIAPI *EFI_OPEN_PROTOCOL)(
  IN  EFI_HANDLE Handle,           // 指定的 Handle，将查询并打开此 Handle 中安装的 Protocol
  IN  EFI_GUID *Protocol,          // 要打开的 Protocol（指向该 Protocol GUID 的指针）
  OUT VOID **Interface, OPTIONAL          // 返回打开的 Protocol 对象
  IN  EFI_HANDLE AgentHandle,             // 打开此 Protocol 的 Image
  IN  EFI_HANDLE ControllerHandle,        // 使用此 Protocol 的控制器
  IN  UINT32 Attributes                   // 打开 Protocol 的方式
  );
```

Handle 是 Protocol 的提供者，如果 Handle 的 Protocols 链表中有该 Protocol，则 Protocol 对象的指针写到 *Interface（例如，打开 BlockIo 时，成功返回后，*Interface 将指向 EFI_BLOCK_IO_PROTOCOL 对象），并返回 EFI_SUCCESS；否则返回 EFI_UNSUPPORTED。

如果在驱动中调用 OpenProtocol()，则 ControllerHandle 是拥有该驱动的控制器，也就是请求使用这个 Protocol 的控制器；AgentHandle 是拥有该 EFI_DRIVER_BINDING_PROTOCOL 对象的 Handle。EFI_DRIVER_BINDING_PROTOCOL 是 UEFI 驱动开发中一定会用到的一个 Protocol，它负责驱动的安装与卸载。UEFI 驱动模型及驱动开发将在第 9 章讲解。

如果调用 OpenProtocol 的是应用程序，那么 AgentHandle 是该应用程序的 Handle，也就是 UefiMain 函数的第一个参数，ControllerHandle 此时可以忽略。

Attributes 可以取以下 6 种值。

```
#define EFI_OPEN_PROTOCOL_BY_HANDLE_PROTOCOL    0x00000001
#define EFI_OPEN_PROTOCOL_GET_PROTOCOL          0x00000002
#define EFI_OPEN_PROTOCOL_TEST_PROTOCOL         0x00000004
#define EFI_OPEN_PROTOCOL_BY_CHILD_CONTROLLER   0x00000008
```

```
// 如果已经打开，则被同一个控制器再次打开时将失败
#define EFI_OPEN_PROTOCOL_BY_DRIVER            0x00000010
// 如果 Protocol 已经打开，则再次打开时将失败
#define EFI_OPEN_PROTOCOL_EXCLUSIVE            0x00000020
```

示例 4-1 改编自 Spec 2.3.1 第 165 页中的示例程序，展示了打开 BlockIo 的方法。

【示例 4-1】 打开 BlockIo。

```
externEFI_BOOT_SERVICES *gBS;
EFI_HANDLE ImageHandle;
EFI_DRIVER_BINDING_PROTOCOL *This;
IN EFI_HANDLE ControllerHandle,
extern EFI_GUID gEfiBlockIoProtocolGuid;
EFI_BLOCK_IO_PROTOCOL  *BlockIo;
EFI_STATUS Status;
// 在应用程序中使用 OpenProtocol，一般 ControllerHandle 会设为 NULL
Status = gBS->OpenProtocol (
  ControllerHandle,                          // Handle
  &gEfiBlockIoProtocolGuid,                  // Protocol
  &BlockIo,                                  // Interface
  ImageHandle,                               // AgentHandle
  NULL,                                      // ControllerHandle
  EFI_OPEN_PROTOCOL_BY_HANDLE_PROTOCOL // Attributes
);
// 在驱动中使用 OpenProtocol，ControllerHandle 是控制器的 Handle
Status = gBS->OpenProtocol (ControllerHandle, &gEfiBlockIoProtocolGuid,
                            &BlockIo, This->DriverBindingHandle,ControllerHandle,
                            EFI_OPEN_PROTOCOL_GET_PROTOCOL);
```

示例 4-1 中，gEfiBlockIoProtocolGuid 是全局变量，变量定义在 AutoGen.c 中。AutoGen.c 是由 build 工具根据 .inf 和 .dec 文件生成的。gEfiBlockIoProtocolGuid 是 EFI_GUID 类型的变量，其 EFI_GUID 值定义在 .dec 中。如果要在应用程序或驱动中使用 gEfiBlockIoProtocolGuid，那么必须在 .inf 文件的 [Protocols] 中声明以便预处理时将其包含在 AutoGen.c 中。

4.2.2　HandleProtocol 服务

OpenProtocol 功能很强大，但使用起来也比较复杂，除了要提供 Handle 和 Protocol 的 GUID，还要提供 AgentHandle、ControllerHandle 和 Attributes。大部分情况下，开发者只是想通过 Protocol 的 GUID 得到 Protocol 对象，并不关心打开方式的细节。为了方便开发者使用 Protocol，启动服务提供了 HandleProtocol 以简化打开 Protocol。

代码清单 4-5 是 HandleProtocol 服务的函数原型。

代码清单 4-5　BS 的 HandleProtocol 服务的函数原型

```
/**gBS->HandleProtocol
查询指定的 Handle 中是否支持指定的 Protocol。如果支持，打开该 Protocol
```

```
  @retval EFI_SUCCESS             成功打开指定的 Protocol
  @retval EFI_UNSUPPORTED         指定的设备 (Handle) 不支持该 Protocol
  @retval EFI_INVALID_PARAMETER   参数 Handle 、Protocol 或 Interface 是 NULL
**/
typedef EFI_STATUS(EFIAPI *EFI_HANDLE_PROTOCOL)(
  IN  EFI_HANDLE  Handle,         //查询该 Handle 是否支持 Protocol
  IN  EFI_GUID  *Protocol,        //带查询的 Protocol
  OUT VOID **Interface            //返回待查询的 Protocol
);
```

HandleProtocol 是 OpenProtocol 的简化版，在调用 HandleProtocol 时不必再传入 AgentHandle、ControllerHandle 和 Attributes 的值。AgentHandle 使用 gDxeCoreImageHandle，ControllerHandle 使用 NULL 值，Attributes 则使用 EFI_OPEN_PROTOCOL_BY_HANDLE_PROTOCOL。代码清单 4-6 展示了 HandleProtocol 是如何实现的。

代码清单 4-6 HandleProtocol 的实现

```
EFI_STATUS EFIAPI CoreHandleProtocol (
  IN EFI_HANDLE       UserHandle,
  IN EFI_GUID         *Protocol,
  OUT VOID            **Interface
  )
{
  return CoreOpenProtocol (
          UserHandle,
          Protocol,
          Interface,
          gDxeCoreImageHandle,
          NULL,
          EFI_OPEN_PROTOCOL_BY_HANDLE_PROTOCOL
          );
}
```

对于如下打开 BlockIo 的示例（见示例 4-1），用示例 4-2 可以实现与之完全相同的功能。

```
Status = gBS->OpenProtocol (
  ControllerHandle,
  &gEfiBlockIoProtocolGuid,
  &BlockIo,
  ImageHandle,
  NULL,
  EFI_OPEN_PROTOCOL_BY_HANDLE_PROTOCOL
);
```

【示例 4-2】 用 HandleProtocl 得到 BlockIo 的示例。

```
Status  = gBS->HandleProtocol (
```

```
  ControllerHandle,
  &gEfiBlockIoProtocolGuid,
  &BlockIo);
```

4.2.3　LocateProtocol 服务

OpenProtocol 和 HandleProtocol 用于打开指定设备上的某个 Protocol，要使用这两个函数，首先要得到这个设备的句柄，而找出这个设备的句柄并不是很容易。有时开发者并不关心 Protocol 在哪个设备上，尤其是系统仅有一个该 Protocol 的实例时。启动服务提供了 LocateProtocol(...) 服务，它可以从 UEFI 内核中找出指定 Protocol 的第一个实例。例如，通常在系统中仅有一个 FAT 分区，因而只有一个 EFI_SIMPLE_FILE_SYSTEM_PROTOCOL，这时可以 LocateProtocol 准确地找出这个 Protocol。代码清单 4-7 是 LocateProtocol 服务的函数原型。

代码清单 4-7　BS 的 LocateProtocol 服务的函数原型

```
/** gBS->LocateProtocol
  从 UEFI 内核中找出匹配 Protocol 和 Registration 的第一个实例
  @retval EFI_SUCCESS                    成功找到匹配的 Protocol
  @retval EFI_NOT_FOUND                  系统中无法找到匹配的 Protocol
  @retval EFI_INVALID_PARAMETER Interface 为空
**/
typedef EFI_STATUS(EFIAPI *EFI_LOCATE_PROTOCOL)(
  IN EFI_GUID *Protocol,          // 待查询的 Protocol
  IN VOID *Registration, OPTIONAL        // 可选参数，从 RegisterProtocolNotify() 获得的 Key
  OUT VOID **Interface                   // 返回系统中第一个匹配到的 Protocol 实例
  );
```

UEFI 内核中某个 Protocol 的实例可能不止一个，例如，每个硬盘及每个分区都有一个 EFI_DISK_IO_PROTOCOL 实例。LocateProtocol 顺序搜索 HANDLE 链表，返回找到的第一个该 Protocol 的实例。

下面的例子用于找出系统中的第一个 EFI_SIMPLE_FILE_SYSTEM_PROTOCOL 实例。

```
EFI_SIMPLE_FILE_SYSTEM_PROTOCOL* SimpleFs;
gBS->LocateProtocol( gEfiSimpleFileSystemProtocolGuid,  NULL, &SimpleFs);
```

4.2.4　LocateHandleBuffer 服务

前面介绍了从设备上找出 Protocol 的方法。有时候开发者需要找出支持某个 Protocol 的所有设备，例如找出系统中的所有安装了 BlockIo 的设备，LocateHandleBuffer 和 LocateHandle 服务提供了这个功能。代码清单 4-8 是 LocateHandleBuffer 服务的函数原型。

代码清单 4-8 BS 的 LocateHandleBuffer 服务的函数原型

```
/** gBS->LocateHandleBuffer
获得所有支持指定 Protocol 的 HANDLE。Buffer 数组由系统负责分配，由调用者负责释放
  @retval EFI_SUCCESS      成功返回。Buffer 返回 Handle 数组，NoHandles 返回 Handle 数目
  @retval EFI_NOT_FOUND    系统中没有发现支持指定 Protocol 的 Handle
  @retval EFI_OUT_OF_RESOURCES   资源耗尽
  @retval EFI_INVALID_PARAMETER NoHandles 或 Buffer 参数非法
**/
typedef EFI_STATUS(EFIAPI *EFI_LOCATE_HANDLE_BUFFER)(
  IN EFI_LOCATE_SEARCH_TYPE SearchType,          // 查找方式
  IN EFI_GUID *Protocol OPTIONAL,                // 指定的 Protocol
  IN VOID *SearchKey OPTIONAL,                   // PROTOCOL_NOTIFY 类型
  IN OUT UINTN *NoHandles,                       // 返回找到的 Handle 数量
  OUT EFI_HANDLE **Buffer                        // 分配 Handle 数组并返回
);
```

SearchType 有三种：AllHandles 用于找出系统中的所有 Handle；ByRegisterNotify 用于从 RegisterProtocolNotify 中找出匹配 SearchKey 的 Handle；ByProtocol 用于从系统 Handle 数据库中找出支持指定 Protocol 的 HANDLE。

SearchType 的类型定义如下所示：

```
typedef enum {
  AllHandles,
  ByRegisterNotify,
  ByProtocol
} EFI_LOCATE_SEARCH_TYPE;
```

代码清单 4-9 是 LocateHandle 服务的函数原型。LocateHandle 服务与 LocateHandleBuffer 的最大区别是需有调用者负责管理 Buffer 数组占用的内存。

代码清单 4-9 BS 的 LocateHandle 服务的函数原型

```
/** gBS->LocateHandle
获得所有支持指定 Protocol 的 HANDLE。调用者负责分配和释放 Buffer 数组。若调用者提供的 Buffer
太小，则返回 EFI_BUFFER_TOO_SMALL
**/
typedef EFI_STATUS(EFIAPI *EFI_LOCATE_HANDLE) (
  IN EFI_LOCATE_SEARCH_TYPE SearchType,          // 同 LocateHandleBuffer
  IN EFI_GUID *Protocol OPTIONAL,                // 同 LocateHandleBuffer
  IN VOID *SearchKey OPTIONAL,                   // 同 LocateHandleBuffer
  IN OUT UINTN *BufferSize,                      // 返回找到的 Handle 数量
  OUT EFI_HANDLE *Buffer                         // 返回找到的 Handle
);
```

示例 4-3 演示了如何通过 LocateHandleBuffer 找出系统中的所有可启动分区。

【示例 4-3】 通过 LocateHandleBuffer 找到可启动分区。

```
EFI_STATUS  DetectBootablePartition(void* Bootable)
{
    EFI_STATUS  Status;
    EFI_HANDLE  *ControllerHandles = NULL;
    UINTN       HandleIndex,  NumHandles;
    //1. 找出所有安装了 SimpleFileSystemProtocol 的 Handle，也就是找出所有 FAT 分区
    Status = gBS->LocateHandleBuffer(ByProtocol,
        &gEfiSimpleFileSystemProtocolGuid,
        NULL,
        &NumHandles,
        &ControllerHandles);
    if (EFI_ERROR(Status)) {
        returnStatus;
    }
    //2. 遍历找到的 Handle
for (HandleIndex = 0; HandleIndex < NumHandles; HandleIndex++) {
        EFI_SIMPLE_FILE_SYSTEM_PROTOCOL *SimpleFileSystem;
        //打开 Handle 上的 SimpleFileSystemProtocol
        Status = gBS->HandleProtocol(
                        ControllerHandles[HandleIndex],
                        &gEfiSimpleFileSystemProtocolGuid,
                        (VOID**)&SimpleFileSystem);
        //检查 efi\boot\bootx64.efi 文件
         ...
}
    //3. 释放 ControllerHandles 占用的内存
    if (ControllerHandles != NULL)
        Status = gBS->FreePool(ControllerHandles);
    return Status;
}
```

4.2.5　其他一些使用 Protocol 的服务

除了打开 Protocol 和根据 Protocol 找出设备这些常用服务，启动服务中关于使用 Protocol 的服务还有 ProtocolsPerHandle 和 OpenProtocolInformation。

ProtocolsPerHandle 用于获得指定设备所支持的所有 Protocol。这些 Protocol 的 GUID 通过 ProtocolBuffer 返回给调用者，UEFI 负责分配内存给 ProtocolBuffer，调用者负责释放该内存。代码清单 4-10 是该服务的函数原型。

代码清单 4-10　BS 的 ProtocolsPerHandle 服务的函数原型

```
/** gBS-> ProtocolsPerHandle
  返回指定设备所支持的所有 Protocol
  @retval EFI_SUCCESS             成功返回 Handle 上安装的所有 Protocol
  @retval EFI_OUT_OF_RESOURCES    资源耗尽
  @retval EFI_INVALID_PARAMETER   参数非法，或 Handle 不是有效的 Handle
```

```
**/
typedef EFI_STATUS(EFIAPI *EFI_PROTOCOLS_PER_HANDLE) (
IN EFI_HANDLE Handle,                    // 找出这个 Handle 上的所有 Protocol
OUT EFI_GUID  ***ProtocolBuffer,         // 返回 Protocol GUID 数组
OUT UINTN     *ProtocolBufferCount       // 返回 Protocol 的数目
);
```

OpenProtocolInformation 用于获得指定设备上指定 Protocol 的打开信息。代码清单 4-11 是该服务的函数原型。

代码清单 4-11　BS 的 OpenProtocolInformation 服务的函数原型

```
// gBS ->OpenProtocolInformation
// 返回指定设备上指定 Protocol 的打开信息
typedef EFI_STATUS(EFIAPI *EFI_OPEN_PROTOCOL_INFORMATION) (
  IN EFI_HANDLE Handle,                                  // 设备句柄
  IN EFI_GUID    *Protocol,                              // 待查询的 Protocol
  OUT EFI_OPEN_PROTOCOL_INFORMATION_ENTRY **EntryBuffer, // 打开信息通过此数组返回
  OUT UINTN      *EntryCount                             // EntryBuffer 数组元素个数
);
```

返回的打开信息包括使用者句柄（AgentHandle）、控制器句柄（ControllerHandle）、打开属性和打开个数，其数据结构如代码清单 4-12 所示。

代码清单 4-12　EFI_OPEN_PROTOCOL_INFORMATION_ENTRY 数据结构

```
typedef struct {
  EFI_HANDLE AgentHandle;
  EFI_HANDLE ControllerHandle;
  UINT32 Attributes;
  UINT32 OpenCount;
} EFI_OPEN_PROTOCOL_INFORMATION_ENTRY;
```

Handle 是这个设备的句柄。回顾一下 4.1 节的内容，可以看出这些打开信息存放在 PROTOCOL_INTERFACE 的 OpenList 列表中。设备上的同一 Protocol 可能被打开和关闭很多次。Protocol 每一次被成功打开和关闭后都会更新 OpenList 列表。成功打开后，都会在设备句柄上添加一项 EFI_OPEN_PROTOCOL_INFORMATION_ENTRY，若 OpenList 列表中已经存在一项与当前的（AgentHandle，ControllerHandle，Attributes）完全相同，则将该项的 OpenCount 增加一。关闭 Protocol 则将 OpenList 对应项的 OpenCount 减一，OpenCount 为零时删除对应的项。

4.2.6　CloseProtocol 服务

Protocol 使用完毕后要通过 CloseProtocol 关闭打开的 Protocol。其函数原型如代码

清单 4-13 所示。

代码清单 4-13　BS 的 CloseProtocol 服务的函数原型

```
typedef EFI_STATUS(EFIAPI *EFI_CLOSE_PROTOCOL) (
  IN EFI_HANDLE Handle,
  IN EFI_GUID *Protocol,
  IN EFI_HANDLE AgentHandle,
  IN EFI_HANDLE ControllerHandle
);
```

通过 HandleProtocol 和 LocateProtocol 打开的 Protocol 因为没有指定 AgentHandle，所以无法关闭。如果一定要关闭它，则要调用 OpenProtocolInformation() 获得 AgentHandle 和 ControllerHandle，然后关闭它。

4.3　Protocol 服务示例

示例 4-4 是一个完整的例子，功能是用 EFI_DISK_IO_PROTOCOL 读取 GPT 硬盘的分区表。在这个例子中展示了 LocateHandleBuffer、HandleProtocol、OpenProtocol、LocateProtocol 和 CloseProtocol 的用法。

【示例 4-4】 读取 GPT 硬盘分区表。

```
#include <Uefi.h>
#include <Base.h>
#include <Library/UefiLib.h>
#include <Library/BaseLib.h>
#include <Library/BaseMemoryLib.h>
#include <Library/UefiBootServicesTableLib.h>
#include <Library/PrintLib.h>
#include <Protocol/DiskIo.h>
#include <Protocol/BlockIo.h>
#include <Protocol/DevicePath.h>
#include <Protocol/DevicePathToText.h>
#include <Uefi/UefiGpt.h>
#include <Library/DevicePathLib.h>
EFI_STATUS EFIAPI UefiMain(IN EFI_HANDLE ImageHandle,
                           IN EFI_SYSTEM_TABLE  *SystemTable)
{
  EFI_STATUS Status;
  UINTN HandleIndex, HandleCount;
  EFI_HANDLE *DiskControllerHandles = NULL;
  EFI_DISK_IO_PROTOCOL  *DiskIo;
/* 找到所有提供 EFI_DISK_IO_PROTOCOL 的设备 */
  Status = gBS->LocateHandleBuffer( ByProtocol,
```

```
  &gEfiDiskIoProtocolGuid, NULL, &HandleCount, &DiskControllerHandles);
if (!EFI_ERROR(Status)) {
  CHAR8 gptHeaderSector[512];
  EFI_PARTITION_TABLE_HEADER* gptHeader =
    (EFI_PARTITION_TABLE_HEADER*)gptHeaderSector;
  for (HandleIndex = 0; HandleIndex < HandleCount; HandleIndex++) {
    /* 打开 EFI_DISK_IO_PROTOCOL  */
    Status = gBS->HandleProtocol(DiskControllerHandles[HandleIndex],
      &gEfiDiskIoProtocolGuid, (VOID**)&DiskIo);
    if (!EFI_ERROR(Status)){
    {
      EFI_DEVICE_PATH_PROTOCOL *DiskDevicePath;
      EFI_DEVICE_PATH_TO_TEXT_PROTOCOL  *Device2TextProtocol = 0;
      CHAR16* TextDevicePath = 0;
      /*1. 打开 EFI_DEVICE_PATH_PROTOCOL  */
      Status = gBS->OpenProtocol(DiskControllerHandles[HandleIndex],
        &gEfiDevicePathProtocolGuid, (VOID**)&DiskDevicePath,
        ImageHandle, NULL, EFI_OPEN_PROTOCOL_GET_PROTOCOL);
      if(!EFI_ERROR(Status)){
        if(Device2TextProtocol == 0)
         Status = gBS->LocateProtocol(&gEfiDevicePathToTextProtocolGuid,
            NULL, (VOID**)&Device2TextProtocol);
        /*2. 使用 EFI_DEVICE_PATH_PROTOCOL 得到文本格式的 Device Path  */
        TextDevicePath = Device2TextProtocol->
          ConvertDevicePathToText(DiskDevicePath, TRUE, TRUE);
        Print(L"%s\n", TextDevicePath);
        if(TextDevicePath)gBS->FreePool(TextDevicePath);
        /*3. 关闭 EFI_DEVICE_PATH_PROTOCOL */
        Status = gBS->CloseProtocol(DiskControllerHandles[HandleIndex],
           &gEfiDevicePathProtocolGuid, ImageHandle, NULL);
      }
    }
    {
      EFI_BLOCK_IO_PROTOCOL* BlockIo=*(EFI_BLOCK_IO_PROTOCOL**)(DiskIo + 1);
      EFI_BLOCK_IO_MEDIA* Media = BlockIo->Media;
      /* 读 1 号扇区  */
      Status = DiskIo->ReadDisk(DiskIo, Media->MediaId, 512, 512, gptHeader);
      /* 检查 GPT 标志  */
      if((!EFI_ERROR(Status)) &&( gptHeader -> Header.Signature == 0x5452415020494645)){
        UINT32 CRCsum, GPTHeaderCRCsum = (gptHeader->Header.CRC32);
        gptHeader->Header.CRC32 = 0;
        gBS -> CalculateCrc32(gptHeader ,(gptHeader->Header.HeaderSize),
          &CRCsum);
        if(GPTHeaderCRCsum == CRCsum){
          // 找到合法的 GPT Header
        }
      } // end if( … 0x5452415020494645))
    }
```

```
        }     //end if
      }       //end for
      gBS->FreePool(DiskControllerHandles);
    }         //end if
  }
```

4.4　本章小结

本章我们讲述了 Protocol 的概念以及如何使用 Protocol。Protocol 提供了一种在 UEFI 应用程序以及 UEFI 驱动之间的通信方式。通过 Protocol，用户可以使用驱动提供的服务，以及系统提供的其他服务。开发 UEFI 应用和驱动时，Protocol 是一定会用到的数据结构之一，一定要熟练掌握其用法。回顾一下，使用 Protocol 时一般需要经过以下三个步骤。

第一步：通过 gBS->OpenProtocol（或 HandleProtocol、LocateProtocol）找出 Protocol 对象。

第二步：使用这个 Protocol 提供的服务。

第三步：通过 gBS->CloseProtocol 关闭打开的 Protocol。

在应用程序中，因为没有控制器，所以可以使用 gBS->HandleProtocol 这个 gBS->OpenProtocol 的简化版。

表 4-1 列出了 Boot Service 提供的使用 Protocol 的服务。

表 4-1　Boot Service 中的 Protocol 服务

OpenProtocol	打开 Protocol
HandleProtocol	打开 Protocol，OpenProtocol 的简化版
LocateProtocol	找出系统中指定 Protocol 的第一个实例
LocateHandleBuffer	找出支持指定 Protocol 的所有 Handle。系统负责分配内存，调用者负责释放内存
LocateHandle	找出支持指定 Protocol 的所有 Handle，调用者负责分配和释放内存
OpenProtocolInformation	返回指定 Protocol 的打开信息
ProtocolsPerHandle	找出指定 Handle 上安装的所有 Protocol
CloseProtocol	关闭 Protocol

Chapter 5 第 5 章

UEFI 的基础服务

作为操作系统和硬件之间的接口，UEFI 的最主要功能就是为操作系统加载器准备软件和硬件资源，接口是以启动服务和运行时服务的形式提供给操作系统和 UEFI 应用程序使用的。应用程序只有通过系统表才能获得启动服务和运行时服务。得到了系统表、启动服务和运行时服务也就控制了整个计算机系统。掌握了这三个基础服务的用法，也就敲开了 UEFI 开发的大门。

5.1 系统表

对 UEFI 应用程序和驱动程序开发人员来讲，系统表是最重要的数据结构之一，它是用户空间通往内核空间的通道。有了它，UEFI 应用程序和驱动才可以访问 UEFI 内核、硬件资源和输入 / 输出设备。

（1）在应用程序和驱动中如何访问系统表

计算机系统进入 DXE 阶段后系统表被初始化，因而系统表只能用于 DXE 阶段以及以后的应用程序和驱动中。系统表是 UEFI 内核的一个全局结构体，其指针作为程序映像（Image）入口函数的参数传递到用户空间。程序映像（包括 UEFI 应用程序、DXE 驱动程序以及 UEFI 驱动程序）的入口函数有统一的格式，其函数原型如下所示：

```
typedef EFI_STATUS(EFIAPI *EFI_IMAGE_ENTRY_POINT)(
  IN  EFI_HANDLE ImageHandle,           //程序映像（Image）的句柄
  IN  EFI_SYSTEM_TABLE *SystemTable     //系统表指针
);
```

（2）系统表指针从内核传递到用户空间的过程

通常，程序映像的入口函数是 _ModuleEntryPoint（当一个 Image 被启动服务的 StartImage 服务启动时，执行的就是这个入口函数）。当应用程序或驱动加载到内存形成 Image 后（ImageHandle 是这个 Image 的句柄），_ModuleEntryPoint 函数地址被赋值给 Image 对象的 EntryPoint，然后 Image->EntryPoint(ImageHandle,SystemTable) 会被执行，最终会从 Image 的入口函数 _ModuleEntryPoint 执行到模块的入口函数（模块的入口函数是通过 .inf 文件中 ENTRY_POINT 指定的那个函数）。通过图 5-1 中的入口函数调用栈可以看出参数 SystemTable 的传递过程。

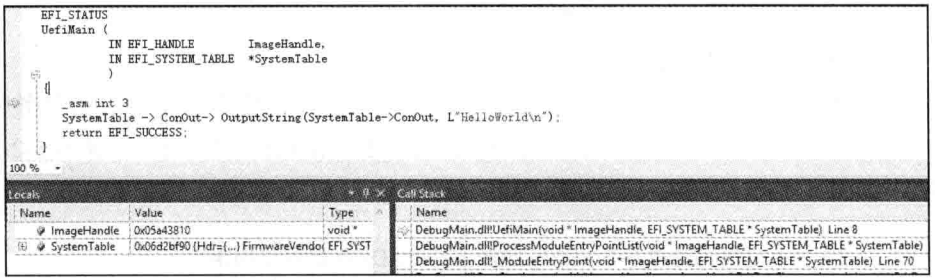


图 5-1　模块入口函数调用栈

5.1.1　系统表的构成

系统表提供了用户空间程序访问内核的接口。通过系统表，应用程序可以得到输入 / 输出控制设备。另外，可以取得启动服务表以及运行时服务表，进而通过启动服务和运行时服务表，访问硬件设备、内核服务等。通过系统表还可以获得固件的开发商名称和固件的版本号。系统表可分为如下 6 个部分。

- 表头：包括表的版本号、表的 CRC 校验码等。
- 固件信息：包括固件开发商名字字符串及固件版本号。
- 标准输入控制台、标准输出控制台、标准错误控制台。
- 启动服务表。
- 运行时服务表。
- 系统配置表。

代码清单 5-1 是系统表的数据结构定义。

代码清单 5-1　系统表数据结构

```
typedef struct {
  EFI_TABLE_HEADER Hdr;                    // 标准 UEFI 表头
  CHAR16 *FirmwareVendor;                  // 固件提供商
  UINT32 FirmwareRevision;                 // 固件版本号
```

```
  EFI_HANDLE ConsoleInHandle;                    // 输入控制台设备的句柄
  EFI_SIMPLE_TEXT_INPUT_PROTOCOL *ConIn;
  EFI_HANDLE ConsoleOutHandle;                   // 输出控制台设备的句柄
  EFI_SIMPLE_TEXT_OUTPUT_PROTOCOL *ConOut;
  EFI_HANDLE StandardErrorHandle;                // 标准错误控制台设备
  EFI_SIMPLE_TEXT_OUTPUT_PROTOCOL *StdErr;
  EFI_RUNTIME_SERVICES *RuntimeServices;         // 运行时服务表
  EFI_BOOT_SERVICES *BootServices;               // 启动服务表
  UINTN NumberOfTableEntries;                    // ConfigurationTable 数组大小
  EFI_CONFIGURATION_TABLE *ConfigurationTable;   // 系统配置表数组
} EFI_SYSTEM_TABLE;
```

下面简单介绍一下系统表中的重要组成部分。

（1）表头 EFI_TABLE_HEADER

UEFI 中的表通常都以 EFI_TABLE_HEADER 开头，5.2 节将要介绍的启动服务表和 5.3 节将要介绍的运行时服务表也都是以 EFI_TABLE_HEADER 开头。代码清单 5-2 是 EFI_TABLE_HEADER 数据结构定义，结构比较简单。

代码清单 5-2　EFI_TABLE_HEADER 数据结构定义

```
typedef struct {
  UINT64 Signature;
  UINT32 Revision;
  UINT32 HeaderSize;
  UINT32 CRC32;
  UINT32 Reserved;
} EFI_TABLE_HEADER;
```

这里介绍一下 Signature、HeaderSize 和 CRC32。

UEFI 中的 Signature 为 64 位的无符号整数，为了帮助开发者使用，EDK2 提供了宏 SIGNATURE_64（A, B, C, D, E, F, G, H），它用于将 ASCII 码串转换为 64 位的无符号整数。例如，EFI_SYSTEM_TABLE 的 Signature 为 SIGNATURE_64（'I', 'B', 'I', ' ', 'S', 'Y', 'S', 'T'）。

HeaderSize 是整个表的长度，对系统表来讲，就是 sizeof（EFI_SYSTEM_TABLE）。

CRC32 是表的校验码。计算 CRC32 校验码时，首先将数据结构中 CRC32 域清零，然后计算整张表（表大小为 HeaderSize）的 CRC32 码，计算后将校验码填充到 CRC32 域。

（2）标准输入控制台、标准输出控制台和标准错误控制台

系统表还提供了三个控制台设备以及操作三个控制台设备的 Protocol。ConIn 用于从输入控制台 ConsoleInHandle 读取字符，通常输入控制台为键盘。ConOut 用于向输出控制台 ConsoleOutHandle 输出字符串，通常这个输出控制台为屏幕。StdErr 用于向标准错误控制台 StandardErrorHandle 输出字符串，通常这个标准错误控制台为屏幕。这三个控制台设备以及 ConIn、ConOut、StdErr 三个 Protocol 在驱动 ConSplitterDxe 中被初始化。

（3）系统配置表

ConfigurationTable 是系统配置表，它指向 EFI_CONFIGURATION_TABLE 数组，数组中每一项是一个表，这个表的数据结构定义如代码清单 5-3 所示。例如，ConfigurationTable 通常会包含 ACPI（Advanced Configuration and Power Interface）表，ACPI 在系统配置表中可表示为：{gEfiAcpiTableGuid，EFI_ACPI_3_0_ROOT_SYSTEM_DESCRIPTION_POINTER*}。

代码清单 5-3　EFI_CONFIGURATION_TABLE 数据结构

```
typedef struct {
  EFI_GUID VendorGuid;            // 配置表标识符
  VOID *VendorTable;              // 指向配置表的数据
} EFI_CONFIGURATION_TABLE;
```

5.1.2　使用系统表

系统表是 UEFI 内核中的全局数据结构，应用程序运行在用户空间。读者可能会有疑问，用户空间是如何获得内核空间系统表指针的呢？其实在 UEFI 中只有一个地址空间，所有程序都运行在 RING0 优先级，应用程序地址空间（习惯上仍称为用户空间）占用 UEFI 地址空间的一部分。既然用户空间和内核空间是一个整体，在应用程序内也就可以直接使用内核空间的任何地址了，那么在应用程序内，只要得到了系统表的地址，也就可以使用系统表了。

1. 在用户空间使用系统表

系统表的地址可以通过模块入口函数的参数得到。示例 5-1 展示了如何利用系统表读取键盘和写屏幕。

【示例 5-1】 系统表使用示例。

```
#include <Uefi.h>
EFI_STATUS UefiMain(IN EFI_HANDLE ImageHandle,IN EFI_SYSTEM_TABLE *SystemTable)
{
  EFI_STATUS Status;
  UINTN Index;
  EFI_INPUT_KEY Key;
  CHAR16 StrBuffer[3] = {0};
  SystemTable->BootServices->WaitForEvent(1, &SystemTable->ConIn->WaitForKey, &Index);
                                                    // 等待按键事件
  Status = SystemTable->ConIn->ReadKeyStroke(SystemTable->ConIn, &Key);// 读取键盘
  StrBuffer[0] = Key.UnicodeChar;                   // 取得按键的 Unicode 码，参见第 12 章
  StrBuffer[1] = '\n';
  SystemTable->ConOut->OutputString(SystemTable->ConOut, StrBuffer); // 显示字符串
  return EFI_SUCCESS;
}
```

下面结合示例 5-1 来讲解系统表的使用方法。在示例 5-1 所示的代码中：

- SystemTable->BootServices 指向系统的启动服务表。
- SystemTable->ConIn 指向安装在标准输入设备上的 EFI_SIMPLE_TEXT_INPUT_PROTOCOL。
- SystemTable->ConOut 指向安装在标准输出设备上的 EFI_SIMPLE_TEXT_OUTPUT_PROTOCOL（简称 ConOut）。

该示例首先用 WaitForEvent 等待键盘事件，然后调用 ConIn 的 ReadKeyStroke 读取键盘，最后调用 ConOut 的 OutputString 服务将按键显示到屏幕。WaitForEvent、ReadKeyStroke 和 OutputString 函数将在后续章节详细介绍，这里仅简单介绍一下其用法。

启动服务 BootServices 提供的 EFI_STATUS WaitForEvent (IN UINTN NumberOfEvents, IN EFI_EVENT *Event, OUT UINTN *Index) 服务用于等待 Event 数组中任一事件的发生。其含义是 Event 数组（NumberOfEvents 是数组的大小）中任一事件被触发时该函数返回，Index 返回被触发的事件在 Event 数组中的下标。该函数是阻塞函数。在示例 5-1 中，WaitForEvent 作用是等待 SystemTable->ConIn->WaitForKey 事件即按键事件的发生。

EFI_STATUS ReadKeyStroke(EFI_SIMPLE_TEXT_INPUT_PROTOCOL *This, OUTEFI_INPUT_KEY *Key) 用于读取按键，它是 ConInProtocol 的成员函数，第一个参数是 This 指针，第二个参数用于返回按键。

EFI_STATUS OutputString(EFI_SIMPLE_TEXT_OUTPUT_PROTOCOL *This, CHAR16* String) 用于向屏幕打印字符串，它是 ConOut Protocol 的成员函数，第一个参数同样是 This 指针，第二个参数是打印到屏幕的字符串。

2. 在用户空间使用 gST 访问系统表

示例 5-1 的模块入口函数 UefiMain 中使用传入的参数 SystemTable 访问系统表。在第 3 章提到过，在其他函数中可以使用 gST 变量访问系统表。现在来解释一下在应用程序和驱动中为什么可以使用 gST 变量。EDK2 为了方便开发者，提供了 UefiBootServicesTableLib，在 UefiLib 定义了全局变量 gST（指向 SystemTable）、gBS（指向 SystemTable->BootServices）和 gImageHandle（ImageHandle）。这三个全局变量在函数 UefiBootServicesTableLibConstructor 中被初始化，这个函数是库 UefiBootServicesTableLib 的构造函数，在 AutoGen.c 中的 ProcessLibraryConstructorList 被调用。我们已经知道 ProcessLibraryConstructorList 是在 UefiMain 之前被调用的。代码清单 5-4 是 UefiBootServicesTableLib 库的构造函数 UefiBootServicesTableLibConstructor 的源码。

代码清单 5-4　构造函数 UefiBootServicesTableLibConstructor 的源码

```
//@file MdePkg\Library\UefiBootServicesTableLib\UefiBootServicesTableLib.c
EFI_HANDLEgImageHandle = NULL;
EFI_SYSTEM_TABLE*gST= NULL;
EFI_BOOT_SERVICES*gBS= NULL;
EFI_STATUS EFIAPI UefiBootServicesTableLibConstructor ( IN  EFI_HANDLE ImageHandle,
IN EFI_SYSTEM_TABLE* SystemTable)
{
  gImageHandle = ImageHandle;
  gST = SystemTable;
  gBS = SystemTable->BootServices;
  return EFI_SUCCESS;
}
```

这里需要强调一点，gST 变量是定义在用户空间的变量，而它指向的系统表定义在 UEFI 内核中。

在应用程序或驱动工程文件的 [LibraryClasses] 里引用 UefiBootServicesTableLib 后，就可以使用 gST 访问系统表了。示例 5-2 中使用了 gST 变量，与示例 5-1 中的作用完全相同。

【示例 5-2】 使用 gST。

```
#include <Uefi.h>
#include <Library/UefiBootServicesTableLib.h>
EFI_STATUS UefiMain(IN EFI_HANDLE ImageHandle,IN EFI_SYSTEM_TABLE *SystemTable)
{
  EFI_STATUS Status;
  UINTN Index;
  EFI_INPUT_KEY Key;
  CHAR16 StrBuffer[4] = {0};
  gST -> ConOut -> OutputString(gST -> ConOut, L"Please enter any key\n");
  gBS->WaitForEvent(1, &gST -> ConIn -> WaitForKey, &Index);
  Status = gST -> ConIn -> ReadKeyStroke (gST -> ConIn, &Key);
  StrBuffer[0] = Key.UnicodeChar;
  StrBuffer[1] = '\n';
  gST -> ConOut -> OutputString(gST -> ConOut, StrBuffer);
  return EFI_SUCCESS;
}
```

本节讲述了系统表的构成、系统表从内核空间传递到用户空间的过程，以及在用户空间访问系统表的两种方法。得到系统表也就可以在用户空间访问 UEFI 内核了，应用程序和驱动对内核的控制是通过系统表的两个核心成员 BootServices(启动服务）和 RuntimeServices(运行时服务）完成的。这两个成员是如此重要，以至于 EDK2 在用户空间分配了两个全局变量 gBS 和 gRT 指代这两个服务。下面两节就详细讲述这两个服务。

5.2 启动服务

在系统启动过程中，系统资源通过启动服务提供的服务来管理。系统进入 DXE 阶段时启动服务表被初始化，最终通过 SystemTable 指针将启动服务（BS）表传递给 UEFI 应用程序或驱动程序。UEFI 应用程序和驱动程序可以通过 gST->BootServices 或 gBS 访问启动服务表。

启动服务是 UEFI 的核心数据结构，有了它，我们才可以使用计算机系统内的资源。它提供的服务可以分为以下几类。

- UEFI 事件服务：事件是异步操作的基础。有了事件的支持，才可以在 UEFI 系统内执行并发操作。
- 内存管理服务：主要提供内存的分配与释放服务，管理系统内存映射。
- Protocol 管理服务：提供了安装 Protocol 与卸载 Protocol 的服务，以及注册 Protocol 通知函数（该函数在 Protocol 安装时调用）的服务。
- Protocol 使用类服务：包括 Protocol 的打开与关闭，查找支持 Protocol 的控制器。
- 驱动管理服务：包括用于将驱动安装到控制器的 connect 服务，以及将驱动从控制器上卸载的 disconnect 服务。
- Image 管理服务：此类服务包括加载、卸载、启动和退出 UEFI 应用程序或驱动。
- ExitBootServices：用于结束启动服务，此服务成功返回后系统进入 RT 期。
- 其他服务。

5.2.1 启动服务的构成

启动服务也称启动服务表，它由 UEFI 表头和表项组成。表中每一项是一个函数指针，这个函数用于提供一项服务。开发 UEFI 应用和驱动离不开启动服务，深入理解启动服务提供的每一个服务是开发者一项必不可少的任务。上文提到过，启动服务中的服务大致可以分为 8 类：UEFI 事件服务、内存管理服务、Protocol 管理服务、Protocol 使用类服务、驱动管理服务、Image 管理服务、ExitBootServices 及其他服务。本节将简单介绍启动服务中的每一类服务的作用，并详细介绍内存管理相关的服务。

1. 启动服务中的 8 类服务简介

（1）UEFI 事件服务

UEFI 事件服务包含事件（Event）、定时器（Timer）和任务优先级（TPL）三类服务[⊖]。

- 事件服务用于产生、关闭、触发、等待事件和检查事件状态。
- 定时器服务用于设置定时器属性。

⊖ 事件服务专指 BS 中函数名以 Event 结尾的 6 个服务。UEFI 事件服务是事件服务、定时器服务和任务优先级服务的统称。

- 任务优先级服务用于提升、降低当前程序的优先级。

详细的用法将在第 6 章介绍。表 5-1 列出了启动服务中 UEFI 事件服务相关的函数。

表 5-1　启动服务中 UEFI 事件服务相关的函数

函数名	作　用
CreateEvent	生成一个事件对象
CreateEventEx	生成一个事件对象并将该事件加入到一个组内
CloseEvent	关闭事件对象
SignalEvent	触发事件对象
WaitForEvent	等待事件数组中的任一事件被触发
CheckEvent	检查事件状态
SetTimer	设置定时器属性
RaiseTPL	提升任务优先级
RestoreTPL	恢复任务优先级

（2）内存管理服务

内存管理服务主要提供内存的分配与释放服务、管理系统内存映射。它主要包括 AllocatePages、FreePages、GetMemoryMap、AllocatePool 及 FreePool 五个服务，具体见表 5-2。

表 5-2　启动服务中的内存管理服务

内存管理服务	作　用
AllocatePool	分配内存
FreePool	释放内存
GetMemoryMap	获得当前内存映射（物理地址⟷虚地址）
AllocatePages	分配内存页
FreePages	释放内存页

（3）Protocol 管理服务

UEFI 提供了安装和卸载 Protocol 的服务，以及注册 Protocol 通知函数（该函数在 Protocol 安装时调用）的服务。此类服务主要供 Protocol 提供者使用，具体用法将在第 8 章和第 9 章详细讲述。

表 5-3　启动服务中的 Protocol 管理服务

Protocol 管理服务	作　用
InstallProtocolInterface	安装 Protocol 到设备上
UninstallProtocolInterface	从设备上卸载 Protocol
ReinstallProtocolInterface	重新安装 Protocol
RegisterProtocolNotify	为指定的 Protocol 注册通知事件，当这个 Protocol 安装时，该事件触发
InstallMultipleProtocolInterfaces	安装多个 Protocol 到设备上
UninstallMultipleProtocolInterfaces	从设备上卸载多个 Protocol

（4）Protocol 使用类服务

该服务包括 Protocol 的打开与关闭、查找支持 Protocol 的控制器。此类服务主要供 Protocol 使用者使用，具体用法已经在第 4 章讲述。表 5-4 为启动服务中的 Protocol 使用类服务说明。

表 5-4　启动服务中的 Protocol 使用类服务

Protocol 使用类服务	作　用
OpenProtocol	打开 Protocol
HandleProtocol	打开 Protocol，OpenProtocol 的简化版
LocateProtocol	找出系统中指定 Protocol 的第一个实例
LocateHandleBuffer	找出支持指定 Protocol 的所有 Handle。系统负责分配内存，调用者负责释放内存
LocateHandle	找出支持指定 Protocol 的所有 Handle，调用者负责分配和释放内存
OpenProtocolInformation	返回指定 Protocol 的打开信息
ProtocolsPerHandle	找出指定 Handle 上安装的所有 Protocol
CloseProtocol	关闭 Protocol
LocateDevicePath	在指定的设备路径下找出支持给定 Protocol 的设备，并返回离指定的设备路径最近的设备

（5）驱动管理服务

此类服务包括用于将驱动安装到控制器的 connect 服务，以及将驱动从控制器上卸载的 disconnect 服务。具体用法将在第 9 章讲述。表 5-5 为启动服务中的驱动管理服务说明。

表 5-5　启动服务中的驱动管理服务

驱动管理服务	作　用
ConnectController	将驱动安装到指定的设备控制器
DisconnectController	将驱动从指定的设备控制器卸载

（6）Image 管理服务

此类服务包括加载、卸载、启动和退出 UEFI 应用程序或驱动。表 5-6 为启动服务中的 Image 管理服务说明。

表 5-6　启动服务中的 Image 管理服务

Image 管理服务	作　用
LoadImage	加载 .efi 文件至内存并生成 Image
StartImage	启动 Image，也就是调用 Image 的入口函数
Exit	退出 Image
UnloadImage	卸载 Image

（7）ExitBootServices

ExitBootServices 用于结束启动服务，此服务成功返回后，系统进入 RT 期。操作系统

加载器从启动服务接过对计算机系统的控制权后必须调用此服务。表 5-7 为启动服务中的 ExitBootServices 说明。

表 5-7　启动服务中的 ExitBootServices

服务名	作　用
ExitBootServices	结束启动服务

（8）其他服务

表 5-8 为启动服务中的其他服务说明。

表 5-8　启动服务中的其他服务

服务名	作　用
InstallConfigurationTable	管理（增加、更新、删除）系统配置表项
GetNextMonotonicCount	获得系统单调计数器的下一个值
Stall	暂停 CPU 指定的微秒数
SetWatchdogTimer	设置“看门狗”定时器，即在指定的时间内若系统无反应，则重启系统
CalculateCrc32	计算 CRC32 校验码
CopyMem	复制内存
SetMem	设置指定内存区域的值

像系统表一样，启动服务也以表的形式存在，由标准表头和表项组成，上面讲的 8 类服务以函数指针的形式组成了启动服务表的表项。代码清单 5-5 列出了启动服务表的数据结构。

代码清单 5-5　启动服务表 EFI_BOOT_SERVICES 数据结构

```
typedef struct {
  EFI_TABLE_HEADER     Hdr;
  //任务优先级服务
  EFI_RAISE_TPL  RaiseTPL;
  EFI_RESTORE_TPL RestoreTPL;
  //内存服务
  EFI_ALLOCATE_PAGES AllocatePages;
  EFI_FREE_PAGES     FreePages;
  EFI_GET_MEMORY_MAP GetMemoryMap;
  EFI_ALLOCATE_POOL  AllocatePool;
  EFI_FREE_POOL      FreePool;
  //事件、定时器服务
  EFI_CREATE_EVENT   CreateEvent;
  EFI_SET_TIMER      SetTimer;
  EFI_WAIT_FOR_EVENT WaitForEvent;
  EFI_SIGNAL_EVENT   SignalEvent;
  EFI_CLOSE_EVENT    CloseEvent;
  EFI_CHECK_EVENT    CheckEvent;
  //Protocol 相关服务
  EFI_INSTALL_PROTOCOL_INTERFACE   InstallProtocolInterface;
```

```
    EFI_REINSTALL_PROTOCOL_INTERFACE ReinstallProtocolInterface;
    EFI_UNINSTALL_PROTOCOL_INTERFACE UninstallProtocolInterface;
    EFI_HANDLE_PROTOCOL              HandleProtocol;
    VOID *Reserved;
    EFI_REGISTER_PROTOCOL_NOTIFY RegisterProtocolNotify;
    EFI_LOCATE_HANDLE LocateHandle;
    EFI_LOCATE_DEVICE_PATH LocateDevicePath;
    EFI_INSTALL_CONFIGURATION_TABLE InstallConfigurationTable;
    // Image 相关服务
    EFI_IMAGE_LOAD LoadImage;
    EFI_IMAGE_START StartImage;
    EFI_EXIT Exit;
    EFI_IMAGE_UNLOAD UnloadImage;
    EFI_EXIT_BOOT_SERVICES ExitBootServices;
    // 其他服务
    EFI_GET_NEXT_MONOTONIC_COUNT GetNextMonotonicCount;
    EFI_STALL Stall;
    EFI_SET_WATCHDOG_TIMER SetWatchdogTimer;
    // 驱动相关服务
    EFI_CONNECT_CONTROLLER ConnectController;
    EFI_DISCONNECT_CONTROLLER DisconnectController;
    // Protocol 相关服务
    EFI_OPEN_PROTOCOL OpenProtocol;
    EFI_CLOSE_PROTOCOL CloseProtocol;
    EFI_OPEN_PROTOCOL_INFORMATION OpenProtocolInformation;
    EFI_PROTOCOLS_PER_HANDLE ProtocolsPerHandle;
    EFI_LOCATE_HANDLE_BUFFER LocateHandleBuffer;
    EFI_LOCATE_PROTOCOL LocateProtocol;
    EFI_INSTALL_MULTIPLE_PROTOCOL_INTERFACES InstallMultipleProtocolInterfaces;
    EFI_UNINSTALL_MULTIPLE_PROTOCOL_INTERFACES UninstallMultipleProtocolInterfaces;
    EFI_CALCULATE_CRC32 CalculateCrc32;
    EFI_COPY_MEM        CopyMem;
    EFI_SET_MEM         SetMem;
    EFI_CREATE_EVENT_EX CreateEventEx;
} EFI_BOOT_SERVICES;
```

2. 内存管理服务详细介绍

内存管理服务的 5 个服务可以分为 3 组：AllocatePool/FreePool、AllocatePages/FreePages, GetMemoryMap。下面分别介绍这三组服务的用法。本部分的示例程序在 uefi\book\systemtable\memory 目录下。

（1）AllocatePool/FreePool 用法

AllocatePool 和 FreePool 必须成对出现，用于分配和释放内存。AllocatePool 服务的函数原型如代码清单 5-6 所示。

代码清单 5-6　BS 中的 AllocatePool 服务的函数原型

```
/**gBS->AllocatePool 分配内存
  @retval EFI_SUCCESS             成功分配所请求的内存
  @retval EFI_OUT_OF_RESOURCES    资源耗尽
  @retval EFI_INVALID_PARAMETER 参数非法
**/
typedef EFI_STATUS(EFIAPI *EFI_ALLOCATE_POOL)(
  IN EFI_MEMORY_TYPE PoolType,          //内存类型
  IN UINTN Size,                        //需分配的内存字节数
  OUT VOID **Buffer                     //返回分配的内存首地址
);
```

其中，EFI_MEMORY_TYPE 是枚举类型，定义了所有 UEFI 支持的内存类型，如代码清单 5-7 所示。有效类型的值在 [1, EfiMaxMemoryType) 之间，而 [0x80000000, 0xFFFFFFFF] 之间的值保留给操作系统加载器使用，[EfiMaxMemoryType,0x7FFFFFFF] 为非法类型。

代码清单 5-7　枚举类型 EFI_MEMORY_TYPE

```
typedef enum {
  EfiReservedMemoryType=0,     //保留类型，未使用
  EfiLoaderCode,               //分配给 OS Loader 的代码
  EfiLoaderData,               //分配给 OS Loader 的数据，应用程序分配内存时的默认类型
  EfiBootServicesCode,         //启动服务驱动 / 应用程序的代码区
  EfiBootServicesData,         //启动服务驱动 / 应用程序的数据区，BS 驱动分配内存的默认类型
  EfiRuntimeServicesCode,      //运行时服务驱动的代码区
  EfiRuntimeServicesData,      //运行时服务驱动，RS 驱动分配内存的默认类型
  EfiConventionalMemory,       //可分配内存
  EfiUnusableMemory,           //该块内存区域出现错误，不能使用
  EfiACPIReclaimMemory,        //用于存放 ACPI 表
  EfiACPIMemoryNVS,            //保留给固件使用
  EfiMemoryMappedIO,           //内存映射 I/O，可被运行时服务使用
  EfiMemoryMappedIOPortSpace,  //内存映射 I/O，被 CPU 用于转换内存周期到 IO 周期
  EfiPalCode,                  //保留给固件
  EfiMaxMemoryType
} EFI_MEMORY_TYPE;
```

在调用 gBS->ExitBootServices(…) 之后，EfiBootServicesCode 和 EfiBootServicesData 类型的内存被回收；EfiLoaderCode 和 EfiLoaderData 由 OS Loader 和操作系统自行决定是否回收；EfiACPIReclaimMemory 类型的内存在 APCI 启用后回收。其他使用的内存将保留。

使用 AllocatePool 分配的内存用完后必须使用 FreePool 释放。代码清单 5-8 给出了 FreePool 服务的函数原型。

代码清单 5-8　BS 中的 FreePool 服务的函数原型

```
/** gBS->FreePool 释放内存
  @retval EFI_SUCCESS             内存成功被系统回收
```

```
  @retval EFI_INVALID_PARAMETER Buffer 无效
**/
typedef EFI_STATUS(EFIAPI *EFI_FREE_POOL)(
  IN  VOID *Buffer //待释放内存
);
```

示例 5-3 展示了如何使用 AllocatePool 分配内存以及如何使用 FreePool 释放内存。

【示例 5-3】 AllocatePool 与 FreePool 应用示例。

```
UINTN BufSize = 1024;
CHAR16* Buf = NULL;
Status = gBS->AllocatePool(EfiBootServicesCode, BufSize, (VOID**)&Buf);
...
Status = gBS->FreePool(Buf);
```

（2）AllocatePages/FreePages 用法

AllocatePool 用于分配指定大小的内存。内核和驱动开发中经常会要求分配到的内存不得跨页，或需要分配完整的内存页，此时用 AllocatePool 就十分不方便。为此，启动服务提供了分配页面的服务 AllocatePages。代码清单 5-9 展示了其函数原型，以及其配对服务 FreePages 的函数原型。

代码清单 5-9　BS 中的 AllocatePages、FreePages 服务的函数原型

```
/**gBS->AllocatePages 分配内存页
  @retval EFI_SUCCESS                    成功分配页面
  @retval EFI_INVALID_PARAMETER          参数不合法
  @retval EFI_OUT_OF_RESOURCES           资源耗尽
  @retval EFI_NOT_FOUND                  请求的物理页不存在
**/
typedef EFI_STATUS(EFIAPI *EFI_ALLOCATE_PAGES)(
  IN EFI_ALLOCATE_TYPE Type,             //分配方式
  IN EFI_MEMORY_TYPE MemoryType,         //内存类型
  IN UINTN Pages,                        //分配的页面数
  //输入：物理地址或NULL；输出：分配到的页面的虚拟地址
  IN OUT EFI_PHYSICAL_ADDRESS *Memory
);
//释放内存页 :gBS->FreePages
typedef EFI_STATUS (EFIAPI *EFI_FREE_PAGES)(
  IN EFI_PHYSICAL_ADDRESS Memory,        //要释放的页面的起始物理地址
  IN UINTN Pages                         //要释放的页面数
);
```

参数 MemoryType 在 AllocatePool 中已经讲过。参数 Type 指明了页面的分配方式，分配方式有三种，定义在枚举类型 EFI_ALLOCATE_TYPE 中，如代码清单 5-10 所示。

代码清单 5-10　枚举类型 EFI_ALLOCATE_TYPE

```
typedef enum {
  AllocateAnyPages,                    //分配任何可用的页面
  AllocateMaxAddress,                  //分配到的页面的末地址必须不超过指定的地址
  AllocateAddress,                     //分配指定物理地址上的页面
  MaxAllocateType
} EFI_ALLOCATE_TYPE;
```

示例 5-4 展示了如何分配任意地址处的 3 个页面。示例 5-5 演示了如何分配指定地址（1024 *1024 * 10）处的 3 个页面。

【示例 5-4】 分配任意地址处的 3 个页面。

```
//@file uefi\book\systemtable\memory\BSmem.c
EFI_STATUS TestAllocateAnyPages()
{
  EFI_STATUS Status = 0;
  EFI_PHYSICAL_ADDRESS pages;
  Status = gBS->AllocatePages(AllocateAnyPages,
    EfiBootServicesData, 3, &pages);
  Print(L"AllocatePages:%r %x\n", Status, pages);
  if(Status == 0){
    //此处可以使用地址 Pages 处的内存页面
    CHAR16* str = (CHAR16*)pages;
    ...
    Status = gBS->FreePages(pages, 3);
}else{
      //未分配到页面
}
    return Status;
}
```

【示例 5-5】 分配指定地址处的 3 个页面。

```
EFI_STATUS TestAllocateAddress()
{
  EFI_STATUS Status = 0;
  EFI_PHYSICAL_ADDRESS pages = 1024 *1024 * 10;
  Status = gBS->(AllocateAddress, EfiBootServicesData, 3, &pages);
  Print(L"AllocateAddress:%r %x\n", Status, pages);
  if(Status == 0){
    //此处可以使用地址 Pages 处的内存页面
    CHAR16* str = (CHAR16*)pages;
    ...
    Status = gBS->FreePages(pages, 3);
  }else {
    //错误处理
  }
```

```
    return Status;
}
```

通过服务 AllocatePages 得到的是页面的物理地址，要将这个物理地址转换成指针才能操作页面所表示的内存，如示例 5-4 和示例 5-5 中的代码：CHAR16* str = (CHAR16*)pages。

有一个需要特别注意的地方，在 32 位系统中，如果物理内存容量超过 4GB，AllocatePages 可能会分配到地址大于 4GB 的页面，造成分配的页面永远无法访问的问题。

（3）GetMemoryMap 用法

GetMemoryMap 用于取得系统中所有的内存映射，其函数原型如代码清单 5-11 所示。参数 MemoryMapSize 作为输入参数时表示缓冲区 MemoryMap 字节数，作为输出参数时，若缓冲区不够大，返回需要的缓冲区大小；若缓冲区足够大，返回写入 MemoryMap 的字节数，输出缓冲区 MemoryMap 将返回 EFI_MEMORY_DESCRIPTOR 数组。代码清单 5-12 是 EFI_MEMORY_DESCRIPTOR 的数据结构定义，可以看出内存映射包含了内存的物理地址、虚拟地址、内存类型、内存属性和内存区域大小。

代码清单 5-11　BS 中的 GetMemoryMap 服务的函数原型

```
/** 返回系统内存映射
  @retval EFI_SUCCESS                    成功返回当前内存映射
  @retval EFI_BUFFER_TOO_SMALL           缓冲区太小，MemoryMapSize 将返回需要的大小
  @retval EFI_INVALID_PARAMETER          非法参数
**/
typedef EFI_STATUS(EFIAPI *EFI_GET_MEMORY_MAP)(
  IN OUT UINTN *MemoryMapSize,
  IN OUT EFI_MEMORY_DESCRIPTOR *MemoryMap,      // 输出缓冲区
  OUT UINTN *MapKey,                            // 当前内存映射的 Key
  OUT UINTN *DescriptorSize,                    // EFI_MEMORY_DESCRIPTOR 大小
  OUT UINT32 *DescriptorVersion                 // EFI_MEMORY_DESCRIPTOR 版本
  );
```

代码清单 5-12　EFI_MEMORY_DESCRIPTOR 数据结构

```
typedef struct {
  UINT32 Type;                                  // EFI_MEMORY_TYPE，内存类型
  EFI_PHYSICAL_ADDRESS PhysicalStart;           // 首字节物理地址，必须按 4KB 对齐
  EFI_VIRTUAL_ADDRESS VirtualStart;             // 首字节的虚拟地址，必须按 4KB 对齐
  UINT64 NumberOfPages;                         // 此区域的页面数
  UINT64 Attribute;                             // 内存属性
} EFI_MEMORY_DESCRIPTOR;
```

示例 5-6 展示了如何利用 GetMemoryMap 打印出系统中内存映射的信息。

【示例 5-6】 GetMemoryMap 示例。

```
EFI_STATUS    TestMMap()
{
  EFI_STATUS Status = 0;
  UINTN MemoryMapSize = 0;
  EFI_MEMORY_DESCRIPTOR *MemoryMap = 0;
  UINTN MapKey = 0, DescriptorSize = 0, DescriptorVersion = 0,i = 0;
  EFI_MEMORY_DESCRIPTOR *MMap = 0;
  //首先取得缓冲区大小
  Status = gBS->GetMemoryMap(&MemoryMapSize, MemoryMap, &MapKey,
    &DescriptorSize, &DescriptorVersion);
  if(Status != EFI_BUFFER_TOO_SMALL){
    return Status;
  }
  //分配内存
  Status = gBS->AllocatePool(EfiBootServicesData, MemoryMapSize, &MemoryMap);
  //调用 GetMemoryMap
  Status = gBS->GetMemoryMap(&MemoryMapSize, MemoryMap, &MapKey,
    &DescriptorSize, &DescriptorVersion);
  //数组 MemoryMap 中每个项的大小为 DescriptorSize,
  //DescriptorSize 通常不等于 sizeof(EFI_MEMORY_DESCRIPTOR)
  for( i = 0; i< MemoryMapSize / (DescriptorSize); i++)  {
    MMap = (EFI_MEMORY_DESCRIPTOR*) ( ((CHAR8*)MemoryMap) +
      i * DescriptorSize);
    Print(L"MemoryMap %4d %10d :", MMap[0].Type, MMap[0].NumberOfPages);
    Print(L"%10lx<->", MMap[0].PhysicalStart);
    Print(L"%10lx\n",  MMap[0].VirtualStart);
  }
  Status = gBS->FreePool(MemoryMap);
  return Status;
}
```

5.2.2　启动服务的生存期

从启动服务可以看出其功能之强大，通过它可以完全掌控整个计算机系统，其功能与操作系统提供的功能有很多重叠之处。同时它也占用了大量的计算机系统资源。设计启动服务的主要目的是帮助操作系统加载器初始化计算机系统，当操作系统加载器彻底取得对计算机系统的控制权之后，启动服务也就完成了它的使命，启动服务占用的系统资源也需要转交给操作系统加载器。此时需要调用 gBS->ExitBootServices，其作用正是结束启动服务、释放启动服务占用的资源以及使用启动服务分配到的启动期资源，将控制权交给操作系统加载器。也就是说，启动服务的生存期在 DxeMain 与 gBS->ExitBootServices 之间。

下面通过一段代码（见示例 5-7）来看一下执行 gBS->ExitBootServices 前后系统表及启动服务表的变化。

【示例 5-7】 执行 gBS->ExitBootServices 前后系统表和启动服务表的变化。

```
#include <Uefi.h>
#include <Library/UefiBootServicesTableLib.h>
#include <Library/DebugLib.h>
EFI_STATUSUefiMain (IN EFI_HANDLE ImageHandle,IN EFI_SYSTEM_TABLE *SystemTable)
{
  SystemTable->ConOut->OutputString(SystemTable->ConOut,
  L"ExitBootServices\n");
  {
    UINTN MemMapSize = 0;
    EFI_MEMORY_DESCRIPTOR* MemMap = 0;
    UINTN MapKey = 0;
    UINTN DesSize = 0;
    UINT32 DesVersion = 0;
    CHAR16* vendor = SystemTable->FirmwareVendor;      //固件供应商的名字
    SystemTable -> ConOut-> OutputString(SystemTable->ConOut, vendor);
    _asm int 3;
    //取得当前的MapKey
    gBS->GetMemoryMap(&MemMapSize, MemMap, &MapKey, &DesSize, &DesVersion);
    //结束启动服务
    gBS->ExitBootServices(ImageHandle, MapKey);
    _asm int 3;
    //此时，启动服务占用的内存应该已经释放
    ASSERT(SystemTable -> BootServices == NULL);
    while(DesSize >0){}
    }
    return (EFI_STATUS)-1;
}
```

本例仅仅是为了说明调用 ExitBootServices 前后系统的变化。在正常的 Boot Loader 中，执行完 gBS->ExitBootServices 后，Boot Loader 会继续引导进入操作系统。

图 5-2 是执行 gBS->ExitBootServices(…) 之前的系统表。

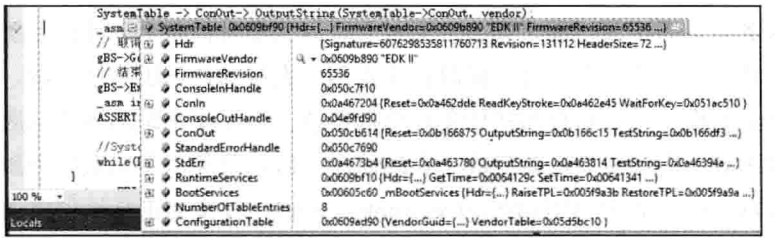


图 5-2　执行 ExitBootServices(…) 之前的系统表

图 5-3 是执行 gBS->ExitBootServices 之后的系统表，可以看出，SystemTable->BootServices 已经清零，同时清零的还有 ConIn、ConOut、StdErr，以及它们的控制器设备句柄。

其他释放的资源还包括 EfiBootServicesCode 和 EfiBootServicesData 类型的内存资源。

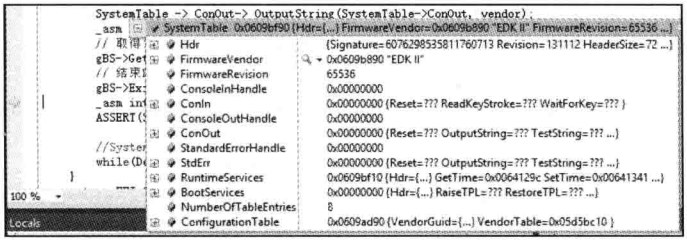


图 5-3　执行 ExitBootServices(…) 之后的系统表

仍然可以继续被 OS Loader 和操作系统使用的资源包括运行时服务和 ConfigurationTable，以及固件版本号和固件开发商名称字符串。

5.3　运行时服务

从进入 DXE 阶段运行时服务被初始化，直到操作系统结束，运行时服务都一直存在并向上层（操作系统、操作系统加载器、UEFI 应用程序或 UEFI 驱动）提供服务。

下面介绍运行时服务的构成。

运行时服务（RT）也称为运行时服务表，其结构同启动服务表类似，由表头和表项组成。每一个表项是一个函数指针，表示一个服务。

运行时服务提供的服务主要包括如下几个。

- 时间服务：读取 / 设定系统时间。读取 / 设定系统从睡眠中唤醒的时间。
- 读写系统变量：读取 / 设置系统变量，例如 BootOrder 用于指定启动项顺序。
- 虚拟内存服务：将物理地址转换为虚拟地址。
- 其他服务：包括重启系统的 ResetSystem、获取系统提供的下一个单调单增值等。

代码清单 5-13 列出了运行时服务表的数据结构。

代码清单 5-13　运行时服务表

```
typedef struct {
  EFI_TABLE_HEADER                    Hdr;
  // 时间服务
  EFI_GET_TIME                        GetTime;
  EFI_SET_TIME                        SetTime;
  EFI_GET_WAKEUP_TIME                 GetWakeupTime;
  EFI_SET_WAKEUP_TIME                 SetWakeupTime;
  // 虚拟内存服务
  EFI_SET_VIRTUAL_ADDRESS_MAP         SetVirtualAddressMap;
  EFI_CONVERT_POINTER                 ConvertPointer;
  // UEFI 系统变量服务
  EFI_GET_VARIABLE                    GetVariable;
```

```
  EFI_GET_NEXT_VARIABLE_NAME            GetNextVariableName;
  EFI_SET_VARIABLE                      SetVariable;
  //其他服务
  EFI_GET_NEXT_HIGH_MONO_COUNT          GetNextHighMonotonicCount;
  EFI_RESET_SYSTEM                      ResetSystem;
  EFI_UPDATE_CAPSULE                    UpdateCapsule;
  EFI_QUERY_CAPSULE_CAPABILITIES        QueryCapsuleCapabilities;
  EFI_QUERY_VARIABLE_INFO               QueryVariableInfo;
} EFI_RUNTIME_SERVICES;
```

下面详细讲述 RT 提供的这 4 类服务。

1. 时间服务

时间服务包括两大块：读取 / 设置硬件时间、读取 / 设置唤醒定时器。

（1）GetTime/SetTime 用法

GetTime 服务用于读取计算机硬件时间，SetTime 服务用于设置计算机硬件时间。计算机硬件时钟由单独的电池供电，因而计算机未接通电源时硬件时钟仍然正常工作。操作系统启动时通过读取硬件时钟获得时间。代码清单 5-14 是这两个服务的函数原型。

代码清单 5-14　RT 的 GetTime、SetTime 服务函数原型

```
/**gRT->GetTime
返回当前时间和日期，以及硬件时钟设备的性能
**/
typedef EFI_STATUS(EFIAPI *EFI_GET_TIME)(
  OUT EFI_TIME *Time,                                    //当前时间
  OUT EFI_TIME_CAPABILITIES *Capabilities OPTIONAL       //时钟硬件的性能
);
/** gRT->SetTime
设置 RTC 时钟的时间和日期
  @retval EFI_SUCCESS                                    成功设置时钟
  @retval EFI_INVALID_PARAMETER                          Time 超出范围
  @retval EFI_DEVICE_ERROR                               RTC 硬件返回错误
**/
typedef EFI_STATUS(EFIAPI *EFI_SET_TIME)(IN  EFI_TIME *Time);
```

GetTime 服务返回硬件时钟的时间和时钟的性能。时间的格式如代码清单 5-15 所示。

代码清单 5-15　结构体 EFI_TIME

```
typedef struct {
  UINT16 Year;                                           //1900 ~ 9999
  UINT8  Month;                                          //1 ~ 12
  UINT8  Day;                                            //1 ~ 31
  UINT8  Hour;                                           //0 ~ 23
```

```
  UINT8   Minute;                                  //0 ~ 59
  UINT8   Second;                                  //0 ~ 59
  UINT8   Pad1;                                    //填充
  UINT32  Nanosecond;                              //0 ~ 999
  INT16   TimeZone;                                //-1440 ~ 1440
  UINT8   Daylight;                                //夏时制
  UINT8   Pad2;                                    //填充
} EFI_TIME;
```

时钟性能用 EFI_TIME_CAPABILITIES 表示。代码清单 5-16 列出了 EFI_TIME_CAPABILITIES 的数据结构。

- Resolution（时钟分辨率）是实时时钟每秒报告的次数。实时时钟每报告一次输出一个时钟脉冲。RTC（Real-Time Clock）设备的分辨率通常为 1Hz，即每一秒钟报告一次。
- Accuracy 是时钟守时（timekeeping）精度，以百万分之一（ppm）为单位。实时时钟通常采用 32.768KHz 的晶体振荡器，其守时精度是 20ppm。一天有 24 × 60 × 60=86400 秒，20ppm 的误差意味着一天的误差是 86400 ×（20/1000000）=1.728 秒。
- SetsToZero 是布尔类型，True 表示设置时钟的操作会清除（低于时钟分辨率的计数）状态；False 表示设置时钟的操作不会清除（低于时钟分辨率的计数）计数状态。通常 PC-AT CMOS RTC 设备的值为 False。例如，对于 1Hz 的 RTC，如果 SetsToZero 为 Ture，调用 SetTime 后 1 秒产生下一次报告；如果 SetsToZero 为 False，调用 SetTime 不会影响产生下一次报告的时间。

代码清单 5-16　时钟性能 EFI_TIME_CAPABILITIES 结构体

```
typedef struct {
  UINT32  Resolution;
  UINT32  Accuracy;
  BOOLEAN SetsToZero;
} EFI_TIME_CAPABILITIES;
```

（2）GetWakeupTime/SetWakeupTime 用法

GetWakeupTime 用于读取唤醒定时器状态；SetWakeupTime 用于启用或禁用唤醒定时器。代码清单 5-17 列出了这两个服务的函数原型。

代码清单 5-17　RT 的 GetWakeupTime、SetWakeupTime 服务函数原型

```
/**gRT->GetWakeupTime
返回当前唤醒定时器的状态
**/
typedef EFI_STATUS(EFIAPI *EFI_GET_WAKEUP_TIME)(
  OUT BOOLEAN  *Enabled,                           //返回定时器启用或禁用状态
  OUT BOOLEAN  *Pending,                           //定时器信号是否处于 Pending 状态
```

```
  OUT EFI_TIME *Time                          //返回定时器设置
  );

/**gRT -> SetWakeupTime
设置唤醒定时器
**/
Typedef EFI_STATUS(EFIAPI *EFI_SET_WAKEUP_TIME)(
  IN BOOLEAN Enable,                          //启用或禁用唤醒定时器
  IN EFI_TIME *Time OPTIONAL                  //启用时，定时器的设置
  );
```

2. 系统变量服务

UEFI 系统变量服务包括读取变量的 GetVariable、更新或创建变量的 SetVariable，以及用于遍历系统变量的 GetNextVariableName。

（1）GetVariable 用法

GetVariable 用于根据变量名获取变量值和变量属性，代码清单 5-18 是其函数原型。

代码清单 5-18　RT 的 GetVariable 服务函数原型

```
/**gRT-> GetVariable
  根据变量名获得系统变量的值和属性
  @retval EFI_SUCCESS                         成功返回
  @retval EFI_NOT_FOUND                       变量不存在
  @retval EFI_BUFFER_TOO_SMALL                缓冲区大小 DataSize 太小
  @retval EFI_INVALID_PARAMETER               VariableName/VendorGuid/DataSize/Data 为空
  @retval EFI_DEVICE_ERROR                    设备返回错误
  @retval EFI_SECURITY_VIOLATION              该变量需要身份验证，但身份验证没通过
**/
typedef EFI_STATUS(EFIAPI *EFI_GET_VARIABLE)(
  IN CHAR16 *VariableName,                    //变量名字
  IN EFI_GUID *VendorGuid,                    //变量所有者 GUID
  OUT UINT32 *Attributes, OPTIONAL            //变量属性
  //输入值：缓冲区 Data 大小；输出值：返回到 Data 的字节数
  IN OUT UINTN *DataSize,
  OUT VOID *Data                              //返回变量的值
  );
```

属性值 Attributes 是 32 位的无符号整数，每一位表示一种属性，有效的属性值定义在代码清单 5-19 中。例如，某变量的属性为 0x00000011（即 0x00000001|0x00000010），表示该变量关机后仍然有效，并且写该变量时需提供身份验证信息。所谓关机后仍然有效，意思是该变量要写入非易失性存储器中。若变量没有 EFI_VARIABLE_NON_VOLATILE 属性，则该变量仅保存在内存中。

代码清单 5-19　UEFI 系统变量的属性

```
#define EFI_VARIABLE_NON_VOLATILE 0x00000001                    //关机后仍然有效
#define EFI_VARIABLE_BOOTSERVICE_ACCESS 0x00000002              //启动服务期间有效
#define EFI_VARIABLE_RUNTIME_ACCESS 0x00000004                  //Runtime 期间有效
#define EFI_VARIABLE_HARDWARE_ERROR_RECORD 0x00000008           //硬件错误记录
#define EFI_VARIABLE_AUTHENTICATED_WRITE_ACCESS 0x00000010      //写时需验证身份
#define EFI_VARIABLE_TIME_BASED_AUTHENTICATED_WRITE_ACCESS 0x00000020
#define EFI_VARIABLE_APPEND_WRITE 0x00000040                    //附加写
```

VendorGuid 可以作为命名空间使用。不同的操作系统或第三方厂商可以管理属于自己的变量，使用 VendorGuid 可以避免命名冲突。最常用的 VendorGuid 是 gEfiGlobalVariableGuid，它管理了全局的系统变量，如 L"Lang"、L"BootOrder"、L"BootCurrent" 等。

示例 5-9 演示了如何读取系统变量 L"BootOrder"。

【示例 5-8】 读取 UEFI 系统变量。

```
//@file  ShellPkg\Library\UefiShellDebug1CommandsLib\Bcfg.c:1217
Length = 0;
//首先获得变量所需的内存大小
Status = gRT->GetVariable(CurrentOperation.Target ==
  BcfgTargetBootOrder?(CHAR16*)L"BootOrder":(CHAR16*)L"DriverOrder",
  (EFI_GUID*)&gEfiGlobalVariableGuid,
  NULL,
  &Length,
  CurrentOperation.Order);
if (Status == EFI_BUFFER_TOO_SMALL) {
  //分配内存
  CurrentOperation.Order =
    AllocateZeroPool(Length+(4*sizeof(CurrentOperation.Order[0])));
  //读取变量
  Status = gRT->GetVariable( CurrentOperation.Target ==
    cfgTargetBootOrder?(CHAR16*)L"BootOrder":(CHAR16*)L"DriverOrder",
    (EFI_GUID*)&gEfiGlobalVariableGuid,
    NULL,
    &Length,
    CurrentOperation.Order);
  }
```

注意，利用 EFI_BUFFER_TOO_SMALL 读取数据是一种常用的模式，在 UEFI 开发中会经常用到。

（2）SetVariable 用法

SetVariable 有三项功能：新建、更新和删除变量。若变量不存在，则执行新建操作；若变量存在并且该变量允许写，则执行更新操作；删除一个变量（非 AT 变量）的两种方式：

1）DataSize 设为 0，并且 Attributes 中不包含下列属性中任意一个。

- EFI_VARIABLE_AUTHENTICATED_WRITE_ACCESS。
- EFI_VARIABLE_APPEND_WRITE。
- EFI_VARIABLE_TIME_BASED_AUTHENTICATED_WRITE_ACCESS。

2）Attributes 设为 0。

代码清单 5-20 列出了 SetVariable 服务的函数原型。

代码清单 5-20　RT 的 SetVariable 服务函数原型

```
/**gRT->SetVariable
设置（新建、更新或删除）变量
**/
typedef EFI_STATUS(EFIAPI *EFI_SET_VARIABLE)(
  IN CHAR16 *VariableName,                    //变量名字
  IN EFI_GUID *VendorGuid,                    //变量所有者 GUID
  IN UINT32 *Attributes,                      //变量属性
  IN UINTN DataSize,                          //缓冲区 Data 大小
  IN VOID *Data                               //变量的值
);
```

（3）GetNextVariableName 用法

GetNextVariableName 用于获取下一个系统变量，通过这个服务可以遍历系统中的变量，其函数原型如代码清单 5-21 所示。

代码清单 5-21　RT 的 GetNextVariableName 服务函数原型

```
/**gRT->GetNextVariableName
返回变量 (VendorGuid, VariableName) 后面的一个变量
**/
typedef EFI_STATUS(EFIAPI *EFI_GET_NEXT_VARIABLE_NAME)(
  IN OUT UINTN      *VariableNameSize,
  IN OUT CHAR16     *VariableName,
  IN OUT EFI_GUID   *VendorGuid
  );
```

在代码清单 5-21 中：

1）参数 *VariableNameSize 作为输入时是缓冲区 VariableName 的字节数；作为输出时，表示的是返回的变量名字符串占用的字节数（包括字符串结尾的 NULL 字符）。

2）参数 *VariableName 作为输入时上一个变量的名字；作为输出时，返回的是当前的变量名字。

3）参数 *VendorGuid 作为输入时上一个变量的 VendorGuid；作为输出时，是当前变量的 VendorGuid。

要启动搜索（即获得第一个变量），需使 VariableName 指向空字符串（即 VariableName 缓冲区第一个字符为 NULL）。到达变量数据库末尾时，函数返回 EFI_NOT_FOUND。

示例 5-9 演示了 GetNextVariableName 的用法。

【示例 5-9】 遍历系统变量。

```
//@file book\systemtable\gRT\RT.c
#include <Uefi.h>
#include <Library/UefiBootServicesTableLib.h>
#include <Library/UefiRuntimeServicesTableLib.h>
#include <Library/MemoryAllocationLib.h>
#include <Library/BaseMemoryLib.h>
#include <Library/UefiLib.h>
#define INIT_NAME_BUFFER_SIZE  128
#define INIT_DATA_BUFFER_SIZE  1024
EFI_STATUS UefiMain (IN EFI_HANDLE ImageHandle, IN EFI_SYSTEM_TABLE *SystemTable )
{
  EFI_STATUS Status;
  CHAR16 *FoundVarName, *OldName;
  UINT32 Atts;
  EFI_GUID FoundVarGuid;
  UINT8 *DataBuffer;
  UINTN DataSize,NameBufferSize,NameSize,OldNameBufferSize, DataBufferSize;
  NameBufferSize = INIT_NAME_BUFFER_SIZE;
  DataBufferSize = INIT_DATA_BUFFER_SIZE;
  //为变量名分配内存，并且必须初始化为 0（至少第一个字符必须为 0）
  FoundVarName   = AllocateZeroPool (NameBufferSize);
  if (FoundVarName == NULL) {
    return (EFI_OUT_OF_RESOURCES);
  }
  DataBuffer = AllocatePool (DataBufferSize);
  if (DataBuffer == NULL) {
    FreePool (FoundVarName);
    return (EFI_OUT_OF_RESOURCES);
  }
  for (;;){
    NameSize = NameBufferSize;
    Status = gRT->GetNextVariableName (&NameSize, FoundVarName,
      &FoundVarGuid);
    if (Status == EFI_BUFFER_TOO_SMALL) {
      //如果缓冲区不够大，那么在重新分配内存后重新执行 GetNextVariableName
      OldName = FoundVarName;
      OldNameBufferSize = NameBufferSize;
      //分配足够多的内存
      NameBufferSize=NameSize>NameBufferSize*2?NameSize:NameBufferSize*2;
      FoundVarName = AllocateZeroPool (NameBufferSize);
```

```
        if (FoundVarName == NULL) {
          Status = EFI_OUT_OF_RESOURCES;
          FreePool (OldName);
          break;
        }
        //将当前变量名字从旧缓冲区复制到新缓冲区
        CopyMem (FoundVarName, OldName, OldNameBufferSize);
        FreePool (OldName);
        NameSize = NameBufferSize;
        //重新执行 GetNextVariableName
        Status = gRT->GetNextVariableName (&NameSize, FoundVarName,
          &FoundVarGuid);
        }
        if (Status == EFI_NOT_FOUND) {
          //到达变量数据库末尾时退出循环
          break;
        }
        //找到一个新的变量，读取该变量的值和属性
        Print(L"%s\n",  FoundVarName);
        DataSize = DataBufferSize;
        Status = gRT->GetVariable (FoundVarName, &FoundVarGuid, &Atts,
          &DataSize, DataBuffer);
        if (Status == EFI_BUFFER_TOO_SMALL) {
          //错误处理
        }
      }//End for(;;)
      return Status;
}
```

3. 虚拟内存服务

虚拟内存服务包括 SetVirtualAddressMap 和 ConvertPointer，这两个服务只能在运行时期间被操作系统加载器调用，即调用了 gBS->ExitBootServices() 之后才能调用这两个服务。SetVirtualAddressMap 用于设置系统的内存映射。ConvertPointer 用于查询指定物理地址的虚拟地址。SetVirtualAddressMap 返回后，系统将从物理地址模式切换到虚拟地址模式。SetVirtualAddressMap 返回之前，所有 EVT_SIGNAL_VIRTUAL_ADDRESS_CHANGE 类型的事件都将触发，在这些事件的触发函数中需调用 ConvertPointer 将物理地址转换成虚拟地址。这几个主要函数的调用流程如下所示：

1）调用 gBS->GetMemoryMap 获得系统的内存映射。

2）调用 gBS->ExitBootServices 从 BS 转换到 RT 阶段。

3）此时内存映射中属性为 EfiBootServicesCode 和 EfiBootServicesData 的内存被释放，其他类型的内存将在 RT 阶段使用。

4）准备虚拟地址环境，生成新的内存映射表，并为内存映射表设置虚拟地址和物理地址映射。

5）调用 SetVirtualAddressMap。

6）SetVirtualAddressMap 返回前，所有 EVT_SIGNAL_VIRTUAL_ADDRESS_CHANGE 类型的事件都将触发，这些事件的触发函数被调用，在触发函数中需调用 ConvertPointer 将物理地址转换成虚拟地址（这些地址转换后才能在 RT 阶段使用）。

7）SetVirtualAddressMap 返回。系统进入虚拟地址模式。

代码清单 5-22 是 SetVirtualAddressMap 服务的函数原型。

代码清单 5-22　RT 的 SetVirtualAddressMap 服务函数原型

```
/** gRT->SetVirtualAddressMap
从物理地址模式转换为虚拟地址模式
  @retval EFI_SUCCESS
  @retval EFI_UNSUPPORTED                系统未运行在 RT 期，或系统已经运行在虚拟地址模式
  @retval EFI_NO_MAPPING                 内存映射表中需要提供虚拟地址的项未提供虚拟地址
  @retval EFI_NOT_FOUND                  内存映射表中存在无效物理地址
**/
typedef EFI_STATUS (EFIAPI *EFI_SET_VIRTUAL_ADDRESS_MAP)(
  IN UINTN MemoryMapSize,                //VirtualMap 数组的字节数
  IN UINTN DescriptorSize,               //VirtualMap 数组内每个数据项的字节数
  IN UINT32 DescriptorVersion,           //内存映射描述符的版本号
  IN EFI_MEMORY_DESCRIPTOR *VirtualMap   //内存映射描述符数组
  );
```

代码清单 5-23 是服务 ConvertPointer 的函数原型。*Address 作为输入参数时是物理地址，作为输出参数时返回对应的虚拟地址。ConvertPointer 通过查询系统的内存映射表计算出给定物理地址的虚拟地址。当 DebugDisposition 设置了 EFI_OPTIONAL_PTR 标志时，输入参数 *Address 允许为空。

代码清单 5-23　RT 的 ConvertPointer 服务函数原型

```
/**ConvertPointer 查询给定物理地址的虚拟地址
  @retval EFI_SUCCESS              成功查询到虚拟地址
  @retval EFI_INVALID_PARAMETER    Address 为空；或 *Address 为 NULL 并且
                                   DebugDisposition 没有设置 EFI_OPTIONAL_PTR 位
  @retval EFI_NOT_FOUND            指定的物理地址不存在
**/
typedef EFI_STATUS (EFIAPI *EFI_CONVERT_POINTER)(
  IN UINTN DebugDisposition,
  IN OUT VOID **Address
);
```

5.4 本章小结

本章讲述了 UEFI 提供的基础服务，也是 UEFI 开发中常用的三个数据结构：系统表、启动服务、运行时服务。

系统表以参数的形式从 UEFI 内核传递给应用程序和驱动程序。通过系统表，应用程序才可以得到启动服务、运行时服务、输入 / 输出设备以及系统配置表。

本章还介绍了启动服务和运行时服务提供给应用程序、驱动程序及操作系统的服务接口。了解启动服务的生存期对理解 UEFI 系统十分关键，读者一定要仔细体会其意义。

UEFI 应用程序和 UEFI 驱动运行在启动服务期。启动服务在开发中占有重要地位，本章讲述了启动服务中的内存管理服务，其他服务将在后续章节详细介绍。下一章我们将讲解 UEFI 事件的用法。

第 6 章 Chapter 6

事　件

UEFI 不再支持中断（准确地说，UEFI 不再为开发者提供中断支持，但在 UEFI 内部还是使用了时钟中断），所有的异步操作都要通过事件（Event）来完成。事件的重要性是不言而喻的。

先通过示例 6-1 来熟悉一下 Event 的作用，这段代码的作用是等待用户输入，然后根据用户输入执行指定的任务，如果用户在一定时间内没有任何输入，则执行默认任务。

【示例 6-1】 事件示例。

```
EFI_STATUS Task(EFI_KEY key);
EFI_STATUS TimeOut();
EFI_STATUS TestInput()
{
  EFI_STATUS  Status;
  EFI_EVENT myEvents[2];
  UINTN index=0;
  //生成定时器事件，并将事件对象赋值给 myEvents 数组的 0 号元素
  Status = gBS->CreateEvent(EVT_TIMER, TPL_CALLBACK, (EFI_EVENT_NOTIFY)NULL,
    (VOID*)NULL, &myEvent[0]);
  //设置定时器属性，1 秒 (10*1000*1000*100ns = 1s) 钟后到期
  Status = gBS->SetTimer(myEvent[0], TimerRelative, 10 * 1000 * 1000);
  myEvents[1] = gST->ConIn->WaitForKey; //myEvents 数组的 1 号事件为键盘事件
  while(1){
    //等待 myEvent 事件数组中任一事件发生，index 返回触发的事件在 myEvents 中的位置
    Status = gBS->WaitForEvent(2, myEvents, &index);
    //停止定时器事件计时
    Status = gBS->SetTimer(myEvent[0],TimerCancel, 10 * 1000 * 1000);
```

```
    if(index == 0){                                    //定时器事件，执行默认函数
      TimeOut();
    }else if(index == 1){                              //键盘事件发生
        EFI_INPUT_KEY  Key;
        Status = gST->ConIn->ReadKeyStroke (gST->ConIn, &Key);        //读取键盘
        Task(Key);
    }
    //重新启动定时器事件
  Status = gBS->SetTimer(myEvent[0], TimerRelative, 10 * 1000 * 1000);
  }
  return EFI_SUCCESS;
}
```

上面的程序展示了事件的基本用法。事件使得在等待键盘输入的过程中，CPU 资源不再浪费在无谓的等待上，大大提高了系统的性能。

启动服务为开发者提供了表 6-1 所示的函数，用于操作事件、定时器及 TPL（任务优先级）。这些函数可以分为三类：事件相关函数、定时器相关函数及 TPL 相关函数。

表 6-1　UEFI 事件服务

函数名	作　用
CreateEvent	生成一个事件对象
CreateEventEx	生成一个事件对象并将该事件加入到一个组内
CloseEvent	关闭事件对象
SignalEvent	触发事件对象
WaitForEvent	等待事件数组中的任一事件触发
CheckEvent	检查事件状态
SetTimer	设置定时器属性
RaiseTPL	提升任务优先级
RestoreTPL	恢复任务优先级

或许读者会感到疑惑，UEFI 不支持中断，那么为什么还要支持不同的任务优先级呢？这个问题我们会在后续内容中慢慢回答。当我们理解了事件机制后，这个问题的答案其实也就很清楚了。

下面就来讲解一下这三类函数的用法。

6.1　事件函数

启动服务中事件相关函数有 6 个，函数名大部分以 Event 结尾。提供给事件生产者的函数有 CreateEvent/CreateEventEx、SignalEvent 及 CloseEvent。提供给事件使用者的有 WaitForEvent 和 CheckEvent。对事件生产者来说，CreateEvent/CreateEventEx 是比较重要的函数。对事件使用者来说，WaitForEvent 是比较重要的函数。下面详细介绍一下这些函数。

6.1.1 等待事件的服务 WaitForEvent

WaitForEvent 用于等待事件的发生，类似于 Windows 提供的 WaitForMultipleObjects(...)。代码清单 6-1 是 WaitForEvent 的函数原型。

代码清单 6-1 启动服务中的 WaitForEvent 服务的函数原型

```
/** 等待 Event 数组内任一事件被触发
  @retval EFI_SUCCESS                    下标为 *index 的事件被触发
  @retval EFI_UNSUPPORTED                当前的 TPL 不是 TPL_APPLICATION
  @retval EFI_INVALID_PARAMETER          下标为 *index 的事件类型为 EVT_NOTIFY_SIGNAL
**/
typedef EFI_STATUS (EFIAPI *EFI_WAIT_FOR_EVENT) (
  IN UINTN NumberOfEvents,               //Event 数组内 Event 的个数
  IN EFI_EVENT *Event,                   //Event 数组
  OUT UINTN *Index                       //返回处于触发态的事件在数组内的下标
);
```

WaitForEvent 是阻塞操作，直到 Event 数组内任一事件被触发，或任一事件导致错误出现，WaitForEvent 才返回。WaitForEvent 从前到后依次检查 Event 数组内的事件，发现有被触发的事件或遇到错误则返回，如果所有事件都没有被触发，则从头开始重新检查。

当检查到某个事件处于触发态时，*Index 赋值为该事件在 Event 数组中的下标，返回前该事件将重置为非触发态。

当检查到某个事件是 EVT_NOTIFY_SIGNAL 类型时，*Index 赋值为该事件在 Event 数组中的下标，并返回 EFI_INVALID_PARAMETER。

WaitForEvent 必须运行在 TPL_APPLICAION 级别，否则将返回 EFI_UNSUPPORTED。

示例 6-2 展示了如何等待键盘按键事件的发生。

【示例 6-2】 等待按键事件。

```
UINTN EventIndex = 0;
gBS->WaitForEvent (1, &gST->ConIn->WaitForKey, &EventIndex);
```

示例 6-3 展示了如何等待按键并读取这个按键。

【示例 6-3】 等待并读取按键。

```
void WaitKey()
{
    EFI_STATUS   Status = 0;
    UINTN index=0;
    EFI_INPUT_KEY  Key;
    Status = gBS->WaitForEvent(1, &gST->ConIn->WaitForKey, &index);
    Status = gST->ConIn->ReadKeyStroke (gST->ConIn, &Key);
}
```

WaitForEvent 没有超时属性，如果想让 WaitForEvent 只等待一定的时间，则需要在事件等待数组加入定时器事件。示例 6-4 用于在 1 秒内等待键盘事件。

【示例 6-4】 WaitForEvent 超时示例。

```
EFI_STATUS Status;
UINTN EventIndex = 0;
EFI_EVENT Events[2] = {0};
Events[0]  =gST->ConIn->WaitForKey;
Status = gBS->CreateEvent(EVT_TIMER  , TPL_CALLBACK, (EFI_EVENT_NOTIFY)NULL,
    (VOID*)NULL, &Events[1]);
Status = gBS->SetTimer(Events[1],TimerPeriodic , 10 * 1000 * 1000);
Status = gBS->WaitForEvent (2, Events, &EventIndex);
if(EFI_SUCCESS == Status)
{
if(EventIndex == 0){
       //键盘有输入
}else if(EventIndex == 1){
       //1 秒内无键盘操作
}
}
```

6.1.2 生成事件的服务 CreateEvent

CreateEvent 用于生成一个事件。代码清单 6-2 是该函数的函数原型。

代码清单 6-2 启动服务中的 CreateEvent 服务的函数原型

```
//生成一个事件
typedef EFI_STATUS (EFIAPI *EFI_CREATE_EVENT) (
  IN UINT32 Type,                                  //事件类型
  IN EFI_TPL NotifyTpl,                            //事件 Notification 函数的优先级
  IN EFI_EVENT_NOTIFY NotifyFunction, OPTIONAL     //事件 Notification 函数
  IN VOID *NotifyContext, OPTIONAL                 //传给事件 Notification 函数的参数
  OUT EFI_EVENT *Event                             //生成的事件
);
```

1. 事件的类型

代码清单 6-3 定义了事件的基本类型，事件的类型可以是以下一种或几种基本类型的组合。例如，某个事件可以是 EVT_TIMER，也可以是 EVT_TIMER|EVT_NOTIFY_WAIT 或者 EVT_TIMER|EVT_NOTIFY_SIGNAL。

代码清单 6-3 事件类型的宏定义

```
#define EVT_TIMER                          0x80000000
#define EVT_RUNTIME                        0x40000000
```

```
#define EVT_NOTIFY_WAIT                          0x00000100
#define EVT_NOTIFY_SIGNAL                        0x00000200
#define EVT_SIGNAL_EXIT_BOOT_SERVICES            0x00000201
#define EVT_SIGNAL_VIRTUAL_ADDRESS_CHANGE        0x60000202
#define EVT_RUNTIME_CONTEXT                      0x20000000
```

表 6-2 列出了几种常用的事件类型。这些常用类型有基本事件类型，也有基本事件类型的组合。

表 6-2 常用事件类型

事件类型	事件特征
EVT_TIMER	定时器事件。普通 Timer 事件，没有 Notification 函数。生成事件后需调用 SetTimer 服务设置时钟属性。事件可以： ❑ 通过 SetTimer() 设置等待事件 ❑ 到期后通过 SignalEvent() 触发 ❑ 通过 WaitForEvent() 等待事件被触发 ❑ 通过 CheckEvent() 检查状态
EVT_NOTIFY_WAIT	普通事件。这个事件有一个 Notification 函数，当这个事件通过 CheckEvent() 检查状态或通过 WaitForEvent() 等待时，这个 Notification 函数会被放到待执行队列 gEventQueue[Event->NotifyTpl] 中
EVT_NOTIFY_SIGNAL	普通事件。这个事件有一个 Notification 函数，当这个事件通过 SignalEvent() 被触发时，这个 Notification 函数会被放到待执行队列 gEventQueue[Event->NotifyTpl] 中等待执行
0x00000000	普通事件。此类事件没有 Notification 函数。事件可以： ❑ 通过 SignalEvent() 被触发 ❑ 通过 WaitForEvent() 等待事件被触发 ❑ 通过 CheckEvent() 检查状态
EVT_TIMER | EVT_NOTIFY_WAIT	带 Notification 函数的定时器事件。此类事件除了具有 EVT_TIMER 的特性外，还有 EVT_NOTIFY_WAIT 的特性，即到期后通过 SignalEvent() 触发。 当事件通过 CheckEvent() 检查状态或通过 WaitForEvent() 等待时，这个 Notification 函数会被放到待执行队列 gEventQueue[Event->NotifyTpl] 中
EVT_TIMER | EVT_NOTIFY_SIGNAL	带 Notification 函数的定时器事件。此类事件除了具有 EVT_TIMER 的特性外，还有 EVT_NOTIFY_WAIT 的特性，即到期后通过 SignalEvent() 触发。 当事件通过 SignalEvent() 被触发时，这个 Notification 函数会被放到待执行队列 gEventQueue[Event->NotifyTpl] 中

还有两种特殊的事件，它们用在操作系统系统加载器从启动期向运行时期转换的过程中，这两种事件介绍如下。

1）**EVT_SIGNAL_EXIT_BOOT_SERVICES**：此类事件是一种特殊的 EVT_NOTIFY_SIGNAL，实际上它是 EVT_NOTIFY_SIGNAL 和 0x00000001 的组合。当 ExitBootServices() 执行时，事件被触发。EVT_SIGNAL_EXIT_BOOT_SERVICES 不能和其他类型混合使用。它的 Notification 函数和子函数不能使用启动服务中的内存分配服务；在 Notification 函数执行前所有的定时器服务都已失效，因而在 Notificaiton 函数中也不能使用定时器服务。

2）EVT_SIGNAL_VIRTUAL_ADDRESS_CHANGE：它是 EVT_RUNTIME_CONTEXT、EVT_RUNTIME、EVT_NOTIFY_SIGNAL 和 0x00000002 的组合。它不能和这 4 种类型之外的类型组合使用。当 SetVirtualAddressMap() 被调用时触发此类事件。

2. 优先级

CreateEvent 的第二个参数为 NotifyTPL（即任务优先级），它可以是 0 ~ 31 的一个整数。UEFI 预定义了以下 4 个优先级。

- #define TPL_APPLICATION 4
- #define TPL_CALLBACK 8
- #define TPL_NOTIFY 16
- #define TPL_HIGH_LEVEL 31

优先级别高的任务可以中断级别低的任务，并且从高优先级返回低优先级前会完成所有高于低优先级的任务。表 6-3 列出了 UEFI 预定义的 4 个任务优先级。

表 6-3 UEFI 预定义任务优先级

任务优先级	用　法	函　数
TPL_APPLICATION	这是预定义的 4 个级别中最低的一个优先级。应用程序运行（包括 Boot Manager 和 OS Loader）在这个级别。当程序运行在这个级别时，任务队列中没有任何处于就绪状态的事件 Notification 函数	下列函数运行在此级别： ExitBootServices()、WaitForEvent()、User Manager Protocol/Identify()、Form Browser2 Protocol/SendForm 下列函数运行在此级别或更低级别： Simple Input Protocol
TPL_CALLBACK	比较耗时的操作通常在这个优先级执行，如文件系统、磁盘操作等	下列函数运行在此级别或更低级别： Exit();Serial I/O Protocol、UnloadImage()、Variable Services、NetWork Service Binding、Network Protocol 下列函数运行在低于 8 的级别： LoadImage()、StartImage()
TPL_NOTIFY	运行在这个级别的程序不允许阻塞，必须尽快执行完毕并且返回。如果需要更多操作，则需要使用 Event 由内核重新调度。通常，底层的 IO 操作允许在这个级别，例如 UEFI 内核中读取键盘状态的代码。大部分 Event 的 Notification 函数允许在这个级别	下列函数运行在此级别或更低级别： Protocol Handler Services 、Memory Allocation Services、Simple Text Output Protocol、ACPI Table Protocol、User Manager Protocol、User Credential Protocol、User Info Protocol、Authentication Info、Device Path Utilities、Device Path From Text、EDID Discovered、EDID Active、Graphics Output EDID Override、iSCSI Initiator Name、Tape IO、Deferred Image Load Protocol、HII Protocols、Driver Health 下列函数运行在低于 16 的级别： ACPI Table Protocol

（续）

任务优先级	用 法	函 数
TPL_HIGH_LEVEL	优先级最高级别。在此级别，中断被禁止。UEFI 内核全局变量的修改需要允许在这个级别	下列函数运行在此级别或更低级别：SignalEvent()、Stall() 下列函数运行在低于 31 的级别：CheckEvent()、CloseEvent()、CreateEvent()、SetTimer()、Event Notification Levels 运行在 (TPL_APPLICATION，TPL_HIGH_LEVEL) 区间的优先级上

3. Notification 函数 NotifyFunction

CreateEvent 的第三个参数 NotifyFunction 是 EFI_EVENT_NOTIFY 类型的函数指针，它的函数原型如代码清单 6-4 所示。

代码清单 6-4 Notification 函数的函数原型

```
typedef VOID(EFIAPI *EFI_EVENT_NOTIFY)(
  IN  EFI_EVENT Event,            // 拥有此函数的事件
  IN  VOID *Context               // 上下文指针，此指针在 CreateEvent 时设置
);
```

如果事件的类型是 EVT_NOTIFY_WAIT，则 EFI_EVENT_NOTIFY 函数会在等待此事件的过程中调用；如果事件的类型是 EVT_NOTIFY_SIGNAL，则 EFI_EVENT_NOTIFY 函数会在事件触发时调用。既没有 EVT_NOTIFY_WAIT 属性也没有 EVT_NOTIFY_SIGNAL 属性的事件，Notification 参数将被忽略。

CreateEvent 的第 4 个参数是 NotifyContext，将在 Notification 函数被调用时作为第 2 个参数传递给该函数，用于指向这个 Notification 函数的上下文。

4. CreateEvent 示例

下面的示例用于生成一个简单的、没有 Notification 函数的事件。

【示例 6-5】 生成简单事件。

```
EFI_STATUS Status;
EFI_EVENT myEvent;
Status = gBS->CreateEvent(0, TPL_APPLICATION, (EFI_EVENT_NOTIFY)0, (VOID*) 0,
&myEvent);
```

下面的示例展示了如何使用 EVT_NOTIFY_WAIT 类型的事件。

【示例 6-6】 使用 EVT_NOTIFY_WAIT 类型的事件。

```
VOIDmyEventNoify (IN EFI_EVENT Event, IN VOID *Context)
```

```
{
    static UINTN times = 0;
    Print(L"myEventNotif Wait %d\n", times);
    times ++;
    if(times >5)
        gBS->SignalEvent(Event);
}
EFI_STATUS TestNotify()
{
    EFI_STATUS Status;
    UINTN index=0;
    EFI_EVENT myEvent;
Status = gBS->CreateEvent(EVT_NOTIFY_WAIT, TPL_NOTIFY,
         (EFI_EVENT_NOTIFY)myEventNoify, (VOID*)NULL, &myEvent);
    Status = gBS->WaitForEvent(1, &myEvent, &index);
    return EFI_SUCCESS;
}
```

在 WaitForEvent 的循环中，每检查一次 myEvent 的状态，myEventNotify 就执行一次。检查 6 次后，myEvent 被触发，从而 WaitForEvent 结束等待。下面是执行该示例程序时的输出。

```
f8:\>myEventNotif Wait 0
f8:\>myEventNotif Wait1
f8:\>myEventNotif Wait2
f8:\>myEventNotif Wait3
f8:\>myEventNotif Wait4
f8:\>myEventNotif Wait5
```

键盘按键事件 gST->ConIn->WaitForKey 正是这样一个 EVT_NOTIFY_WAIT 事件。当该事件等待时，WaitForKey 的 Notification 函数不断被调用，在这个 Notification 函数中会检查键盘设备是否有按键，有按键时触发 WaitForKey 事件。

EVT_NOTIFY_SIGNAL 类型的事件通常要配合定时器使用。EVT_NOTIFY_SIGNAL 事件的示例将在讲述了定时器事件后再给出。

6.1.3 CreateEventEx 服务

CreateEventEx 服务用于生成事件并将事件加入事件组。代码清单 6-5 是该服务的函数原型。

代码清单 6-5 BS 中的 CreateEventEx 服务的函数原型

```
// 为 EventGroup 生成一个 Event
typedef EFI_STATUS (EFIAPI *EFI_CREATE_EVENT_EX) (
  IN UINT32 Type,                                    // 事件类型
```

```
  IN EFI_TPL NotifyTpl,                          //事件Notification函数的优先级
  IN EFI_EVENT_NOTIFY NotifyFunction, OPTIONAL   //事件Notification函数
  IN VOID *NotifyContext, OPTIONAL               //传给事件Notification函数的参数
  IN CONST EFI_GUID *EventGroup OPTIONAL,        //事件组
  OUT EFI_EVENT *Event                           //生成的事件
);
```

由 CreateEventEx 生成的事件会加入到 EventGroup 中。当 EventGroup 中的任一事件被触发后，组中的所有其他事件都会被触发，进而同组内所有的 Notification 函数都将被加入到待执行队列。同组内 NotifyTpl（优先级）高的 Notification 函数会先被执行。

如果输入参数 EventGroup 为 NULL，则 CreateEventEx 退化为 CreateEvent。

Type 不能是 EVT_SIGNAL_EXIT_BOOT_SERVICES 或 EVT_SIGNAL_VIRTUAL_ADDRESS_CHANGE，因为这两种类型有各自对应的 Group，也就是说，

```
gBS -> CreateEvent(EVT_SIGNAL_EXIT_BOOT_SERVICES , TPL_NOTIFY,
myNotifyFunction, NULL, &myEvent);
```

与

```
gBS -> CreateEventEx(EVT_NOTIFY_SIGNAL, TPL_NOTIFY, myNotifyFunction, NULL,
&gEfiEventExitBootServicesGuid, &myEvent);
```

作用完全相同。

下面列出了 UEFI 预定义的 4 个 Event 组。

（1）EFI_EVENT_GROUP_EXIT_BOOT_SERVICES

GUID：gEfiEventExitBootServicesGuid。

ExitBootServices() 执行时触发该组内所有 Event。组内 Event 属性同于 EVT_SIGNAL_EXIT_BOOT_SERVICES 类型的 Event。

（2）EFI_EVENT_GROUP_VIRTUAL_ADDRESS_CHANGE

GUID：gEfiEventVirtualAddressChangeGuid。

SetVirtualAddressMap() 执行时触发该组内所有 Event。组内 Event 属性同于 EVT_SIGNAL_VIRTUAL_ADDRESS_CHANGE 类型的 Event。

（3）EFI_EVENT_GROUP_MEMORY_MAP_CHANGE

GUID：gEfiEventMemoryMapChangeGuid。

Memory Map 改变时触发该组内所有 Event。在 Notification 函数中不能使用启动服务的内存分配函数。

（4）EFI_EVENT_GROUP_READY_TO_BOOT

GUID：gEfiEventReadyToBootGuid。

Boot Manager 加载并且执行一个启动项时触发该组内所有 Event。

6.1.4 事件相关的其他函数

前面介绍了事件相关的两类比较重要的函数：WaitForEvent 和 CreateEvent/CreateEventEx。WaitForEvent 通常由事件的消费者调用。CreateEvent/CreateEventEx 通常由事件的生产者调用。相比其他函数来说，这三个函数最重要也最复杂。除了这三个函数，还有其他三个函数供生产者和消费者调用。生产者通常还要调用 SignalEvent 设置事件状态，程序或服务结束时调用 CloseEvent 关闭事件。消费者还可以使用 CheckEvent 查询事件的状态。下面简单介绍一下这三个函数。

（1）检查事件状态的服务 CheckEvent

CheckEvent 用于检测事件的状态。与 WaitForEvent 不同的是，CheckEvent 调用后立刻返回。代码清单 6-6 是 CheckEvent 服务的函数原型。

代码清单 6-6　BS 中的 CheckEvent 服务的函数原型

```
// 检查事件是否处于触发态
typedef EFI_STATUS (EFIAPI *EFI_CHECK_EVENT) ( IN EFI_EVENT Event);
```

根据事件的属性和状态，返回值有如下 4 种情况：

1）如果事件是 EVT_NOTIFY_SIGNAL 类型，则返回 EFI_INVALID_PARAMETER。

2）如果事件处于触发态，则返回 EFI_SUCCESS，并且在返回前，事件重置为非触发态。

3）如果事件处于非触发态并且事件无 Notification 函数，则返回 EFI_NOT_READY。

4）如果事件处于非触发态并且事件有 Notification 函数（此事件只可能是 EVT_NOTIFY_WAIT 类型），则执行 Notification 函数。然后，检查事件状态标志，若事件处于触发态，则返回 EFI_SUCCESS，否则返回 EFI_NOT_READY。

（2）触发事件的服务 SignalEvent

SignalEvent 用于将事件的状态设置为触发态。如果事件类型为 EVT_NOTIFY_SIGNAL，则将其 Notification 函数添加到就绪队列准备执行。如果该事件属于一个组，则将该组内所有事件都设置为触发态，并将组内所有 EVT_NOTIFY_SIGNAL 事件的 Notification 函数添加到就绪队列准备执行。

下面的代码是 SignalEvent 的函数原型。

```
typedef EFI_STATUS SignalEvent ( IN EFI_EVENT Event );
```

（3）关闭事件的服务 CloseEvent

事件使用完毕后，必须调用 CloseEvent 关闭这个事件。下面的代码是 CloseEvent 的函数原型。

```
// 关闭一个事件。CloseEvent 执行后，该事件从内核各个队列中清除，并释放内存
typedef EFI_STATUS CloseEvent ( IN EFI_EVENT Event);
```

通常的原则是由事件的所有者（即调用CreateEvent产生该事件的调用者）调用CloseEvent函数。调用该函数后，指定的事件将从内核中删除。

6.2 定时器事件

定时器是一类特殊的事件，生成定时器事件后，可以通过SetTimer服务设置定时器属性。代码清单6-7是SetTimer服务的函数原型。

代码清单6-7 BS中的SetTimer服务的函数原型

```
/** 设置定时器属性（类型及时间）
  @retval EFI_SUCCESS              属性设置成功
  @retval EFI_INVALID_PARAMETER  参数Event不是EVT_TIMER，或参数Type非法
**/
typedef EFI_STATUS (EFIAPI *EFI_SET_TIMER) (
  IN EFI_EVENT Event,              //Timer事件
  IN EFI_TIMER_DELAY Type,         //定时器类别
  IN UINT64 TriggerTime            //定时器过期时间，100ns为一个单位
);
```

Type是定时器类别，它可以是表6-4中的任意一个值。

表6-4 定时器类别

Type取值	作 用
TimerCancel	用于取消定时器触发时间。设置后定时器不再触发
TimerPeriodic	重复型定时器。每TriggerTime*100ns，定时器触发一次
TimerRelative	一次性定时器。TriggerTime*100ns时触发

- 如果Type为TimerPeriodic并且TriggerTime是0，则定时器每个时钟滴答触发一次。
- 如果Type为TimerRelative并且TriggerTime是0，则定时器在下个时钟滴答触发。

生成定时器事件一般分为两步：

第一步，通过CreateEvent生成一个EVT_TIMER事件。

第二步，通过SetTimer设置这个定时器事件的属性。

示例6-7展示了如何生成和使用一个简单的定时器事件。

【示例6-7】 定时器示例。

```
EFI_STATUS TestTimer()
{
  EFI_STATUS Status;
  EFI_EVENT myEvent;
```

```
  UINTN index=0;
  Print(L"Test EVT_TIMER | EVT_NOTIFY_SIGNAL");
  //第一步：生成 EVT_TIMER 事件
  Status = gBS->CreateEvent(EVT_TIMER, TPL_CALLBACK, (EFI_EVENT_NOTIFY)NULL,
  (VOID*)NULL, &myEvent);
  //第二步：设置定时器属性。该定时器每 10 秒触发一次
  Status = gBS->SetTimer(myEvent,TimerPeriodic , 10 * 1000 * 1000);
  while(1){
    Status = gBS->WaitForEvent(1, &myEvent, &index);
    //定时器到期
  }
  gBS->CloseEvent(myEvent);
  return EFI_SUCCESS;
}
```

在介绍 CreateEvent 时讲过，通常 EVT_NOTIFY_SIGNAL 会配合定时器使用。示例 6-8 展示了如何使用 EVT_NOTIFY_SIGNAL。在这个示例中，myEventNoify30(...) 每 10 秒自动执行一次，直到用户按下任意按键。

【示例 6-8】 带 Notification 函数的定时器示例。

```
VOID myEventNoify30 (IN EFI_EVENT Event, IN VOID *Context)
{
  static UINTN times = 0;
  Print(L"%s\n myEventNotif signal %d\n", Context, times);
  times++;
}
EFI_STATUS TestEventSingal()
{
  EFI_STATUS Status;
  EFI_EVENT myEvent;
  Print(L"Test EVT_TIMER | EVT_NOTIFY_SIGNAL");
  //生成 Timer 事件，并设置触发函数
  Status = gBS->CreateEvent(EVT_TIMER | EVT_NOTIFY_SIGNAL, TPL_CALLBACK,
  (EFI_EVENT_NOTIFY)myEventNoify30, (VOID*)L"Hello, Time Out!", &myEvent);
  //设置 Timer 等待时间为 10 秒，属性为循环等待
  Status = gBS->SetTimer(myEvent,TimerPeriodic , 10 * 1000 * 1000);
  WaitKey();
  Status = gBS->CloseEvent(myEvent);
  return EFI_SUCCESS;
}
```

6.3 任务优先级

UEFI 标准虽然不支持多线程，但是 UEFI 中有任务的概念。什么是任务呢？一个应用程序是一个任务，事件的 Notification 函数也是一个任务。UEFI 没有给开发者提供中断接

口，但 UEFI 内核的运行需要时钟中断的支持，时钟中断处理函数也是一个任务。不同的任务重要性不同。例如，时钟中断任务的重要性要大于定时器的 Notification 函数，而定时器的 Notification 函数的重要性大于普通应用程序。UEFI 为任务定义了任务级别，以便有限的计算机资源可以相对合理地分配给众多的任务。任务是如何调度的呢？任务的调度同事件有不可分割的联系，同时任务的调度也离不开时钟中断的支持。下面详细讲述任务优先级相关的两个服务以及 UEFI 中的时钟中断和任务调度。

6.3.1 提升和恢复任务优先级

RaiseTPL（NewTpl）用于提升当前任务的任务优先级至 NewTpl，该函数的返回值为原来的任务优先级。RestoreTPL 用于恢复（通常是降低）任务优先级至原来的优先级。下面是这两个服务的函数原型。

```
typedef EFI_TPL (EFIAPI *EFI_RAISE_TPL) ( IN EFI_TPL NewTpl );
// 恢复当前任务的任务优先级至原来的值 OldTpl
typedef VOID (EFIAPI *EFI_RESTORE_TPL) ( IN EFI_TPL OldTpl )
```

RaiseTPL 和 RestoreTPL 必须成对出现，执行了 RaiseTPL 后，必须尽快调用 RaiseTPL 将任务优先级恢复到原来的值。

当任务优先级提升至 TPL_HIGH_LEVEL 时，将关闭中断。当任务优先级从 TPL_HIGH_LEVEL 恢复到原来的（比 TPL_HIGH_LEVEL 低的）值时，中断被重新打开。

在任务优先级恢复到原优先级之前，所有高于原优先级的触发态事件的 Notification 函数都要执行完毕，详细过程将在下文讨论。

UEFI 是单 CPU 单线程系统，产生数据竞争的唯一可能来自中断处理函数，因而可以利用这一特性实现锁，这正是 UEFI 锁的实现机制。代码清单 6-8 是请求加锁的函数 CoreAcquireLock。

代码清单 6-8 CoreAcquireLock 函数源码

```
VOID CoreAcquireLock (  IN EFI_LOCK  *Lock  )
{
  ASSERT (Lock != NULL);
  ASSERT (Lock->Lock == EfiLockReleased);
  Lock->OwnerTpl = CoreRaiseTpl (Lock->Tpl);
  Lock->Lock = EfiLockAcquired;
}
struct EFI_LOCK {
  EFI_TPL Tpl;
  EFI_TPL OwnerTpl;
  EFI_LOCK_STATE Lock;
} gEventQueueLock = { TPL_HIGH_LEVEL, TPL_APPLICATION, EfiLockReleased};
```

```
VOID CoreAcquireEventLock ( VOID )
{
  CoreAcquireLock (&gEventQueueLock);
}
```

代码清单 6-9 为释放锁的函数 CoreReleaseLock。

代码清单 6-9 CoreReleaseLock 函数源码

```
VOID CoreReleaseLock (  IN EFI_LOCK  *Lock  )
{
  EFI_TPL Tpl;
  ASSERT (Lock != NULL);
  ASSERT (Lock->Lock == EfiLockAcquired);
  Tpl = Lock->OwnerTpl;
  Lock->Lock = EfiLockReleased;
  CoreRestoreTpl (Tpl);
}
VOID CoreReleaseEventLock (  VOID  )
{
  CoreReleaseLock (&gEventQueueLock);
}
```

6.3.2 UEFI 中的时钟中断

UEFI 用事件机制取代了 BIOS 中的中断机制，虽然 UEFI 不再提供中断接口，但其实现却离不开中断尤其是时钟中断。时钟中断是事件机制的基础。本节就来介绍时钟中断在 UEFI 中的作用以及 EDK2 是如何实现时钟中断处理的。

1. 时钟处理函数 CoreTimerTick

UEFI 的时钟处理函数是 CoreTimerTick，位于 MdeModulePkg\Core\Dxe\Event\Event.c 文件中。它在时钟中断中调用，是时钟中断处理函数的主体。该函数执行期间必须关中断并且不能被其他任何任务干扰，因而进入函数时需要加锁，离开函数时需要解锁。它的主要功能是维持系统时间，检查定时器事件列表中是否有到期的事件。代码清单 6-10 是 CoreTimerTick 函数的实现。

代码清单 6-10 时钟处理函数 CoreTimerTick

```
VOID EFIAPI CoreTimerTick (  IN UINT64   Duration  )
{
  IEVENT          *Event;
  CoreAcquireLock (&mEfiSystemTimeLock);        // 申请锁，获得该锁后会关闭中断
  // 任务一：更新系统时间
  mEfiSystemTime += Duration;
```

```
  //任务二：检查系统中的定时器事件是否到期
  if (!IsListEmpty (&mEfiTimerList)) {
    Event = CR (mEfiTimerList.ForwardLink, IEVENT, Timer.Link, EVENT_SIGNATURE);
    //检查定时器事件列表中的第一个事件是否到期，如果到期，则触发mEfiCheckTimerEvent事件，
    //该事件会检查并触发所有到期的定时器事件
    if (Event->Timer.TriggerTime <= mEfiSystemTime) {
      CoreSignalEvent (mEfiCheckTimerEvent);
    }
  }
  CoreReleaseLock (&mEfiSystemTimeLock); //释放锁并开中断
}
```

2. 设置时钟处理函数及安装时钟中断

下面来看一下函数 CoreTimerTick 是如何注册到时钟中断处理函数中的。

（1）安装时钟中断处理函数的时机

在 MdeModulePkg\Core\Dxe\DxeMain\DxeProtocolNotify.c 中有一个数组 mArchProtocols（见代码清单 6-11），它存放了体系结构相关的 Protocol，这个数组是 UEFI 系统 DXE 阶段的全局变量。

代码清单 6-11 系统 Protocol 表 mArchProtocols

```
typedef struct {
  EFI_GUID *ProtocolGuid;
  VOID **Protocol;
  EFI_EVENT Event;                    //当ProtocolGuid安装时，Event会触发
  //生成Event后，该Event会注册到内核中，以便ProtocolGuid安装时找到这个Event
  VOID *Registration;
  BOOLEAN Present;
} EFI_CORE_PROTOCOL_NOTIFY_ENTRY;
EFI_CORE_PROTOCOL_NOTIFY_ENTRY  mArchProtocols[] = {
  {&gEfiSecurityArchProtocolGuid,(VOID **)&gSecurity, NULL, NULL, FALSE},
  {&gEfiCpuArchProtocolGuid, (VOID **)&gCpu, NULL, NULL, FALSE},
  {&gEfiMetronomeArchProtocolGuid,(VOID **)&gMetronome, NULL, NULL, FALSE},
  {&gEfiTimerArchProtocolGuid, (VOID **)&gTimer, NULL, NULL, FALSE},
  {&gEfiBdsArchProtocolGuid, (VOID **)&gBds, NULL, NULL, FALSE},
  {&gEfiWatchdogTimerArchProtocolGuid,(VOID **)&gWatchdogTimer,NULL,NULL, FALSE},
  {&gEfiRuntimeArchProtocolGuid, (VOID **)&gRuntime, NULL, NULL, FALSE},
  {&gEfiVariableArchProtocolGuid, (VOID **)NULL, NULL, NULL, FALSE},
  {&gEfiVariableWriteArchProtocolGuid, (VOID **)NULL, NULL, NULL, FALSE},
  {&gEfiCapsuleArchProtocolGuid, (VOID **)NULL, NULL, NULL, FALSE},
  {&gEfiMonotonicCounterArchProtocolGuid, (VOID **)NULL, NULL, NULL, FALSE},
  {&gEfiResetArchProtocolGuid, (VOID **)NULL, NULL, NULL, FALSE},
  {&gEfiRealTimeClockArchProtocolGuid, (VOID **)NULL, NULL, NULL, FALSE},
  {NULL, (VOID **)NULL, NULL, NULL, FALSE}
};
```

系统初始化时会为 mArchProtocols 中的每个元素生成一个事件，当这个元素对应的 Protocol 安装时，该事件会触发，在事件的回调函数中会对该 Protocol 做相应的初始化。

例如，mArchProtocols[3] 为 {&gEfiTimerArchProtocolGuid，（VOID**）&gTimer，NULL，NULL，FALSE}，是 EFI_TIMER_ARCH_PROTOCOL 对应的 EFI_CORE_PROTOCOL_NOTIFY_ENTRY 项。函数 CoreNotifyOnProtocolInstallation（见代码清单 6-12）执行后，mArchProtocols[3] 为 {&gEfiTimerArchProtocolGuid，（VOID**）&gTimer，timerEvent，Registration，FALSE}，TimerEvent 的 Notification 函数被 CoreRegisterProtocolNotify 函数注册到系统。

代码清单 6-12　CoreNotifyOnProtocolInstallation 函数

```
// 为 mArchProtocols 和 mOptionalProtocols 中的每个 Protocol 生成一个事件
VOID CoreNotifyOnProtocolInstallation ( VOID )
{
  CoreNotifyOnProtocolEntryTable (mArchProtocols);
  CoreNotifyOnProtocolEntryTable (mOptionalProtocols);
}
// 这个函数的功能是为 Entry 中的 Protocol 生成一个事件并注册（当 Protocol 安装时触发这个事件）
VOID CoreNotifyOnProtocolEntryTable(EFI_CORE_PROTOCOL_NOTIFY_ENTRY  *Entry)
{
  EFI_STATUS  Status;
  for (; Entry->ProtocolGuid != NULL; Entry++) {
    // 为对应的 Protocol 生成事件，并将 GenericProtocolNotify 设置为响应该事件的回调函数
    Status = CoreCreateEvent ( EVT_NOTIFY_SIGNAL,
            TPL_CALLBACK,
            GenericProtocolNotify,
            Entry,
            &Entry->Event);
    ASSERT_EFI_ERROR(Status);
    // 为 Protocol 注册通知函数，当 Entry->ProtocolGuid 对应的 Protocol 安装时，
    // Entry->Event 会触发
    Status = CoreRegisterProtocolNotify ( Entry->ProtocolGuid,
            Entry->Event,
            &Entry->Registration);
    ASSERT_EFI_ERROR(Status);
  }
}
```

介绍 CoreRegisterProtocolNotify 之前，先来看一下 Protocol 在 UEFI 内核中的表示。图 6-1 展示了 Protocol 是如何组织起来的。

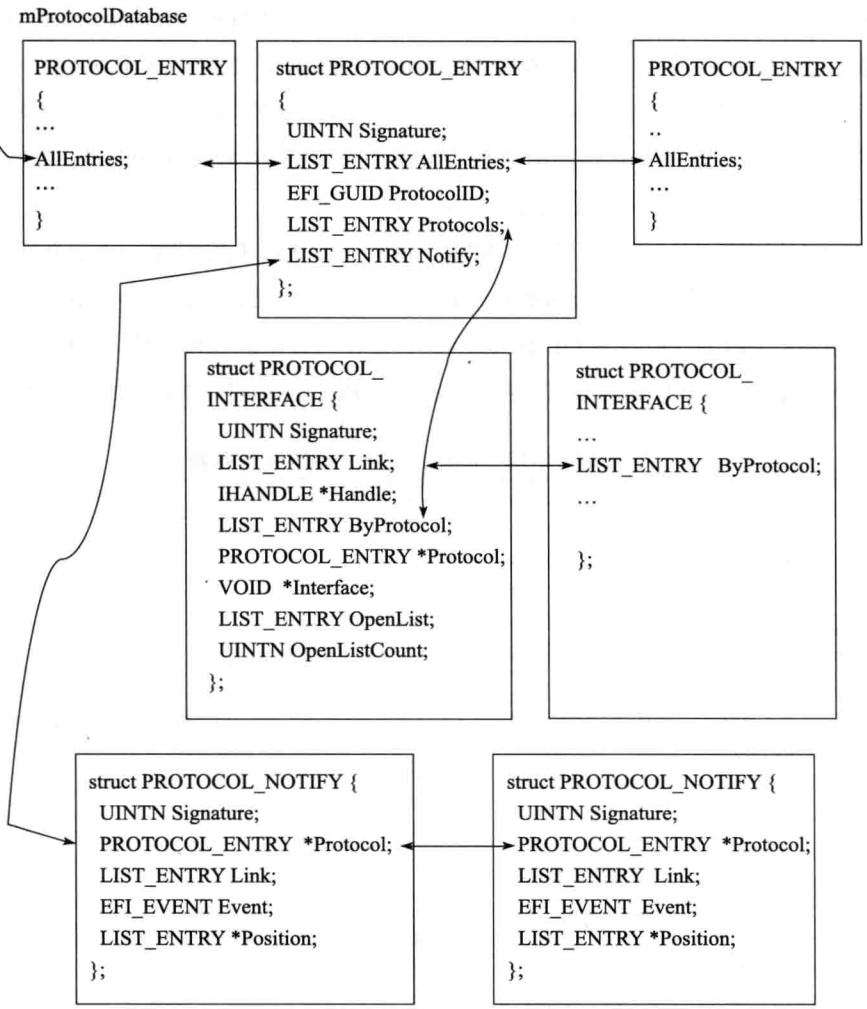


图 6-1 Protocol 在内核中的组织

所有的 Protocol 均放在 mProtocolDatabase 指向的 PROTOCOL_ENTRY 链表中。PROTOCOL_ENTRY 包含三个链表。

AllEntries 是 PROTOCOL_ENTRY 链。

Protocols 指向此 Protocol 的所有实例。例如，系统中可能有很多块设备，每个块设备都有一个 EFI_BLOCK_IO_PROTOCOL 实例，所有的 EFI_BLOCK_IO_PROTOCOL 实例都会插入到 PROTOCOL_ENTRY.Protocols 链表中。

Notify 指向 PROTOCOL_NOTIFY 链表，当 PROTOCOL_ENTRY.ProtocolID 对应的 Protocol 安装时，Notify 链表中所有 Event 都会触发。

理解了 Protocol 在内核中的组织结构，再来看 CoreRegisterProtocolNotify(…) 函数的主要功能。其主要功能是在 mProtocolDatabase 数据库中注册 PROTOCOL_NOTIFY。当 Protocol 安装时，会检查该 Protocol 对应的 PROTOCOL_ENTRY.Notify。如果 PROTOCOL_ENTRY.Notify 存在，则触发它指向的事件，相关的源代码在 MdeModulePkg\Core\Dxe\Hand\Handle.c 文件的 CoreInstallProtocolInterfaceNotify 函数中。

现在回到时钟中断这个主题。当 EFI_TIMER_ARCH_PROTOCOL 安装时，mArchProtocols[3].Event 事件会触发，然后这个事件的响应函数 GenericProtocolNotify 会执行，在 GenericProtocolNotify 中通过 EFI_TIMER_ARCH_PROTOCOL 的 RegisterHandler 时钟中断处理函数。代码清单 6-13 展示了 GenericProtocolNotify 是如何响应 EFI_TIMER_ARCH_PROTOCOL 被安装事件的。

代码清单 6-13　GenericProtocolNotify 函数

```
VOID EFIAPI GenericProtocolNotify(IN  EFI_EVENT  Event,  IN  VOID *Context)
{
  EFI_CORE_PROTOCOL_NOTIFY_ENTRY  *Entry;
  Entry = (EFI_CORE_PROTOCOL_NOTIFY_ENTRY *)Context;
  //该函数被调用时，说明 Entry->ProtocolGuid
  //通过 gBS->InstallMultipleProtocolInterfaces(…) 安装了，
  //现在打开这个 Protocol
  Status = CoreLocateProtocol (Entry->ProtocolGuid, Entry->Registration,
    &Protocol);
  if (EFI_ERROR (Status)) {
    return;
  }
  //Protocol 打开成功，设置存在标志
  Entry->Present = TRUE;
  //将 Protocol 的 Handle 赋值给对应的全局变量。例如，如果 Entry->ProtocolGuid 是
  //gEfiTimerArchProtocolGuid，则 Protocol 会赋值给 gTimer
  if (Entry->Protocol != NULL) {
    *(Entry->Protocol) = Protocol;
  }
  //如果该事件对应于 EFI_TIMER_ARCH_PROTOCOL
  if (CompareGuid (Entry->ProtocolGuid, &gEfiTimerArchProtocolGuid)) {
    //注册时钟中断处理函数 CoreTimerTick
    gTimer->RegisterHandler (gTimer, CoreTimerTick);
  }
  …
}
```

（2）向 gTimer 注册 CoreTimerTick 函数

下面我们看一下 gTimer->RegisterHandler(gTimer，CoreTimerTick)。gTimer 是 EFI_TIMER_ARCH_PROTOCOL* 类型的全局变量。EFI_TIMER_ARCH_PROTOCOL 的实现位于 PcAt

ChipsetPkg\8254TimerDxe，代码如下所示。gTimer 是指向 mTimer 的指针。

```
EFI_TIMER_ARCH_PROTOCOL   mTimer = {
  TimerDriverRegisterHandler,        //RegisterHandler
  TimerDriverSetTimerPeriod,         //SetTimerPeriod
  TimerDriverGetTimerPeriod,         //GetTimerPeriod
  TimerDriverGenerateSoftInterrupt   //GenerateSoftInterrupt
};
```

可以看出，gTimer->RegisterHandler 这个函数指针指向了函数 TimerDriverRegisterHandler。该函数源码如代码清单 6-14 所示。

代码清单 6-14　TimerDriverRegisterHandler 函数源码

```
EFI_STATUS EFIAPI TimerDriverRegisterHandler(IN EFI_TIMER_ARCH_PROTOCOL *This,
  IN EFI_TIMER_NOTIFY NotifyFunction)
{
  //检查参数
  if (NotifyFunction == NULL && mTimerNotifyFunction == NULL) {
    return EFI_INVALID_PARAMETER;
  }
  if (NotifyFunction != NULL && mTimerNotifyFunction != NULL) {
    return EFI_ALREADY_STARTED;
  }
  //设置时钟中断响应函数
  mTimerNotifyFunction = NotifyFunction;
  return EFI_SUCCESS;
}
```

可以猜想 mTimerNotifyFunction 这个函数指针将在时钟中断处理函数中被调用，那么 mTimerNotifyFunction 是如何被调用的呢？它在函数 TimerInterruptHandler 中被调用。代码清单 6-15 是 TimerInterruptHandler 的源码。

代码清单 6-15　TimerInterruptHandler 函数源码

```
VOID EFIAPI TimerInterruptHandler (
  IN EFI_EXCEPTION_TYPE   InterruptType,
  IN EFI_SYSTEM_CONTEXT   SystemContext)
{
  EFI_TPL OriginalTPL;
  //将任务优先级提升至 TPL_HIGH_LEVEL，在 TPL_HIGH_LEVEL 时会关闭时钟中断
  OriginalTPL = gBS->RaiseTPL (TPL_HIGH_LEVEL);
  mLegacy8259->EndOfInterrupt (mLegacy8259, Efi8259Irq0);
  if (mTimerNotifyFunction != NULL) {
    //调用 mTimerNotifyFunction 指向的函数，即 CoreTimerTick 函数
    mTimerNotifyFunction (mTimerPeriod);
  }
```

```
  //恢复任务优先级
  gBS->RestoreTPL (OriginalTPL);
}
```

从 mTimerNotifyFunction 可以看出，mTimerNotifyFunction(mTimerPeriod) 将会运行在最高任务优先级 TPL_HIGH_LEVEL。mTimerNotifyFunction 也就是我们通过“gTimer->RegisterHandler (gTimer, CoreTimerTick);”设置的 CoreTimerTick 函数。

（3）向 CPU 注册中断处理函数 TimerInterruptHandler

TimerInterruptHandler 是真正的时钟中断处理函数吗？下面我们继续追踪 TimerInterrupt Handler 是如何被调用的。不难发现，它在函数 TimerDriverInitialize（见代码清单 6-16）中被调用。

代码清单 6-16　TimerDriverInitialize 函数源码

```
/** PcAtChipsetPkg\8254TimerDxe\Timer.c*/
EFI_CPU_ARCH_PROTOCOL  *mCpu = NULL;
EFI_STATUS EFIAPI TimerDriverInitialize (
  IN EFI_HANDLE        ImageHandle,
  IN EFI_SYSTEM_TABLE  *SystemTable)
{
  EFI_STATUS  Status;
  UINT32      TimerVector;
  ...
  //打开 CPU architectural protocol
  Status = gBS->LocateProtocol (&gEfiCpuArchProtocolGuid, NULL, (VOID **) &mCpu);
  //打开 the Legacy8259 protocol
  Status = gBS->LocateProtocol (&gEfiLegacy8259ProtocolGuid, NULL,
    (VOID **) &mLegacy8259);
  //找到 8259 芯片中 IRQ0 对应的中断号，这个中断号将会用于时钟中断
  Status = mLegacy8259->GetVector(mLegacy8259, Efi8259Irq0,
      (UINT8 *) &TimerVector);
  //注册 8254 时钟中断
  Status = mCpu->RegisterInterruptHandler(mCpu,
      TimerVector,TimerInterruptHandler);
  ASSERT_EFI_ERROR (Status);
...
}
```

在 TimerDriverInitialize 中，最终通过 EFI_CPU_ARCH_PROTOCOL 的 RegisterInterruptHandler 注册了 TimerInterruptHandler。另外，在中断处理函数中将会调用 TimerInterruptHandler，而 TimerInterruptHandler 又会调用 mTimerNotifyFunction，即 CoreTimerTick。

EFI_CPU_ARCH_PROTOCOL 的实现位于 UefiCpuPkg\CpuDxe\CpuDxe.c，如代码清单 6-17 所示。

代码清单 6-17 gCpu 的实现

```
EFI_CPU_ARCH_PROTOCOL  gCpu = {
  CpuFlushCpuDataCache,          // FlushDataCache
  CpuEnableInterrupt,            // EnableInterrupt
  CpuDisableInterrupt,           // DisableInterrupt
  CpuGetInterruptState,          // GetInterruptState
  CpuInit,                       // Init
  CpuRegisterInterruptHandler,   // RegisterInterruptHandler
  CpuGetTimerValue,              // GetTimerValue
  CpuSetMemoryAttributes,        // SetMemoryAttributes
  1,                             // NumberOfTimers
  4                              // DmaBufferAlignment
};
```

可以看出，mCpu->RegisterInterruptHandler 将会调用 CpuRegisterInterruptHandler 函数（见代码清单 6-18）。

代码清单 6-18 CpuRegisterInterruptHandler 函数源码

```
EFI_STATUS EFIAPI CpuRegisterInterruptHandler (
  IN EFI_CPU_ARCH_PROTOCOL          *This,
  IN EFI_EXCEPTION_TYPE             InterruptType,
  IN EFI_CPU_INTERRUPT_HANDLER      InterruptHandler
  )
{
  ...
  if (InterruptHandler != NULL) {
    // 设置中断 InterruptType 的中断处理函数为默认函数
    SetInterruptDescriptorTableHandlerAddress ((UINTN)InterruptType, NULL);
  } else {
    RestoreInterruptDescriptorTableHandlerAddress ((UINTN)InterruptType);
  }
  // 保存函数指针 InterruptHandler,
  // ExternalVectorTable[InterruptType] 将在中断处理函数中调用
  ExternalVectorTable[InterruptType] = InterruptHandler;
  return EFI_SUCCESS;
}
```

CpuRegisterInterruptHandler 主要做了两件事情，一是通过 SetInterruptDescriptorTableHandlerAddress（见代码清单 6-19）将 InterruptType 的中断处理函数设为默认函数，二是保存传入的函数指针 InterruptHandler 到 ExternalVectorTable[InterruptType] 中。

代码清单 6-19 SetInterruptDescriptorTableHandlerAddress 函数源码

```
VOID SetInterruptDescriptorTableHandlerAddress(IN UINTN Index, IN VOID *Handler
OPTIONAL)
{
  UINTN UintnHandler;
```

```
  if (Handler != NULL) {
    UintnHandler = (UINTN) Handler;
  } else {
    // 如果 Handler 为 NULL，则使用默认的中断处理函数
    UintnHandler = ((UINTN) AsmIdtVector00) + (8 * Index);
  }
   // 设置 Index 对应的中断门描述符
  gIdtTable[Index].Bits.OffsetLow   = (UINT16)UintnHandler;
  gIdtTable[Index].Bits.Reserved_0  = 0;
  gIdtTable[Index].Bits.GateType    = IA32_IDT_GATE_TYPE_INTERRUPT_32;
  gIdtTable[Index].Bits.OffsetHigh  = (UINT16)(UintnHandler >> 16);
#if defined (MDE_CPU_X64)
  gIdtTable[Index].Bits.OffsetUpper = (UINT32)(UintnHandler >> 32);
  gIdtTable[Index].Bits.Reserved_1  = 0;
#endif
}
```

（4）在 CPU 时钟中断向量中调用时钟中断处理函数

默认的中断向量在 UefiCpuPkg\CpuDxe\Ia32\IvtAsm.S 中定义，其主要功能是调用 CommonInterruptEntry。代码清单 6-20 是默认中断向量源码。

代码清单 6-20　默认中断向量

```
; UefiCpuPkg\CpuDxe\Ia32\ IvtAsm.S
PUBLIC     AsmIdtVector00
AsmIdtVector00 LABEL BYTE
REPEAT  256
    call     CommonInterruptEntry
    dw       ($ - AsmIdtVector00 - 5) / 8 ; vector number
    nop
ENDM
```

CommonInterruptEntry 定义在 UefiCpuPkg\CpuDxe\[Ia32|x64]\CpuAsm.S 中，该函数主要完成以下任务。

1）保存寄存器。

2）调用 ExternalVectorTable[InterruptType]。

3）恢复寄存器，从中断处理返回。

在时钟中断向量中，ExternalVectorTable[InterruptType] 指向函数 TimerInterruptHandler。

下面简单回顾一下时钟中断的执行过程以及注册时钟中断处理函数的执行过程。图 6-2 展示了时钟中断的执行过程。

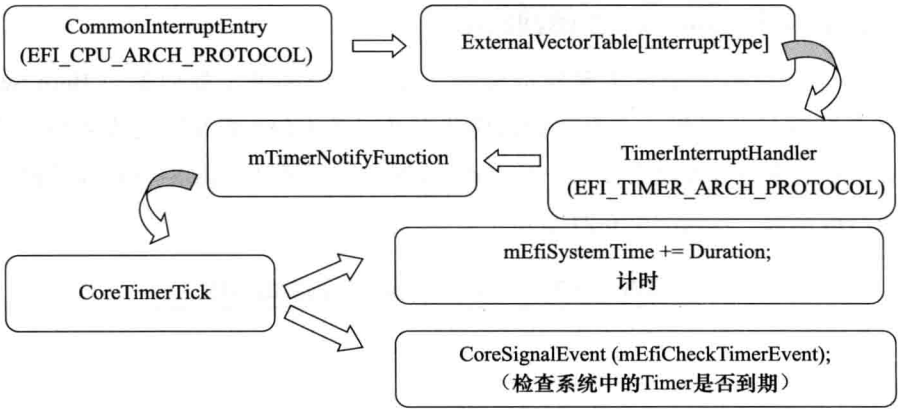


图 6-2 时钟中断的执行过程

图 6-3 展示了注册时钟中断函数和注册时钟处理函数的过程。

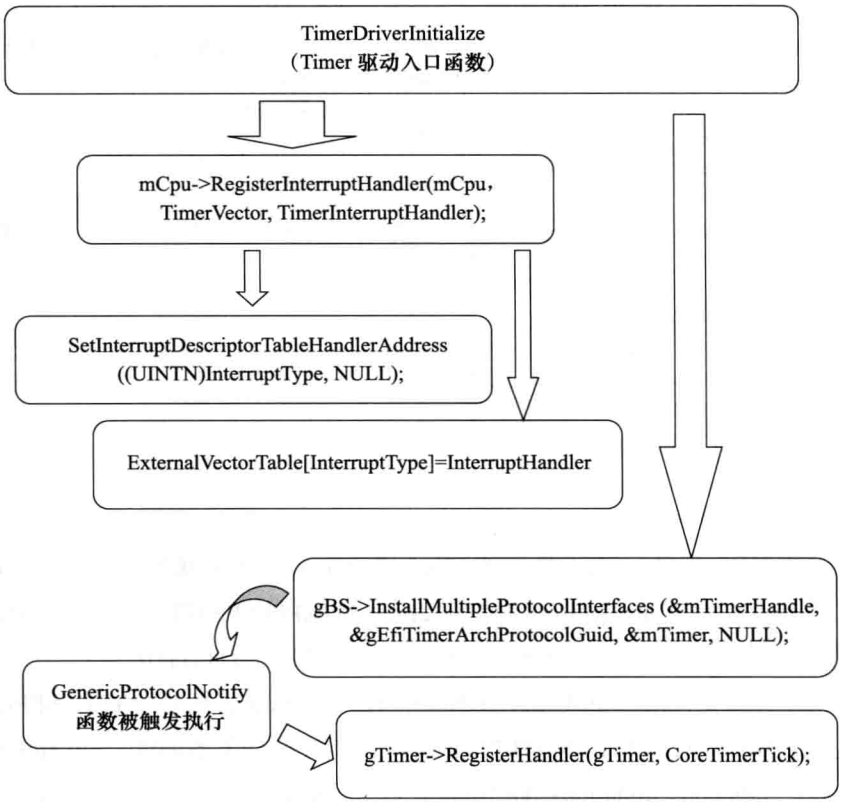


图 6-3 向时钟中断注册时钟处理函数的过程

6.3.3 UEFI 事件 Notification 函数的派发

Event 的一个重要作用是实现异步操作，前面已经讲解了如何通过 Boot Services 的 Event 的服务实现异步操作。下面我们来深入分析一下 UEFI 是如何完成异步操作的。事件 Notification 函数的派发是在 gBS->RestoreTpl 服务中完成的。gBS->RestoreTpl 实际指向 CoreRestoreTpl 函数，代码清单 6-21 展示了其实现。

代码清单 6-21 Core RestoreTpl 函数的实现

```
VOID EFIAPI CoreRestoreTpl (  IN EFI_TPL NewTpl  )
{
  EFI_TPL     OldTpl;
  OldTpl = gEfiCurrentTpl;
  ASSERT (NewTpl <= OldTpl);
  ASSERT (VALID_TPL (NewTpl));
  if (OldTpl >= TPL_HIGH_LEVEL  &&  NewTpl < TPL_HIGH_LEVEL) {
    gEfiCurrentTpl = TPL_HIGH_LEVEL;
  }
  while (((-2 << NewTpl) & gEventPending) != 0) { //事件列表中有高优先级的任务就绪
    //gEfiCurrentTpl 赋值为 gEventPending 中的为 1 的最高位
    gEfiCurrentTpl = (UINTN) HighBitSet64 (gEventPending);
    if (gEfiCurrentTpl < TPL_HIGH_LEVEL) {
      CoreSetInterruptState (TRUE);
}
    //执行 gEventQueue[gEfiCurrentTpl] 队列中的所有 Event 的 Notification 函数
    CoreDispatchEventNotifies (gEfiCurrentTpl);
  }
  //设置新的任务优先级
  gEfiCurrentTpl = NewTpl;
  //如果优先级小于 TPL_HIGH_LEVEL，打开中断
  if (gEfiCurrentTpl < TPL_HIGH_LEVEL) {
    CoreSetInterruptState (TRUE);
  }
}
```

在上面的代码中，((-2 << NewTpl) & gEventPending) != 0 意味着事件列表中有高优先级的任务等待执行。-2 也就是 0XFFFFFFF7（32bit）或 0xFFFFFFFFFFFFFFF7（64bit）。以 32 位 CPU 为例，用二进制表示就是 11111111 11111111 11111111 11111110。

下面以 NewTpl 是 TPL_APPLICATION(4)、gEventPending 是 (2<<TPL_APPLICATION)|(2<<TPL_CALLBACK)|(2<<TPL_NOTIFY) 的情况为例介绍任务的调度。gEventPending 二进制 为 00000000 00000001 00000001 00010000。-2 << NewTpl 是 11111111 11111111 11111111 11100000。（-2<<NewTpl）& gEventPending 是 00000000 00000001 00000001 00000000。 结果中为 1 的位表示该优先级的事件列表中有就绪的任务。可以看出，此时系统中有 TPL_

CALLBACK 和 TPL_NOTIFY 两种优先级的任务需要执行。调度优先级高于 NewTpl 并且处于触发状态的 Event，按 TPL 从高到低的顺序依次执行。本例中首先调度所有任务优先级为 TPL_NOTIFY(值为 16）的任务，然后调度优先级为 TPL_CALLBACK(值为 8）的所有任务。

时钟中断每 10ms（DEFAULT_TIMER_TICK_DURATION）中断一次。进入时钟中断时，会调用 RaiseTpl，将优先级提升至 TPL_HIGH_LEVEL，中断调用返回时，会调用 RestoreTpl，将优先级恢复，因而中断调用返回后，所有高于原优先级的 Notification 都已被执行。在 WaitForEvent 函数中，同样会调用 RestoreTpl 函数，造成高优先级的任务被调度。

6.4 鼠标和键盘事件示例

下面用一个完整的示例展示如何使用鼠标事件和键盘事件。在示例 6-9 中，每当鼠标事件发生时（鼠标移动或鼠标按键），打印鼠标状态；按〈q〉键退出函数。

【示例 6-9】 使用鼠标和键盘事件。

```
EFI_STATUS testMouseSimple()
{
  EFI_STATUS Status;
  EFI_SIMPLE_POINTER_PROTOCOL* mouse = 0;
  EFI_SIMPLE_POINTER_STATE State;
  EFI_EVENT events[2]; //= {0, gST->ConIn->WaitForKey};
  //显示光标
  gST->ConOut->EnableCursor (gST->ConOut, TRUE);
  //找出鼠标设备
  Status = gBS->LocateProtocol(&gEfiSimplePointerProtocolGuid,
    NULL,(VOID**)&mouse);
  //重置鼠标设备
  Status = mouse->Reset(mouse, TRUE);
  //将鼠标事件放到等待事件数组
  events[0] = mouse->WaitForInput;
  //将键盘事件放到等待数组
  events[1] = gST->ConIn->WaitForKey;
  while(1){
    EFI_INPUT_KEY    Key;
    UINTN index;
    //等待 events 中的任一事件发生
    Status = gBS->WaitForEvent(2, events, &index);
    if(index == 0){
      //获取鼠标状态并输出
      Status = mouse->GetState(mouse, &State);
      Print(L"X:%d Y:%d Z:%d L:%d R:%d\n",
        State.RelativeMovementX,
        State.RelativeMovementY,
        State.RelativeMovementZ,
```

```
        State.LeftButton,
        State.RightButton);
    } else{
      Status = gST->ConIn->ReadKeyStroke (gST->ConIn, &Key);
        //按' q' 键退出
        if (Key.UnicodeChar == 'q')
          break;
    }
  }
  return EFI_SUCCESS;
}
```

本示例只是在文本模式下打印鼠标的状态，并没有显示鼠标。关于如何显示鼠标，将在第 12 章讲解。

6.5 本章小结

事件是 UEFI 中的一个重要概念，它是异步操作的基础设施，取代了中断这种传统的耗时的操作方式，提高了系统的性能和效率。本章详细讲述了事件的实现原理。虽然 UEFI 用事件取代了中断，但还是保留了时钟中断。在时钟中断处理函数中，UEFI 内核会检查系统中的定时器事件并处理到期的定时器事件，并在合适的时机调度事件的 Notification 函数，可以说事件的实现基础是时钟中断。

本章还介绍了事件的使用，包括事件的生成（CreateEvent）、事件的等待（WaitForEvent）以及事件的触发（SignalEvent）。事件的使用在 UEFI 开发中不可回避。正确地使用事件可以大大提高程序的性能，尤其是在程序中需要访问外部设备时。下一章我们将讲述 UEFI 计算机系统中比较重要的外部设备，即硬盘。

第 7 章 Chapter 7

硬盘和文件系统

硬盘是计算机系统极其重要的一个部分，对当今大部分计算机来讲，操作系统和操作系统加载器是存放在硬盘内的。固件读写硬盘的机制将极大地影响系统的启动速度。相对于 BIOS，UEFI 对硬盘的支持有明显的进步，主要体现在如下几个方面：

1）支持 GPT 硬盘。

2）支持 FAT 文件系统。

3）支持异步硬盘 / 文件读取。

下面将详细介绍 GPT 硬盘的格式、硬盘读写方法，以及文件系统和文件的读写方式。

7.1 GPT 硬盘

20 世纪末期，Legacy BIOS+MBR 分区一直“统治”着计算机系统。但是，随着计算机硬件的发展，它们暴露出了越来越多的缺点。20 世纪 90 年代末，英特尔（Inter）公司开发了 GPT 分区格式以弥补这些不足，但当时硬件发展水平远没有达到 Legacy BIOS+MBR 分区的极限，因而没有广泛流行。但当 64 位系统逐渐取代 32 位系统、硬盘容量超过 2TB 后，这些缺点实实在在地阻碍了计算机的发展，Legacy BIOS+MBR 分区也就终于完成了它们的使命，让位于 UEFI+GPT 的组合。先来回顾一下 MBR 分区及基于 MBR 的硬盘相关内容。

7.1.1 基于 MBR 分区的传统硬盘

MBR 的全称是主引导记录（Master Boot Record），是硬盘的首个扇区（也称为 0 号扇

区)。0 号扇区用于存放启动代码和主分区表。0 ~ 439 字节为启动代码; 446 ~ 509 为主分区表。主分区表区域可容纳最多 4 个分区表项。最后 2 字节是结束标志。图 7-1 展示了 0 号扇区的格式。图 7-2 展示了分区表项的格式。

<table>
<tr><th>偏移
偏移</th><th>0</th><th>1</th><th>2</th><th>3</th><th>4</th><th>5</th><th>6</th><th>7</th></tr>
<tr><td>0</td><td colspan="8" rowspan="3">启动代码</td></tr>
<tr><td>⋮</td></tr>
<tr><td>432</td></tr>
<tr><td>440</td><td colspan="4">唯一 MBR 磁盘标识符</td><td colspan="2">0</td><td colspan="2" rowspan="2"></td></tr>
<tr><td>448</td><td colspan="6" rowspan="2">分区 1</td></tr>
<tr><td>456</td><td colspan="2"></td></tr>
<tr><td>464</td><td colspan="6" rowspan="2">分区 2</td><td colspan="2"></td></tr>
<tr><td>472</td><td colspan="2" rowspan="2"></td></tr>
<tr><td>480</td><td colspan="6" rowspan="2">分区 3</td></tr>
<tr><td>488</td><td colspan="2"></td></tr>
<tr><td>496</td><td colspan="6" rowspan="2">分区 4</td><td colspan="2"></td></tr>
<tr><td>504</td><td>0x55</td><td>0xAA</td></tr>
</table>

图 7-1 MBR 硬盘 0 号扇区格式

<table>
<tr><th>偏移
偏移</th><th>0</th><th>1</th><th>2</th><th>3</th><th>4</th><th>5</th><th>6</th><th>7</th></tr>
<tr><td>0</td><td>引导标志</td><td colspan="3">起始 CHS</td><td colspan="2">分区类型描述</td><td colspan="3">结束 CHS</td></tr>
<tr><td>8</td><td colspan="4">起始逻辑扇区号</td><td colspan="4">扇区个数</td></tr>
</table>

图 7-2 MBR 硬盘分区表项格式

从分区表项格式可以看出，它带有浓重的 32 位体系结构的印记。在每个分区表项中，用 4 个字节（StartingLBA）表示该分区的起始扇区号，4 个字节（SizeInLBA）表示该分区拥有的总扇区数，因而 MBR 分区最多可用到硬盘的前 4G 个扇区（4G 扇区数 ×512B 每扇区 =2TB)。BootIndicator 是分区状态,00 为非活动分区,80 为活动分区（表示该分区可启动)。“分区类型描述”是分区类型标志，也称为文件系统标志位，表示该分区的类型。“起始 CHS”和“结束 CHS”是比较“古老”的寻址方式，Windows 2000 之后的操作系统都会忽略这两个域。

MBR 分区规范没有提供分区表备份机制，一旦 0 号扇区分区表区域被破坏，分区表将很难恢复。

7.1.2 GPT 硬盘详解

为了弥补 MBR 硬盘的这些不足，英特尔公司在 20 世纪 90 年代末开发了一种全新的硬

盘格式，它最终成为 UEFI 的一部分，这便是 GUID Patition Table(GPT) 格式。表 7-1 列出了 MBR 分区和 GPT 分区的差异。

表 7-1　MBR 分区和 GPT 分区对比

要素 \ 分区类型	MBR 分区	GPT 分区
块地址位数	32 位	64 位
分区数量	4 个主分区	很多（现在的 UEFI 版本支持 128 个分区）
可扩展性	无	有版本号，可扩展
分区表安全性	无备份	有备份分区表
分区表完整性	无完整性检查	分区表有 CRC32 校验

总体来说，GPT 硬盘支持更大容量、更多分区及更好的安全性。下面来看一下 GPT 硬盘的结构，如图 7-3 所示。

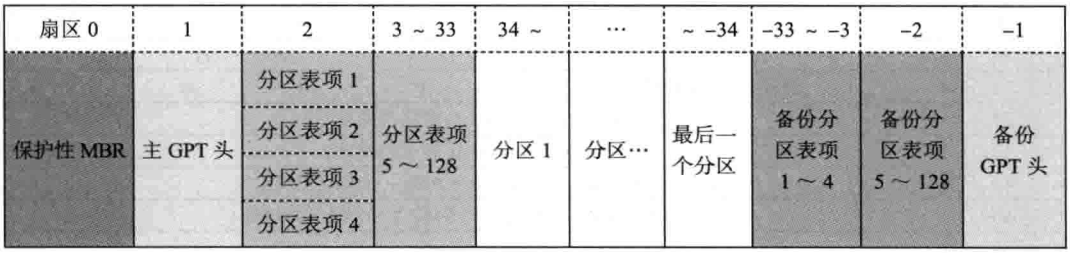


图 7-3　GPT 硬盘结构

GPT 硬盘仍然将硬盘划分为扇区。所有扇区统一编址，地址从 0 开始编号，扇区地址称为 LBA（Logic Block Address）。0 号扇区是保护性 MBR，用于兼容 MBR 硬盘；1 号扇区是 GPT 头，描述了 GPT 硬盘的结构；2 ~ 33 号分区，共 32 个扇区，是 GPT 硬盘的分区表。硬盘末尾的 33 个分区用于备份分区表和 GPT 头。下面分别讲述这几个分区。

1. 保护性 MBR

0 号扇区称为保护性 MBR，同 MBR 硬盘的 0 号扇区格式相同。其主要作用是兼容 MBR 硬盘时代的工具，防止这些工具因不识别 GPT 格式而破坏硬盘。虽然格式与 MBR 硬盘的 0 号扇区格式相同，但对各个域有不同的要求：启动代码域（0 ~ 439 字节）对 UEFI 系统无效，将被 UEFI 系统忽略；分区表项的第一个项必须包含整个硬盘，并且 BootIndicator 必须为 0，OSType 标志为 0xEE（此类型称为保护性 GPT）；其他分区表项必须为 0；标志位（510 ~ 511 字节）必须为 0xAA55；其他字节必须为 0。图 7-4 列出了保护性 MBR 的第一个分区表项。

偏移 \ 偏移	0	1	2	3	4	5	6	7
0	0	0x000200			0xEE	MIN（0xFFFFFF，尾扇区 CHS 地址）		
8	0x00000001				MIN（0xFFFFFFFF，尾扇区逻辑地址）			

图 7-4　保护性 MBR 分区表第一个表项

2. GPT 头

硬盘的 1 号扇区称为 GPT 头，存储了硬盘的结构信息，主要包括：GPT 头标志“EFIPART”、表头字节数、表头 CRC32 校验码、GPT 头和备份表头地址、可用于分区的扇区范围（First Usable LBA 和 Last Usable LBA）、硬盘 GUID、分区表地址和大小。图 7-5 所示为 GPT 头的结构。

偏移 \ 偏移	0	1	2	3	4	5	6	7
0	E	F	I	‘ ’	P	A	R	T
8	版本号				表头字节数			
16	表头 CRC32 校验码				0	0	0	0
24	MyLBA（存储 GPT 头的扇区的地址）							
32	AlternateLBA（备份表头的地址）							
40	FirstUsableLBA（可用于分区的第一个扇区的地址）							
48	LastUsableLBA（可用于分区的最后一个扇区的地址）							
56	硬盘 GUID							
64								
72	PartitionEntryLBA（分区表地址）							
80	NumberOfPartitionEntries（分区表项的数目）				SizeOfPartitionEntry（每个分区表项的字节数）			
88	PartitionEntryArrayCRC32				0	0	0	0
⋮	从 92 字节直到本扇区结束，所有字节全为 0							

图 7-5　GPT 头

硬盘最后一个扇区用于备份 GPT 头。当计算机系统读取硬盘 GPT 头时，会计算表头的 CRC32 校验码，如果计算出的校验码与硬盘中存储的校验码不一致，则系统会尝试从备份表头恢复 GPT 头。

代码清单 7-1 是 GPT 头的结构体。

代码清单 7-1　GPT 头结构体

```
typedef struct {
  EFI_TABLE_HEADER Header;          // 表头，含有 EFI_PTAB_HEADER_ID(“EFI PART”) 标志
  EFI_LBA MyLBA;                    // 存储该表的 LBA(Logic Block Address)
  EFI_LBA AlternateLBA;             // 备份分区表的地址
  EFI_LBA FirstUsableLBA;           // 可用于分区的第一个扇区的地址
```

```
  EFI_LBA LastUsableLBA;              // 可用于分区的最后一个扇区的地址
  EFI_GUID DiskGUID;                  // 硬盘的 GUID
  EFI_LBA PartitionEntryLBA;          // 分区表地址
  UINT32 NumberOfPartitionEntries;    // 分区表项的数目
  UINT32 SizeOfPartitionEntry;        // 每个分区表项的字节数，必须是 128 的 2^n 倍
  UINT32 PartitionEntryArrayCRC32;    // 分区表的 CRC32 码
} EFI_PARTITION_TABLE_HEADER;
```

3. GPT 分区表

GPT 表头的 PartitionEntryLBA 域指明了分区表的 LBA 地址，NumberOfPartitionEntries 域指明了分区的数目。从理论上讲，GPT 分区的数目可以有 264 个，但在实际中，2 ~ 33 号扇区用于存放分区表，每个分区表项占 256 字节，因而最多有 128 个分区。

分区表项包括分区类型、分区 GUID、分区占用的扇区范围、分区属性和分区名称。图 7-6 展示了一个分区表项的结构。

<table>
<tr><th>偏移
偏移</th><th>0</th><th>1</th><th>2</th><th>3</th><th>4</th><th>5</th><th>6</th><th>7</th><th>8</th><th>9</th><th>A</th><th>B</th><th>C</th><th>D</th><th>E</th><th>F</th></tr>
<tr><td>0x00</td><td colspan="16">PartitionTypeGUID（分区类型）</td></tr>
<tr><td>0x 10</td><td colspan="16">UniquePartitionGUID（分区标志）</td></tr>
<tr><td>0x 20</td><td colspan="8">StartingLBA（分区首扇区）</td><td colspan="8">EndingLBA（分区尾扇区）</td></tr>
<tr><td>0x 30</td><td colspan="8">Attributes</td><td colspan="8"></td></tr>
<tr><td>0x 40</td><td colspan="16" rowspan="4">PartitionName（以 NULL 结尾的字符串）</td></tr>
<tr><td>0x 50</td></tr>
<tr><td>0x 60</td></tr>
<tr><td>0x 70</td></tr>
</table>

图 7-6　GPT 分区表项结构

代码清单 7-2 列出了分区表项的结构体。

代码清单 7-2　GPT 分区表项结构体

```
typedef struct {
  EFI_GUID PartitionTypeGUID;         // 分区类型 GUID，0 表示此项为空
  EFI_GUID UniquePartitionGUID;       // 分区标志 GUID
  EFI_LBA StartingLBA;                // 分区的首扇区
  EFI_LBA EndingLBA;                  // 分区的尾扇区
  UINT64 Attributes;                  // 分区属性
  CHAR16 PartitionName[36];           // 分区名字符串，必须以 0x0000 结尾
} EFI_PARTITION_ENTRY;
```

分区表项每一个域的含义都显而易见，这里仅解释一下分区属性。表 7-2 列出了分区属性各个位的含义。

表 7-2 分区属性

位	名　称	属　性
0	必需分区	若此标志位为 1，表明该分区对系统至关重要，本分区被破坏或删除将导致系统不能正常工作
1	无 BlockIo 分区	若此标志位为 1，UEFI 不为该分区生成块设备
2	传统 BIOS 引导	仅供传统 BIOS 系统使用，UEFI 系统忽略该位
3~47	保留区 1	必须为 0
48~63	保留区 2	保留给 PartitionTypeGUID 的拥有者使用

硬盘倒数第 2 ~ 33 号扇区是备份分区表。分区表的校验码存放在 GPT 头的 PartitionEntryArrayCRC32 域。当系统分析硬盘的分区表时，会计算分区表的 CRC32 校验码，如果计算出的校验码与 PartitionEntryArrayCRC32 不同，则会尝试从倒数 2 ~ 33 扇区恢复分区表。分区表发生任何变化时都需要更新 PartitionEntryArrayCRC32 域并更新备份分区表，否则会造成不可预料的严重错误。

7.2 设备路径

系统中的每个设备都有一个唯一的路径。例如，每次进入 Shell 时，都会打印出系统中的硬盘设备及设备路径。图 7-7 是从 QEMU 进入 UEFI Shell 时的屏幕输出，这个虚拟机包含了一块空硬盘，PciRoot(0x0)/Pci(0x1，0x1)/Ata(0x0) 是这个硬盘设备的设备路径。

```
UEFI Interactive Shell v2.0
EDK II
UEFI v2.40 (EDK II, 0x00010000)
Mapping table
      BLK2: Alias(s):
          PciRoot(0x0)/Pci(0x1,0x1)/Ata(0x0)
```

图 7-7 硬盘设备路径示例

设备路径中的节点称为设备节点。设备路径由设备节点组成的列表构成，列表由“结束设备节点”结束。

设备路径：

设备节点	…	设备节点	结束设备节点

每个设备节点都以 EFI_DEVICE_PATH_PROTOCOL 开始，也就是说，EFI_DEVICE_PATH_PROTOCOL 是设备节点的第一个域。从面向对象的角度来说，EFI_DEVICE_PATH_PROTOCOL 是所有设备节点的基类。代码清单 7-3 是 EFI_DEVICE_PATH_PROTOCOL 的结构体，它包含了设备的类型次类型以及设备节点的长度。该结构体及其他设备路径节点结构体定义在 MdePkg/Include/Protocol/DevicePath.h 中。

代码清单 7-3　EFI_DEVICE_PATH_PROTOCOL 的结构体

```
typedef struct _EFI_DEVICE_PATH_PROTOCOL {
  UINT8 Type;              // 类型
  UINT8 SubType;           // 次类型
  UINT8 Length[2];         // 节点字节数
} EFI_DEVICE_PATH_PROTOCOL;
```

代码清单 7-4 是 PCI 总线设备的设备路径节点结构体，该节点是一个典型的设备路径节点，结构体中第一个域是 EFI_DEVICE_PATH_PROTOCOL，其他域（Function 和 Device）是 PCI 设备特有的属性。Header.length（即 PCI_DEVICE_PATH）的长度是 6。

代码清单 7-4　PCI 总线设备的设备路径节点结构体

```
typedef struct {
  EFI_DEVICE_PATH_PROTOCOL Header;
  UINT8 Function;          // PCI 功能号
  UINT8 Device;            // PCI 设备号
} PCI_DEVICE_PATH;
```

结束设备节点是一个特设的设备节点，它的类型为 0x7F，次类型为 0xFF 或 0x01，节点长度为 2 字节。表 7-3 列出了结束设备节点各域的值。

表 7-3　结束设备节点

字节偏移	域　名　称	值	值的意义
0	类型	0x7F	结束标志
1	次类型	0xFF/0x01	0xFF：整个设备路径结束 0x01：前一设备路径结束；新的设备路径开始
2	节点长度	4	结束节点没有数据，因而长度为 4 字节
3			

设备类型 Type 和次类型 SubType 决定了设备节点的类型。表 7-4 列出了 UEFI 支持的设备的类型和次类型。

表 7-4　UEFI 支持的设备的类型和次类型

设备路径类型	类型 Type 值	设备次类型
硬件设备路径	0x01	硬件设备，次类型包括如下几个： 1：PCI 设备；2：PCCARD；3：内存映射设备；4：vendor 自定义设备；5：控制器设备
ACPI 设备路径	0x02	ACPI 设备，次类型包括如下几个。 1：ACPI 设备；2：扩展 ACPI 设备；3：_ADR 设备
Messaging 设备路径	0x03	次类型包括如下几个： 1：ATAPI；2：SCSI；3：光纤；4：1394；5：USB；12：IPv4；13：IPv6；15：USB；18：SATA；20：802.1q

（续）

设备路径类型	类型 Type 值	设备次类型
介质设备路径	0x04	次类型包括如下几个。 1：硬盘；2：光驱；3：vendor 自定义设备；4：文件路径；5：介质 Protocol；6：固件文件路径；7：固件卷
BIOS 启动设备路径	0x05	用于启动不支持 UEFI 的操作系统
结束节点设备路径	0x7F	

总结一下，设备节点以 EFI_DEVICE_PATH_PROTOCOL 开始，它称为设备节点的头。节点头中声明了节点的类型和次类型。节点的长度定义在节点头的 Length 域中。表 7-5 显示了设备节点的结构。

表 7-5　设备节点结构

字节偏移	节点域名称	值	域的意义
0	类型	1、2、3、4、5、0x7F 中的一个	主设备号
1	次类型		次设备号
2	节点字节数	N	节点字节数
3			
4	本节点数据	—	节点内容
⋮			
n−1			

既然本章的主题是硬盘，那就不得不讲述一下硬盘设备（HARDDRIVE_DEVICE_PATH）节点（严格来说，其名称应为分区设备节点），它对应于一个硬盘分区。表 7-6 列出了硬盘设备节点的域及含义。

表 7-6　硬盘设备节点

域	偏移	域长度	域类型	域的含义
Type	0	1	UINT8	介质设备路径，值为 0x4
SubType	1	1	UINT8	硬盘，值为 1
Length	2	2	UINT16	长度，大小为 42 字节
PartitionNumber	4	4	UINT32	该分区在分区表中的位置，从 1 开始编号。 0 表示整个硬盘设备。 对 MBR 硬盘来说，有效值是 1、2、3、4 中的一个；对 GPT 硬盘来说，有效值的范围是 [1，分区个数]
PartitionStart	8	8	UINT64	该分区第一个扇区的 LBA 地址
PartitionSize	16	8	UINT64	该分区的扇区数
Signature	24	16	UINT8[16]	该分区的标识符。 如果 SignatureType 为 0，本域必须为 0；如果 SignatureType 为 1，本域前 4 字节有效，后面 12 字节为 0；如果 SignatureType 为 2，本域必须是一个有效的 GUID
MBRType	40	1	UINT8	1 表示 MBR 硬盘；2 表示 GPT 硬盘
SignatureType	41	1	UINT8	分区标识符的类型，0 表示无分区标识符；1 表示 MBR 分区（32 位）的标识符；2 表示标识符为 GUID（16 字节）

图 7-8 是一个 GPT 硬盘 FAT32 分区的设备路径示例。该示例中，FAT32 分区从扇区 0x22 开始，共 0xC8000 个扇区。该设备路径共有 4+1（结束设备节点）个设备节点。该设备路径对应的各个节点的设备号为 (2,1)/(1,1)/(3,1)/(4,1)/(0x7F,0xFF)。

```
Mapping table
      FS0: Alias(s):HD0a1::BLK3:
          PciRoot(0x0)/Pci(0x1,0x1)/Ata(0x0)/HD(1,GPT,2E53B480-1DD2-1000-8B0B-9F
001F000000,0x22,0xC8000)
```

图 7-8　FAT32 分区的设备路径

图 7-8 中 map 命令显示出的设备路径由可打印字符组成。这个设备路径字符串是如何得到的呢？ UEFI 提供了 EFI_DEVICE_PATH_TO_TEXT_PROTOCOL 用于将设备路径转换为字符串。代码清单 7-5 是 EFI_DEVICE_PATH_TO_TEXT_PROTOCOL 的结构体及其主要函数 ConvertDevicePathToText 的函数原型。

代码清单 7-5　EFI_DEVICE_PATH_TO_TEXT_PROTOCOL 结构体及成员函数原型

```
typedef  CHAR16*(EFIAPI *EFI_DEVICE_PATH_TO_TEXT_PATH)(
  IN CONST EFI_DEVICE_PATH_PROTOCOL *DevicePath,
  IN BOOLEAN DisplayOnly,
  IN BOOLEAN AllowShortcuts
  );
typedef  struct {
  EFI_DEVICE_PATH_TO_TEXT_NODE ConvertDeviceNodeToText; //将设备节点转换为字符串
  EFI_DEVICE_PATH_TO_TEXT_PATH ConvertDevicePathToText; //将设备路径转换为字符串
} EFI_DEVICE_PATH_TO_TEXT_PROTOCOL;
```

EFI_DEVICE_PATH_TO_TEXT_PROTOCOL 的成员函数 ConvertDevicePathToText 用于将设备路径 DevicePath 转换为字符串。若 DisplayOnly 为 TRUE，则生成的字符串为短格式，短格式不能用于被分析后生成设备路径；若 DisplayOnly 为 FALSE，则生成的字符串为长格式，长格式可以被分析后反向生成设备路径。调用者负责释放字符串占用的内存。示例 7-1 用于将设备路径 DiskDevicePath 转换为字符串并输出，字符串使用完毕后调用 gBS->FreePool 释放内存。

【示例 7-1】 打印硬盘的设备路径。

```
CHAR16* TextDevicePath = 0;
TextDevicePath = Device2TextProtocol->ConvertDevicePathToText(DiskDevicePath,
    TRUE, TRUE);
Print(L"%s\n", TextDevicePath);
//注意：使用完毕后调用者一定要释放此内存
if(TextDevicePath)gBS->FreePool(TextDevicePath);
```

为了方便开发者分析设备路径，EDK2 提供了 DevicePathLib（设备路径库）。IsDevicePathEnd (CONST VOID *Node) 用于判断设备节点 Node 是否为设备路径的设备结束节点。

NextDevicePathNode(CONST VOID *Node) 用于返回设备节点 Node 的下一个设备节点。示例 7-2 中的代码展示了如何遍历设备路径里的各个设备节点，并打印节点类型。

【示例 7-2】 遍历设备路径中的各个设备节点并打印节点类型。

```
EFI_STATUS PrintNode(EFI_DEVICE_PATH_PROTOCOL * Node)
{
  Print(L"(%d %d)/", Node->Type, Node->SubType);
  return 0;
}
EFI_DEVICE_PATH_PROTOCOL* WalkthroughDevicePath( EFI_DEVICE_PATH_PROTOCOL*
  DevPath,EFI_STATUS (*Callbk)(EFI_DEVICE_PATH_PROTOCOL*))
{
  EFI_DEVICE_PATH_PROTOCOL* pDevPath = DevPath;
  while(!IsDevicePathEnd (pDevPath)){
    Callbk(pDevPath);
    pDevPath = NextDevicePathNode (pDevPath);
  }
  return pDevPath;
}
```

现在已经学习了如何遍历设备路径的各个节点，以及如何将设备路径转换为字符串，下面我们将尝试找出系统中的硬盘设备，打印找到的硬盘设备路径，并分析硬盘设备路径的各个设备节点，相关代码如示例 7-3 所示。要完成这个任务，可以分如下几步来实现。

1）首先要用 gBS->LocateHandleBuffer 服务找出所有支持 DiskIo 的设备。

2）然后找到 DiskIo 设备的设备路径。

3）接着，按照示例 7-1 调用 ConvertDevicePathToText，得到设备路径字符串。

4）最后，按照示例 7-2 所示遍历设备路径的各个节点。

【示例 7-3】 分析硬盘设备路径。

```
EFI_STATUS
UefiMain(IN EFI_HANDLE ImageHandle, IN EFI_SYSTEM_TABLE  *SystemTable)
{
  EFI_STATUS Status;
  UINTN HandleIndex, NumHandles;
  EFI_HANDLE *ControllerHandle = NULL;
  EFI_DEVICE_PATH_TO_TEXT_PROTOCOL* Device2TextProtocol = 0;
  //找出 DevicePathToTextProtocol
  Status = gBS->LocateProtocol(&gEfiDevicePathToTextProtocolGuid,NULL,
    (VOID**)&Device2TextProtocol);
  //1: 找出所有支持 DiskIo 的设备
  Status = gBS->LocateHandleBuffer(ByProtocol,&gEfiDiskIoProtocolGuid,
    NULL, &NumHandles, &ControllerHandle);
  if (EFI_ERROR(Status)) {
    return Status;
  }
```

```
    // 2: 对于每个 DiskIo 设备，打开该设备上的 DevicePathProtocol
    for (HandleIndex = 0; HandleIndex < NumHandles; HandleIndex++) {
      EFI_DEVICE_PATH_PROTOCOL* DiskDevicePath;
      Status = gBS->OpenProtocol(ControllerHandle[HandleIndex],
        &gEfiDevicePathProtocolGuid, (VOID**)&DiskDevicePath,
        gImageHandle, NULL, EFI_OPEN_PROTOCOL_GET_PROTOCOL);
      if (EFI_ERROR(Status)) {
        continue;
      }
      // 3: 将设备路径 DiskDevicePath 转换为字符串
      {
        CHAR16* TextDevicePath = 0;
        TextDevicePath = Device2TextProtocol ->
          ConvertDevicePathToText(DiskDevicePath, TRUE, TRUE);
        Print(L"%s\n", TextDevicePath);
        if ( TextDevicePath ) gBS->FreePool(TextDevicePath);
      }
      // 4: 遍历设备路径 DiskDevicePath 里的各个设备节点
      WalkthroughDevicePath(DiskDevicePath,PrintNode);
      Print(L"\n\n");
    }
    return 0;
  }
```

图 7-9 是示例 7-3 的输出。系统有一块 GPT 硬盘，扇区 0x22 ~ 0xC8021 是 FAT32 分区，共 0xC8000 个扇区。扇区 0xC8022 ~ 0x1387FDE 是一个未格式化的分区，共 0x 12BFFBD 个扇区。输出分为 4 块，第 1 块表示主 Ata 设备（即主硬盘设备）；第 2 块是 FAT32 分区设备；第 3 块是未格式化分区设备；第 4 块是次 Ata 设备（即光驱设备）。每块 3 行，表示一个 DiskIo 设备，其中，第一行是设备路径字符串，第二行是各个设备节点的设备号。

```
FS0:\> TestDevicePath.efi
PciRoot(0x0)/Pci(0x1,0x1)/Ata(Primary,Master,0x0)
 (2 1)/(1 1)/(3 1)/

PciRoot(0x0)/Pci(0x1,0x1)/Ata(Primary,Master,0x0)/HD(1,GPT,2E53B480-1DD2-1000-BB
0B-9F001F000000,0x22,0xC8000)
 (2 1)/(1 1)/(3 1)/(4 1)/

PciRoot(0x0)/Pci(0x1,0x1)/Ata(Primary,Master,0x0)/HD(2,GPT,2431B600-1DD2-1000-BA
CF-9F001F000000,0xC8022,0x12BFFBD)
 (2 1)/(1 1)/(3 1)/(4 1)/

PciRoot(0x0)/Pci(0x1,0x1)/Ata(Secondary,Master,0x0)
 (2 1)/(1 1)/(3 1)/
```

图 7-9　硬盘设备路径及设备节点的设备号

7.3　硬盘相关的 Protocol

因为硬盘是一种块设备，所以每个硬盘设备（硬盘设备包括分区设备）控制器都安装有一个 BlockIo 实例、一个 BlockIo2 实例。BlockIo 提供了访问设备的阻塞函数，BlockIo2 提

供了访问设备的异步函数。

每个硬盘设备控制器还安装一个 DiskIo 实例、一个 DiskIo2 实例。DiskIo 提供了访问硬盘的阻塞函数，DiskIo2 提供了访问硬盘的异步函数。

BlockIo 与 DiskIo 的区别是，BlockIo 只能按块（扇区）读写设备，而 DiskIo 可以从任意偏移处（以字节为单位）读写磁盘，并且可以读取任意字节数。

UEFI 也会为每个分区生成一个控制器。如果这个分区的属性 Bit 1 值为 0，则为这个分区控制器安装 BlockIo、BlockIo2、DiskIo、DiskIo2。

7.3.1 BlockIo 解析

BlockIo 是 EFI_BLOCK_IO_PROTOCOL 的缩写。代码清单 7-6 是 BlockIo 的结构体，它包含了两个属性：Revision 和 Media，以及 4 个成员函数：Reset、ReadBlocks、WriteBlocks 和 FlushBlocks。

代码清单 7-6　BlockIo 结构体

```
typedef struct _EFI_BLOCK_IO_PROTOCOL {
  UINT64 Revision;                  // Protocol 版本号
  EFI_BLOCK_IO_MEDIA *Media;        // 设备信息
  EFI_BLOCK_RESET Reset;            // 重置设备
  EFI_BLOCK_READ ReadBlocks;        // 读扇区
  EFI_BLOCK_WRITE WriteBlocks;      // 写扇区
  EFI_BLOCK_FLUSH FlushBlocks;      // 将缓冲中的数据写入扇区
} EFI_BLOCK_IO_PROTOCOL;
```

1. BlockIo 中的设备信息 Media

Media 指向该设备的 EFI_BLOCK_IO_MEDIA 结构体（见代码清单 7-7），它存储了设备的属性信息。这个结构体具有只读属性，内容只能由 EFI_BLOCK_IO_PROTOCOL 的生产者更改。

代码清单 7-7　EFI_BLOCK_IO_MEDIA 结构体

```
typedef struct {
  UINT32  MediaId;
  BOOLEAN RemovableMedia;
  BOOLEAN MediaPresent;             // 设备中是否有介质
  BOOLEAN LogicalPartition;         // 该介质是分区 (TRUE)，或者整个设备 (FALSE)
  BOOLEAN ReadOnly;
  BOOLEAN WriteCaching;             // 是否用 cache 方式写硬盘
  UINT32  BlockSize;
  UINT32  IoAlign;          // 读写硬盘时缓冲区地址对齐字节数，0 或 2^n。0 和 1 表示可以是任意地址
  EFI_LBA LastBlock;        // 设备最后一个块的 LBA 地址
  // 以下三项只出现在 EFI_BLOCK_IO_PROTOCOL_REVISION2 及以上版本
```

```
  // 当该 Protocol 安装在分区设备上时，数值全为 0
  EFI_LBA LowestAlignedLba;
  UINT32 LogicalBlocksPerPhysicalBlock;
  UINT32 OptimalTransferLengthGranularity;
} EFI_BLOCK_IO_MEDIA;
```

EFI_BLOCK_IO_MEDIA 各个成员意义都比较直观。需要特殊说明的是，成员 LogicalPartition，当它为 TRUE 时，表示该介质是分区设备；当它为 FALSE 时，表示该介质是整块硬盘设备。

2. ReadBlocks 函数

ReadBlocks 用于读取块设备，它只能按块读取设备。代码清单 7-8 列出了 ReadBlocks 的函数原型。

代码清单 7-8　ReadBlocks 函数原型

```
/**ReadBlocks 函数
从块 LBA 开始读取 BufferSize（块大小的整数倍）字节。阻塞操作，读取成功或失败后返回
*/
typedef  EFI_STATUS (EFIAPI *EFI_BLOCK_READ) (
  IN EFI_BLOCK_IO_PROTOCOL *This,
  IN UINT32 MediaId,                  // 设备中的介质号
  IN EFI_LBA LBA,                     // 读取设备（或分区）起始块的 LBA 地址
  IN UINTN BufferSize,                // 读取的字节数，必须是块大小的整数倍
  OUT VOID *Buffer                    // 读取的数据存到此缓冲区，调用者负责管理此内存
);
```

示例 7-4 展示了如何用 ReadBlocks 读取硬盘或分区的扇区。

【示例 7-4】 用 ReadBlocks 读扇区。

```
EFI_STATUS TestReadBlocks( EFI_BLOCK_IO_PROTOCOL* BlockIo)
{
  EFI_STATUS Status = 0;
  UINT8* Buf;
  UINT32 BlockSize = BlockIo->Media->BlockSize;
  SAFECALL(Status = gBS->AllocatePool(EfiBootServicesCode, BlockSize,
 (VOID**)&Buf));
  SAFECALL(Status = BlockIo->ReadBlocks(BlockIo, BlockIo->Media->MediaId,
      0, BlockSize, (VOID**)Buf));
  SAFECALL(Status = gBS->FreePool(Buf));
  return Status;
}
```

SAFECALL 是宏，用于检查 Status 是否为错误值，定义如下：

```
#define SAFECALL(x) if(EFI_SUCCESS != (x)){return Status;}
```

3. WriteBlocks 函数

WriteBlocks 用于按块写设备，其接口与 ReadBlocks 接口相同。代码清单 7-9 是 WriteBlocks 函数原型。

代码清单 7-9　WriteBlocks 函数原型

```
/** WriteBlocks 函数
从块 LBA 开始写 BufferSize(块大小的整数倍)字节。阻塞操作，写成功或失败后返回
*/
typedef  EFI_STATUS (EFIAPI *EFI_BLOCK_WRITE) (
  IN EFI_BLOCK_IO_PROTOCOL *This,
  IN UINT32 MediaId,                    //设备中的介质号
  IN EFI_LBA LBA,                       //写设备(或分区)起始块的 LBA 地址
  IN UINTN BufferSize,                  //写字节数，必须是块大小的整数倍
  OUT VOID *Buffer                      //从此缓冲区写数据到设备，调用者负责管理此内存
);
```

【示例 7-5】 WriteBlocks 示例。

```
EFI_STATUS TestWriteBlocks( EFI_BLOCK_IO_PROTOCOL* BlockIo)
{
  EFI_STATUS Status = 0;
  UINT8* Buf;
  UINT32 BlockSize = BlockIo->Media->BlockSize;
  SAFECALL(Status = gBS->AllocatePool(EfiBootServicesCode, BlockSize,
      (VOID**)&Buf));
  gBS->SetMem(Buf, BlockSize, 0);
  SAFECALL(Status = BlockIo->WriteBlocks(BlockIo, BlockIo->Media->MediaId, 0,
      BlockSize, (VOID**)Buf));
  SAFECALL(Status = gBS->FreePool(Buf));
  return Status;
}
```

4. FlushBlocks 函数

FlushBlocks 用于将设备缓存中修改过的数据全部更新到介质中。代码清单 7-10 展示了 FlushBlocks 的函数原型。

代码清单 7-10　FlushBlocks 函数原型

```
//将设备缓存中修改过的数据全部更新到介质中
typedef  EFI_STATUS (EFIAPI *EFI_BLOCK_FLUSH)(IN EFI_BLOCK_IO_PROTOCOL *This);
```

7.3.2　BlockIo2 解析

IO 操作通常非常消耗时间。ReadBlocks、WriteBlocks 及 FlushBlocks 这种阻塞操作通常

会浪费大量时间在等待上。为了提高性能，UEFI 提供了异步读写块设备的 Protocol，这便是 BlockIo2。BlockIo2 是 EFI_BLOCK_IO2_PROTOCOL 的简称。BlockIo2 是 UEFI Spec 2.4 中新增的 Protocol。代码清单 7-11 是 BlockIo2 的结构体。

代码清单 7-11　BlockIo2 结构体

```
typedef struct _EFI_BLOCK_IO2_PROTOCOL {
  EFI_BLOCK_IO_MEDIA *Media;          // 设备信息
  EFI_BLOCK_RESET_EX Reset;           // 重置设备（阻塞函数）
  EFI_BLOCK_READ_EX ReadBlocksEx;     // 异步读设备
  EFI_BLOCK_WRITE_EX WriteBlocksEx;   // 异步写设备
  EFI_BLOCK_FLUSH_EX FlushBlocksEx;   // 异步 Flush 设备
} EFI_BLOCK_IO2_PROTOCOL;
```

BlockIo2 的属性 Media 及函数 Reset 与 BlockIo 的完全相同，但读、写及 Flush 这三个函数有所不同。

1. ReadBlocksEx 函数

ReadBlocksEx 采用异步方式读硬盘或分区的扇区。

通常异步操作需要事件的支持，还需要上下文用于传递数据。在 UEFI 规范中，异步操作的事件和上下文封装起来称为令牌（Token），不同的异步操作有不同类型的令牌。一个拥有令牌的异步操作也称为一个事务。ReadBlocksEx 的令牌为 EFI_BLOCK_IO2_TOKEN。代码清单 7-12 是令牌 EFI_BLOCK_IO2_TOKEN 的定义。

代码清单 7-12　EFI_BLOCK_IO2_TOKEN 定义

```
typedef struct {
  EFI_EVENT Event;                    // 事务对应的事件
  EFI_STATUS TransactionStatus;       // 事务完成（成功或失败）后的状态
} EFI_BLOCK_IO2_TOKEN;
```

ReadBlockEx 函数向设备发出读指令后立刻返回。设备完成操作后会触发 Token 中的事件。如果函数返回错误，则表示指令没有发送到设备，事件将不会触发。如果 Token 为 NULL 或 Token->Event 为 NULL，则该函数将退化为阻塞函数，功能与 BlockIo 的 ReadBlocks 完全相同。Token 由调用者负责创建和删除。代码清单 7-13 展示了 ReadBlocksEx 的函数原型。

代码清单 7-13　ReadBlocksEx 函数原型

```
// 异步读取块设备中的块
typedef  EFI_STATUS(EFIAPI *EFI_BLOCK_READ_EX) (
  IN EFI_BLOCK_IO2_PROTOCOL *This,
  IN UINT32 MediaId,                  // 设备中的介质号
```

```
  IN EFI_LBA LBA,                          //读取设备（或分区）起始块的 LBA 地址
  IN OUT EFI_BLOCK_IO2_TOKEN *Token,       //此事务对应的 Token，调用者负责生成、关闭它
  IN UINTN BufferSize,                     //读取的字节数，必须是块大小的整数倍
  OUT VOID *Buffer                         //读取的数据存到此缓冲区，调用者负责管理此内存
 );
```

配合 UEFI 中的事件，有两种方式等待异步读硬盘完成：一种是使用 WaitForEvent 等待异步读结束后再执行后续任务；另一种是在事件的 Notification 函数中完成后续任务。

前一种方式异步读取硬盘扇区的流程是：

1）生成 EFI_BLOCK_IO2_TOKEN 对象，主要是生成 Token 中的事件对象。

2）通过 ReadBlocksEx 函数向设备发出读指令。

3）执行其他任务。

4）通过 WaitForEvent 服务等待 Token 中的事件，然后处理读回的数据。

示例 7-6 展示了使用前一种方式读取硬盘扇区的方法。

【示例 7-6】 WaitForEvent 方式的异步读。

```
//在 Token 中使用普通事件，通过 WaitForEvent 等待事务结束，然后执行后续任务
EFI_STATUS TestReadBlocks2( EFI_BLOCK_IO2_PROTOCOL* BlockIo2)
{
  EFI_STATUS Status = 0;
  UINT8* Buf;
  EFI_BLOCK_IO2_TOKEN b2Token;
  UINTN Index;
  UINT32 BlockSize = BlockIo2->Media->BlockSize;
  //首先为 b2ToKen 生成一个普通事件，这个事件将在硬盘执行完读命令后触发
  SAFECALL(Status = gBS->CreateEvent(0, TPL_NOTIFY, NULL, NULL, &b2Token.Event));
  //分配内存，用于接收从硬盘读回的数据
  SAFECALL(Status=gBS->AllocatePool(EfiBootServicesCode, BlockSize, (VOID**)&Buf));
  //向硬盘发送异步读命令
  Status = BlockIo2->ReadBlocksEx(BlockIo2, BlockIo2->Media->MediaId, 0,
      &b2Token, BlockSize, (VOID**)Buf);
  if(EFI_ERROR(Status)){
    //ReadBlocksEx 失败，处理错误
  }else{
    //这时硬盘正在执行上面发出的读操作，我们可以在 b2Token.Event 触发前做一些其他工作
    DoWhateverelse();
    //然后等待 b2Token.Event 触发
    gBS->WaitForEvent(1, &b2Token.Event, &Index);
    //处理读回的数据
 ProcessData(Buf);
  }
  //释放内存
  SAFECALL(Status = gBS->FreePool(Buf));
  return Status;
}
```

后一种方式异步读取硬盘扇区的流程是：

1）生成 EFI_BLOCK_IO2_TOKEN 对象，主要是生成 Token 中的事件对象，定义事件的 Notification 函数，该函数的主要作用是处理读到的数据。

2）通过 ReadBlocksEx 函数向设备发出读指令。

3）执行其他任务。

4）通过 WaitForEvent 服务等待 Token 中的事件。WaitForEvent 返回时，Notification 函数已经由系统执行完毕。

示例 7-7 展示了后一种异步读硬盘扇区的方式。

【示例 7-7】 配合 Notification 函数的异步读。

```
// Token 中事件类型为 EVT_NOTIFY_SIGNAL，读成功的后续任务将在该 Event 的 Notification 函数
// 中执行
VOID EFIAPI BlockIo2OnReadWriteComplete(IN EFI_EVENT Event, IN VOID *Context)
{
  // 获得接收缓冲区地址
  UINT8* Buf = (UINT8*)Context;
  ProcessData(Buf);
  // 释放接收缓冲区内存
  gBS->FreePool(Buf);
}
EFI_STATUS TestReadBlocks2Signal( EFI_BLOCK_IO2_PROTOCOL* BlockIo2)
{
  EFI_STATUS Status = 0;
  UINT8* Buf = NULL;
  EFI_BLOCK_IO2_TOKEN b2Token;
  UINT32 BlockSize = BlockIo2->Media->BlockSize;
  SAFECALL(Status = gBS->AllocatePool(EfiBootServicesCode, BlockSize, (VOID**)&Buf));
  // 生成 EVT_NOTIFY_SIGNAL 类型的事件。事件触发后，BlockIo2OnReadWriteComplete 会自动
  // 被内核调用。接收缓冲区 Buf 作为参数传递给 BlockIo2OnReadWriteComplete 函数
  SAFECALL(Status = gBS->CreateEvent(EVT_NOTIFY_SIGNAL, TPL_NOTIFY,
      BlockIo2OnReadWriteComplete, (VOID*)Buf, &b2Token.Event));
  Status = BlockIo2->ReadBlocksEx(BlockIo2, BlockIo2->Media->MediaId, 0,
      &b2Token, BlockSize, (VOID**)Buf);
  if(EFI_ERROR(Status)){
    SAFECALL(Status = gBS->FreePool(Buf));
    // 处理错误
  }else{
    UINTN Index = 0;
    DoWhateverelse();
    // 退出前确保事务已经执行完毕
    gBS->WaitForEvent(1, &b2Token.Event, &Index);
  }
  return Status;
}
```

2. WriteBlocksEx 函数

WriteBlocksEx 用于异步写数据到硬盘扇区。代码清单 7-14 是 WriteBlocksEx 的函数原型，该函数执行后立刻返回，设备完成操作后会触发 Token 中的 Event，这个操作也称为一个事务，每个事务都应该有一个 Token。如果函数返回错误，则表示指令没有发送到设备，Event 将不会触发。如果 Token 为 NULL 或 Token->Event 为 NULL，则该函数将退化为阻塞函数，功能与 BlockIo 的 WriteBlocks 完全相同。

代码清单 7-14　WriteBlocksEx 函数原型

```
//异步写数据到块设备的块中
typedef  EFI_STATUS(EFIAPI *EFI_BLOCK_WRITE_EX) (
  IN EFI_BLOCK_IO2_PROTOCOL *This,
  IN UINT32 MediaId,                    //设备中的介质号
  IN EFI_LBA LBA,                       //写设备（或分区）起始块的 LBA 地址
  IN OUT EFI_BLOCK_IO2_TOKEN *Token,    //此事务对应的 Token，调用者负责生成、关闭 Token
  IN UINTN BufferSize,                  //写字节数，必须是块大小的整数倍
  OUT VOID *Buffer                      //从此缓冲区写数据到设备，调用者负责管理此内存
  );
```

WriteBlocksEx 的调用方式与 ReadBlocksEx 相同，作为练习，读者可以试着自己写硬盘的一个扇区（当然最好是在 QEMU 虚拟机上练习）。

3. FlushBlocksEx 函数

FlushBlocksEx 是 FlushBlock 的异步版，用于将设备缓存中修改过的数据全部更新到介质中。代码清单 7-15 展示了 FlushBlocksEx 函数的原型，该函数执行后立刻返回，设备完成操作后会触发 Token 中的 Event，这个操作也是一种事务，每个事务都应该有一个 Token。如果函数返回错误，则表示指令没有发送到设备，Event 将不会触发。如果 Token 为 NULL 或 Token->Event 为 NULL，则该函数将退化为阻塞函数，功能与 BlockIo 的 FlushBlocks 完全相同。

代码清单 7-15　FlushBlocksEx 函数原型

```
//异步操作将设备缓存中修改过的数据全部更新到介质中
typedef  EFI_STATUS(EFIAPI *EFI_BLOCK_FLUSH_EX) (
  IN EFI_BLOCK_IO2_PROTOCOL *This,
  IN OUT EFI_BLOCK_IO2_TOKEN *Token      //事务所拥有的 Token
);
```

7.3.3　DiskIo 解析

BlockIo 提供了按块访问设备的功能。它虽然有很好的性能，但是也让我们操作磁盘变得烦琐。为了方便对磁盘的操作，UEFI 提供了 DiskIo（全称 EFI_DISK_IO_PROTOCOL），

利用 DiskIo 我们可以从磁盘任意地址读写任意长度的数据。DiskIo 建立在 BlockIo 基础之上。硬盘设备和分区设备都会安装一个 DiskIo 实例。代码清单 7-16 是 DiskIo 结构体，它包含了一个属性 Revision，以及两个成员函数：ReadDisk 和 WriteDisk。

代码清单 7-16　DiskIo 结构体

```
typedef struct _EFI_DISK_IO_PROTOCOL {
  UINT64 Revision;              //版本号
  EFI_DISK_READ ReadDisk;       //读磁盘
  EFI_DISK_WRITE WriteDisk;     //写磁盘
} EFI_DISK_IO_PROTOCOL;
```

1. ReadDisk 函数

ReadDisk 用于读取硬盘，其函数原型定义在代码清单 7-17 中。

代码清单 7-17　ReadDisk 函数原型

```
//从磁盘偏移 Offset 处读 BufferSize 字节数据
typedef  EFI_STATUS(EFIAPI *EFI_DISK_READ)(
  IN EFI_DISK_IO_PROTOCOL *This,
  IN UINT32 MediaId,       //介质号
  IN UINT64 Offset,        //从偏移 Offset(以字节为单位)处开始读数据
  IN UINTN BufferSize,     //读取数据的长度
  OUT VOID *Buffer         //数据读取到此缓冲区，调用者负责管理此内存
);
```

2. WriteDisk 函数

WriteDisk 用于写硬盘，其函数原型定义在代码清单 7-18 中。WriteDisk 函数中各个参数的意义可参考 ReadDisk。

代码清单 7-18　WriteDisk 函数原型

```
//向磁盘地址 Offset 处写 BufferSize 字节数据
typedef  EFI_STATUS(EFIAPI *EFI_DISK_WRITE)(
  IN EFI_DISK_IO_PROTOCOL *This,
  IN UINT32 MediaId,
  IN UINT64 Offset,
  IN UINTN BufferSize,
  IN VOID *Buffer
);
```

7.3.4　DiskIo2 解析

上文提到过，DiskIo 建立在 BlockIo 基础之上，提供了访问磁盘设备的阻塞操作。UEFI

同样定义了异步的磁盘操作协议，那就是 DiskIo2，它建立在 BlockIo2 基础之上。DiskIo2 是 EFI_DISK_IO2_PROTOCOL 的简称。代码清单 7-19 展示了 DiskIo2 的结构体，它包含了一个 Revision 属性，以及 4 个成员函数：ReadDiskEx、WriteDiskEx、FlushDiskEx 和 Cancel。

代码清单 7-19　DiskIo2 结构体

```
typedef struct _EFI_DISK_IO2_PROTOCOL {
  UINT64 Revision;
  EFI_DISK_CANCEL_EX Cancel;              // 取消磁盘设备上处于等待状态的事务
  EFI_DISK_READ_EX ReadDiskEx;            // 异步读磁盘
  EFI_DISK_WRITE_EX WriteDiskEx;          // 异步写磁盘
  EFI_DISK_FLUSH_EX FlushDiskEx;          // 异步 Flush 磁盘
} EFI_DISK_IO2_PROTOCOL;
```

异步操作需要令牌，DiskIo2 中的异步操作也不例外，代码清单 7-20 列出了它用到的令牌 EFI_DISK_IO2_TOKEN。事务结束后，无论事务是否成功，都会触发令牌中的事件。如果事务成功，则令牌中的事务状态设为 0；如果事务失败，则将令牌中的事务状态设为错误代码。

代码清单 7-20　EFI_DISK_IO2_TOKEN

```
typedef struct {
  EFI_EVENT Event;                        // 无论事务成功或失败，事务结束后，Event 会触发
  EFI_STATUS TransactionStatus;           // 表示 Event 触发后事务的状态
} EFI_DISK_IO2_TOKEN;
```

这几个异步读写磁盘的函数，其调用方法与 7.3.2 节中的 BlockIo2 的异步读写磁盘扇区函数的调用方法大同小异，下面仅仅介绍一下这几个函数的函数原型。

1. ReadDiskEx 函数

ReadDiskEx 是 ReadDisk 的异步版，用于异步读硬盘，代码清单 7-21 列出了 ReadDiskEx 的函数原型。如果 Token 为 NULL 或 Token 中的事件无效，则该函数退化为阻塞操作，即相当于 DiskIo 的 ReadDisk 函数。如果 Token 中的事件有效，则该函数向设备发出读指令后立即返回，系统在该事务完成后自动触发 Token 中的事件。

代码清单 7-21　ReadDiskEx 函数原型

```
// 异步操作，从磁盘偏移 Offset 处读 BufferSize 字节数据
typedef EFI_STATUS(EFIAPI *EFI_DISK_READ_EX)(
  IN EFI_DISK_IO_PROTOCOL *This,
  IN UINT32 MediaId,                      // 介质号
  IN UINT64 Offset,                       // 从 Offset（以字节为单位）处开始读数据
  IN OUT EFI_DISK_IO2_TOKEN *Token,
  IN UINTN  BufferSize,                   // 读取数据的长度
```

```
  OUT VOID  *Buffer                          // 数据读取到此缓冲区，调用者负责管理此内存
  );
```

2. WriteDiskEx 函数

WriteDiskEx 用于异步写硬盘，其函数原型定义在代码清单 7-22 中。WriteDiskEx 函数各个参数的意义可参考 ReadDiskEx。同其他异步操作一样，如果 Token 为 NULL 或 Token 中的事件无效，则该函数退化为阻塞操作，即相当于 DiskIo 的 WriteDisk 函数。如果 Token 中的事件有效，则该函数向设备发出读指令后立即返回，系统在该事务完成后自动触发 Token 中的事件。

代码清单 7-22　WriteDiskEx 函数原型

```
// 异步操作：向磁盘地址 Offset 处写 BufferSize 字节数据
typedef EFI_STATUS(EFIAPI *EFI_DISK_WRITE_EX) (
  IN EFI_DISK_IO2_PROTOCOL  *This,
  IN UINT32  MediaId,
  IN UINT64  Offset,
  IN OUT EFI_DISK_IO2_TOKEN  *Token,
  IN UINTN BufferSize,
  IN VOID *Buffer
  );
```

3. FlushDiskEx 函数

FlushDiskEx 用于将设备上所有修改过的数据更新到设备。代码清单 7-23 是 FlushDiskEx 函数原型。

代码清单 7-23　FlushDiskEx 函数原型

```
// 异步操作：Flush 所有修改过的数据到设备上
typedef EFI_STATUS(EFIAPI *EFI_DISK_FLUSH_EX) (
  IN EFI_DISK_IO2_PROTOCOL *This,
  IN OUT EFI_DISK_IO2_TOKEN *Token
);
```

4. Cancel 函数

Cancel 用于终止所有已经向设备发出但未完成的异步请求。代码清单 7-24 是 Cancel 函数原型。

代码清单 7-24　Cancel 函数原型

```
/**  终止所有向设备发出的但未完成的异步请求
  @retval EFI_SUCCESS              // 所有未完成的异步请求被成功终止
```

```
  @retval EFI_DEVICE_ERROR        //设备报告错误
**/
typedef EFI_STATUS(EFIAPI *EFI_DISK_CANCEL_EX)(IN EFI_DISK_IO2_PROTOCOL *This);
```

7.3.5 PassThrough 解析

虽然 BlockIo 和 DiskIo 极大地方便了我们操作磁盘，但是其功能十分有限。通过 BlockIo 和 DiskIo，我们只能对硬盘设备进行读、写、Flush 操作，如果想对硬盘进行更多的操作，则需要通过 PassThrough 向硬盘发送命令。

UEFI 标准中定义了两种 PassThrough：一种用于 ATA(Advanced Technology Attachment) 硬盘，另一种用于 SCSI 硬盘和 ATAPI(AT Attachment Program Interface) 硬盘。代码清单 7-25 列出了 ATA 设备的 PassThrough Protocol 数据结构。

代码清单 7-25　ATA 设备的 PassThrough Protocol 数据结构

```
struct _EFI_ATA_PASS_THRU_PROTOCOL {
  EFI_ATA_PASS_THRU_MODE *Mode;                          //设备模式
  EFI_ATA_PASS_THRU_PASSTHRU PassThru;                   //向 ATA 设备发送 ATA 命令
  EFI_ATA_PASS_THRU_GET_NEXT_PORT GetNextPort;           //列出 ATA 设备的端口号
  //列出指定端口上的 Port Multiplier
  EFI_ATA_PASS_THRU_GET_NEXT_DEVICE GetNextDevice;
  //为设备生成 DevicePath 节点
  EFI_ATA_PASS_THRU_BUILD_DEVICE_PATH BuildDevicePath;
  //获得 DevicePath 对应的端口和 Port Multiplier
  EFI_ATA_PASS_THRU_GET_DEVICE GetDevice;
  EFI_ATA_PASS_THRU_RESET_PORT ResetPort;                //重置指定端口及其端口上的设备
  EFI_ATA_PASS_THRU_RESET_DEVICE ResetDevice;            //重置指定的设备
};
```

EFI ATA PASS THRU PROTOCOL 中比较重要的成员函数是 PassThru，其函数原型定义在代码清单 7-26 中，其作用是向设备发送命令包，设备由 Port 和 PortMultiplierPort 决定。若指定参数 Event，则 PassThru 采用异步方式；若参数 Event 设为 NULL，则 PassThru 采用阻塞模式。PassThru 命令格式复杂，感兴趣的读者可以参阅 ATA Spec[⊖]。示例 7-8 展示了利用 PassThru 读硬盘扇区的方法，其作用与 BlockIo 中的 ReadBlocks 相同。

代码清单 7-26　PassThru 函数原型

```
typedef EFI_STATUS(EFIAPI *EFI_ATA_PASS_THRU_PASSTHRU)(
  IN EFI_ATA_PASS_THRU_PROTOCOL *This,
  IN UINT16 Port,
  IN UINT16 PortMultiplierPort,
```

⊖ www.t13.org。

```
  IN OUT EFI_ATA_PASS_THRU_COMMAND_PACKET *Packet,
  IN EFI_EVENT Event OPTIONAL
  );
```

【示例 7-8】 用 PassThru 读硬盘扇区。

```
EFI_STATUS ReadSectors(UINT16 Port, UINT16 PortMultiplierPort, UINT64 StartLba,
    UINT32 SectorCount, CHAR8* Buffer, UINT32 DeviceId)
{
  EFI_STATUS Status;
  UINT32 IsExt = (StartLba > 0xFFFFFFFFFFFFFFFF);
  EFI_ATA_PASS_THRU_COMMAND_PACKET Packet;
  //第一步：构造命令包
  Packet.Protocol = EFI_ATA_PASS_THRU_PROTOCOL_UDMA_DATA_IN;
  Packet.Acb->AtaCommand   = IsExt?  ATA_CMD_READ_DMA_EXT : ATA_CMD_READ_DMA ;
  Packet.Timeout = 0;
  Packet.Length = EFI_ATA_PASS_THRU_LENGTH_SECTOR_COUNT;
  Packet.OutTransferLength = 0;
  Packet.OutDataBuffer = NULL;
  Packet.InTransferLength = SectorCount;
  Packet.InDataBuffer = Buffer;
  Packet.Acb->AtaFeatures = 0;
  Packet.Acb->AtaFeaturesExp = 0;
  Packet.Acb->AtaSectorNumber = (UINT8) StartLba;
  Packet.Acb->AtaCylinderLow = (UINT8) RShiftU64 (StartLba, 8);
  Packet.Acb->AtaCylinderHigh = (UINT8) RShiftU64 (StartLba, 16);
  Packet.Acb->AtaSectorCount = (UINT8) SectorCount;
  Packet.Acb->AtaDeviceHead = (UINT8) (0xE0 | (PortMultiplierPort<< 4));
  if(IsExt != 0){
    Packet.Acb->AtaSectorNumberExp = (UINT8) RShiftU64 (StartLba, 24);
    Packet.Acb->AtaCylinderLowExp = (UINT8) RShiftU64 (StartLba, 32);
    Packet.Acb->AtaCylinderHighExp = (UINT8) RShiftU64 (StartLba, 40);
    Packet.Acb->AtaSectorCountExp = (UINT8) (SectorCount >> 8);
  }else{
    Packet.Acb->AtaDeviceHead = (UINT8) (Packet.Acb->AtaDeviceHead | RShiftU64
        (StartLba, 24));
  }
  //第二步：通过 PassThru 向设备发送命令包
  Status = AtaPassThroughProtocol->PassThru(AtaPassThroughProtocol,
        Port, PortMultiplierPort, &Packet, NULL);
  //第三步：获得命令结果。参数 Event 为 NULL，当 PassThru 返回时，命令已经执行完毕
  return Status;
}
```

这里总结一下前面讲述的硬盘及读写硬盘的方法。PassThrough 通过向硬盘设备发送命令包来操作硬盘，这种读写硬盘的方式非常烦琐。BlockIo 建立在 PassThrough 基础之上，用

于读写硬盘扇区，其接口非常简洁。DiskIo 建立在 BlockIo 基础之上，提供了读写硬盘任意地址处数据的能力。通过图 7-10 可以看出这三个 Protocol 之间的关系。

UEFI 和传统 BIOS 都支持对硬盘的读写。UEFI 优于传统 BIOS 的地方在于，UEFI 还提供了对文件系统的支持。下面就来看一下在 UEFI 中是如何操作文件的。

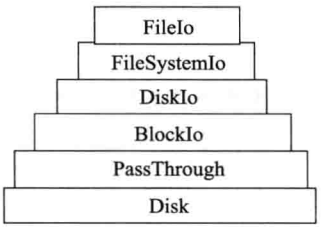


图 7-10 文件系统 Protocol 栈

7.4 文件系统

通常，每个 UEFI 系统至少有一个 ESP(EFI System Partition) 分区，在这个分区上存放了启动文件。既然操作系统加载器以文件的形式存放在 ESP 分区内，UEFI 就需要有读写文件的功能。文件的读写与管理必须通过文件系统来完成，要支持读写文件，UEFI 必须首先能操作 ESP 上的文件系统。ESP 主要用来存放操作系统加载器相关的文件，因而对文件系统的要求比较简单，所以采用 FAT 文件已经可以满足需求。UEFI 内置了 EFI_SIMPLE_FILE_SYSTEM_PROTOCOL（简称 FileSystemIo）用于操作 FAT 文件系统。从图 7-10 显示的硬盘上的 Protocol 栈可以看出，FileSystemIo 建立在 DiskIo 基础之上。代码清单 7-27 是 FileSystemIo 的结构体。

代码清单 7-27 EFI_SIMPLE_FILE_SYSTEM_PROTOCOL 数据结构

```
typedef struct _EFI_SIMPLE_FILE_SYSTEM_PROTOCOL {
  UINT64 Revision;
  EFI_SIMPLE_FILE_SYSTEM_PROTOCOL_OPEN_VOLUME OpenVolume;//打开卷并获得根目录句柄
} EFI_SIMPLE_FILE_SYSTEM_PROTOCOL;
 //打开卷，获得该卷上的根目录句柄，也就是根目录文件操作接口 EFI_FILE_PROTOCOL
typedef EFI_STATUS (EFIAPI *EFI_SIMPLE_FILE_SYSTEM_PROTOCOL_OPEN_VOLUME) (
  IN EFI_SIMPLE_FILE_SYSTEM PROTOCOL *This,
  OUT EFI_FILE_PROTOCOL **Root                          //返回根目录句柄
);
```

通过 EFI_SIMPLE_FILE_SYSTEM_PROTOCOL 中的 OpenVolume，我们就可以获得 FAT 文件系统上的根目录句柄，目录句柄（EFI_FILE_PROTOCOL）包含了操作该目录里文件的文件操作接口。

【示例 7-9】 使用 EFI_SIMPLE_FILE_SYSTEM_PROTOCOL。

```
EFI_STATUS Status;
EFI_SIMPLE_FILE_SYSTEM_PROTOCOL *SimpleFileSystem;
EFI_FILE_PROTOCOL *Root;
Status = gBS->LocateProtocol(&gEfiSimpleFileSystemProtocolGuid,
        NULL,(VOID**)&SimpleFileSystem);
```

```
if (EFI_ERROR(Status)) {
  //未找到 EFI_SIMPLE_FILE_SYSTEM_PROTOCOL
  return Status;
}
//得到根目录句柄
Status = SimpleFileSystem->OpenVolume(SimpleFileSystem, &Root);
```

在得到该分区文件系统上的根目录 EFI_FILE_PROTOCOL 实例的指针后，就可以操作该分区上的文件了。关于 FAT 文件系统的具体实现，感兴趣的读者可以到 EDK II FAT-Driver PROJECT 中查看。

7.5　文件操作

前面我们讲到 OpenVolume 用于打开卷，并得到卷上根目录的句柄，利用这个句柄可以操作卷上的文件。这个句柄的类型是 *EFI_FILE_PROTOCOL。或许读者会感到奇怪，为什么文件句柄的类型不是 xx_HANDLE，而是指向 xx_PROTOCOL 的指针？通常 Protocol 被安装到设备控制器中，通过 Protocol 来操作设备，也就是说，PROTOCOL 对应于设备。文件句柄与 EFI_FILE_PROTOCOL，文件与设备分别是什么关系呢？先来看一下代码清单 7-28。

代码清单 7-28　EFI_FILE_PROTOCOL 与 EFI_FILE_HANDLE 结构体定义

```
//SimpleFileSystem.h
typedef struct _EFI_FILE_PROTOCOL EFI_FILE_PROTOCOL;
typedef struct _EFI_FILE_PROTOCOL *EFI_FILE_HANDLE;
```

可以看出，EFI_FILE_HANDLE 是指向 EFI_FILE_PROTOCOL（简称 FileIo）的指针。在 Linux 系统中，设备总是视为一种特殊的文件。在这里，文件视为一种特殊的设备，EFI_FILE_PROTOCOL 被安装到文件设备上，其指针被当做文件句柄。EFI_FILE_PROTOCOL 用于操作设备上的文件或目录。代码清单 7-29 是 EFI_FILE_PROTOCOL 的结构体。

代码清单 7-29　EFI_FILE_PROTOCOL 的结构体

```
typedef struct _EFI_FILE_PROTOCOL {
  UINT64 Revision;          EFI_FILE_OPEN Open;
  EFI_FILE_CLOSE Close;     EFI_FILE_DELETE Delete;
  EFI_FILE_READ Read;
  EFI_FILE_WRITE Write;
  EFI_FILE_GET_POSITION GetPosition;  EFI_FILE_SET_POSITION SetPosition;
  EFI_FILE_GET_INFO GetInfo;    //获得文件属性
  EFI_FILE_SET_INFO SetInfo;    //设置文件属性
  EFI_FILE_FLUSH Flush;         //将文件中修改过的内容全部更新到设备
  //下面四个函数是异步操作
  EFI_FILE_OPEN_EX OpenEx;
```

```
  EFI_FILE_READ_EX ReadEx;
  EFI_FILE_WRITE_EX WriteEx;
  EFI_FILE_FLUSH_EX FlushEx;
} EFI_FILE_PROTOCOL;
```

读写文件与我们常用的标准 C 读写文件大同小异。示例 7-10 展示了如何利用 FileIo 写文件。

【示例 7-10】 通过 FileIo 写文件。

```
UINTN BufSize;
CHAR16 *Textbuf= (CHAR16*)L"This is test file\n";
EFI_FILE_PROTOCOL *FileHandle = 0;
//创建新文件，如果文件已经存在，则打开
Status = Root->Open(Root,
  &FileHandle,                                                    //返回文件句柄
  (CHAR16*)L"testfileprotocol.txt",                               //文件名
  EFI_FILE_MODE_CREATE | EFI_FILE_MODE_READ | EFI_FILE_MODE_WRITE, //打开模式
  0);
//CheckStatus 作用是检查状态值 Status 是否错误，发现错误时输出错误信息
CheckStatus(L" EFI_FILE_PROTOCOL Create file ");
if( FileHandle && !EFI_ERROR(Status)) {
  //写文件
  BufSize = StrLen(Textbuf) * 2;
  Status = FileHandle ->Write (FileHandle, & BufSize, Textbuf);
  CheckStatus(L"EFI_FILE_PROTOCOL Write file ");
  //关闭文件
  Status = FileHandle ->Close (FileHandle);
  CheckStatus(L"EFI_FILE_PROTOCOL Close file ");
}
```

下面我们详细介绍 EFI_FILE_PROTOCOL 各个函数的用法。

7.5.1 打开文件

FileIo 的 Open 函数用于打开文件或目录，代码清单 7-30 是其函数原型。

代码清单 7-30 FileIo 的 Open 函数原型

```
//打开或创建文件
typedef EFI_STATUS(EFIAPI *EFI_FILE_OPEN) (
  IN EFI_FILE_PROTOCOL *This,     //EFI_FILE_PROTOCOL 实例，通常它是一个目录的句柄
  OUT EFI_FILE_PROTOCOL **NewHandle,//返回打开文件的句柄
  IN CHAR16 *FileName,            //文件名，以 NULL 结尾的 Unicode 字符串
  IN UINT64 OpenMode,             //打开模式，表明打开文件用于读、写、创建，或者三种的组合
  IN UINT64 Attributes            //文件属性，仅在产生文件时有效
);
```

在 Open 函数中，This 指针是一个目录的句柄，如果 FileName 指定的文件路径是相对

路径（以非 \ 字符开头），那么这个相对路径起始于 This 对应的目录；如果 FileName 指定的文件路径是绝对路径（以 \ 开头），那么这个文件路径起始于 This 所在文件系统的根目录。在 FileName 路径中，“.”表示当前目录，“..”表示父目录。

OpenMode 是打开模式，表示文件用于读、写、创建或者三种的组合。表 7-7 给出了文件打开模式的简要说明。

表 7-7　文件打开模式

文件打开模式	用　　途
EFI_FILE_MODE_READ	文件用于读
EFI_FILE_MODE_WRITE	文件用于写
EFI_FILE_MODE_CREATE	若文件不存在，则创建

目前，EDK2 仅支持三种打开模式，其余的打开模式将会返回 EFI_INVALID_PARAMETER。这三种有效的打开模式或组合是：EFI_FILE_MODE_READ、EFI_FILE_MODE_READ|EFI_FILE_MODE_WRITE 和 EFI_FILE_MODE_READ|EFI_FILE_MODE_WRITE|EFI_FILE_MODE_CREATE。创建文件时，可以指定文件属性，见表 7-8。

表 7-8　文件属性

文件属性	功　　能
EFI_FILE_READ_ONLY	只读文件
EFI_FILE_HIDDEN	隐藏文件
EFI_FILE_SYSTEM	系统文件
EFI_FILE_RESERVED	保留
EFI_FILE_DIRECTORY	目录
EFI_FILE_ARCHIVE	归档文件
EFI_FILE_VALID_ATTR	有效属性位

打开成功后，系统为文件（或目录）生成文件对象 FAT_IFILE，并返回文件对象中的 EFI_FILE_PROTOCOL 指针作为文件句柄。文件对象 FAT_IFILE 是 FAT 文件系统的内部数据结构，不会暴露给开发者，在这里我们仅列出其结构（见代码清单 7-31），感兴趣的读者可以到 EDK II FAT-Driver PROJECT 中查看其详细代码。

代码清单 7-31　FAT_IFILE 数据结构

```
//FAT 文件对象，&Handle 会作为文件句柄
typedef struct {
  UINTN Signature;
  EFI_FILE_PROTOCOL Handle;      //此地址将作为文件（或目录）句柄
  UINT64 Position;
  BOOLEAN ReadOnly;
  struct _FAT_OFILE *OFile;      //打开文件，文件打开后生成 OFile 对象
```

```
  LIST_ENTRY Tasks;
  LIST_ENTRY Link;
} FAT_IFILE;
```

下面我们用几个简单的示例来讲述 Open 的用法（见示例 7-11 ~示例 7-13）。

【示例 7-11】 在根目录下生成 efi 目录。

```
EFI_FILE_PROTOCOL *EfiDirectory = 0;
Status = Root->Open(
  Root,                                                          //指向目录
  &EfiDirectory,                                                 //新目录的句柄
  (CHAR16*)L"efi",                                               //全路径为 \efi\
  EFI_FILE_MODE_CREATE|EFI_FILE_MODE_READ|EFI_FILE_MODE_WRITE,//创建并打开
  EFI_FILE_DIRECTORY                                             //指定的文件名为目录
);
```

【示例 7-12】 在 efi 目录下生成 readme.txt 文件。

```
EFI_FILE_PROTOCOL *ReadMe = 0;
Status = EfiDirectory->Open(
  EfiDirectory,                                  //This 指针，指向目录 \efi\
  &ReadMe,                                       //新文件的句柄
  (CHAR16*)L"readme.txt",                        //文件全路径为 \efi\readme.txt
  EFI_FILE_MODE_CREATE|EFI_FILE_MODE_READ|EFI_FILE_MODE_WRITE, //创建并打开
  0                                              //生成普通文件，0 表示默认属性
);
```

【示例 7-13】 在 efi 的目录下生成隐藏文件 system。

```
EFI_FILE_PROTOCOL *SystemFile = 0;
Status = EfiDirectory->Open(
  EfiDirectory,                                  //This 指针，指向目录 \efi\
  &SystemFile,                                   //新文件句柄
  (CHAR16*)L"..\\system",                        //完整路径为 \efi\..\system
  EFI_FILE_MODE_CREATE | EFI_FILE_MODE_READ | EFI_FILE_MODE_WRITE, //创建并打开
  EFI_FILE_SYSTEM | EFI_FILE_HIDDEN              //生成系统隐藏文件
);
```

打开文件之后，就可以对文件进行读、写等操作了。

7.5.2 读文件

Read 函数用于读文件或目录，代码清单 7-32 所示是其函数原型。

代码清单 7-32 FileIo 的 Read 函数原型

```
//从文件当前位置或目录读数据
typedef EFI_STATUS(EFIAPI *EFI_FILE_READ)(
  IN EFI_FILE_PROTOCOL *This,                    //文件句柄
```

```
  IN OUT UINTN *BufferSize,        // 输入：要读出的数据长度；输出：实际读出的数据长度
  OUT VOID *Buffer                 // 读缓冲区
  );
```

This 指定的句柄可以是文件句柄，也可以是目录句柄。如果 This 指向文件，则 Read 函数将从文件当前位置处读 BufferSize 个字节到 Buffer 缓冲区。如果从文件当前位置到文件末尾小于 BufferSize，则读至文件末尾。返回后，BufferSize 存放实际读取的字节数。示例 7-14 展示了如何读取文件。

【示例 7-14】 用 Read 读文件。

```
// 下面的示例演示了读取根目录下的 system 文件
EFI_STATUS TestReadFile()
{
  EFI_STATUS  Status = 0;
  EFI_FILE_PROTOCOL                *Root;
  EFI_FILE_PROTOCOL *SystemFile = 0;
  CHAR16 Buf[65];
  UINTN BufSize = 64;
  // 获得 FAT 文件系统根目录句柄
  SAFECALL(Status = GetFileIo(&Root));
  // 打开根目录下的 system 文件
  SAFECALL(Status = Root ->Open(Root, & SystemFile, (CHAR16*)L"system",
    EFI_FILE_MODE_READ | EFI_FILE_MODE_WRITE, 0));
  // 读文件，最多读 64 字节
  Status = SystemFile -> Read ( SystemFile, &BufSize, Buf);
  if(!EFI_ERROR(Status)){
    Buf[BufSize] = 0;
    Print(L"%s\n", Buf);
  }
  // 关闭文件句柄
  Status = SystemFile->Close(SystemFile);
  return Status;
}
```

如果 This 指向目录，那么 Read 函数将读取该目录下的文件及目录（不包含子目录下的文件和目录）的目录项 EFI_FILE_INFO。每次 Read 读取当前位置的一个文件（或目录）的目录项，然后当前位置指向下一个文件（或目录）的目录项。读取目录项时，如果给定的缓冲区小于目录项的大小，则返回 EFI_BUFFER_TOO_SMALL，同时 BufferSize 返回需要的缓冲区大小。如果当前位置到达该目录下目录项的末尾，则返回 EFI_SUCCESS，同时 BufferSize 返回 0。代码清单 7-33 是目录项的数据结构。示例 7-15 展示了如何用 Read 函数读取目录项。

代码清单 7-33　EFI_FILE_INFO 结构体

```
typedef struct {
  UINT64 Size;            // EFI_FILE_INFO 结构的大小，包括 FileName 字符串（及串尾的 0）的长度
  UINT64 FileSize;        // 文件的长度
```

```
  UINT64 PhysicalSize;            //文件在文件系统卷上占用的实际字节数
  EFI_TIME CreateTime;
  EFI_TIME LastAccessTime;
  EFI_TIME ModificationTime;
  UINT64 Attribute;
  CHAR16 FileName[1];
} EFI_FILE_INFO;
```

【示例 7-15】 用 Read 函数读取目录项。

```
typedef VOID (*AccessFileInfo)(EFI_FILE_INFO* FileInfo);
VOID ListFileInfo(EFI_FILE_INFO* FileInfo)
{
  Print(L"Size : %d\n FileSize:%d \n Physical Size:%d\n",
    FileInfo->Size,
    FileInfo->FileSize,
    FileInfo->PhysicalSize);
  Print(L"%s\n", FileInfo->FileName);
}
EFI_STATUS ListDirectory(EFI_FILE_PROTOCOL* Directory, AccessFileInfo callbk)
{
  UINTN BufferSize;
  UINTN ReadSize;
  EFI_STATUS  Status = 0;
  EFI_FILE_INFO* FileInfo;
  //预先分配内存给 FileInfo
  BufferSize = sizeof(EFI_FILE_INFO) + sizeof(CHAR16) * 512;
  RETURN_ERR(Status=gBS->AllocatePool( EfiBootServicesCode,
    fferSize, (VOID**)&FileInfo));
  //遍历目录项
  while(1){
    ReadSize = BufferSize;
    //读当前目录项
    Status = Directory -> Read(Directory, &ReadSize, FileInfo);
    if(Status == EFI_BUFFER_TOO_SMALL){
      //缓存区太小，重新分配足够的内存
      BufferSize = ReadSize;
      BREAK_ERR(Status = gBS -> FreePool( FileInfo));
      BREAK_ERR(Status = gBS -> AllocatePool( EfiBootServicesCode,
        BufferSize, (VOID**)&FileInfo));
      //当成功分配到足够的内存后，重新读当前目录项
      BREAK_ERR(Status = Directory-> Read(Directory, &ReadSize,FileInfo));
    }
    //读到目录项末尾时，退出循环
    if(ReadSize == 0) break;
    //Read 返回错误，退出循环
    BREAK_ERR(Status);
    //处理当前目录项
```

```
    callbk(FileInfo);
  }
  Status = gBS -> FreePool( FileInfo);
  return 0;
}
```

7.5.3 写文件

Write 函数用于写数据到文件，代码清单 7-34 是其函数原型。

代码清单 7-34　FileIo 的 Write 函数原型

```
//写数据到文件
typedef EFI_STATUS(EFIAPI *EFI_FILE_WRITE)(
  IN EFI_FILE_PROTOCOL *This,          //文件句柄
  IN OUT UINTN *BufferSize,            //输入：要写入的数据长度；输出：实际写入的数据长度
  IN VOID *Buffer                      //待写入数据
  );
```

Write 只能写数据到文件，不能写数据到目录。通常，Write 函数会写 BufferSize 指定的字节数到文件中（实际写的字节数等于指定要写的字节数），仅在遇到错误时（例如，卷上没有多余空间时）会写部分数据到文件，此时 BufferSize 返回实际写的字节数。示例 7-16 展示了如何用 Write 函数写文件。

【示例 7-16】 用 Write 函数写文件。

```
//示例：写数据到文件 readme.txt 中
EFI_STATUS TestWrite( EFI_FILE_PROTOCOL* Root)
{
  EFI_STATUS Status = 0;
  UINTN BufSize;
  CHAR16 *Buf= (CHAR16*)L"This is test file\n";
  EFI_FILE_PROTOCOL *ReadMe = 0;
  //创建新文件，如果文件已经存在，则打开
  Status = Root->Open(Root,
    &ReadMe, //获得文件句柄
    (CHAR16*)L"readme.txt",
    EFI_FILE_MODE_CREATE | EFI_FILE_MODE_READ | EFI_FILE_MODE_WRITE, //打开模式
    0);
  if( ReadMe && !EFI_ERROR(Status)) {
    //写文件
    BufSize = StrLen(Buf) * 2;
    Status = ReadMe ->Write (ReadMe, &BufSize, Buf);
    //关闭文件
    Status = ReadMe ->Close (ReadMe);
  }
  return Status;
}
```

7.5.4 关闭文件（句柄）

文件操作完毕后，要关闭文件句柄。代码清单 7-35 是 Close 函数原型。

代码清单 7-35 FileIo 的 Close 函数原型

```
//关闭文件句柄
typedef EFI_STATUS(EFIAPI *EFI_FILE_CLOSE)(  IN EFI_FILE_PROTOCOL  *This  );
```

Close 函数很简单，只有一个参数，即要关闭的文件句柄，如在 TestWrite 示例中：

```
Status = ReadMe->Close(ReadMe);
```

7.5.5 其他文件操作

文件的创建、打开、读、写、关闭是比较常用的几种文件操作，除此之外，还有一些其他的文件操作，如设定或读取文件的读写位置，设定或读取文件属性、信息，删除文件，Flush 文件等。下面详细介绍这几种操作。

1. 文件读写位置

文件打开后，系统为文件生成 FAT_IFILE 数据结构，这个结构中保存了文件位置等信息，每次读写都从当前位置处开始，读写后自动更新 FAT_IFILE 内的文件位置。EFI_FILE_PROTOCOL 提供了 GetPosition 用于获取文件的当前位置；SetPosition 用于设置文件的当前位置。代码清单 7-36 是这两个函数的原型。

代码清单 7-36 FileIo 的 GetPosition 和 SetPosition 函数原型

```
/** 返回文件的当前读写位置（以字节为单位）
  @retval EFI_SUCCESS              成功获取当前位置
  @retval EFI_UNSUPPORTED          This 指向目录句柄，无法获得目录的当前位置
  @retval EFI_DEVICE_ERROR         This 指向的文件已经删除或其他设备错误
**/
typedef EFI_STATUS(EFIAPI *EFI_FILE_GET_POSITION)(
  IN EFI_FILE_PROTOCOL *This,      //文件句柄
  OUT UINT64 *Position             //返回文件的当前位置（从文件开头算起）
  );

/** 设置文件的当前读写位置（以字节为单位）
  @retval EFI_SUCCESS              成功设置文件位置
  @retval EFI_UNSUPPORTED          This 指向目录句柄，无法设置目录的当前位置
  @retval EFI_DEVICE_ERROR         This 指向的文件已经删除或其他设备错误
**/
typedef EFI_STATUS(EFIAPI *EFI_FILE_SET_POSITION)(
  IN EFI_FILE_PROTOCOL *This,      //文件句柄
  IN UINT64 Position               //要设定的文件位置（从文件开头算起）
  );
```

2. 读写文件信息

有时我们希望能得到文件的属性等信息，而不是文件的内容。例如，在读文件之前，我们希望能根据文件大小准备好读缓冲区。EFI_FILE_PROTOCOL 提供了 GetInfo 用于读取文件信息；SetInfo 用于设置文件信息。代码清单 7-37 是这两个函数的原型。

代码清单 7-37　FileIo 的 GetInfo、SetInfo 函数原型

```
//获取文件或文件系统的相关信息
typedef EFI_STATUS(EFIAPI *EFI_FILE_GET_INFO)(
  IN EFI_FILE_PROTOCOL *This,      //文件（或目录）句柄
  IN EFI_GUID *InformationType,    //要获取信息的类型标识符
  IN OUT UINTN *BufferSize,        //输入：缓冲区大小；输出：返回数据的长度
  OUT VOID *Buffer                 //读缓冲区，数据类型由 InformationType 决定
  );
//设置文件（或文件系统）信息
typedef EFI_STATUS(EFIAPI *EFI_FILE_SET_INFO)(
  IN EFI_FILE_PROTOCOL *This,      //文件（或目录）句柄
  IN EFI_GUID *InformationType,    //要设置信息的类型标识符
  IN UINTN BufferSize,             //InformationType 指定的数据的字节数
  IN VOID *Buffer                  //InformationType 指定的数据
  );
```

InformationType 可以是下面三种类型：EFI_FILE_INFO_ID、EFI_FILE_SYSTEM_INFO_ID 和 EFI_FILE_SYSTEM_VOLUME_LABEL_ID，见表 7-9。

表 7-9　InformationType 的三种类型

信息标识	数据类型	GUID
EFI_FILE_INFO_ID	EFI_FILE_INFO	gEfiFileInfoGuid
EFI_FILE_SYSTEM_INFO_ID	EFI_FILE_SYSTEM_INFO	gEfiFileSystemInfoGuid
EFI_FILE_SYSTEM_VOLUME_LABEL_ID	EFI_FILE_SYSTEM_VOLUME_LABEL	gEfiFileSystemVolumeLabelInfoIdGuid

EFI_FILE_INFO 在 Read 函数中已经讲过，我们再回顾一下这个数据结构。

```
typedef struct {
  UINT64 Size;          //EFI_FILE_INFO 结构的大小，包括 FileName 字符串（及串尾的 0）的长度
  UINT64 FileSize;      //文件的长度
  UINT64 PhysicalSize;//文件在文件系统卷上占用的实际字节数
  EFI_TIME CreateTime;
  EFI_TIME LastAccessTime;
  EFI_TIME ModificationTime;
  UINT64 Attribute;
  CHAR16 FileName[1];
} EFI_FILE_INFO;
```

EFI_FILE_SYSTEM_INFO 包含了文件系统的相关信息，代码清单 7-38 是其结构体。

代码清单 7-38 EFI_FILE_SYSTEM_INFO 结构体

```
typedef struct {
  UINT64 Size; //EFI_FILE_SYSTEM_INFO 结构的大小，包括 VolumeLabel（及串尾的 0）的长度
  BOOLEAN ReadOnly;        //只读卷
  UINT64 VolumeSize;       //卷大小
  UINT64 FreeSpace;        //文件系统剩余空间
  UINT32 BlockSize;        //文件以此大小的块增加长度
  CHAR16 VolumeLabel[1]; //卷名字符串（以 NULL 结尾）
} EFI_FILE_SYSTEM_INFO;
```

EFI_FILE_SYSTEM_INFO 中只有 VolumeLabel 可写，其他成员都是只读的。也就是说，调用 SetInfo 设置 EFI_FILE_SYSTEM_INFO 时只能更新 VolumeLabel 域。

EFI_FILE_SYSTEM_VOLUME_LABEL 包含了用于表示卷标的字符串，其数据结构如下所示。

```
typedef struct { CHAR16 VolumeLabel[]; } EFI_FILE_SYSTEM_VOLUME_LABEL;
```

3. 删除文件

Delete 函数用于删除文件，代码清单 7-39 是其函数原型。

代码清单 7-39 FileIo 的 Delete 函数原型

```
/** 关闭文件句柄（This）并从介质上删除文件
  @param This                          要删除文件（或目录）的句柄
  @retval EFI_SUCCESS                  成功删除。句柄被关闭
  @retval EFI_WARN_DELETE_FAILURE      文件句柄被关闭，但文件没有被删除
**/
typedef EFI_STATUS(EFIAPI *EFI_FILE_DELETE)( IN EFI_FILE_PROTOCOL  *This);
```

4. Flush 文件

Flush 用于将文件所有改变的内容更新到介质中，代码清单 7-40 是其函数原型。

代码清单 7-40 FileIo 的 Flush 函数原型

```
//将文件所有改变的内容更新到介质
typedef EFI_STATUS(EFIAPI *EFI_FILE_FLUSH)(IN EFI_FILE_PROTOCOL  *This);
```

7.5.6 异步文件操作

前面讲过，BlockIo 和 DiskIo 都提供异步操作，DiskIo 之上的 FileIo 同样也提供异步文件操作。EFI_FILE_PROTOCOL 提供了异步打开（OpenEx）、异步读（ReadEx）、异步写（WriteEx）及异步 Flush(FlushEx) 操作。

异步文件操作需提供一个 EFI_FILE_IO_TOKEN 类型的令牌，代码清单 7-41 是这个令

牌（Token）的数据结构。与 EFI_DISK_IO2_TOKEN 和 EFI_BLOCK_IO2_TOKEN 稍有不同，EFI_FILE_IO_TOKEN 中多了 BufferSize 和 Buffer 这两个成员。BufferSize 和 Buffer 与阻塞操作中的 BufferSize 和 Buffer 含义相同。BufferSize 作为输入时，表示请求操作的字节数；作为输出时，表示实际操作的字节数。

代码清单 7-41　EFI_FILE_IO_TOKEN 结构体

```
typedef struct {
  EFI_EVENT Event;          //此事务对应的事件
  EFI_STATUS Status;        //Event 触发后，事务的状态
  //下面两个域仅用于 ReadEx 和 WriteEx，对 OpenEx 和 FlushEx 无效
  UINTN BufferSize;
  VOID  *Buffer;
} EFI_FILE_IO_TOKEN;
```

1. 异步打开

OpenEx 用于异步打开文件或目录，代码清单 7-42 是其函数原型。当 Token 无效（Token 为 Null 或 Token->Event 为空）时，该操作退化为阻塞操作。

代码清单 7-42　FileIo 的 OpenEx 函数原型

```
/** OpenEx 函数
异步打开或创建文件
**/
typedef EFI_STATUS(EFIAPI *EFI_FILE_OPEN_EX) (
  IN EFI_FILE_PROTOCOL *This,     //EFI_FILE_PROTOCOL 实例，通常它是一个目录的句柄
  OUT EFI_FILE_PROTOCOL **NewHandle, //返回打开文件的句柄
  IN CHAR16 *FileName,            //文件名，以 NULL 结尾的 Unicode 字符串
  IN UINT64 OpenMode,             //打开模式，表明打开文件用于读、写、创建，或者三种的组合
  IN UINT64 Attributes,           //文件属性，仅在产生文件时有效
  IN OUT EFI_FILE_IO_TOKEN *Token //每个事务都需要一个有效的 Token
);
```

示例 7-17 演示了如何使用异步 Open 打开文件。

【示例 7-17】 异步打开文件。

```
EFI_STATUS TestAsyncOpen(EFI_FILE_PROTOCOL* Root)
{
  EFI_STATUS  Status = 0;
  EFI_FILE_IO_TOKEN OpenToken;
  EFI_FILE_PROTOCOL* AyncFile;
  UINTN Index = 0;
  //为 Open 操作的 Token 生成 Event
  SAFECALL(Status=gBS->CreateEvent(0, TPL_NOTIFY, NULL, NULL,&OpenToken.Event));
  Status = Root->OpenEx(Root, &AyncFile, L"aync.txt",
```

```
    EFI_FILE_MODE_CREATE | EFI_FILE_MODE_READ | EFI_FILE_MODE_WRITE,
    0, &OpenToken);
  IF_ERR(Status){
    gBS->CloseEvent(OpenToken.Event);
    return Status;
  }
  //异步 Open 请求成功，此时可以做其他的工作
  ...
  //等待异步 Open 完成
  gBS->WaitForEvent(1, &OpenToken.Event, &Index);
  //检查事务状态
  Print(L"Anyc Open Status:%d\n", OpenToken.Status);
  gBS->CloseEvent(OpenToken.Event);
  return Status;
}
```

2. 异步读 / 写文件

ReadEx 用于异步读文件；WriteEx 用于异步写文件。代码清单 7-43 是这两个函数的函数原型。像其他异步操作一样，Token 无效时，操作退化为阻塞操作。若 Token 有效，则该异步操作加入到执行队列中。

执行 WriteEx 写文件时，如果文件句柄为目录项，则函数会返回 EFI_UNSUPPORTED；如果试图写数据到被删除的文件，则函数会返回 EFI_DEVICE_ERROR；如果文件文件以只读方式打开，则函数会返回 EFI_ACCESS_DENIED。

代码清单 7-43　FileIo 的 ReadEx 和 WriteEx 函数原型

```
//ReadEx 函数，用于异步读文件
typedef EFI_STATUS (EFIAPI *EFI_FILE_READ_EX) (
  IN EFI_FILE_PROTOCOL *This,            //文件（或目录）句柄
  IN OUT EFI_FILE_IO_TOKEN *Token        //事务令牌
);
//WriteEx 函数，用于异步写数据到文件
Typedef EFI_STATUS(EFIAPI *EFI_FILE_WRITE_EX) (
  IN EFI_FILE_PROTOCOL *This,            //文件（或目录）句柄
  IN OUT EFI_FILE_IO_TOKEN *Token        //事务 Token
);
```

有两种方式可以完成异步文件读写：一种是使用 WaitForEvent 等待异步读结束后再执行后续任务；另一种是在事件的 Notification 函数中完成后续任务。示例 7-18 展示了前一种方式的异步写，示例 7-19 展示了后一种方式的异步读。

【示例 7-18】 WaitForEvent 方式的异步写文件。

```
//使用普通事件（事件类型为 0）异步写示例
EFI_STATUS TestAsyncWrite(EFI_FILE_PROTOCOL* File)
```

```
{
  EFI_STATUS  Status = 0;
  EFI_FILE_IO_TOKEN WriteToken;
  UINTN Index = 0;
  //1: 初始化 Token
  SAFECALL(Status = gBS->CreateEvent(0, TPL_NOTIFY, NULL, NULL,&WriteToken.Event));
  WriteToken.Buffer = L"Hello Async Write";
  WriteToken.BufferSize= StrLen((CHAR16*)WriteToken.Buffer) * 2;
  //2: 发出异步写请求
  Status = File -> WriteEx(File, &WriteToken);
  IF_ERR(Status){
    gBS->CloseEvent(WriteToken.Event);
    Print(L"Aync Write %r\n", Status);
    return Status;
  }
  //3: 等待写事务完成
  gBS->WaitForEvent(1, &WriteToken.Event, &Index);
  //4: 检查写事务状态，并进行其他后续工作
  Print(L"Anyc Open Status:%d\n", WriteToken.Status);
  //5: 退出函数前清理资源
  gBS->CloseEvent(WriteToken.Event);
  return Status;
}
```

【示例 7-19】 Notification 方式的异步读文件。

```
VOID ReadNotification(EFI_EVENT event, VOID* Context)
{
  EFI_FILE_IO_TOKEN *ReadToken = (EFI_FILE_IO_TOKEN *)Context;
  //检查 ReadToken.Status
  //检查 ReadToken.BufferSize
  (VOID)ReadToken;
}
//使用 EVT_NOTIFY_SIGNAL 事件进行异步读文件
EFI_STATUS TestAsyncRead(EFI_FILE_PROTOCOL* File)
{
  EFI_STATUS  Status = 0;
  EFI_FILE_IO_TOKEN ReadToken;
  UINTN Index = 0;
  //1: 初始化 Token
  SAFECALL(Status = gBS->CreateEvent(EVT_NOTIFY_SIGNAL, TPL_NOTIFY,ReadNotification,
      (VOID*)&ReadToken, &ReadToken.Event));
  ReadToken.BufferSize= 1024;
  IF_ERR(Status = gBS -> AllocatePool( EfiBootServicesCode, ReadToken.BufferSize,
      (VOID**)&ReadToken.Buffer)){
    goto ERR_0;
  }
  //2: 发出异步读请求
  Status = File ->ReadEx(File, &ReadToken);
```

```
  IF_ERR(Status){
    goto ERR_1;
  }
  //3: 等待异步读完成
  gBS->WaitForEvent(1, &ReadToken.Event, &Index);
  //此时 ReadNotification 已执行完毕
  //4: 检查读事务状态
  Print(L"Anyc Write Status:%d\n", ReadToken.Status);
  Print(L"Anyc Write %d bytes\n", ReadToken.BufferSize);
  //5: 退出函数前清理资源
ERR_1:
  gBS->FreePool(ReadToken.Buffer);
ERR_0:
  gBS->CloseEvent(ReadToken.Event);
  return Status;
}
```

3. 异步 Flush

FlushEx 是 Flush 的异步版本，代码清单 7-44 是其函数原型。异步 Flush 比较简单，此处不再赘述。

代码清单 7-44　FileIo 的 FlushEx 函数原型

```
//FlushEx 函数，用于将指定文件中修改过的内容全部更新到设备
typedef EFI_STATUS(EFIAPI *EFI_FILE_FLUSH_EX) (
  IN EFI_FILE_PROTOCOL *This,        //文件（或目录）句柄
  IN OUT EFI_FILE_IO_TOKEN *Token    //事务令牌
 );
```

下面总结一下异步操作文件的步骤：

1）初始化 Token，包括 Token 中的 Event，如果是 ReadEx 或 WriteEx，还要初始化 Buffer 和 BufferSize。

2）发出异步命令。

3）等待 Token 中的事件触发。

4）检查 Token 中的 Status。

5）清理资源。

7.5.7　EFI_SHELL_PROTOCOL 中的文件操作

在 UEFI 中还有一组接口用于操作文件，那便是 EFI_SHELL_PROTOCOL 提供的文件操作函数。EFI_SHELL_PROTOCOL 中的文件操作接口对 EFI_FILE_PROTOCOL 进行了封装和扩充，除了提供基本的文件访问函数之外，还提供了相对目录的操作方式，以及 stdin、

stdout、stderr 及文件查找函数。上文讲过，只有 Shell 环境下运行的程序才能使用 EFI_SHELL_PROTOCOL，因而只有 Shell 下的应用程序才能使用 EFI_SHELL_PROTOCOL 中的文件操作函数。

代码清单 7-45 列出了 EFI_SHELL_PROTOCOL 提供的文件操作函数。

代码清单 7-45　EFI_SHELL_PROTOCOL 中的文件操作函数

```
typedef struct _EFI_SHELL_PROTOCOL {
  EFI_SHELL_GET_DEVICE_PATH_FROM_FILE_PATH  GetDevicePathFromFilePath;
  EFI_SHELL_GET_FILE_PATH_FROM_DEVICE_PATH  GetFilePathFromDevicePath;
...
  EFI_SHELL_GET_CUR_DIR GetCurDir;
  EFI_SHELL_SET_CUR_DIR SetCurDir;
  EFI_SHELL_GET_FILE_INFO GetFileInfo;
  EFI_SHELL_SET_FILE_INFO SetFileInfo;
  EFI_SHELL_OPEN_FILE_BY_NAME OpenFileByName;
  EFI_SHELL_CLOSE_FILE CloseFile;
  EFI_SHELL_CREATE_FILE CreateFile;
  EFI_SHELL_READ_FILE ReadFile;
  EFI_SHELL_WRITE_FILE WriteFile;
  EFI_SHELL_DELETE_FILE DeleteFile;
  EFI_SHELL_DELETE_FILE_BY_NAME DeleteFileByName;
  EFI_SHELL_GET_FILE_POSITION GetFilePosition;
  EFI_SHELL_SET_FILE_POSITION SetFilePosition;
  EFI_SHELL_FLUSH_FILE FlushFile;
  EFI_SHELL_FIND_FILES FindFiles;
  EFI_SHELL_FIND_FILES_IN_DIR FindFilesInDir;
  EFI_SHELL_GET_FILE_SIZE GetFileSize;
} EFI_SHELL_PROTOCOL;
```

相信读者根据函数名就能够知道函数的作用，在此就不一一解释了。下面我们通过文件读写的例子看一下 EFI_SHELL_PROTOCOL 中常用文件操作的用法。代码清单 7-46 列出了 CreateFile、WriteFile、ReadFile、OpenFileByName 这几个常用操作的函数原型。示例 7-20 展示了如何利用这几个函数读写文件。

代码清单 7-46　EFI_SHELL_PROTOCOL 常用文件操作的函数原型

```
// 函数 CreateFile 用于生成新文件
typedef EFI_STATUS(EFIAPI *EFI_SHELL_CREATE_FILE)(
  IN CONST CHAR16 *FileName,            // 文件名
  IN UINT64 FileAttribs,                // 文件属性
  OUT SHELL_FILE_HANDLE *FileHandle     // 文件句柄
);
// 函数 OpenFileByName 用于打开文件
  typedef EFI_STATUS(EFIAPI *EFI_SHELL_OPEN_FILE_BY_NAME) (
  IN CONST CHAR16 *FileName,            // 文件名
```

```
  OUT SHELL_FILE_HANDLE *FileHandle,    //文件句柄
  IN UINT64 OpenMode                    //打开模式
 );
//函数 WriteFile 用于写文件
typedef EFI_STATUS(EFIAPI *EFI_SHELL_WRITE_FILE)(
  IN SHELL_FILE_HANDLE FileHandle,      //文件句柄
  IN OUT UINTN *BufferSize,             //输入：缓冲区大小；输出：实际写字节数
  IN VOID *Buffer                       //写缓存区
 );
//函数 ReadFile 用于读文件
typedef EFI_STATUS(EFIAPI *EFI_SHELL_READ_FILE) (
  IN SHELL_FILE_HANDLE FileHandle,      //文件句柄
  IN OUT UINTN *ReadSize,               //输入：请求读的字节数；输出：实际读出的字节数
  IN OUT VOID *Buffer                   //读缓存区
  );
```

【示例 7-20】 使用 EFI_SHELL_PROTOCOL 读写文件。

```
EFI_STATUS TestShellFile()
{
  EFI_STATUS Status;
  SHELL_FILE_HANDLE FileHandle;
  UINTN WbufSize, RbufSize = 256;
  CHAR16 *Wbuf= (CHAR16*)L"This is test file\n";
  CHAR16 Rbuf[256] ;
  WbufSize = StrLen(Wbuf) * 2;
  //写文件 testfile.txt
  Status = gEfiShellProtocol -> CreateFile((CONST CHAR16*)L"testfile.txt", 0,
      FileHandle);
  Status = gEfiShellProtocol->WriteFile(FileHandle, &WbufSize, Wbuf);
  Status = gEfiShellProtocol->CloseFile(FileHandle);
  RbufSize = 256;
  //读文件 testfile.txt
  Status = gEfiShellProtocol->OpenFileByName((CONST CHAR16*)L"testfile.txt",
      &FileHandle, EFI_FILE_MODE_READ);
  Status = gEfiShellProtocol->ReadFile(FileHandle, &RbufSize ,Rbuf);
  Status = gEfiShellProtocol->CloseFile(FileHandle);
  return EFI_SUCCESS;
}
```

EFI_SHELL_PROTOCOL 中文件操作相关函数的第一个参数不再是 This 指针，通常是 Shell 文件句柄。

EFI_FILE_PROTOCOL 中可以使用绝对路径，也可以使用相对路径。如果是相对路径，则相对路径起始于 Shell 的当前路径。当前路径可以由 SetCurDir 设置，由 GetCurDir 获得当前路径。回忆一下，EFI_FILE_PROTOCOL 中 Open 函数的相对路径则起始于 Open 函数 This 指定的目录。

在 CreateFile/OpenFileByName/DeleteFileByName 函数中，如果文件名起始于“>v”，那么文件句柄表示以此文件名为名字的环境变量。

例如，下面的代码可以生成环境变量 TestVar。

```
Status = gEfiShellProtocol->CreateFile(L">vTestVar", 0, &FileHandle);
```

还可以使用表 7-10 中的文件名通过 OpenFileByName 获得特殊文件。

表 7-10　特殊文件名

文件名	文件对象
L " >i "	标准输入
L " >o "	标准输出
L " >e "	标准错误
L " NUL "	标准 NUL

示例 7-21 展示了如何创建环境变量 TestVar，并为 TestVar 赋值 L"This is test file\n"。

【示例 7-21】 创建环境变量 TestVar。

```
EFI_STATUS TestVar(CONST CHAR16* var, CHAR16* Wbug)
{
  EFI_STATUS Status;
  SHELL_FILE_HANDLE FileHandle;
  UINTN WbufSize = StrLen(Wbuf) * 2;
  EFI_SHELL_PROTOCOL *gEfiShellProtocol;
  Status = OpenShellProtocol(&gEfiShellProtocol);
  Status = gEfiShellProtocol->CreateFile((CONST CHAR16*)var, 0, &FileHandle);
  Status = gEfiShellProtocol->WriteFile(FileHandle, &WbufSize, Wbuf);
  Status = gEfiShellProtocol->CloseFile(FileHandle);
  return EFI_SUCCESS;
}
VOID Test()
{
  TestVar(L">vTestVar", (CHAR16*)L"This is test file\n");
}
```

执行上述这段代码后，可以通过 set 命令查看环境变量，如图 7-11 所示。

```
FS0:\uefi\> set
     path = .\;FS0:\efi\tools\;FS0:\efi\boot\;FS0:\
profiles = ;Driver1;Debug1;Install;network1;
uefishellsupport = 3
uefishellversion = 2.0
uefiversion = 2.40
     cwd = FS0:\uefi\
 TestVar = This is test file
```

图 7-11　执行示例 7-21 后的环境变量

前面讲过，在 UEFI 中，文件是一种特殊的设备，既然是设备，那么文件也拥有 DevicePath。EFI_SHELL_PROTOCOL 提供了 GetDevicePathFromFilePath 用于获取文件的 DevicePath。

```
typedef EFI_DEVICE_PATH_PROTOCOL *
(EFIAPI *EFI_SHELL_GET_DEVICE_PATH_FROM_FILE_PATH) (IN CONST CHAR16 *Path);
```

- GetFilePathFromDevicePath 用于将 DevicePath 转换为文件路径。

```
typedef CHAR16 *(EFIAPI *EFI_SHELL_GET_FILE_PATH_FROM_DEVICE_PATH)
( IN CONST EFI_DEVICE_PATH_PROTOCOL *Path  );
```

EFI_SHELL_PROTOCOL 还提供了用于搜索目录的函数。FindFiles 在所有文件和目录中搜索，返回所有匹配的文件和目录。

```
typedef EFI_STATUS(EFIAPI *EFI_SHELL_FIND_FILES)(
  IN CONST CHAR16 *FilePattern,          // 文件名模式，可以使用通配符
  OUT EFI_SHELL_FILE_INFO **FileList     // 匹配 FilePattern 的文件列表
);
```

例如，下面的代码可以找出所有的 bmp 文件。

```
EFI_SHELL_FILE_INFO *FileList;
Status = gEfiShellProtocol -> FindFiles(L"*.bmp", &FileList);
```

7.6 本章小结

本章介绍了 GPT 硬盘的格式。相比传统的硬盘，GPT 硬盘有如下优点：

1）使用 64bit 地址，有更大的容量。

2）最多可以有 128 个分区。

3）备份分区表，容错能力增强。

硬盘是一种块设备，利用 BlockIo 可以读取硬盘扇区，利用 DiskIo 可以读取分区内容。在 DiskIo 基础之上是文件系统。UEFI 内置了对 FAT 文件系统的支持，利用 SIMPLE FILE SYSTEM PROTOCOL 可以操作 FAT 分区上的文件系统。FAT 文件系统之上是 EFI_FILE_PROTOCOL。本章讲述了 EFI_FILE_PROTOCOL 读写文件的方法。此外，本章还讲述了 EFI_SHELL_PROTOCOL 中对文件操作的服务。下一章我们以视频解码服务为例讲述如何利用 Protocol 提供服务。

第 8 章 Chapter 8

开发 UEFI 服务

在第 4 章，我们从使用者的角度学习了 Protocol，而本章将从 Protocol 提供者的角度来学习 Protocol。作为提供者，要了解 4 个问题：① Protocol 是什么？② Protocol 安装到什么地方？③ 怎样安装 Protocol？④ 什么时候安装 Protocol？前两个问题在第 4 章已经解释过，这里再简单回顾一下。从语言的角度讲，Protocol 是包含了属性和函数指针的结构体；从功能上讲，Protocol 是提供者和使用者对服务方式的一种约定。从 Protocol 在 UEFI 内核内的组织可以看出，Protocol 安装在 Image 对象句柄中。再来回答第 3 个问题，启动服务提供了 InstallProtocolInterface，用于把 Protocol 安装到设备控制器上，其函数原型如代码清单 8-1 所示。

代码清单 8-1　InstallProtocolInterface 函数原型

```
EFI_STATUS InstallProtocolInterface (
  IN OUT EFI_HANDLE *Handle,                    // Protocol 将安装到这里
  IN EFI_GUID *Protocol,                        // 要安装的 Protocol 的 GUID
  IN EFI_INTERFACE_TYPE InterfaceType,          // 通常为 EFI_NATIVE_INTERFACE
  IN VOID *Interface                            // Protocol 实例
);
```

通常，服务要能够常驻内存。应用程序是不能常驻内存的，只有驱动才可以常驻内存。那么，我们就要用驱动的形式来提供服务，这种驱动称为“服务型驱动”。但服务又与通常的驱动不同，驱动需要特定硬件的支持，而服务则不需要。这就使得设计服务型驱动变得简单，驱动需要安装到特定的控制器上，而服务安装到任何控制器或 Handle 对象上都可以。在第 3 章中讲过，UEFI 中的驱动大致分为两类：一类是符合 UEFI 驱动模型的驱动，称为“UEFI 驱动”；另一类是不遵循 UEFI 驱动模型的驱动，称为“DXE 驱动”。任何一个模块

（包括应用程序模块和驱动模块）启动后都会执行模块入口函数。通常，一个 UEFI 驱动被加载到内存后会在模块入口函数内执行 InstallProtocolInterface，以将 Driver Binding Protocol 和 Component Name Protocol 安装到自身 Handle（或者其他 Handle）上，然后由 Driver Binding Protocol 负责管理驱动 Protocol。相比 UEFI 驱动，服务型驱动要简单许多，在 Image 初始化的时候（即在执行模块入口函数时），将 Protocol 安装到自身 Handle 即可，也就是说，我们要采用 DXE 型驱动开发服务型驱动。通过上面的分析，关于如何开发服务型驱动，我们得出两个重要结论：① 使用服务型驱动提供服务；② 在模块入口函数中安装 Protocol。

服务型驱动分为 3 个部分：Protocol 服务接口设计、Protocol 服务的实现以及 DXE 驱动框架。下面就以视频解码服务为例，从这 3 个方面分别讲述 Protocol 服务的开发。此外，视频解码是通过调用 ffmpeg 接口完成的，因而还要讲述一下 ffmpeg 在 UEFI 中的移植。

8.1 Protocol 服务接口设计

首先要分析一下视频解码 Protocol 需要提供哪些服务。播放视频的一般流程是打开视频文件，逐帧取出视频并显示到屏幕直到视频结束，最后关闭视频文件。那么，视频解码 Protocol 就需要提供相应的服务，它需要提供 OpenVideo 用于打开视频，QueryFrame 用于取得一帧，CloseFrame 用于关闭视频，还需要一个函数来获取视频信息。通常，播放器的尺寸要能根据用户要求放大或缩小，因而还要提供一个函数用于缩放视频。根据 UEFI 命名规范，可以将这个视频解码 Protocol 命名为 EFI_FFDECODER_PROTOCOL。在第 4 章讲过，每个 Protocol 都是一个结构体，按照 EDK2 命名规范，Protocol 结构体名字通常是 Protocol 名字再加上前面的“_”，因而视频解码 Protocol 结构体名字为 _EFI_FFDECODER_PROTOCOL。EFI_FFDECODER_PROTOCOL 是结构体 _EFI_FFDECODER_PROTOCOL 的类型定义。还可以为 EFI_FFDECODER_PROTOCOL 定义一个别名 EFI_FFDECODER。在 Protocol 中还有一个域 Revision，用于表示 Protocol 的版本号。示例 8-1 是视频解码 Protocol 的结构体定义。

【示例 8-1】 EFI_FFDECODER_PROTOCOL 结构体。

```
struct _EFI_FFDECODER_PROTOCOL{
 UINT64 Revision;
 EFI_OPEN_VIDEO OpenVideo;
 EFI_CLOSE_VIDEO CloseVideo;
 EFI_QUARY_FRAME QueryFrame;
 EFI_QUARY_FRAME_SIZE QueryFrameSize;
 EFI_SET_FRAME_SIZE SetFrameSize;
};
typedef struct _EFI_FFDECODER_PROTOCOL EFI_FFDECODER_PROTOCOL;
typedef EFI_FFDECODER_PROTOCOL  EFI_FFDECODER;
```

每个 Protocol 都必须有一个 GUID，并且这个 GUID 必须是唯一的，即不能与其他 Protocol 重复。GUID 可以看做一个 16 字节的数字，按照微软公司的定义，它是 4B-2B-2B-8B 的结构体。GUID 可以通过 GUID 生成器生成，也可以手动定义。通常 GUID 的名字是 Protocol 名字加“_GUID”后缀。对视频解码 Protocol，其 GUID 名字为 EFI_FFDECODER_PROTOCOL_GUID。这个 EFI_FFDECODER_PROTOCOL_GUID 是宏，通常设计 Protocol 时还需提供一个同名的全局变量，这个全局变量的命名规范是以 g 开头，变量中单词首字符大写，变量中没有“_”。根据这个命名规范，视频解码 Protocol 的 GUID 变量为 gEfiFFDecoderProtocolGUID。示例 8-2 定义了视频解码 Protocol 的 GUID。

【示例 8-2】 视频解码 Protocol 的 GUID 宏和 GUID 变量。

```
#define EFI_FFDECODER_PROTOCOL_GUID \
{     0xce345171,0xabcd,0x11d2,{0x8e,0x4f,0x0,0xa0,0xc9,0x69,0x72,0x3b } }

// 为了兼容 EFI 1.1 而定义的 Protocol GUID 名字
#define FFDECODER_PROTOCOL  EFI_FFDECODER_PROTOCOL_GUID
extern EFI_GUID gEfiFFDecoderProtocolGUID
```

最后需要为视频解码 Protocol 里的服务定义接口，也就是声明成员函数的函数类型。Protocol 的成员函数必须遵循如下两个准则：

1）函数使用 EFIAPI 调用约定。微软编译环境下，EFIAPI 定义为 __cdecl；Linux 环境下，EFIAPI 被定义为 __attribute__((cdecl))。总而言之，EFIAPI 采用 cdecl 调用约定。

2）函数第一个参数必须是指向 Protocol 自身的 This 指针。

另外，还有建议遵循的两个准则：

1）函数返回值类型为 EFI_STATUS，用于返回错误代码。

2）在用于输入的参数前加宏 IN，用于提示 Protocol 的使用者该参数为输入参数；在用于输出的参数前加 OUT；既作为输入又作为输出的参数前加 IN OUT。IN 和 OUT 定义为空，仅用于提示调用者参数的作用，没有实际意义。

示例 8-3 列出了视频解码 Protocol 成员函数的函数声明。

【示例 8-3】 视频解码 Protocol 的成员函数声明。

```
/**
  打开视频
  @param  This            This 指针，指向视频上下文
  @param  FileName        视频文件路径
  @retval EFI_SUCCESS     成功打开视频
  @retval EFI_NOT_FOUND   找不到视频文件
**/
typedef EFI_STATUS(EFIAPI* EFI_OPEN_VIDEO)(IN EFI_FFDECODER_PROTOCOL* This,
        IN CHAR16* FileName);
```

```
/**
  关闭视频
  @param  This            This 指针，指向视频上下文
  @retval EFI_SUCCESS     视频被成功关闭
**/
typedef EFI_STATUS(EFIAPI* EFI_CLOSE_VIDEO)( IN EFI_FFDECODER_PROTOCOL* This);

/**
  从视频当前位置取出一帧，当前位置前进一帧
  @param This This 指针
  @param pFrame 返回取出的帧
  @retval EFI_SUCCESS           成功解码当前帧并返回当前帧
  @retval EFI_NO_MEDIA          没有打开有效的视频文件
  @retval EFI_END_OF_MEDIA      已经到达视频结尾
**/
typedef EFI_STATUS(EFIAPI* EFI_QUARY_FRAME)(IN EFI_FFDECODER_PROTOCOL *This,
        OUT AVFrame **pFrame);

/**
  视频帧的宽和高（以像素为单位）
  @param  This        This 指针
  @param  Width       返回一帧的宽度
  @param  Height      返回一帧的高度
  @retval EFI_SUCCESS           成功取得视频帧的宽和高
  @retval EFI_NO_MEDIA          没有打开有效的视频文件
**/
typedef EFI_STATUS(EFIAPI* EFI_QUARY_FRAME_SIZE)(
IN EFI_FFDECODER_PROTOCOL *This, OUT UINT32 *Width, OUT UINT32 *Height);

/**
  设置输出帧的宽度和高度，若此宽度和高度与帧的原始宽度和高度不一致，则将原始帧缩放后作为输出帧
  @param  This        This 指针
  @param  Width       返回一帧的宽度
  @param  Height      返回一帧的高度
  @retval EFI_SUCCESS    成功设置宽与高
  @retval EFI_NO_MEDIA   没有打开有效的视频文件
**/
typedef EFI_STATUS(EFIAPI* EFI_SET_FRAME_SIZE)(IN  EFI_FFDECODER_PROTOCOL
*This, IN UINT32  Width, IN UINT32 Height);
```

示例 8-1 ~ 示例 8-3 组合起来形成了 ffdecoder.h 文件，作为视频解码 Protocol 的接口文件。

8.2 Protocol 服务的实现

头文件 ffdecoder.h 最后要提供给用户使用。下面进入视频解码 Protocol 的实现部分。实现部分放在文件 ffdecoder.c 中。ffdecoder.c 包含两部分内容：Protocol 服务的实现和驱动框架。

本节讲述 Protocol 服务的实现，即如何实现视频解码 Protocol 的 5 个成员函数。

在 8.1 节提供的 EFI_FFDECODER_PROTOCOL 中，仅包含了 5 个成员函数和 1 个表示版本号的成员变量。视频解码 Protocol 还需要一些私有变量用于存放视频上下文，如打开的视频文件、视频当前位置等，这些私有变量存放在什么地方呢？按照 UEFI Protocol 的设计模式，在实现中需提供一个私有数据结构，Protocol 就是这个私有数据结构的一部分，这样通过 This 指针就能找到这块私有数据区。图 8-1 展示了 Protocol 与私有数据区之间的关系。

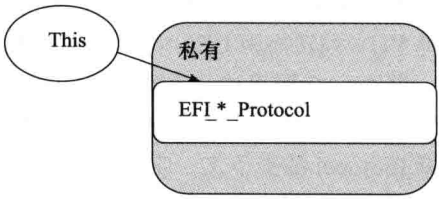


图 8-1　Protocol 私有数据与 Protocol 的关系

根据 Protocol 的这种设计模式，在视频解码 Protocol 实现中，首先要设计一个私有数据结构，用于存放 EFI_FFDECODER_PROTOCOL 的上下文。示例 8-4 展示了视频解码 Protocol 的私有数据结构。

【示例 8-4】 视频解码 Protocol 的私有数据。

```
#define FFDECODER_PRIVATE_DATA_SIGNATURE  SIGNATURE_32 ('V', 'I', 'D', 'O')
/**
  @member Signature         Protocol 上下文的签名
  @member FFDecoder         视频解码 Protocol
  @member pFormatCtx        视频格式上下文
  @member videoStream       视频流（在所有流的数组中）的编号
  @member pCodecCtx         解码器上下文
  @member pFrame            YUV 格式的帧
  @member pFrameRGBA        RGBA 格式的帧
  @member buffer            缓冲区
  @member img_convert_ctx   帧格式转换上下文
 **/
typedef struct {
  UINTN Signature;
  EFI_FFDECODER_PROTOCOL FFDecoder;
  AVFormatContext *pFormatCtx;
  int  videoStream;
  AVCodecContext *pCodecCtx;
  AVFrame *pFrame;
  AVFrame *pFrameRGBA;
  uint8_t *buffer;
  struct SwsContext *img_convert_ctx ;
} FFDECODER_PRIVATE_DATA;
static FFDECODER_PRIVATE_DATA gFFDecoderPrivate;
```

如果用户对 ffmpeg[⊖]比较熟悉，那么很快就会明白需要在私有数据中存放什么，就可以忽

⊖ ffmpeg 是一套完整的跨平台音频和视频开源解决方案，主要包括音视频编码、解码、编辑、格式转换等功能。开发者可以 ffmpeg 提供的 API 定制自己的音视频产品。

略这段内容继续往下看。在视频解码 Protocol 的各个服务中，将调用 ffmpeg 的 API 实现视频解码的功能。解码流程大体如下：首先解析视频文件的格式（需要变量 pFormatCtx），根据视频格式生成相应的解码器（需要变量 pCodecCtx）。从视频流中（需要变量 videoStream。视频文件通常包含视频流、音频流）取出一帧，此时帧是 YUV 编码的帧（需要变量 pFrame），因而还需要将 YUV 帧转换为 UEFI 可以显示的 RGBA 帧（需要变量 img_convert_ctx 和 pFrameRGBA）。

那么，在程序中是如何通过 This 指针得到私有数据的呢？ EDK2 提供了 CR 宏，根据 CR 宏我们可以设计 FFDECODER_PRIVATE_DATA_FROM_THIS(This) 用于根据 This 指针取得 Protocol 的上下文，定义如示例 8-5 所示。

【示例 8-5】 This 到私有数据区的转换。

```
#define FFDECODER_PRIVATE_DATA_FROM_THIS(a) CR(a, FFDECODER_PRIVATE_DATA,
FFDecoder, FFDECODER_PRIVATE_DATA_SIGNATURE)
```

宏展开后如代码清单 8-2 所示，从中不难看出 This 是如何转换到 FFDECODER_PRIVATE_DATA 的。

代码清单 8-2　This 到 FFDECODER_PRIVATE_DATA 的转换

```
((FFDECODER_PRIVATE_DATA *) (
  (CHAR8 *) (This) - (CHAR8 *) &((( FFDECODER_PRIVATE_DATA *) 0)-> FFDecoder) )
)
```

下面我们要实现的 5 个函数，对应 EFI_FFDECODER_PROTOCOL 的 5 个成员函数。示例 8-6 展示了这 5 个成员函数的框架。本书重点讲述 Protocol，在这 5 个函数中如何调用 ffmpeg 的 API 已超出本书的范围，在此不再详述，感兴趣的读者可以参阅本书附带的源码[⊖]。这里只讲述框架部分。这些函数需遵循如下规则：

- 函数类型必须加 EFIAPI 修饰表示该函数采用 cdecl 的调用约定。
- 函数的第一个参数必须是 This 指针。
- 进入函数后要用 FFDECODER_PRIVATE_DATA_FROM_THIS 宏根据 This 指针获得 FFDECODER_PRIVATE_DATA 指针，从而得到视频上下文。

【示例 8-6】 EFI_FFDECODER_PROTOCOL 的成员函数实现。

```
EFI_STATUS EFIAPI
OpenVideo(IN EFI_FFDECODER_PROTOCOL* This,IN CHAR16* FileName)
{
  FFDECODER_PRIVATE_DATA* Private;
  Private = FFDECODER_PRIVATE_DATA_FROM_THIS(This);
  ...
}

EFI_STATUS EFIAPI CloseVideo(IN EFI_FFDECODER_PROTOCOL* This)
```

⊖ 相关代码在 book\ffdecoder\ffdecoder.c 文件中。

```
{
  FFDECODER_PRIVATE_DATA* Private;
  Private = FFDECODER_PRIVATE_DATA_FROM_THIS(This);
  ...
}

EFI_STATUS EFIAPI QueryFrame(IN  EFI_FFDECODER_PROTOCOL  *This,
        OUT AVFrame **ppFrame)
{
  FFDECODER_PRIVATE_DATA* Private;
  Private = FFDECODER_PRIVATE_DATA_FROM_THIS(This);
  ...
}

EFI_STATUS EFIAPI QueryFrameSize(IN  EFI_FFDECODER_PROTOCOL  *This,
        OUT UINT32 *Width, OUT UINT32 *Height)
{
  FFDECODER_PRIVATE_DATA* Private;
  Private = FFDECODER_PRIVATE_DATA_FROM_THIS(This);
  ...
}

EFI_STATUS EFIAPI SetFrameSize(IN  EFI_FFDECODER_PROTOCOL  *This,
        IN UINT32 Width, IN UINT32 Height)
{
  FFDECODER_PRIVATE_DATA* Private;
  Private = FFDECODER_PRIVATE_DATA_FROM_THIS(This);
  ...
}
```

除了 Protocol 的成员函数，还需要提供产生 Protocol 实例的函数和初始化 Protocol 的函数。在 ffdecoder.c 中，声明一个 FFDECODER_PRIVATE_DATA 类型的变量 gFFDecoderPrivate，自然也就产生了一个 Protocol 的实例。初始化 Protocol 主要是设置 Protocol 的成员函数和成员变量，如示例 8-7 所示。

【示例 8-7】 初始化视频解码 Protocol 示例。

```
EFI_FFDECODER_PROTOCOL*  EFIAPI  InitFFdecoderPrivate ()
{
  EFI_STATUS Status = 0;
  FFDECODER_PRIVATE_DATA* Private = &gFFDecoderPrivate;
  Private-> Signature= FFDECODER_PRIVATE_DATA_SIGNATURE;
  Private->FFDecoder.Revision = 1;
  Private->FFDecoder.OpenVideo = OpenVideo;
  Private->FFDecoder.CloseVideo = CloseVideo;
  Private->FFDecoder.QueryFrame= QueryFrame;
  Private->FFDecoder.QueryFrameSize= QueryFrameSize;
  Private->FFDecoder.SetFrameSize= SetFrameSize;
  return & Private->FFDecoder;
}
```

8.3 服务型驱动的框架

DXE 驱动是视频解码服务的载体。实现了视频解码服务后还需要将该服务插入到 DXE 驱动中。下面就来看看如何实现 DXE 驱动。DXE 驱动的实现有以下 3 个要点：

- 在模块工程 .inf 文件中，MODULE_TYPE 设置为 DXE_DRIVER 或 UEFI_DRIVER。
- 在模块工程 .inf 文件中，[LibraryClasses] 中加入 UefiDriverEntryPoint。
- 在模块入口函数中安装驱动服务。

对于视频解码服务驱动，在模块入口函数中要安装 EFI_FFDECODER_PROTOCOL。本章开头介绍过安装 Protocol 的函数 InstallProtocolInterface，正是通过 InstallProtocolInterface 将 EFI_FFDECODER_PROTOCOL 安装到驱动的 ImageHandle 上。因为视频解码驱动用到 Shell Protocol 和 StdLib，所以安装 Protocol 之前还需初始化本模块的 Shell Protocol 和 StdLib，如示例 8-8 所示。

【示例 8-8】 视频解码驱动的模块入口函数。

```
EFI_STATUS EFIAPI
InitFFdecoder (IN EFI_HANDLE ImageHandle, IN EFI_SYSTEM_TABLE *SystemTable)
{
  EFI_STATUS Status;
  FFDECODER_PRIVATE_DATA* Private = &gFFDecoderPrivate;
  //通过初始化 ShellLib 来初始化本模块的 Shell Protocol
  ShellLibConstructorWorker2(NULL,NULL,NULL,NULL);
  //初始化 StdLib
  (void) DriverInitMain(0, NULL);
  (void) InitFFdecoderPrivate();
  //将 EFI_FFDECODER_PROTOCOL 的实例 &(Private->FFDecoder ) 安装到自身 Handle 中
  Status = gBS->InstallProtocolInterface(&ImageHandle,
            &gEfiFFDecoderProtocolGUID ,
            EFI_NATIVE_INTERFACE,
            &Private->FFDecoder
            );
}
```

下面看视频解码驱动的工程文件，示例 8-9 详细列出了该工程文件的内容。除了 MODULE_TYPE 和 UefiDriverEntryPoint 这两个关键点之外，具体到视频解码服务，还需要说一下 [Sources] 和 [LibraryClasses]。在 [Sources] 中除了包含 ffdecoder.c 文件之外，还加入了 math.c 和 InitShell.c。math.c 主要提供了 ffmpeg 用到的数学函数。InitShell.c 包含了 Shell Protocol 的初始化函数 ShellLibConstructorWorker2 和 StdLib 的初始化函数 DriverInitMain，这部分内容将在下文讲述。在 [LibraryClasses] 中，需要包含 StdLib 的相关库以及 ffmpeg 的类库 libavcodec、libavformat、libavutil、libswscale 和 zlib。关于 ffmpeg 的移植，将在 8.4 节讲述。

【示例 8-9】 视频解码驱动的工程文件。

```
 [Defines]
  INF_VERSION = 0x00010006
  BASE_NAME = ffdecoder
  FILE_GUID = 33a97c46-7491-4dfd-b442-74798713ce5f
  VERSION_STRING = 0.1
  MODULE_TYPE = UEFI_DRIVER
  ENTRY_POINT = InitFFdecoder
[Sources]
  ffdecoder.c
  math.c
  InitShell.c
[Packages]
  StdLib/StdLib.dec
  MdePkg/MdePkg.dec
  MdeModulePkg/MdeModulePkg.dec
  ShellPkg/ShellPkg.dec
  ffmpeg/ffmpeg.dec
  StdLibPrivateInternalFiles/DoNotUse.dec
[LibraryClasses]
  UefiDriverEntryPoint
  LibC
  LibStdio
  LibMath
  LibString
  BsdSocketLib
  EfiSocketLib
  UseSocketDxe
  DevShell
  zlib
  libavcodec
  libavutil
  libswscale
  libavformat
  LibUefi
  LibNetUtil
```

编译得到 ffdecoder.efi。使用命令 load ffdecoder.efi 加载之后，就可以像使用其他 Protocol 一样使用 EFI_FFDECODER_PROTOCOL 了。

8.4　ffmpeg 的移植与编译

EFI_FFDECODER_PROTOCOL 是通过调用 ffmpeg 的 API 完成视频解码的，因而在视频解码驱动的工程中需要连接 ffmpeg 库，还需要使用的库有 libavcodec、libavformat、libavutil、libswscale 及 zlib。libavcodec 提供音视频编解码接口。libavformat 用于从视频文件

中解析出（或生成）音频流、视频流、字幕流。libavutil 提供辅助函数，如字符串函数、随机数生成器等。libswscale 主要用于视频帧的缩放和颜色空间转换。下面就以 ffmpeg-0.10.2 为例来介绍如何移植 ffmpeg。ffmpeg 的移植将在 Linux 环境下完成，在 SVN 版本号为 13087 的 EDK2 开发环境下编译通过。

1）准备 ffmpeg 源码和 zlib 源码。

下载 ffmpeg-0.10.2 源码[⊖]到 EDK2 根目录，并重命名为 ffmpeg。在 ffmpeg 目录下下载 zlib 的源码[⊜]。

2）建立 ffmpeg.dec 文件。

在 ffmpeg 目录下建立 ffmpeg.dec 文件，这个文件是 ffmpeg 包的声明文件，为其他包内的模块使用 ffmpeg 包内的库时提供信息。

在 [Defines] 块内定义 PACKAGE_NAME 为 EdkffmpegPkg；为 ffmpeg 定义一个 PACKAGE_GUID；还要定义 DEC_SPECIFICATION 和包的版本号 PACKAGE_VERSION。

在 [Includes] 块内加入两个头文件目录：./ 及 ./zlib 目录。./ 表示 ffmpeg 目录。

ffmpeg.dec 文件的详细代码如示例 8-10 所示。

【示例 8-10】 ffmpeg.dec 文件。

```
 [Defines]
  DEC_SPECIFICATION  = 0x00010005
  PACKAGE_NAME  = EdkffmpegPkg
  PACKAGE_GUID = f2805c44-8985-11db-9e98-0040d0111111
  PACKAGE_VERSION = 0.1
[Includes]
  ./
  ./zlib
[LibraryClasses]
```

3）配置 ffmpeg。

执行 ffmpeg 目录下的 configure 命令以配置 ffmpeg，因为 UEFI 中没有线程的概念，所有要禁止 ffmpeg 中的多线程支持。我们也不需要播放网络视频，因而也要禁掉网络支持。命令参数如下所示：

```
~/edk2/ffmpeg:/>./configure --disable-network --disable-pthreads
    --disable-w32threads --disable-os2threads --disable-yasm
```

执行上述命令后会在 ffmpeg 目录下生成 config.h。为了让 config.h 能兼容 UEFI 系统，还要修改 config.h 文件（或者从本书附带的源码中复制该文件，位于 book\ffmpeg\ 目录下），注释掉如下 3 行：

⊖ https://ffmpeg.org/releases/。

⊜ http://www.zlib.net。

```
#define av_restrict restrict
#define ARCH_X86_32 1
#define ARCH_X86_64 0
```

因为 StdLib 没有提供 align_malloc 函数，所有还要修改 HAVE_ALIGNED_MALLOC 为 0，如下所示：

```
#define HAVE_ALIGNED_MALLOC 0
```

这样，准备工作就完成了。下面介绍 libavcodec、libavformat、libavutil、libswscale、zlib 的建立和移植以及 StdLib 的移植，主要是建立这几个库的工程文件：libavcodec.inf、libavformat.inf、libavutil.inf、libswscale.inf、zlib.inf。

移植完成后，整个 ffmpeg 包的结构如图 8-2 所示。

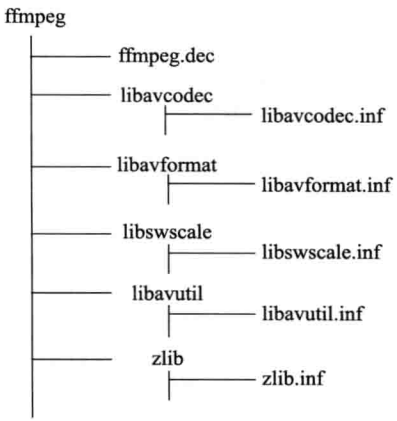


图 8-2　ffmpeg 包结构图

下面来看这几个库的建立和移植。

8.4.1　libavcodec 的建立和移植

在 ffmpeg/libavcodec 目录下生成 libavcodec.inf 文件。工程文件 libavcodec.inf 用于生成库 libavcodec，那么 libavcodec.inf 就要符合库模块工程文件的格式。

在 [Defines] 块中，将 MODULE_TYPE 设置为 BASE，将 LIBRARY_CLASS 设置为 libavcodec，将 BASE_NAME 设置为 libavcodec。

在 [Sources] 块中包含所有在 UEFI 环境中可用的源文件，源文件列表可到本书附带的源文件 book\ffmpeg\libavcodec\libavcodec.inf 查看，限于篇幅，此处不再一一列出。

在 [Packages] 块中需要包含 MdePkg/MdePkg.dec、MdeModulePkg/MdeModulePkg.dec、ffmpeg/ffmpeg.dec 及 StdLib/StdLib.dec 这 4 个包。

最重要的是 [BuildOptions] 编译选项的设置。编译 32 位 libavcodec 时需定义宏 ARCH_X86_32 为 1；编译 64 位 libavcodec 时需定义宏 ARCH_X86_64 为 1。编译选项中还要定义 _ISOC99_SOURCE 等宏，具体编译选项可参见示例 8-11。

完整的工程文件 libavcodec.inf 见示例 8-11。

【示例 8-11】 生成工程文件 libavcodec.inf。

```
[Defines]
  INF_VERSION = 0x00010005
  BASE_NAME = libavcodec
  FILE_GUID = 348C4D62-BFBD-4882-9ECE-C80BB1C63736
  VERSION_STRING = 1.0
  MODULE_TYPE = BASE
  LIBRARY_CLASS = libavcodec
```

```
[Sources]
  4xm.c
  ...
  x86/vp8dsp_init.c
[Packages]
  MdePkg/MdePkg.dec
  MdeModulePkg/MdeModulePkg.dec
  ffmpeg/ffmpeg.dec
  StdLib/StdLib.dec
 [BuildOptions]
   GCC:*_*_IA32_CC_FLAGS = -DARCH_X86_32=1 -D__UEFI__ -D_ISOC99_SOURCE
       -D_POSIX_C_SOURCE=200112 -D_FILE_OFFSET_BITS=64 -D_LARGEFILE_SOURCE
       -DHAVE_AV_CONFIG_H -std=c99 -fomit-frame-pointer -Wno-cast-qual
       -Wno-traditional-conversion -Wno-sign-conversion -Wno-pointer-sign -w
   GCC:*_*_X64_CC_FLAGS = -DARCH_X86_64=1 -D__UEFI__ -D_ISOC99_SOURCE
       -D_POSIX_C_SOURCE=200112 -D_FILE_OFFSET_BITS=64 -D_LARGEFILE_SOURCE
       -DHAVE_AV_CONFIG_H -std=c99 -fomit-frame-pointer -Wno-cast-qual
       -Wno-traditional-conversion -Wno-sign-conversion -Wno-pointer-sign -w
```

生成 libavcodec.inf 工程文件后，还要在包的 .dsc 文件中声名，然后才能被其他模块调用。在包含 ffdecoder.inf 的包的 .dsc 文件中添加如下代码：

```
[LibraryClasses]
  libavcodec|ffmpeg/libavcodec/libavcodec.inf
```

这样，库 libavcodec 就制作完成了。在 ffdecoder.inf 文件的 [LibraryClasses] 中引用 libavcodec 后，build 工具就会自动编译链接 libavcodec。

8.4.2 其他库的建立与移植

除了 libavcodec，我们还需要移植 libavformat、libavutil、libswscale 及 zlib。这几个库的源文件都已经位于 ffmpeg 目录下，各自以库名字为目录名，我们只需要在各个目录下建立相应的工程文件即可。

这些库的工程文件结构及内容与 libavcodec 的工程文件相似。[Packages] 及 [BuildOptions] 部分完全相同。[Defines] 块中 MODULE_TYPE 统一为 BASE。不同的是，[Sources] 块需要包含自己的源文件；[Defines] 块中的库名字各不相同，FILE_GUID 各不相同。在此不再详述，相信读者通过 libavcodec 的学习已经掌握了库移植的方法，下面仅列出工程文件供读者参考。

1. Libavformat 的建立与移植

在 ffmpeg/libavformat 目录下生成 libavformat.inf 文件，如示例 8-12 所示。

【示例 8-12】 库 libavformat 的工程文件 libavformat.inf。

```
[Defines]
  INF_VERSION = 0x00010005
  BASE_NAME = libavformat
  FILE_GUID = 348C4D62-BFBD-4882-9ECE-C80BBaaaa736
  VERSION_STRING = 1.0
  MODULE_TYPE = BASE
  LIBRARY_CLASS = libavformat
[Sources]
  4xm.c
  a64.c
  aacdec.c
  ...
  yuv4mpeg.c
[Packages]
  MdePkg/MdePkg.dec
  MdeModulePkg/MdeModulePkg.dec
  ffmpeg/ffmpeg.dec
  StdLib/StdLib.dec
[LibraryClasses]
  PrintLib
[BuildOptions]
GCC:*_*_IA32_CC_FLAGS = -DARCH_X86_32=1 -D__UEFI__ -D_ISOC99_SOURCE
    -D_POSIX_C_SOURCE=200112 -D_FILE_OFFSET_BITS=64 -D_LARGEFILE_SOURCE
    -DHAVE_AV_CONFIG_H -std=c99 -fomit-frame-pointer -Wno-cast-qual
    -Wno-traditional-conversion -Wno-sign-conversion -Wno-pointer-sign -w
GCC:*_*_X64_CC_FLAGS = -DARCH_X86_64=1 -D__UEFI__ -D_ISOC99_SOURCE
    -D_POSIX_C_SOURCE=200112 -D_FILE_OFFSET_BITS=64 -D_LARGEFILE_SOURCE
    -DHAVE_AV_CONFIG_H -std=c99 -fomit-frame-pointer -Wno-cast-qual
    -Wno-traditional-conversion -Wno-sign-conversion -Wno-pointer-sign -w
```

2. libavutil 的建立与移植

在 ffmpeg/libavutil 目录下生成 libavutil.inf 文件，如示例 8-13 所示。

【示例 8-13】 库 libavutil 的工程文件 libavutil.inf。

```
[Defines]
  INF_VERSION = 0x00010005
  BASE_NAME = libavutil
  FILE_GUID = 348C4D62-BFBD-4882-9ECE-C80BBbbbb736
  VERSION_STRING = 1.0
  MODULE_TYPE = BASE
  LIBRARY_CLASS = libavutil
[Sources]
  adler32.c
  ...
  xtea.c
```

```
[Packages]
  MdePkg/MdePkg.dec
  MdeModulePkg/MdeModulePkg.dec
  ffmpeg/ffmpeg.dec
  StdLib/StdLib.dec
[LibraryClasses]
  PrintLib
[BuildOptions]
GCC:*_*_IA32_CC_FLAGS = -DARCH_X86_32=1 -D__UEFI__ -D_ISOC99_SOURCE
    -D_POSIX_C_SOURCE=200112 -D_FILE_OFFSET_BITS=64 -D_LARGEFILE_SOURCE
    -DHAVE_AV_CONFIG_H -std=c99 -fomit-frame-pointer -Wno-cast-qual
    -Wno-traditional-conversion -Wno-sign-conversion -Wno-pointer-sign -w
GCC:*_*_X64_CC_FLAGS = -DARCH_X86_64=1 -D__UEFI__ -D_ISOC99_SOURCE
    -D_POSIX_C_SOURCE=200112 -D_FILE_OFFSET_BITS=64 -D_LARGEFILE_SOURCE
    -DHAVE_AV_CONFIG_H -std=c99 -fomit-frame-pointer -Wno-cast-qual
    -Wno-traditional-conversion -Wno-sign-conversion -Wno-pointer-sign -w
```

3. libswscale 的建立与移植

在 ffmpeg/libswscale 目录下生成 libswscale.inf 文件，如示例 8-14 所示。

【示例 8-14】 库 libswscale 的工程文件 libswscale.inf。

```
[Defines]
  INF_VERSION = 0x00010005
  BASE_NAME = libswscale
  FILE_GUID = 348C4D62-BFBD-4882-9ECE-C80BBbbbb736
  VERSION_STRING = 1.0
  MODULE_TYPE = BASE
  LIBRARY_CLASS = libswscale
[Sources]
  input.c
  ...
  yuv2rgb.c
[Packages]
  MdePkg/MdePkg.dec
  MdeModulePkg/MdeModulePkg.dec
  ffmpeg-trunk/ffmpeg.dec
  StdLib/StdLib.dec
[LibraryClasses]
  PrintLib
[BuildOptions]
   GCC:*_*_IA32_CC_FLAGS = -DARCH_X86_32=1 -D__UEFI__ -D_ISOC99_SOURCE
       -D_POSIX_C_SOURCE=200112 -D_FILE_OFFSET_BITS=64 -D_LARGEFILE_SOURCE
       -DHAVE_AV_CONFIG_H -std=c99 -fomit-frame-pointer -Wno-cast-qual
       -Wno-traditional-conversion -Wno-sign-conversion -Wno-pointer-sign -w
   GCC:*_*_X64_CC_FLAGS = -DARCH_X86_64=1 -D__UEFI__ -D_ISOC99_SOURCE
       -D_POSIX_C_SOURCE=200112 -D_FILE_OFFSET_BITS=64 -D_LARGEFILE_SOURCE
```

```
-DHAVE_AV_CONFIG_H -std=c99 -fomit-frame-pointer -Wno-cast-qual
-Wno-traditional-conversion -Wno-sign-conversion -Wno-pointer-sign -w
```

libswscale 用到了内联汇编，需要将源文件中的 asm 修改为 __asm__。

4. zlib 的建立与移植

在 zlib 目录下生成 zlib.inf 文件，如示例 8-15 所示。

【示例 8-15】 库 zlib 的工程文件 zlib.inf。

```
[Defines]
  INF_VERSION = 0x00010005
  BASE_NAME = zlib
  FILE_GUID = 348aaa62-BFBD-4882-9ECE-C80BBbbbb736
  VERSION_STRING = 1.0
  MODULE_TYPE = BASE
  LIBRARY_CLASS = zlib
[Sources]
  adler32.c
  ...
  gzwrite.c
[Packages.common.DXE_DRIVER]
[Packages]
  MdePkg/MdePkg.dec
  MdeModulePkg/MdeModulePkg.dec
  ffmpeg/ffmpeg.dec
  StdLib/StdLib.dec
 [BuildOptions]
   GCC:*_*_*_CC_FLAGS = -D__UEFI__ -D_ISOC99_SOURCE -D_POSIX_C_SOURCE=200112
       -D_FILE_OFFSET_BITS=64 -D_LARGEFILE_SOURCE -DHAVE_AV_CONFIG_H -std=c99
       -fomit-frame-pointer -Wno-cast-qual -Wno-traditional-conversion
       -Wno-sign-conversion -Wno-pointer-sign -w
```

下面介绍一下库的声明。

在生成工程文件后，还要在包的 .dsc 文件中声明这些工程文件，然后才能被其他模块调用。在包含 ffdecoder.inf 的包的 .dsc 文件中添加示例 8-16 所示的代码。

【示例 8-16】 在 .dsc 文件中声明库的工程文件。

```
[LibraryClasses]
  libavformat|ffmpeg/libavformat/libavformat.inf
  libavcodec|ffmpeg/libavcodec/libavcodec.inf
  libavutil|ffmpeg/libavutil/libavutil.inf
  libswscale|ffmpeg/libswscale/libswscale.inf
  zlib|ffmpeg/zlib-1.2.6/zlib.inf
```

这样，库 libavcodec、libavformat、libswscale、libavutil 及 zlib 就制作完成了。

8.4.3 在驱动型服务中使用 StdLib

在 ffmpeg 中使用了 C 标准库里的函数，如文件操作类函数 open、数学函数 log 等。因此，要想编译 EFI_FFDECODER_PROTOCOL，还要能链接 StdLib 库。

1. 改造 StdLib 兼容驱动程序

读者应该知道，C 标准库是为应用程序准备的，驱动中不能使用。例如，在 Linux 环境下，开发 C 应用程序会经常用到标准库的 printf 函数，但在驱动程序中需使用 printk 函数。UEFI StdLib 继承了 C 标准库的这一特性。而 EFI_FFDECODER_PROTOCOL 是一种 driver，如何在 driver 中使用 StdLib 呢？

首先将 StdLib.inc 中的 [LibraryClasses.Common.UEFI_APPLICATION] 修改为 [LibraryClasses.Common.UEFI_APPLICATION，LibraryClasses.common.UEFI_DRIVER]，然后将 StdLib 中的各个 inf 工程的 MODULE_TYPE 由 UEFI_APPLICATION 改为 BASE。

```
#MODULE_TYPE = UEFI_APPLICATION
MODULE_TYPE = BASE
```

这样就可以完成编译了。但现在编译出的程序还不能正常运行，为什么呢？因为 StdLib 需要初始化，在 main 类型的应用程序中，模块的入口函数调用 main 函数之前对 StdLib 进行了初始化，因而不需要开发者手工初始化 StdLib。但在视频解码驱动中，需要在模块入口函数中初始化 StdLib。

2. 初始化 StdLib

EDK2 的 StdLib 初始化相对比较简单，主要是为全局变量 gMD 分配内存并初始化 gMD。gMD 是 struct_MainData 类型的变量，该类型定义在 StdLibPrivateInternalFiles/Include/MainData.h 中。在视频解码服务中，主要用到了 StdLib 的文件操作函数和数学函数，因而初始化 gMd 中的文件相关域即可。文件相关初始化包括初始化文件描述符表 gMD->fdarray，打开标准输入、标准输出和标准错误控制台。具体初始化代码如示例 8-17 所示。

【示例 8-17】 初始化 StdLib。

```
extern FILE_HANDLE_FUNCTION_MAP FileFunctionMap;
INTN EFIAPI DriverInitMain (IN UINTN Argc,  IN CHAR16 **Argv)
{
  struct __filedes *mfd;
  INTN ExitVal;
  int i;
  ExitVal = (INTN)RETURN_SUCCESS;
  // 为 gMD 分配内存
  gMD = AllocateZeroPool(sizeof(struct __MainData));
```

```
  if( gMD == NULL ) {
    ExitVal = (INTN)RETURN_OUT_OF_RESOURCES;
  }
  else {
    /* 初始化全局状态标志 */
    extern int __sse2_available;
    __sse2_available = 0;
    _fltused = 1;
    errno = 0;
    EFIerrno = 0;
    gMD->ClocksPerSecond  = 1;
    gMD->AppStartTime = (clock_t)((UINT32)time(NULL));
    //初始化文件描述符表
    mfd = gMD->fdarray;
    for(i = 0; i < (FOPEN_MAX); ++i) {
      mfd[i].MyFD = (UINT16)i;
    }
    i = open("stdin:", O_RDONLY, 0444);
    if(i == 0) {
      i = open("stdout:", O_WRONLY, 0222);
      if(i == 1) {
        i = open("stderr:", O_WRONLY, 0222);
      }
    }
    if(i != 2) {
      Print(L"ERROR Initializing Standard IO: %a.\n    %r\n",
            strerror(errno), EFIerrno);
    }
  }
  return ExitVal;
}
```

StdLib 的文件操作是通过调用 ShellLib 中的文件接口完成的，因而还需要初始化 ShellLib。视频解码驱动用到了 ShellLib 中的文件操作 API，因而初始化 ShellLib 中的文件操作相关变量即可。ShellLib 的文件操作接口定义在变量 FileFunctionMap 中。FileFunctionMap 是 FILE_HANDLE_FUNCTION_MAP 类型的变量，其中包含了文件操作的 API，如 ReadFile、GetFileInfo 等。FILE_HANDLE_FUNCTION_MAP 定义在 ShellPkg/Library/UefiShellLib/UefiShellLib.h 中。具体的初始化 ShellLib 代码如示例 8-18 所示。

【示例 8-18】 初始化 ShellLib。

```
EFI_STATUS EFIAPI ShellLibConstructorWorker2 (
  void* aEfiShellProtocol, void* aEfiShellParametersProtocol,
  void* aEfiShellEnvironment2, void* aEfiShellInterface)
{
  if (gEfiShellProtocol != NULL) {
    FileFunctionMap.GetFileInfo = gEfiShellProtocol->GetFileInfo;
```

```
    ...
  } else {
    FileFunctionMap.GetFileInfo = (EFI_SHELL_GET_FILE_INFO)FileHandleGetInfo;
    FileFunctionMap.SetFileInfo = (EFI_SHELL_SET_FILE_INFO)FileHandleSetInfo;
    FileFunctionMap.ReadFile = (EFI_SHELL_READ_FILE)FileHandleRead;
    ...
  }
  return (EFI_SUCCESS);
}
```

回忆一下 8.3 节相关内容，在模块入口函数 InitFFdecoder 中安装视频解码 Protocol 前调用了这两个函数。

```
//初始化 ShellLib
ShellLibConstructorWorker2(NULL,NULL,NULL,NULL);
//初始化 StdLib
(void) DriverInitMain(0, NULL);
```

因为 StdLib 还在开发中，所以缺失了某些函数，补足这些缺失的函数才能成功编译 ffmpeg。这些缺失的函数可以从本书附带的源文件 book\ffmpeg\ffdecoder\InitShell.c 中获得。

8.5 使用 Protocol 服务

下面是一个简单的 fplayer.c 用于播放视频的示例。为了讲述 Protocol 的使用方法，该示例程序没有添加任何播放控制功能。EFI_FFDECODER_PROTOCOL 使用流程如下：

第 1 步：打开 EFI_FFDECODER_PROTOCOL，得到视频解码服务 FFDecoder。

第 2 步：打开视频文件。

```
FFDecoder -> OpenVideo( FFDecoder, “视频文件” );
```

第 3 步：查询视频帧的长与宽。

```
FFDecoder-> QueryFrameSize(FFDecoder, &Width, &Height);
```

第 4 步：逐一取出每一帧，直到视频结束。

```
while( !EFI_ERROR( FFDecoder-> QueryFrame(FFDecoder, &pFrame)))
{
            ...
}
```

第 5 步：关闭视频文件。

```
FFDecoder -> CloseVideo(FFDecoder );
```

打开 EFI_FFDECODER_PROTOCOL，用 gBS->LocateProtocol 服务即可，该服务根据

Protocol GUID 从系统中找出第一个符合要求的 Protocol。

输出一帧图像到屏幕利用的是 GraphicsOutput Protocol 提供的 Blt 服务。

完整的播放器代码如示例 8-19 所示。

【示例 8-19】 简单视频播放器。

```
#include <Uefi.h>
#include <Base.h>
#include <Library/DebugLib.h>
#include <Library/PrintLib.h>
#include <Protocol/GraphicsOutput.h>
#include "ffdecoder.h"
EFI_GRAPHICS_OUTPUT_PROTOCOL *GraphicsOutput;
//将一帧图像显示到屏幕
void ShowFrame(AVFrame *pFrame, int width, int height, int iFrame)
{
  GraphicsOutput -> Blt ( GraphicsOutput,
    (EFI_GRAPHICS_OUTPUT_BLT_PIXEL *)pFrame->data[0],
    EfiBltBufferToVideo, 0,0, 0,0,  width, height, 0);
}

int main(int argc, char *argv[])
{
  AVFrame *pFrame;
  EFI_GUID gEfiFFDecoderProtocolGUID = EFI_FFDECODER_PROTOCOL_GUID ;
  EFI_FFDECODER_PROTOCOL *FFDecoder;
  UINT32 Width, Height;
  CHAR16* FileName = 0;

  //找到视频解码 Protocol
  EFI_STATUS Status = gBS->LocateProtocol ( &gEfiFFDecoderProtocolGUID ,
    NULL, (VOID **)&FFDecoder );
  if (EFI_ERROR(Status)) {
    Print(L"LocateProtocol %r\n", Status);
    return Status;
  }
  //打开图像输出 Protocol
  Status = gBS->LocateProtocol(&gEfiGraphicsOutputProtocolGuid,
    NULL, (VOID **)&GraphicsOutput);

  //打开视频
  Status = gBS -> AllocatePool
( EfiLoaderData, AsciiStrLen(argv[1]) *2 + 2, (VOID**)&FileName );
  AsciiStrToUnicodeStr(argv[1], FileName);
  Status = FFDecoder -> OpenVideo( FFDecoder, FileName);
  (void) gBS->FreePool ( FileName);
  if (EFI_ERROR(Status)) {
    Print(L"Open %r\n", Status);
```

```
    return Status;
  }
  //查询视频一帧的大小 (Width、Height)
  Status = FFDecoder-> QueryFrameSize(FFDecoder, &Width, &Height);
  //读取一帧
  while( !EFI_ERROR( FFDecoder-> QueryFrame(FFDecoder, &pFrame))){
    ShowFrame(pFrame, Width, Height, 0);
  }
  //关闭视频
  Status = FFDecoder -> CloseVideo(FFDecoder );
  return 0;
}
```

在 UEFI shell 里面执行如下命令就可以播放视频了。

```
f0:/>load ffdecoder.efi
fo:/>ffplayer test.avi
```

总之，开发 UEFI 服务过程中需要掌握以下要点：

- 工程模块类型需设为 UEFI_DRIVER，这样我们的模块就可以常驻内存了。
- 在工程模块入口函数中调用 gBS->InstallProtocolInterface 将 Protocol 实例安装到指定的 Handle。这样用户可以调用 gBS->OpenProtocol 得到我们提供的 Protocol 实例，从而使用我们提供的服务。
- 如果工程中需要使用库，库的类型一般设置为 BASE。
- Protocol 成员函数的第一个参数为 This 指针，在成员函数的实现中需要用 CR 宏根据 This 指针获得上下文对象指针。

8.6 本章小结

本章介绍了如何利用 Protocol 提供服务。首先从服务提供者的角度理解 Protocol 的构成，然后介绍了 Protocol 服务接口的设计、实现及服务型驱动的框架，最后介绍了 ffmpeg 在 UEFI 中的移植，以及在驱动中使用 StdLib 的具体做法。

下一章我们将讲解 UEFI 开发最重要的内容之一，即 UEFI 驱动模型，并以 AC97 驱动为例介绍如何开发 UEFI 驱动。

第 9 章 Chapter 9

开发 UEFI 驱动

第 8 章介绍了服务型驱动，总结一下，服务型驱动有如下几个特点：

- 在 Image 的入口函数中执行安装。
- 服务型驱动不需要驱动特定硬件，可以安装到任意控制器上。
- 没有提供卸载函数。

一个设备 / 总线驱动程序在安装时首先要找到对应的硬件设备（在 UEFI 中是要找到对应的控制器），然后执行安装操作，将驱动程序安装到硬件设备的控制器上。有时，还需要卸载驱动、更新驱动（先卸载旧的驱动，然后安装新的驱动）。有时，安装操作可能需要执行多次，例如，第一次安装时发现设备没有准备好，或者所依赖的某个 Protocol 没有安装，就需要退出安装，执行其他操作，然后进行第二次安装。可以总结出一个完整的驱动程序框架需要三个部分：

- Findout()：找出对应的硬件设备。
- Install()/Start()：安装驱动到指定的硬件设备。
- Uninstall()/Stop()：从硬件设备中卸载驱动。

另外，系统中支持该驱动的设备可能不止一个，因而框架必须支持多次安装。注意在第 8 章实现的服务型驱动是不能多次安装的（仅在模块入口函数中执行安装）。如果首次安装失败，例如 InstallProtocolInterface(...) 返回错误，就只能卸载驱动文件，重新执行 loadImage 命令。

为了方便驱动的开发和管理，UEFI 提供了驱动模型。

9.1 UEFI 驱动模型

UEFI 驱动模型的核心是通过 EFI Driver Binding Protocol 管理驱动程序。一个完整的驱动程序包含两个核心部分：EFI Driver Binding Protocol 以及驱动服务本身。作为一个用户友好的驱动程序，通常它还要包含一个 EFI Component Name Protocol。

9.1.1 EFI Driver Binding Protocol 的构成

首先来看 UEFI 驱动模型是如何管理驱动的。在 UEFI 驱动的入口函数中，安装 EFI Driver Binding Protocol（EDBP）到某个 Handle（大部分情况下是自身，即 ImageHandle，有时也会安装到其他 Handle 上），这个 EBDP 实例会常驻内存，用于驱动的安装和卸载。使用 EBDP[⊖]使得我们可以多次操作（查找设备，安装卸载）驱动。代码清单 9-1 是 Driver Binding Protocol 的结构体。

代码清单 9-1　Driver Binding Protocol 结构体

```
//@file  MdePkg\Include\Protocol\DriverBinding.h
struct _EFI_DRIVER_BINDING_PROTOCOL {
  EFI_DRIVER_BINDING_SUPPORTED Supported;
  EFI_DRIVER_BINDING_START Start;
  EFI_DRIVER_BINDING_STOP Stop;
  UINT32 Version;                        // EDBP 版本号
  EFI_HANDLE ImageHandle;                // ImageHandle 是生成 EDBP 的映像文件句柄
  EFI_HANDLE DriverBindingHandle;        // DriverBindingHandle 是安装了 EDBP 的 Handle
};
```

EDBP 有 3 个成员函数和 3 个成员变量。成员变量 ImageHandle 是生成 EDBP 的映像文件句柄。DriverBindingHandle 是安装了 EDBP 的 Handle。通常这个 Handle 就是 driver 的 ImageHandle，但并非绝对如此。

EDBP 的核心是 Supported、Start 和 Stop 这 3 个成员函数，对应我们总结的 3 个部分。

- Supported 函数用于检测一个设备是否支持该驱动。
- Start 用于将驱动安装到设备上。
- Stop 用于将驱动从设备上卸载。

Version 是驱动的版本号。在所有支持同一个控制器的 EDBP 中，版本号高的具有较高的优先级，优先被安装到设备上。0x0 ~ 0x0F 和 0xFFFFFFF0 ~ 0xFFFFFFFF 保留给平台和 OEM 驱动。0x10 ~ 0xFFFFFFEF 保留给 IHV 驱动。

（1）Supported 函数

Supported 函数用于检查一个设备控制器是否支持该驱动。如果控制器支持该驱动，则

⊖ EDBP 用于测试驱动是否支持给定的控制器，并提供了启动和停止控制器的函数。

该函数返回 EFI_SUCCESS，否则返回 EFI_UNSUPPORTED、EFI_ACCESS_DENIED 或 EFI_ALREADY_STARTED 等。代码清单 9-2 列出了 Supported 的函数原型。

代码清单 9-2　EDBP 的 Supported 函数原型

```
typedef EFI_STATUS(EFIAPI *EFI_DRIVER_BINDING_PROTOCOL_SUPPORTED) (
  IN EFI_DRIVER_BINDING_PROTOCOL *This,
  IN EFI_HANDLE ControllerHandle,                    //检查这个驱动是否支持这个控制器
  //设备驱动会忽略该参数，总线驱动才会使用该参数
  IN EFI_DEVICE_PATH_PROTOCOL *RemainingDevicePath OPTIONAL
);
```

（2）Start 函数

Start 函数用来将驱动安装到设备上并启动硬件设备，在函数中最重要的事情是调用 InstallProtocolInterface() 或者 InstallMultipleProtocolInterfaces() 在 ControllerHandle 上安装驱动 Protocol。其函数原型如代码清单 9-3 所示。

代码清单 9-3　EDBP 的 Start 函数原型

```
typedef EFI_STATUS(EFIAPI *EFI_DRIVER_BINDING_PROTOCOL_START) (
  IN EFI_DRIVER_BINDING_PROTOCOL *This,
  IN EFI_HANDLE ControllerHandle,                    //驱动将被安装到这个 Handle 上
  IN EFI_DEVICE_PATH_PROTOCOL *RemainingDevicePath OPTIONAL
);
```

（3）Stop 函数

Stop 函数用于停止硬件设备并卸载驱动（调用 UninstallProtocolInterface() 或 UninstallMultipleProtocolInterfaces() 从 ControllerHandle 卸载驱动协议）。其函数原型如代码清单 9-4 所示。

代码清单 9-4　EDBP 的 Stop 函数原型

```
typedef EFI_STATUS(EFIAPI *EFI_DRIVER_BINDING_PROTOCOL_STOP) (
  IN EFI_DRIVER_BINDING_PROTOCOL *This,
  IN EFI_HANDLE ControllerHandle,                    //停止这个控制器上对应的驱动
  IN UINTN NumberOfChildren,                         //子控制器数量
  IN EFI_HANDLE *ChildHandleBuffer OPTIONAL          //子控制器数组
);
```

对设备驱动来讲，NumberOfChildren 为 0，ChildHandleBuffer 为 NULL。对 Bus Driver 来讲，如果 NumberOfChildren 不为 0，那么 ChildHandleBuffer 中的子节点都要被释放。

我们分别研究了驱动框架的三个部分，这三个部分是如何联系起来的呢？通过代码清单 9-5 可以大致了解 UEFI 驱动程序框架是如何工作的。

代码清单 9-5 CoreConnectSingleController 核心代码

```
//@file MdeModulePkg/Core/Dxe/Hand/DriverSupport.c:CoreConnectSingleController
do {
  DriverBinding = NULL;
  DriverFound = FALSE;
  for (Index = 0;
    (Index < NumberOfSortedDriverBindingProtocols) && !DriverFound;
    Index++) {
    if (SortedDriverBindingProtocols[Index] != NULL) {
      DriverBinding = SortedDriverBindingProtocols[Index];
      //调用该 DriverBinding 的 Supported 函数，检查 Driver 是否支持这个 Controller
      Status = DriverBinding->Supported(DriverBinding,
        ControllerHandle, RemainingDevicePath);
      if (!EFI_ERROR (Status)) { //Supported 返回 TRUE，该驱动支持指定的控制器
        SortedDriverBindingProtocols[Index] = NULL; //将找到的 EDBP 从列表中移除
        DriverFound = TRUE;
        //在 ControllerHandle 指定的 Controller 上启动 Driver
        Status = DriverBinding->Start(DriverBinding,
          ControllerHandle, RemainingDevicePath);
        if (!EFI_ERROR (Status)) {
          //Driver 启动成功，设置标志
          OneStarted = TRUE;
        }
      }
    }
  }
} while (DriverFound);
```

在上面的代码中，SortedDriverBindingProtocols[] 数组存放了所有的 EDBP 实例，并且数组中的 Protocol 按优先级排序，前面的 EDBP 被优先测试和安装。从前至后遍历 SortedDriverBindingProtocols[] 中的 EDBP，找到支持该控制器的驱动并安装该驱动，直到没有任何驱动支持这个设备后才退出 while 循环。

CoreConnectSingleController 用于为指定的设备控制器安装驱动，它是 gBS->ConnectController 服务的核心。开发驱动时一般不会用到这个函数，这里讲述这个函数只是为了让读者更深入地了解驱动的结构。CoreConnectSingleController 函数原型如代码清单 9-6 所示。

代码清单 9-6 CoreConnectSingleController 函数原型

```
EFI_STATUS CoreConnectSingleController (
  IN  EFI_HANDLE ControllerHandle,
  IN  EFI_HANDLE *ContextDriverImageHandles OPTIONAL,
  IN  EFI_DEVICE_PATH_PROTOCOL *RemainingDevicePath OPTIONAL
);
```

如果 ContextDriverImageHandles 为空，则遍历系统中的所有 DriverBindingProtocol，

否则就只遍历指定的 DriverBindingProtocol。SortedDriverBindingProtocols[] 存放了需要测试的 DriverBindingProtocol，对于每一个需要测试的 DriverBindingProtocol，首先调用 DriverBinding->Supported(...) 测试该 DriverBindingProtocol 是否支持 ControllerHandle，如果 Supported 函数返回 EFI_SUCCESS，则调用 DriverBinding->Start(...) 向 ControllerHandle 安装驱动，启动设备。

CoreDisconnectController 用于卸载 ControllerHandle 上指定的驱动，其函数原型如代码清单 9-7 所示。

代码清单 9-7　CoreDisconnectController 函数原型

```
EFI_STATUS CoreDisconnectController(
  IN  EFI_HANDLE ControllerHandle,
  IN  EFI_HANDLE DriverImageHandle OPTIONAL,
  IN  EFI_HANDLE ChildHandle OPTIONAL
)
```

CoreDisconnectController 对应于 gBS->DisconnectController 服务。若 DriverImageHandle 为空，则卸载 ControllerHandle 上的所有驱动。下面来看一下 CoreDisconnectController 的核心代码（见代码清单 9-8）。

代码清单 9-8　CoreDisconnectController 核心代码

```
//@file MdeModulePkg/Core/Dxe/Hand/DriverSupport.c
if (ChildHandle == NULL || ChildHandleValid) {
  ChildrenToStop = 0;
  Status = EFI_SUCCESS;
  if (ChildBufferCount > 0) {
    if (ChildHandle != NULL) {
    ChildrenToStop = 1;
    //调用 Stop 停止驱动
    Status = DriverBinding->Stop (DriverBinding, ControllerHandle,
       ChildrenToStop, &ChildHandle);
    } else {
      ChildrenToStop = ChildBufferCount;
      //调用 Stop 停止驱动
      Status = DriverBinding->Stop (DriverBinding, ControllerHandle,
         ChildrenToStop, ChildBuffer);
    }
  }
  if (!EFI_ERROR (Status)
       && ((ChildHandle == NULL) || (ChildBufferCount == ChildrenToStop))) {
    //调用 Stop 停止驱动
    Status = DriverBinding->Stop (DriverBinding, ControllerHandle, 0, NULL);
  }
  if (!EFI_ERROR (Status)) {
```

```
        StopCount++;
      }
    }
```

了解了各个部分的细节后，我们再看一遍加载驱动的整个过程：首先在 Shell 中使用命令 load 将驱动文件加载到内存，加载后 UEFI 会调用 gBS->StartImage(...) 执行 DriverImage 的入口函数，在入口函数里，Driver Binding Protocol 被加载到 Handle 上（Driver Image handle 或者其他的 Controller Handle），然后 UEFI 会遍历所有的控制器，为每个控制器调用 CoreConnectSingleController 函数，在 CoreConnectSingleController 中会调用 EDBP 的 Supported 函数测试这个驱动是否支持该控制器，如果支持，则调用 Start() 安装驱动。

9.1.2 EFI Component Name Protocol 的作用和构成

通常每个驱动都还有一个可打印的名字，便于向用户显示驱动的信息。这个可打印名字是由 EFI Component Name Protocol（ECNP）或 EFI Component Name2 Protocol（ECN2P）提供的。ECNP 和 ECN2P 不是驱动必需的 Protocol，但建议驱动开发者提供这个 Protocol。

代码清单 9-9 列出了 ECNP 的结构体及成员函数原型。它包括一个成员变量 SupportedLanguages，以及两个成员函数：GetDriverName 和 GetControllerName。

- SupportedLanguages 是此 Protocol 所支持的语言列表。这是一个由 ISO 639-2 语言代码组成的 ASCII 字符串。例如，"zho;eng" 表示 ECNP 支持中文和英文。
- GetDriverName 用于取得驱动程序的名字。
- GetControllerName 用于取得控制器或子控制器的名字。若在参数列表中指定了子控制器，则输出参数 ControllerName 返回子控制器的名字，否则返回控制器 ControllerHandle 的名字。如果对应的驱动是设备驱动，则子控制器 ChildHandle 一定为 Null。如果对应的驱动是总线驱动，则该驱动可以有子控制器。

代码清单 9-9 EFI Component Name Protocol 结构体及成员函数原型

```
struct _EFI_COMPONENT_NAME_PROTOCOL {
  EFI_COMPONENT_NAME_GET_DRIVER_NAME       GetDriverName;       //取得驱动的名字
  EFI_COMPONENT_NAME_GET_CONTROLLER_NAME   GetControllerName;   //取得控制器的名字
  CHAR8 *SupportedLanguages;                    //此 Protocol 所支持的语言列表字符串
};
//GetDriverName 根据指定的语言代码返回驱动的名字
typedef EFI_STATUS(EFIAPI *EFI_COMPONENT_NAME_GET_DRIVER_NAME)(
  IN EFI_COMPONENT_NAME_PROTOCOL *This,
  IN CHAR8 *Language,                                          //语言代码
  OUT CHAR16 **DriverName                                      //返回驱动的名字
);
//GetControllerName 根据指定的语言代码返回控制器或子控制器的名字
typedef EFI_STATUS(EFIAPI *EFI_COMPONENT_NAME_GET_CONTROLLER_NAME)(
```

```
  IN EFI_COMPONENT_NAME_PROTOCOL *This,
  IN EFI_HANDLE ControllerHandle,                    //控制器Handle
  IN EFI_HANDLE ChildHandleOPTIONAL,                 //子控制器Handle
  IN CHAR8 *Language,                                //语言代码
  OUT CHAR16 **ControllerName                        //控制器（或子控制器）名字
  );
```

代码清单9-10是EFI Component Name2 Protocol结构体，它与EFI Component Name Protocol的区别仅仅在于语言代码格式不同。ECNP使用ISO 639-2语言代码，ECNP2使用RFC 4646语言代码。例如，RFC 4646语言代码字符串"zh-Hans; zh-Hant; en"表示支持中文简体、中文繁体和英文。除了语言代码格式使用RFC 4646外，GetDriverName和GetControllerName含义及参数列表与ECNP的这两个函数完全相同，不再赘述。

代码清单9-10 EFI Component Name2 Protocol结构体

```
struct _EFI_COMPONENT_NAME2_PROTOCOL {
  EFI_COMPONENT_NAME2_GET_DRIVER_NAME GetDriverName;          //取得驱动的名字
  EFI_COMPONENT_NAME2_GET_CONTROLLER_NAME GetControllerName;  //取得控制器的名字
  CHAR8 *SupportedLanguages; //此Protocol所支持的语言列表，语言代码格式为RFC 4646
};
//GetDriverName 根据指定的语言代码返回驱动的名字
Typedef EFI_STATUS(EFIAPI *EFI_COMPONENT_NAME2_GET_DRIVER_NAME)(
  IN EFI_COMPONENT_NAME2_PROTOCOL *This,
  IN  CHAR8 *Language,                               //语言代码
  OUT CHAR16 **DriverName                            //返回驱动的名字
  );
//GetControllerName 根据指定的语言代码返回控制器或子控制器的名字
typedef EFI_STATUS(EFIAPI *EFI_COMPONENT_NAME2_GET_CONTROLLER_NAME)(
  IN EFI_COMPONENT_NAME2_PROTOCOL *This,
  IN  EFI_HANDLE ControllerHandle,                   //控制器Handle
  IN  EFI_HANDLE ChildHandle  OPTIONAL,              //子控制器Handle
  IN  CHAR8 *Language,                               //语言代码
  OUT CHAR16 **ControllerName                        //控制器（或子控制器）名字
);
```

9.2 编写设备驱动的步骤

驱动分为两部分，一部分是硬件相关的部分，这部分是驱动的内容，用于驱动硬件设备，为用户提供服务，以协议的形式出现，如DiskIo、BlockIo；另一部分是驱动的框架部分，需要实现Driver Binding Protocol，主要是其三个接口（Supported、Start和Stop），这部分用于驱动的安装与卸载。硬件相关部分不是本节讲述的重点。本节主要讲述设备驱动如何实现框架部分。Spec中详细描述了如何编写Supported、Start和Stop三个函数，下面的内容翻译自Spec 2.3。

（1）Supported 函数要点 (Spec 2.3:339)

1）忽略参数 RemainingDevicePath。

2）使用函数 OpenProtocol() 打开所有需要的 Protocol。标准的驱动要用 EFI_OPEN_PROTOCOL_BY_DRIVER 属性打开 Protocol。如果要独占某个 Protocol，首先要关闭所有使用该 Protocol 的其他驱动，然后用属性 EFI_OPEN_PROTOCOL_BY_DRIVER | EFI_OPEN_PROTOCOL_EXCLUSIVE 打开 Protocol。

3）如果 2）中 OpenProtocol() 返回错误，则调用 CloseProtocol() 关闭所有已经打开的 Protocol 并返回错误代码。

4）所需的所有 Protocol 成功打开后，测试这个 Driver 是否支持此 Controller。有时使用这些 Protocol 足以完成测试，有时还需要此 Controller 的其他特征。如果任意一项测试失败，则用 CloseProtocol() 关闭所有打开的 Protocol，返回 EFI_UNSUPPORTED。

5）测试成功，调用 CloseProtocol() 关闭所有已经打开的 Protocol。

6）返回 EFI_SUCCESS。

（2）Start 函数要点（Spec 2.3:345)

1）忽略参数 RemainingDevicePath。

2）使用函数 OpenProtocol() 打开所有需要的 Protocol。标准的驱动要用 EFI_OPEN_PROTOCOL_BY_DRIVER 属性打开 Protocol。如果要独占某个 Protocol，首先要关闭所有使用该 Protocol 的其他驱动，然后用属性 EFI_OPEN_PROTOCOL_BY_DRIVER | EFI_OPEN_PROTOCOL_EXCLUSIVE 打开 Protocol。

3）如果 2）中 OpenProtocol() 返回错误，则调用 CloseProtocol() 关闭所有已经打开的 Protocol 并返回错误代码。

4）初始化 ControllerHandle 所指定的设备。如果有错误，则关闭所有已打开的 Protocol 并返回 EFI_DEVICE_ERROR。

5）分配并初始化要用到的数据结构，这些数据结构包括驱动 Protocol 及其他相关的私有数据结构。如果分配资源时发生错误，则关闭所有已打开的 Protocol，释放已经得到的资源，返回 EFI_OUT_OF_RESOURCES。

6）用 InstallMultipleProtocolInterfaces() 安装驱动协议到 ControllerHandle。如果有错误发生，则关闭所有已打开的 Protocol，并返回错误代码。

7）返回 EFI_SUCCESS。

（3）Stop 函数要点 (Spec 2.3:353)

1）用 UninstallMultipleProtocolInterfaces() 卸载所安装的 Protocol。

2）关闭所有已打开的 Protocol。

3）释放所有已申请的资源。

本章将会以 AC97 设备驱动为例，介绍如何编写设备驱动。南桥芯片中集成的 AC97 控制器是一种 PCI 设备。在开始编写驱动之前，先补充一些 PCI 设备驱动及 AC97 声卡的背景知识。首先要建立实验环境，在 QEMU 中选择开启 Audio，类型选择为 Intel AC97。我们将在 QEMU 模拟器的 UEFI 环境下测试 AC97 驱动。

9.3　PCI 设备驱动基础

每个 PCI 设备都有三种地址空间：配置空间、IO 空间和内存空间。

系统初始化时系统会初始化每个 PCI 设备的配置空间寄存器。配置地址空间大小为 256 字节，前 64 字节是标准的，后面的寄存器由设备自定义用途。图 9-1 列出了 PCI 配置空间的结构。

偏移 / 偏移	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0x00	Vendor ID		Device ID		Command Reg.		Status Reg.		Revision ID	Class Code			Cache Line Sive	Latency Timer	Header Type	BIST
0x10	Base Address 0				Base Address 1				Base Address 2				Base Address 3			
0x20	Base Address 4				Base Address 5				Card Bus CIS pointer				Subsystem Vendor ID		Subsystem Device ID	
0x30	Expansion ROM Base Address												IRQ Line	IRQ Pin	Min_ Gnt	Max_ lat

图 9-1　PCI 设备配置空间

例如，QEMU 中 AC97 的配置空间内容如下：

```
PciRoot(0x0)/Pci(0x4,0x0)
UINT16  VendorId :8086
UINT16  DeviceId :2415
UINT16  Command  :7
UINT16  Status   :280
UINT8   RevisionID : 1
UINT8   ClassCode2 : 4
UINT8   ClassCode1 : 1
UINT8   ClassCode0 : 0
UINT8   CacheLineSize :0
UINT8   LatencyTimer : 0
UINT8   HeaderType :   0
UINT8   BIST :         0
UINT32 BaseAddress0 :  C001
UINT32 BaseAddress1 :  C401
UINT32 BaseAddress2 :  0  UINT32 BaseAddress3 :  0      UINT32 BaseAddress4 :  0
    UINT32 BaseAddress5 :  0
......
```

PCI 设备中的 IO 和内存空间被划分为 1 ~ 6 个互不重叠的子空间，每个子空间用于完成一组相对独立的子功能。Base Address 0 ~ 5 表示子空间的基地址（物理地址）。对设备的操作主要是通过对子空间的读写来实现的。UEFI 提供了 EFI_PCI_IO_PROTOCOL（简称 PciIo）来操作 PCI 设备，其结构体如代码清单 9-11 所示。PciIo 用于访问 PCI 总线或设备的内存空间、IO 空间、配置空间和 DMA 接口。PCI 总线的每一个 PCI 控制器都会安装一个 PciIo 实例，安装后，PCI 总线驱动负责为这个 PCI 设备的 ROM Image 分配内存，并将设备的 option ROM 读到分配的内存中，或者从系统平台特定的地址读取 ROM。PCI 设备的 option ROM 存放了用于配置或驱动该设备的代码。

代码清单 9-11　EFI_PCI_IO_PROTOCOL 结构体

```
struct _EFI_PCI_IO_PROTOCOL {
  EFI_PCI_IO_PROTOCOL_POLL_IO_MEM  PollMem;
  EFI_PCI_IO_PROTOCOL_POLL_IO_MEM  PollIo;
  EFI_PCI_IO_PROTOCOL_ACCESS Mem;                    //访问该 PCI 设备上的内存空间
  EFI_PCI_IO_PROTOCOL_ACCESS Io;                     //访问该 PCI 设备上的 IO 空间
  EFI_PCI_IO_PROTOCOL_CONFIG_ACCESS Pci;             //访问该 PCI 设备上的配置空间
  EFI_PCI_IO_PROTOCOL_COPY_MEM CopyMem;
  EFI_PCI_IO_PROTOCOL_MAP Map;
  EFI_PCI_IO_PROTOCOL_UNMAP Unmap;
  EFI_PCI_IO_PROTOCOL_ALLOCATE_BUFFER AllocateBuffer;
  EFI_PCI_IO_PROTOCOL_FREE_BUFFER FreeBuffer;
  EFI_PCI_IO_PROTOCOL_FLUSH Flush;
  EFI_PCI_IO_PROTOCOL_GET_LOCATION GetLocation;
  EFI_PCI_IO_PROTOCOL_ATTRIBUTES Attributes;
  EFI_PCI_IO_PROTOCOL_GET_BAR_ATTRIBUTES GetBarAttributes;
  EFI_PCI_IO_PROTOCOL_SET_BAR_ATTRIBUTES SetBarAttributes;
  UINT64 RomSize;      //ROM Image 的字节数
  VOID *RomImage;      //指向 ROM Image 的指针
};
```

在 AC97 设备驱动中只用到 EFI_PCI_IO_PROTOCOL_CONFIG_ACCESS Pci 和 EFI_PCI_IO_PROTOCOL_ACCESS Io 这两个服务，其中 Pci 服务用于读写配置空间，Io 服务用于读写 Io 空间。

先来看一下 PciIo 的 Pci 服务的用法。代码清单 9-12 列出了 Pci 服务 EFI_PCI_IO_PROTOCOL_CONFIG_ACCESS 的结构体及其成员函数的函数原型。

代码清单 9-12　Pci 服务 EFI_PCI_IO_PROTOCOL_CONFIG_ACCESS 的结构体及成员函数原型

```
//读写 PCI 配置空间寄存器
typedef EFI_STATUS(EFIAPI *EFI_PCI_IO_PROTOCOL_CONFIG)(
  IN EFI_PCI_IO_PROTOCOL *This,
  IN EFI_PCI_IO_PROTOCOL_WIDTH Width,                //每个寄存器字节数
  IN UINT32 Offset,                                  //在 PCI 配置空间上的偏移
```

```
  IN UINTN Count,                                    //寄存器个数
  IN OUT VOID *Buffer                                //读写缓冲区
);
typedef struct {
  EFI_PCI_IO_PROTOCOL_CONFIG  Read;                  //读 PCI 配置空间的寄存器
  EFI_PCI_IO_PROTOCOL_CONFIG  Write;                 //写 PCI 配置空间的寄存器
} EFI_PCI_IO_PROTOCOL_CONFIG_ACCESS;
```

Pci 服务的 Read 函数用于读取 PCI 配置空间内从偏移 Offset 处开始的 Count 个寄存器，每个寄存器的大小为 Width，读取的总字节数为 Count × Width。Write 函数用于写 PCI 配置空间内从偏移 Offset 处开始的 Count 个寄存器，每个寄存器的大小为 Width，写入的总字节数为 Count × Width。Width 必须是 EFI_PCI_IO_PROTOCOL_WIDTH 中的某一个，其枚举值如代码清单 9-13 所示。如果由 Offset、Width 和 Count 指定的地址不被控制器接受，那么 Read 函数和 Write 函数将返回 EFI_UNSUPPORTED 错误值。

代码清单 9-13　EFI_PCI_IO_PROTOCOL_WIDTH 枚举类型

```
typedef enum {
  EfiPciIoWidthUint8       = 0,
  EfiPciIoWidthUint16,
  EfiPciIoWidthUint32,
  EfiPciIoWidthUint64,
  EfiPciIoWidthFifoUint8,
  EfiPciIoWidthFifoUint16,
  EfiPciIoWidthFifoUint32,
  EfiPciIoWidthFifoUint64,
  EfiPciIoWidthFillUint8,
  EfiPciIoWidthFillUint16,
  EfiPciIoWidthFillUint32,
  EfiPciIoWidthFillUint64,
  EfiPciIoWidthMaximum
} EFI_PCI_IO_PROTOCOL_WIDTH;
```

例如，如果要读取 PCI 配置空间的 Device ID，该寄存器位于 PCI 配置空间偏移 2 字节处，宽度为 2 字节，寄存器个数为 1 个，则代码如示例 9-1 所示。

【示例 9-1】 读取 PCI 配置空间的 Device ID。

```
UINT16 DeviceId;
Status = PciIo->Pci.Read(PciIo,
                         EfiPciIoWidthUint16,          //读两个字节
                         2,                            //偏移 2 字节
                         1,                            //一个寄存器
                         &DeviceId);                   //返回读取的内容
```

再来看 PciIo 的 Io 服务的用法。Io 服务用于读写 PCI 设备的 IO 空间上的寄存器，其结

构体及成员函数原型如代码清单 9-14 所示。前面介绍 PCI 配置空间时讲过，PCI 的 IO 和内存空间分为 6 个不同的 Bar 子空间，读写 IO 空间时要通过参数 BarIndex 指明读写哪个 Bar 子空间。Io 服务的 Read 函数用于读 BarIndex 子空间上偏移 Offset 处的 Count 个寄存器，每个寄存器的宽度为 Width。Write 函数用于写 BarIndex 子空间上偏移 Offset 处的 Count 个寄存器，每个寄存器的宽度为 Width。Width 为枚举值，所有的枚举值可参考代码清单 9-13。

代码清单 9-14　EFI_PCI_IO_PROTOCOL_ACCESS 结构体及成员函数原型

```
//读、写 PCI 内存或 IO 空间
typedef EFI_STATUS(EFIAPI *EFI_PCI_IO_PROTOCOL_IO_MEM)(
  IN EFI_PCI_IO_PROTOCOL *This,
  IN EFI_PCI_IO_PROTOCOL_WIDTH Width,          //每个寄存器的大小
  IN UINT8 BarIndex,                           //操作哪个 Bar，0 ~ 5 中的一个
  IN UINT64 Offset,                            //在 Bar 上的偏移
  IN UINTN Count,                              //寄存器个数
  IN OUT VOID *Buffer                          //读、写缓冲区
);

typedef struct {
  EFI_PCI_IO_PROTOCOL_IO_MEM  Read;            //读 PCI 内存或 I/O 空间
  EFI_PCI_IO_PROTOCOL_IO_MEM  Write;           //写 PCI 内存或 I/O 空间
} EFI_PCI_IO_PROTOCOL_ACCESS;
```

例如，读第 0 个 Bar 里偏移为 4 字节、长度为 1 字节的一个寄存器，代码如示例 9-2 所示。

【示例 9-2】 PciIo 的 Io 服务示例。

```
UINT8 Data;
Status = PciIo->Io.Read(PciIo,
    EfiPciIoWidthUint8 ,                       //读一个字节
    0,                                         //Bar 0
    4,                                         //偏移 4 字节
    1,                                         //一个元素
    &Data);
```

9.4　AC97 控制器芯片的控制接口

英特尔（Intel）芯片组南桥中一般有 AC97 控制器芯片，AC97 控制器通过 AC-LINK 与 AC97 Codec 进行通信，AC97 音频驱动其实是 AC97 控制器驱动。如何操作 AC97 控制器可以参考 Intel I/O Controller Hub 6(ICH6)High Definition Audio / AC'97，这里只做简单介绍。

AC97 IO 空间有两个部分：Native Audio Bus Master 控制寄存器和 Native Mixer 寄存器。配置空间的 Bar0 表示 Native Mixer Register（NAMBAR）的基地址，访问以 16bit 为一个单

位（与 AC-LINK 单位数据大小一致），可用于配置 AC97 参数，如音量、采样率等。Bar1 表示 Bus Master Control Registers，用于控制 AC97 完成音频的输入与输出。表 9-1 列出了 NAMBAR 各寄存器的功能。

表 9-1　NAM BAR 各寄存器的功能

寄存器偏移地址	寄存器名称	作　用
00h	Reset	重置设备
02h	Master Volume	主输出音量
04h	Aux Out Volume	从输出音量
06h	Mono Volume	单声道音量
08h	Master Tone(R & L)	主音调控制（高、低音）
0Ah	PC_BEEP Volume	蜂鸣器控制
0Ch	Phone Volume	耳机音量
0Eh	Mic Volume	麦克音量
10h	Line In Volume	Line In 音量
12h	CD Volume	CD 音量
14h	Video Volume	视频音量
16h	Aux In Volume	Aux In 音量
18h	PCM Out Volume	PCM Out 音量
1Ah	Record Select	录音源选择
1Ch	Record Gain	录音增益控制
1Eh	Record Gain Mic	麦克录音增益控制
20h	General Purpose	通用设置
22h	3D Control	3D 控制
24h	AC'97 RESERVED	保留
26h	Powerdown Ctrl/Stat	节点控制 / 状态
28h	Extended Audio	支持哪些扩展寄存器
2Ah	Extended Audio Ctrl/Stat	扩展寄存器控制 / 状态
2Ch	PCM Front DAC Rate	输出 PCM 前 DAC 采样率
2Eh	PCM Surround DAC Rate	输出 PCM 环绕 DAC 采样率
30h	PCM LFE DAC Rate	输出 PCM LFE DAC 采样率
32h	PCM LR ADC Rate	输入 PCM L/R ADC 采样率
34h	MIC ADC Rate	MIC 采样率
36h	Center/LFE Volume	中间 /LFE 音量
38h	Surround Volume	环绕声音量
3Ah	S/PDIF Control	S/PDIF 控制
3C~56h	Intel RESERVED	保留
58h	AC'97 Reserved	保留
5Ah	Vendor Reserved	保留
7Ch	Vendor ID1	开发商 ID1
7Eh	Vendor ID2	开发商 ID2

表 9-2 列出了 Bus Master 各寄存器的功能。

表 9-2 Bus Master 寄存器

偏 移	助记符	全 名	默认值	读写权限
00h	PI_BDBAR	PCM In BDL（Buffer Descriptor List）表基地址	00000000h	R/W
04h	PI_CIV	PCM In BDL 表当前项 Index	00h	RO
05h	PI_LVI	PCM In BDL 表最后有效项 Index	00h	R/W
06h	PI_SR	PCM In 状态	0001h	R/WC、RO
08h	PI_PICB	PCM In 当前缓冲中的位置	0000h	RO
0Ah	PI_PIV	PCM In 预加载的 BDL 项 Index	00h	RO
0Bh	PI_CR	PCM In 控制寄存器	00h	R/W、R/W（特殊）
10h	PO_BDBAR	PCM Out BDL 表基地址	00000000h	R/W
14h	PO_CIV	PCM Out 表当前项 Index	00h	RO
15h	PO_LVI	PCM Out BDL 表最后有效项 Index	00h	R/W
16h	PO_SR	PCM Out 状态	0001h	R/WC、RO
18h	PO_PICB	PCM Out 当前缓冲中的位置	0000h	RO
1Ah	PO_PIV	PCM Out 预加载的 BDL 项 Index	00h	RO
1Bh	PO_CR	PCM Out 控制寄存器	00h	R/W、R/W（特殊）
20h	MC_BDBAR	Mic.In BDL 表基地址	00000000h	R/W
24h	MC_CIV	Mic.In 表当前项 Index	00h	RO
25h	MC_LVI	Mic.In BDL 表最后有效项 Index	00h	R/W
26h	MC_SR	Mic.In 状态	0001h	R/WC、RO
28h	MC_PICB	Mic.In 当前缓冲中的位置	0000h	RO
2Ah	MC_PIV	Mic.In 预加载的 BDL 项 Index	00h	RO
2Bh	MC_CR	Mic.In Control	00h	R/W、R/W（特殊）
2Ch	GLOB_CNT	全局控制寄存器	00000000h	R/W、R/W（特殊）
30h	GLOB_STA	全局状态寄存器		R/W、R/WC、RO
34h	CAS	Codec Access Semaphore	00h	R/W（特殊）
40h	MC2_BDBAR	Mic.2 BDL 表基地址	00000000h	R/W
44h	MC2_CIV	Mic.2 表当前项 Index	00h	RO
45h	MC2_LVI	Mic.2 BDL 表最后有效项 Index	00h	R/W
46h	MC2_SR	Mic.2 状态	0001h	RO、R/WC
48h	MC2_PICB	Mic.2 当前缓冲中的位置	0000h	RO
4Ah	MC2_PIV	Mic.2 预加载的 BDL 项 Index	00h	RO
4Bh	MC2_CR	Mic.2 Control	00h	R/W、R/W（特殊）
50h	PI2_BDBAR	PCM In 2 BDL 表基地址	00000000h	R/W
54h	PI2_CIV	PCM In 2 表当前项 Index	00h	RO
55h	PI2_LVI	PCM In 2 BDL 表最后有效项 Index	00h	R/W
56h	PI2_SR	PCM In 2 状态	0001h	R/WC、RO
58h	PI2_PICB	PCM In 2 当前缓冲中的位置	0000h	RO
5Ah	PI2_PIV	PCM In 2 预加载的 BDL 项 Index	00h	RO
5Bh	PI2_CR	PCM In 2 Control	00h	R/W、R/W（特殊）

（续）

偏　移	助记符	全　　名	默认值	读写权限
60h	SPBAR	S/PDIF BDL 表基地址	00000000h	R/W
64h	SPCIV	S/PDIF 表当前项 Index	00h	RO
65h	SPLVI	S/PDIF BDL 表最后有效项 Index	00h	R/W
66h	SPSR	S/PDIF 状态	0001h	R/WC、RO
68h	SPPICB	S/PDIF 当前缓冲中的位置	0000h	RO
6Ah	SPPIV	S/PDIF 预加载的 BDL 项 Index	00h	RO
6Bh	SPCR	S/PDIF Control	00h	R/W、R/W（特殊）
80h	SDM	SData_IN Map	00h	R/W、RO

表 9-2 中有 6 个 DMA 通道：

- PI = PCM in channel
- PO = PCM out channel
- MC = Mic in channel
- MC2 = Mic 2 channel
- PI2 = PCM in 2 channel
- SP = S/PDIF out channel

每个通道有一个 16bit DMA 引擎，用于传输音频数据（PCM 格式）。DMA 引擎使用 Buffer Descriptor List 数据结构获得数据缓冲区地址。Buffer Descriptor List 是 BufferDescriptor 类型的数组。Buffer Descriptor List 最多允许 32 项。Buffer Descriptor 结构体如代码清单 9-15 所示。

代码清单 9-15　Buffer Descriptor 结构体

```
typedef struct {
    UINT32 addr;
    UINT16 len;
    unsigned short reserved:14;
    unsigned short BUP:1;
    unsigned short IOC:1;
} BufferDescriptor;
```

IOC 是 Interrupt on Complete 的缩写，表示在 DMA 结束时是否触发中断。BUP 是 Buffer Underrun Policy 的缩写，0 表示在当前 buffer 传输结束但下一 buffer 未准备完毕时继续传送最后一个有效采样；1 表示在当前 buffer 传输结束并且当前 buffer 是最后一个有效 buffer 时，当前 buffer 被传输完毕后传输 0。

注意，List 中最后一个 BufferDescriptor 的 BUP 需为 1。每个 buffer 最多有 65535 个采样，同时采样个数必须是每帧采样数的整数倍。例如，对双声道 16bit 采样值（每个采样占两字节，每一帧左右声道各一个采样）的音频，每个 buffer 最多 65534 个采样（左声道 32767 个采样，右声道 32767 个采样）。地址 addr 的第 0 个 bit 必须是 0。

启动 DMA 的过程如下（翻译自 Intel I/O Controller Hub 6(ICH6)High Definition Audio / AC'97 程序员参考手册第 30 页）。

1）建立 Duffer Bescriptor List（BDL）。

2）将 BDL 的基地址写入 Buffer Descriptor List Base Address Register（对 PCM Out 而言，是 MBBAR+10h（POBAR））。

3）填充 BDL 中的缓冲描述符，并将音频数据填充到该描述符中的数据缓冲区内。

4）设置 Last Valid Index（LVI）寄存器（PCM OUT: MBBAR+15h（POLVI））。LVI 寄存器表示 BDL 中最后一个已经准备好缓冲区的缓冲描述符。

5）设置 Control Register 中的 run bit，启动 DMA 传输。

这时就可以听到声音了。

9.5 AC97 驱动

现在开始设计 AC97 驱动。前面讲过，驱动分为两个部分：硬件相关部分和框架部分。硬件相关部分是驱动服务本身，而框架部分主要是指 Driver Binding Protocol。

9.5.1 AC97 驱动的驱动服务 EFI_AUDIO_PROTOCOL

我们把要提供的服务命名为 EFI_AUDIO_PROTOCOL。在 EFI_AUDIO_PROTOCOL 中，我们提供播放音频的服务，首先要提供硬件初始化服务、PCM 音频播放服务、音量调节服务，还要提供一个 Event，用来通知用户音频播放结束。

1. EFI_AUDIO_PROTOCOL 头文件

在 audio.h 中，要定义 EFI_AUDIO_PROTOCOL 相关结构体，如示例 9-3 所示。EFI_AUDIO_PROTOCOL 将包含一个用于重置设备的服务 Reset、一个用于调节音量的服务 Volume、一个用于播放音频的服务 Play，以及一个成员变量 WaitForEndEvent。

【示例 9-3】 EFI_AUDIO_PROTOCOL 结构体。

```
struct _EFI_AUDIO_PROTOCOL{
 UINT64 Revision;
 EFI_AC97_RESET Reset;                    //重置设备
 EFI_AC97_PLAY Play;                      //播放声音
 EFI_AC97_VOLUME Volume;                  //调节音量
 EFI_EVENT WaitForEndEvent;               //声音播放结束时，触发该事件
}
typedef _EFI_AUDIO_PROTOCOL EFI_AUDIO_PROTOCOL;
```

设计 EFI_AUDIO_PROTOCOL 的 GUID，如示例 9-4 所示。

【示例 9-4】 EFI_AUDIO_PROTOCOL_GUID 宏。

```
#define EFI_AUDIO_PROTOCOL_GUID \
{ \
  0xce345171, 0xabcd, 0x11d2, {0x8e, 0x4f, 0x0, 0xa0, 0xc9, 0x69, 0x72, 0x3b }\
}
```

定义 Reset、Play、Volume 三个服务的函数原型，如示例 9-5 所示。

【示例 9-5】 Reset、Play、Volume 函数原型。

```
/** 重置 AC97 设备
  @retval EFI_SUCCESS          成功重置设备
  @retval EFI_DEVICE_ERROR     设备错误
**/
typedef EFI_STATUS(EFIAPI* EFI_AC97_RESET)(IN  EFI_AUDIO_PROTOCOL  *This);

/** 播放音频
  @retval EFI_SUCCESS          播放成功
  @retval EFI_DEVICE_ERROR     设备错误
**/
typedef EFI_STATUS(EFIAPI* EFI_AC97_VOLUME) (
  IN  EFI_AUDIO_PROTOCOL  *This,          // This 指针
    IN  UINT8* PcmData,                   // 要播放的 PCM 音频数据
    IN  UINT32 Format,                    // 音频格式
    IN  UINTN Size                        // PcmData 数据字节数
)

/** 调节音量
  @retval EFI_SUCCESS          成功设置音量
  @retval EFI_DEVICE_ERROR     设备错误
**/
typedef EFI_STATUS(EFIAPI* EFI_AC97_VOLUME)(
IN  EFI_AUDIO_PROTOCOL  *This,
    // 音量的改变量，正数表示提高音量，负数表示降低音量
    // 如果 *NewVolume >=0, Increase 将被忽略，新音量设为 *NewVolume;
    // 如果 *NewVolume <0,  Increase 有效
    // 作为输出参数，*NewVolume 返回改变后的音量的值
    IN  INT32 Increase,
    IN OUT INT32 * NewVolume
);
```

2. EFI_AUDIO_PROTOCOL 的私有数据

在 accdriver.c 中，还要定义用于标识音频播放上下文的数据结构，一般命名为 X_PRIVATE_DATA。在上下文中，要包含一个驱动实例，以及其他的设备相关信息。具体到 EFI_AUDIO_PROTOCOL，在上下文中我们要包含 EFI_AUDIO_PROTOCOL 实例、设备的 EFI_PCI_IO_PROTOCOL 实例和 BufferDescriptor，如下所示。

```
typedef struct {
    UINTN Signature;
    EFI_AUDIO_PROTOCOL Audio;            //EFI_AUDIO_PROTOCOL
    EFI_PCI_IO_PROTOCOL *PciIo;          //AC97 控制器上的 PciIo
    BufferDescriptor Bdes[32];           //缓冲区描述符数组
} AUDIO_PRIVATE_DATA;
```

定义 EFI_AUDIO_PROTOCOL 模板，这个模板将用于初始化 EFI_AUDIO_PROTOCOL 实例，其代码如示例 9-6 所示。

【示例 9-6】 EFI_AUDIO_PROTOCOL 模板。

```
AUDIO_PRIVATE_DATA        gDiskIoPrivateDataTemplate = {
  AUDIO_PRIVATE_DATA_SIGNATURE,
  {
    EFI_AUDIO_PROTOCOL_REVISION,         //Revision
    AC97Reset,                           //Reset
    AC97Play,                            //Play
    AC97Volume,                          //Volume
    0                                    //WaitForEndEvent
  },
  NULL,                                  //PciIo 初始化为 NULL
  {0}                                    //BufferDescriptor 初始化，各元素为 0
};
```

定义 AUDIO_PRIVATE_DATA_FROM_THIS 宏，用于根据 This 指针找到设备上下文，该宏如示例 9-7 所示。

【示例 9-7】 AUDIO_PRIVATE_DATA_FROM_THIS 宏。

```
#define AUDIO_PRIVATE_DATA_FROM_THIS(a) CR (a, AUDIO_PRIVATE_DATA,
Audio, AUDIO_PRIVATE_DATA_SIGNATURE)
```

下面要实现 EFI_AUDIO_PROTOCOL 中定义的三个服务，每个成员函数（服务）的第一个参数是 This 指针，在函数里首先要根据 This 指针获得上下文 Private，然后根据上下文执行相应操作。

3. 驱动服务的重置设备函数 Reset

Reset 主要是设置 Mixer Register 里的 Powerdown Ctrl/Stat 和 Reset 寄存器，可以参考 WinDDK 里的 AC97 驱动示例。EFI_AUDIO_PROTOCOL 中的 AC97Reset 函数，如示例 9-8 所示。

【示例 9-8】 EFI_AUDIO_PROTOCOL 的 AC97 Reset 函数。

```
EFI_STATUS EFIAPI AC97Reset(IN  EFI_AUDIO_PROTOCOL  *This)
{
  NTSTATUS InitAC97 (void);
  EFI_STATUS Status;
```

```
  //获得控制器上下文
  AUDIO_PRIVATE_DATA* Private = AUDIO_PRIVATE_DATA_FROM_THIS(This);
  gAudioPciIo = Private->PciIo;          //设置全局变量 gAudioPciIo
  Status = InitAC97 ();                  //初始化 AC97 设备
  return Status;
}
```

AC97Reset 函数的第一个参数是指向 EFI_AUDIO_PROTOCOL 的指针，EFI_AUDIO_PROTOCOL 没有包含任何有关设备的信息，通常我们要通过这个 This 指针获得设备的上下文才能操作设备。设备的上下文信息定义在 AUDIO_PRIVATE_DATA 中，通过 AUDIO_PRIVATE_DATA_FROM_THIS(This) 获得上下文指针。InitAC97 函数用于初始化设备，其实现主要是根据 AC97 规范写相应的 PCI 寄存器。

【示例 9-9】 InitAC97 函数。

```
NTSTATUS InitAC97 (void)
{
   //通过写 Bus Master 寄存器 GLOB_CNT 设置 AC97 设备状态为就绪状态
   NTSTATUS ntStatus = PrimaryCodecReady ();
   if (NT_SUCCESS (ntStatus)){
       //写 Native Mixer 寄存器 AC97REG_RESET
       WriteCodecRegister (AC97REG_RESET, 0x00, 0xFFFF);
       //通过写 Native Mixer 寄存器 AC97REG_POWERDOWN 启动 AC97 设备
       ntStatus = PowerUpCodec ();
   }
   return ntStatus;
}
NTSTATUS PrimaryCodecReady (void)
{
  ULONG WaitCycles = 200;
  DWORD dwRegValue = ReadBMControlRegister32 (GLOB_CNT);
  //写 Bus Master 控制寄存器 GLOB_CNT
  dwRegValue = (dwRegValue | GLOB_CNT_COLD) & ~(GLOB_CNT_ACLOFF | GLOB_CNT_PRIE);
  WriteBMControlRegister32 (GLOB_CNT, dwRegValue);
  do{
    //读 Bus Master 控制寄存器 GLOB_STA，GLOB_STA_PCR 位为 1 时，设备已就绪
    if (MASTER_READ_PORT_ULONG ( GLOB_STA) & GLOB_STA_PCR){
      return STATUS_SUCCESS;
    }
    KeStallExecutionProcessor(200);
  } while (WaitCycles--);
  return STATUS_IO_TIMEOUT;
}
```

WriteBMControlRegister32 用于写 Bus Master 控制寄存器，其功能是通过 PciIo->Io 写 BAR1 中的寄存器实现的，其实现如示例 9-10 所示。

【示例 9-10】 WriteBMControlRegister32 函数。

```
void WriteBMControlRegister32(IN ULONG ulOffset, IN ULONG ulValue)
{
  MASTER_WRITE_PORT_ULONG (ulOffset, ulValue);
}
EFI_STATUS MASTER_WRITE_PORT_ULONG(u32 addr, u32 data)
{
  //Bus Master 寄存器的基地址在 Bar1 中
  return WRITE_IO(1, addr, data, EfiPciIoWidthUint32);
}
EFI_STATUS WRITE_IO(UINT8 bar, u32 addr, u32 data, EFI_PCI_IO_PROTOCOL_WIDTH width)
{
  EFI_STATUS Status;
  Status = gAudioPciIo->Io.Write(
    gAudioPciIo,                                        //PciIo
    width,                                              //写入的每个寄存器大小
    bar,                                                //写入哪个 Bar
    addr,                                               //在 Bar 中的偏移
    1,                                                  //写入的寄存器个数
    &data                                               //写缓冲区
  );
  Delay(100);                                           //延时，确保指令发送到设备
  return Status;
}
```

ReadBMControlRegister32 用来读取 Bus Master 寄存器，其功能是通过 PciIo->Io 读 Bar1 中的寄存器完成的，其实现如示例 9-11 所示。

【示例 9-11】 ReadBMControlRegister32 函数。

```
ULONG ReadBMControlRegister32(IN  ULONG ulOffset)
{
  ULONG ulValue = (ULONG)-1;
  ulValue = MASTER_READ_PORT_ULONG (( ulOffset));
return ulValue;
}
u32 MASTER_READ_PORT_ULONG (u32 addr)
{
  u32 data;
  READ_IO(1, addr, (void*)&data, EfiPciIoWidthUint32);
  return data;
}
EFI_STATUS READ_IO(UINT8 bar, u32 addr, void* data, EFI_PCI_IO_PROTOCOL_WIDTH width)
{
  EFI_STATUS Status;
  Status = gAudioPciIo->Io.Read(
    gAudioPciIo,                                        //This
    width,                                              //写入的每个元素大小
    bar,                                                //写入哪个 Bar
```

```
    addr,                                                    //在 Bar 中的偏移
    1,                                                       //写入的元素个数
    data                                                     //读缓冲区
  );
  Delay(100);
  return Status;
}
```

读写 Native Mixer 寄存器是通过 PciIo->Io 读写 BAR0 中的寄存器完成的，与 Read BMControlRegister32/WriteBMControlRegister32 相似，不再赘述。

4. 驱动服务的播放函数 Play

Play 函数执行流程如下：

1）首先建立 Buffer Descriptor List（BDL，缓冲区描述符数组），将 PCM 音频数据存入缓冲区描述符数组。

2）然后将缓冲区描述符数组的地址写入设备寄存器中的 POBAR（音频输出缓冲区描述符数组基址）寄存器。向 PO_LVI 寄存器写入 BDL 最后一项的索引值，表示所有的缓冲区都已经准备完毕。

3）最后启动 DMA 传输数据并播放音频。

EFI_AUDIO_PROTOCOL 中的 AC97 Play 函数的实现如示例 9-12 所示。

【示例 9-12】 EFI_AUDIO_PROTOCOL 的 AC97 Play 函数。

```
EFI_STATUS EFIAPI AC97Play( IN  EFI_AUDIO_PROTOCOL  *This,
  IN  UINT8* PcmData,                                        //要播放的 PCM 音频数据
  IN  UINT32 Format,                                         //音频格式
  IN  UINTN Size                                             //PcmData 数据字节数
)
{
  //目前仅支持双声道 16bit 采样值
  EFI_STATUS Status;
  CONST UINT16 FRAMES_PER_BLOCK = 65534/2;
  UINT8 i=0;
  UINTN FramesLeft = Size / 4;
  //获得控制器上下文
  AUDIO_PRIVATE_DATA* Private = AUDIO_PRIVATE_DATA_FROM_THIS(This);
  gAudioPciIo = Private->PciIo;
  WriteBMControlRegisterMask(PO_CR, 0,1);
  //设置缓冲区描述符数组 Bdes
  for( i=0; i< 32; i++, FramesLeft-= FRAMES_PER_BLOCK){
     if( i * FRAMES_PER_BLOCK >= Size/4) break;
     Private->Bdes[i].addr = (u32)(PcmData + FRAMES_PER_BLOCK* 4 * i);
     Private->Bdes[i].len = (u16)( FramesLeft <= FRAMES_PER_BLOCK
         ? FramesLeft*2 : FRAMES_PER_BLOCK*2);              //采样数是帧数的两倍
```

```
    Private->Bdes[i].BUP = 0;
    Private->Bdes[i].IOC = 0;
  }
  //设置 Bdes 数组结束标志
  Private->Bdes[i-1].BUP = 1;
  Private->Bdes[i-1].IOC = 0;
  //写 AC97 PCI 设备寄存器
  WriteBMControlRegister32(PO_BDBAR, (u32)Private->Bdes);  //写 Bdes 基址
  WriteBMControlRegister(PO_LVI, i-1);                     //Bdes 中的所有缓冲区都可用
  WriteBMControlRegisterMask(PO_CR, 1,1);                  //启动 DMA
  (void) Status;
  return EFI_SUCCESS;
}
```

5. 驱动服务的调节音量函数 Volume

下面介绍一下设置音量的函数 Volume，其功能是通过写 Native Mixer 寄存器 AC97REG_PCM_OUT_VOLUME 实现的。EFI_AUDIO-PROTOCOL 中的 AC97Volume 函数代码如示例 9-13 所示。

【示例 9-13】 EFI_AUDIO_PROTOCOL 的 AC97 Volume 函数。

```
EFI_STATUS EFIAPI AC97Volume(IN  EFI_AUDIO_PROTOCOL  *This,
    IN  INT32 Increase,  IN OUT INT32 * NewVolume)
{
  AUDIO_PRIVATE_DATA* Private = AUDIO_PRIVATE_DATA_FROM_THIS(This);
  gAudioPciIo = Private->PciIo;
  if(*NewVolume >= 0){
    WORD data= (WORD) (long ) NewVolume;
    WriteCodecRegister (AC97REG_PCM_OUT_VOLUME, data, 0xFFFF);
  }else{
    WORD data= 0;
    ReadCodecRegister(AC97REG_PCM_OUT_VOLUME, &data);
    data += (INT16) Increase;
    WriteCodecRegister (AC97REG_PCM_OUT_VOLUME, data, 0xFFFF);
    *NewVolume = (INT32)data;
  }
  return EFI_SUCCESS;
}
```

6. WaitForEndEvent 事件

WaitForEndEvent 事件将会在音频驱动被启动时创建（具体创建过程将在 9.5.2 节实现 Start 服务中讲述），它被定义为 EVT_NOTIFY_WAIT 类型的事件。当用户调用 WaitForEvent 等待这个事件时，该事件的 notification 函数 PlayEndEventNoify 会不断被调用，这个 notifciation 函数的作用是检查 AC97 的状态寄存器 PO_SR，检查到结束标志时触发事件 WaitForEndEvent。PlayEndEventNoify 函数源码如下所示：

```
VOID PlayEndEventNoify(IN EFI_EVENT Event, IN VOID *Context)
{
  AUDIO_PRIVATE_DATA    *Private = (AUDIO_PRIVATE_DATA   * )Context;
  ULONG Status;
  gAudioPciIo = Private->PciIo;
  Status = ReadBMControlRegister32(PO_SR);        //读状态寄存器
  if(Status & (1<<3)){                            //检测结束标志
      gBS->SignalEvent(Event);                    //触发事件 WaitForEndEvent
      WriteBMControlRegisterMask(PO_SR              , 0,1<<3);
  }
}
```

9.5.2　AC97 驱动的框架部分

驱动的框架部分主要是完成三个功能：

1）实现 EFI_DRIVER_BINDING_PROTOCOL。

2）实现 Component Name Protocol 和 Component Name2 Protocol。

3）在模块的初始化函数中安装 EDBP、ECNP 和 ECN2P。

1. 实现 EFI_DRIVER_BINDING_PROTOCOL

实现 EDBP 最关键是实现 EDBP 的三个服务：Supported、Start 和 Stop。在 AC97 驱动的 EDBP 中，我们将实现 AC97DriverBindingSupported 函数作为 EDBP 的 Supported 服务；实现 AC97DriverBindingStart 函数作为 EDBP 的 Start 服务；实现 AC97DriverBindingStop 函数作为 EDBP 的 Stop 服务。

首先要产生一个 EDBP 实例并初始化，如示例 9-14 所示。

【示例 9-14】 AC97 驱动的 EDBP 实例。

```
//AC97 驱动的 Driver Binding Protocol
EFI_DRIVER_BINDING_PROTOCOL gAudioDriverBinding = {
  AC97DriverBindingSupported,          //Supported 服务
  AC97DriverBindingStart,              //Start 服务
  AC97DriverBindingStop,               //Stop 服务
  0xa,                                 //Version
  NULL,                                //ImageHandle
  NULL                                 //DriverBindingHandle
};
```

下面分别讲述三个核心服务（Supported、Start 和 Stop）是如何实现的。

（1）实现 Supported 服务

函数 AC97DriverBindingSupported 实现了 Supported 服务，当待检测的设备是 AC97 控制器时，返回 EFI_SUCCESS，否则返回错误代码。其处理流程为：

1）判断 Controller 是否有 EFI_PCI_IO_PROTOCOL，若没有，则返回错误。

2）读取 PCI 配置空间，根据配置空间的 ClassCode 判断设备是否是 AC97 驱动器。

AC97DriverBindingSupported 函数的具体代码如示例 9-15 所示。

【示例 9-15】 实现 Supported 服务的 AC97DriverBindingSupported 函数。

```
/** 测试AC97 驱动是否支持该控制器
  @retval EFI_SUCCESS              AC97 驱动支持 ControllerHandle
  @retval EFI_ALREADY_STARTED      AC97 驱动已经在 ControllerHandle 启动
  @retval other                    AC97 驱动不支持 ControllerHandle
**/
EFI_STATUS EFIAPI AC97DriverBindingSupported (
  IN EFI_DRIVER_BINDING_PROTOCOL  *This,          //AC97 Driver Binding Protocol
  IN EFI_HANDLE ControllerHandle,                 //待测试的控制器
  IN EFI_DEVICE_PATH_PROTOCOL *RemainingDevicePath OPTIONAL
  )
{
  EFI_STATUS Status;
  PCI_TYPE00 PciData;
  EFI_PCI_IO_PROTOCOL *PciIo;
  //打开 ControllerHandle 控制器上的 PciIo
  Status = gBS->OpenProtocol(ControllerHandle,
          &gEfiPciIoProtocolGuid,
          (VOID**)&PciIo,
          This->DriverBindingHandle,
          ControllerHandle,
          EFI_OPEN_PROTOCOL_BY_DRIVER);
  if (EFI_ERROR (Status)) {
    return Status;     //ControllerHandle 不存在 PciIo 或打开 PciIo 失败，返回错误
  }
   //读取整个 PCI 配置空间内容
  Status = PciIo->Pci.Read (PciIo,  EfiPciIoWidthUint32,
           0,  sizeof (PciData) / sizeof (UINT32),  &PciData);
  gBS->CloseProtocol (ControllerHandle,  &gEfiPciIoProtocolGuid,
        This->DriverBindingHandle,  ControllerHandle);
  if (EFI_ERROR (Status)) {
    return Status;
  }
  //判断该 PCI 设备是不是音频设备
  if (!(PciData.Hdr.ClassCode[2] == PCI_CLASS_MEDIA
       && PciData.Hdr.ClassCode[1] == PCI_CLASS_MEDIA_AUDIO
       && PciData.Hdr.ClassCode[0] == 0x00) ) {
    return EFI_UNSUPPORTED;
  }
  return EFI_SUCCESS;
}
```

（2）实现 Start 服务

函数 AC97DriverBindingStart 实现了 Start 服务，其功能是启动 AC97 设备并安装 EFI_

AUDIO_PROTOCOL。启动过程包括三部分：

1）打开 EFI_PCI_IO_PROTOCOL。

2）为 AC97 控制器上下文分配内存。

3）在 ControllerHandle 上安装 AC97 驱动 Protocol，即 EFI_AUDIO_PROTOCOL。

AC97DriverBindingStart 函数的具体代码如示例 9-16 所示。

【示例 9-16】 实现 Start 服务的 AC97DriverBindingStart 函数。

```
/** 在控制器 ControllerHandle 上启动 AC97 驱动
  @param  RemainingDevicePath  该参数对 AC97 Driver Binding Protocol 无效
  @retval EFI_SUCCESS          成功将 AC97 驱动加载到 ControllerHandle 上
  @retval EFI_ALREADY_STARTED  驱动已经在该控制器上启动
  @retval other                驱动不支持该控制器
**/
EFI_STATUS EFIAPI AC97DriverBindingStart (
  IN EFI_DRIVER_BINDING_PROTOCOL *This,          //AC97 Driver Binding Protocol
  IN EFI_HANDLE  ControllerHandle,               //AC97 控制器
  IN EFI_DEVICE_PATH_PROTOCOL *RemainingDevicePath OPTIONAL
  )
{
  EFI_STATUS Status;
  EFI_PCI_IO_PROTOCOL *PciIo;
  AUDIO_PRIVATE_DATA *Private;
  //打开 PCI IO Protocol
  Status = gBS->OpenProtocol(ControllerHandle, &gEfiPciIoProtocolGuid,
    (VOID**)&PciIo,  This->DriverBindingHandle, ControllerHandle,
    EFI_OPEN_PROTOCOL_BY_DRIVER);
  if (EFI_ERROR (Status)) {
    return Status;
  }
  //为 AC97 控制器上下文分配内存，并初始化
  Private = AllocateCopyPool (sizeof (AUDIO_PRIVATE_DATA),
       &gDiskIoPrivateDataTemplate );
  if (Private == NULL) {
    goto ErrorExit;
  }
  Private->PciIo = PciIo;
  Status = gBS->CreateEvent(EVT_NOTIFY_WAIT, TPL_NOTIFY,
      (EFI_EVENT_NOTIFY)PlayEndEventNoify, (VOID*)Private,
      &Private->Audio.WaitForEndEvent);
  //在 ControllerHandle 上安装 AC97 驱动
  Status = gBS->InstallProtocolInterface (&ControllerHandle,
    &gEfiAudioProtocolGUID, EFI_NATIVE_INTERFACE, &Private->Audio);
ErrorExit:
  if (EFI_ERROR (Status)) {
     //启动驱动失败
    if (Private != NULL) {
```

```
      FreePool (Private);
    }
    gBS->CloseProtocol (ControllerHandle,
          &gEfiPciIoProtocolGuid,
          This->DriverBindingHandle,
          ControllerHandle);
  }else{
    //初始化 Ac97 设备
    AC97Reset(&Private->Audio);
    //AC97 设备就绪
  }
  return Status;
}
```

（3）实现 Stop 服务

AC97DriverBindingStop 函数实现了 Stop 服务，同 Start 函数相对应，其主要过程包括三部分：

1）关闭设备控制器上的 PCI IO Protocol。

2）卸载 ControllerHandle 上的 Audio IO Protocol。

3）释放资源。

AC97DriverBindingStop 函数的具体实现如示例 9-17 所示。

【示例 9-17】 实现 Stop 服务的 AC97DriverBindingStop 函数。

```
/** 停止 ControllerHandle 上的 AC97 驱动
  @retval EFI_SUCCESS     驱动从 ControllerHandle 上停止并卸载
  @retval other           驱动未能从 ControllerHandle 上卸载
**/
EFI_STATUS EFIAPI AC97DriverBindingStop (
  IN  EFI_DRIVER_BINDING_PROTOCOL *This,
  IN  EFI_HANDLE ControllerHandle,       //AC97 控制器
  IN  UINTN NumberOfChildren,            //0（设备控制器的子控制器数目为 0）
  IN  EFI_HANDLE *ChildHandleBuffer      //子控制器 Handle 数组
  )
{
  EFI_STATUS              Status;
  AUDIO_PRIVATE_DATA      *Private;
  EFI_AUDIO_PROTOCOL      *Audio;
  //获得 AC97 控制器上下文
  Status = gBS->OpenProtocol (ControllerHandle, &gEfiAudioProtocolGUID,
    (VOID **) &Audio, This->DriverBindingHandle,
    ControllerHandle, EFI_OPEN_PROTOCOL_GET_PROTOCOL);
  if (EFI_ERROR (Status)) {
    return EFI_UNSUPPORTED;
  }
  Private = AUDIO_PRIVATE_DATA_FROM_THIS(This);
```

```
  //卸载该控制器上的AC97驱动
  Status = gBS->UninstallProtocolInterface (ControllerHandle,
    &gEfiAudioProtocolGUID, &Private->Audio);
  if (!EFI_ERROR (Status)) {
    Status = gBS->CloseProtocol (ControllerHandle,
      &gEfiPciIoProtocolGuid,  This->DriverBindingHandle, ControllerHandle);
  }
  if (!EFI_ERROR (Status)) {
    gBS->CloseEvent(Private->Audio.WaitForEndEvent);
    FreePool (Private);
  }
  return Status;
}
```

2. 实现ECNP和ECN2P

作为一个完整的、对用户友好的驱动程序，我们还要提供EFI Component Name Protocol（简称ECNP）和EFI Component Name2 Protocol（简称ECN2P）。ECNP和ECN2P的实现如示例9-18所示。

【示例9-18】 ECNP和ECN2P的实现。

```
//@file uefi\book\audio\ComponentName.c
//初始化EFI Component Name Protocol实例
GLOBAL_REMOVE_IF_UNREFERENCED EFI_COMPONENT_NAME_PROTOCOL  gAudioComponentName = {
  AudioComponentNameGetDriverName,              //对应GetDriverName服务
  AudioComponentNameGetControllerName,          //对应GetControllerName服务
  "eng"                                         //对应SupportedLanguages成员变量
};
//初始化EFI Component Name2 Protocol 实例
GLOBAL_REMOVE_IF_UNREFERENCED EFI_COMPONENT_NAME2_PROTOCOL gAudioComponentName2 = {
  (EFI_COMPONENT_NAME2_GET_DRIVER_NAME) AudioComponentNameGetDriverName,
  (EFI_COMPONENT_NAME2_GET_CONTROLLER_NAME) AudioComponentNameGetControllerName,
  "en"
};
//字符串列表，该表被Component Name 和Component Name2共同使用
GLOBAL_REMOVE_IF_UNREFERENCED EFI_UNICODE_STRING_TABLE mAudioDriverNameTable[]
    = {  {"eng;en",  (CHAR16 *)L"Generic AC 97 Driver" },  {NULL,  NULL } };
EFI_STATUS EFIAPI AudioComponentNameGetDriverName (
  IN  EFI_COMPONENT_NAME_PROTOCOL  *This,
  IN  CHAR8 *Language,
  OUT CHAR16 **DriverName )
{
  //从字符串表mAudioDriverNameTable中找出指定语言Language的字符串
  return LookupUnicodeString2 (
           Language,
           This->SupportedLanguages,
           mAudioDriverNameTable,
```

```
            DriverName,
            (BOOLEAN)(This == &gAudioComponentName)
            );
}
EFI_STATUS EFIAPI AudioComponentNameGetControllerName (
  IN EFI_COMPONENT_NAME_PROTOCOL *This,
  IN EFI_HANDLE ControllerHandle,
  IN EFI_HANDLE ChildHandle OPTIONAL,
  IN CHAR8 *Language,
  OUT CHAR16 **ControllerName)
{
  return EFI_UNSUPPORTED;
}
```

在上述示例中，gAudioComponentName2 的 GetDriverName 函数共享 gAudioComponentName 的 GetDriverName 函数。gAudioComponentName2 的 GetControllerName 函数共享 gAudioComponentName 的 GetControllerName 函数。

在 AC97 的 ComponentName 中，我们用到了 EFI_UNICODE_STRING_TABLE mAudioDriverNameTable[] 字符串表，该表以空项结尾，每一项表示一种语言的字符串。LookupUnicodeString2 函数用于从字符串表中找出指定语言的字符串，其函数原型如代码清单 9-16 所示。

代码清单 9-16　EFI_UNICODE_STRING_TABLE 结构体及 LookupUnicodeString2 函数原型

```
typedef struct {
  CHAR8    *Language;                    //语言代码
  CHAR16   *UnicodeString;               //该语言下的字符串
} EFI_UNICODE_STRING_TABLE;
//从字符串表中找出对应于 Language 语言的字符串
EFI_STATUS EFIAPI LookupUnicodeString2 (
  IN CONST CHAR8 *Language,              //语言代码，函数将返回该语言对应的字符串
  IN CONST CHAR8 *SupportedLanguages,    //UnicodeStringTable 支持的语言列表
  IN CONST EFI_UNICODE_STRING_TABLE *UnicodeStringTable,      //字符串表
  OUT CHAR16 **UnicodeString,                                 //返回的字符串
 //判断语言代码是否为 ISO639 格式：TURE，为 ISO639-2 格式；FALSE，为 RFC 4646 格式
 IN BOOLEAN Iso639Language
);
```

3. 安装 EDBP、ECNP 和 ECN2P

最后要在模块的入口函数内安装 EDBP、ECNP 和 ECN2P 这三个 Protocol，如示例 9-19 所示。

【示例 9-19】AC97 驱动模块入口函数。

```
EFI_STATUS EFIAPI InitializeACC(IN EFI_HANDLE ImageHandle,
        IN EFI_SYSTEM_TABLE  *SystemTable)
```

```
{
  EFI_STATUS Status;
  //安装 EDBP、ECNP 及 ECN2P
  Status = EfiLibInstallDriverBindingComponentName2 (ImageHandle,
    SystemTable,
    &gAudioDriverBinding,
    ImageHandle,
    &gAudioComponentName,
    &gAudioComponentName2);
  return Status;
}
```

EfiLibInstallDriverBindingComponentName2 是 UefiLib 提供的用于安装 Driver Binding Protocol、Component Name Protocol 及 Component Name2 Protocol 的函数，如代码清单 9-17 所示。

代码清单 9-17　EfiLibInstallDriverBindingComponentName2 函数

```
EFI_STATUS EFIAPI EfiLibInstallDriverBindingComponentName2 (
  IN CONST EFI_HANDLE ImageHandle,                          //驱动的 ImageHandle
  IN CONST EFI_SYSTEM_TABLE *SystemTable,                   //EFI 系统表
  IN EFI_DRIVER_BINDING_PROTOCOL *DriverBinding,            //Driver Binding Protocol
  IN EFI_HANDLE DriverBindingHandle,                        //Protocol 安装到此 Handle
  IN CONST EFI_COMPONENT_NAME_PROTOCOL *ComponentName, OPTIONAL
  IN CONST EFI_COMPONENT_NAME2_PROTOCOL *ComponentName2 OPTIONAL
  )
```

4. AC97 驱动的工程文件

另外，还要提供一个工程文件以便编译整个 AC97 驱动工程。在该工程文件中，需要注意以下两点。

1）MODULE_TYPE 设置为 UEFI_DRIVER。

2）ENTRY_POINT 设置为 InitializeACC。

AC97 驱动的工程文件如示例 9-20 所示。

【示例 9-20】 AC97 驱动的工程文件。

```
[Defines]
  INF_VERSION    = 0x00010005
  BASE_NAME    = ac97
  FILE_GUID    = 6987936E-ED34-44db-AE97-1FA5E4ED2117
  MODULE_TYPE   = UEFI_DRIVER
  VERSION_STRING   = 1.0
  ENTRY_POINT   = InitializeACC
 [Sources]
  accdriver.c
  ac97.c
  ComponentName.c
```

```
[Packages]
  MdePkg/MdePkg.dec
  MdeModulePkg/MdeModulePkg.dec
[LibraryClasses]
  UefiDriverEntryPoint
  UefiLib
  UefiBootServicesTableLib
[Protocols]
  gEfiDriverBindingProtocolGuid
  gEfiComponentNameProtocolGuid
  gEfiPciIoProtocolGuid
[BuildOptions]
 MSFT:DEBUG_*_*_CC_FLAGS   = /wd4201  /wd4305
 MSFT:RELEASE_*_*_CC_FLAGS = /wd4201
```

9.5.3 AC97 驱动实验

现在可以尝试在 UEFI 环境下播放音频了。

下面介绍如何建立实验环境和播放音频。

1）首先，按照第 2 章的相关讲述在 QEMU 中建立 UEFI 虚拟机。然后，在“Hardware”标签页中设置“Enable Sound Card”为“Yes”，设置“Sound Card”为“Intel AC97 Audio”，如图 9-2 所示。

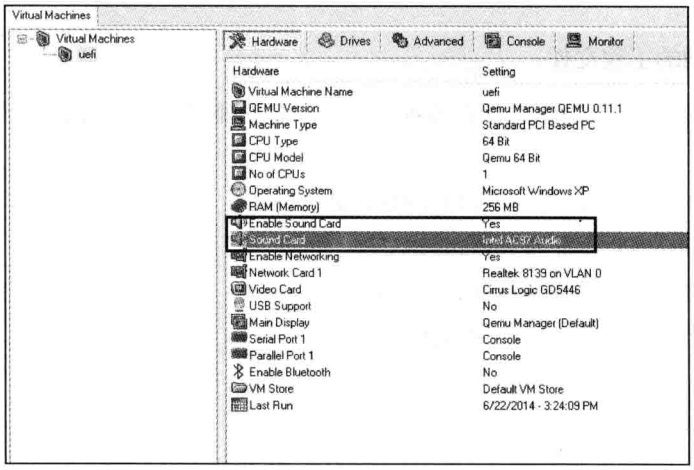


图 9-2　在 QEMU 的 UEFI 虚拟机中设置 AC97 声卡

2）将 AC97 驱动文件 ac97.efi 和音频播放文件 testac97.efi 复制到 U 盘的 efi\boot\x64 目录中。

3）双击“Drives”标签页中的“Hard Disk 1”，然后单击“Use Physical Disk”按钮选择表示 U 盘的 \\.\PhysicalDrive1（地址 PhysicalDrive1 仅为示例，用户的机器配置不同，U 盘地址也会不同），如图 9-3 所示。

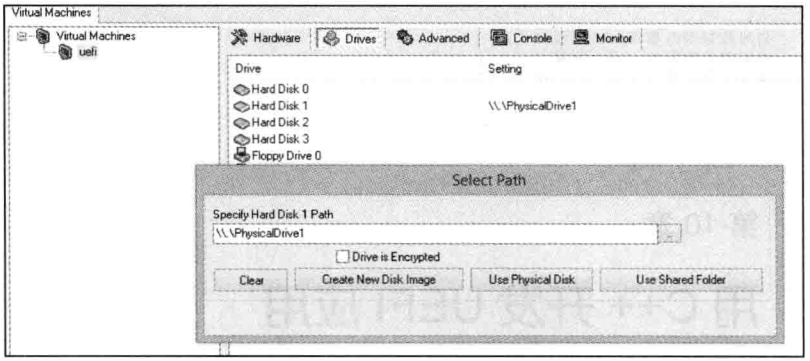


图 9-3　在虚拟机上使用 U 盘

4）启动虚拟机进入 UEFI Shell，并进入 U 盘的 efi\boot\x64 目录。

```
Shell> fs0:
fs0:> cd efi\boot\x64
```

5）执行如下命令加载 AC97 驱动。

```
fs0:> load ac97.efi
```

6）执行如下命令播放音频。

```
fs0:> testac.efi
fs0:> testac.efi piano 2.wav
```

“testac.efi”命令将会播放一段默认音频。“testac.efi piano2.wav”命令将会播放当前目录下的音频文件 pinao2.wav。

9.6　本章小结

本章讲述了 UEFI 驱动模型，驱动模型的核心是通过 Driver Binding Protocol 管理驱动。本章还详细讲述了 Driver Binding Protocol 的 Supported、Start、Stop 函数的作用及用法。本章还以 AC97 驱动为例讲述了 PCI 设备驱动的开发过程，主要讲述了如何实现驱动所涉及的 4 个 Protocol。

1）驱动的主体 Protocol，如本章介绍的 EFI_AUDIO_PROTOCOL。

2）驱动框架部分的 Driver Binding Protocol。

3）驱动框架部分的 Component Name Protocol 及 Component Name2 Protocol。

前面几章介绍了 UEFI 基础知识以及 UEFI 应用程序、驱动程序的开发，接下来的章节将利用这些基础知识开发更加复杂的应用。下一章将讲述如何利用 C++ 开发 UEFI 应用程序。

Chapter 10 第 10 章

用 C++ 开发 UEFI 应用

EDK2 采用 C 语言开发，UEFI 标准也是采用 C 语法编制，采用 C 语言开发 UEFI 应用和驱动是顺理成章的选择。但是，有些情况下不得不选择 C++，例如开发大型 GUI 应用，或者历史遗留代码采用 C++ 开发等情况下。本章就来讲述如何采用 C++ 开发 UEFI 应用。本章的源码在 uefi\book\CppPkg 及 uefi\book\GcppPkg 目录下。

10.1 从编译器角度看 C 与 C++ 的差异

从源码到可执行文件，分为两个过程：编译和连接。编译 C 源文件生成的目标文件和编译 C++ 源文件生成的目标文件格式是完全相同的，然后由相同的连接器连接生成可执行代码。那么我们就可以将重点放在编译阶段。首先来看 C 语法和 C++ 语法几个主要的不同点。

1. 名字修饰约定不同

C/C++ 程序编译后，函数名、数组名、结构体名等会根据一定的规则在原名字的基础上生成新的名字，这个过程称为名字修饰（name mangling 或 name decoration），这些规则称为名字修饰约定。原名字由程序开发者使用，经过修饰后的名字供编译器和连接器使用。C 与 C++ 的名字修饰约定有根本上的不同。C 语言修饰后的名字仅与原名字和调用约定相关。C++ 修饰后的名字除了与原名字和调用约定相关外，还与参数列表及参数类型相关。尽管不同编译器的名字修饰约定各不相同，但是这两个规则却大体相似。例如，对于函数名 int foo(int*)，VC 编译后的修饰名为 _foo，VC++ 编译后的修饰名为 ?foo@@YAHPAH@Z，gcc

编译后的修饰名为 foo，g++ 编译后的修饰名为 _Z3fooPi。因为名字修饰约定的不同，在 C++ 中调用 C 函数就不能那么直观了。代码清单 10-1 是一个 C++ 源文件，在该代码中试图调用 C 标准库中的 puts 函数，当用“g++ -o main main.cpp”命令编译该程序时，会报告如下错误：

```
main.cpp:(.text+0x11): undefined reference to `puts(char const*)'
```

原因就在于 C 标准库中的 puts 的修饰名为 puts，而 main.cpp 编译后 puts 的修饰名为 _Z4putsPKc，连接器是无法在 C 标准库中找到名为 _Z4putsPKc 的函数的。

代码清单 10-1　C++ 程序中调用 puts 函数

```
//main.cpp
extern int puts (__const char *__s);
int main()
{
  puts("call puts from C++");
}
```

要想在 C++ 代码中调用 C 函数或数组，需要在函数和数组声明前加上“extern "C"”。将代码清单 10-1 中的 puts 声明改为“extern "C" int puts(__const char *__s);”就可以了。为了让 C++ 程序能使用 C 标准库，在 stdio.h 等头文件中有代码清单 10-2 所示的代码。C++ 源程序中包含在“extern "C"{}”内的代码会使用 C 语言名字修饰约定。

代码清单 10-2　C++ 中声明 C 语言函数的方式

```
#ifdef __cplusplus
extern "C"{
#endif
```

因为 EDK2 在设计之初并没有考虑对 C++ 的支持，所以早期的 EDK2 的所有头文件中都没有加入以上代码。但随着 UEFI 的不断完善，新版本的 EDK2 头文件中已经加入了上述代码。

2. 对 NULL 的定义不同

EDK2 中 NULL 定义为 (void*)0，这是 C 风格的 NULL 定义。当在 C++ 中调用代码清单 10-3 中的语句时，会产生类型转换警告 C2440：从“void*”转换为非 void 类型时必须进行显式类型转换。

代码清单 10-3　NULL 隐式类型转换

```
CHAR16* String = NULL;
```

注意，在 C++ 代码中 NULL 定义为 0。

3. 对布尔类型的处理不同

C 中基本类型不包含布尔类型，而 C++ 中基本类型包含布尔类型。EDK2 中定义了 BOOLEAN 表示布尔类型，其类型定义如下所示：

```
typedef unsigned char BOOLEAN;
```

在 C 语言中，对于代码清单 10-4 中的语句，(a == b) 将会生成一个 int 类型的临时值，然后两个 int 临时值相除，最后转换为 BOOLEAN 值并赋值给 Equal。而在 C++ 语言中，(a == b) 将会生成一个 Boolean 类型的临时值，两个 Boolean 类型的值相除时会产生 C4804 警告。

代码清单 10-4　BOOLEAN 类型转换

```
BOOLEAN Equal = (a == b) / ( a == b);
```

10.2　在 EDK2 中支持 C++

EDK2 中的程序遵循的是 C 语言的规则，要想让 EDK2 支持 C++ 就必须让 EDK2 兼容 C++ 的规则。改变 EDK2，让它兼容 C++ 规则后，开发 EDK2 应用程序时就可以使用 C++ 的基本特性了，这些基本特性包括：类、多态性等。然而作为一个“懒惰”的程序员，能用 new 就不用 malloc，能自动初始化就不手工初始化变量，能重用代码就不多写一遍。要做到这些，仅靠 C++ 的基本特性还不够，还要让我们的程序能支持全局构造函数和析构函数，支持 new 和 delete 运算符，支持标准模板库。下面就开始让我们的程序一一支持 C++ 的这些特性。

10.2.1　使 EDK2 支持 C++ 基本特性

根据 10.1 节讲述的 C 与 C++ 的三点不同之处，下面就开始讲述如何让 EDK2 兼容 C++ 的这三个规则。

1. 在 C++ 源程序中包含 UEFI 头文件

针对 C++ 与 C 的名字修饰约定不同，在 C++ 文件中，只需将 C 的头文件放入“extern "C"”中即可解决大部分问题，如示例 10-1 所示。

【示例 10-1】 C++ 包含 UEFI 头文件。

```
#ifdef __cplusplus
extern "C"{
#endif
#include <Uefi.h>
#include <Base.h>
#ifdef __cplusplus
}
#endif
```

对于老版本的 EDK2，比较棘手之处在于 EDK 在编译选项中加入了 /FIAutoGen.h，而在 AutoGen.h 中包含了 #include<Base.h>、#include<Uefi.h>、#include<Library/PcdLib.h> 三个头文件，并且这 4 个头文件没有被“extern "C"”包围。针对该情况，有两种解决办法：一种方法是修改编译选项 [BuildOptions]；另一种方法是避免在 C++ 文件中使用 AutoGen.h 中的头文件。下面就详细讲述这两种方法。对较新版本的 EDK2，AutoGen.h 自动添加了“extern "C"”，就没有这种麻烦了，读者可以略过此小节。

方法一：修改编译选项 [BuildOptions]。

在 .inf 文件的 [BuildOptions] 中使用 == 清除 EDK 默认选项，如示例 10-2 所示。

【示例 10-2】 自定义编译选项。

```
[BuildOptions]
  MSFT:*_*_IA32_CC_FLAGS ==  /nologo /c /WX /GS- /W4 /Gs32768 /D UNICODE /O1ib2
      /GL /FIAutoGenLocal.h  /EHs-c- /GR- /GF /Gy /Zi /Gm /D
      EFI_SPECIFICATION_VERSION=0x0002000A /D TIANO_RELEASE_VERSION=0x00080006
      /FAs  /Zc:wchar_t-  /GL- /Od
```

然后在工程文件所在的目录生成 AutoGenLocal.h。AutoGenLocal.h 代码如示例 10-3 所示。

【示例 10-3】 AutoGenLocal.h 头文件。

```
#ifdef __cplusplus
extern "C"{
#endif
#include "AutoGen.h"
#ifdef __cplusplus
}
#endif
```

方法二：避免在 C++ 文件中使用 AutoGen.h 中的头文件。

如果不想使用第一种方法，就要保证所有在 <AutoGen.h>、<Base.h>、<Uefi.h> 和 <Library/PcdLib.h> 中声明过的函数和数组只在 .c 文件中调用。

2. 在 C++ 中使用 NULL

如果想在 C++ 中使用 NULL，则需要将 NULL 定义为 0，如示例 10-4 所示。

【示例 10-4】 在 C++ 中重定义 NULL。

```
#undef NULL
#ifdef __cplusplus
#define NULL 0
#else
#define NULL ((VOID*)0)
#endif
```

3. 在 C++ 中使用布尔类型

在 MdePkg\Include\Base.h:42 定义了 VERIFY_SIZE_OF 宏，其代码如代码清单 10-5 所示，这个宏会导致 C4804 警告。

代码清单 10-5　VERIFY_SIZE_OF 宏

```
#define VERIFY_SIZE_OF(TYPE, Size) extern UINT8
_VerifySizeof##TYPE[(sizeof(TYPE) == (Size)) / (sizeof(TYPE) == (Size))]
```

有两种方法可以消除这种警告：

1）将 VERIFY_SIZE_OF 改为如下内容：

```
 #define VERIFY_SIZE_OF(TYPE, Size) extern UINT8 _VerifySizeof##TYPE
[((UINTN)(sizeof(TYPE) == (Size))) / ((UINTN)(sizeof(TYPE) == (Size)))]
```

2）在 .inf 中的编译选项里添加 /wd4804，具体如下：

```
[BuildOptions]
MSFT:*_*_*_CC_FLAGS = /wd4804
```

到目前为止，我们应该可以用类、模板等 C++ 语法来开发 UEFI 应用了。但是，我们目前还不支持全局类实例，不支持 new 和 delete 操作符。为了支持全局的构造函数和析构函数，我们先来研究一下 Windows 或 Linux 下的程序是如何支持这些特性的。

10.2.2　在 Windows 系统下的程序启动过程

源程序要有 main 函数才能被编译成可执行文件。main 是入口函数吗？像 printf 这样的标准库里的函数是要初始化之后才能被使用的，那么初始化函数在什么地方呢？在 C++ 中声明一个全局对象，这些全局的构造函数是在 main 之前被执行的，这又是如何实现的呢？

一个程序启动后首先进入程序入口点。由 VS 编译的程序，其入口点定义见表 10-1。

表 10-1　VS 程序入口函数[⊖]

程序类型	入口函数	入口函数定义选项
控制台程序（unicode）	wmainCRTStartup	/SUBSYSTEM:CONSOLE
控制台程序（非 unicode）	mainCRTStartup	/SUBSYSTEM:CONSOLE
窗口程序（unicode）	WinMainCRTStartup	/SUBSYSTEM:WINDOWS
窗口程序（非 unicode）	wWinMainCRTStartup	/SUBSYSTEM:WINDOWS
DLL	_DLLMainCRTStartup	/DLL

下面以控制台程序（非 unicode）为例来研究程序启动过程。入口函数 mainCRTStartup 定义在 vc\crt\src 目录下的 crt0.c 文件中，它的主要工作是调用 __tmainCRTStartup 函数。而

⊖ http://msdn.microsoft.com/en-us/library/f9t8842e.aspx。

__tmainCRTStartup 函数的主要工作是调用 main。现在来看看 __tmainCRTStartup 具体做了哪些工作。__tmainCRTStartup 简单代码如代码清单 10-6 所示。

代码清单 10-6　__tmainCRTStartup 函数

```
int __tmainCRTStartup( void )
{
  ...
  if ( !_heap_init(1) )                                  /* 初始化堆 */
   ...
  if( !_mtinit() )                                       /* 初始化多线程 */
   ...
  if ( _ioinit() < 0 )                                   /* 初始化 io */
   ...
  _tcmdln = (_TSCHAR *)GetCommandLineT();                /* 得到命令行参数 */
  _tenvptr = (_TSCHAR *)GetEnvironmentStringsT();        /* 得到环境变量 */
  if ( _tsetargv() < 0 )
    ...
  if ( _tsetenvp() < 0 )
     ...
  initret = _cinit(TRUE);                                /* C 数据初始化 */
  mainret = _tmain(__argc, _targv, _tenviron);
   _cexit();
}
```

可以看到，在 main 之前有大量的初始化工作，我们要关心的是 _cinit，其功能是初始化 C 相关数据，核心代码如代码清单 10-7 所示。

代码清单 10-7　Windows 程序的 _cinit 函数的核心代码

```
initret = _initterm_e( __xi_a, __xi_z );                 //C 初始化
_initterm( __xc_a, __xc_z );                             //C++ 初始化
```

_initterm_e 函数与 _initterm 函数结构相似。下面以 _initterm 为例继续初始化之旅，其代码如代码清单 10-8 所示。

代码清单 10-8　_initterm 函数

```
static void __cdecl _initterm (_PVFV * pfbegin, _PVFV * pfend)
{
  while ( pfbegin < pfend ) {
    if ( *pfbegin != NULL ) (**pfbegin)();
    ++pfbegin;
  }
}
```

可以看出，在 _initterm 中依次执行了从 pfbegin 开始到 pfend 结束的数组里的函数。下面再看 _initterm(__xc_a, __xc_z) 中的“__xc_a, __xc_z”的声明，如代码清单 10-9 所示。

代码清单 10-9 __xc_a 等数组的声明

```
extern _CRTALLOC(".CRT$XIA") _PIFV __xi_a[];
extern _CRTALLOC(".CRT$XIZ") _PIFV __xi_z[];          /* C 初始化器 */
extern _CRTALLOC(".CRT$XCA") _PVFV __xc_a[];
extern _CRTALLOC(".CRT$XCZ") _PVFV __xc_z[];          /* C++ 初始化器 */
extern _CRTALLOC(".CRT$XPA") _PVFV __xp_a[];
extern _CRTALLOC(".CRT$XPZ") _PVFV __xp_z[];          /* C pre-terminators */
extern _CRTALLOC(".CRT$XTA") _PVFV __xt_a[];
extern _CRTALLOC(".CRT$XTZ") _PVFV __xt_z[];          /* C terminators */
```

_CRTALLOC(segment) 的作用是将其后紧跟的数据放到指定的段中。在连接的时候，连接器会将不同文件中名字相同（$ 前的字符串）的段合并在一起，并按照 $ 后面的字符串排序。

编译时，编译器若遇到全局类实例，会生成一个无参数的函数来调用该实例的构造函数，并将该函数指针放置到 CRT$XCU 段中。如果该实例有析构函数，则生成一个无参数的函数调用该析构函数，并将该函数通过 atexit(...) 注册。例如，代码清单 10-10 是一个全局类变量的示例，编译后生成的汇编代码如代码清单 10-11 所示。从代码清单 10-11 可以看出，CRT$XCU 段放置了指向函数 ??__Ehello@@YAXXZ 的指针，该函数调用了构造函数 Hello::Hello，并向 _atexit 注册了退出函数 ??__Fhello@@YAXXZ，而 ??__Fhello@@YAXXZ 则调用了析构函数 Hello::~Hello。

代码清单 10-10 全局类变量示例

```
class Hello
{
public:
    Hello(){printf("Hello! Global Constructor!\n");}
    ~Hello(){printf("Hello! Global Destructor!\n");}
};
Hello hello;
int _tmain(int argc, _TCHAR* argv[])
{
    return 0;
}
```

代码清单 10-11 编译代码清单 10-10 后的汇编代码

```
PUBLIC ??0Hello@@QAE@XZ; Hello::Hello
PUBLIC ??1Hello@@QAE@XZ; Hello::~Hello

CRT$XCU SEGMENT
_hello$initializer$ DD FLAT:??__Ehello@@YAXXZ
CRT$XCU ENDS

text$yc SEGMENT
??__Ehello@@YAXXZ PROC; '该函数初始化 'hello'、COMDAT
```

```
    push ebp
    mov ebp, esp
    sub esp, 192; 000000c0H
    ...
    mov ecx, OFFSET ?hello@@3VHello@@A ; hello
    call ??0Hello@@QAE@XZ; Hello::Hello
    push OFFSET ??__Fhello@@YAXXZ; '通过 atexit 服务注册全局变量 'hello' 的析构函数
    call_atexit
    add esp, 4
    ...
    mov esp, ebp
    pop ebp
    ret 0
??__Ehello@@YAXXZ ENDP; 全局变量 'hello' 的析构函数
text$yc ENDS
text$yd SEGMENT
??__Fhello@@YAXXZ PROC; 该函数被 atexit 服务注册，用于调用全局变量 'hello' 的析构函数
    push ebp
    mov ebp, esp
    sub esp, 192; 000000c0H
    ...
    mov ecx, OFFSET ?hello@@3VHello@@A; hello
    call ??1Hello@@QAE@XZ; Hello::~Hello
    ...
    mov esp, ebp
    pop ebp
    ret 0
??__Fhello@@YAXXZ ENDP;
```

10.2.3　在 Windows 系统下支持全局构造和析构

下面总结一下 Windows 系统下程序执行全局构造函数和析构函数的机制。段（".CRT$XIA"）到段（".CRT$XIZ"）之间存放了 _PIFV 类型的函数指针，这些函数指针在函数 _initterm_e 中被一一执行。段 (".CRT$XCA") 到段 (".CRT$XCZ") 之间存放了 _PVFV 类型的函数指针，这些函数指针在函数 _initterm 中被一一执行。main 函数执行之前，_initterm_e(__xi_a, __xi_z) 和 _initterm(__xc_a, __xc_z) 被调用，也就是说，段 (".CRT$XIA") 到段 (".CRT$XIZ") 之间的函数指针以及段 (".CRT$XCA") 到段 (".CRT$XCZ") 之间的函数指针被一一执行。与初始化类似，main 函数结束后，段 (".CRT$XPA") 到段 (".CRT$XPZ") 之间的函数指针以及段 (".CRT$XTA") 到段 (".CRT$XTZ") 之间的函数指针被一一执行。明白了以上原理，为了支持全局的构造和析构，我们需要注意如下 4 点。

1）声明段 (".CRT$XIA")、段 (".CRT$XIZ")、段 (".CRT$XCA")、段 (".CRT$XCZ")、段 (".CRT$XPA")、段 (".CRT$XPZ")、段 (".CRT$XTA")、段 (".CRT$XTZ")，如示例 10-5 所示。

2）提供 atexit() 服务，如示例 10-6 所示。

3）在进入 UefiMain 时调用 _global_crt_init() 函数，如示例 10-7 所示。而 _global_crt_init 中调用了 _initterm_e(__xi_a, __xi_z); 和 _initterm(__xc_a, __xc_z);。

4）退出之前调用 _global_crt_finish()，如示例 10-7 所示。而 _global_crt_finish() 函数则调用了 _initterm_e(__xp_a, __xp_z); 和 _initterm(__xt_a, __xt_z);。

【示例 10-5】 crt0.cpp 源文件。

```
typedef void (__cdecl *_PVFV)(void);
typedef int  (__cdecl *_PIFV)(void);
typedef void (__cdecl *_PVFI)(int);
#define _CRTALLOC(x) __declspec(allocate(x))
#undef NULL
#define NULL 0
_CRTALLOC(".CRT$XIA") _PIFV __xi_a[] = { NULL };
_CRTALLOC(".CRT$XIZ") _PIFV __xi_z[] = { NULL };
_CRTALLOC(".CRT$XCA") _PVFV __xc_a[] = { NULL };
_CRTALLOC(".CRT$XCZ") _PVFV __xc_z[] = { NULL };
_CRTALLOC(".CRT$XPA") _PVFV __xp_a[] = { NULL };
_CRTALLOC(".CRT$XPZ") _PVFV __xp_z[] = { NULL };
_CRTALLOC(".CRT$XTA") _PVFV __xt_a[] = { NULL };
_CRTALLOC(".CRT$XTZ") _PVFV __xt_z[] = { NULL };
#pragma comment(linker, "/merge:.CRT=.rdata")

template<typename _Functor>
void minitterm(_Functor* pfbegin, _Functor* pfend)
{
  while(pfbegin < pfend) {
    if (*pfbegin != NULL)(**pfbegin)();
    ++pfbegin;
  }
}
void _global_crt_init()
{
    minitterm(__xc_a,__xc_z);
    minitterm(__xi_a,__xi_z);
}
void _global_crt_finish()
{
    minitterm(__xp_a,__xp_z);
    minitterm(__xt_a,__xt_z);
}
```

【示例 10-6】 atexit 服务。

```
_PVFV *atexits  = NULL;
static int num_atexit = 0;
```

```
static int max_atexit =-1;
int atexit(void (*handler)(void))
{
    if(handler == NULL)   return 0;
    if(num_atexit >= max_atexit){
        max_atexit += 32;
        _PVFV* old_handler = atexits;
        atexits    = new _PVFV[max_atexit];
        if(atexits == NULL) {
            atexits = old_handler;
            return -1;
        }
        for(int i=0;i<max_atexit-32;i++)
            atexits[i] = old_handler[i];
        delete old_handler;
    }
    atexits[num_atexit++] = handler;
    return 0;
}
//从后至前调用 atexits 中的函数指针
void static global_finish()
{
    for(int i =num_atexit-1; i>= 0; i--){
        if ( atexits[i] != NULL )  (*atexits[i])();
    }
    delete atexits;
}
//将 global_finish 放到段 ".CRT$XPA 和段 ".CRT$XPZ" 之间，以便退出前调用
#pragma section(".CRT$XPYZ", long ,read)
_CRTALLOC(".CRT$XPYZ") _PVFV __xp_finitz[] = { global_finish };
```

【示例 10-7】 C++ 程序入口函数。

```
EFI_STATUS
UefiMain(IN EFI_HANDLE ImageHandle, IN EFI_SYSTEM_TABLE *SystemTable)
{
  _global_crt_init ();            //初始化，该函数实现在示例 10-5 中
  //此处编写用户自己的代码
  _global_crt_finish();           //析构，该函数实现在示例 10-5 中
}
```

10.2.4　在 Linux 系统下的程序启动过程

回顾一下 UEFI 应用程序的启动过程：UEFI 首先通过 LoadImage Protocol 将可执行文件（.efi）加载到内存以形成程序映像，然后调用该映像的入口函数，在入口函数中首先进行一系列的初始化，然后执行 .inf 指定的入口函数（如 UefiMain），结束后执行一系列析构函数。

Linux 系统下程序启动过程与之相似（其实应该说是 UEFI 应用程序的启动过程与 Linux 下应用程序的启动过程相似），首先由操作系统内核将可执行文件加载到内存，然后调用入口函数 _start，其后过程分为以下三个部分：

1）初始化。

2）调用 main 函数。

3）析构并退出。

Linux 下程序启动过程如图 10-1 所示。

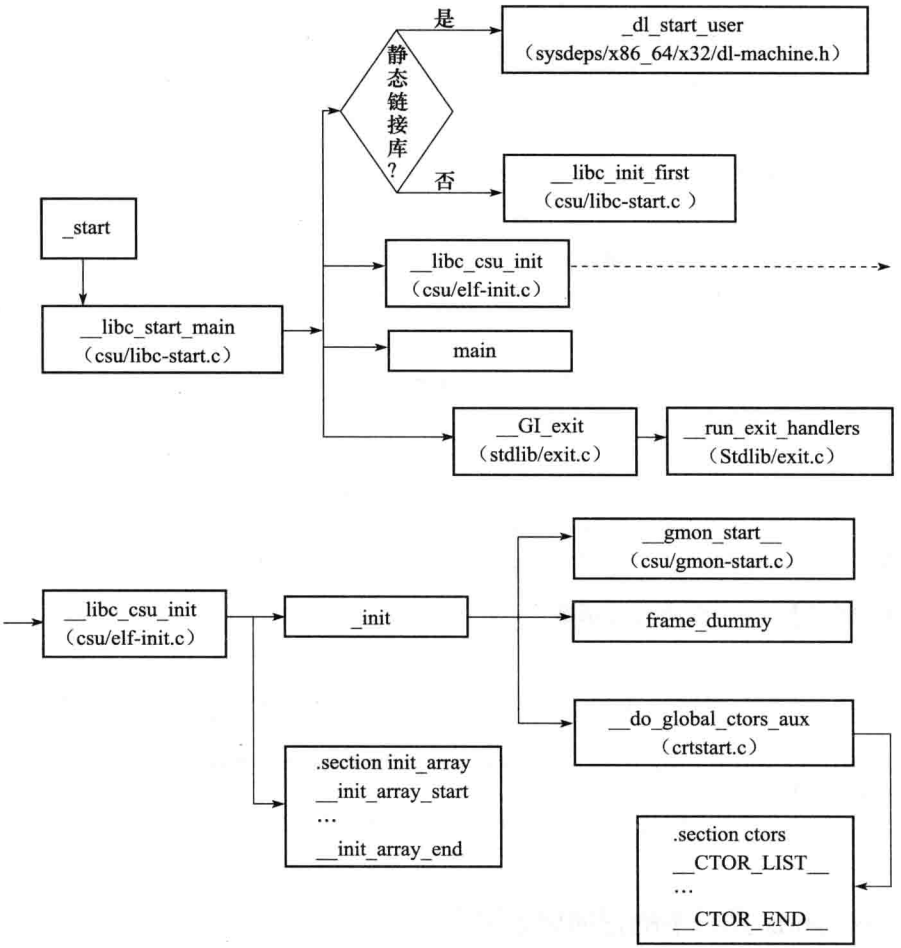


图 10-1　Linux 下程序启动过程

1. 启动过程概述

下面以代码清单 10-12 中的程序为例来说明应用程序的启动过程。

代码清单 10-12　Linux 应用程序

```
#include <stdio.h>
// __attribute__ ((constructor)) 告诉编译器此函数需要在过程 1（初始化）中执行
void __attribute__ ((constructor)) premain()
{
printf(" premain\n" );
}
// __attribute__ ((destructor)) 告诉编译器此函数需要在过程 3（析构）中执行
void __attribute__ ((destructor)) postmain()
{
    printf( "postmain\n" );
}

int main()
{
    printf( "main\n" );
    return 0;
}
```

首先编译、执行以上代码，从其输出可以看出程序的执行顺序正是 premain → main → postmain。

```
$ gcc -g -o testStart testStart.c
$ ./testStart
premain
main
    postmain
$
```

从入口函数看起，ELF 文件头的 e_entry 项指明了入口地址，可以通过 objdump –f 命令查看 ELF 文件头信息，如下所示。

```
$ objdump -f /home/zhdai/test/testStart

/home/zhdai/test/testStart:     file format elf32-i386
architecture: i386, flags 0x00000112:
EXEC_P, HAS_SYMS, D_PAGED
start address 0x08048330
```

从上面的结果可以看出，应用程序 testStart 入口地址是 0x08048330。用 objdump –S 命令可以反编译可执行文件，反汇编后的代码如代码清单 10-13 所示。

代码清单 10-13　testStart 反汇编后的代码

```
08048430 <__libc_csu_fini>:
...
08048440 <__libc_csu_init>:
...
```

```
08048330 <_start>:
 8048330: 31 ed              xor  %ebp,%ebp
 8048332: 5e                 pop  %esi
 8048333: 89 e1              mov  %esp,%ecx
 8048335: 83 e4 f0           and  $0xfffffff0,%esp
 8048338: 50                 push %eax            ; 填充任意数
 8048339: 54                 push %esp            ;stack_end
 804833a: 52                 push %edx            ;rtld_fini
 804833b: 68 30 84 04 08     push $0x8048430      ;fini
 8048340: 68 40 84 04 08     push $0x8048440      ;init
 8048345: 51                 push %ecx            ;ubp_av
 8048346: 56                 push %esi            ;argc
 8048347: 68 0c 84 04 08     push $0x804840c      ;main
 804834c: e8 bb ff ff ff     call 804830c <__libc_start_main@plt>
 8048351: f4                 hlt
 8048352: 90                 nop
 8048353: 90                 nop
```

从上面的代码中可以看出，0x08048330 正是 _start 的地址。_start 的工作就是调用函数 __libc_start_main。将上面的汇编代码翻译成 C 语言代码，如代码清单 10-14 所示。

代码清单 10-14 _start 函数

```
void start()
{
  __libc_start_main(argc, ubp_av,  __libc_csu_init,  __lib_csu_finit,
    rtld_fini, stack_end);
}
```

代码清单 10-15 是 __libc_start_main 的函数原型，后面我们会详细介绍。

代码清单 10-15 __libc_start_main 函数原型

```
STATIC int LIBC_START_MAIN (int (*main) (int, char **, char **MAIN_AUXVEC_DECL),
                 int argc,
                 char *__unbounded *__unbounded ubp_av,
#ifdef  LIBC_START_MAIN_AUXVEC_ARG
                 ElfW(auxv_t) *__unbounded auxvec,
#endif
                 __typeof (main) init,
                 void (*fini) (void),
                 void (*rtld_fini) (void),
                 void *__unbounded stack_end)
      __attribute__ ((noreturn));
```

__libc_start_main 到底做了什么呢？为了进一步看清程序的启动过程，我们用 gdb 来查看函数调用栈。首先将断点设在 premain，查看初始化过程的函数调用栈。

```
(gdb) b premain
Breakpoint 1 at 0x80483ea: file testStart.c, line 21.
(gdb) r
Starting program: /home/zhdai/test/testStart

Breakpoint 1, premain () at testStart.c:21
21          printf( __FUNCTION__);
(gdb) bt
#0  premain () at testStart.c:21
#1  0x080484ad in __do_global_ctors_aux ()
#2  0x080482e8 in _init ()
#3  0x08048449 in __libc_csu_init ()
#4  0x00144b74 in __libc_start_main (main=0x80483f8 <main>, argc=1, ubp_av=0xbffff744,
    init=0x8048430 <__libc_csu_init>, fini=0x8048420 <__libc_csu_fini>,
    rtld_fini=0x11e030 <_dl_fini>, stack_end=0xbffff73c) at libc-start.c:185
#5  0x08048351 in _start ()
```

可以看出初始化的大致过程是 _start → __libc_start_main → __libc_csu_init → _init → __do_global_ctors_aux。

再将断点设在 postmain，查看退出过程的函数调用栈。

```
(gdb) b postmain
Breakpoint 1 at 0x80483fe: file testStart.c, line 25.
(gdb) r
Starting program: /home/zhdai/test/testStart

Breakpoint 1, postmain () at testStart.c:25
25          printf( __FUNCTION__);
(gdb) bt
#0  postmain () at testStart.c:25
#1  0x0804839f in __do_global_dtors_aux ()
#2  0x080484e4 in _fini ()
#3  0x0011e216 in _dl_fini () at dl-fini.c:248
#4  0x0015d1bf in __run_exit_handlers (status=0, listp=0x284324, run_list_
    atexit=true) at exit.c:78
#5  0x0015d22f in *__GI_exit (status=0) at exit.c:100
#6  0x00144bde in __libc_start_main (main=0x804840c <main>, argc=1, ubp_av=0xbffff724,
    init=0x8048440 <__libc_csu_init>, fini=0x8048430 <__libc_csu_fini>,
    rtld_fini=0x11e030 <_dl_fini>, stack_end=0xbffff71c) at libc-start.c:258
#7  0x08048351 in _start ()
```

可以看出整个退出的大致过程是 _start → __libc_start_main → __GI_exit → __run_exit_handlers → _dl_fini → _fini → __do_global_dtors_aux。

2. 初始化过程

从入口函数 _start 到 __libc_start_main，再到 main 函数，将会有一段漫长的路要走，这

个过程是程序的初始化过程，为 main 函数创建执行环境。下面从 __libc_start_main（见代码清单 10-16）入手开始深入分析程序的执行过程。

代码清单 10-16 __libc_start_main 函数

```
STATIC int
LIBC_START_MAIN (int (*main) (int, char **, char ** MAIN_AUXVEC_DECL),
         int argc, char *__unbounded *__unbounded ubp_av,
#ifdef LIBC_START_MAIN_AUXVEC_ARG
         ElfW(auxv_t) *__unbounded auxvec,
#endif
         __typeof (main) init,
         void (*fini) (void),
         void (*rtld_fini) (void), void *__unbounded stack_end)
{
// 步骤 1: 初始化
 ...
  if (__builtin_expect (rtld_fini != NULL, 1))
     __cxa_atexit ((void (*) (void *)) rtld_fini, NULL, NULL);
#ifndef SHARED
  __libc_init_first (argc, argv, __environ);
  if (fini)
      __cxa_atexit ((void (*) (void *)) fini, NULL, NULL);
  if (__builtin_expect (__libc_enable_secure, 0))
      __libc_check_standard_fds ();
#endif
  ...
  if (init)
    (*init) (argc, argv, __environ MAIN_AUXVEC_PARAM);
  ...
  // 步骤 2: 调用 main 函数
  result = main (argc, argv, __environ MAIN_AUXVEC_PARAM);
  // 步骤 3: 清理
  exit (result);
}
```

先来看初始化部分，这部分主要做了三件事：注册退出函数 rtld_fini 和 __libc_csu_fini(_fini)，调用 __libc_init_first（在静态链接方式下），调用 __libc_csu_init（_init）。__libc_csu_init 函数如代码清单 10-17 所示。

代码清单 10-17 __libc_csu_init 函数

```
void __libc_csu_init (int argc, char **argv, char **envp)
{
  _init ();
  const size_t size = __init_array_end - __init_array_start;
  for (size_t i = 0; i < size; i++)
      (*__init_array_start [i]) (argc, argv, envp);
}
```

__libc_csu_init 调用了 _init 函数，并且执行 __init_array 段内的所有函数。_init 函数的主要任务是调用 __gmon_start__、frame_dummy、__do_global_ctors_aux 三个函数，如代码清单 10-18 所示。

代码清单 10-18　_init 函数

```
8048291:        e8 1e 00 00 00          call   80482b4 <__gmon_start__@plt>
8048296:        e8 d5 00 00 00          call   8048370 <frame_dummy>
804829b:        e8 70 01 00 00          call   8048410 <__do_global_ctors_aux>
```

__gmon_start__ 用于 profiling，在 gcc 编译选项加入 -pg 时，这个函数被 __gmon_start__@plt 调用；当 gcc 编译选项中没有 -pg 选项时，__gmon_start__ 的地址为 0，从而被 __gmon_start__@plt 略过。更多关于 __gmon_start__ 的信息，可以查看 csu/gmon-start.c。frame_dummy 用于异常处理，不是我们讲述的重点，不再赘述。我们重点来看 __do_global_ctors_aux，源码如代码清单 10-19 所示。

代码清单 10-19　__do_global_ctors_aux 函数

```
static void __attribute__((used))
__do_global_ctors_aux (void)
{
  func_ptr *p;
  for (p = __CTOR_END__ - 1; *p != (func_ptr) -1; p--)
    (*p) ();
}
```

该函数倒序执行数组中的所有函数，而该数组以 __CTOR_END__ 结尾。这个数组是如何产生的呢？前面我们介绍 Windows 平台应用程序启动过程的时候讲过，VC 会把全局构造函数放到 .CRT$XI 段内组成一个数组，与之类似，GCC 也要将全局构造函数放到段 .ctors 中，只是这个行为是由连接脚本控制的。连接时，连接器要将所有文件连接成一个文件，因而要将所有目标文件的 .ctors 段组织为一个 .ctors 段，由连接脚本指定组织的规则及顺序。对连接器脚本而言，目标文件的段称为输入段，连接生成的文件的段称为输出段。脚本中的 SECTIONS 部分定义了如何由输入段生产输出段。通过“ld –verbose”命令可以查看连接脚本，默认的连接脚本如代码清单 10-20 所示。

代码清单 10-20　连接脚本

```
.ctor ALIGN(4) :
  {
    __CTOR_START = .;   /* . 表示定位符，__CTOR_START 位于 .ctor 段的最前面 */
/* 必须保证 crtbegin.o 中的 .ctors 段在最前面 */
    KEEP (*crtbegin.o(.ctors))        /* 所有 *crtbegin.o 文件的 .ctors 段 */
    KEEP (*crtbegin?.o(.ctors))       /* ? 匹配一个有效符号 */
```

```
  /* crtend.o 中的 .ctors 在其他文件 .ctors.* 段的后面,
   * 因而下面这一规则中要排除 *crtend.o 及 *crtend?.o 文件
   * EXCLUDE_FILE(file) 表示除 file 之外的文件
   */
    KEEP (*(EXCLUDE_FILE (*crtend.o *crtend?.o) .ctors))
    KEEP (*(SORT(.ctors.*)))        /* 按段名排序 */
    KEEP (*(.ctors))
    __CTOR_END = .;                  /*__CTOR_END 位于 .ctor 段最末尾 */
  }
  .dtor ALIGN(4) :
  {
    __DTOR_START = .;
    KEEP (*crtbegin.o(.dtors))
    KEEP (*crtbegin?.o(.dtors))
    KEEP (*(EXCLUDE_FILE (*crtend.o *crtend?.o) .dtors))
    KEEP (*(SORT(.dtors.*)))
    KEEP (*(.dtors))
    __DTOR_END = .;
  }
```

从连接脚本可以看出符号（或者称为变量）__CTOR_START 位于 .ctor 段最开始位置，__CTOR_END 位于 .ctor 段最末尾位置。__CTOR__START 和 __CTOR_END 之间是所有定义在 .ctors 段的函数指针。因此，我们可以用 __CTOR_END 定位这些函数指针。现在回头看 __do_global_ctors_aux，可以很容易理解它正是倒序执行了 .ctor 段的所有函数。

回顾一下代码清单 10-12 中的示例程序，premain 函数是这样定义的：

```
void __attribute__ ((constructor)) premain()
```

编译器会在 .ctors 段中声明一个指针，并将该指针指向 premain。

自此，初始化部分已经基本结束。另外，再说点题外话，看一下 crtbegin.o 及 crtend.o，这两个文件是 gcc 的一部分，对应的源文件是同一个文件，都是 crtstuff.c。代码清单 10-21 是 libgcc 的 Makefile 中的 crtstuff.c 相关部分。

代码清单 10-21　libgcc 的 Makefile 中的 crtstuff.c 相关部分

```
crtbegin$(objext): $(gcc_srcdir)/crtstuff.c
   $(crt_compile) $(CRTSTUFF_T_CFLAGS) \
     -c $(gcc_srcdir)/crtstuff.c -DCRT_BEGIN

crtend$(objext): $(gcc_srcdir)/crtstuff.c
   $(crt_compile) $(CRTSTUFF_T_CFLAGS) \
     -c $(gcc_srcdir)/crtstuff.c -DCRT_END
```

稍后我们还会讲到 crtstuff.c 定义处理全局初始化及析构函数的辅助函数的内容。

3. 调用主函数 main

__libc_start_main 三个部分中的第一部分初始化介绍完毕，下面开始介绍第二部分，即调用 main 函数，如代码清单 10-22 所示。

代码清单 10-22　__libc_start_main 的第二部分

```
// 步骤 2：调用 main 函数
result = main (argc, argv, __environ MAIN_AUXVEC_PARAM);
```

4. 清理过程

接下来是最后一步，即处理退出函数，如代码清单 10-23 所示。

代码清单 10-23　__libc_start_main 的第三部分

```
// 步骤 3：清理
exit (result);
```

exit 函数执行了所有通过 atexit(…) 注册的函数。值得一提的是，__do_global_ctors_aux 被 atexit 注册，如代码清单 10-24 所示。

代码清单 10-24　__do_global_ctors_aux 函数

```
static void __attribute__((used))
__do_global_ctors_aux_1 (void)
{
  atexit(__do_global_ctors_aux);
}
```

因而 __do_global_ctors_aux 在程序退出前会被执行。__do_global_dtors_aux 执行了 .dtors 段的所有函数，也就是 __DTOR_LIST__ 和 __DTOR_END__ 之间的所有函数指针，其代码如代码清单 10-25 所示。

代码清单 10-25　__do_global_dtors_aux 函数

```
static void __attribute__((used))
__do_global_dtors_aux (void)
{
...
#ifdef FINI_ARRAY_SECTION_ASM_OP
#elif defined(HIDDEN_DTOR_LIST_END)
    {
      extern func_ptr __DTOR_END__[] __attribute__((visibility ("hidden")));
      static size_t dtor_idx;
      const size_t max_idx = __DTOR_END__ - __DTOR_LIST__ - 1;
      func_ptr f;
```

```
        while (dtor_idx < max_idx) {
            f = __DTOR_LIST__[++dtor_idx];
            f ();
        }
    }
#else /* !defined (FINI_ARRAY_SECTION_ASM_OP) */
    {
        static func_ptr *p = __DTOR_LIST__ + 1;
        func_ptr f;

        while ((f = *p)) {
            p++;
            f ();
        }
}
#endif /* !defined(FINI_ARRAY_SECTION_ASM_OP) */

...
}
```

10.2.5 在 Linux 系统下支持全局构造和析构

明白了 Linux 应用程序的执行过程，我们就可以用类似的方法使 .efi 文件支持全局构造和析构了。其基本原理可以用两句话概括：① 在 main(或者 UefiMain/ShellAppMain) 函数入口处执行 .ctors 段的所有函数；② 在 main（或者 UefiMain/ShellAppMain）返回前执行 .dtors 段的所有函数。

首先要在 .ctors 段声明 __CTOR_LIST__ 和 __CTOR_END__ 变量，在 .dtors 段声明 __DTOR_LIST__ 和 __DTOR_END__ 变量，如示例 10-8 所示。

【示例 10-8】 声明 __CTOR_LIST__、__CTOR_END__、__DTOR_LIST__ 和 __DTOR_END__。

```
typedef void (*func_ptr) (void);
func_ptr __CTOR_LIST__[1]
  __attribute__ ((__used__, section(".ctors"), aligned(sizeof(func_ptr))))
  = { (func_ptr) (-1) };
func_ptr __CTOR_END__[1]
  __attribute__((__used__, section(".ctors"), aligned(sizeof(func_ptr))))
  = { (func_ptr) 0 };
func_ptr __DTOR_LIST__[1]
  __attribute__ ((__used__, section(".dtors"), aligned(sizeof(func_ptr))))
  = { (func_ptr) (-1) };
func_ptr __DTOR_END__[1]
  __attribute__((__used__, section(".dtors"), aligned(sizeof(func_ptr))))
  = { (func_ptr) 0 };
```

然后告诉连接器将所有 .ctors 段的其他变量放到 __CTOR_LIST__ 和 __CTOR_END__ 之间，将所有 .dtors 段的其他变量放到 __DTOR_LIST__ 和 __DTOR_END__ 之间。

在 Conf/tools_def.txt 中修改连接选项，指定连接器脚本，如示例 10-9 所示。

【示例 10-9】 指定连接器脚本。

```
DEFINE GCC44_IA32_X64_DLINK_COMMON   = -nostdlib -n -q --gc-sections
--script=$(EDK_TOOLS_PATH)/Scripts/gcc4.4-ld-script
```

gcc4.4-ld-script 中关于 .ctors 和 .dtors 的内容如示例 10-10 所示。

【示例 10-10】 GCC 编译器下的 UEFI 连接脚本。

```
.ctors          :
  {
      __CTOR_LIST__ = .;
      /*LONG((__CTOR_END__ - __CTOR_LIST__) / 4 - 2) */
      KEEP (*(SORT(.ctors.*)))
          KEEP (*(.ctors))
      __CTOR_END__ = .;
  }
  .dtors          :
  {
      __DTOR_LIST__ = .;
      /*LONG((__DTOR_END__ - __DTOR_LIST__) / 4 - 2)*/
      KEEP (*(SORT(.dtors.*)))
      KEEP (*(.dtors))
      __DTOR_END__ = .;
  }
```

接下来，要在 main(或者 UefiMain/ShellAppMain) 入口处调用 __do_global_ctors_aux 函数，此函数会依次调用每一个 .ctors 段的函数，如示例 10-11 所示。

【示例 10-11】 UEFI 中的 __do_global_ctors_aux 函数。

```
extern "C"     void __attribute__((used)) __do_global_ctors_aux (void)
{
    func_ptr *p =  __CTOR_END__ - 1;
    for (p = __CTOR_LIST__; p < __CTOR_END__ ; p++) {
        if (*p && *p != (func_ptr)-1)
            (*p) ();
    }
}
```

然后要在 main（或者 UefiMain/ShellAppMain）返回前调用 __do_global_dtors_aux 函数，该函数将依次调用 .dtors 段内的每一个函数，如示例 10-12 所示。

【示例 10-12】 UEFI 中的 __do_global_dtors_aux 函数。

```
extern "C" void __attribute__((used)) __do_global_dtors_aux (void)
```

```
{
    func_ptr *p =  __DTOR_END__ - 1;
    for (p = __DTOR_END__ - 1 ; p >= __DTOR_LIST__; p--)  {
        if (*p && *p != (func_ptr)-1)
            (*p) ();
    }
}
```

然后还要提供 atexit 服务。Linux 下的 atexit 服务与 Windows 下的 atexit 服务基本一样，代码如示例 10-6 所示。用户需要将指向函数 global_finish 的指针放到 .dtors 段，以便 __do_global_dtors_aux 函数调用，如示例 10-13 所示。

【示例 10-13】 在段“.ctors”中声明 global_finish 函数指针。

```
 __xp_finitz[] __attribute__((__used__, section(".ctors"),
aligned(sizeof(func_ptr))))= { global_finish };
```

最后要在 main（或 UefiMain）函数中调用 __do_global_ctors_aux 和 __do_global_dtors_aux，如示例 10-14 所示。

【示例 10-14】 在入口函数中调用 __do_global_ctors_aux 和 __do_global_dtors_aux。

```
EFI_STATUS

UefiMain(IN EFI_HANDLE ImageHandle,IN EFI_SYSTEM_TABLE  *SystemTable)
{
    __do_global_ctors_aux (void);
    //此处加入用户的代码
    __do_global_dtors_aux (void);
}
```

至此，在 Linux 下编译的 .efi 应用程序已经能够支持全局的构造函数和析构函数了。

10.2.6 支持 new 和 delete

在 C++ 中遇到 new 操作符时，首先调用 operator new 函数获得内存，然后在获得的内存上调用构造函数，而在遇到 delete 操作符时，则首先调用析构函数，然后调用 operator delete 函数释放内存。另外，遇到 new operator[] 时，首先调用 operator new[] 函数获得内存，然后记录数组的大小，并在获得的内存上依次调用构造函数，而遇到 delete operator[] 时，则首先根据内存记录的数组大小依次调用析构函数，然后调用 operator delete[] 函数释放内存。

如此我们只需提供“void * operator new(size_t Size);”、“void * operator new[](size_t cb); ”、“ void operator delete(void * p);”、“ void operator delete[](void * p);” 函数即可支持基本的 new 和 delete 操作符了，相关代码如示例 10-15 所示。

【示例 10-15】 operator new 和 operator delete。

```
 __inline__ void *  operator new( size_t Size )
{
    void *RetVal;
    EFI_STATUS  Status;
    if( Size == 0) {
        return NULL;
    }
    Status = gBS->AllocatePool( EfiLoaderData, (UINTN)Size, &RetVal);
    if( Status != EFI_SUCCESS) {
        RetVal  = NULL;
    }
return RetVal;
}

__inline__ void *  operator new[]( size_t cb )
{
    void *res = operator new(cb);
    return res;
}
__inline__ void operator delete( void * p )
{
    if(p != NULL)
(void) gBS->FreePool (p);
}
__inline__ void operator delete[]( void * p )
{
    operator delete(p);
}
```

在 GcppLib.cpp 中要提供 operator new、operator new[]、operator delete、operator delete[] 函数，以及 STL 所需函数。

10.2.7　支持 STL

我们不能直接使用系统的 STL，因为系统的 STL 会调用自己的 Std 函数。我们需将 STL 从系统目录中剥离出来，然后连接 EDK2 中的 StdPkg，再补上缺失的函数（C++ run time 库中的一些函数）即可。相关的代码在 uefi\book\GcppPkg\Include 目录中。

10.3　GcppPkg 概览

回顾一下，要完全支持 C++，我们需要做以下几件事：

1）用“extern"C"”引用 UEFI 头文件；消除 BOOLEAN 类型产生的错误；修改 NULL 的定义。

2）支持全局构造和析构函数。

3）支持 new 和 delete 操作符。

4）支持标准模板库 STL。

每次写 C++ 程序都考虑以上几件事将是非常烦琐的事情，我们可以将 C++ 支撑函数组织成一个包，方便使用 C++ 开发 UEFI 工程。下面我们以 Linux 开发环境为例介绍 GcppPkg。GcppPkg 的相关代码在 uefi\book\GcppPkg 目录中。以后利用 C++ 开发 UEFI 应用时，只需要使用 GcppPkg 就可以了。

1. GcppPkg 的组成

GcppPkg 主要作用是提供 C++ 运行时库，这个运行时库提供的功能有：C++ 程序的初始化与析构，new 和 delete 操作符，标准模板库以及其他函数。

一个 Package 通常包含 .dec 文件、.dsc 文件、Include 文件夹和 Library 文件夹，GcppPkg 也不例外。下面详细介绍这 4 个部分的组成。

（1）GcppPkg 的 .dec 文件

.dec 文件用于引用该 Package，包含了 include 目录、package 定义的 Protocol 的 GUID，以及要放到系统里的变量。在 GcppPkg 中，我们只需要定义 [Includes]。GcppPkg 的 .dec 文件如示例 10-16 所示。

【示例 10-16】 GcppPkg.dec 文件。

```
[Defines]
  DEC_SPECIFICATION = 0x00010005
  PACKAGE_NAME = EdkGCppPkg
  PACKAGE_GUID = e2805c44-8985-11db-9e98-0040d0c0d0cc
  PACKAGE_VERSION = 1.0
[Includes]
  Include
  Include/parallel
[Includes.IA32]
  Include/i486-linux-gnu
```

（2）GcppPkg 的 .dsc 文件

.dsc 文件用于编译该 Package，包含了依赖的 library，以及要编译的 library。因为我们不需要单独编译 GcppPkg，所以可以不实现 GcppPkg.dsc 文件。

（3）GcppPkg 的 Include 文件夹

在 Include 文件夹内，将放置 STL 模板。

（4）GcppPkg 的 Library 文件夹

在 Library 文件夹内将放置 GcppLib，主要包含全局构造和析构支撑函数，以及 new 和 delete 支撑函数。

GcppLib 包含三个文件，其中 GcppLib.inf 是工程文件，GcppLib.cpp 包含 STL 所需函数，GcppCrt.cpp 包含全局构造和析构的支撑函数以及 main 函数。

在 GcppLib.inf 中，要指定 MODULE_TYPE 为 BASE，并包含源文件 GcppLib.cpp 和 GcppCrt.cpp，具体代码如示例 10-17 所示。

【示例 10-17】 GcppLib.inf 文件。

```
[Defines]
  INF_VERSION = 0x00010005
  BASE_NAME = libgcpp
  FILE_GUID = 348C4D62-BFBD-4882-9ECE-C80BB1C64336
  VERSION_STRING = 1.0
  MODULE_TYPE = BASE
  LIBRARY_CLASS = GcppLib
[Sources]
  GcppLib.cpp
  GcppCrt.cpp
[Packages]
  MdePkg/MdePkg.dec
  MdeModulePkg/MdeModulePkg.dec
  StdLib/StdLib.dec
[LibraryClasses]
  UefiBootServicesTableLib
  UefiLib
```

在 GcppCrt.cpp 中要声明 _CTOR_LIST__、__CTOR_END__、__DTOR_LIST__ 和 __DTOR_END__ 变量，实现 __do_global_ctors_aux、__do_global_dtors_aux 以及 atexit 函数。

最主要的是，我们要帮助用户调用全局初始化和析构函数，并调用用户提供的 cppMain（Argc,Argv）函数，这样用户只需要实现 cppMain(int Argc, char** Argv) 就可以了。实现这些代码的 C++main 函数代码如示例 10-18 所示，这部分代码已经包含在 GcppLib 中。

【示例 10-18】 UDFI 应用程序 C++ main 函数。

```
int main (IN int Argc, IN char **Argv  )
{
    int result;
    __do_global_ctors_aux ();            // 全局初始化函数
    result = cppMain(Argc, Argv);        // 调用用户提供的 cppMain 函数
    __do_global_dtors_aux ();            // 全局析构函数
    return result;
}
```

2. 让编译器自动识别 C 与 C++ 源文件

通常 C 程序要用 gcc 编译，C++ 程序要用 g++ 编译。有很多种方法使得可以根据源程序类型自动选择编译器，这里介绍一种比较简单的方法。在 ~/bin 目录下建立可执行脚本文件

gcc，其内容如示例 10-19 所示。脚本 gcc 的主要功能是根据源程序类型选择编译器，然后调用相应的编译器编译源文件。

【示例 10-19】 脚本 gcc。

```
#!/bin/bash
iscpp=0
for i in "$@" ;do
    if [ "-" != "${i:0:1}" ]; then
        if [ "cpp" = "${i##*.}" ]; then
            iscpp=1
        fi
    fi
done
if [ $iscpp = 0 ]; then
    /usr/bin/gcc $@
else
    /usr/bin/g++ $@
fi
```

然后在命令行执行如下命令将默认的 gcc 命令设置为 ~/bin/gcc。

```
EDK2$ export PATH=~/bin;$PATH
```

10.4 测试 GcppPkg

GcppPkg 提供 Linux 环境下 C++ 程序的运行时库，因而该 GcppPkg 只能在 Linux 环境下使用。下面我们以一个简单的示例介绍如何在 Linux 开发环境下使用 GcppPkg 建立一个 C++ UEFI 应用程序模块。任何一个应用程序模块都包含工程文件和源文件两部分，C++ 应用程序模块也不例外，下面就从这两个方面来讲述这个示例。

1. C++ 应用程序模块的工程文件

示例 10-20 是这个示例的工程文件，要使用 GcppPkg，必须对工程文件做如下设置：

1）ENTRY_POINT 定义为 ShellCEntryLib。

2）STL 模板中的函数、类会调用 Std 函数，因而 C++ UEFI 应用程序需要连接 StdLib 库，使用 ShellCEntryLib 也就是一种很自然的选择。

3）[Packages] 块中包含 GcppPkg、StdLib 等。

4）[LibraryClasses] 块中包含 GcppLib、LibC 等库。为了包含 GcppLib，在包的 .dsc 文件中要定义 GcppLib，定义方法是在 .dsc 文件中的 [LibraryClasses] 块中包含下面一行代码：GcppLib|Uefi/GcppPkg/Library/GcppLib.inf。

【示例 10-20】 testcpp.inf 文件。

```
 [Defines]
  INF_VERSION = 0x00010006
  BASE_NAME = testcpp
  FILE_GUID = 4ea97c46-7491-4dfd-b442-74798713ce5f
  VERSION_STRING = 0.1
  MODULE_TYPE = UEFI_APPLICATION
  ENTRY_POINT = ShellCEntryLib
[Sources]
  testcpp.cpp
[Packages]
  MdePkg/MdePkg.dec
  MdeModulePkg/MdeModulePkg.dec
  uefi/book/GcppPkg/GcppPkg.dec
  StdLib/StdLib.dec
[LibraryClasses]
  ShellCEntryLib
  PrintLib
  LibC
  LibStdio
  GcppLib
```

2. C++ 应用程序模块的源文件

根据 10.3 节中 GcppPkg 对 GcppLib 的说明，使用 GcppPkg 的应用程序应提供 cppMain 作为入口函数。应用程序的启动过程为 ShellCEntryLib（模块入口函数）→ ShellAppMain(LibC) → main(GcppLib) → cppMain。这个测试程序的源文件如示例 10-21 所示，在本示例中，定义了一个全局类变量 gGlobal，gGlobal 的构造函数将在 cppMain 之前被执行。在 cppMain 中使用了标准模板库中的容器 std::vector。UEFI 和 C 标准库的头文件必须包含在“extern "C"”之内，C++ 标准库的头文件在“extern "C"”之外。

【示例 10-21】 testcpp.cpp 源文件。

```
extern "C"{                 //extern "C" 中包含 UEFI 头文件及 C 标准库的头文件
#include <Uefi.h>
#include <Library/BaseLib.h>
#include <Library/UefiLib.h>
#include <stdlib.h>
}
#include <vector>           //extern "C" 之外包含 C++ 标准库的头文件
class Global{
public:
  Global(){Print((const CHAR16*)L"Hello, global constructor!\n");}
};
Global gGlobal;             //此处会自动调用一个构造函数，并且该构造函数在 cppMain 之前执行
int cppMain (IN int Argc,  IN char **Argv)
```

```
{
    std::vector<int> a;  //使用 STL 容器
    a.push_back(1);
    a.push_back(10);
    Print((const CHAR16*)L"Hello CPP\n");
    for(std::vector<int>::iterator it = a.begin(); it != a.end(); it++) {
        Print((const CHAR16*)L"%d\n", *it);
    }
    return 0;
}
```

10.5 本章小结

本章主要讲述了如何使用 C++ 开发 UEFI 应用。

要支持 C++ 特性，首先要从编译器的角度理解 C 与 C++ 的区别。开发 UEFI 应用时我们主要关注 C 与 C++ 的 3 点区别：

1）编译后函数名的区别。

2）BOOL EAN 类型的区别。

3）NULL 类型的区别。

正确理解上述 3 点后，在 UEFI 开发中使用 C++ 时也就可以使用 C++ 的大部分特性，如类、重载、多态等。

本章分析了 Windows 应用程序和 Linux 应用程序的启动过程以及全局构造 / 析构函数实现机制，并讲述了让 C++ UEFI 应用程序支持全局构造 / 析构函数的方法。本章还讲述了如何支持 new/delete 操作符，以及如何支持标准模板库。

本章介绍了如何编制一个 C++ 包，并以 testcpp 工程为例介绍了如何编写一个简单的 C++ 应用程序。

在接下来的两章中，我们将用 C++ 实现一个简单的视频播放器。

第 11 章 Chapter 11

GUI 基础

字符串、GraphOut（图像输出）和字体是 UEFI GUI（图形用户界面）的基础，学习 GUI 之前首先要熟悉这三个基础元素。同字符串密切相关的一个概念是字符串的本地化，字符串的本地化需要多语言（英文等）的支持。GraphOut 的作用是将位图显示到屏幕上。而字体的作用是提供一种将字符或字符串转换成位图的机制。下面就这三个方面分别详细介绍使用方法。

11.1 字符串

UEFI 中字符串有两种，一种是 Unicode16 编码的字符串，另一种是 ASCII 字符串。Unicode16 字符串是 UEFI 默认使用的字符串。ASCII 字符串以 '\0' 结尾，每个字符占用 1 字节。Unicode16 字符串以 L'\0' 结尾，每个字符占用 2 字节。按 C 语言习惯，以 L 开头的字符串定义为 Unicode16 字符串。例如，L"Hello World" 是 Unicode16 字符串，而 "Hello World" 是 ASCII 字符串。下面分别介绍字符串、字符串操作函数、字符串资源及字符串资源管理。除非有特殊说明，否则本章中的字符串均指 Unicode16 字符串。

11.1.1 字符串函数

凡是操作 ASCII 字符串的函数其名字中都以“Ascii”为前缀，没有“Ascii”前缀的字符串函数都是 Unicode16 字符串函数。例如，StrLen(L"Hello World") 返回 Unicode16 字符串 L"Hello World" 的长度 11，而 ASCII 字符串 "Hello World" 的长度要用函数 AsciiStrLen("Hello

World") 来获得。下面逐一讲述 EDK2 提供的字符串函数。

- StrLen 函数用于返回字符串中字符的个数，字符个数不包含结尾的 NULL 字符。例如，StrLen（L"Hello World"）返回值为 11。
- StrSize 函数用于返回字符串占用的字节数，包含结尾的 NULL 字符。例如，StrSize（L"Hello World"）返回值为 24。
- StrCpy 函数用于将源字符串复制到目的内存中，并返回复制后的目的字符串地址。调用者负责内存的分配与释放。例如，CHAR16 helloworld[32]；StrCpy（helloworld，L"Hello World"）。
- StrnCpy 函数用于将源字符串复制到目的内存中，但至多复制指定的字节数，并返回复制后的目的字符串地址。调用者负责内存的分配与释放。例如，“CHAR16 helloworld[32]；StrnCpy（helloworld，L"Hello World"，5）;”代码执行后，helloworld 数组将包含字符串 L"Hello"。
- StrnCmp 函数用于比较两个字符串，返回第一对不匹配的两个字符之间的大小关系。若两个字符串相同，则返回 0。例如，StrCmp（"Hello World"，"Hello"）返回 1。
- StrnCmp 函数用于比较两个字符串，但至多比较指定个数（Length）的字符，返回第一对不匹配的两个字符之间的大小关系，两个字符串前 Length 个字符相同则返回 0。例如，StrnCmp（"Hello World"，"Hello"，5）返回 0。
- StrnCat 函数用于拼接字符串，即将第二个字符串复制到第一个字符串的末尾。调用者负责内存的管理。例如，“CHAR16 helloworld[32]；StrCpy（helloworld, L"Hello"）; StrCat（helloworld, L"World"）;”执行后，helloworld 数组将包含 L"HelloWorld"。
- StrnCat 函数用于拼接字符串，即将第二个字符串复制到第一个字符串的末尾，但至多复制指定的字符个数。调用者负责内存的管理。例如，“CHAR16 helloworld[32]；StrCpy(helloworld, L"Hello"); StrnCat(helloworld, L"World"，1);”执行后，helloworld 数组将包含 L"HelloW"。
- StrStr 用于在第一个字符串中查找第二个字符串，并返回匹配处的首地址，若没有找到，则返回 NULL。例如，“CHAR16 *helloworld=L"HelloWorld"; StrStr(helloworld, L"lloW")”执行后返回 L"lloWorld"（即 helloworld+3）。StrStr(L"HelloWorld", L"llow") 返回 NULL。
- StrDecimalToUintn 返回十进制数字符串表示的无符号整数（UINTN）。例如，StrDecimalToUintn(L"1f") 返回 1。该函数将忽略字符串中的前导空白和前导 0。
- StrDecimalToUint64 返回十进制数字符串表示的无符号 64 位整数（UINTN64）。例如，StrDecimalToUint64(L"1f") 返回 1。该函数将忽略字符串中的前导空白和前导 0。

- StrHexToUintn 返回十六进制数字符串表示的无符号整数（UINTN）。字符串可以以 0x 开头，也可以以任何十六进制符号开头。例如，StrDecimalToUintn(L"1f") 和 StrDecimalToUintn(L"0x1f") 都返回 0x1F。该函数将忽略字符串中的前导空白和前导 0。
- StrHexToUint64 返回十六进制数字符串表示的无符号 64 位整数（UINTN64）。字符串可以以 0x 开头，也可以以任何十六进制符号开头。例如，StrHexToUintn64(L"0xffffffffffffffff") 返回 0xFFFFFFFFFFFFFFFF。该函数将忽略字符串中的前导空白和前导 0。
- UnicodeStrToAsciiStr 用于将 Unicode16 字符串转换为 ASCII 字符串，转换时 Unicode16 字符的低 8 位复制到 ASCII 字符串。调用者负责内存管理。例如，“CHAR helloworld[32]; UnicodeStrToAsciiStr(L"Hello World", helloworld);”调用后，helloworld 数组将包含字符串 "Hello World"。

操作 ASCII 字符串的函数用法与这些操作 Unicode16 字符串的函数相似，不再赘述，仅简单讲一下 AsciiStrToUnicodeStr 函数。AsciiStrToUnicodeStr 用于将 ASCII 字符串转换为 Unicode16 字符串，转换时将 ASCII 字符复制到 Unicode16 字符的低 8 位，高 8 位填 0。例如，“CHAR16 helloworld[32]; AsciiStrToUnicodeStr ("Hello World", helloworld);”调用后，helloworld 数组将包含字符串 L"Hello World"。

11.1.2　字符串资源

程序开发中经常会用到一些字符串常量，如菜单中的字符串、向用户显示的消息字符串等。这些字符串内容不会改变，并且每个字符串在不同语言下有不同的翻译，对这些字符串最常用的操作是取出并显示。EDK2 提供了一套方案用于管理这些字符串，被管理起来的这些字符串称为字符串资源，这些字符串资源都是 Unicode16 字符串。Unicode 字符几乎囊括了世界上所有语言用到的字符，每种语言都有自己的编码区间。例如，中文、日文、韩语字符的编码区间为 0x4E00 ~ 0x9FFF，基本拉丁语（即英文字符）的编码区间为 0 ~ 0x007F。UEFI 是如何支持多语言以及如何在多种语言之间切换的呢？看一下字符串资源文件我们就大致明白了。

1. 字符串资源文件

代码清单 11-1 是一个字符串资源文件 example.uni，编译时 EDK 会帮我们把它转化为 C 语言能够识别的格式。

代码清单 11-1　字符串资源文件 example.uni

```
/=#
#langdef   en-US "English"
#langdef   fr-FR "Français"
```

```
#langdef   zh-Hant "繁體中文"
#langdef   zh-Hans "简体中文"
#string STR_LANGUAGE_SELECT        #language en-US  "Select Language"
                                   #language fr-FR  "Choisir la Langue"
                                   #language zh-Hant "選擇語言"
                                   #language zh-Hans "选择语言"
```

#langdef 用于声明本字符串资源文件所支持的语言，第一个参数是语言的名字（字符串代码），第二个参数是该语言的可显示名字字符串。代码清单 11-1 的资源文件支持 4 种形式的语言。

#string 用于定义字符串。第一个参数是字符串标识符。其后是不同形式语言中的字符串，每个字符串以 #language 开头。#language 的第一个参数是语言的名字，第二个参数是该语言下的字符串。

编译后，EDK2 会将 .uni 资源文件编译成头文件和 C 源代码。在头文件里面定义了字符串的编号。C 源代码放在 AutoGen.c 文件里，在 AutoGen.c 文件生成名为 exampleStrings 的数组，这个数组中的数据是根据 example.uni 文件生成的字符串包。代码清单 11-2 是编译 example.uni 后生成的头文件 exampleStrDefs.h。在这个头文件里，字符串标识符 STR_LANGUAGE_SELECT 定义为 0x0002。字符串资源文件名加 Strings 后缀组合成了字符串资源包的名字 exampleStrings。

代码清单 11-2　字符串资源头文件 exampleStrDefs.h

```
#ifndef _STRDEFS_6987936E_ED34_44db_AE97_1FA5E4ED2117
#define _STRDEFS_6987936E_ED34_44db_AE97_1FA5E4ED2117
//Unicode 字符串 ID
#define STR_LANGUAGE_SELECT 0x0002
extern unsigned char exampleStrings[];
#define STRING_ARRAY_NAME exampleStrings
#endif
```

编译 example.uni 后生成的字符串包组合 exampleStrings 定义在 AutoGen.c 中，如代码清单 11-3 所示。exampleStrings 定义为一维字符数组，它构成了一个字符串包组合（String Pack），包组合（Pack）将用于生成系统的包列表（Package List）。包列表将在 11.3 节详细介绍，这里简单介绍字符串 Pack 的结构。Pack 的首 4 字节是 Pack 的长度，其后是由包（Package）组成的列表。例如，exampleStrings，其长度是 0x172，其后由 4 个包组成：en-US 字符串包、fr-FR 字符串包、zh-Hans 字符串包和 zh-Hant 字符串包。每个包（Package）由包头和数据组成，不同类型的包会有不同类型的包头，所有包头都以基本包头（EFI_HII_PACKAGE_HEADER）起始。字符串包由 EFI_HII_STRING_PACKAGE_HDR 头和字符串数据构成。包头的前三个字节构成了包的长度，第 4 个字节是包的类型。例如，代码清单 11-3

中的 en-US 字符串包长度为 0x000067，类型为 0x04，其包数据由两个字符串 L"English"（语言可打印名词，通常标识符为 1）和 "Select Language" 组成。

代码清单 11-3　AutoGen.c 中字符串相关代码

```
//Unicode String Pack Definition
unsigned char exampleStrings [] = {
//exampleStrings 的长度 :0x0172
  0x72,  0x01,  0x00,  0x00,
//////////////////////en-US /////////////////////////
//PACKAGE HEADER
  0x67,  0x00,  0x00,  0x04,  0x34,  0x00,  0x00,  0x00,  0x34,  0x00,  0x00,
      0x00,  0x00,  0x00,  0x00,  0x00,
   ...
  0x2D,  0x55,  0x53,  0x00, //语言名称 "-US\0"
//PACKAGE DATA
//0x0001: $PRINTABLE_LANGUAGE_NAME:0x0001
  0x14,  0x45,  0x00,  0x6E,  0x00,  0x67,  0x00,  0x6C,  0x00,  0x69,  0x00,
      0x73,  0x00,  0x68,  0x00,  0x00,
  0x00,
//0x0002: STR_LANGUAGE_SELECT:0x0002
  0x14,  0x53,  0x00,  0x65,  0x00,  0x6C,  0x00,  0x65,  0x00,  0x63,  0x00,
      0x74,  0x00,  0x20,  0x00,  0x4C,
  0x00,  0x61,  0x00,  0x6E,  0x00,  0x67,  0x00,  0x75,  0x00,  0x61,  0x00,
      0x67,  0x00,  0x65,  0x00,  0x00,
  0x00,
  0x00,
//////////////////////fr-FR /////////////////////////
  ......
//////////////////////zh-Hant /////////////////////////
  ......
//////////////////////zh-Hans /////////////////////////
//PACKAGE HEADER
  0x4D,  0x00,  0x00,  0x04,  0x36,  0x00,  0x00,  0x00,  0x36,  0x00,  0x00,
      0x00,  0x00,  0x00,  0x00,  0x00,
  0x00,  0x00,  0x00,  0x00,  0x00,  0x00,  0x00,  0x00,  0x00,  0x00,  0x00,
      0x00,  0x00,  0x00,  0x00,  0x00,
  0x00,  0x00,  0x00,  0x00,  0x00,  0x00,  0x00,  0x00,  0x00,  0x00,  0x00,
      0x00,  0x01,  0x00,  0x7A,  0x68,
  0x2D,  0x48,  0x61,  0x6E,  0x73,  0x00, //"-0ans\0"
//PACKAGE DATA
//0x0001: $PRINTABLE_LANGUAGE_NAME:0x0001
  0x14,  0x80,  0x7B,  0x53,  0x4F,  0x2D,  0x4E,  0x87,  0x65,  0x00,  0x00,
//0x0002: STR_LANGUAGE_SELECT:0x0002
  0x14,  0x09,  0x90,  0xE9,  0x62,  0xED,  0x8B,  0x00,  0x8A,  0x00,  0x00,
  0x00,
}
```

2. 在程序中使用字符串资源

在 AutoGen.c 中，EDK2 为我们生成了字符串包组合 exampleStrings，但这个包组合还不能直接使用（即不能直接从包组合中检索出字符串）。要使用字符串包，需要先将字符串包组合添加到系统 HII[⊖]数据库中，生成 HII 包列表，通过 HiiLib 库提供的 HiiAddPackages 完成这项工作。示例 11-1 演示了如何将字符串包组合 exampleStrings 添加到 HII 数据库中和使用字符串。该示例中，首先为将要生成的 HII 包列表生成一个 GUID，然后调用 HiiAddPackages 注册字符串资源，注册后可以通过 ShellPrintHiiEx 函数向屏幕输出标识符为 STR_LANGUAGE_SELECT 的英文字符串。通常不直接使用 .uni 文件中定义的字符串标识符，而是用宏 STRING_TOKEN 取得标识符。例如，本例中用 STRING_TOKEN(STR_LANGUAGE_SELECT) 取得 STR_LANGUAGE_SELECT 对应的标识符。

【示例 11-1】 使用字符串资源。

```
// Package 列表的 GUID，用于在数据库中识别 Package 列表
EFI_GUID   mStrPackageGuid = { 0xedd31def, 0xf262, 0xc24e, 0xa2, 0xe4, 0xde,
    0xf7, 0xde, 0xcd, 0xcd, 0xee };
// 首先注册字符串资源文件：将字符串包组合加入到 HII 数据库中
EFI_HANDLE HiiHandle = HiiAddPackages (&mStrPackageGuid , ImageHandle,
    exampleStrings, NULL);
// 打印标识符为 STR_LANGUAGE_SELECT 的字符串
ShellPrintHiiEx(-1,-1,(const CHAR8*)"en-US",
STRING_TOKEN (STR_LANGUAGE_SELECT), HiiHandle);
```

HiiAddPackages 函数原型如代码清单 11-4 所示。向系统 HII 数据库注册一组 HII 资源包组合，并返回这组 HII 资源包组合在系统中的 Handle。如果注册失败，则返回 NULL。DeviceHandle 后面的参数是由这组 HII 资源包组合组成的一个列表，列表最后一项为 NULL。包组合的结构前面已经讲过，即由 4 字节的包组合长度和一系列资源包组成。

代码清单 11-4 HiiAddPackages 函数原型

```
// 向系统注册资源包组合
EFI_HII_HANDLE EFIAPI HiiAddPackages (
  IN CONST EFI_GUID *PackageListGuid,    // Package 列表的 GUID
  IN EFI_HANDLE DeviceHandle OPTIONAL,   // 注册的这一组包从属于 DeviceHandle
  ...      // 由一组 HII 包组合组成的列表，列表最后一个元素（即该函数最后一个参数）必须是 NULL
);
```

下面总结一下如何使用字符串资源。

1）首先生成 .uni 文件，在 .uni 文件里定义语言与所支持语言下的字符串。

⊖ HII 是 Human Interface Infrastructure 的简写，主要用于管理人机交互时用到的资源，如字体、字符串、图像、窗体（Form）等。

2）在源程序中通过 HiiAddPackages 注册字符串资源。

3）通过字符串标识符使用对应的字符串。

示例 11-1 展示了使用字符串资源中的字符串的一种方法，即用 ShellPrintHiiEx 将字符串资源中的字符串输出到屏幕。ShellPrintHiiEx 又是如何取得该字符串的呢？ UEFI 中的服务通常是以 Protocol 的形式提供的。ShellPrintHiiEx 是 EDK2 提供的 ShellLib 库中的函数。真正去访问 HII 数据库取得字符串的是字符串 Protocol——EFI_HII_STRING_PROTOCOL。字符串 Protocol 用于管理 UEFI 中的字符串资源。下面就来讲解字符串 Protocol 的用法。

11.1.3　管理字符串资源

UEFI 提供了 EFI_HII_STRING_PROTOCOL 以管理字符串资源。EDK2 提供的 HiiLib 包含了用于管理字符串资源的函数。本节将介绍这两种管理字符串资源的方式。

1. 通过 EFI_HII_STRING_PROTOCOL 管理字符串

.uni 文件里的字符串资源最终会注册到系统 HII 数据库中形成包列表。字符串 Protocol EFI_HII_STRING_PROTOCOL 用于管理 HII 数据库中指定包列表的字符串资源，包括检索、添加、更新字符串以及检索字符串资源包列表所支持的语言。EFI_HII_STRING_PROTOCOL 是一种服务型 Protocol，因而整个系统仅需要一个 EFI_HII_STRING_PROTOCOL 实例。代码清单 11-5 是字符串 Protocol 的结构体。下面逐一介绍字符串 Protocol 的 5 个服务。

代码清单 11-5　EFI_HII_STRING_PROTOCOL 结构体

```
typedef struct _EFI_HII_STRING_PROTOCOL EFI_HII_STRING_PROTOCOL;
struct _EFI_HII_STRING_PROTOCOL {
  EFI_HII_NEW_STRING NewString;                          //添加字符串
  EFI_HII_GET_STRING GetString;                          //检索字符串
  EFI_HII_SET_STRING SetString;                          //更新字符串
  EFI_HII_GET_LANGUAGES GetLanguages;                   //返回指定资源包列表支持的语言
  EFI_HII_GET_2ND_LANGUAGES GetSecondaryLanguages;      //返回指定包列表的次要语言
};
```

（1）NewString 服务：向资源数据库中添加字符串

字符串 Protocol 的 NewString 服务，用于添加字符串到指定的 PackageList 中，并返回该字符串的标识符 StringId。该标识符在这个 PackageList 中是唯一的。代码清单 11-6 是其函数原型。

代码清单 11-6　NewString 函数原型

```
//NewString: 添加字符串到 PackageList 中
typedef EFI_STATUS(EFIAPI *EFI_HII_NEW_STRING)(
```

```
  IN CONST EFI_HII_STRING_PROTOCOL   *This,
  IN EFI_HII_HANDLE PackageList,              //字符串添加到 PackageList 的字符串包中
  OUT EFI_STRING_ID *StringId,                //返回添加的字符串的 ID
  IN CONST CHAR8 *Language,                   //待添加字符串的语言，例如" en-US"
  IN CONST CHAR16 *LanguageName, OPTIONAL     //语言的名字，例如" English"
  IN CONST EFI_STRING String,                 //要添加的字符串，以 NULL 结尾
  IN CONST EFI_FONT_INFO *StringFontInfo OPTIONAL   //指定字符串字体信息
);
```

添加新的字符串资源时，可以同时指定该字符串的字体信息 EFI_FONT_INFO。字体信息是可选参数，如果为 NULL，则使用默认的系统字体、大小和风格。字体信息中包括了字体风格、字符高度和字体名称，其结构体如代码清单 11-7 所示。

代码清单 11-7　EFI_FONT_INFO 结构体

```
//EFI_FONT_INFO  数据结构
typedef struct {
  EFI_HII_FONT_STYLE FontStyle;          //字体风格
  UINT16             FontSize;           //字符高度（以像素为单位）
  CHAR16             FontName[1];        //字体名称
} EFI_FONT_INFO;
```

字体信息（EFI_FONT_INFO）中的字体风格 EFI_HII_FONT_STYLE 是 32 位无符号整数。UEFI 定义了 8 种字体风格，代码清单 11-8 列出了这些字体风格。

代码清单 11-8　系统字体风格

```
//下面定义了系统的字体风格
typedef UINT32  EFI_HII_FONT_STYLE;
#define EFI_HII_FONT_STYLE_NORMAL            0x00000000  //普通字体
#define EFI_HII_FONT_STYLE_BOLD              0x00000001  //粗体
#define EFI_HII_FONT_STYLE_ITALIC            0x00000002  //斜体
#define EFI_HII_FONT_STYLE_EMBOSS            0x00010000  //浮雕
#define EFI_HII_FONT_STYLE_OUTLINE           0x00020000  //轮廓
#define EFI_HII_FONT_STYLE_SHADOW            0x00040000  //阴影
#define EFI_HII_FONT_STYLE_UNDERLINE         0x00080000  //下划线
#define EFI_HII_FONT_STYLE_DBL_UNDER         0x00100000  //双下划线
```

（2）GetString 服务：从资源数据库中检索字符串

字符串 Protocol 的 GetString 服务用于从 PackageList 中找出 ID 为 StringId、语言为 Language 的字符串。检索出的字符串将复制到缓冲区 String 中，它是 EFI_STRING 类型的变量，而 EFI_STRING 是 CHAR16* 类型。*StringSize 作为输入参数，表示缓存区的大小；作为输出参数，表示字符串的长度。如果缓冲区的大小小于检索出的字符串所需的内存大小，则 *StringSize 返回字符串所需内存大小，调用者应重新分配缓冲区后再次调用 GetString 服

务。StringFontInfo 是可选参数，若其不为 NULL，则 *StringFontInfo 返回指向 EFI_FONT_INFO 的指针，调用者负责释放该内存。代码清单 11-9 是 GetString 的函数原型。

代码清单 11-9　GetString 函数原型

```
//GetString: 从 PackageList 中检索字符串
typedef EFI_STATUS(EFIAPI *EFI_HII_GET_STRING)(
  IN CONST EFI_HII_STRING_PROTOCOL *This,
  IN CONST CHAR8 *Language,                  //函数将返回此语言下的字符串
  IN EFI_HII_HANDLE PackageList,             //将在此 Package List 检索字符串
  IN EFI_STRING_ID StringId,                 //待检索字符串的 ID
  OUT EFI_STRING String,                     //检索出的字符串将复制到此缓存中
  IN OUT UINTN *StringSize,
  OUT EFI_FONT_INFO **StringFontInfo OPTIONAL   //若非 NULL，则返回字符串字体信息
);
```

（3）SetString 服务：更新资源数据库中的字符串

字符串 Protocol 的 SetString 服务用于在 PackageList 中更新 ID 为 StringId、语言为 Language 的字符串。StringFontInfo 是可选参数，若非 NULL，则字符串使用此字体；若为 NULL，则字体保持不变。代码清单 11-10 是其函数原型。

代码清单 11-10　SetString 函数原型

```
//SetString: 更新 PackageList 中的字符串
typedef EFI_STATUS(EFIAPI *EFI_HII_SET_STRING)(
  IN CONST  EFI_HII_STRING_PROTOCOL *This,
  IN EFI_HII_HANDLE PackageList,             //包含字符串 StringId 的 Package 列表
  IN EFI_STRING_ID StringId,                 //待更新字符串的 ID
  IN CONST CHAR8 *Language,                  //待更新字符串的语言
  IN EFI_STRING String,                      //将 ID 为 StringId 的字符串更新为此字符串
  IN CONST EFI_FONT_INFO *StringFontInfo OPTIONAL   //字符串的字体信息
);
```

（4）GetLanguages 服务：获得资源包列表所支持的语言

字符串 Protocol 的 GetLanguages 服务用于返回 PackageList 支持的所有语言。Languages 作为返回值，是一个标准字符串，该字符串包含了所有支持的语言的代码，语言代码以分号分隔。代码清单 11-11 是其函数原型。

代码清单 11-11　GetLanguages 函数原型

```
//返回 PackagList 支持的所有语言
typedef EFI_STATUS(EFIAPI *EFI_HII_GET_LANGUAGES)(
  IN CONST EFI_HII_STRING_PROTOCOL *This,
  IN EFI_HII_HANDLE PackageList,             //待查询的 Package 列表
  IN OUT CHAR8 *Languages,                   //返回所有支持的语言
  IN OUT UINTN *LanguagesSize                //输入：缓冲区大小；输出：Languages 字符串字节数
);
```

（5）GetSecondaryLanguages 服务：获得资源包列表所支持的次要语言

每个字符串包列表都有一个主语言，零个或多个次要语言。字符串 Protocol 的 GetSecondaryLanguages 服务用于查询包列表的次要语言。若所查询包列表的次要语言为空，则通过 * SecondaryLanguagesSize 返回 0。代码清单 11-12 是其函数原型。

代码清单 11-12　GetSecondaryLanguages 函数原型

```
//GetSecondaryLanguages 函数返回包列表的次要语言
typedef EFI_STATUS(EFIAPI *EFI_HII_GET_2ND_LANGUAGES)(
  IN CONST EFI_HII_STRING_PROTOCOL *This,
  IN EFI_HII_HANDLE PackageList,          //待查询的 Package 列表
  IN CONST CHAR8 *PrimaryLanguage,        //返回主语言
  IN OUT CHAR8 *SecondaryLanguages,       //返回次要语言
  //作为输入：缓冲区大小；作为输出：字符串 SecondaryLanguages 占用的字节数
  IN OUT UINTN *SecondaryLanguagesSize
);
```

2. 通过 HiiLib 使用字符串 Protocol 的服务

除了直接使用 EFI_HII_STRING_PROTOCOL，也可以利用 HiiLib 提供的函数管理字符串资源。HiiLib 是 EDK2 提供的 HII 操作库，主要是对 Hii Protocol（如字符串 Protocol、字体 Protocol 等）的封装，还包括一些辅助函数。HiiLib 中字符串相关的函数还有 HiiGetString、HiiSetString 及 HiiGetSupportedLanguage。

HiiGetString 从字符串包中根据语言和 ID 取得字符串，如果没有指定语言，则使用系统当前语言。如果使用指定的语言和系统语言均无法取得字符串，则使用 Package 支持语言列表中的第一种语言。其返回值为获取的字符串，内存由 AllocatePool() 分配，调用者负责内存的释放（通过 FreePool()）。HiiGetString 是对字符串 Protocol 中的 GetString 服务的封装。代码清单 11-13 是其函数原型。

代码清单 11-13　HiiGetString 函数原型

```
//从字符串包列表中检索字符串
EFI_STRING EFIAPI HiiGetString (
  IN EFI_HII_HANDLE  HiiHandle,  //HII 数据库中注册的 Package
  IN EFI_STRING_ID   StringId,   //字符串的 ID
  //若 Language 为 NULL，则表示要取得此种语言的字符串。若 Language 为 NULL，则使用系统语言
  IN CONST CHAR8     *Language  OPTIONAL
);
```

HiiSetString 用于建立新的字符串或更新字符串，它包含了 EFI_HII_STRING_PROTOCOL 中的 NewString 和 SetString 这两个服务的功能。

CHAR8* HiiGetSupportedLanguages(EFI_HANDLE HiiHandle) 的功能相当于 EFI_HII_STRING_PROTOCOL 的 GetLanguages 服务的功能，用于获得 Package 列表支持的所有语言。例如，编译代码清单 11-1 所示的字符串资源后会生成的包列表，对该包列表调用 HiiGetSupportedLanguages(HiiHandle)，会返回 ASCII 编码的 LangCode 串 "en-US;fr-FR;zh-Hant;zh-Hans"。

3. 示例

示例 11-2 展示了从字符串包 HiiHandle 中取出英文字符串的两种方法：一种是使用字符串 Protocol 的 GetString 服务；另一种是使用 HiiGetString 函数。

使用 GetString 服务取得字符串时，假设我们明确知道字符串 STR_LANGUAGE_SELECT 小于 64 个字符。通常开发者会明确知道待检索字符串的长度，因而这种假设是可行的。

【示例 11-2】 从字符串资源中检索字符串。

```
CHAR16 StrBuf[64];
UINTN  StrSize= 63;
//方法一：使用字符串 Protocol 的 Get String 服务检索字符串
Status = gHiiString->GetString(gHiiString, "en-us",
    HiiHandle ,STRING_TOKEN(STR_LANGUAGE_SELECT), StrBuf, &StrSize,NULL);
//方法二：使用 HiiGetString 函数取得字符串
CHAR16* Str=HiiGetString(HiiHandle,STRING_TOKEN(STR_LANGUAGE_SELECT),"en-us");
//使用 Str 完毕后释放内存
FreePool(Str);
```

从示例 11-2 可以看出 gHiiString->GetString 与 HiiGetString 的区别。gHiiString->GetString 需要调用者负责内存的分配与释放，HiiGetString 只需调用者负责释放内存。示例 11-2 为了简单清楚地讲述 gHiiString->GetString 的用法，假设调用者明确知道字符串的长度。很多时候调用者是不知道取出的字符串大小的，这时需要通过如下几步获取字符串。

- 调用 gHiiString->GetString 获得字符串长度。
- 根据字符串长度分配内存。
- 再次调用 gHiiString->GetString 获得字符串。
- 字符串使用完毕，释放内存。

示例 11-3 是检索未知长度的字符串的方法。

【示例 11-3】 使用 GetString 服务检索未知长度的字符串。

```
EFI_STATUS TestString(EFI_HANDLE HiiHandle )
{
  EFI_STATUS Status = 0;
```

```
  CHAR8* BestLanguage = "en-US";
  EFI_STRING TempString = NULL;
  UINTN StringSize = 0;
  //步骤一：获得字符串长度
  Status = gHiiString -> GetString(gHiiString,
              BestLanguage, HiiHandle, STRING_TOKEN (STR_LANGUAGE_SELECT),
              TempString, &StringSize, NULL);
  if(Status == EFI_BUFFER_TOO_SMALL){
    //步骤二：分配内存
    TempString = AllocatePool (StringSize);
    //步骤三：取得字符串
    Status = gHiiString -> GetString (gHiiString,
              BestLanguage, HiiHandle, STRING_TOKEN (STR_LANGUAGE_SELECT),
              TempString, &StringSize, NULL);
    Print(L"%s\n", TempString);
    //步骤四：释放内存
    FreePool(TempString);
  }else{
    Print(L"GetString %r\n", Status);
  }
  return 0;
}
```

11.2 管理语言

既然字符串资源支持多语言，程序开发中就少不了同语言“打交道”。例如，系统要提供更改当前语言的能力、语言相关的产品要能根据用户的配置显示最佳语言下的字符串。下面来看一下设置系统语言的方法以及 HiiLib 提供的语言相关的库函数。

1. 系统的当前语言及更改方法

UEFI 系统的当前语言存放在 UEFI 全局系统变量 L"PlatformLang" 中。更改全局变量 L"PlatformLang" 即可更改系统的语言。全局变量 L"PlatformLangCodes" 存放了系统所支持的全部语言。全局系统变量要通过运行时服务提供的 GetVariable 和 SetVariable 服务访问读取和设置。回忆一下第 5.3 节讲述的系统变量相关知识，变量 L"PlatformLang" 在 gEfiGlobalVariableGuid 标识的名字空间内。示例 11-4 展示了读取和设置系统语言的方法。

【示例 11-4】 读取、设置系统语言。

```
//获得系统支持的语言
CHAR8* PlatFormLan = GetEfiGlobalVariable (L"PlatformLangCodes");
//获得系统支持的语言
```

```
CHAR8* SystemLanguage = GetEfiGlobalVariable (L" PlatformLang ");
//设置系统语言
CHAR8* DefaultLang = "zh-Hans";
gRT->SetVariable(L"PlatformLang", &gEfiGlobalVariableGuid,
EFI_VARIABLE_NON_VOLATILE | EFI_VARIABLE_BOOTSERVICE_ACCESS |
EFI_VARIABLE_RUNTIME_ACCESS,
   AsciiStrSize (DefaultLang), DefaultLang);
```

2. 操作语言的辅助函数

EKD2 提供了几个语言相关的函数，这些函数不在 UEFI 规范范围内，是 EDK2 提供的库函数。这些函数包括 HiiGetSupportedLanguages、GetNextLanguage 及 GetBestLanguage。

HiiGetSupportedLanguages 前面讲过，用于获取指定包列表支持的所有语言。该函数返回的通常是以分号分隔的一个字符串，例如 "en-US；fr-FR；zh-Hant；zh-Hans"，通过 GetNextLanguage 函数可以依次获取其中的每一个语言代码。GetNext Language 函数原型如下所示。

```
VOID EFIAPI GetNextLanguage (IN OUT CHAR8 **LangCode,  OUT CHAR8 *Lang );
```

执行上述代码后，*LangCode 指向的语言代码被复制到 Lang 中，*LangCode 指向下一个语言代码。示例 11-5 展示了如何使用 GetNextLanguage 函数。

【示例 11-5】 解析语言字符串。

```
CHAR8 Lang[32];
CHAR8* SupportedLangCode = "en-US;fr-FR;zh-Hant;zh-Hans";
CHAR8** LangCode = &SupportedLangCode;
while(*LangCode != NULL){
  GetNextLanguage(LangCode, Lang);
}
```

While 循环第一遍执行后，LangCode 指向 SupportedLangCode + 7；Lang 赋值为“en-US”。

另一个比较重要的函数是 GetBestLanguage，它从语言支持列表中找出最匹配参数列表的语言代码，参数列表中前面的参数有较高的优先级，若未找到，则返回 NULL。代码清单 11-14 是其函数原型。

代码清单 11-14　GetBestLanguage 函数原型

```
CHAR8 * EFIAPI GetBestLanguage (
  IN CONST CHAR8 *SupportedLanguages,    //此字符串包含了所有支持的语言代码
  IN BOOLEAN Iso639Language,
  ...                                    //以 NULL 结尾的参数列表
);
```

Iso639Language 为 TRUE 时，语言代码为 ISO 639-2 格式；为 FALSE 时，语言代码为 RFC 4646 格式。函数参数列表末尾的“…”为可变参数列表，每个参数指向一个语言代码字符串（这个字符串包含一个或多个语言代码），参数列表以 NULL 结束。

示例 11-6 的代码取自 MdeModulePkg\Library\HiiLib\HiiString.c:HiiGetString，展示了 GetBestLanguage 的用法。在该示例中，SupportedLanguages 是由所有支持的语言组成的字符串（语言代码以分号分隔）。若 Language 指向的语言代码在 SupportedLanguages 中出现，则返回 Language；否则继续匹配 PlatformLanguage，若其在 Platform Languages 中出现，则返回 PlatformLanguage；否则，继续匹配列表中下一个参数，即 SupportedLanguages。

【示例 11-6】 GetBestLanguage 用法示例。

```
BestLanguage = GetBestLanguage ( SupportedLanguages,
  FALSE,                                                  //RFC 4646 mode
  Language,                                               //最高优先级
  PlatformLanguage != NULL ? PlatformLanguage : "",       //中等优先级
  SupportedLanguages,                                     //最低优先级
  NULL
 );
```

11.3 包列表

UEFI 将资源组织在包（也可称为 HII 包）中，例如前面我们讲的字符串资源就被放在 HII 数据库的包中。几个不同的包组织在一起构成包列表（Package List）。包列表由 GUID、包列表大小以及一系列包组成。每个包列表以 EFI_HII_PACKAGE_END 类型的包结束。代码清单 11-15 是包列表的结构体。

代码清单 11-15 包列表结构体

```
typedef struct {
  UINT32 Length:24;              //包含此 Header 在内的整个 Package 的长度
  UINT32 Type:8;                 //包类型
  UINT8 Data[ … ];
} EFI_HII_PACKAGE_HEADER;
typedef struct {
  EFI_GUID PackageListGuid;
  UINT32 PackagLength;           //包含此 Header 在内的整个 Package List 的长度
} EFI_HII_PACKAGE_LIST_HEADER;
```

图 11-1 是一个包列表的示例。这个例子中的包列表包含两个包，第一个包从偏移 0x14 处开始，第二个包从偏移 0x14+Length 处开始。

偏移 \ 偏移	0x0	0x1	0x2	0x3
0x00	PackageListGuid			
0x04				
0x08				
0x0C				
0x10	Package Length			
0x14	Length			Type
…	Data[…]			
0x14+Length	Length			Type
…	Data[…]			
PackageLength–4	4			EFI_HII_PACKAGE_END

图 11-1　包列表示例

UEFI 规范定义的包类型见表 11-1。

表 11-1　HII 包的类型

包类型值	包 ID	包类型说明
EFI_HII_PACKAGE_TYPE_ALL	0x00	伪类型，用于匹配所有类型
EFI_HII_PACKAGE_TYPE_GUID	0x01	厂商定义类型，Data 的格式由 Header 后的 GUID 决定
EFI_HII_PACKAGE_FORMS	0x02	文本型窗口
EFI_HII_PACKAGE_STRINGS	0x04	字符串
EFI_HII_PACKAGE_FONTS	0x05	字体
EFI_HII_PACKAGE_IMAGES	0x06	图像
EFI_HII_PACKAGE_SIMPLE_FONTS	0x07	简单字体，包含窄（8×19）和宽（16×19）两种
EFI_HII_PACKAGE_DEVICE_PATH	0x08	Device Path
EFI_HII_PACKAGE_END	0x09	Package List 结束标志
EFI_HII_PACKAGE_ANIMATIONS	0x0A	动画
EFI_HII_PACKAGE_TYPE_SYSTEM_BEGIN	0xDF	［OXDF,OXEO］区间内的值为系统保留值
…	…	
EFI_HII_PACKAGE_TYPE_SYSTEM_END	0xE0	

11.4　图形界面显示

在文本模式下，我们可以使用 Print 系列的函数显示字符串。在图形模式下，要使用 EFI_GRAPHICS_OUTPUT_PROTOCOL（简称 GraphicsOut）将图像显示到屏幕上。EFI_GRAPHICS_OUTPUT_PROTOCOL 包含三个成员函数和一个成员变量。

- 成员函数 QueryMode，用于查询显示模式。
- 成员函数 SetMode，用于设置显示模式，如设置为文本界面模式。
- 成员函数 Blt，块传输的简称，用于将图像输出到屏幕或从屏幕读取图像。
- 成员变量 Mode，指向了当前的显示模式。

代码清单 11-16 是这个 Protocol 的结构体。

代码清单 11-16　EFI_GRAPHICS_OUTPUT_PROTOCOL 结构体

```
struct _EFI_GRAPHICS_OUTPUT_PROTOCOL {
  EFI_GRAPHICS_OUTPUT_PROTOCOL_QUERY_MODE QueryMode;
  EFI_GRAPHICS_OUTPUT_PROTOCOL_SET_MODE SetMode;
  EFI_GRAPHICS_OUTPUT_PROTOCOL_BLT Blt;
  EFI_GRAPHICS_OUTPUT_PROTOCOL_MODE *Mode;
}
```

大部分系统只有一个显示设备，可以通过 Boot Service 的 LocateProtocol 获得 GraphicsOut 实例，如示例 11-7 所示。当系统有多个显示设备时，可以通过 OpenProtocol 得到各个显示设备的 GraphicsOut 实例。

【示例 11-7】 获取 GraphicsOut 实例。

```
EFI_GRAPHICS_OUTPUT_PROTOCOL* g_GraphicsOutput;
EFI_STATUS Status;
Status = gBS->LocateProtocol(&gEfiGraphicsOutputProtocolGuid,
    NULL, (VOID **)&g_GraphicsOutput);
if (EFI_ERROR(Status)) {
    return Status;
}
```

下面来具体介绍一下图形界面的显示模式，以及如何传输图像和显示字符串。

11.4.1　显示模式

显示模式包括分辨率、颜色深度等。我们可以通过 QueryMode(…) 查询显示模式，通过 SetMode(…) 设置显示模式，通过指针 Mode 读取当前的显示模式。Mode 是指向 EFI_GRAPHICS_OUTPUT_PROTOCOL_MODE 的指针，其结构体如代码清单 11-17 所示。

代码清单 11-17　显示模式结构体

```
typedef struct {
  UINT32 MaxMode;                                  // 显示设备支持的模式数量
  UINT32 Mode;                                     // 当前显示模式
  EFI_GRAPHICS_OUTPUT_MODE_INFORMATION *Info;      // 当前显示模式下的模式信息
  UINTN SizeOfInfo;                                // Info 数据结构的大小
  EFI_PHYSICAL_ADDRESS FrameBufferBase;            // 帧缓冲区物理地址
  UINTN FrameBufferSize;                           // 帧缓冲区大小
} EFI_GRAPHICS_OUTPUT_PROTOCOL_MODE;
```

一般而言，系统已经初始化，系统帧缓冲区的物理地址及大小就不会再发生改变，而其他属性（包括分辨率、像素颜色深度）是可以改变的，因而这些会发生改变的信息（称为“显示模式信息”）放在 Info 指向的内存区域。显示模式信息的结构体如代码清单 11-18 所示。

代码清单 11-18　显示模式信息结构体

```
typedef struct {
  UINT32 Version;                                  //版本号，不同版本必须向后兼容
  UINT32 HorizontalResolution;                     //垂直分辨率
  UINT32 VerticalResolution;                       //水平分辨率
  EFI_GRAPHICS_PIXEL_FORMAT PixelFormat;           //像素格式
  EFI_PIXEL_BITMASK PixelInformation;              //仅当像素格式为 PixelBitMask 时有效
  UINT32 PixelsPerScanLine;                        //每扫描行的像素数
} EFI_GRAPHICS_OUTPUT_MODE_INFORMATION;
```

可以通过如下代码获得屏幕分辨率，也就是帧缓冲区的宽度与高度。

```
UINT32 frameWidth, frameHeight;
frameWidth = (UINT32)m_GraphicsOutput->Mode->Info->HorizontalResolution;
frameHeight = (UINT32)m_GraphicsOutput->Mode->Info->VerticalResolution;
```

显示模式信息的像素格式 EFI_GRAPHICS_PIXEL_FORMAT 共有 4 个类型，它们的定义如代码清单 11-19 所示。

代码清单 11-19　显示模式信息的像素格式枚举值

```
typedef enum {
  PixelRedGreenBlueReserved8BitPerColor,        //RGBA 格式，每个分量为 8bit
  PixelBlueGreenRedReserved8BitPerColor,        //BGRA 格式，每个分量为 8bit
  PixelBitMask,            //使用像素掩码
  PixelBltOnly,            //只能通过 Blt 函数访问帧缓冲区，FrameBufferBase 无效
  PixelFormatMax
} EFI_GRAPHICS_PIXEL_FORMAT;
```

当 PixelFormat 是 PixelBitMask 时，像素的格式由显示模式信息的 PixelInformation 决定。PixelInformation 是 EFI_PIXEL_BITMASK 类型的变量，其结构体如代码清单 11-20 所示。

代码清单 11-20　EFI_PIXEL_BITMASK 结构体

```
typedef struct {
  UINT32 RedMask;
  UINT32 GreenMask;
  UINT32 BlueMask;
  UINT32 ReservedMask;
} EFI_PIXEL_BITMASK;
```

RedMask 共 32bit，其中为 1 的位是红色分量的有效位。例如，RedMask=0x0000007F（对

应的二进制表示为 01111111）表示 32 位中的 0 ~ 6 位表示红色。其他三个分量与之相似。

例如，PixelInformation={0x000000FF, 0x0000FF00, 0x00FF0000, 0xFF000000} 格式为 RGBA，与 PixelRedGreenBlueReserved8BitPerColor 相同。

而像素格式为 PixelBltOnly，则意味着帧缓冲区的线性物理地址无效，我们不能通过向 g_GraphicsOutput->FrameBufferBase 中写内容而改变屏幕。在这种格式下，我们只能用 g_GraphicsOutput->Blt 读写屏幕。

明白了显示模式 EFI_GRAPHICS_OUTPUT_PROTOCOL_MODE，GraphicsOut 的两个成员函数 QueryMode(…) 和 SetMode(…) 就好理解了。QueryMode 的函数原型如代码清单 11-21 所示。

代码清单 11-21　QueryMode 的函数原型

```
typedef EFI_STATUS (EFIAPI *EFI_GRAPHICS_OUTPUT_PROTOCOL_QUERY_MODE) (
  IN EFI_GRAPHICS_OUTPUT_PROTOCOL *This,
  IN UINT32 ModeNumber,
  OUT UINTN *SizeOfInfo,
  OUT EFI_GRAPHICS_OUTPUT_MODE_INFORMATION **Info );
```

QueryMode(…) 用于获得模式 ModeNumber 的模式信息。模式信息由 Info 返回，Info 指向的内存由 QueryMode 负责分配，大小由 SizeOfInfo 返回。Info 的大小之所以由 SizeOfInfo 决定，是因为 EFI_GRAPHICS_OUTPUT_MODE_INFORMATION 在不同的 UEFI 版本中的格式会有所不同，不能由 sizeof(EFI_GRAPHICS_OUTPUT_MODE_INFORMATION) 确定大小。调用者负责释放 Info 指向的内存。示例 11-8 展示了如何使用 QueryMode 查询系统支持的所有显示模式。

【示例 11-8】 查询系统支持的所有显示模式。

```
EFI_GRAPHICS_OUTPUT_MODE_INFORMATION *Info;
UINTN      InfoSize;
UINT32  Mode;
/* 查询显示模式 */
for (UINT32 i = 0; i < GraphicsOutput->Mode->MaxMode; i++) {
  Print(L"Mode %d\n", i);
  Status = GraphicsOutput->QueryMode(GraphicsOutput, i, &InfoSize, &Info);
  if (EFI_ERROR(Status)) {
    Print(L"QueryMode %r\n", Status);
    break;
  } else {
    Print(L"version %x, Width:%d, Height: %d, ScanLine: %d\n",
        Info->Version, Info->HorizontalResolution,
        Info->VerticalResolution,Info->PixelsPerScanLine);
  }
  FreePool(Info);
}
```

SetMode 的函数原型如代码清单 11-22 所示。如果设置系统显示模式采用编号的 ModeNumber 的模式，那么 ModeNumber 取值必须在 [0，This->MaxMode-1]。若指定的模式不存在，则返回 EFI_UNSUPPORTED。

代码清单 11-22　SetMode 函数原型

```
typedef EFI_STATUS (EFIAPI *EFI_GRAPHICS_OUTPUT_PROTOCOL_SET_MODE) (
    IN EFI_GRAPHICS_OUTPUT_PROTOCOL *This,
    IN UINT32 ModeNumber );
```

11.4.2　Block Transfer（Blt）传输图像

GraphicsOut 可以让我们操作显卡中的帧缓冲（FrameBuffer）。其主要操作是通过 GraphicsOut 的 Blt 服务实现的。Blt 函数原型如代码清单 11-23 所示。通过 Blt 可以执行如下 4 种操作。

- 将整个屏幕填充为某个单一颜色。
- 将图像显示到屏幕。
- 将屏幕区域复制到图像。
- 复制屏幕区域到屏幕另一片区域。

Blt 的参数说明参见其函数原型，值得一提的是参数 Delta。Delta 是图像缓冲区 BltBuffer 每行的字节数，若为 0，则 Delta 的大小为 Width* sizeof(EFI_GRAPHICS_OUTPUT_BLT_PIXEL)。

代码清单 11-23　Blt 函数原型

```
typedef struct {
  UINT8 Blue;
  UINT8 Green;
  UINT8 Red;
  UINT8 Reserved;
} EFI_GRAPHICS_OUTPUT_BLT_PIXEL;

typedefenum {
  EfiBltVideoFill,                                  // 用某个 RGBA 颜色填充整个屏幕
  EfiBltVideoToBltBuffer,                           // 屏幕到缓冲区
  EfiBltBufferToVideo,                              // 缓存区到屏幕
  EfiBltVideoToVideo,                               // 屏幕到屏幕
  EfiGraphicsOutputBltOperationMax
} EFI_GRAPHICS_OUTPUT_BLT_OPERATION;

typedef EFI_STATUS(EFIAPI *EFI_GRAPHICS_OUTPUT_PROTOCOL_BLT) (
  IN EFI_GRAPHICS_OUTPUT_PROTOCOL *This,
  IN OUT EFI_GRAPHICS_OUTPUT_BLT_PIXEL *BltBuffer, OPTIONAL    // 图像缓冲区
```

```
  IN EFI_GRAPHICS_OUTPUT_BLT_OPERATION BltOperation,   //进行的操作
  IN UINTN SourceX,                                    //源的 X 坐标
  IN UINTN SourceY,                                    //源的 Y 坐标
  IN UINTN DestinationX,                               //目的缓冲区的 X 坐标
  IN UINTN DestinationY,                               //目的缓冲区的 Y 坐标
  IN UINTN Width,                                      //操作区域的宽度
  IN UINTN Height,                                     //操作区域的高度
  IN UINTN Delta OPTIONAL
);
```

EFI_GRAPHICS_OUTPUT_BLT_PIXEL 定义了像素的格式，在一些老旧的显卡的帧缓冲中，一个像素占 3 字节，而现在的显卡帧缓冲都是 4 字节了。读取 BMP 图像的时候要小心，因为 BMP 中每个像素占 3 字节。从 BMP 文件读取的图像要转换成 EFI_GRAPHICS_OUTPUT_BLT_PIXEL 格式。

EFI_GRAPHICS_OUTPUT_BLT_OPERATION 定义了 GraphicsOut 允许的操作。相关操作有 4 种类型，下面的 4 个示例分别展示了如何使用 Blt 执行这 4 种操作。

例如，将 ImageOutput 指定的内存区域显示到屏幕上，如示例 11-9 所示。

【示例 11-9】 将图像显示到屏幕。

```
extern EFI_GRAPHICS_OUTPUT_PROTOCOL *gGraphicsOutput;
externEFI_GRAPHICS_OUTPUT_BLT_PIXEL *BltBuffer;
Status = InitializeBuffer(&BltBuffer);
if(Status != EFI_SUCCESS){
  return Status;
}
Status = gGraphicsOutput->Blt(gGraphicsOutput,
  BltBuffer,                                  //BltBuffer 中的图像将显示到屏幕上
  EfiBltBufferToVideo,                        //输出到屏幕
  0, 0,                                       //源起始坐标
  0, 0,                                       //目的起始坐标
  1024, 768,                                  //图像大小
  0
);
```

示例 11-10 展示了如何用红色填充屏幕。

【示例 11-10】 用红色填充屏幕。

```
extern EFI_GRAPHICS_OUTPUT_PROTOCOL *gGraphicsOutput;
EFI_GRAPHICS_OUTPUT_BLT_PIXEL BltBuffer[1] = {0,0,255,0};
Status = gGraphicsOutput->Blt(gGraphicsOutput,
  BltBuffer,                                  //BltBuffer 中的图像将显示到屏幕上
  EfiBltVideoFill,                            //用 BltBuffer[0] 填充屏幕
  0, 0,                                       //源起始坐标
  0, 0,                                       //目的起始坐标
  1024, 768,                                  //图像大小
```

```
  0                                                    // 此参数不起作用
);
```

示例 11-11 是将 100 × 100 大小的图像从 [0，0] 复制到 [100，0] 位置。

【示例 11-11】 复制屏幕区域。

```
extern EFI_GRAPHICS_OUTPUT_PROTOCOL*     gGraphicsOutput;
Status = gGraphicsOutput->Blt(gGraphicsOutput,
  0,                                                   // 此参数不起作用
  EfiBltVideoToVideo,                                  // 屏幕到屏幕
  0, 0,                                                // 源起始坐标
  100, 0,                                              // 目的起始坐标
  100, 100,                                            // 图像大小
  0                                                    // 此参数不起作用
);
```

示例 11-12 是将屏幕区域复制到用户缓冲区。

【示例 11-12】 将屏幕区域复制到缓冲区。

```
extern EFI_GRAPHICS_OUTPUT_PROTOCOL*     gGraphicsOutput;
EFI_GRAPHICS_OUTPUT_BLT_PIXEL *BltBuffer;
UINTN BufferSize = 1024*768*sizeof(EFI_GRAPHICS_OUTPUT_BLT_PIXEL );
Status = gBS->AllocatePool( EfiLoaderData, (UINTN)Size, &BltBuffer);
if(Status != EFI_SUCCESS)
  return Status;
Status = gGraphicsOutput->Blt(gGraphicsOutput,
  BltBuffer,                                           // BltBuffer 接收从屏幕复制来的图像
  EfiBltBufferToVideo,                                 // 从屏幕复制到 BltBuffer
  0, 0,                                                // 源起始坐标
  0, 0,                                                // 目的起始坐标
  1024, 768,                                           // 复制区域大小
  1024*sizeof(EFI_GRAPHICS_OUTPUT_BLT_PIXEL)           // BltBuffer 每行字节数
);
```

Blt 用于显示或复制图像，那么如何在图形界面下显示字符串呢？下一节将要回答这个问题。

11.4.3　在图形界面下显示字符串

若要在图形界面下显示字符串，则需要字体（Font）及字体 Protocol 的支持。UEFI 中的 HiiDatabase 有两种类型的字体，这两种字体分别是 EFI_HII_PACKAGE_SIMPLE_FONTS 和 EFI_HII_PACKAGE_FONTS。其中，EFI_HII_PACKAGE_SIMPLE_FONTS 结构简单，但性能较差，适合在显示少量字符时使用；EFI_HII_PACKAGE_FONTS 复杂，但查找字符的性能较好。

如何生成字体将在接下来的几节讲述，本节讲述如何使用字体 Protocol 显示字符。字体

Protocol EFI_HII_FONT_PROTOCOL 可以把字符（串）转化成位图，然后我们可以用 Blt 把字符串位图显示到屏幕上。字体 Protocol 的结构体如代码清单 11-24 所示。

代码清单 11-24　EFI_HII_FONT_PROTOCOL 结构体

```
typedef struct _EFI_HII_FONT_PROTOCOL {
  EFI_HII_STRING_TO_IMAGE StringToImage;          // 将字符串渲染到位图
  EFI_HII_STRING_ID_TO_IMAGE StringIdToImage;     // 将资源中的字符串 StringId 渲染到位图
  EFI_HII_GET_GLYPH GetGlyph;
  EFI_HII_GET_FONT_INFO GetFontInfo;
} EFI_HII_FONT_PROTOCOL;
```

下面介绍一下 StringToImage 是如何使用的，其函数原型如代码清单 11-25 所示。此函数将字符串根据 StringInfo 给定的格式渲染到目的位图 Blt 中，如果参数 Blt 指向屏幕并且 Flags 中包含 EFI_HII_DIRECT_TO_SCREEN，则字符串位图输出到屏幕。

代码清单 11-25　StringToImage 函数原型

```
typedef EFI_STATUS(EFIAPI *EFI_HII_STRING_TO_IMAGE) (
  IN CONST EFI_HII_FONT_PROTOCOL *This,
  IN EFI_HII_OUT_FLAGS Flags,
  IN CONST EFI_STRING String,
  IN CONST EFI_FONT_DISPLAY_INFO *StringInfo OPTIONAL,
  IN OUT EFI_IMAGE_OUTPUT **Blt,                  // 目的位图
  IN UINTN BltX,                                  // 字符串在位图中的起始位置的 X 坐标
  IN UINTN BltY,                                  // 字符串在位图中的起始位置的 Y 坐标
  OUT EFI_HII_ROW_INFO **RowInfoArray OPTIONAL,
  OUT UINTN *RowInfoArraySize OPTIONAL,
  OUT UINTN *ColumnInfoArray OPTIONAL
);
```

StringInfo 是 EFI_FONT_DISPLAY_INFO 类型（结构体定义在代码清单 11-26 中）的变量，是可选参数，包含了前景和背景色，以及字体的大小、名字等。若 StringInfo 为 NULL，则使用系统默认值。EFI_FONT_DISPLAY_INFO 的 FontInfoMask 定义了使用 FontInfo 的方式，通常可以设为 EFI_FONT_INFO_ANY_FONT，这样 UEFI 会首先查找 HiiDatabase 中有没有 FontInfo 指定的 Font，如果没有，则使用 SimpleFont（SimpleFont 将在 11.5 节讲述）。FontInfoMask 是 EFI_FONT_INFO_MASK（定义为 32 为无符号整数）类型的变量，其有效预定义值可参见代码清单 11-26。

代码清单 11-26　EFI_FONT_DISPLAY_INFO 结构体

```
typedef struct _EFI_FONT_DISPLAY_INFO {
  EFI_GRAPHICS_OUTPUT_BLT_PIXEL ForegroundColor;
  EFI_GRAPHICS_OUTPUT_BLT_PIXEL BackgroundColor;
  EFI_FONT_INFO_MASK FontInfoMask;
```

```
  EFI_FONT_INFO FontInfo
} EFI_FONT_DISPLAY_INFO;
typedef struct {
  EFI_HII_FONT_STYLE FontStyle;
  UINT16 FontSize;
  CHAR16 FontName[…];
} EFI_FONT_INFO;
typedef UINT32  EFI_FONT_INFO_MASK;
#define EFI_FONT_INFO_SYS_FONT 0x00000001
#define EFI_FONT_INFO_SYS_SIZE 0x00000002
#define EFI_FONT_INFO_SYS_STYLE 0x00000004
#define EFI_FONT_INFO_SYS_FORE_COLOR 0x00000010
#define EFI_FONT_INFO_SYS_BACK_COLOR 0x00000020
#define EFI_FONT_INFO_RESIZE 0x00001000
#define EFI_FONT_INFO_RESTYLE 0x00002000
#define EFI_FONT_INFO_ANY_FONT 0x00010000
#define EFI_FONT_INFO_ANY_SIZE 0x00020000
#define EFI_FONT_INFO_ANY_STYLE 0x00040000
```

参数 Blt 是 EFI_IMAGE_OUTPUT（参见代码清单 11-27）类型的变量，它定义了目的位图的大小及位图地址。如果目的位图地址为 NULL，则该 StringToImage 服务创建一个新的位图作为目的位图。如果目的位图指向屏幕，则字符串位图输出到屏幕。

代码清单 11-27　EFI_IMAGE_OUTPUT 结构体

```
typedef struct _EFI_IMAGE_OUTPUT {
  UINT16 Width;
  UINT16 Height;
  union {
    EFI_GRAPHICS_OUTPUT_BLT_PIXEL *Bitmap;
    EFI_GRAPHICS_OUTPUT_PROTOCOL *Screen;
  } Image;
} EFI_IMAGE_OUTPUT;
```

示例 11-13 展示了 StringToImage 的用法。在该示例中，OutText 函数将字符串输出到指定的位图；ShowText 函数显示字符串到屏幕。

【示例 11-13】 用字体 Protocol 的 StringToImage 服务输出字符串。

```
static EFI_FONT_DISPLAY_INFO DefaultSimpleFont = {
    {0xFF,0x00,0x00,0x00},              //BGRA 格式前景色
    {0x00,0x00,0x00,0x00},              //BGRA 格式背景色
    EFI_FONT_INFO_ANY_FONT,
    {EFI_HII_FONT_STYLE_NORMAL, 0, L'0'}
};
static EFI_IMAGE_OUTPUT ImageOutput = {1024,768,NULL};
extern EFI_HII_FONT_PROTOCOL *gHiiFont;
void OutText(CHAR16* str, UINTN x, UINTN y)
```

```
{
  EFI_STATUS Status;
  ImageOutput.Image.Bitmap = (EFI_GRAPHICS_OUTPUT_BLT_PIXEL *)malloc(1024*768);
  Status = gHiiFont->StringToImage ( gHiiFont,
            EFI_HII_IGNORE_IF_NO_GLYPH | EFI_HII_OUT_FLAG_CLIP |
            EFI_HII_OUT_FLAG_CLIP_CLEAN_X | EFI_HII_OUT_FLAG_CLIP_CLEAN_Y |
            EFI_HII_IGNORE_LINE_BREAK | EFI_HII_OUT_FLAG_TRANSPARENT,
            (CHAR16*)str, (const EFI_FONT_DISPLAY_INFO*)(&DefaultSimpleFont),
            &ImageOutput,
            (UINTN) x, (UINTN) y,
            NULL, NULL, NULL);
  //使用 ImageOutput 完毕后释放内存
  free(ImageOutput.Image.Bitmap);
}
//输出字符串到屏幕
void ShowText(CHAR16* str, UINTN x, UINTN y)
{
  EFI_STATUS Status;
  //设置执行输出到屏幕的 EFI_IMAGE_OUTPUT
  extern EFI_GRAPHICS_OUTPUT_PROTOCOL* g_GraphicsOutput;
  EFI_IMAGE_OUTPUT Screen = {
    g_GraphicsOutput->Mode->Info->HorizontalResolution,
    g_GraphicsOutput->Mode->Info-> VerticalResolution, NULL};
  Screen.Image.Screen = g_GraphicsOutput;
  //输出到屏幕
  Status = gHiiFont->StringToImage ( gHiiFont,
            EFI_HII_IGNORE_IF_NO_GLYPH | EFI_HII_OUT_FLAG_CLIP |
            EFI_HII_OUT_FLAG_CLIP_CLEAN_X | EFI_HII_OUT_FLAG_CLIP_CLEAN_Y |
            EFI_HII_IGNORE_LINE_BREAK | EFI_HII_OUT_FLAG_TRANSPARENT |
            EFI_HII_DIRECT_TO_SCREEN,
           (CHAR16*)str, (const EFI_FONT_DISPLAY_INFO*)(&DefaultSimpleFont),
            &Screen,
            (UINTN) x, (UINTN) y,
            NULL, NULL, NULL);
}
```

在 ShowText 函数中，StringToImage 将字符串输出到屏幕上，本质上，它先将字符串输出到位图，然后将位图通过 EFI_GRAPHICS_OUTPUT_PROTOCOL 的 Blt 服务输出到屏幕。而 Blt 服务本质上是一种读写帧缓冲的服务，它通过写帧缓冲完成了显示功能。

11.5 用 SimpleFont 显示中文

11.1 节已经讲述了 UEFI 字符串相关知识，UEFI 设计之初就考虑到了对多种语言的支持，UEFI 中的字符串采用 Unicode16 编码，因而可以表示所有的语言。字符串的显示还需要字体的支持。然而 EDK2 仅提供了英语和法语的字体库（或称字库），其他语言的字库需要开发者

来实现。EDK2 在 MdeModulePkg/Universal/Console/GraphicsConsoleDxe/LaffStd.c 文件中定义了英文和法文字库，这个字库中的字体采用 SimpleFont（简单字体）格式。

11.5.1　SimpleFont 格式

SimpleFont 是一种点阵字体，有两种字体格式，一种是窄字符，另一种是宽字符。窄字符用 8（宽度）×19（高度）个 bit 的点阵表示。宽字符用 16×19 个 bit 的点阵表示，前 8×19 个 bit 表示左半部分，后 8×19 个 bit 表示右半部分。每个 bit 表示一个像素（1 表示显示该像素；0 表示不显示）。这两种字符的字体格式如代码清单 11-28 所示。

代码清单 11-28　窄字符和宽字符结构体

```
#define EFI_GLYPH_NON_SPACING                    0x01
#define EFI_GLYPH_WIDE                           0x02
#define EFI_GLYPH_HEIGHT                         19
#define EFI_GLYPH_WIDTH                          8
//窄字符，占 8×19 个像素
typedef struct {
  CHAR16 UnicodeWeight;                          //字符编码 (Unicode16 编码)
  UINT8  Attributes;                             //字符属性
  UINT8  GlyphCol1[EFI_GLYPH_HEIGHT];            //字符位图
} EFI_NARROW_GLYPH;
//宽字符，占 16×19 个像素
typedef struct {
  CHAR16 UnicodeWeight;                          //  字符编码 (Unicode16 编码)
  UINT8  Attributes;                             //字符属性
  UINT8  GlyphCol1[EFI_GLYPH_HEIGHT];            //字符位图左半部
  UINT8  GlyphCol2[EFI_GLYPH_HEIGHT];            //字符位图右半部
  //pad 确保 sizeof (EFI_WIDE_GLYPH) == 2* sizeof (EFI_NARROW_GLYPH)
  UINT8  Pad[3];                                 //必须填充为零
} EFI_WIDE_GLYPH;
```

例如，字符 'A' 为窄字符，其 EFI_NARROW_GLYPH 数据为：{ 0x0041,0x00,{0x00,0x00,0x00,0x10,0x38,0x6C,0xC6,0xC6,0xC6,0xFE,0xC6,0xC6,0xC6,0xC6,0xC6,0x00,0x00,0x00,0x00}}。中文字符 '一' 为宽字符，其 EFI_NARROW_GLYPH 数据为：{ 0x4e00, 0x00, { 0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x7f,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00}, { 0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0xfe,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00}, {0x00, 0x00, 0x00} }。示例 11-14 为窄字符 'A' 和宽字符 '一' 的字体点阵。

【示例 11-14】 窄字符 'A' 和宽字符 '一' 的字体点阵。

```
#define S ((((((((((((((((0
#define x )<<1)+1
#define _ )<<1)+0
```

```
S _ _ _ _ _ _ _ _,
S _ _ _ _ _ _ _ _,
S _ _ _ _ _ _ _ _,
S _ _ _   x _ _ _,
S _ _   x x x _ _,
S _ x x _ x x _ _,
S x x _ _ _ x x _,
S x x _ _ _ x x _,
S x x _ _ _ x x _,
S x x x x x x x _,
S x x _ _ _ x x _,
S x x _ _ _ x x _,
S x x _ _ _ x x _,
S x x _ _ _ x x _,
S x x _ _ _ x x _,
S _ _ _ _ _ _ _ _,
S _ _ _ _ _ _ _ _,
S _ _ _ _ _ _ _ _,
S _ _ _ _ _ _ _ _
字符 'A'
```

```
S _ _ _ _ _ _ _|_ _ _ _ _ _ _ _,
S _ _ _ _ _ _ _|_ _ _ _ _ _ _ _,
S _ _ _ _ _ _ _|_ _ _ _ _ _ _ _,
S _ _ _ _ _ _ _|_ _ _ _ _ _ _ _,
S _ _ _ _ _ _ _|_ _ _ _ _ _ _ _,
S _ _ _ _ _ _ _|_ _ _ _ _ _ _ _,
S _ _ _ _ _ _ _|_ _ _ _ _ _ _ _,
S _ _ _ _ _ _ _|_ _ _ _ _ _ _ _,
S _ _ _ _ _ _ _|_ _ _ _ _ _ _ _,
S _ x x x x x x|x x x x x x x x _,
S _ _ _ _ _ _ _|_ _ _ _ _ _ _ _,
S _ _ _ _ _ _ _|_ _ _ _ _ _ _ _,
S _ _ _ _ _ _ _|_ _ _ _ _ _ _ _,
S _ _ _ _ _ _ _|_ _ _ _ _ _ _ _,
S _ _ _ _ _ _ _|_ _ _ _ _ _ _ _,
S _ _ _ _ _ _ _|_ _ _ _ _ _ _ _,
S _ _ _ _ _ _ _|_ _ _ _ _ _ _ _,
S _ _ _ _ _ _ _|_ _ _ _ _ _ _ _,
S _ _ _ _ _ _ _|_ _ _ _ _ _ _ _,
S _ _ _ _ _ _ _|_ _ _ _ _ _ _ _
字符 '一'
```

字体资源也要注册到 HII 数据库中形成包才能使用。下面来看一下 SimpleFont 包的包头数据结构，代码清单 11-29 列出了 Simple Font 包头的数据结构。

代码清单 11-29　Simple Font 包的包头结构体

```
typedef struct _EFI_HII_SIMPLE_FONT_PACKAGE_HDR {
  EFI_HII_PACKAGE_HEADER Header;
  UINT16 NumberOfNarrowGlyphs;                    // 窄字符个数
  UINT16 NumberOfWideGlyphs;                      // 宽字符个数
  // EFI_NARROW_GLYPH NarrowGlyphs[];
  // EFI_WIDE_GLYPH   WideGlyphs[];
} EFI_HII_SIMPLE_FONT_PACKAGE_HDR;
```

由 EFI_HII_SIMPLE_FONT_PACKAGE_HDR 结构体可以看出，SimpleFont 包中的字符数据是以数组的形式附在 EFI_HII_SIMPLE_FONT_PACKAGE_HDR 后面。每个 SimpleFont 包包含一个 EFI_NARROW_GLYPH 数组及一个 EFI_WIDE_GLYPH 数组。图 11-2 给出了 SimpleFont 包格式。

偏移 \ 偏移	0x0	0x1	0x2	0x3
0x00	包大小			Type（7）
0x04	窄字符个数		宽字符个数	
0x08	字符点阵数组			
⋮				

图 11-2　SimpleFont 包格式

11.5.2　如何生成字体文件

对每个中文字符进行手工编码是一件很费时的事情。如何快速产生 EFI_WIDE_GLYPH 格式的字体数据呢？可以用 JavaScript。因为浏览器可以显示各种语言，所以我们可以将一个字符渲染到 canvas 里，然后分析 canvas，生成 EFI_WIDE_GLYPH 格式的数据。依次处理每一个字符，就可以得到全部数据。其处理流程为：

1）建立画布，取得画布上下文 context。

2）依次处理 Unicode16 编码从 0x4E00 到 0x9FA5 的每一个字符。

3）将字符写到 context 区域。

4）从 context 区域取出 16 × 19 位图。

5）将该位图转化为 EFI_WIDE_GLYPH 格式。

示例 11-15 是取得中文字符 EFI_WIDE_GLYPH 数据的 JavaScript 程序，可生成字符点阵数据。

【示例 11-15】 用 JavaScpirt 生成字符点阵数据。

```
<canvas id="a" width="32" height="32"></canvas>
<script type="text/javascript">
  var a_canvas = document.getElementById("a");
  var context = a_canvas.getContext("2d");
  const FW = 16;                                          //宽字符，宽度为 16 像素
  const FH = 19                                           //宽字符，高度为 19 像素
  context.strokeRect(10, 10, FW, FH);
  context.font = "bold 15px sans-serif";                  //此处可以设置用户想要的字体
  context.fillstyle='#00f';
  var fontstr="<BR>";
  function formatHH(x){                                   //十六进制数，必须为 2 字符
    var l = x.length;
    var s = x;
    if (l == 1)  s = "0" + x;
    return s;
  }
  for(i=0x4e00; i< 0x9fa5;i++){ //依次处理 Unicode16 编码从 0x4E00 到 0x9FA5 的每一个字符
    context.clearRect(10,   10, FW, FH);
    context.fillText(String.fromCharCode(i), 10, 10+FW);      //写字符到画布
    var bitmapimg = context.getImageData(10,10, FW,FH);       //读画布位图
    var bitmap = bitmapimg.data;
    //将表示字符的画布位图转化为 EFI_WIDE_GLYPH 数据
    var left = " ";
    var right = " ";
    for( row =0;row <19; row++){
      var a = 0;
      for(col =0; col <8; col ++){
        a = (bitmap[row * 16 *4 + col*4 +3] >10)?( (a<<1) | 1) : (a<< 1);
```

```
        }
        left = left + "0x" + formatHH(a.toString(16)) ;
        if(row <18) left = left + ",";
        a = 0;
        for(;col<16; col++){
          a = (bitmap[row * 16 *4 + col*4+3] >10)? ((a<<1) | 1) : (a<< 1);
        }
        right = right + "0x" + formatHH(a.toString(16)) ;
        if(row <18) right= right+ ",";
      }
      fontstr = fontstr + "{ 0x" + i.toString(16)
        + ", 0x00, "  +  "{" +  left   +"}, {" + right +"}, {0x00,0x00,0x00} },"
        + "<BR>";
    }
    fontstr = fontstr
        + "{ 0x00, 0x00, { 0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00, "
        + "0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00}} <BR> };"
  document.write(fontstr);
  </script>
```

11.5.3 如何注册字体文件

注册字体文件，即将字体点阵数据注册到 HII 数据库中，生成包列表。回忆一下 11.1 节讲述的注册字符串资源的过程，注册字体文件与其相似，首先生成字体包组合，然后调用 HiiAddPackages 注册到系统内。字体包组合由包组合头、SimpleFont 包头和点阵数组组成。示例 11-16 展示了如何将字体文件（即字体点阵数据）注册到系统内。

【示例 11-16】 注册字体文件。

```
//将字体点阵数据注册到系统内
EFI_STATUS CreatesimpleFontPkg(
  EFI_WIDE_GLYPH* WideGlyph,      //字体点阵数组
  UINT32 SizeInBytes              //点阵数组大小
)
{
  EFI_STATUS Status;
  EFI_HII_SIMPLE_FONT_PACKAGE_HDR *simpleFont;
  UINT8 *Package;
  //生成包组合头及 SimpleFont 包头
  {
    UINT32 packageLen = sizeof (EFI_HII_SIMPLE_FONT_PACKAGE_HDR) +
      SizeInBytes + 4;
    Package = (UINT8*)AllocateZeroPool (packageLen);
    WriteUnaligned32((UINT32 *) Package,packageLen);
    simpleFont = (EFI_HII_SIMPLE_FONT_PACKAGE_HDR *) (Package + 4);
    simpleFont->Header.Length = (UINT32) (packageLen - 4);
    simpleFont->Header.Type = EFI_HII_PACKAGE_SIMPLE_FONTS;
```

```
    simpleFont->NumberOfNarrowGlyphs = 0;
    simpleFont->NumberOfWideGlyphs=
      (UINT16) (SizeInBytes/sizeof (EFI_WIDE_GLYPH));
  }
  //复制字体数据
  {
    UINT8 * Location = (UINT8 *) (&simpleFont->NumberOfWideGlyphs + 1);
    CopyMem (Location, WideGlyph, SizeInBytes);
  }
  //把生成的 Simple Font Package 安装到系统中
  {
    EFI_HII_HANDLE  simpleFontHiiHandle = HiiAddPackages (
        &gSimpleFontPackageListGuid, NULL, Package, NULL);
    if(simpleFontHiiHandle == NULL){
      return -1;
    }
  }
  FreePool (Package);
  return EFI_SUCCESS;
}
```

调用函数 CreatesimpleFontPkg 注册了中文字体后，就可以显示中文字符了。

11.6 开发 SimpleFont 字库程序

为了使制作的字体能够方便使用，我们可以制作单独的字库程序，这样就可以随时注册和卸载我们制作的字体了。字库程序作为一个应用程序，应该根据命令行参数的不同，在程序入口函数内提供注册字库和卸载字库的功能。11.5.3 节中的 CreatesimpleFontPkg 函数可以将字体注册到 UEFI 系统中，下面主要来看如何卸载字库。示例 11-17 中的 UnloadFont 用于卸载字库。

【示例 11-17】 卸载字库程序。

```
EFI_STATUS UnLoadFont()
{
  EFI_STATUS                Status = 0;
  EFI_HII_HANDLE  * handles = 0;
  UINT32 i = 0;
  //根据 GUID 找出包列表
  handles = HiiGetHiiHandles(&gSimpleFontPackageListGuid);
  if(handles == 0){
    return -1;
  }
  while(handles[i] != 0){
    HiiRemovePackages( handles[i++]);          //从 HII 数据库中删除找到的包列表
```

```
    }
    FreePool(handles);
    return Status;
  }
  extern EFI_WIDE_GLYPH  gSimpleFontWideGlyphData[];
  extern UINT32 gSimpleFontBytes;
  static EFI_GUID gSimpleFontPackageListGuid = {0xf6643673, 0x6006, 0x3738, {0x5c,
0xcd, 0xda, 0x1a, 0xeb, 0x3b, 0x26, 0xa2}};
  int main( int argc, char** argv)
  {
     EFI_STATUS Status = 0;
     if(argc == 1) {
      //注册字体
      EFI_HII_HANDLE  * handles = 0;
      handles = HiiGetHiiHandles(&gSimpleFontPackageListGuid);
       //如果没有注册过此 SimpleFont，则注册
      if(handles == 0)
      Status = CreatesimpleFontPkg(gSimpleFontWideGlyphData, gSimpleFontBytes);
      else
      FreePool(handles);
    }else if(argv[1][0]== '-'&& (argv[1][1] == 'u' || argv[1][1] == 'U')){
      //卸载字体
      Status = UnLoadFont();
      if(EFI_SUCESS(Status)
        Print(L" 字体加载成功 ");
      else
        Print(L"Failed to load font: %r\n", Status);//字体注册失败
    }
  }
```

上面例子中用到了 HiiLib 里的两个函数：HiiGetHiiHandles 与 HiiRemovePackages。这两个函数的函数原型如代码清单 11-30 所示。

代码清单 11-30　HiiGetHiiHandles 和 HiiRemovePackages 函数原型

```
//根据 Package 的 GUID 获得 Package 的 Handle，如果 Package 不存在，则返回 NULL
EFI_HII_HANDLE *EFIAPI HiiGetHiiHandles (
  IN CONST EFI_GUID  *PackageListGuid  OPTIONAL
  );
//从系统中删除由 HiiHandle 指定的 Package
VOID EFIAPI HiiRemovePackages ( IN  EFI_HII_HANDLE   HiiHandle  );
```

11.7　字体 Font

11.5 节讲述了 SimpleFont，它的优点是格式简单，制作和使用容易，但缺点也很明显，一是性能较差，渲染任一个字符都需要在 SimpleFontPackage 中从头依次检索；二是功能有

限，只有 Narrow（8 × 19）和 Wide（16 × 19）两种格式的字符。

为了获得更好的性能或者渲染复杂的字符，UEFI 提供了 Font 格式的字体。

11.7.1　Font 的格式

同 SimpleFont 相比，Font 提供了更加复杂和灵活的点阵格式。它不再限制字符点阵大小，同时增加了边距（OffsetX 和 OffsetY）及步长。代码清单 11-31 是 Font 点阵信息 EFI_HII_GLYPH_INFO 的结构体。

代码清单 11-31　EFI_HII_GLYPH_INFO 的结构体

```
typedef struct _EFI_HII_GLYPH_INFO {
  UINT16 Width;          // 字符宽度（像素个数）
  UINT16 Height;         // 字符高度（像素个数）
  INT16 OffsetX;         // 水平边距
  INT16 OffsetY;         // 垂直边距
  INT16 AdvanceX;        // 步长
} EFI_HII_GLYPH_INFO;
```

通过图 11-3 中的两个字符可以明确 EFI_HII_GLYPH_INFO 的含义。内框（实线框）表示字符的宽度和高度，外框（虚线框）的宽度表示 AdanceX，内外框之间的距离为 OffsetX 和 OffsetY。

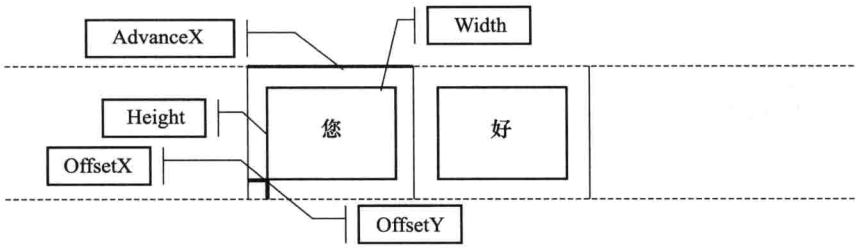


图 11-3　Font 点阵信息示例

11.7.2　字体包的格式

相对于 SimpleFont 的点阵，Font 的点阵更加灵活，能支持更加复杂的字体。Font 包相对于 SimpleFont 包，在更加复杂的同时，也带来了更好的性能。SimpleFont 点阵列表没有索引，Font 则采用带索引性质的点阵块列表。

Font 包由两部分组成，首先是 Font 包头 EFI_HII_FONT_PACKAGE_HDR，然后是点阵块（Glyph Block）列表。在 Font 包头中，定义了该包中默认的字体信息。点阵块列表中的每个点阵块定义了一个连续区域内字符的字体信息和点阵数据，因为其连续性，所以访问点阵块中的某个字符时就可以用访问数组的方式，从而加快了速度。EFI_HII_FONT_PACKAGE_

HDR 的结构体如代码清单 11-32 所示。

代码清单 11-32 EFI_HII_FONT_PACKAGE_HDR 结构体

```
typedef struct _EFI_HII_FONT_PACKAGE_HDR {
  EFI_HII_PACKAGE_HEADER Header;
  UINT32 HdrSize;
  UINT32 GlyphBlockOffset;              //包中首个字符的 Unicode16 码
  EFI_HII_GLYPH_INFO Cell;              //字体信息，定义了字符大小、边距等
  EFI_HII_FONT_STYLE   FontStyle;       //字体风格，如是否斜体等
  CHAR16          FontFamily[1];        //字体名字
} EFI_HII_FONT_PACKAGE_HDR;
```

图 11-4 展示了 Font 包的结构。

偏移 \ 偏移	0x0	0x1	0x2	0x3
0x00	Font 包大小			Type（5）
0x04	EFI_HII_FONT_PACKAGE_HDR 头大小			
0x08	Glyph 块的偏移字节数			
0x0C	Cell.Width		Cell.Height	
0x10	Cell.OffsetX		Cell.OffsetY	
0x14	Cell.AdvanceX		FontStyle	
0x18	FontStyle			
0x1C	FontFamily[1];			
0x20				
GlyphBlockOffset	GlyphBlock 列表			
⋮				

图 11-4 Font 包结构

接下来看点阵块（GLYPH Blocks）列表，这个列表由一个或多个 EFI_HII_GLYPH_*_BLOCK 按顺序（字符 Unicode 编码）排列而成，每个块都继承自 EFI_HII_GLYPH_BLOCK。

```
typedef struct _EFI_HII_GLYPH_BLOCK {
  UINT8          BlockType;
} EFI_HII_GLYPH_BLOCK;
```

下面以 Time New Rome 字体来解释点阵块的结构。我们将在该字体中包含 Unicode 编码为 0 ~ 255 的可打印字符，以及 0x4E00 ~ 0x9FA4 的中文字符。

首先，Font 包头的 GlyphBlockOffset 将定义为 0，让该字体库从字符 0 开始定义。

字符 0 ~ 31 是不可打印字符，没有点阵数据，这个区域用 EFI_HII_GIBT_SKIP2_BLOCK 块表示。

字符 32 ~ 255 是字母字符，每个字符点阵数据较小，因而用不同于默认点阵信息的点阵表示。这个区域的字体用 EFI_HII_GIBT_GLYPHS_BLOCK 表示。

字符 0x100 ~ 0x 4DFF 是我们不需要关心的字符（0 × 100 是 ASCII 后的第一个字符，0 × 4DFF 是中文字符前的最后一个字符），这个区域用 EFI_HII_GIBT_SKIP2_BLOCK 块表示。

字符 0x4E00 ~ 0x9FA4 包含了中文字符，这部分字符采用默认的点阵信息，点阵信息定义在包头中。这个区域用 EFI_HII_GIBT_GLYPHS_DEFAULT_BLOCK 表示。

最后的是结束块，用 EFI_GLYPH_GIBT_END_BLOCK 表示。

图 11-5 是这个包中点阵块列表的结构。代码清单 11-33 列出了这几种常用点阵块的结构体。

图 11-5　点阵块类别示例

代码清单 11-33　常用点阵块结构体

```
typedef struct _EFI_GLYPH_GIBT_END_BLOCK {
  EFI_HII_GLYPH_BLOCK  Header;
} EFI_GLYPH_GIBT_END_BLOCK;
typedef struct _EFI_HII_GIBT_GLYPHS_BLOCK {
  EFI_HII_GLYPH_BLOCK  Header;
  EFI_HII_GLYPH_INFO   Cell;
  UINT16          Count;
  UINT8           BitmapData[1];
} EFI_HII_GIBT_GLYPHS_BLOCK;
typedef struct _EFI_HII_GIBT_GLYPHS_DEFAULT_BLOCK {
  EFI_HII_GLYPH_BLOCK  Header;
  UINT16          Count;
  UINT8           BitmapData[1];
} EFI_HII_GIBT_GLYPHS_DEFAULT_BLOCK;
typedef struct _EFI_HII_GIBT_SKIP2_BLOCK {
  EFI_HII_GLYPH_BLOCK  Header;
  UINT16          SkipCount;
} EFI_HII_GIBT_SKIP2_BLOCK;
```

11.7.3　为什么 Font 性能高于 SimpleFont

对比 SimpleFont 和 Font 的数据结构可以看出，要想检索某个字符的 GLYPH 数据，对于 SimpleFont，我们需要依次取出 Simple Package 中的每个 EFI_NARROW_GLYPH 或 EFI_

WIDE_GLYPH，与待检索字符的 Unicode 码做比较，直到取出的 GLYPH 的 Unicode 码与待检索字符的 Unicode 码相同；对于 Font Package 而言，只需依次查询 EFI_HII_GIBT_*_BLOCK，这些 EFI_HII_GIBT_*_BLOCK 相当于为 Package 的字符设置了索引。代码清单 11-34 是在 SimpleFont 包中查找字符的代码。代码清单 11-35 是在 Font 包中查找字符的代码。这两处代码均位于 MdeModulePkg/Universal/HiiDatabaseDxe/Font.c 文件中。

代码清单 11-34　在 SimpleFont 中查找字符

```
HeaderSize = sizeof (EFI_HII_SIMPLE_FONT_PACKAGE_HDR);
for (Link = Private->DatabaseList.ForwardLink;
   Link != &Private->DatabaseList;
   Link = Link->ForwardLink) {
  //遍历 DatabaseList 每一个元素
  Node = CR (Link, HII_DATABASE_RECORD, DatabaseEntry,
    HII_DATABASE_RECORD_SIGNATURE);
  for (Link1 = Node->PackageList->SimpleFontPkgHdr.ForwardLink;
       Link1 != &Node->PackageList->SimpleFontPkgHdr;
       Link1 = Link1->ForwardLink) {
    //遍历 SimpleFontPackage 中的每一个 SimpleFont
    SimpleFont = CR (Link1, HII_SIMPLE_FONT_PACKAGE_INSTANCE, SimpleFontEntry,
      HII_S_FONT_PACKAGE_SIGNATURE);
    //在 narrow glyph 数组中查找 Char。首先获得 narrowglyph 数组的首地址 NarrowPtr
    NarrowPtr = (EFI_NARROW_GLYPH *) ((UINT8 *) (SimpleFont->SimpleFontPkgHdr)
       + HeaderSize);
    //按顺序依次比较数组 NarrowPtr 中的每个字符，直到找到待寻找 Char 的位置
    for (Index = 0;
      Index < SimpleFont->SimpleFontPkgHdr->NumberOfNarrowGlyphs;
      Index++) {
      CopyMem (&Narrow, NarrowPtr + Index,sizeof (EFI_NARROW_GLYPH));
      if (Narrow.UnicodeWeight == Char) {
      *GlyphBuffer = (UINT8 *) AllocateZeroPool (EFI_GLYPH_HEIGHT);
      if (*GlyphBuffer == NULL) {
        return EFI_OUT_OF_RESOURCES;
      }
      Cell->Width  = EFI_GLYPH_WIDTH;
      Cell->Height   = EFI_GLYPH_HEIGHT;
      Cell->AdvanceX = Cell->Width;
      CopyMem (*GlyphBuffer, Narrow.GlyphCol1, Cell->Height);
      if (Attributes != NULL) {
        *Attributes = (UINT8) (Narrow.Attributes | NARROW_GLYPH);
      }
      return EFI_SUCCESS;
      }
    }
    //在 wide glyph 数组中查找 Char。首先计算 wide glyph 数组的首地址 widePtr
    WidePtr = (EFI_WIDE_GLYPH *) (NarrowPtr +
      SimpleFont->SimpleFontPkgHdr->NumberOfNarrowGlyphs);
    for (Index = 0;
```

```
        Index < SimpleFont->SimpleFontPkgHdr->NumberOfWideGlyphs; Index++) {
        ...
        // 与在 narrow glyph (首地址为 NarrowPtr) 数组中查找 Char 相似
}
```

代码清单 11-35 在 Font 包中查找字符

```
  BlockPtr  = FontPackage->GlyphBlock;
  CharCurrent = 1;
  BufferLen   = 0;
  // 以此检索 Font 包中的每个块，每个块相当于整个包的索引
  while (*BlockPtr != EFI_HII_GIBT_END) {
  switch (*BlockPtr) {
  case EFI_HII_GIBT_SKIP2: // 略过此块中的字符
    CopyMem (&Length16, BlockPtr + sizeof (EFI_HII_GLYPH_BLOCK), sizeof (UINT16));
    CharCurrent = (UINT16) (CharCurrent + Length16);
    BlockPtr  += sizeof (EFI_HII_GIBT_SKIP2_BLOCK);
    break;
case EFI_HII_GIBT_GLYPHS_DEFAULT:
    CopyMem (&Length16, BlockPtr + sizeof (EFI_HII_GLYPH_BLOCK), sizeof (UINT16));
    Status = GetCell (CharCurrent, &FontPackage->GlyphInfoList, &DefaultCell);
    if (EFI_ERROR (Status)) {
      return Status;
    }
    BufferLen = BITMAP_LEN_1_BIT (DefaultCell.Width, DefaultCell.Height);
    BlockPtr += sizeof (EFI_HII_GIBT_GLYPHS_DEFAULT_BLOCK) - sizeof (UINT8);
// BlockPtr 指向 GLYPHS 数组，
// 如果 CharValue 落在本块内，此处应该可以根据 Index 直接访问数组 BlockPtr 以提高效率
    for (Index = 0; Index < Length16; Index++) {
      if (CharCurrent + Index == CharValue) {
        return WriteOutputParam (BlockPtr, BufferLen,&DefaultCell,
             GlyphBuffer, Cell, GlyphBufferLen);
      }
      BlockPtr += BufferLen;
    }
    CharCurrent = (UINT16) (CharCurrent + Length16);
    break;
case EFI_HII_GIBT_GLYPHS:
    BlockPtr += sizeof (EFI_HII_GLYPH_BLOCK);
    CopyMem (&Glyphs.Cell, BlockPtr, sizeof (EFI_HII_GLYPH_INFO));
    BlockPtr += sizeof (EFI_HII_GLYPH_INFO);
    CopyMem (&Glyphs.Count, BlockPtr, sizeof (UINT16));
    BlockPtr += sizeof (UINT16); // BlockPtr 指向 GLYPHS 数组
    if (CharValue == (CHAR16) (-1)) {
      if (BaseLine < Glyphs.Cell.Height + Glyphs.Cell.OffsetY) {
        BaseLine = (UINT16) (Glyphs.Cell.Height + Glyphs.Cell.OffsetY);
      }
      if (MinOffsetY > Glyphs.Cell.OffsetY) {
        MinOffsetY = Glyphs.Cell.OffsetY;
```

```
        }
      }
  BufferLen = BITMAP_LEN_1_BIT (Glyphs.Cell.Width, Glyphs.Cell.Height);
  // 如果 CharValue 落在本块内，此处应该可以根据 Index 直接访问数组 BlockPtr 以提高效率
      for (Index = 0; Index < Glyphs.Count; Index++) {
        if (CharCurrent + Index == CharValue) {
          return WriteOutputParam ( BlockPtr, BufferLen,&Glyphs.Cell,
               GlyphBuffer, Cell, GlyphBufferLen);
        }
        BlockPtr += BufferLen;
      }
      CharCurrent = (UINT16) (CharCurrent + Glyphs.Count);
      break;
  case EFI_HII_GIBT_GLYPH:
    ...
  case EFI_HII_GIBT_SKIP1:
    ...
  default:
      ASSERT (FALSE);
      break;
    }
```

11.8 本章小结

本章主要介绍了图形界面开发的基础知识：字符串（多种语言）、字体，以及如何输出图形图像。

（1）字符串（多种语言）

本章介绍了字符串资源的使用方法，包括 .uni 文件的格式、字符串 Protocol 的使用、字符串包的格式与使用。

（2）字体

本章介绍了 SimpleFont 和 Font 两种字体和字体包的格式。TianoCore 只实现了英文和法文字体，若要显示中文，则要制作中文字体。本章还介绍了窄字符和宽字符的格式，并介绍了一种使用 JavaScript 快速生成中文字体的方式。

（3）如何输出图形图像

本章介绍了如何利用 EFI_GRAPHICS_OUTPUT_PROTOCOL 操作显卡中的 FrameBuffer，从而显示图像到显示设备中，还介绍了如何利用字符串 Protocol 将字符串显示到图形界面上。

字符串、字体与 GraphicsOut 构成了 GUI 的基础，下一章我们讲述如何实现简单的 GUI 界面。

第 12 章 Chapter 12

GUI 应用程序

本章以播放器为例介绍如何开发 GUI 程序。读者可以回想一下播放器 GUI，它一般包括一个播放 / 暂停按钮、一个进度条。

GUI 是基于事件驱动的，开发 GUI 之前，要弄清楚三个问题：一是 UEFI 下如何捕获 GUI 事件；二是控件如何响应事件；三是如何显示 GUI 控件。

12.1 UEFI 事件处理

第 6 章我们讲述了事件的使用方法。事件处理是 GUI 开发的核心。下面先来回顾一下启动服务提供的事件处理相关函数。

- CreateEvent：用于生成一个事件对象。
- CloseEvent：用于关闭事件对象。
- SignalEvent：用于触发事件对象。
- WaitForEvent：用于等待事件数组中的任一事件被触发。
- CheckEvent：用于检查 Event 状态。
- SetTimer：用于设置定时器事件属性。

在与用户交互的过程中，键盘事件、鼠标事件和定时器事件是其中比较重要的三个事件。

12.1.1 键盘事件

用户按键方式分为两种，一种是单个按键，另一种是组合键。下面分别讲述如何处理这

两种按键方式。

1. 处理单个按键

如果只是处理单个按键，则可以使用 EFI_SIMPLE_TEXT_INPUT_PROTOCOL（简称 ConIn），其结构体如代码清单 12-1 所示。它包含了两个成员函数：Reset 和 ReadKeyStroke，以及一个成员变量 WaitForKey。

WaitForKey 是一个普通类型的事件，在有按键时触发。事实上，UEFI 内核为键盘设备生成了一个定时器，每隔一段时间会检查键盘设备，如果发现有按键，则会触发 WaitForEvent 事件。

ReadKeyStroke 用于读取键盘设备的下一个击键。读取击键后会重置 WaitForEvent 事件。如果键盘设备没有击键，则该函数返回 EFI_NOT_READY。

代码清单 12-1　EFI_SIMPLE_TEXT_INPUT_PROTOCOL 结构体

```
struct _EFI_SIMPLE_TEXT_INPUT_PROTOCOL {
  EFI_INPUT_RESET Reset;                    //重置设备
  EFI_INPUT_READ_KEY ReadKeyStroke;         //读键盘
  EFI_EVENT WaitForKey;                     //按键时触发此事件
};
//ReadKeyStroke 函数原型
typedef EFI_STATUS(EFIAPI *EFI_INPUT_READ_KEY)(
  IN EFI_SIMPLE_TEXT_INPUT_PROTOCOL *This,
  OUT EFI_INPUT_KEY *Key                    //返回按键，用户负责管理（分配和释放）此处内存
  );
```

得到系统 EFI_SIMPLE_TEXT_INPUT_PROTOCOL 的实例有两种方法，一种是使用 BS 的 OpenProtocol/LocateProtocal/HandleProtocol 服务打开该 Protocol；另一种是直接使用系统表提供的该 Protocol 实例 ConIn。第二种方法是较常用的方法。下面的代码展示了如何等待按键事件。

```
UINTN Index;
Status = gBS->WaitForEvent(1, &gST->ConIn->WaitForKey, &Index);
```

下面的代码展示了如何读取按键。

```
EFI_INPUT_KEY Key;
Status = gST->ConIn->ReadKeyStroke (gST->ConIn, &Key);
```

读取按键一般应遵循以下这两个步骤，首先用 WaitForEvent 服务等待按键的发生，然后用 ReadKeyStroke 读取这个按键。得到的 Key 是 EFI_INPUT_KEY 类型的变量。EFI_INPUT_KEY 结构体如代码清单 12-2 所示。

代码清单 12-2　EFI_INPUT_KEY 结构体

```
typedef struct {
  UINT16 ScanCode;                    // 不可打印字符按键使用 ScanCode
  CHAR16 UnicodeChar;                 // 可打印字符按键使用 UnicodeChar
} EFI_INPUT_KEY;
```

对于可打印字符按键，ScanCode 为零，使用 UnicodeChar；对于不可打印字符按键，UnicodeChar 为零，使用 ScanCode。例如，<a> 键为 {0x00，0x61}；<F1> 键为 {0x0B，0x00}。

代码清单 12-3 列出了常用按键的扫描码。

代码清单 12-3　常用按键的扫描码

```
#define SCAN_NULL              0x0000
#define SCAN_UP                0x0001
#define SCAN_DOWN              0x0002
#define SCAN_RIGHT             0x0003
#define SCAN_LEFT              0x0004
#define SCAN_HOME              0x0005
#define SCAN_END               0x0006
#define SCAN_INSERT            0x0007
#define SCAN_DELETE            0x0008
#define SCAN_PAGE_UP           0x0009
#define SCAN_PAGE_DOWN         0x000A
#define SCAN_F1                0x000B
#define SCAN_F2                0x000C
#define SCAN_F3                0x000D
#define SCAN_F4                0x000E
#define SCAN_F5                0x000F
#define SCAN_F6                0x0010
#define SCAN_F7                0x0011
#define SCAN_F8                0x0012
#define SCAN_F9                0x0013
#define SCAN_F10               0x0014
#define SCAN_ESC               0x0017
```

代码清单 12-4 列出了常用控制字符（回车键、制表键、退格键）的 Unicode 码。

代码清单 12-4　常用控制字符的 Unicode 码

```
#define CHAR_NULL              0x0000
#define CHAR_BACKSPACE         0x0008
#define CHAR_TAB               0x0009
#define CHAR_LINEFEED          0x000A
#define CHAR_CARRIAGE_RETURN   0x000D
```

2. 处理组合键

从 EFI_INPUT_KEY 可以看出，EFI_SIMPLE_TEXT_INPUT_PROTOCOL 只能处理单个按键，对组合键（如 <Ctrl+C>）就无能为力了。要处理组合键，需用 EFI_SIMPLE_TEXT_INPUT_EX_PROTOCOL，其结构体如代码清单 12-5 所示。

代码清单 12-5　EFI_SIMPLE_TEXT_INPUT_EX_PROTOCOL 结构体

```
struct _EFI_SIMPLE_TEXT_INPUT_EX_PROTOCOL{
  EFI_INPUT_RESET_EX Reset;                                  //重置设备
  EFI_INPUT_READ_KEY_EX ReadKeyStrokeEx;                     //读键盘
  EFI_EVENT WaitForKeyEx;                                    //按键时触发此事件
  EFI_SET_STATE SetState;                                    //设置状态，如 <Num> 键
  EFI_REGISTER_KEYSTROKE_NOTIFY RegisterKeyNotify;           //注册热键
  EFI_UNREGISTER_KEYSTROKE_NOTIFY UnregisterKeyNotify;       //注销热键
};
```

（1）EFI_KEY_DATA 结构体

与 EFI_SIMPLE_TEXT_INPUT_PROTOCOL 相比，EFI_SIMPLE_TEXT_INPUT_EX_PROTOCOL 多了三个函数，即 SetState、RegisterKeyNotify 和 UnregisterKeyNotify，并且用于表示按键的结构体也更加复杂。先来看表示按键的结构体 EFI_KEY_DATA，如代码清单 12-6 所示。

代码清单 12-6　EFI_KEY_DATA 结构体

```
typedef struct {
  EFI_INPUT_KEY Key;                                  //按键的编码（Unicode 码或 ScanCode 码）
  EFI_KEY_STATE KeyState;                             //按键时的键盘状态
} EFI_KEY_DATA;
typedef struct _EFI_KEY_STATE {
  //最高位为 1 时有效，Shift、Ctrl、Windows 徽标键、菜单徽标键、SysRq 键是否按下
  UINT32 KeyShiftState;
  //最高位为 1 时有效，< Num Lock> < Caps Lock> 等开关的状态
  EFI_KEY_TOGGLE_STATE  KeyToggleState;
} EFI_KEY_STATE;
```

EFI_KEY_DATA 包含了按键的编码 EFI_INPUT_KEY，以及按键时键盘的状态 EFI_KEY_STATE。通过 EFI_KEY_STATE，我们可以知道按键的同时有哪些特殊按键被同时按下，以及 <Num Lock>、<Caps Lock> 键功能是否打开。

KeyShiftState 最高位为 1 时，KeyShiftState 有效，其中某个位为 1 表示对应的按键被按下。通过 KeyShiftState 可以检测的按键包括：左右 Shift 键、左右 Ctrl 键、左右 Alt 键、左右 Windows 徽标键、菜单徽标键以及拷屏键（<SysRq> 键）。示例 12-1 展示了如何判断左 <Ctrl> 键是否按下。代码清单 12-7 定义了 KeyShiftState 各个位对应的按键。

【示例 12-1】 判断左 <Ctrl> 键是否按下。

```
EFI_KEY_STATE Key;
…// 读取键盘，具体方法将在下文讲述
if( (Key.KeyState.KeyShiftState & EFI_SHIFT_STATE_VALID )
&&(Key.KeyState.KeyShiftState & EFI_LEFT_CONTROL_PRESSED)){
    // 左 <Ctrl> 键按下
}
```

代码清单 12-7　KeyShiftState 各个位对应的按键

```
#define EFI_SHIFT_STATE_VALID           0x80000000
#define EFI_RIGHT_SHIFT_PRESSED         0x00000001
#define EFI_LEFT_SHIFT_PRESSED          0x00000002
#define EFI_RIGHT_CONTROL_PRESSED       0x00000004
#define EFI_LEFT_CONTROL_PRESSED        0x00000008
#define EFI_RIGHT_ALT_PRESSED           0x00000010
#define EFI_LEFT_ALT_PRESSED            0x00000020
#define EFI_RIGHT_LOGO_PRESSED          0x00000040
#define EFI_LEFT_LOGO_PRESSED           0x00000080
#define EFI_MENU_KEY_PRESSED            0x00000100
#define EFI_SYS_REQ_PRESSED             0x00000200
```

KeyToggleState 最高位为 1 时，KeyToggleState 有效，某个位为 1 表示对应的按键处于打开状态。通过 KeyToggleState 可检测的状态包括：Caps Lock 状态、Num Lock 状态、Scroll Lock 状态以及是否允许不完整按键。所谓不完整按键即 EFI_KEY_DATA 中 Key 为 {0，0} 的按键，例如长按 <Ctrl> 键但不按其他任何键时出现的按键状况。示例 12-2 展示了如何判断 Caps Lock 的状态。代码清单 12-8 列出了 KeyToggleState 中各个位表示的状态。

【示例 12-2】 判断 Caps Lock 的状态。

```
EFI_KEY_STATE Key;
...     // 读取键盘
if( (Key.KeyState.KeyToggleState & EFI_TOGGLE_STATE_VALID)
&&(Key.KeyState.KeyToggleState & EFI_CAPS_LOCK_ACTIVE)){
// Caps Lock 键功能被打开
}
```

代码清单 12-8　KeyToggleState 中各个位表示的状态

```
// Toggle state
#define EFI_TOGGLE_STATE_VALID          0x80
#define EFI_KEY_STATE_EXPOSED           0x40
#define EFI_SCROLL_LOCK_ACTIVE          0x01
#define EFI_NUM_LOCK_ACTIVE             0x02
#define EFI_CAPS_LOCK_ACTIVE            0x04
```

（2）读取按键

读键盘函数 ReadKeyStrokeEx 与 EFI_SIMPLE_TEXT_INPUT_PROTOCOL 的 ReadKeyStroke

服务相似，不同之处仅仅在于得到的按键为 EFI_KEY_DATA 类型。ReadKeyStrokeEx 函数原型如代码清单 12-9 所示。

代码清单 12-9 ReadKeyStrokeEx 函数原型

```
typedef EFI_STATUS(EFIAPI *EFI_INPUT_READ_KEY_EX)(
  IN  EFI_SIMPLE_TEXT_INPUT_EX_PROTOCOL *This,
  OUT EFI_KEY_DATA *KeyData  //调用者负责该缓冲区的分配与释放
);
```

ReadKeyStrokeEx 返回键盘设备输入队列中下一个按键，如果键盘设备输入队列为空，则返回 EFI_NOT_READY。如果按键为 Shift+ 可打印字符，则返回修正后的按键（例如，Shift+f，返回值 KeyData.Key.Unicode 为 'F'），KeyData.KeyShiftState 的 Shift 对应的位为 0。

示例 12-3 展示了如何利用 EFI_SIMPLE_TEXT_INPUT_EX_PROTOCOL 读取按键。该示例首先利用 BS 的 LocateProtocol 服务得到 EFI_SIMPLE_TEXT_INPUT_EX_PROTOCOL 实例，然后的步骤与读取单个按键相似，即先等待按键事件 WaitForKeyEx，再用 ReadKeyStrokeEx 读取按键。

【示例 12-3】 利用 EFI_SIMPLE_TEXT_INPUT_EX_PROTOCOL 读取按键。

```
EFI_SIMPLE_TEXT_INPUT_EX_PROTOCOL* InputEx = NULL;
UINTN Index;
EFI_KEY_DATA key = {0};
Status = gBS->LocateProtocol(&gEfiSimpleTextInputExProtocolGuid,
   NULL, (VOID**)&InputEx);
gBS->WaitForEvent(1, &(InputEx->WaitForKeyEx), &Index);
Status = InputEx->ReadKeyStrokeEx(InputEx, &key);
```

（3）热键

EFI_SIMPLE_TEXT_INPUT_EX_PROTOCOL 还提供了热键功能。RegisterKeyNotify 用于注册热键，通过该服务注册的热键回调函数将在用户按下热键后由系统自动执行，其函数原型如代码清单 12-10 所示。注册成功后，系统为这个热键分配一个 NotifyHandle，这个 Handle 可以用来访问这个热键。

代码清单 12-10 RegisterKeyNotify 函数原型

```
typedef EFI_STATUS(EFIAPI *EFI_KEY_NOTIFY_FUNCTION)(IN EFI_KEY_DATA *KeyData);
typedef EFI_STATUS(EFIAPI *EFI_REGISTER_KEYSTROKE_NOTIFY)(
  IN  EFI_SIMPLE_TEXT_INPUT_EX_PROTOCOL *This,
  IN  EFI_KEY_DATA *KeyData,
  IN  EFI_KEY_NOTIFY_FUNCTION KeyNotificationFunction,
  OUT VOID **NotifyHandle
);
```

调用该函数注册热键 KeyData 后，该热键按下时将触发回调函数 KeyNotificationFunction。

热键使用完毕后，需调用 UnregisterKeyNotify 注销热键，其函数原型如代码清单 12-11 所示。NotifyHandle 是注册该热键时分配的句柄。

代码清单 12-11　UnregisterKeyNotify 函数原型

```
typedef EFI_STATUS(EFIAPI *EFI_UNREGISTER_KEYSTROKE_NOTIFY)(
  IN EFI_SIMPLE_TEXT_INPUT_EX_PROTOCOL  *This,
  IN VOID *NotifyHandle
);
```

示例 12-4 展示了如何使用热键 <Ctrl+C>。当用户按下 <Ctrl+C> 热键后，将会在屏幕上打印字符串“Hot Key”。

【示例 12-4】 使用热键 <Ctrl+C>。

```
#include <Uefi.h>
#include <Protocol/SimpleTextInEx.h>
EFI_HANDLE notifyHandle;
EFI_STATUS myNotify(EFI_KEY_DATA *key)
{
  Print((CONST CHAR16*)L"Hot Key\n");
  return 0;
}
EFI_STATUS testHotKey()
{
  EFI_STATUS  Status;
  EFI_KEY_DATA hotkey={0};
  EFI_KEY_DATA key = {0};
  EFI_SIMPLE_TEXT_INPUT_EX_PROTOCOL* InputEx = NULL;
  Status = gBS->LocateProtocol(&gEfiSimpleTextInputExProtocolGuid,
        NULL,   (VOID**)&InputEx);
  Print(L"%r\n", Status);
  hotkey.Key.ScanCode = 0;
  hotkey.Key.UnicodeChar = 'c';
  hotkey.KeyState.KeyShiftState = EFI_LEFT_CONTROL_PRESSED |
    EFI_SHIFT_STATE_VALID;
  hotkey.KeyState.KeyToggleState = EFI_TOGGLE_STATE_VALID;
  //注册热键
  Status = InputEx->RegisterKeyNotify(InputEx, &hotkey, myNotify,
                    (VOID**)&notifyHandle);
  Print(L"%r\n", Status);
  //等待按 < Q > 键退出
  while( key.Key.UnicodeChar != 'q'){
    UINTN Index;
    gBS->WaitForEvent(1, &(InputEx->WaitForKeyEx), &Index);
    Status = InputEx->ReadKeyStrokeEx(InputEx, &key);
    if(key.Key.UnicodeChar == 'q')
      break;
  }
```

```
  //程序退出前一定要注销热键
  Status = InputEx->UnregisterKeyNotify(InputEx, notifyHandle);
  return Status;
}
```

提示：如果 notifyHandle 定义在应用程序中，那么在程序退出前必须调用 UnregisterKeyNotify 注销该热键，否则程序退出后会造成系统崩溃。

12.1.2 鼠标事件

UEFI 提供了 EFI_SIMPLE_POINTER_PROTOCOL 用于获取鼠标事件，其结构体如代码清单 12-12 所示。它包括两个成员函数：Reset、GetState，以及两个成员变量：WaitForInput、Mode。

- Reset 用于重置鼠标设备。
- GetState 用于获取鼠标状态。
- WaitForInput 是鼠标事件，当鼠标设备状态变化时触发。
- Mode 是指向鼠标设备属性的指针。

代码清单 12-12 EFI_SIMPLE_POINTER_PROTOCOL 结构体

```
typedef struct_EFI_SIMPLE_POINTER_PROTOCOL{
  EFI_SIMPLE_POINTER_RESET Reset;
  EFI_SIMPLE_POINTER_GET_STATE GetState;
  EFI_EVENT WaitForInput;
  EFI_SIMPLE_INPUT_MODE *Mode;
} EFI_SIMPLE_POINTER_PROTOCOL;
```

通过 Mode 可以获得鼠标设备的属性。鼠标设备属性是 EFI_SIMPLE_POINTER_MODE 类型的变量。其结构体如代码清单 12-13 所示。

代码清单 12-13 EFI_SIMPLE_POINTER_MODE 的结构体

```
typedef struct {
  UINT64 ResolutionX;          //X 分辨率，0 表示不支持 X 轴
  UINT64 ResolutionY;          //Y 分辨率，0 表示不支持 Y 轴
  UINT64 ResolutionZ;          //Z 分辨率，0 表示不支持 Z 轴
  BOOLEAN LeftButton;          //是否支持左键
  BOOLEAN RightButton;         //是否支持右键
} EFI_SIMPLE_POINTER_MODE;
```

GetState 函数用于获取鼠标当前状态，其函数原型如代码清单 12-14 所示。鼠标状态是 EFI_SIMPLE_POINTER_STATE 类型的变量，其结构体如代码清单 12-14 所示。鼠标状态包含了 X、Y、Z 三个方向的位移以及左键和右键是否按下。这里需要特别注意，鼠标状态中 X、Y、Z 仅仅记录了鼠标的（从上次位置到当前位置的）位移，鼠标的位置要由上层应用程序管理和维护。

代码清单 12-14　GetState 函数原型及 EFI_SIMPLE_POINTER_STATE 结构体

```
typedef EFI_STATUS(EFIAPI *EFI_SIMPLE_POINTER_GET_STATE)
  IN EFI_SIMPLE_POINTER_PROTOCOL *This,
  IN OUT EFI_SIMPLE_POINTER_STATE *State                    //输出鼠标当前状态
);
//EFI_SIMPLE_POINTER_STATE
typedef struct {
  INT32 RelativeMovementX;                                  //X 方向位移
  INT32 RelativeMovementY;
  INT32 RelativeMovementZ;
  BOOLEAN LeftButton;                                       //左键是否按下
  BOOLEAN RightButton;                                      //右键是否按下
} EFI_SIMPLE_POINTER_STATE;
```

在第 6 章最后我们给出了一个示例用于展示如何使用事件，该示例中演示了 EFI_SIMPLE_POINTER_PROTOCOL 的用法，读者可以回忆一下。通过 GetState 服务获得鼠标状态 State 后，我们还要根据鼠标状态计算出鼠标在屏幕上的位移。通过如下公式 State.RelativeMovementX * MOUSE_RESOLUTION_PIXS/(mouse->Mode->ResolutionX) 可以计算出鼠标的水平位移（以像素为单位）。MOUSE_RESOLUTION_PIXS 为调节因子，可以调节鼠标移动的灵敏度，其值可由开发者定义。其他三个方向的位移可由类似的公式获得。

12.1.3　定时器事件

定时器可分为以下三大类：

- 不带 Notification 函数，其事件类型为 EVT_TIMER。
- 带 Notification 函数，并且 Notification 在定时器到期时执行，其类型为 EVT_TIMER|EVT_NOTIFY_SIGNAL。
- 带 Notification 函数，并且 Notification 在定时器等待时执行，其类型为 EVT_TIMER|EVT_NOTIFY_WAIT。

本章我们会用到第一类定时器。生成定时器事件需要以下两步：第一步通过 BS 的 CreateEvent 服务生成一个类型为 EVT_TIMER 的定时器事件；第二步通过 BS 的 SetTimer 服务设置触发时间。代码清单 12-15 列出了 CreateEvent 服务和 SetTimer 服务的函数原型，示例 12-5 展示了如何生成和使用定时器。

代码清单 12-15　BS 的 CreateEvent 服务和 SetTimer 服务函数原型

```
typedef EFI_STATUS CreateEvent (
  IN UINT32 Type,
  IN EFI_TPL NotifyTpl,
  IN EFI_EVENT_NOTIFY NotifyFunction, OPTIONAL
  IN VOID *NotifyContext, OPTIONAL
```

```
  OUT EFI_EVENT *Event
);
typedef EFI_STATUS SetTimer (
  IN EFI_EVENT Event,
  IN EFI_TIMER_DELAY Type,
  IN UINT64 TriggerTime                                    //以 100ns 为单位
);
```

【示例 12-5】 生成和使用定时器。

```
EFI_EVENT TimerEvent;
EFI_STATUS Status = gBS-> CreateEvent ( EVT_TIMER,  TPL_APPLICATION, (EFI_EVENT_
    NOTIFY)NULL, (VOID*)NULL, &TimerEvent);
//每 1/24s, TimerEvent 触发
Status = gBS->SetTimer(TimerEvent, TimerPeriodic , 10 * 1000 * 1000/24);
while(1){
   Status = gBS->WaitForEvent(1, &TimerEvent, &index);
   //定时器到期

}
```

12.1.4 UI 事件服务类

12.1.1 ~ 12.1.3 节分别讲述了键盘、鼠标、定时器事件。为了方便程序处理这些事件，可以将这些事件组织起来，供 UI 程序统一管理使用。代码清单 12-16 是 UIEvents 类声明，它封装了键盘、鼠标及定时器事件。第 6 章介绍事件的时候讲过，BS 的 WaitForEvent 服务会从前至后依次检查事件数组中的事件状态，发现有事件处于触发态时返回事件在数组中的位置。因而在数组中靠前的事件具有较高的优先级，即如果有多个事件处于触发态，在数组最前面的首先返回。在 UIEvents 类中，键盘事件（UI_KEY）在最前面，其次为鼠标事件（UI_MOUSE），最后为定时器事件（UI_TIMER）。UIEvents 有两个核心函数：Init 和 Wait。Init 主要用于初始化事件数组中的三个事件；Wait 用于等待事件的发生。UIEvents 还提供了读取键盘、读写鼠标位置及控制定时器的服务。

代码清单 12-16 UIEvents 类声明

```
class UIEvents {
public:
  //UI_TIMER 处于最低优先级，优先响应键盘和鼠标事件
  enum{UI_KEY = 0, UI_MOUSE = 1, UI_TIMER };
  static const UINTN MOUSE_RESOLUTION_PIXS = 10;        //用于调节鼠标灵敏度
public:
  UIEvents(){}
  ~UIEvents(){ gBS->CloseEvent(mTimerEvent); mTimerEvent = 0;}
  void Init();                                          //初始化
  UINTN Wait();                                         //等待事件
```

```
    //键盘相关服务
    EFI_STATUS ReadKey(EFI_KEY_DATA* key);
    EFI_STATUS ReadKey(EFI_INPUT_KEY* key);
    //鼠标相关服务
    UMouseEvent* GetMouse();
    UMouseEvent GetMousePos(){return mMouseCurPos;}
    UMouseEvent GetMouseLastPos(){return mMouseLastPos;}
    void SetMousePos(UINT16 x, UINT16 y){ mMouseCurPos.X =x; mMouseCurPos.Y =y;}
    //定时器相关服务
    void SetTimer(UINT64 t = 1 * 1000 * 1000/24) {
          gBS->SetTimer(mTimerEvent,TimerPeriodic , t);
    }
    void HandleTimer() { if(mTimerHandler)mTimerHandler(mTimerEvent, NULL);}
  private:
    EFI_STATUS ReadKey();
  private:
    EFI_EVENT WaitArray[3];
    EFI_EVENT mTimerEvent;
    EFI_EVENT_NOTIFY mTimerHandler;
    EFI_SIMPLE_POINTER_PROTOCOL* mMouse;
    UMouseEvent mMouseLastPos, UMouseEvent mMouseCurPos;
    EFI_SIMPLE_TEXT_INPUT_EX_PROTOCOL* mInputEx;
    EFI_KEY_DATA   mKeyData;
  };
```

UIEvents 的 Wait 函数用于等待键盘、鼠标或定时器事件的发生，返回值为 UI_KEY、UI_MOUSE 或 UI_TIMER，表示触发的事件的类型，其实现如代码清单 12-17 所示。

代码清单 12-17　UIEvents 的 Wait 函数实现

```
  inline UINTN UIEvents::Wait(){
     UINTN Index;
     gBS->WaitForEvent(3, WaitArray, &Index);
     return Index;
  }
```

UIEvents 的 Init 函数用于初始化该类的实例，主要是初始化事件数组 WaitArray 中的三大事件。初始化键盘事件时，首先通过 BS 的 LocateProtocol 找出 EFI_SIMPLE_TEXT_INPUT_EX_PROTOCOL 的实例 mInputEx，从而获得 mInputEx 的键盘事件 WaitForKeyEx。鼠标事件的初始化与键盘事件相似，首先找出 EFI_SIMPLE_POINTER_PROTOCOL 的实例 mMouse，然后获得其鼠标事件 WaitForInput。定时器事件则需要通过 BS 的 CreateEvent 生成，最后程序退出时要调用 CloseEvent 关闭这个定时器事件。Init 的具体实现可参见代码清单 12-18。

代码清单 12-18 UIEvents 的 Init 函数实现

```
inline void UIEvents::Init(){
  EFI_STATUS Status;
  memset(this, 0, sizeof(UIEvents));
  Status = gBS->CreateEvent(EVT_TIMER, TPL_APPLICATION, (EFI_EVENT_NOTIFY)NULL ,
      (VOID*)NULL, &mTimerEvent);
  Status = gBS->SetTimer(mTimerEvent,TimerPeriodic , 1 * 1000 * 1000/24);
  WaitArray[UI_TIMER] = mTimerEvent;
  Status = gBS->LocateProtocol( &gEfiSimplePointerProtocolGuid,
     NULL, (VOID**)&mMouse);
  Status = gBS->LocateProtocol(&gEfiSimpleTextInputExProtocolGuid,
     NULL, (VOID**)&mInputEx );
  if(mInputEx){
    WaitArray[UI_KEY] = mInputEx->WaitForKeyEx;
  }else{
    WaitArray[UI_KEY] = gST->ConIn->WaitForKey;
  }
  if(mMouse){
    WaitArray[UI_MOUSE] = mMouse->WaitForInput;
 }
  mMouseCurPos.X =  mMouseCurPos.Y =  mMouseLastPos.X =  mMouseLastPos.Y = 0;
}
```

UIEvents 的 GetMouse 函数会返回鼠标的当前坐标及动作，其实现如代码清单 12-19 所示。

代码清单 12-19 UIEvents 的 GetMouse 函数实现

```
UMouseEvent* UIEvents::GetMouse(){
  EFI_STATUS Status;
  EFI_SIMPLE_POINTER_STATE tMouseState;
  Status = mMouse->GetState(mMouse, &tMouseState);      //取得鼠标位移及动作
  if(!EFI_ERROR(Status)){
    mMouseLastPos = mMouseCurPos;
    UINT64 mMouseResolutionX = mMouse->Mode->ResolutionX;
    UINT64 mMouseResolutionY = mMouse->Mode->ResolutionY;
if(mMouseResolutionX != 0 && mMouseResolutionY != 0){
  //计算鼠标位置
     mMouseCurPos.X = mMouseCurPos.X + (UINT16)
(tMouseState.RelativeMovementX*MOUSE_RESOLUTION_PIXS/mMouseResolutionX);
     mMouseCurPos.Y = mMouseCurPos.Y + (UINT16)
(tMouseState.RelativeMovementY*MOUSE_RESOLUTION_PIXS/mMouseResolutionY);
    }
    ...                                                   //修正鼠标位置，以使其在屏幕之内
    if(tMouseState.LeftButton == 1)
     mMouseCurPos.A = UM_LBUTTONDOWN;
    else if (tMouseState.RightButton == 1)
     mMouseCurPos.A = UM_RBUTTONDOWN;
```

```
    else
     mMouseCurPos.A = 0;
    return &mMouseCurPos;
  }
  return NULL;
}
```

UIEvents 的 ReadKey 函数用于读取键盘的输入，其实现如代码清单 12-20 所示。

代码清单 12-20　UIEvents 的 ReadKey 函数

```
inline EFI_STATUS UIEvents::ReadKey()
{
  EFI_STATUS Status;
  if(mInputEx){
    Status = mInputEx->ReadKeyStrokeEx(mInputEx, &mKeyData);
  }else{
    Status = gST->ConIn->ReadKeyStroke (gST->ConIn, &mKeyData.Key);
  }
  return Status;
}
```

事件驱动的核心是在循环中不断侦听事件并响应事件。例如，Windows 应用程序的消息处理循环。

```
while (GetMessage(&msg, NULL, 0, 0)) { ... }
```

有了 UIEvents 帮助处理 UI 事件，我们就可以实现这个事件处理循环了。

12.2　事件处理框架

事件处理框架包括两大部分，一是事件的侦听，二是事件的响应（或派遣）。事件的侦听和派遣都在事件处理循环中。事件处理循环是事件处理框架的核心部分，也是事件驱动的核心，任何一个 GUI 系统都有这样一个循环。如果用户曾开发过 Win32 的窗口应用程序，一定会对代码清单 12-21 中的代码不陌生。

代码清单 12-21　Win32 窗口应用程序事件处理循环

```
while(GetMessage( &msg, hWnd, 0, 0 ))
{
   TranslateMessage(&msg);
   DispatchMessage(&msg);
}
```

每个 Win32 窗口应用程序都有一个事件队列，这个循环不断从消息队列中取出消息并派

遣消息。在 iOS（iPhone OS）的窗口系统中，这个循环是通过 NSRunLoop 对象完成的。下面我们就来设计和实现这样一个循环。

1. 事件处理循环

我们的 GUI 程序只响应 3 种事件，即鼠标、键盘和定时器事件，控件的绘制和更新在事件处理循环中实现而不是作为事件被派遣。事件循环的循环体流程如下：

- 重绘需要更新的控件，重绘鼠标。
- 等待事件发生。
- 派遣事件至活动控件，并由活动控件响应该事件。
- 如果当前窗口有事件处理函数，则派遣事件至窗口的事件处理函数中。

代码清单 12-22 是事件处理循环的实现。

代码清单 12-22　事件处理循环

```
void Brick::Run()
{
  EFI_KEY_DATA CurKeyData = {0, 0};
  EFI_INPUT_KEY& CurChar = CurKeyData.Key;
  UINTN Index = 0;
  //初始化屏幕
  Brick::UpdateFullScreen();
  Brick::DrawMouse();
  Brick::ReDrawScreen();
  WPARAM wParam = 0;
  LPARAM lParam = 0;
  //1）开始事件处理循环
  while(Brick::Running == true){
    //绘制到后缓冲
    //首先，擦除（移动前）鼠标
    Brick::EraseMouse();
    //其次，绘制 UI 控件
    ((Window*)(Brick::CurWindow))->DrawTo(Brick::FrameBuffer, 0);
    //最后，绘制（移动后）鼠标
    Brick::DrawMouse();
    //2）将后缓冲中的内容更新到屏幕
    Brick::UpdateScreen();
    //3）等待事件发生
    Index = pUiEvent->Wait();
    if(Index == UIEvents::UI_TIMER){                    //派遣定时器事件
      pUiEvent->HandleTimer();
    }else if(Index == UIEvents::UI_MOUSE){              //派遣鼠标事件，选择目标并派发
      Brick::DispatchMouseMessage( *(pUiEvent->GetMouse()));
    }else{                                              //派遣键盘事件
      EFI_STATUS Status;
```

```
        Status = pUiEvent->ReadKey(&CurKeyData);
        *( EFI_INPUT_KEY*)&wParam = CurChar ;
        if(HotKeys::ExeHotKey(CurKeyData) == HotKeys::HOTKEY_QUIT_PROPAGATE){
                                                    //处理热键
        }else if(CurChar.UnicodeChar ==0){          //控制键
          if(Brick::ActiveBrick) Brick::ActiveBrick->OnExtKey(CurChar.ScanCode);
        }else if(CurChar.UnicodeChar == CHAR_CR){   //回车
          if(Brick::ActiveBrick) Brick::ActiveBrick->OnEnter();
        }else if(CurChar.UnicodeChar == CHAR_TAB){  //Tab 键
          if(Brick::ActiveBrick) Brick::ActiveBrick->OnTab();
        }else{                                      //普通字符
          if(Brick::ActiveBrick) Brick::ActiveBrick->OnKey(CurChar.UnicodeChar);
        }
      }
      if(((Window*)(Brick::CurWindow))->mEventHandler)
  ((Window*)(Brick::CurWindow))->mEventHandler((UINT32)Index,wParam,lParam);
    } //结束 while(Running)
  }
```

在我们的播放器示例中，定时器事件用于以影片要求的帧率显示每一帧。键盘事件主要用于快捷键（播放、暂停、停止）和控制键（<Tab>、<Enter>）。鼠标用于控制播放按钮。从事件循环中得到事件后就要派遣事件并由目标控件响应该事件。在介绍事件的派遣之前，首先要设计控件系统。播放器用到了播放按钮、进度条及窗口。图 12-1 展示了控件之间的继承关系。

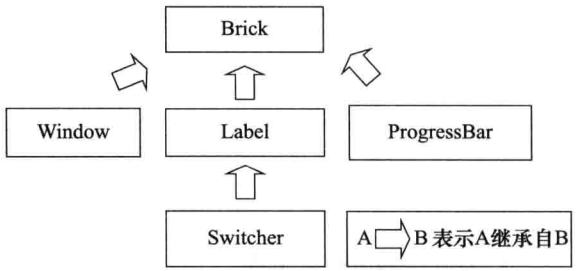


图 12-1　控件继承关系图

Brick 类是所有控件的基类，其类声明如代码清单 12-23 所示。其中，OnXXX 表示事件响应函数；testHit 函数用于测试鼠标是否落于该控件内；DrawTo 用于绘制控件。类静态变量 ActiveBrick 保存了当前活动空间，类静态变量 TabList 是 Tab 列表，类静态变量 BrickList 是控件列表。有了这些变量和函数，事件处理循环就可以派遣事件了。

代码清单 12-23　控件基类 Brick

```
class Brick
{
public:
  typedef int (*EventHandler)(Brick* O, void* context);
public:                                          //事件响应函数
  virtual void OnEnter();
  virtual void OnTab();
  virtual void OnKey(CHAR16 CurChar);
```

```
    virtual void OnExtKey(UINT16 CurChar);
    virtual void OnClick();
    virtual bool testHit(UINT16 X, UINT16 Y);
  public:
    virtual void LoseFocus();
    virtual void SetFocus();
    virtual bool IsFocused();
    //将控件绘制到指定的帧缓冲区
    virtual void DrawTo(void* pFrameBuffer, UINT32 ToScreen) ;
    void reDraw(){DrawTo(FrameBuffer,0);}          //重绘该控件
    //判断该控件是否发生了改变（若发生了改变，则需要重绘）
    bool IsChanged(){return m_IsChanged == 1;}
  public:                                          //辅助函数
    static void DrawMouse();
    static void EraseMouse();
    static void DispatchMouseMessage(MPOINT MouseState);
  public:
    static Brick* ActiveBrick;                     //当前活动控件
    static Brick* CurWindow;                       //当前窗口
    static Brick* TabList;                         //当前 Tab 列表
    static Brick* BrickList;                       //所有控件的列表
    static void* FrameBuffer;                      //系统帧缓冲区
    static UIEvents gUiEvent;
    ...
  public:
    Brick* NextinTabList;                          //Tab 链
    Brick* NextinBrickList;                        //控件列表链
  ...
  };
```

2. 派遣和响应键盘事件

当键盘事件发生时，会由当前活动控件响应按键。

按键可以分为以下 5 类。

1）热键：例如，本例中用空格键控制播放与暂停。在事件处理循环中，每当键盘事件发生，首先由函数 ExeHotKey（EFI_KEY_DATA aKey）检查是否发生了热键。如果有热键，ExeHotKey 会执行相应的处理函数。此处的热键不同于 EFI_SIMPLE_TEXT_INPUT_EX_PROTOCOL 提供的热键，EFI_SIMPLE_TEXT_INPUT_EX_PROTOCOL 提供的热键处理函数执行在 TPL_NOTIFY 级别，而我们 GUI 中的热键只需要执行在 TPL_APPLICATION 级别，因而 UGUI 实现了一套简单的热键注册机制。代码清单 12-24 列出了注册热键函数 RegisterHotKey 和注销热键函数 UnRegisterHotKey 的函数原型。

代码清单 12-24　注册热键函数 RegisterHotKey 和注销热键函数 UnRegisterHotKey 的函数原型

```
/* 注册热键
* @param EFI_KEY_DATA      要注册的热键
```

```
* @param handler          热键对应的处理函数
* @return                 注册成功后，返回该热键对应的 Handle
*/
int RegisterHotKey(EFI_KEY_DATA Key, void(*handler)())

 /* 注销热键
* @param hotKeyHandle     待注销热键的 Handle，即 RegisterHotKey() 返回的 Handle
*/
int UnRegisterHotKey(int hotKeyHandle)
```

2）Tab 键：大部分情况下按下 Tab 键会导致切换焦点；由当前控件的 OnTab( ) 函数响应。

3）Enter 键：大部分情况下按下 Enter 时会执行当前控件中的事件处理函数；由当前控件的 OnEnter( ) 响应。

4）可打印字符键：由当前控件的 OnKey（UINT16 Key）响应。

5）不可打印字符键：ASCII 码为 0 的按键，如方向键。由当前控件的 OnKeyExt（UINT16 Key）响应。

3. 派遣和响应鼠标事件

当鼠标事件发生时，首先需要根据鼠标位置查找目标控件，然后由目标控件响应该事件。此项工作由函数 DispatchMouseMessage(MPOINT MouseState) 完成，其实现如代码清单 12-25 所示。

代码清单 12-25　派遣和响应鼠标事件

```
void Brick::DispatchMouseMessage(MPOINT MouseState)
{
   if(Brick::TabList == NULL) return;
   Brick* ListHeader = Brick::TabList;
   Brick* O = ListHeader;
do{
   // 检查鼠标位置是否落在控件 O 内
     if(O->testHit(MouseState.X, MouseState.Y)){
     // 目前我们只支持单击事件
       if(MouseState.Action == MOUSE_LEFT_CLICK)
          O->OnClick();
        break;
     }else{
        O = O->NextinTabList;
     }
   }while(O != ListHeader);
}
```

在 DispatchMouseMessage 函数中，我们简单地遍历 TabList 中的所有控件，直到

testHit(UINT16 posX, UINT16 posY) 返回 TRUE。testHit(UINT16 posX, UINT16 posY) 的作用是检查点 (posX, posy) 是否落在本控件内。

4. 派遣定时器事件

定时器事件属于全局事件，不隶属于某个特定的控件，因而其派遣函数相比鼠标和键盘事件比较简单。定时器事件首先派发给全局定时器事件处理函数（pUiEvent->HandleTimer()），然后派发给窗口处理函数。

5. 窗口的事件处理函数

在事件处理循环中，每发生一个事件，首先由全局派遣函数将事件派遣到相应的处理函数，然后将事件派遣到当前窗口的窗口处理函数。

```
((Window*)(Uwnd::CurWindow))->m_EventHandler((UINT32)Index,wParam, lParam);
```

- 第一个参数 Index 表示事件类别，第二个参数 wParam 和第三个参数 lParam 是事件的参数，具体意义由事件类别定义。
- 定时器事件（Index 是 UI_TIMER）和鼠标事件（Index 是 UI_MOUSE），以及 wParam 和 lParam 未使用。
- 键盘事件（Index 是 UI_KEY），wParam 为 EFI_INPUT_KEY。

窗口的事件处理函数框架如代码清单 12-26 所示。

代码清单 12-26　窗口的事件处理函数

```
int WndProc(UINT32 msgid, WPARAM wParam, LPARAM lParam)
{
  if(msgid == UIEvent::UI_TIMER){                          // 响应定时器事件
  }else if(msgid == UIEvent::UI_MOUSE){                    // 响应鼠标事件
  }else if(msgid == UIEvent::UI_KEY){                      // 响应键盘事件
    EFI_INPUT_KEY Key= *(EFI_INPUT_KEY*)&wParam;           // 取得按键
  }else {
  }
}
```

12.3　鼠标与控件的绘制

在实现了事件处理循环以及事件的派遣和响应后，我们的应用程序已经实现了事件驱动。窗口应用程序相对于文本界面应用程序的最大优点在于其可视化的图形界面，因此，下面我们还要完成的一项非常重要的工作就是绘制控件。注意，在屏幕上显示的除了控件还有

鼠标，鼠标的绘制不同于一般控件。下面介绍一下鼠标及控件的绘制方法。

12.3.1　鼠标的绘制

鼠标总是显示在所有其他图像的前面。每次移动鼠标或重绘屏幕时，都要更新鼠标。更新鼠标时，要先将原来位置的鼠标擦除（用原始图像覆盖鼠标），然后绘制其他控件，最后在新位置绘制鼠标。绘制鼠标之前还要保存新位置处的原始图像，以备擦除鼠标时恢复原始图像。其绘制流程如代码清单 12-27 所示。

代码清单 12-27　绘制鼠标的流程

```
while(Brick::Running == true){
  //1) 擦除（移动前）鼠标
  Brick::EraseMouse();
  //2) 绘制 UI 控件
...
  //3) 绘制（移动后）鼠标
  Brick::DrawMouse();
  //将后缓冲中的内容更新到屏幕
  Brick::UpdateScreen();
  //等待事件发生
  ...
}
```

Brick::EraseMouse( ) 的作用就是用原始图像覆盖移动前的鼠标，达到擦除鼠标的目的，其实现如代码清单 12-28 所示。

代码清单 12-28　擦除鼠标函数 EraseMouse( ) 的实现

```
void Brick::EraseMouse()
{
  POINT LastMousePos = LastDrawMousePos;          //上一次绘制鼠标时鼠标的位置
  CopyToFrameBuffer(Brick::LocalBitMap(),
        Brick::FrameBufferWidth(),                //屏幕每行宽度
        EraseMouseBitmap,                         //鼠标位图
        MOUSE_BITMAP_WIDTH,                       //鼠标位图宽度
     LastMousePos.X, LastMousePos.Y,              //鼠标坐标
     MOUSE_BITMAP_WIDTH, MOUSE_BITMAP_HEIGHT);
}
```

绘制鼠标时，首先保存目标位置的原始图像，然后绘制鼠标位图，其实现如代码清单 12-29 所示。

代码清单 12-29　绘制鼠标

```
EFI_GRAPHICS_OUTPUT_BLT_PIXEL* MouseBitmap = DefaultMouseBitmap; //默认鼠标位图
static POINT LastDrawMousePos;
```

```
void Brick::DrawMouse()
{
  POINT CurMousePos = Widget::pUiEvent->GetMousePos();
  LastDrawMousePos = CurMousePos;
  //保存原始图像，以备擦除鼠标
  CopyFromFrameBuffer(Widget::LocalBitMap(),Widget::FrameBufferWidth(),
      EraseMouseBitmap, MOUSE_BITMAP_WIDTH,
      CurMousePos.X, CurMousePos.Y, MOUSE_BITMAP_WIDTH, MOUSE_BITMAP_HEIGHT);
  //绘制鼠标，即复制鼠标位图到显示缓冲区
  MaskedCopyToFrameBuffer(Widget::LocalBitMap(),Widget::FrameBufferWidth(),
    MouseBitmap, MOUSE_BITMAP_WIDTH,
    CurMousePos.X, CurMousePos.Y, MOUSE_BITMAP_WIDTH, MOUSE_BITMAP_HEIGHT);
}
```

绘制鼠标时用到了三个函数：CopyFromFrameBuffer(…)、CopyToFrameBuffer(…) 和 MaskedCopyToFrameBuffer(…)。

CopyFromFrameBuffer 的作用是从源缓冲区（Framebuffer）中复制一块子区域（偏移为 [StartX,StartY]，大小为 [Width,Height]）到目的缓冲区 Buffer 中（目标区域偏移为 [0,0]，大小为 [Width,Height]）。其函数原型如代码清单 12-30 所示。

CopyToFrameBuffer 的作用是将源缓冲区 Buffer 内的图像（偏移为 [0,0]，大小为 [Width,Height]）复制到目的缓冲区 FrameBuffer 的子区域（偏移为 [StartX,StartY], 大小为 [Width,Height]）。其函数原型如代码清单 12-30 所示。

代码清单 12-30　CopyFromFrameBuffer 和 CopyToFrameBuffer 函数原型

```
void CopyFromFrameBuffer( IN EFI_GRAPHICS_OUTPUT_BLT_PIXEL* FrameBuffer, //源
  IN UINT16 FrameBufferWidth,                    //FrameBuffer 的宽度（以像素为单位）
  EFI_GRAPHICS_OUTPUT_BLT_PIXEL* Buffer,         //目的缓冲区
  UINT16 BufferStride,                           //目的缓冲区图像每一行的宽度
  UINT16 StartX, UINT16 StartY,
  UINT16 Width, UINT16 Height
);
void CopyToFrameBuffer(OUT EFI_GRAPHICS_OUTPUT_BLT_PIXEL* FrameBuffer, //目的
  IN UINT16 FrameBufferWidth,                    //FrameBuffer 的宽度（以像素为单位）
  IN EFI_GRAPHICS_OUTPUT_BLT_PIXEL* Buffer,      //源
  IN UINT16 BufferStride,                        //源缓冲区图像每一行的宽度
  IN UINT16 StartX, IN UINT16 StartY,
  IN UINT16 Width, IN UINT16 Height);
```

MaskedCopyToFrameBuffer(…) 与 CopyToFrameBuffer(…) 功能相似，二者的区别是 MaskedCopyToFrameBuffer 只复制非零像素。

鼠标位图可以从图像文件加载，也可以利用程序生成。例如，我们可以利用宏来生成鼠标位图，如代码清单 12-31 所示，其中二维数组 CommonMouseBitmap 是一个鼠标位图，指针 MouseBitmap 指向了这个鼠标位图。

代码清单 12-31　利用宏生成鼠标位图

```
#define MOUSE_BITMAP_BASEWIDTH \
(sizeof(CommonMouseBitmap[0])/sizeof(CommonMouseBitmap[0][0]))// 鼠标位图宽度
#define MOUSE_BITMAP_BASEHEIGHT \
(sizeof(CommonMouseBitmap)/sizeof(CommonMouseBitmap[0]))        // 鼠标位图高度
#define HH {0x10,0x10,0x10,0x00}
#define __ {0xFF,0xFF,0xFF,0x00}
#define _  {0x00,0x00,0x00,0x00}
static UEFI_PIXEL CommonMouseBitmap[][0x0c]={
    /*****{00,01,02,03,04,05,06,07,08,09,0a,0b},*****/
    /*00*/{HH,_ ,_ ,_ ,_ ,_ ,_ ,_ ,_ ,_ ,_ ,_ },/*00*/
    /*01*/{HH,HH,_ ,_ ,_ ,_ ,_ ,_ ,_ ,_ ,_ ,_ },/*01*/
    /*02*/{HH,__,HH,_ ,_ ,_ ,_ ,_ ,_ ,_ ,_ ,_ },/*02*/
    /*03*/{HH,__,__,HH,_ ,_ ,_ ,_ ,_ ,_ ,_ ,_ },/*03*/
    /*04*/{HH,__,__,__,HH,_ ,_ ,_ ,_ ,_ ,_ ,_ },/*04*/
    /*05*/{HH,__,__,__,__,HH,_ ,_ ,_ ,_ ,_ ,_ },/*05*/
    /*06*/{HH,__,__,__,__,__,HH,_ ,_ ,_ ,_ ,_ },/*06*/
    /*07*/{HH,__,__,__,__,__,__,HH,_ ,_ ,_ ,_ },/*07*/
    /*08*/{HH,__,__,__,__,__,__,__,HH,_ ,_ ,_ },/*08*/
    /*09*/{HH,__,__,__,__,__,__,__,__,HH,_ ,_ },/*09*/
    /*0a*/{HH,__,__,__,__,__,__,__,__,__,HH,_ },/*0a*/
    /*0b*/{HH,__,__,__,__,__,__,_ ,_ ,_ ,_ ,HH},/*0b*/
    /*0c*/{HH,__,__,__,__,__,__,HH,HH,HH,HH,HH},/*0c*/
    /*0d*/{HH,__,__,_ ,HH,__,__,HH,_ ,_ ,_ ,_ },/*0d*/
    /*0e*/{HH,__,_ ,HH,_ ,HH,__,__,HH,_ ,_ ,_ },/*0e*/
    /*0f*/{HH,_ ,HH,_ ,_ ,HH,__,__,HH,_ ,_ ,_ },/*0f*/
    /*10*/{HH,HH,_ ,_ ,_ ,_ ,HH,__,_ ,HH,_ ,_ },/*10*/
    /*11*/{_ ,_ ,_ ,_ ,_ ,_ ,_ ,HH,HH,_ ,_ ,_ } /*11*/
    /*****{00,01,02,03,04,05,06,07,08,09,0a,0b} *****/
};
UEFI_PIXEL* MouseBitmap = (UEFI_PIXEL*)CommonMouseBitmap;
```

12.3.2　控件的绘制

在事件处理循环中，每次循环都会调用以下语句绘制当前窗口。

```
((Window*)(Brick::CurWindow))->DrawTo(Brick::FrameBuffer, 0);
```

Window 的 DrawTo 函数会重绘该窗口内所有需要更新的控件。

从绘制的角度讲，按钮分为两类，一类是一旦初始化就不发生改变的控件，如静态文本框；另一类是内容经常发生改变的控件，如文本框、按钮等。对于第一类控件，只有重绘整个窗口时才需要绘制；而对于第二类控件，控件发生改变后就需要重新绘制。Window 的 DrawTo 函数如代码清单 12-32 所示。

代码清单 12-32　Window 的 DrawTo 函数

```
void Window::DrawTo(void* pFrameBuffer, UINT32 ToScreen )
{
  if(Brick::IsRedrawFullScreen){                    // 需要重绘整个窗口
```

```
    if(Brick::BrickList){                          //重绘列表中的每一个控件
      ReDrawList<BrickNextHelper, DONOT_CHECK_CHANGE_BEFORE_REDRAW>
          (Brick::BrickList);
    }
  }else if(Brick::TabList){
    //只需重绘可能发生改变的控件，TabList 包含了所有可由 Tab 键到达的控件
    ReDrawList<TabNextHelper, CHECK_CHANGE_BEFORE_REDRAW>(Brick::TabList);
  }
  Brick::IsRedrawFullScreen = false;
}
class TabNextHelper{
public:
  static Brick* Next(Brick* o){return Brick::GetNextInTabList(o);}
};
class BrickNextHelper{
public:
  static Brick* Next(Brick* o){return Brick::GetNextInBrickList(o);}
};
//绘制列表 Header 中的控件
template<class T, bool CheckChanged> void ReDrawList(Brick* Header){
  Brick* ListHeader = Header;
  Brick* O = ListHeader;
  do{
    if((CheckChanged && O->IsChanged()) || (!CheckChanged))O->reDraw();
    O = T::Next(O);
  }while(O != ListHeader);
}
```

到目前为止，开发 GUI 应用程序的基础工作就已经完成了。通过相关知识的介绍，我们知道了如何响应用户输入（鼠标和键盘事件）、如何使用定时器、如何绘制窗口。利用这些完成的基础工作就可以开发视频播放器界面了。另外，我们还希望能让这些基础工作能应用到其他 GUI 应用程序中，因此，可以将这些基础工作组织成一个包，以方便开发 GUI 应用程序。

12.4 控件系统包 GUIPkg

可以将我们的控件系统组织成一个包并命名为 GUIPkg。这样，开发 GUI 应用程序时就可以很方便地使用 GUIPkg 中的控件。在 GUIPkg 目录下有 GUIPkg.dec、GUIPkg.dsc 文件，以及 Include、Library、test 目录。

Include 目录下包含 bmp.h（主要提供位图相关函数）、fb.h（封装了 GraphOut Protocol）、UEvent.h（提供了事件处理接口）和 UGui.h（提供了预定义控件，如按钮、进度条等）。

Library 目录下是 UGui 库。test 目录下是测试 UGui 的测试程序。GUIPkg 的目录结构如图 12-2 所示。

```
GUIPkg
    |-----Include
    |         |-----bmp.h
    |         |-----fb.h
    |         |-----UEvent.h
    |         |-----UGui.h
    |-----Library
    |         |-----bmp.cpp
    |         |-----fb.cpp
    |         |-----UGui.cpp
    |         |-----UGui.inf
    |-----test
    |         |-----testgui.cpp
    |         |-----testgui.inf
|-----GUIPkg.dec
| -----GUIPkg.dsc
```

图 12-2　GUIPkg 目录结构

1. GUIPkg.dec 文件

GUIPkg.dec 用于对外提供资源，主要是提供 GUI 相关头文件的路径，如代码清单 12-33 所示。

代码清单 12-33　GUIPkg.dec 文件

```
[Defines]
  DEC_SPECIFICATION = 0x00010005
  PACKAGE_NAME = EdkGUIPkg
  PACKAGE_GUID = f4805c44-8985-11db-9e98-0040d0c0d0cc
  PACKAGE_VERSION = 1.0
[Includes]
  Include
```

2. GUIPkg.dsc 文件

GUIPkg.dsc 文件用于定义 Library 目录下提供的 LibGui 库，如代码清单 12-34 所示。

代码清单 12-34　GUIPkg.dsc 文件

```
[Defines]
  PLATFORM_NAME = GUIPkg
  PLATFORM_GUID = 1358dada-8b6e-4e45-b773-1b27cbda3e06
  PLATFORM_VERSION = 0.01
```

```
  DSC_SPECIFICATION = 0x00010005
  OUTPUT_DIRECTORY = Build/GUIPkg
  SUPPORTED_ARCHITECTURES = IA32|IPF|X64
  BUILD_TARGETS = DEBUG|RELEASE
  SKUID_IDENTIFIER = DEFAULT
[LibraryClasses]
  ...
  LibGui|uefi/GUIPkg/Library/UGui.inf
```

3. UGui.inf 工程文件

GUIPkg 中比较重要的是 Library 目录下的 UGui，它包含了开发 GUI 界面所需的控件系统，其工程文件如代码清单 12-35 所示。库工程文件的格式在第 3 章已经讲述，这里不再赘述，但有两点需要注意：

1）控件系统采用 C++ 编写，因而需要 CppPkg 的支持。在 [Packages] 块需加入 CppPkg.dec。

2）在编译选项中需加入 /wd4804/wd4201，以关闭编译 C++ 程序时引起的编译警告。

代码清单 12-35　UGui.inf 工程文件

```
[Defines]
  INF_VERSION = 0x00010005
  BASE_NAME = libGui
  FILE_GUID = 348C4D62-BFBD-4882-9ECE-C80BB1C64336
  VERSION_STRING = 1.0
  MODULE_TYPE = BASE
  LIBRARY_CLASS = GUILib
  DEFINE UEFI_BOOK_DIR = uefi/book
[Sources]
  UGui.cpp
  fb.cpp
  bmp.cpp
[Packages]
  StdLib/StdLib.dec
  MdePkg/MdePkg.dec
  MdeModulePkg/MdeModulePkg.dec
  ShellPkg/ShellPkg.dec
  $(UEFI_BOOK_DIR)/CppPkg/CppPkg.dec
  $(UEFI_BOOK_DIR)/GUIPkg/GUIPkg.dec
[LibraryClasses]
  UefiBootServicesTableLib
  UefiLib
[Protocols]
  gEfiSimpleTextInputExProtocolGuid
[BuildOptions]
   MSFT:*_*_*_CC_FLAGS = /wd4804 /wd4201
```

12.5　简单视频播放器的实现

利用上面提供的 GUIPkg，我们可以实现一个简单的视频播放器，该播放器包含一个播放 / 暂停按钮，一个进度条，这两个控件都在播放器窗口内。

1. 播放 / 暂停按钮 PlayerSwitcher

播放 / 暂停按钮是一个 Switcher 控件，Switcher 的 OnClick 函数会响应鼠标单击事件，而 OnClick 函数会调用 OnEnter。为了实现单击该 Switcher 控件能够播放或暂停视频，需要生成一个 Switcher 的子类并实现 OnEnter 函数，在 OnEnter 函数中实现视频控制，这个子类可命名为 PlayerSwitcher，如代码清单 12-36 所示。同时，当播放按钮获得输入焦点时，OnEnter 会响应键盘的回车事件。在 OnEnter 中，播放 / 暂停的控制是通过调用窗口处理函数完成的。

代码清单 12-36　播放控制按钮类 PlayerSwitcher

```
//播放/暂停按钮，通过重载 OnEnter 函数实现播放和暂停功能
class PlayerSwitcher : public Switcher
{
public:
  PlayerSwitcher(CHAR8* on, CHAR8* off):Switcher(on, off){}
  virtual void OnEnter(){
    Switcher::OnEnter();                                    //调用父类函数，用于显示按钮
    EFI_INPUT_KEY   Key = {0, 'p'};
    WndProc(UIEvent::UI_KEY, *(WPARAM*)&Key, 0);            //调用功能函数，实现暂停/播放
  };
};
```

2. 窗口的消息处理函数

窗口的消息处理函数有两项功能：一是响应定时器事件，用于按照一定的帧率显示视频；二是响应按键事件，实现对播放器的快捷键控制。代码清单 12-37 为窗口的消息处理函数。

代码清单 12-37　窗口的消息处理函数

```
//播放器窗口的事件处理函数
int WndProc(UINT32 msgid, WPARAM wParam, LPARAM lParam)
{
  if(msgid == UIEvent::UI_TIMER){
    if(PlayPause == FF_PLAYING){
      DecShowFrame();                                       //显示一帧
    Brick::ReDrawScreen();                                  //重绘整个窗口
   }
  }else if(msgid == UIEvent::UI_MOUSE){
```

```
  }else if(msgid == UIEvent::UI_KEY){
    EFI_INPUT_KEY Key;
    Key = *(EFI_INPUT_KEY*)&wParam;
    if (Key.UnicodeChar == 'q'){
      //按 <Q> 键退出播放器
      Brick::Running = false;
    }else if (Key.UnicodeChar == 'f'){
      //按 <F> 键全屏播放，计算全屏时视频帧的长、宽，并设置视频帧大小
      Width = ... ;
      Height = ... ;
      EFI_STATUS Status = pFFDecoder-> SetFrameSize(pFFDecoder, Width, Height);
    }else if (Key.UnicodeChar == 'p'){
      //按 <P> 键暂停或继续播放
      if(PlayPause == FF_PLAYING){
        PlayPause = FF_PAUSE;
      }else if(PlayPause == FF_PAUSE){
        PlayPause = FF_PLAYING;
      }
    }
  }
  return 0;
}
```

3. 视频显示

视频的显示是通过函数 DecShowFrame 实现的，其功能是从视频流中取出一帧，并将该帧复制到 GUI 的帧缓冲中，最后更新进度条。代码清单 12-38 是 DecShowFrame 函数的实现。

代码清单 12-38　在屏幕上显示一个视频帧

```
void DecShowFrame()
{
  AVFrame *pFrame;
  //取得 GUI 的帧缓冲
  EFI_IMAGE_OUTPUT  *pLocalFrameBuffer = (EFI_IMAGE_OUTPUT*)Brick::FrameBuffer;
  EFI_GRAPHICS_OUTPUT_BLT_PIXEL *Bitmap = 0;
  Bitmap = pLocalFrameBuffer->Image.Bitmap;
  //取出一帧
  if(EFI_ERROR( pFFDecoder-> QueryFrame(pFFDecoder, &pFrame))) return;
  //复制到帧缓冲
  CopyToFrameBuffer(Bitmap, FULL_WIDTH,
    (EFI_GRAPHICS_OUTPUT_BLT_PIXEL *)pFrame->data[0], Width,
    (FULL_WIDTH - Width) /2, (FULL_HEIGHT - Height) /2,
    Width, Height);
  progress->Set(FrameNo++);                          //更新进度条
}
```

4. 播放器主函数

在播放器主函数中，我们需要做如下工作：

- 取得视频解码服务并打开视频。
- 初始化 GUI。
- 生成窗口，并生成窗口内的控制按钮和进度条。
- 进入消息处理循环。
- 清理 GUI 并退出程序。

播放器主函数的代码如代码清单 12-39 所示。

代码清单 12-39　播放器主函数

```
EFI_STATUS ffMainGui (IN UINTN Argc,  IN CHAR16 **Argv)
{
  EFI_STATUS Status;
  //打开 FFDecoder 服务
  Status = gBS->LocateProtocol(&gEfiFFDecoderProtocolGUID,
        NULL, (VOID **)&pFFDecoder );
  if (EFI_ERROR(Status)) {
    return Status;
  }
  //打开视频
  Status = pFFDecoder -> OpenVideo2( pFFDecoder, Argv[1], &pFFDecoder);
  Status = pFFDecoder-> QueryFrameSize(pFFDecoder, &Width, &Height);
  //初始化 GUI
  Brick::Initialize();
  //生成窗口
  Window AdminWindow;
  //设置窗口消息处理函数，视频的显示与播放控制在 WndProc 中完成
  AdminWindow.SetEventHandler(WndProc);
  const int START_X = 100;
  const int START_Y = FULL_HEIGHT - 200;
  //生成播放器按钮
  PlayerSwitcher* btn = new PlayerSwitcher("on.bmp", "off.bmp");
  btn->Pos(START_X, START_Y);                          //设置控件在屏幕上的位置
  btn->Size(PLAY_WIDTH,PLAY_WIDTH);                    //设置控件大小
  btn->SetFocus();
  btn->AddtoTabList();
  //生成播放器进度条
  progress = new ProgressBar();
  progress->Pos(START_X + PLAY_WIDTH , START_Y+ PLAY_WIDTH/2 - 8);
  //设置 progress 控件的长度
  progress->Size<UINT16>((UINT16)(FULL_WIDTH - (START_X + PLAY_WIDTH)*2),10);
  progress->AddtoTabList();
  progress->SetMax(1000);          //设置 progress 控件最大值，最好是同视频长度相同
  //进入消息循环
```

```
  Brick::MsgLoop();
  //清理 GUI
  Brick::Finish();
}
```

最后，将以上程序组合起来就得到了视频播放器的整个程序，如代码清单 12-40 所示。

代码清单 12-40　视频播放器完整代码

```
const INT32 FF_PLAYING = 1;
const INT32 FF_PAUSE = 0;
EFI_FFDECODER_PROTOCOL *pFFDecoder;
UINT32 Width=0, Height=0;
INT32 PlayPause = FF_PLAYING;
int FrameNo = 0;
//从视频流中取出一帧并显示到帧缓存，更新进度条
void DecShowFrame()
{
  AVFrame *pFrame;
  EFI_IMAGE_OUTPUT  *pLocalFrameBuffer = (EFI_IMAGE_OUTPUT*)Brick::FrameBuffer;
  EFI_GRAPHICS_OUTPUT_BLT_PIXEL *Bitmap = 0;
  Bitmap = pLocalFrameBuffer->Image.Bitmap;
  if(EFI_ERROR( pFFDecoder-> QueryFrame(pFFDecoder, &pFrame))) return;
  CopyToFrameBuffer(Bitmap, FULL_WIDTH,
    (EFI_GRAPHICS_OUTPUT_BLT_PIXEL *)pFrame->data[0], Width,
    (FULL_WIDTH - Width) /2,(FULL_HEIGHT - Height) /2,
    Width, Height);
  progress->Set(FrameNo++);
}
//播放器窗口的事件处理函数
int WndProc(UINT32 msgid, WPARAM wParam, LPARAM lParam)
{
  if(msgid == UIEvent::UI_TIMER){
    if(PlayPause == FF_PLAYING){
    DecShowFrame();              //显示一帧
    Brick::ReDrawScreen();       //重绘整个窗口
    }
  }else if(msgid == UIEvent::UI_MOUSE){
  }else if(msgid == UIEvent::UI_KEY){
    EFI_INPUT_KEY Key;
    Key = *(EFI_INPUT_KEY*)&wParam;
    if (Key.UnicodeChar == 'q'){
      //按 <Q> 键退出播放器
      Brick::Running = false;
    }else if (Key.UnicodeChar == 'f'){
      //按 <F> 键全屏播放
      #define TMPFACTOR_TYPE UINT32
      #define FAXTOR_NUM 1000
      TMPFACTOR_TYPE factor = 0, factorh = 0;
      factor=(((TMPFACTOR_TYPE )FULL_WIDTH)*FAXTOR_NUM)/TMPFACTOR_TYPE)Width;
```

```
      factorh=(((UINT32)FULL_HEIGHT)*FAXTOR_NUM)/(TMPFACTOR_TYPE)Height;
      if(factor > factorh) factor = factorh;
      //缩放视频
      if( factor != 1.0 ){
        Width = (UINT32)((TMPFACTOR_TYPE )Width * factor) / FAXTOR_NUM;
        Height = (UINT32)((TMPFACTOR_TYPE )Height * factor) / FAXTOR_NUM;
        EFI_STATUS Status = pFFDecoder-> SetFrameSize(pFFDecoder,Width,Height);
        if(EFI_ERROR(Status))
          Brick::Running = false;
      }
    }else if (Key.UnicodeChar == 'p'){
      //按 <P> 键暂停或继续播放
      if(PlayPause == FF_PLAYING){
        PlayPause = FF_PAUSE;
      }else if(PlayPause == FF_PAUSE){
        PlayPause = FF_PLAYING;
      }
    }
  }else{
  }
  return 0;
}
//播放/暂停按钮，通过重载 OnEnter 函数实现播放和暂停功能
class PlayerSwitcher : public Switcher
{
public:
  PlayerSwitcher(CHAR8* on, CHAR8* off):Switcher(on, off){}
  virtual void OnEnter(){
  Switcher::OnEnter();                          //调用父类函数，用于显示按钮
  EFI_INPUT_KEY   Key = {0, 'p'};
  WndProc(UIEvent::UI_KEY, *(WPARAM*)&Key, 0); //调用功能函数，实现暂停 / 播放
  };
};
EFI_STATUS ffMainGui (IN UINTN Argc,  IN CHAR16 **Argv)
{
  EFI_STATUS Status;
  //打开 FFDecoder 服务
  Status = gBS->LocateProtocol(&gEfiFFDecoderProtocolGUID,
    NULL, (VOID **)&pFFDecoder );
  if (EFI_ERROR(Status)) {
    return Status;
  }
  //打开视频
  Status = pFFDecoder -> OpenVideo2( pFFDecoder, Argv[1], &pFFDecoder);
  Status = pFFDecoder-> QueryFrameSize(pFFDecoder, &Width, &Height);
  //初始化 GUI
  Brick::Initialize();
  //生成窗口
  Window AdminWindow;
```

```
  //设置窗口消息处理函数，视频的显示与播放控制在 WndProc 中完成
  AdminWindow.SetEventHandler(WndProc);
  const int START_X = 100;
  const int START_Y = FULL_HEIGHT - 200;
  //生成播放器按钮
  PlayerSwitcher* btn = new PlayerSwitcher("on.bmp", "off.bmp");
  btn->Pos(START_X, START_Y);            //设置控件在屏幕上的位置
  btn->Size(PLAY_WIDTH,PLAY_WIDTH);      //设置控件大小
  btn->SetFocus();
  btn->AddtoTabList();
  //生成播放器进度条
  progress = new ProgressBar();
  progress->Pos(START_X + PLAY_WIDTH , START_Y+ PLAY_WIDTH/2 - 8);
  //设置 progress 控件的长度
  progress->Size<UINT16>((UINT16)(FULL_WIDTH - (START_X + PLAY_WIDTH)*2),10);
  progress->AddtoTabList();
  progress->SetMax(1000);                //设置 progress 控件最大值，最好是同视频长度相同
  //进入消息循环
  Brick::MsgLoop();
  //清理 GUI
  Brick::Finish();
}
int cppMain(int argc, char **argv)
{
  INTN  ReturnFromMain = -1;
  EFI_SHELL_PARAMETERS_PROTOCOL *EfiShellParametersProtocol = NULL;
  EFI_STATUS Status;
  Status = gST ->BootServices->OpenProtocol(gImageHandle,
    &gEfiShellParametersProtocolGuid, VOID **)&EfiShellParametersProtocol,
    gImageHandle, NULL, EFI_OPEN_PROTOCOL_GET_PROTOCOL);
  if (!EFI_ERROR(Status)) {
    ReturnFromMain = ffMainGui(EfiShellParametersProtocol->Argc,
      EfiShellParametersProtocol->Argv);
  } else {
    ...
  }
  ...
  return (int)Status;
}
```

5. 播放器工程文件

有了源文件，还要有工程文件才能在 EDK2 中编译，ffplayer.inf 工程文件的代码如代码清单 12-41 所示。在此工程文件中，需要注意以下几点：

- [Defines] 块中 MODULE_TYPE 设为 UEFI_APPLICATION。ENTRY_POINT 设为 ShellCEntryLib，这是使用 CppPkg 的标准设置。

- [Packages] 块中要引用 CppPkg.dec 和 GUIPkg.dec。
- [LibraryClasses] 要引用 ShellCEntryLib、LibC、LibGui 和 LibCpp。
- [BuildOptions] 编译选项要添加 MSFT:*_*_*_CC_FLAGS = /wd4804 /wd4201。这是 EDK2 中用 VS 编译 C++ 程序的标准选项配置。

代码清单 12-41　播放器工程文件

```
[Defines]
  INF_VERSION = 0x00010006
  BASE_NAME = ffplayer
  FILE_GUID = 4ea97c46-7491-4dfd-b442-74798713ce5f
  MODULE_TYPE = UEFI_APPLICATION
  VERSION_STRING = 0.1
  ENTRY_POINT = ShellCEntryLib
  DEFINE UEFI_BOOK_DIR = uefi\book
[Sources]
  fplayer.cpp
[Packages]
  StdLib/StdLib.dec
  MdePkg/MdePkg.dec
  MdeModulePkg/MdeModulePkg.dec
  ShellPkg/ShellPkg.dec
  $(UEFI_BOOK_DIR)/CppPkg/CppPkg.dec
  $(UEFI_BOOK_DIR)/GUIPkg/GUIPkg.dec
[LibraryClasses]
  ShellCEntryLib
  UefiLib
  LibC
  LibString
  LibStdLib
  LibGui
  LibCpp
[Protocols]
  gEfiGraphicsOutputProtocolGuid
  gEfiSimplePointerProtocolGuid
  gEfiSimpleTextInputExProtocolGuid
[BuildOptions]
   MSFT:*_*_*_CC_FLAGS = /wd4804 /wd4201
```

将 ffplayer.inf 文件添加到 GUIPkg.dsc 文件的 [Components] 部分，然后就可以通过以下命令编译了。

```
build -p uefi\book\GUIPkg\GUIPkg.dsc
```

12.6　本章小结

本章我们介绍了 GUI 开发的基本原理。首先实现了一个简单的 GUI 开发库 GUIPkg，然

后使用这个 GUI 开发库开发了一个视频播放器。利用 GUIPkg 的开发主要是围绕以下三个问题展开的。

（1）UEFI 下如何捕获 GUI 事件

在 GUIPkg 中，使用 gBS->WaitForEvents 捕获键盘、鼠标和定时器事件。

（2）控件如何响应事件

在事件循环中捕获一个事件后，调用事件派遣函数将事件派遣到目标控件。每个窗口有一个当前活动控件，键盘事件总是派遣到当前活动控件；鼠标事件则需要首先确定活动控件，然后派遣。最后，所有的事件还会派遣到当前窗口。

（3）如何显示 GUI 控件

控件的显示主要包括控件的绘制、控件的更新及鼠标的绘制。

下一章将介绍多任务的相关知识。

第 13 章 Chapter 13

深入了解多任务

一般而言，UEFI 的主要作用是检测和初始化设备，加载操作系统的引导程序，然后将控制权交给操作系统，整个过程不需要大量的运算，在单个 CPU 核上运行单线程程序已经可以满足需求，因此，EDK2 没有提供多线程机制。如果我们需要在 UEFI 应用中支持多任务，则可以使用 EFI_MP_SERVICES_PROTOCOL 启动从 CPU，也可以自己实现一个多线程库，在每个 CPU 上运行并发任务。下面我们就从这两个方面开始讲述。

13.1 多处理器服务

在多处理器系统中，系统初始化时，总是由一个处理器执行这些初始化指令，这个 CPU 称为 BSP（Boot-Strap Processor）。系统中 BSP 之外的 CPU 称为 AP（Application Processor）。本节将要讲述如何利用 EFI_MP_SERVICES_PROTOCOL 在 AP 上执行任务，以及 BSP 启动 AP 的原理和过程。

13.1.1 EFI_MP_SERVICES_PROTOCOL 功能及用法

PI（Platform Initalization）标准中定义了 EFI_MP_SERVICES_PROTOCOL（简称 MP 服务），这个 Protocol 用于在多处理器系统中管理处理器，包括查询处理器信息、启动或停止 AP、设定 BSP，其结构体如代码清单 13-1 所示。

代码清单 13-1 EFI_MP_SERVICES_PROTOCOL 结构体

```
typedef struct _EFI_MP_SERVICES_PROTOCOL EFI_MP_SERVICES_PROTOCOL;
struct _EFI_MP_SERVICES_PROTOCOL {
```

```
  EFI_MP_SERVICES_GET_NUMBER_OF_PROCESSORS GetNumberOfProcessors;  //获取 CPU 数目
  EFI_MP_SERVICES_GET_PROCESSOR_INFO GetProcessorInfo; //获取某个处理器的信息
  EFI_MP_SERVICES_STARTUP_ALL_APS StartupAllAPs;       //在所有 AP 上启动指定的任务
  EFI_MP_SERVICES_STARTUP_THIS_AP StartupThisAP;       //在指定 AP 上启动指定的任务
  EFI_MP_SERVICES_SWITCH_BSP SwitchBSP;                //选择某个 AP 作为 BSP
  EFI_MP_SERVICES_ENABLEDISABLEAP EnableDisableAP;     //启用或禁用某个 AP
  EFI_MP_SERVICES_WHOAMI WhoAmI;                       //返回当前处理器的编号
};
```

除了 WhoAmI 服务外，EFI_MP_SERVICES_PROTOCOL 中的其他服务都只能在 BSP 上调用，在 AP 上调用这些服务时会返回 EFI_DEVICE_ERROR 错误。下面详细讲述 EFI_MP_SERVICES_PROTOCOL 各个服务的用法。

1. 查询服务

EFI_MP_SERVICES_PROTOCOL 提供的查询服务有以下三个。

- GetNumberOfProcessors 用于查询系统中的逻辑处理器个数，以及启用的逻辑处理器个数。
- GetProcessorInfo 用于获取指定处理器的相关信息。
- WhoAmI 用于获取当前处理器的编号。

GetNumberOfProcessors 服务的函数原型如代码清单 13-2 所示。

代码清单 13-2　GetNumberOfProcessors 函数原型

```
//查询系统中的逻辑处理器个数，以及启用的逻辑处理器个数
typedef EFI_STATUS(EFIAPI *EFI_MP_SERVICES_GET_NUMBER_OF_PROCESSORS)(
  IN  EFI_MP_SERVICES_PROTOCOL  *This,
  OUT UINTN *NumberOfProcessors,                  //返回系统中存在的所有逻辑处理器的个数
  OUT UINTN *NumberOfEnabledProcessors            //返回系统启用的逻辑处理器的个数
  );
```

GetProcessorInfo 服务的函数原型如代码清单 13-3 所示。其功能是查询指定逻辑处理器有关多处理器的信息。此服务仅返回多处理器相关信息（如处理器编号、处理器状态等），不提供平台相关的处理器信息（如 Cache 大小、处理器频率等）。

查询时通过处理器编号（参数 ProcessorNumber）指定待查询的处理器。处理器编号从 0 开始。系统启动后，UEFI 把系统中的处理器从 0 开始编号，ProcessorNumber 就是这个编号，EFI_MP_SERVICES_PROTOCOL 中其他服务中的 ProcessorNumber 意义与之相同。如果指定的处理器编号超出系统中存在的处理器个数，则返回 EFI_NOT_FOUND。

代码清单 13-3　GetProcessorInfo 函数原型

```
//获取指定处理器的相关信息
typedef EFI_STATUS(EFIAPI *EFI_MP_SERVICES_GET_PROCESSOR_INFO)(
```

```
  IN   EFI_MP_SERVICES_PROTOCOL    *This,
  IN   UINTN ProcessorNumber,                                  //处理器编号
  OUT EFI_PROCESSOR_INFORMATION *ProcessorInfoBuffer           //返回处理器（多处理器相关）信息
  );
```

GetProcessorInfo 服务返回的处理器信息存放在 EFI_PROCESSOR_INFORMATION 结构体内，该结构体如代码清单 13-4 所示。

代码清单 13-4　处理器信息 EFI_PROCESSOR_INFORMATION 结构体

```
//每个逻辑 CPU 包含的信息
typedef struct {
  UINT64 ProcessorId;
  UINT32 StatusFlag;                                           //CPU 状态
  EFI_CPU_PHYSICAL_LOCATION  Location;                         //CPU 物理地址
} EFI_PROCESSOR_INFORMATION;
```

其中，ProcessorId 是由系统硬件决定的地址。对 IA32 和 x64 处理器来说，它与本地 APIC ID 地址相同，低 8 位有效。对安腾处理器来说，低 16 位地址有效。要注意，ProcessorId 与 ProcessorNumber（处理器编号）不同。

StatusFlag 是处理器状态，说明如下。

- 第 0 位是 BSP 位（PROCESSOR_AS_BSP_BIT），为 0 时说明该处理器是 AP；为 1 时说明该处理器是 BSP。
- 第 1 位是 Enabled 位（PROCESSOR_ENABLED_BIT），0 表示该处理器被禁用；1 表示该处理器被启用。
- 第 2 位是 Health 位（PROCESSOR_HEALTH_STATUS_BIT），0 表示该处理器出现故障；1 表示该处理器正常。
- 3 ~ 31 位必须是 0。

处理器信息还包含一个域 Location，它是 EFI_CPU_PHYSICAL_LOCATION 类型的变量，表示处理器的物理地址，用于给每一个 CPU 定位。在一个计算机系统中，可能有多颗处理器（一颗处理器称为一个 Package），每颗处理器可能有多个物理核（物理核称为 Core），每个物理核又可能有多个逻辑核（逻辑核称为 Thread）。所有的编号都从 0 开始。EFI_CPU_PHYSICAL_LOCATION 结构体如代码清单 13-5 所示。

代码清单 13-5　EFI_CPU_PHYSICAL_LOCATION 结构体

```
typedef struct {
  UINT32  Package;                                             //计算机系统内 Package 的编号
  UINT32  Core;                                                //每个 Package 内 Core 的编号
  UINT32  Thread;                                              //每个 Core 内 Thread 的编号
} EFI_CPU_PHYSICAL_LOCATION;
```

对于一个逻辑处理器，有三种地址：

1）ProcessNumber，作为 MP Service 服务的参数，假设系统中有 n 个处理器，ProcessNumber 在 0 ~ n–1 之间，称为处理器编号。

2）Location 是由 Package、Core、Thread 组成的三元组，可给一个逻辑处理器定位。

3）ProcessorId，由系统硬件决定的地址。

WhoAmI 用于获得当前处理器的编号（ProcessorNumber），其函数原型如代码清单 13-6 所示。

代码清单 13-6　WhoAmI 函数原型

```
typedef EFI_STATUS(EFIAPI *EFI_MP_SERVICES_WHOAMI)(
  IN EFI_MP_SERVICES_PROTOCOL *This, OUT UINTN *ProcessorNumber );
```

2. 管理处理器

EFI_MP_SERVICES_PROTOCOL 中用于管理处理器的服务有以下 4 个：

- StartupThisAP 服务用在指定的 AP 上执行传入的函数。
- StartupAllAPs 服务用于在所有 AP 上执行传入的函数。
- SwitchBSP 服务用于选定某个 AP 作为 BSP。
- EnableDisableAP 服务用于启用或禁用某个 AP。

（1）StartupThisAP 服务

StartupThisAP 服务的函数原型如代码清单 13-7 所示。此服务只能在 BSP 上执行，用于在指定的 AP（此 AP 必须没有被禁用）上执行调用者提供的函数。调用者可以指定 Timeout 时间。

- 如果逾期时 AP 上的任务还未完成，那么 AP 上的任务将被终止。
- 如果 AP 未处于 IDLE 状态，那么该服务会返回 EFI_NOT_READY。
- 如果 AP 未处于 Enable 状态，那么该服务会返回 EFI_INVALID_PARAMETER。

此服务有阻塞和异步两种模式。参数 WaitEvent 为 NULL 时，该服务使用阻塞模式；否则，使用异步模式。

- 阻塞模式下，该服务向 AP 发出指令后，等待 AP 把调用者提供的函数执行完毕，然后该服务成功返回。若 AP 完成任务前已逾期，则逾期时服务返回 EFI_TIMEOUT。
- 异步模式下，该服务向 AP 发出指令后立即返回。AP 执行完任务后触发该服务提供的事件 WaitEvent，并将 Finished 标志设为 TRUE；若 AP 完成任务前已逾期，则逾期时终止 AP 上的任务，触发事件 WaitEvent，并将 Finished 标志设为 FALSE。

系统事件 EFI_EVENT_GROUP_READY_TO_BOOT 被触发后，该服务仅提供阻塞模式。

代码清单 13-7　StartupThisAP 函数原型

```
//启动指定的 AP
typedef EFI_STATUS(EFIAPI *EFI_MP_SERVICES_STARTUP_THIS_AP)(
  IN EFI_MP_SERVICES_PROTOCOL  *This,
  IN EFI_AP_PROCEDURE Procedure,                    //在 AP 上执行的函数
  IN UINTN ProcessorNumber,                         //逻辑处理器编号
  //若该参数为 0，则服务执行阻塞模式；若该参数有效，则 AP 结束任务或超时后 WaitEvent 被触发
  IN EFI_EVENT WaitEvent OPTIONAL,
  //AP 上任务的最长执行时间，逾期则 AP 上的任务被终止。若该参数为 0，则 AP 要执行任务直到完毕
  IN UINTN TimeoutInMicroseconds,
  IN VOID *ProcedureArgument OPTIONAL,              //函数 Procedure 的参数
  //仅异步模式有效，当 WaitEvent 触发后，*Finshed 被设置，
  //TURE 表示 AP 上的任务执行完毕，FALSE 表示任务终止
  OUT BOOLEAN *Finished OPTIONAL
  );
```

（2）StartupAllAPs 服务

StartupAllAPs 用于在所有的 AP 上执行指定的函数。如果有任何 AP 未处于 IDLE 状态，则该函数返回 EFI_NOT_READY。如果参数 SingleThread 为 TRUE，则 AP 会依次执行 Procedure 函数，前一 AP 从 Procedure 返回后下一 AP 才会执行 Procedure。如果参数 SingleThread 为 FALSE，则所有的 AP 同时执行 Procedure，开发者负责 Procedure 的线程安全。

StartupAllAPs 服务有阻塞和异步两种模式。参数 WaitEvent 为 NULL 时，该服务使用阻塞模式；否则，使用异步模式。

- 阻塞模式下，所有 AP 从 Procedure 返回或超时后，StartupAllAPs 返回。FailedCpuList 列表返回未成功执行 Procedure 的 CPU 的编号，系统为 FailedCpuList 分配内存，调用者负责释放该内存。
- 异步模式下，所有 AP 从 Procedure 返回或超时后，WaitEvent 事件触发，系统同样会设置 FailedCpuList 列表。

StartupAllAPs 服务的函数原型如代码清单 13-8 所示。

代码清单 13-8　StartupAllAPs 函数原型

```
typedef EFI_STATUS(EFIAPI *EFI_MP_SERVICES_STARTUP_ALL_APS)(
  IN EFI_MP_SERVICES_PROTOCOL  *This,
  IN EFI_AP_PROCEDURE Procedure,                    //此函数将在所有 AP 上执行
  IN BOOLEAN  SingleThread,                         /是否并发启动所有 AP
  IN EFI_EVENT  WaitEvent OPTIONAL,                 //仅对异步模式有效
  IN UINTN  TimeoutInMicroSeconds,                  //超时时间
  IN VOID *ProcedureArgument OPTIONAL,              //Procedure 的参数
  OUT UINTN **FailedCpuList OPTIONAL                //未成功执行 Procedure 的 CPU 列表
  );
```

（3）SwitchBSP 服务

SwitchBSP 用于将指定的 AP 切换为 BSP。成功切换后，该服务返回 EFI_SUCCESS，新的 BSP 将无缝接管原 BSP 的工作。当系统事件 EFI_EVENT_GROUP_READY_TO_BOOT 触发后，该服务不再有效。其函数原型如代码清单 13-9 所示。

代码清单 13-9　SwitchBSP 函数原型

```
typedef EFI_STATUS(EFIAPI *EFI_MP_SERVICES_SWITCH_BSP)(
  IN EFI_MP_SERVICES_PROTOCOL  *This,
  IN UINTN ProcessorNumber,       //将此 AP 设置为 BSP
  IN BOOLEAN EnableOldBSP         //切换 BSP 后，判断原来作为 BSP 的 CPU 是否启用
  );
```

（4）EnableDisableAP 服务

EnableDisableAP 服务用于启动或禁用指定的 AP，其函数原型如代码清单 13-10 所示。

代码清单 13-10　EnableDisableAP 函数原型

```
typedef EFI_STATUS(EFIAPI *EFI_MP_SERVICES_ENABLEDISABLEAP)(
  IN EFI_MP_SERVICES_PROTOCOL  *This,
  IN UINTN ProcessorNumber,       //指定 CPU 的编号
  IN BOOLEAN EnableAP,            //启用或禁用
  IN UINT32 *HealthFlag OPTIONAL //新的健康标志，仅 PROCESSOR_HEALTH_STATUS_BIT 有效
  );
```

3. EFI_MP_SERVICES_PROTOCOL 示例

下面通过具体的示例展示 EFI_MP_SERVICES_PROTOCOL 的用法。首先要取得 EFI_MP_SERVICES_PROTOCOL 实例，如示例 13-1 所示。

【示例 13-1】 取得 EFI_MP_SERVICES_PROTOCOL 实例。

```
//@file book/multitask/processor/process.c
#include <Uefi.h>
#include <Library/UefiBootServicesTableLib.h>
#include <Library/UefiLib.h>
#include "Protocol/MpService.h"
EFI_MP_SERVICES_PROTOCOL* mpp;
EFI_STATUS LocateMPP(EFI_MP_SERVICES_PROTOCOL** mpp)
{
  return  gBS->LocateProtocol( &gEfiMpServiceProtocolGuid, NULL, (void**)mpp);
}
```

使用 GetNumberOfProcessors 和 GetProcessorInfo 服务获得系统内处理器的信息，如示例 13-2 所示。

【示例 13-2】 获取系统内处理器信息。

```
EFI_STATUS testMPPInfo()
{
  EFI_STATUS Status;
  UINTN nCores = 0, nRunning = 0;
  Status = mpp -> GetNumberOfProcessors(mpp, & nCores, &nRunning);
  Print(L"System has %d cores, %d cores are running\n", nCores, nRunning);
  {
    UINTN i = 0;
    for(i =0; i< nCores; i++){
      EFI_PROCESSOR_INFORMATION mcpuInfo;
      Status = mpp -> GetProcessorInfo( mpp, i, &mcpuInfo);
      Print(L"CORE %d:\n", i);
      Print(L"  ProcessorId\t:%d\n", mcpuInfo.ProcessorId);
      Print(L"  StatusFlag\t:%x\n", mcpuInfo.StatusFlag);
      Print(L"  Location\t:(%d %d %d)\n", mcpuInfo.Location.Package,
        mcpuInfo.Location.Core, mcpuInfo.Location.Thread);
    }
  }
  return Status;
}
```

使用 StartupThisAP 服务，在 1 号 AP 处理器上执行 APFoo(&APParam1)，同时在 BSP 上执行 APFoo(&APParam0)，如示例 13-3 所示。

【示例 13-3】 使用 StartupThisAP 服务。

```
typedef struct { UINTN id, loops, UINTN ret; }APParam;
//在 AP 上运行的函数
VOID EFIAPI APFoo(VOID* context)
{
  APParam *param = (APParam*) context;          //取得输入参数
  UINTN i = 0;
  for(i=0;i< param->loops; i++){
    gBS->Stall(1000);
    Print(L"AP %d runs %d\n", param->id, i);
  }
  param -> ret = param -> loops * 10;           //设置返回值
}
EFI_STATUS testMPPStartup()
{
  EFI_STATUS Status;
  BOOLEAN Finshed = FALSE;
  APParam APParam0 = {0, 5, 0};
  APParam APParam1 = {1, 20, 0};
  EFI_EVENT ApEvent;
  UINTN Index;
  Status = gBS -> CreateEvent(0, TPL_CALLBACK, NULL, NULL, &ApEvent);
  //以异步方式在 1 号处理器上执行 APFoo
  Status = mpp -> StartupThisAP ( mpp,
           APFoo,                               //将在 AP 上执行的函数
           1,                                   //AP 的处理器编号
```

```
            ApEvent,                        //
            0,                              //TimeOut 设置为 0，即 AP 上的任务执行完毕才结束
            (VOID*)&APParam1,               //函数 APFoo 的参数
             &Finshed);
  Print(L"StatupThisAP :%r\n", Status);
  if(Status == 0){
    APFoo((VOID*)&APParam0);
    gBS->WaitForEvent(1, &ApEvent, &Index);     //等待 AP 上的任务执行完毕
    //检查 Finshed 状态，确定 AP 任务是否成功结束
    Print(L"AP %s and return %d\n", Finshed == TRUE?L"Finished":L"failed",
          APParam1.ret);
  }
  return Status;
}
```

process.inf 是标准的 UEFI 应用模块。在 process.inf 文件的 [Protocols] 中加入以下内容：

```
gEfiMpServiceProtocolGuid
```

编译 process.inf 后会生成可执行文件 process.efi。要测试 process.efi，需在 UEFI 模拟器中执行这个程序。

简单讲一下如何编译和运行模拟器，UEFI 模拟器的编译非常简单，在 Linux 执行以下命令：

```
EmulatorPkg/build.sh
```

把编译好的 process.efi 复制到 Build/Emulator32/DEBUG_GCC44/IA32/，然后运行模拟器。运行模拟器用以下命令：

```
EmulatorPkg/build.sh run
```

然后就可以在模拟器上执行 process.efi。

13.1.2 启动 AP 的过程

下面介绍一下启动 AP 的原理和过程。

1. 通过发送 IPIs 消息启动 AP

每个处理器 Core 上都有一个 APIC（Advanced Programmable Interrupt Controller）芯片，称为本地 APIC。它接收来自本地或外部的中断信号，发送给 Core 上的处理器处理这个中断。这些中断信号包括：来自处理器 Core 中断引脚的信号、来自 I/O APIC 的中断、来自性能监视寄存器的中断、本地 APIC 时钟中断、本地 APIC 错误中断、温度传感器中断、来自其他处理器的中断（Inter-Processor Interrupts，简称 IPIs）。AP 的启动是通过在 BSP 上

向 AP 发送特定的 IPIs 完成的，而发送 IPIs 是通过写本地 APIC 的 ICR（Interrupt Command Register）完成的。

多处理器系统启动时，首先根据 MP 初始化协议选择一个处理器为 BSP，其他处理器作为 AP。BSP 执行初始化代码，初始化系统的 APIC 环境，建立系统的全局数据结构，初始化 AP（向 AP 发送 INIT-SIPI-SIPI IPIs）。等 BSP 和 AP 初始化完成后，BSP 会执行 OS Loader 加载操作系统。

AP 重置（收到 INIT IPI 中断信号）或加电后，首先简单地进行自我配置，然后等待来自 BSP 的启动信号（来自 BSP 的 IPI），这个启动信号称为 SIPI。AP 收到 SIPI 后，会从实模式 0x000*XX*000 地址处开始执行。*XX* 是 SIPI 消息中的 8bit 的地址向量。

（1）寄存器 ICR 及 IPI 类型

发送 IPI 是通过写寄存器 ICR 完成的，先来看一下 ICR。ICR 是 64 位寄存器，可分为两个 32 位寄存器。其中低 32 位的寄存器偏移为 0x300，高 32 位的寄存器偏移为 0x310。ICR 数据结构如代码清单 13-11 所示。

代码清单 13-11　ICR 数据结构

```
// ICR 中的低 32 位
typedef union {
  struct {
    UINT32  Vector:8;              // 中断向量
    UINT32  DeliveryMode:3;        // IPI 类型
    UINT32  DestinationMode:1;     // 0: 物理目标模式; 1: 逻辑目标模式
    UINT32  DeliveryStatus:1;      // IPI 投递状态, 0(Idle): 成功; 1(Peding): 未完成
UINT32  Reserved0:1;
UINT32  Level:1;                   // 在 INIT level de-assert IPI 类型下必须是 1, 其他模式为 0
    UINT32  TriggerMode:1;         // 0: 边沿触发; 1: 电平触发
    UINT32  Reserved1:2;
    // 修饰目标 AP。0: 使用 Destination; 1: 仅投递给自己;
    // 2: 投递给所有处理器 (包括自己);
    // 3: 投递给所有处理器 (不包括自己)
    UINT32  DestinationShorthand:2;
    UINT32  Reserved2:12;          // 保留
  } Bits;
  UINT32    Uint32;
} LOCAL_APIC_ICR_LOW;
// ICR 的高 32 位
typedef union {
  struct {
    UINT32  Reserved0:24;          // 保留
    UINT32  Destination:8;         // 目标处理器的 ProcessorId, 即 APIC 地址
  } Bits;
  UINT32    Uint32;
} LOCAL_APIC_ICR_HIGH;
```

可用 IPI 类型（DeliveryMode）共有 7 种：

1）0：向目的处理器发送 Vector 指定的中断。

2）1：同 000，但仅向目标处理器中优先级最低的处理器发送。

3）2：向目标处理器发送 SMI 中断，Vector 必须是 0。

4）4：向目标处理器发送 NMI 中断，Vector 被忽略。

5）5：向目标处理器发送 INIT 中断（称为 INITIPI），目标处理器收到 INITIPI 后进行初始化，然后等待 SIPI。

6）6：向目标处理器发送 SIPI（startup）中断，目标处理器收到 SIPI 后执行 Vector 指定的代码。

7）7：向目标处理器发送信号，目标处理器收到信号后向外部 8259A 控制器发出请求从而获得 Vector。

类型 3 被保留。IPI 类型的宏定义如代码清单 13-12 所示。

代码清单 13-12　IPI 类型的宏定义

```
#define LOCAL_APIC_DELIVERY_MODE_FIXED                    0
#define LOCAL_APIC_DELIVERY_MODE_LOWEST_PRIORITY          1
#define LOCAL_APIC_DELIVERY_MODE_SMI                      2
#define LOCAL_APIC_DELIVERY_MODE_NMI                      4
#define LOCAL_APIC_DELIVERY_MODE_INIT                     5
#define LOCAL_APIC_DELIVERY_MODE_STARTUP                  6
#define LOCAL_APIC_DELIVERY_MODE_EXTINT                   7
```

（2）向目标处理器发送 IPI

下面来看如何向目标处理器发送 IPI。前面讲过，发送 IPI 是通过写寄存器 ICR 实现的。发送流程为：

1）写 ICR 高 32 位的寄存器 XAPIC_ICR_HIGH_OFFSET，该寄存器包含了目的处理器的 APCI 地址。

2）写 ICR 低 32 位的寄存器 XAPIC_ICR_LOW_OFFSET。写入该寄存器后 IPI 被发送。目的 CPU 收到消息后，发送方的 XAPIC_ICR_LOW_OFFSET 的 DeliveryStatus 会设置为 0。

3）检查 XAPIC_ICR_LOW_OFFSET 的 DeliveryStatus，直到该值变为 0。

代码清单 13-13 中的 SendIpi 函数实现了这个流程，该函数位于 UefiCpuPkg\Library\BaseXApicLib\BaseXApicLib.c。

代码清单 13-13　发送 IPI 的 SendIpi 函数的实现

```
// 通过写寄存器 ICR 向目标处理器发送 IPI，函数等到目标处理器接收 IPI 后返回
VOID SendIpi (
  IN UINT32          IcrLow,          // 此值写入 ICR 低 32 位
```

```
  IN UINT32          ApicId              // 目标处理器本地 APCI 地址，即 ProcessorId
  )
{
  LOCAL_APIC_ICR_LOW IcrLowReg;
  ASSERT (GetApicMode () == LOCAL_APIC_MODE_XAPIC);
  ASSERT (ApicId <= 0xff);
  // 写 ICR 高 32 位
  WriteLocalApicReg (XAPIC_ICR_HIGH_OFFSET, ApicId << 24);
  // 写 ICR 低 32 位。写入低 32 位将导致 IPI 消息被发送给目标处理器
  WriteLocalApicReg (XAPIC_ICR_LOW_OFFSET, IcrLow);
  // 检查 IcrLowReg 寄存器的 DeliveryStatus 标志位，直到 DeliveryStatus 为 0 时才返回.
  // DeliveryStatus 为 0 表示目标处理器已接收 IPI 消息
  do {
    IcrLowReg.Uint32 = ReadLocalApicReg (XAPIC_ICR_LOW_OFFSET);
  } while (IcrLowReg.Bits.DeliveryStatus != 0);
}
```

（3）启动 AP 的 IPI 序列

启动 AP 是通过向 AP 发送 INIT-SIPI-SIPI 序列完成的，有了 SendIpi 函数，发送 INIT-SIPI-SIPI 的任务就不难完成了。先来看发送 INIT IPI 是如何实现的。代码清单 13-14 中的 SendInitIpi 函数正是用于发送 INIT IPI 消息的。在 SendInitIpi 函数中，首先构造 INIT IPI 消息，然后调用 SendIpi 函数发送该消息。

代码清单 13-14　SendInitIpi 函数

```
// 向目标处理器发送 INIT IPI 消息
VOID EFIAPI SendInitIpi ( IN UINT32 ApicId  )
{
  LOCAL_APIC_ICR_LOW IcrLow;
  // 构造 INIT IPI
  IcrLow.Uint32 = 0;
  IcrLow.Bits.DeliveryMode = LOCAL_APIC_DELIVERY_MODE_INIT;
  IcrLow.Bits.Level = 1;
  // 发送 IPI 消息
  SendIpi (IcrLow.Uint32, ApicId);
}
```

下面再来看 INIT-SIPI-SIPI 序列的发送过程。代码清单 13-15 列出的 SendInitSipiSipi 函数用于发送这个消息序列。其实现过程分为三步：发送 INIT IPI 消息，发送 SIPI 消息，再次发送 SIPI 消息。在 SIPI 消息中包含了 AP 启动向量 StartupRoutine。目标处理器接收 IPI 序列后 SendInitSipiSipi 函数返回。目标 AP 接收到 INIT-SIPI-SIPI IPI 序列后，从实模式物理地址 StartupRoutine 处开始执行代码。StartupRoutine 必须在 1MB 地址之内，并且必须按 4KB 对齐。

代码清单 13-15 SendInitSipiSipi 函数

```
//@file UefiCpuPkg/Library/BaseXApicLib/BaseXApicLib.c
VOID EFIAPI SendInitSipiSipi (
  IN UINT32 ApicId,                                    //目标处理器的本地 APIC 地址
  IN UINT32 StartupRoutine                             //启动代码物理地址
  )
{
  LOCAL_APIC_ICR_LOW IcrLow;
  ASSERT (StartupRoutine < 0x100000);
  ASSERT ((StartupRoutine & 0xfff) == 0);
  //1）发送 INIT IPI
  SendInitIpi (ApicId);
  MicroSecondDelay (10);                               //延时
  //2）发送 SIPI
  IcrLow.Uint32 = 0;
  IcrLow.Bits.Vector = (StartupRoutine >> 12);         //设置启动代码向量
  IcrLow.Bits.DeliveryMode = LOCAL_APIC_DELIVERY_MODE_STARTUP;
  IcrLow.Bits.Level = 1;
  SendIpi (IcrLow.Uint32, ApicId);
  MicroSecondDelay (200);
  //3）再次发送 SIPI
  SendIpi (IcrLow.Uint32, ApicId);
}
```

2. AP 启动向量

现在我们知道可以使用函数 SendInitSipiSipi(TargetCpu,mStartupVector) 启动一个 AP。AP 收到启动消息后会执行 mStartupVector。那么 BSP 是如何准备 mStartupVector 这个启动向量的呢?

启动向量 mStartupVector 是一段代码的基址，在程序中它定义为 EFI_PHYSICAL_ADDRESS 类型的变量。地址 mStartupVector 必须在 1MB 地址之内，并且必须按 4KB 对齐。

（1）为启动向量分配内存

BSP 首先要为 mStartupVector 分配内存。代码清单 13-16 列出的 AllocateStartupVector 用于为启动向量 mStartupVector 分配内存。该函数在物理地址为 0x02000 ~ 0x7F000 的内存中找到可供使用的内存并分配给 mStartupVector 使用。

代码清单 13-16 为启动向量分配内存的函数 AllocateStartupVector

```
//@file
<EdkCompatibilityPkg/Compatibility/MpServicesOnFrameworkMpServicesThunk
/MpServicesOnFrameworkMpServicesThunk.c
VOID AllocateStartupVector (  UINTN   Size  )
{
  EFI_STATUS  Status;
  Status = gBS->LocateProtocol (&gEfiGenericMemTestProtocolGuid,
```

```
                NULL, (VOID **) &mGenMemoryTest );
  if (EFI_ERROR (Status)) {
    mGenMemoryTest = NULL;
  }
  //在 0x02000 ~ 0x7F000 的内存中找到可供使用的内存
  for(mStartupVector=0x7F000;mStartupVector>=0x2000;
    mStartupVector-= EFI_PAGE_SIZE) {
    if (mGenMemoryTest != NULL) {
      //检查该地址是否可用
      Status = TestReservedMemory (EFI_SIZE_TO_PAGES (Size) * EFI_PAGE_SIZE);
      if (Status == EFI_DEVICE_ERROR) {
        continue;
      }
    }
    //分配 mStartupVector 指定的内存页
    Status = gBS->AllocatePages (AllocateAddress, EfiBootServicesCode,
      EFI_SIZE_TO_PAGES (Size), &mStartupVector );
    if (!EFI_ERROR (Status)) {
      break;                        //分配成功，退出循环
    }
  }
  ASSERT_EFI_ERROR (Status);
}
```

有了为启动向量分配内存的函数后，再看 BSP 是如何初始化 mStartupVector 的。

（2）初始化启动向量

代码清单 13-17 中列出的 PrepareAPStartupVector 函数用于初始化启动向量 mStartupVector。该函数首先取得启动代码的地址、大小等信息，这些信息包含在 MP_ASSEMBLY_ADDRESS_MAP 类型的变量 AddressMap 中。然后根据启动代码的大小调用 AllocateStartupVector(…) 为启动向量 mStartupVector 分配内存。接着将启动代码复制到 mStartupVector 指向的内存，并设置启动向量中的相关跳转地址，为 AP 分配 GDT（全局描述符表）和 IDT（中断描述符表）并存入中断向量。

代码清单 13-17　初始化启动向量的 PrepareAPStartupVector 函数

```
VOID PrepareAPStartupVector (  VOID  )
{
  MP_ASSEMBLY_ADDRESS_MAP        AddressMap;
  IA32_DESCRIPTOR                GdtrForBSP;
  IA32_DESCRIPTOR                IdtrForBSP;
  EFI_PHYSICAL_ADDRESS           GdtForAP;
  EFI_PHYSICAL_ADDRESS           IdtForAP;
  EFI_STATUS                     Status;
  //取得启动代码的地址及大小
  AsmGetAddressMap (&AddressMap);
  //为 mStartupVector 分配内存
```

```
  AllocateStartupVector (AddressMap.Size + sizeof (MP_CPU_EXCHANGE_INFO));
  //将启动代码复制到 mStartupVector
  CopyMem ((VOID *) (UINTN) mStartupVector, AddressMap.RendezvousFunnelAddress,
      AddressMap.Size);
  //设置启动代码内的跳转地址
  *(UINT32 *) (UINTN) (mStartupVector + AddressMap.FlatJumpOffset + 3) = (UINT32)
      (mStartupVector + AddressMap.PModeEntryOffset);
  ...
  //设置 AP 的 GDT 和 IDT，详细代码读者可参阅 EDK2 中的源码
}
```

（3）取得启动代码地址和大小

下面来看 AsmGetAddressMap 是如何取得启动代码地址和大小的。AsmGetAddressMap(MP_ASSEMBLY_ADDRESS_MAP*) 函数通过 MP_ASSEMBLY_ADDRESS_MAP 类型的变量返回启动代码的相关信息。MP_ASSEMBLY_ADDRESS_MAP 的结构体如代码清单 13-18 所示。

代码清单 13-18　MP_ASSEMBLY_ADDRESS_MAP 结构体

```
typedef struct {
  UINT8 *RendezvousFunnelAddress;          //启动代码地址
  UINTN PModeEntryOffset;                  //保护模式代码的偏移
  UINTN FlatJumpOffset;                    //跳转地址的偏移
  UINTN LModeEntryOffset;
  UINTN LongJumpOffset;
  UINTN Size;                              //启动代码大小
} MP_ASSEMBLY_ADDRESS_MAP;
```

AsmGetAddressMap 函数的实现如代码清单 13-19 所示。从该函数可以看出，RendezvousFunnelProcStart 是 AP 的启动代码地址。

代码清单 13-19　AsmGetAddressMap 函数

```
;@file EdkCompatibilityPkg/Compatibility/MpServicesOnFrameworkMpServicesThunk
/IA32/MpFuncs.asm
AsmGetAddressMap   PROC  near C  PUBLIC
  pushad
  mov   ebp,esp
  mov   ebx, dword ptr [ebp+24h]
  ;取得启动代码地址 RendezvousFunnelProcStart
  mov   dword ptr [ebx], RendezvousFunnelProcStart
  ;取得保护模式代码的偏移
  mov   dword ptr [ebx+4h], ProtectedModeStart-RendezvousFunnelProcStart
  ;跳转地址的偏移
  mov   dword ptr [ebx+8h], FLAT32_JUMP - RendezvousFunnelProcStart
  mov   dword ptr [ebx+0ch], 0
  mov   dword ptr [ebx+10h], 0
```

```
  ;启动代码的大小
  mov  dword ptr [ebx+14h], RendezvousFunnelProcEnd - RendezvousFunnelProcStart
  popad
  ret
AsmGetAddressMap   ENDP
```

mStartupVector 分为两个部分，开头是启动代码，启动代码后面紧跟一个 MP_CPU_EXCHANGE_INFO 结构体。启动代码主要用于使 AP 从实模式转换到保护模式，主要是系统相关代码。在 MP_CPU_EXCHANGE_INFO 中包含了一个重要的数据，即 ApFunction，AP 进入保护模式后会执行 ApFunction，ApFunction 是保护模式下的函数。代码清单 13-20 是 MP_CPU_EXCHANGE_INFO 的结构体。

代码清单 13-20　MP_CPU_EXCHANGE_INFO 的结构体

```
typedef struct {
  UINTN Lock;
  VOID *StackStart;
  UINTN StackSize;
  VOID *ApFunction;                    //AP 进入保护模式后会执行此函数
  IA32_DESCRIPTOR GdtrProfile;
  IA32_DESCRIPTOR IdtrProfile;
  UINT32 BufferStart;
  UINT32 Cr3;
  UINT32 ProcessorNumber[MAX_CPU_NUMBER];
} MP_CPU_EXCHANGE_INFO;
```

mStartupVector 初始化完成后，我们就可以通过 SendInitSipiSipi(TargetCpu, mStartupVector) 启动 AP 了。

3. 启动 AP 的完整示例

代码清单 13-21 中的 WakeUpAp 函数给出了一个使用 SendInitSipiSipi 的范例。该函数位于 EdkCompatibilityPkg/Compatibility/MpServicesOnFrameworkMpServicesThunk/MpServicesOnFrameworkMpServicesThunk.c 文件内。

代码清单 13-21　WakeUpAp 函数

```
//启动指定 AP，并使 AP 执行指定的函数
VOID WakeUpAp (
  IN UINTN ProcessorNumber,                    //目标处理器编号
  IN EFI_AP_PROCEDURE Procedure,               //将要在 AP 上执行的函数
  IN VOID *ProcArguments                       //Procedure 函数的参数
  )
{
  EFI_STATUS Status;
```

```
  CPU_DATA_BLOCK *CpuData;
  EFI_PROCESSOR_INFORMATION ProcessorInfoBuffer;

  ASSERT (ProcessorNumber < mNumberOfProcessors);
  ASSERT (ProcessorNumber < MAX_CPU_NUMBER);

  //取得目标 CPU 的数据
  CpuData = &mMPSystemData.CpuData[ProcessorNumber];
  AcquireSpinLock (&CpuData->CpuDataLock);
  //将 Procedure 和 ProcArguments 写入属于目标处理器的 CpuData
  CpuData->Parameter = ProcArguments;
  CpuData->Procedure = Procedure;
  ReleaseSpinLock (&CpuData->CpuDataLock);
  Status = GetProcessorInfo (&mMpService,
           ProcessorNumber, &ProcessorInfoBuffer);
  ASSERT_EFI_ERROR (Status);
  //设置 mExchangeInfo->ApFunction 为 ApProcWrapper, AP 执行完启动代码后
  //将执行 mExchangeInfo->ApFunction, 即 ApProcWrapper,
  //在 ApProcWrapper 中, 将执行 CpuData->Procedure
  mExchangeInfo->ApFunction = (VOID *) (UINTN) ApProcWrapper;
  mExchangeInfo->ProcessorNumber[ProcessorInfoBuffer.ProcessorId] = (UINT32)
      ProcessorNumber;
  //发送 INIT-SIPI-SIPI 消息序列
  SendInitSipiSipi (
    (UINT32) ProcessorInfoBuffer.ProcessorId,
    (UINT32) (UINTN) mStartupVector );
}

VOID ApProcWrapper (VOID)
{
  ...
  //取得自己的处理器编号
  WhoAmI (&mMpService, &ProcessorNumber);
  //取得属于该 CPU 的数据
  CpuData = &mMPSystemData.CpuData[ProcessorNumber];
  AcquireSpinLock (&CpuData->CpuDataLock);
  Procedure = CpuData->Procedure;
  Parameter = CpuData->Parameter;
  ReleaseSpinLock (&CpuData->CpuDataLock);
  if (Procedure != NULL) {
    Procedure (Parameter);
    ...
  }
  ...
}
```

至此，我们已经介绍了 AP 的启动过程，以及 EFI_MP_SERVICES_PROTOCOL 的用法。下面我们将要介绍线程部分。

13.2　内联汇编基础和寄存器上下文的保存与恢复

在介绍多线程之前，我们需要学习一点内联汇编，以及寄存器上下文切换方面的知识。

13.2.1　内联汇编基础

在线程切换过程中，我们要操作栈寄存器，因而需要使用要使汇编指令。因为 VS 编译 64 位程序时不支持内联汇编，所以我们选择使用 GCC 编译程序。这里简要介绍一下 GCC 的内联汇编。

GCC 内联汇编使用关键字 asm、__asm__ 和 __asm，这三种符号意义相同。

单行汇编基本格式：

```
asm(" 汇编语句 "); 或 __asm__(" 汇编语句 ");
```

多行汇编基本格式：

```
asm(" 语句 1\n\t"" 语句 2\n\t");
```

语句中，% 用于引用寄存器，如 %eax、%rax。$ 用于引用立即数，如 $8、$0x10。语句的基本格式是如下所示。

```
指令 [b | w | l | q] 源操作数 , 目的操作数
```

b(byte)、w(word)、l(long)、q(quad) 表示操作数宽度。

示例 13-4 是 GCC 内联汇编的一个示例。

【示例 13-4】 GCC 内联汇编示例。

```
__asm__("movl %ebx, %eax\n\t"    // ebx -> eax
        "movl $4,   %ecx\n\t"    // 4 -> ecx
        );
```

GCC 还提供了扩展内联格式。使用扩展内联格式时 GCC 会帮我们在一定程度上管理寄存器。语法如代码清单 13-22 所示。示例 13-5 展示了如何用 GCC 扩展内联汇编交换变量 a、b 的值。

代码清单 13-22　GCC 内联汇编语法

```
asm (" 语句模板 "
"..."
" 语句模板 "
    : "[=][&] 约束 " ( 左值 ) [,…]                    // 输出操作数列表
    : " 约束 " ( 表达式 ) [,…]                        // 输入操作数列表
    : " 寄存器 " [,…]                                 // 使用的寄存器列表
    );
```

【示例 13-5】 GCC 扩展内联汇编

```
int a=2, b =3;
//交换 a、b 的值
asm("movl %0, %ecx"
  "movl %1, %0"
  "movl %ecx, %0"
  :"=a","=b"(b)
  "0"(a),"1"(b)
  :"ecx");
```

GCC 处理扩展格式汇编时，需要如下三步：

1）首先处理使用的寄存器列表和输入操作数列表，将输入操作数读到指定的寄存器。

2）根据语句模板生成代码。

3）根据输出参数列表将寄存器中的数值复制到输出操作数中。

大部分情况下，第 1 步会与前面的代码优化合并，第 3 步会与后面的代码优化合并，这样第 1 步和第 3 步只存在于逻辑上。例如，在上面的示例中，a 会分配成寄存器 eax，b 会分配成寄存器 ebx。

把输出操作数列表和输入操作数列表中列出的寄存器从前至后编号，在语句模板中通过“% 编号”引用这些寄存器。例如，在上例中，%0 指第 0 个寄存器 eax，%1 指第 1 个寄存器 ebx。通过“%% 寄存器名”引用其他寄存器，例如用 %%ecx 引用 ecx 寄存器。

列表内每一项用逗号分隔。三个列表均可为空。

在输出操作数列表中，= 表示只写，& 表示此值在读取输入操作数之前产生。括号内必须是左值，最后“约束”中的值会传给括号内的值。

在输入操作数列表中，约束可以为数字，表示与第 *n* 个（输出，输入操作数统一编号）操作数使用相同的约束。例如，"0"(a) 表示与第 0 个操作数即 "=a"(a) 使用相同的约束，也就是使用寄存器 a(eax)。

常用的约束见表 13-1。

表 13-1 GCC 内联汇编寄存器约束

约　束	约束的意义	约　束	约束的意义
" r "	寄存器，由编译器分配具体的寄存器	" a "	%rax、%eax、%ax 或 %al
" b "	%rbx、%ebx、%bx 或 %bl	" c "	%rcx、%ecx、%cx 或 %cl
" d "	%rdx、%edx、%dx 或 %dl	" S "	%rsi、%esi 或 %si
" D "	%rdi、%edi 或 %di	" m "	操作数为内存操作数
" o "	内存操作数，并且前后一小块区域都是有效地址	" V "	内存操作数，前后一小块区域内有非法地址
" i "	立即数，汇编时必须能确定其值	" n "	常量

使用 " m " 约束时，指令直接使用内存操作数。与之相对的是 " r " 等寄存器约束，操

作数首先被从内存读到寄存器，然后在寄存器中操作此数，最后从寄存器写回内存。

13.2.2　寄存器上下文的保存与恢复

SetJump 用于保存寄存器上下文。寄存器上下文 (包括调用 SetJump 函数后的返回地址) 保存到 JumpBuffer 中。调用后返回值总是 0，其函数原型如下所示。

```
UINTN EFIAPI SetJump (OUT     BASE_LIBRARY_JUMP_BUFFER  *JumpBuffer );
```

LongJump 用于恢复寄存器上下文。调用该函数后，JumpBuffer 保存的寄存器上下文恢复到当前寄存器中，eip 恢复为 JumpBuffer 中保存的 SetJump 函数返回地址。eax 设置为 Value，作为 SetJump 函数的返回值，其函数原型如下所示。

```
VOID EFIAPI LongJump (
  IN BASE_LIBRARY_JUMP_BUFFER *JumpBuffer,
  IN UINTN Value
  );
```

BASE_LIBRARY_JUMP_BUFFER 是体系结构相关变量，它用于存放寄存器上下文，其结构体如代码清单 13-23 所示。目前，它支持 IA32、X64、IPF、EBC、ARM 几种体系结构。以 IA32 和 X64 两种体系结构为例看一下 BASE_LIBRARY_JUMP_BUFFER 的内容。

代码清单 13-23　BASE_LIBRARY_JUMP_BUFFER 结构体

```
// IA32
typedef struct {
  UINT32 Ebx;   UINT32 Esi;   UINT32 Edi;
  UINT32 Ebp;   UINT32 Esp;   UINT32 Eip;
} BASE_LIBRARY_JUMP_BUFFER;
// x64
typedef struct {
  UINT64 Rbx;   UINT64 Rsp;   UINT64 Rbp;
  UINT64 Rdi;   UINT64 Rsi;   UINT64 R12;
  UINT64 R13;   UINT64 R14;   UINT64 R15;
  UINT64 Rip;   UINT64 MxCsr;
  UINT8 XmmBuffer[160];          // XMM6-XMM15
} BASE_LIBRARY_JUMP_BUFFER;
```

可以看出，在 X64 系统下，SIMD 寄存器也由 SetJump 函数保存，因而，在 X64 系统下，如果程序中使用了 SIMD 指令，那么也可以使用 SetJump/LongJump 长跳转；但在 IA32 系统下，仅常规寄存器被保存，如果程序中使用了 SIMD 指令，则使用 SetJump/LongJump 时要非常小心。

下面通过示例 13-6 说明 SetJump/LongJump 的用法。SetJumpFlag=SetJump(JumpBuffer) 执行后，SetJumpFlag 值为 0。其后的 if 语句进入第一个分支，执行 LongJump(JumpBuffer,1) 后流程跳转到 AfterJump，并且 SetJumpFlag 变为 1。因而 if 语句将进入第二个分支，并执行 LongJump(JumpBuffer,2)，执行流程跳转到 AfterJump，并且 SetJumpFlag 变为 2。此后，if

语句将不进入任何分支，程序退出。

【示例 13-6】 SetJump/LongJump 示例。

```
//@file uefi\book\thread\setjmp\Main.c
#include <Uefi.h>
#include <Library/BaseLib.h>
#include <Library/MemoryAllocationLib.h>
EFI_STATUS UefiMain (IN EFI_HANDLE ImageHandle,
  IN EFI_SYSTEM_TABLE  *SystemTable)
{
  BASE_LIBRARY_JUMP_BUFFER* jmpBuf;
  BASE_LIBRARY_JUMP_BUFFER* JumpBuffer;
  UINTN SetJumpFlag;
  //为 jmpBuf 分配内存，并保证 JumpBuffer 指向 BASE_LIBRARY_JUMP_BUFFER 要求的对齐地址
  jmpBuf = AllocatePool(sizeof(BASE_LIBRARY_JUMP_BUFFER) + BASE_LIBRARY_JUMP_BUFFER_
      ALIGNMENT);
  JumpBuffer = ALIGN_POINTER (jmpBuf, BASE_LIBRARY_JUMP_BUFFER_ALIGNMENT);
  //调用 SetJump 保存寄存器上下文
  SetJumpFlag = SetJump (JumpBuffer);
AfterJump:
  if(SetJumpFlag == 0){
    SystemTable -> ConOut-> OutputString(SystemTable->ConOut, L"SetJump return 0\n");
    //跳转到 JumpBuffer 保存的上下文并设置返回值 SetJumpFlag 为 1
    LongJump(JumpBuffer, 1);
  }else if(SetJumpFlag == 1) {
    SystemTable -> ConOut-> OutputString(SystemTable->ConOut, L"SetJump return 1\n");
    //跳转到 JumpBuffer 保存的上下文并设置返回值 SetJumpFlag 为 2
    LongJump(JumpBuffer, 2);
  }
  SystemTable -> ConOut-> OutputString(SystemTable->ConOut, L"SetJump return 2\n");
  FreePool(jmpBuf);
  return EFI_SUCCESS;
}
```

执行上述代码后，输出如下：

```
FS0:\> TestJmp
SetJump return 0
SetJump return 1
SetJump return 2
```

13.3 多线程

UEFI Spec 没有提供多线程机制，如果在开发 UEFI 应用的过程中需要多线程，那么需要用户自己实现多线程库。下面我们实现一个简单的多线程库，这个线程库仅仅实现了线程

的切换与调度，并没有涉及信号量与锁。

编写多线程库，首先要理解线程的两个核心内容：线程切换时机与线程切换机制。

（1）线程切换时机

切换分两种，一种是主动切换，另一种是被动切换。当某个线程需要等待某个事件的发生，如网络应用等待远程服务器响应，可以主动切换到其他线程。当线程时间片用完时，需要切换到其他线程，这是被动切换。

（2）线程切换机制

要理解线程切换机制，首先要清楚每个线程执行过程中都拥有哪些资源。每个线程拥有独立的栈，还拥有执行上下文即寄存器环境。当我们将旧的线程切换到新的线程时，需要保存旧线程的寄存器上下文，加载新线程的寄存器上下文，在切换寄存器上下文的同时发生了栈的切换。寄存器切换我们使用 SetJump/LongJump 函数。

为了能更清晰地说明线程切换机制，我们将要设计的线程库所有线程有相同的优先级，所有线程均分时间片。

13.3.1　生成线程

生成线程的主要作用是分配线程所需的资源，包括线程结构体及栈，以及初始化栈及线程寄存器上下文。先来看线程结构体，如示例 13-7 所示。系统用双向链表将所有线程链接起来，因而 thread_list 除了包含线程结构体 dmthread_t 外，还包含两个链表指针。因为所有线程有同样的优先级，所以系统有一个线程列表即可。

【示例 13-7】 线程结构体 dmthread_t 及 thread_list。

```
typedef struct dmthread_t{
    dmtcontext sig_context;              // 线程寄存器上下文
    unsigned short status;               // 线程状态，如就绪、消亡
    unsigned short priority;             // 优先级
    thread_func_t kernel;                // 线程函数
    void *  arg;                         // kernel 的参数
    char *  stack;                       // 线程的栈
} dmthread_t;
typedef struct thread_list{
    dmthread_t thread;
    volatile struct thread_list *   next;
    struct thread_list *   prev;
} thread_list;
```

下面介绍如何生成线程。示例 13-8 是 create_thread 函数，用于生成线程，其工作流程为：

1）生成线程结构体并加入线程列表。

2）调用 start_thread，为线程分配栈，并启动新线程，进入新线程初始化执行环境后立

即返回主线程。

3）执行权回到主线程。

【示例 13-8】 生成线程函数 create_thread。

```
void create_thread(thread_func_t f, void *  arg)
{

  //此函数必须以 __asm enter 开头
  thread_list *  new_thread;
  if(  myEvent == 0){
    initTimer();
  }
  //生成新的线程
  new_thread = _new_thread(f, arg);
  if(sys.threads == 0){
    thread_list *  main_thread = _new_thread(0,0);
    sys.threads = sys.current = main_thread;
  }
  //将新生成的线程加入线程列表
  _insert_thread(new_thread, sys.threads->prev);
  sys.current = new_thread;
  //初始化新建线程的栈及寄存器上下文并启动该线程
  start_thread(& new_thread->thread);
  //执行权返回主线程
  sys.current = sys.threads;
  //必须保证本函数以 __asm leave 结束，否则将无法切换回主线程栈

}
```

再来看 start_thread 是如何启动新线程的，其核心功能是栈的分配与切换。start_thread 函数代码如示例 13-9 所示，其主要流程是：

1）保存 start_thread 函数栈帧的栈顶与栈底。

2）为新线程分配栈。

3）将主线程栈内 start_thread 函数栈帧的数据复制到新线程栈底。

4）切换栈为新线程的栈。

5）调用 SetJump 保存寄存器上下文。

6）返回到上级函数。

【示例 13-9】 启动新线程的函数 start_thread。

```
void start_thread(dmthread_t*  thread)
{
  UINTN i = 0;
  ptr_size* gsp = 0;
  ptr_size* gbp = 0;
```

```
  int disp;
  char *  stack_btm;
  __asm( "mov %%" bp_register ",%0":"=m"(gbp):);        //保存当前栈的栈底到 gdp
  __asm( "mov %%" sp_register ",%0":"=m"(gsp):);        //保存当前栈的栈顶到 gsp
  gThreads++;
  //分配栈
gBS->AllocatePool(EfiBootServicesData,STACK_SIZE+128,(void**)&thread->stack);
  disp = (char*)gbp - (char*)gsp;
  stack_btm =((char*)thread->stack)+ STACK_SIZE-(gbp-gsp+4)*sizeof(ptr_size);
  //复制主线程栈到新分配的栈内存中
  for(i =0;i<disp /sizeof(ptr_size)+ 4;i++){
    ((ptr_size*)stack_btm)[i] = gsp[i];
  }
  gbp = (ptr_size*)((char*)stack_btm + disp);
  //切换栈，即设置 BP、SP 寄存器的值
  __asm( "mov %0, %%" sp_register ::"m"(stack_btm));
  __asm( "mov %0, %%" bp_register ::"m"(gbp));
  //保存寄存器上下文
  i = SetJump(&thread->sig_context);
  if( i > 0){
    //从线程将会执行到此处
    (thread->kernel)(thread->arg);
    //执行完毕，主动切换到下一线程
    if  (sys.threads -> next ){
      sys.current->thread.status=STATUS_DEAD;
      gThreads --;
      Skedule();
    }else{
      //从线程不可能执行到此处
      *(int*)0 = 0;
    }
  }
  //返回到主线程
}
```

下面解析一下上面的过程。

首先，进入 start_thread 函数后，保存当前函数栈帧的栈顶与栈底，并为新线程栈分配内存。

然后，复制 start_thread 栈帧到新分配的栈内存中，目的有两个：

1）栈切换后，函数退出时也能返回到 create_thread 函数。

2）栈切换后，仍然可以使用栈上的数据，如传入 create_thread 函数的参数 thread。

新栈准备完毕后就切换栈，即设置 BP、SP 寄存器的值。

之后，栈已经由主线程栈切换到新建线程栈，从线程的执行环境已经就绪了。这里将成为主从线程的分界点。保存寄存器上下文后，主线程从这里返回，从线程将从这里开始。

最后是 start_thread 函数的终点，即 start_thread 函数退出的地方。只有主线程将会执行到此处，从线程永远不会执行到这里。那么主线程执行到这里之后会返回到哪里呢？回

顾一下代码，我们新建栈的时候将主线程栈的数据复制到新栈中，也就是说，返回地址已经复制到新栈中，那么它就会返回到正确的位置，即 create_thread 中 start_thread（& new thread → thread）；的下一条语句。

13.3.2 调度线程

线程的被动调度在定时器回调函数中完成。因此，在创建线程之前要先初始化 Timer。

示例 13-10 是初始化定时器的函数 initTimer。相信读者已经非常熟悉定时器的使用了，此处不再赘述。

【示例 13-10】 初始化定时器的函数 initTimer。

```
void initTimer()
{
  EFI_STATUS  Status;
  Status = gBS->CreateEvent(EVT_TIMER | EVT_NOTIFY_SIGNAL, TPL_CALLBACK, (EFI_EVENT_
      NOTIFY)Skedule, (VOID*)NULL, &myEvent);
  Status = gBS->SetTimer(myEvent,TimerPeriodic , 1 * 1000 * 1000);
}
```

再来看线程的调度函数 Skedule，该函数执行后，线程将发生切换，其实现如示例 13-11 所示。

线程的调度主要有以下三部分工作：

1）清除已经结束的线程。

2）通过 SetJump 保存正在执行的线程的上下文。

3）通过 LongJump 跳转到新线程的上下文。

【示例 13-11】 线程调度函数 Skedule。

```
void Skedule()
{
  UINTN i = 0;
  //
  thread_list *  oldThread;
  oldThread = sys.current;
  if(oldThread->thread.status == STATUS_DEAD){          //如果当前线程执行完毕，
      //将当前线程结构体从列表中删除，sys 指向下一个线程
    remove_thread();
    //调度下一个线程
    LongJump(& sys.current->thread.sig_context, 1);
  }
  free_dead_stack();
  sys.current = getNext(sys.current);
  //保存旧线程的上下文
  i = SetJump(&oldThread->thread.sig_context);
```

```
    if( i  ==  0) {
      gBS->RestoreTPL(TPL_APPLICATION);
      //调度下一个线程
      LongJump(& sys.current->thread.sig_context, 1);
    }
  }
```

13.3.3　等待线程结束

在主线程退出之前一定要确保其他线程已经结束，并关闭定时器事件。因为主线程结束意味着程序结束，若不关闭定时器时间，会造成系统崩溃。thread_join 函数用于完成这两项工作，如示例 13-12 所示。

【示例 13-12】 等待线程结束的函数 thread_join。

```
void thread_join()
{
  while(sys.current->next && (sys.current->next != sys.current)){
    //如果还有从线程，调度从线程
    Skedule();
  }
  //所有线程都执行完毕，关闭 Timer
  closeTimer();
}

void closeTimer()
{
  gBS->CloseEvent(myEvent);
  myEvent = 0;
}
```

13.3.4　SimpleThread 服务

我们可以将线程库封装成 Protocol，以服务的形式提供给开发者使用。如何开发服务在第 8 章已经讲述，这里不再详述，仅仅列出源码供读者参考。完整的代码可参见 uefi\book\thread\SimpleThread\，编译该目录下的工程后将生成 SimpleThread.efi 文件。

1. EFI_SIMPLETHREAD_PROTOCOL 服务

示例 13-13 是 SimpleThread.h 文件，用于声明 EFI_SIMPLETHREAD_PROTOCOL。

【示例 13-13】 SimpleThread.h 文件。

```
#define EFI_SIMPLETHREAD_PROTOCOL_GUID \
{ \
  0xda445171, 0xabcd, 0x11d2, {0x8e, 0x4f, 0x0, 0xa0, 0xc9, 0x69, 0x72, 0x3b } \
}
//兼容 EFI 1.1
```

```
#define SIMPLETHREAD_PROTOCOL  EFI_SIMPLETHREAD_PROTOCOL_GUID
typedef EFI_SIMPLETHREAD_PROTOCOL  EFI_SIMPLETHREAD;
//线程函数
typedef void ( * THREAD_FUNC_T)(void * );
//该函数用于生成新的线程，并使新线程处于就绪状态
typedef EFI_STATUS(EFIAPI* EFI_CREATE_THREAD  )(
      IN  EFI_SIMPLETHREAD_PROTOCOL  *This,
      IN  THREAD_FUNC_T Thread,                    //线程函数
      IN  VOID *  Arg                              //传给线程函数的参数
      );
//等待所有从线程结束，必须在主线程内调用
typedef EFI_STATUS(EFIAPI* EFI_JOIN_THREAD  )(
      IN  EFI_SIMPLETHREAD_PROTOCOL  *This
      );
struct _EFI_SIMPLETHREAD_PROTOCOL{
   UINT64 Revision;
   EFI_CREATE_THREAD  create_thread;               //生成线程
   EFI_JOIN_THREAD  thread_join;                   //等待从线程结束
};
```

示例 13-14 是 SimpleThread.c 文件，该文件是驱动型服务 EFI_SIMPLETHREAD_PROTOCOL 的驱动框架部分。

【示例 13-14】 SimpleThread.c 文件。

```
#include <Uefi.h>
#include <Library/BaseLib.h>
#include <Library/UefiBootServicesTableLib.h>
#include "dmthread.h"
#include "SimpleThread.h"
typedef struct {
  UINTN Signature;
  EFI_SIMPLETHREAD_PROTOCOL SimpleThread;
} SIMPLETHREAD_PRIVATE_DATA;
#define SIMPLETHREAD_PRIVATE_DATA_SIGNATURE  SIGNATURE_32 ('T', 'R', 'E', 'D')
#define SIMPLETHREAD_PRIVATE_DATA_FROM_THIS(a) CR (a, SIMPLETHREAD_PRIVATE_DATA,
    SIMPLETHREAD, SIMPLETHREAD_PRIVATE_DATA_SIGNATURE)
static SIMPLETHREAD_PRIVATE_DATA gSIMPLETHREADPrivate;
EFI_GUID gEfiSimpleThreadProtocolGUID = EFI_SIMPLETHREAD_PROTOCOL_GUID;
EFI_STATUS EFIAPI  CreateThread( IN  EFI_SIMPLETHREAD_PROTOCOL  *This,
  IN  THREAD_FUNC_T Thread, IN  VOID *  Arg )
{
  create_thread(Thread, Arg);
  return 0;
}
EFI_STATUS EFIAPI JoinThread(IN EFI_SIMPLETHREAD_PROTOCOL* This )
{
  thread_join();
```

```
  return 0;
}
EFI_STATUS EFIAPI
UefiMain(IN EFI_HANDLE ImageHandle, IN EFI_SYSTEM_TABLE *SystemTable)
{
  EFI_STATUS Status;
  SIMPLETHREAD_PRIVATE_DATA* Private = &gSIMPLETHREADPrivate;
  //初始化 EFI_SIMPLETHREAD_PROTOCOL 实例
  gSIMPLETHREADPrivate.Signature = SIMPLETHREAD_PRIVATE_DATA_SIGNATURE;
  gSIMPLETHREADPrivate.SimpleThread.create_thread = CreateThread;
  gSIMPLETHREADPrivate.SimpleThread.thread_join = JoinThread;
  //安装 EFI_SIMPLETHREAD_PROTOCOL
  Status = gBS->InstallProtocolInterface (&ImageHandle,
           &gEfiSimpleThreadProtocolGUID,  EFI_NATIVE_INTERFACE,
           &Private->SimpleThread);
  return 0;
}
```

2. 测试 EFI_SIMPLETHREAD_PROTOCOL

示例 13-15 利用 EFI_SIMPLETHREAD_PROTOCOL 生成了两个线程，加上主线程，共三个线程。使用 EFI_SIMPLETHREAD_PROTOCOL 分为三步：

1）得到 EFI_SIMPLETHREAD_PROTOCOL 服务。

2）调用 create_thread 服务生成线程。

3）调用 thread_join 服务等待线程结束并清理资源。

【示例 13-15】 线程示例。

```
//@file uefi/book/thread/TestSthread/TestSimpleThread.c
#include "SimpleThread.h"
UINTN gStep = 0;
void thread0(void* arg )
{
  UINTN step = 0;
  while(step<20) {
    Print(L"THREAD 0 : step%d ;globalStep:%d\n", step++, gStep++);
    gBS->Stall(1000* 1000);
  }
}
void thread1(void*arg )
{
  UINTN step = 0;
  while(step < 6 + (UINTN)arg * 20) {
    Print(L"This is thread %d : step%d; global Step:%d\n", arg, step++, gStep++);
    gBS->Stall(1000 * 1000);
  }
}
```

```
EFI_GUID gEfiSimpleThreadProtocolGUID = EFI_SIMPLETHREAD_PROTOCOL_GUID;
EFI_STATUS EFIAPI
UefiMain(IN EFI_HANDLE ImageHandle, IN EFI_SYSTEM_TABLE *SystemTable)
{
  EFI_STATUS  Status;
  EFI_SIMPLETHREAD_PROTOCOL * SimpleThread;
  //获得 Simple Thread Protocol 实例
  Status = gBS->LocateProtocol(&gEfiSimpleThreadProtocolGUID , 0,(VOID**)&SimpleThread);
  //生成线程 1
  SimpleThread ->  create_thread(SimpleThread, thread1, (void*) 1);
  //生成线程 2
  SimpleThread ->  create_thread(SimpleThread, thread1, (void*) 2);
  //主线程执行函数 thread0(0)
  thread0(0);
  //等待所有线程结束
  SimpleThread ->  thread_join(SimpleThread );
  return 0;
}
```

测试时，首先通过命令“load SimpleThread.efi”加载 Simple Thread 服务，然后执行测试程序。上述代码将会有如下输出。

```
FS0:/> load SimpleThread.efi
FS0:/> TestSimpleThread.efi
THREAD 0 :              step1; global Step:1
THREAD 0 :              step2; global Step:2
THREAD 0 :              step3; global Step:3
THREAD 0 :              step4; global Step:4
THREAD 0 :              step5; global Step:5
This is thread 1 :      step1; global Step:6
This is thread 1 :      step2; global Step:7
This is thread 1 :      step3; global Step:8
This is thread 1 :      step4; global Step:9
This is thread 1 :      step5; global Step:10
This is thread 2 :      step1; global Step:11
This is thread 2 :      step2; global Step:12
This is thread 2 :      step3; global Step:13
This is thread 2 :      step4; global Step:14
This is thread 2 :      step5; global Step:15
THREAD 0 :              step6; global Step:16
THREAD 0 :              step7; global Step:17
THREAD 0 :              step8; global Step:18
THREAD 0 :              step9; global Step:19
THREAD 0 :              step10; global Step:20
This is thread 1 :      step6; global Step:21
This is thread 1 :      step7; global Step:22
This is thread 1 :      step8; global Step:23
This is thread 1 :      step9; global Step:24
```

```
This is thread 1 :        step10; global Step:25
This is thread 2 :        step6;  global Step:26
This is thread 2 :        step7;  global Step:27
This is thread 2 :        step8;  global Step:28
...
```

13.4　本章小结

本章介绍了如何在 UEFI 环境下执行多任务，主要内容包括：

1）如何使用多核执行任务。

多核中的 CPU 分为 BSP 和 AP 两种。BSP 只有一个，用于启动系统。AP 可以有很多个，系统启动后执行 BSP 分配的任务。EFI_MP_SERVICES_PROTOCOL 提供了在 AP 上执行任务的服务。

2）启动多核的过程和原理。

3）利用时钟中断实现简单的多线程。

4）简单介绍了多线程的原理。利用时钟中断及 SetJump/LongJump 实现了线程的切换。

下一章我们将介绍如何开发网络应用。

Chapter 14

第 14 章

网络应用开发

UEFI 支持从网络设备启动。为了支持网络启动，UEFI 提供了支持 TCP/IP 的网络协议栈，整个网络协议栈如图 14-1 所示。

在本章中，“协议”与 UEFI 中的 Protocol 作为两个不同的术语使用。“协议”指代网络协议栈中的“协议”。例如，Protocol EFI_IP4_PROTOCOL 实现了 IP。

1）链路层有协议 ARP、MNP、SNP，以及网卡驱动。

- ARP（EFI_ARP_PROTOCOL）用于将 IP 地址转换为物理 MAC 地址。
- MNP（EFI_MANAGED_NETWORK_PROTOCOL）提供异步数据包 I/O 操作。
- SNP（EFI_SIMPLE_NETWORK_PROTOCOL）用于初始化和关闭网络接口，将网络数据帧提交给网络接口传输到目的地址，从网络接口接收网络数据帧。发送方和接收方使用 MAC 地址。
- 网卡驱动是 UEFI 网络协议栈中必不可少的部分，一般应由网卡提供商提供。

2）网络层协议有 IP。

IP（EFI_IP4_PROTOCOL/EFI_IP6_PROTOCOL）用于点到点的主机间（每个逻辑主机用一个 IP 地址标识）传输数据。传输过程如下：

3）从传输层得到数据段（segment）后，在数据前添加 IP 头生成数据报（datagram）。

然后根据目的 IP 地址，查询路由表得到下一跳地址（如果目的地址与本机处于同一网段，则下一跳地址为目的地址）。

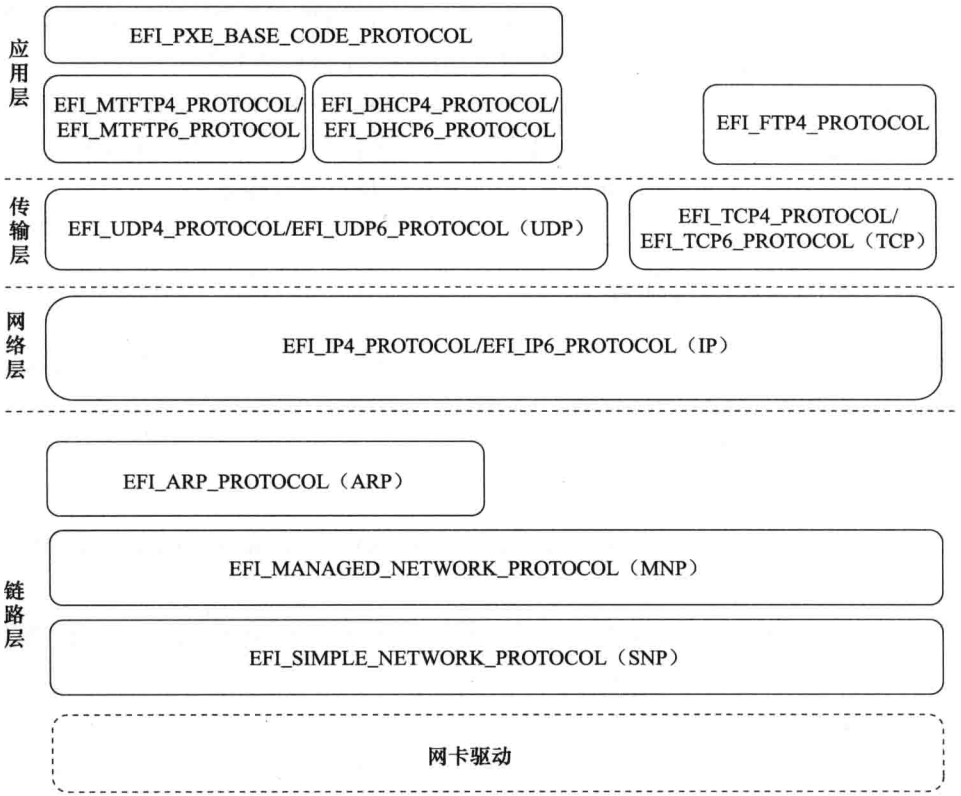


图 14-1　整个网络协议栈

通过 ARP 得到下一跳 IP 地址的 MAC 地址，将数据报交给下层的 MNP，由 MNP 发送数据给指定的 MAC 地址。

传输层的协议有 TCP 和 UDP，这两个协议提供了端到端的数据传输（“端”由 IP 地址 + 端口表示）。

- TCP（EFI_TCP4_PROTOCOL/EFI_TCP6_PROTOCOL）是面向连接的、可靠的数据传输协议，它保证数据包流按正确的顺序被目的方接收，确保数据包成功送到目的地。
- UDP（EFI_UDP4_PROTOCOL/EFI_UDP6_PROTOCOL）提供无连接的、不可靠的数据报投递服务。它不负责数据包流能按正确的顺序接收、不确保数据包能成功地被目的方接收。UDP 适合少量数据的传输。

4）应用层的协议有 MTFTP 和 DHCP。

- MTFTP（EFI_MTFTP4_PROTOCOL/EFI_MTFTP6_PROTOCOL）是建立在 UDP 之上的多播小型文件传输协议。UEFI 用该协议下载远程启动文件。
- DHCP（EFI_DHCP4_PROTOCOL/EFI_DHCP6_PROTOCOL）是动态主机配置协议，用于为 IP 提供配置信息，提供发现网络启动服务器的服务。

EFI_PXE_BASE_CODE_PROTOCOL 是建立在 MTFTP、DHCP、UDP 之上的服务型 Protocol，启动管理器利用这个服务发现网络启动设备、下载启动文件。

对应用程序而言，主要使用 TCP、UDP 等传输层协议，MTFTP、DHCP 等应用层协议，以及 EFI_PXE_BASE_CODE_PROTOCOL 服务。下面以 EFI_TCP4_PROTOCOL 为例介绍 UEFI 网络协议的用法。在介绍 TCP 之前，先要配置好网络开发环境。

14.1 在 UEFI 中使用网络

开始编程之前，首先要配置好开发环境。开发环境分两种情况，一种是使用 Nt32 模拟环境，另一种是使用真实的 UEFI 环境。

1. 在 Nt32 模拟器中使用网络

Nt32 网络设置可以参考 UEFI Network Stack for EDK Getting Started Guide，简略说明如下。

1）下载并安装 Winpcap。

Winpcap 用于从网络设备抓取所有以太网帧，它在模拟器中的作用相当于网卡驱动。可以从如下网址下载 Winpcap：http://www.winpcap.org/。

2）下载 SnpNt32Io 源码并编译。

SnpNt32Io 源码可以从如下网址下载：http://tianocore.sourceforge.net/wiki/Network-io。下载后将之解压到 C:\ NetNt32Io，该目录下将会有 Makefile 文件。然后下载 Winpcap 开发包（下载地址为 http://www.winpcap.org/devel.htm），将 WpdPack 文件夹复制到 C:\NetNt32Io 目录。

用如下命令编译 SnpNt32Io。

```
c:\> cd c:\NetNt32Io
c:\ NetNt32Io> nmake TARGET=RELEASE
```

将编译后的动态链接库 SnpNt32Io.dll 复制到 UEFI 模拟器的根目录，命令如下所示。

```
c:\SnpNt32Io>  copy /y c:\ NetNt32Io\release\SnpNt32Io.dll   c:\edk2\build\Nt32Pkg\vs
    2008\IA32\
```

3）启动 Nt32 模拟器，命令如下所示。

```
c:\EDK2> edksetup.bat --nt32
c:\EDK2> build run
```

4）进入 UEFI Shell 后，加载网络协议。

```
Shell > fsnt0:
fsnt0:\>load SnpNt32Dxe.efi MnpDxe.efi ArpDxe.efi Ip4Dxe.efi Ip4ConfigDxe.efi
Udp4Dxe.
```

```
efi Dhcp4Dxe.efi Mtftp4Dxe.efi Tcp4Dxe.efi
```

5）配置网卡。

可以通过 ifconfig 命令设置动态 IP 地址，命令如下所示。

```
fsnt0:\> ifconfig -s eth0 DHCP
```

也可以通过如下命令配置静态 IP 地址。

```
fsnt0:\> ifconfig -s eth0 static 192.168.0.125 255.255.255.0 192.168.0.1
```

其中，192.168.0.125 是本机的 IP 地址，255.255.255.0 是子网掩码，192.168.0.1 是网关 IP 地址。

6）测试网卡

通过 ping 命令测试网络，命令如下所示。

```
fsnt0:\>ping 192.168.0.1
```

2. 真实 UEFI 环境下使用网络

如果要在真实的 UEFI 环境下使用网络，首先要加载网卡驱动和网络协议。网卡驱动可以从英特尔网站下载。将下载后位于 APPS/EFI/EFIx64 目录下的驱动文件复制到 U 盘 \efi\bootX64H 目录下。然后，按如下步骤配置网络。

1）进入 UEFI Shell，并进入 \efi\boot\X64 目录，所执行的命令如下所示。

```
Shell > fsnt0:
fsnt0:\>cd efi\boot\X64
```

2）加载网卡驱动。根据网卡类型选择驱动，E3522X2.EFI 对应 PCI 千兆网卡，E6327X3.EFI 对应 PCI-E 千兆网卡，E4431X4.EFI 对应 10Gbit/s 网卡。然后，使用 load 命令加载该驱动，命令如下所示。

```
fsnt0:\>load E3522X2.EFI
```

3）加载网络协议。

```
Shell > fsnt0:
fsnt0:\>load SnpDxe.efi MnpDxe.efi ArpDxe.efi Ip4Dxe.efi Ip4ConfigDxe.efi
Udp4Dxe.efi
Dhcp4Dxe.efi Mtftp4Dxe.efi Tcp4Dxe.efi
```

4）配置网卡。

可以通过 ifconfig 命令设置动态 IP 地址，命令如下所示。

```
fsnt0:\> ifconfig -s eth0 DHCP
```

也可以通过如下命令配置静态 IP 地址。

```
fsnt0:\> ifconfig -s eth0 static 192.168.0.125 255.255.255.0 192.168.0.1
```

其中，192.168.0.125 是本机的 IP 地址，255.255.255.0 是子网掩码，192.168.0.1 是网关 IP 地址。

5）测试网卡。

通过 ping 命令测试网络，命令如下所示。

```
fsnt0:\>ping 192.168.0.1
```

网络配置成功后，就可以利用 UEFI 的网络协议栈开发网络应用了。

14.2 使用 EFI_TCP4_PROTOCOL

下面我们介绍一下 TCP Protocol(EFI_TCP4_PROTOCOL) 的用法。如果读者曾经用 Socket 写过程序，那么会很容易理解 EFI_TCP4_PROTOCOL 的用法。

1. 使用 Socket 客户端的流程

在 Windows 或 Linux 下使用 Socket 客户端，需要如下几步：

1）生成 Socket 对象。

2）向服务器发起连接。

3）传输数据。

4）关闭 Socket。

UEFI 中的 EFI_TCP4_PROTOCOL 与之相似，使用客户端时需要如下几步：

1）生成 EFI_TCP4_PROTOCOL 对象。

2）配置生成的 EFI_TCP4_PROTOCOL 对象。

3）发起连接。

4）传输数据。

5）关闭 EFI_TCP4_PROTOCOL 对象。

其实我们可以把配置（Configure）和连接（Connect）看成一步。

对于 Windows 或 Linux 下的 Socket 服务端，下面是常用的处理流程。

```
Socket* server = new Socket(…)
   while ( Socket connection = server->Listen(…)){
     // 侦听到新连接时产生新的 Socket 对象 connection
  // 异步处理 connection
```

```
  }
Server->Close();
```

与之类似，EFI_TCP4_PROTOCOL 服务端处理流程如下。

```
EFI_TCP4_PROTOCOL* TCP = …;
TCP->Configure(…);
while (TCP.Accept(…)){ //  侦听到新连接时产生新的 EFI_TCP4_PROTOCOL 对象
  //异步处理新生成的 EFI_TCP4_PROTOCOL 对象
}
TCP->Close();
```

下面介绍一下 EFI_TCP4_PROTOCOL 提供的服务。代码清单 14-1 列出了该 Protocol 的结构体。

代码清单 14-1　EFI_TCP4_PROTOCOL 结构体

```
typedef struct _EFI_TCP4_PROTOCOL {
  EFI_TCP4_GET_MODE_DATA GetModeData;   //获取网络协议栈当前 TCP、IP、MNP、SNP 状态
  EFI_TCP4_CONFIGURE  Configure;        //设置 TCP 地址、端口、连接属性等
  EFI_TCP4_ROUTES  Routes;              //添加或删除此 TCP 的路由
  EFI_TCP4_CONNECT Connect;             //建立 TCP 连接
  EFI_TCP4_ACCEPT  Accept;              //侦听端口，接受连接，此函数为异步函数
  EFI_TCP4_TRANSMIT  Transmit;          //发送数据
  EFI_TCP4_RECEIVE  Receive;            //接收数据
  EFI_TCP4_CLOSE  Close;                //关闭连接
  EFI_TCP4_CANCEL  Cancel;              //取消当前连接上的异步操作
  EFI_TCP4_POLL  Poll;                  //完成当前连接上的发送或接收操作
} EFI_TCP4_PROTOCOL;
```

本章将会讲述 Configure、Connect、Transmit、Receive、Close 这 5 个网络应用客户端常用的函数。不过，对 C 程序员来说，Socket 接口更容易被接受。下面我们把 EFI_TCP4_PROTOCOL 封装成 Socket 接口，并以之为例详细解释 EFI_TCP4_PROTOCOL 的用法。

2. 利用 Socket 接口使用 EFI_TCP4_PROTOCOL

下面我们将要讲述如何实现 Socket 接口中的 5 个函数，这 5 个函数包括 int Socket()、Connect、Send、Recv 及 Close，示例 14-1 列出了这 5 个函数的函数原型。通过它们的函数名字，读者可以比较容易地猜出它们的作用。为了将重点放在 EFI_TCP4_PROTOCOL 的使用上，这 5 个函数都设计为阻塞函数并且仅能使用 TCP，以简化叙述。

下面的讲述中将会用到如下三个概念：Socket 接口、Socket 对象及 int Socket() 函数。为了避免引起混淆，这里阐述一下它们的区别。Socket 接口由一组函数组成，这组函数用于操作 Socket 对象。Socket 对象由网络协议栈组成，通过 Socket 对象可以完成网络包的发送和

接收。int Socket() 函数是 Socket 接口中的一个函数，用于创建新的 Socket 对象。

【示例 14-1】 Socket 函数原型。

```
//@file uefi\book\Network\socket.h
#define IPV4(a,b,c,d) (a | b<<8 | c << 16 | d <<24)
typedef EFI_STATUS SOCKET_STATUS;
//生成 Socket 对象并返回 Socket 的描述符
int Socket();
//与远程服务器建立 TCP 连接
SOCKET_STATUS Connect(int fd, UINT32 Ip32, UINT16 Port);
//通过 fd 表示的 TCP 连接向服务器发送数据
SOCKET_STATUS Send(int fd, CHAR8* Data, UINTN Lenth);
//接收远程服务器发到 fd 的数据
SOCKET_STATUS Recv(int fd, CHAR8* Buffer, UINTN Lenth);
//关闭 Socket 对象
SOCKET_STATUS Close(int fd);
```

下面一步步来看如何利用 EFI_TCP4_PROTOCOL 实现这 5 个函数。

14.2.1 生成 Socket 对象

函数 Socket 用于生成 Socket 对象。这个函数的功能主要是产生 Socket 对象并返回 Socket 对象的 ID（或称为描述符），此 ID 将作为该 Socket 对象的句柄。若返回值小于 0，则返回值表示错误代码。

1. Socket 结构体

我们定义了 struct Socket 这个结构体，实例化这个结构体就可以得到一个 Socket 对象。如示例 14-2 所示。

说明：此处的 struct Socket 是我们自定义的数据结构，不是 EDK2 中的结构体。Socket 是一个极易产生冲突的名字，程序开发中应尽量避免直接作为变量、结构体或函数名。此处使用是为了方便讲述 TCP 的用法。

【示例 14-2】 Socket 结构体。

```
//@file uefi\book\Network\socket.c
struct Socket{
  EFI_HANDLE  m_SocketHandle;                    //m_pTcp4Protocol 存在于该句柄上
  EFI_TCP4_PROTOCOL* m_pTcp4Protocol;
  EFI_TCP4_CONFIG_DATA* m_pTcp4ConfigData;       //用于配置 m_pTcp4Protocol
  EFI_TCP4_TRANSMIT_DATA* m_TransData;           //用于发送数据
  EFI_TCP4_RECEIVE_DATA* m_RecvData;             //用于接收数据
  EFI_TCP4_CONNECTION_TOKEN ConnectToken;
  EFI_TCP4_CLOSE_TOKEN CloseToken;
  EFI_TCP4_IO_TOKEN  SendToken, RecvToken;
};
static struct Socket* Socketfd[32];              //下标表示 Socket 对象的描述符
```

2. 如何生成 EFI_TCP4_PROTOCOL 实例

Socket 对象中最重要的成员自然是 EFI_TCP4_PROTOCOL 实例，有了 EFI_TCP4_PROTOCOL 实例，这个 Socket 对象才能发挥作用。那么如何得到一个 EFI_TCP4_PROTOCOL 实例呢？在一台计算机中，可能同时存在成百上千个 TCP 连接，这些连接分时复用网络设备并且不能相互干扰。在 UEFI 系统中，每一个 TCP 连接都由一个独立的 EFI_TCP4_PROTOCOL 实例控制。对开发者而言，要建立一个 TCP 连接，首先要生成一个 EFI_TCP4_PROTOCOL 实例。BS 中的 LocateProtocol、HandleProtocol 及 OpenProtocol 仅能用于得到已存在于系统中的 Protocol 实例的指针，而不能生成新的 Protocol 实例。如何生成新的 EFI_TCP4_PROTOCOL 实例呢？ UEFI 为 EFI_TCP4_PROTOCOL 提供了一个 EFI_SERVICE_BINDING_PROTOCOL 实例（用 gEfiTcp4ServiceBindingProtocolGuid 标识），利用这个 Protocol 可以创建和销毁一个 EFI_TCP4_PROTOCOL 实例。gEfiTcp4ServiceBindingProtocolGuid 的 CreateChild 服务用于产生一个新的虚拟句柄，此句柄上挂载了 EFI_TCP4_PROTOCOL，用 OpenProtocol 或 LocateProtocol 可以获得此句柄上的 EFI_TCP4_PROTOCOL 对象。

代码清单 14-2 列出了 EFI_SERVICE_BINDING_PROTOCOL 的结构体及成员函数的原型，该 Protocol 有两个成员函数：CreateChild 和 DestroyChild。CreateChild 用于生成子设备并在该设备上安装对应的 Protocol，TCP4 对应的 EFI_SERVICE_BINDING_PROTOCOL 的 CreateChild 函数生成的子设备上会安装 EFI_TCP4_PROTOCOL。DestroyChild 用于销毁生成的子设备。EFI_SERVICE_BINDING_PROTOCOL 相当于设计模式中的工厂方法。

代码清单 14-2　EFI_SERVICE_BINDING_PROTOCOL 结构体及成员函数原型

```
//CreateChild，用于生成子设备并初始化该子设备
typedef EFI_STATUS(EFIAPI *EFI_SERVICE_BINDING_CREATE_CHILD)(
  IN EFI_SERVICE_BINDING_PROTOCOL *This,
  IN OUT EFI_HANDLE *ChildHandle
);
//DestroyChild,  销毁子设备
typedef EFI_STATUS(EFIAPI *EFI_SERVICE_BINDING_DESTROY_CHILD)(
  IN EFI_SERVICE_BINDING_PROTOCOL *This,
  IN EFI_HANDLE ChildHandle
  );
struct _EFI_SERVICE_BINDING_PROTOCOL {
  EFI_SERVICE_BINDING_CREATE_CHILD CreateChild;
  EFI_SERVICE_BINDING_DESTROY_CHILD DestroyChild;
};
```

3. Socket() 函数

在 int Socket() 函数中，每生成一个 Socket 对象，将其指针放入 Socketfd 数组中，下

标作为该 Socket 对象的描述符被该函数返回。该函数最主要的功能是生成 EFI_TCP4_PROTOCOL 的实例。生成 EFI_TCP4_PROTOCOL 的实例是通过 TCP4 对应的 EFI_SERVICE_BINDING_PROTOCOL 完成的。

下面来看 int Socket() 函数的具体实现，如示例 14-3 所示。它主要有以下三部分：

1）为 Socket 对象分配描述符，并为该对象分配内存。

2）为生成的 Socket 对象分配网络资源，主要是生成 EFI_TCP4_PROTOCOL 实例。

- 打开 TCP4 的 EFI_SERVICE_BINDING_PROTOCOL。
- 利用 EFI_SERVICE_BINDING_PROTOCOL 的 CreateChild 服务生成子虚拟设备句柄。
- 打开所生成句柄上的 EFI_TCP4_PROTOCOL。

3）初始化 Socket 对象中除 m_SocketHandle（Socket 对象的描述符）、m_pTcp4Protocol（EFI_TCP4_PROTOCOL 的实例）之外的变量。

【示例 14-3】 Socket 函数。

```
//产生 Socket 对象，初始化 Socket 对象，并返回 Socket 对象的 ID
int Socket()
{
  EFI_STATUS Status;
  EFI_SERVICE_BINDING_PROTOCOL* pTcpServiceBinding;
  struct Socket* this = NULL;
  int myfd = -1;
  //从 Socketfd 数组中分配 Socket 对象
  {
    int i;
    for(i =0;i<32; i++){
      if(Socketfd[i] == NULL){
        Socketfd[i] = this = ( struct Socket*) malloc(sizeof(struct Socket));
    myfd = i;
    break;
    }
  }
 }
 if(this == NULL){
   return myfd;
 }
 memset((void*)this, 0, sizeof(struct Socket));
 this->m_SocketHandle = NULL;
 //为生成的 Socket 对象分配网络资源，分为三步
 //第一步：打开 TCP4 的 EFI_SERVICE_BINDING_PROTOCOL
 Status = gBS->LocateProtocol ( &gEfiTcp4ServiceBindingProtocolGuid,
     NULL, (VOID **)& pTcpServiceBinding);
 if(EFI_ERROR(Status))
   return Status;
 //第二步：生成 TCP 虚拟设备
```

```
  Status = pTcpServiceBinding ->CreateChild (pTcpServiceBinding,
&this->m_SocketHandle );
  if(EFI_ERROR(Status))
    return Status;
  //第三步：打开子设备上的 TCP4 Protocol
  Status = gBS->OpenProtocol ( this->m_SocketHandle, &gEfiTcp4ProtocolGuid,
       (VOID **)&this->m_pTcp4Protocol, gImageHandle, this->m_SocketHandle,
       EFI_OPEN_PROTOCOL_BY_HANDLE_PROTOCOL );
  if(EFI_ERROR(Status))
    return Status;
  //初始化 Socket 对象
  Initialize(myfd);
  return myfd;
}
```

Initialize 函数用于初始化 struct Socket 中除 m_SocketHandle、m_pTcp4Protocol 之外的变量，其主要工作有：

1）为 Configure 服务分配内存。

2）为 Connect 服务初始化 Token。

3）为 Transmit 服务分配内存，初始化 Token。

4）为 Receive 服务分配内存，初始化 Token。

5）为 Close 服务初始化 Token。

Initialize 函数的代码如示例 14-4 所示。

【示例 14-4】 Initialize 函数。

```
static SOCKET_STATUS Initialize(int sk)
{
  EFI_STATUS Status;
  struct Socket* this = Socketfd[sk];
  //建立 Configure Data
  this->m_pTcp4ConfigData = (EFI_TCP4_CONFIG_DATA*) malloc(sizeof(EFI_TCP4_CONFIG_DATA));
  //建立 Connect Data
  this->ConnectToken.CompletionToken.Status = EFI_ABORTED;
  Status = gBS->CreateEvent(EVT_NOTIFY_SIGNAL, TPL_CALLBACK,
    (EFI_EVENT_NOTIFY)NopNoify , (VOID*)&this->ConnectToken,
    &this->ConnectToken.CompletionToken.Event );
  if(EFI_ERROR(Status)) return Status;
  //建立 Transmit Data
  Status = gBS->CreateEvent(EVT_NOTIFY_SIGNAL, TPL_CALLBACK,
(EFI_EVENT_NOTIFY)NopNoify , (VOID*)&this->SendToken,
&this->SendToken.CompletionToken.Event);
  if(EFI_ERROR(Status)) return Status;
  this->SendToken.CompletionToken.Status  =EFI_ABORTED;
  this->m_TransData = (EFI_TCP4_TRANSMIT_DATA*)malloc(sizeof(EFI_TCP4_TRANSMIT_DATA));;
```

```
    //建立 Recv Data
  Status = gBS->CreateEvent(EVT_NOTIFY_SIGNAL, TPL_CALLBACK,
    (EFI_EVENT_NOTIFY)NopNoify , (VOID*)&this->RecvToken,
    &this->RecvToken.CompletionToken.Event);
  this->RecvToken.CompletionToken.Status  =EFI_ABORTED;
  this->m_RecvData = (EFI_TCP4_RECEIVE_DATA*) malloc(sizeof(EFI_TCP4_RECEIVE_DATA));
  if(EFI_ERROR(Status)) return Status;
//建立 Close Data
  this->CloseToken.CompletionToken.Status = EFI_ABORTED;
  Status = gBS->CreateEvent(EVT_NOTIFY_SIGNAL, TPL_CALLBACK,
    (EFI_EVENT_NOTIFY)NopNoify , (VOID*)&this->CloseToken,
    &this->CloseToken.CompletionToken.Event );
  return Status;
}
```

14.2.2 连接

生成 Socket 后，就可以向服务器发起连接了，该功能由 SOCKET_STATUS Connect(int fd, UINT32 Ip32, UINT16 Port) 实现。此函数主要有两个功能，一是配置 Socket，二是通过配置好的 Socket 发起连接。示例 14-5 展示了 Connect 函数的代码。

【示例 14-5】 Socket 的 Connect 函数。

```
SOCKET_STATUS Connect(int fd, UINT32 Ip32, UINT16 Port)
{
  Config(fd, Ip32, Port);
  return Connect0(fd);
}
```

Config(fd, Ip32, Port) 用于设置服务端 IP 地址和端口，本地端 IP 地址和端口，其功能是通过调用 EFI_TCP4_PROTOCOL 的 Configure 服务实现的。需要注意的是，Configure 完成之后，连接还没有建立。代码清单 14-3 展示了 Configure 服务的函数原型。

代码清单 14-3　EFI_TCP4_PROTOCOL 的 Configure 服务的函数原型

```
typedef EFI_STATUS(EFIAPI *EFI_TCP4_CONFIGURE)(
  IN EFI_TCP4_PROTOCOL *This,
  IN EFI_TCP4_CONFIG_DATA *TcpConfigData OPTIONAL
  );
```

在 Configure 服务中，配置数据的类型是 EFI_TCP4_CONFIG_DATA，它的结构体如代码清单 14-4 所示。

代码清单 14-4　EFI_TCP4_CONFIG_DATA 结构体

```
typedef struct {
  UINT8 TypeOfService;          //服务类型，将填充到 IP 报头的第二字节（服务类型字段）
  UINT8 TimeToLive;             //生存期，将填充到 IP 报头的生存期字段
```

```
  EFI_TCP4_ACCESS_POINT AccessPoint;          //本地和服务器端的 IP 地址和端口
  EFI_TCP4_OPTION * ControlOption;            //TCP 控制选项，若为 NULL，则使用默认选项
} EFI_TCP4_CONFIG_DATA;
typedef struct {
  BOOLEAN UseDefaultAddress;                  //True 表示使用本机默认 IP 地址；False 则要指定
StationAddress
  EFI_IPv4_ADDRESS StationAddress;            //本地 IP 地址
  EFI_IPv4_ADDRESS SubnetMask;                //本地 IP 地址的子网掩码
  UINT16 StationPort;                         //本地端口
  EFI_IPv4_ADDRESS RemoteAddress;             //服务端 IP 地址
  UINT16 RemotePort;                          //服务端端口
  BOOLEAN ActiveFlag;                 //TRUE：主动模式（用于客户端）；False：被动模式（用于服务端）
} EFI_TCP4_ACCESS_POINT
```

通常，TypeOfService 可以设置为 0，表示普通服务。TimeToLive 表示的生存期是数据报最多被路由器转发的次数，通常可设置为 16。ControlOption 可设为 NULL，表示使用默认选项。客户端通常将 ActiveFlag 设为 TRUE。Config 函数的代码如示例 14-6 所示。

【示例 14-6】 Config 函数。

```
static SOCKET_STATUS Config(int sk, UINT32 Ip32, UINT16 Port)
{
  EFI_STATUS  Status = EFI_NOT_FOUND;
  struct Socket* this = Socketfd[sk];
  if(this->m_pTcp4ConfigData == NULL) return Status;
  this->m_pTcp4ConfigData->TypeOfService = 0;  //普通类型
  this->m_pTcp4ConfigData->TimeToLive = 16;    //常规值
  //服务器 IP 地址和端口
  *(UINTN*)(this->m_pTcp4ConfigData->AccessPoint.RemoteAddress.Addr) = Ip32;
  this->m_pTcp4ConfigData->AccessPoint.RemotePort = Port;
  *(UINT32*)(this->m_pTcp4ConfigData->AccessPoint.SubnetMask.Addr) =
(255 | 255 << 8 | 255 << 16 | 0 << 24);
  //使用本机默认 IP 地址，使用 61558 端口
  this->m_pTcp4ConfigData->AccessPoint.UseDefaultAddress = TRUE;
  this->m_pTcp4ConfigData->AccessPoint.StationPort = 61558;
  this->m_pTcp4ConfigData->AccessPoint.ActiveFlag = TRUE;
  //默认控制选项
  this->m_pTcp4ConfigData->ControlOption = NULL;
  Status = this->m_pTcp4Protocol ->Configure(this->m_pTcp4Protocol,
           this->m_pTcp4ConfigData);
  return Status;
}
```

配置成功后，就可以发起连接了。发起连接是通过 EFI_TCP4_PROTOCOL 的 Connect 服务完成的。Connect 服务的函数原型及该服务所使用的 EFI_TCP4_CONNECTION_TOKEN 的结构体如代码清单 14-5 所示。

代码清单 14-5 Connect 服务的函数原型及 EFI_TCP4_CONNECTION_TOKEN 结构体

```
typedef EFI_STATUS(EFIAPI *EFI_TCP4_CONNECT) (IN EFI_TCP4_PROTOCOL *This,
  IN EFI_TCP4_CONNECTION_TOKEN *ConnectionToken,
);
typedef struct {
  EFI_EVENT Event;
  EFI_STATUS Status;
} EFI_TCP4_COMPLETION_TOKEN;
typedef struct {
  EFI_TCP4_COMPLETION_TOKEN CompletionToken;
} EFI_TCP4_CONNECTION_TOKEN;
```

Connect 是非阻塞函数，调用后立即返回。连接完成（成功或失败）后，系统会触发 ConnectionToken 中的事件并设置 Connect 操作的状态 Status，因此，我们要在适当的时机查询或等待该事件。为了简化示例程序，我们在调用 Connect 服务后，立即调用 BS 的 WaitForEvent 服务等待对应的事件。

Socket 的 Connect0 函数用于调用 EFI_TCP4_PROTOCOL 的 Connect 服务发起连接，其代码如示例 14-7 所示。该函数流程为：

1）调用 EFI_TCP4_PROTOCOL 的 Connect 服务。

2）等待事件 ConnectToken.CompletionToken.Event。

3）检查 ConnectToken.CompletionToken.Status 以获取 Connect 操作的状态。

【示例 14-7】 Socket 的 Connect0 函数。

```
EFI_STATUS Connect0(int sk)
{
  EFI_STATUS  Status = EFI_NOT_FOUND;
  struct Socket* this = Socketfd[sk];
  if(this->m_pTcp4Protocol == NULL) return Status;
  Status=this->m_pTcp4Protocol->Connect(this->m_pTcp4Protocol,
    &this->ConnectToken);  //发起连接
  if(EFI_ERROR(Status)) return Status;
  SocketWait(this->ConnectToken.CompletionToken.Event);        //等待连接成功或失败
  if( !EFI_ERROR(Status)){
    Status = this->ConnectToken.CompletionToken.Status;        //Connect 的状态
  }
  return Status;
}
```

14.2.3 传输数据

建立连接后就可以利用这个 TCP 连接传输数据了。

1. 发送数据

先来看如何发送数据。发送数据需要使用 EFI_TCP4_PROTOCOL 的 Transmit 服务，其函数原型如代码清单 14-6 所示。

代码清单 14-6　EFI_TCP4_PROTOCOL 的 Transmit 服务的函数原型

```
typedef EFI_STATUS(EFIAPI *EFI_TCP4_TRANSMIT) (IN EFI_TCP4_PROTOCOL *This,
  IN EFI_TCP4_IO_TOKEN *Token
);
```

Transmit 是异步操作，要传输的数据放在 EFI_TCP4_IO_TOKEN 中传递给该操作。传输完成后（成功或失败），Token->CompletionToken.Event 被触发，Token->CompletionToken.Status 设置为该操作的状态。EFI_TCP4_IO_TOKEN 比 EFI_TCP4_CONNECTION_TOKEN 复杂很多，其结构体如代码清单 14-7 所示。

代码清单 14-7　EFI_TCP4_IO_TOKEN 结构体

```
typedef struct {
  EFI_TCP4_COMPLETION_TOKEN CompletionToken;
  union {
    EFI_TCP4_RECEIVE_DATA *RxData;
    EFI_TCP4_TRANSMIT_DATA *TxData;
  } Packet;
} EFI_TCP4_IO_TOKEN;
typedef struct {
  BOOLEAN Push;                                  // 对应 TCP 头中的 PSH 位
  BOOLEAN Urgent;                                // 对应 TCP 头中的 URG 位
  UINT32 DataLength;                             // 数据总长度
  UINT32 FragmentCount;                          // 数据段数
  EFI_TCP4_FRAGMENT_DATA FragmentTable[1];       // 数据段数组
} EFI_TCP4_TRANSMIT_DATA;
typedef struct {
  UINT32 FragmentLength;
  VOID *FragmentBuffer;
} EFI_TCP4_FRAGMENT_DATA;
```

发送数据之前，调用者需填充 Token 中的 TxData 域。TxData 包含了待发送的数据。待发送数据可能在几个不连续的缓冲区内，我们可以将这些缓冲区指针放到 FragmentTable 数组内，数组中每个元素表示一个缓冲区。FragmentCount 是 FragmentTable 中的缓冲区个数。DataLength 是数据总长度（FragmentTable 中各个缓冲区长度之和）。

Push 是 PSH 标志，若该标志为 1，则数据被立刻发送，接收方收到带 PSH 标志的报文后会尽快提交给上层应用，而不是等待缓冲区填满再提交给上层应用。否则，该数据可能会与后续的 Transmit 操作的数据合并后发送以提高效率。

若 Urgent 为 TRUE，则 FragmentTable 中的数据为紧急数据。

为了简化说明，Send 函数中只用到了一个缓冲区，即 FragmentTable 数组长度为 1。若 FragmentTable 长度大于 1，创建 TxData 时要按如下方式分配内存：

malloc（sizeof（EFI_TCP4_TRANSMIT_DATA）+（数据段数 -1）*sizeof（EFI_TCP4_FRAGMENT_DATA））

同 Connect 一样，调用 Transmit 之后要通过 WaitForEvent 等待发送完成。

Send 函数的源码如示例 14-8 所示，其执行流程为：

- 设置 Token 中的 TxData，主要是设置其中的缓冲区及发送数据长度。
- 调用 Transmit 服务发送数据。
- 等待 Token 中的事件。
- 检查 Token 中状态值，以确定发送是否成功。

【示例 14-8】 Send 函数。

```
SOCKET_STATUS Send(int sk, CHAR8* Data, UINTN Lenth)
{
  EFI_STATUS Status = EFI_NOT_FOUND;
  struct Socket* this = Socketfd[sk];
  if(this->m_pTcp4Protocol == NULL) return Status;
  this->m_TransData->Push = TRUE;
  this->m_TransData->Urgent = TRUE;
  this->m_TransData->DataLength = (UINT32)Lenth;
  this->m_TransData->FragmentCount = 1;
  this->m_TransData->FragmentTable[0].FragmentLength =
    this->m_TransData->DataLength;
  this->m_TransData->FragmentTable[0].FragmentBuffer =Data;
  this->SendToken.Packet.TxData=  this->m_TransData;
  Status = this->m_pTcp4Protocol->Transmit(this->m_pTcp4Protocol,
    &this->SendToken);
  if(EFI_ERROR(Status)) return Status;
  SocketWait(this->SendToken.CompletionToken.Event);
  return this->SendToken.CompletionToken.Status;
}
```

2. 接收数据

接收数据是通过 EFI_TCP4_PROTOCOL 的 Receive 服务完成的。Receive 与 Transmit 的接口相似，只是使用 Receive 时需设置 Token 中的 RxData 域。RxData 是指向 EFI_TCP4_RECEIVE_DATA 的指针。Receive 服务的函数原型及 EFI_TCP4_RECEIVE_DATA 的结构体如代码清单 14-8 所示。

代码清单 14-8　EFI_TCP4_PROTOCOL 的 Receive 服务的函数原型及 EFI_TCP4_RECEIVE_DATA 结构体

```
typedef EFI_STATUS(EFIAPI *EFI_TCP4_RECEIVE) (IN EFI_TCP4_PROTOCOL *This,
  IN EFI_TCP4_IO_TOKEN *Token
);
```

```
//EFI_TCP4_RECEIVE_DATA
typedef struct {
  BOOLEAN UrgentFlag;
  UINT32 DataLength;
  UINT32 FragmentCount;
  EFI_TCP4_FRAGMENT_DATA FragmentTable[1];
} EFI_TCP4_RECEIVE_DATA;
```

EFI_TCP4_RECEIVE_DATA 与 EFI_TCP4_TRANSMIT_DATA 的唯一区别是没有了 Push 标志。调用者负责分配和释放 FragmentTable 中的缓冲区。调用 Receive 服务时，DataLength 是缓冲区总长度。接收完成后，Token 中的事件被设置，数据依次复制到 FragmentTable 中的各个缓冲区内，DataLength 也设置为接收到的数据的长度。示例 14-9 演示了 Recv 函数是如何调用 Receive 服务接收数据的。为了简化程序，也便于理解，同样，在该示例中，EFI_TCP4_TRANSMIT_DATA 仅有一个缓冲区。

【示例 14-9】 Recv 函数。

```
SOCKET_STATUS Recv(int sk, CHAR8* Buffer, UINTN Lenth)
{
  EFI_STATUS Status = EFI_NOT_FOUND;
  struct Socket* this = Socketfd[sk];
  if(this->m_pTcp4Protocol == NULL) return Status;
  //设置 Token 中的接收缓冲区
  this->m_RecvData->DataLength = (UINT32)Lenth;
  this->m_RecvData->FragmentCount = 1;
  this->m_RecvData->FragmentTable[0].FragmentLength =
    this->m_RecvData->DataLength ;
  this->m_RecvData->FragmentTable[0].FragmentBuffer = (void*)Buffer;
  this->RecvToken.Packet.RxData=  this->m_RecvData;
  //调用 Receive 服务接收数据
  Status=this->m_pTcp4Protocol->Receive(this->m_pTcp4Protocol,
    &this->RecvToken);
  if(EFI_ERROR(Status)) return Status;
  //等待 Receive 操作完成
  SocketWait(this->RecvToken.CompletionToken.Event);
  //返回 Receive 操作的状态
  return this->RecvToken.CompletionToken.Status;
}
```

14.2.4　关闭 Socket

Socket 使用完毕后，要调用 Close 函数关闭 Socket。Close 函数的主要功能是断开连接，并回收资源，具体代码参加示例 14-10。其处理流程为：

1）调用 EFI_TCP4_PROTOCOL 的 Close 服务关闭 TCP 连接。

2）调用 pTcpServiceBinding->DestroyChild 服务销毁虚拟句柄 m_SocketHandle。

3）关闭所有用到的事件，释放内存。

4）释放 Socket 描述符。

【示例 14-10】 Socket 的 Close 函数。

```
SOCKET_STATUS Close(int sk)
{
  EFI_STATUS Status;
  struct Socket* this = Socketfd[sk];
  Status=this->m_pTcp4Protocol->Close(this->m_pTcp4Protocol,&this->CloseToken);
  Destroy(sk);                          //释放 Socket 对象中的资源
  free(this);                           //释放 Socket 本身占用的内存资源
  Socketfd[sk] = NULL;                  //释放 Socket 占用的 ID
  return Status;
}
static int Destroy(int sk)
{
  EFI_STATUS Status;
  struct Socket* this = Socketfd[sk];
  if(this->m_SocketHandle){
    EFI_SERVICE_BINDING_PROTOCOL*  pTcpServiceBinding;
    Status = gBS->LocateProtocol ( &gEfiTcp4ServiceBindingProtocolGuid,
              NULL, (VOID **)&pTcpServiceBinding );
    Status = pTcpServiceBinding->DestroyChild ( pTcpServiceBinding,
              this->m_SocketHandle );   //销毁 m_SocketHandle
  }
  //关闭所有事件，释放内存
  if(this->ConnectToken.CompletionToken.Event)
    gBS->CloseEvent(this->ConnectToken.CompletionToken.Event);
  if(this->SendToken.CompletionToken.Event)
    gBS->CloseEvent(this->SendToken.CompletionToken.Event);
  if(this->RecvToken.CompletionToken.Event)
    gBS->CloseEvent(this->RecvToken.CompletionToken.Event);
  if(this->m_pTcp4ConfigData){
    free(this->m_pTcp4ConfigData);
  }
  if(this->SendToken.Packet.TxData){
    free(this->SendToken.Packet.TxData);
    this->SendToken.Packet.TxData = NULL;
  }
  if(this->RecvToken.Packet.RxData){
    free(this->RecvToken.Packet.RxData);
    this->RecvToken.Packet.RxData = NULL;
  }
  return 0;
}
```

14.2.5 测试 Socket

下面我们将用前面实现的 Socket 接口中的 5 个函数来请求一个网页。HTTP 建立在 TCP

基础之上，请求网页是通过向服务器发送 HTTP 请求头实现的，具体代码如示例 14-11 所示。

【示例 14-11】 利用 Socket 接口请求网页。

```
EFI_STATUS TestNetwork (IN EFI_HANDLE ImageHandle)
{
  EFI_STATUS Status = 0;
  CHAR8 RequestData[]= "GET / HTTP/1.1\n"
                  "Host:localhost\nAccept:* / * \n"
                  "Connection:Keep-Alive\n\n";
  CHAR8 *RecvBuffer = (CHAR8*) malloc(1024);
  int WebSocket = socket();
  Status = Connect(WebSocket, IPV4(192,168,10,1), 80); //修改为要测试的服务器 IP 地址
  Status = Send(WebSocket, RequestData, AsciiStrLen(RequestData)+2 ); //发送
  Status = Recv(WebSocket, RecvBuffer, 1024); //收取前 1KB 字节
  Status = Close(WebSocket);                  //关闭
  free(RecvBuffer);
  return Status;
}
```

14.3　本章小结

本章主要介绍了 UEFI 提供的网络协议栈，以及如何在 UEFI 中开发网络应用。UEFI 提供了完整的 TCP/IP 协议栈，利用这些协议我们可以开发任意的网络应用。EDK2 还提供了 FTP 服务器和 Web 服务器的示例代码 (AppPkg\\Applications\Sockets\[TftpServer|WebServer])。本章以请求一个网页为例介绍了 EFI_TCP4_PROTOCOL 在客户端的使用方法。

EFI_TCP4_PROTOCOL 的使用比较烦琐，对 C 语言程序员来说，Socket 接口更容易使用。幸运的是，StdLib 提供了 Socket 接口。使用 StdLib 将会让开发应用程序变得简单。下一章开始讲述如何使用 StdLib 开发应用程序。

Chapter 15 第 15 章

使用 C 标准库

C 标准库是 ANSI C 标准为 C 语言定义的标准库[⊖]。C 标准库包含 15 个头文件：assert.h、ctype.h、errno.h、float.h、limits.h、locale.h、math.h、setjmp.h、signal.h、stdarg.h、stddef.h、stdio.h、stdlib.h、string.h、time.h。C 标准库仅能被应用程序连接，使用 C 标准库的应用程序以 main 函数作为程序入口。

15.1 为什么使用 C 标准库函数

还记得刚学 C 语言时写的第一个程序吗？相信读者跟我一样，最初学习和写下的是这样一段代码：

```
#include <stdio.h>
int main () //或者 int main(int argc, char** argv)
{
  printf ("Hello World!\n");
  return 0;
}
```

当我们写了无数程序，回顾一下我们最熟悉的几个函数，标准库中的函数占了相当大的比例。标准库函数与 C 语言的紧密结合给我们开发程序带来了极大的便利。另外，使用 C 标准库开发的应用程序有很强的可移植性，标准库为上层应用程序屏蔽了底层平台的差异，使得这些程序可以在任意平台通过编译。但同时它也给了我们很大的约束，尤其是当我们将旧

⊖ The C Library Reference Guide，Eric Huss。

代码移植到 UEFI 平台的时候，不得不进行权衡：是继续使用 C 标准库函数，还是全部改写成 UEFI 标准里定义的函数。

当我们决定在工程中使用标准库函数时，有两种使用方法。如果我们只是使用有限的几个简单标准库函数，则可以使用宏或 inline 函数实现这些函数。如果简单的封装不能实现这些函数，就不得不使用 EDK2 中 StdLib。StdLib 为 UEFI 开发者提供了 C 标准库。

15.2　实现简单的 Std 函数

对某些标准库中的函数，EDK2 有对应的函数，例如标准库里的 memset 函数，在 UEFI 中的 BootServices 里面有 SetMem 与之对应，只是参数顺序稍有区别，这两个函数的对比如代码清单 15-1 所示。

代码清单 15-1　C 标准库的 memset 函数与 BS 的 SetMem 函数对比

```
// C 标准库中的 memset 函数
void * memset ( void * ptr, int value, size_t num );
// 启动服务中的 SetMem 函数
VOID EFIAPI SetMem(IN VOID* Buffer, IN UINTN Size, IN UINT8 Value);
```

再如常用的函数 malloc，在 EDK2 中，MemoryAllocationLib 提供了 AllocatePool 函数与之对应，它们的对比如代码清单 15-2 所示。

代码清单 15-2　C 标准库的 malloc 函数与 EDK2 中的 AllocatePool 函数对比

```
// C 标准库中的 malloc 函数
void* malloc (size_t size);
// EDK2 中 MemoryAllocationLib 库的 AllocatePool 函数
VOID *EFIAPI AllocatePool (IN UINTN AllocationSize);
```

对于这些简单的标准库函数，可以使用宏来做替换或使用内联函数做简单的转换，如代码清单 15-3 所示。

代码清单 15-3　用宏定义 C 标准库函数

```
#ifndef __SSTD__
#define __SSTD_
#ifdef __cplusplus
extern "C"{
#endif
#include <Library\MemoryAllocationLib.h>
#include <Library\BaseLib.h>
#define malloc(size) AllocatePool((UINTN) size)
#define free(ptr) FreePool(ptr)
#define memcpy(dest,source,count) CopyMem(dest,source,(UINTN)(count))
```

```
#define memset(dest,ch,count) gBS->SetMem(dest,(UINTN)(count),(UINT8)(ch))
#define memchr(buf,ch,count) ScanMem8(buf,(UINTN)(count),(UINT8)ch)
#define memcmp(buf1,buf2,count) (int)(CompareMem(buf1,buf2,(UINTN)(count)))
#define memmove(dest,source,count) CopyMem(dest,source,(UINTN)(count))
#define strcmp AsciiStrCmp
#define strncmp(str1,str2,n) (int)(AsciiStrnCmp(str1,str2,(UINTN)(n)))
#define strcpy(strDest,strSource) AsciiStrCpy(strDest,strSource)
#define strncpy(Dest,Source,count) AsciiStrnCpy(Dest,Source,(UINTN)count)
#define strlen(str) (size_t)(AsciiStrLen(str))
#define strcat(strDest,strSource) AsciiStrCat(strDest,strSource)
#define strchr(str,ch) ScanMem8((VOID *)(str),AsciiStrSize(str),(UINT8)ch)
#define strstr(text, pattern) AsciiStrStr(text, pattern)
#define wcscmp(s1, s2) StrCmp( (CONST CHAR16 *)s1, (CONST CHAR16 *)s2)
#define wcsstr(text, pattern) StrStr(text, pattern)
#define abort()  ASSERT (FALSE)
#define assert(expression) ASSERT(expression)
#define localtime(timer)  NULL
#define gmtime_r(timer,result) (result = NULL)
#ifdef __cplusplus
}
#endif
#endif
```

15.2.1 简单标准库函数包 sstdPkg

我们可以创建一个简单标准库函数包 sstdPkg，方便以后调用这些简单的标准库函数。图 15-1 展示了 sstdPkg 的目录结构。

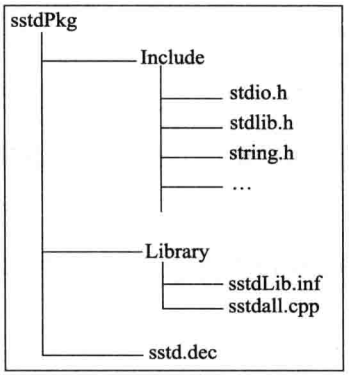


图 15-1 sstdPkg 目录结构

（1）Include 文件夹

Include 文件夹中放置 Std 头文件。sstd.h 文件包含了我们用于封装 Std 函数的宏或内联函数。其他所有头文件都只需 #include <sstd.h>。代码清单 15-4 列出了 stdio.h 文件。曾经，

EDK2 的 SecurityPkg 就采用这种方法使用标准库函数。

代码清单 15-4　简单标准库函数包中的 stdio.h 文件

```
#ifndef _STDIO_H_
#define _STDIO_H
#include <sstd.h>
#endif
```

（2）Library 文件夹

Library 文件夹中放置 sstdLib，sstdLib 包含了无法内联的 Std 函数，如 atexit、_allmul 等，这些函数放在 sstdall.cpp 文件中。Library 文件夹还包含了用于编译 sstdLib 的工程文件 sstdLib.inf。

代码清单 15-5 列出了工程文件 sstdLib.inf，相关格式已经在前面的章节中讲述过，相信读者已经了解，这里仅列出其内容，如代码清单 15-5 所示。

代码清单 15-5　sstdLib.inf 工程文件

```
[Defines]
  INF_VERSION = 0x00010005
  BASE_NAME = libsstd
  FILE_GUID = 348C4D62-BFBD-4882-9EDE-C80BB1C64336
  VERSION_STRING = 1.0
  MODULE_TYPE = BASE
  LIBRARY_CLASS = libsstd
[Sources]
  sstdall.cpp
[Packages]
  MdePkg/MdePkg.dec
  MdeModulePkg/MdeModulePkg.dec
[LibraryClasses]
  BaseLib
[BuildOptions]
  MSFT:*_*_*_CC_FLAGS = /wd4804
```

atexit 函数在第 10 章中已经讲述过，在此不再赘述，下面仅列出 sstdall.cpp 中的其他函数，如代码清单 15-6 所示。

代码清单 15-6　sstdall.cpp 文件

```
//这里是atexit函数
...
//atexit end

extern  "C"  int __cdecl _purecall ( void )
{
  return  0 ;
}
```

```
extern  "C"  INT64 _allmul(INT64 sm1, INT64 sm2)
{
  return (INT64) MultS64x64(sm1, sm2);
}
extern  "C"  INT64 _alldiv(INT64 sm1, INT64 sm2)
{
  return   DivS64x64Remainder(sm1, sm2, NULL);
}
extern  "C"  UINT64 _aulldiv(UINT64 sm1, UINT64 sm2)
{
  return   DivU64x64Remainder(sm1, sm2, NULL);
}
```

sstdPkg 还需要一个 sstd.dec 文件，将 sstd 提供给其他模块使用，如代码清单 15-7 所示。

代码清单 15-7　sstd.dec 文件

```
[Defines]
  DEC_SPECIFICATION = 0x00010005
  PACKAGE_NAME = EdksstdPkg
  PACKAGE_GUID = e2805c44-8985-11db-9e98-0040d0c0d0cd
  PACKAGE_VERSION = 1.0
[Includes]
  Include
```

15.2.2　使用 sstdPkg

下面我们就可以使用 sstdPkg 提供的函数了。示例 15-1 展示了如何使用 malloc 函数。

【示例 15-1】 使用 sstdPkg 中的函数。

```
VOID Test()
{
  CHAR16* pStr;
  pStr = malloc(1024);
  free(pStr);
}
EFI_STATUS UefiMain(IN EFI_HANDLEImageHandle, IN EFI_SYSTEM_TABLE*SystemTable)
{
  EFI_STATUS Status = 0;
  Test();
  return Status;
}
```

在测试程序的工程文件 Testsstd.inf 中，需要在 [Packages] 块引用 sstdPkg，在 [LibraryClasses] 中引用 sstdLib，如示例 15-2 所示。

【示例 15-2】 测试程序的工程文件 Testsstd.inf。

```
[Defines]
  INF_VERSION = 0x00010005
```

```
  BASE_NAME = Testsstd
  FILE_GUID = 6987936E-ED34-54db-AE97-1FA5E4ED2127
  MODULE_TYPE = UEFI_APPLICATION
  VERSION_STRING = 1.0
  ENTRY_POINT = UefiMain
[Sources]
  Testsstd.c
[Packages]
  MdePkg/MdePkg.dec
  MdeModulePkg/MdeModulePkg.dec
  uefi/book/sstdPkg/sstd.dec
[LibraryClasses]
  UefiApplicationEntryPoint
  UefiBootServicesTableLib
  BaseMemoryLib
  BaseLib
  MemoryAllocationLib
  sstdLib
```

然后将下面一行加入到 .dsc 文件的 [LibraryClasses] 中，就可以用 build 命令编译 Testsstd.inf 了。

```
sstdLib|uefi/book/sstd/Library/sstdLib.inf
```

例如，通过如下命令编译 32 位的 Testsstd 工程：

```
build -a IA32 -p xx\xx.dsc -m uef\book\Testsstd\Testsstd.inf
```

15.3 使用 EDK2 的 StdLib

有时可能依靠简单的封装不能满足需求，例如要使用 fopen 等文件相关的标准库函数，这时就要使用 StdLib。并且 StdLib 不仅包含 C 标准库，还提供了 GNU C 库，例如 Socket 相关函数，使用 StdLib 会节省很多开发时间。

标准的使用 StdLib 的方式是使用 main 函数工程。相对的，非标准的使用 StdLib 的方式是在非 main 函数工程中使用 StdLib。前面我们讲过，使用标准库函数之前必须初始化标准库。这两种使用方式的区别在于如何初始化 StdLib 库。使用 main 函数时，StdLib 已经自动初始化；而在非 main 函数工程中，需要手动初始化 StdLib。下面我们首先讲述如何在 main 函数工程中使用 StdLib，然后讲述应用程序的执行过程，最后讲述如何在非 main 函数工程中手动初始化 StdLib。

15.3.1 main 函数工程

在第 3 章中已经讲述了如何使用 main 函数，下面再来回顾一下 main 函数工程的相关文件。

1. Main 函数工程的源文件及工程文件

在源文件中，我们需要实现 int main(int argc,char**argv) 作为应用程序的入口函数，如示例 15-3 所示。

【示例 15-3】 main 函数工程的源文件。

```
#include <Uefi.h>
#include <stdio.h>
int main (int argc, char **argv  )
{
  printf("HelloWorld\n");
  return 0;
}
```

在工程文件（.inf）中需要设置 ENTRY_POINT 为 ShellCEntryLib，引用 StdLib.dec 包和 ShellPkg.dec 包，引用库 ShellCEntryLib 及 LibC，如示例 15-4 所示。

【示例 15-4】 main 函数工程的工程文件。

```
[Defines]
 INF_VERSION = 0x00010006
 BASE_NAME = Main
 FILE_GUID = 4ea97c46-7491-4dfd-b442-747010f3ce5f
 MODULE_TYPE = UEFI_APPLICATION
 VERSION_   = 0.1
 ENTRY_POINT = ShellCEntryLib
[Sources]
  Main.c
[Packages]
  MdePkg/MdePkg.dec
  ShellPkg/ShellPkg.dec
  StdLib/StdLib.dec
[LibraryClasses]
  LibC                          # 提供了 ShellAppMain 函数
  LibStdio                      # 提供了 printf 函数
  ShellCEntryLib                # 提供了 ShellCEntryLib 函数
  UefiLib
```

要想编译 main.inf 工程，还需要将 StdLib/StdLib.inc 添加到 .dsc 文件中。在 .dsc 文件的末尾添加代码清单 15-8 所示的代码，StdLib.inc 文件中定义了 StdLib 包中的库。

代码清单 15-8　在 .dsc 文件末尾添加 StdLib 库的声明

```
!include StdLib/StdLib.inc
```

最后将 main.inf 添加到 .dsc 文件的 [Components] 中，这样就可以用 build 命令编译了。

2. C 标准库的初始化

下面我们介绍一下 StdLib 是如何初始化的。main 函数程序模块的入口函数是 ShellCEntryLib，

该函数调用了 LibC 提供的 ShellAppMain，StdLib 的初始化是在 ShellAppMain 中完成。代码清单 15-9 列出了 ShellAppMain 函数的源码。该函数定义在 stdlib\libc\main\Main.c 文件中。ShellAppMain 函数分为三个部分：

1）初始化。主要是初始化全局变量 gMD，gMD 初始化后，StdLib 中的函数才能被调用。保存执行上下文，以便 main 函数执行中遇到错误时返回。

2）调用 main 函数，用户可以调用 exit 或 _Exit 从 main 中直接跳转到程序退出点。

3）清理资源，退出程序。

代码清单 15-9　ShellAppMain 函数源码

```
INTN EFIAPI ShellAppMain (IN UINTN Argc,  IN CHAR16 **Argv)
{
  struct __filedes *mfd;
  char **nArgv;
  INTN ExitVal;
  int i;
  ExitVal = (INTN)RETURN_SUCCESS;
  //第 1 步：初始化
  //分配内存给 gMD。gMD 是 StdLib 中最重要的数据之一，StdLib 初始化过程就是对 gMD 的初始化
  gMD = AllocateZeroPool(sizeof(struct __MainData));
  if( gMD == NULL ) {
    ExitVal = (INTN)RETURN_OUT_OF_RESOURCES;
  }
  else {
    /* 初始化全局变量 */
    __sse2_available = 0;
    _fltused = 1;
    errno = 0;
    EFIerrno = 0;
    gMD->ClocksPerSecond = 1;
    gMD->AppStartTime = (clock_t)((UINT32)time(NULL));
    //初始化文件描述符
    //gMD->fdarray[0] stdin
    //gMD->fdarray[1] stdout
    //gMD->fdarray[2] stderr
    mfd = gMD->fdarray;
    for(i = 0; i < (FOPEN_MAX); ++i) {
      mfd[i].MyFD = (UINT16)i;
    }
    i = open("stdin:", (O_RDONLY | O_TTY_INIT), 0444);
    if(i == 0) {
      i = open("stdout:", (O_WRONLY | O_TTY_INIT), 0222);
      if(i == 1) {
        i = open("stderr:", O_WRONLY, 0222);
      }
    }
```

```
    if(i != 2) {
      Print(L"ERROR Initializing Standard IO: %a.\n    %r\n",
            strerror(errno), EFIerrno);
    }

    /* 将 CHAR16* 命令行类型的参数字符串转换为 char* 类型的字符串 */
    nArgv = ArgvConvert(Argc, Argv);
    if(nArgv == NULL) {//严重错误，直接退出
      ExitVal = (INTN)RETURN_INVALID_PARAMETER;
    }else {
      //成功初始化，下面需调用用户的 main 函数
      //首先保存执行上下文
      if( setjmp(gMD->MainExit) == 0) {
        //第 2 步 ：调用 main 函数
        ExitVal = (INTN)main( (int)Argc, gMD->NArgV);
        exitCleanup(ExitVal);
      }
      //程序退出点
      ExitVal = (INTN)gMD->ExitValue;
    }
    //第 3 步 ： 程序结束前清理和释放资源
    if( ExitVal == EXIT_FAILURE) {
      ExitVal = RETURN_ABORTED;
    }

    /* 关闭打开的文件 */
    for(i = OPEN_MAX - 1; i >= 0; --i) {
      (void)close(i);
    }

    /* 释放 gMD 占用的内存 */
    if(gMD != NULL) {
      if(gMD->NCmdLine != NULL) {
        FreePool( gMD->NCmdLine );
      }
      FreePool( gMD );
    }
  }
  return ExitVal;
}
```

在 ShellAppMain 函数中，gMD 是 struct__MainData 类型的变量，包含了很多重要的变量，其结构体如代码清单 15-10 所示。

代码清单 15-10 _MainData 结构体

```
//@file StdLibPrivateInternalFiles\Include\MainData.h
struct  __MainData {
```

```
  struct __filedes  fdarray[OPEN_MAX];          //文件描述符
  ConInstance       *StdIo[3];                  //stdin、stdout、stderr
  __sighandler_t *sigarray[SIG_LAST];           //信号处理函数数组 SIG_LAST 是 16
  char *NArgV[ARGC_MAX];                        //命令行参数数组
  char *NCmdLine;                               //命令行字符串
  void (*cleanup)(void);                        //清理函数
  __xithandler_t *atexit_handler[ATEXIT_MAX];   //存放 atexit 注册的函数指针
  clock_t AppStartTime;
  clock_t ClocksPerSecond;
  int num_atexit;                               //通过 atexit 注册的函数数量
  CHAR16 UString[UNICODE_STRING_MAX];           //临时缓冲区，供 setenv/getenv 使用
  CHAR16 UString2[UNICODE_STRING_MAX];          //临时缓冲区，供 setenv/getenv 使用
  struct tm BDTime;
  EFI_TIME TimeBuffer;
  char ASgetenv[ASCII_STRING_MAX];
  char ASasctime[ASCTIME_BUFLEN];
  jmp_buf MainExit;                             //供 _Exit 函数跳转使用
  int ExitValue;                                //程序退出时的状态
  BOOLEAN aborting;                             //确保 cleanup 函数仅执行一次
};
```

ShellAppMain 函数在调用 main 函数前，通过 setjmp 保存寄存器上下文，以便在 main 函数中遇到错误时返回到此处。main 函数后面是程序退出点，执行到这里时，有以下两种可能：

1）从 main 函数中正常返回，状态码为 0。

2）在 main 函数中执行了 exit（status）或 _Exit（status）。

无论哪种情况，错误状态码都将保存在 gMD->ExitValue，最后返回给 Shell。

3. 程序的安全退出

在执行 main 函数及其子函数的过程中，如果遇到严重错误需要退出程序，可以调用 exit 函数或 _Exit 函数，这两个函数最终会通过 longjmp 安全跳转到程序退出点。关于 setjmp 和 longjmp，读者可以参阅第 13 章相关内容。exit 函数源码可参见代码清单 15-11。

代码清单 15-11　退出函数 exit

```
//@file StdLib\LibC\StdLib\Environs.c
void exit(int status)
{
  exitCleanup((INTN) status);     //执行清理函数 gMD->cleanup 及 gMD->atexit_handler
  _Exit(status);                  //退出到程序退出点
}
void_Exit(int status)
{
  gMD->ExitValue = status;        //保存错误状态码
  longjmp(gMD->MainExit, 0x55);   //长跳转到程序退出点
```

```
#ifdef __GNUC__
  __builtin__Exit(status);          // 避免 GCC 报错
#endif
}
```

4. main 函数应用程序的执行过程

在上文中我们讲述了 StdLib 的初始化过程，下面我们从宏观的角度看一下 main 函数应用程序的执行过程。

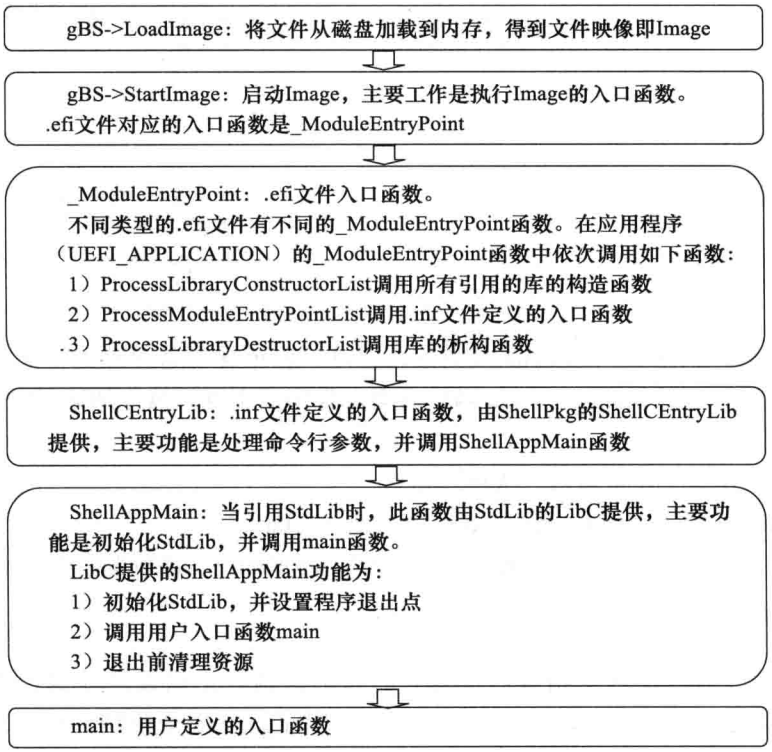


图 15-2　使用 main 函数的应用程序执行过程

15.3.2　非 main 函数工程

有时候我们需要使用 StdLib 中的个别函数，但又不希望完整地连接 StdLib 库，或者我们希望在某些特殊驱动中使用 StdLib 中的函数，例如前面提供的 FFDecoder 服务，当然这类情况比较少见，大部分情况下我们还是应该按照 UEFI 规范使用 StdLib。可以在不引用 LibC 的情况下使用 StdLib 里的函数，使用 StdLib 函数前必须初始化 StdLib，主要是初始化 gMD 数据结构，如同 LibC 里的 ShellAppMain 那样。示例 15-5 是用于手动初始化 StdLib 的函数 InitStdLib。

【示例 15-5】 手动初始化 StdLib 的函数。

```
#include   <errno.h>
#include   <stdio.h>
#include   <stdlib.h>
#include   <string.h>
#include   <time.h>
#include   <MainData.h>
#include   <unistd.h>
INTN EFIAPI InitStdLib (IN UINTN Argc,  IN CHAR16 **Argv)
{
  struct __filedes  *mfd;
  INTN ExitVal;
  int i;
  ExitVal = (INTN)RETURN_SUCCESS;
  gMD = AllocateZeroPool(sizeof(struct __MainData));
  if( gMD == NULL ) {
    ExitVal = (INTN)RETURN_OUT_OF_RESOURCES;
  }
  else {
/* 初始化全局变量 */
extern   int __sse2_available;
    __sse2_available = 0;
    _fltused = 1;
    errno = 0;
    EFIerrno = 0;
    gMD->ClocksPerSecond  = 1;
    gMD->AppStartTime = (clock_t)((UINT32)time(NULL));
    //初始化文件描述符
    mfd = gMD->fdarray;
    for(i = 0; i < (FOPEN_MAX); ++i) {
      mfd[i].MyFD = (UINT16)i;
    }
    i = open("stdin:", O_RDONLY, 0444);
    if(i == 0) {
      i = open("stdout:", O_WRONLY, 0222);
      if(i == 1) {
        i = open("stderr:", O_WRONLY, 0222);
      }
    }
  }
  return ExitVal;
}
```

退出程序前，还要释放 gMD 使用的内存资源，如示例 15-6 所示。

【示例 15-6】 Destroy StdLib 函数。

```
void EFIAPI DestroyStdLib()
```

```
 if(gMD != NULL) {
   FreePool( gMD );
  }
}
```

在调用 StdLib 函数前调用 InitStdLib 函数，使用完毕后调用 DestroyStdLib 函数释放 StdLib 占用的资源，如下所示。

```
#include <Uefi.h>
#include <stdio.h>
EFI_STATUS UefiMain(IN EFI_HANDLEImageHandle, IN EFI_SYSTEM_TABLE*SystemTable)
{
  EFI_STATUS Status = 0;
  Status = InitStdLib();
  printf("Hello StdLib");
  DestroyStdLib();
  return Status;
}
```

在驱动中如何使用 StdLib 函数可以参考第 8 章相关内容。

15.4 本章小结

在本章中，我们介绍了使用 C 标准库函数的三种方式。

第一种方式是通过宏或内联函数封装 StdLib 函数。这种方式仅仅可以使用一些简单的 StdLib 函数，如 malloc、free 等。当我们需要使用一些复杂的函数时，如 socket、select 等，这种方法就无能为力了。

第二种方式是引用 StdLib Package。在这种方式下，用户可以使用 main 函数，可以调用 StdLib 中的所有函数。用户的 C 语言代码可以直接从其他平台移植过来。

第三种方式是指在有些情况下我们不能使用 main 函数，但又希望能使用 StdLib 函数，这时我们需要在使用 StdLib 前初始化 StdLib。

第 16 章 Chapter 16

Shell 及常用 Shell 命令

UEFI Shell 作为一个特殊的 UEFI_APPLICATION，在 UEFI 系统中有着特殊的地位。它提供了用户与 UEFI 系统之间的接口，为其他的 UEFI_APPLICATION 提供了 Shell 服务。说它特殊，是因为它功能强大，但通常没有“用武之地”，正常启动计算机系统时不会用到 UEFI Shell。只有当系统启动遇到错误，或者计算机用户主动进入 UEFI Shell 对计算机系统进行配置时，Shell 才有机会“展示身手”。

16.1 Shell 的编译与执行

ShellPkg 目录下包含了 Shell 的源代码，我们可以编译 ShellPkg 以获得 shell.efi。

通过以下命令编译 32 位的 shell.efi。

```
build -a IA32 -p ShellPkg\ShellPkg.dsc
```

通过以下命令编译 64 位的 shell.efi。

```
build -a X64 -p ShellPkg\ShellPkg.dsc
```

如果目标 UEFI 平台是 32 位系统，则将 32 位的 shell.efi 复制到 ESP 分区的 efi\boot 目录下并重命名为 BootIA32.efi，启动 UEFI 系统时就会执行 BootIA32.efi 从而进入 Shell。

如果目标 UEFI 平台是 64 位系统，则将 64 位的 shell.efi 复制到 ESP 分区的 efi\boot 目录下并重命名为 BootX64.efi，启动 UEFI 系统时就会执行 BootX64.efi 从而进入 Shell。

使用这种方式启动 Shell 时通常是不带参数的（或者说使用了系统参数），UEFI 进入 Shell 时，会将 Shell Protocol 安装到 Shell 的 ImageHandle 上。然后，UEFI 系统中的应用才可以使用 Shell Protocol 服务。进入 UEFI 后，Shell 首先检查 efi\Boot\ 目录下是否有 startup.nsh 脚本，如果有该脚本，则执行该脚本，然后进入 Shell 命令行等待用户输入；若 efi\Boot\ 目录下没有 startup.nsh 脚本，则直接进入命令行等待用户输入。在命令行中，可以执行 Shell 内部命令，也可以执行 Shell 外部命令。执行外部命令时，Shell 先用 Load Image Protocol 将可执行文件载入内存生成 Image，然后在该 Image 上安装 Shell Parameter Protocol，然后调用这个 Image 的入口函数从而执行该外部命令。

有时我们希望启动 Shell 时传入参数以控制 Shell 的行为，那么可以进入 Shell 后设置 Shell 的启动参数。

先来看一下 Shell 的启动参数。

```
shell.efi [ShellOpt-options] [options] [file-name [file-name-options]]
```

表 16-1 列出了 Shell 的所有启动参数。

表 16-1 Shell 启动参数表

启动参数	作　用
-nostartup	进入 Shell 时不执行脚本 startup.nsh
-noconsoleout	Shell 标准输出不显示
-noconsolein	Shell 无标准输入
-delay[:*n*]	指定等待 startup.nsh 启动的时间。默认时间是 5 秒。-delay:0 表示立即执行
-nointerrupt	不支持 <Ctrl+C> 的中止程序功能
-nomap	启动后不显示块设备的 map 信息
-noversion	启动后不显示 version
-startup	进入 Shell 时执行脚本 startup.nsh

启动选项也可以放在 UEFI 全局变量“ShellOpt”中，称为 ShellOpt-Options。启动 Shell 时，ShellOpt-Options 总是被最先处理。

当 ShellOpt-Options 和 options 中均不包含 -startup 时，可以使用 file-name [file-name-options] 指定 Shell 启动后要执行的文件。如果启动参数中既有 -startup 又有 file-name [file-name-options]，则 file-name [file-name-options] 被忽略。file-name 可以是绝对路径，如 fs0:\init.nsh，也可以是相对路径，如 init.nsh，Shell 将在环境变量 path 指定的路径中搜索指定文件。

这里还需要解释一下 -nomap 中的 map 的意义。在 Windows 系统中我们总是用英文字母加一个冒号（如 C:）表示一个分区。“C:”这种符号就是一种 map，通常它实际指向设备 \Device\HardDisk1\Partition0。UEFI 中的 map 与之相似，它把实际物理地址映射成一个短小

易记的名字供用户使用，通常一个块设备的 map 名字为 BLK*x*:，一个 FAT 分区的 map 名字为 FS*x*:，*x* 为一个阿拉伯数字，map 名字不区分大小写。通过 Shell 的命令可以查看系统的 map，用户可以用 map 命令自定义物理设备的 map 名字。通常，Shell 启动后会列出系统的 map，例如：

```
Mapping table
 FS0: Alias(s):F8:
          VenHw(58C518B1-76F3-11D4-BCEA-0080C73C8881)/VenHw(0C95A935-A006-1
1D4-BCFA-0080C73C8881,00000000)
BLK0: Alias(s):
          VenHw(58C518B1-76F3-11D4-BCEA-0080C73C8881)/VenHw(0C95A928-A006-1
1D4-BCFA-0080C73C8881,00000000)
```

图 16-1　进入 Shell 后打印的系统 map

Shell 进入命令行后，支持“Ctrl+C”命令用于终止脚本任务。与 Windows 命令行和 Linux 命令行不同的是，UEFI Shell 的“Ctrl+C”命令仅对脚本有效，会执行完脚本中正在执行的命令，在执行下一条命令前退出脚本。

上文讲过，UEFI 应用分为三种：UefiMain 类型的应用、ShellAppMain 类型的应用、main 类型的应用。ShellAppMain 和 main 类型的应用都需要 Shell Protocol 和 Shell Parameters Protocol 的支持。从 Shell 的执行过程可以看出，只有在 Shell 环境下，ShellAppMain 和 main 类型才能成功执行。也只有在 Shell 环境下，应用程序才能使用 Shell Protocol 提供的服务。UefiMain 类型的应用不需要 Shell 的支持，因而可以在 Shell 环境下执行，也可以在没有 Shell 的环境下执行。

16.2　Shell 服务

ShellAppMain 和 main 类型可以使用 Shell 提供的服务。Shell 为程序开发者提供了两种使用 Shell 服务的方式：一种是使用 Shell Protocol 和 Shell Parameters Protocol；另一种是使用 UefiShellLib 提供的函数。在讲述如何使用 Shell 服务之前，我们先来详细了解一下 Shell 究竟提供了哪些服务。

1. Shell Protocol 提供的服务

在第 7 章中，我们曾讲述过 Shell Protocol 中文件相关的服务，这里再来熟悉一下除文件操作之外的服务。代码清单 16-1 列出了文件操作之外的服务。

代码清单 16-1　EFI_SHELL_PROTOCOL 结构体

```
typedef struct _EFI_SHELL_PROTOCOL {
  EFI_SHELL_EXECUTE Execute;      //执行 Shell 命令行命令
```

```
  EFI_SHELL_GET_ENV GetEnv;        //获得环境变量的值
  EFI_SHELL_SET_ENV SetEnv;        //设置环境变量的值
  EFI_SHELL_GET_ALIAS GetAlias;  //获得 Shell 命令的别名
  EFI_SHELL_SET_ALIAS SetAlias;  //设置 Shell 命令的别名
  EFI_SHELL_GET_HELP_TEXT GetHelpText;  //获得 Shell 命令的帮助信息
  EFI_SHELL_GET_DEVICE_PATH_FROM_MAP GetDevicePathFromMap; //从 Map 名获得 DevicePath
  EFI_SHELL_GET_MAP_FROM_DEVICE_PATH GetMapFromDevicePath; //从 DevicePath 获得 Map 名
  ...       //文件相关服务
  EFI_SHELL_SET_MAP SetMap         //设置设备的 Map 名
  ...       //文件相关服务
  EFI_SHELL_BATCH_IS_ACTIVE BatchIsActive;       //是否正在执行脚本
  //是否为 Root Shell。在 Shell 中可以启动一个子 Shell
  EFI_SHELL_IS_ROOT_SHELL IsRootShell;
  EFI_SHELL_ENABLE_PAGE_BREAK EnablePageBreak;  //启动分屏显示模式
  EFI_SHELL_DISABLE_PAGE_BREAK DisablePageBreak;        //禁止分屏显示模式
  EFI_SHELL_GET_PAGE_BREAK GetPageBreak;         //获取当前分屏显示模式状态
  EFI_SHELL_GET_DEVICE_NAME GetDeviceName;       //获取设备的名字
  ...        //文件相关服务
  EFI_SHELL_OPEN_ROOT OpenRoot;  //打开设备的根目录
  EFI_SHELL_OPEN_ROOT_BY_HANDLE OpenRootByHandle; //打开 Handle 指定的设备的根目录
  EFI_EVENT ExecutionBreak;        //用户按下“Ctrl+C”时触发该事件
  UINT32 MajorVersion;
  UINT32 MinorVersion;
} EFI_SHELL_PROTOCOL;
```

这里我们重点讲述一下 EFI_SHELL_PROTOCOL 的 Execute 服务，如代码清单 16-2 所示。该函数会启动一个子 Shell，并在子 Shell 中执行指定命令，命令的退出码将作为 Execute 函数的返回值。

如果 Environment 为 NULL，当前的环境变量会传递到子 Shell 中，从子 Shell 返回时，子 Shell 对环境变量的更改被保存。

如果 Environment 不为 NULL，子 Shell 使用指定的环境变量，从子 Shell 返回时，子 Shell 对环境变量的更改被丢弃。

子 Shell 从当前设备的当前路径上执行。

成功执行 Shell 命令完毕，Shell 命令的返回值通过 *StatusCode 返回。

代码清单 16-2　EFI_SHELL_PROTOCOL 的 Execute 服务的函数原型

```
//在子 Shell 中执行命令行命令
typedef EFI_STATUS(EFIAPI *EFI_SHELL_EXECUTE) (
  IN EFI_HANDLE *ParentImageHandle,      //执行 Execute 函数的 ImageHandle
  IN CHAR16 *CommandLine OPTIONAL,       //Shell 命令行命令
  IN CHAR16 **Environment OPTIONAL,      //环境变量
  OUT EFI_STATUS *StatusCode OPTIONAL    //Shell 命令返回值
  );
```

说明：

- 参数 ParentImageHandle 是执行 Execute 命令的 Image 的 ImageHandle，通常是 gImageHandle；
- 参数 CommandLine 是前面讲过的启动 Shell 时的命令行参数（不包含第 0 个参数 Shell.efi）；
- 参数 Environment 是字符串数组，数组以 NULL 为最后一项，其余每一项表示一个环境变量，格式为 Var=Value；
- 参数 StatusCode 是 Shell 命令的返回值，该返回值是通过 exit 命令设置的。

示例 16-1 展示了使用 Execute 服务的 3 个示例。

【示例 16-1】 Execute 服务的示例。

```
#include <Uefi.h>
#include <Library/UefiLib.h>
#include <Library/UefiBootServicesTableLib.h>
INTN ShellAppMain (IN UINTN Argc, IN CHAR16 **Argv)
{
  EFI_STATUS Status = 0;
  EFI_STATUS RStatus = 0;
  EFI_SHELL_PROTOCOL            *mEfiShellProtocol;
  CHAR16* Environment[] = { L"Var1=1", L"Var2=2", 0};  // 子 Shell 的环境变量
  // 首先获得 EFI_SHELL_PROTOCOL，也可以直接使用 gEfiShellProtocol
  Status = gBS->LocateProtocol( &gEfiShellProtocolGuid, NULL, &mEfiShellProtocol);
  // 示例 1：在子 Shell 中执行一个内部命令
  Status = mEfiShellProtocol -> Execute(&gImageHandle,
    L"-nomap ls fs0:",  NULL, &RStatus);
  Print(L"Execute return %r with Status %d\n", Status, RStatus);
  // 示例 2：在子 Shell 中设置返回值
  Status = mEfiShellProtocol -> Execute(&gImageHandle,
    L"-nomap -noversion -nostartup exit 1",  NULL, &RStatus);
  Print(L"Execute return %r with Status %d\n", Status, RStatus);
  // 示例 3：在子 Shell 中使用传入的环境变量
  Status = mEfiShellProtocol -> Execute(&gImageHandle,
    L"-nomap -noversion echo Var1 = %Var1%",  Environment, &RStatus);
  Print(L"Execute return %r with Status %d\n", Status, RStatus);
  return Status;
}
```

图 16-2 是执行示例 16-1 所示代码后的输出结果。

上文讲过，除了直接使用 Shell Protocol，还可以通过 UefiShellLib 提供的函数使用 Shell Protocol 提供的服务。例如，对 Shell Protocol 中的 Execute 函数，UefiShellLib 提供了对应的 ShellExecute 函数，ShellExecute 是对 Shell Protocol 中的 Execute 进行的简单封装。基本上对于所有 Shell Protocol 中的函数，在 UefiShellLib 中都有与之对应的函数，函数名命名规则是以“Shell”为前缀。代码清单 16-3 是 ShellExecute 函数原型。

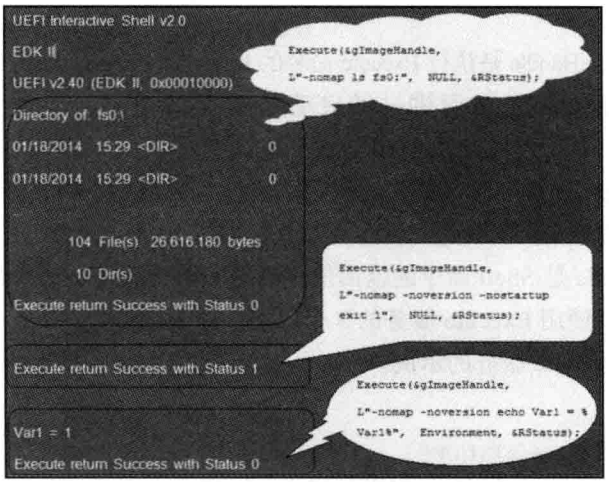


图 16-2 示例 16-1 的输出

代码清单 16-3 ShellExecute 函数原型

```
EFI_STATUS EFIAPI ShellExecute (
  IN EFI_HANDLE *ParentHandle,
  IN CHAR16 *CommandLine OPTIONAL,
  IN BOOLEAN Output OPTIONAL,
  IN CHAR16 **EnvironmentVariables OPTIONAL,
  OUT EFI_STATUS *Status OPTIONAL
  )
```

Shell Execute 函数的参数与 Shell Protocol 中的 Execute 函数的参数相似，只是多了参数 Output。Output 为 TRUE，表示显示调试信息。代码清单 16-4 是 Shell Execute 函数的实现。

代码清单 16-4 ShellExecute 函数实现

```
//@file ShellPkg/Library/UefiShellLib/UefiShellLib.c
{
  if (gEfiShellProtocol != NULL) {
   //如果存在 UEFI Shell 2.0，则使用 gEfiShellProtocol
   return (gEfiShellProtocol->Execute(ParentHandle,
     CommandLine, EnvironmentVariables, Status));
  }
  //不存在 gEfiShellProtocol。如果存在 ShellEnvironment2，则使用 ShellEnvironment2
  if (mEfiShellEnvironment2 != NULL) {
      return (mEfiShellEnvironment2->Execute(*ParentHandle, CommandLine,
Output));
  }
  return (EFI_UNSUPPORTED);
}
```

示例 16-2 通过 ShellExecute 函数重新实现示例 16-1 中用 mEfiShellProtocol->Execute 所实现的功能。在该源码中需加入头文件 <Library/ShellLib.h>。

【示例 16-2】 用 ShellExecute 函数替代 mEfiShellProtocol->Execute 服务。

```
#include <Uefi.h>
#include <Library/UefiLib.h>
#include <Library/UefiBootServicesTableLib.h>
#include <Library/ShellLib.h>
INTN EFIAPI ShellAppMain (IN UINTN Argc,  IN CHAR16 **Argv)
{
  EFI_STATUS Status = 0;
  EFI_STATUS RStatus = 0;
  CHAR16* Environment[] = { L"Var1=1", L"Var2=2", 0};
  //首先确保 gEfiShellProtocol 指向了 Shell Protocol 实例
  if(gEfiShellProtocol == NULL){
    Status = gBS -> LocateProtocol( &gEfiShellProtocolGuid, NULL,
  &gEfiShellProtocol);
  }
  Status=ShellExecute(&gImageHandle, L"-nomap ls fs0:",FALSE, NULL, &RStatus);
  Print(L"Execute return %r with Status %d\n", Status, RStatus);
  ...
}
```

UefiShellLib 中的其他函数与 ShellExecute 类似，我们不再一一讲述。

下面我们介绍一下 gEfiShellProtocol 及 UefiShellLib 的初始化。

2. UefiShellLib 的初始化

gEfiShellProtocol 是 EFI_SHELL_PROTOCOL* 类型的变量，由 UefiShellLib 提供。我们曾经讲过，Library 可以提供构造函数，构造函数在 ENTRY_POINT 之前执行。在 UefiShellLib 的工程文件中定义了 CONSTRUCTOR 和 DESTRUCTOR，如代码清单 16-5 所示。

代码清单 16-5　UefiShellLib 的工程文件

```
[Defines]
  INF_VERSION = 0x00010006
  BASE_NAME = UefiShellLib
  FILE_GUID = 449D0F00-2148-4a43-9836-F10B3980ECF5
  MODULE_TYPE = UEFI_DRIVER
  VERSION_STRING = 1.0
  LIBRARY_CLASS = ShellLib|UEFI_APPLICATION UEFI_DRIVER DXE_RUNTIME_DRIVER
  CONSTRUCTOR = ShellLibConstructor
  DESTRUCTOR = ShellLibDestructor
```

当我们引用 UefiShellLib 时，“Status=ShellLibConstructor(ImageHandle, SystemTable);”会被添加到 ProcessLibraryConstructorList 函数中，从而在调用 ShellAppMain 之前执行。代

码清单 16-6 列出了构造函数 ShellLibConstructor 的源码。

代码清单 16-6 UefiShellLib 的构造函数 ShellLibConstructor

```
EFI_STATUS EFIAPI ShellLibConstructor(IN EFI_HANDLE ImageHandle,
  IN EFI_SYSTEM_TABLE *SystemTable)
{
  //全局变量清零
  mEfiShellEnvironment2 = NULL;
  gEfiShellProtocol = NULL;
  gEfiShellParametersProtocol = NULL;
  mEfiShellInterface = NULL;
  mEfiShellEnvironment2Handle = NULL;
  ...
  //设置全局变量
  return (ShellLibConstructorWorker(ImageHandle, SystemTable));
}
EFI_STATUS EFIAPI ShellLibConstructorWorker(IN EFI_HANDLE ImageHandle,
  IN EFI_SYSTEM_TABLE *SystemTable)
{
  EFI_STATUS  Status;
  //打开 Shell Protocol
  Status = gBS->OpenProtocol( ImageHandle, &gEfiShellProtocolGuid,
    (VOID **)&gEfiShellProtocol, ImageHandle, NULL,
     EFI_OPEN_PROTOCOL_GET_PROTOCOL);
  ...
  //设置文件操作函数
  if ((mEfiShellEnvironment2 != NULL && mEfiShellInterface != NULL) ||
      (gEfiShellProtocol != NULL && gEfiShellParametersProtocol != NULL)    ) {
    if (gEfiShellProtocol != NULL) {
      FileFunctionMap.GetFileInfo = gEfiShellProtocol->GetFileInfo;
      ...
    } else {
      FileFunctionMap.GetFileInfo= (EFI_SHELL_GET_FILE_INFO)FileHandleGetInfo;
      ...
    }
    return (EFI_SUCCESS);
  }
}
```

FileFunctionMap 数据结构将被 UefiShellLib 中的函数使用。例如，ShellReadFile 函数就是通过调用 FileFunctionMap.ReadFile(…) 实现的，如代码清单 16-7 所示。

代码清单 16-7 UefiShellLib 中的 ReadFile 函数

```
EFI_STATUS EFIAPI ShellReadFile( IN SHELL_FILE_HANDLE FileHandle,
  IN OUT UINTN *BufferSize,  OUT VOID *Buffer)
{
  return (FileFunctionMap.ReadFile(FileHandle, BufferSize, Buffer));
}
```

正是因为 gEfiShellProtocol、FileFunctionMap 等全局变量被 UefiShellLib 的构造函数初始化，我们才可以在 ShellAppMain 中调用 UefiShellLib 中的函数。

16.3　Shell 脚本

在 UEFI Shell 中可以执行 Shell 脚本。Shell 脚本是以 .nsh 为扩展名的文件。在脚本中可以执行 Shell 命令和外部命令，也可以使用内置的流程控制命令 for、endfor、goto、if、else、endif、exit。

16.3.1　Shell 脚本语法简介

1. 基本语法

脚本中的每一行表示一条语句，格式为：

```
命令 [参数 1][参数 2]   #参数为可选项
```

例如，下面的命令将递归列出 uefi 目录中的所有文件和子目录。

```
ls -r uefi
```

说明：

- 注释：脚本中每一行内符号 # 后面的内容表示注释。
- 命令行参数：执行脚本时可以带命令行参数，最多可以带 10 个参数。在脚本中用 %*n* 可以获得第 *n* 个参数。%0 是当前执行的这个脚本的文件名。
- 环境变量：在脚本中可以通过 %var% 获得环境变量 var 的值。例如，%lasterror% 可以得到 lasterror 的值。
- 脚本返回值：脚本结束时可以通过 Exit 命令设置返回值。

脚本示例如下所示：

```
exit [/b] [exit-code] # /b 和 exit-code 是可选项
```

其中，exit /b 表示退出当前脚本；exit 则退出 UEFI Shell。

使用 exit 退出脚本时，exit-code 将被赋值给环境变量 lasterror。退出一个 Shell 实例时，exit-code 将返回给调用者。执行完一个脚本后可以通过“echo %lasterror%”在屏幕输出脚本的退出码。

2. if 条件语句

if 条件语句格式如下：

```
if expression then
  语句
else
  语句
endif
```

在条件语句表达式中，有如下几种情况。

可以使用下列比较运算符：gt、lt、eq、ne、ge、le、==、ugt、ult、uge、ule。例如：

```
if %lasterror% == 0 then
endif
```

可以使用下列逻辑运算符：and、or、not。例如：

```
if %lasterror% ne 0 and %lasterror%  ne 1 then
else
endif
```

条件语句也可以使用内置的布尔函数：IsInt、Exists、Avaliable、Profile。这些函数返回 TRUE 或 FALSE。

```
if Exists fs0:\efi\boot\startup.nsh then # 存在文件
endif

if not Avaliable startup.nsh then # 在 %path% 和当前目录均找不到文件
endif
```

if 后面可以跟 /i、/s、/i/s 参数。/s 强制后面语句为字符串比较，/i 表示字符串比较时区分大小写。

```
if /s %uefishellversion% == “2.0” then
endif
```

条件语句中可以使用函数 UefiError、PiEroor、OemError，这些函数用于将整数转化为相应的错误码。

```
if %lasterror% == UefiError(8) then
endif
```

条件语句中的比较运算符见表 16-2。

表 16-2 比较运算符

运算符	含　义	运算符	含　义	运算符	含　义	运算符	含　义
gt	大于	ugt	无符号值大于	ge	大于或等于	uge	无符号值大于或等于
lt	小于	ult	无符号值小于	le	小于或等于	ule	无符号值小于或等于
ne	等于	eq	相等	==	相等		

布尔函数 IsInt、Exists、Avaliable、Profile 均接收一个字符串作为参数，返回值为 TRUE 或 FALSE。表 16-3 列出了这 4 个布尔函数的作用。

表 16-3　布尔函数说明

布尔函数	当返回值为 TRUE 时	当返回值为 FALSE 时
isInt（para）	para 是数字	para 不是数字
Exists file	file 存在	file 不存在
Avaliable file	file 存在于 path 指定的目录或当前目录	当前目录和 path 指定的目录均不存在 file
Profile para	para 匹配 profile 中某项	para 不匹配 profile 中任一项

3. for 循环语句

for 循环语句有以下两种格式。

第一种：

```
for %indexvar in set [;]
endfor
```

第二种：

```
for %indexvar run (start end [step])
endfor
```

其中，%indexvar 中的 indexvar 是 'a' ~ 'z' 或 'A' ~ 'Z' 中的一个字符。在 for 循环体中，可以使用 %indexvar。

在第一种格式中，set 中的元素是以空格分隔的字符串或文件名。如果字符串长度小于 255 并且含有通配符，那么这个字符串将被当成文件名，并且在 for 循环执行前展开。

例如：

```
for %a in *.efi readme
    echo %a
endfor
```

假设当前目录为 FS0:，目录中存在 readme、set.efi、set.c、test.efi、test.c 几个文件，执行上述 Shell 脚本后将输出：

```
FS0:\set.efi
FS0:\test.efi
FS0:\readme
```

在第二种格式中，%indexvar 将从 start 变化到 end，每次递增 step。step 为 1 或 –1 时可以省略。

goto 用于跳转到标号位置处。标号的声明是“:”在前，标号名在后。例如：

```
if  not Exist %1 then
    goto FileNotFound
endif
```

```
...
:FileNotFound
exit /b 1
```

16.3.2 自动运行指定应用程序

通过前面的学习，可以看出，有两种方式可以让计算机系统启动（执行 OS Loader 前）时执行我们的程序。

- 64 位系统：将我们的可执行文件（.efi 文件）重命名为 bootX64.efi，复制到 ESP 分区的 efi\boot 目录下。
- 32 位系统：将我们的可执行文件（.efi 文件）重命名为 bootIa32.efi，复制到 ESP 分区的 efi\boot 目录下。这种方式比较适合 OS Loader 类型的应用程序。

UEFI 的 bootX64.efi(bootIa32.efi) 是 Shell，这样我们可以将 startup.sh 放在 %path% 能找到的目录下，并在 startup.nsh 脚本中调用我们的应用程序。

16.4 Shell 内置命令

Shell 提供了丰富的内部命令。按照类别，可分为调试（Debug1）、驱动（Driver1）、网络（Network1）、安装（Install1）、Level1、Level2 和 Level3。

用 help 命令可以查看各种命令的帮助信息，格式为 help cmd 表示查看命令 cmd 的帮助信息。例如，查看 ifconfig 的帮助信息：

```
Shell:\>help ifconfig
```

若不特殊说明，则 Shell 内置命令的命令行参数中的数值使用十六进制格式。Shell 命令不区分大小写。表 16-4 列出了 Shell 命令通用的一些选项。

表 16-4 Shell 命令通用选项

命令行参数	意　义	命令行参数	意　义
-b、-break	输出信息分屏显示	-t、-terse	用简洁格式输出信息
-q、-quiet	不输出任何信息	-v、-verbose	输出辅助信息
-sfo	用标准格式输出	-?	输出帮助信息

16.4.1 调试设备的相关命令

调试设备的相关命令主要用于查询、读取或写设备。

dmem 命令用于查看内存或设备内存。address、size 都是十六进制数，-MMIO 指定地址为设备内存。

```
dmem [-b] [address] [size] [-MMIO]
```

不带参数的 dmem 将会输出 EFI 系统表，例如：

```
Shell:\> dmem
System: Table Structure size 00000048 revision 00020028
ConIn (000000000A937204) ConOut (000000000547B210) StdErr(000000000A9373B4)
Runtime Services 000000000656BF10
Boot Services 0000000000D65C60
SAL System Table 0000000000000000
ACPI Table 0000000000000000
ACPI 2.0 Table 0000000000000000
MPS Table 0000000000000000
SMBIOS Table 0000000006567000
```

mm 命令用于查看或修改 MEM（系统内存）、MMIO（设备内存）、IO（寄存器）、PCI（PCI 配置空间）和 PCIE（PCIE 配置空间）。示例如下所示：

```
mm address [value] [-w 1|2|4|8] [-MEM| -MMIO | -IO | -PCI | -PCIE] [-n]
```

在上面的代码中，address 为地址；value 为要写入的值；-w 后跟访问宽度；-n 表示非交互模式，如果不指定 -n，则该命令会进入交互模式。

进入交互模式后，按回车键显示下一条；数值加回车，写入数值后显示下一条；Q，退出命令。

例如，用 dmem 命令查看到启动服务的地址为 D65C60，我们可以用 mm 命令操作地址 D65C60，以下为两个示例。

1）非交互模式显示地址 D65C60 处的 8 字节值：

```
Shell:\>mm D65C60 -w 8
MEM 0x0000000000D65C60 : 0x56524553544F4F42
```

2）非交互模式显示地址 D65C60 处的 8 字节值，最后输入 q 并按回车键，退出交互模式：

```
Shell:\>mm D65C60-w 8 -n
MEM 0x0000000000D65C60 : 0x56524553544F4F42 >
MEM 0x0000000000D65C68 : 0x000000C800020028  > q
```

pci 命令显示 PCI 设备列表或显示 PCI 配置空间，例如：

```
pci [Bus Dev [Func] [-s Seg] [-i]]
```

不带参数的 pci 命令用于列出所有 PCI 设备。若 pic 命令带参数时，总线号（Bus）\ 设备号（Dev）\ 功能号（Func）用于指定 PCI 设备。-s Seg 用于指定 Segment。Func 和 Seg 默认值为 0。

其他命令如 edit、Comp、Compress 等较为简单，用到时可查阅相关帮助信息。

16.4.2 驱动相关命令

驱动类命令主要用于加载、卸载、查询驱动及驱动设备控制器。

1）dh 命令用于列出系统中的所有设备的信息，或某个设备的相关信息，格式如下所示：

```
dh[-l<lang>] [handle | -p <prot_id>] [-d] [-v]
```

说明：

- -l <lang> 表示用指定的语言显示。
- handle 是指 UEFI Handle 在系统中的编号。若 Shell 命令中不特别说明，则 handle 作为参数时都是指 UEFI handle 在系统中的编号。若没有指明 handle，dh 命令将列出所有设备的信息。
- -p <prot_id> 列出所有安装了 protocol prot_id 的设备的信息。
- -d 用于列出驱动相关信息。
- -v 用于输出 verbose 信息。

下面的示例是 dh 命令不带参数时的情况。

```
Shell:\>dh

01: UnknownDevice LoadedImage
02: Decompress
...
78: Shell ShellParameters SimpleTextOut UnknownDevice ImageDevicePath LoadedImage
```

不带参数的 dh 命令将列出所有设备的信息。第一列为 handle（UEFI handle 编号），冒号后为该设备上安装的 Protocol。

2）device 命令用于显示所有被驱动管理的设备。下面是该命令显示的设备情况：

```
  C  T   D
  T  Y C I
  R  P F A
  L  E G G #P #D #C Device Name
  == = = = == == == ============================================================
 1E R - -  0  1  11 VenHw(58C518B1-76F3-11D4-BCEA-0080C73C8881)
 46 D - -  2  0  0 Primary Console Input Device
 47 D - -  2  0  0 Primary Console Output Device
```

说明：

- 第一列 CTRL 是设备 handle 的编号。
- 第二列 TYPE 是设备类型，R 表示 root，B 表示 Bus，D 表示 Device。
- 第三列 CFG 表示该设备是否有 DriverConfigurationProtocol，X 表示有，- 表示没有。
- 第四列 DIAG 表示该设备是否有 DriverDiagnostics2Protocol，X 表示有，- 表示没有。

- 第五列 #P 是父句柄的数量。
- 第六列 #D 是该设备控制的设备的数量。
- 第七列 #C 是子设备的数量。
- 第八列是设备名称。

3）drivers 命令用于列出系统中的 driver。下面是该命令的输出示例：

```
            T   D
D           Y C I
R           P F A
V  VERSION E G G #D #C DRIVER NAME                              IMAGE NAME
=== ======= = = = == == ================================== ==========
3F 0000000A D N N 2  0   Platform Console Management Driver Fv(6D99E806-3D38-
42C2-A095-5F4300BFD7DC)/FvFile(51CCF399-4FDF-4E55-A45B-E123F84D456A)
40 0000000A D N N 2  0   Platform Console Management Driver
60 0000000A ? N N 0  0           Tcp Network Service Driver Fv(6D99E806-3D38-42
C2-A095-5F4300BFD7DC)/FvFile(6D6963AB-906D-4A65-A7CA-BD40E5D6AF4D)
```

说明：

- 第一列 DRV 是驱动 handle 的编号。
- 第二列 VERSION 是驱动的版本号。
- 第三列 TYPE 是驱动类型，A/B 表示总线驱动，D 表示设备驱动。
- 第四列 CFG 表示该驱动是否支持 DriverConfigurationProtocol，X 表示支持，- 表示不支持。
- 第五列 DIAG 表示该驱动是否支持 DriverDiagnostics2Protocol，X 表示支持，- 表示不支持。
- 第六列 #D 是该驱动控制的设备的数量。
- 第七列 #C 是子设备的数量。
- 第八列是驱动名称。
- 第九列是驱动来源。

对于网络驱动来说，#C 为 0 表示驱动没有加载成功。

4）connect 用于加载驱动到设备上并启动加载的驱动。下面是该命令的格式：

```
connect [[DeviceHandle] [DriverHandle] | [-c] | [-r]]
```

说明：

- -c 用于连接控制台设备。
- -r 用于递归扫描所有 handle，发现匹配的设备与驱动则加载驱动到该设备。加载过程中产生的新设备将加入到扫描队列中，直到没有任何可匹配的设备 / 驱动对。没有 -r 选项时，新产生的设备将不会被连接。

```
Shell:\>connect -r
```

若命令行中仅有一个 handle 参数，并且该 handle 上安装了 EFI_DRIVER_BINDING_PROTOCOL，它将被当做驱动 handle，该驱动将加载到匹配的设备上。假设 15 是驱动 handle，执行下面的命令后，15 将会加载到匹配的设备上。

```
Shell:\>connect 15
```

若命令行中仅有一个 handle 参数，并且该 handle 上没有安装 EFI_DRIVER_BINDING_PROTOCOL，它将被当做设备 handle，connect 命令将为该设备 handle 找出匹配的驱动，并将找到的驱动安装到该设备上。例如，17 是设备 handle，执行下面的命令后，17 将会加载到匹配的驱动上。

```
Shell:\>connect 17
```

若命令行中有两个 handle，而且驱动 DriverHandle 支持设备 DeviceHandle，则会将驱动 DriverHandle 加载到设备 DeviceHandle 上，并启动该设备。

5）disconnect 用于将驱动从设备上卸载下来。下面是该命令的格式：

```
disconnect DeviceHandle [DriverHandle [ChildHandle]] [-r]
```

其参数与 connect 参数意义相同。

6）load 命令用于加载驱动。下面是该命令的格式：

```
load [-nc] file [file2 …]
```

-nc 表示只加载驱动到内存，不进行 connect。不带 -nc 选项时，load 加载驱动后会调用 connect 将该驱动加载到匹配的设备上。

7）unload 用于将驱动从内存中清除。下面是该命令的格式：

```
unload [-n] [-v] Handle
```

-n 表示在执行 unload 过程中跳过所有提示信息，不需要用户确认。

16.4.3 网络相关命令

下面介绍一些网络相关的命令。

1）ifconfig 用于配置网络设备。下面是该命令的格式：

```
ifConfig [-?] [-c [Name]] [-l [Name]] [-s <Name> dhcp | <static <IP><Mask><Gateway>>
[permanent]]
```

说明：

- Name 是网络适配器的名字，如 eth0 等。

- -c 用于清除网络适配器的配置。
- -l 用于列出网络适配器的配置。
- -s 用于设置网络适配器 IP 地址。

若要列出所有网络适配器及其配置，则示例代码如下：

```
Shell:\>ifconfig -l
```

配置 eth0，使用 DHCP 获取 IP 地址：

```
Shell:\>ifconfig -s eth0 DHCP
```

配置 eth0，使用静态 IP 地址：

```
Shell:\>ifconfig -s eth0 static 192.168.1.2 255.255.255.0 192.168.1.1 permanent
```

2）ping 命令用于 ping 目标机器。下面是该命令的格式：

```
ping [-n count] [-l size] TargetIp
```

说明：

- -l size 用于发送 size 字节的数据。
- -n number 表示发送数据的次数。

文件及目录操作命令比较简单，与 Windows 命令行文件类操作命令格式相同。

前面分类介绍了 Shell 的内置命令，最后通过表 16-5 简单总结一下 Shell 中的这些命令。

表 16-5　Shell 内置命令表

Shell 命令	功　能	Shell 命令	功　能
alias	显示、创建、删除别名	dh	显示设备句柄
attrib	显示、更改文件或目录的属性	disconnect	从指定的设备卸载驱动
bcfg	管理启动项	dmem	显示系统或设备内存的内容
cd	更改当前工作目录	dmpstore	管理 UEFI NVRAM 变量
cls	清空标准输出；改变背景颜色	drivers	显示驱动列表
comp	比较两个文件	drvcfg	配置驱动
connect	将 driver 绑定到指定的设备并启动 driver	drvdiag	调用 Driver Diagnostics Protocol
cp	将文件或文件夹复制到另一个位置	echo	回显
date	显示或设置日期	edit	编辑 ASCII 或 UCS-2 文件
dblk	显示块设备里的块	eficompress	压缩文件
devices	列出所有设备	efidecompre	解压文件
devtree	显示设备树	exit	退出 Shell 或脚本

（续）

Shell 命令	功　能	Shell 命令	功　能
help	显示帮助	reconnect	重新连接驱动与设备
hexedit	二进制编辑器，可编辑文件、块设备或内存	reset	重启系统
ifconfig	配置 IP 地址	rm	删除文件或目录
load	加载 UEFI 驱动	sermode	设置串口属性
loadpcirom	加载 PCI ROM	set	显示或修改 Shell 中的环境变量
ls	列出目录内容或文件信息	setsize	调整文件大小
map	显示 Mapping	Setvar	设置 UEFI 变量
memmap	显示内存映射	smbiosview	显示 SMBIOS 信息
mkdir	创建目录	stall	在指定的时间内暂停执行
mm	列出或修改 MEM/MMIO/IO/PCI/PCIE 地址空间	time	显示 / 设置系统时间
mode	列出或修改输出设备的模式	timezone	显示 / 设置时区
mv	移动文件或目录	touch	更新文件时间戳
openinfo	显示 Protocols 打开信息	type	显示文件类型
pause	暂停执行脚本，等待用户输入	unload	卸载驱动
pci	显示 PCI 设备	vol	显示 / 设置卷标
ping	ping		

16.5 本章小结

Shell 是一种特殊的应用程序，它以交互的方式给用户提供了使用计算机系统的接口，主要目的是提供一种诊断计算机硬件错误的手段。

本章介绍了 Shell 脚本的语法及编写方法，Shell 的执行过程，以及如何利用 Shell 自动执行用户指定的程序或脚本。

在 Shell 中运行的应用程序可以使用 Shell 提供的 Shell 服务。本章介绍了 Shell 服务的具体内容及使用方法。

在本章中还介绍了几种常用的命令，例如用这些命令可以查看计算机软硬件资源的状态，以及配置计算机系统硬件资源。

附录 A Appendix A

UEFI 常用术语及简略语

1）ACPI：高级配置及电源管理接口（Advanced Configuration and Power Management Interface）。

2）BIOS：基本输入 / 输出系统（Basic Input/Output System）。

3）Boot Manager ：系统固件的一部分，负责实现系统启动策略。主要功能是发现可启动设备并根据系统配置（或同用户交互）确定启动设备，同时将系统控制权转交给该设备上的操作系统加载器或应用程序。

4）Boot Service Table：启动服务表。

5）Boot Service Time：从系统启动到调用 ExitBootServices（ ）之间的时间。

6）Boot Services：启动服务表中的服务，这些服务用于在启动服务时间管理整个计算机系统。

7）COFF：通用对象文件格式（Common Object File Format）。

8）Devices Handle：指向一个或一组 Protocol 的句柄，通过这些 Protocol 可以使用设备提供的服务。

9）Firmware：固件，存放在只读存储器（ROM）中的软件。

10）Font glyph：字体中某个字符的点阵位图。

11）GPT：GUID Partition Table。

12）HII：Human Interface Infrastructure。UEFI 中人机交互基础服务。

13）IFR：Internal Form Representation。在 EFI 中用于描述窗口。

14）Image ：符合 UEFI 规范的位于文件系统上的可执行文件，可以是 UEFI 应用程序、UEFI 驱动、操作系统加载器。也称为 EFI Image。

15）LBA：Logical Block Address。扇区逻辑地址，从 0 开始编号。

16）Legacy Platform：传统的基于 BIOS 的平台。

17）MBR：Master Boot Record。主引导记录，支持 BIOS 的传统硬盘的 0 号扇区。

18）NV：Nonvolatile，非易失性。通常指变量或代码将存放在非易失性存储上。

19）Runtime Services Table：运行时服务表。

20）Runtime Services：运行时服务表中的服务。操作系统运行期间仍然可以使用这些服务访问特定的硬件设备。

21）System Table：系统表，包含了输入 / 输出设备的句柄、启动服务表和运行时服务表指针，以及其他标准表。每个 Image 执行时都会获得一个系统表指针。

22）Watchdog Timer：一种定时器，在启动运行阶段如果系统出现严重错误并且无法回到正常执行流程中时，该定时器会触发，从而重新获得对系统的控制。

附录 B Appendix B

RFC 4646 常用语言列表

表 B-1 列出了 RFC 4646 常用语言。

表 B-1　RFC 4646 常用语言

代　码	语　言	代　码	语　言
zh-Hans	简体中文	zh-Hant	繁体中文
en	英语	fr-FR	法语
de	德语	ja	日语
ru	俄语	it	意大利语

Appendix C 附录 C

状 态 值

UEFI 系统中，通常一个服务或函数会返回一个 EFI_STATUS 类型的值。EFI_STATUS 是 UINTN 的一种类型定义。EFI_STATUS 最高位为 1 时，该值表示错误代码；最高位为 0 时，表示警告代码或状态代码；值为 0（即 EFI_SUCCESS）时表示成功。通常可用宏 EFI_ERROR 判断状态值是否为错误代码，例如：

```
if (EFI_ERROR(Status)) {
  //错误处理
}
```

这里需要明确一下 EFI_SUCCESS 与 EFI_ERROR 的区别，EFI_SUCCESS 被定义为 0，而 EFI_ERROR 是一个函数宏。它们的定义如下所示：

```
#define EFI_ERROR(A) EFI_ERROR(A)
#define EFI_ERROR(StatusCode) (((INTN)(EFI_STATUS)(StatusCode)) < 0)
#define EFI_SUCCESS  EFI_SUCCESS
#define EFI_SUCCESS 0
```

状态码分为 7 个区域，表 C-1 列出了除 0 之外的其他 6 个区域及其含义。

表 C-1　状态码各个区域的意义

32 位整型	64 位整型	各个区域状态码的意义
0x00000001~0x1 FFFFFFF	0x0000000000000001 ~ 0x1 FFFFFFFFFFFFFFF	UEFI 标准使用的警告码
0x20000000~0x3 FFFFFFF	0x2000000000000000 ~ 0x3 FFFFFFFFFFFFFFF	PI 标准使用的警告码
0x40000000~0x7 FFFFFFF	0x4000000000000000 ~ 0x7 FFFFFFFFFFFFFFF	OEM 使用的警告码
0x80000000~0x9 FFFFFFF	0x8000000000000000 ~ 0x9 FFFFFFFFFFFFFFF	UEFI 使用的错误码

（续）

32 位整型	64 位整型	各个区域状态码的意义
0xa0000000 ~ 0xbFFFFFFF	0xa000000000000000 ~ 0xb FFFFFFFFFFFFFFF	PI 标准使用的错误码
0xc0000000 ~ 0xFFFFFFFF	0xc000000000000000 ~ 0x FFFFFFFFFFFFFFFF	OEM 使用的错误码

函数宏 ENCODE_ERROR（_a）或 EFIERR（_a）用于根据值 _a 生成错误码。函数宏 ENCODE_WARNING（_a）用于根据 _a 生成警告码。表 C-2 列出了常用的错误码。

表 C-2　UEFI 常用错误码

助记符	错误码	含　义
EFI_LOAD_ERROR	ENCODE_ERROR（1）	加载 Image 失败
EFI_INVALID_PARAMETER	ENCODE_ERROR（2）	无效参数
EFI_UNSUPPORTED	ENCODE_ERROR（3）	不支持该操作
EFI_BAD_BUFFER_SIZE	ENCODE_ERROR（4）	缓冲区大小不符合要求
EFI_BUFFER_TOO_SMALL	ENCODE_ERROR（5）	给定的缓冲区太小。满足要求的大小值将由相应参数返回
EFI_NOT_READY	ENCODE_ERROR（6）	设备未就绪
EFI_DEVICE_ERROR	ENCODE_ERROR（7）	设备出现错误
EFI_WRITE_PROTECTED	ENCODE_ERROR（8）	设备写保护，因而不能被写
EFI_OUT_OF_RESOURCES	ENCODE_ERROR（9）	资源耗尽
EFI_VOLUME_CORRUPTED	ENCODE_ERROR（10）	文件系统错误
EFI_VOLUME_FULL	ENCODE_ERROR（11）	文件系统卷已满
EFI_NO_MEDIA	ENCODE_ERROR（12）	设备无介质
EFI_MEDIA_CHANGED	ENCODE_ERROR（13）	设备介质已更换
EFI_NOT_FOUND	ENCODE_ERROR（14）	请求的事项未找到
EFI_ACCESS_DENIED	ENCODE_ERROR（15）	访问拒绝
EFI_NO_RESPONSE	ENCODE_ERROR（16）	发出的请求没有回应
EFI_NO_MAPPING	ENCODE_ERROR（17）	设备映射不存在
EFI_TIMEOUT	ENCODE_ERROR（18）	超时
EFI_NOT_STARTED	ENCODE_ERROR（19）	Protocol 未启动，通常表示驱动未安装
EFI_ALREADY_STARTED	ENCODE_ERROR（20）	Protocol 已启动，通常表示驱动已经安装
EFI_ABORTED	ENCODE_ERROR（21）	操作中止
EFI_ICMP_ERROR	ENCODE_ERROR（22）	ICMP 错误
EFI_TFTP_ERROR	ENCODE_ERROR（23）	TFTP 错误
EFI_PROTOCOL_ERROR	ENCODE_ERROR（24）	网络操作中的 Protocol 错误
EFI_INCOMPATIBLE_VERSION	ENCODE_ERROR（25）	不兼容的版本号
EFI_SECURITY_VIOLATION	ENCODE_ERROR（26）	由于安全原因该操作未完成
EFI_CRC_ERROR	ENCODE_ERROR（27）	CRC 校验错误
EFI_END_OF_MEDIA	ENCODE_ERROR（28）	到达介质末尾或开始
EFI_END_OF_FILE	ENCODE_ERROR（31）	到达文件末尾
EFI_INVALID_LANGUAGE	ENCODE_ERROR（32）	无效的语言代码
EFI_COMPROMISED_DATA	ENCODE_ERROR（33）	数据的安全状态未知或安全状态已被破坏

Appendix D 附录 D

参考资料

（1）Unified Extensible Firmware Interface Specification

简介：它定义了 UEFI 为操作系统和操作系统加载器提供的服务和 Protocol，介绍了 GPT 硬盘的结构，描述了安全启动的原理及密钥管理，是 UEFI 开发者的必读文献。

（2）UEFI Platform Initialization Specification

简介：它描述了 UEFI 系统的初始化过程，可以认为它定义了 UEFI 的实现。分为 5 个部分：Pre-EFI Initialization Core Interface、Driver Execution Environment Core Interface、Shared Architectural Elements、System Management Mode Core Interface、Standards。

（3）EDK II Package Declaration（DEC）File Format Specification（August 2013，Revision 1.22）

简介：它定义了 .dec 文件的格式。

（4）EDK II Platform Description（DSC）File Specification（August 2013，Revision 1.22）

简介：它定义了 .dsc 文件的格式。

（5）EDK II Flash Description（FDF）File Specification（August 2013，Revision 1.22）

简介：它定义了 .fdf 文件的格式。

（6）EDK II Module Writer's Guide（March 2010，Revision 0.7）

简介：EDK2 模块开发指南。介绍了 EDK2 中各种模块的开发方法。

（7）EDK II Module Information（INF）File Specification（August 2013，Revision 1.22）

简介：它定义了 .inf 文件的格式。

（8）EDK II Platform Configuration Database Entries（An Introduction to PCD Entries，May 2011）

简介：介绍了 PCD 的概念及使用方法。

（9）UEFI Shell Specification（May 22，2012，Revision 2.0）

简介：它定义了 EFI_SHELL_PROTOCOL 以及 Shell 内置命令。

（10）VFR Programming Language（May 18，2012，Revision 1.7）

简介：VFR 是 Visual Forms Representation 的缩写，它用来编写文本界面下的窗口程序。

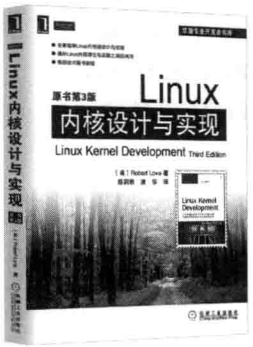
（11）Driver Writer's Guide for UEFI 2.3.1

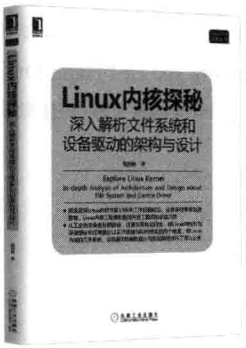
简介：UEFI 驱动开发指南。介绍了 UEFI 驱动模型，以及驱动开发中常用的 Protocol。它给出了总线（PCI、USB、SCSI、ATA）驱动及设备（显示设备、存储设备等）驱动的开发准则。

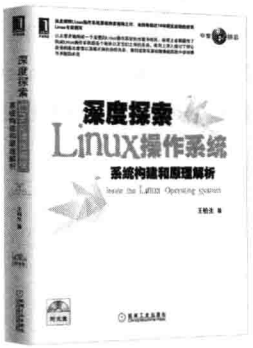
（12）Intel Platform Innovation Framework for EFI Hand-Off Block（HOB）Specification

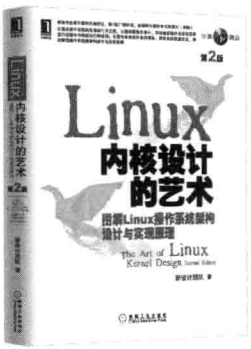
简介：它定义了 HOB 的类型及结构。HOB 用于从 PEI 阶段向 DXE 阶段传送数据。

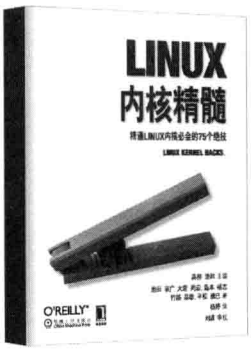
推荐阅读

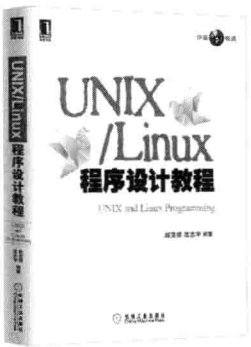












Linux内核设计与实现（原书第3版）

Linux内核开发人员Robert Love的力作，畅销多年的经典著作

深度探索Linux操作系统：系统构建和原理解析

百度核心系统部门资深专家力作，Linux操作系统领域的里程碑作品

Linux内核精髓：精通Linux内核必会的75个绝技

日本多位一线内核技术专家的经验和智慧结晶

Linux内核探秘：深入解析文件系统和设备驱动的架构与设计

腾讯顶级Linux系统专家和存储系统专家10年经验结晶

Linux内核设计的艺术：图解Linux操作系统架构设计与实现原理（第2版）

中国首部将版权输出到美国的计算机图书，中美两国取得骄人成绩

UNIX/Linux程序设计教程

UNIX/Linux权威著作，多所高校选定为教材

實戰

6年前，正华还是英特尔中国研究院并行编程团队的一员虎将。他话虽不多，但勤于思考，用心钻研，技术扎实。不觉已是六年，正华转战UEFI，取得了斐然成绩，并且笔耕不辍成就此书。我非常感佩于正华的严谨治学态度，本书几易其稿，反复修改了近一年，凝聚了他的学识和心血。过去十几年，UEFI的应用场景从PC扩展到手机、嵌入式甚至物联网设备，极大地缩短了系统开发时间，在安全性、兼容性、可扩展性和易用性上都各具亮点。矢志享受驾驭硬件之趣的同学们，请翻开此书，系统上电，在作者真诚的分享中，开始与硬件的亲密接触。

—— 吴甘沙

英特尔中国研究院院长

作为操作系统与硬件之间的连接桥梁，BIOS“统治”了计算机系统20余年，在整个计算机系统中扮演着非常重要的角色。近些年来，随着计算机硬件技术的快速发展，以及进入云计算和大数据时代以后，应用对计算机性能和计算能力的要求不断提升，BIOS因为设计上的先天缺陷，已经越来越不能满足需求，存在开发效率低、性能低、功能扩展性差、升级慢、安全性差等多方面的弊端。UEFI很好地解决了这些问题，逐渐成为BIOS的替代品。

本书是国内第一本关于UEFI的专著，对UEFI的使用、原理和开发做了细致而深入的讲解，对广大研究、使用和从事UEFI开发的读者来说，有着重要的意义，显得格外珍贵！

上架指导：计算机\程序设计
ISBN 978-7-111-48729-6
9 787111 487296 >
定价：89.00元

投稿热线：(010) 88379604
客服热线：(010) 88378991 88361066
购书热线：(010) 68326294 88379649 68995259
华章网站：www.hzbook.com
网上购书：www.china-pub.com
数字阅读：www.hzmedia.com.cn