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。

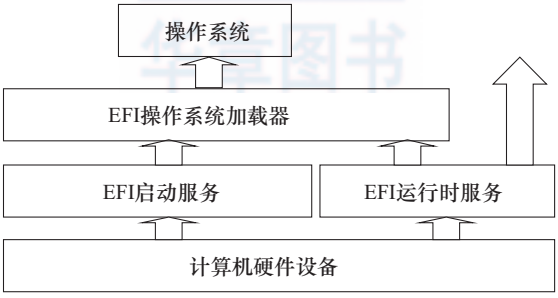

图 1-1　EFI 系统组成

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。

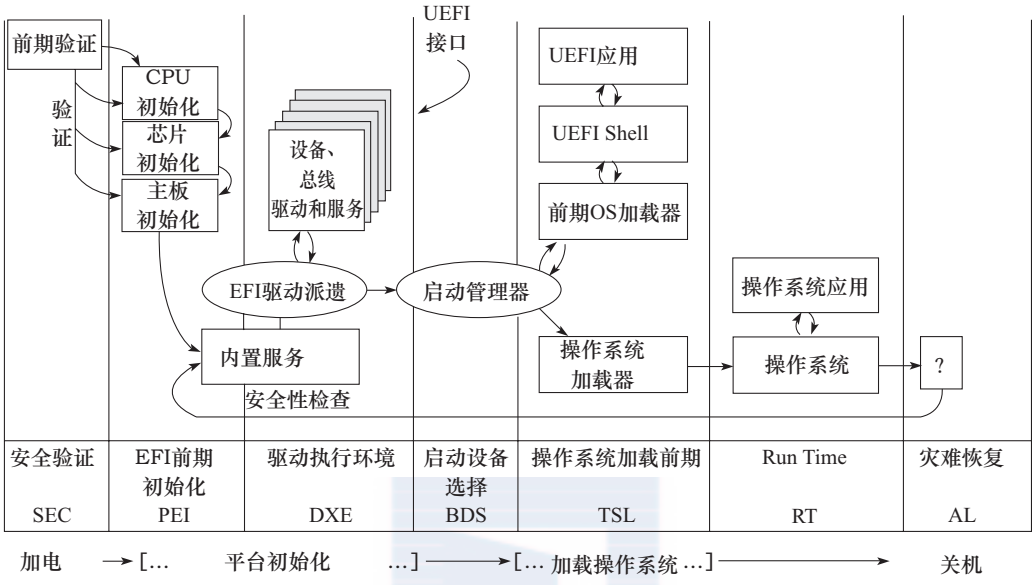

图 1-2 UEFI 系统的 7 个阶段

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。

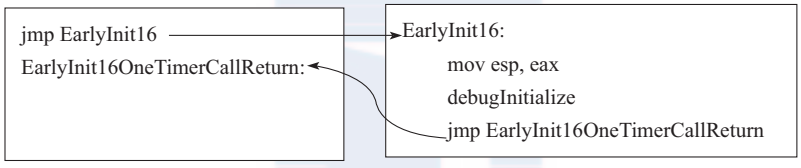

图 1-4　OneTimeCall 示例

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。

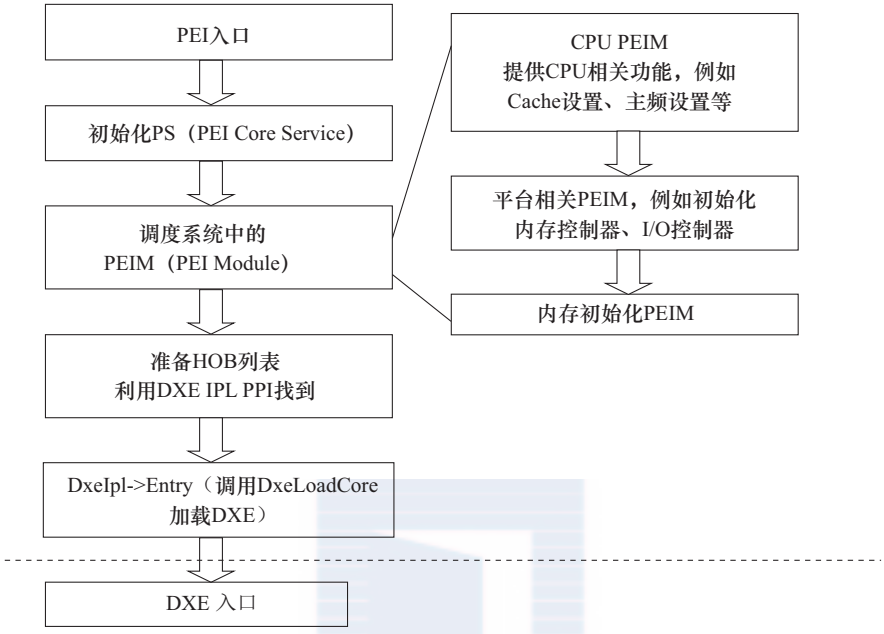

图 1-5 PEI 执行流程

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。

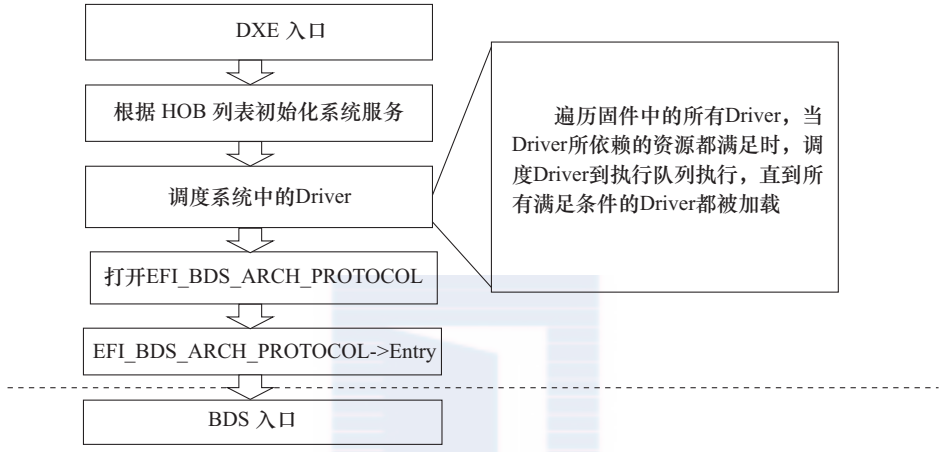

图 1-6　DXE 执行流程

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[㊀]。

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

㊀ 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。