

申请上海交通大学工程硕士学位论文

**基于 UEFI 系统的 LINUX 通用应用平台的设计与实现**

学校代码:	10248
作者姓名:	万象
学 号:	1090372232
第一导师:	戚正伟
第二导师:	张晓辉
学科专业:	软件工程
答辩日期:	2012 年 06 月 21 日

上海交通大学软件学院

2012 年 4 月

A Dissertation Submitted to Shanghai Jiao Tong University  
for Master Degree of Engineering

**DESIGN AND IMPLEMENTATION OF UEFI BASED  
GENERAL APPLICATION PLATFORM**

University Code:	10248
Author:	Wan Xiang
Student ID:	1090372232
Mentor 1:	Qi Zhengwei
Mentor 2:	Zhang Xiaohui
Field:	Software Engineering
Date of Oral Defense:	June 21, 2012


School of Software  
Shanghai Jiaotong University  
April, 2012

## 上海交通大学

### 学位论文原创性声明

本人郑重声明：所呈交的学位论文，是本人在导师的指导下，独立进行研究工作所取得的成果。除文中已经注明引用的内容外，本论文不包含任何其他个人或集体已经发表或撰写过的作品成果。对本文的研究做出重要贡献的个人和集体，均已在文中以明确方式标明。本人完全意识到本声明的法律结果由本人承担。

学位论文作者签名：



日期：2012年4月29日

## 上海交通大学

### 学位论文版权使用授权书

本学位论文作者完全了解学校有关保留、使用学位论文的规定，同意学校保留并向国家有关部门或机构送交论文的复印件和电子版，允许论文被查阅和借阅。本人授权上海交通大学可以将本学位论文的全部或部分内容编入有关数据库进行检索，可以采用影印、缩印或扫描等复制手段保存和汇编本学位论文。

保密 ☐，在\_\_\_\_\_年解密后适用本授权书。

本学位论文属于

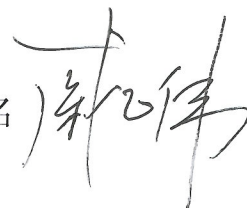
不保密 ☒。

(请在以上方框内打“√”)

学位论文作者签名：



指导教师签名



日期：2012年7月11日

日期：2012年7月11日

# 基于 UEFI 系统的 LINUX 通用应用平台的设计与实现

## 摘 要

统一可扩展固件接口(Unified Extensible Firmware Interface, UEFI)是新一代的硬件和操作系统之间的接口,它采用模块化的设计,C 语言风格的参数堆栈传递方式,并且以动态链接的形式构建系统,充分体现了软件工程的思想,具有统一的调用接口和良好的扩充性能。

它突破了传统的基本输入输出接口(Basic Input Output System, BIOS)的实模式下仅 16 位代码的寻址能力,能够运行于 32 位或 64 位保护模式。它采用 UEFI 驱动运行环境(Driver Execution Environment, DXE)加载 UEFI 驱动模块并解释运行的方式,识别并操作硬件,并具有向下兼容和跨平台支持的特性。较 BIOS 而言,大大减少了重复开发工作,降低了对开发人员专业性质的要求,并大大加快了系统研发的周期。

相比针对 Linux 系统启动的 Coreboot,它还不仅定义了清晰全面的接口,能够广泛的支持 Linux、Windows、Mac OS 等多种操作系统,同时还得到了底层硬件厂商如 Intel、AMD、HP、Dell 的支持,使得它更具有通用性。

得益于这些优点,UEFI 格外受到主板厂家青睐而得以迅速推广。UEFI 在近 10 年的推广后,已经成为主流,将使用传统的 BIOS 的或者其他接口的微型计算机系统逐渐淘汰出市场。

但是 UEFI 由于本身的设计上的原因,如在 SMP 系统中仍然只使用 BSP、使用单进程而不支持多进程、没有用户权限控制、仅使用时钟中断等等,导致其无法完全发挥硬件的全部能力。与此同时软件供应商支持力度也不够,使得其应用通常只有硬件相关厂商开发,而且应用的范围也相当局限,不同厂商所自定义的接口也有差异,也降低了其本身的价值。开发难度相比传统 BIOS 来讲虽有很大提高,但是相对 Linux 而言仍然太大。

本文为了进一步提高采用 UEFI 接口的系统的应用能力和使用价值并同时降低开发难度和减少重复开发工作,研究如何在普通的 UEFI 环境内下实现一个嵌入式 Linux 应用平台,并使其具有以下特点:

(1) 平台的构架采用模块化的设计方式。整体上作为 UEFI 的标准模块，完全符合 UEFI 接口规范，使其可以采用多种 UEFI 部署方案，灵活的部署在 UEFI 平台的系统上。模块内部采分层和模块化的设计，使其模块可以灵活的定制和更改；

(2) 具有完整的操作系统级别的支持，能够以 Linux 操作系统的方式进行开发和扩展。能充分发挥出操作系统级别的能力，提供比 UEFI 更加全面的软件和硬件支持，使其能充分利用硬件计算资源并同时提供高级电源管理功能；

(3) 通过采用抽象和虚拟技术进一步扩充系统的通用性，高可用性。充分利用虚拟化技术以及云计算技术，使平台可以充分发挥网络和并行运算的能力，同时减少资源浪费并提高资源的利用率。

本文在基于 COM-Express 2.0 规范的硬件模块上实现了一个 Embedded Generic Platform (EGP)，通过以模块方式内置在固件中的 EGP，能直接以操作系统的方式使用本地终端的计算资源以及网络资源，也可以更进一步经过模块化的扩展，使用云端计算资源。

**关键词** UEFI, LINUX, 云计算, 通用计算

# DESIGN AND IMPLEMENTATION OF UEFI BASED GENERAL APPLICATION PLATFORM

## ABSTRACT

The UEFI (Unified Extensible Firmware Interface) is the next-generation interface between hardware and operating system. UEFI uses software engineering theory, modularized design, C-Style parameter passing method. And it uses dynamic linked method to construct the system and has very good extensibility and unified calling interface.

It breaks the barrier of 16bit addressing which is the only mode the legacy BIOS (Basic Input Output System) could support, and allow processer runs under 32bit or 64bit protected mode or any other native mode introduced in the future. UEFI can detect and operate any hardware by loading the device's UEFI driver module. The UEFI driver module is run by the DXE (Driver Execution Environment) and is both backward and cross-platform compatible. Compare with BIOS, it has better error tolerance and correcting capability and it reduces the professional requirement and the development effort very much.

Compare with Coreboot which targets on booting Linux system, UEFI has clearer and more complete interface which can support not only Linux but also Windows, Mac OS. It also gets more support from the hardware vender like Intel, HP, Dell which makes it more popular.

Regarding these advantages of UEFI, it gets very popular by the motherboard venders. After 10 years of development, it is now the mainstream interface which obsoleted the legacy BIOS.

However, the design of UEFI can only use BSP in SMP system, uses one process instead of multiprocess, doesn't have ACL and uses Timer Interrupt only which limits it from utilizing the full capability of the hardware. Meanwhile, it doesn't get enough support from software venders and only hareware oriented venders develop application for it and these

applications are limited in very small area. Besides, different vendors may implement different interfaces which also makes the applications may have problems when they are using on different vendors' UEFI environment. At last, it may be much easier to develop than BIOS, however it is still not as easier as to develop for Linux.

To increase the capability and value of system which uses UEFI, and also to decrease the difficulty and increase reusing, this thesis researchs how to embed a linux into generic UEFI systems with the following features:

(1) The platform uses modularized design and its interface is completely compliant with the specification of UEFI. It can be deployed to UEFI system conveniently in different ways.

(2) The platform has full OS level support and full Linux application support. And it will provide more support to hardware than UEFI which can utilize the use of hardware computing resource.

(3) The platform has very generic character and high availability by using abstraction and virtualization.

The implementation of this thesis is an Embed Generic Platform (EGP) which in form of a UEFI module can utilize the computing resource in local endpoint, network and even cloud resource by further modularized extending.

**Keywords** UEFI, Linux, Cloud computing, Generic computing



## 目 录

1 绪 论 .....	1
1.1 研究背景 .....	1
1.1.1 PC 系统硬件构架 .....	1
1.1.2 PC 系统软件构架 .....	1
1.1.3 PC 系统硬件和操作系统间的接口 .....	2
1.2 研究的目标和意义 .....	3
1.3 论文结构 .....	4
2 重点相关技术分析 .....	6
2.1 多处理器系统构架 .....	6
2.2 统一可扩展固件接口 .....	7
2.2.1 UEFI 系统构架 .....	7
2.2.2 UEFI 运行周期 .....	7
2.2.3 UEFI 固件的存储构架 .....	9
2.2.4 UEFI 系统服务 .....	11
2.2.5 UEFI 可执行文件镜像 .....	12
2.3 虚拟化技术 .....	15
2.3.1 硬件抽象层虚拟化 .....	16
2.3.2 指令集虚拟化 .....	17
2.3.3 操作系统层虚拟化 .....	18
2.3.4 应用层虚拟化 .....	19
2.3.5 函数库层虚拟化 .....	19
2.4 云计算和云存储 .....	19
2.4.1 云计算 .....	20
2.4.2 云存储 .....	20
2.5 虚拟机封装方式 .....	20

2.6	本章小结 .....	21
3	通用计算平台需求分析与设计 .....	22
3.1	现状分析 .....	22
3.2	平台需求分析 .....	22
3.2.1	用例图 .....	22
3.2.2	用例规约 .....	23
3.2.3	平台性能需求 .....	24
3.3	平台总体架构 .....	25
3.4	平台硬件设计 .....	26
3.5	存储子系统 .....	27
3.5.1	本地存储方案 .....	28
3.5.2	远程存储方案 .....	29
3.6	平台软件设计 .....	30
3.6.1	整体构架 .....	30
3.6.2	基础设施层 .....	32
3.6.3	平台服务层 .....	32
3.6.4	引导头 .....	36
3.7	本章小结 .....	39
4	通用计算平台的实现 .....	41
4.1	硬件平台 .....	41
4.2	存储子系统 .....	42
4.3	基础设施层实现 .....	43
4.3.1	内核模块 .....	43
4.3.2	初始化模块 .....	46
4.3.3	基础服务模块 .....	46
4.4	平台服务层实现 .....	49
4.4.1	终端管理模块 .....	49
4.4.2	虚拟服务模块 .....	50
4.5	引导头实现 .....	53
4.5.1	引导环境 .....	53

---

4.5.2 引导流程 .....	53
4.5.3 引导头部署 .....	62
4.6 本章小结 .....	64
5 通用平台应用验证和扩展 .....	65
5.1 平台的验证环境 .....	65
5.2 基于终端的应用 .....	66
5.3 基于网络的应用 .....	68
5.4 基于云端的应用 .....	70
5.5 本章小结 .....	73
6 总结和展望 .....	74
6.1 本文总结 .....	74
6.2 项目成功因素分析 .....	74
6.3 展望 .....	75
参考文献 .....	76
附录 Console 输出日志 .....	78
致谢 .....	86
攻读学位期间发表的学术论文目录 .....	87

## 1 绪 论

### 1.1 研究背景

#### 1.1.1 PC 系统硬件构架

个人计算机 PC-AT 构架的目前主要采用如图 1-1 所示的 Intel 的 x86 体系构架，也即 IA32（Intel Architecture, 32-bit）<sup>[1]</sup>。

x86 架构属于冯·诺伊曼结构，于 1978 年推出的 Intel 8086 中央处理器中首度出现。Intel 80286 中央处理器中导入了 16 位保护模式。Intel 80386 中央处理器中导入了 32 位保护模式。Intel Pentium 4 中央处理器中导入了 EMT64，也就是后来的 Intel 64。在 Intel Pentium 4 中央处理器的 672 和 662 处理器上，Intel 导入了硬件的虚拟化支持，也即 Intel VT 技术。

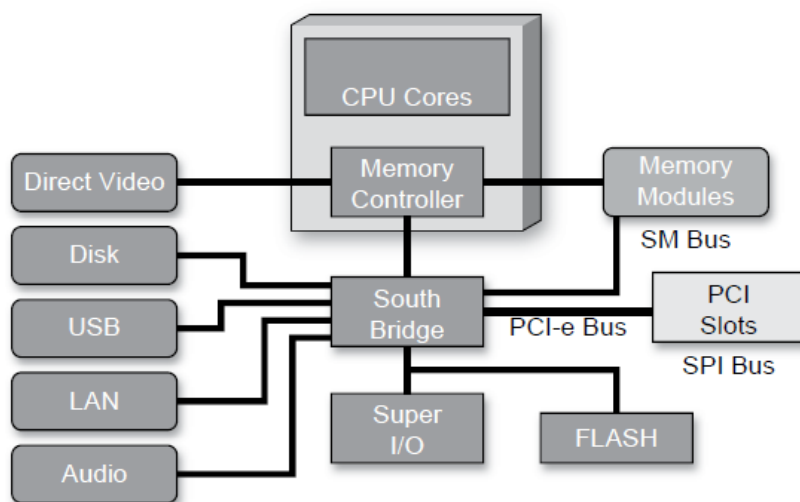


图 1-1 典型 PC 硬件构架<sup>[2]</sup>

Fig. 1-1 Typical PC hardware architecture<sup>[2]</sup>

#### 1.1.2 PC 系统软件构架

PC 系统的软件通常具有如图 1-2 结构特点。

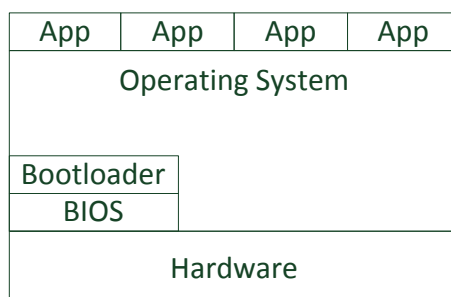


图 1-2 典型 PC 软件构架

Fig. 1-2 Typical PC software architecture

可以看出所有应用程序都运行在操作系统提供的运行环境下，只能从操作系统获取服务并访问底层硬件。

### 1.1.3 PC 系统硬件和操作系统间的接口

为了初始化硬件系统并使操作系统能够正常的工作，PC 的硬件厂商在硬件出厂之前便固化了一段代码，这段代码通常为 BIOS，Coreboot 或者 UEFI<sup>[3]</sup>。

#### (1) 基本输入输出系统(Basic Input Output System)

Basic Input Output System (BIOS)是传统 PC 上所采用的一种接口。它主要采用汇编语言编写，通常由各厂商以私有方式开发，因此代码很难通用或者重复利用。通常仅运行于实模式下，因此仅有 16 位代码的寻址能力。BIOS 利用挂载真实模式中断的方式增加硬件功能，无法支持所有的硬件，也无法发挥硬件的全部功能和能力。

#### (2) Coreboot

Coreboot 起源于 LinuxBIOS 的开源项目。它主要采用 C 语言开发，对 Linux 操作系统做了很多优化。然而，历经数年的发展，由于得不到操作系统厂商如 Microsoft 和 Apple 的支持，应用仍然局限在 Linux 方面，并未得到很好的发展。

#### (3) 统一可扩展固件接口(Unified Extensible Firmware Interface)

UEFI (Unified Extensible Firmware Interface, UEFI) 技术起源于英特尔安腾处理器 (Itanium) 平台，它有业界领导地位的英特尔和微软，惠普等众多企业共同推出的一项新的行业标准，目前主流的操作系统如 Windows vista 之后的 Windows 桌面操作系统，Windows CE .NET，采用 3.0 版本以上内核的 Linux 发行版，以及 Apple 公司的 Mac OS X 均已支持 UEFI，已发展成为新一代的计算机硬件和操作系统之间的主流接口。

UEFI 充分体现了软件工程的思想，采用模块化的设计，C 语言风格的参数堆栈传递方式，并且以动态链接的形式构建系统，具有良好的扩充性能和统一的调用接口。它能够运行于 32 位或 64 位保护模式，乃至未来增强的处理器模式下，可以轻松寻址到处理器的最大寻址范围最大运算能力。它利用加载 UEFI 驱动模块的形式，识别并操作硬件。UEFI 驱动被 UEFI 驱动运行环境（Driver Execution Environment, DXE）解释运行，具有向下兼容和跨平台支持的特性。较 BIOS 而言，其代码重用性更好，降低了对开发人员专业性质的要求和重复性劳动，并缩短了系统研发的时间。

得益于 UEFI 上述优点，UEFI 格外受到主板厂家青睐而得以迅速推广。UEFI 在近 10 年的推广后，已经成为主流，将使用传统的 BIOS 的微型计算机系统逐渐淘汰出市场。

## 1.2 研究的目标和意义

在目前的 PC 构架背景下，作为硬件系统提供商，往往希望能够从比操作系统更底层的层面来确保整个平台的稳定。过去在 BIOS 环境下，受限于诸多构架上的限制，很多功能的实现非常困难或者代价昂贵。UEFI 的出现，使得复杂的应用也有可能得以实现。

但是 UEFI 只是被设计为一种定义良好的接口，其主要目的依旧是为硬件和操作系统提供调用接口，使得硬件平台能够启动到操作系统。因此，UEFI 本身并不具备操作系统的很多重要特性，无法做到如操作系统一样，能充分发挥硬件的能力。由于这些的设计上的原因，使其在 SMP 系统中仍然只使用 BSP、使用单进程而不支持多进程、没有用户权限控制、仅使用时钟中断等等，限制了 UEFI 下的应用范围和并降低了其使用价值。

不仅如此，接口的具体实现仍然需要由 UEFI Firmware 厂商、主板厂商或者是 UEFI 的独立软件商(Independent Software Vender)来开发和维护。而 Firmware 厂商和主板厂商通常不具备独立完成大量不同环境下的应用的能力，不同厂商所自定义的接口也有差异。与此同时软件供应商支持力度也不够，使得其应用通常只有硬件相关厂商开发，也降低了其本身的价值。开发难度相比传统 BIOS 来讲虽有很大提高，但是相对 Linux 而言仍然太大，而且应用的范围也相当局限。

相比之下，大量的 Linux 优秀发行版的出现使得 Linux 系统迅速普及，在 GPL 版权的保护下，自由软件也得以蓬勃发展，共同催生出了大量的各种各样的优秀的 Linux 应用程序。这些应用不仅包含大量商业的私有版权的专业应用，也包含更加广泛、更

具有专业专用功能的免费开源软件。

针对传统的计算模式中，Legacy BIOS、Coreboot 以及 UEFI 作为固件方面的先天不足，本文主要研究如何在基于 UEFI 的 x86 微型处理器的平台上，提供一个通用的应用平台。

为了实现通用性，本文以模块化的方式将一个特别定制的 Linux 系统平台嵌入到主板的 UEFI 环境，如板载非易失性存储介质(NVRAM)或者 UEFI 的系统分区内，使其能直接从 UEFI 的运行环境下，进入到一个操作系统级别的平台甚至是虚拟的应用平台，从而能够不受 UEFI 接口和本地硬件限制，提供一个更通用的计算模式。

在 UEFI 环境内提供一个操作系统级别的应用平台，使得在该操作系统的平台上运行的应用，能够充分发挥硬件的能力，并且有了操作系统的支持，应用就就能突破 UEFI 的一些设计上的限制，更轻易的实现各种更灵活的应用。

同时，由于 Linux 经过多年的发展，其对硬件设备，网络协议的支持都已非常完善，并经过大量商业应用的验证。采用这种 Linux 应用间接提供支持的方式，能加快软件开发周期，降低开发成本，提高软件质量，提升企业的竞争力。并且采用 Linux 作为应用平台，应用更加通用，代码的重用性和稳定性能够较 UEFI 应用更进一步提高，极大的减少资源浪费。

### 1.3 论文结构

全论文共由 6 章组成。

第 1 章绪论，介绍本文研究的背景及意义和论文的主要研究目标以及论文的大致结构。

第 2 章重点技术分析，对于本文所使用到的技术进行简要的分析和总结，提炼出重点和关键技术。

第 3 章需求分析与平台设计，结合研究目标的现状及现有技术，从计算的通用性方面，对本文的设计进行需求挖掘。然后从技术层面，对本文平台的硬件和软件分别系统的展开设计。

第 4 章平台实现，按照设计要求，运用对应的工程技术实现本文。

第 5 章应用验证和扩展，采用针对性不同的应用方案，对平台进行验证也能够，也从侧面提供平台的实用性参考示例。

第 6 章总结和展望，对本文的阶段性成果进行总结，对不足进行进行反思，对未来的研究方向展开思考。



## 2 重点相关技术分析

为了实现本平台，将重点采用 UEFI 接口，在多处理器构架的硬件平台上，采用虚拟化和云计算技术来实现计算的通用性。本章对本平台使用的关键技术进行分析和总结。

### 2.1 多处理器系统构架

传统上的计算都是串行的运算模型，当 CPU 摩尔定率的增长速度无法满足人们对计算的需求时，采用多处理器进行并行运算变成了很自然的解决方案。

多处理器并行处理的构架包括三种主要的构架：

(1) 也被称为一致存储访问结构 (UMA : Uniform Memory Access): UMA 通常也称为对程多处理器(Symmetric Multi-Processor)技术。SMP 系统中的处理器间无主次或从属关系，对称工作。处理器间共享相同的物理内存空间，每个处理器访问内存中的任何地址所需时间也是相同的。SMP 的主要问题在 CPU 和内存之间的通信能力会成为整个系统的瓶颈。

(2) 非一致存储结构 (NUMA: None Uniform Memory Access) : NUMA 技术针对 SMP 扩展能力的不足进行了改进。它将多个 CPU 组成 CPU 模块，每个模块可以访问本地的存储，I/O 资源，然后用专用的高速互联模块将多个 CPU 模块连接起来，组成多 CPU 的高性能主机，克服了 SMP 结构的服务器在多 CPU 共享内存总线带宽时产生的系统性能瓶颈。由于 NUMA 技术访问远地内存的延时远远超过本地内存，因此当 CPU 数量增加时，系统性能无法线性增加。NUMA 的应用目前已经非常普及，ACPI 规范中已经定义了 NUMA 的支持<sup>[4]</sup>。

(3) 大规模并行处理(Massive Parallel Processing): MPP 是一种松耦合多机系统，它由多个 SMP 服务器通过一定的节点互联网络进行连接，协同工作，完成相同的任务，从用户的角度来看是一个服务器系统。其基本特征是由多个 SMP 服务器(每个 SMP 服务器称节点)通过节点互联网络连接而成，每个节点只访问自己的本地资源(内存、存储等)，是一种完全无共享(Share Nothing)结构，因而扩展能力最好，理论上其扩展无限制。在 MPP 系统中，每个 SMP 节点也可以运行自己的操作系统、数据库等。和 NUMA 不同的是，它不存在异地内存访问的问题。换言之，每个节点内的

CPU 不能访问另一个节点的内存。节点之间的信息交互是通过节点互连网络实现的，这个过程一般称为数据重分配 (Data Redistribution)。

UEFI 对于 SMP 构架的系统有所支持，但是由于设计上的原因，无法完全发挥 SMP 的应有能力，对于并行处理也没有支持。本平台设计中，将充分使用这三种技术来提高计算能力，充分利用计算资源，减少浪费。

## 2.2 统一可扩展固件接口

本平台采用完全符合 UEFI 接口规范的方式实现，本系统调用方式以及部署方式均符合其相应规范，因此本节对 UEFI 接口的关键接口部分进行分析。

### 2.2.1 UEFI 系统构架

一个典型的 UEFI 系统可如图 2-1 所示。

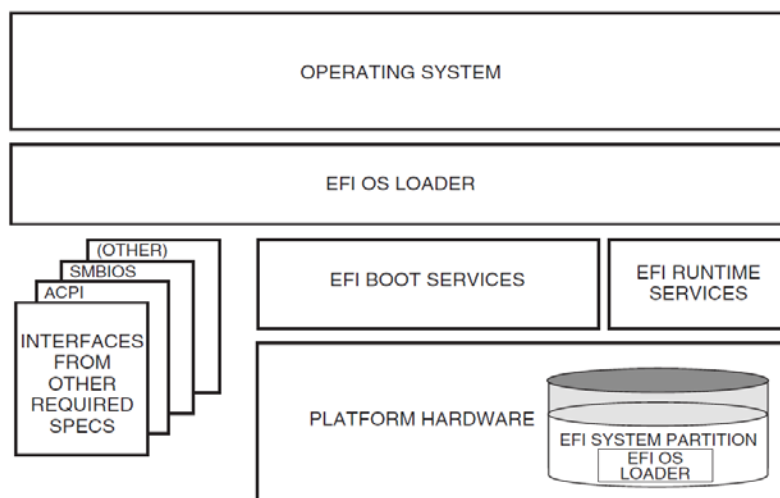


图 2-1 UEFI 系统构架

Fig. 2-1 UEFI system architecture

Boot Service 和 Runtime Service 是 UEFI 最重要的两大类型的服务，其中 Boot Services 将是本平台使用的主要服务。

### 2.2.2 UEFI 运行周期

UEFI 定义了如图 2-2 所示的 6 个典型的运行周期。

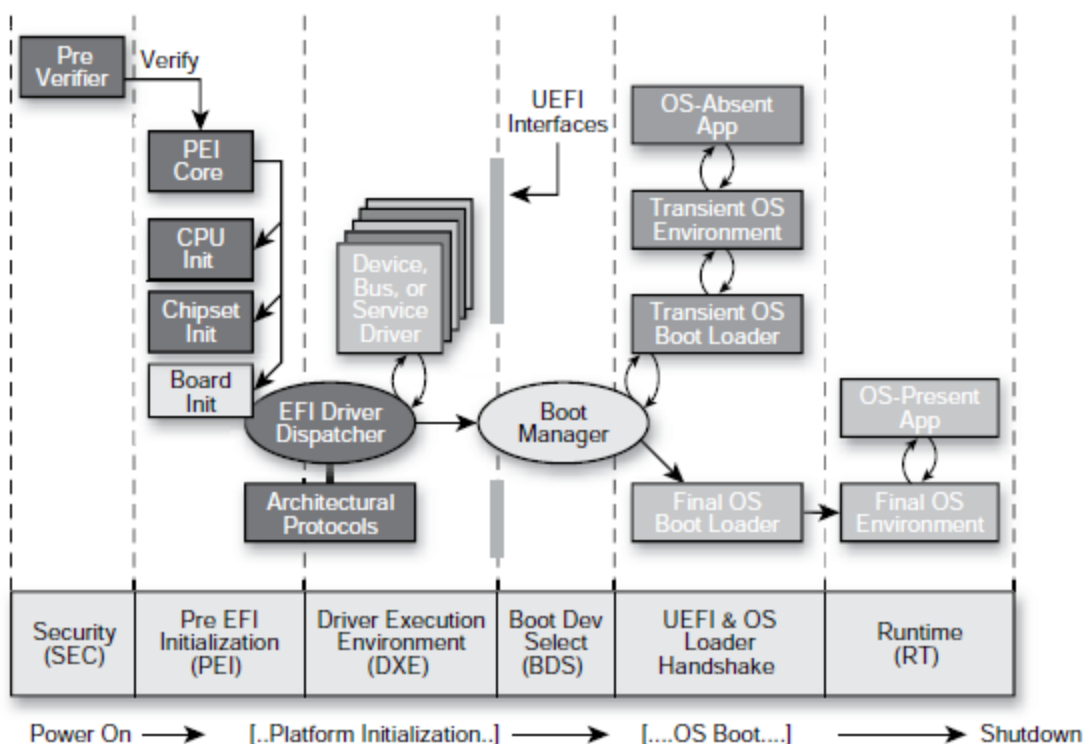


图 2-2 UEFI 周期

Fig. 2-2 UEFI phases

### (1) SEC Phase (Security)

开机之后，CPU 开始执行的第一条指令便是 SEC 阶段的指令。这时的内存还没有被初始化，因此，这一阶段最主要的工作将建立一些临时的 Memory，它可以是处理器的 Cache，或是 System Static RAM(SRAM)。然后，它将把 CPU 带入 Protect Mode。

### (2) PEI (Pre-EFI Initialization) Phase

PEI 阶段最主要的工作就是 Memory 的初始化以及一些必要的 CPU、Chipset 等等的初始化。由于这些都是没有压缩的 Code，所以要求越精简越好。另外，PEI Phase 还要确定系统的 Boot Path，初始化和描述最小数量的包含 DXE foundation 和 DXE Architecture Protocols 的 System RAM 及 firmware volume。

### (3) DXE (Driver Execution Environment) Phase

DXE 阶段是实现 EFI 的最重要的阶段，大部分的工作都是在这个阶段实现的。

### (4) BDS (Boot Device Select) Phase

BDS 阶段的主要工作包括初始化控制台设备，尝试加载列在环境变量 `Driver####` 和 `DriverOrder` 上的 `Driver`，并尝试从列在环境变量 `Boot####` 和 `BootOrder` 上的启动设备列表中启动。

#### (5) UEFI & OS Loader Handshake Phase

该阶段是 OS Loader 运行的第一阶段，OS Loader 和 UEFI 系统进行握手动作，尝试引导操作系统。

#### (6) RT (Runtime) Phase

当 OS 或者 OS Loader 呼叫了 Boot Service 的 `ExitBootService()` 之后，系统就进入了 Runtime 阶段。此时，DXE Foundation 和 Boot Service 都已终止，只有 EFI Runtime Service 和 EFI System Table 还可以继续被使用（需要操作系统支持）。

在以上这六个阶段中，本平台将从 UEFI & OS Loader Handshake Phase 即开始运行，并持续运行至 Shutdown(After life)。

### 2.2.3 UEFI 固件的存储构架

为了在有限的存储器件内有效的存储固件，UEFI 在其 Platform Initialization Specification 规范中定义了固件所使用的文件系统 Firmware File System<sup>[5]</sup>。

#### (1) Firmware Device (FD)

Firmware Device 是一个物理的存储空间，用来保存固件的数据和代码。通常存储固件的器件为 Flash ROM，它在逻辑上被看作成一个设备，就是 Firmware Device。在实际应用中，固件可能被存储在多个 Flash ROM 中，各个 Flash ROM 也可能大小不一并采用不同的存储技术，但是，他们在逻辑上成为一个完整的更大的 Firmware Device。

#### (2) Firmware Volume (FV)

Firmware Volume 是逻辑分区设备，它是固件存储数据或者代码的基础单位。Firmware Volume 类似于磁盘分区中的卷的概念。如图 2-3 所示，每个 Firmware Volume 都是一个文件系统。

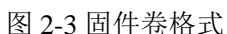


Fig. 2-3 Firmware volume format

### (3) Firmware File

FF 类似文件系统下的文件（File）的概念，在 FV 内同样以 File 来组织管理。FV 内的 Firmware File 都采用 GUID 命名以避免重复。

#### (4) Firmware File Section

FF 按照 FFS 进行组织。每个 File 里面允许分成不同的段 (Section)，每种类型的 Section 都存储特定的数据。比如 PE32 类型的 Section 存储的是代码。而 RAW 类型的 Section 则存储包括 ACPI Tables 这样的纯数据。段可以是任何支持的类型，而且段内还可以存储另一个压缩的 FV。

### (5) 存储系统访问接口

UEFI 的 PEI 阶段和 DXE 阶段均有定义读取 FV 的标准接口 Firmware Volume Block Protocol，使用这些协议可以以文件级别访问固件。

由于 UEFI 的固件存储机制并不适合通常的文件操作，如删除或者改写。所以，在进行 UEFI 固件的更新时，通常采用直接访问物理硬件的方式进行更新。

UEFI 存储系统的访问接口如图 2-4 所示。

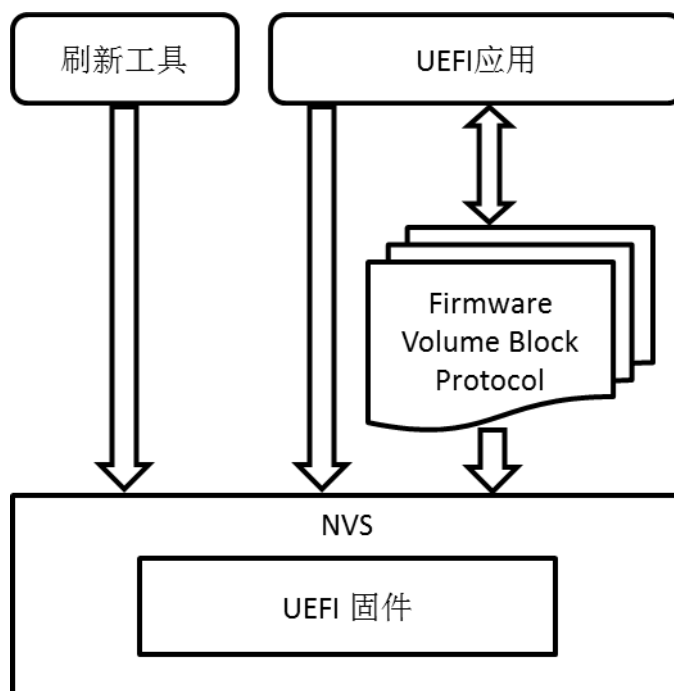


图 2-4 UEFI 固件读写方式

Fig. 2-4 Methods to access UEFI firmware

本平台根据应用方式的不同，可采用 FSS 的方式，将平台模块存放在 UEFI 固件内。

#### 2.2.4 UEFI 系统服务

UEFI 采用句柄数据库(Handle Database)和协议(Protocol) 的概念来管理可执行代码和提供这些系统服务。Handles 由一个或多个 Protocol 组成。Protocol 是由 GUID 来命名的数据结构体，它可能是空，可能包含数据，可能包含服务程序，或者同时包含两者。

如图 2-5 所示，UEFI 通过 Handle 和 Protocol 来定位所有的可执行代码，而本平台在运行时，将被 UEFI 系统传入 EFI\_SYSTEM\_TABLE，籍此本平台也将能够访问到所有的可执行代码。

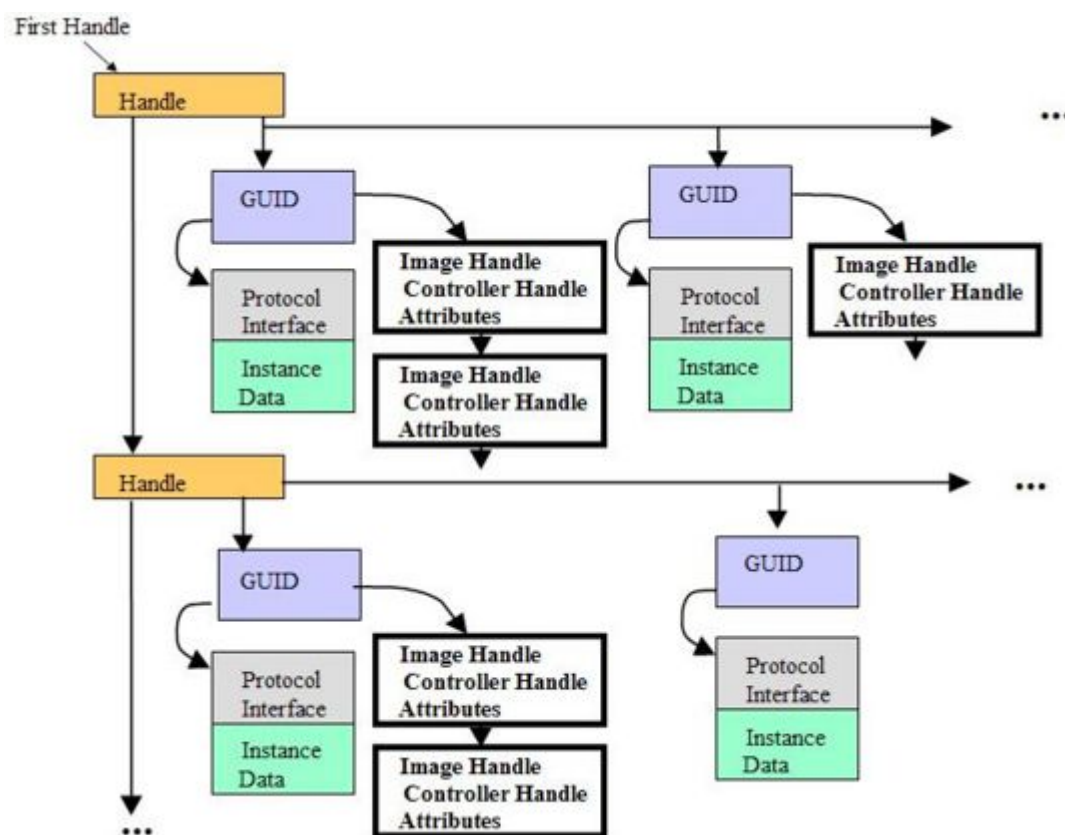


图 2-5 UEFI 句柄和 Protocol

Fig. 2-5 UEFI Handles and protocols

### 2.2.5 UEFI 可执行文件镜像

UEFI 可执行文件镜像 (UEFI Image) 是一类包含可执行代码的文件。

UEFI Image 依据驻留内存的方式和运行周期，可分成 UEFI 应用程序(UEFI Application)、操作系统引导程序(OS Loader)和 UEFI 驱动(UEFI Driver)三类。

UEFI Image 类型之间的关系如图 2-6 所示。



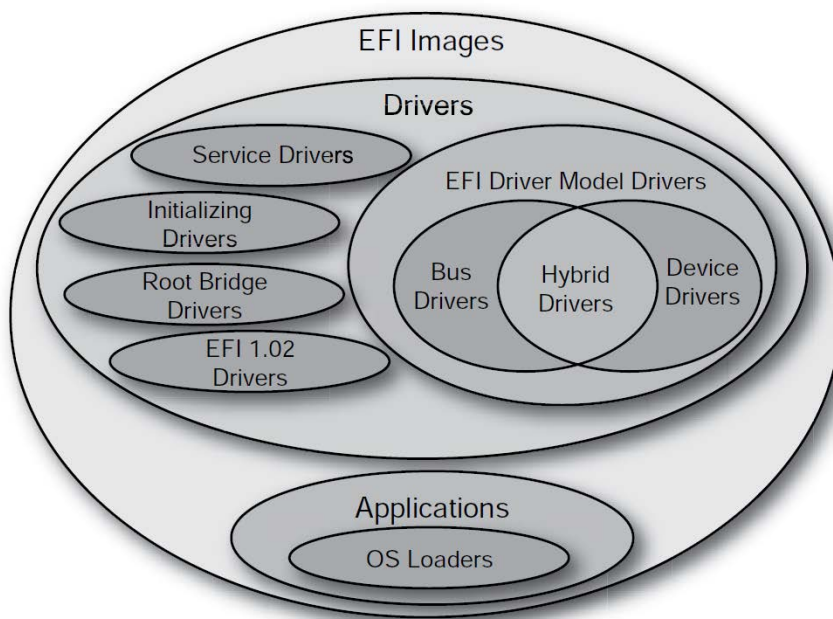


图 2-6 UEFI Image 类型和相互关系

Fig. 2-6 UEFI Image Types and Their Relationship to One Another

为了保持良好的兼容性，UEFI 依照如图 2-7 所示 PE/COFF 的文件格式<sup>[6]</sup>，定义出了 PE+文件格式子集。它新增了多个文件头标志，用以便区分 UEFI Image 和普通的 PE 格式的可执行文件。+号表示它在 PE32 基础上提供了 64-bit relocation fix-up 的扩展。经过扩展后的镜像文件便可在 IA-32 处理器、Itanium 处理器、x64 处理器、ARM 或者 UEFI 的 EFI Byte Code (EBC)环境下运行。



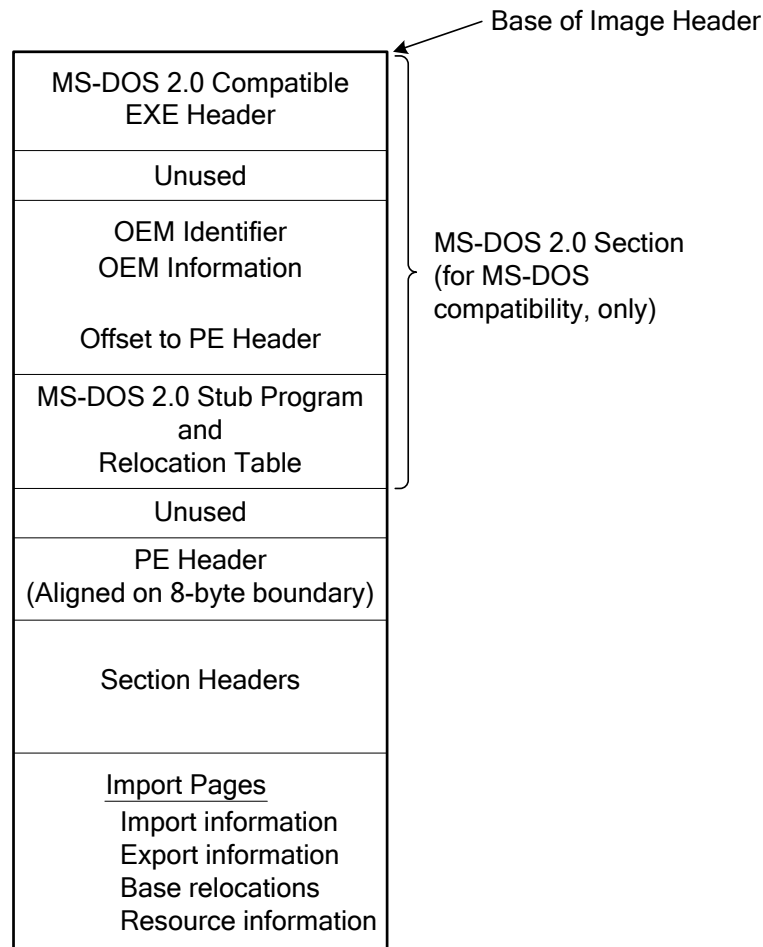


图 2-7 PE 文件头格式

Fig. 2-7 PE Header format

镜像所支持处理器类型和镜像类型信息被存放在在文件的头部，目前已有如图 2-8 所示的处理器类型：

```
// PE32+ Machine type for EFI images
#define EFI_IMAGE_MACHINE_IA32 0x014c
#define EFI_IMAGE_MACHINE_IA64 0x0200
#define EFI_IMAGE_MACHINE_EBC 0x0EBC
#define EFI_IMAGE_MACHINE_x64 0x8664
#define EFI_IMAGE_MACHINE_ARMTHUMB_MIXED 0x01C2
```

图 2-8 典型的 PE+文件头标识

Fig. 2-8 Typical PE+ File Header Signature

UEFI Image 被 Boot Service 的 LoadImage()协议加载。UEFI Application 类型的 Image, 可作为可执行文件在 UEFI Shell<sup>[7]</sup>内运行。

为了保持良好的通用性, 本平台的模块整体上也采用 UEFI Image 的一种或者多种实现。

### 2.3 虚拟化技术

虚拟机技术首次出现于 20 世纪 60 年代的 IBM7044 计算机上。它基于麻省理工学院(MIT)开发的分时系统 CTSS(Compatible Time Sharing System)和曼彻斯特大学的 Atlas 项目。

自 20 世纪 90 年代后期开始, 虚拟化技术随着计算机硬件的飞速发展逐渐在个人计算机领域迅速发展。

虚拟化技术可在不同的抽象层实现, 按照抽象程度的不同, 分成如图 2-9 所示的虚拟化技术层次:



图 2-9 虚拟化抽象层次

Fig. 2-9 VM abstract level

UEFI 在其驱动运行环境（Driver Execution Environment, DXE）中采用了 EFI Byte Code 的构架，它用虚拟化的手段来实现驱动的跨平台支持。但是这种虚拟化的支持仅局限与其驱动执行环境，本平台将进一步通过使用虚拟化技术，来实现跨硬件平台的通用应用。

### 2.3.1 硬件抽象层虚拟化

硬件抽象层的虚拟机技术利用客户系统环境和虚拟机宿主平台的相似性来减少执行客户系统指令的延迟。目前，大多数的商业服务器虚拟化产品，都是通过使用这种技术来实现高效、实用的虚拟化系统。这种技术利用虚拟机监视器（Virtual Machine Monitor, VMM）作为隔离代码运行环境的中间层。

这类虚拟机通过 VMM 提供了一个物理机器的抽象，它允许操作系统假设自身可以直接在硬件上运行，VMM 为其上运行的客户操作系统提供硬件映射。从操作系统的角度看，运行在虚拟机上与运行在其对应的物理计算机系统上一样。

按照 Goldberg 的定义，虚拟机监视器(Virtual Machine Monitor)是能够为计算机系统创建高效、隔离的副本的软件<sup>[8]</sup>。这些副本即为虚拟机（Virtual Machine, VM）。根据 VMM 在系统中的位置和实现方法的不同，Goldberg 定义了两种虚拟机监视器模型，即 Type I VMM 和 Type II VMM:

#### (1) Type I VMM

Type I VMM 通常都是以一个轻量级操作系统的形式实现。Type I VMM 在操作系统之前预先安装，然后在此虚拟机监视器之上安装客户操作系统，其特点包括:

- 需要硬件支持
- 虚拟机监视器作为主操作系统
- 运行效率高

个人计算机领域的常见产品包括:

- VMware ESX/ ESXi, ESX Server<sup>[9]</sup>
- Xen 3.0 以后版本<sup>[10]</sup>
- Virtual PC 2005
- Hyper-V
- KVM

#### (2) Type II VMM

Type II VMM 则是安装在已有的主机操作系统（宿主操作系统）之上，此类虚拟机监视器通过宿主主操作系统来管理和访问各类资源（如文件和各类 I/O 设备等），其特点包括：

- 虚拟机监视器作为应用程序运行在主操作系统环境内
- 运行效率通常比类型 I 低

个人计算机领域的常见的产品包括：

- VMware Server, Workstation 8
- Xen3.0 以前版本
- Virtual PC 2004
- VirtualBox

为了更好地实现虚拟化，硬件厂商还提供了硬件辅助虚拟化（hardware-assisted virtualization）技术来提高虚拟化的效率：

- AMD 虚拟化（AMD Virtualization）
- 英特尔虚拟化技术（Intel Virtualization Technology, Intel VT），IO 虚拟化（Virtualization for Directed I/O, VT-d）<sup>[11]</sup>
- IOMMU（Input/Output Memory Management Unit）

### 2.3.2 指令集虚拟化

指令集虚拟化也被称为或者二进制转换。一个典型的计算机系统由处理器、内存、BUS、硬盘驱动器、磁盘控制器、定时器、多种 I/O 设备等等部件组成。在这种虚拟化方式下，模拟器通过将客户虚拟机发出的所有指令翻成本地指令集，然后才在真实的硬件上执行。指令的转换通常都是动态实时的，只有在当代码执行时，才会会对代码的某个段进行转换。如果出现分支情况，就会导入新代码集并进行转换。这使它与缓存操作非常类似，后者是将指令块从内存移动到本地快速缓存中执行。

指令级虚拟化的代表系统包括 Bochs 和 QEMU 等。这种模型在 Transmeta 设计的 Crusoe 中央处理单元（CPU）中也得到了使用，其二进制转换由专利技术 Code Morphing Software (CMS)<sup>[12]</sup>实现。

完全虚拟化解决方案中也有采用这种方式，在运行时进行代码扫描来查找和重定向特权指令，以解决特定处理器指令集的一些问题<sup>[13]</sup>。

### 2.3.3 操作系统层虚拟化

一个应用程序的典型运行环境包括：操作系统、函数库、文件系统、网络资源、环境变量等。作为应用程序，通常只跟这些运行环境交互，而自身是无法区分是运行在物理系统内，还是运行在虚拟系统中的。操作系统层虚拟化技术，即在宿主操作系统上动态维护多个软件运行环境，以此来同时运行维护多个虚拟系统。

在服务器领域广泛应用的 Virtual Private Server (VPS) 技术，便是将服务器的操作系统环境分割成多个彼此隔离的虚拟运行环境。

常见的技术比如：

- FreeBSD 上的 Jail。它首创了 Jail 技术，用于把操作系统划分成多个独立的环境。每个 Jail 都有可以独立管理的典型操作系统资源，如进程、函数库、文件系统、网络资源等。用户对资源的访问范围被限制在该用户所在 Jail 的内部。Jail 虚拟化技术在隔离应用程序方面具有非常肯定的应用价值。
- Solaris 的 Zone 以及 Containers。Zone 是一组处理器资源、内存资源以及 I/O 总线资源等物理硬件资源的定义，每个 Zone 都可以运行独立的 Solaris 副本。Container 只需运行一份 Solaris 操作系统软件，但对这份 Solaris 里的资源进行了优化和提效，使得虚拟出多份特制的 Solaris 资源供应用程序相对独立地运行。
- Linux-VServer。对 Linux 通过修改 Linux 内核，对内核进行虚拟化，这样多个用户空间环境（又称为 Virtual Private Server）就可以独立运行，而无需要互相了解，实现用户空间的隔离。
- OpenVZ，它与 Linux-VServer 类似，基于 Linux 操作系统内核的虚拟化技术，只能虚拟 Linux 系统。
- Virtuozzo 的 Parallels Virtuozzo Containers，基于 OpenVZ 并在区域密度、虚拟机管理工具等多方面提供了大量改进。
- UML (User Mode Linux)，它是一个是让一个 Linux 作为一个独立进程运行在另一个 Linux (宿主 Linux) 之上的开源项目，它提供了同时运行多 Linux 的虚拟化方式。
- IBM 的工作负载分区(Workload Partition)，可以实现新一级别的 AIX 虚拟化功能，实现了在不修改应用程序代码的情况下，在不同的 AIX 实例之间的移动以及实现了应用程序对于资源的更好的使用。

- iCore Virtual Accounts

#### 2.3.4 应用层虚拟化

编程语言级虚拟，它与系统库级虚拟都一样是处于应用层的。但是，编程语言级虚拟有些不同。它提供了一种语言的运行环境，比如 JVM、微软.net CLI 和 Parrot。

Java 编译器将 Java 语言翻译成 Java 字节码（一种中间语言），在运行时采用 JVM 为所在平台的操作，比如 windows 或 Linux，其提供指令解释并执行。JVM 将这些字节码翻译到合适平台的指令，而这种字节码指令具体如何执行，是隐藏在 JVM 中的，用户只需要在 OS 下安装对应的 JVM 版本即可。

微软.net CLI 也采用了类似的技术，不同的是它能提供更多语言的虚拟执行环境。

#### 2.3.5 函数库层虚拟化

应用通常通过 API，调用系统提供的函数库，实现各种操作等。因此可以在应用程序和提供提供的运行库函数之间引入中间层来虚拟库函数的 API 接口，通过虚拟 API 来接口来提供不同的操作。典型的例子有 Cygwin、Wine 等。

### 2.4 云计算和云存储

最早的云计算由并行化计算衍生而来。当计算机硬件能力无法满足人们对运行速度越来越高的要求时，人们便开始采用计算机并行的进行计算。

然而并行计算模式虽然提供了强大的运算能力，但是在实际应用过程中如果为了满足特定服务而构建出了一个并行系统，却经常会因为大部分时间服务并不需要如此高性能的运算能力而造成大量的资源浪费。在环境和能源问题越来越严峻的情况下，为了解决这个矛盾，各专家机构提出由一个强大的计算中心来提供计算服务，按照客户的需求来弹性的提供计算服务的方式，也就是“按需服务”的云计算。

目前对于云计算还没有统一的定义。在典型的云计算模式中，用户通过终端接入网络，向“云”端提出计算需求；“云”接受请求后分配组织资源，通过网络为“终端”提供计算服务。用户终端占用的资源可以大大精简，计算与处理的实际过程都靠“云”端上的资源去完成。用户所需的应用程序并不实际运行在用户的个人电脑、手机等终端设备上，而是运行在网络另一端的大规模服务器集群中；用户所处理的数据也无需存储在本地，而是保存在网络另一端的数据中心里。云服务提供商提供专业的服务，保证

数据中心和计算资源的安全和稳定以及其他管理服务。因此，在任何时间和任何地点，用户只要能够通过网络连接到达云端，即可使用云端的资源。

#### 2.4.1 云计算

计算云依照计算服务的层次可细分为基础设施即服务（IaaS）、平台即服务（PaaS）和软件即服务（SaaS）：

（1）IaaS(Infrastructure-as-a-Service)：基础设施即服务。消费者可以通过 IaaS 提供商获取服务器、操作系统、磁盘存储、数据库和/或信息资源等基础资源的服务。IaaS 的代表商业产品是亚马逊的 AWS(Elastic Compute Cloud, EC2)。

（2）PaaS(Platform-as-a-Service)：平台即服务。PaaS 实际上是指将软件研发的平台作为一种服务，以 SaaS 的模式提交给用户。因此，PaaS 也是 SaaS 模式的一种应用。Google 的 App Engine 和微软的 Azure(微软云计算平台)都采用了 PaaS 的模式。

（3）SaaS(Software-as-a-Service)：软件即服务。它是一种通过 Internet 提供软件的模式，用户无需购买软件，而是向提供商租用基于 Web 的软件应用，来获取相应的软件服务。

#### 2.4.2 云存储

云存储是在云计算概念上延伸和发展出来的一个新的概念，是指通过集群应用、网格技术或分布式文件系统等功能，将网络中大量各种不同类型的存储设备通过应用软件集合起来协同工作，共同对外提供数据存储和业务访问功能的一个系统。云存储的代表产品是 Amazon 的 Amazon Simple Storage Service (S3)，Google 的 Cloud Storage 以及 Openstack 的 OpenStack Object Storage (SWIFT)等。

### 2.5 虚拟机封装方式

虚拟机通常按照特定格式封装在一个或多个文件当中。

#### （1）厂商自有文件格式

常见的虚拟机和云服务厂商自有文件格式如表 2-1 所示。



表 2-1 虚拟机封装格式

Table2-1 VM Packing Format

厂商	配置文件	数据镜像文件
Amazon		AKI, ARI, AMI
Eucalyptus	XML	EMI
qemu-kvm	XML	QCOW2, RAW
Microsoft	XML(Hyper-V)	VHD, VHDX
VirtualBox	.vbox	VDI (兼容 VHD, VMDK 等)
Vmware	VMX	VMDK 等
OpenStack	XML	AKI, ARI, AMI, RAW, ISO, VHD, VDI, QCOW2, 以及 VMDK

## (2) 开放式虚拟机格式(Open Virtualization Format)<sup>[14]</sup>

DMTF 组织为了规范虚拟机的封装、描述以及部署方式，制定了 Virtualization Management (VMAN) 系列标准。开放式虚拟机格式(Open Virtualization Format) 作为其中的重要标准之一，以开放、安全、可移植、高效率和可扩展的特点被越来越多的云计算厂商所支持。目前 OVF 版本 1.1.0 已被 ANSI 接纳，成为 ANSI standard INCITS 469-2010 标准<sup>[15]</sup>。

OVF 是一种 XML 格式，描述一个 VM（磁盘格式）的所有特性，包括组成 VM 的磁盘，VM 的网络配置，VM 需要的处理器和内存资源，描述虚拟设备创建程序的各种元数据，VM 的目标以及操作系统描述。

采用 OVF，可以高效、灵活、安全的实现虚拟机在不同的虚拟化平台之间转移，从而为客户提供相对的平台独立性和供应商独立性，客户完全可以将一个 OVF 格式的虚拟机部署到另一个虚拟化平台上。

## 2.6 本章小结

本章以个人计算机构架为基础，对实现通用计算所需的技术如虚拟化及云计算等关键技术进行了分析，对需要遵循的业界规范如 UEFI 中的进行了总结。



## 3 通用计算平台需求分析与设计

本章对平台的需求进行捕获，然后针对平台的需求进行分析和设计。

### 3.1 现状分析

统一可扩展固件接口做为新一代的硬件和操作系统之间的桥梁，得到了硬件和固件厂商的大力推广。

由于构架上的突破，使得在 UEFI 环境中开发应用更加容易。并可在 UEFI 中内置 Shell，为 UEFI 应用程序提供命令解释和运行环境支持，大大降低了对 MS-DOS 的依赖。同时 UEFI 系统目前不仅支持 IA32，IA64，AMD64 全系列处理器构架，而且对于 ARM 平台也提供了支持。

然而，UEFI 仅仅被设计为一个硬件和操作系统的接口。所以在设计上仍然有很多缺陷，导致其无法和操作系统中运行的应用程序一样容易开发。而且不同主板的 UEFI BIOS 由于搭载的模块和驱动并不一致，UEFI 和 OS 还必须同为 32 bit 或者 64 bit 模式，这也导致跨平台的通用性大打折扣。

对比之下 Linux 类系统由于开源，近年得到了飞速的发展。Linux 下不仅拥有非常丰富而且功能强大的自由软件，而且硬件驱动也更加丰富和完善。相比 UEFI，Linux 具有更多的应用和更好的兼容性和通用性。

得益于硬件性能的飞速发展，虚拟化技术的应用也越来越广泛。通过抽象和虚拟化的手段，云计算方式也逐渐被接受并发展成熟。由于云计算具有按需提供计算能力的特点，其具有更加灵活的应用方式，因此也更具有通用性。

### 3.2 平台需求分析

为了提高应用的通用性，本文在 UEFI 系统之上结合虚拟机和云计算技术实现应用程序的跨硬件平台应用。

#### 3.2.1 用例图

根据本通用平台的需求，可定义如图 3-1 所示用例图<sup>[16]</sup>。

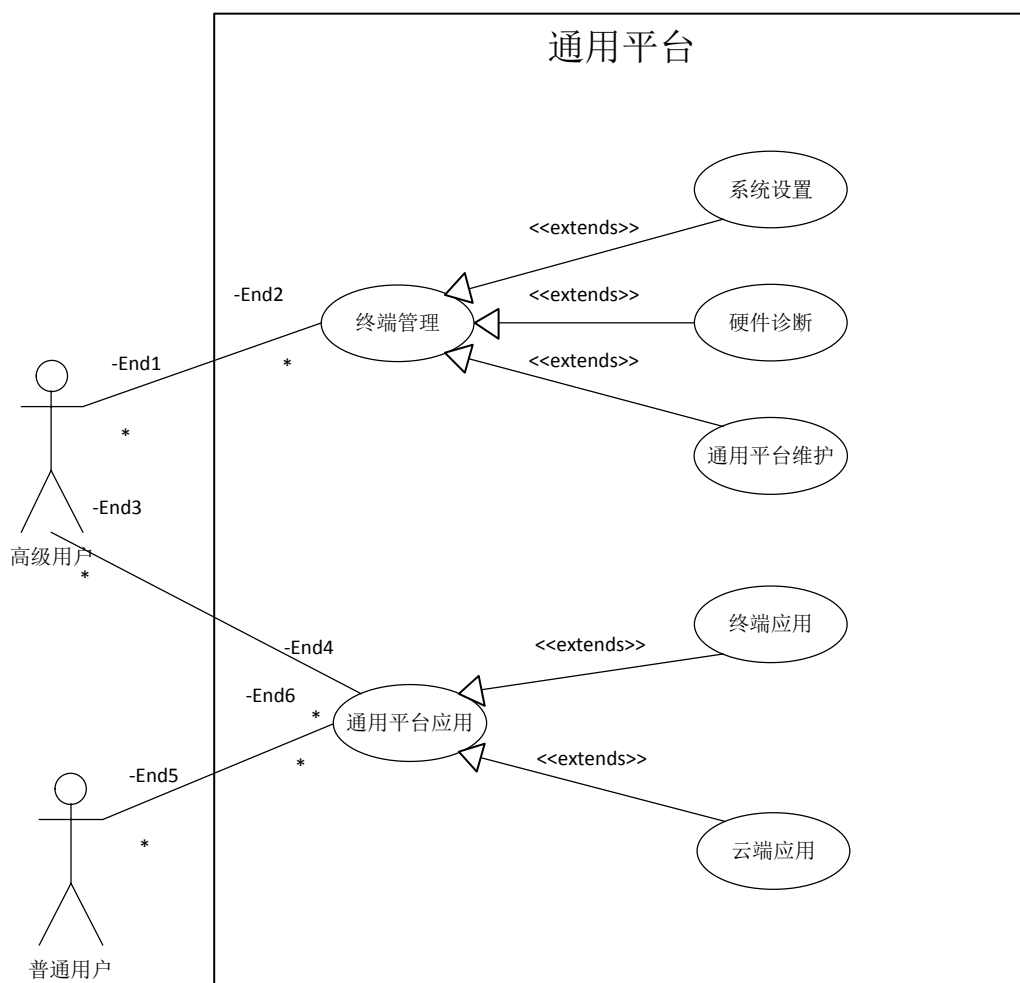


图 3-1 系统用户用例图

Fig. 3-1 End User Usecase Diagram

在图 3-1 的用例图中，高级用户为系统管理员用户，普通用户为最终使用本平台的业务人员。

### 3.2.2 用例规约

在平台用例图的基础上，本节详细定义关键用例的用例规约。

#### 1) 用例名称：终端管理

- a) 参与者：高级用户
- b) 简要说明：高级用户如 IT 人员或者厂商的客户服务人员
- c) 前置条件：具有应用授权。

d) 基本事件流:

参与者选择进入终端管理平台;

进入平台后, 根据管理的需要, 选择对应的管理应用;

2) 用例名称: 通用平台应用

e) 参与者: 所有人员

f) 简要说明: 参与者根据应用的需求, 选择合适的应用平台以完成应用。

g) 前置条件: 具有应用授权。

h) 基本事件流:

参与者选择进入通用应用平台;

用户登录对应平台后, 完成所需应用;

### 3.2.3 平台性能需求

针对平台的应用环境, 本节对平台的性能需求方面进行详细的分析和定位。

#### (1) 可用性需求

为了使该平台在实际业务实施环境中更加通用, 本平台需要保持高可用性。平台的可用性, 即平均一天由于宕机或系统故障等原因停止服务的时间应越短越好。典型的云服务可用性可高达 99.99%, 即年度停机时间不超过 53 分钟, 达到具有故障自动恢复能力的可用性。

通常 PC 机服务器在可靠性方面有所欠缺, 因此, 本平台需充分考虑容错和灾难备份。

#### (2) 敏捷性需求

实际应用中, 计算资源的负载通常处于变化中。本平台须具有灵活敏捷的扩展性能。平台不仅需要支持目前主流的应用, 还需要对未来的主流或者行业应用加以考虑。

### 3.3 平台总体架构

根据本文通用平台的需求，采用抽象和虚拟的方法兼容多种构架，并充分考虑可扩展性进行设计，如图 3-2 所示。

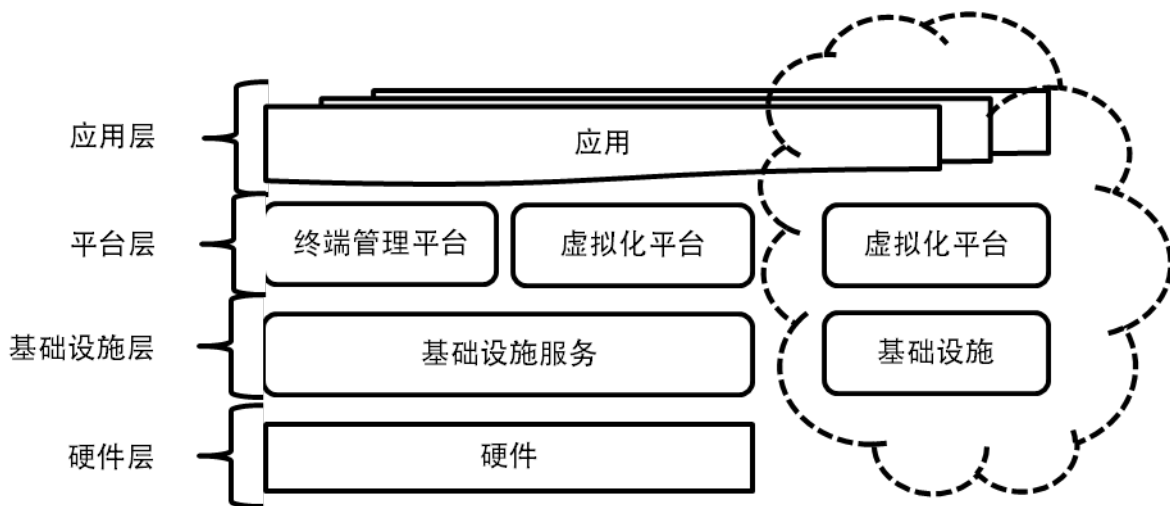


图 3-2 平台的架构

Fig. 3-2 Platform architecture

本平台的系统架构可分为 4 层：

#### （1）应用层：

普通的通用应用应用层内运行。类型不同的应用，由不同的平台服务模块提供对应的服务。

#### （2）服务平台层

本层通过虚拟的方式，将计算资源抽象，淡化用户对物理资源的依赖，为应用层提供底层的调用和操作系统级别的其他支持。根据应用的不同，虚拟平台层可配置为不同构架的服务，如 ARM 平台的 Linux 操作系统或者 x86 构架的 64bit Windows 操作系统。

#### （3）基础设施层

基础设施层向上层的虚拟平台提供虚拟的硬件访问和控制服务。根据部署的位置，分为终端基础设施和云端基础设施。

#### （4）硬件层

硬件层是物理上提供运算和存储等计算能力的抽象层。根据计算产生的位置，类似的，也分成终端硬件层和云端硬件层。

通过细化并分层，我们可以大大降低系统的复杂性，提高系统的可维护性和可扩展性。

### 3.4 平台硬件设计

本文设计的通用平台主要基于 UEFI 系统，UEFI 系统目前已经支持了 IA32，IA64，AMD64 全系列处理器构架平台，并且 UEFI 从 2.3 以后的版本还增加了对于 ARM 处理器平台的支持。因此平台硬件设计，根据实际需求可有多种灵活选择。

本文中的终端硬件层以普通的 PC-AT 构架的个人计算机系统为设计基础，系统构架如图 3-3 所示。

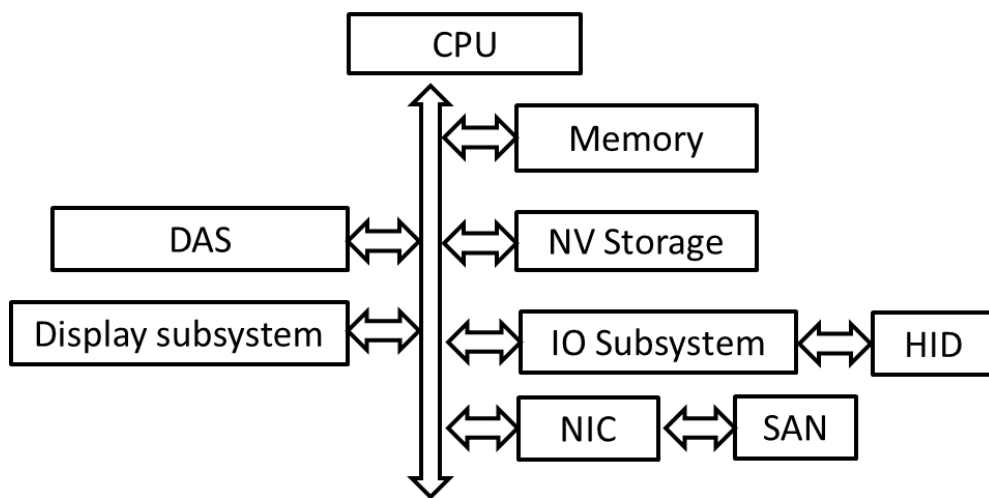


图 3-3 PC-AT 的硬件平台构架

Fig. 3-3 PC-AT hardware platform architecture

#### (1) 中央微处理器（CPU）

CPU 是个人计算机的主要的通用核心计算资源。为了更好地将硬件资源抽象化，硬件平台均采用支持硬件虚拟化(如 Intel VT 或者 AMD-V)技术的平台搭建。

## （2）内存（Memory）

内存在冯洛伊曼体系的计算机构架中，不仅存储可执行代码也存储数据。个人计算机目前多采用 DRAM 作为主存。

## （3）非易失性存储（NV Storage）

为了存放 Boot Loader，UEFI 或传统 BIOS，一般采用非易失性存储。在 ARM 及 SOC 系统中，通常采用 NOR 和 NAND Flash，在目前主流的 PC-AT 系统中，一般采用 SPI Flash 芯片，常见容量多数从 2MB 到 16MB。

## （4）显示子系统和人机接口设备（Display Subsystem and HID）

为了实现人机交互，个人计算机系统一般都包含显示子系统提供视觉输出，键盘鼠标设备来提供用户输入。

PC 的显示子系统，通常都带有 Framebuffer 支持，可以很好的提供通用性支持。键盘和设备类输入设备，通常都作为 HID（Human Interface Device），在各种操作系统和硬件平台上均可很好的工作。

无端系统(Headless System)没有配置上述这两类设备，通常它们采用网络或者 IO 接口来提供输入输出。常见的网络接口方式为 Telnet 或者 SSH，服务器或者高端 PC 机也可采用 IPMI、SNMP 以及 SOL（Serial Over Lan）等方式。IO 接口的方式一般为 Serial Port，早期还有 Parallel Port（Pinter Port）的方式。

## （5）网络接口（NIC）

通过网络接口，个人计算机可连接至网络，实现信息交换和资源共享。个人计算机一般都采用千兆以太网（Gigabit Ethernet）作为标准配置，但在某些特殊应用领域，40GB 的以太网网络也有展开应用。

## （6）输入输出子系统（I/O Subsystem）

I/O Subsystem 通常被用来连接低速的扩展设备，如串行口（Serial Port，RS-232），PS/2，HID，USB，SATA 等设备均通过 IO 总线连接。

# 3.5 存储子系统

根据应用需要，结合不同的存储环境、网络环境，本平台可以灵活的选择存储子系统。

本文采用的存储系统(Storage Subsystem)主要包括：非易失性存储（NVS）、本地直接存储（DAS）、存储区域网络（SAN）以及云存储(Cloud Storage)。

- 非易失性存储（NV Storage）：NVS 通常采用 EEPROM 或者 SRAM 芯片提供。EEPROM 可以存放代码或者数据，但是速度较慢，通常在使用过程中需要将其中的内容压拷贝到主存中运行。
- 本地直接存储（DAS）：DAS 是比较常见的存储方案，常用的方案如 IDE、SCSI、SATA、CD、DVD 等。DAS 使用比较简单，但是容量通常不大，为了提高存取速度和增加安全性，通常会使用 RAID 方案来提供更好的性能。
- 存储区域网络（Storage Area Network）：SAN 一种采用存储区域的网络来扩充存储的方案。目前 SAN 主要包括 FC-SAN 或者 iSCSI。前者通过专用的 Fiber Channel 作为接口，后者可通过专用的 HBA 卡，或者基于以太网卡的专用软件实现。
- 云存储（Cloud Storage）：云存储在云计算的基础上发展起来。云存储通常采用对象（Object）的方式来存储数据，采用 RESTful API 方式提供通用接口。

各存储方案的简单比较如表 3-1 所示。

表 3-1 存储子系统方案比较

Table 3-1 Storage Subsystem Comparison

特性	NVS	DAS	SAN	Cloud Storage
容量级别	MB	TB	PB	PB
访问方式	Block	Block	Block	Object
访问速度	慢	一般	快	快
可靠性	好(只读)	一般	一般	好
可用性	好	好	好	好
可扩充性	不好	一般	好	非常好

### 3.5.1 本地存储方案

#### （1）固件存储区

本文采用 UEFI 规范的 Platform Initialization Specification 定义的 UEFI 的文件系统 Firmware File System 来存储部分模块。

一般固件厂商会遵循 PI 的规范采用多个固件分区(FV)。其中包含了启动代码的 FV 被称作 Boot Firmware Volume (BFV)。其他 FV 可能包含 CPU 的微码补丁及非易失性存储区(NVRAM)等。

本平台将引导模块存放在 FV 中，但为了能够安全的独立更新，故存放在 BFV 以外的其它 FV 当中。

固件存储区可通过 UEFI 提供的接口按照 Firmware File System 访问，或者通过内存映射（Memory Mapping）的方式直接读取原始二进制文件。

### （2）EFI System Partition (ESP)

EFI System Partition 是一块 EFI 的扩充存储区域。这块区域通常以分区的方式部署在硬盘上。

ESP 分区格式为 FAT 分区格式的变体，在 GUID Partition Table 中采用 GUID C12A7328-F81F-11D2-BA4B-00A0C93EC93B 来标识，在传统 MBR 格式的分区方式下，采用 0xEF 来标识。

### （3）块设备存储区

常见的主流块设备包括 ATA/ATAPI 设备，SCSI 设备等 DAS 存储方式。块设备可以用来存储操作系统的可执行文件以及数据文件等。

块设备一般都有专用的控制器，操作系统通过控制器的驱动程序，访问块设备内存储的内容。

## 3.5.2 远程存储方案

远程存储主要由存储区域网络和云存储来实现：

### （1）iSCSI

SAN 是近年兴起的优秀存储方案。SAN 分为 FC-SAN 和 IP-SAN 两类。本平台采用 iSCSI（IP SAN），作为为本地 Block Device 存储的扩充，并可为平台其他模块或用户应用提供简单存储服务。

本平台采用以太网卡加软件模拟的方式实现 iSCSI 的 Initiator 端。

### （2）云存储



云存储主要是为云端计算资源提供镜像和数据存储空间。也可为本项目终端平台模块或用户应用提供服务用于扩充存储空间以及提高数据的可靠性、可用性以及安全性。

目前比较成熟的云存储方案如 S3、Google Cloud Storage 和 SWIFT。

本平台采用 RESTful API 访问方式访问云存储资源。

### 3.6 平台软件设计

本节按照软件工程思想，对平台架构的软件部分进行设计，确定出各层相应的设计。

第一，采用分层的方式，基于抽象程度递增的将一个复杂系统设计按递增的步骤进行分解。遵循实现和接口分离原则，使相邻层提供同样的接口和与之相邻的上层和下层交互。

第二，采用模块化的方式，将特定功能封装成在独立的组件中，形成具有特定输入、输出的程序实体。模块划分时应遵从是高内聚低耦合的准则。

#### 3.6.1 整体构架

根据平台的构架，软件部分经过分层和模块化，采用如图 3-4 所示构架，从上到下依次分成三层，在各层内，再按照功能分成较小的模块。

平台的各模块运行周期分成如图 3-5 所示的三个典型周期。

##### (1) UEFI 阶段

该阶段为 UEFI 的主要运行阶段，在该阶段结束后，其 Runtime Service 将会被平台保留并继续运行，其他 UEFI 服务如 Boot Service 等均停止服务，并回收其所占用的内存。

##### (2) 平台引导阶段

该阶段由平台引导头负责将系统运行环境从 UEFI 下接管，并进入平台服务阶段。

##### (3) 平台服务阶段

该阶段是平台提供服务的主要阶段。该阶段由本地的终端硬件或云端提供计算资源，终端和云端的计算资源可按需各自分别提供服务。

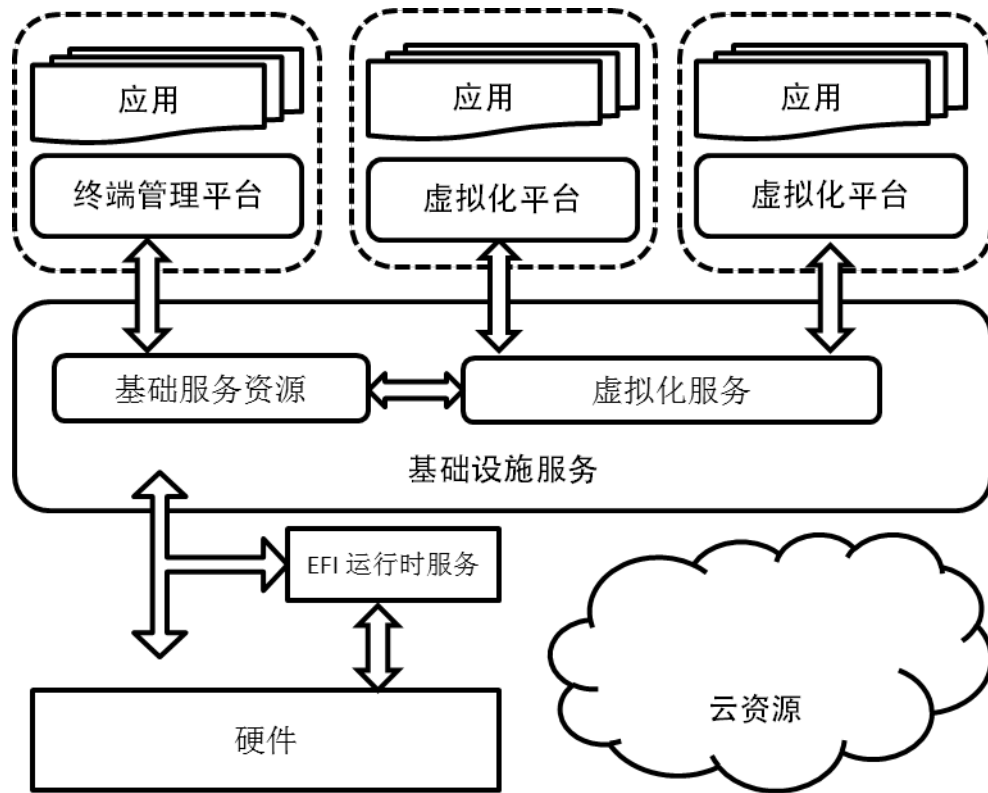


图 3-4 平台软硬件构架

Fig. 3-4 Software and hardware architecture of platform

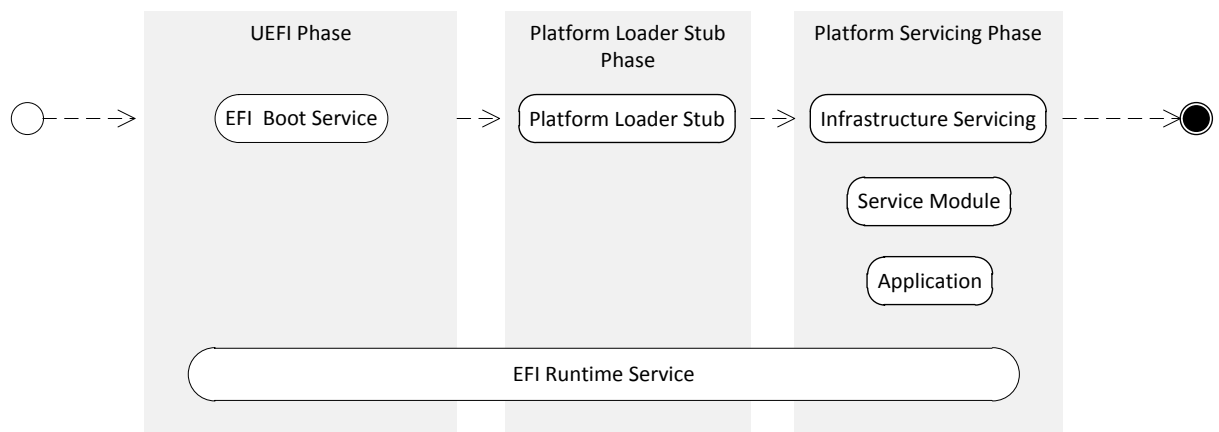


图 3-5 平台软件运行周期生命周期

Fig. 3-5 Life cycle of platform software

### 3.6.2 基础设施层

本平台的基础设施模块基于底层硬件构架进行设计，如图 3-6 所示，通过调用或者连接不同的服务模块，为应用提供虚拟平台支持。

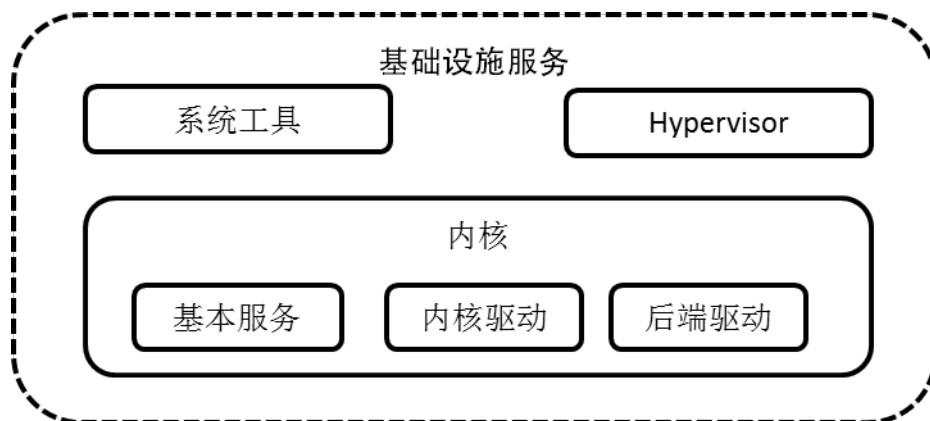


图 3-6 基础设施层软件构架

Fig. 3-6 Software architecture of infrastructure Layer

基础设施层为其上层提供网络、文件系统、内存管理、进程管理、存储子系统等服务和计算资源。

虚拟化平台通过虚拟化服务模块（Hypervisor），间接从其他模块以及内核(KVM)获得基础服务。

终端管理平台直接从基础设施层的各种基础设施资源获取计算资源服务。

### 3.6.3 平台服务层

本层为不同类型的应用提供不同类型的通用服务。

平台服务模块可以是针对终端的底层硬件的管理服务模块，也可以是抽象的本地虚拟机服务模块，还可以是云计算模块。

#### （1）终端管理平台

为了提高终端的可靠性、可用性，本文针对终端本地的应用提供一个如图 3-7 所示构架的终端管理模块。终端管理模块完全基于下层的终端硬件，向上层提供完全的硬件控制能力，可为终端的本地应用提供平台级别的支持，以便能从本地或远程对本地

终端进行系统维护、诊断、调试。由于应用的特殊性，终端管理平台将独占终端的所有物理硬件资源，并具有实际全面的控制能力。

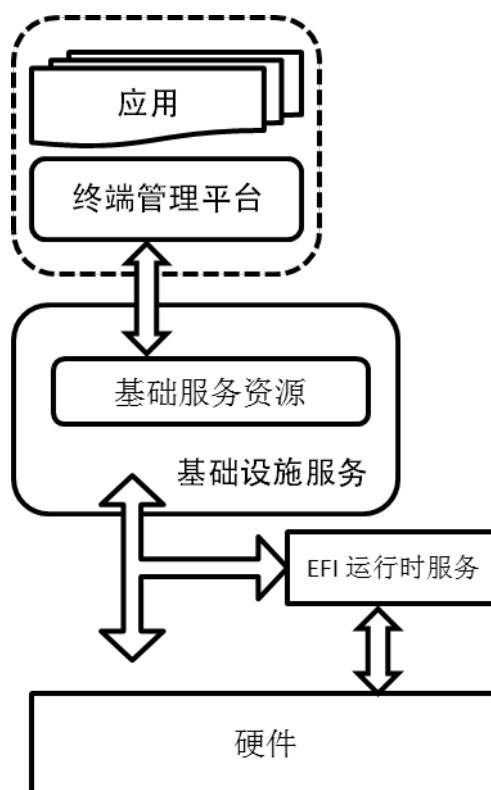


图 3-7 终端管理平台软件构架

Fig. 3-7 Software architecture of endpoint mangement platform

## (2) 虚拟平台

该模块将物理的计算资源抽象化和虚拟化，淡化用户对物理资源的依赖，向上层提供虚拟的硬件，并为应用层提供虚拟的底层的调用和操作系统级别的其他支持，构架如图 3-8 所示。根据应用的不同，虚拟平台层可配置为不同构架的服务，如 ARM 平台的 Linux 操作系统或者 x86 构架的 64bit Windows 操作系统。根据抽象和部署方式，可以进一步分成本地的终端虚拟平台或者云端虚拟平台。

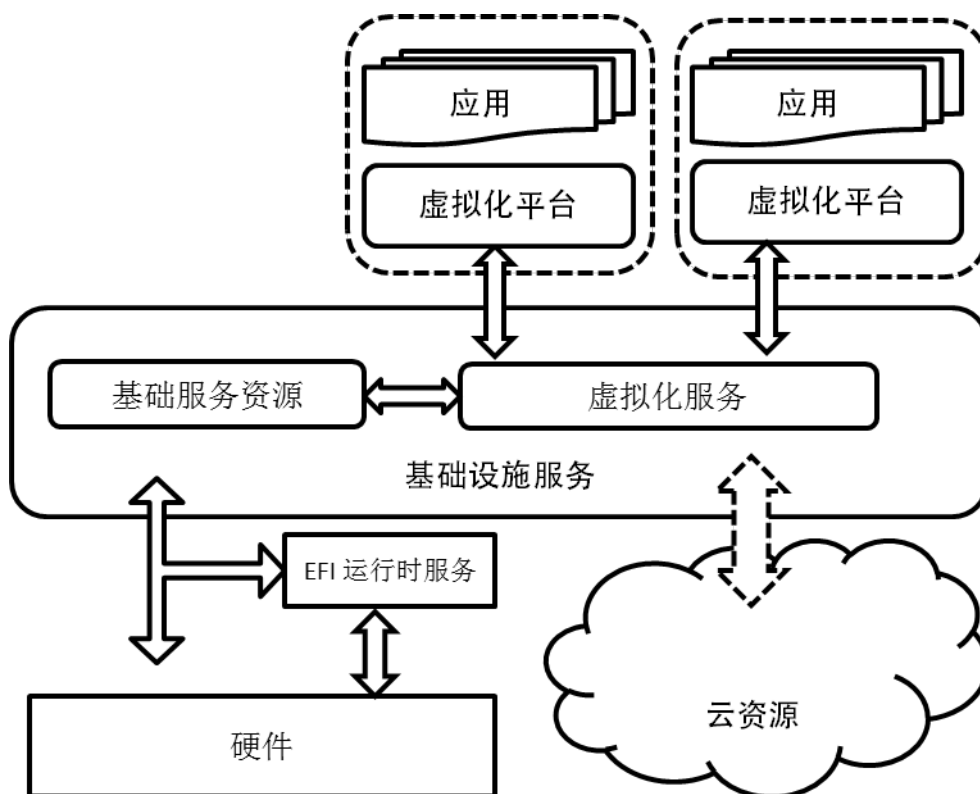


图 3-8 虚拟平台软件构架

Fig. 3-8 Software architecture of Virtual Platform

虚拟化服务主要有 Hypervisor 提供，本文采用包含 KVM 的 Linux Kernel<sup>[17]</sup>作为 Hyperviosr。

采用 KVM 构架的虚拟化技术，如图 3-9 所示。

KVM 内核模块提供一个名为 /dev/kvm 的设备<sup>[18]</sup>，它在非传统的内核模式(Kernel Mode)和用户模式(User Mode)之外，提供了客户模式（Guest Mode）。Guest Mode 拥有自己的 Kernel Mode 和 User Mode。进程通过打开/dev/kvm 设备来创建 VM，然后通过 KVM API 和 IOCTL 来控制 VM 中的设备。设备树 (/dev) 中的设备对于所有用户空间进程来说都是通用的，但是每个打开 /dev/kvm 的进程看到的是不同的 VM 映射。VM 的地址空间独立于内核或正在运行的任何其他 VM 的地址空间，实现了 VM 间的隔离。

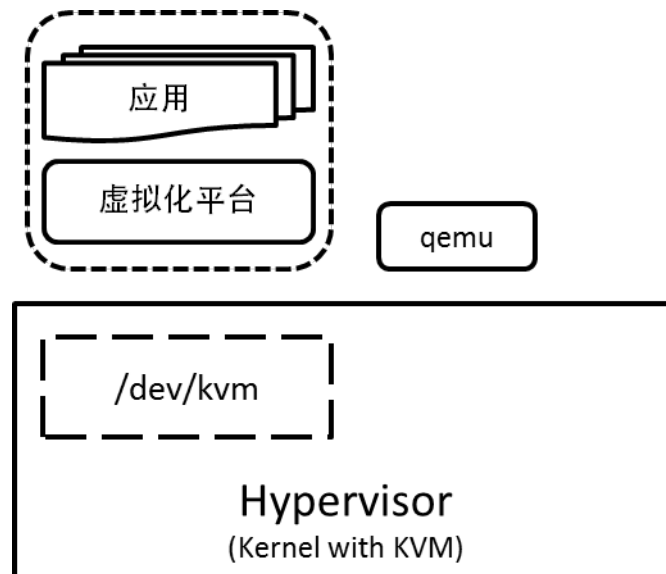


图 3-9 KVM 虚拟化构架

Fig. 3-9 KVM virtualization architecture

KVM 可以采用如下伪代码测试：

```

int main()
{
    void *vm_mem;

    // open /dev/kvm and get handle to VM
    kvm = kvm_init(&test_callbacks, 0);
    if (!kvm) {
        fprintf(stderr, "kvm_init failed\n");
        return 1;
    }

    // use ioctl to create VCPU and memory
    if (kvm_create(kvm, 128 * 1024 * 1024, &vm_mem) < 0) {
        kvm_finalize(kvm);
        fprintf(stderr, "kvm_create failed\n");
        return 1;
    }

    // load image to ram
    if (ac > 1)
        if (strcmp(av[1], "-32") != 0)
            load_file(vm_mem + 0xf0000, av[1]);
  
```

```
else
    enter_32(kvm);
if (ac > 2)
    load_file(vm_mem + 0x100000, av[2]);
kvm_show_regs(kvm, 0);

// launch the VM
kvm_run(kvm, 0);

return 0;
}
```

### 3.6.4 引导头

引导头用于将系统的控制权从 UEFI 转移到本平台，同时引导头还将负责平台的初始运行环境设置，以便基础设施层能够接下来继续完成初始化。

引导头的基本功能包括：

- 获取 UEFI 变量，确定引导参数。
- 运行环境分析，确定当前系统构架，内存分配表，启动设备位置。
- 根据平台内核要求，设置正确的启动环境和参数。
- 引导基础设施层：如果引导失败，需返回错误信息；如果成功，则将控制权交给基础设施层。

其运行流程如图 3-10 所示。

引导头通过调用 UEFI 系统服务来进行系统检测检测，可以使用的关键服务定义如下：

- BootService→ GetMemoryMap(), 获取系统内存分配信息。
- BootService→ AllocatePages(), 向 UEFI 申请页面
- BootService→ FreePages(), 释放从 UEFI 申请得到的页面
- BootService → ExitBootServices(), 结束 Boot Service
- RuntimeService → GetVariable(), 获取 UEFI 变量
- RuntimeService → SetVariable(), 设置 UEFI 变量

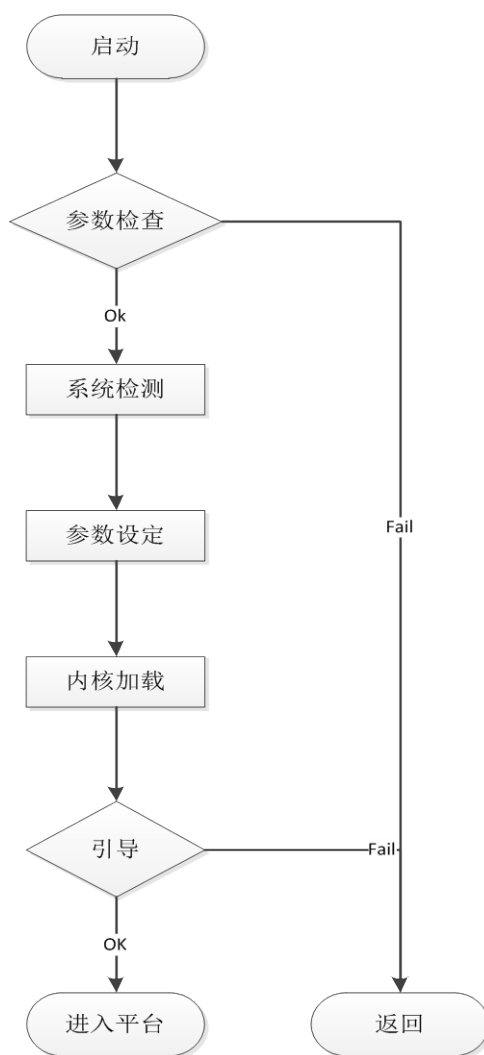


图 3-10 平台引导流程图

Fig. 3-10 Platform loading flowchart

同时，采用定义如下的 UEFI Protocol 来获取系统相关信息：

- EFI\_GRAPHICS\_OUTPUT\_PROTOCOL (GUID: {0x9042a9de,0x23dc,0x4a38,0x96,0xfb,0x7a,0xde,0xd0,0x80,0x51,0x6a}), 提供显示设备支持。
- EFI\_SMBIOS\_PROTOCOL (GUID: {0x5e90a50d, 0x6955, 0x4a49, 0x90, 0x32, 0xda, 0x38, 0x12, 0xf8, 0xe8, 0xe5}), 通过 SMBIOS 提供系统信息。

引导头还需要从基础服务层的内核头中确定重要的信息，其中的关键信息如表 3-2 所示。



表 3-2 内核头部关键信息

Table 3-2 Key Information in Kernel Header

Offset/Size	Name	Meaning
01F1/1	setup_sects	The size of the setup in sectors
01F2/2	root_flags	If set, the root is mounted readonly
01F4/4	syssize	The size of the 32-bit code in 16-byte paras
01FA/2	vid_mode	Video mode control
01FC/2	root_dev	Default root device number
01FE/2	boot_flag	0xAA55 magic number
0200/2	jump	Jump instruction
0202/4	header	Magic signature "HdrS"
0206/2	version	Boot protocol version supported
0208/4	realmode_swth	Boot loader hook (see below)
020E/2	kernel_version	Pointer to kernel version string
0210/1	type_of_loader	Boot loader identifier
0211/1	loadflags	Boot protocol option flags
0212/2	setup_move_size	Move to high memory size (used with hooks)
0214/4	code32_start	Boot loader hook (see below)
0218/4	ramdisk_image	initrd load address (set by boot loader)
021C/4	ramdisk_size	initrd size (set by boot loader)
0224/2	heap_end_ptr	Free memory after setup end
0226/1	ext_loader_ver	Extended boot loader version
0227/1	ext_loader_type	Extended boot loader ID
0228/4	cmd_line_ptr	32-bit pointer to the kernel command line
022C/4	ramdisk_max	Highest legal initrd address
0230/4	kernel_alignment	Physical addr alignment required for kernel
0234/1	relocatable_kernel	Whether kernel is relocatable or not
0235/1	min_alignment	Minimum alignment, as a power of two
0238/4	cmdline_size	Maximum size of the kernel command line

表 3-3 关键引导参数

Table 3-3 Key Boot Parameter

Offset/Size	Name	Meaning
000/040	screen_info	Text mode or frame buffer information
140/080	edid_info	Video mode setup
1C0/020	efi_info	EFI 32 information
1E8/001	e820_entries	Number of entries in e820_map
2D0/A00	e820_map	E820 memory map table

引导头在最终引导为基础服务层内核之前，需要为其设定的关键引导参数如表 3-3 所示。

引导头通过编译的方式，整合基础设施层的内核镜像文件及初始化内存盘（INITRD）文件，采用如图 3-11 所示结构，构成单一的 UEFI 可执行文件。

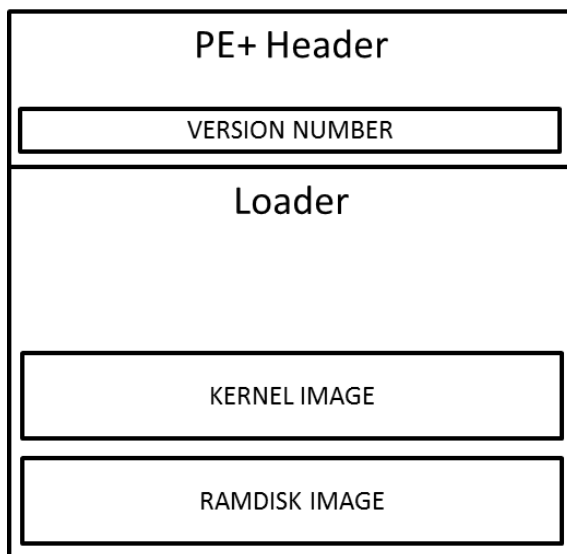


图 3-11 引导头镜像文件布局

Fig. 3-11 Bootloader stub file layout

VERSION NUMBER 用于存放版本信息，如表 3-4 所示，采用 PE+文件头中可选文件头中的版本定义以保持兼容性。

表 3-4 平台版本控制

Table 3-4 Platform version control

Offset	Size	Field	Description
44	2	MajorImageVersion	镜像文件主版本号
46	2	MinorImageVersion	镜像文件次版本号

### 3.7 本章小结

本章首先对平台的需求进行了深入的分析与研究，找出了本平台在设计和实施过程中需要考虑的各个方面。对平台的功能、性能需求进行深入调研与分析，并对用例图进行建模，撰写出用例规约。

然后根据通用平台的需求，采用虚拟化、云存储和云计算技术，充分运用模块化和抽象化的手段对平台进行了整体设计，并结合软件工程思想和面向服务技术，对平台软硬件的架构进行了分析与设计，为下一章对平台的具体实施进打造了坚实的技术基础和指导。

本章采用前一章中的设计，对设计进行实现。

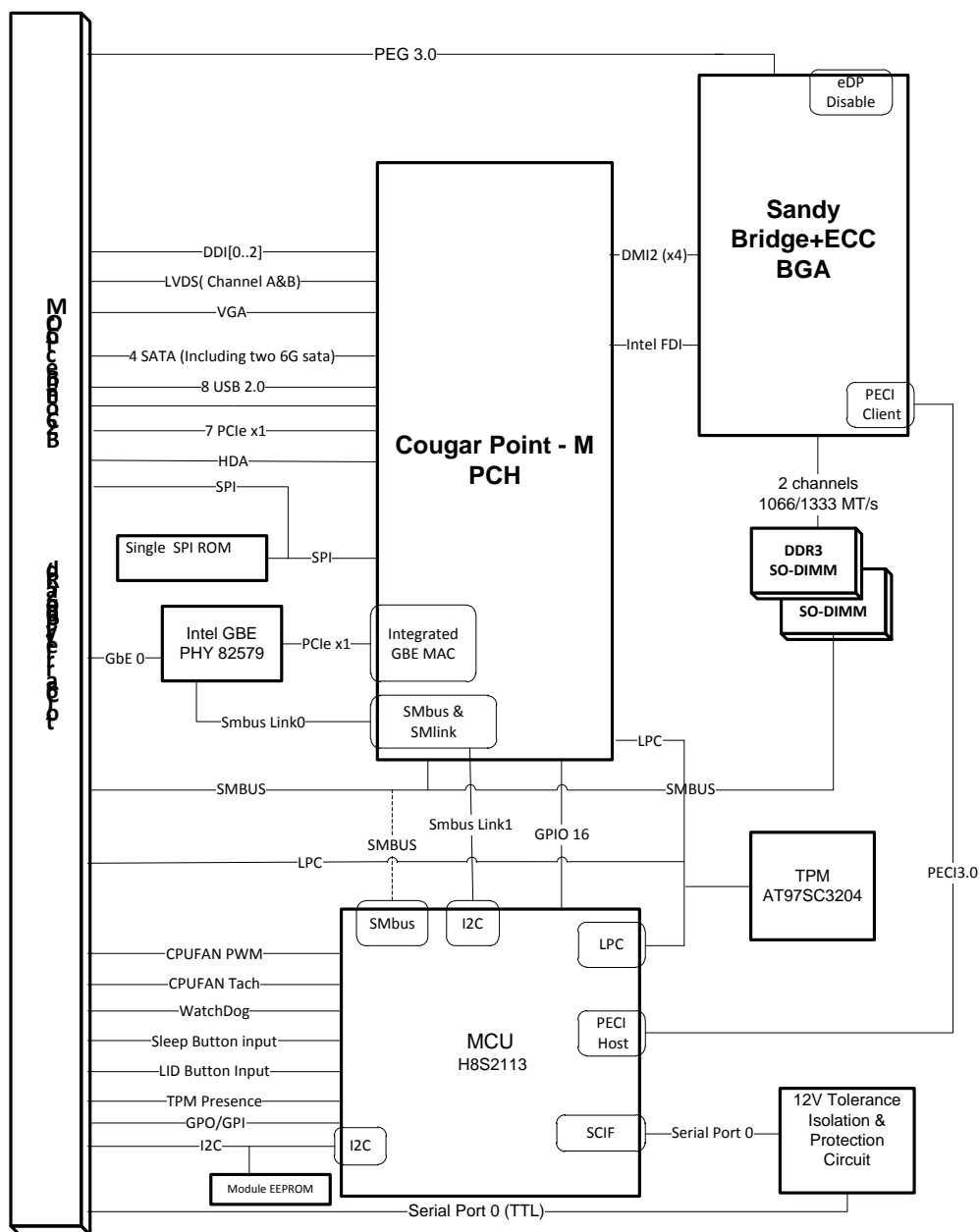


图 4-1 系统框架图

Fig. 4-1 Block diagram of platform hardware

本平台的开发、实现及验证采用 COM-Express 2.0<sup>[19]</sup>模块。

COM-Express 规范由 PICMG 联合众多厂商共同制定。该规范采用 Computer-On-Module 的 form factor，采用高度集成的模块化设计，使得系统的更新、升级和维护非常简便，是本文的硬件平台的理想选择。

以 Intel X86 构架为例，可采用如图 4-1 所示构架实现 COM-E 规范的模块。

## 4.2 存储子系统

本基础设施层的存储系统(Storage Subsystem)主要包括：非易失性存储（NVS）、本地直接存储（DAS）、存储区域网络（SAN）以及云存储(Cloud Storage)。

- 非易失性存储（NV Storage）：NVS 由 16M 的 SPI 接口的 Flash 芯片提供。SPI Flash 支持片上执行，但是速度仍然比较慢，因此在运行时，UEFI 会将其内容拷贝到内存中执行。它可通过 UEFI 提供的接口按照 Firmware File System 访问，或者通过内存映射的方式直接读取原始二进制文件。
- 本地直接存储（DAS）：DAS 由传统的 SATA 接口实现。目前主流的 SATA 控制器均支持 AHCI 模式，可以直接采用 Linux 内核的 AHCI 控制器的驱动以更好地利用硬件并获得更好的 IO 性能。
- 网络存储：常用的网络存储包括简单的文件共享、NAS、SAN 等。通常通过 Linux 内核、Linux 驱动模块或者系统服务即可实现。
- 云存储（Cloud Storage）：云存储采用 Openstack 的 SWIFT 开源项目实现。SWIFT 可很方便的通过兼容 Amazon S3 的 REST API 接口进行访问，可用于搭建私用云，或者共有云方案，如 RackSpace 的 Cloud Files。

基础设施层的内核模块(bzImage)和初始化模块(initrd)，与引导模块将其以二进制文件的方式，打包在引导模块的 UEFI 镜像文件内。

基础设施层的的其他部分，可用只读文件系统 Squashfs 封装在一个.squashfs 镜像文件中，可根据提供的基础服务的多少以及最终文件镜像的大小选择如图 4-2 所述的部署方式。

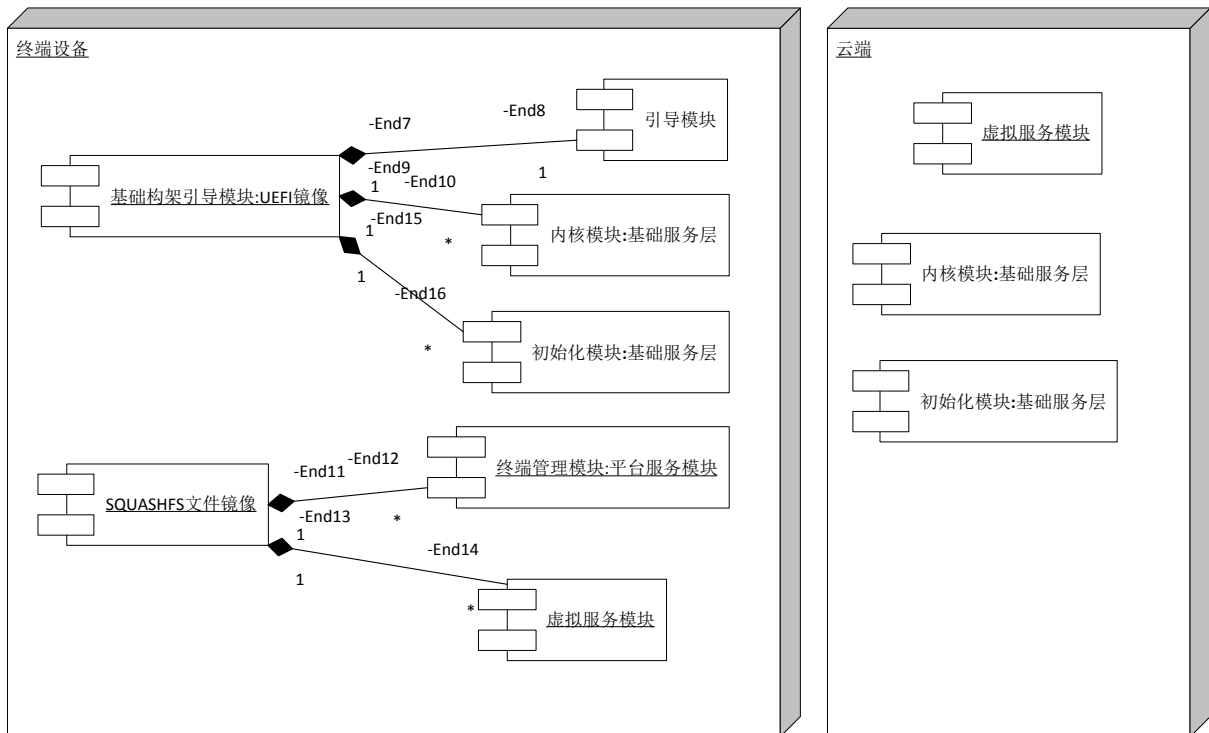


图 4-2 系统部署框架图

Fig. 4-2 System deployment diagram

### 4.3 基础设施层实现

基础设施层主要包含两个模块内核模块(bzImage)和存放在 RAMDISK(initrd)模块。基础设施层按照部署位置分成部署在本地的终端基础设施和部署在云端的云端基础设施。

#### 4.3.1 内核模块

内核模块基于 2.6 版本以上的 Linux 内核定制，采用 x86\_64 构架。

如图 4-3 所示，内核主要由内核子系统和支持软件两大部分组成。

(1) 内核子系统：提供进程管理、内存管理、文件系统、设备管理、网络等操作系统级别的支持。

(2) 支持软件：除了包含 x86\_64 构架默认的必需模块外，还包含 KVM（及 VirtIO 框架），网络子系统，硬件设备驱动，虚拟设备驱动等支持。

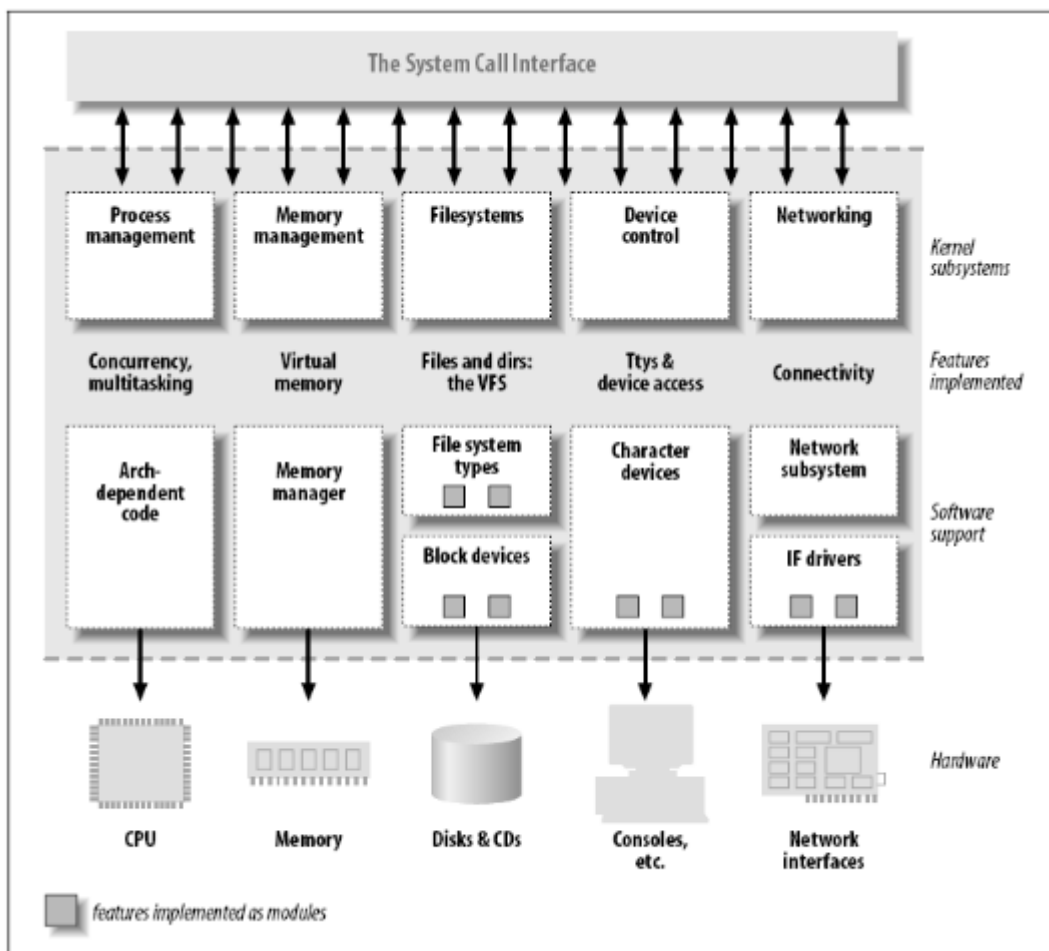


图 4-3 Linux 内核构架框图

Fig. 4-3 Block diagram of Linux kernel

定制时，参考 x86\_64\_defconfig 文件，或者采用 make 生成标准 x86\_64 的配置文件：

```
make x86_64_defconfig
```

然后编辑源代码根目录下的.config 文件，自定义本平台所需的内核模块。确保其中比较重要的配置定义如下：

```
#添加基本设置
CONFIG_BLK_DEV_INITRD=y
CONFIG_KERNEL_BZIP2=y
```

```
CONFIG_RELOCATABLE=y
CONFIG_MCORE2=y
CONFIG_IA32_EMULATION=y
#添加 EFI 支持
CONFIG_EFI=y
CONFIG_EFI_VARS=y
CONFIG_EFI_PARTITION=y
CONFIG_FB_EFI=y
CONFIG_FRAMEBUFFER_CONSOLE=y
#添加网络及 iSCSI 相关支持
CONFIG_INET=y
CONFIG_E1000=y
CONFIG_ISCSI_TCP=y
CONFIG_SCSI_ISCSI_ATTRS=y
CONFIG_ISCSI_BOOT_SYSFS=y
#添加虚拟化相关支持
CONFIG_VIRTUALIZATION=y
CONFIG_KVM=y
CONFIG_VIRTIO=y
CONFIG_VIRTIO_RING=y
CONFIG_VIRT_DRIVERS=y
CONFIG_VIRTIO_BLK=y
CONFIG_VIRTIO_BALLOON=y
CONFIG_VIRTIO_MMIO=y
CONFIG_VIRTIO_PCI=y
CONFIG_VIRTIO_NET=y
CONFIG_VHOST_NET=y
```

在内核编译完成后，将得到平台的内核镜像文件 **bzImage**。该文件将为整个基础设施层提供操作系统的内核服务。



### 4.3.2 初始化模块

内核被引导模块加载到特定位置，并开始运行。内核首先做完整性检查，然后将调用自解压模块解压到内存中继续执行。解压以后的内核首先进行各硬件的初始化，然后按照 FHS 规范<sup>[20]</sup>创建根文件系统(rootfs)，最后调用内核启动 init 程序，成为第一个进程。

init 程序以 ext2 文件系统格式存放在 initrd 文件中，它会创建用于启动的临时文件系统，检查/proc/cmdline 的输入参数，挂载服务模块的真实根文件系统，并最终将最终的根文件系统切换到基础服务模块中的文件系统上。

Init 的可以可执行脚本文件<sup>[21]</sup>的方式示范如下：

```
#!/bin/sh

# Create necessary file and directory
create_sysdir

# Mount necessary system filesystem
mount_sysfs

# Parse command line options and set boot variable
for x in $(cat /proc/cmdline); do
    parse_cmdline x
done

# Change to real filesystem
change_fs
```

### 4.3.3 基础服务模块

初始化完成后，平台的基础服务模块会被加载，以便进入可服务状态。根据部署位置可分为终端和云端基础服务：

#### （1）终端基础服务模块

包含如表 4-1 所示的初始化的工具和管理上层模块的工具集，以及相应硬件的驱动程序模块：

表 4-1 基础设施层工具一览

Table 4-1 Overview of Utilities of Infrastructure Layer

工具名称	功能
初始化工具集	包含 busybox, mkfs, mount, modprobe 等系统工具，用于初始化及加载上层模块
基本管理工具集	提供管理工具以及设备驱动模块，如 vncviewer, ssh, telnet 等工具
虚拟机管理工具集	提供基本的虚拟机管理（Hypervisor），如：qemu-kvm <sup>[22]</sup> , virsh, virtio 驱动（包括 virt block 等），pthon-nova client 等等

## (2) 云端基础服务模块

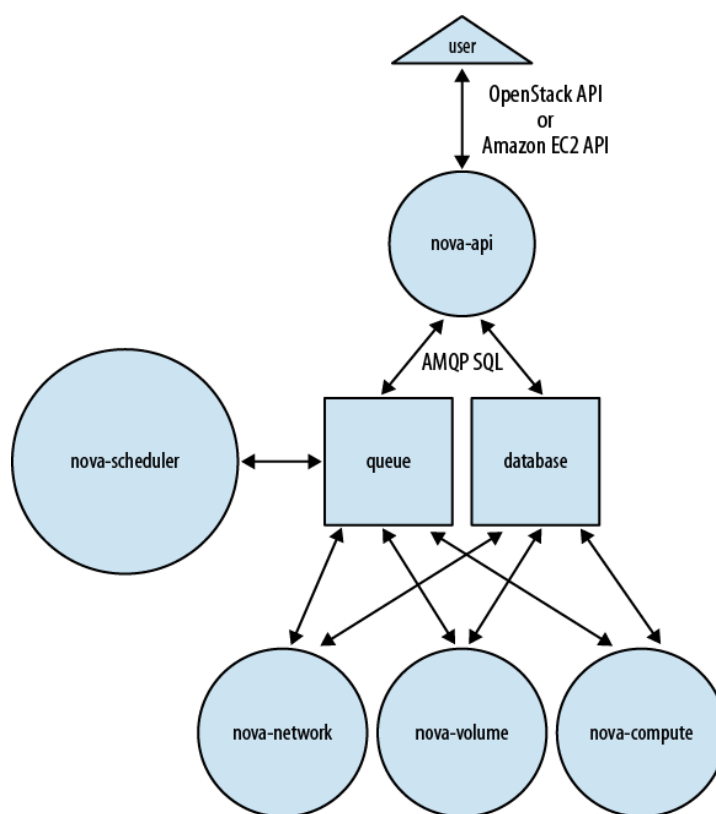


图 4-4 OpenStack Nova 逻辑构架<sup>[23]</sup>

Fig. 4-4 OpenStack Nova logical architecture<sup>[23]</sup>

云端的基础服务由 OpenStack 的 nova, glance, swift 项目实现。OpenStack 的云解决方案的各项目和模块之间的关系如图 4-4 所示。

本文采用的 Openstack 项目和工具如表 4-2 所示。

表 4-2 云服务管理工具一览

Table 4-2 Overview of Utilities for Cloud Service

工具名称	功能	部署位置
OpenStack glance	提供虚拟机镜像查找及检索系统	云端
OpenStack nova	提供 OpenStack Compute 服务, 用于为单个用户或使用群组启动虚拟机实例	云端
OpenStack swift	采用 OpenStack Object Storage 实现的大规模可扩展存储系统, 内置冗余及容错机制	云端
eucalyptus 工具集	提供多种云服务客户端管理工具, 可兼容多种云服务, 提供多种虚拟机镜像格式转换功能	终端
python-nova	提供 Compute API <sup>[24]</sup> 接口的 nova shell 命令, 可以命令行的方式向云端发送命令	终端

在产品化时, 按照官方要求, 各节点的服务器的最低硬件配置要求如表 4-3 所示。

表 4-3 服务节点推荐配置

Table 4-3 Hardware Recommendations

服务器	最低配置
<b>控制器节点</b> <b>Cloud Controller node</b> (运行 network, volume, API, scheduler and image 服务)	Processor: 64-bit x86 Memory: 12 GB RAM Disk space: 30 GB (SATA or SAS or SSD) Volume storage: two disks with 2 TB (SATA) for volumes attached to the compute nodes Network: one 1 GB Network Interface Card (NIC)
<b>计算节点</b> <b>Compute nodes</b> (运行虚拟机实例)	Processor: 64-bit x86 Memory: 32 GB RAM Disk space: 30 GB (SATA) Network: two 1 GB NICs

如为验证所需，可采用 OpenStack 的官方 all-in-one 解决方案，将所有节点（Controller/Network/Volume/Compute Node）和 Horizon Dashboard 都搭建在同一台服务器上，作为单一节点(Single Node)。

节点上安装的组件包括：

- nova-api
- nova-scheduler
- nova-objectstore
- nova-manage
- nova-vncproxy
- glance
- keystone
- horizon
- nova-network
- nova-volume
- nova-compute
- KVM/QEMU hypervisor
- libvirt library
- 其他 Nova 相关组件

#### 4.4 平台服务层实现

初始化完成后，内核参照将挂载不同的平台服务层的模块为用户提供服务。

平台服务层采用模块的方式，针对应用层的应用提供相应服务。根据应用的不同，可分为终端管理模块和虚拟服务模块。

##### 4.4.1 终端管理模块

终端管理模块采用完整的 Linux 文件系统整合后打包而成。服务模块用只读文件系统 Squashfs 封装成.squashfs 镜像文件。根据管理软件的多少以及封装后文件的大小，可部署在本地的非易失性存储介质(FLASH ROM)内或者存储子系统的其他介质(如本地 SATA 硬盘、USB Flash Disk、NFS 或者 iSCSI Target)上。

终端管理模块包含完整的终端硬件驱动支持，通过驱动和基础设施层的系统调用接口和系统交互。

可向平台镜像添加驱动模块增加新增硬件的支持。

#### 4.4.2 虚拟服务模块

虚拟服务模块采用基于 KVM 构架的虚拟机实现。每个虚拟服务模块包含一个或多个虚拟机的磁盘镜像和配置。

##### (1) 本地部署方式

部署在本地时，采用如表 4-4 所示的基础设施层内置工具进行管理。

表 4-4 虚拟服务模块管理工具一览

Table 4-4 Overview of Utilities for Virtual Service Module

工具名	功能描述	UI 类型
<b>UCS / UVMM</b>	Univention Virtual Machine Manager is a high-performance management system for KVM and XEN.	WEB
<b>Plain qemu/kvm</b>	You can run qemu/kvm straight from the command line	CLI
<b>virsh</b>	A minimal shell around libvirt for managing VMs	CLI
<b>VMM / Virtual Machine Manager</b>	Also known as virt-manager. A desktop user interface for managing virtual machines.	Desktop
<b>virt-install/clone/convert</b>	Command line tools for provisioning new VMs, cloning existing VMs and importing / converting appliance images.	CLI
<b>Red Hat Enterprise Virtualization</b>	Commercial management solution for RHEL / KVM.	Web
<b>Witsbits</b>	Free KVM Virtualization Management delivered as a cloud service.	Web REST API
<b>AQemu</b>	a Qt4 user interface for KVM	Desktop
<b>Proxmox VE</b>	Open Source virtualization platform for running Virtual Appliances and Virtual Machines	WEB CLI
<b>ConVirt</b>	ConVirt 2.0 Open Source is the leading open source product for managing Xen and KVM, enabling you to standardize and proactively manage your virtualized environment in a centralized fashion.	WEB
<b>oVirt</b>	oVirt is a virtualization management framework consisting of a small host image, the oVirt Node, that provides the libvirt service to host virtual machines, and a robust vm management software stack, controlled by a web-based management interface, the oVirt Server.	WEB
<b>OpenNebula</b>	an open source virtual infrastructure engine	CLI xml-rpc

续表 4—4

工具名	功能描述	UI 类型
<b>OpenNode</b>	RHEL/CentOS based open-source server virtualization and management solution - simple bare-metal installer, providing KVM+OpenVZ host and standard libvirt, func management interfaces together with standard cli tools like virsh and vzctl. OpenNode Management Server with ajax web-based management console available - as is RPC-JSON API interface.	WEB CLI API
<b>Ganeti</b>	Ganeti is a cluster virtual server management software tool built on top of existing virtualization technologies	CLI
<b>Karesansui</b>	Karesansui is an open-source virtualization management application. It's smart graphical user interface lowers your management cost, and brings a total management/audit solution for both physical and virtual servers.	WEB REST API
<b>openQRM</b>	openQRM is the next generation, open-source Data-center management platform.	WEB
<b>Abiquo</b>	Abiquo is an open source infrastructure software for the creation and integral management of Public & Private Clouds based on heterogeneous environments.	WEB
<b>CloudStack</b>	Cloudstack is an open source project that enables the deployment, management, and configuration of multi-tier and multi-tenant infrastructure cloud services using Xen, KVM and VMware hypervisors.	WEB
<b>Nimbula Director</b>	Nimbula Director is a Cloud Operating System that enables Infrastructure as a Service using the KVM.	WEB CLI REST API
<b>Archipel</b>	Archipel is an Open Source project that aims to bring push notifications to virtualization orchestration using XMPP.	WEB
<b>kvm-wrapper</b>	kvm-wrapper is a lightweight, simple and intended to be hackable set of shell scripts that help manage kvm virtual machines a great deal.	CLI
<b>kvm-admin</b>	Python scripts for managing the guests (boot, shutdown ...) and include a commandline monitor .	CLI
<b>Virtualbricks</b>	Python-gtk GUI to manage guest and hybrid (host/guest) networks.	CLI
<b>SolusVM</b>	The most popular control panel for commercial use.	WEB
<b>Stackops Openstack Distro</b>	Stackops is an Openstack Nova distribution verified and tested for KVM. You only need to download the ISO image with the distro and install it on one or more servers.	CLI REST API
<b>WebVirtMgr</b>	Web service for managing VMs based on the KVM	WEB

用于封装 kvm 虚拟机所用的文件包括：

- .img 文件：磁盘镜像文件
- .xml 文件：虚拟机配置文件

- bzImage 文件：内核的文件镜像（包含 VirtIo 支持）
- initrd 文件：初始化内存文件系统文件

以 qemu-kvm 为例，可用如下参数加载虚拟机：

```
kvm -boot c -drive file=images.qcow2,if=virtio -m 1024 -netdev  
type=tap,script=/etc/kvm/qemu-ifup,id=net0 -device virtio-net-pci,netdev=net0 -nographic -  
vnc :0
```

## （2）云端部署方式

部署在云端时，采用 OpenStack Glance 进行镜像的查找和管理，采用 OpenStack Nova 提供虚拟机运行实例。

部署完成后，可用客户端工具 python-nova 进行管理，如：

```
$nova list
```

本文在验证阶段，采用 ssh client 的方式访问云端的虚拟服务模块（即虚拟机实例）。

为了保持虚拟服务模块在终端和云端的可迁移性，对服务模块的镜像模板做如下要求：

- 虚拟机的磁盘基于镜像文件，采用 ext4 文件系统
- 包含 SSH Server，VNC Server 服务

## （3）虚拟服务模块迁移

如果有必要，虚拟服务模块借助对应的工具，可以在终端和云端之间静态迁移（Offline Migration）。常用的迁移工具如表 4-5 所示。

表 4-5 虚拟机迁移工具示例

Table 4-5 Example of VM Migration Tool

工具名称	功能
qemu-img	虚拟机磁盘镜像文件处理工具，可支持多种虚拟机磁盘镜像格式的创建，转化。
qemu-kvm	虚拟机管理工具，用于管理虚拟机的配置、运行实例、迁移等。
euca2ools	向云端提交镜像模板的工具集，包含多个工具，支持多种云计算方案
libvirt-bin	包含多种虚拟机管理工具如 virsh, libvirt-migrate-qemu-disks 等。
glance-client	Openstack glance 项目提供的命令行工具。

## 4.5 引导头实现

引导头的主要任务是在 UEFI 环境下，检测系统环境，为平台基础设施层提供必要信息和设置，并从 UEFI 接过系统控制权转交给基础设施层的内核。

### 4.5.1 引导环境

依照设计构架，平台从 UEFI 固件内直接引导，因此平台的引导头部分将从 UEFI 运行环境内，以为一个标准的 UEFI Image 来运行。

引导头以普通 UEFI 应用程序的方式调用并运行，此时可以使用 UEFI 系统的所有 Boot Service，并可以 UEFI 提供的设备驱动访问底层硬件。

在引导阶段的最后，通过调用 Boot Service 的 ExitBootServices()，使 UEFI 系统停止所有的 Boot Service 和 Driver Protocol，同时保留 Runtime Service，此时所有 Boot Server 和驱动均不再可用。

### 4.5.2 引导流程

引导头按照设计要求，设计成 UEFI 环境下的 OS Loader 类型的应用，支持多种平台构架的硬件层。



### (1) 加载 (Load)

引导头和普通 UEFI Image 文件一样, 通过 Boot Service 的 LoadImage()协议加载到内存并重定位, 然后同样通过 Boot Service 的 StartImage()协议调用。

在被调用时, 系统会传入 ImageHandle 和 SystemTable 这两个参数。ImageHandle 是 UEFI 系统分配给的句柄(Handle), SystemTable 是指向 UEFI 系统的 System Table 的指针。

通过这两个参数, 引导头便可以调用 UEFI 系统的 Boot Service, 及 UEFI 里其他 Handle 和 Protocol。

引导头被调用时入口函数伪代码如下:

```
// main entry
EFI_STATUS
efi_main (EFI_HANDLE image, EFI_SYSTEM_TABLE *system_tab)
{
    EFI_STATUS efi_status = EFI_LOAD_ERROR;
    InitializeLib(image_handle, systab);
    return efi_status;
}
```

### (2) 参数检测(Argument Parser)

引导头在加载完成后, 将检查启动参数和 EFI 变量(EFI Variable), 检查基础服务层内核模块的校验和。

### (3) 配置(Setup)

引导头需要根据内核头部信息, 获取系统构架、内存分配等信息, 并需要根据需要启动的内核的构架, 构建正确的参数。

内核头采用 C 语言的 Stuct 定义<sup>[25]</sup>如下:

```
#pragma pack(1)
typedef struct _LINUX_KERNEL_HEADER
{
    UINT8 code1[0x0020];
    UINT16 cl_magic;          /* Magic number 0xA33F */
}
```

```

UINT16 cl_offset;          /* The offset of command line */
UINT8 code2[0x01F1 - 0x0020 - 2 - 2];
UINT8 setup_sects;        /* The size of the setup in sectors */
UINT16 root_flags;        /* If the root is mounted readonly */
UINT16 syssize;           /* obsolete */
UINT16 swap_dev;          /* obsolete */
UINT16 ram_size;          /* obsolete */
UINT16 vid_mode;          /* Video mode control */
UINT16 root_dev;          /* Default root device number */
UINT16 boot_flag;         /* 0xAA55 magic number */
UINT16 jump;              /* Jump instruction */
UINT32 header;             /* Magic signature "HdrS" */
UINT16 version;           /* Boot protocol version supported */
UINT32 realmode_swth;     /* Boot loader hook */
UINT16 start_sys;         /* The load-low segment (obsolete) */
UINT16 kernel_version;    /* Points to kernel version string */
UINT8 type_of_loader;      /* Boot loader identifier */
UINT8 loadflags;          /* Boot protocol option flags */
UINT16 setup_move_size;   /* Move to high memory size */
UINT32 code32_start;      /* Boot loader hook */
UINT32 ramdisk_image;     /* initrd load address */
UINT32 ramdisk_size;      /* initrd size */
UINT32 bootsect_kludge;   /* obsolete */
UINT16 heap_end_ptr;      /* Free memory after setup end */
UINT16 pad1;              /* Unused */
char *cmd_line_ptr;       /* Points to the kernel command line */
UINT32 initrd_addr_max;   /* Highest address for initrd */
} LINUX_KERNEL_HEADER,*PLINUX_KERNEL_HEADER;

```

不同于传统 BIOS 下来的 OS Loader，引导头采用 BootService->GetMemoryMap() 获取系统内存分配情况，用于提供 E820 内存分配表。

伪代码实现如下：

```

EFI_STATUS
get_e820_mmap(PMMAP_DESCRIPTOR desc)
{
    EFI_STATUS    efi_status = EFI_SUCCESS;

    desc->desc_size = DEFAULT_MEMMAP_SIZE;

    do{
        desc->md = (EFI_MEMORY_DESCRIPTOR*)allocatepool(EfiLoaderData,desc-
>desc_size);
        if(NULL == desc->md)

```

```

    {
        DBG_PRT((L"[ GetMemoryMap failed (%r) ]",efi_status));
        break;
    }

    efi_status = uefi_call_wrapper(BS->GetMemoryMap, 5, &desc->map_size, desc-
>md, &desc->cookie, &desc->desc_size, &desc->desc_version);

    if (EFI_SUCCESS == efi_status)
    {
        break;
    }
    else
    {
        freepool(desc->md);
        if(EFI_BUFFER_TOO_SMALL == efi_status)
        {
            desc->desc_size += DEFAULT_MEMMAP_INC;
        }
        else
        {
            DBG_PRT((L"[ GetMemoryMap failed (%r) ]",efi_status));
            break;
        }
    }

}while(1);

return efi_status;
}

```

#### (4) 引导(Boot)

完成所有前期工作以后，引导模块根据内核的文件头部信息，将把内核镜像拷贝到内存中的指定位置，调用 Boot Service 的 ExitBootServices()，然后采用 Zero Page 配置并切换处理器到新的保护模式环境下，并将控制权交给内核完成引导工作。

引导内核采用的参数定义如下：

```

#pragma pack(1)
typedef struct _BOOT_PARAMETERS {
    UINT8 video_cursor_x;    /* 0 */

```

```
UINT8 video_cursor_y;
UINT16 ext_mem;    /* 2 */
UINT16 video_page; /* 4 */
UINT8 video_mode;  /* 6 */
UINT8 video_width; /* 7 */
UINT8 padding1[0xa - 0x8];
UINT16 video_ega_bx; /* a */
UINT8 padding2[0xe - 0xc];
UINT8 video_height; /* e */
UINT8 have_vga;     /* f */
UINT16 font_size;   /* 10 */
UINT16 lfb_width;   /* 12 */
UINT16 lfb_height;  /* 14 */
UINT16 lfb_depth;   /* 16 */
UINT32 lfb_base;    /* 18 */
UINT32 lfb_size;    /* 1c */
UINT16 cl_magic;    /* 20 */
UINT16 cl_offset;
UINT16 lfb_line_len; /* 24 */
UINT8 red_mask_size; /* 26 */
UINT8 red_field_pos;
UINT8 green_mask_size;
UINT8 green_field_pos;
UINT8 blue_mask_size;
UINT8 blue_field_pos;
UINT8 reserved_mask_size;
UINT8 reserved_field_pos;
UINT16 vesapm_segment; /* 2e */
UINT16 vesapm_offset;  /* 30 */
UINT16 lfb_pages;      /* 32 */
UINT16 vesa_attrib;    /* 34 */
UINT32 capabilities;   /* 36 */
```

```
UINT8 padding3[0x40 - 0x3a];
UINT16 apm_version;    /* 40 */
UINT16 apm_code_segment; /* 42 */
UINT32 apm_entry;      /* 44 */
UINT16 apm_16bit_code_segment; /* 48 */
UINT16 apm_data_segment; /* 4a */
UINT16 apm_flags;      /* 4c */
UINT32 apm_code_len;    /* 4e */
UINT16 apm_data_len;    /* 52 */
UINT8 padding4[0x60 - 0x54];
UINT32 ist_signature;    /* 60 */
UINT32 ist_command;      /* 64 */
UINT32 ist_event;        /* 68 */
UINT32 ist_perf_level;   /* 6c */
UINT8 padding5[0x80 - 0x70];
UINT8 hd0_drive_info[0x10]; /* 80 */
UINT8 hd1_drive_info[0x10]; /* 90 */
UINT16 rom_config_len;    /* a0 */
UINT8 padding6[0xb0 - 0xa2];
UINT32 ofw_signature;     /* b0 */
UINT32 ofw_num_items;     /* b4 */
UINT32 ofw_cif_handler;   /* b8 */
UINT32 ofw_idt;           /* bc */
UINT8 padding7[0x1b8 - 0xc0];
union
{
    struct
    {
        UINT32 efi_system_table; /* 1b8 */
        UINT32 padding7_1; /* 1bc */
        UINT32 efi_signature; /* 1c0 */
        UINT32 efi_mem_desc_size; /* 1c4 */
```

```
        UINT32 efi_mem_desc_version; /* 1c8 */
        UINT32 efi_mmap_size; /* 1cc */
        UINT32 efi_mmap; /* 1d0 */
    } v0204;
    struct
    {
        UINT32 padding7_1; /* 1b8 */
        UINT32 padding7_2; /* 1bc */
        UINT32 efi_signature; /* 1c0 */
        UINT32 efi_system_table; /* 1c4 */
        UINT32 efi_mem_desc_size; /* 1c8 */
        UINT32 efi_mem_desc_version; /* 1cc */
        UINT32 efi_mmap; /* 1d0 */
        UINT32 efi_mmap_size; /* 1d4 */
        UINT32 efi_system_table_hi; /* 1d8 */
        UINT32 efi_mmap_hi; /* 1dc */
    } v0206;
};
UINT32 alt_mem; /* 1e0 */
UINT8 padding8[0x1e8 - 0x1e4];
UINT8 mmap_size; /* 1e8 */
UINT8 padding9[0x1f1 - 0x1e9];
UINT8 setup_sects; /* The size of the setup in sectors */
UINT16 root_flags; /* If the root is mounted readonly */
UINT16 syssize; /* obsolete */
UINT16 swap_dev; /* obsolete */
UINT16 ram_size; /* obsolete */
UINT16 vid_mode; /* Video mode control */
UINT16 root_dev; /* Default root device number */
UINT8 padding10; /* 1fe */
UINT8 ps_mouse; /* 1ff */
UINT16 jump; /* Jump instruction */
```

```
UINT32 header;      /* Magic signature "HdrS" */
UINT16 version;     /* Boot protocol version supported */
UINT32 realmode_swth; /* Boot loader hook */
UINT16 start_sys;   /* The load-low segment (obsolete) */
UINT16 kernel_version; /* Points to kernel version string */
UINT8 type_of_loader; /* Boot loader identifier */
UINT8 loadflags;    /* Boot protocol option flags */
UINT16 setup_move_size; /* Move to high memory size */
UINT32 code32_start; /* Boot loader hook */
UINT32 ramdisk_image; /* initrd load address */
UINT32 ramdisk_size;  /* initrd size */
UINT32 bootsect_kludge; /* obsolete */
UINT16 heap_end_ptr;  /* Free memory after setup end */
UINT16 pad1;         /* Unused */
UINT32 cmd_line_ptr;  /* Points to the kernel command line */
UINT8 pad2[164];     /* 22c */
E820_MAP e820_map[E820_MAX_ENTRY]; /* 2d0 */
} BOOT_PARAMETERS,*PBOOT_PARAMETERS;
#pragma pack()
```

伪代码表示如下：

```
// kernel image and initrd image file starting and ending offset
extern char * vmlinuz_start[];
extern char * vmlinuz_end[];
extern char * initrd_start[];
extern char * initrd_end[];

// main entry
EFI_STATUS
efi_main (EFI_HANDLE image, EFI_SYSTEM_TABLE *system_tab)
{
    // define variables
    EFI_STATUS efi_status = EFI_LOAD_ERROR;
    CHAR16 optstring[MAX_ARGS];
```

```
PoolAllocationType = EfiLoaderData;

// Initialize
InitializeLib(image, system_tab);

// get command line option
ParseOpt(image, system_tab, optstring);

// check sanity
If(!sanity())
    return efi_status;

// setup command line, zero page, fill E820 memory map
setup_boot_parameters(pBP, &mmap_desc, optstring);

// move kernel
load_kernel(kernel_load_address, vmlinuz_start + real_part_size, kernel_size);

// load ramdisk
load_ramdisk(ramdisk_load_address, initrd_start, ramdisk_size);

// exit boot service
efi_status =
    uefi_call_wrapper(BS->ExitBootServices, 2, image_handle, mmap_desc.cookie);

// setup GDT, IDT, and then switch to compatible mode and run kernel
trampoline(kernel_load_address);

exit:
//error: should never reach here
DBG_PRT((L"*** Fail to boot Embedded Generic Platform! ***"));
return efi_status;
}
```

以上流程如图 4-5 所示。



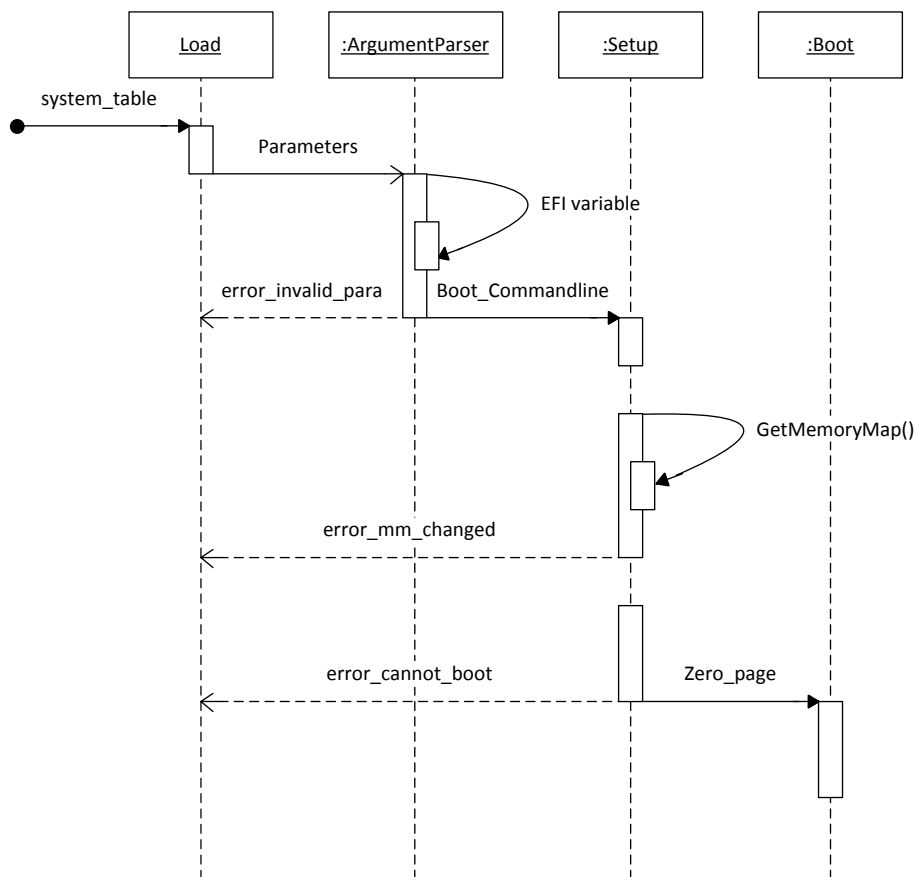


图 4-5 引导头运行顺序图

Fig. 4-5 Sequence diagram of bootloader stub

### 4.5.3 引导头部署

引导头采用 C 语言编写，采用开源的 GNU EFI 3.0i 开发工具链进行编译。通过配置管理工具，生成单一 PE+ 格式的 UEFI 可执行文件，并在 PE+ 文件头内添加版本信息以便于管理。

根据图 4-6 所示依赖关系，可通过配置管理工具 **make** 最终的生成文件。

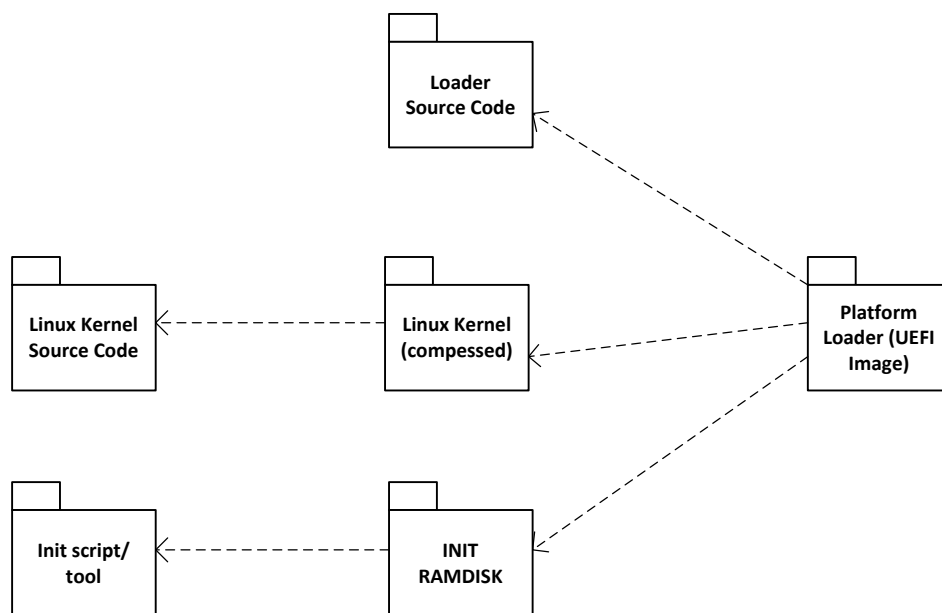


图 4-6 平台引导文件依赖关系

Fig. 4-6 Dependency of platform loader file

该文件可采用如下方式之一进行部署：

#### (1) Shell 内部命令扩展

UEFI Shell 是 UEFI 环境下的一个特殊可执行应用，它可以提供命令解释功能。由于引导头采用标准的 UEFI 可执行文件设计，因此，可以采用 EDK 编译到 UEFI Shell 内部，成为 UEFI BIOS 内置 Shell 的内部命令，如内部命令编译为 `egp` (Embed Generic Platform)，可在 Shell 下直接运行如下命令即可。

```
Shell> egp --boot-option
```

#### (2) 独立可执行镜像

为了方便更新和版本控制，本平台采用独立可执行文件的方式实现，镜像文件命名为 `egp.efi` (Embed Generic Platform)。编译完成后，可从 UEFI Shell 下，在文件系统中以可执行文件的方式调用：

```
Shell> fs0:
fs0:> cd pach-to-egp
fs0:> egp.efi --boot-option
```

独立镜像在应用环境时，按照 UEFI Firmware File System 规范，在编译时封装到一个单独的 FFS 中，并采用新产生的 GUID 命名。

#### 4.6 本章小结

本章根据平台的设计，从配置硬件开始，然后建立开发环境，创建基础设施层，最后生成终端管理模块和虚拟服务模块，对前一章的设计进行了全面的实现。

## 5 通用平台应用验证和扩展

本平台被设计为一个通用的应用平台，因此具有良好的扩展能力。本章以不同的针对性，并从不同的程度的通用层次上，对平台进行验证。

### 5.1 平台的验证环境

本平台的验证方案如表 5-1 所示。

表 5-1 验证方案列表

Table5-1 Verification Plan List

方案	描述	部署方式
终端服务验证	从 NVS 内直接加载本平台，为终端提供基础服务	NVS
网络方案验证	从 UEFI Shell 下，加载本平台引导模块，然后使用 NFS 上的提供桌面系统服务模块作为基础服务	UEFI SHELL, NFS
云服务验证	从 UEFI Shell 下，加载本平台引导模块，然后使用 DAS 上的包含云计算的服务模块作为基础服务	UEFI SHELL, DAS, CLOUD

验证方案中的完整硬件环境如图 5-1 所示。

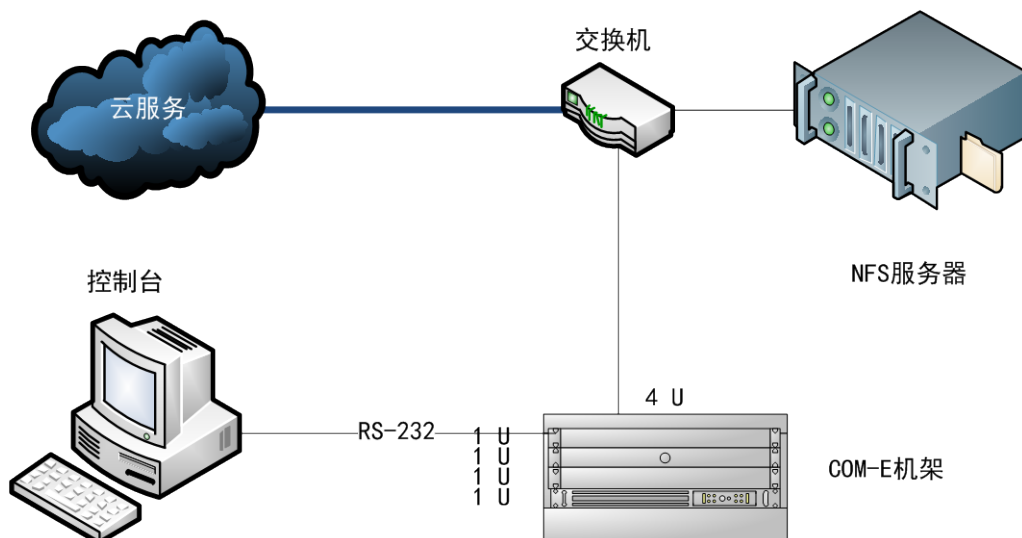


图 5-1 验证方案硬件配置全览

Fig. 5-1 Overview of Hardware Configuration of Verification Plan

本平台所使用的硬件平台采用 RadiSys CEQM67 COM-E 2.0 模块搭建。该模块采用 Basic Form Factor，在 95mm×125mm 的 PCB 上集成 PC 的核心控制电路，如：

- Intel Core i5 Sandy Bridge 嵌入式版本 CPU（含内存控制器）
- Intel HD 3000 集成显卡
- Intel Cougar Point-M 移动平台专用芯片组
- SATA 3.0 控制器（支持 AHCI 和 RAID 模式）
- Intel 82579LM 自适应 10/100/1000Mbps Base-T 以太网卡
- 16MB SPI Flash ROM（板载 8MB SPI Flash，另通过载板 Carrier 上的 8MB SPI Flash 进一步扩展 SPI Flash 的容量）
- AMI UEFI BIOS（版本 3.0.0.5）

验证时配置 2GB 内存和其他必要外围设备用于验证。

NFS 服务器采用 Radisys Rack Mount Server，主要硬件配置如下：

- 处理器: Intel Xeon E5620\*2
- 内存: 32 GB ECC RAM
- 磁盘: 150 GB SATA（15000RPM）
- 网络: 2 个 Intel 82576 Gigabit 自适应以太网口，1 个 Intel 82574L Gigabit 自适应以太网口
- 操作系统: Ubuntu 11.10 x86\_64
- IP 地址: 10.130.11.63/22

云服务采用 RackSpace 的共有云解决方案：

- 云计算: Cloud Servers (OpenStack NOVA)
- 云存储: Cloud Files (OpenStack SWIFT)
- 应用程序接口: RestFul API version 1.0

控制台采用任何带 RS-232 接口的普通 PC 即可。

## 5.2 基于终端的应用

传统 UEFI 环境下的应用程序，尤其是 Shell 内置的应用，能够在操作系统启动之前便提供系统管理工具。为了考察本平台在 OS 崩溃或启动之前，对本地硬件进行控制的能力，我们可以采用本平台的终端管理模块来提供直接管理平台。

在验证过程中，采用控制 Serial Port 作为控制台输出，从 UEFI 环境下，在仅配备 NVS（16MB SPI Flash）的环境下，按如下步骤启动到可用于终端管理的环境，用于操作系统彻底崩溃时或者无盘工作站系统的系统维护：

（1）搭建 COM-E 平台，配置硬件环境，连接 COM-E 模块上的 Serial Port 到 Console 的 Serial Port。

（2）基于 5.3 节配置要求，生成内核文件。定制时，根据 Flash 芯片容量的大小，仅保留必要的功能。同时制作一个包含必要工具和模块的 RAMDISK 文件 (initrd)，定制时采用单镜像多命令的 Busybox 提供系统命令支持。

（3）修改引导头，去掉命令行参数检测，更改内核启动的命名行参数为阈值参数，其中添加“console=ttyS0,115200n8”以便启动 ttyS<sup>[26]</sup>。然后和内核、initrd 一起编译生成新的平台引导模块，更名为 shellx64.efi，大小略小于 8MB；

（3）用平台引导模块替带原固件中 UEFI Shell 镜像文件(shellx64.efi)，并重新部署到 SPI Flash 芯片中（重新编译 UEFI BIOS，并用厂商提供的工具更新 Flash 芯片内容）。

（4）启动终端，选择启动位置为从内置 UEFI Shell 启动。由于 Shellx64.efi 文件已被替换成为本平台的引导模块，所以 UEFI 实际启动的是本平台。

（5）终端直接启动到部署在 FLASH 内的文件系统，并且通过 Console 输入输出。

通过终端软件(如 SecureCRT)截获的输出（完整输出参见附录）如下：

```
Linux version 2.6.35.10-swan (root@ubuntu) (gcc version 4.4.3 (Ubuntu 4.4.3-4ubuntu5) ) #28 SMP Tue Mar 15 00:07:45 PDT 2011
Command line: ro console=ttyS0,115200n8
....
（略）
....
Welcome to Embedded Generic Platform.
Press <Enter> to continue.
```

至此，平台成功已 Headless 的方式启动到 Linux 运行环境，该环境下可运行大量文字界面或者命令行方式的系统维护工具。以通过/proc 获取系统信息为例，可获取如图 5-2 所示处理器信息。

```

serial-com1 - SecureCRT
File Edit View Options Transfer Script Tools Help
Enter host <Alt+R>
serial-com1
DirectMap2M: 1982464 kB
/proc # cat cpuinfo
processor       : 0
vendor_id      : GenuineIntel
cpu family     : 6
model          : 42
model name     : Intel(R) Core(TM) i5-2515E CPU @ 2.50GHz
stepping       : 7
cpu MHz        : 2499.998
cache size     : 3072 KB
physical id    : 0
siblings       : 1
core id        : 0
cpu cores      : 1
apicid         : 0
initial apicid : 0
fpu            : yes
fpu_exception  : yes
cpuid level    : 13
wp             : yes
flags          : fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov
pat pse36 clflush dts acpi mmx fxsr sse sse2 ss ht tm pbe syscall nx rdtscp lm c
onstant_tsc up arch_perfmon pebs bts rep_good xtopology nonstop_tsc aperfmperf p
ni pclmulqdq dtes64 monitor ds_cpl vmx smx est tm2 ssse3 cx16 xtpr pdcm sse4_1 s
se4_2 x2apic popcnt aes xsave avx lahf_lm ida arat tpr_shadow vnmi flexpriority
ept vpid
bogomips       : 4999.99
clflush size   : 64
cache_alignme  : 64
address sizes  : 36 bits physical, 48 bits virtual
power managem  :

/proc #
Ready Serial: COM1, 115200 33, 9 33 Rows, 80 Cols Linux CAP NUM

```

图 5-2 获取硬件信息

Fig. 5-2 Inquiry hardware information

可见本平台由于采用 Linux 操作系统，因此获取系统信息非常方便，简便程度和 UEFI 内置工具不相上下。但是 Linux 有丰富的驱动，能支持诸如带 CUDA 支持的显卡适配器，能支持 NTFS 格式的文件系统。在这一点上 UEFI 内置工具远远不及。

### 5.3 基于网络的应用

UEFI 接口定义了网络部分的接口，包含 IPv4，也包含了 IPv6，FTP，iSCSI Boot 的接口定义。而本平台的模块化设计，在网络扩展方面更加灵活。

以采用 NFS 启动演示为例，本平台能直接利用 Linux 内企业级网络服务能力以及桌面支持。

在搭建好硬件环境后，按照如下步骤配置平台软件环境：

- (1) 提取 ubuntu 12.04 LTS 的桌面系统发行版安装文件中的 casper/filesystem.squashfs 文件作为平台服务模块，部署到 NFS 服务器目录下 /srv/nfsboot。
- (2) 提取 Ubuntu 12.04 安装光盘中的内核(casper/vmlinuz)及 INITRD(casper/initrd.lz)文件，编译生成引导模块 egp.efi，大小约为 21MB：
- (3) 将引导模块部署到 UEFI Shell 可识别的位置，如 FAT32 分区格式的 U 盘或者 EFI System Partition 上。
- (4) 启动终端系统，在 UEFI Shell 下，通过命令行的方式，运行 egp.efi。

egp.efi “noprompt boot=casper netboot=nfs nfsroot=10.130.11.63:/srv/nfsboot”

- (5) 稍后，系统以 DHCP 方式自动获取 IP 地址，采用 NFS 上的文件系统启动到 GNOME 的图形桌面系统并稳定运行。

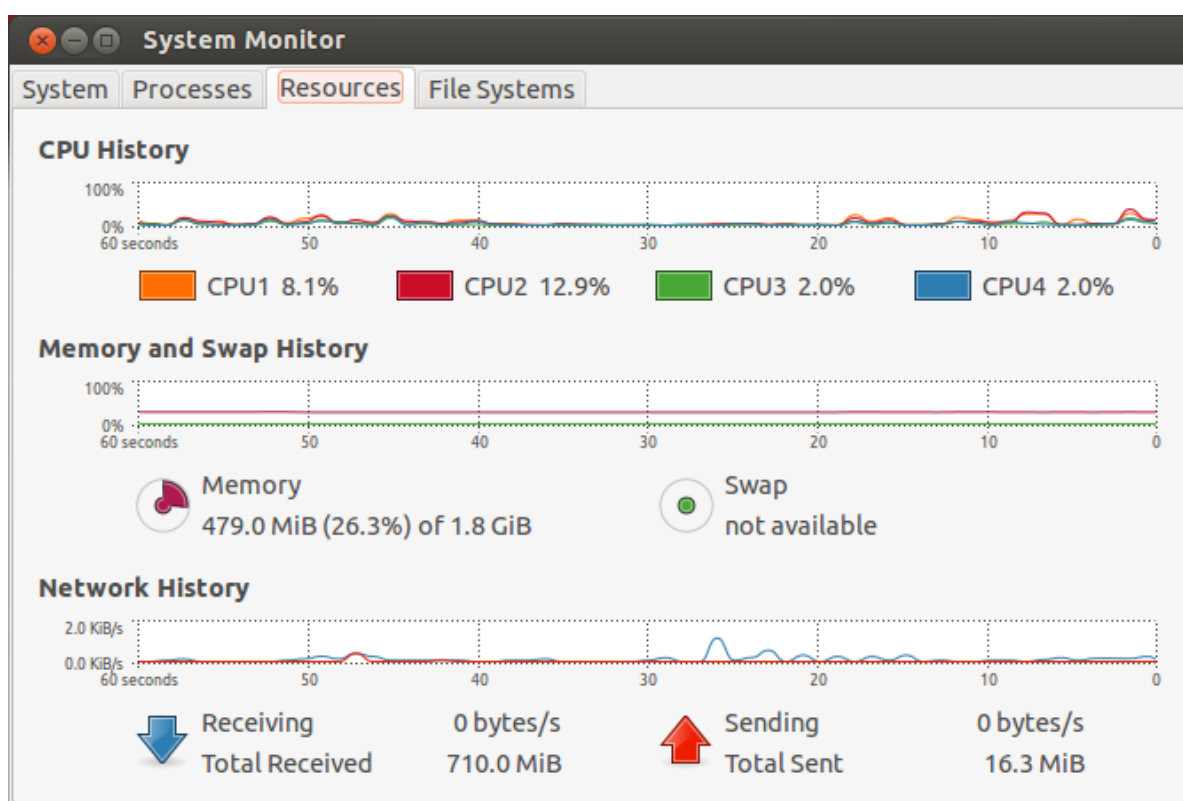


图 5-3 硬件资源监控

Fig. 5-3 Monitoring of hardware resource

平台运行时，系统硬件资源使用情况如图 5-3 所示。可以看到，平台能够识别并充分支持 SMP 系统，相比 UEFI 单进程的效率有本质性的提高。



从图 5-4 可以看出，本平台采用的只读 SQUASHFS 文件系统，在 NFS 网络环境下安全稳定运行。相比原 UEFI 系统，本平台对网络服务支持完善，资源的利用率、效率以及稳定性都非常高。

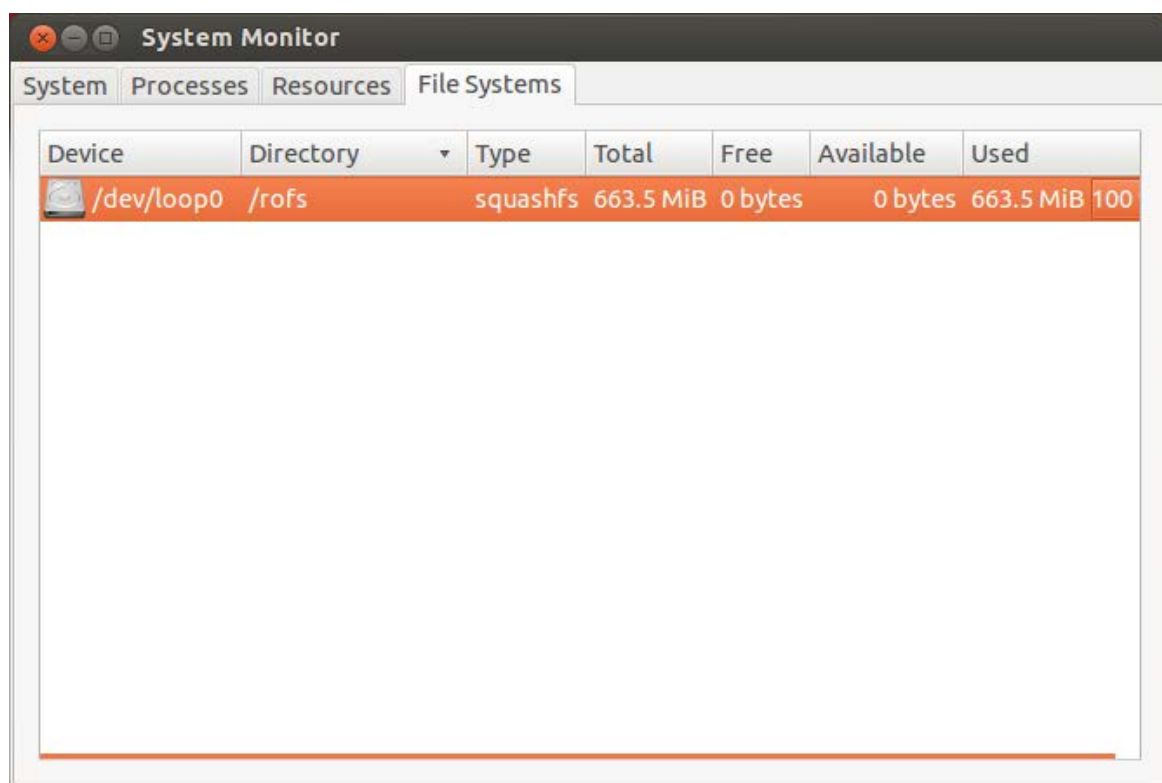


图 5-4 网络存储文件系统

Fig. 5-4 Network storage filesystem

UEFI 接口虽定义了网络接口，但其网络功能使用者甚少，而且基本没有商业应用，性能和稳定性均无法保障。相比之下 Linux 系统的网络功能稳定，已在商业上应用多年，其性能已得到业界普遍认可。而本平台可以模块化的方式，轻松获得 Linux 的全部网络优势。

#### 5.4 基于云端的应用

本平台可进一步定制上节中的服务模块，在其中添加云计算服务客户端，籍此我们可以轻易的实现对 Amazon, Google, Eucalyptus 等云解决方案的支持，实现了通过网络和云计算的扩展。

以 RackSpace 公共云计算为例，在平台中添加 python 编写的 python-rackspace-cloudservers 客户端(cloudservices)验证。

验证步骤简述如下：

(1) 首先从共有云厂商 RackSpace 获取云帐号及 RestFul API 的 API KEY

(2) 提取 casper/filesystem.squashfs 文件作为平台服务模块，向该镜像文件中加入 python-rackspace-cloudservers 及其依赖模块。

(2) 提取 Ubuntu 12.04 安装光盘中的内核及 INITRD 文件，编译生成引导模块 egp.efi，大小约为 21MB。

(3) 将引导模块部署到 UEFI Shell 可识别的位置，如 FAT32 分区格式的 U 盘或者 EFI System Partition 上；将 casper/filesystem.squashfs 文件部署到其他分区(如 ESP 或者 NTFS 分区上)

(4) 启动终端系统，在 UEFI Shell 下，通过命令行的方式，运行 egp.efi。

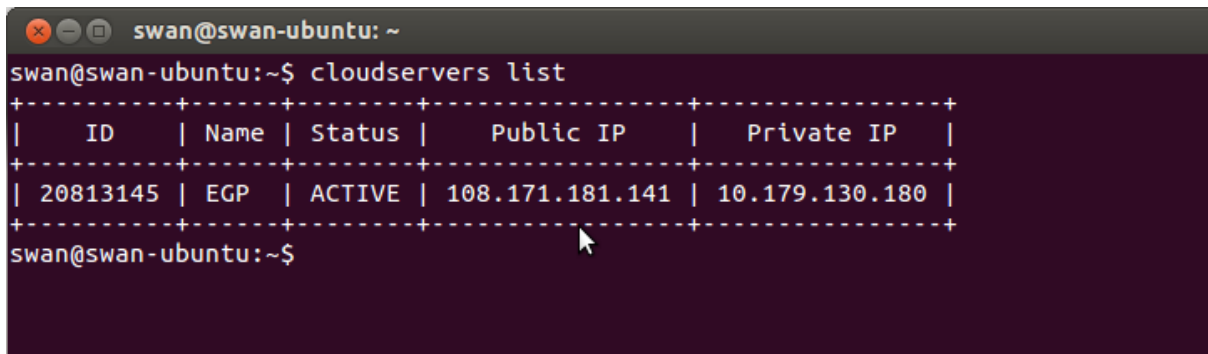
egp.efi “noprompt boot=casper”

(5) 稍后系统进入 GNOME 图形桌面系统并稳定运行。

(6) 进入平台的控制台终端(Terminal)，设置环境变量

```
export CLOUD_SERVERS_USERNAME=seanwan
export CLOUD_SERVERS_API_KEY=egpapikey
```

(7) 运行 cloudservers 客户端，查看已创建的云主机实例，验证时的截图如图 5-5 所示；



```
swan@swan-ubuntu: ~
swan@swan-ubuntu:~$ cloudservers list
+-----+-----+-----+-----+-----+
| ID    | Name | Status | Public IP | Private IP |
+-----+-----+-----+-----+-----+
| 20813145 | EGP  | ACTIVE | 108.171.181.141 | 10.179.130.180 |
+-----+-----+-----+-----+-----+
swan@swan-ubuntu:~$
```

图 5-5 云端实例列表

Fig. 5-5 Cloud instance list

(8) 进入本平台终端管理环境，调用 SSH，从云端的云主机实例获取计算服务，验证时的截图如图 5-6 所示；

```

root@EGP: ~
swan@swan-ubuntu:~$ ssh root@108.171.181.141
root@108.171.181.141's password:
Welcome to Ubuntu 11.10 (GNU/Linux 3.0.0-12-virtual x86_64)

* Documentation:  https://help.ubuntu.com/

System information as of Fri May  4 05:52:31 UTC 2012

System load:  0.0                Processes:            68
Usage of /:   9.7% of 9.35GB      Users logged in:     0
Memory usage: 14%                IP address for eth0: 108.171.181.141
Swap usage:   0%                IP address for eth1: 10.179.130.180

Graph this data and manage this system at https://landscape.canonical.com/
Last login: Fri May  4 05:31:44 2012 from 218.83.212.14
root@EGP:~# lscpu
Architecture:        x86_64
CPU op-mode(s):      32-bit, 64-bit
Byte Order:          Little Endian
CPU(s):              4
On-line CPU(s) list: 0-3
Thread(s) per core:  1
Core(s) per socket:  1
CPU socket(s):       4
NUMA node(s):        1
Vendor ID:           AuthenticAMD
CPU family:           16
Model:               4
Stepping:            2
CPU MHz:             2200.088
BogoMIPS:            4400.17
Hypervisor vendor:   Xen
Virtualization type: para
L1d cache:           64K
L1i cache:           64K
L2 cache:            512K
L3 cache:            6144K
NUMA node0 CPU(s):   0-3
root@EGP:~#

```

图 5-6 云端服务演示

Fig. 5-6 Cloud service demo

(9) 根据应用的需要，进入 RackSpace 管理平台，通过 Resize 功能，动态调整云端计算资源。

```

swan@swan-ubuntu:~$ cloudservers flavor-list
+-----+-----+-----+-----+
| ID | Name | RAM | Disk |

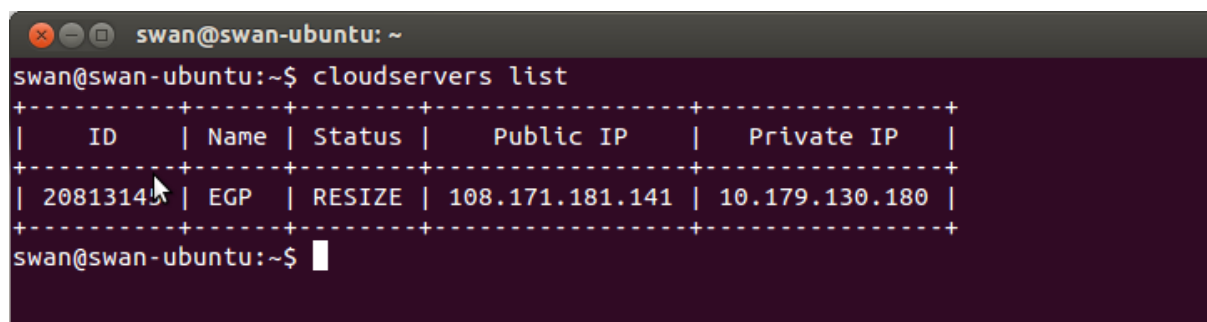
```

1	256 server	256	10
2	512 server	512	20
3	1GB server	1024	40
4	2GB server	2048	80
5	4GB server	4096	160
6	8GB server	8192	320
7	15.5GB server	15872	620
8	30GB server	30720	1200

以从 ID 1 方案变更到 ID 2 方案为例：

```
swan@swan-ubuntu:~$ cloudservers resize 20813145 2
```

此时如图 5-7 所示，云端开始在后台调整实例资源，稍有停顿后，便可重新进入服务状态。



```
swan@swan-ubuntu:~$ cloudservers list
+-----+-----+-----+-----+-----+
| ID | Name | Status | Public IP | Private IP |
+-----+-----+-----+-----+-----+
| 20813145 | EGP | RESIZE | 108.171.181.141 | 10.179.130.180 |
+-----+-----+-----+-----+-----+
swan@swan-ubuntu:~$
```

图 5-7 云端计算资源调整演示

Fig. 5-7 Cloud service resource resizing demo

相比 UEFI 下单进程并且完全依赖硬件的方式，云计算方式采用抽象和虚拟的计算资源，将计算的实际发生位置隐藏到云端，更进一步的提高了计算的通用性和可用性。

## 5.5 本章小结

本章采用针对性不同的三种方案对本平台进行了实际应用验证，不仅验证了本平台达到设计目标，还证明了平台的在本地终端中的实际性能至少能达到 UEFI 原生应用的水平，而且在网络应用方面的扩展性能也在 UEFI 之上，并从侧面举例说明了本平台实际生产应用的方向。

## 6 总结和展望

### 6.1 本文总结

本文采用云计算和虚拟化库技术对现有 UEFI 构架的个人计算机平台进行了扩充，主要工作包括：

- 1) 对现有 UEFI 平台下的应用存在的问题进行分析，采用扩展的新平台进行全新的建模和定义；
- 2) 采用模块化和层次化的方式，从软件工程的角度构架出解决方案，使得系统具有良好的可扩展性能；
- 3) 充分利用了抽象和虚拟化的思想和技术手段，最大化的实现了计算通用；
- 4) 充分实现了对 UEFI 的完全兼容；
- 5) 充分借助开源项目，采用 Linux 下的应用来对 UEFI 平台进行扩展，实现了对资源利用率的提高

目前，采用平台的系统搭载了经过定制的终端管理模块已交付客户，并已应用于最终产品中。测试和试运行表明，该平台由于其高可靠性、高可用性及易扩展的特性，不仅保证了产品的功能，同时还极大的方便了通用应用的开发，取得了良好的市场反响。

### 6.2 项目成功因素分析

基于 UEFI 系统的 Linux 通用应用平台的项目的成功实施，得益于多方面的因素：

- 1) 明确的接口定义。UEFI 规范的明确和规范化的接口定义，使得传统上靠约定俗成的方式来连接硬件和操作系统的方式成为过去。统一的接口也使得跨平台的应用变得简便和灵活，不仅可以大大降低重复性劳动，还提高了代码的重用性和可移植性。
- 2) 完善的模块化设计。从底层的 UEFI 到上层的服务模块，充分考虑了模块化的设计理念。模块化的设计使得本平台在做扩展时能够很安全方便的通过模块替换来变更服务以及按需分配。

- 3) 良好的计算抽象化。抽象大大降低了应用对于底层硬件的依赖性。虚拟化技术所推动的云计算，在平衡资源和简化计算模型方面取得了很好的成果。得益于云计算提供的计算方式的变革，传统的分散独立的计算方式也将产生巨大的变革，逐渐朝着通用的计算模式发展。
- 4) 将更加通用的 Linux 平台作为扩展的目标。在扩充后的平台上，Linux 提供了比 UEFI 更强通用性。

### 6.3 展望

回顾已经完成的工作，本通用平台已满足了客户目前的需求，但是下一步的研究目标将在通用性的方面做进一步的提高，比如：

深度定制本平台，通过构建终端到云端的映射框架，使本地的特定计算资源可直接映射到远程的云端虚拟机上。映射以内核加驱动的方式，对上层应用保持透明，这样能使得计算更加通用，更进一步提高资源的利用率。

## 参考文献

- [1]. Intel, Intel® 64 and IA-32 Architectures Software Developer's Manual [S], Version 042, 2012
- [2]. Vincent Zimmer, Michael Rothman, Robert Hale, Beyond BIOS: Developing with the Unified Extensible Firmware Interface [M], 2nd Edition, Intel Press, 2011, 1-266
- [3]. Unified EFI, Inc., Unified Extensible Firmware Interface Specification [S], Version 2.3, 2009
- [4]. Intel, HP, Microsoft, Advanced Configuration and Power Interface Specification [S], Revision 4.0, 2009, 5-496
- [5]. Unified EFI, Inc., UEFI Platform Initialization Specifications VOLUME 3 [S], Version 1.2, 2009, 5-28
- [6]. Microsoft, Microsoft Portable Executable and Common Object File Format Specification [S], Revision 8.2, 2010
- [7]. Michael Rothman, Tim Lewis, Vincent Zimmer, Robert Hale, Harnessing the UEFI Shell: Moving the Platform Beyond DOS [M], Intel Press, 2010, 25-26
- [8]. Goldberg R P, Architecture of Virtual Machines, Proceedings of the Workshop on Virtual Computer Systems [C], 1973, 74-112
- [9]. Waldspurger C A, Memory Resource Management in VMware ESX Server [C], Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI '02), 2002, 181-194.
- [10]. Ian P, Keir F, Steve H 等, Xen 3.0 and the Art of Virtualization [C], Proceedings of the Ottawa Linux Symposium, 2005
- [11]. Robin J S, Irvine C E, Analysis of the Intel Pentium's Ability to Support a Secure Virtual Machine Monitor [C], Proceedings of the 9th USENIX Security Symposium, 2000, 129-144
- [12]. Combining hardware and Software to Provide an Improved Microprocessor [P], United States, Patent Number 6031992, 2000

- [13]. Yaozu D, Shaofan L, Asit M 等, Extending Xen with Intel Virtualization Technology[J], Intel Technology Journal, 2006, 193-203
- [14]. DMTF, Open Virtualization Format Specification [S], Version: 1.1.0, 2012
- [15]. ANSI, ANSI standard INCITS 469-2010 [S], 2011
- [16]. Craig Larman 著, 李洋等译, UML 和模式应用 (原书第 3 版) [M], 机械工业出版社, 2006, 5
- [17]. (美) Robert Love 著, 陈莉君等译, LINUX 内核设计与实现 [M], 机械工业出版社, 2006, 8-15
- [18]. Jonathan Corbet, Linux Device Driver [M], O'Reilly Media, 2005, 5-9
- [19]. PICMG, PICMG COM.0 Express Module Base Specification [S], Revision 2.0, 2010, 1-13
- [20]. Filesystem Hierarchy Standard Group, Filesystem Hierarchy Standard [S], Version 2.3, 2004
- [21]. (美) 坦思利著, 徐焱等译, LINUX 与 UNIX SHELL 编程指南[M], 机械工业出版社, 2006, 1-55
- [22]. Bellard F, QEMU, a Fast and Portable Dynamic Translator[C], Proceedings of the USENIX Annual Technical Conference (USENIX'05), 2005, 41-46.
- [23]. Ken Pepple, Deploying OpenStack: Creating Open Source Clouds [M], O'Reilly Media, 2011
- [24]. Rackspace, Cloud Server Development Guide[S], Version API 1.0, 2012, 3-20
- [25]. 谭浩强, C 程序设计 (第三版) [M], 北京, 清华大学出版社, 2005
- [26]. (美) W.Richard Stevens Stephen A. Rago 著, 尤晋元, 张亚英, 戚正伟译, UNIX 环境高级编程 (第二版) [M], 北京, 人民邮电出版社, 2006



## 附录 Console 输出日志

```
Linux version 2.6.35.10-swan (root@ubuntu) (gcc version 4.4.3 (Ubuntu 4.4.3-4ubuntu5) ) #28 SMP
Tue Mar 15 00:07:45 PDT 2011
Command line: ro console=ttyS0,115200n8
BIOS-provided physical RAM map:
BIOS-e820: 0000000000000000 - 00000000000a0000 (usable)
BIOS-e820: 00000000000a0000 - 0000000000100000 (reserved)
BIOS-e820: 0000000000100000 - 0000000002000000 (usable)
BIOS-e820: 0000000002000000 - 0000000002020000 (reserved)
BIOS-e820: 0000000002020000 - 0000000004000000 (usable)
BIOS-e820: 0000000004000000 - 0000000004020000 (reserved)
BIOS-e820: 0000000004020000 - 00000000078d2000 (usable)
BIOS-e820: 00000000078d2000 - 00000000078d2100 (unusable)
BIOS-e820: 00000000078d2100 - 00000000078d3200 (reserved)
BIOS-e820: 00000000078d3200 - 00000000078d4800 (unusable)
BIOS-e820: 00000000078d4800 - 00000000078d7800 (reserved)
BIOS-e820: 00000000078d7800 - 00000000078d8e00 (unusable)
BIOS-e820: 00000000078d8e00 - 00000000078d9e00 (usable)
BIOS-e820: 00000000078d9e00 - 00000000078da000 (reserved)
BIOS-e820: 00000000078da000 - 00000000078da200 (usable)
BIOS-e820: 00000000078da200 - 00000000078db800 (reserved)
BIOS-e820: 00000000078db800 - 00000000078db900 (usable)
BIOS-e820: 00000000078db900 - 00000000078de800 (reserved)
BIOS-e820: 00000000078de800 - 00000000078f2e00 (usable)
BIOS-e820: 00000000078f2e00 - 00000000078f7c00 (ACPI NVS)
BIOS-e820: 00000000078f7c00 - 00000000078f7f00 (usable)
BIOS-e820: 00000000078f7f00 - 00000000078fe800 (ACPI NVS)
BIOS-e820: 00000000078fe800 - 00000000078ffd00 (usable)
BIOS-e820: 00000000078ffd00 - 0000000007900000 (ACPI data)
BIOS-e820: 0000000007900000 - 0000000007da0000 (reserved)
BIOS-e820: 0000000007da0000 - 0000000007fc0000 (reserved)
BIOS-e820: 0000000007fc0000 - 0000000007fec01000 (reserved)
BIOS-e820: 0000000007fec01000 - 0000000007fed14000 (reserved)
BIOS-e820: 0000000007fed14000 - 0000000007fed1a000 (reserved)
BIOS-e820: 0000000007fed1a000 - 0000000007fed20000 (reserved)
BIOS-e820: 0000000007fed20000 - 0000000007fee01000 (reserved)
BIOS-e820: 0000000007fee01000 - 0000000007fee00000 (reserved)
BIOS-e820: 0000000007fee00000 - 0000000007ffa00000 (reserved)
BIOS-e820: 0000000007ffa00000 - 0000000007ffe00000 (reserved)
BIOS-e820: 0000000007ffe00000 - 0000000007ff000000 (reserved)
BIOS-e820: 0000000007ff000000 - 0000000007ff600000 (usable)
NX (Execute Disable) protection: active
DMI not present or invalid.
No AGP bridge found
last_pfn = 0x100600 max_arch_pfn = 0x400000000
x86 PAT enabled: cpu 0, old 0x7040600070406, new 0x7010600070106
last_pfn = 0x78ffd max_arch_pfn = 0x400000000
```

Scanning 1 areas for low memory corruption  
modified physical RAM map:  
modified: 0000000000000000 - 0000000000010000 (reserved)  
modified: 0000000000010000 - 00000000000a0000 (usable)  
modified: 00000000000a0000 - 00000000000100000 (reserved)  
modified: 00000000000100000 - 0000000002000000 (usable)  
modified: 0000000002000000 - 0000000002020000 (reserved)  
modified: 0000000002020000 - 0000000004000000 (usable)  
modified: 0000000004000000 - 0000000004020000 (reserved)  
modified: 0000000004020000 - 0000000078d20000 (usable)  
modified: 0000000078d20000 - 0000000078d21000 (unusable)  
modified: 0000000078d21000 - 0000000078d32000 (reserved)  
modified: 0000000078d32000 - 0000000078d48000 (unusable)  
modified: 0000000078d48000 - 0000000078d78000 (reserved)  
modified: 0000000078d78000 - 0000000078d8e000 (unusable)  
modified: 0000000078d8e000 - 0000000078d9e000 (usable)  
modified: 0000000078d9e000 - 0000000078da0000 (reserved)  
modified: 0000000078da0000 - 0000000078da2000 (usable)  
modified: 0000000078da2000 - 0000000078db8000 (reserved)  
modified: 0000000078db8000 - 0000000078db9000 (usable)  
modified: 0000000078db9000 - 0000000078de8000 (reserved)  
modified: 0000000078de8000 - 0000000078f2e000 (usable)  
modified: 0000000078f2e000 - 0000000078f7c000 (ACPI NVS)  
modified: 0000000078f7c000 - 0000000078f7f000 (usable)  
modified: 0000000078f7f000 - 0000000078fe8000 (ACPI NVS)  
modified: 0000000078fe8000 - 0000000078ffd000 (usable)  
modified: 0000000078ffd000 - 0000000079000000 (ACPI data)  
modified: 0000000079000000 - 000000007da00000 (reserved)  
modified: 000000007da00000 - 00000000fc000000 (reserved)  
modified: 00000000fc000000 - 00000000fec01000 (reserved)  
modified: 00000000fec01000 - 00000000fed14000 (reserved)  
modified: 00000000fed14000 - 00000000fed18000 (reserved)  
modified: 00000000fed18000 - 00000000fed1a000 (reserved)  
modified: 00000000fed1a000 - 00000000fed20000 (reserved)  
modified: 00000000fed20000 - 00000000fee01000 (reserved)  
modified: 00000000fee01000 - 00000000ffa00000 (reserved)  
modified: 00000000ffa00000 - 00000000ffc00000 (reserved)  
modified: 00000000ffc00000 - 00000000100000000 (reserved)  
modified: 00000000100000000 - 00000000100600000 (usable)  
init\_memory\_mapping: 0000000000000000-0000000078ffd000  
init\_memory\_mapping: 00000000100000000-00000000100600000  
RAMDISK: 02b65000 - 02eaa000  
ACPI Error: A valid RSDP was not found (20100428/tbxroot-219)  
No NUMA configuration found  
Faking a node at 0000000000000000-00000000100600000  
Initmem setup node 0 0000000000000000-00000000100600000  
  NODE\_DATA [00000000100000000 - 00000000100004fff]  
Zone PFN ranges:  
  DMA  0x00000010 -> 0x000001000  
  DMA32 0x000001000 -> 0x000100000  
  Normal 0x000100000 -> 0x000100600  
Movable zone start PFN for each node

```
early_node_map[11] active PFN ranges
 0: 0x00000010 -> 0x000000a0
 0: 0x00000100 -> 0x00020000
 0: 0x00020200 -> 0x00040000
 0: 0x00040200 -> 0x00078d20
 0: 0x00078d8e -> 0x00078d9e
 0: 0x00078da0 -> 0x00078da2
 0: 0x00078db8 -> 0x00078db9
 0: 0x00078de8 -> 0x00078f2e
 0: 0x00078f7c -> 0x00078f7f
 0: 0x00078fe8 -> 0x00078ffd
 0: 0x00100000 -> 0x00100600
SMP: Allowing 1 CPUs, 0 hotplug CPUs
Allocating PCI resources starting at 7da00000 (gap: 7da00000:7a600000)
setup_percpu: NR_CPUS:64 nr_cpumask_bits:64 nr_cpu_ids:1 nr_node_ids:1
PERCPU: Embedded 26 pages/cpu @ffff880001800000 s76800 r8192 d21504 u2097152
pcpu-alloc: s76800 r8192 d21504 u2097152 alloc=1*2097152
pcpu-alloc: [0] 0
Built 1 zonelists in Node order, mobility grouping on. Total pages: 481292
Policy zone: Normal
Kernel command line: ro console=ttyS0,115200n8
PID hash table entries: 4096 (order: 3, 32768 bytes)
xsave/xrstor: enabled xstate_bv 0x7, cntxt size 0x340
Checking aperture...
No AGP bridge found
Subtract (68 early reservations)
#1 [0001000000 - 00016804e8] TEXT DATA BSS
#2 [0002b65000 - 0002eaa000] RAMDISK
#3 [000009fc00 - 0000100000] BIOS reserved
#4 [0000010000 - 0000012000] TRAMPOLINE
#5 [0000012000 - 0000014000] PGTABLE
#6 [0000014000 - 0000015000] PGTABLE
#7 [0100000000 - 0100005000] NODE_DATA
#8 [0001680500 - 0001681500] BOOTMEM
#9 [0001a81500 - 0001a81698] BOOTMEM
#10 [0100005000 - 0100006000] BOOTMEM
#11 [0100006000 - 0100007000] BOOTMEM
#12 [0003000000 - 0004e00000] MEMMAP 0
#13 [0001681500 - 0001681680] BOOTMEM
#14 [0001681680 - 0001699680] BOOTMEM
#15 [0001699680 - 0001699740] BOOTMEM
#16 [000169a000 - 000169b000] BOOTMEM
#17 [0001699740 - 0001699ee8] BOOTMEM
#18 [0001699f00 - 0001699f68] BOOTMEM
#19 [0001699f80 - 0001699fe8] BOOTMEM
#20 [000169b000 - 000169b068] BOOTMEM
#21 [000169b080 - 000169b0e8] BOOTMEM
#22 [000169b100 - 000169b168] BOOTMEM
#23 [000169b180 - 000169b1e8] BOOTMEM
#24 [000169b200 - 000169b268] BOOTMEM
```

```
#25 [000169b280 - 000169b2e8] BOOTMEM
#26 [000169b300 - 000169b368] BOOTMEM
#27 [000169b380 - 000169b3e8] BOOTMEM
#28 [000169b400 - 000169b468] BOOTMEM
#29 [000169b480 - 000169b4e8] BOOTMEM
#30 [000169b500 - 000169b568] BOOTMEM
#31 [000169b580 - 000169b5e8] BOOTMEM
#32 [000169b600 - 000169b668] BOOTMEM
#33 [000169b680 - 000169b6e8] BOOTMEM
#34 [000169b700 - 000169b768] BOOTMEM
#35 [000169b780 - 000169b7e8] BOOTMEM
#36 [000169b800 - 000169b868] BOOTMEM
#37 [000169b880 - 000169b8e8] BOOTMEM
#38 [000169b900 - 000169b968] BOOTMEM
#39 [000169b980 - 000169b9e8] BOOTMEM
#40 [000169ba00 - 000169ba68] BOOTMEM
#41 [000169ba80 - 000169bae8] BOOTMEM
#42 [000169bb00 - 000169bb68] BOOTMEM
#43 [000169bb80 - 000169bbe8] BOOTMEM
#44 [000169bc00 - 000169bc68] BOOTMEM
#45 [000169bc80 - 000169bce8] BOOTMEM
#46 [000169bd00 - 000169bd68] BOOTMEM
#47 [000169bd80 - 000169bde8] BOOTMEM
#48 [000169be00 - 000169be68] BOOTMEM
#49 [000169be80 - 000169bee8] BOOTMEM
#50 [000169bf00 - 000169bf68] BOOTMEM
#51 [000169bf80 - 000169bfe8] BOOTMEM
#52 [000169c000 - 000169c01e] BOOTMEM
#53 [000169c040 - 000169c05e] BOOTMEM
#54 [0001800000 - 000181a000] BOOTMEM
#55 [000169e080 - 000169e088] BOOTMEM
#56 [000169e0c0 - 000169e0c8] BOOTMEM
#57 [000169e100 - 000169e104] BOOTMEM
#58 [000169e140 - 000169e148] BOOTMEM
#59 [000169e180 - 000169e2d0] BOOTMEM
#60 [000169e300 - 000169e380] BOOTMEM
#61 [000169e380 - 000169e400] BOOTMEM
#62 [000169e400 - 00016a6400] BOOTMEM
#63 [000169c080 - 000169c3c0] BOOTMEM
#64 [0004e00000 - 0008e00000] BOOTMEM
#65 [00016a6400 - 00016c6400] BOOTMEM
#66 [00016c6400 - 0001706400] BOOTMEM
#67 [0000016800 - 000001e800] BOOTMEM
```

Memory: 1875604k/4200448k available (3617k kernel code, 2217852k absent, 106992k reserved, 1961k data, 596k init)

SLUB: Genslabs=14, HWalign=64, Order=0-3, MinObjects=0, CPUs=1, Nodes=1

Hierarchical RCU implementation.

RCU-based detection of stalled CPUs is disabled.

Verbose stalled-CPU detection is disabled.

NR\_IRQS:4352 nr\_irqs:256

Console: colour dummy device 80x25  
console [ttyS0] enabled  
Fast TSC calibration using PIT  
Detected 2500.257 MHz processor.  
Calibrating delay loop (skipped), value calculated using timer frequency.. 5000.51 BogoMIPS  
(lpj=2500257)  
pid\_max: default: 32768 minimum: 301  
Dentry cache hash table entries: 262144 (order: 9, 2097152 bytes)  
Inode-cache hash table entries: 131072 (order: 8, 1048576 bytes)  
Mount-cache hash table entries: 256  
CPU: Physical Processor ID: 0  
CPU: Processor Core ID: 0  
mce: CPU supports 7 MCE banks  
using mwait in idle threads.  
Performance Events: PEBS fmt1+, generic architected perfmon, Intel PMU driver.  
... version: 3  
... bit width: 48  
... generic registers: 4  
... value mask: 0000fffffffffffff  
... max period: 000000007fffffff  
... fixed-purpose events: 3  
... event mask: 0000000070000000f  
SMP alternatives: switching to UP code  
Freeing SMP alternatives: 20k freed  
Setting APIC routing to flat  
weird, boot CPU (#0) not listed by the BIOS.  
SMP motherboard not detected.  
SMP disabled  
Brought up 1 CPUs  
Total of 1 processors activated (5000.51 BogoMIPS).  
devtmpfs: initialized  
NET: Registered protocol family 16  
PCI: Using configuration type 1 for base access  
bio: create slab <bio-0> at 0  
ACPI: Interpreter disabled.  
vgaarb: loaded  
SCSI subsystem initialized  
usbcore: registered new interface driver usbfs  
usbcore: registered new interface driver hub  
usbcore: registered new device driver usb  
PCI: Probing PCI hardware  
pci 0000:00:1c.0: PCI bridge to [bus 01-01]  
pci 0000:00:1c.4: PCI bridge to [bus 02-02]  
pci 0000:00:1c.5: PCI bridge to [bus 03-03]  
pci 0000:00:1c.6: PCI bridge to [bus 04-0b]  
vgaarb: device added: PCI:0000:00:02.0,decodes=io+mem,owns=io+mem,locks=none  
pci 0000:00:1f.0: PIIX/ICH IRQ router [8086:1c4f]  
Switching to clocksource tsc  
Slow work thread pool: Starting up  
Slow work thread pool: Ready

FS-Cache: Loaded  
pnp: PnP ACPI: disabled  
pci 0000:00:1c.0: PCI bridge to [bus 01-01]  
pci 0000:00:1c.0: bridge window [io disabled]  
pci 0000:00:1c.0: bridge window [mem disabled]  
pci 0000:00:1c.0: bridge window [mem pref disabled]  
pci 0000:00:1c.4: PCI bridge to [bus 02-02]  
pci 0000:00:1c.4: bridge window [io disabled]  
pci 0000:00:1c.4: bridge window [mem disabled]  
pci 0000:00:1c.4: bridge window [mem pref disabled]  
pci 0000:00:1c.5: PCI bridge to [bus 03-03]  
pci 0000:00:1c.5: bridge window [io disabled]  
pci 0000:00:1c.5: bridge window [mem disabled]  
pci 0000:00:1c.5: bridge window [mem pref disabled]  
pci 0000:00:1c.6: PCI bridge to [bus 04-0b]  
pci 0000:00:1c.6: bridge window [io 0xa000-0xdfff]  
pci 0000:00:1c.6: bridge window [mem 0xf6c00000-0xf7cfffff]  
pci 0000:00:1c.6: bridge window [mem 0xf0000000-0xf10fffff 64bit pref]  
pci 0000:00:1c.0: found PCI INT A -> IRQ 11  
pci 0000:00:1c.0: sharing IRQ 11 with 0000:00:02.0  
pci 0000:00:1c.0: sharing IRQ 11 with 0000:00:16.0  
pci 0000:00:1c.0: sharing IRQ 11 with 0000:00:1a.0  
pci 0000:00:1c.0: sharing IRQ 11 with 0000:00:1c.4  
pci 0000:00:1c.4: found PCI INT A -> IRQ 11  
pci 0000:00:1c.4: sharing IRQ 11 with 0000:00:02.0  
pci 0000:00:1c.4: sharing IRQ 11 with 0000:00:16.0  
pci 0000:00:1c.4: sharing IRQ 11 with 0000:00:1a.0  
pci 0000:00:1c.4: sharing IRQ 11 with 0000:00:1c.0  
pci 0000:00:1c.5: found PCI INT B -> IRQ 11  
pci 0000:00:1c.6: found PCI INT C -> IRQ 10  
pci 0000:00:1c.6: sharing IRQ 10 with 0000:00:1f.3  
NET: Registered protocol family 2  
IP route cache hash table entries: 65536 (order: 7, 524288 bytes)  
TCP established hash table entries: 262144 (order: 10, 4194304 bytes)  
TCP bind hash table entries: 65536 (order: 8, 1048576 bytes)  
TCP: Hash tables configured (established 262144 bind 65536)  
TCP reno registered  
UDP hash table entries: 1024 (order: 3, 32768 bytes)  
UDP-Lite hash table entries: 1024 (order: 3, 32768 bytes)  
NET: Registered protocol family 1  
Trying to unpack rootfs image as initramfs...  
Freeing initrd memory: 3348k freed  
PCI-DMA: Using software bounce buffering for IO (SWIOTLB)  
Placing 64MB software IO TLB between ffff880004e00000 - ffff880008e00000  
software IO TLB at phys 0x4e00000 - 0x8e00000  
platform rtc\_cmos: registered platform RTC device (no PNP device found)  
microcode: CPU0 sig=0x206a7, pf=0x10, revision=0x18  
microcode: Microcode Update Driver: v2.00 <tigran@aivazian.fsnet.co.uk>, Peter Oruba  
Scanning for low memory corruption every 60 seconds  
HugeTLB registered 2 MB page size, pre-allocated 0 pages

```
fuse init (API version 7.14)
msgmni has been set to 3669
Block layer SCSI generic (bsg) driver version 0.4 loaded (major 253)
io scheduler noop registered
io scheduler deadline registered
io scheduler cfq registered (default)
Non-volatile memory driver v1.3
Linux agpgart interface v0.103
agpgart-intel 0000:00:00.0: Intel Sandybridge Chipset
agpgart-intel 0000:00:00.0: detected 65532K stolen memory, trimming to 32768K
agpgart-intel 0000:00:00.0: AGP aperture is 256M @ 0xe0000000
[drm] Initialized drm 1.1.0 20060810
i915 0000:00:02.0: found PCI INT A -> IRQ 11
i915 0000:00:02.0: sharing IRQ 11 with 0000:00:16.0
i915 0000:00:02.0: sharing IRQ 11 with 0000:00:1a.0
[drm] detected 63M stolen memory, trimming to 32M
[drm] set up 32M of stolen space
Console: switching to colour frame buffer device 128x48
fb0: inteldrmfb frame buffer device
drm: registered panic notifier
[drm] Initialized i915 1.6.0 20080730 for 0000:00:02.0 on minor 0
Serial: 8250/16550 driver, 4 ports, IRQ sharing enabled
serial8250: ttyS0 at I/O 0x3f8 (irq = 4) is a 16550A
brd: module loaded
loop: module loaded
ata_piix 0000:00:1f.2: found PCI INT B -> IRQ 10
ata_piix 0000:00:1f.2: sharing IRQ 10 with 0000:00:1f.5
ata_piix 0000:00:1f.2: MAP [ P0 P2 P1 P3 ]
ata_piix 0000:00:1f.2: SCR access via SIDPR is available but doesn't work
scsi0 : ata_piix
scsi1 : ata_piix
ata1: SATA max UDMA/133 cmd 0xe130 ctl 0xe120 bmdma 0xe0f0 irq 10
ata2: SATA max UDMA/133 cmd 0xe110 ctl 0xe100 bmdma 0xe0f8 irq 10
ata_piix 0000:00:1f.5: found PCI INT B -> IRQ 10
ata_piix 0000:00:1f.5: sharing IRQ 10 with 0000:00:1f.2
ata_piix 0000:00:1f.5: MAP [ P0 -- P1 -- ]
ata_piix 0000:00:1f.5: SCR access via SIDPR is available but doesn't work
scsi2 : ata_piix
scsi3 : ata_piix
ata3: SATA max UDMA/133 cmd 0xe0d0 ctl 0xe0c0 bmdma 0xe090 irq 10
ata4: SATA max UDMA/133 cmd 0xe0b0 ctl 0xe0a0 bmdma 0xe098 irq 10
e1000e: Intel(R) PRO/1000 Network Driver - 1.0.2-k4
e1000e: Copyright (c) 1999 - 2009 Intel Corporation.
e100: Intel(R) PRO/100 Network Driver, 3.5.24-k2-NAPI
e100: Copyright(c) 1999-2006 Intel Corporation
ehci_hcd: USB 2.0 'Enhanced' Host Controller (EHCI) Driver
ehci_hcd 0000:00:1a.0: found PCI INT A -> IRQ 11
ehci_hcd 0000:00:1a.0: sharing IRQ 11 with 0000:00:16.0
ehci_hcd 0000:00:1a.0: EHCI Host Controller
ehci_hcd 0000:00:1a.0: new USB bus registered, assigned bus number 1
```



```
ehci_hcd 0000:00:1a.0: debug port 2
ehci_hcd 0000:00:1a.0: irq 11, io mem 0xf7d27000
ehci_hcd 0000:00:1a.0: USB 2.0 started, EHCI 1.00
hub 1-0:1.0: USB hub found
hub 1-0:1.0: 3 ports detected
ehci_hcd 0000:00:1d.0: found PCI INT A -> IRQ 5
ehci_hcd 0000:00:1d.0: EHCI Host Controller
ehci_hcd 0000:00:1d.0: new USB bus registered, assigned bus number 2
ehci_hcd 0000:00:1d.0: debug port 2
ehci_hcd 0000:00:1d.0: irq 5, io mem 0xf7d26000
ehci_hcd 0000:00:1d.0: USB 2.0 started, EHCI 1.00
hub 2-0:1.0: USB hub found
hub 2-0:1.0: 3 ports detected
ohci_hcd: USB 1.1 'Open' Host Controller (OHCI) Driver
uhci_hcd: USB Universal Host Controller Interface driver
Initializing USB Mass Storage driver...
usbcore: registered new interface driver usb-storage
USB Mass Storage support registered.
usbcore: registered new interface driver libusual
PNP: No PS/2 controller found. Probing ports directly.
i8042.c: No controller found.
mice: PS/2 mouse device common for all mice
rtc_cmos rtc_cmos: rtc core: registered rtc_cmos as rtc0
rtc0: alarms up to one day, 114 bytes nvram
i801_smbus 0000:00:1f.3: found PCI INT C -> IRQ 10
device-mapper: ioctl: 4.17.0-ioctl (2010-03-05) initialised: dm-devel@redhat.com
cpuidle: using governor ladder
cpuidle: using governor menu
usbcore: registered new interface driver hiddev
usbcore: registered new interface driver usbhid
usbhid: USB HID core driver
Netfilter messages via NETLINK v0.30.
nf_conntrack version 0.5.0 (16384 buckets, 65536 max)
ctnetlink v0.93: registering with nfnetlink.
ip_tables: (C) 2000-2006 Netfilter Core Team
TCP cubic registered
Initializing XFRM netlink socket
NET: Registered protocol family 10
ip6_tables: (C) 2000-2006 Netfilter Core Team
IPv6 over IPv4 tunneling driver
NET: Registered protocol family 17
Freeing unused kernel memory: 596k freed

Welcome to Embedded Generic Platform.
Press <Enter> to continue.
```



## 致 谢

在论文完成之际，我诚挚地感谢所有帮助过我的人。首先，我要特别感谢我的导师戚正伟，在准备学位论文时间里，戚教授不仅给了我学术上指导和支持，还为我创造了自由宽松的学术氛围和良好的科研环境，指导我参与项目研究，给我提供充足的研究材料，让我能够接触到行业领域的技术动态。在论文撰写期间，戚教授给我提出了许多宝贵的建议，使我能正确把握论文的研究方向，从论文选题到完成，戚教授都给了我悉心深入的指导，使我能够顺利完成研究。戚教授实事求是的治学态度、宽以待人的作风、高度的责任感给了我很大的影响，使我受益终身。

还要提到的是，在完成本文的写作过程中，得到了公司领导和其他部门的鼎力支持，对论文的完成和平台的研发提出了宝贵的意见和大力的支持。特别感谢论文答辩委员会的诸位老师能在百忙之中审阅我的论文，并出席论文答辩。

## 攻读学位期间发表的学术论文目录

- [1]. 万象，戚正伟，UEFI 模块化的 Linux 平台设计与实现[C]，上海交通大学软件学院网上公示，2012