跳过基础知识请选择：
操作系统安装指南
操作系统修复指南

一. MBR、PBR 与 GPT

1. MBR 磁盘管理

这两个都是硬盘地址管理方式（物理层面的管理方式，不同于诸如 fat32、NTFS 等文件系统），可以进行转换。在 windows 中使用 diskpart 命令后输入 list disk 可以看到自己电脑的磁盘管理方式。

MBR（Master Boot Record，主引导记录）是老版本的硬盘管理方式，磁盘的第一个扇区 512B（1B=8bit）用于记录磁盘的全部信息，之后的扇区才进入磁盘分区。第一个分区包括了：

（1） 446B 的 MBR 程序，用于引导系统；

（2） 64B DPT 硬盘分区表，每 16kb 为一组，一共 4 组，用于说明磁盘分区起始位置和结束为止；

（3） 2B magic number，用于表示该硬盘是否可以引导操作系统。

由于 DPT 硬盘分区表的大小限制，MBR 的硬盘格式最多只能有 4 个分区，并且由于 DPT 中每一个记录值的长度限制，因此其记录内容最多只能表示 2TB 的存储空间。由于硬盘生产厂商以 1000 为进制，而计算机以 1024 位进制，因此 2TB 的理论存储空间实际对标的是 2.2TB 的硬盘标称。有些厂商为了打破 2TB 的存储空间限制，将扇区的大小从 512B 升级为 4KB。因此，MBR 能够支持 16 TB。但是，这种方式将导致另一个新问题：如何为具有大块的设备完美地划分磁盘分区。

通过一幅4分区的磁盘结构图可以看到磁盘的大致组织形式。如图5

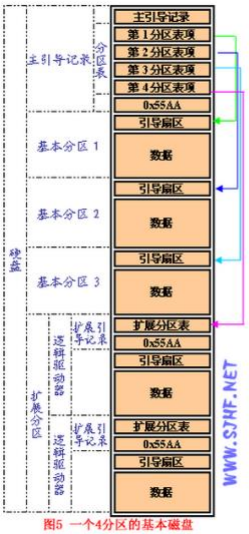


图5 一个4分区的基本磁盘

按照 MBR 的规定，磁盘分区可以有 2 种，一种是主分区（也就是图的基本分区），另一种是扩展分区。主引导记录 MBR 的 446 字节的程序作用就是寻找包含了操作系统的分区，并移交自己的控制权。MBR 如何判断哪一个分区包含操作系统呢？就是靠 DPT 磁盘分区表中的记录，其中包含了操作系统的分区会被标记为主分区，MBR 找到了主分区位置后，将控制权交给主分区的引导扇区 PBR。

同时 MBR 规定，用于引导系统的磁盘中最少也要包含 1 个主分区（不然没法引导操作系统）。但是如果安装了多个系统会怎么办呢？这时候就会有多个主分区，主分区的数量上限为 3，也就是一块 MBR 硬盘理论上限就是安装 3 个系统（为什么 DPT 分区表最多支持 4

个分区，现在规定主分区最多 3 个，后面会说明）。MBR 如何交接控制权呢？这就靠活动记录。被 MBR 直接调用的主分区，在其 PBR 引导扇区中会加入一个特殊的标志，标明是活动扇区。其他的主分区没有这个标志，也就不会被 MBR 调用。多个操作系统，用户如何选择操作系统，会在后面说明。

如果用户想要分 10 个区怎么办？这就要用到扩展分区。MBR 通过 PBT 得知哪一个是扩展分区，而并不对其进行操作。扩展分区的内容完全由操作系统来掌握，因此操作系统可以将这 1 个扩展分区在系统内部划分成 10 个区，在整个硬盘的角度来看，硬盘仍然是 1 个扩展分区，而在操作系统的层面则有 10 个分区，因此也被称之为 LPAR（逻辑分区）。而文件系统的格式，诸如 fat32、NTFS 这些系统，均位于逻辑分区的内部。因此可以将这 10 个分区划分成不同的文件格式。

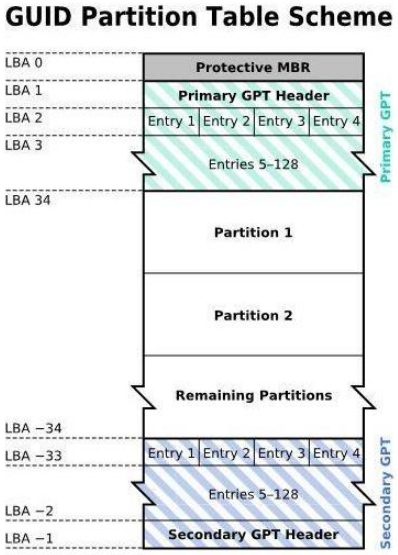
但是，MBR 的标准规定了，一个硬盘的 4 个分区中，最多有 1 个扩展分区。也就是说，当电脑中有上限的 3 个操作系统（即 3 个主分区）时，只能有 1 个操作系统可以拥有 1 扩展分区，可以分出无数个逻辑分区。剩下的操作系统只能有一个分区（也就是只有 1 个 C 盘）。

2. GPT 磁盘管理

GPT（Globally Unique Identifier Partition Table），也叫作 GUID Partition Table，简称 GUID 或者是 GPT，是 MBR 的升级版，现在的磁盘管理方式都是 GPT。GPT 的第一个扇区是 PMBR（Protective MBR），目的是向下兼容。其格式为该扇区的最后 2kb 的 magic number 标明此次盘无法作为启动盘，防止 BIOS 将其识别为启动盘。

如果电脑中存在多个磁盘，那么 BIOS 会正确的找到启动盘并引导操作系统。但是老版本的操作系统（windows XP/2000/NT/9x）由于没有 GPT 磁盘的支持，因此也无法读取 GPT 磁盘的文件。而新版本操作系统（windows server 2003 及以上版本、windows vista/7/8/10）可以使用 MBR 启动方式（向下兼容），同时还能读取非启动盘的 GPT 格式磁盘。

GPT 第一个扇区后，磁盘的第一个正式分区之前，会采用很多额外的扇区来记录 DPT 磁盘分区表（而不是 MBR 的 64kb），因此理论上可以拥有很多的分区。由于 GPT 从理论上可以划分无数个分区的特性，也就不需要使用逻辑分区这个方式来管理分区了。因此 GPT 的磁盘格式，没有主分区、扩展分区、逻辑分区这种说法。所以从理论上来讲也可以安装很多个操作系统。



其中：

（1） LBA0：Protective MBR，上面已经说过

（2） LBA 1：分区表头。定义了硬盘的可用空间以及组成分区表的项的大小和数量。在使用 64 位 Windows Server 2003 的机器上，最多可以创建 128 个分区，即分区表中保留了 128 个项，其中每个都是 128 字节（该值一直为 128，即使分区没有 128 项，也先写入 128）。分区表头还记录了这块硬盘的 UUID（Universally Unique Identifier，即通用唯一识别码，GUID 是微软对 UUID 这个标准的实现。UUID 是由开放软件基金会（OSF）定义的。UUID 还有其它各种实现，不止 GUID 一种），记录了分区表头本身的位置和大小以及备份分区表头和分区表的位置和大小（在硬盘的最后）。它还储存着它本身和分区表的 CRC32 校验。固件、引导程序和操作系统在启动时可以根据这个校验值来判断分区表是否出错，如果出错了，可以使用软件从硬盘最后的备份 GPT 中恢复整个分区表，如果备份 GPT 也校验错误，硬盘将不可使用。一下是分区表头结构的具体信息。

（3） LBA 2~33：LBA 2–33 的位置存放的是分区表项。GPT 分区表使用简单而直接的方式表示分区：

a) 分区类型 GUID，16 字节；

b) 分区唯一的 GUID，16 字节，这个 GUID 指的是该分区本身，而之前的 GUID 指的是该分区的类型，用于标明这个分区是哪一家公司的，用来干什么；

分区类型GUID

操作系统	分区类型	GUID
无	EFI文件系统（标准） [5]	C12A7328-F81F-11D2-BA4B-00A0C93EC93B
Windows	微软保留分区	E3C9E316-0B5C-4DB8-817D-F92DF00215AE
	基本数据分区	EBD0A0A2-B9E5-4433-87C0-68B6B72699C7
	逻辑软盘管理工具元数据分区	5808C8AA-7E8F-42E0-85D2-E1E90434CFB3
	逻辑软盘管理工具数据分区	AF9B60A0-1431-4F62-BC68-3311714A69AD
Linux [4]	数据分区	EBD0A0A2-B9E5-4433-87C0-68B6B72699C7
	RAID分区	A19D880F-05FC-4D3B-A006-743F0F84911E
	交换分区	0657FD6D-A4AB-43C4-84E5-0933C84B4F4F
	逻辑卷管理员（LVM）分区	E6D6D379-F507-44C2-A23C-238F2A3DF928
	保留	8DA63339-0007-60C0-C436-083AC8230908

c) 分区起始和末尾的 64 位 LBA 编号，一共 2*64/8=16 字节；

d) 分区的属性：8 字节；

分区属性

Bit	解释
0	系统分区(磁盘分区工具必须将此分区保持原样，不得做任何修改)
1	EFI隐藏分区(EFI不可见分区)
2	传统的BIOS的可引导分区标志
60	只读
62	隐藏
63	不自动挂载，也就是不自动分配盘符

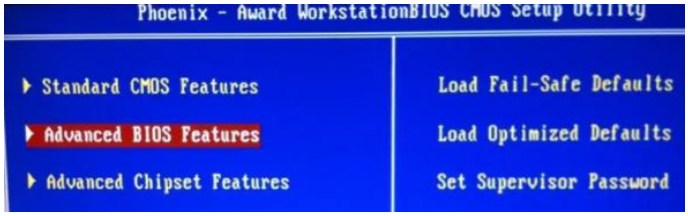
一个 GPT 的磁盘，如果要引导操作系统，至少要有 1 个 EFI 分区，并且 EFI 分区的文件系统是 fat32 格式。EFI 分区是什么，如何引导操作系统，后面会介绍。

e) 分区的名字：最长 72 字节，由用户定义。

（4） 后面的内容为备份。

二. BIOS 与 UEFI 启动方式简单对比

BIOS（Basic Input/Output System，基本输入输出系统）全称是 ROM－BIOS，为了与新版本的作区分，也叫 Legacy BIOS，是开机启动程序。BIOS 是写入到主板上 ROM 中的程序，不可更改，即便断电信息也不会丢失，由 BIOS 厂家制作，世界上只有 AMI、AWARD、Insyde、Byosoft、PHONIX 几大主流厂商。用户配置的 BIOS 数据（比如各种选项，先从哪一个硬盘读取操作系统等）被写在了 COMS 中，是一个 RAM 存储器，数据可更改，断电后信息丢失，由主板上的电池供电。BIOS 采用 16 位汇编语言编写，是电脑启动的时候第一个读取的程序。BIOS 中所有的操作都是以触发各种中断作为基础。下面的是 BIOS 的界面。



UEFI，也叫 UEFI BIOS 是 BIOS 的升级版，现在的电脑都采用 UEFI 启动，在 windows 中使用 msinfo32 命令可以看到 BIOS 模式是否为 UEFI。UEFI 采用 C 语言编写，有更加复杂的功能，支持图形化操作界面（但是进入 BIOS 后是否呈现可以鼠标操作的图形化界面，取决于主板厂商是否编写了图形化界面的程序），甚至可以连接 WIFI。UEFI 中采用的事件 event 轮询的方式，并且可实现异步操作，降低了等待中断的空操作时间，提高了执行效率。用户在 UEFI 启动中设置的参数将会被保存在 NVRAM（Non-Volatile Random Access Memory，非易失性随机访问存储器，U 盘就是采用的这种存储方式）中，即使断电数据也不会消失。在 NVRAM 中保存的参数包括了：各种选项、系统启动顺序、还有 UEFI 独有的启动管理器数据等

BIOS 和 UEFI 启动的流程不同，同时支持的功能也不同，具体区别在后面会介绍。

三. 主板设置中的 Secure Boot 选项、legacy 选项

Secure Boot（微软倡导）是 BIOS 设置中的一项内容，secure boot 的目的是为了防止恶意软件侵入，旨在进入系统的时候通过主板里面内置 windows 的公钥进行校验。但后来微软规定，所有预装 win8 操作系统的厂商都必须打开 secure boot，预装 win8 系统电脑，一旦关闭这个功能，将导致无法进入系统（因为 secure boot 选项是否开启本身也是用户数据，被存储在 NVRAM 中，当然可以被访问），后面允许使用非 secure boot 模式进入系统了。因此在装双系统的时候必须关闭 Secure Boot。

Legacy 兼容模式也是 BIOS 设置中的一项，设置 Legacy 项后 BIOS 将以老版的标准启动，而不是 UEFI，此时系统盘启动方式需要设置为 MBR，否则无法引导系统。用于安装老版本操作系统（Windows Xp、Windows 2003）

四. BIOS 与 UEFI 的启动流程

（1） BIOS（或者 Legacy 模式）的启动流程（重点内容为步骤 C、D）

A. 电脑通电，CPU 中的 CS 寄存器初始化成 0xFFFF，IP 寄存器为 0x0000。自动读取 BIOS 芯片的程序（BIOS 上的 ROM 与内存共用地址空间），然后一条一条的顺次执行 BIOS 的指令；

B. 执行 BIOS 程序中的自检（POST），初始化各种主板芯片组，初始化键盘控制器 8042，初始化中断向量，中断服务例程，初始化 VGA BIOS 控制器，显示 BIOS 的版本和公司名称，扫描软驱和各种介质容量，读取 CMOS 的启动顺序配置（由用户配置），并检测启动装置是否正常，调用 INT 19h 启动自举程序；

C. BIOS 程序进行自举，按照 CMOS 中的启动顺序配置，依次检测磁盘设备，当遇到磁盘的第一个扇区中的最后 2B 的数据 magic number 为 0x55AA 时，表明该磁盘是 MBR 格式磁盘，并且存在系统盘，于是选择该磁盘；

D. 执行中断 INT19 的中断服务例程 INT19_VECT。读取该磁盘第一个扇区的 MBR，将其读入到内存 0000:7C00h，并将控制权交给 MBR。到此 BIOS 程序执行完毕，开始执行 MBR 程序；

MBR 的程序以及之后的操作已经不属于 BIOS 的功能，本文将会在后面讲述操作系统启动的时候继续说明。

（2） BIOS 中断是什么，有什么用？

BIOS 初始化中断控制器、内存后，将中断向量写入内存中，同时提供了几十种中断调用函数，例如 INT 12h 用来获取常规内存容量。这些中断函数为后面的系统引导（MBR 等）、操作系统（DOS 系统等）提供函数功能。可以直接理解为函数调用。

例如 BIOS 将系统控制权交给 MBR、PBR 以及后面的引导程序后，引导程序希望获取内存容量，此时还没有加载文件系统等基本管理系统，只能通过汇编语言语言获取，那么直接调用中断 INT 12h，中断控制器接到中断信号，通知 CPU 中断当前执行的指令，优先执行 INT 12h。然后 CPU 读取中断向量表 12h 的位置，这个地方记录了中断服务程序所在内存的位置。之后 CPU 会执行一段由 BIOS 进行 POST 过程时写入的中断服务程序，通过一系列的汇编语言获取到内存容量，然后返回给当前调用的 BPR。

（3） UEFI 的启动流程

UEFI 的启动流程比 BIOS 复杂的多，分为 7 个阶段。（重点关注步骤 D，UEFI 移交控制权）

A.SEC（Security Phase 安全验证）。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、重启信号等；
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，临时 RAM 只能位于 CPU 内部，这种技术称为 CAR（Cache As Ram）；
3) 作为可信系统的根。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），同时移交控制权；

其流程分为两个阶段：

1）临时 RAM 生效之前称为 Reset Vector 阶段，此时系统还没有 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，Reset Vector 执行流程如下：

2）进入固件入口；

3）从实模式转换到 32 位平坦模式；

4）定位固件中的 BFV（Boot Firmware Volume）；

5）定位 BFV 中的 SEC 映像；

6）若是 64 位系统，从 32 位模式转换到 64 位模式；

7）调用 SEC 入口函数。临时 RAM 生效后调用 SEC 入口函数从而进入 SEC 功能区。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

B. PEI（Pre-EFI Initialization，EFI 前期初始化）。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。功能分析：

1) PEI 内核（PEI Foundation）：负责 PEI 基础服务和流程

2) 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。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。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。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。

执行流程：

1) 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

2) 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。

3) 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。

C. DXE（Driver Execution Environment 驱动执行环境）。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。

从功能上讲，DXE 可分为以下两部分：

1) DXE 内核：负责 DXE 基础服务和执行流程。

2) DXE 派遣器：负责调度执行 DXE 驱动，初始化系统设备。

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

从流程上将，分为以下步骤

1) 根据 HOB 列表初始化服务

2) 遍历固件中的全部驱动 Driver，当 Driver 所依赖的资源全部满足的时候，调度 Driver 到执行队列，直到全部的 Driver 都被加载；

3) 打开 EFI_BDS_ARCH_PORTOCOL ；

4) EFI_BDS_ARCH_PORTOCOL->ENTRY；

D. BDS（Boot Device Selection，启动设备选择）：主要功能是执行启动策略，其主要功能包括：

1) 初始化控制台设备，如键盘、鼠标等等；
2) 读取NVRAM中的非易失变量Bootoption获得启动必要的参数，NVRAM在主板上，并不在硬盘中；
3) 选择启动系统，并移交控制权。选择过程较为复杂，在后面（4）中说明

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。

E. TSL（Transient System Load 操作系统加载前期）：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 阶段称为临时系统，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。

F. RT（Run Time）：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。

G. AL（系统灾难恢复期）：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

可见 UEFI 已经足够复杂，加载全部的基本驱动，

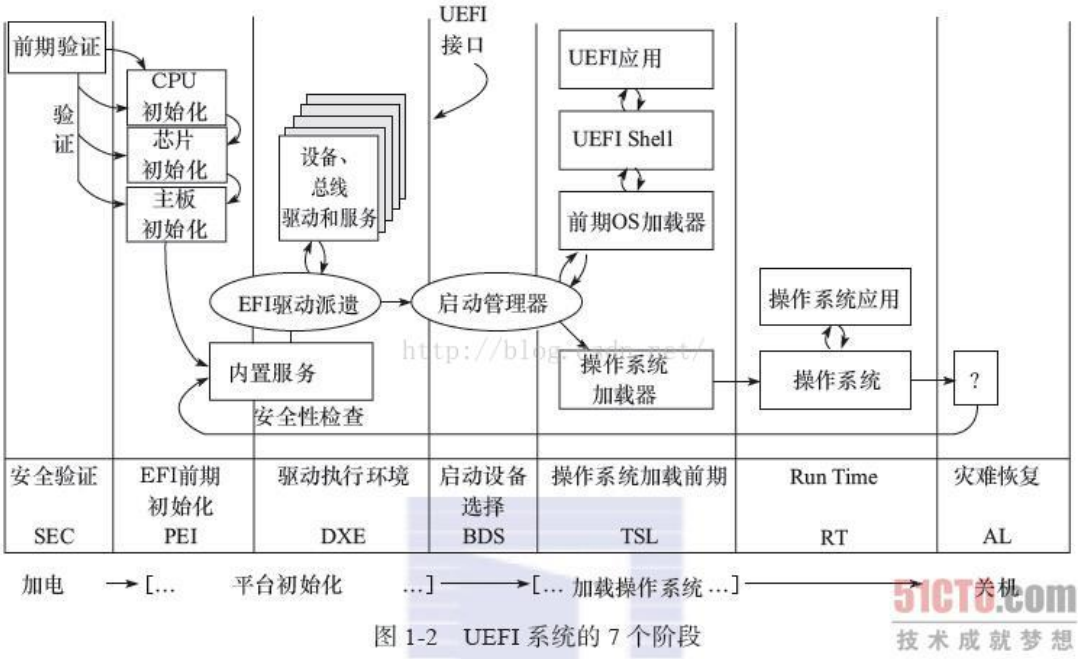
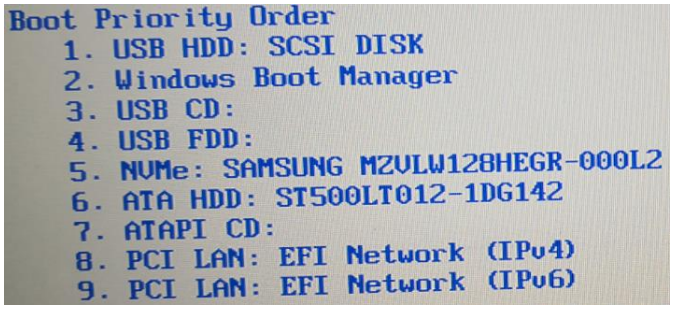


图 1-2 UEFI 系统的 7 个阶段

（4） UEFI 如何移交控制权

A. UEFI 系统根据启动列表的启动顺序逐个对设备进行访问，直到遇到第一个系统启动引导的标志性文件为止。启动列表可能如下图所示：



UEFI 可以选择的启动模式，常用的分两种：

1） 存储设备：硬盘、U 盘等，如上面的 1、3、4、5、6、7、8、9。这些是 UEFI 自动获取的设备接口，并不一定真的存在设备，比如 3 就是空的，只是 U 盘接口存在，但并没有插入 U 盘。

2） 系统启动引导：如上图的 2。这是 UEFI 从 NVRAM 中读取到的系统启动引导数据，由操作系统/第三方软件自动写入

B. 如果是启动项不为空，且为存储设备，那么：

1） UEFI 会根据第一个扇区来判断这个磁盘是否为 GPT 模式的磁盘。如果不是，则继续搜索列表下一项；

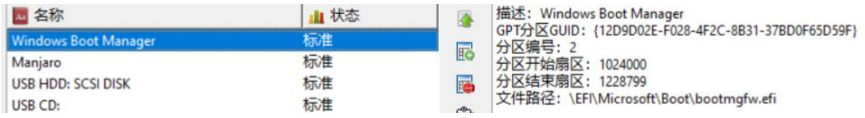
2） UEFI 会从磁盘的 GUID 分区中判断这个磁盘的第一个扇区的分区属性是否为 EFI 系统引导分区（因为 UEFI 只能读取 EFI 分区的内容所以如果 GPT 磁盘作为启动盘，至少需要有一个 EFI 分区才能引导系统），如果不是，则继续搜索列表下一项；

3） UEFI 判断磁盘第一个分区是否为 fat32 文件格式，因为 UEFI 只加载了 fat32 文件系统驱动，因此只能识别文件系统下的文件。如果不是，则继续搜索列表下一项；

4） UEFI 读取这个分区中的 /EFI/Boot/bootx64.efi，如果不存在，则继续搜索列表下一项。如果存在，则执行这个文件。注意此时的控制权还在 UEFI，bootx64.efi 只是被执行（UEFI 只读取.efi 程序并执行）。

C. 如果启动项为系统启动引导数据，例如安装 windows 系统，在 NVRAM 写入的 windows boot manager，那么：

1） 根据这个引导数据的描述，选择对应的磁盘、分区、执行文件的路径。其描述包括了名字、GPT 分区的 GUID 值等，以及执行文件的路径：



Windows 安装的过程中会默认将 efi 路径写入到第一个扇区的 \EFI\Microsoft\Boot\bootmgfw.efi 中

2） 如果上述路径存在，那么执行这个 bootmgfw.efi，否则继续搜索列表下一项；

D. 此时，一旦.efi 程序开始执行，UEFI 已经基本完成任务了，UEFI 为.efi 程序的执行提供了一个微型操作系统的环境，.efi 程序用来引导操作系统。如果遇到不会引导的操作系统，也可以将其转移到另一个.efi 程序去执行。只有当.efi 程序执行 ExitBootServices()的时候才会正式将控制权移交出去。

五. Windows 操作系统启动顺序

前面已经说明了 BIOS 和 UEFI 的启动流程。无论是 BIOS 还是 UEFI，最终的目的都是要将

控制权交给下一个程序。下面就来详细介绍每一种启动流程，这里仅以 windows 为例，linux 的启动过程在后面涉及。这里只说明 3 种引导流程：

A. BIOS + Windows NT 5.X + 引导老版 windows（XP / 2000）
B. BIOS + Windows NT 6.X + 引导新版 Windows（7 / 8 / 10）
C. UEFI + Windows NT 6.X + 引导新版 windows（7 / 8 / 10）

（新版 windows 同时可以使用 BIOS 和 UEFI 引导，相当于向上兼容）

（1） BIOS + Windows NT 5.X 引导 Windows XP / 2000，重点步骤为 A~E

当 BIOS 将控制权交给 MBR 时，执行 Windows NT 5.X 的 MBR 程序：

A. MBR 根据 DPT 中的磁盘分区表，找到活动的分区，把系统控制权交给该分区的 PBR（这里说的是完全交付控制权，因为此时电脑只能执行汇编指令，MBR 通过 JMP 一条跳转指令就完全移交了控制权）。在安装系统时，windows 会把引导用的 PBR 写入主分区中；
B. PBR 的工作是去找同盘符下的 NTLDR（老版本 windows 操作系统加载器）文件，移交控制权，自此 PBR 执行结束。由于此时没有文件系统，PBR 没法加载 NTLDR 路径，不过 windows 将 NTLDR 的路径硬编码到了 PBR 中，因此可以直接跳转；
C. 由于 BIOS 并没有加载软件驱动，因此此时电脑只知道分区，尚不知道每个分区的文件格式。所以 NTLDR 首先会寻找系统自带的一个微型的文件系统驱动，通过文件系统驱动找到硬盘上被格式化为 NTFS 或者 FAT/FAT32 文件系统的分区，并读取其中的 boot.ini 启动配置文件；

```
[boot loader]
timeout=5
default=multi(0)disk(0)rdisk(0)partition(1)\WINDOWS
[operating systems]
multi(0)disk(0)rdisk(0)partition(1)\WINDOWS="Microsoft
Windows" /noexecute=optin /fastdetect
```

Boot.ini 是记事本格式的文件，记录了启动数据，包括了操作系统列表、操作系统所在盘符、操作系统选择时间、以及下一步移交的程序等。

现在执行的 NTLDR 与 boot.ini 已经在活动分区了，证明已经是操作系统所在分区了，为什么 boot.ini 还要描述操作系统所在分区？这是为多个操作系统来考虑的。电脑中可能有多个磁盘，每个磁盘都可能有多个操作系统，通过在 boot.ini 中添加描述，BTLDR 就知道下一步选择哪一个操作系统、以及每个操作系统需要把控制权交给谁。

D. 现在根据 boot.ini 的描述加载操作系统所在盘符，选择操作系统，这一步并非必须的，只有在计算机中安装了多个操作系统的时候才会出现。下图就是经典的 NTLDR 引导界面。

但是，如果电脑安装了 linux 双系统，由于 linux 系统使用的文件系统并非 fat32 或 NTFS，因此如果在描述中引入了 linux 系统的路径，NTLDR 肯定是无法识别的。这时候需要将 linux 的启动文件导出，并将其拷贝到 C 盘中，然后在 boot.ini 中添加与之对应的描述即可。这样 NTLDR 在显示界面的时候就可以正确显示 linux 的引导，当用户选择后，NTLDR 并不会执行默认的 windows 系统操作，而是直接移交控制权。

E. 选择操作系统后，如果是 windows 系统，NTLDR 加载这个操作系统下的 Ntdetect.com，用来收集计算机所有的硬件信息，将其交给 NTLDR，然后开始载入系统内核；
F. Ntldr 会载入 Windows XP 的内核文件 ntoskrnl（windows 内核进程：处理机管理、存储

管理、设备管理、文件管理、接口）但并不执行，随后被载入的是硬件抽象层（hal.dll）。硬件抽象层是内存中运行的一个程序，这个程序在 Windows XP 内核和物理硬件之间起到了桥梁的作用。相当于驱动的作用，提供硬件调用接口，使 windows 可以跨平台调用硬件。

G. Ntldr 载入 HKEY_LOCAL_MACHINE\System 注册表键。Ntldr 会根据载入的 Select 键的内容判断接下来需要载入哪个 Control Set 注册表键，而这些键会决定随后系统将载入哪些设备驱动或者启动哪些服务。这些注册表键的内容被载入后，系统将进入初始化内核阶段，这时候 ntldr 会将系统的控制权交给操作系统内核 ntoskrnl；

H. 操作系统内核正式启动，计算机屏幕上显示 Windows XP 的标志，同时还会显示一条滚动的进度条。在这一阶段中主要会完成这四项任务：创建 Hardware 注册表键、对 Control Set 注册表键进行复制、载入和初始化设备驱动，以及启动服务。

I. 在注册表中创建 Hardware 键，Windows 内核会使用在前面的硬件检测阶段收集到的硬件信息来创建 HKEY_LOCAL_MACHINE\Hardware 键，也就是说，注册表中该键的内容并不是固定的，而是会根据当前系统中的硬件配置情况动态更新；

J. 系统内核将会对 Control Set 键的内容创建一个备份。这个备份将会被用在系统的高级启动菜单中的“最后一次正确配置”选项。例如，如果我们安装了一个新的显卡驱动，重启动系统之后 Hardware 注册表键还没有创建成功系统就已经崩溃了，这时候如果选择“最后一次正确配置”选项，系统将会自动使用上一次的 Control Set 注册表键的备份内容重新生成 Hardware 键，这样就可以撤销掉之前因为安装了新的显卡驱动对系统设置的更改；

K. 载入和初始化设备驱动。在这一阶段里，操作系统内核首先会初始化之前在载入内核阶段载入的底层设备驱动，然后内核会在注册表的 HKEY_LOCAL_MACHINE\System\CurrentControlSet\Services 键下查找所有 Start 键值为“1”的设备驱动。这些设备驱动将会在载入之后立刻进行初始化；

L. 启动服务。系统内核成功载入，并且成功初始化所有底层设备驱动后，会话管理器会开始启动高层子系统和服务，然后启动 Win32 子系统。Win32 子系统的作用是控制所有输入/输出设备以及访问显示设备。当所有这些操作都完成后，Windows 的图形界面就可以显示出来了，同时我们也将可以使用键盘以及其他 I/O 设备；

M. 登录阶段。会话管理器启动的 winlogon.exe 进程将会启动本地安全性授权（Local Security Authority，lsass.exe）子系统。到这一步之后，屏幕上将会显示 Windows XP 的欢迎界面（图 5）或者登录界面。与此同时，系统的启动还没有彻底完成，后台可能仍然在加载一些非关键的设备驱动。

N. 随后系统会再次扫描 HKEY_LOCAL_MACHINE\System\CurrentControlSet\Services 注册表键，并寻找所有 Start 键的数值是“2”或者更大数字的服务。这些服务就是非关键服务，系统直到用户成功登录之后才开始加载这些服务。Strat 键的数值会从小到大一次加载。首先取出全部数值是 2 的服务，进行加载，然后取出所有数值为 3 的服务进行加载，以此类推。

为什么 Windows XP 的启动速度要比 Windows 2000 快？因为 Windows 2000 在系统加载完全部的服务后才显示用户登录界面。而 Windows XP 采用了投机取巧的办法，只加载 start 为 1 的服务，在用户登录后再逐渐加载其他服务。因此 Windows XP 系统启动后仍然要等待很长时间。

概念：系统盘（System Volume）是第一个盘符，保存 ntldr 文件。而引导盘则是 C 盘。当 C

盘是第一个盘符的时候 C 盘既是系统盘又是引导盘。

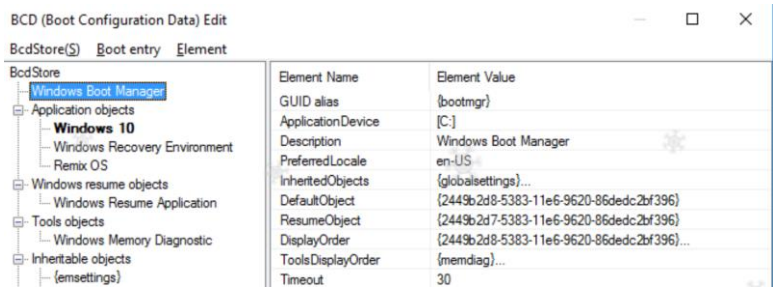
（2） BIOS + Windows NT 6.X 引导 Windows（7 / 8 / 10）系统，重点关注步骤 A~E

如果电脑是老版本的 BIOS 引导方式，想要安装新版本 windows，就需要用 windows NT6.0x 来引导。新版本 Windows 向下兼容，支持老版本的 BIOS 安装 windows。

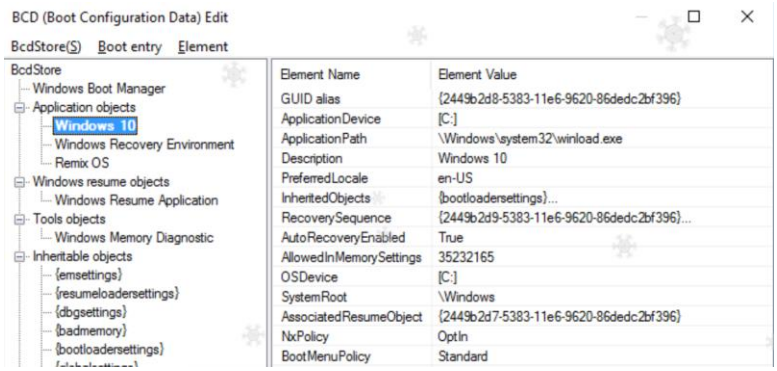
A. 一样的内容，MBR 检查硬盘 DPT 分区表以确定引导分区，之后把系统控制权交给硬盘第一个分区的 PBR；

B. 此时 PBR 不再找 NTLDR，而是找该分区下的启动管理器文件 bootmgr（也叫 Windows Boot Manager），同样采用硬编码形式，直接将 bootmgr 的地址写入了 PBR 中。

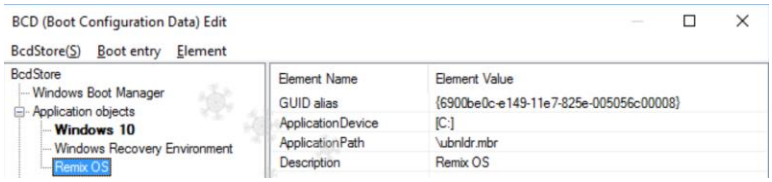
C. bootmgr 中查找同盘符中的 BCD（Boot Configuration Date 系统引导配置数据）文件，这个文件类似于 boot.ini，但并不是文本格式的数据，这个文件实际是一个注册表格式的文件，里面的数据保存了系统的引导信息。使用 bcdedit 命令、BCE Editor、easyBCD 可以查看、编辑其内容。



上图中，BCD 的数据记录了 bootmgr 的位置



上图中，BCD 的数据描述了 windows10 启动项需要加载的 GUID 编号、盘符名称、加载 winload.exe 等信息。



上图中，BCD 的数据描述了第三方操作系统的显示描述与系统加载器 ubnldr.mbr 等信息。

D. BCD 中包含了每一个系统的加载器以及路径，可以添加多种系统启动的入口。当有多个系统的入口的时候，windows 会提供系统选择界面。

E. 在选择操作系统后，Bootmgr 将会按设定启动内核加载程序 Winload.exe（C:\Windows\System32\Boot\winload.exe），并将控制权交给 Winload

F. Winload 加载内核程序（Ntoskrnl.exe）、硬件抽象层（hal.dll）、注册表 SYSTEM 项

（system32\config\system）、设备驱动，然后控制权交给 Ntoskrnl.exe。

G. Ntoskrnl 初始化执行体子系统，并初始化引导的和系统的设备驱动启动程序，为原生应用程序（如 SMSS 等）初始化运行环境，控制权交给 SMSS.exe（Session Manager Subsystem 会话管理子系统）

H. SMSS 初始化注册表，创建系统环境变量，加载 Win32 子系统（Win32k.sys），启动子系统进程（CSRSS、WinInit、Winlogon），控制权交给 WinInit.exe 和 Winlogon.exe

I. WinInit 启动服务控制管理器（SCM），本地安全子系统（LSASS），本地会话管理（LSM）

J. Winlogon 加载登录界面程序（LogonUI），显示交互式登录对话框。等待用户登录后，根据注册表配置启动 UserInit.exe 和 Explorer.exe

K. UserInit 启动用户所有自启动进程，建立网络连接，启动生效的组策略

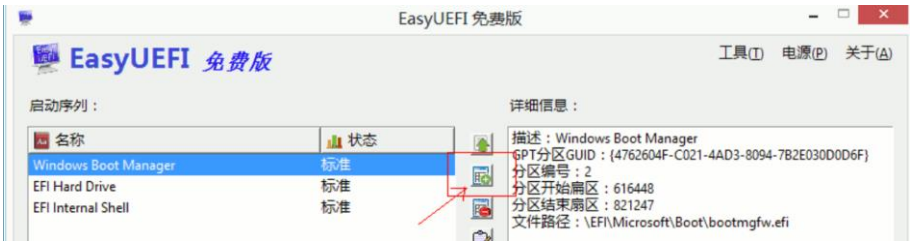
L. xplorer 提供交互式图形界面，包括桌面和文件管理

（3） UEFI 引导 windows（7 / 8 / 10）启动

在 UEFI 安装完操作系统后，Windows 至少使用两个分区，一个叫做 ESP 分区（EFI SYSTEM PARTITION，它只是 Windows 给这个分区起的名字，实际上这个分区在 linux 里面没有名字，因为 linux 是树形目录结构，根本不存在分区名），用于存放启动文件，完全独立于其他分区，另一个则是 BIOS 下正常的系统分区（C 盘）。

UEFI 已经足够复杂，不需要 MBR 和 PBR 这种汇编语言的方式来引导寻找了，UEFI 已经加载了 fat32 文件系统，可以直接寻找 EFI 类型的、fat32 文件系统的磁盘分区了。

在上述的 UEFI 的第四个环节 BDS 阶段，UEFI 会读取 NVRAM 中的非易失变量 Bootoption 获得启动必要的参数。从而会提供可供用户选择的启动顺序、系统引导等操作，这时候 windows 会在上面写入自己的启动引导 windows boot manager，以供用户选择。当选择了这项后，UEFI 会直接读取其写入的磁盘编号、磁盘位置、以及用于引导 windows 的 bootmgfw.efi 的位置。

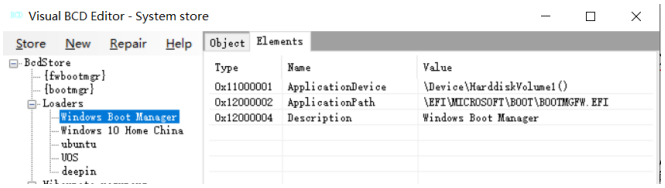


通过 easyUEFI 第三方软件，查看 NVRAM 中的记录数据。可以看到，windows 写入的信息包括了描述信息 windows boot manager，GPT 分区的 GUID、交给 efi 执行的文件路径等信息。easyUEFI、Bootice 等第三方软件同时可以编辑这些项目。

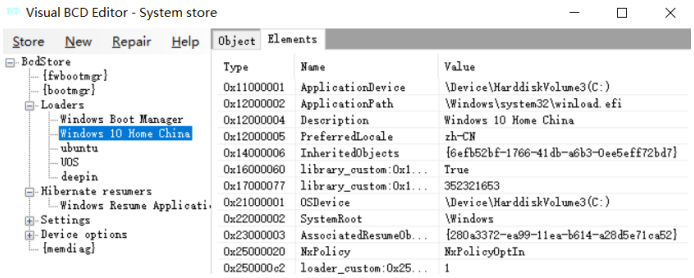
启动流程（重点关注 A~B 步骤）：

A. UEFI 直接将控制权移交给 windows 启动管理器\EFI\Microsoft\Boot\bootmgfw.efi。虽然都是 windows NT 6.x 的引导方式，但是 BIOS 下使用了 bootmgr，而 UEFI 下使用的则是 bootmgfw.efi。

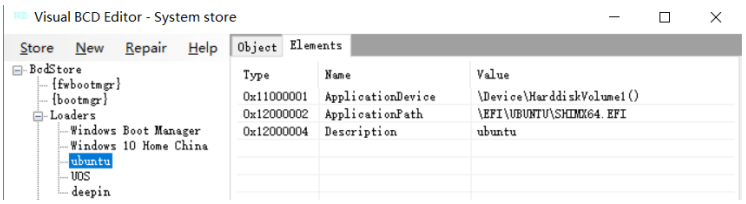
B. bootmgfw.efi 会首先加载 BCD 文件（\EFI\Microsoft\Boot\BCD），此时的 BCD 与作为 BIOS 引导中的 BCD 存储的内容并不一样，由于 BCD 不是文本格式的文件，因此只能借助工具来查看



BCD 的第一项还是自身 bootmgfw.efi



BCD 的第二项是下一个移交控制权的对象，位于 C 盘的 winload.efi。之所以不是 winload.exe，因为之前的 BIOS 模式下，系统控制权完全由 bootmgr 掌握（BIOS 将控制权交给 MBR，MBR 给 PBR，PBR 给 bootmgr），这期间没有加载任何驱动和环境，因此 bootmgr 只是一个汇编程序，可以做任何操作，当然可以直接将控制权交给 winload.exe。但是此时还处于 UEFI 系统控制期间，加载了 UEFI 的环境，只能执行 efi 程序，bootmgfw.efi 除非退出 UEFI 环境，否则只能选择将控制权交个别的 efi 程序（比如下一级的 winload.efi），再由 winload.efi 退出 UEFI 环境，完全接管控制权。



这时电脑中安装的 Ubuntu 系统，其内容已经在安装的时候被 Ubuntu 自动记录在了 windows 的 BCD 中，目的是为了告诉 windows 自己的存在。在描述中写入了 Ubuntu，从理论上来讲，此时 windows 的 bootmgfw.efi 会出现经典的系统选择界面，并且在里面列出来 Ubuntu 选项，当用户选择的时候，bootmgfw.efi 会将控制权交给 SHIMX64.efi（至于后面执行什么操作，在 linux 的模块会讲到）。

但实际上并没有系统选择界面出现，因为 windows 的 bootmgfw.efi 选择性无视所有非 windows 系统，只会加载 windows 的选项。使用之前提到的 easyBCD 工具无效，因为即使加入了 linux 选项，windows 仍然会无视掉，同时 easyBCD 还会警告 UEFI 模式下 windows 禁止了相关功能，无法进行编辑。具体如何操作，后面会有提到。

C. 如果这时找不到 winload.efi，或者 winload.efi 签名校验失败，就会蓝屏 0xc000000e，即找不到引导文件，加载完 winload.efi 后，控制权就正式交给 winload.efi

D. Winload.efi 做的第一件事就是通过 BootService 的 GetMemoryMap 提供的物理内存信息来构造页表以及 PFN 数据库

E. Winload.efi 将 ntoskrnl，hal 以及 SYSTEM\Service 下的所有 Boot 型驱动以及他们需要的导入库读取加载到内存中。因为此时还是保护模式下，所以需要页表中建立这些文件的映射信息。（在 IA32e 下的 UEFI 开机后是会默认开启分页的，但是虚拟内存和物理内存是 1：1 映射的，所以进了 Windows 内核后不可能还用原来的地址）
F. 读取完成后，会对这些文件进行签名校验，如果校验失败，那么会蓝屏 INACCESSIBLE_BOOT_DEVICE，即启动设备无效。
G. 这一步完成后，Windows 会进一步初始化 GDT 和 IDT，然后在页表中分配内核堆栈，初始化 SystemPTE。最后调用 ExitBootService 退出引导阶段，调用 SetVirtualAddress 将 EFI 部分固件内存映射到虚拟内存，然后把页表基址载入 CR3 寄存器，开启分页并跳转到 ntoskrnl 的 KiSytemStartup 进入内核。

六. Linux 系统的引导顺序

Linux 系统于 windows 系统不同，Linux 系统只是一个内核，并没有所谓的系统引导，因此 Linux 使用的是第三方的系统引导。而 windows 则是全家桶，包含了引导、内核、图形化操作界面等。Linux 使用了第三方 GRUB 或者 LILO 作为引导程序，下面仅以最常用的 GRUB 为例。GRUB 功能极强，几乎能引导全部的 X86 平台的操作系统。GRUB 分为 GRUB 1 和 GRUB 2 两个大版本，分别为：

GRUB 1：也叫 GRUB legacy，或者 GRUB，版本为 0.97。grub 是指 grub1.97 和以前的版本。其配置文件为 grub.conf 或 menu.lst，引导程序称之为 stage1、stage1.5 和 stage2。同时提供 grub 命令，即用户可以在进入系统之前使用 GRUB 命令与之交互，用命令的方式进入操作系统等；

GRUB 2：版本号为 1.98 之后的版本叫做 GRUB2，grub2 新增加了很多功能。grub2 配置文件名为 grub.cfg。GRUB2 增添了许多语法，更接近于脚本语言了，但同时也变得极为复杂。例如支持变量、条件判断、循环。GRUB2 提供了一个进入命令行的选项卡，放在操作系统选择界面中。

A. BIOS 启动方式由 GRUB 0.97 引导的 Linux
B. BIOS 启动方式由 GRUB 2.04 引导的 Linux
C. UEFI 启动方式由 GRUB 2.04 引导的 Linux
D. UEFI 启动方式由 rEFInd 引导的 Linux

（1） BIOS 启动方式由 GRUB 0.97 引导的 Linux，重点为 A~C 步骤，E 步骤可忽略

A. GRUB 的第一部分程序，是由 stage1/stage1.S 的源码汇编而来的，写入到了 MBR 的前 446B 中，使用 BIOS 中断 int 13H，用于将第 2 个扇区（0 柱面 0 磁道 2 扇区）处的 512B 载入到内存 0x0000:0x8000 处，然后跳到 0x0000:0x8000 处执行；

Window 中的 MBR 找到的是活动分区，跳转到活动分区的 PBR 执行，而 PBR 位于 DPT 分区表所描述的磁盘分区中的内部空间。而 GRUB 0.97 则直接跳转到了磁盘第 2 个扇区，这个扇区位于 DPT 分区表所描述的磁盘第 1 个正式分区开始之前。

B. 第二个扇区 512B 的内容是固定的，由 stage2/start.S 汇编生成，其作用是判断文件系统类型，并加载相应的驱动。由于此时还没有文件系统，因此第二扇区的内容也是汇编直接写成的。这个文件系统驱动被嵌入在第 3 个分区到第 n 个（n 一般为 62）分区中，同

样位于磁盘第一个正式的分区之前。n 不能太大，否则会占用到第一个磁盘分区的空间。具体 n 为多少，取决于第 2 扇区的程序对于第 3 扇区的文件系统类型的标志判断。

由于 Linux 支持诸如 ext2/3/4、fat32 等多种文件系统格式，因此必须要加载与之对应的文件系统才能匹配。这个步骤叫做 stage1.5，由/boot/grub 目录中的程序汇编而成，stage1.5 的程序有很多，包括了 fat_stage1_5/ e2fs_stage1_5/ xfs_stage_1.5 等等。在安装 Linux 系统的时候，系统根据用户选择的文件系统类型，将不同的 stage1.5 驱动进行编译，然后嵌入到第 2 个扇区到第 n（或者说 62）个扇区之间。这些不同的 stage1.5 其编译结果的前 512B 是完全一样的，即：汇编的结果无论占用多少扇区，其第一个 512B 的内容都和 stage2/start.S 汇编的结果一样。

为什么有 fat_stage1_5 文件系统了，还要 start.S？这两个有什么区别和联系？start.S 编译完成只有 512B，不携带任何文件系统。fat_stage1_5 编译结果的前 512B 就是 start.S 的编译结果，后面则携带了 fat32 文件系统。相当于 start.S 是精简版。当用户只安装 start.S 的情况下，这 512B 的程序会检测到后面没有 stage1.5，则它跳过 stage1.5，直接加载 stage2（2~N 号扇区）到内存 0x0000:0x8200，并跳到 0x0000:0x8200 执行。否则如果检测到了 stage1.5，加载 stage1_5（2~N 号扇区）到内存 0x0000:0x2200，并跳到 0x0000:0x2200 执行。

若加载了 stage1.5，此时已经有了文件系统，不需要使用 BIOS 中断来读取磁盘块了。这时将活动分区中的/boot/grub/stage2 读入内存并执行。如果没有加载 stage1.5，则默认 2~N 号扇区的内容就是 stage2，然后进行执行。

在 windows 中 PBR 分区引导通过硬编码方式直接跳转 NTLDR，并由 NTLDR 加载一个微型文件系统识别 fat32 和 NTFS 文件系统。由于 windows 中的驱动加载过程在 NTLDR 中，不占用分区外的空间，因此可以比较大。但是 GRUB 的文件驱动不能太大，否则会占用到第一个磁盘分区的空间。

C. stage2 的作用是加载各种环境和加载内核。把系统切入保护模式，设置好 c 运行环境然后寻找 menu.lst / grub.conf 配置文件（相当于 windows 中的 boot.ini），如果没有的话就执行一个 shell，等待我们输入命令，并在执行 boot 命令以后就会把控制权转交出去。

menu.lst 和 grub.conf 使用 ln 连接，是一个文件的两个别名。这里不要以为出现了命令行就标明电脑安装错误了，需要重装系统，其实很有可能是 menu.lst 配置错误导致的。menu.lst 位于/boot/grub/menu.lst，使用 vim 编译器进行编写。如果没有了 menu.lst 则可以直接创建，通过 touch /boot/grub/menu.lst。然后 cd /boot/grub 后建立连接 ln -s menu.lst grub.conf。

menu.lst 格式：

```
default=0
timeout=5
#splashimage=(hd0,6)/boot/grub/splash.xpm.gz
hiddenmenu
title Fedora Core (2.6.11-1.1369_FC4)
root (hd0,6)
kernel /boot/vmlinuz-2.6.11-1.1369_FC4 ro root=LABEL=/
initrd /boot/initrd-2.6.11-1.1369_FC4.img
title WinXp
rootnoverify (hd0,0)
chainloader +1
```

default=0 是默认启动哪个系统，从 0 开始；每个操作系统的启动的定义都从 title 开始的，第一个 title 在 GRUB 的启动菜单上显示为 0,第二个启动为 1，以此类推；

timeout=5：表示在开机后，GRUB 画面出现几秒后开始以默认启动；如果在启动时，移动上下键，则解除这一规则；

splashimage=(hd0,6)/boot/grub/splash.xpm.gz ：GRUB 的背景画面，这个是可选项；

hiddenmenu：隐藏 GRUB 的启动菜单，这项也是可选的，也可以用#号注掉；

title XXXXX：title 后面加一个空格，title 是小写的，后面可以自己定义；

root (hd[0-n],y)：设置接下来内容的根分区在磁盘中的位置，然后尝试挂接该分区以得到分区大小。安装 Linux 的时候，大多时候不给 boot 单独设置分区（毕竟 MBR 磁盘格式最多 4 个分区），这时 stage2、menu.lst 与 Linux 启动的 kernel 和 initrd 在同一个分区，不需要切换磁盘查找。如果不在一个分区，则需要这种写法来切换分区位置；

kernel：指定内核及 Linux 的/分区所在位置。因为本例中内核是处在/boot 目录中的，如果/boot 是独立的一个分区，则需要把 boot 省略；

ro：只读；

root=LABEL=/ ：Linux 的根所处的分区；

LABEL=/ ：硬盘分区格式化为相应文件系统后所加的标签；

rootnoverify：设置接下来内容的根分区在磁盘中的位置，但是不尝试测试该分区，因为这是双系统下的 windows 分区，GRUB 无法正确的识别；

chainloader：直接移交控制权；

+1 将控制权移交给上面 rootnoverify 设置的分区的第一个扇区。这时候控制权将会被 windows 系统的 PBR 引导所接管，成功从 GRUB 切换到 windows。

所以，当/和/root 在一个分区时，最简洁的写法如下：

```
default=0
timeout=5
title FC4x
kernel (hd0,6)/boot/vmlinuz-2.6.11-1.1369_FC4 ro root=/dev/hda7
initrd (hd0,6)/boot/initrd-2.6.11-1.1369_FC4.img
```

只有需要加载的内核位置

D. 接下来 GRUB 执行 kernel（也叫 vmlinuz）和 initrd（initial ram disk 初始化的内存盘）。kernel 是 Linux 系统内核，是最核心的部分。initrd 提供开机必需的但 kernel 文件没有提供的驱动模块（modules），同时负责加载硬盘上的根文件系统并执行其中的/sbin/init 程序进而将开机过程持续下去。

GRUB 将 kernle 加载到内存并执行，kernel 在运行的后期会读取并执行 initrd 文件中的 init 脚本文件并按照其中命令逐行执行。

与 kernel 相关的软件包括了：

linux-firmware：kernel 提供的固件程序，位于/lib/firmware 目录下。操作系统用驱动程序驱动硬件，而驱动程序就是和硬件上的固件（如 PCI 设备上的 PCI BIOS）进行交互以控制对应硬件的，所以固件是驱动程序与硬件的执行机构间的桥梁。

linux-headers：与特定的 linux kernel 版本相关的头文件，位于/usr/src/linux-headers-2.6.32-22/include 和/usr/include 目录下，在编译一些 modules 和软件如 virtualbox additional tools 时使用。

linux-image：linux kernel 的二进制压缩文件，即/boot/vmlinu 文件，此文件就是 linux 内核文件；

linux-generic：主要提供相应版本 linux kernel 的 metadata 即更新文件。

在 GRUB 加载完 kernel 后，会继续加载 initrd。GRUB 将它(通常是 initrd.img-xxx...xxx 文件)加载到内存中。内核启动的时候会将这个文件解开，并作为根文件系统使用。设计 initrd 的主要目的是让系统的启动分为两个阶段。首先，带有最少但是必要的驱动（这些驱动是在配置内核时选择嵌入方式，直接编译进了内核中）的内核启动。然后，其它需要的模块将从 initrd 中根据实际需要加载（使用 udev 机制，最重要的根文件系统所在硬盘的控制器接口 module）。这样就可以不必将所有的驱动都编译进内核，而根据实际情况有选择地加载。对于启动较慢的设备，如 usb 设备等，如果将驱动编译进内核，当内核访问其上的文件系统时，通常设备还没有准备好，就会造成访问失败。所以，通常在 initrd 中加载 usb 驱动，然后休眠几秒钟，带设备初始化完成后，再挂载其中的文件系统。

initrd 的具体形式有两种：cpio-initrd 和 image-initrd。image-initrd 的制作相对麻烦，处理流程相对复杂（内核空间->用户空间->内核空间，与“初始化尽量在用户空间进行”的趋势不符），主要是 2.4 及以前的 kernle 使用，本文不对其进行介绍。

cpio-initrd 的处理流程（内核空间->用户空间）

1. GUID 把内核以及 initrd 文件加载到内存的特定位置，然后将控制权交给 kernel；
2. kernel 判断 initrd 的文件格式是否为 cpio 格式；
3. kernel 将 initrd 的内容释放到 rootfs 中；
4. kernel 执行 initrd 中的/init 文件，执行到这一点，内核的工作全部结束，完全交给 init 文件处理。

E. 执行 initrd 中的 init 文件，该 init 文件是一个由 sh 解释并执行的脚本文件，内核通过文件头来确定应该怎样执行。

```
[ -d /dev ] || mkdir -m 0755 /dev
[ -d /root ] || mkdir -m 0700 /root   #这是硬盘上的根分区预先挂载到的目录
[ -d /sys ] || mkdir /sys
[ -d /proc ] || mkdir /proc
[ -d /tmp ] || mkdir /tmp
mkdir -p /var/lock
mount -t sysfs -o nodev,noexec,nosuid none /sys   #udev会参考的vfs，udev会根据其中的信息加载
modules和创建设备文件
mount -t proc -o nodev,noexec,nosuid none /proc
```

init 文件建立了相关目录和挂载点，并将 kernel 运行过程中产生的信息挂载到/sys 和/proc 目录下。注意/sys 目录是 udev 会参考的 vfs，udev 会根据其中的信息加载 modules 和创建设备文件，当不使用 udev 机制（后面会讲）时/sys 目录可以不建立。/proc 目录和相应的 proc 文件系统必须建立和挂载，因为脚本会参考其中的/proc/cmdline 文件获得 kernel 命令行上的参数。

```
grep -q '/<quiet/>' /proc/cmdline || echo "Loading, please wait..."
```

如果在 GRUB 的 kernel 行上有 quiet 关键字，则在 kernel 启动和 initrd 中 init 脚本执行的过程中不会在屏幕上显示相关信息而是一个闪烁的下划线，否则将显示"Loading, please wait..."

```
# Note that this only becomes /dev on the real filesystem if udev's scripts
# are used; which they will be, but it's worth pointing out
if ! mount -t devtmpfs -o mode=0755 none /dev; then  #其实then的代码一般不会执行
    mount -t tmpfs -o mode=0755 none /dev
    mknod -m 0600 /dev/console c 5 1
    mknod /dev/null c 1 3
fi
```

这部分在/dev 目录下建立 devtmpfs 文件系统，devtmpfs 是一个虚拟的文件系统，被挂载后会自动在/dev 目录生成很多常见的设备节点文件（设备驱动），当挂载 devtmpfs 失败时会手动建立/dev/console 和/dev/null 设备节点文件。/dev/console 总是代表当前终端，用于输出 kernel 启动时的输出内容，在最后通过 exec 命令用指定程序替换当前 shell 时使用。/dev/null 也是很常用的，凡是重定向到它的数据都将消失得无影无踪。

```
mkdir /dev/pts #主要用于nfs等启动时使用，对于本地/dev/pts不使用，故下面一段代码可忽略
mount -t devpts -o noexec,nosuid,gid=5,mode=0620 none /dev/pts || true
> /dev/.initramfs-tools
mkdir /dev/.initramfs
```

usplash 会使用/dev/.initramfs 目录。usplash 会在机器启动的时候提供类似 windows 的启动画面，ubuntu linux 的启动画面就是通过 usplash 实现的。由于在/sbin 目录当中没有任何 usplash 相关的文件，可以忽略这个目录的存在。

```
# Export the dpkg architecture
export DPKG_ARCH=
. /conf/arch.conf
```

DPKG_ARCH 表明了当前运行 linux 的计算机的类型，对一般的 pc 是大多 i386，也可能是别的比如 powerpc 一类的。用 export 是为了让这个变量不仅在此 shell 环境中有效，而且在它的子 shell 环境中仍然有效。而且在 27 行 export DPKG_ARCH 变量的时候，让 DPKG_ARCH 变量等于空。这样，当前运行的计算机的类型就完全由/conf/arch.conf 决定了

```
# Set modprobe env
export MODPROBE_OPTIONS="-qb"
```

设置 modprobe 默认的选项，-b

表示用 use-blacklist(主要是系统的硬件有多个 modules 支持，选择使用哪一个，其他的加入到黑名单中以防冲突) ，-q 表示 quiet

```
# Export relevant variables
export ROOT=      #从kernel中提取的realfs所在的设备
export ROOTDELAY=
export ROOTFLAGS=
export ROOTFSTYPE=
export IPOPTS=
export HWADDR=
export break=
```

输出一些变量到环境中：

ROOT：保存 GRUB 的 kernel 命令行上的 root 参数即硬盘上根分区所对应的设备节点文件；

ROOTDELAY：指定将要进入的系统的根目录所在的分区必须在多少秒之内准备好

ROOTFLAGS：指定将要进入的系统的根目录所在的分区挂载到 ${rootmnt}目录时的参数

ROOTFSTYPE：指定根分区所在的文件系统类型

IPOPTS：当 kernel 为 nfs 时指定的服务器 IP

HWADDR：当 kernel 为 nfs 时指定的服务器 MAC

Break：由 maybe_break 函数使用。若 break 的值同 maybe_break 的第一个参数相同，则 maybe_break 函数调用 panic 函数（注意 panic 函数和 panic 变量是不同的）。若 panic 变量为"0"（此处是字符串，其内容是"0"，不是整数），则 panic 函数将重新启动

机器。其他情况下（包括 panic 变量为空的情况）都将以交互的方式调出 shell，此 shell 的输入输出使用已经创建好的节点 /dev/console。

```
export init=/sbin/init    #realfs中的第一个执行的程序位置
export quiet=n
export readonly=y
export rootmnt=/root      #realfs在rootfs中的临时挂载点
export debug=             #将本脚本的输出定向到一个文件，以便启动系统后分析
export panic=
export blacklist=         #设置modules的黑名单
export resume_offset=
```

init：指定硬盘 realfs 中的第一个执行的程序位置，此变量指定在这个脚本最后要执行的进程。此处/sbin/init 是系统上所有进程的父进程，负责开启其它进程。当然，你也可以把它换成其他的程序，甚至是 ls，不一定非要是/sbin/init，虽然这样你的系统启动之后什么都不能做。

quiet=n：指定为非"y"，会显示一些启动的状态信息；若指定为"y"则不显示这些信息。

readonly=y：如果 readonly 等于字符串"y"，则以只读方式挂载最终要进入的系统的根目录所在分区到${rootmnt}目录，其他情况(包括 readonly 为空)以读写方式挂载。

rootmnt=/root：指定硬盘上的 realfs 在 rootfs 中的临时挂载点

debug：将本脚本的输出定向到一个文件，以便启动系统后分析

panic：描述见 break 参数的说明。

blacklist：设置 modules 的黑名单

resume_offset：一般用不到

```
# Bring in the main config
. /conf/initramfs.conf    #最重要的是BOOT参数，定义了是本地启动还是nfs启动
for conf in conf/conf.d/*; do  #对于不支持kernel命令行选项的bootloader很有用
   [ -f ${conf} ] && . ${conf}
done
. /scripts/functions    #这个脚本文件很重要，在其中定义了很多以后要用到的工具函数
```

在当前 shell 中引入主配置文件/conf/initramfs.conf。这个配置文件实际上是 mkinitramfs(8)的配置文件，其中定义了一些变量，并赋予了适当的值，如 BOOT=local 则默认从本地磁盘启动（可以是可移动磁盘）。BOOT 变量的值实际上是/scripts 目录下的一个文件，可以是 local 或是 nfs。在此 init 脚本挂载将要进入的系统的根目录所在分区的时候，会先读取并运行/scripts/${BOOT}文件。在这个文件中定义了 mountroot 函数，对于 local 启动和 nfs 启动此函数的实现不同。这样通过对不同情况引入不同的文件，来达到同样名称的函数行为不同的目的。这就导致了具体挂载的行为和启动方式相关。

引入/conf/conf.d 下的所有文件，注意在引入的时候用-f 参数判断，这样只有普通的文件才会被引入，链接(硬链接除外)、目录之类的非普通文件不会被引入，当使用不支持命令行参数的开机引导程序时，可以在该目录下建立各种参数设置文件。（Ubuntu 使用的 GRUB 支持 kernel 命令行参数，所以这个目录下就上面提到的两个文件 initramfs.conf 和 arch.conf）

```
# Parse command line options
for x in $(cat /proc/cmdline); do   #为export的变量赋值，最重要的是root
   case $x in
   init=*)              #指定切换到磁盘上的rootfs上时执行的第一个程序，默认为/sbin/init
```

```
        …此处省略几百行代码
      netconsole=*)
        netconsole=${x#netconsole=}
        ;;
      esac
    done
```

以上的位于 for 循环中的脚本的主要功能是获取 kernel 的命令行选项（/proc/cmdline）然后赋给相应的变量。其中最重要的是 root，其它的可有可无。

```
if [ -z "${noresume}" ]; then
   export resume=${RESUME}
else
   export noresume
fi
```

若 NORESUME 变量不为空，则将 RESUME 变量的值赋给 resume 变量，并把 resume 变量 export 出去，使得在子 shell 环境中也可以使用 resume 变量。resume 变量将在/scripts/local-premount/resume 脚本中使用。

```
[ -n "${netconsole}" ] && modprobe netconsole netconsole=${netconsole}
基于网络的终端控制，一般用不到。

maybe_break top
```

参见上文中对 break 变量的解释，kernel 中指定的 break 参数包含 top 时，maybe_break 会查看 panic 变量是否为 0，若否则以交互的方式调出 shell，此 shell 的输入输出使用已经创建好的节点/dev/console。

```
# export BOOT variable value for compcache,
# so we know if we run from casper
export BOOT


# Don't do log messages here to avoid confusing usplash
run_scripts /scripts/init-top
```

按照/scripts/init-top/ORDER 文件的配置依次执行其下的脚本文件，这里最重要的是开启了 udev daemon

udev 以 daemon 的方式启动 udevd，接着执行 udevtrigger 触发在机器启动前已经接入系统的设备的 uevent，然后调用 udevsettle 等待，直到当前 events 都被处理完毕。之后，如果 ROOTDELAY 变量不为空，就 sleep ROOTDELAY 秒以等待 usb/firewire disks 准备好。

```
maybe_break mount              #这一步主要是执行/scripts/local中的mountroot函数，将realfs挂载到
${rootmnt},在mountroot函数中最重要的是检测realfs的fytype
log_begin_msg "Mounting root file system..."
. /scripts/${BOOT}
parse_numeric ${ROOT}          #这一步主要用于解析lilo启动管理程序传递的参数，对于GRUB用不
到。
mountroot
log_end_msg
```

这一部分的功能主要是把硬盘上的 realfs 挂载到${rootmnt}，特别重要，大部分的系统启动问题都是这一部分存在问题引起的。其中很大一部分的原因是硬盘控制器的 module 没有被 udev 或上面的 load_modules 加载，导致 kernel 不能读取硬盘中的数据。

mountroot 定义在/scripts/local 中

mountroot 的详细解释详见 https://blog.csdn.net/geekard/article/details/6455502

```
maybe_break bottom              #/scripts/init-bottom中的主要文件是udev，这一步要停止udev服务并
执行mount -n -o move /dev ${rootmnt}/dev
[ "$quiet" != "y" ] && log_begin_msg "Running /scripts/init-bottom"
run_scripts /scripts/init-bottom
[ "$quiet" != "y" ] && log_end_msg
```

这一部分的主要功能是按照/scripts/init-buttom/ORDER 文件的配置依次执行其下的脚本文件，这里最重要的是关闭了 udev daemon

```
# Move virtual filesystems over to the real filesystem
mount -n -o move /sys ${rootmnt}/sys
mount -n -o move /proc ${rootmnt}/proc
```

将开机过程中 kernel 产生的信息转移到硬盘上 rootfs 对应的目录中。

```
# Check init bootarg
if [ -n "${init}" ] && [ ! -x "${rootmnt}${init}" ]; then
    echo "Target filesystem doesn't have ${init}."
    init=
fi
```

查看 init 变量中指定的文件能否执行，一般是/sbin/init

```
# Search for valid init
if [ -z "${init}" ] ; then
   for init in /sbin/init /etc/init /bin/init /bin/sh; do
      if [ ! -x "${rootmnt}${init}" ]; then
         continue
      fi
      break
   done
fi
```

当 init 变量为空时(一般都是这样)，查找可以找到并使用的 init 程序

```
# No init on rootmount
if [ ! -x "${rootmnt}${init}" ]; then
   panic "No init found. Try passing init= bootarg."
fi
```

找不到可以使用的 init 程序时，kernel panic。

```
# Confuses /etc/init.d/rc
if [ -n ${debug} ]; then
   unset debug
fi
```

因为在最终要使用的系统的 /etc/init.d/rc 中通过 debug 变量来显示要执行的一些命令，其中 debug=echo 那一行是注释掉的。所以这里要 unset debug 变量，否则;/etc/init.d/rc 的执行会出问题。

```
# Chain to real filesystem
maybe_break init
exec run-init ${rootmnt} ${init} "$@" <${rootmnt}/dev/console >${rootmnt}/dev/console 2>&1
panic "Could not execute run-init."
```

这一段代码是这个 init 脚本的最后部分，把系统的启动交给了将要进入的系统的

${init} (上面初始化为 "/sbin/init")，并用/dev/console 作为输入与输出的设备。

（2） BIOS 启动方式由 GRUB 2.04 引导的 Linux

grub2 使用 img 文件，不再使用 grub 中的 stage1、stage1.5 和 stage2，img 文件就是 grub2 的引导文件。在 grub2 软件安装完后，会在/usr/lib/grub/i386-pc/目录下生成很多模块文件和 img 文件，还包括一些 lst 列表文件。

```
[root@server7 ~]# ls -lh /usr/lib/grub/i386-pc/*.lst
-rw-r--r--. 1 root root 3.7K Nov 24  2015 /usr/lib/grub/i386-pc/command.lst
-rw-r--r--. 1 root root  936 Nov 24  2015 /usr/lib/grub/i386-pc/crypto.lst
-rw-r--r--. 1 root root  214 Nov 24  2015 /usr/lib/grub/i386-pc/fs.lst
-rw-r--r--. 1 root root 5.1K Nov 24  2015 /usr/lib/grub/i386-pc/moddep.lst
-rw-r--r--. 1 root root  111 Nov 24  2015 /usr/lib/grub/i386-pc/partmap.lst
-rw-r--r--. 1 root root   17 Nov 24  2015 /usr/lib/grub/i386-pc/parttool.lst
-rw-r--r--. 1 root root  202 Nov 24  2015 /usr/lib/grub/i386-pc/terminal.lst
-rw-r--r--. 1 root root   33 Nov 24  2015 /usr/lib/grub/i386-pc/video.lst

[root@server7 ~]# ls -lh /usr/lib/grub/i386-pc/*.img
-rw-r--r--. 1 root root  512 Nov 24  2015 /usr/lib/grub/i386-pc/boot_hybrid.img
-rw-r--r--. 1 root root  512 Nov 24  2015 /usr/lib/grub/i386-pc/boot.img
-rw-r--r--. 1 root root 2.0K Nov 24  2015 /usr/lib/grub/i386-pc/cdboot.img
-rw-r--r--. 1 root root  512 Nov 24  2015 /usr/lib/grub/i386-pc/diskboot.img
-rw-r--r--. 1 root root  28K Nov 24  2015 /usr/lib/grub/i386-pc/kernel.img
-rw-r--r--. 1 root root 1.0K Nov 24  2015 /usr/lib/grub/i386-pc/lnxboot.img
-rw-r--r--. 1 root root 2.9K Nov 24  2015 /usr/lib/grub/i386-pc/lzma_decompress.img
-rw-r--r--. 1 root root 1.0K Nov 24  2015 /usr/lib/grub/i386-pc/pxeboot.img
```

grub2 将 boot.img 转换后的内容安装到 MBR 中的 boot loader 部分，将 diskboot.img 和 kernel.img 结合成为 core.img，同时还会嵌入一些模块或加载模块的代码到 core.img 中，然后将 core.img 转换后的内容安装到磁盘的指定位置处。

同时，不同有 GRUB1 的只有 1 中安装方式，GURB2 有两种方式安装：

1. 嵌入到 MBR 和第一个分区中间的空间，这部分就是大众所称的"boot track"，"MBR gap"或"embedding area"，它们大致需要 31kB 的空间。这种方式与安装 GRUB1 的原理一致；
2. 将 core.img 安装到某个文件系统中，然后使用分区的第一个扇区（严格地说不是第一个扇区，而是第一个 block）存储启动它的代码。这种方式和 windows 的 PBR 引导出 NTLDR 类似。

这两种方法有不同的问题：

1. 方法 1 使用嵌入的方式安装 grub，就没有保留的空闲空间来保证安全性，例如有些专门的软件就是使用这段空间来实现许可限制的；另外分区的时候，虽然会在 MBR 和第一个分区中间留下空闲空间，但可能留下的空间会比这更小。
2. 方法 2 安装 grub 到文件系统，但这样的 grub 是脆弱的。例如，文件系统的某些特性需要做尾部包装，甚至某些 fsck 检测，它们可能会移动这些 block。那么通过硬编码写入到 PBR 中的引导程序将无法找到 core.img。

GRUB 开发团队建议将 GRUB 嵌入到 MBR 和第一个分区之间，除非有特殊需求，但仍必须要保证第一个分区至少是从第 31kB(第 63 个扇区)之后才开始创建的。不过现在的磁盘设备，一般都会有分区边界对齐的性能优化提醒，所以第一个分区可能会自动从第 1MB 处开始创建。

GRUB2 还有强大的功能，能够让 BIOS 模式的电脑启动 GPT 格式的磁盘的系统，具体内容不做介绍。

启动流程：

A. boot.img 是 grub 启动的第一个 img 文件，它被写入到 MBR 中或分区的 boot sector 中，

因为 boot sector 的大小是 512 字节，所以该 img 文件的大小也是 512 字节。

B. MBR 与分区 boot sector 中的 boot.img 唯一的作用是读取属于 core.img 的第一个扇区并跳转到它身上，将控制权交给该扇区的 img。由于体积大小的限制，boot.img 无法理解文件系统的结构，因此 grub2-install 将会把 core.img 的位置硬编码到 boot.img 中，这样就一定能找到 core.img 的位置。这种寻找方式和 windows 中的 PBR 找到 NTLDR 一样。

C. core.img 根据 diskboot.img、kernel.img 和一系列的模块被 grub2-mkimage 程序动态创建，并编译到了一起，形成了 core.img 一个程序。core.img 中嵌入了足够多的功能模块以保证 grub 能访问/boot/grub，并且可以加载相关的模块实现相关的功能，例如加载启动菜单、加载目标操作系统的信息等，由于 grub2 大量使用了动态功能模块，使得 core.img 体积变得足够小。core.img 中包含了多个 img 文件的内容，包括 diskboot.img/kernel.img 等。core.img 的安装位置随 MBR 磁盘和 GPT 磁盘而不同，这在上文中已经说明过了。

因为上面的性质，所以安装 grub2 的过程大体分两步：一是根据/usr/lib/grub/i386-pc/目录下的文件生成 core.img，并拷贝 boot.img 和 core.img 涉及的某些模块文件到/boot/grub2/i386-pc/目录下；二是根据/boot/grub2/i386-pc 目录下的文件向磁盘上写 boot loader。

D. 如果启动设备是硬盘，即从硬盘启动时，core.img 中的第一个扇区的内容就是 diskboot.img。diskboot.img 的作用是读取 core.img 中剩余的部分到内存中，并将控制权交给 kernel.img，由于此时还不识别文件系统，所以将 core.img 的全部位置以 block 列表的方式编码，使得 diskboot.img 能够找到剩余的内容。该 img 文件因为占用一个扇区，所以体积为 512 字节。

1) cdboot.img：如果启动设备是光驱(cd-rom)，即从光驱启动时，core.img 中的第一个扇区的的内容就是 cdboo.img。它的作用和 diskboot.img 是一样的。
2) pexboot.img：如果是从网络的 PXE 环境启动，core.img 中的第一个扇区的内容就是 pxeboot.img。
3) kernel.img：kernel.img 文件包含了 grub 的基本运行时环境：设备框架、文件句柄、环境变量、救援模式下的命令行解析器等等。很少直接使用它，因为它们已经整个嵌入到了 core.img 中了。注意，kernel.img 是 grub 的 kernel，和操作系统的内核无关。如果细心的话，会发现 kernel.img 本身就占用 28KB 空间，但嵌入到了 core.img 中后，core.img 文件才只有 26KB 大小。这是因为 core.img 中的 kernel.img 是被压缩过的。
4) lnxboot.img：该 img 文件放在 core.img 的最前部位，使得 grub 像是 linux 的内核一样，这样 core.img 就可以被 LILO 的"image="识别。当然，这是配合 LILO 来使用的，现在已经不用 LILO 了。
5) *.mod：各种功能模块，部分模块已经嵌入到 core.img 中，或者会被 grub 自动加载，但有时也需要使用 insmod 命令手动加载

E. 当 core.img 加载完驱动后，会读取配置文件/boot/grub2/grub.cfg，该配置文件的写法弹性非常大，但绝大多数需要修改该配置文件时，都只需修改其中一小部分内容就可以达成目标。

grub2-mkconfig 程序可用来生成符合绝大多数情况的 grub.cfg 文件，默认它会自动尝试探测磁盘中所有有效的操作系统内核，并生成对应的操作系统菜单项。使用方法非常简单，只需一个选项"-o"指定输出文件即可。

```
shell> grub2-mkconfig -o /boot/grub2/grub.cfg
```

grub.cfg 配置内容如下，可以设置的全局变量极多，下面仅挑选部分作说明。同时 grub.cfg 支持的语法也极为丰富

```
[root@xuexi ~]# cat /etc/default/grub
GRUB_TIMEOUT=5
GRUB_DISTRIBUTOR="$(sed 's, release.*,g' /etc/system-release)"
GRUB_DEFAULT=saved
GRUB_DISABLE_SUBMENU=true
GRUB_TERMINAL_OUTPUT="console"
GRUB_CMDLINE_LINUX="crashkernel=auto biosdevname=0 net.ifnames=0 rhgb quiet"
GRUB_DISABLE_RECOVERY="true"
```

1) GRUB_DEFAULT：默认的菜单项，默认值为 0。其值可为数值 N，表示从 0 开始计算的第 N 项是默认菜单，也可以指定对应的 title 表示该项为默认的菜单项。
2) GRUB_TIMEOUT：在开机选择菜单项的超时时间，超过该时间将使用默认的菜单项来引导对应的操作系统，默认值为 5 秒。
3) GRUB_DISTRIBUTOR：设置发行版的标识名称，一般该名称用来作为菜单的一部分，以便区分不同的操作系统。
4) GRUB_CMDLINE_LINUX：添加到菜单中的内核启动参数。例如：
   GRUB_CMDLINE_LINUX="crashkernel=ro root=/dev/sda3 biosdevname=0 net.ifnames=0 rhgb quiet"
5) GRUB_DISABLE_LINUX_UUID：默认情况下，grub2-mkconfig 在生产菜单项的时候将使用 uuid 来标识 Linux 内核的根文件系统，即"root=UUID=…"。

```
linux16 /vmlinuz-3.10.0-327.el7.x86_64
        root=UUID=b2a70faf-aea4-4d8e-8be8-c7109ac9c8b8 ro
        crashkernel=auto biosdevname=0 net.ifnames=0 quiet LANG=en_US.UTF-8
initrd16 /initramfs-3.10.0-327.el7.x86_64.img
```

6) GRUB_DISABLE_OS_PROBER：默认情况下，grub2-mkconfig 会尝试使用 os-prober 程序(如果已经安装的话，默认应该都装了)探测其他可用的操作系统内核，并为其生成对应的启动菜单项。设置为"true"将禁用自动探测功能。
7) GRUB_DISABLE_SUBMENU：默认情况下，grub2-mkconfig 如果发现有多个同版本的或低版本的内核时，将只为最高版本的内核生成顶级菜单，其他所有的低版本内核菜单都放入子菜单中，设置为"y"将全部生成为顶级菜单。

例如在 msdos 磁盘上安装了两个操作系统，CentOS 7 和 CentOS 6：

```
# 设置一些全局环境变量
set default=0
set fallback=1
set timeout=3

# 将可能使用到的模块一次性装载完
# 支持msdos的模块
insmod part_msdos
# 支持各种文件系统的模块
insmod exfat
insmod ext2
insmod xfs
insmod fat
insmod iso9660

# 定义菜单
menuentry 'CentOS 7' --unrestricted {
        search --no-floppy --fs-uuid --set=root 367d6a77-033b-4037-bbcb-416705ead095
        linux16 /vmlinuz-3.10.0-327.el7.x86_64 root=UUID=b2a70faf-aea4-4d8e-8be8-c7109ac9c8b8 ro biosdevname=0 net.ifnames=0 quiet
        initrd16 /initramfs-3.10.0-327.el7.x86_64.img
}
menuentry 'CentOS 6' --unrestricted {
        search --no-floppy --fs-uuid --set=root f5d8939c-4a04-4f47-a1bc-1b8cbabc4d32
        linux16 /vmlinuz-2.6.32-504.el6.x86_64 root=UUID=edb1bf15-9590-4195-aa11-6dac45c7f6f3 ro quiet
        initrd16 /initramfs-2.6.32-504.el6.x86_64.img
}
```

linux16：表示以传统的 16 位启动协议启动内核，linux 表示以 32 位启动协议启动内核，但 linux 命令比 linux16 有一些限制。但绝大多数时候，它们是可以通用的。

F. 当用户选择了启动项后，GRUB2 会以和 GRUB1 类似的方式加载操作系统内核文件，并完成系统启动。

（3） UEFI 启动方式由 GRUB 2.04 引导的 Linux

根据 windows 中使用 UEFI 引导的方式，可以得知 UEFI 启动方式，大部分的驱动都已经在 UEFI 中加载完毕了，因此只需要编写.efi 文件即可，因此 GRUB2 的 UEFI 引导方式与 windows 极为类似。

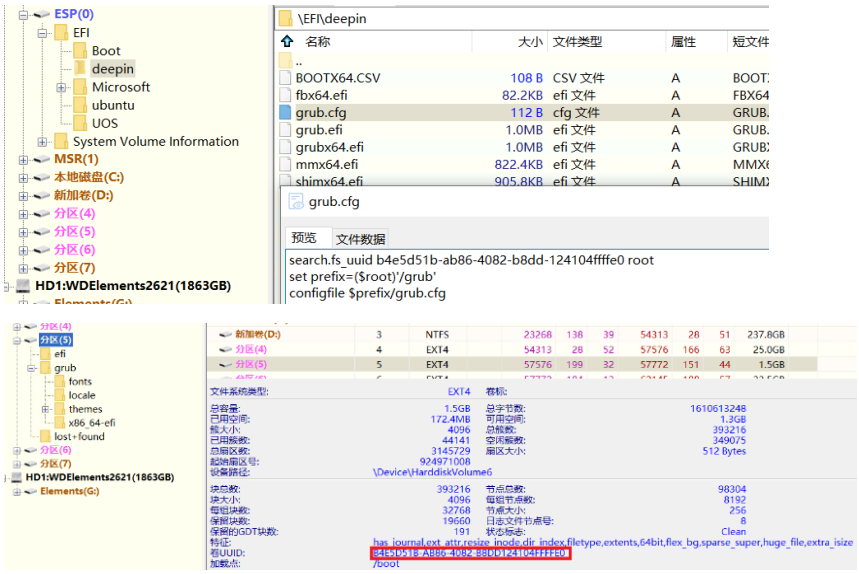
启动流程：

A. 安装 Linux 系统的时候，默认将 Linux 的启动项写入了 UEFI 中的 NVRAM 里，当用户选择此项，会执行 NVRAM 中写入的.efi 程序路径。默认为 /EFI/Boot/bootx64.efi。

安装 Linux 系统时，需要选择安装盘符。一种是直接安装在整个磁盘，另一种为安装在用户指定盘符。这时候需要涉及到挂载点问题。因为 Linux 是树形文件结构，每一个目录可以选择安装不同的盘符。按照 UEFI 的要求，UEFI 启动阶段，只加载了 fat32 文件系统，因此 Linux 必须选择一个 fat32 的分区去挂载 GRUB2，此挂载点为 /boot。/swap 为交换分区，与内存共同组成虚拟内存，不设置/swap 的情况下内存超出上限可能会导致 Linux 无法分配过多内存（在 windows 中叫做虚拟内存空间，存储与 C 盘，当然这个叫法并不准确，因为内存+交换分区=虚拟内存）。当然电脑 16g 内存的情况下不设置/swap 分区也无妨。剩下的挂载点 / 可以全部选择另一个分区，并将其格式化为 ext4 文件系统。

在安装 Linux 时，Linux 会检测到当前安装的 windows 操作系统，因此会将其启动信息写入到 windows 的 BCD 中，然而 windows 会无视 Linux 启动项，这一点前面已经提到。

B. 此时电脑中将会有两个 EFI 分区，一个是 windows 自带的，一个是 Linux 的。Linux 会在 windows 的 EFI 分区中添加自己的项目 grub.efi，如下图所示。



打开其中的配置 grub.cfg 可以看到，这个 grub.efi 通过分区的 UUID 先去寻找 Linux 的 EFI 分区，然后找到配置文件为这个分区中的/grub/grub.cfg，并读取后执行。

C. Linux 的 EFI 分区中的 grub.cfg 文件的部分内容如下：

```
### BEGIN /etc/grub.d/10_linux ###
function gfxmode {
        set gfxpayload="${1}"
}
set linux_gfx_mode=
export linux_gfx_mode
menuentry 'Deepin 20.2.4 GNU/Linux' --class deepin --class gnu-linux --class gnu --class os $menuentry_id_option 'gnulinux-simple-86d7cee2-6caa-454e-87af-24ef6419023c' {
        load_video
        insmod gzio
        if [ x$grub_platform = xxen ]; then insmod xzio; insmod lzopio; fi
        insmod part_gpt
        insmod ext2
        if [ x$feature_platform_search_hint = xy ]; then
          search --no-floppy --fs-uuid --set=root  b4e5d51b-ab86-4082-b8dd-124104ffffe0
        else
          search --no-floppy --fs-uuid --set=root b4e5d51b-ab86-4082-b8dd-124104ffffe0
        fi
        linux   /vmlinuz-5.10.60-amd64-desktop root=UUID=86d7cee2-6caa-454e-87af-24ef6419023c ro  splash quiet  DEEPIN_GFXMODE=$DEEPIN_GFXMODE
        initrd  /initrd.img-5.10.60-amd64-desktop
        boot
}
```

设置其中一个菜单项，名称为 Deepin 20.2.4 GNU/Linux，添加系统启动文件 linux 和 initrd。

```
### BEGIN /etc/grub.d/30_os-prober ###
menuentry 'Windows Boot Manager (在 /dev/nvme0n1p1)' --class windows --class os $menuentry_id_option 'osprober-efi-F6F8-BE08' {
        insmod part_gpt
        insmod fat
        if [ x$feature_platform_search_hint = xy ]; then
          search --no-floppy --fs-uuid --set=root  F6F8-BE08
        else
          search --no-floppy --fs-uuid --set=root F6F8-BE08
        fi
        chainloader /EFI/Microsoft/Boot/bootmgfw.efi
}
### END /etc/grub.d/30_os-prober ###
```

这里可以看到注释部分为系统探测器默认探测到的计算机存在的操作系统，复制了 windows 系统引导的名字 windows boot manager，同时使用上面提到过的 chainloader 函数，直接将控制权交给 bootmgfw.efi。

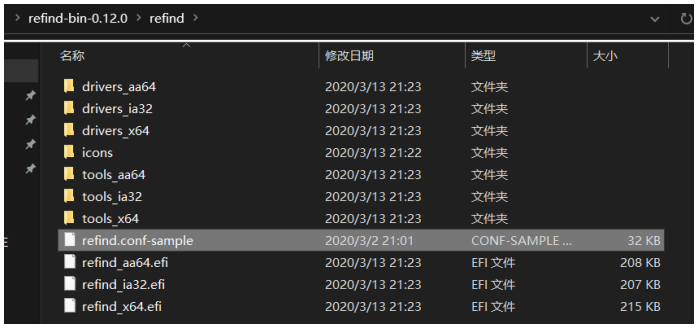
D. 后面的内容 BIOS 模式下与 GURB2 引导的 Linux 一样

可见，UEFI 下使用 GRUB2 引导操作系统和 BIOS 下几乎一样。不同的是 BIOS 下 GRUB2 需要先加载必要的 img 驱动。在 UEFI 下，UEFI 已经加载好了文件驱动和各种环境，GRUB2 直接使用即可。

（4） UEFI 启动方式由 rEFInd 引导的 Linux

为什么这里要提到使用 rEFInd 作为系统引导，因为 GRUB2 虽然功能极多，但是操作十分复杂。既然 UEFI 提供了如此完美的环境，就应该物尽其用。UEFI 有很多的安装方式，推荐使用 linux 环境下命令直接安装。这里为了说明 rEFInd 的原理，所以使用 windows 手动配置。

（1） 首先下载 rEFInd 压缩包，用 notepad++、vscode 之类的编辑工具（不要用记事本）打开配置文件。



rEFInd 中用于配置的文件为 refind.conf，与之对应的包括了 windows 中的 boot.ini、BCD，

以及 GRUB 中的 grub.conf、menu.lst、grub.cfg。下面只对部分的 refind.conf 的配置参数做介绍，详细内容请参考 https://www.cnblogs.com/dawnlz/p/14118246.html

```
# Timeout in seconds for the main menu screen. Setting the timeout to 0
# disables automatic booting (i.e., no timeout). Setting it to -1 causes
# an immediate boot to the default OS *UNLESS* a keypress is in the buffer
# when rEFInd launches, in which case that keypress is interpreted as a
# shortcut key. If no matching shortcut is found, rEFInd displays its
# menu with no timeout.
#
timeout 20
```

用于系统选择时间的 timeout

```
#dont_scan_dirs ESP:/EFI/boot,EFI/Dell,EFI/memtest86
```

位于 375 行的排除多余的引导目录，不然展示的条目多，且指向同一个系统，这里是因为 deepin 后装的缘故，grub2 产生了过多引导项目，按实际情况实际排除

```
#scan_all_linux_kernels false
```

位于 413 行的不扫描其他 linux 启动内核文件，这样 refind 会快些

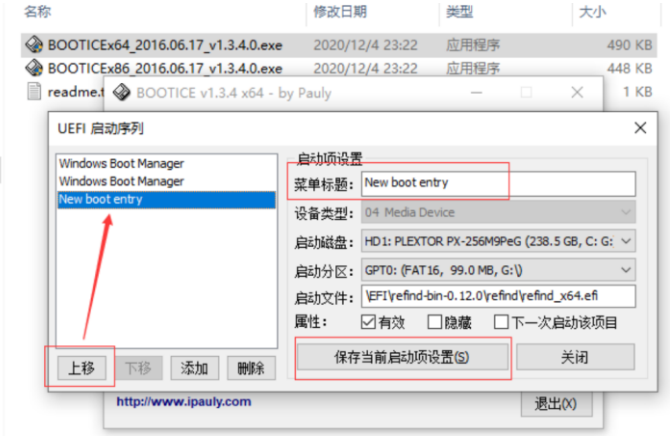
```
include themes/mythme/theme.conf
```

在最后一行通过简单的一句，就可以添加自定义的主题。

这样，通过简单几行参数，就完成了 rEFInd 的配置，没有 GRUB 中极为复杂的各种语句。rEFInd 会自动扫描操作系统并进行相应的操作。

（2） 将上面的 rEFInd 文件夹通过 DiskGenius 分区工具复制到 windows 自带的 ESP 中即可，无需单独建立 ESP。

（3） 现在只需要告诉 UEFI，存在这样的一个引导即可。那么这个内容则需要被写入到 UEFI 的 NVRAM 中。可以使用 easyEEFI 或 Bootice 引导工具。下面以 Bootice 为例



添加启动分区、启动文件为刚刚复制的 refind_x64.efi，然后将启动项顺序放到第一个即可。

参考资料：

[1] https://zhuanlan.zhihu.com/p/26098509 MBR 与 GPT 的区别

[2] https://blog.csdn.net/qq_36761831/article/details/104772269 BIOS 与 UEFI

[3] https://blog.csdn.net/housecarl/article/details/89329860 windows 启动过程

[4] https://blog.csdn.net/xinlan3618/article/details/78860317 MBR 引导程序类型

[5] https://blog.csdn.net/antdz/article/details/51296529 BIOS、MBR、PBR 等基础知识

[6] https://www.cnblogs.com/mayingkun/archive/2013/03/22/2974929.html windows NT 启动过程详解

[7] https://blog.csdn.net/alais/article/details/5129005 BIOS 启动流程分析

[8] https://blog.csdn.net/daydayup_668819/article/details/90172355 UEFI 系统启动流程分析

[9] https://blog.csdn.net/weixin_35830006/article/details/112173872 UEFI 下开机顺序

[10] https://blog.csdn.net/mycwq/article/details/51863075 windows 7 系统启动过程

[11] https://blog.csdn.net/zhongjin616/article/details/17630357 MBR 及 Linux 下 GRUB 执行原理

[12] http://www.360doc.com/content/11/0531/11/1074365_120667417.shtml 主引导扇区、分区表和分区引导扇区

[13] https://zhidao.baidu.com/question/26020552.html 扩展分区与逻辑分区

[14] https://blog.csdn.net/li33293884/article/details/50562527 GPT 分区表 GUID 详解

[15] https://blog.csdn.net/iware99/article/details/92641373 grub 引导启动过程详解

[16] https://www.cnblogs.com/yinheyi/p/7279508.html GRUB——系统的引导程序简单介绍

[17] https://blog.csdn.net/wu5795175/article/details/9134931 grub 0.97 浅析

[18] https://blog.csdn.net/jackailson/article/details/84109450 BIOS 中断 INT 13H

[19] https://blog.csdn.net/zhan570556752/article/details/88409802 GRUB 启动分析之 stage1.5

[20] https://blog.csdn.net/gebitan505/article/details/45057965 grub 的 menu.lst 写法

[21] https://blog.csdn.net/wmlwonder/article/details/41680473 menu.lst 和 grub.conf

[22] https://blog.csdn.net/geekard/article/details/6455502 initrd 和 kernel 的作用

[23] https://blog.csdn.net/seaship/article/details/99677384 GRUB2 详解

[24] https://www.cnblogs.com/dawnlz/p/14118246.html 多系统引导-refind

操作系统安装指南

1. 下载操作系统 iso 文件

Windows：windows(10/8/7/xp)、windows server(2019/2016/2012/2008)、ninjustu

Linux：Red Hat、CentOS、Ubuntu、SuSE、Gentoo、Debian、Fedora、Manjaro（Xface 最快/KDE 可定制化/Gnome 最便捷）、Deepin（国产）、Kali 等

由于 windows 系统不开源，因此 windows 没有第三方发行版。不过有国外大神改装的 ninjustu（忍者渗透系统）用于网络渗透。由于 ninjustu 对 windows 进行了大量的深度定制，导致发布第三版后被微软起诉侵权，目前 github 中已经下架了，不过可以从其他渠道下载。

2. UEFI 引导下的系统安装

使用 USBwriter 将系统镜像写入空 U 盘，然后插入 U 盘后，重启计算机，进入 BIOS / UEFI 界面，修改以下内容：

（1） Secure Boot。开启状态下 UEFI 拒绝使用除去微软以外的第三方 efi 系统引导。因此必须关闭；

（2） 模式调为 UEFI，而不是 Legacy；

（3） 启动顺序为 U 盘第一。

挂载点选择：如果读过上面的内容，那么应该无需过多说明。/boot 放在单独的 EFI 分区，格式化为 fat32 文件系统。/swap 为交换分区，与内存共同组成虚拟内存，不设置/swap 的情况下内存超出上限可能会导致 Linux 无法分配过多内存（在 windows 中叫做虚拟内存空间，存储与 C 盘，当然这个叫法并不准确，因为内存+交换分区=虚拟内存）。当然电脑 16g 内存的情况下不设置/swap 分区也无妨。其他挂载点随便选，或者直接把/挂到全部磁盘即可，格式化为 ext4

操作系统修复指南（仅限 UEFI 模式）

前提条件：关闭安全启动 secure boot

1. 系统进不去系列

准备一个 PE（Preinstallation Environment）系统盘，和一个空的 U 盘。

下载微 PE 工具（推荐）、大白菜工具。安装在 U 盘中。进入 BIOS，选择 U 盘启动顺序第一、关闭 secure boot、启动模式设置为 UEFI。

（1） 装双系统，误操作把 windows 的引导覆盖了。各种文件（例如 BCD）找不到了导致系统蓝屏。

使用微 PE 中的 Bootice 修复。教程网上搜索

（2） 忘记开机密码了

使用微 PE 中的修改启动密码功能

（3） C 盘被格式化了

使用微 PE 进入系统，把磁盘文件拷贝到其他移动设备，然后重装系统

2. 安装 windows，又安装了 Linux，最后只能进入 windows

回顾 windows 的 UEFI 启动流程：

UEFI 调用 NVRAM 中的启动项 -> UEFI 执行 windows 启动项中的 bootmgfw.efi 程序 -> bootmgfw.efi 读取 BCD 配置 -> 将控制权交给 winload.efi

此时的磁盘分区应该是：windows 的 EFI 分区、windows 其他分区、Linux 的 EFI 分区、Linux 其他分区。目前所有的文件都没有丢失，理论上讲，只需要添加上面 windows 启动流程的任意一个环节，让其引导到 Linux 即可。

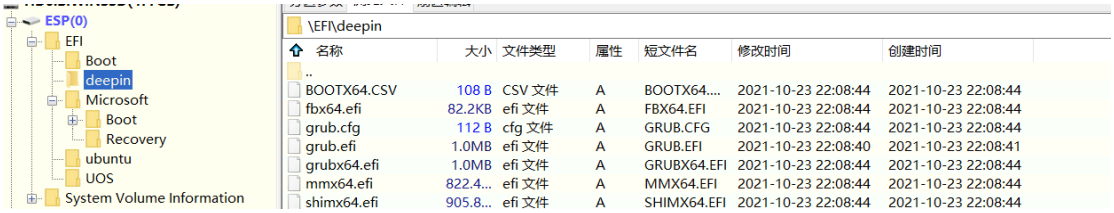
可使用的方法：

（1） 让 UEFI 调用 Linux 的启动项即可。Linux 启动项一般情况为 GRUB2，并且能够正确的将控制权交给 windows 系统的 bootmgfw.efi。此时需要使用 easyUEFI 工具或者 Bootice 工具编辑 UEFI 启动项

a) UEFI 启动项已经有 Linux 系统了，但是顺序在后面，直接将其排在第一个
b) UEFI 启动项没有 Linux 系统，那么添加一个 Linux 启动项，启动文件设置为 Linux 中的 EFI 分区下的 Grub2.efi
c) 前面步骤都做了还是进不去 Linux，进入 BIOS 主板设置，手动调整启动顺序为 Linux 优先。还是不行，只能说明一个问题：电脑主板厂商和 windows 合作，在这块主板的 UEFI 中只能显示 windows 的启动项，以及其他硬件（比如 U 盘），其他一律禁止加载。那么请看步骤 2

（2） 现在 UEFI 将会执行 windows 启动项中的 bootmgfw.efi 程序。使用 easyUEFI 或 Bootice 工具可以看到，这个程序的路径为/EFI/Microsoft/Boot/bootmgfw.efi，那么只需要偷梁换柱，将其换成自己想要的执行 efi 即可。

打开 DG（DiskGenius）



通常情况下，为了防止微软的流氓行为，Linux 系统在安装时，不但在自己的 EFI 分区写入了启动项、在 UEFI 写入了引导项，还会在微软启动 EFI 分区写入自己的引导项。其作用是调用自己所在 EFI 分区的文件。

如果没有这个，也没关系，从网上下载一个 rEFInd 引导，修改参数后，放进 EFI 分区，也可以使用。

因此，直接看准路径，将微软在 UEFI 中写入的路径改成自己的路径。

```
C:\Windows\system32>bcdedit

Windows 启动管理器
--------------------
标识符                  {bootmgr}
device                  partition=\Device\HarddiskVolume1
path                    EFI\MICROSOFT\BOOT\BOOTMGFW.EFI
description             Windows Boot Manager
locale                  zh-CN
inherit                 {globalsettings}
default                 {current}
resumeobject            {280a3372-ea99-11ea-b614-a28d5e71ca52}
displayorder            {current}
toolsdisplayorder       {memdiag}
timeout                 3
```

管理员权限打开命令行，输入 bcdedit，看到上面的 path 是 windows boot manager 写入到 UEFI 的路径，将其修改

```
C:\Windows\system32>
C:\Windows\system32>bcdedit /set {bootmgr} path EFI\deepin\grub.efi
```

（3） 在 bootmgfw.efi 读取 BCD 的时候，使用 easyBCD 增加 BCD 参数，让其引导 Linux。实测不可行，因为 windows 禁止了加载除 windows 以外的启动项。

3. 安装 windows，又安装了 Linux，最后只能进入 Linux

（1） Windows 的 EFI 分区还在：

a) 在 Linux 中用命令更新 grub 引导的配置文件 grub.cfg，让其重新检测 windows 的 bootmgfw.efi

b) 改成使用 rEFInd 引导

c) 手动修改 grub.cfg 并添加 windows 引导的方式可行，但是难度太大

（2） Windows 的 EFI 分区被覆盖了：参考 1，重新写 windows 的 EFI 分区。因为只有 bootmgfw 能引导 windows，bootmgfw 没有了只能重写。

4. 安装 Linux 后再装 windows，只能进入 windows 了

使用 rEFInd 引导。