**计算机的启动**

**下面是一个LINUX的完整启动过程，含BIOS细节。**

**最下方还给出了一个windows xp/server2003的启动细节，就是本程序虚拟机机上的系统。**

**两者的原理类似。**

**零、boot的含义**

先问一个问题，”启动”用英语怎么说？

回答是boot。可是，boot原来的意思是靴子，”启动”与靴子有什么关系呢？ 原来，这里的boot是bootstrap（鞋带）的缩写，它来自一句谚语：

"pull oneself up by one's bootstraps"

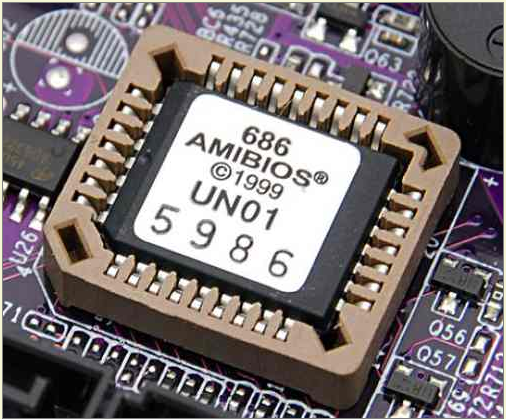
字面意思是”拽着鞋带把自己拉起来”，这当然是不可能的事情。最早的时候，工程师们用它来比喻，计算机启动是一个很矛盾的过程：必须先运行程序，然后计算机才能启动，但是计算机不启动就无法运行程序！

早期真的是这样，必须想尽各种办法，把一小段程序装进内存，然后计算机才能正常运行。所以，工程师们把这个过程叫做”拉鞋带”，久而久之就简称为boot了。

计算机的整个启动过程分成四个阶段。

**一、第一阶段：BIOS**

上个世纪70年代初，”只读内存”（read-only memory，缩写为ROM）发明，开机程序被刷入ROM芯片，计算机通电后，第一件事就是读取它。



这块芯片里的程序叫做”基本輸出輸入系統”（Basic Input/Output System），简称为BIOS。

**1.1 硬件自检**

BIOS中主要存放的程序包括：自诊断程序（通过读取CMOS RAM中的内容识别硬件配置，并对其进行自检和初始化）、CMOS设置程序（引导过程中，通过特殊热键启动，进行设置后，存入CMOS RAM中）、系统自动装载程序（在系统自检成功后，将磁盘相对0道0扇区上的引导程序装入内存使其运行）和主要I/O驱动程序和中断服务（BIOS和硬件直接打交道，需要加载I/O驱动程序）。

BIOS程序首先检查，计算机硬件能否满足运行的基本条件，这叫做”硬件自检”（Power-On Self-Test），缩写为POST。

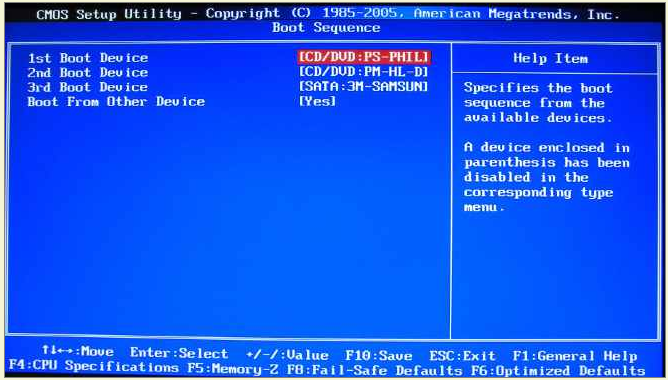
如果硬件出现问题，主板会发出不同含义的蜂鸣，启动中止。如果没有问题，屏幕就会显示出CPU、内存、硬盘等信息。

**1.2 启动顺序**

硬件自检完成后，BIOS把控制权转交给下一阶段的启动程序。

这时，BIOS需要知道，”下一阶段的启动程序”具体存放在哪一个设备。也就是说，BIOS需要有一个外部储存设备的排序，排在前面的设备就是优先转交控制权的设备。这种排序叫做”启动顺序”（Boot Sequence）。

打开BIOS的操作界面，里面有一项就是”设定启动顺序”。



**二、第二阶段：主引导记录**

BIOS按照”启动顺序”，把控制权转交给排在第一位的储存设备。即根据用户指定的引导顺序从软盘、硬盘或是可移动设备中读取启动设备的MBR，并放入指定的位置（0x7c000）内存中。

这时，计算机读取该设备的第一个扇区，也就是读取最前面的512个字节。如果这512个字节的最后两个字节是0x55和0xAA，表明这个设备可以用于启动；如果不是，表明设备不能用于启动，控制权于是被转交给”启动顺序”中的下一个设备。

这最前面的512个字节，就叫做”主引导记录”（Master boot record，缩写为MBR）。

**2.1 主引导记录的结构**

“主引导记录”只有512个字节，放不了太多东西。它的主要作用是，告诉计算机到硬盘的哪一个位置去找操作系统。

主引导记录由三个部分组成：

（1） 第1-446字节：调用操作系统的机器码。

（2） 第447-510字节：分区表（Partition table）。

（3） 第511-512字节：主引导记录签名（0x55和0xAA）。

其中，第二部分”分区表”的作用，是将硬盘分成若干个区。

**2.2 分区表**

硬盘分区有很多好处。考虑到每个区可以安装不同的操作系统，”主引导记录”因此必须知道将控制权转交给哪个区。

分区表的长度只有64个字节，里面又分成四项，每项16个字节。所以，一个硬盘最多只能分四个一级分区，又叫做”主分区”。

每个主分区的16个字节，由6个部分组成：

（1） 第1个字节：如果为0x80，就表示该主分区是激活分区，控制权要转交给这个分区。四个主分区里面只能有一个是激活的。

（2） 第2-4个字节：主分区第一个扇区的物理位置（柱面、磁头、扇区号等等）。

（3） 第5个字节：主分区类型。

（4） 第6-8个字节：主分区最后一个扇区的物理位置。

（5） 第9-12字节：该主分区第一个扇区的逻辑地址。

（6） 第13-16字节：主分区的扇区总数。

最后的四个字节（”主分区的扇区总数”），决定了这个主分区的长度。也就是说，一个主分区的扇区总数最多不超过2的32次方。

如果每个扇区为512个字节，就意味着单个分区最大不超过2TB。再考虑到扇区的逻辑地址也是32位，所以单个硬盘可利用的空间最大也不超过2TB。如果想使用更大的硬盘，只有2个方法：一是提高每个扇区的字节数，二是增加扇区总数。

**三、第三阶段：硬盘启动**

这时，计算机的控制权就要转交给硬盘的某个分区了，这里又分成三种情况。

**3.1 情况A：卷引导记录**

上一节提到，四个主分区里面，只有一个是激活的。计算机会读取激活分区的第一个扇区，叫做”卷引导记录”（Volume boot record，缩写为VBR）。

“卷引导记录”的主要作用是，告诉计算机，操作系统在这个分区里的位置。然后，计算机就会加载操作系统了。

**3.2 情况B：扩展分区和逻辑分区**

随着硬盘越来越大，四个主分区已经不够了，需要更多的分区。但是，分区表只有四项，因此规定有且仅有一个区可以被定义成”扩展分区”（Extended partition）。

所谓”扩展分区”，就是指这个区里面又分成多个区。这种分区里面的分区，就叫做”逻辑分区”（logical partition）。

计算机先读取扩展分区的第一个扇区，叫做”扩展引导记录”（Extended boot record，缩写为EBR）。它里面也包含一张64字节的分区表，但是最多只有两项（也就是两个逻辑分区）。

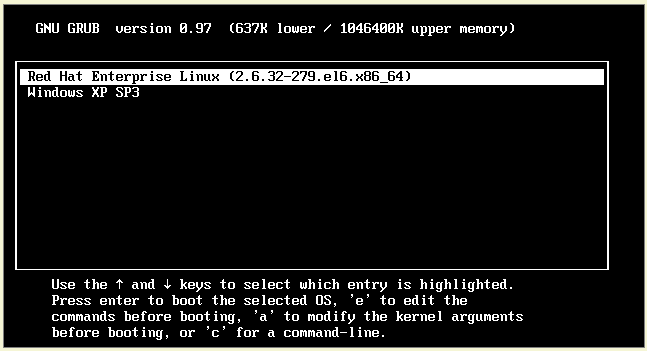
计算机接着读取第二个逻辑分区的第一个扇区，再从里面的分区表中找到第三个逻辑分区的位置，以此类推，直到某个逻辑分区的分区表只包含它自身为止（即只有一个分区项）。因此，扩展分区可以包含无数个逻辑分区。

但是，似乎很少通过这种方式启动操作系统。如果操作系统确实安装在扩展分区，一般采用下一种方式启动。

**3.3 情况C：启动管理器**

在这种情况下，计算机读取”主引导记录”前面446字节的机器码之后，不再把控制权转交给某一个分区，而是运行事先安装的”启动管理器”（boot loader），由用户选择启动哪一个操作系统。

Linux环境中，目前最流行的启动管理器是Grub。



对于grub而言，在MBR中的446字节的引导程序属于GRUB的开始执行程序，通过这段程序，进一步执行stage1.5或是stage2的执行程序，将在下面详细介绍执行过程。

其中stage1.5或是stage2便属于阶段2引导的过程了，stage2过程也是作为GRUB kernel的核心代码出现。Stage1.5过程（对于GRUB而言存在stage1.5，GRUB2则不存在）的功能很单一，主要就是为了引导stage2过程服务。由于stage2过程的代码存放在文件系统下的boot分区目录中，因此stage1.5过程就是需要提供一个文件系统的环境，而该文件系统环境需要保证系统可以找到stage2过程的文件，那么stage1.5阶段提供的文件系统需要是boot文件系统所对应的，这个在执行grub install过程中就已经确定了。stage2过程中，主要会把系统切换到保护模式，设置好C运行时环境，找到config文件（事实上就是menulist文件），如果没有找到就执行一个shell，等待用户的执行。然后的工作就变成了输入命令->解析命令->执行命令的循环中。当然该阶段引导的最终状态就是执行boot命令，将内核和initrd镜像加载进入内存中，进而将控制权转交给内核。

**四、第四阶段：操作系统**

控制权转交给操作系统后，操作系统的内核首先被载入内存。

以Linux系统为例，先载入/boot目录下面的kernel。内核加载成功后，第一个运行的程序是/sbin/init。它根据配置文件（Debian系统是/etc/initab）产生init进程。这是Linux启动后的第一个进程，pid进程编号为1，其他进程都是它的后代。

然后，init线程加载系统的各个模块，比如窗口程序和网络程序，直至执行/bin/login程序，跳出登录界面，等待用户输入用户名和密码。

至此，全部启动过程完成。

**BIOS启动细节：**

a) 按下电源开关，电源就开始向主板和其它设备供电；当芯片组检测到电源已经开始稳定供电了(当然从不稳定到稳定的过程只是一瞬间的事情)，它便撤去RESET信号(如果是手工按下计算机面板上的Reset按钮来重启机器，那么松开该按钮时芯片组就会撤去RESET信号)；CPU马上就从地址FFFF:0000H 处开始执行指令，放在这里的只是一条跳转指令，跳到系统BIOS中真正的启动代码处。(严格来说，CPU复位以后，是通过其内部的电压值来完成初始化的，其数字信号值为CS=ffffh, IP=0h。  
因为初始化之前，ROM-BIOS是不能运行的。于是CPU在各主板芯片的协调下开始建立内存数据，由FFFFH：0H开始读取数据，这个区域为ROM-BIOS的逻辑映射地址，放在此处的是一条BIOS程序的跳转指令，于是开始转去执行BIOS程序，开始完成硬件检测、系统配置，建立中断向量等，执行完毕后根据用户设置的启动设备优先级顺序执行该设备中的[主引导记录](https://www.baidu.com/s?wd=%E4%B8%BB%E5%BC%95%E5%AF%BC%E8%AE%B0%E5%BD%95&tn=44039180_cpr&fenlei=mv6quAkxTZn0IZRqIHckPjm4nH00T1Y4nWT4uWFhnWIBrHb3mWbz0ZwV5Hcvrjm3rH6sPfKWUMw85HfYnjn4nH6sgvPsT6KdThsqpZwYTjCEQLGCpyw9Uz4Bmy-bIi4WUvYETgN-TLwGUv3En1DYPWTsPjnd)。最后移交控制权限给[主引导记录](https://www.baidu.com/s?wd=%E4%B8%BB%E5%BC%95%E5%AF%BC%E8%AE%B0%E5%BD%95&tn=44039180_cpr&fenlei=mv6quAkxTZn0IZRqIHckPjm4nH00T1Y4nWT4uWFhnWIBrHb3mWbz0ZwV5Hcvrjm3rH6sPfKWUMw85HfYnjn4nH6sgvPsT6KdThsqpZwYTjCEQLGCpyw9Uz4Bmy-bIi4WUvYETgN-TLwGUv3En1DYPWTsPjnd)，读入0000h：7c00h，完成[操作系统](https://www.baidu.com/s?wd=%E6%93%8D%E4%BD%9C%E7%B3%BB%E7%BB%9F&tn=44039180_cpr&fenlei=mv6quAkxTZn0IZRqIHckPjm4nH00T1Y4nWT4uWFhnWIBrHb3mWbz0ZwV5Hcvrjm3rH6sPfKWUMw85HfYnjn4nH6sgvPsT6KdThsqpZwYTjCEQLGCpyw9Uz4Bmy-bIi4WUvYETgN-TLwGUv3En1DYPWTsPjnd)的引导。)

b) 系统BIOS的启动代码首先进行POST(Power－On Self Test，加电后自检)。POST的主要检测系统中一些关键设备是否存在和能否正常工作，例如内存和显卡等设备；由于POST是最早进行的检测过程，此时显卡还没有初始化，如果系统BIOS在进行POST的过程中发现了一些致命错误，例如没有找到内存或者内存有问题(此时只会检查640K常规内存)，那么系统BIOS就会直接控制喇叭发声来报告错误，声音的长短和次数代表了错误的类型；在正常情况下，POST过程进行得非常快，几乎无法感觉到它的存在。POST结束之后就会调用其它代码来进行更完整的硬件检测。

c) 接下来系统BIOS将查找显卡的BIOS。前面说过，存放显卡BIOS的ROM芯片的起始地址通常设在C0000H处，系统BIOS在这个地方找到显卡BIOS之后就调用它的初始化代码，由显卡BIOS来初始化显卡。此时多数显卡都会在屏幕上显示出一些初始化信息，介绍生产厂商、图形芯片类型等内容，不过这个画面几乎是一闪而过。系统BIOS接着会查找其它设备的BIOS程序，找到之后同样要调用这些BIOS内部的初始化代码来初始化相关的设备。

d) 查找完所有其它设备的BIOS之后，系统BIOS将显示出它自己的启动画面，其中包括有系统BIOS的类型、序列号和版本号等内容。

e) 接着系统BIOS将检测和显示CPU的类型和工作频率，测试所有的RAM，并同时在屏幕上显示内存测试的进度。可以在CMOS设置中自行决定使用简单耗时少或者详细耗时多的测试方式。

f) 内存测试通过之后，系统BIOS将开始检测系统中安装的一些标准硬件设备，包括硬盘、CD-ROM、串口、并口和软驱等设备，另外绝大多数较新版本的系统BIOS在这一过程中还要自动检测和设置内存的定时参数、硬盘参数和访问模式等。

g) 标准设备检测完毕后，系统BIOS内部支持即插即用的代码将开始检测和配置系统中安装的即插即用设备。每找到一个设备之后，系统BIOS都会在屏幕上显示出设备的名称和型号等信息，同时为该设备分配中断、DMA通道和I/O端口等资源。

h) 到这一步为止，所有硬件都已经检测配置完毕了，多数系统BIOS会重新清屏并在屏幕上方显示出一个表格，其中概略地列出了系统中安装的各种标准硬件设备，以及它们使用的资源和一些相关工作参数。

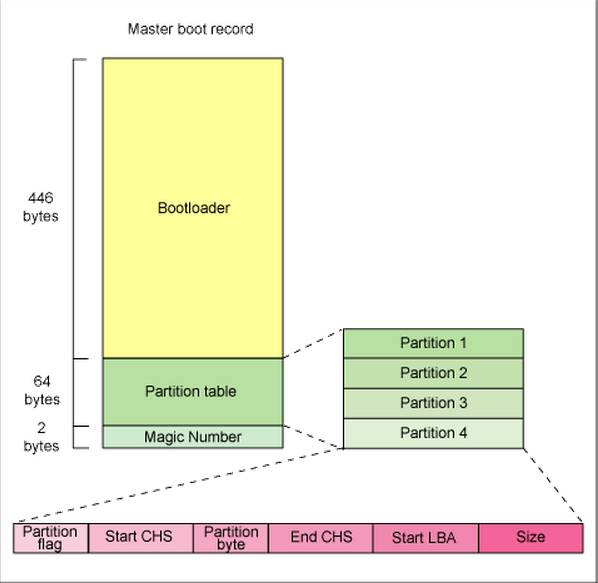
i) 接下来系统BIOS将更新ESCD(Extended System Configuration Data，扩展系统配置数据)。ESCD是系统BIOS用来与操作系统交换硬件配置信息的一种手段，这些数据被存放在CMOS(一小块特殊的RAM，由主板上的电池来供电)之中。通常ESCD数据只在系统硬件配置发生改变后才会更新，所以不是每次启动机器时都能够看到“Update ESCD… Success”这样的信息。不过，某些主板的系统BIOS在保存ESCD数据时使用了与Windows 9x不相同的数据格式，于是Windows 9x在它自己的启动过程中会把ESCD数据修改成自己的格式。但在下一次启动机器时，即使硬件配置没有发生改变，系统BIOS也会把ESCD的数据格式改回来。如此循环，将会导致在每次启动机器时，系统BIOS都要更新一遍ESCD，这就是为什么有些机器在每次启动时都会显示出相关信息的原因。

j) ESCD更新完毕后，系统BIOS的启动代码将进行它的最后一项工作：即根据用户指定的启动顺序从软盘、硬盘或光驱启动MBR。在这个过程中会按照启动顺序顺序比较其放置MBR的位置的结尾两位是否为0xAA55，通过这种方式判断从哪个引导设备进行引导。在确定之后，将该引导设备的MBR内容读入到0x7C00[1]的位置，并再次判断其最后两位，当检测正确之后，进行阶段1的引导。

**EFI启动细节**

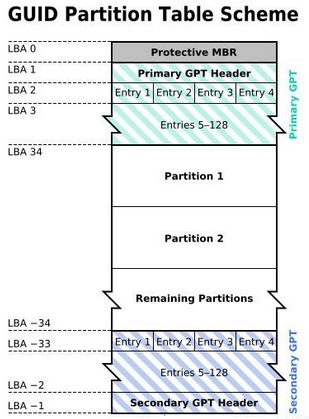
与传统MBR相比，GPT采用了不同的分区方式。

对于传统MBR，其结构主要如下：



上图即对上文中所述的很形象的说明，在图中看到MBR被分成三个部分，分别是：Bootloader、分别表以及Magic Number。其中Bootloader部分为stage1中被执行的起始部分。

相反，对于EFI系统中所采用的GPT分区方式，则采用了不同于MBR分区方式的形式，从下图中可以发现：



如上图所示，GPT分区表主要包括：保护MBR、首要GPT头、首要GPT、备用GPT、备用GPT头和磁盘数据区。保护MBR与正常的MBR区别不大，主要是分区表上的不同，在保护MBR中只要一个表示为0xEE的分区，以此来表示这块硬盘使用GPT分区表。首要GPT头包含了众多信息，具体内容如下：



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。分区表头还记录了这块硬盘的GUID，记录了分区表头本身的位置和大小（位置总是在LBA1）以及备份分区表头和分区表的位置和大小（在硬盘的最后）。它还存储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有错误，如果出错了，可以使用软件从硬盘最后的备份GPT分区表恢复整个分区表，如果备份GPT也校验错误，那么磁盘将不可用，系统拒绝启动。

接下来主要是128个分区表项，GPT分区表使用简单而直接的方式表示分区。一个分区表项的前16字节是分区类型GUID。例如，EFI系统分区的GUID类型是{C12A7328-F81F-11D2-BA4B-00A0C93EC93B} 。接下来的16字节是该分区的唯一的GUID（这个指的是该分区本身，而之前的GUID指的是该分区的类型）。在接下来是分区其实和末尾的64位LBA编号，以及分区的名字和属性。具体结构如下表：



**MBR引导**

接下来开始真正的引导过程了，主要说明GRUB的引导。总体上GRUB更像是一个mini os，只不过这个mini os的作用只是加载其他的操作系统，在GRUB中包括stage1、stage1.5（可选）和stage2，其中stage1和stage1.5属于boot loader，stage2属于mini os的内核部分。GRUB中stage1过程主要位于MBR的前446字节中（对于支持GPT分区的磁盘，同样有最开始的512字节作为保护MBR，保护MBR与正常的MBR区别不大，主要是分区表上的不同，在保护MBR中只要一个表示为0xEE的分区，以此来表示这块硬盘使用GPT分区表，不能识别GPT硬盘的操作系统通常会识别出一个未知类型的分区，并且拒绝对硬盘进行操作），之后的64字节为硬盘的分区表，最后两个字节为MBR结束标志位（0xAA55）。

stage1部分占用了446字节，其代码文件是源码目录下stage1/stage1.S文件，汇编后生成一个512字节的boot.img，被写在硬盘的0面0道1扇区中，作为硬盘的MBR。stage1的工作很简单，就是加载0面0道2扇区上的512字节到0×8000，然后跳转到0×8000执行。

在0面0道2扇区上的512字节内容为stage1/start.S文件汇编后生成。该扇区上的内容的作用是加载stage1.5或是stage2过程，并将控制权转交。

**Grub引导**

在start过程将控制权转交后，接下来就是GRUB的核心过程了。该过程之所以区分stage1.5和stage2，主要原因是GRUB和GRUB2的区别。在GRUB2中，将stage1.5过程集成到了stage2的过程中，所以stage1.5过程仅仅是针对GRUB的。下面将会分别介绍两种GRUB版本的两种过程。

4.1 GRUB中stage1.5过程

Stage1.5过程很无辜，它的作用很单一，但是非常关键。它的主要功用就是构造一个boot分区系统对应的文件系统，这样可以通过文件系统的路径（/boot/grub/）寻找stage2过程需要的core.img，进而加载到内存中开始执行。

Stage1.5存在于0面0道3扇区开始的地方，并一直延续十几k字节的区域，具体的大小与相应的文件系统的大小有关（文中涉及到了0面0道1-3+x扇区，这部分扇区为保留扇区，BIOS不会放置任何数据。正因为如此如果转换到GPT分区形式，系统将不能被正确引导，如上文所示，MBR后面的扇区都被其他内容所占据）。Stage1.5过程被构建成多种不同类型，但是功能类似，下面简单介绍一下基本的stage1.5过程的文件系统。e2fs\_stage1\_5（针对ext2fs，可引导ext2和ext3文件系统）、fat\_stage1\_5（针对fat文件系统，可引导fat32和fat16）、ffs\_stage1\_5、jfs\_stage1\_5、minix\_stage1\_5、reiserfs\_stage1\_5、vstafs\_stage1\_5和xfs\_stage1\_5，这些文件被称为stage1.5过程，这些文件每个至少都在11k以上。除此之外还有两个比较特殊的文件，分别为nbgrub和pxegrub，这两个文件主要是在网络引导时使用，只是格式不同而已，他们很类似与stage2，只是需要建立网络来获取配置文件。

由于stage1.5过程中会涉及到多个文件系统对应的文件，因此本文中主要以ext2fs为例进行说明，其他文件系统与此类似，可以同样进行分析理解。

对于ext2fs文件系统，用于生成该文件系统的stage1.5过程文件（e2fs\_stage1\_5）的代码为stage2/fsys\_ext2fs.c文件。

在stage2/filesys.h文件中定义了每个文件系统对外的接口，用于上层调用，作为stage2过程寻找核心代码使用，文件系统一般被定义的接口主要就是三个函数，分别是mount、read和dir函数。对应ext2fs，其定义的函数为：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | #ifdef FSYS\_EXT2FS  #define FSYS\_EXT2FS\_NUM 1  int ext2fs\_mount (void);  int ext2fs\_read (char \*buf, int len);  int ext2fs\_dir (char \*dirname);  #else  #define FSYS\_EXT2FS\_NUM 0  #endif |

针对ext2fs有如上的函数名称，每个函数将具体在stage2/fsys\_ext2fs.c文件中被定义，这里面没有包含任何的写的过程，对于bootloader而言仅仅读就可以完成任务了，没必要对其系统进行写操作。其中ext2fs\_mount函数用于检查文件系统类型，并将superblock读入到内存中；ext2fs\_read函数和ext2fs\_dir函数用于对文件系统具体的操作。在stage2/fsys\_ext2fs.c文件中除了需要对这三个函数的定义之外，还需要文件系统的属性的数据结构（superblock、inode和group结构，这些结构最初被定义在include/linux/ext2\_fs.h文件中），通过这些数据结构描述一个文件系统。

如果读者有兴趣可以试着创建新的文件系统的支持，可以参照目前存在的一些文件系统的模板（实例）编写。

4.2 GRUB中stage2过程

GRUB中的核心过程也就是stage2过程了，该过程主要是在文件系统建立以后选择合适的操作系统进行加载并转交控制权，达到最后引导操作系统的目标。由于GRUB属于multi boot loader，因此在引导的时候要进行选择，选择哪种操作系统来运行。在GRUB内部主要包括两种方式，首先是从menu.list中读取显示到屏幕让用户选择，其次是通过grub-shell中定义的命令手动进行启动。本文将在后面介绍这两种方式如何运行，接下来先介绍一下stage2的具体的执行过程。

在上面一节中介绍过，stage1.5过程中将boot分区的文件系统加载了，之后又做了一件事情，就是将控制权转交给stage2，而stage2入口的地方就是stage2/asm.S文件。Stage2/asm.S文件属于汇编代码，主要作用是初始化C语言的运行环境，为下面执行C语言的函数做好准备，在准备好之后，将执行init\_bios\_info(stage2/common.c)函数。init\_bios\_info函数的作用是执行一些底层的函数，然后跳转到cmain执行，cmain函数位于stage2/stage2.c文件中。cmain函数内部进行一个死循环，在循环内部首先加载配置文件，显示给用户，在这同时循环一个内层循环，在内层循环中，获取配置文件中的命令，并解析执行。过程中如果没有可用的配置文件，那么进入命令行模式（enter\_cmdline函数），如果找到可用的menu，那么开始执行menu的对应的内容（run\_menu函数）。

对于enter\_cmdline（stage2/stage2.c）函数，将调用find\_command（stage2/cmdline.c），进而执行相应命令的函数。

对于run\_menu（stage2/stage2.c）函数，将调用stage2/cmdline.c文件中的run\_script函数，进而调用find\_command，执行相应命令的函数。

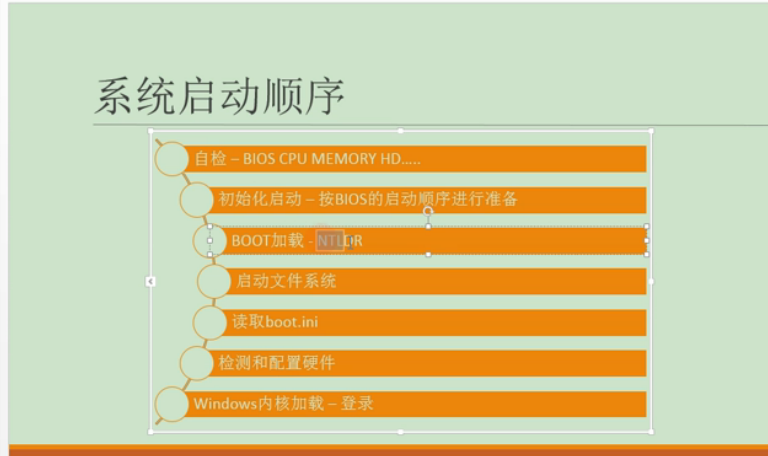
这两种方式虽然经过了不同的过程，对用户输入的行为进行分析和处理，最终调用的函数为find\_command，在该函数中顺序循环比较“输入”的命令是否与系统内部定义的相同，如果相同转到执行该函数。在这个比较的过程中包含了一个全局的数据结构为struct builtin（stage2/shared.h），由该数据结构组成了一个table类型（stage2/builtins.c），将命令与相对应的builtin结构对应一起并进行串联。下面描述一下builtin结构的定义：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | struct builtin {  /\* 命令名称，重要，是搜索命令时的依据\*/  char \*name;  /\* 命令函数，重要，是搜索匹配后调用的函数\*/  int (\*func) (char \*, int);  /\* 功能标示，一般未用到. \*/  int flags;  /\* 简短帮助信息\*/  char \*short\_doc;  /\* 完整帮助信息\*/  char \*long\_doc;  };  struct builtin \*builtin\_table[]； |

有兴趣的读者可以对上面的内容进行扩展，形成自己的命令，主要在stage2/builtins.c文件中按照预定的格式更新，并添加到builtin\_table中即可。

在上面打开配置文件的过程中，主要是通过一些文件操作函数（被定义在stage2/disk\_io.c中）完成。这些文件操作函数主要包括：grub\_open、grub\_read、grub\_seek和grub\_close等，这些函数属于grub对外的上层接口，具体的函数内部将调用前文中提到的boot分区对应的文件系统的相应的函数完成，这个过程主要是通过回调函数来完成。该过程整体思路类似于面向对象过程，通过对象操作具体的函数

windows xp系统:



NTLDR是一个隐藏的，只读的系统文件，位置在[系统盘](http://baike.baidu.com/view/299690.htm)的根目录，用来装载操作系统。

NTLDR全称是NT Loader，是系统加载程序，NTLDR文件是Windows NT/windows 2000/windows XP/windows server 2003的[引导文件](http://baike.baidu.com/view/2141961.htm)

**正常系统引导过程**

NTLDR文件的是一个隐藏的，只读的系统文件，位置在[系统盘](http://baike.baidu.com/view/299690.htm)的根目录，用来装载操作系统。

一般情况系统的引导过程是这样的代码

1：电源自检程序开始运行

2：[主引导记录](http://baike.baidu.com/view/418401.htm)被装入内存，并且程序开始执行

3：[活动分区](http://baike.baidu.com/view/493740.htm)的[引导扇区](http://baike.baidu.com/view/409354.htm)被装入内存

4：NTLDR从引导扇区被装入并初始化

5：将处理器的实模式改为32位平滑内存模式

6：NTLDR开始运行适当的小文件系统[驱动程序](http://baike.baidu.com/view/1048.htm)。

小文件系统驱动程序是建立在NTLDR内部的，它能读FAT或NTFS。

7：NTLDR读[boot.ini](http://baike.baidu.com/view/275448.htm)文件

8：NTLDR装载所选操作系统

如果windows NT/windows 2000/windows XP/windows server 2003这些操作系统被选择，NTLDR运行Ntdetect。

对于其他的操作系统，NTLDR装载并运行Bootsect.dos然后向它传递控制。

windows NT过程结束。

9：Ntdetect搜索计算机硬件并将列表传送给NTLDR，以便将这些信息写进\\HKE Y\_LOCAL\_MACHINE\HARDWARE中。

10：然后NTLDR装载Ntoskrnl.exe，[Hal.dll](http://baike.baidu.com/view/326601.htm)和[系统信息](http://baike.baidu.com/view/4036790.htm)集合。

11：Ntldr搜索系统信息集合，并装载[设备驱动](http://baike.baidu.com/view/547398.htm)配置以便设备在启动时开始工作

12：Ntldr把控制权交给Ntoskrnl.exe，这时，启动程序结束，装载阶段开始

**详细过程：**

NTLDR一般存放于C盘根目录下，是一个具有隐藏和只读属性的系统文件。它的主要职责是解析Boot.ini文件。如果你对它的理解还不是很清楚，那么下面我们就以Windows XP为例介绍NTLDR在系统引导过程中的作用。

Windows XP在引导过程中将经历预引导、引导和加载[内核](http://baike.baidu.com/view/1366.htm)三个阶段，这与Windows 9X直接读取[引导扇区](http://baike.baidu.com/view/409354.htm)的方式来启动系统是完全不一样的，NTLDR在这三个阶段的引导过程中将起到至关重要的作用。

### 预引导

在预引导阶段里计算机所做的工作有：运行POST程序，POST将检测系统的总内存以及其他硬件设备的状况，将磁盘第一个物理[扇区](http://baike.baidu.com/view/201129.htm)加载到内存，加载硬盘[主引导记录](http://baike.baidu.com/view/418401.htm)并运行，主引导记录会查找[活动分区](http://baike.baidu.com/view/493740.htm)的起始位置。接着活动分区的[引导扇区](http://baike.baidu.com/view/409354.htm)被加载并执行，最后从引导扇区加载并初始化NTLDR文件。

### 引导

在引导阶段中，Windows XP将会依次经历初始引导加载器阶段、操作系统选择阶段、硬件检测阶段以及配置选择阶段这四个小的阶段。

1：在初始引导加载器阶段中，NTLDR将把计算机的微处理器从实模式转换为32位[平面内存模式](http://baike.baidu.com/view/3658394.htm)，在实模式中，系统会为MS－DOS预留640KB大小的内存空间，其余的内存都被看做是[扩展内存](http://baike.baidu.com/view/404404.htm)，在32位平面模式中系统将所有内存都视为可用内存，然后NTLDR执行适当的小型文件系统[驱动程序](http://baike.baidu.com/view/1048.htm)，这时NTLDR可以识别每一个用NTFS或FAT格式的文件系统分区，至此初始引导加载器阶段结束。

2：当初始引导加载器阶段结束后将会进入操作系统选择阶段，如果计算机上安装了多个操作系统，由于NTLDR加载了正确的Boot.ini文件，那么在启动的时候将会出现要求选择操作系统的菜单，NTLDR正是从[boot.ini](http://baike.baidu.com/view/275448.htm)文件中查找到系统文件的分区位置。如果选择了NT系统，那么NTLDR将会运行[NTDETECT](http://baike.baidu.com/view/1513463.htm)文件，否则NTLDR将加载BOOTSECT.DOS，然后将控制权交给BOOTSECT.DOS。如果Boot.ini文件中只有一个操作系统或者其中的timeout值为0，那么将不会出现选择操作系统的菜单画面，如果Boot.ini文件非法或不存在，那么NTLDR将会尝试从默认系统卷启动系统。

小提示：NTLDR启动后，如果在系统根目录下发现有Hiberfil.sys文件且该文件有效，那么NTLDR将读取Hiberfil.sys文件里的信息并让系统恢复到休眠以前的状态，这时并不处理Boot.ini文件。

3：当操作系统选择阶段结束后将会进入硬件检测阶段，这时NTDETECT文件将会收集计算机中硬件信息列表，然后将列表返回到NTLDR，这样NTLDR将把这些硬件信息加载到注册表“[HKEY\_LOCAL\_MACHINE](http://baike.baidu.com/view/1387918.htm)”中的Hardware中。

4：硬件检测阶段结束后将会进入配置选择阶段，如果有多个[硬件配置](http://baike.baidu.com/view/3163917.htm)列表，那么将会出现配置文件选择菜单，如果只有一个则不会显示。

### 加载内核

在加载内核阶段中，NTLDR将加载NTOSKRNL.EXE内核程序，然后NTLDR将加载硬件抽象层(HAL.dll)，接着系统将加载注册表中的“HKEY\_LOCAL\_MACHINE\System”键值，这时NTLDR将读取“HKEY\_LOCAL\_MACHINE\System\select”键值来决定哪一个ControlSet将被加载。所加载的ControlSet将包含设备的驱动程序以及需要加载的服务。再接着NTLDR加载注册表“HKEY\_LOCAL\_MACHINE\System\service”下的start键值为0的底层[设备驱动](http://baike.baidu.com/view/547398.htm)。当ControlSet的镜像CurrentControlSet被加载时，NTLDR将把控制权传递给NTOSKRNL.EXE，至此引导过程将结束。

小提示：如果在启动的时候按F8键，那么我们将会在启动菜单中看到多种选择启动模式，这时NTLDR将根据用户的选择来使用启动参数加载NT[内核](http://baike.baidu.com/view/1366.htm)，用户也可以在Boot.ini文件里设置启动参数。

NTLDR具有向下兼容性，也就是说高版本的NTLDR可以启动低版本的Windows NT操作系统，目前最新版本的NTLDR是在Windows Server 2003 SP1附带的版本。

如果你已经安装了一个高版本的操作系统，那么如果再想安装一个低版本的操作系统，那么低版本的操作系统在安装后会出现无法进入原来高版本的操作系统的故障。问题的原因是低版本的NTLDR和NTDETECT不能识别高版本操作系统的NTOSKRNL.EXE，从而造成系统不能正常启动。