**解析 Linux 中的 VFS 文件系统机制**

本文阐述 Linux 中的文件系统部分，源代码来自基于 IA32 的 2.4.20 内核。总体上说 Linux 下的文件系统主要可分为三大块：一是上层的文件系统的系统调用，二是虚拟文件系统 VFS(Virtual Filesystem Switch)，三是挂载到 VFS 中的各实际文件系统，例如 ext2，jffs 等。本文侧重于通过具体的代码分析来解释 Linux 内核中 VFS 的内在机制，在这过程中会涉及到上层文件系统调用和下层实际文件系统的如何挂载。文章试图从一个比较高的角度来解释 Linux 下的 VFS 文件系统机制。

5https://dw1.s81c.com/developerworks/i/v17/dw-cmts-arrow.png [评论](https://www.ibm.com/developerworks/cn/linux/l-vfs/#icomments)

[Ricard Chen](https://www.ibm.com/developerworks/cn/linux/l-vfs/#authorN10017) ([ricard\_chen@yahoo.com](mailto:ricard_chen@yahoo.com?subject=%E8%A7%A3%E6%9E%90%20Linux%20%E4%B8%AD%E7%9A%84%20VFS%20%E6%96%87%E4%BB%B6%E7%B3%BB%E7%BB%9F%E6%9C%BA%E5%88%B6))

|  |
| --- |
| XML error: Please enter a value for the author element's jobtitle attribute, or the company-name element, or both. |

2005 年 4 月 01 日

* [expand](https://www.ibm.com/developerworks/cn/linux/l-vfs/#toggle)**内容**



在 IBM Bluemix 云平台上开发并部署您的下一个应用。

[**开始您的试用**](https://developer.ibm.com/sso/bmregistration?lang=zh_CN&ca=dwchina-_-bluemix-_-l-vfs-_-sidebar)

**1. 摘要**

本文阐述 Linux 中的文件系统部分，源代码来自基于 IA32 的 2.4.20 内核。总体上说 Linux 下的文件系统主要可分为三大块：一是上层的文件系统的系统调用，二是虚拟文件系统 VFS(Virtual Filesystem Switch)，三是挂载到 VFS 中的各实际文件系统，例如 ext2，jffs 等。本文侧重于通过具体的代码分析来解释 Linux 内核中 VFS 的内在机制，在这过程中会涉及到上层文件系统调用和下层实际文件系统的如何挂载。文章试图从一个比较高的角度来解释 Linux 下的 VFS 文件系统机制，所以在叙述中更侧重于整个模块的主脉络，而不拘泥于细节，同时配有若干张插图，以帮助读者理解。

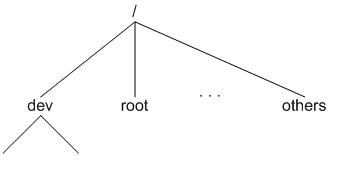
相对来说，VFS 部分的代码比较繁琐复杂，希望读者在阅读完本文之后，能对 Linux 下的 VFS 整体运作机制有个清楚的理解。建议读者在阅读本文前，先尝试着自己阅读一下文件系统的源代码，以便建立起 Linux 下文件系统最基本的概念，比如至少应熟悉 super block, dentry, inode，vfsmount 等数据结构所表示的意义，这样再来阅读本文以便加深理解。

[**回页首**](https://www.ibm.com/developerworks/cn/linux/l-vfs/#ibm-pcon)

**2. VFS 概述**

VFS 是一种软件机制，也许称它为 Linux 的文件系统管理者更确切点，与它相关的数据结构只存在于物理内存当中。所以在每次系统初始化期间，Linux 都首先要在内存当中构造一棵 VFS 的目录树(在 Linux 的源代码里称之为 namespace)，实际上便是在内存中建立相应的数据结构。VFS 目录树在 Linux 的文件系统模块中是个很重要的概念，希望读者不要将其与实际文件系统目录树混淆，在笔者看来，VFS 中的各目录其主要用途是用来提供实际文件系统的挂载点，当然在 VFS 中也会涉及到文件级的操作，本文不阐述这种情况。下文提到目录树或目录，如果不特别说明，均指 VFS 的目录树或目录。图 1 是一种可能的目录树在内存中的影像：

**图 1：VFS 目录树结构**



[**回页首**](https://www.ibm.com/developerworks/cn/linux/l-vfs/#ibm-pcon)

**3. 文件系统的注册**

这里的文件系统是指可能会被挂载到目录树中的各个实际文件系统，所谓实际文件系统，即是指VFS 中的实际操作最终要通过它们来完成而已，并不意味着它们一定要存在于某种特定的存储设备上。比如在笔者的 Linux 机器下就注册有 "rootfs"、"proc"、"ext2"、"sockfs" 等十几种文件系统。

**3.1 数据结构**

在 Linux 源代码中，每种实际的文件系统用以下的数据结构表示：

struct file\_system\_type {

const char \*name;

int fs\_flags;

struct super\_block \*(\*read\_super) (struct super\_block \*, void \*, int);

struct module \*owner;

struct file\_system\_type \* next;

struct list\_head fs\_supers;

};

注册过程实际上将表示各实际文件系统的 struct file\_system\_type 数据结构的实例化，然后形成一个链表，内核中用一个名为 file\_systems 的全局变量来指向该链表的表头。

**3.2 注册 rootfs 文件系统**

在众多的实际文件系统中，之所以单独介绍 rootfs 文件系统的注册过程，实在是因为该文件系统 VFS 的关系太过密切，如果说 ext2/ext3 是 Linux 的本土文件系统，那么 rootfs 文件系统则是 VFS 存在的基础。一般文件系统的注册都是通过 module\_init 宏以及 do\_initcalls() 函数来完成(读者可通过阅读module\_init 宏的声明及 arch\i386\vmlinux.lds 文件来理解这一过程)，但是 rootfs 的注册却是通过 init\_rootfs() 这一初始化函数来完成，这意味着 rootfs 的注册过程是 Linux 内核初始化阶段不可分割的一部分。

init\_rootfs() 通过调用 register\_filesystem(&rootfs\_fs\_type) 函数来完成 rootfs 文件系统注册的，其中rootfs\_fs\_type 定义如下：

struct file\_system\_type rootfs\_fs\_type = { \

name: "rootfs", \

read\_super: ramfs\_read\_super, \

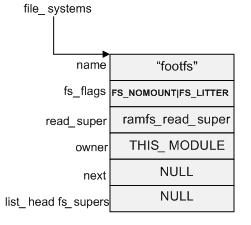
fs\_flags: FS\_NOMOUNT|FS\_LITTER, \

owner: THIS\_MODULE, \

}

注册之后的 file\_systems 链表结构如下图2所示：

**图 2: file\_systems 链表结构**



[**回页首**](https://www.ibm.com/developerworks/cn/linux/l-vfs/#ibm-pcon)

**4. VFS 目录树的建立**

既然是树，所以根是其赖以存在的基础，本节阐述 Linux 在初始化阶段是如何建立根结点的，即 "/"目录。这其中会包括挂载 rootfs 文件系统到根目录 "/" 的具体过程。构造根目录的代码是在 init\_mount\_tree（） 函数 （fs\namespace.c） 中。

首先，init\_mount\_tree() 函数会调用 do\_kern\_mount("rootfs", 0, "rootfs", NULL) 来挂载前面已经注册了的 rootfs 文件系统。这看起来似乎有点奇怪，因为根据前面的说法，似乎是应该先有挂载目录，然后再在其上挂载相应的文件系统，然而此时 VFS 似乎并没有建立其根目录。没关系，这是因为这里我们调用的是 do\_kern\_mount()，这个函数内部自然会创建我们最关心也是最关键的根目录(在 Linux 中，目录对应的数据结构是 struct dentry)。

在这个场景里，do\_kern\_mount() 做的工作主要是：

1）调用 alloc\_vfsmnt() 函数在内存里申请了一块该类型的内存空间（struct vfsmount \*mnt），并初始化其部分成员变量。

2) 调用 get\_sb\_nodev（） 函数在内存中分配一个超级块结构 (struct super\_block) sb，并初始化其部分成员变量，将成员 s\_instances 插入到 rootfs 文件系统类型结构中的 fs\_supers 指向的双向链表中。

3) 通过 rootfs 文件系统中的 read\_super 函数指针调用 ramfs\_read\_super() 函数。还记得当初注册rootfs 文件系统时，其成员 read\_super 指针指向了 ramfs\_read\_super() 函数，参见图2.

4) ramfs\_read\_super() 函数调用 ramfs\_get\_inode() 在内存中分配了一个 inode 结构 (struct inode) inode，并初始化其部分成员变量，其中比较重要的有 i\_op、i\_fop 和 i\_sb：

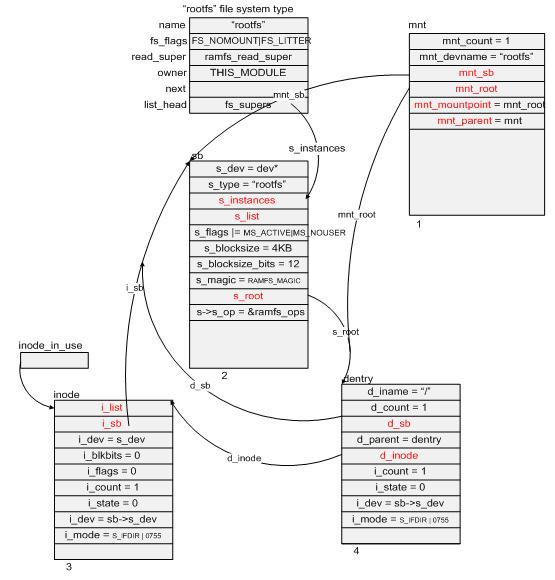
inode->i\_op = &ramfs\_dir\_inode\_operations;

inode->i\_fop = &dcache\_dir\_ops;

inode->i\_sb = sb;

这使得将来通过文件系统调用对 VFS 发起的文件操作等指令将被 rootfs 文件系统中相应的函数接口所接管。

**图3**



5) ramfs\_read\_super() 函数在分配和初始化了 inode 结构之后，会调用 d\_alloc\_root() 函数来为 VFS的目录树建立起关键的根目录 (struct dentry)dentry，并将 dentry 中的 d\_sb 指针指向 sb，d\_inode 指针指向 inode。

6) 将 mnt 中的 mnt\_sb 指针指向 sb，mnt\_root 和 mnt\_mountpoint 指针指向 dentry，而 mnt\_parent指针则指向自身。

这样，当 do\_kern\_mount() 函数返回时，以上分配出来的各数据结构和 rootfs 文件系统的关系将如上图 3 所示。图中 mnt、sb、inode、dentry 结构块下方的数字表示它们在内存里被分配的先后顺序。限于篇幅的原因，各结构中只给出了部分成员变量，读者可以对照源代码根据图中所示按图索骥，以加深理解。

最后，init\_mount\_tree() 函数会为系统最开始的进程(即 init\_task 进程)准备它的进程数据块中的namespace 域，主要目的是将 do\_kern\_mount() 函数中建立的 mnt 和 dentry 信息记录在了 init\_task 进程的进程数据块中，这样所有以后从 init\_task 进程 fork 出来的进程也都先天地继承了这一信息，在后面用sys\_mkdir 在 VFS 中创建一个目录的过程中，我们可以看到这里为什么要这样做。为进程建立 namespace 的主要代码如下：

namespace = kmalloc(sizeof(\*namespace), GFP\_KERNEL);

list\_add(&mnt->mnt\_list, &namespace->list); //mnt is returned by do\_kern\_mount()

namespace->root = mnt;

init\_task.namespace = namespace;

for\_each\_task(p) {

get\_namespace(namespace);

p->namespace = namespace;

}

set\_fs\_pwd(current->fs, namespace->root, namespace->root->mnt\_root);

set\_fs\_root(current->fs, namespace->root, namespace->root->mnt\_root);

该段代码的最后两行便是将 do\_kern\_mount() 函数中建立的 mnt 和 dentry 信息记录在了当前进程的 fs结构中。

以上讲了一大堆数据结构的来历，其实最终目的不过是要在内存中建立一颗 VFS 目录树而已，更确切地说， init\_mount\_tree() 这个函数为 VFS 建立了根目录 "/"，而一旦有了根，那么这棵数就可以发展壮大，比如可以通过系统调用 sys\_mkdir 在这棵树上建立新的叶子节点等，所以系统设计者又将 rootfs 文件系统挂载到了这棵树的根目录上。关于 rootfs 这个文件系统，读者如果看一下前面图 2 中它的file\_system\_type 结构，会发现它的一个成员函数指针 read\_super 指向的是 ramfs\_read\_super，单从这个函数名称中的 ramfs，读者大概能猜测出这个文件所涉及的文件操作都是针对内存中的数据对象，事实上也的确如此。从另一个角度而言，因为 VFS 本身就是内存中的一个数据对象，所以在其上的操作仅限于内存，那也是非常合乎逻辑的事。在接下来的章节中，我们会用一个具体的例子来讨论如何利用 rootfs所提供的函树为 VFS 增加一个新的目录节点。

VFS 中各目录的主要用途是为以后挂载文件系统提供挂载点。所以真正的文件操作还是要通过挂载后的文件系统提供的功能接口来进行。

[**回页首**](https://www.ibm.com/developerworks/cn/linux/l-vfs/#ibm-pcon)

**5. VFS 下目录的建立**

为了更好地理解 VFS，下面我们用一个实际例子来看看 Linux 是如何在 VFS 的根目录下建立一个新的目录 "/dev" 的。

要在 VFS 中建立一个新的目录，首先我们得对该目录进行搜索，搜索的目的是找到将要建立的目录其父目录的相关信息，因为"皮之不存，毛将焉附"。比如要建立目录 /home/ricard，那么首先必须沿目录路径进行逐层搜索，本例中先从根目录找起，然后在根目录下找到目录 home，然后再往下，便是要新建的目录名 ricard，那么前面讲得要先对目录搜索，在该例中便是要找到 ricard 这个新目录的父目录，也就是 home 目录所对应的信息。

当然，如果搜索的过程中发现错误，比如要建目录的父目录并不存在，或者当前进程并无相应的权限等等，这种情况系统必然会调用相关过程进行处理，对于此种情况，本文略过不提。

Linux 下用系统调用 sys\_mkdir 来在 VFS 目录树中增加新的节点。同时为配合路径搜索，引入了下面一个数据结构：

struct nameidata {

struct dentry \*dentry;

struct vfsmount \*mnt;

struct qstr last;

unsigned int flags;

int last\_type;

};

这个数据结构在路径搜索的过程中用来记录相关信息，起着类似"路标"的作用。其中前两项中的 dentry记录的是要建目录的父目录的信息，mnt 成员接下来会解释到。后三项记录的是所查找路径的最后一个节点(即待建目录或文件)的信息。 现在为建立目录 "/dev" 而调用 sys\_mkdir("/dev", 0700)，其中参数 0700 我们不去管它，它只是限定将要建立的目录的某种模式。sys\_mkdir 函数首先调用 path\_lookup("/dev", LOOKUP\_PARENT, &nd)；来对路径进行查找，其中 nd 为 struct nameidata nd 声明的变量。在接下来的叙述中，因为函数调用关系的繁琐，为了突出过程主线，将不再严格按照函数的调用关系来进行描述。

path\_lookup 发现 "/dev" 是以 "/" 开头，所以它从当前进程的根目录开始往下查找，具体代码如下：

nd->mnt = mntget(current->fs->rootmnt);

nd->dentry = dget(current->fs->root);

记得在 init\_mount\_tree() 函数的后半段曾经将新建立的 VFS 根目录相关信息记录在了 init\_task 进程的进程数据块中，那么在这个场景里，nd->mnt 便指向了图 3 中 mnt 变量，nd->dentry 便指向了图 3 中的 dentry 变量。

然后调用函数 path\_walk 接着往下查找，找到最后通过变量 nd 返回的信息是 nd.last.name="dev"，nd.last.len=3，nd.last\_type=LAST\_NORM，至于 nd 中 mnt 和 dentry 成员，在这个场景里还是前面设置的值，并无变化。这样一圈下来，只是用 nd 记录下相关信息，实际的目录建立工作并没有真正展开，但是前面所做的工作却为接下来建立新的节点收集了必要的信息。

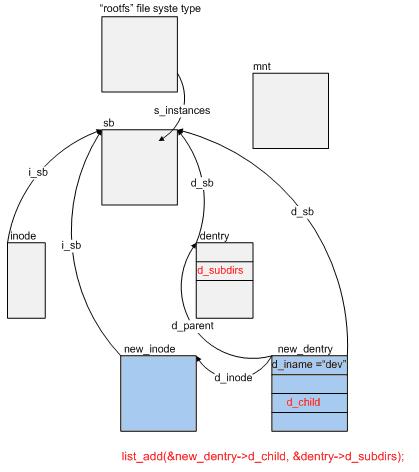
好，到此为止真正建立新目录节点的工作将会展开，这是由函数 lookup\_create 来完成的，调用这个函数时会传入两个参数：lookup\_create(&nd, 1)；其中参数 nd 便是前面提到的变量，参数1表明要建立一个新目录。

这里的大体过程是：新分配了一个 struct dentry 结构的内存空间，用于记录 dev 目录所对应的信息，该dentry 结构将会挂接到其父目录中，也就是图 3 中 "/" 目录对应的 dentry 结构中，由链表实现这一关系。接下来会再分配一个 struct inode 结构。Inode 中的 i\_sb 和 dentry 中的 d\_sb 分别都指向图 3 中的 sb，这样看来，在同一文件系统下建立新的目录时并不需要重新分配一个超级块结构，因为毕竟它们都属于同一文件系统，因此一个文件系统只对应一个超级块。

这样，当调用 sys\_mkdir 成功地在 VFS 的目录树中新建立一个目录 "/dev" 之后，在图 3 的基础上，新的数据结构之间的关系便如图 4 所示。图 4 中颜色较深的两个矩形块 new\_inode 和 new\_entry 便是在sys\_mkdir() 函数中新分配的内存结构，至于图中的 mnt,sb,dentry,inode 等结构，仍为图 3 中相应的数据结构，其相互之间的链接关系不变(图中为避免过多的链接曲线，忽略了一些链接关系，如 mnt 和 sb,dentry之间的链接，读者可在图 3 的基础上参看图 4)。

需要强调一点的是，既然 rootfs 文件系统被 mount 到了 VFS 树上，那么它在 sys\_mkdir 的过程中必然会参与进来，事实上在整个过程中，rootfs 文件系统中的 ramfs\_mkdir、ramfs\_lookup 等函数都曾被调用过。

**图 4: 在 VFS 树中新建一目录 "dev"**



[**回页首**](https://www.ibm.com/developerworks/cn/linux/l-vfs/#ibm-pcon)

**6. 在 VFS 树中挂载文件系统**

在本节中，将描述在 VFS 的目录树中向其中某个目录(安装点 mount point)上挂载(mount)一个文件系统的过程。

这一过程可简单描述为：将某一设备(dev\_name)上某一文件系统(file\_system\_type)安装到VFS目录树上的某一安装点(dir\_name)。它要解决的问题是：将对 VFS 目录树中某一目录的操作转化为具体安装到其上的实际文件系统的对应操作。比如说，如果将 hda2 上的根文件系统(假设文件系统类型为 ext2)安装到了前一节中新建立的 "/dev" 目录上(此时，"/dev" 目录就成为了安装点)，那么安装成功之后应达到这样的目的，即：对 VFS 文件系统的 "/dev" 目录执行 "ls" 指令，该条指令应能列出 hda2 上 ext2 文件系统的根目录下所有的目录和文件。很显然，这里的关键是如何将对 VFS 树中 "/dev" 的目录操作指令转化为安装在其上的 ext2 这一实际文件系统中的相应指令。所以，接下来的叙述将抓住如何转化这一核心问题。在叙述之前，读者不妨自己设想一下 Linux 系统会如何解决这一问题。记住：对目录或文件的操作将最终由目录或文件所对应的 inode 结构中的 i\_op 和 i\_fop 所指向的函数表中对应的函数来执行。所以，不管最终解决方案如何，都可以设想必然要通过将对 "/dev" 目录所对应的 inode 中 i\_op 和 i\_fop 的调用转换到 hda2 上根文件系统 ext2 中根目录所对应的 inode 中 i\_op 和 i\_fop 的操作。

初始过程由 sys\_mount() 系统调用函数发起，该函数原型声明如下：

asmlinkage long sys\_mount(char \* dev\_name, char \* dir\_name, char \* type,

unsigned long flags, void \* data);

其中，参数 char \*type 为标识将要安装的文件系统类型字符串，对于 ext2 文件系统而言，就是"ext2"。参数 flags 为安装时的模式标识数，和接下来的 data 参数一样，本文不将其做为重点。

为了帮助读者更好地理解这一过程，笔者用一个具体的例子来说明：我们准备将来自主硬盘第 2 分区(hda2)上的 ext2 文件系统安装到前面创建的 "/dev" 目录中。那么对于 sys\_mount() 函数的调用便具体为：

sys\_mount("hda2","/dev ","ext2",…)；

该函数在将这些来自用户内存空间(user space)的参数拷贝到内核空间后，便调用 do\_mount() 函数开始真正的安装文件系统的工作。同样，为了便于叙述和讲清楚主流程，接下来的说明将不严格按照具体的函数调用细节来进行。

do\_mount() 函数会首先调用 path\_lookup() 函数来得到安装点的相关信息，如同创建目录过程中叙述的那样，该安装点的信息最终记录在 struct nameidata 类型的一个变量当中，为叙述方便，记该变量为nd。在本例中当 path\_lookup() 函数返回时，nd 中记录的信息如下：nd.entry = new\_entry; nd.mnt = mnt; 这里的变量如图 3 和 4 中所示。

然后，do\_mount() 函数会根据调用参数 flags 来决定调用以下四个函数之一：do\_remount()、 do\_loopback()、do\_move\_mount()、do\_add\_mount()。

在我们当前的例子中，系统会调用 do\_add\_mount() 函数来向 VFS 树中安装点 "/dev " 安装一个实际的文件系统。在 do\_add\_mount() 中，主要完成了两件重要事情：一是获得一个新的安装区域块，二是将该新的安装区域块加入了安装系统链表。它们分别是调用 do\_kern\_mount() 函数和 graft\_tree() 函数来完成的。这里的描述可能有点抽象，诸如安装区域块、安装系统链表等，不过不用着急，因为它们都是笔者自己定义出来的概念，等一下到后面会有专门的图表解释，到时便会清楚。

do\_kern\_mount() 函数要做的事情，便是建立一新的安装区域块，具体的内容在前面的章节 VFS 目录树的建立中已经叙述过，这里不再赘述。

graft\_tree() 函数要做的事情便是将 do\_kern\_mount() 函数返回的一 struct vfsmount 类型的变量加入到安装系统链表中，同时 graft\_tree() 还要将新分配的 struct vfsmount 类型的变量加入到一个hash表中，其目的我们将会在以后看到。

这样，当 do\_kern\_mount() 函数返回时，在图 4 的基础上，新的数据结构间的关系将如图 5 所示。其中，红圈区域里面的数据结构便是被称做安装区域块的东西，其中不妨称 e2\_mnt 为安装区域块的指针，蓝色箭头曲线即构成了所谓的安装系统链表。

在把这些函数调用后形成的数据结构关系理清楚之后，让我们回到本章节开始提到的问题，即将 ext2 文件系统安装到了 "/dev " 上之后，对该目录上的操作如何转化为对 ext2 文件系统相应的操作。从图 5上看到，对 sys\_mount() 函数的调用并没有直接改变 "/dev " 目录所对应的 inode (即图中的 new\_inode变量)结构中的 i\_op 和 i\_fop 指针，而且 "/dev " 所对应的 dentry(即图中的 new\_dentry 变量)结构仍然在 VFS 的目录树中，并没有被从其中隐藏起来，相应地，来自 hda2 上的 ext2 文件系统的根目录所对应的 e2\_entry 也不是如当初笔者所想象地那样将 VFS 目录树中的 new\_dentry 取而代之，那么这之间的转化到底是如何实现的呢？

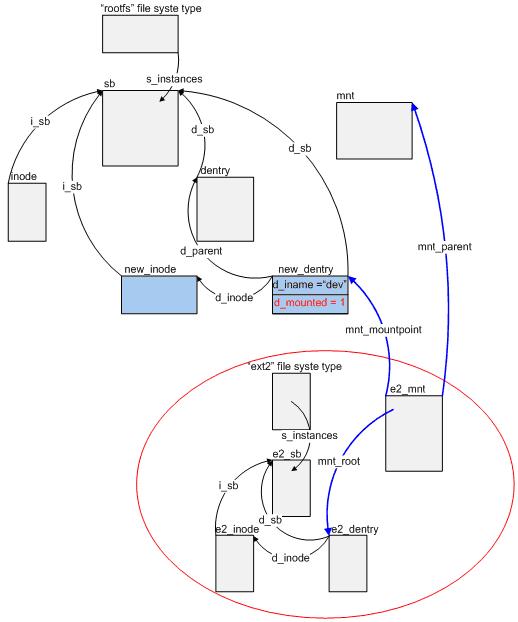
请读者注意下面的这段代码：

while (d\_mountpoint(dentry) && \_\_follow\_down(&nd->mnt, &dentry));

这段代码在 link\_path\_walk() 函数中被调用，而 link\_path\_walk() 最终又会被 path\_lookup() 函数调用，如果读者阅读过 Linux 关于文件系统部分的代码，应该知道 path\_lookup() 函数在整个 Linux 繁琐的文件系统代码中属于一个重要的基础性的函数。简单说来，这个函数用于解析文件路径名，这里的文件路径名和我们平时在应用程序中所涉及到的概念相同，比如在 Linux 的应用程序中 open 或 read 一个文件 /home/windfly.cs 时，这里的 /home/windfly.cs 就是文件路径名，path\_lookup() 函数的责任就是对文件路径名中进行搜索，直到找到目标文件所属目录所对应的 dentry 或者目标直接就是一个目录，笔者不想在有限的篇幅里详细解释这个函数，读者只要记住 path\_lookup() 会返回一个目标目录即可。

上面的代码非常地不起眼，以至于初次阅读文件系统的代码时经常会忽略掉它，但是前文所提到从 VFS 的操作到实际文件系统操作的转化却是由它完成的，对 VFS 中实现的文件系统的安装可谓功不可没。现在让我们仔细剖析一下该段代码： d\_mountpoint(dentry) 的作用很简单，它只是返回 dentry 中 d\_mounted 成员变量的值。这里的dentry 仍然还是 VFS 目录树上的东西。如果 VFS 目录树上某个目录被安装过一次，那么该值为 1。对VFS 中的一个目录可进行多次安装，后面会有例子说明这种情况。在我们的例子中，"/dev" 所对应的new\_dentry 中 d\_mounted=1，所以 while 循环中第一个条件满足。下面再来看\_\_follow\_down(&nd->mnt, &dentry)代

**图 5：安装 ext2 类型根文件系统到 "/dev " 目录上**



码做了什么？到此我们应该记住，这里 nd 中的 dentry 成员就是图 5 中的 new\_dentry，nd 中的 mnt成员就是图 5 中的 mnt，所以我们现在可以把 \_\_follow\_down(&nd->mnt, &dentry) 改写成\_\_follow\_down(&mnt, &new\_dentry)，接下来我们将 \_\_follow\_down() 函数的代码改写(只是去处掉一些不太相关的代码，并且为了便于说明，在部分代码行前加上了序号)如下：

static inline int \_\_follow\_down(struct vfsmount \*\*mnt, struct dentry \*\*dentry)

{

struct vfsmount \*mounted;

[1] mounted = lookup\_mnt(\*mnt, \*dentry);

if (mounted) {

[2] \*mnt = mounted;

[3] \*dentry = mounted->mnt\_root;

return 1;

}

return 0;

}

代码行[1]中的 lookup\_mnt() 函数用于查找一个 VFS 目录树下某一目录最近一次被 mount 时的安装区域块的指针，在本例中最终会返回图 5 中的 e2\_mnt。至于查找的原理，这里粗略地描述一下。记得当我们在安装 ext2 文件系统到 "/dev" 时，在后期会调用 graft\_tree() 函数，在这个函数里会把图 5 中的安装区域块指针 e2\_mnt 挂到一 hash 表(Linux 2.4.20源代码中称之为 mount\_hashtable)中的某一项，而该项的键值就是由被安装点所对应的 dentry(本例中为 new\_dentry)和 mount(本例中为 mnt)所共同产生，所以自然地，当我们知道 VFS 树中某一 dentry 被安装过(该 dentry 变成为一安装点)，而要去查找其最近一次被安装的安装区域块指针时，同样由该安装点所对应的 dentry 和 mount 来产生一键值，以此值去索引 mount\_hashtable，自然可找到该安装点对应的安装区域块指针形成的链表的头指针，然后遍历该链表，当发现某一安装区域块指针，记为 p，满足以下条件时：

(p->mnt\_parent == mnt && p->mnt\_mountpoint == dentry)

P 便为该安装点所对应的安装区域块指针。当找到该指针后，便将 nd 中的 mnt 成员换成该安装区域块指针，同时将 nd 中的 dentry 成员换成安装区域块中的 dentry 指针。在我们的例子中，e2\_mnt->mnt\_root成员指向 e2\_dentry，也就是 ext2 文件系统的 "/" 目录。这样，当 path\_lookup() 函数搜索到 "/dev"时，nd 中的 dentry 成员为 e2\_dentry，而不再是原来的 new\_dentry，同时 mnt 成员被换成 e2\_mnt，转化便在不知不觉中完成了。

现在考虑一下对某一安装点多次安装的情况，同样作为例子，我们假设在 "/dev" 上安装完一个 ext2文件系统后，再在其上安装一个 ntfs 文件系统。在安装之前，同样会对安装点所在的路径调用path\_lookup() 函数进行搜索，但是这次由于在 "/dev" 目录上已经安装过了 ext2 文件系统，所以搜索到最后，由 nd 返回的信息是：nd.dentry = e2\_dentry, nd.mnt = e2\_mnt。由此可见，在第二次安装时，安装点已经由 dentry 变成了 e2\_dentry。接下来，同样地，系统会再分配一个安装区域块，假设该安装区域块的指针为 ntfs\_mnt，区域块中的 dentry 为 ntfs\_dentry。ntfs\_mnt 的父指针指向了e2\_mnt，mnfs\_mnt 中的 mnt\_root 指向了代表 ntfs 文件系统根目录的 ntfs\_dentry。然后，系统通过 e2\_dentry和 e2\_mnt 来生成一个新的 hash 键值，利用该值作为索引，将 ntfs\_mnt 加入到 mount\_hashtable 中，同时将 e2\_dentry 中的成员 d\_mounted 值设定为 1。这样，安装过程便告结束。

读者可能已经知道，对同一安装点上的最近一次安装会隐藏起前面的若干次安装，下面我们通过上述的例子解释一下该过程：

在先后将 ext2 和 ntfs 文件系统安装到 "/dev" 目录之后，我们再调用 path\_lookup() 函数来对"/dev" 进行搜索，函数首先找到 VFS 目录树下的安装点 "/dev" 所对应的 dentry 和 mnt，此时它发现dentry 成员中的 d\_mounted 为 1，于是它知道已经有文件系统安装到了该 dentry 上，于是它通过 dentry 和 mnt 来生成一个 hash 值，通过该值来对 mount\_hashtable 进行搜索，根据安装过程，它应该能找到 e2\_mnt 指针并返回之，同时原先的 dentry 也已经被替换成 e2\_dentry。回头再看一下前面已经提到的下列代码： while (d\_mountpoint(dentry) && \_\_follow\_down(&nd->mnt, &dentry)); 当第一次循环结束后, nd->mnt 已经是 e2\_mnt，而 dentry 则变成 e2\_dentry。此时由于 e2\_dentry 中的成员 d\_mounted 值为 1，所以 while 循环的第一个条件满足，要继续调用 \_\_follow\_down() 函数，这个函数前面已经剖析过，当它返回后 nd->mnt 变成了 ntfs\_mnt，dentry 则变成了 ntfs\_dentry。由于此时 ntfs\_dentry 没有被安装过其他文件，所以它的成员 d\_mounted 应该为 0，循环结束。对 "/dev" 发起的 path\_lookup() 函数最终返回了 ntfs 文件系统根目录所对应的 dentry。这就是为什么 "/dev" 本身和安装在其上的 ext2 都被隐藏的原因。如果此时对 "/dev" 目录进行一个 ls 命令，将返回安装上去的 ntfs 文件系统根目录下所有的文件和目录。

[**回页首**](https://www.ibm.com/developerworks/cn/linux/l-vfs/#ibm-pcon)

**7. 安装根文件系统**

有了前面章节 5 的基础，理解 Linux 下根文件系统的安装并不困难，因为不管怎么样，安装一个文件系统到 VFS 中某一安装点的过程原理毕竟都是一样的。

这个过程大致是：首先要确定待安装的 ext2 文件系统的来源，其次是确定 ext2 文件系统在 VFS中的安装点，然后便是具体的安装过程。

关于第一问题，Linux 2.4.20 的内核另有一大堆的代码去解决，限于篇幅，笔者不想在这里去具体说明这个过程，大概记住它是要解决到哪里去找要安装的文件系统的就可以了，这里我们不妨就认为要安装的根文件系统就来自于主硬盘的第一分区 hda1.

关于第二个问题，Linux 2.4.20 的内核把来自于 hda1 上 ext2 文件系统安装到了 VFS 目录树中的"/root" 目录上。其实，把 ext2 文件系统安装到 VFS 目录树下的哪个安装点并不重要(VFS 的根目录除外)，只要是这个安装点在 VFS 树中是存在的，并且内核对它没有另外的用途。如果读者喜欢，尽可以自己在 VFS 中创建一个 "/Windows" 目录，然后将 ext2 文件系统安装上去作为将来用户进程的根目录，没有什么不可以的。问题的关键是要将进程的根目录和当前工作目录设定好，因为毕竟只用用户进程才去关心现实的文件系统，要知道笔者的这篇稿子可是要存到硬盘上去的。

在 Linux 下，设定一个进程的当前工作目录是通过系统调用 sys\_chdir() 进行的。初始化期间，Linux 在将 hda1 上的 ext2 文件系统安装到了 "/root" 上后，通过调用 sys\_chdir("/root") 将当前进程，也就是 init\_task 进程的当前工作目录(pwd)设定为 ext2 文件系统的根目录。记住此时 init\_task进程的根目录仍然是图 3 中的 dentry，也就是 VFS 树的根目录，这显然是不行的，因为以后 Linux 世界中的所有进程都由这个 init\_task 进程派生出来，无一例外地要继承该进程的根目录，如果是这样，意味着用户进程从根目录搜索某一目录时，实际上是从 VFS 的根目录开始的，而事实上却是从 ext2 的根文件开始搜索的。这个矛盾的解决是靠了在调用完 mount\_root() 函数后，系统调用的下面两个函数：

sys\_mount(".", "/", NULL, MS\_MOVE, NULL);

sys\_chroot(".");

其主要作用便是将 init\_task 进程的根目录转化成安装上去的 ext2 文件系统的根目录。有兴趣的读者可以自行去研究这一过程。

所以在用户空间下，更多地情况是只能见到 VFS 这棵大树的一叶，而且还是被安装过文件系统了的，实际上对用户空间来说还是不可见。我想，VFS 更多地被内核用来实现自己的功能，并以系统调用的方式提供过用户进程使用，至于在其上实现的不同文件系统的安装，也只是其中的一个功能罢了。

[**回页首**](https://www.ibm.com/developerworks/cn/linux/l-vfs/#ibm-pcon)

**8. 结束语**

文件系统在整个 Linux 的内核中具有举足轻重的地位，代码量也很复杂繁琐。但是因为其重要的地位，要想对 Linux 的内核有比较深入的理解，必须要能越过文件系统这一关。当然阅读其源代码便是其中最好的方法，本文试图给那些已经尝试着去阅读，但是目前尚有困惑的读者画一张 VFS 文件系统的草图，希望能对读者有些许启发。但是想在如此有限的篇幅里去阐述清楚 Linux 中整个文件系统的来龙去脉，是根本不现实的。而且本文也只是侧重于剖析 VFS 的机制，对于象具体的文件读写，为提高效率而引入的各种 buffer,hash 等内容以及文件系统的安全性方面，都没有提到。毕竟，本文只想帮助读者理清一个大体的脉络，最终的理解与领悟，还得靠读者自己去潜心研究源代码。最后，对本文相关的任何问题或建议，都欢迎用 email 和笔者联系。