Linux 虚拟文件系统（VFS）介绍

1. 通用文件模型

Linux内核支持装载不同的文件系统类型，不同的文件系统有各自管理文件的方式。Linux中标准的文件系统为Ext文件系统族，当然，开发者不能为他们使用的每种文件系统采用不同的文件存取方式，这与操作系统作为一种抽象机制背道而驰。

为支持各种文件系统，Linux内核在用户进程（或C标准库）和具体的文件系统之间引入了一个抽象层，该抽象层称之为“虚拟文件系统（VFS）”。

VFS一方面提供一种操作文件、目录及其他对象的统一方法，使用户进程不必知道文件系统的细节。另一方面，VFS提供的各种方法必须和具体文件系统的实现达成一种妥协，毕竟对几十种文件系统类型进行统一管理并不是件容易的事。

为此，VFS中定义了一个通用文件模型，以支持文件系统中对象（或文件）的统一视图。

Linux对Ext文件系统族的支持是最好的，因为VFS抽象层的组织与Ext文件系统类似，这样在处理Ext文件系统时可以提高性能，因为在Ext和VFS之间转换几乎不会损失时间。

内核处理文件的关键是**inode**，每个文件（和目录）都有且只有一个对应的inode（struct inode实例），其中包含元数据和指向文件数据的指针，但inode并不包含文件名。系统中所有的inode都有一个特定的编号，用于唯一的标识各个inode。文件名可以随时更改，但是索引节点对文件是唯一的，并且随文件的存在而存在。

对于每个已经挂载的文件系统，VFS在内核中都生成一个超级块结构（struct **super_block**实例），超级块代表一个已经安装的文件系统，用于存储文件系统的控制信息，例如文件系统类型、大小、所有inode对象、脏的inode链表等。

inode和super block在存储介质中都是有实际映射的，即存储介质中也存在超级块和inode。但是由于不同类型的文件系统差异，超级块和inode的结构不尽相同。而VFS的作用就是通过具体的设备驱动获得某个文件系统中的超级块和inode节点，然后将其中的信息填充到内核中的struct super_block和struct inode中，以此来试图对不同文件系统进行统一管理。

由于块设备速度较慢（于内存而言），可能需要很长时间才能找到与一个文件名关联的inode。Linux使用目录项（**dentry**）缓存来快速访问此前的查找操作结果。在VFS读取了一个目录或文件的数据之后，则创建一个dentry实例（struct dentry），以缓存找到的数据。

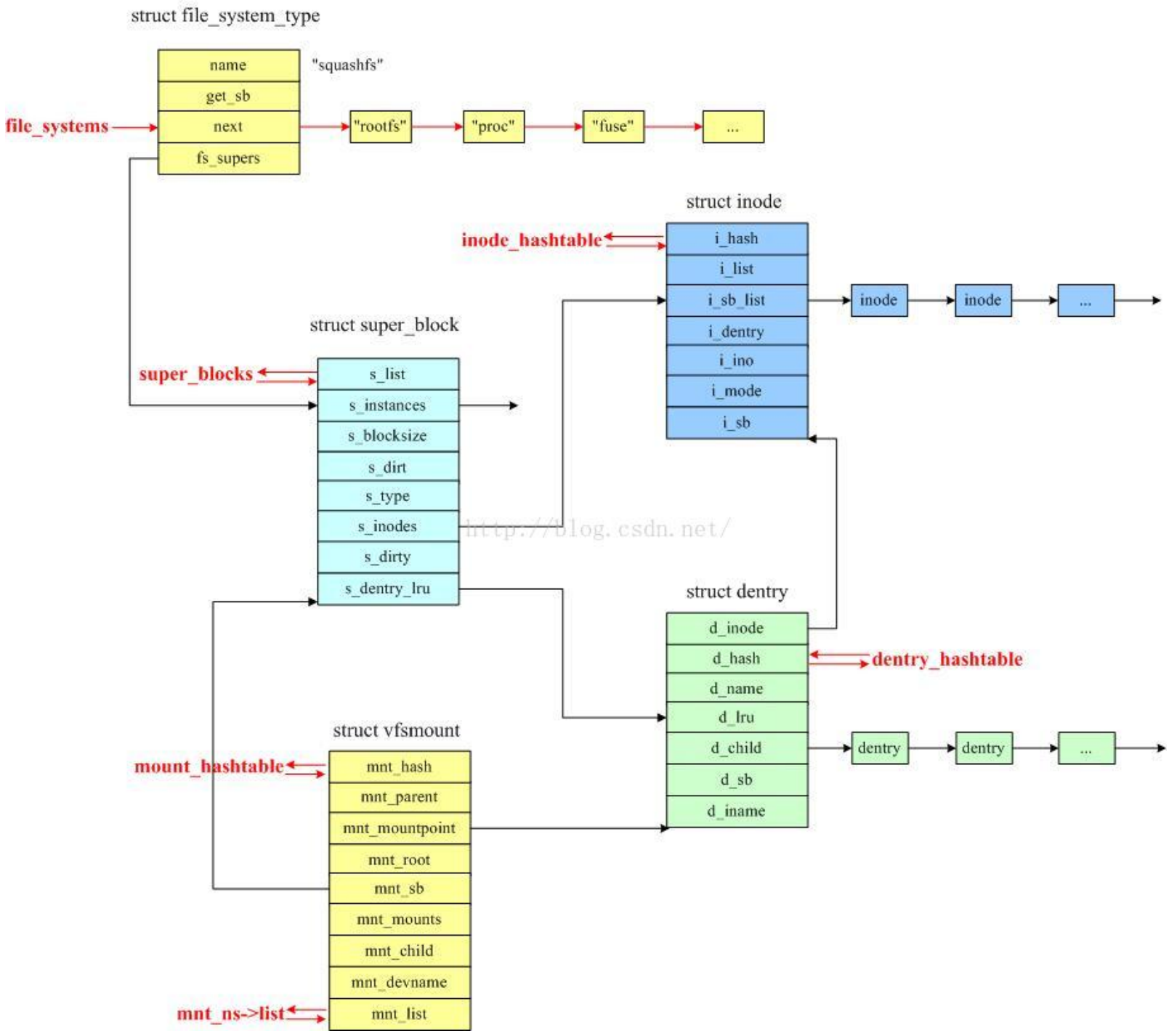
dentry结构的主要用途就是建立文件名和相关的inode之间的联系。一个文件系统中的dentry对象都被放在一个散列表中，同时不再使用的dentry对象被放到超级块指向的一个LRU链表中，在某个时间点会删除比较老的对象以释放内存。

另外简单提一下两个数据结构：

每种注册到内核的文件系统类型以struct file_system_type结构表示，每种文件系统类型中都有一个链表，指向所有属于该类型的文件系统的超级块。

当一个文件系统挂载到内核文件系统的目录树上，会生成一个挂载点，用来管理所挂载的文件系统的信息。该挂载点用一个struct vfsmount结构表示，这个结构后面会提到。

上面的这些结构的关系大致如下：



其中红色字体的链表为内核中的全局链表。

2. 挂载文件系统

在用户程序中，使用mount系统调用来挂载文件系统，相应的使用umount卸载文件系统。当然，内核必须支持将要挂载的文件系统类型，在内核启动时或者在安装内核模块时，可以注册特定的文件系统类型到内核，注册的函数为register_filesystem()。

mount命令最常用的方式是mount [-t fstype] something somewhere

其中something是将要被挂载的设备或目录，somewhere指明要挂载到何处。-t选项指明挂载的文件系统类型。由于something指向的设备是一个已知设备，即其上的文件系统类型是确定的，所以-t选项必须设置正确才能挂载成功。

每个装载的文件系统都对应一个vfsmount结构的实例。

由于装载过程是向内核文件系统目录树中添加装载点，这些装载点就存在一种父子关系，这和父目录与子目录的关系类似。例如，我的根文件系统类型是squashfs，装载到根目录“/”，生成一个挂载点，之后我又在/tmp目录挂载了ramfs文件系统，在根文件系统中的tmp目录生成了一个挂载点，这两个挂载点就是父子关系。这种关系存储在struct vfsmount结构中。

在下图中，根文件系统为squashfs，根目录为“/”，然后创建/tmp目录，并挂载为ramfs，之后又创建了/tmp/usbdisk/volume9和/tmp/usbdisk/volume1两个目录，并将/tmp/dev/sda1和/tmp/dev/sdb1两个分区挂载到这两个目录上。其中/tmp/dev/sda1设备上有如下文件：

gccbacktrace/

----> gcc_backtrace.c

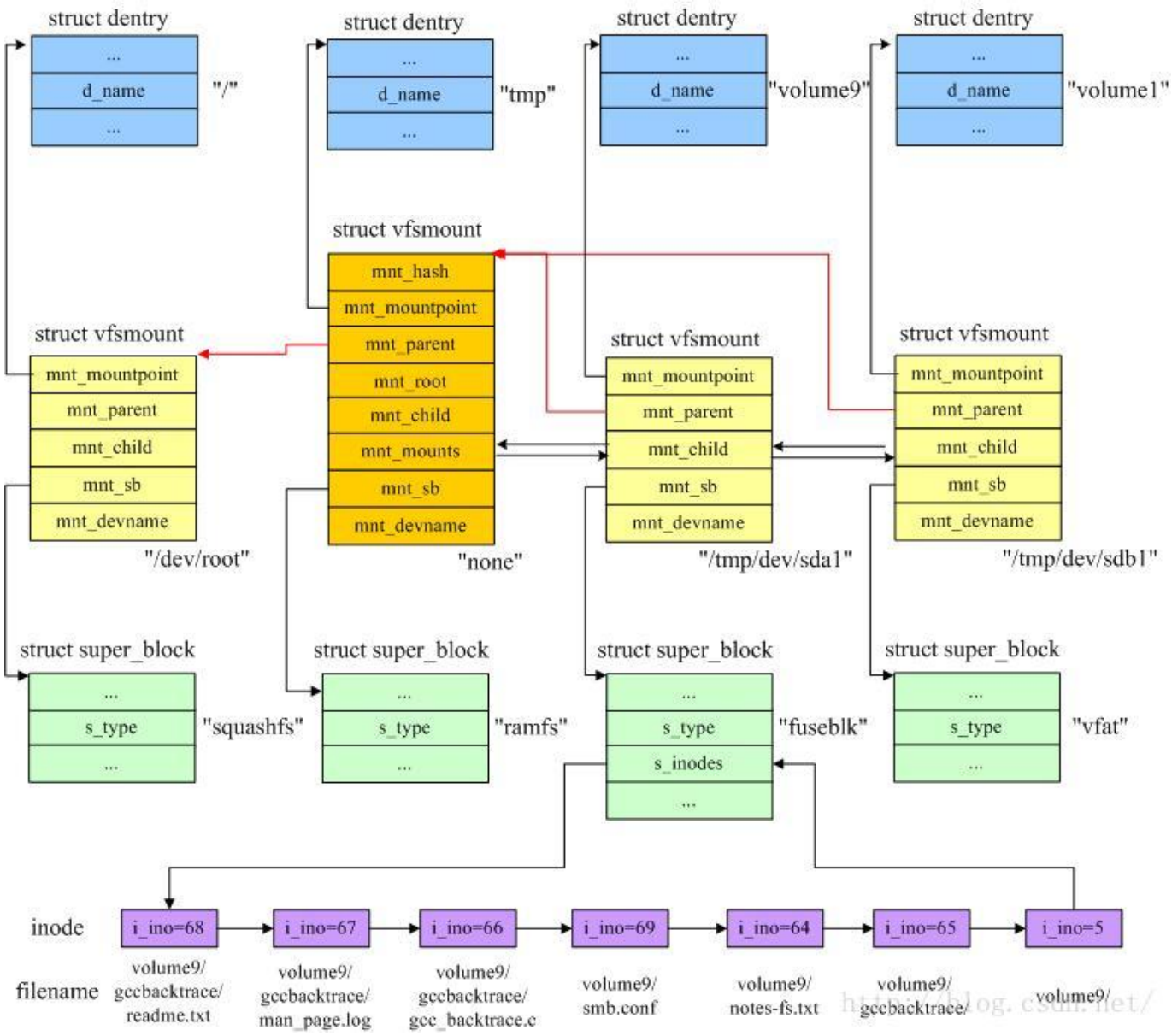
---->man_page.log

---->readme.txt

notes-fs.txt

smb.conf

挂载完成后，VFS中相关的数据结构的关系如图所示。



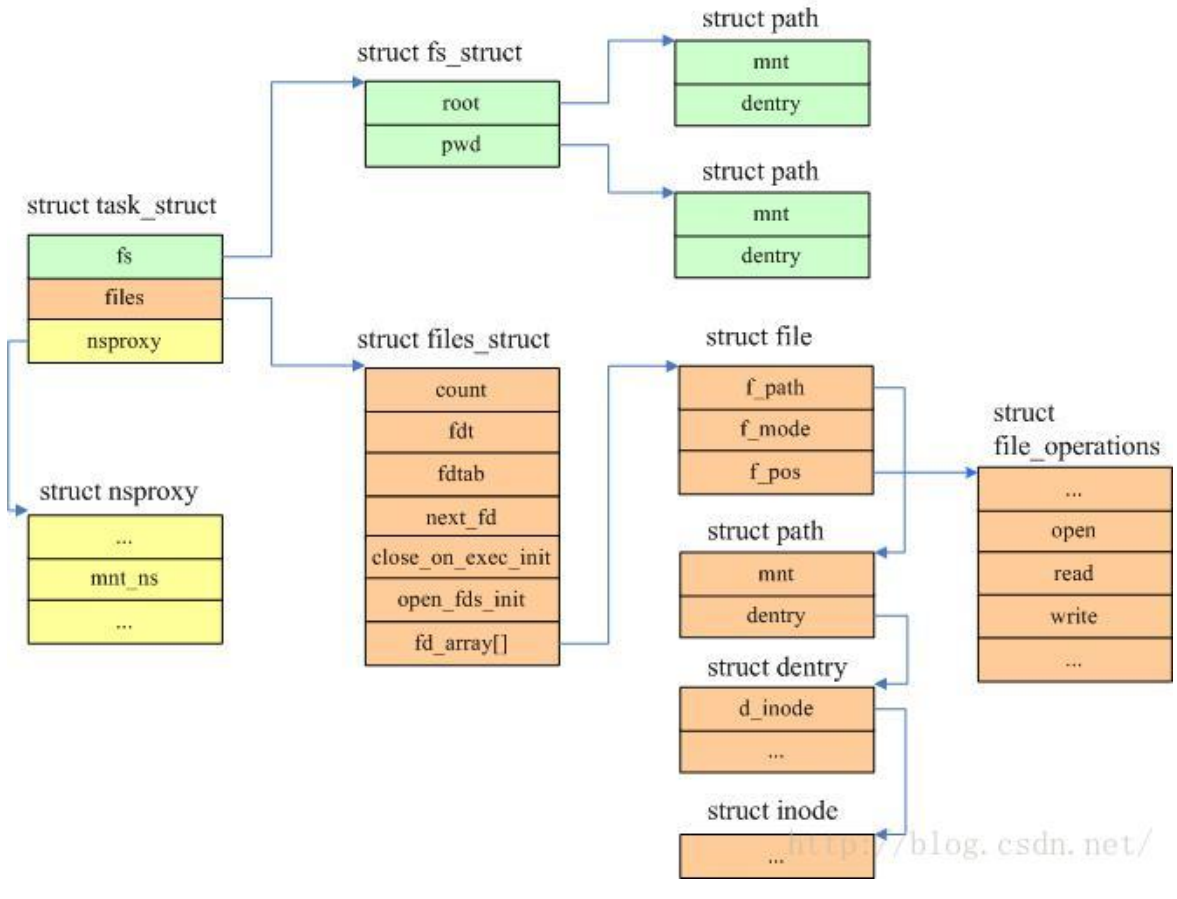
mount系统调用在内核中的入口点是sys_mount函数，该函数将装载的选项从用户态复制一份，然后调用do_mount()函数进行挂载，这个函数做的事情就是通过特定文件系统读取超级块和inode信息，然后建立VFS的数据结构并建立上图中的关系。

在父文件系统中的某个目录上挂载另一个文件系统后，该目录原来的内容就被隐藏了。例如，/tmp/samba/是非空的，然后，我将/tmp/dev/sda1挂载到/tmp/samba上，那这时/tmp/samba/目录下就只能看到/tmp/dev/sda1设备上的文件，直到将该设备卸载，原来目录中的文件才会显示出来。这是通过struct vfsmount中的mnt_mountpoint和mnt_root两个成员来实现的，这两个成员分别保存了在父文件系统中挂载点的dentry和在当前文件系统中挂载点的dentry，在卸载当前挂载点之后，可以找回挂载目录在父文件系统中的dentry对象。

3. 一个进程中与文件系统相关的信息

```
struct task_struct {
        ……
        /* filesystem information */
        struct fs_struct *fs;
        /* open file information */
        struct files_struct *files;
        /* namespaces */
        struct nsproxy *nsproxy;
        ……
}
```

其中fs成员指向进程当前工作目录的文件系统信息。files成员指向了进程打开的文件的信息。nsproxy指向了进程所在的命名空间，其中包含了虚拟文件系统命名空间。



从上图可以看到，fs中包含了文件系统的挂载点和挂载点的dentry信息。而files指向了一系列的struct file结构，其中struct path结构用于将struct file和vfsmount以及dentry联系起来。struct file保存了内核所看到的文件的特征信息，进程打开的文件列表就存放在task_struct->files->fd_array[]数组以及fdtable中。

task_struct结构还存放了其打开文件的文件描述符fd的信息，这是用户进程需要用到的，用户进程在通过文件名打开一个文件后，文件名就没有用处了，之后的操作都是对文件描述符fd的，在内核中，fget_light()函数用于通过整数fd来查找对应的struct file对象。由于每个进程都维护了自己的fd列表，所以不同进程维护的fd的值可以重复，例如标准输入、标准输出和标准错误对应的fd分别为0、1、2。

struct file的mapping成员指向属于文件相关的inode实例的地址空间映射，通常它设置为inode->i_mapping。在读写一个文件时，每次都从物理设备上获取文件的话，速度会很慢，在内核中对每个文件分配一个地址空间，实际上是这个文件的数据缓存区域，在读写文件时只是操作这块缓存，通过内核有相应的同步机制将脏的页写回物理设备。super_block中维护了一个脏的inode的链表。

struct file的f_op成员指向一个struct file_operations实例（图中画错了，不是f_pos），该结构保存了指向所有可能文件操作的指针，如read/write/open等。

```
struct file_operations {
        struct module *owner;
        loff_t (*llseek) (struct file *, loff_t, int);
        ssize_t (*read) (struct file *, char __user *, size_t, loff_t *);
        ssize_t (*write) (struct file *, const char __user *, size_t, loff_t *);
        ssize_t (*aio_read) (struct kiocb *, const struct iovec *, unsigned long, loff_t);
        ssize_t (*aio_write) (struct kiocb *, const struct iovec *, unsigned long, loff_t);
        int (*readdir) (struct file *, void *, filldir_t);
        unsigned int (*poll) (struct file *, struct poll_table_struct *);
        int (*ioctl) (struct inode *, struct file *, unsigned int, unsigned long);
        long (*unlocked_ioctl) (struct file *, unsigned int, unsigned long);
        long (*compat_ioctl) (struct file *, unsigned int, unsigned long);
        int (*mmap) (struct file *, struct vm_area_struct *);
        int (*open) (struct inode *, struct file *);
        int (*flush) (struct file *, fl_owner_t id);
        int (*release) (struct inode *, struct file *);
        int (*fsync) (struct file *, struct dentry *, int datasync);
        ……
};
```