# 与VFS对接设计

## 1.基本概述

VFS为了提供对不同底层文件系统的统一接口，需要有一个高度的抽象和建模，这就是VFS的核心设计——统一文件模型。目前的Linux系统的VFS都是源于Unix家族，因此这里所说的VFS对所有Unix家族的系统都适用。Unix家族的VFS的文件模型定义了四种对象，这四种对象构建起了**统一文件模型**。

superblock：存储文件系统基本的元数据。如文件系统类型、大小、状态，以及其他元数据相关的信息（元元数据）

index node（inode）：保存一个文件相关的元数据。包括文件的所有者（用户、组）、访问时间、文件类型等，但不包括这个文件的名称。文件和目录均有具体的inode对应

directory entry（dentry）：保存了文件（目录）名称和具体的inode的对应关系，用来粘合二者，同时可以实现目录与其包含的文件之间的映射关系。另外也作为缓存的对象，缓存最近最常访问的文件或目录，提示系统性能

file：一组逻辑上相关联的数据，被一个进程打开并关联使用

统一文件模型是一个标准，各种具体文件系统的实现必须以此模型定义的各种概念来实现。

我们要做的就是根据统一文件模型实现文件系统的VFS层。

## 2.实现方法

文件系统的vfs实现需要经过以下四个步骤，依次为文件系统的注册、文件系统的安装、VFS目录树的建立和在VFS挂载文件系统。

### 2.1.文件系统的注册。

一个具体的文件系统必须先向vfs注册，才能被使用。通过register\_filesystem() ，可以将一个“文件系统类型”结构 file\_system\_type注册到内核中一个全局的链表file\_systems 上。实际上我们需要做的就是实现file\_system\_type结构体，实现后，进行注册，此时，文件系统能被系统识别，但是不能使用，因为其还没有被安装到某个特定挂载点上。文件系统注册的主要目的，就是让VFS创建该文件系统的“超级块“结构。

以下是文件系统内核中的*file\_system\_type数据结构：*

struct file\_system\_type {

const char \*name;

int fs\_flags;

int (\*get\_sb) (struct file\_system\_type \*, int,

const char \*, void \*, struct vfsmount \*);

void (\*kill\_sb) (struct super\_block \*);

struct module \*owner;

struct file\_system\_type \* next;

struct list\_head fs\_supers; /\*超级块对象链表\*/

struct lock\_class\_key s\_lock\_key;

struct lock\_class\_key s\_umount\_key;

struct lock\_class\_key i\_lock\_key;

struct lock\_class\_key i\_mutex\_key;

struct lock\_class\_key i\_mutex\_dir\_key;

struct lock\_class\_key i\_alloc\_sem\_key;

};

这个结构中最关键的就是 get\_sb() 这个函数指针，它就是用于创建并设置 super\_block 的目的地。  
 因为安装一个文件系统的关键一步就是要为“被安装设备”创建和设置一个 super\_block，而不同的具体的文件系统的 super\_block 有自己特定的信息，因此要求具体的文件系统首先向内核注册，并提供 read\_super() 的实现。

### 2.2.文件系统的安装

一个注册了的文件系统必须经过安装才能被VFS所接受。安装一个文件系统，必须指定一个目录作为安装点。一个设备可以同时被安装到多个目录上。 一个目录节点下可以同时安装多个设备。

#### 2.2.1.“根安装点“、”根设备“、”根文件系统“

安装一个文件系统，除了需要“被安装设备”外，还要指定一个“安装点”。“安装点”是已经存在的一个目录节点。例如把 /dev/sda1 安装到 /mnt/win 下，那么 /mnt/win 就是“安装点”。 可是文件系统要先安装后使用。因此，要使用 /mnt/win 这个“安装点”，必然要求它所在文件系统已也经被安装。 也就是说，安装一个文件系统，需要另外一个文件系统已经被安装。

**这是一个鸡生蛋，蛋生鸡的问题：最顶层的文件系统是如何被安装的？**

答案是，最顶层文件系统在内核初始化的时候被安装在“根安装点”上的，而根安装点不属于任何文件系统，它对应的 dentry 、inode 等结构是由内核在初始化阶段构造出来的。

#### 2.2.2。安装连接件vfsmount

“安装”一个文件系统涉及“被安装设备”和“安装点”两个部分，安装的过程就是把“安装点”和“被安装设备”关联起来，这是通过一个“安装连接件”结构 vfsmount 来完成的。  
 vfsmount 将“安装点”dentry 和“被安装设备”的根目录节点 dentry 关联起来。  
 所以，在安装文件系统时，内核的主要工作就是：  
 1、 创建一个 vfsmount  
 2、 为“被安装设备”创建一个 super\_block，并由具体的文件系统来设置这个 super\_block。  
 3、 为被安装设备的根目录节点创建 dentry  
 4、 为被安装设备的根目录节点创建 inode， 并由 super\_operations->read\_inode() 来设置此 inode  
 5、 将 super\_block 与“被安装设备“根目录节点 dentry 关联起来  
 6、 将 vfsmount 与“被安装设备”的根目录节点 dentry 关联起来

以下是内核中的代码：

int get\_sb\_single(struct file\_system\_type \*fs\_type,

int flags, void \*data,

int (\*fill\_super)(struct super\_block \*, void \*, int),

struct vfsmount \*mnt)

{

struct super\_block \*s;

int error;

s = sget(fs\_type, compare\_single, set\_anon\_super, NULL);

if (IS\_ERR(s))

return PTR\_ERR(s);

if (!s->s\_root) {

s->s\_flags = flags;

error = fill\_super(s, data, flags & MS\_SILENT ? 1 : 0);

if (error) {

deactivate\_locked\_super(s);

return error;

}

s->s\_flags |= MS\_ACTIVE;

} else {

do\_remount\_sb(s, flags, data, 0);

}

simple\_set\_mnt(mnt, s);

return 0;

}

这个函数中的fill\_super是个函数指针，是由我们自己实现的，去填充super\_block,并且为被安装设备的根目录分配inode和dentry。最后通过simple\_set\_mnt()函数将super和dentry与vfsmount连接起来。（这样做的目的就是为了后面查找文件）

#### 2.2.3.寻找目标节点

VFS 中一个最关键以及最频繁的操作，就是根据路径名寻找目标节点的 dentry 以及 inode 。  
 例如要打开 /mnt/win/dir1/abc 这个文件，就是根据这个路径，找到‘abc’ 对应的 dentry ，进而得到 inode 的过程。  
 1、首先找到根文件系统的根目录节点 dentry 和 inode  
 2、由这个 inode 提供的操作接口 i\_op->lookup()，找到下一层节点 ‘mnt’ 的 dentry 和 inode  
 3、由 ‘mnt’ 的 inode 找到 ‘win’ 的 dentry 和 inode  
 4、由于 ‘win’ 是个“安装点”，因此需要找到“被安装设备”/dev/sda1 根目录节点的 dentry 和 inode，只要找到 vfsmount B，就可以完成这个任务。  
 5、然后由 /dev/sda1 根目录节点的 inode 负责找到下一层节点 ‘dir1’ 的 dentry 和 inode  
 6、由于 dir1 是个“安装点”，因此需要借助 vfsmount C 找到 /dev/sda2 的根目录节点 dentry 和 inode  
 7、最后由这个 inode 负责找到 ‘abc’ 的 dentry 和 inode

### 2.3.文件的读写

一个文件每被打开一次，就对应着一个 file 结构。 我们知道，每个文件对应着一个 dentry 和 inode，每打开一个文件，只要找到对应的 dentry 和 inode 不就可以了么？为什么还要引入这个 file 结构？

这是因为一个文件可以被同时打开多次，每次打开的方式也可以不一样。 而dentry 和 inode 只能描述一个物理的文件，无法描述“打开”这个概念。

因此有必要引入 file 结构，来描述一个“被打开的文件”。每打开一个文件，就创建一个 file 结构。

**实际上，打开文件的过程正是建立file, dentry, inode 之间的关联的过程。**

文件一旦被打开，数据结构之间的关系已经建立，后面对文件的读写以及其它操作都变得很简单。就是根据 fd 找到 file 结构，然后找到 dentry 和 inode，最后通过 inode->i\_fop 中对应的函数进行具体的读写等操作即可。

### 2.4.范例

**源文件：**

#include <linux/module.h>

#include <linux/string.h>

#include <linux/fs.h>

#include <linux/time.h>

#include <linux/slab.h>

#include <linux/init.h>

#include <linux/blkdev.h>

#include <linux/parser.h>

#include <linux/smp\_lock.h>

#include <linux/buffer\_head.h>

#include <linux/exportfs.h>

#include <linux/vfs.h>

#include <linux/random.h>

#include <linux/mount.h>

#include <linux/namei.h>

#include <linux/quotaops.h>

#include <linux/seq\_file.h>

#include <asm/uaccess.h>

//mount -t wzjfs /root/t1/ /root/t1/

MODULE\_LICENSE("GPL");

MODULE\_AUTHOR("wzj");

#define wzjfs\_MAGIC 0x19980122

static DEFINE\_RWLOCK(file\_systems\_lock);

static struct inode \*wzjfs\_make\_inode(struct super\_block \*sb, int mode)

{

struct inode \*ret = new\_inode(sb);

if (ret) {

ret->i\_mode = mode;

ret->i\_uid = ret->i\_gid = 0;

ret->i\_blocks = 0;

ret->i\_atime = ret->i\_mtime = ret->i\_ctime = CURRENT\_TIME;

}

return ret;

}

static int wzjfs\_open(struct inode \*inode, struct file \*filp)

{

filp->private\_data = inode->i\_private;

return 0;

}

#define TMPSIZE 20

static ssize\_t wzjfs\_read\_file(struct file \*filp, char \*buf,

size\_t count, loff\_t \*offset)

{

atomic\_t \*counter = (atomic\_t \*) filp->private\_data;

int v, len;

char tmp[TMPSIZE];

v = atomic\_read(counter);

if (\*offset > 0)

v -= 1;

else

atomic\_inc(counter);

len = snprintf(tmp, TMPSIZE, "%d\n", v);

if (\*offset > len)

return 0;

if (count > len - \*offset)

count = len - \*offset;

if (copy\_to\_user(buf, tmp + \*offset, count))

return -EFAULT;

\*offset += count;

return count;

}

static ssize\_t wzjfs\_write\_file(struct file \*filp, const char \*buf,

size\_t count, loff\_t \*offset)

{

atomic\_t \*counter = (atomic\_t \*) filp->private\_data;

char tmp[TMPSIZE];

if (\*offset != 0)

return -EINVAL;

if (count >= TMPSIZE)

return -EINVAL;

memset(tmp, 0, TMPSIZE);

if (copy\_from\_user(tmp, buf, count))

return -EFAULT;

atomic\_set(counter, simple\_strtol(tmp, NULL, 10));

return count;

}

static struct file\_operations wzjfs\_file\_ops = {

.open = wzjfs\_open,

.read = wzjfs\_read\_file,

.write = wzjfs\_write\_file,

};

static struct dentry \*wzjfs\_create\_file (struct super\_block \*sb,

struct dentry \*dir, const char \*name,

atomic\_t \*counter)

{

struct dentry \*dentry;

struct inode \*inode;

struct qstr qname;

qname.name = name;

qname.len = strlen (name);

qname.hash = full\_name\_hash(name, qname.len);

dentry = d\_alloc(dir, &qname);

if (! dentry)

goto out;

inode = wzjfs\_make\_inode(sb, S\_IFREG | 0644);

if (! inode)

goto out\_dput;

inode->i\_fop = &wzjfs\_file\_ops;

inode->i\_private = counter;

d\_add(dentry, inode);

return dentry;

out\_dput:

dput(dentry);

out:

return 0;

}

static struct dentry \*wzjfs\_create\_dir (struct super\_block \*sb,

struct dentry \*parent, const char \*name)

{

struct dentry \*dentry;

struct inode \*inode;

struct qstr qname;

qname.name = name;

qname.len = strlen (name);

qname.hash = full\_name\_hash(name, qname.len);

//dentry的主要作用是建立文件名和inode之间的关联。

/\*所以该结构体包括两个最主要的字段，d\_inode和d\_name。

其中，d\_name为文件名。qstr是内核对字符串的封装（可以理解为带有散列值的char\*）。

d\_inode是与该文件名对应的inode。\*/

dentry = d\_alloc(parent, &qname);

if (! dentry)

goto out;

inode = wzjfs\_make\_inode(sb, S\_IFDIR | 0644);

if (! inode)

goto out\_dput;

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

inode->i\_fop = &simple\_dir\_operations;

d\_add(dentry, inode);

return dentry;

out\_dput:

dput(dentry);

out:

return 0;

}

static atomic\_t counter, subcounter;

static void wzjfs\_create\_files (struct super\_block \*sb, struct dentry \*root)

{

struct dentry \*subdir;

atomic\_set(&counter, 0);

wzjfs\_create\_file(sb, root, "counter", &counter);

atomic\_set(&subcounter, 0);

subdir = wzjfs\_create\_dir(sb, root, "subdir");

if (subdir)

wzjfs\_create\_file(sb, subdir, "subcounter", &subcounter);

}

static struct super\_operations wzjfs\_s\_ops = {

.statfs = simple\_statfs,

.drop\_inode = generic\_delete\_inode,

};

static int wzjfs\_fill\_super (struct super\_block \*sb, void \*data, int silent)

{

struct inode \*root;

struct dentry \*root\_dentry;

sb->s\_blocksize = PAGE\_CACHE\_SIZE;

sb->s\_blocksize\_bits = PAGE\_CACHE\_SHIFT;

sb->s\_magic = wzjfs\_MAGIC;

sb->s\_op = &wzjfs\_s\_ops;

printk(KERN\_INFO "wzjfs\_fill\_super is here\n");

root = wzjfs\_make\_inode (sb, S\_IFDIR | 0755);

if (! root)

goto out;

root->i\_op = &simple\_dir\_inode\_operations;

root->i\_fop = &simple\_dir\_operations;

root\_dentry = d\_alloc\_root(root);

if (! root\_dentry)

goto out\_iput;

sb->s\_root = root\_dentry;

wzjfs\_create\_files (sb, root\_dentry);

return 0;

out\_iput:

iput(root);

out:

return -ENOMEM;

}

static int wzjfs\_get\_super(struct file\_system\_type \*fst,int flags, const char \*devname, void \*data,struct vfsmount \*mount)

{

printk(KERN\_INFO "mount from user\n");

return get\_sb\_single(fst, flags, data, wzjfs\_fill\_super,mount);

}

static struct file\_system\_type wzjfs\_type = {

.owner = THIS\_MODULE,

.name = "wzjfs",

.get\_sb = wzjfs\_get\_super,

.kill\_sb = kill\_litter\_super,

};

static int \_\_init wzjfs\_init(void)

{

struct file\_system\_type \* tmp;

printk("wzjfs\_init ok\n");

return register\_filesystem(&wzjfs\_type);

}

static void \_\_exit wzjfs\_exit(void)

{

unregister\_filesystem(&wzjfs\_type);

printk("wzjfs\_exit ok\n");

}

module\_init(wzjfs\_init);

module\_exit(wzjfs\_exit);

**Makefile：**

ifneq ($(KERNELRELEASE),)

obj-m := wzj.o

else

KERNELDIR ?= /lib/modules/$(shell uname -r)/build

PWD := $(shell pwd)

default:

$(MAKE) -C $(KERNELDIR) M=$(PWD) modules

endif

clean:

rm -rf \*.o \*~ core .depend .\*.cmd \*.ko \*.mod.c .tmp\_versions \*.order \*.symvers \*.unsigned

insmod 模块后，执行mount -t wzjfs /root/t1 /root/t1(目录自己指定)，再 去你指定的目录下，你会发现属于自己文件系统的文件。当你在看上面的代码有想不通的时候，看看上面的知识点，或许就会明白了。

## 3. file\_system\_type实现设计

### 3.1.file\_system\_type需要实现哪些接口

### 3.2上述接口实现设计

参考网址：

1、vfs概述和数据结构介绍：[linux的VFS详解\_jinking01的专栏-CSDN博客\_vfs](https://blog.csdn.net/jinking01/article/details/90669534)

2、Linux环境下自制文件系统的VFS实现：[Linux环境下自制文件系统的VFS实现\_爱学术 (ixueshu.com)](https://www.ixueshu.com/document/9ccb7b1c7a71e8ffd1736dd6fe2c5e61318947a18e7f9386.html)

3、基于vfs实现自己的文件系统：[基于vfs实现自己的文件系统 - 开发者知识库 (itdaan.com)](https://www.itdaan.com/blog/2016/04/07/2945b4320443b0ca9d67da4a4af103d0.html)