“一切皆文件” linux 中的至理名言。对于 linux 网络子系统，依然适用。sockfs 就是最直接的证明。接下来我们分析 linux 网络子系统的基石--sockfs。后续我们会基于 sockfs 来对 linux 网络子系统做全面深入的分析。

分析任何东西，都要先找到他的入口。net/socket.c--sock_init()就是我们需要分析的网络子系统的入口。

```
static int __init sock_init(void)
{
        skb_init(); //初始化 skbuff 的 slab cache  名为 skbuff_head_cache  skb_init 定义与 net/core/skbuff.c 中
                                //对 slab cache 不了解的同学可以参考下深入 linux 内核架构中内存管理的相关章节
        init_inodecache();// 初始化 socket_alloc 类型的 slab cache cache 名为 sock_inode_cache
                                // 返回地址存储在全局变量 sock_inode_cachep 指针中
        err = register_filesystem(&sock_fs_type); //注册 sockfs 文件系统 注册原理同其他文件系统注册一直，都是将
                                //对应的 file_system_type 变量插入全局变量 file_systems 指向的链表中
        sock_mnt = kern_mount(&sock_fs_type); //执行文件系统挂在操作，接下来重点讨论
        ptp_classifier_init();//后续再分析
}
```

以上是 sockfs 初始化并挂载的关键步骤，省略部分代码。也是网络子系统的初始化。在 sockfs 注册过程用到了 sock_fs_type 变量。那么我们看下其定义及其结构的定义：

```
net/socket.c
static struct file_system_type sock_fs_type = {
        .name = "sockfs",
        .mount = sockfs_mount,
        .kill_sb = kill_anon_super,
};

struct file_system_type {
        const char *name;
        int fs_flags;
        #define FS_REQUIRES_DEV 1
        #define FS_BINARY_MOUNTDATA 2
        #define FS_HAS_SUBTYPE 4
        #define FS_USERNS_MOUNT 8 /* Can be mounted by userns root */
        #define FS_RENAME_DOES_D_MOVE 32768 /* FS will handle d_move() during rename() internally. */
        struct dentry *(*mount) (struct file_system_type *, int, const char *, void *);
        void (*kill_sb) (struct super_block *);
```

```
        struct module *owner;
        struct file_system_type * next;
        struct hlist_head fs_supers;
}
```

注册过程比较简单，就是向 file_systems 链表中插入 file_system_type 变量。接下来分析 kern_mount 挂载过程。

```
include/linux/fs.h
#define kern_mount(type) kern_mount_data(type, NULL) //宏定义

fs/namespace.c
最终实现：
struct vfsmount *kern_mount_data(struct file_system_type *type, void *data)
{
    struct vfsmount *mnt;
    mnt = vfs_kern_mount(type, MS_KERNMOUNT, type->name, data);
    if (!IS_ERR(mnt)) {
    /*
     * it is a longterm mount, don't release mnt until
     * we unmount before file sys is unregistered
     */
        real_mount(mnt)->mnt_ns = MNT_NS_INTERNAL;
    }
    return mnt;
}

调用 vfs_kern_mount()
在 fs/namespace.c 文件中定义
实现：
  struct vfsmount *
  vfs_kern_mount(struct file_system_type *type, int flags, const char *name, void
*data)
  {
        struct mount *mnt;
        struct dentry *root;

        if (!type)
            return ERR_PTR(-ENODEV);

        mnt = alloc_vfsmnt(name); //分配 mount 结构，并初始化
        if (!mnt)
            return ERR_PTR(-ENOMEM);
```

```
        if (flags & MS_KERNMOUNT)
            mnt->mnt.mnt_flags = MNT_INTERNAL; //设置内部挂载标志

        root = mount_fs(type, flags, name, data);
        if (IS_ERR(root)) {
                mnt_free_id(mnt);
                free_vfsmnt(mnt);
                return ERR_CAST(root);
          }

        mnt->mnt.mnt_root = root;
        mnt->mnt.mnt_sb = root->d_sb;
        mnt->mnt_mountpoint = mnt->mnt.mnt_root;
        mnt->mnt_parent = mnt;
        lock_mount_hash();
        list_add_tail(&mnt->mnt_instance, &root->d_sb->s_mounts);
        unlock_mount_hash();
        return &mnt->mnt;
    }
```

这一部分代码比较简单，就是分配 mount(挂载点)结构，同时调用 mount_fs 执行挂载操作，如下

```
    // fs/super.c 中定义
    struct dentry *
    mount_fs(struct file_system_type *type, int flags, const char *name, void *data)
    {
            struct dentry *root;
            struct super_block *sb;
            char *secdata = NULL;
            int error = -ENOMEM;

            if (data && !(type->fs_flags & FS_BINARY_MOUNTDATA)) {
                secdata = alloc_secdata();
                if (!secdata)
                 goto out;

                error = security_sb_copy_data(data, secdata);
                if (error)
                     goto out_free_secdata;
            }

            root = type->mount(type, flags, name, data);// 调用 sock_fs_type 成员 mount 即 sockfs_mount（）函数
            if (IS_ERR(root)) {
```

```
                error = PTR_ERR(root);
                goto out_free_secdata;
            }
            sb = root->d_sb;
            BUG_ON(!sb);
            WARN_ON(!sb->s_bdi);
            sb->s_flags |= MS_BORN;

            error = security_sb_kern_mount(sb, flags, secdata);
            if (error)
            goto out_sb;

/*
 * filesystems should never set s_maxbytes larger than MAX_LFS_FILESIZE
 * but s_maxbytes was an unsigned long long for many releases. Throw
 * this warning for a little while to try and catch filesystems that
 * violate this rule.
 */
            WARN((sb->s_maxbytes < 0), "%s set sb->s_maxbytes to "
                "negative value (%lld)\n", type->name, sb->s_maxbytes);

            up_write(&sb->s_umount);
            free_secdata(secdata);
            return root;
        out_sb:
            dput(root);
            deactivate_locked_super(sb);
        out_free_secdata:
            free_secdata(secdata);
        out:
                return ERR_PTR(error);
    }
```

该函数调用 type->mount() 执行挂载操作，返回 sockfs 根目录 root。type->mount()指向 sockfs_mount()函数。如下：

```
net/socket.c
static struct dentry *sockfs_mount(struct file_system_type *fs_type,
        int flags, const char *dev_name, void *data)
{
    return mount_pseudo_xattr(fs_type, "socket:", &sockfs_ops,
            sockfs_xattr_handlers,
            &sockfs_dentry_operations, SOCKFS_MAGIC);
}
```

该函数简单封装 mount_pseudo_xattr()函数，该函数主要创建 super_block 同时创建根节点 inode，dentry 等相关关系。完成 sockfs 的初始化

最终 sockfs 形成如下框架：