Linux 文件系统和 GFS2

Linux Virtual File Switch
Global File System 2

龚溪东
湖南麒麟 编著

长沙 • CHANGSHA

目　录

序 v

前言 vii

第一部分　Linux Virtual File Switch 1

第一章　概述 3

第二章　文件系统对象 7

2.1　file_system_type 文件系统类型 8

附录 A　Angular CLI 9

序

本书献给所有热爱 Linux 内核的读者。我们学习它而获得的最大的乐趣和收获来源于满足了我们的好奇心。最早接触到 Linux 操作系统是在 2005 年暑假在大学校园，在一次中科红旗的推广活动中，接着开始使用 RetHat 9。印象最为深刻的是在 RedHat 9 上安装 Oracle 7，整整 27 次才安装成功，可见当时的折腾劲儿。之后所有的系统都安装成了 Linux，在 Linux 上做专业作业，上网，看视频，看课件，写应用。后来在研究生复试期间，面试老师问我会不会 Linux 内核，我才幡然醒悟，内核这一块是我完全不了解的，对我来说它还只是停留在 Linus 的传说中。虽然有所感触，但后来进入实验室之后，学习的分布式系统这一大领域，而 Linux 内核当时是不那么迫切需要去开发的。包括后来毕业之后参加的几分工作，都不需要面向 Linux 内核进行开发。直到现在的公司现在的岗位，现在的公司有一条 Linux 发行版的生产线，也有基于这一发行版的加密存储生态圈，定制和优化 Linux 内核态的存储软件栈必不可免，这才开始系统学习心仪已久的 Linux 内核。

由于之前接触过不少开源软件，也仔细研读过几个著名的软件代码，尤其是刚开始的 Lighttpd[1]到 PostgreSQL[2]和后来的 Redis[3]，简直就是工程代码的典范，可读性上佳，阅读这样的代码对程序员来说可谓不可多得的享受。当时太年轻太天真了，以为这就是开源代码应该有的样子。直到看到 Linux 内核，对开源代码的刻板印象才被推翻，想直接通过 Linux 内核的源代码学习和掌握操作系统的原理，机制和策略，简直就是图样图森破。于是我改变策略，先老老实实从驱动开发开始，去实践 Linux 内核的 API，然后再切入 Linux IO Stack[4]。在 DEBUG 的过程中，通过上网、翻书、读代码、查自带文档的方式解决一个又一个问题，上班编码写文档，下班看书读源码，从刚开始只是简单地实现功能，通过 3 年来的不断学习和实践，到现在有了定制和优化 Linux 内核软件栈的信心。

市面上关于 Linux 文件系统的书籍汗牛充栋，其中不乏经典。《Linux 内核设计与实现》，《Linux 设备驱动程序》，《深入理解 Linux 内核》，《深入 Linux 内核架构》等都是豆瓣[5]上评分超 8.5 的经典。《Linux 内核设计与实现》我通读过两遍，《Linux 设备驱动程序》被我画满了记号，《深入理解 Linux 内核》和《深入 Linux 内核架构》虽没有完整地过一遍，但其中工作相关的章节拜读多次。不得不说，中文翻译还是存在问题的，鉴于原本实在经典，译本确实值得批评：阅读前三本，尤其是第一遍的时候，那酸爽简直呲牙，存在机器翻译

[1] https://www.lighttpd.net/

[2] https://www.postgresql.org/

[3] http://www.redis.cn/, http://www.redis.io/

[4] https://www.thomas-krenn.com/en/wiki/Linux_Storage_Stack_Diagram

[5] https://book.douban.com

的嫌疑；第四本的问题在于用语习惯有点水土不服，甚至把不应该翻译的 C 语言结构体翻译成了中文，存在不少印刷错误；如果外文功底非常好，还是建议阅读原版，。

当然，本书并不打算讲 Linux 内核的方方面面，事无巨细，否则难免会有狗尾续貂之嫌。本书仅仅围绕文件系统这一块。鉴于在中文世界里市面上专门讲解 Linux 下文件系统的书籍一直缺席，我就想是不是写一本以填补空白。写书本身是一件很严肃的事情，不比在工作中按照公司规范写的技术文档，也不同于业余撰写的博客。至少于我而言，有如下方面的好处：

1. 锻炼下自己的脸皮，所谓内向者的优势；
2. 把新近所学的 LaTeX 赶出来练练兵；
3. 给自己前一段时间的工作做一个总结；
4. 温故而知新，解决 GFS2 产生的各种奇葩问题。

由于 Linux 文件系统在内核中至关重要，但它那集市般的开发模式，拿 C 语言强怼 OOP，导致这一子系统代码虽极度精炼但错综复杂，与企业级开发的工程代码相比，其可读性非常之差。而我精力和能力有限，难免存在理解上的偏差。对于书中的不足与疏漏，欢迎读者将问题反馈到hillgreen012@hotmail.com。如果读者能在阅读本书的过程中，快速建立起“比较完整”和“大体正确”的认识，那将是我最大的欣慰。

龚溪东

湖南麒麟，长沙总部

前言

致谢

本书成书首先要感谢湖南麒麟—为我提供了一个舒适自在的工作环境和专职研究 Linux 存储软件栈的机会。当然是要感谢我的老板 YT 博士，是他的信任我，将这个遗留了多年的技术难点交给我攻关。也要感谢我们技术中心主任 SKK 先生，是他的信任和关怀，为我安排了合理的工作量，并为我提供了足够的资料和信息，从而我有时间完成本书。同样也要感谢部门经理 PY 先生和 JL 先生，他们作为遗留系统的主要设计人员和第一用户，给我提供了非常宝贵的建议，并且在我对这一块基本掌握之前，接收了很多关于这方面的工作，使得我的工作具有持续性。感谢我的老领导 LGH 先生，他作为系统的产品经理，给我提供了非常宝贵的不入文档的需求工程细节，使得我能在更高屋建瓴的角度审视设计决策和实现方案。

其次要感谢我的组员 QB，是他，就是他，与我一同前行，分担了一部分我的工作，让我能够释放精力写下本书。

最后，其实我心里最感谢的是我的家人。作为新时代的毛脚女婿，我要感谢我的丈母娘还有我的老婆小蜜蜂，感谢丈母娘大人天天不辞辛苦给我们做有营养又好吃的饭菜，感谢蠢萌蠢萌的老婆大人天天肚子里揣着一个、手里还牵着她那只得意的、只会卖萌的、得了前列腺炎的傻逼狗仔 Lucky 给我带来的跌宕起伏的欢乐，在程序员要么猝死要么自杀要么脱发的今天，她们比我还提心吊胆，视加班为洪水猛兽的同时却仍然能够充分地同情我的职业、理解我忧虑、支持我的工作，真真难能可贵，让我胖得比她还快。还有我自己的亲生父母，为了成书，我牺牲了多个本应陪伴在他们身边的周末，他们也仍然同情我、理解我、支持我，每次回去还是做那么多好吃的，让我胖得更快了。还有我那个活泼可爱的小外甥 YWW，我一直想教他编程呢……这些都让我感觉很幸福。

感谢你们！我会继续拯救全宇宙来换取你们更多的支持。

PART I

第一部分

Linux Virtual File Switch

第一章

概述

磁盘设备,都有以下性质:

- 是一组线性排列的磁盘块;
- 可以访问其中的任意磁盘块;
- 可以独立地读 / 写磁盘块;
- 如果在磁盘块中写入数据,将被记录下来,并在以后的读访问中返回这些数据。

文件系统 (File System) 是存储和组织文件 (即一系列相关的数据),以便可以方便地进行查找和访问一种机制。不同的文件系统有不同的文件存储和组织方式。以基于磁盘的文件系统为例,文件是以磁盘块为单位存储的,文件系统设计的一个重要问题是记录各个文件分别用到哪些磁盘块。基于磁盘的文件系统至少有以下几种文件数据存储方式。

- 连续存储: 即把每个文件作为连续数据块存储在磁盘上。这一方案简单、容易实现,记录文件用到的磁盘块仅需记住第一块的地址,并且性能较好,一次操作就可读出整个文件。但是,它有一个致命的缺陷。在创建文件时,必须知道文件的最大长度,否则无法确定需要为它保留多少磁盘空间。因此,连续存储主要适合于文件数据一次性写入的系统 (例如 romfs)。
- 链接表存储:即将每个文件使用的磁盘块顺序链接起来。虽然用于链接磁盘块的指针可以使用磁盘块本身的空间,但这将使得每个磁盘块实际存储的数据字节数不再是 2 的幂,给上层应用程序带来不便。文件系统从磁盘块中取出一些指针,作为索引存放在文件分配表 FAT。只需要记录文件的起始磁盘块编号,顺着文件分配表中索引指针组织成的链表,就可以找到文件的所有磁盘块。
- inode 存储: 每个文件都有一张称为 inode (索引节点) 的表,通过它得到文件所有磁盘块的编号。小文件的所有磁盘块编号都直接存放在 inode 内。稍大的一些文件, inode 记录了一个称为一次间接块的磁盘块编号,这个磁盘块存放着文件的其他磁盘块编号。如果文件再扩大,可以在 inode 中记录二次间接块编号,二次间接块存放着多个一次间接块的编号,而每个一次间接块又存放着文件的其他磁

盘块编号。如果这也不够的话，还可以使用三次间接块。本章要讨论的 GFS2 文件系统，就使用的这种分配方案。

即便是基于磁盘的文件系统，在实现方法上也有很大的差异。实际上，还有各种 “伪” 文件系统，如 sysfs 等，以及分布式文件系统，如 GFS，GFS2 等。

Linux 设计人员很早就注意到了如何使 Linux 支持各种不同文件系统的问题。此外，为了保证 Linux 的开放性，还必须方便用户开发新的文件系统。为了实现这一目的，就必须从种类繁多的各种具体文件系统中提取出它们的共同部分，设计出一个抽象层，让上层应用程序可以通过统一的界面进行操作，当需要具体文件系统介入时，由抽象层调用具体文件系统的回调函数来处理。

Linux 文件系统被分为两层，如图 1.1 所示。

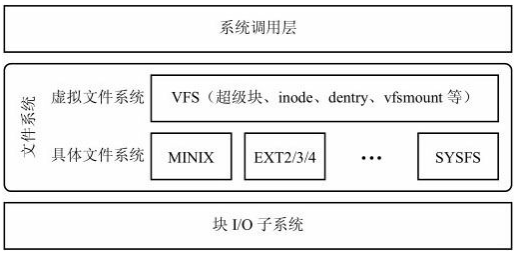

图 1.1: Linux 文件系统分层

上层为虚拟文件系统开关（Virtual Filesystem Switch）层，简称为虚拟文件系统（VFS）。它是具体文件系统和上层应用之间的接口层，将各种不同文件系统的操作和管理纳入一个统一的框架，使得用户不需要关心各种不同文件系统的实现细节。严格说来，VFS 并不是一种实际的文件系统。它只存在于内存中，不存在于任何外存空间。VFS 在系统启动时建立，在系统关闭时消亡。VFS 由超级块、inode、dentry、vfsmount 等信息组成。

下层是具体的文件系统实现，如 Minix、EXT2/3/4、sysfs、GFS2 等。具体文件系统实现代码组织成模块形式，向 Linux VFS 注册回调函数，处理和具体文件系统密切相关的细节操作。

Linux 基于公共文件模型（Common File Model）构造虚拟文件系统。这里所谓的公共文件模型，有两个层次的含义：对于上层应用程序，它意味着统一的系统调用，以及可预期的处理逻辑；而相对于具体文件系统，则是各种具体对象的公共属性以及操作接口的提取。

在公共文件模型中，文件是文件系统最基本的单位，如同磁盘块之相对于磁盘设备。每个文件都有文件名，以方便用户引用其数据。此外，文件还具有一些其他信息，例如，文件创建日期、文件长度等。我们把这些信息称为文件属性（File Attribute）。

文件使用一种层次的方式来管理。层次中的节点被称为目录（Directory），而叶子就是文件。目录包含了一组文件和 / 或其他目录。包含在另一个目录下的目录被称为子目录，而前者被称为父目录，这样，就形成了一个层次的、或者称为树状结构。层次的第一个、或者称为最顶部的目录，称为根（Root）目录。它有点类似树的根——所有的分支都是从这个点开始。根目录或者没有父目录，或者说其父目录为自身。

每个文件系统并不是独立使用的。相反，系统有一个公共根目录和全局文件系统树，要访问一个文件系统中的文件，必须先将这个文件系统放在全局文件系统树的某个目录下。这个过程被称为文件系统装载（Mount），所装载到的目录被称为装载点（Mount Point）。

文件通过路径（Path）来标识。路径指的是从文件系统树中的一个节点开始，到达另一个节点的通路。路径通常表示成中间所经过的节点（目录或文件）的名字，加上分隔符，连接成字符串的形式。如果从根目录开始，则称为绝对路径。如果从某特定目录开始，则称为相对路径。

在目录下，还可以有符号链接。符号链接（Symlink）实际上是独立于它所链接目标存在的一种特殊文件，它包含了另一个文件或目录的任意一个路径名。

在 Linux 公共文件模型下，目录和符号链接也是文件，只不过它们有不同的操作接口，或者有不同的操作实现。上层应用程序通过系统调用对文件或文件系统进行操作。Linux 提供了 open、read、write、mount 等标准的系统调用接口。

Minix 是 Linux 最早的文件系统。Minix 文件系统的磁盘布局由 6 部分组成：引导块、超级块、i 节点位图、逻辑块位图、i 节点和逻辑块，如图 1.2 所示。

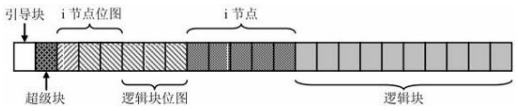

图 1.2: Minix 文件系统磁盘布局

- 在文件系统的开头，通常为一个扇区，其中存放引导程序，用于读入并启动操作系统。
- 超级块用于存放磁盘设备上文件系统结构的信息，并说明各部分的大小。
- i 节点位图用于描述磁盘上每个 i 节点的使用情况。除第 1 个比特位（位 0）以外，i 节点位图中每个比特位依次代表盘上 i 节点区中的一个 i 节点。因此 i 节点位图的比特位 1 代表盘上 i 节点区中第一个 i 节点。当一个 i 节点被使用时，则 i 节点位图中相应比特位被置位。由于当所有磁盘 i 节点都被使用时查找空闲 i 节点的函数会返回 0 值，因此 i 节点位图最低比特位（位 0）闲置不用，并且在创建文件系统时会预先将其设置为 1。在这样的设计下，编号为 0 的 i 节点未被使用，i 节点编号从 1 开始，而编号为 1 保留给根目录对应的 i 节点。
- 逻辑块位图描述磁盘上每个逻辑块的使用情况。除第 1 个比特位（位 0）以外，逻辑块位图中每个比特位依次代表盘上逻辑块区中的一个逻辑块。因此逻辑块位图的比特位 1 代表盘上逻辑块区中第一个逻辑块。当一个逻辑块被使用时，则逻辑块位图中相应比特位被置位。由于当所有磁盘逻辑块都被使用时查找空闲逻辑块的函数会返回 0 值，因此逻辑块位图最低比特位（位 0）闲置不用，并且在创建文件系统时会预先将其设置为 1。在这样的设计下，编号为 0 的逻辑块未被使用，逻辑块编号从 1 开始。
- i 节点反映的是文件的元数据。
- 逻辑块编号则保存了文件的数据。每个文件有且仅有一个 i 节点，但可以有 0、1 或多个逻辑块。i 节点最重要的作用莫过于作为寻址文件数据的出发点，因此 i 节点中需要保存包含文件数据的逻辑块编号。

Linux 源代码树中和文件系统（包括虚拟文件系统和各种具体文件系统）相关的代码主要放在两个目录下，其中头文件所在的目录是`include/linux/`，而 c 文件所在的目录是`fs/`。

第二章

文件系统对象

Linux 文件系统对象之间的关系可以概括为 file_system_type 、super_block 、inode 、dentry 和 vfsmount 之间的关系。文件系统类型规定了某种类型文件系统的行为，它存在的主要目的是为了构造这种类型文件系统的实例，或者被称为超级块实例。

超级块反映了文件系统整体的控制信息，超级块以多种方式存在。对于基于磁盘的文件系统，它以特定格式存在于磁盘的固定区域（取决于文件系统类型），为磁盘上的超级块。在文件系统被装载时，其内容被读入内存，构建内存中的超级块。其中某些信息为各种类型的文件系统所共有，被提炼成 VFS 的超级块结构。如果某些文件系统不具有磁盘上超级块和内存中超级块形式，则它们必须负责从零构造出 VFS 的超级块。

inode 反映了某个文件系统对象的一般元信息，dentry 反映了某个文件系统对象在文件系统树中的位置。同超级块一样，inode 和 dentry 也有磁盘上、内存中以及 VFS 三种形式，其中 VFS inode 和 VFS dentry 是被提炼出来，为各种类型文件系统共有的，而磁盘上、内存中 inode 和 dentry 则为具体文件系统特有，根据实际情况，也可能根本不需要。

Linux 有一棵全局文件系统树，反映了 Linux VFS 对象之间的关系（如图 2.1 所示）。文件系统要被用户空间使用，必须先装载到这棵树上。每一次装载被称为一个装载实例，某些文件系统只在内核中使用，也需要这样一个装载实例。每个文件系统装载实例有四个必备元素：vfsmount、超级块、根 inode 和根 dentry。

需要强调一下文件系统类型、超级块实例以及装载实例之间的关系。一个文件系统类型可能有多个超级块实例，而每个超级块实例又可以有多个装载实例。

设想分区 /dev/sda1 和 /dev/sda2 都被格式化为 Minix 文件系统类型。当 /dev/sda1 和 /dev/sda2 上的文件系统实例先后被装载到系统中时，假设在 /mnt/d10 和 /mnt/d2 下，则会有两个超级块实例，分别对应一个装载实例。透过 /mnt/d10 和 /mnt/d2 所作的改动（例如创建文件）分别反映到 /dev/sda1 和 /dev/sda2 上的文件系统实例中。

然后，/dev/sda1 再次被装载到 /mnt/d11 下，则依然还是两个超级块实例，但是 /dev/sda1 对应的超级块实例将对应两个装载实例，一个对应在 /mnt/d10 上的装载，另一个对应在 /mnt/d11 上的装载。透过 /mnt/d10 和 /mnt/d11 所做的改动（例如创建文件）都会被反映到 /dev/sda1 上的文件系统实例中。

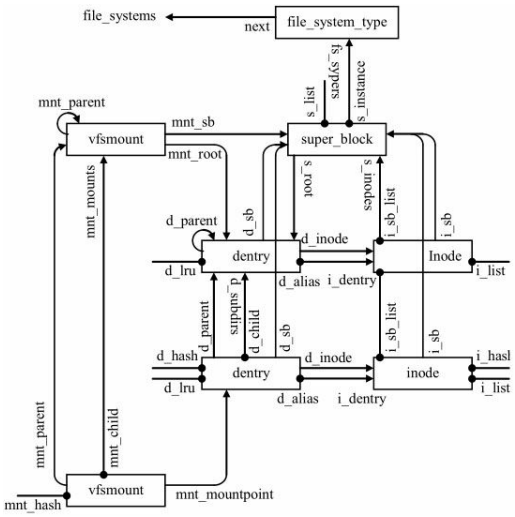

图 2.1: Linux VFS 对象之间的关系

2.1 file_system_type 文件系统类型

Linux 支持多种文件系统，每种文件系统对应一个文件系统类型 `file_system_type` 结构（其结构中的域如表 8-1 所示）。不论是编译到内核，还是作为模块动态装载，文件系统类型需要调用 `register_filesystem` 向 VFS 核心进行注册。如果该文件系统类型不再使用，应该调用 `unregister_filesystem` 从 VFS 核心中注销。上述两个函数都在文件 `fs/filesystems.c` 实现。

</> 代码 2.1: struct file_system_type

```
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;

        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;
};
```

附录 A

Angular CLI

```
ng new appname --style scss --skip-install
```