**文件系统是**大数据、Al 、自动驾驶及物联网数据存储的基石

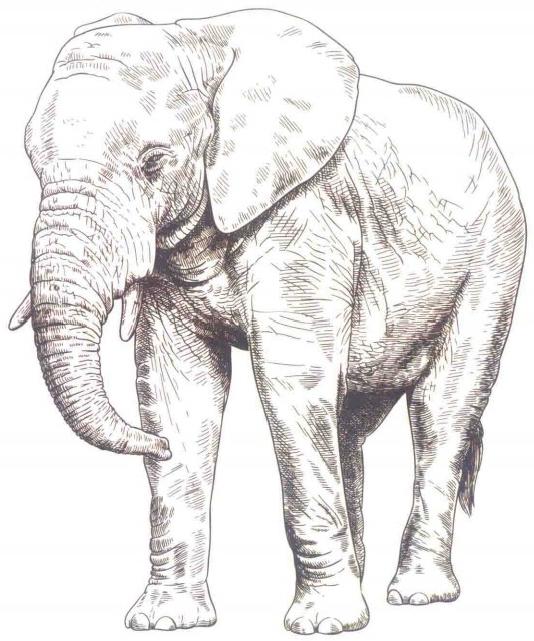
**专业程序员、架构师值得学习和收藏的文件系统宝典**

文件系统 技术内幕

**大数据时代海量数据存储之道**

**张** **书** **宁** **著**

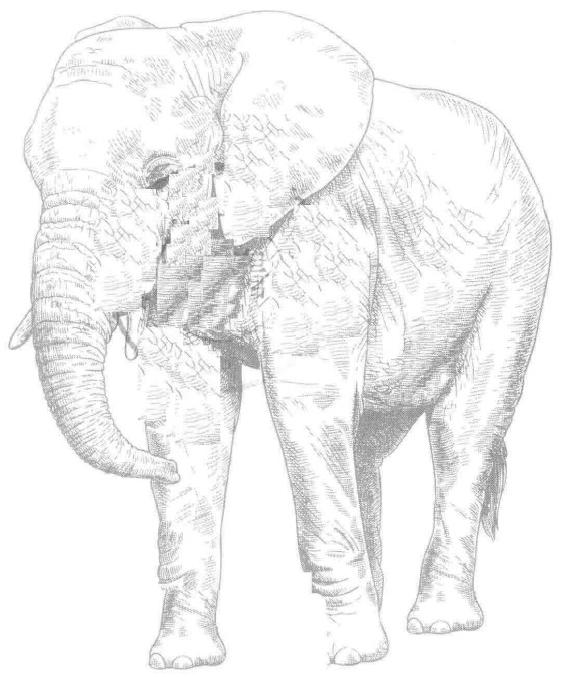
壮



 **中国工信出版集团**

文件系统 技术内幕

**大数据时代海量数据存储之道**

张 书 宁 著

電子工案出版社 ·

Publishing House of Electronics Industry

北京 ·BEIJING

**内** **容** **简** **介**

本书通过理论与实践相结合的方式，深入浅出地介绍了文件系统的概念、原理和具体实现。 本书涵盖本地文件系统、网络文件系统、分布式文件系统和对象存储等内容，可以说涵盖了数据 持久化文件系统的主要领域。为了使读者更加深入地理解文件系统的原理，本书不仅介绍了文件 系统的原理和关键技术，还结合开源项目介绍了文件系统的实现细节。最后，本书介绍了在互联 网领域广泛使用的对象存储、承载海量访问请求的原理及可存储海量数据的架构。希望读者通过 阅读本书对文件系统有全面、深入的认识。

本书既可以作为文件系统及其他存储系统开发人员的指导用书，也可以作为软件架构师、程 序员和Linux 运维人员的参考用书。

未经许可，不得以任何方式复制或抄袭本书之部分或全部内容。 版权所有，侵权必究。

**图书在版编目(CIP) 数据**

文件系统技术内幕：大数据时代海量数据存储之道/张书宁著.一北京：电子工业出版社， 2022.1

ISBN 978-7-121-42478-6

I.① 文 … Ⅱ.①张…Ⅲ .①数据管理IV.①TP274 中国版本图书馆CIP 数据核字(2021)第257070号

责任编辑：林瑞和 特约编辑：田学清 印 刷：三河市龙林印务有限公司

装 订：三河市龙林印务有限公司 出版发行：电子工业出版社

北京市海淀区万寿路173信箱 邮编：100036

开 本：720×1000 1/16 印张：19.25 字数：367千字

版 次：2022年1月第1版

印 次：2022年1月第1次印刷 定 价：102.00元

凡所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行 部联系，联系及邮购电话：(010)88254888,88258888。

质量投诉请发邮件至zlts@phei.com.cn, 盗版侵权举报请发邮件至dbqq@phei.com.cn。 本书咨询联系方式：010-51260888-819,faq@phei.com.cn。

… **推荐序**



文件系统复杂而有趣。文件系统可以说是计算机软件系统中最复杂的子系统。 登上文件系统这座高峰，可以一览众山小，俯视任何其他复杂的软件系统。

文件系统使用的数据结构，能够帮助用户解决各种类型的问题。文件系统的 实现与计算资源管理、内存资源管理、网络资源管理相互作用，了解这些充满历 史故事和智慧的技术方案是一个有趣的学习旅程。

作者任职于存储业界的翘楚企业，长期从事统一存储的研发，是负责文件系 统研发的首席工程师。博观而约取，厚积而薄发。作者在长期知识的积累下撰写 了本书。“知道”是一种本事，把“知道”讲得清楚是另一种本事。本书既包括 丰富的文件系统最前沿知识，其内容讲解又通俗易懂。

在互联网与搜索引擎时代，知识的获取变得容易和便捷。在自媒体时代，信 息的产生、信息的多样性和信息呈现的质量获得呈爆炸性增长。在视频博客时 代，文字、图片、音频、视频的多媒体让知识的展现方式丰富多彩、形象生动。 那么,是否有一本书可以让人们暂时放下其他事情，花时间来静静阅读呢?这 必然是一本极易吸收，学习效率极高的书。阅读时能让人因似曾相识而会心一 笑，时而让人因为新收获而喜悦无比。《文件系统技术内幕：大数据时代海量数 据存储之道》就是带给你这种体验的一本书。本书讲解透彻，语言平实自然，从 文件系统的初始问题出发， 一个问题一个问题地深入， 一个知识点一个知识点 地介绍，这种剥洋葱式层层展开，通过层层台阶登山的方式，读者在闲庭信步之 间不断积累所学的知识，轻松掌握文件系统的知识要点。



文件系统技术内幕

大数据时代海量数据存储之道

为了更好地做到知其然更知其所以然，本书除了适当地介绍了代码，还介绍 了方便好用的实验工具和方法。例如，“第4章\4.1\4.1.1基于文件构建文件系统” 主要介绍使用dd 命 令 和loop 设备方式，就可以不对自己的计算机做任何改动， 模拟出一个文件系统进行实验。“第4章\4.1\4.1.2了解函数调用流程的利器”主 要介绍使用 ftrace 跟踪文件系统的内部API 调用情况，有助于读者理解代码调 用的流程。

这些工具和方法类似《庖丁解牛》中庖丁的牛刀和秘诀，读者可以借此逐步学 习文件系统的知识，了解文件系统的工作原理。相信读者通过学习本书，不仅可以 掌握文件系统的理论知识，还能从工程实践中获取文件系统实现之精华。

DellTechnologies 中端存储部门高级经理高雷

… **前** **言**



从最初的文件系统雏形到现在，文件系统已经发展了六七十年了。文件系统的 特性变得越来越丰富，适用的场景也越来越多。目前，传统文件系统除个别互联网 业务外，基本上能满足现有各种类型业务的需求。同时，很多应用也都直接构建在 文件系统之上。特别是非结构化的数据，通常都是以文件的形式存储在文件系统中 的，如音频、视频和日志等。

随着互联网技术的发展，互联网应用对传统文件系统提出了更高的要求，传统 文件系统很难满足互联网业务的需求。很多互联网公司基于自身业务特性构建了自 己的存储系统。互联网存储系统更多的是基于自己业务特点简化存储系统的某些方 面，而增强另外一些方面的。比如，对文件系统附加特性进行弱化，而对性能和扩 展性进行增强等。虽然互联网公司的存储系统都是一些私有化的存储系统，但核心 技术并没有太大变化。

互联网领域有很多典型的存储系统，其中比较著名的有谷歌的GFS、开源产品 HDFS 、Facebook的 Haystack 及淘宝的TFS 等。每一种存储系统都是针对其应用进 行了特殊的优化，通常只能应用在某种特定的业务模式中。

以Haystack 存储系统为例，其主要应用在Facebook 社交软件的照片应用中。 照片应用有一个非常典型的特征是一次写入、多次读取、不会修改。而该应用对文 件系统的其他特性则没有要求，如扩展属性和快照等。

虽然文件系统具有非常广泛的应用，但是目前国内并没有一本系统介绍文件系 统的书籍。作者在学习文件系统时曾经阅读了很多计算机书籍，发现它们大多只是 对文件系统进行了比较简要的介绍。比如， 一些操作系统类的书籍，其中某些章节 对文件系统的概念和原理进行了介绍，但距离实践还有一些差距，特别是与现在互 联网相关的技术相差甚远。



文件系统技术内幕

大数据时代海量数据存储之道

通过学习本书内容，希望读者能够对文件系统技术有一个全面深入的了解，并 结合源代码进行实例解析。同时，本书对文件系统在互联网和云计算等领域的应用 进行了进一步的介绍和原理分析，让读者对文件系统技术在最前沿的应用有所了解。

**主要内容**

本书分为7章，第1章和第2章主要介绍文件系统的概念、原理和基本使用， 希望读者能够对文件系统有整体、感性的认识。第3章和第4章主要对本地文件系 统的关键技术、原理进行介绍，并且结合实例进行代码分析。本地文件系统是学习 其他文件系统的基础，因此这两章对其进行了详细的介绍。第5章主要对传统网络 文件系统进行介绍，并结合NFS 的代码介绍了实现细节。第6章主要对分布式网 络文件系统进行介绍，并结合目前常用的分布式文件系统CephFS 和 GlusterFS 介 绍了具体实现。第7章主要介绍了文件系统的其他形态，对目前互联网应用最广的 对象存储进行了深入的介绍。

**读者对象**

虽然本书是介绍文件系统知识的专业书籍，但是并非只针对存储系统开发人员。 软件开发人员、运维人员和系统架构师等都可以从本书获得有用的知识。

· 软件开发人员：理解文件系统的原理对软件开发人员如何合理使用文件系统 的相关API 会非常有帮助。比如，软件开发人员不清楚文件系统缓存的存 在，那么在使用API 时可能就不知道如何保证掉电时数据不丢失。

·运维人员：有一些系统参数是与文件系统相关的，如当进程打开时最大文件 的数量。如果能够对文件系统的原理有所了解，相信可以帮助运维人员合理 地设置系统参数。

· 系统架构师：文件系统中的很多技术是通用技术，了解这些技术可以帮助系 统架构师进行其他系统的设计，还可以帮助系统架构师将文件系统中的一些 技术迁移到其他软件设计中。

**前** **言**

**软件及代码版本**

本书涉及的软件比较多，分别是Linux 内 核 、Ceph 、GlusterFS 和 NFS-Ganesha 等。本书涉及的Linux 内核代码为5.8版本，涉及的Ceph 相关代码为13.2(Mimic)

版本，涉及的GlusterFS 相关代码为release-8 版本，涉及的NFS-Ganesha 的代码为 2.8.3版本。

本书介绍了从本地文件系统到分布式文件系统等众多技术，涉及的技术点比较 多。作者在阐述时尽量结合源代码和图示将相关内容解释清楚。由于作者水平有限， 书中难免存在一些疏漏和不足，希望同行专家和广大读者给予批评与指正。

特别要感谢电子工业出版社的林瑞和编辑，没有他的鼓励和指导，就没有本书 的问世。在撰写本书的过程中，林瑞和编辑给予了很多非常专业的建议。还要感谢 我的好友刘占宁，他对整本书稿进行了很认真的阅读，无论是遣词造句，还是技术 内容的准确性方面都提出了很多建议，使得本书的内容更加精准。

我在撰写本书时得到了家人，特别是我的妻子路欢欢的很大支持，她承担了很 多的家务，让我有更多的时间专注写作。另外，还有很多其他朋友和同事对本书提 了建议，在此一并表示感谢!

张书宁

2021年11月于北京

**读** **者** **服** **务**



微信扫码回复：42478

●加入本书读者交流群，与作者互动

●获取【百场业界大咖直播合集】(持续更新),仅需1元

●●●●  **录**

[**第** **1** **章** **从文件系统是什么说起** 1](#bookmark2)

[1.1 什么是文件系统 1](#bookmark3)

[1.1.1 普通用户角度的文件系统 3](#bookmark4)

[1.1.2 操作系统层面的文件系统 9](#bookmark5)

[1.1.3 文件系统的基本原理 12](#bookmark6)

[1.2 常见文件系统及分类 13](#bookmark7)

[1.2.1 本地文件系 统… …………………………………………………14](#bookmark8)

[1.2.2 伪 文 件 系 统… …………………………………………………14](#bookmark9)

[**1**22.3.4 **文件系统** 1615](#bookmark10)

[1.2.5 分布式文件系统 16](#bookmark11)

[**第** **2** **章** **知其然——如何使用文件系统** 17](#bookmark12)

[2.1 巧妇之炊——准备开发环境 17](#bookmark13)

[2.2 文件内容的访问——读/写文件 18](#bookmark14)

[2.2.1 文件系统的API 18](#bookmark15)

[2.2.2 文件访问的一般流程 19](#bookmark16)

[2.2.3 文件内容的读/写实例 20](#bookmark17)

[2.2.4 关于API 函数的进一步解释 22](#bookmark18)

[2.3 如何遍历目录中的文件 24](#bookmark19)

[2.4 格式化文件系统与挂载 27](#bookmark20)

[2.5 文件系统与权限管理 29](#bookmark21)

[2.5.1 Linux 权限管理简介 30](#bookmark22)

[2.5.2 设置文件的RWX 权限 31](#bookmark25)

[2.5.3 设置文件的ACL 权 限 34](#bookmark23)

[2.6 文件系统的锁机制 37](#bookmark24)

目录

[2.6.1 文件锁的分类与模式 37](#bookmark26)

[2.6.2 Linux 文件锁的使用 38](#bookmark27)

[2.7 文件系统的扩展属性 41](#bookmark28)

[2.8 文件的零拷贝 42](#bookmark29)

[2.8.1 零拷贝的基本原理 42](#bookmark30)

[2.8.2 零拷贝的系统API 44](#bookmark31)

[**第** **3** **章** **知其所以然——本地文件系统原理及核心技术** 46](#bookmark32)

[3.1 Linux 文件系统整体架构简介 46](#bookmark33)

[3.1.1 从 V F S到具体文件系统 48](#bookmark34)

[3.1.2 关键处理流程举例 50](#bookmark35)

[3.2 本地文件系统的关键技术与特性 61](#bookmark36)

[3.2.1 磁盘空间布局(Layout) 61](#bookmark37)

[3.2.2 文件的数据管理 69](#bookmark38)

[3.2.3 缓存技术 77](#bookmark39)

[3.2.4 快照与克隆技术 82](#bookmark40)

[3.2.5 日志技术 84](#bookmark41)

[3.2.6 权 限 管 理 85](#bookmark42)

[3.2.7 配额管理 89](#bookmark43)

[3.2.8 文件锁的原理 91](#bookmark44)

[3.2.9 扩展属性与ADS 92](#bookmark45)

[3.2.10 其他技术简介 94](#bookmark46)

[3.3 常见本地文件系统简介 95](#bookmark47)

[3.3.1 ExtX 文件系统 95](#bookmark48)

[3.3.2 XFS 文件系统 95](#bookmark49)

[3.3.3 ZFS 文件系统 96](#bookmark50)

[3.3.4 Btrfs 文件系统 97](#bookmark51)

[3.3.5 FAT 文件系统 98](#bookmark52)

[3.3.6 NTFS 文件系统 98](#bookmark53)

[**第** **4** **章** **从理论到实战——Ext2文件系统代码详解** 9](#bookmark54)

[4.1 本地文件系统的分析方法与工具 9](#bookmark55)

[4.1.1 基于文件构建文件系统 100](#bookmark56)

[4.1.2 了解函数调用流程的利器 100](#bookmark57)

**文件系统技术内幕**

大数据时代海量数据存储之道

[4.2 从Ext2文件系统磁盘布局说起 102](#bookmark58)

[4.2.1 Ext 2文件系统整体布局概述 102](#bookmark59)

[4.2.2 超级块(SuperBlock) 103](#bookmark60)

[4.2.3 块组描述符(Block Group Descriptor) 106](#bookmark61)

[4.2.4 块位图(Block Bitmap) 108](#bookmark62)

[4.2.5 inode 位图(inode Bitmap) 109](#bookmark63)

[4.2.6 inode 与 inode表 109](#bookmark64)

[4.3 Ext 2文件系统的根目录与目录数据布局 113](#bookmark65)

[4.4 Ext 2文件系统的挂载 116](#bookmark66)

[4.5 如何创建一个文件 117](#bookmark67)

[4.5.1 创建普通文件 118](#bookmark68)

[4.5.2 创建软硬链接 121](#bookmark69)

[4.5.3 创 建 目 录 125](#bookmark70)

[4.6 Ext 2文件系统删除文件的流程 125](#bookmark71)

[4.7 Ext 2文件系统中文件的数据管理与写数据流程 129](#bookmark72)

[4.7.1 Ext2文件系统中的文件数据是如何管理的 129](#bookmark73)

[4.7.2 从 V F S到Ext 2文件系统的写流程 130](#bookmark74)

[4.7.3 不同写模式的流程分析 134](#bookmark75)

[4.7.4 缓存数据刷写及流程 138](#bookmark76)

[4.8 读数据的流程分析 140](#bookmark77)

[4.8.1 缓存命中场景 141](#bookmark78)

[4.8.2 非缓存命中场景 142](#bookmark79)

[4.8.3 数据预读逻辑 143](#bookmark80)

[4.9 如何分配磁盘空间 148](#bookmark81)

[4.9.1 计算存储路径 150](#bookmark82)

[4.9.2 获取存储路径 152](#bookmark83)

[4.9.3 分配磁盘空间 153](#bookmark84)

[4.10 Ext2文件系统的扩展属性 154](#bookmark85)

[4.10.1 Ext2文件系统扩展属性是怎么在磁盘存储的 154](#bookmark86)

[4.10.2 设置扩展属性的VFS 流 程 157](#bookmark87)

[4.10.3 Ext 2文件系统扩展属性接口实现 159](#bookmark88)

[4.11 权限管理代码解析 163](#bookmark89)

[4.11.1 ACL 的设置与获取 163](#bookmark90)

·X ·

目录

[4.11.2 ACL 权限检查 164](#bookmark91)

[4.12 文件锁代码解析 165](#bookmark92)

[4.12.1 flock()函数的内核实现 165](#bookmark93)

[4.12.2 fcntl )函数的内核实现 167](#bookmark94)

[**第** **5** **章** **基于网络共享的网络文件系统** 172](#bookmark95)

[5.1 什么是网络文件系统 172](#bookmark96)

[5.2 网络文件系统与本地文件系统的异同 174](#bookmark97)

[5.3 常见的网络文件系统简析 174](#bookmark98)

[5.3.1 NFS 文 件 系 统 174](#bookmark99)

[5.3.2 SMB 协议与CIFS 协议 175](#bookmark100)

[5.4 网络文件系统关键技术 175](#bookmark101)

[5.4.1 远程过程调用(RPC 协 议 ) 176](#bookmark102)

[5.4.2 客户端与服务端的语言——文件系统协议 177](#bookmark103)

[5.4.3 文件锁的网络实现 178](#bookmark104)

[5.5 准备学习环境与工具 179](#bookmark105)

[5.5.1 搭建一个NFS 服务 179](#bookmark106)

[5.5.2 学习网络文件系统的利器 180](#bookmark107)

[5.6 网络文件系统实例 181](#bookmark108)

[5.6.1 NFS 文件系统架构及流程简析 181](#bookmark109)

[5.6.2 RPC 协议简析 185](#bookmark110)

[5.6.3 NFS 协议简析 186](#bookmark111)

[5.6.4 NFS 协议的具体实现 191](#bookmark112)

[5.7 NFS 服务端及实例解析 203](#bookmark113)

[5.7.1 NFSD 203](#bookmark114)

[5.7.2 NFS- Ganesha 210](#bookmark115)

[**第** **6** **章** **提供横向扩展的分布式文件系统** 216](#bookmark116)

[6.1 什么是分布式文件系统 216](#bookmark117)

[6.2 分布式文件系统与网络文件系统的异同 217](#bookmark118)

[6.3 常见分布式文件系统 217](#bookmark119)

[6.3.1 GFS 218](#bookmark120)

[6.3.2 CephFS 219](#bookmark121)

[6.3.3 GlusterFS 219](#bookmark122)

**文件系统技术内幕**

大数据时代海量数据存储之道

[6.4 分布式文件系统的横向扩展架构 20](#bookmark123)

[6.4.1 中心架构 220](#bookmark124)

[6.4.2 对等架构 21](#bookmark125)

[6.5 分布式文件系统的关键技术 222](#bookmark126)

[6.5.1 分布式数据布局 222](#bookmark127)

[6.5.2 分布式数据可靠性(Reliability) 224](#bookmark128)

[6.5.3 分布式数据一致性(Consistency) 228](#bookmark129)

[6.5.4 设备故障与容错(Fault Tolerance) 229](#bookmark130)

[6.6 分布式文件系统实例之CephFS 230](#bookmark131)

[6.6.1 搭建一个CephFS 分布式文件系统 230](#bookmark132)

[6.6.2 CephFS 分布式文件系统架构简析 231](#bookmark133)

[6.6.3 CephFS 客户端架构 234](#bookmark134)

[6.6.4 CephFS 集群端架构 236](#bookmark135)

[6.6.5 CephFS 数据组织简析 239](#bookmark136)

[6.6.6 CephFS 文件创建流程解析 244](#bookmark137)

[6.6.7 CephFS 写数据流程解析 251](#bookmark138)

[6.7 分布式系统实例之GlusterFS 253](#bookmark139)

[6.7.1 GlusterFS 的安装与使用 253](#bookmark140)

[6.7.2 GlusterFS 整体架构简析 257](#bookmark141)

[6.7.3 转换器与转换器树 259](#bookmark142)

[6.7.4 GlusterFS 数据分布与可靠性 263](#bookmark143)

[6.7.5 GlusterFS 客户端架构与I/O 流程 268](#bookmark144)

[6.7.6 GlusterFS 服务端架构与I/O 流程 270](#bookmark145)

[**第** **7** **章** **百花争艳——文件系统的其他形态** 272](#bookmark146)

[7.1 用户态文件系统框架 272](#bookmark147)

[7.1.1 Linux 中的用户态文件系统框架 Fuse 272](#bookmark148)

[7.1.2 Windows 中的用户态文件系统框架 Dokany 279](#bookmark149)

[7.2 对象存储与常见实现简析 282](#bookmark150)

[7.2.1 从文件系统到对象存储 282](#bookmark151)

[7.2.2 S3对象存储简析 287](#bookmark152)

[7.2.3 Haystack 对象存储简析 288](#bookmark153)

[**参考文献** 295](#bookmark154)

· XII ·



第 章

**从文件系统是什么说起**

我们无时无刻不在使用文件系统，进行开发时在使用文件系统，浏览网页时在 使用文件系统，玩手机时也在使用文件系统。

对于非专业人士来说，可能根本不知道文件系统为何物。因为，通常来说，我 们在使用文件系统时一般不会感知到文件系统的存在。即使是程序开发人员，很多 人对文件系统也是一知半解。

虽然文件系统经常不被感知，但是文件系统是非常重要的。在Linux 中，文件 系统是其内核的四大子系统之一；微软的DOS(Disk Operating System,磁盘管理 系统),核心就是一个管理磁盘的文件系统，由此可见文件系统的重要性。

**1.1** **什么是文件系统**

想要更加深入地理解文件系统，先要弄明白什么是文件系统。业界并没有给文 件系统下一个明确的定义，作者翻阅《操作系统概念》和《现代操作系统》等书籍， 也没有找到关于文件系统的明确定义。在《微软英汉双解计算机百科辞典》1]中有 关于文件系统的如下描述。

在操作系统中，文件系统是指文件命名、存储和组织的总体结构。 一个文件 系统包括文件、目录，以及定位和访问这些文件与目录所必需的信息。文件系统 也可以表示操作系统的一部分，它把应用程序对文件操作的要求翻译成低级的、 面向扇区的并能被控制磁盘的驱动程序所理解的任务。

关于文件系统的定义，《微软英汉双解计算机百科辞典》给出的描述比较详细，



**文件系统技术内幕**

大数据时代海量数据存储之道

但过于烦琐。《计算机科学技术名词》(第三版)[2给出的定义如下。

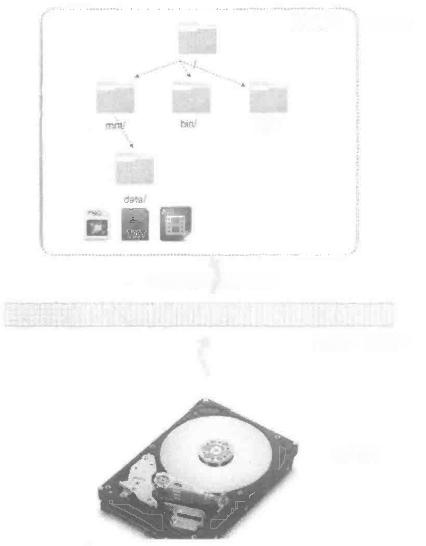
(1)存储、管理、控制、保护计算机系统中持久数据的软件模块。

(2)存储在外存的具有某种组织结构的数据集合。

从前文对文件系统的描述可以知道，文件系统是一个控制数据存取的软件系 统，它实现了文件的增、删、改、查。而通常我们所说的文件系统是构建在硬盘(SSD 卡 和SD 卡等)中的。因此，文件系统其实就是一个对硬盘(或者说块设备)空间 进行管理，实现数据存取的软件系统。

从狭义上来说，文件系统实现了对磁盘数据的存取。而从广义上来说，文件系 统未必需要构建在磁盘中，它还可以构建在网络或内存中。无论构建在哪种设备上， 最为核心的功能是实现对数据的存取。

除对数据的存取外，文件系统更重要的一个功能是抽象了一个更加容易访问存 储空间的接口。这里所说的接口包括用于程序开发的API 接口和普通用户的操作 接口。为了便于理解，我们可以将文件系统对磁盘空间的管理用图1-1表示。



操作接口 文件系统

root

将线性地址拍象力层级结构

抽象层一线性空间

硬件设备

图1-1 文件系统空间管理原理示意图

我们对图1-1进行简单的解释。底层是硬件设备，这里以硬盘为例。中间层是 硬盘驱动器和操作系统把硬盘抽象为的一个连续的线性空间。顶层是文件系统，将

第 1 章 从 文 件 系 统 是 什 么 说 起

线性空间进行管理和抽象，呈现给用户一个层级结构。这里的层级结构就是我们平 常看到的目录、子目录和文件等元素的集合，即目录树。

**1.1.1** **普通用户角度的文件系统**

大家对文件系统的了解可能还是比较抽象，我们看一个Windows 中文件系统 的实例。在 Windows 中，通常大家不太清楚文件系统为何物。因为， 一般安装 Windows 时都是一键安装，安装完成后磁盘已经被格式化(格式化是对硬盘、优盘 或其他块设备进行初始化的过程)。通过Windows 的资源管理器，我们只需要移动 鼠标就能实现文件的所有操作。所以，我们通常并不会感知到在Windows 中还有 文件系统的存在。

如图1-2所示，这些文件夹与文件都是存储在磁盘上的，但我们并不知道具体 磁盘空间是什么样的，文件系统软件呈现出来一个非常清晰的表象，我们可以非常 容易地创建、删除和复制文件夹与文件。而这些功能是通过一个软件实现的，这个 软件就是文件系统。



蜜 m+TR

Wis-SD

F

9



下



P

什 e



写

裕域



此电

10



所

圆文雎 下醒



 m

Meous192168215010

Do1921682152



bt

ybar

09

1 小 1 0

tog

NET



g



Tae

kbag

fos ap

pwene



witerr



l

Fepor

TAN



w P



pI



四A



IA

el

069



Windous s0C DaD



me



serupocf

cwuipit

Wedow



#

9

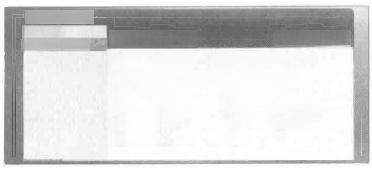
图 1 - 2 Windows 资 源 管 理 器

不仅在Windows 中可以通过鼠标实现对文件系统的管理，而且在Linux 中也可 以通过鼠标实现对文件系统的管理。图1-3所示为Ubuntu (一个以桌面应用为主的 Linux 操作系统)中的GUI 管理界面，可以看出其管理方式与Windows 非常类似。



文件系统技术内幕

大数据时代海量数据存储之道



*o*me

m

boot lostfound h

cdrom medi sbn u

de mot elinuk nird limg

ee opt  wlinr

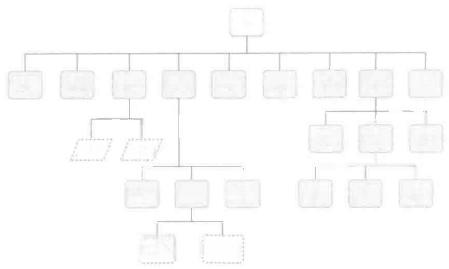
home proe 

hne D →a ser

bin  ot p

图1-3 Ubuntu中的GUI管理界面

目前，主流操作系统的文件系统数据组织形式大致都是这样的。在目录下面有 子目录和文件，子目录下面又有子目录和文件，形成一个层级的树形结构。这种方 式非常方便用户实现对文件的分类管理。以Linux 为例，最终形成的层级目录树形 结构如图1-4所示。





dey home ib sbn met

lb sda

sunry zhang  bin  man

heloc worae

bin boot

sbn

local

uS7

sda

var

图1-4 层级目录树形结构

这种树形结构的数据组织方式是非常实用的。通过目录可以实现对内容的分类 管理，而主类又可以包含子类。这在现实中有很多类似的场景，如公司销售按照区 域管理客户资料。如图1-5所示，在每个目录中存储不同省份的客户资料；而每个 省份又划出为不同的地区。通过这种层级结构，非常便于用户管理资料。

itworld123@sunnyzhang:~/test/flesystems tree

上海 北京 山东

德州

济南 山 西

河北

唐 山

石 家 庄 邢 台

10 directories,0 files

图1-5 公司客户资料层级结构

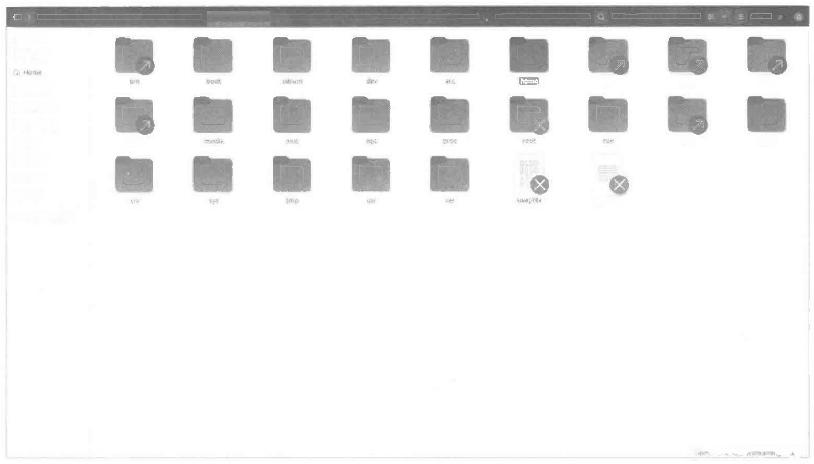
第 1 章 从 文 件 系 统 是 什 么 说 起

同时还需要说明的是，文件系统对文件进行了抽象化处理，对文件系统而言， 所有文件都是字节流，它并不关注文件的格式与内容。文件的格式是由具体的应用 软件来负责的。例如，文本文件由文本编辑软件来处理(如vim); 图片文件则由图 片浏览工具或编辑工具来处理(如Windows 中的画图工具)。只有具体的软件才会 关注文件的格式和内容。

下面对文件系统常见的概念进行一些简要的介绍。

[**1.1.1.1**](1.1.1.1) **目** **录** **(Directory) 的概念**

前文已经提到过"目录"这个术语，但并没有解释。在文件系统中目录是一种 容器，它可以容纳子目录和普通文件。目录就像日常生活中的文件夹一样，它可以 容纳文件。在GUI 终端中可以很容易地分辨出目录和普通文件的差别，目录的图 标与日常生活中的文件夹也非常像。如图1-6所示，选中的home 就 是Linux 下目 录的图标。





nt

stare

ab2

Drktap boumen pou

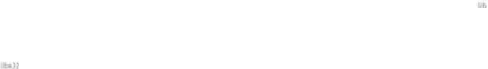
shn

Mdun 曰 Veos

re

ourliscom

hrwwus ketengreG



vslog

wp

64

图1-6 GUI 终端中的目录与普通文件

在命令行中区分目录和普通文件就不太直观了，但也并不太困难。有些 Shell 会将目录和文件显示成不同的颜色，以方便用户区分目录和文件。另外，在属性中 会有标识，如图1-7所示，“唐山”这一目录最前面的字符为d, 这 个d 就 是directory 的缩写，因此我们可以通过第1个字符来区分目录和文件。



文件系统技术内幕

大数据时代海量数据存储之道

itworld123esunnyzhang:/root/test/filesystem/ 河北s ls -alh

total 24K

drwxr-xr-x5 root root 4.0K Sep 1201:32

drwxr-xr-x8 root root 4.0K Sep 1209:05.



dr wxr- Xr - ×2 root root 4 . 0K Sep 1201:23唐山

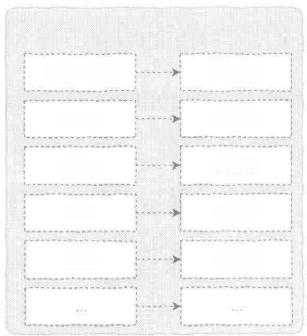
W-rr1 root root 4 Sep 1201:32 投 标 .doc

drwxr-xr-x2 root root 4.0K Sep 1201:23石家庄

drwxr-xr-x2 root root 4.0K Sep 1201:23那 台

图1-7 Shell 中的目录与普通文件

如果深入了解了目录的实现原理，就会知道其实目录本身也是一种文件。只不 过目录中存储的数据是特殊的数据，这些数据就是关于文件名称等元数据(管理数 据的数据)的信息。以“河北”目录为例，其中存储的数据其实是文件名与一个数 字的对应关系，如图1-8所示，这个数字就是所谓的inodeID 。 在文件系统层面中， 普通用户通过文件名读取数据的过程需要这种映射关系。



河北

zhaiyao.txt 543919

zyh.txt 543919

zyl.txt 543881

唐山 543913

邢 台 549016

**图1-8** **目录中的数据格式示意图**

[**1.1.1.2**](1.1.1.2) **文件** **(File) 的概念**

在文件系统中，最基本的概念是文件，文件是存储数据的实体。从用户的角度 来看，文件是文件系统中最小粒度的单元。文件的大小不是固定的，最小可以是0 字节，最大可以是几十太字节(根据具体文件系统而定)。

为了便于用户对文件进行识别和管理，文件系统为每个文件都分配了一个名 称，称为文件名。文件名就好像人名一样，它是一个标识。比如，我们去学校找张 三，让班主任帮忙把张三叫出来，此时班主任就能通过人名很容易找到张三。

文件系统也是这样，当普通用户想要访问某个文件时，告诉文件系统自己想要 访问的文件名，此时文件系统就可以根据文件名找到该文件的数据。比如，在

第1章从文件系统是什么说起

Windows中双击某个视频或图片文件，那么就有相应的软件将其打开。底层原理方

面就涉及文件系统对文件数据查找的流程。

文件名通常包含两部分，并通过“. ”进行了分 隔，但并非绝对。以图1-9中的test1.jpg文件为例， 该文件名可以分为两部分：第一部分称为文件主名， 它表示该文件的标识，就好像人名一样；第二部分 称为扩展名，它的作用是标识文件的类型。这种命



test.ipg

文件主名 扩展名

图1-9文件名的格式

名方式便于用户能够对文件有一个快速的整体认识。比如，我们可以一眼就能知道 某个文件存储的是视频、音频还是图片。

下面进一步介绍文件的内部。从普通用户(开发人员)的角度来看，文件就是 一个线性空间，这就好比程序开发中的数组一样。与数组不同的是文件的大小是可 以变化的，当写入更多的数据时，文件的容量就会变大。虽然文件数据以普通用户 角度来看是线性的、连续的，但是在文件系统层面并非如此。其真实位置可能在磁 盘的任意位置。如图1-10所示， 一个文件通常在逻辑上被划分为若干等份，每一 份被称为一个逻辑块(Block) 。 文件的逻辑块在磁盘中的物理位置并不固定，逻辑 块是连续的，物理块却可能散布在很多地方。

文件



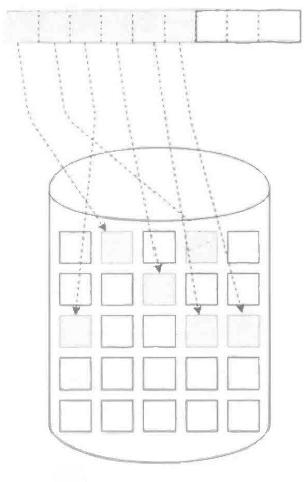
磁盘

图1-10 文件数据的组织形式



**文件系统技术内幕**

大数据时代海量数据存储之道

对文件系统而言，它并不关心文件是什么格式的，而是把所有文件看作字节流。 但是在普通用户层面需要关心文件的格式。因为，不同格式的文件需要使用不同的 工具访问。对Windows 和桌面版的Linux 而言，操作系统层面建立了文件格式与 软件的关联，因此当双击文件的图标时就会自动使用对应的软件打开该文件。但也 不一定，因为有时可能系统缺少相关的软件，或者关联关系被破坏，此时就无法打 开 该 文 件 。

文件格式的种类非常多，如常见的.txt、.pptx、.docx或 .mp3 等，都由特定的工 具软件打开。 .mp3 格式的文件只有通过播放器软件打开才有意义，才可以播放音 乐。如果使用文本编辑工具打开一个.mp3 格式的文件，看到的只能是一堆乱码。

[**1.1.1.3**](1.1.1.3) **链接** **(Link) 的概念**

链接是Linux 文件系统的概念，在Windows 和 macOS 中通常被称为快捷方式。 Linux中的链接分为软链接 (Soft Link) 和硬链接(Hard Link) 两种。其中，软链 接又被称为符号链接(Symbolic Link),它是文件的另外一种形态，其内容指向另 外一个文件路径(相对路径或绝对路径)。硬链接则不同，它是一个已经存在文件 的附加名称，也就是同一个文件的第2个或第N 个名称。

为了更加直观地理解软链接和硬链接的概念，在 test 目录中创建一个源文件 src\_file.txt。同时为该文件分别创建 一 个软链接 (softlink.txt)和 一 个硬链接 (hardlink.txt) 。然后使用1s 命令查看该目录的详细信息，如图1-11所示。

itworld123@sunnyzhang:/root/test/filesystem/links Ls -alhi total 16K

147848 dnwxr-xr-x2 root root 4.0K Sep 1209:06

99 dnwxr-xn-X.8 root root 4.0K Sep 1209:05

147849 NW-r -2 root root 7 Sep 1209:05 hardlink,txt

147850 LrwxrwxrwX I root root 12 Sep 1209:05 softlink.txt-src\_file.txt 147849 -nw-r-r-2 root root 7 Sep 1209:05sre\_file.txt

图1-11 文件的软链接与硬链接

通过上面的结果可以看出，软链接有一个“->”符号，该符号指示了该链接所 指向的目的文件。而硬链接并没有“>”符号，也就是我们无法明确地知道哪个是 硬链接的目的文件。但是如果我们观察一下硬链接与源文件最前面的数字就会发现 是一样的。这个数字是inode ID,说明它们是指向同一个文件的。

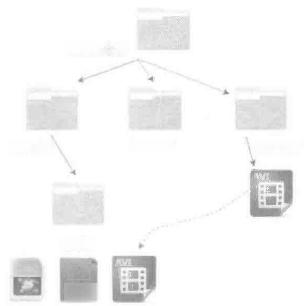
从原理上来理解，硬链接其实是在目录中增加了一项，而该项的 inode ID是 源 文 件 的inode ID。因 此 硬 链 接 与 源 文 件 的 内 容 是 完 全 一 样 的 。

那么链接的作用是什么呢?主要是为了实现对源文件的快速访问，并且节省存 储空间。在有些情况下，我们需要在B 目录使用A 目录中的某个文件。这时使用

第1章从文件系统是什么说起

链接要比复制功能更加方便、合适。因为通过链接的方式，在源文件发生变化的情 况下可以马上感知，不需要重新复制，同时又节省存储空间。

为了更加直观地理解链接的作用，通过图1-12的实例进行简要介绍。以培训 机构教学为例。假设已经有一个教学素材库，有很多素材在目录data 中。某学期需 要开始一个新的课程，该课程要用到素材库中的一个视频文件。该课程的素材都在 course 目录中。此时就可以在course 目录中建立一个到素材库的链接。这样course 目录中既包含了该视频，又不会占用太多存储空间。另外，即使对素材库中的视频 文件进行了修改，course目录中也只是一个链接，因此其内容也会跟着修改，不会 出现不一致的情况。



/mnt/data

media/ books/

data/

PNc

TXT

course/

图1-12 链接的使用示意图

**1.1.2** **操作系统层面的文件系统**

上文从普通用户的角度介绍了文件系统。其目的是为用户提供一个方便管理文 件(数据)的方式。而从操作系统角度来说，文件系统则主要实现对硬件资源的管 理，也就是对磁盘资源的管理。

任何技术的出现都是为了解决问题，文件系统也是为了解决某些问题。那么文 件系统是为了解决什么问题呢?

文件系统解决的是对磁盘空间使用的问题。通常一台计算机配置一个磁盘，而 磁盘的空间就是一个线性空间，就好比一个非常大的数组。然而在一个操作系统上 会运行很多软件，如视频软件、浏览器、音频软件和文本编辑软件等。这些软件通 常都要使用磁盘空间。如果这些软件都直接使用磁盘空间则会有如下很多问题。

(1)磁盘空间的访问会存在冲突。由于没有软件统一管理磁盘空间，各个软件 各自为政，那么在访问磁盘空间时就有可能存在冲突的情况。



文件系统技术内幕

大数据时代海量数据存储之道

(2)磁盘空间的管理会非常复杂。由于各种不同格式的文件，以及不同大小的 文件，没有文件系统将导致磁盘空间很难管理。

在计算机领域中有一个非常有用的定律，任何复杂的问题都可以通过分层来解 决。文件系统就是这样一种思路。操作系统实现了文件系统，而文件系统是应用程 序与磁盘驱动程序之间的一层软件。

文件系统对下实现了对磁盘空间的管理，对上为用户(应用程序)呈现层级数 据组织形式和统一的访问接口。

基于文件系统，用户(应用程序)只需要创建、删除或读取文件即可，他们并 不需要关注磁盘空间的细节，所有磁盘空间管理相关的动作则由文件系统来处理。 文件系统所处的位置如图1-13所示。

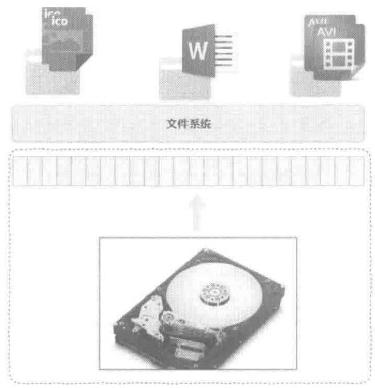


图1-13 文件系统所处的位置

其实文件系统不仅可以构建在磁盘上，它还可以构建在任何块设备上，甚至网 络上。在Linux 中，最常见的块设备包括裸磁盘、分区、LVM 卷 和RAID 等。我们 可以对上述任何块设备进行格式化，构建文件系统。Windows 中的文件系统也是可 以构建在其卷组上的。

文件系统不仅仅可以构建在块设备上，甚至可以构建在一个普通的文件上。磁 盘是一个线性空间，而文件也是一个线性空间。因此，在一个文件上构建文件系统 是没有任何问题的。这也是我们在后面学习文件系统用到的一个便捷方法。

下面先简单看一下如何在一个文件上构建文件系统。首先要有一个内容全为0 的文件。生成方法如下：

dd if=/dev/zero of=./img.bin bs=1M count=1

第 1 章 从 文 件 系 统 是 什 么 说 起

执行命令后，查看一下当前目录，可以看到生成了一个容量为1.0MB 的新文

**件，如图1-14所示。**

rootesunnyzhang:-/test/ext2=dd if=/dev/zero,of=./ing.bin bs=1H count=l 1+0 records in

1+0 records out

1048576 bytes (1.8NB,1.0RIB)copied.0.00275223 s.381MB/s rootesunnyzhang:-/test/ext2m 11

total 1032

dnxr-xr-x 2 root root 4096 Jan 3111;27/

dnxr-xr-x5 root root 4096 Jan 3111:26/

T1 raot rat 1048576 lan 11027 inobin

图1-14 生成的新文件信息

然后对该文件进行格式化。例如，构建一个Ext2 文件系统，并对该文件系统

**进行格式化，具体方法及结果如图1-15所示。**

rootesunnyzhang:-/test/ext2#mkfs.ext2 ing.bin mke2fs 1.44.1(24-Har-2018)

Discarding device blocks:done

Creating filesysten with 1024 Ik blocks and 1281nodes

ALlocating group tables:done Writing inode tab les:done

Writing superblocks and filesysten accounting information:done 图1-15 Ext2文件系统的格式化

从执行命令的结果可以看出，Ext2 文件系统已经完成格式化。如何验证一下 呢?一个简单的方法是使用dumpe2fs 命令，该命令可以获取文件系统的描述信息。

另外一个复杂的方法是借助Linux 的循环设备(回环设备)。通过该循环设备可以 将一个文件虚拟成块设备，然后将该块设备挂载到目录树中。具体需要执行的命令 如 下 ：



losetup /dev/loop10./img.bin mkdir/tmp/ext2

mount /dev/loop10/tmp/ext2

执行完成上述命令后，如果没有出现错误，且可以看到如图1-16所示的目录 内容，则说明Ext2 文件系统格式化成功。当然，为了进一步的验证，可以向该目 录拷贝文件。

rootesunnyzhang:-/test/extzIL/top/ext2/ total 17

dnexr-xrx 3 root root 1024 Jan 3111:32/ drwxnwint 10 root root 4096 Jan 3111:33|

dnx 2 root roof 12288 an 3111:92 lost+found/

图1-16 Ext2文件系统挂载后的目录

综上所述，文件系统实现了对线性存储空间的管理，这里的线性存储空间既可 以是磁盘等块设备，还可以是一个文件。



**文件系统技术内幕**

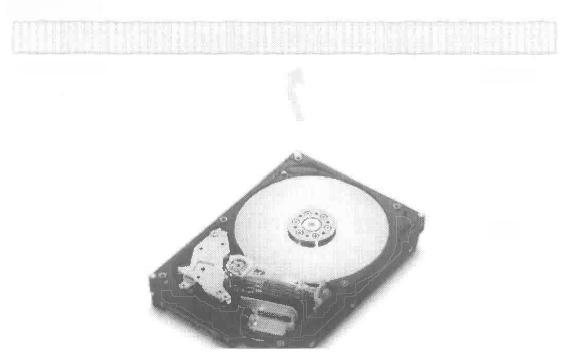
大数据时代海量数据存储之道

**1.1.3** **文件系统的基本原理**

前文从用户角度和操作系统角度分别对文件系统进行了介绍，可以知道文件系 统实现了对磁盘空间的管理，并提供了便于使用的接口。本节介绍一下文件系统的 基本原理。

要想理解文件系统，先要从磁盘说起，毕竟文件系统是构建在磁盘中的。虽然 磁盘的内部非常复杂，但是磁盘厂商做了很多工作，将磁盘的复杂性掩盖起来。对 于普通用户来说，磁盘就是一个线性空间，就好像C 语言中的数组一样，通过偏移 就可以访问其空间(读/写数据)。

如图1- 17所示， 一个包含多个盘片的磁盘，经过磁盘控制器和驱动程序之后， 普通用户看到的是一个线性的存储空间。其地址空间从0开始， 一直到磁盘的最大 容量。



0246810

1.357911

线性空间

磁盘

图1-17 磁盘与空间的线性化

虽然这种线性空间已经极大地简化了对磁盘的访问，但是对普通用户而言还是 非常难以使用的。因此，对操作系统而言，需要为用户提供一个更加直观和易用的 使用接口。

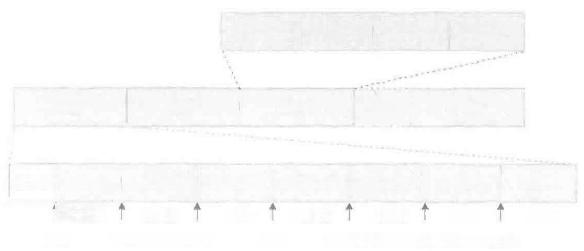
文件系统正是操作系统中用于解决磁盘空间管理问题的软件， 一方面文件系统 对磁盘空间进行统一规划，另一方面文件系统提供给普通用户人性化的接口。就好 比仓库中的货架，将空间进行规划和编排，这样根据编号就可以方便地找到具体的 货物。而文件系统也有类似功能，将磁盘空间进行规划和编号处理。这样普通用户 通过文件名就可以找到具体的数据，而不用关心数据到底是怎么存储的。

以Ext4文件系统为例，它将磁盘空间进行划分。首先将磁盘空间划分为若干

第 1 章 从 文 件 系 统 是 什 么 说 起

个子空间(见图1-18),这些子空间称为块组。然后将每个子空间划分为等份的逻

辑块。这里逻辑块是最小的管理单元，逻辑块的大小可以是1KB、2KB或4KB等， 由用户在格式化时确定。

数据块位图 inode位图 inode表 数据块

块组0 块组1 块组2  块组π

引导块 超级块 块组描述符 预留GDT 块 数据块位图 inode 位图 inode表 数据块

↑1biock 1 block

1KB 4KB 8KB

many 1 block Tblock

WKB n+4KE i+8)KB

many many

mKB

图1-18 Ext4文件系统的磁盘布局(Layout)

为了管理这些逻辑块，需要一些区域来记录哪些逻辑块已经被使用了， 哪些还 没有被使用。 记录这些数据的数据通常在磁盘的特殊区域，我们称这些数据为文件 系统的元数据(Metadata), 如图1- 18所示中的数据块位图和inode 位图等。通过 元数据，文件系统实现了对磁盘空间的管理，最终为用户提供了简单易用的接口。

这样，用户对文件的操作就转化为文件系统对磁盘空间的操作。比如，当用户 向某个文件写入数据时，文件系统会将该请求转换为对磁盘的操作，包括分配磁盘 空间、写入数据等。而对文件的读操作则转换为定位到磁盘的某个位置、从磁盘读 取数据等。

至此，相信大家对文件系统的基本原理有了一个感性的认识，但是有可能还有 一种云里雾里的感觉。不用太着急，作者在后续章节会进行更加详细的介绍。

**1.2** **常见文件系统及分类**

目前，常见的文件系统有几十个。虽然文件系统的具体实现形式纷繁复杂，具 体特性也各不相同，但是有一定规律可循。下面将介绍一下常见的文件系统都有哪 些种类。

通过前文我们了解了基于磁盘的本地文件系统，对其基本原理也进行了简要的 介绍。其实文件系统发展到现在，其种类也丰富多样。比如，基于磁盘的普通本地 文件系统除了Ext4, 还包括XF S 、ZFS和Btrfs等。其中Btrfs和ZFS 不仅可以管 理一块磁盘，还可以实现多块磁盘的管理。不仅如此，这两个文件系统实现了数据



**文件系统技术内幕**

大数据时代海量数据存储之道

的冗余管理，这样可以避免磁盘故障导致的数据丢失。

除了对磁盘数据管理的文件系统，还有一些网络文件系统。也就是说，这些文

件系统看似在本地，但其实数据是在远程的专门设备上。 客户端通过一些网络协议 实现数据的访问，如NFS和GlusterFS等文件系统。

经过几十年的发展，文件系统的种类非常多，我们没有办法逐一进行介绍。本 节就对主要的文件系统进行介绍。

**1.2.1** **本地文件系统**

本地文件系统是对磁盘空间进行管理的文件系统，也是最常见的文件系统形 态。从呈现形态上来看，本地文件系统就是一个树形的目录结构。本地文件系统本 质上就是实现对磁盘空间的管理，实现磁盘线性空间与目录层级结构的转换，如

图1-19所示。

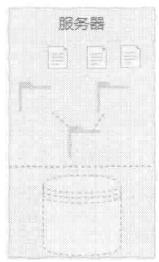


图1-19 从磁盘线性空间到目录层级结构

从普通用户的角度来说，本地文件系统主要方便了对磁盘空间的使用，降低了 使用难度，提高了利用效率。常见的本地文件系统有Ext4 、Btrfs 、XFS 和 ZFS 等。

**1.2.2** **伪文件系统**

伪文件系统是Linux中的概念，它是对传统文件系统的延伸。伪文件系统并不 会持久化数据，而是内存中的文件系统。它是以文件系统的形态实现用户与内核数

据交互的接口。常见的伪文件系统有proc 、sysfs和configfs等。

在Linux 中，伪文件系统主要实现内核与用户态的交互。 比如，我们经常使用 的 iostat工具，其本质上是通过访问/proc/diskstats文件获取信息的，如图1-20所 示。而该文件正是伪文件系统中的一个文件，但其内容其实是内核中对磁盘访问的 统计，它是内核某些数据结构的实例。

**第** **1** **章** **从** **文** **件** **系** **统** **是** **什** **么** **说** **起**

rootg5unnyzhang:-/test/ext2a cat /proc/diskstats

o loope 55.0214053610000日4336

.loopi 110340241221319080 日002776128364

2 loop25016/00000000

7 3.loop300000000.000

4 7 5

6

7

7

7

loop500000000.00夏

Loop400000.080000

loop<6.0000.008.0000>

loop70000000D00

11 0.sr00.0|000000000

8 0.5da 5655129712720381571125836758100348189670716043688127888

8 1sdal 228e 12018 B8000008888

2 sda255777297126891565669228706581003481896.647401037408121492 8 16 sdb 17708396.1000000044100

7 10 loop10.340158.020.4-12.00.0

**图1-20** **磁盘访问统计信息**

**1.2.3** **网络文件系统**

网络文件系统是基于TCP/IP协议(整个协议可能会跨层)的文件系统， 允许 一台计算机访问另一台计算机的文件系统，就如访问本地文件系统一样[2。网络文 件系统通常分为客户端和服务端，其中客户端类似本地文件系统，而服务端则是对 数据进行管理的系统。网络文件系统的使用与本地文件系统的使用没有任何差别， 只需要执行 mount 命令挂载即可。网络文件系统也有很多种类，如NFS 和 SMB 等 。

在用户层面，完成挂载后的网络文件系统与本地文件系统完全一样，看不出任 何差异，对用户是透明的。网络文件系统就好像将远程的文件系统映射到了本地。 如图1-21所示，左侧是客户端，右侧是文件系统服务端。

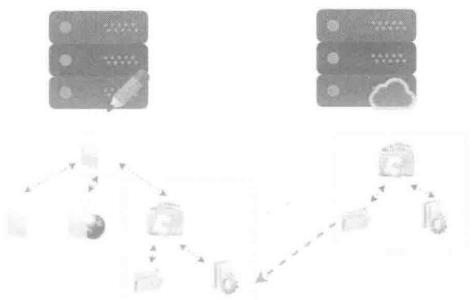


图1-21 网络文件系统的映射

通过图1-21 可以看到，当在客户端对服务端导出的文件系统进行挂载后，服 务端的目录树就成为客户端目录树的一颗子树。这个子目录对普通用户来说是透明 的，不会感知到这是一个远程目录，但实际上读/写请求需要通过网络转发到服务端 进行处理。



**文件系统技术内幕**

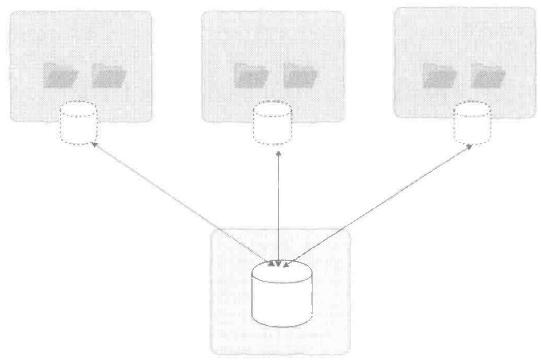
大数据时代海量数据存储之道

**1.2.4** **集群文件系统**

集群文件系统本质上也是一种本地文件系统，只不过它通常构建在基于网络的 SAN 设备上， 且在多个节点中共享SAN磁盘。集群文件系统最大的特点是可以实 现客户端节点对磁盘介质的共同访问，且视图具有一致性，如图1-22所示。

这种视图的一致性是指，如果在节点0创建一个文件，那么在节点1和节点2 都可以马上看到。这个特性其实跟网络文件系统类似，网络文件系统也是可以在某 个客户端看到其他客户端对文件系统的修改的。

但是两者是有差异的，集群文件系统本质上还是构建在客户端的，而网络文件 系统则是构建在服务端的。



节点0 节点1 节点2

存储系统

图1-22 集群文件系统访问示意图

同时，对于集群文件系统来说，其最大的特点是多个节点可以同时为应用层提 供文件系统服务，特别适合用于业务多活的场景， 通过集群文件系统提供高可用集

群机制，避免因为宕机造成服务失效。

**1.2.5** **分布式文件系统**

·从本质上来说，分布式文件系统其实也是一种网络文件系统。在《计算机科学 技术名词》中给出的定义为“一种文件系统，所管理的数据资源存储在分布式网络 节点上，提供统一的文件访问接口"[2],可以看出，分布式文件系统与网络文件系 统的差异在于服务端包含多个节点，也就是服务端是可以横向扩展的。从使用角度 来说，分布式文件系统的使用与网络文件系统的使用没有太大的差异，也是通过执 行mount 命令挂载，客户端的数据通过网络传输到服务端进行处理。



第

章

知其然——如何使用文件系统

本章重点介绍一下如何使用文件系统，如果大家对文件系统的使用比较熟悉， 则可以直接跳过本章。文件系统的使用分为两个不同的角度：一个是普通用户角度； 另一个是程序员角度或开发者角度。需要注意的是，这里的开发是指应用级别的开 发，而非内核文件系统的开发。

从普通用户角度来说，文件系统的使用是非常简单的。对于文件系统的使用无 非四个字，即增、删、改、查。也就是创建文件(夹)、删除文件(夹)、修改或移 动文件(夹)和检索文件(夹)。

从开发者角度来说，也主要集中在上面所述4项内容。另外，可能包含其他一 些高级特性的使用，但差别不大。开发者除了基本使用，还需对文件系统有更深入 的理解，如写数据是如何绕过缓存的，如何创建一个稀疏文件，如何给文件加锁等。

由于从普通用户角度来说使用文件系统是非常简单的，特别是目前文件系统的 管理都是通过GUI ( 如Windows 资源管理器)来完成的，这就更加降低了文件系 统使用的门槛。因此，本节主要从开发者的角度介绍文件系统的使用。

2.1 巧妇之炊——准备开发环境

正所谓“巧妇难为无米之炊”,在开始工作之前需要先准备一下环境。主要指 开发环境，该开发环境用于编译代码，实现对文件系统相关API 的验证。这里以 Linux为主，建议使用Ubuntu 18.04版本。当然，其他Linux开发环境问题也不大， 毕竟Linux 的文件系统API 是遵循POSIX 标准的。

以 Ubuntu 18.04为例，需要安装一些用于开发的软件包。具体安装过程非常简



**文件系统技术内幕**

大数据时代海量数据存储之道

单，可以通过如下命令安装软件包：

sudo apt-get install build-essential manpages manpages-dev manpages-posix manpages-posix-dev

上述软件包主要是开发(编译)工具和帮助文档。 Linux下的开发与Windows 下的开发有着比较明显的差异，在Linux下开发通常不使用IDE环境。Linux下的 开发基本上是先通过文本编辑器编辑代码，再通过编译工具生成可执行文件。

**2.2** **文件内容的访问——读/写文件**

对于普通用户来说，通过命令或单击鼠标就可以进行文件的操作。Linux 的 桌 面 版 、Windows 和 macOS ( 图 2 - 1 为macOS 的 GUI) 等都提供基于GUI 的方式来 访问文件系统。我们可以通过单击鼠标实现文件的基本操作。但是作为程序员，如 果想通过程序实现文件操作又应该如何做呢?





L







on

soo

文下



O







nesmen



wmre

oeDkcmy



Dnm

le

惬品

ny

6 76

202 m624



n





Fam R



ent g



tnc br



A

图2-1 macOS的GUI

本节的实例已经在Ubuntu 18.04下通过测试，理论上在CentOS 等其他发行版

也不会有问题。

**2.2.1** **文件系统的API**

程序员对文件系统的访问是通过系统API 或系统调用来完成的。每种操作系 统都有一套对文件系统进行访问的API。在类UNIX中，这套API是遵循POSIX 标准的。在 Windows中，虽然API 与POSIX 不兼容，但用法基本一致。

表2- 1所示为部分Linux 和 Windows 中文件操作相关的API。由于本书的重点

第2章知其然 — — 如何使用文件系统

并非API 介绍手册，因此这里列举的只是整个文件API 集合中非常小的一个子集， 其目的是让大家对文件系统的API 有一个整体的认识。

**表2-1部分Linux和Windows中文件操作相关的API**

|  |  |  |
| --- | --- | --- |
| **功** **能** **描** **述** | Linux | Windows |
| 打开文件 | open | CreateFileA |
| 向文件写数据 | write | WriteFile |
| 从文件读数据 | read | ReadFile |
| 关闭文件 | close | FileClose |
| 移动文件指针位置 | Iseek | SetFilePointer |
| 删除文件 | remove | DeleteFileA |

通过表2-1可以看出，无论是Windows 还 是Linux, 其提供的API 基本是一致 的，而且可以从名称很容易猜出该API 的具体作用。

操作系统为用户提供的API 是经过简化的，主要是方便用户的使用。以Linux 中的open() 函数为例，该函数用于打开一个文件，其语法格式如下：

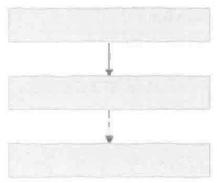
int open(const char\*pathname,int flags);

open()函数关键的输入参数为文件名称(路径),输出结果为一个整数。这个整 数被称为文件描述符(File Descriptor) 或 句 柄(Handle) 。 文件的读/写等操作通过 文件描述符来确定具体的文件，不再关心文件名称。可以看出，文件系统的API 是 非常简洁的。但是，在文件系统内部，具体实现却是要复杂很多。

**2.2.2** **文件访问的一般流程**

前文介绍了文件访问的几个主要的接口，现在主要介绍一下文件访问的一般流 程。操作系统给用户提供了非常简洁和直观的文件访问接口，通常来说一个文件的 访问(读或写)包含打开文件、访问(读或写)文件和关闭文件3个主要步骤。

以 Linux 的接口为例，文件访问的一般流程如图2-2所示。 在该流程中通过文 件名称打开文件，并返回一个文件描述符；之后通过该文件描述符向文件写数据； 完成访问后关闭该文件。



fd=open("flename")

write(fld,data,data\_len)

close(fd)

图2-2 文件访问的一般流程



**文件系统技术内幕**

大数据时代海量数据存储之道

当然，这只是一个简单的实例。实际上操作系统提供的API 和参数要丰富得 多，而且用户的应用场景可能比较复杂，具体使用起来也多种多样。

虽然本文以Linux 平台为例进行的说明，但其他文件系统(如Windows 等 ) 对 文件的访问流程也都大致相同，没有本质的差异。

**2.2.3** **文件内容的读/写实例**

前面我们介绍了访问文件的一般流程，可能大家感觉还会有点抽象。本节将通 过一个实例来实际演示如何读/写一个文件。这个实例主要模拟Linux 的 cp 命令， 也就是实现文件的拷贝功能。

本实例主要用到了文件操作的4个函数，open() 、read() 、write() 和close() 等。 这些函数很简单，我们通过其名称就可以看出作用。下面看一下该实例的代码(见 代码2-1)。

**代码2-1** **拷贝文件的实现**

**copy\_file.c**

|  |  |
| --- | --- |
| 1 |  |
| 2 | \*文件名称：copy\_file.c |
| 3 | \*作 者：SunnyZhang |
| 4 | \*功能描述：拷贝一个文件，模拟Linux的cp命令 |
| 5 | \*== |
| 6 |  |
| 7 | #include <stdio.h> |
| 8 | #include<stdlib.h> |
| 9 | #include <fcntl.h> |
| 10 | #include <errno.h> |
| 11 | #include<sys/types.h> |
| 12 | #include <unistd.h> |
| 13 |  |
| 14 | #define BUF\_LEN 4096 |
| 15 | int main(intargc,char\*argv[]) |
| 16 | { |
| 17 | int src\_fd,dest\_fd; //源文件和目标文件的文件描述符 |
| 18 | char data\_buf[BUF\_LEN]; //用于临时存储读取的数据 |
| 19 | ssize\_t read\_count=0; |
| 20 | ssize\_t write\_count=0; |
| 21 | int ret=0; |
| 22 |  |
| 23 | //打开源文件，源文件只读模式 |
| 24 | src\_fd=open(argv [1],O\_RDONLY); |
| 25 | if(-1=str\_fd ){ |
| 26 | printf("open src fileerror!"); |
| 27 | goto ERR\_OUT; |
| 28 |  |

**第** **2** **章** 知其然——如何使用文件系统



/打开目标文件，目标文件以写模式打开

31 dest\_fd=open(argv[2],0\_WRONLY|O\_CREAT,0644);

32 if(-1=dest\_fd){

printf("open dest file error!");

/\*注意这里跳到的位置，需要将前面打开的文件关闭。在本实例中不关闭也没关系，因进程

\*退出时会自动关闭\*/

36 goto OUT;



/拷贝数据

39 while((read\_count=read(sre\_fd,&databuf,BUF\_LEN ))>0){

40 ssize\_t data\_remain=read\_count;

41 /我们无法保证读取的数据能否被 一 次性写完，所以这里循环写入 while(data\_remain>0){

write\_count=write(dest\_fd,&data\_buf,data\_remain);

if(write\_count<0){// 在写入失败的情况下退出 printf("copy dataerror!");

goto FIN\_OUT;



data\_remain=write\_count;



51

/任何读取或写入失败都要提示用户

53 if (read\_count<0||write\_count<0){

54 printf("copy data error!");



FIN\_OUT:

/ 关 闭 文 件

58 close(dest\_fd); 59

60 close(sre\_fd);

61 ERR\_OUT: 62

63 

在该实例中，分别打开两个文件(第24行～第31行),如果目标文件不存在 则创建新文件。然后不断循环地从源文件读取数据并写入目标文件(第39行～第 50行),直到读完源文件的数据为止。最后将两个文件关闭(第58行～第60行)。

完成上述代码的编写后，我们可以将其编译为一个可执行文件，然后就可以使 用该功能了。具体编译的方法如下：

gcc-ocopy file copy file.c

如果编译没有问题，就可以进行如下测试：

/copy file copy file.c dest.c



**文件系统技术内幕**

大数据时代海量数据存储之道

执行完成上述代码后，我们可以对比一下copy\_file.c 和 dest.c 文件的内容。比如 使用diff命令，可以发现两者的内容是完全 一样的，也就是我们实现了拷贝文件的 功能。

**2.2.4** **关于API** **函数的进一步解释**

前文只是给出一个实例，并没有对所用到的API 做任何解释。本节将对所使用 的API 函数进行一个简单的解释。限于篇幅，本节无法解释所有内容，关于API 的 详细描述大家可以通过man 命令查看帮助文件，或者阅读参考文献[3]。

[**2.2.4.1**](2.2.4.1) **open()函** **数**

open() 函数用于打开/创建一个文件。该函数的语法格式如下： intopen(const char \*pathname,int flags,mode\_t mode);

其中，包含3个参数，分别是文件路径、旗标和模式。

文件路径是文件位置和名称的描述。旗标是对该接口功能的精细化控制，如只 读打开，读写打开等。模式是指文件的具体权限信息，也就是文件的RWX-GUO 属 性，该参数可以省略。

open() 函数执行成功后会返回一个整型变量，这个返回值就是文件描述符。文 件描述符用于标识一个文件，后续的操作都要依赖该文件描述符。

对于文件的访问特性，可以通过open) 函数的flags 参数指定。比如，在打开 文件时，如果flags 包含0\_SYNC 时则表示同步写入，此时要求文件系统将数据写 入持久化设备后再返回。而在默认情况下则是数据写入缓存后就会直接返回。

open() 函数的功能特别丰富，限于篇幅，本节不再逐一介绍。大家可以通过Linux 的 man 命令获得关于该函数的更多解释。可以在Linux命令行执行如下命令获得 帮助信息：

man 2open

执行上述命令后可以输出open(函数的详细说明，如图2-3所示。在上述命令 中数字用于选择具体的章节。这是因为在Linux的手册中可能同一个关键字会有多 个不同的说明，如有些是API 函数、有些是命令等。

**第2章** 知其然——如何使用文件系统



OPEN(2)

NANE

open,openat,creat-open and possibly create a file SYNOPSIS

#include #include #include

<sys/types.h>

ssys/stat.h>

cfcntl.h>

int open(const char \*pathname,int flags);

int open(const char \*pathname,int flags,mode\_t mode); int creat(const char pathname,mode\_t mode);

int openat(int dirfd,const char \*pathname,int flags);

int openat(int dirfd,const char pathname,int flags,mode\_t mode); Feature Test Hacro Requirements far glibc (see feature\_test\_macros(7)):

openat():

Since glibc 2.10:

POSTXC\_SOURCE =200809L Before glibc 2.10:

ATFILE\_SOURCE

\_

图2-3 open(函数的详细说明

[**2.2.4.2**](2.2.4.2) **read()函** **数**

打开文件之后就可以对文件进行读/写操作。先介绍一下读取操作，可以使用 read()函数来实现，该函数的语法格式如下：

ssize\_t read(int fd,void \*buf,size\_t count);

该函数有3个参数，第1个参数是文件描述符，用于确定从哪个文件读取数 据；第2个参数是缓冲区，用于存储读取的数据，在使用前需要分配内存空间；第 3个参数是读取的字节数。read(函数的返回值如果大于0则表示实际读取的字节 数，小于0则表示该函数出错。

代码2-1就有read(函数的应用，while 循环的条件是读取的字节数。如果实际 读取了数据，则进入循环体，否则跳出循环。

[**2.2.4.3**](2.2.4.3) **write()函** **数**

write()函数的用法与read() 函数的用法类似，也包含3个参数。其中，第1个 参数是文件描述符，用于确定向哪个文件写入数据；第2个参数是缓冲区，用于存 储待写入的数据；第3个参数是写入的字节数。该函数的语法格式如下：

ssize\_t write(int fd,const void \*buf,size\_t count);



文件系统技术内幕

大数据时代海量数据存储之道

在通常情况下，write()函数的返回值与参数count的值相同，在某些情况下可 能会出现返回值小于count。因此，在一般情况下，通过一个循环来保证读取的数 据被全部写入。在极端情况下会出现写入出错，如磁盘容量不足或磁盘出现故障等， 此时返回值小于0。

[**2.2.4.4**](2.2.4.4) **close()函** **数**

close()函数的作用是将文件关闭，只有一个参数，就是之前打开的文件描述符。 其语法格式如下：

int close(int fd);

本节主要介绍了Linux 文件系统在程序员层面的API 接口。主要集中在单个文 件访问层面。其实文件系统的访问接口很多，除了文件访问，还有目录访问、文件 锁和映射等，这些内容后续再做介绍。

读到这里，不知道大家是否有如下几个疑问。

(1) 为什么这些API通过一个整数(文件描述符)来标识一个文件?

(2)当多个进程打开同一个文件时，文件描述符在不同进程中是怎样的?

要想解答上述问题，还得继续深入挖掘文件系统的实现细节，在后面章节对文 件系统原理的介绍中我们会逐渐拨开疑云。

**2.3** **如何遍历目录中的文件**

文件系统有一个常用的功能就是查看某个目录的文件列表。这个功能对应的命 令行工具就是Is 命令。而在GUI 管理工具中，其实就是展示在我们面前的目录和 文件列表等内容。

从本质上来说，目录与文件并没有太大的差异。我们也可以将目录理解为一个 文件，其中的数据是一些此目录下的所有文件名相关的内容。关于目录内容的相关 原理，会在后续章节进行详细介绍，本节不再赘述。

本节从程序员开发的角度介绍一下如何查看目录的内容，也就是遍历目录中的 文件。Linux 有一个专门的API 来实现目录的遍历，这个API 就 是readdir。下面实 现一个类似ls 的命令，但是功能上要比1s 命令弱很多，如代码2-2所示。

第2章知其然 — — 如何使用文件系统

**代** **码** **2** **-** **2** **遍** **历** **目** **录** **的** **实** **现**

list\_dir.c

|  |  |
| --- | --- |
| 1 | /\*: |
| 2 | \*文件名称：list\_dir.c |
| 3 | \*作者：SunnyZhang |
| 4 | \*功能描述：本程序用于模拟Linux的ls命令 |
| 5 | \* |
| 6 |  |
| 7 | #include <stdio.h> |
| 8 | #include <sys/types.h> |
| 9 | #include <dirent.h> |
| 10 | #include ≤unistd.h> |
| 11 |  |
| 12 | int main(void) |
| 13 | { |
| 14 | DIR\*dir; |
| 15 | struct dirent\*ptr; |
| 16 |  |
| 17 | //打开当前目录，本实例只实现了遍历当前目录的功能 |
| 18 | dir=opendir("/"); |
| 19 | /逐个读取目录项 |
| 20 | while ((ptr=readdir(dir))!=NULL){ |
| 21 | printf("%s",ptr->d name);//输出目录项 |
| 22 |  |
| 23 | printf("\n"); |
| 24 | closedir(dir); |
| 25 | return 0; |
| 26 |  |

这个程序的实现很简单，只是输出当前目录下的所有文件和子目录。编译上述 代码后运行，可以得到如下结果。对比 readdir 命令的结果和1s 命令的结果，可以 看出没有太大的差异。差异在于我们实现的命令输出了当前目录(.)和父目录(..)。

rootasunnyzhang-VirtualBox:-/code/filesystem#./list list list\_dir.c.

rootesunnyzhang-VirtualBox:~/code/filesystem 共ls

list list\_din.c

图2-4 输出结果

通过阅读上述代码我们可以知道，这里目录项的内容是以结构体 dirent (directory entry的缩写)存储的。我们可以看一下该结构体的定义(来自glibc2.32), 如代码2-3所示。

**代** **码** **2** **-** **3** **目** **录** **项** **数** **据** **结** **构**

**linux/bits/dirent.h**

|  |  |
| --- | --- |
| 22 | struct dirent |
| 23 | { |
| 24 | #ifndef USE FILE OFFSET64 |



**文件系统技术内幕**

大数据时代海量数据存储之道

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 25 | \_ino\_td\_ino; //inode ID | | | |
| 26 | \_off\_td\_off; //在目录文件中的偏移 | | | |
| 27 | #else | | | |
| 28 | \_ino64\_td\_ino; | | | |
| 29 | off64\_td\_off; | | | |
| 30 | #endif | | | |
| 31 | unsigned short int d\_reclen; //记录的长度 | | | |
| 32 | unsigned char d\_type; //文件类型 | | | |
| 33 | char d\_name[256]; //文件名称 | | | |
| 34 | ; | | | |
| 通过观察该数据结构就会发现，这个数据结构只有inode ID和文件名称等信息 与ls 命令展示的相关。那么Is命令显示的文件的详细信息(如文件的创建时间、 大小和权限等)又是如何获取的呢?  要想回答这个问题，就要看一下系统提供的一个API 函数。这个API 函数就 是statO函数，该函数的语法格式如下：  int stat(const char \*path,struct stat \*buf);  从 stat (函数的语法格式可以看出，该函数最主要的功能是返回一个stat 类 型 的结构体。该结构体的定义如代码2-4所示(来自glibc 2.32)。从该结构体的定义 我们可以看出，这里面包含文件非常详细的信息。通过这些信息，我们完全可以实 现一个完整版的ls 命令。  **代** **码** **2** **-** **4** **文** **件** **属** **性** **数** **据** **结** **构** | | | | |
| **linux/bits/stat.h** | | | | |
| 58 | struct stat | | | |
| 59 | { | | | |
| 60  61 | dev tst fcld64C | dev; imo | t\_ino64Lst imo); | / / 设 备 I D  //文件序列号，也就是inode ID |
| 62 | fmode\_t st\_fmode; | | | //文件模式 |
| 63 | nlink\_t st\_nlink; //链接数量 | | | |
| 64 | uid\_t st\_uid; //文件所有者的用户ID | | | |
| 65 | gid\_t st gid; //文件所属组的组ID | | | |
| 66 | \_dev\_t st\_rdev; //文件是设备的情况下，此成员为设备号 | | | |
| 67 | \_dev t padl; | | | |
| 68 | \_field64( off\_t, off64\_t,st\_size); //以字节为单位的文件大小 | | | |
| 69 | \_blksize\_t st\_blksize; //I/O最佳块大小 | | | |
| 70 | int\_pad2; | | | |
| 71 | field64( blkcnt t, blkcnt64\_t,st\_blocks);/字节块 | | | |
| 72 | #ifdef USE\_XOPEN2K8 | | | |
| 73 | /\*以纳秒为单位的时间截采用与"struct timespec"结构体 | | | |
| 74 | \* 等 效 的 格 式 存 储 。 | | | |
| 75 | \*尽量使用该类型，但UNIX命名空间的规则不允许 | | | |
| 76 | \*标识符"timespec"出现在<sys/stat.h>头文件中。 | | | |
| 77 | \*因此，我们必须在严格兼容原始特殊性的标准下处理 | | | |
| 78 | \* 此 头 文 件 的 使 用 \* / | | | |

**第2章** 知其然——如何使用文件系统

|  |  |  |
| --- | --- | --- |
| 79 | struct timespec st\_atim; //最后访问该文件的时间 | |
| 80 | struct timespec st\_mtim; | //最后修改该文件的时间 |
| 81 | struct timespec st\_ctim; | 最后状态发生变化的时间 |
| 82 | #define st\_atime st\_atim.tv\_sec |  |
| 83 | #define st\_mtime st\_mtim.tv\_sec |  |
| 84 | #define st\_ctime st\_ctim.tv\_sec |  |
| 85 | #else |  |
| 86 | time\_t st\_atime; | //最后访问该文件的时间(单位是秒) |
| 87 | unsigned longint st\_atimensec; | //最后访问该文件的时间(单位是纳秒) |
| 88 | \_time\_tst\_mtime; | /最后修改该文件的时间(单位是秒) |
| 89 | unsigned longint st\_mtimensec; | //最后修改该文件的时间(单位是纳秒) |
| 90 | \_time\_tst\_ctime; | //最后状态发生变化的时间(单位是秒) |
| 91 | unsigned longint st\_ctimensec; | //最后状态发生变化的时间(单位是纳秒) |
| 92 | #endif |  |
| 93 | int glibc\_reserved[2]; |  |
| 94 | }; |  |

关于如何实现一个完整版的Is 命令本书不再赘述，大家可以自己思考并试着 实现。如果实在不知道怎么写，则可以参考Is 命令的源代码实现。

**2.4** **格式化文件系统与挂载**

实际上格式化与挂载(Windows不需要手动挂载)文件系统才是文件系统使用 的第一步。格式化文件系统相当于在块设备上创建一个文件系统，而挂载则是将该 文件系统激活(在操作系统目录树呈现)的过程。

在安装操作系统时，安装程序已经对系统磁盘进行了格式化操作。所以，在通 常情况下我们不太会感知到在使用磁盘之前需要格式化。但是，如果计算机配置了 多块硬盘，则非系统硬盘在使用之前需要格式化才可以使用。

如果是Windows, 则格式化操作非常简单。只需要右击盘符弹出一个快捷菜单， 然后选择“格式化”命令，如图2-5所示，打开“格式化”对话框，如图2-6所示。

在“格式化”对话框中，单击“开始”按钮，系统就可以帮我们完成磁盘整个 格式化的过程。当然，在单击“开始”按钮之前可以根据需要调整文件系统的参数， 如文件系统类型、分配单元的大小等。

当系统完成格式化之后，双击磁盘盘符进入该磁盘，然后我们就可以做一些具 体的操作了，如拷贝文件或新建文件等。

**文件系统技术内幕**

**大数据时代海量数据存储之道**



窗量(P:

852 GB

文件系统(F) NTFS ( 默 认

分配单元大小A) 4096字节

国定到开始”屏莓(P) 添加到压孀文件(AL

酒加到"Archive.rar'() 压嘴并E-mail

压缩到"Archive.rar ' 并Emall 添加到压难文件(2)-

回扫猫病毒(电脑营部 格式化 (A)

复制(9

创建快建方式(S) 重命名(M)

居性(R)

图2-5 选择“格式化”命令

还原设备的默认值(D)

卷标(L)

Data

格式化选项(O)

☑快速格式化(Q)

开始(S 关闭(9

图2-6“格式化”对话框

在 Linux 操作系统进行格式化稍微有些门槛，但并没有太大的难度。Linux命

令行终端通过命令实现块设备的格式化操作。 其语法格式如下： mkfs.ext4/dey/sdb

这里/dev/sdb就是一个块设备，可以理解为磁盘。命令名称分为两部分，mkfs (make filesystem的 简 写 ) 表 示 格 式 化 ， 而ext4 则表示文件系统的类型。当然，该 命令其实具有非常丰富的参数，如设置文件系统块大小等，大家可以通过 man 命 令进一步了解，本文不再赘述。

但是在Linux操作系统中完成格式化后，我们并不能像Windows那样直接进 入/dev/sdb这个磁盘设备拷贝文件，或者进行其他文件操作。这里需要额外操作一 步，也就是将该磁盘设备挂载到某个目录下面。

假 设 现在有一个目录(/mnt/ext4\_test),执 行如下命令就可以将刚才格式化的文 件系统挂载了。

mount/dev/sdb/mnt/ext4 test

如果没有提示错误，那么这个格式化后的磁盘就挂载到Linux文件系统目录树 的/mnt/ext4 test 目录下面。此时，我们对该目录的访问就是对磁盘数据的访问。这 个似乎是 一个很神奇的动作，具体原理是什么呢?请参考后续章节的解释。

通过手动挂载的文件系统在操作系统重启后就不存在了，如果想要访问该磁盘 的内容，则此时还需要重新执行mount 命令进行挂载。 有什么方法可以在操作系统

第 2 章 知 其 然 — — 如 何 使 用 文 件 系 统

启动过程中自动挂载?当然有，那就是通过fstab配置文件来实现，如图2-7所示， 第3行代码是针对本实例增加的配置项。

1 D=574279fd-876c-4cdB-b30e-bf122fa069f3/ext4 defaults 00

2./swap.img none swap sW 0 0

3.u00-c0210506-564a-4236-86eb-9cb7757ba8d6/mnt/ext4\_test ext4 defaults 00

图2-7 fstab配置文件实例

在上述配置项中每行分为6段。其中，第1个表示待挂载的设备，如磁盘，其 实这里不仅可以是具体的设备，还可以是标签或文件系统UUID; 第2个是挂载点 (挂载点是一个挂载了新文件系统的目录);第3个是文件系统类型；第4个是挂载 选项，本书选用默认值；第5个是被dump 命令使用的选项；第6个是被fsck 命令 使用的选项。每个选项的详细含义可以通过执行man fstab命令获得。

**2.5** **文件系统与权限管理**

现代操作系统通常支持多用户操作。也就是说同一个操作系统可以允许很多用 户登录并操作其中的资源。这样多用户场景就存在一个资源隔离和保护的问题，也 就是说在通常情况下A 用户应该只能访问A 用户的资源，B 用户只能访问B 用 户 的资源，避免相互访问，造成资源使用的混乱和安全问题。

下面介绍文件系统的权限相关的内容。以Windows 为例，右击文件，在弹出的 快捷菜单中选择“属性”命令，打开“属性”对话框，在该对话框中可以查看某个 用户对该文件的访问权限。

在 “tmp.txt 属性”对话框中(见图2-8),通过选择“安全”选项卡就可以看到 系统的用户列表及权限信息。当选择某个用户时就可以看到该用户对本文件的访问 权限情况。

除了可以查看文件的权限信息，在“tmp.txt 属性”对话框中还可以修改某个用 户对当前文件的访问权限。

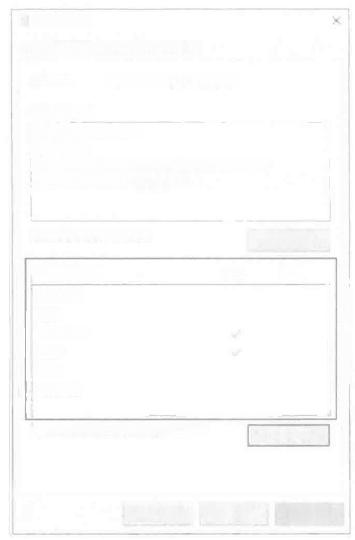
除了资源隔离的场景，还有一种资源共享的场景。比如， 一个公司有多个部门。 A 部门的所有用户应该都可以访问该部门下的资源，而不允许访问其他部门的资 源。这些需求都与文件系统的权限管理有关系。

当 然 ，Linux 也有权限相关的特性。只不过Linux中的操作大多是通过命令来

完成的。接下来以Linux 为例从实际操作和原理方面详细解释一下文件系统是如何 进行权限控制的。

**文件系统技术内幕**

大数据时代海量数据存储之道

tmp.bt属性

常规安全 详细信里以前的版本

对象名称 DNCodes\filesystem\tmp.bt

组或用户名(GH

腿 Authenticated Users SYSTEM

随 Administrators SUNNYZHANGAdministrators) 这 Users (SUNNYZHANG\Users)

要更改权限，调单击编辑”。

Users的权限(P) 完全控制

修改

读取和执行 读取

写入

特操权限

有关特殊权限或离级设置，清单击”高级。

编辑(F)-

充许 拒绝

高级(M)



确定 取消 直沟

图 2 - 8 “tmp.txt 属性”对话框

**2.5.1 Linux权限管理简介**

Linux最常用的权限管理就是RWX-UGO权限管理。其中， RWX是Read 、

Write和eXecute的缩写。而 UGO则是User 、Group和Other的缩写 。通过该机制 建立用户和组实现对文件的不同的访问权限。

如果在Linux 的某个目录下执行Is -alh 命令，就可以看到如图2-9所示的结 果。其中，就包含了文件的所属用户和组的信息，以及对应的权限信息。

total 3ZK

drnxr-xr×

2,root

2s'root

Troot

i'sunnyzhangi 21root

z,root

root

liroot

Ft

iroot iroot

root Iroot

root root

root

4.0K Feb 4.08:55.

4.0K Feb 408:54

7 Feb 4.08:54.testi

7 Feb 4.08:54 test2 Z Feb 408:54 test3

7 Feb 408:54 test3pl

5 Feh 4.08:55 test3\_st>test3

7 Feb 4.08:54 test4

7 Feb 408:54 test5

dnx--- -n-rr

-n-rr-

-ner

-N-r--r

inwxnaxna

rw-r

Lr-rr

I'itworld12siitoorld123

权陨信里 所属用户 所厘祖

图2 - 9 Linux 文件的权限信息

权限描述信息是每行前面rw- 等字符描述的内容，而后面的sunnyzhang 和 root 等字符则是用户和组的信息。通过两者的结合，在读/写等流程中就可以判定访问者 是 否 有 相 应 的 权 限 。

**第2章知其然——如何使用文件系统**



**2.5.2** **设置文件的** **RWX** **权限**

[**2.5.2.1**](2.5.2.1) **基于API** **的权限设置**

在Linux 中有相关的API 来修改这个权限。修改权限的语法格式如下：

intchmod(const char \*pathname,mode t mode);

其中，第1个参数是文件名，第2个参数是目标权限。执行chmod (函数可以 将文件设置为目标权限。

接下来看一下chmod() 函数的用法。假设这里有一个测试文件，名称为test.bin。 该文件的初始权限信息如图2-10上半部分所示(-rW-r---) 。 通过代码2- 5,将文件 的权限设置为S\_IRUSR, 也就是所属用户可读。编译运行该代码后，发现文件的权 限变成如图2-10下半部分所示的内容(------)。

**代** **码** **2** **-** **5** **文** **件** **的** **权** **限** **修** **改**

**change\_mode.c**

|  |  |
| --- | --- |
| 1 |  |
| 2 | \*文件名称：change\_mode.c |
| 3 | \*作者：SunnyZhang |
| 4 | \*功能描述：修改文件的权限信息 |
| 5 |  |
| 6 | \*= |
| 7 |  |
| 8 | #include<stdio.h> |
| 9 | #include<fentL.h> |
| 10 | #include <errno.h> |
| 11 | #include <sys/types.h> |
| 12 | #include<sys/stat.h> |
| 13 |  |
| 14 | int main(int arge,char\*argv[) |
| 15 |  |
| 16 | int ret=0; |
| 17 |  |
| 18 | ret=chmod("test.bin",S\_IRUSR);//S\_IRUSR是一个宏定义，其值为00400 |
| 19 | if(ret<0){ |
| 20 | printf("change mode failed"); |
| 21 |  |
| 22 | return ret; |
| 23 |  |

这种方法的缺点是将文件原始的权限都覆盖了。比如，想要为某个文件添加某 一项权限，似乎并不太好实现。



文件系统技术内幕

大数据时代海量数据存储之道

rootesunnyzhang:s/code/chmod 1 total 24

dnwxr xr-x2 root root 40960t 31.00:22- dnvxr-xr-x5 root root 40960ct 3100:12

-nOr-xr-x1 root root 8352/0t 3100:22 change mode MrrI root root 538 Oct 3160:22 change mode.c

ln-rri root root 00ct 31.00:22 test.bin

中otasunnyzhang:-/code/chmode ./change\_mode

poresunnyzhang:s/code/chmde il btal 24

xr-xr-x2 root root 40960ct 31.00:22

r-xr-x6 rout root 40960ct 31100:12

hor-xr-xI root root 83520ct 3100:22 change\_mode -I root root 5380ct 3100:22 change mde.C

二 ront root 00ct 3100:22 test.bin

**图2-10** **设置文件的权限**

其实仍有实现的办法，就是先通过stat接口来获取文件的原始权限信息， 添 加

期望的权限后再设置该文件的权限。这次给该文件增加执行权限，如代码2-6所示。

**代** **码** **2** **-** **6** **文** **件** **的** **权** **限** **修** **改**

**change** **mode** **c**

|  |  |  |
| --- | --- | --- |
| 1 | **.**    /\*= |  |
| 2 | \*文件名称：change\_mode.c |  |
| 3 | \*作 者：SunnyZhang |  |
| 4 | \*功能描述：修改文件的权限信息 |  |
| 5 | \*— | \*/ |
| 6 |  |  |
| 7 | #include <stdio.h> |  |
| 8 | #include <fcntl.h> |  |
| 9 | #include <errno.h> |  |
| 10 | #include<sys/types.h> |  |
| 11 | #include<sys/stat.h> |  |
| 12 | #include <unistd.h> |  |
| 13 |  |  |
| 14 | int main(int argc,char\*argv[]) |  |
| 15 | { |  |
| 16 | int ret=0; |  |
| 17 | struct stat file\_info; |  |
| 18 | int mode=0; |  |
| 19 |  |  |
| 20 | ret=stat("test.bin",&file\_info); | /获取文件原始的权限信息 |
| 21 | if (ret<0){ |  |
| 22 | printf("get file info error!"); |  |
| 23 |  |  |
| 24 |  |  |
| 25 | mode=file info.st\_mode|S\_IXUSR; | //为了区分，这里增加执行权限 |
| 26 | ret=chmod("test.bin",mode); | //设置文件的新权限 |
| 27 | if (ret<0){ |  |
| 28 | printf("change mode failed"); |  |
| 29 |  |  |
| 30 | return ret; |  |
| 31 |  |  |

第2章知其然 — — 如何使用文件系统

重新编译并执行该代码可以看出，这次是在原始权限的基础上增加了执行权

限，而不是把原来的权限都给覆盖了，执行结果如图2-11所示。

rootesunnyzhang:~/code/chmod L1 total 24

drwxr-xr-x2 root root 40960ct 3100:50

drwxr-xr-x6 root root 40960ct 3100:12../

-rwxr-xr-x1 root root 85040ct 3100:50 change\_mode\* =NW-rr=1 root root 7390ct 31.00:50 change\_mode.e -Nw-r-11 root root 00ct 3100:49 test.bin

rootasunnyzhang:~/code/chmod#./change\_mode

rootesunnyzhang:~/code/chmod#L total 24

dnwxr-xr-x2 root root 40960ct 3100:50./ dnwxr-xr-x6 root root 40960ct 3100:12

rwxr-xr-xI root root 85040ct 31.00:50 change\_node\*

-W- root root 7390ct 3100:50 change\_mode.c

-TwXr-I rootroot 0Oct 3100:49 test.bin

图2-11 为文件增加执行权限

这个API 只能修改文件的权限信息，无法修改文件的所属用户和组的信息。如

果想要修改所属用户和组的信息，则可以使用chown() 函数，该函数的语法格式 如 下 ：

nt chown(const char\*pathname,uid towner,gid t group);

如果想要设置文件的所属用户和组，只需要将用户ID 和 组ID 传进去即可。由 于比较简单，这里不再举例说明。

[**2.5.2.2**](2.5.2.2) **基于命令的权限设置**

通过编程的方式可以实现文件的权限修改，但是在日常操作中非常不方便。不 必着急，Linux 已经为我们提供了相关的命令行工具，这些命令行工具与函数同名， 如chmod 和chown 等。

**1.chmod** **命令**

还是以前面添加执行权限为例，可以在命令行中执行如下命令： sudo chmod+xtest.bin

执行上述命令后，就可以得到与前面程序一样的结果，如图2-11所示。可以 看出，通过命令行的方式对文件的权限进行修改要简单快捷得多。

由于底层是采用二进制的方式存储的，chmod命令也是支持通过数字的方式修 改其权限属性的。比如，执行如下命令：

sudo chmod 777 test.bin

其 中 ，777就是使所有的RWX都设置为1,即可以被任何用户和组访问。执 行结果如图2-12所示。



**文件系统技术内幕**

**大数据时代海量数据存储之道**

rootesunnyzhang:-/code/chmode l1 total 24

drwxr-xr-x2 root root 40960ct 31102:25./ drwxr-Xr-x6root root 40960ct 3100:12.

-rwxr-xr-x1 root root 85040ct,3100:50 change\_mode\* Tw-r 1 root raot 7390ct3100:50 change\_mode. -rwxnwxrwx I root root 00ct 3100:49 test.bin

**图2-12** **通过命令设置权限**

**2.chown** **命令**

chown命令用于修改文件的所属用户信息。比如，将属于root 的 文 件test.bin 改为属于sunnyzhang, 可以执行如下命令：

chown sunnyzhang:sunnyzhang test.bin

**3.chgrp** **命令**

从名字上可以看出，chgrp 命令用于修改文件的所属组信息。使用方法很简单， 本节不再赘述。

关于RWX-UGO的权限访问控制就先介绍到这里。其实除了RWX-UGO权限 控制，还有其他类型的权限控制，如ACL 权限控制。

**2.5.3** **设置文件的ACL** **权限**

前文介绍了RWX-UGO的权限控制方法，但是这种方法过于简单，很多场景 无法满足要求。为了让大家理解为什么有些问题无法通过RWX-UGO 权限管理解 决，列举一个大家都会遇到的实例。

假设有一个工资单目录，该工资单目录存储了公司所有人员的工资单文件。对 于工资单目录中的文件，显然财务人员(可能是张三、张五等多个人)是可以读/写 的，因为他们要生成这个工资单，并可以更正错误。例如，对于李四的工资单文件， 李四可以读但不允许写。出于对工资的保密，其他人不允许读，也不允许写。为了 让大家更加清楚地理解上述关系，通过图2-13进行表示。

这种权限的要求采用RW X - UGO的方式就很难实现。因为采用RW X - UGO进 行权限控制只能包含文件所有者和其他人， 而无法控制多个不同的具体人。为了解 决这种复杂的权限管理问题，Linux还有另外一套权限控制方法，也就是ACL权限

控制方法。

第 2 章 知 其 然 — — 如 何 使 用 文 件 系 统



财务人员 不允许读 其他人

可读，可写

不允许写

李四的工资单文件

可读 不允许写

李四本人

图2-13 工资单的权限设置

ACL(Access Control List,访问控制列表),一个针对文件/目录的访问控制列 表。它在RWX-UGO 权限管理的基础上为文件系统提供一个额外的、更灵活的权 限管理机制。ACL允许给任何用户或用户组设置任何文件/目录的访问权限，这样 就能形成网状的交叉访问关系。

ACL的原理很简单，就是在某个文件中增加一些描述用户名/组名与权限的 “键-值”对。比如，用户sunnyzhang 具有读/写权限 (rw), 可以为该文件添加 sunnyzhang:rw 信息。这样在内核中就可以根据该描述信息确定某个用户对该文件 的权限。

Linux 有几个命令行工具来对文件/目录的ACL 属性进行设置。使用起来也比 较简单。接下来看一下如何获取一个文件的ACL 属性，或者为一个文件设置ACL

属性。

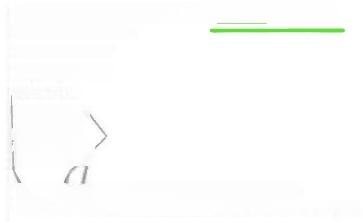
**1** **.** **获** **取** **文** **件** **的ACL** **属** **性**

通 过getfacl 命令可以获取文件的ACL 属性。比如test.bin 文件，我们通过下 面命令就可以获取该文件的ACL 属性，如图2-14所示。由于ACL 兼 容RWX, 因 此即使在没有做ACL 设置的情况下也是可以获得相关内容的。我们对比一下图中 箭头所指向的内容可以发现，ACL 属 性 与RWX 属性是完全一致的。



文件系统技术内幕

大数据时代海量数据存储之道

rootesunnyzhang:-/code/acla getfacl test.bin

file:test.bin

#owner:root

共group:root user::Nw-

group:

mask::r

otherir

cotesenglzhang:-/code/aclll test.bin

-T +1 root root θ0ct 3104:00 test.bin

图2-14 ACL属性与RWX属性对照

**2.设置文件的ACL** **属性**

如何设置一个文件的ACL属性呢?可以通过setfacl命令用来设置文件或目录 的ACL 属性，该命令的语法格式如下：

setfacl [-bkRd][-m|-x}acl 参 数 ]文件/目录名

虽然选项比较多，但是常用的选项主要是-m和-x, 前者用于给文件/目录添加 ACL 参数，后者用于删除某个ACL 参数。其他选项的作用请参考man 手册，本节 不再赘述。

下面列举一个简单的实例，test.bin 文件本来属于 root 用户，但是期望该文件 被zhangsan用户读/写，代码如下：

setfacl -m u:zhangsan:rw test.bin

当设置完成后，再次通过getfacl 命令获取该文件的ACL 属性时会发现结果中 多了一行代码“user:zhangsan:rw-”, 如图2-15所示。这一行代码就是我们添加的用 户zhangsan 对该文件的读写权限(rw)。

rootesunnyzhang:~/code/acle getfacl test.bin file:test.bin

共owner:root 来group:root

user::

user:zhangsan:T二 新增的ACL 权限

group:!r mask::- othersir

图2-15 ACL权限设置实例

ACL 除了拥有命令行工具，还拥有一套API, 以方便程序员通过编程的方式来 修改文件的ACL 属性。但是ACL 中 的API 与 RWX 中 的API 相比，使用起来还是 比较复杂的。如果想要通过编程的方式修改ACL属性，则首先需要安装一个libacl 库，然后使用该库中的API来做相关的操作。关于libacl 库 中API 的用法本章不再 介绍，后续章节在原理介绍时会详述相关代码。

至此，大家应该对如何修改文件的权限有了一个整体的认识，但是对其实现原

第2章知其然——如何使用文件系统

理可能还不太清楚。不过没关系，我们在后续章节将结合代码详细介绍Linux 是 如 何实现权限管理的。

**2.6** **文件系统的锁机制**

我们知道对于临界区的资源处理需要锁机制。比如，在多线程情况下，如果访 问某些共享的数据结构，那么需要自旋锁或互斥锁来保护，防止并发读/写导致数据 的不一致。对于文件系统的文件，同样存在多线程或进程同时访问的问题，如果没 有锁机制，则可能导致文件数据的损坏或不一致。

本节将介绍文件系统中文件锁的相关内容，包括文件锁的类型、API 和基本 用法。

**2.6.1** **文件锁的分类与模式**

从大类上来分，文件锁分为劝告锁(Advisory Lock)和强制锁(Mandatory Lock)

两种类型。

劝告锁是一种建议性的锁，通过该锁告诉访问者现在该文件被其他进程访问， 但不强制阻止访问。这就好比我们去景区旅游，看到一个牌子写着“游客勿入”, 但是门是开着的。如果我们不在乎警告，还是可以进去的。

强制锁则是在有进程对文件锁定的情况下，其他进程是无法访问该文件的。还 以旅游为例，你走到一个地方，虽然没有牌子写着“游客勿入”,但是大门是紧锁 的。在这种情况下即使你想冲进去，也是没办法进去的。

在使用模式上，无论是劝告锁还是强制锁都分为共享锁和排他锁两种。共享锁

与排他锁的差异在于当进程A 持有锁的情况下，其他进程试图持有该锁时产生的 行为不同。

共享锁(Shared Lock): 在仟意时间内， 一个文件的共享锁可以被多个进程拥 有，共享锁不会导致系统进程的阻塞。也就是说，当进程A 持有共享锁的情况下， 进程B 试图持有该共享锁也是可以的，而且不会造成对进程B 的阻塞。这非常适 用于两个进程同时读取文件数据的场景，如图2-16所示。

排他锁(Exclusive Lock):在任意时间内， 一个文件的排他锁只能被一个进程 拥有。也就是说，当一个进程A 持有排他锁时，另外一个试图获取该锁的进程B 将 被阻塞，直到占用锁的进程释放后，进程B 才能继续，如图2-17所示。



文件系统技术内幕

大数据时代海量数据存储之道

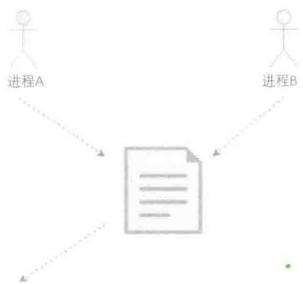


图2-16 共享锁示意图 图2-17 排他锁示意图

为了让大家更加清晰地理解共享锁与排他锁的关系，下面通过表2-2进行一个 比较全面的表述。其中第一列表示某个进程已经获取了某种类型的锁。后面两列则 表示当另一个进程期望获取该类型的锁时是否可以获得。

**表2-2共享锁与排他锁的互斥关系**

|  |  |  |
| --- | --- | --- |
| **存在的锁/请求类型** | **共** **享** **锁** | **排** **他** **锁** |
| 无 | 可获取 | 可获取 |
| 共享锁 | 可获取 | 拒绝 |
| 排他锁 | 拒绝 | 拒绝 |

例如，文件的读/写可以结合共享锁与排他锁来实现，写文件使用排他锁，读文 件使用共享锁。当有进程在写文件时，其他所有进程都不允许写或读此文件。当没 有进程在写文件时，多个进程可以同时读此文件。

本节主要对文件锁的概念和类型进行了介绍，下面以Linux 中 API 为例介绍一 下如何使用文件锁。

**2.6.2** **Linux** **文件锁的使用**

在Linux中，文件锁的特性是通过flock()和fcntl(两个函数对外提供的。这两

个函数都可以实现文件加锁和解锁的流程，但是后者要比前者的特性更加丰富。

[**2.6.2.1**](2.6.2.1) **flock()函** **数** **的** **使** **用** **简** **析**

下面先介绍一下flock API的 使 用 ，flock() 函数的语法格式如下：

int flock(int fd,int operation)

可以看出，该函数有两个参数， 一个是文件的句柄，另一个是具体的动作。

第2章知其然—— 如何使用文件系统

flock()函数行为的差异依赖于operation参数，该参数可以是如下几种情况。 ·LOCK SH:表示加共享锁(Shared Lock) 。

·LOCK EX:表示加排他锁(Exclusive Lock) 。 ·LOCK UN:表示释放锁(Unlock) 。

可以看出，文件锁的加锁和解锁都是由flock(函数实现的。在了解了上述参数 的含义之后，再使用该函数就不太难了，此处就不再举例说明。

[**2.6.2.2**](2.6.2.2) **fcntl()函数的使用简析**

fcntl()函数实现的特性要更加丰富一些，它不仅可以用于锁操作，还可以用于 其他操作，这主要依赖参数cmd 的值。fcntl()函数的语法格式如下：

int fcntl(int fd,int cmd,…//arg );

可以看出，该函数有两个主要的参数，一个是fd,表示目标文件描述符：另一 个是cmd,用于确定具体的操作，对于文件锁cmd 来说，可以是F GETLK 、F SETLK 和 F\_SETLKW。其实后面还有第3个参数，这个参数是可变参数。对于文件锁操 作，第3个参数的类型为struct flock。此时函数的语法格式如下：

int fcntl(int fd,int cmd,struct flock \*lock);

lock参数是文件锁的详细属性信息，它描述了我们想要添加什么类型的文件 锁，以及其他一些描述信息。该结构体包含的内容如代码2-7所示。

代码2-7 文件锁结构体



**bits/fentl.h**

36 struct flock

37 {

38 short **intl type;**// 类型包括FRDLCK、FWRLCK和FUNLCK

39 shortintl\_whence;

40 #if\_WORDSIZE=64||Idefined\_USE\_FILE\_OFFSET64

41 \_off\_t1\_start; 42 \_off\_t1 len;

43 #else

44 off64tl start; 文件锁的起始位置

45 off64 tI len; //锁定区间的长度，0 表示到文件尾

46 #endif

47 pid\_tl pid; //持有文件锁的进程ID

48 ;

通过上述参数可以看出，该方法不仅可以实现各种文件锁类型，文件锁的粒度 也会更细一些。其中，成员1\_start 与1\_len 用于描述锁定的文件的范围。

成员1\_type 描述了文件锁操作的具体类型，它的值可以是 F\_RDLCK、

F\_WRLCK 和 F\_UNLCK 等。其中，F\_RDLCK 是 加 读 ( 共 享 ) 锁 ，F\_WRLCK 是



**文件系统技术内幕**

大数据时代海量数据存储之道

写(排他)锁，而F\_UNLCK 是解锁操作。当对文件进行加锁或解锁操作时，只需 要填充相应的参数，并调用该接口即可。

为了更加清楚地说明上述各个参数的用途，下面列举一个实例。在该实例中定 义了一个排他锁。同时，通过F\_SETLKW 参数让 fcntl(函数添加一个需要等待 (Wait) 模式的锁(第30行),如代码2-8所示。

**代码2-8文件锁使用实例**

**lock\_file.c**

|  |  |
| --- | --- |
| 1 | /\*— |
| 2 | \*文件名称：lock\_file.c |
| 3 | \*作 者：SunnyZhang |
| 4 | \*功能描述：对一个文件进行加锁 |
| 5 |  |
| 6 |  |
| 7 | #include <stdio.h> |
| 8 | #include≤stdlib.h> |
| 9 | #include <fcntl.h> |
| 10 | #include <errno.h> |
| 11 | #include <sys/types.h> |
| 12 | #include <unistd.h> |
| 13 |  |
| 14 | #define BUF\_LEN 4096 |
| 15 | int main(int argc,char\*argv[]) |
| 16 | { |
| 17 | int ret=0; |
| 18 | struct flock test\_lock={ |
| 19 | 1\_whence=SEEK\_SET |
| 20 | 1\_type= F WRLCK //排他锁 |
| 21 | }; |
| 22 |  |
| 23 | int fd=open("test.bin",O)RDWR); |
| 24 | if(fd<0){ |
| 25 | printf("openfile failein"); |
| 26 | goto OUT; |
| 27 | } |
| 28 |  |
| 29 | printf("before lock fileln") |
| 30 | ret=fcntl(fd,F SETLKW,&test\_lock); /加锁操作，如果已经被加锁则等待 |
| 31 | if(ret<0){ |
| 32 | printf("lockfile failedn"); |
| 33 | goto OUT; |
| 34 |  |
| 35 | printf("after lockfilein"); |
| 36 | sleep(150); //休眠150秒，用于模拟访问碰撞 |
| 37 |  |
| 38 | OUT: |
| 39 | return(ret); |
| 40 | 1 |

第2章知其然—— 如何使用文件系统

编译并执行上述代码，然后开启另一个窗口再次执行上述代码。我们会发现该 程序被阻塞了。如果将第30行代码中的F\_SETLKW 修改为F\_SETLK, 此时程序 并不会被阻塞，而是会返回一个错误。

如果这时使用其他软件访问该文件，会出现什么结果呢?比如，使用cat 命令 读取文件数据。

结果是可以正常读取数据。这时大家可能会产生疑问。加锁不是实现对数据的 排他保护吗?怎么还可以读取数据呢?这是因为在Linux 中默认使用的是劝告锁。 如果进程没有对锁的状态进行询问而直接访问数据，则锁并不会保护数据。

为了对某个特定文件施行强制性上锁，需要使用强制锁。使田强制锁需要满足 加下几个条件

(1) 挂载文件系统时要指定mand选项 (mount-o mand)。

(2) 必须关闭文件的组成员执行位(chmod g-x file)。

(3) 必须打开文件的SGID位( chmod g+s file)。这 里SGID(Set Group ID)是 文件/目录的一种特殊权限，用于用户临时获得组权限。

完成上述操作后，如果在第1个窗口运行该程序，则在第2个窗口执行cat 命 令或vim 命令查看文件数据时会被阻塞。

**2.7** **文件系统的扩展属性**

在文件系统中，文件的基础属性比较有限，如文件的inode ID、创建时间、大 小和访问属性等。通用文件系统的用户往往有很多个性化的需求，因此文件系统通 过扩展属性允许用户自定义一些功能。

文件的扩展属性(xattrs)通过“键-值”对(Key/Value)方式提供了一种存储 附加信息的方式，扩展属性与文件或目录相关联。每个扩展属性可以通过唯一的键 来区分，键的内容必须是有效的UTF-8 编码，格式为namespace.attribute, 每个键 采用完全限定的形式，也就是键必须有一个确定的前缀(如user)。

在Linux 中 ，对扩展属性的管理可以通过setfattr命令和getfattr命令完成。前 者是设置文件的扩展属性，后者是获取文件的扩展属性。以设置文件的扩展属性为 例 ，setfattr 命令的语法格式如下：

setfattr -n user.sunnyzhang-v itworld123 fl.txt

执行上述命令后就为文件fi1.txt设置了扩展属性。需要注意的是，该扩展属性 的数据并不在文件内容中，而是在其他地方。



**文件系统技术内幕**

大数据时代海量数据存储之道

在图2-18中，第1个命令用于设置文件的扩展属性和获取文件的扩展属性。 在设置扩展属性时，-n 后面是扩展属性的名称，而-v 后面则是扩展属性的值。

通过getfattr 命令获取文件的扩展属性。在图2-18中，第2个命令可以获取该 文件的所有扩展属性，当然也可以配合选项来获取某些特定名称的扩展属性。

rootesunnyzhang-VirtualBox:/mnt/ext2#setfat n user,sunnyzhang -v itworld123 f1.txt

rootesunnyzhang-VirtualBox:/mnt/ext2# getfattr f1.txt

#file:f1.txt

user.sunnyzhang

图2-18 文件扩展属性的设置与获取

这两个命令的功能很丰富，大家可以自行阅读man 手册，此处不再赘述。除了 可以使用上述命令来对扩展属性进行管理，还可以通过API 来管理扩展属性，这更 适合程序员使用。使用API 设置和获取扩展属性的语法格式如下：

int setxattr(const char \*path,const char \*name,const void\*value,size\_t size,int flags); ssize\_t getxattr(const char \*path,const char \*name,void \*value,size t size);

这里需要说明的是，setxattr 中 的flags 参数用于指定setxattr 的行为。该参数有 两种可能的值，分别是XATTR\_CREATE 和 XATTR\_REPLACE 。如果参数的值是 XATTR\_CREATE, 在添加扩展属性时，遇到同名属性，则返回错误码EEXIST 。如 果是XATTR\_REPLACE, 则会用新值替换该属性的旧值。

扩展属性的具体应用要根据用户的用途而定。比如，在Ceph 分布式存储中， 使用本地文件系统的扩展属性来存储对象的一些属性信息。一些桌面应用使用扩展 属性存储一些附属信息，如文档的作者和描述信息等。

**2.8** **文件的零拷贝**

**2.8.1** **零拷贝的基本原理**

Linux 包含内核态和用户态。如果学习过内核的相关知识就会了解到内核态的 内存和用户态的内存是隔离的。当用户态的程序向文件写入数据时，需要将用户态

的数据拷贝到内核态的内存中：当用户态的程序读取数据时，需要将内核态的内容

拷贝到用户态的内存中。

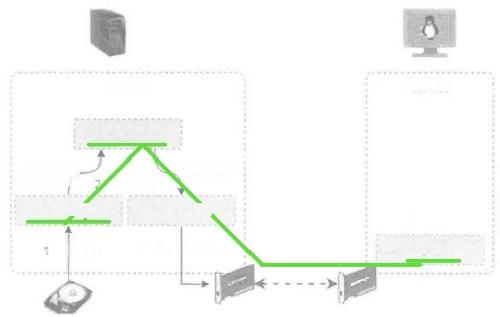
读取文件的过程分为两个步骤，首先从磁舟中读取数据并将其保存到内核内存

中，然后从内核内存中将数据拷贝到用户分配的data huf中。在写入数据时，先将 data buf中的 数据拷贝到内核内存中 ，然后写入磁盘。这种数据的拷贝过程其实是 非常消耗内存资源的，如果能减少内存拷贝过程，则一方面可以提高性能，另一方

第 2 章 知 其 然 — — 如 何 使 用 文 件 系 统

面可以减少延时。

不仅文件系统存在类似的问题，网络也存在类似的文件。如果想要将一个文件 通过网络发送到某个节点，则要经过两次用户态与内核态的内存拷贝。第一次将文 件系统缓存中的数据拷贝到用户态缓冲区，第二次将用户缓冲区的内容拷贝到传输 协议的缓冲区，如图2-19所示。除了用户态与内核态之间的内存拷贝，还有硬件 与系统内存之间的数据传输(通常为DMA 方式)。

服务端

客户端

用户态缓冲区

内存拷贝

内存薄贝 3

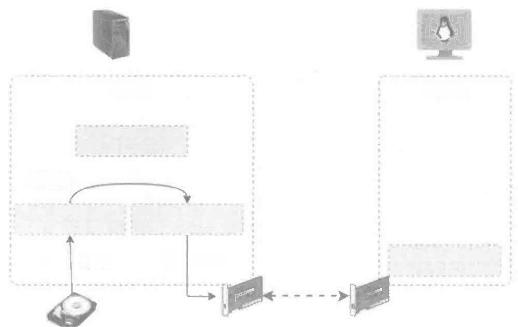
文件系红乐存 套接字缓在

DMA 4 DMA

传输协议

图2-19 数据访问的内存拷贝

我们观察一下会发现，对于单纯地将文件数据发送到网络的场景(如Web 服 务端发送照片),其实没必要经过用户态缓冲区转发，完全可以直接将文件系统缓 存的数据从内核中拷贝到传输协议的内核缓存中。这样本质上就减少了一次内核态 与用户态之间的内存拷贝，如图2-20所示。其实如果在内核实现两个模块的内存 拷贝不仅会减少内存拷贝的开销，而且也会减少内核态与用户态上下文切换的开销。

服务端

客户端

用户态缓冲区

内存拷贝 2

文件系统缓存 套接宇缓存

1 DMA 3 DMA 传输协议

图2-20 避免用户态拷贝的示意图



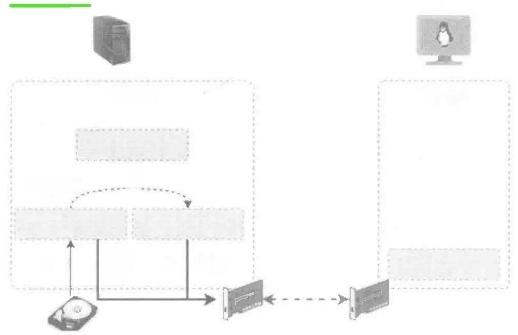
文件系统技术内幕

大数据时代海量数据存储之道

虽然使用上述方式消除了内核态与用户态之间的内存拷贝过程，但是在内核内 部还是有一次拷贝的。后来Linux 内核又做了进一步的优化，消除了内核内部的内 存拷贝。在该优化中，当执行2时，并不是进行 全内存拷贝， 而是将一个描述数据

位置的信息拷贝到套接字缓存中(图2-21中步骤2),通过传输协议发送数据时根

据描述信息利甲DMA机制直接将数据发送出去(图2-21中的粗线)

服务端

客户端

用户态缓冲区

描述信息 2.

文件系统缓存 套接字缓存

1 oMA 3 DMA

传输协议

图2-21 避免磁盘缓存与网络缓存拷贝的示意图

通过上面描述我们发现，其实所谓零拷贝技术并非没有任何内存拷贝，它主要 是消除数据的拷贝，描述数据的拷贝是不可缺少的。

**2.8.2** **零拷贝的系统API**

Linux通过sendfile(函数来使用零拷贝，sendfile()函数的语法格式如 :

ssize tsendfileint out fd,intin fd,offt\*offet size t coint);

在 sendfile() 函数中，int out\_fd 参数是输入数据的文件描述符，可以与前面的 文件对应；int in\_fd参数是输出数据的文件描述符，可以与前面的套接字对应。实 际上，在目前的Linux 中，输出数据的位置可以是网络，也可以是文件。

off t\*offset和size t count两个参数分别是偏移和大小，这两个参数的组合用 来确定从源文件的哪个位置读取多少数据。

虽 然sendfile() 函数可以直接在内核中实现文件内容的读取和数据的发送(写 入),但是我们无法对数据进行修改。这样就 限制了零拷贝技术的使用，毕竟很多 场景是需要对原始文件做一些处理的。

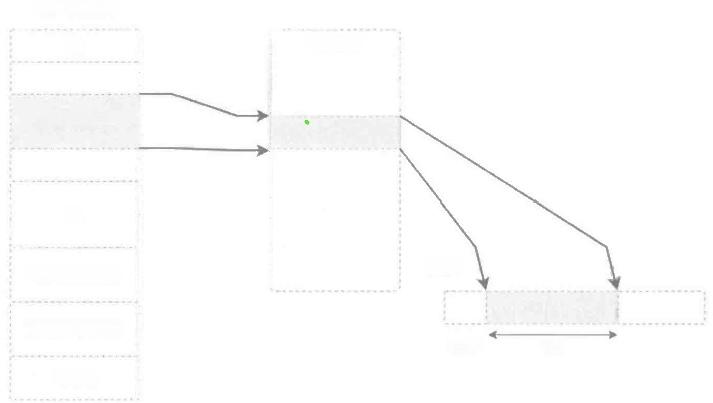
Linux 为了解决该问题实现了另一个API, 也就是mmap(函数。通过m map() 函数可以将一个文件中一定区域的数据直接映射到进程的虚拟地址空间，并返回内 存空间的地址，mmapO函数的语法格式如下：

第 2 章 知 其 然 — 如 何 使 用 文 件 系 统

void\*mmap(void addr size\_t length **int prot,int flags,**int fd,off\_t offset); int munmap(void\*addr,size\_t length);

在 mmap() 函数中，a ddr参数是期望返回的地址，其值可以为NULL,此时系 统会自动分配地址 fd 参数是对应的文件的文件描述符；offset 参数是对应的数据 在文件的位置；length 参数是映射的长度；另外，还有两个确定附加特性的参数， prot 和 flags。

构建映射后，文件内容与用户态内存之间的关系如图2-22所示。这样我们就 可以通过访问这个内存来访问文件数据，也就是当修改该内存的内容时，也就对文 件的内容进行了修改。通过使用这种方式也就不需要使用write()函数和read(函数， 避免内存拷贝和上下文切换的消耗。



进程地址空间

栈

文件映射区

页缓存

堆

文 件

已初始化数据段

offset len

代码段

未初始化数据段

物理内存

图2-22 文件内容与用户态内存之间的关系

第 章

**知其所以然——本地文件系统** **原理及核心技术**

我们知道文件系统最早是用来管理磁盘等存储设备的。为了区分，我们将直接

管理磁盘等存储设备的文件系统称为本地文件系统(Local File System)。本 地 文 件系统是最常用的文件系统，在不同的操作系统中往往有不同的文件系统，如Linux 中的Ext4和XFS 、Windows中的NTFS 、macOS中的HFS+和AFS等。

本地文件系统是最典型、最常用、最简单的文件系统。因此，这里先以本地文 件系统为例来进行介绍。由于Linux 中的文件系统是开源的，可以通过阅读代码实 现，因此这里主要以Linux 文件系统为例来进行介绍。

**3.1** **Linux文件系统整体架构简介**

文件系统是Linux 内核四大子系统(进程管理、内存管理、文件系统和网络栈) 之一，文件系统的地位可见不一般。为了便于理解具体的文件系统，下面先介绍一

下Linux文件系统的整体架构 如图3- 1所示，在具体文件系统(如Ext2、Ext4和 XFS等)与应用程序之间有一层抽象层，称为虚拟文件系统(Virtual File System,

VFS) 。

由图3-1可以看出，该架构的核心是VFS 。VFS 提供了一个文件系统框架，本 地文件系统可以基于VFS 实现，其主要做了如下几方面的工作。

(1)VFS 作为抽象层为应用层提供了统一的接口(read、write和chmod 等)。 ( 2 ) 在VFS 中实现了一些公共的功能，如inode 缓 存(Inode Cache)和页缓存

第3章知其所以然——本地文件系统原理及核心技术

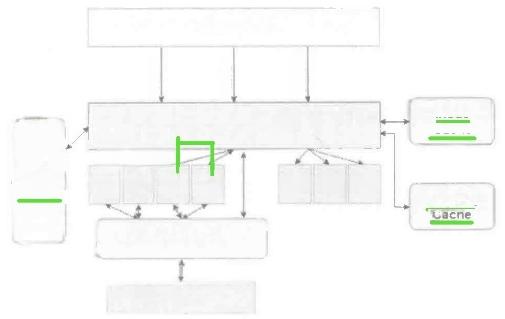
(Page Cache) 等。

(3)规范了具体文件系统应该实现的接口。

read

write

…



Application

Inode

Cache



Pache Ext2 Ext4 XFS NFs proc sysfsramfs

Directory

Buffer Cache

Device Drivers

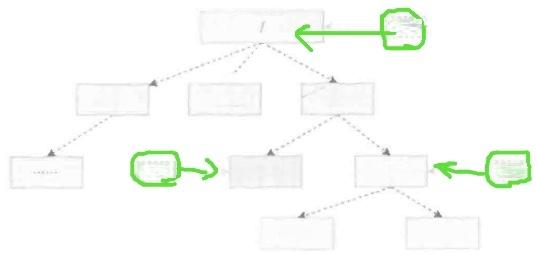
Virtual File System

图3-1 Linux文件系统的整体架构

基于上述设定，其他具体的文件系统只需要按照VFS 的约定实现相应的接口 及内部逻辑，并注册在系统中，就可以完成文件系统的功能了。

在Linux中，在格式化磁盘后需要通过mount命令将其挂载到系统目录树的某 个目录下面，这个目录被称为挂载点(mount point)。完成挂载后，我们就可以使用 基干该文件系统格式化的磁盘空间了。在Linux 中，挂载点几乎可以是任意目录， 但为了规范化，挂载点通常是mnt目录下的子目录。

图3-2所示为一个相对比较复杂的Linux 目录树。在该Linux 目录树中，根文 件系统为Ext3 文件系统，而在mnt 目录下又有ext4 test和 xfs test 两个子目录， 并且分别挂载了Ext4 文件系统和XFS 文件系统。

Ext3

bin

Ext4



home mnt

ext4\_test xfs\_test XFS

file1 file2

图3-2 Linux目录树

在Linux目录树中，多个文件系统的关系是由内核中的一些数据结构表示的。 在进行文件系统挂载时会建立文件系统之间的关系，并且注册具体文件系统的



**文件系统技术内幕**

大数据时代海量数据存储之道

API。当 用户态调用打开文件的API 时，会找到对应的文件系统API, 并关联到文 件相关的结构体(如file 和 inode 等)。

上面的描述比较抽象，大家可能还是有点不太明白。不要着急，在后面的章节 中，我们会结合代码更加详细地介绍VFS 及如何实现对多种文件系统的支持。

**3.1.1** **从VFS** **到具体文件系统**

Linux 中 的VFS 并不是一开始就有的，最早发布的Linux 版本并没有VFS。而 且，最初VFS 并非是基于Linux 发明的，它最早于1985年由Sun公司在其SunOS 2.0中开发，主要目的是适配其本地文件系统和NFS文件系统。

VFS 通过一套公共的API 和数据结构实现了对具体文件系统的抽象。 当用户 调用操作系统提供的文件系统API时，会通过软中断的方式调用内核VES实现的 函 数 。表3-1所示为部分文件系统API 与内核函数的对应关系。

**表3-1部分文件系统API与内核函数的对应关系**

|  |  |  |
| --- | --- | --- |
| **用户态API** | **内** **核** **函** **数** | **说** **明** |
| open | ksys\_open( | 打开文件 |
| close | .ksys\_close( | 关闭文件 |
| read | ksys\_read( | 读取数据 |
| write | ksys\_write) | 写入数据 |
| mount | do\_mount() | 挂载文件系统 |

由表3- 1可以看出，每个用户态API都有一个内核函数与之对应。当应用程序 调用文件系统的API 时会触发与之对应的内核函数。这里列举的只是文件系统API 中的一个比较小的子集，目的是说明API 与 VFS 的关系。如果大家想了解其他API 则自行阅读内核源代码，此处不再赘述。

为了让大家能够对VFS 与具体文件系统的关系有一个基本的认识，本节以Ext2

的写接口为例来展示一下从AP I VF S承数，再Ext2文件系统函数的调用关 系。如图3-3所示，API 函 数write(通过软中断触发内核ksys write()函数，该函数 经过若干处理后最终会通过函数指针(file->f op->wirte iter)的方式调用Ext2 文件

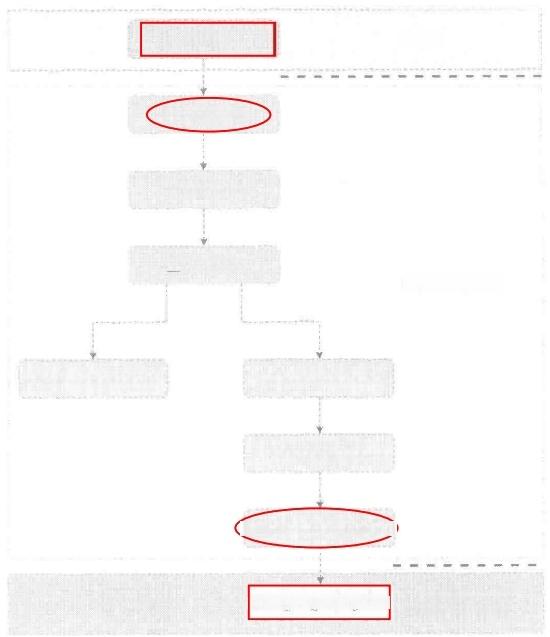
系统中的ext2\_file\_write\_iter() 函数。

看上去很简单，VFS 只要调用具体文件系统注册的函数指针即可。但是这里有 个问题没有解决，VFS 中的函数指针是什么时候被注册的呢?

Ext2文件系统中的函数指针是在打开文件时被初始化的(具体细节请参考 <3.1.2.2>节)。大家都知道，用户态的程序在打开一个文件时返回的是一个文件描述 符，在内核中表示文件的结构体file与之对应。在这个结构体中有几个比较重要的

第3章知其所以然 — — 本地文件系统原理及核心技术

成员，包括f inode 、f ops和f mapping等， 如图3-4所示。



write0 应用层

ksys\_write(

vfs\_write()

vfs\_write

虚拟文件系统层

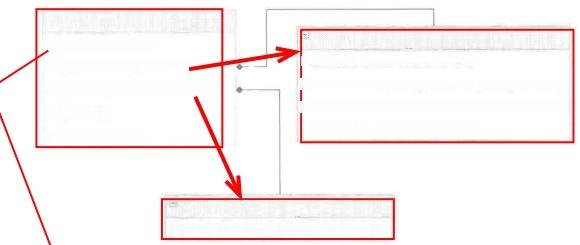
file->f\_op->write() new\_sync\_writeo

call\_write\_iter(

file->f\_op->write\_iter

ext2\_file\_write\_iterd Ext2文件系统

图3-3 Linux 文件系统写入数据函数调用流程



file

structinode

- open(struct inode"struct fleint

- readrstruct fie char\_user.size\_,lolt\_sslze -wnte(stuct fie char\_user'size\_t Joff\_12.ssize

-fush(stuct ie\*fi\_owner\_tid)in

address\_space

+a\_ops shuct address\_spac8\_operations

+t\_ops struct filo\_operabions'

+tmapping struct address\_space° +pos lof\_t

Lpath structpath

file\_operations

inode



图3-4 文件访问的核心数据结构

在图3- 4中，f inode是该文件对应的inode节点。f ops是具体文件系统(如 Ext2)文件操作的函数指针集合，它是在打开文件时被初始化的。VFS正是通过该 函数指针集合来实现对具体文件系统访问的。

至此，大家应该对VFS 与具体文件系统交互有了一个大致的了解。但是还有



文件系统技术内幕

大数据时代海量数据存储之道

很多细节有待层层剥开。比如，在打开文件时函数指针是如何被注册的，具体文件

系统是如何使用VFS页缓存的等，相关实现请参考下一节的内容。

**3.1.2** **关键处理流程举例**

为了更加清楚地理解VFS 与具体文件系统的关系，本节以Ext2文件系统的挂 载、打开文件与写入数据为例介绍一下用户态接口、VFS 和 Ext2 接口之间的调用 关系。通过上述流程的分析，我们会对VFS 的架构及关键数据结构和流程有比较 清晰的认识。基于这个基础，在学习其他流程时也就相对轻车熟路了。

[**3.1.2.1**](3.1.2.1) **文件系统的注册**

在Linux 中，具体文件系统通常是一个内核模块。在内核模块被加载(初始化) 时完成文件系统的注册。以Ext2文件系统为例，其初始化代码如代码3-1所示 ，调 用 register filesystem(函数将Ext2文件系统注册到系统中。这个函数其实就是Linux 中模块初始化的代码，任何内核模块都有一个类似的初始化函数。

代 码 3 - 1 E x t 2 文 件 系 统 初 始 化

**ext2/super.c**

|  |  |
| --- | --- |
| 1637 | static int init init ext2 fs(void) |
| 1638 | { |
| 1639 | int err; |
| 1640 |  |
| 1641 | err=init inodecacheO; //初始化inode缓存 |
| 1642 |  |
| 1643 | if(err) |
| 1644 | return err; |
| 1645 | erT=register filesystem(&ext2 fs type);//注册文件系统 |
| 1646 |  |
| 1647 | if(err) |
| 1648 | goto out; |
| 1649 | return 0; |
| 1650 | out: |
| 1651 | destroy\_inodecache(); |
| 1652 | return err; |
| 1653 |  |

register\_filesystemO函数调用了两个主要的函数， 一个是初始化inode缓存 ( 第 1641行):另一个是调用文件系统注册函数(第1645行 )。文件系统的注册很简单， 该流程就是将表示某种类型的文件系统的结构体(file system type)实例添加到一 个全局的链表中。这个结构体实例主要实现的函数是mount()。以 Ext2文件系统为 例，该结构体如代码3-2所示。

第3章知其所以然——本地文件系统原理及核心技术

**代** **码** **3** **-** **2** **E** **x** **t** **2** **文** **件** **系** **统** **的** **结** **构** **体**

**ext2/super.c**

|  |  |  |  |
| --- | --- | --- | --- |
| 1628 |  | static struct | file\_system\_type ext2\_fs\_type={ |
| 1629 |  | .owner | =THIS\_MODULE, |
| 1630 |  | .name | ="ext2", |
| 1631 |  | .mount | =ext2\_mount, |
| 1632 |  | kill\_sb | =kill\_block\_super, |
| 1633 |  | fs\_flags | =FS\_REQUIRES\_DEV, |
| 1634 | }; |  |  |

由于文件系统实例被添加到一个全局链表中，当用户态执行挂载命令时就可以

调用这里的mount()函数指针(对于Ext2文件系统，其具体实现为ext2 mount)。

mount(O函数的主要作用是从存储介质读取超级块信息，并创建该文件系统根目录

的dentry实例。这个dentry实例在后面挂载流程中将被用到。

大家在这里只需要知道文件系统注册了一个mountO函数即可，关于挂载的更 多细节会在后面章节再详细介绍。

[**3.1.2.2**](3.1.2.2) **打开文件的流程**

按理说应该先介绍一下文件系统的挂载流程，毕竟文件系统的挂载才是一个从 无到有的过程。但是直接介绍挂载流程，大家理解上有点困难，因此先介绍打开文 件的流程。

当打开一个文件时，调用的是open(函数，其语法格式如下： int fd=open(const char \*pathname,int flags,mode\_t mode);

open(函数传入一个字符串的路径参数(如/mnt/data/dirl/file.log), 然后返回一 个文件描述符。返回的文件描述符就是一个整数，后续用该整数表示一个文件。这 样就可以通过这个文件描述符进行访问，如读/写数据相关接口。

本节将介绍打开文件的流程，重点解释清楚如下几个问题。

(1) 如何通过一个字符串路径来打开一个文件?

(2) 为什么通过一个文件描述符就可以实现文件的访问?

要回答上述问题，需要更加深入地分析内核打开文件的流程。在内核中，打开 的文件是通过file 结构体表示的，而且该结构体与进程关联。因此，使用进程打开 的所有文件都保存在表示进程结构体(task struct)的files成员中。进程结构体 (task\_struct) 与 file 的关系比较复杂，如图3-5所示。其中，fdtable 是一个文件描

述符表，fd 成员以数组的形式存储file 结构体中的指针，而上文所述的文件描述符

其实就是该数组的索引。

在3.1.1节提到file 结构体中最重要的是函数指针。正是通过这些函数指针，

dentry 与 inode 的关系

映射关系：每个 dentry 指向一个特定的 inode 。dentry. 中保存了文件的名称和对应

的 inode 的指针，操作系统通过 dentry 可以快速获取到inode,从而访问文件的元数

文 件 系 统 技 术 内 幕 据和数据内容。

大 数 据 时 代 海 量 数 据 存 储 之 道 缓存机制：当访问一个文件时，操作系统首先检查dentry 缓存以确定该文件的 inode

是否已经被解析。如果缓存命中，系统可以直接使用已存储的inode 进行后续操作；如

果未命中，则需要在磁盘上查找并加载 inode。

当 读 / 写 该 文 件 时 就 可 以 访 问 具 体 文 件 系 统 的 函 数 。 接 下 来 介 绍 一 下 这 些 函 数 指 针 是 如 何 被 初 始 化 的 。

首 先 ， 从 整 体 上 看 一 下 打 开 文 件 的 流 程 ， 如 图 3 - 6 所 示 。 该 流 程 忽 略 了 缓 存 情 况 ， 只 展 示 了 核 心 流 程 。 该 流 程 有 两 个 主 要 分 支 ， 分 支 1 用 来 对 文 件 路 径 进 行 解 析 ， 并 逐 级 构 建 每 级 目 录 名 / 文 件 名 对 应 的inode 和 dentry; 分 支 2 则 进 行 文 件 打 开 必 要 的 设 置 工 作 ， 具 体 内 容 根 据 不 同 的 文 件 系 统 而 定 。

dentry(目录项)≤

定义：dentry 是文件系统中用于表示目录中的一个条目(如文件或子目录)的

通

open0

ksys\_openp

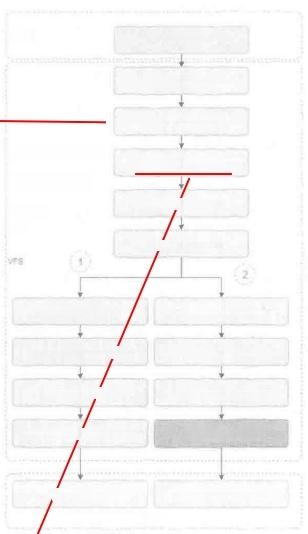
结构。每个 dentry 代表一个文件或目录的名称及其在文件系统中的位置。

功能：

dentry 主要用于快速查找文件路径。它们存储文件名与对应 inode 的映射关 系。

通过 dentry, 操作系统可以避免重复的磁盘访问，因为dentry 缓存了文件路

径的解析结果。

do\_sys\_openo

每个dentry 通常包含：

结构：

dentry与文件的关系

)当访用问户文通件过时路，径操名作(ho：me/user/file.txt

从根目录开始，逐级解析每个目录的dentry,

do\_sys\_openat20

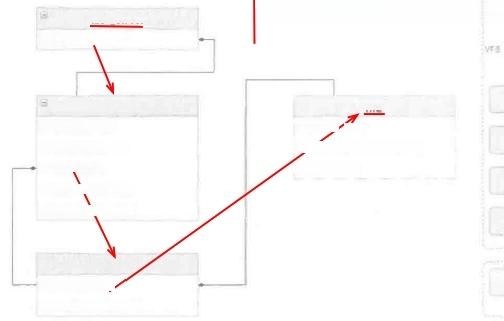
do\_ open0

n\_openat(

文件名(如 file.txt)

指向对应inode 的指针

交目录的dentry 指针(用于建立目录树结构)

通过找到的 dentry 获取对应的 inode。

直到找到目标文件的dentry。

访问 inode 中的元数据和文件内容。

task struct +hies fiesgthuet

files\_struct +count atomic\_

hink\_path\_wako

walk\_componeo

lookup s/wo

do\_openo

Vis\_openo

do\_denty\_open0

1sf\_op>open)

日 file

+Lnod bde

oable\_rcu

appng addres5\_space

-utyftlable

+t\_pampah

-fie\_ospiniock\_ +f0\_a \_reu

inode b>lookupo

回 fdtable

fd fie\_r>

04\_lookupo

ext4\_fie\_open0

max\_fds unsigned in

E

图 3 - 5 进 程 结 构 体 (task struct) 与 files 的 关 系 图 3 - 6 打 开 文 件 的 流 程

图 3 - 6 中 的 核 心 函 数 是do\_sys\_openat2(, 如 代 马 3 - 3 所 示 。 在do\_sys\_openat2O

函 数 中 ， 首 先 调 用get\_unused\_fd\_flags() 函 数 ( 第 / 1 1 7 7 行 ) 分 配 一 个 可 用 的 文 件 描

述 符 ； 然 后 调 用do flip\_open函 数 ( 第 1 1 7 9 行 / 打 开 文 件 ， 返 回file 指 针 ； 最 后 调

用 fd\_install() 函 数 ( 第 1 1 8 5 行 ) 将 file 指 针 子 联 到 进 程 结 构 体(task\_struct) 中 文

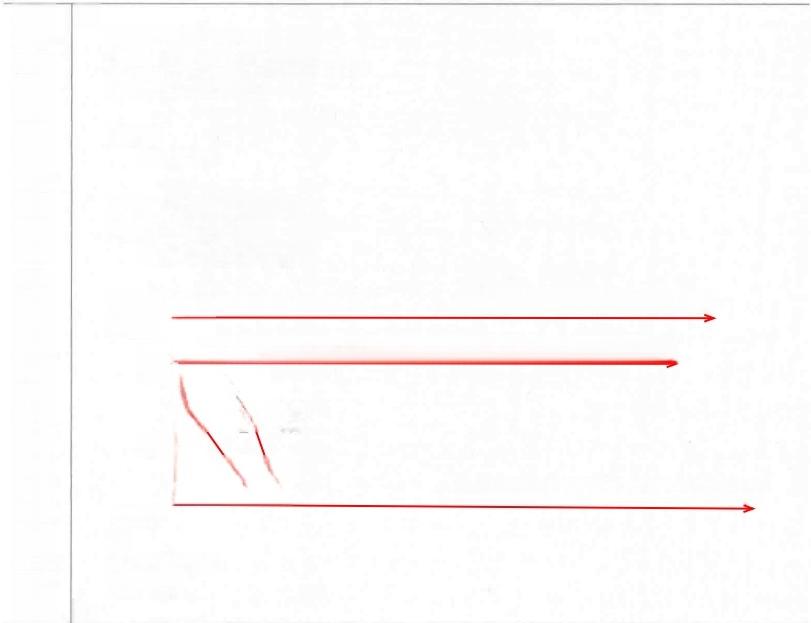
件 描 述 符 所 在 的 数 据 项 。

完 成 上 述 关 联 操 作 后 ， 后 续 对 文 件 进 行 读 / 写 等 操 作 ， 就 可 以 通 过 文 件 描 述 符 找 到 对 应 的file 结 构 体 指 针 。

代码3-3 do\_ s ys\_open at2()函数 fs/open.c ksys\_open->do\_sys\_open->do\_sys\_openat2

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 1163  1164 | static long | do\_sys\_openat2(int struct open how | dfd,const \*how) | char user | \*filename, |

第3章知其所以然 — — 本地文件系统原理及核心技术



{

struct open\_flags op;

nt fd=build\_open\_flags(how,&op);

struct filename \*tmp;

if(fd)

returnfd;

tmp=getname(filename); if(IS\_ERR(tmp))

return PTR\_ERR(tmp);

fd=get\_unused\_fd\_flags(how->flags);

ind>=0){

ruct file\*f=do filp

if NS\_ERR( {

ut\_unuse fd(fd);

fd PTR\_ARR(f); else{

fsnotify open 0); fd installd,t)



putname(tmp); returm fd;



//获取 一 个可以使用的文件描述符

open(dfd,tmp,&op);//打 开 文 件 ，file 在 其 中 分 配

1165

1166

1167

1168

1169

1170

1171

1172

1173

1174

1175

1176

1177

1178

1179

1180

1181

1182

1183

1184

1185

1186

1187

1188

1189

1190

/ / 将file 指针关联到进程的文件描述符中

更进一步，我们看一下d o filp open()函数是如何分配并初始化file结构体指针 的 。在图3- 6中，分支1通过link\_path\_walk()函数实现字符串路径的解析，该函数 的语法格式如下：

int link path walk(const char \*name,struct nameidata\*nd)

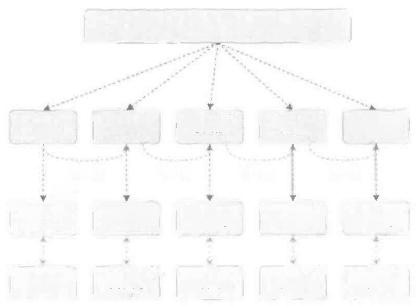
其 中 ，name参数是字符串表示的路径：nd参数类似一个迭代器，用于存储中 间结果和最终结果。

路径(Path) 字符串被"/"拆分为若干部分，每一部分称为一个组件 (Component), 如图3 - 7所示。在打开文件时，link\_path\_walk (函数正是通过逐个 组件遍历的方式最终打开文件的。

组件的遍历就是逐渐实例化该组件对应的inode 和 dentry 的过程。在没有任何 缓存的情况下，dentry 会先被初始化，在dentry 中包含文件/目录名字符串。在具体 某一级目录中，会调用该目录 inode 的 lookupO 函数查找该目录中的对应子项(子 目录或子文件),然后完成子项dentry 和 inode 的初始化。

**文件系统技术内幕**

**大数据时代海量数据存储之道**



Imnt/data/dir1/file.log

1 mnt/ data/ dirl/ file.log

下一设 下一级 下一级

dentry inode

dentry inode

dentry

inode

dentry

inode

dentry

inode

下一级 创建

图3-7 路径与组件

以 Ext4 文件系统中的lookupO 函数为例，通过其关键代码(见代码3-4)可以 看出共有3个关键步骤。

(1)从目录中查找对应子项：根据dentry 存储的名称字符串从目录中查找是否

有对应的项目。如果有该名称对应的文件/目录，则返回目录项数据结构de。同时， dentry会被插入到哈希表中。

(2)创建并初始化inode:根 据de 中保存的inode ID信息从磁盘查找inode 数 据，并初始化内存数据结构inode。该 inode 与具体文件系统相关。

(3)关联inode 与 dentry: 在 dentry 中有一个成员用于存储inode 信息，这一 步骤会建立两者之间的关系。

代码3 - 4 ext4\_ lookup()函数

fs/ext4/namei.c

|  |  |  |
| --- | --- | --- |
| 1682  1683  1684  1685  1686  1691  1707  1735  1736 | 上 | static struct dentry \*ext4\_lookup(struct inode \*dir,struct dentry\*dentry,unsigned intflags)  struct inode \*inode;  struct ext4\_di/entry\_2\*de; struct buffer/head \*bh;  bh=exi\_lookup\_entry(dir,dentry,&de); //从目录中查找对应项 … /制除部分代码  inodé=ext4\_iget(dir->i\_sb,ino,EXT4\_IGET\_NORMAL); //创建并初始化inode  ……/删除部分代码  retum d\_splice\_alias(inode,dentry); //关联inode与dentry |

在分支1完成路径解析，获得inode和dentry之后，分支2负责file指针的设 置。主要代码在do\_dentry\_open (函数中，将该函数中无关代码删除，只保留核心 代码，如代码3-5所示。

第3章知其所以然 — — 本地文件系统原理及核心技术

代码3-5 do dentry open()函数

**fs/open.c** **do** **open->vfs** **open->do** **dentry** **open**

|  |  |
| --- | --- |
| 768  769  770  771  772  773  774  775  776  777  778  779  809  876 | static int do\_dentry\_open(struct file \*f, struct inode\*inode,  int(\*open)(struct inode\*,struct file\*) {  static const struct file\_operations empty\_fops=0: int error;  path get(&f->f path); f->f\_inode=inode;  f->f\_mapping=inode->i\_mapping;  f->f\_wb\_err=filemap\_sample\_wb\_err(f->f\_mapping); f->f\_sb\_err=file\_sample\_sb\_err(f);  ……//删除部分代码  f>f op=fops get(inode->i fop);//这里是函数指针的初始化 |

通过上述代码可以看出，这里完成了file指针的主要初始化工作，特别是函数指 针的初始化(第809行)。通过上文介绍的打开文件的流程，我们对如何从一个路径字 符串打开一个具体的文件，最终生成 file指针和文件描述符的过程有了一定的了解。

上面介绍的打开文件的流程是缓存中没有期望内容的情况。如果在缓存中已经 有dentry 和 inode, 那么就不用调用lookupO 函数，而是可以直接从缓存中获得dentry 和inode, 因此打开文件的流程会简单一些。

接下来再看一看用户态的文件描述符为什么可以表示一个文件。其实前面已经 提及，在Linux 中，打开文件必须要与进程(线程)关联。也就是说， 一个打开的 文件必须隶属于某个进程。 在Linux内核中一个进程通过task struct结构体描述， 而打开的文件则用file结构体描述。

通过图3 - 5可知， file指针其实就是task struct结构体中的一个数组项 。而 用

户态的文件描述符其实就是数组的下标。这样通过文件描述符就可以很容易到找到

file结构体指针，然后通过其中的函数指针访问数据。

接下来看一下具体的代码。以写入数据流程为例，在内核中是ksys\_write() 函 数。如代码3-6所示，其中，第622行中的fdget\_pos() 函数根据fd 返 回fd 类 型 的 结构体，而该结构体中包含file结构体指针。后续的操作则是以该指针来表示这 个文件的。

**代码3-6** **ksys\_write()函数**

**fs/read** **write** **c**

|  |  |  |
| --- | --- | --- |
| 620 | **.**    **ssize\_tksys\_write(unsigned** **int** | **fd,const** **char** **user** **\*buf,size\_t** **count)** |
| 621 | **{** |  |
| 622 | **struct** **fdf=fdget** **pos(fd);** | **//从进程结构体中找到fd对应的file结构体指针** |

**文件系统技术内幕**

大数据时代海量数据存储之道

|  |  |  |
| --- | --- | --- |
| 623 |  | ssize\_t ret=-EBADF; |
| 624 |  |  |
| 625 |  | if(f.file){ |
| 626 |  | loff\_t pos,\*ppos=file\_ppos(f.file); |
| 627 |  | if(ppos){ |
| 628 |  | pos=\*ppos; |
| 629 |  | ppos=&pos; |
| 630 |  |  |
| 631 |  | ret=vfs write(f.file,buf,count,ppos);//ffile就是file结构体指针 |
| 632 |  | if(ret>=0&&ppos) |
| 633 |  | f.file->f pos=pos; |
| 634 |  | fdput pos(f); |
| 635 |  |  |
| 636 |  |  |
| 637 |  | return ret; |
| 638 | 上 |  |

完成本节阅读后，大家应该对字符串路径与dentry 和 inode 结构体的关系，以 及file结构体中指针的内容有所了解。基于这个基础，我们再学习挂载的过程就要 简单一些。

[**3.1.2.3**](3.1.2.3) **挂载文件系统**

挂载是用户态发起的命令，也就是大家都知道的mount命令。当执行该命令时

需要指定文件系统的类型(本文假设为XFS) 、文件系统数据的位置(也就是设备) 及希望挂载到的位置(挂载点)。通过这些关键信息，VFS就可以完成具体文件系 统的初始化，并将其关联到当前已经存在的文件系统中。假设操作系统使用的是 Ext4 文件系统，有一个磁盘(sdc) 并格式化为XFS 文件系统。我们期望将磁盘sdc 挂载到/mnt/xfs\_test 目录下，命令如下：

|  |
| --- |
| mount-t xfs/dev/sde/mnt/xfs\_test  执行上述命令后，XFS文件系统就被挂载到xfs test目录了。这样，当访问 xfs\_test目录时就是访问的XFS根目录，而不是原Ext4文件系统的目录了。 |
| 为了更加清楚地说明该问题，给出一个实例，如图3-8所示。假设有一个磁盘 |

并格式化为XFS 文件系统，在根目录有xfs\_filel 和 xfs\_file2 两个文件。此时，期 望以xfs\_test 作为挂载点对XFS 进行挂载。在挂载之前该目录中有file1 和 file2 两 个文件，上述文件是Ext4 中的数据(见图3-8上半部分)。如果执行上面挂载命令 后，则该目录的内容就变成XFS根目录中的内容，也就是xfs filel和xfs file2(见 图3-8下半部分)。

结合前文，我们知道文件/目录名是与dentry 相关联的，而dentry 又和inode 相

关联。因此，无论是访问文件还是目录中的内容，关键是找到对应的元数据并初始

第3章知其所以然——本地文件系统原理及核心技术

化为inode。其中，比较重要的是对inode中操作函数的初始化。

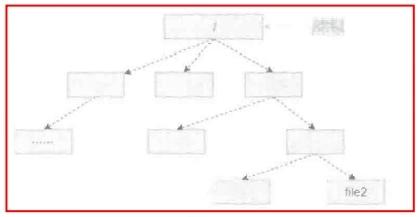
|  |
| --- |
| 由此我们可以猜测，对于挂载操作，应该是将dentry中的d\_inode成员由Ext4 的inode替换为XFS的inode。这样在打开文件流程中遍历路径时，获取的就是已 挂载文件系统的inode,访问的数据自然就是已挂载文件系统的数据。是否如猜想 的那样?我们接下来具体分析一下文件系统挂载的代码。 |

mount 命令本质上调用的是mount API,其函数原型如下：

int mount(const char \*source,const char \*target,const char \*filesystemtype, unsigned long mountflags,const void\*data);

从参数可以看出，主要包括设备路径、挂载点和文件系统类型等参数。

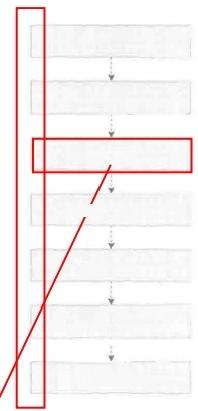
以文件系统API 为入口，挂载操作的核心流程如图3-9所示。由于Linux 的 挂 载命令支持的特性比较多，所以代码的各种分支流程很多。限于篇幅，本节以基本 挂载流程为例进行介绍，其他流程大同小异，大家可以自行阅读内核相关代码。

Ex14

mnt

bin

home

mount0

xts test

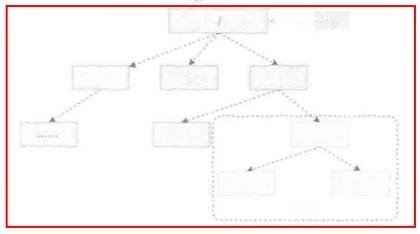
test

do\_mount0

fie1

挂载

do\_new\_mount0

Ex4

do\_fw mount fco

mnt

bin

home

do\_add\_mount0

graft\_reeo

test

xts test

xfs\_fiet xfs\_file2

attach\_recursive\_mnto

XFS

图3-8 文件系统挂载示意图 图9 文件系统挂载操作的核心流程

在上述核心流程中，涉及挂载的关键信息的初始化是在 do new mount()函 数 中完成的，包括获取待挂载的文件系统类型数握结构、创建文件 系统上下文数据结 构体和获取待挂载文件系统的根目录等，如仪码3-2 所 示 。

代码3-7 do new mount()函数

fs/namespace.c

|  |  |
| --- | --- |
| 2834  2835 | static int do\_new\_mount(struct path \*path,const char\*fstype,int sb\_flags, int mnt\_flags,const char\*name,void\*data) |

文件系统技术内幕

大数据时代海量数据存储之道

|  |  |
| --- | --- |
| 2836  2837  2838  2839  2840  2841  2842  2843  2844  2845  2846  2847  2848  2860  2874  2875  2876  2877  2878  2879  2880  2881 | struct file\_system\_type \*type;  struct fs\_context \*fc; //文件系统上下文结构体 const char \*subtype=NULL;  int err=0;  if(Ifstype)  return-EINVAL;  //获取待挂载的文件系统类型数据结构，它是在文件系统模块初始化时注册的    return-ENODEV;  ……/删除部分代码  //创建文件系统上下文数据结构体，该结构体用于存储挂载流程中需要的一些参数 fc=fs context for mount(type,sb flags)  ……/删除部分代码  if(lerr)  err=vfs get tree(fc): //获取待挂载文件系统的根目录，根目录会填充到fc中 if(lerr)  err=do new mount fc(fc,path,mnt flags);//完成后续挂载动作  put\_fs\_context(fc); return err; |
| 其 中 ，文件系统上下文数据结构体包含了挂载文件系统必需的信息，最十要的 | |

是在调用vfs get tree()函数时会调用具体文件系统中的mount()函数，然后将该函数

|  |
| --- |
| 返回的根目录对应的dentry填充到文件系统上下文数据结构体(以下简称为fc)中。 |
| 有了上述信息的准备后，接下来就调用 do new mount fc()函数来完成后续的 挂载动作。在该函数中会根据fc 中的信息创建一个vfsmount 的 实 例 ，vfsmount 结 |
| 构体定义如代码3-8所示。  **代码3-8** **vfsmount结构体定义** |

**fs/mount.h**

|  |  |  |
| --- | --- | --- |
| 70 | struct vfsmount{ |  |
| 71 | struct dentry \*mnt\_root; | //文件系统的根目录 |
| 72 | struct super\_block\*mnt\_sb; | //超级块数据 |
| 73 | int mnt\_flags; |  |
| 74 | \_randomize\_layout; |  |

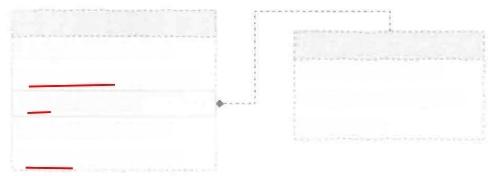
在vfsmount 结构体中有两个非常重要的成员： 一个是mnt\_root, 它是文件系统 根目录的dentry;另一个是mnt sb,它是文件系统的超级块数据。

另一个与vfsmount 关联的结构体是mount, 前者是后者的一个成员，两者关系 如图3-10所示。mount 结构体有很多成员，我们这里不再逐一介绍，比较重要的成

第3章知其所以然 — — 本地文件系统原理及核心技术

员包括mnt\_mountpoint ( 挂 载 点 目 录 项 ) 、mnt ( 挂 载 文 件 系 统 的 信 息 ) 和mnt\_mp

(挂载点)。



简 mount

mnt\_parent struct mount\* + mnt mountpoint struct dentry

+mnt struct vfsmount

+mnt\_chid.struct list\_head +mnt\_mpstruc.mountpoint

 vfsmount

mnt\_root struct dentry

+mnt\_sb.struct super\_block

+mnt\_flags int

图3-10 挂载相关数据结构

除了上面成员， mount结构体还有mnt parent和mnt child成员，通过上述成 员将mount构成一个树形结构。另外，在mount结构体中还有一个用于哈希表的成 员，用于将mount结构体添加到哈希表中。

了解了上述数据结构，接下来看一 下挂载流程中几个比较重要的函数。其中， 一个是d\_set\_mounted()函数，该函数的实现如代码3-9所示。d\_set\_mounted()函数

|  |
| --- |
| 最主要的语句是第1459行，用于为dentry增加DCACHE\_MOUNTED旗标。通过 |
| 该旗标标识该子目录是一个特殊的子目录，也就是挂载了文件系统的子目录，这个 |
| 旗标在打开文件时会用到。 |

代码3 - 9 d\_ set\_mounted()函数

fs/dcache.c do\_new\_mount\_fc->lock\_mount->get\_mountpoint->d\_set\_mounted

|  |  |
| --- | --- |
| 1441 | int d\_set\_mounted(struct dentry\*dentry) |
| 1442 | { |
| 1443 | struct dentry\*p; |
| 1444 | int ret=-ENOENT; |
| 1445 | write\_seqlock(&rename\_lock); |
| 1446 | for(p=dentry->d\_parent;!IS\_ROOT(p);p=p->d parent){ |
| 1447 |  |
| 1448 | spin\_lock(&p->d\_lock); |
| 1449 | if(unlikely(d\_unhashed(p))){ |
| 1450 | spin\_unlock(&p->d\_lock); |
| 1451 | gotoout; |
| 1452 | } |
| 1453 | spin\_unlock(&p->d\_lock); |
| 1454 |  |
| 1455 | spin\_lock(&dentry->d\_lock); |
| 1456 | if(Id\_unlinked(dentry)){ |
| 1457 | ret=-EBUSY; |
| 1458 | if(!d\_mountpoint(dentry)){ |
| 1459 | dentry->d\_flags|=DCACHE MOUNTED;//设置为挂载状态 |
| 1460 | ret=0; |
| 1461 |  |
| 1462 |  |

**文件系统技术内幕**

大数据时代海量数据存储之道

|  |  |
| --- | --- |
| 1463 | spin\_unlock(&dentry->d\_lock); |
| 1464 | out: |
| 1465 | write\_sequnlock(&rename\_lock); |
| 1466  1467 | returm ret; |
| 另 外两个比较重要的函数是mnt set mountpoint(和commit tree()(这 两个函 数通过do\_new\_mount\_fc->do\_add\_mount->grlft\_tree->attach\_recursive\_mnt 路径被 先后调用),通过这两个函数建立了父子mount 之间的关联，并且将待挂载的mount 添加到哈希表中。当完成上述函数的处理流程后，文件系统也就挂载成功了。  由 于dentry 在 mount 中，因此父子mount 关联之后，在文件系统层面也就建 立了挂载点dentry 和待挂载设备根目录dentry 之间的关联。也就是说，通过挂载点 中 的dentry, 我们就能找到挂载设备中的dentry。  返回打开文件遍历路径的流程，看一看对挂载点有什么特殊的处理。当每次遍 历路径中的 一 个组件时，最后都会调用 step\_into() 函数，该函数最终会调用 traverse mounts()函数进行挂载点的处理。 traverse\_mounts() 函数与挂载点相关 的代码如代码3-10所示。  **代码3-10** **traverse\_mounts()函数** | |
| **fs/namei.c** **step\_into->handle\_mounts->traverse\_mounts->\_traverse\_mounts** | |
| 1207  1208  1209  1210  1211  1212  1213  1214  1223  1224  1225  1226  1227  1228  1229  1230  1231  1232  1233  1234  1235  1236 | staticint traverse\_mounts(struct path \*path,unsigned flags,bool \*jumped, int \*count,unsigned lookup\_flags)  {  structvfsmount \*mnt=path->mnt; bool need\_mntput=false;  int ret=0;  while(flags&DCACHE\_MANAGED\_DENTRY){ …//省略部分代码  if(flags&DCACHE\_MOUNTED){ //确认是否有挂载的文件系统 struct vfsmount\*mounted=lookup\_mnt(path);//查找哈希表  if (mounted){ //命令空间  dput(path->dentry); if(need\_mntput)  mntput(path->mnt); path->mnt=mounted;  path->dentry=dget(mounted->mnt\_root);  //前面语句更新为该目录下已挂载文件系统根目录对应的目录项 flags=path->dentry->d\_flags;  need\_mntput=true; continue; |

第3章知其所以然 — — 本地文件系统原理及核心技术

 1237 /省略部分代码

通过代码3-10可以看出，第1224行代码判断该组件是否为挂载点。如果是挂 载点，则通过调用lookup mnt()函数来找到对应的vfsmount实例。由于该实例保存 着已挂载文件系统的根目录dentry,因此，可以使用该dentry更新path中的dentry, 而忽略原始的dentry 。

有 了dentry 之后，也就相当于找到了该文件系统根目录对应的inode 。从 而 使 用 该inode 的函数指针就可以访问已挂载文件系统的数据。

通过上述分析，我们对挂载流程如何实现将一个具体文件系统挂载到当前目录 树的一个子目录有了比较清晰的认识。可以看出上述流程主要是在VFS 中完成的， 而具体文件系统方面主要是调用了其实现的mount() 函数来创建 一 个根目录的 dentry。

[**3.1.2.4**](3.1.2.4) **读/写数据流程**

打开文件的知识，理解文件系统操作VFS 与具体文件系统的关系就简单多了。 由于文件的绝大部分操作都是通过在inode注册的函数指针完成的，而在打开文件 时，函数指针会被赋值给file 结构体中的成员f\_op 。因此，对于文件的读/写等访 问，经过VFS 后都可以找到对应的具体文件系统的函数指针进行具体文件系统的 操作。

**3.2** **本地文件系统的关键技术与特性**

文件系统应该是存储领域最复杂的领域之一，其原因在于文件系统需要实现的 特性太多，支持的场景太多。涉及文件系统相关的技术非常多，很难一一介绍，本 节主要介绍一下本地文件系统的关键技术与特性。这些几乎是所有文件系统都要考 虑的内容。

另外，本节更偏重于理论层面，更多实际代码层面的内容会在第4章进行详细 讲解。

**3.2.1 磁盘空间布局(Layout)**

文件系统的核心功能是实现对磁盘空间的管理。对磁盘空间的管理是指要知道 哪 些空间被使用了，哪些空间没有被使用。这样，在用户层需要使用磁盘空间时， 文件系统就可以从未使用的区域分配磁盘空间。

文件系统技术内幕

大数据时代海量数据存储之道

为了对磁盘空间进行管理，文件系统往往将磁盘划分为不同的功能区域。简单 来说，磁盘空间通常被划分为元数据区与数据区两个区域， 如图3- 11所示。其中， 数据区就是存储数据的地方，用户在文件中的数据都会存储在该区域；而元数据区 则是对数据区进行管理的地方。前文提到，文件系统需要知道磁盘的哪些区域已经 被分配出去了。所以，必须要有一个地方进行记录，这个地方就是元数据区。





图3-11 磁盘空间管理的基本原理

当然，实际文件系统的区域划分要复杂很多，这里主要是让大家容易理解一些。 接下来结合实例来介绍一 下关于文件系统磁盘空间布局与空间管理的相关内容。

3 **.2.1.1** **基** **于** **固** **定** **功** **能** **区** **的** **磁** **盘** **空** **间** **布** **局**

基于固定功能区的磁盘空间布局 是指将磁盘的空间按照 功 能划 分为不同的子 空间。每种子空间有具体的功能，以Linux 中 的ExtX 文件系统为例，其空间被划 分为数据区和元数据区，而元数据区又被划分为数据块位图、inode 位图和 inode 表 等区域，如图3- 12所示。

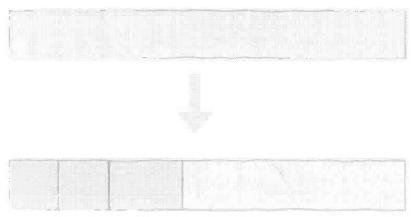
这里Ext X是Ext 、Ext2 、Ext3和Ext4文件系统的总称， 该系列文件系统是

Linux 原生的文件系统。

但在实际实现时，ExtX 并不是将整个磁盘划分为如图3- 12所示的功能区，而 是先将磁盘划分为等份(最后 一份的空间可能会小 一 些)的若干个区域，这个区域 被称为块组(Block Group)。磁盘空间的管理是以块组为单位进行管理的， 在所有

第3章知其所以然—— 本地文件系统原理及核心技术

块组中第1个块组(块组0)是最复杂的。

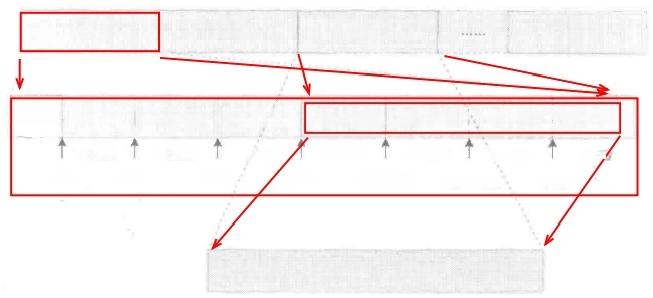
**磁盘空间(块组)**

inode 表 数据块

数据块 inode 位 图 位 图

图3-12 基于固定功能区的磁盘空间布局

图3-13所示为Ext2 文件系统的磁盘空间布局及块组0的细节(以4KB 逻辑 块大小为例，如果是其他逻辑块大小则略有差异)。块组0最前面空间是引导块， 这个是预留给操作系统使用的，接下来分别是超级块、块组描述符、预留GDT 块、 数据块位图、inode 位 图 、inode 表和数据块。除块组0及一些备份超级块的块组 外，其他块组并没有这么复杂，大多数块组只有数据块位图、inode位图、inode表 和数据块等关键的信息。



块组0 块组1 块组2 块组n

引导块 超级块 块组插述符 预留GDT块 数据块位图 inode位图 inode表 数据块

1 blocK 1 block many, 1 piock 1block many ma

1KB 4KB 8KB KB n+4)KB n+8)KB mKB

数据块位图 inode位图 inode表 数 据 块

图3-13| Ext2文件系统的磁盘空间布局及块组0的细节

超级块(SuperBlock):也就是不一般的块，这是相对文件系统的其他块来说的。 超级块存储了文件系统级别的信息，如文件系统的逻辑块大小、挂载点等，它是文 件系统的起点。

inode(index node,索引节点),所谓索引节点也就是索引数据的节点。 在Linux

文件系统中一个inode 对应着一个文件，它是文件数据的起点。在ExtX 文件系统 中 ，inode 是放在一个固定的区域的，通常在每个块组中都有若干个 inode, 称 为 inode表，类似数组。由 于inode 数量固定，且存储形式固定，因此可以根据inode

A

**文件系统技术内幕**

大数据时代海量数据存储之道

偏移给予编号，称为inode ID(或者inode number) 。 反 过来，也可以根据该inode ID 快 速 定 位inode 的具体位置，进而读到inode 的 内 容 。

位 图(Bitmap): 在 ExtX 文件系统中包含数据块位图和inode 位图，用来描述

对应资源的使用情况。位 图 通 过 1 个 位 (bit) 的数据来描述对应资源的使用情况，

0表示没有被使用，1表示已经被使用。

为了能够更加直观地理解ExtX 文件系统的布局情况，我们可以格式化一个文 件系统，然后通过dumpe2fs 命令来获取其描述信息。对于实验验证，我们不必在 一个磁盘上来进行文件系统的格式化。其实在一个空白文件即可进行文件系统的格 式化。比如，这里 格式化 一 个30MB 的 Ext2 文件系统，并且指定文件系统块大小 是 1KB, 当 使 用dumpe2fs 命令查看时可以看到 一共有4个块组，如图3- 14所示。

Grou-0:(BLocks-1-8 192)

Primary perblock at 1,Group descriptors at 2-2

Reserved GDT blocks at 3-121

Block bitmp at 122(+12I Inode bitmap at 123(+122)

Inode table at 124-363(+123)

7815 free blocks,1909 free inodes,2 directories Free blocks:378-8192

Free inodes:12-1920

Groum 1:(Blocks 8193-16384) Backup superblock at 8193,Group descriptors at 8194-8194

Reserved GDT blocks at 8195-8313 BLock bItmap at 8314(+121

Inode bitmap at 8315(+122)

Inode table at 8316-8555(+123)

7829 free blocks,1920 free inodes,0 directeries

Free blocks:8556-16384

Free inodes:1921-3840

Group 2:(B1ocks 16385-24576)

Block bitmap at 16385(+0)

Inode bitmap at 16386(+1)

Inode table at 16387-16626(+2)

7950 free blocks,1920 free inodes,0 directories Free blocks:16627-24576

Free inodes:3841-5760

Group 3:(BIocks 24577-30719)

Backup superblock at 24577.Group\_descriptors at 24578-24578

Reserved GDT blocks at 24579-24697

Block bitmap at 24698(+121)

Inode bitmap at 24699(+122)

Inode table at 24700-24939(+123)

5780 free blocks,1920 free inodes,0 directories

Free blocks:24940-30719

Free inodes:5761-7680

图3-14 Ext2 文件系统元数据信息(1KB 逻辑块)

可以看到该文件系统中的第1个块组包含超级块及GDT 保留信息，第2个块 组包含 一 个备份(Backup) 超 级 块 和GDT 保留信息，而第3个块组则不包含超级 块的信息。正如前文所述，如果块设备的存储空间充足，其实大部分块组是不包含 超级块信息的。

对比图3- 13中的磁盘空间布局和图3- 14格式化的实例，如超级块的位置、位 图的位置等，我们可以更加直观地了解磁盘空间布局的细节。通过这种方式可以加 深我们对文件系统相关原理的理解。

第3章知其所以然 — — 本地文件系统原理及核心技术

如果在格式化时选择文件系统块的大小是4KB, 则此时我们可以看到文件系 统中只有一个块组，如图3-15所示。为什么文件系统块大小不同，块组的数量会 有变化呢?

Group 0:(Blocks 0-7679)

Primary superblock at 0,Group descriptors at 1-1

Reserved GDT blocks at 2-2

B1ock bitmap at 3(+3) Inodebitmap at 4(+4)

feblck,-5ree inodes,2 directories

Free blocks:251-7679

Free inodes:12-7680

图3-15 Ext2 文件系统元数据信息(4KB逻辑块)

原因很简单，因为ExtX 文件系统通过一个逻辑块来存储数据块位图，如果将 文件系统格式化为1KB的块大小，那么对应的数据块位图可以管理8192(1024×8) 个数据块，也就是8MB(1024×1024×8)空间。因此30MB的存储空间被划分为4 个块组。

而对于4KB大小逻辑块的文件系统，一个块可以管理32768(4×1024×8)个 数据块，也就是128MB(32768×4KB)。因此30MB的存储空间只需要划分为一 个块组。

不仅块组的大小受限于此，在ExtX文件系统中inode的数量也受限于此。表3-2 所示为官网给出的不同块大小情况下相关数据。由于上述限制，在使用时也就随之 会有限制。比如，对于1KB 大小逻辑块的文件系统，由于一个块组中最大只有8192 个 inode, 因此也就最多只能创建8192个文件，超过该规格则无法继续创建新的 文件。

**表3-2官网给出的不同块大小情况下相关数据**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 上限\块大小 | 1KB | 2KB | 4KB | 8KB |
| 文件系统块数 | 2,147,483,647个 | 2,147.483,647个 | 2,147,483,647个 | 2,147,483,647个 |
| 每块组块数 | 8,192个 | 16,384个 | 32,768个 | 65,536个 |
| 每块组inode数 | 8,192个 | 16,384个 | 32,768个 | 65,536个 |
| 每块组字节数 | 8MB | 32MB | 128MB | 512MB |
| 文件系统大小 | 2TB | 8TB | 16TB | 32TB |
| 单文件最大块数 | 16,843,020个 | 134,217,728个 | 1,074,791,436个 | 8,594,130,956个 |
| 文件大小 | 16GB | 256GB | 2TB | 2TB |

<3.2.1.2> **基于非固定功能区的磁盘空间布局**

基于功能分区的磁盘空间布局空间职能清晰，便于手动进行丢失数据的恢复。 但是由于元数据功能区大小固定，因此容易出现资源不足的情况。比如，在海量小

**文件系统技术内幕**

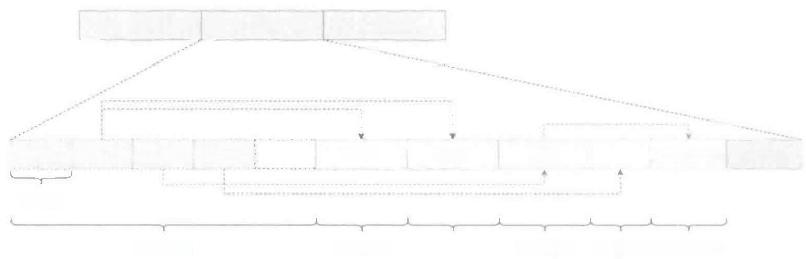
大数据时代海量数据存储之道

文件的应用场景下，有可能会出现磁盘剩余空间充足，但inode不够用的情况。

在磁盘空间管理中有一种非固定功能区的磁盘空间管理方法。这种方法也分为 元数据和数据，但是元数据和数据的区域并非固定的，而是随着文件系统对资源的 需求而动态分配的，比较典型的有XFS 和 NTFS 等。本节将以XFS 为例介绍一下 XFS 的磁盘空间布局情况及管理磁盘空间的方法。

XFS 文件系统先将磁盘划分为等份的区域，称为分配组(Allocate Group,简 称 AG) 。XFS 对每个分配组进行独立管理，这样可以避免在分配空间时产生碰撞，影 响性能。不同于ExtX文件系统，XFS文件系统的AG容量可以很大，最大可以达到 1TB 。

如图3-16所示，每个AG 包含的信息有超级块(xfs\_sb\_t)、剩余空间管理信息 (xfs\_agf\_t) 和 inode 管理信息(xfs\_agi\_t) 。 在 XFS 文件系统中，AG 中的磁盘空间 管理不同于ExtX 文件系统中的磁盘空间管理，它不是通过固定的位图区域来管理 磁盘空间的，而是通过B+树管理磁盘空间的。xfs agf t和 xfs agi t则是用来磁盘 空 间B+ 树和inode B+树的树根和统计信息等内容的数据结构。



AG

ABTB ABTC IABT xfs\_dinode\_t DATA

1个房区

1个逻辑块 1个逻辑块 1个逻辅块 4个逻墙块64个inode

xts sb t xfs\_agt\_xts\_agi\_t xts\_agn\_t

AG AG

图3-16 XFS文件系统磁盘空间布局

剩余空间的管理通过两个B+树来实现。其中，一个B+ 树通过块的编号来管理 剩余空间；另一个B+ 树通过剩余块的大小来管理。通过两个不同的B+ 树可以实现 对剩余空间的快速查找。

同样，在XFS 文件系统中inode 也是通过一个B+ 树来管理的，这一点与ExtX 文件系统不同。在XFS 文件系统中，将64个inode (默认大小是256字节)打包 为一个块 (chunk), 而该块作为B+ 树的一个叶子节点。

XFS 文件系统中的 inode 通 过B+ 管理，位置并不确定。因此XFS 文件系统无 法像ExtX 文件系统那个根据inode 的偏移来确定编号。XFS 文件系统通过另外一 种方式确定inode 编号，从而方便根据 inode 编号查找inode 节点中的数据。

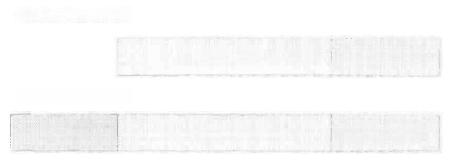
第3章知其所以然 — — 本地文件系统原理及核心技术

inode编号分为相对inode编号和绝对inode编号两种。相对inode编号是指针对 AG的编号，也就是AG 中的编号；绝对 inode 编号则是在整个文件系统中的编号。

相对inode编号格式分为两部分，高位部分是该inode 所在的逻辑块在AG 中 的偏移，而低位部分则是该inode在该块中的偏移。这样文件系统根据inode编 号

就可以在AG 中定位具体的inode。

绝对inode编号格式就比较好理解了，它是在高位增加了AG 的编号。这样在 文件系统级别根据AG 编号就可以定位AG, 然后根据AG 内块偏移定位具体的块， 进而可以知道具体的inode 信息。图3-17是两种模式 inode 编号格式示意图。



相对inode 编号格式

sb\_inopblog

绝对inode 编号格式

AG 号

sb\_inopblog

sb\_agblklog

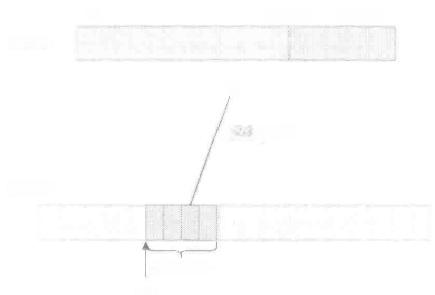
sb\_agblklog

MSB LSB

图3-17 XFS文件系统的inode编号格式示意图

为了容易理解，列举一个具体的实例。我们知道XFS 文件系统默认inode 大 小 是256KB,假设文件系统逻辑块大小是1KB,那么一个块可以包含4个inode。假 设存储inode 的块在AG 偏移为100的地方，而inode 在该逻辑块的第3个位置，

相对inode 编号示意图如图3-18所示。

31

inodeiD 100

磁盘空间

3 2



对应的inode ID

0

2

inode道辑块

100

图3-18 相对inode编号示意图

根据上述信息可以得到，该inode的 ID 为100<<3+2,也就是802。 绝对inode 编号相对于相对 inode 编号只是在高位增加了一个AG 编号。

**文件系统技术内幕**

**大数据时代海量数据存储之道**

[**3.2.1.3**](3.2.1.3) **基于数据追加的磁盘空间布局**

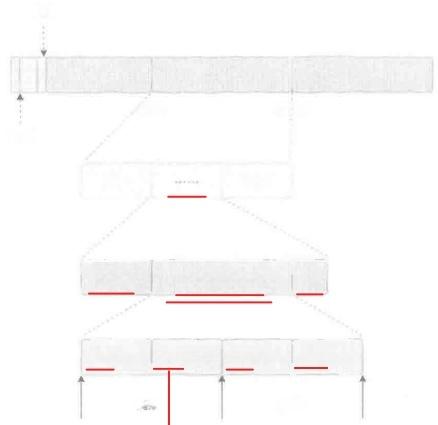
前文介绍的磁盘空间布局方式对于数据的变化都是原地修改的，也就是对于已 经分配的逻辑块，当对应的文件数据改动时都是在该逻辑块进行修改的。在文件随 机I/O比较多的情况下，不太适合使用SSD设备，这主要由SSD设备的修改和擦

写特性所决定。

有一种基于数据追加的磁盘空间布局方式，也被称为基于日志(Log-structured) 的磁盘空间布局方式。这种磁盘空间布局方式对数据的变更并非在原地修改，而是 以追加写的方式写到后面的剩余空间。这样，所有的随机写都转化为顺序写，非常 适合用于SSD 设备。

Linux 也有基于日志的文件系统实现，这就是NILFS2。为了便于磁盘空间的管 理和回收，NILFS2 文件系统将磁盘划分为若干个Segment,Segment 默认大小是 8MB。这 里 第 1 个Segment 的大小略有差异，由于前面引导扇区和超级块占用了 4KB的空间，因此第1个Segment 的大小是4KB～8MB。

如图3-19所示，每一个Segment 包含若干个日志(log)。每一个日志由摘要块 (Summary Blocks)、有效载荷块(Payload Blocks)和可选的超级根块(SR) 组成。 这里有效载荷块就是存储实际数据的单元。



4KB

+8MB +16MB

+1K8

logl logn

Summary Blocks

Data Blocks |B-uree Blocks|Data Blocks|B-tee Blocks

文件1 文件2

Payoad Blocks

SR

图3-19 NILFS2文件系统磁盘空间布局

如图3-19所示，有效载荷块以文件为单位进行组织，每个文件包含数据块和 B 树块。其中，B树块是元数据，实现对数据块的管理。但是实际情况可能要比图

第3章知其所以然 — — 本地文件系统原理及核心技术

示的格式复杂一些，因为随着文件的修改，数据块和B 树块会发生很大的变化。

在NILFS2文件系统中，文件分为若干类，分别是常规文件、目录文件、链接 文件和元数据文件。元数据文件是用于维护文件系统元数据的文件。目前，Linux 内核版本中的NILFS2 文件系统的元数据文件如下。

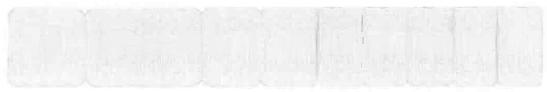
(1)inode 文 件(ifile): 用于存储inode。

(2)检查点文件(Checkpoint file,简 称cpfile): 存储检查点。

(3)段使用文件(Segment usage file,简 称sufile): 用于存储段(segment) 的 使用状态。

(4)数据地址转换文件(DAT): 进行虚拟块号与常规块号的映射。

图3-20所示为NILFS2文件系统中各种类型的文件在磁盘的布局情况，这里 的文件包括内部文件和常规文件。



摘要块 常规文件或目录 文件 …… ifile cpfie sufile DAT SR

图3-20 NILES2文件系统中各种类型的文件在磁盘的布局情况

通过图3-20可以看出，NILFS2 文件系统中的元数据都是在段的尾部，而数据 则是在段的开始位置。这个与实际使用是相关的，因为段数据的分配是从头到尾追 加的。这种布局模式便于数据和元数据的管理。

上面介绍的都是单磁盘文件系统。除了单磁盘文件系统，目前还有很多文件系 统可以管理多个磁盘。也就是一个文件系统可以构建在多个磁盘之上，并且实现数 据的冗余保护，如ZFS 和 Btrfs 等。

**3.2.2** **文件的数据管理**

本节主要介绍在文件系统中文件(目录)中的数据是如何被管理的。前文已经 介绍了磁盘空间的管理方式，知道磁盘会被划分为多个文件块，文件块的大小可以 是 1KB、2KB 或 4KB 等。但是一个文件可能会大于这些文件块的大小， 如一个电 影的大小约为1GB 。这就涉及文件数据管理的问题。

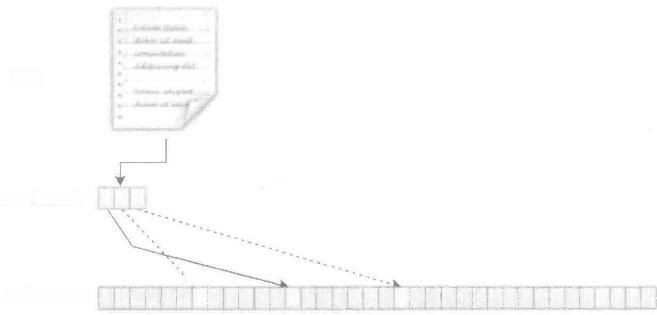
对于文件系统来说，无论文件是什么格式，存储的是什么内容，它都不关心。 文件就是一个线性空间，类似一个大数组。而且文件的空间被文件系统划分为与文 件系统块一样大小的若干个逻辑块。文件系统要做的事情就是将文件的逻辑块与磁 盘的物理块建立关系。这样当应用访问文件的数据时，文件系统可以找到具体的位

置 ， 进 行 相 应 的 操 作 。

文件系统技术内幕

大数据时代海量数据存储之道

文件数据的位置通过文件的元数据进行描述，这些元数据描述了文件逻辑地址 与磁盘物理地址的对应关系，如图3-21所示为文件逻辑块与磁盘数据的对应关系。

文件

文件逻辑空间



磁盘物理空间

图3-21 文件逻辑块与磁盘数据的对应关系

以 Linux 为例，文件的起点是inode, 文件数据的位置信息是存储在inode 中 的。这样就可以根据inode 存储的关于文件数据的位置信息找到具体的数据。我们 能想到的最直观的方式就是在inode 中存储每一个块的位置信息。比如，在逻辑块 大小为1KB 的文件系统中有一个3KB 的 文 件 。那么在inode中有一个数组，前3 项的值分别存储磁盘的地址信息，这样就可以根据数组的内容找到磁盘上存储的文 件数据。

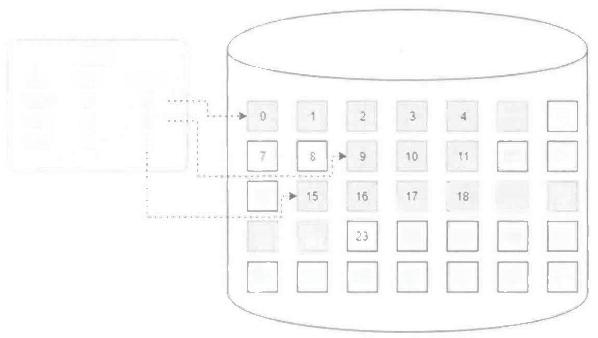
实际上，文件的数据管理方式大致如此，但又不完全是这样。不同的文件系统 采用了不同的管理方式，下面就介绍一下文件数据的管理方式。

[**3.2.2.1**](3.2.2.1) **基于连续区域的文件数据管理**

基于连续区域的文件数据管理方式是一次性为文件分配其所需要的空间，且空 间在磁盘上是连续的。由于文件数据在磁盘上是连续存储的，因此只要知道文件的 起始位置所对应的磁盘位置和文件的长度就可以知道文件数据在磁盘上是如何存 储的。

举例说明，如图3-22所示，假设某个目录有3个文件，分别是test1 、test2 和 test3, 其中，每个文件数据在磁盘的位置及长度如图3-22所示(左侧)。每个文件 的数据如图3-22所示(箭头的指向及深色方块处)。

第3章知其所以然—— 本地文件系统原理及核心技术



**目** **录**

length

6

3

8

13

14

21

28

file test1

test2 test3

start

0

9

15

29 30

27

20

26

25

22

24

31

32

33

34

19

12

6

5

图3-22 基于连续区域的文件数据管理

假设要访问test2 文件，根据目录中记录的数据，可以知道文件起始位置对应 磁盘第9个逻辑块，因此根据该信息和文件内部偏移就可以很容易地计算出文件任 意偏移的数据在磁盘上的位置。

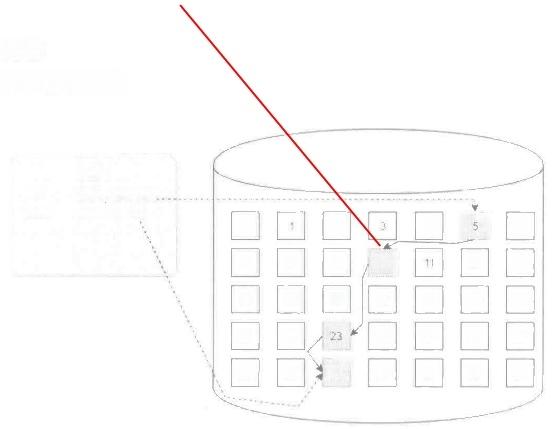
这种文件数据管理方式的最大缺点是不够灵活，特别是对文件进行追加写操作 非常困难。如果该文件后面没有剩余磁盘空间，那么需要先将该文件移动到新的位 置，然后才能追加写操作。如果整个磁盘的可用空间没有能够满足要求的空间，那 么会导致写入失败。

除了追加写操作不够灵活，该文件数据管理方式还有另一个缺点就是容易形成 碎片空间。由于文件需要占用连续的空间，因此很多小的可用空间就可能无法被使 用，从而降低磁盘空间利用率。

鉴于上述缺点，在磁盘等需要经常修改数据的存储介质的文件系统通常都不采 用基于连续区域的文件数据管理方式。该方式目前主要应用在光盘等存储介质的文 件系统中，如ISOFS。

[**3.2.2.2**](3.2.2.2)**基于链表的文件数据管理**

基于链表的文件数据管理方式将磁盘空间划分为大小相等的逻辑块。在 目录项 中包含文件名、数据的起始位置和终正位置。在每个数据块的后面用一个指针来指 向下 一 个数据块，如图3-23所示。

文件系统技术内幕

大数据时代海量数据存储之道

tile fest1

目录

start end

5 30

0

7

14

21

28

2

9

8

16

15

22

30

29

4

10

18

17

25

24

32

31

12

19

26

33

6

13

20

27

34

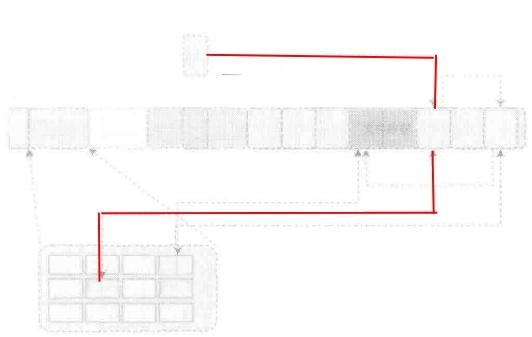


图3-23 基于链表的文件数据管理

这种方式可以有效地解决连续区域的碎片问题，但是对文件的随机读/写却无 能为力。这主要是因为在文件的元数据中没有足够的信息描述每块数据的位置。为 了实现随机读/写，一些文件系统在具体实现时做了一些调整。

以FAT12文件系统为例，该文件系统其实使用的就是链表方式，但是FAT12 又不完全使用的是链表方式。为了支持对文件的随机读/写，FAT12将文件的元数据 抽取出来，而不是存储在数据块的结尾部分。这样，文件的元数据可以一次性加载 到内存中，从而实现对随机访问的支持。

为了便于理解FAT12的原理，列举一个具体的实例，如图3-24所示。假设有 一 个 file1.txt 文件，我们根据目录文件项知道其起始的簇地址是0x05, 这个是 file1.txt文件第1个簇的位置，然后根据簇地址就能从文件分配表(FAT) 中找到对 应的表项，两者是一一对应的(图3-24中双向箭头处)。根据表项内容，我们可以 知道下一个簇的位置，以此类推，就可以找到该文件的所有数据。

file1 txt

add=0x05

FAT1 FAT2

目我京件项



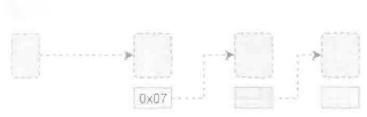
0x07

0x03

图3-24 FAT12文件数据管理示意图

第3章知其所以然 — — 本地文件系统原理及核心技术

如果简化一下这个结构，则整个关系就是一个单向链表的关系。我们可以将 FAT表项理解为next 指针，簇是data 数据。只不过FAT 表项和簇是通过地址偏移 建立了两者之间的关系的。图3-24可以简化为图3-25。

file1 xt

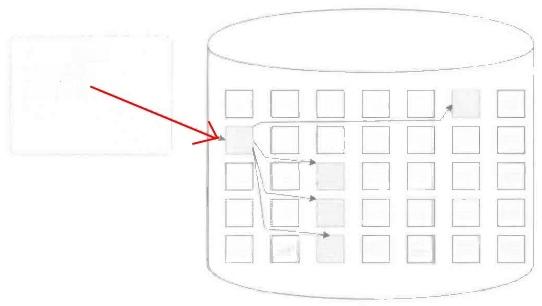
0x03 0xff

图3-25 FAT12文件数据管理方式简化图

通过对比可以看出，FAT12 本质上是基于链表数据管理方式的，但是由于文件 分配表本身比较小，可以一次性加载到内存中，因此也是可以满足随机访问需求的。 但是这种方式对随机访问的支持度还是不够的，毕竟内存中的链表访问也是相当低 效的，特别是针对链表项比较多的情况。

[**3.2.2.3**](3.2.2.3)**基于索引的文件数据管理**

索引方式的数据管理是指通过索引项来实现对文件内数据的管理。如图3-26 所 示 ，与文件名称对应的是索引块在磁盘的位置，索引块中存储的并非用户数据， 而是索引列表。当读/写数据时，根据文件名可以找到索引块的位置，然后根据索引 块中记录的索引项可以找到数据块的位置，并访问数据。



|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  | 目 录 |  |  |  |  |  |  |
| file | index |  |  |  |  |  |  |
| fest→ | 7 |  |  |  |  |  |  |
|  | 0  7  14  21  28 | 1  8  15  22  29 | 2  9  18  23  30 | 3  10  17  24  31 | 4  11  18  25  32 | 5  12  19  26  33 | 6  13  20  27  34 |

图3-26 基于索引的文件数据管理

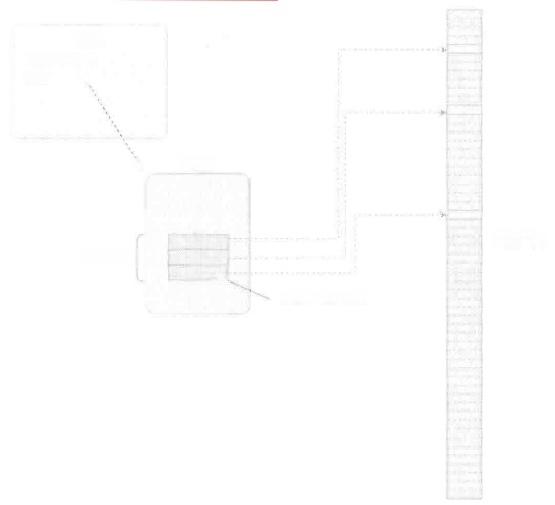
上文只是对索引方式进行了非常简单的说明。在实际工程实现时会有各种差 异，但本质上是一样的。接下来介绍两种常见的索引方式： 一种是基于间接块的文 件数据管理方式；另一种是基于Extent 的文件数据管理方式。

文件系统技术内幕

大数据时代海量数据存储之道

**1.基于间接块的文件数据管理**

在索引方式中，最为直观、简单的就是对文件的每个逻辑块都有一个对应的索 引项，并将索引项用一个数组进行管理，如图3-27所示。通过这种方式，文件的逻 辑地址与上述数组的索引就会有一一对应的关系。因此，当想要访问文件某个位置 的数据时，就可以根据该文件逻辑偏移计算出数组的索引值，然后根据数组的索引 值找到索引项，进而找到磁盘上的数据。

目 录

file inode

fest1 7

inode

引数组

0

1

2

4

11

23

索引项-磁世地址

磁 盘 空 间



图3-27 直接索引示意图

但是这种方式有个问题，就是对于大文件来说无法将索引数据一次性加载到内 存中，形成索引用的数组。假设以32位数来表示位置信息，文件块大小是1KB,

那么一个 10GB的文件需要4MB的数组数据。因此，虽然这种实现方式非常直观、 简单，但是并不实用。

在实际工程中通常会做一些变通。以Ext2文件系统为例，在实现索引时通过 多级索引的方式实现对数据的管理，最终形成一个索引树。在索引树中，只有叶子 节点存储的是用户数据，而中间节点存储的是索引数据。

Ext2 文件系统在实现这个索引树时做了很多变通，这种变通可以对很多场景 有优化的作用。对于Ext2文件系统，在inode中存储一个索引数组，该索引数组前 12项存储着文件数据的物理地址，称为直接索引。这样对于小文件来说，可以实现 次检索就能找到数据。

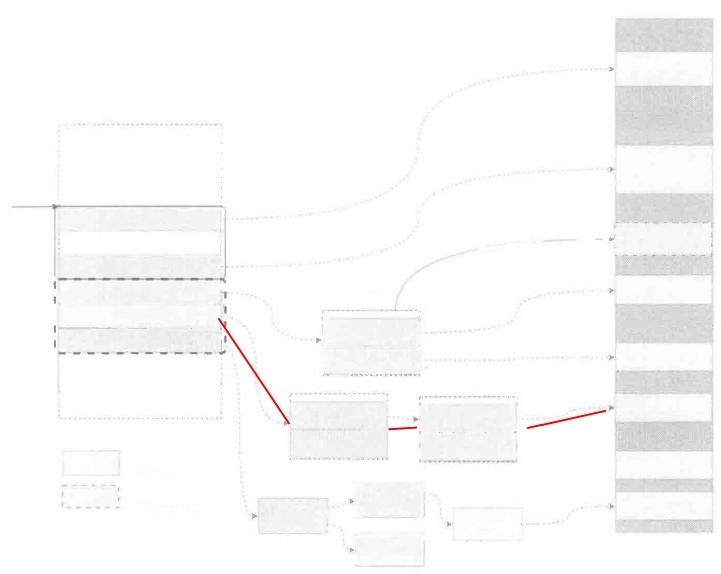
第3章知其所以然 — — 本地文件系统原理及核心技术

当文件太大，超出直接索引的范围时会通过间接索引来管理。间接索引通过3 棵独立的树来实现，分别是1级间接索引树、2级间接索引树和3级间接索引树。 这里级数的含义是该树中中间节点的层数。

为了能够更清楚地理解Ext2 文件系统是如何管理文件数据的，这里给出如 图3-28所示的示意图。在图3-28中，block0～blockl1 是直接索引，存储的是用户 数据的物理地址。而blockl2～block14 则是间接索引，存储的是索引块(或间接块， 简写为IB) 的物理地址，索引块中的数据并非用户数据，而是索引数据，是用于管理 用户数据的。

以block13形成的索引树为例，这里会形成一个2级间接索引树，可以将block13 理解为该索引树的树根。如果文件数据的逻辑地址在该树管理范围内，则需要经过 2级检索才能找到用户数据。

当然，我们也可以将整个索引数组理解为一个树根，不同的逻辑地址由不同的 子树进行管理。



disk blocks

ext2\_inode

Lbock

block0

直按家

block11

block12

B

biock13

block14

1B

又件数据

B,营理文件 数据的元数挥

IB

IB

IB

面接素引

IB

IB

图3-28 Ext2文件系统间接块数组组织形式

文件系统技术内幕

大数据时代海量数据存储之道

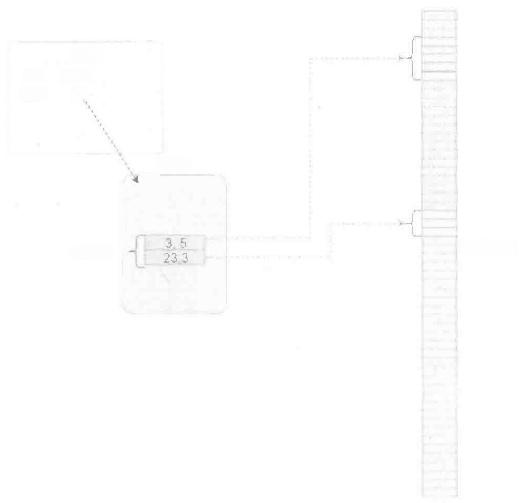
**2** **.** **基** **于Extent 的文件数据管理**

使用间接块可以很好地管理文件的数据，但是其最大的缺点在于元数据与数据 有一个固定的对应关系，也就是数据越多，需要的元数据越多。这种方式在某些场 景下其实并不划算，如视频文件场景。

简单地表述上述问题，就是每个数据块都要有一个元数据指针来记录其位置。 以32位指针，逻辑块大小为1KB 的文件系统为例，在1级间接块中每1KB 的 数 据都需要4字节的指针。而2级间接块除了每1KB 的数据需要4字节的指针，每 个2级间接块本身还需要指针来记录。

但是某些场景并不需要记录每个逻辑块的位置，最常见的就是视频文件或音频 文件。以视频文件为例，通常文件比较大，而且是一次存入，基本不会出现修改。 如果使用间接块的方式，则必须要有一定量的元数据。我们回忆一下前面介绍连续 区域的文件数据管理方式就会发现这种方式非常适合此场景。

但是连续区域的文件数据管理方式最大的缺点是对追加写操作处理不好，容易 形成存储空间的碎片化。于是，结合连续区域的文件数据管理方式和间接块的文件 数据管理方式的优点就有了现在这种方式，也就是Extent 文件数据管理的方式。在 Extent 文件数据管理方式中，每一个索引项记录的值不是一个数据块的地址，而是 数据块的起始地址和长度，如图3-29所示。

目录 bile inode test1 7

inode

激组 磁盘空间

图3-29 基于Extent的文件数据管理

第3章知其所以然——本地文件系统原理及核心技术

在图3 - 29中，实例文件有两块数据，前 一 块数据存储在磁盘偏移为3的位置， 大小是5个逻辑块；后 一 块数据存储在磁盘偏移为23的位置，大小是3个逻辑块。 对比间接块的文件数据管理方式可以看出，使用 Extent 的 文 件 数 据 管 理 方 式 可 以 有效减少元数据的相对数量。

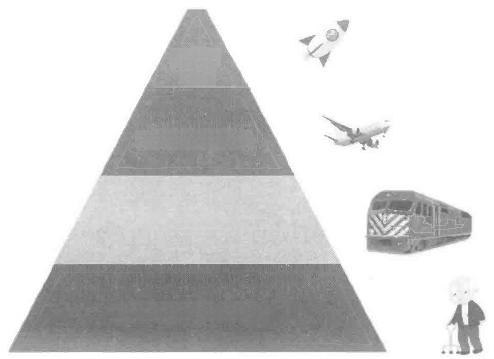
对 于Extent 的文件数据管理方式，如果出现追加写数据的场景，则文件系统只 需要分配 一 个新的Extent 。 因此该种方式并没有前文介绍的连续区域的文件数据管 理方式的缺点。

虽然本书实例描述数据位置的信息在内存中，但实际情况是并不会全部在内存 中 。 通 常Extent 是 通 过B+ 树 的 方 式 组 织 的 ，B+ 树 的 树 根 在inode 初 始 化 时 被 加 载 到内存中。而该树的中间节点则在磁盘上，会按需加载到内存中。由于B+ 树 是 一 个有序的多叉树，因此基于B+ 树实现从文件逻辑地址到磁盘物理地址的映射还是 比较快的。

**3.2.3** **缓存技术**

文 件 系 统 的 缓 存(Cache) 的作用主要用来解决磁盘访问速度慢的问题。缓存 技术是指在内存中存储文件系统的部分数据和元数据而提升文件系统性能的技术。 由于内存的访问延时是机械硬盘访问延时的十万分之 一 (见图3 - 30,以寄存器为基 准 单 位 1s), 因此采用缓存技术可以大幅提升文件系统的性能。文件系统缓存从读 和写两个角度来解决问题，并且应用在多个领域。





奇存器 (1s)

**CPU缓存(10s)**

内存(1005)

**磁盘(10,000,000s)**

图3-30 存储性能金字塔

**文件系统技术内幕**

大数据时代海量数据存储之道

文件系统缓存的原理主要还是基于数据访问的时间局部性和空间局部性特性。 时间局部性和空间局部性是应用访问数据非常常见的特性。所谓时间局部性就是如 果一块数据之前被访问过，那么最近很有可能会被再次访问。具体的实例就是文本 编辑器，在写代码或写文档的过程中，通常会对一个某一个区域进行不断的修改。 空间局部性则是指在访问某一个区域之后，通常会访问临近的区域。比如，视频文 件通常是连续播放的，当前访问某一个区域后，紧接着就是访问后面区域的内容。

以 Linux 文件系统为例，在文件系统初始化时会创建一个非常大的用于管理 inode的哈希表。哈希表的大小与系统内存的大小相关，对于2GB 左右的内存，哈 希表的大小有百万个，对于服务器等大内存的计算机，该哈希表的大小可达千万个 甚至上亿个。因此，当打开文件之后，文件对应的inode 就会缓存在该哈希表中。 这样，当再次访问该文件时就不需要从磁盘读取 inode 的数据，而是直接从内存读 取inode 的数据，其访问性能得到大幅提升。

还有一个应用是对用户数据的缓存，这里包含读缓存和写缓存。划分为读/写缓 存主要是在读/写的不同路径实现的功能特性不同。读缓存更多是实现对磁盘数据 的预读，而写缓存则主要是对写入数据的延迟。虽然读/写缓存的特性有所差异，但 本质是减少对磁盘的访问。

[**3.2.3.1**](3.2.3.1) **缓存的替换算法**

由于内存的容量要比磁盘的容量小得多，因此文件系统的缓存自然也不会太 大，这样缓存只能存储文件系统数据的一个子集。当用户持续写入数据时就会面临缓 存不足的情况，此时就涉及如何将缓存数据刷写到磁盘，然后存储新数据的问题。

这里将缓存数据刷写到磁盘，并且存储新数据的过程称为缓存替换。缓存替换有 很多种算法，每种算法用于解决不同的问题。接下来介绍几种常见的缓存替换算法。

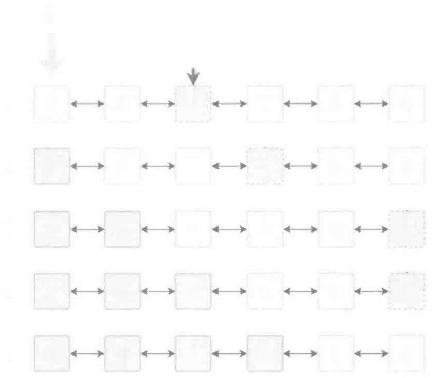
**1.LRU** **算法**

LRU(Least Recently Used)算法依据的是时间局部性原理，也就是如果一个数 据最近被使用过，那么接下来有很大的概率还会被使用。因此该算法会将最近没有 使用过的缓存释放。

LRU 算法通常使用一个链表来实现，刚被使用过的缓存会被插到表头的位置， 而经常没有被使用过的数据则慢慢被挤到链表的尾部。为了更加清晰地理解 LRU 算法的原理，结合图3-31进行说明。

第3章知其所以然——本地文件系统原理及核心技术



表头



2

3

4

5

1

3

4

6

5

2

1

3

4

6

3

2

1

3

4

4

业

4

2

1

3

5

5

5

2

1

6

6

业 6

业 5

2

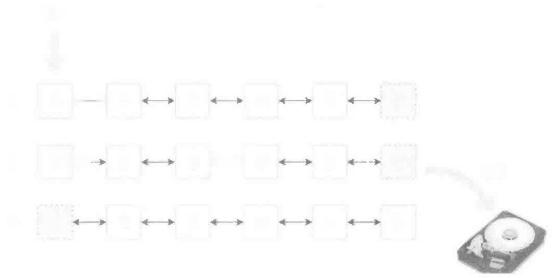
图3-31 LRU算法原理示意图

本实例以全命中进行介绍。假设缓存中有6个数据块，在第1行，方块中的数 字表示该数据块的编号。假设第一次访问(可以是读或写)的是3号数据块，由于 3号数据块被访问过，因此将其移动到链表头。

第二次访问的是4号数据块，按照相同的原则，该数据块也被移动到链表头。

以此类推，当经过4轮访问后，被访问过的数据块都被前移了，而没有被访问 过的数据块(如1号数据块和2号数据块)则被慢慢挤到了链表的后面。这在一定 程度上预示着这两个数据块在后面被访问的可能性也比较小。

如果是全命中也就不存在缓存被替换的情况。实际情况是会经常出现缓存空间 不足，而需要将其中的数据释放(视情况确定是否需要刷新到磁盘)来存储新的数 据。此时，LRU 算法就派上用场了，该算法将尾部的数据块拿来存储新数据，然后 放到链表头，如图3-32所示。如果这个数据块里面是脏数据则需要刷写到磁盘，

否则直接释放即可。 事头

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 1 | 1 ◀ → 2 | 3 | 4 5 6 |  |
| 2 | 1 2 | 3 | 4 5 6 | 监耳 |
| 3 | 6 1 | 2 | 3 4 5 |  |

图3-32 LRU 算法缓存替换流程示意图

**文件系统技术内幕**

大数据时代海量数据存储之道

LRU 算法原理和实现都比较简单，用途却非常广泛。但是LRU 算法有一个缺 点，就是当突然有大量连续数据写入时会替换所有的缓存块，从而导致之前统计的 缓存使用情况全部失效，这种现象被称为缓存污染。

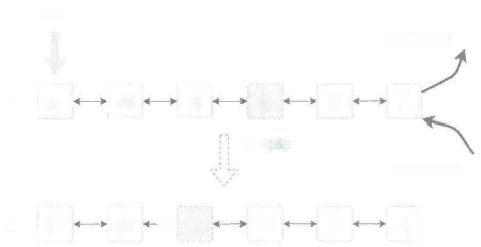
为了解决缓存污染问题，有很多改进的LRU 算法，比较常见的有LRU-K4、 2Q5 和LIRS6 等算法。

以LRU-K 算法为例，为了避免缓存污染问题，该算法将原来的LRU 链表由一 个拆分为两个。其中，一个链表用于存储临时的数据，可以理解为辅助缓存；另一 个链表采用LRU 算法进行维护。

**2.LFU** **算法**

LFU(Least Frequently Used)算法是根据数据被访问的频度来决定释放哪一个 缓存块的。访问频度最低的缓存块会被最先释放。

图3-33所示为LFU 算法缓存替换流程示意图。其中，第1行是原始状态，方 块中的数字表示该缓存块被访问的次数。新数据的加入和缓存块的淘汰都是从尾部 进行的。假设某一个数据(虚线框)被访问了4次，则其访问次数从12变成16, 因此需要移动到新的位置，也就是第2行。

表头

要淘大的数据

 29 18 15 12 8 1

4次访问后

新加入的数据

2 29 18 → 16 15 8 1

图3-33 LFU算法缓存替换流程示意图

本节以链表为例说明LFU 算法的原理以便于大家理解，但是在工程实现时是 绝对不会用链表来实现的。因为当数据块的访问次数变化时需要找到新的位置，链 表查找操作是非常耗时的。为了能够实现快速查找， 一般采用搜索树来实现。

LFU 算法也有其缺点，如果某个数据块在很久之前的某个时间段被高频访问， 而以后不再被访问，那么该数据会一直停留在缓存中。但是由于该数据块不会被访 问了，所以减少了缓存的有效容量。也就是说，LFU 算法没有考虑最近的情况。

本节主要介绍了LRU 和 LFU 两种非常基础的替换算法。除了上述替换算法，

第3章知其所以然——本地文件系统原理及核心技术

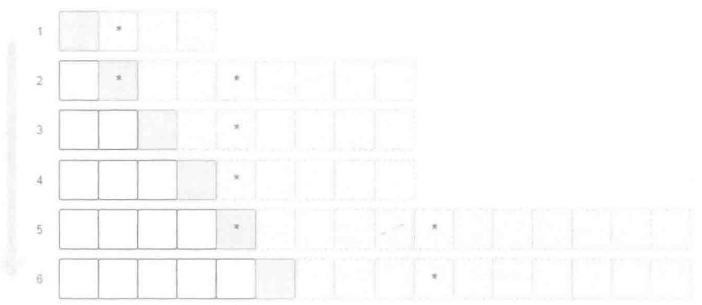
还有很多替换算法，大多以LRU 算 法 和LFU 算法的理论为基础，如2Q 、MQ 、 LRFUI7、TinyLFU 和 ARC8 等算法。限于篇幅，本节不再赘述，大家可以自行阅 读相关的书籍。

[**3.2.3.2**](3.2.3.2) **预读算法**

预读算法是针对读数据的一种缓存算法。预读算法通过识别I/O 模式方式来提 前将数据从磁盘读到缓存中。这样，应用读取数据时就可以直接从缓存读取数据， 从而极大地提高读数据的性能。

预读算法最为重要的是触发条件，也就是在什么情况下触发预读操作。通常有 两种情况会触发预读操作： 一种是当有多个地址连续地读请求时会触发预读操作； 另一种是当应用访问到有预读标记的缓存时会触发预读操作。这里，预读标记的缓 存是在预读操作完成时在缓存页做的标记，当应用读到有该标记的缓存时会触发下 一次的预读，从而省略对I/O 模式的识别。

为了更加清晰地解释预读算法，我们通过图3-34来介绍一下缓存预读操作流 程。当文件系统识别I/O 模式需要预读时，会多读出一部分内容(称为同步预读), 如时间点1(第1行)所示。同时，对于同步预读的数据，文件系统会在其中某个 缓存块上打上标记。这个标记是为了在缓存结束前能够尽早触发下一次的预读。



时间线

当前读到的位置 已经预读到爱存、但还未被应用读到的位置



有预读标记的位置

图3-34 缓存预读操作流程

**文件系统技术内幕**

大数据时代海量数据存储之道

在时间点2中，当应用继续读取数据时，由于读到了有标记的缓存块，因此会 同时触发下一次的预读。此时数据会被从磁盘上读取，缓存空间增加。

在时间点3和时间点4中，应用可以直接从缓存读取数据。由于没有读到有标 记的缓存块，因此也不会触发下一次的预读。在时间点5中，由于有预读标记，因 此又会触发预读的流程。

通过上述分析可以看出，由于预读特性将数据提前读到缓存中。应用可以直接 从缓存读取数据，而不用再访问磁盘，因此整个访问性能将得到大幅提升。

**3.2.4** **快照与克隆技术**

快照 (Snapshot) 和克隆 (Clone) 技术可以应用于整个文件系统或单个文件， 本节以文件为例进行介绍。快照技术可以实现文件的可读备份，而克隆技术则可以 实现文件的可写备份。针对文件，用的更多的是克隆技术。

文件的克隆技术用途非常广泛，最常见的是对一个虚拟机打快照。在给一个虚 拟机打快照时，其实就是对所有的虚拟磁盘做克隆。而对桌面版的虚拟机而言，虚 拟磁盘其实就是宿主机中的一个文件。因此虚拟磁盘的快照其实就是文件的克隆。

很多文件系统具有快照或克隆的功能，如Linux 中 的Btrfs可以实现文件系统 级的快照，OCFS2 可以实现单个文件的克隆(又被称为链接克隆)。Solaris 中 的ZFS (目前已经移植到Linux) 可以实现对快照和克隆的完整支持。并非所有的文件系统 都支持克隆功能，如ExtX 和 XFS 等文件系统是没有克隆功能的。

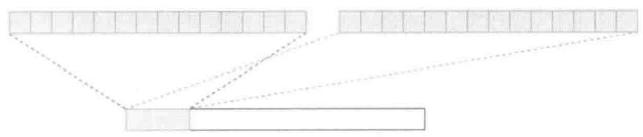
[**3.2.4.1**](3.2.4.1) **快照技术原理简析**

接下来介绍一下快照技术的基本原理。目前快照技术有两种实现方式： 一种是 写时拷贝(Copy-On-Write, 简 称 COW), 这种技术是对做过快照的原始文件写数 据时会将原始数据拷贝到新的地方。当然，并不是每次写数据都会拷贝，只是第一 次写数据时才会拷贝。通常会有一个位图记录已经拷贝过的数据，如果已经拷贝过 数据，则下次写数据时将不会再拷贝。

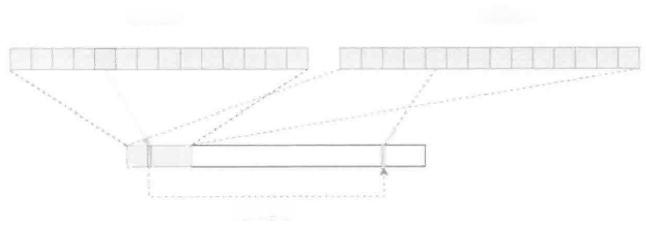
图3- 35所示为COW 原理示意图。当刚开始创建快照时原始文件和快照文件 指向相同的存储区域(图A)。当原始文件被修改时，如某个地方写入新的数据， 此时需要将该位置的原始数据拷贝到新的位置，并且更新快照文件中的地址信息 ( 图B) 。这样，虽然原始文件发生了变化，而快照文件的内容却没有发生变化。

第3章知其所以然—— 本地文件系统原理及核心技术

源始文件 快照文件



图A

原始文件 快照文件



拷贝数据

图B

图3-35 CoW 原理示意图

另一种实现方式为写时重定向(Redirect-On-Write, 简 称ROW), 这种实现方 式的基本原理是当原始文件写数据时并不在原始位置写入数据，而是分配一个新的 位置。在这种情况下更新文件逻辑地址与实际数据位置的对应关系即可。图3-36所 示为ROW 原理示意图。

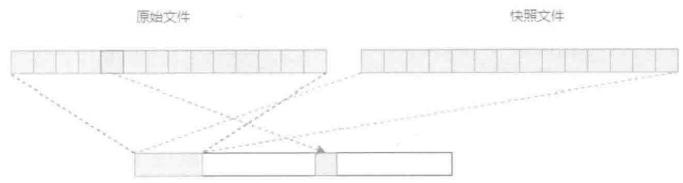


图3-36 ROW 原理示意图

[**3.2.4.2**](3.2.4.2) **克隆技术原理简析**

克隆技术的原理与快照技术的原理类似，其相同点在于其实现方式依然是 ROW 或 COw, 而差异点则主要表现在两个方面： 一个方面，克隆生成的克隆文件 是可以写的；另一个方面，克隆的数据最终会与原始文件的数据完全隔离。

对文件系统而言，用得最多的是ROW 方式。我们知道，每个inode 本身包含

文件系统技术内幕

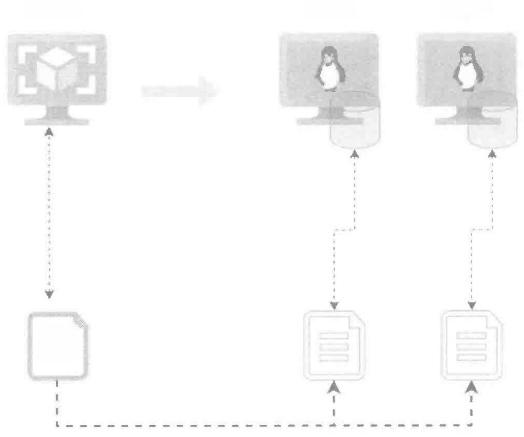
大数据时代海量数据存储之道

一个指针信息用于记录文件中每个逻辑块对应的物理块。当打快照时，我们只需要 拷贝该部分数据就可以表示一个快照文件。而当原始文件的数据发生变化时，只需 要将数据写入新的位置，并将原始文件中的地址信息进行变更即可。

[**3.2.4.3**](3.2.4.3) **应用实例**

文件的快照技术与克隆技术在云计算和虚拟化方面有着非常普遍的应用，使用 最多的是基于模板镜像的虚拟机快速发放和虚拟机的整体快照等相关特性。

以克隆技术在虚拟机快速发放中的应用为例，某公司有云模板镜像(如CentOS 或 Ubuntu 镜像),其实就是文件系统中的一个文件。而基于该模板镜像创建虚拟机 其实就是基于该镜像创建系统盘的过程，如图3-37所示。



模板镜像

三

虚拟机0

虚拟机1

克隆

图3-37 克隆技术在虚拟机发放中的应用

这里系统盘使用的其实就是模板镜像文件所生成的克隆文件。由于克隆可以瞬 间完成，为虚拟机提供与模板镜像一致的数据，因此虚拟机可以基于该镜像完成虚 拟机启动的过程。在后续，文件系统通过其内部的克隆模块完成原始文件到克隆文 件数据的拷贝，最终完成数据的隔离。

**3.2.5** **日志技术**

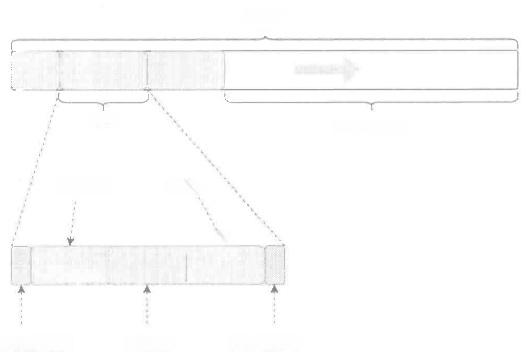
文件的一个写操作会涉及很多地方的修改。以Ext4 文件系统为例，当创建一

第3章知其所以然 — — 本地文件系统原理及核心技术

个文件时涉及向目录中添加一项，分配inode,更新inode位图等。如果在创建文件 的中间环节出现系统宕机或掉电，则会导致数据的不一致，甚至导致文件系统的不 可用。

在文件系统中，通过日志(Journal) 技术可以解决上述问题，该技术最早应用 在数据库中，后来被IBM 引 入JFS 文件系统中。目前，许多文件系统都具备日志 技术，如Ext4 、JFS 和 XFS 等。凡事没有绝对，并非所有的文件系统都采用日志技 术，如Btrfs。

日志技术的原理并不复杂，复杂的地方是工程实现。下面介绍一下文件系统的 日志是如何工作的。文件系统中的日志需要一块独立的空间，整个空间类似一个环 形缓冲区。当进行文件修改操作时，相关数据块会被打包成一组操作写入日志空间， 再更新实际数据。这里的一组操作被称为一个事务。一个完整的事务包括一个日志 起始标记、若干个inode 的块、若干个位图的块、若干个数据块和一个日志完成标 记，如图3-38所示。

日志区 人

I T T

事务

未使用的空间

inode的 块 数据块



日志起始标记 位图的块 日志完成际记

图3-38 文件系统中的日志原理示意图

由于实际数据的更新在日志之后，如果在数据更新过程中出现了系统崩溃，那 么通过日志可以重新进行更新。这样就能保证数据是我们所期望的数据。还有一种 异常场景是日志数据刷写过程中。由于此时日志完成标记还没有置位，而且实际数 据还没有更新，那么只需要放弃该条日志即可。

**3.2.6** **权限管理**

无论是Linux 还 是Windows, 都是多用户操作系统。由于多个用户的存在，就

文件系统技术内幕

大数据时代海量数据存储之道

必须实现用户之间资源访问的隔离。也就是说，用户A 的资源(主要是文件)不应 该让用户B 访问，或者需要授权后才可以访问。这种管理用户可访问资源的特性就 是 权 限 管 理 。

[**3.2.6.1**](3.2.6.1) **RWX 权** **限** **控** **制** **的** **原** **理**

Linux 最常见的访问控制方法被称为RWX 访问控制。当通过ls 命令获取文件 的详细信息时，其前面的rwx 字符串就是对文件权限的标示，而后面的两个 root 则 是其所属用户和组的信息。图3-39所示为RWX 权限实例。

total 32K

drwxr-xr 冈 2root

lroot

root

rot

23'root 1troot

dnx

N-rr-

-Nw-r--r- a'sumnyzhangiroot

4.0K Feb 488:55 4.0K Feb 4.08:54

7 Feb 408:54 test1

7 Feb:4.08:54 test2

7 Feb 408:54 test3

root

root

root

root

21root

2iroot Iroot

-nw-r-r

-w-r-r irxnwxrw

-W-r-r

w-r~r

7 Feb 468:54 test3pl

5 Feb 408:55 test3\_sl>test3 7 Feb 4.08:54 test4

lrroat

7 Feb 4.08:54 test5

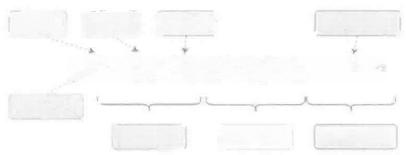
tituorld12kitworld123

所厘用户 所属组

权限信息

图3-39 RWX 权限实例

那么RWX 是什么意思呢?RWX 是 Read 、Write和 eXecute 的缩写，它描述了 用户和组对该文件的不同的访问权限。RWX 权限属性含义如图3-40所示。整个权 限描述分为4段，第1段用于描述该文件的类型，可以是常规文件(-)、目录(d) 、 块设备 (b) 、链接(1)和字符设备 (c) 等。



可读 可写 可执行 无存取权限

**-rWXrWX---**

文件类型

文件主权限 组用户权限 其他用户权限

图3-40 RWX 权限属性含义

后面3段是文件具体的权限描述信息，分别是文件主权限、组用户权限和其他 用户权限。通过上述3段的组合就可以实现比较复杂的权限控制。比如，允许某个 用户的文件可以被其他用户读取，但是不可以改写和执行等。

上述权限控制信息包含r、W 、x 、-共4种字符，具体含义如下。

(1)r 表示对于该用户可读。对于文件来说，r表示允许用户读取内容；对于目

录来说 ，r表示允许用户读取其中的文件 。

(2)w 表示对于该用户可写。对于文件来说，w 表示允许用户修改其内容；对

第3章知其所以然 — — 本地文件系统原理及核心技术

于目录来说，w 表示允许用户将信息写到目录中，即可以创建文件、删除文件、移 动文件等。

(3)x 表示对于该用户可执行。对于文件来说，x 表示允许用户执行该文件； 对于目录来说，x 表示允许用户进入目录搜索目录内容(能用该目录名称作为路径 名去访问它所包含的文件和子目录)。

(4)-表示对于该用户没有对应位的权限。具体禁用的功能请参考r 、w 和 x 的 含义理解。以读权限为例，如果用户没有该权限，则对应位置不是“r”, 而是“-”。

文件的访问权限是通过文件的RWX 属性和所属属性共同来控制的。RWX 属性描 述了不同的用户和组的访问权限，而所属属性则描述了该文件所属用户和组的信息。

以 test5 文件为例，其主用户是itworld123, 组 是itworld123 。而文件的RWX 属性如图3-41所示。这样，主用户itworld123是可以读/写的，而其他用户则只能 读，不可以写。当然，我们可以修改属性，让itworld123 组内的所有用户都可以写。

**-rW-r--r--1 itworld123 itworld1237 Feb 408:54 test5**

可读 可写 可执行 **无存取权限**

**-rWXrWX---**

文件类型

文件主权限 组用户权限 其他用户权限

人其地用户

用 白

图3-41 文件的RWX属性

RWX 权限控制的入口是打开文件，在打开文件流程中会调用inode permission() 函数，该函数判断进程对文件的访问权限。其判断的依据就是 RWX 属 性 (inode->i\_mode 中的内容)和进程的用户信息。

需要注意的是，目录中的文件对目录属性的继承性。也就是说，如果目录属于 用户 itworld123, 则其中的文件属于itworld123t1 。如果用户itworld123 对该文件没 有写权限，则在强制写数据的情况下该文件的所属用户会变为itworld123。

[**3.2.6.2**](3.2.6.2) **ACL权限控制的原理**

ACL 也是一种对资源进行访问控制的方式，第2章已经介绍过ACL 的场景和

**文件系统技术内幕**

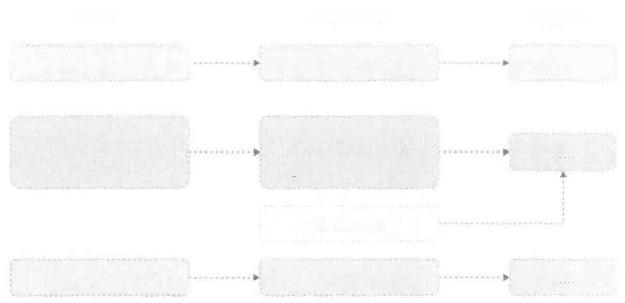
大数据时代海量数据存储之道

用法。我们可以手动设置文件或目录的ACL 以实现对文件或目录的访问控制。同 时ACL 还有一个特性是实现对父目录ACL 属性的继承。根据是否继承，ACL 分 为以下两类。

(1)access ACL:每一个对象(文件/目录)都可以关联一个ACL 来控制其访 问权限，这样的ACL 被称为access ACL。

(2)default ACL:目录关联的一种ACL 。当目录具备该属性时，在该目录中 创建的对象(文件或子目录)默认具有相同的ACL。

通过ACL 可以实现比较复杂的访问权限组合，权限的设置通过一个ACL 条目 实现。一个ACL 条目指定一个用户或一组用户对所关联对象的读、写、执行权限。 图3-42展示了ACL 条目的类型。



POSIX.le ACL

user::rwx

user:<name>:TWX group::wX

group:<name>:rwX

mask::rwx

other::rwX

文件类

**所有者(Owner)**

**其** **他(Other)**

组 (Group)

文件允许位

rwX

rwX

rwx

图3-42 ACL 条目的类型

例如，user::rwx 指定了文件的所有者对该文件的访问权限，user:<name>:rwx 指 定了某个特定的用户对该文件的访问权限，mask::rwx 表示该文件最大的允许权限， other::rwx 表示没有在规则列表中的用户所具备的权限。

ACL 在操作系统内部是通过文件的扩展属性实现的。当用户为文件添加一个 ACL 规则时，其实就是为该文件添加一个扩展属性。这样，当后续有某个用户访问 该文件时，文件系统根据规则就可以确定访问权限。 一个文件可以添加很多扩展属 性，因此ACL 的规则自然也可以有很多，这就保证了ACL 的灵活性。

需要注意的是，ACL 的数据与文件的普通扩展属性数据存储在相同的位置，只 不过通过特殊的标记进行了区分。这样，当普通用户查询扩展属性时ACL 的数据 是可以被文件系统屏蔽的。

本节不再赘述关于Linux 中 ACL 的实现细节，我们将在第4章介绍Ext2 具体 实现时再来分析ACL 在代码实现层面的内容。

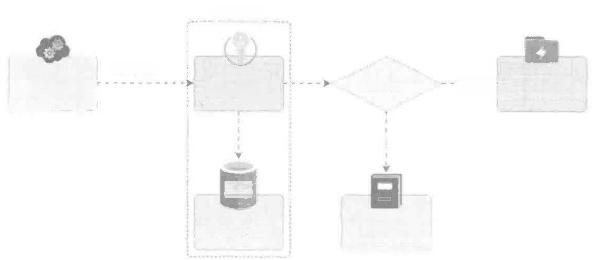
第3章知其所以然 — — 本地文件系统原理及核心技术

[**3.2.6.3**](3.2.6.3) **SELinux权限管理**

SELinux(Security-Enhanced Linux,安全增强式)是一个在内核中实现的强制 存取控制(MAC) 安全性机制。SELinux 与 RWX 、ACL 最大的区别是基于访问者 (应用程序)与资源的规则，而不是用户与资源的规则，因此其安全性更高。这里 基于应用程序与资源的规则是指规定了哪些应用程序可以访问哪些资源，而与运行 该应用程序的用户无关。

为什么说SELinux 这种方式更安全呢?前面基于用户的安全策略，如果某个应 用程序被黑客攻破，那么黑客可以基于运行该应用程序的用户启动其他应用程序实 现对该用户所属其他资源的访问。而SELinux 限定了应用程序可以访问的资源，即 使黑客攻破了该应用程序，也只能访问被限定的资源，而不会扩散到其他地方。

SELinux 的原理架构如图3-43所示，最左侧是访问者，也就是服务、进程或用 户等。最右侧是被访问者，也就是具体的资源，如文件、目录或套接字等。



Linux 内核

请求fmad

**SELinux**

**安全服务**

否

DB

SELinux AVC:

策略数据库 用止日志

**被访问者**

(文件，目录或套接字)

**访问者**

(服务，进程哦用户

是否合法 是

图3-43 SELinux的原理架构

当访问者访问被访问者(资源)时，需要调用内核的接口。以读取某个目录的 文件为例，需要读取接口read 。此时会经过SELinux 内核的判断逻辑，该判断逻辑 根据策略数据库的内容确定访问者是否有权访问被访问者，如果允许访问则放行， 否则拒绝该请求并记录审计日志。

通过图3-43可以看出，SELinux 的基本原理是比较简单的，关键是SELinux 策 略数据库的建立。由于应用繁多，资源也很多，因此规则数据库就比较繁杂。在实际 使用过程中经常会出现缺少规则而出现访问异常的情况，这时就需要手动添加规则。

**3.2.7** **配额管理**

在多用户环境中，不仅要防止用户对其他用户数据的非法访问，还要确保某些 用户所使用的存储空间不能太多。在这种情况下，就需要一种配额(Quota) 管

文件系统技术内幕

大数据时代海量数据存储之道

理技术。

配额管理是一种对使用空间进行限制的技术，其主要包括针对用户(或组)的 限制和针对目录的限制两种方式。针对用户(或组)的限制是指某一个用户(或组) 对该文件系统的空间使用不能超过设置的上限，如果超过上限则无法写入新的数 据。针对目录的限制是指该目录中的内容总量不能超过设置的上限。

在配额管理中，通常涉及3个基本概念，分别是软上限(Soft Limit)、硬上限 (Hard Limit) 和宽限期 (Grace Period)。这里简单介绍一下上述概念。

(1)软上限：指数据总量可以超过该上限。如果超过该上限则会有一个告警信息。

(2)硬上限：指数据总量不可以超过该上限，如果超过该上限则无法写入新的 数据。

(3)宽限期：宽限期通常是针对软上限而言的。如果设置了该值(如3天), 则在3天内允许数据量超过软上限，当超过3天后，无法写入新的数据。

配额管理是文件系统的一个特性，并非一种特殊的技术，在实现层面上也不需 要什么特殊的技术。以用户配额为例，当有新的写请求时，配额子系统会对该请求 进行分析。如果需要计入配额管理中，则进行配额上限的检查，在小于上限的情况 下会更新配额管理数据，否则将阻止新的数据写入，并发出告警信息。

Windows 中 的NTFS 文件系统实现了配额管理。在“Data(D:) 的配额设置” 对话框中可以进行配额的基本设置。单击“配额项”按钮，可以进行更加详细的设 置，如图3-44所示。

置 配眼

状态磁应配酸已被禁用

☑起用配管理间

□拒给网进画空给超过潮保制的用户(D) 为该卷上的新用户选择默认配额限制

◎限制理量使用()

O 将磁鱼空司限制为(U) 无阳

 天潭

选择该卷的配额记录选项

□用户超出配期限制时记录事件(9

□用户超过容告等级对记录事件(M

配期项(Q.

确定 取通 应用A)

图3-44 "Data(D:) 的配额设置"对话框

第3章知其所以然——本地文件系统原理及核心技术

Linux 也实现了对配额的支持。以XFS 为例，当挂载文件系统时使用配额选项 就可以实现对配额的支持。以启用用户配额为例，可以执行如下挂载命令：

mount-o quota xfs\_bak /tmp/xfs/

然后通过如下命令实现对 sunnyzhang 用户的配额的设置，这里将软上限和硬 上限分别设置为5MB 和 6MB。当然，这里只是一个实例，大家应该根据自己的需 要 设 置 。

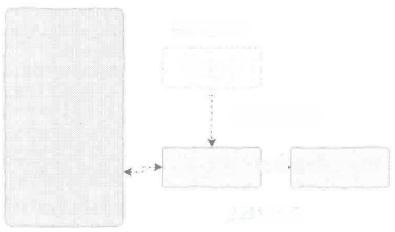
xfs\_quota-x-c 'limit-u bsoft=5mbhard-6m sunnyzhang'/tmp/xfs/

上面以数据量为例，实际工程实现包含的特性可能会更多。比如，针对文件数 量、子目录数量等都可以实现配额管理。

**3.2.8** **文件锁的原理**

第2章已经介绍过Linux 中文件锁的基本用法，并且给出了具体的实例。文件 锁的基本作用就是保证多个进程对某个文件并发访问时数据的一致性。如果没有文 件锁，就有可能出现多个进程同时访问文件相同位置数据的问题，从而导致文件数 据 的 不 一 致 性 。

文件锁的原理并不复杂，主要是要有一个地方记录目前文件的加锁情况，然后 当有新的加锁请求时可以基于记录的信息进行对比判断，如图3-45所示。新的锁 请求会与已有的锁信息进行逐一对比，确定是否存在锁冲突的情况。然后根据锁冲 突的情况来进一步确定后续的动作。



新的锁请求

file\_lock

A

inode 对比是否冲突

file\_lock 一 file\_lock

历史锁请求

图3- 45 文件锁的基本原理

如果没有任何冲突，则说明该进程是第1个加锁的，因此可以将锁信息加入链 表后返回；如果有冲突，则说明前面已经有进程对该段数据加锁。文件锁的后续动 作与调用函数的参数相关，如果在调用函数时参数指定需要休眠，则该进程进行休 眠 状 态 ， 否 则 返 回 一 个 错 误 码 指 示 存 在 锁 冲 突 ， 具 体 由 应 用 程 序 决 策 后 续 如 何 访 问 该文件 。

**文件系统技术内幕**

大数据时代海量数据存储之道

**3.2.9 扩展属性与ADS**

前文已经对Linux 文件系统的扩展属性进行了介绍。Linux 文件系统的扩展属 性以“键-值”对的形式在文件外存储。

Linux 中的扩展属性被划分为不同的空间，具体包含如下几种。

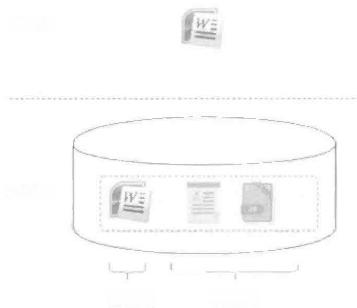
(1)system: 用于实现利用扩展属性的内核功能，如访问控制表ACL。

(2)security: 用于实现内核安全模块，如SELinux。

(3)trusted: 把受限制的信息存入用户空间。

(4)user: 用于为文件或目录添加一些附加信息，如文件的MIME、 文件编码 和字符集等信息。

Windows 中 的NTFS 文件系统并没有类似扩展属性的特性，但是有一个ADS (Alternate Data Streams) 的特性。在NTFS 文件系统中，将数据流(Data Stream) 分为主数据流和备数据流。其中，主数据流就是用户可以看到的文件内容；备数据 流则是文件内容之外的数据，通常用户是看不到的。比如，在某个Word 文档中， 我们创建了两个ADS, 分别用于存储字符串和脚本代码，如图3-46所示。

用户现再

存储视角

主数描流 备数据流

图3-46 NTFS文件系统中的ADS示意图

在图3-46中，我们看到的Word 文档本身是主数据流(又被称为匿名数据流), 而与之关联的字符串数据和代码则是备数据流(又被称为命名数据流)。如果通过 资源管理器浏览文件，那么只能看到Word 文档，却看不到备数据流。

备数据流被称为命名数据流的原因是在创建该数据流时需要给定一个名称，名 称命名规则与Windows 中文件名的命名规则一致。而文件主数据流则不需要特意 去命名，实际上它根本就没有名字，因此称为匿名数据流。

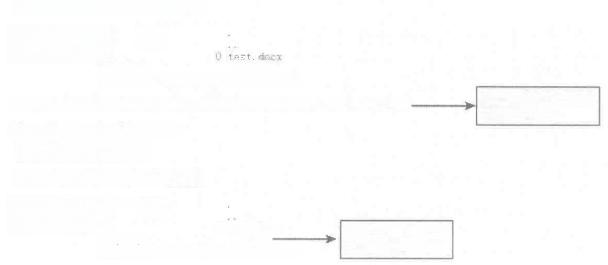
为了能够对ADS 有一个更加形象直观的认识，下面列举一个实例介绍ADS 。 以Windows 10为例，在cmd 命令行中可以进行ADS 的操作。这里有一个大小为0

第3章知其所以然 — — 本地文件系统原理及核心技术

字节的Word 文档，通过echo 命令可以向该文档写入一个名称为test\_stream 的 备 数据流。备数据流的内容为streamdata, 大小为10字节。

执 行ADS 操作的整个流程如图3-47所示，可以看到向该文档写入命名数据流 后，该文档的大小并没有发生变化。

D:test\rilesystonltest1>dir 驱动器D 中的卷是 Data 卷的序列号是A88E-D437

D:\test\filesystem\testi 的目录

|  |  |
| --- | --- |
| 2020/11/2117:20 2020/11/2117:20 2020/11/2117:20 | <DIR> DIR  1个文件 0宇节  2不督票418,952,196,096可用字节 |

D:\test\filesystem\test1 echostreandata >test.doex:test\_stream

Dtest\rilesystm\testl>dir

驱动器D 中的卷是Data 卷的序列号是A88E-D437

D:{test\filesystemltest1的目录

|  |  |
| --- | --- |
| 2020/11/2117:20 DIR>  2020/11/2117:20 <DIR> |  |
| 2020/11/2117:21 0.test.docx  1个文件 0学节一  2个自录418,.951.409.664可用字节 | 写入命名数据流后，该 文档大小依然是0字节 |

向文档写入命名数据流后

名称为test stream

图3-47 ADS设置操作演示

ADS 不仅可以存储字符串，它还可以存储任何类型、任意大小的数据。比如，可 以通过type 命令将本目录下一个名为ceph.jpg 图片文件存储到ADS 中，代码如下：

type ceph.jpg>test.docx:image\_stream

如果使用PowerShell, 则有另一套工具可以实现对ADS 的管理。上面创建的 两个ADS, 可 以 在PowerShell 中通过Get-Item 命令获取相关的描述信息，如图3-47 所示。

PS D:\test\filesysten\test1S Get-Item path test.docx=ztraa

PSPath :Micraaoft.PowerShell.Core\FileSystam::D:\taat\filesystem\test1\test.docx::$DATA

:Microspft.PowerShell.Core\FileSystem::D:\test\filesystem\test1 :tesf.docx::SDATA

:Micrasoft.PowerShel1.Core\FileSystem

PSIsContainer

FileName D:\test\filesysten\test1Xtest.docx

:$DATA

:0

PSPath :Microsoft.PowerShell.Core\FileSystem::D:\teat\fileaystem\test1\test.docx:image\_stream

PSParentPath :Microsoft.PowerShel1.Core\FileSystem::D:\test\filesystem\test1

PSChildName :test.docx:image\_stream

:Microsoft.PowerShell.Core\FileSysten

PSIsContainer False

FileNane :D:\test\filesystemlteat1\test.docx

Stream

PSPath Microsoft.PowerShel1.Core\FileSystem::D:\test\fileaystem\test1\test.docx:test\_strea

Microsoft.PowerShell.Care\Filesystem::D:\test\filesystea\test1

PSChildName test.docx:test\_strean

D

Microzoft.PowerShel1.Core\FileSysten

PSIsContainer False

:D:\test\fileayatemltest1Vtest.docx test stream

图3-48 Get-Item命令的使用

文件系统技术内幕

大数据时代海量数据存储之道

如何访问文件中的备数据流呢?通常来说，某些特殊的应用程序可以调用操作 系统的API 来访问。另外，可以通过Windows 一些程序访问，如前文我们创建的 图片格式的备数据流，可以通过如下命令来打开这个图片：

mspaint D:\test\filesystem\testI\test.docx:image\_stream

**3.2.10** **其他技术简介**

除了上面介绍的一些通用的技术，很多文件系统还有一些自己特色的技术。本 节再介绍一些比较常用的技术。

[**3.2.10.1**](3.2.10.1) **数据加密**

我们在使用计算机时应该有这样的经历，如忘记了操作系统密码，但是又想将 里面的数据导出。这时我们可以将硬盘拆卸下来，然后安装到另一台计算机上，通 过该计算机来访问这个硬盘的数据。

显然，对于数据安全来说，这是一个安全隐患。因为，任何人都可以将你的硬盘 拆卸下来，然后读取其中的数据。为了解决上述问题，文件系统的加密技术应运 而生。

文件系统的数据加密又被称为透明数据加密，因为文件系统的数据加密对用户 来说是感知不到的、透明的。当用户向文件系统写入数据时，文件系统的加密模块 会通过加密算法将加密后的数据写入磁盘。而当用户读取某个文件的数据时，文件 系统的加密模块会首先对该数据进行解密，然后将数据返回给应用。

可以看出，这里面的关键是磁盘上的数据是经过加密的。因此，经过加密的磁 盘，即使将该磁盘拆卸下来安装到其他计算机上，也是没有办法将数据读取出来的。

[**3.2.10.2**](3.2.10.2) **数据压缩**

文件系统的数据压缩技术是通过对文件系统内的数据块进行压缩来提升空间 使用率的。文件系统中的数据压缩对用户也是透明的。当用户向该文件系统写入数 据时，文件系统的压缩模块会将数据压缩后存储在磁盘上，在读取数据时，该模块 会先进行解压，然后返回给应用。

数据压缩技术主要是为了节省用户存储空间的使用。因为有些文件其实有很多 重复的数据，通过数据压缩，其冗余的数据将会大大减少。

目前，很多文件系统都支持数据压缩技术，比较常见的有Windows 中 的NTFS、 Solaris 中 的ZFS 和 Linux 中 的Btrfs 等。

第3章知其所以然—— 本地文件系统原理及核心技术

**3.3** **常见本地文件系统简介**

**3.3.1 ExtX 文件系统**

前文已经对ExtX 文件系统做过不少介绍，本节简单介绍一下ExtX 文件系统 的历史和特性。如果能了解一下ExtX 文件系统的发展历史，也能对该文件系统的 发展有一个更加全面的认识。

Linux 最早是参考Minix 实现的，其文件系统也是参考Minix 实现的。Linux 的 第1个版本并没有自己的文件系统。

Minix 毕竟只是一个用来教学的文件系统，其最大支持的磁盘的容量只有 64MB。 而且对文件名也有长度限制，最大为14个字符。这些限制导致该文件系统 很难在实际生产环境中使用。

针对上述问题，1992年Remy Card设计并实现了Ext 文件系统，即扩展文件 系统。其寓意也是实现了对Minix 的扩展。为了同时支持Minix 和 Ext, 同时实现 了VFS, 并 在Linux 内核的0.96c版本中集成发布。

1993年Remy Card开发出了Ext2 文件系统，该版本对Ext 文件系统又做了很 多扩展。这个版本是第1个商业级的文件系统，应用时间也非常长。但是Ext2 文 件系统没有日志特性，因此无法解决系统崩溃导致数据不一致的问题。

针对没有日志的问题，2001年Stephen Tweedie主导开发了Ext3 文件系统。该 文件系统主要是在Ext2 文件系统的基础上增加了日志(Journal) 特性。通过日志 特性，在系统出现崩溃的情况下可以通过扫描日志快速实现对文件系统的修复。

2008年，Ext4 文件系统出现，该版本引入了很多新特性，如Extent、预分配、 延迟分配和加密等。作者认为Extent 是所有特性中最突出的特性，它提供了一种新 的管理文件数据的方法，使得某些场景下文件的元数据大幅减少。

可以看出来，ExtX 文件系统也是经过几十年的发展才慢慢壮大的，才有了特性 丰富的Ext4 文件系统。

**3.3.2 XFS文件系统**

XFS 文件系统于1994年由SGI 开发，运行在IRIX 操作系统中I9] 。1999年 SGI 将XFS 文件系统开源，并移植到Linux 中。此后，XFS 作为一个可选文件系统一 直存储在Linux 内核中。

XFS 是一个64位的文件系统，因此其可以管理非常大的空间，并且支持非常

文件系统技术内幕

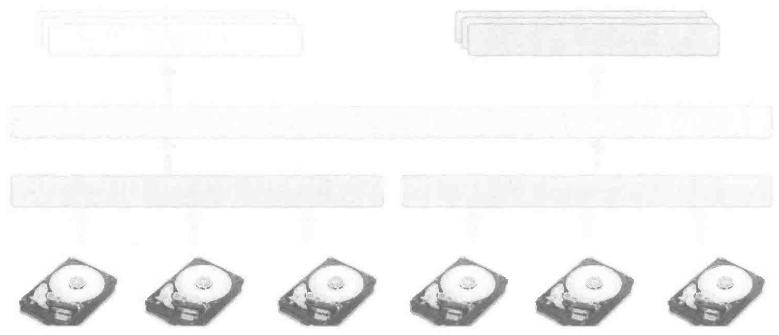
大数据时代海量数据存储之道

大的文件。XFS 文件系统最大可以支持18EB 的存储空间，并且可以创建最大为 9EB 的文件，这是Linux 原生文件系统ExtX 所无法企及的。由于XFS 没 有ExtX 文件系统中的inode 表，它可以随意创建inode, 因 此XFS 文件系统没有文件数量 的限制。

**3.3.3 ZFS文件系统**

ZFS 是 由Sun ( 已 被Oracle 收购)开发的高级文件系统。2001年由Matthew Ahrens 和 Jeff Bonwick领导开发，并于2005年随OpenSolaris 一起发布。ZFS 是 一 个128位的文件系统，因此可以支持非常大的存储空间，被称为下一代文件系统。 在ZFS 文件系统上可以创建16EB 的文件，并且该文件系统最大可以支持256千万 亿ZB(1ZB=1000EB)。

ZFS 不仅实现了文件系统，还实现了卷管理的功能。也就是说，ZFS 文件系统 实现了对多个磁盘的管理，并基于多个磁盘构建软RAID。如图3-49所示，ZFS 文 件系统可以基于多个设备构建存储池，然后在存储池中创建文件系统或卷。

Fle Systam

ZVOL

ZES Pool

VDEV

VDEV

图3-49 ZFS 文件系统存储池原理

虽然ZFS 文件系统是基于Solaris 开发的，但目前已经可以移植到Linux。遗 憾 的 是ZFS 文件系统无法直接集成到Linux 内核中，因此我们在Linux 内核中看不到 ZFS 文件系统的身影。无法集成到Linux 内核的主要原因是ZFS 文件系统遵循的 是CDDL 协议，与GPL 协议有冲突。无论如何，我们还是可以在Linux 中试用一 下ZFS 文件系统的。

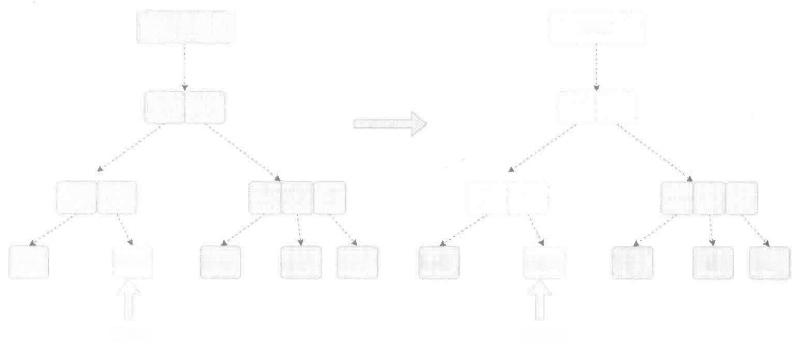
第3章知其所以然—— 本地文件系统原理及核心技术

**3.3.4 Btrfs 文件系统**

Btrfs的开发其实是为了有一个Linux 版本的ZFS 文件系统，因此其大部分特 性与ZFS 文件系统相同。前文在介绍ZFS 时，由于ZFS 文件系统的协议的问题无 法直接集成到Linux 内核中。在Oracle 收 购Sun 之后，看到了ZFS 文件系统的诸 多优点，于2007年着手在Oracle Linux中开发一个类似ZFS 的文件系统，由于其 采用B+ 树来管理数据，因此命名为Btrfs(B-Tree-FS) 文件系统。

Btrfs 文件系统实现了ZFS 文件系统的很多特性，如对多磁盘的管理、文件系 统快照和写时拷贝等。其中，写时拷贝是Btrfs 文件系统最大的特性。由于该特性 的存在，Btrfs 文件系统通过COW 日志保证系统崩溃时文件系统的数据一致性。 Btrfs 文件系统写时拷贝的原理很简单，当数据被修改时，新数据并不会覆盖旧数 据，而是写到新的地方。同时，与该数据相关的元数据也不会原地修改，而是在新 位置重新写一份。

如图3-50所示，左侧树是一个管理文件数据的B+ 树。如果用户修改其中的某 数据块，那么该树上相关的中间节点的数据并非在原地修改，而是分配新的空间来 存储修改后的数据。由于分配新的空间，所以中间节点存储的地址信息需要同时进 行修改，采用相同规则，该中间节点也需要重新分配空间。以此类推，从根节点到 待修改数据块的所有节点都需要分配新的空间，因此修改该数据块所影响的所有节 点如右侧树虚线框所示。



根节点

数摇块 数据块

写人数据

数掘块 数猫块

芎入数

数握块

数摇块

数编块

根节点

数猫块

数据块

数端块

图3-50 Btrfs 文件系统写时拷贝原理

Btrfs 文件系统有很多优点，但是其性能和稳定性相比Ext4 文件系统和XFS 文 件系统还是要差一些，因此在实际生产环境的应用比较少一些。

**文件系统技术内幕**

大数据时代海量数据存储之道

**3.3.5 FAT文件系统**

FAT(File Allocation Table,文件分配表)是1977年微软为DOS 开发的管理软 盘的文件系统。FAT 文件系统的最早版本是FAT12, 由于其管理的容量非常有限， 后来又陆续开发了FAT16 文件系统和FAT32文件系统。这里的阿拉伯数字表示数 据地址的位数，位数越大，可以表示的空间也就越大。

FAT32 文件系统最大可以创建4GB 的文件，所管理的空间最大为8TB。虽 然 FAT32 文件系统已经做得比较大了，但是跟Linux 下的几个动辄EB 级的文件系统 相比还是差很多。

后来微软又开发了一套新的文件系统，即exFAT 文件系统。该文件系统主要是 为了适应闪存介质而开发的，并且突破了FAT32 文件系统对容量管理的限制，可以 实现EB 级容量管理。

**3.3.6 NTFS文件系统**

NTFS(New Technology File System)是微软用于代替FAT文件系统的第二代 文件系统，于1993年首次被引入操作系统中。

NTFS 在容量方面有了很大的突破，整个文件系统可以管理16EB 的空间，而 单个文件大小可以达到256TB。除了容量的突破，NTFS 还有很多现代文件系统的 高级特性，如日志、压缩和加密等。

第 章

**从理论到实战——Ext2** **文件** **系统代码详解**

前文主要介绍文件系统的理论，大家应该对文件系统的原理有了基本的认识。 接下来以一个具体的文件系统为例来介绍一下文件系统原理的更多细节和具体实 现，这样大家能够更加具体地理解文件系统的原理。

对于文件系统代码，本章主要以Linux 下 的Ext2 文件系统为例进行介绍。选 择Ext2 的原因是它是Linux 原生的文件系统，并且具备文件系统的主要特性，另 外该文件系统又不过于复杂(约1万行代码)。正所谓“麻雀虽小，五脏俱全”,非 常适合入门者学习。

Ext2 是一个非常有历史的文件系统。1997年就应用在了RedHat 的发行版中。 Ext2 文件系统的前身是Ext 文件系统，Ext 是为了克服Minix 的诸多缺点，由Rémy Card 开发的基于虚拟文件系统的第一代扩展文件系统。

**4.1** **本地文件系统的分析方法与工具**

虽然 Ext2 文件系统的代码量并不多，但是直接阅读代码，理解整个流程还是 有一定难度的。Linux 内核程序的最大问题就是不太好调试。虽然不好调试，但是 Linux 提供了其他工具来窥探其内部。同时，Linux 也提供了很多其他工具来帮助 我们学习文件系统。

正所谓“欲善其事，必先利其器”,在真正进入Ext2 文件系统的学习之前，先 了解一些可以帮助我们学习Ext2 文件系统的工具。

**文件系统技术内幕**

大数据时代海量数据存储之道

**4.1.1** **基于文件构建文件系统**

构建文件系统并不一定需要磁盘或其他类型的块设备，我们可以直接在一个文 件上构建一个文件系统。在Linux 中基于文件构建文件系统非常方便，而且也便于 我们对文件系统的内容进行分析。接下来看一下如何基于文件构建一个文件系统， 总体来说分为以下几个步骤。

(1)生成一个全0的二进制文件。

可以通过dd 命令来生成一个全0的二进制文件。下面生成一个100MB 的 文 件，采用100MB 的文件足够构建一个Ext2 文件系统，而且便于后面分析，命令 如 下 ：

dd if=/dev/zero of=/ext2.bin bs=IM count=100

(2)格式化Ext2 文件系统。

有了100MB 文件之后就可以在该文件上格式化文件系统。方法很简单，采用 平时格式化磁盘的命令即可，命令如下：

mkfs.ext2 ext2.bin

( 3 ) 使 用loop 设备，仿真块设备。

虽然可以格式化文件系统，但是无法像块设备一样挂载到目录树中。通过Linux 的 loop 设备，可以将一个文件模拟成一个设备，这样就可以挂载访问了，命令 如 下 ：

losetup/dev/loop10./ext2.bin

(4)挂载文件系统。

完成上述过程后就已经有一个块设备了(名称为/dev/loop10), 然后就可以挂载 该文件系统，命令如下：

mount /dev/loop10/tmp/ext2/

完成挂载后就可以访问文件系统，如在里面创建文件和目录等。当然，我们也 可以向文件系统的根目录拷贝文件。所有操作的数据都会更新到创建的ext2.bin 文 件中。然后我们可以通过查看 ext2.bin 文件的内容学习Ext2 文件系统磁盘空间布 局的相关内容。

**4.1.2** **了解函数调用流程的利器**

在 Linux 中有一个可以非常方便地跟踪内核API 调用的工具——ftrace。我 们 可以通过该工具跟踪某些模块的函数调用，这样有助于理解代码调用流程。

ftrace的使用并不复杂，我们只需要执行以下几个步骤(基于Ubuntu 20.04,其

第4章从理论到实战 — — Ext2文件系统代码详解

他发行版本可能略有不同)即可。

(1)切换到debug 目录。

为了方便操作，先切换到ftrace 的工作目录，具体路径为： /sys/kernel/debug/tracing

( 2 ) 启 用 图 形 化 函 数 跟 踪 。

ftrace 的功能非常强大，这里选择其图形化跟踪函数调用的功能。具体设置方 式是执行如下命令：

echo function\_graph>current\_tracer ( 3 ) 设 置 过 滤 参 数 。

在默认情况下会跟踪所有函数调用，瞬间可能就有几万条记录，不方便我们分 析 。ftrace 支持过滤设置，可以只跟踪某些函数或不跟踪某些函数。例如，下面设

置 只 跟 踪xfs\_ 开头的函数，命令如下： echo "xfs\_\*">set\_ftrace\_filter

(4)查看跟踪到的内容。

以上就完成了设置。然后制造一些函数调用。比如，在一个XFS 文件系统的 目录中执行Is命令，然后打开ftrace目录下的trace文件，可以看到如图4-1所示 的内容。这个函数调用栈就是XFS 文件系统遍历目录时涉及的函数。

5650 θ) xfs\_file\_readdir [xfs](《

56510) xfs\_readdir [xfs](){

56530) xfs\_bmap\_last\_offset [xfs]0{

56520) xfs\_dir2\_isblock [xfs]0(

56540) xfs\_bmap\_last\_extent [xfs]0{

xfs\_fext\_last [xfs]0;

0.163 us xfs\_iext\_get\_extent [xfs10:

0.908 us 

1.210 us 

1.585 us 

xfs\_dir2\_leaf\_getdents [xfs]0{

xfs\_ilock\_data\_map\_shared [xfs]0 亻

0.178 us xfs\_ilock [xfs](0):

0.471

us

0)

0.177 0.480 0.155 1.762 3.929

us us

us us us us

56680) 56690)

56700)

4.253



xfs\_dir2\_leaf\_readbuf

xfs\_iext\_lookup\_extent



xfs\_iunlock [xfs]0;

1

I 1

[xfs]0{

[xfs]():

图4-1 ftrace 捕获结果

这里只是 一 个简要的介绍。ftrace 的功能非常强大，可跟踪的内容也非常多。 更多的介绍不在本书的范围内，请参考相关书籍，这里就不再赘述。

文件系统技术内幕

大数据时代海量数据存储之道

**4.2** **从** **Ext2** **文件系统磁盘布局说起**

前文已经介绍过关于Ext2 文件系统磁盘空间布局的相关内容，但是介绍的内 容相对比较概要。本节将更加深入地介绍 Ext2 文件系统磁盘布局的相关内容。 理解文件系统的磁盘布局是阅读代码的基础，因此有必要详细介绍一下这部分 的内容。

**4.2.1** **Ext2文件系统整体布局概述**

Ext2 文件系统将磁盘划分为大小相等的逻辑块 (Block) 进行管理。在格式化 时 ，mkfs.ext2 命令会根据块设备大小自行选择逻辑块大小。Ext2 文件系统逻辑块 的大小也可以在格式化时手动设置，可以是1KB 、2KB 和 4KB 等 。Ext2 文件系统 将磁盘划分为逻辑块，就好像将一栋大厦划分为若干个房间，或者将超市划分为若 干个货架区一样，主要是为了人们方便管理。

同时为了便于管理和避免访问冲突，将若干个逻辑块组成一个大的逻辑块，称 为块组 (Block Group)。块组是 Ext2 文件系统对磁盘管理的一个子空间。通常来 说，Ext2 文件系统是以块组作为一个相对独立的空间来进行管理的。块组的数据被 划分为两部分， 一部分是元数据区；另一部分是数据区。

元数据区存储的是文件系统的元数据，元数据是文件系统的管理数据，用于对 数据区的数据进行管理。数据区中的内容则是用户文件中的实际数据。为了更加直 观地理解上述概念的关系，图4-2展示了磁盘的块组与数据管理。

通过图4-2可以看出， 一个磁盘的线性空间被划分为相等大小的块组(最后一 个块组的容量可能要小一些)。每个块组都包含元数据区和数据区两个区域。

如果还是不太清楚，我们可以将磁盘理解为一个大厦。大厦整个空间好比磁盘 的整个存储空间；而房间是对大厦规划后的结果，好比对磁盘的格式化；大厦每层 的布局图好比元数据。我们可以通过楼层和每层的布局图很容易地找到房间。文件 系统与此类似，它通过元数据查找和管理逻辑块，也就是数据。

每个块组内部都有相关的元数据对该块组的空间进行管理。实际上 Ext2 文件 系统块组的内部结构还要复杂得多。以第1个块组(块组0)为例，元数据包括超 级块、块组描述符、数据块位图、inode 位图、inode 表和其他数据块。

第4章从理论到实战 — — Ext2文件系统代码详解

元数据区 数据区

文件系统块

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 块组0 | 块组! | 块组2 | 块组3 | 块组4 | 块组5 |

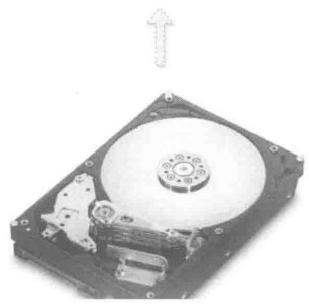


图4-2 磁盘的块组与数据管理

当然，并非每个块组都这么复杂。如果磁盘的存储空间充足，除第1个块组和 另外几个对超级块进行备份的块组外，大部分块组只有数据块位图、inode 位 图 和 inode 表等元数据信息。也就是说，块组其实分为两种类型： 一种是有超级块的， 比较复杂的块组；另一种是没有超级块的，比较简单的块组。

还有一个需要说明的地方是引导块。引导块并非文件系统中的一部分，而是预 留给引导操作系统用的。在操作系统加电启动时，其内容由BIOS 自动装载到内存 并执行。它包含一个启动装载程序，用于从计算机安装的操作系统中选择一个启动， 还负责后续启动过程。因此，Ext2 文件系统把这个区域预留出来，不作为文件系统 管理的磁盘区域。

**4.2.2 超级块(SuperBlock)**

超级块是文件系统的起始位置，它是整个文件系统的入口。文件系统的挂载(初 始化)就是从读取这里的数据开始的。由于它是一个数据块，但是又是一个非常特 别的块，因此被称为超级块。

在超级块中记录了整个文件系统的描述信息，如格式化时指定的文件系统逻辑

文件系统技术内幕

大数据时代海量数据存储之道

块大小信息、逻辑块的数量、inode 的数量、根节点的ID 和文件系统的特性等信 息。我们可以通过dumpe2fs 命令查看文件系统的超级块信息。

另外，为了保证整个文件系统的可靠性，Ext2 文件系统对超级块进行了备份。 备份的目的主要是应对突然断电或系统崩溃等异常场景。可以保证在上述场景下， 即使第1个超级块出现损坏，仍然可以通过其他块组中的超级块进行恢复，不至于 整个文件系统都不可访问。

对 于 4KB 大小的逻辑块，超级块位于第1个逻辑块内。由于第1个块组预留 了 1KB 的空间作为系统引导区，因此该块组的超级块的位置在1KB 偏移处，而其 他备份块组中的超级块都在该块组偏移为0的地方。超级块会占用一个逻辑块的空 间(实际占用空间要小于该值),也就是说，块组描述符(ext2\_group\_desc) 位于超 级块下一个逻辑块开始的地方。以4KB 为例，则块组描述符位于4KB 偏移的地方； 以 1KB 为例，则块组描述符位于2KB 偏移的地方。

代码4 - 1是Ext2 文件系统超级块在磁盘存放的结构体，磁盘数据被读取后按 照该结构体的格式进行解析。其中， \_lexx 变量表示小端对齐，使用时需要转换为 CPU 的对齐方式。在文件系统中还有另一个名为super\_block 的结构体，这个结构 体用于代码逻辑中使用。

**代码4-1** **Ext2文件系统超级块在磁盘存放的结构体**

|  |  |  |  |
| --- | --- | --- | --- |
| fs/ext2/ext2.h | | | |
| 417 | struct ext2\_super\_block | |  |
| 418 | \_le32 | s\_inodes\_count; | //inode总数量 |
| 419 | le32 | s\_blocks\_count; | //逻辑块总数量 |
| 420 | le32 | s r blocks\_count; | //保留的逻辑块数量 |
| 421 | le32 | s\_free\_blocks\_count; | //可用的逻辑块数量 |
| 422 | le32 | s\_free\_inodes\_count; | //可用的inode数量 |
| 423 | le32 | s\_first\_data\_block; | //第1个数据块的位置 |
| 424 | le32 | s\_log\_block\_size; | //逻辑块大小 |
| 425 | le32 | s\_log\_frag\_size; | //碎片大小 |
| 426 | le32 | s\_blocks\_per\_group; | //每个块组中逻辑块的数量 |
| 427 | le32 | s\_frags\_per group; | //每个块组中碎片的数量 |
| 428 | le32 | s\_inodes\_per\_group; | //每个块组中inode的数量 |
| 429 | le32 | s\_mtime; | //挂载时间 |
| 430 | le32 | s\_wtime; | //写数据时间 |
| 431 | le16 | s\_mnt\_count; | //挂载数量 |
| 432 | le16 | s\_max\_mnt\_count; | //最大挂载数量 |
| 433 | le16 | s\_magic; | //魔数标记 |
| 434 | le16 | s\_state; | //文件系统的状态 |
| 435 | le16 | s\_errors; | //检测到错误时的行为 |
| 436 | \_le16 | s\_minor\_rev\_level; | //次级修订版本 |
| 437 | le32 | s\_lastcheck; | //上次检查的时间 |
| 438 | le32 | s\_checkinterval; | //两次检查的间隔 |

第4章从理论到实战 — — Ext2文件系统代码详解

|  |  |
| --- | --- |
| 439 | \_le32 s\_creator\_os; //操作系统 |
| 440 | \_le32 s\_rev\_level; / 修 订 版 本 |
| 441 | \_le16 s\_def\_resuid; //保留块的缺省用户ID |
| 442 | lel6 s\_def\_resgid; //保留块的缺省组ID |
| 443 |  |
| 444 |  |
| 445 | /\*这些域仅仅被EXT2\_DYNAMIC\_REV使用。 |
| 446 |  |
| 447 | \*注意：兼容特性集与非兼容特性集的差异在于， |
| 448 | \*非兼容特性集中包含一个内核感知不到的位集合， |
| 449 | \*内核应该拒绝挂载该文件系统。 |
| 450 |  |
| 451 | \*e2fsck的需求更加严格。如果一个特性既不属于兼容特性集， |
| 452 | \*又不属于非兼容特性集，则必须放弃它，不会试图干预 |
| 453 | 它不理解的内容 |
| 454 | \*/ |
| 455 |  |
| 456 | le32 s\_first\_ino; //第1个非保留inode |
| 457 | le16 s\_inode\_size; //inode.结构体的大小 |
| 458 | le16 s\_block\_group\_nr; //超级块的块组号 |
| 459 | le32 s\_feature\_compat; //兼容特性集 |
| 460 | le32 s\_feature\_incompat; //非兼容特性集 |
| 461 | le32 s\_feature\_ro\_compat; //只读兼容特性集 |
| 462 | \_u8 s\_uuid[16]; //卷的128位UU ID |
| 463 | char s\_volume\_name[16]; / 卷 名 称 |
| 464 | char s\_last\_mounted[64]; //上次挂载的目录 |
| 465 | le32 s\_algorithm\_usage\_bitmap;//用于压缩 |
| 466 |  |
| 467 |  |
| 468 |  |
| 469 | /性能提示，如果打开EXT2\_COMPAT\_PREALLOC旗标，则会有目录预分配 |
| 470 | \_u8 s\_prealloc\_blocks; //试图预分配块编号 |
| 471 | \_u8 s\_prealloc\_dir\_blocks; //针对目录预分配块编号 |
| 472 | u16 s paddingl; |
| 473 |  |
| 474 |  |
| 475 | //如果设置了EXT3\_FEATURE\_COMPAT\_HAS\_JOURNAL,则开启日志特性 |
| 476 | u8 s\_journal\_uuid[16]; //日志超级块的UUID |
| 477 | u32 s\_journal\_inum; //日志文件的inode编号 |
| 478 | u32 s\_journal\_dev; Ⅱ日志文件的设备编号 |
| 479 | u32 s\_last\_orphan; //预删除inode链表的起始位置 |
| 480 | u32 s\_hash\_seed[4]; //哈希种子 |
| 481 | u8 s\_def\_hash\_version; //要使用的缺省哈希版本 |
| 482 | u8 s\_reserved\_char\_pad; |
| 483 | u16 s\_reserved\_word\_pad; |
| 484 | le32 s\_default\_mount\_opts; |
| 485 | le32 s\_first\_meta\_bg; //块组的第1个元块 |
| 486 | u32 s\_reserved[190]; //填充本块的尾部 |
| 487 |  |

**文件系统技术内幕**

大数据时代海量数据存储之道

虽然超级块中的内容非常多，但并不难理解，其中的信息大多是描述性内容。 目前，不理解超级块中的内容也没有关系，随着深入学习后续内容，相信读者会慢 慢理解。

**4.2.3** **块组描述符(Block** **Group** **Descriptor)**

块组描述符是对块组进行描述的一个数据结构。块组描述符紧跟在超级块之 后，需要注意的是，块组描述符信息是以列表的形式跟在超级块之后的，它包含所 有块组的描述信息。

块组描述符的描述信息包括对应块组中数据块位图的位置、inode 位图的位置 和 inode 表的位置等信息。另外，还包括数据块和inode 的剩余情况等信息。代码 4-2是块组描述符的结构体，该结构体是磁盘上的数据内容，可以看出该结构体占 用32字节的空间。

**代** **码** **4** **-** **2** **块** **组** **描** **述** **符** **的** **结** **构** **体**

**fs/ext2/ext2.h**

|  |  |  |
| --- | --- | --- |
| 199 | struct ext2\_group\_desc |  |
| 200 | { |  |
| 201 | le32 bg\_block\_bitmap; | //数据块位图的位置 |
| 202 | le32 bg\_inode\_bitmap; | //inode位图的位置 |
| 203 | le32 bg\_inode\_table; | 1/inode表的位置 |
| 204 | \_le16 bg\_free\_blocks\_count; | //剩余可用块的数量 |
| 205 | le16 bg\_free\_inodes\_count; | //剩余可用inode的数量 |
| 206 | \_le16 bg\_used\_dirs\_count; | //目录数量 |
| 207 | \_le16 bg pad; |  |
| 208 | \_le32 bg\_reserved[3]; |  |
| 209 | h |  |

为了更加直观地了解块组描述符，我们通过一个实例进行分析。首先创建一个 100MB 的空白文件，然后使用1KB 块进行格式化。由于1KB 块大小时块组大小为 8MB, 因此该文件系统会创建13个块组，最后一个块组大小为4MB(100-12×8)。

可以通过dumpe2fs 命令获取关于该文件系统的块组信息，如图4-3所示。可 以看到，块组的信息是一字排开的，每个块组中包含各个关键的位置信息和剩余的 资源信息。

第4章从理论到实战——Ext2文件系统代码详解

Group θ:(Blocks 1-8192)

Primary superblock at 1,Group descriptors at 2-2 Reserved GDT blocks at 3-258

Block bitmap at 259(+258)

Inode bitmap at 260(+259)

Inode table at 261-507(+260)

7671 free blocks,1965 free inodes,2 directories Free blocks:522-8192

Free inodes:12-1976

Group 1:(Blocks 8193-16384)

Backup superblock at 8193,Group descriptors at 8194-8194

Reserved GDT blocks at 8195-8450 Block bitmap at 8451(+258)

Inode bitmap at 8452(+259)

Inode table at 8453-8699(+260)

7685 free blocks,1976 free inodes,θdirectories Free blocks:8700-16384

Free inodes:1977-3952

Group 2:(BLocks 16385-24576)

Block bitmap at 16385(+0)

Inode bitmap at 16386(+1)

Inode table at 16387-16633(+2)

7943 free blocks,1976 free inodes,0directories Free blacks:16634-24576

Free inodes:3953-5928

图4-3 块组的描述信息

虽然通过dumpe2fs 命令可以很容易获取块组的信息，但还想要更进一步看一

看这些数据在磁盘上是如何存储的。可以通过vim 命令和hexdump 命令查看前面 创建文件系统时的文件内容。在Linux 中，可以通过vim 命令或hexdump 命令查 看 其 中 的 内 容 。

由于格式化的文件系统逻辑块大小是1KB, 通过前文我们知道引导块和超级 块各占用了1个逻辑块，因此块组描述符的起始位置应该在2KB 偏移的地方。根 据这个信息，我们可以通过hexdump 命令来读取该区域的数据，命令如下：

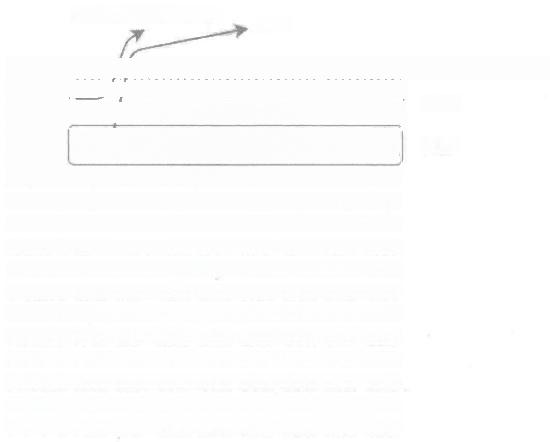
hexdump-n1024-s2048 ext2\_ 1kb.bin

从前文我们知道块组描述符占用32字节的空间，而且13个块组描述符一字排 开，就像结构体数组一样。如图4-4所示，块组0和块组1的数据分别由虚线框和

实现框框住。以块组描述符中逻辑块位图的位置信息为例，两个块组的值分别是259 和8451,对比通过hexdump 命令读取的数据和通过dumpe2fs 命令读取的数据，可 以看出两者是匹配的 。

文件系统技术内幕

大数据时代海量数据存储之道

259逻辑块位图的位置

8451

rootesunnyzhahf:~/test/ext2#hexdump -n 1024-s 2048 ext2\_ 1kb.bin

|  |  |
| --- | --- |
| 0000800:0103/do0001040000010500001df707ad: | 块组0 |
| 0000810:0002 boo400000000000000000000000 |  |
| 00008202103/000021040000210500001e0507b8  0000830000000040000000000001000000000000  0000840400100004002000040030000 If0707b8 000085000000004000000000000000000000000  0000860610300006104000061050000 le0507b8 000087000000004000000000000000000000000  00008808001000080020000-80030000 If0707b8 000089000000004000000000000000000000000  00008a0 a1030000 a1040000 a1050000 le0507b8 00008be 00000004000000000000000000000000 | 块组1 |

00008c0 c0010000 c0020000 c00300001f0707b8 00008d000000004000000000000000000000000

00008e0 e1030000 e1040000 e1050000 le0507b8 00008f000000004900000000000000090000000

图4-4 块组详细信息对比

本节只给出了逻辑块位图的位置数据，大家可以自行分析其他信息。比如，根 据数据结构对比分析 inode 位图的位置，剩余逻辑块数量和inode 数量等块组内的 其他信息。这样，大家就对块组的磁盘数据就有了比较具体的理解。

**4.2.4 块位图(Block Bitmap)**

块位图标识了块组中哪个数据块被使用了，哪个数据块没有被使用。在块位图 区中将1字节 (Byte) 划分为8份，也就是用1位(bit) 对一个逻辑块进行标记。 如果该位是0,则表示该位对应的逻辑块未被使用；如果该位是1,则表示该位对 应的逻辑块已经被使用。如图4-5所示，块位图区中的深灰色表示1,浅灰色表示 0;逻辑块区中的实线表示已经分配的空间。



图4-5 块位图磁盘空间管理的基本原理

块位图位于每个块组中，其大小为一个文件系统逻辑块。由于文件系统逻辑块 的大小是可变的(格式化时指定),而块位图永远占用一个逻辑块的空间，这就是

第4章从理论到实战—— Ext2文件系统代码详解

为什么块组大小是变化的原因。

以默认块大小4KB 为例，可以管理4096×8个逻辑块，即4096×8×4096=128MB 的空间。而在前面我们格式化的1KB 的文件系统，则块组的大小是1024×8×1024= 8MB。

为了更加直观地了解块位图，我们可以使用hexdump 命令将本实例中第1个 块组(块组0)的块位图导出，如图4-6所示。通过图4-6可以清晰地了解到块组 中哪些块被分配了，哪些没有被分配。

Footasunnyzhang:-/test/ext2#hexdump-V-n 1024-S 265216 ext2\_1kb.bin

0040c00 ffff ffff ffff ffff ffff ffff ffff ffff 0040c10 ffff ffff ffff ffff ffff ffff ffff ffff 0040c20 ffff ffff ffff ffff ffff ffff ffff ffff 0040c30 ffff ffff ffff ffff ffff ffff ffff ffff 0040c4001ff 0000000000000000000000000000 0040c5000000000000000000000000000000000 0040c6000000000000000000000000000000000 0040c7000000000000000000000000000000000

图4-6 块组中的块位图

计算该块组已经使用的逻辑块的数量，可以得到其数值为521(16×4×8+9)。 再结合图4-3可以看出两者是匹配的。

**4.2.5 inode位** **图(inode Bitmap)**

inode 位图与逻辑块位图类似，标识inode 的使用情况，并对其进行管理。其 中 ，inode 位图中的每一位与inode 表中的 一 个inode 对应。如果这 一位为1,则说 明 inode 表中的inode 已经被分配出去，否则就表示该 inode 可以被使用。inode 位 图占用一个文件系统逻辑块的空间。

**4.2.6 inode与** **inode表**

在Linux 中 ，inode 是一个非常基础的概念，它用于标识和管理一个文件/目录。 这里面涉及两个含义， 一个是标识一个文件；另一个是管理文件的元数据和数据。

标识一个文件是指这个inode 包含 一 个inode ID,而 该inode ID在文件系统中 是唯一的。从文件的查找和访问本质上来说是通过该 inode ID来查找的，最终定位 到 inode。

管理元数据和数据是指通过该inode 可以获取文件的元数据和数据信息。元数 据信息很容易被得到，基本全在inode 结构体中；而数据信息则是通过该结构体中

文件系统技术内幕

大数据时代海量数据存储之道

的一个数组描述的，在该数组中存储着数据的位置信息(可以先这么简单理解，后 面进行详细介绍)。代码4-3是inode 结构体在磁盘上的数据结构，可以看出inode 包含的内容还是比较多的，大部分内容比较直观，本节不再赘述。

**代码4-3** **inode结构体在磁盘上的数据结构**

fs/ext2/ext2.h

|  |  |
| --- | --- |
| 302 | struct ext2\_inode{ |
| 303 | le16 i\_mode; /文件访问模式，描述不同用户的访问权限 |
| 304 | le16 i\_uid; //用户ID信息 |
| 305 | le32 i\_size; //文件大小 |
| 306 | le32 i\_atime; //访问时间 |
| 307 | le32 i\_ctime; //创建时间 |
| 308 | le32 i\_mtime; //修改时间 |
| 309 | le32 i\_dtime; /删除时间 |
| 310 | le16 igid; //组ID的低16位 |
| 311 | le16 i\_links\_count; //链接数量 |
| 312 | \_le32 i\_blocks; //块数量 |
| 313 | \_le32 i\_flags; //文件旗标 |
| 314 | union{ |
| 315 | struct |
| 316 | \_le32 l i reservedl; |
| 317 | linuxl; |
| 318 | struct{ |
| 319 | le32 h i translator; |
| 320 | }hurdl; |
| 321 | struct{ |
| 322 | \_le32 m i reservedl; |
| 323 | }masixl; |
| 324 | osdl; //与操作系统相关的内容1 |
| 325 | \_le32 i\_block[EXT2 N BLOCKS]; |
| 326 | //数据块位置指针，用于记录数据的位置 |
| 327 | \_le32 igeneration; |
| 328 | \_le32 i\_file\_acl; //ACL数据的位置 |
| 329 | \_le32 i\_dir\_acl; //目录ACL数据的位置 |
| 330 | le32 i\_faddr; /碎片地址 |
| 331 | union{ |
| 332 | struct { |
| 333 | u8 Li\_frag; //碎片编号 |
| 334 | u8 i\_fsize; //碎片大小 |
| 335 | u16 i padl; |
| 336 | le16 l i uid\_high; |
| 337 | le16 I\_i gid\_high; |
| 338 | u32 L i reserved2; |
| 339 | }linux2; |
| 340 | struct { |
| 341 | \_u8 h i frag; //碎片编号 |
| 342 | \_u8 h i fsize; //碎片大小 |
| 343 | le16 h i mode\_high; |

**第** **4** **章** 从理论到实战——Ext2 文件系统代码详解

|  |  |  |
| --- | --- | --- |
| 344 | le16 | h i uid\_high; |
| 345 | le16 | h\_i gid\_high; |
| 346 | le32 | hi\_author; |
| 347 | }hurd2; |  |
| 348 | struct{ |  |
| 349 | u8 | m i frag; //碎片编号 |
| 350 | \_u8 | m i fsize; //碎片大小 |
| 351 | u16 | m\_padl; |
| 352 | \_u32 | m i reserved2[2]; |
| 353 | masix2; |  |
| 354 | }osd2; | //与操作系统相关的内容2 |
| 355 |  |  |

inode表以列表的形式紧跟在 inode 位图之后，每一项就是一个inode 节点。在 Ext2文件系统中，inode 的默认大小是128字节。另外，可以在格式化文件系统时 通过选项指定 inode 的大小，因此inode 表所占的空间并非固定的。

为了更加直观地理解inode 位图与inode 表及两者之间的关系，我们通过分析 磁盘空间的数据进行实际解析。在根目录下创建5个文件，如图4-7所示。

rootesunnyzhang:/tmp/ext2e tree /tap/ext2/ /tap/et2/

一 1.txt

12.txt -f3.txt

f4.txt

-f5.txt

一 tost+found

I directory 5 files

图4-7 在根目录下创建5个文件

由于创建的文件数量很少，因此数据都在第1个块组中，也就是块组0。关于 inode 位 图 和inode 表的位置可以通过块组0的描述符获得。首先看 一 下inode 位 图 的信息，如图4-8所示。

被使用的inode位图

rotesunyzhang vargblex/test hedunp n 10244:256240 ext2 akb.hin

0041000 t 0000 boe 0000000000000000000 06410100006e 0000.c0000000 D8Ce 20500000

00410200000000 oo0000000000000000000000 0041030.60300000000.000000008000000.0000

ce41040006000000000 00e 00000000

00420500006000600000000000000600

6041068000 000060800000000.00006000! 00410700o0000000000000000000000000000c0 0410 0000 g90000010800.0060.0000.60008000

00410900000000000050000000000000000000

00410a00000 co000000.00 0005000006000060 0041060000000000000.00000000.000000000oo0

本实例每个块组中inode的，

数量为1976,共需消耗247字节， 因此该逻辑块的剩余部分也被

00410e06000100000000000000000000000900 置位，表示不可用

20410d00000106000200000e 080000000000.0000 00416e0000 me00000000000 Q0000090000 60410r0030.00000300 ffo0 ffff ffff ffff ffff

0041100 ftftff f ffft ffff

004:110 tfff ffff ffff fftt fff ffff ftff ffff

**图4-8** **inode位图的信息**

文件系统技术内幕

大数据时代海量数据存储之道

通过图4-8可以看出，Ext2 文件系统共有16个inode被分配。这时可能会感 觉有些奇怪，明明创建了5个文件，为什么是16个inode 被分配呢?

这是因为Ext2 文件系统保留了前11个inode 作为内部使用。特别是inode ID 为 2 的inode是用来作为根目录的。因此，如果查看创建的文件，则inode ID是从 12开始的，如图4-9所示。

rootasunnyzhang-VirtualBox:-/test#Is

total 22K

2 drwxr-xr-x 3 root root

262145 drwxr-xr ·x3 root root

12-w-r 1 root root

79月

1.0K 9月

4.0K 9月

13 -rw-rr-1 root root

79 月

14 w-r I root root

79月

79月

79 月

15-rw-r 1 root root

16wr I root root

11 drwx……2 root root

12K 9月

alhi./mnt/ext2/

|  |  |
| --- | --- |
| 520:58 520:56 520:57 520:57 5.20:57 520:57 520:58 421:31 | f1.txt f2.txt f3.txt f4,txt f5.txt  lost+found |

图4-9 文件与inode ID的对应关系

然后返回inode 表 ，inode 表的位置信息在块组描述符中有记录。由于每个 inode 的大小固定，我们可以根据inode ID很容易找到inode 的 数 据 。 以f1.txt 文件为例， inode ID为12,于是我们可以找到该区域的数据。

通过inode 结构体的定义和图4-10中的数据，我们可以确定获取的数据正是该 inode的数据。以文件fl.txt(inodeID 为12)为例，其访问模式是rw-r--r--, 转换 成数字为0644。而对应 inode 中 i\_mode 成员的值为0100644(0x81a4), 可以看出 后半部分与访问模式是匹配的。前半部分不 一致是因为i\_mode 成员不仅用于访问 模式，还用于标识文件类型。这里fl.txt 是普通文件，因此对应的值是0100000(请 参考宏定义S\_IFREG), 两者进行或运算得到0100644(0x81a4)。

第4章从理论到实战 — — Ext2文件系统代码详解

004190041c00000300000008b2c 5f5341cf 5f52 004191041cf 5f52000000000000000200180000 0041920000000000000000001fd 000001fe 0000 004193001ff 0000020000000201000002020000

004194002030000020400000205000002060000

004195002070000020800000000000000000000

004196000000000000000000000000000000000

004197000000000000000000000000000000000

004198081a40000000700008b3e 5f538b3e 5f53 00419908b3e 5f53000000000000000100020000

00419a00000000000010000.0401000000000000 00419b000000000000000000000000000000000

00419c000000000000000000000000000000000

00419d0000000000000000010000000000000000 00419e0,000010000882c bf8d 0000000000000000

00419f0000000000000000000000000\_00000000

0041a0081a40000000700008b455f53/86455f53

0041a1086455f53000000000000.000100020000

0041a2000000000000100000402000000000000

004la3000000000000000000000000000000000

0041a4000000000000000000000.000000000000

004la5000000000000000000000000000000000

0041a6000000000868941c00000000000000000

0041a7000000000000000000000000000000000

0041a8081a410000000700008b4c 5f53 /8b4c 5f53 0041a908b4c 5f53000000000000000100020000

0041aa000000000000100000403000000000000 0041ab000000000000000000000000000000000 0041ac000000000000000000000000000000000

0041ad000000000000000000000000000000000

0041ae00000000020cf e5750000000000000000 0041af000000000000000000000000000000000 0041b0081a40000000700008b535f538b535f53

0041b108b53/5f530000000000000001100020000

**12**

13

图4-10 inode 表的磁盘数据

**4.3** **Ext2文件系统的根目录与目录数据布局**

Linux 中的文件系统采用的是层级目录树结构，因此任何文件必须要位于某个 目 录 中 。Linux 中的每个文件系统都要有一个根目录，这样才能基于该根目录来创 建文件和子目录。当文件系统的根目录挂载到Linux 文件系统目录树时，该根目录 就变成了目录树中的一个子目录。

目录本质上也是一个文件，只不过其存储的数据是一个特定的、格式化的数据。 而不像文件那样是一些文件系统不感知的字节流。那么这种格式化数据是什么样的 呢 ? 我 们 看 一 看Ext2 文件系统目录项的数据结构，如代码4-4所示。

文件系统技术内幕

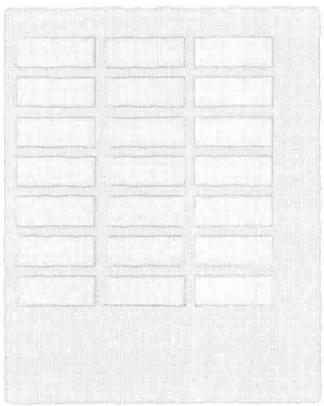
大数据时代海量数据存储之道

**代码4-** **4** **Ext2文件系统目录项的数据结构**

fs/ext2/ext2.h

|  |  |  |  |
| --- | --- | --- | --- |
| 598 | struct ext2\_dir\_entry\_2{ | |  |
| 599 | le32 | inode; | //inode ID |
| 600 | \_le16 | rec\_len; | //目录项的长度 |
| 601 | u8 | name\_len; | /名称长度 |
| 602 | u8 | file\_type; |  |
| 603 | char | name[]; | //文件名 |
| 604 | }; |  |  |

上述数据结构有多个成员，最主要的成员是inode 和 name,name 就是普通用 户看到的文件名，而inode 则是该文件的inode ID。另 外 ，rec\_len 和 name\_len 是 辅 助我们实现目录项遍历和管理的。在本实例中，目录项在目录中的组织如图4-11所 示(这里省略了部分内容)。



2 …… ……

262145  

12  11.txt

13 … 12.txt

14  3.txt

15  14.txt

16  f5.txt

图4-11 目录项在目录中的组织

可以看出，目录中的目录项主要建立了文件名与inode ID的一一对应关系。这 样，当用户通过文件名访问某个文件时，文件系统就可以根据文件名找到对应的 inode ID。由 于inode 在 inode 表中依次排列，因此也就可以根据inodeID 找到对应 的inode, 从而进行文件访问。

为了更加深刻地理解目录内容与数据结构的关系，我们可以对磁盘上的数据进 行分析。由于创建的文件位于根目录中，因此我们需要分析根目录的内容。具体方 法是先找到根目录的inode, 然后根据inode 中记录的数据位置信息找到该目录的 数据。

**第4章** 从理论到实战——Ext2文件系统代码详解



通过前文我们已经知道根目录的inode ID是2,因此可以很容易地根据该 inode ID从 inode 表中找到对应inode 的内容。由于 inode 表的起始位置是第260个逻辑 块，也就是在267,264(260×1024)字节偏移的位置，使用hexdump 命令可以获取 inode表前几个inode 的数据，如图4-12所示。

rootesunnyzhang:-/test/ext2s,hexdump n 10245.267264

0041400:0000000000009000,4ce160304ce16030 00414104cc16030000000000000000000000000 004142000000080000000000000000000000000

0041430:00000000000000000000000000000000

00414400o00000080000000.0o0e 000000000000 004145000000000000060000000000000000000

0041460|0000100000000000000001000000006000

0041470.0000.0000.0000.0000.00000000.0000.0000

00414804led 0000 D40000000712603106fa 6031 0041490 06fa 6031000000000000000300020000 00414a0 000000000005.000001fe 000000000000 00414b0 000010000000000000000000000000000

00414ce 000000000000.00000000000000000000 00414do 00000000000000000000000000000000

00000000000000000000000000000000

ext2\_ikb.bin

node ID=1

**根目录数据** **的位置**

inode ID=2

004150000000000000000000000000000000000

004151000000000000000000000000000000000

004152000900000000000000000000000000000

004153000000000000000000000000000000000

004154000000000000000000000|900000000000

0041550 D0000000000000000000000000000000

004156000000000000000000000000000000000

004157000000000000000000000000000000000

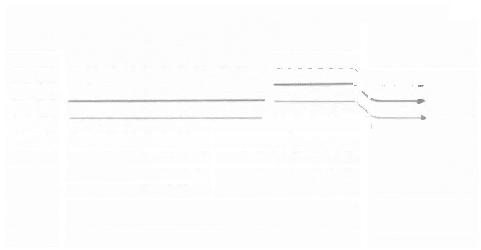
00415809000.0000000000006000.0000000010000 004159000000000000000000000000000000000

00415a0 D000000000000000p000000000000000

图 4 - 1 2 Ext2 根 目 录 的inode 内 容

对照磁盘上的数据与ext2\_inode (见代码4-3),我们可以找到该目录数据的存 储位置。图4-12中的01fc就是存储目录数据的位置，它是以文件系统逻辑块为单 位的。我们根据该偏移可以输出目录数据，如图4-13所示。

再次结合磁盘数据和目录项(见代码4-4)的定义，我们可以知道每一项的相 关内容。其中，fl.txt与f2.txt对应的目录项的数据如图4-13所示。

rootesunnyzhang:-/test/ext2a hexdump -n1024-5.520192 ext2\_Ikb.bin-w-C

0007f000 g2\_000000\_00.0192\_2990g00,02000000 I

0007f0100c 0002.022e 2e 00000b 00.000014.000a 02  0007f020 6c 6f 737425665f 756e 5400 0ek000\_00 ILost+found.

0007f030 10.0096016631.2e 747874.00 000d 000000、 f1.txt

0007f040 0 品 2 品000e 00.0000 2xt

0007f050 10.00.06.0166332e 747874.00 00 ef 00.0000 …f3.Fxt …

0007f060 100006.0166342e 74 7874.00 00100080100 …f4.fxt …

6007f070 94.03.060166352e 74 78 74.00 00.0000.0000 f5.txt

0097f080 000000.00 00 000000 00 00 00 0000008000 

0007f090 009000.0000) 00 .00 00 00 00 0000000000 

0007foa0 00000000 0000001 00 00 09 000000000000 

0007f0bo 00000000000000.000000.000000.0000.00 1 l

0007f0c0 0000.00000o <00100.000000.000000.000000> 

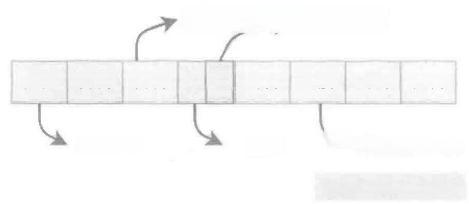
0007f0d 00.0000 D00000.00.0000.00.000000.000000 

**图4** **-** **13** **Ext2 文** **件** **系** **统** **目** **录** **数** **据**

**文件系统技术内幕**

大数据时代海量数据存储之道

如图4- 13所示，不同粗细和类型的线段表示一个目录项。通过分析上面数据 可以看出其内容与实例中目录内容是一致的。以fl.txt 文件为例，该文件对应相应 的目录项内容，对照上面介绍的数据结构，我们可以得到如图4- 14所示的内容。



目录项长度、文件名长度

000c 0000 0010 01 06 3166 742e 7448 0000

inode ID 类型  文 件 名 ：f1.txt

66312e744874

图4-14 目录项在磁盘中的数据

这里需要注意的是字节的对齐方式。以文件名为例，可以看出上述磁盘数据的 顺序与文件名并不一致，这与磁盘数据及内存数据的大小端对齐相关。

综上所述，目录项的本质是建立文件名与inode ID的关联关系。当通过文件名 访问文件时，其实本质上是找到文件对应的inode ID。然 后 根 据inode ID找 到inode, 之后就可以访问该文件的数据了。

**4.4** **Ext2文件系统的挂载**

对 于Linux 来说，在使用文件系统之前先要挂载文件系统。 <3.1.2.3>节已经比较 详细地介绍了文件系统的挂载流程。对于Ext2 文件系统来说，在挂载的流程中主 要 调 用 了ext2\_mount (函数，该函数主要从磁盘读取超级块的信息，完成超级块和 根目录的初始化，最终返回 一个dentry 指针，如代码4 - 5所示。

代码4 - 5 Ext2文件系统ext2\_ mount()函数

**fs/ext2/super.c**

|  |  |
| --- | --- |
| 1472 | static struct dentry \*ext2\_mount(struct file\_system\_type \*fs\_type, |
| 1473 | int flags,const char\*dev\_name,void\*data) |
| 1474 | { |
| 1475 | return mount\_bdev(fs\_type,flags,dev\_name,data,ext2\_fill\_super); |
| 1476 |  |

从代码4 - 5可以看出，ext2\_mount() 函数主要调用mount\_bdev 来完成工作，这 里主要传入了块设备名称和函数指针 ext2\_fill\_super 。 超级块结构体的填充主要在 ext2\_file\_super 函数指针完成，该函数指针会从块设备读取超级块的数据，并且填 充 到ext2\_sb\_info 结构体中，最终完成ext2\_sb\_info 和 super\_block 结 构 体 的 初 始化。

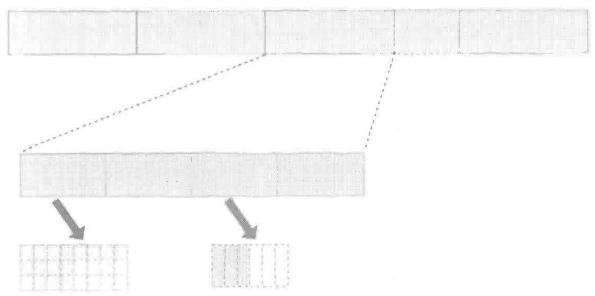
第4章从理论到实战 — Ext2文件系统代码详解

除了完成超级块相关结构体的初始化，ext2\_fill\_super 函数指针还有一个重要 的工作是从磁盘读取根目录的inode 信息，然后完成inode 和 dentry 的初始化及关 联。而这里的dentry 则是在VFS 挂载流程中必不可少的内容。

我们知道 inode 初始化时会完成操作函数指针的初始化。在挂载时Ext2 文件 系统完成了自己的根目录inode的初始化，这样在后续通过该inode访问数据时也 就是使用了Ext2 文件系统的函数来访问。

**4.5** **如何创建一个文件**

通过前文我们知道在Linux 中文件是由inode 标识的，每个文件在磁盘上都有 一 个inode 节点。对于Ext2 文件系统来说，通常这些 inode 节点会被相对集中地放 在一个区域，这个区域叫作inode 表，如图4-15所示。



块组0 块组1 块组2 …… 块组n

数据块位图 inode位图 inode表 数据块

图4-15 inode位图与inode表的关系

同时，通过前文我们了解了Ext2 文件系统的目录是如何组织数据的，并且了 解了目录与文件及文件数据的组织关系。如果简单地概括一下，则创建文件应该包 含以下几个步骤。

( 1 ) 从inode 表中申请一个inode。

(2)在目录中创建一个目录项。

(3)申请磁盘空间存储数据(如果存在写数据)。

本节将介绍Ext2 文件系统创建一个文件的流程及关键代码。在整个流程中将 涉及如何更改目录数据和申请inode 等内容。

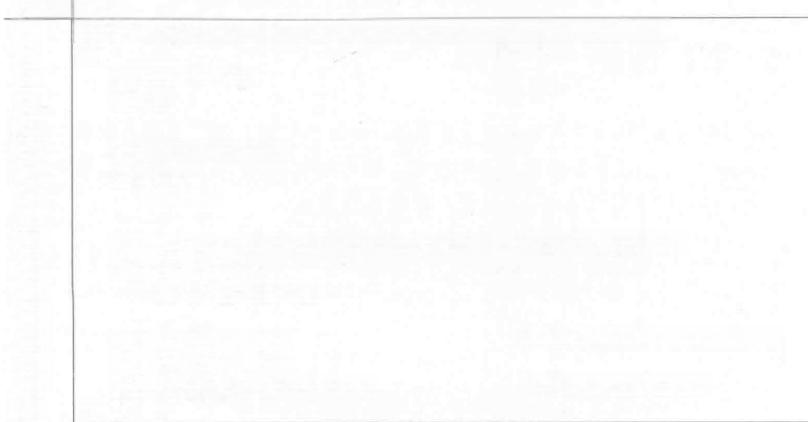
**文件系统技术内幕**

大数据时代海量数据存储之道

**4.5.1** **创建普通文件**

创建文件的操作通常由用户态发起，通过虚拟文件系统中的vfs\_create() 函数调 用文件系统的 create() 函数完成具体工作。对于 Ext2 文件系统，调用的接口是 ext2\_create() 函数，该函数的源代码如代码4-6所示。

**代码4-6** **Ext2文件系统创建文件的接口**



**fs/ext2/namei.c**

95 static int ext2\_create(struct inode\*dir,struct dentry\*dentry,umode\_tmode,bool excl) 96 (

97 structinode \*inode;

98 int err; 99

100 err=dquot\_initialize(dir);

101 if(err)

102 return err; 103

104 inode=ext2\_new\_inode(dir,mode,&dentry->d\_name); 分配 一 个新的inode

105 f(IS\_ERR(inode))

106 return PTR\_ERR(inode); 107

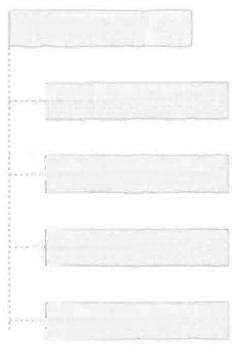
108 ext2\_set\_file\_ops(inode); / 为inode 设置操作文件的函数指针

109 mark\_inode\_dirty(inode);

110 return ext2\_add\_nondir(dentry,inode); //在目录中添加一项内容

111 

我们将上述函数简化为如图4-16所示的流程图。通过该流程图可以比较清晰 地看到创建文件的主要流程，分别是创建inode、设置函数指针和向目录中添加目 录项。接下来详细介绍每个函数的实现。



ext2\_create()

ext2\_new\_inode()

ext2\_set\_file\_ops0

mark\_inode\_dirty0

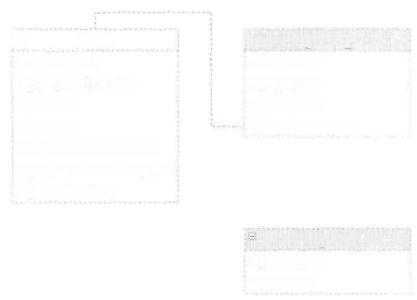
ext2\_add\_nondir(0

图4-16 创建文件的流程图

第4章从理论到实战——Ext2文件系统代码详解

ext2\_new\_inodeO 函数用于创建内存中的 inode 节点。根据 Ext2 文件系统的 inode 位图查找可以使用的 inode 表项，然后填充ext2\_inode\_info 结构体。最后完 成 inode 节点基本的初始化工作。需要注意的是，ext2\_new\_inode() 函数返回的inode 的数据结构(第104行)为struct inode,并非前文磁盘数据结构ext2\_inode 。struct inode 是内存中表示文件 inode 的数据结构，是VFS 中一个通用的数据结构。

与inode 相关的数据结构涉及3个，如图4-17所示(只展示部分成员)。其中， inode是 VFS 内存中的数据结构，提供一个抽象的文件节点。ext2\_inode\_info 是 Ext2 文件系统文件inode 在内存中的数据结构。ext2\_inode 是 Ext2 文件系统在磁盘上的 数据结构。

河 inode

+ mode umode\_t

\_optags unsigned shor

+ uitkuid\_I

gd kgid\_

+L\_op stuct inode\_operations°

+i\_mapping stuctaddress\_space\* +\_no unsigned long

汩 ext2 inode\_info

+ldata\_le32 \*\_lags\_u32 +Lfaddr\_032

+vs\_jnode:struct inode

ext2\_inode

+ mode\_le16

\_

+Lud:\_ie16

图4-17 与inode相关的数据结构

磁盘数据结构 ext2\_inode 只用于磁盘读/写的场景。当从磁盘读取数据时，会 先将磁盘数据读到该数据结构，然后将其中的数据赋值给ext2\_inode\_info 的成员。 对于Ext2 文件系统来说，这个赋值操作在ext2\_iget(函数中完成。当向磁盘写数据 时，涉及反向赋值的操作，也就是将ext2\_inode\_info 中的数据赋值给ext2\_inode 的 成员，这需要在\_ext2\_write\_inode() 函数中实现。

这 回ext2\_new\_inode() 函数的流程。首先分配内存数据结构；其次根据位图查 找可以使用的inode 表项，并且标记为已使用状态；最后根据分配的inode 表项来 初始化内存数据结构。

ext2\_new\_inode() 函数的流程图如图4-18所示。在该流程图中，new\_inode() 函 数是VFS 中的函数，其返回一个inode 指针。对于Ext2 文件系统来说，该函数实 质上是调用ext2\_alloc\_inode() 函数分配内存的，大家可以阅读一下该函数的实现代 码。上述返回的inode 指针本质上是ext2\_inode\_info 结构体的内存。返回inode 地 址实际上是一种实现抽象的方法，类似C++ 中父类的概念。可以很容易地根据inode

**文件系统技术内幕**

大数据时代海量数据存储之道

指针反向获取 ext2\_inode\_info 的指针，其他文件系统类似。Ext2 文件系统中的具

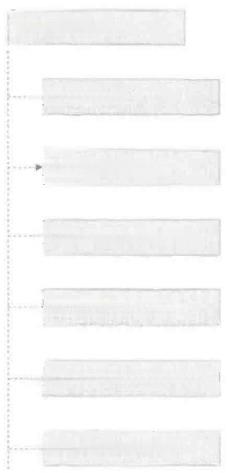
体实现如代码4-7所示。

**代码4-7** **EXT2** **I()函数**

fs/ext2/ext2.h

|  |  |  |
| --- | --- | --- |
| 714 | static | inline struct ext2\_inode\_info \*EXT2\_I(struct inode\*inode) |
| 715 | { |  |
| 716 | return | container\_of(inode,struct ext2\_inode\_info,vfs\_inode); |
| 717 | } |  |

接下来的几个函数主要从磁盘读取inode 的位图信息，确定可用inode 的过程。 最终，会从inode 表中选择一个可用 inode, 并将位图置位。然后是对 inode 和 ext2\_inode\_info 成员初始化的过程。



ext2\_new\_inode()

new\_inode()

find\_group\_other()

ext2\_get\_group\_desc()

read\_inode\_bitmap()

ext2\_find\_next\_zero\_bit()

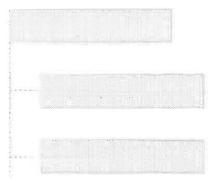
ext2\_set bit\_atomic()

图4-18 ext2\_new\_inode()函数的流程图

ext2\_set\_file\_ops() 函数用于设置Ext2 文件系统文件操作相关的函数指针，不同 类型的文件，函数指针略有不同。比如，目录文件与常规文件的函数指针是不同的。 这部分逻辑比较简单，大家可以自行阅读代码。

ext2\_add\_nondirO函数用于在目录数据中添加目录项，ext2\_add\_nondir()函数的 流程图如图4-19所示。前文已经介绍过目录数据的存储格式，大家可以参考一下。 目录项的具体查找和添加工作是通过调用ext2\_add\_link() 函数完成的。d\_instantiate\_ new() 函数用于建立dentry 与 inode 的关联，并更新 inode 的状态。

第4章从理论到实战 — — Ext2文件系统代码详解



ext2\_add\_nondir(O

ext2\_add\_link()

d instantiate new()

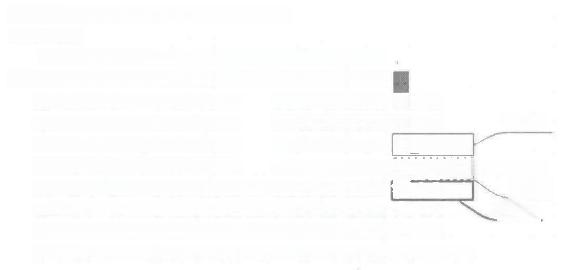
图4 - 19 ext2\_add\_nondir() 函 数 的 流 程 图

至此，就在Ext2 文件系统中成功创建了文件。由于Ext2 文件系统不支持日志 和事务，创建文件的整体流程还是比较简单的，更多细节大家可以自行阅读代码。

**4.5.2** **创建软硬链接**

前文已经介绍了关于链接的概念，本节以Ext2 文件系统为例看一 下软(符号) 链接和硬链接具体是怎么实现的。

如图4-20所示，我们在测试目录中创建了f3.txt 文件的两个链接，分别是硬链 接 f3\_pl.txt 和软链接f3\_sl.txt 。 图中第一列是文件的inode, 从 inode ID我们可以看 出硬链接f3\_pl.txt 和源文件f3.txt 的 inode ID是一样的，而软链接f3\_sl.txt 的 inode ID 是 一个新的ID。

rootesunnyzhang:/tmp/ext2#Ls -alhi

total 23K

2 drwxr-xr-x 3 root root 1.0K Feb 2013:33

24578 drwxnwxnwt 10 root root 4.0K Feb 2013:33

12-rw-r-r--I root root 8 Feb 2012:54 f1.txt

13-rw-厂-r 1 root root 8 Feb 2012:56 f2.txt 硬链接

14-Tw=r-r 2 root root 8 Feb 2012:56 f3\_pl.txt

17 lrwxrwxrwx I root root 6 Feb 2013:33:13\_sL.txt:->f3.txt 14 -nw-r-P-2 root root 8 Feb 2012:56 3.txt 软链接 15-nw-r--r 1 root root 8 Feb 2012:56 f4.txt 源文件 16-rw-r--r-1 root root 8 Feb 20.12:56 f5.txt

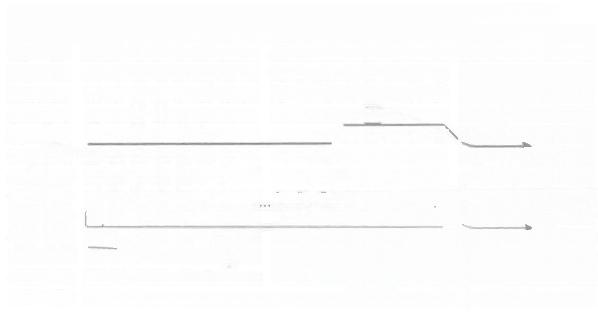
11 dnwx- 2 root root 12K Feb 1923:41 lost+found 图4-20 文件的软链接与硬链接

由此可知，硬链接其实就是在目录中添加了一个目录项，但并没有创建新的 node 。而软链接则是新建了一个inode, 也就是新文件。也就是说，软链接是通过 新文件指向目的文件的，那么它们具体是怎么实现的呢?

首先来看一下硬链接在磁盘上的数据。从图4-21中获取的目录中的数据，细 实线的内容为硬链接的内容，对比可以发现其 inode ID与 f3.txt 文件的inode ID相 同，也就是指向同一个inode。相同的 inode, 其中的内容是完全一致的。

**文件系统技术内幕**

**大数据时代海量数据存储之道**

rootesunnyzhang:-/test/ext2#hexdump-n 1024-s 520192 ext2\_ikb.bin-V-C 0007f00002000000 Bc 0001022e 000000020000001

0097f010 Bc 00102022e 2e l0000 eb 0080001400 aa D21 .

0007f020 6c 6f 73742b 666f 75 6e 640000 Bc 000000 Ilost+found...

0007f030 1000060166312e 74 78740000 d 000000 1.f1.txt …

0007f040 10100060166322e 74 78740000e 00.0000 1..f2.txt t

0007f050 1000060166332e 74 787400 000f 000000 JL….f3.txt.I

0007f060 10100060166342e 74 787400 0010000000 1f4.txt. 1

0007f070 1000060166352e 74 78.7400 的90 0 1 f5.txt.....

0007f080 14008907.66 33.5f.73 6c.2e. 7478.740000N f35L.txt…!

0007f090 e 00000070030901 66335f 706c 2e 7478 ….p.f3\_pl.tx|

0007f0a0 7400 0000000000 0000000000000000 It

0007f0be 00000000001000000 00000000000000801 l

0007f0c0 0000900000000000 0000000000000000 l

0007fed0 00000080.00000000000000000000 .00001..

图4-21 目录数据内容实例

然后看一下软链接在磁盘上的数据。图4-21中虚线部分的内容是软链接inode 的内容。对比可以发现软链接的inode ID为 0x11, 也就是17,与使用1s 命令获得 的内容一致。也就是说，软链接是另一个inode。那么两者是如何关联的呢?

我们可以通过软链接inode 的内容一探究竟。是否可以通过vim 命令或cat 命 令来查看内容呢?显然不行，通过命令查看的内容自然是f3.txt文件的内容。我们 可以通过直接查看磁盘内容的方式来看一看软链接inode 中存储的数据。

根据软链接的 inode ID可 以 从inode 表 中 获 取 该inode 的内容。本实例的inode ID 是17,因此可以计算出其偏移为269312字节。如图4- 22所示，可以看出在该 inode 中保存目的文件的名称。由于在创建软链接时使用的是相对路径(In-s f3.txt f3\_sl.txt), 因此这里保存的是文件名称。

rootesunnyzhang:~/test/ext2#hexdump-n 1024-5269312 ext2\_ 1kb.bin -v-C

00041c00 ff al000006000000 cb al 3560 aa f 3160. …5).11 00041c10 aa of 3160000000000000.010000000000[.1..

00041c20000000000100000066332e 74787400001.. f3.txt..

00041c3000000000000000000000000000000000|

00041c400000000000000000900000.0000000000i 1

00041c50000000.000000.00000000000000000000 !

00041c6080000000 fb 4b d2190000.0000000080.00[ K.. … …

00041c70900000000000900000000000000000001

80041c80000000001000000000000.000000000000 …

图4-22 基于相对路径软链接的内容

如果使用绝对路径创建软链接 (In-s/tmp/ext2/f3.txt f3\_sl.txt),则 会 得 到 如

图4- 23所示的内容。

**第4章从理论到实战** **—** **—** **Ext2文件系统代码详解**

rootesunnyzhang:~/test/ext2#hexdump -n 128 -s 269312 ext2\_1kb.bin =v-C

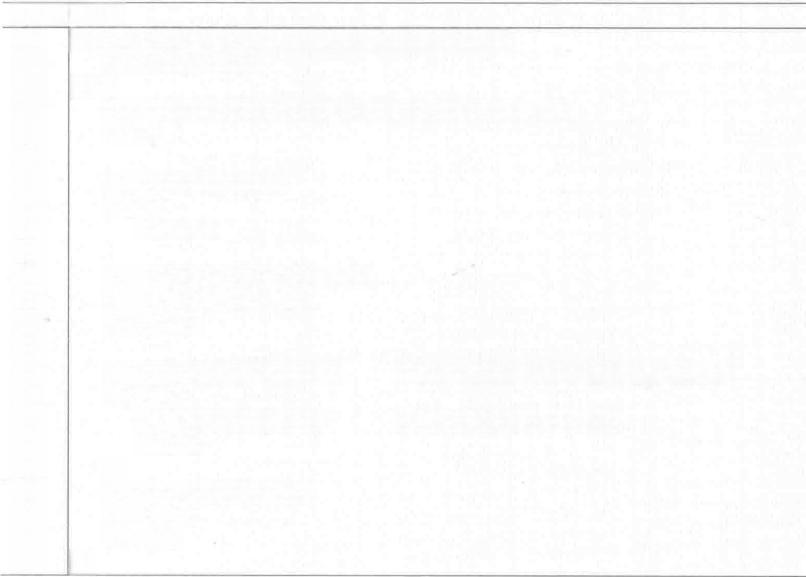
|  |  |
| --- | --- |
| 00041c00 ff al00001000.00003c a4356035 a435601 | s.55.5\*T |
| 00041c1035 a43560.000000000000010000000000 |5.5° |  |
| 00041c2000000000010000002f 746d 702f 657874 | /tmp/ext| |
| 00041c30322f 66332e 7478740000000000000000 12/f3.txt. |  |

00041c4000000000000000000000000000000000 1

00041c5000000000000000000000000000000000 I 1

00041c600000.0000 bf 1d 896100000000001000000 1a I

00041c7000000000.0000.0000000000.00.000000.00 I

图4-23 基于绝对路径软链接的内容

我们知道inode 默认大小是128字节。试想一下，如果路径的长度很长，则此 时这里的内容会是什么?请大家思考一下。如果不确定，则可以自行进行实验，本 节不再赘述。

**1.硬链接的创建流程**

了解了软链接和硬链接的数据组织，接下来看一下实现代码。硬链接的创建在 经过虚拟文件系统后会调用ext2\_link() 函数(见代码4-8)来完成。硬链接与源文件 共 用inode, 因此硬链接其实就是在目录数据中添加了一个与源文件一致的目录项， 只是文件名称不同而已。

**代码4-8** **Ext2文件系统创建链接主函数**

**fs/ext2/namei.c**

196

197

198

199

200

201

202

203

204

205

206

207

208

209

210

211

212

213

214

215

216

217

218

static intext2\_link(struct dentry\*old\_dentry,struct inode\*dir,

struct dentry \*dentry)



struct inode \*inode=d\_inode(old\_dentry);//找到源文件的inode

int err;

er=dquot\_initialize(dir); if(err)

return err;

inode->i\_ctime=current\_time(inode); inode\_inc\_link\_count(inode);

hold(inode);

er=ext2\_add\_link(dentry,inode); if(lerr){

//向目录中添加一个目录项，名称为链接名称 //建立dentry 与 inode 的 关 联

d\_instantiate(dentry,inode); return 0;



inode\_dec\_link\_count(inode); put(inode);

return err;

上

文件系统技术内幕

大数据时代海量数据存储之道

在 ext2\_link(函数中主要调用了ext2\_add\_link()函数，该函数完成目录项的添 加工作，前文创建文件时也是调用该函数完成的相关功能。由此可以看出创建硬链 接与创建文件的相同之处。

**2** **.** **软链接的创建流程**

由前文可知，软链接要创建新的 inode, 并 且 在inode 中填充源文件路径相关 数据(注意，这里说的是相关数据，并非一定是文件路径内容)。根据这个思路， 看一下其核心的代码，如代码4-9所示。

**代码4-9** **Ext2** **文件系统创建软链接主函数**

**fs/ext2/namei.c**

|  |  |
| --- | --- |
| 146 | static int ext2\_symlink(struct inode\*dir,struct dentry \*dentry, |
| 147 | const char\*symname) |
| 148 | { |
| 149 | struct super\_block\*sb=dir->i\_sb; |
| 150 | int err=-ENAMETOOLONG; |
| 151 | unsignedl=strlen(symname)+1; //源文件路径长度 |
| 152 | struct inode \*inode; |
| 153 |  |
| 154 | if(l>sb->s\_blocksize) |
| 155 | goto out; |
| 156 |  |
| 157 | err=dquot\_initialize(dir); |
| 158 | if(err) |
| 159 | goto out; |
| 160 | //创建一个新的inode |
| 161 | inode=ext2\_new\_inode(dir,S\_IFLNK|S\_IRWXUGO,&dentry->d\_name); |
| 162 | err=PTR\_ERR(inode); |
| 163 | if(IS\_ERR(inode)) |
| 164 | goto out; |
| 165 | //下面进行inode的初始化工作 |
| 166 | if(l>sizeof(EXT2\_I(inode)->i\_data)){ /源文件路径长度比较大的场景 |
| 167 |  |
| 168 | inode->i\_op=&ext2\_symlink\_inode\_operations;//软链接特殊的操作函数 |
| 169 | inode\_nohighmem(inode); |
| 170 | if(test\_opt(inode->i\_sb,NOBH)) |
| 171 | inode->i\_mapping->a\_ops=&ext2\_nobh\_aops; |
| 172 | else |
| 173 | inode->i\_mapping->a\_ops=&ext2\_aops; |
| 174 | err=page\_symlink(inode,symname,); //分配磁盘空间，写数据 |
| 175 | if(err) |
| 176 | goto out\_fail; |
| 177 | }else{ //inode内可以容纳的场景 |
| 178 |  |
| 179 | inode->i\_op=&ext2\_fast\_symlink\_inode\_operations; |
| 180 | inode->i link=(char\*)EXT2 I(inode)->i data; |

第4章从理论到实战——Ext2文件系统代码详解

181 memcpy(inode->i\_link,symname,1); / 直 接 存 储 在 inode 中

node->i\_size=1-1;



184 mark\_inode\_dirty(inode); 185

186 er=ext2\_add\_nondir(dentry,inode); //在目录中添加目录项

187 out:

188 return err;

190

192

194

out\_fail:

node\_dec \_linkcount(inode); discard\_new\_inode(inode);

goto out;

整体流程与创建文件差异不大，关键差异在于需要根据源文件路径长度判断数 据的存储位置。如果i\_data 数组(60字节)可以存储原始路径，则通过该数组存 储；否则需要从数据区分配新的存储空间进行存储。

**4.5.3** **创建目录**

创建目录由ext2\_mkdir(函数完成。整体流程与创建文件差异不大，因此本节 就不再重复介绍了。

**4.6** **Ext2文件系统删除文件的流程**

前文已经介绍过创建文件的流程，删除文件是创建文件的逆过程。根据前文创 建文件的流程，我们应该能够推测出删除文件的流程，主要包括以下几个步骤。

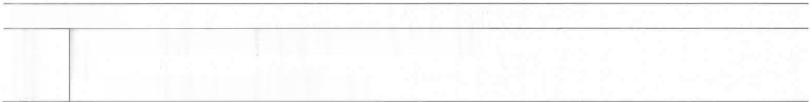
(1)删除目录中的对应目录项。

(2)释放文件使用的存储空间。

(3)释放该文件对应的inode。

在删除文件时先会调用VFS 中 的vfs\_unlink() 函数，该函数调用具体文件系统 的函数实现，如本实例中的ext2\_unlink() 函 数 。ext2\_unlink() 函数的具体实现如代 码4-10所示，首先对目录内容进行查询(第279行),确认是否存在想要删除的文 件；如果查询成功则返回一个目录项数据结构，然后删除该目录项(第286行)。

**代码4-10** **Ext2文件系统删除文件主函数**

fs/ext2/namei.c

|  |  |
| --- | --- |
| 266 | static int ext2\_unlink(struct inode\*dir,struct dentry \*dentry) |
| 267 | 1 |
| 268 | struct inode\*inode=d\_inode(dentry); |

**文件系统技术内幕**

大数据时代海量数据存储之道

|  |  |
| --- | --- |
|  | struct ext2\_dir\_entry\_2\*de;  struct page\*page; |
| 271 | int err; |
| 273 | err=dquot\_initialize(dir); |
| 274 | if(err) |
| 275 | goto out; |
| 276 |  |
| 277 | /\*根据想要删除的文件名，查询目录中是否有该文件 |
| 278 | 如果存在则返回对应的目录项\*/ |
| 279 | de=ext2\_find\_entry(dir,&dentry->d\_name,&page); |
| 280 | if(!de){ |
| 281 | err=-ENOENT; goto out; |
| 283 | 重重 |
| 284 |  |
| 285 | //删除查询到的目录项 |
| 286 | er=ext2\_delete\_entry(de,page); if(err) |
| 288 | goto out; |
| 289 |  |
|  | inode->i\_ctime=dir->i\_ctime; |
| 291 | node\_dec\_link\_count(inode); |
| 292 |  |
|  | out:  return err; |

删除目录项是最关键的步骤，代码4- 11是删除目录项的具体实现。通过 ext2\_delete\_entry (函数代码可以看出，该函数首先查找前一个目录项，然后更新这 个目录项中的长度信息。简而言之，在Ext2 文件系统中删除文件，并不是直接将 目录项删除，而是将该目录项的空间合并到前一个目录项中。

**代** **码** **4** **-** **1** **1** **从** **目** **录** **中** **删** **除** **目** **录** **项**

**fs/ext2/dir.c**

|  |  |
| --- | --- |
| 560 | int ext2\_delete\_entry(struct ext2\_dir\_entry\_2\*dir,struct page\*page) |
| 561 | { |
| 562 | struct inode \*inode=page->mapping->host; |
| 563 | char \*kaddr=page\_address(page); |
| 564 | unsigned from=((char\*)dir-kaddr)&~(ext2\_chunk\_size(inode)-1); |
| 565 | unsigned to=((char \*)dir-kaddr)+ |
| 566 | ext2\_rec\_len\_from\_disk(dir->rec\_len); |
| 567 | loff\_tpos; |
| 568 | ext2\_dirent\*pde=NULL; |
| 569 | ext2\_dirent\*de=(ext2\_dirent\*)(kaddr+from); |
| 570 | int err; |
| 571 |  |
| 572 | //查询想要删除目录项的前一个目录项 |

第4章从理论到实战 — — Ext2文件系统代码详解

573 while((char\*)de<(char\*)dir){ if(de->rec\_len=0){

575 ext2\_error(inode->i\_sb,\_func\_,

"zero-lengthdirectory entry");

577 err=-EIO;

goto out;



580 pde=de;

de=ext2\_next\_entry(de);

582 

583 if(pde)

584 from=(char+)pde-(char\*)page\_address(page);

585 pos=page\_offset(page)+from;

586 ock\_page(page);

587 err=ext2\_prepare\_chunk(page,pos,to-from);

BUG\_ON(err);

/斤更新前一个目录项的长度

590 if(pde)

591 pde->rec\_len=ext2\_rec\_len\_to\_disk(to-from);

592 dir->inode=0;

593 err=ext2\_commit\_chunk(page,pos,to-from);

node->i\_ctime=inode->i\_mtime=current time(inode);

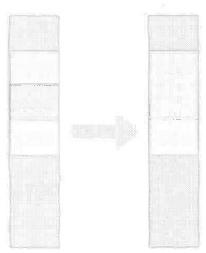
595 EXT2\_I(inode)->i\_flags &=~EXT2\_BTREE\_FL;

596 mark\_inode\_dirty(inode);

598 ext2\_put\_page(page);

599 return err; 600

目录项的合并其实非常简单，具体实现如代码4-11中第591行所示。这里只 需要更新一下前一个目录项的rec\_len 即可，这样想要删除的目录项就失效了(见 图4-24,后续进行目录项遍历会跳过删除的目录项)。为了标识该目录项失效，这 里同时将该目录项中的inode 的值更新为0。这样，我们通过该值就可以知道目录 项已经被删除。

f1.txt

f1.txt

f2.txt

f3.txt

f3.txt

图4-24目录项删除后目录内容变化

**文件系统技术内幕**

大数据时代海量数据存储之道

细心的读者可能会问，为什么没有看到释放数据和释放 inode 的代码逻辑?释 放数据和 inode 的逻辑确实不在这里，而是在内存 inode 释放时，代码4 - 12是 ext2\_evict\_inode() 函数的具体实现。

代码4-12 ext2 evict inode()函数

**fs/ext2/inode.c**

|  |  |
| --- | --- |
| 73 | void ext2\_evict\_inode(struct inode\*inode) |
| 74 | { |
| 75 | struct ext2\_block\_alloc\_info \*rsv; |
| 76 | int want\_delete=0; |
| 77 | //判断是否要删除该inode,如果i\_nlink的值大于0(在有目录引用的情况下)则不删除 |
| 78 | if(linode->i\_nlink &&lis\_bad\_inode(inode)){ |
| 79 | want\_delete=1; |
| 80 | dquot\_initialize(inode); |
| 81 | }else{ |
| 82 | dquot\_drop(inode); |
| 83 |  |
| 84 |  |
| 85 | truncate\_inode\_pages\_final(&inode->i\_data); |
| 86 |  |
| 87 | if(want\_delete){ |
| 88 | sb\_start\_intwrite(inode->i\_sb); |
| 89 | //更新删除时间戳 |
| 90 | EXT2\_I(inode)->i\_dtime =ktime\_get\_real\_seconds(); |
| 91 | mark\_inode\_dirty(inode); |
| 92 | \_ext2\_write\_inode(inode,inode\_needs\_sync(inode)); |
| 93 |  |
| 94 | inode->i\_size=0; |
| 95 | if(inode->i\_blocks) |
| 96 | ext2\_truncate\_blocks(inode,O);//释放文件占用的空间 |
| 97 | ext2\_xattr\_delete\_inode(inode); |
| 98 |  |
| 99 |  |
| 100 | invalidate\_inode\_buffers(inode); |
| 101 | clear\_inode(inode); |
| 102 |  |
| 103 | ext2\_discard\_reservation(inode); |
| 104 | rsv=EXT2\_I(inode)->i\_block\_alloc\_info; |
| 105 | EXT2\_I(inode)->i\_block\_alloc\_info=NULL; |
| 106 | if(unlikely(rsv)) |
| 107 | kfree(rsv); |
| 108 |  |
| 109 | if(want\_delete){ //如果确实要删除inode,则释放该inode,位图清零 |
| 110 | ext2\_free\_inode(inode); |
| 111 | sb\_end\_intwrite(inode->i\_sb); |
| 112 |  |
| 113 |  |

**第4章** 从理论到实战——Ext2文件系统代码详解



在ext2\_evict\_inode() 函数中，首先会判断是否真的要删除(释放)inode, 如 果 是则更新 inode 元数据，释放文件数据和扩展属性等内容，然后调用ext2\_free\_ inode()函数释放inode 。ext2\_free\_inode() 函数的作用就是根据inode ID找到对应的 位图来清理对应位的，此时表示 inode 表中的该inode 已经被释放。

至此，也就完成了删除文件的所有流程。删除链接和删除子目录的方法与删除 文件的方法类似，本节不再赘述。

**4.7** **Ext2文件系统中文件的数据管理与写数据流程**

第3章已经介绍了常见的文件数据的管理方法。Ext2 文件系统使用了基于索 引的文件数据管理方式，被称为间接块的方式。本节先对Ext2 文件系统间接块的 管理方式进行简要介绍，再详细介绍一些文件写数据的流程。

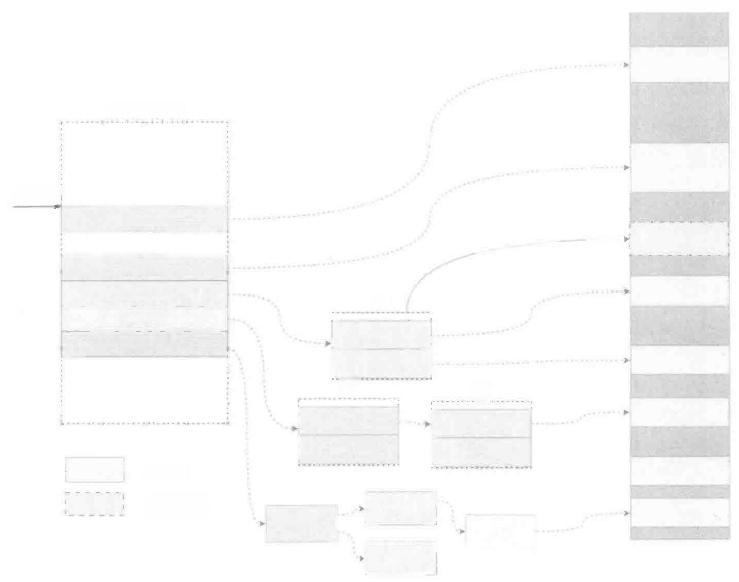
**4.7.1** **Ext2文件系统中的文件数据是如何管理的**

在Linux 文件系统中，文件是通过inode 来唯一标识的，而文件数据的位置是 通过 inode 中的成员记录的。在Ext2 文件系统中，文件数据的位置信息存储在 ext2\_inode 的 i\_block 成员变量中。该变量是一个32位整型数组，共有15个成员， 前12个成员中的内容为文件数据的物理地址，后3个成员存储的内容指向磁盘数 据块。数据块中存储的数据并不是文件的数据，而是地址数据。由于在这种情况下 数据块存储的并非是文件数据，而是inode 与文件数据中间的数据，因此被称为间 接块(Indirect Block,简 称IB)。

通过上面的描述大家可能理解的还不够清楚，我们结合图4-25来看一下一个 文件的 inode 与间接块和磁盘数据的对应关系。通过图4-25可以看出，数组i\_block 中 的block0～blockl1 中的地址对应的磁盘数据就是文件的用户数据。而block12 中 的地址对应的磁盘数据(1级间接块)则是元数据，该磁盘数据中的地址指向的数 据才是用户数据。2级和3级间接块与1级间接块类似，差异在于2级间接块需要 经过两级间接块才能找到用户数据，而3级间接块则需要经过3级间接块才能找到 用户数据。

**文件系统技术内幕**

**大数据时代海量数据存储之道**



disk blocks

ext2\_inode

Lblock.

biock0

block11 block12 block13

biock14

IB

IB

文件数据

IB,管理文性

数癣的元数擦

IB

IB

IB

IB

IB

图4-25 文件数据索引示意图

对于小文件来说，通过直接引用就可以完成数据的存储和查找。比如，在格式 化时文件逻辑块大小是4KB,48KB(4K×12) 以内的文件都可以通过直接引用完 成。但是，如果文件大于48KB, 直接引用则无法容纳所有的数据，48KB以外的数 据先要通过一级间接引用进行存储。以此类推，当超过本级存储空间的最大值时， 要启用下一级进行文件数据的管理。

理解了Ext2 文件系统对文件数据的管理方式之后，再阅读读/写数据的相关代 码就相对简单了。由于Ext2 文件系统基于VFS 的框架，因此在介绍Ext2 文件系统 的写流程之前，先介绍一下涉及VFS 的相关内容。

**4.7.2** **从** **VFS** **到Ext2文件系统的写流程**

前文已经介绍过缓存的相关知识，在用户接口与持久化存储之间是有一个缓存 层的。由于该缓存层的存在，因此写数据操作就会存在多种场景。

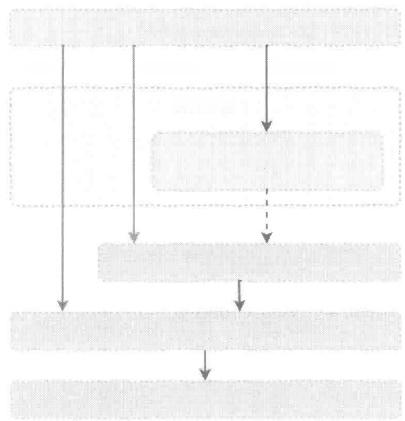
(1)DAX 模式：数据不经过文件系统缓存，也不经过通用块I/O 栈，直接通过 驱动程序将数据写入物理设备。

(2)Direct I/O模式：数据绕过文件系统缓存，但需要经过通用块I/O 栈 。

第4章从理论到实战 — — Ext2文件系统代码详解

(3)Normal I/O(常规)模式：数据会先写入缓存。同时有两种不同的处理方 式：一种是直接写入缓存层后返回；另一种是写入缓存层并等待数据写入持久化设 备后返回。

上述不同的场景可以通过图4-26进行描述，该图同时展示了上述3种场景。 本节将针对上述场景详细地分析一下Linux的 VFS 和Ext2文件系统的具体处理 流程。



应用程序

Drect iVo Normall/O

文件系统

缓存

通用块1/O 栈

设备驱动

持久化设备

DAX

图4-26 Linux文件系统写入数据的模式

首先，我们看一下VFS 是如何调用Ext2 文件系统接口的。我们知道VFS 通 过 具体文件系统注册的函数指针集来与具体文件系统交互。Ext2 文件系统的文件操 作函数指针集如代码4-13所示，这里包含了Ext2 文件系统实现的可以对文件进行 的操作，包括对文件内容的读/写、查找和缓存同步等。对Ext2 文件系统而言，VFS 对文件的操作都会调用Ext2 文件系统的函数指针来完成。

**代码4-13** **Ext2文件系统的文件操作函数指针集**

**fs/ext2/file.c**

181 const structfile\_operations ext2\_file\_operations={

182 .llseek =generic\_file\_llseek,

183 .read\_iter =ext2\_file\_read\_iter,

184 .write\_iter =ext2\_file\_write iter,

185 unlocked\_ioctl=ext2\_ioctl,

186 #ifdef CONFIG COMPAT

.compat\_ioctl =ext2\_compat\_ioctl,

188 #endif

mmap =ext2 file mmap,

**文件系统技术内幕**

大数据时代海量数据存储之道

190 .open =dquot\_file\_open,

191 .release =ext2 release\_file,

192 .fsync =ext2\_fsync,

193 .get\_unmapped\_area=thp get\_unmapped\_area,

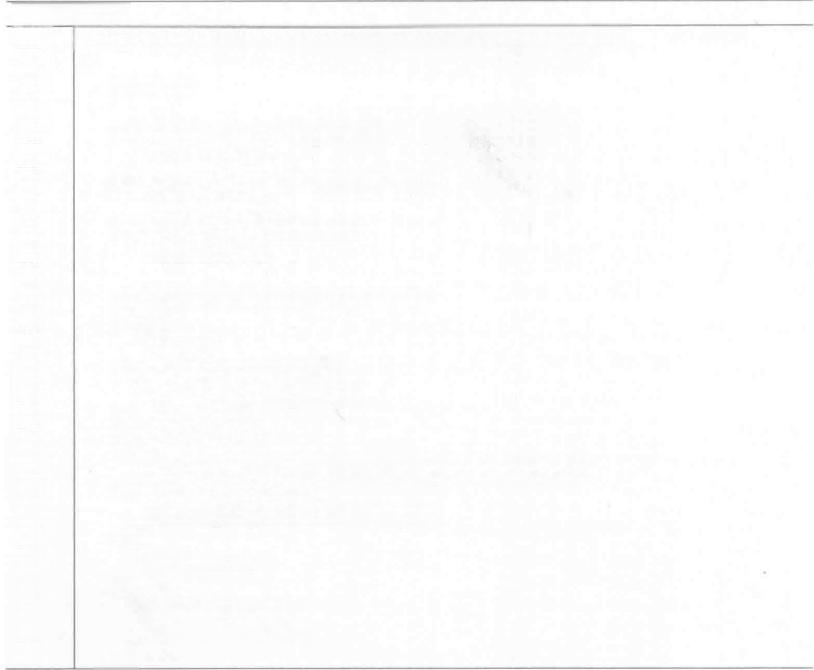
194 .splice\_read=generic\_file\_splice\_read,

195 .splice\_write=iter\_file\_splice\_write, };

对于写操作，系统API会首先调用VFS 的函数，然后经过VFS 的一些处理后 会调用具体文件系统注册的API 函数。对Ext2 文 件 系 统 而 言 ，VFS 会 调 用Ext2 文 件系统的函数，也就是ext2\_file\_write\_iter ( 函 数 。

然后，看 一 下VFS 文件系统写操作的代码实现，如代码4- 14所示。从该代码 中可以看出vfs\_write() 函数将调用具体文件系统的写数据的接口(第575行～第578 行),这里会根据具体文件系统对函数指针的初始化情况而执行不同的流程，Ext2 文件系统实现了write\_iter, 而 没 有 实 现write, 因 此Ext2 文件系统会走第2个分支 (第578行)。

**代码4-14** **VFS 文件系统写数据接口**

**fs/read\_write.c**

558 ssize\_t vfs\_write(struct file\*file,const char\_user\*buf,size\_t count,loff\_t\*pos) 559 (

560 ssize\_t ret; 561

562 if(!(file->f\_mode &FMODE\_WRITE)

563 return-EBADF;

564 if(!(file->f\_mode&FMODE\_CAN\_WRITE))

565 return-EINVAL;

566 if(unlikely(laccess\_ok(buf,count))

567 return -EFAULT;

568

569 ret=rw\_verify\_area(WRITE,file,pos,count);

570 if(ret)

571 return ret;

572 if(count>MAX\_RW\_COUNT)

573 count=MAX\_RW\_COUNT;

574 file\_start\_write(file);

575 if(file->f\_op->write)

576 ret=file->f\_op->write(file,buf,count,pos);// 具体函数指针

577 else if(file->f\_op->write\_iter)

578 ret=new\_sync\_write(file,buf,count,pos);

579 else

580 ret=-EINVAL;

581 if(ret>0){

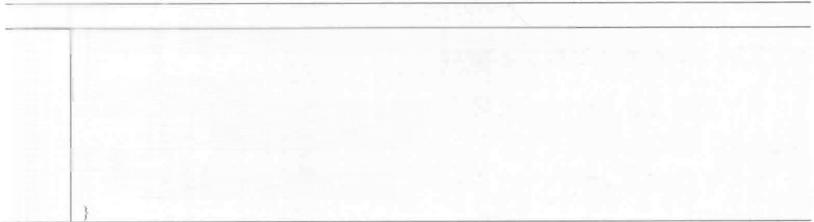
582 fsnotify\_modify(file);

583 add\_wchar(current,ret); 584 

第4章从理论到实战 — — Ext2文件系统代码详解

585 nc\_syscw(current); file\_end\_write(file); return ret;

通过上文描述，我们基本知道用户态的接口调用是如何触发Ext2 文件系统的 函数的，也就是如何调用Ext2 文件系统中的ext2\_file\_write\_iter() 函数，该函数的 具体实现如代码4- 15所示。可以看出，ext2\_file\_write\_iter() 函数可能会走不同的流 程，包括DAX 模式流程(第176行)和常规模式流程(第178行)。



**代** **码** **4** **-** **1** **5** **ext2\_file\_write\_iter() 函** **数** **的** **具** **体** **实** **现**

fs/ext2/file.c

172 static ssize\_t ext2\_file\_write\_iter(struct kiocb \*iocb,structiov\_iter \*from)

173 

174 #ifdef CONFIG\_FS\_DAX

175 if(IS\_DAX(iocb->ki\_filp->f\_mapping->host))

176 return ext2\_dax\_write\_iter(iocb,from);

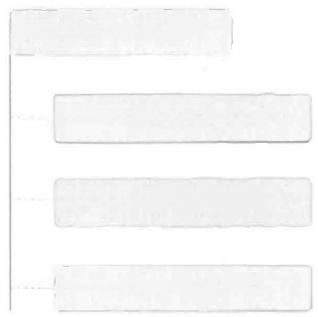
177 #endif

178 return generic\_file\_write iter(iocb,from);

179

如果选择DAX 模式流程，则会调用Ext2 文件系统中的ext2\_dax\_write\_iter() 函 数完成；如果选择常规模式流程，则会调用VFS 中 的generic\_file\_write\_iter() 函数。 对 于Ext2 文件系统来说，并不是简单的VFS 调 用Ext2 文件系统的流程，它还有一 个反向调用流程，这是因为VFS 提供了很多公共的功能(API)。

接着看一下generic\_file\_write\_iter() 函数，该函数涉及调用的主要函数，如图4- 27所示。其中， g eneric file write iter()为写流程的主要函数，该函数实现磁盘空 间分配和实际的写数据操作。generic\_write\_sync (函数在文件具备同步刷写属性的 情况下，实现缓存写数据的同步刷写。



generic\_file\_write\_itero

generic\_write\_checks0

\_generic\_file\_write\_itero

generic\_write\_synco

图4-27 VFS通用写数据的流程

**文件系统技术内幕**

大数据时代海量数据存储之道

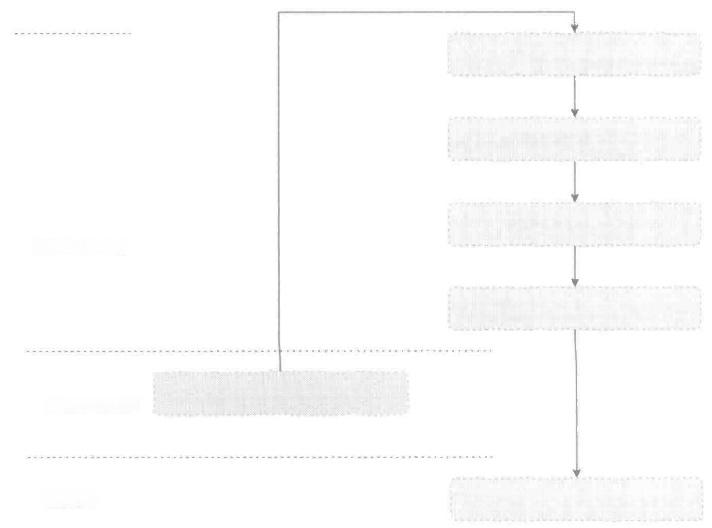
g eneric file write iter()也 是VFS 中的 一个函数 (mm/filemap.c 中实现)。在 该函数中，针对用户打开文件时设置的属性有两种不同的执行分支，如果设置了 O\_DIRECT 属性，则调用generic\_file\_direct\_write() 函数进行直写的流程；如果没有 设置0 \_DIRECT 属性，则调用generic\_perform\_write() 函数执行缓存写的流程。

**4.7.3** **不同写模式的流程分析**

通过前文分析我们知道了Linux 文件系统的不同写模式，并且厘清了每种模式 的入口函数。接下来将深入介绍每种模式的处理流程。

[**4.7.3.1**](4.7.3.1) **DAX模式写数据的流程**

对 于Ext2 文件系统的DAX 模式写数据的流程来说，其入口是ext2\_dax\_write\_ iter() 函数。该函数并没有太多自己的逻辑，核心功能还是调用VFS 关 于DAX 的 功 能，如图4-28所示。最终，VFS 会根据传入的DAX 设备，调用设备驱动接口来完 成数据的写入。



dax\_iomap\_w0

iomap\_applyo

dax\_iomap\_actor0

虚拟文件系统层

dax\_copy\_from\_itero

ext2\_dax\_write\_iter)

dax\_dev->ops->copy\_from\_iter()

Ext2文件系统层

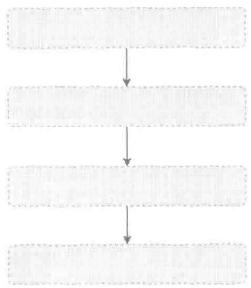
块设备层

图4-28 DAX 模式写数据的流程

以持久内存驱动(Persistent Memory Driver)为例，使用pmem\_copy\_from\_iter()

第4章从理论到实战 — — Ext2文件系统代码详解

函数实现上述设备的函数指针，该函数最终调用 memcpy\_flushcache() 函数将数据 写到物理设备上，如图4-29所示。



pmem\_copy\_from\_iter()

\_copy\_from\_iter\_flushcache0

memcpy\_flushcache(

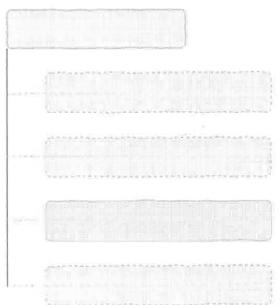
\_memcpy\_flushcache(0

图4-29 pmem\_copy\_from\_iter()函数写数据的流程

通过如图4-28和图4-29所示的流程可以看出，在DAX 模式下，数据并不会 经过页缓存，也不会经过块设备的I/O 栈，而是直接将数据拷贝到持久内存驱动 上存储。

[**4.7.3.2**](4.7.3.2) **Direct** **I/O模式写数据的流程**

Ext2 文件系统的 Direct I/O 模式写数据的流程入口是VFS 的 generic\_file\_ direct\_write(函数，该函数流程相对比较简单，主要调用了以下4个函数，如图4-30 所示。



genenic\_file\_direct\_write()

filemap\_write\_and\_wait\_range()

\*invalidate\_inode\_pages2\_range()

a\_ops->direct\_100)

invalidate\_inode\_pages2\_range()

图4-30 generic\_file\_direct\_write()函数写数据的流程

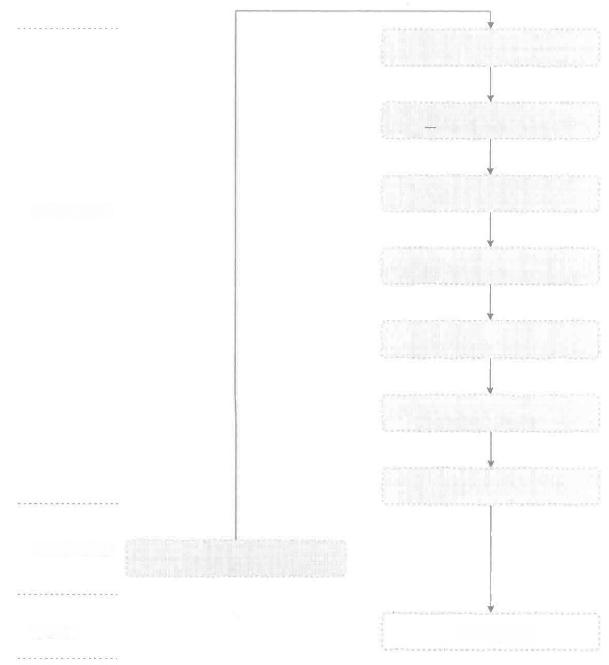
在上述4个函数中，前面两个函数是对目的区域的缓存进行刷写，并使缓存页 失效。进行这一步的主要原因是缓存中可能有脏数据，如果不进行处理就可能会出

文件系统技术内幕

大数据时代海量数据存储之道

现缓存的数据覆盖直写的数据，从而导致数据不一致。第3个函数direct\_IOO 是文 件系统的实现，执行真正的写数据操作。最后一个函数在上面已经执行过，主要是 避免预读等操作导致缓存数据与磁盘数据的不一致。

对 于Ext2 文件系统来说，其实现为ext2\_direct\_IO0 函数，该函数主要通过调 用 VFS 的函数来完成数据写入，如图4-31所示。关于该流程的更多细节我们不做 介绍，大家可以根据该流程图自行阅读代码，理解起来并不困难。这里需要说明的 是，该流程并不会经过缓存，而是最后调用块设备的submit\_bio (函数将数据提交到 块设备进行处理。

blockdev\_direct\_I00

\_blockdev\_direct\_100

do\_blockdev\_direct\_I00

成拟文件系统星

do\_direct\_I00

do\_direct\_l00

submit\_page\_section()

dio\_bio\_submit)

Exd2文件系统层

ext2\_direct\_IO0

submit\_bio(

块设备层

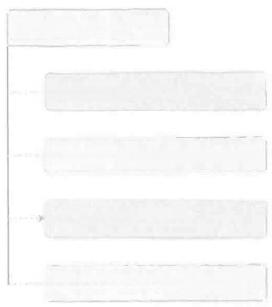
图4-31 ext2\_direct\_IO(函数写数据的流程

[**4.7.3.3**](4.7.3.3) **缓存写数据的流程**

在 非DAX 模 式 和Direct I/O模式的情况下，数据会首先写入缓存中，此时会 调用一个名为generic\_perform\_write (的函数。缓存写数据的流程也有4个主要步

第4章从理论到实战 — — Ext2文件系统代码详解

骤，分配磁盘空间和缓存页、将数据从用户态拷贝到内核态内存、收尾、页缓存均 衡，如图4-32所示。



genenic\_perform\_wrte()

a\_ops->write\_begin()

iov\_iter\_copy\_from\_user\_atomic0 a\_ops->wite\_end()

balance\_drty\_pages\_ratelimited0

图4-32 generic\_perform\_write()函数写数据的流程

其中，分配磁盘空间和缓存页、收尾工作是通过调用Ext2 文件系统注册的地 址空间操作函数指针完成的，相关函数指针如代码4-16所示。这里的函数指针是 文件系统对下层(如块设备)的接口。

**代码4-16** **Ext2文件系统中的相关函数指针**

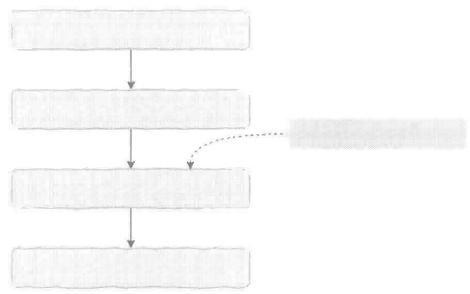
fs/ext2/inode.c

|  |  |  |
| --- | --- | --- |
| 967 | const struct address\_space\_operations ext2\_aops={ | |
| 968 | readpage | =ext2\_readpage, |
| 969 | readahead | =ext2\_readahead, |
| 970 | writepage | =ext2\_writepage, |
| 971 | write\_begin | =ext2\_write\_begin, |
| 972 | write end | =ext2\_write\_end, |
| 973 | .bmap | =ext2\_bmap, |
| 974 | .direct\_IO | =ext2\_direct\_IO, |
| 975 | .writepages | =ext2\_writepages, |
| 976 | .migratepage | =buffer\_migrate\_page, |
| 977 | is\_partially\_uptodate | =block\_is partially\_uptodate, |
| 978 | .error remove page | =generic\_error\_remove\_page, |
| 979 |  |  |

分配磁盘空间和缓存页的功能由ext2\_write\_begin() 函数来完成，如图4-33所 示 。 当ext2\_write\_begin() 函 数 调 用block\_write\_begin() 函数时会传入 一 个名为 ext2 get blockO的函数，从名称也可以看出，该函数是用来分配存储空间的。

文件系统技术内幕

大数据时代海量数据存储之道

ext2\_write\_begin)

block\_write\_begino

grab\_cache\_page\_wrte\_begin(O

\_block\_write\_begino

\_block\_write\_begin\_into

图4-33 ext2\_write\_beginO函数写数据的流程

在上述流程中，在调用 block\_write\_begin (函数之前，会通过grab\_cache\_ page write begin()函数获取缓存页。然后在 block\_write\_begin() 函数完成缓存页与 buffer 的映射关系的建立。

缓存写数据的流程可能存在两种不同的场景： 一种场景是缓存中已经有对应位 置的数据，此时只需要返回该缓存页即可；另一种场景是缓存中没有对应位置的数 据，此时需要分配新的缓存页，并且确定具体的数据是否在磁盘上已经存在，如果 磁盘上没有数据则存在分配磁盘空间的流程。

完成缓存页分配后，接下来调用 iov\_iter\_copy\_from\_user\_atomic() 函数将数据 从传入的参数拷贝到内核缓存页中。

最后，在该逻辑中会调用balance\_dirty\_pages\_ratelimited) 函数确定是否要进行 缓存数据的刷写。在每次写缓存时，都会调用该函数来检查一下页缓存的总容量， 如果超过设定的水线则会强制将数据刷写到持久化设备上。

需要说明的是，在图4-27的VFS 通用写数据流程中，最后会调用generic\_write\_ sync() 函数。该函数只对打开文件时设置O\_SYNC 选项有意义。在设置O\_SYNC 选 项的场景下，当数据写入缓存后并不会马上向用户返回结果，而是必须等待数据刷 写到持久化设备后才会返回。

**4.7.4** **缓存数据刷写及流程**

在前面写数据流程的介绍中，缓存数据刷写通常只是写入缓存后就返回。但缓 存的数据最后还是要刷写到持久化设备上的。在Linux 文件系统中有以下多种方式 可以将缓存刷写到持久化设备。

第4章从理论到实战——Ext2文件系统代码详解

(1)基于缓存水线的强制刷写。

(2)基于系统定时器的定时刷写。

(3)基于用户命令的手动刷写。

(4)基于挂载选项的同步刷写。

(5)基于打开文件选项的同步刷写。

上述刷写中基本上会走两个不同的流程： 一个是基于I/O 路径内容的同步刷 写；另一个是基于BDI 的异步刷写。基于BDI 的异步刷写需要借助后台线程来进 行刷写。虽然对于缓存刷写的路径略有不同，但最终调用的接口是一致的，其流程 如图4-34所示，最终会调用内存管理模块的do\_writepages(函数。

|  |  |
| --- | --- |
| 同步刷写 | 异步用写 |
| generic\_write\_sync0 | wb\_workfn0 |
| vfs\_fsync\_range() | wb\_do\_writeback0 |
| file->f\_op->fsync0 | wb\_writeback( |

ext2\_fsync0 writeback\_sb\_inodes(

|  |  |
| --- | --- |
| generic\_file\_fsync() | \_writeback\_single\_inode() |
| generic\_file\_fsync0 | \_writeback\_single\_inode( |

file\_write\_and\_waitrange()

\_filemap\_fdatawrite\_range()

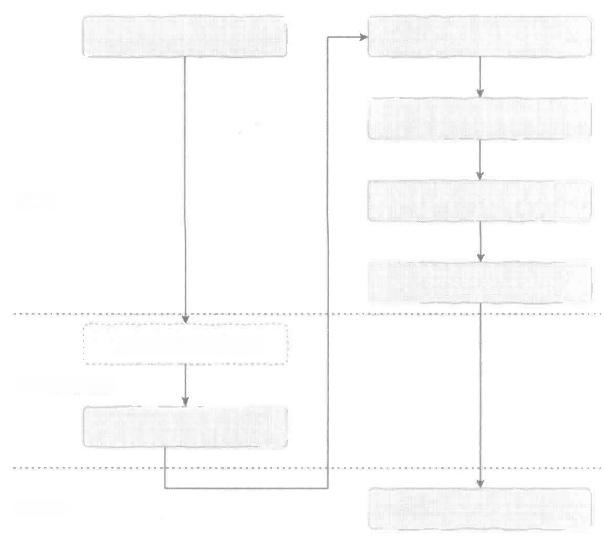
do\_writepages()

图4-34 同步刷写与异步刷写的流程

文件系统技术内幕

大数据时代海量数据存储之道

do\_writepages (函数是内存模块的接口，它会调用Ext2 文件系统注册的函数指 针，也就是ext2\_writepages() 函数。而该函数又反过来调用VFS 实现的函数，最终 VFS 层会调用块设备层的submit\_bio() 函数将数据提交到持久化设备(如磁盘),其 流程如图4- 35所示。



do\_writepages()

write\_cache\_pages()

VFS层

mpage\_bio\_submit()

a\_ops->writepages()

Ext2文件系统层

ext2\_writepages()

块没备层

\_mpage\_writepage()

mpage\_writepages)

submit\_bio()

图4-35 向磁盘写数据的流程

至此，用户应用通过API 写入的数据也就最终被写入持久化设备。

**4.8** **读数据的流程分析**

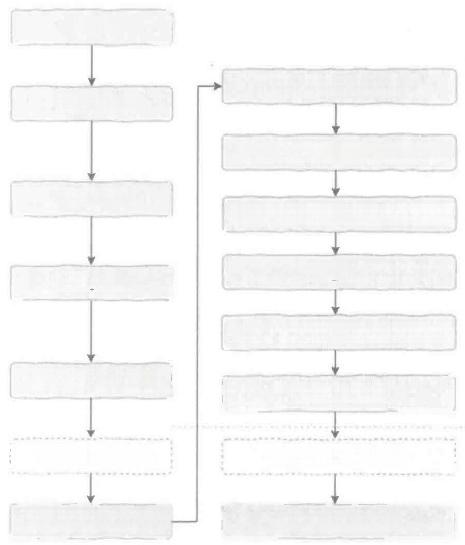
读数据的流程与写数据的流程类似，本节不再赘述如何通过VFS 调 用Ext2 文 件系统的流程细节，本节给出如图4- 36所示从用户态到 Ext2 文件系统的主线流 程。需要注意的是，读数据也涉及DAX 、Direct I/O和缓存读等模式。由于前两种 模式代码逻辑比较简单，本节不再赘述，本节重点介绍一下缓存读模式的相关流程， 如图4- 36所示。

第4章从理论到实战 — — Ext2文件系统代码详解

用户空间API层

VFS 层

Ext2文件系统层



generic\_file\_read\_iter()

generic\_file\_buffered\_read()

page\_cache\_sync\_readahead(

ondemand\_readahead()

\_do\_page\_cache\_readahead()

read\_pages()

a\_ops->readahead)

ext2\_readahead(

read()

sys\_read()

vfs\_read()

\_vfs\_read()

new\_sync\_read()

f\_op->read\_iter()

ext2\_file\_read\_jter()

图4-36 读数据的流程

对 于DAX 模 式 和 Direct I/O0模式，在执行generic\_file\_buffered\_read() 函数之 前就会返回，只有缓存读的场景下才会执行generic\_file\_buffered\_read() 函数，本节 主要介绍一下该函数的一些实现细节。

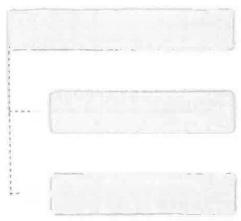
缓存读的流程概括分为两个主要步骤： 一个是从页缓存查找数据；另一个是根 据页缓存的状态从磁盘读取数据并填充页缓存(如果页缓存数据是最新的则不需要 从磁盘读取数据)。

**4.8.1** **缓存命中场景**

在读数据的流程中，缓存命中并且数据可用的情况下，那么整个读数据的流程 将非常简单。generic\_file\_buffered\_read (函数的处理逻辑可以简化为图4-37。也就 是主要流程包含两个步骤，分别是从缓存找到缓存页，然后是将内核数据拷贝到用 户缓冲区。

文件系统技术内幕

大数据时代海量数据存储之道



generic\_file\_buffered\_read()

find\_get\_page()

copy\_page\_to\_iter()

图4-37 读数据缓存命中的流程

当然，这里的前提是数据可用。实际还存在其他一些情况，如虽然页缓存存在 该数据，但不是最新的数据，需要从磁盘读取数据。或者页缓存存在该数据，但正 在进行预读操作，需要等待预读完成等。虽然需要做一些特殊处理，但总体来说函 数调用是比较简单的。

**4.8.2** **非缓存命中场景**

在非缓存命中场景中，文件系统需要向块设备发起读数据的请求。在VFS 的 实现中，会先调用同步预读的接口，也就是page\_cache\_sync\_readahead (函数，但 该函数只有在符合预读条件的情况下才会预读，否则直接返回。

这样在非缓存命中场景中有可能会执行两个不同的分支： 一个是同步预读分 支；另一个是普通分支。同步预读分支主要是针对连续读的场景，而普通分支则是 针对随机读的场景。本节主要介绍随机读场景的代码实现(见代码4-17),关于预 读分支请参考下一节的介绍。

非预读场景核心在于同步预读函数的执行结果。由于随机读不符合预读条件， 因此同步预读函数不会预读数据，这样通过第2036行代码就无法获取缓存页，最 终会执行第2218行代码(标签no\_cached\_page 的位置)。

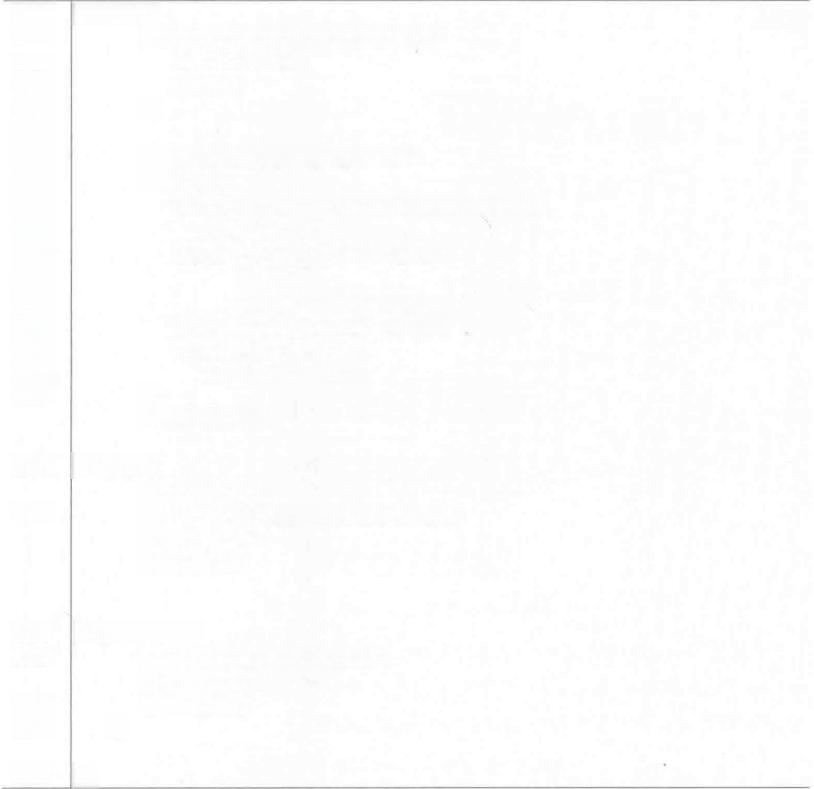
在这里首先会分配缓存页，然后跳到第2166行代码读取数据。读取数据调用 的是具体文件系统注册的 readpage()函数，对于 Ext2 文件系统来说就是 ext2\_readpage() 函数，该函数的功能就是从后端磁盘读取一页的内容。

**代** **码** **4** **-** **1** **7** **随** **机** **读** **场** **景** **的** **代** **码** **实** **现**

**mm/filemap.c**

|  |  |
| --- | --- |
| 1992  1992  1993  2016 | ssize\_t generic\_file\_buffered\_read(struct kiocb \*iocb,  struct iov\_iter \*iter,ssize\_t written)  {  //删除部分代码 for(;){  //删除部分代码 |

第4章从理论到实战——Ext2文件系统代码详解

2023

2024

2025

2026

2027

2028

2029

2030

2031

2032

2033

2034

2035

2036

2037

2038

2039

2166

2179

2218

2223

2238

2239

2251

find\_page:

if(fatal\_signal\_pending(current)){

error=-EINTR; goto out;

}

page=find get\_page(mapping,index);

if(lpage){

if(iocb->ki\_flags &(IOCB\_NOWAIT|IOCB\_NOIO))

goto would\_block;

page\_cache\_sync\_readahead(mapping,

ra,filp,

index,last\_index-index);

page=find\_get page(mapping,index); if(unlikely(page=NULL))

goto no\_cached\_page; 删除部分代码

readpage:

//删除部分代码

error=mapping->a\_ops->readpage(filp,page);

//删除部分代码 goto page\_ok;

no\_cached\_page:

page=page\_cache\_alloc(mapping);

/删除部分代码

goto readpage;



上

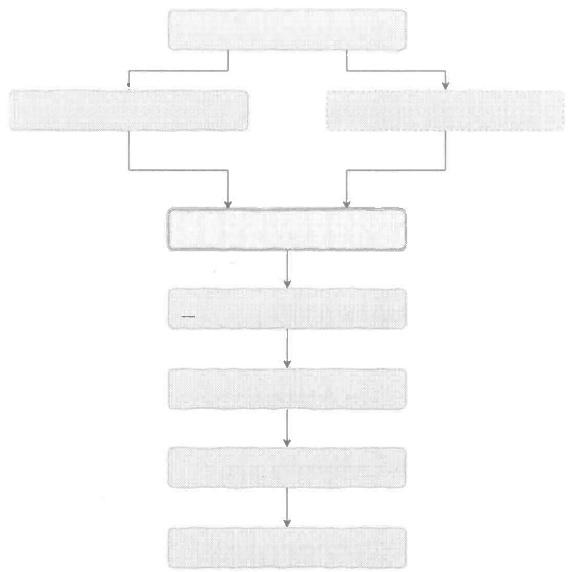
完成数据读取后的逻辑与缓存命中没有差别，也就是将数据从缓存页拷贝到用 户空间等，最终返回用户态。

**4.8.3** **数据预读逻辑**

第3章已经介绍过了关于预读的原理，本节主要针对 VFS 的代码介绍一下 Linux 中预读的具体实现。在VFS 中实现了两种预读， 一 种是同步预读；另 一 种是 异步预读。两者实现的功能略有不同，但它们通过一个公共的函数 ondemand\_readahead() 实现了具体的功能，而且预读算法也是在该函数中实现的。 同步预读与异步预读的核心流程如图4 - 38所示。

文件系统技术内幕

大数据时代海量数据存储之道



generic\_file\_buffered\_read()



page\_cache\_sync\_readahead() page\_cache\_async\_readahead()

ondemand\_readahead()

\_do\_page\_cache\_readahead()

read\_pages()

ext2\_readahead()

mpage\_readahead()

图4-38 同步预读与异步预读的核心流程

预读操作主要是通过一个名为file\_ra\_state 结构体来控制的，该结构体的定义 如代码4-18所示，具体成员的含义请参考其中的注释。这个数据结构会在后面预 读算法中使用，请结合后面的解释来理解该结构体。

**代码4-18** **file** **ra** **state结构体的定义**

**include/linux/fs.h**

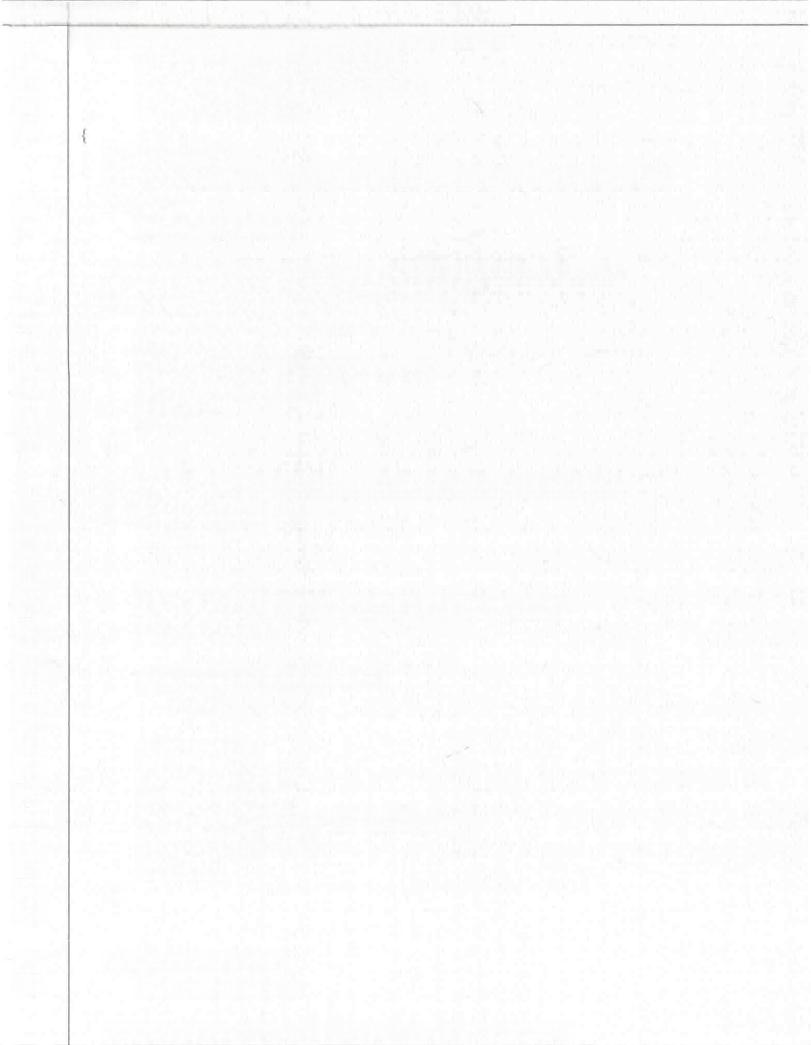
|  |  |  |
| --- | --- | --- |
| 924 | struct file\_ra\_state{ |  |
| 925 | pgoff\_t start; | //预读开始的位置 |
| 926 | unsigned intsize; | //预读的页数 |
| 927 | unsigned int async\_size; | //异步预读的触发条件，当剩余页等于该数值时触发 |
| 928 |  |  |
| 929 |  |  |
| 930 | unsigned int ra\_pages; | //预读窗口，用于设置缓存页异步预读标记 |
| 931 | unsigned int mmap\_miss; | //针对mmap访问缓存非命中的次数 |
| 932 | loff\_tprev\_pos; | //上次读缓存的位置 |
| 933 | }; |  |

在Linux 中，同步预读的触发有两种场景： 一种是从文件开头读取数据时；另 一种是识别连续I/O 时。而异步预读则是读到有预读标记的页面(PG\_readahead page) 时，才触发异步预读。

第4章从理论到实战 — — Ext2文件系统代码详解

代码4- 19是预读算法的核心代码，包括同步预读和异步预读。我们先从同步 预读进行介绍。

**代码4-19** **预读算法的核心代码**

**mm/readahead.c**

440

441

442

443

444

460

461

467

468

469

470

471

472

473

481

482

483

484

485

486

487

488

489

490

491

492

493

494

495

496

497

498

502

503

static voidondemand\_readahead(struct address\_space \*mapping,

struct file\_ra\_state \*ra,struct file\*filp,

bool hit\_readahead\_marker,pgoff\_t index, unsigned long req\_size)

//删除部分代码

//判断请求是否从文件开始的位置，如果是从文件开始的位置则进行同步预读 if(lindex)

goto initial\_readahead;

//根据上次预读起始位置和页数及当前请求的偏移，判断是否为顺序读 if(index=(ra->start+ra->size-ra->async\_size)Ⅱ

index=(ra->start+ra->size))){

ra->start +=ra->size;

ra->size=get\_next\_ra\_size(ra,max\_pages); ra->async\_size=ra->size;

goto readit;



//根据参数判断是否命中的有预读标记的缓存页，这是异步预读的标记

if(hit\_readahead\_marker){

pgoff\_t start;

rcu\_read\_lock);

start=page\_cache\_next\_miss(mapping,index+1,max\_pages); rcu\_read\_unlock();

if(!start l start-index>max\_pages)

return;

ra->start=start;

ra->size=start -index; ra->size+=req\_size;

ra->size=get\_next\_ra\_size(ra,max\_pages); ra->async\_size=ra->size;

goto readit;

}

if(req\_size>max\_pages)

goto initial\_readahead;

//如果是连续读，且之前没有缓存的情况，则再次触发同步预读

**文件系统技术内幕**

大数据时代海量数据存储之道

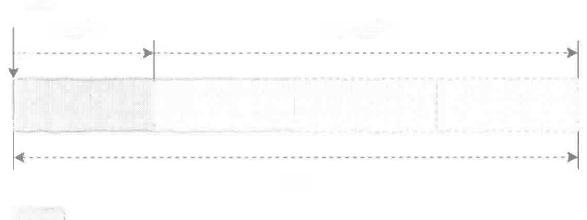
|  |  |
| --- | --- |
| 510  511  512  525  526  528  529  530  531  532  533  540  541  542  543  544  545  546  547  548  549  550  551  552 | prev\_index=(unsigned long long)ra->prev pos>>PAGE\_SHIFT; if(index-prev\_index<=1UL)  goto initial\_readahead; //删除部分代码  \_do\_page\_cache\_readahead(mapping,filp,index,req\_size,0); return;  //更新预读窗口的相关信息 initial\_readahead:  ra->start=index;  ra->size=get\_init\_ra\_size(req\_size,max\_pages);//预读窗口更新算法 ra->async\_size=ra->size>req\_size ?ra->size-req\_size:ra->size;  readit://判断当前请求是否会命中预读标记，如果命中则需要更新预读窗口  if (index=ra->start &&ra->size=ra->async\_size){  add\_pages=get\_next\_ra\_size(ra,max\_pages); if(ra->size+add\_pages<=max\_pages){  ra->async\_size=add\_pages; ra->size+=add\_pages;  }else {//如果预读数据的页数超过最大页数则限定为最大页数 ra->size=max\_pages;  ra->async\_size=max\_pages>>1; }  }  ra\_submit(ra,mapping,filp); //根据预读窗口信息提交读数据的请求  } |

首先，从文件起始位置读的场景，根据请求偏移就可以判断出来，如第460行 代码所示。此时代码会跳转到initial\_readahead 的位置，这里用于更新预读窗口信 息，包括预读的起始位置、大小和设置预读标记的位置。然后，判断读请求结束的 位置是否命中了将要设置预读标记的位置，如果命中则需要重新调整预读窗口(请 思考一下为什么要重新调整预读窗口?)。最后，调用ra\_submit(函数进行数据读 取，根据预读窗口的信息为某个页设置预读标记。需要注意的是，这里的“页”是 指存储数据的容器。

为了能够理解上述逻辑，我们列举一个具体的实例。假设当前请求从文件开始 的位置读取数据，且读取1个页的数据。此时预读窗口将被初始化为4个页的大 小，预读标记的位置是倒数第3个(4-1)页的位置，如图4-39所示。

**第4章** 从理论到实战——Ext2文件系统代码详解



start

req\_size

-

async\_size

-



size

当前请求的大小

★ 有预读标记的位置

图4-39 从文件头读取1个页的数据时预读窗口信息

如果当前请求的大小变大(如2个页或3个页),则预读窗口的大小和预读标 记的位置都会根据请求的大小来计算。

同步预读的另一种场景是连续读，该请求不是从文件头开始的，而是与前一个 请求衔接的。该部分的判断在代码4-19的第510行～第512行。如果符合连续读 的条件，则会跳转到initial\_readahead 位置进行预读窗口的初始化及后续操作。

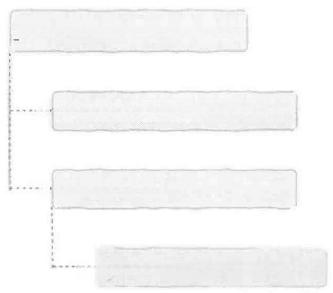
对于异步预读，情况要简单一些。由于在预读流程中会为某个页设置预读标记， 因此当读请求读到该页时就会发起一个异步预读，进而调用ondemand\_readahead() 函数。对于异步预读，在调用ondemand\_readahead() 函数时参数hit\_readahead\_marker 的值为真，因此会执行第481～第497行的代码。可以看出该部分代码主要实现对 预读窗口的调整，之后就跳转到读数据的流程。

前文介绍了预读算法的实现，但并没有介绍数据具体是如何从磁盘读取的。读 取数据的操作最终是在 do\_page\_cache\_readahead(函数中实现的。这里面主要完 成两个功能：一个是分配页缓存；另一个是调用具体文件系统读取数据的接口。

具体到Ext2 文件系统，图4-40中的函数指针为ext2\_readahead (函数，该函数 的调用流程如图4-41所示，该函数主要实现缓存页的映射和从磁盘读取数据的操 作，最后调用块设备层的submit\_bio() 函数完成数据的读取。

文件系统技术内幕

大数据时代海量数据存储之道



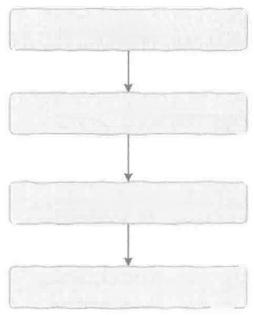
\_do\_page\_cache\_readahead()

page\_cache\_alloc\_readahead()

read\_pages()

→a\_ops->readahead/readpages()

图4-40 读数据缓存基本逻辑



ext2\_readahead()

mpage\_readahead()

mpage\_bio\_submit()

submit\_bio()

图4-41 ext2\_readahead(函数的调用流程

至此，就将磁盘上的数据读取到了页缓存中，其后的流程与缓存命中场景的流 程一致，即拷贝页缓存的内容到用户态缓冲区等。

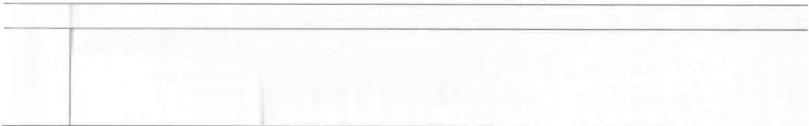
**4.9** **如何分配磁盘空间**

前章节已经介绍了写数据的主要流程，但没有介绍数据如何写到磁盘，以及磁 盘的空间是如何分配的等内容。本节介绍一下Ext2 文件系统分配磁盘空间的相关 逻辑。

实际上，在向磁盘写数据之前需要分配磁盘空间，也就是告诉文件系统数据应 该写在磁盘的什么位置。这里的写数据包括写文件数据、在目录中创建文件和添加 扩展属性等。但凡需要存储新数据的场景都需要分配磁盘空间。分配磁盘空间的主 要功能在ext2\_get\_blocks (函数中实现，该函数的原型如代码4-20所示。

第4章从理论到实战 — — Ext2文件系统代码详解

**代码4-20** **ext2\_get\_blocks()函数的原型**



**fs/ext2/inode.c**

624 staticint ext2\_get\_blocks(struct inode \*inode,

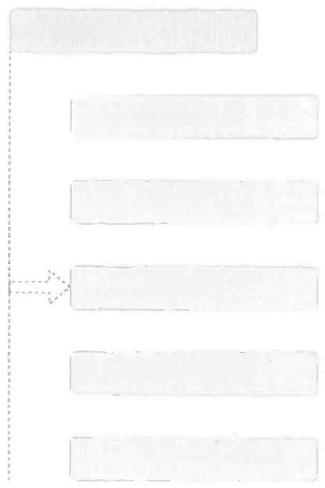
625 sector\_t iblock,unsigned long maxblocks,

626 u32 \*bno,bool\*new,bool\*boundary,

627 int create)

在 ext2\_get\_blocks() 函数原型中，需要重点说明的是 iblock 参数，该参数表示 文件的逻辑位置，位置以文件系统的块大小为单位，以0为起始位置逻辑地址。列 举一个简单的实例，假如文件系统在格式化时块大小是2KB, 而此时写入数据的偏 移 为 4KB, 那 么 此 时iblock 的值是2。也就是说，ext2\_get\_blocks() 函数通过数据在 文件中的逻辑位置计算需要分配多少磁盘空间。

使 用ext2\_get\_blocks() 函数进行磁盘空间分配的主流程如图4-42所示，该函数 主要完成以下3个方面的工作。



ext2\_get\_blocks()

ext2\_block to path()

ext2\_get\_branch)

ext2\_find\_goal()

ext2\_blks to allocate()

ext2\_alloc\_branch()

图4-42 使用ext2\_get\_blocks(函数进行磁盘空间分配的主流程

(1)计算并获取存储路径。我们知道文件数据是通过间接块的方式存储的，因 此这里主要是根据数据逻辑地址计算出其存储路径情况，也就是会经过哪个间接块 及在间接块中的偏移。

(2)计算需要分配的空间。由于某些间接块可能已经被分配，还有一些间接块 没有被分配，因此需要根据分配的实际情况计算出需要分配的间接块。

文件系统技术内幕

大数据时代海量数据存储之道

(3)分配磁盘空间。根据上一步计算出的磁盘空间，调用ext2\_alloc\_branch ( 函 数来分配需要的磁盘空间，具体就是将空间管理的位图置位。

为了使大家更容易理解整个分配磁盘空间的流程，先回顾一下Ext2 文件系统 中文件数据的管理方式，也就是间接块的管理方式(见图3-28)。

由于索引树的结构是固定的，因此根据请求的逻辑地址和大小，就可以计算出 间接块和数据块的数量，也就知道了本次请求需要申请的磁盘空间数量。最后根据 这些信息来分配具体的磁盘空间。下面分别详细介绍一下各个流程的实现细节。

**4.9.1** **计算存储路径**

计算存储路径是指根据请求的文件逻辑地址计算所涉及的间接块及所在间接 块中的偏移。该功能是由ext2\_block\_to\_path() 函数完成的，在该函数中数组offsets 用于存储每一级的具体偏移位置。

前文已经提到Ext2 文件系统根据文件大小的不同，采用不同级别的间接块来 管理文件数据。如果文件很小，则采用直接块(0级间接块),然后是1级间接块、 2级间接块和3级间接块。因此ext2\_block\_to\_path() 函数在具体实现时也是按照这 4种场景实现了4个分支，如代码4-21所示。

**代码4-21** **ext2** **block** **to** **path()函数**

**\_** **\_** **\_**

**fs/ext2/inode.c**

|  |  |
| --- | --- |
| 164 | staticintext2\_block\_to\_path(struct inode \*inode, |
| 165 | longi\_block,int offsets[4],int \*boundary) |
| 166 | {  //根据超级块信息获得每个逻辑块存储地址的数量 |
| 167 | int ptrs=EXT2\_ADDR\_PER\_BLOCK(inode->i\_sb); |
| 168 | int ptrs\_bits=EXT2\_ADDR\_PER\_BLOCK\_BITS(inode->i\_sb); |
| 169 | const long direct\_blocks=EXT2\_NDIR\_BLOCKS, |
| 170 | indirect\_blocks=ptrs, //对于IKB的逻辑块，1级间接块可以管理256个地址 |
| 171 | double\_blocks=(1<<(ptrs\_bits\*2)); //2级间接块可以管理65536个地址 |
| 172 | int n=0; |
| 173 | int final=0; |
| 174 |  |
| 175 | if(i\_block<0){ |
| 176 | ext2\_msg(inode->i\_sb,KERN\_WARNING, |
| 177 | "warning:%s:block<0",\_func\_; |
| 178 | }else if(i\_block<direct\_blocks){ //直接块场景 |
| 179 | offsets[n++]=i\_block; |
| 180 | final=direct\_blocks; |
| 181 | }else if((i\_block -=direct\_blocks)<indirect\_blocks){ //1级间接块场景 |
| 182 | offsets[n++]=EXT2\_IND\_BLOCK; |
| 183 | offsets[n++]=i\_block; |
| 184 | final=ptrs; |

第4章从理论到实战 — — Ext2文件系统代码详解

}else if(i\_block=indirect\_blocks)<double\_blocks){ //2 级间接块场景

186 offsets[n++]=EXT2\_DIND\_BLOCK;

187 offsets[n++]=i\_block>>ptrs\_bits;

188 offsets[n++]=i\_block&(ptrs-1);

189 final=ptrs;

else if((i\_block-=double\_blocks)>>(ptrs\_bits\*2))<ptrs){//3级间接块场景

191 offsets[n++]=EXT2\_TIND\_BLOCK; offsets[n++]=i\_block>>(ptrs\_bits\*2);

193 offsets[n++]=(i\_block>>ptrs\_bits)&(ptrs-1);

194 offsets[n++]=i\_block &(ptrs-1);

195 final=ptrs; else(

ext2\_msg(inode->i\_sb,KERN\_WARNING,

198 "warning:%s:block is too big",\_func\_);

199 

if(boundary)

201 \*boundary=final-1-(i\_block&(ptrs-1);

203 return n; 204

上述代码是按照逻辑地址来确定执行哪个分支的。为了便于理解上述代码，列 举一个具体的实例。以逻辑块大小1KB 为例，在Ext2文件系统中地址为4字节， 因此一个间接块可以存储256个地址。假设写数据的位置是13KB, 当不满足第178 行代码的条件时，会继续执行第181行的代码进行条件判断。因此，会执行该分支 中 的 代 码 ， 最 后offset[0] 的 值 为 1 2 ,offset[1] 的 值 为 1 。

我们再举一个复杂点的实例，假设写数据的位置是65808KB。此时满足第190 行代码的条件，因此当满足第190行代码的条件时，会执行该分支中的代码。也就 是该位置在3级间接块子树中。接下来看一下在该子树中各级间接块的值是如何计 算 的 。

**1** **.** **1级间接块偏移值为0**

由于执行到这里时，前面的判断代码(第181行，第185行)都会执行一次， 因此逻辑地址会分别减去直接块管理的数量、1级间接块最大管理数量和2级间接 块最大管理数量，在这里i\_block的值为65808-12-256-256×256=4。

由于3级间接块可以管理256个逻辑块，2级间接块可以管理256个3级间接 块，因此1级间接块的一个地址可以管理65536(256×256,也就是1<16)个逻 辑块。由此可以知道第192行的代码可以转换为如下内容，也就是用新逻辑块的地 址(4)除以65536得到。

(4)>(8×2)=0

**文件系统技术内幕**

大数据时代海量数据存储之道

**2.2级间接块偏移值为0**

2级间接块偏移值的计算与1级间接块偏移值的计算类似。对于2级间接块中 的偏移，除以256后对256取模即可得到。

(4>8)&0xFF=0

**3.3级间接块偏移值为4**

对于3级间接块，直接对256取模即可得到偏移值。

**4&0xFF=4**

这里需要注意的是，除了返回深度和每一层的位置，还会返回在最后的间接块 上可管理的地址数量。比如，计算出在最后1级间接块的位置是250,那么最多可 以管理6个地址。在这种情况下，如果申请的空间比较多，则会出现跨3级间接块 的场景。

**4.9.2** **获取存储路径**

上文计算出了深度和每一级间接块的偏移信息，但具体涉及的间接块目前处于 什么状态并不清楚。仍然以上面的实例进行说明，可能会出现以下几种情况。

(1)用户访问的数据位置所需要的间接块已经全部分配。

(2)1级间接块和2级间接块已存在，3级间接块不存在。

(3)1级间接块已存在，2级间接块和3级间接块不存在。

(4)所有间接块都不存在。

因此，这一步的工作就是根据当前信息及上一步计算出的信息进行综合判断， 确定已经具备的间接块，并返回关键信息，为后续流程分配磁盘空间做准备。可以 在ext2\_get\_branch (函数中实现，如代码4-22所示。

**代** **码** **4** **-** **2** **2** **ex** **t** **2** **\_** **g** **e** **t** **\_** **b** **r** **a** **n** **ch** **(** **)** **函** **数**

fs/ext2/inode.c

|  |  |
| --- | --- |
| 235 | static Indirect\*ext2\_get\_branch(structinode\*inode, |
| 236 | int depth, |
| 237 | int \*offsets, |
| 238 | Indirect chain[4], |
| 239 | int\*err) |
| 240 |  |
| 241 | struct super\_block \*sb=inode->i\_sb; |
| 242 | Indirect\*p=chain; |
| 243 | struct buffer\_head \*bh; |
| 244 |  |

第4章从理论到实战—— Ext2文件系统代码详解

\*err=0;

//根据inode 索引树根，初始化0级间接块

247 add\_chain(chain,NULL,EXT2\_I(inode)->i\_data+\*offsets); if(lp->key)/

goto no\_block; while(--depth)

bh=sb\_bread(sb,le32\_to\_cpu(p->key);// 根据上 一 级间接块中的地址读取信息

252 if(!bh)

goto failure;

read\_lock(&EXT2\_I(inode)->i\_meta lock);

if(!verify\_chain(chain,p))

256 goto changed;

add\_chain(++p,bh,(\_le32\*)bh->b\_data+\*++offsets); read\_unlock(&EXT2\_I(inode)->i\_meta\_lock);

if(lp->key) //如果地址的值为0,则表示下 一 级间接块没有被分配 goto no\_block;

261 

return NULL; //如果所有间接块都具备，则返回空指针

263

changed:

read\_unlock(&EXT2\_I(inode)->i\_meta lock);

\*err=-EAGAIN;

goto no\_block; failure:

err=-EIO;

no block:

\_

272 return p; 某些间接块不具备的情况



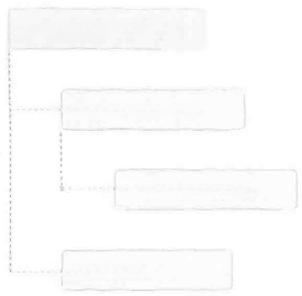
在ext2\_get\_branch() 函数中会逐级对间接块进行初始化，然后根据已经初始化 的间接块中的地址从缓存或磁盘读取下一级间接块的信息。如果地址为空，则表示 下一级间接块没有被分配，此时将会跳出while 循环。

**4.9.3** **分配磁盘空间**

完成间接块情况分析之后，再经过简单的计算，就可以计算出总共需要分配的 磁盘空间的数量。然后就可以使用 ext2\_alloc\_branch() 函数分配磁盘空间了，该函 数主要调用了其他两个函数，如图4-43所示。其中，ext2\_alloc\_blocks() 函数用于分 配磁盘空间，本质是将管理磁盘空间的位图的对应位进行置位操作；另外， sb\_getblk() 函数用于从磁盘读取该块的数据，并进行初始化。

文件系统技术内幕

大数据时代海量数据存储之道



ext2\_alloc\_branch()

ext2\_alloc\_blocks()

ext2\_new\_blocks()

sb\_getblk()

图4-43 使用ext2\_alloc\_branch()函数分配磁盘空间的流程

ext2\_alloc\_blocks (函数用于分配磁盘空间，该函数调用ext2\_new\_blocks() 函数 进行分配磁盘空间的具体操作。后者的逻辑也是比较清晰的，主要是读取组描述符 和位图信息，然后根据位图信息确定可分配的磁盘空间，并进行分配和更新位图。

sb\_getblk() 函数初始化的目的比较明确，因为间接块用来存储地址信息，如果 是从磁盘读取的新间接块数据可能是未知值，因此需要进行清零操作，并且完成本 次请求地址的初始化操作。

至此，磁盘空间分配的主要流程执行完成，仍然有一些小的处理流程，如更新 inode 中的记录、最后一次分配位置、更新时间和将inode 变脏等，这些细节读者可 以自行阅读代码理解。

**4.10** **Ext2** **文件系统的扩展属性**

前面章节已经介绍过关于文件系统扩展属性的概念及应用，本节不再赘述。本 节主要结合Ext2 文件系统的实现代码介绍一下扩展属性是如何实现的。

**4.10.1** **Ext2文件系统扩展属性是怎么在磁盘存储的**

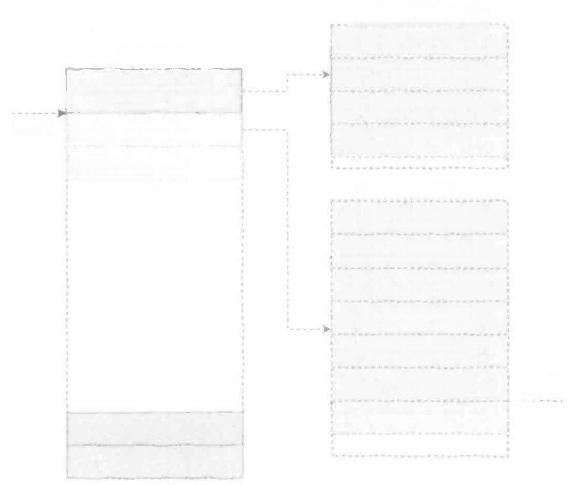
本节主要介绍一下Ext2 文件系统中扩展属性的相关内容，包括磁盘数据布局 和创建流程等。在Ext2 文件系统中，扩展属性存储在一个单独的磁盘逻辑块中， 其位置由 inode 中 的i\_file\_acl 成员指定。

图4-44所示为扩展属性“键-值”对在磁盘逻辑块中的布局示意图。前32字 节是一个描述头(ext2\_xattr\_header), 描述磁盘逻辑块的基本信息。而下面紧跟着 的是扩展属性项(ext2\_xattr\_entry), 描述了扩展属性的键名称等信息，同时包含值

**第** **4** **章** 从理论到实战 — —Ext2 文件系统代码详解



的偏移信息等内容。从图4-44中可以看出，扩展属性项是一字排开的。而且需要 注意的是，扩展属性项的值是从上往下生长的，而扩展属性的值则是从下往上生长。

0x20

disk block

ext2\_xattr\_header

ext2\_xattr\_entry

ext2\_xattr\_entry

value value

ext2\_xattr header h\_magic

h\_refcount h\_blocks h\_hash

ext2\_xattr\_entry

e\_name\_len

e\_name\_index

e\_value\_offs

e\_value\_block

e\_value\_size

e\_hash

e name

0x10

-

图4-44 扩展属性“键-值”对在磁盘逻辑块中的布局示意图

代码4-23是描述头(ext2\_xattr\_header) 的结构体定义，包括魔数、引用计数 和哈希值等内容 。 魔数的作用是确认该逻辑块的内容是扩展属性逻辑块 ， 避免代码 Bug 或 者 磁 盘 损 坏 等 情 况 下 给 用 户 返 回 错 误 的 结 果 。 引 用 计 数 和 哈 希 值 的 作 用 是 实 现多个文件的扩展属性共享 。 扩展属性共享是指在多个文件的扩展属性完全一样的 情况下 ， 这些文件的扩展属性将采用相同的磁盘逻辑块存储 ， 这样可以极大地节省 存 储 空 间 。 另 外 ，Ext2 文 件 系 统 使 用 哈 希 缓 存 存 储 文 件 属 性 的 哈 希 值 ， 用 于 快 速 判 断文件是否存在相同的扩展属性逻辑块 。

**代码4-23描述头(ext2** **xattr** **header)的结构体定义**

fs/ext2/xattr.h

|  |  |  |  |
| --- | --- | --- | --- |
| 27 | struct ext2\_xattr\_header{ | |  |
| 28 | le32 | h\_magic; | //用于识别的魔数 |
| 29 | le32 | h\_refcount; | //引用计数 |
| 30 | le32 | h\_blocks; | //使用的磁盘块的数量 |
| 31 | \_le32 | h\_hash; | //所有属性的哈希值 |
| 32 | u32 | h\_reserved[4]; | / 当 前 值 为 0 |
| 33 |  |  |  |

文件系统技术内幕

大数据时代海量数据存储之道

扩展属性项在磁盘上是从上往下生长的，需要注意的是由于每个扩展属性的键 名称的长度不一定相同，因此描述头(ext2\_xattr\_header) 结构体的大小也是变化的。 由于上述原因，我们无法直接找到某一个扩展属性项的位置，必须从头到尾进行遍 历。由于描述头(ext2\_xattr\_header) 的大小是确定的，这样就可以很容易找到第1 个扩展属性项，而下一个扩展属性项就可以根据已经找到的扩展属性项的位置及其 中的e\_name\_len 成员计算得到。

**代** **码** **4** **-** **2** **4** **扩** **展** **属** **性** **项** **结** **构** **定** **义**

fs/ext2/file.c

|  |  |  |
| --- | --- | --- |
| 35 | struct ext2\_xattr\_entry{ |  |
| 36 | \_u8 e\_name\_len; | //名称长度 |
| 37 | u8 e\_name\_index; | //属性名称索引 |
| 38 | le16 e\_value\_offs; | //值在磁盘块中的偏移 |
| 39 | le32 e\_value\_block; | //属性所在的磁盘块 |
| 40 | le32 e\_value\_size; | //属性值的大小 |
| 41 | le32 e\_hash; | //名称和值的哈希值 |
| 42 | char e\_name[]; | //属性名称 |
| 43 |  |  |

下面列举一个实例看一下扩展属性是如何存储在磁盘上的。我们首先可以获取

fl.txt 对应的inode 信息，如图4-45所示。从该信息中可以得到i\_file\_acl 的值为 0x00000406。

rootesunnyzhang:-/test/ext2#hexdump -n 128-s 268672 ext2\_ 1kb.binv-C

00041980 a481.00000800 B0008C 063160 a2823f 60L 1..?1

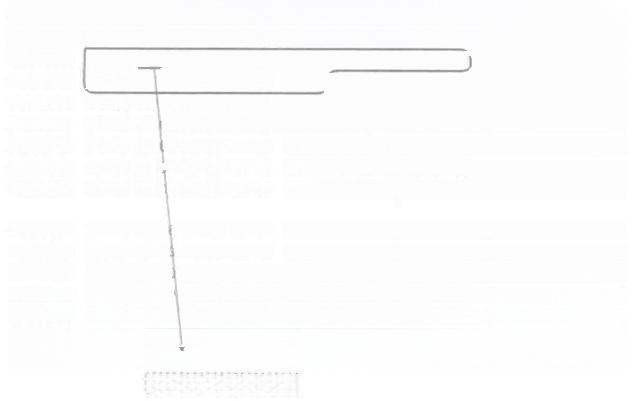
|  |  |
| --- | --- |
| 00041990 Bc10631600000000000000100040000001..1 |  |
| 000419a0 00000000010000000104.000000000000 L | l |
| 000419b0000000000000.0000900000800000 B000 L | 1 |
| 000419c00000800000000000000e 008000.000080 l | 1 |
| 000419do 000000000000000000000080000000.001 | … |

000419e0000000007b fe 67510604000000000000 .1.g0……1 000419f000000000000000800000000000.000000I …

图4-45 扩展属性的地址

通过该物理地址，我们可以读取磁盘上的数据，如图4-46所示。这里扩展属 性逻辑块头会占用32字节。之后是扩展属性项的内容(红线圈起来的内容)。在上 述内容中，标注的内容(0x03f4) 是扩展属性值在本逻辑块中的偏移位置。通过该 逻辑块的基地址和偏移地址就可以指定该扩展属性值的位置，本实例为0x101bf4。

第4章从理论到实战 — — Ext2文件系统代码详解

rootasunnyzhang:-/test/ext2e hexdump -n 1024-51054720 ext2\_ 1kb.bin-v-0

0010180000.00102 ea 0100000001000000 a403c99c I …1 00101810000000000000000000000000000000001… …I

00101820

00101830

00101840

00101850

00101860

00101870

00101880

80101b90 00101ba0 00101b60

0a 01 f403000000000a 000000 a403 c9991 .I

73756e 6e 797a 6861 6e 67000000000000 |sunnyzhang.

|  |  |
| --- | --- |
| 000000 l00000000000000000000000000 L  800000 pe 0000000000000000000000.00|..…  0000|00 00 00 00 0000 008000000000.00001  000000  00 00 0000 0000000000000090 l  000000 do 000000.00 00009000 B0000000 L  00.000000 00 00 0000 0000000000009000  00.00100 映 0000/0000 00000000.00000000 L 00000000 00000000901000000001000000 L. | ….I …I …1  …  I l l |

00101bc0 00000000 0000000000100000000000000 ..I

00101bde 00000000000000009000000000000000 1. |

00101be080000000l0000.000000000.0000000000| …I

001016f000000006974776f 726c 64.3132330000 L.itworld123.1

101bf4=101bfO+0314

**图4-46** **扩展属性的内容**

通过关键的字符串可以看出，这些内容正是我们设置的扩展属性。

**4.10.2 设置扩展属性的VFS流程**

操作系统提供了 一 些函数来设置文件的扩展属性，分别是setxattr() 、fsetxattrO

和 lsetxattr() 。 这几个函数的应用场景略有差异，但功能基本 一致。代码4- 25所示 为上述函数的原型，可以看出其核心参数是 一样的，参数意义很明确，本节不再 赘述。

**代码4-25** **设置文件的扩展属性的函数的原型**

int setxattr(const char \*path,const char \*name,

const void \*value,size\_t size,int flags); int lsetxattr(const char \*path,const char\*name,

const void\*value,size\_t size,intflags); int fsetxattr(int fd,const char\*name,

const void\*value,size tsize,int flags);

本节以 fsetxattr() 函数为例进行介绍。假设用户调用该函数为某个文件设置user 前缀的扩展属性，此时整个函数调用栈的流程如图4-47所示。本调用栈包含三部 分内容，分别是用户态接口、VFS 调 用 栈 和Ext2 文件系统调用栈。

文件系统技术内幕

大数据时代海量数据存储之道

fsetxattr()

用户态接口

sys\_fsetxattr(s/xattrc)

setxattr()

vfs\_setxatr()

\_vfs\_setxattr\_noperm()

VFS 调用栈

\_vfs\_setxattr()

xattr\_resolve\_name()

handler->set()

ext2\_xattr\_user\_set()

Ext2文件系统调用栈

ext2\_xatt\_set()

图4-47 fsetxattr()函数调用栈的流程

通过图4-47可以看出在VFS 中做了很多事情，最后通过函数指针的方式调用 Ext2 文件系统的扩展属性设置接口。在这个流程中比较重要是 vfs\_setxattr() 函数， 在该函数内部根据扩展属性名称前缀获取句柄指针(见代码4-26第143行),然后 利用该句柄指针进行具体的处理。相当于在\_vfs\_setxattr() 函数中会根据不同的扩 展属性执行不同的处理分支。

**代码4-** **26** **vfs\_** **setxattr()函数**

**fs/xattr.c**

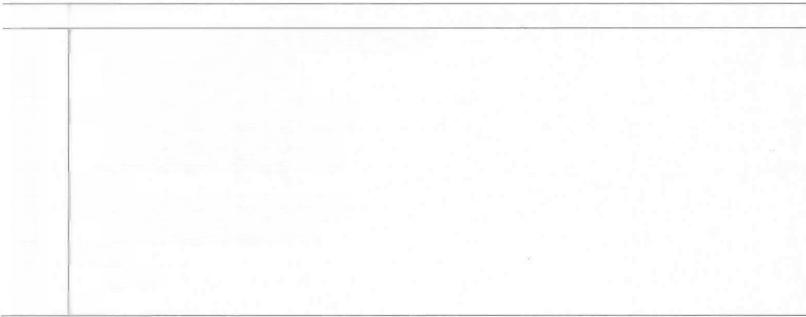
|  |  |
| --- | --- |
| 137 | int |
| 138 | \_vfs\_setxattr(struct dentry \*dentry,struct inode \*inode,const char \*name, |
| 139 | const void \*value,size\_t size,int flags) |
| 140 | { |
| 141 | const struct xattr\_handler \*handler; |
| 142 |  |
| 143 | handler=xattr\_resolve\_name(inode,&name); 根据扩展属性名称前缀获取句柄指针 |
| 144 | if(IS\_ERR(handler)) |
| 145 | return PTR\_ERR(handler); |
| 146 | if(!handler->set) |
| 147 | return -EOPNOTSUPP; |
| 148 | if(size=0) |

第 4 章 从 理 论 到 实 战 — — E x t 2 文 件 系 统 代 码 详 解

149 value="";

150 return handler->set(handler,dentry,inode,name,value,size,flags);





151

对于Ext2 文件系统来说，这个句柄指针的定义如代码4-27所示，可以看出它 就是各种不同类型(如user、trusted和 system 等)扩展属性的数组。这个数组在 Ext2 文件系统挂载时会初始化到超级块数据结构中。因此，在xattr\_resolve\_name() 函数中通过扩展属性名称前缀就可以找到对应的句柄指针。

**代码4-27** **句柄指针的定义**

fs/ext2/xattr.c

113 const struct xattr\_handler \*ext2\_xattr\_handlers[]={ 114 &ext2\_xattr\_user\_handler,

115 &ext2\_xattr\_trusted\_handler,

116 #ifdef CONFIG\_EXT2\_FS\_POSIX\_ACL 117 &posix\_acl\_access\_xattr\_handler,

118 &posix\_acl\_default\_xattr\_handler, 119 #endif

120 #ifdef CONFIG\_EXT2\_FS\_SECURITY 121 &ext2\_xattr\_security\_handler,

122 #endif

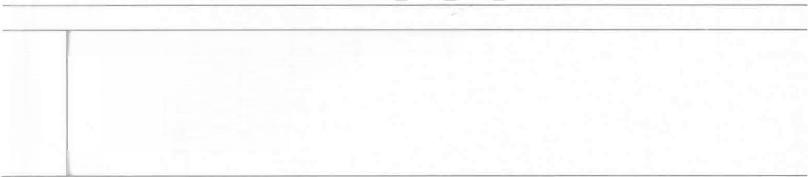
123 NULL

124 ;

对于Ext2 文件系统的user 扩展属性来说，会定位到ext2\_xattr\_user\_handler 句 柄指针，而在该句柄指针中定义了对Ext2文件系统扩展属性查询和设置的接口。

**4.10.3** **Ext2文件系统扩展属性接口实现**

对于user 类型的扩展属性，其函数集为ext2\_xattr\_user\_handler, 其定义如代码 4-28所示。这里面实现了该类型扩展属性的查询和设置等接口。



**代码4-28** **函数集为ext2\_xattr\_user\_handler** **的定义**

**fs/ext2/file.c**

44 const struct xattr\_handler ext2\_xattr\_user\_handler={ 45 .prefix=XATTR\_USER\_PREFIX,

46 .list=ext2\_xattr\_user\_list,

47 .get=ext2\_xattr\_user get, 48 .set=ext2\_xatr\_user\_set, 49 );

上述代码中的各个函数的意义很明确，可以通过函数名称知道其具体的功能。

通过使用ext2\_xattr\_user\_set (函数设置扩展属性的接口，如代码4-29所示，从 代 码 中 可 以 看 出 该 函 数 主 要 调 用 了ext2\_xattr\_set() 函 数 ， 这 个 函 数 实 现 了 对 扩 展 属 性的增加 、 删除和修改操作 。

文件系统技术内幕

大数据时代海量数据存储之道

代码4-29使用ext2 xattr user set()函数设置扩展属性的接口

**fs/ext2/xattr\_user.e**

|  |  |
| --- | --- |
| 31 | static int |
| 32 | ext2\_xattr\_user\_set(const struct xattr\_handler \*handler, |
| 33 | struct dentry \*unused,struct inode \*inode, |
| 34 | const char \*name,const void \*value, |
| 35 | size\_t size,int flags) |
| 36 |  |
| 37 | if (Itest\_opt(inode->i\_sb,XATTR\_USER)) |
| 38 | return-EOPNOTSUPP; |
| 39 |  |
| 40 | return ext2\_xattr\_set(inode,EXT2\_XATTR INDEX\_USER, |
| 41 | name,value,size,flags); |
| 42 | 1 |
| 具体操作的类型依赖value 参数和同名属性的存在情况。如果value 的值为空， 则表示要删除这个扩展属性。如果当前没有同名的扩展属性，且value 的值不为空， 则创建一个新的扩展属性。如果有同名扩展属性，并且 value的值不为空，则对现 有的扩展属性进行更新。  ext2\_xattr\_set (函数的实现非常长，大概有300多行代码，为了减少篇幅，本 节不会介绍所有场景，以更新一个扩展属性为例进行介绍。在理解该函数之前，应 该先对4.10.1节介绍的 Ext2 文件系统扩展属性的磁盘布局有所了解，这样理解起 来就比较简单了。  对于更新扩展属性的场景，就是找到该扩展属性和对应的值，然后在原地更新 值，或者移除原始的值，添加新值。至于在值的原地更新还是移除后添加，依赖于 新值的长度，如代码4-30所示。  **代码4-30** **ext2\_xattr\_set()函数** | |
| fs/ext2/xattr.c | |
| 406  407  408  409  441  442  443  450  451  465  466 | int  ext2\_xattr\_set(struct inode \*inode,int name\_index,const char\*name, const void \*value,size\_t value\_len,int flags)  {  //删除部分代码  /对于更新扩展属性的场景，已经分配了空间，因此该成员的值为非空  if(EXT2\_I(inode)->i\_file\_acl){ //inode已经具备一个扩展属性块  bh=sb\_bread(sb,EXT2\_I(inode)->i\_file\_acl); //从磁盘读取数据到内存  //删除部分代码  header=HDR(bh);  end=bh->b\_data+bh->b\_size;  //删除部分代码  last=FIRST\_ENTRY(bh);  while(IIS LAST ENTRY(last)){ //循环查找扩展属性 |

第 4 章 从 理 论 到 实 战 — — E xt 2 文 件 系 统 代 码 详 解

if(lext2\_xattr\_entry\_valid(last,end,sb->s\_blocksize))

goto bad\_block;

if(last->e\_value\_size)

size\_t offs=le16\_to\_cpu(last->e\_value\_offs); if(offs<min\_offs)

471

472

min\_offs=offs;



if(not\_found>0){ //对比名称，确认是否有该扩展属性 not\_found=ext2\_xattr\_cmp\_entry(name\_index,

475

name len,

name,last);

477

if(not\_found<=0)

here=last; /如果找到了扩展属性，则here 是当前扩展属性

t-EXT2XATTR NEXTOap

482

if(not\_found>0) here=last;

483

484

485

//计算剩余的可用空间

free=min\_offs-((char\*)last-(char\*)header)-sizeof(\_u32);

}

//删除部分代码 if(not\_found){ //删除部分代码 }else{

//请求创建一个已经存在的属性 error=-EEXIST;

if(flags &XATTR\_CREATE) //指定在创建标记的情况下不允许更新

goto cleanup;

//更新剩余的可用空间

free+=EXT2\_XATTR\_SIZE(le32\_to\_cpu(here->e\_value\_size)); free+=EXT2\_XATTR\_LEN(name\_len);



error=-ENOSPC;

510

511

512

if(free<EXT2\_XATTR\_LEN(name\_len)+EXT2\_XATTR\_SIZE(value\_len)) goto cleanup;

//设置新属性

//删除部分代码 }else{

if(here->e\_value\_size){

571

char \*first\_val=(char\*)header+min\_offs;

size\_t offs=le16\_to\_cpu(here->e\_value\_offs); char \*val=(char\*)header +offs;

573

574

575

size\_t size=EXT2\_XATTR\_SIZE( le32\_to\_cpu(here->e\_value\_size));

if(size=EXT2 XATTR\_SIZE(value len))

文件系统技术内幕

大数据时代海量数据存储之道

|  |  |
| --- | --- |
| 579  580  581  582  583  584  585  586  587  588  589  590  591  592  593  594  595  596  597  598  599  600  601  602  614  615  616  617  618  619  620  621  622  623  624  625  626  647  648 | /如果新的扩展属性值的长度与原始扩展属性值的长度相同，则可以直接在原地更新 here->e\_value\_size=cpu\_to\_le32(value\_len);  memset(val+size-EXT2\_XATTR\_PAD,0,  EXT2\_XATTR\_PAD); //清理填充字节  memcpy(val,value,value\_len); goto skip\_replace;  /如果新的扩展属性值的长度与原始扩展属性值的长度不同，则需要删除原始扩展属性值 memmove(first\_val+size,first\_val,val-first\_val);  memset(first\_val,0,size); here->e\_value\_offs=0;  min\_offs+=size;  /调整所有值的偏移  last=ENTRY(header+1);  while(IIS\_LAST\_ENTRY(last)){  size\_to=le16\_to\_cpu(last->e\_value\_offs); if(o<offs)  last->e\_value\_offs=  cpu\_to\_le16(o+size); last=EXT2\_XATTR\_NEXT(last);  //将新的扩展属性值拷贝到指定位置  if(value!=NULL){  //插入一个新值  here->e\_value\_size=cpu\_to\_le32(value\_len); if(value\_len){  size\_t size=EXT2\_XATTR\_SIZE(value\_len); char\*val=(char\*)header+min\_offs-size;  here->e\_value\_offs=  cpu\_to\_le16(char\*)val-(char\*)header); memset(val+size-EXT2\_XATTR\_PAD,0,  EXT2\_XATTR\_PAD);  memcpy(val,value,value\_len);  //删除部分代码 return error; |

在代码4-30中，首先会根据扩展属性的名称从头到尾遍历已经存在的扩展属 性(第465行～第488行),并与新扩展属性名称进行对比。最终，这部分代码会 确定是否已经存在该名称的扩展属性及剩余的可用空间。

针对新的扩展属性值的长度与原始扩展属性值的长度相同的场景，由于不需要 新的存储空间，因此可以直接在原始地址进行更新(第578行～第586行)。

第4章从理论到实战——Ext2文件系统代码详解

针对新的扩展属性值的长度与原始扩展属性值的长度不同的场景，需要通过该 值前面(低地址)的值后移(向高地址移动)的方式覆盖旧的值，同时由于前面的 值的地址发生了变化，因此需要调整每个扩展属性项中记录值位置的成员(第589 行～第602行)。

最后更新扩展属性项中值的长度和偏移信息(第614行～第621行),并将新 的值拷贝到目的地址(第622行～第625行),也就是存储扩展属性值的块中。从 上面逻辑也可以看出，扩展属性名称的排列顺序与扩展属性值的排列顺序并非一 致，这一点需要注意。

在上述代码中需要注意的是，用户在调用接口时可以传递附加标识，如 XATTR\_REPLACE和XATTR\_CREATE等 。XATTR\_REPLACE表示用户期望进行 扩展属性值的替换操作，如果没有找到扩展属性的键，则返回失败信息。 XATTR\_CREATE 表示只进行创建操作，如果已经存在扩展属性的键，则返回失败 信息。

**4.11** **权** **限** **管** **理** **代** **码** **解** **析**

前文已经对如何进行ACL 设置进行了介绍，本节重点介绍一下Ext2 文件系统 中关于ACL 部分的实现。该部分内容我们分两部分进行介绍， 一部分是如何设置 文件的ACL 属性；另一部分是当访问文件时如何进行ACL 检查。

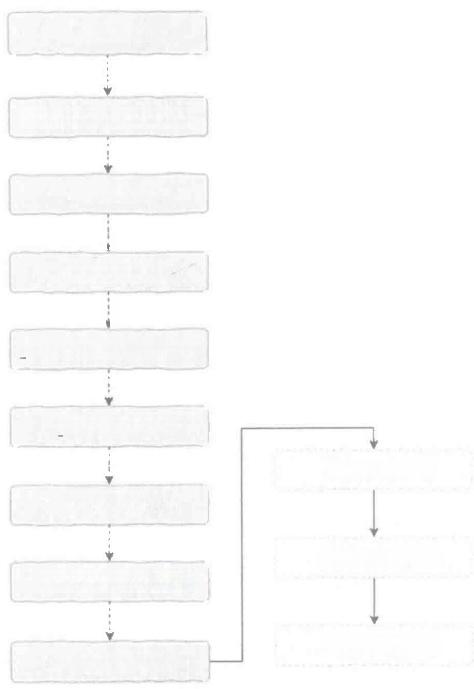
**4.11.1** **ACL的设置与获取**

前文已经提到ACL 是基于扩展属性实现的。我们先看一下设置ACL 的流程， 从setfacl(函数开始，到最终文件系统的函数调用栈，如图4-48所示。从图4-48中 可以看出，设置 ACL 的 API 其实调用的主要是扩展属性的接口，只是到 vfs setxattr()函数中执行了不同的分支，也就是ACL 的分支。

当整个流程到Ext2 文件系统后，最后也是调用扩展属性的实现来进行数据的 相关操作。所以，ACL 本质上就是扩展属性，只是名称比较特殊而已。

文件系统技术内幕

大数据时代海量数据存储之道

setfacl()

path\_setxattr(fs/xattr.c)

setxattr()

w

vfs\_setxattr()

事…

\_vfs\_setxattr\_noperm()

\_vfs\_setxattr()

ext2\_set\_acl()

handler->set()

\_ext2\_set\_acl()

posix\_acl\_xattr\_set()

ext2\_xattr\_set()

set\_posix\_acl()

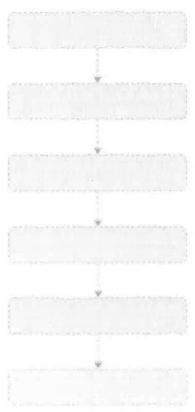
图4-48 设置ACL的流程

本节只介绍设置ACL 的流程，获取流程与查询流程类似，本节不再赘述。

**4.11.2** **ACL权限检查**

设置完成ACL 属性后就起作用了，那么当用户在访问文件系统时内核就会进 行相应的检查。以打开文件为例，权限检查的入口与RWX 权限管理相同，都是 may\_open(函数，其流程如图4-49所示。

第4章从理论到实战——Ext2文件系统代码详解



**may\_open()**

**inode\_permission()**

do\_inode\_permission()

generic\_permission (

acl\_permission\_check()

check\_acl()

图4-49 ACL权限检查流程

通过图4-49可以看出，在权限检查中会调用一个check\_aclO 函数，该函数就 是根据磁盘上存储的用户和权限信息，以及线程中的用户ID 和 组ID 等进行比对， 从而确定该用户是否有访问文件的权限。关于该函数的具体实现请大家自行阅读相 关代码，本节不再赘述。

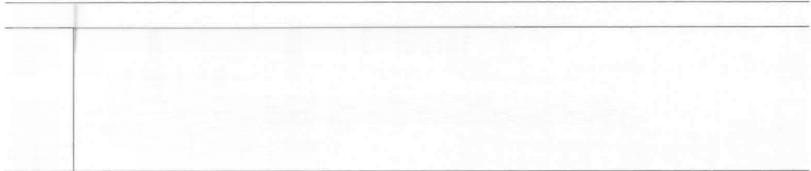
**4.12** **文** **件** **锁** **代** **码** **解** **析**

前文已经介绍了文件锁主要API 的用法及文件锁的基本原理。本节将介绍一 下文件锁在Linux 内核中是如何实现的。

**4.12.1** **flock()函数的内核实现**

系统API flock(函数的实现代码在fs/locks.c 文件中，通过函数名我们可以看 到该函数的具体实现，如代码4-31所示，在该函数中主要调用了两个函数，分别 创建一个锁结构体(第2237行)和执行锁操作(第2250行～第2255行)。

**代** **码** **4** **-** **3** **1** **flock() 函** **数** **的** **实** **现**



**fs/locks.c**

2218 SYSCALL\_DEFINE2(flock,unsigned int,fd,unsigned int,cmd)

2219 {

Ⅱ删除部分代码

2237 lock=flock\_make\_lock(f.file,cmd,NULL);// 创建并初始化锁结构体

2238 if(IS\_ERR(lock)){

2239 error=PTR\_ERR(lock);

**文件系统技术内幕**

大数据时代海量数据存储之道

|  |  |
| --- | --- |
| 2240  2241  2242  2243  2244  2245  2246  2247  2248  2249  2250  2251  2252  2253  2254  2255  2264 | goto out putf;  if(can\_sleep)  lock->fl\_flagsl=FL\_SLEEP;  error=security\_file\_lock(f.file,lock->fl\_type);//进行权限的判断  if (error)  goto out\_free;  //进行锁处理的逻辑  if (f.file->f\_op->flock)  error=f.file->f\_op->flock(ffile,  (can sleep)?F\_SETLKW:F\_SETLK, lock);  else  error=locks\_lock\_file\_wait(f.file,lock); //删除部分代码 |
| 对于锁相关的操作，如果具体文件系统实现了flock() 函数，则调用具体文件系 统实现(第2251行),否则调用VFS 中 的locks\_lock\_file\_wait (函数实现(第2255 行)。  locks\_lock\_file\_wait() 是执行锁操作函数，如果存在互斥的情况，那么进程将被 阻塞，直到调用者释放锁为止。图4-50所示为locks\_lock\_file\_wait() 函数的核心调 用流程。 | |
| locks\_lock\_file\_wait()  locks\_lock\_inode\_wait()  古  fiock\_lock\_inode\_wait()  flock\_lock\_inode() | |
| 图4-50 locks\_lock\_file\_waitO函数的核心调用流程  我 们 重 点 关 注 一 下 flock\_lock\_inode\_wait ( 函 数 ， 可 以 看 到 该 函 数 调 用 flock\_lock\_inode() 函数实现锁的判断，然后调用wait\_event\_interruptible() 函数实现 调用者进程的阻塞操作。当然，具体是否阻塞进程需要依赖锁的属性。  代 码4 - 3 2 f l o c k l o c k i n o d e w a i t ( )函 数 的 实 现  fs/locks.c | |
| 2160 | static int flock lock inode wait(struct inode\*inode,struct file\_lock\*fl) |

第4章从理论到实战 — — Ext2文件系统代码详解



int error;

2163 might\_sleep(;

2164 for(;;){

2165 error=flock\_lock\_inode(inode,fl); /判断锁定状态

2166 f(error !=FILE\_LOCK\_DEFERRED)

2167 break;

2168 error=wait\_event\_interruptible(fl->fl\_wait,

list\_empty(&fl->fl\_blocked\_member);// 将线程调度出

2170 if(error)

2171 break; 2172 

locks\_delete\_block();

2174 return error;

2175 1

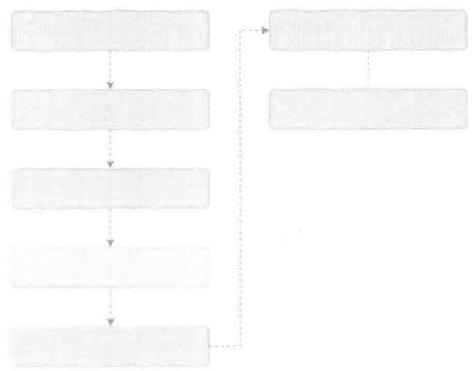
可以看到，锁定实现并不复杂。这里需要说明的是，文件锁的信息存储在inode 中 struct file\_lock\_context类型的成员变量i flctx,它记录着该节点所有文件锁的

信 息 。

**4.12.2** **fcntl()函数的内核实现**

fcntl() 函数的具体实现是在fs/fcntl.c 文件中。下面直接看一 下fcntl ( 函 数 的 核

心调用流程。



fentl()



do\_fentl()

fcntl\_setlk()

do\_lock\_file\_wait()

vfs\_lock\_file()

posix\_lock\_inode()

posix\_lock\_file()

图 4 - 5 1 fcntIO 函 数 的 核 心 调 用 流 程

在 图 4 - 5 1 中 ，do\_lock\_file\_waitO 为锁的关键实现函数，这里首先判断是否需 要将进程置于锁定状态，然后根据情况实现线程的调度。该函数的实现如代码4-33 所 示 。

**文件系统技术内幕**

大数据时代海量数据存储之道

代码4-33 do\_ lock\_file\_wait()函数的实现

**fs/locks.c**

|  |  |
| --- | --- |
| 2433 | static int do\_lock\_file\_wait(struct file \*filp,unsigned intcmd, |
| 2434 | struct file\_lock\*fl) |
| 2435 |  |
| 2436 | int error; |
| 2437 |  |
| 2438 | error=security\_file\_lock(filp,fl->fl\_type); |
| 2439 | if(error) |
| 2440 | return error; |
| 2441 |  |
| 2442 | for(;){ |
| 2443 | error=vfs\_lock\_file(filp,cmd,fl,NULL); //进行锁的具体处理和判断 |
| 2444 | if(error !=FILE\_LOCK\_DEFERRED) //异步场景进程不休眠 |
| 2445 | break; |
| 2446 | error=wait\_event\_interruptible(fl->fl\_wait, |
| 2447 | list\_empty(&fl->l\_blocked\_member)); |
| 2448 | if(error) |
| 2449 | break; |
| 2450 |  |
| 2451 | locks\_delete\_block(f1); |
| 2452 |  |
| 2453 | return error; |
| 2454 |  |
| 从上述代码中可以看出，锁相关的处理主要在vfs\_lock\_file() 函数中完成，该函 数的实现如代码4-34所示。如果具体文件系统实现了锁相关的函数，则调用具体 文件系统的函数，否则使用POSIX 锁的实现。  代码4-34 vfs\_lock\_file()函数的实现 | |
| fs/locks.c | |
| 2424  2425  2426  2427  2428  2429  2450 | int vfs\_lock\_file(struct file\*filp,unsignedint cmd,structfile\_lock\*fl,structfile\_lock\*conf)  if(filp->f\_op->lock)  return filp->f\_op->lock(filp,cmd,fl); else  return posix\_lock\_file(filp,fl,conf); |

Ext2 文件系统并没有实现自己锁相关的逻辑，Ext2 文件系统其实使用的是 POSIX 锁的实现。因此，本节也是以虚拟文件系统中的POSIX 锁的实现为例来介 绍文件锁的实现。

在文件系统中需要一些数据结构来记录锁的状态。与文件锁相关的数据结构主 要有两个： 一个是file\_lock 结构体，它表示一个文件锁，包括所有者、类型、进程 ID 和锁的起止位置等；另一个是file\_lock\_context 结构体，它是inode 中的 一 个成

第4章从理论到实战—— Ext2文件系统代码详解

员，用于记录该文件上已经加锁的信息。

file\_lock 结构体是一个锁的实例，用于记录锁的各种属性和状态，该结构体的 定义如代码4-35所示。在该结构体中除了前文所述的内容，还有一些用于链表的 成员，这些成员用于将一个实例关联到具体的链表或哈希表中。

**代码4** **-** **35** **file** **\_** **lock结构体的定义**

**fs/locks.c**

|  |  |  |
| --- | --- | --- |
| 1090 | struct file\_lock{ |  |
| 1091 | struct file\_lock\*fl\_blocker; | //阻塞锁 |
| 1092 | struct list\_head fl\_list; | //通过变量链接到file\_lock\_context结构体 |
| 1093 | struct hlist\_node fl\_link; | //全局链表中的节点 |
| 1094 | struct list\_head fl\_blocked\_requests; | //请求链表指向这里 |
| 1095 |  |  |
| 1096 |  |  |
| 1097 |  |  |
| 1098 |  |  |
| 1099 |  |  |
| 1100 | fl\_owner\_tfl\_owner; |  |
| 1101 | unsigned intfl\_flags; | //锁的特性 |
| 1102 | unsigned char fl\_type; | //锁的类型 |
| 1103 | unsigned int fl pid; | //持有该锁的进程ID |
| 1104 | int f\_link\_cpu; |  |
| 1105 | wait\_queue\_head\_tf\_wait; |  |
| 1106 | struct file \*fl\_file; |  |
| 1107 | loff\_tfl\_start; | //范围锁的起始位置 |
| 1108 | loff\_tfl\_end; /删除部分代码 | //范围锁的终止位置 |
| 1126 | randomize\_layout; |  |

file\_lock\_context 结构体的定义如代码4-36所示，其中，有3个链表成员，用 于记录已经在该文件上加锁的信息。由于虚拟文件系统的结构体要兼容不同的锁类 型，因此这里有3个不同的链表。对于POSIX 接口来说可以使用flc\_posix 来存储 锁的内容。

**代码4-36** **file\_** **lock\_context结构体的定义**

fs/locks.c

|  |  |
| --- | --- |
| 1128 | struct file\_lock\_context{ |
| 1129 | spinlock\_t flc\_lock; |
| 1130 | struct list\_head flc\_flock; |
| 1131 | struct list\_head flc\_posix; |
| 1132 | struct list\_head flc\_lease; |
| 1133 | ; |

接下来分析一下POSIX 锁的具体实现，看一看是如何操作这些数据结构的。 前文已述，对于POSIX 锁来说，具体由posix\_lock\_file() 函数实现。该函数的部分

**文件系统技术内幕**

大数据时代海量数据存储之道

代码如代码4-37所示，该函数主要对inode 上已经添加的锁与新的锁请求进行比 较，确定是否存在冲突。该函数逻辑相对复杂，除了检测是否与其他进程产生冲突， 还对本进程之前添加的锁进行合并等处理。

代码4-37 posix\_lock\_inode()函数的实现

|  |  |
| --- | --- |
| fs/locks.c | |
| 1131  1132  1133  1144  1145  1146  1161  1162  1168  1169  1170  1171  1172  1173  1174  1175  1176  1177  1178  1179  1180  1181  1182  1183  1184  1185  1186  1187  1188  1189  1190  1191  1192  1193  1194  1195 | static int posix\_lock\_inode(struct inode \*inode,struct file\_lock\*request, struct file\_lock\*conflock)  {  //删除部分代码  //从inode中查找file\_lock\_context结构体，如果没有则创建 ctx=locks\_get\_lock\_context(inode,request->fl\_type);  if(!ctx)  return(request->fl\_type=F\_UNLCK)20:-ENOMEM;  /删除部分代码  percpu\_down\_read(&file\_rwsem); spin\_lock(&ctx->flc\_lock);  //从file\_lock\_context结构体中获取所有inode已经持有的锁，并进行逐个遍历  if(request->fl\_type!=F\_UNLCK){  list\_for\_each\_entry(fl,&ctx->flc\_posix,fl\_list){  if(!posix\_locks\_conflict(request,fl))//检查是否有冲突的锁存在 continue; //如果没有冲突则进行下一个检查  if(conflock)  locks\_copy\_conflock(conflock,fl); error=-EAGAIN;  if(!(request->fl\_flags &FL\_SLEEP)) goto out;  //进行死锁检测。另外，对于持有相同锁的情况下，需要将其加入阻塞列表中 error=-EDEADLK;  spin\_lock(&blocked\_lock\_lock);  //在进行死锁检查时确保在该节点上没有任何锁被阻塞 locks\_wake\_up\_blocks(request);  if (likely(!posix\_locks\_deadlock(request,fl))){ error=FILE\_LOCK\_DEFERRED;  //如果存在锁，且冲突，则此时将请求锁加入阻塞列表，然后返回  locks\_insert\_block(fl,request, posix\_locks\_conflict);  spin\_unlock(&blocked\_lock\_lock); goto out; |

第4章从理论到实战 — — Ext2文件系统代码详解



H196

/删除部分代码，这部分代码逻辑是处理冲突的场景，此时主要进行锁的合并等处理 if(ladded){

1304 if(request->fl\_type=F\_UNLCK){

1305 if(request->fl\_flags&FL\_EXISTS)

1306 error=-ENOENT;

goto out;

1308 

1309

if(Inew\_fl){

1311 error=-ENOLCK;

1312 goto out;

/如果没有任何锁，则将该锁加入inode 的 ctx 中，此时返回值为0

1314 ocks\_copy\_lock(new\_fl,request);

ocks\_move\_blocks(new\_fl,request);

1316 locks\_insert\_lock\_ctx(new\_fl,&fl->fl\_list);

1317 fl=new\_fl;

new\_fl=NULL;

1319

1320 //删除部分代码

1352 

通过上述代码可以看出，posix\_lock\_inode (函数会根据处理结果的不同返回不 同的返回值。而返回值决定了进程的后续状态。这部分代码就是代码4- 33中相关 的逻辑。



第

章

**基于网络共享的网络文件系统**

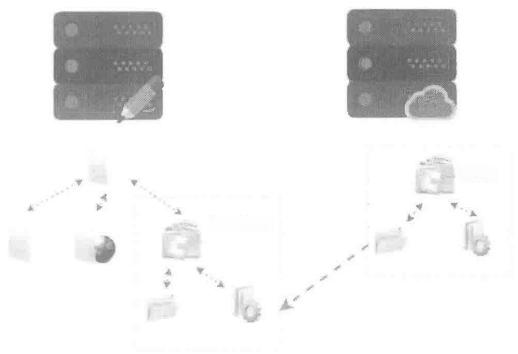
前面章节已经对本地文件系统进行了比较详细的分析。接下来介绍一下在实际 生产环境中应用非常广泛的一种文件系统，即网络文件系统。

**5.1** **什么是网络文件系统**

网络文件系统 (Network File System) 是一种将远端的文件系统映射到本地的 文件系统。这里的远端是指一个存储系统，通常称为NAS 存储。本地是指客户端， 通常是运行业务的服务器。存储系统可以是专属的存储设备，如一些存储厂商的存 储产品，也可以是基于普通服务器搭建的存储系统。

早些时候的企业级架构，数据存储普遍采用网络文件系统，最为著名的就是 Sun 的 NFS。 微软也有类似的网络文件系统，如CIFS。网络文件系统的目的就是 将存储系统上的文件系统映射到计算节点(如Web 服务器),如图5-1所示。

第5章基干网络共享的网络文件系统



/data

/mnt/data

图5-1 网络文件系统访问示意图

在图5-1中，网络文件系统将远程的目录树映射到本机，成为本机目录树中的 一棵子树。对于普通用户来说，访问该子树中的内容与访问本地其他子目录内容没 有任何差异。也就是用户不会感知到该子树的内容是在远端，也没必要感知这种 存在。

同时，通过图5-1可以看出，对于网络文件系统来说通常存在两部分组件： 一 部分是客户端的文件系统；另一部分是服务端的服务程序。客户端文件系统的逻辑与 本地文件系统无异，差异是在读/写数据时不是访问磁盘等设备，而是通过网络将请求 传输到服务端。服务端负责客户端请求的处理，将数据存储到磁盘等存储介质上。

客户端的实现可以基于操作系统的文件系统框架来实现一个文件系统，该文件 系统负责接收应用的请求，并将请求转发到服务端，如NFS 文件系统在Linux 或 Windows 的实现。但是也不一定，由于网络文件系统通信本身是基于以太网的协 议，因此也可以用一个库函数来实现相关功能。当然，这种方式就无法实现直观的 文件操作了，只能供开发应用程序使用。

服务端数据的存储可以借助于普通的本地文件系统，也可以实现自己的文件系 统。比如，Linux 下 的NFS 实现，在NFS 服务端通常是使用常规的本地文件系统 来存储数据的，如Ext4 、XFS 或 Btrfs 等。而一些存储设备提供商通常会实现自己 的文件系统，如EMC 的 UFS64 文件系统和NetApp 的 WAFL 文件系统等。

服务端数据的存储并不一定需要基于某种文件系统，甚至可以基于KV 数据块 来实现。对于服务端来说，只需要能够实现对客户端请求的解析即可。这些请求包 括创建/删除文件、读/写数据、扩展属性和权限等。如果大家对这里的描述不理解 也没关系，我们在后续章节会详细介绍。

**文件系统技术内幕**

大数据时代海量数据存储之道

**5.2** **网络文件系统与本地文件系统的异同**

从普通用户的角度来看，网络文件系统与本地文件系统没有明显的差异。在 Linux平台下挂载文件系统时，本地文件系统是将一个块设备挂载到某个目录下面， 而网络文件系统则是将一个包含IP 地址的远程路径挂载到某个目录下面。

网络文件系统与本地文件系统的差异在于数据的访问过程。以写数据为例，本 地文件系统的数据是持久化存储到磁盘(或者其他块设备)上的，而网络文件系统 则需要将数据传输到服务端进行持久化处理。

另外一个差异点是本地文件系统需要进行格式化处理才可以使用。而网络文件 系统则不需要客户端进行格式化操作，通常只需要挂载到客户端就可以直接使用。 当然在服务端通常是要做一些配置工作的，包括格式化操作。

网络文件系统最主要的特性是实现了数据的共享。基于数据共享的特性，使得 网络文件系统有很多优势，如增大存储空间的利用效率(降低成本)、方便组织之 间共享数据和易于实现系统的高可用等。

**5.3** **常见的网络文件系统简析**

网络文件系统有很多，其中有两个标准的网络文件系统最为出名：一个是Linux 经常使用的NFS(Network File System);另一 个是 Windows 经常使用的CIFS (Common Internet File System)。除此之外，其实还有很多其他的网络文件系统，如 AFS(Andrew File System) 等 。

**5.3.1 NFS文件系统**

NFS 是一个非常老牌的网络文件系统，于1980年由Sun 公司开发，并随SunOS 一起发布。NFS 不仅是指网络文件系统，现在更多是指一种协议，而且是一种开放 的网络文件系统协议。

关于NFS 最新的描述可以通过RFC7531 获得更多的描述信息。NFS 从第一次 发布到现在已经几十年了，也发展了不同的版本，从最初内部使用的1.0版本到现 在的4.X 版本。

前文提到网络文件系统包含客户端文件系统和服务端服务两个组件。对于客户 端来说，目前主流的操作系统都支持NFS, 如 Linux、UNIX 和 Windows 等 。

在服务端，通常需要一个服务软件，实现对本地文件系统的导出。这样通过NFS

**第5章** 基于网络共享的网络文件系统



协议可以将文件系统导出到客户端。由于NFS 协议是开放的，因此有很多NFS 的 服务端实现，Linux 内核中本身就集成了一个内核模块(NFSD) 。 在用户态有一个 比较有名的NFS 服务端软件NFS-Ganesha。

**5.3.2 SMB协议与CIFS 协议**

Windows 也有一套用于在网络实现文件共享的协议，即SMB(Server Message Block, 服务器消息块)协议，它是一套基于NetBIOS 的文件共享协议。随着以太 网技术的发展，SMB 协议逐渐与NetBIOS 脱离。在Windows 2000中 ，SMB 协 议 可以直接运行在TCP/IP 协议之上。

SMB 协议不仅可以用于文件共享，其本身是用于进行节点之间消息传输的协 议，可以用在文件共享、打印机共享及其他信息共享领域。

随着互联网的发展，微软基于SMB 协议定义了一个通用的文件共享协议，即 CIFS(Common Internet File System,通用因特网文件系统)协议，可以认为CIFS 协议是SMB 协议的一个具体实现。

CIFS/SMB 协议的情况与NFS 协议的情况非常类似，其架构也是C/S 架构，而 且协议也是开放的。因此，在客户端和服务端都有很多具体的实现。

主流的操作系统都有CIFS 协议的实现，Windows 自然不用多说。Linux 也有CIFS 协议的实现，如果阅读源代码，就会发现CIFS 协议就在与NFS 协议同级的目录下。

服务端的实现也很多，除了Windows 本身的实现，在Linux 下也有一个组件 能够提供CIFS 协议的服务，那就是Samba。除了开源软件，还有很多商用的CIFS 协议服务，如MoSMB 、Tuxera SMB和 Likewise 等 。

**5.4** **网络文件系统关键技术**

网络文件系统本质上也是一个文件系统，因此本地文件系统所使用的技术在网 络文件系统中通常也都是要使用的，如缓存技术、文件锁、快照克隆和权限管理等。 区别在于，这些特性大多是借助于位于服务端的文件系统实现的，网络文件系统本 身并不能实现。对于网络文件系统来说，只需要通过协议将请求传输到服务端进行 处理即可。

由于网络文件系统基于网络实现，除了具有本地文件系统的一些特性，还有其 特殊的地方。比如，需要通过应用层的协议传输命令、文件锁需要考虑跨网络的情 况等。本节将介绍一些相对本地文件系统来说网络文件系统特有的技术。

文件系统技术内幕

大数据时代海量数据存储之道

**5.4.1 远程过程调用(RPC 协议)**

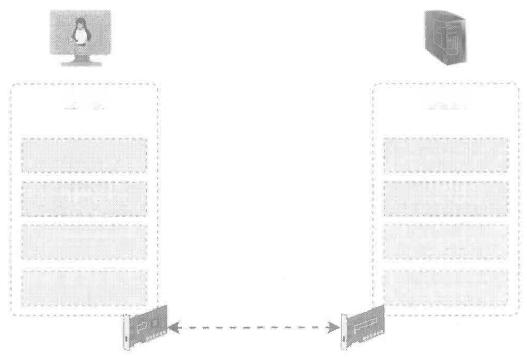
网络文件系统定义了客户端(又被称为主机端)与服务端交互的协议(NFS 协 议),网络文件系统的协议是通过函数调用的方式定义的，主要内容包含ID、参 数 和返回值等。客户端到服务端访问通常是通过网络来访问的，因此在具体实现时需 要将协议定义的函数形态转化为网络数据包，然后在服务端收到数据包再执行预定 的动作后给客户端发送反馈。

由于在客户端与服务端都要实现对协议数据的封装和解析，因此实现起来比较 复杂。为了降低复杂性，通常会在文件系统业务层与TCP/IP 层之间实现一层交互 层，这就是RPC 协议。这种分层的方式是计算机领域经常用到的处理问题的方式， 如TCP/IP 的协议栈，MVC 模式等。

RPC(Remote Procedure Call,远程过程调用)是TCP/IP 模型中应用层的网络 协议(OSI 模型中会话层的协议)。RPC 协议通过一种类似函数调用的方式实现了 客户端对服务端功能的访问，简化了客户端访问服务端功能的复杂度。

在客户端调用RPC 函数时，会调用RPC 库的接口将该函数调用转化为一个网 络消息转发到服务端，而服务端的RPC 库则对网络数据包进行反向解析，调用服 务端注册的函数集(存根)中的函数实现功能，最后将执行的结果反馈给客户端。

图5-2所示为RPC 协议架构示意图，通常包括应用、客户端/服务端存根(stub)、 RPC运行时库和传输协议。

**客户端**

应用

客(户st端u)根

RPC运行时库

传输协议

**服务端**

应用

服务端存根 (stub)

RPC 运行时库

传输协议



图5-2 RPC协议架构示意图 下面介绍图5-2中的几个关键概念。

应用是基于PRC 协议实现的具体应用程序。以网络文件系统为例，客户端

第5章基干网络共享的网络文件系统

的应用是指文件系统，而服务端的应用则是指文件系统服务，如NFSD 或 NFS- Ganesha 等。

存根(stub) 是定义的函数集，该函数集根据应用业务的需求而确定。以网络 文件系统为例，函数集包括创建文件、删除文件、写数据和读数据等。函数集通常 需要分别在客户端和服务端定义一套接口，而且客户端的函数集与服务端的函数集 是一一对应的。

RPC运行时库通常是一个公共库，实现了RPC 协议的公共功能，如请求的封 装与解析、消息收发和网络层面的错误处理等。

需要注意的是，RPC 并不是网络文件系统专用的协议，而是在分布式系统的很 多地方都有应用。RPC 协议不仅在操作系统内核中有实现，也有很多用户态实现， 如gRPC 、Dubbo 和 Thrift 等。不同的RPC 协议的定义并不一样。

**5.4.2** **客户端与服务端的语言——文件系统协议**

网络文件系统本质上是一个基于C/S (客户端/服务端)架构的应用，其大部分 功能是通过客户端与服务端交互来实现的。因此，对于网络文件系统来说，其核心 之一是客户端与服务端的交互语言——文件系统协议。

由于网络文件系统通常基于以太网进行连接，因此网络文件系统的协议通常也 是基于TCP 协议或UDP 协议来实现的。我们可以将网络文件系统的协议理解为 TCP/IP 应用层的协议(但实际情况要复杂一些)。

网络文件系统的协议的定义类似函数调用，包含ID (可以理解为函数名称), 参数和返回值。其定义是非常清晰的，其语义与文件系统操作的语义基本上一一对 应。以经常使用的NFSv3 协议为例，这里列出该协议的部分命令，如表5-1所示。

**表5-1** **NFSv3协议的部分命令**

|  |  |  |  |
| --- | --- | --- | --- |
| 名 称 | 编码(ID) | 操作系统API(Linux) | **说** **明** |
| CREATE | 8 | create/open | 创建一个常规文件 |
| REMOVE | 12 | remove/unlink | 删除一个常规文件 |
| WRITE | 7 | write | 向文件写入数据 |
| READ | 6 | read | 从文件读取数据 |
| LOOKUP | 3 |  | 查找文件 |
| MKDIR | 9 | mkdir | 创建一个目录 |
| READDIR | 16 | readdir | 读取目录中的内容 |
| RMDIR | 13 | rmdir | 删除目录 |
| COMMIT | 21 | flush | 提交缓存中的数据 |

**文件系统技术内幕**

大数据时代海量数据存储之道

通过表5-1可以看出，在NFSv3 协议中定义的语义与我们对文件系统的操作 有非常明确的对应关系。对于文件来说有创建、删除、读/写和查找等命令，对于目 录来说也有类似的命令。当然，本节只是展示了NFSv3 协议的部分内容，更多细 节请参考其他资料。

SMB2 协议与NFSv3 协议类似，也是实现了一些与文件系统语义对应的协议 命令。表5-2所示为SMB2 协议的部分命令。

**表5-2** **SMB2协议的部分命令**

|  |  |  |
| --- | --- | --- |
| 名 称 | 编码(ID) | **说** **明** |
| CREATE | 0x0005 | 创建一个文件 |
| CLOSE | 0x0006 | 删除一个文件 |
| FLUSH | 0x0007 | 刷新缓存 |
| READ | 0x0008 | 向文件写入数据 |
| WRITE | 0x0009 | 从文件读取数据 |
| QUERY\_DIRECTORY | 0x000e | 获取目录中的内容 |
| QUERY\_INFO | 0x0010 | 查询文件和命名管道等对象的信息 |

通过NFSv3 协议和SMB2 协议可以看出，无论哪种协议，都有一组与文件系 统语义对应的协议命令。这样，客户端对网络文件系统的访问都可以通过协议传输 到服务端进行相应的处理。由于文件系统语义与协议命令清晰的对应关系，网络文 件系统协议并不复杂，只是内容比较多。

**5.4.3** **文件锁的网络实现**

我们知道在文件系统中有一个文件锁的特性，该特性类似多线程编程中的锁机 制。通过文件锁可以保证当多线程访问相同文件时只能有一个线程进行更新，其他 线程只能等待，避免出现同时更新线程导致出现数据不一致的问题。

为了能够支持该功能，网络文件系统也应该支持类似的特性。但是由于网络文 件系统实际功能在服务端实现，而且可以有多个客户端同时访问同一个文件系统。 因此，网络文件系统的文件锁的实现需要经过网络在服务端实现。

在NFS 协议族中，基于网络文件锁是有一个协议的，称为NLM(Network Lock Manager, 网络锁管理)。在NFSv2 和 NFSv3 版本中没有定义锁协议，因此都是通 过NLM 协议实现一个独立的服务。而NFSv4 版本的协议中已经将锁相关的协议考 虑进来了，因此没有独立的NLM 协议了。

第5章基于网络共享的网络文件系统

**5.5** **准** **备** **学** **习** **环** **境** **与** **工** **具**

学习一项技术最高效的方法就是动手实践。为了能够更好地学习网络文件系统 相关的内容，我们搭建一个NFS 文件系统，包括服务端的安装配置及客户端的挂 载等内容。

**5.5.1 搭建一个NFS 服务**

在 Linux 平台搭建NFS 服务并不复杂。很多公司利用服务器和Linux 来搭建 NFS服务。我们甚至还可以通过树莓派搭建一个家用NFS 服务。接下来以Linux 为 例介绍一下如何搭建一个NFS 服务。

本书重点在于讲解实现原理，因此本节并不会详细地讲解安装过程。按照本节 步骤安装是可以保证运行成功的，只是这里是简化的安装和配置步骤，缺少安全等 相关的设置。更详细的安装步骤可以参考《鸟哥的Linux 私房菜：服务器架设篇》 (第3版)[10]和《UNIX/Linux 系统管理技术手册》第4版[,其对NFS 的安装和 配置进行了非常详细的介绍。

**1.NFS** **服务的安装**

以Ubuntu 18.04为例，在Linux 服务器执行如下命令就可以将NFS 服务端软 件安装成功：

sudo apt install nfs-kernel-server

如果是CentOS 则可以执行如下命令进行安装： sudo yum install nfs-utils

**2** **.** **导出目录**

完成安装之后就可以导出某个目录进行测试。以Ubuntu 18.04为例，在/srv 目 录下创建一个新目录，并设置该目录的访问权限，命令如下：

mkdir/srv/nfs

chmod 777/srv/nfs

完成资源准备之后就可以进行NFS 服务端的配置。其目的是让服务端软件识 别该目录，并且能够进行管理，也就是让服务端导出该目录。

打开/etc/exports, 并将如下内容添加到该文件中：

/srv/nfs \*(rw,sync,no\_subtree\_check)

文件系统技术内幕

大数据时代海量数据存储之道

重 启NFS 服务即可(NFS 其实是可以不用重启服务使配置生效的)。这里需要 注意的是，Ubuntu 环境和CentOS 环境重启服务的命令是不同的。

**3** **.挂载文件系统**

在客户端节点安装需要的软件包，命令如下： sudo aptinstall nfs-common

如果没有报错，则说明安装成功。然后执行如下命令就可以将服务端的目录挂 载到本地。之后我们就可以在客户端访问该目录，这时对/mnt/nfs 目录的读/写其实 就是对服务端/srv/nfs 目录的读/写：

mount <192.168.2.113>:/srv/nfs/mnt/nfs/

上面的IP 地址是服务端的IP 地址。这里需要注意的是，在执行mount 命令之 前需要创建本地目录/mnt/nfs。如果没有这个本地目录，则在挂载时会出现挂载失 败的情况。

**5.5.2** **学习网络文件系统的利器**

网络文件系统除客户端与服务端的架构和代码逻辑外，最为核心的内容就是其 协议。对于协议的学习我们可以借助网络抓包工具，经常用到的有 tcpdump 和 WireShark 等 。

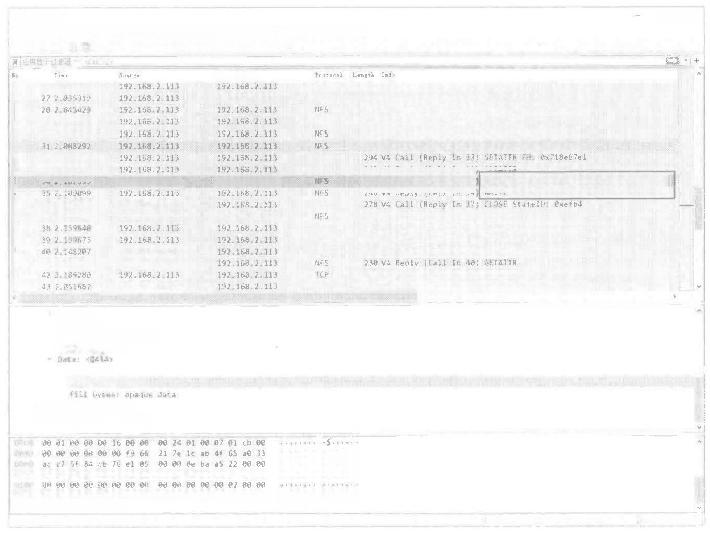
tcpdump 是一个命令行的抓包工具，非常适合在服务器版本的Linux 上使用。 使用方法也比较简单，如下是一个具体的实例：

tcpdump-i lo-w/tmp/dump.pcap tcp port 2049

其中，-i 表示要监测的网络接口，-w 表示将抓取的数据写入的文件，后面的参 数则表示监测的协议和端口号。通过条件过滤可以抓取我们关心的数据包。毕竟网 络数据非常多，如果没有条件过滤，则在分析数据时会有大海捞针的感觉。

WireShark 是一个具有GUT的网络抓包工具，该工具的功能与tcpdump 工具的 功能一样，但最大的特点是可视化做得非常好，而且实现了很多应用层协议的支持 ( 如HTTP 协 议 、NFS 协议和SMB 协议等)。图5-3所示为抓取的NFS 协议的部分 数据包。

**第5章基干网络共享的网络文件系统**



口 x

无Ew. 工具n 附 H

登

Brctinati

230V4 Reply (cll Tn 25)6ETAITTR

66702+2049 [ACK]Seq-1621 Ack-1477 Mn-3477 Len-e TSval-85209176 246V4 Call (Raply In 29)GETATIRFH;nd6318dad

230 V4 Reply(call In 28)GETATTR

322V4 Call(Hepiy In )OPEN DH:Ox718e67e1/ 994V4 Reply(Call En 3e)oPEM Statel0:@ef4

322.888334

3

246VA Feply (Call In 36)CLO5E

246V4 Call (Reply In 39)GFTATTR FH ad6318dad

230 V4 Reply(Call In 38)GETATTR

246 VCall (Reply In 41)GETATTR FH:Qxd6318dad

412.148245

66702-2049 [AEX]5ea-3085ack-2933 Win-3494 [en-g TSval-8520919:

246 Va Cal1 (Reply In 44)GETATTR FH:Ox6318dad

StatelD

offset!

stable/FLE SWIK4() Wrlte length:

length:3

contentst DATAS

Opcode:6ETATT8(9)

TMuin Opcode:MRITE(38)1

… f-e-3

ae 26000e00009cd95a

àne 000 C

001800300e00

O7 he fan3.a bnst

<197.168.2.11>

<192.168.2.113> <192.168.2.113>

<192.168.2.113> <192.168.2.113>

192,168.2.113

umppcap

文件用调F) 画 M 

2.

362.110026 37.2.118960

Q咨分析深沟统f(S)@电话M既



分塘：5 · 已示 Goo0)

IES

FS

FS

NFS

29.2.845468 3e 2.088198

FS

TCP

HFS

f80fa

8082e

A 置：pafmlt

<192.168.2.113>

<192.168.2.113>

192.168.2.m

NFS HES

262.855316

f7

0



0900

IFS

…

5f

图5-3 抓取的NFS协议的部分数据包

前文提到WireShark 一个好处是实现了对多种常见协议的支持。这里的支持是 指它能将抓取的二进制数据与具体的协议字段对应起来，直接展示解析后的结果， 非常直观。

**5.6** **网络文件系统实例**

本节以NFS 协议为例深入地解析文件系统软件架构与协议等相关内容。对于 SMB 协议来说，其实差别不大，限于篇幅有限，本节不再赘述。

**5.6.1** **NFS文件系统架构及流程简析**

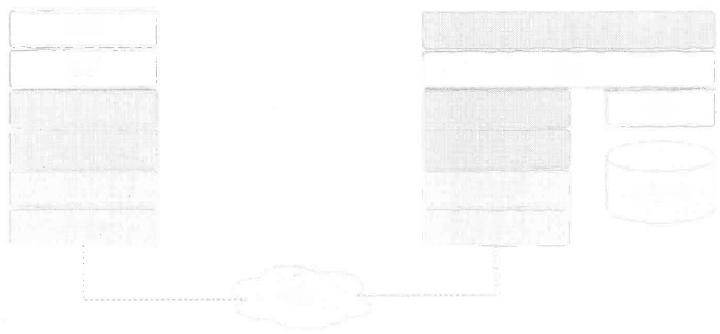
通过前文大家能够比较形象地认识一下NFS, 也为后续深入学习NFS 文件系 统构建一个测试验证环境奠定基础。下面来看一下NFS 的整体架构。

NFS 分布式文件系统是一个C/S (客户端/服务端)架构。其客户端是Linux 内 核中的一个文件系统，跟Ext4 和 XFS 类似，差异在于其数据请求不存储在本地磁 盘，而是通过网络发送到服务端进行处理。

从图5-4可以看出，NFS 也是位于VFS 下的文件系统。因此当NFS 挂载后， 其与本地文件系统并没有任何差异，用户在使用时也是透明的。

**文件系统技术内幕**

大数据时代海量数据存储之道



Client

VFS

NFS

RPC/XDR

TCP/IP

以太网

网络

NFS

RPC/XDR

NFS 服务

VFS

TCP/IP

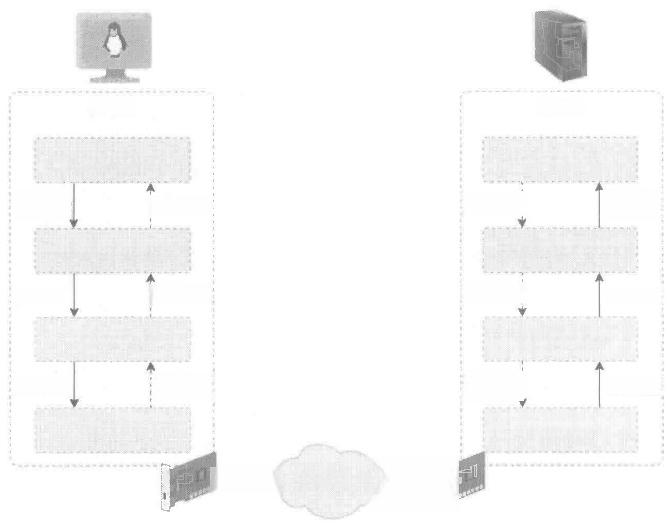
以太网

本地磁盘

Ext4

图5-4 NFS的整体架构

NFS 的通信使用的是RPC 协议，该协议也是Sun 公司发明的一种网络通信协 议 。RPC 协议基于TCP 协议或UDP 协议，是一个会话层的协议，可以与应用层的 HTTP 协议类比理解。RPC 协议的通信流程如图5-5所示。



客户端

应用(调用者)

函数调用 通数返回

客户端存根(stub)

RPC 调用 RPC 返回

RPC 运行时库

发送数据 接收数据

网络服务

- 以太网

→

**服务端**

服务(被调用者)

函数返回 函 数 调 用

服务端存根(stub)

函数返回 函数调用

RPC 运行时库

发送数据 接收数据

网络服务

图5-5 RPC协议的通信流程

在该流程中，当应用想完成某个功能时，可以调用客户端存根中的函数，而该 函数封装消息后调用RPC 接口。此时，RPC 运行时库会将消息封装后通过网络发 送到服务端。服务端RPC 运行时库接收到该消息后会进行消息的解析，然后调用

**第5章** 基于网络共享的网络文件系统



服务端的存根函数，服务端的存根函数调用服务端的业务处理函数完成相关处理。

完成处理后，服务端的存根函数会封装一个应答消息，然后调用PRC 运行时 库的API 进行发送。后续的整个流程与请求发送一致。最后在客户端的应用会收到 其所调用函数的返回值，这个返回值其实就是服务端发送的应答消息。对于客户端 的应用，这个函数调用与本地函数调用并没有明显的差异，其具体工作都是通过 RPC运行时库传输到服务端完成的。

为了使大家更加清晰地理解NFS 的架构，下面以创建子目录为例来介绍一下 NFS 文件系统与服务端通信的过程。NFS 文件系统有很多版本，很难一一介绍所有 版本。为了便于大家理解和学习，下面以NFSv3文件系统为例进行介绍。

由于NFS 文件系统基于VFS 文件系统框架，因此不可避免地需要实现一套函 数指针，并在挂载时进行注册。这主要是保证从VFS 文件系统下来的请求可以转 发到NFS 文件系统进行处理。代码5-1是NFS 文件系统实现的目录函数指针集合， 该函数指针集合实现了目录相关的操作。

**代码5-1** **NFS文件系统实现的目录函数指针集合**

**fs/nfs/nfs3proc.c**

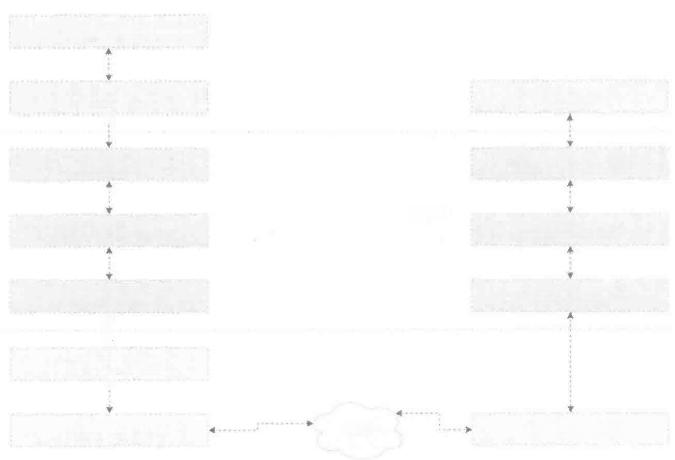
|  |  |
| --- | --- |
| 963 | static const struct inode\_operations nfs3\_dir\_inode\_operations={ |
| 964 | .create 三nfs\_create, |
| 965 | lookup 三nfs\_lookup, |
| 966 | .link =nfs\_link, |
| 967 | .unlink nfs\_unlink, |
| 968 | symlink nfs\_symlink, |
| 969 | .mkdir =nfs\_mkdir, |
| 970 | .rmdir =nfs\_rmdir, |
| 971 | .mknod =nfs\_mknod, |
| 972 | .rename 三nfs\_rename, |
| 973 | .permission=nfs permission, |
| 974 | .getattr =nfs\_getattr, |
| 975 | .setattr =nfs\_setattr, |
| 976 | #ifdef CONFIG\_NFS\_V3\_ACL |
| 977 | .listxattr =nfs3\_listxattr, |
| 978 | get\_acl nfs3\_get\_acl, |
| 979 | set\_acl 三nfs3\_set\_acl |
| 980 | #endif |
| 981 | k: |

以创建子目录为例，在NFS 文件系统中的具体实现函数为nfs\_mkdir( 。 当 用 户通过程序调用mkdir(O函数或执行mkdir 命令时，会通过软终端触发VFS 文件系 统的vfs\_mkdir(函数，最后调用NFS 文件系统的nfs\_mkdir() 函数。创建目录的整 体流程如图5-6所示，其中包含服务端的处理流程。

**文件系统技术内幕**

**大数据时代海量数据存储之道**

客户端 服务端

do\_mkdirat()

VFS

vfs\_mkdir()

vts\_mkdir(

▲

nfsd\_create\_locked(

nfs\_mkdir0

NFSD

nfs3\_proc\_mkdir(

NFS

nfsd\_create()

nfsd3\_proc\_mkdiro

nfs3\_do\_create(

… … … 事

rpc\_call\_sync0

A

RPC

pc\_run\_task() 网络 SVC\_process0

图5-6 创建目录的整体流程

nfs\_mkdir(函数首先会进行必要的权限检查，然后调用客户端(主机端)存根 中 的nfs3\_proc\_mkdir (函数。该函数进行RPC 调用的基本数据的准备，然后间接调 用RPC 服务的API(rpc\_call\_sync() 函数),将请求发送到服务端。

服务端收到消息后会根据消息中的关键信息调用服务端存根函数，本实例为 nfsd3\_proc\_mkdir() 函数。存根函数会调用业务函数(nfsd\_create()) 来完成具体的操 作。在本实例中，NFSD 最终会调用VFS 文件系统中的vfs\_mkdir() 函数，然后 vfs\_mkdir() 函数调用具体文件系统(与导出目录相关，如XFS 文件系统)中创建子 目录的函数完成子目录的创建。

对比客户端与服务端对VFS 文件系统函数的调用可以看出，两边都使用了 vfs\_mkdir()函数。因此，我们可以将NFS 理解为实现了将客户端对文件系统的操作 搬到了服务端。

在 Linux 内 核 中 ，NFS 文件系统的整体架构和逻辑还是比较清晰的。主要是 Linux内核同时支持了NFSv2、NFSv3 和 NFSv4 等多个版本，整体比较复杂，但难 度并不是非常大。本节主要介绍了NFS 的整体架构，后续章节将深入介绍其他处 理流程。

**第5章** 基于网络共享的网络文件系统



**5.6.2 RPC协议简析**

前文介绍了NFS 的整体架构，其核心是将客户端的函数调用通过网络传输到 服务端，并转化为服务端的函数调用。其主要实现是客户端与服务端的一一对应的 存根。那么这种转化是如何进行的呢?这就涉及RPC 协议。

虽然5.4.1节介绍了RPC 协议，但主要从概念和功能上对RPC 协议进行了简 要的介绍，并没有深入细节。本节将深入RPC 内部介绍其实现原理。由于目前RPC 协议的具体实现非常多，而且协议细节也不同，因此很难逐一介绍清楚。本节以Sun 公司的RPC 协议为例进行详细介绍，毕竟它是NFS 协议的基础。

RPC 协议与TCP/IP 协议类似，以二进制的方式传输数据。RPC 协议先要解决 的问题是如何将一个客户端的函数调用转换为服务端的函数实现。

另 外 ，Sun 公司的RPC 协议在设计时期望实现对多种服务的支持，如NFS 协 议、挂载协议和NLM 协议等。因此在设计RPC 协议时，有3个相关的字段来进行 标识，其中，Program 字段标识程序，区分NFS 、MOUNT 和 NLM 等其他程序类 型；Program Version字段标识程序版本，考虑升级的兼容性；Procedure 字段标识程 序中的过程(函数),如图5-7所示。

|  |  |
| --- | --- |
| 4字节 | 4字节 |
|  |  |
| XID | Message Type |
| RPC Version | Program |
| Program Version | Procedure |
|  | |

图5-7 RPC协议数据包格式(局部)

通过上述Program 和 Procedure 等关键信息的讲解，当服务端收到该消息时就 可以知道应该由哪个版本的哪个程序来处理该消息，而且进一步知道应该调用哪个 存根函数(函数指针)来进行处理。

我们通过WireShark 抓包看一看RPC 是如何传输数据的，以及数据的格式。 图5-8所示为抓取的挂载命令的数据包，我们可以对比一下该数据包的内容与协议 的格式。

文件系统技术内幕

大数据时代海量数据存储之道

Frame 31:130 bytes on wire(1040 bits),130 bytes captured (1840 bits)

Ethernet II,Src:00:00:00\_00:00:00(08:00:00:00:00:00),Dst:00:00:00\_00:00:00 Internet Protocol Version 4,Src: <192.168.2.115>,Dst: <192.168.2.115>

User Datagram Protocol,Src Port:990,Dst Port:54644



Remote Procedure Call,Type:Cal1 XID:0xd8de1c31

XID:0xd8d01c31(3637517361) Message Type:Call(8)

RPC Version:2

Program:MOUNT(100005) Program Version:3

Procedure:MNT(1)

[The reply to this request is in frame 321 Credentials

Verifier

Mount Service

80008e 000000 8888

887426 e74080 4011

8273.03 de d574 8060

ee ee 80 80 0002 0001

39时00 80080100 ee 000a 73 756e 6e 797a 000080808e

88 ee 0080 e0088880

6673

80 00800088004580

8d 5b ce a80273 ce a8 86 a8 d8 de 1c 318000 86 a500000e030昵 802401062c 2380.00 68616e 6700000000 800080800188008080.0000 ee 08.2f 73.72762f 6e

t&@ [-s

St 1

· ………5-,



/srv/n

fs

8

18

8028

8838

0840

0850

0860

0878

9988

sunnyz hang

E

图5- 8 抓取的挂载命令的数据包

从图5-8中可以看到，在这个数据包中Program 是100005;Program 版本是3, 也就是NFSv3的数据；Procedure 的值为1,也就是挂载操作。由于WireShark是支 持RPC 协议和NFS 协议的，因此可以在其中展示出各个协议的解释信息(图5-8 的上半部分是具体的描述信息，图5-8的下半部分则是原始的数据包数据)。

正是由于在RPC 数据包中包含的这些关键信息，当客户端发送的消息被服务 端接收后，服务端根据这些信息就能知道应该调用哪个存根函数。

**5.6.3 NFS协议简析**

NFS 协议从最初的1.0版本到目前的4.X 版本已经有四大版本，但是1.0版本 只由Sun 公司内部使用，并没有对外开放。从2.0版本开始，Sun 公司开放了NFS 协议，并被其他很多公司使用。

本节将以NFSv3协议为例来介绍一下NFS 协议，选用该协议的原因是其应用 非常多，而又不至于太复杂。当然，如果熟悉了NFSv3 协议，再学习NFSv4 协议 将会比较简单。两者的差别在于前者是无状态的，而后者是有状态的。

上面所述的状态是指文件系统中对象的状态。以文件为例，当客户端访问一个 文件时，NFSv3 协议在服务端并不会维护该文件的状态。也就是说，当在客户端打 开一个文件时，在服务端其实并没有对应的动作。而当向该文件写入数据时，服务 端才真正地打开文件并写入数据 ， 完成写入数据后自动关闭文件 。

第5章基干网络共享的网络文件系统

对 于NFSv4 协议，当在客户端打开一个文件时，服务端也会对应着打开一个 文件；当写入数据时，会被写入已经打开的文件，完成后不会关闭该文件。只有等 到客户端调用关闭文件的接口时服务端才会关闭该文件。

另外，NFSv3 实际上有3个独立的协议：第1个是文件系统访问协议，它是对 文件系统常规的“增加”"删除”“修改”“查询";第2个是对文件系统进行挂载和 卸载操作的协议，即挂载协议；第3个是网络锁协议。

[**5.6.3.1**](5.6.3.1) **挂载** **(MOUNT) 协议**

任何文件系统在使用之前都先要挂载到客户端，网络文件系统自然也不例外。 NFS 协议在早期有一个独立的挂载协议。挂载协议相对简单，共有6个命令，如 表5-3所示。

**表5-3挂载协议过程列表**

|  |  |  |
| --- | --- | --- |
| 名 称 | 过 程 编 码 | **说** **明** |
| NULL | 0 | 空操作，什么都不做 |
| MNT | 1 | 挂载文件系统 |
| DUMP | 2 | 显示挂载项列表 |
| UMNT | 3 | 卸载一个文件系统 |
| UMNTALL | 4 | 卸载所有文件系统 |
| EXPORT | 5 | 显示导出的文件系统列表 |

在表5-3中，最主要的是各个过程的编码，当然除了过程编码，还有一些参数 信息。这里的过程编码虽然定义在NFS 协议中，但实际上是给RPC 协议使用的。 在介绍RPC 协议时，我们以MNT 为例进行了介绍，并且抓取了实际的网络数据 包，网络数据包中的过程(Procedure) 其实就是表5-3中对应的过程编码。

在 NFS 的整个协议中，文件和目录都是通过文件句柄来标识的。在本地文件 系统中挂载过程是从磁盘上找到根目录的信息，NFS 协议逻辑与此类似，它是通过 MNT请求让服务端返回一个根目录的文件句柄。

我们以挂载请求为例，当用户执行挂载命令时，其实核心的内容是客户端向服 务端发送了挂载(MNT) 数据包。图5-9所示为抓取的挂载请求网络数据包。

从图5-9中可以看出，对于挂载过程只有一个参数，也就是要挂载的路径。路 径是用XDR 协议来表示的，其前面4字节表示字符串的长度，后面才是真正的路 径内容。

**文件系统技术内幕**

**大数据时代海量数据存储之道**

Frame 31:130 bytes on wire(1040 bits),130 bytes captured (1048 bits)

Ethernet II,Src:80:00:00\_00:00:00(00:00:00:00:00:08),Dst:00:00:00\_00:00:08 Internet Protocol Version 4,Src:<192.168.2.115>,Dst:<192.168.2.115>

User Datagram Protocol,Src Port:990,Dst Port:54644



Remote Procedure Call,Type:Call XID:0xd8d01c31

Mount Service

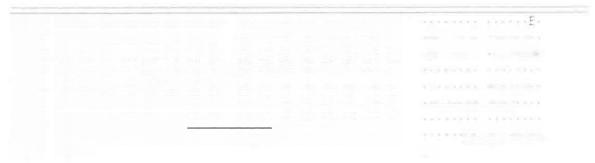
[Program Version:3]

[V3 Procedure:MNT(1)]

Path:/srv/nfs

length:8

contents:/srv/nfs



8000800080000000 008e 808088004588

0074.26 e7401884011 8d 5b c0 a80273c0 a8

8273 03 de ds 748060 86 a8 d8 de 1c 31:8080 800880008002000186 a500.0000030e 00 8001800088 e10080902401062c 238000 800a 73756e 6e 797a 68616e 6700080080 ee ee 008e ee ee e⁰0e ee 0100 ee ee ee e0 ee ee 88.0e ee ee ae 80 ee 8e E 2f 7372762f 6e 6673

8000

0018

0828

8838

B840 0858 806

0070

0880

t8-@@[·s st 1

,#

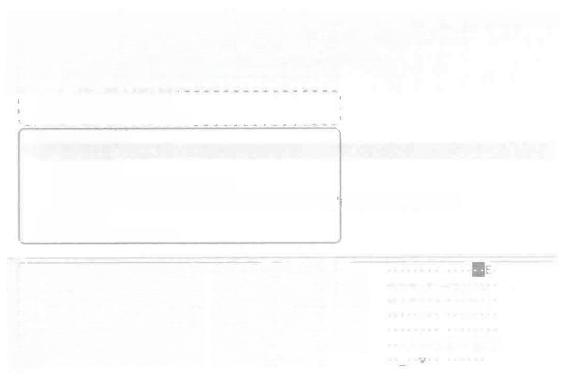
sunnyz hang

/srv/n

fs

图 5 - 9 抓 取 的 挂 载 请 求 网 络 数 据 包

前面只是发送MNT 请求的过程，每个请求都会有一个应答。当服务端收到挂 载数据包时会进行解析，最终调用服务端注册的挂载处理函数进行处理。该函数完 成处理后会发送一个应答数据包。图5-10所示为获取的MNT 应答数据包。对于 MNT 应答来说，主要包含的内容是状态、句柄和一些其他特性信息。

Ethernet II,Src:00:00:00\_00:00:00(00:00:00:00:80:00),Dst:00:08:00\_00:08:00( Internet Protocol Version 4,Src: <192.168.2.115>,Dst: <192.168.2.115>

User Datagram Protocol,Src Port:54644,Dst Port:998

Remote Procedure Call,Type:Reply XID:0xd8d01c31

Mount.Servige

[Program Version:3]

IV3 Procedure:MTS1] Status:OK(0)

fhandle

length:28

[hash(CRC-32):0xd6318dad]

FileHandle:01008700cbe0000000900880f966217-1cab4f65a833acc7- Flavors:1

Flavor:AUTH\_UNIX(1)

08000 80800080 00 800880 08008808 38的4588

ee10 086026 e8 48004811 8d 6e ce a80273ce a8 @ 母 n-[5](#bookmark234)

0028 0273d5 7403 de 004c 8694 d8 de 1c 318008 stL [1](#bookmark235)

038 088188 800000 88 88 808e 8088 00 ea 0800

8e0 800000 8e 0080 ee080 801e e100.0780 cb [00](#bookmark236)

005988808080008e f966 217e Ie ab 4F 65 a833 f Oe [3](#bookmark237)

8060 ac c75f 84 cb 768000 800100808081

图 5 - 1 0 获 取 的MNT 应 答 数 据 包

由于挂载操作可能成功，也可能失败，因此通过状态信息进行描述。在服务端 处理挂载请求成功的情况下，会为根目录生成一个文件句柄，该文件句柄会返回客 户端。客户端的后续请求都以该文件句柄为基础。

**第5章** 基于网络共享的网络文件系统



需要说明的是，在图5-10中，虚线框中的内容并不是NFS 协议数据包的内容， 而是RPC 协议中的内容。这里使用WireShark 工具将其展示出来是为了使信息更 加易读。

本节以挂载过程为例介绍了挂载协议的数据包格式和主要数据域。在挂载协议 中，除了挂载过程，还有卸载过程等另外5个过程，不过原理大同小异，本节不再 赘述，大家可以自行阅读相关协议或在环境中抓包分析。

[**5.6.3.2**](5.6.3.2) **访问** **(NFS) 协议**

完成文件系统的挂载后就可以访问文件系统的数据，这时就需要使用NFS 协 议了。由于NFS 协议主要完成数据的访问操作，这里简称为访问协议。访问协议 是 NFS 协议族最核心的部分，这里的内容才是实现文件系统访问的必需命令，如 表5-4所示。

**表5-4访问协议过程列表**

|  |  |  |
| --- | --- | --- |
| 名 称 | 过 程 编 码 | **说** **明** |
| NULL | 0 | 空操作 |
| GETATTR | 1 | 获取文件或目录属性 |
| SETATTR | 2 | 设置文件或目录属性 |
| LOOKUP | 3 | 查找文件或目录 |
| ACCESS | 4 | 检查访问权限 |
| READLINK | 5 | 从链接读取数据 |
| READ | 6 | 从文件读取数据 |
| WRITE | 7 | 向文件写入数据 |
| CREATE | 8 | 创建文件 |
| MKDIR | 9 | 创建目录 |
| SYMLINK | 10 | 创建符号链接 |
| MKNOD | 11 | 创建特殊设备 |
| REMOVE | 12 | 删除文件 |
| RMDIR | 13 | 删除目录 |
| RENAME | 14 | 修改文件或目录名称 |
| LINK | 15 | 创建硬链接 |
| READDIR | 16 | 遍历目录 |
| READDIRPLUS | 17 | 遍历目录(扩展) |
| FSSTAT | 18 | 获取文件系统的动态信息 |
| FSINFO | 19 | 获取文件系统的静态信息 |
| PATHCONF | 20 | 返回POSIX信息 |
| COMMIT | 21 | 刷写客户端缓存 |

文件系统技术内幕

大数据时代海量数据存储之道

本节以写过程为例介绍一下其主要的参数，并结合实际数据包进行分析。对于 写过程，在RFC1813 中的定义如下：

WRITE3res NFSPROC3\_WRITE(WRITE3args)=7;

通过上述定义可以看出，NFS 协议对过程的定义类似函数的形式。其中， WRITE3args 为过程的参数，而WRITE3res 则是其返回值。后面的数字7是过程的 ID。在实际通信时，服务端正是通过该ID 来找到对应的处理程序的。

接下来看一看该过程的参数，主要包括句柄、逻辑偏移、长度和具体的数据。 对比前面介绍的write() 函数可以看出，NFS 协议的写过程与文件系统API write非 常类似。当向文件写数据时，必须要告诉文件系统要在哪个文件写数据，写到文件 的什么位置，大小是多少及数据是什么。

NFS 协议的WRITE 过程的定义如下：

struct WRITE3args{ nfs\_fh3 file;

offset3 offset;

count3 count;

stable\_how stable;

opaque data◇; };

图5-11所示为通过WireShark 工具获取的数据包，可以找到NFS 协议相关的 文件句柄(FileHandle) 、偏 移(offset) 、大 小(count) 、稳定性(Stable) 和数据(Data) 等 内 容 。

蠹器

的 思



Fra 18:218 bytes on Mtre(1744 bits),218 bytes captured (1744 bits)

Internet Protocol Version 4,Sre: <192.168.43.237>,Dst: <192.168.43.237>

Transalsslan Contral Protocol,See Port:734,Dst Port 2049,Sea:841,Ack

fesote Procedure Call,Type:Call X1D:0x6711236

m Gl acz29 w ProereIE(71

length:36

[hash(CRC-32):Qxac2te259]

Fuletundle:01808781cbeB00e0000601966217elcab4F65aeJ3acc7-

offset:a

count:18

S Sk (a) length:10

contents:<DATAy

f11 bytes:opaque dat

55.7b 4ea048.06 6  2b ed 87 de 8801 ba c 2a ee52B

8200d909000e 0101088a61ab [48](#bookmark239)

40ce 888000.946712360eoe 00 ee oe [00](#bookmark240)

098200018543808e ee83.00.0080 e7.8e [00](#bookmark241)

ee ea 8024.018629 008e 00a刀 7 5

68.616e67 ee 00 ae0000

e00000006e a1 oe se 8ee eeape ee ae00

0701 cbee 8oeeo00e

66 7e Ic

80d208000的0ee e000

Etheret,Sre:68:00:00\_00:00:e(e:00:00:00:00:00),Dst:00:00:0\_00:

oe80000 D

80k 8000

880024810e

图5-11 通过WireShark 工具获取的数据包

**第5章** 基于网络共享的网络文件系统



所以，NFS 协议的定义还是挺清晰的，也很容易理解。需要说明的是稳定性 (Stable)参数，该参数用于告知服务端数据是否需要在服务端持久化，也就是写入 磁盘。如果该参数的值是0(UNSTABLE) 则表示将数据写入服务端缓存即可；如 果该参数的值非0则有DATA\_SYNC和FILE\_SYNC 两种情况，都需要将数据写入 磁盘，差异是对文件系统元数据的处理。

当 然 ，WRITE 过程也是有应答的，通过应答客户端才知道其所发送的请求是 否执行成功。这部分内容比较简单，本节不再赘述。

[**5.6.3.3**](5.6.3.3) **锁** **(NLM)** **协议**

由于NFSv2 和 NFSv3 版本的协议是无状态的，这样也就无法维护文件锁的状 态。因此，在NFS 协议族中有一个专门的网络锁管理(Network LockManager,简 称NLM) 协议。

NLM 是文件锁的网络版。本地文件系统可以在文件系统内实现文件锁。但由 于网络文件系统会有多个不同的客户端文件系统访问同一个服务端的文件系统，文 件锁是无法在客户端的文件系统中实现的，只能在服务端实现。这样就需要一个协 议将客户端的加锁、解锁等请求传输到服务端，并且在服务端维护文件锁的状态。

由于NFS 的文件锁是跨客户端与服务端的，因此场景就变得复杂很多。特别是 服务端宕机、客户端宕机和网络分区等几种异常场景是必须要考虑的。以客户端宕 机为例，如果持有锁的客户端宕机，这样就没有机会释放锁。如果设计时不考虑这 种情况，则可能致使其他客户端永远无法获得锁，进而导致死锁的现象。

在NLM 协议中主要定义了一些过程，包括加锁、解锁和获取资源等20多个 过程。本节不再介绍这些过程的内容，大家可以自行阅读RFC 协议白皮书。

**5.6.4 NFS协议的具体实现**

通过前文我们知道Sun 公司在实现NFS 文件系统时进行了分层处理，底层实 现了RPC 协议，而在RPC 协议的上层实现了NFS 协议。通过分层，简化了NFS 协议的实现。本节将结合Linux 介绍一下NFS 协议的实现细节。

[**5.6.4.1**](5.6.4.1) **内** **核RPC** **协议处理流程分析**

RPC 协议承载了NFS 协议，因此在介绍实现NFS 协议的具体流程之前，先简 单介绍一下Linux 内核中RPC 协议的实现。在Linux 内核中，RPC 是一个独立的 内核模块，位于网络子目录中。RPC 模块为NFS 提供了基本的API, 包括客户端

**文件系统技术内幕**

大数据时代海量数据存储之道

的API函数和服务端的API函数。表5-5所示为Linux内核RPC提供的主要API 函数。

**表5-5** **Linux内核RPC提供的主要API函数**

|  |  |
| --- | --- |
| **函** **数** **名** **称** | **说** **明** |
| rpc\_create() | 创建一个RPC客户端，返回类似文件句柄 |
| rpc\_call\_sync() | 在客户端执行一个同步的RPC调用 |
| rpc\_call\_async() | 在客户端执行一个异步的RPC调用 |
| svc\_create() | 创建一个RPC服务端，单线程模式 |
| sve\_create \_pooled( | 创建一个RPC服务端，线程池模式 |
| sve\_recv() | 服务端接口，接收来自客户端的请求 |
| sve process( | 服务端接口，处理来自客户端的请求 |

表5-5中的接口分为两部分： 一部分是客户端的接口；另一部分是服务端的接 口。在客户端通常调用rpc\_create() 函数创建一个客户端的结构体指针(rpc\_clnt),

该指针类似文件句柄或套接字。完成客户端指针创建后，客户端程序就可以通过该 指针来向服务端发送RPC 请求了，具体涉及rpc\_call\_sync() 和rpc\_call\_async() 两个 函数，分别用于发送同步和异步请求。

客户端的请求都是通过rpc\_call\_syncO 和rpc\_call\_async) 两个函数来实现与服 务端交互的。客户端存根函数依照 NFS 协议准备必要的参数，然后调用 rpc\_call\_sync() 函数或rpc\_call\_async() 函数来向服务端发送请求。以创建目录为例， 在存根函数 nfs3\_proc\_mkdir ( 中 根 据NFS 协议完成目录句柄、子目录名称和属性 等参数的初始化(第575行~第579行),然后调用nfs3\_do\_create() 函数，该函数

实际调用的是RPC 模 块 的rpc\_call\_sync() 函数，如代码5-2所示。 代码5-2 nfs3\_proc\_mkdir()函数的实现

**net/sunrpc/nfs3proc.e**

|  |  |
| --- | --- |
| 557  558  559  575  576  577  578  579  580  581  582  600 | static int  nfs3 proc\_mkdir(struct inode \*dir,struct dentry\*dentry,struct iattr \*sattr) {  /删除部分代码  data->msg.rpe\_proc=&nfs3\_procedures[NFS3PROC\_MKDIR]; data->arg.mkdir.fh=NFS\_FH(dir);  data->arg.mkdir.name=dentry->d\_name.name; data->arg.mkdir.len=dentry->d\_name.len;  data->arg.mkdir.sattr=sattr;  d\_alias=nfs3\_do\_create(dir,dentry,data);//nfs3\_do\_create()函数内部调用了rpe\_call\_sync(函数 status=PTR\_ERR\_OR\_ZERO(d\_alias);  //删除部分代码 } |

第 5 章 基 干 网 络 共 享 的 网 络 文 件 系 统

其他存根函数的逻辑与nfs3\_proc\_mkdir() 函数的逻辑类似，都是根据协议创建 需要的参数，然后直接或间接调用rpc\_call\_sync() 函数或rpc\_call\_async() 函数来将 请求发送到服务端。

接下来深入 RPC 模块的内部，看一看消息是如何被编码并发送的。以同步接 口为例，对rpc\_call\_sync (函数进行基本参数的封装，然后调用rpc\_run\_task() 函 数 运行一个RPC 任务(第1173行),如代码5-3所示。异步接口与此类似，也是调用 rpc\_run\_task() 函数来运行一个RPC 任务。

**代码5** **-** **3** **rpc** **call** **sync()函数的实现**

**net/sunrpc/clnt.e** **nfs3\_proc\_mkdir->nfs3\_do\_create->rpc\_call\_sync**

|  |  |
| --- | --- |
| 1155  1156  1157  1158  1159  1160  1161  1162  1163  1164  1165  1166  1167  1168  1169  1170  1171  1172  1173  1174  1175  1176  1177  1178  1179 | int rpc\_call\_sync(struct rpc\_clnt \*clnt,conststruct rpc\_message\*msg,int flags) {  struct rpc\_task \*task;  structrpc\_task\_setup task\_setup\_data={ rpc\_client=clnt,  rpc message=msg,  .callback\_ops=&rpc\_default\_ops, .flags=flags,  :  int status;  WARN\_ON\_ONCE(flags&RPC\_TASK\_ASYNC); if (flags &RPC\_TASK\_ASYNC){  rpc\_release\_calldata(task\_setup\_data.callback\_ops, task\_setup\_data.callback\_data);  returmn -EINVAL; —  task=rpc\_run\_task(&task\_setup\_data); if (IS\_ERR(task))  return PTR\_ERR(task); status=task->tk\_status;  rpe\_put\_task(task);  return status; } |

rpc\_run\_task() 是运行一个RPC 任务的函数，该函数主要是创建一个任务(task),

然 后 调 用 rpc\_execute() 函数来执行任务。在对任务初始化的过程中完成了对 tk\_action() 函数的初始化，这个就是在某个状态时执行的动作(action), 如代码5- 4 所示。

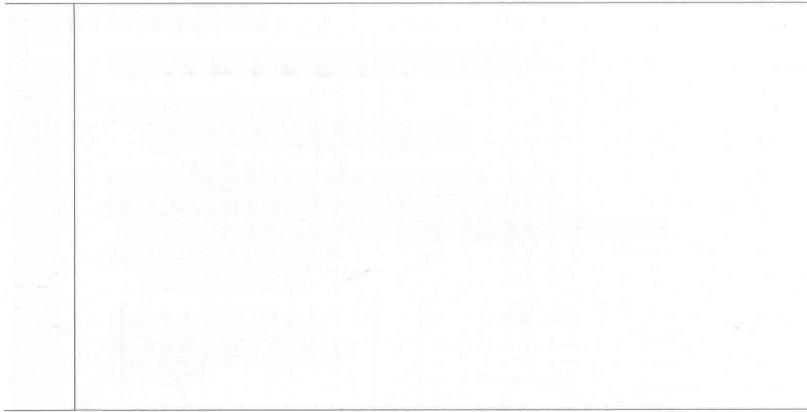
**代码5-4** **rpc\_run\_task()函数的实现**

n**et/sunrpc/clnt.e** **nfs3** **proc** **mkdir->nfs3** **do** **create->rpc** **call** **sync->rpc** **run** task

|  |  |
| --- | --- |
| 1128  1129 | struct rpc\_task\*rpc\_run\_task(const struct rpc\_task\_setup \*task\_setup\_data) |

文件系统技术内幕

大数据时代海量数据存储之道



struct rpc\_task \*task;

task=rpe\_new\_task(task\_setup\_data);// 新 建 任 务 结 构 体

f(!RPC\_IS\_ASYNC(task))

task->tk\_flags|=RPC\_TASK\_CRED\_NOREF;

rpc\_task\_set\_client(task,task\_setup\_data->rpc\_client);

rpc\_task\_set\_rpc\_message(task,task\_setup\_data->rpc\_message);

/如果没有初始化动作，将初始动作初始化为start, 这在后面状态机中使用

if(task->tk\_action=NULL) rpc\_call\_start(task);

atomic\_inc(&task->tk\_count);

rpc\_execute(task);// 执行任务

return task;

1130

1131

1132

1133

1134

1135

1136

1137

1138

1139

1140

1141

1142

1143

1144

1145

1146



rpc\_execute) 函数并不一定马上执行任务，这要根据是同步任务还是异步任务 而定。如果是同步任务，则rpc\_execute() 函数会调用\_rpc\_execute() 函数执行任务； 如果是异步任务，则将任务放入队列中，如代码5-5所示。

代码5-5 rpc execute()函数的实现

**net/sunrpc/sched.c**

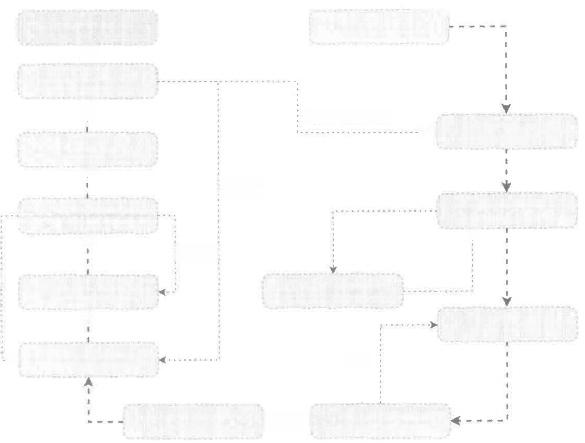
|  |  |
| --- | --- |
| 984 | void rpc\_execute(struct rpc\_task \*task) |
| 985 |  |
| 986 | bool is\_async=RPC\_IS\_ASYNC(task); |
| 987 |  |
| 988 | pc\_set\_active(task);  //如果是异步任务，则将任务放入队列中 |
| 989 | rpc\_make\_runnable(rpciod\_workqueue,task); |
| 990 | if(!is\_async)  //如果是同步任务，则rpc\_execute(函数会调用\_rpc\_execute()函数执行任务 |
| 991 | rpe\_execute(task); |
| 992 |  |

\_rpc\_execute()函数内部核心是for循环，这就是前文提到的状态机的实现。状 态机的实现原理是在for 循环中不断地执行任务中的 tk\_action 函数指针。在执行 tk\_action 函数指针时更新任务中tk\_action 的值，从而实现状态的转换。当tk\_action 的值更新为NULL 时，说明没有新的状态，此时退出状态机。

在本实例中，任务在初始化时调用rpc\_call\_start) 函数完成了动作 (tk\_action) 的初始化，该动作函数为call\_start() 。 然后在状态机中会调用call\_start() 函数，而该 函数除了完成其基本功能，还会将任务中 tk\_action 的值更新，也就是更新为 call\_reserve。这样，当进行下次循环时就会执行call\_reserve(函数。依次类推，就 可以完成整个状态的切换。图5- 12所示为RPC 发送消息状态转换图。

第 5 章 基 干 网 络 共 享 的 网 络 文 件 系 统

EBADMSG



call\_starto



call\_decode0

EKEYREJECTED

→ call\_reserveo

call\_statusO A

EAGAIN

call\_reserveresulto

A

EAGAIN:

call\_retry\_reserveo

call\_refresho

 call\_encodeo 失败

call\_allocateo ----call\_refreshresulto

call\_transmito 个

call\_transmit\_status0

rpc\_exit\_task()

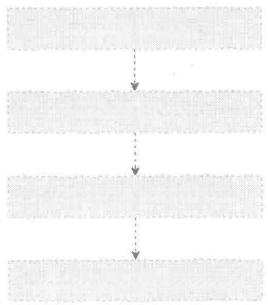
个



图5-12 RPC 发送消息状态转换图

这里状态机实现的对任务各个状态的转换其实就是处理任务的不同阶段，如编 码、发送消息和解码等。以发送消息为例，主要业务逻辑在call\_transmit() 函数中实 现。图5-13所示为call\_transmit (函数调用具体 xprt 处理函数的主线流程，最终调 用xprt 的 send\_request 函数指针。这里xprt 是指具体的数据传输协议类型，可以是 TCP协议或UDP协议等。

以 TCP 协议为例，函数指针send\_request 是 在xs\_tcp\_send\_request() 函数中实 现 的 。xs\_tcp\_send\_request() 函数会调用xprt\_sock\_sendmsg() 函数进行数据发送，最 终调用内核socket 的 sock\_sendmsgO 函数将数据发送到服务端。



call\_transmit()

xprt\_transmit(

xprt\_request\_transmit0

xprt->ops->send\_request)

图5-13 call\_transmit()函数调用具体xprt处理函数的主流程

文件系统技术内幕

大数据时代海量数据存储之道

[**5.6.4.2**](5.6.4.2) **NFS协议挂载流程分析**

任何文件系统在使用之前都需要进行挂载，网络文件系统当然也不例外。前文 已述，在NFS 协议中挂载的是一个独立的协议，主要实现了挂载、卸载和查看导 出目录等功能。本节将主要介绍一下NFS 协议的挂载流程。

通过对本地文件系统挂载分析，我们知道文件系统挂载的主要动作是从磁盘读 取超级块，然后生成inode 和 dentry 节点，并与挂载点进行结合。网络文件系统的 挂载大概也是如此，但是网络文件系统需要通过网络从服务端获取根目录的信息。

在Linux 内核中实现的NFS 协议是基于RPC 实现的，因此发送消息的过程是 调用的RPC 的接口。代码5-6是挂载流程中发送消息的函数实现，该函数的主要 功能是封装消息，然后调用RPC 服务的API(rpc\_call\_sync() 函数将消息发送到 服务端进行处理，具体内部实现细节请参考代码5-6及其中的注释。

**代码5-6** **nfs\_mount()函数的实现**

**fs/nfs/mount\_clnt.c**

|  |  |
| --- | --- |
| 145 | int nfs\_mount(struct nfs\_mount\_request \*info) |
| 146 | { |
| 147 | structmountres result={ |
| 148 | .fh =info->fh, |
| 149 | .auth\_count =info->auth\_flav\_len, |
| 150 | .auth\_flavors =info->auth\_flavs, |
| 151 | };/用于存储从服务端返回的内容 |
| 152 | struct rpc\_message msg={ |
| 153 | .rpc\_argp =info->dirpath, |
| 154 | .rpc\_resp =&result, |
| 155 | };//请求消息体，主要包含路径信息，也就是期望挂载的路径 |
| 156 | structrpc\_create\_args args={ |
| 157 | .net =info->net, |
| 158 | protocol =info->protocol, |
| 159 | .address =info->sap, |
| 160 | addrsize =info->salen, |
| 161 | .servername =info->hostname, |
| 162 | program =&mnt\_program, |
| 163 | .version =info->version, |
| 164 | authflavor =RPC\_AUTH\_UNIX, |
| 165 | .cred =current\_cred(, |
| 166 | }; |
| 167 | struct rpc\_clnt \*mnt\_clnt; |
| 168 | int status; |
| 169 |  |
| 170 | dprintk("NFS:sending MNT request for%s:%s\n", |
| 171 | (info->hostname?info->hostname:"server"), |
| 172 | info->dirpath); |
| 173 |  |

第 5 章 基 干 网 络 共 享 的 网 络 文 件 系 统

174 if(strlen(info->dirpath)>MNTPATHLEN)

175 return -ENAMETOOLONG;

177 f(info->noresvport)

args.flagsl=RPC\_CLNT\_CREATE\_NONPRIVPORT;

180 mnt\_clnt=rpc\_create(&args);

181 if(IS\_ERR(mnt\_clnt))

182 goto out\_clnt\_er;

/ 使 用 挂 载(MNT) 函数处理填充消息

184 if(info->version=NFS\_MNT3\_VERSION)

185 msg.rpc\_proc=&mnt\_clnt->cl\_procinfo[MOUNTPROC3\_MNT]; 186

187 msg.rpc\_proc=&mnt\_clnt->cl\_procinfo[MOUNTPROC\_MNT]; 188 //调用PRC 的 接 口 ，pe\_call\_sync() 函数发送前面填充的消息

189 status=rpc\_call\_sync(mnt\_clnt,

&msg,RPC\_TASK\_SOFTRPC\_TASK\_TIMEOUT);

190 rpe\_shutdown\_client(mnt\_clnt); //删除部分非关键代码

上述函数调用的是RPC 的同步接口，服务端返回的结果会存储在局部变量 result 中(第147行～第151行)。通过mountres 结构体的定义和NFS 协议的介绍， 我们知道挂载操作主要从服务端获取根目录的句柄信息，这个句柄在后续的操作中 都会用到。

在nfs\_mount() 函数中，需要说明的是消息的初始化，这里除了需要初始化必要 的参数，还要对rpc\_proc 成员进行初始化。该成员是一个结构体，它不仅包含注册 具体的例程ID, 还包括进行数据编码和解码的处理函数等内容，如代码5-7所示。

**代码5-7** **rpc\_procinfo结构体**

**fs/nfs/mount\_clnt.c**

|  |  |
| --- | --- |
| 490 | static const struct rpc\_procinfo mnt3\_procedures[]=( |
| 491 | [MOUNTPROC3\_MNT]={ |
| 492 | P\_proc =MOUNTPROC3\_MNT,//例程ID |
| 493 | p\_encode =mnt\_xdr\_enc\_dirpath,//参数编码函数 |
| 494 | .p\_decode =mnt\_xdr\_dec\_mountres3,//解码函数 |
| 495 | p\_arglen =MNT\_enc\_dirpath\_sz, |
| 496 | P\_replen MNT\_dec\_mountres3\_sz, |
| 497 | -P statidx MOUNTPROC3\_MNT, |
| 498  499 | p\_name ="MOUNT", |
| 500  501  502  503  504  505  506 | ,  [MOUNTPROC3\_UMNT]={  P\_proc MOUNTPROC3\_UMNT,  P\_encode mnt\_xdr\_enc\_dirpath,  p\_arglen MNT\_enc\_dirpath\_sz,  p\_statidx =MOUNTPROC3\_UMNT,  p\_name ="UMOUNT", |

文件系统技术内幕

大数据时代海量数据存储之道

507

片 ；,

[**5.6.4.3**](5.6.4.3) **NFS协议读/写数据流程分析**

有了前面本地文件系统相关章节的介绍，学习NFS 客户端文件系统的读/写流 程就没那么困难了。对于NFS 来说，也包含同步写、异步写和直接写等模式，关于 这部分内容与本地文件系统没有差异。

为了实现与VFS 的 对 接 ，NFS 也要实现一套函数指针接口，以文件相关的操 作为例，其实现的函数指针如代码5-8所示。对于写数据来说，VFS 会调用NFS 的 nfs\_file\_write()函数。

**代码5-8** **NFS文件系统函数指针**

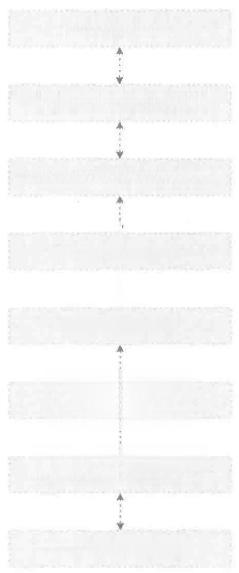
fs/nfs/file.e

|  |  |
| --- | --- |
| 842 | const struct file\_operations nfs\_file\_operations={ |
| 843 | llseek =nfs\_file\_llseek, |
| 844 | read\_iter =nfs\_file\_read, |
| 845 | .write\_iter =nfs\_file\_write, |
| 846 | .mmap =nfs\_file\_mmap, |
| 847 | .open =nfs\_file\_open, |
| 848 | .flush =nfs\_file\_flush, |
| 849 | release =nfs\_file\_release, |
| 850 | .fsync =nfs\_file\_fsync, |
| 851 |  |
| 852 | .lock =nfs\_lock, |
| 853 | flock =nfs\_flock, |
| 854 | splice\_read =generic\_file\_splice\_read, |
| 855 | .splice\_write =iter\_file\_splice\_write, |
| 856 | .check\_flags =nfs\_check\_flags, |
| 857 | .setlease =simple\_nosetlease, |
| 858 | }; |
| 859 | EXPORT\_SYMBOL\_GPL(nfs\_file\_operations); |

在nfs\_file\_write (函数中，如果有SYNC 标记则会触发同步写的流程，否则写 入缓存后就会返回给调用者。在本节中，我们主要关注触发同步写的流程，也就是 数据是如何从NFS 文件系统发送到服务端的。

直接写和同步写都会触发将数据发送到服务端的流程，本节以同步写为例介绍 数据是如何发送到服务端的。如果触发同步写，则会调用nfs\_file\_fsync() 函数，该 函数可以将缓存数据传输到服务端的入口，如图5-14所示。

第5章基干网络共享的网络文件系统



nfs\_file\_fsync()

tile\_write\_and\_wait\_range()

\_filemap\_fdatawrite\_range()

\*

do\_writepages()

nfs\_writepages()



switepesclack0

nfs\_do\_writepage()

本 一 乘

图 5 - 1 4 nfs\_file\_fsync() 函 数 的 主 线 流 程

这里nfs\_do\_writepage() 函数用于将一个缓存页发送到服务端，具体实现如代 码5-9所示。其中，主要功能由nfs\_page\_async\_flush() 函数完成。这里比较重要的 参数是pgio, 在该参数中有页数据传输相关的函数指针，关于该参数类型的详细定 义请参考内核源代码。

**代码5-9** **nfs\_do\_writepage()函数的实现**

**net/sunrpe/nfs3proc.e**

649 static intnfs\_do\_writepage(structpage \*page,struct writeback\_control\*wbc,

650 struct nfs\_pageio\_descriptor \*pgio) 651

652 int ret;

653

654 nfs\_pageio\_cond\_complete(pgio,page\_index(page));

655 ret=nfs\_page\_async\_flush(pgio,page);

656 if(ret=-EAGAIN){

657 redirty page\_for\_writepage(wbc,page);

658 ret=AOP\_WRITEPAGE\_ACTIVATE;

659 

660 return ret;

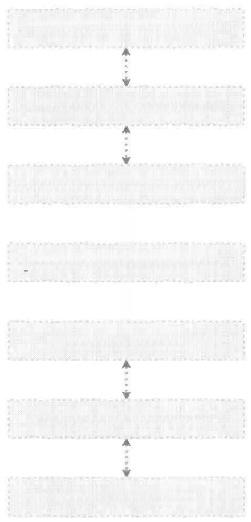
661 

**文件系统技术内幕**

大数据时代海量数据存储之道

nfs\_page\_async\_flush()函数的主线流程如图5-14所示。nfs\_generic\_pg\_pgios() 函数就是pgio 初始化的函数指针，其在nfs\_pageio\_doio() 函数中被调用。该主线流 程最终调用nfs\_initiate\_pgio (函数，该函数完成 PRC 消息和参数的封装后，调用 RPC服务的API 函数完成请求。

…



nfs\_page\_async\_flush(

nfs\_pageio\_add\_request0

nfs\_pageio\_add\_request\_mirror0

\_nfs\_pageio\_add\_request)

nfs\_pageio\_doio0

nfs\_generic\_p9\_pgios)

nfs\_initiate\_pgio()

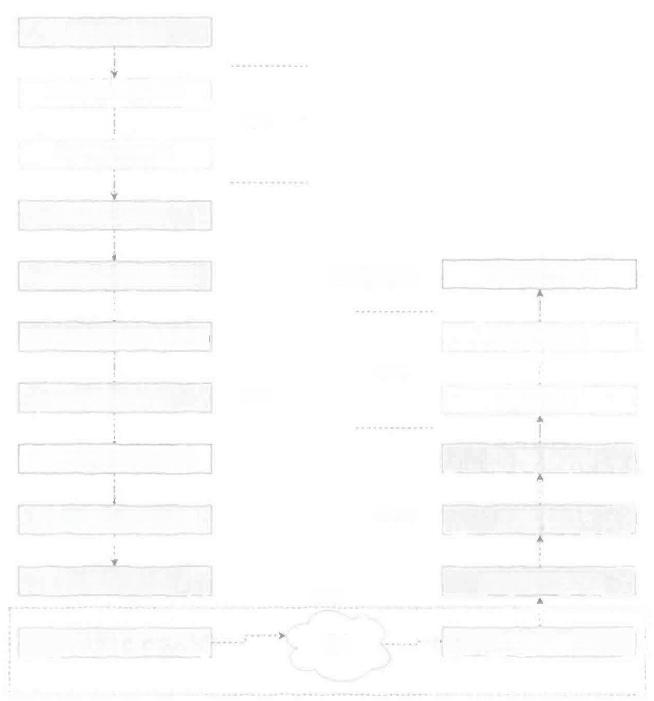
… …

图5-15 nfs\_page\_async\_flush(函数的主线流程

当nfs\_initiate\_pgio() 函数调用rpc\_run\_task() 函数之后，整个流程就进入RPC 服 务内部，也就是进入RPC 服务状态机的流程。

图5-16所示为NFS 写数据的整体流程。服务端向RPC 注册了各种回调函数， 当接收到客户端的请求时会调用具体的回调函数进行处理。本实例将调用 nfsd3\_proc\_write() 函数，该函数最后调用VFS 层的写数据函数，而VFS 层的写数 据函数则调用具体文件系统(如Ext4) 的函数完成最终的写数据操作。

第 5 章 基 干 网 络 共 享 的 网 络 文 件 系 统

nfs\_file\_write0

generic\_write\_synco



vfs\_fsync\_rangeo

nfs\_writepageso



nfs\_pageio\_complete()

Y

nfs\_pageio\_complete\_mirroro



nfs\_pageio\_doio0



desc->pg\_ops->pg\_doio0



nfs\_generic\_pg\_pgios0

nfs\_initiate\_pgio0

rpc\_run\_task0

VES

具体文件系统

VFS

NFS

NFSD

RPC

网 络

f\_op->wite\_itero

do ter wie

vfs\_writevo

nfsd\_vfs\_write()

nfsd\_write()

nfsd3\_proc\_write0

 sVc\_process)

图5-16 NFS 写数据的整体流程

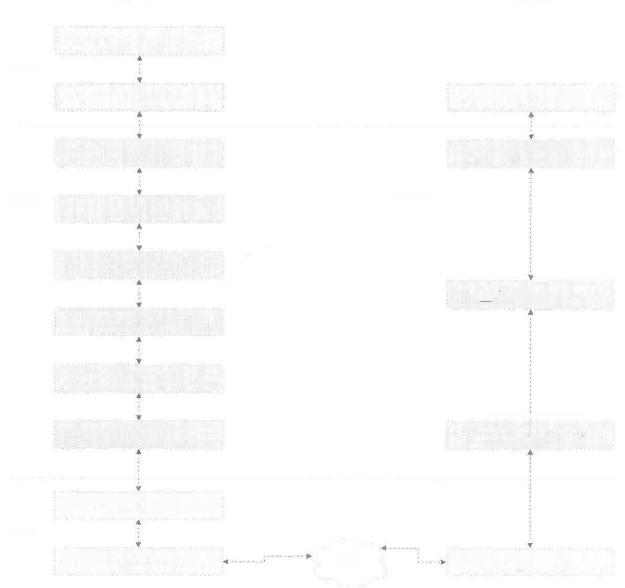
[**5.6.4.4**](5.6.4.4) **NFS协议文件锁实现分析**

通过前文已知NFSv3 及之前版本有独立的锁协议，而NFSv4 之后则通过自有 协议实现锁相关的特性。内核实现的NFS 、NFSv2 、NFSv3 和 NFSv4 相关内容都 在nfs 目录中实现，只不过通过不同的函数指针来调用不同版本的锁功能。对于服 务端来说，文件锁相关的功能在lockd 目录中。

对 于NFS 文件系统来说，在调用锁的API 时会首先经过VFS 层，然后进入 NFS 相关的处理逻辑。图5-17所示为NFS 协议文件锁的流程。

文件系统技术内幕

大数据时代海量数据存储之道

服务端

do\_lock\_fie\_wato

VFS

Vs\_lock\_file0

Vs\_lock\_fie()

nfs\_lockO

nimsvc\_lock0

NFSD

do\_getlk()

NFS

nts3\_proc\_lock(

nlmsw vc\_proc\_lock(

nimcint proc0

nimcnt\_lock0

nimcnt\_cal0

sc pcc0

rpc\_call\_sync()

RPC

rpc\_run\_task0

网 络

Svc\_process)

图5-17 NFS协议文件锁的流程

本节以NFSv3对应的文件锁协议为例介绍一下内核代码是如何实现文件锁的。 文件锁的实现与NFS 协议的读写等操作类似，其主要原理是将客户端的请求传送 到服务端。由于NFSv3 协议中的文件没有状态，因此在服务端需要一个专门的锁 服务来记录这些状态。

对于客户端的流程来说，当用户调用锁相关 API 时会先触发VFS 层中的 vfs\_lock\_file()函数，而该函数则会调用具体文件系统的实现，本实例为NFS 注册 的函数，也就是nfs\_lock() 函数，如代码5-10所示。

**代码5-10** **nfs\_lock()函数的实现**

|  |  |
| --- | --- |
| fs/nfs/file.c | |
| 773  774  798  799  800  801 | int nfs\_lock(struct file \*filp,int cmd,struct file\_lock\*f)  //删除部分代码  if (IS\_GETLK(cmd))  ret=do\_getlk(filp,cmd,fl,is\_local); else if(fl->fl\_type=F\_UNLCK)  ret=do unlk(filp,cmd,fl,is\_local); |

第5章基干网络共享的网络文件系统

802

ret=do\_setlk(filp,cmd,f,is\_local);

out err:

return ret;

806

从上述代码可以看出，nfs\_lock() 函数针对加锁、解锁等有不同的流程。对于加 锁来说是调用do\_getlk() 函数来实现的。接下来则是通过层层调用，最终调用RPC 的 API。

服务端锁服务的主要作用是记录文件的加锁情况。在NFSv3 协议中，每个文

件的加锁情况是通过nlm\_block 数据结构来记录的。同时为了记录文件系统中所有 文件的加锁情况，在锁服务中有一个全局变量nlm\_blocked 来记录所有的nlm\_block 信 息 。

当有来自客户端的加锁请求时，锁服务通过调用nlmsvc\_lookup\_block() 函数从 全局变量nlm\_blocked 中 查 找nlm\_block 实例，该实例记录着文件锁的情况。然后 从该实例中获取锁相关的信息，以该信息作为参数调用VFS 中 的vfs\_lock\_file() 函 数来完成加锁的操作，这与本地文件系统加锁就没有什么差异了。

由此可以看出，对于网络锁来说，本质上要通过nlm\_block 数据结构将文件锁 维护起来，这样在后续加锁、解锁时能够查到该信息即可。

**5.7** **NFS服务端及实例解析**

由 于NFS 本身是一个开放协议，无论是Windows 还 是Linux,NFS 服务端有 很多具体实现。Linux 本身就有一个原生的NFS 服务端软件。Windows 也有许多 NFS 服务端软件，如ProNFS 等，不过很多是商业的，我们无法看到其源代码。

想要深入学习NFS, 还得依赖于Linux 的开源项目。目前，比较流行的有原生 NFS 服务端软件NFSD 和 NFS-Ganesha。本 节 以Linux 内核中原生NFS 服务端软 件为例进行介绍。

**5.7.1 NFSD**

在 Linux 中，有一个NFS 服务端，该服务端由内核态的模块和用户态的守护 进程构成。其中，内核态模块负责数据处理，而用户态守护进程则负责内核态的配 置管理等功能。由于核心功能在内核态实现，因此与Linux 中的本地文件系统有很 好的兼容性，性能也比较好。

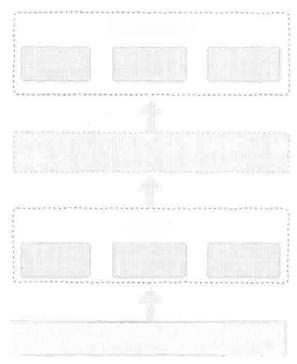
由于网络锁和挂载等协议与NFS 协议不统一，因此都有独立的服务来处理相

文件系统技术内幕

大数据时代海量数据存储之道

关的逻辑，这样整个NFS 服务略显繁杂。但是到NFSv4 协议之后，NFS 协议将网 络锁协议和挂载协议都融入其中，因此具体实现也简洁了。

前文已经简要地描述了NFS 协 议 在Linux 内核中的层次结构。本节详细介绍 一下服务端的软件架构。NFSD 的软件架构并不复杂，其整体架构如图5-18所示。

具体文件系统

XFS

Btrfs

Ext4

VFS

NFSD

NFSv4

NFSv3

NLM

sunrpc

图5-18 NFSD的整体架构

从图5-18可以看出，当RPC 服务收到来自客户端的请求时，它会对请求进行 分发，由具体的程序(如NFS 或 NLM) 来完成相关请求。其中请求的分发依据是 数据包中的程序ID 和 例 程ID, 根据这两个信息就可以找到具体的函数指针。

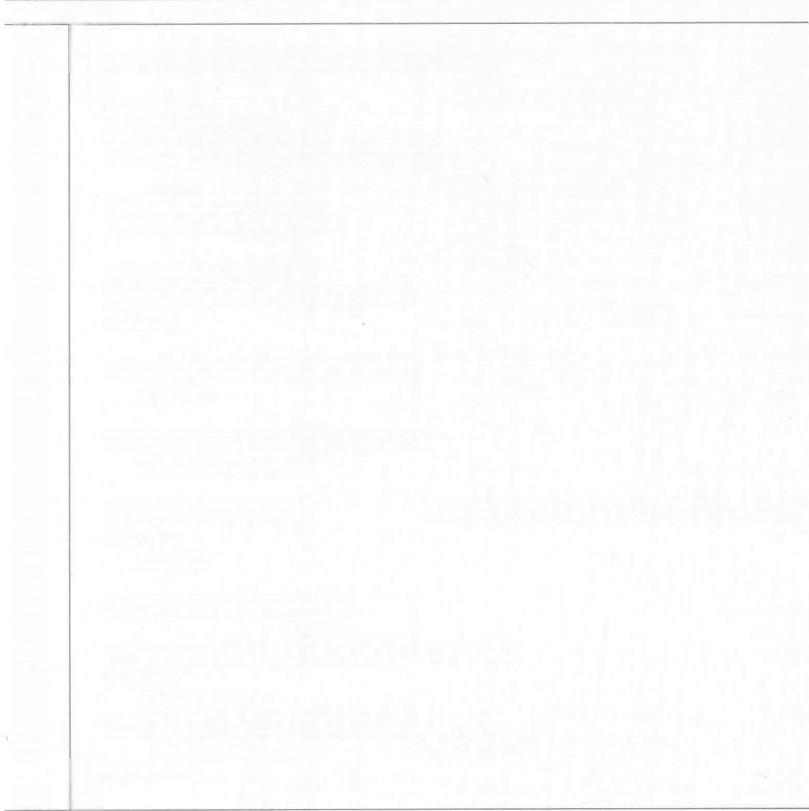
如果相关请求涉及文件操作，那么在例程中会直接调用VFS 的接口(如读数 据)进行下一步的处理。而VFS 则根据导出的目录信息调用本地文件系统(如Ext4 和XFS 等)的接口实现具体的操作。

在内核中实现了所有的NFS 协议，如NFSv3 、NFSv4 、MOUNT 和 NLM 等 。 由于从RPC 服务到协议程序的流程是一样的，限于篇幅我们并不会对每种协议都 做介绍。本节主要以NFSv3 协议为例介绍一下从网络收到消息到最终完成协议层 处理的整个流程。如果大家熟悉了这个流程，再按照此流程来理解其他流程将非常 容易。

首先，我们分析一下NFSD 的启动过程，该过程主要完成函数指针集向RPC 服务注册的过程。以处理NFS 协议的服务端为例，关键是启动了一个线程池。该 内核线程池不断地接收网络消息，译码之后调用注册的回调函数进行具体命令的处 理。代码5- 11是NFS 服务的主函数，函数指针的注册和线程池的创建都在其中 实现。

第 5 章 基 干 网 络 共 享 的 网 络 文 件 系 统

代码5-11 NFS 服务的主函数

**net/nfsd/nfssvc.e**

740 int

741 nfsd\_svc(int nrservs,struct net \*net,const struct cred \*cred) 742 {

743 int error;

744 bool nfsd\_up\_before;

745 struct nfsd\_net \*nn=net\_generic(net,nfsd\_net\_id); 746

747 mutex\_lock(&nfsd\_mutex);

748 dprintk("nfsd:creating serviceln"); 749

750 nrservs=max(nrservs,O);

751 nrservs=min(nrservs,NFSD\_MAXSERVS);

752 error=0;

753

754 if(nrservs==0&&nn->nfsd\_serv=NULL)

755 goto out; 756

757 strlcpy(nn->nfsd\_name,utsname()->nodename,

758 sizeof(nn->nfsd\_name)); 759

760 error=nfsd\_create\_serv(net); / 在nfsd\_create \_serv()函数内部实现函数指针的注册

761 if(error)

762 goto out; 763

764 nfsd\_up\_before=nn->nfsd\_net\_up; 765

766 error=nfsd\_startup\_net(nrservs,net,cred); ; 启 动 网 络

767 if(error)

768 gotoout\_destroy;

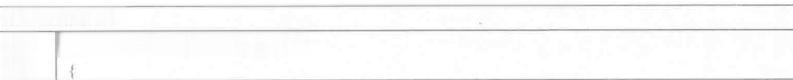
769 error=nn->nfsd\_serv->sv\_ops->svo\_setup(nn->nfsd\_serv,

770 NULL,nrservs); / 启 动 线 程 池 删除部分代码

786 

使 用nfsd\_svc() 函数主要完成三件事，分别是注册函数指针、启动网络和启动 线程池。其中，注册函数指针会完成NFS 协议中的函数指针(nfsd\_program) 和 服 务端的任务处理函数指针(nfsd\_thread\_sv\_ops) 的注册；启动线程池调用的函数是 svc\_set\_num\_threads(), 该 函 数 是RPC 服务的接口，它会以任务处理函数作为线程 函数来启动线程池。

然后，我们看一 下注册函数指针的函数实现，如代码5- 12所示。



代码5-12 nfsd\_create\_serv()函数的实现

net/nfsd/nfssvc.c

609 int nfsd\_create\_serv(struct net \*net)

610

文件系统技术内幕

大数据时代海量数据存储之道

|  |  |
| --- | --- |
| 611  612  613  614  615  616  617  618  619  620  621  622  623  624  625  626  627  628  629  630  631  632  645 | int error;  struct nfsd\_net\*nn=net generic(net,nfsd\_net\_id);  WARN\_ON(!mutex\_is\_locked(&nfsd\_mutex)); if (nn->nfsd\_serv){  sVc\_get(nn->nfsd\_serv); return 0;  if (nfsd\_max\_blksize=0)  nfsd\_max\_blksize=nfsd\_get\_default\_max\_blksize(; nfsd\_reset\_versions(nn);  nn->nfsd\_serv=sVc\_create\_pooled(&nfsd\_program,nfsd\_max\_blksize, &nfsd\_thread\_sv\_ops);/注册函数指针  if(nn->nfsd\_serv=NULL) return -ENOMEM;  nn->nfsd\_serv->sv\_maxconn=nn->max\_connections; error=svc\_bind(nn->nfsd\_serv,net);  if (error<0){  svc\_destroy(nn->nfsd\_serv); return error;  //删除部分代码 上 |

其 中 ，nfsd\_program 和 nfsd\_thread\_sv\_ops 分别是协议函数指针集实例和服务 函数指针集实例。这两个函数指针集的定义如代码5-13和代码5-14所示。

代码5- 13 nfsd\_program协议函数指针集的定义

**net/nfsd/nfssvc.c**

|  |  |
| --- | --- |
| 135 | struct svc\_program nfsd\_program={ |
| 136 | #if defined(CONFIG\_NFSD\_V2\_ACL) ||defined(CONFIG\_NFSD\_V3\_ACL) |
| 137 | pg\_next =&nfsd\_acl\_program, |
| 138 | #endif |
| 139 | Pg\_prog =NFS\_PROGRAM, //程序号 |
| 140 | Pg\_nvers =NFSD\_NRVERS, |
| 141 | Pg\_vers =nfsd\_version, //保存所有NFS版本的信息 |
| 142 | Pg\_name ="nfsd", /程序名称 |
| 143 | Pg\_class ="nfsd", //认证类 |
| 144 | Pg\_stats =&nfsd\_svcstats, |
| 145 | Pg\_authenticate=&svc\_set\_client, /导出认证 |
| 146 | pg\_init\_request =nfsd\_init\_request, |
| 147 | pg\_rpcbind\_set =nfsd\_rpcbind\_set, |
| 148 | }; |

在代码5 - 13中，nfsd\_program 协议函数指针集由全局变量nfsd\_version 定义，

该全局变量中包含多个版本(NFSv2、NFSv3、NFSv4) 的 NFS 函数指针。关于这

部分代码逻辑比较简单，请大家自行阅读相关代码，本节不再赘述。

第 5 章 基 干 网 络 共 享 的 网 络 文 件 系 统

在代码5-14中，nfsd\_thread\_sv\_ops 服务函数指针集主要包括用于处理消息的 线程函数nfsd(和启动线程池的svc\_set\_num\_threads() 函数等。

代码5-14 nfsd thread sv ops服务函数指针集的定义

**net/nfsd/nfssve.e**

|  |  |  |
| --- | --- | --- |
| 596 | static const struct svc\_serv\_ops nfsd\_thread\_sv\_ops={ | |
| 597 | SVo\_shutdown | =nfsd\_last\_thread, |
| 598 | svo\_function | =nfsd, |
| 599 | svo\_enqueue\_xprt | =svc\_xprt\_do\_enqueue, |
| 600 | svo\_setup | =sVc\_set\_num\_threads, |
| 601 | svo\_module | =THIS\_MODULE, |
| 602 | ; |  |

通过上面分析，我们能够对NFSD 有一个整体的认识。NFSD 通过线程函数 nfsd) 调用RPC 的接口接收和处理消息，而RPC 服务则根据解析的消息将消息分 发给注册的函数指针。虽然大家对NFSD 有了一个整体的认识，但如果不阅读代码 则可能还是比较模糊。为了让大家更加清晰地了解处理流程，下面以写数据为例分 析一下整个处理流程。

在nfsd() 函数中，分别调用RPC 服务的svc\_recv ( 函 数 和svc\_process() 函数来 接收和处理数据。接收数据部分逻辑比较简单，请大家自行阅读代码，这里重点介 绍一下处理数据的实现。

svc\_process() 接口函数最终调用svc\_proces\_common() 函数来完成实际的处理 流程，如代码5-15所示。这里展示的代码并非该函数的全部代码，而是删减了部 分非关键代码后的主要处理逻辑。

**代码5** **-** **15** **svc\_** **process\_** **common()函数的实现**

**net/nfsd/nfssve.c**

|  |  |  |
| --- | --- | --- |
| 1271  1272  1273  1310  1311  1312  1313  1314  1315  1316  1352  1353  1354 | static int  svc\_process\_common(struct svc\_rqst \*rqstp,struct kvec \*argv,struct kvec \*resv) | |
| {  //删除部分代码  rqstp->rq\_prog=prog=svc\_getnl(argv); rqstp->rq\_vers=svc\_getnl(argv);  rqstp->rq\_proc=sve\_getnl(argv); | //程序号 //版本号 //例程号 |
| /根据程序ID(prog)我到具体的程序指针，对于NFS来说就是nfsd\_program for(progp=serv->SV\_program;progp;progp=progp->pg\_next)  if (prog=progp->pg prog) break;  //删除认证相关的代码  /完成请求的初始化，对于NFS来说是nfsd\_init\_requestO函数，该函数根据接收到的数据包的数 据完成对rqstp的初始化，包括例程(如read)、write()等例程函数)和参数等  rpc\_stat=progp->pg\_init\_request(rqstp,progp,&process); switch(rpc\_stat)  case rpc success: | |

**文件系统技术内幕**

大数据时代海量数据存储之道

break;

case rpc prog\_unavail:

1356

1357

1358

1359

1360

1361

goto err\_bad\_prog; case rpc\_prog\_mismatch:

goto err\_bad\_vers; case rpc\_proc\_unavail:

goto err\_bad\_proc;

1362

1363

procp=rqstp->rq\_procinfo;

//Should this check go into the dispatcher? if(!procpl[lprocp->pc\_func)

1365

1366

goto err\_bad\_proc;

1368

//完成语法检查

serv->sv\_stats->rpccnt+t;

1370

1371

trace\_svc\_process(rqstp,progp->pg\_name);

1372

//构建回复头

1373

1374

1375

statp =resv->iov\_base +resv->iov\_len; svc\_putnl(resv,RPC\_SUCCESS);

1376

1377

1378

1379

if(procp->pc\_xdrressize)

1380

svc\_reserve\_auth(rqstp,procp->pc\_xdrressize<<2);

1382

//调用处理请求的函数

1383

1384

1385

1386

1387

1388

1389

1390

1391

1392

1393

1394

1395

if(process.dispatch)( // 请求处理

if(svc\_generic\_dispatch(rqstp,statp)

goto release\_dropit;

if(\*statp=pc\_garbage\_args)

goto err garbage;

auth\_stat=svc\_get\_autherr(rqstp,statp); if(auth\_stat !=rpc\_auth\_ok)

goto err\_release\_bad\_auth;

}else{

dprintk("svc:calling dispatcherln"); if(lprocess.dispatch(rqstp,statp))

goto release\_dropit;//释 放 应 答 信 息

1396

//删除部分代码

1485

请求初始化的流程并不复杂，主要是根据请求数据包中的数据完成对请求结构 体的初始化，核心是完成函数指针的初始化(第1352行)。完成初始化后就可以调 用 svc\_generic\_dispatch (函数进行请求处理(第1385行),该函数包含两部分功能：

**第** **5** **章** 基于网络共享的网络文件系统



一部分是对数据包进行译码，主要解析NFS 需要的参数(第1188行～第1192行); 另一部分是调用对应的函数指针进行进一步处理(第1194行)。svc\_generic\_dispatch() 函数的实现如代码5-16所示。

**代码5-16** **svc\_generic\_dispatch()函数的实现**

**net/sunrpe/sve.c**

static int

1178 svc\_generic\_dispatch(struct svc\_rqst\*rqstp,\_be32\*statp)

1179 

struct kvec \*argv=&rqstp->rq\_arg.head[0];

1181 struct kvec \*resv=&rqstp->rq\_res.head[0];

const struct sve\_procedure \*procp=rqstp->rq procinfo;

1183

1187

if (procp->pc\_decode &&

procp->pc\_decode(rqstp,argv->iov\_base)){/ 对二进制数据进行译码

1190 \*statp=rpc\_garbage\_args;



/调用注册的函数指针进行处理，如nfsd3\_proc\_write 等

1194 \*statp=procp->pe\_func(rqstp);

1195

1196 if(\*statp=rpc\_drop\_replyll

1197 test\_bit(RQ\_DROPME,&rqstp->rq\_flags))

if(test\_bit(RQ\_AUTHERR,&rqstp->rq\_flags))

return

if(\*statp

return

1205

1;

I=rpc\_success) 1;

if(procp->pc\_encode &&

1208 lprocp->pe\_encode(rqstp,resv->iov\_base +resv->iov\_len))

1209 dprintk("svc:failed to encode replyin"); 1210 //serv->sv\_stats->rpcsystemerr++;

1211 \*statp=rpc\_system\_err;

1212 

1213 return 1;

1214 

在 svc\_generic\_dispatch (函数中调用的函数指针正是服务启动时注册的，并在 请求解析时在代码5-15中完成了初始化。返回svc\_generic\_dispatch (函数中就可以

**文件系统技术内幕**

大数据时代海量数据存储之道

使用调用的函数指针进行相关处理。

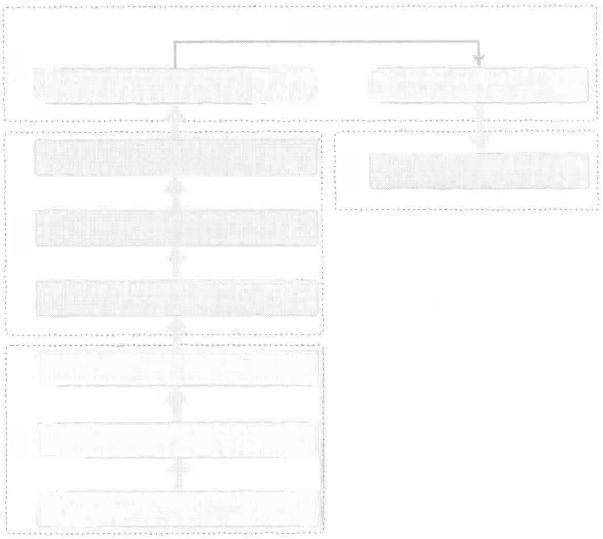
当调用到具体的函数指针(如nfsd3\_proc\_write) 时就进入了具体的流程，这部 分逻辑相对比较简单。以读/写为例，相关函数通常调用VFS 的文件访问接口来完 成具体的操作。

上述逻辑实际上已经比较清晰了，但是由于很多地方使用了函数指针，所以给 大家的感觉不够直观。为了让大家能够更加清晰地理解NFS 服务端处理请求的流 程，下面以写数据为例给出整个函数调用流程，如图5-19所示。

VFS NFSD

Ext4

RPC



略部分函数调用细节

vfs\_iter\_writeo

nfsd\_vfs\_write()

nfsd\_wrte()

nfsd3\_proc\_write()

SVC\_generic\_dispatch0

Svc\_process\_common(

sVC\_process(

ext4\_file\_write\_iter()

vfs\_iter\_write(

图5-19 NFSD写数据调用流程

通过图5-19可以看清楚写数据的处理流程及各个模块的函数。需要注意的是， 图5-19只罗列了关键的函数，并非所有函数。

对于协议其他函数的处理，在RPC 模块的流程是一样的，差异在NFSD 模块 中，主要是NFSD 模块中被调用的函数指针有所不同。

**5.7.2 NFS-Ganesha**

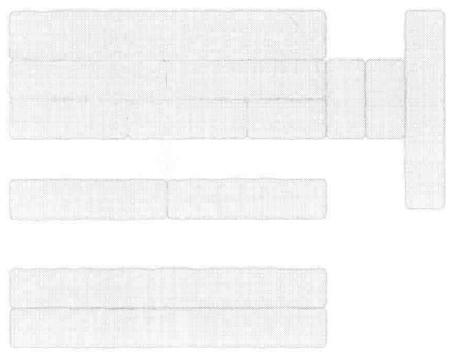
NFS-Ganesha 是一个用户态的NFS 服务端，提供了与操作系统内核NFSD 相 同的功能。NFS-Ganesha 具有在用户态实现、多协议支持和后端多存储类型支持3 个特点。由于在用户态实现这个特点，因此很多其他特性的实现也比较方便。同时

第5章基于网络共享的网络文件系统

由于在用户态实现，读/写会出现内核态与用户态拷贝的情况，因此性能相对NFSD 要差一些。

NFS-Ganesha的整体架构(见图5-20)与内核NFSD的整体架构类似，也是通 过守护进程和函数指针的方式实现请求的分发处理。NFS-Ganesha 没有实现自己的 RPC 库，而是使用了TI-RPC。

GSSAPI



RPC分发

RPCSEC\_GSS Dup Req模 块

挂载协议 NFSv2/NFSv4 NFSv4

**元数据缓存** 数据缓存

文件系统抽象层

命名空间API

哈希表

安全模块

图5-20 NFS-Ganesha的整体架构

由于基于第三方RPC库，因此NFS-Ganesha只需要将关键的函数指针注册到 该RPC 库即可。这样，当客户端有请求时，RPC 库会自动找到对应的函数指针进 行处理，这部分逻辑将会比较简单。

我们重点看一下NFS-Ganesha 初始化的过程，如代码5-17所示。通过阅读代 码可以知道，nfs\_Init\_svc (函数的主要功能是RPC 模块的初始化(第1262行)和 函数指针的注册(第1324行以下)。其中，函数指针的注册又包含XPRT 的创建 (第1324行)和具体函数注册两部分逻辑。

**代码5-17** **nfs\_Init\_svc()函数**

**sre/MainNFSD/nfs\_rpc\_dispacher\_thread.c**

|  |  |
| --- | --- |
| 1214  1239  1240  1241  1242  1243  1244  1245 | void nfs\_Init\_svc(void) {  //删除部分代码  sve\_params.disconnect\_cb=NULL;  sve\_params.alloc\_cb=alloc\_nfs\_request; sVc\_params.free\_cb=free\_nfs\_request;  sve\_params.flags=SVC\_INIT\_EPOLL; //使用EPOLLmgmt.事件管理机制 sve\_params.flagsF=SVC\_INIT\_NOREG\_XPRTS;/不调用xprt\_register  sVc params.max\_connections=nfs param.core param.rpc.max\_connections; |

**文件系统技术内幕**

大数据时代海量数据存储之道

sve\_params.max\_events=1024; /事件队列的长度

1246

1247

SVc\_params.ioq\_send\_max=

nfs\_param.core\_param.rpc.max\_send\_buffer\_size; sve\_params.channels=N\_EVENT\_CHAN;

1249

1250

1251

1252

svc\_params.idle\_timeout=nfs param.core param.rpc.idle\_timeout\_s; svc\_params.ioq\_thrd\_min=nfs\_param.core\_param.rpc.ioq\_thrd\_min;

svc\_params.ioq\_thrd\_max=nfs\_param.core\_param.rpc.ioq\_thrd\_max;

1253

sve\_params.gss\_ctx\_hash\_partitions=

1254

1255

1256

nfs\_param.core\_param.rpc.gss.ctx\_hash\_partitions; svc\_params.gss\_max\_ctx=

nfs\_param.core\_param.rpc.gss.max\_ctx; sve\_params.gss\_max\_gc=

1258

nfs param.core\_param.rpc.gss.max\_gc;

1260

//调用TI-RPC 的初始化函数，完成RPC 模块的初始化 if(lsve\_init(&sve\_params))

1261

LogFatal(COMPONENT\_INIT,"SVCinitialization failed");

1264

for(ix=0;ix<EVCHAN\_SIZE;++ix){ rpc\_evchan[ix].chan\_id=0;

1266

code=sve\_rqst\_new\_evchan(&rpc\_evchan[ix].chan\_id,

1268

sVC\_RQST\_FLAG\_NONE);

1269

1270

1271

1272

if(code)

LogFatal(COMPONENT\_DISPATCH,

"Cannot createTI-RPCevent channel(%d,%d)",

ix,code);

}

1275

//删除部分代码

//为RPC 分 配UDP 和 TCP 套接字 Allocate\_sockets();

1313

1314

1315

if(NFS\_options &CORE\_OPTION\_ALL\_NFS\_VERS)!=0){

1316

1317

Bind\_sockets();

1318

1319

1320

1321

unregister\_rpc();

1322

1323

1324

Create\_SVCXPRTsO;// 创 建XPRT, 完 成XPRT 的初始化

1

//删除部分代码 #ifdef RPCBIND

第5章基于网络共享的网络文件系统

1356 \*在NFS\_V3和NFS\_V4上为UDP 和TCP执行

1357 \* 所 有RPC 注册。需要注意的是，V4 服务器不需要

1358 \* 在 rpcbind 上注册，因此，如果注册失败，则不会让启动失败

1359

#ifdef\_USE\_NFS3

1361 if(NFS\_options&CORE\_OPTION\_NFSV3){

1362 Register program(P\_NFS,NFS\_V3);//注册函数指针

1363 Register program(P\_MNT,MOUNT\_V1);

1364 Register\_program(P\_MNT,MOUNT\_V3); 1365 #ifdef\_USE\_NLM

1366 if(nfs param.core\_param.enable\_NLM)

1367 Register\_program(P\_NLM,NLM4\_VERS);

#ifdef USE\_NFSACL3

1370 if(nfs\_param.core\_param.enable\_NFSACL)

1371 Register program(P\_NFSACL,NFSACL\_V3); 1372

1373 

1374 #endif

1375

1376 I/NFS\_V4 版本的注册是可选的

if(NFS\_options &CORE\_OPTION\_NFSV4)

Register program(P\_NFS,NFS\_V4);

1379

1380 #ifdef\_USE\_RQUOTA

1381 if(nfs\_param,core\_param.enable\_RQUOTA&&

|  |  |  |
| --- | --- | --- |
| 1382 |  | (NFS\_options&CORE\_OPTION\_ALL\_NFS\_VERS))( |
| 1383 |  | Register\_program(P\_RQUOTA,RQUOTAVERS); |
| 1384 |  | Register\_program(P\_RQUOTA,EXT\_RQUOTAVERS); |
| 1385 |  |  |
| 1386 | #endif |  |
| 1387 | #endif |  |



使 用Create\_SVCXPRTs(O函数创建XPRT, 其内部完成了对不同XPRT 的创建 和初始化。以TCP 协议为例，在Create\_tcp(O函数中主要完成对全局变量tcp\_xprt 的初始化。在全局变量初始化中最重要的是将不同协议分发函数(如 nfs\_rpc\_valid\_NFSO) 赋 值 给XPRT, 如代码5-18所示。

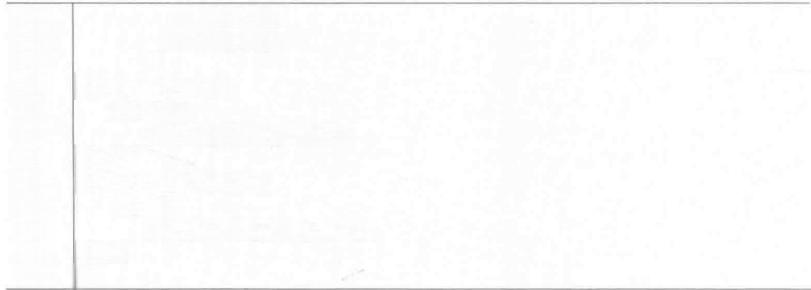
**代码5-** **18** **Create\_** **SVC** **XP** **RTs()函数的实现**

**sre/MainNFSD/nfs\_rpe\_dispatcher\_thread.c**

|  |  |
| --- | --- |
| 573 | void Create\_SVCXPRTs(void) |
| 574 | { |
| 575 | protos p: |
| 576 |  |
| 577 | LogFullDebug(COMPONENT\_DISPATCH,"Allocation of the SVCXPRT"); |
| 578 | for(p=P\_NFS;p<P\_COUNT;p++) |
| 579 | if(nfs protocol\_enabled(p)){ |

**文件系统技术内幕**

大数据时代海量数据存储之道



580 Create\_udp(p);

581 Create\_tcp(p); 582 

583 #ifdef RPC\_VSOCK

584 if(vsock)

585 Create\_tcp(P\_NFS\_VSOCK); 586 #endif

587 #ifdef\_USE\_NFS\_RDMA

588 if(rdma)

589 Create\_RDMA(P\_NFS\_RDMA);

590 #endif

591 )

函数指针的注册是通过Register\_program() 函数完成的，该函数最终调用了TI- RPC 中 的svc\_reg() 函数来将XPRT ( 比 如tcp\_xprt) 注 册 到RPC 服务中。这样，当 有 请 求 时 ，RPC 服务就可以根据解析的请求来调用前面注册的函数指针(如 nfs\_rpc\_valid\_NFS)。以 Register\_program()函数为入口，后续根据消息中的参数来 调用具体的处理函数来完成相关的处理，以创建文件为例，调用的函数为 nfs3\_create()。

NFS-Ganesha还实现对各种后端存储的支持，包括Ceph 、GlusterFS和GPFS 等文件系统。而对多种文件系统的支持是通过文件系统抽象层(FSAL) 的模块来 实现的。

下面重点介绍一下NFS-Ganesha 的 FSAL 模块，该模块类似于Linux 内核中的 虚拟文件系统(VFS), 它为协议处理函数提供了一个抽象层接口。所有协议的处理 都调用该层面的接口，然后FSAL 的接口根据配置信息来调用具体后端的接口。这 里具体后端是为Ceph 和 GlusterFS 等实现的操作接口，可以将其与VFS 中的具体 文件系统相对应。图5-21所示为Ganesha 源代码中FSAL 与不同后端的目录结构。

DUS 田 doc

日 FSAL

FSAL\_CEPH

FSAL\_GLUSTER

用 词FSAL\_GPFS

FSAL\_KVSFS

FSALLZARDFS FSAL\_MEM

FSAL\_PROXY\_V3 田 FSAL\_PROXY\_V4

FSAL\_PSEUDO

FSAL\_RGW 田 FSAL\_VFS

田 Stackable\_FSALS FSAL\_UP

图5-21 Ganesha源代码中FSAL与不同后端的目录结构

**第5章** 基于网络共享的网络文件系统



由于NFS-Ganesha 中的接口繁多，而逻辑又极其相似，因此这里就不再详细介 绍每个接口的逻辑细节。下面以为Ceph 为例介绍协议处理函数、FSAL 函数、后 端函数和Ceph 文件系统API 函数之间的对应关系。

表5-6列出了最主要的几个函数，大家可以将该表对照源代码阅读，这样可以 非常快速地厘清NFS-Ganesha 各个模块的关系。由于其他函数大同小异，所以此处 并没有罗列全部函数。

**表5-6** **NFS-Ganesha各层API对应举例**

|  |  |  |  |
| --- | --- | --- | --- |
| **N** **FSv3协议** | FSAL | FSAL\_CEPH | Ceph API |
| nfs3\_create() | fsal\_open20 | ceph\_fsal\_open20 | ceph\_ll\_openO |
| nfs3\_mkdir() | fsal\_create() | ceph\_fsal\_mkdir( | ceph\_ll\_mkdir( |
| nfs3\_lookupO | fsal\_lookupO | ceph\_fsal\_lookup() | ceph\_ Ⅱ\_lookupO |
| nfs3\_read( |  | ceph\_fsal\_read20 | ceph\_ll\_read() |
| nfs3\_write( |  | ceph\_fsal\_write2O | ceph\_ Ⅱ\_write() |
| nfs3\_remove() | fsal\_remove() | ceph\_fsal\_unlink() | ceph\_l un link(文件) ceph\_ll\_rmdir(目录) |

以写数据的接口为例，回调函数nfs\_rpc\_valid\_NFSO 会调用nfs3\_write(), 而 该 函数则会通过fsal\_obj\_handle 对象中注册的函数进行进一步的处理。对于Ceph 来 说，注册的函数为ceph\_fsal\_write2(, 该函数会调用Ceph API库中的ceph ll write( 函数完成写数据的操作。

fsal\_obj\_handle 对象的实例则是各个后端通过自己的create\_handle() 函数来创 建的。以Ceph 为 例 ，create\_handle() 函数是在FSAL\_CEPH/export.c 中实现的，该 函数通过调用construct\_handle 来完成fsal\_obj\_handle 的分配和初始化。

关于其他处理流程与写数据流程并没有太大的差异，大家可以参考写数据的流 程进行理解。至此，我们也完成了本章所有内容的介绍。

章

**提供横向扩展的分布式文件系统**

前文已经对网络文件系统进行了深入的介绍，并且对一些开源的网络文件系统 的代码实现进行了解析。同时，我们发现常规的网络文件系统最大的缺点是服务端 无法实现横向扩展。这个缺点对大型互联网应用来说几乎是不可容忍的。

本章将介绍一下在互联网领域应用非常广泛的分布式文件系统。分布式文件系 统最大的特点是服务端通过计算机集群实现，可以实现横向扩展，存储端的存储容 量和性能可以通过横向扩展的方式实现近似线性的提升。

**6.1** **什么是分布式文件系统**

分布式文件系统(Distributed File System,简称DFS) 是网络文件系统的延伸， 其关键点在于存储端可以灵活地横向扩展。也就是可以通过增加设备(主要是服务 器)数量的方法来扩充存储系统的容量和性能。同时，分布式文件系统还要对客户 端提供统一的视图。也就是说，虽然分布式文件系统服务由多个节点构成，但客户 端并不感知。在客户端来看就好像只有一个节点提供服务，而且是一个统一的分布 式文件系统。

在分布式文件系统中，最出名的就是谷歌的GFS。 除此之外，还有很多开源的 分布式文件系统，比较有名且应用比较广泛的分布式文件系统有HDFS、GlusterFS、 CephFS 、MooseFS 和 FastDFS 等 。

分布式文件系统的具体实现有很多方法，不同的文件系统通常用来解决不同的问 题，在架构上也有差异。虽然分布式文件系统有很多差异，但是有很多共性的技术点。 本章将介绍一下分布式文件系统通用的关键技术点，并且结合实例介绍一下实现细节。

第6章提供横向扩展的分布式文件系统

**6.2** **分布式文件系统与网络文件系统的异同**

在有些情况下，NFS 等网络文件系统也被称为分布式文件系统。但是在本书 中，分布式文件系统是指服务端可以横向扩展的文件系统。也就是说，分布式文件 系统最大的特点是可以通过增加节点的方式增加文件系统的容量，提升性能。

当然，分布式文件系统与网络文件系统也有很多相同的地方。比如，分布式文 件系统也分为客户端的文件系统和服务端的服务程序。同时，由于客户端与服务端 分离，分布式文件系统也要实现网络文件系统中类似RPC 的协议。

另外，分布式文件系统由于其数据被存储在多个节点上，因此还有其他特点。 包括但不限于以下几点。

(1)支持按照既定策略在多个节点上放置数据。

(2)可以保证在出现硬件故障时，仍然可以访问数据。

(3)可以保证在出现硬件故障时，不丢失数据。

(4)可以在硬件故障恢复时，保证数据的同步。

(5)可以保证多个节点访问的数据一致性。

由于分布式文件系统需要客户端与多个服务端交互，并且需要实现服务端的容 错，通常来说，分布式文件系统都会实现私有协议，而不是使用NFS 等通用协议。

**6.3** **常见分布式文件系统**

分布式文件系统的具体实现方法有很多，其实早在互联网兴盛之前就有一些分 布式文件系统，如Lustre 等。早期分布式文件系统更多应用在超算领域。

随着互联网技术的发展，特别是谷歌的GFS 论文的发表，分布式文件系统又 得到进一步的发展。目前，很多分布式文件系统是参考谷歌发布的关于GFS 的 论 文实现的。比如，大数据领域中的HDFS 及一些开源的分布式文件系统FastDFS 和 CephFS 等 。

在开源分布式文件系统方面，比较知名的项目有大数据领域的HDFS 和通用的 CephFS 和 GlusterFS等。这几个开源项目在实际生产中使用得相对比较多一些。接 下来将对常见的分布式文件系统进行简要的介绍。

文件系统技术内幕

大数据时代海量数据存储之道

**6.3.1 GFS**

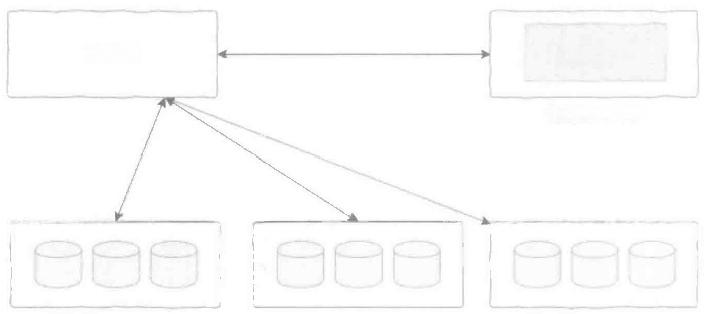
GFS 是谷歌的一个分布式文件系统，该分布式文件系统因论文The Google File System12] 广为世人所知。GFS 并没有实现标准的文件接口，也就是其实现的接口并 不 与POSIX 兼容。但包含创建、删除、打开、关闭和读/写等基本接口。

GFS 集群节点包括两个基本角色： 一个是master, 该角色的节点负责文件系统 级元数据管理；另一个是chunkserver, 该角色的节点通常有很多个，用于存储实际 的数据。GFS 对于文件的管理是在master 完成的，而数据的实际读/写则可以直接 与chunkserver 交互，避免master 成为性能瓶颈。

GFS 在实现时做了很多假设，如硬件为普通商用服务器、文件大小在数百兆甚 至更大及负载以顺序大块读者为主等。其中，对于文件大小的假设尤为重要。基于 该假设，GFS 默认将文件切割为64MB 大小的逻辑块(chunk), 每 个chunk 生成 一 个64位的句柄，由master 进行管理。

这里需要重点强调的是，每个chunk 生命周期和定位是由master 管理的，但是 chunk 的数据则是存储在chunkserver 的。正是这种架构，当客户端获得chunk 的 位 置和访问权限后可以直接与chunkserver 交互，而不需要master 参与，进而避免了 master 成为瓶颈。

图6-1所示为GFS 架构示意图。



客户端 **管理数据**

服务器-master

服务器-chunkserver 服务器-chunkserver 服务器-chunkserver

图6-1 GFS 架构示意图

除 了GFS, 还有很多类似架构的分布式文件系统。比如，在大数据领域中的 HDFS, 它是专用于Hadoop 大数据存储的分布式文件系统。其架构与GFS 的架构 类似，包含一个用于管理元数据的节点和多个存储数据的节点，分别为 namenode

第 6 章 提 供 横 向 扩 展 的 分 布 式 文 件 系 统

和datanode。

HDFS 主要用来进行大文件的处理，它将文件按照固定大小切割，然后存储到 数据节点。同时为了保证数据的可靠性，这些数据被放到多个不同的数据节点。文 件被切割的大小和同时放置数据节点的数量(副本数)是可配置的。

虽 然HDFS 是针对大文件设计的，但是也可以处理小文件。只不过对于小于切 割单元的文件不进行切割。另外，HDFS 对小文件也做了一些优化，如HAR 和 SequenceFile 等方案，但HDFS 终究不是特意为小文件设计的，因此在性能方面还 有些欠缺。

除此之外，还有很多模仿GFS 的开源分布式文件系统，如FastDFS 、MooseFS 和 BFS 等。但大多数开源项目只实现了文件系统最基本的语义，严格来说不能称 为分布式文件系统，更像是对象存储。

**6.3.2 CephFS**

有必要单独介绍一下CephFS 的 原 因 是CephFS 不仅实现了文件系统的所有语 义，而且实现了元数据服务的多活横向扩展13]。

CephFS 的 架 构 与GFS 的架构没有太大差别，其突出的特点是在架构方面将 GFS 的 单 活master 节点扩展为多活节点。不仅可以元数据多活，而且可以根据元 数据节点的负载情况实现负载的动态均衡。这样，CephFS 不仅可以通过增加节点 来实现元数据的横向扩展，还可以调整节点负载，最大限度地使用各个节点的CPU 资源。

同 时 ，CephFS 实现了对POSIX 语言的兼容，在客户端完成了内核态和用户态 两个文件系统实现。当用户挂载CephFS 后，使用该文件系统可以与使用本地文件 系统一样方便。

**6.3.3 GlusterFS**

GlusterFS 是一个非常有历史的分布式文件系统，其最大的特点是没有中心节 点。也就是GlusterFS 并没有一个专门的元数据节点来管理整个文件系统的元数据。

GlusterFS 抽象出卷(Volume) 的概念，需要注意的是，这里的卷与Linux LVM 中的卷并非同一个概念。这里的卷是对文件系统的一个抽象，表示一个文件系统实 例。当我们在集群端创建一个卷时，其实是创建了一个文件系统实例。

GlusterFS有多种不同类型的卷，如副本卷、条带卷和分布式卷等。正是通过这 些卷特性的组合，GlusterFS 实现了数据可靠性和横向扩展的能力。

文件系统技术内幕

大数据时代海量数据存储之道

**6.4** **分布式文件系统的横向扩展架构**

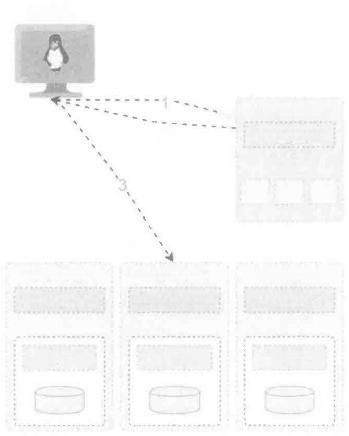
前文介绍了分布式文件系统概念、架构和实例等内容。从前文介绍中我们发现， 对于存储集群端主要有两种类型的架构模式： 一种是以GFS 为代表的有中心控制 节点的分布式架构(以下简称为“中心架构”),另一种是对等的分布式架构(以下 简称为"对等架构"),也就是没有中心控制节点的架构。本节将介绍一下这两种架 构的模式。

**6.4.1** **中心架构**

中心架构是指在存储集群中有一个或多个中心节点，中心节点维护整个文件系 统的元数据，为客户端提供统一的命名空间。在实际生产环境中，中心节点通常是 多于一个的，其主要目的是保证系统的可用性和可靠性。

在中心架构中，集群节点的角色分为两种： 一种是前文所述的中心节点，又被 称为控制节点或元数据节点，这种类型的节点只存储文件系统的元数据信息；另一 种是数据节点，这种类型的节点用于存储文件系统的用户数据。

图6-2所示为中心架构示意图，在该示意图中包含1个控制节点和3个数据节 点。当客户端创建一个文件时，首先会访问控制节点，控制节点会进行元数据的相 关处理，然后给客户端应答。客户端得到应答后与数据节点交互，在数据节点完成 数据访问。



**客户端**



**服务端**

数据节点03

服务端

data

数据节点02 服务端

data

数据节点01

服务端

data

2

图6-2 中心架构示意图

第6章提供横向扩展的分布式文件系统

在分布式文件系统中，控制节点除了存储元数据还有很多其他功能，如对于访 问权限的检查、文件锁的特性和文件的扩展属性。

图6-2只是简单地描述了一下中心架构的访问流程，实际流程要复杂一些，还 要有一些细节需要处理。在具体的实现方面，不同的文件系统会有差异，这个我们 在后面分析具体文件系统时再深入了解。

为了保证文件系统的可用性和可靠性，控制节点通常不止一个。比如，GFS 的 控制节点是主备的方式，其中一个作为主节点对外提供服务。在主节点出现故障的 情况下，业务会切换到备节点[12]。由于中心架构的控制节点同一时间只有一个节点 对外提供服务，因此也称为单活控制节点。

单活控制节点有一个明显的缺点就是元数据管理将成为性能瓶颈。因此有些分 布式文件系统实现了控制节点的横向扩展，也就是多活控制节点。比较有名的就是 CephFS14,CephFS 包含多个活动的控制节点(MDS), 通过动态子树的方式实现 元数据的分布式管理和负载均衡。

**6.4.2** **对等架构**

对等架构是没有中心节点的架构，集群中并没有一个特定的节点负责文件系统 元数据的管理。在集群中所有节点既是元数据节点，也是数据节点。在实际实现中， 其实并不进行角色的划分，只是作为一个普通的存储节点。

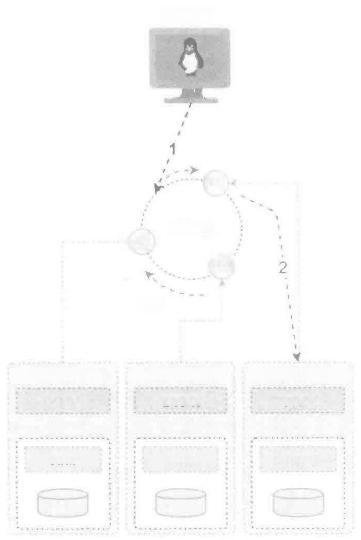
由于在对等架构中没有中心节点，因此主要需要解决两个问题： 一个是在客户 端需要一种位置计算算法来计算数据应该存储的位置；另一个是需要将元数据存储 在各个存储节点，在某些情况下需要客户端来汇总。

这里的客户端可以是代理层。位置计算算法的基本逻辑是根据请求的特征来计 算数据具体应该放到哪个节点。请求特征可以是文件名称或数据偏移等具备唯一性 的字符串，不同文件系统的实现略有所不同。

在类似算法中常用的一种算法为一致性哈希算法(Consistent Hashing)。一致性 哈希算法建立了文件特征值与存储节点的映射关系。当客户端访问文件时，根据特 征值可以计算出一个数值，然后根据这个数值可以从哈希环上找到对应的设备，如 图6-3所示。

**文件系统技术内幕**

**大数据时代海量数据存储之道**



客户端

03

哈希环

01

02

方向

数据节点01 数据节点02

服务端 服务端

data data

数据节点03

服务端

data

图6-3 基于计算位置的数据布局架构示意图

在图6-3中，当客户端访问某一个数据时，首先根据请求特征值的位置按照顺 时针规则确定为数据节点03。然后，客户端会直接与数据节点03交互，实现数据 访问。

**6.5** **分布式文件系统的关键技术**

分布式文件系统本身也是文件系统，因此它与本地文件系统和网络文件系统等 具备一些公共技术。除此之外，鉴于其分布式的特点，还涉及一些分布式的技术。 本节将介绍一下分布式文件系统相关的关键技术。

**6.5.1** **分布式数据布局**

在介绍本地文件系统时我们已经介绍过文件系统数据布局的相关内容，主要介 绍了数据在磁盘上是如何进行存储和管理的。分布式文件系统的数据布局关注的不 是数据在磁盘的布局，而是数据在存储集群各个节点的放置问题。

在分布式文件系统中，数据布局解决的主要问题是性能和负载均衡的问题。其 解决方案就是通过多个节点来均摊客户端的负载，也就是实现存储集群的横向扩

第6章提供横向扩展的分布式文件系统

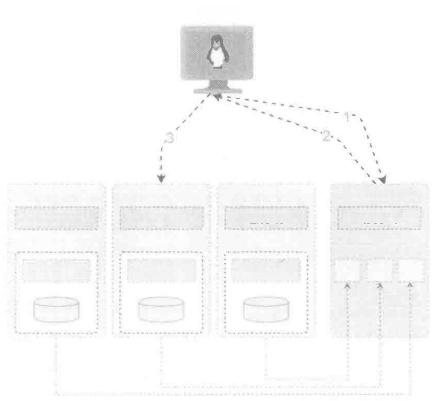
展。因此，在分布式文件系统中，不仅要解决数据量的均衡问题，还要解决负载的 均衡问题。

[**6.5.1.1**](6.5.1.1) **基于动态监测的数据布局**

基于动态监测的数据布局是指通过监测存储集群各个节点的负载、存储容量和 网络带宽等系统信息来决定新数据放置的位置。另外，集群节点之间还要有一些心 跳信息，这样当有数据节点故障的情况下，控制节点可以及时发现，保证在决策时 剔除。

由于需要汇总各个节点的信息进行决策，因此基于动态监测的数据布局通常需 要一个中心节点。中心节点负责汇总各种信息并进行决策，并且会记录数据的位置 信息等元数据信息。使用类似技术的分布式文件系统有BFS 等 。

图6-4所示为基于动态监测的数据布局架构示意图，展示了写入数据的基本流 程。在这种数据布局中，数据节点会定时地将存储容量、节点负载和网络带宽等汇 报给控制节点。当客户端需要写入数据时，客户端首先与控制节点交互(步骤1); 控制节点根据汇总的信息计算出新数据的位置，然后反馈给客户端(步骤2);客户 端根据位置信息，直接与对应的数据节点交互(步骤3)。

**客户端**

**敬据节点02** **数握节点03**

**数据节点01**

**服务端**

**控制节点**

**服务端**

**服务端** **服务端**

data data

data

图6-4 基于动态监测的数据布局架构示意图

[**6.5.1.2**](6.5.1.2) **基于计算位置的数据布局**

基于计算位置的数据布局是一种固定的数据分配方式。在该架构中通过一个算 法来计算文件或数据存储的具体位置。当客户端要访问某个文件时，请求在客户端

**文件系统技术内幕**

大数据时代海量数据存储之道

或经过的某个代理节点计算出数据的具体位置，然后将请求路由到该节点进行处理。

图6-3所示为基于计算位置的数据布局架构示意图。当客户端访问集群数据 时，首先计算出数据的位置(哪个节点),然后与该节点交互。

读到这里，大家可能觉得6.5节中的架构与6.4节中的架构有所重复。其实两 者并没有绝对的关系。前面架构侧重于集群节点的角色，区分点在于集群中是否有 控制节点。这里数据布局架构则侧重于数据的放置方式，与架构无关。

以CephFS 存储为例，其架构是中心架构，元数据由MDS 来管理。但是其数 据布局则是基于计算位置的。在CephFS 中，通过inode ID和数据的逻辑偏移来计 算数据具体的位置。对于GlusterFS 来说，其本身无中心架构，但是数据也是通过 计算位置来放置的。所以，架构与数据布局并没有绝对的关联，这一点需要注意。

**6.5.2 分布式数据可靠性(Reliability)**

分布式数据的可靠性是指在出现组件故障的情况下依然能够能提供正常服务 的能力。对于本节来说，数据的可靠性限定在出现故障的情况下存储系统仍然能够 提供完整数据的能力。

在传统存储系统中，通常采用RAID 技术来保证存储数据的可靠性。RAID 技 术通过一个或多个冗余的磁盘来保证在出现磁盘故障的情况下不会导致数据的丢 失，并仍然对外提供数据访问的能力。

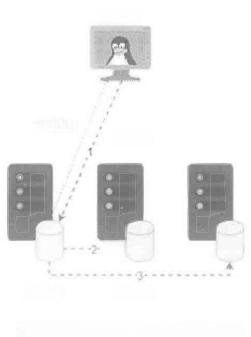
对于分布式文件系统来说，由于数据会分散在多个节点上，因此在出现故障的 情景下会变得更加复杂。很多组件出现故障后都会导致无法访问数据，如磁盘故障、 服务器故障、网卡故障和交换机故障等。因此，在分布式文件系统中，数据的可靠 性是必须要考虑的内容。

[**6.5.2.1**](6.5.2.1) **复制技术** **(Replication)**

复制技术是通过将数据复制到多个节点的方式来实现系统的高可靠。由于同一 份数据会被复制到多个节点，这样同一个数据就存在多个副本，因此也称为多副本 技术，这样当出现节点故障时就不会影响数据的完整性和可访问性。

多副本技术有两种不同的模式：一种是基于主节点的多副本技术；另一种是无 主节点的多副本技术。基于主节点的多副本技术是指在副本节点中有一个节点是主 节点，所有的数据请求先经过主节点，如图6-5所示。对于一个写数据请求，客户 端将请求发送到主节点，主节点将数据复制到从节点，再给客户端应答。

第6章提供横向扩展的分布式文件系统



客户端

读数据

写数据



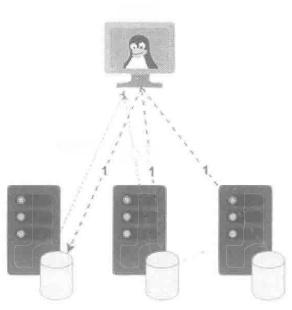
主副本 从副本

写数据 读数据

图6-5主节点模式

对于无主节点的副本模式，在集群端并没有一个主节点，副本逻辑在客户端或 代理层完成。当客户端发送一个写数据请求时，客户端会根据策略自行(或者通过 代理层)找到副本服务器，并将多个副本发送到副本服务器上。

如图6-6所示，当客户端向存储器写数据时，同时将数据写入3个节点中，而 读数据时则从其中两个节点读取。当然，这个读/写策略是可以根据业务需求调整 的，主要视业务对数据一致性和性能等因素的要求而定。



客户端

读数据

写数据

- - - 写 数 据 读数据

图6-6 无中心节点的多副本机制

在无中心多副本技术中通常需要一定的策略来保证数据一致性的，目前主要是 通过Quorum 一致性协议来保证的。该协议通过规定副本数量、写成功副本数量和 读数据副本数量等来保证数据一致性，具体见参考文献中的资料[15]。

多副本技术的基本原理非常简单，难点在于工程实现。在一个大规模分布式系 统中随时会出现网络、磁盘或服务器故障，而这些部件的故障会导致一个副本集中

**文件系统技术内幕**

大数据时代海量数据存储之道

的某些副本是不完整的。除了要考虑数据完整性的问题，还要考虑性能、数据恢复 和数据一致性等问题。然而，性能和数据一致性等问题的解决往往存在一定的矛盾， 很难满足所有的要求。

如果要求强一致性，那么就需要数据同步到副本集的所有节点，而此时延时会 增大，对系统的性能产生影响。为了保证系统的性能，最好能够让写操作完成一个 副本的写数据后就返回，然后在后台实现数据同步。但是，这种处理方式在读数据 时就有可能存在一致性的问题。

[**6.5.2.2**](6.5.2.2) **纠删码技术** **(Erasure Code)**

副本技术存在多个数据副本，因此需要消耗很多额外的存储空间。以3个副本 为例，需要额外消耗2倍的存储空间来保证数据的可靠性。也就是说，有67%的存 储空间是被无效占用的，有效存储空间大概是33%。

副本技术在性能和可靠性方面优势明显，但成本明显比较高。为了降低存储的 成本，很多公司采用纠删码技术来保证数据的可靠性。现在，很多分布式存储都支 持纠删码技术，如Ceph 、HDFS 和 Azure。

纠删码是一种通过校验数据来保证数据可靠性的技术，也就是该技术通过保存 额外的一个或多个校验块来提供数据冗余。与副本技术不同，这种数据冗余技术不 能通过简单复制来恢复数据，而是经过计算来得到丢失的数据。纠删码可以节省空 间的原理是校验块通常只占用户数据的50%左右，存储数据的有效率可以达到 66%,甚至更高。

传统磁盘阵列的RAID 技术可以认为是纠删码的一个特例，如RAID5 可以通 过一个校验块来提供一份冗余，RAID6 可以提供两份冗余。而在分布式存储中通常 使用的是RS(Reed-Solomon) 纠删码算法，这种算法可以提供更大的冗余数据量， 如微软的Azure 可以提供4个冗余数据块，谷歌的GFS 可以提供3个冗余数据块。

下面以RS 纠删码为例介绍一下纠删码的基本原理。限于篇幅，本节不会介绍 太多实现细节，如果想要知道更多关于纠删码的细节则可以见参考文献中的资料1617。

在描述一个使用纠删码的存储系统时通过采用RS(n,m) 的方式。其中，RS 表示 纠删码算法，而n 表示用户数据的块数，m 表示校验数据的块数。如果将这些数据 块分散在n+m 个独立的存储设备上(如磁盘),那么在该系统中最多可以容忍m 个 设备故障。以谷歌的GFS Ⅱ存储为例，其采用的是RS(6,3), 因此可以容忍最多3 个磁盘故障。

RS 纠删码的基本原理是采用矩阵运算，将n 个数据转换为n+m 个数据进行存

第6章提供横向扩展的分布式文件系统

储。其核心是找到一个生成矩阵(Generator Matrix),通过该矩阵与原始数据的运 算可以得到最终要存储的数据，如图6-7所示。

对于编码过程理解是相对简单的，由于生成矩阵上部是对角线为1的单元矩 阵，因此在与原始数据乘法的计算结果得到的依然是原始数据。而下部的m×n 的 子矩阵与原始数据计算得到的结果则为校验数据。

在图6-7的生成矩阵中，要求该生成矩阵的任意n×n 的子矩阵是可逆的。这里 的关键就是如何构建下部m×n 的矩阵，保证生成矩阵任意n×n 子矩阵可逆的特性。 上述m×n 子矩阵并不需要我们自己构造，数学家在这方面已经做了很多工作，常 见的有范德蒙德 (Vandermonde) 和柯西(Cauchy) 两种矩阵。

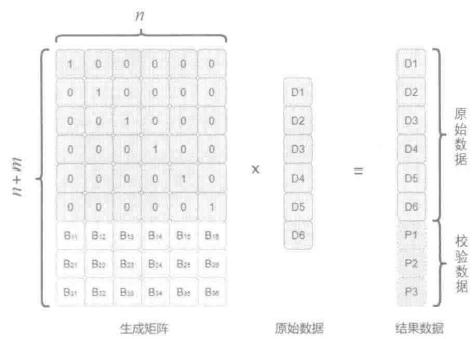


图6-7 RS纠删码运算原理

以范德蒙德矩阵为例[18,本实例中3行的范德蒙德矩阵的格式如图6-8所示。

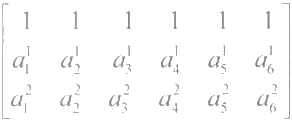


图6-8 范德蒙德矩阵的格式

数据恢复过程就是利用生成矩阵和剩余可用数据来计算原始数据的过程。由于 生成矩阵的任意一个n×n 矩阵都是可逆的，因此当出现任何小于m 个设备故障的 情况下，我们仍然能够从生成矩阵中找到对应的一个n×n 的子矩阵B', 该子矩阵 与原始数据的乘积为结果数据的子集R'。

*B'×D=R'*

两边同时乘以B'¹:

**文件系统技术内幕**

大数据时代海量数据存储之道

*B'- ¹×B'×D=B'¹×R'*

于是可以得到如下等式：

D=B'¹×R'

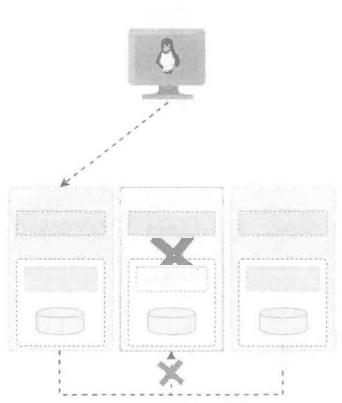
由于逆矩阵B'可以根据生成矩阵计算得到，结果数据子集R'是已知的，因此 我们可以根据两者计算出原始数据集D。

**6.5.3** **分布式数据一致性(Consistency)**

在分布式文件系统中，由于同一个数据块被放置在不同的节点上，我们无法保 证多个节点的数据时时刻刻是相同的，因此会出现一致性的问题。这里的一致性包 括两个方面： 一个方面是各个节点数据的一致性问题；另一个方面是从客户端访问 角度一致性的问题。

在分布式文件系统中，我们经常会遇到各个节点间数据的不一致性。这主要是 因为在由成千上万个组件(包括服务器、交换机和硬盘等)构成的存储系统中，组 件出现故障是非常常见的。

如图6-9所示，由于网络或服务器等故障，服务器02无法被访问。当客户端 更新文件系统中的数据时，就会导致服务器02的数据无法更新，从而导致服务器 02与集群其他节点数据的不一致。在这种情况下，如果服务器02恢复了访问性， 当客户端访问该服务器时就会访问旧的数据。这就出现了数据不一致的情况，可能 会对业务产生影响。



客户端

服务器02

服务端

data=9

个

服务器03

服务端

服务器01

服务端

data=10

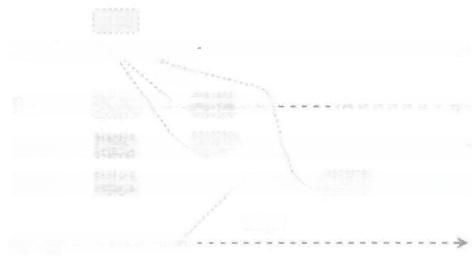
data=10

图6-9 故障导致的数据不一致

第6章提供横向扩展的分布式文件系统



除了故障，还有其他原因会导致各个服务器之间存在数据不一致的情况。比如， 当客户端A 向存储系统写入数据，但其中某个节点(副本3所在节点)由于网络 延迟导致更新延迟。导致副本3所在节点的数据更新比较晚，那么在更新之前3个 节点的数据就存在不一致的情况，如图6-10所示。

x=2 客户端A“

写数据

副本1 x=1 -x=2-

→

-→

副本2 -x=1----x=2--------------→

副本3 -x=1--- ------x=3----→

读数据

客户端B--

图6-10 网络延迟导致的数据不一致

在时间窗口中，如果有一个客户端B 从延迟节点读取数据，那么读到的就是更 新之前的数据。由于存储系统对于客户端来说是个黑盒，这种读取数据与写入数据 不一致的情况会让使用者感到困惑，从客户端角度来说就是客户端访问不一致。

通常来说，我们是无法保证各个节点上数据是完全一致的，只能保证客户端访 问的一致性。为了保证客户端访问数据的一致性，通常需要对存储系统进行特殊的 设计，从而在系统层面保证数据的一致性。这里的一致性最常见的包括强一致性和 最终一致性两种。

强一致性是指当数据的写入操作反馈给客户端后，任何对该数据的读操作都会 读到刚刚写入的数据。

最终一致性是指数据的一个写入操作，如果没有新的写入操作的情况下，该写 入的数据会最终同步到所有副本节点上，但中间会有时间窗口。

6.5.4 设备故障与容错(Fault Tolerance)

在分布式文件系统中必须要解决设备故障的问题。这是因为在大规模分布式文 件系统中设备的总量达到数万个甚至数十万个，设备发生故障就会成为常态。如何 保证经常出现故障的计算机集群能够不间断地对外处理请求，同时又不丢失数据自 然是头等重要的事情。

设备的故障分为两种类型：一种是暂时故障；另一种是永久故障。暂时故障是 指短时间可以恢复的故障，如服务器重启、网线松动或交换机掉电等。永久故障是

文件系统技术内幕

大数据时代海量数据存储之道

指设备下线，且永远不会恢复，如硬盘损坏等。

为了应对系统随时出现的故障，分布式文件系统在设计时必须要考虑容错处 理。主要包括两个方面的内容： 一个方面是在出现故障时系统能够及时发现故障； 另一个方面是发现故障时，系统仍然可以无损地提供服务，并且存储的数据不会 丢失。

为了能够发现设备故障，分布式文件系统应该具备故障检测能力，如检测磁盘、 通信链路或服务的故障等。针对不同的设备，通常有不同的检测方法，如针对服务 器宕机或通信链路故障，通常采用网络心跳的方法。如果在规定时间内没有收到发 送端的心跳包，则可以判定服务器出现了宕机或网络故障。对于磁盘来说，通常通 过读/写访问的方式来检测磁盘故障。

除了上述故障实时检测的方法，还有一种故障预测的方法。故障预测可以预知 设备故障，然后有计划地将该设备下线，避免突然下线导致的性能等问题。以磁盘 为例，有些公司将SMART 技术和深度学习技术结合来预测磁盘的故障[19,在预测 故障的前提下可以提前处理数据，避免因为故障导致出现更严重的问题。

为了保证组件在出现故障的情况下系统仍然能够对外提供无损的服务，分布式 文件系统使用了部件冗余。比如，谷歌的GFS 会将同一份数据放置到不同的数据 节点，在出现磁盘故障甚至节点故障的情况下，仍然能够通过其他节点提供服务。 Ceph 的 CRUSH 算法，不仅可以保证数据的冗余，而且考虑了故障域的因素。它可 以将数据放置在不同的节点、机柜、机房，甚至数据中心。这样，就可以通过不同 的故障域来应对不同级别的故障。

**6.6** **分布式文件系统实例之CephFS**

Ceph 本身实现了对块存储、对象存储和文件系统等多种存储形态的支持。Ceph 对前两者的支持非常成熟，但对文件系统的支持略有欠缺，主要是稳定性欠佳。

Ceph 的作者Sage 本来想实现一个非常高大上的分布式文件系统，但由于想要 支持的特性太多，而功能又过于复杂，因此文件系统一直不够稳定。直到2016年， CephFS 在禁用了很多特性的情况下宣布可以将其应用在生产环境中。

**6.6.1 搭建一个CephFS分布式文件系统**

在创建CephFS 分布式文件系统之前，先部署一个Ceph 集群。关于Ceph 集 群 的安装部署不在本书的讲解范围内，本书不再赘述，见参考文献中的资料[20][21]。

第 6 章 提 供 横 向 扩 展 的 分 布 式 文 件 系 统

基于已有的Ceph 集群，通过两个主要步骤就可以提供文件系统服务，一个是 安装和启动MDS 服务，该服务是文件系统的元数据管理服务；另一个是创建存储 数据的存储池资源。先在gfs1 节点部署MDS 服务，命令如下：

ceph-deploymds create gfs1

对于CephFS, 需要创建两个存储池来存储数据，一个存储池用于存储文件系 统的元数据，另一个存储池用于存储用户数据。创建存储池的步骤如下：



ceph osd pool create fs\_data 256

ceph osdpoolcreate fs\_metadata 256 ceph fs new cephfs fs\_metadata fs\_data

然后就可以使用该文件系统。以内核态文件系统为例，其挂载方法与其他文件 系统很类似 。

mount -t ceph <192.168.1.100>:6789://mnt/cephfs-o name=admin,secret=secretID

其 中 ，secretID 是一个安全密钥，在启用安全认证的情况下需要该选项，如果 部署Ceph 时没有启用安全认证则不需要该选项。以作者部署的测试环境为例， secretID的实际值为AQDNnfBcuLkBERAAeNj60b+tlY/t31NSScIRhg==, 这个值在 不同的环境中通常是不同的 ， 这一点需要注意 。 我们可以通过如下命令得到该信息 ：

ceph auth get client.admin

当执行上述命令后可以得到如下信息，其中，key 的值就是上文中需要的

secretID。

exported keyring for client.admin [client admin]

key=AQC8r/NcAPmgHBAAxjP9/knwdXjBVnE4zXlqmg—

caps mds="allow

caps mgr="allow\*" caps mon="allow\* caps osd="allow\*"

如果一切正常，那么在客户端就可以使用该文件系统了。CephFS 的使用与本 地文件系统并无任何差别，换句话说，用户不会感觉到该文件系统是Ceph 集群提 供 的 。

**6.6.2** **CephFS分布式文件系统架构简析**

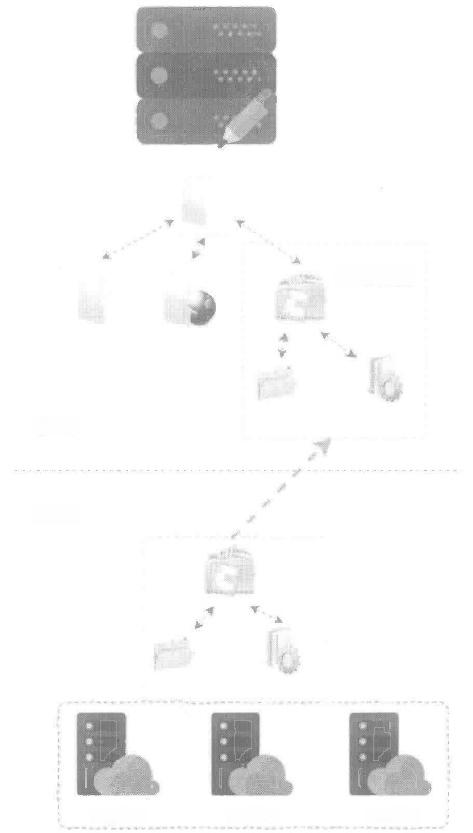
Ceph 提供了块、对象和文件等多种存储形式，实现了统一存储。Ceph 的文件 系统是基于RADOS 集群的，也就是说CephFS 对用户呈现的是文件系统，而在其 内部则是基于对象来存储数据的 。

文件系统技术内幕

大数据时代海量数据存储之道

CephFS 是分布式文件系统，这个分布式从两个方面理解， 一个方面是底层存储数 据依赖的是RADOS 集群；另一个方面是其架构是C/S (客户端/服务端)架构，文件 系统的使用是在客户端，客户端与服务端通过网络通信进行数据交互，类似NFS。

如图6-11所示，客户端通过网络的方式连接到CephFS 集 群 ，CephFS 集群的文件 系统映射到客户端，呈现出一个本地的目录树。从用户的角度来看，这个映射是透明的。



/mnt/data

客户端

集群端

ceph-m ceph-m ceph-m

CephFS 集群

图6-11 客户端访问存储集群原理示意图

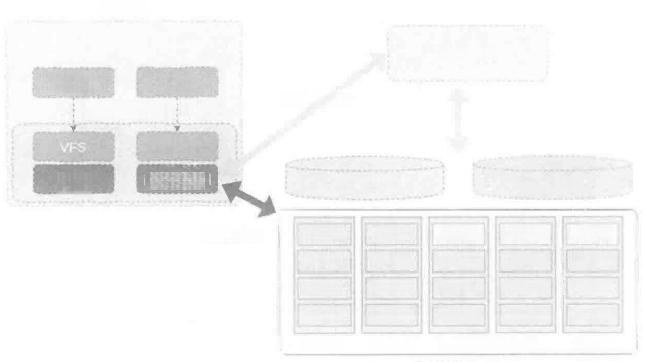
第6章提供横向扩展的分布式文件系统

对于CephFS 集群来说，数据并非以目录树的形式存储。在CephFS 集群中， 数据是以对象的形式存储的，也就是文件系统的所有数据都是以对象的形态平铺在 存储池中的。文件数据的访问最终也会转换为RADOS 对象的访问。

由于CephFS 本身是基于其对象存储RADOS 的，因此RADOS 的组件在CephFS 中都是需要的。RADOS 的核心组件包括两部分： 一部分是Monitor ( 简 称MON) 集群；另一部分是OSD 集群。由于本书主要聚焦文件系统的内容，因此关于RADOS 相关的内容不再赘述，见参考文献中的资料[21][2][23]。

Ceph 文件系统是在RADOS 集群的基础上增加了MDS 组件集群，如图6-12 所示。这里的MDS(Meta Data Server,元数据服务)负责文件系统元数据的管理。 根据CephFS 的组件组成，我们可以知道CephFS 是一个有中心节点的分布式文件

系 统 。



客户端

MDS 集群

lbhue wnte Linux内 核

VFS

数据池

MON

OSD

OSD

OSD

Server

RADOS Custer

|  |  |
| --- | --- |
| MON  OSD  OSD  OSD  Server | MON  OSD  OSD  OSD  Server |

OSD

OSD

OSD

OSD

Server

OSD

OSD

OSD

OSD

Server

1元数据操作

2.数据操作

元数据池

CephFS

Fuse

图6-12 CephFS的主要流程

对于客户端来说，访问Ceph 文件系统的流程大致分为两个子流程： 一个是通 过MDS 访问集群文件系统的元数据；另一个流程是客户端对数据的访问(读/写), 该流程是客户端直接与RADOS 集群交互的。

Ceph 文件系统架构的一个特点是尽量减少对MDS 的访问。我们知道本地文件 系统的元数据中包含文件数据的位置信息，但是在Ceph 中却与众不同。Ceph 文件 系统中文件的元数据并不包含数据的位置信息，而是通过计算的方式获得。也就是 说 ，CephFS 采用的是基于计算位置的数据布局方式。

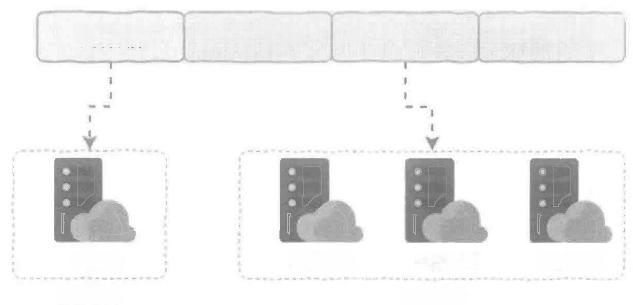
由于客户端对文件数据的访问直接与数据节点(OSD) 交互，因此Ceph 对 于 文件数据的存取并不需要经过元数据节点，而是直接与RADOS 集群交互。当然，

**文件系统技术内幕**

大数据时代海量数据存储之道

由于文件的变化会引起文件大小和时间戳等信息的变化是需要在MDS 更新的，有 些元数据的访问是不可避免的。

以CephFS 的默认配置为例，Ceph 文件系统中的文件会被以4MB 为粒度切割 为大小相等的逻辑块。如图6-13所示，元数据是文件的属性数据，其内容由MDS 集群来处理；用户数据则按照切割后的粒度，以对象的形式存储在OSD 集群中。 我们将切割后的数据称为逻辑块，每个逻辑块以 inode ID和逻辑偏移为核心信息， 以一定的规则命名。因此，客户端可以根据规则直接在客户端生成对象的名称，而 不需要与MDS 过多交互。由于确定了对象名称，而且也确定了文件系统的数据池， 于是客户端就可以直接进行数据读/写/操作，并将逻辑块以对象的形式存储在 RADOS 集群中。



**数据(4MB)**  **数据(4MB)**

ceph-osd ceph-osd ceph-osd

OSD 集 群

ceph-m

MDS 集 群

**元数据**

图6-13 集群对文件数据的存储

读者可能会疑惑，文件的大小是不确定的，对象是如何生成的呢?实际上 CephFS 中的文件对应的对象是按需生成的。也就是只有写数据的区域才有对应的 对象，而空洞部分则是没有对象的。

**6.6.3** **CephFS客户端架构**

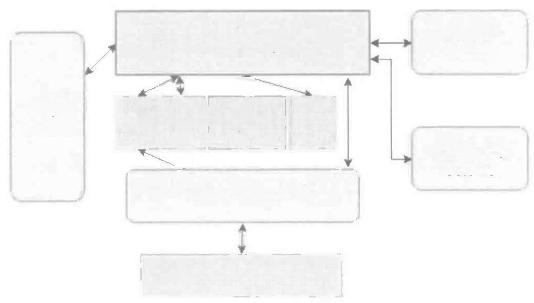
CephFS 的客户端有多种实现方式， 一种是基于Linux 内核中客户端的实现， 还有一种是基于Fuse 框架(更多细节请参考7.1.1)的实现。虽然是两种不同的实 现方式，但是没有本质的区别。

由于基于Fuse 框架的实现封装了很多细节，整体逻辑还是比较简单的，因此 本节不对该实现进行介绍。本节主要介绍一下CephFS 基 于Linux 内核中客户端的 实现。

**第** **6** **章** 提供横向扩展的分布式文件系统



内核客户端是基于VFS 实现的，因此其整体架构与其他Linux 文件系统的整 体架构非常相似。如图6-14所示，CephFS 与 VFS 、Ext4 和 NFS 的关系。可以看 出CephFS 是一个和NFS 与 Ext4 非常类似的文件系统。



**Virtual File System**

空

Ext2 Ext4 CephFS NFS

金

**Buffer Cache**

Device Drivers

Inode Cache

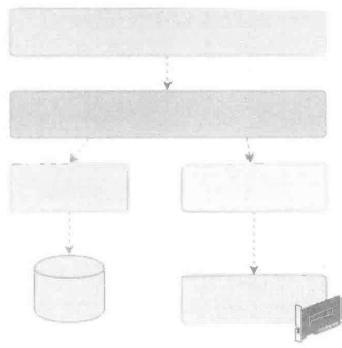
Directory Cache

Page Cache

图6-14 CephFS 与 VFS 、Ext4 和 NFS 的关系

与Ext4 等本地文件系统相比，CephFS 的差异点在于它是通过网络将数据存储 在RADOS 集群的。如图6-15所示，当CephFS 的数据需要持久化时，可以通过网 络模块将数据发送到MDS 或 RADOS 集群进行处理。

RADOS 集群



system call layer

VFS

local fs CephFS

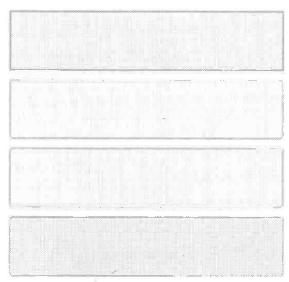
Network

图6-15 CephFS 与本地文件系统的对比

如果按照CephFS 的逻辑架构来划分，CephFS可以分为如图6-16所示的4层。 其中，最上面是接口层，这一层是注册到VFS 的函数指针。用户态的读/写函数最 终会调用该层的对应函数API, 而该层的函数会优先与缓存交换。

文件系统技术内幕

大数据时代海量数据存储之道



**接口层**

**页缓存(Page** **Cache)**

**数据读/写层**

**消息层** **(Net/ceph)**

图6-16 CephFS客户端软件模块

页缓存是所有文件系统公用的，并非CephFS 独享。我们暂且将页缓存归为 CephFS 客户端的一层。以写数据为例，请求可能将数据写入缓存后就返回了。而 缓存数据的刷写并不一定实时同步，而是根据适当的时机通过数据读/写层的接口 将数据发送出去。

然后是数据读/写层，数据读/写层实现的是对请求数据与后端交互的逻辑。对 于传统文件系统来说是对磁盘的读/写，对于CephFS 来说是通过网络对集群的 读/写。

消息层位于最下面，消息层主要完成网络数据收发的功能。该模块在Linux 内 核的网络模块中，不仅 CephFS 使用该模块，块存储RBD 也使用该模块网络收发 的功能。

关于客户端的内容这里介绍的比较少，大家看完后估计有可能还是云里雾里 的。大家先不用着急，我们在后面对代码解析部分会详细地介绍各层的细节。

**6.6.4** **CephFS集群端架构**

通过前文我们了解到，CephFS 的集群端分为MDS 集群和RADOS 集群两部 分。其中，MDS 集群负责管理文件系统的元数据，而RADOS集群负责管理数据。 RADOS是公共部分，本书不做介绍，我们主要聚焦在MDS组件的架构上。

CephFS 的 作 者Sage 有一个想法，就是将MDS 做成一个可以随意横向扩展的

集 群 。Sage 使用动态子树分区(DynamicSubtree Partitioning)14的方式将不同的子

目录根据负载情况分布在不同的MDS 上。从理论上来说，MDS 可以横向扩展到数 百个，因此整个文件系统的承载能力可谓非常强劲。但是由于其太复杂，目前仍未 在实际生产环境中使用，在生产环境中使用更多的还是MDS 主备模式。

第6章提供横向扩展的分布式文件系统

虽然CephFS 想实现的功能非常复杂，但是其软件架构并不复杂，模块之间的 逻辑也比较清晰。图6-17列出了MDS 的主要模块。

**服务模块** 缓存模块 锁模块 **负载均衡模块**

通信与消息分发模块

图6-17 MDS的主要模块

服务模块负责处理文件相关的操作，如创建文件、删除文件、重命名和设置获 取扩展属性等。每一个操作都有一个具体的函数相对应，如NFS 协议中的例程函数。

缓存模块实现对关键元数据的缓存，通过将热点数据缓存到内存中以提升元数 据访问的性能。在CephFS 中被缓存的元数据包括inode 和 dentry 等内容。

锁模块负责分布式锁相关的特性。对分布式文件系统而言，被多个客户端同时 访问是很正常的，因此实现一种锁机制必不可少。

负载均衡模块是多活MDS 的实现，负责在MDS 多活场景实现元数据的负载 均衡。

通信与消息分发模块负责消息的收发。并且该模块在收到客户端的消息后会转 发给MDS 的不同模块进行处理，如服务模块和锁模块等。以文件相关操作为例， 从消息模块接收消息后，其分发到服务模块的顺序如图6-18所示。

AsyncMessenger MDSDaemon MDSRankDispatcher Server

Text

ms\_dispatch2

ms\_dispatch

ms\_dispatch

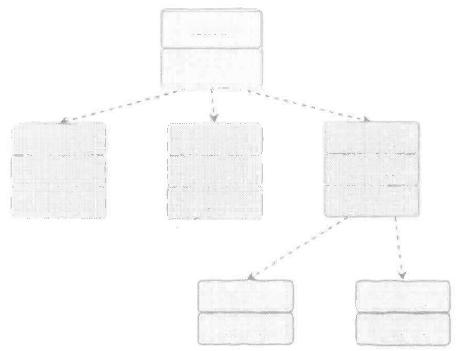
图6-18 消息处理顺序

在具体实现层面中，CephFS 通过以下3个数据结构来表示文件系统中的文件 和目录等信息。这些数据结构的关系如图6-19所示。这些数据结构都是内存中的

文件系统技术内幕

大数据时代海量数据存储之道

数据结构，而且CephFS 也将该数据结构用作缓存内容。



CInode

CDir

CDentry CInode

CDir

CDentry

CInode

CDentry

CInode

CDir

CDentry CInode CDir

CDentry

CInode

图6- 19 CephFS 服务端数据组织相关数据结构

可以看到这里主要有3个数据结构来维护文件的目录树关系，分别是CInode、 CDentry 和 CDir。下面介绍一下这些数据结构的作用。

**1.Clnode 数据结构**

CInode 包含了文件的元数据，这个跟Linux 内核中的inode 类似，每个文件都

有一个CInode 数据结构对应。该数据结构包含文件大小和拥有者等信息。

**2.CDentry 数据结构**

CDentry 是一个黏合层，它建立了inode 与文件名或目录名之间的关系。一个 CDentry只可以连接一个CInode。但是一个CInode 可以被多个CDentry 连接。这

是因为连接的存在，同一个文件的多个连接名称是不同的，因此需要多个CDentry 数 据 结 构 。

**3.CDir 数据结构**

CDir 用于目录属性的inode, 它用于在目录下建立与CDentry 的连接。如果某 个目录有分片，那么一个CInode 是可以有多个CDir 的。

其中，CDir 存在一个与CDentry 一对多的关系，表示目录中的文件或子目录 关系。CInode 与 CDentry 则是文件的元数据信息与文件名的对应关系。

总而言之，在CephFS 的集群端通过CInode 、CDir和 CDentry 等数据结构来组 织 了 文 件 系 统 的 层 级 结 构 。

第6章提供横向扩展的分布式文件系统

**6.6.5** **CephFS数据组织简析**

在本地文件系统中，通常文件系统管理的是一个线性空间。但是CephFS 有些 差异，因为其底层是RADOS 对象集群，其提供的是一个对象的集合。因此，虽然 CephFS 对外呈现的是一个层级结构的文件系统，但是底层数据则是以对象的方式 存储的。

[**6.6.5.1**](6.6.5.1) **数据组织格式**

我们在创建文件系统时其实是创建了两个对象存储池。CephFS 的数据和元数 据也是以对象的形式存储在该存储池中的，那么 CephFS 是如何实现层级结构与对 象之间的转换呢?

我们在前面介绍本地文件系统时知道，文件系统主要实现了对整个磁盘(块设 备)空间管理和文件/目录的数据管理。可以将上述管理概述如下。

(1)磁盘空间管理：主要是负责磁盘空间的申请和释放，通过某种方式标识哪 些空间是被占用的。

(2)文件数据管理：主要建立文件逻辑地址与数据存储位置的关系，能够访问 指定文件位置的数据。

(3)目录数据管理：目录作为一种特殊的文件，其内容是格式化的，保存着文

件名与inode ID之间的对应关系。

由 于CephFS 基于对象来存储数据，因此其实现方式略有不同，接下来介绍一 下CephFS 是如何管理这些内容的。

**1.对于整个存储空间的管理**

因为CephFS 是基于对象存储数据的，其空间的使用基于命名对象，而不是硬 盘那样的线性空间。也就是说，当Ceph 文件系统需要新的存储空间时，只需要向 RADOS 集群申请，即可创建一个对象。对于对象的创建及数据的管理，都是由 RADOS 管理的，因此不存Ceph 文件系统管理整个存储空间的问题。

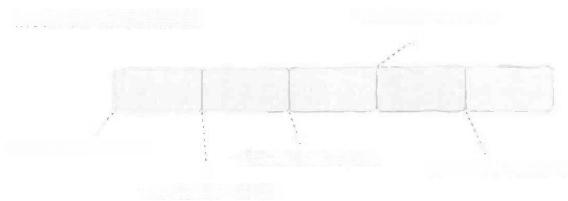
**2** **.对于文件的数据管理**

对于文件的数据管理，目前使用比较多的是索引方式。CephFS 对文件的数据 管理也是采用索引方式，但CephFS 并没有索引块的概念，它采用一种基于计算的 方式来获得索引关系。CephFS 计算规则很简单，就是将文件拆分为固定大小的数 据块(默认为4MB), 然后给每个数据块一个名字。最终，以这个名字作为对象的 名称进行存储。

**文件系统技术内幕**

大数据时代海量数据存储之道

由于存储池是扁平的，因此要求对象名称的唯一性。CephFS 的做法是通过 inode ID和逻辑偏移的方式来标记该数据块。这里的偏移是按照固定大小拆分后的 索引，而非逻辑地址偏移，如图6-20所示。

**inode ID:10000000000** 10000000000.00000003





1000000000000000000







1000000000000000002

10000000000.00000004

10000000000.00000001

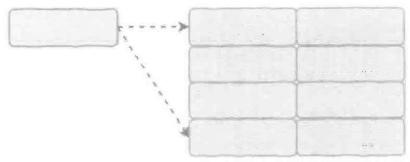
图6-20 文件的数据管理

在图6-20中，文件按照逻辑地址被拆分为4MB的逻辑块。其中0～4MB的数 据位于第1个逻辑块，也就是名称为10000000000.00000000的对象中，其他地址 的数据以此类推。因此，当访问某个地址的数据时，可以很容易地知道对象名称并 进行访问。

**3.对于目录的数据管理**

在本地文件系统相关介绍中我们知道目录是一种特殊的文件，只是其中的数据 有特定的格式，这种格式内容是文件名与inode ID呈现一一对应关系。由于inode ID 确 定 后 ，inode 的内容和数据都可以确定，因此本质上来说通过这种方式就建立 了目录与其中文件的对应关系。

在CephFS 中，目录的实现方式略有不同，它并没有将目录中的文件名等信息 存储在目录对象的数据中，而是存储在目录对象的元数据中。在CephFS 中，目录 中的文件组织(目录项)是以omap(Key-Value) 的形式存储的。换句话说，每个 目录会以自己 inode ID 作为名称在元数据存储池创建一个对象，而目录中的文件 (子目录)等数据则是以该对象的omap 的形式(文件名-inode 对)存在的，而非对 象数据的形式，如图6-21所示。



**目录对象** **文件名**

**文件名**

…

**文件名**

**inode信息** **inode信息**



**inode信息**

图6-21 CephFS 客户端模块

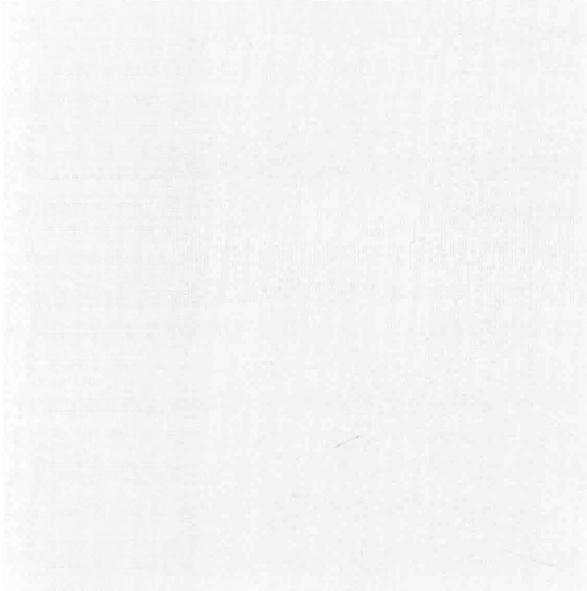
第6章提供横向扩展的分布式文件系统

本质上，CephFS 的这种方式主要借助了KV 数据库(如LevelDB 或 RocksDB 等)的能力。这种目录数据组织方式并非是CephFS 独有的，很多分布式文件系统 都有类似的实现 。

[**6.6.5.2**](6.6.5.2) **数据实例分析**

我们在前面对主要的数据组织与管理方式进行了介绍 ， 大家可能还会觉得有些 抽象，不太容易理解。下面结合实例介绍一下文件系统的数据是如何存储在对象 中 的 。

首先，在创建一个Ceph 文件系统后，其实元数据存储池已经有很多对象了。 这主要包括根目录和一些管理数据所对应的对象，如图6-22所示。其中，根目录 对应的对象为1.00000000,这是因为CephFS 约定了根目录的inode ID是 1 。

sunnyzhang@itworld123:~#rados ls -p fs\_metadata 601.00000000

602.00000000

600.00000000

603.00000000

1.00000000.inode 200.000 -00000

200.00000001

606.00000000

607.00000000

mds0\_openfiles.0 608.00000000

604.00000000

500.00000000

mds\_snaptable

605.00000000

mdsO\_inotable

100.00000000

mds0\_sessionmap 609.00000000

400.00000000

100.00000000.inode 1.00000000

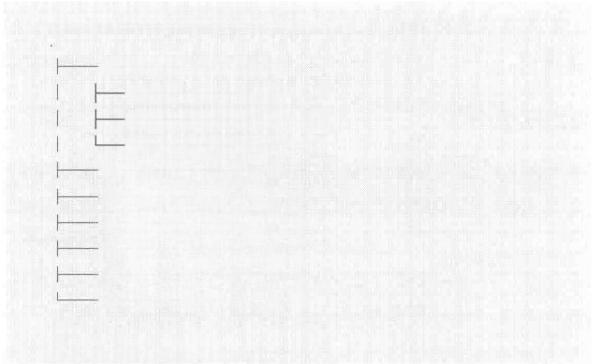
图6-22 元数据对象

为了分析CephFS 数据的放置情况，我们需要在根目录创建如下目录和文件(括 号中的内容是十六进制形式的inode ID)。其 中 ，file1 、file2 和 file3 分别存储6 字节的1、2和3。file4 和 file5 存储12MB 的全0数据。文件和目录的结构如

文件系统技术内幕

大数据时代海量数据存储之道

图6-23所示。



sunnyzhang@itworld123:/mnt/cephfs#tree

dirl(10000000000)

filel (10000000005) file2(10000000006) file3(10000000007)

dir2(10000000001) filel(10000000002) file2(10000000003) file3(1000000-0004)

file4(10000000008) file5(10000000009)

2 directories,8 files

图6-23 文件和目录的结构

需要注意的是，当创建文件或目录时，CephFS 先将数据写到日志，这个逻辑 跟本地文件系统是一致的，其目的是保证在出现系统崩溃等异常情况下文件系统数 据的一致性。因此，创建完成上述数据后需要强制将元数据从日志刷新到存储池。 需要执行如下命令 ：

ceph daemon /var/run/ceph/ceph-mds.gfsl.asok flush journal

**1** **.** **目录数据的验证**

由构造的数据，我们在根目录创建了dir1 和 dir2 两个目录和file1～file5 共 5 个文件。由于这些信息都是存储在根目录对象的omap 中的，我们先看一看omap 的“键”信息。我们可以通过如下命令从根目录的对象中获取omap 的所有“键” 信 息 。

rados listomapkeys 1.00000000-p fs\_metadata

执行命令后可以得到如图6-24所示的结果，可以看到这里的“键”就是我们 创建的文件名和目录名作为关键字构成的，其后面跟着字符“\_head”。

rootegfsl:~共rados listomapkeys 1.00000000 -p fs\_metadata dirl\_head

dir2\_head filel\_head file2\_head file3\_head file4\_head file5\_head

图6-24 目录数据的存储

这 里 面 的omap 是 以 “ 键 ” 的 形 式 存 在 的 ，Value 对 应 的 为inode 信息，如图6 - 25

第 6 章 提 供 横 向 扩 展 的 分 布 式 文 件 系 统

所示为file1 对应的元数据(inode) 信息，这些信息包括该文件关键的元数据信息，如 inode ID、用 户ID 、组 ID 和创建时间等。

|  |  |  |
| --- | --- | --- |
| filel\_head  value(462 bytes): |  |  |
| 00000000020000000000.0000490f | 06 a3010000021 | I.. |
| 00000010000000000100.0000000000 | Ic 7c.855f 78 1 | 1.\_xI |

900000209d 66.33 a4.810000909000000000000001 1.f3

00000030000010000000000000000000002021800 I

0000004000000000400001000000000040000100 @ @.1

0000005000000000000000000000070000000000 1

00000060000001000000 ff ff ff ff ff ff ff ff 0000 I l 00000070 000000000000000000001c 7c 855f 789d 1 …1.\_x.I 0000008066331c 70855f 789d 6633000000000000 |f3.1.\_x.f3.

00000090009003022800000090000000000010000 1. (.

000000a0000000100.0000.000000000000.000010000 1… 1

图6-25 filel 对应的元数据(inode) 信息

如果想要进一步了解inode的详细内容，则可以通过阅读inode序列化的代码 得到，如代码6-1所示(该结构体的部分代码)。该代码的开始部分与图6-25中第 1 0 个 字 节(Of) 对应，前面部分是在其他类中序列化的。

**代** **码** **6** **-** **1** **C** **e** **p** **h** **F** **S** **中** **的** **i** **n** **o** **d** **e** **\_** **t** **数** **据** **结** **构**

**Mds/mdstypes.h**

|  |  |
| --- | --- |
| 633 | template<template<typename>class Allocator> |
| 634 | void inode\_t<Allocator>::encode(ceph::buffer:list &bl,uint64\_t features)const |
| 635 | { |
| 636 | ENCODE\_START(15,6,bl); |
| 637 |  |
| 638 | encode(ino,bl); |
| 639 | encode(rdey,bl); |
| 640 | encode(ctime,bl); |
| 641 |  |
| 642 | encode(mode,bl); |
| 643 | encode(uid,bl); |
| 644 | encode(gid,bl); |
| 645 |  |
| 646 | encode(nlink,bl); |
| 647 | { |
| 648 |  |
| 649 | bool anchored=0; |
| 650 | encode(anchored,bD); |
| 651 | 量 |
| 652 |  |
| 653 | encode(dir\_layout,bl); |
| 654 | encode(layout,bl,features); |
| 655 | encode(size,bl); |
| 656 | encode(truncate\_seq,bl); |
| 657 | encode(truncate size,bl); |

**文件系统技术内幕**

大数据时代海量数据存储之道

encode(truncate\_from,bl);

659 encode(truncate\_pending,bl);

660 encode(mtime,bl): 从上述代码中可以看出，在CephFS 中对数据的持久化是需要经过一个编码的

过程的。而从磁盘读取数据后也是要有解码之后才会实例化到结构体的成员中。这 里需要明白inode 等结构体的成员是如何保存的。

在Ceph 文件系统中，文件的元数据存储在MDS 集群中，而数据则是直接与 OSD 集群交互的。以默认配置为例。由于原则确定，当客户端通过MDS 创建文件 后，客户端可以直接根据请求在文件中逻辑位置确定数据所对应的对象名称。

**2.文件的验证**

前文已述，文件数据对应的对象名称为文件的 inode ID与逻辑偏移的组合，这 样就可以根据该对象名称实现数据的读/写。

以 file4 为例，我们在其中写入12MB的全0数据，因此该文件对应着3个对 象。通过前文我们知道 inode ID为1099511627784(0x10000000008)。查看一下数 据存储池中的对象，如图6-26所示。

rootegfsl:/mnt/cephfs#rados ls-p fs\_data Igrep 10000000008

10000000008.00000001 10000000008.00000000 10000000008.00000002

图6-26 文件数据对应的对象

通过图6-26可以看到，与该文件相关的对象列表，其前半部分为inode ID,后 半部分是文件以4MB 为单位的逻辑偏移。

**6.6.6** **CephFS文件创建流程解析**

CephFS 的流程分析略显复杂，主要包括客户端和集群端两部分的代码。而且 客户端的代码在Linux 内核中，依赖VFS 框架的代码。当然，如果大家有了前面本 地文件系统的基础，就容易理解CephFS 的客户端代码。

[**6.6.6.1**](6.6.6.1) **客户端代码解析**

对于创建文件来说，该请求仍然需要经过VFS 的各种逻辑，差异在于调用的 函 数 指 针 有 所 不 同 。 对 于 创 建 文 件 来 说 ， 其 核 心 代 码 在lookup\_open ( 函 数 中 。 如 果 用户在调用open() 函数时使用了O\_CREAT 选项，且lookup\_open()函数在执行第 3074行代码时没有查到要打开的文件，此时就会执行第3096行代码，也就是调用

第 6 章 提 供 横 向 扩 展 的 分 布 式 文 件 系 统

具体文件系统(CephFS) 的创建函数。可以看出这里调用的是目录函数指针集的 create()函数指针，如代码6- 2所示。

**代码6-2** **lookup\_open()函数**

**fs/namei.c path\_openat->open\_last\_lookups->lookup\_open**

3003 static struct dentry \*lookup\_open(structnameidata\*nd,struct file \*file,

const struct open\_flags \*op,

bool got\_write)

3073

3074

3075

3076

3077

3078

3079

3080

3081

3082

3083

3084

3085

3086

3087

3088

3089

3090

3091

3092

3093

3094

3095

3096

3097

3098

3099

3100

3101

3102

3103

3104

3105

3106

3107

3108

3109

3110

/ 删 除 部 分 代 码

if(d\_in\_lookup(dentry)){

struct dentry\*res=dir\_inode->i\_op->lookup(dir\_inode,dentry,

nd->flags); / 查 找 目 的 文 件

d\_lookup\_done(dentry); f(unlikely(res)){

f(IS\_ERR(res)){

error=PTR\_ERR(res); goto out\_dput;

}

dput(dentry); dentry=res;



if(ldentry->d\_inode &&(open\_flag&O\_CREAT)) // 如果有创建选项

file->f\_mode|=FMODE\_CREATED;

audit\_inode\_child(dir\_inode,dentry,AUDIT\_TYPE\_CHILD\_CREATE);

if(!dir\_inode->i\_op->create){

error=-EACCES; goto out\_dput;

}

error=dir\_inode->i\_op->create(dir\_inode,dentry,mode, open\_flag&O\_EXCL);

if(error)

goto out\_dput;



if(unlikely(create\_error)&&!dentry->d\_inode){

error=create\_error; goto out\_dput;

}

return dentry;

out\_dput:

dput(dentry);

return ERR\_PTR(error);



文件系统技术内幕

大数据时代海量数据存储之道

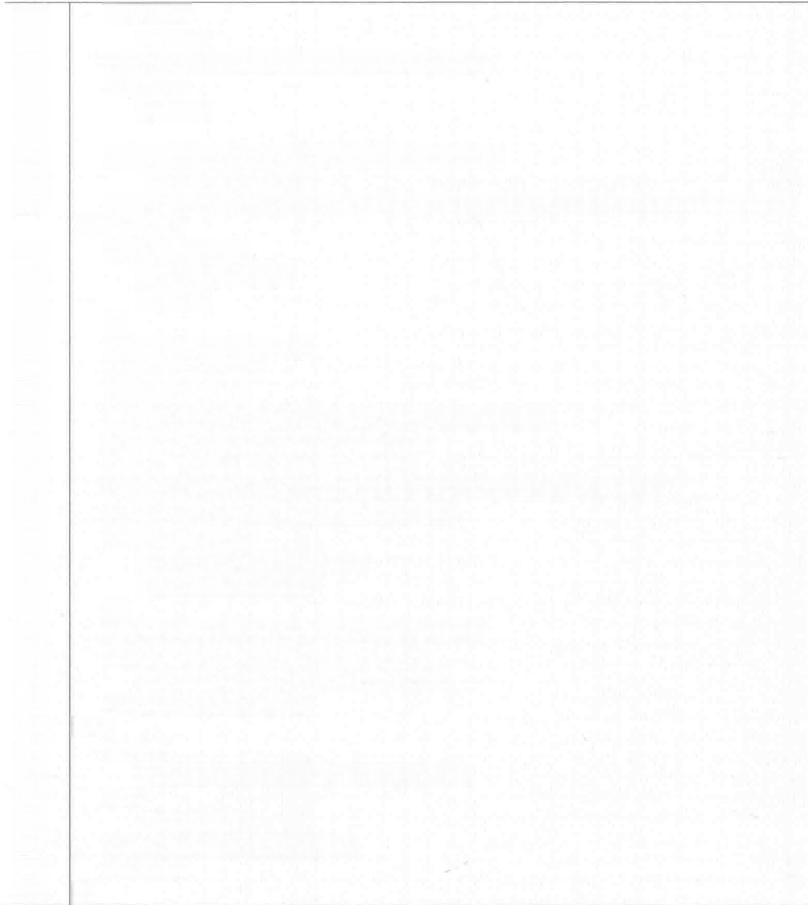
对 于CephFS 文件系统来说，其目录函数指针集如代码6-3所示。而create 指 针对应的函数为ceph\_create() 。也 就 是 说 ，CephFS 文件系统客户端创建文件的代码 逻辑在ceph\_create() 函数中实现。

**代码6-3** **ceph\_dir\_** **iops()函数**

**fs/ceph/dir.e**

|  |  |  |
| --- | --- | --- |
| 1949 | const struct inode\_operations ceph\_dir\_iops={ | |
| 1950 | lookup=ceph\_lookup, | |
| 1951 | .permission=ceph\_permission, | |
| 1952 | getattr=ceph\_getattr, | |
| 1953 | .setattr=ceph\_setattr, | |
| 1954 | .listxattr=ceph\_listxattr, | |
| 1955 | .get\_acl=ceph\_get\_acl, | |
| 1956 | set\_acl=ceph\_set\_acl, | |
| 1957 | mknod=ceph\_mknod, | |
| 1958 | symlink=ceph\_symlink, | |
| 1959 | mkdir=ceph\_mkdir, | |
| 1960 | link=ceph\_link, | |
| 1961 | unlink=ceph\_unlink, | |
| 1962 | rmdir=ceph\_unlink, | |
| 1963 | rename=ceph\_rename, | |
| 1964 | .create=ceph\_create, //创建文件的函数指针 | |
| 1965 | .atomic\_open=ceph\_atomic\_open, | |
| 1966 | }; | |
| 然后进 一 步分析，其实 ceph\_create() 函数没做什么事情，主要是调用了 ceph\_mknod) 函数。创建文件的主要逻辑也是在 ceph\_mknod() 函数中实现的。  **代码6-4** **ceph\_mknod()函数** | | |
| **fs/ceph/dir.cceph\_create->ceph\_mknod** | | |
| 832 | static intceph\_mknod(struct inode \*dir,struct dentry \*dentry, | |
| 833  834 | umode\_t mode,dev\_t rdev) | |
| 835 | struct ceph\_fs\_client \*fsc=ceph\_sb\_to\_client(dir->i\_sb); | |
| 836 | struct ceph\_mds\_client\*mdsc=fsc->mdsc; | |
| 837 | struct ceph\_mds\_request \*req; | |
| 838 | struct ceph\_acl\_sec\_ctx as\_ctx={}; | |
| 839  840 | int err; | |
| 841 | if(ceph\_snap(dir)!=CEPH\_NOSNAP) | |
| 842  843 | return-EROFS; | |
| 844 | if(ceph\_quota\_is\_max\_files\_exceeded(dir)){ | |
| 845 | err=-EDQUOT; | |
| 846  847 | } | goto out; |
| 848 |  | |
| 849 | err=ceph pre\_init\_acls(dir,&mode,&as\_ctx); | |

第 6 章 提 供 横 向 扩 展 的 分 布 式 文 件 系 统

if(err<0)

850

851

852

853

854

goto out;

erT=ceph\_security\_init\_secctx(dentry,mode,&as\_ctx);

if(err<0)

goto out;

855

dout("mknod indir %p dentry %p mode0%ho rdev %din",

856

857

858

dir,dentry,mode,rdev);

req=ceph\_mdsc\_create\_request(mdsc,CEPH\_MDS\_OP\_MKNOD,USE\_AUTH\_MDS);//创建一 个请求结构体

if(IS\_ERR(req)){ err=PTR\_ERR(req); goto out;

859

860

861

862

req->r\_dentry=dget(dentry); req->r\_num\_caps=2;

863

864

865

866

867

868

869

870

871

872

873

req->r\_parent=dir;

set\_bit(CEPH\_MDS\_R\_PARENT\_LOCKED,&req->r\_req\_flags); req->r\_args.mknod.mode=cpu\_to\_le32(mode);

req->r\_args.mknod.rdev=cpu\_to\_le32(rdev);

req->r\_dentry\_drop=CEPH\_CAP\_FILE\_SHARED|CEPH\_CAP\_AUTH\_EXCL; req->r\_dentry\_unless=CEPH\_CAP\_FILE\_EXCL;

if(as\_ctx.pagelist){

req->r\_pagelist=as\_ctX.pagelist; as\_ctx.pagelist=NULL;

874

err=ceph\_mdsc\_do\_request(mdsc,dir,req);// 发送请求 if(lerr &&Ireq->r\_reply\_info.head->is\_dentry)

875

876

877

878

879

880

881

882

883

884

885

886

err=ceph\_handle\_notrace\_create(dir,dentry); ceph\_mdsc\_put\_request(req);

out:

if(lerr)

ceph\_init\_inode\_acls(d\_inode(dentry),&as\_ctx);

else

d\_drop(dentry);

ceph release\_acl\_sec\_ctx(&as\_ctx);

return err; 上

第858行代码用于创建一个请求结构体，这个结构体用于描述一个请求，请求

的类型为CEPH\_MDS\_OP\_MKNOD 。 接下来是关键信息的填充工作，包括文件的 模式、父目录等内容。第875行代码调用ceph\_mdsc\_do\_request() 函数将请求发送 到 MDS 处理。后续的消息发送逻辑并不复杂，本节不再赘述。

[**6.6.6.2**](6.6.6.2) **集群端代码解析**

当客户端发送消息后，MDS 服务的消息接收模块就会收到该消息。然后该模

文件系统技术内幕

大数据时代海量数据存储之道

块将消息分发给MDS 守护进程模块，最后会被路由到服务模块(整个过程见图6-18)。 服务模块负责CephFS 中文件相关的操作，自然创建文件也在其中(第2204行~第 2209行，如代码6-5所示。

代码6-5 dispatch\_client\_request()函数

**mds/server.ce**

2097 void Server::dispatch\_client\_request(MDRequestRef&mdr) 2098 {

2099 //删除部分代码

2100 switch (req->get\_opO)(

2101 case CEPH\_MDS\_OP\_LOOKUPHASH:

2102 case CEPH\_MDS\_OPLOOKUPINO:

2103 handle\_client\_lookup\_ino(mdr,false,false);

2104 break;

2105 case CEPH\_MDS\_OP\_LOOKUPPARENT:

2106 handle\_client\_lookup\_ino(mdr,true,false);

2107 break;

2108 case CEPH\_MDS\_OP\_LOOKUPNAME:

2109 handle\_client\_lookup\_ino(mdr,false,true);

2110 break;

//删除文件查找等分支

2203

2204 case CEPH\_MDS\_OP\_CREATE:// 创建文件的实现逻辑

2205 if(mdr->has\_completed)

2206 handle\_client\_open(mdr);

2207 else

2208 handle\_client\_openc(mdr);

2209 break; 2210

2211 case CEPH\_MDS\_OP\_OPEN:

2212 handle\_client\_open(mdr);

2213 break; 2214

2215

2216

2217 case CEPH\_MDS\_OP\_MKNOD:

2218 handle\_client\_mknod(mdr);

2219 break;

2220 case CEPH\_MDS\_OP\_LINK

2221 handle\_client\_link(mdr);

2222 break;

2223 case CEPH\_MDS\_OP\_UNLINK:

2224 case CEPH\_MDS\_OP\_RMDIR:

2225 handle\_client\_unlink(mdr);

2226 break;

//删除其他分支

第6章提供横向扩展的分布式文件系统

default:

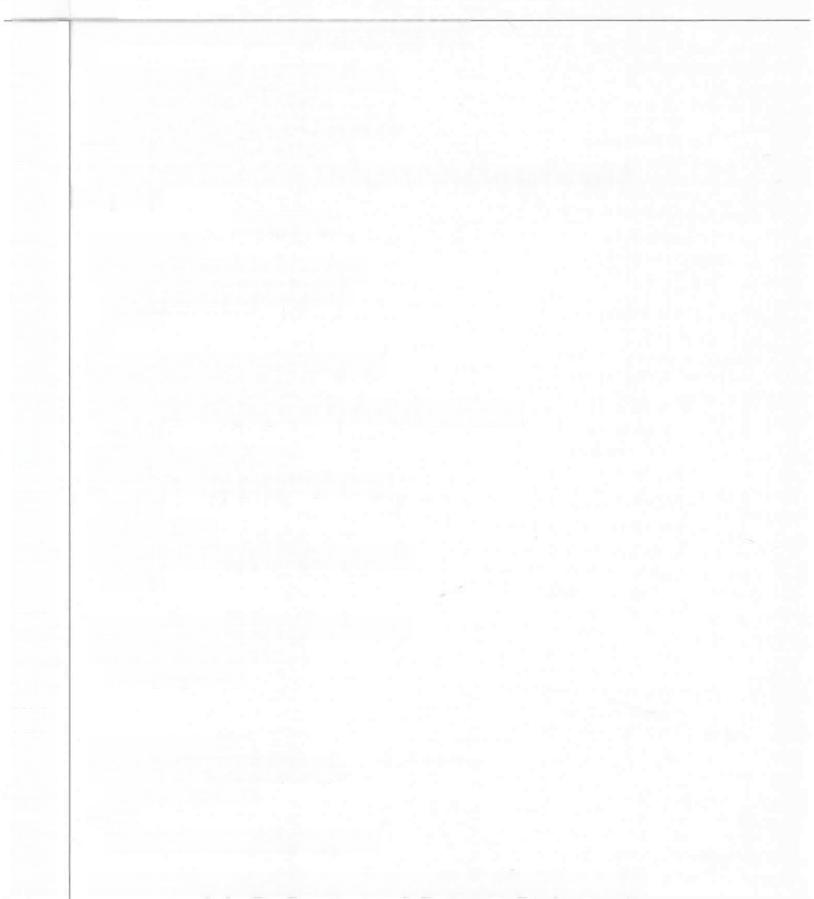
dout(1)<<"unknown client op"<<req>get\_op()<<dendl;

2254 respond\_to\_request(mdr,-EOPNOTSUPP); 2255 

2256 

通过上面代码可以看到，每一个命令都有一个对应的函数进行处理。其实可以 理解 CephFS 实现了 一个自定义的 RPC 。 对于创建文件则是调用 handle\_client\_ mknod() 函数来完成服务端的工作的，如代码6-6所示。

**代码6-6** **handle\_client\_mknod()函数**

**mds/server.ce**

5524 void Server:handle\_client\_mknod(MDRequestRef&mdr)

5525 

5526 MClientRequest \*req=mdr->client\_request;

5527 client\_t client=mdr->get\_client();

5528 set<SimpleLock\*>rdlocks,wrlocks,xlocks;

5529 file\_layout\_t\*dir\_layout=NULL;

5530 CDentry\*dn=rdlock\_path\_xlock\_dentry(mdr,0,rdlocks,wrlocks,xlocks,false,

5531 false,false,

|  |  |  |
| --- | --- | --- |
| 5532 |  | &dir\_layout); |
| 5533 | if(!dn)return; |  |

5534 if(mdr->snapid!=CEPH\_NOSNAP){

5535 respond\_to\_request(mdr,-EROFS);

5536 return;

5537 

5538 CInode \*diri=dn->get\_dirO->get\_inode();

5539 rdlocks.insert(&diri->authlock);

5540 if(Imds->locker->acquire\_locks(mdr,rdlocks,wrlocks,xlocks))

5541 return;

5542 查看该用户对目录的权限

5543 if (lcheck\_access(mdr,diri,MAY\_WRITE))

5544 return;

5545 //检查分片空间

5546 if(lcheck\_fragment\_space(mdr,dn->get\_dirO))

5547 return; 5548

5549 unsigned mode=req->head.args.mknod.mode;

5550 if((mode &S\_IFMT)=0)

5551 mode F=S\_IFREG; 5552

5553

5554 file\_layout\_t layout;

5555 if(dir\_layout &&S\_ISREG(mode))

5556 layout=\*dir\_layout;

5557 else

5558 layout=mdcache->default\_file\_layout; 5559

5560 Clnode\*newi=prepare\_new\_inode(mdr,dn->get\_dir(O,inodeno\_t(req->head.ino),

**文件系统技术内幕**

大数据时代海量数据存储之道

5561 mode,&layout);//创建一个inode节点

5562 assert(newi); 5563

5564 dn->push projected\_linkage(newi); 5565

5566 newi->inode.rdev=req->head.args.mknod.rdev;

5567 newi->inode.version=dn->pre\_dirty);

5568 newi->inode.rstat.rfiles=1;

5569 if(layout.pool\_id!=mdcache->default\_file\_layout.pool\_id)

5570 newi->inode.add\_old\_pool(mdcache->default\_file\_layoutpool\_id);

5571 newi->inode.update\_backtrace(); 5572

5573 snapid\_t follows=mdcache->get global\_snaprealm(->get\_newest seqO;

5574 SnapRealm\*realm=dn->get\_dir()->inode->find\_snaprealm);

5575 assert(follows>=realm->get\_newest\_seqO); 5576

5577

5578 //如果客户端通过MKNOD创建了 一 个常规文件，则会向该文件写入数据(如正在导出NFS)

5579 if(S\_ISREG(newi->inode.mode)){

5580 //在文件上创建一个Capability实例

5581 int cmode=CEPH\_FILE\_MODE\_RDWR;

5582 Capability \*cap=mds->locker->issue\_new\_caps(newi,cmode,mdr->session,

5583 realm,req->is\_replayO);

5584 if(cap){

5585 cap->set\_wanted(0); 5586

5587

5588 newi->filelock.set\_state(LOCK\_EXCL);

5589 newi->authlock.set\_state(LOCK\_EXCL);

5590 newi->xattrlock.set\_state(LOCK\_EXCL); 5591

5592 dout(15)<<"setting a client\_range too,since this is a regular file"<<dendl;

5593 newi->inode.client\_ranges[client].range.first=0;

5594 newi->inode.client\_ranges[client].range.last=

5595 newi->inode.get\_layout\_size\_increment();

5596 newi->inode.client\_ranges[client].follows=follows;

5597 cap->mark\_clientwriteable(); 5598

5599

5600

5601 assert(dn->first=follows+1);

5602 newi->first=dn->first; 5603

5604 dout(10)<<"mknodmode"<<newi->inode.mode<<"rdev"<<newi->inode.rdev 5605 <<dendl;

5606

5607

5608 mdr->ls=mdlog->get\_current\_segment();

5609 EUpdate \*le=new EUpdate(mdlog,"mknod");//日志相关逻辑

第6章提供横向扩展的分布式文件系统

|  |  |
| --- | --- |
| 5610 | mdlog->start\_entry(le); |
| 5611 | le->metablob.add\_client\_req(req->get\_reqid(,req->get\_oldest\_client\_tidO); |
| 5612 | journal\_allocated\_inos(mdr,&le->metablob); |
| 5613 |  |
| 5614 | mdcache->predirty\_journal\_parents(mdr,&le->metablob,newi,dn->get\_dir(), |
| 5615 | PREDIRTY\_PRIMARY|PREDIRTY\_DIR,1); |
| 5616 | le->metablob.add\_primary\_dentry(dn,newi,true,true,true); |
| 5617 |  |
| 5618 | journal\_and\_reply(mdr,newi,dn,le,newC\_MDS\_mknod\_finish(this,mdr, |
| 5619 | dn,newi)); //提交事务 |
| 5620 | mds->balancer->maybe\_fragment(dn->get\_dir(),false); |
| 5621 |  |

在 handle\_client\_mknod() 函数中，前面是一些加锁和检查类的实现(第5540 行～第5550行);然后是创建一个新的inode, 并进行基本信息的初始化工作(第 5560行～第5570行);最后启动一个事务，将数据写入日志中(第5609行～第 5618行)。完成日志落盘后就可以给客户端返回处理结果。

需要注意的是，上述文件虽然创建成功了，并且落盘到日志中，但是真正的 inode 创建则并不一定完成。目前，inode 只是被添加到缓存中，只有缓存刷写时 inode 才会被真正创建。

**6.6.7** **CephFS写数据流程解析**

CephFS 文件系统相关的流程很多，限于篇幅，不可能逐一介绍。除了上节介 绍的创建文件的流程，本节再介绍一下写数据的流程。该流程的典型特点是不需要 与MDS 交互，数据读/写与OSD 直接交互。

[**6.6.7.1**](6.6.7.1) **客户端代码解析**

应用程序的写操作经过VFS 会 由CephFS 的代码逻辑处理。以同步写为例， CephFS 客户端写数据的主线流程如图6-27所示。在该流程中，ceph\_write\_iter() 是 CephFS 注册到VFS 的函数，也是CephFS 写数据流程的起点。最后，CephFS 调 用 ceph\_con\_send() 函数将数据通过网络发送到服务端进行处理。

整个主线的代码逻辑主要是将VFS 发 送 的I/O 请求转换为CephFS 的请求 ceph\_osd request,然后通过网络发送出去。根据VFS 请求的 inode ID和偏移等信 息计算出在CephFS 集群中对应的对象名称，并选择OSD。

需要说明的是，如果VFS 的写请求数据比较大，超出了一个对象的大小，那 么将会被拆分。然后以拆分后的数据为单位进行请求的转换和发送。

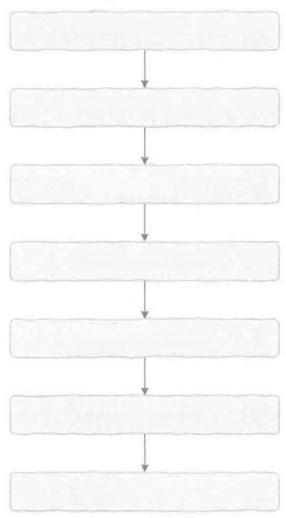
在请求发送之前先要创建CephFS 请求，该过程由函数ceph\_osdc\_new\_request(O

文件系统技术内幕

大数据时代海量数据存储之道

完成。该函数除完成请求的创建外，主要的工作是进行关键信息的初始化。其中， 最主要的是调用calc\_layout() 函数，根据VFS 请求的偏移和大小计算出对应的对象 编号，以及在对象中的偏移和将要写入的数据长度等信息。然后根据计算得到的信 息完成CephFS 请求的初始化。

完成CephFS 请求的初始化后，调用发送请求的接口完成消息的发送。在图6-27 的发送流程中，比较复杂的是 submit\_request(函数，该函数会调用calc\_target() 函 数完成OSD 计算和选择的过程。

ceph\_wite\_tero

ceph\_sync\_write0

ceph\_osdc\_start\_requesto

submit\_requesto

submit\_requesto

send\_requesto

ceph\_con\_send0

图6-27 CephFS客户端写数据的主线流程

最后，调用ceph\_con\_send (函数，该函数其实并不是真正地发送网络数据包， 而是将请求放到一个队列中。真正的数据发送是由CephFS 的网络模块负责的，位 于内核代码的net/ceph 目录下面。这部分内容与CephFS 文件系统无关，本节不再 赘述。

[**6.6.7.2**](6.6.7.2) **集群端代码解析**

对于写数据流程，在CephFS 集群端其实MDS 并不参与什么工作。因为在客 户端已经根据写请求的偏移等信息计算出了应该由哪个OSD 来处理该写请求。因 此，此时写请求就是客户端与对应的OSD 直接交互的过程。

第6章提供横向扩展的分布式文件系统

关于OSD 如何接收写数据请求，大家可以参考《Ceph源码分析》[24中的第6 章内容，该书对对象数据的读/写流程进行了非常详细的分析。

**6.7** **分** **布** **式** **系** **统** **实** **例** **之GlusterFS**

GlusterFS是一个非常著名和典型的开源分布式文件系统，该开源分布式文件 系统与GFS 、CephFS 最大的区别是没有专属的元数据节点，也就是GlusterFS 采 用 的是无中心架构。本节将深入地介绍一下GlusterFS 的相关内容。

关于GlusterFS 有一些特殊的概念，我们这里做一下简单的介绍。了解这些概 念有助于我们对GlusterFS 架构及后续代码的理解。

( 1 ) 卷 (Volume): 在 GlusterFS 中，卷是一个逻辑的存储单元，与NFS 导 出 目录类似。GlusterFS 通过逻辑的卷来实现分布式的特性，如数据分布、数据副本和 数据分片等。另外，需要注意区分Linux 中 LVM 的卷概念。

(2)存储块(Brick): 存储块是服务端最基本的存储单元，通常在服务端对应 着一个目录。

(3)转换器(Translator): 转换器是GlusterFS 的基本功能单元，每个转换器实 现一个小特性，GlusterFS 通过不同的转换器堆叠的方式实现复杂的功能和特性。

**6.7.1** **GlusterFS的安装与使用**

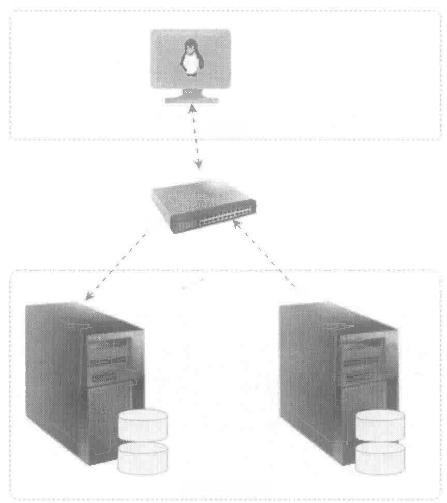
为了方便大家学习，我们先介绍一下如何安装部署该分布式存储。当然，这里 安装的系统只是为了方便大家学习，并不可以用于生产环境。如果想要用于生产环 境，则大家需要进行严格的规划，并且对安全问题进行处理。

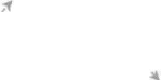
[**6.7.1.1**](6.7.1.1) **安装环境说明**

在安装之前，先说明一下需要的资源。如图6-28所示，构建一个包含两个节 点的GlusterFS 集群，同时有另外一个客户端实现对文件系统的访问。如果没法找 到这些资源，则可以用其中一个服务端兼做客户端。

**文件系统技术内幕**

**大数据时代海量数据存储之道**

客户端



GlusterFS集群

图6-28 GlusterFS集群拓扑

为了方便后续安装，列出GlusterFS 集群各个节点的配置信息，如表6-1所示。 在该表中我们主要关注每个节点的主机名、IP 地址和需要的磁盘资源。列出这些信 息主要是为了后续方便描述，大家在具体安装时可以适当调整。

**表6-1** **GlusterFS集群各个节点的配置信息**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **节** **点** **名** **称** | IP地址 | 主 机 名 | 磁 盘 | 描 述 |
| gfs01 | <192.168.2.117> | gfsl | /dev/sdd | 集群节点 |
| gfs02 | <192.168.2.115> | gfs2 | /dev/sdd | 集群节点 |
| client | <192.168.2.113> | client | 无 | 客户端 |

一般很难准备这么多物理服务器。得益于虚拟化技术，我们可以通过虚拟机来 模拟。目前，市面上的虚拟机软件比较多，如VirtualBox、VMWare 等。关于虚拟 机的安装与配置的内容不在本书范围内，请参考相关书籍或自行上网查找。

由于是测试环境，因此我们可以将系统的防火墙关闭，这样可以避免很多安装 问题。下面介绍一下环境的基本设置流程，具体操作命令不做介绍，大家可以自行 上网检索相关配置命令。

(1)关闭防火墙。

(2)配置主机名。

(3)格式化磁盘。为了简单起见，我们可以不对磁盘进行分区，而是直接对整 个磁盘进行格式化。

**第** **6** **章** 提供横向扩展的分布式文件系统



(4)设置自动挂载。由于GlusterFS 服务是守护进程，随系统自动启动。为了 保 证 GlusterFS 在运行时可以访问存储资源，因此这里需要设置随系统启动自动 挂载。

[**6.7.1.2**](6.7.1.2) **安** **装GlusterFS**

接下来介绍 一 下 GlusterFS 在 Ubuntu 18.04 Server 版本的安装过程。由于 GlusterFS 已 经 有Ubuntu 18.04 Server配套的发行包，因此安装还是比较简单的。下 面介绍一下如何安装GlusterFS集群端的软件包。

(1)先添加GlusterFS 的 PPA 库，命令如下：

sudo apt-get install software-properties-common sudo add-apt-repository ppa:gluster/glusterfs-5

(2)更新系统，命令如下： apt update

(3)安装服务端软件包，命令如下： apt install glusterfs-server

两个服务端的节点都有执行上述命令。如果系统没有报错，服务端的软件就安 装成功了。

客户端的安装与集群端的安装没有太大差别，差别在于安装的软件包不同。首 先添加PPA 库和更新系统(参考集群端安装),然后安装软件包，命令如下：

apt install glusterfs-client

[**6.7.1.3**](6.7.1.3) **系统配置与使用**

完成集群端的软件安装之后，我们就可以配置集群。GlusterFS 的配置也是比

较简单的。下面介绍一下配置步骤。

**1.创建集群**

创建集群用于将多个节点构建为一个存储集群。本实例一共有两个节点，因此 只需要执行一条命令即可。这里在gfs2 节点执行如下命令：

gluster peer probe gfsl

创建成功后可以通过命令查看一下状态，如图6-29所示。

**文件系统技术内幕**

**大数据时代海量数据存储之道**

rootagfs2:~#qluster peer status Number of Peers:1

Hostname:gfsl

Uuid:fabad51b-d126-4e9b-821b-B15bc3c0a391 State:Peer in Cluster(Connected)

图6-29 GlusterFS 集群的状态

如果集群的节点数量多于两个，那么需要将gfsl 逐次替换为其他节点。其原 理也就是建立集群中各个节点的联系。

**2** **.** **创建逻辑卷**

集群创建成功后就可以创建文件系统了，也就是创建 一 个逻辑卷。本实例是创 建 一 个副本卷，命令如下：

gluster volume create rep\_volreplica 2\ gfs1:/mnt/gluster/rv1\

gfs2:/mnt/gluster/rv1

在上述命令中，rep\_vol 表示逻辑卷的名称，replica 表示副本卷，数字2表示 副本卷的数量。接下来的两行命令是两个服务器上的资源(brick), 分别表示服务 器的主机名和路径。后续写入的数据会存储在该路径下。

逻辑卷创建成功后需要执行如下命令来启动该逻辑卷： glustervolume start rep\_vol

然后可以通过命令查看该逻辑卷的信息或状态。以查看信息为例，其相关命令 和获得的信息如图6-30所示。

rootegfs2:~#gluster volume info rep\_vol

Volume Name:rep\_vol Type:Replicate

Volume ID:83cccbb7-e348-43bd-a4dc-ccaccabb1d84 Status:Started

Snapshot Count:0

Number of Bricks:1x2=2

Transport-type:tcp Bricks:

Brick1:gfsl:/mnt/gluster/rvi Brick2:gfs2:/mnt/gluster/rv1 Options Reconfigured:

transport.address-family:inet nfs.disable:on

performance.client-1o-threads:off

图6-30 GlusterFS 逻辑卷的信息

**第6章** 提供横向扩展的分布式文件系统



也可以通过命令来获取逻辑卷的状态，具体命令及获得的状态实例如图6-31 所示。通过图6-31可以看出，其中，包括监听端口和在线状态等信息。

rootegfs2:~#gluster volume status rep\_vol Status of volume;rep\_vol

Gluster process TCP Port RDMA Port Online Pid

Brick gfsl:/mnt/gluster/rvi Brick gfs2:/mnt/gluster/rv1 Self-heal Daemon on localhost Self-heal Daemon on gfsl

Task Status of Volume rep\_vol



There are no active volume tasks

49152

N/A

5421

N/A N/A

图6 - 31 GlusterFS 逻辑卷的状态

**3** **.** **在客户端挂载GlusterFS** **分布式文件系统**

如果逻辑卷运行正常，那么可以在客户端挂载该逻辑卷。执行如下命令后，在 /mnt/glusterfs 目 录 中 的 内 容 就 是 逻 辑 卷 根 目 录 的 内 容 ：

mkdir-p/mnt/glusterfs

mount-tglusterfs gfs1:/rep\_vol/mnt/gfsclient/

至此，创建了一个完整的GlusterFS 环境，我们可以以此环境为基础来完成后 续GlusterFS 相关内容的学习。如果大家想了解更多的安装细节及生产环境的配置 内容，可以参考GlusterFS 的官网。

**6.7.2** **GlusterFS整体架构简析**

GlusterFS 属于无中心节点架构的分布式文件系统。对于GlusterFS 来说，无中 心架构是指服务端(也就是存储集群)并没有一个或多个专门的元数据服务器来维 护整个文件系统的元数据。那么在没有元数据服务器的情况下，GlusterFS 如何进 行整个文件系统集群的数据管理?而整个存储系统又如何对外提供统一的命名 空间?

在GlusterFS 中，整个文件系统的元数据是借助本地文件系统来实现的。比如， 在客户端创建一个文件，文件的管理是在本地文件系统完成的。对于GlusterFS 来 说，并没有一个特殊的地方来对这个文件进行管理。

在GlusterFS中，整个存储系统对外提供的统一命名空间是由服务端的配置信 息和客户端的软件来完成的。其中，服务端卷配置信息描述了组成一个卷的所有存 储块及其关系，而客户端的软件则负责将各个存储块的内容聚合，然后将结果展示

**文件系统技术内幕**

大数据时代海量数据存储之道

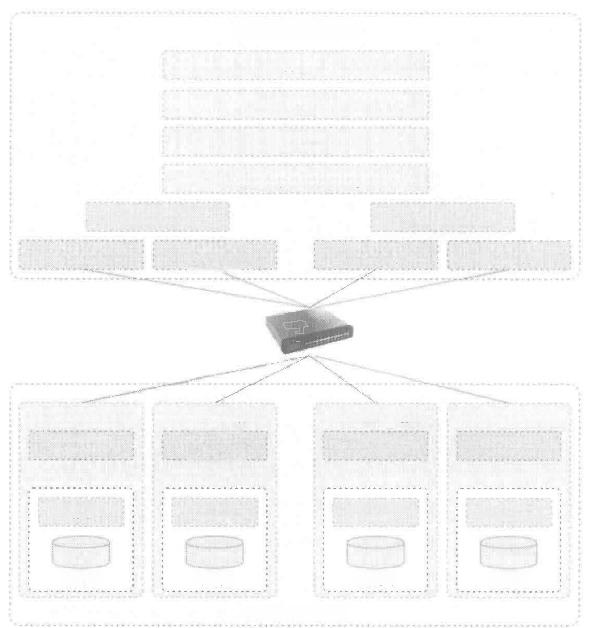
给 用 户 。

在软件实现层面，GlusterFS 分为客户端和服务端两部分软件。GlusterFS 的大 部分特性都是在客户端实现的，如数据副本、数据条带化、数据分布和I/O 缓存等。 在客户端的这些特性采用堆叠的方式实现，也就是一个特性在另外一个特性的上面 来 实 现 。

采用堆叠的方式实现这些特性使得GlusterFS 的代码逻辑非常清晰，降低了阅 读成本。其实这种分层结构也并非GlusterFS 的原创，很多软件都采用类似的架构 模式，如协议栈、Linux 的块设备栈和Windows 的 I/O 栈等。

在GlusterFS 中，实现堆叠的基本组件称为转换器 (Translator, 简 称xlator)。

转换器实现了一些基本特性，而通过不同转换器的组合形成了GlusterFS 所具备的 特性。图6-32所示为GlusterFS 的整体架构。其中，主要功能都是在客户端实现的， 如I/O 缓存、预读、数据条带化和数据副本等。另外，用于实现客户端与服务端之 间通信的模块也是通过转换器实现的。



GlusterFS客户端

VFS

I/O缓存

预读

**数据分布/数据条带化**

**数据副本** **数据副本**

|  |  |  |  |
| --- | --- | --- | --- |
| 客户端 | 客户端 | 客户端 | 客户端 |
| GlusterFS服务器  服务端 | GlusterFS服务器  服务端 | GlusterFS服务器  服务端 | GlusterFS 服务器  服务端 |
| XFS | XFS | XFS | XFS |
| brick | brick | brick | brick |

GlusterFS集群

图6 - 32 GlusterFS 的整体架构

第 6 章 提 供 横 向 扩 展 的 分 布 式 文 件 系 统

在GlusterFS 的整体架构中，服务端实现得相对简单，其入口是服务端(Server) 转换器，该转换器用于实现网络通信。然后是资源管理和磁盘访问等相关的转换器。

客户端的转换器实现得要复杂一些，而且多个转换器之间通过堆叠可以实现丰 富的特性。以图6-32为例，底层是网络客户端转换器，实现与服务端的通信。再往 上是分布式数据副本和数据分布/数据条带化转换器，两者结合可以实现一个复合 卷，这样可以满足性能和数据可靠性的要求。再往上，还有预读和I/O 缓存等转换 器，实现了文件系统在某些场景下加速的功能。

**6.7.3** **转换器与转换器树**

在GlusterFS 中最为重要的就是转换器(为了匹配代码，容易理解，本书使用 xlator来代替转换器)了。如前文所述，GlusterFS的所有特性都是通过xlator实现 的。前文在介绍GlusterFS 的整体架构时就已经介绍过xlator 的内容。对应的实现 代码在xlators 目录中，图6-33是xlators 目录结构，该目录下的子目录是对xlator 的分类。而每个分类下面又有一个或多个xlator。以 cluster为例，其下是涉及集群 相关的xlator, 如副本卷、分布式卷和纠删码卷的具体实现都在其中。

曰 glusterfs 田 git

github 田 api

build-aux 田 cli

田 contrib

田 doc

田 events 困 extras

田 geo-replication

田 glusterfssi4project 田 glusterfsd

田 heal

田 libglusterd 田 libglusterfs

田 tests

tltors

图6-33

田 l

features

田 meta

mgmt 田 mount 田 nfs

performance 匣 playground 田 protocol

田 storage 田 system

xlators目录结构

无论是在客户端还是服务端，在进程启动之后都会初始化一个xlator 树，存储 的特性正是通过这个树来实现的。以客户端为例，如代码6-7所示为GlusterFS 客 户 端 卷 的 配 置 信 息 。 配 置 信 息 分 为 若 干 段 ， 每 段 由volume 关 键 字 开 头 ，end-volume

结尾。这里虽然以volume 作为关键字，其实是一个xlator 实例，我们可以将其理

文件系统技术内幕

大数据时代海量数据存储之道

解为具备某个 xlator 特性的逻辑卷。而不同的逻辑卷堆叠到一起形成用户使用的 卷，这种堆叠关系也就是xlator树 。

在每段的配置信息中有两个内容是比较关键的， 一个是类型(type), 另 一个是 子 卷(subvolumes) 。 其中，类型表示该逻辑卷所对应的xlator, 如第2行代码所示， 它对应着xlators 目录下面的xlator 的子路径。子卷则是该逻辑卷下层的逻辑卷，如 果没有子卷字段，则表示该逻辑卷已经是xlator 树的叶子节点，如rep\_vol-client-0 和 rep\_vol-client-1。

**代码6-7** **GlusterFS客户端卷的配置信息**

|  |  |
| --- | --- |
| 1 | volume rep\_vol-client-0 |
| 2 | type protocol/client |
| 3 | //删除一些选项 |
| 4 | option transport-type tcp |
| 5 | option remote-subvolume/mnt/gluster/rvI |
| 6 | option remote-host gfs1 |
| 7 | option ping-timeout 42 |
| 8 | end-volume |
| 9 |  |
| 10 | volume rep\_vol-client-1 |
| 11 | type protocol/client |
| 12 | 删除一些选项 |
| 13 | option transport-type tcp |
| 14 | option remote-subvolume/mnt/gluster/rvl |
| 15 | optionremote-host gfs2 |
| 16 | option ping-timeout 42 |
| 17 | end-volume |
| 18 |  |
| 19 | volume rep\_vol-replicate-0 |
| 20 | typecluster/replicate |
| 21 | option use-compound-fops off |
| 22 | optionafr-pending-xattr rep\_vol-client-0,rep\_vol-client-1 |
| 23 | subvolumes rep\_vol-client-0 rep\_vol-client-1 |
| 24 | end-volume |
| 25 |  |
| 26 | volume rep\_vol-dht |
| 27 | type cluster/distribute |
| 28 | option lock-migrationoff |
| 29 | subvolumes rep\_vol-replicate-0 |
| 30 | end-volume |
| 31 |  |
| 32 | volume rep\_vol |
| 33 | type debug/io-stats |
| 34 | option count-fop-hits off |
| 35 | option latency-measurementoff |
| 36 | option log-level INFO |
| 37 | subvolumes rep\_vol-dht |

第6章提供横向扩展的分布式文件系统

|  |  |
| --- | --- |
| 38 end-volume | |
| 如果我们按照卷与子卷的关系看一下代码6-7就会发现，xlator 树的关系与配 置文件各个段的关系是倒置的。也就是树根是最下面的一个配置段，而配置最上面 的内容是树的叶子节点。    在服务端的每个存储块都会有一个配置信息，该配置信息也是一棵xlator 树。 代码6-8所示为一个服务节点简化后的配置信息。可以看到，其树根是server,它  是 RPC 的服务端，然后依次向下，最后是posix 。这个其实就是访问文件系统的 xlator。  **代** **码** **6** **-** **8** **一** **个** **服** **务** **节** **点** **简** **化** **后** **的** **配** **置** **信** **息** | |
| 1 | volume rep\_vol-posix |
| 2 | type storage/posix |
| 3 | option shared-brick-count 0 |
| 4 | option volume-id 83cccbb7-e348-43bd-a4dc-ccaccabbld84 |
| 5 | option directory /mnt/gluster/rvl |
| 6 | end-volume |
| 7 |  |
| 8 | volume rep\_vol-io-threads |
| 9 | type performance/io-threads |
| 10 | subvolumesrep\_vol-posix |
| 11 | end-volume |
| 12 |  |
| 13 | volume/mnt/gluster/rvl |
| 14 | type performance/decompounder |
| 15 | subvolumes rep\_vol-io-threads |
| 16 | end-volume |
| 17 |  |
| 18 | volume rep\_vol-server |
| 19 | type protocol/server |
| 20 | //删除部分选项 |
| 21 | option auth-path /mnt/gluster/rvl |
| 22 | option auth.login.f8a8b...d9a797.password b3e7...300534 |
| 23 | option auth.login./mnt/gluster/rv1.allow f8a8b...d9a797 |
| 24 | option transport.address-family inet |
| 25 | option transport-type tcp |
| 26 | subvolumes /mnt/gluster/rvl |
| 27 | end-volume |

前文从配置文件方面介绍了GlusterFS是如何组织和构建一个xlator树的。那 么从代码实现层面又是什么样的呢?各层xlator 之间的请求又是怎么传递的呢?接 下来介绍相关的内容。

在GlusterFS 中 ，xlator 是通过一个名为xlator 的结构体来表示的。如代码6-9 所示，在该结构体中包含子卷列表和父卷列表(第777行～第778行),正是这两

文件系统技术内幕

大数据时代海量数据存储之道

个成员使得xlator 可以构建一棵树。

xlator 结构体有两个比较重要的成员是fops 和 cbks, 这两个成员分别是操作函 数集和回调函数集。操作函数集对应文件操作，从Fuse 触发的文件操作基本上通 过调用下一级xlator 对应的函数来逐层传递，最后通过网络发送到服务端。

**代码6-9** **xlator结构体的定义** **libglusterfs/src/glusterfs/xlator.h**

|  |  |  |
| --- | --- | --- |
| 770  771  772  773  774  775  776  777  778  779  780  781  782  783  784  785  786  787  788  789  790  791  792  793  794  870 | struct\_xlator{  char \*name; char\*type;  char \*instance\_name; xlator\_t\*next;  xlator\_t\*prev;  xlator\_list\_t\*parents; xlator\_list\_t\*children; dict\_t\*options; | |
| void \*dlhandle;  struct xlator\_fops\*fops; struct xlator\_cbks\*cbks;  struct xlator\_dumpops\*dumpops;  struct list\_head volume\_options;  void(\*fini)(xlator\_t \*this); nt32\_t(\*init)(xlator\_t\*this); | //操作函数集 //回调函数集  //卷选项链表  //用于实现对当前xlator的反初始化 //用于实现对当前xlator的初始化 |
| int32\_t(\*reconfigure)(xlator\_t\*this,dict\_t\*options); int32\_t(\*mem\_acct\_init)(xlator\_t\*this);  int32\_t(\*dump\_metrics)(xlator\_t\*this,int fd);  event\_notify\_fn\_t notify;  /删除部分代码 上 | |

当系统完成初始化后，xlator 树已经构建。xlator 之间通过children 成员和parents 成员构建了各级节点的关系。父节点可以通过children 成员知道当前节点的下一级 节点的xlator, 子节点通过parents 成员可以知道其父节点。

在GlusterFS 中定义了两个宏来实现上一级节点到下一级节点的调用和反向调 用，它们分别是STACK\_WIND 和 STACK\_UNWIND 。另外，在调用下一级函数之 前都会分配一个结构体 call\_frame\_t 的空间，被称为帧。帧用于存储下一级xlator 需要的关键信息，如回调函数、xlator 指针、上一级帧和其他信息。

代码6-10所示为STACK\_WIND\_COMMON 的定义，在该宏定义中首先创建

**第** **6** **章** 提供横向扩展的分布式文件系统



一个新的帧并填充必要信息(第325行～第338行),然后在第346行代码中调用 下 一 级xlator 的函数，并且会将新的帧传递给该函数。可见，通过这种方式实现了 请求从上一级xlator 到下 一 级xlator 的发送。

**代码6-10** **STACK** **WIND** **COMMON的定义** **libglusterfs/sre/glusterfs/stack.h**

|  |  |
| --- | --- |
| 319 | #define STACK\_WIND\_COMMON(frame,rfn,has\_cookie,cky,obj,fn,params...) |
| 320 | do{ |
| 321 | call\_frame\_t\*\_new=NULL; |
| 322 | xlator\_t\*old\_THIS=NULL; |
| 323 | typeof(fn)next xl\_fn=fn; |
| 324 |  |
| 325 | \_new=mem\_get0(frame->root->pool->frame\_mem\_pool); |
| 326 | if(!\_new){ |
| 327 | break; |
| 328 |  |
| 329 | typeof(fn##\_cbk)tmp\_cbk=rfn; |
| 330 | \_new->root=frame->root; |
| 331 | \_new->this=obj; |
| 332 | \_new->ret=(ret\_fn\_t)tmp\_cbk;/上一级的回调函数，任务完成时调用 |
| 333 | \_new->parent=frame; //上一级的帧 |
| 334 |  |
| 335 | new->cookie=((has\_cookie=1)?(void\*)(cky):(void \*)\_new); |
| 336 | \_new->wind\_from=\_FUNCTION\_; |
| 337 | \_new->wind\_to=#fn; |
| 338 | \_new->unwind\_to=#rfn; |
| 339 |  |
| 340 | fn##\_cbk=rfn; |
| 341 | old\_THIS=THIS; |
| 342 | THIS=obj; |
| 343 |  |
| 344 | new->op=get\_fop\_index\_from\_fn((new->this),(fn)); |
| 345 |  |
| 346 | next\_xl\_fn(\_new,obj,params);/调用下一级函数 |
| 347 | THIS=old\_THIS; |
| 348 | }while(0) |

由于在STACK\_WIND\_COMMON 宏定义中进行帧初始化时会注册一个回调 函 数 ( 第 3 3 2 行 ) 。 当 底 层 的xlator 完成任务需要通知上 一 级时，通过调用 STACK\_UNWIND 实现。而该宏定义本质上是通过这里注册的回调函数来通知上一 级的。

**6.7.4** **GlusterFS数据分布与可靠性**

对于分布式文件系统来说，最关键的是实现横向扩展和数据的可靠性。

**文件系统技术内幕**

大数据时代海量数据存储之道

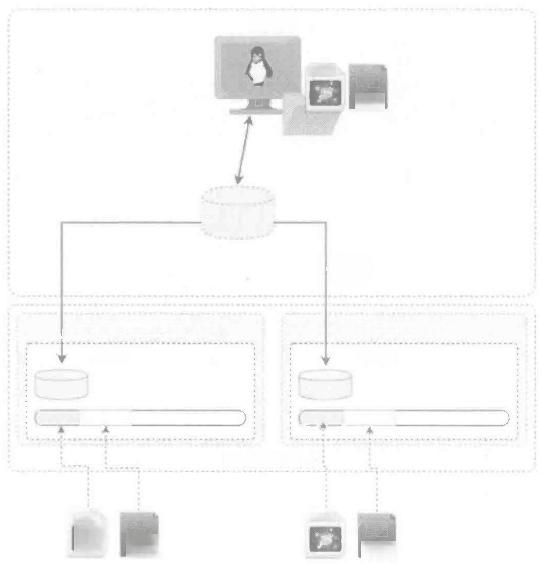
GlusterFS 通 过 3 种xlator 来实现上述特性，并且可以将上述特性进行堆叠，进而实 现更加复杂的特性。

[**6.7.4.1**](6.7.4.1) **数据的副本**

数据的可靠性是任何存储系统要解决的问题。在Ceph 中通过副本技术和纠删 码技术来保证数据的可靠性。在GlusterFS 中也包含副本和纠删码两种数据冗余技 术，本节以副本技术为例进行介绍。

在GlusterFS 中是通过卷来实现数据的组织和管理的，卷表示一个存储单元， 在该单元中的所有数据采用了相同的数据处理算法。而且卷也是GlusterFS 集群导 出文件系统目录的单元，客户端通过挂载该卷来实现一个目录的挂载。

GlusterFS 的副本技术通过副本卷(Replication Volume)实现。 一个副本卷可以 由一个或多个存储块组成。当由多个存储块组成时，这个卷就包含多个副本，多个 存储块中的数据是相同的。图6-34所示为具有两个副本的副本卷示意图，可以看 出，当在客户端写入一个文件时，客户端的软件会将数据分发到两个不同的服务器， 并存储相同的数据。

客户端

PNG

TXT

**gfs1:/mnt/glusterfs/rv1**

brick

副本卷

gfs1:/mnt/glusterts/rv1 brick



GlusterFS集群

PNG

2

PNG

TXT TXT

图6-34 具有两个副本的副本卷示意图

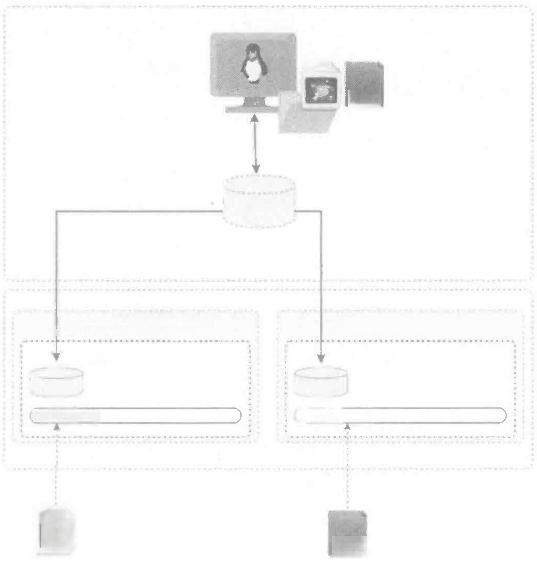
第6章提供横向扩展的分布式文件系统

[**6.7.4.2**](6.7.4.2) **数据的分布/分片**

副本卷可以实现数据的保护，数据的分布则通过另外一种类型的卷实现，这就 是分布式卷(Distributed Volume)。数据的分布/分片是GlusterFS 对一个数据集放置 在集群多个节点的方法。

如图6-35所示，对于有两个存储块的分布式卷，当客户端写入两个文件时， 这两个文件通常会被存储在两个不同的物理节点上。当然，这依赖于分布算法的计 算结果，如果只有两个文件则可能会被放置在同一个节点。但是，当客户端写入比 较多的文件时，通常这两个存储块上的文件数量是均衡的。

在实际生产环境中，承载分布卷的存储块要更多一些，而且通常与业务的负载 相关。如果业务的负载非常大，则存储块的数量有可能达到几十个。



客户端

PNG

TXT

分布卷

**gfs1:/mnt/glusterfs/rv1**

brick

PNG

2

**TXT**

gfs1:/mnt/glusterfs/rv1

brick

图6-35 分布卷原理示意图

[**6.7.4.3**](6.7.4.3) **数据的条带化**

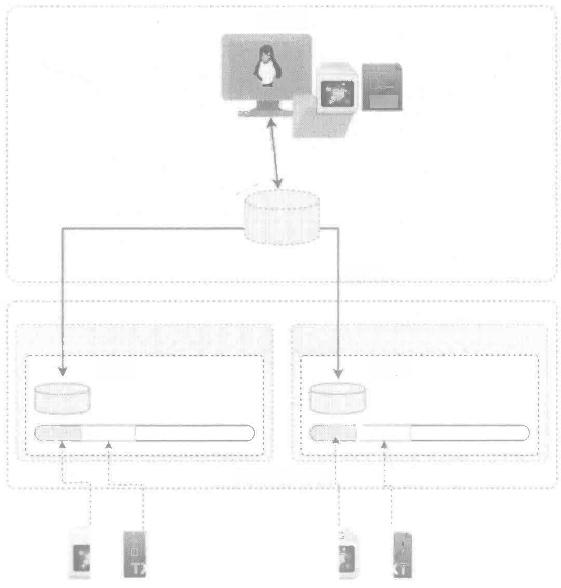
条带化处理是GlusterFS 对单个数据对象(也就是文件)进行分布式处理的技 术。对于客户端的一个文件，客户端会将其拆分成指定大小的数据块分别写入不同 的节点。数据的条带化是通过条带卷(Stripe Volume)实现的。如图6-36所示，当

文件系统技术内幕

大数据时代海量数据存储之道

客户端写入一个文件时，该文件的一部分被写入节点gfsl, 另一部分被写入节点

gfs2 。当然，这里只是一个示意图，实际情况要比较复杂。



客户端

PNG

TXT

条带卷 .

gfs1:/mnt/glusterfs/rv1 brick

PN

gfs1:/mnt/glusterfs/rv1 brick

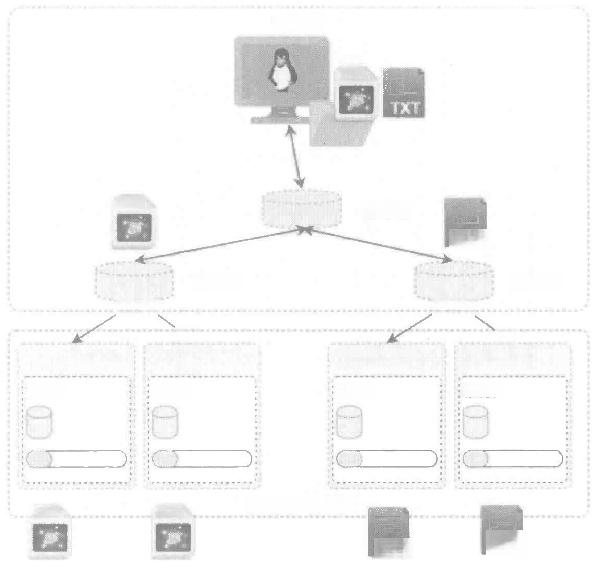
图6-36 条带卷原理示意图

[**6.7.4.4**](6.7.4.4) **副本与分片的堆叠**

前文介绍的3种不同类型的卷解决了分布式中常见的3种问题，但有其局限 性。比如，在分布式文件系统中，我们既希望数据是高可靠的，又希望可以横向扩 展。但是似乎上述3种类型的卷都无法同时搞定。

在 GlusterFS 中可以实现上述特性的堆叠，如将分布卷和副本卷堆叠。也就是 先基于存储块创建副本卷，再将多个副本卷创建为一个分布卷，如图6-37所示。 此时最终的卷既具有分布卷的特性，可以将数据按照哈希算法分散存储；又具有副 本卷的特性，将一份数据同时存储两个或两个以上数据副本。

第 6 章 提 供 横 向 扩 展 的 分 布 式 文 件 系 统



客户端

PNG

分布卷

TXT

副本卷

K



gfs3 gfs4

brick brick

GlusterFS集群

TXT TXT

副本卷

brick

brick



gfs2

gfs1



PNG

PNG

PNG

图6 - 37 分布卷与副本卷的堆叠

通过这种方式的堆叠，可以在保证数据可靠性的前提下实现集群的横向扩展， 也就是通过多个物理节点来为用户提供更大的容量和更高的性能。

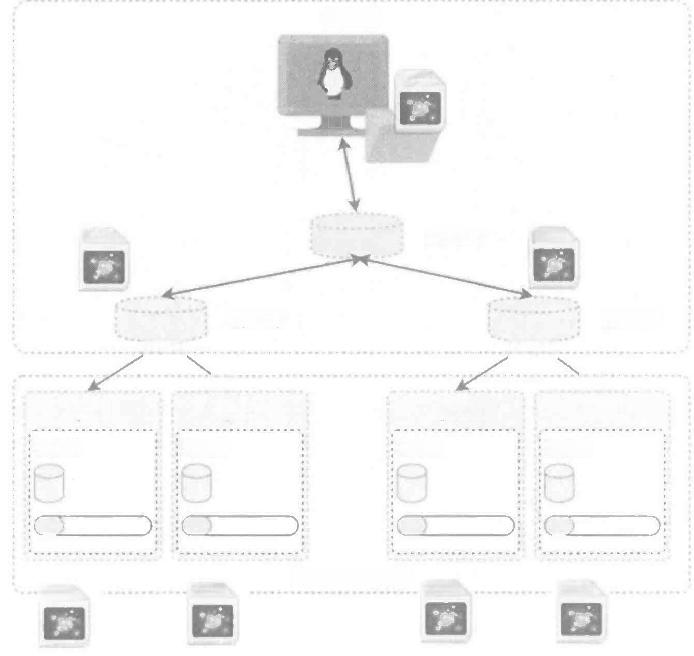
[**6.7.4.5**](6.7.4.5) **副本与条带的堆叠**

在GlusterFS 中，不仅可以将分布卷与副本卷堆叠，而且还可以将条带卷与副 本卷堆叠。图6-38所示为条带卷与副本卷堆叠。在该类型的卷中，当客户端写入 一个文件时，会按照分片大小写入不同的副本卷中，然后副本卷进一步将数据分别 写入不同的存储块中。当然，这里展示的只是一个示意图，实际条带数量根据配置 情 况 而 定 。

这种堆叠卷通过副本来保证集群数据的可靠性。通过条带解决大文件访问的性 能问题，对大文件进行访问时会被分散到集群的多个节点上，通过多个节点来承载 大 文 件 的 访 问 。

**文件系统技术内幕**

**大数据时代海量数据存储之道**



客户端

PNG

PNG

副本卷

 K



gfs2 gfs3

brick brick

GlusterFS集群

PNG PNG

gfs4 brick

gfs1 brick

条带卷 PNG

副本卷



PNG

PNG

图6-38 条带卷与副本卷的堆叠

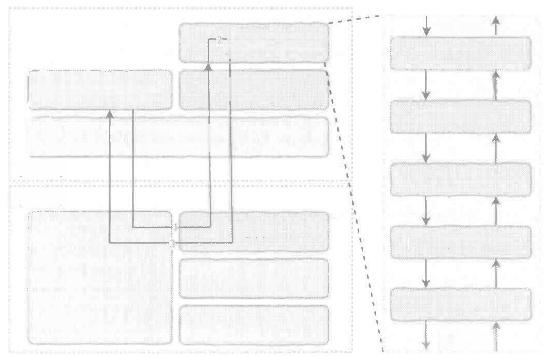
本节主要从系统的可靠性和性能方面对 GlusterFS 的特性进行了介绍。其实 GlusterFS 还有非常多的特性，如客户端的缓存、缓存预读和压缩等，这些特性都是 通过转换器的方式实现的。由于内容众多，很难逐一介绍，大家可以自行阅读xlators 目录下的相关代码。

**6.7.5** **GlusterFS客户端架构与1/O** **流程**

GlusterFS 是一个客户端的分布式文件系统，其功能特性大多在客户端实现。 这些特性包括I/O 缓存、预读、数据副本、数据分片和数据条带化等。

GlusterFS 的客户端文件系统没有在内核态实现，而是借助Fuse 在用户态实现。 Fuse 是 Linux 中的 一 个框架，通过该框架隐藏了Linux 内 核VFS 复杂的架构，提 供了在用户态实现的文件系统接口。图6- 39所示为基于Fuse 的 GlusterFS 客 户 端 的架构。

第6章提供横向扩展的分布式文件系统



Taluster客户端

Is-I/tmp/gluster libfuse

system ca

用户空间

内核空间

Fuse

VFS Ext4



libfuse X-fuse

X-lo-stats

X-DHT

X-replication

X-client

网络API

图6-39 基于Fuse的GlusterFS客户端的架构

在图6-39中，GlusterFS 客户端的实现在gluster 客户端模块中。图中的右侧是 对客户端的细节描述，其特性通过xlator 堆叠的方式实现。这里是以前文介绍的卷 配置为例构建的，包括X-fuse(fuse Xlator) 、X-io-stats 、X-DHT和 X-client 等。其 中 ，X-fuse 是基于libfuse 实现的用户态文件系统，而X-client 则是与服务端通信的 网络客户端模块。其他模块都是实现的文件系统的具体特性，如X\_DHT 是分布卷 的实现，而X\_replication 则是副本卷的实现。

GlusterFS 并没有特别复杂的架构，其I/O 栈主要是由多个xlator 堆叠起来的一 个栈。上层xlator 对下层xlator 的调用方式已经在6.7.3节中做过介绍，本节不再 赘述。本节主要以创建文件为例介绍一下处理流程。

客户端会调用Fuse 的 API 将文件处理的API 注册给Fuse。在 GlusterFS 中 ， 注册的API 是一个名为fuse\_std\_ops 的全局变量，如代码6-11所示。当客户端挂载 文件系统后，挂载点中的访问会触发这里注册的某一个函数指针。

**代码6-** **11** **fuse\_** **std\_ops全局变量的定义**

**xlator/mount/fuse/src/fuse-bridge.c**

|  |  |
| --- | --- |
| 6544 | static fuse\_handler\_t\*fuse\_std\_ops[FUSE\_OP\_HIGH]= |
| 6545 | [FUSE\_LOOKUP]=fuse\_lookup. |
| 6546 | [FUSE\_FORGET]=fuse\_forget, |
| 6547 | [FUSE\_GETATTR]=fuse\_getattr, |
| 6548 | [FUSE\_SETATTR]=fuse\_setattr, |
| 6549 | [FUSE\_READLINK]=fuse\_readlink, |
| 6550 | [FUSE\_SYMLINK]=fuse\_symlink, |
| 6551 | [FUSE\_MKNOD]=fuse\_mknod, |
| 6552 | [FUSE\_MKDIR]=fuse\_mkdir, |
| 6553 | [FUSE\_UNLINK]=fuse\_unlink, |
| 6554 | [FUSE\_RMDIR]=fuse rmdir, |

**文件系统技术内幕**

大数据时代海量数据存储之道

[FUSE\_RENAME]=fuse\_rename,

6556 [FUSE\_LINK]=fuse\_link,

6557 [FUSE\_OPEN]=fuse\_open,

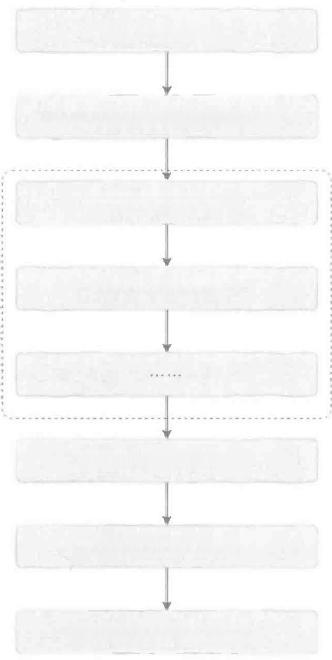
6558 [FUSE\_READ]=fuse\_readv,

[FUSE\_WRITE]=fuse\_write,

以创建文件为例，当用户在挂载的文件系统创建新文件时，会触发上述接口中

的 fuse\_create() 函 数 ， 而 该 函 数 最 终 会 调 用fuse\_create\_resume() 函 数 ， 如 图 6 - 4 0 所

示 。fuse\_create\_resume(函数通过宏定义 FUSE\_FOP 来调用STACK\_WIND 宏定 义 ， 也 就 会 调 用 下 层 的xlator。



fuse\_create()

fuse\_resolve\_and\_resume()

fuse\_resolve\_all)

fuse\_resolve()

fuse\_resolve\_done()

fuse\_fop\_resume()

fuse\_create\_resume()

图6-40 GlusterFS 客户端代码流程

当然，对于客户端来说，最终会调用client xlator,该 xlator 会将消息发送到集 群端的server xlator。

**6.7.6** **GlusterFS服务端架构与I/O流程**

服务端与客户端类似 ， 也没有复杂的软件架构 ， 其核心是服务器启动时会根据

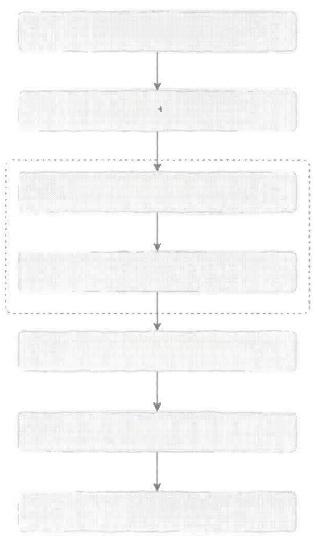
**第6章** 提供横向扩展的分布式文件系统



存储块的配置构建一个xlator 树，其中server xlator是入口层的xlator, 负责与客户 端通信。其实server 就是向RPC 注册了一个函数指针集，该函数指针集为全局变 量glusterfs3 3 fop\_actors,大家可以自行看一下这个变量。

对于创建文件操作来说，当server 收到客户端发来的创建文件的消息后会调用 server3 3 create() 函数，而该函数经过层层调用后最终会调用server\_create\_resume()

函数，如图6-41所示。server\_create\_resume(函数会调用宏定义STACK\_WIND, 这也就启动了对下一级xlator 的调用过程。

server3 3 create()

resolve\_and\_resume()

server\_resolve\_all()



server\_resolve\_done()

server\_create\_resume()

STACK\_WIND

图6-41 GlusterFS服务端代码流程

在6.7.3节介绍过一个存储块的配置文件，以该配置为例，最后会调用poxis xlator。该 xlator 的主要作用是对文件系统进行访问，其本质是调用与文件系统相关 的 API 。对于本实例中的创建文件操作，在posix 中调用了posix\_create (函数，而 该函数最终会调用open() 函数或openat(函数来创建文件。完成文件创建后，通过 反向调用，最终反馈给客户端结果。

至此，我们完成了对CephFS 和 GlusterFS 等分布式文件系统相关内容的介绍。 相信大家通过对上述两种不同类型分布式文件系统原理和代码的学习能够对分布 式文件系统有一个比较清晰的理解。

第 章

**百花争艳——文件系统的其他形态**

通过前文的介绍，我们知道文件系统包括客户端(主机端)的文件系统和服务 端软件服务。本章分别介绍一下文件系统在这两方面的演化。

**7.1** **用** **户** **态** **文** **件** **系** **统** **框** **架**

操作系统为了实现多种文件系统的支持，通常会实现一个虚拟文件系统 (VFS), 然后具体的文件系统基于VFS 框架实现。但是，无论是Linux还是Windows, 文件系统都必须在内核态实现，这样实现门槛就比较高了。

为了降低文件系统开发的门槛，有些人开发了在用户态实现的文件系统框架。 这些框架通常通过钩子的方式捕获用户对文件系统的访问，然后转发到用户态进一 步处理。这样，原来在内核态的文件系统逻辑也就可以在用户态实现，普通开发者 就可以在用户态开发一个自己的文件系统。

本节介绍一下常用的用户态文件系统框架，包括 Linux 中 的 fuse 框架和 Windows中 的Dokany 框架。

**7.1.1 Linux中的用户态文件系统框架** **Fuse**

[**7.1.1.1**](7.1.1.1) **Fuse整体架构简介**

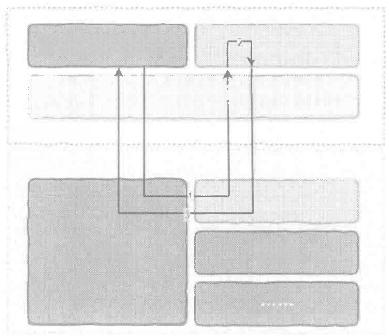
在Linux 中有一个非常有名的用户态文件系统框架，这就是Fuse 框架。有了 Fuse框架，我们就可以在用户态开发文件系统的逻辑，而不用关心Linux内核的相

第 7 章 百 花 争 艳 — — 文 件 系 统 的 其 他 形 态

关内容。从而大大降低了开发文件系统的门槛。

Fuse与 VFS 及其他文件系统的关系如图7-1所示。Fuse 本身包含一个用户态 库和一个内核态模块。用户态库为文件系统开发提供了一套接口，内核态模块则实 现了一套内核的文件系统，其功能是将文件系统访问请求转发到用户态。

用户态库提供了一套API, 同时还提供了一套接口规范，这套规范实际上是一 组函数集合。基于Fuse 开发文件系统就是实现Fuse 定义的函数集合的某些或全部 函数。然后调用Fuse 用户态库的API 将实现的函数注册到内核态模块中。

ceph\_fuse

s4/mplceph\_tost

system ca

User Space

**Kernel Space**

Fuse

VFS Ext4

图7- 1 Fuse 与 VFS 及其他文件系统的关系

内核态的模块基于VFS 实现了一个文件系统，可以与NFS 或 CephFS 客户端 的文件系统对比理解。但不同的是，当用户请求到达该文件系统时，该文件系统不 是访问磁盘或通过网络发送请求，而是调用用户态注册的回调函数。

如果大家对NFS 的流程或CephFS 的流程熟悉，则比较容易理解Fuse 的工作 原理。与NFS 类相比，Fuse 将 NFS中通过网络转化请求换成通过函数调用(严格 来说并非简单的函数调用，因为涉及内核态到用户态的转换)来转化请求。

[**7.1.1.2**](7.1.1.2) **具体实现代码解析**

接下来从代码实现层面分析一下Fuse 的实现细节。我们先从开发者角度来看 一下如何基于Fuse 进行开发。

基于Fuse 开发文件系统入门并不复杂，官方也提供了非常多的实例。以其最 简单的HelloWorld 为例，其实现代码不到200行。我们截取其中关键的代码进行 分析，如代码7-1所示。

**文件系统技术内幕**

大数据时代海量数据存储之道

代码7-1 Fuse实例代码

**example/hello.c**

|  |  |
| --- | --- |
| 55 | static void \*hello\_init(struct fuse\_conn\_info \*conn, |
| 56 | struct fuse\_config \*cfg) |
| 57 | { |
| 58 | (void)conn; |
| 59 | cfg->kernel\_cache=1; |
| 60 | return NULL; |
| 61 |  |
| 62 | /获取文件的属性 |
| 63 | static inthello\_getattr(const char\*path,struct stat \*stbuf, |
| 64 | struct fuse\_file\_info \*fi) |
| 65 | { |
| 66 | (void)fi; |
| 67 | int res=0; |
| 68 |  |
| 69 | memset(stbuf,0,sizeof(struct stat)); |
| 70 | if(strcmp(path,"/")=0){ |
| 71 | stbuf->st\_mode=S\_IFDIR|0755; |
| 72 | stbuf->st\_nlink=2; |
| 73 | else if(strcmp(path+1,options.filename)=0){ |
| 74 | stbuf->st\_mode=S\_IFREG|0444; |
| 75 | stbuf->st\_nlink=1; |
| 76 | stbuf->st\_size=strlen(options.contents); |
| 77 | }else |
| 78 | res=-ENOENT; |
| 79 |  |
| 80 | returm res; |
| 81 | } |
| 82 | //读取目录中的数据 |
| 83 | static int hello\_readdir(const char \*path,void \*buf,fuse\_fill\_dir\_t filler, |
| 84 | off\_t offset,struct fuse\_file\_info \*fi, |
| 85 | enum fuse\_readdir\_flags flags) |
| 86 | { |
| 87 | (void)offset; |
| 88 | (void)fi; |
| 89 | (void)flags; |
| 90 |  |
| 91 | if(strcmp(path,"/")!=0) |
| 92 | return-ENOENT; |
| 93 |  |
| 94 | filler(buf,"",NULL,0,0); |
| 95 | filler(buf,"..",NULL,0,0); |
| 96 | filler(buf,options.filename,NULL,0,0); |
| 97 |  |
| 98 | return 0; |
| 99 |  |
| 100 | //打开文件 |
| 101 | static int hello\_open(const char\*path,struct fuse file info\*fi) |

第7章百花争艳—— 文件系统的其他形态

102

103 if(strcmp(path+1,options.filename)!=0)

104 return-ENOENT; 105

106 if((fi->flags&O\_ACCMODE)!=0\_RDONLY)

107 retun -EACCES; 108



111 //从文件读数据的实现可以看出，这里返回的是预存的字符串

112 static inthello\_read(const char\*path,char \*buf,size\_t size,off t offset,

113 struct fuse\_file\_info\*fi)



size\_t len; (void)fi;

117 if(stremp(path+1,options.filename)I=0) return -ENOENT;

120 en=strlen(options.contents);

121 if(offset<len){

122 if(offset+size>len) size=len-offset;

124 memcpy(buf,options.contents+offset,size); 125 }else

126 size=0; 127

128 returnsize; 129 

八这个全局变量是实现的函数集，这里只实现了几个简单的函数集，如初始化、遍历目录、打开文件和 读文件等

131 static structfuse\_operationshello\_oper={ init =hello\_init,

|  |  |  |
| --- | --- | --- |
| 133 | .getattr | =hello\_getattr |
| 134 |  | =hello\_readdir, |
| 135 |  | =hello\_open, |
| 136 | read | =hello\_read, |

137

138

static void show\_help(const char \*progname)



141 printf("usage:%s [options]<mountpoint>\n\n",progname);

printf("File-system specificoptions:\n"

143

--name=<s>



Name of the \"hello\"file\n" (default:\"hello\")n"

-contents=<s> Contents \"hello\"file\n"

146  (default\"Hello,World!\n\")n"

"\n");



**文件系统技术内幕**

大数据时代海量数据存储之道

int main(int argc,char \*argv[J)

151 {

152 intret;

153 struct fuse\_args args=FUSE\_ARGS\_INIT(argc,argv) 154

155

157

options.filename=strdup("hello");

158

options.contents=strdup("Hello World!\n");

161

f(fuse\_opt \_parse(&args,&options,option\_spec,NULL)=-1)

return 1;

163

164

165

166

167

169

if(options.show\_help){

170

171

172

173

show\_help(argv[0J);

assert(fuse\_opt\_add\_arg(&args,"--help")=0); args.argv[0]=(char\*)"";

174

ret=fuse\_main(args.argc,args.argv,&hello\_oper,NULL);//Fuse fuse\_opt\_free\_args(&args);

的入口函数

176

177

return ret;

179

上述代码的主要工作是实现fuse\_operations 定义的函数集(第131行),并且 调用fuse\_main() 函数(第176行)将函数集注册到Fuse 中。而在本实例中，函数 集实现得很简单，只实现了遍历目录、打开文件、读文件和获取属性等接口。同时， 本实例通过固定数据模拟了一个文件系统，这里所有的数据只不过是使用一个全局 变量存储的字符串。

可以看出，这里只调用了Fuse 的一个函数，也就是fuse\_main() 函数。接下来 深入讲解Fuse 的内部，看一看Fuse 是如何工作的。

先看一下 fuse\_main() 函数的核心逻辑。我们通过阅读整个调用栈的代码可以发 现，关键业务逻辑是在fuse\_session\_process\_buf\_int() 函数中实现的。由于篇幅有限， 我们截取其中关键代码，如代码7-2所示。

第7章百花争艳——文件系统的其他形态

代码7-2 fuse\_session\_process\_buf\_ int()函数

lib/fuse\_lowleveLc fuse\_main->fuse\_main\_real->fuse\_loop->fuse\_session\_loop

**->fuse\_session\_process\_buf\_int**

|  |  |
| --- | --- |
| 2432  2433  2434  2529  2530  2543  2544  2545  2546  2547  2548  2551  2552  2553  2554  2555  2556  2557 | void fuse\_session\_process\_buf\_int(struct fuse\_session \*se,  conststruct fuse\_buf\*buf,struct fuse\_chan \*ch)  if(buf->flags&FUSE\_BUF\_IS\_FD)&&write\_header\_size<buf->size&& (in->opcode!=FUSE\_WRITE||!se->op.write\_buf)&&  in->opcode!=FUSE\_NOTIFY\_REPLY){ void \*newmbuf;  res=fuse\_ll copy from\_pipe(&tmpbuf,&bufv);/从内核中读取请求数据  erT=-res; if(res<0)  goto reply\_err; in=mbuf;  inarg=(void\*)&in[1];  if(in->opcode=FUSE\_WRITE&&se->op.write\_buf)//判断是否为写请求 do\_write\_buf(req,in->nodeid,inarg,buf);  else if (in->opcode=FUSE\_NOTIFY\_REPLY)  do\_notify\_reply(req,in->nodeid,inarg,buf);  else  fuse I ops[in->opcode].func(req,in>nodeid,inarg);//其他请求类型，通过函数指针进行处理 |

在上述代码中，首先从内核中读取请求数据(第2543行),然后将读取的数据 转化为fuse\_in\_header 结构体类型。这个结构体又被称为请求头，里面包括操作码、 节点ID、用 户ID 和进程ID 等信息。

接下来针对请求头的内容进行处理，主要实现代码为第2551行～第2557行。 这里主要通过操作码找到fuse l ops 中预定义的函数，然后进行后续处理。每一个 处理函数都会调用在一开始实现并注册的函数。

我们再回过头看一看请求头的数据结构。通过代码7-3可以看出，比较关键的 是第2个成员opcode, 它表示请求的类型。正是通过该操作码来确定具体由哪个函 数来进行下一步的处理。

**代码7-3** **Fuse请求头的数据结构**

**include/kernel.h**

|  |  |
| --- | --- |
| 690 | struct fuse\_in\_header{ |
| 691 | uint32\_t len; |
| 692 | uint32\_t opcode;//操作码表示请求的类型 |
| 693 | uint64\_t unique; |

**文件系统技术内幕**

大数据时代海量数据存储之道

|  |  |  |
| --- | --- | --- |
| 694 | uint64\_t | nodeid; |
| 695 | uint32\_t | uid; |
| 696 | uint32\_t | gid; |
| 697 | uint32\_t | pid; |
| 698 | uint32\_t | padding; |
| 699 |  |  |

通过上文介绍我们基本上清楚了用户态如何从内核态获取请求，并进行相关处 理。接下来介绍一下内核态的实现，看一看内核态是如何捕获用户对文件的操作的， 并将请求转发到用户态的Fuse 模块。

通过前文可知，Fuse 的内核态模块其实就是一个客户端文件系统，因此该模块 主要是实现VFS 定义的函数集。以文件相关的函数集为例，具体定义如代码7-4 所示。

**代** **码** **7** **-** **4** **内** **核** **态** **模** **块** **F** **u** **s** **e** **函** **数** **指** **针** **集**

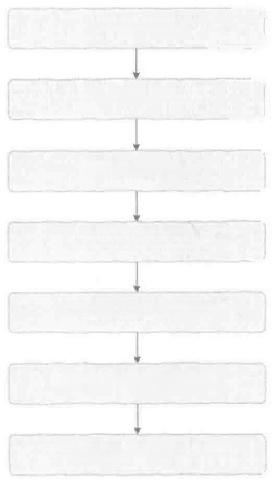
**fs/fuse/file.c**

|  |  |
| --- | --- |
| 3398 | static const structfile\_operations fuse\_file\_operations ={ |
| 3399 | llseek =fuse\_file\_llseek, |
| 3400 | read\_iter =fuse\_file\_read\_iter, |
| 3401 | write\_iter =fuse\_file\_write\_iter, |
| 3402 | mmap =fuse\_file\_mmap, |
| 3403 | open =fuse\_open, |
| 3404 | flush =fuse\_flush, |
| 3405 | release =fuse\_release, |
| 3406 | fsync =fuse\_fsync, |
| 3407 | lock =fuse\_file\_lock, |
| 3408 | flock =fuse\_file\_flock, |
| 3409 | splice\_read =generic\_file\_splice\_read, |
| 3410 | splice\_write =iter\_file\_splice\_write, |
| 3411 | unlocked\_ioctl =fuse\_file\_ioctl, |
| 3412 | compat\_ioctl =fuse\_file\_compat\_ioctl, |
| 3413 | poll =fuse\_file\_poll, |
| 3414 | fallocate =fuse\_file\_fallocate, |
| 3415 | copy\_file\_range =fuse\_copy\_file\_range, |
| 3416 | }; |

以打开文件为例，当用户通过API 打开通过Fuse 挂载目录中的文件时，会触 发 VFS 的函数调用，进而会调用 fuse\_open() 函数。关于如何通过VFS 调 用 fuse\_open() 函数，这个逻辑与前文介绍的本地文件系统及NFS 一致，本节不再 赘述。

我们主要看一下fuse\_open (函数的处理逻辑，该函数经过层层调用，最终会调 用queue\_request\_and\_unlock()函数，如图7 - 2所示。fuse\_open()函数就是将请求挂 接到一个链表中。

第7章百花争艳 — — 文件系统的其他形态



fuse\_open()

fuse\_open\_common()

fuse\_do\_open()

fuse\_send\_open()

fuse\_simple\_request()

\_fuse\_request\_send0

queue\_request\_and\_unlock()

图7-2 Fuse打开文件主流程

那么链表的请求是如何被用户态的Fuse 模块读取的呢?

其主要原理是内核态模块注册了一个混杂设备，设备名称是Fuse。我们在用户 态挂载执行文件系统流程时会打开该设备。因此，通过该设备可以实现与内核态之 间的数据交互。

至此，我们对Fuse 的使用、原理和实现代码进行了比较详细的介绍。当然， 限于篇幅，本节以打开文件为例进行介绍，并没有介绍所有流程。不过，其他流程 都是类似的，了解了打开文件流程，再通过阅读代码可以很容易地熟悉其他流程。

**7.1.2 Windows中的用户态文件系统框架Dokany**

对于用户来说，提起Linux 就不能不提Windows, 毕 竟Windows 在服务端也 占有一席之地。Windows 也有用户态文件系统框架，如Dokany 和 WinFsp 等。本 节以Dokany 为例介绍一下Windows 中的用户态文件系统框架和部分细节。

Dokany 整体架构与Fuse 整体架构没有本质的区别，我们完全可以参考图7-1 来理解Dokany 的架构。说到Dokany 就不得不提一下Dokan, 其 实Dokany 是一些 热心的开发者对Dokan 进行了封装，增加了兼容Linux Fuse的 API, 而这个增加了 API兼 容Linux Fuse的 API 的项目就是Dokany。

图7-3所示为从github 下载的Dokany 项目的目录结构。这里面核心部分包含

文件系统技术内幕

大数据时代海量数据存储之道

两部分，一个是Windows 内核态模块，另一个是用户态的动态库。另外，实现与 Fuse兼容接口的代码在目录dokan\_fuse中 。

cert

由 documentations

 *dokan* *→* *用户态模块*

田 dokany.si4project dokan\_control

dokan\_np 由 dokan\_vsix 由 dokan\_wix

渔工dokan fuse -Fuse 抽象层

samples → 实例程序

申用

scripts

sys 内核态模块 图7-3 Dokany 项目的目录结构

对于用户态的模块，其实现方式与Fuse 的实现方式非常类似，逻辑也比较简 单，本节不再赘述。其核心逻辑在DokanLoopO函数中实现，该函数会从内核获取 请求，并且根据请求的类型分发到注册的函数。

对于内核态的模块，如果想要彻底理解，则需要掌握一些Windows 内核的知

识。如果想要深入学习Windows 内核编程的知识，则可以见参考文献中的资料 [25J[26]。当然，如果没有Windows内核的知识也没关系，也不会对理解内核态模块 的逻辑产生太大的影响 。

Windows 内核态模块的IO 栈是基于分层架构的，Dokany 实现的内核态模块

会插入某两层之间。Windows 内核驱动有一个统一的入口，当模块初始化时会调用 该函数，类似用户态的main) 函数。对于Dokany 来说，其函数的实现如代码7-5 所示 ， 可以看出这里主要注册了一些回调函数 。

**代码7** **-** **5** **Do** **k** **any函数的实现**

**sys/dokan.c**

NTSTATUS

DriverEntry(in PDRIVER\_OBJECT DriverObject,\_inPUNICODE\_STRING RegistryPath) {

//删除部分代码

DriverObject->DriverUnload=DokanUnload;

DriverObject->MajorFunction[IRP\_MJ\_CREATE]=DokanBuildRequest; DriverObject->MajorFunction[IRP\_MJ\_CLOSE]=DokanBuildRequest;

DriverObject->MajorFunction[IRP\_MJ\_CLEANUP]=DokanBuildRequest;

DriverObject->MajorFuncion[IRP\_MJ\_DEVICE\_CONTROL]=DokanBuildRequest;

DriverObject->MajorFunction[IRP\_MJ\_FILE\_SYSTEM\_CONTROL]=DokanBuildRequest; DriverObject->MajorFunction[IRP\_MJ\_DIRECTORY\_CONTROL]=DokanBuildRequest;

第7章百花争艳—— 文件系统的其他形态

DriverObject->MajorFunction[IRP\_MJ\_QUERY\_INFORMATION]=DokanBuildRequest; DriverObject->MajorFunction[IRP\_MJ\_SET\_INFORMATION]=DokanBuildRequest;

DriverObject->MajorFunction[IRP\_MJ\_QUERY\_VOLUME\_INFORMATION]= DokanBuildRequest;

DriverObject->MajorFunction[IRP\_MJ\_SET\_VOLUME\_INFORMATION]= DokanBuildRequest

DriverObject->MajorFunction[IRP\_MJ\_READ]=DokanBuildRequest; DriverObject->MajorFunction[IRP\_MJ\_WRITE]=DokanBuildRequest;

DriverObject->MajorFunction[IRP\_MJ\_FLUSH\_BUFFERS]=DokanBuildRequest;

DriverObject->MajorFunction[IRP\_MJ\_SHUTDOWN]=DokanBuildRequest; DriverObject->MajorFunction[IRP\_MJPNP]=DokanBuildRequest;

DriverObject->MajorFunction[IRP\_MJ\_LOCK\_CONTROL]=DokanBuildRequest;

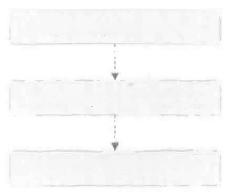
DriverObject->MajorFunction[IRPMJ\_QUERY\_SECURITY]=DokanBuildRequest; DriverObject->MajorFunction[IRP\_MJ\_SET\_SECURITY]=DokanBuildRequest;

RtIZeroMemory(&FastIoDispatch,sizeof(FAST\_IO\_DISPATCH);



这里只实现了一个回调函数，也就是DokanBuildRequest(), 该函数没做具体的 工作，主要调用DokanDispatchRequest() 函数。而DokanDispatchRequest() 函数在内 部实现了对不同请求的分发。

以创建文件为例，当请求到达 DokanDispatchRequest()函数时，会被分发到 DokanDispatchCreate() 函数来进行处理，而该函数通过如图7-4所示的流程将请求 插入一个链表中。

DokanDispatchCreate()

DokanRegisterPendingIrp(0

RegisterPendingIrpMain()

图7-4 Dokan创建文件流程

本节主要介绍一下Dokany 的架构和创建文件的流程，很多细节没有进行介绍。 如果大家想了解Dokany 更多的细节，则可以自行阅读源代码，本节不再赘述。

**文件系统技术内幕**

大数据时代海量数据存储之道

**7.2** **对象存储与常见实现简析**

接下来介绍一下文件系统在服务端的演化。服务端的演化主要是根据应用对存 储系统需求转变的。有些应用对文件系统的某些方面有特殊的需求，对另外一些方 面则没有任何需求。比如，电商应用，其图片需要存储在文件系统上，它关注的是 存储容量、扩展性和性能。但是对是否可以对文件加锁、文件访问权限、扩展属性 和层级结构等没有太多要求。

鉴于上述应用的特殊化需求，对象存储出现了。对象存储与分布式文件系统 类似，但是其对文件组织和访问语义进行了简化，使得客户端对数据访问更加 高效。

**7.2.1** **从文件系统到对象存储**

随着互联网应用的发展，互联网业务对文件系统提出了特殊的需求。于是演化 出了管理文件的一种类似文件系统，但又不是文件系统，这就是对象存储系统。本 节将介绍一下文件系统与对象存储的渊源，以及常见对象存储的相关内容。

**7** **.2** **.** **1** **.** **1从网络文件系统说起**

早些时候的企业级架构普遍采用网络文件系统，如Sun 公司的NFS 和微软 CIFS 等。关于网络文件系统的原理本书在前面章节已经进行了深入的介绍，这里 不再赘述。但是网络文件系统有以下几个缺点。

(1)客户端与存储端交互太多，特别是存在多级目录的情况下。

(2)一次数据访问需要多次访问磁盘。

(3)存储端无法通过横向扩展的方式来提升性能和容量。

虽然分布式文件系统解决了横向扩展的问题，但是由于文件系统层级结构的存 在，在主机访问存储端文件时仍然存在多次与存储端交互的问题。而且以文件系统 的方式，应用访问数据的整个访问路径是比较长的。以Web 应用为例，存储系统 的访问一般只能通过挂载到Web 主机的方式访问，无法让用户直接访问。

由于文件系统空间组织的特点，对文件访问时需要比较多次的磁盘访问。以 Ext4 文件系统为例，文件系统将磁盘空间分为两个主要的区域： 一个是元数据区， 用于存储文件的inode 等信息；另一个是数据区，用于存储文件的数据，也就是用 户数据。

这样，当访问一个文件时，首先需要找到文件对应的inode, 然后根据inode 信

第7章百花争艳 — — 文件系统的其他形态

息找到数据的位置，并读取数据。整个过程可能要涉及2~3次的磁盘访问。对于 互联网应用来说，多次磁盘访问会显著降低性能，影响用户的体验。

[**7.2.1.2**](7.2.1.2) **对象存储解决的问题**

由于上述缺点，传统的网络文件系统很难完全满足互联网领域的应用需求。我 们列举一个实例，以Facebook 为例，其每秒钟都有几十万次的照片检索请求。其 存储的照片总量每天新增3.5亿张，对应的存储增量约为300TB。如果对应物理设 备，则每天大概需要新增上百块硬盘。

许多大型互联网公司都会遇到这种问题。比如，今日头条、淘宝或京东等，在 它们的平台上每天也要产生海量的图片资源，并且访问量也是惊人的。这种场景， 传统存储很难满足其性能和扩展性的要求。

虽然互联网应用对性能和容量的要求极高，但是对其他特性的要求并不高，甚 至可以说基本上没有要求，如对文件内容的修改和文件锁等。由于上述场景存储的 主要是图片，而且存储特点是一次存储、多次访问、没有修改、很少删除。所以， 文件系统的很多特性都是可以简化的。

针对上述特点，为了解决性能和容量的问题，对象存储应运而生。可以看出对 象存储要解决的问题很集中，就是保证横向扩展能力、降低访问延时。而不需要实 现文件系统的其他额外特性。

对象存储在数据处理层面的特点是将待处理的数据看作一个整体，无法进行局 部修改，这也就是为什么把它称为对象，而不是文件了。目前，大多数对象存储只 能创建、删除和读取对象，而不支持修改对象。同时，如果有多个客户端同时创建 同一个对象，则存储系统不会保护这些对象，对象的数据以后传输的为准。

但对象存储整体上也不是那么非常简单，很多对象存储在其他方面实现了比较 丰富的特性。比如，S3 对象存储可以支持大数据处理、扩展属性和二次处理(如照 片的转换，水印)等特性。接下来介绍对象存储着重解决的问题。

**1.客户端与存储端交互次数多的问题**

客户端与存储端的交互次数太多是由协议造成的。以NFSv3 协议为例，如果 客户端要读取某个目录下的文件，在打开文件时需要确定父目录和每个祖先目录的 存在性。在这种情况下就需要多次向存储系统发送GETATTR 命令。

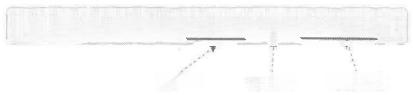
对象存储主要通过两种方法解决该问题： 一种是数据采用扁平化的方式管理； 另一种是采用了新的访问协议。

文件系统技术内幕

大数据时代海量数据存储之道

在对象存储中，所有数据是存储在一个或多个类似文件系统目录的容器中的， 这个容器在S3 中称为桶(bucket), 在 Swift 中称为容器(container) 。 但与文件系 统中目录不同的地方是容器是不可以嵌套的，也就是不能在容器中创建子容器。

对象存储的访问通常采用基于HTTP 协议的RESTFul 风格的API 来直接访问， 通过 一个URL 就可以直接定位到具体的对象，如在Swift 中访问对象的格式如 图7-5所示。



hp /sumyzhang comv/123456789/mages test imagcjpg

租 户ID 桶/容器 对象名称

图7-5 对象存储访问URL格式

与文件系统相比，通过这种方式将对存储的多次访问减少为一次。另外，由于 基于HTTP 协议，客户端也可以直接访问对象存储中的对象。显然，通过这种方式 不仅可以减少客户端与存储端的交互次数，甚至可以将很多服务器的负载转移到对 象存储系统。

**2** **.** **多次访问磁盘问题**

元数据的访问在海量小文件的场景下，性能影响最为显著[27],因此，如果能够 减少元数据操作，那么可以极大地提升存储系统的性能。目前，有很多存储系统对 此进行了优化，如Haystack 等存储系统。普遍的做法是存储端不采用本地文件系 统，或者将多个小文件聚合为一个大文件，并将元数据全部缓存到内存中。

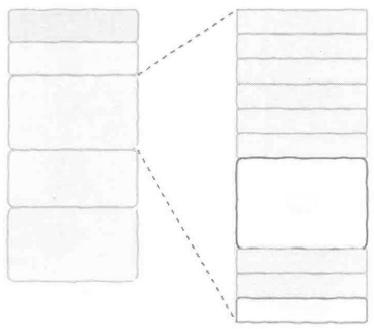
以Haystack 为例，其核心特点就是存储小文件，如照片。因为照片的大小通常 在10MB 以下，大部分在KB 级别，如果按照常规每张照片以一个文件的方式存储 将会产生大量的元数据。下面看一下Haystack 是如何解决磁盘访问问题的。

Haystack的做法非常简单，它将多个小文件作为一个大文件的局部数据，这个 局部数据称为 needle。同 时 ，Haystack 构建了一个描述 needle 在大文件中位置的索 引文件。由于索引文件比较小，因此可以一次性加载到内存中。

关 于needle 在大文件中的布局如图7-6所示。其中，每个needle 的前半部分是 一个固定长度的描述信息，特别是里面有一个描述数据大小的域。这样，即使在没 有索引文件的情况下，我们也可以很容易地找到第1个needle,然后计算出后续所 有needle 的位置，进而重构索引文件。

**第7章** 百花争艳——文件系统的其他形态





头魔数

Cookie

Key

可变Key

Flags

大小

大小

尾魔数

数据校验和

填充区

超级块

needle 1

needle 2

needle 3

……

图7-6 Haystack 数据布局

由于索引数据是存储在内存中的，因此当客户端需要访问数据时，存储节点可 以直接从内存中得到数据的位置，并一次从磁盘上读取数据。从而使存储性能得到 大幅提升。

**3** **.横向扩展问题**

单个节点的处理能力总归是有限的。如果能够通过增加节点数量的方式实现对 存储系统的扩容(包括容量和承载能力),那么理论上存储系统的能力可以无限增 加，当然实际上会有各种局限。

对象存储中的横向扩展是基础特性。以OpenStack Swift为例，该对象存储其 实除实现对象存储的基本特性外，其最主要的就是实现了横向扩展。

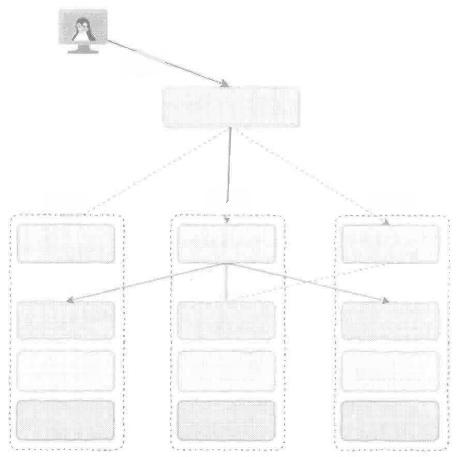
OpenStack Swift的横向扩展是通过其前端的Proxy 组件和数据放置算法实现 的 。Proxy 组件实现了数据分发的功能，所有请求都要经过Proxy 组件。在Proxy 组件内部有数据放置算法和系统拓扑描述，该组件根据上述信息可以确定对象 的存储位置。

Proxy 组件最大的特点是可以具备多个实例，每个实例可以安装在一台物理服 务器上。由于算法确定，只要每个Proxy 组件上的信息一致，那么每个Proxy 组 件 都可以对请求的数据进行定位，而且结果一致。加上Proxy 组件可以横向扩展，因 此整个系统没有任何性能瓶颈节点。

在实际部署时可以在Proxy 组件前面部署一个负载均衡器，这样来自客户端的 请求经过负载均衡器后会被均匀地分发到Proxy 节点，而Proxy 组件经过计算后将 请求发送到具体的存储节点进行处理。OpenStack Swift部署结构如图7-7所示。

文件系统技术内幕

大数据时代海量数据存储之道



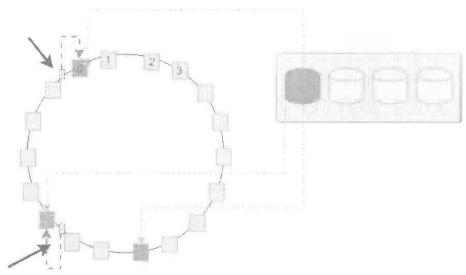
|  |  |  |  |
| --- | --- | --- | --- |
| Zone1    Proxy | PUT | 负载均衡器  zohe2  Proxy | Zone3  Proxy |
| Account |  | Account | Account |
| Container  Object |  | Container  Object | Container  Object |

图 7 - 7 OpenStack Swift部 署 结 构

在Proxy 组件中最核心的算法是进行数据放置的一致性哈希算法，该算法实现 了将一个对象映射到物理设备的过程。为了保证整个系统的可靠性和可用性， OpenStack Swift将设备划分为若干等级，如Zone 、Host 和 Disk。通过不同设备的 分发，实现故障域的隔离。

一致性哈希算法是对哈希算法的改进。哈希算法是通过对哈希表长度取模的方 式来定位的算法。当哈希表长度发生变化时整个映射关键也会发生非常大的变化。 图7-8所示为基于一致性哈希算法的数据放置，先要构建一个哈希环，哈希环 由0～32位整数(或64位最大值)构成，每个值为一个槽位。哈希环的初始化是

将设备映射到哈希环的某些槽位上。 一致性哈希算法的流程大致如下。



服务器

图 7 - 8 基 于 一 致 性 哈 希 算 法 的 数 据 放 置

**第7章** 百花争艳——文件系统的其他形态



(1)首先将物理设备映射到哈希环上，建立物理设备与哈希环槽位的映射 关系。

(2)当有对象访问时，根据对象名称计算出其哈希值。

(3)哈希值以顺时针的方式映射到哈希环具有物理设备的某个槽位上。

经过上述3个流程，对象存储利用一致性哈希算法就可以根据对象名称轻松地 找到对应的物理设备，然后与物理设备交互，完成数据的访问操作。按照上述流程， 当出现物理节点故障时只会影响落到该节点的数据，而落到其他节点的数据位置并 不会发生变化。

在物理设备数量比较少的情况下可能会出现物理设备在哈希环分布不均匀的 情况。特别是出现故障时，故障节点的数据会被前移到同一个物理节点，导致该节 点负载和数据大增。为了改善上述问题，常用的方法是通过虚节点的方式。比如， 为一个物理节点构建100个虚节点，然后将虚节点映射到哈希环。由于虚拟节点到 哈希环槽位的映射是伪随机的，因此当出现物理节点故障时，故障节点的数据会被 不同的物理节点处理，从而分摊了负载。

本节对传统文件系统的缺点进行了分析，并且结合实例对对象存储解决的主要 问题进行了简要的分析。

**7.2.2** **S3对象存储简析**

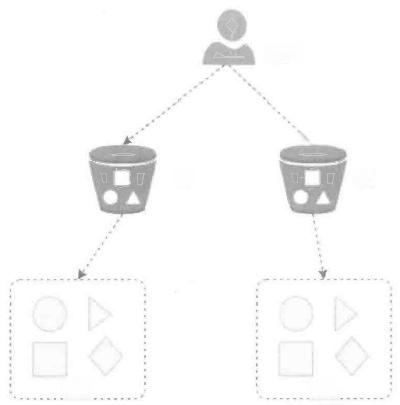
亚马逊在2006年推出了云计算领域的第 一 款产品 S3(Simple Storage Service, 对象存储服务),虽然S3 不是最早的对象存储产品，却是最出名的对 象存储产品。

S3 抛弃了文件系统的层级结构，通过扁平化的方式方便用户使用数据。在S3 中通过一种桶(bucket) 的容器实现了对数据的组织管理。桶的作用类似文件系统 的目录，但是桶只能保存对象，不能创建“子桶”。

在公有云环境中，资源都是属于某个租户的，S3 中的桶也是属于某个租户的。 于是S3 中的租户、桶与对象的关系如图7-9所示。在S3 中，默认一个租户最多可 以创建100个桶。

**文件系统技术内幕**

大数据时代海量数据存储之道

租户



桶

桶

对象 对象

图7-9 S3 中的租户、桶与对象的关系

对象是存储在桶中的，其存储方式是Key-Value, 也就是一个对象名称对应着 对象的数据。在桶中对象的数量是不受限制的，但对象的大小是受限制的，最大可 以创建5TB 的对象。

S3 并不局限在存储数据对象，它现在的功能非常丰富，特别是支持许多附加 特性。比如，数据的智能分层、桶数据的跨区复制、访问控制和数据传输等特性。 随着云计算的兴起，S3 无疑已经成为对象存储的实时性标准，大多数对象存储实 现都兼容S3 接 口 。

**7.2.3** **Haystack对象存储简析**

Haystack 是 Facebook 用来存储照片的存储系统。在其论文发表时，声称该系 统已经存储了大约2600亿张照片，容量多达20PB[28]。

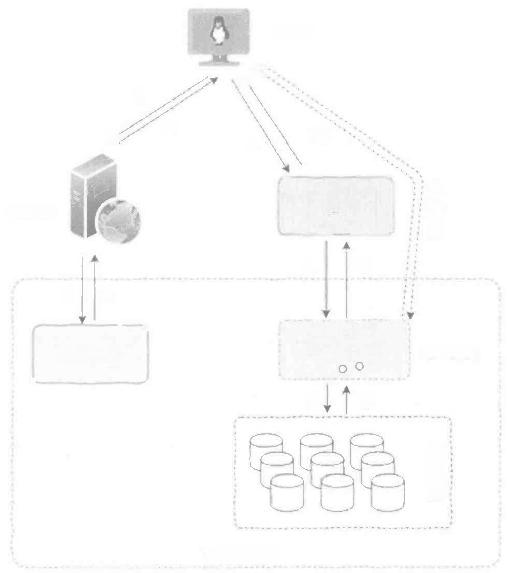
Facebook 早期也是用NAS 设备来存储系统的，但是随着数据规模的增大，传 统 NAS 存储已经完全不能胜任其照片业务的需求。随后，Facebook 的工程师分析 其业务特点，开发了适合自身的存储系统，也就是Haystack。

Facebook 开 发 的Haystack 存储系统就是用来解决上述问题的。照片服务有一 个非常明显的特点就是一次写入、多次读取并且很少删除。对于修改方面，Haystack 先假设很少修改，即使出现修改的情况，在Haystack 内部也是将原来的照片标记 为删除，而将新的照片追加到最后。

在了解了上述假设之后，我们看一下Haystack 的整体架构，如图7-10所示。 其 中 ，Haystack 集群组件在灰色方框中，包含目录服务、缓存服务和存储服务3个

第7章百花争艳——文件系统的其他形态

组 件 。Web 服务和CDN 不属于Haystack,CDN 是一个外部服务，Web 服务提供 Web访问，可以将Web 服务理解为Haystack 服务的使用者。

浏览器



1

4

10

5

Web 服务

CDN

6

9

2 3

o

0

O

**目录服务**

0

o

O

o

7

8

直接访问

缓存服务

存储服务

HayStack

图7-10 Haystack的整体架构

在图7-10中给出了一个访问的详细流程。当浏览器访问一个页面时，Web 服 务会通过目录服务为每个图片构建一个URL ( 步 骤 1 ~ 4 ) 。URL 稍微有点复杂， 一个访问CDN 的 URL 格式如下：

http:/<CDN>/<Cache>/<Machine id>/<Logical volume,Photo>

从URL 格式可以看出，该路径包含CDN 地址信息、缓存信息、存储节点信息 和逻辑卷及图片ID 信息等内容。

有了URL 之后，浏览器就可以加载图片了。请求会发送到CDN 服 务 ，CDN 会对图片进行查找。如果CDN 没有该图片，CDN 服务会向URL 中的缓存服务发 送获取图片的请求。如果缓存服务依然没有该图片，缓存服务会根据URL 中 的 Machine id信息将请求转发到具体的存储节点。最后存储节点根据逻辑卷信息和图 片ID 信息找到该图片。

目录服务相当于我们在前面分布式文件系统架构中介绍的控制节点，在目录服 务中维护着整个对象存储系统的元数据信息，其主要作用如下。

**文件系统技术内幕**

大数据时代海量数据存储之道

(1)提供逻辑卷到物理卷的映射。

(2)提供负载均衡，可以实现写操作的跨逻辑卷负载均衡和读操作的跨物理卷 负载均衡。

(3)确定请求走CDN还是直接发送到缓存服务。

(4)识别某些逻辑卷是否为只读状态。

缓存服务用于缓存热点数据，这样请求就不用访问磁盘，而是直接从缓存读取。

存储服务是一个由普通服务器构成的集群，它们是实际存储持久化数据的节 点。存储服务中每个节点上所有硬盘通过RAID 卡 做 成RAID6, 然后将该RAID 划 分为比较大的(如100GB) 物理卷，如图7-11所示。假设有12块1TB的硬盘，做 成RAID6 后有10TB 的空间，可以划分出100个100GB 的物理卷。物理卷实际上 是本地文件系统中的一个大文件。在Haystack 的存储节点上使用的是XFS 文 件 系统。

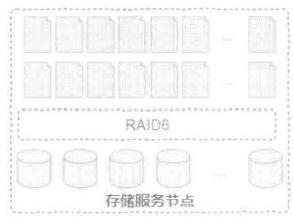


图7-11 存储服务节点示意图

[**7.2.3.1**](7.2.3.1) **数据可靠性**

Haystack 解决了两种级别的组件故障： 一种是磁盘级的故障；另一种是节点级 的故障。这里节点级故障不局限于节点宕机问题，网线或网卡等故障也有可能引起 节点级的故障。

Haystack 从两个方面实现上述故障的解决： 一个是在节点内通过RAID 技术保 证磁盘故障下的数据可靠性；另一个是通过节点之间的数据副本来保证节点级故障 下的数据可靠性。对于数据副本，Haystack 是通过逻辑卷来实现的。

RAID 技术通过某种算法将多个块设备(如磁盘)抽象为一个块设备来使用。 通过RAID 技术构建的块设备不仅可以基于冗余的方式实现容错，而且可以通过条 带化提高性能。关于RAID 的更多细节本节不再赘述，有兴趣的读者可以见参考文 献中的资料[29]。

节点之间的可靠性是通过逻辑卷实现的。逻辑卷由多个物理卷构成，物理卷中

**第7章百花争艳——文件系统的其他形态**



存储的内容相同，也就是通过多个物理卷实现多副本。对于逻辑卷中物理卷的选择 来说，可以跨节点、机柜甚至数据中心。通过对设备拓扑的识别，可以实现更高 级别的容错。

在关于Haystack 的论文中并没有对逻辑卷做太多的解释，但根据我们对分布 式文件系统相关技术的理解，可以做一些推测。在GlusterFS中，我们也看到过关 于卷的概念，知道副本卷可以实现多副本存储，并保证数据的可靠性。

在 Haystack 中，逻辑卷的作用与GlusterFS 中副本卷的作用类似。不过在 Haystack 中逻辑卷并不会呈现卷的形态，也就是数据并不经过该卷。在Haystack 中，卷只是一个逻辑概念，本质上只是目录服务中的一个项目，它建立了与物理卷 之间的关系。

为了更加清楚地理解逻辑卷与物理卷之间的关系，假设有一个3个节点的集 群，每个节点有两个物理卷。假设创建了3个逻辑卷，副本数为2,此时逻辑卷与 物理卷之间的对应关系如图7-12所示。

目录服务节点

RAID6

存储服务节点0

RAID8

存储眼务节点1 存储细务节点2

图7-12 逻辑卷与物理卷之间的对应关系

从图7-12中可以看出，每个逻辑卷对应着两个物理卷，为了更加清晰地了解 逻辑卷与物理卷的关系，本书列出表7-1所示的对应关系。

**表7-1逻辑卷与物理卷的关系**

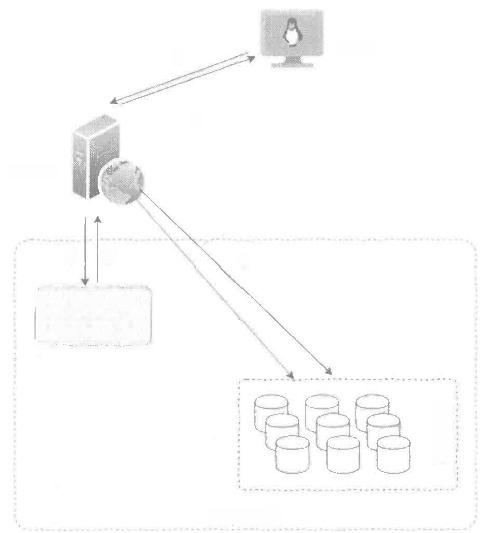
|  |  |  |
| --- | --- | --- |
| **逻** **辑** **卷** | 物 理 卷 0 | 物 理 卷 1 |
| 0 | 0-0 | 10 |
| 1 | 0-1 | 2-0 |
| 2 | 1-1 | 2-1 |

由于有上述对应关系，当用户上传图片时，Web 服务首先从目录服务查询一个 逻辑卷。然后Web 服务根据该逻辑卷对应的物理卷直接将请求发送到存储服务。

**文件系统技术内幕**

大数据时代海量数据存储之道

写数据的流程如图7-13所示。

Web 服务

1

5

阅览器



2 3 4

**目录服务**

存储服务

Haystack

图7-13 写数据的流程

[**7.2.3.2**](7.2.3.2) **横向扩展**

理解了逻辑卷的概念，以及逻辑卷与物理卷之间的对应关系，就容易理解其横 向扩展的概念。

由于存储系统的负载和容量由逻辑卷承载，因此扩容其实就是增加逻辑卷数量 的过程。又因为逻辑卷是由物理卷构成的，因此扩容其实就是增加节点，并且创建 逻辑卷的过程。当创建了新的逻辑卷后，在目录服务中就会有相关的信息。由于新 逻辑卷使用率和负载等是最小的，因此写数据会优先分配给它，这也就达到了横向 扩展的目的。

Haystack 的整体架构其实与GFS 的整体架构非常类似，这里目录服务类似GFS 中 的master 节点，而存储服务则类似chunkserver 。 不同之处是Haystack 目录服务 维护的元数据相对比较简单，没有复杂的层级关系。

[**7.2.3.3**](7.2.3.3) **访问性能**

对于访问性能方面，Haystack 在多个层面进行了设计，如CDN 的使用、缓存

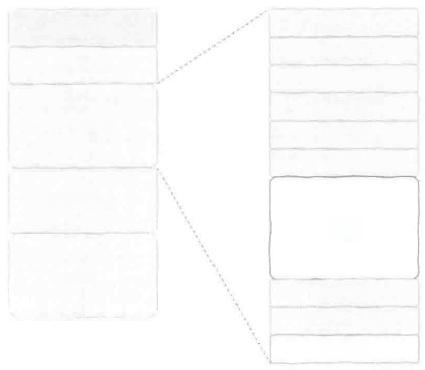
**第7章百花争艳——文件系统的其他形态**

的使用和本地数据组织等。由于CDN 和缓存是基础组件，这方面的内容有很多介 绍，本节不再赘述，具体见参考文献中的资料30131132]。

Haystack 性能的优化有两个方面： 一个方面是目录服务实现对物理卷和逻辑卷 的负载均衡；另一个方面是对小文件的聚合处理。

在负载均衡方面，目录服务实时地收集各个存储节点上物理卷的访问热度信息 和容量信息。同时，在目录服务中通过平衡搜索树或某种方式快速检索负载最小逻 辑卷。这样，当客户端有读/写请求时，目录服务就可以快速找到最优的逻辑卷。

在小文件聚合方面，Haystack 通过一个大文件(物理卷)和一个索引文件来实 现。其中，大文件用来存储多个小文件，索引文件则建立了对象关键字与数据在大 文件偏移的关系。对象数据在大文件中是依次排列的，如图7-14所示。



头魔数

Cookie

Key

可变Key

Flags 大小

数据

**尾魔数**

**数据校验和**

**填充区**

超级块 needle 1

needle 2

needle3

…

图7-14 物理卷数据布局

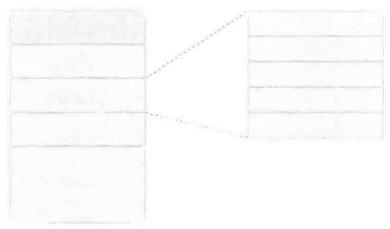
通过图7-14可以看出，在Haystack 中每个对象通过一个needle 进行表示。每 个needle的布局如图7-14右侧所示。在needle中包含关键字、大小和数据等信息， 这样通过上述信息可以找到期望的对象。

通过遍历的方式查找对象已经太落后了，也没达到使用Haystack 想要解决的 问题。在Haystack 中，可以通过索引文件来实现对象查找。索引文件可以一次性地 被加载到内存中，因此对象的查找就变成索引文件遍历的过程。

索引文件数据布局如图7-15所示，其中，每个needle 与大文件中的needle 是 一一对应的。这样，根据索引文件中Key 与偏移的关系就可以确定要查找的对象在 大文件的位置。可见，对于读取请求，可以不经过读磁盘而一次性确定对象位置， 然后直接从大文件读取数据。

**文件系统技术内幕**

大数据时代海量数据存储之道

超级块

needle 1 needle 2 needle 3

Key

可变Key

Flags

偏移

大小





图7-15 索引文件数据布局

对于索引文件在内存中的表示，我们可以通过平衡搜索树或哈希表实现，这样 在内存中的查找也是非常迅速的。

假设小文件平均大小为IMB, 那么一个物理卷大概可以存储1万个小文件。 这样，对于文件的元数据信息就少得多了。

至此，我们完成了本书所有内容的介绍。从本地文件系统到网络文件系统、分 布式文件系统、再到对象存储。每一种存储技术的出现都是为了解决某些问题，而 也都有其适用场景。通过本书的介绍，相信大家对文件系统有了一个整体和系统的 认识，也希望能够对大家的工作有所帮助。

**参考文献**

[1]Microsoft 公司.微软英汉双解计算机百科辞典[M]. 汉扬天地科技发展有限公司， 译.北京：北京希望电子出版社，1999.

[2]全国科学技术名词审定委员会审定.计算机科学技术名词(第3版)[M]. 北京：科 学出版社，2018.

[3]W.Richard Stevens.UNIX环境高级编程[M].无晋元，译.北京：机械工业出版社， 2000.

[4]ONeil,et al.The LRU-K Page Replacement Algorithm for Database Disk Buffering[J]. Acm Sigmod Record,1993.

[5]Johnson T,Shasha D.2Q:A Low Overhead High Performance Buffer Management Replacement Algorithm[C].PROCEEDINGS OF THE INTERNATIONAL CONFERENCE ON VERY LARGE DATA BASES.Morgan Kaufmann Publishers Inc. 1994.

[6]Jiang S,Zhang X.LIRS:An Efficient Low Inter-reference Recency Set Replacement to Improve Buffer Cache Performance[C].Proceedings of the International Conference on Measurements and Modeling of Computer Systems,SIGMETRICS 2002,June 15- 19,2002,Marina Del Rey,California,USA.ACM,2002.

[7]Lee D,ChoiJ,KimJH,et al.LRFU:A Spectrum of Policies that Subsumes the Least Recently Used and Least Frequently Used Policies[J].Computers IEEE Transactions on, 1999,27(1):134-143.

[8]Megiddo N.ARC:A self-tuning,low overhead Replacement cache[C]USENIX File and Storaqe Technologies Conference(FAST'03),San Francisco,CA.2003.

[9]A.Sweeney,D.Doucette,W.Hu,C.Anderson,M.Nishimoto,and G.Peck.Scalability in the xfs file system.In ATEC'96:Proceedings of the 1996 annual conference on USENIX Annual Technical Conference,pages 1-1,Berkeley,CA,USA,1996. USENIX Association.

[10]鸟哥.鸟哥的Linux 私房菜：服务器架设篇(第3版)[M]. 北京：机械工业出版 社，2012.

[11]EviNemeth.UNIX/Linux 系统管理技术手册(第4版)[M]. 张辉，译.北京：人民 邮电出版社，2016.

[12]Ghemawat S,Gobioff HB,Leung S.The Google file system[J].ACM SIGOPS Operating Systems Review,2003.

[13]Sage A.Weil,Scott A.Brandt,Ethan L.Miller,DarrellD.E.Long.Ceph:A Scalable,

**文件系统技术内幕**

大数据时代海量数据存储之道

High-Performance Distributed File System.

[14]S.A.Weil,K.T.Pollack,S.A.Brandt,and E.L.Miller.Dynamic Metadata Management for Petabyte-scale File Systems.In Proceedings of the 2004 ACM/IEEE Conference on Supercomputing(SC'04).ACM,Nov.2004.

[15]Mustaque Ahamad,Mostafa H.Ammar,Shun Yan Cheung.Replicated Data Management in Distributed Systems.

[16]黄建忠，曹强，秦啸.纠删码存储集群系统设计与优化[M]. 北京：科学出版社， 2016.

[17]Huang C,Simitci H,Xu Y,et al.Erasure coding in windows azure storage[C] Proceedings ofthe 2012 USENIX conference onAnnual Technical Conference.USENIX Association,2012.

[18]GautschiW.On inverses of Vandermonde and confluent Vandermonde matrices II[J]. Numerische Mathematik,1978,29(4):445-450.

[19]MurrayJF,Hughes GF,Kreutz-Delgado K.Machine Learning Methods for Predicting Failures in Hard Drives:A Multiple-Instance Application[J].Journal of Machine Learning Research,2005,6(1):783-816.

[20]卡伦 ·辛格.Ceph 分布式存储学习指南[M].Ceph 中国社区，译.北京：机械工业出 版社，2017.

[21]Ceph 中国社区.Ceph 分布式存储实战[M]. 北京：机械工业出版社，2016.

[22]谢型果.Ceph 设计原理与实现[M]. 北京：机械工业出版社，2017.

[23]Sage A.Weil,Andrew W.Leung,Scott A.Brandt,Carlos Maltzahn.RADOS:A Scalable,Reliable Storage Service for Petabyte-scale Storage Clusters.

[24]常涛.Ceph 源码分析[M]. 北京：机械工业出版社，2016.

[25]谭文，陈铭霖.Windows 内核编程[M]. 北京：电子工业出版社，2020.

[26]谭文，陈铭霖.Windows 内核安全与驱动开发[M]. 北京：电子工业出版社，2015.

[27]D.Roselli,J.Lorch,and T.Anderson.A comparisonof file system workloads.In Proceedings of the 2000 USENIX Annual Technical Conference,pages 41-54, June2000.

[28]Beaver,Doug,Kumar,Sanjeev,Li,Harry C,et al.Finding a needle in Haystack: facebook's photo storage[C]Usenix Conference on Operating Systems Design & Implementation.USENIX Association,2010.

[29]鲁士文.存储网络技术及应用[M]. 北京：清华大学出版社，2010.

[30]雷葆华，孙颖，王峰，等.CDN 技术详解[M]. 北京：电子工业出版社，2012.

[31]唐宏，陈戈，陈步华，等.内容分发网络原理与实践[M]. 北京：人民邮电出版社， 2018.

[32]Carlson,Josiah L.Redis in Action[J].Media.johnwiley.com.au,2013.

**博文视点** **·** **IT出版旗舰品牌**

技术凝聚实力 ·专业创新出版

文件系统是用来组织管理和存储用户数据的重要系统软件。随着用户对数据存储的可用性、可靠性和 可扩展性需求的提升，以及硬件技术和设备的不断发展，文件系统的设计和实现也在不断发展演进。 本书采用图文并茂、通俗易懂的语言对不同类型的文件系统进行了详细的介绍，是带领读者入门学习 文件系统的一本好书。

— ——Dell Technologies中端存储部门高级经理陈欢

在阅读本书过程中，读者能够体会到作者撰写本书的用心。在每一章的开头，作者都会提示前面章节 的知识，后面章节知识的衔接，引导读者阅读本书。通过学习本书的内容，希望读者了解、掌握更多 的文件系统技术内幕。

Dell Technologies中端存储部门高级经理高雷

初读本书给人一种耳目一新的感觉， 一方面本书聚焦在存储技术的文件系统领域，另一方面本书有非 常多的插图，图文并茂的方式能够使读者对文件系统原理的理解变得更加容易。相信本书对存储技术 开发人员或想要更深入学习文件系统原理的开发人员都会有很大的帮助。

阿里达摩院城市大脑实验室产品总监、资深产品专家戴文军

作为从事十多年存储研发的老将看到本书，有种相见恨晚的感觉。本书涵盖了文件系统的诸多技术领 域，而最大的特点则是在代码层面进行了解析，使得读者能够具体地理解文件系统的原理，而非停留 在理论层面。

— —同有研发管理中心总经理文中领

对于专业程序员来说，操作系统和文件系统都是必须掌握的知识。本书从文件系统的概念和使用开始， 逐渐深入到原理和代码，不但对本地文件系统进行了充分而详尽的介绍，还涉及网络文件系统和分布 式文件系统。语言翔实、通俗易懂，是程序员的案头书。

—IBM 前架构师、“码农翻身”公众号作者刘欣

文件系统在存储中举足轻重，从本地文件系统到分布式文件系统，正在向全闪存和智能存储趋势发展。 本书深入浅出，侧重实践操练和源代码剖析，对理解和学习文件系统具有很高的参考价值，特别建议 存储科研人员和行业从事者阅读。

— 大道行云CEO 刘 爱贵

在数字时代，大数据、AI、自动驾驶等半结构化、非结构化数据新应用推动着企业存储系统不断变革。 文件系统作为存储的基石，过去数十年间不断推陈出新，以承载规模更大、更快、更多样、更有价值 的数据。本书“知行合一”,兼顾理论和开发实战，为要深耕文件系统的读者点亮了理论和实践的明灯。  ~~——~~浪潮信息存储首席架构师孙斌



**读者服务**

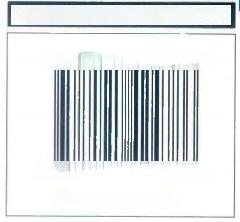
微信扫码回复：42478

·加入本书读者交流群，与作者互动

获取【百场业界大咖直播合集】(持续更新),仅需1元

责任编辑：林瑞和

封面设计：吴 海燕



**上架建议：计算机技术/数据存储**

ISEN⁹78-7-121-42478-6

978712124786

定价：102.00元

[Gener al I nf or ma t i on]

**书名=文件系统技术内幕%大数据时代海量数据储存之道** **页** **数** **=** **2** **9** **6**

SS号=15060832