Veritas™ File System 程序员参考指南

Solaris

6.0

Symantec™

Veritas File System 程序员参考指南

本手册所述软件是根据许可协议而提供，仅可按该协议的条款使用。

产品版本： 6.0

文档版本： 6.0.0

法律声明

Copyright © 2011 Symantec Corporation. © 2011 Symantec Corporation 版权所有。All rights reserved. 保留所有权利。

Symantec、Symantec 徽标、Veritas、Veritas Storage Foundation、CommandCentral、NetBackup、Enterprise Vault 和 LiveUpdate 是 Symantec Corporation 或其附属公司在美国和其他国家/地区的商标或注册商标。“Symantec”和“赛门铁克”是 Symantec Corporation 在中国的注册商标。其他名称可能为其各自所有者的商标，特此声明。

本文档中介绍的产品根据限制其使用、复制、分发和反编译/逆向工程的授权许可协议分发。未经 Symantec Corporation（赛门铁克公司）及其特许人（如果存在）事先书面授权，不得以任何方式任何形式复制本文档的任何部分。

本文档按“现状”提供，对于所有明示或暗示的条款、陈述和保证，包括任何适销性、针对特定用途的适用性或无侵害知识产权的暗示保证，均不提供任何担保，除非此类免责声明的范围在法律上视为无效。Symantec Corporation（赛门铁克公司）不对任何与提供、执行或使用本文档相关的伴随或后果性损害负责。本文档所含信息如有更改，恕不另行通知。

根据 FAR 12.212 中的定义，授权许可的软件和文档被视为“商业计算机软件”，受 FAR 第 52.227-19 节“Commercial Computer Software - Restricted Rights”（商业计算机软件受限权利）和 DFARS 第 227.7202 节“Rights in Commercial Computer Software or Commercial Computer Software Documentation”（商业计算机软件或商业计算机软件文档权利）中的适用规定，以及所有后续法规中规定的权利的制约。美国政府仅可根据本协议的条款对授权许可的软件和文档进行使用、修改、发布复制、执行、显示或披露。

Symantec Corporation
350 Ellis Street
Mountain View, CA 94043

http://www.symantec.com

技术支持

Symantec 技术支持具有全球性支持中心。技术支持的主要任务是响应有关产品特性和功能的特定查询。技术支持小组还负责编写我们的联机知识库文章。技术支持小组与 Symantec 内的其他职能部门相互协作，及时解答您的问题。例如，技术支持小组与产品工程和 Symantec 安全响应中心协作，提供警报服务和病毒定义更新服务。

Symantec 提供的维护服务包括：

- 一系列支持服务，使您能为任何规模的单位选择适用的支持服务
- 通过电话和 Web 支持快速响应并提供最新信息
- 升级保证可保证软件顺利升级
- 全天候提供全球支持
- 高级功能，包括“客户管理服务”

有关 Symantec 维护计划的更多信息，请访问我们的网站：

http://www.symantec.com/zh/cn/support/index.jsp

与技术支持联系

具有有效维护协议的客户可以通过以下网址访问技术支持信息：

http://www.symantec.com/zh/cn/support/index.jsp

在联系技术支持之前，请确保您的计算机符合产品文档中所列的系统要求。而且您应当坐在发生问题的计算机旁边，以便需要时重现问题。

联系技术支持时，请准备好以下信息：

- 产品版本信息
- 硬件信息
- 可用内存、磁盘空间和 NIC 网卡信息
- 操作系统
- 版本和补丁程序级别
- 网络结构
- 路由器、网关和 IP 地址信息
- 问题说明：
  - 错误消息和日志文件
  - 联系 Symantec 之前执行过的故障排除操作

- 最近所做的软件配置更改和网络更改

授权许可与产品注册

如果您的 Symantec 产品需要注册或许可证密钥，请访问我们的技术支持网页：

https://licensing.symantec.com/

客户服务

可从以下网站获得客户服务信息：

http://www.symantec.com/zh/cn/support/index.jsp

客户服务可帮助您解决一些非技术性问题，例如以下几类问题：

- 有关产品许可或序列号的问题
- 产品注册更新（例如，更改地址或名称）
- 一般产品信息（功能、可用的语言、当地经销商）
- 有关产品更新和升级的最新信息
- 有关升级保障和维护合同的信息
- Symantec 采购计划的相关信息
- 有关 Symantec 技术支持选项的建议
- 非技术性的售前问题
- 与光盘或手册相关的问题

维护协议资源

如果想就现有维护协议事宜联络 Symantec，请通过以下方式联络您所在地区的维护协议管理部门：

国家/地区	销售热线	电子邮件
中国大陆	800 810 8826	China-Sales@symantec.com
中国台湾	0080 1611 391	Taiwan-Sales@symantec.com
中国香港特别行政区	800 963 421	HongKong-Sales@symantec.com

文档

您对产品文档的反馈对我们很重要。请发送改进建议和有关错误或疏漏的报告。请在您的报告中包括所报告的文本内容的文档标题和文档版本（位于第二页上）以及章节标题。请将反馈发送到：

doc_feedback@symantec.com

关于 Symantec Connect

Symantec Connect 是为 Symantec 企业客户提供的点对点技术社区网站。参与者可以与其他产品用户联络并共享信息，包括创建论坛帖子、文章、视频、下载、博客和提出建议，并可与 Symantec 产品团队和技术支持进行交流。内容会由社区进行评分，成员可凭其贡献获得奖励积分。

http://www.symantec.com/connect/storage-management

其他企业服务

Symantec 全面提供各种服务以使您能够充分利用您对 Symantec 产品的投资，并拓展您的知识、技能和全球视野，让您在管理企业安全风险方面占据主动。

现有下列企业服务：

安全托管服务	托管服务消除了管理和监控安全设备和事件的负担，确保能够对实际威胁快速响应。
咨询服务	Symantec 咨询服务由 Symantec 及其可信赖的合作伙伴提供现场专业技术指导。Symantec 咨询服务提供各种预先包装和可自定义的服务选项，其中包括评估、设计、实施、监控和管理功能。每种功能都注重于建立和维护您的 IT 资源的完整性和可用性。
教育服务	教育服务提供全面的技术培训、安全教育、安全认证和安全意识交流计划。

要访问有关企业服务的更多信息，请通过以下 URL 访问我们的网站：

http://www.symantec.com/zh/cn

目录

技术支持 3

第 1 章 Veritas File System 软件开发人员工具包 11

关于软件开发人员工具包 11
File System 软件开发人员工具包的功能 11
API 库接口 11
文件更改日志 12
多卷支持 12
Veritas File System I/O 13
软件开发人员工具包软件包 13
所需的库和头文件 13
编译环境 14
使用不同的编译器重新编译 14

第 2 章 文件更改日志 17

关于文件更改日志文件 17
记录的更改 17
使用文件更改日志文件 18
文件更改日志记录的激活 19
文件更改日志文件布局 20
记录类型 22
特殊记录 23
典型记录序列 23
文件更改日志可调参数 24
可调参数如何处理文件更改日志增长的大小 25
文件更改日志的应用程序编程接口 26
易于使用 26
向后兼容 26
API 函数 27
文件更改日志记录 34
复制文件更改日志记录 39
Veritas File System 和文件更改日志的升级和降级 43
反向路径名称查找 44
Inode 45
vxfs_inotopath_gen 45

第 3 章 多卷支持 47
关于多卷支持 47
多卷支持的用途 48
卷应用程序编程接口 48
管理卷集 48
查询文件系统的卷集 49
修改文件系统内的卷 49
封装和取消封装卷 50
分配策略应用程序编程接口 50
指示文件分配 50
创建并指派策略 52
查询已定义的策略 52
强制执行策略 53
数据结构 53
使用策略和应用程序编程接口 54
定义并指派分配策略 54
使用卷应用程序编程接口 55

第 4 章 指定数据流 57
关于指定数据流 57
指定数据流的用途 58
指定数据流应用程序编程接口 58
列出指定数据流 60
指定数据流的命名空间 60
其他系统调用中的行为更改 61
查询指定数据流 61
应用程序编程接口 62
命令参考资料 63

第 5 章 Veritas File System I/O 65
关于 Veritas File System I/O 65
冻结和解冻 65
缓存顾问 67
直接 I/O 68
并行 I/O 68
无缓冲的 I/O 69
其他顾问 69
扩展区 70
扩展区属性 70
保留：为文件预分配空间 71
固定扩展区大小 72

扩展区属性的应用程序编程接口 72
分配标志 73
用于固定扩展区大小的分配标志 75
如何使用扩展区属性 API 75
设置固定扩展区大小 75

第 6 章 精简回收 77

关于精简存储 77
关于精简回收 77
精简回收应用程序编程接口 77
vxfs_ts_reclaim 返回值 79

索引 81

Veritas File System 软件开发人员工具包

本章节包括下列主题：

■ 关于软件开发人员工具包

■ File System 软件开发人员工具包的功能

■ 软件开发人员工具包软件包

■ 所需的库和头文件

■ 编译环境

关于软件开发人员工具包

Veritas File System (VxFS) 软件开发人员工具包 (SDK) 为开发人员提供了使用应用程序编程接口 (API) 修改和调整 Veritas File System 的各种功能和组件的必要信息。这些 API 随 VxFS 软件开发人员工具包一起提供。

VxFS 4.0 版本及其后续版本中提供了本文档所涉及的大多数 API。

File System 软件开发人员工具包的功能

本节提供可使用 SDK 访问的 VxFS 功能的概述。

API 库接口

此 SDK 中提供的重要的 API 库接口是 `vxfsutil` 库和 VxFS IOCTL 指令。 该库包含 API 调用的集合，应用程序可以使用这些调用来利用 VxFS 文件系统的功能。 所有 API 接口均提供有手册页。

表 1-1 列出了 VxFS API 库中提供的 API 调用和功能。

表 1-1 库 API 和功能

API	功能
inotopath	Inode-to-path 查询
nattr	指定数据流
FCL	文件更改日志
MVS	多卷支持
Caching Advisories	IOCTL 指令
Extents	IOCTL 指令
Freeze/Thaw	IOCTL 指令

VxFS API 库 vxfsutil，可独立于 Veritas File System 产品安装。 此库通过使用 stubs 库和动态库组合来实现。 使用 stubs 库 libvxfsutil.a 编译应用程序，使应用程序可移植到任何 VxFS 目标环境。 随后应用程序就可以在 VxFS 目标上运行，stubs 库将找到随 VxFS 目标一起提供的动态库。

stubs 库使用 vxfsutil.so 动态库位置的默认路径。 大多数情况下，应使用默认路径。 但可以通过将环境变量 *LIBVXFSUTIL_DLL_PATH* 设置为 vxfsutil.so 库的路径来改写默认路径。 此结构使得部署应用程序时出现的、与其他版本 VxFS 兼容的问题减至最低限度。

文件更改日志

VxFS 文件更改日志 (FCL) 跟踪对文件系统中文件和目录的更改。 应用程序（如，备份产品、Web crawler、搜索和索引引擎以及副本软件）可以使用文件更改日志，它们通常扫描整个文件系统，搜索自上一次扫描以来的修改。

请参见第 17 页的“关于文件更改日志文件”。

多卷支持

多卷支持 (MVS) 功能允许 VxFS 文件系统使用多个 Veritas™ Volume Manager (VxVM) 卷作为基础存储。 管理员和应用程序可以控制存储文件的位置，从而最大化有效性能，同时将开销降至最低。 此功能只能在 Veritas Volume Manager 中使用。 此外，某些功能需要提供附加的许可证密钥。

请参见第 47 页的“关于多卷支持”。

Veritas File System I/O

VxFS 遵循系统 V 接口定义 (SVID) 要求并支持用户通过网络文件系统 (NFS) 进行访问。 其他文件系统中不能提供所要求的性能功能的应用程序，可利用 VxFS 增强版。

软件开发人员工具包软件包

VRTSfssdk软件包由SDK组成。VRTSfssdk软件包包含库、头文件和示例程序的源代码及二进制文件格式，说明了 VxFS API 接口在开发和编译应用程序时的用法。VRTSfssdk 软件包也包含本指南和 API 手册页。

VRTSfssdk 软件包的目录结构如下：

src	包含若干子目录，并针对每个感兴趣的主题提供有示例程序和基于GNU 的 Makefile 文件。
bin	包含指向源目录中所有示例程序的符号链接，以便于轻松访问二进制文件。
include	包含 API 库和 ioctl 接口的头文件。
lib	包含预编译的 vxfsutil API 接口 stubs 库。
libsrc	包含 vxfsutil API 接口 stubs 库的源代码。

可以单独从 VxFS 软件包获取 VRTSfssdk 软件包。要运行应用程序或示例程序，具有授权的 VxFS 目标必不可少。此外，应在目标系统上安装所需功能的 VxFS 许可证。

所需的库和头文件

VRTSfssdk 软件包安装在 /opt 目录中。关联的库和头文件安装在以下位置：

- /opt/VRTSfssdk/6.0/lib/libvxfsutil.a
- /opt/VRTSfssdk/6.0/include/vxfsutil.h
- /opt/VRTSfssdk/6.0/include/sys/fcl.h
- /opt/VRTSfssdk/6.0/include/sys/fs/vx_ioctl.h

还提供有从以下标准 Veritas 路径指向这些文件的符号链接：/opt/VRTS/lib 和 /opt/VRTS/include。在最新版本的VxFS和VxFS SDK中，标准路径是默认路径。

编译环境

SDK 软件包将示例程序与编译后的二进制文件一同安装。

运行示例程序的要求如下：

- 装有 `VRTSvxfs` 适用版本的目标系统
- Root 权限，这对某些程序而言是必不可少的
- 已装入的 VxFS 6.0 或更高版本的文件系统。某些程序可能要求在 Veritas Volume Set 上装入文件系统。

注意：某些程序可能要求特殊的卷配置（卷集）。

此外，某些程序要求在卷集上装入文件系统。

使用不同的编译器重新编译

重新编译 `src` 或 `libsrc` 目录所需的工具如下：

- `gmake` 或 `make` 命令
- `gmake` 命令
- `gmake` 命令
- `gmake` 命令
- `gcc` 编译器或 `cc` 命令
- `gcc` 编译器
- `gcc` 编译器
- `gcc` 编译器

重新编译 src 和 libsrc 目录

1. 编辑 `make.env` 文件并用指向您编译器的路径对其进行修改。
2. 转至 `src` 或 `libsrc` 目录，然后运行 `gmake` 或 `make` 命令：

   ```
   # gmake
   ```

3. 在写入应用程序后，请对其进行如下编译：

   ```
   # gcc -I /opt/VRTSfssdk/6.0/include
   -L /opt/VRTSfssdk/6.0/lib -ldl -o MyApp \
   MyApp.c libvxfsutil.a
   ```

要编译 src 或 libsrc 目录，请按以下操作编辑 `/opt/VRTSfssdk/6.0/make.env` **文件：**

1. 在本地系统上选择编译器路径。将 `CC` 变量设置为系统上的此路径：

   ```
   CC=/opt/SUNWspro/SUNWspro/bin
   #CC=/usr/local/bin/gcc
   ```

   使用适合您的编译器的路径。

2. 转至 `src` 或 `libsrc` 目录，然后键入：

   # **gmake**

   或

   # **make**

2

文件更改日志

本章节包括下列主题：

- 关于文件更改日志文件
- 记录类型
- 文件更改日志可调参数
- 文件更改日志的应用程序编程接口
- 反向路径名称查找

关于文件更改日志文件

VxFS 文件更改日志 (FCL) 跟踪对文件系统中文件和目录的更改。

以使用 FCL 为代表的应用程序通常需要执行以下任务：

- 扫描整个文件系统或子集
- 发现自上一次扫描以来的更改

这些应用程序可能包括： 备份实用程序、webcrawler、搜索引擎和副本程序。

注意： FCL 跟踪数据更改的时间并记录更改类型，但不跟踪实际的数据更改。 由应用程序负责检查文件以确定更改过的数据。

记录的更改

文件更改日志记录下列文件系统更改：

- 创建
- 链接

- 取消链接
- 重命名
- 数据附加
- 数据重写
- 数据裁截
- 扩展属性的修改
- 打孔
- 其他文件属性更新

注意：仅磁盘布局版本 6 及更高版本支持 FCL。

在文件系统命名空间中，FCL 在稀疏文件中存储更改，也称为 FCL 文件。FCL 文件总是位于 /*mount_point*/lost+found/changelog。 FCL 文件行为类似于普通文件，但一些用户级操作（例如写）会被禁止。 标准系统调用 `open`(2)、`lseek`(2)、`read`(2) 和 `close`(2) 可访问 FCL 文件中的数据。 所有其他系统调用，如 `mmap`(2)、`unlink`(2) 和 `ioctl`(2) 在 FCL 文件上则不被允许。

警告：为与将来的 VxFS 版本兼容，FCL 文件可能被取出命名空间，这些标准系统调用可能不再有效。因此，Symantec 建议使用编程接口来开发所有新的应用程序。

请参见第 26 页的"文件更改日志的应用程序编程接口"。

使用文件更改日志文件

VxFS 通过向 FCL 文件附加与文件系统更改相关的信息，来跟踪对文件系统所做的更改。

这样，您就可以执行以下操作：

- 使用 FCL 确定通常在文件系统上进行的或在一个特定即时点之后在特定文件上进行的操作序列。
  例如，增量备份应用程序可扫描 FCL 文件，确定自文件系统上一次备份后哪些文件被添加或修改。
- 配置 FCL 跟踪其他信息（例如，文件打开、I/O 统计信息）和访问信息（例如，用户 ID）。

  然后，可使用此信息收集下列数据：

  - 空间使用量统计信息，用于确定不同类型数据的空间使用方式。

■ 针对不同用户在文件系统中使用不同文件的用法配置文件，可帮助确定最近访问过的数据及访问者。

空间使用量

当文件系统接近满时，可使用 FCL 文件跟踪空间使用量。 可搜索 FCL 文件查找最近创建的文件（ 文件创建 ）或写入记录确定新增的文件或最近增大的现有文件。

根据应用需要，可对整个 FCL 文件进行搜索，或者对与特定时间范围对应的部分 FCL 文件进行搜索。此外，您可以查找用特定名称创建的文件。例如，如果用户正在下载占用过多空间的 `*.mp3` 文件，可读取 FCL 文件查找用名称 `*.mp3` 创建的文件。

减少全面系统扫描

VxFS 为在启用 FCL 的文件系统上执行的每个更新操作创建并记入 FCL 记录。 这些操作包括创建、删除、重命名、模式更改和写入。因此，增量备份应用程序或根据文件名、文件属性或者内容维护文件系统索引的应用程序，可以通过读取 FCL 文件，检测自从上一次备份或上一次索引更新以来发生更改的文件，来避免全面系统扫描。

文件历史记录跟踪

您可以通过扫描 FCL 文件和整合文件的 FCL 记录序列来跟踪文件的历史记录。 还可以使用与文件的创建、属性更改、写入记录和删除相关的 FCL 记录，以跟踪文件的历史记录。

文件更改日志记录的激活

默认情况下，停用 FCL 日志记录，但可以使用 `fcladm` 命令针对每个文件系统激活日志记录。

请参见 `fcladm`(1M) 手册页。

激活 FCL 日志记录后，新的 FCL 记录将在文件系统发生更改时被附加到 FCL 文件。关闭 FCL 日志记录后，将进一步停止记录，但是 FCL 文件仍然为 `/lost+found/changelog`。使用 `fcladm` 命令，您只能删除 FCL 文件。

FCL 文件中包含一个表示布局或是 FCL 文件的内部表示形式的关联版本，以及在 FCL 文件中记录的事件列表。

每当发布 VxFS 的新版本时，将会出现下列情况：

■ 在 FCL 文件中可能有记录的其他事件

■ FCL 文件的内部表示形式可能会发生更改

通过这种方式 FCL 文件版本获得更新。例如，在 VxFS 4.1 中，默认版本是版本 3。但是，由于 VxFS 5.0 及更高版本会记录版本 3 中不可用的其他事件集（例如，文件打开），因此，VxFS 5.0 及更高版本中的默认版本是版本 4。为了向 VxFS 4.1 上开发的应用程序提供向后兼容，VxFS 5.0 及更高版本提供了一个用于在激活过程中指定 FCL 版本的选项。根据指定的版本，可以允许或禁止记录新记录类型的日志记录。

对于大多数在 VxFS 5.0 及更高版本中新添加的记录的日志记录（例如，文件打开和 I/O 统计数据）都是可选的，默认情况下处于关闭状态。可以使用 `fcladm` 命令的 [set] 和 [clear] 选项来启用或禁用这些事件的记录。

包含文件系统状态、版本以及被跟踪事件集的 FCL 元数据信息，在重新启动前后，以及文件系统卸载或装入时都是持久性的。版本和事件信息在重新激活 FCL 日志记录前后也是持久性的。

文件更改日志文件布局

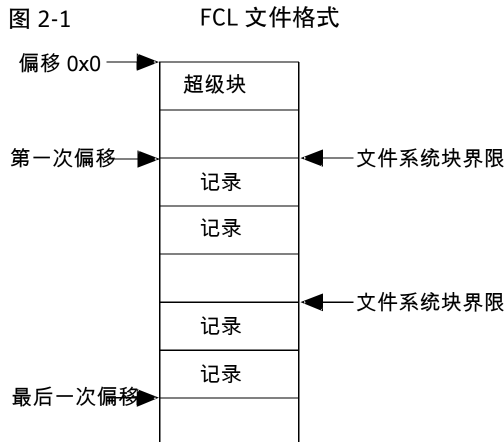
在 VxFS 4.1 中，FCL 文件的内部布局对用户是透明的，并且应用程序可使用标准文件系统接口（例如，`open`(2)、`read`(2) 和 `lseek`(2)）来访问 FCL 文件。不过，此方法可能会导致将来的兼容性问题，这是因为如果基本的 FCL 布局和 FCL 版本发生了更改，则必须更改和重新编译应用程序，以适应这些更改。

VxFS 5.0 引入了一个新的编程接口，即使磁盘上的 FCL 布局发生更改，该接口也会提供改进的兼容性。有了该 API，对于应用程序来讲，FCL 布局变得不再那么重要。因此，本节仅提供了对 FCL 布局的简要说明。

FCL 文件通常是一个包含 FCL 超级块和 FCL 记录的稀疏文件。FCL 文件中的第一个信息块是 FCL 超级块。此块可以后接一个可选孔，以及包含有关文件系统中更改信息的 FCL 记录。

图 2-1 描述了 FCL 文件格式。

图 2-1　　FCL 文件格式



文件更改日志超级块

对文件系统中文件和目录的更改将存储为 FCL 记录。 超级块（当前存储在 FCL 文件中的第一个块）描述了 FCL 文件的状态。

超级块指明下列操作：

- 是否启用 FCL 日志记录
- 它是何时激活的
- 第一条和最后一条 FCL 记录的当前偏移
- FCL 文件版本
- 当前被跟踪的事件集的事件掩码
- 上次更改事件掩码的时间

使用 `fcladm on` 命令首次激活 FCL 时，将创建仅包含超级块的 FCL 文件。 使用 `fcladm rm` 命令删除 FCL 文件时，仅删除超级块。

使用 `fcladm on` 激活 FCL 时，超级块中的状态及其激活时间将发生更改。 只要任何文件系统活动导致记录被附加到 FCL 文件，最后一个偏移就会获得更新。

当 FCL 文件增大时，根据文件系统可调参数 `fcl_maxalloc` 和 `fcl_keeptime`，在 FCL 文件开头的最旧记录将在第一个偏移获得更新时被丢掉，以释放一些空间。使用 `fcladm` 命令的 [set] 或 [clear] 选项更改在 FCL 文件中被跟踪的事件集时，事件掩码和事件掩码更改时间也将随之更新。事件掩码更改还会导致事件掩码更改记录（包含旧事件掩码更改和新事件掩码更改）被记录在 FCL 文件中。

文件更改日志记录

FCL 记录包含有关这些典型更改的信息：

- 更改后的文件的 inode 编号
  请参见第 45 页的“Inode”。
- 更改的时间
- 更改的类型
- 基于记录类型的可选信息

  根据记录类型， FCL 记录还可能包括下列信息：

  - 父级 inode 编号
  - 文件删除、链接和相似的操作的文件名
  - 文件打开记录的命令名
  - I/O 统计信息记录的实际统计信息

请参见第 20 页的图 2-1。

记录类型

表 2-1 列出了生成 FCL 记录类型的操作。

表 2-1 FCL 记录类型

创建 FCL 记录的操作	记录类型
添加指向现有的文件或目录的链接	VX_FCL_LINK
附加写入到文件	VX_FCL_DATA_EXTNDWRITE
创建文件或目录	VX_FCL_CREATE
创建指定数据流目录	VX_FCL_CREATE
创建符号链接	VX_FCL_SYMLINK
在共享和可写入模式下对文件执行 mmap 操作	VX_FCL_DATA_OVERWRITE
从存储检查点提升文件	VX_FCL_UNDELETE
在文件中打孔	VX_FCL_HOLE_PUNCHED
删除文件或目录	VX_FCL_UNLINK
删除指定数据流目录	VX_FCL_UNLINK
重命名文件或目录	VX_FCL_RENAME
将文件重命名为现有的文件名	VX_FCL_UNLINKVX_FCL_RENAME
设置文件属性（分配策略、ACL 和扩展属性）	VX_FCL_EATTR_CHG
设置文件扩展区保留	VX_FCL_INORES_CHG
设置文件扩展区大小	VX_FCL_INOEX_CHG
设置文件组所有权	VX_FCL_IGRP_CHG
设置文件模式	VX_FCL_IMODE_CHG
设置文件大小	VX_FCL_DATA_TRUNCATE
设置文件用户所有权	VX_FCL_IOWN_CHG
设置文件的 mtime	VX_FCL_MTIME_CHG
裁截文件	VX_FCL_DATA_TRUNCATE

创建 FCL 记录的操作	记录类型
写入到文件的现有的块	VX_FCL_DATA_OVERWRITE
打开文件	VX_FCL_FILEOPEN
将文件的 I/O 统计数据写入到 FCL	VX_FCL_FILESTATS
更改在 FCL 中跟踪的事件集	VX_FCL_EVNTMSK_CHG

注意：表 2-1 列出了当 fcladm on 命令激活 FCL 日志记录时，默认记录的所有事件（除 fileopen 和 filestat 之外）。

默认情况下，也不会记录这些事件中每个事件的访问信息。使用 fcladm 命令的 [set] 选项可以记录打开信息、I/O 统计数据和访问信息。

请参见 fcladm(1M) 手册页。

这些记录类型属于 fcl_chgtype.t. fcl_chgtype.t，它是在 fcl.h 头文件中定义的枚举。

请参见第 38 页的表 2-2。

特殊记录

下列记录类型通过 API 不再是可见的：

- VX_FCL_HEADER
- VX_FCL_NOCHANGE
- VX_FCL_ACCESSINFO

典型记录序列

在文件系统中，文件从创建到删除的生命周期记录在 FCL 文件中。

创建文件时，下列是写入日志的 FCL 记录的典型序列：

```
VX_FCL_CREATE
VX_FCL_FILEOPEN (if tracking file opens is enabled)
VX_FCL_DATA_EXTNDWRITE
VX_FCL_IMODE_CHG
```

写入文件时，会将下列其中一项 FCL 记录写入到每次写操作的日志中。该记录取决于是在当前文件末尾写入还是在文件内写入。

VX_FCL_DATA_EXTNDWRITE
VX_FCL_DATA_OVERWRITE

以下显示了当文件 a 重命名为 b 并且这两个文件均位于文件系统中时，写入日志的 FCL 记录的典型序列：

VX FCL_UNLINK（适用于文件 b，如果它已经存在）

VX_FCL_RENAME（适用于将文件 a 重命名为 b）

文件更改日志可调参数

您可以使用 vxtunefs 命令设置四个 FCL 可调参数。

请参见 vxtunefs(1M) 手册页。

下列是四个可用 FCL 可调参数：

fcl_keeptime
指定 FCL 记录在它们可以被清除之前在 FCL 文件保留的持续时间（以秒为单位）。最旧的记录将第一批被清除，这些记录位于文件的开头。此外，如果对 FCL 文件的分配超过 fcl_maxalloc 字节数，则将清除文件开头的记录。 默认值为 0。 请注意，fcl_keeptime 的优先级高于 fcl_maxalloc。 如果 FCL 文件超过 fcl_maxalloc 字节数，但最旧的记录的生命周期尚未到达 fcl_keeptime 秒，则不会打孔。

调整建议： 只有当管理员希望确保记录在 FCL 中的保留时间达到 fcl_keeptime 时，才需要调整 fcl_keeptime 可调参数。 应将 fcl_keeptime 参数设置为大于 FCL 扫描时间间隔的任意值。 例如，如果 FCL 每 24 个小时扫描一次，则可以将 fcl_keeptime 设置为 25 个小时。 这样可以防止在读取和处理 FCL 记录之前将其清除。

fcl_maxalloc
指定要分配给 FCL 文件的最大空间量（以字节为单位）。 当分配的空间超过 fcl_maxalloc 时，将在文件的开头打孔。结果，将清除记录并更新 FCL 超级块中的第一个有效偏移。 fcl_maxalloc 的最小值是 4MB。 默认值是 *fs_size*/33。

fcl_winterval
指定 FCL 为同一 inode 记录多次重写、扩展写入或裁截记录之前所经历的时间（以秒为单位）。 这样可以减少 FCL 中重复记录的数。 fcl_winterval 超时基于每个 inode。 如果 inode 出现缓存不足并返回，则其写入时间间隔将重置。这样，在同一写入时间间隔内，该文件会有多条写入记录。 默认值为 3600 秒。

调整建议： 应该将 fcl_winterval 可调参数设置为小于 FCL 扫描时间间隔的值。 例如，如果 FCL 每 24 个小时扫描一次，则可以将 fcl_winterval 设置为少于 24 个小时。 这样可以确保在两次扫描期间对于将要被重写、扩展或裁截的每个文件，在 FCL 中至少有一条记录。

`fcl_ointerval` 指定后续打开文件的操作不生成其他 FCL 记录的时间间隔（以秒为单位）。 这有助于减少在 FCL 中记录的重复性文件打开记录数，特别是在经常通过 NFS 进行访问的情况下。 如果还启用了访问信息的跟踪，则在 fcl_ointerval 内的后续文件打开事件可能会生成一条记录（如果后面的打开是由另一个用户执行的）。 与 fcl_ointerval 类似，如果 inode 缓存不足并返回，或存在 FCL 同步，则在同一打开时间间隔内可能会有多个文件打开记录。 默认值为 600 秒。

调整建议：如果使用文件打开记录的应用程序只需要知道，自从它上次扫描 FCL 以来是否有任何用户访问了文件，则可以将 fcl_ointerval 设置为两次扫描之间的时间范围内的一个时间段。如果应用程序关注对每次访问进行跟踪，则可以将可调参数设置为零。

在通过 NFS 大量访问文件系统的情况下，根据具体的平台和 NFS 实施，将可能记录大量的文件打开记录。在此情况下，建议将可调参数设置为更高的值，以避免 FCL 中存在大量的重复性记录。

可调参数如何处理文件更改日志增长的大小

图 2-2 显示了当 FCL 文件增大时清除记录的示例。

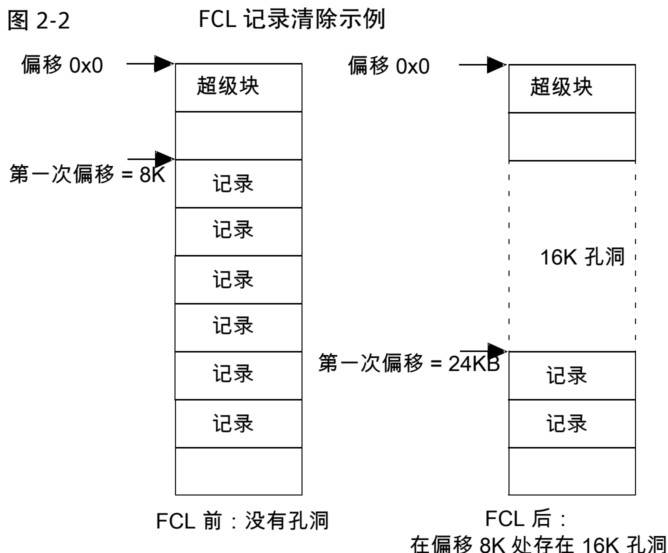
左边的 FCL 文件包含 8K 的块且没有打孔。 当文件系统发生活动时，便将活动记录在 FCL 中，这导致右边的 FCL 文件增大。

当 FCL 文件大小到达 fcl_maxalloc 可调参数指定的所允许的最大大小时，将会清除更旧的记录并释放空间。 FCL 功能只清除其时间早于 fcl_keeptime 指定的时间的那些记录。 被释放的空间始终以一个内部孔大小为单位。

图 2-2 显示了在 FCL 文件中以 8K 为单位释放空间的文件系统。

当 FCL 文件首次超过最大分配时，旧的记录数是 20K，程序将清除 16K 的记录。这样会在 FCL 超级块后面留下一个 16K 的孔。 FCL 超级块中的第一个有效偏移将更新为 24K。

图 2-2 FCL 记录清除示例



文件更改日志的应用程序编程接口

除通过 `libvxfsutil:vxfs_fcl_sync` 提供的现有编程接口之外，VxFS 5.0 及更高版本还提供了一组新的编程接口，可替换通过标准系统调用（比如 `open`(2)、`lseek`(2)、`read`(2) 和 `close`(2)）集访问 FCL 文件的机制。此 API 提供了以下改进：

易于使用

API 减少了写入其他代码以解析 FCL 条目的需求。

大多数磁盘上的 FCL 记录的大小都是固定的，并且仅包含默认的信息，例如，inode 编号或时间戳。但是，有些记录的大小是可变的，例如，文件删除记录或重命名记录。这些记录包含其他信息，例如，被删除或重命名的文件的名称。

若要确保任何文件系统块开头的少数字节始终是有效的 FCL 记录（如果文件名跨越块边界），可以跨磁盘上的多条记录对文件系统块进行拆分。以前，您需要写入其他代码来组合这些记录以获得文件名。现在，VxFS 5.0 及更高版本中的 API 提供了一种机制，可以直接读取单个组合逻辑记录。这使得应用程序对 API 的应用更容易。通过 API，应用程序还可以指定一个筛选器，以指示所关注事件的子集并仅返回所需记录。

向后兼容

通过 API，应用程序可读取不受 FCL 布局更改影响的 FCL。 例如，考虑这样一个方案，应用程序可直接访问并解释磁盘上的 FCL 记录。 如果下一个 VxFS 发行版添加

了新记录或更改了记录在FCL文件中的存储方式，则应用程序需要重新编写或至少重新编译，以适应这种更改（在先前的 VxFS 版本中）。

通过中间 API，磁盘上的 FCL 布局对于应用程序是隐藏的，因此，即使 FCL 的磁盘布局发生了更改，API 也会在内部转换数据，并向用户返回期望的输出记录。用户应用程序仍可以继续运行而无需重新编译或重新编写。这隔离了 FCL 布局更改对程序的影响，并为现有的应用程序提供更好的兼容性。

API 函数

API 使用下列类型的函数：

- 访问 FCL 记录的函数
- 寻找偏移和时间戳的函数

访问文件更改日志记录的函数

以下是访问 FCL 记录的常见函数：

vxfs_fcl_open	打开 FCL 文件并返回进一步操作可以使用的句柄。所有通过 API 对 FCL 文件进行的后续访问都必须使用该句柄。
vxfs_fcl_close	关闭 FCL 文件并清除与该句柄关联的资源
vxfs_fcl_getinfo	返回 FCL 版本号以及 FCL 文件的状态（打开/关闭）
vxfs_fcl_read	将用户感兴趣的 FCL 记录读取到用户使用的缓冲区
vxfs_fcl_copyrec	复制FCL记录。如果源记录包含指针，则会重定位这些指针以指向新的位置。

寻找文件更改日志中的偏移和时间戳的函数

用户可选择根据他们停止使用的偏移寻找文件更改日志中的特定点，或寻找指定时间后的第一条记录。

下列函数可以寻找 FCL 中的偏移和时间戳：

vxfs_fcl_getcookie	返回一个内含当前FCL激活时间和当前偏移的不透明的结构（以下称为 cookie）。可以保存该 cookie，并稍后将其传递到 `vxfs_fcl_seek` 中，以便从上次应用程序停止的位置继续读取。
vxfs_fcl_seek	从传递的 cookie 中提取数据并寻找指定的偏移。cookie 中将内含 FCL 激活时间和文件偏移。
vxfs_fcl_seektime	寻找指定时间后 FCL 中的第一条记录。

vxfs_fcl_open

以下是 `vxfs_fcl_open`() 函数的语法：

```
int vxfs_fcl_open(char *pathname, int flags, void **handle);
```

此函数可打开 FCL 文件，并且返回通过 API（例如，`vxfs_fcl_read`、`vxfs_fcl_seek` 等）对 FCL 进一步访问需要使用的句柄。

`vxfs_fcl_open` 有两个参数：**pathname* 和 ***handle*。**pathname* 可以是一个指向 FCL 文件名或装入点的指针。如果 **pathname* 是一个装入点，`vxfs_fcl_open` 将会自动确定是否激活装入点上的 FCL 并打开与装入点（当前为 *mount_point*`/lost+found/changelog`）关联的 FCL 文件。

`vxfs_fcl_open` 然后确定它是否为有效 FCL 文件，以及 FCL 文件版本是否与库兼容。然后，`vxfs_fcl_open`() 函数将有关 FCL 文件的元数据信息提取到一个不透明的内部数据结构中，并用指针填充 ***handle*。

就像 `lseek`(2) 和 `read`(2) 系统调用一样，FCL 文件 ***handle* 中有一个内部偏移，用于指示文件中下次读取开始的位置。成功打开 FCL 文件后，将此偏移设置为 FCL 文件中的第一个有效偏移。

返回值

成功完成后，将向调用方返回 0 并且句柄非空。 否则，API 返回一个非零值，并且句柄设置为 NULL。 此外还将设置全局值 errno 以指明错误。

vxfs_fcl_close

`vxfs_fcl_close` 关闭句柄引用的 FCL 文件。 所有与该句柄一起分配的数据结构将被清除。 调用 `vxfs_fcl_close` 之后不应再使用该句柄。

参数

```
void vxfs_fcl_close(void *handle)
```

**handle* 是先前调用 `vxfs_fcl_open` 时返回的有效句柄。

vxfs_fcl_getinfo

```
int vxfs_fcl_getinfo(void *handle, struct fcl_info*fclinfo);
```

`vxfs_fcl_getinfo`() 函数会返回有关 *fcl_info* 所指向的 FCL 信息结构中的 FCL 文件的信息。它从 FCL 超级块获得此信息。

```
struct fcl_info {
    uint32_tfcl_version;
```

```
    uint32_tfcl_state;
};
```

能够识别与每条记录关联的记录类型的智能应用程序，可以通过 *fcl_version* 确定 FCL 文件是否包含所需信息。例如，版本 3 FCL 文件从不包含访问信息以及 FCL 记录。如果 fcl_state 为 FCLS_OFF，则应用程序可以推断出，由于文件系统活动而没有任何记录被添加到 FCL 文件。

返回值

0 表示成功；否则，将为错误设置错误号，并返回一个非零值。

vxfs_fcl_read

通过此函数，应用程序可以读取实际文件或作为逻辑记录在 FCL 中记录的目录更改信息。 每个记录都返回一个 struct fcl_record 类型。通过 vxfs_fcl_read，应用程序可以指定一个包含一组期望事件的筛选器。

参数

以下是 vxfs_fcl_read() 函数的语法：

```
int vxfs_fcl_read(void *hndl, char *buf, size_t *bufsz,
uint64_t eventmask, uint32_t *nentries);
```

输入

此函数包含下列输入：

- **hndl* 是先前调用 vxfs_fcl_open 时返回的指针
- **buf* 是指向缓冲区大小的指针，缓冲区大小至少为 **bufsz*
- **bufsz* 用于指定缓冲区的大小
- *eventmask* 是位掩码，用于指定应用程序感兴趣的事件集。它应该是在 fcl.h 头文件中指定的事件掩码集的“逻辑或”。例如，如果 *eventmask* 为 (VX_FCL_CREATE_MASK | VX_FCL_UNLINK_MASK)，vxfs_fcl_read 仅返回文件创建和删除记录。
  如果应用程序需要读取所有记录类型，该应用程序可以将默认 *eventmask* 掩码指定为 FCL_ALL_V4_EVENTS。这将返回 FCL 文件中的所有有效版本 4 FCL 记录。
  请参见第 22 页的表 2-1。

注意：如果在 *eventmask* 中设置了 VX_FCL_EVENTMASKCHG_MASK，并且 vxfs_fcl_read 返回的记录包含一条 VX_FCL_EVENTMASK_CHG 记录，则它始终是缓冲区中的最后一条记录。这样，应用程序可以根据需要重新调整 *eventmask*。此外，如果应用程序从 *eventmask* 更改记录中发现某个特定事件不再被记录，则它可以决定停止进一步读取。

- **nentries* 用于指定在调用 vxfs_fcl_read 过程中应读入缓冲区的条目数。如果 **nentries* 为 0，vxfs_fcl_read 将读取缓冲区能够容纳的最大条目数。

输出

如果没有错误，**buf* 包含 struct fcl_record 类型的 **nentries* FCL 记录。

如果请求的条目数量与传递大小的缓冲区不匹配，则将返回 FCL_ENOSPC 错误。 在这种情况下，将更新 **bufsz* 以便使包含的缓冲区大小能够容纳请求数量的记录。应用程序可以使用此方式来重新分配一个更大缓冲区，然后重新调用 vxfs_fcl_read。 如果没有错误，**bufsz* 将不会发生更改。

如果调用 vxfs_fcl_read 时没有错误，**nentries* 将会更新以包含读入缓冲区的条目数。 如果应用程序已到达文件的末尾并且没有要读取的记录，则 **nentries* 和返回的值将均为零。

返回值

0 表示成功；非零表示出现错误。

注意：如果缓冲区中没有足够的空间来存储当前记录，将返回 FCL_ENOSPC。 **bufsz* 返回的是缓冲区的最小大小。

成功调用 vxfs_fcl_read 后，当前的文件位置将会向前移动，以便下次调用 vxfs_fcl_read 时可以读取下一记录集。

vxfs_fcl_getcookie

vxfs_fcl_getcookie 和 vxfs_fcl_seek() 函数是一种记住应用程序先前在 FCL 文件中处理过的位置的有效方法。随后，它可用作重新启动点。对应用程序而言，这是一种非常有用的工具。

请参见第 31 页的“vxfs_fcl_seek”。

vxfs_fcl_getcookie() 函数返回一个不透明的 fcl_cookie 结构，该结构内含了包含 FCL 文件的当前激活时间和一个指示 FCL 文件中当前位置的偏移的信息。此 cookie 可传递到 vxfs_fcl_seek 以寻找此 cookie 定义的 FCL 文件中的位置。

典型的增量备份或索引更新程序可读取至 FCL 文件的末尾并根据 FCL 记录执行操作。该应用程序可使用 vxfs_fcl_getcookie 获取关于 FCL 文件中当前位置的信息，然后将此cookie存储在一个持久性结构（例如文件）中。当应用程序下次需要执行增量操作时，它将读取 cookie 并将其传递给 vxfs_fcl_seek 以寻找其停留的点。这样应用程序将只读取新的 FCL 记录。

参数

以下是 vxfs_fcl_getcookie() 函数的语法：

```
int vxfs_fcl_getcookie(void *handle, struct fcl_cookie *cookie)
```

函数参数如下：

- *handle 是调用 vxf_fcl_open 时返回的 FCL 文件句柄
- *cookie 是指向不透明的数据块的指针，其定义如下：

```
struct fcl_cookie {
    char    fc_bytes[24];
};
```

存储在 cookie 中的数据位于 VxFS 库的内部。应用程序不应采用 cookie 的任何内部表示形式，也不应篡改 cookie 中的数据。

vxfs_fcl_seek

可以使用 vxfs_fcl_seek 根据传递给它的标志来寻找 FCL 文件的起始或结尾。

请参见第 30 页的“vxfs_fcl_getcookie”。

参数

下列是 vxfs_fcl_seek() 函数的语法：

```
int vxfs_fcl_seek(void *handle, struct fcl_cookie *cookie, int where)
```

函数参数如下：

- *handle参数与最近调用vxfs_fcl_open()时返回的句柄应是同一句柄。它不必与在 vxfs_fcl_getcookie() 中使用的句柄相同。应用程序可以在一次会话中打开 FCL 文件、获取 cookie 并关闭 FCL 文件，然后在随后的会话中打开 FCL 文件并提交被保存的 cookie。 对于 FCL 文件上每个打开的会话，有效句柄是该会话的 vxfs_fcl_open() 返回的句柄。
- *cookie 参数应指向一个有效的 cookie（该 cookie 是从同一 FCL 文件上调用 vxfs_fcl_getcookie() 时返回的），或指向同一 FCL 文件的一个检查点，或同一 FCL 文件的一个转储或还原的副本。 用户应用程序负责确定哪个 FCL 文件对于特定的 cookie 是有效的，并以合理的组合方式应用它们。

注意：如果 *where* 的值为 FCL_SEEK_SET 或 FCL_SEEK_END，则 ***cookie*** 参数可以为 NULL。

- *where* 参数的值应为 FCL_SEEK_SET、FCL_SEEK_END 或 FCL_SEEK_COOOKIE。
  - 如果 *where* 为 FCL_SEEK_SET 或 FCL_SEEK_END，**cookie* 参数将被忽略，并且 vxfs_fcl_seek() 将分别寻找 FCL 文件的开头或末尾（即，第一条 FCL 记录的开始位置或最后一条记录结束的位置）。
  - 如果 *where* 的值为 FCL_SEEK_COOKIE，vxfs_fcl_seek() 会提取存储在 **cookie* 参数中的激活时间和偏移。

  如果 FCL 自应用程序上次调用 vxfs_fcl_getcookie() 以来已停用（关闭），或者在 **cookie* 中包含的偏移处的记录由于要打孔而被清除，vxfs_fcl_seek() 将返回 FCL_EMISSEDRECORD 错误。否则，vxfs_fcl_seek 将当前的文件位置设置为 cookie 中包含的偏移。进一步调用 vxfs_fcl_read() 将从该偏移返回记录。

返回值

0 表示成功；非零表示出现错误。

注意：如果重新激活 FCL（即，FCL 中的激活时间不同于 cookie 中传递的激活时间，或 FCL 文件中的第一个有效偏移大于 cookie 中现有的偏移），vxfs_fcl_seek() 将返回 FCL_EMISSEDRECORD。

---

vxfs_fcl_seektime

vxfs_fcl_seektime() 函数会寻找 FCL 文件中时间戳大于或等于指定时间的第一条记录。

参数

以下是 vxfs_fcl_seektime() 函数的语法：

```
int vxfs_fcl_seektime(void *handle, struct fcl_timeval time)
```

函数参数如下：

- **handle* 是先前调用 vxfs_fcl_open 时返回的有效句柄
- *time* 是一种 fcl_time_t 结构类型，其定义如下：

```
struct fcl_time {
    uint32_t tv sec;
    unit32_t tv_nsec;
} fcl_time t;
```

注意：在 fcl_time_t 中指定的时间是秒或纳秒，而由标准系统调用（例如，gettimeofday）返回的时间可以为秒或微秒。因此，可能需要进行转换。

---

vxfs_fcl_seektime 假定 FCL 中的条目是以时间戳的非降序排列的，而且执行比线性（二进制）更快的搜索，以确定其时间戳大于指定时间的 FCL 记录。这意味着当与通过线性搜索的寻找进行比较时，vxfs_fcl_seektime 可能会寻找到一条不同的记录。因此，vxfs_fcl_seektime 接口并不完全可靠。

FCL 中的时间戳在下列情况下可能是无序的：

- 如果修改了系统时间
- 如果 FCL 文件位于集群装入的文件系统上，并且不同的节点上的时间不同步

警告：在集群文件系统上，必须使用一个机制来保持系统时钟同步（例如，网络时间协议 - NTP），以确保 vxfs_fcl_seektime 接口保持适当的准确性。

返回值

vxfs_fcl_seektime 成功时返回 0。 如果 FCL 文件中没有任何比 *time* 参数指示的时间更新的记录，则 vxfs_fcl_seektime 将返回 EINVAL。

vxfs_fcl_sync

vxfs_fcl_sync() 函数在 FCL 文件内设置同步点。保留此函数是为了实现向后兼容。

在可以使用 VxFS 5.0 API 访问 FCL 文件之前，应用程序通常会调用 vxfs_fcl_sync 使 FCL 处于稳定的状态，并在 FCL 中设置偏移以用作停止读取的参考点。然后应用程序将存储偏移，并使用它来确定自上次 FCL 读取时间以来文件的更改。vxfs_fcl_sync() 调用可确保，如果写入或打开一个文件，在同步偏移之后的 FCL 中将至少有一条相应的写入或打开记录。即使自上次记录写入以来，由 fcl_winterval 或 fcl_ointerval 指定的时间尚未过去，也会发生这种情况。

如果在 VxFS 5.0 和更高版本中使用 FCL 访问 API，当通过 vxfs_fcl_open() 打开 FCL 文件时，将自动进行同步。vxfs_fcl_open() 函数可设置同步点，并在内部确定参考端点偏移。

参数

以下是 vxfs_fcl_sync() 函数的语法：

```
int vxfs_fcl_sync(char *fname, uint64_t *offp);
```

函数参数如下：

- **fname* 是指向 FCL 文件名的指针

- *offp* 是 64 位偏移的地址

vxfs_fcl_sync 使 FCL 文件处于稳定状态，并且使用可用作应用程序参考点的偏移来更新 *offp*。

文件更改日志记录

应用程序通过 vxfs_fcl_read() 函数来读取 FCL 文件。

vxfs_fcl_read 执行下列任务：

- 从 FCL 文件读取数据
- 将数据组合到 fcl_record 结构
- 将这些记录填充到由应用程序传递的缓冲区

每个 fcl_record 结构均表示一个在 FCL 记录的逻辑事件，其定义如下：

```
struct fcl_record {
  uint32_t  fr_reclen;           /* Record length */
  uint16_t  fr_op;               /* Operation type. */
  uint16_t  fr_unused1;          /* unused field */
  uint32_t  fr_acsinfovalid : 1;   /* fr_acsinfo field valid */
  uint32_t  fr_newnmvalid : 1; /* fr_newfilename field is valid */
  uint32_t  fr_pinogenvalid : 1; /* fr_fr_pinogen field is valid */
  uint32_t  fr_unused2 : 29;       /* Future use */
  uint64_t  fr_inonum;           /* Inode Number. */
  uint32_t  fr_inogen;           /* Inode Generation Count. */
  fcl_time_t fr_time;            /* Time. */
  union fcl_vardata {
    char                     *fv_cmdname;
    struct fcl_nminfo        fv_nm;
    struct fcl_iostats       *fv_stats;
    struct fcl_evmaskinfo    fv_evmask;
} fr_var;
  uint64_t              fr_tdino;             /* Target dir ino */
  char                   *fr_newfilename;      /* For rename */
  struct fcl_acsinfo     *fr_acsinfo;          /* Access Info */
};
struct fcl_nminfo {
  uint64_tfn_pinonum;/* Parent Inode Number. */
  uint32_tfn_pinogen;/* Parent Inode Gen cnt. */
  char*fn_filename;
};
struct fcl_evmaskinfo {
```

```
  uint64_toldmask;/* Old event mask. */
  uint64_tnewmask;/* New event mask. */
};
```

定义

提供了下列定义以更加便于访问：

```
#define fr_cmdname     fr_var.fv_cmdname
#define fr_stats       fr_var.fv_stats
#define fr_oldmask     fr_var.fv_evmask.oldmask
#define fr_newmask     fr_var.fv_evmask.newmask
#define fr_pinonum     fr_var.fv_nm.fn_pinonum
#define fr_pinogen     fr_var.fv_nm.fn_pinogen
#define fr_filename    fr_var.fv_nm.fn_filename
```

fcl_iostats 结构

通过 VxFS 5.0 和更高版本，您可以收集诸如在文件上发生的读取和写入次数等统计数据。您可以通过 `fiostat` 命令来启用该结构。收集的统计数据维护在基于文件的内核结构中，文件更改日志充当这些统计数据的持久性支持存储区。

在下列情况下将统计数据写入到 FCL：

- 需要释放内核结构
- 重置统计数据
- 在定期时间间隔

可以将这些统计数据作为 `VX_FCL_FILESTAT` 记录从 FCL 读取。每个记录均包含按下列 `fcl_iostat` 结构定义的信息：

```
struct fcl_iostats {
    uint64_t nbytesread; /* Number of bytes read from the file*/
    uint64_t nbyteswrite;/* Number of bytes written to the file*/
    uint32_t nreads;     /* Number of reads from the file */
    uint32_t nwrites;    /* Number of writes to the file */
    uint32_t readtime;   /* Total time in seconds for the reads */
    uint32_t writetime;  /* Total time in seconds for the writes */
    struct {
        uint32_t   tv_sec;
        uint32_t   tv_nsec;
} lastreset;/*           Last reset time for the stats */
uint32_tnodeid;          /* Node from which the record was written */
uint32_treset;           /* Stats have been written due to a reset */
};
```

FCL 中的每个 iostat 记录均包含从 *lastreset* 时间到写入 FCL 记录之间这段时间内累积的 I/O 统计数据。

经过一段时间后，累积的统计数据和总计可以通过以下方式计算：

- 遍历 FCL
- 查找 VX_FCL_FILESTATS 类型的记录

例如，计算一段时间内读取的总量需求遍历一组 FCL 文件以获得 I/O 统计数据记录。此信息包含在同一 `lastreset` 时间内 `VX_FCL_FILESTATS` 类型的一系列记录，后接在随后的 `lastreset` 时间内特定文件的一系列记录。

总计仅考虑同一 `lastreset` 时间内的记录中的最新记录中的值，然后对为每个这样的记录读取的次数进行汇总。

fcl_acsinfo 结构

当启用跟踪访问信息时，VxFS 将记录如下访问信息：

- 访问应用程序的真实和有效的用户 ID 和组 ID
- 从其上访问文件的节点
- 用户应用程序的进程 ID 以及每条记录

当应用程序读取 FCL 时，将在 `fr_acsinfo` 字段中返回信息。

`fr_acsinfo` 指向 FCL`_acsinfo` 结构，其定义如下：

```
struct fcl_acsinfo {
    uint32_tfa_ruid;
    uint32_tfa_rgid;
    uint32_tfa_euid;
    uint32_tfa_egid;
    uint32_tfa_pid;
    uint32_tfa_nodeid;
};
```

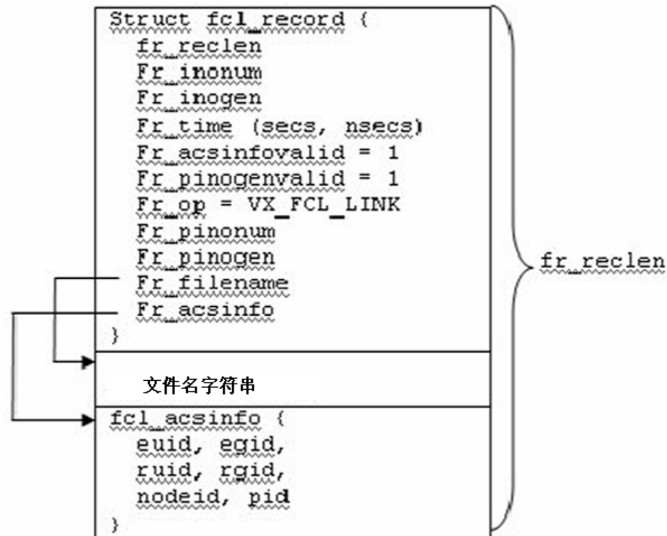
注意：`accessinfo` 不作为单独的记录类型返回，而是作为其他信息随其他记录返回。 此外，accessinfo 信息并不始终随每条记录出现（例如，当未启用跟踪 `accessinfo` 时）。 但是，在某些文件系统内部的操作（例如，删除文件时裁截文件）中，即使启用 `accessinfo`，访问信息也可能不会出现。 为了帮助确定访问信息是否可用，FCL 记录包含一个名为 `fcl_acsinfovalid` 的标志，只有当 `accessinfo` 与特定记录一起出现时，该标志才是一个非零值。

`fcl_acsinfo` 结构中的多个字段均是指针，并且需要内存存储实际的内容。这可以通过在FCL记录后立即存储实际的数据，并更新指向该数据的指针来进行处理。记

录长度 *fr_reclen* 字段将被更新，以描述整个数据。因此，由 vxfs_fcl_read 返回的每个 FCL 记录的大小都是可变的，其长度由 *fr_reclen_field* 指示。

图 2-3 显示了示例链接记录中数据的布局方式。

图 2-3 示例链接记录



下列代码示例遍历了调用 vxfs_fcl_read 时返回的记录集，并输出了用户 ID。

```
Struct fcl_record*fr;
Char          *tbuf;
...
      error = vxfs_fcl_read(fh, buf, &bufsz,
                         FCL_ALL_V4_EVENTS,
                         &nentries);
    tbuf = buf;
    while (--nentries) {
        fr = (struct fcl_record *)tbuf;
if (fr->fr_acsinfovalid) {
        printf("Uid %ld\n", fr->fr_acsinfo->uid;
}
        tbuf += fr->fr_reclen;
   }
```

注意： FCL_ALL_V4_EVENTS 是事件掩码。

请参见第 29 页的“vxfs_fcl_read”。

记录结构字段

表 2-2 简要描述了 fcl_record 结构的每个字段，并指明有效字段的记录类型。

表 2-2　　FCL 记录结构字段

字段	说明	有效性
fr_reclen	FCL 记录的长度。这包括 FCL 记录结构长度和紧随结构之后存储的数据的长度。当遍历缓冲区中由 vxfs_fcl_read 返回的 fcl 记录时，应使用该长度。	对所有记录均有效。
fr_inonum	正在被更改的文件的 inode 编号。若要生成已更改对象的完整路径名，可以将 inode 编号和生成计数 (*fr_inogen*) 与 vxfs_inotopath_gen 一起使用。	对除 FCL_EVNTMSK_CHG 之外的所有 FCL 记录有效。对于事件掩码更改，该文件是隐含的 FCL 文件。
fr_op	针对此 FCL 记录的操作。例如，创建、取消链接、写入、文件属性更改或者其他更改。fr_op 显示为表 2-1 中所列出记录类型的其中一个值。 使用此参数可以确定 FCL 记录的哪些字段是有效的。	对所有记录均有效。
fr_time	在 FCL 文件中记录更改的大约时间。使用 ctime() 调用可解释此字段。	对所有记录均有效。
fr_inogen	已更改文件的生成计数。将生成计数与（文件的）inode 编号的组合传递到 vxfs_inotopath_gen 中，以提供对象的准确完整的路径名。如果不用生成计数，返回的路径名可以是一个重新使用的 inode。	对所有 FCL 记录有效，但对事件掩码更改和取消链接例外。对于事件掩码更改来说，inode 编号和生成计数是隐含的。对于取消链接来说，为了通过反向名称查找来获取文件名时，生成计数是不需要的；因为它已经与记录一起出现。

字段	说明	有效性
fr_pinonum fr_pinogen fr_filename	对于像文件删除或重命名这样的 FCL 记录来说，其中的目录条目已经被删除，因此不能通过反向名称查找来确定文件名。同样，在链接记录的情况下，也无法明确地确定文件名。因此，在这些情况下，将会记录父级目录（包含正在更改的文件）的文件名、inode 编号和生成计数。可以将父级目录 inode (*fr_pinonum*) 和生成计数 (*fr_pinogen*) 与反向名称查找 API 一起使用，以标识父级目录的完整路径名。添加后续文件名可生成对象的完整名称。	当 FCL 记录为 VX_FCL_UNLINK、VX_FCL_RENAME 或 VX_FCL_LINK 时有效。取消链接和重命名，文件名和父级 inode 编号，以及生成计数，包含有关删除的旧文件的信息。对于链接，它们表示新的文件名。
fr_cmdname	打开了由 *fr_inonum* 和 *fr_inogen* 提供的文件的命令的简短名称。	只有当 FCL 记录为 VX_FCL_FILEOPEN 时有效。
fr_stats	指向 FCL_iostat 记录的指针。fcl_iostat 记录包含一些 I/O 统计数据，例如，在文件上发生的读取/写入的次数、读取/写入的平均时间等。这些即时点记录可以用于计算在一段时间内某个文件的总计或平均 I/O 统计数据。	只有当 FCL 记录为 VX_FCL_FILESTATS 时有效。
fr_oldmask fr_newmask	这些字段分别包含新的和旧的事件掩码。每个事件掩码都是在 fcl.h 中定义的一组掩码的“逻辑或”。	只有当 FCL 记录为 VX_FCL_EVNTMASK_CHG时有效。
fr_acsinfo	指向 FCL_acsinfo 结构的指针。此结构包含如下信息，例如，执行特定操作的应用程序的用户 ID 和组 ID、进程 ID 和访问节点的 ID。	有效性由 fcl_acsinfovalid bit-field 确定。它潜在地可与各种记录放置在一起。这是一个可选字段。

复制文件更改日志记录

由 vxfs_fcl_read 返回的每个 FCL 记录的大小都是可变的，并且包含 fcl_record 结构，后接与该记录相关联的其他数据。fcl_record 结构中的指针指向存储在 fcl_record 结构之后的数据，并且记录长度指定大小可变的记录的大小。但是，生成 FCL 记录的内核副本包含从源到副本多次复制的 fr_reclen 数据字节数。

简单的内存复制仅通过指针将数据从源记录复制到目标记录。这使得目标记录中的指针指向源记录中的数据。最终，当重新使用或释放为源记录提供的内存时，将会出现问题。必须修改副本中的指针，使其指向目标记录中的数据。因此，要生成 FCL 记录的内核副本，应用程序必须使用 `vxfs_fcl_copyrec()` 函数来复制和执行指针重定位。用户应用程序必须为此复制操作分配所需要的内存。

索引维护应用程序

此示例应用程序适用于维护文件系统中所有文件的索引，以启用快速搜索功能的系统，类似于 Liux 系统中的 `locate` 程序。系统需要定期更新索引，或根据自上次索引更新以来有关文件更改的需要进行更新。以下列出了执行并显示了调用 FCL API 示例的基本步骤。

准备应用程序

1 启用 FCL。

$ **fcladm on *mount_point***

2 将 *fcl_keeptime* 和 *fcl_maxalloc* 调整为所需值。

$ **vxtunefs -o fcl_keeptime=*value* *mount_point***
$ **vxtunefs -o fcl_maxalloc=*value* *mount_point***

测试应用程序

1 打开 FCL 文件。

$ **vxfs_fcl_open(*mount_point*, 0, &fh);**

2 寻找文件末尾。

$ **vxfs_fcl_seek(fh, NULL, FCL_SEEK_END);**

3 获取 cookie 并将其存储在文件中。

$ **vxfs_fcl_getcookie(fh, &cookie)**
write(fd, cookie, sizeof(struct fcl_cookie));

4 创建索引。

更新应用程序

1 打开 FCL 文件。

$ **vxfs_fcl_open(*mount_point*, 0, &fh);**

2 读取 cookie 并寻找 cookie。

$ **read(fd, &cookie, sizeof(struct fcl_cookie))**

$ **vxfs_fcl_seek(fh, cookie, FCL_SEEK_COOKIE)**

3 读取 FCL 文件并相应地更新索引。

$ **vxfs_fcl_read(fh, buf, BUFSZ, FCL_ALL_v4_EVENTS, &nentries)**

4 获取 cookie 并将其存储回文件中。

$ **vxfs_fcl_getcookie(fh, &cookie)**

$ **write(fd, cookie, sizeof(struct fcl_cookie));**

计算用法配置文件

此示例应用程序计算特定文件的用法配置文件，即，在最后一小时内有权访问特定文件的用户。

初始安装

此示例应用程序需要其他的信息（例如，跟踪文件打开和访问信息），这些信息仅在 FCL 版本 4 中可用，请确保启用了正确的 FCL 版本。

下列步骤执行所需的初始安装。

设置应用程序

1 打开 FCL 版本 4。

```
$ fcladm -o version=4 on mount_point
```

如果此步骤失败，请使用 `fcladm print` 检查是否存在 FCL 版本 3 文件。如果有，请使用 `fcladm rm` 删除该文件，然后尝试打开 FCL 版本 4。

在 VxFS 5.0 和更高版本中，默认 FCL 版本是版本 4。如果不存在 FCL 文件，`fcladm on` *`mount_point`* 命令将自动创建一个版本 4 的 FCL。

2 根据需要启用跟踪访问信息、文件打开和 I/O 统计数据。

```
$ fcladm set fileopen,accessinfo mount_point
```

3 根据需要设置可调参数 `fcl_keeptime`、`fcl_maxalloc` 和 `fcl_ointerval`。例如：

```
$ vxtunefs fcl_ointerval=value mount_point
```

示例步骤

下列提供了应用程序可能使用的示例步骤。

示例应用程序安装

1 打开 FCL 文件。

```
vxfs_fcl_open(mount_point, 0, &fh);
```

2 设置执行寻找的时间。

3 使用 `gettimeofday` 获取当前时间。

4 构造一小时前时间的 `fcl_time_t`。

5 寻找当时 FCL 文件中的记录。

```
gettimeofday(&tm, NULL);
tm.sec -= 3600
vxfs_fcl_seektime(fh, tm);
```

6 使用适当的事件掩码读取文件，直到到达文件尾为止。应用程序仅关注文件打开记录和访问信息。

7 检查文件 inode 编号和生成计数是否与寻找每个 FCL 记录的 inode 编号和生成计数相同。

8 输出有关已访问该文件的用户的信息（如果适用）。

```
vxfs_fcl_read(fh, buf, BUFSZ, VX_FCL_FILEOPEN_MASK |
\VX_FCL_ACCESSINFO_MASK, &nentries);
```

脱离主机处理

在某些情况下，用户应用程序可能会选择节省实际生产服务器的带宽，并将处理 FCL 的工作外包给不同的系统。对于脱离主机处理，需要将 FCL 文件发送到脱离主机系统。由于 FCL 文件不是一个普通文件，`cp` 或 `ftp` 等命令不起作用。

要成为“可发送”文件，必须首先使用 `fcladm dump` 命令将 FCL 文件转储为普通文件，然后，使用普通文件传输程序将该文件发送到脱离主机系统。请参见下列示例。

```
$ fcladm -s savefile dump mount_point$ rcp savefile offhost-path
```

在脱离主机系统上，必须使用 *restore* 选项通过 `fcladm` 命令还原 FCL 文件。与原始 FCL 文件不同，还原后的文件是一个普通文件。

```
$ fcladm -s savefile restore restorefile
```

可以将还原后的 FCL 文件作为一个参数传递到 `vxfs_fcl_open`，以进一步用于 FCL API。

警告：反向名称查找 API 在脱离主机系统上不能正常工作。仅当应用程序可以使用 inode 编号和生成计数时，或它有一个独立方法可根据 inode 编号确定文件名时，才应使用脱离主机处理机制。

Veritas File System 和文件更改日志的升级和降级

VxFS 4.1 仅支持 FCL 版本 3。VxFS 5.0 和更高版本支持 FCL 版本 3 和版本 4，默认情况下使用版本 4。将系统从 VxFS 4.1 升级到 VxFS 5.0 或更高版本，且文件系统已打开 FCL 时，现有的版本 3 FCL 文件仍保持不变。VxFS 5.0 和更高版本会继续跟踪版本 3 FCL 中的文件系统更改，操作方式与在 VxFS 4.1 中完全相同。

如果 FCL 文件是版本 3，则使用 `read`(2) 系统调用直接访问 FCL 文件的 VxFS 4.1 应用程序在 VxFS 5.0 和更高版本中仍可以继续工作，但是，必须使用 API 来开发新应用程序。API 的支持版本为 FCL 版本 3 和 4。

如果新的应用程序使用在 VxFS 5.0 版本中添加的记录类型（例如，文件打开或访问信息），则 FCL 应该为版本 4。

如果您运行的应用程序仍可直接读取 FCL 版本 3，则在重新编写这些应用程序以使用新的 API 之前，则无法将这些应用程序升级到 FCL 版本 4。API 可以解释版本 3 和版本 4，因此在版本 3 仍可用时可以升级应用程序以使用该 API。

将文件更改日志版本 3 文件转换为版本 4 文件

将 VCL 版本 3 文件转换成版本 4 文件

1 关闭 FCL。

```
$ fcladm off mount_point
```

2 删除现有的 FCL 文件。

```
$ fcladm rm mount_point
```

3 重新激活所需版本。

```
$ fcladm [-oversion=4] on mount_point
```

降级 Veritas File System 版本

将来，特定系统上的 VxFS 版本可能需要从较新的 VxFS 版本降级到 VxFS 5.0。将文件系统从一个使用较新 VxFS 版本的操作系统上迁移到另一个使用 VxFS 5.0 版本的系统上时，可能会发生这种情况。如果此将来的 VxFS 版本创建的 FCL 文件是版本 3 或版本 4，则在安装 VxFS 5.0 后仍可正常使用。在同一 FCL 上将继续跟踪更改。

但是，如果 FCL 版本高于版本 4，则无法激活 FCL，并且对 API 函数的调用将会失败。在这种情况下，需要使用 `fcladm rm` 删除现有的 FCL 文件，并重新激活 FCL 版本 3 或版本 4 中的该文件。

反向路径名称查找

反向路径名称查找功能可以从文件或目录的 inode 编号获得该文件或目录的完整路径名称。 inode 编号作为 `vxfs_inotopath_gen` 库函数的参数提供。 有关更多信息，请参见 `vxfs_inotopath_gen`(3) 联机手册页。

反向路径名称查找功能对于各种应用程序均十分有用，这些应用程序包括：

- VxFS 文件更改日志功能的客户端
- 备份和还原实用程序
- 复制产品

通常情况下，这些应用程序按照 inode 编号来存储信息，因为文件或目录的路径名称可能会很长，应用程序需要一个简单的方法来获取路径名称。

Inode

Inode 是文件系统中每个文件的唯一标识号。 Inode 包含与该文件相关联的数据和元数据，但不包括 inode 对应的文件名。 因此通过 inode 编号确定文件名称相对来说是困难的。 ncheck 命令提供了一个通过扫描文件系统中的每个目录，从 inode 标识号获得文件名的机制，但此过程可能需要较长时间。 VxFS 反向路径名称查找功能可相对快速地获得路径名称。

注意：由于符号链接不构成文件的路径，反向路径名称查找功能不能跟踪文件的符号链接。

文件 inode 编号、生成计数、以及在 VX_FCL_LINK， VX_FCL_UNLINK 或 VX_FCL_RENAME 记录情况下的尾部文件名，当与反向路径名称查找结合使用时，可为每个 FCL 记录生成完整路径名称。

vxfs_inotopath_gen

vxfs_inotopath_gen() 函数使用装入点名称、inode 编号和 inode 生成计数，并返回一个缓冲区，该缓冲区包含表示 inode 的一个或多个（在多个链接指向一个 inode 的情况下）完整路径名称。inode 生成计数参数确保返回的路径名称不是重新使用的 inode 的一个错误值。因此，请尽可能地使用 vxfs_inotopath_gen() 函数。

包含 vxfs_inotopath() 函数只是为了向后兼容。vxfs_inotopath() 函数不使用 inode 生成计数。

下列是 vxfs_inotopath 和 vxfs_inotopath_gen 函数的语法：

```
int vxfs_inotopath(char *mount_point, uint64_t inode_number,
                      int all, char ***bufp, int *inentries)
int vxfs_inotopath_gen(char *mnt_pt, uint64_t inode_number,
                      unint32_t inode_generation, int all,
                      char ***bufp, int *nentries)
```

对于 vxfs_inotopath() 调用，所有参数要么为 0 以获得一个路径名称，要么为 1 以获得所有路径名称。*mount_point* 参数指定文件系统装入点。在成功返回后，*bufp* 指向一个包含路径名称的二维字符指针，并且 nentries 包含项的数量。返回的二维阵列中的每个项的大小为 MAXPATHLEN，并且必须将其与阵列一起由调用应用程序释放。

vxfs_inotopath_gen() 调用还使用了另一个参数 *inode_generation*，除此之外，它与 vxfs_inotopath() 调用完全相同。如果传递的 *inode_generation* 与 inode

编号的当前生成计数匹配，则 vxfs_inotopath_gen() 函数将返回一个或多个与给定 inode 编号关联的路径名称。如果生成计数有所不同，它将返回一个错误。当生成计数未知时，指定 inode_generation=0。这与使用 vxfs_inotopath() 调用等效。

仅版本 6 及更高版本的磁盘布局支持 vxfs_inotopath_gen() 和 vxfs_inotopath() 调用。

3

多卷支持

本章节包括下列主题：

- 关于多卷支持
- 多卷支持的用途
- 卷应用程序编程接口
- 分配策略应用程序编程接口
- 数据结构
- 使用策略和应用程序编程接口

关于多卷支持

通过多卷支持 (MVS) 功能，VxFS 文件系统可使用多个 VxVM 卷作为基础存储，来代替传统的每个文件系统一个卷。这些不同的卷可以具有不同的特性，如性能、冗余或开销，或者它们可以出于性能或管理目的用于将文件系统的不同部分彼此隔离。

管理员和应用程序可通过使用分配策略来控制哪些文件和元数据进入哪些卷。分配空间的每个文件系统操作都将检查应用的分配策略以查看为该操作指定了哪些卷。分配策略通常只影响新的分配，但也有一些接口用于移动现有数据以匹配新的分配策略。

以下策略级别可应用每个分配：

- 基于每个文件的策略
- 基于每个存储检查点的策略
- 基于每个文件系统的策略

将使用为给定分配操作生效的最特定的分配策略。

MVS API 划分为以下基本类别：

■ 在文件系统内处理卷

■ 处理分配策略定义

■ 应用分配策略

每个 API 也可通过 fsvoladm(1M) 和 fsapadm(1M) 命令的选项提供。

请参见 fsvoladm(1M) 和 fsapadm(1M) 手册页。

多卷支持的用途

多卷支持功能的可能用途包括如下：

■ 控制文件存储的位置，以便将特定文件或文件层次结构指派到不同的卷

■ 分隔存储检查点，以便将分配给存储检查点的数据与文件系统的其余部分隔开

■ 将文件系统元数据与文件数据分隔开

■ 封装卷，以便使卷作为文件出现于文件系统中，这对于在原始卷上运行的数据库特别有用

■ 从卷迁移文件，以便可以替换卷或为其提供服务

■ 执行存储优化应用程序，该应用程序定期地扫描文件系统，并且修改分配策略，以响应对存储使用模式的更改

卷应用程序编程接口

可以使用卷 API 向文件系统添加卷、从文件系统删除卷、列出文件系统中的卷，并检索卷中有关空间的使用情况和可用性的信息。

多卷文件系统只能与 VxVM 卷集一起使用。卷集是通过 vxvset 命令来管理的。

请参见《Veritas Storage Foundation 管理指南》。

管理卷集

下列示例显示了如何管理卷集。

将卷转换为卷集

■ 若要将 myvol1 转换成卷集，请使用以下函数调用：

```
# vxvset make myvset myvol1
```

将卷添加到卷集

■ 若要将 myvol2 添加到卷集 myvset 中，请使用以下函数调用：

```
# vxvset addvol myvset myvol2
```

列出卷集中的卷

- 若要列出 myvset 中的卷，请使用以下函数调用：

```
# vxvset list myvset
```

从卷集中删除卷

- 若要从 myvset 删除 myvol2，请使用以下函数调用：

```
# vxvset rmvol myvset myvol2
```

查询文件系统的卷集

下列函数调用可查询文件系统的卷集。

查询所有与文件关联的卷

- 若要查询所有与文件系统关联的卷，请使用以下函数调用：

```
vxfs_vol_enumerate(fd, &count, infop);
```

查询单个卷

- 若要查询单个卷，请使用以下函数调用：

```
vxfs_vol_stat(fd, vol_name, infop);
```

修改文件系统内的卷

下列函数调用可修改文件系统内的卷。

增大或减小卷的大小

- 若要增大或减小卷的大小，请使用以下函数调用：

```
vxfs_vol_resize(fd, vol_name, new_vol_size);
```

从文件系统删除卷

- 若要从文件系统删除卷，请使用以下函数调用：

```
vxfs_vol_remove(fd, vol_name);
```

向文件系统添加卷

- 若要向文件系统添加卷，请使用以下函数调用：

```
vxfs_vol_add(fd, new_vol_name, new_vol_size);
```

封装和取消封装卷

下列函数调用可封装卷。

封装原始卷

- 若要封装现有的原始卷并使卷内容在文件系统中显示为文件，请使用以下函数调用：

```
vxfs_vol_encapsulate(encapsulate_name, vol_name, vol_size);
```

取消封装原始卷

- 若要取消封装现有的原始卷以便从文件系统中删除文件，请使用以下函数调用：

```
vxfs_vol_deencapsulate(encapsulate_name);
```

请参见《Veritas Storage Foundation 管理指南》。

分配策略应用程序编程接口

若要充分应用多卷支持功能，VxFS 支持多种分配策略，允许将待分配的文件或文件组指派到卷集内的指定卷。

分配策略指定卷列表并尝试分配卷的顺序。可以将策略指派给文件、文件系统或者从文件系统创建的存储检查点。在将策略指派给文件系统中的对象时，必须指定策略同时映射到元数据和文件数据的方式。例如，如果将策略指派给单个文件，则文件系统必须知道放置文件数据和元数据的位置。如果未指定策略，文件系统会随机放置数据。

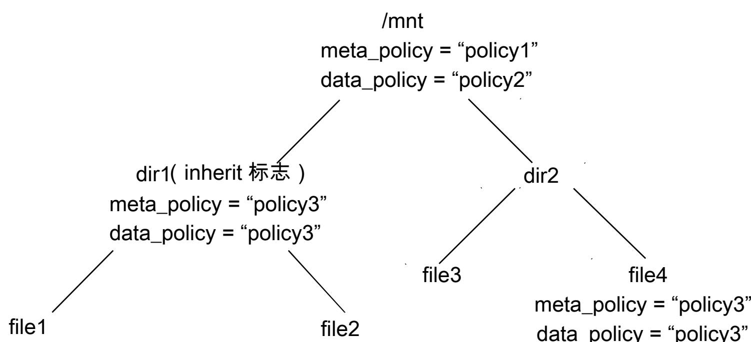
分配策略按文件系统进行定义，并且是持久性的。在文件系统中，对于分配策略定义的数量没有固定的限制。指派了策略之后，新的文件分配将由该策略进行管理。对于在定义或分配策略之前分配的文件，或者应用于文件的策略发生更改的情况，则可以强制执行策略，从而可以将文件重新分配到适当的卷。分配策略可以由新创建的文件从其父级目录继承。 这可以通过在将策略指派给父级目录时指定 `FSAP_INHERIT` 标志来完成。

目前，没有接口可用于确定现有文件的当前分配位置。但是，使用这些 API 来指派和强制执行文件上的策略，可确保能够正确分配块。

指示文件分配

图 3-1 显示如何使用分配策略来指示文件分配。

图 3-1 指示文件分配



/mnt 文件系统在其卷集有 3 个卷： vol-01、vol-02 和 vol-03。 这些卷分别对应于 policy1、policy2 和 policy3。

该文件系统采用策略指派，按 policy1 的指示分配数据，按 policy2 的指示分配元数据。 通过这些策略可将文件分配到 vol-01 和 vol-02，但 dir1 策略除外，它优先指派 vol-03 上的分配。

创建 file3 和 file4 文件后，它们将按照 policy1 和 policy2 指派的指示在 vol-02 上进行分配。 创建 file1 和 file2 后，它们将按照 policy3 的指定在 vol-03 上进行分配。

创建 file4 后，初始分配在 vol-01 和 vol-02 上进行。 若要将 file4 移动到 vol-03，请将 policy3 指派给 file4，并在该文件上强制执行该策略。 这样会将 file4 重新分配给 vol-03。

指示文件分配

1 在 /mnt 文件系统创建分配策略。

2 将数据和元数据分配策略指派给 /mnt 文件系统，作为 policy1 和 policy2。

3 将数据和元数据分配策略指派给 dir1（带有 INHERIT 标志），作为 policy3。

4 创建 file4 (100MB)，分配给 vol-02。

5 创建 file3 (10MB)，分配给 vol-02。

6 创建 file2 (100MB)，分配给 vol-03。

7 创建 file1 (100MB)，分配给 vol-03。

8 将数据和元数据分配策略指派给 file4，作为 policy3。

9 强制执行 file4 上的分配策略，这样可将文件重新分配给 vol-03。

创建并指派策略

以下函数调用将使用多卷 API 创建并指派一项策略。

- 若要为文件系统定义策略，请使用以下函数调用：

```
vxfs_ap_define(fd, fsap_info_ptr, 0);
```

- 若要将策略指派给文件系统，请使用以下函数调用：

```
vxfs_ap_assign_fs(fd, data_policy, meta_policy);
```

- 若要将策略指派给文件或目录，请使用以下函数调用：

```
vxfs_ap_assign_file(fd, data_policy, meta_policy, 0);
```

- 若要将策略指派给存储检查点，请使用以下函数调用：

```
vxfs_ap_assign_ckpt(fd, checkpoint_name, data_policy, meta_policy);
```

- 若要将策略指派给所有存储检查点，请使用以下函数调用：

```
vxfs_ap_assign_ckptchain(fd, data_policy, meta_policy);
```

- 若要为新创建的存储检查点设置默认分配策略，请使用以下函数调用：

```
vxfs_ap_assign_ckptdef(fd, data_policy, meta_policy);
```

查询已定义的策略

下列函数调用用于查询已定义的策略。

- 若要查询文件系统上的所有策略，使用以下函数调用：

```
vxfs_ap_enumerate(fd, &count, fsap_info_ptr);
```

- 若要查询一项已定义的策略，请使用以下函数调用：

```
vxfs_ap_query(fd, fsap_info_ptr);
```

- 若要查询文件中已指派的策略，请使用以下函数调用：

```
vxfs_ap_query_file(fd, data_policy, meta_policy, 0);
```

- 若要查询存储检查点中已指派的策略，请使用以下函数调用：

```
vxfs_ap_query_ckpt(fd, check_point_name, data_policy, meta_policy);
```

■ 若要查询文件系统中已指派的策略，请使用以下函数调用：

```
vxfs_ap_query(fd, data_policy, meta_policy);
```

■ 若要查询文件系统中默认的存储检查点策略，请使用以下函数调用：

```
vxfs_ap_query_ckptdef(fd, data_policy, meta_policy);
```

强制执行策略

以下函数调用可强制执行策略。

■ 若要对文件强制执行策略，请使用以下函数调用：

```
vxfs_ap_enforce_file(fd, data_policy, meta_policy);
```

强制执行策略可能会导致将文件重新分配到另一个卷。

■ 若要对存储检查点中的所有文件强制执行策略，请使用以下函数调用：

```
vxfs_ap_enforce_ckpt(fd, check_point_name, data_policy,
meta_policy, flags);
```

■ 若要对主文件集及其所有存储检查点强制执行策略，请使用以下函数调用：

```
vxfs_ap_enforce_ckptchain(fd, data_policy, meta_policy, flags);
```

数据结构

您可以在 vxfsutil.h 头文件和 libvxfsutil.a 库文件中查看 fsap_info 和 fsdev_info 数据结构。

请参见 vxfsutil.h 头文件和 libvxfsutil.a 库文件。

此处提供的数据结构可作为快速参考：

```
#define FSAP_NAMESZ            64
#define FSAP_MAXDEVS            256
#define FSDEV_NAMESZ            32
struct fsap_info {              /* policy structure */
      char ap_name[FSAP_NAMESZ];/* policy name */
      uint32_t ap_flags;        /* FSAP_CREATE | FSAP_INHERIT |
                                    FSAP_ANYUSER */
      uint32_t ap_order;        /* FSAP_ORDER_ASGIVEN |
                                   FSAP_ORDER_LEASTFULL |
```

```
                                FSAP_ORDER_ROUNDROBIN */
      uint32_t ap_ndevs;        /* number of volumes */
      char ap_devs[FSAP_MAXDEVS][FSDEV_NAMESZ];
                                /* volume names associated with
                                            this policy */
};
struct fsdev_info {             /* volume structure */
      int dev_id;               /* a number from 0 to n */
      uint64_t dev_size;        /* size in bytes of volume */
      uint64_t dev_free;
      uint64_t dev_avail;
      char dev_name[FSDEV_NAMESZ];/* volume name */
};
```

使用策略和应用程序编程接口

下列示例假定有一个卷集 volset，其中的卷为 vol-01、vol-02 和 vol-03。 文件系统装入点 /mnt 被装入到 volset 上。

定义并指派分配策略

下列伪代码提供了使用分配策略 API 来定义并指派分配策略的示例。

定义并指派分配策略，以将现有的文件的数据块重新分配给特定的卷

■ 若要将现有的文件的数据块重新分配给特定的卷 (vol-03)，请创建类似如下内容的代码：

```
/* Create a data policy for moving file's data */
strcpy((char *) ap.ap_name, "Data_Mover_Policy");
ap.ap_flags = FSAP_CREATE;
ap.ap_order = FSAP_ORDER_ASGIVEN;
ap.ap_ndevs = 1;
strcpy(ap.ap_devs[0], "vol-03");
fd = open("/mnt", O_RDONLY);
vxfs_ap_define(fd, &ap, 0);
file_fd = open ("/mnt/file_to_move", O_RDONLY);
vxfs_ap_assign_file(file_fd, "Data_Mover_Policy", NULL, 0);
vxfs_ap_enforce_file(file_fd, "Data_Mover_Policy", NULL);
```

创建在目录下分配新文件的策略

在此示例中，文件位于 dir1 下、元数据分配给了 vol-01，而文件数据分配给了 vol-02。

■ 若要创建在 dir1 目录下分配新文件的策略，请创建类似如下的代码：

```
/* Define 2 policies */
/* Create the RAID5 policy */
strcpy((char *) ap.ap_name, "RAID5_Policy");
ap.ap_flags = FSAP_CREATE | FSAP_INHERIT;
ap.ap_order = FSAP_ORDER_ASGIVEN;
ap.ap_ndevs = 1;
strcpy(ap.ap_devs[0], "vol-02");
fd = open("/mnt", O_RDONLY);
dir_fd = open("/mnt/dir1", O_RDONLY);
vxfs_ap_define(fd, &ap, 0);
/* Create the mirror policy */
strcpy((char *) ap.ap_name, "Mirror_Policy");
ap.ap_flags = FSAP_CREATE | FSAP_INHERIT;
ap.ap_order = FSAP_ORDER_ASGIVEN;
ap.ap_ndevs = 1;
strcpy(ap.ap_devs[0], "vol-01");
vxfs_ap_define(fd, &ap, 0);
/* Assign policies to the directory */
vxfs_ap_assign_file(dir_fd, "RAID5_Policy", “Mirror_Policy”,
                   0);
/* Create file under directory dir1 */
/* Meta and data blocks for file1 will be allocated on
    vol-01 and vol-02 respectively. */
file_fd = open("/mnt/dir1/file1");
write(file_fd, buf, 1024);
```

使用卷应用程序编程接口

下列伪代码提供了使用卷 API 的示例。

减小或增大文件系统内的卷大小

1 使用 vxresize 命令增大物理卷的大小。

2 若要使用 vxfs_vol_resize() 调用减小或增大文件系统，请创建类似如下的代码：

```
/* stat volume "vol-03" to get the size information */
fd = open("/mnt");
vxfs_vol_stat(fd, "vol-03", infop);
/* resize (shrink/grow) accordingly. This example shrinks
       the volume by half */
vxfs_vol_resize(fd, "vol-03", infop->dev_size / 2);
```

将原始卷封装为文件

1 将卷添加到卷集。

2 若要在文件系统 /mnt 中将原始卷 vol-03 封装为名为 encapsulate_name 的文件，请创建类似如下的代码：

```
/* Take the raw volume vol-03 and encapsulate it. The
    volume’s contents will be accessible through the given
    path name. */
vxfs_vol_encapsulate("/mnt/encapsulate_name", "vol-03",
                                            infop->dev_size);
/* Access to the volume is through writes and reads of file
     "/mnt/encapsulate_name" */
encap_fd = open("/mnt/encapsulate_name");
write(encap_fd, buf, 1024);
```

取消封装原始卷

- 若要在文件系统 /mnt 中将原始卷 vol-03 取消封装名称 encapsulate_name，请创建类似如下的代码：

```
/* Use de-ecapsulate to remove raw volume. After
   de-encapsulation, vol-03 is still part of volset, but is
   not an active part of the file system. */
vxfs_vol_deencapsulate("/mnt/encapsulate_name");
```

4

指定数据流

本章节包括下列主题：

- 关于指定数据流
- 指定数据流的用途
- 指定数据流应用程序编程接口
- 列出指定数据流
- 指定数据流的命名空间
- 其他系统调用中的行为更改
- 查询指定数据流
- 应用程序编程接口
- 命令参考资料

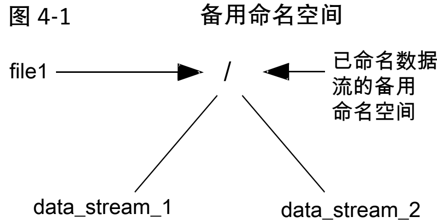
关于指定数据流

指定数据流会将多个数据流与某个文件关联起来。默认的未指定数据流可以通过文件描述符进行访问，其中文件描述符由针对文件名调用的 open() 函数返回。其他数据流存储在与该文件关联的一个备用命名空间中。

注意：指定数据流也称为命名属性。

图 4-1 显示了与文件关联的备用命名空间。

图 4-1 备用命名空间



file1 文件有两个指定数据流：data_stream_1 和 data_stream_2。

每文件都可以有其自己的备用命名空间来存储指定数据流。 可以通过 VxFS 支持的指定数据流 API 访问备用命名空间。

对指定数据流的访问可以使用指定数据流库函数通过文件描述符来完成。应用程序可以打开指定数据流，获得文件描述符，然后使用该文件描述符执行 read()、write() 和 mmap() 操作。 这些系统调用的工作方式和对普通文件执行操作类似。 文件的指定数据流存储在与该文件关联的隐藏的指定数据流目录 inode 中。 该文件的隐藏目录 inode 只能通过指定数据流应用程序编程接口进行访问。

VxFS 提供的管理命令中没有命令使用此功能。 提供的 VxFS API 用于创建、读取和写入文件的指定数据流。

此功能与 Solaris 10 管理命令兼容。

指定数据流的用途

通过指定数据流，应用程序可以将信息附加到隐藏的文件。管理程序可以使用该功能来附加文件使用信息、备份信息等等。应用程序可以使用此功能隐藏或收集文件附件。 例如，多媒体文件可以包含在一个文件（而非多个文件）中组织的所有文本、音频剪辑和视频剪辑。有多个用户在同时查看的文档可以将每个人的注释作为指定数据流附加到文件。

指定数据流应用程序编程接口

下列标准系统调用可以操作指定数据流：

open()	打开指定数据流
read()	读取指定数据流
write()	写入指定数据流
getdents ()	读取目录条目并以文件系统的独立格式放置
mmap ()	映射内存页面

readdir () 读取目录

VxFS 指定数据流功能可通过下列应用程序编程接口函数获得：

vxfs_nattr_open()

工作方式类似于 open() 系统调用，但路径被解释为文件描述符的指定数据流这一点除外。如果 vxfs_nattr_open() 操作成功完成，则返回值是与指定数据流相关联的文件描述符。文件描述符可以由其他输入/输出函数引用该指定数据流。如果指定数据流的路径设置为.，则返回的文件描述符将指向指定数据流目录 vnode。

下列是 vxfs_nattr_open() API 的语法：

```
int vxfs_nattr_open(int fd, char *path,
         int oflag, int cmode);
```

vxfs_nattr_link()

为现有的指定数据流创建新的目录条目，并将其链接计数递增 1。在指定数据流命名空间中，有一个指向现有指定数据流的指针，一个指向在指定数据流命名空间中创建的新目录条目的指针。调用函数必须具有写入权限，才能链接指定数据流。

下列是 vxfs_nattr_link() API 的语法：

```
int vxfs_nattr_link(int sfd, char *spath,
      char *tpath);
```

vxfs_nattr_unlink()

删除指定路径上的指定数据流。调用函数必须具有写入权限，才能删除指定数据流的目录条目。

下列是 vxfs_nattr_unlink() API 的语法：

```
int vxfs_nattr_unlink(int fd, char *path);
```

vxfs_nattr_rename()

将在 path1 上的指定命名空间更改为 path2 上的另一指定命名空间。指定路径相对于指向指定数据流目录 vnode 的指针进行解析。

下列是 vxfs_nattr_rename() API 的语法：

```
int vxfs_nattr_rename(int sfd, char *old,
    char *tnew);
```

vxfs_nattr_utimes() 设置指定数据流的访问和修改时间。

下列是 vxfs_nattr_utimes() API 的语法：

```
int vxfs_nattr_utimes(int sfd,
    const char *path,
    const struct timeval times[2]);
```

请参见 vxfs_nattr_open(3)、vxfs_nattr_link(3)、vxfs_nattr_unlink(3)、vxfs_nattr_rename(3) 和 vxfs_nattr_utimes(3) 手册页。

列出指定数据流

文件的指定数据流可以通过调用指定数据流目录 inode 上的 getdents() 列出，如以下示例所示。

列出指定数据流

1 若要列出指定数据流，请创建类似如下的代码：

```
fd = open("foo", O_RDWR);  /* open file foo */
afd = vxfs_nattr_open(fd, "stream1",
         O_RDWR|O_CREAT, 0777);/* create named data stream
                                  stream1 for file foo */
write(afd, buf, 1024);       /* writes to named stream file */
read(afd, buf, 1024);        /* reads from named stream file */
dfd = vxfs_nattr_open(fd, ".", O_RDONLY);
                                 /* opens named stream directory
                                               for file foo */
getdents(dfd, buf, 1024);/* reads directory entries for
                                 named stream directory */
```

2 使用反向名称查找调用解析流文件路径名。 生成的路径名格式与如下格式类似：

```
/mount_point/file_with_data_stream/./data_stream_file_name
```

指定数据流的命名空间

以 $vxfs: 开头的名称 供将来使用。 创建一个名称以 $vxfs: 开头的数据流 在失败时返回 EINVAL 错误。

其他系统调用中的行为更改

尽管指定数据流目录在命名空间中是隐藏的，但是可以通过 fchdir() 或 fchroot() 调用打开指定数据流目录 inode。没有为指定数据流目录定义某些属性，例如..等。任何访问这些字段的操作均会失败。尝试在指定数据流目录中创建目录、符号链接或设备文件将会失败。 在指定数据流目录或指定数据流 inode 上调用的 VOP_SETATTR() 也会失败。

下列是一种使用 fchdir() 调用读取隐藏目录的替代方法：

```
fd = open(filename, O_RDONLY)
dfd = vxfs_nattr_open(fd, ".", O_RDONLY, mode)
fchdir(dfd);
dirp = opendir(".");
readdir_r(dirp, (struct dirent *)&entry, &result);
```

注意： getcwd(3C) 手册页的使用部分指出，在协调使用 chdir() 调用与 chdir() 调用时，应谨慎运行应用程序。对于某个进程中的所有线程来讲，当前工作目录是全局性的。 如果多个线程调用 chdir() 以更改工作目录，则后续对 getcwd() 的调用可能会产生意想不到的结果。

查询指定数据流

在以下示例中，named_stream_file 文件是通过 API 调用使用 20 个指定数据流创建的。

ls 命令不会显示指定数据流。 创建指定数据流后，将在一个隐藏的目录中对它们进行组织。 例如：

```
# ls -al named_stream_file
-r-xr-lr-x1 root  other   1024   Aug 12 09:49named_stream_file
```

查询指定数据流

■ 使用 getdents() 或 readdir_r() 系统调用查询 named_stream_file 文件的目录内容，该文件包含 20 个指定流文件：

```
Attribute Directory contents for
        /vxfstest1/named_stream_file
0x1ff root other 1K Thu Aug 12 09:49:17 2004 .
0x565 root other 1K Thu Aug 12 09:49:17 2004 ..
0x177 root other 1K Thu Aug 12 09:49:17 2004 stream0
0x177 root other 1K Thu Aug 12 09:49:17 2004 stream1
0x177 root other 1K Thu Aug 12 09:49:17 2004 stream2
```

```
.
.
.
0x177 root other 1K Thu Aug 12 09:49:17 2004 stream17
0x177 root other 1K Thu Aug 12 09:49:17 2004 stream18
0x177 root other 1K Thu Aug 12 09:49:17 2004 stream19
```

应用程序编程接口

指定数据流 API 使用标准系统调用和 VxFS API 调用的组合来利用其功能。

下列是查询指定数据流所使用的伪代码的示例：

```
/* Create and open a file */
if ((fd = open("named_stream_file", O_RDWR | O_CREAT | O_TRUNC,
    mode)) < 0) {
    sprintf(error_buf, "%s, Error Opening File %s ", argv[0],
            filename);
    perror(error_buf);
    exit(-1);
}
/* Write to the regular file as usual */
write(fd, buf, 1024);
/* Create several named data streams for file
   named_stream_file */
for (i = 0; i < 20; i++) {
   sprintf(attrname, "%s%d", "stream", i);
   nfd = vxfs_nattr_open(fd, attrname, O_WRONLY | O_CREAT,
       mode);
   if (nfd < 0) {
      sprintf(error_buf,
         "%s, Error Opening Attribute file %s/./%s ",
         argv[0], filename, attrname);
      perror(error_buf);
      exit(-1);
   }
   /* Write some data to the stream file */
   memset(buf, 0x41 + i, 1024);
   write(nfd, buf, 1024);
   close(nfd);
}
```

命令参考资料

当您使用 cp、tar、ls 或相似的命令复制或列出包含指定数据流的文件时，将会复制或列出该文件，但不会复制附加的指定数据流，也不会将其列出。

注意：Solaris 9 运行环境和更高版本提供了 -@ 选项，可以使用这些命令来指定该选项，以处理这些指定数据流。

5

Veritas File System I/O

本章节包括下列主题：

- 关于 Veritas File System I/O
- 冻结和解冻
- 缓存顾问
- 扩展区

关于 Veritas File System I/O

Veritas File System (VxFS) I/O 控制 VxFS 文件系统上的数据访问。 VxFS API 用于冻结和解冻文件系统、管理缓存顾问以及管理扩展区属性。

注意：与本文档描述的其他 VxFS API 不同，本章描述的 API 在 VxFS 早期版本的所有平台上提供。 通过 VxFS 缓存顾问提供并行 I/O 访问的 API 例外，该 API 在 VxFS 4.1 及更高版本中提供。

冻结和解冻

临时冻结文件系统将会阻塞对文件系统的所有 I/O 操作，然后在文件系统上执行同步。 当前操作完成后，文件系统将与磁盘同步。 冻结文件系统是在卷级别获取文件系统的稳定和一致性映象的必要步骤。

可以获取一致性卷级别文件系统映象，并将其与文件系统快照工具一起使用。 冻结操作可刷新包含脏的元数据和用户数据的文件系统缓存中的所有缓冲区和页面。 在解冻文件系统之前，该操作会挂起文件系统上的所有新活动。

VxFS 提供了应用程序的 ioctl 接口，可以冻结和解冻 VxFS 文件系统。 这些接口是 `VX_FREEZE`、`VX_FREEZE_ALL` 和 `VX_THAW`。

VX_FREEZE ioctl 运行在单一文件系统上。 执行此 ioctl 操作的程序可以冻结指定的文件系统，并阻塞所有尝试对该文件系统的访问，直到该文件系统被解冻为止。只要 VX_FREEZE ioctl 指定的超时值过期，或 VX_THAW ioctl 运行在文件系统上，该文件系统就会解冻。

VX_THAW ioctl 在冻结的文件系统上运行。在冻结超时周期过去之前，它可以用于解冻指定的文件系统。

VX_FREEZE_ALL ioctl 接口可以冻结一个或多个文件系统。 当有多个由冻结操作指定的文件系统时，VX_FREEZE_ALL ioctl 将以原子方式运行。 VxFS 可同时阻塞对多个指定文件系统的访问，并禁止用户执行的写入操作，该操作可以通过一次写入操作修改多个文件系统。 由于 VX_FREEZE_ALL 可以在单一文件系统中使用，因此 VX_FREEZE_ALL 是连接 VX_FREEZE ioctl 的首选接口。

执行 VX_FREEZE 或 VX_FREEZE_ALL ioctls 将会生成一个“干净”的文件系统映像，该文件系统映像可以在从文件系统设备拆分后再装入。为了响应冻结请求，所有修改过的文件系统元数据均会刷新到磁盘，同时在装入拆分的映像之前日志中没有必须被重放的暂停文件系统事务。

可以使用 VX_FREEZE 和 VX_FREEZE_ALL 接口冻结本地装入的文件系统，或本地或远程装入的集群文件系统。

下表显示了 VxFS 发行版中的冻结或解冻兼容性：

表 5-1　　VxFS 发行版中的冻结/解冻兼容性

	VxFS 3.5	VxFS 4.0	VxFS 4.1	VxFS 5.0 及更高版本
VX_FREEZE	本地文件系统	本地文件系统 集群文件系统	本地文件系统 集群文件系统	本地文件系统 集群文件系统
VX_FREEZE_ALL	本地文件系统	本地文件系统	本地文件系统 集群文件系统	本地文件系统 集群文件系统

在冻结文件系统时，应为冻结谨慎地选择一个合理的超时值，以便降低对以文件系统为目标的外部资源的影响。 当冻结文件系统时，用户或系统进程和资源将被阻塞。 如果指定的超时值太大，资源被阻塞的时间将更长。

在文件系统冻结期间，所有从文件系统的根目录获取文件描述符以便与 VX_THAW ioctl 一起使用的尝试，都将由于文件系统的冻结的状态而导致调用进程被阻塞。必须在执行 VX_FREEZE_ALL 或 VX_FREEZE ioctl 之前获取文件描述符。

在超时周期过期之前，请使用 VX_THAW ioctl 解冻通过 VX_FREEZE_ALL ioctl 冻结的文件系统。

编程接口如下：

```
include <sys/fs/vx_ioctl.h>
int              timeout;
int              vxfs_fd;
/*
 * A common mistake is to pass the address of "timeout".
 * Do not pass the address of timeout, as that would be
 * interpreted as a very long timeout period
 */
 if (ioctl(vxfs_fd, VX_FREEZE, timeout))
   {perror("ERROR: File system freeze failed");
}
```

对于多个文件系统：

```
int              vxfs_fd[NUM_FILE_SYSTEMS];
struct           vx_freezeall freeze_info;
freeze_info.num = NUM_FILE_SYSTEMS
freeze_info.timeout = timeout;
freeze_info.fds = &vxfs_fd[0];
if (ioctl(vxfs_fd[0], VX_FREEZE_ALL, &freeze_info))
   {perror("ERROR: File system freeze failed");
}
for (i = 0; i < NUM_FILE_SYSTEMS; i++)
    if (ioctl(vxfs_fd[i], VX_THAW, NULL))
       {perror("ERROR: File system thaw failed");
    }
```

缓存顾问

VxFS 允许应用程序设置缓存顾问，以便在访问文件时使用。 缓存顾问是应用程序访问文件时的首选选择。该选择可能基于通过指定顾问获得最佳性能，或用于确保用户数据的完整性。 例如，数据库应用程序可选择使用直接 I/O 访问包含数据库数据的文件，或者应用程序可通过选择缓冲 I/O 顾问来选择利用文件系统级缓存。 应用程序可选择使用哪个缓存顾问。

要在文件上设置缓存顾问，首先请打开该文件。当请求缓存顾问时，该顾问将记录在内存中。在内存中记录顾问意味着缓存顾问在重新启动或重新装入后不会保留。一些顾问是在“每文件”而不是“每文件-描述符”基础上进行维护。 因此，通过文件描述符设置这样一个顾问会影响其他进程对同一文件的访问。这也意味着相互冲突的顾问实际上不能用于访问同一个文件。 如果两个应用程序设置了不同的顾问，则这两个应用程序都将使用文件上设置的最后一个顾问。 VxFS 不协调顾问也不设定顾问优先级。

在最后一次关闭文件之后，有些顾问没有从内存中清除。顾问记录在内存中保留的时间，与用于管理文件访问的文件系统元数据保留在内存中的时间一样长。何时将文件的文件系统元数据从内存中删除，是无法预测的。

所有顾问都使用文件描述符设置，通过使用 VX_SETCACHE ioctl 命令进行 open() 和 ioctl() 调用返回。

请参见 vxfsio(7) 手册页。

缓存顾问将在以下各节进行介绍。

直接 I/O

直接 I/O 是用于访问文件的无缓冲的 I/O 形式。如果设置了 VX_DIRECT 顾问，用户将请求在磁盘和用户提供的缓存之间进行直接数据传输以执行读取和写入操作。这将绕过数据的内核缓冲，并通过消除内核缓冲区和用户缓冲区之间的数据复制减少与 I/O 相关的 CPU 开销。直接 I/O 也可避免占用可能用作其他用途（如，应用程序缓存）的缓冲区缓存空间。直接 I/O 功能可显著提升某些应用程序的性能。

要使 I/O 操作能作为直接 I/O 执行，必须满足特定的对齐条件。磁盘驱动程序、磁盘控制器以及系统内存管理硬件和软件通常决定对齐约束。文件偏移必须按扇区边界 (DEV_BSIZE) 对齐。所有用户缓冲区必须按一个长的边界或扇区边界对齐。如果文件偏移没有按扇区边界对齐，VxFS 将执行正常的读取或写入，而非直接读取或写入。

如果请求不能满足直接 I/O 的对齐约束，该请求将作为数据同步 I/O 执行。如果通过使用内存映射的I/O访问文件，则任何直接I/O访问都将作为数据同步I/O完成。

由于直接 I/O 维护和同步 I/O 相同的数据完整性，它可用于当前使用同步 I/O 的许多应用程序。如果直接 I/O 请求未分配存储或扩展文件，则不会立即写入 inode 元数据。

直接 I/O 的 CPU 开销与原始磁盘传输相同。对于在巨大文件上执行的顺序 I/O，使用具有较大传输大小的直接 I/O 可提供与缓冲 I/O 相同的速度，但 CPU 开销将更少。

如果扩展文件或分配存储，直接 I/O 必须在返回应用程序之前写入 inode 更改。该写入会抵消直接 I/O 的一些性能优势。

直接 I/O 顾问基于每个文件描述符进行维护。

并行 I/O

并行 I/O (VX_CONCURRENT) 是文件访问的一种 I/O 形式。该形式的 I/O 允许多个进程读取或写入同一文件，而不会阻止其他 read() 或 write() 操作。POSIX 语义要求对文件的 read() 和 write() 操作与其他 read() 和 write() 操作序列化。使用 POSIX 语义，读取操作可以在写入操作前后读取数据。对文件设置 VX_CONCURRENT 顾问后，读写操作的顺序就会不同于字符设备。该顾问通常用于访问数据时需要较高性

能并且不会对同一文件执行重叠写入操作的应用程序。数据库应用程序是一个示例。此类应用程序在应用程序级执行它们自己的锁定以避免重叠写入到文件的同一区域。

在使用 VX_CONCURRENT 顾问时，应用程序或线程负责协调对同一文件的写入活动，从而避免重叠写入。对同一文件进行两次重叠写入的后果是无法预测的。最佳做法是应用程序避免同时向同一文件的同一区域写入数据。

如果文件设置了 VX_CONCURRENT 顾问，VxFS 将针对文件的读写操作执行直接 I/O。同样地，并行 I/O 与直接 I/O 具有相同的对齐要求。

请参见第 68 页的"直接 I/O"。

当启用并行 I/O 时，读取和写入行为将如下进行：

- write() 系统调用将获取共享的读写锁而不是独占锁。
- write() 系统调用对磁盘执行直接 I/O，而不是先复制用户数据然后将其写入系统页缓存的页面中。
- read() 系统调用获取共享的读写锁，并从磁盘执行直接 I/O，而不是先将数据读入到系统页缓存的页面中，然后再将其从这些页面复制到用户缓冲区。
- read() 和 write() 系统调用不是原子操作。应用程序必须确保两个线程不会同时写入文件的同一区域。

可使用带有 VX_CONCURRENT 顾问标志的 VX_SETCACHE ioctl 命令，通过文件描述符和 ioctl() 操作设置并行 I/O (CIO)。仅通过此文件描述符发生的 read() 和 write() 操作使用并行 I/O。通过其他文件描述符发生的 read() 和 write() 操作仍将遵循 POSIX 语义。可通过 VX_SETCACHE ioctl 描述符设置文件的 VX_CONCURRENT顾问。

CIO 是 VxFS 的一个可授权功能。

无缓冲的 I/O

VX_UNBUFFERED 顾问的 I/O 行为与 VX_DIRECT 顾问（设置与直接 I/O 相同的对齐约束）相同。但对于无缓冲的 I/O，如果扩展文件，或者将存储分配给文件，则在写操作返回给用户之前，由于扩展文件而发生的磁盘元数据更新不会同步进行。VX_UNBUFFERED 顾问基于每文件描述符进行维护。

其他顾问

VX_SEQ 顾问是基于每个文件的顾问，指示将按顺序访问文件。 通过文件描述符在文件上设置此顾问的进程，将影响当前访问同一文件的其他进程的访问模式。当读取带有 VX_SEQ 顾问的文件时，将执行最大预读。 当写入带有 VX_SEQ 顾问的文件时，假定按顺序进行写入访问，并且使用写操作修改的页不会立即刷新。修改的页将保留在系统页缓存中，这些页将在当前写入点之后的若干距离点进行刷新（后刷新）。

VX_RANDOM顾问是基于每个文件的顾问，指示将随机访问文件。通过文件描述符在文件上设置此顾问的进程，将影响当前访问同一文件的其他进程的访问模式。此顾问禁用文件读操作的预读，并禁用文件的后刷新。禁用后刷新的结果是，通过最近的写操作修改的系统页缓存中的页将不会刷新到磁盘，直至系统分页进程被调度并运行以刷新脏页。 系统分页进程调度的速率视可用内存和争用的情况而定。

注意：VX_SEQ 和 VX_RANDOM 是互相排斥的顾问。

扩展区

磁盘空间通常被分配为 512 字节或 1024 字节 (DEV_BSIZE) 的扇区以形成逻辑块。 VxFS 支持 1024、2048、4096 和 8192 字节的逻辑块大小。 对于 2 TB 以下的文件系统，默认逻辑块大小为 1K，其他文件系统默认逻辑块大小为 8K。 当用户使用 mkfs 命令创建文件系统时，可选择任何块。VxFS 按由一个或多个相邻块构成的组（称为扩展区 ）向文件分配磁盘空间。 扩展区 是一组一个或多个连续的逻辑块。如果为存储分配的是连续的块，扩展区将允许以多个块为单位执行磁盘 I/O。 对于顺序 I/O，多块操作的速度较之一次一块操作的速度显著加快。

VxFS 使用积极的分配策略向文件分配扩展区。 它还允许应用程序预分配空间或请求连续空间。这将获得改进的I/O性能以及更少的文件系统开销用于进行分配。对于扩展写操作，策略尝试按写操作的大小扩展先前分配的扩展区或扩展至更大。当检测到连续的扩展写操作时，将尝试更大的分配。如果最后的扩展区无法扩展到满足整个写操作的需求，将分配一个新的不连贯扩展区。此策略将产生额外的分配，将在文件最后一次关闭时，或文件在一定时间内未被写入时对其进行修整。文件系统仍可被分段为许多非连续扩展区，特别是较小的文件系统。

扩展区属性

VxFS按由一个或多个扩展区构成的组向文件分配磁盘空间。通常，VxFS内部分配策略尝试实现两个目标： 分配扩展区以获得最佳 I/O 性能以及减少碎片。VxFS 分配策略尝试从文件系统中最适合数据的可用空间中进行分配，通过大的分配和最少的文件系统碎片来平衡这两个目标。这些基于扩展区的分配策略提供了一个相对于基于块的分配策略的优势。基于扩展区的策略很少在分配中使用间接块，并消除了许多源于间接引用的磁盘访问实例。

VxFS 允许通过两个管理工具 setext 和 getext 以及 API 对给定文件的扩展区分配策略的某些方面进行控制。与文件相关联的应用程序强加的策略称为扩展区属性。VxFS 提供 API 以允许应用程序设置或查看与文件相关联的扩展区属性，并向文件预分配空间。

请参见 setext(1) 和 getext(1) 手册页

属性特定

有两个扩展区属性与文件相关联：保留和固定扩展区大小。可以通过控制文件的保留空间来为文件预分配空间。也可以通过设置固定扩展区大小覆盖文件系统的默认分配策略。其他策略确定在分配过程中这些属性的表示方式。

可以指定以下内容：

- 为文件保留的空间必须是连续的
- 在当前保留之外将不向文件进行分配
- 当文件关闭时，释放未使用的保留空间
- 分配空间，但不指派保留空间
- 更改文件大小以立即包含分配的空间

某些扩展区属性是持久性的，并成为关于文件的磁盘上信息的一部分，而其他属性则是临时的，在文件关闭后或系统重新启动后将会丢失。持久性属性与文件的权限类似，写入在文件的 inode 中。当复制，移动或存档文件时，只有源文件的持久性属性会保留在新文件中。

保留：为文件预分配空间

空间保留用于确保应用程序不会因为文件系统空间不足而失败。应用程序在开始任何工作之前，可为其所需的所有文件预分配空间。通过提前分配空间，可最佳化地分配文件以提高性能，并且文件访问不会由于需要分配存储而减速。这种资源的分配在需要有保证的响应时间的应用程序中是很重要的。对于非常大的文件，使用空间保留可避免使用间接扩展区的需求。它还可以通过保证文件包含大的连续扩展区来改进性能和减少碎片。

VxFS 提供 API，能够在请求时即向文件预分配空间，而不是在数据写入文件时才分配空间。预分配（或保留）通过确保在数据写入文件之前文件的所需空间与文件相关联，防止文件系统上出现意外的空间不足情况。可以在任何时间为文件保留存储空间，并且不会将文件的保留空间分配给文件系统的其他文件。API 为应用程序提供选项来更改文件的大小以包含保留空间。

保留过程中不对分配给文件的块进行清零。因此，此功能仅局限于以适当权限运行的应用程序，文件的大小没有通过保留请求进行更改的情况除外。在为文件最近分配的块中出现的数据，可能先前已包含在另一个文件中。

保留是保存在磁盘上的文件的一个持久性属性。文件设置了此属性后，当文件被裁截时，将不会释放此属性。必须通过同一 API 清除保留，否则必须删除文件以释放保留的空间。在指定保留时，如果文件大小比保留量小，将为文件分配空间，使之从当前文件大小增加至保留量。当文件被裁截时，低于保留量的空间将不被释放。

固定扩展区大小

VxFS 使用写请求的 I/O 大小和默认分配策略为文件分配空间。对于某些应用程序，默认分配策略可能不是最佳策略。为文件设置固定扩展区大小将覆盖该文件的默认分配策略。应用程序可设置固定扩展区大小以匹配应用程序的 I/O 大小，这样分配给文件的所有新扩展区即为固定大小。通过使用固定扩展区大小，应用程序可以减少文件的分配尝试并保证最佳扩展区大小。通过固定扩展区大小属性，扩展写操作将触发 VxFS 使用固定扩展区大小量来扩展先前分配的扩展区，以维护扩展区的连续性。如果最后的扩展区无法以固定扩展区大小量来扩展，将分配一个新的不连续扩展区。固定扩展区的大小应该将适合于应用程序的文件 I/O 大小计算在内。不要使用小的固定扩展区大小消除基于扩展区的分配策略的优势。

固定扩展区大小的另一用途是用于稀疏文件。VxFS 通常以系统定义的页大小的倍数执行 I/O。当为稀疏文件进行分配时，VxFS 根据需要分配的页 I/O 量，按页大小的倍数分配空间。如果应用程序总是执行子页 I/O，则会减少按页大小的倍数分配的固定扩展区大小。

应用程序不应使用大的固定扩展区大小。分配大的固定扩展区可能由于该大小的扩展区不可用而失败，而较小的扩展区则更易于获得以用于分配。

自定义应用程序还可能出于特定原因使用固定扩展区大小，例如需要在磁盘上对扩展区进行柱面或条带边界对齐。

固定扩展区大小属性以文件系统块大小为单位指定。属性指定要为新扩展区分配的连续文件系统块的数量，或要为现有扩展区分配并附加到现有扩展区末尾的连续块的数量。拥有此属性的文件具有固定大小扩展区或更大扩展区（为固定大小扩展区的数倍）。

扩展区属性的应用程序编程接口

扩展区属性的当前 API 是 `ioctl()`。 应用程序可打开文件，并使用返回的文件描述符进行 `ioctl()` 调用以检索、设置或更改扩展区属性。 要设置或更改现有扩展区属性，请使用 `VX_SETEXT` **ioctl**。要检索现有扩展区属性（如果有），请使用 `VX_GETEXT` **ioctl**。 应用程序可通过在类型 **vx_ext** 的结构中提供属性信息以及使用 `ioctl()` 调用的第三个参数传递 `VX_SETEXT` **iotcl** 和该结构的地址来设置或更改文件的扩展区属性。 应用程序还可以通过传递 `VX_GETEXT` **ioctl** 以及类型 **vx_ext** 的同一结构的地址作为进行 `ioctl()` 调用的第三个参数，来检索现有扩展区属性（如果有）。

```
struct vx_ext {
    off_t   ext_size;   /* extent size in fs blocks */
    off_t   reserve;    /* space reservation in fs blocks */
    int     a_flags;    /* allocation flags */
}
```

设置 *ext_size* 参数用于指定固定扩展区大小。 固定扩展区大小的值以文件系统块大小为单位进行指定。 请确保在设置固定扩展区大小之前已知文件系统块大小。 如果

不需要固定扩展区大小，则使用零以允许将默认分配策略用于分配扩展区。固定扩展区分配策略在 VX_SETEXTioctl 成功执行之后立即生效。对于已包含间接块的文件将出现例外，在这种情况下固定扩展区策略将不起作用，但文件裁截释放所有当前间接块的情况除外。

可以设置 *reserve* 参数，指定为文件预分配的空间大小。该空间大小以文件系统块大小为单位进行指定。请确保在设置预分配空间大小之前已知文件系统块大小。如果文件已预分配，可使用 VX_SETEXT ioctl 更改其当前保留量。如果指定保留量大于当前保留，文件的分配将增加以匹配新指定的保留量。如果保留量小于当前保留，保留量将减少，且分配将减少至新设置的保留量或当前文件大小。文件预分配要求 root 权限，但文件大小未更改且预分配大小没有增大到超出请求进程的 ulimit 时除外。

请参见 VX_CHGSIZE 标志。

请参见 ulimit(2) 手册页。

分配标志

可通过 VX_SETEXTioctl 指定分配标志以对分配策略进行更多的控制。

分配标志在 vx_ext 结构的 *a_flag* 参数中指定，以确定以下内容：

- 分配是否对齐
- 分配是否连续
- 是否可以超出保留写文件
- 当文件关闭时，是否释放未使用的保留
- 保留是否是文件的持久性属性
- 为文件保留的空间何时将实际成为文件的一部分

用于保留的分配标志

VX_TRIM、VX_NOEXTEND、VX_CHGSIZE、VX_NORESERVE、VX_CONTIGUOUS 和 VX_GROWFILE 标志可用于修改保留请求。VX_NOEXTEND 是唯一持久的标志；其他标志也许具有持久性作用，但是 VX_GETEXTioctl 不返回标志。非持久性标志在文件系统缓存中为文件保持活动，直至该文件不再被访问并从缓存删除。

保留修整

VX_TRIM 标志指定必须修整保留大小，才能匹配最后一次关闭文件时的文件大小。在最后一次关闭时，将清除 VX_TRIM 标志，并释放所有超出文件大小的未使用的保留空间。如果应用程序需要为文件提供足够空间，但不知该文件将有多大，此标志

将十分有用。可保留足够空间来存放最大预期的文件，当写入文件并将其关闭时，将释放所有额外空间。

非持久性保留

如果不想将保留作为持久性属性，可指定 VX_NORESERVE 标志以请求分配空间，而不将保留作为文件的持久性属性。此标志可由对临时保留感兴趣，但当文件关闭时希望释放所有超出文件末尾的空间的应用程序使用。例如，如果应用程序在复制一个长为 1 MB 的文件，该程序可通过 VX_NORESERVE 标志设置，请求 1 MB 的保留。空间被分配，但文件的保留剩下 0。如果程序由于某种原因而中止或系统崩溃，将释放超出文件末尾的未使用的空间。当程序完成时，由于磁盘上未记录保留，将不进行清理。

不进行超出保留的写操作

VX_NOEXTEND 标志指定所有尝试超出当前保留的写操作都会失败。超出当前保留的写操作要求为文件分配新的空间。要为文件分配新的空间，必须增加空间保留。其用法与 ulimit 命令的功能相似，可防止文件使用过多空间。

连续保留

VX_CONTIGUOUS 标志指定分配给文件的所有空间必须满足单个扩展区分配的要求。如果没有一个扩展区足够大以满足保留要求，请求将失败。例如，如果创建了一个文件，且请求了 1 MB 的连续保留，文件大小设置为零且保留设置为 1 MB。 文件将具有一个长为 1 MB 的扩展区。如果发出另一个保留请求，要求 3 MB 的连续保留，新请求将发现前 1 MB 已分配，将分配一个 2 MB 扩展区以满足请求。 如果没有可用的 2 MB 扩展区，请求将失败。 根据定义，扩展区是连续的。 因为 VX_CONTIGUOUS 不是持久性标志，将不会分配连续空间用于还原先前通过 VX_CONTIGUOUS 标志分配的文件。

包含文件大小的保留

通过指定 VX_CHGSIZE，保留请求可影响文件大小以包含保留量。 此标志可增加文件大小以匹配保留量，而无需将保留空间清零。由于此标志作用于文件中未初始化的数据，这些数据可能先前已包含在在其他文件中，此标志的使用仅限于有相应权限的用户。没有此标志，文件将不包含保留空间，直至扩展写操作要求该空间。立即更改文件大小的保留可生成大的临时文件。应用程序可通过消除强加于写操作的开销来分配空间和更新文件大小，进而利用此类保留。

可能组合使用这些标志。例如，使用 VX_CHGSIZE 和 VX_NORESERVE 更改文件大小，但不设置任何保留。 当文件被裁截时，将释放空间。 如果未使用 VX_NORESERVE 标志，保留与文件大小一起在磁盘上设置。

读取文件的增长部分

当分配标志 (*a.flag*) 设置为 VX_GROWFILE 时，文件大小会发生更改以包含保留。 此标志读取文件的“增长”部分（介于文件当前大小和操作成功后的大小之间）。VX_GROWFILE 具有持久性作用，但不显示为分配标志。此标志通过 VX_GETEXTioctl 显示。

用于固定扩展区大小的分配标志

VX_ALIGN 标志可用于为固定扩展区大小指定分配标志。 如果通过保留请求指定标志，则该标志不起任何作用。 VX_ALIGN 标志指定对齐要求，指明在分配未来扩展区时，按与分配单元的起始块相对的固定扩展区大小边界对齐。此标志可用于对扩展区进行磁盘条带边界或物理磁盘边界对齐。 VX_ALIGN 标志是持久的，VX_GETEXTioctl 返回标志。

如何使用扩展区属性 API

首先，验证目标文件系统是否为 VxFS，然后使用 statfs() 调用确定文件系统块的大小。 在大多数平台上，VxFS 的类型为 MNT_VXFS，并在 *statfs.f_bsize* 中返回文件系统块的大小。 通过 VxFS 扩展区属性 API 设置或解释扩展区属性信息必须知道块大小。

VX_SETEXTioctl 的每个调用会影响 vx_ext 结构中的所有元素。

使用 VX_SETEXT

1. 调用 VX_GETEXTioctl 读取当前设置（如果有）。
2. 修改要更改的当前值。
3. 调用 VX_SETEXTioctl 设置新的值。

警告：请认真遵循此过程。当更改保留时，固定扩展区大小可能被无意间清除。当在 VxFS 和非 VxFS 文件系统之间复制文件时，将不能保留扩展区属性。 对于文件系统块大小与源文件系统不同的另一个 VxFS 文件系统，vx_ext 结构中为文件返回的属性值将产生不同的作用。当在具有不同块大小的两个文件系统之间通过属性复制文件时，为不同的块大小转换属性值可能是必要的。

设置固定扩展区大小

以下是一个示例代码段，用于在新文件 MY_FILE 上设置 MY_PREFERRED_EXTSIZE 属性的固定扩展区大小，并假定 MY_PREFFERED_EXTSIZE 是文件系统块大小的倍数。

```
#include <sys/fs/vx_ioctl.h>
struct vx_ext myext;
```

```
fd = open(MY_FILE, O_CREAT, 0644);
myext.ext_size = MY_PREFERRED_EXTSIZE;
myext.reserve = 0;
myext.flags = 0;
error = ioctl(fd, VX_SETEXT, &myext);
```

以下是一个示例代码段，用于在新文件 MY_FILE 上预分配 MY_FILESIZE_IN_BYTES 字节的空间，并假定目标文件系统块大小为 THIS_FS_BLOCKSIZE：

```
#include <sys/fs/vx_ioctl.h>
struct vx_ext myext;
fd = open(MY_FILE, O_CREAT, 0644);
myext.ext_size =0;
myext.reserve = (MY_FILESIZE_IN_BYTES + THIS_FS_BLOCKSIZE)
/THIS_FS_BLOCKSIZE;
myext.flags = VX_CHGSIZE;
error = ioctl(fd, VX_SETEXT, &myext);
```

6

精简回收

本章节包括下列主题：

- 关于精简存储
- 关于精简回收
- 精简回收应用程序编程接口

关于精简存储

精简存储是使用支持精简置备功能的阵列的结果。精简存储是阵列供应商提供的解决方案，只有当应用程序真正需要存储时才从可用存储池中为其分配存储。精简存储解决方案试图解决可用阵列容量利用率低的问题。支持精简存储回收的阵列和 LUN 允许管理员将之前用过的存储释放到可用存储池中。

关于精简回收

有些供应商支持在精简存储阵列执行回收功能，管理员可启动回收进程，从文件系统、LUN或阵列的磁盘上回收空闲存储，以便将此存储释放到可用存储池中。这类阵列和 LUN 被称为精简置备阵列和 LUN。Storage Foundation 精简回收功能通过命令行和编程接口回收空闲块。

注意： Storage Foundation 精简回收功能在 Solaris x64 操作环境中不受支持。

精简回收应用程序编程接口

可通过以下 API 来使用精简回收：

```
uint vxfs_ts_reclaim(char *mountpoint, uint64_t offset,
uint64_t length, int32_t volindex, uint64_t unit_size,
uint64_t *bytes_reclaimed, uint32_t flag)
```

这是一个非重入 API。当 `fsadm` 命令的实例或文件系统的重新组织正在运行时，无法调用该 API。

mountpoint	装入到 Veritas Volume Manager (VxVM) 卷上的 VxFS 文件系统的路径名。
offset	卷中的偏移（单位为字节），将从其开始回收操作。
length	从 *offset* 开始回收空闲存储的长度（单位为字节）。
volindex	卷集中卷的索引。当 *volindex* 的值为 -1 时，此文件系统所有的卷都忽略 *offset* 和 *length*。在非多卷文件系统中，*volindex* 应为零。
unit_size	文件系统向 VxVM 发送回收请求时使用的是该大小的倍数。每个精简置备阵列都支持按某个单位大小的因数进行回收。
bytes_reclaimed	返回文件系统尝试回收的字节数。此值并不表明实际回收的字节数。

flag flag 的可能值为：

- VXFS_TS_RECLAIM_AGGRESSIVE - 执行附加的数据和元数据重新组织以最大限度地回收空闲空间。此操作可能会触发来自基础精简存储的附加空间分配，在操作结束时会释放该存储。此操作可以对现有的大型分配进行分段。
  仅当VxVM将卷报告为thinrclm时，才执行积极的回收。对于多卷文件系统，仅考虑VxVM报告为thinrclm的卷。
  如果 *volindex* 参数的值为 -1，则将涵盖整个文件系统。将忽略 *offset* 和 *length* 参数。如果为 *volindex* 参数指定值，则当 *offset* 参数和 *length* 参数的值都为 0 时，涵盖整个卷。如果为 *offset* 参数和 *length* 参数指定值，则 vxfs_ts_reclaim() 不执行积极的回收，而改为执行默认回收，而不管是否为 *volindex* 参数指定了值。
- VXFS_TS_RECLAIM_ANALYSE | VXFS_TS_RECLAIM_ANALYZE - 对文件系统执行分析以建议您应使用正常的回收还是积极的回收。您可以不考虑建议使用任一回收策略。仅当 Veritas Volume Manager (VxVM) 将卷报告为 thinrclm 时，才执行分析回收。在多卷文件系统的情况下，vxfs_ts_reclaim() API 仅考虑 VxVM 报告为 thinrclm 的卷。
- VXFS_TS_RECLAIM_AUTO - 对文件系统执行分析，确定适用此时的回收策略，并根据分析以用户的名义执行该策略。仅当 VxVM 将卷报告为 thinrclm 时，才执行自动回收。在多卷文件系统的情况下，vxfs_ts_reclaim() API 仅考虑 VxVM 报告为 thinrclm 的卷。

VxFS 会在内部对齐 *offset* 和 *length* 以进行校正。

注意：精简回收是一个缓慢的过程，根据文件系统的大小，可能需要花几个小时的时间才能完成。精简回收不保证回收 100% 的空闲空间。

---

vxfs_ts_reclaim 返回值

API 的返回值如下：

-3	如果 vxfs_ts_reclaim() API 的 analyze 选项建议积极回收，将返回此值。
-2	如果 vxfs_ts_reclaim() API 的 analyze 选项建议正常回收，将返回此值。
0	成功回收。

EPERM	非 root 用户调用此 API。
ENOENT	装入点不存在。
EBADF	装入点不是文件系统的根。
EPIPE	不能访问装入点的设备。
ENOTSUP	装入点不是 VxFS 文件系统的根，指定的卷不支持 `thinrclm`，或者卷集中没有一个卷支持 `thinrclm`。
ENOMEM	不能在 API 操作期间分配内存。
EBUSY	不能修改文件系统的元数据。
ENXIO	指定的设备不存在。
EINVAL	指定的偏移或长度无效。
EAGAIN	另一 `fsadm` 命令实例或重新组织实例正在运行。
EFAULT	由于其他系统相关问题回收操作失败。

索引

B

保留 71
备用命名空间 57
编译环境 14
并行 I/O 68

C

close 18
存储检查点 48

D

DEV_BSIZE 68, 70
冻结/解冻 65
多卷支持 12, 47
- 查询文件系统的卷集 49
- 查询已定义的策略 52
- 创建并指派策略 52
- 对文件强制执行策略 53
- 分配策略 API 50
- 卷 API 48
- 卷封装 50
- 卷集操作示例 48
- 数据结构 53
- 修改文件系统内的卷 49
- 用途 48

F

fchdir 61
fchroot 61
fcl_keeptime 24
fcl_maxalloc 24
fcl_winterval 24
FSAP_INHERIT 50
fsapadm 48
fsvoladm 48
反向路径名称查找 44
分配标志 73
分配策略 47
- 多卷支持 50

G

getcwd 61
getdents 58, 60
getext 70
固定扩展区大小 71–72

H

缓存顾问 67

I

I/O
- 顺序 68
- 同步 68
- 直接 68

ioctl 11, 68, 72

J

记录类型 22
- 特殊记录 23

精简存储 77
精简回收 77
卷 API 48
卷集 48

K

库 13
扩展区 70
扩展区属性 70

L

lseek 18
逻辑块 70

M

mkfs 70
mmap 58
命名属性 57

N
ncheck 45

O
open 18, 57–58, 68

Q
其他顾问 69

R
read 18, 58, 69
readdir 59
软件开发人员工具包 11
    软件包 13

S
setext 70
statfs 75
使用扩展区属性 API 75
数据传输 68
数据副本 68
顺序 I/O 68, 70

T
特殊记录 23
同步 I/O 68
头文件 13

U
ulimit 73

V
VOP_SETATTR 61
VRTSfssdk 13
VX_ALIGN 75
VX_CHGSIZE 73–74
VX_CONCURRENT 69
VX_CONTIGUOUS 73–74
VX_DIRECT 69
vx_ext 72, 75
VX_FREEZE 65–66
VX_FREEZE_ALL 65–66
VX_GETEXT 72, 75
VX_NOEXTEND 73–74
VX_NORESERVE 73–74
VX_RANDOM 70
VX_SEQ 69
VX_SETCACHE 69
VX_SETEXT 72, 75
VX_THAW 65
VX_TRIM 73–74
VX_UNBUFFERED 69
VxFS I/O 13
    冻结/解冻 65
    缓存顾问 67
        并行 I/O 68
        其他顾问 69
        无缓冲的 I/O 69
        直接 I/O 68
    扩展区 70
        API 72
        保留 71
        分配标志 73
        固定扩展区大小 72
        扩展区属性 70
        使用扩展区属性 API 75
        属性特定 71
        用于固定扩展区大小的分配标志 75
vxfs_inotopath 45
vxfs_inotopath_gen 44
vxfs_nattr_link 59
vxfs_nattr_open 59
vxfs_nattr_rename 59
vxfs_nattr_unlink 59
vxfs_nattr_utimes 60
vxfsio 68
vxtunefs 24
vxvset 48

W
文件更改日志 12, 17
    超级块 24
    记录类型 22
        特殊记录 23
    可调参数 24
    应用程序编程接口 26
文件更改日志文件 17
无缓冲的 I/O 69

X
写入 58, 69

Y
应用程序接口 11

用于固定扩展区大小的分配标志 75

Z

直接 I/O 68
直接数据传输 68
指定数据流 57
    程序员的参考资料 63
    列表 60
    命名空间 60
    其他系统调用中的行为更改 61
    示例 61
    应用程序编程接口 58, 62