文件系统模块

RocketOS 的文件系统模块构建在一套抽象统一、模块解耦的 VFS 框架之上，支持挂载多个后端文件系统（如 Ext4、Tmpfs、Devfs 等），并通过统一的 Dentry、Inode、File 抽象，协调路径解析、权限控制与文件访问等操作流程。下图展示了 RocketOS 文件系统在内核中的总体架构，体现了从系统调用到具体后端文件系统的协作流程。

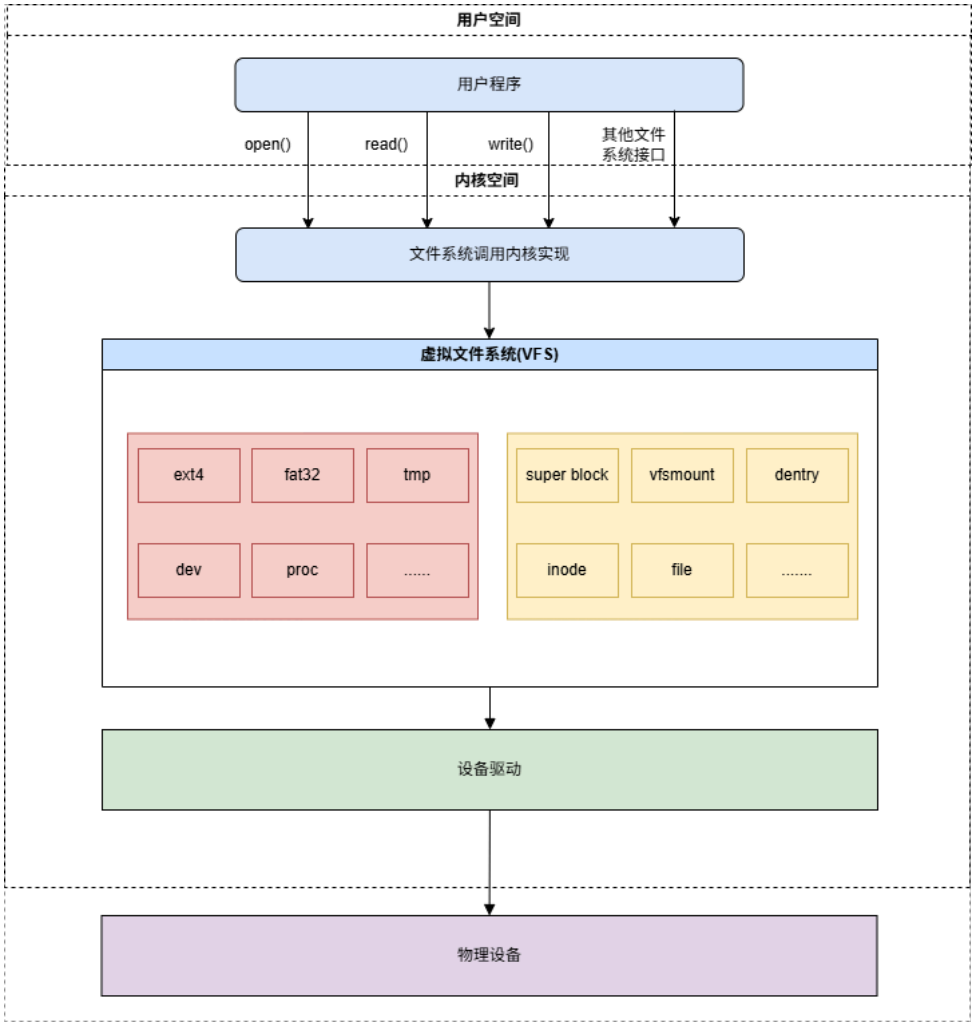

图 1: 文件系统总体框架

挂载子系统

RocketOS 的挂载子系统由三类核心结构组成：`VfsMount` 表示一次具体的文件系统挂载，`Mount` 表示一个挂载点的元数据，`MountTree` 维护所有挂载关系的全局视图。三者协同构成一棵挂载树，实现文件系统命名空间的统一视图。

`VfsMount` 封装了文件系统挂载的核心信息，包括挂载点的根目录、挂载的文件系统（超级块）以及挂载标志。`Mount` 则表示一个具体的挂载点及其拓扑位置， `mountpoint` 表示挂载操作发生的目录项，`vfs_mount` 指向被挂载的文件系统实例，parent 指向父挂载点，children 存储子挂载点。`MountTree` 则是一个全局的挂载点列表，维护所有 `Mount` 对象并提供必要的插入与查询接口。该结点通过全局静态变量 `MOUNT_TREE` 暴露，具备线程安全性。

```rust
pub struct VfsMount {
    root: Arc<Dentry>,              // 挂载点的根目录
    fs: Arc<dyn FileSystem>,        // 挂载的文件系统(超级块)
    flags: i32,                     // 挂载标志
}
pub struct Mount {
    mountpoint: Arc<Dentry>,        // 挂载点目录
    vfs_mount : Arc<VfsMount>,      // 挂载的文件系统
    parent: Option<Weak<Mount>>,    // 父挂载点
    children: Vec<Arc<Mount>>,      // 子挂载点
}
struct MountTree {
    mount_table: Vec<Arc<Mount>>, // 挂载点列表
}
```

代码 1: 挂载命名空间和挂载树

根文件系统初始化流程

系统启动时，通过 do_ext4_mount 函数挂载根文件系统。该函数执行以下步骤：

1. 打开指定的块设备, 并初始化 `Ext4FileSystem` 实例；
2. 从根块组构造根目录对应的 Inode；
3. 创建根目录的 Dentry，设置自身为其父项以形成闭环；
4. 构建 VfsMount 和对应的 Mount；
5. 将根挂载点加入全局 MountTree；
6. 初始化 /dev, /proc, /tmp 等虚拟文件系统。

VFS 模块设计

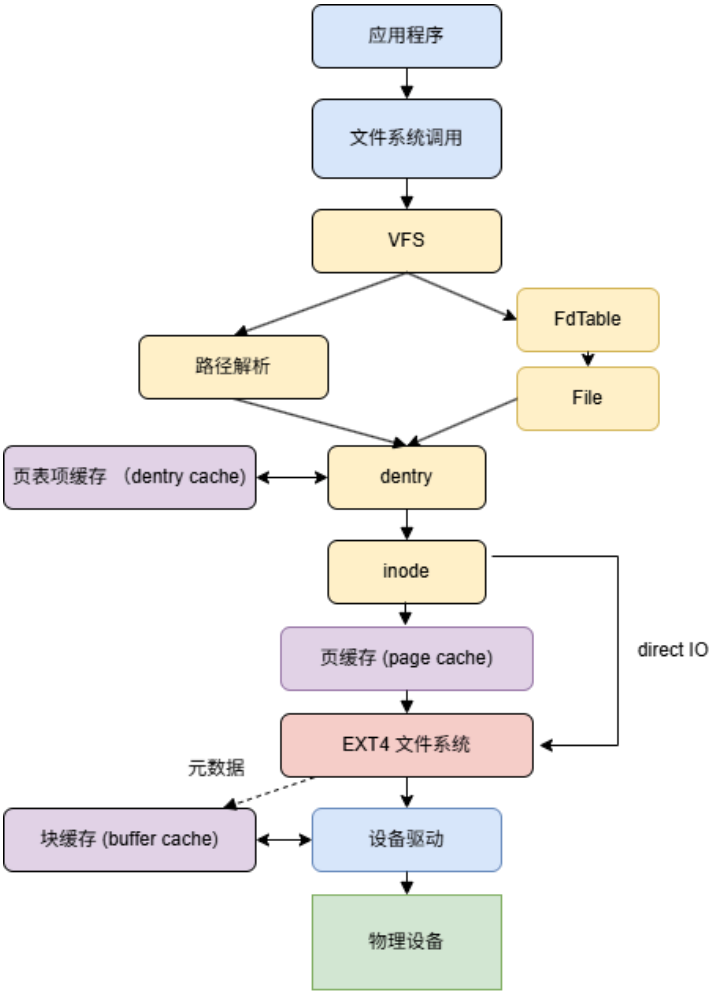

图 2: VFS 层结构与核心组件关系

如上图所示，RocketOS 的 VFS 层以 Inode、Dentry、File 为核心抽象，通过 MountTree 管理挂载关系，形成统一的文件命名空间。该设计借鉴了 Linux VFS 架构，同时引入 Rust 类型系统以强化安全性与并发管理，支持多个后端文件系统的透明接入。

Inode 与页缓存

在 RocketOS 中，`inode` 抽象的核心体现在 `InodeOp` trait 接口之中。该接口定义了文件与目录的基本操作语义，包括读写、截断、页缓存交互、目录项查找与创建、符号链接、权限修改、文件状态查询等。每个具体的文件系统（如 Ext4）通过实现 `InodeOp` 接口，使其能以统一方式与 VFS 层交互，从而实现跨文件系统的透明访问能力。

为了支持运行时的类型动态调度与多文件系统扩展，`InodeOp` 被设计为 trait 对象（`dyn InodeOp`），并通过 `Arc<dyn InodeOp>` 进行引用计数与共享，确保线程安全与生命周期管理。

```rust
pub trait InodeOp: Any + Send + Sync {
  fn read(&self, offset: usize, buf: &mut [u8]) -> usize;
  fn write(&self, offset: usize, buf: &[u8]) -> usize;
  fn get_page(&self, page_index: usize) -> Option<Arc<Page>>;
  fn truncate(&self, size: usize) -> SyscallRet;
  fn fsync(&self) -> SyscallRet;
  fn lookup(&self, name: &str, parent: Arc<Dentry>) -> Arc<Dentry>;
  fn create(&self, dentry: Arc<Dentry>, mode: u16);
  ...
}
```

代码 2: InodeOp

与其他组件的协作

1. 页缓存与块映射
   - InodeOp 提供了 get_page 和 lookup_extent 等接口，用于访问页缓存与磁盘块映射逻辑。该设计体现了块设备抽象与页缓存机制的耦合：
     - get_page: 返回指定页索引对应的缓存页，如无缓存则从磁盘加载或分配新页。
     - lookup_extent: 查找给定页是否被分配块，并返回物理块号区间。
     - write_dio: 支持绕过页缓存的直接写入（Direct IO）。
2. 目录与路径
   - lookup, create, unlink, rename, mkdir, symlink, link 提供了标准 POSIX 文件系统语义的目录操作。lookup 的语义约定如下：
     - 若目标存在，则返回包含有效 inode 的 dentry；
     - 若目标不存在，则返回负目录项（inode = None）；
     - 上层调用者负责缓存管理（如放入 dentry cache）。
3. 临时与特殊文件支持
   - tmpfile: 创建匿名临时 inode，不注册 dentry，适用于 tmpfs 等场景。
   - mknod: 创建字符/块设备文件，支持设置设备号（dev_t）。
4. 属性访问与修改
   - 类型与权限：get_mode, set_mode, set_perm
   - UID/GID：get_uid, set_uid, ...
   - 时间戳：get_atime, set_mtime, ...
   - 文件大小与页数：get_size, get_resident_page_count

具体实现

以 Ext4 为例，Ext4Inode 实现了 InodeOp trait，并结合地址空间与块设备访问能力，具有如下核心特性

- 页缓存集成与统一地址空间管理： `Ext4Inode` 内含的 `address_space` 字段管理与 inode 关联的所有内存页缓存。该地址空间支持以下功能：
  - 按页粒度加载磁盘数据，实现懒加载与访问局部性优化；
  - 页面替换与回写策略的挂接点；
  - 支持透明地将逻辑页索引映射到物理块（通过 extent tree）；
  - 提供 Direct IO（`write_dio`）能力以绕过缓存写入，适用于性能敏感路径。
- 块设备访问桥接能力： 结构中的 `block_device: Arc<dyn BlockDevice>` 提供了统一的块设备访问通道，使 inode 在页缓存失效、回写等场景中可以直接触发块级 IO 请求，从而与存储设备紧密耦合。

- 文件系统耦合与元信息结构共享： 借助 `ext4_fs: Weak<Ext4FileSystem>` 字段，`Ext4Inode` 能够访问所属文件系统的共享信息，如块组描述符、inode bitmap、数据块分配器等。该引用也是 inode 实现 extent tree 查找、块分配等操作的基础。

```rust
pub struct Ext4Inode {
    pub ext4_fs: Weak<Ext4FileSystem>,
    pub block_device: Arc<dyn BlockDevice>,
    pub address_space: Mutex<AddressSpace>,
    pub inode_num: usize,
    pub link: RwLock<Option<String>>,
    pub inner: FSMutex<Ext4InodeInner>,
    pub self_weak: Weak<Self>,
}
```

代码 3: Ext4Inode

RocketOS 中的 `Inode` 抽象设计充分借鉴了 Linux 虚拟文件系统（VFS）架构，同时结合 Rust 的语言特性与现代内核设计理念，形成了清晰、模块化、并具备良好扩展性的实现框架。通过 `InodeOp` trait 明确定义 inode 的操作语义，使具体文件系统（如 Ext4）可以在不侵入 VFS 核心的前提下实现定制逻辑，从而支持多种后端并存。通过 `address_space` 与 `block_device` 字段，`Inode` 成为连接缓存管理层与块存储层的重要桥梁，便于实现回写策略、直接 IO、预读优化等机制。

Dentry 与目录树

Dentry（目录项）是文件系统中用于表示目录结构的核心数据结构。它将文件名与对应的 Inode 关联起来，形成一个树形结构。Dentry 的主要作用是加速文件路径解析和目录项查找。

```rust
pub struct Dentry {
    pub absolute_path: String,
    pub flags: RwLock<DentryFlags>,
    inner: Mutex<DentryInner>,
}

pub struct DentryInner {
    inode: Option<Arc<dyn InodeOp>>,
    parent: Option<Arc<Dentry>>,
    children: HashMap<String, Weak<Dentry>>,
}
```

代码 4: Dentry

在路径解析过程中，文件系统需要将用户输入的路径字符串（如 /usr/bin/bash）逐级解析为实际的 inode 对象。这一解析过程中的每一级目录组件（如 usr、bin）都会对应一个 Dentry 实例，从而构成路径节点链表（Dentry chain）。该链条从根目录向下逐级查找，通过每级 Dentry 提供的父子关联关系，结合 children 完成快速定位。

为提高路径查找的性能，RocketOS 在 VFS 层引入 DentryCache。该缓存以路径字符串为键，存储解析过的 Dentry 对象，并通过引用计数与负目录项机制（用于标识不存在的路径），标识路径查找失败的目录项，用于避免重复错误查找，提升路径解析性能。在绝大多数文件访问场景中，路径查找都可在 DENTRY_CACHE 命中，大幅减少 inode 查找开销。

同时为了确保 VFS 操作的安全性与合规性，Dentry 提供了一套完整的权限检查逻辑：

- dentry_check_access：依据当前任务用户 ID（UID）与组 ID（GID）判断是否具备指定的读、写、执行权限；

- dentry_check_open：结合 OpenFlags 检查文件类型兼容性、写权限、是否允许截断、是否允许创建等；
- chown：实现 POSIX 要求的 chown 逻辑，支持 root 修改权限与 setuid/setgid 位逻辑处理。

RocketOS 的 Dentry 模块是其 VFS 层的关键组成，旨在高效管理路径命名空间、提升路径查找性能并实现访问控制。其设计充分借鉴了 Linux VFS 架构，在保持功能通用性的同时融合 Rust 的类型安全与并发管理优势，达到了良好的系统可维护性、扩展性与运行效率。通过引入 DentryCache、负目录项、引用计数与类型标志机制，RocketOS 的路径解析框架具备现代操作系统所需的高性能路径解析能力。

FileOp 与文件

在 RocketOS 的虚拟文件系统（VFS）框架中，File 结构体承担着进程级别的打开文件表示，是系统中从用户态调用（如 read, write, lseek, ioctl 等）到内核态 inode 操作之间的关键桥接抽象。其设计参考了 Linux 中 struct file 的语义，在类型安全与并发访问控制方面结合了 Rust 的所有权模型与同步原语。每一个打开的文件描述符在内核中都映射到一个唯一的 File 对象，封装了文件状态、访问语义与当前偏移量。

File 结构的核心包括一个受互斥锁保护的 FileInner 成员，后者保存了当前访问的 inode 引用、打开路径对应的 Path 结构体、打开标志（OpenFlags），以及动态维护的读写偏移量（offset）。该偏移量以字节为单位，反映了当前进程视角下对该文件的访问位置。在具有 O_APPEND 标志的场景下，偏移量初始化时即被设置为文件大小，以模拟追加写入的语义行为；否则偏移量初始化为零。

RocketOS 中的 File 实现了 FileOp trait，该 trait 统一了所有类型文件所需支持的操作接口，如 read, write, pread, pwrite, seek, truncate, fallocate 等。针对不同语义特征的文件类型（如常规文件、目录、字符设备、命名管道等），系统可通过实现该 trait 的多态对象（trait object）以适配各自行为。对于常规文件的实现而言，read 和 write 操作均基于当前偏移量进行，而 pread 与 pwrite 提供了带偏移量的无状态访问能力。此种抽象设计允许内核模块在实现如文件映射、direct I/O（write_dio）、文件同步（fsync）等功能时，不依赖具体文件类型，从而提升了模块解耦性。

File 抽象在路径语义上也具备重要角色。每个 File 持有其打开路径对应的 Path 引用，而 Path 内部又可追溯至打开文件的 Dentry。通过此设计，File 可在运行时访问其路径相关元信息，如判断是否为目录（通过 is_dir 查询 Dentry 类型标志），执行目录遍历（如 readdir）等。此外，系统也提供了权限检查函数（如 readable, writable），用于根据打开标志与 O_PATH 模式等判断用户对文件的操作权限。

偏移量管理亦为 File 抽象中一项关键机制。除基本的 get_offset 与 add_offset 操作外，系统支持标准化的文件指针移动接口 seek，并扩展支持 `SEEK_DATA` 与 `SEEK_HOLE` 等现代语义，便于处理稀疏文件。在这类语义下，RocketOS 通过 inode 提供的页级映射信息（如 lookup_extent）判断实际存在的数据区域或空洞位置，从而实现对非连续数据布局的有效识别。

综合来看，File 抽象在 RocketOS 中不仅仅是打开文件的简单句柄，更是一个承载访问语义、权限标志、偏移控制与路径绑定的中间层，其设计显著提高了文件系统的通用性与模块化程度。通过与 inode、dentry、path 等子系统的紧密协作，File 为内核提供了一个精确、可扩展且高度类型安全的文件访问模型，适配多种文件类型与访问模式，为构建现代类 UNIX 操作系统内核的文件语义奠定了坚实基础。

总结

RocketOS 的文件系统模块在设计上深度借鉴了 Linux VFS 架构，通过 Dentry、InodeOp、File 等抽象结构，实现了统一而灵活的文件访问语义。在挂载子系统方面，VfsMount、Mount

与 MountTree 协同构建全局命名空间，使多个后端文件系统得以共存，并通过路径解析机制高效完成用户请求的映射。在访问路径中，Dentry 提供了结构化的目录项管理与缓存机制，有效提升查找性能；而 InodeOp 则定义了文件系统核心操作接口，使具体文件系统如 Ext4 可在不修改 VFS 核心的前提下平滑接入。在此基础上，File 抽象则作为用户态系统调用与内核态文件操作之间的桥梁，承担偏移管理、访问控制与文件类型多态分发等职责。

整个文件系统框架强调模块解耦、抽象统一与类型安全，充分利用 Rust 所提供的并发与内存管理特性，在确保安全性的同时提升了系统可维护性与扩展性。通过这一体系，RocketOS 构建起一套具有现代操作系统特征的文件系统模型，为后续支持设备文件、网络文件系统、文件权限模型与高性能 I/O 提供了坚实基础。