第 7 章 文件系统

各种外部存储介质，如磁盘（分区）、U 盘等，在使用前需要对其进行格式化，格式化后才能用于保存文件。格式化就是按照某一文件系统类型的要求，对存储介质（分区）进行初始化。

文件系统类型定义了文件在存储介质中的保存形式，例如：Windows 系统中的 FAT32、NTFS 文件系统类型等，Linux 系统中的 ext2、ext3、ext4 文件系统类型等。文件系统是文件系统类型的一个实例，可以认为是按文件系统类型要求格式化的存储介质（分区）。

各种文件系统类型虽然组织形式各不相同，但是其功能都是用于保存文件。文件系统中通过目录项对文件进行分层次管理，每个普通目录项及文件在文件系统都有一个目录项，文件系统中目录项构成层次的树状结构。

内核定义了虚拟文件系统（VFS），其中包含一个由目录项组成的单一根文件系统。具体文件系统通过挂载操作关联到根文件系统中某个目录项，将具体文件系统的内容导出到此目录项下。

Linux 内核支持几十种文件系统类型，VFS 抽象出所有文件系统类型的共性部分，定义了统一的结构表示文件系统、目录项和文件等，定义了统一的文件系统、目录项、文件等操作接口，具体文件系统类型需要实现这些接口。VFS 通过这些接口操作文件系统、目录项和文件。

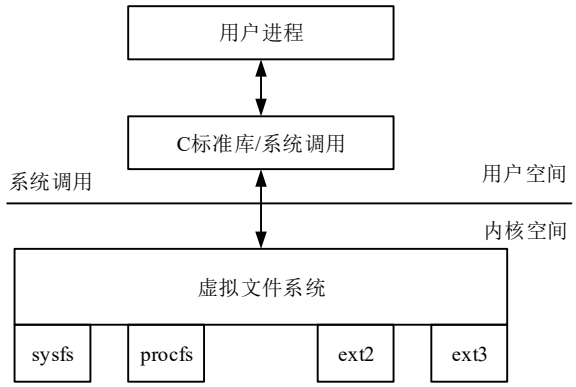
内核根文件系统是内核管理所有文件的一个结构，进程在对文件进行操作前，需要建立进程与根文件系统中文件的关联（打开文件），建立关联后，进程通过一个文件描述符（整数）标识文件，随后进程可通过 VFS 中定义的文件操作接口对文件进行操作。

Linux 内核奉行“万物皆文件”的哲学，外部设备、网络、系统控制参数、进程间通信对象等都被视为文件，内核都以文件操作的形式对它们进行访问。

本章首先介绍虚拟文件系统的组成和实现、文件系统的挂载、根文件系统的挂载、进程文件/目录的操作等，然后介绍几种常见具体文件系统类型的实现。

7.1 虚拟文件系统

本章所说的文件系统包括内核中的虚拟文件系统和实际用于存放文件的具体文件系统。具体文件系统包括用于块设备的文件系统，如：ext2、ext3、ext4 等，以及用于内存或内核数据结构实例的文件系统（非持久文件系统），如：ramfs、proc、sysfs 文件系统等。虚拟文件系统与具体文件系统的关系如下图所示：

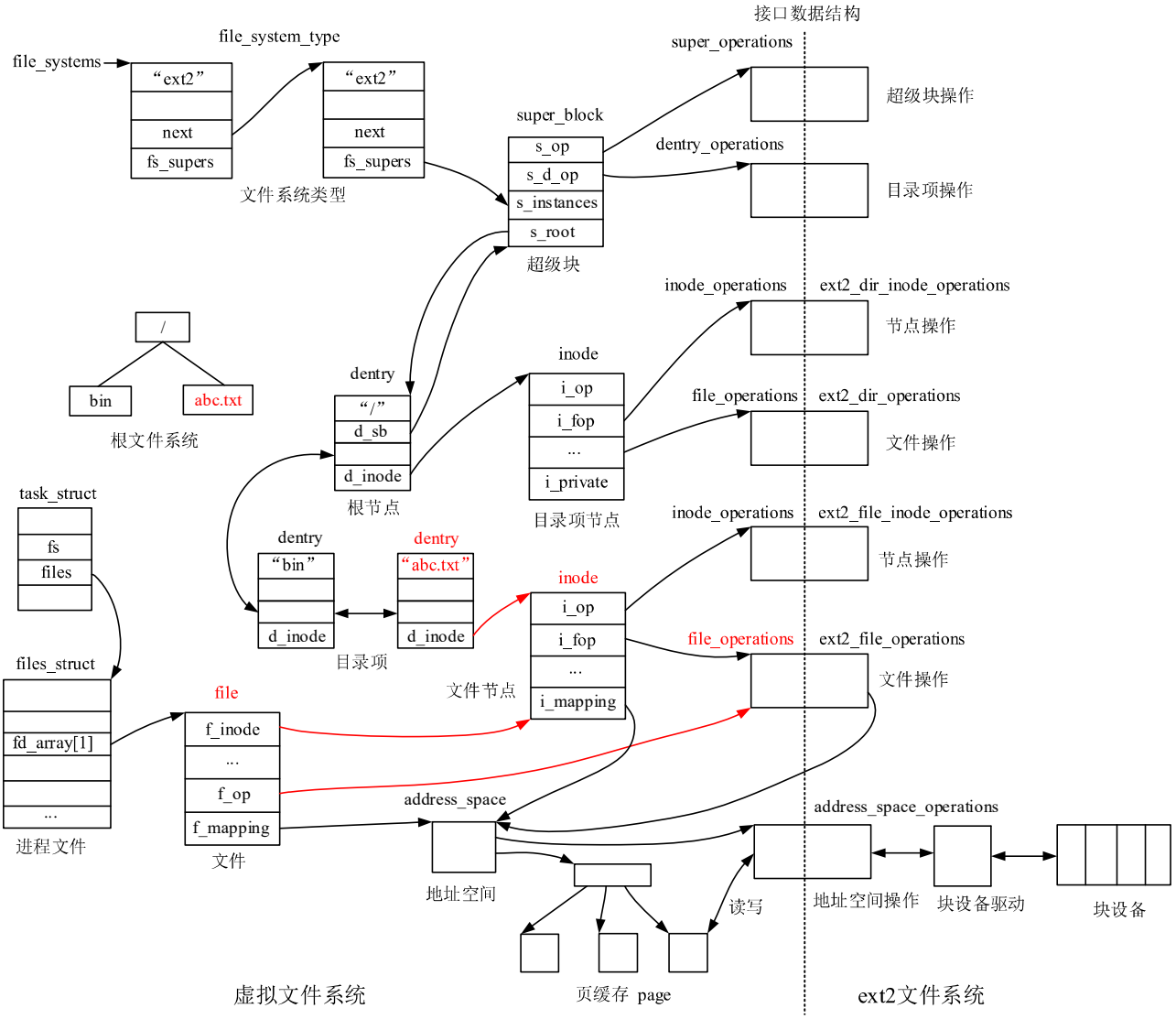


用户进程通过库函数访问虚拟文件系统中的文件，库函数最终通过系统调用访问文件。对于用户进程来说不必关心文件保存在什么类型的文件系统、什么介质中，因为虚拟文件系统采用相同的接口操作所有的文件，虚拟文件系统向用户屏蔽了不同文件系统操作的细节。

内核还定义了伪文件系统，用于管理系统资源，如用于块设备的 bdev 文件系统，伪文件系统是不能挂载到根文件系统中的文件系统，只对内核可见，对用户进程不可见。本章暂不介绍伪文件系统，在介绍使用伪文件系统的内核子系统时再做介绍。

虚拟文件系统的结构如下图所示，每个编译入内核（加载模块）的文件系统类型在内核中由 file_system_type 结构体实例表示，所有的实例在内核中组成单链表，表头为 file_systems。

根文件系统是由目录项 dentry 结构体实例组成的层次树状结构，用于管理所有内核打开的文件系统及文件。dentry 结构体中包含目录项的名称（普通目录项名称或文件名称），并构成父子、兄弟的层次关系。树状结构中具有唯一的根节点，其名称为“/”。每个 dentry 实例关联一个 inode 结构体实例，inode 结构体实例表示目录项对应的文件。不只是表示文件的目录项具有 inode 实例，表示普通目录项的 dentry 实例也具有 inode 实例，因为目录项也是文件，只不过目录项文件的内容是目录项，即此目录下的子目录项和文件目录项。



inode 结构体（节点）用于在 VFS 中唯一地表示一个文件，结构体中包含了文件元信息、缓存文件内容的文件地址空间指针、节点操作结构指针、文件操作结构指针等成员。其中节点操作结构 inode_operations 和文件操作结构 file_operations 是两个非常重要的数据结构，主要包含对文件元信息及文件内容执行操作的函数指针，具体函数由具体文件系统类型实现。即使对于同一种文件系统类型，目录项文件和普通文件的 inode_operations 和 file_operations 实例也是不同的，因为这两种文件的类型是不同的。

具体文件系统（硬盘、分区中文件系统等）在使用前需要挂载到根文件系统某一目录项下，内核通过此目录项进入此文件系统。挂载操作会创建文件系统的超级块 super_block 结构体实例，添加到文件系统类型链表中，创建表示此文件系统根目录项的 dentry 和 inode 实例。超级块 super_block 结构体表示整个文件

系统的组织结构信息，而不是单个文件的信息。

用户进程对文件进行操作前需要执行打开操作，打开操作即依次遍历文件路径中的目录项分量，在根文件系统中查找各目录项对应的 dentry 实例。如果目录项尚不存在，则在父目录项对应文件内容中查找目录项，并在根文件系统中为其创建 dentry 和 inode 实例。当在搜索路径中遇到挂载点时（挂载了文件系统的目录项），将会跳转至挂载文件系统的根目录项下继续往下搜索。

路径名中最末尾的分量为文件名，进程打开的文件由 file 结构体实例表示，file 实例建立进程与最末尾目录项（文件目录项）对应节点 inode 实例之间的关联，如上图中打开的 abc.txt 文件所示。随后，进程就可以通过 inode 实例指向的 inode_operations 和 file_operations 实例中的函数对文件进行操作。

对于保存在外部块设备中的文件系统，其文件内容一般在内存中建立缓存（页缓存）由 inode 实例中的地址空间成员管理，进程对文件内容的操作转换成对页缓存的操作（复制或写入数据），页缓存中的数据在适当的时候由内核通过块设备驱动程序与块设备同步。用户进程也可以不通过页缓存，直接操作块设备中的文件内容（效率较低）。

本章后面小节将详细介绍虚拟文件系统中各组成部分的实现，页缓存相关的内容到第 11 章再做介绍。

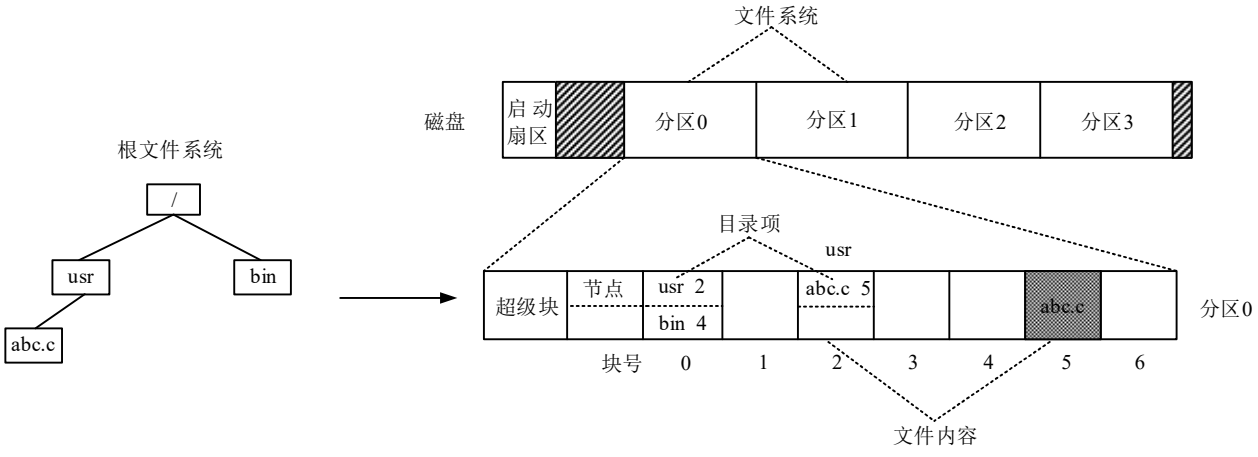
7.2 目录项

从根文件系统某个目录项至另一个目录项的路径，称为目录。例如：/mnt/abc.txt，表示一个目录。而目录中每个分量，如 mnt、abc.txt 等称目录项。

VFS 中目录项是对具体文件系统中的普通目录项和文件目录项的抽象。目录项在虚拟文件系统中由 dentry 结构体表示，dentry 实例构成了根文件系统的树状层次结构。在打开文件时会创建路径中各目录项对应的 dentry 实例和 inode 实例。

7.2.1 物理结构

文件在块设备（分区）文件系统中保存，通常分为保存文件名称的目录项和保存文件内容的数据块，文件名称保存在父目录项的文件内容中。下图示意了分区中文件系统的物理结构，注意这只是一个示意，用于说明文件系统的结构，并不是一个真实的文件系统。



上图中磁盘被分为 4 个分区，磁盘中第一个扇区（启动扇区）保存有一个分区表，表示每个分区的起始扇区号，我们以分区 0 为例说明文件系统的结构。格式化分区 0 时，将按块（包含若干扇区）对分区进行划分，分区开头是超级块，保存整个文件系统的信息。数据块 0 保存的是文件系统根目录项文件的内容，即根目录项下的子目录项（含文件目录项）。目录项包含名称字符串和存放文件内容的数据块号等信息。

如上图中所示，根目录下包含 usr 和 bin 子目录项，usr 子目录项的文件内容保存在数据块 2 中，bin

子目录项文件内容保存在块 4 中。由于 usr 为普通目录项，所以其文件内容保存的依然是目录项，由其文件内容（块 2 中数据）可知 usr 目录项下存在名称为 abc.c 的文件，其文件内容保存在块 5 中。由此文件系统的组织结构可提取出如图中左侧所示的目录项层次结构。

将分区 0 文件系统导入根文件系统时，每个目录项由 dentry 结构体实例表示，根目录项实例在挂载文件系统时创建，名称赋值为“/”，表示是挂载文件系统的根目录项。假设分区 0 被挂载为内核根文件系统，内核在打开/usr/abc.c 文件时，需要查找 3 个目录项，分别是“/”、“usr”和“abc.c”。内核将从文件系统根目录项“/”开始，依次查找 usr 和 abc.c 路径分量对应的目录项，如果目录项 dentry 实例尚不存在，则到分区 0 文件系统下查找，找到分区中的各目录项，提取其中的信息，创建并填充 dentry 和 inode 结构体实例，并添加到内核根文件系统中。

实际的文件系统要复杂一些，并稍有区别。例如，在 ext2 文件系统中，在分区中的超级块之后划分出了一个区域用于保存节点。节点中保存了文件的元信息和保存文件内容的数据块号等信息，此节点导出 VFS 时，由 inode 结构体表示。目录项中并没有存放保存文件内容的数据块号，而仅保存了表示对应节点的编号，在节点中保存了文件内容数据块号等信息。在上图中为了简化，直接把保存文件内容的块号写在目录项中，实际目录项中保存的是节编号，节点中保存了文件内容数据块号。

目录项除了能表示普通的目录和文件外，还可以表示链接。链接用于将本目录项关联到其它文件，链接分为符号链接和硬链接。如果目录项表示的是符号链接，则目录项文件内容保存的是链接文件的路径名。即符号链接的文件内容是另一个文件路径。硬链接是指多个目录项指向同一个文件，即目录项中保存的文件节点编号相同。

7.2.2 数据结构

在虚拟文件系统中目录项由 dentry 结构体表示，目录项操作由 dentry_operations 结构体表示。

1 dentry

内核在打开文件时，将在具体文件系统中查找各目录项分量，提取其中信息，创建 denty 实例，并将实例添加到根文件系统树状层次结构中。

dentry 结构体定义在/include/linux/dcache.h 头文件：

```
struct dentry {
    unsigned int d_flags;                   /*目录项标志*/
    seqcount_t    d_seq;                    /* per dentry seqlock */
    struct hlist_bl_node    d_hash;         /*散列表节点成员，将实例链入管理散列表*/
    struct dentry    *d_parent;             /*指向父目录项*/
    struct qstr    d_name;                  /*目录项名称，文件名称*/
    struct inode    *d_inode;          /*指向表示文件的 inode 实例*/
    unsigned char d_iname[DNAME_INLINE_LEN];      /*用于存放较短的目录项名称*/
    struct lockref    d_lockref;                  /*锁和引用计数*/
    const struct dentry_operations    *d_op;     /*目录项操作结构指针*/
    struct super_block    *d_sb;                  /*指向所在具体文件系统的超级块实例*/
    unsigned long    d_time;                      /* used by d_revalidate */
    void    *d_fsdata;                            /*指向具体文件系统定义的私有数据结构*/

    struct list_head d_lru;             /*将实例链入超级块 LRU  链表*/
    struct list_head d_child;      /*链接同一父目录项下的兄弟目录项*/
```

```
    struct list_head d_subdirs;     /*链接子目录项*/
    union {
        struct hlist_node    d_alias;      /*用于在 inode 中链接 dentry，链接文件具有多个目录项*/
        struct rcu_head      d_rcu;
    } d_u;
};
```

下面简要介绍 dentry 结构体中比较重要的几个成员：

●**d_flags**：目录项标志，标记 dentry 实例的属性，各标志位定义如下（/include/linux/dcache.h）：

```
#define   DCACHE_OP_HASH              0x00000001   /*dentry_operation 实例定义了 d_hash()函数*/
#define   DCACHE_OP_COMPARE           0x00000002   /*目录项操作定义了 d_compare()函数*/
#define   DCACHE_OP_REVALIDATE        0x00000004   /*目录项操作定义了 d_revalidate()函数*/
#define   DCACHE_OP_DELETE            0x00000008   /*目录项操作定义了 d_delete()函数*/
#define   DCACHE_OP_PRUNE             0x00000010   /*目录项操作定义了 d_prune()函数*/
#define   DCACHE_OP_WEAK_REVALIDATE   0x00000800
                                                   /*目录项操作定义了 d_weak_revalidate()函数*/
#define   DCACHE_OP_SELECT_INODE      0x02000000
                                                   /*目录项操作定义了 d_select_inode()函数*/
#define   DCACHE_DISCONNECTED         0x00000020   /*dentry 没有添加到根文件系统*/
#define   DCACHE_REFERENCED           0x00000040   /*最近被使用，不要释放*/
#define   DCACHE_RCUACCESS            0x00000080   /* Entry has ever been RCU-visible */
#define   DCACHE_CANT_MOUNT           0x00000100   /*此目录项不能挂载文件系统*/
#define   DCACHE_GENOCIDE             0x00000200
#define   DCACHE_SHRINK_LIST          0x00000400
#define   DCACHE_NFSFS_RENAMED        0x00001000
#define   DCACHE_COOKIE               0x00002000    /* For use by dcookie subsystem */
#define   DCACHE_FSNOTIFY_PARENT_WATCHED   0x00004000
#define   DCACHE_DENTRY_KILLED              0x00008000

#define   DCACHE_MOUNTED                0x00010000   /*此目录项是挂载点*/
#define   DCACHE_NEED_AUTOMOUNT         0x00020000   /*在此目录项处理自动挂载*/
#define   DCACHE_MANAGE_TRANSIT         0x00040000   /* manage transit from this dirent */
#define   DCACHE_MANAGED_DENTRY   \
    (DCACHE_MOUNTED|DCACHE_NEED_AUTOMOUNT|DCACHE_MANAGE_TRANSIT)
#define   DCACHE_LRU_LIST             0x00080000       /*实例在超级块 LRU 链表中*/

 /*以下表示目录项类型*/
#define   DCACHE_ENTRY_TYPE           0x00700000
#define   DCACHE_MISS_TYPE            0x00000000 /* Negative dentry (maybe fallthru to nowhere) */
#define   DCACHE_WHITEOUT_TYPE        0x00100000 /* Whiteout dentry (stop pathwalk) */
#define   DCACHE_DIRECTORY_TYPE         0x00200000   /*普通目录项*/
#define   DCACHE_AUTODIR_TYPE         0x00300000   /* Lookupless directory*/
#define   DCACHE_REGULAR_TYPE         0x00400000   /*文件目录项*/
#define   DCACHE_SPECIAL_TYPE         0x00500000    /*特殊文件目录项*/
```

#define **DCACHE_SYMLINK_TYPE** 0x00600000 /*符号链接目录项*/

#define DCACHE_MAY_FREE 0x00800000
#define DCACHE_FALLTHRU 0x01000000 /* Fall through to lower layer */

●**d_hash**：散列表节点成员，hlist_bl_node 结构体实例，用于将 dentry 实例链入全局散列表。

●**d_parent**：指向父目录项。

●**d_child，d_subdirs**：双链表成员，d_child 用于链接处于同一父目录项下的兄弟目录项，d_subdirs 表示双链表头，管理其下子目录项。

●**d_inode**：指向目录项对应的 inode 实例。

●**d_sb**：指向目录项所在文件系统的超级块实例，每个挂载的文件系统在虚拟文件系统中具一个 super_block 结构体实例。

●**d_fsdata**：指向具体文件系统定义的私有数据结构，例如，在 sysfs 文件系统中指向表示该目录项的 kernfs_node 结构体实例。

●**d_lru**：lru 链表成员，将暂时不使用的目录项链入超级块 super_block.s_dentry_lru 链表。

●**d_alias，d_rcu**：链表元素，将 dentry 实例链入 inode 实例中链表。

●**d_lockref**：带引用计数的锁，lockref 结构体实例，结构体定义在/include/linux/lockref.h 头文件：

```
struct lockref {
    union {
      #if USE_CMPXCHG_LOCKREF
        ...
      #endif
        struct {
            spinlock_t lock;        /*保护自旋锁*/
            int count;              /*引用计数*/
        };
    };
};
#define d_lock      d_lockref.lock      /*d_lock 表示自旋锁成员*/
```

lockref 结构体中包含一个保护自旋锁成员和引用计数成员。d_count(const struct dentry *dentry)函数返回 d_lockref 成员中引用计数值，dget_dlock(struct dentry *dentry)用于增加引用计数值，不加锁。dget(struct dentry *dentry)用于在持有自旋锁的情况下增加引用计数值。减小引用计数值的操作后面再做介绍，因为涉及到目录项实例的释放。

●**d_name、d_iname[DNAME_INLINE_LEN]**：这两个成员用于保存目录项名称字符串。d_name 成员是 qstr 结构体实例。qstr 结构体定义在/include/linux/dcache.h 头文件内：

```
#ifdef __LITTLE_ENDIAN                          /*小端系统*/
    #define HASH_LEN_DECLARE u32 hash; u32 len;
    #define bytemask_from_count(cnt)    (~(~0ul << (cnt)*8))
#else                                           /*大端系统*/
    ...
#endif

struct qstr {
```

```
    union {
        struct {            /*32 位系统*/
            HASH_LEN_DECLARE;     /*包含 2 个 32 位无符号整数，hash 和 len。*/
        };
        u64 hash_len;       /*64 位系统*/
     };
    const unsigned char *name;              /*指向目录项名称字符串*/
};
```

在 32 位系统中，qstr 结构体中 HASH_LEN_DECLARE 成员包含 2 个 32 位无符号整数分别是 hash 和 len，用于表示目录项实例的散列值和名称字符串长度，hash 散列值在创建 dentry 实例时计算（或赋值），计算函数为 full_name_hash(name, len)。

注意此散列值并不是 denrty 添加到散列表时的散列值，散列值由 d_hash(const parent,hash)函数计算而得，此函数中使用了父目录项指针和本目录项的 hash 值作为参数。

当目录项名称字符串长度比较短时，将直接保存在 d_iname[DNAME_INLINE_LEN]字符数组成员中，d_name 成员 name 指针指向该数组。数组长度定义如下：

```
#ifdef CONFIG_64BIT
    # define DNAME_INLINE_LEN    32
#else
    # ifdef CONFIG_SMP
        #    define DNAME_INLINE_LEN 36
    # else
        #    define DNAME_INLINE_LEN 40     /*40 个字符*/
    # endif
#endif
```

在 32 位系统中 d_iname[DNAME_INLINE_LEN]字符数组长度为 40 个字符，因此一般目录项名称都会保存在 d_iname[]成员中。如果名称字符数多于 DNAME_INLINE_LEN，内核将另外申请内存空间保存名称字符串。

2 dentry_operations

dentry_operations 结构体包含特定于文件系统类型的目录项操作，须由具体文件系统类型代码实现。dentry 结构体中 d_op 成员指向 dentry_operations 结构体实例，实例来源于超级块 super_block 结构体 s_d_op 成员。在挂载文件系统时，dentry_operations 实例由具体文件系统的挂载函数赋予超级块实例，再由超级块实例传递给其下目录项 dentry 实例。

dentry_operations 结构体定义在/include/linux/dcache.h 头文件内：

```
struct dentry_operations {
    int    (*d_revalidate)    (struct dentry *, unsigned int);   /*用于网络文件系统，检查是否仍然有效*/
    int    (*d_weak_revalidate)    (struct dentry *, unsigned int);
    int    (*d_hash)    (const struct dentry *, struct qstr *);              /*计算目录项散列值*/
    int    (*d_compare)    (const struct dentry *, const struct dentry *,     \
                    unsigned int, const char *, const struct qstr *);          /*比较目录项名称*/
    int    (*d_delete)    (const struct dentry *);       /*引用计数小于等于 1 时调用 d_delete()*/
    void    (*d_release)    (struct dentry *);          /*最后销毁目录项前调用 d_release()*/
    void    (*d_prune)    (struct dentry *);
```

```
    void    (*d_iput)    (struct dentry *, struct inode *);    /*释放目录项对应的 inode 实例*/
    char    *(*d_dname)    (struct dentry *, char *, int);    /*设置目录项名称*/
    struct vfsmount    *(*d_automount)(struct path *);
    int    (*d_manage)    (struct dentry *, bool);
    struct inode    *(*d_select_inode)    (struct dentry *, unsigned);
} ____cacheline_aligned;
```

dentry_operations 结构体中函数功能简要说明如下（/Documentation/filesystems/vfs.txt）：

●**d_revalidate**：目录项是否有效。在根文件系统中查找到 dentry 实例时，需调用此函数确认目录项是否有效，一般文件系统此函数为空，因为目录项在缓存中就表示目录项有效，此函数主要用于网络文件系统。目录项仍然有效返回正值，否则返回 0 或负值。

●**d_weak_revalidate**：与 d_revalidate 相似，在 VFS 需要重新判断“跳动”目录项的有效性时调用，如"/"， "." 和 ".."。一般文件系统为空操作，目录项仍然有效返回正值，否则返回 0 或负值。

●**d_hash**：计算目录项的散列值；

●**d_compare**：将目录项名称与指定字符串比较。第一个参数为父目录项，第二个参数为子目录项，len 和 name 为被比较目录项长度和名称，qstr 为要与目录项比较的字符串。例如，在 FAT32 文件系统中，目录项名称是不区分大小写的，因此需要定义专门的比较函数。

●**d_delete**：当 dentry 引用计数值小于等于 1 时调用，返回 1 表示立即销毁目录项，0 表示缓存目录项至超级块 LRU 链表，如果函数指针为 NULL 表示默认缓存目录项。

●**d_release**：dentry 被销毁时由 dentry_kill()函数调用。

●**d_prune**：dentry 被销毁时由 dentry_kill()函数调用。

●**d_iput**：dentry_kill()函数销毁 dentry 实例时，调用此函数释放对应 inode 实例，如果函数指针为 NULL，将调用 VFS 提供的 **iput()**函数释放 inode 实例。

●**d_dname**：用于产生目录项名称，通常用于伪文件系统。

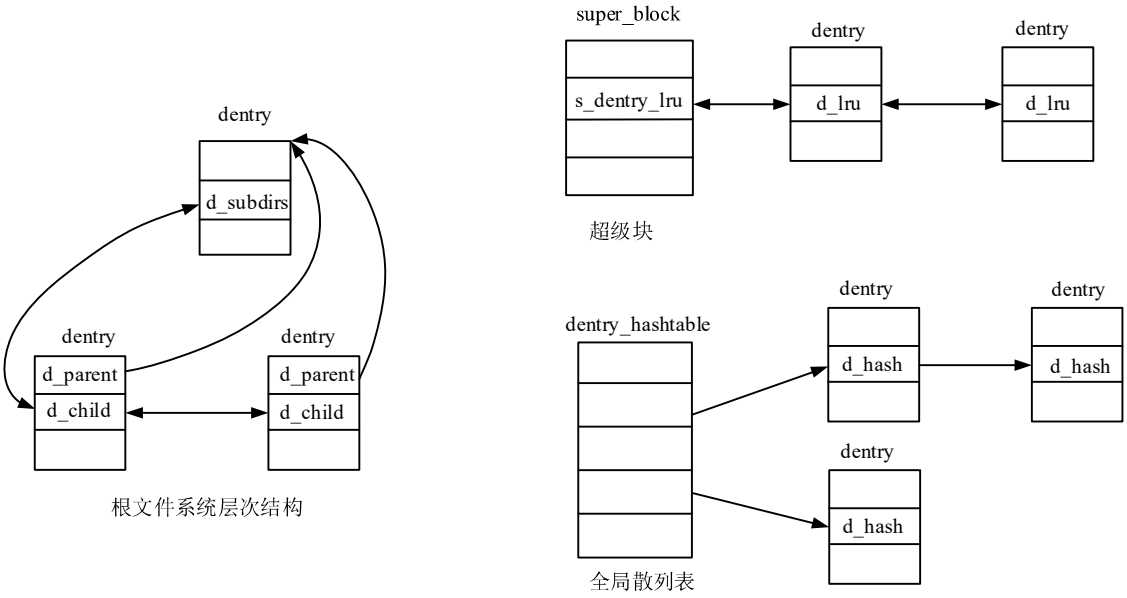
●**d_automount**：当 dentry 实例设置了 DCACHE_NEED_AUTOMOUNT 标记，调用此函数产生 vfsmount 结构体返回给调用者，调用者可挂载文件系统到此目录项。

●**d_manage**：只有设置了 DCACHE_MANAGE_TRANSIT 标记的 dentry 实例才有效，用于向 dentry 实例传递信息，正常返回 0，不允许操作返回负的错误码。

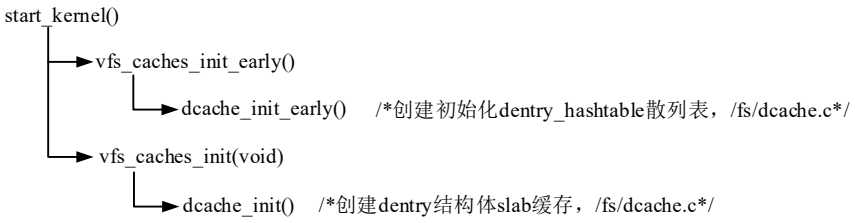
7.2.3 目录项操作

内核在打开文件时，会为各路径分量创建对应的 dentry 实例，新创建的实例首先添加到根文件系统层次结构中，在为 dentry 实例创建并关联对应的 inode 实例时会将其添加到全局散列表 dentry_hashtable。当 dentry 实例被释放时被添加到超级块 LRU 链表。在关闭文件时，内核会释放 dentry 实例。

dentry 实例管理结构如下图所示：



内核在启动阶段调用 dcache_init_early()和 dcache_init()函数分别为 dentry 结构体创建全局散列表（并初始化）和 slab 缓存，函数调用关系如下：



以上函数都定义在/fs/dcache.c 文件内，源代码请读者自行阅读。

1 创建目录项

内核在打开文件时为路径中的每个分量（目录项）创建 dentry 实例，并加入到管理结构中。

创建目录项的接口函数有 d_alloc()、d_alloc_name()、d_make_root()等，这些函数都定义在/fs/dcache.c 文件内，下面分别对它们做简要介绍。

d_alloc()函数定义如下（/fs/dcache.c）：

```
struct dentry *d_alloc(struct dentry * parent, const struct qstr *name)
/*parent：父目录项指针，name：qstr 结构体指针，表示目录项名称*/
{
    struct dentry *dentry = __d_alloc(parent->d_sb, name);
                                        /*创建 dentry 实例，需要超级块实例指针参数，/fs/dcache.c*/
    if (!dentry)
        return NULL;

    spin_lock(&parent->d_lock);
    __dget_dlock(parent);               /*增加父目录项引用计数*/
    dentry->d_parent = parent;          /*指向父目录项*/
    list_add(&dentry->d_child, &parent->d_subdirs);     /*添加到父目录项的子目录项链表中*/
    spin_unlock(&parent->d_lock);
```

```
    return dentry;     /*返回目录项实例指针*/
}
```

d_alloc()函数实现比较简单，主要是调用__d_alloc()函数从 slab 分配器中分配 dentry 实例并初始化，然后增加父目录项的引用计数，并建立与父目录项之间的关联（添加到根文件系统），但是实例尚未添加到全局散列表。

__d_alloc()函数实现也比较简单，主要工作是：从 slab 分配器中分配 dentry 实例，根据 name 参数设置名称成员，引用计数置 1，标记成员置 0，各链表成员初始化为空，设置目录项操作结构指针成员为**parent->d_sb->s_d_op**，即目录项操作结构体实例来自于父目录项所在的超级块实例。需要注意的是在__d_alloc()函数中，如果名称字符串过长，则从通用缓存中分配空间存储目录项名称字符串，否则直接保存在 d_iname[]字符数组成员中。

d_alloc_name()函数与 d_alloc()函数类似，只不过参数中可以直接传递目录项名称字符串，而不需要传递 qstr 实例指针。函数定义如下：

```
struct dentry *d_alloc_name(struct dentry *parent, const char *name)
{
    struct qstr q;          /*qstr 实例*/

    q.name = name;       /*设置 qstr 实例*/
    q.len = strlen(name);
    q.hash = full_name_hash(q.name, q.len);
                            /*计算目录项 d_name 成员中散列值 hash，非散列表的散列值，/fs/namei.c*/
    return d_alloc(parent, &q);
}
```

d_alloc_name()函数内通过目项名称字符串构建 qstr 实例，并计算散列值，最后调用 d_alloc()函数创建 dentry 实例。需要注意的是此处计算的散列值是 d_name 成员（qstr 结构体）中的散列值，而不是 dentry 实例添加到全局散列表中的散列值，后者需要以前者为参数通过 d_hash()函数计算而得。

另一个比较常用的创建 dentry 实例的接口函数是 d_make_root()，通常用于文件系统类型的挂载函数中，用于创建挂载文件系统的根目录项。函数代码如下：

```
struct dentry *d_make_root(struct inode *root_inode)
/*root_inode：目录项对应 inode 实例指针*/
{
    struct dentry *res = NULL;

    if (root_inode) {
        static const struct qstr name = QSTR_INIT("/", 1);      /*目录项名称为"/"，表示根目录项*/

        res = __d_alloc(root_inode->i_sb, &name);       /*创建 dentry 实例*/
        if (res)
            d_instantiate(res, root_inode); /*关联 inode 实例，但 dentry 实例没有加入全局散列表*/
        else
            iput(root_inode);          /*创建 dentry 失败，释放 inode*/
    }
```

```
    return res;
}
```

2 关联节点

d_add()函数用于建立 dentry 实例和 inode 实例之间的关联，函数定义如下（/include/linux/dcache.h）：

```
static inline void d_add(struct dentry *entry, struct inode *inode)
{
    d_instantiate(entry, inode);      /*关联 inode 实例，设置 dentry 实例标记成员等，/fs/dcache.c*/
    d_rehash(entry);                  /*添加到全局散列表，由 d_hash()确定计算散列值，/fs/dcache.c*/
}
```

d_add()函数调用 d_instantiate()函数建立 dentry 实例与 inode 实例之间的关联，调用 d_rehash()函数将 dentry 实例添加到全局散列表 dentry_hashtable。

d_rehash()函数定义在/fs/dcache.c 文件内，源代码请读者自行阅读，需要注意的是函数内调用 d_hash()函数计算散列值，由此确定散列链表头，d_hash()函数需要父 dentry 实例指针和本 dentry 实例 d_name.hash 成员值做为参数。

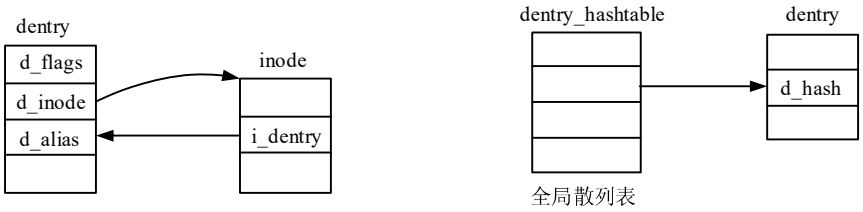
下面看一下 d_instantiate()函数的实现，定义如下：

```
void d_instantiate(struct dentry *entry, struct inode * inode)
{
    BUG_ON(!hlist_unhashed(&entry->d_u.d_alias));
    if (inode)
        spin_lock(&inode->i_lock);
    __d_instantiate(entry, inode);      /*/fs/dcache.c*/
    if (inode)
        spin_unlock(&inode->i_lock);
    security_d_instantiate(entry, inode);
}
```

__d_instantiate(entry, inode)函数定义在/fs/dcache.c 文件内，主要工作是由 inode 实例生成 dentry 实例的标志成员值，然后将 dentry 实例添加到 inode 实例中的散列链表，最后设置 dentry 实例 d_inode（指向 inode）和 d_flags 成员。

```
static void __d_instantiate(struct dentry *dentry, struct inode *inode)
{
    unsigned add_flags = d_flags_for_inode(inode);   /*生成 dentry 标志值，/fs/dcache.c*/

    spin_lock(&dentry->d_lock);
    if (inode)
        hlist_add_head(&dentry->d_u.d_alias, &inode->i_dentry); /*添加到 inode 实例散列链表*/
    __d_set_inode_and_type(dentry, inode, add_flags);
                        /*设置 dentry 实例 d_inode 和 d_flags 标志成员，/fs/dcache.c*/
    dentry_rcuwalk_invalidate(dentry);
    spin_unlock(&dentry->d_lock);
    fsnotify_d_instantiate(dentry, inode);
}
```

执行 d_add(struct dentry *entry, struct inode *inode)函数后，dentry 和 inode 实例关系如下图所示：



3 查找目录项

dentry 实例与 inode 实例关联后，同时添加到全局散列表。内核在打开文件时，对目录分量先调用 d_lookup()函数在全局散列表中查找是否已存在对应的 dentry 实例，如果存在则无需创建，不存在则需要创建。

d_lookup()函数定义如下（/fs/dcache.c）：

```
struct dentry *d_lookup(const struct dentry *parent, const struct qstr *name)
/*parent：父目录项指针，name：目录项名称 qstr 结构体指针，非字符串指针*/
{
   struct dentry *dentry;
   unsigned seq;

   do {
      seq = read_seqbegin(&rename_lock);
      dentry = __d_lookup(parent, name);      /*在全局散列表中查找 dentry 实例，/fs/dcache.c*/
      if (dentry)
            break;
   } while (read_seqretry(&rename_lock, seq));
   return dentry;      /*返回查找的 dentry 实例指针，没找到返回 NULL*/
}
```

__d_lookup()函数在全局散列表中查找 name 对应的 dentry 实例，成功返回 dentry 实例指针，否则返回 NULL，函数代码如下：

```
struct dentry *__d_lookup(const struct dentry *parent, const struct qstr *name)
{
      unsigned int len = name->len;
      unsigned int hash = name->hash;          /*qstr 中保存的散列值*/
      const unsigned char *str = name->name;
      struct hlist_bl_head *b = d_hash(parent, hash);      /*计算散列值，确定散列表链表头，/fs/dcache.c*/
      struct hlist_bl_node *node;
      struct dentry *found = NULL;
      struct dentry *dentry;

      rcu_read_lock();

      hlist_bl_for_each_entry_rcu(dentry, node, b, d_hash) {     /*遍历 d_hash()计算的散列链表*/
            if (dentry->d_name.hash != hash)        /*目录项中散列值不相等，跳过*/
                  continue;
```

```
            spin_lock(&dentry->d_lock);
            if (dentry->d_parent != parent)       /*父目录项不同，跳过*/
                goto next;
            if (d_unhashed(dentry))              /*目录项不在散列链表中，跳过*/
                goto next;
            if (parent->d_flags & DCACHE_OP_COMPARE) { /*父目录项操作结构中具有比较函数*/
                int tlen = dentry->d_name.len;        /*比较 name 中传递的名称与查找的 dentry 中的名称*/
                const char *tname = dentry->d_name.name;
                if (parent->d_op->d_compare(parent, dentry, tlen, tname, name))
                    goto next;
            } else {                    /*没有定义比较函数，直接进行字符串比较*/
                if (dentry->d_name.len != len)      /*如果目录项名称、长度不相同，跳过*/
                    goto next;
                if (dentry_cmp(dentry, str, len))        /*字符串比较*/
                    goto next;
            }

            /*找到 name 对应的 dentry 实例*/
            dentry->d_lockref.count++;        /*增加引用计数*/
            found = dentry;                   /*返回值 dentry 实例指针*/
            spin_unlock(&dentry->d_lock);
            break;
        next:
            spin_unlock(&dentry->d_lock);
        }
        rcu_read_unlock();
        return found;      /*返回实例指针*/
    }
```

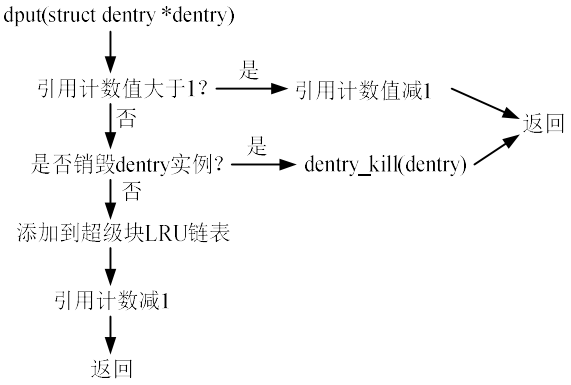
__d_lookup()函数内根据 name 传递的目录项名称参数，遍历全局散列表中对应的散列链表，根据目录项散列值、父目录项指针、本目录项名称及长度等信息确定链表中成员是否是需要查找的 dentry 实例，如果是则增加实例引用计数值，返回 dentry 实例指针，没找到则返回 NULL。

这里需要注意的是在名称匹配时，如果具体文件系统定义了目录项操作函数 d_compare()，则调用它判断目录项名称是否相同，否则直接比较字符串值及长度。例如，在 FAT32 文件系统中，目录项字符串是不区分大小写的，因此不能直接比较字符串值，而需要定义专门的比较函数。

4 释放目录项

内核在打开文件时创建目录项 dentry 实例，在关闭文件或其它不再需要 dentry 实例的情况时，调用函数 dput()释放 dentry 实例。

释放目录项是一个稍显复杂的操作，首先需要判断目录项当前引用计数值是否大于 1，如果是则直接将引用计数值减 1 即可，不需要做其它处理。如果引用计数值小于等于 1，则需要判断是将 dentry 实例暂时缓存到超级块 LRU 链表中，还是直接销毁（释放回 slab 缓存），从而分别采取不同的操作。释放目录项函数 dput()执行流程简列如下所示：



dput(struct dentry *dentry)函数定义如下（/fs/dcache.c）：

```
void dput(struct dentry *dentry)
{
    if (unlikely(!dentry))
        return;
repeat:
    rcu_read_lock();
    if (likely(fast_dput(dentry))) {   /*引用计数值是否大于 1，是则减 1 后返回，/fs/dcache.c*/
        rcu_read_unlock();           /*函数返回*/
        return;
    }

    /*引用计数值小于等于 1*/
    rcu_read_unlock();

    if (unlikely(d_unhashed(dentry)))     /*dentry 不在全局散列表中，销毁 denrty 实例*/
        goto kill_it;

    if (unlikely(dentry->d_flags & DCACHE_DISCONNECTED))
                                    /*dentry 不在根文件系统中，销毁*/
        goto kill_it;

    /*dentry 实例还在全局散列表和根文件系统中*/
    if (unlikely(dentry->d_flags & DCACHE_OP_DELETE)) {
                                                /*调用 dentry 操作结构体 d_delete()函数*/
        if (dentry->d_op->d_delete(dentry))       /*具体文件系统判断是否需要销毁 dentry 实例*/
            goto kill_it;
    }

    /*不销毁 dentry 实例，缓存到超级块 LRU 链表*/
    if (!(dentry->d_flags & DCACHE_REFERENCED))
        dentry->d_flags |= DCACHE_REFERENCED;       /*设置标记位*/
    dentry_lru_add(dentry);       /*添加到超级块 LRU 链表，并设置 DCACHE_LRU_LIST 标志位*/
```

```
        dentry->d_lockref.count--;      /*引用计数值减 1*/
        spin_unlock(&dentry->d_lock);
        return;

    kill_it:
        dentry = dentry_kill(dentry);       /*销毁 dentry 实例，/fs/dcache.c*/
        if (dentry)
            goto repeat;
    }
```

fast_dput(dentry)函数定义在/fs/dcache.c 文件内，主要工作是判断 dentry 实例当前引用计数值是否大于 1，如果是则将其减 1 后，返回 1，dput()函数也将直接返回。如果引用计数值小于等于 1，fast_dput()函数返回 0，dput()函数将继续往下执行。

如果引用计数值小于等于 1，dput()函数将判断 dentry 实例是否存在于散列表或根文件系统中，如果不是，则调用 dentry_kill()函数将其销毁（释放到 slab 分配器），否则将其添加到超级块 LRU 链表（具体文件系统还可确定是否将其销毁）。

dentry_kill()函数内调用__dentry_kill()函数完成 dentry 实例的销毁工作，函数代码如下（/fs/dcache.c）：

```
static void __dentry_kill(struct dentry *dentry)
{
    struct dentry *parent = NULL;
    bool can_free = true;
    if (!IS_ROOT(dentry))
        parent = dentry->d_parent;

    lockref_mark_dead(&dentry->d_lockref);

    if (dentry->d_flags & DCACHE_OP_PRUNE)
        dentry->d_op->d_prune(dentry);

    if (dentry->d_flags & DCACHE_LRU_LIST) {        /*如果在超级块 LRU 链表中，则移出*/
        if (!(dentry->d_flags & DCACHE_SHRINK_LIST))
            d_lru_del(dentry);
    }

    __d_drop(dentry);       /*如果还在散列表中，则将其移出*/
    __list_del_entry(&dentry->d_child);     /*从父目录项的子目录项链表中移出*/

    dentry->d_flags |= DCACHE_DENTRY_KILLED;
    if (parent)
        spin_unlock(&parent->d_lock);
    dentry_iput(dentry);        /*释放关联 inode 实例，见下一节，/fs/dcache.c*/

    BUG_ON(dentry->d_lockref.count > 0);
```

```
        this_cpu_dec(nr_dentry);
        if (dentry->d_op && dentry->d_op->d_release)
            dentry->d_op->d_release(dentry);

        spin_lock(&dentry->d_lock);
        if (dentry->d_flags & DCACHE_SHRINK_LIST) {
            dentry->d_flags |= DCACHE_MAY_FREE;
            can_free = false;
        }
        spin_unlock(&dentry->d_lock);
        if (likely(can_free))
            dentry_free(dentry);        /*dentry 实例释放回 slab 分配器*/
    }
```

__dentry_kill()函数内将调用目录项操作结构中的 d_prune()函数，随后将实例从超级块 LRU 链表、散列表中移出，从父目录项的子目录项链表中移出，调用 dentry_iput(dentry)函数释放对应的 inode 实例，调用目录项操作结构中的 d_release()函数，最后调用 dentry_free(dentry)函数将 dentry 实例释放回 slab 分配器。

dentry_iput(dentry)函数将调用 d_op->d_iput(dentry, inode)函数（或 iput(inode)函数）释放关联的 inode 实例，详见下一节。

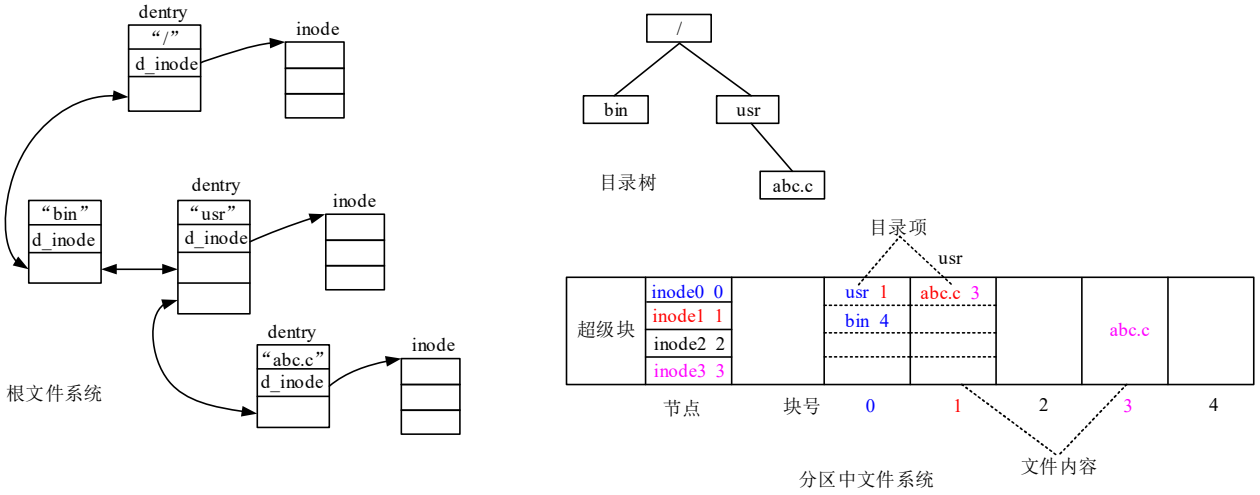
7.3 节点

目录项中保存了普通目录、文件的名称等少量信息，主要用于文件的管理，构成文件树状层次结构。文件的具体信息，如读写权限、修改时间、文件内容等，保存在被称为节点的结构中。

在虚拟文件系统中，节点由 inode 结构体表示。在具体文件系统中，有的文件系统类型具体与节点相对应的结构，如 ext2、ext3 等，有的没有与节点对应的结构，如 FAT32 等。内核在打开文件（目录）时，需要从具体文件系统中抽取文件的相关信息，构建 inode 结构体实例。也就是说 inode 是文件信息（内容）在内核中的抽象表示，不管理具体文件系统中以什么形式保存文件信息，导入内核后都由 inode 结构体表示。

7.3.1 物理结构

在前一节介绍目录项物理结构中，简要示意了块设备文件系统的结构。在实际的文件系统中，保存文件内容的数据块号等信息不是保存在目录项中，而是保存在被称为节点的结构中，目录项中保存的是对应文件节点的编号，如下图所示。



图中以 Linux 操作系统中标准的 ext2 文件系统为例，说明节点的物理结构。文件系统中在超级块之后划分出一个区域用于保存节点，节点是固定长度的数据结构，每个节点具有唯一的编号。节点中保存了文件的访问权限、时间戳等信息，以及最重要的保存文件内容的数据块号信息。在目录项中保存了对应节点的编号，通常节点 0 表示根目录项文件。

如图中所示，根目录下存在 usr、bin 二个目录项，usr 目录项中保存的节点编号为 1，对应 inode1，inode1 中保存了存放文件内容的数据块号，此处为 1，即 usr 目录项对应的文件内容保存在块号 1 中。数据块 1 中保存了 abc.c 文件目录项，目录项中保存的节点编号为 3，inode3 中保存的数据块号为 3，表示 abc.c 文件内容保存在数据块 3 中。

实际的节点中用于保存数据块号的是一个整数数组，数组中包含的数据块号可以直接是表示保存文件内容的数据块，也可以是保存数据块号的数据块（相当于二级指针），详细信息请参考 7.10 节。现在我们只需要知道每个目录和文件都对应一个节点，节点中包含了目录/文件的访问权限、各种时间值、保存文件内容的数据块号等信息即可。

虚拟文件系统中 inode 结构体即是对文件系统中节点的抽象。内核中 inode 结构体与具体文件系统中的节点结构并不完全一样，因为 inode 结构需要适用所有的文件系统，而不是只适用于一种文件系统。具体的文件系统中可能不存在与这里介绍的节点相对应的结构，它可能以其它的形式来保存文件的信息。具体文件系统类型的实现代码需要定义构建 inode 实例的函数，从文件系统中抽取相关信息生成 inode 实例，以实现内核定义的统一接口。

内核在打开块设备中文件时，在文件系统中逐级查找目录项，创建 dentry 实例，并创建 inode 实例。inode 实例的创建通常由超级块操作结构中的 alloc_inode()函数完成，此函数创建 inode 实例，并执行特定于具体文件系统类型的操作，从块设备文件系统中抽取文件信息，设置 inode 实例（包括节点操作结构、文件操作结构等）。以此在内核中建立文件的表示（inode 实例），完成文件的打开操作。

7.3.2 数据结构

inode 结构体是所有文件在虚拟文件系统中的抽象表示，用于在内核中用于唯一地表示文件。内核在打开文件时，在创建目录项 dentry 实例的同时创建 inode 实例，并从具体文件系统类型的实现代码完成 inode 实例的设置。

1 inode

inode 结构体定义在/include/linux/fs.h 头文件，主要包含文件的元信息和缓存文件内容的文件地址空间结构体等成员：

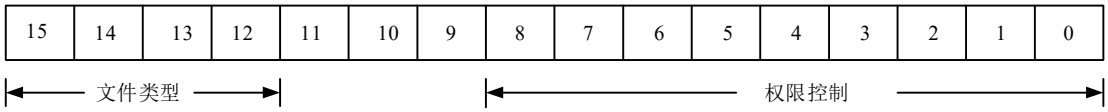
```
struct inode {
    umode_t         i_mode;      /*文件类型及访问权限*/
    unsigned short  i_opflags;   /*进程打开文件的标记*/
    kuid_t          i_uid;       /*文件用户主 ID*/
    kgid_t          i_gid;       /*文件用户组 ID*/
    unsigned int    i_flags;     /*表示文件属性*/
#ifdef CONFIG_FS_POSIX_ACL       /*访问控制列表（ACL）*/
    struct posix_acl    *i_acl;
    struct posix_acl    *i_default_acl;
#endif
    const struct inode_operations *i_op;      /*节点操作结构指针*/
    struct super_block      *i_sb;            /*文件系统超级块结构指针*/
    struct address_space    *i_mapping;       /*文件地址空间结构体指针，通常指向 i_data 成员*/
#ifdef CONFIG_SECURITY
    void            *i_security;
#endif
    unsigned long   i_ino;       /*inode 编号，文件系统内（分区内）编号唯一*/
    union {
        const unsigned int  i_nlink;
        unsigned int    __i_nlink;
    };
    /*inode 元信息*/
    dev_t                   i_rdev;     /*表示设备文件时，保存设备号*/
    loff_t                  i_size;     /*文件大小，字节数*/
    struct timespec         i_atime;    /*最后访问时间*/
    struct timespec         i_mtime;    /*最后修改时间*/
    struct timespec         i_ctime;    /*最后修改 inode 时间*/
    spinlock_t              i_lock;     /*锁定 i_blocks, i_bytes, maybe i_size */
    unsigned short          i_bytes;    /*块大小，字节数*/
    unsigned int            i_blkbits;  /*块大小字节数取对数*/
    blkcnt_t                i_blocks;   /*文件大小，块大小*/

#ifdef __NEED_I_SIZE_ORDERED
    seqcount_t      i_size_seqcount;
#endif

    unsigned long   i_state;     /*状态信息*/
    struct mutex    i_mutex;     /*互斥锁用于保护地址空间*/
    unsigned long   dirtied_when;   /* jiffies of first dirtying */
    unsigned long   dirtied_time_when;

    struct hlist_node   i_hash;         /*将实例链入全局散列表*/
    struct list_head    i_wb_list;      /*将实例链入后备存储设备回写 inode 链表*/
```

```
#ifdef CONFIG_CGROUP_WRITEBACK     /*组回写*/
    struct bdi_writeback    *i_wb;          /* the associated cgroup wb */
    int             i_wb_frn_winner;
    u16             i_wb_frn_avg_time;
    u16             i_wb_frn_history;
#endif
    struct list_head    i_lru;          /*将实例链入超级块 LRU 链表，表头 sb.s_inode_lru */
    struct list_head    i_sb_list;      /*将实例链入超级块 inode 链表，表头 sb.s_inodes*/
    union {
        struct hlist_head       i_dentry;       /*散列链表头，链接关联的 dentry 实例*/
        struct rcu_head         i_rcu;
    };
    u64             i_version;
    atomic_t        i_count;            /*引用计数*/
    atomic_t        i_dio_count;
    atomic_t        i_writecount;       /*大于 0 表示用户正在写入数据*/
#ifdef CONFIG_IMA
    atomic_t        i_readcount;            /* struct files open RO */
#endif
    const struct file_operations    *i_fop;     /*文件操作结构指针，文件操作的接口*/
    struct file_lock_context  *i_flctx;
    struct address_space        i_data;     /*文件地址空间实例，缓存文件内容（页缓存）*/
    struct list_head            i_devices;
                /*inode 表示设备文件时，将实例链接到 cdev.list 或 block_device.bd_inodes 链表*/
    union {
        struct pipe_inode_info  *i_pipe;    /*管道文件*/
        struct block_device     *i_bdev;    /*指向块设备结构体，block_device*/
        struct cdev             *i_cdev;    /*指向字符设备结构体，cdev*/
        char                    *i_link;    /*链接文件内容字符串*/
    };
    __u32               i_generation;
#ifdef CONFIG_FSNOTIFY
    __u32               i_fsnotify_mask;    /* all events this inode cares about */
    struct hlist_head   i_fsnotify_marks;
#endif
    void            *i_private;     /*指向具体文件系统私有数据*/
};
```

inode 结构体主要成员成员简介如下：

●**i_mode**：标记文件类型及访问权限。数据类型为 umode_t，它是一个 16 位的无符号整型数，定义在/include/linux/types.h 头文件内。各标记位定义如下：

umod_t结构



i_mode 成员高 4 位表示文件类型，低 9 位表示文件访问权限，中间 4 位表示用户 ID 信息，各比特位语义定义在/include/uapi/linux/stat.h 头文件内，这些是可由用户程序使用的宏：

```
/*高 4 位定义，bit[15:12]，以下为八进制数，4 个二进制位可表示 16 种文件类型*/
#define   S_IFMT      00170000      /*掩码[15:12]=1111*/
#define   S_IFSOCK    0140000       /*套接字文件，1100*/
#define   S_IFLNK     0120000       /*符号链接，1010*/
#define   S_IFREG     0100000       /*普通文件，1000*/
#define   S_IFBLK     0060000       /*块设备文件，0110*/
#define   S_IFDIR     0040000       /*目录文件，0100*/
#define   S_IFCHR     0020000       /*字符设备文件，0010*/
#define   S_IFIFO     0010000       /*管道，0001*/

/*中间三位，bit[11:9]*/
#define   S_ISUID     0004000       /*100(bit11)，设置执行本文件的进程 euid 为文件 i_uid*/
#define   S_ISGID     0002000       /*010(bit10)，设置执行本文件的进程 egid 为文件 i_gid*/
#define   S_ISVTX     0001000       /*001(bit9)，对目录项设置该位，表示仅当非特权进程对目录有
                                    *写权限时，且为文件或目录属主时，才能对目录下文件删除和重命名*/

/*文件访问权限控制  bit[8:0]*/
/*bit[8...6]用于设置文件主（用户）的访问权限，每位分别表示读、写、执行权限*/
#define   S_IRWXU   00700       /*文件主可读写执行，111*/
#define   S_IRUSR   00400       /*文件主可读，100*/
#define   S_IWUSR   00200       /*文件主可写*，010/
#define   S_IXUSR   00100       /*文件主可执行，001*/

/*bit[6...3]用于设置文件主同组用户的访问权限*/
#define   S_IRWXG   00070       /*文件主同组用户可读写执行*/
#define   S_IRGRP   00040       /*文件主同组用户可读*/
#define   S_IWGRP   00020       /*文件主同组用户可写*/
#define   S_IXGRP   00010       /*文件主同组用户可执行*/

/*bit[6...3]用于设置其它用户的访问权限*/
#define   S_IRWXO   00007       /*其它用户可读写执行*/
#define   S_IROTH   00004       /*其它用户可读*/
#define   S_IWOTH   00002       /*其它用户可写*/
#define   S_IXOTH   00001       /*其它用户可执行*/
```

例如，用命令“chmode 764 file_name”表示设置文件主具有读写执行权限，同组用户具有读写权限，其它用户只具有读权限。

内核在/include/linux/stat.h 头文件定义了以下标记组合，用于内核内部使用：

```
#define   S_IRWXUGO      (S_IRWXU|S_IRWXG|S_IRWXO)      /*所有用户具有所有权限*/
#define   S_IALLUGO      (S_ISUID|S_ISGID|S_ISVTX|S_IRWXUGO)      /*设置所有标记位*/
#define   S_IRUGO        (S_IRUSR|S_IRGRP|S_IROTH)      /*所有用户具有读权限*/
#define   S_IWUGO        (S_IWUSR|S_IWGRP|S_IWOTH)    /*所有用户具有写权限*/
#define   S_IXUGO        (S_IXUSR|S_IXGRP|S_IXOTH)    /*所有用户具有执行权限*/
```

目录文件的读写执行权限意义与普通文件稍有不同：

可读：允许列出目录内容，即可读目录文件内容。

可写：允许对目录文件内容进行更改，即可以创建文件、子目录，删除/修改文件/子目录名等。

可执行：有时也称可搜索，表示可以访问目录下的文件和子目录。访问文件时，进程需对路径名所有分量（目录项）具有可执行权限。

●i_flags：inode 标记成员，表示文件属性，取值定义在/include/linux/fs.h：

```
#define   S_SYNC          1          /*写文件时立即同步，写出到块设备*/
#define   S_NOATIME       2          /*不要修改访问时间*/
#define   S_APPEND        4          /*只能在文件尾部添加内容*/
#define   S_IMMUTABLE     8         /* Immutable file */
#define   S_DEAD          16         /*仍然打开，但被移出的目录项*/
#define   S_NOQUOTA       32         /* Inode is not counted to quota */
#define   S_DIRSYNC       64         /* Directory modifications are synchronous */
#define   S_NOCMTIME      128        /*不需要更新 c/mtime */
#define   S_SWAPFILE      256        /* Do not truncate: swapon got its bmaps */
#define   S_PRIVATE       512        /* Inode is fs-internal */
#define   S_IMA           1024       /* Inode has an associated IMA struct */
#define   S_AUTOMOUNT     2048       /* Automount/referral quasi-directory */
#define   S_NOSEC         4096        /* no suid or xattr security attributes */
#ifdef   CONFIG_FS_DAX
    #define   S_DAX       8192      /*直接访问文件，不经过页缓存*/
#else
    #define   S_DAX       0        / *不考虑直接访问文件代码*/
#endif
```

●i_state：表示 inode 实例状态（内核中文件状态），取值定义在/include/linux/fs.h：

```
#define   I_DIRTY_SYNC        (1 << 0)    /*inode 元数据修改，不需要在 fdatasync()中回写*/
#define   I_DIRTY_DATASYNC (1 << 1)
                    /*inode 数据将要修改，当只有 mtime 修改时无需在 fdatasync()中回写*/
#define   I_DIRTY_PAGES       (1 << 2)  /*缓存页脏（文件内容有修改），元数据可能没有修改*/
#define   __I_NEW             3
#define   I_NEW               (1 << __I_NEW)    /*inode 实例是新的，需要与块设备同步*/
#define   I_WILL_FREE         (1 << 4)    /*当调用 write_inode_new()，i_count 为 0 时，置位*/
#define   I_FREEING           (1 << 5)    /*准备释放但还有脏页或 inode 脏时设置*/
#define   I_CLEAR             (1 << 6)    /*调用 clear_inode()时设置*/
```

```
#define   __I_SYNC                7
#define   I_SYNC          (1 << __I_SYNC)     /*inode 正在回写*/
#define   I_REFERENCED        (1 << 8)        /*标记 inode 最近入 LRU 链表*/
#define   __I_DIO_WAKEUP          9
#define   I_DIO_WAKEUP        (1 << __I_DIO_WAKEUP)
#define   I_LINKABLE          (1 << 10)
#define   I_DIRTY_TIME        (1 << 11)
#define   __I_DIRTY_TIME_EXPIRED    12
#define   I_DIRTY_TIME_EXPIRED   (1 << __I_DIRTY_TIME_EXPIRED)
#define   I_WB_SWITCH           (1 << 13)

#define   I_DIRTY   (I_DIRTY_SYNC | I_DIRTY_DATASYNC | I_DIRTY_PAGES)
#define   I_DIRTY_ALL   (I_DIRTY | I_DIRTY_TIME)
```

●i_mapping：文件地址空间 address_space 结构体指针成员，通常指向 inode 结构体中 **i_data** 成员。

●i_data：文件地址空间 address_space 结构体实例，主要用于在基于块设备的文件系统中管理文件内容的页缓存和实现文件缓存页数据与块设备的同步。通用的文件读写操作是直接对页缓存进行读写，而后由地址空间中定义的读写缓存页函数实现数据的同步，详见第 11 章。

●i_op：inode_operations 结构体指针，此结构体主要作用于文件元信息以及对目录项文件内容的操作（目录项的操作），如查找目录项、创建目录项、删除目录项、文件重命名等，后面再详细介绍。

●i_fop：file_operations 结构体指针，此结构体主要用于对文件内容的操作（包括目录项文件），如读写文件内容等，这是文件操作的接口，后面将详细介绍。

2 inode_operations

inode_operations 结构体中函数指针主要用于对文件元信息的操作以及目录项文件内容中目录项的操作。目录项文件的内容是目录项，因此对目录项文件内容的操作就是对其下目录项的操作，例如：在目录下查找指定名称的子目录项、创建新目录项（新文件）、删除目录项、文件重命名等。inode_operations 结构体还包含对文件元属性的操作函数等。

inode_operations 结构体实例由具体文件系统类型实现，在创建 inode 实例时由具体文件系统类型代码将 inode_operations 实例指针赋予 inode 实例 i_op 成员。

inode_operations 结构体体定义在/include/linux/fs.h 头文件：

```
struct inode_operations {
    struct dentry    * (*lookup)   (struct inode *,struct dentry *, unsigned int);      /*查找子目录项*/
    const char    * (*follow_link)   (struct dentry *, void **);
                                                    /*读取符号链接文件内容*/
    int    (*permission)   (struct inode *, int);          /*文件访问权限检查*/
    struct posix_acl    * (*get_acl)(struct inode *, int);
    int    (*readlink)    (struct dentry *, char __user *,int);
    void    (*put_link)    (struct inode *, void *);     /*搜索完符号链接文件内容中的路径后调用的函数*/
    int    (*create)    (struct inode *,struct dentry *, umode_t, bool);     /*创建普通文件*/
    int    (*link)    (struct dentry *,struct inode *,struct dentry *);     /*创建链接*/
    int    (*unlink)    (struct inode *,struct dentry *);
```

```
    int    (*symlink)    (struct inode *,struct dentry *,const char *);
    int    (*mkdir)    (struct inode *,struct dentry *,umode_t);       /*创建目录项*/
    int    (*rmdir)    (struct inode *,struct dentry *);                /*删除目录项*/
    int    (*mknod)    (struct inode *,struct dentry *,umode_t,dev_t);    /*创建设备节点、命名管道等*/
    int    (*rename)    (struct inode *, struct dentry *,struct inode *, struct dentry *);    /*重命名*/
    int    (*rename2)    (struct inode *, struct dentry *,struct inode *, struct dentry *, unsigned int);
    int    (*setattr)    (struct dentry *, struct iattr *);                /*文件属性*/
    int    (*getattr)    (struct vfsmount *mnt, struct dentry *, struct kstat *);       /*获取文件属性*/
    int    (*setxattr)    (struct dentry *, const char *,const void *,size_t,int);       /*设置文件扩展属性*/
    ssize_t    (*getxattr)    (struct dentry *, const char *, void *, size_t);          /*获取文件扩展属性*/
    ssize_t    (*listxattr)    (struct dentry *, char *, size_t);        /*列出扩展属性*/
    int    (*removexattr)    (struct dentry *, const char *);          /*移除一条扩展属性*/
    int    (*fiemap)    (struct inode *, struct fiemap_extent_info *, u64 start,u64 len);
    int    (*update_time)    (struct inode *, struct timespec *, int);
    int    (*atomic_open)    (struct inode *, struct dentry *,struct file *, unsigned open_flag,
                                                    umode_t create_mode, int *opened);
    int    (*tmpfile)    (struct inode *, struct dentry *, umode_t);
    int    (*set_acl)    (struct inode *, struct posix_acl *, int);
} ____cacheline_aligned;
```

inode_operations 结构体中主要函数指针成员简介如下（/Documentation/filesystems/vfs.txt）：

●**lookup**：在 inode 表示目录文件内容中查找子目录项，并填充 dentry 实例（dentry 实例已创建，实例中包含目录项（文件）名称，用于查找），这是一个非常重要的函数，在此函数内需要创建 inode 实例并初始化。

●**create**：用于 open()和 creat()系统调用在目录文件内创建文件目录项（创建新文件）。

●link，unlink：在系统调用 link()和 unlink()中调用，只有在支持硬链接时才需要此函数；

●symlink：系统调用 symlink()调用此函数，用于支持符号链接；

●**mkdir**：系统调用 mkdir()调用此函数，在支持创建子目录项时需要此函数；

●**rmdir**：系统调用 rmdir()调用此函数，只有在支持删除子目录项时需要此函数；

●**mknode**：系统调用 mknod()调用此函数，用于创建设备文件或命名管道或 socket；

●rename：系统调用 rename()调用此函数，用于重命名文件；

●rename2：与 rename()相比具有额外的标记；

●readlink：系统调用 readlink()调用；

●**follow_link**：用于在打开文件时，读取符号链文件中的内容；

●**permission**：检查当前用户对 inode 表示的文件的访问权限，第二个参数表示所要求的权限，其取值定义在/include/linux/fs.h 头文件内：

```
#define    MAY_EXEC        0x00000001          /*执行权限*/
#define    MAY_WRITE       0x00000002          /*写权限*/
#define    MAY_READ        0x00000004          /*写权限*/
#define    MAY_APPEND          0x00000008
#define    MAY_ACCESS          0x00000010
#define    MAY_OPEN            0x00000020
#define    MAY_CHDIR           0x00000040
```

#define MAY_NOT_BLOCK 0x00000080

如果 inode_operations 实例中未定义此函数，在打开文件时内核将调用通用的 **vfs_permission()**函数进行权限检查，如果定义了 permission()函数，则只调用它而不调用 vfs_permission()函数。

●setatrr：用于设置文件属性，在 chmod()和相关的系统调用中调用；
●getattr：用于获取文件属性，在 stat()和相关的系统调用中调用；
●setxattr：设置文件扩展属性，扩展属性是 “name=vlaue” 字符串，被系统调用 setxattr()调用；
●getxattr：用于获取文件扩展属性，被系统调用 getxattr()调用；
●listxattr：用于列出给定文件所有的扩展属性，被系统调用 listxattr()调用；
●removexattr：用于移除文件的一条扩展属性；
●update_time：用于更新特定时间和 inode 的 i_version；
●atomic_open：在打开路径中最后一个分量（目录项，通常是文件）时调用此函数，只有在最后一个分量是 negative 或需要查找时调用，正常分量用 f_op->open()函数打开；
● tmpfile: 设置 O_TMPFILE 标记位的打开操作最后调用此函数。

3 file_operations

file_operations 结构体主要包含对文件内容的操作函数指针，是内核操作文件的主要接口，是非常重要的一个数据结构。因为外部设备也被视为文件（设备文件），file_operations 结构体也是内核操作外部设备的接口，设备驱动程序的主要工作就是实现 file_operations 结构体实例。

file_operations 结构体与 inode_operations 结构体一样由具体文件系统类型实现，在打开文件创建 inode 实例时将结构体实例指针赋予 inode 实例 i_fop 成员。

file_operations 结构体定义在/include/linux/fs.h 头文件内：

```
struct file_operations {
    struct module *owner;
    loff_t (*llseek) (struct file *, loff_t, int);        /*修改文件当前位置指针*/
    ssize_t (*read) (struct file *, char __user *, size_t, loff_t *);        /*读文件内容*/
    ssize_t (*write) (struct file *, const char __user *, size_t, loff_t *);      /*写文件内容*/
    ssize_t (*read_iter) (struct kiocb *, struct iov_iter *);      /*同步读*/
    ssize_t (*write_iter) (struct kiocb *, struct iov_iter *);      /*同步写*/
    int (*iterate) (struct file *, struct dir_context *);              /*读取目录项*/
    unsigned int (*poll) (struct file *, struct poll_table_struct *);        /*设备文件轮询操作*/
    long (*unlocked_ioctl) (struct file *, unsigned int, unsigned long); /*向文件（设备）发送控制命令*/
    long (*compat_ioctl) (struct file *, unsigned int, unsigned long);
    int (*mmap) (struct file *, struct vm_area_struct *);      /*映射文件内容到进程 VMA*/
    int (*mremap)(struct file *, struct vm_area_struct *);
    int (*open) (struct inode *, struct file *);        /*打开操作*/
    int (*flush) (struct file *, fl_owner_t id);
    int (*release) (struct inode *, struct file *);
    int (*fsync) (struct file *, loff_t, loff_t, int datasync);      /*文件同步函数，写文件函数中调用*/
    int (*aio_fsync) (struct kiocb *, int datasync);
    int (*fasync) (int, struct file *, int);
    int (*lock) (struct file *, int, struct file_lock *);
```

```
    ssize_t (*sendpage) (struct file *, struct page *, int, size_t, loff_t *, int);
    unsigned long (*get_unmapped_area)(struct file *, unsigned long, unsigned long,
                                       unsigned long, unsigned long);
    int (*check_flags)(int);
    int (*flock) (struct file *, int, struct file_lock *);
    ssize_t (*splice_write)(struct pipe_inode_info *, struct file *, loff_t *, size_t, unsigned int);
    ssize_t (*splice_read)(struct file *, loff_t *, struct pipe_inode_info *, size_t, unsigned int);
    int (*setlease)(struct file *, long, struct file_lock **, void **);
    long (*fallocate)(struct file *file, int mode, loff_t offset, loff_t len);
    void (*show_fdinfo)(struct seq_file *m, struct file *f);
#ifndef CONFIG_MMU
    unsigned (*mmap_capabilities)(struct file *);
#endif
};
```

file_operations 结构体主要函数功能如下（/Documentation/filesystems/vfs.txt）：

●**llseek**：修改文件当前位置，lseek()系统调用用于重设文件当前位置；

●**read**：读文件函数，系统调用 read(2)，主要用于对外部设备的操作，文件系统一般不定义；

●**read_iter**：读文件函数，用于基于块设备的文件系统，read()系统调用中调用；

●**write**：写文件函数，系统调用 write(2)，主要用于对设备的操作，文件系统一般不定义；

●**write_iter**：写文件函数，用于基于块设备的文件系统，write()系统调用中调用；

●iterate：读取目录文件内容（目录项）；

●poll：检查文件是否有活动，系统调用 select(2) 和 poll(2)用于查询设备状态；

●**unlocked_ioctl**: ioctl(2) 系统调用，用于向设备发送控制命令；

●**mmap**: 创建文件映射时调用，主要对 VMA 操作结构体成员赋值；

●**open**：打开文件函数，在系统调用 open()打开文件的后期将调用此函数完成打开操作（如设备的激活等），一般可用于初始化文件私有数据“private_data”成员，设备文件常用。

●flush：close(2)系统调用，用于刷新文件；

●release：当打开文件的引用计数清 0 时调用；

●fsync：同步文件内容函数，用于系统调用 fsync(2)，写文件 write()系统调用中也会调用此函数；

●fasync：用于系统调用 fcntl(2)；

●lock：用于系统调用 fcntl(2)，用于 F_GETLK, F_SETLK, F_SETLKW 命令；

●get_unmapped_area： mmap(2) 系统调用，用于在进程虚拟地址空间创建未映射区域；

●check_flags：fcntl(2) 系统调用 F_SETFL 命令；

●flock：flock(2)系统调用；

●splice_read：用于将数据从文件向管道分割，系统调用 splice(2)；

●splice_write：用于将数据从文件向管道分割，系统调用 splice(2)；

●setlease：用于设置或释放文件锁；

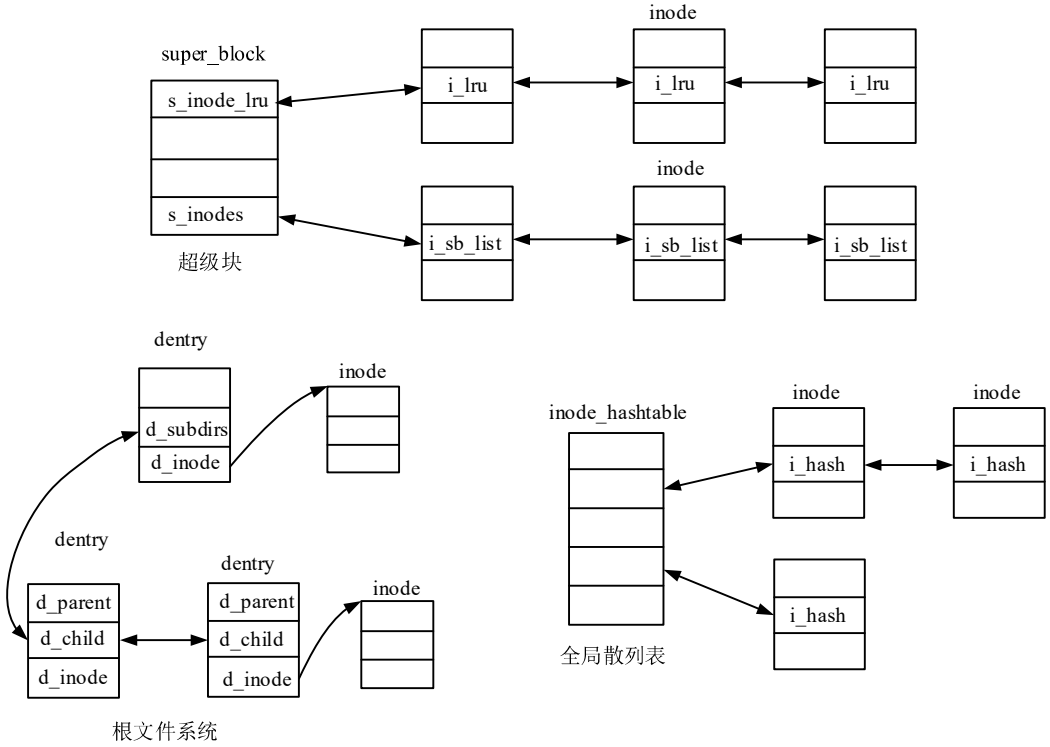
● fallocate：文件预读。

7.3.3 节点操作

虚拟文件系统对节点 inode 实例的管理与目录项 dentry 实例类似，内核为 inode 结构体建立了 slab 缓存和全局散列表 inode_hashtable，实例还添加到超级块中双链表。

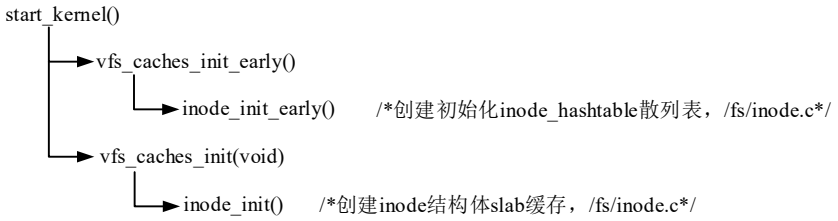
inode 实例与 dentry 实例关联，添加到内核根文件系统结构中，文件系统超级块 super_block 结构体中包含两个 inode 实例双链表，表头分别是 super_block.s_inode_lru（LRU 链表）和 super_block.s_inodes。

inode 实例管理结构如下图所示：



内核在搜索文件时为每个目录项 dentry 实例创建关联的 inode 实例（或关联到现有 inode 实例），并将 inode 实例添加到全局散列表和超级块 s_inodes 双链表中，释放 inode 实例时，暂时不用的实例将被添加到超级块 s_inode_lru 双链表中。

内核在启动阶段调用 inode_init_early()和 inode_init()函数对 inode 实例管理结构进行初始化，函数调用关系如下：



这两个函数都比较简单，都定义在/fs/inode.c 文件内，主要完成 inode 结构体 slab 缓存的创建和全局散列表 inode_hashtable 的创建和初始化。

1 获取节点

在打开文件操作中，查找某一目录项时将调用父目录文件 inode 实例中 inode_operations ->lookup()函数查找目录项，设置 dentry 实例，并创建对应的 inode 实例。具体文件系统类型定义的 lookup()函数在查找到目录项，获取 inode 编号后将调用 **iget_locked**(struct super_block *sb, unsigned long ino)函数创建 inode 实例。

iget_locked()函数定义如下（/fs/inode.c）：

struct inode *iget_locked(struct super_block *sb, unsigned long ino)

/*sb：超级块指针，ino：文件系统中 inode 编号*/

```
{
    struct hlist_head *head = inode_hashtable + hash(sb, ino);     /*inode 实例对应的散列链表头*/
    struct inode *inode;

    spin_lock(&inode_hash_lock);
    inode = find_inode_fast(sb, head, ino); /*在全局散列表中查找 inode 实例，找到则增加引用计数*/
    spin_unlock(&inode_hash_lock);
    if (inode) {
        wait_on_inode(inode);       /*等待节点可用，/include/linux/writeback.h*/
        return inode;               /*返回查找到的 inode 实例指针*/
    }

    /*若未在散列表中查找到 inode 实例，则重新创建 inode 实例*/
    inode = alloc_inode(sb);        /*创建 inode 实例，/fs/inode.c*/
    if (inode) {      /*将 inode 实例添加到管理结构中*/
        struct inode *old;
        spin_lock(&inode_hash_lock);
        old = find_inode_fast(sb, head, ino);   /*再次在全局散列表中查找，看其它进程是否已创建*/
        if (!old) {                 /*初始化实例*/
            inode->i_ino = ino;                 /*编号赋值*/
            spin_lock(&inode->i_lock);
            inode->i_state = I_NEW;         /*设置 inode 状态是新的，需要从块设备读取信息*/
            hlist_add_head(&inode->i_hash, head);       /*将 inode 实例添加到全局散列表*/
            spin_unlock(&inode->i_lock);
            inode_sb_list_add(inode);           /*将 inode 实例添加到超级块 sb->s_inodes 链表*/
            spin_unlock(&inode_hash_lock);
            return inode;       /*返回 inode 实例指针*/
        }
        spin_unlock(&inode_hash_lock);
        destroy_inode(inode);
        inode = old;
        wait_on_inode(inode);
    }
    return inode;
}
```

iget_locked()函数首先在全局散列表中查找 inode 实例，如果找到则增加其引用计数，返回实例指针。如果没有找到，则调用 **alloc_inode(sb)**函数创建新 inode 实例（并初始化），将其添加到全局散列表和超级块结构体中 **s_inodes** 链表，并设置节点状态为 I_NEW，返回 inode 实例指针。

iget_locked()函数返回后，在 inode_operations ->lookup()函数中将判断 inode 实例是否设置了 I_NEW 状态标记，如果是则从文件系统中读取节点信息，填充至 inode 实例，并最后建立与 dentry 实例之间的关联。

iget_locked()函数中调用 alloc_inode(sb)函数创建新 inode 实例，函数定义如下（/fs/inode.c）：

```
static struct inode *alloc_inode(struct super_block *sb)
```

```
{
    struct inode *inode;

    if (sb->s_op->alloc_inode)        /*如果超级块操作结构中定义了分配 inode 实例函数，则调用它*/
        inode = sb->s_op->alloc_inode(sb);      /*超级块操作结构中定义的分配函数*/
    else
        inode = kmem_cache_alloc(inode_cachep, GFP_KERNEL);     /*从 slab 缓存中分配*/

    if (!inode)
        return NULL;

    if (unlikely(inode_init_always(sb, inode))) {     /*初始化 inode 实例，/fs/inode.c*/
        if (inode->i_sb->s_op->destroy_inode)          /*如果初始化失败，销毁 inode 实例*/
            inode->i_sb->s_op->destroy_inode(inode);
        else
            kmem_cache_free(inode_cachep, inode);
        return NULL;
    }
    return inode;       /*返回 inode 实例指针*/
}
```

创建 inode 实例的函数比较简单，如果超级块操作结构中定义了创建 inode 实例的函数则调用它，没有则从 slab 缓存中分配实例，然后调用 **inode_init_always()**函数对 inode 实例进行初始化，将设置各成员的初始值（请读者仔细阅读此函数），最后返回 inode 实例指针。

2 释放节点

当节点 inode 实例不再需要时，或调用 dentry_kill(dentry)函数销毁 dentry 实例时，将调用 iput(inode)函数释放 inode 实例。函数定义如下（/fs/inode.c）：

```
void iput(struct inode *inode)
{
    if (!inode)
        return;
    BUG_ON(inode->i_state & I_CLEAR);
retry:
    if (atomic_dec_and_lock(&inode->i_count, &inode->i_lock)) {       /*如果引用计数减 1 后为 0*/
        if (inode->i_nlink && (inode->i_state & I_DIRTY_TIME)) {
            atomic_inc(&inode->i_count);        /*增加引用计数*/
            inode->i_state &= ~I_DIRTY_TIME;
            spin_unlock(&inode->i_lock);
            trace_writeback_lazytime_iput(inode);
            mark_inode_dirty_sync(inode);   /*设置 inode 状态为 I_DIRTY_SYNC，/inlcude/linux/fs.h*/
            goto retry;
        }
        iput_final(inode);      /*释放 inode 实例，/fs/inode.c*/
```

```
    }
}
```

iput()函数判断 inode 实例引用计数减 1 后是否为 0，如果不是则函数返回，无需释放 inode 实例。若引用计数减 1 后为 0，则调用 iput_final(inode)函数释放 inode 实例。

iput_final(inode)函数定义如下（/fs/inode.c）：

```
static void iput_final(struct inode *inode)
{
    struct super_block *sb = inode->i_sb;
    const struct super_operations *op = inode->i_sb->s_op;     /*超级块操作结构*/
    int drop;

    WARN_ON(inode->i_state & I_NEW);

    if (op->drop_inode)
        drop = op->drop_inode(inode);          /*若超级块操作结构体定义了释放 inode 函数*/
    else
        drop = generic_drop_inode(inode);      /*通用释放函数，/include/linux/fs.h*/

    if (!drop && (sb->s_flags & MS_ACTIVE)) {     /*i_nlink 不为 0 或 inode 还在散列表中*/
        inode->i_state |= I_REFERENCED;
        inode_add_lru(inode);          /*通过 inode->i_lru 添加到 sb->s_inode_lru 双链表*/
        spin_unlock(&inode->i_lock);
        return;           /*函数返回*/
    }

    if (!drop) {        /*i_nlink 不为 0 或 inode 还在散列表中，且挂载文件系统不活跃*/
        inode->i_state |= I_WILL_FREE;
        spin_unlock(&inode->i_lock);
        write_inode_now(inode, 1);          /*立即回写 inode，/fs/fs-writeback.c*/
        spin_lock(&inode->i_lock);
        WARN_ON(inode->i_state & I_NEW);
        inode->i_state &= ~I_WILL_FREE;
    }

    inode->i_state |= I_FREEING;        /*正在释放 inode 实例*/
    if (!list_empty(&inode->i_lru))
        inode_lru_list_del(inode);     /*将 inode 实例从超级块 LRU 链表中移出*/
    spin_unlock(&inode->i_lock);

    evict(inode);        /*等待回写完成，释放 inode 实例，/fs/inode.c*/
}
```

iput_final()函数内首先调用 sb->s_op->drop_inode(inode)或 generic_drop_inode(inode)函数判断是否需要释放 inode 实例，如果不需要释放且当前文件系统处于活跃状态则只需要将 inode 实例添加到超级块 LRU

链表即可。如果需要释放 inode 实例，则调用 evict()函数等待回写完成后，再释放 inode 实例。

iput_final()函数中调用的通用函数 generic_drop_inode()定义在/include/linux/fs.h 头文件内：

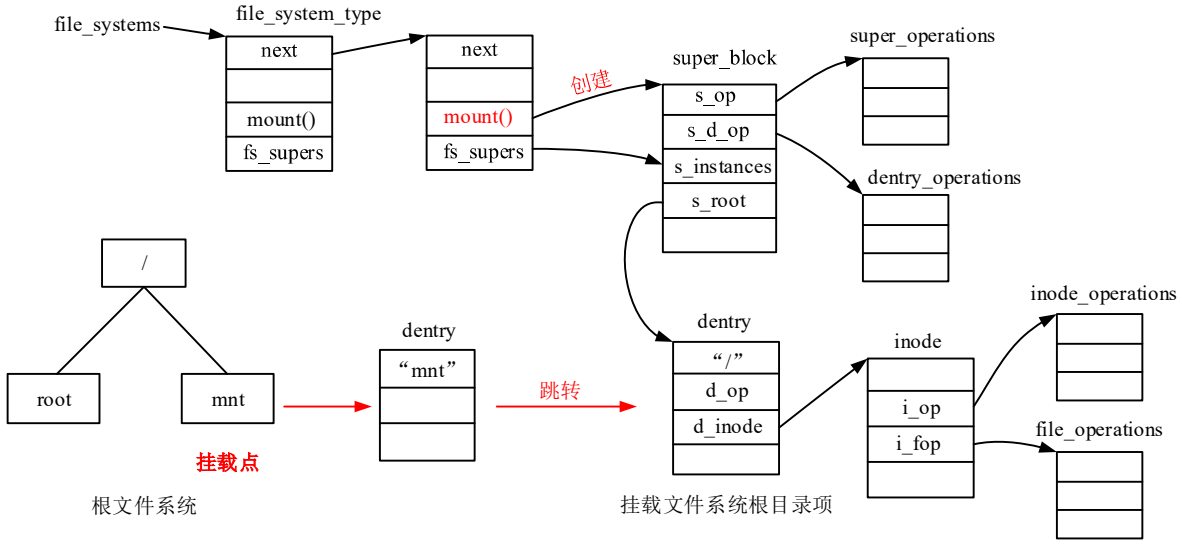
```
static inline int generic_drop_inode(struct inode *inode)
{
    return !inode->i_nlink || inode_unhashed(inode);    /*i_nlink 为 0 或 inode 不在散列表中，返回真*/
}
```

iput_final()函数中调用的 write_inode_now()和 evict(inode)函数涉及到 inode 的回写操作（与块设备同步），等到第 11 章再做介绍。

7.4 文件系统操作

前面介绍的对文件系统目录项、文件的操作都假定文件系统已经挂载到内核根文件系统中的某个目录项，这个目录项称为挂载点。本节将介绍如何将一个外部的文件系统挂载到根文件系统中的目录项。

若要内核能够识别、挂载某一类型的文件系统，内核需支持相关文件系统类型。文件系统类型代码需要定义并注册 file_system_type 结构体实例。在挂载操作中，将调用文件系统类型 file_system_type 实例中的 mount()函数，此函数将创建并设置表示超级块的 super_block 实例，创建表示挂载文件系统根目录项的 dentry 实例及其关联的 inode 实例，如下图所示。挂载操作还将建立内核根文件系统中挂载点与挂载文件系统根目录项之间的关联。



在打开文件，搜索各路径分量的操作中，如果某个目录项是挂载点，将跳转到关联的挂载文件系统根目录项下继续搜索，进入挂载的文件系统，而不进入挂载点原来的子目录树。

如上图中所示，假设某个文件系统被挂载到根文件系统/mnt 目录下。现在要打开/mnt/abc.c 文件，在打开操作中，当搜索到 mnt 目录项时，发现其为挂载点，将跳转到挂载文件系统的根目录项。随后，在此根目录下搜索 abc.c 文件的目录项，也就是在挂载文件系统的根目录下查找 abc.c 文件。

具体文件系统类型代码除了要定义并注册 file_system_type 实例外，还需要定义超级块操作结构、目录项操作结构、节点操作结构、文件操作结构等结构体实例，赋予目录项、节点等实例，实现与虚拟文件系统的对接。

本节先介绍文件系统类型 file_system_type 结构体的定义和实例注册、超级块 super_block 结构体的定义和实例的创建，然后重点介绍挂载文件系统操作的实现，最后简要介绍卸载文件系统操作。

7.4.1 文件系统类型

内核支持的每种文件系统类型由 file_system_type 结构体表示，结构体定义在/include/linux/fs.h 头文件：

```
struct file_system_type {
    const char    *name;        /*文件系统类型名称*/
    int    fs_flags;            /*标记*/
    struct dentry    *(*mount) (struct file_system_type *, int, const char *, void *);    /*挂载函数*/
    void    (*kill_sb)    (struct super_block *);      /*删除超级块实例函数，在卸载文件系统时调用*/
    struct    module    *owner;                        /*模块指针*/
    struct    file_system_type    * next;              /*单链表成员，指向一下个文件系统类型实例*/
    struct    hlist_head    fs_supers;      /*散列链表头，链接已挂载同类型文件系统的超级块实例*/

    struct lock_class_key    s_lock_key;        /*没有选择 LOCKDEP 配置选项为空结构体*/
    struct lock_class_key    s_umount_key;     /*/include/linux/lockdep.h*/
    struct lock_class_key    s_vfs_rename_key;
    struct lock_class_key    s_writers_key[SB_FREEZE_LEVELS];

    struct lock_class_key    i_lock_key;
    struct lock_class_key    i_mutex_key;
    struct lock_class_key    i_mutex_dir_key;
};
```

file_system_type 结构体内主要成员简介如下：

●**fs_flags**：标记成员，取值定义在结构体内：

```
#define    FS_REQUIRES_DEV                1      /*文件系统保存在外部块设备中*/
#define    FS_BINARY_MOUNTDATA            2
#define    FS_HAS_SUBTYPE                 4
#define    FS_USERNS_MOUNT                8      /*可在自定义用户命名空间挂载*/
#define    FS_USERNS_DEV_MOUNT            16
#define    FS_USERNS_VISIBLE              32     /*文件系统必须已可见*/
#define    FS_RENAME_DOES_D_MOVE          32768
```

●**mount()**：文件系统类型定义的挂载函数，第一个参数为指向文件系统类型的指针，第二个参数为挂载标记，第三个参数为文件系统所在块设备文件名称字符串，第四个参数为文件系统私有数据指针。

mount()函数主要完成超级块 super_block 结构体、文件系统根目录项 dentry 和节点 inode 结构体实例的创建和初始化。mount()函数一般调用通用的挂载函数来完成这些工作，例如（/fs/super.c）：

```
struct dentry *mount_bdev(struct file_system_type *fs_type,int flags, const char *dev_name, void *data, \
                          int (*fill_super)(struct super_block *, void *, int))
                           /*用于挂载存储于外部块设备的文件系统*/
struct dentry *mount_nodev(struct file_system_type *fs_type,int flags, void *data,     \
                           int (*fill_super)(struct super_block *, void *, int))
                           /*用于挂载无需外部块设备的文件系统*/
```

其中 fill_super 参数为函数指针，由具体文件系统类型实现，用于填充超级块 super_block 实例，以及创建挂载文件系统根目录项 dentry 和节点 inode 实例。

●**fs_supers**：散列链表头，链接内核挂载的同类型文件系统的超级块实例。内核可以挂载同类型的多个文件系统，例如：硬盘中有两个分区被格式化成 ext2 文件系统，将两个分区都挂载到内核根文件系统后，具有两个 ext2 文件系统的超级块实例，它们被链接到 ext2 文件系统类型实例的 fs_supers 链表中。

●**next**：指向下一个 file_system_type 实例，所有注册的 file_system_type 实例在内核中组成单链表。

内核中所有注册的文件系统类型 file_system_type 实例由单链表管理，表头为 file_systems，定义在/fs/filesystem.c 文件内，注册 file_system_type 实例就是将其插入到单链表。

文件系统类型实例的操作函数简列如下（/fs/filesystem.c）：

●**int register_filesystem(struct file_system_type * fs)**：向内核注册文件系统类型实例，函数只是简单地将 file_system_type 实例添加到 file_systems 单链表，成功返回 0，否则返回错误码。

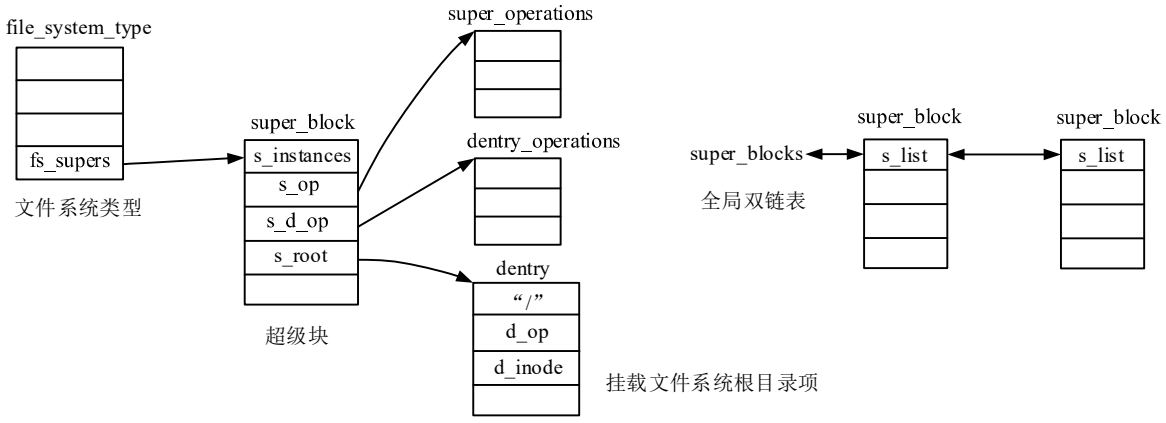
●**int unregister_filesystem(struct file_system_type * fs)**：将 file_system_type 实例从全局单链表中移出，成功返回 0，否则返回错误码。

●**struct file_system_type *get_fs_type(const char *name)**：由文件系统类型名称字符串查找文件系统类型实例。

各文件系统类型的实现代码位于/fs/目录下，每个类型对应一个子目录。例如：ext2 文件系统类型的代码位于/fs/ext2/目录下。文件系统类型代码可在配置内核时选择直接编译入内核，也可以选择编译成模块，在需要的时候加载到内核。文件系统类型的初始化函数在内核初始化子系统或加载模块时调用，主要完成文件系统类型的注册，私有数据结构的创建、初始化等工作。

7.4.2 超级块

内核中每个挂载的文件系统需要创建超级块 super_block 结构体实例。super_block 结构体表示挂载文件系统的整体信息，例如：文件系统类型、数据块的大小和数量等。通常块设备（分区）中文件系统的开头划分出一个指定大小的区域，用于保存此分区文件系统的信息，此区域对应虚拟文件系统中定义的超级块。在挂载文件系统时，内核从此区域读取信息，填充至超级块 super_block 实例中。super_block 实例由文件系统类型 file_system_type 实例和全局双链表 super_blocks 管理，如下图所示。



1 数据结构

超级块 super_block 结构体定义在/include/linux/fs.h 头文件：

```
struct super_block {
    struct list_head    s_list;          /*将实例添加到全局双链表 super_blocks*/
    dev_t               s_dev;           /*所在块设备（分区）设备号*/
```

```
    unsigned char       s_blocksize_bits;       /*数据块大小以 2 为底取对数*/
    unsigned long       s_blocksize;            /*数据块大小，字节数*/
    loff_t              s_maxbytes;             /*最大文件长度，字节数*/
    struct file_system_type  *s_type;           /*指向文件系统类型实例*/
    const struct super_operations  *s_op;       /*超级块操作结构指针*/
    const struct dquot_operations  *dq_op;      /*用户磁盘配额管理*/
    const struct quotactl_ops  *s_qcop;
    const struct export_operations  *s_export_op;
    unsigned long       s_flags;        /*标记，对整个文件系统的控制*/
    unsigned long       s_iflags;       /*内部标记*/
    unsigned long       s_magic;        /*魔数，内核为每种文件系统类型分配唯一的标识数字*/
    struct dentry       *s_root;        /*指向文件系统根目录项 dentry 实例*/
    struct rw_semaphore  s_umount;
    int                 s_count;
    atomic_t            s_active;
#ifdef CONFIG_SECURITY
    void                *s_security;
#endif
    const struct xattr_handler  **s_xattr;    /*扩展属性处理函数*/
    struct list_head s_inodes;              /*文件系统中打开文件的 inode 实例双链表*/

    struct hlist_bl_head  s_anon;           /* anonymous dentries for (nfs) exporting */
    struct list_head s_mounts;    /*挂载 mount 结构体实例双链表，一个分区可以执行多个挂载操作*/
    struct block_device  *s_bdev;       /*块设备（分区）对应块设备数据结构指针*/
    struct backing_dev_info  *s_bdi;    /*后备存储设备信息，结构体定义见第 10 章*/
    struct mtd_info      *s_mtd;        /*MTD 设备信息*/
    struct hlist_node    s_instances;     /*散列表节点，链入文件系统类型散列链表，表头 fs_supers*/
    unsigned int         s_quota_types;  /*Bitmask of supported quota types */
    struct quota_info    s_dquot;         /*Diskquota specific options */
    struct sb_writers    s_writers;
    char  s_id[32];                 /*名称*/
    u8  s_uuid[16];                 /* UUID */

    void                *s_fs_info;         /*具体文件系统私有数据指针*/
    unsigned int     s_max_links;
    fmode_t          s_mode;
    u32              s_time_gran;
    struct mutex  s_vfs_rename_mutex;       /* Kludge */
    char  *s_subtype;
    char __rcu  *s_options;
    const struct dentry_operations  *s_d_op;  /*赋予其下所有目录项 dentry 实例的 d_op 成员*/
    int  cleancache_poolid;
```

```
    struct shrinker    s_shrink;          /*slab 缓存收缩器，用于页面回收机制，见第 11 章*/
    atomic_long_t    s_remove_count;
    int    s_readonly_remount;
    struct workqueue_struct    *s_dio_done_wq;    /*直接读写操作工作队列，详见第 11 章*/
    struct hlist_head    s_pins;          /*散列链表头*/

    struct list_lru          s_dentry_lru    ____cacheline_aligned_in_smp;    /*dentry 实例 LRU 链表*/
    struct list_lru          s_inode_lru    ____cacheline_aligned_in_smp;    /*inode 实例 LRU 链表*/
    struct rcu_head          rcu;
    int    s_stack_depth;
};
```

super_block 结构体中主要成员简介如下：

●**s_list**：双链表成员，将实例添加到全局双链表 super_blocks。

●**s_blocksize，s_blocksize_bits**：表示分区（块设备）数据块的大小，即读写块设备的数据单位，s_blocksize 表示数据块大小，字节数，s_blocksize_bits 表示数据块大小字节数以 2 为底取对数，即 $2^{s\_blocksize\_bits}$=s_blocksize。在对块设备进行读写操作时需要此数据，数据块大小来源于块设备的请求队列，在挂载文件系统创建超级块实例之后，从请求队列中获取此数值。

●**s_op**：超级块操作 super_operations 结构体指针，结构体定义后面再详细介绍。

●**s_flags**：标记成员，取值定义在/include/uapi/linux/fs.h 头文件内，表示整个文件系统的挂载属性，例如：

```
#define    MS_RDONLY          1    /*只读挂载 */
#define    MS_NOSUID          2    /*忽略 suid 和 sgid 位*/
...
```

这里的标记取值同 mount()系统调用中的标记参数取值，后面将详细介绍。

●**s_root**：指向挂载文件系统根目录项 dentry 实例。

●**s_inodes**：双链表成员，链接文件系统中打开文件的 inode 实例。

●**s_mounts**：双链表成员，链接挂载 mount 结构体实例，文件系统可以挂载到多个挂载点，每个 mout 结构体实例表示一次挂载操作，结构体定义后面再介绍。

●**s_bdev**：block_device 结构体指针，表示文件系统所在块设备数据结构。

●**s_bdi**：backing_dev_info 结构体指针，文件系统所在后备存储设备的底层信息和状态，结构体实例位于请求队列 request_queue 实例中。在块设备驱动程序中创建请求队列时，对其 backing_dev_info 结构体成员进行初始化，详见第 10 章。基于后备存储设备的文件系统，在调用 sget()函数创建超级块 super_block 实例时，设置超级块的函数指针赋值 set_bdev_super()，此函数内将会使 s_bdi 成员指向块设备请求队列中 backing_dev_info 结构体成员。

●**s_instances**：散列链表节点成员，将实例链接到对应文件系统类型散列链表，表头为 fs_supers 成员。

●**s_fs_info**：文件系统私有数据结构指针，通常在文件系统类型的挂载函数中创建数据结构实例，赋予此成员。

●**s_d_op**：目录项操作 dentry_operations 结构体指针，此成员在文件系统类型的挂载函数中赋值，并传递给所有本文件系统下的 dentry 实例。

● **s_dentry_lru**：dentry 实例 LRU 链表，缓存暂时不用的实例。

●**s_inode_lru**：打开文件 inode 实例 LRU 链表，缓存暂时不用的实例。

2 超级块操作结构

super_block 结构体中包含两个虚拟文件系统的接口数据结构，dentry_operations 和 super_operations。dentry_operations 表示目录项操作结构，在介绍目录项时介绍过了。在创建 dentry 实例时，将超级块指向的 dentry_operations 实列将赋予 dentry 实例。

super_operations 结构体表示超级块操作结构，其中包含对整个文件系统的操作、控制函数指针，如：分配 inode、销毁 inode、同步文件（文件内容写入块设备）等。

super_operations 结构体定义在/include/linux/fs.h 头文件：

```
struct super_operations {
   struct inode *(*alloc_inode)(struct super_block *sb);      /*创建 inode 实例*/
   void (*destroy_inode)(struct inode *);           /*销毁 alloc_inode()函数创建的实例*/

   void (*dirty_inode) (struct inode *, int flags);     /*标记 inode 脏，表示 inode 需要回写*/
   int (*write_inode) (struct inode *, struct writeback_control *wbc);      /*回写 inode 元数据*/
   int (*drop_inode) (struct inode *);          /*释放 inode 时调用*/
   void (*evict_inode) (struct inode *);      /*在释放 inode 的 evict(inode)函数内调用此函数*/
   void (*put_super) (struct super_block *);
   int (*sync_fs)(struct super_block *sb, int wait);      /*同步文件系统中所有脏 inode*/
   int (*freeze_super) (struct super_block *);
   int (*freeze_fs) (struct super_block *);
   int (*thaw_super) (struct super_block *);
   int (*unfreeze_fs) (struct super_block *);
   int (*statfs) (struct dentry *, struct kstatfs *);        /*获取文件系统统计量*/
   int (*remount_fs) (struct super_block *, int *, char *);     /*重新挂载文件系统*/
   void (*umount_begin) (struct super_block *);                /*卸载文件系统前调用*/

   int (*show_options)(struct seq_file *, struct dentry *);
   int (*show_devname)(struct seq_file *, struct dentry *);
   int (*show_path)(struct seq_file *, struct dentry *);
   int (*show_stats)(struct seq_file *, struct dentry *);
#ifdef CONFIG_QUOTA
   ssize_t (*quota_read)(struct super_block *, int, char *, size_t, loff_t);
   ssize_t (*quota_write)(struct super_block *, int, const char *, size_t, loff_t);
   struct dquot **(*get_dquots)(struct inode *);
#endif
   int (*bdev_try_to_free_page)(struct super_block*, struct page*, gfp_t);
   long (*nr_cached_objects)(struct super_block *, struct shrink_control *);
   long (*free_cached_objects)(struct super_block *,struct shrink_control *);
};
```

super_operations 结构体中常用的函数成员简介如下（/Documentation/filesystems/vfs.txt）：

●**alloc_inode**：创建 inode 实例，如果定义了此函数，创建 inode 实例的 alloc_inode(sb)函数将调用此函数创建 inode 实例，而不是从 slab 缓存中分配 inode 实例。通常此函数创建的数据结构实例是 inode 结构体的包装器，函数返回其中的 inode 结构体成员指针。

●**destroy_inode**：与 alloc_inode()函数对应，用于释放其创建的 inode 实例，只有在定义了 alloc_inode()

函数时才需要此函数.

●**dirty_inode**：标记 inode 为脏，需要回写 inode（与块设备同步）。

●**write_inode**：将 inode 实例回写到块设备，第二个参数表示写是否是同步，不是所有文件系统都会检查该标记。

●**drop_inode**：当最后一次访问 inode 结束时，调用此函数。

●**put_super**：当虚拟文件系统要释放 super_block 时调用此函数。

●**sync_fs**：回写所有脏 inode（文件同步），第二个参数表示是否等待回写完成。

●**freeze_fs**：当虚拟文件系统正在锁定一个文件系统并强制其进入一致状态时调用此函数，此方法目前被 Logical Volume Manager (LVM)采用。

●**unfreeze_fs**：当虚拟文件系统解锁文件系统并使其重新可写时调用此函数。

●**statfs**：获取文件系统统计信息时调用此函数。

●**remount_fs**：重新装载文件系统时调用此函数。

●**umount_begin**：卸载文件系统前调用此函数。

●quota_read，quota_write：分别用于 VFS 读写文件系统 quota 文件。

●**nr_cached_objects**：超级块缓存收缩函数调用此函数，返回可释放对象的数量。

●**free_cache_objects**：超级块缓存收缩函数调用此函数，用于扫描可释放对象，并释放它们，此函数必须定义 nr_cached_objects 函数。

3 获取超级块实例

在文件系统类型 file_system_type 实例的挂载函数 mount()中将创建并填充（或查找）文件系统超级块 super_block 实例。由于同一个文件系统可以同时挂载到多个挂载点，在本次挂载前文件系统可能已经执行了挂载操作，文件系统超级块实例可能已经存在，因此在挂载时先查找超级块实例，若不存在再创建。

文件系统类型的挂载函数 mount()中通常调用 **sget()**函数，查找或创建超级块 super_block 实例。sget()函数首先查找是否已经存在所挂载文件系统的超级块实例，如果存在则无需再创建，只需增加其引用计数即可，如果不存在则重新创建。

sget()函数定义如下（/fs/super.c）：

```
struct super_block *sget(struct file_system_type *type,int (*test)(struct super_block *,void *),     \
                                        int (*set)(struct super_block *,void *),int flags,void *data)
/*
*type：文件系统类型指针，
*test：检查超级块实例是否是本次挂载文件系统的超级块，是则返回真，否则返回假，
*set：设置新创建超级块实例的函数指针，
*flags：挂载标记，mount()系统调用传递的挂载标记，见下文，
*data：test 和 set 函数第二个参数，具有后备存储介质的文件系统，为 block_device 实例指针。
*/
{
    struct super_block *s = NULL;
    struct super_block *old;
    int err;

retry:
    spin_lock(&sb_lock);
    if (test) {          /*如果定义了 test 函数，则先在文件系统类型实例中查找超级块实例*/
```

```
        hlist_for_each_entry(old, &type->fs_supers, s_instances) {
            if (!test(old, data))          /*检查超级块实例是否是挂载文件系统的超级块*/
                continue;
            if (!grab_super(old))          /*如果是，增加实例引用计数，/fs/super.c*/
                goto retry;
            if (s) {
                up_write(&s->s_umount);
                destroy_super(s);
                s = NULL;
            }
            return old;       /*如果找超级块实例，返回实例指针*/
        }
    }

    if (!s) {        /*没有定义 test()函数或不存在所需超级块实例*/
        spin_unlock(&sb_lock);
        s = alloc_super(type, flags);
                    /*从 slab 通用缓存分配 super_block 实例，并初始化其成员，/fs/super.c*/
        if (!s)
            return ERR_PTR(-ENOMEM);
        goto retry;       /*返回 retry，再次执行检查函数，
                    *新创建实例还没有添加到文件系统类型实例中的散列链表，
                    *此处检查其它进程有没有创建并添加 super_block 实例。
                    */
    }

    /*super_block 实例已经创建且 test 测试不成功*/
    err = set(s, data);              /*设置 super_block 实例*/
    ...
    s->s_type = type;                /*指向文件系统类型实例*/
    strlcpy(s->s_id, type->name, sizeof(s->s_id));        /*复制名称*/
    list_add_tail(&s->s_list, &super_blocks);             /*添加到全局双链表末尾*/
    hlist_add_head(&s->s_instances, &type->fs_supers);    /*添加到文件系统类型中散列链表*/
    spin_unlock(&sb_lock);
    get_filesystem(type);
    register_shrinker(&s->s_shrink);      /*注册收缩器，用于页面回收*/
    return s;
}
```

sget()函数执行的流程如下：

（1）如果定义了 test()函数，则查找文件系统类型中是否已经存在此文件系统的超级块实例，如果已经存在则增加其引用计数后返回实例指针。

（2）如果定义了 test()函数但没有找到匹配的实例，或者没有定义 test()函数，则调用 **alloc_super()**函数创建新超级块实例，并进行初始化，跳转至步骤（1）再次执行测试操作。如果其它进程已经创建了超

级块实例，则释放本次创建的实例，函数返回其它进程创建实例的指针。如果其它进程没有创建超级块实例，或没有定义 test()函数，即新创建超级块实例有效，执行步骤（3）。

（3）调用 set(s, data)函数设置新创建超级块实例，将其添加到文件系统类型散列链表和全局双链表，注册缓存收缩器、关联请求队列中 backing_dev_info 实例等。

7.4.3 挂载文件系统

内核对某一实际文件系统（磁盘分区）最常用的操作就是挂载和卸载。挂载操作在内核中为文件系统创建（或查找）超级块 super_block 结构体实例，创建文件系统根目录项对应 dentry 和 inode 结构体实例，并通过挂载 mount 结构体将根文件系统中挂载点目录项与挂载文件系统根目录项关联起来。

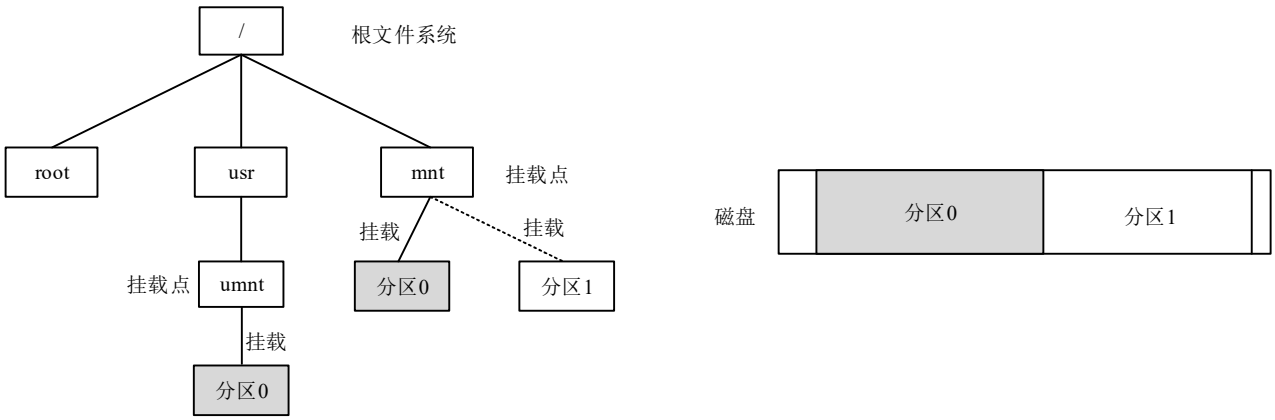
内核文件搜索路径（目录项）至挂载点时，发现目录项为挂载点，将查找挂载点对应的 mount 实例，跳转至 mount 实例关联的挂载文件系统根目录项下继续搜索，从而进入挂载的文件系统。

1 挂载概述

虚拟文件系统通过目录项 dentry 实例组成的根文件系统管理内核所有的文件系统和文件。具体文件系统根目录项需要关联到内核根文件系统中某一目录项（挂载点）才能接入根文件系统。

如下图所示，将磁盘分区 0 的文件系统根目录项与根文件系统中/mnt 目录项建立关联，称之为挂载，/mnt 目录项称之为挂载点。分区 0 文件系统可同时挂载到多个挂载点，如其根目录项可与/mnt 目录项关联，也可同时与/usr/umnt 目录项关联，内核可以通过这两个路径进入分区 0 文件系统。

根文件系统中的挂载点也可以同时挂载多个文件系统，例如：/mnt 目录项可同时与分区 0 和分区 1 文件系统根目录项建立关联。当打开文件搜索路径至/mnt 目录项时，将跳转至最近挂载的文件系统根目录项下继续搜索。假如，分区 1 在分区 0 之后被挂载，则分区 1 文件系统可见，分区 0 被隐藏，当分区 1 文件系统被卸载时，分区 0 文件系统将自动可见。



挂载操作需要为文件系统创建（或查找）超级块 super_block、根目录项 dentry 和 inode 结构体实例，这些结构体在前面都介绍过了，现在还有一个重要的关节没有打通，那就是如何建立挂载点目录项与挂载文件系统根目录项之间的关联？

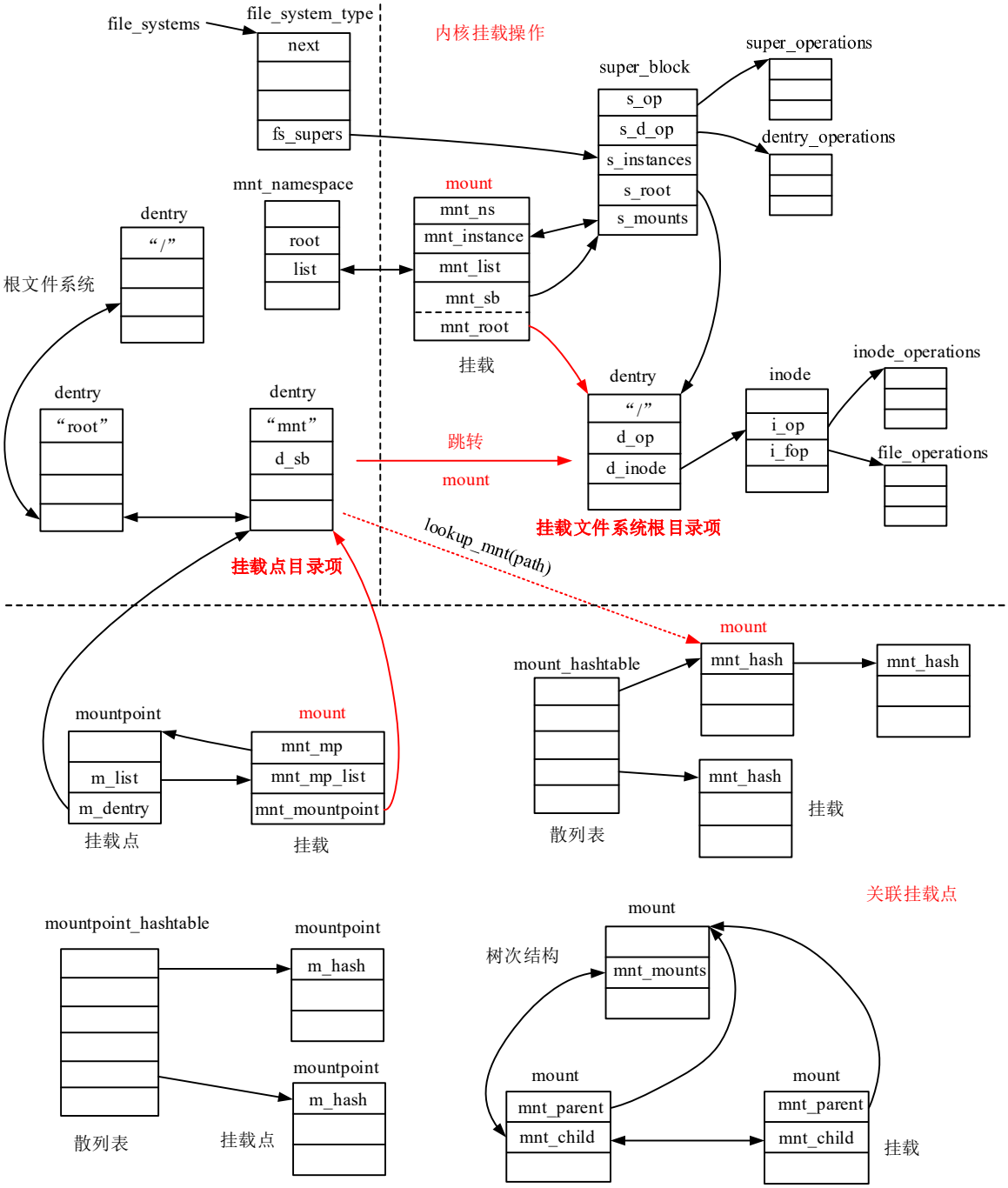
内核定义了 mountpoint 结构体表示根文件系统中的一个挂载点，挂载点对应根文件系统中一个 dentry 实例，内核还定义了 mount 结构体表示一次挂载操作（挂载描述符）。用户进程通过 mount()系统调用实现文件系统的挂载，下图示意了在/mnt 挂载点执行挂载操作后，内核创建的数据结构实例及组织关系。

mount()挂载系统调用主要分两步：

（1）内核挂载：创建（或查找）文件系统超级块 super_block、根目录项 dentry 和 inode 结构体实例，并创建描述本次挂载操作的 mount 结构体实例。mount 实例添加到超级块实例 s_mounts 成员链表的末尾，并与挂载文件系统根目录项 dentry 实例建立关联。因为一个文件系统可以挂载到多个挂载点，因此执行挂

载操作时，其超级块可能已经存在了。

（2）关联挂载点：创建（或查找）挂载点 mountpoint 结构体实例，并添加到全局散列表。mountpoint 实例关联到挂载点 dentry 实例（根文件系统中目录项），并将挂载 mount 实例添加到 mountpoint 实例中链表和全局散列表，建立 mount 实例与挂载点 dentry 实例之间的联系。



若支持挂载命名空间，一个挂载点的挂载，可能在多个挂载命名空间中有副本，因此 mountpoint 实例中包含一个挂载 mount 实例的链表，链接挂载点在各挂载命名空间下的 mount 实例。

执行完这两步，通过 mount 实例就建立了挂载点 dentry 实例和挂载文件系统根目录项 dentry 实例之间的联系。当内核打开文件搜索路径到达挂载点时（挂载点 dentry 实例设置 DCACHE_MOUNTED 标记位），将调用函数 **lookup_mnt(path)**，在 mount 实例全局散列表中查找第一个关联到挂载点 dentry 实例且符合条件的 mount 实例，搜索路径随后进入 mount 实例关联的挂载文件系统根目录项继续搜索。

同一文件系统可执行多个挂载操作，挂载到多个挂载点，因此超级块实例中包含 mount 实例链表，每

个实例表示一次挂载操作，通过 mount 实例可将文件系统挂载到多个挂载点。

mount 实例由全局散列表管理，计算散列值函数的参数是挂载点 dentry 实例指针和挂载点所在文件系统挂载信息 vfsmount 实例指针（mount 实例中成员）。新 mount 实例，将被插入到散列链表的头部，在内核搜索文件路径中调用 lookup_mnt(path)函数查找挂载点对应 mount 实例时，也是从链表头部开始查找第一个符合条件的 mount 实例，因此最近的挂载将覆盖同一挂载点之前的挂载。mount 实例根据对应挂载点在根文件系统中的层次关系还构成父子层次关系，称为挂载树。

2 数据结构

mountpoint 结构体表示根文件系统中的挂载点，定义如下（/fs/mount.h）：

```
struct mountpoint {
    struct hlist_node    m_hash;     /*散列链表节点成员，将实例添加到全局散列表*/
    struct dentry *m_dentry;         /*指向挂载点 dentry 实例（根文件系统中目录项）*/
    struct hlist_head m_list;        /*链接不同挂载命名空间中 mount 实例*/
    int m_count;                     /*挂载点挂载操作的次数*/
};
```

mountpoint 结构体成员简介如下：

●m_hash：散列链表节点成员，将 mountpoint 实例添加到全局散列表 mountpoint_hashtable。

●**m_dentry**：指向挂载点 dentry 实例，根文件系统中的目录项，不是挂载文件系统的根目录项。

●**m_list**：链接同一挂载点在不同挂载命名空间下的挂载 mount 实例。

●m_count：挂载点执行挂载操作的次数。

mountpoint 结构体实例由全局散列表管理，内核在初始化虚拟文件系统时为其创建全局散列表并初始化。

mount 结构体表示一次挂载操作，也称为挂载描述符，用于建立挂载点目录项与挂载文件系统根目录项的关联，以实现跳转。mount 结构体定义在/fs/mount.h 头文件：

```
struct mount {
    struct hlist_node    mnt_hash;      /*散列链表节点成员，将实例添加到全局散列表*/
    struct mount    *mnt_parent;        /*父 mount 实例*/
    struct dentry    *mnt_mountpoint;   /*挂载点 dentry 实例指针（根文件系统目录项）*/
    struct vfsmount    mnt;             /*vfsmount 结构体实例，非指针，表示在 VFS 中的挂载信息*/
    union {
        struct rcu_head    mnt_rcu;     /*回调函数*/
        struct llist_node    mnt_llist;
    };
#ifdef CONFIG_SMP
    struct mnt_pcp __percpu *mnt_pcp;
#else
    int mnt_count;
    int mnt_writers;
#endif
    struct list_head    mnt_mounts;     /*子 mount 实例双链表头*/
    struct list_head    mnt_child;      /*链接兄弟 mount 实例，添加到父实例的 mnt_mounts 双链表*/
    struct list_head    mnt_instance;   /*添加到超级块中双链表，表头为 sb->s_mounts */
```

```
    const char    *mnt_devname;        /*文件系统所在块设备文件名称，如：/dev/dsk/hda1 */
    struct list_head    mnt_list;      /*将实例添加到挂载命名空间双链表*/
    struct list_head    mnt_expire;    /*用于特定于文件系统的过期链表*/
    struct list_head    mnt_share;     /*共享挂载，链接对等组内挂载实例*/
    struct list_head    mnt_slave_list;   /*从属挂载双链表头*/
    struct list_head    mnt_slave;     /*挂载为从属挂载时，添加到主挂载的 mnt_slave_list 双链表*/
    struct mount    *mnt_master;       /*主挂载指针*/
    struct mnt_namespace    *mnt_ns;      /*指向所属挂载命名空间，见下一小节*/
    struct mountpoint    *mnt_mp;         /*挂载点结构体指针*/
    struct hlist_node    mnt_mp_list ;    /*将实例添加到挂载点的 m_list 链表*/
#ifdef CONFIG_FSNOTIFY
    struct hlist_head mnt_fsnotify_marks;
    __u32 mnt_fsnotify_mask;
#endif
    int    mnt_id;                  /*ID 标记*/
    int    mnt_group_id;            /*对等组 ID，用于共享挂载，表示同一挂载点的共享挂载*/
    int    mnt_expiry_mark;         /*标记挂载时否过期，true 表示过期*/
    struct hlist_head    mnt_pins;
    struct fs_pin    mnt_umount;
    struct dentry    *mnt_ex_mountpoint;
};
```

mount 结构体实例在内核中组成父子的层次结构，同时由全局散列表管理，结构体中主要成员简介如下（挂载命名空间相关的成员下一小节再介绍）：

●**mnt_hash**：散列链表节点成员，将实例添加到全局散列表 mount_hashtable。

●**mnt_mountpoint**：指向挂载点 dentry 实例，注意不是挂载文件系统根目录项 dentry 实例，而是根文件系统中挂载点的 dentry 实例。

●**mnt_instance**：双链表节点成员，将 mount 实例链入超级块中的双链表，链表头为 sb->s_mounts。

●**mnt_mp**：指向挂载点 mountpoint 实例。

●**mnt_mp_list**：散列链表节点成员，将实例链接到挂载点 mountpoint 实例的 m_list 链表。

●**mnt_list**：双链表节点成员，将 mount 实例链接到挂载命名空间 mnt_namespace 实例中的双链表。挂载也支持命名空间，内核在启动阶段挂载 rootfs 文件系统时（初始根文件系统），调用 **create_mnt_ns(mnt)** 创建初始挂载命名空间 mnt_namespace 实例。如果用户不指示创建新挂载命名空间 mnt_namespace 实例，内核中所有挂载将由初始挂载命名空间 mnt_namespace 实例管理，在关联挂载点的 commit_tree()函数中，mount 实例将添加到 mnt_namespace 实例中管理的双链表。

●**mnt**：vfsmount 结构体成员，用于建立 mount 实例与挂载文件系统之间的关联，结构体定义在头文件/include/linux/mount.h：

```
struct vfsmount {
    struct dentry    *mnt_root;       /*指向挂载文件系统根目录项 dentry 实例*/
    struct super_block    *mnt_sb;    /*指向文件系统超级块实例*/
    int mnt_flags;                    /*内部表示的挂载标记*/
};
```

mnt_flags 标记成员取值定义如下（/include/linux/mount.h）：

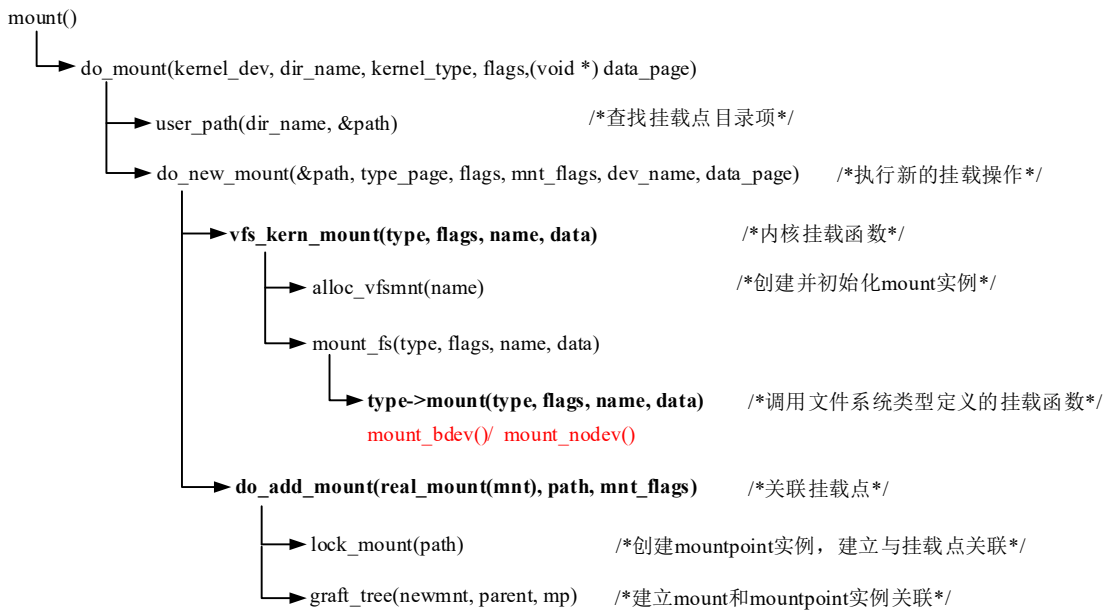
```
#define    MNT_NOSUID              0x01      /*执行文件时忽略文件模式中 suid 和 sgid 位*/
```

```
#define   MNT_NODEV              0x02      /*文件系统不在外部存储设备中*/
#define   MNT_NOEXEC             0x04      /*不可以执行文件系统中文件*/
#define   MNT_NOATIME            0x08
#define   MNT_NODIRATIME         0x10
#define   MNT_RELATIME           0x20
#define   MNT_READONLY           0x40    /*只读挂载*/

#define   MNT_SHRINKABLE       0x100    /*专用于 NFS 和 AFS 文件系统，用于标记子装载*/
#define   MNT_WRITE_HOLD       0x200

#define   MNT_SHARED           0x1000          /*共享挂载*/
#define   MNT_UNBINDABLE       0x2000          /*不可绑定挂载*/
#define   MNT_SHARED_MASK      (MNT_UNBINDABLE)
#define   MNT_USER_SETTABLE_MASK     (MNT_NOSUID | MNT_NODEV | MNT_NOEXEC \
                  | MNT_NOATIME | MNT_NODIRATIME | MNT_RELATIME \
                  | MNT_READONLY)
#define   MNT_ATIME_MASK    (MNT_NOATIME | MNT_NODIRATIME | MNT_RELATIME )

#define   MNT_INTERNAL_FLAGS     (MNT_SHARED | MNT_WRITE_HOLD | MNT_INTERNAL | \
                        MNT_DOOMED | MNT_SYNC_UMOUNT | MNT_MARKED)

#define   MNT_INTERNAL           0x4000            /*内核发起的挂载操作*/

#define   MNT_LOCK_ATIME         0x040000
#define   MNT_LOCK_NOEXEC        0x080000
#define   MNT_LOCK_NOSUID        0x100000
#define   MNT_LOCK_NODEV         0x200000
#define   MNT_LOCK_READONLY      0x400000
#define   MNT_LOCKED             0x800000       /*锁定，挂载文件系统不可卸载*/
#define   MNT_DOOMED             0x1000000
#define   MNT_SYNC_UMOUNT        0x2000000
#define   MNT_MARKED             0x4000000
#define   MNT_UMOUNT             0x8000000
```

mnt_flags 标记成员值大部分通过 mount()系统调用的标记参数转换得来，详见下文。

3 用户挂载

用户进程通过 mount()系统调用挂载具体文件系统，函数调用关系如下图所示：



上图所示的是执行新挂载操作的函数调用关系，mount()系统调用中首先在根文件系统中查找挂载点 dentry 实例，然后调用内核挂载函数创建超级块 super_block、根目录项 dentry 和 inode 结构体实例以及 mount 实例，并建立各实例之间的关联。文件系统类型中的挂载函数通常为 **mount_bdev()**或 **mount_nodev()**等，前者用于基于外部存储介质的文件系统类型（如 ext2），后者用于不需要外部存储介质的文件系统类型（如基于内存盘、内核数据结构的文件系统类型）。

mount()系统调用最后调用关联挂载点函数建立 mount 实例与挂载点 mountpoint 实例、挂载点 dentry 实例之间的关联，并将 mount 实例插入到全局散列表，挂载操作完成。

mount()系统调用参数中需包含挂载点的路径名、文件系统所在存储介质的设备文件名、文件系统类型、挂载标记等信息。在介绍系统调用实现之前，先了解一下系统调用传递的挂载标记参数取值，定义在头文件/include/uapi/linux/fs.h：

```
#define   MS_RDONLY          1      /*只读方式挂载*/
#define   MS_NOSUID          2      /*执行文件时忽略文件模式 suid 和 sgid  位*/
#define   MS_NODEV           4      /*不能访问文件系统中的设备文件*/
#define   MS_NOEXEC          8      /*不能执行文件系统中的文件*/
#define   MS_SYNCHRONOUS     16     /*写时同步*/
#define   MS_REMOUNT         32     /*对已经挂载的文件系统修改标记*/
#define   MS_MANDLOCK        64     /*允许对文件系统中文件加强制锁*/
#define   MS_DIRSYNC         128    /*目录项修改是同步的*/
#define   MS_NOATIME         1024   /*不需要修改访问时间*/
#define   MS_NODIRATIME      2048   /*不要修改目录访问时间*/
#define   MS_BIND            4096   /*绑定挂载*/
#define   MS_MOVE            8192   /*移动现有挂载*/
#define   MS_REC             16384  /*将挂载操作（类型改变）应用于当前挂载下的子树*/
#define   MS_VERBOSE         32768  /*不再使用的标记位*/
#define   MS_SILENT          32768
#define   MS_POSIXACL        (1<<16)  /* VFS does not apply the umask */
#define   MS_UNBINDABLE      (1<<17)  /*改为不可绑定挂载*/
#define   MS_PRIVATE         (1<<18)  /*改为私有挂载*/
#define   MS_SLAVE           (1<<19)  /*改为从属挂载*/
```

```
#define   MS_SHARED          (1<<20)   /*改为共享挂载*/
#define   MS_RELATIME        (1<<21)   /* Update atime relative to mtime/ctime. */
#define   MS_KERNMOUNT       (1<<22)   /*内核发起的挂载操作*/
#define   MS_I_VERSION       (1<<23)   /* Update inode I_version field */
#define   MS_STRICTATIME     (1<<24)   /* Always perform atime updates */
#define   MS_LAZYTIME        (1<<25)   /* Update the on-disk [acm]times lazily */

/*以下是超级块内部标记*/
#define   MS_NOSEC        (1<<28)
#define   MS_BORN         (1<<29)
#define   MS_ACTIVE       (1<<30)
#define   MS_NOUSER       (1<<31)

#define   MS_RMT_MASK  (MS_RDONLY|MS_SYNCHRONOUS|MS_MANDLOCK|MS_I_VERSION|\
                                                              MS_LAZYTIME)

/*古老的挂载标记魔数和掩码，现在不再使用*/
#define   MS_MGC_VAL    0xC0ED0000
#define   MS_MGC_MSK    0xffff0000

mount()系统调用在/fs/namespace.c 文件内实现，代码如下：
SYSCALL_DEFINE5(mount, char __user *, dev_name, char __user *, dir_name,      \
                                                char __user *, type, unsigned long, flags, void __user *, data)
/*
*dev_name：文件系统所在存储介质设备文件名，如/dev/hd0；
*dir_name：挂载点目录，如/mnt；
*type：文件系统类型字符串，如“ext2”；
*flags：挂载操作标记，会传递给超级块实例，见上文 sget()函数；
*data：文件系统私有数据结构，大小不能超过一页，由文件系统类型的挂载函数使用。
*/
{
    int ret;
    char *kernel_type;       /*保存文件系统类型字符串*/
    char *kernel_dev;        /*保存设备文件名称*/
    unsigned long data_page;      /*保存私有数据内存地址*/

    kernel_type = copy_mount_string(type);        /*复制 type 参数（文件系统类型）至内核空间*/
    ...     /*错误处理*/
    kernel_dev = copy_mount_string(dev_name);     /*复制 dev_name 参数（设备文件名）至内核空间*/
    ...     /*错误处理*/
    ret = copy_mount_options(data, &data_page);        /*复制私有数据至内核空间*/
    ...     /*错误处理*/
    ret = do_mount(kernel_dev, dir_name, kernel_type, flags,(void *) data_page);
```

```
                                                    /*执行挂载操作，/fs/namespace.c*/
        free_page(data_page);        /*释放暂存的数据*/
    out_data:
        kfree(kernel_dev);
    out_dev:
        kfree(kernel_type);
    out_type:
        return ret;        /*成功返回 0*/
    }
```

mount()系统调用内将用户进程传递的文件系统类型参数、存储设备文件名称参数和私有数据复制到内核空间（没有复制挂载点目录参数），然后调用 do_mount()函数执行实际的挂载操作。

在介绍 do_mount()函数前，先了解一下 path 结构体的定义，结构体在挂载操作中用于传递 VFS 中挂载点的信息，结构体定义在/include/linux/path.h 头文件内：

```
struct path {
    struct vfsmount    *mnt;
                        /*指向 vfsmount 实例，mount.mnt 成员（挂载点所在文件系统的挂载信息）*/
    struct dentry * dentry;          /*指向挂载点 dentry 实例（根文件系统中目录项）*/
};
```

do_mount()函数在/fs/namespace.c 文件内实现，代码如下：

```
long do_mount(const char *dev_name, const char __user *dir_name,      \
                                        const char *type_page, unsigned long flags, void *data_page)
{
    struct path path;        /*path 结构体实例*/
    int retval = 0;
    int mnt_flags = 0;        /*挂载标记*/

    /*去掉标记参数中的魔数*/
    if ((flags & MS_MGC_MSK) == MS_MGC_VAL)
        flags &= ~MS_MGC_MSK;

    /*安全性检查*/
    if (data_page)
        ((char *)data_page)[PAGE_SIZE - 1] = 0;        /*用户数据页最后一个字节设为 0*/

    /*查找挂载点*/
    retval = user_path(dir_name, &path);    /*path 保存挂载点目录项信息，/include/linux/namei.h*/
    ...      /*错误处理*/

    retval = security_sb_mount(dev_name, &path,type_page, flags, data_page);
                                                    /*没有选择 SECURITY，返回 0*/
    if (!retval && !may_mount())        /*进程需具有 CAP_SYS_ADMIN 能力，才能执行挂载操作*/
```

```
        retval = -EPERM;
...       /*错误处理*/

/*系统调用挂载标记转成 mount 实例中的挂载标记*/
if (!(flags & MS_NOATIME))
    mnt_flags |= MNT_RELATIME;

if (flags & MS_NOSUID)
    mnt_flags |= MNT_NOSUID;
if (flags & MS_NODEV)
    mnt_flags |= MNT_NODEV;
if (flags & MS_NOEXEC)
    mnt_flags |= MNT_NOEXEC;
if (flags & MS_NOATIME)
    mnt_flags |= MNT_NOATIME;
if (flags & MS_NODIRATIME)
    mnt_flags |= MNT_NODIRATIME;
if (flags & MS_STRICTATIME)
    mnt_flags &= ~(MNT_RELATIME | MNT_NOATIME);
if (flags & MS_RDONLY)
    mnt_flags |= MNT_READONLY;

/* The default atime for remount is preservation */
if ((flags & MS_REMOUNT) &&((flags & (MS_NOATIME | MS_NODIRATIME | MS_RELATIME |
                                                    MS_STRICTATIME)) == 0)) {
    mnt_flags &= ~MNT_ATIME_MASK;
    mnt_flags |= path.mnt->mnt_flags & MNT_ATIME_MASK;
 }

 flags &= ~(MS_NOSUID | MS_NOEXEC | MS_NODEV | MS_ACTIVE | MS_BORN |
    MS_NOATIME | MS_NODIRATIME | MS_RELATIME| MS_KERNMOUNT |
    MS_STRICTATIME);              /*清零系统调用标记参数以上标记位，非内部标记*/

 if (flags & MS_REMOUNT)          /*修改已经挂载文件系统的挂载标记*/
      retval = do_remount(&path, flags & ~MS_REMOUNT, mnt_flags, data_page);
 else if (flags & MS_BIND)        /*绑定挂载*/
      retval = do_loopback(&path, dev_name, flags & MS_REC);
 else if (flags & (MS_SHARED | MS_PRIVATE | MS_SLAVE | MS_UNBINDABLE))
      retval = do_change_type(&path, flags);       /*处理共享、私有、从属和不可绑定挂载*/
 else if (flags & MS_MOVE)
    retval = do_move_mount(&path, dev_name);
                        /*移动一个已经挂载的文件系统，注意 dev_name 表示旧的挂载点目录*/
 else
```

```
        retval = do_new_mount(&path, type_page, flags, mnt_flags, dev_name, data_page);
                                                        /*执行新的挂载操作，/fs/namespace.c*/
 dput_out:
        path_put(&path);
        return retval;
}
```

do_mount()函数首先调用 user_path(dir_name, &path)函数查找挂载点的目录项信息（user_path()函数在后面再介绍），path.dentry 指向挂载点目录项 dentry 实例，path.mnt 指向挂载点所在文件系统挂载 mount 实例中的 vfsmount 成员，然后将系统调用标记参数转换出 mount 实例使用的标记，最后根据标记参数调用不同的挂载执行函数。

MS_REMOUNT：表示修改一个挂载的标记，即重新挂载（不改变挂载点），执行函数为 do_remount()。

MS_BIND：执行绑定挂载，执行函数为 do_loopback()。

MS_SHARED | MS_PRIVATE | MS_SLAVE | MS_UNBINDABLE：修改已有挂载的类型，类型可以是共享、私有、从属和不可绑定挂载中的一种，执行函数为 do_change_type()。

MS_MOVE：移动一个已经挂载的文件系统，执行函数为 do_move_mount()。

若没有指定以上的标记位，将执行新的挂载，执行函数为 do_new_mount()。下面将介绍执行新挂载的函数，MS_REMOUNT 和 MS_MOVE 挂载请读者自行阅读源代码，下一小节将简要介绍共享、私有、从属挂载的处理和绑定挂载等。

do_new_mount()函数用于执行新的挂载操作，函数定义在/fs/namespace.c 文件内，代码如下：

```
static int do_new_mount(struct path *path, const char *fstype, int flags,      \
                                                        int mnt_flags, const char *name, void *data)
/*
*path：挂载点信息，fstype：文件系统类型名称字符串指针，
*flags：系统调用传递的挂载标记参数，mnt_flags：mount 实例中的挂载标记，
*name：文件系统所在存储设备文件名称，data：私有数据指针，由文件系统类型挂载函数使用。
*/
{
    struct file_system_type *type;          /*文件系统类型*/
    struct user_namespace *user_ns = current->nsproxy->mnt_ns->user_ns;      /*用户命名空间*/
    struct vfsmount *mnt;
    int err;

    if (!fstype)
        return -EINVAL;

    type = get_fs_type(fstype);
    if (!type)
        return -ENODEV;      /*没有指定文件系统类型或没有注册的文件系统类型，返回*/

    if (user_ns != &init_user_ns) {        /*如果当前用户处于自定义的用户命名空间中*/
        if (!(type->fs_flags & FS_USERNS_MOUNT)) {
```

```
                                        /*文件系统类型是否支持在自定义用户命名空间挂载*/
                put_filesystem(type);
                return -EPERM;
            }
            if (!(type->fs_flags & FS_USERNS_DEV_MOUNT)) {
                flags |= MS_NODEV;
                mnt_flags |= MNT_NODEV | MNT_LOCK_NODEV;
            }
            if (type->fs_flags & FS_USERNS_VISIBLE) {
                if (!fs_fully_visible(type, &mnt_flags))
                    return -EPERM;
            }
        }

        mnt = vfs_kern_mount(type, flags, name, data);          /*内核挂载函数，/fs/namespace.c*/
        if (!IS_ERR(mnt) && (type->fs_flags & FS_HAS_SUBTYPE) &&!mnt->mnt_sb->s_subtype)
            mnt = fs_set_subtype(mnt, fstype);

        put_filesystem(type);
        ...     /*错误处理*/

        err = do_add_mount(real_mount(mnt), path, mnt_flags);     /*关联挂载点，/fs/namespace.c*/
        ...     /*错误处理*/
        return err;         /*成功返回 0*/
    }
```

do_new_mount()函数由文件系统类型名称查找 file_system_type 实例，然后调用 vfs_kern_mount()函数执行内核挂载操作，主要完成创建超级块 super_block、根目录项 dentry 和 inode 结构体实例（由文件系统类型挂载函数完成），创建 mount 结构体实例并建立各结构体实例之间的关联，最后调用关联挂载点函数 do_add_mount()建立 mount 和挂载点 mountpoint 实例、挂载点目录 dentry 实例之间的关联，并将 mount 实例插入全局散列表，挂载操作完成。

下面将详细介绍内核挂载函数和关联挂载点函数的实现。

■内核挂载函数

内核挂载函数 vfs_kern_mount()定义在/fs/namespace.c 文件内，代码如下：

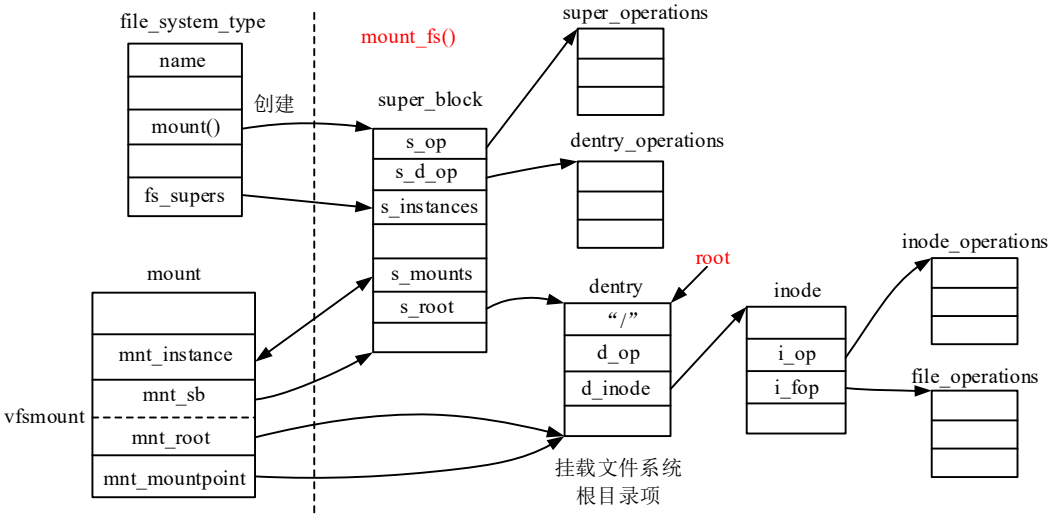
```
struct vfsmount *vfs_kern_mount(struct file_system_type *type, int flags, const char *name, void *data)
/*
*type：文件系统类型指针，flags：mount()系统调用传递的标记参数，
*name：文件系统所在存储设备文件名，data：私有数据指针。
*函数返回 mount 实例 mnt 成员指针（内嵌 vfsmount 实例）。
*/
{
    struct mount *mnt;
    struct dentry *root;
```

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

    /*从 slab 缓存分配 mount 实例，分配 ID 号，并初始化各成员*/
    mnt = alloc_vfsmnt(name);              /*/fs/namespace.c*/
    ...    /*错误处理*/

    if (flags & MS_KERNMOUNT)          /*内核发起的挂载操作*/
        mnt->mnt.mnt_flags = MNT_INTERNAL;          /*内部挂载*/

    root = mount_fs(type, flags, name, data);   /*调用文件系统类型定义的挂载函数，/fs/super.c*/
    ...    /*错误处理*/

    /*建立 mount 实例与 super_block、dentry 实例之间的关联*/
    mnt->mnt.mnt_root = root;            /*指向挂载文件系统根目录项 dentry 实例*/
    mnt->mnt.mnt_sb = root->d_sb;        /*指向超级块实例*/
    mnt->mnt_mountpoint = mnt->mnt.mnt_root;
                        /*设为挂载文件系统根目录项 dentry 实例，后面修改为挂载点 dentry 实例*/
    mnt->mnt_parent = mnt;          /*父 mount 实例指向自身*/
    lock_mount_hash();
    list_add_tail(&mnt->mnt_instance, &root->d_sb->s_mounts);   /*插入超级块中链表的末尾*/
    unlock_mount_hash();
    return &mnt->mnt;     /*返回 mount 实例 mnt 成员指针，vfsmount 结构体成员*/
}
```

内核挂载函数首先调用 alloc_vfsmnt()函数，从 slab 缓存中分配 mount 结构体实例，分配 ID 号，并初始化实例各成员；然后调用 mount_fs()函数，函数内调用文件系统类型实例中定义的挂载函数，创建文件系统超级块 super_block、根目录项 dentry 和 inode 结构体实例，返回挂载文件系统根目录项 dentry 实例指针；最后建立 mount 实例与超级块 super_block、挂载文件系统根目录项 dentry 实例之间的关联，并将 mount 实例添加到超级块 sb->s_mounts 链表末尾。

注意此时 mount 实例 mnt.mnt_root 和 mnt_mountpoint 成员都指向挂载文件系统根目录项 dentry 实例，mnt_mountpoint 成员在关联挂载点时将重新赋值，指向内核根文件系统中挂载点目录项 dentry 实例。

vfs_kern_mount()函数创建的数据结构实例组织关系如下图所示：



vfs_kern_mount()函数内调用 mount_fs()函数，此函数又调用文件系统类型定义的 mount()函数，创建文件系统超级块 super_block、根目录项 dentry 和 inode 结构体实例，返回挂载文件系统根目录项 dentry 实例指针。

mount_fs()函数定义如下（/fs/super.c）：

```
struct dentry *mount_fs(struct file_system_type *type, int flags, const char *name, void *data)
/*
*type：文件系统类型指针，flags：系统调用传递的挂载标记参数，
*name：文件系统存所在储设备文件名，data：私有数据指针。
*/
{
    struct dentry *root;    /*挂载文件系统根目录项的 dentry 实例指针，返回值*/
    struct super_block *sb;
    char *secdata = NULL;
    int error = -ENOMEM;

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

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

    root = type->mount(type, flags, name, data);
                        /*调用文件系统类型挂载函数，创建各数据结构体实例*/
    ...                 /*错误处理*/
    sb = root->d_sb;    /*文件系统超级块指针*/
```

```
    BUG_ON(!sb);
    WARN_ON(!sb->s_bdi);
    sb->s_flags |= MS_BORN;

    error = security_sb_kern_mount(sb, flags, secdata);
    ...
    up_write(&sb->s_umount);
    free_secdata(secdata);
    return root;        /*返回挂载文件系统根目录项 dentry 实例指针*/
    ...
}
```

mount_fs()函数主要就是调用文件系统类型实例中定义的 mount()函数，创建各数据结构实例，最后返回挂载文件系统根目录项 dentry 实例指针。

文件系统类型中的挂载函数通常调用（使用）内核提供的通用函数，例如，基于外部存储介质的文件系统类型，挂载函数通常调用 mount_bdev()函数实现，不需要外部存储介质的文件系统类型通常调用 mount_nodev()函数实现。

例如，ext2 文件系统类型实例定义如下（/fs/ext2/super.c）：

```
static struct file_system_type ext2_fs_type = {
    .owner          = THIS_MODULE,
    .name           = "ext2",
    .mount          = ext2_mount,       /*挂载函数*/
    .kill_sb    = kill_block_super,
    .fs_flags   = FS_REQUIRES_DEV,
};
```

文件系统类型挂载函数 ext2_mount()调用了通用的 mount_bdev()函数，定义如下（/fs/ext2/super.c）：

```
static struct dentry *ext2_mount(struct file_system_type *fs_type,int flags,   \
                                                    const char *dev_name, void *data)
{
    return mount_bdev(fs_type, flags, dev_name, data, ext2_fill_super);    /*通用函数*/
}
```

ext2_mount()函数内直接调用通用的 mount_bdev()函数，需要注意的是最后一个参数 ext2_fill_super 是一个函数指针，mount_bdev()函数内会调用此函数完成超级块实例的填充和初始化，还包括创建 dentry 和 inode 实例。

mount_bdev()函数定义在/fs/super.c 文件内，代码如下：

```
struct dentry *mount_bdev(struct file_system_type *fs_type,int flags, const char *dev_name, void *data, \
                                                    int (*fill_super)(struct super_block *, void *, int))
/*fill_super：填充超级块实例的函数指针，data 和 flags 参数传递给此函数*/
{
    struct block_device *bdev;
    struct super_block *s;
    fmode_t mode = FMODE_READ | FMODE_EXCL;
    int error = 0;
```

```
    if (!(flags & MS_RDONLY))
        mode |= FMODE_WRITE;

    bdev = blkdev_get_by_path(dev_name, mode, fs_type); /*获取块设备结构体实例，详见第 10 章*/
    ...     /*错误处理*/

    mutex_lock(&bdev->bd_fsfreeze_mutex);
    if (bdev->bd_fsfreeze_count > 0) {
        ...  /*错误返回*/
    }

    s = sget(fs_type, test_bdev_super, set_bdev_super, flags | MS_NOSEC,bdev);
                            /*查找或创建超级块实例，测试条件为 sb.s_bdev==bdev，见上文*/
    mutex_unlock(&bdev->bd_fsfreeze_mutex);
    ...    /*错误处理*/

    if (s->s_root) {        /*如果文件系统根目录项存在（已挂载），不需要创建根目录项 dentry 实例*/
        if ((flags ^ s->s_flags) & MS_RDONLY) {
            deactivate_locked_super(s);
            error = -EBUSY;
            goto error_bdev;
        }
        up_write(&s->s_umount);
        blkdev_put(bdev, mode);
        down_write(&s->s_umount);
    } else {            /*如果文件系统根目录项不存在，则需要填充超级块实例，并创建根目录项*/
        char b[BDEVNAME_SIZE];

        s->s_mode = mode;
        strlcpy(s->s_id, bdevname(bdev, b), sizeof(s->s_id));
        sb_set_blocksize(s, block_size(bdev)); /*设置超级块 s_blocksize 成员初始值（PAGE_SIZE）*/
        error = fill_super(s, data, flags & MS_SILENT ? 1 : 0);
                                    /*具体文件系统类型定义的函数，注意参数的传递*/
        ...    /*错误处理*/

        s->s_flags |= MS_ACTIVE;
        bdev->bd_super = s;        /*block_device.bd_super 指向超级块实例*/
    }

    return dget(s->s_root);    /*返回根目录项 dentry 实例指针*/
    ...
}
```

mount_bdev()函数首先根据块设备文件名查找块设备数据库，找到块设备对应的 block_device 实例（内核中唯一地表示块设备）赋予 bdev 变量，然后查找块设备超级块是否已经创建，测试函数 test_bdev_super()检测查找超级块实例 sb.s_bdev 成员是否等于 bdev，如果是则表示超级块实例是挂载块设备的超级块。如果超级块已经存在且具有对应的根目录项 dentry 实例，则无需再填充超级块，mount_bdev()函数直接返回 sb.s_root 即可。

如果挂载文件系统超级块实例不存在，则需要创建并填充超级块实例。如果超级块实例已经存在，但对应根目录项 dentry 实例不存在，也需要重新填充超级块实例。在 sget()函数创建超级块实例时，将调用函数 set_bdev_super()设置实例，随后调用 sb_set_blocksize()函数设置实例中数据块大小成员初始值为内存页大小（PAGE_SIZE）。

文件系统类型定义的 **fill_super()**函数用于进一步填充超级块实例，创建根目录项 dentry 实例，并且一般会再次调用 sb_set_blocksize()函数设置文件系统中数据块大小值到超级块 s_blocksize_bits 和 s_blocksize 成员，这两个成员值在读写文件内容时需要，详见第 11 章。

sget()函数参数中指定的设置超级块实例的 set_bdev_super()函数定义如下（/fs/super.c）：

```
static int set_bdev_super(struct super_block *s, void *data)
{
    s->s_bdev = data;                    /*指向 block_device 实例*/
    s->s_dev = s->s_bdev->bd_dev;        /*块设备号*/

    s->s_bdi = &bdev_get_queue(s->s_bdev)->backing_dev_info;
                                /*s_bdi 成员指向请求队列结构中 backing_dev_info 结构体成员*/
    return 0;
}
```

ext2 文件系统类型定义的超级块填充函数为 ext2_fill_super()，函数实现在/fs/ext2/super.c 文件内，有兴趣的读者可阅读其源代码。

对于不需要外部存储设备的文件系统类型，如：ramfs、proc 文件系统类型等，文件系统类型的挂载函数通常调用 mount_nodev()函数，函数定义如下：

```
struct dentry *mount_nodev(struct file_system_type *fs_type,   \
                           int flags, void *data,int (*fill_super)(struct super_block *, void *, int))
/*不需要块设备文件名称 dev_name 参数*/
{
    int error;
    struct super_block *s = sget(fs_type, NULL, set_anon_super, flags, NULL);
        /*test()参数为 NULL，总是创建新超级块实例，
        *set_anon_super()用于设置超级块虚拟块设备号，主设备号为 0。
        */

    ...  /*错误处理*/
    error = fill_super(s, data, flags & MS_SILENT ? 1 : 0);      /*填充超级块函数*/
    ...  /*错误处理*/
    s->s_flags |= MS_ACTIVE;
```

```
    return dget(s->s_root);
}
```

mount_nodev()函数内总是创建并填充新的超级块实例（每次挂载创建一个文件系统类型实例），set_anon_super()函数用于将虚拟块设备号赋予超级块 sb->s_dev 成员，其中主设备号为 0。mount_nodev()函数调用文件系统类型定义的 fill_super()函数填充超级块实例，并创建根目录项 dentry 和 inode 实例。

综上所述，内核挂载函数的主要工作包括：

（1）调用文件系统类型定义的挂载函数，创建（或查找）超级块 super_block、根目录项 dentry 和 inode 结构体实例，对数据结构实例进行初始化，并建立各实例之间的关联。

（2）创建表示挂载操作的 mount 结构体实例，并建立其与超级块 super_block 实例和挂载文件系统根目录项 dentry 实例之间的关联。

■关联挂载点

内核挂载函数只是建立了挂载文件系统相关的数据结构，还没有将挂载文件系统与内核根文件系统关联，此时挂载的文件系统对用户进程还不可见。

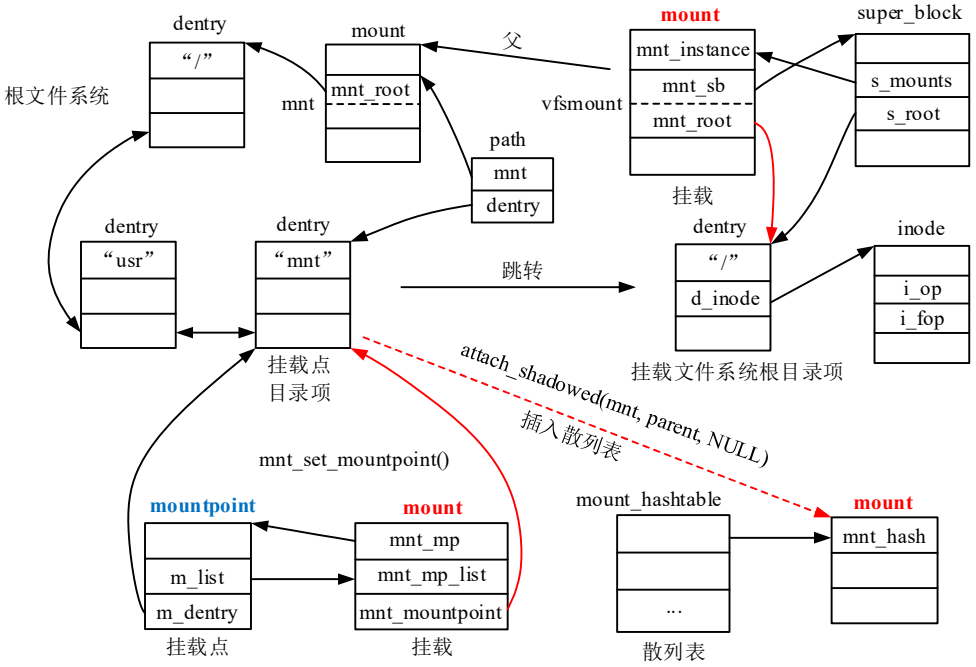
内核启动阶段利用内核挂载函数创建内核根文件系统，此时 vfs_kern_mount()函数中创建的挂载文件系统根目录项就是内核根文件系统的根目录项。

do_new_mount()函数执行完内核挂载函数 vfs_kern_mount()之后，接下来的工作就是通过 mount 实例建立挂载点 dentry 实例与挂载文件系统根目录项 dentry 实例之间的关联，并将 mount 实例添加到全局散列链表头部。以便将挂载的文件系统导入内核根文件系统，使之对用户进程可见。

关联挂载点的操作由 do_add_mount(real_mount(mnt), path, mnt_flags)函数完成，其中 real_mount(mnt)表示内核挂载函数中创建的 mount 实例指针，path 保存挂载点的信息，mnt_flags 为 mount 实例使用的挂载标记。

do_add_mount()函数内首先创建表示挂载点的 mountpoint 结构体实例，并关联挂载点目录项 dentry 实例，然后建立 mount 实例与 mountpoint 实例、挂载点 dentry 实例之间的关联，并将 mount 实例插入全局散列表。

do_add_mount()函数创建的数据结构实例及组织关系如下图所示：



上图中红色标记的 mount 实例是同一个实例，只是位于不同的地方。mount 实例 mnt_mountpoint 指向挂载点目录项。mount 实例根据 path.mnt（挂载点所在文件系统的挂载信息）和挂载点目录项 dentry 实例地址计算散列值，添加到全局散列链表头部。

do_add_mount()函数在/fs/namespace.c 文件内实现，代码如下：

```
static int do_add_mount(struct mount *newmnt, struct path *path, int mnt_flags)
/*newmnt：新创建 mount 实例指针，path：挂载点目录信息，mnt_flags：内部挂载标记*/
{
    struct mountpoint *mp;
    struct mount *parent;
    int err;

    mnt_flags &= ~MNT_INTERNAL_FLAGS;

    mp = lock_mount(path);   /*创建 mountpoint 实例，并建立与挂载点 dentry 关联，/fs/namespace.c*/
    ...    /*错误处理*/
    parent = real_mount(path->mnt);     /*vfsmount 指针转 mount 实例指针，父 mount 实例*/
    err = -EINVAL;
    if (unlikely(!check_mnt(parent))) {
        ...
    }

    /*避免同一文件系统重复挂载到同一挂载点*/
    err = -EBUSY;
    if (path->mnt->mnt_sb == newmnt->mnt.mnt_sb &&path->mnt->mnt_root == path->dentry)
        goto unlock;

    err = -EINVAL;
    if (d_is_symlink(newmnt->mnt.mnt_root))        /*挂载文件系统根目录项不能是符号链接*/
        goto unlock;

    newmnt->mnt.mnt_flags = mnt_flags;          /*挂载标记赋予 mount 实例*/
    err = graft_tree(newmnt, parent, mp);
                /*建立 mount 与 mountpoint、挂载点 dentry 实例关联，并插入散列表，/fs/namespace.c*/
 unlock:
    unlock_mount(mp);
    return err;
}
```

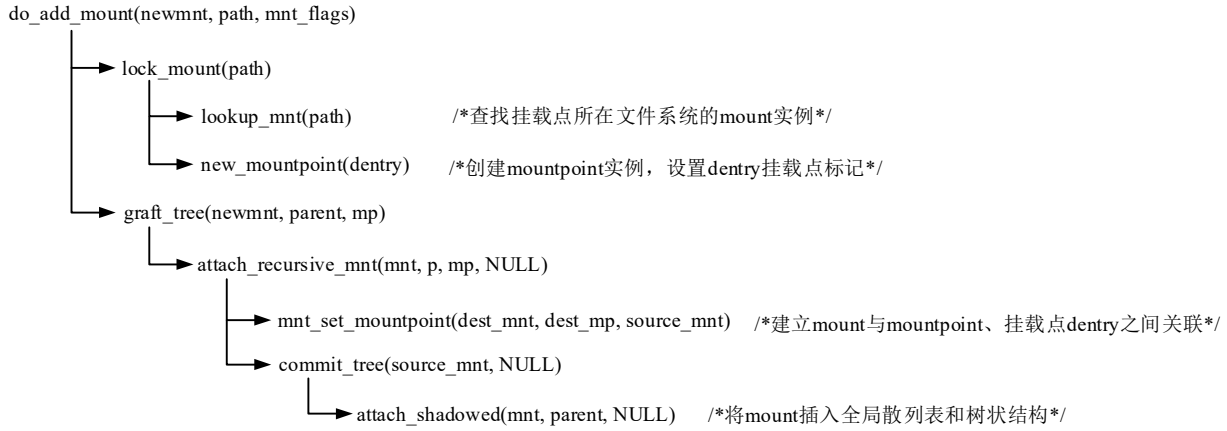
do_add_mount()函数的工作主要分为两步：

（1）调用 lock_mount(path)函数创建（或查找）挂载点 mountpoint 实例，建立其与挂载点 dentry 实例的关联，设置挂载点 dentry 实例 DCACHE_MOUNTED 标记位（d_set_mounted(dentry)），并将 mountpoint 实例添加到全局散列表。

（2）调用 graft_tree()函数，建立 mount 实例与 mountpoint、挂载点 dentry 实例之间的关联，并将 mount

实例插入到全局散列链表的头部，以及添加到 mount 树中（挂载点所在文件系统 mount 实例为其父实例）。

do_add_mount()函数调用关系如下图所示：



以上各调用函数简介如下：

●**lock_mount(path)**：以挂载点 dentry 实例及挂载点所在文件系统 vfsmount 实例指针为参数，调用 m_hash(mnt, dentry)函数计算散列值，通过 **lookup_mnt(path)**函数在全局散列链表中查找第一个关联挂载点 dentry 实例的 mount 实例（vfsmount 实例），以便找到挂载点 mountpoint 实例。

如果挂载点是第一次挂载，lookup_mnt(path)函数返回 NULL，随后 lock_mount()函数将调用函数 new_mountpoint()创建 mountpoint 实例（进行初始化）并插入到全局散列表，设置挂载点 dentry 实例的 **DCACHE_MOUNTED** 标记位。

打开文件的搜索目录项操作中，如果当前目录项为挂载点（检查 DCACHE_MOUNTED 标记位），将调用 **lookup_mnt(path)**函数查找符合条件的 mount 实例，实例中 mnt.mnt_root 成员指向的是挂载文件系统的根目录项，搜索路径进而跳转至此目录项，进入挂载文件系统。

●**mnt_set_mountpoint(dest_mnt, dest_mp, source_mnt)**：建立 mount 实例与挂载点 mountpoint、dentry 实例之间的关联，函数代码如下：

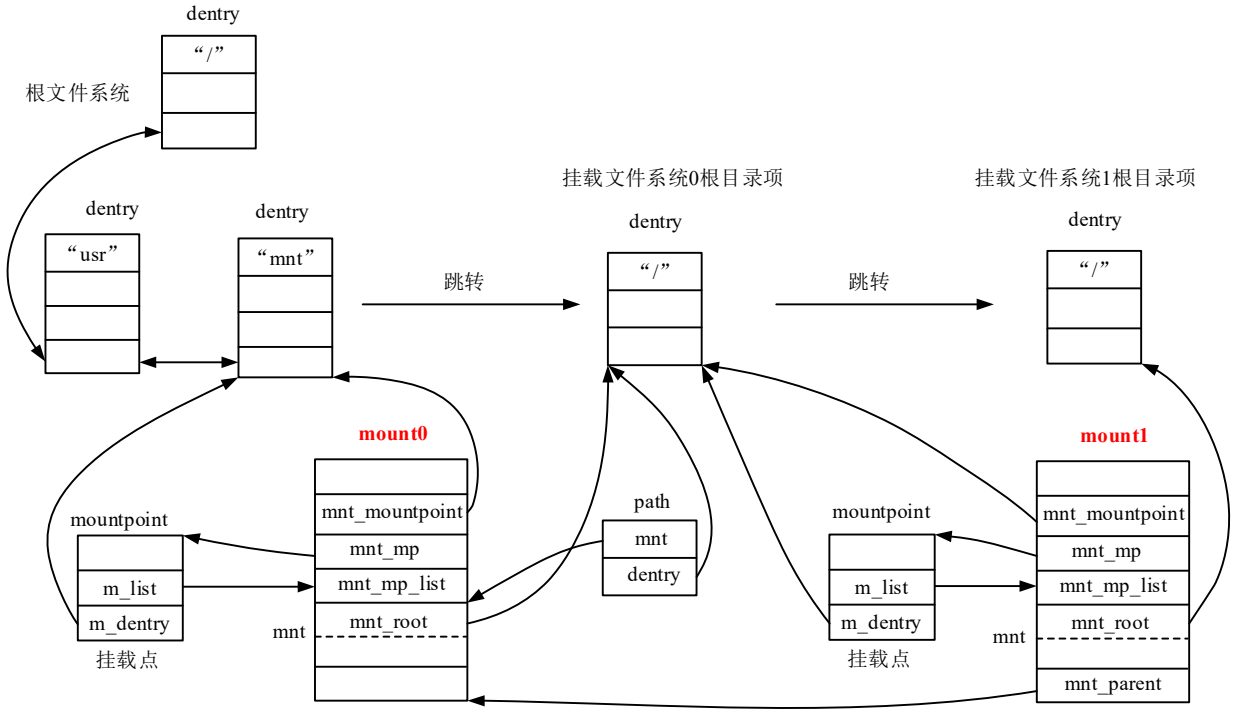
```
void mnt_set_mountpoint(struct mount *mnt,struct mountpoint *mp,struct mount *child_mnt)
/*mnt：父 mount 实例，mp：本次挂载点 mountpoint 实例，child_mnt：本次挂载 mount 实例*/
{
    mp->m_count++;              /*挂载点挂载计数加 1*/
    mnt_add_count(mnt, 1);
    child_mnt->mnt_mountpoint = dget(mp->m_dentry);    /*指向挂载点目录项 dentry 实例*/
    child_mnt->mnt_parent = mnt;              /*父 mount 实例*/
    child_mnt->mnt_mp = mp;                   /*指向挂载点 mountpoint 实例*/
    hlist_add_head(&child_mnt->mnt_mp_list, &mp->m_list);
                                              /*mount 实例添加到 mountpoint 实例管理的链表中*/
}
```

●**attach_shadowed(mnt, parent, NULL)**：将 mount 实例插入到全局散列表及树结构中，函数代码如下：

```
static void attach_shadowed(struct mount *mnt,struct mount *parent,struct mount *shadows)
/*mnt：本次挂载 mount 实例，parent：父 mount 实例，shadows：此处为 NULL*/
{
    if (shadows) {
        hlist_add_behind_rcu(&mnt->mnt_hash, &shadows->mnt_hash);
```

```
            list_add(&mnt->mnt_child, &shadows->mnt_child);
        } else {
            hlist_add_head_rcu(&mnt->mnt_hash,m_hash(&parent->mnt, mnt->mnt_mountpoint));
                                                    /*mount 实例插入到全局散列链表头部*/
            list_add_tail(&mnt->mnt_child, &parent->mnt_mounts);
                                                /*mount 实例插入父 mount 实例的子实例双链表末尾*/
        }
    }
```

下面我们来讨论一下在同一挂载点执行多次挂载操作，挂载多个文件系统的情形。假设根文件系统/mnt 目录项已经挂载了文件系统 0，现在再执行 mount()系统调用挂载文件系统 1。mount()系统调用中查找挂载点信息时，path 实例 mnt 成员指向/mnt 目录项第一次挂载的 mount.mnt 实例，dentry 成员指向文件系统 0 根目录项，如下图所示。mount()系统调用依然调用 do_new_mount()函数执行挂载操作，内核挂载函数不涉及到 path 实例，因此执行的结果和第一次挂载操作完全相同。第二次挂载主要的区别在于 do_new_mount()函数执行的结果不同，此时函数将以文件系统 0 的根目录项作为挂载点，而不是/mnt，do_new_mount()函数将为文件系统 0 根目录项创建 mountpoint 实例，并关联第二次挂载创建的 mount 实例，最后各数据结构体实例组织关系如下图所示。



内核文件搜索路径到达/mnt 目录项时，将首先跳转到文件系统 0 根目录项，当发现它依然是一个挂载点时，将继续跳转至文件系统 1 根目录项，并进入文件系统 1 下子目录。因此在一个挂载点执行多次挂载操作时，最近（最后）挂载的文件系统将覆盖之前挂载的文件系统，将其卸载后之前挂载的文件系统将自动可见。

7.4.4 挂载命名空间

命名空间可用于隔离资源，挂载命名空间用来隔离挂载点，每个进程属于某一挂载命名空间，各挂载命名空间是平等（并行）的关系，不存在父子关系。

内核采用单一的根文件系统，具体文件系统可挂载到根文件系统某个目录项下，挂载操作通过挂载

mount 实例建立挂载点与挂载文件系统根目录项的关联。每个挂载命名空间包含一个挂载 mount 实例树，此树决定了在此挂载命名空间内的根文件系统视图。改变挂载点目录项关联的 mount 实例，就可以改变挂载点挂载的文件系统，也就是改变根文件系统的视图。

由于每个挂载命名空间都有自己的挂载树，默认情况下挂载/卸载操作只在挂载命名空间内挂载树中执行，以此来实现挂载点的隔离。也就是说在一个挂载命名空间内的挂载/卸载操作不会影响其它挂载命名空间。

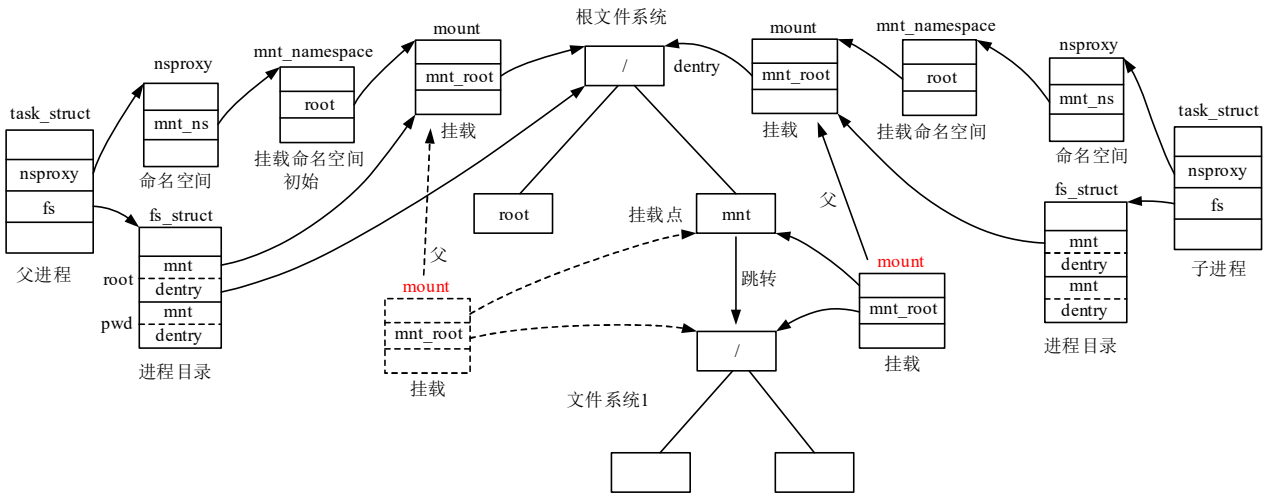
隔离太严重了有时也不好，因此内核在不同挂载命名空间之间也支持共享挂载、从属挂载等，即挂载/卸载操作可在多个挂载命名空间内可见。

1 概述

先看下图，简要说明一下挂载命名空间如何实现挂载点的隔离。

如下图中左侧所示，内核在创建根文件系统后（内核挂载），将创建初始的挂载命名空间，它关联到根文件系统根目录项对应的挂载 mount 实例。进程 task_struct 实例关联的 fs_struct 实例中，包含进程看到的根文件系统视图。root 和 pwd 成员（path 结构体）分别表示进程看到的根目录项和当前工作目录项，以及目录项所在文件系统的挂载 mount 实例。root 和 pwd 初始指向根文件系统根目录项，进程可通过系统调用修改这两个成员值。子进程会继承父进程的根文件系统视图信息。

如果不再创建新的挂载命名空间，则所有进程位于初始挂载命名空间，随后的挂载/卸载操作对所有进程可见。



现在父进程创建子进程时，创建新挂载命名空间（需要有 CAP_SYS_ADMIN 能力），子进程关联到新的挂载命名空间，新挂载命名空间复制一份父进程所在挂载命名空间的挂载树（mount 实例），如图中右侧所示。

若子进程在/mnt 目录下挂载文件系统 1，则文件系统 1 只对位于新挂载命名空间的进程可见，其它挂载命名空间内进程不可见。这是如何做到的呢？

子进程在/mnt 挂载文件系统 1，将创建新的 mount 实例，并添加到新挂载命名空间的 mount 实例树中，设置/mnt 目录项为挂载点。子进程打开文件搜索到 mnt 目录项时，发现其为挂载点，查找到其对应 mount 实例，并判断 mount 实例是否在本挂载命名空间的 mount 实例树中（检查父实例，见 lookup_mnt()函数），如果是则跳转至 mount 实例关联的挂载文件系统的根目录项（见 follow_managed()函数），进入文件系统 1。

父进程打开文件搜索到 mnt 目录项时，发现其为挂载点，也将查找到其对应的 mount 实例，但是此时 mount 实例不在父进程所有挂载命名空间中的 mount 实例树中，因此此挂载对父进程无效，不会跳转，不会进入文件系统 1，而是进入原 mnt 目录项（见 follow_managed()函数）。也就是说子进程挂载文件系统 1 对父进程是不可见的，父进程还是看到原来的 mnt 目录项下内容。

若要将在一个挂载命名空间下的挂载对其它挂载命名空间可见，则需要复制挂载 mount 实例，并添加到目标挂载命名空间的 mount 实例树中（类似关联挂载点操作），这样目标挂载命名空间也能看到这个挂载了，如上图左侧虚线所示。

2 创建挂载命名空间

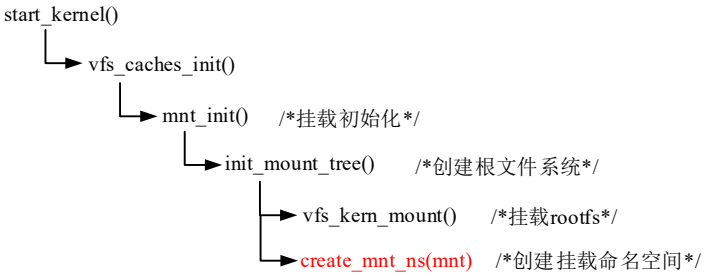
挂载命名空间由 mnt_namespace 结构体表示，定义如下（/fs/mount.h）：

```
struct mnt_namespace {
    atomic_t        count;          /*引用计数*/
    struct ns_common   ns;
    struct mount * root;            /*根挂载，挂载树根节点*/
    struct list_head    list;       /*链接挂载树中 mount 实例*/
    struct user_namespace    *user_ns;      /*用户命名空间*/
    u64             seq;            /* Sequence number to prevent loops */
    wait_queue_head_t poll;
    u64 event;
};
```

内核在挂载初始根文件系统后，将创建初始的挂载命名空间。创建子进程时，若设置了标记位 CLONE_NEWNS 将创建新挂载命名空间，子进程位于其中。

■初始挂载命名空间

内核在启动阶段后期将挂载 rootfs 文件系统（执行内核挂载）作为内核根文件系统，并创建初始的挂载命名空间，函数调用关系如下图所示：



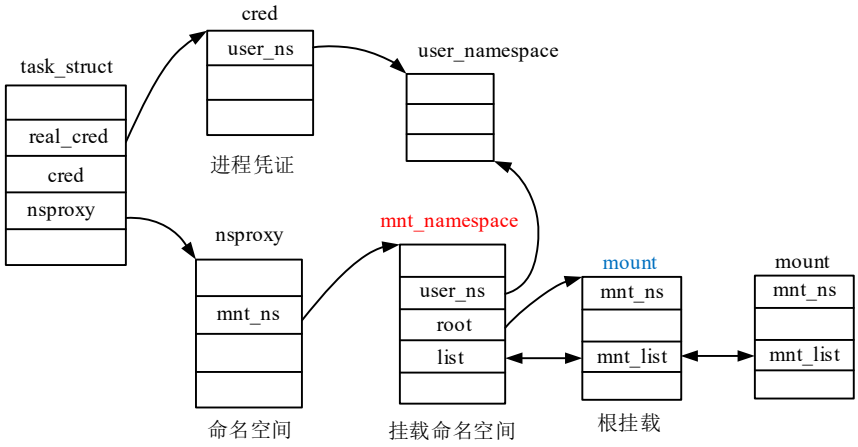
create_mnt_ns(mnt)函数用于创建初始挂载命名空间，函数定义如下（/fs/namespace.c）：

```
static struct mnt_namespace *create_mnt_ns(struct vfsmount *m)
{
    struct mnt_namespace *new_ns = alloc_mnt_ns(&init_user_ns);
                                        /*从 slab 缓存中分配 mnt_namespace 实例并初始化*/
    if (!IS_ERR(new_ns)) {
        struct mount *mnt = real_mount(m);      /*挂载 rootfs 时创建的 mount 实例*/
        mnt->mnt_ns = new_ns;       /*mount 实例关联挂载命名空间*/
        new_ns->root = mnt;         /*指向 mount 实例，根挂载*/
        list_add(&mnt->mnt_list, &new_ns->list);    /*mount 实例添加到挂载命名空间 list 双链表*/
    } else {
        mntput(m);
    }
```

return new_ns;　　/*返回挂载命名空间指针*/

}

create_mnt_ns(mnt)函数创建的挂载命名空间如下图所示，新挂载创建的 mount 实例将添加到挂载命名空间的 list 双链表和 mount 实例树中（图中未画出）：



■新挂载命名空间

在创建子进程时，若设置了 CLONE_NEWNS 标记位，将为子进程创建新的挂载命名空间（父进程须具有 CAP_SYS_ADMIN 能力），函数调用关系如下：

copy_process()->copy_namespaces()->create_new_namespaces()->**copy_mnt_ns()**。

copy_mnt_ns()函数定义如下（/fs/namespace.c）：

```
struct mnt_namespace *copy_mnt_ns(unsigned long flags, struct mnt_namespace *ns,
                                  struct user_namespace *user_ns, struct fs_struct *new_fs)
/*flags：创建进程标记，ns：父进程所在挂载命名空间，user_ns：子进程用户命名空间，
*new_fs：子进程文件系统信息指针。
*/
{
    struct mnt_namespace *new_ns;          /*指向新挂载命名空间*/
    struct vfsmount *rootmnt = NULL, *pwdmnt = NULL;
    struct mount *p, *q;
    struct mount *old;
    struct mount *new;        /*新 mount 树根节点*/
    int copy_flags;

    BUG_ON(!ns);

    if (likely(!(flags & CLONE_NEWNS))) {      /*没有设置 CLONE_NEWNS 标记位*/
        get_mnt_ns(ns);        /*增加引用计数，返回，不创建新挂载命名空间*/
        return ns;
    }

    old = ns->root;          /*父进程挂载命名空间中 mount 树根节点*/
    new_ns = alloc_mnt_ns(user_ns);          /*创建并初始化新挂载命名空间*/
```

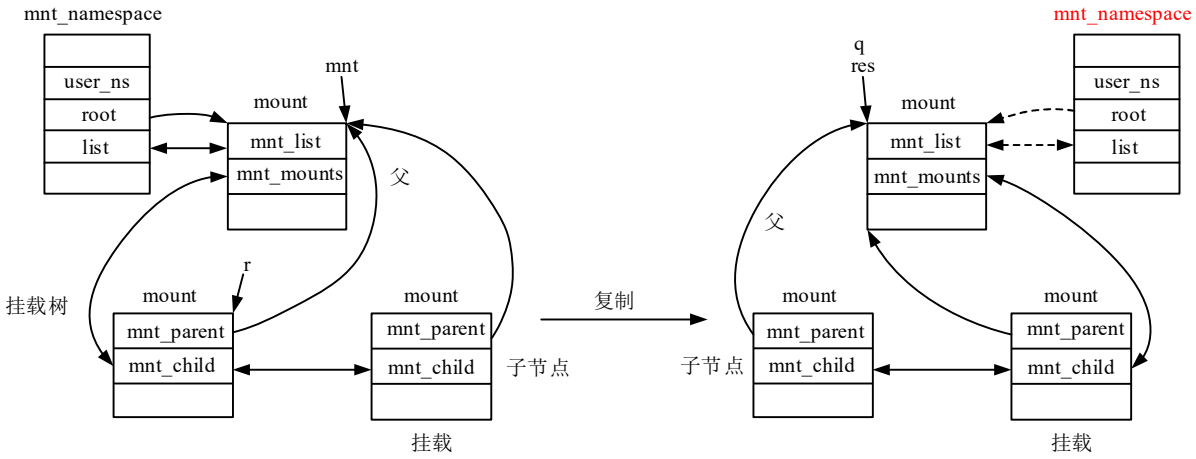
```
    ...    /*错误处理*/

    namespace_lock();
    copy_flags = CL_COPY_UNBINDABLE | CL_EXPIRE;
    if (user_ns != ns->user_ns)          /*子进程具有与父进程不同的用户命名空间*/
        copy_flags |= CL_SHARED_TO_SLAVE | CL_UNPRIVILEGED;
    new = copy_tree(old, old->mnt.mnt_root, copy_flags);          /*复制挂载树，返回根节点指针*/
    ...                                  /*错误处理*/
    new_ns->root = new;       /*新挂载命名空间关联 mount 树根节点*/
    list_add_tail(&new_ns->list, &new->mnt_list);   /*根 mount 实例添加到挂载命名空间中双链表*/

    p = old;
    q = new;
    while (p) {
        q->mnt_ns = new_ns;         /*新 mount 树中实例关联新挂载命名空间*/
        if (new_fs) {               /*修改子进程的文件系统（目录）信息*/
            if (&p->mnt == new_fs->root.mnt) {
                new_fs->root.mnt = mntget(&q->mnt);
                rootmnt = &p->mnt;
            }
            if (&p->mnt == new_fs->pwd.mnt) {
                new_fs->pwd.mnt = mntget(&q->mnt);
                pwdmnt = &p->mnt;
            }
        }
        p = next_mnt(p, old);
        q = next_mnt(q, new);
        if (!q)
            break;
        while (p->mnt.mnt_root != q->mnt.mnt_root)
            p = next_mnt(p, old);
    }
    namespace_unlock();

    if (rootmnt)
        mntput(rootmnt);
    if (pwdmnt)
        mntput(pwdmnt);

    return new_ns;          /*返回新挂载命名空间指针*/
}
```

copy_mnt_ns()函数主要分两步，一是创建新挂载命名空间实例，并复制父进程挂载命名空间中的挂载树，二是设置子进程文件系统（目录）信息，使其关联到新挂载树中 mount 实例。

进程的文件系统（目录）信息将在本章后面介绍，下面主要看一下复制挂载树 copy_tree()函数的实现。

●复制挂载树

某一挂载命名空间下的挂载 mount 实例按父子关系组成层次树状结构，并添加到挂载命名空间中的双链表，挂载命名空间关联挂载树根节点。复制挂载树就是复制一份以 mount 实例为根节点的挂载树副本，并返回新挂载树根节点指针，如下图所示。



在介绍 copy_tree()函数实现前，先看一下 mount 树的复制标记（/fs/pnode.h）：

```
#define   CL_EXPIRE                0x01
#define   CL_SLAVE                 0x02
#define   CL_COPY_UNBINDABLE       0x04
#define   CL_MAKE_SHARED           0x08
#define   CL_PRIVATE               0x10
#define   CL_SHARED_TO_SLAVE       0x20
#define   CL_UNPRIVILEGED          0x40
#define   CL_COPY_MNT_NS_FILE      0x80
#define   CL_COPY_ALL              (CL_COPY_UNBINDABLE | CL_COPY_MNT_NS_FILE)
```

copy_tree()函数中标记 flag 设为 **CL_COPY_UNBINDABLE | CL_EXPIRE**，如果子进程与父进程不在同一个用户命名空间，还会设置 CL_SHARED_TO_SLAVE | CL_UNPRIVILEGED 标记位。

copy_tree()函数用于复制 mount 树，返回新 mount 树的根节点指针，函数定义如下（/fs/namespace.c）：

```
struct mount *copy_tree(struct mount *mnt, struct dentry *dentry,int flag)
/*mnt：原 mount 树根节点指针，dentry：新 mount 根节点需关联的挂载文件系统根目录项，
*flag：复制标记，见上文。
*/
{
    struct mount *res, *p, *q, *r, *parent;

    if (!(flag & CL_COPY_UNBINDABLE) && IS_MNT_UNBINDABLE(mnt))
        return ERR_PTR(-EINVAL);                    /*设置了 CL_COPY_UNBINDABLE 标记*/

    if (!(flag & CL_COPY_MNT_NS_FILE) && is_mnt_ns_file(dentry))
        return ERR_PTR(-EINVAL);
```

```
res = q = clone_mnt(mnt, dentry, flag);          /*复制并初始化 mount 实例，/fs/namespace.c*/
...     /*错误处理*/

q->mnt_mountpoint = mnt->mnt_mountpoint;          /*新 mount 实例指向挂载点*/

p = mnt;          /*旧 mount 树根节点*/
list_for_each_entry(r, &mnt->mnt_mounts, mnt_child) {        /*遍历子挂载*/
      struct mount *s;
      if (!is_subdir(r->mnt_mountpoint, dentry))
            continue;

      for (s = r; s; s = next_mnt(s, r)) {        /*遍历后代挂载，并复制*/
            struct mount *t = NULL;
            if (!(flag & CL_COPY_UNBINDABLE) &&IS_MNT_UNBINDABLE(s)) {
                  s = skip_mnt_tree(s);
                  continue;
            }
            if (!(flag & CL_COPY_MNT_NS_FILE) &&is_mnt_ns_file(s->mnt.mnt_root)) {
                  s = skip_mnt_tree(s);
                  continue;
            }
            while (p != s->mnt_parent) {
                  p = p->mnt_parent;
                  q = q->mnt_parent;
            }
            p = s;
            parent = q;
            q = clone_mnt(p, p->mnt.mnt_root, flag);          /*复制 mount 实例*/
            if (IS_ERR(q))
                  goto out;
            lock_mount_hash();
            list_add_tail(&q->mnt_list, &res->mnt_list);        /*添加到根挂载 mount 实例中双链表*/
            mnt_set_mountpoint(parent, p->mnt_mp, q);       /*设置挂载点*/
            if (!list_empty(&parent->mnt_mounts)) {
                  t = list_last_entry(&parent->mnt_mounts,struct mount, mnt_child);
                  if (t->mnt_mp != p->mnt_mp)
                        t = NULL;
            }
            attach_shadowed(q, parent, t);          /*新 mount 实例添加到散列表和 mount 树*/
            unlock_mount_hash();
      }
}
return res;          /*返回新挂载命名空间的 mount 树的根节点*/
```

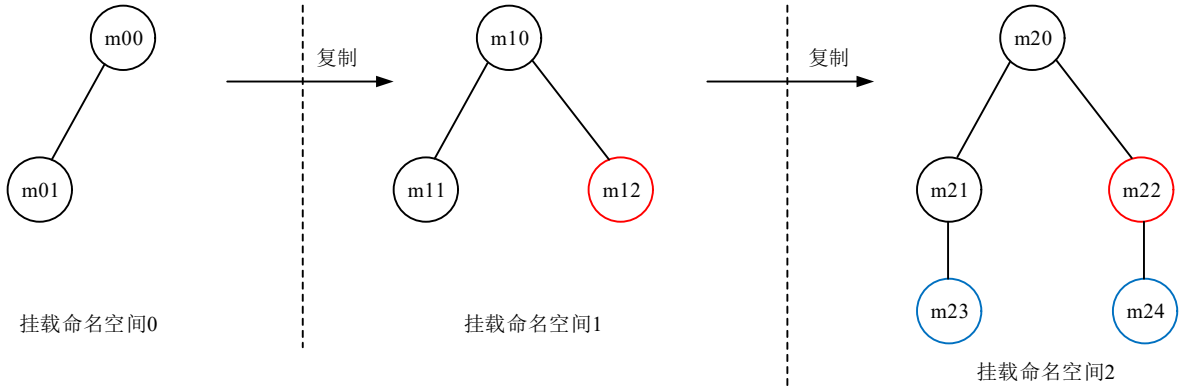
...

}

copy_tree()函数的主要功能是复制以参数 mnt 指向 mount 实例为根的 mount 树，返回新树根节点 mount 实例指针。在复制过程中会遍历原树中各层各实例，复制整棵树。具体复制 mount 实例的函数为 clone_mnt()，这里一个比较复杂的函数，需要考虑共享子树挂载的情况，详见下文。

创建新挂载命名空间的 copy_mnt_ns()函数，在复制完挂载树后，会将新挂载命名空间关联到新 mount 树根节点，修改子进程根/当前目录信息至新挂载树。

总之，创建挂载命名空间时，将为新创建的挂载命名空间复制一份父进程所在挂载命名空间的挂载树的副本，如下图所示：



默认情况下（不考虑下面介绍的共享子树），各挂载命名空间是独立的，互不干涉。执行挂载/卸载操作时，只改变所在挂载命名空间的挂载树，不影响其它挂载命名空间。创建新挂载命名空间时，挂载树会传递下去，如上图中红色圈所示。

复制挂载树的操作有两个地方会用到，一是创建新挂载命名空间时，要为新挂载命名空间复制一份完整的挂载树，二是在共享挂载下执行挂载时，挂载要传递到对等组的其它挂载之下，因此要复制挂载（子树），并添加到目标挂载树中，详见下文。

3 共享子树

前面介绍过，默认情况下，挂载命名空间是相互隔离的，在一个挂载命名空间中挂载/卸装操作，对其它挂载命名空间不可见，这称为私有挂载。在某些情况下，这种隔离程度太重了，有时用户需要执行一次挂载对其它挂载命名空间可见，为此内核引入了共享子树。

共享子树简单地说就是将某个挂载点（或以此挂载点为根的挂载子树）设为共享，在其下的挂载/卸载对本挂载命名空间，以及所有源于（复制于）本挂载命名空间的其它挂载命名空间可见。

共享子树（挂载子树）提供了 4 种类型的挂载：

●**共享挂载**：挂载/卸载对本挂载命名空间，以及所有源于本挂载命名空间的其它挂载命名空间可见。

●**从属挂载**：共享挂载的弱化版，其它共享挂载下的操作可传递到本挂载之下，本挂载之下的操作是私有的。

●**私有挂载**：默认的挂载类型，挂载/卸载操作只在本挂载命名空间内可见。

●**不可绑定挂载**：不可绑定挂载是私有挂载，且不允许被绑定挂载。

这里说的挂载子树类型，是指可以将一个已经存在挂载修改为这 4 种类型之一，而不是执行新的挂载，只是改变其挂载类型。4 种挂载类型其实是 2 种，即共享和私有

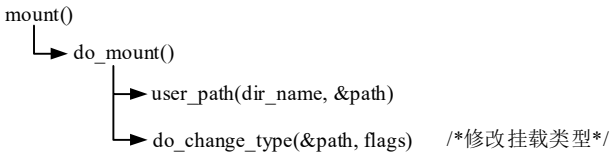
将某个挂载修改为共享挂载，则所有源于（复制于）本挂载的挂载组成一个对等组（都是共享挂载），也就是一个挂载在同源的所有挂载命名空间的副本组成一组。对等组内任一个挂载之下的挂载/卸载操作会传递到对等组内所有挂载之下。

对等组内的某个共享挂载可修改为从属挂载，从属挂载只指它接受对等组内其它挂载之下的挂载/卸载操作，但是从属挂载之下的挂载/卸载操作不会传递到对等组内其它挂载之下，传递是单向的。

执行新挂载操作时，默认执行的是私有挂载，随后可改为共享挂载（从属挂载）。共享挂载（从属挂载）也可以改回私有挂载。不可绑定挂载就是私有挂载，只不过它不允许执行绑定挂载。

修改挂载类型时，mount()系统调用中只有路径名（挂载点）和挂载标记参数有效，其它参数不使用。

mount()系统调用中修改挂载类型相关函数调用如下图所示：



do_change_type()函数用于修改挂载类型（当前可见的挂载），它执行新挂载的 do_new_mount()函数是并列的关系。系统调用参数 flag 设置 MS_SHARED、MS_SLAVE、MS_PRIVATE、MS_UNBINDABLE 标记位中的一位（且只能设置一位），将调用 do_change_type()函数。

do_change_type()函数定义如下（/fs/namespace.c）：

```
static int do_change_type(struct path *path, int flag)
/*flag：必须设置 MS_SHARED、MS_SLAVE、MS_PRIVATE、MS_UNBINDABLE 中的一位*/
/*path：挂载点信息，路径名必须是一个挂载点*/
{
    struct mount *m;
    struct mount *mnt = real_mount(path->mnt);          /*挂载点对应 mount 实例*/
    int recurse = flag & MS_REC;            /*是否作用于以 mount 实例为根的整个子树*/
    int type;
    int err = 0;

    if (path->dentry != path->mnt->mnt_root)          /*系统调用指定的路径必须是一个挂载点*/
        return -EINVAL;

    type = flags_to_propagation_type(flag);
      /*检查 flag 值，返回 MS_SHARED 、MS_PRIVATE 、MS_SLAVE 或 MS_UNBINDABLE*/
    if (!type)
        return -EINVAL;

    namespace_lock();
    if (type == MS_SHARED) {        /*共享挂载*/
        err = invent_group_ids(mnt, recurse);       /*分配对等组 ID，复制的挂载实例共用 ID*/
        if (err)
            goto out_unlock;
    }

    lock_mount_hash();
    for (m = mnt; m; m = (recurse ? next_mnt(m, mnt) : NULL))        /*遍历整个子树中 mount 实例*/
```

```
            change_mnt_propagation(m, type);          /*修改 mount 实例挂载类型*/
        unlock_mount_hash();
    out_unlock:
        namespace_unlock();
        return err;
    }
```

do_change_type()函数检查 mount()系统调用指定的路径必须是一个挂载点，挂载标记必须包含以下 4 个标记 MS_SHARED、 MS_PRIVATE、MS_SLAVE、MS_UNBINDABLE 中的一个，且只能有一个；如果是设置共享挂载可能还需要为挂载分配对等组 ID；最后对当前挂载点对应的 mount 实例及其子树下的所有 mount 实例（如果设置了 MS_REC 标记位）调用 change_mnt_propagation()函数设置挂载类型。

change_mnt_propagation()函数用于修改一个挂载 mount 实例的类型，定义如下（/fs/pnode.c）：

```
void change_mnt_propagation(struct mount *mnt, int type)
{
    if (type == MS_SHARED) {      /*共享挂载*/
        set_mnt_shared(mnt);      /*设置共享挂载标记*/
        return;
    }
    do_make_slave(mnt);          /*处理从属挂载（含私有挂载和不可绑定挂载）*/
    if (type != MS_SLAVE) {      /*处理私有挂载或不可绑定挂载*/
        list_del_init(&mnt->mnt_slave);
        mnt->mnt_master = NULL;
        if (type == MS_UNBINDABLE)
            mnt->mnt.mnt_flags |= MNT_UNBINDABLE;          /*不可绑定挂载*/
        else
            mnt->mnt.mnt_flags &= ~MNT_UNBINDABLE;         /*私有挂载*/
    }
}
```

下面将介绍各修改挂载类型的操作。

■共享挂载

如果是在某个挂载命名空间中将挂载设置为共享挂载，在 do_change_type()函数中将先调用函数 invent_group_ids()，为挂载分配对等组 ID，然后再调用 change_mnt_propagation()函数修改挂载类型。

invent_group_ids()函数定义在/fs/namespace.c 文件内，函数内判断挂载 mount 实例是否没有设置 MNT_SHARED 标记，且 mnt_group_id 成员值为 0，如果是则为 mount 实例分配对等组 ID 值，并写入成员 mnt_group_id。如果 mount()系统调用传递了 MS_REC 标记位，会将此操作作用于 mount 实例子树下所有挂载实例。

change_mnt_propagation()函数随后调用 **set_mnt_shared(mnt)**函数设置挂载 mount 实例为共享挂载类型，函数定义如下（/fs/pnode.h）：

```
static inline void set_mnt_shared(struct mount *mnt)
{
    mnt->mnt.mnt_flags &= ~MNT_SHARED_MASK;        /*清 MNT_UNBINDABLE 标记位*/
    mnt->mnt.mnt_flags |= MNT_SHARED;        /*设置 MNT_SHARED 标记*/
```

}

set_mnt_shared()函数只是简单地清除挂载的 MNT_UNBINDABLE 标记位，并设置 **MNT_SHARED** 标记，也就是说共享挂载不可以是不可绑定挂载。

●复制共享挂载

在创建新挂载命名空间时，将调用 copy_tree()函数复制父进程所在挂载命名空间的挂载树，为新挂载命名空间复制一个挂载树副本。copy_tree()函数中遍历挂载树，调用 clone_mnt()函数复制各 mount 实例。

copy_tree()函数中调用 clone_mnt()函数，复制标记设为 **CL_COPY_UNBINDABLE | CL_EXPIRE**，如果子进程与父进程不在同一个用户命名空间，还会设置 CL_SHARED_TO_SLAVE | CL_UNPRIVILEGED 标记位。

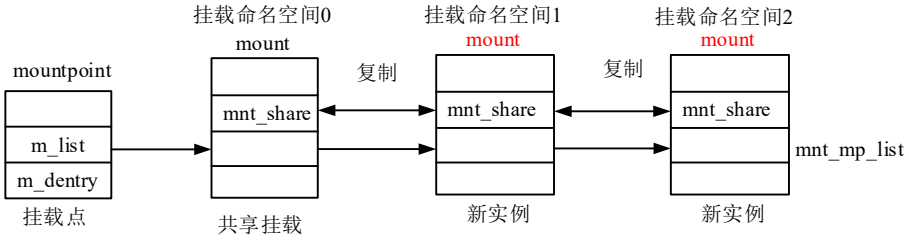
clone_mnt()函数定义如下（/fs/namespace.c）：

```
static struct mount *clone_mnt(struct mount *old, struct dentry *root,int flag)
/*old：旧实例，root：新实例需关联的挂载文件系统根目录项，flag：标记，见上文*/
{
    struct super_block *sb = old->mnt.mnt_sb;
    struct mount *mnt;
    int err;

    mnt = alloc_vfsmnt(old->mnt_devname);          /*分配新 mount 实例，并初始化*/
    ...                 /*错误处理*/

    if (flag & (CL_SLAVE | CL_PRIVATE | CL_SHARED_TO_SLAVE))
        mnt->mnt_group_id = 0;          /* not a peer of original */
    else
        mnt->mnt_group_id = old->mnt_group_id;       /*共享挂载与旧实例共用 ID*/

    if ((flag & CL_MAKE_SHARED) && !mnt->mnt_group_id) {
        err = mnt_alloc_group_id(mnt);           /*分配对等组 ID，如果需要*/
        ...                 /*错误处理*/
    }

    mnt->mnt.mnt_flags = old->mnt.mnt_flags & ~(MNT_WRITE_HOLD|MNT_MARKED);
                                                    /*新挂载标记来自旧挂载标记*/
    /*不允许非特权用户修改挂载标记*/
    if (flag & CL_UNPRIVILEGED) {
        mnt->mnt.mnt_flags |= MNT_LOCK_ATIME;

        if (mnt->mnt.mnt_flags & MNT_READONLY)
            mnt->mnt.mnt_flags |= MNT_LOCK_READONLY;

        if (mnt->mnt.mnt_flags & MNT_NODEV)
            mnt->mnt.mnt_flags |= MNT_LOCK_NODEV;

        if (mnt->mnt.mnt_flags & MNT_NOSUID)
```

```
                mnt->mnt.mnt_flags |= MNT_LOCK_NOSUID;

        if (mnt->mnt.mnt_flags & MNT_NOEXEC)
                mnt->mnt.mnt_flags |= MNT_LOCK_NOEXEC;
    }

    /* Don't allow unprivileged users to reveal what is under a mount */
    if ((flag & CL_UNPRIVILEGED) && (!(flag & CL_EXPIRE) || list_empty(&old->mnt_expire)))
        mnt->mnt.mnt_flags |= MNT_LOCKED;

    /*初始化新 mount 实例*/
    atomic_inc(&sb->s_active);
    mnt->mnt.mnt_sb = sb;
    mnt->mnt.mnt_root = dget(root);          /*新实例的挂载文件系统根目录项*/
    mnt->mnt_mountpoint = mnt->mnt.mnt_root;
    mnt->mnt_parent = mnt;
    lock_mount_hash();
    list_add_tail(&mnt->mnt_instance, &sb->s_mounts);       /*添加到超级块中双链表*/
    unlock_mount_hash();

    if ((flag & CL_SLAVE) ||((flag & CL_SHARED_TO_SLAVE) && IS_MNT_SHARED(old))) {
        list_add(&mnt->mnt_slave, &old->mnt_slave_list);
        mnt->mnt_master = old;
        CLEAR_MNT_SHARED(mnt);          /*从属挂载，清除其共享挂载标记*/
    } else if (!(flag & CL_PRIVATE)) {
        if ((flag & CL_MAKE_SHARED) || IS_MNT_SHARED(old))     /*共享挂载*/
            list_add(&mnt->mnt_share, &old->mnt_share);          /*添加到对等组双链表*/
        if (IS_MNT_SLAVE(old))
            list_add(&mnt->mnt_slave, &old->mnt_slave);
        mnt->mnt_master = old->mnt_master;
    }
    if (flag & CL_MAKE_SHARED)
        set_mnt_shared(mnt);

    if (flag & CL_EXPIRE) {
        if (!list_empty(&old->mnt_expire))
            list_add(&mnt->mnt_expire, &old->mnt_expire);
    }

    return mnt;          /*返回新 mount 实例指针*/
    ...
}
```

clone_mnt()函数需要处理共享挂载、从属挂载、私有挂载、不可绑定挂载等情况，这里先只看共享挂

载。如果原 mount 实例是共享挂载（设置了 MNT_SHARED 标记位），则复制的新实例也是共享挂载，因为新实例标记与旧实例标记基本相同，并且新实例添加到旧实例的 mnt_share 双链表中，如下图所示：

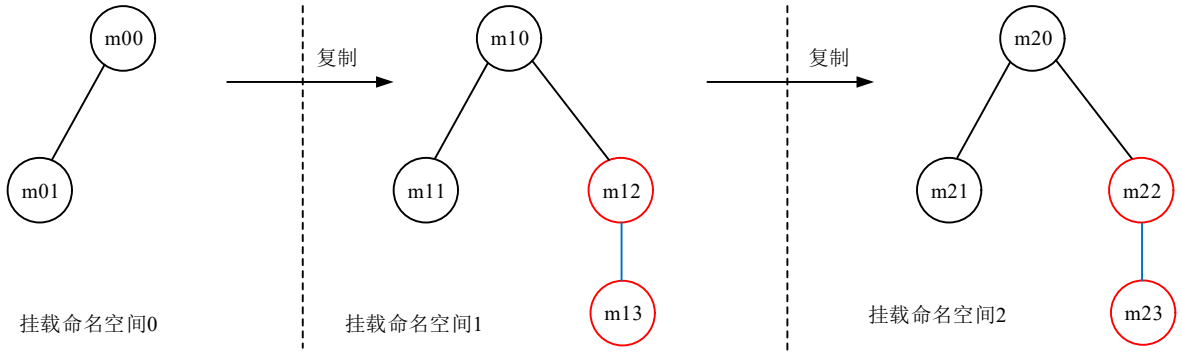


mnt_share 双链表管理的是所在源于同一共享 mount 实例，在不同挂载命名空间中的实例副本，这些实例组成对等组，内核为其分配一个对等组 ID。在对等组内任一 mount 实例下的挂载/卸载操作都会传递到组内其它 mount 实例之下。

所有复制的 mount 实例在 copy_tree()函数中随后会添加到挂载命名空间的挂载树和双链表中，以及挂载点中的单链表。

●执行挂载

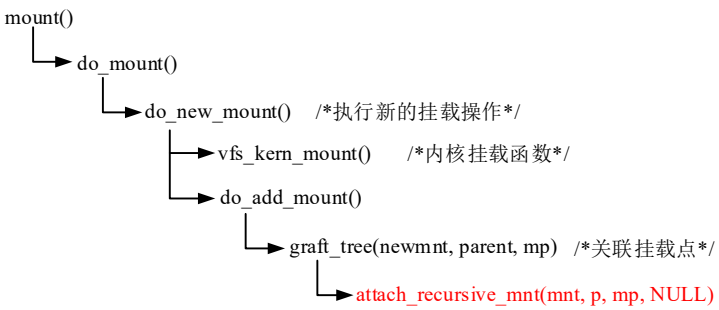
设置某个挂载点为共享挂载，只是修改了其标记位，并没有执行其它操作。如果在设为共享挂载的文件系统下执行挂载/卸载操作，此操作将传递到对等组其它挂载之下，如下图所示：



假设将挂载命名空间 1 下的 m12 设为共享挂载，而后挂载命名空间 2 复制于挂载命名空间 1。m22 是 m12 的副本（也是共享挂载），它们组成一个对等组。在 m12 下执行挂载 m13，将会在对等组内 m22 下创建 m13 挂载的副本 m23（共享挂载），m13 和 m23 也构成对等组。同理 m22 下的挂载/卸载也会传递到 m12 之下。

如果挂载命名空间 2 是在 m12 设为共享挂载之前复制的，则 m22 不是共享挂载，不会与 m12 组成对等组，挂载/卸载操作也不会相互传递。只有在复制挂载（子树）时才会向对等组添加成员。

在设为共享挂载的文件系统下（不是设置为共享挂载的挂载点，而是其下的目录项）执行新的挂载操作时，将挂载操作传递到对等组内其它挂载之下的操作，由关联挂载点的 attach_recursive_mnt()函数执行，函数调用关系如下：



attach_recursive_mnt()函数定义如下（/fs/namespace.c）：

static int attach_recursive_mnt(struct mount ***source_mnt**,struct mount ***dest_mnt**,

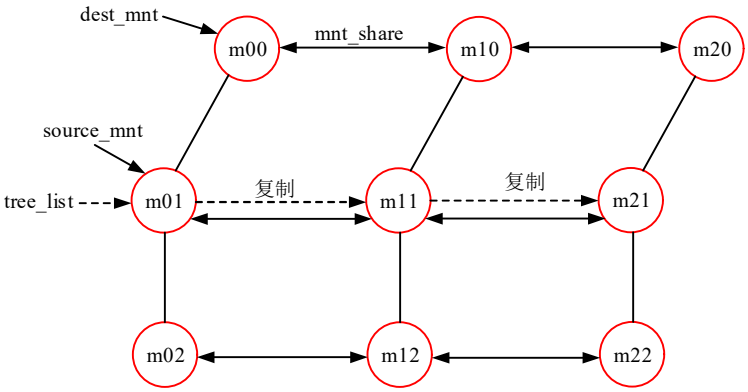
```
                                                    struct mountpoint *dest_mp,struct path *parent_path)
/*source_mnt：本次挂载创建的 mount 实例，dest_mnt：父 mount 实例，这里为共享挂载，
*dest_mp：本次挂载的挂载点，parent_path：此处为 NULL。
*/
{
    HLIST_HEAD(tree_list);        /*临时链表头*/
    struct mount *child, *p;
    struct hlist_node *n;
    int err;

    if (IS_MNT_SHARED(dest_mnt)) {        /*父挂载是共享挂载*/
        err = invent_group_ids(source_mnt, true);      /*为本次 mount 及其下子树 mount 实例分配 ID*/
        ...                /*错误处理*/
        err = propagate_mnt(dest_mnt, dest_mp, source_mnt, &tree_list);
          /*为 mount 及其下实例在对等组挂载之下复制副本并添加到 tree_list 双链表，/fs/pnode.c*/
        lock_mount_hash();
        ...                /*错误处理*/
        for (p = source_mnt; p; p = next_mnt(p, source_mnt))
            set_mnt_shared(p);          /*设置 source_mnt 及其下实例为共享挂载*/
    } else {
        lock_mount_hash();
    }
    if (parent_path) {            /*parent_path 为 NULL*/
        ...
    } else {
        mnt_set_mountpoint(dest_mnt, dest_mp, source_mnt);      /*关联挂载点*/
        commit_tree(source_mnt, NULL);      /*mount 实例添加到挂载树和挂载命名空间中双链表*/
    }

    /*将 tree_list 双链表中实例添加到挂载命名空间中挂载树，以及挂载命名空间中双链表*/
    hlist_for_each_entry_safe(child, n, &tree_list, mnt_hash) {
        struct mount *q;
        hlist_del_init(&child->mnt_hash);
        q = __lookup_mnt_last(&child->mnt_parent->mnt,child->mnt_mountpoint);
        commit_tree(child, q);
    }
    unlock_mount_hash();
    return 0;
    ...
}
```

如下图所示，假设 m00 在挂载命名空间 0，且为共享挂载，在其下执行挂载 m01。m01 执行正常的挂载，且设为共享挂载。也就是说在共享挂载之下的挂载也是共享挂载。

由于父挂载 m00 为共享挂载，将执行额外的工作。在 attach_recursive_mnt()函数中将调用 propagate_mnt()

函数遍历父挂载 m00 的 mnt_share 双链表，为双链表中各对等组挂载实例复制 m01 实例副本，各副本与原实例构成对等组，且由 tree_list 临时双链表管理。

在 attach_recursive_mnt()函数最后会将 tree_list 双链表中各实例添加到其所在的挂载树和挂载命名空间。propagate_mnt()函数定义在/fs/pnode.c 文件内，源代码请读者自行阅读。



由上可知，m01、m11 和 m21 构成对等组，在其中任意一个实例下的挂载/卸载，都传递到其它对等组内的挂载之下，且它们也构成对等组（如 m02、m12 和 m22）。

简单地说共享挂载下的挂载也是共享挂载，共享挂载下的挂载/卸载操作会传递到对等组内所有挂载之下。

■从属挂载

从属挂载是指将对等组内的某个共享挂载修改为从属挂载，去掉共享挂载标记（对等组 ID 清 0），从属挂载从对等组双链表中移出。从属挂载关联到一个主挂载，通常是从属挂载在原对等组内的下一个挂载。从属挂载添加到主挂载的从属挂载双链表中，从属挂载与主挂载并不是主从或父子关系，而是平等的关系，从属挂载只是借用主挂载获取原对等组内其它挂载下的挂载/卸载操作。因为从属挂载已经从对等组中移出了，需要通过主挂载保持与对等组的联系。

对等组内其它共享挂载之下的挂载/卸载操作还是会传递到从属挂载之下，但是从属挂载之下的挂载/卸载不会传递到对等组内其它挂载之下，也就是说传递是单向的。

在 mount()系统调用中如果设置了 **MS_SLAVE** 标记位，表示将一个共享挂载改为从属挂载，相应函数调用关系如下：

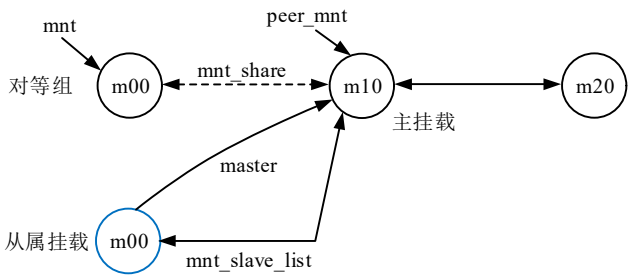
```
mount()
    └► do_mount()
        ├► user_path(dir_name, &path)
        └► do_change_type(&path, flags)      /*修改挂载类型*/
            └► change_mnt_propagation(m, type)        /*MS_SLAVE*/
                └► do_make_slave(mnt)      /*处理从属挂载*/
```

do_make_slave()函数用于处理从属挂载，函数定义如下（/fs/pnode.c）：

```
static int do_make_slave(struct mount *mnt)
{
    struct mount *peer_mnt = mnt, *master = mnt->mnt_master;        /*从属挂载的原主挂载*/
    struct mount *slave_mnt;

    while ((peer_mnt = next_peer(peer_mnt)) != mnt &&peer_mnt->mnt.mnt_root != mnt->mnt.mnt_root) ;

    if (peer_mnt == mnt) {        /*对等组内挂载都挂载的是同一文件系统*/
```

```
        peer_mnt = next_peer(mnt);      /*对等组内下一个挂载*/
        if (peer_mnt == mnt)            /*若只有一个挂载，peer_mnt 为 NULL*/
            peer_mnt = NULL;
    }
    if (mnt->mnt_group_id && IS_MNT_SHARED(mnt) &&list_empty(&mnt->mnt_share))
        mnt_release_group_id(mnt);      /*对等组内只有一个成员，释放对等组 ID*/

    list_del_init(&mnt->mnt_share);     /*从属挂载从 mnt_share 双链表中移出*/
    mnt->mnt_group_id = 0;              /*从属挂载的对等组 ID 设为 0*/

    if (peer_mnt)
        master = peer_mnt;

    if (master) {       /*存在主挂载*/
        list_for_each_entry(slave_mnt, &mnt->mnt_slave_list, mnt_slave)
            slave_mnt->mnt_master = master;         /*改变 mnt 下从属挂载的主挂载*/
        list_move(&mnt->mnt_slave, &master->mnt_slave_list);  /*添加到主挂载下的从属双链表*/
        list_splice(&mnt->mnt_slave_list, master->mnt_slave_list.prev);
                                        /*mnt 下从属挂载移动到主挂载下的从属挂载双链表*/
        INIT_LIST_HEAD(&mnt->mnt_slave_list);
    } else {            /*master 为 NULL*/
        struct list_head *p = &mnt->mnt_slave_list;
        while (!list_empty(p)) {        /*从属链表非空*/
            slave_mnt = list_first_entry(p,struct mount, mnt_slave);
            list_del_init(&slave_mnt->mnt_slave);       /*从从属链表移出*/
            slave_mnt->mnt_master = NULL;       /*主挂载设为 NULL*/
        }
    }
    mnt->mnt_master = master;           /*指向主挂载，可能为 NULL*/
    CLEAR_MNT_SHARED(mnt);      /*清共享挂载 MNT_SHARED 标记位*/
    return 0;
}
```

下面用图示的方法来说明上面的 do_make_slave()函数，如下图所示：



假设要将 mnt 指向的 mount 实例（共享挂载）修改为从属挂载，do_make_slave()函数将遍历 mnt 实例所在对等组，查找第一个与 mnt 挂载不同文件系统的 mount 实例 peer_mnt，如果所有 mount 实例挂载同一个文件系统（通常是这样），则 peer_mnt 指向 mnt 下一个实例。若对等组中只有 mnt 一个实例，则 peer_mnt

为 NULL。peer_mnt 就是为 mnt 查找到的主挂载，赋予 master（也可能是 mnt 原来的主挂载）。

随后，将 mnt 从对等组双链表中移出，其对等组 ID 设为 0。如果 master 不为 NULL，则将 mnt->mnt_slave 添加到 master->mnt_slave_list 双链表，如果 mnt 下从属挂载双链表不为空，则将其下成员也移入 master->mnt_slave_list 双链表（具有共同的主挂载）。

●复制从属挂载

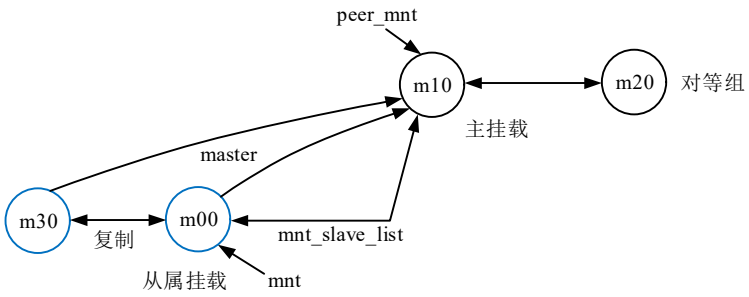
下面看一下在 copy_tree()函数中调用的 clone_mnt()函数如何复制从属挂载。

clone_mnt()函数标记设为 **CL_COPY_UNBINDABLE | CL_EXPIRE**，如果子进程与父进程不在同一个用户命名空间，还会设置 CL_SHARED_TO_SLAVE | CL_UNPRIVILEGED 标记位。

clone_mnt()函数代码简列如下（/fs/namespace.c）：

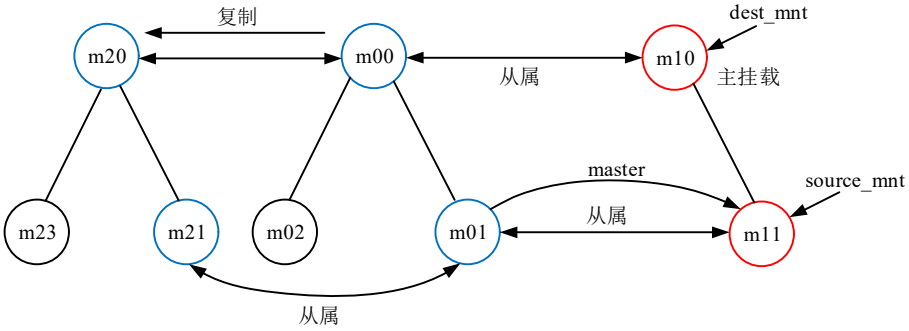
```
static struct mount *clone_mnt(struct mount *old, struct dentry *root,int flag)
/*old：旧实例，root：新实例需关联的挂载文件系统根目录项，flag：标记，见上文*/
{
    struct super_block *sb = old->mnt.mnt_sb;
    struct mount *mnt;
    int err;

    mnt = alloc_vfsmnt(old->mnt_devname);          /*分配新 mount 实例，并初始化*/
    ...                 /*错误处理*/
    /*初始化新 mount 实例*/
    ...

    if ((flag & CL_SLAVE) ||((flag & CL_SHARED_TO_SLAVE) && IS_MNT_SHARED(old))) {
        list_add(&mnt->mnt_slave, &old->mnt_slave_list);            /*转为从属挂载*/
        mnt->mnt_master = old;
        CLEAR_MNT_SHARED(mnt);
    } else if (!(flag & CL_PRIVATE)) {
        if ((flag & CL_MAKE_SHARED) || IS_MNT_SHARED(old))     /*共享挂载*/
            list_add(&mnt->mnt_share, &old->mnt_share);         /*对等组链表*/
        if (IS_MNT_SLAVE(old))          /*从属挂载，具有主挂载*/
            list_add(&mnt->mnt_slave, &old->mnt_slave);         /*添加到从属挂载双链表*/
        mnt->mnt_master = old->mnt_master;          /*指向共同的主挂载*/
    }
    ...
    return mnt;          /*返回新 mount 实例指针*/
    ...
}
```

如下图所示，假设 mnt 指向的 m00 为从属挂载（清除了共享挂载标记），为其复制一个副本 m30（挂载命名空间 3），副本 m30 将与 mnt 一起添加到 peer_mnt 主挂载的从属挂载双链表下，并且它们具有相同的主挂载。



●执行挂载

在从属挂载下执行挂载/卸载其实就是私有挂载，不传递到其它挂载命名空间，因此不需要特殊处理。在主挂载所在对等组内挂载之下执行挂载/卸载时，将通过主挂载，将此操作传递到主挂载的所有从属挂载之下。下面还是用图示来说明问题。



如上图所示，假设最开始 m00 和 m10 都是共享挂载，位于对等组内。m00 而后设为从属挂载，从对等组中移出，m10 为其主挂载，m00 添加到 m10 的从属挂载双链表。m20 复制于 m00，也为 m10 的从属挂载，添加到从属双链表。

现在在 m10 下挂载 m11，它将通过 m10 的从属挂载双链表传递到 m00 之下，创建挂载 m01，也传递到 m20 之下创建挂载 m21。m01 和 m21 是 m11 的从属挂载，而不是共享挂载，m11 是 m01 和 m21 的主挂载。

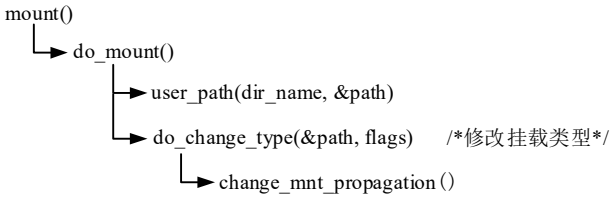
在 m11 之下的挂载/卸载也会传递到其下的从属挂载之下（m01 和 m21），但是从属挂载之下的挂载/卸载是私有的，对其它挂载命名空间不可见，详见 attach_recursive_mnt()->propagate_mnt()->propagate_one()函数，源代码请读者自行阅读。

■私有与不可绑定挂载

私有挂载很容易理解，它既不是共享挂载，也不是从属挂载，这也是内核默认的挂载类型。私有挂载下的挂载/卸载操作不会传递到其它挂载命名空间，其它挂载命名空间内的操作也不会传递到本私有挂载之下。不可绑定挂载就是私有挂载，只不过它不允许被绑定挂载。

在 mount()系统调用中设置 MS_PRIVATE 标记位可将挂载修改为私有挂载，设置 MS_UNBINDABLE 标记位可将挂载修改为不可绑定挂载。修改前挂载可以是共享、从属挂载等。

mount()系统调用中最终由 change_mnt_propagation()函数处理私有和不可绑定挂载，函数调用关系如下：



下面再次列出 change_mnt_propagation()函数代码（/fs/pnode.c）：

```
void change_mnt_propagation(struct mount *mnt, int type)
{
    if (type == MS_SHARED) {       /*共享挂载*/
        set_mnt_shared(mnt);       /*处理共享挂载*/
        return;
    }
    do_make_slave(mnt);            /*处理从属挂载，私有挂载和不可绑定挂载也会调用此函数*/
    if (type != MS_SLAVE) {        /*处理私有挂载或不可绑定挂载*/
        list_del_init(&mnt->mnt_slave);        /*从从属挂载双链表中移出，取消从属挂载*/
        mnt->mnt_master = NULL;
        if (type == MS_UNBINDABLE)
            mnt->mnt.mnt_flags |= MNT_UNBINDABLE;        /*设置不可绑定标记*/
        else
            mnt->mnt.mnt_flags &= ~MNT_UNBINDABLE;       /*清不可绑定标记，私有挂载*/
    }
}
```

这里需要注意，对于私有挂载和不可绑定挂载，也会调用 do_make_slave(mnt)函数进行处理。如果挂载原为共享挂载，do_make_slave(mnt)函数会将挂载从对等组中移出，对等组 ID 清 0，然后将挂载转为从属挂载。

change_mnt_propagation()函数随后判断如果不是将挂载修改为从属挂载，则将挂载从从属挂载双链表中移出，取消其的从属挂载类型。最后，如果是转为不可绑定挂载，则设置 MNT_UNBINDABLE 标记位，否则清除 MNT_UNBINDABLE 标记位（即为私有挂载）。

创建新挂载命名空间复制私有挂载（含不可绑定挂载）时，并不需要做更多的处理，直接复制就可以，原实例与副本之间相互独立，没有什么关联。

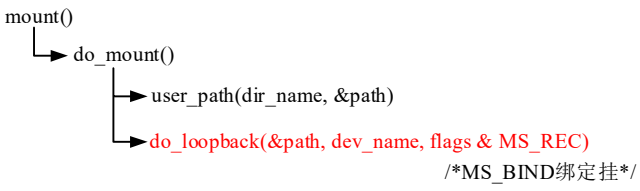
在私有挂载（含不可绑定挂载）之下执行挂载/卸载时，只执行在本挂载命名空间内的正常操作，不需要做其它工作。

4 绑定挂载

绑定挂载（bind mount）用来把目录树中的一棵子树挂载到其它地方。执行绑定挂载后从原目录和新目录都可以看到相同的内容。绑定挂载相当于将根文件系统中某个目录在另一个目录中建立硬链接，从两个地方都可以访问相同的内容。

绑定挂载甚至可以把一个文件绑定到另一个文件，访问这两个文件时看到的数据完全相同，相当于文件硬链接。

执行绑定挂载时，mount()系统调用中需设置 MS_BIND 标记位，处理函数调用关系如下：



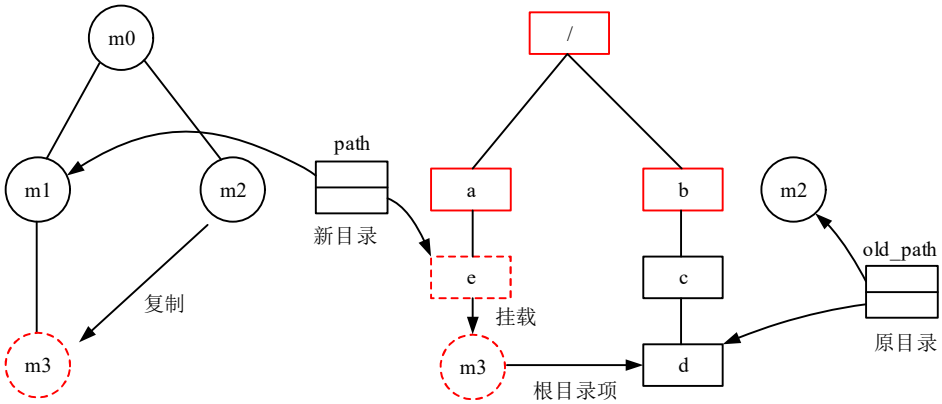
do_loopback()函数定义如下（/fs/namespace.c）：

```
static int do_loopback(struct path *path, const char *old_name,int recurse)
/*path：新目录信息，old_name：旧目录信息，recurse：是否设置 MS_REC 标记位*/
```

```
{
    struct path old_path;
    struct mount *mnt = NULL, *old, *parent;
    struct mountpoint *mp;
    int err;
    if (!old_name || !*old_name)
        return -EINVAL;
    err = kern_path(old_name, LOOKUP_FOLLOW|LOOKUP_AUTOMOUNT, &old_path);
                                                        /*查找旧目录信息*/
    ...        /*错误处理*/

    err = -EINVAL;
    if (mnt_ns_loop(old_path.dentry))
        goto out;

    mp = lock_mount(path);       /*新目录对应的挂载点，没有则创建*/
    ...
    old = real_mount(old_path.mnt);
    parent = real_mount(path->mnt);

    err = -EINVAL;
    if (IS_MNT_UNBINDABLE(old))       /*非绑定挂载，则不能执行绑定挂载*/
        goto out2;

    if (!check_mnt(parent))       /*parent 与当前进程需处于同一挂载命名空间*/
        goto out2;

    if (!check_mnt(old) && old_path.dentry->d_op != &ns_dentry_operations)
        goto out2;

    if (!recurse && has_locked_children(old, old_path.dentry))
        goto out2;

    if (recurse)        /*是否递归*/
        mnt = copy_tree(old, old_path.dentry, CL_COPY_MNT_NS_FILE);       /*复制挂载树*/
    else
        mnt = clone_mnt(old, old_path.dentry, 0);         /*复制挂载*/

    ...    /*错误处理*/

    mnt->mnt.mnt_flags &= ~MNT_LOCKED;

    err = graft_tree(mnt, parent, mp);          /*mnt 关联挂载点及挂载目录项*/
```

```
        ...     /*错误处理*/
out2:
        unlock_mount(mp);
out:
        path_put(&old_path);
        return err;
}
```

下面还是用图示的方式来说明绑定挂载的操作，如下图所示：



如上图所示，假设要将/b/c/d 目录绑定挂载到/a/e 目录下，根文件系统根目录项、a 和 b 目录项都是挂载点，分别对应挂载 m0、m1 和 m2。旧目录 old_path 实例关联到 m2 实例和 d 目录项，新目录 path 实例关联到 m1 和 e 目录项。

要在 e 目录项下建立挂载就要创建 mount 实例 m3，m3 复制于 m2（旧目录 old_path 实例关联的 mount 实例），m3 关联的挂载文件系统根目录项设为 d，也就是说将 d 当成一个挂载文件系统的根目录项，挂载到 e 目录项下。m3 要添加到挂载树（挂载命名空间）和 e 目录项挂载点的管理结构中。目录项搜索到达 e 目录项时，发现其为挂载点，将跳转到 d 目录项下继续搜索，进入目录项 d 之下，这就是绑定挂载的语义。

如果 m2 是共享挂载，m3 也将是共享挂载，并且绑定挂载会传递到其它 m2 对等组挂载之下。如果设置了 MS_REC 标记位，则 m2 之下的挂载子树也会一并复制到 m3 之下。

7.4.5 卸载文件系统

卸载文件系统操作是挂载文件系统操作的逆操作，即断开通过 mount 实例建立的根文件系统挂载点目录项与挂载文件系统根目录项之间的关联，使文件系统对内核不再可见，并释放相关数据结构实例。

卸载文件系统的系统调用为 umount()，实现函数定义在/fs/namespace.c 文件内，系统调用需要两个参数，分别是路径名和标记参数。卸载操作卸载的是挂载点最后（最近）一次挂载的文件系统。

umount()系统调用标记参数取值如下（/include/linux/fs.h）：

```
#define   MNT_FORCE             0x00000001   /* Attempt to forcibily umount */
#define   MNT_DETACH            0x00000002   /*只是将 mount 实例从管理结构体中移出*/
#define   MNT_EXPIRE            0x00000004   /* Mark for expiry */
#define   UMOUNT_NOFOLLOW       0x00000008   /* Don't follow symlink on umount */
#define   UMOUNT_UNUSED         0x80000000   /* Flag guaranteed to be unused */

SYSCALL_DEFINE2(umount, char __user *, name, int, flags)
{
```

```
    struct path path;
    struct mount *mnt;
    int retval;
    int lookup_flags = 0;     /*路径搜索标记*/

    if (flags & ~(MNT_FORCE | MNT_DETACH | MNT_EXPIRE | UMOUNT_NOFOLLOW))
        return -EINVAL;       /*卸载标记参数只能设置这些标记位，不能设置除此之外的标记位*/

    if (!may_mount())         /*权限检查*/
        return -EPERM;

    if (!(flags & UMOUNT_NOFOLLOW))
        lookup_flags |= LOOKUP_FOLLOW;

    retval = user_path_mountpoint_at(AT_FDCWD, name, lookup_flags, &path);
                                        /*挂载点目录项信息（最后一次挂载），/fs/namei.c*/
    if (retval)
        goto out;
    mnt = real_mount(path.mnt);              /*返回挂载 mount 实例指针*/
    retval = -EINVAL;
    if (path.dentry != path.mnt->mnt_root)        /*指定路径必须是挂载点*/
        goto dput_and_out;
    if (!check_mnt(mnt))
        goto dput_and_out;
    if (mnt->mnt.mnt_flags & MNT_LOCKED)          /*锁定挂载的文件系统不可卸载*/
        goto dput_and_out;
    retval = -EPERM;
    if (flags & MNT_FORCE && !capable(CAP_SYS_ADMIN))
        goto dput_and_out;

    retval = do_umount(mnt, flags);     /*执行卸载操作，/fs/namespace.c*/
dput_and_out:
    dput(path.dentry);                  /*释放挂载文件系统根目录项 dentry 实例*/
    mntput_no_expire(mnt);              /*释放 mount 和 super_block 实例，/fs/namespace.c*/
out:
    return retval;
}
```

umount()系统调用首先通过 user_path_mountpoint_at()函数查找挂载点最后（最近）一次挂载文件系统的根目录项和 mount 实例，卸载此文件系统，然后将卸载工作交给 do_umount(mnt, flags)函数完成，最后调用 dput(path.dentry)函数释放挂载文件系统根目录项 dentry 实例，调用 mntput_no_expire(mnt)函数释放 mount 实例和文件系统超级块 super_block 实例。

do_umount(mnt, flags)函数完成的主要工作是将 mount 实例从全局散列表和 mountpoint 实例管理链表中

移出，释放 mountpoint 实例，清除挂载点 dentry 实例 DCACHE_MOUNTED 标记位，即断开挂载点目录项和挂载文件系统根目录项之间的关联。

do_umount(mnt, flags)函数定义如下（/fs/namespace.c）：

```
static int do_umount(struct mount *mnt, int flags)
{
    struct super_block *sb = mnt->mnt.mnt_sb;
    int retval;

    retval = security_sb_umount(&mnt->mnt, flags);
    if (retval)
        return retval;

    if (flags & MNT_EXPIRE) {        /*自动过期文件系统*/
        ...
    }

    if (flags & MNT_FORCE && sb->s_op->umount_begin) {
        sb->s_op->umount_begin(sb);              /*调用超级块操作结构体中定义的卸载前操作函数*/
    }

    if (&mnt->mnt == current->fs->root.mnt && !(flags & MNT_DETACH)) {
        if (!capable(CAP_SYS_ADMIN))
            return -EPERM;
        down_write(&sb->s_umount);
        if (!(sb->s_flags & MS_RDONLY))     /*没有定义只读挂载标记*/
            retval = do_remount_sb(sb, MS_RDONLY, NULL, 0);
                                /*如果卸载的是进程根目录所在的文件系统，则将其转为只读挂载*/
        up_write(&sb->s_umount);
        return retval;
    }

    namespace_lock();
    lock_mount_hash();
    event++;

    if (flags & MNT_DETACH) {                /*强制卸载文件系统*/
        if (!list_empty(&mnt->mnt_list))        /*mount 在挂载命名空间链表中*/
            umount_tree(mnt, UMOUNT_PROPAGATE);     /*/fs/namespace.c*/
        retval = 0;
    } else {
        shrink_submounts(mnt);
        retval = -EBUSY;
        if (!propagate_mount_busy(mnt, 2)) {
```

```
                if (!list_empty(&mnt->mnt_list))
                    umount_tree(mnt, UMOUNT_PROPAGATE|UMOUNT_SYNC);
                retval = 0;
            }
        }
        unlock_mount_hash();
        namespace_unlock();
        return retval;
    }
```

do_umount()函数内调用 umount_tree()函数完成 mount 实例及其子 mount 实例的卸载操作。如果是共享挂载，会将此操作应用于对等组内其它挂载。

umount_tree()函数对各 mount 实例，将其从全局散列表、mountpoint 实例链表及挂载树中移出。如果没有其它挂载了，将释放 mountpoint 实例，并清挂载点 dentry 实例的 DCACHE_MOUNTED 标记位。请读者自行阅读源代码。

在 do_umount(mnt, flags)函数之后，umount()系统调用随后调用 **mntput_no_expire(mnt)**函数，将 mount 引用计数减 1，如果为引用计数至 0，会将 mount 实例从超级块实例 s_mounts 成员链表中移出，触发工作队列中工作完成 mount 和 super_block 实例的释放。由于超级块实例的释放还涉及到文件系统的回写、文件系统 dentry 和 inode 缓存的释放等工作（第 11 章再做介绍），因此将其交由工作队列延迟进行，工作队列执行函数为**__cleanup_mnt()**（/fs/namespace.c）。

另外，在 VFS 的通用函数中可能会调用 mntget(mnt)和 mntput(mnt)函数，增加或减小 mount 引用计数，后者是 mntput_no_expire(mnt)函数的包装器，在 mount 引用计数为 0 时将触发 mount 和 super_block 实例的释放。也就是说，在卸载文件系统后，其 mount 和 super_block 实例，以及 dentry 和 inode 实例仍将保存在内存中，直至引用计数为 0（回写后），释放各实例，不过文件系统卸载后则对用户进程已经不可见了。

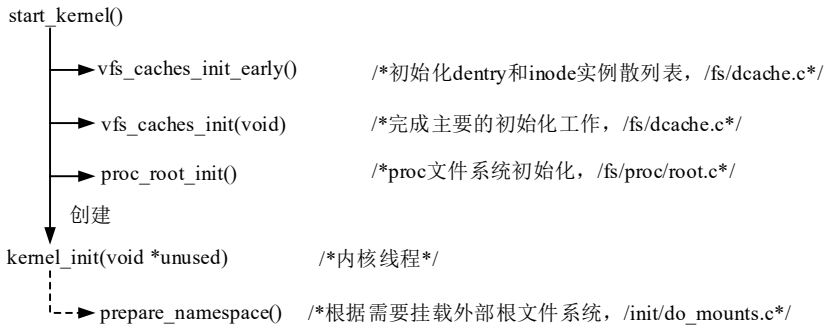
文件系统卸载操作完成后，挂载点在被卸载文件系统之前挂载的文件系统将自动恢复可见。

7.5 VFS 初始化

前面几节介绍了虚拟文件系统（VFS）的结构，具体文件系统的挂载和卸载，本节介绍虚拟文件系统的初始化。虚拟文件系统初始化工作主要包括各数据结构 slab 缓存的创建、管理散列表的创建，内核初始根文件系统的创建，以及挂载外部根文件系统等。

内核在启动阶段后期创建的 kernel_init 线程中需要根据情况决定是否挂载外部介质文件系统作为内核根文件系统。具体文件系统类型的注册在内核启动后期初始化子系统时或加载模块时完成。

内核启动 start_kernel()函数中调用的与 VFS 初始化相关的函数如下图所示：



vfs_caches_init_early()函数（早期初始化函数）创建并初始化管理 dentry 和 inode 实例的全局散列表，vfs_caches_init()函数（后期初始化函数）完成 VFS 主要的初始化工作，包括挂载初始根文件系统等，

proc_root_init()函数主要完成 proc 文件系统的初始化，详见本章下文。

start_kernel()函数在后期创建内核线程 kernel_init，线程内根据情况决定是否调用 prepare_namespace()函数挂载外部介质文件系统作为内核根文件系统。

7.5.1 早期初始化

vfs_caches_init_early()函数定义在/fs/dcache.c 文件内，代码如下：

```
void __init vfs_caches_init_early(void)
{
    dcache_init_early();      /*创建并初始化管理 dentry 实例全局散列表，/fs/dcache.c*/
    inode_init_early();       /*创建并初始化管理 inode 实例全局散列表，/fs/inode.c*/
}
```

dcache_init_early()函数创建并初始化管理 dentry 实例的全局散列表。

inode_init_early()函数创建并初始化管理 inode 实例的全局散列表。

7.5.2 后期初始化

vfs_caches_init()函数完成 VFS 主要的初始化工作，函数定义在/fs/dcache.c 文件内，代码如下：

```
void __init vfs_caches_init(void)
{
    names_cachep = kmem_cache_create("names_cache", PATH_MAX, 0,   \
                        SLAB_HWCACHE_ALIGN|SLAB_PANIC, NULL);
                                        /*创建保存路径名称字符串的 slab 缓存*/

    dcache_init();          /*创建 dentry 结构体 slab 缓存，/fs/dcache.c*/
    inode_init();           /*创建 inode 结构体 slab 缓存，/fs/inode.c*/
    files_init();           /*创建 file 结构体（表示进程文件）slab 缓存，见下一节，/fs/file_table.c*/
    files_maxfiles_init();  /*设置 files_stat.max_files 成员（files_stat_struct 实例），/fs/file_table.c*/
    mnt_init();             /*挂载 roofs 作为初始根文件系统，见下文，/fs/namespace.c*/
    bdev_cache_init();      /*块设备 bdev 伪文件系统初始化，见第 10 章*/
    chrdev_init();          /*字符设备库初始化，见第 9 章*/
}
```

vfs_caches_init()函数主要完成 VFS 中各数据结构 slab 缓存的创建，挂载初始化等工作。下面主要介绍进程文件结构的初始化函数和挂载初始化函数。

1 进程文件初始化

进程打开的文件由 file 结构体表示，它将关联到内核根文件系统中的 inode 实例，file 结构体定义详见下一节。

files_init()函数定义在/fs/file_table.c 文件内，主要完成 file 结构体 slab 缓存的创建，代码如下：

```
void __init files_init(void)
{
    filp_cachep = kmem_cache_create("filp", sizeof(struct file), 0,   \
                    SLAB_HWCACHE_ALIGN | SLAB_PANIC, NULL);   /*创建 file 结构体 slab 缓存*/
    percpu_counter_init(&nr_files, 0, GFP_KERNEL);        /*初始化全局变量 nr_files*/
```

```
}

files_maxfiles_init()函数定义在/fs/file_table.c 文件内，代码如下：
void __init files_maxfiles_init(void)
{
    unsigned long n;
    unsigned long memreserve = (totalram_pages - nr_free_pages()) * 3/2;

    memreserve = min(memreserve, totalram_pages - 1);
    n = ((totalram_pages - memreserve) * (PAGE_SIZE / 1024)) / 10;

    files_stat.max_files = max_t(unsigned long, n, NR_FILE);     /*/fs/file_table.c*/
                    /*设置全局 files_stat_struct 结构体实例 files_stat 最大文件数量成员*/
}
```

2 挂载初始化

挂载初始化函数 mnt_init()主要是创建挂载相关数据结构的 slab 缓存，创建初始根文件系统，此函数定义在/fs/namespace.c 文件内，代码如下：

```
void __init mnt_init(void)
{
    unsigned u;
    int err;

    mnt_cache = kmem_cache_create("mnt_cache", sizeof(struct mount),     \
                    0, SLAB_HWCACHE_ALIGN | SLAB_PANIC, NULL);
                                                    /*创建挂载 mount 结构体 slab 缓存*/
    mount_hashtable = alloc_large_system_hash("Mount-cache",
                    sizeof(struct hlist_head),
                    mhash_entries, 19,
                    0,
                    &m_hash_shift, &m_hash_mask, 0, 0);          /*创建管理 mount 实例散列表*/
    mountpoint_hashtable = alloc_large_system_hash("Mountpoint-cache",
                    sizeof(struct hlist_head),
                    mphash_entries, 19,
                    0,
                    &mp_hash_shift, &mp_hash_mask, 0, 0);        /*创建管理 mountpoint 实例散列表*/

    if (!mount_hashtable || !mountpoint_hashtable)
                panic("Failed to allocate mount hash table\n");

    for (u = 0; u <= m_hash_mask; u++)                          /*初始化散列表*/
            INIT_HLIST_HEAD(&mount_hashtable[u]);
    for (u = 0; u <= mp_hash_mask; u++)
```

```
        INIT_HLIST_HEAD(&mountpoint_hashtable[u]);

    kernfs_init();      /*创建 kernfs_node 结构体 slab 缓存（用于 kernfs 文件系统），见本章下文*/

    err = sysfs_init();     /*sysfs 文件系统初始化，见本章下文*/
    ...    /*错误处理*/

    fs_kobj = kobject_create_and_add("fs", NULL);     /*在 sysfs 文件系统根目录下创建 fs 目录项*/
    ...    /*错误处理*/
    init_rootfs();              /*注册 rootfs 文件系统类型，/init/do_mounts.c*/
    init_mount_tree();      /*挂载 rootfs 文件系统作为初始根文件系统，/fs/namespace.c*/
}
```

mnt_init()函数中包含 kernfs、sysfs 文件系统的初始化，后面再介绍这两个文件系统。

这里需要重点介绍一下 init_rootfs()和 init_mount_tree()函数，前者用于注册 rootfs 文件系统类型，后者用于挂载 rootfs 文件系统作为初始根文件系统。

rootfs 是只存在于内存中的文件系统，它可以是 ramfs 或 tmpfs 其中之一。

内核在 mnt_init()函数中先挂载 rootfs 文件系统作为初始根文件系统，随后在 kernel_init 内核线程中根据需要挂载外部介质文件系统至 rootfs 文件系统，并将挂载的外部文件系统作为内核根文件系统（新）。

在介绍函数实现前我们先了解一下几个非常重要的概念：

●**initrd**：内存盘，保存内核初始挂载的内存文件系统。它实际上是保存在内存中的 ext2 文件系统，独立于内核镜像存在，引导加载程序加载完内核镜像后，还需要将 initrd 加载到内存，并将起始地址和大小通过命令行参数（rd_start 和 rd_size）传递给内核。目前已经弃用而被 initramfs 代替。

●**ramfs**：基于内存的简易文件系统类型，其代码位于/fs/ramfs/目录下。ramfs 文件系统是完全基于虚拟文件系统数据结构实例的文件系统（不存在后备存储设备），由 dentry 实例表示目录项，inode 实例表示节点，由 inode 实例中文件地址空间基树表示的页缓存保存文件内容。文件系统没有大小限制，文件内容不能交换至外部交换区。ramfs 文件系统类型代码始终编译入内核。

●**tmpfs**：ramfs 文件系统类型的增强版，对文件系统大小进行了限制，文件内容可交换至交换区。需选择 TMPFS（需选择 SHMEM 选项）配置选项，文件系统类型代码位于/mm/shmem.c 文件内。tmpfs 文件系统类型不仅可用于内核根文件系统，还可用于进程间通信的共享内存机制等。

●**rootfs**：初始根文件系统类型，可以是 ramfs 或 tmpfs 其中之一。内核在以下条件都成立时选择 tmpfs 作为初始根文件系统类型，否则选用 ramfs 文件系统类型：

（1）选择了 TMPFS 配置选项，表示支持 tmpfs 文件系统。

（2）传递了命令行参数"rootfstype=tmpfs"或未定义。

（3）命令行参数"root=***"未定义。

●**initramfs**：保存初始根文件系统内容，它是一个.cpio 类型的文件，链接内核时保存在内核镜像的初始化段中。内核在 do_basic_setup()函数中，初始化子系统时调用 **populate_rootfs()**函数（/init/initramfs.c）将 initramfs 的内容解压至根文件系统中。initramf 具有默认的内容（/usr/），用户可通过配置选项指定编入其中的文件夹，编译内核时会将指文件夹的内容编译入 initramfs 内，目标文件格式为.cpio。使用 initramfs 传递根文件系统内容需要选择 BLK_DEV_INITRD 配置选项，并在命令行参数中添加"noinitrd"参数。

■注册 rootfs

下面来看一下 init_rootfs()函数的实现，函数代码如下（/init/do_mounts.c）：

```
int __init init_rootfs(void)
{
    int err = register_filesystem(&rootfs_fs_type);      /*注册 rootfs 文件系统类型*/
    ...     /*错误处理*/

    if (IS_ENABLED(CONFIG_TMPFS) && !saved_root_name[0] &&
                                (!root_fs_names || strstr(root_fs_names, "tmpfs"))) {   /*rootfs 为 tmpfs*/
        err = shmem_init();       /*注册 tmpfs 文件系统类型并挂载（内核挂载），/mm/shmem.c*/
        is_tmpfs = true;
    } else {                /*rootfs 为 ramfs*/
        err = init_ramfs_fs();          /*注册 ramfs 文件系统类型，/fs/ramfs/inode.c*/
    }
    if (err)
        unregister_filesystem(&rootfs_fs_type);
    return err;
}
```

init_rootfs()函数内首先注册 rootfs 文件系统类型，然后判断 rootfs 文件系统类型是 tmpfs 还是 ramfs，分别对 tmpfs 或 ramfs 文件系统进行初始化。rootfs 文件系统类型的 mount()函数中挂载 tmpfs 或 ramfs 文件系统。

注意 shmem_init()函数内挂载（内核挂载）的 tmpfs 文件系统并不是作为内核根文件系统，而是用于共享内存等用途。

■初始化根文件系统

挂载初始化函数 mnt_init()中调用的 init_mount_tree()函数用于挂载 rootfs 文件系统作为初始根文件系统，函数定义在/fs/namespace.c 文件内：

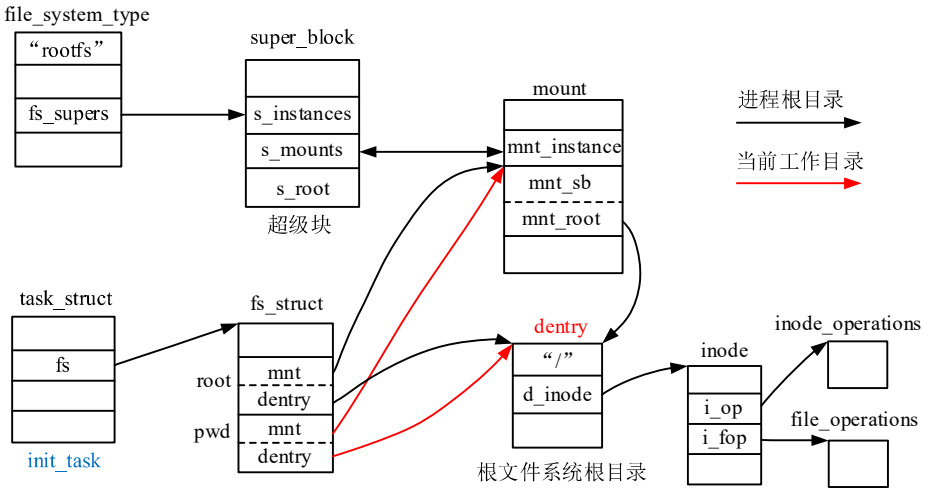
```
static void __init init_mount_tree(void)
{
    struct vfsmount *mnt;
    struct mnt_namespace *ns;
    struct path root;
    struct file_system_type *type;

    type = get_fs_type("rootfs");       /*获取文件系统类型实例指针*/
    ...  /*错误处理*/
    mnt = vfs_kern_mount(type, 0, "rootfs", NULL);          /*内核挂载*/
                                /*挂载 rootfs 文件系统（没有关联到挂载点），/fs/namespace.c*/
    put_filesystem(type);
    ...  /*错误处理*/

    ns = create_mnt_ns(mnt);        /*创建初始挂载命名空间，见上一节，/fs/namespace.c*/
    ...  /*错误处理*/
    init_task.nsproxy->mnt_ns = ns;         /*指向初始挂载命名空间*/
    get_mnt_ns(ns);
```

```
    root.mnt = mnt;
    root.dentry = mnt->mnt_root;            /*rootfs 文件系统根目录项*/
    mnt->mnt_flags |= MNT_LOCKED;       /*锁定，不可卸载*/

    set_fs_pwd(current->fs, &root);    /*设置 init 进程当前工作目录为 rootfs 文件系统根目录*/
    set_fs_root(current->fs, &root);   /*设置 init 进程根目录为 rootfs 文件系统根目录*/
}
```

init_mount_tree()函数对 rootfs 文件系统执行内核挂载，即创建超级块 super_block、根目录项 dentry、节点 inode 实例、挂载 mount 结构体实例等。此时内核还不存在根文件系统，因此无法关联挂载点。实际上此时创建的 rootfs 文件系统根目录项，就是初始内核根文件系统的根目录项。这时 rootfs 文件系统的内容为空，内核在启动后期，初始化子系统时调用 **populate_rootfs()**函数将 initramfs 中的内容解压至 rootfs 文件系统。

在 init_mount_tree()函数最后，将 rootfs 文件系统根目录项设为内核（init 进程）的根目录和当前工作目录，即 init 进程（内核自身）能看见整个内核根文件系统。

init_mount_tree()函数执行结果如下图所示：



每个进程有一个根目录和当前工作目录（由 fs_struct 结构体表示），这两个目录指向内核根文件系统中的一个目录。根目录是进程能看见内核根文件系统的起点，也就是说此目录以上的部分对进程不可见，进程只能看到此目录以下的部分。进程能看到的文件系统是根文件系统的子集。当前工作目录，即在不指定的情况下，进程在当前工作目录下搜索、打开文件等。

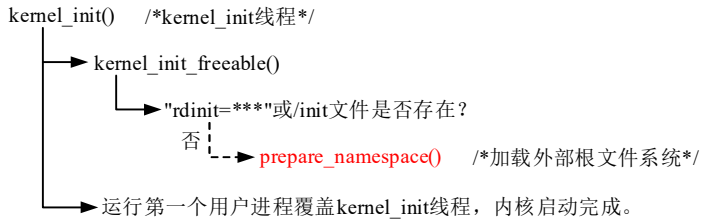
在 init_mount_tree()函数中将内核 init 进程的根目录和当前工作目录都设为 rootfs 文件系统根目录，即 init 进程可以看见整个根文件系统。init 进程创建子进程时，其根目录和当前工作目录信息会传递给子进程。

7.5.3 挂载外部根文件系统

在挂载初始化函数 mnt_init()中已经挂载了 rootfs 文件系统作为初始根文件系统。在启动阶段末期创建的 kernel_init 内核线程中会将 initramfs 中的内容导入到根文件系统。kernel_init 内核线程随后会判断能否从 rootfs 文件系统运行第一个用户进程，如果可以则从中运行第一个用户进程（覆盖 kernel_init 线程），内核启动完成。

如果不能从 rootfs 中运行第一个用户进程，则调用 prepare_namespace()函数挂载外部介质文件系统作为根文件系统，并从中运行第一个用户进程，内核启动完成。

kernel_init 内核线程执行函数如下图所示：



从 rootfs 文件系统中运行第一个用户进程时，可执行文件由命令行参数"**rdinit=***"指定，若未指定则设为"/init"。

从外部介质根文件系统中运行第一个用户进程时，可执行文件由命令行参数"**init=***"指定，若未指定则依次尝试运行以下可执行文件：

●/sbin/init
●/etc/init
●/bin/init
●/bin/sh

由上可知，运行第一个用户进程加载可执行文件的优先顺序为：
/*rootfs 根文件系统*/
●rdinit=***
●/init
/*加载外部介质根文件系统*/
●init=***
●/sbin/init
●/etc/init
●/bin/init
●/bin/sh

若需要加载外部介质根文件系统，kernel_init 线程将调用 **prepare_namespace()**函数挂载外部介质文件系统作为根文件系统，并从中运行第一个用户进程。

挂载外部介质文件系统相关的命令行参数有：
●**"root=***"**：块设备文件名，参数值保存至 saved_root_name 变量（字符数组）。
●**"ro"，"rw"**：只读、读写模式挂载外部介质文件系统，参数值保存至 root_mountflags 变量。
●"rootdelay=***"：延时挂载的时间，等待块设备准备好，参数值保存至 root_delay 变量。
●**"rootfstype=***"**：外部介质文件系统类型，参数值保存至 root_fs_names 变量。
●"rootflags=***"：挂载数据，参数值保存至 root_mount_data 变量。
●"rootwait"：等待挂载介质准备好，类似于"rootdelay=***"，内核设置 root_wait 变量为 1。

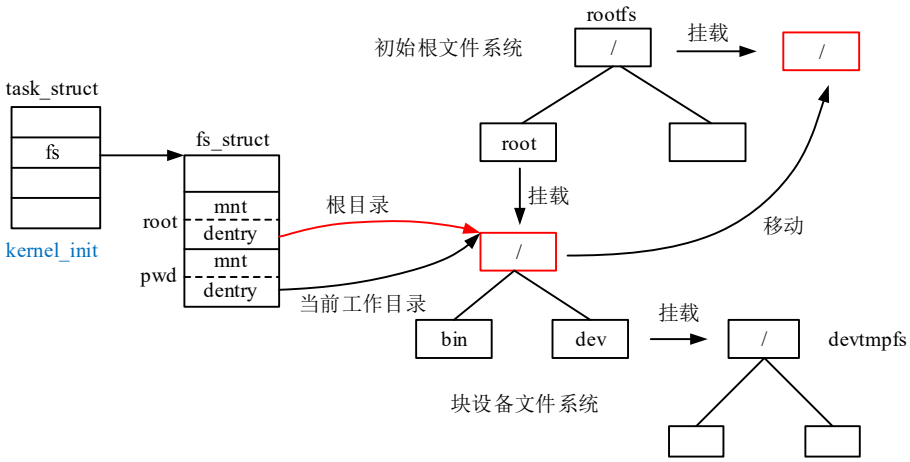
prepare_namespace()函数定义在/init/do_mounts.c 文件内，代码如下：

```
void __init   prepare_namespace(void)
{
    int is_floppy;

    if (root_delay) {
        /*延时加载，等待块设备初始化完成，root_delay 值由"rootdelay=*"命令行参数传递*/
```

```
        printk(KERN_INFO "Waiting %d sec before mounting root device...\n",root_delay);
        ssleep(root_delay);         /*kernel_init 线程睡眠*/
    }
    wait_for_device_probe();    /*等待设备激活，/drivers/base/dd.c*/

    md_run_setup();

    if (saved_root_name[0])    /*如果定义了命令行参数“root=***”*/
    {
        root_device_name = saved_root_name;    /*外部介质设备名称字符串指针*/
        if (!strncmp(root_device_name, "mtd", 3) ||!strncmp(root_device_name, "ubi", 3))
        {       /*如果 root 参数值为"mtd"或"ubi"，运行以下代码*/
            mount_block_root(root_device_name, root_mountflags);
                                                    /*挂载外部介质，/init/do_mounts.c*/
            goto out;
        }
         /*如果 root 参数值不为"mtd"或"ubi"，运行以下代码*/
        ROOT_DEV = name_to_dev_t(root_device_name);      /*设备名称转设备号*/
        if (strncmp(root_device_name, "/dev/", 5) == 0)
            root_device_name += 5;        /*跳过名称字符串前面的"/dev/"5 个字符*/
    }

    /*使用 initramfs（设置命令行参数“noinitrd”），ininrd_load()返回 0*/
    if (initrd_load())       /*/init/do_mounts_initrd.c*/
        goto out;

    if ((ROOT_DEV == 0) && root_wait)       /*没有传递 root 命令行参数（或无效）*/
    {
        printk(KERN_INFO "Waiting for root device %s...\n",saved_root_name);
        while (driver_probe_done() != 0 ||(ROOT_DEV = name_to_dev_t(saved_root_name)) == 0)
            msleep(100);
        async_synchronize_full();
     }

     is_floppy = MAJOR(ROOT_DEV) == FLOPPY_MAJOR;

     if (is_floppy && rd_doload && rd_load_disk(0))
          ROOT_DEV = Root_RAM0;

     mount_root();
        /*挂载外部介质文件系统至/root，并设置为线程当前工作目录，/init/do_mounts.c*/
out:
     devtmpfs_mount("dev");
```

```
                                /*挂载 devtmpfs 文件系统至/dev，/drivers/base/devtmpfs.c*/
    sys_mount(".", "/", NULL, MS_MOVE, NULL);
                                /*移动块设备文件系统至 rootfs 文件系统根目录（挂载）*/
    sys_chroot(".");    /*设置当前目录为 kernel_init 内核线程根目录，最后传递给用户进程*/
}
```

prepare_namespace()函数将命令行参数"root=***"传递的设备名称转换成块设备号，保存至全局变量 ROOT_DEV，随后调用 mount_root()函数将外部块设备文件系统挂载至 rootfs 文件系统/root 目录下，并设置为线程当前工作目录，挂载 devtmpfs 文件系统至块设备文件系统/dev 目录，然后，将块设备文件系统移动到初始根文件系统根目录（挂载），最后设置块设备文件系统根目录为 kernel_init 内核线程根目录。至此外部介质文件系统根目录成为 kernel_init 内核线程根目录和当前工作目录，如下图所示（初始根文件系统不可见了）。



kernel_init 线程随后从外部介质文件系统中运行第一个用户进程，由于第一个用户进程将覆盖 kernel_init 线程，而所有其它用户进程都是由第一个用户进程派生而来，因此外部介质文件系统根目录默认成为所有用户进程的根目录。

下面介绍 prepare_namespace()函数中调用的 name_to_dev_t()和 mount_root()函数的实现。

1 块设备文件名转设备号

命令行参数“root=***”向内核传递挂载外部文件系统存储介质的信息，其值可以是设备文件名称、主从设备号等形式，例如：/dev/hd1，2：0。更多的参数形式请参考/init/do_mounts.c 文件内函数说明。

name_to_dev_t(const char *name)函数负责将 root 命令行参数传递的存储介质信息转换成设备号，保存至 ROOT_DEV 全局变量，以便在内核中查找介质对应的数据结构，以执行挂载操作。

name_to_dev_t()函数定义在/init/do_mounts.c 文件内，代码如下：

```
dev_t name_to_dev_t(const char *name)
/*name：设备名称字符串指针*/
{
    char s[32];
    char *p;
    dev_t res = 0;        /*外部介质设备号*/
    int part;

#ifdef CONFIG_BLOCK
    if (strncmp(name, "PARTUUID=", 9) == 0) {     /*字符串前 9 个字符为"PARTUUID="*/
```

```
        name += 9;
        res = devt_from_partuuid(name);
        if (!res)
            goto fail;
        goto done;
    }
#endif

    if (strncmp(name, "/dev/", 5) != 0) {
                        /*前 5 个字符不是“/dev/”的情形，处理直接传递主从设备号的情形*/
        unsigned maj, min, offset;
        char dummy;

        if ((sscanf(name, "%u:%u%c", &maj, &min, &dummy) == 2) ||
            (sscanf(name, "%u:%u:%u:%c", &maj, &min, &offset, &dummy) == 3)) {
              res = MKDEV(maj, min);       /*生成设备号*/
              if (maj != MAJOR(res) || min != MINOR(res))
                 goto fail;
        } else {
            res = new_decode_dev(simple_strtoul(name, &p, 16));
            if (*p)
                goto fail;
        }
        goto done;
    }

    /*处理“/dev/hd1”形式的参数值*/
    name += 5;                  /*跳过前面“/dev/”字符*/
    res = Root_NFS;
    if (strcmp(name, "nfs") == 0)       /*后面字符为"nfs"*/
        goto done;
    res = Root_RAM0;
    if (strcmp(name, "ram") == 0)       /*后面字符为"ram"*/
        goto done;

    if (strlen(name) > 31)
        goto fail;
    strcpy(s, name);          /*复制字符串*/
    for (p = s; *p; p++)        /*将‘/’字符换成‘!’，p 指向字符串末尾，*/
      if (*p == '/')
          *p = '!';
    res = blk_lookup_devt(s, 0);       /*不带从设备号，返回查找的主设备号，/block/genhd.c*/
              /*以设备名称在块设备库中查找块设备数据结构实例获取设备号，/block/genhd.c*/
```

```
    ...     /*错误处理*/
    while (p > s && isdigit(p[-1]))
        p--;
    if (p == s || !*p || *p == '0')
        goto fail;

    /*处理带从设备号的情况*/
    part = simple_strtoul(p, NULL, 10);       /*获取从设备号*/
    *p = '\0';
    res = blk_lookup_devt(s, part);           /*查找并生成块设备号*/
    ...     /*错误处理*/

    /*处理形如“/dev/hdp1”的情况，从设备号前有 p 字符*/
    if (p < s + 2 || !isdigit(p[-2]) || p[-1] != 'p')
        goto fail;
    p[-1] = '\0';
    res = blk_lookup_devt(s, part);         /*查找并生成块设备号*/
    ...     /*错误处理*/
fail:
    return 0;
done:
    return res;     /*返回块设备号*/
}
```

root 命令行参数值的形式主要有三种，分别是"PARTUUID=***"，“2：1”和“/dev/hd1”（或“/dev/hdp1”）。设备名称转设备号函数需要对这三个情形进行处理。通常使用设备文件名的形式，函数内通过设备名称在块设备驱动数据结构中查找相应的实例，从中获取设备号。

2 挂载块设备文件系统

在获取块设备号之后，prepare_namespace()函数调用 mount_root()函数挂载块设备文件系统，代码如下：

```
void __init mount_root(void)
{
    #ifdef CONFIG_ROOT_NFS
        ...
    #endif
    #ifdef CONFIG_BLK_DEV_FD
       ...
    #endif
    #ifdef CONFIG_BLOCK
     {
        int err = create_dev("/dev/root", ROOT_DEV);
                        /*在 rootfs 文件系统中创建块设备文件/dev/root，/init/do_mounts.h*/
        ...     /*错误处理*/
        mount_block_root("/dev/root", root_mountflags);
```

```
                    /*将块设备文件系统挂载到 rootfs 文件系统/root 目录下，/init/do_mounts.c*/
        }
        #endif
}
```

这里以挂载普通的块设备文件系统为例来说明挂载的过程，首先调用 create_dev()函数在 rootfs 文件系统中创建块设备文件/dev/root（/dev 目录已经存在），然后调用 mount_block_root()函数将块设备文件系统挂载到 rootfs 文件系统/root 目录下，并设为 kernel_init 线程的当前工作目录。

mount_block_root()函数实现如下，函数内需要命令行参数传递的块设备文件系统类型及加载数据参数：

```
void __init mount_block_root(char *name, int flags)
/*name：块设备文件名称，flags：挂载标记*/
{
    struct page *page = alloc_page(GFP_KERNEL |__GFP_NOTRACK_FALSE_POSITIVE);
    char *fs_names = page_address(page);
    char *p;
#ifdef CONFIG_BLOCK
    char b[BDEVNAME_SIZE];
#else
    const char *b = name;
#endif
    /*"rootfstype=***"命令行参数传递的块设备（外部介质）文件系统类型*/
    get_fs_names(fs_names);
            /*fs_names 保存文件系统类型名称，如果未传递则保存所有注册的文件系统类型名称*/
retry:
    for (p = fs_names; *p; p += strlen(p)+1) {      /*遍历所有注册的文件系统类型*/
        int err = do_mount_root(name, p, flags, root_mount_data); /*执行挂载操作，/init/do_mounts.c*/
        switch (err) {
            case 0:
                goto out;       /*挂载成功，跳转至 out 处*/
            case -EACCES:
            case -EINVAL:
                continue;
        }
            ...   /*输出信息*/
    }
    if (!(flags & MS_RDONLY)) {
            flags |= MS_RDONLY;
            goto retry;
    }
    ...     /*输出信息*/
out:
    put_page(page);
}
```

mount_block_root()函数内分配一个物理页面（fs_names）保存文件系统类型字符串，如果传递了命令

行参数 rootfstype，则只保存指定的文件系统类型名称字符串，否则查找内核 file_systems 链表，提取所有注册的文件系统类型的名称字符串，保存至物理页面。

随后，调用 do_mount_root()函数将块设备文件系统挂载到 rootfs 文件系统/root 目录下，并将挂载点（/root）设为 kernel_init 线程当前工作目录。

do_mount_root()函数定义如下：

```
static int __init do_mount_root(char *name, char *fs, int flags, void *data)
{
    struct super_block *s;
    int err = sys_mount(name, "/root", fs, flags, data);    /*mount()系统调用，挂载点/root*/
    ...   /*错误处理*/

    sys_chdir("/root");          /*修改 kernel_init 线程当前工作目录为/root*/
    s = current->fs->pwd.dentry->d_sb;
    ROOT_DEV = s->s_dev;
    ...     /*输出信息*/
    return 0;
}
```

prepare_namespace()函数在调用 mount_root()函数挂载外部介质文件系统至/root，并设置为线程当前工作目录后，将挂载的外部介质文件系统移动至 rootfs 文件系统的根目录上（移动挂载），最后设置 kernel_init 线程根目录也为挂载外部介质文件系统的根目录，挂载外部文件系统成为内核根文件系统，原初始根文件系统对进程不可见了。

7.6 进程与 VFS

内核通过根文件系统管理所有挂载的文件系统及文件，这相当于一个文件库。进程在操作文件前，需要建立进程与根文件系统中文件之间的关联（打开操作）。用户进程可能能看到整个内核根文件系统，也可能只能看见根文件系统的一部分。进程根目录指向内核根文件系统中某个目录项，这是进程能看到内核根文件系统的起点，其上的目录对进程不可见，其下的目录对进程可见。

进程通过 file 结构体建立与根文件系统中文件之间的关联，进程具有一个 file 结构体指针数组，数组下标就是进程文件描述符，这是一个无符号整数。本节介绍进程目录、进程文件相关内容。

7.6.1 进程目录

根文件系统是内核全局唯一的目录项层次结构，管理所有挂载的文件系统和打开的文件。通常整个根文件系统对进程可见，但是有时内核为模拟多个独立的系统容器，根文件系统可能只有部分对进程可见。进程目录信息包括进程根目录和当前工作目录，根目录是进程通过绝对路径搜索文件的起点，也就是内核根文件系统中进程可见部分的根节点，当前工作目录是进程通过相对路径搜索文件的起点。

进程 task_struct 结构体中 fs 成员指向的 fs_struct 实例，包含进程的根目录和当前工作目录信息：

```
task_struct{
    ...
    struct fs_struct *fs;          /*进程目录信息*/
   ...
 }
```

fs_struct 结构体定义在/include/linux/fs_struct.h 头文件：

```
struct fs_struct {
    int users;              /*实例用户数量*/
    spinlock_t lock;
    seqcount_t seq;
    int umask;              /*新建文件屏蔽访问权限掩码，置 1 的位表示屏蔽该权限*/
    int in_exec;            /**/
    struct path    root, pwd;     /*进程根目录和当前工作目录*/
};
```

fs_struct 结构体主要成员简介如下：

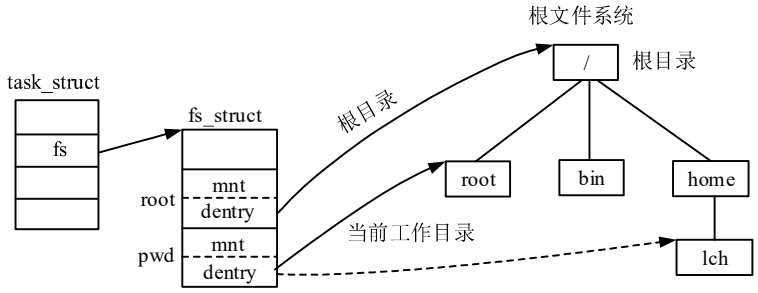
●**umask**：进程新建文件访问权限掩码，置 1 的位表示新建文件屏蔽该属性。

●**root，pwd**：表示进程根目录和当前工作目录信息，path 结构体实例，结构体定义如下：

```
struct path {
    struct vfsmount *mnt;       /*目录项所在文件系统挂载信息，vfsmount.mnt*/
    struct dentry *dentry;      /*目录项指针*/
};
```

root 成员表示进程访问内核根文件系统的起点（顶点），通常为根文件系统的根节点，如下图所示，但也可以通过 chroot()系统调用修改进程根目录。进程以绝对路径搜索文件时，从进程根目录开始。

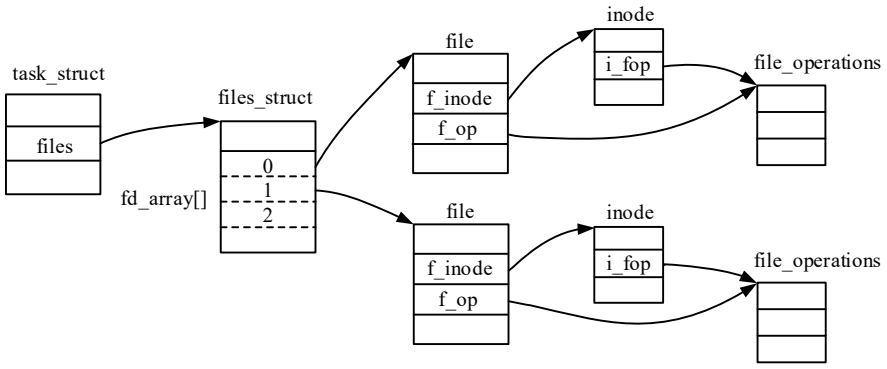
pwd 成员表示进程当前工作目录。进程以相对路径访问文件时，将会从当前工作目录开始查找。chdir()系统调用用于改变进程当前工作目录。在前面介绍的 VFS 初始化中，将创建初始根文件系统，并设置内核线程的根目录、当前工作目录为根文件系统根目录。



7.6.2 进程文件

打开文件由内核根文件系统统一管理，进程访问文件前，需要先通过打开文件操作建立与内核文件的关联。进程通过 file 结构体建立进程与内核文件（inode）之间的关联，如下图所示。进程 task_struct 结构体管理着 file 实例的指针数组（由 files_struct 结构体表示），通过 file 建立与内核文件 inode 之间的关联，数组项索引值即进程文件描述符。通常，进程与内核文件建立关联后，inode 指向的 file_operations 实例将赋予 file 实例，进程通过此 file_operations 实例中的函数操作文件。

另外，file 与 inode 实例并不是一一对应的关系。inode 唯一地表示内核中的文件，而多个进程可同时打开同一个文件，甚至同一个进程都可以对同一个文件拥有多个描述符（通过复制文件描述符实现），因此，file 与 inode 实例是多对一的关系。



1 文件描述符

进程 task_struct 结构体 files 成员指向 files_struct 实例，用于管理进程打开的文件：

```
task_struct{
    ...
    struct files_struct *files;     /*进程打开文件信息*/
    ...
}
```

files 成员指向 files_struct 实例，结构体定义在/include/linux/fdtable.h 头文件：

```
struct files_struct {
    atomic_t    count;                  /*实例引用计数*/
    bool    resize_in_progress;         /*进程正在扩展 fdtab 实例*/
    wait_queue_head_t    resize_wait;       /*进程等待队列，等待 fdtab 扩展的进程*/

    struct fdtable    __rcu    *fdt;        /*fdtable 结构体指针，初始值指向 fdtab 成员*/
    struct fdtable    fdtab;                /*fdtable 结构体成员*/
    spinlock_t    file_lock ____cacheline_aligned_in_smp;
    int    next_fd;              /*下一个打开文件的文件描述符，初始值为 0，每次分配描述符后设置*/
    unsigned long    close_on_exec_init[1];   /*执行 execve()系统调用时关闭文件的位图*/
    unsigned long    open_fds_init[1];        /*打开文件位图（比特位位置值就是文件描述符）*/
    struct file __rcu    * fd_array[NR_OPEN_DEFAULT];   /*打开文件 file 指针数组*/
};
```

files_struct 结构体主要成员简介如下：

●**fdt**：fdtable 结构体指针，初始值指向 fdtab 成员。

●**fdtab**：fdtable 结构体成员，用于管理文件位图，结构体定义如下（/include/linux/fdtable.h）：

```
struct fdtable {
    unsigned int max_fds;               /*fdtable 能管理的打开文件最大数量，由位图大小决定*/
    struct file __rcu **fd;             /*指向 file 指针数组的指针*/
    unsigned long *close_on_exec;   /*执行 execve()系统调用时关闭文件的位图*/
    unsigned long *open_fds;        /*进程打开文件位图*/
    struct rcu_head rcu;            /*回调函数，释放 fdtable 实例*/
};
```

文件位图就是 file 指针数组对应的位图，每位对应指针数组中一项，比特位位置就是数组项索引，即

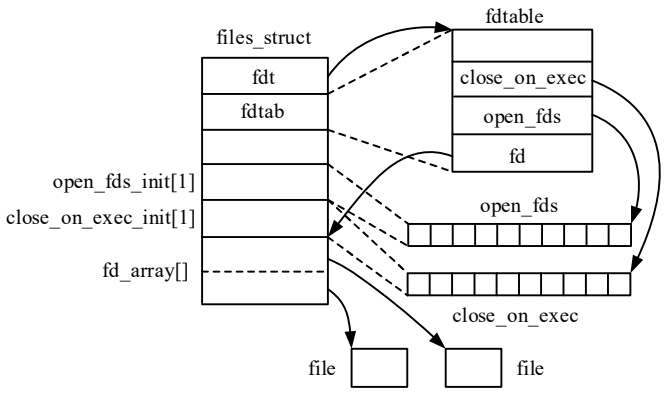
文件描述符。

●**open_fds_init[1]**：进程打开文件位图，与打开文件 file 指针数组对应。每个比特位表示对应数组项是否为空，0 表示数组项为空，尚未关联 file 实例，1 表示数组项已经关联了 file 实例。open_fds_init[1]成员为一个无符号整数，对于 32 位系统，此位图最多可表示 32 个打开文件。

●**close_on_exec_init[1]**：进程关闭文件位图，与 open_fds_init[1]类似，比特位置 1 表示对应文件在进程执行 execve()系统调用时需要关闭此文件，即打开文件不传递到新进程。0 表示不关闭文件，打开文件传递给新进程。close_on_exec_init[1]成员也是一个无符号整数，对于 32 位系统，此位图最多可表示 32 个文件。

●**fd_array[]**：file 指针数组，数组项指向 file 实例，指针数组项索引为文件描述符，无符号整数。数组项数 NR_OPEN_DEFAULT 与整型数比特位数相同。

由以上分析可知，files_struct 结构体初始能管理 NR_OPEN_DEFAULT（整数比特位数）个进程文件，files_struct 实例初始结构如下图所示：



内核自身（init 线程）初始 files_struct 结构体实例定义在/fs/file.c 文件内：

```
struct files_struct init_files = {
    .count          = ATOMIC_INIT(1),
    .fdt        = &init_files.fdtab,
    .fdtab          = {                     /*各成员指向 files_struct 结构体体中相应成员*/
        .max_fds = NR_OPEN_DEFAULT,
        .fd         = &init_files.fd_array[0],
        .close_on_exec = init_files.close_on_exec_init,
        .open_fds= init_files.open_fds_init,
    },
    .file_lock = __SPIN_LOCK_UNLOCKED(init_files.file_lock),
};
```

init_files 实例与上面分析的 files_struct 实例初始结构是一致的。

对于 32 位系统来说整型数的位数为 32，即 files_struct 实例初始状态能管理的打开文件最大数量为 32，这显然是不够用的。当打开文件数量超过 32 时，需要对 fdtable 实例进行扩展，扩展操作在进程申请未使用的文件描述符函数中进行。

扩展操作即为 fdtable 实例重新分配更大的 close_on_exec、open_fds 位图和 file 实例指针数组，将原 files_struct 实例中的信息复制到新分配的位图和 file 实例指针数组，fdtable 实例不再使用 files_struct 实例中的位图和 file 指针数组。如果一次扩展之后还需要再进行扩展，则重复以上操作，fdtable 实例使用新扩

展的位图和 file 指针数组（复制原位图和指针数组信息），而不使用扩展前的位图和 file 指针数组。

get_unused_fd_flags()函数用于申请未使用的进程文件描述符，函数定义如下（/fs/file.c）：

```
int get_unused_fd_flags(unsigned flags)
{
    return    __alloc_fd(current->files, 0, rlimit(RLIMIT_NOFILE), flags);     /*/fs/file.c*/
}
```

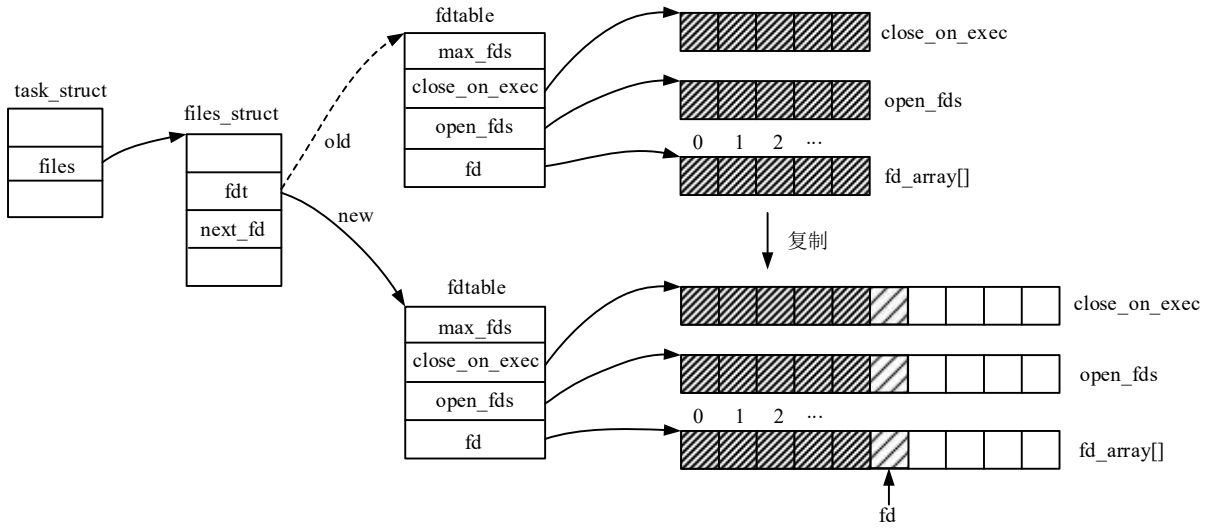
__alloc_fd()函数用于搜索/分配最小未使用文件描述符，函数定义如下：

```
int __alloc_fd(struct files_struct *files,unsigned start, unsigned end, unsigned flags)
/*start：起始搜索文件描述符，end：进程可用的最大文件描述符（资源限制），
*flags：标记参数，似乎并未使用。*/
{
    unsigned int fd;
    int error;
    struct fdtable *fdt;

    spin_lock(&files->file_lock);
repeat:
    fdt = files_fdtable(files);       /*files->fdt*/
    fd = start;
    if (fd < files->next_fd)
        fd = files->next_fd;

    if (fd < fdt->max_fds)       /*起始文件描述符没有超过 fdt 最大限制值*/
        fd = find_next_zero_bit(fdt->open_fds, fdt->max_fds, fd);
                /*在 fdt->open_fds 位图中查找 fd 之后第一个为 0 的比特位，返回其位置值*/

    error = -EMFILE;
    if (fd >= end)       /*超过了内核对进程的资源限制值，返回错误码*/
        goto out;

    error = expand_files(files, fd);     /*按需扩展 fdtable 实例，/fs/file.c*/
    if (error < 0)
        goto out;

    if (error)
        goto repeat;       /*执行 expand_files()函数时，进程可能进入睡眠，*/

    if (start <= files->next_fd)
        files->next_fd = fd + 1;   /*下次打开文件的描述符*/

    __set_open_fd(fd, fdt);       /*设置打开文件位图中相应位*/
    if (flags & O_CLOEXEC)   /*设置或清零关闭文件位图中的相应位*/
```

```
            __set_close_on_exec(fd, fdt);
        else
            __clear_close_on_exec(fd, fdt);
        error = fd;          /*文件描述符*/
    #if 1
        /*安全检查，fdt->fd[fd]是否为 NULL，尚未关联 file 实例*/
        if (rcu_access_pointer(fdt->fd[fd]) != NULL) {
            printk(KERN_WARNING "alloc_fd: slot %d not NULL!\n", fd);
            rcu_assign_pointer(fdt->fd[fd], NULL);
        }
    #endif

    out:
        spin_unlock(&files->file_lock);
        return error;     /*返回文件描述符*/
    }
```

files_struct 结构体中 next_fd 成员表示下次打开文件的描述符，初始值为 0。分配未使用的文件描述符时，从 files->fdt->open_fds 位图中 max(start,next_fd)位置开始查找第一个为 0 的比特位，如果所查找的比特位位于 open_fds 位图内部则无需扩展 fdtable 实例，返回比特位位置值作为文件描述符值，如果超出了位图范围则需要对 fdtable 实例进行扩展。

expand_files(files, fd)函数负责根据 fd 值决定是否扩展 fdtable 实例，如果需要则执行扩展操作，函数源代码请读者自行阅读。下图示意了扩展 fdtable 实例的过程：



__alloc_fd()函数随后对 next_fd 成员设置成当前分配的文件描述符值加 1，设置或清除位图中的相应位，返回获得的文件描述符。

内核在关闭文件时，如果关闭文件的文件描述符小于 next_fd 值，next_fd 将被设置为关闭文件的描述符，也就是说某一文件描述符被释放后将会被再次使用。

2 文件表示

进程打开的文件由 file 结构体表示，结构体定义在/include/linux/fs.h 头文件：

```
struct file {
```

```
    union {
        struct   llist_node    fu_llist;        /*单链表成员*/
        struct   rcu_head      fu_rcuhead;
    } f_u;
    struct path   f_path;                  /*文件路径信息*/
    struct inode      *f_inode;            /*指向内核文件 inode 实例*/
    const struct file_operations    *f_op;
                                /*文件操作结构指针，通常在打开文件时设为 inode->i_fop*/
    spinlock_t        f_lock;
    atomic_long_t         f_count;
    unsigned int          f_flags;      /*open()系统调用传递的 flags 标记参数*/
    fmode_t               f_mode;       /*标记进程以何种模式访问文件*/
    struct mutex          f_pos_lock;
    loff_t                f_pos;        /*文件当前位置，相对于文件开头处的字节偏移量*/
    struct fown_struct    f_owner;      /*文件属主（进程）信息，/include/linux/fs.h*/
    const struct cred     *f_cred;      /*文件凭证*/
    struct file_ra_state  f_ra;         /*文件预读结构体，见第 11 章*/
    u64           f_version;
#ifdef   CONFIG_SECURITY
    void          *f_security;
#endif
    void          *private_data;   /*文件私有数据指针，例如设备文件指向驱动程序定义的数据结构*/

#ifdef CONFIG_EPOLL             /*支持 epoll 机制*/
    struct list_head     f_ep_links;
    struct list_head     f_tfile_llink;
#endif
    struct address_space   *f_mapping;        /*文件地址空间指针，用于具有外部存储介质的文件*/
} __attribute__((aligned(4)));
```

file 结构体主要成员简介如下：

●**f_inode**：指向内核文件表示的 inode 结构体实例，inode 实例是文件在内核中的唯一表示。

●**f_op**：文件操作 file_operations 结构体指针，进程对文件的操作通过此结构体内定义的函数完成。

●**f_flags**：open()系统调用传递的 flags 标记参数，见下一节。

●**f_mode**：表示进程对文件的操作属性，在 vfs_open()函数中赋值，取值定义在/include/linux/fs.h 头文件内：

```
#define   FMODE_READ          ((__force fmode_t)0x1)    /*读操作打开*/
#define   FMODE_WRITE         ((__force fmode_t)0x2)    /*写操作打开*/
#define   FMODE_LSEEK         ((__force fmode_t)0x4)    /*可定位文件位置*/
#define   FMODE_PREAD         ((__force fmode_t)0x8)    /*可使用 pread()访问*/
#define   FMODE_PWRITE        ((__force fmode_t)0x10)   /*可使用 pwrite()访问*/
#define   FMODE_EXEC          ((__force fmode_t)0x20)   /*可执行*/
#define   FMODE_NDELAY        ((__force fmode_t)0x40)   /*无延迟打开（只适用于块设备文件）*/
```

```
#define   FMODE_EXCL          ((__force fmode_t)0x80)   /*独占式地打开文件，用于块设备文件*/
#define   FMODE_WRITE_IOCTL    ((__force fmode_t)0x100)
#define   FMODE_32BITHASH       ((__force fmode_t)0x200)
                                    /* 32bit hashes as llseek() offset (for directories) */

#define   FMODE_64BITHASH            ((__force fmode_t)0x400)
                                  /* 64bit hashes as llseek() offset (for directories) */

#define   FMODE_NOCMTIME   ((__force fmode_t)0x800)   /*不要更新 ctime 和 mtime. */
#define   FMODE_RANDOM      ((__force fmode_t)0x1000)   /*期望随机访问文件*/
#define   FMODE_UNSIGNED_OFFSET   ((__force fmode_t)0x2000)
                                          /*文件比较大，f_pos 解释为无符号整数*/
#define   FMODE_PATH          ((__force fmode_t)0x4000)       /*O_PATH 打开，文件不能操作*/
#define   FMODE_ATOMIC_POS     ((__force fmode_t)0x8000)   /*需原子操作访问 f_pos */

#define   FMODE_WRITER      ((__force fmode_t)0x10000)    /* Write access to underlying fs */
#define   FMODE_CAN_READ      ((__force fmode_t)0x20000)   /*文件可读*/
#define   FMODE_CAN_WRITE    ((__force fmode_t)0x40000)   /*文件可写*/
#define   FMODE_NONOTIFY      ((__force fmode_t)0x4000000)      /*不产生事件通知（fanotify）*/
...
#define   __FMODE_EXEC            ((__force int) FMODE_EXEC)
#define   __FMODE_NONOTIFY      ((__force int) FMODE_NONOTIFY)
```

●**f_pos**：文件当前位置，它是相对于文件开始处的一个字节偏移量，对文件的读写操作从当前位置开始。读写文件操作中，可能会修改文件的当前位置，lseek()/llseek()系统调用用于重设文件当前位置。

●**private_data**：文件私有数据结构指针，例如在设备文件中指针驱动程序定义的数据结构实例。

●**f_mapping**：文件地址空间 address_space 结构体指针，通常在打开文件的操作中指向 inode 实例内嵌的 address_space 实例成员。具有外部存储介质的文件，address_space 用于缓存文件内容，实现外部存储介质与页缓存的数据同步，详见第 11 章。

7.6.3 复制进程 VFS 信息

内核在创建子进程时，在复制进程的 copy_process()函数内将调用 copy_files()和 copy_fs()函数，复制父进程的打开文件和目录信息至子进程。

1 复制文件信息

复制父进程文件信息的函数 copy_files()定义在/kernel/fork.c 文件内，代码如下：

```
static int copy_files(unsigned long clone_flags, struct task_struct *tsk)
/*clone_flags：复制进程标记，tsk：子进程结构体指针*/
{
    struct files_struct *oldf, *newf;
    int error = 0;
```

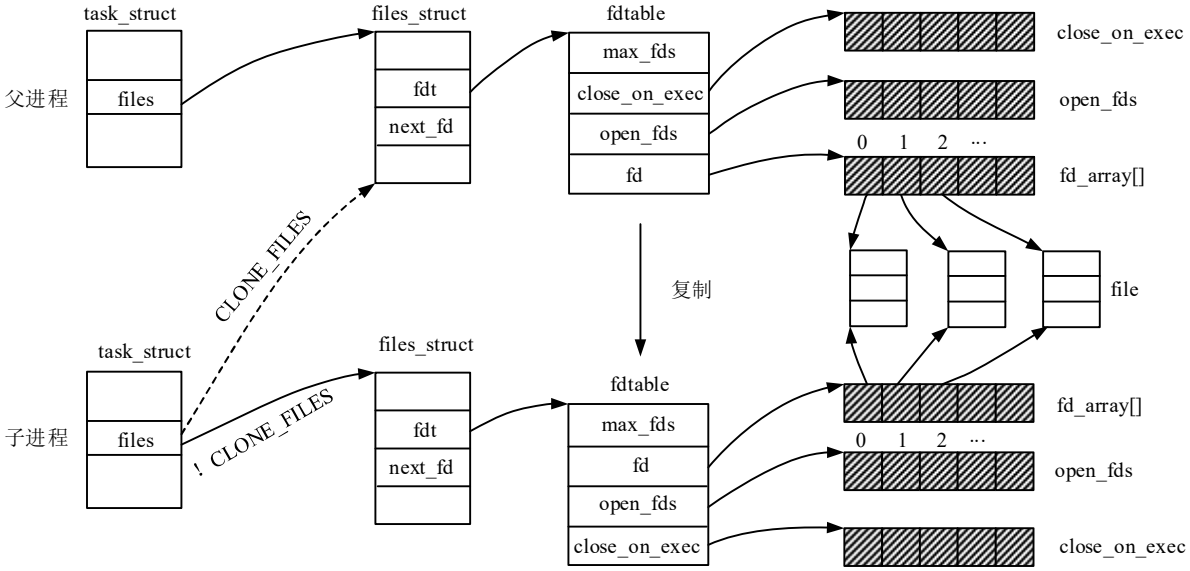
```
    oldf = current->files;     /*父进程 files_struct 实例指针*/
    if (!oldf)
        goto out;

    if (clone_flags & CLONE_FILES) {     /*如果设置了 CLONE_FILES，只需要增加 oldf 引用计数*/
        atomic_inc(&oldf->count);          /*共用 files_struct 实例，子进程与父进程共享打开文件*/
        goto out;
    }
    newf = dup_fd(oldf, &error);   /*没有设置 CLONE_FILES 标记，/fs/file.c*/
                                   /*为子进程创建 files_struct 结构体实例，并复制父进程实例数据*/
    if (!newf)
        goto out;

    tsk->files = newf;   /*新 files_struct 实例赋予新（子）进程*/
    error = 0;
out:
    return error;
}
```

复制进程文件信息的函数首先获取父进程（当前进程）files_struct 实例，如果标记参数设置了 CLONE_FILES 标记位，则表示父子进程使用相同的 files_struct 实例，只需要增加 files_struct 实例的引用计数值即可。

如果没有设置 CLONE_FILES 标记位，则需要调用 dup_fd()函数为子进程创建 files_struct 实例并复制实例数据，父进程 files_struct 实例管理的 fdtable 实例（含位图和 file 指针数组）也需要复制一个副本给子进程，同时增加打开文件 file 实例的引用计数，父子进程共用 file 实例。

这里需要注意共用 file 实例，意味着一个进程对文件当前位置的修改对另一个进程可见，访问模式的修改也是一样的。如果父子进程共用 files_struct 实例，则文件的打开、关闭等也是相互可见的。

dup_fd()函数定义在/fs/file.c 文件内，源代码请读者自行阅读，下图示意了 copy_files()函数执行的结果。



另外，dup()/dup2()/dup3()系统调用用于复制文件描述符，复制的描述符与原描述符共用 file 实例，也就是两个描述符指向同一个文件，系统调用实现在/fs/file.c 文件内，请读者自行阅读源代码。

2 复制目录信息

复制父进程目录信息的函数 copy_fs()定义在/kernel/fork.c 文件内，代码如下：

```
static int copy_fs(unsigned long clone_flags, struct task_struct *tsk)
/*clone_flags：复制进程标记，tsk：子进程结构体指针*/
{
    struct fs_struct *fs = current->fs;        /*父进程 fs_struct 实例指针*/
    if (clone_flags & CLONE_FS) {        /*如果设置了 CLONE_FS 标记位*/
        spin_lock(&fs->lock);
        if (fs->in_exec) {
            spin_unlock(&fs->lock);
            return -EAGAIN;
        }
        fs->users++;        /*增加实例用户数量*/
        spin_unlock(&fs->lock);
        return 0;
    }
    tsk->fs = copy_fs_struct(fs);
                /*没有设置 CLONE_FS 标记位，则创建、复制 fs_struct 实例，/fs/fs_struct.c*/
    if (!tsk->fs)
        return -ENOMEM;
    return 0;
}
```

复制进程目录信息的函数比较简单，如果标记参数设置了 CLONE_FS 标记位，则只需增加 fs_struct 实例的用户数即可，即父子进程共用 fs_struct 实例。

如果没有设置 CLONE_FS 标记位，则调用 copy_fs_struct(fs)函数，为子进程创建 fs_struct 实例并复制父进程实例的数据，父子进程具有相同的根目录和当前工作目录。

copy_fs_struct()定义在/fs/fs_struct.c 文件内：

```
struct fs_struct *copy_fs_struct(struct fs_struct *old)
{
    struct fs_struct *fs = kmem_cache_alloc(fs_cachep, GFP_KERNEL);     /*分配 fs_struct 实例*/
    if (fs) {                /*初始化实例*/
        fs->users = 1;
        fs->in_exec = 0;
        spin_lock_init(&fs->lock);
        seqcount_init(&fs->seq);
        fs->umask = old->umask;     /*复制父进程实例数据*/

        spin_lock(&old->lock);
        fs->root = old->root;     /*父子进程具有相同的根目录和当前工作目录*/
        path_get(&fs->root);       /*增加 dentry 和 vfsmount 实例引用计数*/
        fs->pwd = old->pwd;
        path_get(&fs->pwd);
        spin_unlock(&old->lock);
```

```
    }
    return fs;
}
```

7.7 打开/关闭文件

打开文件是进程对文件进行操作的基础。打开文件操作首先沿着文件路径在根文件系统中查找每个路径分量（目录项）对应的 dentry 和 inode 实例，如果路径分量尚未导入根文件系统，则需要创建 dentry 实例，从具体文件系统中查找目录项信息填充至 dentry 实例，并创建对应的 inode 实例，然后建立进程文件 file 实例与最后路径分量（文件）inode 实例之间的关联，将 inode->i_fop 表示 file_operation 实例指针赋予 file->f_op 成员等，最后调用 file->f_op->open()函数，完成特定于文件的打开操作。

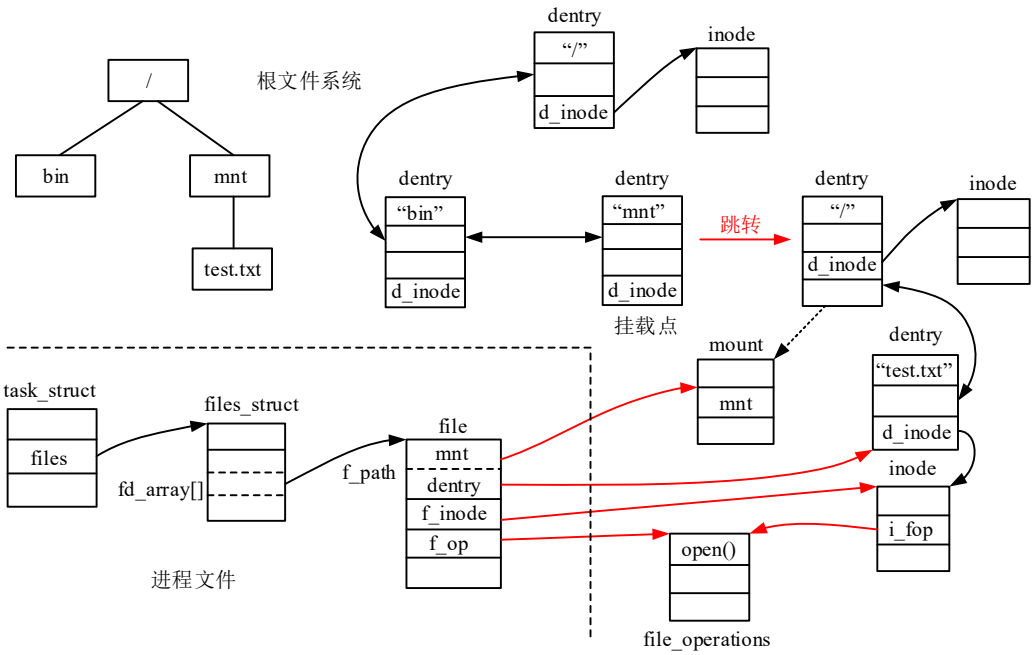
在进程退出或关闭文件时，需要断开进程与文件之间的关联，本节介绍内核打开和关闭文件操作。

7.7.1 open()

用户进程通过 open()系统调用打开文件，系统调用参数需要提供打开文件路径名称。假设，进程要打开/mnt/test.txt 文件（绝对路径），系统调用内将“/mnt/test.txt”路径名以‘/’字符为分隔符进行划分，共划分为三个路径分量（目录项）分别是“/”，“mnt”，“test.txt”，第一个“/”字符表示进程根目录，是查找路径的起点。

此时，open()系统调用中从进程根目录项下开始依次查找“mnt”和“test.txt”路径分量对应的 dentry 实例，如果找到分量对应的 dentry 实例，则在其子链表中查找下一分量的 dentry 实例。如果没找到则需要创建目录项 dentry 实例，进入到具体文件系统中去查找目录项信息，填充 dentry 实例，并为其创建对应的 inode 实例。如果在具体文件系统中也没有找到对应的目录项，说明当前路径分量不存在，系统调用失败。

进程文件 file 实例将与最后路径分量（文件目录项）对应的 dentry 和 inode 实例建立关联。下图示意了打开/mnt/test.txt 文件的执行结果，图中假设某个外部介质文件系统挂载到了/mnt 目录下，test.txt 是外部介质文件系统根目录下的文件。



打开文件操作看似简单，实际上非常复杂。因为需要解析用户输入的特殊的目录项，如：“.”表示当前目录，“..”表示当前目录项的父目录项。还需要考虑目录项是否是符号链接，如果是则需要执行跟踪。

另外，如果目录项是挂载点，还需要跳转到挂载文件系统根目录项下。

open()系统调用不仅可用于打开文件还可用于创建新文件，系统调用在/fs/open.c 文件内实现，代码如下：

```
SYSCALL_DEFINE3(open, const char __user *, filename, int, flags, umode_t, mode)
{
    if (force_o_largefile())      /*返回（BITS_PER_LONG != 32），/include/linux/fcntl.h*/
        flags |= O_LARGEFILE;         /*64 位系统默认设置 O_LARGEFILE 标记*/

    return do_sys_open(AT_FDCWD, filename, flags, mode);
                                        /*默认从当前工作目录开始查找，/fs/open.c*/
}
```

open()系统调用包含三个参数，如下所示：

●**filename**：打开文件的路径名称字符串，如：“/mnt/test.txt”。

●**flags**：用户传递的打开标记参数，对于内核来说这是外部标记，其取值定义如下：

```
/*/include/uapi/asm-generic/fcntl.h*/
#define    O_ACCMODE      00000003
#define    O_RDONLY       00000000      /*只读*/
#define    O_WRONLY       00000001      /*只写*/
#define    O_RDWR         00000002      /*读写*/
...
#ifndef    O_LARGEFILE             /*支持大文件*/
  #define O_LARGEFILE    00100000
#endif

#ifndef    O_DIRECTORY
  #define    O_DIRECTORY 00200000      /*如果不是打开目录，则失败（打开目录）*/
#endif

#ifndef O_NOFOLLOW
  #define O_NOFOLLOW    00400000      /*不跟踪符号链接*/
#endif

#ifndef O_NOATIME
  #define O_NOATIME01000000
#endif

#ifndef O_CLOEXEC
  #define O_CLOEXEC      02000000      /*设置 close_on_exec 中标记位，在 open()中传递无效？ */
#endif

...
#ifndef    O_PATH
```

```
  #define   O_PATH          010000000   /*只获取文件描述符，没有真正打开文件，file->f_op 为空*/
#endif
...
/*/arch/mips/include/uapi/asm/fcntl.h*/
#define   O_APPEND        0x0008    /*写入数据时，追加在文件的末尾*/
#define   O_DSYNC         0x0010    /*写文件时立即同步，只同步文件内容，不同步 inode 元数据*/
#define   O_NONBLOCK      0x0080    /*非阻塞方式打开*/
#define   O_CREAT         0x0100    /*若文件不存在，则创建新文件*/
#define   O_TRUNC         0x0200    /*文件长度设置为零，丢弃已有的内容*/
#define   O_EXCL          0x0400    /*与 O_CREAT 一起使用，确保由调用者创建文件*/
#define   O_NOCTTY        0x0800    /*不要让 filename 成为控制终端*/
#define   FASYNC          0x1000    /* fcntl, for BSD compatibility */
#define   O_LARGEFILE     0x2000    /*支持大文件，用于 64 位系统*/
#define   __O_SYNC        0x4000
#define   O_SYNC      (__O_SYNC|O_DSYNC)   /*写文件时立即同步，同步文件内容和 inode 数据*/
#define   O_DIRECT    0x8000    /*直接读写操作，不经过文件页缓存*/
```

●**mode**：新创建文件的模式参数。当 flags 参数设置了 O_CREAT 标记时，表示文件不存在时创建文件，mode 参数表示新创建文件的访问权限，它与进程所属 fs_struct 实例 umask 成员共同确定创建新文件的访问权限，umask 表示需要屏蔽的权限。

mode 参数值将传递给新创建文件 inode 实例 i_mode 成员，参数取值请参考本章前文 inode 结构体 i_mode 成员的介绍。

open()系统调用内调用 **do_sys_open**(int dfd, const char __user *filename, int flags, umode_t mode)函数完成文件的打开操作。

do_sys_open()函数会把 open()系统调用传递的参数转为内部表示：

●**dfd**：用于控制打开操作的执行，取值定义在/include/uapi/linux/fcntl.h 头文件：

```
#define   AT_FDCWD          -100     /*默认参数值，相对路径起点为进程当前工作目录*/
#define   AT_SYMLINK_NOFOLLOW   0x100    /*不要跟踪符号链接*/
#define   AT_REMOVEDIR          0x200    /*Remove directory instead of unlinking file. */
#define   AT_SYMLINK_FOLLOW     0x400    /*跟踪符号链接*/
#define   AT_NO_AUTOMOUNT       0x800    /* Suppress terminal automount traversal */
#define   AT_EMPTY_PATH         0x1000   /* Allow empty relative pathname */
```

●**filename**：系统调用传递的打开文件路径名称，在 do_sys_open()函数内转换成由 filename 结构体表示。filename 结构体定义在/include/linux/fs.h 头文件：

```
struct filename {
    const char    *name;          /*指向内核空间文件路径名称字符串，从用户空间复制而来*/
    const __user char    *uptr;   /*指向用户空间原始路径名称字符串*/
    struct audit_names  *aname;    /*/kernel/audit.h*/
    int    refcnt;
    const char    iname[];
};
```

●**flags**：系统调用传递的标记参数，在 do_sys_open()函数内转换成 open_flags 结构体实列。open_flags 结构体定义在/fs/internel.h 头文件：

```
struct open_flags {
    int    open_flag;        /*打开文件标记，来自系统调用 flags 参数*/
    umode_t    mode;        /*文件模式（访问权限），来自系统调用 mode 参数*/
    int    acc_mode;
    int    intent;
    int    lookup_flags;     /*查找目录项标记，由 flags 参数确定，/include/linux/namei.h*/
};
```

●**mode**：系统调用传递的模式参数。

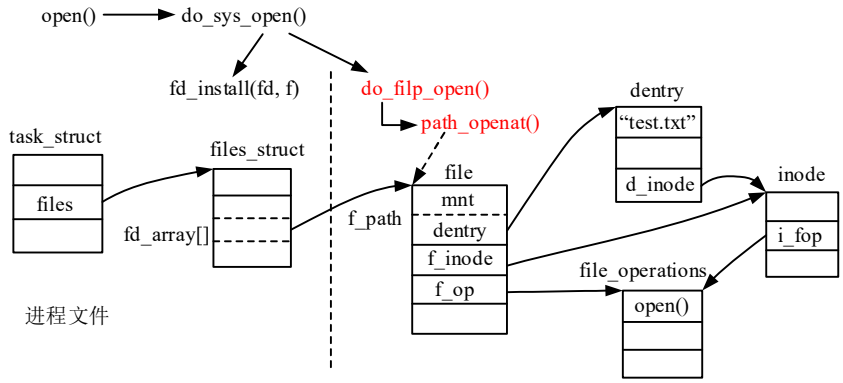
do_sys_open()函数在/fs/open.c 文件内实现，代码如下：

```
long do_sys_open(int dfd, const char __user *filename, int flags, umode_t mode)
/*dfd：AT_FDCWD，表示相对路径从进程当前工作目录开始查找*/
{
    struct open_flags op;
    int fd = build_open_flags(flags, mode, &op);
                    /*由 flags、mode 参数设置 open_flags 实例各成员，成功返回 0，/fs/open.c*/
    struct filename *tmp;
    ...  /*错误处理*/
    tmp = getname(filename);     /*文件路径名称转换成 filename 结构体实例，/fs/namei.c*/

    ...  /*错误处理*/

    fd = get_unused_fd_flags(flags);          /*获取进程最小未使用的文件描述符，见上一节*/
    if (fd >= 0) {          /*获取了文件描述符*/
        struct file *f = do_filp_open(dfd, tmp, &op);
                    /*创建 file 实例，查找各路径分量 dentry 实例，关联 inode 实例等，/fs/namei.c*/
        if (IS_ERR(f)) {
            ...
        } else {
            fsnotify_open(f);     /*向用户空间发送通知，/include/linux/fsnotify.h*/
            fd_install(fd, f);      /*将 file 实例与当前进程 files_struct 实例关联，/fs/file.c*/
        }
    }
    putname(tmp);
    return    fd;     /*返回文件描述符*/
}
```

do_sys_open()函数首先将系统调用 flags、mode 参数转换成 open_flags 实例，将文件路径名称参数转换成 filename 实例，这两个转换而得的实例将作为后面调用 do_filp_open()函数的参数；调用函数 get_unused_fd_flags()获取进程最小未使用文件描述符；调用 **do_filp_open()**函数创建文件 file 实例，依次搜索/创建各路径分量（目录项）对应的 dentry 和 inode 实例，将最后分量 dentry、inode 及 file_operations 实

例与 file 实例建立关联，调用 **file->f_op->open()**函数；最后调用 fd_install()函数建立 file 实例与进程 files_struct 实例之间的关联，返回文件描述符。

do_sys_open()函数执行结果如下图所示：



在以上函数中，打开文件的主要工作由 do_filp_open()函数完成，其中最艰巨的任务就是搜索/创建各路径分量的 dentry 和 inode 实例，下面将重点介绍此函数的实现，其它函数请读者自行阅读源代码。

另外，创建文件的 creat()系统调用是 open()系统调用实现函数的包装器，专门用于创建文件，系统调用实现代码如下（/fs/open.c）：

```
SYSCALL_DEFINE2(creat, const char __user *, pathname, umode_t, mode)
{
    return sys_open(pathname, O_CREAT | O_WRONLY | O_TRUNC, mode);
}
```

7.7.2 路径到节点

do_filp_open()函数要完成打开文件操作中最重要、最繁重的工作，函数内需要创建文件 file 实例，遍历文件路径中每个分量，在根文件系统中搜索/创建对应的 dentry 和 inode 结构体实例，当到达最末尾分量时（文件名称），将其 dentry 实例和 inode 实例与 file 实例建立关联。因此，do_filp_open()函数执行的主要工作可概括为从路径到节点，即由文件路径搜索到文件 inode 实例，并与 file 实例建立关联。

do_filp_open()函数需要通过 filename 和 open_flags 结构体实例传递文件路径和标记等参数。下面再次列出 open_flags 结构体的定义（/fs/internel.h）：

```
struct open_flags {
    int    open_flag;      /*打开文件标记，来自系统调用 flags 参数*/
    umode_t    mode;       /*文件模式（访问权限），来自系统调用 mode 参数*/
    int    acc_mode;
    int    intent;
    int    lookup_flags;   /*目录项搜索标记，由 flags 参数确定，/include/linux/namei.h*/
};
```

open_flags 结构体中 lookup_flags 成员表示搜索目录项（路径分量）标记，取值定义如下：

```
#define   LOOKUP_FOLLOW          0x0001      /*跟踪末尾符号链接*/
#define   LOOKUP_DIRECTORY       0x0002      /*查找目录项*/
#define   LOOKUP_AUTOMOUNT       0x0004      /*路径末尾可以是“/”字符*/

#define   LOOKUP_PARENT          0x0010      /*还有更多的分量*/
#define   LOOKUP_REVAL           0x0020
```

```
                                    /*根文件系统中的 dentry 缓存实例不可靠，要进入具体文件系统查找*/
#define   LOOKUP_RCU              0x0040

#define   LOOKUP_OPEN             0x0100
#define   LOOKUP_CREATE           0x0200
#define   LOOKUP_EXCL             0x0400
#define   LOOKUP_RENAME_TARGET    0x0800
#define   LOOKUP_JUMPED           0x1000
#define   LOOKUP_ROOT             0x2000    /*从 nameidata.root 指定目录项开始搜索*/
#define   LOOKUP_EMPTY            0x4000
```

在 do_filp_open()函数中通过 nameidata 结构体暂存路径分量搜索结果，结构体定义如下（/fs/namei.c）：

```
struct nameidata {
    struct   path   path;       /*当前搜索目录项的父目录项，即在此目录项下搜索下一个路径分量*/
    struct   qstr   last;       /*当前搜索目录项名称*/
    struct   path   root;       /*搜索起始点目录信息（可指定查找起点）*/
    struct   inode   *inode;     /*path 成员中目录项关联的 inode 实例*/
    unsigned int     flags;      /*搜索标记*/
    unsigned   seq,   m_seq;
    int     last_type;          /*当前查找路径分量类型，/include/linux/namei.h*/
    unsigned   depth;           /*符号链接嵌套深度*/
    int     total_link_count;   /*符号链接嵌套计数*/
    struct saved {              /*保存符号链接后面的路径信息*/
        struct path   link;     /*符号链接对应的目录项信息，用于获取符号链接内容*/
        void *cookie;
        const char *name;       /*符号链接后剩余路径名称字符串*/
        struct inode *inode;    /*符号链接目录项关联的 inode 实例*/
        unsigned seq;
    } *stack, internal[EMBEDDED_LEVELS];     /*EMBEDDED_LEVELS=2*/
    struct filename      *name;     /*指向 filename 实例，保存文件路径*/
    struct nameidata   *saved;
    unsigned       root_seq;
    int         dfd;            /*文件描述符*/
};
#define   EMBEDDED_LEVELS   2
```

nameidata 结构体主要成员简介如下：

●**path、inode**：保存查找过程中的中间结果，查找前为父目录项信息，查找后为当前查找目录项信息，也就是下一个查找目录项的父目录项。

●**internal[]**：saved 结构体数组，保存符号链接分量信息，以及返回上一层的路径，数组项数为 2。

●**stack**：指向 saved 结构体数组，初始指向 internal[]成员，如果符号链接嵌套深度大于 2，则重新创建 saved 结构体数组。

●**root**：path 结构体，表示查找起点目录信息。

●**flags**：目录项查找标记，同 open_flags 结构体 lookup_flags 成员取值。
●**last**：当前查找目录项名称信息，含字符串指针和散列值。
●**last_type**：当前查找目录项类型。目录项类型值定义在/include/linux/namei.h 头文件：

```
enum {LAST_NORM, LAST_ROOT, LAST_DOT, LAST_DOTDOT, LAST_BIND};
                        /*依次为：普通分量、根目录项、当前目录项、父目录项、绑定目录项*/
```

set_nameidata()函数根据 dfd 和 filename 结构体参数设置 nameidata 实例，函数定义如下：

```
static void set_nameidata(struct nameidata *p, int dfd, struct filename *name)
{
    struct nameidata *old = current->nameidata;     /*当前进程 nameidata 实例*/
    p->stack = p->internal;         /*指向内嵌实例*/
    p->dfd = dfd;
    p->name = name;             /*指向 filename 实例*/
    p->total_link_count = old ? old->total_link_count : 0;
    p->saved = old;
    current->nameidata = p;
}
```

path_to_nameidata(const struct path *path,struct nameidata *nd)函数将 path 信息设置到 nd.path 成员，用于设置下一次查找的父目录项。

现在来看 do_filp_open()函数的实现，代码如下（/fs/namei.c）：

```
struct file *do_filp_open(int dfd, struct filename *pathname,const struct open_flags *op)
{
    struct nameidata nd;                /*nameidata 实例，暂存搜索结果*/
    int flags = op->lookup_flags;       /*查找标记，来自 open_flags 实例*/
    struct file *filp;

    set_nameidata(&nd, dfd, pathname);      /*设置 nameidata 实例，/fs/namei.c*/
    filp = path_openat(&nd, op, flags | LOOKUP_RCU);
            /*创建 file 实例，依次查找各路径分量，默认设置 LOOKUP_RCU 标记，/fs/namei.c*/
    if (unlikely(filp == ERR_PTR(-ECHILD)))
        filp = path_openat(&nd, op, flags);
    if (unlikely(filp == ERR_PTR(-ESTALE)))
        filp = path_openat(&nd, op, flags | LOOKUP_REVAL);
    restore_nameidata();        /*释放处理符号链接中分配的 save[]数组，见下文*/
    return filp;            /*返回 file 实例指针*/
}
```

do_filp_open()函数内定义了 nameidata 结构体实例 nd，用于暂存目录项查找中间结果，首先调用函数 set_nameidata()设置 nd 实例，然后将主要工作交给 **path_openat**(&nd, op, flags | LOOKUP_RCU)函数完成。

path_openat()函数首先为打开文件创建 file 实例。然后，沿着文件路径，在根文件系统中依次查找（或创建）各路径分量对应的 dentry 和 inode 结构体实例。如果实例尚不存在，则进入到具体文件系统中查找，

若找到对应的目录项，则为其创建并填充 dentry 和 inode 实例。如果没有找到，说明目录项不存在，查找失败，函数返回错误码。最后建立 file 与最末尾路径分量 dentry 和 inode 实例之间的关联，调用 file->f_op->open()函数等，完成文件的打开操作。

1 搜索函数

path_openat()函数定义在/fs/namei.c 文件内，代码如下：

```
static struct file *path_openat(struct nameidata *nd,const struct open_flags *op, unsigned flags)
/*flags：查找标记，op->lookup_flags|LOOKUP_RCU*/
{
    const char *s;
    struct file *file;
    int opened = 0;
    int error;

    file = get_empty_filp();      /*从 slab 缓存中分配 file 实例并初始化（引用计数置 1），/fs/file_table.c*/
    ...    /*错误处理*/

    file->f_flags = op->open_flag;      /*open()系统调用中 flags 参数*/

    if (unlikely(file->f_flags & __O_TMPFILE)) {          /*/include/uapi/asm-generic/fcntl.h*/
        error = do_tmpfile(nd, flags, op, file, &opened);    /*/fs/namei.c*/
        goto out2;
    }

    s = path_init(nd, flags);      /*确定查找起点，s 指向路径名称字符串，/fs/namei.c*/
    ...     /*错误处理*/

    /*遍历每个路径分量，查找（创建）dentry 和 inode 实例*/
    while (!(error = link_path_walk(s, nd)) &&(error = do_last(nd, file, op, &opened)) > 0) {
        nd->flags &= ~(LOOKUP_OPEN|LOOKUP_CREATE|LOOKUP_EXCL);
        s = trailing_symlink(nd);      /*最末尾分量为符号链接，获取路径，继续搜索，/fs/namei.c*/
        ...          /*出错，跳出循环*/
    }    /*遍历路径分量结束，如果最末尾路径分量为符号链接，重新开启循环*/

    terminate_walk(nd);          /*结束搜索*/
out2:
    if (!(opened & FILE_OPENED)) {    /*如果打开操作失败*/
        ...
    }
    if (unlikely(error)) {
        ...
    }
    return file;        /*成功返回 file 实例指针*/
```

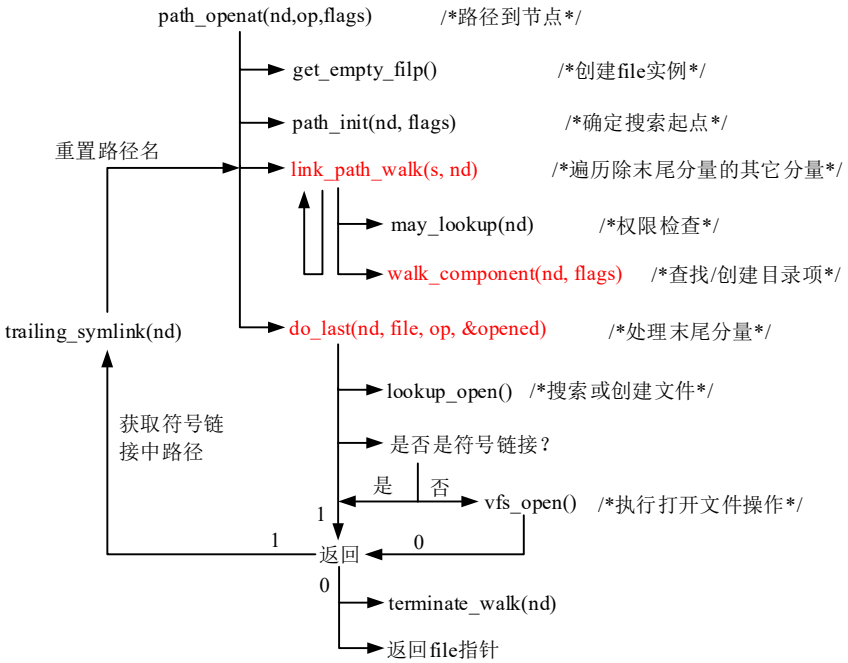
}

由于符号链接的存在，增加了搜索操作的复杂性。前面我们讲过，链接分为硬链接和符号链接。对于硬链接，它实际上就是普通正常的文件，只过目录项与其它某些目录项关联到同一个节点，相当于一个文件具有多个名称（路径），硬链接的打开与普通文件相同，无需做特殊处理。

符号链接与硬链接不同，符号链接其实也是普通的文件，只不过其文件内容是另一个文件的路径名称。当搜索路径到达符号链接时，需要读取文件内容，获取路径名称，进入此路径搜索，搜索到末尾分量后（符号链接中路径搜索完）返回原搜索路径继续搜索。符号链接还可以嵌套，即符号链接中包含的路径中还可以有符号链接。

在 path_openat()函数中，link_path_walk()函数需要处理原始路径中除最末尾分量外的所有其它分量，包括符号链接。do_last()函数处理原始路径中最末尾分量，如果最末尾分量是符号链接，则返回 1，随后将调用 trailing_symlink()函数读取末尾符号链接的内容（路径名），重复上面的搜索。也就是说最末尾分量是符号链接的是话，就将其内容当成一个新的路径，重新开启搜索。

path_openat()函数调用关系如下图所示：



以上调用各函数的功能简介如下：

●**get_empty_filp()**：从 slab 缓存中分配 file 实例，函数代码请读者自行阅读。

●**path_init()**：确定查找起点，例如，如果是绝对路径（如：/a/b）则起点为进程根目录，如果是相对路径（如：./c）则起点为进程当前工作目录。

●**link_path_walk()**：搜索除最末尾分量外的路径分量，包括中间的符号链接分量，以及嵌套的符号链接。此函数内遍历每个路径分量，对每个分量先调用 may_lookup(path)函数检查访问权限，再调用函数 walk_component()查找或创建目录项分量 dentry 和 inode 实例。如果分量是符号链接，则进入符号链接中路径搜索，到达符号链接路径末尾分量后返回符号链接分量在原始路径中的下一个分量继续搜索。如果存在符号链接嵌套，则逐层进入符号链接路径，搜索完后逐层返回。

link_path_walk()函数遍历到原始路径最末尾分量时，返回 0，不进行处理，由 do_last()函数处理最末尾分量。

●**do_last()**：处理搜索路径中最末尾分量，查找（或创建）其 dentry 和 inode 实例，建立 file 实例与此 inode 实例之间的关联，设置 file 实例，调用 file_operations 实例中的 open()函数等。如果需要创建新文件，则创建新文件，函数成功返回 0，失败返回错误码。

另外，如果最末尾分量是符号链接，且需要跟踪符号链接，则函数返回 1，不进行关联 file 实例及以后的操作。

●**trailing_symlink(nd)**：若 do_last()函数返回 1，表示原始路径中最末尾分量是符号链接，则继续调用 trailing_symlink(nd)函数读取符号链接文件内容中的路径名。把末尾符号链接中路径当成原始路径，重新开启路径搜索（前面的原始路径可以抛弃了）。

下面分别介绍以上 path_init()、link_path_walk()和 do_last()函数的实现，其中包括符号链接的处理。

2 确定搜索起点

path_init(nd, flags)函数定义在/fs/namei.c 文件内，通过参数传递的 nd 实例和 flags 标记值确定路径搜索的起点，函数代码如下：

```
static const char *path_init(struct nameidata *nd, unsigned flags)
/*flags：查找标记，op->lookup_flags|LOOKUP_RCU*/
{
    int retval = 0;
    const char *s = nd->name->name;          /*路径名称字符串*/

    nd->last_type = LAST_ROOT;                  /*初始化下一查找分量为根目录项*/
    nd->flags = flags | LOOKUP_JUMPED | LOOKUP_PARENT;      /*设置查找标记*/
    nd->depth = 0;                  /*depth 初始化为 0*/
    nd->total_link_count = 0;
     /*如果设置了 LOOKUP_ROOT 标记，从 nd->root 指定目录项开始查找*/
    if (flags & LOOKUP_ROOT) {     /*open()系统调用中不设置此位，内核调用时可能设置*/
        struct dentry *root = nd->root.dentry;
        struct inode *inode = root->d_inode;
        if (*s) {
            if (!d_can_lookup(root))       /*是否是普通目录项*/
                return ERR_PTR(-ENOTDIR);
            retval = inode_permission(inode, MAY_EXEC);       /*访问权限检查，/fs/namei.c*/
            ...     /*错误处理*/
        }
        nd->path = nd->root;       /*设置搜索起点目录项*/
        nd->inode = inode;
        if (flags & LOOKUP_RCU) {
            rcu_read_lock();
            nd->seq = __read_seqcount_begin(&nd->path.dentry->d_seq);
            nd->root_seq = nd->seq;
            nd->m_seq = read_seqbegin(&mount_lock);
        } else {
            path_get(&nd->path);
        }
        return s;        /*函数返回*/
    }
```

```
/*没有设置 LOOKUP_ROOT 标记的情形，适用于 open()系统调用*/
nd->root.mnt = NULL;
nd->m_seq = read_seqbegin(&mount_lock);
if (*s == '/') {              /*路径名称第一个字符为‘/’，表示从进程根目录开始查找*/
    if (flags & LOOKUP_RCU) {       /*open()系统调用默认设置 LOOKUP_RCU 标记*/
        rcu_read_lock();
        set_root_rcu(nd);       /*nd->root 设为进程根目录项（查找起点）*/
        nd->seq = nd->root_seq;
    } else {
        set_root(nd);
        path_get(&nd->root);
    }
    nd->path = nd->root;        /*进程根目录项设为搜索起点*/
} else if (nd->dfd == AT_FDCWD) {         /*如果第一个字符不是‘/’，且 dfd == AT_FDCWD*/
    if (flags & LOOKUP_RCU) {         /*从进程当前工作目录开始查找*/
        struct fs_struct *fs = current->fs;
        unsigned seq;

        rcu_read_lock();
        do {
            seq = read_seqcount_begin(&fs->seq);
            nd->path = fs->pwd;                 /*从当前工作目录开始查找*/
            nd->seq = __read_seqcount_begin(&nd->path.dentry->d_seq);
        } while (read_seqcount_retry(&fs->seq, seq));
    } else {
        get_fs_pwd(current->fs, &nd->path);     /*当前工作目录，/include/linux/fs_struct.h*/
    }

} else {            /*如果第一个字符不是‘/’，且 dfd != AT_FDCWD*/
    struct fd f = fdget_raw(nd->dfd);
    struct dentry *dentry;

    ...     /*错误处理*/
    dentry = f.file->f_path.dentry;     /*nd->dfd 描述符表示文件所在目录*/

    if (*s) {
        if (!d_can_lookup(dentry)) {
            fdput(f);
            return ERR_PTR(-ENOTDIR);
        }
    }

    nd->path = f.file->f_path;        /*文件所在目录，设为查找起点*/
```

```
            if (flags & LOOKUP_RCU) {
                rcu_read_lock();
                nd->inode = nd->path.dentry->d_inode;
                nd->seq = read_seqcount_begin(&nd->path.dentry->d_seq);
            } else {
                path_get(&nd->path);
                nd->inode = nd->path.dentry->d_inode;
            }
            fdput(f);
            return s;       /*返回路径名称字符串指针*/
        }

        /*没有设置 LOOKUP_ROOT 标记，且第一个字符不是‘/’或 dfd == AT_FDCWD，继续执行*/
        nd->inode = nd->path.dentry->d_inode;
        if (!(flags & LOOKUP_RCU))
            return s;
        if (likely(!read_seqcount_retry(&nd->path.dentry->d_seq, nd->seq)))
            return s;
        if (!(nd->flags & LOOKUP_ROOT))
            nd->root.mnt = NULL;
        rcu_read_unlock();
        return ERR_PTR(-ECHILD);
    }
```

path_init()函数执行结果（查找起点目录项保存在 nd->path 成员）：

（1）若 flags 参数设置了 LOOKUP_ROOT 标记，则从 nd->root 指定目录开始查找；

（2）若没有设置 LOOKUP_ROOT 标记，且路径名称第一个字符为‘/’，则查找起点设为进程根目录；

（3）若没有设置 LOOKUP_ROOT 标记，路径名称第一个字符不是‘/’，且 dfd == AT_FDCWD，则查找起点设为进程当前工作目录；

（4）其它情况起点为 nd->dfd 文件描述符表示文件所在目录。

3 搜索非末尾分量

link_path_walk()函数用于搜索原始路径中除末尾分量外的其它分量。由于符号链接的存在，给搜索路径分量的操作增加了许多复杂度。下面先介绍符号链接的处理方式，然后再介绍 link_path_walk()函数的实现。

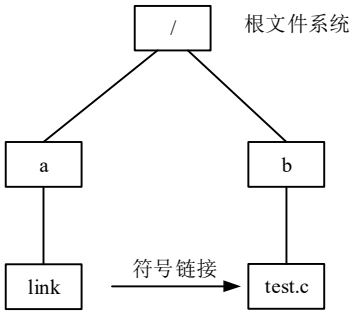
■处理符号链接

这里处理的符号链接是指非末尾分量的符号链接，末尾分量符号链接的处理后面再介绍。

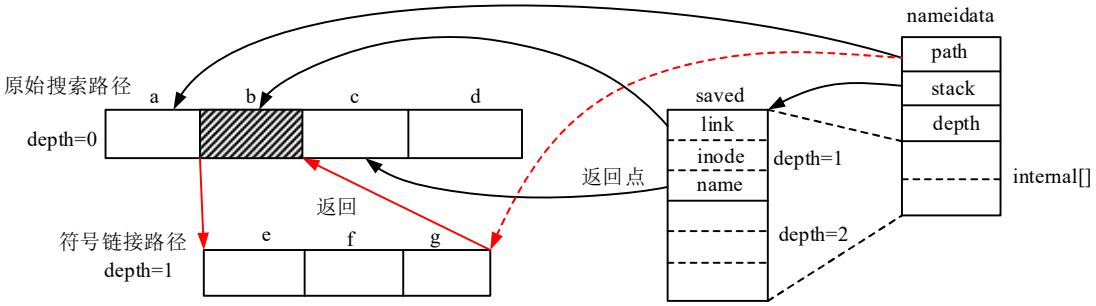
在讲解怎样处理符号链接前，我们先来简单了解一下符号链接。符号链接又称软链接，用户可用 ln -s 命令创建一个符号链接。符号链接实际上就是一个普通的文本文件，文件内容保存的是另一个文件的路径，不过这个路径是相对于符号链接的相对路径。

例如，如下图所示，/a/link 是到/b/test.c 文件的符号链接，/a/link 文件的内容应是 test.c 文件的相对路径名，文件内容为“../b/test.c”。

目标文件的内容以符号链接的父目录项为基准（此处为/a），因为创建符号链接时，是在父目录项下创建的，因此搜索目标文件时也以父目录项为基准。若目标文件与符号链接在同一目录下（相同父目录项），目标文件的路径将以“./”开始，表示在当前目录下查找目标文件。因此下图中，link 的内容应该是“../b/test.c”，表示在 a 的父目录项（此处为根目录）下查找 b/test.c 文件。



下面我们先简要了解一下 link_path_walk()函数如何处理符号链接，然后介绍函数代码的实现。如下图所示，假设原始搜索路径为/a/b/c/d，其中 b 为符号链接，符号链接内容为路径 e/f/g，那么实际的搜索路径应是/a/e/f/g/c/d，相当于用符号链接中的路径代替 b 目录项。符号链接路径相当于一个中断，中断原始路径的搜索，符号链接中路径搜索完后，要回到原始路径继续搜索。



nameidata 结构体中 path 成员表示当前搜索目录项的父目录项，即当前在那个目录项下搜索路径分量。例如，假设在 a 目录项下搜索 b 目录项，path 成员指向 a，b 为当前搜索目录项。

当搜索路径到达 b 目录项时，发现其为符号链接，进而设置 saved 结构体数组中第一个数组项（深度值 depth 加 1），用于保存当前符号链接 b 目录项信息以及 b 分量之后的原始路径名称。saved 结构体 link 成员指向 b 目录项 dentry 实例，inode 指向 b 目录项对应的 inode 实例，name 指向 c/d 字符串（返回路径）。link 保存的目录项信息用于读取符号链接的文件内容，即 e/f/g 路径名称。其实 saved 结构体好比保存中断上下文信息一样，用于搜索完符号链接中路径后返回原搜索路径。

注意，刚进入符号链接的搜索路径时，nameidata 实例中的 path 成员仍然指向 a，a 仍为父目录项，而不是 b，在 a 目录项下继续搜索符号链接路径 e/f/g，简单地说就是直接用 e/f/g 代替 b，在 a 下面搜索 e 目录项。

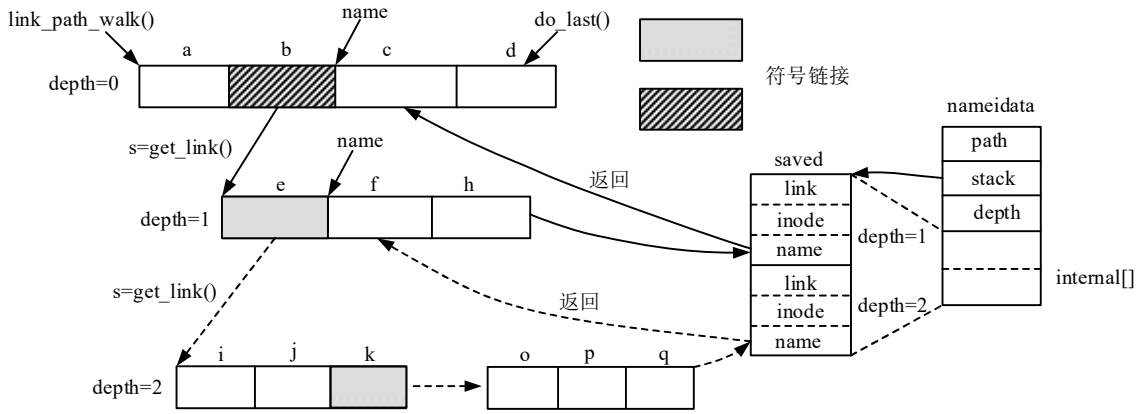
当搜索符号链接路径到达最后一个分量 g 时，nameidata 实例中的 path 成员此时指向 g，将以 g 为父目录项，释放 saved 实例（深度值 depth 减 1），即返回到原搜索路径继续搜索。在 g 目录项下继续搜索 saved.name 中指示剩余路径的第一个分量 c。

符号链接是可以嵌套的，即符号链接文件内容中的路径中还可以有符号链接。每增加一层符号链接，深度加 1，nameidata 实例 stack 指向 saved 结构体数组中使用项加 1，用于保存返回路径信息。

如下图所示，假设 e 分量也是一个符号链接，则搜索 e 时，深度加 1，saved 结构体中第 2 个数组项保存搜索完 e 保存的路径后的返回信息，即搜索完 e 指示的路径后返回搜索 f/h，深度减 1。更多层次的嵌套依次类推。

这里还有一个问题需要注意，那就是如果符号链接的末尾分量还是符号链接，将不需要增加嵌套深度。

如下图所示，假设 k 是末尾分量，也是符号链接，搜索到 k 时不需要增加深度。因为 k 是最末尾分量，k 处理完后，不需要返回 i/j/k 路径了，进入 k 包含的路径搜索完后直接返回上一层的 f/h 路径就可以了。



下面先介绍几个辅助函数，然后再介绍 link_path_walk()函数的实现。

●int **should_follow_link**(struct nameidata *nd, struct path *link,int follow,struct inode *inode, unsigned seq)：判断 link 表示的分量是否是符号链接（父目录项保存在 nd.path 中，inode 指向其 inode 实例），且需要跟踪，如果是则将 nd.stack 指向数组项深度值加 1，将 link 目录项信息等信息保存至 saved 实例中，函数返回 1，这相当于保存路径搜索的上下文信息。若不是符号链接，或不需要跟踪，函数返回 0。

●char ***get_link**(struct nameidata *nd)：读取当前深度符号链接中的文件内容，即获取符号链接中的路径名称字符串。saved.link 保存了符号链接的目录项，saved.inode 保存了其 inode 实例，依此可读取文件内容（调用 inode->i_op->follow_link()函数）。

●void **put_link**(struct nameidata *nd)：搜索完符号链接中的路径后调用此函数，将深度值减 1，调用函数 inode->i_op->put_link()等（如果需要的话）。

■非末尾分量搜索函数

下面可以来看 link_path_walk()函数的实现了。link_path_walk()函数内将路径名称字符串以“/”字符为分隔符，划分成各个路径分量。例如：“/mnt/test.txt”路径名，划分成“mnt”和“test.txt”分量，第一个“/”字符表示起点为进程根目录项，在 path_init()函数中已经处理过了。又如“./usr/test.txt”路径名，将划分成“usr”和“test.txt”分量，起点为进程当前工作目录。

link_path_walk()函数内是一个大的循环，逐个为各个路径分量搜索（或创建）dentry 和 inode 实例（除最末尾路径分量），另外还需要处理符号链接分量。

link_path_walk()函数定义在/fs/namei.c 文件内，代码如下：

```
static int link_path_walk(const char *name, struct nameidata *nd)
/*name：原始路径名称字符串指针，nd：nameidata 实例指针，用于暂存查找结果*/
{
    int err;

    while (*name=='/')      /*跳过开始处的‘/’字符*/
        name++;
    if (!*name)      /*路径名为空，返回 0*/
        return 0;

    /*循环体开始，遍历各个路径分量（目录项）*/
    for(;;) {
```

```
u64 hash_len;       /*保存目录项名称散列值和长度*/
int type;

err = may_lookup(nd);   /*调用 inode_permission()函数检查访问权限，/fs/namei.c*/
...   /*错误处理*/

hash_len = hash_name(name);
                    /*以‘/’字符为分隔符，计算此次路径分量长度和散列值，/fs/namei.c*/

type = LAST_NORM;         /*初始化查找路径分量为普通分量*/
if (name[0] == '.')               /*如果路径分量开始字符是‘.’*/
    switch (hashlen_len(hash_len)) {
        case 2:
            if (name[1] == '.') {        /*路径分量为“..”表示查找目录项为父目录项*/
                type = LAST_DOTDOT;
                nd->flags |= LOOKUP_JUMPED;        /*设置跳转标记位*/
            }
            break;
        case 1:          /*路径分量包含一个“.”表示查找目录项为当前目录项*/
            type = LAST_DOT;
    }
if (likely(type == LAST_NORM)) {        /*如果查找分量为普通目录项*/
    struct dentry *parent = nd->path.dentry;          /*父目录项*/
    nd->flags &= ~LOOKUP_JUMPED;       /*清除跳转标记位*/
    if (unlikely(parent->d_flags & DCACHE_OP_HASH)) {
                                        /*目录项操作结构中定义了计算散列值函数*/
        struct qstr this = { { .hash_len = hash_len }, .name = name };
        err = parent->d_op->d_hash(parent, &this);
        if (err < 0)
            return err;
        hash_len = this.hash_len;
        name = this.name;
    }    /*计算查找目录项名称字符串长度和散列值等信息*/
}
/*保存本次搜索分量信息至 nd 实例*/
nd->last.hash_len = hash_len;      /*本次查找目录项长度*/
nd->last.name = name;           /*本次查找路径分量名称*/
nd->last_type = type;           /*路径分量类型*/

name += hashlen_len(hash_len);
                            /*name 指向下一个查找路径分量，/include/linux/dcache.h*/
if (!*name)
    goto OK;
```

```
        /*name 为 NULL，表示本次查找的是最末尾分量，跳转到 OK 处*/

    /*本次查找的不是最末尾分量或者是末尾分量但后面还有‘/’字符*/
    do {
        name++;
    } while (unlikely(*name == '/'));     /*跳过下一分量前的若干个‘/’字符*/

    if (unlikely(!*name)) {         /*name 为 NULL，表示当前是末尾分量*/
OK:
        if (!nd->depth)             /*原始路径中的末尾分量（深度为 0），返回 0，不处理*/
            return 0;

        /*是末尾分量，但深度非 0，即符号链接中路径的末尾分量*/
        name = nd->stack[nd->depth - 1].name;       /*上一层符号链接分量之后的路径名称*/
        if (!name)              /*末尾符号链接路径中的末尾分量，返回 0，由 do_last()处理*/
            return 0;

        /*
        *处理非末尾分量符号链接路径中的末尾分量（上一层返回路径非空），
        *当前搜索分量是符号链接返回 1，否则返回 0。
        */
        err = walk_component(nd, WALK_GET | WALK_PUT);
    } else {        /*搜索非末尾分量*/
        err = walk_component(nd, WALK_GET);       /*搜索当前分量*/
    }

    if (err < 0)
        return err;

    /*当前搜索分量是符号链接 err 为 1，否则为 0，以下是为进入符号链接路径搜索做准备*/
    if (err) {
        const char *s = get_link(nd);         /*读取符号链接中路径名称*/

        ...         /*错误处理*/
        err = 0;
        if (unlikely(!s)) {
            put_link(nd);         /*为空深度减 1*/
        } else {            /*符号链接中路径非空*/
            nd->stack[nd->depth - 1].name = name;    /*处理完符号链接后返回上一层的路径*/
            name = s;       /*符号链接中的路径名称*/
            continue;       /*进入符号链接路径继续搜索*/
        }
    }
```

```
        /*如果当前分量不是可查找的目录项，返回错误码*/
        if (unlikely(!d_can_lookup(nd->path.dentry))) {
            if (nd->flags & LOOKUP_RCU) {
                if (unlazy_walk(nd, NULL, 0))
                    return -ECHILD;
            }
            return -ENOTDIR;
        }
    }   /*搜索 name 指向的下一个分量，无限循环体结束*/
}
```

link_path_walk()函数首先跳过路径名称字符串开始处的‘/’字符，然后以‘/’字符为分隔符依次获取每个路径分量，在循环体中逐个搜索路径分量。在处理路径分量时“.”表示当前目录（nd->path），“..”表示父目录（nd->path 的父目录）。对每个路径分量调用 **walk_component()**函数在根文件系统中查找对应的 dentry 和 inode 实例，若不存在则创建 dentry 实例，从具体文件系统中查找目录信息填充实例，并创建 inode 实例。查找到最末尾目录项时，将不进行处理，函数返回 0，末尾分量由 do_last()函数处理。

如果搜索到符号链接，会保存符号链接之后的路径名称至 saved.name 成员，然后进入符号链接中的路径继续搜索。当搜索到符号链接路径中的末尾分量时，会从 saved.name 中恢复符号链接分量之后的路径，回到原路径继续搜索。

如果存在符号链接嵌套，则逐层保存返回路径，进入符号链接中路径搜索，搜索完后，逐层返回。

●搜索单个分量

搜索单个路径分量的函数为 walk_component(struct nameidata *nd, int flags)，参数 nd 指向的 nameidata 实例中的 **last** 成员保存了本次搜索分量的名称信息，**last_type** 成员保存了分量类型，**path** 成员保存了父目录项信息。flags 参数表示查找标记，取值由/fs/namei.c 内枚举类型定义：

```
enum {WALK_GET = 1, WALK_PUT = 2};          /*表示可增加/减少深度*/
```

walk_component()函数在/fs/namei.c 文件内实现，代码如下：

```
static int walk_component(struct nameidata *nd, int flags)
/*flags：符号链接末尾分量会设置 WALK_GET|WALK_PUT，其它只设置 WALK_GET*/
{
    struct path path;
    struct inode *inode;
    unsigned seq;
    int err;

    if (unlikely(nd->last_type != LAST_NORM)) {     /*处理“.”或“..”目录项*/
        err = handle_dots(nd, nd->last_type);       /*/fs/namei.c*/
                    /*“.”目录项不需要处理，“..”目录项将当前目录切换到其父目录项*/
        if (flags & WALK_PUT)
            put_link(nd);
        return err;
    }
```

```
    err = lookup_fast(nd, &path, &inode, &seq);      /*在根文件系统中查找目录项，/fs/namei.c*/*/
        /*nd->path 表示父目录项（在此目录项下查找），nd->last 表示本次查找目录项名称信息，
        *path、inode 局部变量保存查找到目录项的信息。
        */
    if (unlikely(err)) {    /*在根文件系统中未找到，进入到具体文件系统中查找*/
        ...     /*错误处理*/

        err = lookup_slow(nd, &path);
                    /*在具体文件系统中查找目录项，path 保存查找结果，/fs/namei.c*/
        ...     /*错误处理*/

        inode = d_backing_inode(path.dentry);    /*dentry 关联的 inode 实例，/include/linux/dcache.h*/
        seq = 0;          /* we are already out of RCU mode */
        err = -ENOENT;
        if (d_is_negative(path.dentry))
            goto out_path_put;
    }

    if (flags & WALK_PUT)      /*非末尾分量符号链接路径中的末尾分量，深度减 1*/
        put_link(nd);
    err = should_follow_link(nd, &path, flags & WALK_GET, inode, seq);
                    /*判断当前分量是否是需要跟踪的符号链接，
                    *是则返回 1（深度加 1），否则返回 0，/fs/namei.c。
                    *如果非末尾分量符号链接路径中的末尾分量还是符号链接，深度不变。
                    */
    if (unlikely(err))          /*如果是符号链接，函数返回 1*/
        return err;

    /*当前分量不是符号链接*/
    path_to_nameidata(&path, nd);      /*/fs/namei.c*/
            /*查找分量成功，用当前分量设置 nd->path 成员，作为下次查找的父目录项*/
    nd->inode = inode;
    nd->seq = seq;
    return 0;          /*搜索成功返回 0*/
    ...
}
```

walk_component()函数对路径分量的处理流程如下：

（1）处理非普通路径分量。如果是“.”或“..”路径分量，则调用 handle_dots(nd, nd->last_type)函数进行处理，“.”分量不需要进行处理，函数返回。如果是“..”则调用 follow_dotdot(struct nameidata *nd)函数将 nd->path 成员设置为其父目录项，函数内需要处理当前目录项是根目录项和挂载点的情形。

（2）对普通路径分量，调用 lookup_fast()函数在根文件系统中查找对应的 dentry 实例。函数内调用前面介绍的__d_lookup()函数在全局散列表中查找对应 dentry 实例，如果找到则 lookup_fast()函数返回 0，表示不需要进入到具体文件系统中查找，否则返回 1。如果目录项是挂载点则调用 **lookup_mnt(path)**函数跳

转至最近（最后）一次挂载文件系统的根目录项。

（3）如果 lookup_fast()函数返回 1，则调用 lookup_slow(nd, &path)函数，创建 dentry 实例，调用文件系统类型定义的 **i_op->lookup()**函数，进入到具体文件系统，在当前查找目录项下查找本次查找路径分量信息，填充 denty 实例，并创建 inode 实例。

（4）如果是符号链接中的末尾分量，深度度减 1。

（5）调用 should_follow_link()函数判断当前分量是否是需要跟踪的符号链接，是则返回 1，并增加嵌套深度。不是符号链接或不需要跟踪，返回 0。

should_follow_link()函数返回 1 后，walk_component()函数也返回 1。link_path_walk()函数随后会将当前分量后面的路径，保存至 saved.name 成员，作为当前符号链接的返回路径，并读取本符号链接的内容，即路径名称（赋予 name 变量），进入此路径继续搜索，符号链接路径搜索完后（到达末尾分量），设置 name=saved.name，即回到符号链接之后的路径继续搜索。

（6）若 should_follow_link()函数返回 0，walk_component()函数将当前搜索的目录项信息赋予 nd->path 和 nd->inode 成员，作为下一个搜索路径分量的父目录项。walk_component()函数返回 0，link_path_walk()函数继续搜索下一分量。

以上只是简单地介绍了各函数的功能，具体源代码请读者自行阅读。

4 处理末尾分量

link_path_walk()函数在到达最末尾路径分量时，将路径分量名称信息保存在 nameidata 实例中，随即返回 0，并没有进行搜索。nameidata 实例中 last 成员保存了末尾分量的名称信息，last_type 成员保存了末尾分量类型，path 成员保存了最末尾分量的父目录项信息。

path_openat()函数随后调用 **do_last()**函数对末尾路径分量进行处理，末尾分量可能是普通目录项，也可能是符号符接。如果是符号链接在 do_last()函数中将调用 should_follow_link()函数使深度加 1，do_last()函数返回 1。随后，path_openat()函数调用 trailing_symlink()函数读取符号链接中的路径名称（返回路径设为 NULL），开启新的搜索（深度为 1）。末尾符号链接路径中的末尾分量也将由 do_last()函数处理。

下面先关注对普通末尾分量的处理，后面再介绍末尾符号链接分量的处理。

do_last()函数需要查找或创建末尾分量对应的 dentry 和 inode 实例，建立 file 实例与 inode 实例之间的关联，设置 file 实例，将 inode 实例 file_operations 实例指针赋予 file 实例，并调用其中的 open()函数（这些工作由 **vfs_open()**函数完成）。另外，如果末尾分量表示的文件不存在，且 open()系统调用指示了创建新文件，则还需要执行创建文件的操作。

do_last()函数定义在/fs/namei.c 文件内，代码如下：

```
static int do_last(struct nameidata *nd,struct file *file, const struct open_flags *op,int *opened)
/*file：新创建 file 实例，op：open_flags 实例指针，*opened：保存文件是否正确被打开*/
{
    struct dentry *dir = nd->path.dentry;
    int open_flag = op->open_flag;      /*open()系统调用传递的打开文件标记 flags 参数*/
    bool will_truncate = (open_flag & O_TRUNC) != 0;
    bool got_write = false;
    int acc_mode = op->acc_mode;
    unsigned seq;
    struct inode *inode;
    struct path save_parent = { .dentry = NULL, .mnt = NULL };
    struct path path;
    bool retried = false;
```

```
    int error;

    nd->flags &= ~LOOKUP_PARENT;
    nd->flags |= op->intent;

    if (nd->last_type != LAST_NORM) {       /*处理末尾分量是“.”或“..”的情况*/
        error = handle_dots(nd, nd->last_type);
        if (unlikely(error))
            return error;
        goto finish_open;
    }

    if (!(open_flag & O_CREAT)) {       /*没有设置 O_CREAT 标记，不创建新文件*/
        if (nd->last.name[nd->last.len])    /*末尾目录项之后还有‘/’字符表示末尾分量是目录项*/
            nd->flags |= LOOKUP_FOLLOW | LOOKUP_DIRECTORY;
        error = lookup_fast(nd, &path, &inode, &seq);
            /*在根文件系统中查找，不创建目录项，查找到了 dentry 实例返回 0，出错返回错误码，
            *需要到具体文件系统中查找，返回 1。
            */
        if (likely(!error))
            goto finish_lookup;  /*在根文件系统中查找到了 dentry 实例，跳转至 finish_lookup 处*/

        if (error < 0)
            return error;

        BUG_ON(nd->inode != dir->d_inode);
     } else {                   /*系统调用设置了 O_CREAT 标记，表示需要创建新文件*/
         error = complete_walk(nd);          /*/fs/namei.c*/
         if (error)
             return error;

         audit_inode(nd->name, dir, LOOKUP_PARENT);
         if (unlikely(nd->last.name[nd->last.len]))    /*目录后面有‘/’字符表示末尾分量是目录项*/
             return -EISDIR;         /*不能创建目录，返回错误码*/
    }

   /*可能需要到具体文件系统中查找路径分量，若需要创建文件则创建新文件*/
retry_lookup:
    if (op->open_flag & (O_CREAT | O_TRUNC | O_WRONLY | O_RDWR)) {
        error = mnt_want_write(nd->path.mnt);
                        /*申请对挂载文件系统的写权限，成功返回 0，/fs/namespace.c*/
        if (!error)
            got_write = true;
```

```
}
mutex_lock(&dir->d_inode->i_mutex);
error = lookup_open(nd, &path, file, op, got_write, opened);       /*/fs/namei.c*/
   /*
   *查找末尾分量 dentry 实例，先在根文件系统中查找，没找到再到具体文件系统中查找，
   *如果还没找到，且设置了 O_CREAT 标记位，则调用 i_op->create()函数创建新文件，
   *设置*opened |= FILE_CREATED，处理 O_EXCL 打开等，源代码请读者自行阅读。
   */
mutex_unlock(&dir->d_inode->i_mutex);
...     /*错误处理*/

if (*opened & FILE_CREATED) {       /*如果创建了新文件*/
     open_flag &= ~O_TRUNC;
     will_truncate = false;
     acc_mode = MAY_OPEN;
     path_to_nameidata(&path, nd);      /*nd->path=path*/
     goto finish_open_created;          /*跳转到 finish_open_created 处*/
}

/*查找到末尾分量 dentry 实例，且没有创建新文件*/
if (d_is_positive(path.dentry))     /*dentry 没有设置 DCACHE_MISS_TYPE 标记位*/
      audit_inode(nd->name, path.dentry, 0);

if (got_write) {
      mnt_drop_write(nd->path.mnt);
      got_write = false;
}
/*如果 open()系统调用同时设置了 O_EXCL 和 O_CREAT 标记，
*若文件已经存在，则返回-EEXIST 错误码。
*/
if (unlikely((open_flag & (O_EXCL | O_CREAT)) == (O_EXCL | O_CREAT))) {
     path_to_nameidata(&path, nd);
     return -EEXIST;
}

error = follow_managed(&path, nd);     /*处理分量是挂载点等情况（执行跳转），/fs/namei.c*/
if (unlikely(error < 0))
      return error;

BUG_ON(nd->flags & LOOKUP_RCU);
inode = d_backing_inode(path.dentry);       /*dentry 关联 inode 实例*/
seq = 0;                /* out of RCU mode, so the value doesn't matter */
if (unlikely(d_is_negative(path.dentry))) {       /*dentry 设置了 DCACHE_MISS_TYPE 标记位*/
```

```
            path_to_nameidata(&path, nd);
            return -ENOENT;
        }

    finish_lookup:              /*在查找到了末尾分量 dentry 实例*/
        if (nd->depth)          /*符号链接路径中的末尾分量，深度减 1*/
            put_link(nd);
        error = should_follow_link(nd, &path, nd->flags & LOOKUP_FOLLOW,inode, seq);
                                    /*末尾分量若是符号链接，深度加 1，保存信息，返回 1，否则返回 0*/
        if (unlikely(error))
            return error;           /*如果是符号链接且需要跟踪，函数返回 1，重新开启搜索*/

        /*末尾分量是符号链接，没有设置 O_PATH，返回错误码*/
        if (unlikely(d_is_symlink(path.dentry)) && !(open_flag & O_PATH)) {
            path_to_nameidata(&path, nd);
            return -ELOOP;
        }
        /*当前分量设置到 nd->path，以便后面对文件进行打开操作*/
        if ((nd->flags & LOOKUP_RCU) || nd->path.mnt != path.mnt) {
            path_to_nameidata(&path, nd);
        } else {
            save_parent.dentry = nd->path.dentry;
            save_parent.mnt = mntget(path.mnt);
            nd->path.dentry = path.dentry;
        }
        nd->inode = inode;
        nd->seq = seq;

finish_open:
        error = complete_walk(nd);
        ...     /*错误处理*/
        audit_inode(nd->name, nd->path.dentry, 0);
        error = -EISDIR;
        if ((open_flag & O_CREAT) && d_is_dir(nd->path.dentry))     /*不能创建目录项*/
            goto out;
        error = -ENOTDIR;
        if ((nd->flags & LOOKUP_DIRECTORY) && !d_can_lookup(nd->path.dentry))
            goto out;
        if (!d_is_reg(nd->path.dentry))         /*不是普通文件目录项*/
            will_truncate = false;

        if (will_truncate) {
            error = mnt_want_write(nd->path.mnt);
```

```
            ...   /*错误处理*/
            got_write = true;
        }

        /*文件已经存在或成功创建了文件*/
finish_open_created:
        error = may_open(&nd->path, acc_mode, open_flag);     /*打开权限检查，/fs/namei.c*/
        ...   /*错误处理*/

        BUG_ON(*opened & FILE_OPENED);     /* once it's opened, it's opened */
        error = vfs_open(&nd->path, file, current_cred());       /*VFS 打开操作，/fs/open.c*/
                        /*建立 file 与 inode 关联，初始化 file 实例，调用 f_op->open()函数等*/
        if (!error) {
            *opened |= FILE_OPENED;        /*标记文件已打开*/
        } else {
            ...   /*错误处理*/
        }
opened:      /*文件已打开*/
        error = open_check_o_direct(file);
                        /*若设置了 O_DIRECT 标记位，检查文件是否可直接读写，/fs/open.c*/
        ...   /*错误处理*/
        error = ima_file_check(file, op->acc_mode, *opened);
                            /*如果没有选择 IMA 选项，返回 0，/include/linux/ima.h*/
        ...   /*错误处理*/
        if (will_truncate) {              /*是否需要截短文件内容*/
            error = handle_truncate(file);        /*/fs/namei.c*/
            ...   /*错误处理*/
        }
    out:
        if (got_write)
            mnt_drop_write(nd->path.mnt);
        path_put(&save_parent);
        return error;
        ...
    }
```

do_last()函数主要工作是处理末尾路径分量，函数首先在根文件系统中查找对应的 dentry 实例，如果没有找到则到具体文件系统中查找，创建并设置 dentry 和 inode 实例。如果没有找到且设置了创建新文件 O_CREAT 标记位，执行创建新文件操作（i_op->create()），随后对查找到的文件或创建新文件调用标准函数 vfs_open()执行文件打开操作。

vfs_open()函数内主要是调用 path_get(&f->f_path)函数增加 nd->path 所指目录项 dentry 和 mount 实例引用计数，建立 file 实例与 inode 实例之间的关联（**file->f_inode = inode，file->f_op =inode->i_fop，file->f_mapping = inode->i_mapping**），设置 file 实例，调用 **file->f_op->open()**函数等。

vfs_open()函数可打开文件或目录项，也就是说 open()系统调用可以打开文件和目录。作者认为有必要

列出 vfs_open()函数代码，以加深读者对打开文件的理解。

```
int vfs_open(const struct path *path, struct file *file,const struct cred *cred)       /*/fs/open.c*/
{
    struct dentry *dentry = path->dentry;            /*dentry*/
    struct inode *inode = dentry->d_inode;        /*inode*/

    file->f_path = *path;          /*复制给 file 实例*/
    if (dentry->d_flags & DCACHE_OP_SELECT_INODE) {
        inode = dentry->d_op->d_select_inode(dentry, file->f_flags);
        if (IS_ERR(inode))
            return PTR_ERR(inode);
    }

    return do_dentry_open(file, inode, NULL, cred);        /*/fs/open.c*/
}
```

do_dentry_open()函数定义如下：

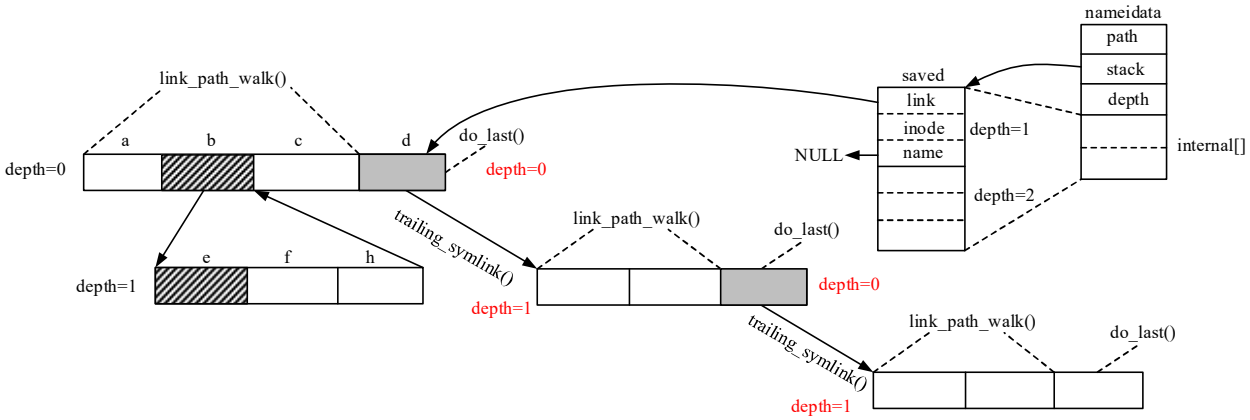
```
static int do_dentry_open(struct file *f,struct inode *inode,
                                    int (*open)(struct inode *, struct file *), const struct cred *cred)
/*open：NULL*/
{
    static const struct file_operations empty_fops = {};
    int error;

    f->f_mode = OPEN_FMODE(f->f_flags) | FMODE_LSEEK |FMODE_PREAD | FMODE_PWRITE;
                                                /*打开文件模式*/
    path_get(&f->f_path);
    f->f_inode = inode;
    f->f_mapping = inode->i_mapping;          /*文件地址空间，文件内容页缓存*/

    if (unlikely(f->f_flags & O_PATH)) { /*O_PATH 标记，file_operations 为空操作，不能操作文件*/
        f->f_mode = FMODE_PATH;
        f->f_op = &empty_fops;          /*空操作*/
        return 0;          /*返回*/
    }
    /*设置 f->f_mode 成员*/
    if (f->f_mode & FMODE_WRITE && !special_file(inode->i_mode)) {
        error = get_write_access(inode);
        ...
        error = __mnt_want_write(f->f_path.mnt);
        ...
        f->f_mode |= FMODE_WRITER;
    }
```

```
    if (S_ISREG(inode->i_mode))
        f->f_mode |= FMODE_ATOMIC_POS;

    f->f_op = fops_get(inode->i_fop);        /*f->f_op=inode->i_fop*/
    ...
    error = security_file_open(f, cred);     /*安全性检查*/
    ...
    error = break_lease(inode, f->f_flags);
    ...

    if (!open)
        open = f->f_op->open;
    if (open) {
        error = open(inode, f);              /*调用 open()函数*/
        ...
    }
    if ((f->f_mode & (FMODE_READ | FMODE_WRITE)) == FMODE_READ)
        i_readcount_inc(inode);
    if ((f->f_mode & FMODE_READ) &&likely(f->f_op->read || f->f_op->read_iter))
        f->f_mode |= FMODE_CAN_READ;
    if ((f->f_mode & FMODE_WRITE) &&likely(f->f_op->write || f->f_op->write_iter))
        f->f_mode |= FMODE_CAN_WRITE;

    f->f_flags &= ~(O_CREAT | O_EXCL | O_NOCTTY | O_TRUNC);     /*清这些标记位*/

    file_ra_state_init(&f->f_ra, f->f_mapping->host->i_mapping);
    return 0;
    ...
}
```

■末尾分量符号链接

最后，我们来看一下末尾分量是符号链接的情形，其它位置的符号链接在 link_path_walk()函数中处理完了。

如下图所示，假设原始路径中末尾分量是符号链接，它将由 do_last()函数处理，此时深度 depth 值为 0。



do_last()函数内判断深度是否大于 0，如果是则减 1，否则不减，然后调用 should_follow_link()函数判断末尾分量是否是要跟踪的符号链接，如果是则深度加 1，符号链接分量信息至 saved 实例，do_last()函数返回 1。

do_last()函数相关代码简列如下：

```
static int do_last(struct nameidata *nd,struct file *file, const struct open_flags *op,int *opened)
{
    ...
finish_lookup:
    if (nd->depth)      /*当前深度值非 0，减深度 1*/
        put_link(nd);
    error = should_follow_link(nd, &path, nd->flags & LOOKUP_FOLLOW,inode, seq);
    if (unlikely(error))
        return error;     /*要跟踪的符号链接，返回 1*/
    ...
}
```

如果末尾分量是符号链接，do_last()函数将返回 1，此时深度为 1。path_openat()函数随后调用 trailing_symlink(nd)函数，将返回路径 nd->stack[0].name 设为 NULL（不需要返回），读取 saved 实例中保存的符号链接分量中的路径名称，重新开启搜索，相关代码如下：

```
static struct file *path_openat(struct nameidata *nd,const struct open_flags *op, unsigned flags)
{
    ...
    while (!(error = link_path_walk(s, nd)) &&(error = do_last(nd, file, op, &opened)) > 0) {
        nd->flags &= ~(LOOKUP_OPEN|LOOKUP_CREATE|LOOKUP_EXCL);
        s = trailing_symlink(nd);       /*读取末尾符号链接分量中的路径，/fs/namei.c*/
        ...    /*错误处理*/
    }   /*遍历路径分量结束，如果末尾路径分量为符号链接，重新开启搜索*/
    ...
}
```

需要注意的是，对符号链接路径进行搜索时，此时深度值是 1。在 link_path_walk(s, nd)函数中搜索到符号链接路径中的末尾分量时，判断出其返回路径为空，函数将返回 0，不处理末尾分量，交由 do_last()

函数处理，相关代码如下。

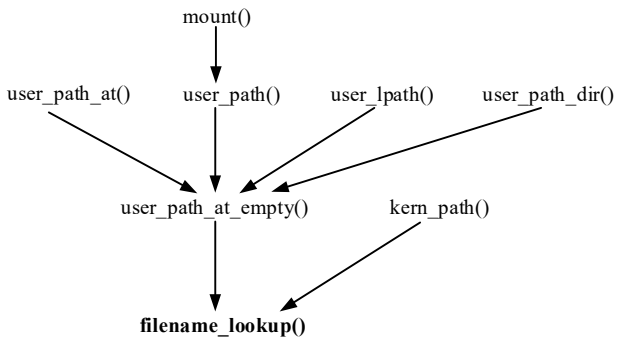
```
static int link_path_walk(const char *name, struct nameidata *nd)
{
    for(;;) {
        ...
        if (unlikely(!*name)) {        /*name 为 NULL，表示当前是末尾分量*/
  OK:
            if (!nd->depth)            /*原始路径中的末尾分量（深度值为 0），返回 0，不处理*/
                return 0;

            /*是末尾分量，但深度值非 0，即符号链接中路径的末尾分量*/
            name = nd->stack[nd->depth - 1].name;      /*符号链接返回路径为 NULL*/
            if (!name)             /*末尾符号链接路径中的末尾分量，返回 0，由 do_last()函数处理*/
                return 0;
            ...
        } else {
            ...
        }
        ...
    }       /*for 循环结束*/
}
```

do_last()函数在处理符号链接路径中的末尾分量时，深度值为 1，将会对其减 1，深度值恢复 0。如果符号链接的末尾分量还是符号链接，do_last()函数将保存符号链接信息（深度加 1），返回 1，而后重复上面对末尾符号链接分量的处理。

7.7.3 路径搜索函数

在内核代码中除 open()系统调用外，还有许多系统调用内会发起路径到节点的搜索，例如：mount()、mkdir()、rmdir()等系统调用。

内核其它路径搜索接口函数声明在/include/linux/namei.h 头文件内，函数定义在/fs/namei.c 文件内。路径搜索接口函数调用关系如下图所示：



以上接口函数最终都是调用 filename_lookup()函数执行路径搜索，函数代码如下（/fs/namei.c）：

```
static int filename_lookup(int dfd, struct filename *name, unsigned flags,struct path *path, struct path *root)
/*name：路径名称，flags：搜索标记，path：保存末尾分量搜索结果，root：通常为 NULL*/
{
```

```
    int retval;
    struct nameidata nd;
    if (IS_ERR(name))
        return PTR_ERR(name);
    if (unlikely(root)) {
        nd.root = *root;        /*root 指定的起点*/
        flags |= LOOKUP_ROOT;
    }
    set_nameidata(&nd, dfd, name);
    retval = path_lookupat(&nd, flags | LOOKUP_RCU, path);          /*搜索函数*/
    if (unlikely(retval == -ECHILD))
        retval = path_lookupat(&nd, flags, path);
    if (unlikely(retval == -ESTALE))
        retval = path_lookupat(&nd, flags | LOOKUP_REVAL, path);

    if (likely(!retval))
        audit_inode(name, path->dentry, flags & LOOKUP_PARENT);
    restore_nameidata();
    putname(name);
    return retval;      /*成功返回 0*/
}
```

这里调用的搜索函数为 **path_lookupat()**，它与 path_openat()函数很类似，代码如下（/fs/namei.c）：

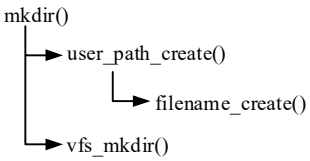
```
static int path_lookupat(struct nameidata *nd, unsigned flags, struct path *path)
{
    const char *s = path_init(nd, flags);
    int err;
    ...     /*错误处理*/

    while (!(err = link_path_walk(s, nd))&& ((err = lookup_last(nd)) > 0)) {
        s = trailing_symlink(nd);
        ...     /*错误处理*/
    }
    if (!err)
        err = complete_walk(nd);

    if (!err && nd->flags & LOOKUP_DIRECTORY)
        if (!d_can_lookup(nd->path.dentry))
            err = -ENOTDIR;
    if (!err) {
        *path = nd->path;       /*保存最末尾分量目录项信息*/
        nd->path.mnt = NULL;
        nd->path.dentry = NULL;
```

```
    }
    terminate_walk(nd);
    return err;        /*成功返回 0，否则返回错误码，注意不是文件描述符*/
}
```

path_lookupat()函数与 path_openat()函数的区别是，调用 lookup_last()函数处理末尾分量，此函数内调用 walk_component()函数搜索末尾分量。

path_lookupat()函数最后通过 path 参数返回搜索末尾分量的目录项信息，函数返回 0。

在创建目录项的 mkdir()等系统调用中，要先搜索最末尾分量的父目录项，在此目录项下查找看要创建的目录项是否已经存在，若已经存在则不能创建，不存在则可以创建，函数调用关系如下：

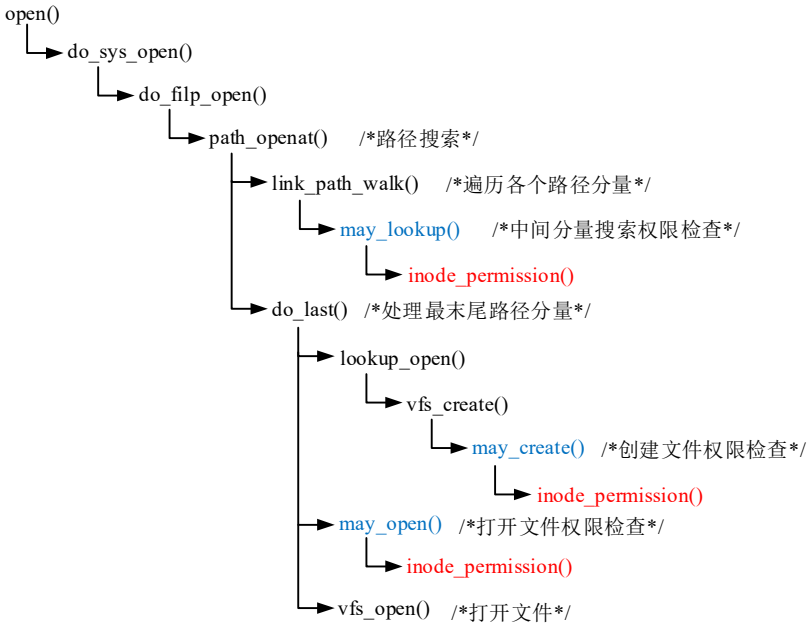


filename_create()函数是一个接口函数，函数内最终调用 link_path_walk()函数查找末尾分量的父目录项，在其下搜索末尾分量的 dentry 实例，如果存在则表示不能创建目录项，不存在则可以创建。vfs_mkdir()函数执行末尾目录项的创建。

filename_create()函数与 filename_lookup()函数类似，请读者自行阅读源代码。

7.7.4 权限检查

在前面介绍的路径到节点的搜索操作中，还有一个重要的操作没有介绍，那就是权限的检查。用户进程对任意一个目录项和文件的访问，内核都要对其权限进行检查，相关函数调用关系如下图所示。



以上 may_lookup()、may_create()和 may_open()函数最终都要调用 **inode_permission()**函数执行权限检查。下面将介绍 inode_permission(struct inode *inode, int mask)函数的实现。

inode_permission()函数需要传递两个参数，如下所述：

inode：inode 实例指针，普通目录项或文件对应的 inode 实例。

mask：标记，标记需要检查（期望获得）的权限，各标记位定义如下（/include/linux/fs.h）：

```
#define   MAY_EXEC          0x00000001      /*可执行权限，如果是目录，则表示搜索权限*/
```

```
#define   MAY_WRITE          0x00000002
                             /*可写权限，如果是创建文件需要 MAY_WRITE | MAY_EXEC 权限*/
#define   MAY_READ           0x00000004      /*可读权限*/
#define   MAY_APPEND         0x00000008      /*在文件末尾增加内容的权限*/
#define   MAY_ACCESS         0x00000010      /*访问权限*/
#define   MAY_OPEN           0x00000020      /*打开权限*/
#define   MAY_CHDIR          0x00000040      /*访问子目录权限*/
#define   MAY_NOT_BLOCK      0x00000080
```

mask 低 3 位从高到低依次是读、写和执行权限，用于与 inode.i_mode 成员比对。

inode_permission()函数定义在/fs/namei.c 文件内，代码如下：

```
int inode_permission(struct inode *inode, int mask)
{
    int retval;

    retval = sb_permission(inode->i_sb, inode, mask);     /*检查挂载权限，通过返回 0，/fs/namei.c*/
    if (retval)
        return retval;
    return   __inode_permission(inode, mask);        /*通过检查返回 0，否则返回错误码，/fs/namei.c*/
}
```

inode_permission()函数首先调用 **sb_permission**()函数检查挂载文件系统时设置的权限，用户进程不能突破文件系统挂载时设置的权限，如果通过则再调用**__inode_permission()**函数检查节点设置的权限，否则返回错误码。

以上两个检查都通过 inode_permission()函数返回 0，否则返回错误码。

1 挂载权限检查

sb_permission()函数定义如下：

```
static int sb_permission(struct super_block *sb, struct inode *inode, int mask)
{
    if (unlikely(mask & MAY_WRITE)) {        /*申请写权限*/
        umode_t mode = inode->i_mode;        /*节点中的模式成员（包含访问权限控制位）*/

        if ((sb->s_flags & MS_RDONLY) && (S_ISREG(mode) || S_ISDIR(mode) || S_ISLNK(mode)))
            return -EROFS;          /*只读挂载，不能申请写权限*/
    }
    return 0;        /*通过检查返回 0*/
}
```

挂载权限的检查比较简单，只有只读挂载申请写权限时不通过，其它情况都通过。

2 节点权限检查

__inode_permission()函数检查节点 inode 实例，看当前进程是否具有申请的权限，函数定义如下：

```
int __inode_permission(struct inode *inode, int mask)
{
    int retval;
    if (unlikely(mask & MAY_WRITE)) {
        if (IS_IMMUTABLE(inode))
            return -EACCES;
    }
    retval = do_inode_permission(inode, mask);      /*通用权限检查，/fs/namei.c*/
    if (retval)
        return retval;
    retval = devcgroup_inode_permission(inode, mask);
                        /*没有选择 CGROUP_DEVICE 配置选项，直接返回 0*/
    if (retval)
        return retval;
    return security_inode_permission(inode, mask);    /*没有选择 SECURITY 配置选项，直接返回 0*/
}
```

do_inode_permission()函数执行节点权限的检查，定义如下（/fs/namei.c）：

```
static inline int do_inode_permission(struct inode *inode, int mask)
{
    if (unlikely(!(inode->i_opflags & IOP_FASTPERM))) {
        if (likely(inode->i_op->permission))
            return inode->i_op->permission(inode, mask);        /*只会调用一次*/

        spin_lock(&inode->i_lock);
        inode->i_opflags |= IOP_FASTPERM;
        spin_unlock(&inode->i_lock);
    }
    return generic_permission(inode, mask);
}
```

如果节点关联的 inode_operations 实例定义了 permission()函数，在第一次对节点进行权限检查时调用此函数，后面再检查时不会再调用此函数了（除非文件被重新打开），而是调用 generic_permission()函数。

generic_permission(inode, mask)函数是通用的权限检查函数，定义如下（/fs/namei.c）：

```
int generic_permission(struct inode *inode, int mask)
{
    int ret;
    ret = acl_permission_check(inode, mask);    /*通过返回 0，不再执行后面的检查了*/
                        /*检查 inode->i_mode 中设置的权限以及访问控制列表，/fs/namei.c*/
    if (ret != -EACCES)
        return ret;

    /*acl_permission_check()返回-EACCES 才需要进行下面的检查，否则不需要执行*/
```

```
    /*目录项权限检查*/
    if (S_ISDIR(inode->i_mode)) {
        if (capable_wrt_inode_uidgid(inode, CAP_DAC_OVERRIDE)) /*能力检查，/kernel/capability.c*/
            return 0;
        if (!(mask & MAY_WRITE))
            if (capable_wrt_inode_uidgid(inode,CAP_DAC_READ_SEARCH))
                return 0;
        return -EACCES;
    }
    /*文件权限检查*/
    if (!(mask & MAY_EXEC) || (inode->i_mode & S_IXUGO))
        if (capable_wrt_inode_uidgid(inode, CAP_DAC_OVERRIDE))
            return 0;

    mask &= MAY_READ | MAY_WRITE | MAY_EXEC;
    if (mask == MAY_READ)
        if (capable_wrt_inode_uidgid(inode, CAP_DAC_READ_SEARCH))
            return 0;

    return -EACCES;
}
```

generic_permission()函数首先调用 acl_permission_check()函数检查 inode->i_mode 中设置的权限以及访问控制列表（ACL），如果通过，则返回 0，generic_permission()函数也将返回 0，检查通过，不再需要进行后面的检查。如果 acl_permission_check()函数检查不通过，将继续执行后面的检查。

上面调用的 capable_wrt_inode_uidgid()函数定义如下（/kernel/capability.c），主要是对进程能力的检查：

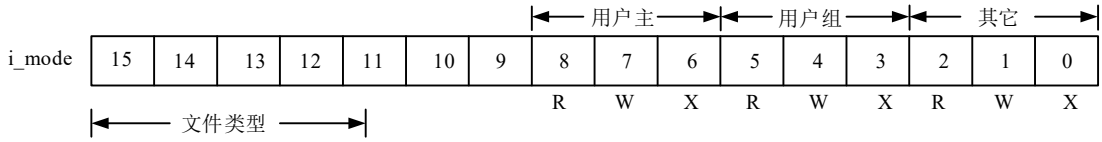
```
bool capable_wrt_inode_uidgid(const struct inode *inode, int cap)
{
    struct user_namespace *ns = current_user_ns();
    return    ns_capable(ns, cap) && kuid_has_mapping(ns, inode->i_uid) &&
                                        kgid_has_mapping(ns, inode->i_gid);
}
```

也就是说，如果 acl_permission_check()函数检查不通过，进程如果具有相应的能力，也具有访问权限，这里就不详细说明了，请读者自行阅读源代码。下面将重点介绍 acl_permission_check()函数的实现。

■访问权限检查

在介绍 acl_permission_check()函数前，先回顾一下 inode.i_mode 成员如何控制用户访问文件的权限。文件与进程类似也具有用户主/用户组属性，即文件属于哪个用户和用户组。inode 结构体中 **i_uid** 成员保存了文件所属用户主 ID，**i_gid** 保存所属用户组 ID。

inode 结构体中 i_mode 成员保存了用户主、用户组内其它用户以及其它用户访问文件的权限，如下图所示：



bit[8...6]标记了用户主对文件是否有读、写和执行权限，bit[5...3]标记同用户组其它用户对文件的读、写和执行权限，bit[2...0]标记其它用户对文件的读、写和执行权限。标记位置 1 表示具有相应的权限，为 0 表示不具有相应权限。

进程访问文件时，利用 task_struct 实例中 fsuid 和 fsgid 成员（通常同 euid 和 egid）去比对 inode 实例中 i_uid 和 i_gid 成员，看进程是不是和文件同用户主或用户组，还是其它用户，以确定访问权限。

acl_permission_check()函数定义如下（/fs/namei.c）：

```
static int acl_permission_check(struct inode *inode, int mask)
{
    unsigned int mode = inode->i_mode;

    if (likely(uid_eq(current_fsuid(), inode->i_uid)))          /*如果 fsuid=i_uid，即与文件同用户主*/
        mode >>= 6;                  /*右移 6 位，用户主权限位移至最低 3 位*/
    else {              /*进程与文件不是同用户主，可能是同用户组，也可能不同组*/
        if (IS_POSIXACL(inode) && (mode & S_IRWXG)) { /*挂载设置于了 MS_POSIXACL 标记*/
            int error = check_acl(inode, mask);        /*访问控制列表检查，/fs/namei.c*/
            if (error != -EAGAIN)
                return error;        /*检查通过，返回，不通过继续往下执行*/
        }

        if (in_group_p(inode->i_gid))        /*如果进程与文件同用户组*/
            mode >>= 3;                      /*右移 6 位，用户组权限位移至最低 3 位*/
    }

    if ((mask & ~mode & (MAY_READ | MAY_WRITE | MAY_EXEC)) == 0)     /*权限位检查*/
        return 0;
    return -EACCES;
}
```

acl_permission_check()函数检查流程如下：

（1）如果进程与文件同用户主，则检查 i_mode 中 bit[8...6]标记位，看用户主是否有 mask 中标记的权限（申请的权限）。

（2）如果进程与文件不是同用户主，并且满足以下 2 个条件，则执行访问控制列表检查，否则执行步骤（3）：

- 文件所在文件系统挂载时设置了 MS_POSIXACL 标记位。
- 文件 inode.i_mode 成员设置同组用户具有读写和执行权限。

访问控制列表是文件扩展属性的一种，扩展属性是“名称=值”的字符串，附加到文件中，可认为是文件的元信息，扩展属性需要具体文件系统类型的支持。用户进程通过 setxattr()系统调用设置扩展属性，通过 getattr()系统调用获取扩展属性。访问控制列表标记了更精细的文件访问权限控制，这里就不详细介绍。

check_acl()函数用于访问控制列表（ACL）检查，如果通过，则 acl_permission_check()函数返回，权限检查通过，否则还要执行步骤（3）。

（3）如果进程与文件同用户组，则检查 i_mode 中 bit[5...3]标记位，看进程是否具有 mask 中标记的权限，否则（进程用户是其它用户）检查 i_mode 中 bit[3...0]标记位，看进程是否具有想要的权限。

如果以上权限检查通过，acl_permission_check()函数返回 0，否则返回错误码。

内核在/fs/open.c 文件内定义了进程检查文件访问权限的系统调用（调用 inode_permission()函数），如下所示：

●**access**(const char __user *filename, int mode)：检查进程对文件（目录项）的访问权限（通过 uid/gid 检查），通过返回 0，否则返回错误码。

●**faccessat**(int dfd, const char __user * filename, int mode)：与 access()类似，只不过可以指定相对路径的搜索起点。

7.7.5 close()

进程关闭文件的系统调用为 close()，系统调用只需要一个参数，就是关闭文件的描述符。关闭文件的主要工作是：清空文件描述符对应 file 指针数组项，清零文件在 close_on_exec 和 open_fds 位图中的相应位，重置进程 files_struct 实例 next_fd 成员为释放文件描述符（如果需要），调用 file->f_op 实例中的释放文件函数，断开 file 实例与 dentry 和 inode 实例之间关联，最后释放各数据结构实例（引用计数减 1）。

close()系统调用在/fs/open.c 文件内实现，代码如下：

```
SYSCALL_DEFINE1(close, unsigned int, fd)        /*fd 表示关闭文件描述符*/
{
    int retval = __close_fd(current->files, fd);     /*/fs/file.c*/
    ...
    return retval;
}
```

关闭文件操作由__close_fd()函数完成，定义如下（/fs/file.c）：

```
int __close_fd(struct files_struct *files, unsigned fd)
{
    struct file *file;
    struct fdtable *fdt;

    spin_lock(&files->file_lock);
    fdt = files_fdtable(files);          /*获取 fdtable 实例指针*/
    if (fd >= fdt->max_fds)
        goto out_unlock;
    file = fdt->fd[fd];          /*file 实例指针*/
    if (!file)
        goto out_unlock;
    rcu_assign_pointer(fdt->fd[fd], NULL);     /*file 指针数组项赋 NULL*/
    __clear_close_on_exec(fd, fdt);            /*清零 close_on_exec 位图中相应位*/
    __put_unused_fd(files, fd);                /*释放文件描述符，重置 files->next_fd，/fs/file.c*/
    spin_unlock(&files->file_lock);
```

```
    return filp_close(file, files);          /*释放各结构体实例，/fs/open.c*/
    ...
}
```

__close_fd()函数在最后调用__put_unused_fd(files, fd)函数释放文件描述符，调用 filp_close(file, files)函数断开 file 实例与 inode 等实例之间的关联，并释放各数据结构实例。

__put_unused_fd(files, fd)函数定义如下：

```
static void __put_unused_fd(struct files_struct *files, unsigned int fd)
{
    struct fdtable *fdt = files_fdtable(files);
    __clear_open_fd(fd, fdt);          /*清零 open_fds 指向位图中相应位*/
    if (fd < files->next_fd)
        files->next_fd = fd;          /*如果 fd < files->next_fd，设置 next_fd 成员*/
}
```

1 释放 file 实例

filp_close(file, files)函数定义在/fs/open.c 文件内，代码如下：

```
int filp_close(struct file *filp, fl_owner_t id)
{
    int retval = 0;
    ...
    if (filp->f_op->flush)
        retval = filp->f_op->flush(filp, id);      /*调用具体文件系统类型定义的函数*/

    if (likely(!(filp->f_mode & FMODE_PATH))) {
        dnotify_flush(filp, id);
        locks_remove_posix(filp, id);
    }
    fput(filp);        /*释放 file 实例，/fs/file_table.c*/
    return retval;
}
```

fput(filp)函数内对 file 实例引用计数减 1，若达到 0 则释放 file 实例。如果当前不是在中断处理程序中且不是内核线程，则将释放 file 实例添加到进程工作队列中，适时释放 file 实例。如果当前是中断处理程序或是内核线程，则将 file 实例添加到全局链表 delayed_fput_list，而后由延时工作释放 file 实例。这两个途径最后都是调用__fput()函数释放 file 实例。

fput(filp)函数定义如下（/fs/file_table.c）：

```
void fput(struct file *file)
{
    if (atomic_long_dec_and_test(&file->f_count)) {        /*引用计数减 1 后是否为 0*/
        struct task_struct *task = current;

        if (likely(!in_interrupt() && !(task->flags & PF_KTHREAD))) {
                                                    /*非中断处理程序，非内核线程*/
```

```
            init_task_work(&file->f_u.fu_rcuhead, ____fput);
                                        /*rcu_head 结构体成员回调函数设为____fput()*/
            if (!task_work_add(task, &file->f_u.fu_rcuhead, true))
                                    /*交由进程执行回调函数，返回 0，/kernel/task_work.c*/
                return;
        }
        /*在中断处理程序中，或是内核线程，将 file 实例添加到 delayed_fput_list 链表*/
        if (llist_add(&file->f_u.fu_llist, &delayed_fput_list))
            schedule_delayed_work(&delayed_fput_work, 1);     /*由延时工作调用__fput()函数释放*/
    }
}
```

如果是用户进程释放 file 实例（非中断处理程序），将最终调用____fput()函数释放 file 实例，函数定义如下（/fs/file_table.c）：

```
static void ____fput(struct callback_head *work)
{
    __fput(container_of(work, struct file, f_u.fu_rcuhead));       /*/fs/file_table.c*/
}
```

__fput()函数定义如下，它同时也是 delayed_fput_work 延时工作释放 file 实例的回调函数：

```
static void __fput(struct file *file)
{
    struct dentry *dentry = file->f_path.dentry;        /*文件目录项*/
    struct vfsmount *mnt = file->f_path.mnt;
    struct inode *inode = file->f_inode;        /*文件节点*/

    might_sleep();

    fsnotify_close(file);
    eventpoll_release(file);
    locks_remove_file(file);

    if (unlikely(file->f_flags & FASYNC)) {       /*同步文件*/
        if (file->f_op->fasync)
            file->f_op->fasync(-1, file, 0);
    }
    ima_file_free(file);
    if (file->f_op->release)
        file->f_op->release(inode, file);          /*文件系统类型定义的释放函数*/
    security_file_free(file);
    if (unlikely(S_ISCHR(inode->i_mode) && inode->i_cdev != NULL &&
                 !(file->f_mode & FMODE_PATH))) {
        cdev_put(inode->i_cdev);                /*释放字符设备文件*/
```

```
    }
    fops_put(file->f_op);        /*减小模块引用计数，/include/linux/fs.h*/
    put_pid(file->f_owner.pid);
    if ((file->f_mode & (FMODE_READ | FMODE_WRITE)) == FMODE_READ)
        i_readcount_dec(inode);
    if (file->f_mode & FMODE_WRITER) {
        put_write_access(inode);
        __mnt_drop_write(mnt);     /*/fs/namespace.c*/
    }
    file->f_path.dentry = NULL;       /*断开与 dentry 和 inode 实例之间关联*/
    file->f_path.mnt = NULL;
    file->f_inode = NULL;
    file_free(file);        /*释放 file 实例，/fs/file_table.c*/
    dput(dentry);       /*释放 dentry 实例及 inode 实例*/
    mntput(mnt);       /*释放 mount 实例（减小引用计数），/fs/namespace.c*/
}
```

7.8 目录操作

描述目录项或文件在根文件系统中位置的路径字符串称为目录，如：/mnt/abc.c 表示根目录下 mnt 目录下的 abc.c 文件。每个路径分量称为目录项，如：/，mnt，abc.c 都是目录项，/表示根目录项，abc.c 称为文件目录项。其实每个目录项就都是一个文件名，对应一个文件，普通目录项的文件内容是其子目录项和文件目录项列表，文件目录项对应文件内容就是普通文件内容。

目录操作其实是对目录项文件内容的操作。例如：创建目录项，就是在其父目录项文件内容中增加一个目录项，重命名就是修改父目录项文件内容中目录项。

本节简要介绍目录项操作相关系统调用。

7.8.1 创建目录项

用户进程创建目录项的系统调用为 mkdir()，实现代码如下（/fs/namei.c）：

```
SYSCALL_DEFINE2(mkdir, const char __user *, pathname, umode_t, mode)
/*pathname：路径名称，mode：模式（访问权限）*/
{
    return sys_mkdirat(AT_FDCWD, pathname, mode);     /*同 mkdirat()系统调用实现函数*/
}
```

mkdir()系统调用内部调用 mkdirat()系统调用的实现函数完成目录项的创建。mkdirat()系统调用的实现代码如下（/fs/namei.c）：

```
SYSCALL_DEFINE3(mkdirat, int, dfd, const char __user *, pathname, umode_t, mode)
{
    struct dentry *dentry;
    struct path path;
    int error;
    unsigned int lookup_flags = LOOKUP_DIRECTORY;     /*搜索标记，表示搜索目录*/
```

```
retry:
    dentry = user_path_create(dfd, pathname, &path, lookup_flags);   /*/fs/namei.c*/
                /*创建 dentry 实例，到具体文件系统中查找是否已存在此目录项，存在则返回错误码*/
    if (IS_ERR(dentry))      /*目录项已存在，返回错误码*/
        return PTR_ERR(dentry);

    /*以下是创建目录项*/
    if (!IS_POSIXACL(path.dentry->d_inode))
        mode &= ~current_umask();      /*屏蔽访问权限*/
    error = security_path_mkdir(&path, dentry, mode);
    if (!error)
        error = vfs_mkdir(path.dentry->d_inode, dentry, mode);          /*创建目录项，/fs/namei.c*/
    done_path_create(&path, dentry);      /*完成目录项创建，/fs/namei.c*/
    if (retry_estale(error, lookup_flags)) {
        lookup_flags |= LOOKUP_REVAL;
        goto retry;
    }
    return error;        /*成功返回 0*/
}
```

mkdirat()系统调用中首先调用 **user_path_create**()函数在根文件系统以及具体文件系统中查找是否已存在相同的目录项，如果存在则返回错误码，否则继续往下执行；然后，修改访问权限后，调用 **vfs_mkdir**()函数创建目录项。

vfs_mkdir()函数首先调用 may_create(dir, dentry)函数检查对 path.dentry->d_inode 表示的父目录项文件是否有权限创建目录项，如果有则调用 **dir->i_op->mkdir(dir, dentry, mode)**函数（inode_operations 实例中 mkdir()函数）在具体文件系统中创建目录项，源代码请读者自行阅读。

7.8.2 创建特殊文件

用户进程可通过 mknod()系统调用创建特殊文件，如设备文件，实现函数如下（/fs/namei.c）：

```
SYSCALL_DEFINE3(mknod, const char __user *, filename, umode_t, mode, unsigned, dev)
/*filename：设备文件名称，mode：文件模式（文件类型，访问权限），dev：设备号*/
{
    return sys_mknodat(AT_FDCWD, filename, mode, dev);
}
```

mknod()系统调用复用 mknodat()系统调用的实现函数，定义如下（/fs/namei.c）：

```
SYSCALL_DEFINE4(mknodat, int, dfd, const char __user *, filename, umode_t, mode,unsigned, dev)
{
    struct dentry *dentry;
    struct path path;
    int error;
    unsigned int lookup_flags = 0;

    error = may_mknod(mode);      /*权限检查*/
    ...
```

```
retry:
    dentry = user_path_create(dfd, filename, &path, lookup_flags);
                /*创建 dentry 实例，到具体文件系统中查找是否已存在此文件，存在则返回错误码*/
    if (IS_ERR(dentry))
        return PTR_ERR(dentry);

    if (!IS_POSIXACL(path.dentry->d_inode))
        mode &= ~current_umask();      /*屏蔽访问权限*/
    error = security_path_mknod(&path, dentry, mode, dev);
    ...
    switch (mode & S_IFMT) {        /*文件类型*/
        case 0: case S_IFREG:       /*普通文件*/
            error = vfs_create(path.dentry->d_inode,dentry,mode,true);      /*也可以创建普通文件*/
            break;
        case S_IFCHR: case S_IFBLK:      /*字符设备、块设备文件，需要 CAP_MKNOD 能力*/
            error = vfs_mknod(path.dentry->d_inode,dentry,mode,new_decode_dev(dev));
            break;
        case S_IFIFO: case S_IFSOCK:      /*创建命名管道或套接字*/
            error = vfs_mknod(path.dentry->d_inode,dentry,mode,0);
            break;
    }
out:
    done_path_create(&path, dentry);
    if (retry_estale(error, lookup_flags)) {
        lookup_flags |= LOOKUP_REVAL;
        goto retry;
    }
    return error;       /*成功返回 0*/
}
```

mknodat()系统调用中根据创建文件类型的不同调用不同的实现函数，对于字符、块设备文件等特殊文件，调用 vfs_mknod()函数创建设备文件（进程需要 CAP_MKNOD 能力）。

vfs_mknod()函数内调用 path.dentry->d_inode->i_op->mknod(dir, dentry, mode, dev)函数创建设备文件，源代码请读者自行阅读。

另外，mknod()/mknodat()系统调用也可以用来创建普通文件。

内核在/fs/namei.c 文件内还实现了文件重命名的 rename()系统调用、创建链接文件的 link()/symlink()系统调用等，其实现方式与上面介绍的系统调用类似，请读者自行阅读源代码。

7.8.3 删除目录项

删除目录项的系统调用为 rmdir()，删除的目录项必须是空目录项，系统调用实现函数如下（/fs/namei.c）：

```
SYSCALL_DEFINE1(rmdir, const char __user *, pathname)
{
    return do_rmdir(AT_FDCWD, pathname);       /*/fs/namei.c*/
```

```
}

rmdir()系统调用由 do_rmdir()函数实现，函数代码如下：
static long do_rmdir(int dfd, const char __user *pathname)
{
    int error = 0;
    struct filename *name;
    struct dentry *dentry;
    struct path path;
    struct qstr last;
    int type;
    unsigned int lookup_flags = 0;
retry:
    name = user_path_parent(dfd, pathname,&path, &last, &type, lookup_flags);
                            /*搜索到达删除目录项的父目录项，name 为删除目录项，/fs/namei.c*/
    if (IS_ERR(name))
        return PTR_ERR(name);

    switch (type) {      /*删除目录项的类型，只能删除普通的目录项*/
    case LAST_DOTDOT:
        error = -ENOTEMPTY;
        goto exit1;
    case LAST_DOT:
        error = -EINVAL;
        goto exit1;
    case LAST_ROOT:
        error = -EBUSY;
        goto exit1;
    }

    error = mnt_want_write(path.mnt);      /*获取挂载的写权限，/fs/namespace.c*/
    if (error)
        goto exit1;

    mutex_lock_nested(&path.dentry->d_inode->i_mutex, I_MUTEX_PARENT);
    dentry = __lookup_hash(&last, path.dentry, lookup_flags);      /*查找删除目录项 dentry 实例*/
    ...
    error = security_path_rmdir(&path, dentry);
    ...
    error = vfs_rmdir(path.dentry->d_inode, dentry);   /*/fs/namei.c*/
     /*调用 may_delete()检查删除权限，调用 path.dentry->d_inode->i_op->rmdir()删除目录项*/
    ...
    return error;
```

}

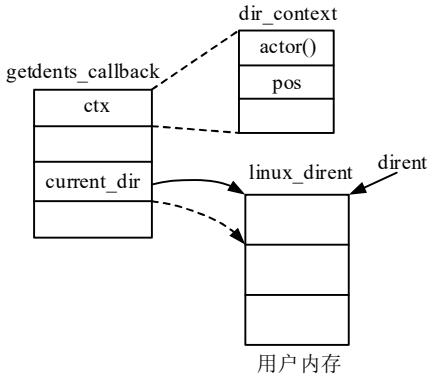
do_rmdir()函数调用 user_path_parent()函数搜索到删除目录项的父目录项，调用**__lookup_hash**()函数查找删除目录项的 dentry 实例，最后调用 **vfs_rmdir**()函数删除目录项。

vfs_rmdir()函数内首先调用 may_delete()函数检查进程删除目录项的权限，然后调用父目录项关联 inode 实例关联的 inode_operations 实例中的 **rmdir(dir, dentry)**函数删除子目录项。

7.8.4 读取目录项

用户进程可读取某个目录项下的所有子目录项（含文件目录项），相关系统调用定义在/fs/readdir.c 文件内，下面以为 getdents()系统调用为例，介绍其实现。

如下图所示，用户进程向系统调用传递一个缓存区指针 dirent，用于保存 linux_dirent 结构体实例数组，每个实例表示一个目录项信息，getdents()系统调用读取目录项信息并添充用户缓存区。getdents()系统调用内部通过 getdents_callback 结构体来暂存读取目录项信息。



getdents_callback 结构体定义如下（/fs/readdir）：

```
struct getdents_callback {
    struct dir_context ctx;        /*dir_context 结构体成员，/include/linux/fs.h*/
    struct linux_dirent __user * current_dir;      /*指向当前 linux_dirent 实例（用户空间内存）*/
    struct linux_dirent __user * previous;         /*指向上一个 linux_dirent 实例*/
    int count;            /*用户缓存区空闲区域长度，填充目录项信息时变小*/
    int error;            /*返回错误码*/
};
```

getdents_callback 结构体主要成员简介如下：

●**ctx**：dir_context 结构体成员，结构体定义在/include/linux/fs.h 头文件：

```
struct dir_context {
    const filldir_t actor;    /*填充 linux_dirent 实例的函数，函数原型定义在/include/linux/fs.h 头文件*/
    loff_t pos;               /*目录项文件当前位置*/
};
```

●**current_dir**：指向用户内存当前 linux_dirent 实例，linux_dirent 结构体定义如下（/fs/readdir）：

```
struct linux_dirent {
    unsigned long d_ino;       /*目录项对应节点编号*/
    unsigned long d_off;       /*下一个目录项在文件内容中偏移量*/
    unsigned short d_reclen;   /*当前 linux_dirent 实例长度*/
    char        d_name[1];     /*目录项名称，长度可变*/
};
```

●**previous**：指向上一个填充的 linux_dirent 实例。

我们先简要介绍一下 getdents()系统调用填充用户缓存区的过程，用户进程需要指定缓存区的基地址和大小（长度），缓存区中填充的是 linux_dirent 实例数组，linux_dirent 实例的长度根据目录项名称长度的不同而不同。linux_dirent 结构体中 d_reclen 成员表示本实例的长度，d_name[]保存名称字符串，d_off 是下一下目录项在目录项文件内容中的偏移量。

getdents_callback 结构体中 current_dir 指向当前填充的 linux_dirent 实例，previous 指向上一个填充的实例，count 表示缓存区剩余空间大小，每填充一个 linux_dirent 实例，count 值减小。

dir_context 结构体中 actor()函数是填充 linux_dirent 实例的回调函数，这里为 **filldir()**。

getdents()系统调用调用目录项文件操作结构中的 iterate()函数，从文件系统中读取目录项信息，然后调用 actor()回调函数，逐个将目录项信息填充至用户缓存区。若缓存区填满，或子目录项读取完毕，iterate()函数返回，系统调用也返回。

getdents()系统调用实现函数如下：

```
SYSCALL_DEFINE3(getdents, unsigned int, fd,struct linux_dirent __user *, dirent, unsigned int, count)
/*fd：文件描述符，dirent：用户缓存区指针，count：缓存区长度*/
{
    struct fd f;
    struct linux_dirent __user * lastdirent;
    struct getdents_callback buf = {
        .ctx.actor = filldir,          /*填充 linux_dirent 实例的函数，/fs/readdir*/
        .count = count,            /*用户缓存区长度*/
        .current_dir = dirent     /*指向用户缓存区*/
    };
    int error;

    if (!access_ok(VERIFY_WRITE, dirent, count))        /*权限检查*/
        return -EFAULT;
    f = fdget(fd);
    ...
    error = iterate_dir(f.file, &buf.ctx);      /*填充目录项信息，/fs/readdir.c*/
    if (error >= 0)           /*填充了缓存区*/
        error = buf.error;            /*若缓存区不够大，返回-EINVAL*/
    lastdirent = buf.previous;      /*本次读取最后一个填充的 linux_dirent 实例指针*/
    if (lastdirent) {
        if (put_user(buf.ctx.pos, &lastdirent->d_off))        /*下一目录项在文件内容中偏移量*/
            error = -EFAULT;
        else
            error = count - buf.count;        /*填充用户缓存区的字节数*/
    }
    fdput(f);
    return error;       /*返回填充的字节数*/
}
```

getdents()系统调用实现函数中调用 iterate_dir()函数读取文件系统中目录项文件内容，填充至用户缓存区，函数定义如下：

```
int iterate_dir(struct file *file, struct dir_context *ctx)
{
    struct inode *inode = file_inode(file);
    int res = -ENOTDIR;
    if (!file->f_op->iterate)
        goto out;
    ...                 /*权限检查*/
    if (!IS_DEADDIR(inode)) {
        ctx->pos = file->f_pos;                    /*目录项文件当前位置*/
        res = file->f_op->iterate(file, ctx);      /*调用文件操作中的 iterate()函数，调用 actor()函数*/
        file->f_pos = ctx->pos;                    /*调整文件位置*/
        fsnotify_access(file);
        file_accessed(file);
    }
    mutex_unlock(&inode->i_mutex);
out:
    return res;        /*读取了目录项，返回 0（读取完了返回-EIO？）*/
}
```

文件系统类型定义的 iterate()函数读取目录项文件内容，调用 actor()回调函数将子目录项信息填充至用户缓存区，直到缓存区填满（函数返回 0）或子目录项读取完了（函数返回-EIO）。

这里有一个问题，用户如何知道读取完了目录项下的所有子目录项呢？因为用户事先并不知道目录项下有多少个子目录项。作者认为用户可以一直执行 getdents()系统调用直到返回-EIO 错误码，表示读取目录项文件完毕。

7.8.5 进程目录

前面介绍过，每个进程有一个根目录和当前工作目录，最开始这两个目录都指向根文件系统的根目录项，进程可以通过系统调用改变这两个目录位置。子进程继承父进程的根目录和当前工作目录。

1 进程根目录

进程根目录是其搜索绝对路径的起点，进程根目录是内核根文件系统中的一个目录项，其目的就是将进程对内核根文件系统的视图限制在进程根目录之下。简单地说，进程只能看到根文件系统的一棵子树。进程根目录只能往下改，不能往上改，即进程只能缩小其对根文件系统的视图，而不能扩大。

进程可通过 **chroot()**系统调用修改自身的根目录位置，系统调用实现如下（/fs/open.c）：

```
SYSCALL_DEFINE1(chroot, const char __user *, filename)
/*filename：新指定目录*/
{
    struct path path;
    int error;
    unsigned int lookup_flags = LOOKUP_FOLLOW | LOOKUP_DIRECTORY;
```

```
retry:
    error = user_path_at(AT_FDCWD, filename, lookup_flags, &path);      /*搜索目录*/
    ...

    error = inode_permission(path.dentry->d_inode, MAY_EXEC | MAY_CHDIR);    /*权限检查*/
    ...

    error = -EPERM;
    if (!ns_capable(current_user_ns(), CAP_SYS_CHROOT))/*进程需具有 CAP_SYS_CHROOT 能力*/
        goto dput_and_out;
    error = security_path_chroot(&path);
    ...

    set_fs_root(current->fs, &path);          /*设置进程根目录指向 path*/
    error = 0;
dput_and_out:
    path_put(&path);
    if (retry_estale(error, lookup_flags)) {
        lookup_flags |= LOOKUP_REVAL;
        goto retry;
    }
out:
    return error;       /*成功返回 0*/
}
```

2 当前工作目录

进程可通过 **getcwd**(char __user * buf, unsigned long size)系统调用获取当前工作目录（保存至 buf 缓存区），这是起始于进程根目录的绝对路径，系统调用实现在/fs/dcache.c 文件内，请读者自行阅读源代码。

进程可通过 **chdir()**系统调用设置进程当前工作目录，这也是用户进程经常进行的操作，进程修改当前工作目录时，不能突破进程根目录的限制。

chdir()系统调用实现函数如下（/fs/open.c）：

```
SYSCALL_DEFINE1(chdir, const char __user *, filename)
{
    struct path path;
    int error;
    unsigned int lookup_flags = LOOKUP_FOLLOW | LOOKUP_DIRECTORY;
retry:
    error = user_path_at(AT_FDCWD, filename, lookup_flags, &path);    /*搜索路径*/
    ..

    error = inode_permission(path.dentry->d_inode, MAY_EXEC | MAY_CHDIR);    /*权限检查*/
    ...
```

```
        set_fs_pwd(current->fs, &path);        /*设置进程当前工作目录为 path*/

dput_and_out:
        path_put(&path);
        if (retry_estale(error, lookup_flags)) {
                lookup_flags |= LOOKUP_REVAL;
                goto retry;
        }
out:
        return error;           /*成功返回 0*/
}
```

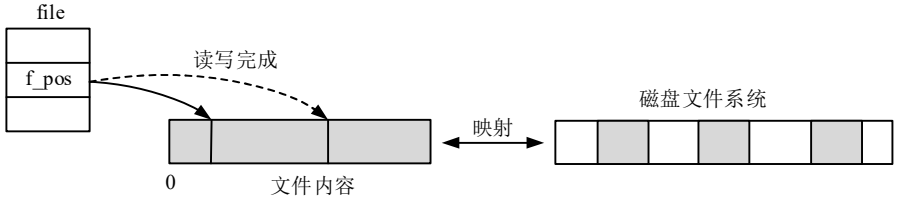
7.9 文件操作

上一节主要介绍了用户进程对目录项文件的操作。对普通文件的操作就比较多了，有定位文件当前位置、读写文件内容、修改文件元信息（修改用户主、访问权限等）、文件控制、监控文件变化、文件锁、删除文件等等。

本节对部分文件操作系统调用做简要介绍。

7.9.1 定位文件

任何文件的内容对内核来说都只是一个连续的字节流，每个字节在文件内容中具有一个相对于文件开始处的偏移量，例如，第一个字节偏移量为 0，第二个字节偏移量为了 1，依此类推。文件内容在逻辑上是连续的，但在磁盘文件系统中可能保存在离散的数据块中，如下所示。



如上图所示，file 实例中 f_ops 成员指向文件当前位置（逻辑偏移量），下一次读写操作时，从 f_pos 指向的字节开始。通常，读写完成后，f_ops 会后移相应字节位置，如读了 10 个字节，f_ops 就后移 10 个字节，f_ops 就像是一个指针。

这里有个问题需要说明，子进程可能与父进程共用 file 实例，任何一方对文件的读写都会修改 f_ops 值，一个进程可能不知道其它进程已经修改了 f_ops 值，编程时要注意。

用户进程可以通过 lseek()/llseek()系统调用直接设置文件的当前位置，系统调用实现在/fs/read_write.c 文件内，下面以 lseek(fd,offset, **whence**)系统调用为例简要说明其实现。

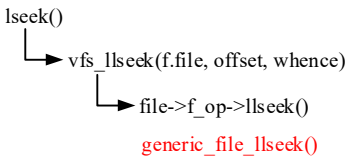
lseek()系统调用中 whence 参数，指示设置的偏移量是相对于何处的偏移，取值定义如下：

```
/*/include/uapi/linux/fs.h*/
#define   SEEK_SET        0    /*offset 是相对于文件开始的偏移量*/
#define   SEEK_CUR        1    /*offset 是相对于文件当前位置的偏移量*/
#define   SEEK_END        2    /*offset 是相对于文件结尾的偏移量*/
#define   SEEK_DATA       3    /*对于普通文件，表示相对于文件开始的偏移量（同 SEEK_SET）*/
#define   SEEK_HOLE       4    /*定位到文件结尾（offset 值不能大于文件大小值）*/
#define   SEEK_MAX     SEEK_HOLE
```

```
SYSCALL_DEFINE3(lseek, unsigned int, fd, off_t, offset, unsigned int, whence)
/*fd：文件描述符，offset：偏移量数值，whence：指示偏移量是相对于哪个位置的偏移*/
{
    off_t retval;
    struct fd f = fdget_pos(fd);      /*获取 file 实例*/
    if (!f.file)
        return -EBADF;

    retval = -EINVAL;
    if (whence <= SEEK_MAX) {
        loff_t res = vfs_llseek(f.file, offset, whence);      /*/fs/read_write.c*/
        retval = res;
        if (res != (loff_t)retval)
            retval = -EOVERFLOW;      /* LFS: should only happen on 32 bit platforms */
    }
    fdput_pos(f);
    return retval;
}
```

lseek()系统调用内调用 vfs_llseek()函数设置文件位置，此函数内又调用文件操作结构 file_operations 实例中的 llseek()函数设置文件位置。

各文件系统类型定义的 file_operations 实例中 llseek()函数通常为通用的 generic_file_llseek()函数，以上函数调用关系如下图所示：



generic_file_llseek()函数定义在/fs/read_write.c 文件，用于修改 file->f_pos 值，源代码请读者自行阅读。

有的文件可能没有当前位置，也不能设置，如设备文件等，这时其 file_operations 实例不会定义 llseek()函数，执行 lseek()系统调用将返回-ESPIPE 错误码。

7.9.2 读写文件

进程对普通文件的操作建立在打开文件的基础之上，打开文件后，进程通过文件描述符来标识文件。进程对普通文件的操作主要是对其内容的读/写操作，读/写操作系统调用最终调用 file_operations 实例中的相关函数实现。

1 读文件

读文件的 read()系统调用，从文件当前位置读出文件内容至用户缓存区，实现如下（/fs/read_write.c）：

```
SYSCALL_DEFINE3(read, unsigned int, fd, char __user *, buf, size_t, count)
/*fd：文件描述符，buf：指向用户内存缓存区，count：欲读取字节数*/
{
    struct fd f = fdget_pos(fd);
```

```
    ssize_t ret = -EBADF;

    if (f.file) {
        loff_t pos = file_pos_read(f.file);          /*保存读之前文件当前位置*/
        ret = vfs_read(f.file, buf, count, &pos);   /*读文件内容，可能会修改 pos 值，也可能不修改*/
        if (ret >= 0)
            file_pos_write(f.file, pos);            /*设置 file->f_pos 为 pos*/
        fdput_pos(f);
    }
    return ret;     /*返回实际读数据的字节数*/
}
```

读文件 read()系统调用具有三个参数，fd 参数为打开文件描述符，buf 参数为保存数据的缓冲区指针（用户空间指针），count 参数表示欲读取数据长度，字节数。

read()系统调用首先获取文件的当前位置写入局部变量 pos，然后调用通用的 vfs_read()函数完成读操作。vfs_read()函数从文件中读取数据写入 buf 缓存区，如果要修改文件当前位置则修改 pos 变量值，否则不修改，函数返回实际读取数据的字节数。read()系统调用随后用 pos 值设置文件当前位置（直接赋值），返回读取数据字节数。

vfs_read()函数定义如下（/fs/read_write.c）：

```
ssize_t vfs_read(struct file *file, char __user *buf, size_t count, loff_t *pos)
/*pos：*pos 保存读取操作后文件的当前位置*/
{
    ssize_t ret;

    if (!(file->f_mode & FMODE_READ))        /*文件不可读（打开模式不对），返回错误码*/
        return -EBADF;
    if (!(file->f_mode & FMODE_CAN_READ))
        return -EINVAL;
    if (unlikely(!access_ok(VERIFY_WRITE, buf, count)))
        return -EFAULT;

    ret = rw_verify_area(READ, file, pos, count);
                    /*读写字节数量检查以及强制锁检查，/fs/read_write.c*/
    if (ret >= 0) {        /*ret 表示可以读取字节数量*/
        count = ret;
        ret = __vfs_read(file, buf, count, pos);     /*/fs/read_write.c*/
        if (ret > 0) {
            fsnotify_access(file);
            add_rchar(current, ret);       /*增加 current->ioac.rchar 值，/include/linux/sched.h*/
         }
         inc_syscr(current);     /*增加 current->ioac.syscr 值，/include/linux/sched.h*/
    }
    return ret;        /*返回实际读数据字节数*/
```

```
}
```

vfs_read()函数调用__vfs_read()函数执行读操作，函数定义如下：

```
ssize_t __vfs_read(struct file *file, char __user *buf, size_t count, loff_t *pos)
{
    if (file->f_op->read)                    /*优先调用 f_op->read()函数*/
        return file->f_op->read(file, buf, count, pos);      /*设备文件通常定义 read()函数*/
    else if (file->f_op->read_iter)              /*没有定义 f_op->read()函数，调用 f_op->read_iter()函数*/
        return new_sync_read(file, buf, count, pos);    /*通用同步读函数，/fs/read_write.c*/
    else
        return -EINVAL;
}
```

如果文件操作结构 file_operations 实例中定义了 read()函数，__vfs_read()函数则调用此函数完成读操作，设备文件通常会定义此函数。

如果没有定义 read()函数则调用通用的同步读文件函数 **new_sync_read()**，此函数内将调用 file_operations 实例中 read_iter()函数完成文件读操作。基于块设备的文件系统类型通常定义 read_iter()函数，而不会定义 read()函数。read_iter()函数从文件地址空间页缓存中读取数据，复制到用户空间。

由于 new_sync_read()函数涉及到文件地址空间（页缓存）的操作，相关内容到第 11 章再做介绍。这里读者只需知道内核为普通文件在内存中建立了文件内容缓存（页缓存），读操作是从此缓存中读取数据，复制到用户空间，如果缓存中无数据，内核会先从介质文件系统中读取数据至缓存，然后再复制到用户空间。

2 写文件

写文件内容系统调用为 write()，其实现与读操作非常相似，系统调用实现如下（/fs/read_write.c）：

```
SYSCALL_DEFINE3(write, unsigned int, fd, const char __user *, buf,size_t, count)
{
    struct fd f = fdget_pos(fd);
    ssize_t ret = -EBADF;
    if (f.file) {
        loff_t pos = file_pos_read(f.file);    /*保存文件当前位置*/
        ret = vfs_write(f.file, buf, count, &pos);   /*/fs/read_write.c*/
        if (ret >= 0)
            file_pos_write(f.file, pos);      /*重置文件位置*/
        fdput_pos(f);
    }
    return ret;              /*返回实际写数据字节数*/
}
```

write()系统调用中调用 vfs_write()函数完成写操作，函数定义在/fs/read_write.c 文件内，代码如下：

```
ssize_t vfs_write(struct file *file, const char __user *buf, size_t count, loff_t *pos)
{
    ssize_t ret;
    if (!(file->f_mode & FMODE_WRITE))      /*写权限检查*/
```

```
        return -EBADF;
    if (!(file->f_mode & FMODE_CAN_WRITE))
        return -EINVAL;
    if (unlikely(!access_ok(VERIFY_READ, buf, count)))
        return -EFAULT;

    ret = rw_verify_area(WRITE, file, pos, count);     /*检查读写数据长度以及强制锁检查*/
    if (ret >= 0) {
        count = ret;
        file_start_write(file);
        ret = __vfs_write(file, buf, count, pos);    /*/fs/read_write.c*/
        if (ret > 0) {
            fsnotify_modify(file);
            add_wchar(current, ret);
        }
        inc_syscw(current);
        file_end_write(file);
    }
    return ret;
}
```

写文件内容操作最终由__vfs_write()函数完成，函数定义如下（/fs/read_write.c）：

```
ssize_t __vfs_write(struct file *file, const char __user *p, size_t count,loff_t *pos)
{
    if (file->f_op->write)
        return    file->f_op->write(file, p, count, pos);     /*设备文件通常定义 write()函数*/
    else if (file->f_op->write_iter)          /*块设备文件系统类型通常定义 write_iter()函数*/
        return new_sync_write(file, p, count, pos);    /*同步写操作函数，/fs/read_write.c*/
    else
        return -EINVAL;
}
```

__vfs_write()函数内优先调用 file->f_op->write()函数执行写文件操作，如果没有定义此函数则调用通用的同步写函数 **new_sync_write()**完成写操作。

new_sync_write()函数调用 f_op->write_iter()函数，将用户空间数据写入文件内容页缓存，由内核在适当的时候将页缓存中数据写入块设备，到第 11 章再介绍此函数的实现。

3 变种读写

前面介绍的 read()/write()系统调用，在文件的当前位置执行读写操作，操作完之后默认会修改文件当前位置（具体文件系统类型可以决定不修改）。read()/write()系统调用中用户空间的缓存区必须是一个连续的内存区。

为此内核定义了其它一些变种的读写系统调用，可以在文件指定位置读写，并且读写操作后文件当前位置不变，或者从用户空间多个缓存区向文件写入数据，读出数据写入多个用户缓存区等。

■在指定位置读写

内核在/fs/read_write.c 文件内定义了 pread64()/pwrite64()系统调用，从参数指定位置读写文件，读写后不修改文件当前位置。下面简单看一下 pread64()系统调用实现（/fs/read_write.c）：

```
SYSCALL_DEFINE4(pread64, unsigned int, fd, char __user *, buf,size_t, count, loff_t, pos)
/*pos：相对于文件开始的偏移量*/
{
    struct fd f;
    ssize_t ret = -EBADF;

    if (pos < 0)
        return -EINVAL;
    f = fdget(fd);
    if (f.file) {
        ret = -ESPIPE;
        if (f.file->f_mode & FMODE_PREAD)
            ret = vfs_read(f.file, buf, count, &pos);     /*参数指定文件位置*/
        fdput(f);
    }
    return ret;
}
```

pread64()系统调用与 read()系统调用实现非常类似，只不过读取文件位置由参数指定，读之后不修改文件当前位置。

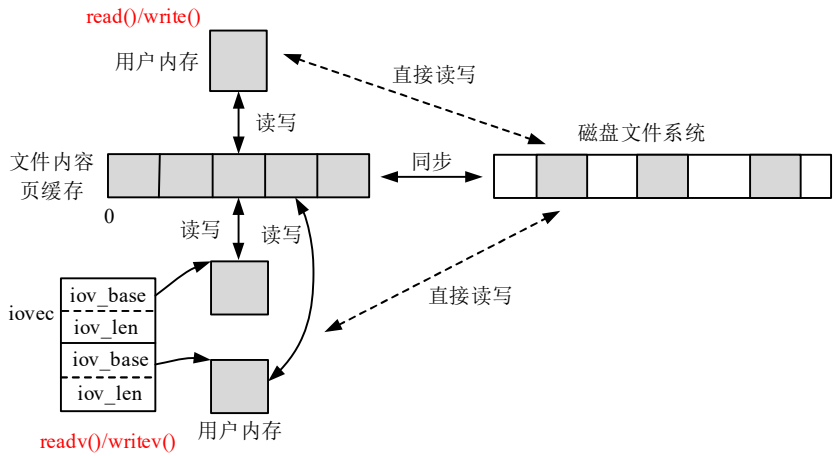
pwrite64()系统调用实现与 pread64()类似，请读者自行阅读源代码。

■分散写和集中读

read()/write()系统调用要求用户空间缓存区是一个连续的区域，如下图所示（页缓存中文件内容逻辑上是连续的）。

readv()/writev()系统调用中用户缓存区可以是多个离散的区域，如下图所示。每个缓存区域由 iovec 结构体表示。标题说的分散写和集中读，是指可以将多个用户空间分散内存区域的数据写入文件内容连续的区域，可以将文件中连续的数据读出到用户空间多个分散的内存区域中。

读写文件系统调用也可以不经过文件页缓存，用户内存直接与块设备交互，称为直接读写，不过这样做的效率不高，不常用，这是后话，到第 11 章再介绍。



用户空间每个分散的内存区域由 iovec 结构体表示，结构体定义如下（/include/uapi/linux/uio.h）：

```
struct iovec
{
    void __user *iov_base;      /*内存区域指针（地址）*/
    __kernel_size_t iov_len;     /*长度*/
};
```

readv()/writev()系统调用实现在/fs/read_write.c 文件内，下面以 readv()为例，说明其实现：

```
SYSCALL_DEFINE3(readv, unsigned long, fd, const struct iovec __user *, vec,unsigned long, vlen)
{
    struct fd f = fdget_pos(fd);
    ssize_t ret = -EBADF;

    if (f.file) {
        loff_t pos = file_pos_read(f.file);
        ret = vfs_readv(f.file, vec, vlen, &pos);       /*read()系统调用为 vfs_read()函数，/fs/read_write.c*/
        if (ret >= 0)
            file_pos_write(f.file, pos);
        fdput_pos(f);
    }

    if (ret > 0)
        add_rchar(current, ret);
    inc_syscr(current);
    return ret;
}
```

readv()与 read()系统调用的实现非常相似，只不过将 vfs_read()函数换成了 vfs_readv()函数。vfs_readv()函数其实也非常好理解，它对每个分散的用户空间缓存区域发起一次读操作，每次读取数据的长度就是缓存区的长度，每次读取填充一个缓存区域，函数源代码请读者自行阅读。

writev()系统调用与 readv()系统调用又很相似，只不过改变了数据流方向，通过多次向文件执行写操作来完成多个缓存区数据的写入，请读者自行阅读源代码。

内核在/fs/read_write.c 文件内还定义了以上两种变种读写操作的组合，即 preadv()/pwritev()系统调用，请读者自行阅读源代码。

另外，Linux 内核还定义了特有的 sendfile()/sendfile64()系统调用，用于简化通过网络在两个本地文件之间进行的数据传输过程，请读者自行阅读源代码。

7.9.3 文件属性

文件属性是指保存在文件节点（inode）中的文件元信息，如文件属主/组、访问权限、时间戳等。内核用 iattr/kstat 结构体来收集文件属性，用户进程可通过系统调用设置/获取文件属性。

1 属性操作

内核对设置和获取文件属性分别使用了不同的数据结构，设置/修改文件属性时使用 iattr 结构体，获取属性时使用 kstat 结构体，但它们操作的对象都是节点中的文件属性。

■设置/修改文件属性

内核在设置/修改文件属性时，通过 iattr 结构体暂存信息，结构体定义如下（/include/linux/fs.h）：

```
struct iattr {
    unsigned int    ia_valid;          /*标记*/
    umode_t         ia_mode;           /*模式（访问权限）*/
    kuid_t          ia_uid;            /*用户主 ID*/
    kgid_t          ia_gid;            /*用户组 ID*/
    loff_t          ia_size;           /*文件大小*/
    struct timespec     ia_atime;      /*创建时间*/
    struct timespec     ia_mtime;      /*修改时间*/
    struct timespec     ia_ctime;      /*修改节点时间*/

    struct file *ia_file;      /*指向 file 实例，不是属性，起辅助作用*/
};
```

iattr 结构体中 ia_valid 是标记成员，标记要执行什么操作，用于权限检查和操作类型识别，成员取值定义如下（/include/linux/fs.h）：

```
#define   ATTR_MODE       (1 << 0)       /*要修改模式（访问权限）*/
#define   ATTR_UID        (1 << 1)       /*要修改用户主*/
#define   ATTR_GID        (1 << 2)       /*要修改用户组*/
#define   ATTR_SIZE       (1 << 3)       /*要修改文件大小*/
#define   ATTR_ATIME      (1 << 4)       /*实例中时间值有效*/
#define   ATTR_MTIME      (1 << 5)
#define   ATTR_CTIME      (1 << 6)
#define   ATTR_ATIME_SET      (1 << 7)      /*要设置时间戳*/
#define   ATTR_MTIME_SET      (1 << 8)
#define   ATTR_FORCE          (1 << 9)    /*强制执行属性操作*/
#define   ATTR_ATTR_FLAG      (1 << 10)
#define   ATTR_KILL_SUID      (1 << 11)
```

```
#define   ATTR_KILL_SGID      (1 << 12)
#define   ATTR_FILE           (1 << 13)
#define   ATTR_KILL_PRIV      (1 << 14)
#define   ATTR_OPEN           (1 << 15) /* Truncating from open(O_TRUNC) */
#define   ATTR_TIMES_SET      (1 << 16)
```

内核在/fs/attr.c 文件内定义了通过 iattr 结构体设置/修改文件属性相关的函数，如下所示：

●int **inode_change_ok**(const struct inode *inode, struct iattr *attr)：检查修改文件属性的权限，通过返回 0，否则返回-EPERM 错误码。

●void **setattr_copy**(struct inode *inode, const struct iattr *attr)：将 attr 中文件属性值写入 inode 实例，只是将属性值写入内核 inode 实例中，没有写入文件系统中。

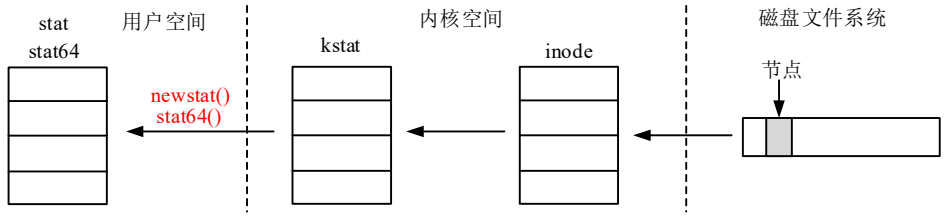
●int **notify_change**(struct dentry * dentry, struct iattr * attr, struct inode **delegated_inode)：修改 dentry 关联 inode 的属性值，调用 inode->i_op->**setattr**(dentry, attr)函数写入文件系统中。

文件系统类型定义的节点操作结构中的 **setattr()**函数需要调用 inode_change_ok()函数进行权限检查，然后调用 setattr_copy()函数将属性值写入 inode 实例，最后将修改后的 inode 实例写入磁盘文件系统。

修改文件属性的 chown()等系统调用中，将调用 notify_change()函数，后面将介绍。

■获取文件属性

用户进程可以通过 newstat()/stat64()等系统调用获取文件属性，这些系统调用实现在/fs/stat.c 文件内，系统调用返回用户空间的文件属性由 stat/stat64 结构体表示，而在内核中文件属性由 kstat 结构体表示，如下图所示。



kstat 定义在/include/linux/stat.h 头文件：

```
struct kstat {
    u64         ino;
    dev_t       dev;
    umode_t     mode;
    unsigned int    nlink;
    kuid_t      uid;
    kgid_t      gid;
    dev_t       rdev;
    loff_t      size;
    struct timespec    atime;
    struct timespec    mtime;
    struct timespec    ctime;
    unsigned long blksize;
    unsigned long long blocks;
};
```

stat/stat64 结构体定义在/include/uapi/asm-generic/stat.h 头文件：

```
struct stat {
    unsigned long st_dev;          /* Device. */
    unsigned long st_ino;          /* File serial number. */
    unsigned int    st_mode;       /*模式（访问权限）*/
    unsigned int    st_nlink;      /* Link count. */
    unsigned int    st_uid;        /*用户主 ID*/
    unsigned int    st_gid;        /*用户组 ID*/
    unsigned long st_rdev;        /* Device number, if device. */
    unsigned long __pad1;
    long        st_size;          /*文件大小，字节数*/
    int         st_blksize;       /* Optimal block size for I/O.   */
    int         __pad2;
    long        st_blocks;        /* Number 512-byte blocks allocated. */
    long        st_atime;         /* Time of last access. */
    unsigned long st_atime_nsec;
    long        st_mtime;         /* Time of last modification. */
    unsigned long st_mtime_nsec;
    long        st_ctime;         /* Time of last status change. */
    unsigned long st_ctime_nsec;
    unsigned int    __unused4;
    unsigned int    __unused5;
};
```

stat64 结构体与 stat 类似，用于在 32 位系统中获取 64 位的属性数据。

内核在/fs/stat.c 文件内定义了获取文件属性的系统调用簇，请读者自行阅读源代码。

2 修变文件属主

创建文件时，文件属主 ID 为创建进程 euid，用户组 ID 为创建进程 egid，进程可以通过系统调用 chown()/lchown()/fchown()等改变文件属主 ID 和用户组 ID。

改变文件属主的系统调用在/fs/open.c 文件内实现，如下所示：

- int **chown**(const char __user * filename, uid_t user, gid_t group);
- int **lchown**(const char __user * filename, uid_t user, gid_t group);
- int **fchownat**(int dfd, const char __user * filename, uid_t user,gid_t group, int flag);
- int **fchown**(unsigned int fd, uid_t user, gid_t group);

chown()改变由 filename 参数命名文件的属主和用户组。

lchown()与 chown()类似，不同之处在于若参数 filename 是符号链接，则将改变符号链接文件本身的属性，而不是符号链接指向文件。

fchown()与 chown()类似，只不过由文件描述符来表示文件。

fchownat()与 chown()类似，不过可以指定相对路径的起点和搜索标记。

下面以 chown()系统调用为例，说明其实现，实现函数如下（/fs/open.c）：

```
SYSCALL_DEFINE3(chown, const char __user *, filename, uid_t, user, gid_t, group)
{
    return sys_fchownat(AT_FDCWD, filename, user, group, 0);    /*fchownat()系统调用实现函数*/
}
```

fchownat()系统调用实现函数定义如下：

```
SYSCALL_DEFINE5(fchownat, int, dfd, const char __user *, filename, uid_t, user,gid_t, group, int, flag)
/*flag：这里为 0*/
{
    struct path path;
    int error = -EINVAL;
    int lookup_flags;

    if ((flag & ~(AT_SYMLINK_NOFOLLOW | AT_EMPTY_PATH)) != 0)
        goto out;

    lookup_flags = (flag & AT_SYMLINK_NOFOLLOW) ? 0 : LOOKUP_FOLLOW;
    if (flag & AT_EMPTY_PATH)
        lookup_flags |= LOOKUP_EMPTY;
retry:
    error = user_path_at(dfd, filename, lookup_flags, &path);      /*搜索文件*/
    ...
    error = mnt_want_write(path.mnt);      /*挂载需有可写属性*/
    ...
    error = chown_common(&path, user, group);        /*修改属主，/fs/open.c*/
    mnt_drop_write(path.mnt);
out_release:
    ...
out:
    return error;
}
```

fchownat()系统调用实现函数搜索到文件后，调用 chown_common()函数执行修改文件属主操作，函数定义如下：

```
static int chown_common(struct path *path, uid_t user, gid_t group)
{
    struct inode *inode = path->dentry->d_inode;
    struct inode *delegated_inode = NULL;
    int error;
    struct iattr newattrs;
    kuid_t uid;
    kgid_t gid;
```

```
    uid = make_kuid(current_user_ns(), user);        /*局部 UID 转全局 UID*/
    gid = make_kgid(current_user_ns(), group);

retry_deleg:
    newattrs.ia_valid =    ATTR_CTIME;
    if (user != (uid_t) -1) {
        if (!uid_valid(uid))
            return -EINVAL;
        newattrs.ia_valid |= ATTR_UID;        /*要修改用户主*/
        newattrs.ia_uid = uid;
    }
    if (group != (gid_t) -1) {
        if (!gid_valid(gid))
            return -EINVAL;
        newattrs.ia_valid |= ATTR_GID;        /*要修改用户组*/
        newattrs.ia_gid = gid;
    }
    if (!S_ISDIR(inode->i_mode))        /*不是普通目录项（就是文件）*/
        newattrs.ia_valid |=ATTR_KILL_SUID | ATTR_KILL_SGID | ATTR_KILL_PRIV;
    mutex_lock(&inode->i_mutex);
    error = security_path_chown(path, uid, gid);
    if (!error)
        error = notify_change(path->dentry, &newattrs, &delegated_inode);      /*修改属性，见上文*/
    mutex_unlock(&inode->i_mutex);
    ...
    return error;
}
```

前面讲到，notify_change()函数调用的文件系统类型定义的 inode->i_op->**setattr**(dentry, attr)函数中需要调用函数 inode_change_ok()检查进程权限，修改文件用户主的权限如下（满足一条即可）：

●进程 fsuid 等于文件 inode->i_uid（非特权用户可以改变自己文件的属主）。

●进程具有 CAP_CHOWN 能力（特权用户，与 inode->i_uid 同用户命名空间，可改变任何文件的属主）。

●进程没有 CAP_CHOWN 能力，但是将文件用户主改为 inode->i_uid（其实就是不修改）。

修改文件用户组的权限如下（满足一条即可）：

●进程 fsuid 等于文件 **inode->i_uid**（非特权用户可以改变自己文件的用户组）。

●进程具有 CAP_CHOWN 能力（特权用户，可改变任何文件的属组）。

●进程没有 CAP_CHOWN 能力，但是同组用户可将用户组改为 inode->i_gid（其实就是不修改）。

3 修变访问权限

创建文件时，会设置文件的访问权限，进程可通过 chmod()/fchmod()等系统调用修改文件访问权限。这些系统调用定义在/fs/open.c 文件内，下面以 chmod()系统调用为例，介绍其实现。

```
SYSCALL_DEFINE2(chmod, const char __user *, filename, umode_t, mode)
/*mode：新访问权限*/
{
    return sys_fchmodat(AT_FDCWD, filename, mode);        /*fchmodat()系统调用实现函数*/
}

SYSCALL_DEFINE3(fchmodat, int, dfd, const char __user *, filename, umode_t, mode)
{
    struct path path;
    int error;
    unsigned int lookup_flags = LOOKUP_FOLLOW;
retry:
    error = user_path_at(dfd, filename, lookup_flags, &path);      /*搜索文件*/
    if (!error) {
        error = chmod_common(&path, mode);      /*修改访问权限，/fs/open.c*/
        path_put(&path);
        ...
    }
    return error;
}
```

chmod_common()函数与 chown_common()函数类似，也是调用 notify_change()函数修改文件属性，请读者自行阅读源代码。

修改文件访问权限的权限如下（满足一条即可）：

●进程 fsuid 等于文件 inode->i_uid（用户可修改自己文件的访问权限）。

●进程具有 CAP_FOWNER 能力。

如果以上两条中有一条通过，则还需要检查设置 S_ISGID 标记位的权限，满足以下一条则可以设置 S_ISGID 标记位，否则将清除 mode 中的 S_ISGID 标记位（即不能设置此位）：

●进程具有 CAP_FSETID 能力。

●进程没有 CAP_FSETID 能力，进程用户组与 inode->i_gid 相同，或进程用户组与 attr->ia_gid（欲修改的文件用户组）相同。

前面讲进程目录信息时提到，进程关联 fs_struct 实例的 **umask** 成员，用于屏蔽进程新创建文件的访问权限，也就是新创建的文件将清除 **umask** 成员中标记的权限。

进程可通过 umask()系统调用设置 **umask** 成员值，如下所示（/kernel/sys.c）：

```
SYSCALL_DEFINE1(umask, int, mask)
{
    mask = xchg(&current->fs->umask, mask & S_IRWXUGO);        /*直接用 mask 设置 umask 成员*/
    return mask;        /*返回原值*/
}
```

7.9.4 进程文件控制

前面介绍的文件读写、获取/设置属性，涉及到文件的底层，即在文件在存储介质中的数据。本小节介绍的文件控制主要是对用户进程文件（file 实例）的控制，涉及文件在内核中的行为和属性，不太涉及文件的底层。

用户进程可通过 **fcntl()**系统调用向进程文件发出控制命令，获取/设置进程文件参数，控制文件行为等。文件控制相关命令定义在头文件/include/uapi/asm-generic/fcntl.h 和/include/uapi/linux/fcntl.h：

```
#define   F_DUPFD          0    /*复制文件描述符*/
#define   F_GETFD          1    /*读 close_on_exec 位图中值（是否在 execve()系统调用中关闭）*/
#define   F_SETFD          2    /*设置/清零 close_on_exec 位图中标记位*/
#define   F_GETFL          3    /*获取 file->f_flags */
#define   F_SETFL          4    /*设置 file->f_flags */
#ifndef   F_GETLK                /*在/arch/mips/include/uapi/asm/fcntl.h 头文件中定义了*/
    #define   F_GETLK     5    /*检测是否可加 POSIX 锁*/
    #define   F_SETLK     6    /*加/解 POSIX 锁，非阻塞*/
    #define   F_SETLKW    7    /*加/解 POSIX 锁，阻塞*/
#endif
#ifndef F_SETOWN                 /*在/arch/mips/include/uapi/asm/fcntl.h 头文件中定义了*/
    #define   F_SETOWN    8    /*设置文件属主进程*/
    #define   F_GETOWN    9    /*获取文件属主进程*/
#endif
#ifndef F_SETSIG
    #define   F_SETSIG      10 /*设置文件信号*/
    #define   F_GETSIG      11 /*获取文件信号*/
#endif

#ifndef   CONFIG_64BIT
    #ifndef     F_GETLK64       /*在/arch/mips/include/uapi/asm/fcntl.h 头文件中定义了*/
        #define     F_GETLK64     12     /*    using 'struct flock64' */
        #define     F_SETLK64     13
        #define     F_SETLKW64    14
    #endif
#endif

#ifndef   F_SETOWN_EX
    #define     F_SETOWN_EX     15      /*设置扩展的文件属主（进程）信息*/
    #define     F_GETOWN_EX     16      /*获取扩展的文件属主（进程）信息*/
#endif

#ifndef   F_GETOWNER_UIDS
    #define   F_GETOWNER_UIDS    17
#endif

#define   F_OFD_GETLK       36
```

```
#define   F_OFD_SETLK       37
#define   F_OFD_SETLKW      38
...
#define F_LINUX_SPECIFIC_BASE   1024

/*/include/uapi/linux/fcntl.h*/
#define   F_SETLEASE      (F_LINUX_SPECIFIC_BASE + 0)       /*加/解租借锁*/
#define   F_GETLEASE      (F_LINUX_SPECIFIC_BASE + 1)       /*获取已加租借锁类型*/

#define   F_CANCELLK      (F_LINUX_SPECIFIC_BASE + 5)
#define   F_DUPFD_CLOEXEC         (F_LINUX_SPECIFIC_BASE + 6)
                                                    /*复制文件描述符并设置 FD_CLOEXEC*/
#define   F_NOTIFY          (F_LINUX_SPECIFIC_BASE+2)       /*只适用于 dnotify*/
#define   F_SETPIPE_SZ      (F_LINUX_SPECIFIC_BASE + 7)
#define   F_GETPIPE_SZ      (F_LINUX_SPECIFIC_BASE + 8)

#define   F_ADD_SEALS       (F_LINUX_SPECIFIC_BASE + 9)
#define   F_GET_SEALS       (F_LINUX_SPECIFIC_BASE + 10)
...
```

fcntl()系统调用在/fs/fcntl.c 文件内实现，实现函数如下：

```
SYSCALL_DEFINE3(fcntl, unsigned int, fd, unsigned int, cmd, unsigned long, arg)
/*cmd：命令值，arg：参数值*/
{
    struct fd f = fdget_raw(fd);
    long err = -EBADF;
    ...      /*错误处理*/

    if (unlikely(f.file->f_mode & FMODE_PATH)) {          /*目录项文件，对命令有限制*/
        if (!check_fcntl_cmd(cmd))
            goto out1;
    }

    err = security_file_fcntl(f.file, cmd, arg);
    if (!err)
        err = do_fcntl(fd, cmd, arg, f.file);       /*分配器函数，/fs/fcntl.c*/
out1:
    fdput(f);
out:
    return err;
}
```

fcntl()系统调用每个命令值由一个无符号整数 arg 传递，系统调用内调用 **do_fcntl()**函数完成命令的执

行。do_fcntl()函数就是一个命令分配器，不同的命令调用不同的执行函数，函数代码如下。

```
static long do_fcntl(int fd, unsigned int cmd, unsigned long arg,struct file *filp)
{
    long err = -EINVAL;

    switch (cmd) {
    case F_DUPFD:
        err = f_dupfd(arg, filp, 0);            /*复制文件描述符*/
        break;
    case F_DUPFD_CLOEXEC:
        err = f_dupfd(arg, filp, O_CLOEXEC);    /*复制文件描述符，并设置 FD_CLOEXEC*/
        break;
    case F_GETFD:               /*获取 close_on_exec 位图中值*/
        err = get_close_on_exec(fd) ? FD_CLOEXEC : 0;
        break;
    case F_SETFD:           /*设置 close_on_exec 位图中值，置 1 或清 0*/
        err = 0;
        set_close_on_exec(fd, arg & FD_CLOEXEC);
        break;
    case F_GETFL:
        err = filp->f_flags;            /*返回打开文件标记*/
        break;
    case F_SETFL:
        err = setfl(fd, filp, arg);       /*设置打开文件标记*/
        break;
#if BITS_PER_LONG != 32
    case F_OFD_GETLK:
#endif
    case F_GETLK:
        err = fcntl_getlk(filp, cmd, (struct flock __user *) arg);   /*检测是否可加 POSIX 锁，/fs/locks.c*/
        break;
#if BITS_PER_LONG != 32
    case F_OFD_SETLK:
    case F_OFD_SETLKW:
#endif
    case F_SETLK:
    case F_SETLKW:
        err = fcntl_setlk(fd, filp, cmd, (struct flock __user *) arg);    /*加/解 POSIX 锁，/fs/locks.c*/
        break;
    case F_GETOWN:
        err = f_getown(filp);               /*获取文件（file 实例）属主进程 PID*/
        force_successful_syscall_return();
        break;
```

```
case F_SETOWN:
        f_setown(filp, arg, 1);             /*设置文件（file 实例）属主进程 PID*/
        err = 0;
        break;
case F_GETOWN_EX:          /*获取文件属主（进程）信息，f_owner_ex 结构体为媒介*/
        err = f_getown_ex(filp, arg);
        break;
case F_SETOWN_EX:          /*设置文件属主（进程）信息，f_owner_ex 结构体为媒介*/
        err = f_setown_ex(filp, arg);
        break;
case F_GETOWNER_UIDS:
        err = f_getowner_uids(filp, arg);
        break;
case F_GETSIG:             /*获取信号*/
        err = filp->f_owner.signum;
        break;
case F_SETSIG:             /*设置信号*/
        if (!valid_signal(arg)) {
                break;
        }
        err = 0;
        filp->f_owner.signum = arg;
        break;
case F_GETLEASE:
        err = fcntl_getlease(filp);                 /*查询 file 加租借锁的类型，/fs/locks.c*/
        break;
case F_SETLEASE:
        err = fcntl_setlease(fd, filp, arg);        /*加/解租借锁，/fs/locks.c*/
        break;
case F_NOTIFY:
        err = fcntl_dirnotify(fd, filp, arg);       /*只适用于 dnotity*/
        break;
case F_SETPIPE_SZ:
case F_GETPIPE_SZ:
        err = pipe_fcntl(filp, cmd, arg);           /*命名管道*/
        break;
case F_ADD_SEALS:
case F_GET_SEALS:
        err = shmem_fcntl(filp, cmd, arg);          /*共享内存控制，/mm/shmem.c*/
        break;
default:
        break;
}
```

```
    return err;      /*返回获取参数*/
}
```

7.9.5 文件锁

在第 5 章介绍过，进程间通过信号量来实现同步，以控制对共享资源的访问。文件毫无疑问是共享资源，内核专门为文件设计了同步技术，即文件锁。若要支持文件锁，需选择 FILE_LOCKING（默认选择）配置选项，文件锁相关代码位于/fs/locks.c 文件内。

1 数据结构

内核根文件系统中的文件由 inode 实例唯一表示，进程每次打开文件时，都会为其创建 file 实例。同一个文件可以被多个进程打开，因此 file 实例与 inode 实例，是多对一的关系。

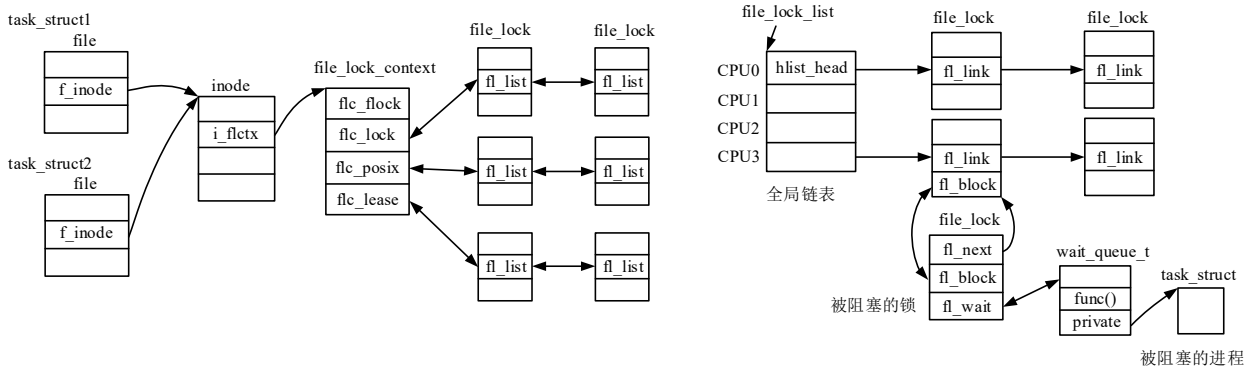
文件锁毫无疑问应关联到公共的唯一的 inode 实例，inode 结构体中与文件锁相关成员如下：

```
inode{
    ...
    struct file_lock_context  *i_flctx;
    ...
}
```

i_flctx 成员指向 file_lock_context 结构体，管理着所有加在文件上的锁，定义如下（/include/linux/fs.h）：

```
struct file_lock_context {
    spinlock_t          flc_lock;       /*保护后面三个双链表的自旋锁*/
    struct list_head    flc_flock;      /*flock 锁*/
    struct list_head    flc_posix;      /*POSIX 锁*/
    struct list_head    flc_lease;      /*租借锁（lease）*/
};
```

file_lock_context 结构体中包含三个双链表，分别管理着三种加在文件上的锁，分别是 flock 锁，POSIX 锁和租借锁（lease 锁），如下图所示。



以上三种锁都由 file_lock 结构体表示，file_lock 实例还添加到全局列表，用于向用户空间输出文件锁信息（/proc/locks）。每钟锁又分为读锁、写锁等类型。一般读锁与读锁可共存，写锁与写锁/读锁是互斥的。

flock 锁是对整体个文件加的建议锁（劝告锁），多个读锁可以同时加在文件上，但是写锁与读锁和其它写锁是互斥的。进程不能加锁时将阻塞，进入睡眠等待（阻塞，非阻塞时返回错误码），直到锁解操作除将其唤醒。建议锁需要操作文件的各进程在读写文件前持有锁，操作完成后释放锁，在 read()和 write()系统调用中不会检查锁。

POSIX 锁是可对文件一个区域加锁的建议锁（默认情况下），但是也可以设置文件上的 POSIX 锁为强制锁，即在 read()和 write()系统调用中检查锁冲突，有冲突则睡眠或返回错误码，无冲突才能继续操作。

租借锁（lease）是对整个文件施加的强制锁，在打开文件或修改文件元信息等时机将检查锁。租借锁与其它两种锁最大的区别是，在操作因锁冲突阻塞时设置了一个期限（45 秒后），并向持锁进程发送信号，如果在期限到期时持锁进程没有降级或解除锁，将强制将当前锁降为读锁或解除，以保证文件操作的正常进行，避免长时间睡眠等待。

关闭文件时，close()系统调用中会释放加在文件上的锁。

file_lock 结构体用于表示三种锁，结构体定义如下（/include/linux/fs.h）：

```
struct file_lock {
    struct file_lock *fl_next;      /*指向阻塞本锁的冲突锁*/
    struct list_head   fl_list;     /*将实例添加到 file_lock_context 中双链表*/
    struct hlist_node   fl_link;     /*添加到 CPU 核的全局链表中（仅用于向用户空间输出信息）*/
    struct list_head   fl_block;   /*管理被本锁阻塞的锁*/
    fl_owner_t fl_owner;       /*指向 files_struct 实例*/
    unsigned int fl_flags;    /*锁标记*/
    unsigned char fl_type;   /*锁类型*/
    unsigned int fl_pid;      /*创建锁的进程的线程组 ID（tgid）*/
    int fl_link_cpu;           /* what cpu's list is this on? */
    struct pid *fl_nspid;
    wait_queue_head_t fl_wait;       /*睡眠等待队列头，因加本锁而阻塞的进程*/
    struct file *fl_file;       /*指向加锁的 file 实例*/
    loff_t fl_start;            /*加锁文件内容起始位置*/
    loff_t fl_end;             /*加锁文件内容结束位置*/

    struct fasync_struct *    fl_fasync;      /*用来向持有租借锁的进程发送信号*/
    /* for lease breaks: */
    unsigned long fl_break_time;            /*租借锁解除时限*/
    unsigned long fl_downgrade_time;      /*租借锁降级时限（改为读锁）*/

    const struct file_lock_operations *fl_ops;            /* Callbacks for filesystems */
    const struct lock_manager_operations *fl_lmops;      /*适用于租借锁，锁操作结构*/
    union {
        struct nfs_lock_info      nfs_fl;
        struct nfs4_lock_info     nfs4_fl;
        struct {
            struct list_head link;      /* link in AFS vnode's pending_locks list */
            int state;         /* state of grant or error if -ve */
        } afs;
    } fl_u;
};
```

file_lock 结构体中主要成员简介如下：

●**fl_type**：锁的类型，取值定义如下（/include/uapi/asm-generic/fcntl.h）：

```
#ifndef  F_RDLCK
  #define  F_RDLCK      0      /*读锁，对应 LOCK_SH 共享锁*/
  #define  F_WRLCK      1      /*写锁，对应 LOCK_EX 互斥锁*/
  #define  F_UNLCK      2      /*解锁文件，对应 LOCK_UN*/
#endif

#define  LOCK_MAND    32  /*强制锁*/
#define  LOCK_READ    64  /*允许同时读操作*/
#define  LOCK_WRITE   128 /*允许同时写操作*/
#define  LOCK_RW      192 /*允许同时读写操作，即 LOCK_READ|LOCK_WRITE*/
```

●**fl_flags**：锁标记，标记锁的种类等，取值定义如下（/include/linux/fs.h）：

```
#define  FL_POSIX          1   /*POSIX 锁*/
#define  FL_FLOCK          2   /*flock 锁*/
#define  FL_DELEG          4    /* NFSv4 delegation */
#define  FL_ACCESS         8    /*只查看锁，不获取锁*/
#define  FL_EXISTS         16   /*解锁时，查看锁是否存在*/
#define  FL_LEASE          32   /*租借锁*/
#define  FL_CLOSE          64   /* unlock on close */
#define  FL_SLEEP          128  /*锁操作是阻塞的（不成功睡眠）*/
#define  FL_DOWNGRADE_PENDING 256   /*租借锁正在等待被降级*/
#define  FL_UNLOCK_PENDING    512   /*租借锁正在等待被解锁*/
#define  FL_OFDLCK            1024  /* lock is "owned" by struct file */
#define  FL_LAYOUT            2048  /* outstanding pNFS layout */
```

2 flock 锁

flock()系统调用用于对整个文件放置或释放 flock 锁，实现函数如下（/fs/locks.c）：

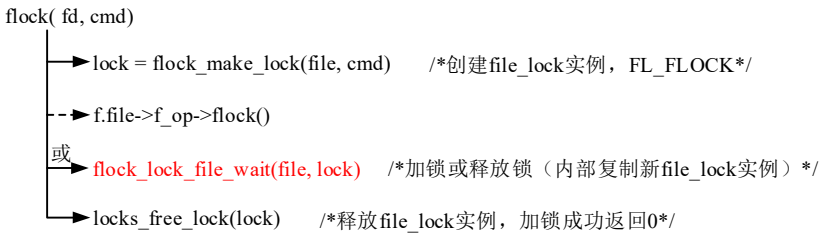
```
SYSCALL_DEFINE2(flock, unsigned int, fd, unsigned int, cmd)
{
    ...      /*实现代码略*/
}
```

fd 参数为欲执行锁操作的文件描述符，cmd 参数取值定义在/include/uapi/asm-generic/fcntl.h 头文件，表示锁操作：

```
/*cmd 参数可取 LOCK_SH、LOCK_EX、LOCK_UN 之一（可并上 LOCK_NB）*/
#define  LOCK_SH      1    /*加共享锁，锁类型为 F_RDLCK*/
#define  LOCK_EX      2    /*加互斥锁，锁类型为 F_WRLCK*/
#define  LOCK_NB      4    /*非阻塞锁操作（不睡眠）*/
#define  LOCK_UN      8    /*释放锁，锁类型设为 F_UNLCK*/

/*以下是强制锁标记，cmd 参数可取 LOCK_MAND 与其它几个标记的并*/
#define  LOCK_MAND    32  /*强制锁，锁类型采用相同的标记，下同*/
#define  LOCK_READ    64  /*允许同时读操作*/
```

```
#define   LOCK_WRITE     128  /*允许同时写操作*/
#define   LOCK_RW        192  /*允许同时读写操作，即 LOCK_READ|LOCK_WRITE*/
```

flock()系统调用实现函数调用关系如下图所示：



flock()系统调用首先会创建一个 flie_lock 实例，锁标记设为**FL_FLOCK**，然后调用 flock_lock_file_wait()函数执行锁操作，最后释放 flie_lock 实例，系统调用返回。若加锁/解锁操作成功返回 0，阻塞加锁不成功将睡眠，直到加锁成功，非阻塞加锁不成功返回-EAGAIN。解锁操作总是会成功，返回 0。

进程只能对同一个 file 实例关联的文件（inode 实例）最多加一把 flock 锁。

下面主要介绍一下 flock_lock_file_wait()函数的实现，如下图所示，此函数内将执行以下操作：

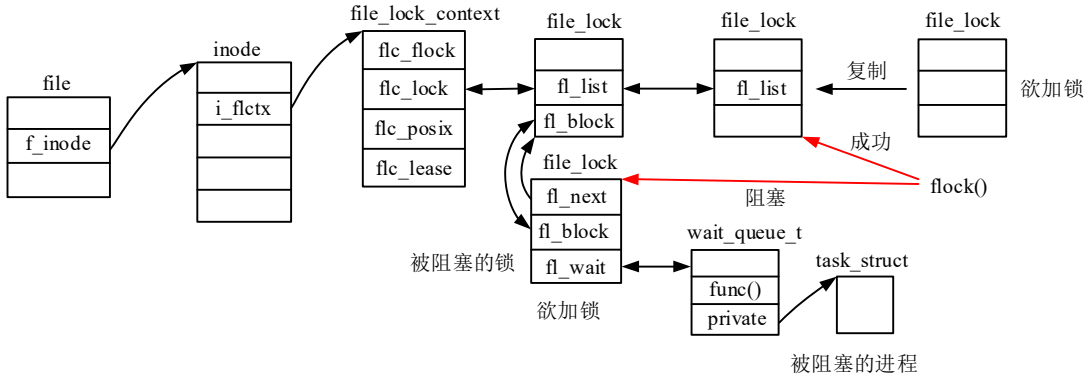
（1）创建一个新 flie_lock 实例（解锁不创建）。

（2）依欲加锁信息，在 file_lock_context.flc_flock 双链表中查找同一个 file 实例添加的锁。

如果是解锁，则将 file 加的锁取出释放（还要唤醒阻塞的进程），函数返回。如果是加锁，且存在同类型的锁，则返回，不需要再加锁了。若是不同类型，则取出原锁释放，执行步骤（3）。

（3）若是加锁操作，再遍历 file_lock_context.flc_flock 双链表中锁，看是否有冲突的锁已经存在，若存在与欲加锁冲突的锁，且是阻塞加锁，则将欲加锁添加到此冲突锁的 fl_block 双链表，其 fl_next 成员指向冲突锁，进程在欲加锁上睡眠等待，冲突锁解除时唤醒。若存在冲突锁，且是非阻塞，直接返回-EAGAIN。

若没有冲突的锁，则将欲加锁复制到（1）中创建的新锁，将其添加到 file_lock_context.flc_flock 双链表末尾和 CPU 核的散列链表（下图中未画出），加锁成功，返回 0。



在第（2）步中的释放锁（解锁）操作中，将遍历其 fl_block 双链表，将每个实例从双链表中移出，对其 fl_next 成员赋 NULL，并唤醒在锁上睡眠的进程。加锁阻塞的睡眠进程唤醒后，将又从第（1）步开始执行，直到加锁成功。

flock_lock_file_wait()函数定义在/include/linux/fs.h 头文件，代码如下：

```
static inline int flock_lock_file_wait(struct file *filp, struct file_lock *fl)
/*fl：指向欲加锁*/
{
    return flock_lock_inode_wait(file_inode(filp), fl);          /*/fs/locks.c*/
}
```

flock_lock_inode_wait()函数定义如下：

```
int flock_lock_inode_wait(struct inode *inode, struct file_lock *fl)
{
    int error;
    might_sleep();
    for (;;) {
        error = flock_lock_inode(inode, fl); /*执行锁操作，成功返回 0，非阻塞不成功返回-EAGAIN*/
        if (error != FILE_LOCK_DEFERRED)      /*阻塞，不成功返回 FILE_LOCK_DEFERRED*/
            break;                                    /*返回 0 或-EAGAIN，则跳出循环*/
        /*阻塞加锁不成功则进入睡眠*/
        error = wait_event_interruptible(fl->fl_wait, !fl->fl_next);
                                                  /*在欲加锁上睡眠等待，直到 fl->fl_next 为空*/
        if (!error)            /*被冲突锁唤醒，继续循环*/
            continue;

        locks_delete_block(fl);     /*由信号唤醒唤醒，将欲加锁从冲突锁的链表中移出，跳出*/
        break;
    }
    return error;
}
```

flock_lock_inode()函数定义如下（/fs/locks.c）：

```
static int flock_lock_inode(struct inode *inode, struct file_lock *request)
{
    struct file_lock *new_fl = NULL;
    struct file_lock *fl;
    struct file_lock_context *ctx;
    int error = 0;
    bool found = false;
    LIST_HEAD(dispose);

    ctx = locks_get_lock_context(inode, request->fl_type);      /*返回 inode->i_flctx，为空则创建*/
    ...    /*错误处理*/

    if (!(request->fl_flags & FL_ACCESS) && (request->fl_type != F_UNLCK)) {
        new_fl = locks_alloc_lock();          /*分配新 file_lock 实例并初始化，非解锁*/
        ...    /*错误处理*/
    }

    spin_lock(&ctx->flc_lock);
    if (request->fl_flags & FL_ACCESS)
        goto find_conflict;
```

```
    list_for_each_entry(fl, &ctx->flc_flock, fl_list) {
                                    /*搜索 flock 锁双链表，检查同一 file 对 inode 添加的锁*/
        if (request->fl_file != fl->fl_file)
            continue;
        if (request->fl_type == fl->fl_type)        /*已有同类型锁，跳至 out 处，返回*/
            goto out;
        found = true;                       /*有同 file 添加的不同类型的锁，取出释放它，解锁也适用*/
        locks_delete_lock_ctx(fl, &dispose);   /*原锁添加到 dispose 双链表（要解的锁会取出）*/
        break;
    }

    if (request->fl_type == F_UNLCK) {        /*解锁*/
        if ((request->fl_flags & FL_EXISTS) && !found)
            error = -ENOENT;
        goto out;
    }

find_conflict:          /*检查冲突锁*/
    list_for_each_entry(fl, &ctx->flc_flock, fl_list) {      /*搜索 flock 锁双链表*/
        if (!flock_locks_conflict(request, fl))      /*只要有一个是写锁（LOCK_EX），就会冲突*/
            continue;
        error = -EAGAIN;                            /*以下是有冲突锁的情形*/
        if (!(request->fl_flags & FL_SLEEP))        /*有冲突，非阻塞，跳转到 out，返回-EAGAIN*/
            goto out;
        error = FILE_LOCK_DEFERRED;           /*阻塞，返回 FILE_LOCK_DEFERRED*/
        locks_insert_block(fl, request);
                    /*被冲突锁阻塞，request 添加到 fl 中 fl_block 双链表，fl_next 指向冲突锁*/
        goto out;
    }
    /*以下是没有冲突的情形，执行加锁*/
    if (request->fl_flags & FL_ACCESS)
        goto out;
    locks_copy_lock(new_fl, request);           /*复制 request 至 new_fl*/
    locks_insert_lock_ctx(new_fl, &ctx->flc_flock);
                        /*new_fl 添加到 ctx->flc_flock 双链表末尾，并添加到 CPU 核的散列链表*/
    new_fl = NULL;
    error = 0;          /*返回值 0*/

out:
    spin_unlock(&ctx->flc_lock);
    if (new_fl)
        locks_free_lock(new_fl);        /*释放新分配的锁，如果需要*/
    locks_dispose_list(&dispose);         /*释放旧锁*/
```

```
    return error;
    /*加锁成功返回 0，有冲突非阻塞返回-EAGAIN，有冲突阻塞返回 FILE_LOCK_DEFERRED*/
}
```

前面已经介绍了 flock()系统调用的主要工作，上面代码中也添加了注释，就不再解释了。这里需要说明一下的是如何判断两个锁存在冲突，即 flock_locks_conflict()函数执行的工作，若两个锁有冲突，返回 1，否则返回 0，函数代码如下。

```
static int flock_locks_conflict(struct file_lock *caller_fl, struct file_lock *sys_fl)
/*检查 sys_fl 是否阻塞 caller_fl*/
{
    if (caller_fl->fl_file == sys_fl->fl_file)          /*同 file，不阻塞*/
        return (0);
    if ((caller_fl->fl_type & LOCK_MAND) || (sys_fl->fl_type & LOCK_MAND))    /*强制锁不阻塞*/
        return 0;

    return (locks_conflict(caller_fl, sys_fl));
}
```

locks_conflict()函数定义如下：

```
static int locks_conflict(struct file_lock *caller_fl, struct file_lock *sys_fl)
{
    if (sys_fl->fl_type == F_WRLCK)       /*有一个锁是写锁，冲突，否则不冲突*/
        return 1;
    if (caller_fl->fl_type == F_WRLCK)
        return 1;
    return 0;
}
```

由上面代码可知，其它 file 的写锁会阻塞欲加锁（不管是读锁还是写锁），欲加锁是写锁时会被其它 file 的任何锁阻塞（强制锁不被阻塞，也不会阻塞其它锁）。

3 POSIX 锁

POSIX 锁和租借锁由 fcntl()系统调用操作，下面先介绍 POSIX 锁，再介绍租借锁。

■加锁

POSIX 锁用于对文件一个区域加锁，一个文件中可以有多个 POSIX 锁，但是各个锁的区域不能重叠。多个进程可以同时持有同一区域的读锁，但是只有一个进程可以持有写锁，写锁与其它写锁和读锁是互斥的。

fcntl()系统调用中 POSIX 锁操作相关的命令有（/include/uapi/asm-generic/fcntl.h）：

```
#define   F_GETLK     5       /*检测是否可加锁*/
#define   F_SETLK     6       /*加锁/解锁，非阻塞*/
#define   F_SETLKW    7       /*加锁/解锁，可阻塞*/
```

fcntl()系统调用中相关命令处理函数如下（fcntl()->do_fcntl()）：

```
static long do_fcntl(int fd, unsigned int cmd, unsigned long arg,struct file *filp)
/*cmd：命令，arg：flock 实例地址*/
{
    ...
    switch (cmd) {
    ...
    case F_GETLK:
        err = fcntl_getlk(filp, cmd, (struct flock __user *) arg); /*检测是否可加 POSIX 锁，/fs/locks.c*/
        break;
#if BITS_PER_LONG != 32
    case F_OFD_SETLK:
    case F_OFD_SETLKW:
#endif
    case F_SETLK:
    case F_SETLKW:
        err = fcntl_setlk(fd, filp, cmd, (struct flock __user *) arg);    /*加/解 POSIX 锁*/
        break;
        ...
    }
    return err;      /*返回获取参数*/
}
```

用户进程通过 flock 结构体与 fcntl()系统调用传递锁信息，定义如下（/include/uapi/asm-generic/fcntl.h）：

```
struct flock {
    short      l_type;                /*锁的类型*/
    short      l_whence;              /*解释 l_start 是相对于哪的偏移量*/
    __kernel_off_t   l_start;         /*加锁文件内容起始偏移量*/
    __kernel_off_t   l_len;           /*加锁文件内容长度*/
    __kernel_pid_t      l_pid;       /*阻塞锁进程的 PID*/
    __ARCH_FLOCK_PAD
};
```

flock 结构体中成员简介如下：

●**l_type**：锁的类型，取值必须是以下三者之一：

```
#define   F_RDLCK       0       /*读锁*/
#define   F_WRLCK       1       /*写锁*/
#define   F_UNLCK       2       /*解锁*/
```

●**l_whence**：表示 l_start 成员是相对于哪个位置的偏移量（/include/uapi/linux/fs.h）：

```
#define   SEEK_SET      0    /*相对文件开始*/
#define   SEEK_CUR      1    /*相对文件当前位置*/
#define   SEEK_END      2    /*相对文件结尾*/
```

l_whence 为 SEEK_SET，l_start 和 l_len 都为 0 表示锁住整个文件。l_len 可以是负值，表示对 l_start 之前的内容加锁。

cmd 命令参数各值语义如下：

●**F_GETLK**：检测是否能够获 arg 参数取指定区域上的锁，但实际上不加锁。l_type 可以是 F_RDLCK 或 F_WRLCK。返回时，arg 指定 flock 实例保存了有关是否能够放置指定锁的信息。如果允许加锁，那么 l_type 字段值为 F_UNLCK，并且剩余字段会保持不变。如果在不能在指定区域加锁，在 flock 实例中返回一把已加锁的信息。只有欲加锁与现有锁有冲突时，才不能加锁，读锁与读锁不互斥，加锁后会合并。

●**F_SETLK**：非阻塞加锁或释放锁。如果另一个进程持有了一把待加锁区域中任意一部分上的不兼容锁，将返回-EAGAIN，否则返回 0。解锁总是会成功，不会阻塞。

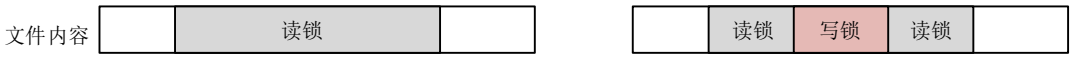
●**F_SETLKW**：阻塞加锁或释放锁。加锁不成功会睡眠，解锁总会成功。

fcntl_setlk()函数执行的操作与 flock()系统调用中执行的操作类似，只不过 POSIX 锁 file_lock 实例添加到 file_lock_context.flc_posix 双链表，且以加锁区域起始位置从小到大，在双链表中从左至右排列。另外还有一些特性需要说明一下：

●多个 file 实例可以在同一个区域加多个读锁，读锁与读锁可以共存。写锁是独占的，与其它读锁和写锁互斥。如果欲加锁与写锁持有区域有重叠，欲加锁操作（不管是读锁还是写锁）不成功（非阻塞）或睡眠等待（阻塞）。如果欲加锁是写锁，且与读锁重叠，欲加锁也将不成功或阻塞。

●同一个 file 对同一个区域只能持有一把同类型的锁，若欲加锁的区域与现有同类型的锁持有区域重叠，两个锁将合并，持锁范围也将合并。

●同一个 file 若加的锁与同一个 file 持有的另一类型的锁有重叠，将会对各锁持有区域进行调整，以消除重叠。如下图所示，在一个读锁持有区域内加一把写锁，将拆分成三把锁，依次是读锁、写锁和读锁。



fcntl_setlk()函数执行解锁操作时，若解锁的区域不是某个锁持有的所有区域，也将对区域进行拆分，请读者自行阅读源代码。

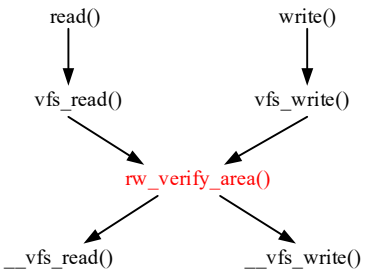
■强制检查锁

到目前为止，介绍的 flock 锁和 POSIX 锁都是建议锁，就是需要依靠进程之间的“君子约定”，在读写文件之前对文件加锁，加锁成功则执行读写操作（持锁操作），操作后释放锁，否则睡眠等待或放弃本次操作。

在 read()/write()系统调用中默认不会去检查当前操作是否与某个文件锁冲突，也就是说 read()/write()系统调用对文件锁是视而不见的。进程可以不检查锁，直接对文件进行读写。

但是，Linux 内核也提供了在 read()/write()系统调用中强制检查 POSIX 锁冲突的机制，若本次操作与文件现有锁冲突就睡眠或放弃操作（flock 锁始终是建议锁）。

read()/write()系统调用中检查 POSIX 锁冲突的函数调用关系如下：



rw_verify_area()函数内可能会强制检查 POSIX 锁冲突，函数定义如下（/fs/read_write.c）：

```
int rw_verify_area(int read_write, struct file *file, const loff_t *ppos, size_t count)
{
    struct inode *inode;
    loff_t pos;
    int retval = -EINVAL;

    inode = file_inode(file);
    ...
    if (unlikely(inode->i_flctx && mandatory_lock(inode))) {          /*检查 POSIX 锁冲突的条件*/
        retval = locks_mandatory_area(
                read_write == READ ? FLOCK_VERIFY_READ : FLOCK_VERIFY_WRITE,
                inode, file, pos, count);              /*检查锁冲突*/
        if (retval < 0)          /*有冲突，不能进行读写操作*/
            return retval;
    }
    ...
}
```

rw_verify_area()函数中判断是否要检查 POSIX 锁冲突，如果需要则执行检查。

●强制检查锁条件

强制执行 POSIX 锁冲突检查的条件是 inode->i_flctx 不为 NULL，且 mandatory_lock()函数返回真。

mandatory_lock()函数定义在/include/linux/fs.h 头文件：

```
static inline int mandatory_lock(struct inode *ino)
{
    return IS_MANDLOCK(ino) && __mandatory_lock(ino);      /*/include/linux/fs.h*/
}
```

IS_MANDLOCK()宏用于检查文件系统超级块是否设置某个挂载标记，如下所示：

```
#define   IS_MANDLOCK(inode)      __IS_FLG(inode, MS_MANDLOCK)
#define   __IS_FLG(inode, flg)    ((inode)->i_sb->s_flags & (flg))
```

由此可知，强制检查锁冲突需要超级块具有 MS_MANDLOCK 挂载标记（挂载操作时设置）。

__mandatory_lock()函数定义如下：

```
static inline int __mandatory_lock(struct inode *ino)
{
    return (ino->i_mode & (S_ISGID | S_IXGRP)) == S_ISGID;
```

}

强制检查锁冲突还需要文件设置 S_ISGID 标记位并关闭 S_IXGRP 标记位。

用户进程可通过 chmod()等系统调用来处理这两个标记位，以要求对文件访问时强制检查锁冲突。

在 fcntl_setlk()函数中，不能对需要强制检查 POSIX 锁冲突，且被映射为共享可写的文件加 POSIX 锁，请读者自行阅读源代码。

●检查锁冲突

locks_mandatory_area()函数用于检查本次操作是否与文件现有的 POSIX 锁冲突，没有冲突返回 0，有冲突则睡眠或返回错误码，函数定义如下（/fs/locks.c）：

```
int locks_mandatory_area(int read_write, struct inode *inode,struct file *filp, loff_t offset,size_t count)
{
    struct file_lock fl;
    int error;
    bool sleep = false;

    locks_init_lock(&fl);        /*初始化 file_lock 实例*/
    fl.fl_pid = current->tgid;
    fl.fl_file = filp;
    fl.fl_flags = FL_POSIX | FL_ACCESS;          /*只检查锁冲突，不加锁*/
    if (filp && !(filp->f_flags & O_NONBLOCK))
        sleep = true;                    /*是否睡眠*/
    fl.fl_type = (read_write == FLOCK_VERIFY_WRITE) ? F_WRLCK : F_RDLCK;
    fl.fl_start = offset;
    fl.fl_end = offset + count - 1;

    for (;;) {
        if (filp) {
            fl.fl_owner = filp;
            fl.fl_flags &= ~FL_SLEEP;
            error = __posix_lock_file(inode, &fl, NULL);        /*检查 POSIX 锁冲突*/
            if (!error)          /*加锁成功跳出循环*/
                break;
        }

        if (sleep)
            fl.fl_flags |= FL_SLEEP;
        fl.fl_owner = current->files;
        error = __posix_lock_file(inode, &fl, NULL);
        if (error != FILE_LOCK_DEFERRED)          /*要睡眠等待*/
            break;
        error = wait_event_interruptible(fl.fl_wait, !fl.fl_next);
        if (!error) {
            if (__mandatory_lock(inode))
```

```
                continue;           /*被冲突锁唤醒，继续检查是否有冲突*/
            }

            locks_delete_block(&fl);        /*被信号唤醒*/
            break;
        }       /*for 循环结束*/

        return error;
    }
```

locks_mandatory_area()函数与前面介绍的 flock_lock_inode_wait()函数很相似，不过这里不会对文件加锁，没有锁冲突函数就返回，执行后面的读写操作。若有冲突且是阻塞访问，则睡眠等待，由冲突锁唤醒，唤醒后再检查锁冲突，直至没有冲突了，函数返回（可由信号中断）。若有冲突且是非阻塞访问，则返回错误码。

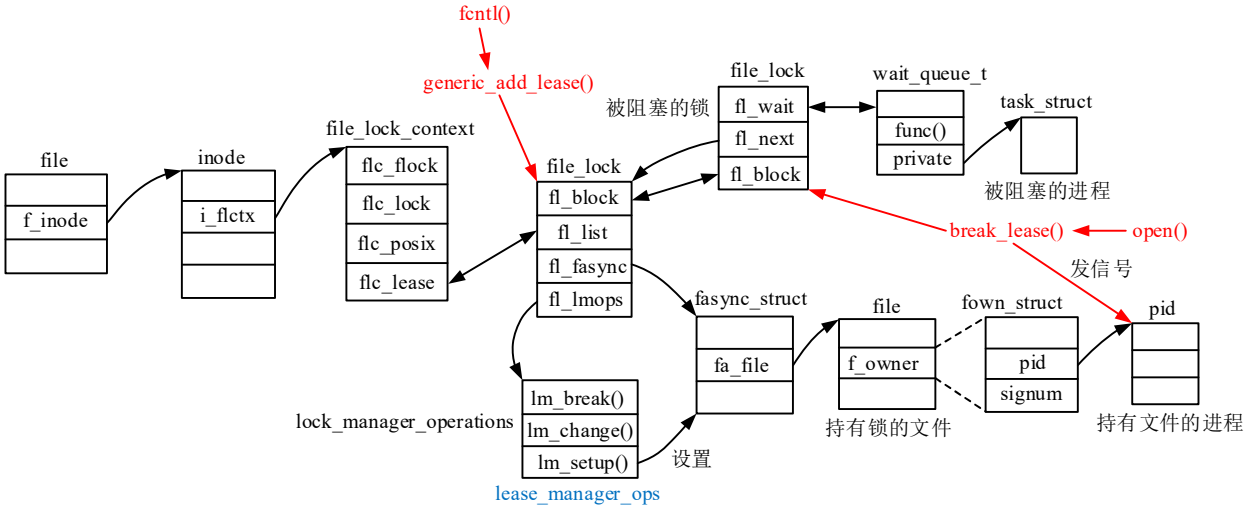
4 租借锁

采用强制锁之后，如果一个进程对某个文件拥有写锁，只要它不释放这个锁，就会导致访问该文件的其他进程全部被阻塞或不断失败重试；即使该进程持有读锁，也会造成后续更新该文件（写）进程的阻塞。为了解决这个问题，Linux 中采用了一种新型的租借锁（lease 锁）。

简单地说，进程打开加锁文件或修改加锁文件属性等时机，会强制进行锁检查，如果有冲突当前进程睡眠，且会向持锁进程发送信号，持锁进程收到信号后（同步文件内容后）可以把锁降为读锁或解锁。如果持锁进程不这么做，内核会在指定时长（45 秒）之后，强制降级（写锁降为读锁）或解除文件当前锁，以使当前进程能正常地执行操作。

进程对文件施加租借锁时，成功返回 0（读者可以同时持锁，写者需排它地持有锁），不成功返回-EAGAIN，不会睡眠。租借锁只能对整个文件生效，不能对文件区域加锁，并且只能对普通文件施加租借锁。租借锁是强制锁。

租借锁实现框架如下：



租借锁（lease）与 POSIX 锁一样，通过 fcntl()系统调用执行加/解锁操作。加锁操作将会创建 file_lock 实例，并添加到 ctx->flc_lease 双链表，其 fl_lmops 成员指向 lease_manager_ops 实例，其中函数用于设置新加的租借锁，修改锁类型和向持锁进程发送信号等。

在 open()和修改文件属性等系统调用中（如 chmod()），将检测本次操作是否与文件现有的租借锁冲突（只检测冲突，不加锁），没有冲突则继续操作。若有冲突，则需要设置所有冲突锁为待降级或待解除，

并设置限期（45 秒后），向持锁进程发送信号，然后依次在冲突锁上睡眠等待，等待冲突锁降级或解除时将其唤醒，等到没有冲突锁时，进程唤醒后继续执行后续操作。如果持有冲突锁的进程不降级或解除锁，则在期限到了之后，内核会强制降级锁或解锁。

如果是非阻塞操作，执行设置期限，发送信号操作后，不睡眠等待，直接返回错误码。

fcntl()系统调用中与租借锁相关的命令如下（/include/uapi/linux/fcntl.h）：

```
#define   F_SETLEASE      (F_LINUX_SPECIFIC_BASE + 0)          /*加锁/解锁*/
#define   F_GETLEASE      (F_LINUX_SPECIFIC_BASE + 1)          /*查询 file 实例已添加锁的类型*/
```

fcntl()系统调用中与租借锁处理相关函数如下：

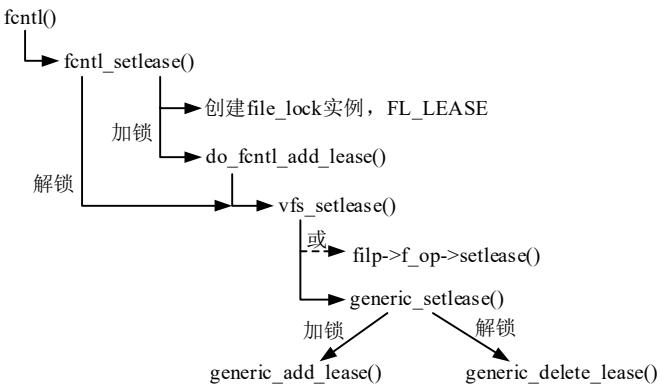
```
static long do_fcntl(int fd, unsigned int cmd, unsigned long arg,struct file *filp)
/*cmd：命令，arg：锁类型（读写锁或解锁）*/
{
    ...
    switch (cmd) {
    ...
    case F_GETLEASE:
        err = fcntl_getlease(filp); /*查询 file 实例添加的租借锁类型，没加锁为 F_UNLCK，/fs/locks.c*/
        break;
    case F_SETLEASE:
        err = fcntl_setlease(fd, filp, arg);      /*加/解租借锁，/fs/locks.c*/
        break;
    ...
    }
    return err;      /*返回获取参数*/
}
```

F_GETLEASE 命令由 fcntl_getlease()函数处理，用于查询通过 file 实例添加的租借锁类型，没有加锁返回 F_UNLCK，函数源代码请读者自行阅读。

F_SETLEASE 命令由 fcntl_setlease()函数处理，用于加/解租借锁，arg 参数表示锁的类型，取值如下：

```
#define   F_RDLCK         0       /*读锁*/
#define   F_WRLCK         1       /*写锁*/
#define   F_UNLCK         2       /*解锁*/
```

fcntl()系统调用中与租借锁处理相关函数调用关系如下：



generic_setlease()函数定义如下（/fs/locks.c）：

```
int generic_setlease(struct file *filp, long arg, struct file_lock **flp,void **priv)
/* *flp：指向新创建的租借锁，*pirv 指向 fasync_struct 实例*/
{
    struct dentry *dentry = filp->f_path.dentry;
    struct inode *inode = dentry->d_inode;
    int error;

    if ((!uid_eq(current_fsuid(), inode->i_uid)) && !capable(CAP_LEASE))        /*权限检查*/
        return -EACCES;
    if (!S_ISREG(inode->i_mode))
        return -EINVAL;
    error = security_file_lock(filp, arg);
    ...
    switch (arg) {
    case F_UNLCK:
        return generic_delete_lease(filp, *priv);            /*解锁*/
    case F_RDLCK:
    case F_WRLCK:
        if (!(*flp)->fl_lmops->lm_break) {          /*lm_break()函数用于向加锁进程发送信号*/
            WARN_ON_ONCE(1);
            return -ENOLCK;
        }

        return generic_add_lease(filp, arg, flp, priv);        /*加锁*/
    default:
        return -EINVAL;
    }
}
```

generic_setlease()函数内要对进程权限进行检查，非特权进程只能对自身拥有的文件进程加/解租借锁，具有 CAP_LEASE 能力的特权进程可对任意文件加/解租借锁。

generic_setlease()函数中*priv 参数指向 fasync_struct 实例（动态分配），定义如下（/include/linux/fs.h）：

```
struct fasync_struct {            /*用于向加锁进程发送信号*/
    spinlock_t      fa_lock;
```

```
    int             magic;
    int             fa_fd;          /*文件描述符*/
    struct fasync_struct  *fa_next;      /* singly linked list */
    struct file     *fa_file;           /*file，关联到文件属主进程*/
    struct rcu_head     fa_rcu;     /*回调函数*/
};
```

generic_setlease()函数中调用 generic_add_lease()函数执行加锁操作，generic_delete_lease()函数用于解锁操作。

下面将介绍加锁 generic_add_lease()函数的实现，解锁 generic_delete_lease()函数比较简单请读者自行阅读。

■加锁

加锁操作 generic_add_lease()函数定义如下（/fs/locks.c）：

```
static int generic_add_lease(struct file *filp, long arg, struct file_lock **flp, void **priv)
/* *flp：指向新创建的 lease 锁，*pirv 指向 fasync_struct 实例*/
{
    struct file_lock *fl, *my_fl = NULL, *lease;
    struct dentry *dentry = filp->f_path.dentry;
    struct inode *inode = dentry->d_inode;
    struct file_lock_context *ctx;
    bool is_deleg = (*flp)->fl_flags & FL_DELEG;
    int error;
    LIST_HEAD(dispose);         /*需要释放的锁，添加到此双链表*/
    lease = *flp;           /*指向新锁*/
    trace_generic_add_lease(inode, lease);

    ctx = locks_get_lock_context(inode, arg);           /*指向 file_lock_context 实例*/
    ...

    if (is_deleg && !mutex_trylock(&inode->i_mutex))
        return -EAGAIN;

    if (is_deleg && arg == F_WRLCK) {
        mutex_unlock(&inode->i_mutex);
        WARN_ON_ONCE(1);
        return -EINVAL;
    }

    spin_lock(&ctx->flc_lock);
    time_out_leases(inode, &dispose);
    /*扫描 ctx->flc_lease 双链表中租借锁，执行以下操作：
    *（1）fl_downgrade_time 过期的锁，清除 FL_DOWNGRADE_PENDING 标记，修改（降级）为
    *读锁，唤醒被阻塞进程。
```

```
*（2）fl_break_time 时间过期的锁，清除 FL_UNLOCK_PENDING 标记，唤醒被阻塞进程，
*添加到 dispose 双链表，准备解除。
*/

error = check_conflicting_open(dentry, arg, lease->fl_flags);          /*检查是否可加新锁*/
if (error)              /*不可加锁跳转到 out 处，返回-EAGAIN*/
      goto out;

/*以下是可继续执行加锁操作*/
error = -EAGAIN;
list_for_each_entry(fl, &ctx->flc_lease, fl_list) {          /*遍历租借锁*/
      if (fl->fl_file == filp &&fl->fl_owner == lease->fl_owner) {          /*同一个 file 添加的锁*/
            my_fl = fl;
            continue;
      }

      if (arg == F_WRLCK)          /*inode 已加锁（双链表非空），不能再加写锁了*/
            goto out;

      if (fl->fl_flags & FL_UNLOCK_PENDING)     /*有进程在申请以写操作文件，不能再加锁*/
            goto out;
}

if (my_fl != NULL) {          /*同一个 file 已加了租借锁，修改锁类型（降级或解锁）*/
      lease = my_fl;
      error = lease->fl_lmops->lm_change(lease, arg, &dispose);
                              /*lease_modify()，改变锁类型，唤醒被锁阻塞的进程等*/
      if (error)
            goto out;
      goto out_setup;
}

error = -EINVAL;
if (!leases_enable)           /*全局变量，值为 1*/
      goto out;

locks_insert_lock_ctx(lease, &ctx->flc_lease);          /*添加锁*/
smp_mb();
error = check_conflicting_open(dentry, arg, lease->fl_flags);          /*再次检查是否可加锁*/
if (error) {
      locks_unlink_lock_ctx(lease);
      goto out;
}
```

```
out_setup:          /*加锁成功*/
        if (lease->fl_lmops->lm_setup)
            lease->fl_lmops->lm_setup(lease, priv);   /*关联并设置 fasync_struct 实例，用于发送信号*/
out:
        spin_unlock(&ctx->flc_lock);
        locks_dispose_list(&dispose);          /*释放 dispose 双链表中锁*/
        if (is_deleg)
            mutex_unlock(&inode->i_mutex);
        if (!error && !my_fl)
            *flp = NULL;
        return error;          /*加锁成功返回 0，否则返回-EAGAIN*/
    }
```

generic_add_lease()函数若加锁成功返回 0，若欲加锁与 inode 现有锁冲突将返回-EAGAIN，不睡眠，另外，generic_add_lease()函数还可以用来修改进程通过 file 向 inode 添加的锁，即修改锁类型（降级）。

time_out_leases()函数用于处理降级、解锁期限已过的锁，对它们进行降级或解锁，唤醒被锁阻塞的进程。

下面主要看一下检测是否可对 inode 加新锁的 check_conflicting_open()函数，定义如下：

```
static int check_conflicting_open(const struct dentry *dentry, const long arg, int flags)
{
    int ret = 0;
    struct inode *inode = dentry->d_inode;

    if (flags & FL_LAYOUT)
        return 0;
    /*加读锁，但是有其它进程对文件写入数据，不能加锁了，因为有写者*/
    if ((arg == F_RDLCK) && (atomic_read(&inode->i_writecount) > 0))
        return -EAGAIN;

    /*加写锁，需是独占地打开文件才能加*/
    if ((arg == F_WRLCK) && ((d_count(dentry) > 1) ||(atomic_read(&inode->i_count) > 1)))
        ret = -EAGAIN;

    return ret;      /*可以加锁，返回 0*/
}
```

check_conflicting_open()函数检测可继续执行加锁操作的条件是（满足其一即可）：

●加读锁，没有进程对文件写入内容。

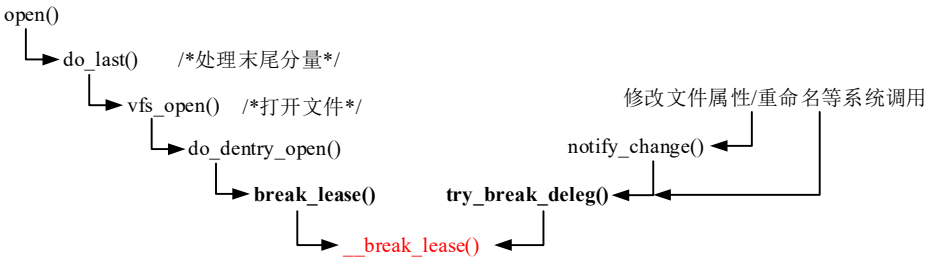
●加写锁，当前进程是独占式地打开文件（没有被其它进程打开）。

■持锁操作

在打开文件，修改文件元信息，重命名等操作中，若文件 inode 中 inode->i_flctx->flc_lease 非空，即有租借锁，则将检查租借锁冲突（能否执行本次操作），有冲突时将向持锁进程发送信号，并设置冲突锁的降级或解除期限。有冲突时，如果进程是以阻塞方式操作文件，则进程会睡眠等待所有冲突锁降级或解

除时将其唤醒，非阻塞操作时，返回错误码-EWOULDBLOCK，函数调用关系如下图所示。没有冲突时，直接返回 0。

持锁进程可以在信号处理函数中降级锁或解锁，唤醒阻塞进程，否则在降级或解除期限到期后，内核会强制将锁降级或解除。



break_lease()函数在打开文件时调用，定义如下（/include/linux/fs.h）：

```
static inline int break_lease(struct inode *inode, unsigned int mode)
{
    smp_mb();
    if (inode->i_flctx && !list_empty_careful(&inode->i_flctx->flc_lease))        /*锁非空*/
        return    __break_lease(inode, mode, FL_LEASE);        /*检查是否有锁冲突，/fs/locks.c*/
    return 0;
}
```

try_break_deleg()函数在修改文件元信息等时机调用，定义如下（/include/linux/fs.h）：

```
static inline int try_break_deleg(struct inode *inode, struct inode **delegated_inode)
{
    int ret;

    ret = break_deleg(inode, O_WRONLY|O_NONBLOCK);      /*只写，非阻塞，/include/linux/fs.h*/
    if (ret == -EWOULDBLOCK && delegated_inode) {
        *delegated_inode = inode;
        ihold(inode);
    }
    return ret;
}

static inline int break_deleg(struct inode *inode, unsigned int mode)
{
    smp_mb();
    if (inode->i_flctx && !list_empty_careful(&inode->i_flctx->flc_lease))      /*锁非空*/
        return    __break_lease(inode, mode, FL_DELEG);
    return 0;
}
```

以上函数都是通过__break_lease()函数执行具体的操作，函数定义如下（/fs/locks.c）：

```
int __break_lease(struct inode *inode, unsigned int mode, unsigned int type)
```

```
/*type：FL_LEASE 或 FL_DELEG*/
{
    int error = 0;
    struct file_lock_context *ctx = inode->i_flctx;
    struct file_lock *new_fl, *fl, *tmp;
    unsigned long break_time;
    int want_write = (mode & O_ACCMODE) != O_RDONLY;       /*是否是写打开*/
    LIST_HEAD(dispose);

    new_fl = lease_alloc(NULL, want_write ? F_WRLCK : F_RDLCK);        /*创建 file_lock 实例*/
    ...
    new_fl->fl_flags = type;       /*FL_LEASE 或 FL_DELEG*/

    ...
    spin_lock(&ctx->flc_lock);
    time_out_leases(inode, &dispose);          /*处理降级、解除期限到期的锁*/

    if (!any_leases_conflict(inode, new_fl))   /*new_fl 是否与现有租借锁有冲突，没有冲突，跳至 out*/
        goto out;

    /*new_fl 为写锁或文件中已有写锁，则有冲突，都是读锁不冲突，以下是有冲突时的情形*/
    break_time = 0;
    if (lease_break_time > 0) {            /*lease_break_time 值为 45，表示 45 秒*/
        break_time = jiffies + lease_break_time * HZ;        /*到期时间为 45 秒后*/
        if (break_time == 0)
            break_time++;
    }
    /*遍历文件的租借锁*/
    list_for_each_entry_safe(fl, tmp, &ctx->flc_lease, fl_list) {
        if (!leases_conflict(fl, new_fl))        /*new_fl 与 fl 没有冲突，遍历下一个*/
            continue;

        /*new_fl 与 fl 有冲突，设置 fl 降级或解锁期限*/
        if (want_write) {        /*本次是写打开*/
            if (fl->fl_flags & FL_UNLOCK_PENDING)        /*fl 正在等待被解除，跳过*/
                continue;
            fl->fl_flags |= FL_UNLOCK_PENDING;        /*fl 待解除*/
            fl->fl_break_time = break_time;                /*期限是 45 秒以后*/
        } else {            /*读打开*/
            if (lease_breaking(fl))        /*fl 正在等待解除或降级，跳过*/
                continue;
            fl->fl_flags |= FL_DOWNGRADE_PENDING;        /*fl 待降级*/
            fl->fl_downgrade_time = break_time;                /*期限是 45 秒以后*/
```

```
            }
            if (fl->fl_lmops->lm_break(fl))          /*发送信号，返回 false*/
                    /*调用 lease_break_callback()函数向持有锁进程发送信号，默认是 SIGIO 信号*/
                locks_delete_lock_ctx(fl, &dispose);        /*不会执行此函数*/
        }

        if (list_empty(&ctx->flc_lease))       /*ctx->flc_lease 为空，跳至 out*/
            goto out;

        if (mode & O_NONBLOCK) {          /*非阻塞操作，返回*/
            trace_break_lease_noblock(inode, new_fl);
            error = -EWOULDBLOCK;
            goto out;
        }

        /*阻塞操作，遍历 ctx->flc_lease 中租借锁，依次在冲突锁中睡眠等待，直到没有冲突锁了*/
    restart:
        fl = list_first_entry(&ctx->flc_lease, struct file_lock, fl_list);        /*ctx->flc_lease 双链表第一个锁*/
        break_time = fl->fl_break_time;          /*解除时限，设置超时时间*/
        if (break_time != 0)
            break_time -= jiffies;
        if (break_time == 0)
            break_time++;
        locks_insert_block(fl, new_fl);        /*new_fl 添加到 fl 阻塞双链表*/
        trace_break_lease_block(inode, new_fl);
        spin_unlock(&ctx->flc_lock);
        locks_dispose_list(&dispose);          /*释放 dispose 双链表中锁*/
        error = wait_event_interruptible_timeout(new_fl->fl_wait,!new_fl->fl_next, break_time);
                                                                    /*带超时的睡眠等待*/
        /*唤醒后执行以下代码*/
        spin_lock(&ctx->flc_lock);
        trace_break_lease_unblock(inode, new_fl);
        locks_delete_block(new_fl);        /*new_fl 从被阻塞双链表移除（没有释放）*/
        if (error >= 0) {
            if (error == 0)          /*降级或解除锁时唤醒*/
                time_out_leases(inode, &dispose);          /*降级/解除到期锁*/
            if (any_leases_conflict(inode, new_fl))          /*是否还有冲突锁，有则继续睡眠*/
                goto restart;
            error = 0;
        }
        /*没有冲突锁了，不加锁*/
    out:
        spin_unlock(&ctx->flc_lock);
```

```
    locks_dispose_list(&dispose);          /*释放 dispose 中锁*/
    locks_free_lock(new_fl);               /*释放 new_fl 锁*/
    return error;
}
```

__break_lease()函数不会向文件加锁，只是检测本次操作是否与现有的租借锁冲突，如果没有冲突，函数返回 0。若有冲突执行以下操作：

（1）若本次操作为写操作，设置所有冲突锁为待解除，解除时限为 45 秒后，并向所有冲突锁的持有者发送信号。若本次操作为读操作（文件加了写锁），设置所有冲突锁为待降级，降级时限为 45 秒后，并向所有冲突锁的持有者发送信号。

（2）若为非阻塞操作，返回错误码。若为阻塞操作，当前进程依次在冲突锁上睡眠等待，冲突锁持有者降级或解除锁时，将其唤醒，或者降级/解除锁时限到了，唤醒当前进程，而后强制降级/解除冲突锁。文件中没有与本次操作冲突的锁后，函数返回 0。

持有冲突锁的进程收到信号后，在信号处理函数中，可以通过 fcntl()系统调用降级锁（改为读锁）或解除锁，这将会唤醒在冲突锁上睡眠的进程。

需要注意的是，默认情况下向加冲突锁进程发送的是 SIGIO 信号，而 SIGIO 信号默认会使进程终止，因此持有锁的进程若不想被终止，必须为 SIGIO 信号设置处理函数。加冲突锁的进程也可以通过系统调用 fcntl(fd,F_SETSIG,sig)修改发送的信号为 sig，而后为其设置处理函数。

__break_lease()函数检查锁冲突的 any_leases_conflict()函数定义如下（/fs/locks.c）：

```
static bool any_leases_conflict(struct inode *inode, struct file_lock *breaker)
/*breaker：__break_lease()函数中新创建的锁，只用于检测锁冲突*/
{
    struct file_lock_context *ctx = inode->i_flctx;
    struct file_lock *fl;

    lockdep_assert_held(&ctx->flc_lock);

    list_for_each_entry(fl, &ctx->flc_lease, fl_list) {          /*遍历租借锁*/
        if (leases_conflict(fl, breaker))       /*检查锁冲突，/fs/locks.c*/
            return true;
    }
    return false;
}

static bool leases_conflict(struct file_lock *lease, struct file_lock *breaker)
{
    if ((breaker->fl_flags & FL_LAYOUT) != (lease->fl_flags & FL_LAYOUT))
        return false;
    if ((breaker->fl_flags & FL_DELEG) && (lease->fl_flags & FL_LEASE))     /*不会与租借锁冲突*/
        return false;
    return locks_conflict(breaker, lease);        /*有一个或两个为写锁时，有冲突*/
}
```

7.9.6 监控文件事件

某些应用程序需要对文件或目录进行监控，以侦测其是否发生了特定事件。例如，当把文件加入或移出一目录时，图形化文件管理器应能判定此目录是否在其当前显示之列，而守护进程可能也想要监控自己的配置文件，以了解其是否被修改。

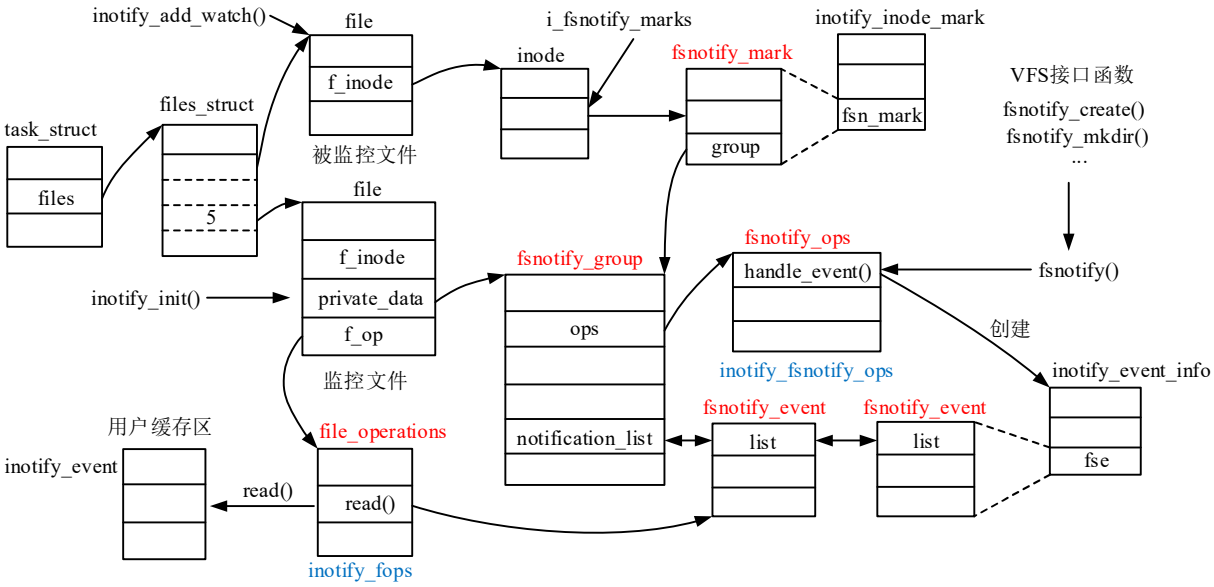
自内核 2.6.13 起，Linux 开始提供 inotify 机制，以允许应用程序监控文件事件。inotify 机制是取代 dnotify 机制的，后者较为陈旧，并且只具备前者的部分功能。inotify 和 dnotify 都是 Linux 专有的机制。Linux 还实现了 fanotify 机制。

内核在/fs/notify/目录下实现了 dnotify、inotify 和 fanotify 机制的代码，要激活这些机制需选择相应的配置选项，如 INOTIFY_USER、FANOTIFY（默认会选择 FSNOTIFY 和 ANON_INODES 选项）。这三个机制这里统称它们为 fsnotify 机制。

下面主要介绍 inotify 和 fanotify 机制的实现。inotify 和 fanotify 机制使用了相同的核心数据结构，这些数据结构的创建等操作在/fs/notify/目录下的文件内实现，inotify 机制实现代码位于/fs/notify/inotify/目录下，fanotify 机制实现代码位于/fs/notify/fanotify/目录下。

1 inotify

下图示意了 inotify 机制的框架，红色标记的数据结构是通用的核心数据结构。监控文件是一个收集被监控文件行为信息的匿名文件，之所以称它为匿名文件是因为它没有导入内核根文件系统，只是使用了进程文件操作接口。



用户进程需要先初始化一个监控文件，返回监控文件的文件描述符。初始化监控文件会创建 file、inode 等实例，并创建和初始化一个 fsnotify_group（监控组）实例（文件私有数据）。

被监控的文件需要添加到监控组，并指示需要监控的文件行为。添加被监控文件会为其创建 inotify_inode_mark 实例，内嵌通用的 fsnotify_mark 结构体成员，表示要监控的文件行为。fsnotify_mark 实例添加到被监控文件 inode 实例中链表，并关联监控组 fsnotify_group 实例。

在文件读写操作、创建设备节点等操作中，会调用 fsnotify_access()、fsnotify_mkdir()等接口函数（/include/linux/fsnotify.h），这些函数最后都会调用通用的 fsnotify()函数。fsnotify()函数调用 fsnotify_group 实例关联的 fsnotify_ops 实例中的 handle_event()函数，最终创建描述被监控文件事件的 fsnotify_event 实例，添加到 fsnotify_group 实例的文件事件双链表中。

监控文件 file_operations 实例中的 read()函数将读取 fsnotify_group 实例中双链表管理的 fsnotify_event

实例信息，转换成 inotify_event 实例，返回给用户空间，表示被监控文件的事件。

通用核心数据结构 fsnotify_group、fsnotify_ops、fsnotify_event 和 fsnotify_mark 结构体都定义在头文件/include/linux/fsnotify_backend.h，请读者自行阅读。

添加被监控文件时，调用的创建/释放、添加 fsnotify_mark 实例的接口函数定义在/fs/notify/mark.c 文件。

创建和释放 fsnotify_group 实例的通用函数定义在/fs/notify/group.c 文件内，向 fsnotify_group 实例中添加/删除 fsnotify_event 实例的函数定义在/fs/notify/notification.c 文件内。

fsnotify()函数定义在/fs/notify/fsnotify.c 文件内。

以上函数请读者自行阅读源代码。

inotify 机制定义了 inotify_inode_mark 结构体，内嵌通用 fsnotify_mark 结构体成员，inotify_event_info 结构体中内嵌通用的 fsnotify_event 结构体成员，这两个结构体都定义在/fs/notify/inotify/inotify.h 头文件。

inotify 监控组 fsnotify_group 实例关联的 fsnotify_ops 实例为 inotify_fsnotify_ops，其事件处理函数 handle_event()为 **inotify_handle_event()**（/fs/notify/inotify/inotify_fsnotify.c），此函数内将创建 inotify 机制私有的 inotify_event_info 结构体实例，并将其 fsnotify_event 结构体成员添加到监控组中的事件双链表。

inotify 机制中监控文件的 file_operations 实例为 inotify_fops（/fs/notify/inotify/inotify_user.c），其 read()函数将读取监控组中 fsnotify_event 实例的信息，转换成 inotify_event 实例，返回给用户空间。

■初始化监控文件

用户进程可通过 inotify_init()或 inotify_init1(int flags)系统调用创建（初始化）一个监控文件，flags 参数表示标记，目前只支持 IN_CLOEXEC 和 IN_NONBLOCK（/include/uapi/linux/inotify.h）。

inotify_init()和 inotify_init1(int flags)系统调用（/fs/notify/inotify/inotify_user.c）创建的是一个只读的匿名文件，返回文件描述符，源代码请读者自行阅读。

监控文件可视为一个普通的文件，普通文件的操作也适用于它，监控文件由 close()系统调用关闭。

■添加被监控文件

inotify_add_watch()系统调用用于向监控文件添加被监控文件（/fs/notify/inotify/inotify_user.c）：

```
SYSCALL_DEFINE3(inotify_add_watch, int, fd, const char __user *, pathname,u32, mask)
{
    ...     /*实现代码略*/
}
```

fd 参数表示前面初始化的监控文件的文件描述符，pathname 表示被监控文件路径，mask 参数表示需要监控的文件事件的位掩码。系统调用返回被监控文件的描述符（被监控文件要被进程打开）。

mask 参数取值定义如下（/include/uapi/linux/inotify.h）：

```
#define IN_ACCESS          0x00000001   /*文件被访问（读操作等）*/
#define IN_MODIFY          0x00000002   /*文件被修改*/
#define IN_ATTRIB          0x00000004   /*元信息被修改*/
#define IN_CLOSE_WRITE         0x00000008   /*可写文件已关闭*/
#define IN_CLOSE_NOWRITE       0x00000010   /*不可写文件关闭*/
#define IN_OPEN                0x00000020   /*文件被打开*/
#define IN_MOVED_FROM          0x00000040   /* File was moved from X */
#define IN_MOVED_TO        0x00000080   /* File was moved to Y */
#define IN_CREATE              0x00000100   /*创建了子文件*/
```

```
#define   IN_DELETE            0x00000200   /*删除了子文件*/
#define   IN_DELETE_SELF       0x00000400   /*被删除*/
#define   IN_MOVE_SELF         0x00000800   /*被移动*/

#define   IN_UNMOUNT           0x00002000  /* Backing fs was unmounted */
#define   IN_Q_OVERFLOW        0x00004000  /* Event queued overflowed */
#define   IN_IGNORED           0x00008000  /* File was ignored */

#define   IN_CLOSE         (IN_CLOSE_WRITE | IN_CLOSE_NOWRITE) /* close */
#define   IN_MOVE          (IN_MOVED_FROM | IN_MOVED_TO)         /* moves */

/*特殊标记*/
#define   IN_ONLYDIR           0x01000000   /* only watch the path if it is a directory */
#define   IN_DONT_FOLLOW       0x02000000   /*不跟踪符号链接*/
#define   IN_EXCL_UNLINK       0x04000000   /* exclude events on unlinked objects */
#define   IN_MASK_ADD          0x20000000   /* add to the mask of an already existing watch */
#define   IN_ISDIR             0x40000000   /* event occurred against dir */
#define   IN_ONESHOT           0x80000000   /* only send event once */

#define   IN_ALL_EVENTS   (IN_ACCESS | IN_MODIFY | IN_ATTRIB | IN_CLOSE_WRITE | \
                IN_CLOSE_NOWRITE | IN_OPEN | IN_MOVED_FROM | \
                IN_MOVED_TO | IN_DELETE | IN_CREATE | IN_DELETE_SELF | \
                IN_MOVE_SELF)

/*sys_inotify_init1()系统调用标记*/
#define   IN_CLOEXEC       O_CLOEXEC        /*运行新进程时关闭*/
#define   IN_NONBLOCK    O_NONBLOCK       /*非阻塞*/
```

移除被监控文件的 inotify_rm_watch()系统调用如下：

```
SYSCALL_DEFINE2(inotify_rm_watch, int, fd, __s32, wd)
{
    ...
}
```

fd 参数表示监控文件描述符，wd 表示被监控文件描述符。

内核会对 inotifty 机制的操作施以各种限制，超级用户可配置/proc/sys/fs/inotify 路径中的 3 个文件（系统控制参数）调整这些限制：

- max_queued_events：监控组事件数量上限。
- max_user_instances：一个用户创建监控文件的上限。
- max_user_watches：一个监控文件中添加被监控文件的上限。

■读取 inotify 事件

用户进程可通过对监控文件执行 read()系统调用，获取被监控文件的事件信息。read()系统调用返回用

户空间缓存区的数据是 inotify_event 结构体数组。

inotify_event 结构体定义如下（/include/uapi/linux/inotify.h）：

```
struct inotify_event {
    __s32       wd;         /*被监控文件描述符*/
    __u32       mask;       /*事件掩码*/
    __u32       cookie;     /*只对文件重命名有效*/
    __u32       len;        /*name[]数组长度，含后面的 NULL 字符*/
    char        name[0];    /*文件名*/
};
```

当被监控目录中有文件发生事件时，name 字段返回一个以空字符结尾的字符串，以标识该文件。若被监控对象自身有事件发生，则不使用 name 字段，len 值为 0。

read()系统调用中将调用 inotify_fops 实例（file_operations）中定义的 read()函数 inotify_read()函数从监控组事件双链表中读取事件信息，下面简要介绍一下此函数实现（/fs/notify/inotify/inotify_user.c）。

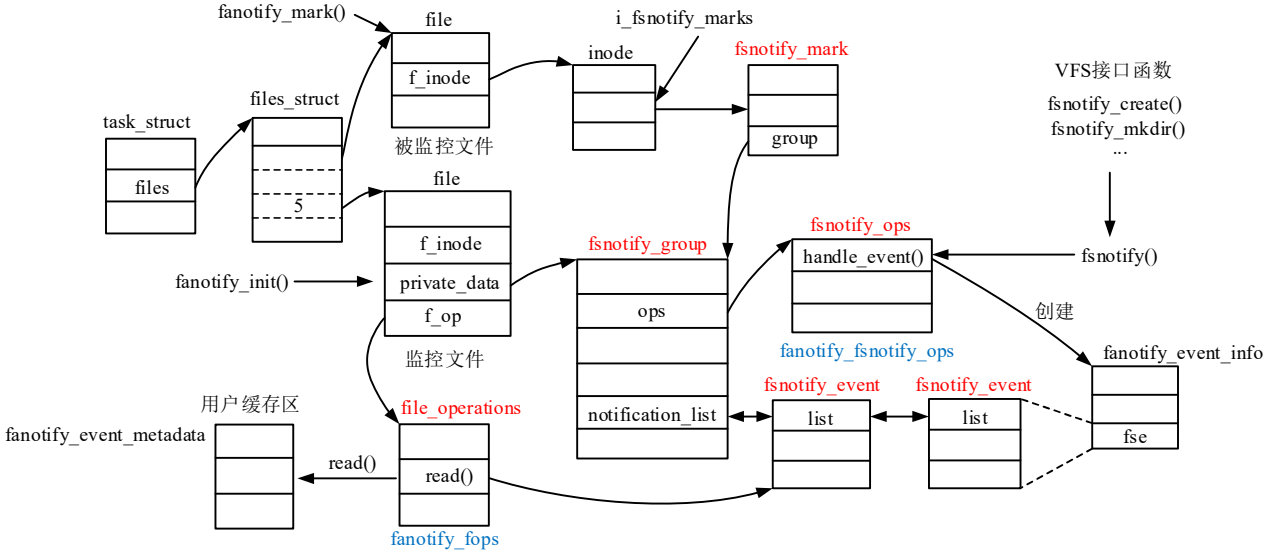
```
static ssize_t inotify_read(struct file *file, char __user *buf,size_t count, loff_t *pos)
/*buf：用户缓存区指针，count：用户缓存区大小*/
{
    struct fsnotify_group *group;
    struct fsnotify_event *kevent;
    char __user *start;
    int ret;
    DEFINE_WAIT_FUNC(wait, woken_wake_function);

    start = buf;
    group = file->private_data;      /*fsnotify_group 实例*/

    add_wait_queue(&group->notification_waitq, &wait);      /*添加到控制组等待队列*/
    while (1) {
        mutex_lock(&group->notification_mutex);
        kevent = get_one_event(group, count);          /*读取一个事件信息写入 fsnotify_event 实例*/
        mutex_unlock(&group->notification_mutex);

        pr_debug("%s: group=%p kevent=%p\n", __func__, group, kevent);

        if (kevent) {
            ret = PTR_ERR(kevent);
            if (IS_ERR(kevent))
                break;
            ret = copy_event_to_user(group, kevent, buf);          /*事件信息转为 inotify_event 实例*/
            fsnotify_destroy_event(group, kevent);        /*销毁 fsnotify_event 实例*/
            if (ret < 0)
                break;
            buf += ret;
            count -= ret;
```

```
            continue;
        }
        /*没有读到数据，可能进入睡眠*/
        ret = -EAGAIN;
        if (file->f_flags & O_NONBLOCK)        /*非阻塞，不睡眠*/
            break;
        ret = -ERESTARTSYS;
        if (signal_pending(current))       /*有挂起信号，退出*/
            break;

        if (start != buf)
            break;

        wait_woken(&wait, TASK_INTERRUPTIBLE, MAX_SCHEDULE_TIMEOUT);
    }
    remove_wait_queue(&group->notification_waitq, &wait);          /*从等待队列移出*/

    if (start != buf && ret != -EFAULT)
        ret = buf - start;
    return ret;        /*返回读取事件信息长度*/
}
```

如果用户缓存区不能容下一个事件信息，系统调用将返回-EINVAL 错误码，否则返回读取事件信息的长度，可能包含多个 inotify_event 实例（非-EFAULT 错误码）。

2 fanotify

fanotify 机制与 inotify 机制类似，实现框架如下：



fanotify 机制实现代码位于/fs/notify/fanotify/目录下。fanotify 机制框架与 inotify 机制不同之处如下：

- 被监控文件事件标记直接由 fsnotify_mark 结构体表示。
- 监控组关联 fsnotify_ops 结构体实例为 fanotify_fsnotify_ops。
- 文件事件信息由 fanotify_event_info 结构体表示，内嵌通用的 fsnotify_event 结构体成员。

●监控文件 file_operations 结构体实例为 fanotify_fops。

●返回用户空间的文件事件由 fanotify_event_metadata 结构体数组表示。

■初始化监控文件

初始化监控文件的系统调用为 **fanotify_init**(flags, event_f_flags)，参数 flags 表示监控文件标记，取值定义如下（/include/uapi/linux/fanotify.h）：

```
#define   FAN_CLOEXEC          0x00000001          /*运行新进程时关闭监控文件*/
#define   FAN_NONBLOCK         0x00000002          /*非阻塞*/

/*标记组合*/
#define   FAN_CLASS_NOTIF              0x00000000
#define   FAN_CLASS_CONTENT            0x00000004
#define   FAN_CLASS_PRE_CONTENT        0x00000008
#define   FAN_ALL_CLASS_BITS     (FAN_CLASS_NOTIF | FAN_CLASS_CONTENT | \
                                  FAN_CLASS_PRE_CONTENT)

#define   FAN_UNLIMITED_QUEUE          0x00000010
#define   FAN_UNLIMITED_MARKS          0x00000020

#define   FAN_ALL_INIT_FLAGS     (FAN_CLOEXEC | FAN_NONBLOCK | \
                                  FAN_ALL_CLASS_BITS | FAN_UNLIMITED_QUEUE |\
                                  FAN_UNLIMITED_MARKS)     /*flags 取值范围*/
```

event_f_flags 参数表示打开监控文件标记，只能在下列标记中取值（/fs/notify/fanotify/fanotify_user.c）：

```
#define   FANOTIFY_INIT_ALL_EVENT_F_BITS              ( \
                    O_ACCMODE | O_APPEND| O_NONBLOCK| \
                __O_SYNC    | O_DSYNC    | O_CLOEXEC | \
                O_LARGEFILE| O_NOATIME)
```

fanotify_init()系统调用实现函数简列如下（/fs/notify/fanotify/fanotify_user.c）：

```
SYSCALL_DEFINE2(fanotify_init, unsigned int, flags, unsigned int, event_f_flags)
{
    struct fsnotify_group *group;
    int f_flags, fd;
    struct user_struct *user;
    struct fanotify_event_info *oevent;
    ...

    if (!capable(CAP_SYS_ADMIN))      /*进程需要管理员能力，inotify 机制不需要*/
        return -EPERM;

    if (flags & ~FAN_ALL_INIT_FLAGS)   /*flags 中只能设置 FAN_ALL_INIT_FLAGS 中标记位*/
        return -EINVAL;
```

```
    /*event_f_flags 只能是 FANOTIFY_INIT_ALL_EVENT_F_BITS 中标记位*/
    if (event_f_flags & ~FANOTIFY_INIT_ALL_EVENT_F_BITS)
        return -EINVAL;

    switch (event_f_flags & O_ACCMODE) {
    case O_RDONLY:
    case O_RDWR:
    case O_WRONLY:
        break;
    default:
        return -EINVAL;
    }

    user = get_current_user();              /*用户*/
    if (atomic_read(&user->fanotify_listeners) > FANOTIFY_DEFAULT_MAX_LISTENERS) {
        ...
    }

    f_flags = O_RDWR | FMODE_NONOTIFY;          /*监控文件标记*/
    if (flags & FAN_CLOEXEC)
        f_flags |= O_CLOEXEC;
    if (flags & FAN_NONBLOCK)
        f_flags |= O_NONBLOCK;

    group = fsnotify_alloc_group(&fanotify_fsnotify_ops);          /*创建监控组*/
    ...
    group->fanotify_data.user = user;
    atomic_inc(&user->fanotify_listeners);

    oevent = fanotify_alloc_event(NULL, FS_Q_OVERFLOW, NULL); /*创建 fanotify_event_info 实例*/
    ...
    group->overflow_event = &oevent->fse;          /*溢出事件*/

    if (force_o_largefile())
        event_f_flags |= O_LARGEFILE;
    group->fanotify_data.f_flags = event_f_flags;
#ifdef CONFIG_FANOTIFY_ACCESS_PERMISSIONS
    ...
#endif
    switch (flags & FAN_ALL_CLASS_BITS) {       /*设置监控组优先级*/
    case FAN_CLASS_NOTIF:
        group->priority = FS_PRIO_0;
```

```
            break;
        case FAN_CLASS_CONTENT:
            group->priority = FS_PRIO_1;
            break;
        case FAN_CLASS_PRE_CONTENT:
            group->priority = FS_PRIO_2;
            break;
        default:
            fd = -EINVAL;
            goto out_destroy_group;
        }

        if (flags & FAN_UNLIMITED_QUEUE) {       /*队列长度不受限*/
            fd = -EPERM;
            if (!capable(CAP_SYS_ADMIN))
                goto out_destroy_group;
            group->max_events = UINT_MAX;
        } else {
            group->max_events = FANOTIFY_DEFAULT_MAX_EVENTS;
        }

        if (flags & FAN_UNLIMITED_MARKS) {
            fd = -EPERM;
            if (!capable(CAP_SYS_ADMIN))
                goto out_destroy_group;
            group->fanotify_data.max_marks = UINT_MAX;
        } else {
            group->fanotify_data.max_marks = FANOTIFY_DEFAULT_MAX_MARKS;
        }

        fd = anon_inode_getfd("[fanotify]", &fanotify_fops, group, f_flags);
                                                    /*创建监控文件，返回文件描述符*/
        ...
        return fd;          /*返回监控文件描述符*/
        ...
    }
```

■被监控文件

fanotify_mark()系统调用用于操作被监控文件，实现函数简列如下（/fs/notify/fanotify/fanotify_user.c）：

```
SYSCALL_DEFINE5(fanotify_mark, int, fanotify_fd, unsigned int, flags,
                                        __u64, mask, int, dfd, const char  __user *, pathname)
{
    ...     /*实现代码略*/
```

}

fanotify_mark()系统调用成功返回 0，而不是被监控文件描述符。

fanotify_mark()系统调用参数语义如下：

●fanotify_fd：监控文件描述符。

●pathname：被监控文件路径。

●dfd：指示被监控文件相对路径的起点。

●**flags**：标记对被监控文件的操作，取值如下（/include/uapi/linux/fanotify.h）：

```
#define  FAN_MARK_ADD                    0x00000001 /*添加被监控文件*/
#define  FAN_MARK_REMOVE                 0x00000002 /*移除被监控文件*/
#define  FAN_MARK_DONT_FOLLOW            0x00000004 /*不跟踪符号链接*/
#define  FAN_MARK_ONLYDIR                0x00000008 /*只打开目录项*/
#define  FAN_MARK_MOUNT                  0x00000010 /*标记是标载点*/
#define  FAN_MARK_IGNORED_MASK           0x00000020 /*忽略 mask 参数*/
#define  FAN_MARK_IGNORED_SURV_MODIFY    0x00000040
#define  FAN_MARK_FLUSH                  0x00000080 /*清空被监控文件 fsnotify_mark*/
```

●**mask**：标记监控文件的事件行为，只取低 32 位，取值定义如下（/include/uapi/linux/fanotify.h）：

```
#define  FAN_ACCESS              0x00000001   /*文件被访问*/
#define  FAN_MODIFY              0x00000002   /*文件被修改*/
#define  FAN_CLOSE_WRITE         0x00000008   /*可写文件已关闭*/
#define  FAN_CLOSE_NOWRITE       0x00000010   /*不可写文件已关闭*/
#define  FAN_OPEN                0x00000020   /*文件被打开*/

#define  FAN_Q_OVERFLOW          0x00004000   /*事件队列溢出*/
#define  FAN_OPEN_PERM           0x00010000   /* File open in perm check */
#define  FAN_ACCESS_PERM         0x00020000   /* File accessed in perm check */

#define  FAN_ONDIR               0x40000000   /* event occurred against dir */
#define  FAN_EVENT_ON_CHILD      0x08000000   /* interested in child events */
#define  FAN_CLOSE               (FAN_CLOSE_WRITE | FAN_CLOSE_NOWRITE) /* close */

#define  FAN_ALL_EVENTS      (FAN_ACCESS |FAN_MODIFY |FAN_CLOSE |FAN_OPEN)
          /*mark 参数只能取 FAN_ALL_EVENTS 中标记，加上 FAN_EVENT_ON_CHILD 标记*/

#define  FAN_ALL_OUTGOING_EVENTS      (FAN_ALL_EVENTS |\
                 FAN_ALL_PERM_EVENTS |FAN_Q_OVERFLOW) /*read()操作中读出事件掩码*/
```

■读取 fanotify 事件

fanotify 机制也是通过 read()系统调用获取被监控文件事件信息，监控文件 file_operation 实例 fanotify_fops 中 read()函数为 fanotify_read()，它与 inotify 实例中 inotify_read()函数类似，源代码请读者自行阅读（/fs/notify/fanotify/fanotify_user.c）。

fanotify 机制中返回用户空间的被监控文件信息由 fanotify_event_metadata 结构体数组表示。

fanotify_event_metadata 结构体定义如下（/include/uapi/linux/fanotify.h）：

```
struct fanotify_event_metadata {
    __u32 event_len;        /*实例长度*/
    __u8 vers;
    __u8 reserved;
    __u16 metadata_len;
    __aligned_u64 mask;     /*读出事件掩码，FAN_ALL_OUTGOING_EVENTS 中标记*/
    __s32 fd;               /*被监控文件描述符*/
    __s32 pid;              /*操作文件进程的 PID（引发文件事件的进程），注意不是执行 read()的进程*/
};
```

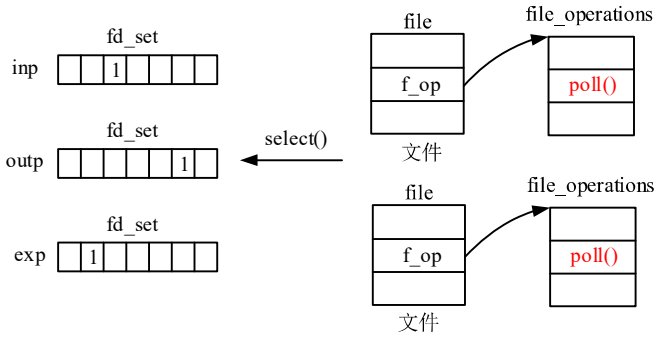
7.9.7 查询文件状态

前面介绍的监控文件是在文件已经被执行了某项操作后，通知内核，进程可获取文件的事件信息。这里说的查询文件状态，主要用于特殊文件，如设备文件、套接字、管道等，也可以是前面介绍的监控文件。查询文件状态，是指查看文件是否准备就绪，可以对其执行 I/O 操作了。监控文件可视其为普通的文件，在有文件事件信息可读时，表示文件就绪。

进程查询文件状态可通过 select()、poll()等系统调用（/fs/select.c）检查一组文件描述符，看是否有可以执行 I/O 操作的文件，这称为 I/O 多路复用。也可以使用信号，在文件就绪时向进程发送信号，通知进程，这称为信号驱动 I/O。

1 select()

select()系统调用实现框架如下图所示：



inp、outp 和 exp 都是指向 fd_set 结构体的实例。fd_set 结构体实际上是一个位图，表示文件描述符的集合，每个比特位的位置表示文件描述符 fd。

inp 位图中置 1 的位表示检测到输入就绪的文件描述符集合。

outp 位图中置 1 的位表示检测到输出就绪的文件描述符集合。

exp 位图中置 1 的位表示检测到异常情况的文件描述符集合。

fd_set 结构体定义在/include/uapi/linux/posix_types.h 头文件：

```
#define __FD_SETSIZE    1024        /*1024 个文件*/

typedef struct {
    unsigned long fds_bits[__FD_SETSIZE / (8 * sizeof(long))];
} __kernel_fd_set;
```

```
typedef    __kernel_fd_set       fd_set;          /*/include/linux/types.h*/
```

select()系统调用实现函数如下（源代码略）：

```
SYSCALL_DEFINE5(select, int, n, fd_set __user *, inp, fd_set __user *, outp,
                                                fd_set __user *, exp, struct timeval __user *, tvp)
/*n：三个位图中文件描述符中最大实际值加 1，tvp：超时时间*/
{
    ...
}
```

进程调用 select()系统调用前，需要设置 inp、outp 和 exp 位图。例如，要检测输入是否就绪的文件，需要设置其文件描述符在 inp 位图中的位，检测输出就绪的文件需要设置 outp 位图中的位，等等。

select()系统调用首先会复制 inp、outp 和 exp 位图，依次检测置位的文件描述符，调用其 file 实例关联文件操作结构中的 poll()函数，确定文件是否准备好，若准备好则置位位图中的位，否则清零。例如，要检测文件描述符 fd 输入是否就绪，如果就绪就会置位其在 inp 位图中位，否则会清零。也就是说 inp、outp 和 exp 位图在 select()系统调用中会改变，只留下就绪文件对应的比特位，非就绪的会清零。

tvp 参数指向 timeval 结构体，表示超时时间，若为 NULL，则 select()系统调用会一直阻塞，直到有文件描述符就绪。不为 NULL，超时后还没有就绪文件描述符时，系统调用返回 0。select()系统调用阻塞时可被信号中断。

file_operations 结构体中 poll()函数的返回值定义在/include/uapi/asm-generic/poll.h 头文件，select()系统调用中会根据此函数返回值，确定文件是输入就绪还是输出就绪等，如下所示（/fs/select.c）：

```
#define    POLLIN_SET    (POLLRDNORM | POLLRDBAND | POLLIN | POLLHUP | POLLERR)
                                                                    /*输入就绪*/
#define    POLLOUT_SET    (POLLWRBAND | POLLWRNORM | POLLOUT | POLLERR)
                                                                    /*输出就绪*/
#define    POLLEX_SET    (POLLPRI)          /*文件异常*/
```

poll()函数根据文件当前状态返回相应值，如果文件未就绪，可将当前文件未就绪时可将进程添加到睡眠等待队列。

select()系统调用会一直阻塞（可中断睡眠），直到有一个或多个文件描述符集合中有就绪的文件描述符（或超时）。系统调用有错误发生时，返回错误码，返回 0 表示超时，返回正整数，表示有多少个文件描述符就绪。若返回正整数，则可检查 inp、outp 和 exp 位图中比特位，看是哪个文件描述符就绪。

pselect6()是带阻塞信号掩码（阻塞时屏蔽哪些信号）的系统调用，实现函数请读者自行阅读。

2 poll()

poll()系统调用与 select()很相似，主要区别在如何指定待检查的文件描述符。在 select()中，提供了三个文件描述符集合，在每个集合中标注感兴趣的文件描述符。而在 poll()系统调用中，每个文件描述符由结构体 pollfd 表示，结构体中通过两个掩码表示欲检查的文件描述符事件和已经发生的事件。

pollfd 结构体定义在/include/uapi/asm-generic/poll.h 头文件：

```
struct pollfd {
    int fd;          /*文件描述符*/
    short events;    /*位图，需要检查的事件*/
```

```
    short revents;   /*位图，文件描述符已发生的事件*/
};
```

用户进程通过 pollfd 结构体数组，向内核传递欲检查的文件描述符及其事件，并通过此结构体数组返回文件描述符事件。

fd 表示文件描述符，events 表示要检查的事件，revents 是系统调用返回的文件描述事件，也就是说若文件描述符发生了事件，系统调用会设置 revents 位图中标记位。

events 和 revents 位图中标记位语义如下（/include/uapi/asm-generic/poll.h）：

```
#define   POLLIN       0x0001          /*可读取非高优先级的数据*/
#define   POLLPRI      0x0002          /*可读取高优先级数据*/
#define   POLLOUT      0x0004          /*普通数据可写*/
#define   POLLERR      0x0008          /*有错误发生*/
#define   POLLHUP      0x0010          /*出现挂断*/
#define   POLLNVAL     0x0020          /*文件描述符未打开*/

/*多少不太标准的标记*/
#define   POLLRDNORM        0x0040     /*等同于 POLLIN*/
#define   POLLRDBAND        0x0080     /*Linux 中不使用*/
#ifndef   POLLWRNORM
  #define   POLLWRNORM      0x0100     /*等同 POLLOUT*/
#endif
#ifndef   POLLWRBAND
  #define   POLLWRBAND      0x0200     /*优先级数据可写入*/
#endif
#ifndef POLLMSG
    #define   POLLMSG         0x0400       /*Linux 中不使用*/
#endif
#ifndef   POLLREMOVE
    #define   POLLREMOVE      0x1000
#endif
#ifndef   POLLRDHUP
    #define   POLLRDHUP       0x2000       /*对端套接字关闭*/
#endif
```

poll()系统调用在/fs/select.c 文件内实现，如下所示：

```
SYSCALL_DEFINE3(poll, struct pollfd __user *, ufds, unsigned int, nfds,int, timeout_msecs)
{
    ...      /*实现代码略*/
}
```

poll()系统调用包含 3 个参数，ufds 指向 pollfd 结构体数组，nfds 表示 ufds 指向 pollfd 结构体数组项数。参数 timeout_msecs 表示设定的超时时间（毫秒数），用于控制 poll()的行为，如下：

●大于 0：表示若在此时间段内没有文件描述符就绪，系统调用将返回 0。

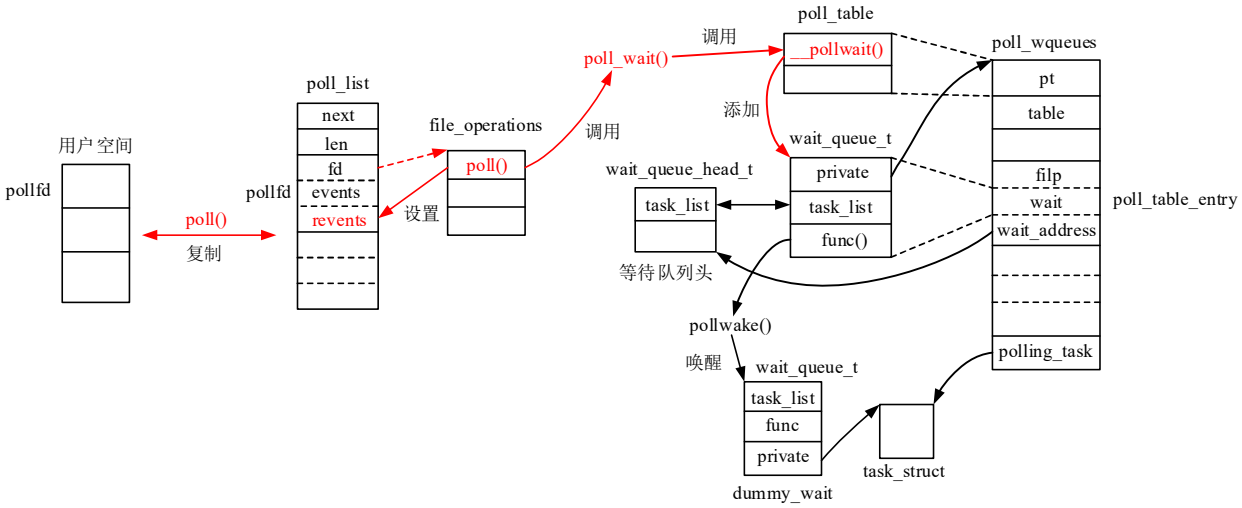
●等于 0：表示 poll()系统调用不阻塞（不睡眠等待）只做一次检查。

●等于-1：表示 poll()系统调用会一直阻塞直到有文件描述符就绪。

若有错误发生 poll()系统调用将返回-1，超时返回 0（没有文件描述符就绪），返回正整数表示就绪的文件描述符数量。

poll()系统调用返回正整数，用户进程可检测 pollfd 结构体数组项中 revents 成员值，检测文件描述符就绪的行为。

下面用图示的方式来说明 poll()系统调用的执行，如下图所示（select()系统调用内部实现其实与 poll()系统调用一样）：



poll()系统调用内会创建和初始化 poll_list 和 poll_wqueues 结构体实例，系统调用最后会释放这两个实例。

poll_list 结构体用来在内核空间暂存 pollfd 结构体数组，返回时，将数据复制到用户空间。

poll()系统调用中会依次调用 pollfd 结构体数组中各文件描述符对应文件关联 file_operations 实例中 poll()函数，poll()函数检查文件状态，返回状态值，设置到 revents 位图中。

如果文件未就绪，需要进程等待，文件操作实现中会定义一个等待队列头，poll()函数中将调用函数 poll_wait()，将 poll_wqueues 实例添加到此等待队列中。poll_wait()函数调用 poll_wqueues 实例中嵌入的 poll_table 结构体成员中的_qproc()函数完成此项工作。

poll_wqueues 结构体中包含一个 poll_table_entry 结构体数组，就是用于将 poll_wqueues 实例添加到各文件定义的等待队列中，可同时在多个文件上等待。

poll()系统调用执行进程并不是在文件操作结构中的 poll()函数中进行入睡眠，而是在扫描完 pollfd 结构体数组项后，发现没有就绪的文件描述符，将进入可中断睡眠状态。

当某个文件描述符就绪后，将会唤醒其定义的等待队列中的进程。poll_wait()函数中将 poll_wqueues 实例添加到等待队列时，其唤醒函数为 pollwake()。

pollwake()函数中将创建一个 wait_queue_t 实例，关联到 poll()系统调用执行进程，并唤醒它。进程可跟踪（等待）多个文件描述符，但只能被一个文件描述符唤醒一次。

进程唤醒后，继续执行 poll()系统调用，将再次从头依次扫描 pollfd 结构体数组中文件描述符，调用其文件操作结构中的 poll()函数，检查文件状态，不过唤醒后 poll()函数中调用的 poll_wait()函数就不会再将 poll_wqueues 实例添加到等待队列了（只会添加一次）。

以上 poll_list 结构体定义在/fs/select.c 文件内，poll_wqueues 结构体定义在/include/linux/poll.h 头文件，poll()系统调用实现函数请读者自行阅读。

文件操作结构 file_operations 中 poll()函数的原型如下：

unsigned int　(***poll)** (struct file *, struct poll_table_struct *);

第一个参数指向文件 file 实例，第二个参数指向 poll_wqueues 结构体中 poll_table 结构体成员 pt，只用于向 poll_wait()函数传递参数。这两个参数是由 poll()系统调用传递下来的参数。

poll()函数可以直接返回文件状态，如 POLLIN、POLLOUT 等。若要进程等待，则调用 **poll_wait()**接口函数将 poll_wqueues 实例添加到文件（设备驱动程序）定义的等待队列中，等待文件就绪唤醒进程。

poll_wait()接口函数定义在/include/linux/poll.h 头文件：

```
static inline void poll_wait(struct file * filp, wait_queue_head_t * wait_address, poll_table *p)
{
    if (p && p->_qproc && wait_address)
        p->_qproc(filp, wait_address, p);
}
```

wait_address 参数指向文件定义的等待队列头，filp 和 p 参数继承至 poll()函数。p 指向 poll_wqueues 结构体中 pt 成员，poll_wait()函数内将调用 p->_qproc()函数（可能为 NULL，就不调用），这个函数是由 poll_wqueues 实例给定的，负责将 poll_wqueues 实例添加到 wait_address 等待队列中。

ppoll()是带阻塞信号的 poll()系统调用，实现函数请读者自行阅读。

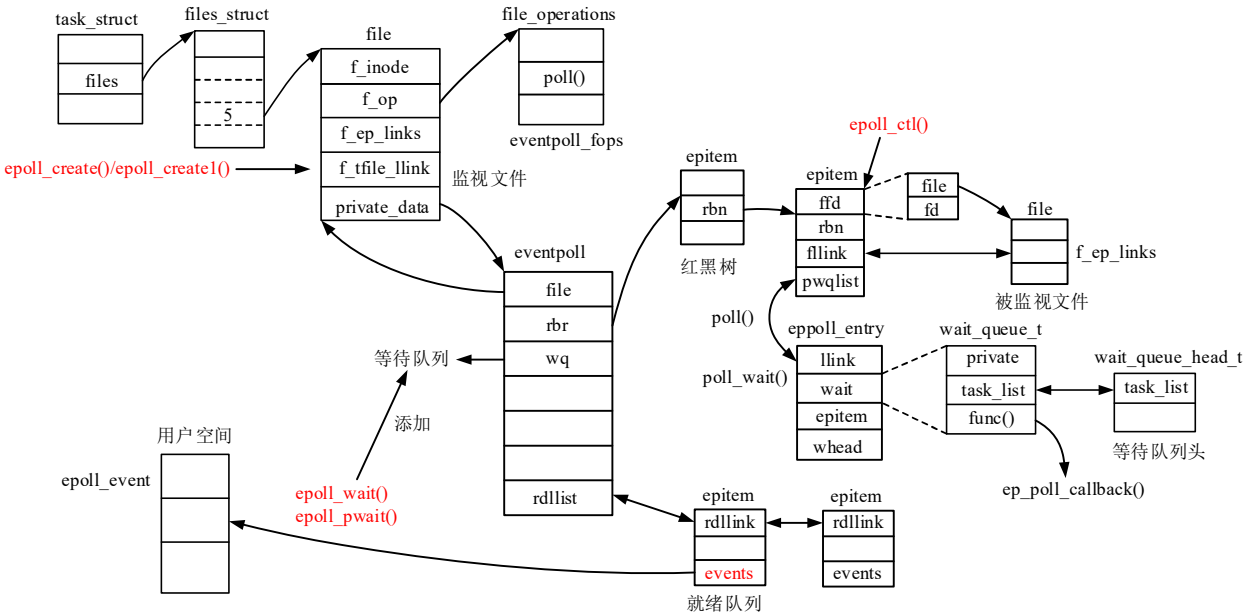
3 epoll

Linux 中 epoll 机制（Linux 专有的）同 select()、poll()系统调用一样，也可以检查多个文件描述符上的 I/O 就绪状态。epoll 机制由 3 个系统调用组成：

- **epoll_create()/epoll_create1()**：创建一个 epoll 实例，返回文件描述符。
- **epoll_ctl()**：向 epoll 实例添加、删除检查的文件描述符等，检查的文件必须定义 poll()函数。
- **epoll_wait()/epoll_pwait()**：返回就绪文件描述符的信息。

epoll 机制类似于前面介绍的 fsnotify 机制，实现代码位于/fs/eventpoll.c 文件内。

epoll 机制框架如下图所示：



eventpoll 结构体定义在/fs/eventpoll.c 文件内，表示监视文件的信息，它对应一个文件描述符：

```
struct eventpoll {
    spinlock_t lock;
    struct mutex mtx;

    wait_queue_head_t wq;              /*epoll_wait()系统调用中等待的进程*/
    wait_queue_head_t poll_wait;       /*等待队列头 file->poll() */

    struct list_head rdllist;          /*准备好的文件描述符双链表，epitem 实例*/
    struct rb_root rbr;                /*管理被监视文件描述符的红黑树根节点，epitem 实例*/
    struct epitem *ovflist;
    struct wakeup_source *ws;

    struct user_struct *user;          /*用户*/
    struct file *file;                 /*监视文件 file 实例*/
    int visited;
    struct list_head visited_list_link;
};
```

每个被监视的文件描述符由 epitem 结构体表示，定义如下（/fs/eventpoll.c）：

```
struct epitem {
    union {
        struct rb_node rbn;    /*红黑树节点，添加到 eventpoll 中红黑树（以文件 file 地址为键值）*/
        struct rcu_head rcu;
    };

    struct list_head rdllink;          /*添加到 eventpoll.rdllist 双链表，表示文件描述符就绪*/
    struct epitem *next;
    struct epoll_filefd ffd;           /*包含被监视文件 file 指针和文件描述符*/

    int nwait;
    struct list_head pwqlist;      /*链接 eppoll_entry 实例，添加到文件定义的等待队列*/
    struct eventpoll *ep;          /*指向 eventpoll 实例*/

    struct list_head fllink;           /*添加到被监视文件 file.f_ep_links 双链表*/
    struct wakeup_source __rcu *ws;          /*唤醒源*/
    struct epoll_event event;          /*文件事件标记，/include/uapi/linux/eventpoll.h*/
};
```

epoll_event 结构体定义在/include/uapi/linux/eventpoll.h 头文件：

```
struct epoll_event {
    __u32 events;           /*就绪事件位掩码*/
    __u64 data;             /*添加被监视文件时由用户进程设置，用于识别文件*/
} EPOLL_PACKED;
```

events 成员表示事件位掩码，大多与 poll()系统调用中的事件掩码相同，另外还有几个额外的标记：

```
#define   EPOLLWAKEUP      (1 << 29)
#define   EPOLLONESHOT     (1 << 30)        /*单次触发*/
#define   EPOLLET          (1 << 31)        /*边缘触发*/
```

epoll_wait()/epoll_pwait()系统调用返回用户空间的就绪文件描述符信息通过 epoll_event 结构体数组传递。

下面结合相关系统调用，简要说明 epoll 机制的实现（监控文件也可以成为被监控文件，以实现嵌套，这里暂时不考虑）。

■创建 epoll 实例

epoll_create(size)/epoll_create1(flags)系统调用用于创建 epoll 实例（eventpoll 实例），size 参数必须大于 0，但是没有实际意义，系统调用内不使用此参数；flags 只能是 EPOLL_CLOEXEC 或为 0。

系统调用内还将创建 eventpoll 实例对应的文件 file 实例，并返回文件描述符，这里称它为监视文件。

用户进程通过 close()系统调用关闭监视文件。

■被监视文件

epoll_ctl()系统调用用于操作被监视文件，系统调用如下：

```
SYSCALL_DEFINE4(epoll_ctl, int, epfd, int, op, int, fd,struct epoll_event __user *, event)
{
    ...  /*实现代码略*/
}
```

epfd 参数是 epoll_create(size)/epoll_create1(flags)系统调用创建的监视文件描述符。

op 参数表示操作命令，取值如下（/include/uapi/linux/eventpoll.h）：

```
#define   EPOLL_CTL_ADD    1        /*添加被监视文件描述符*/
#define   EPOLL_CTL_DEL    2        /*删除被监视文件描述符*/
#define   EPOLL_CTL_MOD    3        /*修改*/
```

fd 参数是被监视文件描述符，其文件操作结构需定义 poll()函数。

event 参数指向 epoll_event 实例，标记文件描述符感兴趣（监视）的事件，其 data 成员需要设置，用于标识文件，在 epoll_wait()/epoll_pwait()系统调用返回 epoll_event 实例时，其 data 成员值不变。

向 epoll_event 实例添加监视文件时，将为其创建并初始化 epitem 实例，调用文件操作结构中的 poll()函数，若 poll()函数中调用了 poll_wait()函数，则会通过 eppoll_entry 实例中的 wait 成员添加到文件定义的等待队列中。wait 成员中的唤醒函数为 ep_poll_callback()，它将唤醒在 epoll_event.wq 等待队列睡眠的进程。eppoll_entry 实例还将添加到 epitem 实例中的 pwqlist 双链表。

epitem 实例还会添加到被监视文件 file 实例的 f_ep_links 双链表和 epoll_event 实例中的红黑树。

如果在前面调用 poll()函数时，文件就有就绪的事件，则 epitem 实例通过 rdllink 成员添加到 epoll_event 实例中就绪队列 rdllist 双链表，并唤醒由 epoll_wait()系统调用在 epoll_event.wq 等待队列睡眠的进程。

删除/修改监视文件描述符的操作请读者自行阅读源代码。

■等待就绪文件

epoll_wait()/epoll_pwait()系统调用用于等待就绪的文件描述符，可返回多个就绪文件描述符信息。

epoll_wait()系统调用实现如下：

```
SYSCALL_DEFINE4(epoll_wait, int, epfd, struct epoll_event __user *, events,int, maxevents, int, timeout)
{
    ...     /*实现代码略*/
}
```

epoll_wait()系统调用各参数语义如下：

●**epfd**：epoll_create()/epoll_create1()系统调用创建的监视文件描述符。

●**events**：用户空间 epoll_event 数组指针，用于返回就绪文件信息。

●**maxevents**：epoll_event 数组项数。

●**timeout**：超时时间，毫秒数。-1 表示一直阻塞直到有文件描述符就绪，0 表示非阻塞，大于 0 则为超时时间值。

epoll_wait()/epoll_pwait()系统调用成功返回 events 数组中项数，超时返回 0，出错返回错误码。

epoll_wait()/epoll_pwait()系统调用内将取出 eventpoll 实例中就绪队列 rdllist 双链表中的 epitem 实例，从中取出 event 成员信息写入用户空间 epoll_event 实例，若没有就绪文件描述符则进程在 epoll_event.wq 等待队列睡眠。

epitem 实例 event.data 成员值在 epoll_ctl()系统调用中传递，内核不会改变此成员值，原样返回用户空间，进程通过此成员值识别是哪个文件就绪了。

epoll 机制中进程不能通过 read()系统调用来获取就绪文件描述符信息，因为监视文件 file_operations 实例中没有定义 read()函数。

前面介绍的 select()和 poll()系统调用，在进入系统调用时创建数据结构实例，返回时注销，也就是说内核不会记住被进程监视的文件。而 epoll 机制，通过一个文件来管理被监视的文件，内核会记住被监视的文件，直到进程将其从监视文件中移出（或监视文件关闭），进程可以多次调用 epoll_wait()/epoll_pwait()系统调用检查监视文件中被监视文件就绪状态，而不需要每次都向内核传递被监视文件的信息。

4 信号驱动 I/O

前面介绍的查询文件状态的系统调用中，在没有就绪文件时，进程进入睡眠等待。而信号驱动 I/O 是在文件就绪时，由内核向文件所属进程发送信号，通知进程文件就绪。进程在信号处理程序中处理就绪文件。

要使用信号驱动 I/O，程序需要按照如下步骤执行：

1、为通知信号设置处理函数，默认是 SIGIO 信号，此信号默认会使进程退出。

2、设定视监视文件（file 实例）的属主。通过 fcntl()系统调用 F_SETOWN 命令执行，如下所示：

```
fcntl(fd,F_SETOWN,pid);          /*pid 为属主进程 PID 值，pid 小于 0，则设为|pid|进程组长*/
```

3、通过设定文件 O_NONBLOCK 标志使能非阻塞 I/O。

4、通过打开文件 O_ASYNC 标志使能信号驱动 I/O。这可以和上一步合并，例如：

```
flags=fcntl(fd,F_GETFL);
fcntl(fd,F_SETFL,flags|O_ASYNC|O_NONBLOCK);
```

5、进程执行其它工作。

当 fd 表示文件就绪时，内核将向进程发送信号，在信号处理函数中处理就绪文件情况。

要想全部利用信号驱动 I/O 的优点，必须执行以下两个步骤（代替前面的第 1 步）：

1、通过 Linux 专属的 fcntl()系统调用的 F_SETSIG 命令，给文件设置一个实时信号，在就绪时向进程发送，以替代 SIGIO 信号。例如：fcntl(fd,F_SETSIG,sig)，sig 为实时信号。

2、使用 sigaction()系统调用为实时信号设置处理函数，并指定 SA_SIGINFO 标记。

文件就绪向进程发送信号时，通过 siginfo 实例传递更详细的信号信息（信号处理函数中第二个参数指向内核传递的 siginfo 实例，第一个参数为信号编号）。

siginfo 结构体中 I/O 就绪相关的成员如下：

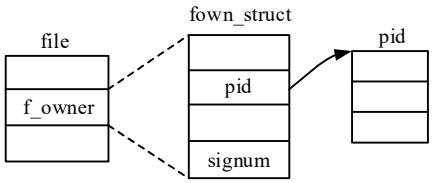
●si_fd：文件描述符。

●si_band：就绪事件位掩码。

●si_code：表示事件类型，取值定义如下（/include/uapi/asm-generic/siginfo.h）：

```
#define  POLL_IN      (__SI_POLL|1)    /*输入就绪*/
#define  POLL_OUT     (__SI_POLL|2)    /*输出就绪*/
#define  POLL_MSG     (__SI_POLL|3)    /*存在输出消息（不使用）*/
#define  POLL_ERR     (__SI_POLL|4)    /*I/O 错误*/
#define  POLL_PRI     (__SI_POLL|5)    /*存在高优先级输入*/
#define  POLL_HUP     (__SI_POLL|6)    /*出现宕机*/
#define  NSIGPOLL     6
```

fcntl()系统调用通过 F_SETOWN 命令设置文件的属主（进程），其调用 f_setown()函数设置文件 file 实例中的 f_owner 成员（如下图所示），函数定义在/fs/fcntl.c 文件内。



file 结构体中 f_owner 成员是 fown_struct 结构体（非指针），fown_struct 结构体定义如下：

```
struct fown_struct {            /*/fs/linux/fs.h*/
    rwlock_t lock;              /*自旋锁*/
    struct pid *pid;            /*pid 实例*/
    enum pid_type pid_type;     /*PID 类型（线程 ID、进程 ID、进程组 ID）*/
    kuid_t uid, euid;           /*属主进程 uid/euid*/
    int signum;                 /*实时信号*/
};
```

F_SETOWN 命令只能将文件属主设为进程组长或线程组长，F_SETOWN_EX 命令可将文件属主设为线程（由线程处理信号），请读者自行阅读命令处理函数 f_setown_ex()。

fcntl(fd,F_SETSIG,sig)系统调用直接就是利用实时信号 sig 设置 file.f_owner.**signum** 成员。

文件在就绪时就会向文件所属进程发送 signum 信号（若为 0 则发送 SIGIO 信号），例如，在前面介绍的 fsnotify 机制中，在向监控文件发送就绪事件后，会调用 kill_fasync()函数向监控文件所属进程发送信号。

7.9.8 删除文件

unlink()系统调用用于删除介质文件系统中的文件，这需要与关闭文件的 close()系统调用区分开来。关闭文件只是断开进程与内核根文件系统中文件之间的关联，并没有删除介质中的文件。

unlink()系统调用实现函数如下（/fs/namei.c）：

```
SYSCALL_DEFINE1(unlink, const char __user *, pathname)
/*pathname：路径名*/
{
    return do_unlinkat(AT_FDCWD, pathname);      /*/fs/namei.c*/
}

do_unlinkat()函数定义如下：
static long do_unlinkat(int dfd, const char __user *pathname)
{
    int error;
    struct filename *name;
    struct dentry *dentry;
    struct path path;
    struct qstr last;
    int type;
    struct inode *inode = NULL;
    struct inode *delegated_inode = NULL;
    unsigned int lookup_flags = 0;
retry:
    name = user_path_parent(dfd, pathname,&path, &last, &type, lookup_flags);      /*查找文件父目录*/
    ...
    error = -EISDIR;
    if (type != LAST_NORM)     /*末尾分量不是普通分量，返回*/
        goto exit1;
    error = mnt_want_write(path.mnt);
    ...
retry_deleg:
    mutex_lock_nested(&path.dentry->d_inode->i_mutex, I_MUTEX_PARENT);
    dentry = __lookup_hash(&last, path.dentry, lookup_flags);        /*查找末尾分量 dentry 实例*/
    error = PTR_ERR(dentry);
    if (!IS_ERR(dentry)) {
        if (last.name[last.len])
            goto slashes;
        inode = dentry->d_inode;
        if (d_is_negative(dentry))
            goto slashes;
        ihold(inode);
        error = security_path_unlink(&path, dentry);
        if (error)
```

```
            goto exit2;
        error = vfs_unlink(path.dentry->d_inode, dentry, &delegated_inode);
                    /*调用 path.dentry->d_inode->i_op->unlink(dir, dentry)函数，/fs/namei.c*/
exit2:
        dput(dentry);       /*释放目录项 dentry 实例*/
    }
    mutex_unlock(&path.dentry->d_inode->i_mutex);
    if (inode)
        iput(inode);    /*释放节点 inode 实例*/
    inode = NULL;
    ...
    mnt_drop_write(path.mnt);
exit1:
    path_put(&path);
    putname(name);
    ...
    return error;
    ...
}
```

do_unlinkat()函数中先查找父目录项，然后调用 vfs_unlink()函数删除其下的文件，此函数内调用父目录项节点操作结构中的 **unlink**(dir, dentry)函数删除文件，主要就是清除文件在父目录项文件内容中的文件目录项，而不是删除文件内容。

unlinkat()系统调用可用于删除文件和目录，实现函数如下（/fs/namei.c）：

```
SYSCALL_DEFINE3(unlinkat, int, dfd, const char __user *, pathname, int, flag)
/*flag：只能是 AT_REMOVEDIR 或 0*/
{
    if ((flag & ~AT_REMOVEDIR) != 0)
        return -EINVAL;

    if (flag & AT_REMOVEDIR)
        return do_rmdir(dfd, pathname);          /*删除目录项*/

    return do_unlinkat(dfd, pathname);              /*删除文件*/
}
```

至此，虚拟文件系统层的相关内容就全部介绍完了。

7.10 ext2 文件系统简介

从本节开始，本章后面几节将介绍几个具体文件系统类型的实现。

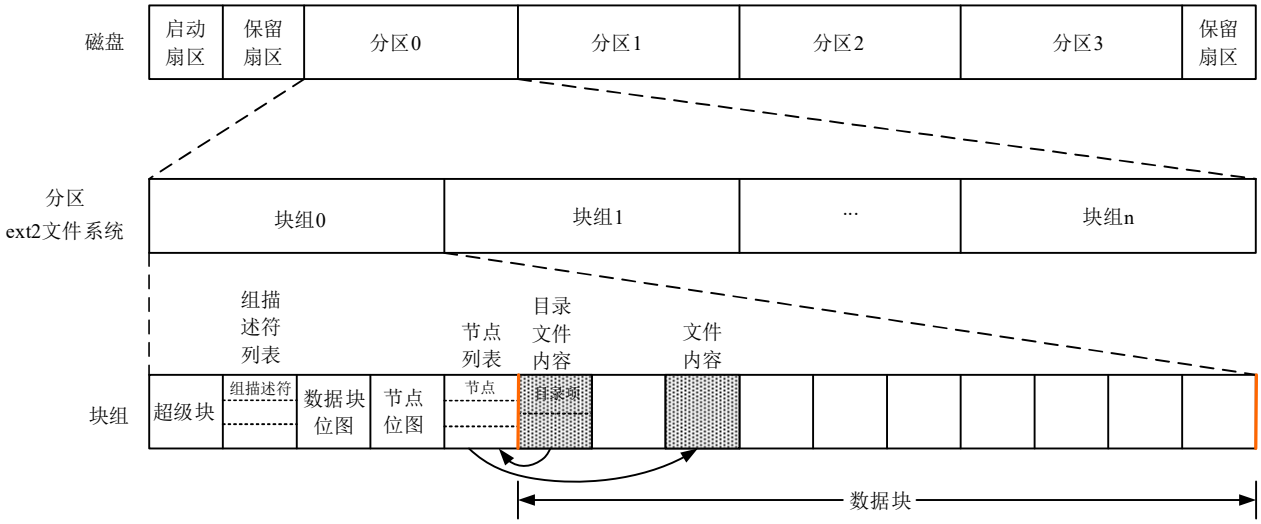
ext2 文件系统曾经是 Linux 系统的标准文件系统类型，因其前面还有一个版本名称为 ext，所以取名为 ext2，其后的版本有 ext3，ext4 等。ext2 文件系统组织结构比较简单，且非常吻合虚拟文件系统的组织形式，因此本节以此为例简要介绍介质文件系统的实现。

7.10.1 组织结构

本小节介绍 ext2 文件系统的在介质中的组织结构和内核中主要的数据结构，ext2 文件系统类型实现代码位于/fs/ext2 目录下（需选择 EXT2_FS 配置选项）。

1 物理结构

下面以磁盘为例简要讲解 ext2 文件系统的组织结构。如下图所示，假设磁盘被分为 4 个分区，磁盘第一个扇区被称为启动扇区，此扇区内包含分区表，即每个分区的起止扇区号。扇区是驱动程序访问磁盘的最小单位，通常是 512 字节。



假设分区 0 被格式化成 ext2 文件系统。ext2 文件系统将分区按块进行划分，一个块通常包含多个扇区，例如，以 2KB，4KB 为大小划分块，则每个块包含 4 个，8 个扇区，分区中每个块赋予一个块号。

多个块组成一个块组，一个分区被分成多个相邻的块组。每个块组又由以下部分组成（各部分占用若干个块）：

●**超级块**：用于存储整个文件系统的元数据，例如：块大小、已使用/空闲块数量、各种时间戳等。超级块存在于每个块组中，内容是一样的（ext2 后续版本有所修改）。

●**组描述符列表**：组描述符列表保存的是组描述符数组。每个块组对应数组中一项，即一个组描述符，记录了本块组的信息，例如：节点位图、节点列表、数据块位图等部分所处的块号等。每个块组中的组描述符列表包含了分区中所有块组的组描述符。组描述符列表占用若干个块。

每个块组中的超级块、组描述符列表都是全局的，表示整个文件系统及块组的信息。

●**数据块位图**：位图大小为 1 个块，位图中每一位表示本块组中数据块的使用情况，0 表示空闲，1 表示已被使用，文件系统通过此位图获取可用的空闲块。数据块用于保存文件内容。块大小决定了块组中数据块的数量，例如，如果块大小为 1024 字节，则块组内数据块数量为 1024*8=8192。

●**节点位图**：位图大小为 1 个块，位图中每一位表示本块组中节点列表中节点的使用情况，0 表示空闲，1 表示被使用。如果块大小为 1024 字节，则位图最多可表示 1024*8=8192 个节点，即后面节点列表中节点的数量。

●**节点列表**：保存 ext2 文件系统中的节点，此处节点的结构并不是与 VFS 中 inode 结构体一样的结构，ext2 实现代码需要从此节点中提取信息填充至 VFS 中 inode 实例。

ext2 文件系统中的节点有其固定的结构和大小，节点中保存了文件的元信息，例如，存放文件内容的数据块号、访问权限等，文件系统中每个节点具有唯一的编号。节点列表需要占用若干个块。

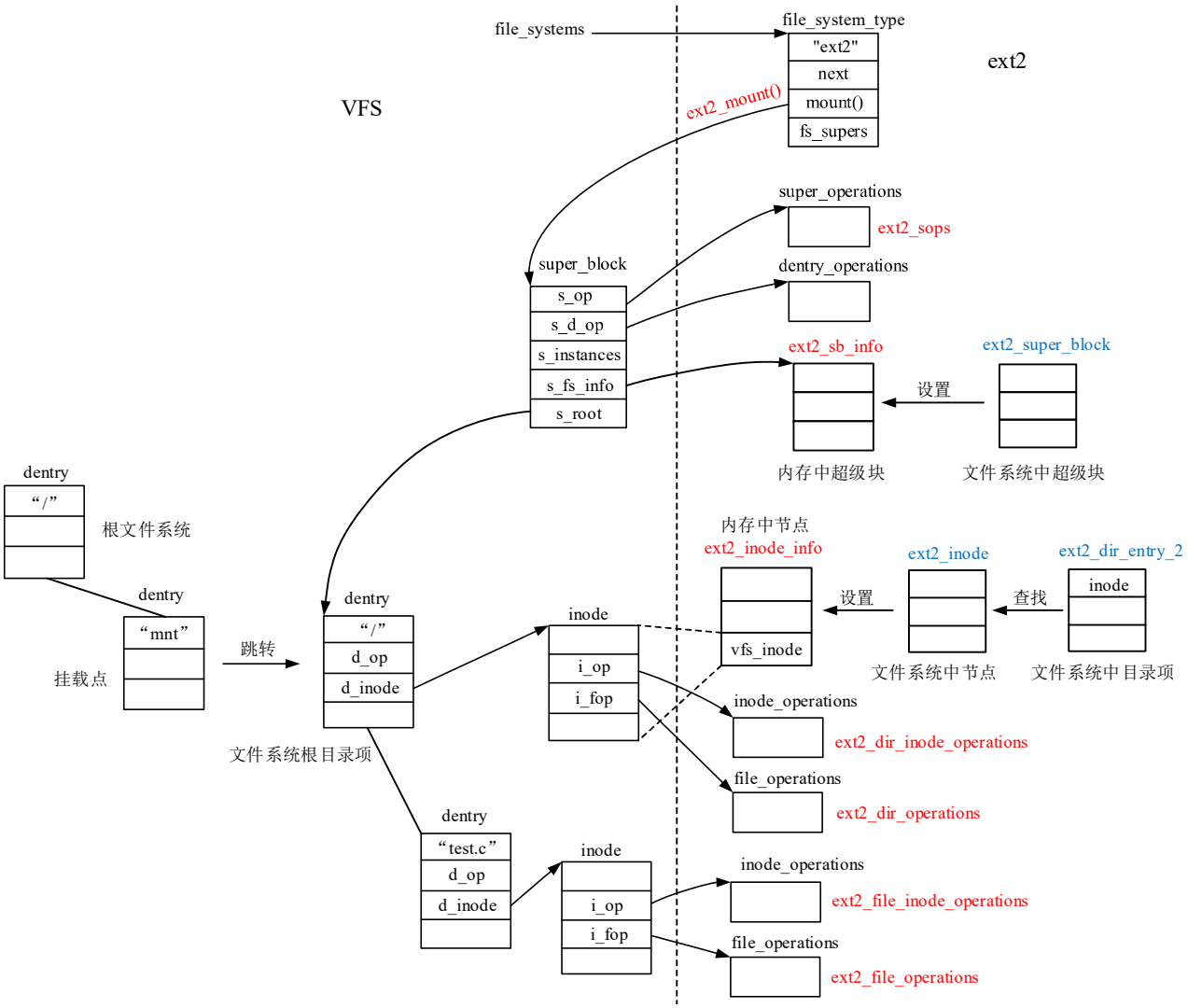
●**数据块**：数据块由若干个块组成，用于保存文件内容。

目录文件的文件内容中保存的是目录项，目录项记录了目录项/文件名称、对应的节点编号等信息，由节点编号可找到节点，节点中记录了文件的元信息，以及文件内容保存在哪些数据块的信息。

2 数据结构

ext2 文件系统中的超级块、节点、目录项的结构并不是和 VFS 中 super_block、inode、dentry 结构体的结构一样。ext2 实现代码在/fs/ext2/ext2.h 头文件中定义了表示在磁盘文件系统中超级块、节点、目录项的结构体，分别是 ext2_super_block、ext2_inode 和 ext2_dir_entry_2（ext2_dir_entry）。

ext2 在/fs/ext2/ext2.h 头文件中还定义了在内存中（文件系统类型实现代码中）表示超级块、节点的结构体 ext2_sb_info 和 ext2_inode_info，它们与 VFS 中超级块 super_block、节点 inode、目录项 dentry 结构体之间的关系如下图所示。



ext2_super_block、ext2_inode 和 ext2_dir_entry_2 结构体的定义是为了便于读取文件系统中的超级块、节点和目录项信息，读取信息后填充至 ext2_sb_info 和 ext2_inode_info 实例。

VFS 中 super_block 实例 s_fs_info 成员指向 ext2_sb_info 实例，ext2_inode_info 结构体中内嵌 inode 结构体成员，用于导入 VFS。

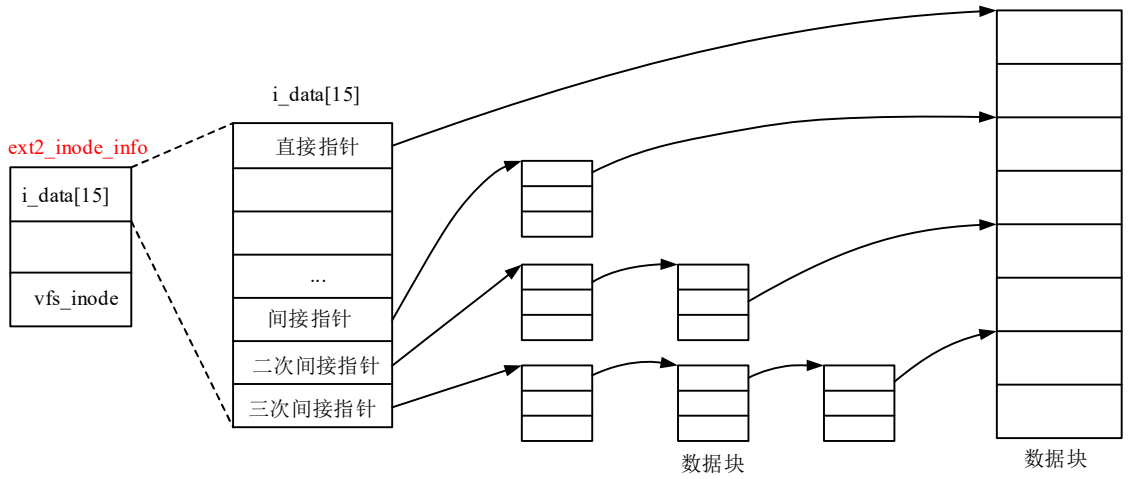
ext2_dir_entry_2 结构体表示文件系统中的目录项，其中包含的最重要的信息就是目录项名称和对应的节点编号。在搜索操作中，由目录项名称查找到目录项时，将读取其中的节点编号，以此编号读取节点信息，填充至 ext2_inode_info 实例。

这里需要说明的一点是，在文件系统节点中如何记录保存文件内容的数据块号，因为文件内容有大有小，而节点的大小是固定的。

在 ext2 节点中用一个整数数组（32 位）来记录保存文件内容的块号，这个数组将传递给 ext2_inode_info 实例。ext2_inode_info 结构体中包含一个 15 项的整数数组成员，如下所示（/fs/ext2/ext2.h）：

```
struct ext2_inode_info {
    __le32    i_data[15];      /*记录保存文件内容的数据块号*/
    ...
    struct inode    vfs_inode;        /*虚拟文件系统中 inode 实例*/
    ...
};
```

如果 i_data[15]数组中每个成员记录一个存放文件内容的数据块号，那么最多只能记录 15 个块，也就是说文件内容将不能超过 15 个数据块。为解决此问题，对 15 个数组项进行了分类，前 12 项直接记录保存文件内容的数据块号，这里称之为数据块的直接指针。剩下的 3 个数组项分别用于间接指针、二次间接指针和三次间接指针，所有间接指针指向的数据块保存的不是文件内容，而是下一级的数据块号（可视为 i_data[]数组的扩展），如下图所示，通过多次间接指针的形式可实现大文件的存储。



7.10.2 挂载文件系统

在 fs/ext2/super.c 文件内定义了 ext2 文件系统类型实例：

```
static struct file_system_type ext2_fs_type = {
    .owner          = THIS_MODULE,
    .name           = "ext2",        /*文件系统类型名称*/
    .mount          = ext2_mount,        /*挂载函数*/
    .kill_sb    = kill_block_super,
    .fs_flags   = FS_REQUIRES_DEV,
};
```

ext2 文件系统类型挂载函数为 ext2_mount()，定义如下（fs/ext2/super.c）：

```
static struct dentry *ext2_mount(struct file_system_type *fs_type,int flags, const char *dev_name, \
                                                                    void *data)
{
    return mount_bdev(fs_type, flags, dev_name, data, ext2_fill_super);     /*/fs/namespace.c*/
```

}

ext2_mount()函数内调用通用的 mount_bdev()函数，参数 ext2_fill_super 为函数指针，在创建完超级块 super_block 实例后将调用 ext2_fill_super()函数对超级块实例进行设置并创建/设置文件系统根目录项对应的 dentry 和 inode（ext2_inode_info）实例。

ext2_fill_super()函数内将创建表示 ext2 超级块信息的 ext2_sb_info 结构体实例，从文件系统所在存储介质中读取文件系统中超级块信息（由 ext2_super_block 结构体表示），设置 ext2_sb_info 实例，super_block 实例的 s_fs_info 成员，将指向 ext2_sb_info 实例。super_block 实例关联的超级块操作结构实例为 ext2_sops。

ext2_fill_super()函数内还将创建表示根目录项的 dentry 和 inode 实例，其中 inode 实例由 ext2_iget(sb, EXT2_ROOT_INO)函数创建。

ext2 定义的超级块操作结构实例 ext2_sops 如下（fs/ext2/super.c）：

```
static const struct super_operations ext2_sops = {
    .alloc_inode    = ext2_alloc_inode,      /*创建 inode 实例，fs/ext2/super.c*/
    .destroy_inode= ext2_destroy_inode,
    .write_inode    = ext2_write_inode,
    .evict_inode    = ext2_evict_inode,
    .put_super      = ext2_put_super,
    .sync_fs   = ext2_sync_fs,
    .freeze_fs= ext2_freeze,
    .unfreeze_fs    = ext2_unfreeze,
    .statfs         = ext2_statfs,
    .remount_fs     = ext2_remount,
    .show_options = ext2_show_options,
#ifdef CONFIG_QUOTA
    ...
#endif
};
```

ext2_sops 实例中 alloc_inode()函数，即 **ext2_alloc_inode()**函数，从 slab 缓存中分配 ext2_inode_info 结构体实例，并返回其中的 vfs_inode 成员（inode 结构体成员）指针。在创建 inode 实例的 ext2_iget()函数中将调用 ext2_alloc_inode()函数创建 ext2_inode_info 实例。

在 ext2_fill_super()函数中调用 ext2_iget()函数创建在 VFS 中表示 ext2 文件系统中节点的 inode 实例，函数定义在/fs/ext2/inode.c 文件内，函数代码简列如下：

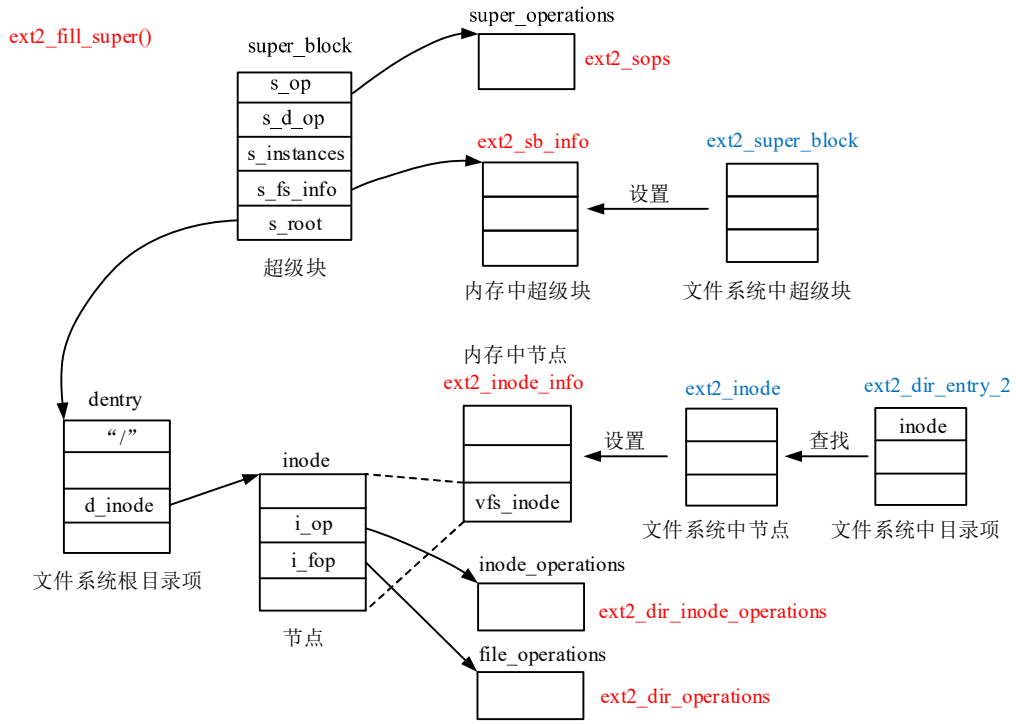
```
struct inode *ext2_iget (struct super_block *sb, unsigned long ino)
/*ino：文件系统中节点编号，根节点为 EXT2_ROOT_INO*/
{
    struct ext2_inode_info *ei;        /*内存中 ext2 节点*/
    struct buffer_head * bh;
    struct ext2_inode *raw_inode;      /*文件系统（介质）中节点*/
    struct inode *inode;       /*VFS 中节点*/
    ...
    inode = iget_locked(sb, ino);
                /*调用超级块操作结构中的 alloc_inode()函数分配 ext2_inode_info 实例*/
    ...
```

```
ei = EXT2_I(inode);        /*指向 ext2_inode_info 实例*/
ei->i_block_alloc_info = NULL;

raw_inode = ext2_get_inode(inode->i_sb, ino, &bh);      /*读取文件系统中节点信息*/
...
/*由 raw_inode 设置 ext2_inode_info 实例（含其中的 inode 结构体成员）*/
...
for (n = 0; n < EXT2_N_BLOCKS; n++)
      ei->i_data[n] = raw_inode->i_block[n];      /*保存文件内容的数据*/

if (S_ISREG(inode->i_mode)) {      /*文件类型，普通文件*/
      inode->i_op = &ext2_file_inode_operations;      /*赋值节点操作结构*/
      if (test_opt(inode->i_sb, NOBH)) {
            inode->i_mapping->a_ops = &ext2_nobh_aops;
            inode->i_fop = &ext2_file_operations;
      } else {
            inode->i_mapping->a_ops = &ext2_aops;      /*地址空间操作结构*/
            inode->i_fop = &ext2_file_operations;      /*赋值文件操作结构*/
      }
} else if (S_ISDIR(inode->i_mode)) {      /*目录项文件*/
      inode->i_op = &ext2_dir_inode_operations;
      inode->i_fop = &ext2_dir_operations;
      if (test_opt(inode->i_sb, NOBH))
            inode->i_mapping->a_ops = &ext2_nobh_aops;
      else
            inode->i_mapping->a_ops = &ext2_aops;
} else if (S_ISLNK(inode->i_mode)) {      /*符号链接*/
      if (ext2_inode_is_fast_symlink(inode)) {
            inode->i_link = (char *)ei->i_data;
            inode->i_op = &ext2_fast_symlink_inode_operations;
            nd_terminate_link(ei->i_data, inode->i_size,
                  sizeof(ei->i_data) - 1);
      } else {
            inode->i_op = &ext2_symlink_inode_operations;
            if (test_opt(inode->i_sb, NOBH))
                  inode->i_mapping->a_ops = &ext2_nobh_aops;
            else
                  inode->i_mapping->a_ops = &ext2_aops;
      }
} else {      /*特殊文件，如设备文件等*/
      inode->i_op = &ext2_special_inode_operations;
      if (raw_inode->i_block[0])
            init_special_inode(inode, inode->i_mode,
```

```
                old_decode_dev(le32_to_cpu(raw_inode->i_block[0])));      /*处理特殊文件*/
        else
            init_special_inode(inode, inode->i_mode,
                               new_decode_dev(le32_to_cpu(raw_inode->i_block[1])));
    }
    brelse (bh);
    ext2_set_inode_flags(inode);
    unlock_new_inode(inode);
    return inode;
    ...
}
```

ext2_iget ()函数中首先创建 ext2_inode_info 实例，然后根据 **ino** 参数传递的节点编号，到磁盘文件系统中读取节点信息，设置 ext2_inode_info 实例（含其中的 inode 成员），最后根据文件类型设置 inode 实例中的节点操作结构和文件操作结构实例。

总之，挂载操作中 ext2_fill_super()函数创建/设置的数据结构实例如下图所示：



7.10.3 文件操作

ext2 文件操作包括目录项文件的操作和普通文件操作，在前面介绍的 ext2_iget ()函数中对文件 inode 的节点操作和文件操作结构指针成员进行了赋值。

1 目录项文件操作

对于目录项文件，其节点操作结构和文件操作结构实例，分别如下：

```
const struct inode_operations ext2_dir_inode_operations = {          /*/fs/ext2/namei.c*/
    .create         = ext2_create,      /*创建文件*/
    .lookup         = ext2_lookup,      /*由名称查找目录项，由节点信息创建 inode 实例*/
```

```
    .link           = ext2_link,          /*创建链接*/
    .unlink         = ext2_unlink,        /*删除文件*/
    .symlink        = ext2_symlink,
    .mkdir          = ext2_mkdir,         /*创建目录项*/
    .rmdir          = ext2_rmdir,         /*删除目录项*/
    .mknod          = ext2_mknod,         /*创建设备文件、命名管道等特殊文件*/
    .rename         = ext2_rename,        /*重命名文件*/
    ...
};

const struct file_operations ext2_dir_operations = {     /*/fs/ext2/dir.c*/
    .llseek     = generic_file_llseek,
    .read       = generic_read_dir,     /*读原始文件内容*/
    .iterate    = ext2_readdir,         /*读取目录项内容*/
    .unlocked_ioctl = ext2_ioctl,
#ifdef CONFIG_COMPAT
    .compat_ioctl  = ext2_compat_ioctl,
#endif
    .fsync          = ext2_fsync,
};
```

这里需要说明一下的是 ext2_dir_inode_operations 实例中的 ext2_lookup()函数。在搜索目录项的操作中，将调用此函数在父目录项文件内容中搜索指定名称的子目录项，获取其中的节点编号。根据节点编号，调用前面介绍的 ext2_iget ()函数，创建并设置 ext2_inode_info 实例（含其中的 inode 成员），请读者自行阅读源代码。

2 普通文件操作

ext2 普通文件的节点操作结构和文件操作结构实例定义如下：（fs/ext2/file.c）：

```
const struct inode_operations ext2_file_inode_operations = {
  #ifdef CONFIG_EXT2_FS_XATTR
    ...
  #endif
    .setattr    = ext2_setattr,       /*设置文件属性*/
    .get_acl    = ext2_get_acl,
    .set_acl    = ext2_set_acl,
    .fiemap     = ext2_fiemap,
};

const struct file_operations ext2_file_operations = {
    .llseek         = generic_file_llseek,          /*通用文件定位函数*/
    .read_iter      = generic_file_read_iter,       /*通用同步读文件操作函数，见第 11 章*/
    .write_iter     = generic_file_write_iter,      /*通用同步写文件操作函数，见第 11 章*/
    .unlocked_ioctl = ext2_ioctl,
```

```
#ifdef CONFIG_COMPAT
    .compat_ioctl  = ext2_compat_ioctl,
#endif
    .mmap          = ext2_file_mmap,       /*文件映射*/
    .open          = dquot_file_open,      /*打开文件*/
    .release       = ext2_release_file,
    .fsync         = ext2_fsync,           /*调用通用的同步函数 generic_file_fsync()，见第 11 章*/
    .splice_read   = generic_file_splice_read,
    .splice_write  = iter_file_splice_write,
};
```

ext2_file_operations 实例中部分函数调用的是通用实现函数，例如，读/写操作函数被赋予通用函数 generic_file_read_iter()和 generic_file_write_iter()，函数内将调用地址空间操作结构实例 ext2_aops 中定义的函数，实现内存中文件内容页缓存与块设备文件系统中文件内容的同步，具体实现到第 11 章再做介绍。

7.11 proc 文件系统

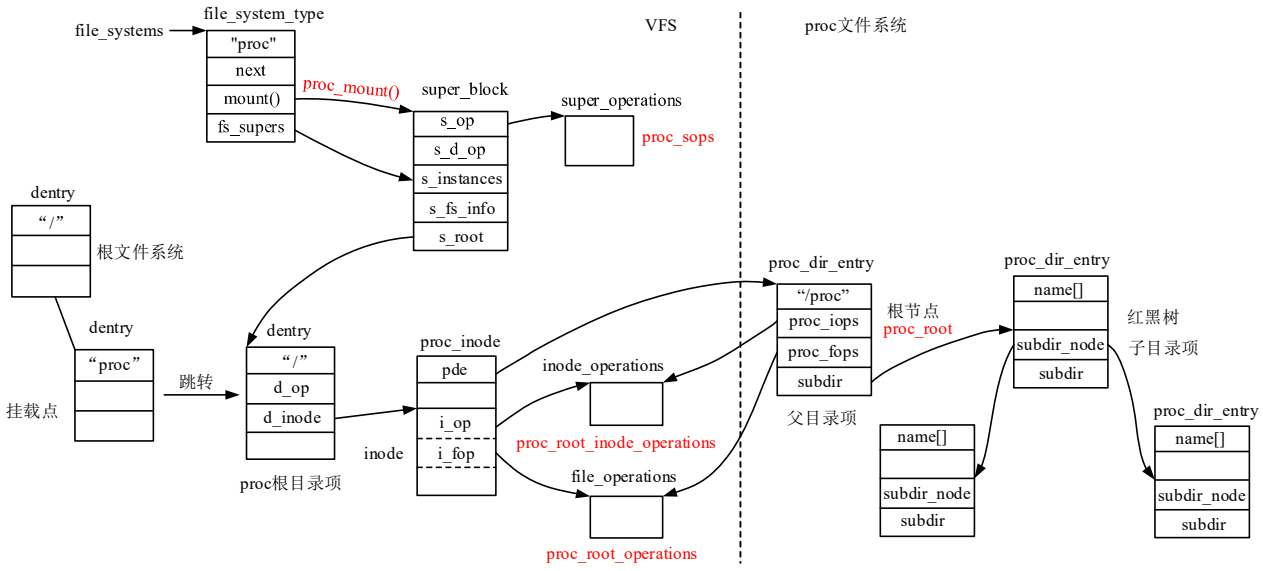
proc 文件系统是由内核 proc_dir_entry 结构体实例构成的非持久文件系统（数据只存在于内存中，关机消失），文件内容在读写操作时动态生成。proc 文件系统主要用于导出进程信息和内核参数信息，还可实现用户对内核的控制等。proc 文件系统主要用于导出内核中“软件”的信息，而后面将介绍的 sysfs 文件系统偏向于导出“硬件”信息，即设备信息。

若要内核支持 proc 文件系统需选择 PROC_FS 配置选项，相关代码位于/fs/proc/目录下。

7.11.1 概述

proc 文件系统结构如下图所示，文件系统中每个目录项，包括文件目录项，由 proc_dir_entry 结构体表示。proc_dir_entry 结构体中包含一个红黑树根节点成员 subdir，它指向的红黑树管理着其下子目录项的 proc_dir_entry 实例。proc_dir_entry 结构体中红黑树节点成员 subdir_node，用于将实例添加到其父目录项的 subdir 红黑树。子目录项 proc_dir_entry 实例以名称长度从小到大，在父目录项 proc_dir_entry 实例管理的红黑树中从左至右排列。

proc 文件系统根目录项对应的 proc_dir_entry 结构体实例为 proc_root，由内核静态创建。



proc_dir_entry 结构体中 proc_iops 成员指向节点操作结构 inode_operations 实例，proc_fops 成员指向文件操作结构 file_operations 实例。

向 proc 文件系统添加普通目录项的接口函数为 **proc_mkdir_data**(const char *name, umode_t mode, struct proc_dir_entry ***parent**, void *data)，name 参数为目录项名称，mode 为访问权限（文件类型），parent 指向父目录项的 proc_dir_entry 实例，data 指向私有数据。此函数会创建 proc_dir_entry 实例，并添加到父目录项 proc_dir_entry 实例的 subdir 红黑树中，proc_iops 和 proc_fops 成员会自动赋值，所有普通目录项（除根目录项）都指向相同的实例。

向 proc 文件系统添加文件目录项的接口函数为 **proc_create_data**(const char *name, umode_t **mode**, struct proc_dir_entry *parent,const struct file_operations ***proc_fops**,void *data)，函数参数与添加普通目录项函数的参数相似，主要区别是添加了 **proc_fops** 参数，它指向 file_operations 实例，将赋予 proc_dir_entry 实例 proc_fops 成员，也就是说每个文件需要指定文件操作结构 file_operations 实例，节点操作结构 proc_iops 成员统一赋值。

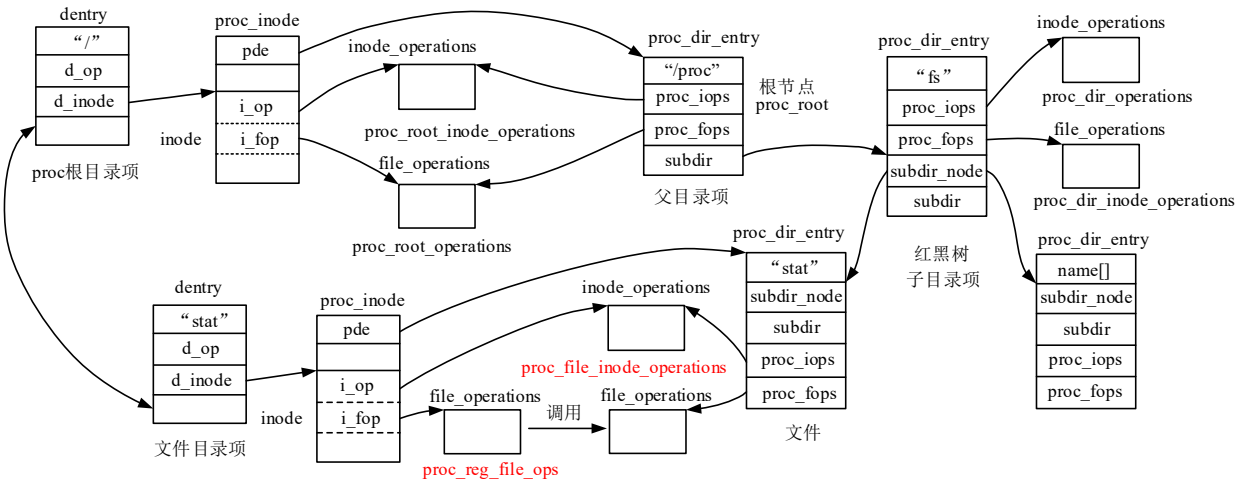
proc 文件系统需要由用户挂载，才能对用户可见。通常在操作系统启动脚本中会将 proc 文件系统挂载到/proc 目录下。挂载操作将创建 proc 文件系统超级块 super_block 实例，文件系统根目录项 dentry 实例，以及 proc_inode 实例。proc_inode 实例中包含节点 inode 结构体成员，其 i_op、i_fop 成员分别指向 proc_root 实例中 proc_iops、proc_fops 指向的实例。

对于根目录项，proc_iops、proc_fops 指向的实例是专用的，它们分别为 proc_root_inode_operations、proc_root_operations。

用户打开 proc 文件系统中文件时，将为每个路径分量（目录项）创建 dentry 和 proc_inode 实例，并建立 proc_inode 实例与 proc_dir_entry 实例之间的关联。

对于普通目录项，proc_inode 内嵌 inode 实例 i_op 和 i_fop 成员赋值为 proc_dir_entry 实例 proc_iops、proc_fops 成员值。对于文件目录项，proc_inode 内嵌 inode 实例 i_op 赋值为 proc_dir_entry 实例 proc_iops 成员值，而 i_fop 成员指向 **proc_reg_file_ops** 实例，如下图所示。

对文件进行操作时，proc_reg_file_ops 实例中的函数将调用 proc_dir_entry 实例 proc_fops 成员指向的 file_operations 实例中的函数完成操作，也就是调用具体文件定义的操作函数。



7.11.2 文件系统结构

proc 文件系统由 proc_dir_entry 结构体实例构成，内核各子系统可通过接口函数向 proc 文件系统添加目录项和文件。

1 数据结构

proc_dir_entry 结构体定义在/fs/proc/internal.h 头文件：

```
struct proc_dir_entry {
    unsigned int low_ino;        /*节点编号*/
    umode_t mode;                /*访问权限*/
    nlink_t nlink;
    kuid_t uid;                  /*用户 ID*/
    kgid_t gid;                  /*用户组 ID*/
    loff_t size;                 /*实例代表文件的大小*/
    const struct inode_operations *proc_iops;   /*节点操作结构实例指针*/
    const struct file_operations *proc_fops;    /*文件操作结构实例指针*/
    struct proc_dir_entry *parent;              /*指向父目录项*/
    struct rb_root subdir;                      /*管理子目录项的红黑树根节点*/
    struct rb_node subdir_node;                 /*红黑树节点，添加到父目录项红黑树中*/
    void *data;                  /*私有数据指针*/
    atomic_t count;              /*使用计数*/
    atomic_t in_use;             /*被用户打开/操作的使用次数*/
    struct completion *pde_unload_completion;
    struct list_head  pde_openers;   /*打开文件者双链表*/
    spinlock_t  pde_unload_lock;     /* proc_fops checks and pde_users bumps */
    u8  namelen;      /*名称长度*/
    char  name[];     /*目录项名称字符串*/
};
```

proc_dir_entry 结构体主要成员简介如下：

●**low_ino**：节点编号，赋予 VFS 中 inode 实例。

●**mode**：访问权限。

●**parent**：指向父目录项。

●**subdir**：管理子目录项的红黑树根节点。

●**subdir_node**：红黑树节点，将实例添加到父目录项管理的红黑树中。

●**data**：指向私有数据。

●**name[]**：目录项名称字符数组。

●**pde_openers**：双链表头，链接 pde_opener 结构体实例，表示一次打开操作，结构体定义如下：

```
struct pde_opener {              /*/fs/proc/internal.h*/
    struct file *file;           /*file 实例*/
    struct list_head lh;         /*添加到 pde_openers 双链表*/
    int closing;
    struct completion *c;
};
```

2 添加目录项

内核在/fs/proc/root.c 文件内静态创建了 proc 文件系统根目录项的 proc_dir_entry 实例 proc_root：

```
struct proc_dir_entry proc_root = {
```

```
    .low_ino  = PROC_ROOT_INO,   /*节点编号为 1，/include/linux/proc_ns.h*/
    .namelen = 5,        /*名称长度为 5 个字符*/
    .mode          = S_IFDIR | S_IRUGO | S_IXUGO,      /*类型及访问权限*/
    .nlink         = 2,
    .count         = ATOMIC_INIT(1),          /*引用计数*/
    .proc_iops     = &proc_root_inode_operations,     /*根目录项节点操作结构实例*/
    .proc_fops     = &proc_root_operations,          /*根目录项文件操作结构实例*/
    .parent        = &proc_root,              /*父目录项指向自身*/
    .subdir        = RB_ROOT,                 /*初始化管理子目录项的红黑树根节点*/
    .name          = "/proc",                 /*名称字符串*/
};
```

向 proc 文件系统添加普通目录项的接口函数为 proc_mkdir_data()，定义如下（/fs/proc/generic.c）：

```
struct proc_dir_entry *proc_mkdir_data(const char *name, umode_t mode,   \
                                       struct proc_dir_entry *parent, void *data)
/*name：目录项名称，mode：访问权限、类型，parent：指向父目录项，data：私有数据指针*/
{
    struct proc_dir_entry *ent;

    if (mode == 0)
        mode = S_IRUGO | S_IXUGO;          /*设置默认访问模式，具有读、执行权限*/

    ent = __proc_create(&parent, name, S_IFDIR | mode, 2);      /*目录项*/
                        /*创建并初始化 proc_dir_entry 实例，/fs/proc/generic.c*/
    if (ent) {
        ent->data = data;       /*指向私有数据*/
        ent->proc_fops = &proc_dir_operations;          /*普通目录项文件操作结构实例*/
        ent->proc_iops = &proc_dir_inode_operations;      /*普通目录项节点操作结构实例*/
        parent->nlink++;
        if (proc_register(parent, ent) < 0) {     /*注册实例，添加到父目录项红黑树，/fs/proc/generic.c*/
            ...
        }
    }
    return ent;
}
```

proc_mkdir_data()函数调用__proc_create()函数创建并初始化 proc_dir_entry 实例，设置普通目录项文件的节点操作结构和文件操作结构实例为 proc_dir_inode_operations 和 proc_dir_operations，这两个实例都是通用，最后调用 proc_register()函数注册 proc_dir_entry 实例，主要是将其添加到父目录项管理的红黑树中。

添加普通目录项的操作比较简单，这里需要说明的一下是对名称字符串的处理。名称字符串可以是 proc 文件系统中的一个路径名或只是创建目录项的名称字符串。当名称字符串是一个路径名时，parent 指向路径中第一个目录项的父目录项 proc_dir_entry 实例，路径分量中除最后目录项外，其余分量必须已经创建，否则创建目录项失败。如果名称字符串只包含创建目录项的名称，则 parent 须指向其父目录项 proc_dir_entry 实例。

例如：假设名称字符串是“a/b/c”且 parent 为空（默认为根目录项），则__proc_create()函数会在 proc 文件系统根目录下依次查找 a 和 b 目录项对应的 proc_dir_entry 实例，最终 parent 指向 b 对应的 proc_dir_entry 实例，在其下创建 c 目录项。

内核还定义了在 proc 文件系统添加普通目录项的其它接口函数，如下所示（/fs/proc/generic.c）：

```
struct proc_dir_entry *proc_mkdir(const char *name,struct proc_dir_entry *parent)
{
    return proc_mkdir_data(name, 0, parent, NULL);
}
```

proc_mkdir()函数用于添加具有默认访问权根，不带私有数据的普通目录项。

```
struct proc_dir_entry *proc_mkdir_mode(const char *name, umode_t mode,struct proc_dir_entry *parent)
{
    return proc_mkdir_data(name, mode, parent, NULL);
}
```

proc_mkdir_mode()函数用于添加指定访问权限、不带私有数据的普通目录项。

3 添加文件

向 proc 文件系统添加文件的操作与添加普通目录项的操作类似，主要区别是添加文件时，需要指定文件操作结构 file_operations 实例，传递给 proc_dir_entry 实例。

proc 文件系统添加文件的接口函数定义如下（/fs/proc/generic.c）：

```
struct proc_dir_entry *proc_create_data(const char *name, umode_t mode,   \
                        struct proc_dir_entry *parent,const struct file_operations *proc_fops,void *data)
/*proc_fops：文件操作结构实例指针，data：私有数据指针*/
{
    struct proc_dir_entry *pde;
    if ((mode & S_IFMT) == 0)
        mode |= S_IFREG;            /*普通文件*/

    if (!S_ISREG(mode)) {
        WARN_ON(1);         /* use proc_mkdir() */
        return NULL;
    }

    BUG_ON(proc_fops == NULL);

    if ((mode & S_IALLUGO) == 0)
        mode |= S_IRUGO;            /*任何用户可读*/
    pde = __proc_create(&parent, name, mode, 1);        /*创建并初始化 proc_dir_entry 实例*/
    if (!pde)
        goto out;
    pde->proc_fops = proc_fops;             /*参数传递的 file_operations 实例指针*/
    pde->data = data;           /*私有数据*/
```

```
    pde->proc_iops = &proc_file_inode_operations;      /*默认的节点操作结构体实例*/
    if (proc_register(parent, pde) < 0)                /*注册 proc_dir_entry 实例*/
        goto out_free;
    return pde;
  out_free:
    kfree(pde);
  out:
    return NULL;
}
```

由以上代码可知，对于表示文件的 proc_dir_entry 实例，其关联的节点操作结构实例统一赋值，文件操作结构实例由参数 proc_fops 传递。

内核还定义了添加文件的其它接口函数，如下所示（/include/linux/proc_fs.h）：

```
static inline struct proc_dir_entry *proc_create(const char *name, umode_t mode,
                                    struct proc_dir_entry *parent,const struct file_operations *proc_fops)
{
    return proc_create_data(name, mode, parent, proc_fops, NULL);
}
```

proc_create()函数用于添加指定访问权限，不带私有数据的文件。

4 创建符号链接

proc 文件系统除了可以添加普通目录项和文件之外，还可以创建符号链接，接口函数为 proc_symlink()，函数定义在/fs/proc/generic.c 文件内，代码如下：

```
struct proc_dir_entry *proc_symlink(const char *name,struct proc_dir_entry *parent, const char *dest)
/*name：名称，parent：父目录项，dest：链接文件的路径名*/
{
    struct proc_dir_entry *ent;

    ent = __proc_create(&parent, name,(S_IFLNK | S_IRUGO | S_IWUGO | S_IXUGO),1);
                                                    /*创建并初始化 proc_dir_entry 实例*/
    if (ent) {
        ent->data = kmalloc((ent->size=strlen(dest))+1, GFP_KERNEL);      /*私有数据*/
        if (ent->data) {
            strcpy((char*)ent->data,dest);      /*保存链接文件路径名至私有数据*/
            ent->proc_iops = &proc_link_inode_operations;      /*节点操作结构，/fs/proc/inode.c*/
            if (proc_register(parent, ent) < 0) {        /*注册 proc_dir_entry 实例*/
                ...
            }
        } else {
            ...
        }
    }
    return ent;
```

}

创建符号链接与创建普通目录项非常相似，主要区别在于 proc_dir_entry 实例 data 成员指向内存保存了链接文件的路径名，节点操作结构实例设为 proc_link_inode_operations。

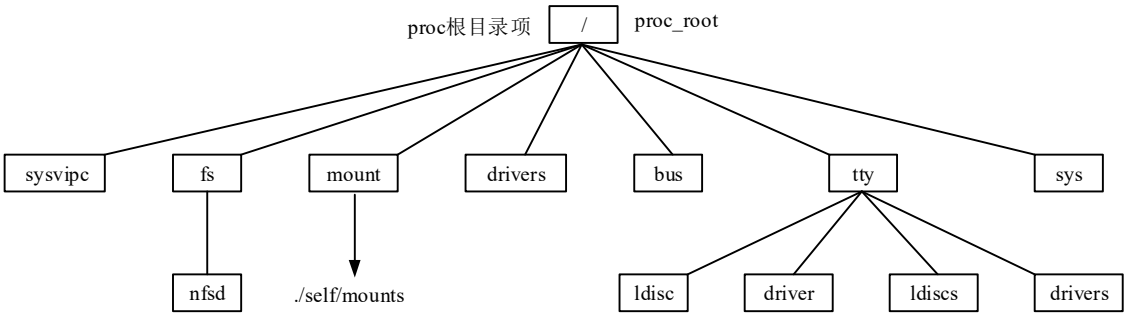
5 初始化

内核在启动函数 start_kernel()中调用 proc_root_init()函数完成 proc 文件系统的初始化工作，函数定义在/fs/proc/root.c 文件内，代码如下：

```
void __init proc_root_init(void)
{
    int err;

    proc_init_inodecache();          /*创建 proc_inode 结构体 slab 缓存，/fs/proc/inode.c*/
    err = register_filesystem(&proc_fs_type);          /*注册 proc_fs_type 文件系统类型*/
    ...

    proc_self_init();          /*为/proc/self 目录项分配节点编号保存至 self_inum，/fs/proc/self.c*/
    proc_thread_self_init();
        /*为/proc/thread_self 目录项分配节点编号保存至 thread_self_inum，/fs/proc/thread_self.c*/
    proc_symlink("mounts", NULL, "self/mounts");
                                /*创建/proc/mounts 符号链接至/proc/self/mounts*/
    proc_net_init();           /*网络相关初始化（创建目录/文件），/fs/proc/proc_net.c*/

#ifdef CONFIG_SYSVIPC
    proc_mkdir("sysvipc", NULL);          /*创建/proc/sysvipc 目录*/
#endif
    proc_mkdir("fs", NULL);          /*创建/proc/fs 目录*/
    proc_mkdir("driver", NULL);          /*创建/proc/driver 目录*/
    proc_create_mount_point("fs/nfsd");   /*创建/proc/fs/nfsd 目录，/fs/proc/generic.c*/
#if defined(CONFIG_SUN_OPENPROMFS) || defined(CONFIG_SUN_OPENPROMFS_MODULE)
    proc_create_mount_point("openprom");
#endif
    proc_tty_init();          /*tty 相关初始化，/fs/proc/proc_tty.c*/
    proc_mkdir("bus", NULL);          /*创建/proc/bus 目录*/
    proc_sys_init();          /*系统控制参数相关初始化，见下文，/fs/proc/proc_sysctl.c*/
}
```

proc_root_init()函数的主要工作是创建 proc_inode 结构体 slab 缓存，注册 proc_fs_type 文件系统类型，为各子系统创建目录项和文件等。

总之，初始化函数在 proc 文件系统中创建的 proc_dir_entry 实例层次关系如下图所示（网络相关目录项未画出），这也是 proc 文件系统初始的内容：



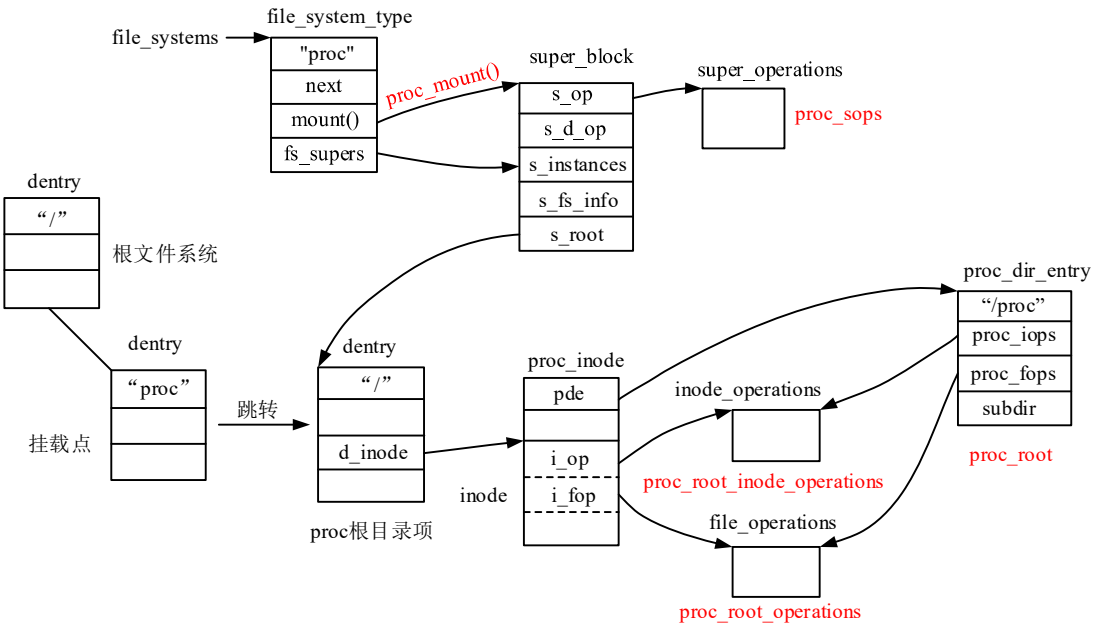
7.11.3 挂载文件系统

前面介绍的是 proc 文件系统的组织结构，相当于存储介质中文件系统的结构。内核各子系统可向其添加目录项和文件。proc 文件系统需要挂载到根文件系统（通常是/proc 挂载点），才能对用户进程可见。下面介绍 proc 文件系统的挂载。

内核在/fs/proc/root.c 文件内定义了 proc 文件系统类型实例：

```
static struct file_system_type proc_fs_type = {
    .name        = "proc",      /*名称*/
    .mount       = proc_mount,      /*挂载函数，/fs/proc/root.c*/
    .kill_sb     = proc_kill_sb,
    .fs_flags    = FS_USERNS_VISIBLE | FS_USERNS_MOUNT,
};
```

proc 文件系统通常由用户挂载到/proc 目录，文件系统类型挂载函数 proc_mount()创建的数据结构实例如下图所示，下文将介绍挂载函数的定义：



1 挂载函数

在 VFS 中 proc 文件系统节点由 proc_inode 结构体表示，它是 inode 结构体的包装器，定义如下：

```
struct proc_inode {      /*/fs/proc/internal.h*/
    struct pid *pid;      /*进程 pid*/
```

```
    int fd;
    union proc_op op;
    struct proc_dir_entry *pde;          /*指向关联的 proc_dir_entry 实例*/
    struct ctl_table_header *sysctl;     /*用于系统控制参数，见下文*/
    struct ctl_table *sysctl_entry;      /*用于系统控制参数，见下文*/
    const struct proc_ns_operations *ns_ops;
    struct inode vfs_inode;              /*inode 结构体成员*/
};
```

在初始化函数 proc_init_inodecache()中为 proc_inode 结构体创建了 slab 缓存。

proc 文件系统类型挂载函数 proc_mount()定义如下（/fs/proc/root.c）：

```
static struct dentry *proc_mount(struct file_system_type *fs_type,int flags, \
                                                    const char *dev_name, void *data)
{
    int err;
    struct super_block *sb;
    struct pid_namespace *ns;
    char *options;

    if (flags & MS_KERNMOUNT) {              /*内核发起的挂载操作*/
        ns = (struct pid_namespace *)data;     /*data 指向 pid 命名空间*/
        options = NULL;
    } else {                                 /*用户发起的挂载操作*/
        ns = task_active_pid_ns(current);      /*PID 命名空间*/
        options = data;

        if (!ns_capable(ns->user_ns, CAP_SYS_ADMIN))       /*需要 CAP_SYS_ADMIN 能力*/
            return ERR_PTR(-EPERM);
    }

    sb = sget(fs_type, proc_test_super, proc_set_super, flags, ns);
                                             /*创建超级块 super_block 实例，s->s_fs_info=ns*/
    ...
    if (!sb->s_root) {
        err = proc_fill_super(sb);
                    /*填充超级块实例，创建设置 denry、proc_inode 实例等，/fs/proc/inode.c*/
        ...
        sb->s_flags |= MS_ACTIVE;
    }
    return dget(sb->s_root);
}
```

proc_mount()函数与前面介绍过的文件系统类型挂载函数类似，下面主要介绍一下填充超级块函数的

实现，proc_fill_super()函数定义如下（/fs/proc/inode.c）：

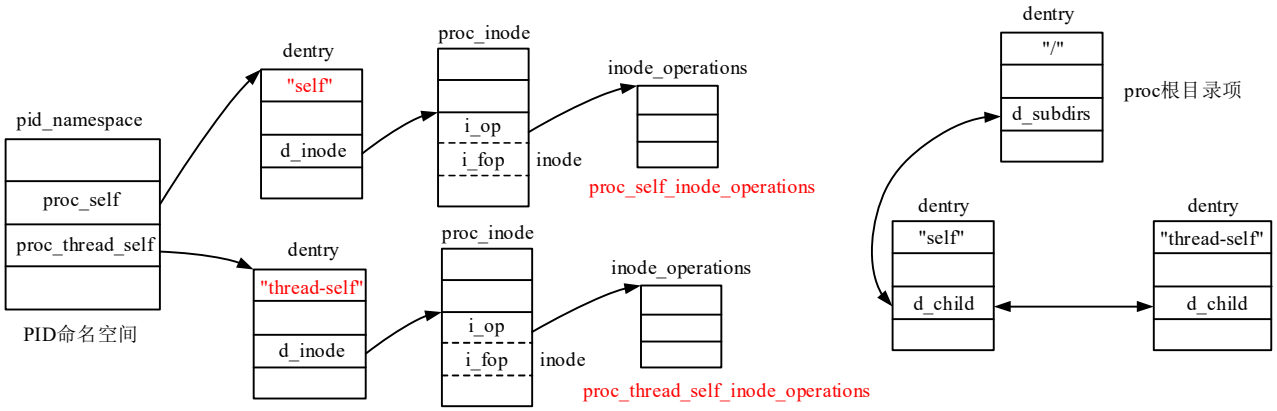
```
int proc_fill_super(struct super_block *s)
{
    struct inode *root_inode;
    int ret;

    s->s_flags |= MS_NODIRATIME | MS_NOSUID | MS_NOEXEC;
    s->s_blocksize = 1024;
    s->s_blocksize_bits = 10;
    s->s_magic = PROC_SUPER_MAGIC;
    s->s_op = &proc_sops;              /*超级块操作结构实例，/fs/proc/inode.c*/
    s->s_time_gran = 1;

    pde_get(&proc_root);          /*增加 proc_root 实例 count 引用计数值*/
    root_inode = proc_get_inode(s, &proc_root);
                        /*创建设置 proc_inode 实例，并建立与 proc_root 实例之间的关联*/
    ...
    s->s_root = d_make_root(root_inode);          /*创建文件系统根目录项 dentry 实例*/
    ...

    ret = proc_setup_self(s);    /*/fs/proc/self.c*/
      /*创建/proc/self 符号链接对应的 dentry（关联到 PID 命名空间）及 proc_inode 实例*/
    ...
    return proc_setup_thread_self(s);       /*/fs/proc/thread_self.c*/
    /*创建/proc/thread-self 符号链接对应的 dentry（关联到 PID 命名空间）及 proc_inode 实例*/
}
```

proc_fill_super()函数首先对 super_block 实例进行初始化，设置其超级块操作结构实例为 **proc_sops**，然后调用 proc_get_inode()函数创建 proc_inode 实例，并建立其与 proc_root 实例的关联，创建文件系统根目录项 dentry 实例，并关联 proc_inode.vfs_inode 成员（inode 实例），最后创建/proc/self 和/proc/thread-self 目录项对应的 dentry 和 proc_inode 实例。

proc 文件系统超级块操作结构实例定义如下（/fs/proc/inode.c）：

```
static const struct super_operations proc_sops = {
    .alloc_inode    = proc_alloc_inode,          /*分配并初始化 proc_inode 实例*/
    .destroy_inode= proc_destroy_inode,
    .drop_inode     = generic_delete_inode,
    .evict_inode    = proc_evict_inode,
    .statfs         = simple_statfs,
    .remount_fs     = proc_remount,
    .show_options = proc_show_options,
};
```

proc_fill_super()函数最后还要为/proc/self 和/proc/thread-self 符号链接创建 dentry 和 proc_inode 实例，dentry 实例关联到 PID 命名空间，并且都添加到根目录项的子目录项链表中，如下图所示：



2 创建 inode 实例

在填充超级块的 proc_fill_super()函数中，将调用的 proc_get_inode()函数创建 inode 实例（proc_inode 实例）。

proc_get_inode()函数定义如下（/fs/proc/inode.c）：

```
struct inode *proc_get_inode(struct super_block *sb, struct proc_dir_entry *de)
/*sb：超级块实例指针，de：关联 proc_dir_entry 实例指针*/
{
    struct inode *inode = new_inode_pseudo(sb);
      /*调用超级块操作结构中 proc_alloc_inode()函数，从 slab 缓存分配并初始化 proc_inode 实例*/

    if (inode) {
        inode->i_ino = de->low_ino;          /*节点编号*/
        inode->i_mtime = inode->i_atime = inode->i_ctime = CURRENT_TIME;
        PROC_I(inode)->pde = de;
                    /*inode 转 proc_inode，proc_inode->pde 指向 proc_dir_entry 实例*/

        if (is_empty_pde(de)) {        /*普通目录项，且 pde->proc_iops 为 NULL*/
            make_empty_dir_inode(inode);
            return inode;
        }
        if (de->mode) {
            inode->i_mode = de->mode;    /*访问模式*/
            inode->i_uid = de->uid;
            inode->i_gid = de->gid;
        }
        if (de->size)
            inode->i_size = de->size;
        if (de->nlink)
            set_nlink(inode, de->nlink);
        WARN_ON(!de->proc_iops);
        inode->i_op = de->proc_iops;         /*节点操作结构实例赋值*/
```

```
            if (de->proc_fops) {          /*如果 proc_dir_entry 实例关联了文件操作结构实例*/
                if (S_ISREG(inode->i_mode)) {     /*普通文件*/
                #ifdef CONFIG_COMPAT
                    ...
                #endif
                    inode->i_fop = &proc_reg_file_ops;       /*普通文件的文件操作结构实例*/
                } else {
                    inode->i_fop = de->proc_fops;   /*普通目录项，来自 proc_dir_entry 实例*/
                }
            }
        } else        /*inode 为 NULL*/
            pde_put(de);
        return inode;       /*返回 proc_inode 实例 inode 结构体成员指针*/
    }
```

proc_get_inode()函数比较简单，函数内调用 new_inode_pseudo(sb)函数，进而调用超级块操作结构中 proc_alloc_inode()函数从 slab 缓存中分配 proc_inode 实例并对初始化。proc_get_inode()函数随后对 inode 实例进行初始化，proc_inode->pde 成员指向参数传递的 proc_dir_entry 实例。

对于普通目录项文件 inode 实例 i_op 和 i_fop 指向实例来自 proc_dir_entry 实例，而对于普通文件 inode 实例，其节点操作结构实例与 proc_dir_entry 实例中指向的实例相同，文件操作结构实例指向公用的 **proc_reg_file_ops** 实例。

用户进程对普通文件的操作将调用 proc_reg_file_ops 实例中的函数，此实例中的函数进而调用 proc_dir_entry 实例 proc_fops 成员指向 file_operations 实例中的相应函数完成对文件的操作。

7.11.4 文件操作

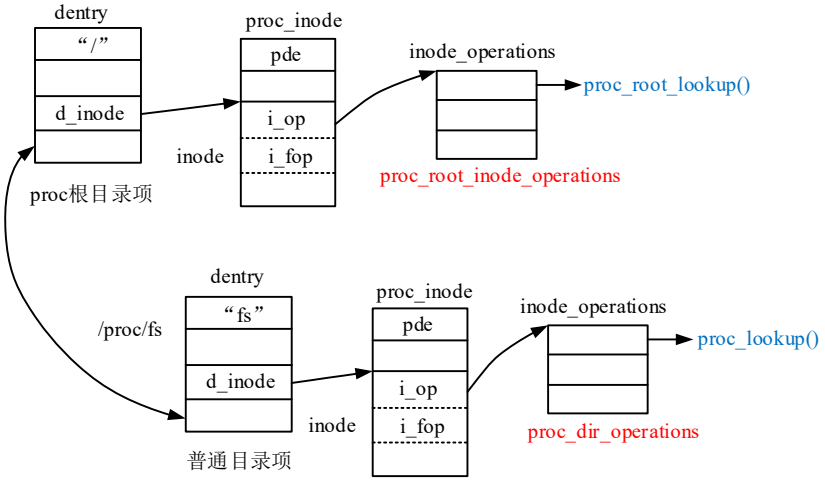
在介绍完 proc 文件系统的结构和挂载后，现在我们就可以从用户角度来考虑对 proc 文件系统中文件的操作。

打开文件是文件操作的基础，下面先介绍如何搜索 proc 文件系统中的文件，然后介绍读写文件操作的实现。

1 打开文件

由前面介绍的 open()系统调用实现可知，打开操作会逐级搜索各路径分量，创建并设置 dentry 和 inode 实例。搜索某个路径分量时，以路径分量名称在其父目录项下搜索匹配的目录项。搜索操作将调用父目录项 inode 实例关联 inode_operations 实例中的 **lookup()**函数到子目录项中去搜索匹配的目录项。

proc 文件系统中路径搜索都是从其根目录项开始的，根目录项与普通目录项关联 inode_operations 实例不同，搜索函数也不同，如下图所示。



根目录项关联 inode_operations 实例定义如下（/fs/proc/root.c）：

```
static const struct inode_operations proc_root_inode_operations = {
        .lookup    = proc_root_lookup,        /*搜索子目录项*/
        .getattr    = proc_root_getattr,
};
```

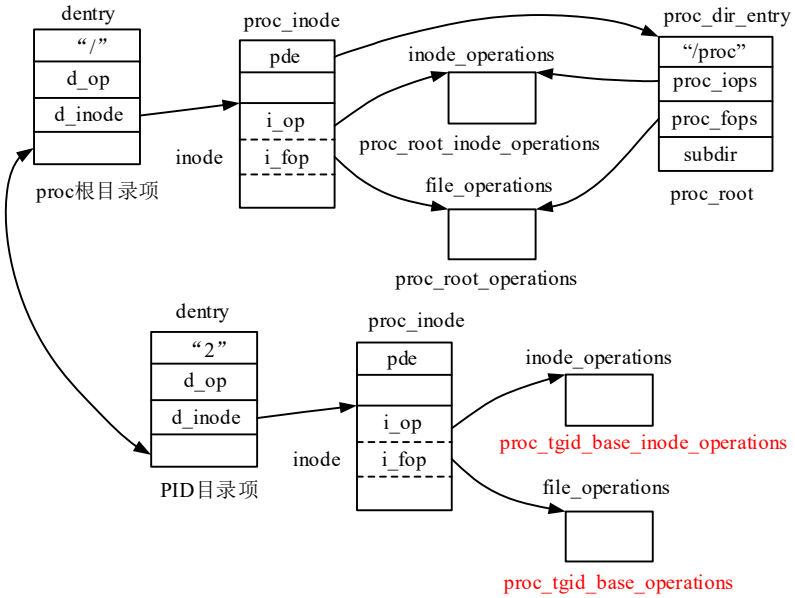
普通目录项关联 inode_operations 实例定义如下（/fs/proc/generic.c）：

```
static const struct inode_operations proc_dir_inode_operations = {
    .lookup     = proc_lookup,     /*搜索子目录项*/
    ...
};
```

根目录项下搜索函数 proc_root_lookup()定义如下（/fs/proc/root.c）：

```
static struct dentry *proc_root_lookup(struct inode * dir, struct dentry * dentry, unsigned int flags)
/*dir：根目录项对应 inode 实例，dentry：子目录项（含名称），flags：标记*/
{
    if (!proc_pid_lookup(dir, dentry, flags))       /*为进程生成 PID 子目录项，/fs/proc/base.c*/
        return NULL;                                /*生成 PID 子目录项成功，dentry 中信息有效*/

    return proc_lookup(dir, dentry, flags);         /*搜索普通子目录项，/fs/proc/generic.c*/
}
```

内核为每个进程（线程组）在 proc 文件系统根目录下生成一个以 PID 为名称的子目录项。proc_root_lookup()函数首先调用 proc_pid_lookup()函数，视子目录项名称为 PID 值（字符串转整型），查找进程 task_struct 实例，若找到则为其创建并设置 proc_inode 实例，并关联到 dentry 实例，函数返回 NULL（成功），如下图所示。



如果 proc_pid_lookup()函数不成功，目录项名称不是进程 PID，将返回错误码。proc_root_lookup()函数继续调用 proc_lookup()函数，搜索普通的子目录项。这个函数也是在普通目录项下搜索子目录项的函数。

proc_lookup()函数定义如下（/fs/proc/generic.c）：

```
struct dentry *proc_lookup(struct inode *dir, struct dentry *dentry,unsigned int flags)
/*dir：父目录项关联 inode 实例，dentry：本次搜索目录项（含名称），flags：搜索标记*/
{
    return proc_lookup_de(PDE(dir), dir, dentry);          /*/fs/proc/generic.c*/
}
```

proc_lookup_de()函数定义如下：

```
struct dentry *proc_lookup_de(struct proc_dir_entry *de, struct inode *dir,struct dentry *dentry)
/*de：父目录项关联的 proc_dir_entry 实例*/
{
    struct inode *inode;

    spin_lock(&proc_subdir_lock);
    de = pde_subdir_find(de, dentry->d_name.name, dentry->d_name.len);     /*/fs/proc/generic.c*/
        /*依目录项名称在父目录项 proc_dir_entry 实例管理的红黑树中搜索 proc_dir_entry 实例*/
    if (de) {       /*找到了名称匹配的 proc_dir_entry 实例*/
        pde_get(de);
        spin_unlock(&proc_subdir_lock);
        inode = proc_get_inode(dir->i_sb, de);     /*创建 proc_inode 实例*/
        if (!inode)
            return ERR_PTR(-ENOMEM);
        d_set_d_op(dentry, &simple_dentry_operations);       /*设置目录操作结构实例*/
        d_add(dentry, inode);              /*建立 dentry 和 inode 实例之间关联*/
        return NULL;
    }
```

```
        spin_unlock(&proc_subdir_lock);
        return ERR_PTR(-ENOENT);
    }
```

proc_lookup_de()函数比较好理解，主要工作是在父目录项对应 proc_dir_entry 实例管理的红黑树中，以子目录项名称查找匹配的 proc_dir_entry 实例，如果找到了匹配的实例，则为其创建 proc_inode 实例，并建立 dentry 和 proc_inode 内嵌 inode 实例成员之间的关联。

在 proc_get_inode()函数中将为 proc_dir_entry 实例创建 proc_inode 实例，也就是为路径分量创建 inode 实例。如果路径分表示的是普通文件，则 inode 实例关联的文件操作结构实例设为 proc_reg_file_ops，定义如下（ /fs/proc/inode.c）：

```
    static const struct file_operations proc_reg_file_ops = {       /*普通文件操作结构实例*/
        .llseek     = proc_reg_llseek,          /*定位文件*/
        .read       = proc_reg_read,            /*读文件操作函数*/
        .write      = proc_reg_write,           /*写文件操作函数*/
        .poll       = proc_reg_poll,
        .unlocked_ioctl      = proc_reg_unlocked_ioctl,
    #ifdef CONFIG_COMPAT
        .compat_ioctl  = proc_reg_compat_ioctl,
    #endif
        .mmap           = proc_reg_mmap,
        .get_unmapped_area = proc_reg_get_unmapped_area,
        .open       = proc_reg_open,            /*打开文件操作函数*/
        .release= proc_reg_release,
    };
```

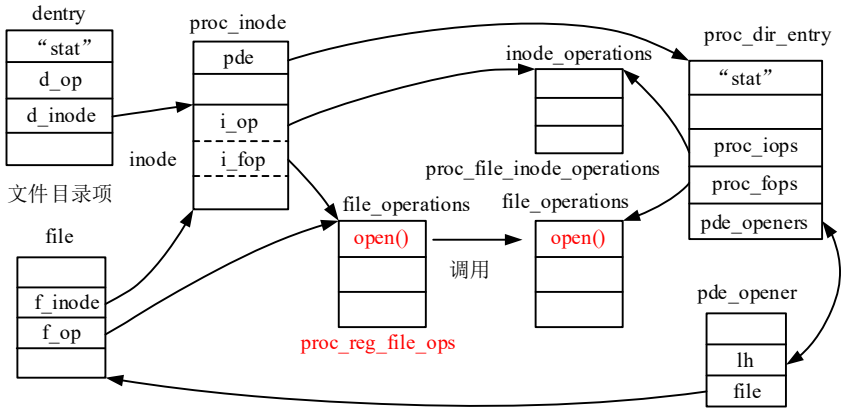
在 open()系统调用打开文件时，在 vfs_open()函数中将调用 file_operations 实例中的 open()函数完成特定于文件的打开操作，在此处即调用 proc_reg_file_ops 实例中的 proc_reg_open()函数。

proc_reg_open()函数定义如下（/fs/proc/inode.c）：

```
    static int proc_reg_open(struct inode *inode, struct file *file)
    {
        struct proc_dir_entry *pde = PDE(inode);     /*inode 转 proc_dir_entry 实例*/
        int rv = 0;
        int (*open)(struct inode *, struct file *);
        int (*release)(struct inode *, struct file *);
        struct pde_opener *pdeo;       /*打开文件者，/fs/proc/internal.h*/

        pdeo = kzalloc(sizeof(struct pde_opener), GFP_KERNEL);
                                            /*分配 pde_opener 实例，/fs/proc/internal.h*/
        ...         /*错误处理*/
        if (!use_pde(pde)) {              /*增加使用次数*/
            ...
        }
        open = pde->proc_fops->open;       /*proc_dir_entry 实例关联 file_operations 实例中 open()函数*/
```

```
    release = pde->proc_fops->release;

    if (open)
        rv = open(inode, file);      /*调用 pde->proc_fops->open()函数*/

    if (rv == 0 && release) {
        pdeo->file = file;
        spin_lock(&pde->pde_unload_lock);
        list_add(&pdeo->lh, &pde->pde_openers);
                                /*将 pde_opener 实例添加到 pde->pde_openers 双链表头部*/
        spin_unlock(&pde->pde_unload_lock);
    } else
        kfree(pdeo);

    unuse_pde(pde);        /*减小使用次数*/
    return rv;
}
```

proc_reg_open()函数执行结果如下图所示，将会创建 pde_opener 实例，关联到 file 实例，并添加到 proc_dir_entry 实例中 pde_openers 双链表，还会调用 pde->proc_fops->open()函数。



2 读写文件

在前面打开文件的基础上，就可以对文件进行操作了。文件操作调用 file 实例关联的 file_operations 实例中的函数，这里 file_operations 实例为 proc_reg_file_ops（适用普通文件），如下所示（/fs/proc/inode.c）：

```
static const struct file_operations proc_reg_file_ops = {
    ...
    .read       = proc_reg_read,        /*读文件操作函数，/fs/proc/inode.c*/
    .write      = proc_reg_write,       /*写文件操作函数，/fs/proc/inode.c*/
    ...
}
```

读写文件函数内将调用 proc_dir_entry 实例 proc_fops 成员指向 file_operations 实例中的相应函数，此实例在创建 proc_dir_entry 实例时赋值。

读文件函数 proc_reg_read()定义如下（/fs/proc/inode.c）：

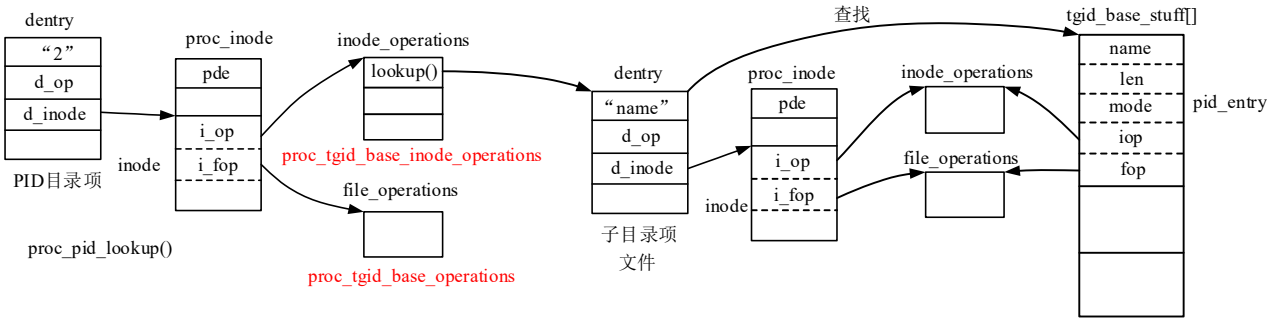
```
static ssize_t proc_reg_read(struct file *file, char __user *buf, size_t count, loff_t *ppos)
{
    ssize_t (*read)(struct file *, char __user *, size_t, loff_t *);
    struct proc_dir_entry *pde = PDE(file_inode(file));
    ssize_t rv = -EIO;
    if (use_pde(pde)) {
        read = pde->proc_fops->read;
        if (read)
            rv = read(file, buf, count, ppos);     /*调用 pde->proc_fops->read()函数*/
        unuse_pde(pde);
    }
    return rv;
}
```

proc_reg_read()函数直接调用 proc_dir_entry->proc_fops->read()函数完成读操作。

写文件函数 proc_reg_write()与读函数相似，调用 proc_dir_entry->proc_fops->write()函数完成写操作，源代码请读者自行阅读。

7.11.5 导出进程信息

内核在 proc 文件系统根目录下为每个进程（线程组）导出一个文件夹（目录项），名称为 PID。在前面介绍的根目录项节点操作结构中的 lookup()函数将调用 proc_pid_lookup()函数为进程目录项创建 dentry 和 proc_inode 实例，如下图所示：



在进程目录项下搜索子目录项时，将调用其节点操作结构中的 lookup()函数，查找 tgid_base_stuff[]数组中匹配的项，以填充 dentry 实例，并创建 proc_inode 实例。

进程目录项节点操作结构中 lookup()函数为 proc_tgid_base_lookup()，如下所示：

```
static const struct inode_operations proc_tgid_base_inode_operations = {
    .lookup    = proc_tgid_base_lookup,        /*/fs/proc/base.c*/
    .getattr   = pid_getattr,
    .setattr   = proc_setattr,
    .permission    = proc_pid_permission,
};
```

proc_tgid_base_lookup()函数依目录项名称查找 tgid_base_stuff[]数组中匹配的项，依此设置 dentry 实例和 proc_inode 实例。

tgid_base_stuff[]数组是 pid_entry 结构体数组，结构体定义如下（/fs/proc/base.c）：

```
struct pid_entry {
```

```
    const char *name;          /*子目录项/文件名称*/
    int len;
    umode_t mode;              /*模式（访问权限）*/
    const struct inode_operations *iop;       /*节点操作结构*/
    const struct file_operations *fop;        /*文件操作结构*/
    union proc_op op;
};
```

pid_entry 结构体表示进程目录项下的一个子目录项或文件，在/fs/proc/base.c 文件内还定义了初始化表示子目录项、文件、符号链接的 pid_entry 实例的宏。

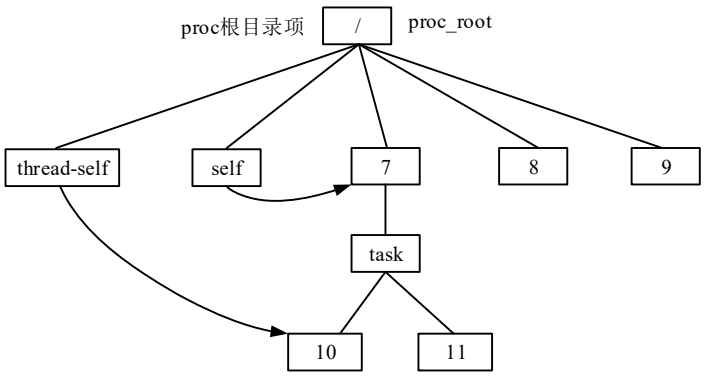
内核在/fs/proc/base.c 文件内定义了 tgid_base_stuff[]数组，表示进程目录项下的子目录项和文件，如下所示：

```
static const struct pid_entry tgid_base_stuff[] = {
    DIR("task",       S_IRUGO|S_IXUGO, proc_task_inode_operations, proc_task_operations),  /*线程*/
    DIR("fd",         S_IRUSR|S_IXUSR, proc_fd_inode_operations, proc_fd_operations),
    ...
    DIR("fdinfo",     S_IRUSR|S_IXUSR, proc_fdinfo_inode_operations, proc_fdinfo_operations),
    DIR("ns",         S_IRUSR|S_IXUGO, proc_ns_dir_inode_operations, proc_ns_dir_operations),
                                                                        /*子目录*/
    ...
    REG("environ",    S_IRUSR, proc_environ_operations),      /*文件*/
    ONE("auxv",       S_IRUSR, proc_pid_auxv),
    ONE("status",     S_IRUGO, proc_pid_status),
    ONE("personality", S_IRUSR, proc_pid_personality),
    ONE("limits",     S_IRUGO, proc_pid_limits),
    ...
}
```

tgid_base_stuff[]数组项中表示进程的信息或参数，多数是只读的，也有可写的，但需要相应的权限。如果要在进程目录项下增加新的子目录项或文件，只需要在 tgid_base_stuff[]数组中添加数组项即可。

读写文件内容时，将调用 pid_entry 实例关联 file_operations 实例中的 read()/write()函数，完成对参数/信息的读写。

在挂载 proc 文件系统时，会在 proc 文件系统根目录下创建/proc/self 和/proc/thread-self 符号链接。前者是到当前进程（线程组）PID 目录项的符号链接，后者是到当前进程（线程组）下线程 PID 目录项的符号链接，如下图所示：



线程 PID 目录项下的内容与进程（线程组）PID 目录项下的内容是一样的。

7.11.6 系统控制参数

内核在 proc 文件系统中为每个进程创建一个目录项，导出进程信息。内核在 proc 文件系统中还将导出内核（系统）的控制的参数。

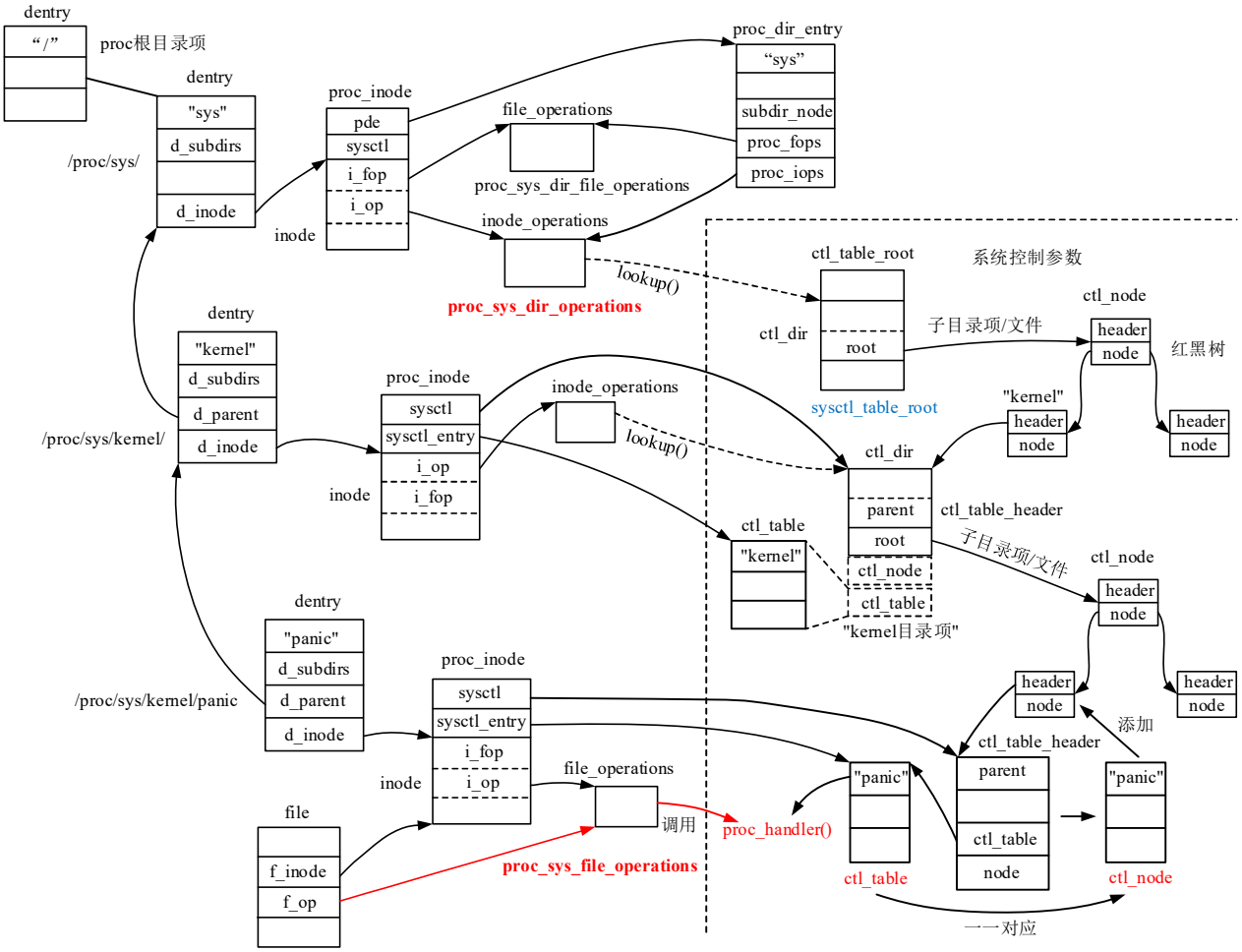
内核在运行时可以通过设置系统控制参数来控制内核的行为，无需重启系统。控制内核行为的传统方法是 sysctl()系统调用，但是此种方法并不便于使用，不建议使用。另一种系统控制的方法就是将控制参数导出到 proc 文件系统，位于/proc/sys/目录下，每个系统控制参数表现为一个文件，用户以读写文件的形式对系统控制参数进行操作。

若要内核支持系统控制参数导出到用户空间，需选择 SYSCTL 和 PROC_SYSCTL 配置选项。系统控制参数相关代码主要位于/fs/proc/proc_sysctl.c 和/kernel/sysctl.c 文件内。

1 参数管理

■框架

下图示意了内核对系统控制参数的管理：



每个系统控制参数视为一个文件，导出到 proc/sys/目录下，可以通过子目录项构成树状层次结构。具有相应权限的进程可通过对文件的读写，获取/设置系统控制参数。

proc/sys/目录下的每个目录项，由 ctl_dir 结构体表示，sys 目录项对应的 ctl_dir 结构体嵌在 ctl_table_root

结构体中，表示系统控制参数的根目录项。ctl_table_root 实例由内核静态定义。

ctl_dir 结构体中 root 成员是一棵红黑树根节点，红黑树中管理的是 ctl_node 结构体实例，通过此结构体管理目录项下的子目录项和文件。

ctl_node 结构体 header 成员指向 ctl_table_header 结构体。ctl_table_header 结构体管理同一目录项下的一批子目录项和文件。注册系统控制参数时，每个子目录项和文件由 ctl_table 结构体实例表示，通常是注册一组系统控制参数，即 ctl_table 数组。注册 ctl_table 数组时会创建一个 ctl_table_header 结构体实例，以及一个 ctl_node 实例数组，它与 ctl_table 数组项一一对应，由 ctl_table_header 实例管理。ctl_table 实例通过对应的 ctl_node 实例，添加到父目录项管理的红黑树中。

目录项 ctl_dir 结构体中第一个成员就是 ctl_table_header 结构体，用于将本目录项通过其管理的 ctl_table 和 ctl_node 实例添加到父目录项的红黑树中。

内核在初始化函数 proc_sys_init()中，在 proc 文件系统根目录下创建了名称为“sys”目录项的 proc_dir_entry 实例，其 proc_iops 成员赋值为 proc_sys_dir_operations 实例指针（节点操作结构），proc_fops 成员赋值为 proc_sys_dir_file_operations 实例指针（文件操作结构）。

proc_sys_dir_operations 实例中的 lookup()函数，在父目录项 proc_inode 实例中 sysctl 成员指向的 ctl_dir 实例的红黑树中搜索 ctl_node 实例，查找路径分量名称对应的 ctl_table 实例，以此创建并填充 proc_inode 实例，实例 sysctl_entry 成员指向表示目录项或文件的 ctl_table 实例。

sys 目录项对应 proc_inode 实例中 sysctl 成员值为 NULL，lookup()函数将默认在 ctl_table_root 实例中内嵌的 ctl_dir 实例中红黑树中搜索。

系统控制参数（文件）对应 proc_inode 实例关联的文件操作结构实例为 proc_sys_file_operations，其读写函数将调用 ctl_table 实例中的 proc_handler()函数，完成参数值的读写。

注册系统参数时，通常是定义一个 ctl_table 实例数组，数组项 ctl_table 实例可以表示一个文件，也可以表示一个子目录项，表示子目录项的 ctl_table 实例中 child 成员将指向表示其下子目录项和文件的 ctl_table 实例数组。注册函数会递归地将目录项下的所有子目录项和文件都一并注册。

注册时若不指定注册到哪个目录下，默认将注册到 proc/sys/目录下，若指定了目录则注册到指定目录项，不存在的目录项将为其创建 ctl_dir 实例。若注册的 ctl_table 实例是表示目录项，也将为其创建 ctl_dir 实例。

■数据结构

系统控制参数管理结构中使用的数据结构基本都在/include/linux/sysctl.h 头文件中定义。下面主要看一个 ctl_table 结构体的定义，它表示一个目录项或文件，注册参数时需要定义此结构体实例，其它结构体定义请读者自行阅读。

ctl_table 结构体定义如下：

```
struct ctl_table
{
    const char *procname;          /*系统控制参数名称，proc 文件系统中文件名称或目录项名称*/
    void *data;                    /*参数私有数据*/
    int maxlen;                    /*读写参数值的最大数据长度*/
    umode_t mode;                  /*参数访问模式，同文件访问模式，文件类型和访问权限*/
    struct ctl_table *child;       /*表示目录项时，指向子目录项和文件的 ctl_table 列表*/
    proc_handler *proc_handler;    /*读写系统控制参数值的接口函数*/
    struct ctl_table_poll *poll;   /*用于 poll()系统调用，包含一个等待队列*/
    void *extra1;          /*额外数据*/
    void *extra2;
```

};

ctl_table 结构体主要成员简介如下：

●**procname**：指向参数名称字符串，表示文件或目录项名称，导出到 proc 文件系统。

●**maxlen**：读写系统控制参数值时，读写数据的最大长度。

●**child**：如果本 ctl_table 实例表示一个目录项，则 child 指向的 ctl_table 列表表示其下子目录项和文件。

●**proc_handler**：函数指针，表示读写控制参数值的函数，函数原型定义如下：

typedef int proc_handler (struct ctl_table *ctl, int write,void __user *buffer, size_t *lenp, loff_t *ppos);

内核在/kernel/sysctl.c 文件内定义了许多标准的读写参数值函数，供定义系统控制参数的 proc_handler()函数调用。

■注册参数

内核在/fs/proc/proc_sysctl.c 文件内定义了注册系统控制参数的接口函数，例如：

●**register_sysctl_table(struct ctl_table *table)**：用于向/proc/sys/目录下注册系统控制参数或子目录项。简单地说，就是向/proc/sys/目录下增加子目录项或文件，子目录项下还可以有子目录项或文件。table 通常指向 ctl_table 实例数组（数组以空项结束）。

●**register_sysctl(const char *path, struct ctl_table *table)**：用于向 path 指定的目录下注册系统控制参数，table 指向 ctl_table 实例数组，path 指向路径名从/proc/sys/之后的目录项开始。例如：path 指向“kernel”，表示将参数注册到/proc/sys/kernel/目录下。如果 path 指定的路径中某个或某些目录项尚不存在，则注册函数中将创建这些目录项。

2 控制参数操作

内核各子系统可以调用前面介绍的注册函数，注册系统控制参数，用户可通过读写这些文件的内容，获取/设置系统控制参数值。

在内核初始化阶段会注册一些默认的系统控制参数。

■初始化

在 proc_root_init()初始化函数中完成了系统控制参数的初始化，函数调用关系如下：

start_kernel()->proc_root_init()->proc_sys_init()

proc_sys_init()函数定义如下（/fs/proc/proc_sysctl.c）：

```
int __init proc_sys_init(void)
{
        struct proc_dir_entry *proc_sys_root;

        proc_sys_root = proc_mkdir("sys", NULL);        /*创建 proc_dir_entry 实例，对应/proc/sys/目录*/
        proc_sys_root->proc_iops = &proc_sys_dir_operations;          /*节点操作结构实例*/
        proc_sys_root->proc_fops = &proc_sys_dir_file_operations;     /*文件操作结构实例*/
        proc_sys_root->nlink = 0;

        return sysctl_init();       /*注册内核定义的默认系统控制参数，/kernel/sysctl.c*/
```

}

proc_sys_init()函数内在 proc 文件系统中创建/proc/sys/目录项作为系统控制参数的根节点（目录项），目录项的节点操作结构和文件操作结构实例进行了重新赋值，并没有采用 proc 文件系统目录项的通用实现，函数内最后调用 sysctl_init()函数注册内核静态定义的系统控制参数列表。

sysctl_init()函数定义如下：

```
int __init sysctl_init(void)
{
    struct ctl_table_header *hdr;
    hdr = register_sysctl_table(sysctl_base_table);      /*sysctl_base_table 为 ctl_table 实例列表*/
    kmemleak_not_leak(hdr);
    return 0;
}
```

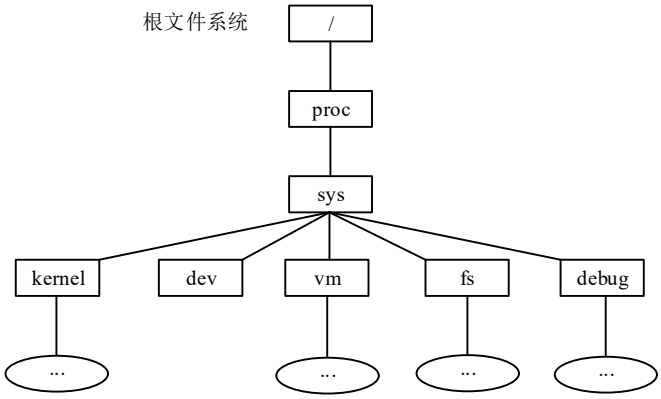
sysctl_init()函数注册的系统控制参数列表（sysctl_base_table）如下所示（/kernel/sysctl.c）：

```
static struct ctl_table sysctl_base_table[] = {
    {
        .procname     = "kernel",          /*内核相关系统控制参数（目录项）*/
        .mode         = 0555,              /*模式（访问权限）*/
        .child        = kern_table,        /*参数列表*/
    },
    {
        .procname     = "vm",              /*虚拟内存管理相关系统控制参数（目录项）*/
        .mode         = 0555,
        .child        = vm_table,          /*参数列表*/
    },
    {
        .procname     = "fs",              /*文件系统相关系统控制参数（目录项）*/
        .mode         = 0555,
        .child        = fs_table,          /*参数列表*/
    },
    {
        .procname     = "debug",           /*调试相关系统控制参数（目录项）*/
        .mode         = 0555,
        .child        = debug_table,       /*参数列表*/
    },
    {
        .procname     = "dev",             /*设备/驱动相关系统控制参数（目录项）*/
        .mode         = 0555,
        .child        = dev_table,         /*参数列表*/
    },
    { }             /*空项结尾，必须有*/
};
```

sysctl_base_table[]列表中包含的五个目录项对应的 ctl_table 实例，其下还是文件列表。例如："kernel"目录项下的文件列表简列如下所示：

```
static struct ctl_table kern_table[] = {
    {
        .procname       = "sched_child_runs_first",
        .data       = &sysctl_sched_child_runs_first,
        .maxlen         = sizeof(unsigned int),
        .mode           = 0644,
        .proc_handler = proc_dointvec,      /*处理函数*/
    },
    ...
    { }
};
```

sysctl_init()函数在调用 register_sysctl_table(sysctl_base_table)函数后，创建的目录项结构如下图所示：



■打开控制参数文件

在打开系统控制参数文件中，在搜索路径分量时，将调用目录项节点操作结构中的 lookup()函数，查找表示子目录项的 ctl_dir 实例或表示文件的 ctl_table 实例。节点操作结构实例如下所示：

```
static const struct inode_operations proc_sys_dir_operations = {
    .lookup         = proc_sys_lookup,      /*搜索子目录或文件，/fs/proc/proc_sysctl.c*/
    .permission     = proc_sys_permission,
    .setattr    = proc_sys_setattr,
    .getattr    = proc_sys_getattr,
};
```

proc_sys_lookup()函数代码请读者自行阅读，其内部会调用 proc_sys_make_inode()函数创建并设置目录项对应的 proc_inode 实例。

proc_sys_make_inode()函数代码简列如下：

```
static struct inode *proc_sys_make_inode(struct super_block *sb,
                                         struct ctl_table_header *head, struct ctl_table *table)
{
    struct inode *inode;
```

```
    struct proc_inode *ei;

    inode = new_inode(sb);        /*调用超级块操作结构中的函数，分配 proc_inode 实例*/
    ...
    inode->i_ino = get_next_ino();        /*节点编号*/

    sysctl_head_get(head);
    ei = PROC_I(inode);            /*指向 proc_inode 实例*/
    ei->sysctl = head;             /*指向 ctl_table_header 实例*/
    ei->sysctl_entry = table;      /*指向 ctl_table 实例*/

    inode->i_mtime = inode->i_atime = inode->i_ctime = CURRENT_TIME;
    inode->i_mode = table->mode;
    if (!S_ISDIR(table->mode)) {          /*文件*/
        inode->i_mode |= S_IFREG;
        inode->i_op = &proc_sys_inode_operations;
        inode->i_fop = &proc_sys_file_operations;        /*文件操作结构实例*/
    } else {          /*目录项*/
        inode->i_mode |= S_IFDIR;
        inode->i_op = &proc_sys_dir_operations;
        inode->i_fop = &proc_sys_dir_file_operations;
        if (is_empty_dir(head))
            make_empty_dir_inode(inode);
    }
out:
    return inode;
}
```

对于系统控制参数文件其文件操作结构实例为 **proc_sys_file_operations**，定义如下：

```
static const struct file_operations proc_sys_file_operations = {
    .open       = proc_sys_open,        /*打开文件操作*/
    .poll       = proc_sys_poll,
    .read       = proc_sys_read,        /*读文件操作*/
    .write      = proc_sys_write,       /*写文件操作*/
    .llseek     = default_llseek,
};
```

在打开文件操作 open()系统调用最后，将调用 file_operations 实例中的 open()函数，执行特定于文件的打开操作。对于系统控制参数文件，其 file_operations 实例中的 open()函数 proc_sys_open()定义如下：

```
static int proc_sys_open(struct inode *inode, struct file *filp)
{
    struct ctl_table_header *head = grab_header(inode);      /*ctl_table 关联 ctl_table_header 实例*/
    struct ctl_table *table = PROC_I(inode)->sysctl_entry;     /*ctl_table 实例*/
```

```
    ...
    if (table->poll)
        filp->private_data = proc_sys_poll_event(table->poll);
                    /*(void*)(unsigned long)atomic_read(&poll->event)，/include/linux/sysctl.h*/
    sysctl_head_finish(head);

    return 0;
}
```

■读写控制参数文件

对于系统控制参数文件，其 file_operations 实例中读写函数为 proc_sys_read()和 proc_sys_write()，这两个函数定义如下（/fs/proc/proc_sysctl.c）：

```
static ssize_t proc_sys_read(struct file *filp, char __user *buf,size_t count, loff_t *ppos)
{
    return proc_sys_call_handler(filp, (void __user *)buf, count, ppos, 0);        /*参数 0 表示读操作*/
}

static ssize_t proc_sys_write(struct file *filp, const char __user *buf,size_t count, loff_t *ppos)
{
    return proc_sys_call_handler(filp, (void __user *)buf, count, ppos, 1);       /*参数 1 表示写操作*/
}
```

读/写操作函数都是调用 proc_sys_call_handler()函数完成操作，函数定义如下：

```
static ssize_t proc_sys_call_handler(struct file *filp, void __user *buf,size_t count, loff_t *ppos, int write)
 /*write：0 表示读，1 表示写*/
{
    struct inode *inode = file_inode(filp);
    struct ctl_table_header *head = grab_header(inode);
    struct ctl_table *table = PROC_I(inode)->sysctl_entry;          /*指向 ctl_table 实例*/
    ssize_t error;
    size_t res;
    ...
    error = -EPERM;
    if (sysctl_perm(head, table, write ? MAY_WRITE : MAY_READ))
                                                /*检查访问权限，/fs/proc/proc_sysctl.c*/
        goto out;

    error = -EINVAL;
    if (!table->proc_handler)       /*ctl_table 实例没有定义 proc_handler()函数直接返回*/
        goto out;

    res = count;
    error = table->proc_handler(table, write, buf, &res, ppos);    /*调用 ctl_table 实例定义的处理函数*/
```

```
    ...
out:
    sysctl_head_finish(head);
    return error;
}
```

proc_sys_call_handler()函数内检查文件的访问权限后，调用 ctl_table 实例定义的 proc_handler()函数完成读写操作。

内核在/kernel/sysctl.c 内定义了许多标准的读写系统控制参数值的函数，例如：proc_dointvec()函数，用于直接从 ctl_table->data 表示的地址读/写数据。

内核各子系统中需要使用系统控制参数时，只需要定义 ctl_table 实例（列表），赋予标准的或自定义的 proc_handler()函数，然后调用 register_sysctl_table(table)或 register_sysctl(path, table)函数向内核注册即可，甚至可以直接将 ctl_table 实例插入到内核在/kernel/sysctl.c 文件内静态定义的系统控制参数列表中，由内核完成注册。注册后，用户就可以通过 proc 文件系统以文件的形式访问系统控制参数。

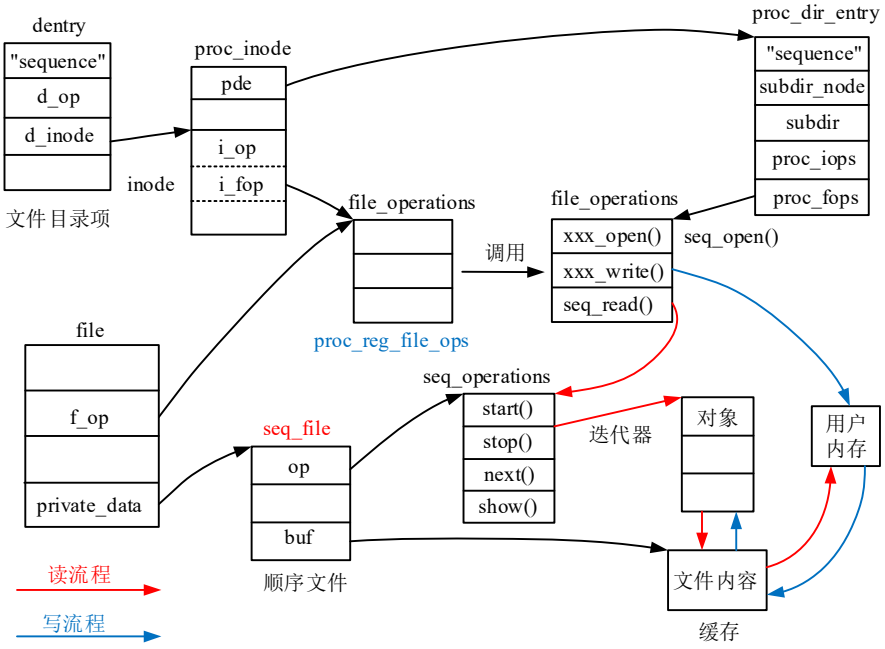
7.11.7 顺序文件

顺序文件提供了使用户进程以文件形式读取/写入内核数据的机制。文件内容通常保存在内核数据结构实例（对象）中。用户进程读取顺序文件内容时，内核将从对象（可能需要遍历多个对象）中获取数据，传递给用户进程。用户进程写入数据时，内核将数据写入指定对象。

顺序文件相关头文件为/include/linux/seq_file.h，实现代码位于/fs/seq_file.c 文件内。

1 概述

下图示意了顺序文件的实现框架，假设在 proc 文件系统中创建了一个名称为"sequence"的文件（其它文件系统中也可以将文件设为顺序文件），定义了文件关联的文件操作结构 file_operations 实例：



现在要以顺序文件的形式对其进行访问，需要定义一个 seq_operations 结构体实例。seq_operations 结构体用于实现一个迭代器，用于从多个内核数据结构实例（对象）中读取数据写入缓存区。

文件关联 file_operations 实例中 xxx_open()函数需要调用通用函数 **seq_open()**，seq_operations 实例指

针须作为seq_open()函数的一个参数。在打开文件时，seq_open()函数为file实例创建表示顺序文件的seq_file实例（赋予 file.private_data 成员），并将 seq_operations 实例指针赋于 op 成员。seq_file 实例还关联一个缓存，用于缓存文件内容。

文件关联 file_operations 实例中读函数通常为（或调用）通用函数 seq_read()函数，seq_read()函数通过由 seq_operations 实例表示的迭代器遍历内核对象，从对象中获取数据写入缓存，然后将缓存中数据复制到用户内存。

file_operations 实例中写函数将用户数据写入缓存，然后将缓存中数据写入指定对象。

2 数据结构

顺序文件由 seq_file 结构体表示，定义如下（/include/linux/seq_file.h）：

```
struct seq_file {
    char *buf;        /*指向保存文件内容的缓存*/
    size_t size;      /*缓存大小*/
    size_t from;      /*从缓存中读取数据的起始偏移量*/
    size_t count;     /*缓存中现有数据大小*/
    size_t pad_until;
    loff_t index;             /*下次对象迭代的起始位置，文件位置*/
    loff_t read_pos;          /*上次读操作结束位置*/
    u64 version;
    struct mutex lock;
    const struct seq_operations *op;        /*迭代器*/
    int poll_event;
#ifdef CONFIG_USER_NS
    struct user_namespace *user_ns;
#endif
    void *private;
};
```

seq_file 结构体中 op 成员指向 seq_operations 结构体，用于实现迭代器，结构体定义如下：

```
struct seq_operations {
    void * (*start) (struct seq_file *m, loff_t *pos);
                                    /*开始迭代，pos 表示文件位置，返回第一个迭代对象指针*/
    void (*stop) (struct seq_file *m, void *v);          /*停止迭代/
    void * (*next) (struct seq_file *m, void *v, loff_t *pos);
                                /*获取下一个对象，v 表示上一个迭代对象，返回下一个迭代对象指针*/
    int (*show) (struct seq_file *m, void *v);       /*从迭代对象中获取数据写入缓存，v 指向对象*/
};
```

seq_file 结构体中函数功能简介如下：

●**start**：开始迭代，参数传递一个文件当前位置值，函数返回第一个迭代对象指针。

●**show**：从对象中获取数据，将数据写入缓存区。

●**next**：获取下一个迭代对象。

●**stop**：停止迭代，释放空间等。

3 接口函数

在/fs/seq_file.c 文件内定义了顺序文件操作的接口函数，供特定于文件的 file_operations 实例引用（调用）。

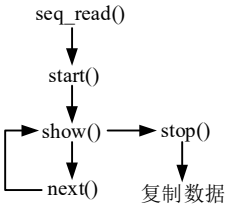
●int **seq_open**(struct file *file, const struct seq_operations ***op**)：打开文件操作，主要是创建并设置 seq_file 实例，将 seq_file 实例指针赋予 file.private_data 成员，op 参数指向由文件定义的 seq_operations 实例。

●loff_t **seq_lseek**(struct file *file, loff_t offset, int whence)：设置文件位置。

●ssize_t **seq_read**(struct file *file, char __user *buf, size_t size, loff_t *ppos)：读文件操作函数，调用迭代器，生成数据写入缓存区，然后从缓存区中复制数据至用户内存。

●int **seq_write**(struct seq_file *seq, const void *data, size_t len)：只是将用户数据写入顺序文件缓存，没有写入对象。file_operations 实例通常自定义 write()函数，而不是调用 seq_write()函数。

通用函数 seq_read()调用迭代器从对象中获取数据，填充至缓存，最后复制缓存数据至用户内存，函数调用关系如下图所示：



start()函数用于获取第一个迭代对象，show()函数从对象中获取数据填充至缓存。如果一个对象获取的数据不够，需要迭代多个对象，则循环调用 next()和 show()函数，获取多个对象中的数据，然后调用 stop()函数停止迭代，最后复制缓存中数据至用户空间。

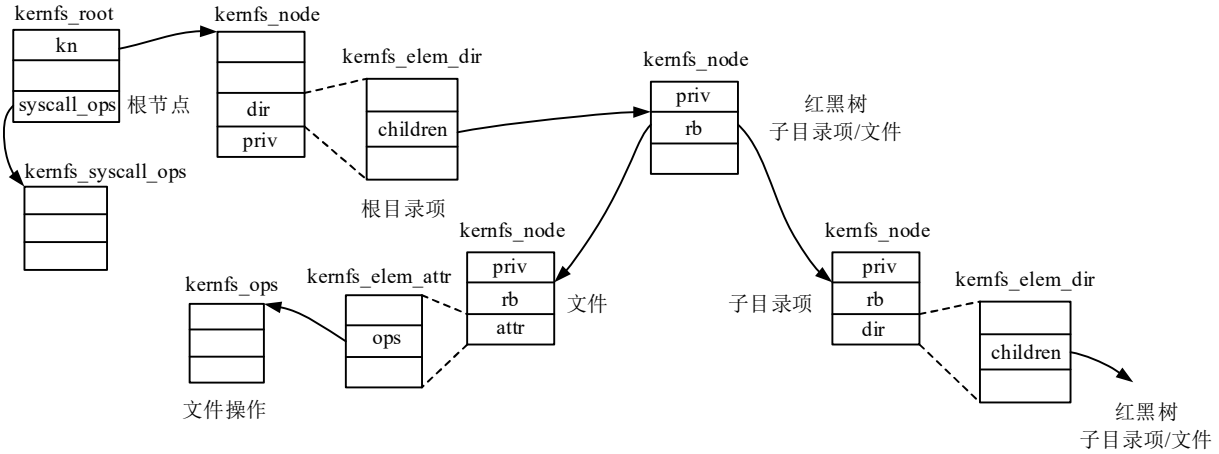
7.12 kernfs 文件系统

kernfs 文件系统与 proc 文件系统一样，是由内核数据结构实例构成的文件系统，它由 kernfs_node 结构体实例组成。kernfs 文件系统并不是一个完整的文件系统，它只是提供构建文件系统的基础组件及相关的接口函数，使用 kernfs 文件系统构成真正文件系统时需要定义并注册文件系统类型 file_system_type 实例。可以把 kernfs 文件系统看成是一个文件系统的内核（组组架构），需要构成真正的文件系统还需要对其进行封装，真正文件系统使用的数据结构实例关联到 kernfs_node 实例。

本节介绍 kernfs 文件系统的结构，以及相关的接口函数，下节介绍使用 kernfs 文件系统构建的 sysfs 文件系统。kernfs 文件系统相关代码位于/fs/kernfs/目录下，内核需选择 KERNFS 配置选项。

7.12.1 组织结构

请读者先看下图所示的 kernfs 文件系统组织结构：



kernfs 文件系统中的每个普通目录项和文件都由 kernfs_node 结构体表示，kernfs_root 结构体表示根节点，其指向的 kernfs_node 实例表示根目录项。

表示普通目录项的 kernfs_node 实例中其 dir 成员为 kernfs_elem_dir 结构体，其中包含一个红黑树根节点成员，红黑树管理其下子目录项和文件目录项。子目录项和文件目录项通过 rb 成员添加到父目录项管理的红黑树中。

表示文件的 kernfs_node 实例中其 attr 成员为 kernfs_elem_attr 结构体，包含了文件操作相关的信息。

另外，kernfs 文件系统也支持符号链接，kernfs_node 实例也可以表示符号链接。

7.12.2 目录项与文件

kernfs 文件系统中根节点由 kernfs_root 结构体表示，目录项和文件（符号链接）由 kernfs_node 结构体表示。

1 数据结构

kernfs_root 结构体用于表示 kernfs 文件系统的根节点，定义在/include/linux/kernfs.h 头文件：

```
struct kernfs_root {
    struct kernfs_node  *kn;      /*指表示根目录项的 kernfs_node 实例*/
    unsigned int        flags;          /*根节点标记*/
    struct ida      ino_ida;            /*管理 kernfs_node 编号，/include/linux/idr.h*/
    struct kernfs_syscall_ops   *syscall_ops;       /*系统调用关联的操作函数*/
    struct list_head    supers;         /*链接 kernfs_super_info 实例*/
    wait_queue_head_t       deactivate_waitq;       /*进程等待队列*/
};
```

kernfs_root 结构体主要成员简介如下：

●**kn**：指向表示根目录项的 kernfs_node 实例，结构体定义见下文。

●**flags**：根节点标记，取值定义在/include/linux/kernfs.h 头文件：

```
enum kernfs_root_flag {
    KERNFS_ROOT_CREATE_DEACTIVATED          = 0x0001,     /**/
    KERNFS_ROOT_EXTRA_OPEN_PERM_CHECK   = 0x0002,     /**/
};
```

●**syscall_ops**：kernfs_syscall_ops 结构体指针，结构体定义在/include/linux/kernfs.h 头文件：

```
struct kernfs_syscall_ops {
```

```
    int (*remount_fs)(struct kernfs_root *root, int *flags, char *data);      /*挂载文件系统*/
    int (*show_options)(struct seq_file *sf, struct kernfs_root *root);

    int (*mkdir)(struct kernfs_node *parent, const char *name,   umode_t mode);    /*创建目录项*/
    int (*rmdir)(struct kernfs_node *kn);          /*删除目录项*/
    int (*rename)(struct kernfs_node *kn, struct kernfs_node *new_parent, const char *new_name);
                                                                        /*重命名*/
};
```

kernfs_syscall_ops 结构体内包含目录项、文件操作系统调用中将会调用的函数指针，例如创建目录项、删除目录项，重命名文件等。

kernfs_node 结构体用于构成 kernfs 文件系统，kernfs_node 实例称为 kernfs 文件系统节点，结构体定义在/include/linux/kernfs.h 头文件：

```
struct kernfs_node {
    atomic_t        count;          /*引用计数*/
    atomic_t        active;         /*节点表示文件被正确打开，可操作*/
#ifdef CONFIG_DEBUG_LOCK_ALLOC
    struct lockdep_map dep_map;
#endif
    struct kernfs_node *parent;     /*指向父节点*/
    const char          *name;      /*节点名称，目录项（文件）名称*/
    struct rb_node      rb;         /*红黑树节点，添加到父节点管理的红黑树中*/
    const void          *ns;        /*命名空间标记*/
    unsigned int        hash;       /*散列值，用作红黑树中键值*/
    union {                         /*联合体，表示目录项或符号链接或文件*/
        struct kernfs_elem_dir      dir;        /*本节点表示目录项*/
        struct kernfs_elem_symlink  symlink;    /*本节点表示符号链接*/
        struct kernfs_elem_attr     attr;       /*本节点表示文件*/
    };
    void                *priv;      /*具体文件系统私有数据，如 sysfs 文件系统中指向 kobject 实例*/
    unsigned short      flags;      /*标记*/
    umode_t             mode;       /*访问权限*/
    unsigned int        ino;        /*节点编号*/
    struct kernfs_iattrs  *iattr;   /*节点属性，访问权限、时间戳等，/fs/kernfs/kernfs-internal.h*/
};
```

kernfs_node 结构体主要成员简介如下：

●**parent**：指向父节点。

●**rb**：红黑树节点成员，父目录项通过红黑树管理其子目录项，hash 成员值作为添加到红黑树的键值。

●**priv**：使用 kernfs 构成具体文件系统时，指向其私有数据，如 sysfs 文件系统中指向 kobject 实例或属性 attribute 实例。

●**flags**：标记成员，低 4 位表示节点类型，如目录项、符号链接、文件等，其余位表示节点属性。标记成员取值定义在/include/linux/kernfs.h 头文件：

```
enum kernfs_node_type {
```

```
    KERNFS_DIR          = 0x0001,       /*目录项*/
    KERNFS_FILE         = 0x0002,       /*普通文件*/
    KERNFS_LINK         = 0x0004,       /*符号链接*/
};

#define   KERNFS_TYPE_MASK     0x000f              /*节点类型标记位掩码*/
#define   KERNFS_FLAG_MASK     ~KERNFS_TYPE_MASK
```

其余标记位的定义如下（/include/linux/kernfs.h）：

```
enum kernfs_node_flag {
    KERNFS_ACTIVATED = 0x0010,      /*节点激活*/
    KERNFS_NS           = 0x0020,
    KERNFS_HAS_SEQ_SHOW    = 0x0040,
    KERNFS_HAS_MMAP      = 0x0080,
    KERNFS_LOCKDEP       = 0x0100,
    KERNFS_SUICIDAL      = 0x0400,
    KERNFS_SUICIDED      = 0x0800,
    KERNFS_EMPTY_DIR     = 0x1000,
};
```

●**联合体**：联合体成员，对于目录项、符号链接、文件，联合体具有不同的解释。

若节点表示的是目录项，则联合体解释为 kernfs_elem_dir 结构体，定义如下：

```
struct kernfs_elem_dir {
    unsigned long       subdirs;        /*子目录项数量*/
    struct rb_root      children;       /*红黑树根节点，管理子目录项*/
    struct kernfs_root  *root;          /*指向文件系统根节点*/
};
```

若节点表示的是符号链接，则联合体解释为 kernfs_elem_symlink 结构体，定义如下：

```
struct kernfs_elem_symlink {
    struct kernfs_node  *target_kn;     /*指向链接对象的 kernfs_node 实例*/
};
```

若节点表示的是文件，则联合体解释为 kernfs_elem_attr 结构体，定义如下：

```
struct kernfs_elem_attr {
    const struct kernfs_ops   *ops;         /*创建文件节点时，由创建者赋值*/
    struct kernfs_open_node *open;
                        /*指向 kernfs_open_node 实例，表示被打开的文件，/fs/kernfs/file.c*/
    loff_t          size;                   /*文件大小*/
    struct kernfs_node  *notify_next;
};
```

内核在/fs/kernfs/mount.c 文件内定义了 kernfs 文件系统初始化函数（由 mnt_init()函数调用），主要工作是为 kernfs_node 结构体创建 slab 缓存，定义如下：

```
void __init kernfs_init(void)
{
    kernfs_node_cache = kmem_cache_create("kernfs_node_cache",  sizeof(struct kernfs_node),  \
                                          0, SLAB_PANIC, NULL);
}
```

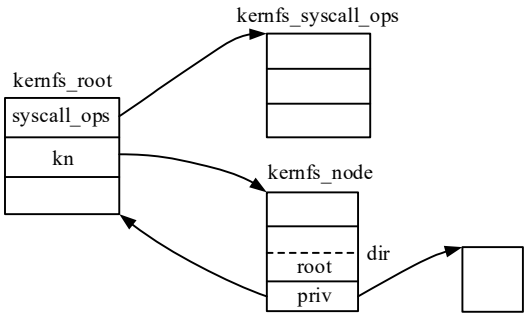
2 创建根节点

使用 kernfs 的具体文件系统类型代码首先需要创建根节点 kernfs_root，然后才能执行挂载操作。创建 kernfs 文件系统根节点 kernfs_root 实例的函数定义在/fs/kernfs/dir.c 文件内，代码如下：

```
struct kernfs_root *kernfs_create_root(struct kernfs_syscall_ops *scops,unsigned int flags, void *priv)
/*scops：kernfs_syscall_ops 实例由具体文件系统类型定义，flags：根节点标记，priv：私有数据*/
{
    struct kernfs_root *root;
    struct kernfs_node *kn;

    root = kzalloc(sizeof(*root), GFP_KERNEL);       /*创建 kernfs_root 实例*/
    ...
    ida_init(&root->ino_ida);       /*初始化 ida 结构体成员，用于分配节点编号*/
    INIT_LIST_HEAD(&root->supers);

    kn = __kernfs_new_node(root, "", S_IFDIR | S_IRUGO | S_IXUGO,KERNFS_DIR);
                                        /*创建并设置 kernfs_node 实例（目录项），/fs/kernfs/dir.c*/
    ...
    kn->priv = priv;       /*私有数据*/
    kn->dir.root = root;

    root->syscall_ops = scops;        /*kernfs_syscall_ops 实例*/
    root->flags = flags;        /*标记*/
    root->kn = kn;              /*指向 kernfs_node 实例*/
    init_waitqueue_head(&root->deactivate_waitq);

    if (!(root->flags & KERNFS_ROOT_CREATE_DEACTIVATED))
        kernfs_activate(kn);

    return root;       /*返回 kernfs_root 实例指针*/
}
```

kernfs_create_root()函数参数 scops 指向的 kernfs_syscall_ops 实例由具体文件系统类型实现，函数内首先创建 kernfs_root 实例，随后调用__kernfs_new_node()函数创建并初始化根节点对应的 kernfs_node 实例，最后对两个数据结构实例进行设置。

__kernfs_new_node()函数主要工作是从 slab 缓存中分配 kernfs_node 实例，并对其进行初始化。执行完 kernfs_create_root()函数后创建的数据结构实例组织关系如下图所示：



3 添加目录项

kernfs_create_dir_ns()函数用于创建并向 kernfs 文件系统添加表示目录项的 kernfs_node 实例，返回添加实例指针，函数定义在/fs/kernfs/dir.c 文件内：

```
struct kernfs_node *kernfs_create_dir_ns(struct kernfs_node *parent, const char *name, umode_t mode, \
                                         void *priv, const void *ns)
{
    struct kernfs_node *kn;
    int rc;

    /*创建表示目录项的 kernfs_node 实例，增加实例引用计数*/
    kn = kernfs_new_node(parent, name, mode | S_IFDIR, KERNFS_DIR);    /*/fs/kernfs/dir.c*/
    ...

    kn->dir.root = parent->dir.root;       /*初始化实例*/
    kn->ns = ns;
    kn->priv = priv;          /*私有数据*/

    /*将 kernfs_node 实例添加到父节点管理的红黑树中，/fs/kernfs/dir.c*/
    rc = kernfs_add_one(kn);
    if (!rc)
        return kn;       /*返回 kernfs_node 实例指针*/
    ...
}
```

kernfs_create_dir_ns()函数比较简单，首先调用 kernfs_new_node()函数创建并初始化 kernfs_node 实例，然后调用 kernfs_add_one()函数将 kernfs_node 实例添加到父节点管理的红黑树中，最后返回 kernfs_node 实例指针。

4 添加文件

向 kernfs 文件系统添加文件的操作与添加目录项类似，但是传递的参数有所不同，返回值依然是 kernfs_node 实例指针。添加文件函数定义在/fs/kernfs/file.c 文件内，代码如下：

```
struct kernfs_node *__kernfs_create_file(struct kernfs_node *parent, const char *name, umode_t mode,   \
        loff_t size, const struct kernfs_ops *ops, void *priv, const void *ns, struct lock_class_key *key)
/*ops：kernfs_ops 实例指针，priv：私有数据指针，这是两个需要具体文件系统类型实现的实例*/
```

```
{
    struct kernfs_node *kn;
    unsigned flags;
    int rc;

    flags = KERNFS_FILE;       /*kernfs_node 实例标记为文件*/

    kn = kernfs_new_node(parent, name, (mode & S_IALLUGO) | S_IFREG, flags);
                                                    /*创建并初始化 kernfs_node 实例*/
    ...
    kn->attr.ops = ops;         /*赋予 kernfs_ops 实例*/
    kn->attr.size = size;       /*文件大小*/
    kn->ns = ns;
    kn->priv = priv;           /*私有数据*/
    ...
    if (ops->seq_show)            /*具有顺序读文件函数指针*/
        kn->flags |= KERNFS_HAS_SEQ_SHOW;          /*以顺序文件形式读*/
    if (ops->mmap)
        kn->flags |= KERNFS_HAS_MMAP;

    rc = kernfs_add_one(kn);       /*添加到父节点管理的红黑树中*/
    ...
    return kn;           /*返回 kernfs_node 实例指针*/
}
```

添加文件操作与添加目录项最大的不同在于需要传递 kernfs_ops 实例指针参数，kernfs_ops 结构体内包含读写文件内容的函数指针（结构体定义后面再介绍）。kernfs 文件系统中文件的内容不是静态保存在 kernfs_node 实例中，而是在需要时通过 kernfs_ops 结构体中的函数从内核中读取。

创建符号链接的函数原型如下，请读者自行阅读源代码：

```
struct kernfs_node *kernfs_create_link(struct kernfs_node *parent,const char *name,
                                                    struct kernfs_node *target);
```

kernfs 文件系统还定义了其它接口函数（声明在/include/linux/kernfs.h），函数源代码请读者自行阅读。

7.12.3 挂载操作

kernfs 文件系统提供了挂载函数，供由 kernfs 构建的具体文件系统类型调用，在文件系统类型的挂载函数中调用 kernfs 文件系统的挂载函数实现挂载操作。在执行挂载前需要创建根节点 kernfs_root 实例，然后才能执行挂载操作。

1 概述

在介绍挂载函数前先看一下 kernfs_super_info 结构体的定义（/fs/kernfs/kernfs-internal.h），它表示 kernfs 文件系统的超级块信息：

```
struct kernfs_super_info {
```

```
    struct super_block  *sb;        /*超级块实例指针*/
    struct kernfs_root  *root;      /*指向文件系统根节点*/
    const void          *ns;        /*命名空间标签*/
    struct list_head    node;       /*将实例链接到 kernfs_root.super 双链表*/
};
```

kernfs_super_info 结构体主要成员简介如下：

●**sb**：指向超级块 super_block 实例。

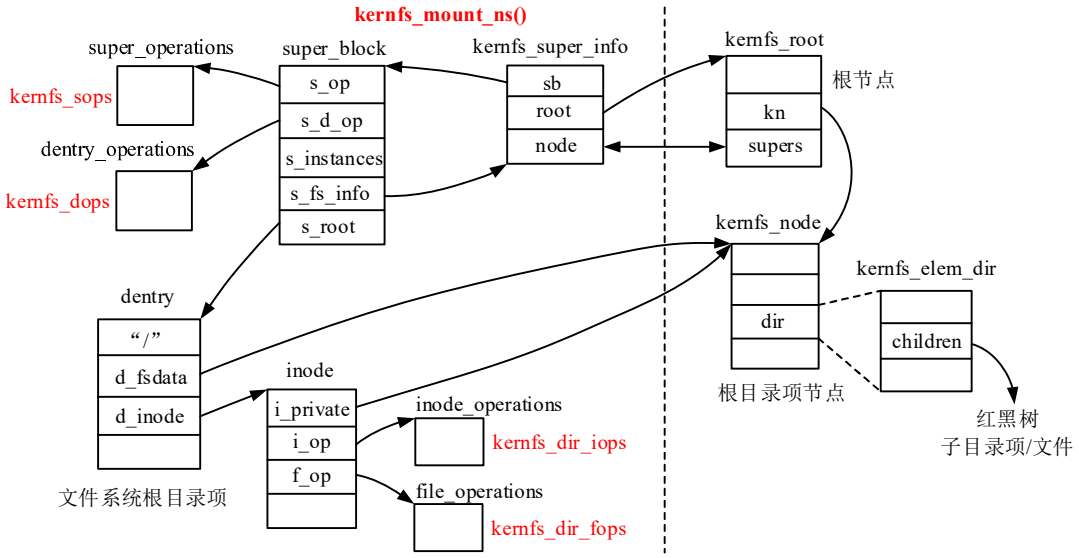
●**root**：指向根节点 kernfs_root 实例。

●**node**：双链表成员，将 kernfs_super_info 实例添加到根节点 kernfs_root.super 双链表。由于一个文件系统可以挂载到多个挂载点，因此 kernfs_root.super 双链表中可能管理多个 kernfs_super_info 实例。

在挂载函数中创建的超级块 super_block 实例 s_fs_info 成员将指向 kernfs_super_info 实例，以下宏用于由超级块 super_block 实例，获取 kernfs_super_info 实例指针。

```
#define kernfs_info(SB) ((struct kernfs_super_info *)(SB->s_fs_info))     /*super_block->s_fs_info*/
```

kernfs 文件系统定义的挂载函数为 kernfs_mount_ns()，执行后创建的数据结构实例如下图所示：



2 挂载函数

kernfs 文件系统挂载函数 kernfs_mount_ns()定义在/fs/kernfs/mount.c 文件内：

```
struct dentry *kernfs_mount_ns(struct file_system_type *fs_type, int flags,struct kernfs_root *root,  \
                               unsigned long magic,bool *new_sb_created, const void *ns)
/*root：指向根节点 kernfs_root 实例，ns：命名空间标签，magic：具体文件系统类型魔数*/
{
    struct super_block *sb;
    struct kernfs_super_info *info;
    int error;

    info = kzalloc(sizeof(*info), GFP_KERNEL);     /*分配 kernfs_super_info 实例*/
    ...
```

```
    info->root = root;      /*指向 kernfs_root 实例*/
    info->ns = ns;

    sb = sget(fs_type, kernfs_test_super, kernfs_set_super, flags, info);
                    /*创建超级块 super_block 实例，一个命名空间标签内只能挂载一次*/
    ...
    if (new_sb_created)
        *new_sb_created = !sb->s_root;      /*是否是新创建的超级块实例*/

    if (!sb->s_root) {              /*文件系统根目录项尚不存在（没有挂载过），则创建根目录项*/
        struct kernfs_super_info *info = kernfs_info(sb);

        error = kernfs_fill_super(sb, magic);
                    /*设置 super_block 实例，创建根目录项 dentry 和 inode 实例，/fs/kernfs/mount.c*/
        ...
        sb->s_flags |= MS_ACTIVE;

        mutex_lock(&kernfs_mutex);
        list_add(&info->node, &root->supers); /*kernfs_super_info 实例添加到 kernfs_root 中双链表*/
        mutex_unlock(&kernfs_mutex);
    }
    return dget(sb->s_root);    /*返回根目录项 dentry 指针*/
}
```

kernfs_mount_ns()函数内首先创建 kernfs_super_info 实例，然后创建超级块 super_block 实例，最后调用 kernfs_fill_super()函数设置超级块实例，并创建根目录项 dentry 和 inode 实例。

kernfs_fill_super()函数定义在/fs/kernfs/mount.c 文件内，代码如下：

```
static int kernfs_fill_super(struct super_block *sb, unsigned long magic)
/*sb：超级块实例指针，magic：具体文件系统类型魔数*/
{
    struct kernfs_super_info *info = kernfs_info(sb);
    struct inode *inode;
    struct dentry *root;

    info->sb = sb;
    sb->s_blocksize = PAGE_CACHE_SIZE;
    sb->s_blocksize_bits = PAGE_CACHE_SHIFT;
    sb->s_magic = magic;
    sb->s_op = &kernfs_sops;        /*超级块操作结构实例，没有定义 alloc_inode()函数*/
    sb->s_time_gran = 1;

    mutex_lock(&kernfs_mutex);
```

```
    inode = kernfs_get_inode(sb, info->root->kn);     /*创建根节点 inode 实例，/fs/kernfs/inode.c*/
    mutex_unlock(&kernfs_mutex);
    ...

    root = d_make_root(inode);  /*创建 dentry 实例，名称为“/”，并关联 inode，/fs/dcache.c*/
    ...
    kernfs_get(info->root->kn);             /*增加节点引用计数*/
    root->d_fsdata = info->root->kn;        /*建立 dentry 与 kernfs_node 实例之间关联*/
    sb->s_root = root;
    sb->s_d_op = &kernfs_dops;              /*目录项操作结构实例*/
    return 0;
}
```

kernfs_fill_super()函数比较容易理解，主要工作是初始化 super_block 实例各成员，创建并设置文件系统根目录项对应的 dentry 和 inode 实例，并建立各数据结构实例之间的关联。

kernfs_get_inode()函数用于创建 inode 实例，定义在/fs/kernfs/inode.c 文件内：

```
struct inode *kernfs_get_inode(struct super_block *sb, struct kernfs_node *kn)
{
    struct inode *inode;
    inode = iget_locked(sb, kn->ino);             /*创建 inode 实例，并赋予节点编号，/fs/inode.c*/
    if (inode && (inode->i_state & I_NEW))
        kernfs_init_inode(kn, inode);             /*初始化 inode 实例，/fs/kernfs/inode.c*/
    return inode;
}
```

kernfs_get_inode()函数中调用 iget_locked()函数直接从 inode 结构体 slab 缓存中分配实例（kernfs_sops 实例中没有定义 alloc_inode()函数），调用 kernfs_init_inode()对 inode 实例进行初始化。

kernfs_init_inode()函数定义如下：

```
static void kernfs_init_inode(struct kernfs_node *kn, struct inode *inode)
{
    kernfs_get(kn);
    inode->i_private = kn;          /*指向 kernfs_node 实例*/
    inode->i_mapping->a_ops = &kernfs_aops;     /*地址空间操作结构*/
    inode->i_op = &kernfs_iops;                 /*节点操作结构实例*/

    set_default_inode_attr(inode, kn->mode);      /*设置 inode->i_mode*/
    kernfs_refresh_inode(kn, inode);

    switch (kernfs_type(kn)) {        /*根据节点类型设置 i_op 和 i_fop 成员*/
     case KERNFS_DIR:          /*目录项节点*/
        inode->i_op = &kernfs_dir_iops;          /*节点操作结构实例，fs/kernfs/dir.c*/
        inode->i_fop = &kernfs_dir_fops;         /*文件操作结构实例*/
        if (kn->flags & KERNFS_EMPTY_DIR)
            make_empty_dir_inode(inode);
```

```
            break;
        case KERNFS_FILE:                    /*文件节点*/
            inode->i_size = kn->attr.size;
            inode->i_fop = &kernfs_file_fops;        /*文件操作结构实例*/
            break;
        case KERNFS_LINK:                    /*符号链接节点*/
            inode->i_op = &kernfs_symlink_iops;
            break;
          default:
            BUG();
    }
    unlock_new_inode(inode);
}
```

7.12.4 文件操作

打开操作是进程对文件进行操作的基础，在下图中假设 kernfs 文件系统被挂载到了/sys/目录下，现在需要打开/sys/abc 文件，并对其进行操作。

在前面介绍的挂载操作中，kernfs 根目录项关联的节点操作结构实例为 kernfs_dir_ops，其中的 lookup()函数在父目录项管理的红黑树中查找指定名称的子目录项，并为其创建和设置 inode 实例。

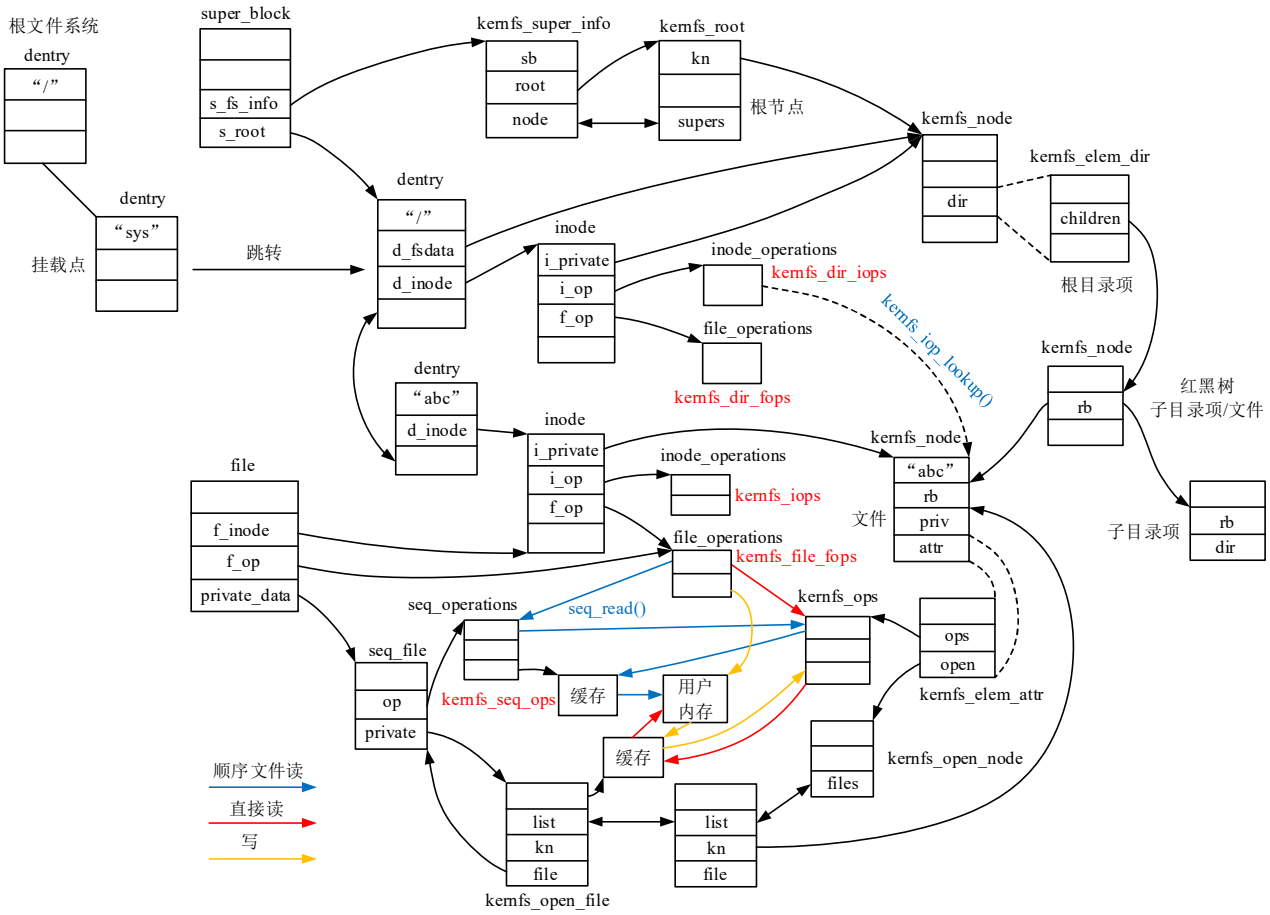
在下图中，在/sys/目录项的红黑树中查找到名称为“abc”的节点后，将为其创建和设置 inode 实例，并建立与 dentry 实例之间的关联。

在创建文件对应的 inode 实例时，其关联的文件操作结构实例设为 kernfs_file_ops，在打开操作的最后将调用此实例中的 open()函数。open()函数将为打开文件创建顺序文件 seq_file 实例，以及 kernfs_open_file 实例。kernfs_open_file 实例表示对文件的一次打开操作，kernfs_node 实例中 kernfs_elem_attr 结构体成员的 open 成员，指向 kernfs_open_node 结构体实例，表示被打开的文件，其中的双链表成员管理着 kernfs_open_file 实例，每个实例对应一次打开操作。

如果添加文件时传递的 kernfs_ops 实例中定义了 seq_show()函数，表示文件是一个顺序文件，文件操作结构 kernfs_file_ops 实例中的读函数将按照读顺序文件进行。迭代器 seq_operations 实例 kernfs_seq_ops 中的函数将调用 kernfs_ops 实例中的相应函数完成读数据至缓存区的操作，缓存中数据最后复制到用户空间。

如果 kernfs_ops 实例没有定义 seq_show()函数，则对文件进行直接读操作。kernfs_file_ops 实例中读函数直接调用 kernfs_ops 实例中的 read()函数读数据至 kernfs_open_file 实例关联的缓存，然后从缓存复制数据至用户空间。

kernfs_file_ops 实例中的写函数将复制用户数据至 kernfs_open_file 实例关联的缓存，然后调用 kernfs_ops 实例中的 write()函数将缓存中数据写入内核。



1 打开文件

打开文件操作主要包含各路径分量（目录项）的搜索，以及对末尾分量（文件）进行的打开操作。

■搜索目录项

由前面的挂载函数可知，在 kernfs_get_inode()函数中，根目录项（包括其它的普通目录项）inode 实例关联的节点操作结构实例和文件操作结构实例分别赋值为 **kernfs_dir_iops** 和 kernfs_dir_fops 实例。

kernfs_dir_iops 实例中的 lookup()函数用于搜索其下的子目录项，并创建/设置 inode 实例，此实例定义在/fs/kernfs/dir.c 文件内：

```
const struct inode_operations kernfs_dir_iops = {
    .lookup        = kernfs_iop_lookup,        /*搜索子目录项函数*/
    ...
};
```

搜索子目录项的 kernfs_iop_lookup()函数定义在/fs/kernfs/dir.c 文件内，代码如下：

```
static struct dentry *kernfs_iop_lookup(struct inode *dir,struct dentry *dentry,unsigned int flags)
/*dir：父目录项对应 inode 实例，dentry：搜索目录项，flags：标记*/
{
    struct dentry *ret;
    struct kernfs_node *parent = dentry->d_parent->d_fsdata;    /*父目录项 kernfs_node 实例*/
```

```
    struct kernfs_node *kn;
    struct inode *inode;
    const void *ns = NULL;

    mutex_lock(&kernfs_mutex);

    if (kernfs_ns_enabled(parent))
        ns = kernfs_info(dir->i_sb)->ns;       /*网络命名空间*/

    kn = kernfs_find_ns(parent, dentry->d_name.name, ns);    /*fs/kernfs/dir.c*/
                /*在父目录项 kernfs_node 实例管理的红黑树中查找名称为 name 的 kernfs_node 实例*/

    ...     /*错误处理*/
    kernfs_get(kn);
    dentry->d_fsdata = kn;     /*搜索目录项关联查找到的 kernfs_node 实例*/

    inode = kernfs_get_inode(dir->i_sb, kn);     /*创建并初始化 inode 实例，见上文，/fs/kernfs/inode.c*/
    ...
    ret = d_splice_alias(inode, dentry);   /*关联 dentry 和 inode 实例，/fs/dcache.c*/
 out_unlock:
    mutex_unlock(&kernfs_mutex);
    return ret;          /*返回 dentry 实例指针*/
}
```

搜索目录项的操作比较简单，在父目录项对应的 kernfs_node 实例中，搜索其管理的红黑树，找到名称为 name 的 kernfs_node 实例，创建对应的 inode 实例并初始化，建立 dentry 实例与 inode、kernfs_node 实例之间的关联，最后返回 dentry 实例指针。

在 kernfs_get_inode()函数中，对于普通目录项 inode 实例关联的节点操作结构和文件操作结构实例分别赋为 kernfs_dir_iops 和 kernfs_dir_fops。对于文件 inode 实例关联的节点操作结构和文件操作结构实例分别为 kernfs_iops 和 kernfs_file_fops 实例。符号链接的节点操作结构实例为 kernfs_symlink_iops。

下面看一下普通文件关联的文件操作结构 file_operations 实例的定义，如下所示（/fs/kernfs/file.c）：

```
const struct file_operations kernfs_file_fops = {
    .read       = kernfs_fop_read,      /*读操作函数，/fs/kernfs/file.c*/
    .write      = kernfs_fop_write,     /*写操作函数，/fs/kernfs/file.c*/
    .llseek     = generic_file_llseek,      /*设置文件当前位置*/
    .mmap       = kernfs_fop_mmap,
    .open       = kernfs_fop_open,          /*打开文件函数，/fs/kernfs/file.c*/
    .release    = kernfs_fop_release,       /*释放文件函数，释放各数据结构实例，/fs/kernfs/file.c*/
    .poll       = kernfs_fop_poll,
};
```

下文将介绍其中的打开操作和读写操作函数，其它函数请读者自行阅读源代码。

■打开操作

在 open()系统调用的最后阶段，将调用 file 实例关联 file_operations 实例的 open()函数，完成特定于文

件的打开操作。对于 kernfs 文件系统中的文件，kernfs_file_fops 实例中的 open()函数为 kernfs_fop_open()。在介绍 kernfs_fop_open()函数的实现前，先介绍打开操作中涉及的几个数据结构。

在 kernfs_node 结构体中，如果实例表示的是文件，则其中的联合体成员解释为 kernfs_elem_attr 结构体成员 attr，kernfs_elem_attr 结构体定义如下（/include/linux/kernfs.h）：

```
struct kernfs_elem_attr {
    const struct kernfs_ops   *ops;        /*创建文件节点时，传递的 kernfs_ops 实例指针*/
    struct kernfs_open_node*open;     /*表示打开文件的 kernfs_open_node 实例，/fs/kernfs/file.c*/
    loff_t            size;                 /*文件大小*/
    struct kernfs_node  *notify_next;
};
```

kernfs_elem_attr 结构体主要成员简介如下：

●**ops**：kernfs_ops 结构体指针，在向 kernfs 文件系统添加文件时，需传递 kernfs_ops 实例指针参数。kernfs_ops 结构体定义如下（/include/linux/kernfs.h）：

```
struct kernfs_ops {
     /*顺序文件迭代器，按顺序文件读 kernfs 文件时需要*/
    int (*seq_show)(struct seq_file *sf, void *v);
    void *(*seq_start)(struct seq_file *sf, loff_t *ppos);
    void *(*seq_next)(struct seq_file *sf, void *v, loff_t *ppos);
    void (*seq_stop)(struct seq_file *sf, void *v);

    /*直接读写文件*/
    ssize_t (*read)(struct kernfs_open_file *of, char *buf, size_t bytes,loff_t off);
                        /*生成数据写入 kernfs_open_file 实例中缓存，而后被复制到用户空间*/
    size_t    atomic_write_len;    /*缓存长度*/
    bool prealloc;          /*1：创建 kernfs_open_file 实例时需要分配缓存，0：读写操作时分配*/
    ssize_t (*write)(struct kernfs_open_file *of, char *buf, size_t bytes, loff_t off);
                                    /*将用户写入 kernfs_open_file 实例缓存中数据写入内核*/
    int (*mmap)(struct kernfs_open_file *of, struct vm_area_struct *vma);      /*文件映射*/

  #ifdef CONFIG_DEBUG_LOCK_ALLOC
    struct lock_class_key    lockdep_key;
  #endif
};
```

kernfs_ops 结构体中包含 kernfs 文件读写操作函数指针，读文件有两种形式，即按顺序文件读和直接读，写文件只有一种形式，即直接写。

如果 kernfs_ops 实例中定义了 seq_show()函数，则以顺序文件的形式读文件（调用实例中定义的迭代器函数），否则调用 read()函数进行直接读。写操作则直接调用 write()函数，直接写。

对于直接读和写操作，将采用后面介绍的 kernfs_open_file 实例关联的缓存暂存数据，顺序读则同前面介绍的顺序文件读操作，采用 seq_file 实例关联的缓存。

●**open**：kernfs_open_node 结构体指针，结构体表示一个被打开的文件（节点）。kernfs_open_node 结构体定义在/fs/kernfs/file.c 文件内：

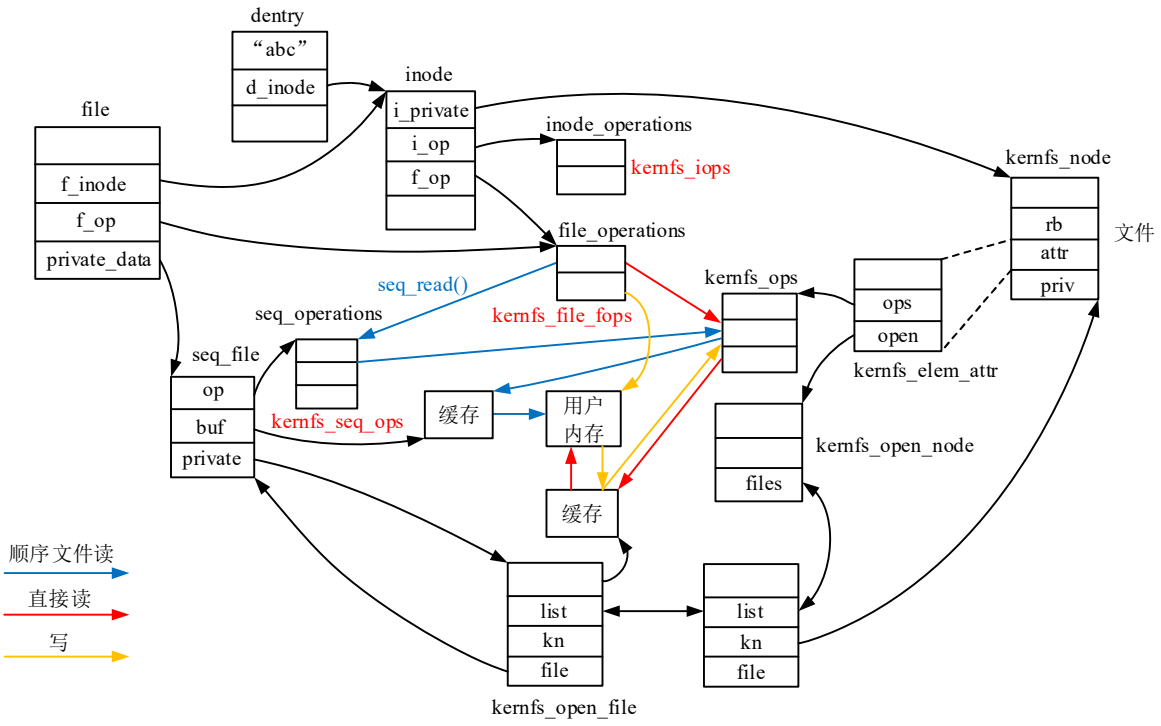
```
struct kernfs_open_node {
    atomic_t        refcnt;      /*引用计数*/
    atomic_t        event;
    wait_queue_head_t  poll;     /*进程等待队列*/
    struct list_head      files;     /*链接 kernfs_open_file 实例，每个实例表示一次打开操作*/
};
```

kernfs 文件系统中文件可能被多个进程同时打开，文件是打开的则创建一个 kernfs_open_node 实例，文件每被进程打开一次将创建一个 kernfs_open_file 实例，表示一次打开操作，kernfs_open_node.files 双链表管理文件每次打开时创建的 kernfs_open_file 实例。

kernfs_open_file 结构体定义在/include/linux/kernfs.h 头文件：

```
struct kernfs_open_file {
    struct kernfs_node    *kn;        /*指向 kernfs_node 实例*/
    struct file   *file;        /*指向进程文件 file 实例*/
    void          *priv;            /*私有数据*/
    struct mutex      mutex;
    int     event;
    struct list_head        list;      /*将实例链接到 kernfs_open_node.files 双链表*/
    char     *prealloc_buf;         /*缓存区，打开文件时分配的，用于直接读写*/

    size_t   atomic_write_len;     /*写操作的长度，分配缓存大小*/
    bool          mmapped;
    const struct vm_operations_struct    *vm_ops;      /*进程虚拟内存域操作结构*/
};
```

下图示意了 kernfs_file_fops 实例中 open()函数 **kernfs_fop_open()**创建的数据结构实例，以及读写文件操作的流程：



kernfs_fop_open()函数内将为 file 实例创建并设置 seq_file、kernfs_open_file、kernfs_open_node（如果需要）实例。

如果 kernfs_ops 中定义了 seq_show()函数，则表示以顺序文件形式读文件，seq_file 实例关联的 seq_operations 结构体实例设为 **kernfs_seq_ops**，否则为 NULL。

顺序读操作中，kernfs_seq_ops 实例中函数将调用 kernfs_ops 实例中相应的迭代器函数，完成顺序读操作（读取的数据来自 kernfs_node.priv 关联的私有数据结构）。

直接读操作将使用 kernfs_open_file 实例关联的缓存，kernfs_ops 实例中读函数从 kernfs_node 实例关联的私有数据（对象）中获取数据写至缓存，然后复制到用户内存。

写操作将数据写入 kernfs_open_file 实例关联的缓存，然后调用 kernfs_ops 实例中写函数将缓存中数据写入 kernfs_node 实例关联的私有数据（对象）中。

kernfs_file_fops 实例中的 open()函数 kernfs_fop_open()定义如下（/fs/kernfs/file.c）：

```
static int kernfs_fop_open(struct inode *inode, struct file *file)
{
    struct kernfs_node *kn = file->f_path.dentry->d_fsdata;      /*kernfs_node 实例*/
    struct kernfs_root *root = kernfs_root(kn);          /*根节点*/
    const struct kernfs_ops *ops;
    struct kernfs_open_file *of;
    bool has_read, has_write, has_mmap;
    int error = -EACCES;

    if (!kernfs_get_active(kn))
        return -ENODEV;

    ops = kernfs_ops(kn);      /*kn->attr.ops，kernfs_ops 实例*/

    has_read = ops->seq_show || ops->read || ops->mmap;      /*是否具有读操作函数*/
    has_write = ops->write || ops->mmap;                     /*是否具有写操作函数*/
    has_mmap = ops->mmap;                        /*具有映射函数*/
    /*检查标记*/
    if (root->flags & KERNFS_ROOT_EXTRA_OPEN_PERM_CHECK) {
        if ((file->f_mode & FMODE_WRITE) &&
            (!(inode->i_mode & S_IWUGO) || !has_write))
            goto err_out;

        if ((file->f_mode & FMODE_READ) &&
            (!(inode->i_mode & S_IRUGO) || !has_read))
            goto err_out;
    }

    error = -ENOMEM;
    of = kzalloc(sizeof(struct kernfs_open_file), GFP_KERNEL);
                        /*创建 kernfs_open_file 实例，每打开一次创建一个实例*/
```

```
    ...

    if (has_mmap)
        mutex_init(&of->mutex);
    else
        mutex_init(&of->mutex);

    of->kn = kn;        /*设置 kernfs_open_file 实例*/
    of->file = file;
    of->atomic_write_len = ops->atomic_write_len;

    error = -EINVAL;
    if (ops->prealloc && ops->seq_show)       /*顺序文件读，不预先分配缓存*/
        goto err_free;
    if (ops->prealloc) {            /*需要预先分配缓存，直接读*/
        int len = of->atomic_write_len ?: PAGE_SIZE;
        of->prealloc_buf = kmalloc(len + 1, GFP_KERNEL);     /*分配缓存*/
        ...
    }
    if (ops->seq_show)
        error = seq_open(file, &kernfs_seq_ops);        /*kernfs_seq_ops 为 seq_operations 实例*/
                                /*创建 seq_file 实例，kernfs_seq_ops 实例定义在/fs/kernfs/file.c*/
    else
        error = seq_open(file, NULL); /*创建 seq_file 实例，seq_operations 实例为 NULL，/fs/seq_file.c*/
    ...     /*错误处理*/
    ((struct seq_file *)file->private_data)->private = of; /*seq_file.private 指向 kernfs_open_file 实例*/

    if (file->f_mode & FMODE_WRITE)
        file->f_mode |= FMODE_PWRITE;

    error = kernfs_get_open_node(kn, of);         /*/fs/kernfs/file.c*/
                                /*将 kernfs_open_file 实例添加到 kernfs_open_node 实例中 files 双链表，
                                 *若 kernfs_open_node 实例尚不存在则先创建。*/
    ...
    kernfs_put_active(kn);      /*/fs/kernfs/dir.c*/
    return 0;
    ...
}
```

2 读写文件

打开的 kernfs 文件系统中文件，其关联的 file_operations 实例为 kernfs_file_fops，下面看一下此实例中定义的读写函数（/fs/kernfs/file.c）：

```
const struct file_operations kernfs_file_fops = {
```

```
    .read       = kernfs_fop_read,          /*读操作函数*/
    .write      = kernfs_fop_write,         /*写操作函数*/
    ...
};
```

■读操作函数

读操作函数 kernfs_fop_read()定义如下（/fs/kernfs/file.c）：

```
static ssize_t kernfs_fop_read(struct file *file, char __user *user_buf, size_t count, loff_t *ppos)
{
    struct kernfs_open_file *of = kernfs_of(file);     /*获取 kernfs_open_file 实例，seq_file ->private*/

    if (of->kn->flags & KERNFS_HAS_SEQ_SHOW)   /*kernfs_ops 实例定义了 seq_show()函数*/
        return   seq_read(file, user_buf, count, ppos);   /*顺序文件读操作函数，/fs/seq_file.c*/
    else
        return   kernfs_file_direct_read(of, user_buf, count, ppos);
                                        /*直接读操作，见下文，/fs/kernfs/file.c*/
}
```

顺序文件的读操作前面介绍过了（见上一节），这里不再重复了。如果不是按顺序文件读，则调用函数 kernfs_file_direct_read()执行直接读操作，函数定义如下：

```
static ssize_t kernfs_file_direct_read(struct kernfs_open_file *of,   char __user *user_buf,   \
                                                                        size_t count, loff_t *ppos)
{
    ssize_t len = min_t(size_t, count, PAGE_SIZE);      /*读数据长度最大不能超过一页*/
    const struct kernfs_ops *ops;
    char *buf;

    buf = of->prealloc_buf;          /*缓存指针*/
    if (!buf)                        /*如果创建 kernfs_open_file 实例时没有分配缓存*/
        buf = kmalloc(len, GFP_KERNEL);     /*分配缓存*/
    if (!buf)
        return -ENOMEM;

    mutex_lock(&of->mutex);
    ...

    of->event = atomic_read(&of->kn->attr.open->event);
    ops = kernfs_ops(of->kn);       /*从 kernfs_open_file->kn 获取 kernfs_ops 实例*/
    if (ops->read)
        len = ops->read(of, buf, len, *ppos);
                    /*调用 kernfs_ops 实例中的 read()函数，生成数据写入缓存*/
    else
        len = -EINVAL;
```

```
    if (len < 0)
        goto out_unlock;

    if (copy_to_user(user_buf, buf, len)) {      /*将缓存中数据复制到用户空间*/
        ...
    }

    *ppos += len;      /*文件位置后移 len 字节*/

 out_unlock:
     kernfs_put_active(of->kn);
     mutex_unlock(&of->mutex);
 out_free:
     if (buf != of->prealloc_buf)     /*释放分配的缓存*/
         kfree(buf);
    return len;
}
```

直接读文件操作函数判断创建 kernfs_open_file 实例时是否预先分配了缓存（打开操作时分配），如果没有则分配缓存，然后调用 kernfs_ops 实例中的 read()函数生成数据写入缓存，最后将缓存中数据复制到用户空间。如果缓存是在读操作中分配的，读操作完成后要释放缓存。

■写操作函数

写操作函数 kernfs_fop_write()定义如下（/fs/kernfs/file.c）：

```
static ssize_t kernfs_fop_write(struct file *file, const char __user *user_buf,size_t count, loff_t *ppos)
{
    struct kernfs_open_file *of = kernfs_of(file);        /*kernfs_open_file 实例*/
    const struct kernfs_ops *ops;
    size_t len;
    char *buf;

    if (of->atomic_write_len) {
        len = count;
        if (len > of->atomic_write_len)
            return -E2BIG;
    } else {
        len = min_t(size_t, count, PAGE_SIZE);
    }

    buf = of->prealloc_buf;        /*预先分配的缓存*/
    if (!buf)
        buf = kmalloc(len + 1, GFP_KERNEL);     /*没有预先分配的缓存，在此分配临时缓存*/
    ...
```

```
        if (copy_from_user(buf, user_buf, len)) {        /*复制用户数据至缓存*/
            ...
        }
        buf[len] = '\0';        /*最后字节为 0*/

        ops = kernfs_ops(of->kn);        /*kernfs_ops 实例*/
        if (ops->write)
            len = ops->write(of, buf, len, *ppos);
                                        /*调用 kernfs_ops.write()函数，将缓存中数据写入内核（对象）*/
        else
            len = -EINVAL;

        if (len > 0)
            *ppos += len;        /*修改文件当前位置*/
        ...
    out_free:
        if (buf != of->prealloc_buf)        /*释放写操作中临时分配的缓存*/
            kfree(buf);
        return len;        /*返回写数据长度（字节数）*/
    }
```

写操作函数比较好理解，首先是将用户数据复制到 kernfs_open_file 实例中缓存，然后调用函数 kernfs_ops.write()，将缓存中数据写入内核数据结构中（具体文件系统类型使用的数结构）。

7.13 sysfs 文件系统

sysfs 文件系统是 kernfs 文件系统的一个实例，sysfs 文件系统由 kobject 结构体实例组成。kobject 实例关联到 kernfs_node 实例或 attribute 实例，作为其私有数据，用于构成 sysfs 文件系统。

kobject 结构体主要用于跟踪内核中数据结构实例，如设备、驱动结构实例等。kobject 实例下包含属性，用于表示跟踪对象的某种参数或信息，如设备参数。属性由 attribute 结构体表示（表现为文件）。

每个 kobject 实例在 sysfs 文件系统中表示一个目录项，其下每个属性在此目录项下表现为一个文件。用户进程可通过对属性文件的读写查看/设置对象的参数。sysfs 文件系统通常由用户挂载到/sys/目录下。

sysfs 文件系统主要用于向用户空间导出设备和驱动等信息。若要支持 sysfs 需选择 SYSFS 配置选项，默认会选择 KERNFS 选项。

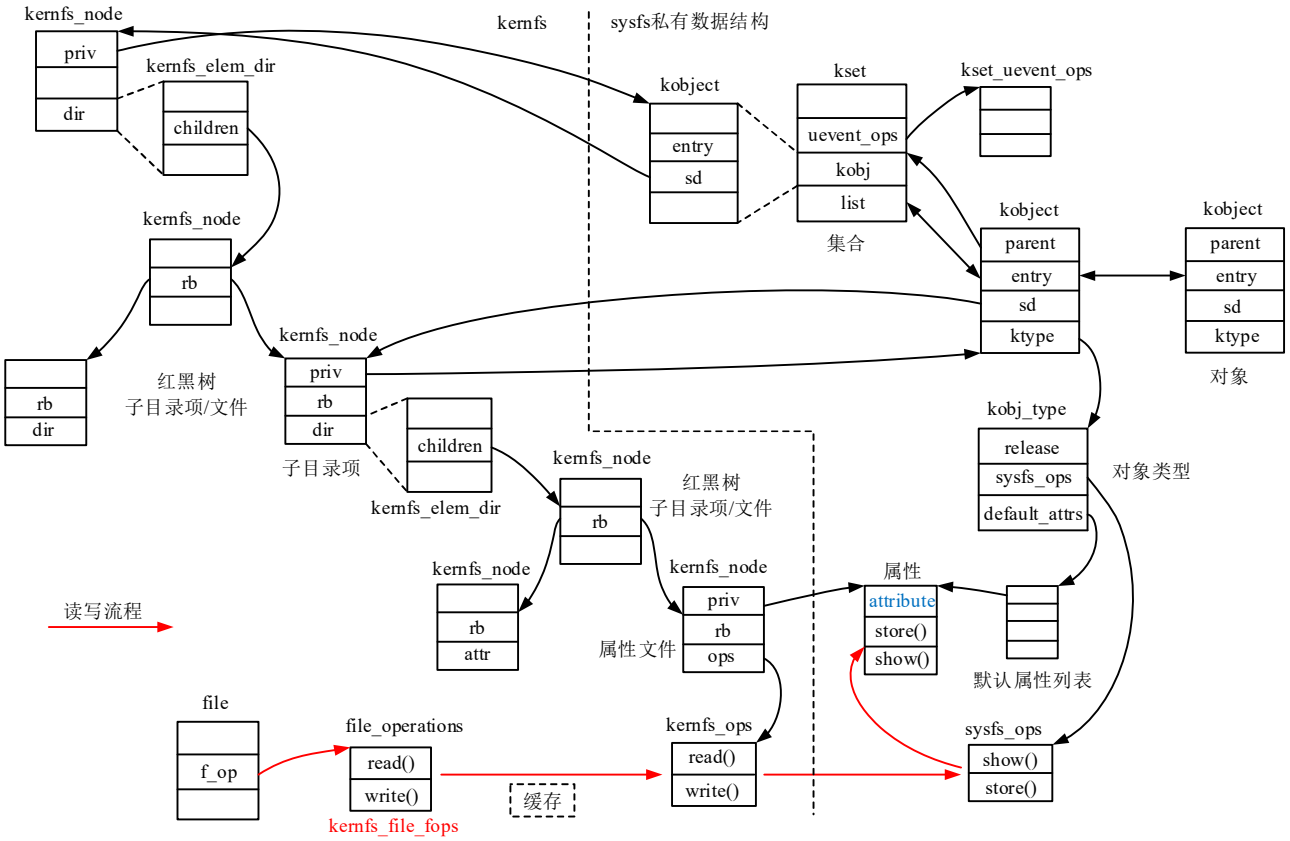
本节先介绍 kobject 结构体的定义和管理结构，然后介绍由此结构体实例构成的 sysfs 文件系统的实现。sysfs 文件系统实现代码位于/fs/sysfs/目录下。

7.13.1 概述

sysfs 文件系统结构如下图所示，sysfs 文件系统中 kobject 结构体实例对应目录项，属性 attribute 结构体实例对应文件。kobject 结构体通常嵌入到内核数据结构中，用于跟踪内核数据结构实例，特别是设备、驱动相关的数据结构实例。跟踪同类型数据结构的 kobject 实例通常归于一个集合，集合由 kset 结构体表示，它通过双链表来管理集合下的 kobject 实例。kset 结构体中也嵌入了 kobject 结构体成员，它也是被跟踪的对象。

kernfs_node 结构体 priv 成员指向 kobject 实例或属性 attribute 实例。kobject 结构体 sd 成员指向

kernfs_node 结构体，用于构建 kernfs 文件系统。kernfs_node 实例红黑树中节点表示其下子目录项或文件，文件由属性 attribute 实例表示。



kobject 实例在内核中组成父子关系的层次结构，kobject 实例的父节点可以是其所属集合 kset 实例中内嵌的 kobject 实例成员，也可以不是。同一个集合之下的 kobject 实例也可以组成父子层次结构。

kobject 实例关联一个表示对象类型的 kobj_type 结构体实例，其中定义了默认属性，以及属性读写操作结构 sysfs_ops 实例。

kobject 实例添加到内核层次管理结构时，将创建对应的 kernfs_node 实例，添加到 kernfs 文件系统中，其也属性也将为其创建 kernfs_node 实例，添加到 kobject 实例对应 kernfs_node 实例的红黑树中。也可以另外单独将属性添加到 kobject 实例对应 kernfs_node 实例的红黑树中（创建 kernfs_node 实例）。kernfs_node 实例的 priv 成员将指向 kobject 实例或属性 attribute 实例。

用户进程可以对 sysfs 文件系统中的属性文件进行读写。属性文件的打开、读写操作按照 kernfs 文件系统中的文件打开、读写操作进行。属性文件 file 实例关联的 file_operations 实例为 **kernfs_file_fops**，其中的读写函数将调用 kernfs_node 实例关联 kernfs_ops 实例中的读写函数。

在创建属性对应的 kernfs_node 实例时将由内核对其中的 ops 成员赋值 kernfs_ops 实例指针，不同类型属性的 kernfs_ops 实例可能不同。kernfs_ops 实例中的读写函数将调用 sysfs_ops 操作结构实例中的 show()、store()函数，最终调用特定于属性的 show()、store()函数完成属性的读写操作。

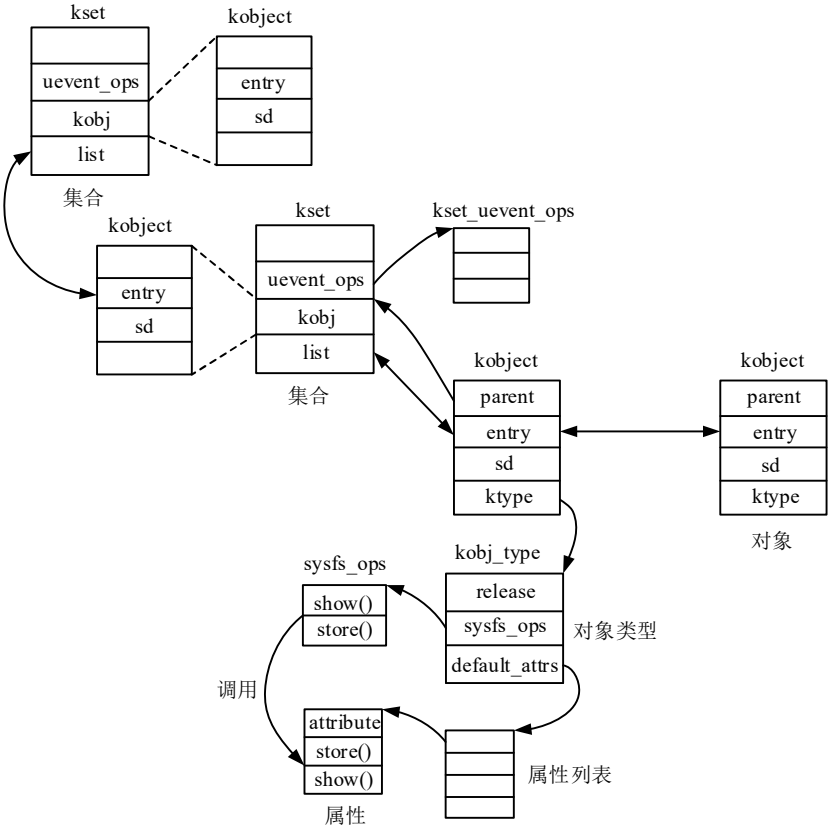
读写属性其实是对属性值（内容）的读写，属性值保存在内核（对象）中或动态生成，而不是保存在属性 attribute 结构体中，属性 attribute 结构体中主要包含属性的名称和访问权限等信息。在定义属性 attribute 实例时，需在其后附上读写本属性值的函数指针，供 sysfs_ops 实例中的 show()、store()函数调用。

7.13.2 数据结构

kobject 结构体通常嵌入到被跟踪对象数据结构中，例如，表示设备的 device 结构体中嵌入了 kobject 结构体成员 kobj。kobject 结构体通过对象类型 kobj_type 结构体实现对（默认）属性的管理和读写操作。

跟踪同类型对象的 kobject 实例通常由一个集合管理，集合由 kset 结构体表示。kset 结构体本身也嵌入了 kobject 结构体成员，成为被跟踪的对象。

kobject 实例链接到集合 kset 实例管理的双链表中。kset 结构体中内嵌的 kobject 实例也可以加入到更高层次的集合中。以此，kobject 实例在内核中构成层次的父子结构，如下图所示。



跟踪对象类型由 kobj_type 结构体表示，其中 sysfs_ops 成员指向 sysfs_ops 结构体实例，结构体中 show()和 store()函数实现对属性的读写操作。kobj_type 结构体 default_attrs 成员指向默认属性列表。属性中主要包含属性名称以及访问权限等成员。属性 attribute 实例后需附上本属性的读写函数，供 sysfs_ops 实例中函数调用。

kobject 结构体中 sd 成员指向 kernfs_node 结构体，用于将实例作为目录项导出到 kernfs 文件系统，kobject 实例下的每个属性将作为文件导出到 kernfs 文件系统。

下面先介绍 kobject、kset 等数据结构的定义，以及向内核添加 kobject、kset 等结构体实例的接口函数，本节最后将介绍 kobject 实例及其属性如何导出到 kernfs 文件系统，以及用户对属性的读写操作。

1 kset

kset 结构体表示一个集合，管理跟踪同类型对象的 kobject 实例，结构体定义在/include/linux/kobject.h 头文件：

```
struct kset {
    struct list_head    list;     /*链接集合中的 kobject 实例*/
    spinlock_t    list_lock;      /*自旋锁*/
    struct    kobject    kobj;    /*内嵌 kobject 实例，本身也是被跟踪的对象*/
    const struct kset_uevent_ops    *uevent_ops;     /*uevent 机制操作函数结构*/
};
```

如果集合中被跟踪对象状态有改变可通过 uevent 机制向用户空间传递事件信息，kset_uevent_ops 结构

体定义了事件操作函数指针，结构体定义如下（/include/linux/kobject.h）：

```
struct kset_uevent_ops {
    int (* const filter)    (struct kset *kset, struct kobject *kobj);
                                                    /*是否屏蔽事件，返回 0 表示不发送事件*/
    const char *(* const name)    (struct kset *kset, struct kobject *kobj);      /*获取子系统名称*/
    int (* const uevent)    (struct kset *kset, struct kobject *kobj,struct kobj_uevent_env *env);
                                        /*向 kobj_uevent_env 添加环境变量，传递到用户空间*/
};
```

kset_uevent_ops 结构体中主要成员简介如下：

●**const filter**：是否屏蔽对象向用户空间传递 uevent 事件，返回 0 表示屏蔽事件。

●**const name**：获取 kobject 实例所属子系统名称。

●**uevent**：向 kobj_uevent_env 添加环境变量，传递至用户空间。

2 kobject

kobject 结构体定义在/include/linux/kobject.h 头文件：

```
struct kobject {
    const char          *name;          /*名称*/
    struct list_head    entry;          /*双链表成员，将实例添加到所属 kset 实例的双链表中*/
    struct kobject      *parent;        /*指向父节点*/
    struct kset         *kset;          /*指向所属集合 kset 实例*/
    struct kobj_type    *ktype;         /*指向对象类型 kobj_type 实例*/
    struct kernfs_node  *sd;        /*指向 kernfs 文件系统节点，表示目录项*/
    struct kref     kref;           /*引用计数*/
#ifdef CONFIG_DEBUG_KOBJECT_RELEASE
    struct delayed_work     release;
#endif
     /*以下是位域成员*/
    unsigned int state_initialized:1;           /*实例是否已经初始化*/
    unsigned int state_in_sysfs:1;              /*是否导出到 kernfs 文件系统*/
    unsigned int state_add_uevent_sent:1;
    unsigned int state_remove_uevent_sent:1;
    unsigned int uevent_suppress:1;     /*置位表示 kobject 状态改变不向用户空间传递事件*/
};
```

kobject 结构体主要成员简介如下：

●**name**：实例名称，导出到 kernfs 文件系统为目录项名称。

●**entry**：双链表成员，将实例链接到 kset 实例双链表中。

●**parent**：指向父 kobject 实例。

●**kref**：对象引用计数，kref 结构体成员，非指针。kref 结构体定义在/include/linux/kref.h 头文件：

```
struct kref {
    atomic_t    refcount;    /*原子变量，记录引用计数*/
};
```

kref_get(struct kref *kref)函数用于增加引用计数，kref_put(struct kref *kref, void (***release**)(struct kref *kref))函数递减引用计数，而当计数值为 0 时将调用 release()函数。

●**ktype**：kobj_type 结构体指针，表示管理对象类型，结构体定义见下文。

●**sd**：kernfs_node 结构体指针，kobject 实例需要导出到 kernfs 文件系统时，需要创建 kernfs_node 实例。

●**kset**：kset 结构体指针，表示 kobject 实例所属集合。

3 kobj_type

kobj_type 结构体表示 kobject 跟踪对象的类型，结构体定义如下（/include/linux/kobject.h）：

```
struct kobj_type {
    void    (*release) (struct kobject *kobj);     /*释放 kobject 实例时调用此函数，(kobject_put())*/
    const struct sysfs_ops    *sysfs_ops;          /*属性操作结构*/
    struct attribute    **default_attrs;           /*默认属性指针列表*/
    const struct kobj_ns_type_operations    *(*child_ns_type) (struct kobject *kobj);
                                                   /*子节点命名空间*/
    const void    *(*namespace) (struct kobject *kobj);          /*返回命名空间标签*/
};
```

kobj_type 结构体主要成员简介如下：

●**release**：函数指针，释放 kobject 实例时调用此函数。

●**default_attrs**：默认属性指针列表。

属性由 attribute 结构体表示，定义如下（/include/linux/sysfs.h）：

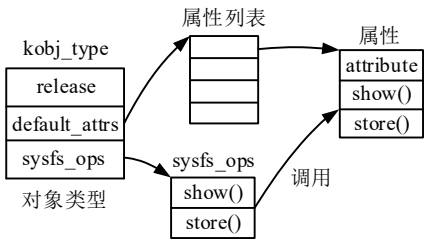
```
struct attribute {
    const char      *name;        /*属性名称*/
    umode_t         mode;         /*属性访问权限（模式）*/
#ifdef CONFIG_DEBUG_LOCK_ALLOC
    ...
#endif
};
```

●**sysfs_ops**：指向 sysfs_ops 结构体，表示读写属性操作结构。

sysfs_ops 结构体定义在/include/linux/sysfs.h 头文件：

```
struct sysfs_ops {
    ssize_t    (*show) (struct kobject *, struct attribute *, char *);                  /*读属性文件函数*/
    ssize_t    (*store) (struct kobject *, struct attribute *, const char *, size_t);   /*写属性文件函数*/
};
```

通常在定义属性时，在 attribute 实例后会附上特定于属性的读写函数。sysfs_ops 实例中的读写函数将调用特定属性定义的读写函数，完成对属性的读写操作，如下图所示。对属性的读写就是获取/设置属性值。



7.13.3 创建/添加 kobject 实例

kobject 结构体通常嵌入到被管理对象数据结构中，在创建管理对象时一同创建 kobject 实例。接口函数 **kobject_add()**用于将现有实例添加到 kobject 实例层次结构中，并将其及其下属性导出到 kernfs 文件系统。调用 kobject_add()函数添加 kobject 实例前，需要对其初始化。

另外，kobject 实例也可以动态单独创建，被跟踪对象通过指针成员关联到 kobject 实例。接口函数 kobject_create_and_add()用于动态创建并添加 kobject 实例。

1 初始化实例

kobject 实例在添加到层次管理结构前都需要对其进行初始化，初始化函数为 kobject_init()，函数定义如下（/lib/kobject.c）：

```
void kobject_init(struct kobject *kobj, struct kobj_type *ktype)
/*kobj：kobject 实例指针，ktype：kobj_type 实例指针*/
{
    char *err_str;

    if (!kobj) {                /*kobj 指针不能为空*/
        ...
    }
    if (!ktype) {           /*ktype 指针不能为空*/
        ...
    }
    if (kobj->state_initialized) {          /*实例已经被初始化，函数返回*/
        ...
    }

    kobject_init_internal(kobj);     /*内部初始化函数，/lib/kobject.c*/
    kobj->ktype = ktype;          /*设置对象类型*/
    return;
    ...
}
```

初始化函数在判断参数的有效性后将工作转交给 kobject_init_internal(kobj)函数，函数实现比较简单，代码如下：

```
static void kobject_init_internal(struct kobject *kobj)
{
    if (!kobj)
        return;
    kref_init(&kobj->kref);                    /*初始化引用计数为 1*/
    INIT_LIST_HEAD(&kobj->entry);
    kobj->state_in_sysfs = 0;
    kobj->state_add_uevent_sent = 0;
    kobj->state_remove_uevent_sent = 0;
    kobj->state_initialized = 1;        /*标记实例已经初始化*/
```

```
}
```

初始化 kobject 实例时，除了调用 kobject_init()函数初始化实例外，可能还需要手工设置 kobj->kset 成员，以设置 kobject 实例所属的集合。如果设置了 kobj->kset 成员，在后面的添加操作中会将 kobject 实例添加到集合 kset 实例管理的双链表中。

2 添加实例

kobject_add()函数用于将指定 kobject 实例添加到层次管理结构中并导出到 kernfs 文件系统，函数定义在/lib/kobject.c 文件内，函数执行成功返回 0，否则返回错误码，函数代码如下。

```
int kobject_add(struct kobject *kobj, struct kobject *parent,const char *fmt, ...)
/*kobj：kobject 实例指针，parent ：父 kobject 实例指针，
*fmt：名称字符串指针，如果实例已设置名称且不需要重设，则 fmt 必须为 NULL。
*/
{
    va_list args;
    int retval;

    if (!kobj)
        return -EINVAL;

    if (!kobj->state_initialized) {        /*添加实例必须已经被初始化*/
        ...
        return -EINVAL;
    }
    va_start(args, fmt);
    retval = kobject_add_varg(kobj, parent, fmt, args);      /*添加实例，/lib/kobject.c*/
    va_end(args);
    return retval;
}
```

kobject_add()函数添加的实例必须已经被初始化，函数内调用 kobject_add_varg(kobj, parent, fmt, args)函数完成向内核添加实例的操作，函数定义如下（/lib/kobject.c）：

```
static __printf(3, 0) int kobject_add_varg(struct kobject *kobj,struct kobject *parent,        \
                                                                    const char *fmt, va_list vargs)
{
    int retval;

    retval = kobject_set_name_vargs(kobj, fmt, vargs);      /*/lib/kobject.c*/
            /*如果实例已经设置了 name 成员且 fmt 为 NULL，则不重设实例名称，否则设为 fmt*/
    ...
    kobj->parent = parent;              /*指向父节点*/
    return    kobject_add_internal(kobj);        /*内部添加函数，/lib/kobject.c*/
}
```

kobject_add_varg()函数内设置实例名称后（如果需要），调用 kobject_add_internal(kobj)函数完成添加实例操作，函数代码如下：

```
static int kobject_add_internal(struct kobject *kobj)
{
    int error = 0;
    struct kobject *parent;

    if (!kobj)
        return -ENOENT;

    if (!kobj->name || !kobj->name[0]) {        /*名称字符串为空，返回错误码*/
        ..
        return -EINVAL;
    }

    parent = kobject_get(kobj->parent);       /*增加父节点引用计数*/

    if (kobj->kset) {          /*如果设置了 kset 指针成员，指明了所属集合*/
        if (!parent)           /*如果 kobject 实例没有指定父节点，则默认 kset->kobj 为其父节点*/
            parent = kobject_get(&kobj->kset->kobj);
        kobj_kset_join(kobj);       /*kobject 实例添加到 kset 双链表中，/lib/kobject.c*/
        kobj->parent = parent;     /*指向父节点，可能是 kset.kobj 成员*/
    }
    ...    /*输出信息*/

    error = create_dir(kobj);       /*将 kobject 实例及其属性导出到 kernfs 文件系统，后面再做介绍*/
    if (error) {
        ...            /*导出至 kernfs 文件系统失败*/
    } else
        kobj->state_in_sysfs = 1;          /*导出成功，设置标记位*/

    return error;
}
```

综上所述，kobject_add()函数将 kobject 实例添加到内核管理结构后，将建立 kobject 实例与其父节点之间的关联，如果 parent 参数为 NULL，则默认所属集合 kset 实例的 kobj 成员为其父节点，并将实例添加到 kset 实例的双链表中（如果没有设置 kobj->kset 成员将不添加）。

另外，kobject_add()函数将通过 **create_dir(kobj)**函数将 kobject 实例及其下属性导出到 kernfs 文件系统（详见下文），并设置实例的 state_in_sysfs 标记位。

3 创建并添加实例

kobject 实例除了静态定义外，还可以动态创建并添加。接口函数 kobject_create_and_add()用于动态创建并添加 kobject 实例。

kobject_create_and_add()函数定义在/lib/kobject.c 文件内，代码简列如下，函数成功实例指针，否则返

回 NULL：

```
struct kobject *kobject_create_and_add(const char *name, struct kobject *parent)
/*name：名称字符串指针，parent：父 kobject 实例指针*/
{
    struct kobject *kobj;
    int retval;

    kobj = kobject_create();      /*创建并初始化实例，/lib/kobject.c*/
    ...

    retval = kobject_add(kobj, parent, "%s", name);      /*添加实例*/
    ...    /*错误处理*/
    return kobj;      /*返回实例指针*/
}
```

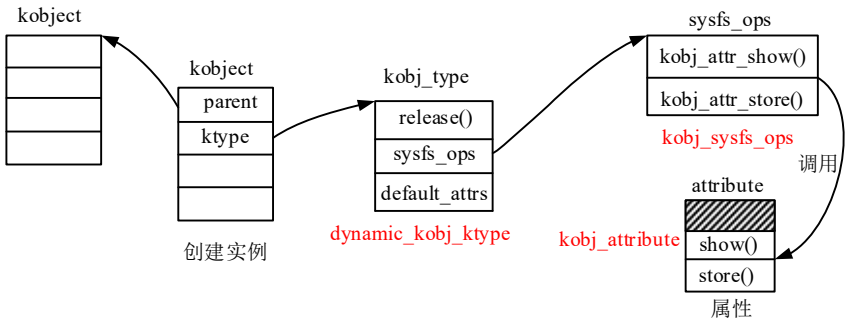
kobject_create_and_add()函数调用 kobject_create()函数创建并初始化 kobject 实例，然后调用函数 kobject_add()添加实例。添加实例的函数前面介绍过了，下面主要看一下 kobject_create()函数的实现。

kobject_create()函数代码如下：

```
struct kobject *kobject_create(void)
{
    struct kobject *kobj;

    kobj = kzalloc(sizeof(*kobj), GFP_KERNEL);    /*从通用缓存中分配 kobject 实例*/
    if (!kobj)
        return NULL;

    kobject_init(kobj, &dynamic_kobj_ktype);      /*初始化 kobject 实例*/
    return kobj;
}
```

kobject_create()函数从通用缓存中分配 kobject 实例，调用 kobject_init()函数初始化实例。

这里需要注意的是 kobject 实例关联的 kobj_type 实例为 dynamic_kobj_ktype，没有指定所属的 kset 实例，创建的 kobject 实例结构如下图所示（没有指定默认属性）：



动态创建 kobject 实例关联的 kobj_type 实例 **dynamic_kobj_ktype** 定义如下（/lib/kobject.c）：

```
static struct kobj_type dynamic_kobj_ktype = {
    .release    = dynamic_kobj_release,    /*直接释放 kobject 实例，因为它是单独分配的，/lib/kobject.c*/
    .sysfs_ops = &kobj_sysfs_ops,          /*/lib/kobject.c*/
```

```
};
```

读写属性操作 kobj_sysfs_ops 实例定义如下：

```
const struct sysfs_ops kobj_sysfs_ops = {
    .show      = kobj_attr_show,          /*读属性函数*/
    .store     = kobj_attr_store,         /*写属性函数*/
};
```

kobj_sysfs_ops 实例中读写属性函数只是一个公共的接口，函数内需要调用具体属性的读写函数，实现从一般到具体的转换。

dynamic_kobj_ktype 类型 kobject 实例的属性由 kobj_attribute 结构体，定义在/include/linux/kobject.h 头文件：

```
struct kobj_attribute {
    struct attribute attr;                  /*属性*/
    ssize_t (*show) (struct kobject *kobj, struct kobj_attribute *attr,char *buf);
    ssize_t (*store) (struct kobject *kobj, struct kobj_attribute *attr,const char *buf, size_t count);
};
```

这里的 show()和 store()函数是特定于属性的读写函数。kobj_sysfs_ops 实例中读写函数将调用此处的读写函数。

kobj_sysfs_ops 实例中读属性函数定义如下（/lib/kobject.c）：

```
static ssize_t kobj_attr_show(struct kobject *kobj, struct attribute *attr, char *buf)
{
    struct kobj_attribute *kattr;
    ssize_t ret = -EIO;

    kattr = container_of(attr, struct kobj_attribute, attr);      /*属性 attribute 实例转 kobj_attribute 实例*/
    if (kattr->show)
        ret = kattr->show(kobj, kattr, buf);     /*kobj_attribute 实例中的 show()函数*/
    return ret;
}
```

kobj_sysfs_ops 实例中写属性函数与读函数类似，函数定义如下：

```
static ssize_t kobj_attr_store(struct kobject *kobj, struct attribute *attr,    const char *buf, size_t count)
{
    struct kobj_attribute *kattr;
    ssize_t ret = -EIO;

    kattr = container_of(attr, struct kobj_attribute, attr);      /*属性 attribute 实例转 kobj_attribute 实例*/
    if (kattr->store)
        ret = kattr->store(kobj, kattr, buf, count);     /*kobj_attribute 实例中的 store()函数*/
    return ret;
}
```

dynamic_kobj_ktype 实例中没有指定默认属性，可以调用后面介绍的 sysfs_create_file(kobj, attr)函数向 kobject 实例添加属性，即在其目录项下添加属性文件，其它类型的 kobject 实例也可以调用这个函数添加属性。

4 释放实例

内核在/lib/kobject.c 文件内定义了操作 kobject 实例的其它接口函数，例如：

●struct kobject ***kobject_get**(struct kobject *kobj)：kobject 实例引用计数加 1。

●void **kobject_put**(struct kobject *kobj)：kobject 实例引用计数减 1。当引用计数为 0 时，将调用函数 kobject_release()释放实例，此函数会将 kobject 实例从层次管理结构及 kernfs 文件系统中移除，最后调用 kobj_type 实例中定义的 release()函数释放 kobject 实例，函数源代码请读者自行阅读。

7.13.4 创建/添加 kset 实例

kset 结构体表示一个集合，用于管理跟踪同类型对象的 kobject 实例。kset 实例在内核中一般是单独动态创建的，不嵌入到其它数据结构中。kset 内嵌的 kobject 实例可作为其下 kobject 实例的父节点。

1 创建并添加实例

kobject_create_and_add()函数用于动态创建并添加 kset 实例，主要工作是将其中 kobject 结构体成员添加到内核管理结构，并导出到 kernfs 文件系统。

kobject_create_and_add()函数用于在 kernfs 文件系统中创建一个目录项，kset 表示集合下的 kobject 实例将位于此目录项下。kobject_create_and_add()函数代码如下（/lib/kobject.c）：

```
struct kset *kset_create_and_add(const char *name, const struct kset_uevent_ops *uevent_ops,   \
                                                        struct kobject *parent_kobj)
/*name：名称字符串，uevent_ops：kset_uevent_ops 实例指针，parent_kobj：父 kobject 实例指针*/
{
    struct kset *kset;
    int error;

    kset = kset_create(name, uevent_ops, parent_kobj);     /*创建 kset 实例，/lib/kobject.c*/
    ...
    error = kset_register(kset);          /*注册 kset 实例，/lib/kobject.c*/
    ...
    return kset;        /*返回 kset 实例指针*/
}
```

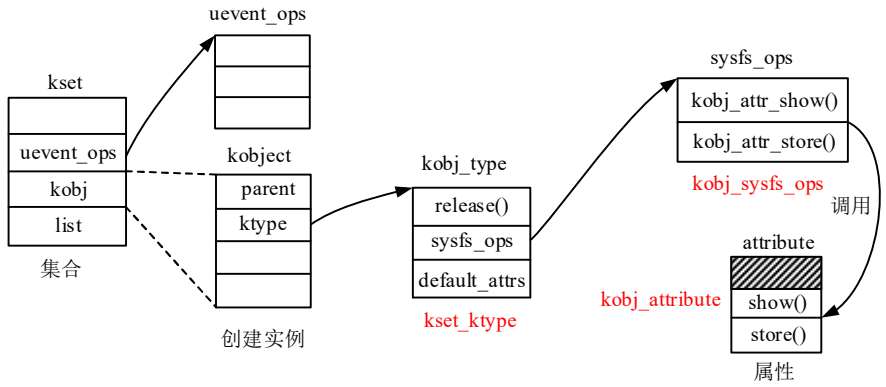
kset_create_and_add()函数主要分两步，一是创建 kset 实例，二是注册 kset 实例。

创建 kset 实例的 kset_create()函数定义如下：

```
static struct kset *kset_create(const char *name,const struct kset_uevent_ops *uevent_ops, \
                                                        struct kobject *parent_kobj)
{
    struct kset *kset;
    int retval;
```

```
    kset = kzalloc(sizeof(*kset), GFP_KERNEL);      /*从通用缓存中分配 kset 实例*/
    ...
    retval = kobject_set_name(&kset->kobj, "%s", name);      /*设置名称*/
    ...
    kset->uevent_ops = uevent_ops;        /*kset_uevent_ops 实例*/
    kset->kobj.parent = parent_kobj;       /*父节点*/

    kset->kobj.ktype = &kset_ktype;      /*kobj_type 实例，/lib/kobject.c*/
    kset->kobj.kset = NULL;

    return kset;        /*返回 kset 实例指针*/
}
```

kset_create()函数创建的 kset 实例如下图所示（没有指定默认属性）：



kset 实例内嵌 kobject 结构体成员函关联的 kobj_type 实例为 **kset_ktype**，定义如下：

```
static struct kobj_type kset_ktype = {
    .sysfs_ops = &kobj_sysfs_ops,     /*同动态创建 kobject 实例*/
    .release    = kset_release,            /*直接释放 kset 实例*/
};
```

kset 实例内嵌 kobject 实例的属性及读写操作结构，同前面介绍的 dynamic_kobj_ktype 实例。

kobject_create_and_add()函数的第二步是调用 kset_register()函数注册 kset 实例，函数定义如下：

```
int kset_register(struct kset *k)
{
    int err;
    ...
    kset_init(k);                /*初始化 kset 实例，调用 kobject_init_internal(&k->kobj)函数*/
    err = kobject_add_internal(&k->kobj);     /*添加内嵌 kobject 结构体成员至内核*/
    ...
    kobject_uevent(&k->kobj, KOBJ_ADD);
                                /*触发 uevent 事件，通过 netlink 套接字向用户空间发送消息*/
    return 0;
}
```

kset_register()函数内调用 kset_init(k)函数对 kset 实例进行初始化，其中包括调用 kobject_init_internal()函数初始化内嵌 kobject 结构体成员，调用 kobject_add_internal()函数添加内嵌的 kobject 结构体成员，最后

调用 kobject_uevent()函数触发 uevent 事件（通过 netlink 套接字向用户空间发送消息，详见第 12 章）。

2 uevent 机制

uevent 机制用于内核向用户空间发送事件消息，内核在对 kobject 实例进行添加、移除等操作时可以向用户空间发送事件消息。

uevent 机制的接口函数为 kobject_uevent(struct kobject *kobj, enum kobject_action action)，kobj 表示 kobject 实例，action 表示传递的事件类型，函数内会构建环境变量（传递的信息）通过 netlink 套接字传递给用户空间（需选择 NET 配置选项）。老的方式是通过创建进程以处理事件，此方法已被弃用（需选择配置选项 UEVENT_HELPER）。

uevent 机制一个重要的应用是在向内核添加设备、驱动时，或设备状态改变时，向用户空间发送事件消息，一般由 udev（mdev）用户进程处理接收并处理事件消息（如创建设备文件等）。

uevent 机制发送的事件类型定义在/include/linux/kobject.h 头文件：

```
enum kobject_action {
    KOBJ_ADD,              /*添加对象*/
    KOBJ_REMOVE,           /*移除对象*/
    KOBJ_CHANGE,           /*对象改变*/
    KOBJ_MOVE,             /*移动对象*/
    KOBJ_ONLINE,           /*上线*/
    KOBJ_OFFLINE,          /*离线*/
    KOBJ_MAX               /*事件类型数量*/
};
```

kobj_uevent_env 结构体用于缓存事件信息，定义如下（/include/linux/kobject.h）：

```
struct kobj_uevent_env {
     char *argv[3];
     char *envp[UEVENT_NUM_ENVP];        /*环境变量指针，数组项为 32*/
     int envp_idx;
     char buf[UEVENT_BUFFER_SIZE];       /*缓存环境变量，2048 字节*/
     int buflen;              /*实际缓存环境变量长度*/
};
```

事件信息是由形如“环境变量名称=值”的字符串组成，字符串保存在 buf[]数组，envp[]指针数组项指向环境变量的开始位置，环境变量数量最大为 32 个。

kobject_uevent()函数定义在/lib/kobject_uevent.c 文件内，主要工作就是创建 kobj_uevent_env 实例，填充环境变量，最后通过 netlink 套接字发送到用户空间，函数代码如下：

```
int kobject_uevent(struct kobject *kobj, enum kobject_action action)
{
     return kobject_uevent_env(kobj, action, NULL);     /*/lib/kobject_uevent.c*/
}
```

kobject_uevent()函数内直接调用 kobject_uevent_env()函数，定义如下：

```
int kobject_uevent_env(struct kobject *kobj, enum kobject_action action,char *envp_ext[])
{
```

```
    struct kobj_uevent_env *env;
    const char *action_string = kobject_actions[action];      /*字符串指针数组，/lib/kobject_uevent.c*/
    const char *devpath = NULL;
    const char *subsystem;
    struct kobject *top_kobj;
    struct kset *kset;
    const struct kset_uevent_ops *uevent_ops;
    int i = 0;
    int retval = 0;
#ifdef CONFIG_NET
    struct uevent_sock *ue_sk;
#endif

    pr_debug("kobject: '%s' (%p): %s\n", kobject_name(kobj), kobj, __func__);

    /*查找 kobject 所属的 kset 实例*/
    top_kobj = kobj;
    while (!top_kobj->kset && top_kobj->parent)
                        /*一直往上查找 kobject 实例，直至找到 kset 指针不为空的实例*/
        top_kobj = top_kobj->parent;

    if (!top_kobj->kset) {          /*没有找到 kset 实例，返回错误码*/
        ...
        return -EINVAL;
    }

    kset = top_kobj->kset;      /*kset 实例*/
    uevent_ops = kset->uevent_ops;          /*kset_uevent_ops 实例*/

    if (kobj->uevent_suppress) {        /*如果 kobject 实例设置了不传递事件消息，直接返回 0*/
        ...
        return 0;
    }

    if (uevent_ops && uevent_ops->filter)       /*调用过滤函数检查是否发送事件消息*/
        if (!uevent_ops->filter(kset, kobj)) {    /*uevent_ops->filter()函数返回 0，则不发送事件*/
            ...
            return 0;
        }

    if (uevent_ops && uevent_ops->name)       /*获取子系统名称*/
         subsystem = uevent_ops->name(kset, kobj);
    else
```

```
            subsystem = kobject_name(&kset->kobj);       /*kobject 名称*/
        ...
        env = kzalloc(sizeof(struct kobj_uevent_env), GFP_KERNEL);   /*创建 kobj_uevent_env 实例*/
        ...

        devpath = kobject_get_path(kobj, GFP_KERNEL);
                                    /*获取 kobject 实例在层次结构中的路径名称，/lib/kobject.c*/
        ...

        /*添加默认的环境变量至 kobj_uevent_env 实例 buf[]缓存区*/
        retval = add_uevent_var(env, "ACTION=%s", action_string);     /*添加环境变量接口函数*/
        ...
        retval = add_uevent_var(env, "DEVPATH=%s", devpath);
        ...
        retval = add_uevent_var(env, "SUBSYSTEM=%s", subsystem);
        ...
        /*调用 uevent_ops->uevent()函数添加环境变量*/
        if (uevent_ops && uevent_ops->uevent) {
            retval = uevent_ops->uevent(kset, kobj, env);        /*添加环境变量*/
            ..
        }

        if (action == KOBJ_ADD)            /*设置 kobject 状态*/
            kobj->state_add_uevent_sent = 1;
        else if (action == KOBJ_REMOVE)
            kobj->state_remove_uevent_sent = 1;

        mutex_lock(&uevent_sock_mutex);
        retval = add_uevent_var(env, "SEQNUM=%llu", (unsigned long long)++uevent_seqnum);
        ...

#if defined(CONFIG_NET)
                    /*通过 netlink 套接字将 kobj_uevent_env 实例中数据发送到用户空间*/
        ...         /*详见第 12 章*/
#endif
        mutex_unlock(&uevent_sock_mutex);

#ifdef CONFIG_UEVENT_HELPER
        ...         /*老的向用户空间传递信息的机制*/
#endif

exit:
        kfree(devpath);
```

```
    kfree(env);          /*释放 kobj_uevent_env 实例*/
    return retval;
}
```

kobject_uevent_env()函数首先确定 kobject 实例所属的 kset 实例，然后判断是否需要向用户空间发送事件，如果不需要则直接返回 0。如果需要发送事件，则创建 kobj_uevent_env 实例，添加默认的环境变量，调用 uevent_ops->uevent(kset, kobj, env)函数添加 kset 实例定义的环境变量，通过 netlink 套接字将环境变量数据传递给用户空间，最后释放 kobj_uevent_env 实例。netlink 套接字到第 12 章再做介绍。

7.13.5 sysfs 实现

sysfs 文件系统是 kernfs 文件系统的一个实例，可以认为 sysfs 就是带私有数据的 kernfs。kobject 结构体中 sd 成员指向 kernfs_node 结构体，用于构建 kernfs 文件系统。

在前面介绍的添加 kobject 实例的 kobject_add()函数中，将调用 create_dir(kobj)函数为 kobject 实例创建 kernfs_node 实例，并添加到文件系统结构中。kobject 实例对应 sysfs 文件系统中的一个目录项，其下属性在此目录项下由一个文件表示，在 create_dir(kobj)函数中还将为属性在 sysfs 文件系统中创建 kernfs_node 实例。用户进程对属性文件的读写操作流程同 kernfs 文件系统中文件的读写操作流程。

1 挂载文件系统

sysfs 文件系统类型定义在/fs/sysfs/mount.c 文件内：

```
static struct file_system_type sysfs_fs_type = {
    .name     = "sysfs",              /*文件系统类型名称*/
    .mount    = sysfs_mount,          /*挂载函数，/fs/sysfs/mount.c*/
    .kill_sb  = sysfs_kill_sb,
    .fs_flags = FS_USERNS_VISIBLE | FS_USERNS_MOUNT,
};
```

在/fs/sysfs/mount.c 文件内定义了 sysfs 文件系统根节点 kernfs_root 和 kernfs_node 实例的指针：

```
static  struct kernfs_root  *sysfs_root;
struct  kernfs_node  *sysfs_root_kn;
```

内核在/fs/sysfs/mount.c 文件内定义了 sysfs 文件系统的初始化函数 sysfs_init()，此函数在 mnt_init()函数内被调用。sysfs_init()主要工作是创建文件系统根节点 kernfs_root 实例，以及注册文件系统类型实例，函数代码如下（/fs/sysfs/mount.c）：

```
int __init sysfs_init(void)
{
    int err;
    sysfs_root=kernfs_create_root(NULL,KERNFS_ROOT_EXTRA_OPEN_PERM_CHECK,NULL);
                                    /*创建 kernfs 文件系统根节点 kernfs_root 和 kernfs_node 实例*/
    ...

    sysfs_root_kn = sysfs_root->kn;        /*根节点对应的 kernfs_node 实例赋予全局变量*/

    err = register_filesystem(&sysfs_fs_type);  /*注册文件系统类型*/
```

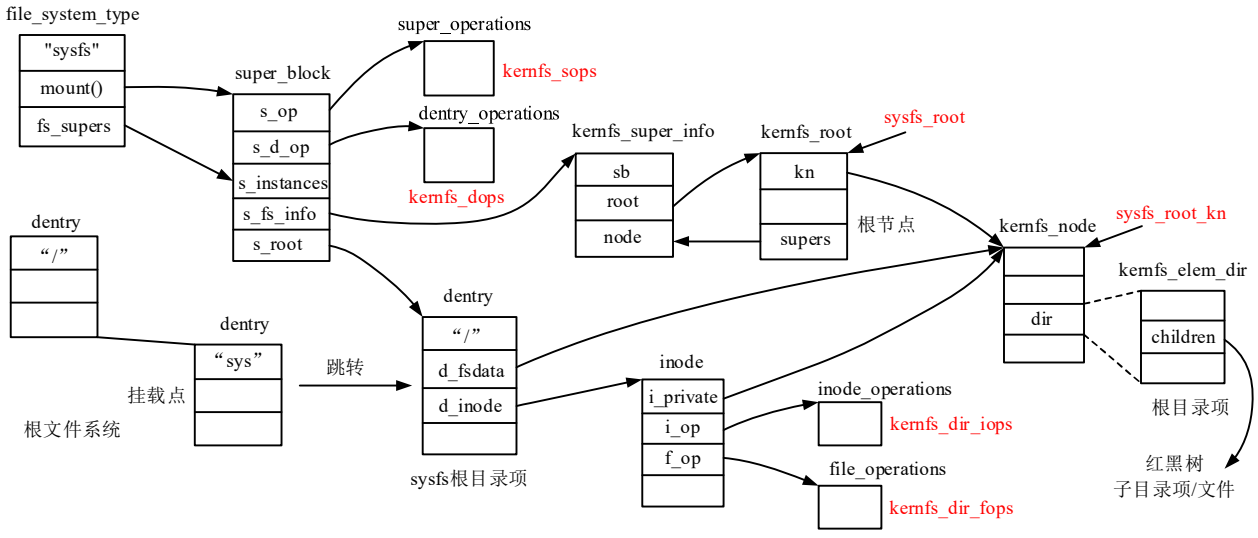
```
    ...
    return 0;
}
```

sysfs 文件系统通常由用户挂载到/sys/目录下，文件系统类型定义的挂载函数为 sysfs_mount()，代码如下（/fs/sysfs/mount.c）：

```
static struct dentry *sysfs_mount(struct file_system_type *fs_type,   \
                                  int flags, const char *dev_name, void *data)
{
    struct dentry *root;
    void *ns;
    bool new_sb;

    if (!(flags & MS_KERNMOUNT)) {
        if (!kobj_ns_current_may_mount(KOBJ_NS_TYPE_NET))
            return ERR_PTR(-EPERM);
    }

    ns = kobj_ns_grab_current(KOBJ_NS_TYPE_NET);    /*获取当前进程命名空间*/
    root = kernfs_mount_ns(fs_type, flags, sysfs_root,SYSFS_MAGIC, &new_sb, ns);
                                    /*kernfs 文件系统定义的挂载函数，见上节*/
    if (IS_ERR(root) || !new_sb)
        kobj_ns_drop(KOBJ_NS_TYPE_NET, ns);
    return root;
}
```

sysfs_mount()函数内调用 kernfs 文件系统中定义的挂载函数 kernfs_mount_ns()，根节点为 sysfs_root 指向的 kernfs_root 实例。

sysfs_mount()函数创建的数据结构实例如下图所示：

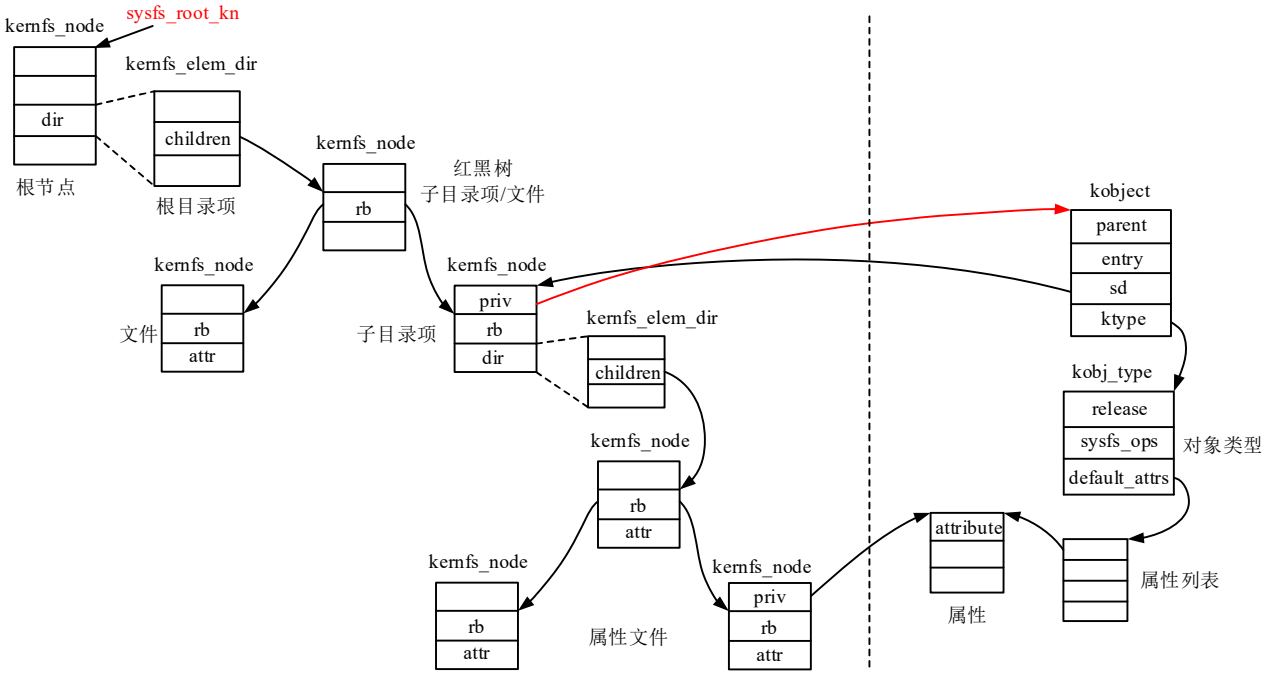


2 添加目录项

在 kobject_add()函数中添加 kobject 实例时，将调用 create_dir(kobj)函数，为 kobject 实例创建 kernfs_node 实例（目录项），并添加到 sysfs_root 指向根节点的 kernfs 文件系统中。如果 kobject 实例中还包含属性，create_dir()函数还将为其下每个属性创建 kernfs_node 实例，作为 kobject 实例关联 kernfs_node 实例的子节点，添加到其下红黑树中。

如下图所示，假设添加的 kobject 实例没有指定父节点，其下具有默认属性。create_dir()函数将为添加的 kobject 实例创建 kernfs_node 实例，并添加到 sysfs 文件系统根节点管理的红黑树中，作为根目录下的子目录项。create_dir()函数还将为 kobject 实例的每个默认属性创建 kernfs_node 实例（下图中只画出一个属性），并添加到 kobject 实例对应的目录项下，作为其下文件。

kobject 实例中的 sd 成员指向 kernfs_node 实例，kernfs_node 实例中的 priv 成员指向 kobject 实例或属性 attribute 实例。



create_dir()函数定义在/lib/kobject.c 文件内，代码如下：

```
static int create_dir(struct kobject *kobj)
{
    const struct kobj_ns_type_operations *ops;
    int error;

    error = sysfs_create_dir_ns(kobj, kobject_namespace(kobj));
                                    /*创建并添加 kobject 实例对应 kernfs_node 实例，/fs/sysfs/dir.c*/
    ...
    error = populate_dir(kobj);   /*为每个默认属性创建并添加 kernfs_node 实例，/lib/kobject.c*/
    ...
    sysfs_get(kobj->sd);     /*增加 kernfs_node 实例引用计数，/include/linux/sysfs.h*/
    ops = kobj_child_ns_ops(kobj);
    if (ops) {
        ...
        sysfs_enable_ns(kobj->sd);
```

```
    }
    return 0;
}
```

create_dir()函数内调用 sysfs_create_dir_ns()函数为 kobject 实例创建并添加表示目录项的 kernfs_node 实例，调用 populate_dir()函数为每个默认属性创建并添加表示文件的 kernfs_node 实例。下面将分别介绍这两个函数的实现。

■创建并添加目录项

sysfs_create_dir_ns()函数定义在/fs/sysfs/dir.c 文件内，代码如下：

```
int sysfs_create_dir_ns(struct kobject *kobj, const void *ns)
/*ns：由 kobj->ktype->namespace()函数返回的命名空间标签*/
{
    struct kernfs_node *parent, *kn;

    BUG_ON(!kobj);

    if (kobj->parent)
        parent = kobj->parent->sd;          /*父节点*/
    else
        parent = sysfs_root_kn;     /*如果没有指定父节点，则 sysfs 文件系统根节点作为其父节点*/
    ...
    kn = kernfs_create_dir_ns(parent, kobject_name(kobj),S_IRWXU | S_IRUGO | S_IXUGO, kobj, ns);
                                /*创建并添加表示目录项的 kernfs_node 实例，kobj 为其私有数据*/
    ...
    kobj->sd = kn;          /*指向创建的 kernfs_node 实例*/
    return 0;
}
```

sysfs_create_dir_ns()函数内首先需要确定添加 kobject 实例在 kernfs 文件系统中的父节点，若指定了父节点，则父 kobject 实例关联的 kernfs_node 实例为父节点，否则默认 sysfs 文件系统根节点为其父节点；然后调用 kernfs_create_dir_ns()函数创建并添加表示目录项的 kernfs_node 实例，注意实例 priv 成员指向 kobject 实例。

■添加属性文件

populate_dir(kobj)函数为每个默认属性创建表示文件的 kernfs_node 实例，函数定义在/lib/kobject.c 文件内：

```
static int populate_dir(struct kobject *kobj)
{
    struct kobj_type *t = get_ktype(kobj);      /*获取 kobject 实例关联 kobj_type 实例*/
    struct attribute *attr;
    int error = 0;
    int i;
```

```
    if (t && t->default_attrs) {        /*具有默认属性*/
        for (i = 0; (attr = t->default_attrs[i]) != NULL; i++) {
            error = sysfs_create_file(kobj, attr);
                /*为每个默认属性创建并添加 kernfs_node 实例（文件），/include/linux/sysfs.h*/
            ...
        }
    }
    /*如果没有指定默认属性则直接返回*/
    return error;
}
```

populate_dir()函数主要工作是对 kobj_type 实例每个默认属性调用 sysfs_create_file()函数，为其创建并添加表示文件的 kernfs_node 实例。

sysfs_create_file()函数定义在/include/linux/sysfs.h 头文件：

```
static inline int __must_check    sysfs_create_file(struct kobject *kobj,const struct attribute *attr)
{
    return    sysfs_create_file_ns(kobj, attr, NULL);     /*/fs/sysfs/file.c*/
}
```

sysfs_create_file_ns()函数定义在/fs/sysfs/file.c 文件内，代码如下：

```
int sysfs_create_file_ns(struct kobject *kobj, const struct attribute *attr,const void *ns)
{
    BUG_ON(!kobj || !kobj->sd || !attr);

    return    sysfs_add_file_mode_ns(kobj->sd, attr, false, attr->mode, ns);
                                    /*attr->mode 传递属性文件模式（访问权限），/fs/sysfs/file.c*/
}
```

sysfs_add_file_mode_ns()定义在/fs/sysfs/file.c 文件内，代码如下：

```
int sysfs_add_file_mode_ns(struct kernfs_node *parent, const struct attribute *attr, bool is_bin,   \
                                                                    umode_t mode, const void *ns)
/*is_bin：是否是二进制属性*/
{
    struct lock_class_key *key = NULL;
    const struct kernfs_ops *ops;          /*kernfs 定义的操作结构*/
    struct kernfs_node *kn;
    loff_t size;

    if (!is_bin)
    {                               /*非二进制属性*/
        struct kobject *kobj = parent->priv;        /*kernfs_node->priv*/
        const struct sysfs_ops *sysfs_ops = kobj->ktype->sysfs_ops; /*对象类型关联的属性操作结构*/
        ...        /*输出信息*/
        /*确定 kernfs_node 实例关联的 kernfs_ops 实例*/
        if (sysfs_ops->show && sysfs_ops->store)        /*sysfs_ops 实例定义了读写函数*/
```

```
        {
            if (mode & SYSFS_PREALLOC)        /*SYSFS_PREALLOC 标记需要分配缓存*/
                ops = &sysfs_prealloc_kfops_rw;        /*预先分配缓存，直接读写*/
            else
                ops = &sysfs_file_kfops_rw;        /*不需要预先分配缓存，顺序读，直接写*/
        }
        else if (sysfs_ops->show)        /*sysfs_ops 实例只定义了读函数，只读属性*/
        {
            if (mode & SYSFS_PREALLOC)
                                /*需预先分配缓存，mode 设置 SYSFS_PREALLOC 标记位*/
                ops = &sysfs_prealloc_kfops_ro;      /*直接读*/
            else
                ops = &sysfs_file_kfops_ro;         /*按顺序文件读*/
        }
        else if (sysfs_ops->store)        /*sysfs_ops 实例只定义了写函数，只写属性*/
        {
            if (mode & SYSFS_PREALLOC)
                ops = &sysfs_prealloc_kfops_wo;      /*需预先分配缓存，直接写*/
            else
                ops = &sysfs_file_kfops_wo;         /*不需要预先分配缓存，直接写*/
        }
        else          /*sysfs_ops 实例没有定义读写函数*/
            ops = &sysfs_file_kfops_empty;

        size = PAGE_SIZE;        /*文件大小，最大为一页*/
    }
    else {                  /*二进制属性*/
        struct bin_attribute *battr = (void *)attr;
        if (battr->mmap)
            ops = &sysfs_bin_kfops_mmap;
        else if (battr->read && battr->write)
            ops = &sysfs_bin_kfops_rw;
        else if (battr->read)
            ops = &sysfs_bin_kfops_ro;
        else if (battr->write)
            ops = &sysfs_bin_kfops_wo;
        else
            ops = &sysfs_file_kfops_empty;

        size = battr->size;
    }

#ifdef CONFIG_DEBUG_LOCK_ALLOC
```

```
    ...
#endif
    kn = __kernfs_create_file(parent, attr->name, mode & 0777, size, ops,(void *)attr, ns, key);
                        /*创建并添加表示文件的 kernfs_node 实例，实例 priv 成员指向属性实例*/
    ...     /*处理错误*/
    return 0;
}
```

sysfs_add_file_mode_ns()函数根据属性类型及 sysfs_ops 实例中的函数定义确定表示文件 kernfs_node 实例关联的 kernfs_ops 实例，调用__kernfs_create_file()函数创建并添加表示文件的 kernfs_node 实例。以上各 kernfs_ops 实例定义在/fs/sysfs/file.c 文件内，后面将简要介绍。

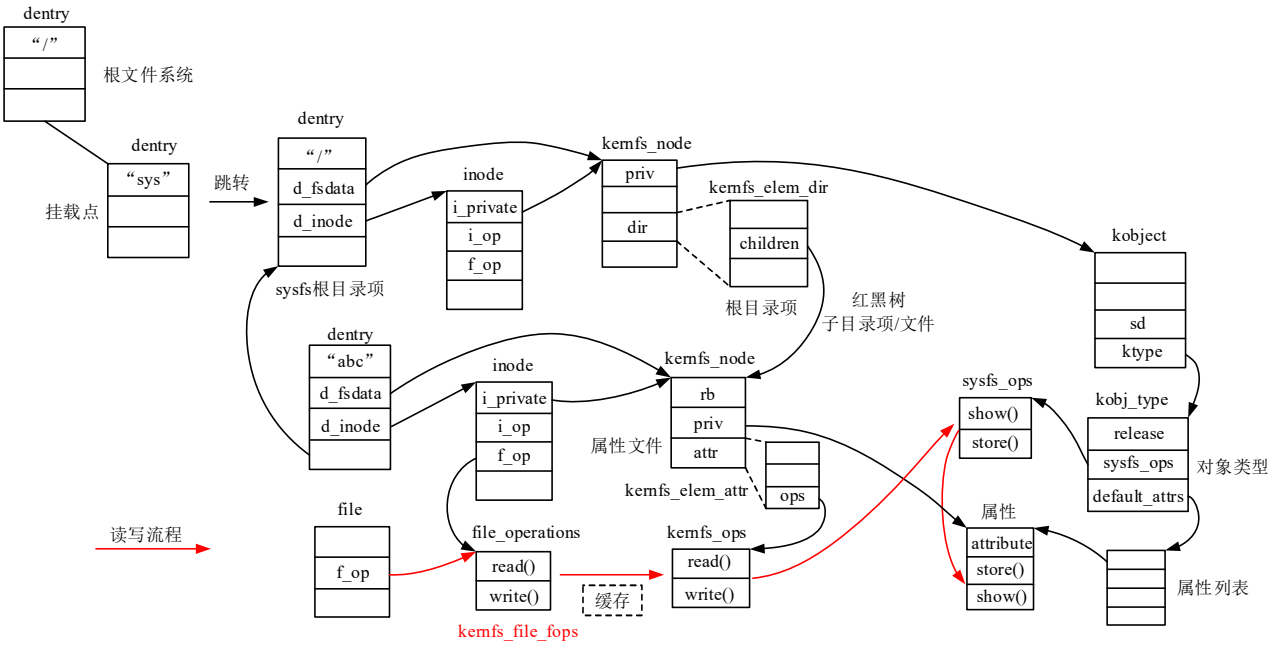
内核还定义了 **sysfs_create_files**(struct kobject *kobj, const struct attribute **ptr)函数用于将属性列表添加到 sysfs 文件系统，函数内遍历属性列表对每个属性调用 sysfs_create_file()函数。

在 populate_dir()函数中只添加了 kobj_type 实例中指定的默认属性，在调用 kobject_add()函数添加 kobject 实例后，还可以调用 **sysfs_create_file()/sysfs_create_files()**函数向 kobject 实例添加属性（组），导出到 sysfs 文件系统。

3 读写属性文件

在 kobject_add()函数中会在 sysfs 文件系统中添加表示 kobject 实例的目录项节点，以及表示属性的文件节点。调用 kobject_add()函数后，还可以调用 sysfs_create_file()/sysfs_create_files()函数为 kobject 实例添加额外的属性（组）文件。

用户挂载 sysfs 文件系统后，可对属性文件进行读写操作。sysfs 文件系统中打开文件、读写文件的操作采用 kernfs 文件系统定义的操作。下图示意了打开、操作 sysfs 文件的流程：



属性文件 file 实例关联的 file_operations 实例为 kernfs_file_fops，详见上一节。kernfs_file_fops 实例中读写函数将最终调用 kernfs_node 关联的 **kernfs_ops** 实例中的读写函数完成操作。

sysfs 文件系统中 kernfs_node 实例关联的 kernfs_ops 实例定义在/fs/sysfs/file.c 文件内，在添加属性文件时自动赋值。下面列出部分 kernfs_ops 实例定义：

```
static const struct kernfs_ops sysfs_file_kfops_ro = {        /*只读，按顺序文件读*/
        .seq_show       = sysfs_kf_seq_show,
};

static const struct kernfs_ops sysfs_file_kfops_wo = {        /*只写，直接写*/
        .write          = sysfs_kf_write,
};

static const struct kernfs_ops sysfs_file_kfops_rw = {        /*可读写属性，顺序文件读，直接写*/
        .seq_show       = sysfs_kf_seq_show,
        .write          = sysfs_kf_write,
};
```

sysfs 文件系统中文件 file 关联的 file_operations 实例为 kernfs_file_fops，定义如下（/fs/kernfs/file.c）：

```
const struct file_operations kernfs_file_fops = {
        .read       = kernfs_fop_read,          /*读操作函数，/fs/kernfs/file.c*/
        .write      = kernfs_fop_write,         /*写操作函数，/fs/kernfs/file.c*/
        ...
};
```

kernfs_file_fops 实例中的读写函数上一节介绍过了，读写函数将调用 kernfs_ops 实例中的函数完成操作。下面以可读写属性文件为例，简要介绍 sysfs_file_kfops_rw 实例中读写操作函数的实现。

■读操作

对于可读写属性，其 kernfs_node 实例关联的 kernfs_ops 结构体实例 **sysfs_file_kfops_rw** 中定义了 seq_show()函数。由前一节介绍的 kernfs_fop_read()函数将调用通用函数 **seq_read()**完成读操作。

seq_read()函数最终将调用 sysfs_file_kfops_rw()实例中的迭代器函数，获取数据写入缓存，然后将数据复制到用户内存。

在 sysfs_file_kfops_rw()实例中只定义了 seq_show()函数，也就是说只需要执行单次的对象读操作，不需要迭代。seq_show()函数为 sysfs_kf_seq_show()，定义如下（/fs/sysfs/file.c）：

```
static int sysfs_kf_seq_show(struct seq_file *sf, void *v)
{
        struct kernfs_open_file *of = sf->private;
        struct kobject *kobj = of->kn->parent->priv;
        const struct sysfs_ops *ops = sysfs_file_ops(of->kn);
                                        /*从父节点获取 kobject 实例，返回 kobj->ktype->sysfs_ops*/
        ssize_t count;
        char *buf;

        count = seq_get_buf(sf, &buf);
        if (count < PAGE_SIZE) {
                seq_commit(sf, -1);
                return 0;
        }
```

```
    memset(buf, 0, PAGE_SIZE);      /*缓存清零*/


    if (ops->show) {
        count = ops->show(kobj, of->kn->priv, buf);   /*调用 sysfs_ops.show()函数，读取数据至缓存*/
        ...
    }

    if (count >= (ssize_t)PAGE_SIZE) {
        print_symbol("fill_read_buffer: %s returned bad count\n",(unsigned long)ops->show);
        count = PAGE_SIZE - 1;
    }
    seq_commit(sf, count);
    return 0;
}
```

sysfs_kf_seq_show()函数调用 sysfs_ops 实例中的 show()函数，获取数据写入顺序文件缓存。seq_read()函数随后将缓存数据复制到用户空间。

■写操作

kernfs 文件系统中文件操作结构实例 kernfs_file_fops 中的 write()函数 kernfs_fop_write()将用户数据写入 kernfs_open_file 实例关联的缓存，然后调用 kernfs_ops 实例中的 write()函数，将数据写入内核。

sysfs_file_kfops_rw 实例中写操作函数为 sysfs_kf_write()，定义如下（/fs/sysfs/file.c）：

```
static ssize_t sysfs_kf_write(struct kernfs_open_file *of, char *buf,size_t count, loff_t pos)
{
    const struct sysfs_ops *ops = sysfs_file_ops(of->kn);
    struct kobject *kobj = of->kn->parent->priv;
    if (!count)
        return 0;
    return ops->store(kobj, of->kn->priv, buf, count);      /*调用 sysfs_ops->store()函数*/
}
```

sysfs_kf_write()函数最后调用 sysfs_ops 实例中的 store()函数，最终调用特定于属性的 store()函数完成属性内容（值）的写操作。

在第 8 章介绍的通用驱动模型中，将频繁使用 sysfs 文件系统。

7.14 管道与命名管道

管道（pipe）和命名管道是进程间通信的机制，用于进程间单向的数据传输。管道和命名管道的两端分别是写进程和读进程。管道和命名管道本质上是一种特殊的文件，其文件内容用于进程之间传递数据。文件内容保存在一个缓存区中，缓存区相当于一个先进先出（FIFO）队列，先写入的数据先读出。

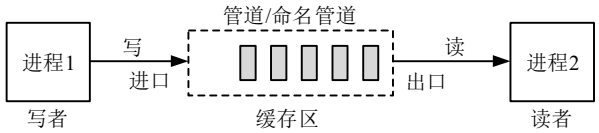
管道没有文件名称，由内核管理，只能用于同源的进程间（fork()出来的进程间）通信，对其它非同源的进程不可见。命名管道与普通文件一样具有文件名称，文件保存在具体文件系统中，导出到内核根文件

系统，对用户可见，可用于任意进程之间的通信。

本节介绍管道与命名管道的创建，以及读写操作的实现，管道/命名管道实现代码位于/fs/pipe.c 文件内。

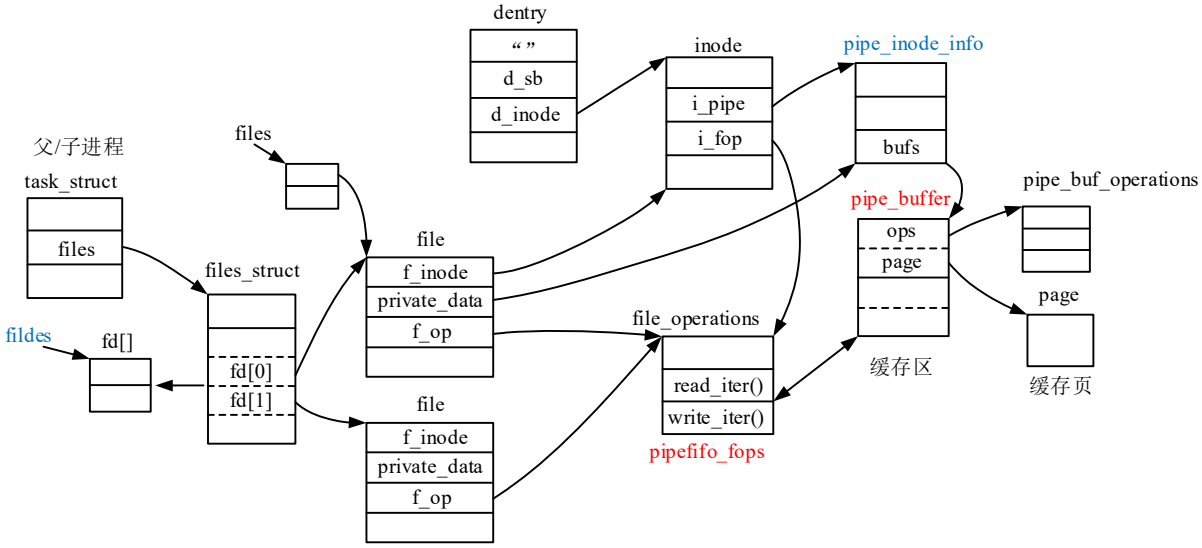
7.14.1 概述

下图示意了进程间通过管道/命名管道传递数据的过程。管道/命名管道包含一个缓存区，缓存区可认为有一个进口和一个出口，写进程从进口写入数据，读进程从出口读取数据，数据在缓存区中按写入时间先后顺序排列，先进的数据先读出，只能按顺序读取，不能任意读取。



由于缓存区的大小是有限制的（固定的），当缓存区写满时，将循环从缓存区开头处开始写。读操作也类似，读到缓存区末尾后，再从缓存区开头开始读取。

管道结构如下图所示，进程通过 pipe()/pipe2()系统调用创建管道，将返回 2 个文件描述符，一个用于读端（只读），一个用于写端（只写），2 个 file 实例关联到同一个 inode 实例。管道由 pipe_inode_info 结构体表示，缓存区由 pipe_buffer 结构体数组表示，每个结构体关联一个缓存页用于保存数据。



父进程创建子进程后，表示管道的两个文件描述符将传递给子进程。如果父进程要通过管道向子进程传递数据，则关闭父进程读端文件描述符和子进程写端文件描述符。父进程则可以向管道写入数据，子进程可从管道读取数据。

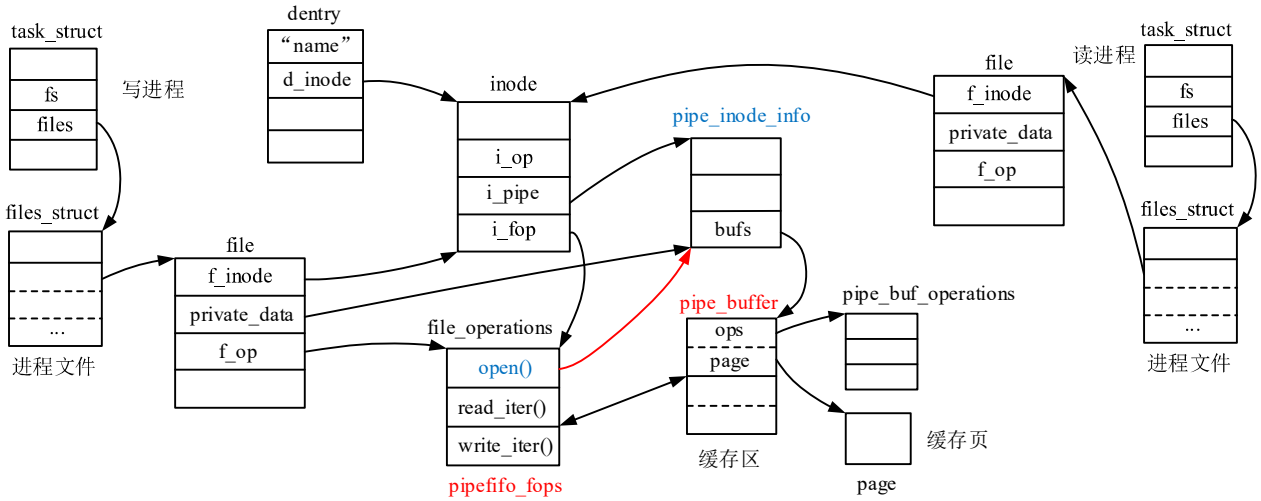
管道文件关联的 file_operations 实例为 pipefifo_fops，其中的读写函数负责向缓存区中写入数据和读取数据。

管道文件没有文件名，其对应的 dentry 实例名称为空，且没有关联到内核根文件系统。管道文件由 pipefs 伪文件系统管理（图中未画出）。进程关闭时管道将销毁，管道只能用于同源进程之间的数据传输。

命名管道克服了管道只能实现同源进程之间数据传输的缺陷。命名管道其实就是保存在具体文件系统中的一种特殊文件，如同设备文件一样。在具体文件系中只保存了命名管道的名称等元数据，而没有保存文件内容，文件内容保存在内存缓存区中，在读写操作中动态产生，同管道一样。

命名管道结构如下图所示。用户进程可通过 mknod()/mknodat()系统调用，或 mkfifo()库函数创建命名管道。命名管道作为一种特殊文件，其关联的 file_operations 实例为 pipefifo_fops（同管道一样）。在打开命名管道时，将调用 pipefifo_fops 实例中的 open()函数，此函数将为 inode 实例创建 pipe_inode_info 实例，表示命名管道（同管道一样）。读写进程分别以只读、只写形式打开命名管道，写进程可向命名管道写入

数据，读进程可从命名管道读取数据。另外，进程也可以以读写形式打开命名管道，即进程可自己给自己发送数据。



7.14.2 管道

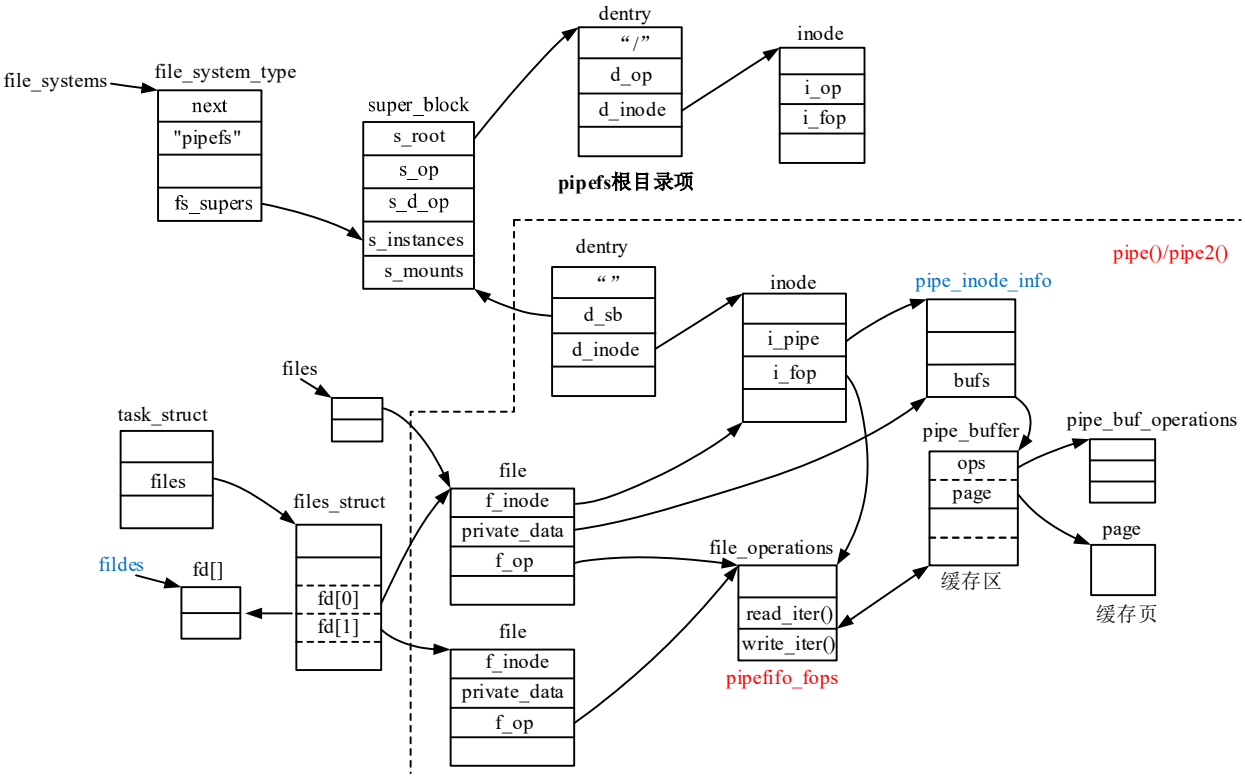
管道实际上就是只存在于内存中的文件。内核定义了 pipefs 伪文件系统用于管理管道。进程创建一个管道后，会获得两个文件描述符，分别用于对管道进行读取和写入操作。通常这两个文件描述符称为管道的读取端和写入端，从写入端写入管道的数据都可以从读取端读取。

进程调用 pipe()/pipe2()系统调用创建一个管道后，还不能实现通过管道在两个进程间通信的目的，因为此时管道的读取端和写入端的文件描述符都属于同一个进程。

父进程通过 fork()系统调用创建子进程后，父进程中打开的文件描述符将传递给子进程。实际的使用中，常常在子进程中调用 execve()等系统调用执行指定的程序，然后根据数据传输的方向分别关闭父进程和子进程中的一个文件描述符。例如：要实现父进程向子进程传输数据，则关闭父进程中的读取端文件描述符和子进程中的写入端文件描述符，父进程成为写端，子进程成为读端。

内核为管道定义了 pipefs 伪文件系统，名义上用于管理管道。pipefs 伪文件系统只能由内核挂载，不能由用户挂载，文件系统内容不导出到内核根文件系统，对用户不可见。

内核在初始化时对 pipefs 伪文件系统执行内核挂载操作。进程通过 pipe()/pipe2()系统调用创建的管道如下图所示：



pipe()/pipe2()系统调用中分配了 2 个文件描述符，创建立了 2 个 file 实例，它们都关联到同一个表示管道的 inode 实例，pipe_inode_info 结构体表示管道信息，pipe_buffer 结构体数组表示缓存区，用于保存管道文件内容，file 关联的文件操作结构实例为 pipefifo_fops，进程通过此实例中的读写函数读写管道。

管道是没有名称的，即其 dentry 实例名称之空。dentry 实例没有添加到 pipefs 文件系统根目录项之下的层次结构中，也就是说表示管道的 dentry 和 inode 实例是单独存在的。

1 pipefs 伪文件系统

pipefs 伪文件系统类型 pipe_fs_type 实例定义如下（/fs/pipe.c）：

```
static struct file_system_type pipe_fs_type = {
    .name       = "pipefs",     /*文件系统类型名称*/
    .mount      = pipefs_mount,     /*挂载函数*/
    .kill_sb    = kill_anon_super,
};
```

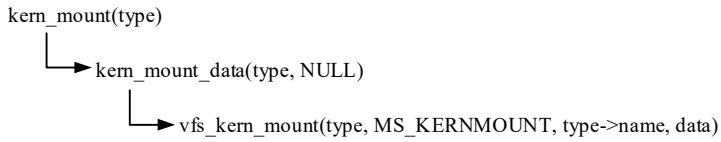
pipefs 伪文件系统初始化函数 init_pipe_fs()中注册了文件系统类型，并执行了内核挂载，函数定义如下：

```
static int __init init_pipe_fs(void)
{
    int err = register_filesystem(&pipe_fs_type);       /*注册文件系统类型*/

    if (!err) {
        pipe_mnt = kern_mount(&pipe_fs_type);    /*内核挂载 pipefs 伪文件系统，/include/linux/fs.h*/
        ...     /*挂载出错处理，注销文件系统类型实例*/
    }
    return err;
```

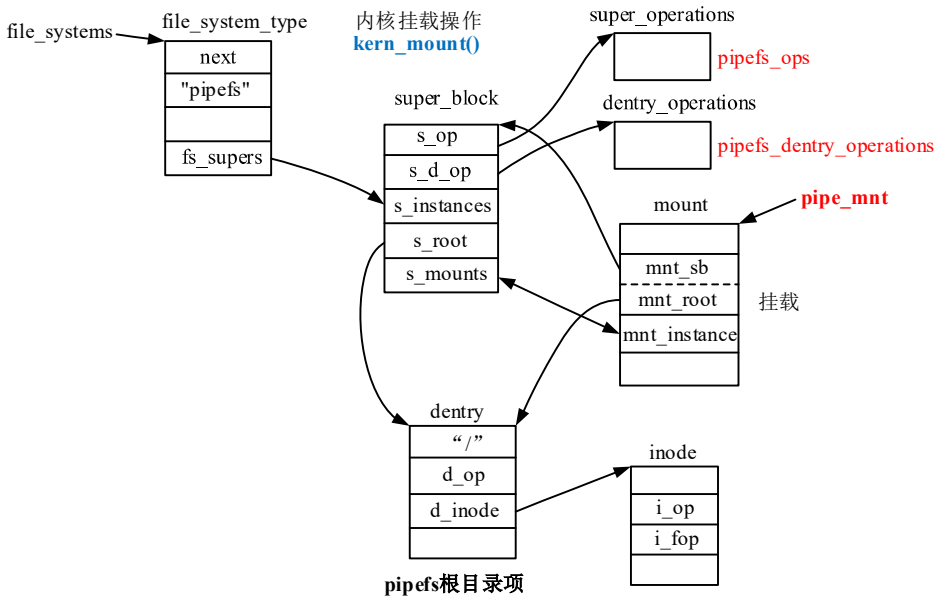
```
}
fs_initcall(init_pipe_fs);
```

init_pipe_fs()函数的主要工作是注册 pipefs 伪文件系统类型，并执行内核挂载操作。

init_pipe_fs()函数中调用了内核挂载函数 kern_mount()，函数调用关系如下图所示：



vfs_kern_mount()函数在前面介绍过了，主要工作是创建挂载 mount 结构体实例，调用文件系统类型中定义的 mount()函数，创建超级块 super_block、根目录项 dentry 和 inode 结构体实例，并建立以上数据结构实例之间的关系，执行结果如下图所示。



pipefs 伪文件系统类型 pipe_fs_type 实例中的挂载函数为 pipefs_mount()，定义如下：

```
static struct dentry *pipefs_mount(struct file_system_type *fs_type,
                                   int flags, const char *dev_name, void *data)
{
    return mount_pseudo(fs_type, "pipe:", &pipefs_ops,&pipefs_dentry_operations, PIPEFS_MAGIC);
                                   /*挂载伪文件系统通用函数，/fs/libfs.c*/
}
```

pipefs_mount()函数内调用挂载伪文件系统的通用函数 mount_pseudo()，此函数定义在/fs/libfs.c 文件内，主要工作是创建超级块 super_block 实例，设置超级块的 **MS_NOUSER** 标记位，创建并设置根目录项 dentry 和 inode 实例，inode 实例没有关联文件操作结构和节点操作结构实例。

pipefs_mount()函数参数 pipefs_ops 表示超级块操作结构 super_operations 实例，pipefs_dentry_operations 表示目录项操作结构 dentry_operations 实例。

mount_pseudo()函数中创建的超级块 super_block 实例设置了 MS_NOUSER 标记位，设置此标记位的文件系统不能由用户挂载（若挂载将在 graft_tree()函数中返回无效错误码），只能由内核挂载，文件系统对用户不可见。

2 创建管道

用户进程创建管道的系统调用为 pipe()/pipe2()，实现如下：

```
SYSCALL_DEFINE1(pipe, int __user *, fildes)
{
    return sys_pipe2(fildes, 0);       /*pipe2()系统调用实现函数*/
}
```

pipe2()系统调用实现函数定义如下：

```
SYSCALL_DEFINE2(pipe2, int __user *, fildes, int, flags)
/*flags：标记，为 0，或 O_CLOEXEC 、O_NONBLOCK 、O_DIRECT 标记（可同时设置多个）*/
{
    struct file *files[2];        /*file 实例指针数组*/
    int fd[2];            /*文件描述符数组*/
    int error;

    error = __do_pipe_flags(fd, files, flags);
    if (!error) {
        if (unlikely(copy_to_user(fildes, fd, sizeof(fd)))) { /*复制文件描述符至用户空间，成功返回 0*/
            ...    /*复制失败时的处理*/
        } else {
            fd_install(fd[0], files[0]);        /*绑定文件描述符与 file 实例*/
            fd_install(fd[1], files[1]);
        }
    }
    return error;
}
```

pipe()/pipe2()系统调用将返回 2 个文件描述符，参数 fildes 指向的数组用于存放返回的文件描述符，fildes[0]表示读取端文件描述符（只读形式），fildes[1]表示写入端文件描述符（只写形式）。

系统调用内通过__do_pipe_flags()函数创建表示管道的 dentry 和 inode 实例，分配并初始化表示管道的 pipe_inode_info 结构体实例，分配 2 个 file 实例，2 个文件描述符，2 个 file 实例关联到同一个 inode 实例，file 和 inode 关联的文件操作结构实例为 **pipefifo_fops**。系统调用内最后将文件描述符写入到 fildes[]数组，建立进程文件描述符与 file 实例之间的关联，函数源代码请读者自行阅读。

用户进程在调用 pipe()/pipe2()系统调用后，表示管道读写端的 2 个文件描述符将传递给随后创建的子进程。

3 读写操作

在讲解管道读写操作前，先介绍表示管道信息的 pipe_inode_info 结构体的定义。

pipe_inode_info 结构体定义如下（/include/linux/pipe_fs_i.h）：

```
struct pipe_inode_info {
    struct mutex mutex;        /*互斥锁，读写管道时需要持有此锁*/
    wait_queue_head_t wait;        /*读写操作等待队列，在管道上睡眠等待的进程*/
    unsigned int nrbufs, curbuf, buffers;
```

```
                /*非空缓存数量（bufs 成员指向数组），当前读取缓存（数组项索引），总缓存数量*/
    unsigned int readers;       /*当前读者数量，初始化为 1*/
    unsigned int writers;       /*当前写者数量，初始化为 1*/
    unsigned int files;         /*关联 file 实例数量*/
    unsigned int waiting_writers;       /*写阻塞进程数量*/
    unsigned int r_counter;         /*读者数量*/
    unsigned int w_counter;       /*写者数量*/
    struct page *tmp_page;         /*暂存分配的缓存页*/
    struct fasync_struct *fasync_readers;
    struct fasync_struct *fasync_writers;
    struct pipe_buffer *bufs;       /*指向 pipe_buffer 数组，初始默认值为 16 项，表示缓存区*/
};
```

pipe_inode_info 结构体主要成员简介如下：

●**mutex**：互斥锁，对管道缓存区进行读写时，需要持有该锁。

●**wait**：在管道上睡眠等待的进程队列。

●**nrbufs, curbuf, buffers**：这个表成员表示 bufs 指向的 pipe_buffer 数组中的项数，每一项代表一个缓存页。buffers 表示数组总的项数，初始默认值为 16。curbuf 表示当前读取缓存在 pipe_buffer 数组中的索引值。nrbufs 表示有数据尚未读取完的缓存数量（pipe_buffer 数组项数）。

●**bufs**：指向 pipe_buffer 结构体数组，每个数组项包含一个缓存页，表示缓存区。pipe_buffer 结构体定义如下（/include/linux/pipe_fs_i.h）：

```
struct pipe_buffer {
    struct page *page;          /*指向内存页，缓存数据*/
    unsigned int offset, len;       /*（未读）数据在页中的起始位置偏移量（初始为 0）、数据长度*/
    const struct pipe_buf_operations *ops;       /*缓存区操作结构*/
    unsigned int flags;         /*标记*/
    unsigned long private;       /*pipe_buf_operations 实例私有数据*/
};
```

ops 成员指向的 pipe_buf_operations 结构体定义如下（/include/linux/pipe_fs_i.h）：

```
struct pipe_buf_operations {
    int can_merge;     /*新写入数据是否与当前缓存页合并，否则分配一个新缓存页*/
    int (*confirm)(struct pipe_inode_info *, struct pipe_buffer *);        /*验证缓存区中数据是否好*/
    void (*release)(struct pipe_inode_info *, struct pipe_buffer *);
                                                    /*缓存数据被读取后，调用此函数释放缓存区*/
    int (*steal)(struct pipe_inode_info *, struct pipe_buffer *); /*建立缓存区与内容（page）之间的关联*/
    void (*get)(struct pipe_inode_info *, struct pipe_buffer *);     /*增加缓存区引用计数*/
};
```

在__do_pipe_flags()函数中，最终将调用 alloc_pipe_info()函数为管道分配 pipe_inode_info 实例，并分配缓存区 pipe_buffer 结构体数组。但是此时没有为 pipe_buffer 实例设置 ops 成员和分配缓存页，分配缓存页需要在写管道操作中进行，详见下文。

由 pipe()/pipe2()系统调用可知，管道（文件）关联的文件操作结构实例为 pipefifo_fops，定义如下：

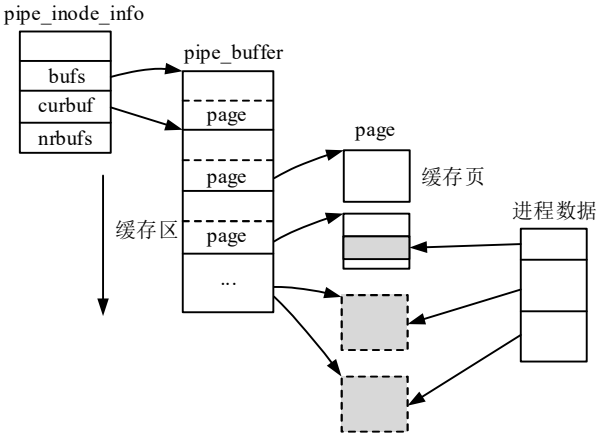
```
const struct file_operations pipefifo_fops = {
```

```
    .open           = fifo_open,    /*打开命名管道，见下一小节*/
    .llseek         = no_llseek,    /*先进先出，顺序读取，不能指定文件位置*/
    .read_iter      = pipe_read,    /*读操作函数*/
    .write_iter     = pipe_write,   /*写操作函数*/
    .poll           = pipe_poll,    /*查询是否可读或可写*/
    .unlocked_ioctl     = pipe_ioctl,
    .release        = pipe_release,
    .fasync         = pipe_fasync,
};
```

读写管道其实就是向管道的缓存区中读取和写入数据，如下图所示。管道缓存区默认包含 16 个缓存页，由 pipe_buffer 结构体数组表示，每个 pipe_buffer 实例包含一个缓存页。缓存区是循环使用的，数据从第一个缓存页开始依次写入缓存区中缓存页，当最后一个缓存页写满时，重新从第一个缓存页开始写。

pipe_inode_info 结构体 curbuf 成员表示当前正在读取数据的 pipe_buffer 实例的数组项索引值（从 0 开始），nrbuf 表示存有数据但未读取完的缓存页数量，已读完的缓存页将被释放，不记入 nrbuf 表示的数量中。也就是说读、写操作分别有自己的路径，curbuf 成员表示读指针，读完一个缓存页后进入下一个缓存页，释放已读完的缓存页。（curbuf+nrbuf-1）表示当前写缓存页数组项索引值。读写操作都是循环使用缓存页的。

如果缓存区都已写满，写进程将进入睡眠等待，等待读进程读取数据后释放缓存区，唤醒写进程。当缓存区中没有数据可读时，读进程进入睡眠等待，等待写进程写入数据，唤醒读进程。



pipefifo_fops 实例中写函数在将进程数据写入缓存区时，将用户数据按页进行划分，如果数据不是页对齐的，将判断按页划分后剩余的数据能否写入 pipe_buffer 数组中最后具有数据的缓存区（ops->can_merge 需不为 0），如果可以则写入（缓存页中需要有足够的空闲空间）。进程数据剩下的整页的数据写入后面的缓存页。

例如，假设上图中的进程数据按页划分后还剩 100 字节，pipe_buffer 数组中数据已经写到了第 3 个缓存页，且缓存页中还有 200 字节的空闲空间，则向此缓存页写入 100 字节数据（注意不是写 200 字节），进程数据剩下的整页的数据写入后面的缓存页。如果第 3 个缓存页中只剩余 50 字节（小于 100 字节）空间，则进程数据不写入此缓存页，而从下一个缓存页开始写入。

下面将介绍写入、读取函数的实现。

■写操作函数

管道要先对其进行写操作，然后才能进行读操作，因此这里先介绍写操作的实现。pipefifo_fops 实例中写操作函数定义如下：

```
static ssize_t pipe_write(struct kiocb *iocb, struct iov_iter *from)
{
    struct file *filp = iocb->ki_filp;
    struct pipe_inode_info *pipe = filp->private_data;      /*pipe_inode_info 实例*/
    ssize_t ret = 0;
    int do_wakeup = 0;
    size_t total_len = iov_iter_count(from);        /*写数据长度*/
    ssize_t chars;

    if (unlikely(total_len == 0))       /*长度为 0，返回*/
        return 0;

    __pipe_lock(pipe);     /*获取 pipe->mutex 锁*/

    if (!pipe->readers) {       /*读者数量必须非 0，初始化时设为 1*/
        ...
    }

    /*试图合并写操作*/
    chars = total_len & (PAGE_SIZE-1);      /*数据按页划分后，剩余的字节数*/
    if (pipe->nrbufs && chars != 0) {       /*有非空缓存页，且 chars 非零（nrbufs 初始值为 0）*/
        int lastbuf = (pipe->curbuf + pipe->nrbufs - 1) &(pipe->buffers - 1);
                                                        /*当前写缓存页索引值*/
        struct pipe_buffer *buf = pipe->bufs + lastbuf;     /*指向当前写缓存页*/
        const struct pipe_buf_operations *ops = buf->ops;
        int offset = buf->offset + buf->len;       /*向当前写缓存页写入数据的起始偏移量*/

        if (ops->can_merge && offset + chars <= PAGE_SIZE) {
                    /*如果当前写缓存页剩余空间可以容下进程数据按页划分后剩余的字节数*/
            int error = ops->confirm(pipe, buf);       /*缓存页数据验证*/
            ...
            ret = copy_page_from_iter(buf->page, offset, chars, from);       /*复制进程数据至缓存页*/
            ...
            do_wakeup = 1;
            buf->len += chars;     /*增加缓存页中数据长度*/
            ret = chars;
            if (!iov_iter_count(from))     /*如果进程数据复制完了，则跳至 out 处*/
                goto out;
        }
    }
    /*写入剩下的整页的进程数据至后续缓存页，或者从后续缓存页开始写入进程数据*/
    for (;;) {      /*循环开始，分配缓存页，写入进程数据至缓存页*/
        int bufs;
```

```
if (!pipe->readers) {
        ...
}
bufs = pipe->nrbufs;
if (bufs < pipe->buffers) {     /*如果缓存页未用完*/
        int newbuf = (pipe->curbuf + bufs) & (pipe->buffers-1);     /*下一个写入缓存页的索引值*/
        struct pipe_buffer *buf = pipe->bufs + newbuf;      /*pipe_buffer 实例*/
        struct page *page = pipe->tmp_page;
        int copied;

        if (!page) {
                page = alloc_page(GFP_HIGHUSER);      /*分配缓存页*/
                ...
                pipe->tmp_page = page;
        }
        do_wakeup = 1;
        copied = copy_page_from_iter(page, 0, PAGE_SIZE, from);      /*复制数据到缓存页*/
        if (unlikely(copied < PAGE_SIZE && iov_iter_count(from))) {
                ...
        }
        ret += copied;      /*写入数据数量*/

        /*缓存页关联到 pipe_buffer 实例*/
        buf->page = page;
        buf->ops = &anon_pipe_buf_ops;      /*赋值 pipe_buf_operations 实例*/
        buf->offset = 0;
        buf->len = copied;
        buf->flags = 0;
        if (is_packetized(filp)) {     /*file 设置了 O_DIRECT 标记位（直接读），/fs/pipe.c*/
                buf->ops = &packet_pipe_buf_ops;
                buf->flags = PIPE_BUF_FLAG_PACKET;
        }
        pipe->nrbufs = ++bufs;     /*非空缓存页数量加 1*/
        pipe->tmp_page = NULL;

        if (!iov_iter_count(from))     /*进程数据复制完，则跳出循环*/
                break;
}
if (bufs < pipe->buffers)      /*还有缓存页可以写数据，循环继续*/
        continue;

/*所有缓存页都已写满*/
```

```
            if (filp->f_flags & O_NONBLOCK) {        /*文件设置了非阻塞操作*/
                if (!ret)
                    ret = -EAGAIN;
                break;      /*跳出循环*/
            }

            if (signal_pending(current)) {     /*有挂起信号，跳出循环*/
                if (!ret)
                    ret = -ERESTARTSYS;
                break;
            }

            if (do_wakeup) {     /*写入了新数据，do_wakeup 就设为 1*/
                wake_up_interruptible_sync_poll(&pipe->wait, POLLIN | POLLRDNORM);
                                                                    /*唤醒等待读进程*/
                kill_fasync(&pipe->fasync_readers, SIGIO, POLL_IN);      /*发送信号*/
                do_wakeup = 0;
            }
            pipe->waiting_writers++;
            pipe_wait(pipe);      /*睡眠等待，TASK_INTERRUPTIBLE*/
            pipe->waiting_writers--;
        }    /*for 循环结束*/
    out:
        __pipe_unlock(pipe);
        if (do_wakeup) {        /*唤醒在管道上睡眠的读进程*/
            wake_up_interruptible_sync_poll(&pipe->wait, POLLIN | POLLRDNORM);
            kill_fasync(&pipe->fasync_readers, SIGIO, POLL_IN);         /*向读者发送信号*/
        }
        if (ret > 0 && sb_start_write_trylock(file_inode(filp)->i_sb)) {
            int err = file_update_time(filp);
            ...
            sb_end_write(file_inode(filp)->i_sb);
        }
        return ret;      /*返回写入数据数量*/
    }
```

由以上函数可知，写操作将进程数据按页写入到缓存页，如果所有缓存页都写满了，写操作进程将进入睡眠等待，等待读取操作释放缓存页后唤醒写进程。写操作向缓存页写入了新数据后会唤醒在管道上睡眠等待的读操作进程（还会向读进程发送信号）。

■读操作函数

pipefifo_fops 实例中管道读操作函数定义如下：

```
static ssize_t pipe_read(struct kiocb *iocb, struct iov_iter *to)
{
```

```
size_t total_len = iov_iter_count(to);     /*欲读取数据长度*/
struct file *filp = iocb->ki_filp;
struct pipe_inode_info *pipe = filp->private_data;       /*pipe_inode_info 实例*/
int do_wakeup;
ssize_t ret;

if (unlikely(total_len == 0))       /*读取长度为 0，返回*/
      return 0;

do_wakeup = 0;
ret = 0;
__pipe_lock(pipe);       /*持有锁*/
for (;;) {       /*循环开始，遍历缓存页，读取其中数据，读完的缓存页将释放*/
      int bufs = pipe->nrbufs;        /*非空缓存页数量*/
      if (bufs) {
            int curbuf = pipe->curbuf;        /*当前读缓存页索引值*/
            struct pipe_buffer *buf = pipe->bufs + curbuf;       /*指向当前读缓存页*/
            const struct pipe_buf_operations *ops = buf->ops;       /*缓存操作结构*/
            size_t chars = buf->len;     /*缓存页中数据长度*/
            size_t written;
            int error;

            if (chars > total_len)       /*缓存页中数据大于欲读数据长度*/
                  chars = total_len;

            error = ops->confirm(pipe, buf);        /*验证数据*/
            ...

            written = copy_page_to_iter(buf->page, buf->offset, chars, to);     /*复制数据至进程空间*/
            ...
            ret += chars;
            buf->offset += chars;
            buf->len -= chars;

            if (buf->flags & PIPE_BUF_FLAG_PACKET) {
                  total_len = chars;
                  buf->len = 0;
            }

            if (!buf->len) {        /*如果缓存页中数据被读完，释放缓存页*/
                  buf->ops = NULL;
                  ops->release(pipe, buf);
                  curbuf = (curbuf + 1) & (pipe->buffers - 1);     /*设置当前读缓存页索引值*/
```

```
                pipe->curbuf = curbuf;
                pipe->nrbufs = --bufs;
                do_wakeup = 1;
            }
            total_len -= chars;
            if (!total_len)      /*读取了需要长度的数据，跳出循环*/
                break;
        }
        if (bufs)     /*还需要读数据，且还有非空缓存页，继续循环*/
            continue;

        /*还需要读数据，但已没有非空缓存页，需要等待写者写入数据*/
        if (!pipe->writers)      /*没有写者，跳出循环*/
            break;
        if (!pipe->waiting_writers) {      /*没有等待的写者，且设置了非阻塞，跳出循环*/
            if (ret)
                break;
            if (filp->f_flags & O_NONBLOCK) {
                ret = -EAGAIN;
                break;
            }
        }
        /*有挂起的信号，跳出循环*/
        if (signal_pending(current)) {
            if (!ret)
                ret = -ERESTARTSYS;
            break;
        }
        if (do_wakeup) {        /*唤醒睡眠等待的写进程*/
            wake_up_interruptible_sync_poll(&pipe->wait, POLLOUT | POLLWRNORM);
            kill_fasync(&pipe->fasync_writers, SIGIO, POLL_OUT);     /*发送信号*/
        }
        pipe_wait(pipe);       /*读进程进入睡眠等待*/
    }      /*循环结束*/
    __pipe_unlock(pipe);        /*释放锁*/

    /*唤醒睡眠等待写进程*/
    if (do_wakeup) {
        wake_up_interruptible_sync_poll(&pipe->wait, POLLOUT | POLLWRNORM);
        kill_fasync(&pipe->fasync_writers, SIGIO, POLL_OUT);
    }
    if (ret > 0)        /*读到了数据*/
        file_accessed(filp);
```

```
    return ret;        /*返回读取数据字节数*/
}
```

读操作函数循环遍历缓存区中具有数据的缓存页，依次读取其中数据至进程空间，读完的缓存页将被释放。如果缓存区中没有足够的数据，且未设置 O_NONBLOCK（非阻塞）标记位，读进程将进入睡眠等待（还会发送信号），等待写进程写入数据后再读取。

7.14.3 命名管道

管道没有名字，只能用于同源的进程之间传递数据。命名管道克服了管道的缺点，可以在任意进程之间传递数据（包括进程自己与自己通信）。命名管道是一种特殊的文件，类似于设备文件，它具有文件名，保存在具体文件系中。

创建命名管道时，命名管道的文件名等元信息保存在具体文件系统中（目录项和节点），系统关闭后，命名管道仍然存在。删除命名管道和删除普通文件一样，需要通过 unlink()系统调用删除。

命名管道与普通文件的区别是，用户进程通常以只读或只写的形式打开命名管道，但也允许以读写的形式打开命名管道。命名管道的文件内容管理同管道一样，由内存缓存区管理，文件内容没有保存到具体文件系统中。

1 创建命名管道

用户进程可以通过 mknod()/mknodat()系统调用，或 mkfifo()库函数创建命名管道，函数原型如下所示：

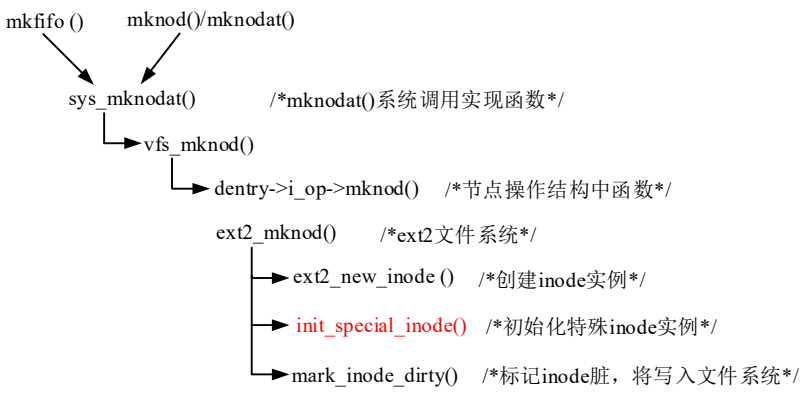
```
int mknod(const char*filename,mode_t mode|S_IFIFO,(dev_t)0);
int mkfifo(const char*filename,mode_t mode);
```

mknod()/mknodat()系统调用是老的接口，如同创建设备文件的操作，filename 表示命名管道名称，mode 参数需指定文件类型为 **S_IFIFO**，第三个设备号参数需为 0。

mkfifo()是一个 glibc 库函数，函数内通过 mknodat()系统调用创建命名管道。mkfifo()函数代码如下所示：

```
int mkfifo (const char *path, mode_t mode)      /*glibc 库函数*/
{
  return __mknod (path, mode | S_IFIFO, 0);      /*调用 mknodat()系统调用*/
}
```

以上函数调用关系如下图所示：



mknodat()系统调用中最终调用文件系统类型定义的目录项节点操作结构中的 mknod()函数创建特殊文

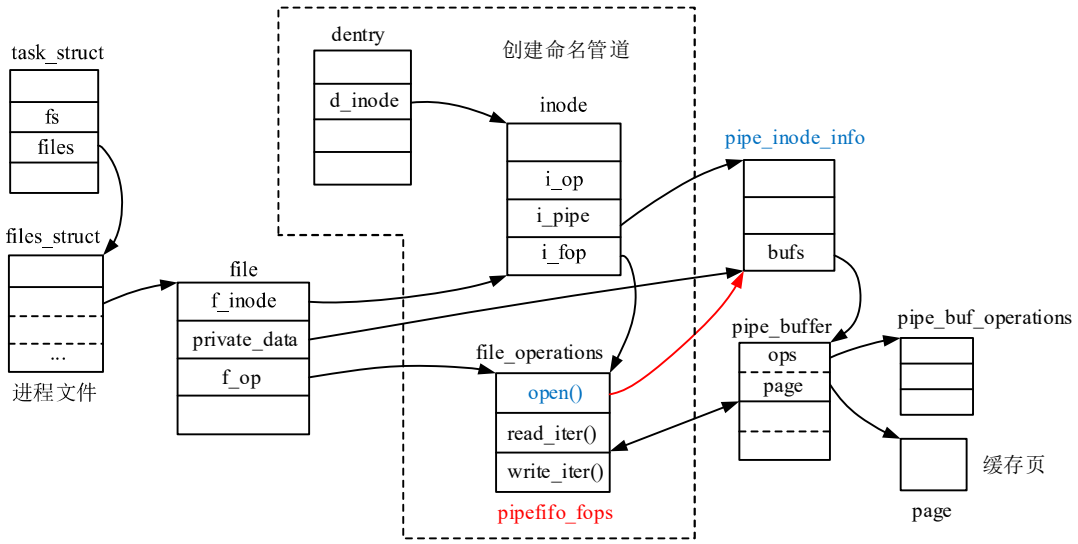
件，其中包括命名管道。

例如，ext2 文件系统类型中 mknod()函数为 ext2_mknod()函数，函数内调用 ext2_new_inode ()函数创建 inode 实例，调用 **init_special_inode()**函数初始化特殊文件的 inode 实例（详见下文），最后设置节点脏标记，节点信息将写入具体文件系统中。

初始化特殊文件 inode 实例的 init_special_inode()函数代码简列如下（/fs/inode.c）：

```
void init_special_inode(struct inode *inode, umode_t mode, dev_t rdev)
{
    inode->i_mode = mode;
    if (S_ISCHR(mode)) {        /*字符设备文件*/
        inode->i_fop = &def_chr_fops;
        inode->i_rdev = rdev;
    } else if (S_ISBLK(mode)) {        /*块设备文件*/
        inode->i_fop = &def_blk_fops;
        inode->i_rdev = rdev;
    } else if (S_ISFIFO(mode))          /*命名管道*/
        inode->i_fop = &pipefifo_fops;      /*文件操作结构实例*/
    else if (S_ISSOCK(mode))
        ;
    else
        ...
}
```

在 init_special_inode()函数中，命名管道 inode 实例关联的文件操作结构实例设为 **pipefifo_fops**，与管道一样。下图虚线框中示意了创建的命名管道在 VFS 中的结构：



注意，不能通过 open()或 creat()系统调用来创建命名管道，这两个系统调用只能用来创建普通文件。

2 打开命名管道

用户进程使用命名管道前需要执行打开操作，进程通常以只读或只写的形式打开命名管道，但是也允许以读写的方式打开命名管道（允许进程自己与自己通信）。

在 open()系统调用中，将查找创建表示文件的 dentry 和 inode 实例。如果节点表示的是一个特殊文件，

将调用前面介绍的 init_special_inode()函数对 inode 实例进行初始化，主要是设置其文件操作结构实例。

open()系统调用最后会将 inode 实例关联的文件操作结构实例赋予 file 实例，并调用实例中的 open()函数，执行特定于文件的打开操作。

命名管道 inode 实例关联的文件操作结构实例为 **pipefifo_fops**，其中的 open()函数定义如下：

```
static int fifo_open(struct inode *inode, struct file *filp)
{
    struct pipe_inode_info *pipe;          /*指向 pipe_inode_info 实例*/
    bool is_pipe = inode->i_sb->s_magic == PIPEFS_MAGIC;
    int ret;

    filp->f_version = 0;

    spin_lock(&inode->i_lock);
    if (inode->i_pipe) {
        pipe = inode->i_pipe;
        pipe->files++;
        spin_unlock(&inode->i_lock);
    } else {        /*inode->i_pipe 为空，创建 pipe_inode_info 实例*/
        spin_unlock(&inode->i_lock);
        pipe = alloc_pipe_info();        /*创建 pipe_inode_info 实例*/
        ...
        pipe->files = 1;
        spin_lock(&inode->i_lock);
        if (unlikely(inode->i_pipe)) {      /*如果其它进程创建了 pipe_inode_info 实例*/
            inode->i_pipe->files++;
            spin_unlock(&inode->i_lock);
            free_pipe_info(pipe);          /*释放此处创建的实例*/
            pipe = inode->i_pipe;
        } else {
            inode->i_pipe = pipe;
            spin_unlock(&inode->i_lock);
        }
    }
    filp->private_data = pipe;        /*指向 pipe_inode_info 实例*/
    __pipe_lock(pipe);

    /*打开方式检查*/
    filp->f_mode &= (FMODE_READ | FMODE_WRITE);

    switch (filp->f_mode) {
    case FMODE_READ:          /*只读方式打开*/
        pipe->r_counter++;
        if (pipe->readers++ == 0)
```

```
                wake_up_partner(pipe);      /*唤醒命名管道上等待的睡眠进程*/
            ...
            break;

        case FMODE_WRITE:       /*只写方式打开*/
            ret = -ENXIO;
            if (!is_pipe && (filp->f_flags & O_NONBLOCK) && !pipe->readers)
                goto err;

            pipe->w_counter++;
            if (!pipe->writers++)
                wake_up_partner(pipe);
            ...
            break;

        case FMODE_READ | FMODE_WRITE:      /*读写方式打开，自己与自己通信*/
            pipe->readers++;
            pipe->writers++;
            pipe->r_counter++;
            pipe->w_counter++;
            if (pipe->readers == 1 || pipe->writers == 1)
                wake_up_partner(pipe);
            break;

        default:
            ret = -EINVAL;
            goto err;
        }
        __pipe_unlock(pipe);
        return 0;
        ...
    }
```

fifo_open()函数主要是为命名管道创建读写操作所需的数据结构实例，同前面介绍的管道。

命名管道与管道关联的文件操作结构实例是相同的，因此读写操作也是相同的，这里就不再介绍命名管道的读写操作了。

7.15 控制组

控制组（cgroups）是 Linux 内核的一个特性，主要用来对共享资源进行隔离、限制、审计等。控制组提供了一种对进程及其子进程进行集合/分组的机制，可以控制（限制）组内进程对某项资源的使用。

7.15.1 概述

内核支持控制组需选择 CGROUPS 配置选项，控制组公共层代码位于/kernel/cgroup.c 文件内。控制组可以认为是对进程的分组，对分组中进程资源的控制由子系统实现。子系统关联到控制组，控制组内进程资源的使用。子系统相关代码由内核相关子系统实现。例如，物理内存控制组子系统相关代码位于/mm/memcontrol.c 文件内。

1 控制组结构

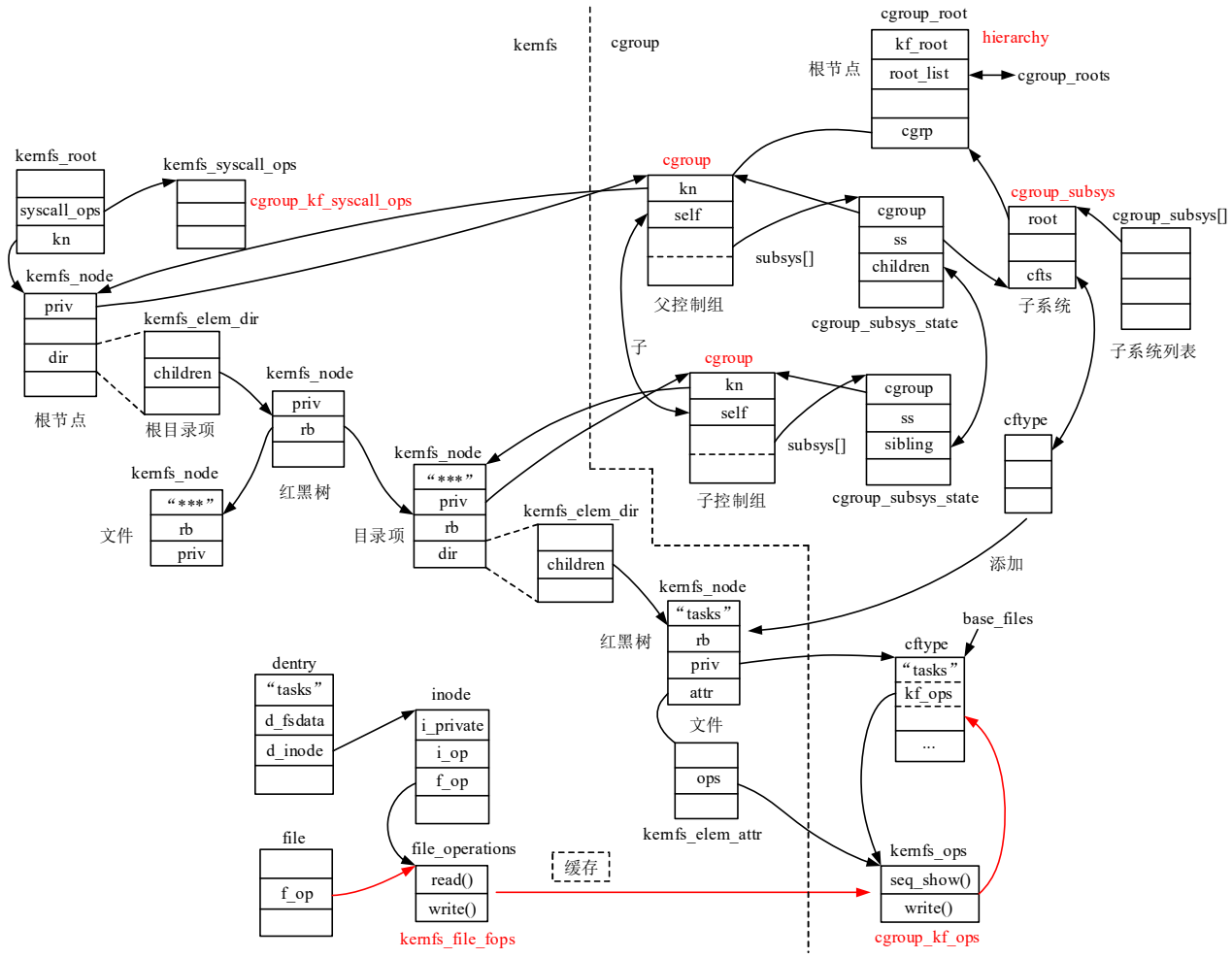
控制组结构如下图所示，先介绍几个概念：

●**hierarchy**：由多个控制组（cgroup）组成的树状层次结构，称之为 hierarchy。内核中可以有多个 hierarchy，内核静态定义了默认的 hierarchy。每个 hierarchy 对应一个 cgroup 文件系统实例，用户挂载 cgroup 文件系统即创建一个 hierarchy。控制组在 cgroup 文件系统中表现为目录项。hierarchy 可以认为是一组控制组的集合。

●**cgroup**：控制组，绑定了一组进程，控制组在 hierarchy 中构成父子层次结构。hierarchy 关联到若干个子系统，每个子系统表示对一种资源的控制（限制），控制组下进程受控制组绑定子系统的控制。

●**subsystem**：子系统，又称资源控制器。子系统控制组内进程的行为。子系统通常用于调度、限制控制组内进程对某种资源的使用，但也可以做其它的事情。

某个子系统只能绑定到一个 hierarchy，而一个 hierarchy 可绑定多个子系统，也就是说可以将多个子系统绑定到同一个 hierarchy。hierarchy 绑定的子系统在其下控制组之间是可继承的。



子系统由 cgroup_subsys 结构体表示，内核静态定义了全局指针数组用于管理 cgroup_subsys 实例。子

系统实例由内核各子系统定义，每个子系统具有一个名称，如内存控制组子系统为“memory”。用户在挂载 cgroup 文件系统（kernfs 文件系统实例），创建 hierarchy 根节点时，可通过挂载参数指定 hierarchy 绑定子系统的名称（可以是多个）。

挂载操作中，在挂载点会自动创建 hierarchy 根节点及对应的控制组。在挂载点下创建目录项，即创建新（子）控制组，子控制组将继承父控制组绑定的子系统。控制组 cgroup 结构体中内嵌 cgroup_subsys_state 结构体成员 self 表示控制组状态，控制组绑定的每个子系统也将创建一个 cgroup_subsys_state 实例，表示子系统在此控制组下的状态。控制组通过内嵌 self 成员构成层次关系，各子系统 cgroup_subsys_state 实例成也构成父子层次关系。

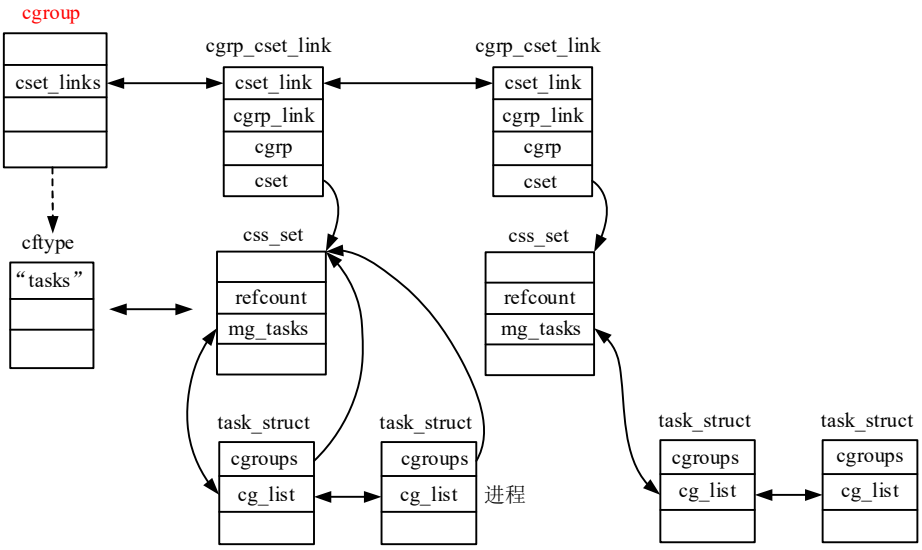
控制组 cgroup 实例对应 cgroup 文件系统中的目录项，而 cftype 实例表示控制组目录项下的文件。cftype 结构体可认为是控制组的参数或子系统参数。内核定义了表示控制组公共参数的 cftype 实例数组，子系统 cgroup_subsys 实例也可关联多个 cftype 实例数组，表示子系统参数。

在创建控制组时，公共 cftype 实例数组以及绑定子系统关联的 cftype 实例数组下的各实例，将在控制组目录项下创建文件。用户进程通过对文件的操作，可控制子系统和控制组的参数。

用户进程对 cftype 实例表示文件的操作，同 kernfs 文件系统中文件的操作，在最后的顺序读/直接写操作中将调用 cftype 实例中定义的函数。

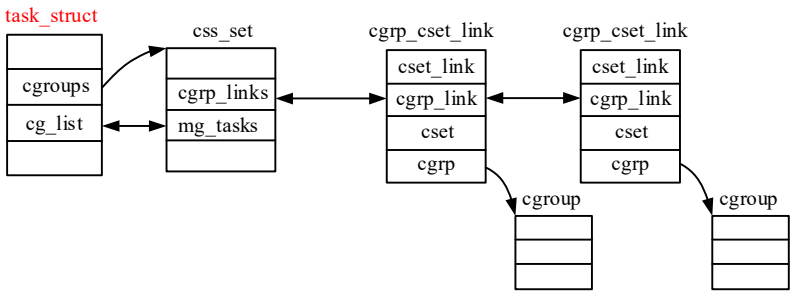
到目前为止，介绍了控制组和子系统，用户通过 cgroup 文件系统操作控制组，子系统通过 cftype 实例表示的文件控制进程的行为。现在还有一个最重要的问题没有介绍，那就是控制组下的进程在哪里？

在内核定义的控制组公共 cftype 实例数组中，有一个名称为“tasks”的 cftype 实例。用户将进程 PID 写入某个控制组下“tasks”文件即将进程移入此控制组，读“tasks”文件即返回控制组下进程 PID。在内核内部，控制组通过以下数据结构管理进程：



进程 task_struct 结构体 cgroups 成员指向 css_set 实例，而 css_set 实例通过 cgrp_cset_link 实例与控制组 cgroup 实例建立关联。css_set 实例还由全局散列表 css_set_table 管理（图中未画出）。

进程可以同时添加到多个控制组，进程通过以下数据结构可查询自己添加进的控制组：



2 使用控制组

内核目前最多支持 12 个控制组子系统，见/include/linux/cgroup_subsys.h 头文件。支持控制组的各内核子系统需定义子系统 cgroup_subsys 实例，名称为 xxx_cgrp_subsys。例如，内存控制组子系统定义如下：

struct cgroup_subsys **memory_cgrp_subsys** __read_mostly;

另外，子系统需关联 cftype 实例数组，表示子系统下的参数，此参数文件将添加到子系统绑定控制组目录项下。

用户通过 cgroup 文件系统实现与控制组的交互。下面简要介绍内核说明文档中控制组使用示例：

```
/*/Documentation/cgroups/cgroups.txt*/
mount -t tmpfs cgroup_root /sys/fs/cgroup        /*挂载 tmpfs 文件系统至/sys/fs/cgroup 目录*/
mkdir /sys/fs/cgroup/cpuset                /*创建目录项*/
mount -t cgroup cpuset -ocpuset /sys/fs/cgroup/cpuset
                        /*在目录项下挂载 cgroup 文件系统，创建 hierarchy，绑定子系统为“cpuset”*/
cd /sys/fs/cgroup/cpuset            /*切换为当前工作目录*/
mkdir Charlie              /*创建子目录项，子控制组 Charlie*/
cd Charlie                 /*切换到子控制组 Charlie 目录项*/
/bin/echo 2-3 > cpuset.cpus     /*向 cpuset.cpus 文件写入“2-3”，表示将 2、3 处理器添加到控制组*/
/bin/echo 1 > cpuset.mems      /*向 cpuset.mems 文件写入“1”，表示将内存节点 1 添加到控制组*/
/bin/echo $$ > tasks         /*当前 shell 进程移入控制组*/
sh        /*子 shell 进程也在此控制组内*/
```

7.15.2 数据结构

本小节主要介绍组成 hierarchy 的根节点 cgroup_root、控制组 cgroup 结构体（cgroup 文件系统中目录项），表示子系统的 cgroup_subsys 结构体，表示子系统参数的 cftypes 结构体（cgroup 文件系统中文件），以及表示控制组、子系统状态的 cgroup_subsys_state 结构体的定义。

1 hierarchy

hierarchy（树状结构）是由多个 cgroup 实例组成的树，此树状结构通过 cgroup 文件系统导出到用户空间。cgroup 文件系统是 kernfs 文件系统的一个实例。在挂载 cgroup 文件系统时，将创建 hierarchy，每个 hierarchy 代表一个 cgroup 文件系统实例。cgroup 文件系统通常挂载到/sys/fs/cgroups/目录下的子目录项。

■cgroup_root

hierarchy 根节点由 cgroup_root 结构体表示，也可以认为是 cgroup 文件系统的根节点，定义如下：

```
struct cgroup_root {          /*/include/linux/cgroup-defs.h*/
    struct kernfs_root *kf_root;        /*指向 kernfs 文件系统中根节点*/
    unsigned int subsys_mask;         /*关联到本 hierarchy 的子系统位图*/
    int hierarchy_id;              /*hierarchy ID 值，由 cgroup_hierarchy_idr 实例管理*/
    struct cgroup cgrp;          /*控制组实例，根目录项 kernfs_node.priv 指向 cgroup 实例*/
    atomic_t nr_cgrps;          /*hierarchy 下控制组数量*/
    struct list_head root_list;       /*将 cgroup_root 实例添加到全局双链表 cgroup_roots*/
    unsigned int flags;        /*hierarchy 标记*/
```

```
    struct idr cgroup_idr;      /*管理 hierarchy 下控制组 ID 值*/

    char release_agent_path[PATH_MAX];
    char name[MAX_CGROUP_ROOT_NAMELEN];     /*hierarchy 名称*/
};
```

cgroup_root 结构体主要成员简介如下：

●**kf_root**：指向 kernfs 文件系统中根节点 kernfs_root 实例。

●**subsys_mask**：hierarchy 关联的子系统位图，每个比特位表示某个 ID 值（由位置表示的值）的子系统。每个子系统只能关联到一个 hierarchy，而一个 hierarchy 可关联一个或多个子系统（也可以不关联子系统）。通常系统设置一个 hierarchy 关联一个子系统。

●**cgrp**：内嵌控制组 cgroup 结构体成员。

●**root_list**：双链表成员，将 cgroup_root 实例添加到全局双链表 cgroup_roots。

●**flags**：标记成员，取值定义如下：

```
enum {
    CGRP_ROOT_SANE_BEHAVIOR     = (1 << 0),   /* __DEVEL__sane_behavior specified */
    CGRP_ROOT_NOPREFIX   = (1 << 1),   /*子系统没有名称前缀*/
    CGRP_ROOT_XATTR      = (1 << 2),   /*支持扩展属性*/
};
```

内核定义了全局双链表用于管理 cgroup_root 实例，如下所示：

```
static LIST_HEAD(cgroup_roots);
```

■cgroup

hierarchy 是控制组的集合，控制组由 cgroup 结构体表示，定义如下（/include/linux/cgroup-defs.h）：

```
struct cgroup {
    struct cgroup_subsys_state self;      /*表示控制组状态，结构体定义见下文*/
    unsigned long flags;          /*标记*/
    int id;      /*控制组 ID 值，由 cgroup_root.cgroup_idr 成员管理*/
    int populated_cnt;

    struct kernfs_node *kn;       /*指向控制组在 kernfs 文件系统中的节点*/
    struct kernfs_node *procs_kn;     /* kn for "cgroup.procs" */
    struct kernfs_node *populated_kn;   /* kn for "cgroup.subtree_populated" */

    unsigned int subtree_control;     /*位图*/
        /*由"cgroup.subtree_control"文件指示的子控制组关联的子系统，并导出到 cgroup 文件系统*/
    unsigned int child_subsys_mask; /*子控制组关联的子系统位图，不一定导出到 cgroup 文件系统*/

    struct cgroup_subsys_state __rcu *subsys[CGROUP_SUBSYS_COUNT];    /*绑定子系统状态*/
    struct cgroup_root *root;         /*指向 hierarchy 根节点*/

    struct list_head cset_links;
```

```
                /*链接 cgrp_cset_link 实例，用于获取绑定进程/线程关联的 css_set 实例，查找进程*/
    struct list_head e_csets[CGROUP_SUBSYS_COUNT];
                                        /*链接各子系统控制进程/线程关联的 css_set 实例*/
    struct list_head pidlists;          /*链接 cgroup_pidlist 实例，管理控制组绑定进程/线程 ID 值*/
    struct mutex pidlist_mutex;
    wait_queue_head_t offline_waitq;
    struct work_struct   release_agent_work;
          /*执行释放代理，添加到 cgroup_destroy_wq 工作队列，删除控制组时，将调用此工作*/
};
```

cgroup 结构体主要成员简介如下：

●**self**：表示控制组状态的 cgroup_subsys_state 结构体成员，结构体定义见下文。

●**flags**：标记，取值定义如下：

```
enum {
    CGRP_NOTIFY_ON_RELEASE,    /*控制组要求释放通知至用户空间*/
    CGRP_CPUSET_CLONE_CHILDREN,       /*复制父控制组属性至子控制组（cpuset cgroup）*/
};
```

2 cgroup_subsys

cgroup_subsys 表示控制组子系统，控制组子系统又称为“资源控制器”，用于调度、限制控制组中进程对某种资源的使用。

cgroup_subsys 结构体定义如下（/include/linux/cgroup-defs.h）：

```
struct cgroup_subsys {
    struct cgroup_subsys_state *(*css_alloc)(struct cgroup_subsys_state *parent_css);
                                    /*为控制组分配子系统状态 cgroup_subsys_state 实例，并初始化*/
    int (*css_online)(struct cgroup_subsys_state *css);
                                        /*为控制组创建 cgroup_subsys_state 实例后调用*/
    void (*css_offline)(struct cgroup_subsys_state *css);
    void (*css_released)(struct cgroup_subsys_state *css);
    void (*css_free)(struct cgroup_subsys_state *css);      /*释放 cgroup_subsys_state 及 cgroup 实例*/
    void (*css_reset)(struct cgroup_subsys_state *css);
    void (*css_e_css_changed)(struct cgroup_subsys_state *css);

    int (*can_attach)(struct cgroup_subsys_state *css,struct cgroup_taskset *tset);
                    /*是否可以移动一个或多个进程至控制组，tset 表示进程集合，返回 0 表示成功*/
    void (*cancel_attach)(struct cgroup_subsys_state *css,struct cgroup_taskset *tset);
                                                        /*取消移动进程（集）*/
    void (*attach)(struct cgroup_subsys_state *css,struct cgroup_taskset *tset);  /*移动进程（集）*/

    void (*fork)(struct task_struct *task);        /*fork()新进程时调用*/
    void (*exit)(struct cgroup_subsys_state *css,
                        struct cgroup_subsys_state *old_css,struct task_struct *task);
                                                        /*进程退出时调用*/
    void (*bind)(struct cgroup_subsys_state *root_css);    /*子系统（迁移）绑定到 hierarchy 时调用*/
```

```
    int disabled;        /*子系统是否关闭（不可使用）*/
    int early_init;      /*子系统是否在内核启动早期执行初始化，见下文*/
    bool broken_hierarchy;
    bool warned_broken_hierarchy;

    int id;        /*子系统 ID 值*/
    const char *name;      /*子系统名称*/

    struct cgroup_root *root;     /*指向绑定 hierarchy 根节点*/
    struct idr css_idr;       /*管理 cgroup_subsys_state 实例 ID 值*/

    struct list_head cfts;        /*cftype 实例数组双链表*/
                                  /*自动注册到控制组下的 cftype 实例数组（专属于子系统的数组）*/
    struct cftype *dfl_cftypes;         /*适用于默认 hierarchy 下的控制组*/
    struct cftype *legacy_cftypes;      /*适用于非默认 hierarchy 下的控制组*/
    unsigned int depends_on;        /**/
};
```

cgroup_subsys 结构体主要成员简介如下：

●**css_alloc()**：为控制组分配并初始化 cgroup_subsys_state 实例的函数指针，返回 cgroup_subsys_state 实例指针。

●**css_free()**：释放 cgroup、cgroup_subsys_state 实例的函数指针。css_alloc()和 css_free()是必须实现的函数，cgroup_subsys 结构体中的其它函数是可选实现的。

●**disabled**：子系统是否关闭，不可使用。

●**early_init**：是否需要在早期初始化中（cgroup_init_early()函数）初始化子系统。

●**id、name**：子系统 ID 值和名称。

●**cfts**：双链表成员，链接 cftype 实例数组，cftype 结构体可以认为是表示子系统的一个参数。控制组在 cgroup 文件系统中由一个目录项表示，受子系统控制的控制组，子系统关联的 cftype 实例在控制组目录项下由一个文件表示。cftype 结构体定义见下文。

●**dfl_cftypes**：指向 cftype 实例数组，数组项以文件的形式添加到默认 hierarchy 中控制组表示的目录项下。

●**legacy_cftypes**：指向 cftype 实例数组，数组项以文件的形式添加到非默认 hierarchy 中控制组表示的目录项下。

内核中控制组子系统需定义 cgroup_subsys 实例，实例名称为 xxx_cgrp_subsys。例如，内存管理子系统 cgroup_subsys 实例定义如下（/mm/memcontrol.c）：

```
struct cgroup_subsys    memory_cgrp_subsys    __read_mostly;
EXPORT_SYMBOL(memory_cgrp_subsys);
```

内核目前支持 12 个控制组子系统，见/include/linux/cgroup_subsys.h 头文件。

3 cftype

cftype 结构体表示子系统的控制参数，在 cgroup 文件系统中表现为控制组目录项下的文件，用户可读

取/设置文件值（内容）。cftype 结构体定义如下：

```
struct cftype {
    char name[MAX_CFTYPE_NAME];      /*名称*/
    int private;          /*私有数据，如文件类型*/
    umode_t mode;         /*访问模式*/
    size_t max_write_len;     /*最大写长度*/
    unsigned int flags;       /*标记*/

    struct cgroup_subsys *ss;     /*所属子系统*/
    struct list_head node;        /*将实例添加到 cgroup_subsys 实例中 cfts 双链表*/
    struct kernfs_ops *kf_ops;    /*指向 kernfs_ops 实例，在 cgroup_init_cftypes()函数中设置*/

    /*读写函数*/
    u64 (*read_u64)(struct cgroup_subsys_state *css, struct cftype *cft);
    s64 (*read_s64)(struct cgroup_subsys_state *css, struct cftype *cft);
    int (*seq_show)(struct seq_file *sf, void *v);

    /*可选操作，读写函数，顺序文件迭代器函数*/
    void *(*seq_start)(struct seq_file *sf, loff_t *ppos);
    void *(*seq_next)(struct seq_file *sf, void *v, loff_t *ppos);
    void (*seq_stop)(struct seq_file *sf, void *v);
    int (*write_u64)(struct cgroup_subsys_state *css, struct cftype *cft,u64 val);
    int (*write_s64)(struct cgroup_subsys_state *css, struct cftype *cft,s64 val);
    ssize_t (*write)(struct kernfs_open_file *of,char *buf, size_t nbytes, loff_t off);

  #ifdef CONFIG_DEBUG_LOCK_ALLOC
    struct lock_class_key     lockdep_key;
  #endif
};
```

cftypes 结构体主要成员简介如下：

●**name[]**：名称，在 cgroup 文件系统中表现为文件名称。

●**mode**：文件访问模式。

●**node**：双链表成员，将实例添加到 cgroup_subsys 实例中 cfts 双链表。通常 cftype 实例以数组的形式定义（最后一个数组项为空），只有数组首项通过 node 成员添加到 cgroup_subsys.cfts 双链表。

●**kf_ops**：指向 kernfs_ops 实例，用于文件的读写操作。此成员在 cgroup_init_cftypes()函数中赋值，kernfs_ops 实例中的函数调用 cftype 实例中定义的读写函数完成参数操作。

●**flags**：标记成员，取值定义如下（/include/linux/cgroup-defs.h）：

```
enum {
    CFTYPE_ONLY_ON_ROOT= (1 << 0),     /*只适用于 hierarchy 根节点中内嵌的控制组*/
    CFTYPE_NOT_ON_ROOT  = (1 << 1),    /*不适用于 hierarchy 根节点中内嵌的控制组*/
    CFTYPE_NO_PREFIX       = (1 << 3),     /* (DON'T USE FOR NEW FILES) no subsys prefix */

    /*内部标记*/
```

```
    __CFTYPE_ONLY_ON_DFL    = (1 << 16),    /*只适用于默认 hierarchy 下的控制组*/
    __CFTYPE_NOT_ON_DFL     = (1 << 17),    /*不适用于默认 hierarchy 下的控制组*/
};
```

●**private**：私有数据，例如用来保存文件类型，文件类型由枚举类型定义，如下所示：

```
enum cgroup_filetype {
    CGROUP_FILE_PROCS,
    CGROUP_FILE_TASKS,
};
```

cftype 实例通常以数组的形式定义，数组最末尾项为空。对于属于特定子系统 cgroup_subsys 实例的 cftype 数组，其首项添加到 cgroup_subsys 实例中 cfts 双链表。

内核定义了适用于默认 hierarchy 下控制组的 cftype 数组 cgroup_dfl_base_files，以及适用于非默认 hierarchy 下控制组的 cftype 数组 cgroup_legacy_base_files。这两个数组项中的 cftype 实例不关联到任何子系统，在创建控制组时将以文件形式自动添加到控制组目录项下。

一个控制组可关联一个或多个子系统，即受多个子系统的控制。在创建控制组时，关联子系统下的 cftypes 实例将以文件的形式添加到控制组表示的目录项下。

4 cgroup_subsys_state

cgroup_subsys_state 结构体表示控制组、子系统的状态。控制组 cgroup 结构体中 self 成员是内嵌的 cgroup_subsys_state 结构体成员，表示控制组状态，用于记录引用计数、构成控制组的父子层次关系等。

控制组 cgroup 结构体 subsys[]指针数组成员，各数组项指向表示控制组绑定子系统在本控制组状态的 cgroup_subsys_state 实例。

在创建控制组时，将为控制组绑定的每个子系统创建对应的 cgroup_subsys_state 实例。控制组绑定子系统是可以继承的，因此表示各子系统状态的 cgroup_subsys_state 实例依 cgroup 实例也组成父子层次结构。

cgroup_subsys_state 结构体定义如下（/include/linux/cgroup-defs.h）：

```
struct cgroup_subsys_state {
    struct cgroup *cgroup;      /*关联的控制组*/
    struct cgroup_subsys *ss;   /*关联的子系统*/

    struct percpu_ref   refcnt;     /*引用计数*/
    struct cgroup_subsys_state *parent;     /*父实例*/

    struct list_head sibling;       /*链接兄弟实例*/
    struct list_head children;      /*子实例双链表*/
    int id;         /*ID 值*/
    unsigned int flags;     /*标记*/
    u64 serial_nr;

    struct rcu_head rcu_head;
    struct work_struct destroy_work;
};
```

cgroup_subsys_state 结构体主要成员简介如下：

●**cgroup**：关联的控制组。

●**ss**：关联的子系统。

●**parent**：指向父 cgroup_subsys_state 实例。

●**children**：双链表头，链接子 cgroup_subsys_state 实例。

●**sibling**：双链表成员，链接兄弟 cgroup_subsys_state 实例，具有相同的父实例。

●**flags**：标记，取值定义如下：

```
enum {
    CSS_NO_REF      = (1 << 0),   /*没有引用计数值*/
    CSS_ONLINE      = (1 << 1),   /*正处于 css_online()和 css_offline() 之间*/
    CSS_RELEASED  = (1 << 2),   /*引用计数为 0，已释放*/
};
```

7.15.3 初始化

控制组初始化工作主要包括默认 hierarchy 根节点的设置和激活，各子系统 cgroup_subsys 实例的初始化，cgroup 文件系统类型的注册等。cgroup 文件系统的挂载，控制组的创建等工作由用户进程通过 cgroup 文件系统进行。

控制组初始化主要分两步：早期初始化和后期初始化。早期初始化在内核启动阶段早期调用函数 cgroup_init_early()完成，后期初始化在内核启动后期调用 cgroup_init()函数完成。

下文将先介绍默认 hierarchy 根节点及各子系统的定义，然后介绍两步初始化函数的实现。

1 默认 hierarchy 与子系统

内核定义了默认的 hierarchy，初始化时所有子系统都绑定到默认 hierarchy。默认 hierarchy 中只包含一个 cgroup，初始阶段内核中进程/线程都位于此控制组下。默认 hierarchy 根节点 cgroup_root 实例为 cgrp_dfl_root，如下所示：

```
struct cgroup_root cgrp_dfl_root;
```

用户在挂载 cgroup 文件系统时，将创建新的 hierarchy，在挂载目录项下创建子目录项，将创建新控制组，详见下一小节。

支持控制组的内核子系统（用于某项资源的控制），需要定义 cgroup_subsys 实例，实例名称为 *xxx*_cgrp_subsys。

在/include/linux/cgroup-defs.h 头文件中通过枚举类型为每个子系统设置了 ID 值：

```
#define SUBSYS(_x)    _x ## _cgrp_id,        /*ID 值标识*/
enum cgroup_subsys_id {
#include <linux/cgroup_subsys.h>        /*调用 SUBSYS(_x) 宏，定义各子系统 ID 值标识*/
    CGROUP_SUBSYS_COUNT,        /*子系统数量*/
};
#undef SUBSYS
```

在/include/linux/cgroup_subsys.h 头文件中调用了 SUBSYS(_x) 宏，设置各子系统 ID 值标识为 *xxx*_cgrp_id，内核目前最多支持 12 个控制组子系统。

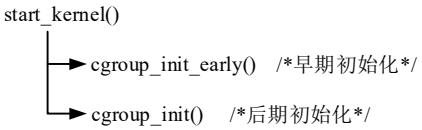
在/kernel/cgroup.c 文件内定义了 cgroup_subsys[]指针数组，数组项指向各子系统 cgroup_subsys 实例，

还定义了 cgroup_subsys_name[]字符串指针数组，表示各子系统名称，如下所示：

```
#define SUBSYS(_x)  [_x ## _cgrp_id] = &_x ## _cgrp_subsys,        /*指向 cgroup_subsys 实例*/
static struct cgroup_subsys *cgroup_subsys[] = {
    #include <linux/cgroup_subsys.h>          /*调用 SUBSYS(_x) 宏*/
};
#undef SUBSYS

#define SUBSYS(_x)  [_x ## _cgrp_id] = #_x,        /*指向名称字符串，_x ## _cgrp_id 为 ID 标识*/
static const char *cgroup_subsys_name[] = {          /*名称字符串指针数组*/
    #include <linux/cgroup_subsys.h>       /*调用 SUBSYS(_x) 宏*/
};
#undef SUBSYS
```

2 早期初始化

控制组初始化分两步，分别由 cgroup_init_early()和 cgroup_init()函数完成，函数调用关系如下：



早期初始化函数 cgroup_init_early()定义如下：

```
int __init cgroup_init_early(void)
{
    static struct cgroup_sb_opts __initdata opts;
                                        /*cgroup_sb_opts 实例，表示 cgroup 文件系统挂载参数*/
    struct cgroup_subsys *ss;
    int i;

    init_cgroup_root(&cgrp_dfl_root, &opts); /*初始化默认的 hierarchy 根节点 cgrp_dfl_root 实例*/
    cgrp_dfl_root.cgrp.self.flags |= CSS_NO_REF;        /*设置根节点控制组状态标记位*/

    RCU_INIT_POINTER(init_task.cgroups, &init_css_set);       /*设置 task_struct.cgroups 成员*/
                                          /*init_task.cgroups 指向 init_css_set 实例*/
    for_each_subsys(ss, i) {       /*遍历 cgroup_subsys[]指向的子系统 cgroup_subsys 实例*/
        ...     /*完成检查工作*/

        ss->id = i;         /*设置子系统 ID 值*/
        ss->name = cgroup_subsys_name[i];         /*设置子系统名称*/

        if (ss->early_init)          /*如果子系统需要在早期执行初始化*/
            cgroup_init_subsys(ss, true);       /*初始化子系统*/
    }
    return 0;
```

}

cgroup_init_early()函数中主要是初始化默认hierarchy的根节点**cgrp_dfl_root**实例；遍历cgroup_subsys[]数组项指向的各子系统cgroup_subsys实例，对其ID值和名称赋值，如果子系统设置了early_init成员，还需要调用cgroup_init_subsys()函数对cgroup_subsys实例进行初始化。

下面分别介绍初始化hierarchy根节点init_cgroup_root()函数，以及初始化子系统cgroup_init_subsys()函数的实现。

■初始化默认的 hierarchy 根节点

在cgroup_init_early()函数中调用init_cgroup_root()函数初始化默认hierarchy的根节点**cgrp_dfl_root**实例。在挂载cgroup文件系统创建根节点后，也将调用init_cgroup_root()函数对根节点进行初始化。

init_cgroup_root()函数定义如下：

```
static void init_cgroup_root(struct cgroup_root *root,struct cgroup_sb_opts *opts)
/*root：指向根节点，opts：指向 cgroup_sb_opts 结构体，表示挂载参数*/
{
    struct cgroup *cgrp = &root->cgrp;        /*根节点内嵌的控制组成员*/

    INIT_LIST_HEAD(&root->root_list);
    atomic_set(&root->nr_cgrps, 1);           /*控制组数量初始化为 1*/
    cgrp->root = root;
    init_cgroup_housekeeping(cgrp);           /*初始化根节点内嵌的控制组*/
    idr_init(&root->cgroup_idr);              /*初始化管理控制组 ID 值的数据结构*/

    root->flags = opts->flags;                /*标记*/
    if (opts->release_agent)
        strcpy(root->release_agent_path, opts->release_agent);
    if (opts->name)          /*名称*/
        strcpy(root->name, opts->name);
    if (opts->cpuset_clone_children)
        set_bit(CGRP_CPUSET_CLONE_CHILDREN, &root->cgrp.flags);
}
```

■初始化子系统

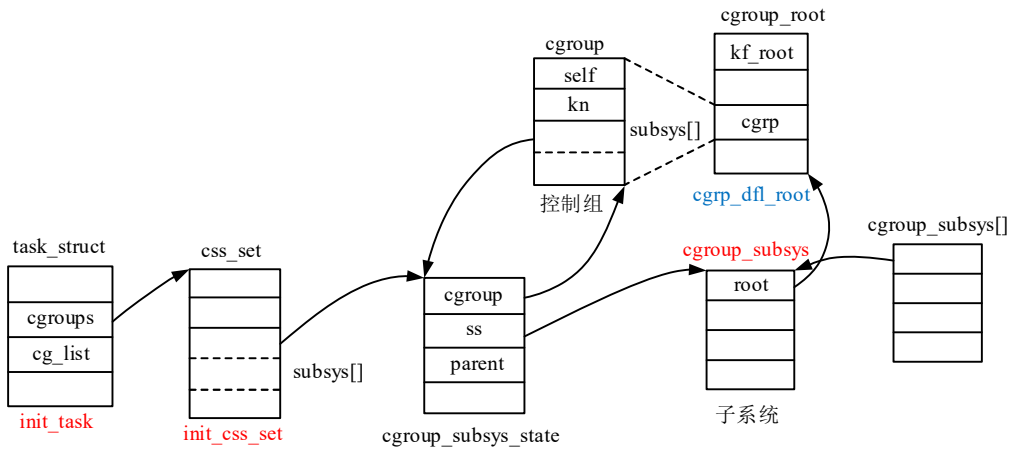
cgroup_init_subsys()函数用于初始化子系统cgroup_subsys实例，函数定义如下：

```
static void __init cgroup_init_subsys(struct cgroup_subsys *ss, bool early)
{
    struct cgroup_subsys_state *css;

    printk(KERN_INFO "Initializing cgroup subsys %s\n", ss->name);
    mutex_lock(&cgroup_mutex);

    idr_init(&ss->css_idr);       /*初始化管理子系统下 cgroup_subsys_state 实例 ID 值的数据结构*/
```

```
    INIT_LIST_HEAD(&ss->cfts);      /*初始化 cftype 实例双链表*/

    ss->root = &cgrp_dfl_root;
                        /*指向默认 hierarchy 的根节点，即子系统初始化绑定至默认 hierarchy*/
    css = ss->css_alloc(cgroup_css(&cgrp_dfl_root.cgrp, ss));
                        /*为根节点内嵌控制组分配并初始化代表子系统的 cgroup_subsys_state 实例*/
    BUG_ON(IS_ERR(css));
    init_and_link_css(css, ss, &cgrp_dfl_root.cgrp);      /*建立 css 与 ss、cgrp_dfl_root.cgrp 之间关联*/

    css->flags |= CSS_NO_REF;

    if (early) {
        css->id = 1;      /*早期初始化分配的 cgroup_subsys_state 实例 ID 值为 1*/
    } else {          /*后期初始化分配 cgroup_subsys_state 实例*/
        css->id = cgroup_idr_alloc(&ss->css_idr, css, 1, 2, GFP_KERNEL);     /*分配 ID 值*/
        BUG_ON(css->id < 0);
    }

    init_css_set.subsys[ss->id] = css;      /*指向 cgroup_subsys_state 实例*/

    have_fork_callback |= (bool)ss->fork << ss->id;      /*具有 fork()回调函数时，设置全局标记位*/
    have_exit_callback |= (bool)ss->exit << ss->id;      /*具有 exit()回调函数时，设置全局标记位*/

    BUG_ON(!list_empty(&init_task.tasks));
    BUG_ON(online_css(css));      /*调用 ss->css_online(css)，设置 cgroup->subsys[ss->id]=css*/

    mutex_unlock(&cgroup_mutex);
}
```

cgroup_init_subsys()函数将子系统绑定到默认 hierarchy，为默认 hierarchy 根节点内嵌的控制组创建表示子系统状态的 cgroup_subsys_state 实例，建立 cgroup_subsys_state 实例与控制组和子系统之间的关联，最后还将建立初始 css_set 实例 init_css_set 与新创建 cgroup_subsys_state 实例之间的关联。

总之，cgroup_init_subsys()函数初始化 cgroup_subsys 实例的结果如下图所示：



3 后期初始化

内核启动函数将在后期调用 cgroup_init()函数，完成控制组后期初始化工作，函数定义如下：

```
int __init cgroup_init(void)
{
    struct cgroup_subsys *ss;
    unsigned long key;
    int ssid, err;

    /*初始化公共 cftype 实例数组，设置 kernfs_ops 实例等*/
    BUG_ON(cgroup_init_cftypes(NULL, cgroup_dfl_base_files));
    BUG_ON(cgroup_init_cftypes(NULL, cgroup_legacy_base_files));

    mutex_lock(&cgroup_mutex);

    /*将 init_css_set 实例添加到全局散列表 css_set_table*/
    key = css_set_hash(init_css_set.subsys);            /*计算散列值*/
    hash_add(css_set_table, &init_css_set.hlist, key);      /*添加到散列链表*/

    BUG_ON(cgroup_setup_root(&cgrp_dfl_root, 0));       /*激活默认 hierarchy 根节点*/

    mutex_unlock(&cgroup_mutex);

    for_each_subsys(ss, ssid) {         /*遍历 cgroup_subsys[]指向的子系统 cgroup_subsys 实例*/
        if (ss->early_init) {
            struct cgroup_subsys_state *css =init_css_set.subsys[ss->id];
            css->id = cgroup_idr_alloc(&ss->css_idr, css, 1, 2,GFP_KERNEL);       /*分配 ID 值*/
            BUG_ON(css->id < 0);
        } else {
            cgroup_init_subsys(ss, false);          /*后期初始化子系统*/
        }

        list_add_tail(&init_css_set.e_cset_node[ssid], &cgrp_dfl_root.cgrp.e_csets[ssid]);
                            /*将 init_css_set 实例添加到默认 hierarchy 根节点控制组 e_csets[]双链表*/
        if (ss->disabled)       /*如果子系统关闭，遍历下一个子系统*/
            continue;

        cgrp_dfl_root.subsys_mask |= 1 << ss->id;       /*绑定子系统至默认 hierarchy*/

        if (cgroup_legacy_files_on_dfl && !ss->dfl_cftypes)
                        /*cgroup_legacy_files_on_dfl 默认为 0，
                        *可通过"cgroup__DEVEL__legacy_files_on_dfl"命令行参数修改。*/
            ss->dfl_cftypes = ss->legacy_cftypes;
```

```
            if (!ss->dfl_cftypes)
                cgrp_dfl_root_inhibit_ss_mask |= 1 << ss->id;

            /*初始化并添加 cftype 实例数组至子系统 cfts 双链表，
            *在绑定 hierarchy 下各控制组下创建文件。
            */
            if (ss->dfl_cftypes == ss->legacy_cftypes) {
                WARN_ON(cgroup_add_cftypes(ss, ss->dfl_cftypes));
            } else {
                WARN_ON(cgroup_add_dfl_cftypes(ss, ss->dfl_cftypes));
                WARN_ON(cgroup_add_legacy_cftypes(ss, ss->legacy_cftypes));
            }

            if (ss->bind)
                ss->bind(init_css_set.subsys[ssid]);        /*子系统绑定至默认 hierarchy*/
        }       /*遍历子系统结束*/

        err = sysfs_create_mount_point(fs_kobj, "cgroup");      /*创建/sys/fs/cgroup 目录（空目录项）*/
        ...
        err = register_filesystem(&cgroup_fs_type);         /*注册文件系统类型 cgroup_fs_type 实例*/
        ...
        proc_create("cgroups", 0, NULL, &proc_cgroupstats_operations);
                                    /*创建/proc/cgroups 文件，输出控制组的整体信息*/
        return 0;
    }
```

cgroup_init()函数主要完成以下工作：

（1）初始化 cftype 实例数组 cgroup_dfl_base_files 和 cgroup_legacy_base_files（调用 cgroup_init_cftypes()函数）。

（2）调用 cgroup_setup_root()函数激活默认 hierarchy 根节点 cgrp_dfl_root 实例。

（3）遍历各子系统，对子系统进行初始化，绑定至默认 hierarchy，并对各子系统关联的 cftype 实例（数组）在默认 hierarchy 下各控制组目录项下创建对应的文件。

（4）创建/sys/fs/cgroup 目录，注册 cgroup 文件系统类型 cgroup_fs_type 实例，创建/proc/cgroups 文件，用于输出 cgroup 整体信息，如内核中一共有几个 hierarchy，各 hierarchy 绑定了哪些子系统等。

通常用户进程挂载 tmpfs 文件系统至/sys/fs/cgroup/目录，并在其下创建表示各 hierarchy 的目录项，然后挂载 cgroup 文件系统至/sys/fs/cgroup/下目录项（创建控制组）。

■处理 cftype 实例数组

cgroup_init()函数中需要初始化内核定义的两个 cftype 实例数组 cgroup_dfl_base_files 和 cgroup_legacy_base_files。前者自动关联到默认 hierarchy 下的所有控制组，在控制组目录项下创建对应的文件。后者自动关联到非默认 hierarchy 下的所有控制组，在控制组目录项下创建文件。这两个数组是公共的，不特定于任何一个子系统。

cgroup_dfl_base_files 和 cgroup_legacy_base_files 数组简列如下：

```
static struct cftype cgroup_dfl_base_files[] = {        /*应用于默认 hierarchy 下控制组*/
```

```
    {
        .name = "cgroup.procs",          /*文件名称*/
        .seq_start = cgroup_pidlist_start,
        .seq_next = cgroup_pidlist_next,
        .seq_stop = cgroup_pidlist_stop,
        .seq_show = cgroup_pidlist_show,
        .private = CGROUP_FILE_PROCS,
        .write = cgroup_procs_write,
                    /*将线程组 TGID 写入文件，表示将线程组中全部线程移入控制组*/
        .mode = S_IRUGO | S_IWUSR,
    },
    ...
    { }  /*结束数组项必须为空*/
};

static struct cftype cgroup_legacy_base_files[] = {     /*应用于非默认 hierarchy 下控制组*/
    {
        .name = "cgroup.procs",     /*将线程组 TGID 写入此文件，将线程组中全部线程移入控制组*/
        .seq_start = cgroup_pidlist_start,
        .seq_next = cgroup_pidlist_next,
        .seq_stop = cgroup_pidlist_stop,
        .seq_show = cgroup_pidlist_show,
        .private = CGROUP_FILE_PROCS,
        .write = cgroup_procs_write,
        .mode = S_IRUGO | S_IWUSR,
    },
    ...
    {
        .name = "tasks",          /*写进程 ID 至此文件，将进程移入控制组*/
        .seq_start = cgroup_pidlist_start,
        .seq_next = cgroup_pidlist_next,
        .seq_stop = cgroup_pidlist_stop,
        .seq_show = cgroup_pidlist_show,
        .private = CGROUP_FILE_TASKS,
        .write = cgroup_tasks_write,        /*添加进程至控制组*/
        .mode = S_IRUGO | S_IWUSR,
    },
    ...
    { }  /*最后一个数组项必须为空*/
};
```

cgroup_init_cftypes()函数用于初始化 cftype 实例数组，函数定义如下：

```
static int cgroup_init_cftypes(struct cgroup_subsys *ss, struct cftype *cfts)
```

```
/*ss：关联的子系统，此处为 NULL（公共 cftype 实例），cfts：指向 cftype 实例数组*/
{
    struct cftype *cft;

    for (cft = cfts; cft->name[0] != '\0'; cft++) {        /*遍历 cftype 数组项*/
        struct kernfs_ops *kf_ops;

        WARN_ON(cft->ss || cft->kf_ops);

        if (cft->seq_start)       /*对 cftype 实例关联的 kernfs_ops 实例赋值*/
            kf_ops = &cgroup_kf_ops;        /*顺序文件，迭代读*/
        else
            kf_ops = &cgroup_kf_single_ops;     /*顺序文件，单次读*/

        if (cft->max_write_len && cft->max_write_len != PAGE_SIZE) {
            kf_ops = kmemdup(kf_ops, sizeof(*kf_ops), GFP_KERNEL);
            ...
            kf_ops->atomic_write_len = cft->max_write_len;
        }

        cft->kf_ops = kf_ops;       /*kernfs_ops 实例指针赋予 cftype 实例*/
        cft->ss = ss;        /*设置关联的子系统*/
    }
    return 0;
}
```

在 cgroup_init()函数中，在激活默认 hierarchy 根节点后，需要遍历各子系统，对子系统初始化，绑定至默认 hierarchy。其中还需要做的一项工作是将 ss->dfl_cftypes/ss->legacy_cftypes 指向的 cftype 实例数组添加到子系统，并在子系统绑定的 hierarchy 下的所有控制组目录项下创建对应的文件。

以上工作由 cgroup_add_cftypes()函数完成，函数定义如下：

```
static int cgroup_add_cftypes(struct cgroup_subsys *ss, struct cftype *cfts)
/*ss：子系统，cfts：指向 cftype 实例数组*/
{
    int ret;

    if (ss->disabled)
        return 0;

    if (!cfts || cfts[0].name[0] == '\0')
        return 0;

    ret = cgroup_init_cftypes(ss, cfts);        /*初始化 cftype 实例数组*/
    if (ret)
```

```
        return ret;

    mutex_lock(&cgroup_mutex);

    list_add_tail(&cfts->node, &ss->cfts);     /*将首个数组项实例添加到 ss->cfts 双链表*/
    ret = cgroup_apply_cftypes(cfts, true);
                                    /*在子系统绑定的 hierarchy 下的所有控制组中创建文件*/
    if (ret)
        cgroup_rm_cftypes_locked(cfts);

    mutex_unlock(&cgroup_mutex);
    return ret;
}
```

■激活 hierarchy 根节点

后期初始化函数 cgroup_init()，有一项重要的工作是激活默认 hierarchy 的根节点 cgroup_root 实例，这项工作由 cgroup_setup_root()函数完成，函数定义如下。

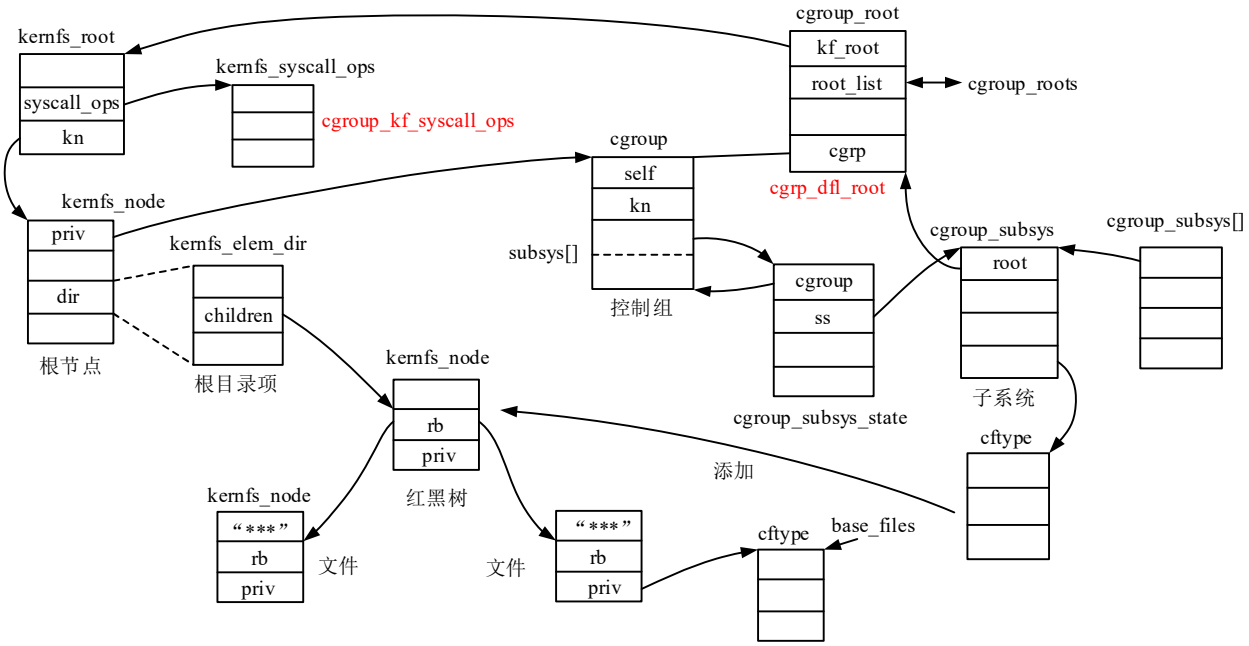
```
static int cgroup_setup_root(struct cgroup_root *root, unsigned long ss_mask)
/*ss_mask：绑定子系统的位掩码，此处为 0*/
{
    LIST_HEAD(tmp_links);
    struct cgroup *root_cgrp = &root->cgrp;        /*内嵌控制组*/
    struct cftype *base_files;
    struct css_set *cset;
    int i, ret;

    lockdep_assert_held(&cgroup_mutex);
    ret = cgroup_idr_alloc(&root->cgroup_idr, root_cgrp, 1, 2, GFP_NOWAIT);
                                                        /*为嵌入 cgroup 分配 ID 值*/
    ...
    root_cgrp->id = ret;          /*设置 ID 值*/
    ret = percpu_ref_init(&root_cgrp->self.refcnt, css_release, 0,GFP_KERNEL);
    ...
    ret = allocate_cgrp_cset_links(css_set_count, &tmp_links);
                    /*css_set_count 为当前 css_set 实例数量*/
                   /*分配 css_set_count 个 cgrp_cset_link 结构体实例，添加到 tmp_links 双链表*/
    ...
    ret = cgroup_init_root_id(root);     /*分配并设置 root->hierarchy_id 值*/
    ...
    root->kf_root = kernfs_create_root(&cgroup_kf_syscall_ops,
                        KERNFS_ROOT_CREATE_DEACTIVATED,root_cgrp);
                            /*创建 kernfs 文件系统根节点，私有为 root_cgrp（内嵌控制组指针）*/
                            /*kernfs_syscall_ops 结构体实例为 cgroup_kf_syscall_ops*/
```

```
    ...
    root_cgrp->kn = root->kf_root->kn;

    /*设置公共 cftype 实例数组*/
    if (root == &cgrp_dfl_root)
        base_files = cgroup_dfl_base_files;        /*默认 hierarchy 根节点*/
    else
        base_files = cgroup_legacy_base_files;     /*其它 hierarchy 根节点*/

    ret = cgroup_addrm_files(root_cgrp, base_files, true);
                                    /*在根节点内嵌 cgroup 目录项下添加（删除）文件*/
    ...
    ret = rebind_subsystems(root, ss_mask);      /*移动子系统绑定的 hierarchy，此处 ss_mask 为 0*/
    ...
    list_add(&root->root_list, &cgroup_roots);      /*cgroup_root 实例添加到全局双链表*/
    cgroup_root_count++;      /*数量值加 1*/

    down_write(&css_set_rwsem);
    hash_for_each(css_set_table, i, cset, hlist)              /*遍历全局散列表中 css_set 实例*/
        link_css_set(&tmp_links, cset, root_cgrp);
          /*对现有的每个 css_set 实例通过 cgrp_cset_link 建立其与当前根节点内嵌控制组的关联，
          *详见下一小节。*/
    up_write(&css_set_rwsem);

    BUG_ON(!list_empty(&root_cgrp->self.children));
    BUG_ON(atomic_read(&root->nr_cgrps) != 1);

    kernfs_activate(root_cgrp->kn);
    ret = 0;
    goto out;
    ...
out:
    free_cgrp_cset_links(&tmp_links);
    return ret;
}
```

cgroup_setup_root()函数主要工作是为 hierarchy 根节点创建对应的 kernfs_root 实例（kernfs 文件系统根节点），其关联的 kernfs_syscall_ops 实例为 cgroup_kf_syscall_ops，向 hierarchy 根节点内嵌控制组目录项下添加公共（默认的）文件（cftype 实例），移动 ss_mask 标记的子系统绑定至本 hierarchy，添加根节点 cgroup_root 实例至全局双链表 cgroup_roots 等。

cgroup_init()函数在调用 cgroup_setup_root()函数时，ss_mask 参数为 0，即不绑定任何子系统。但在随后（及 cgroup_init_early()函数中）将调用 cgroup_init_subsys()函数将各子系统绑定至默认 hierarchy。

cgroup 初始化的最终结果如下图所示（图中只画出一个子系统）：



4 初始化工作队列

控制组初始化中还有一项工作就是创建工作队列，如下所示：

```
static int __init cgroup_wq_init(void)
{
    cgroup_destroy_wq = alloc_workqueue("cgroup_destroy", 0, 1);
    BUG_ON(!cgroup_destroy_wq);

    cgroup_pidlist_destroy_wq = alloc_workqueue("cgroup_pidlist_destroy",0, 1);
    BUG_ON(!cgroup_pidlist_destroy_wq);

    return 0;
}
core_initcall(cgroup_wq_init);
```

cgroup_destroy_wq 工作队列在删除控制组时，调用执行队列中工作。cgroup_pidlist_destroy_wq 工作队列用于释放 cgroup 实例 pidlists 双链表中 cgroup_pidlist 实例。

7.15.4 cgroup 文件系统

前面介绍了控制组的组织结构，默认的 hierarchy 及子系统的初始化。控制组的管理和操作由用户进程通过 cgroup 文件系统进行。cgroup 文件系统由控制组 cgroup 实例组成，将控制组信息导出到用户空间。hierarchy、控制组的创建，进程移入/移出控制组以及控制组（子系统）参数的设置都由用户进程通过 cgroup 文件系统实现。

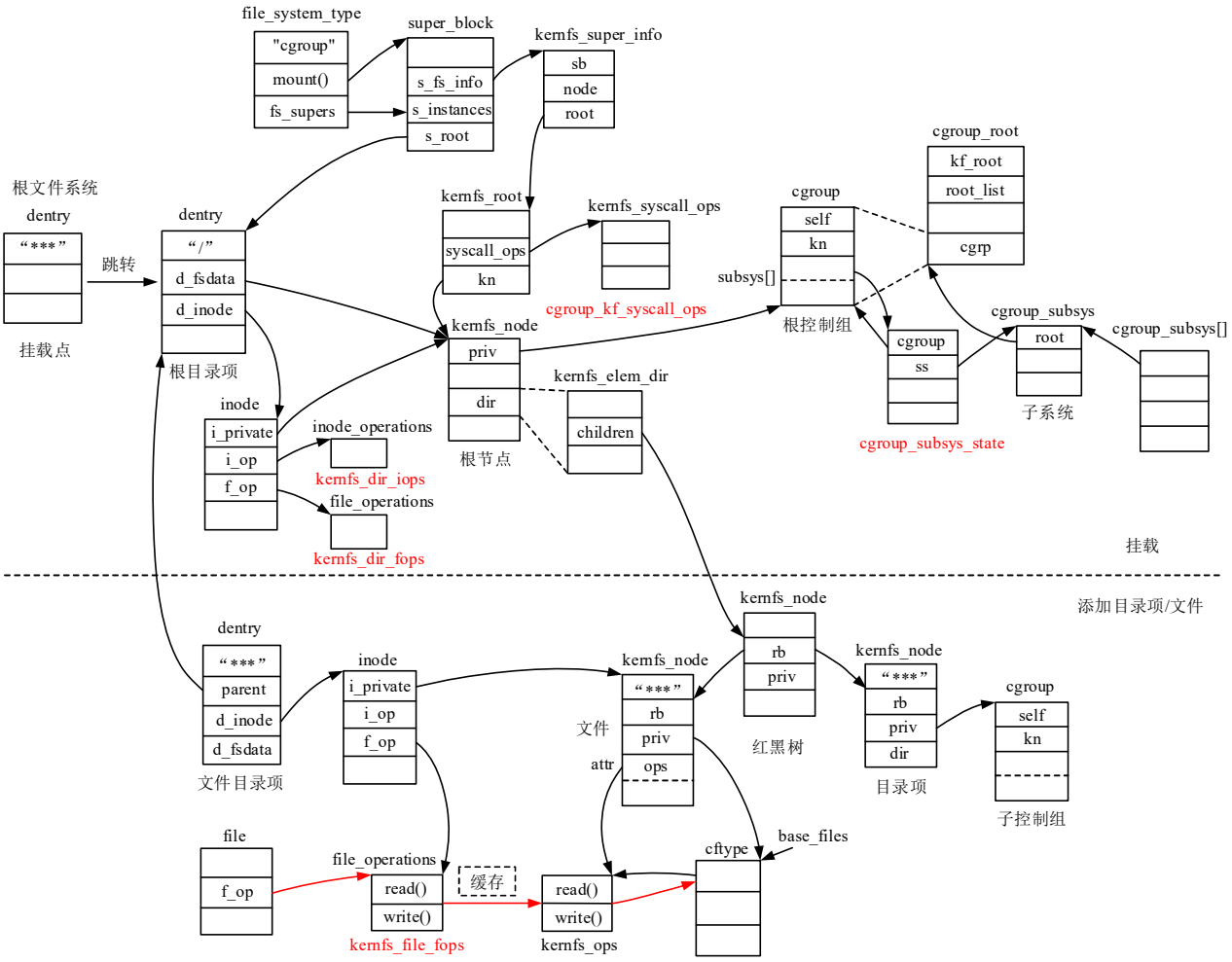
cgroup 文件系统是 kernfs 文件系统的一个实例，如同 sysfs 文件系统。每个 cgroup 文件系统实例代表一个 hierarchy，用户通过在根文件系统某个目录项下挂载 cgroup 文件系统创建 hierarchy。创建 hierarchy 将同时创建一个根控制组，文件系统根目录项关联到此根控制组。

在控制组下可创建子控制组，创建子控制组就是在挂载 cgroup 文件系统的目录项下创建子目录项。创

建子目录项时，会自动创建由 cftype 实例表示的文件。

用户进程将进程（线程组）PID 写入控制组下指定文件则表示将进程（线程组）移入控制组，用户进程还可以通过文件读/写获取/设置控制组（子系统）参数。

cgroup 文件系统结构框图如下所示：



1 挂载 cgroup 文件系统

cgroup 文件系统类型定义如下（/kernel/cgroup.c）：

```
static struct file_system_type cgroup_fs_type = {
    .name = "cgroup",
    .mount = cgroup_mount,        /*挂载函数*/
    .kill_sb = cgroup_kill_sb,
};
```

cgroup 文件系统类型挂载函数为 cgroup_mount()。在介绍 cgroup_mount()函数前，先介绍一下结构体 cgroup_sb_opts 的定义，它用于向文件系统类型挂载函数传递用户 mount()系统调用传递的挂载参数。

cgroup_sb_opts 结构体定义如下（/kernel/cgroup.c）：

```
struct cgroup_sb_opts {
    unsigned long subsys_mask;        /*绑定子系统掩码*/
    unsigned int flags;               /*标记*/
    char *release_agent;
```

```
    bool cpuset_clone_children;
    char *name;         /*创建 hierarchy 的名称，cgroup_root.name[]*/
    bool none;          /*是否不绑定任何子系统*/
};
```

用户进程在 mount()系统调用中将通过字符串的形式，向内核传递 cgroup 文件系统的挂载参数，包括挂载 cgroup 文件系统创建的 hierarchy 绑定哪些子系统等信息。

下面将先介绍 cgroup_mount()函数中如何将用户传递的参数字符串转换成 cgroup_sb_opts 实例，然后介绍 cgroup_mount()函数的实现。

■解析挂载参数

用户进程通过 mount()系统调用挂载 cgroup 文件系统时，通过字符串传递挂载参数。字符串中由“,”隔开各挂载参数，参数表示绑定子系统的名称等。例如：

```
mount -t cgroup cpuset -ocpuset /sys/fs/cgroup/cpuset
```

以上命令中-ocpuset 表示挂载参数，参数值为 cpuset，表示挂载 cgroup 文件系统至/sys/fs/cgroup/cpuset 目录，创建的 hierarchy 绑定 cpuset 子系统。

在 cgroup_mount()函数中将调用 parse_cgroupfs_options()函数解析用户传递的挂载参数，函数定义如下：

```
static int parse_cgroupfs_options(char *data, struct cgroup_sb_opts *opts)
/*data：挂载参数字符串指针，opts：cgroup_sb_opts 实例指针，保存参数信息*/
{
    char *token, *o = data;
    bool all_ss = false, one_ss = false;
    unsigned long mask = -1UL;
    struct cgroup_subsys *ss;
    int nr_opts = 0;        /*以","分隔的参数数量*/
    int i;

#ifdef CONFIG_CPUSETS
    mask = ~(1U << cpuset_cgrp_id);
#endif

    memset(opts, 0, sizeof(*opts));
    /*参数以","隔开，依次取出各参数进行处理，用 NULL 替换","字符*/
    while ((token = strsep(&o, ",")) != NULL) {        /*token 指向当前处理的参数，o 指向下一个参数*/
        nr_opts++;      /*参数数量加 1*/

        if (!*token)
            return -EINVAL;
        if (!strcmp(token, "none")) {       /*参数为"none"，表示不绑定任何子系统*/
            opts->none = true;
            continue;
        }
```

```
if (!strcmp(token, "all")) {        /*参数为"all"，表示绑定所有子系统*/
        if (one_ss)
                return -EINVAL;
        all_ss = true;
        continue;
}
if (!strcmp(token, "__DEVEL__sane_behavior")) {
                                /*统一的 hierarchy，正在开发的下一版本特性*/
                               /*“__DEVEL__sane_behavior”参数不能与其它参数同时存在*/
        opts->flags |= CGRP_ROOT_SANE_BEHAVIOR;          /*挂载默认的 hierarchy*/
        continue;
}
if (!strcmp(token, "noprefix")) {
        opts->flags |= CGRP_ROOT_NOPREFIX;
        continue;
}
if (!strcmp(token, "clone_children")) {
        opts->cpuset_clone_children = true;
        continue;
}
if (!strcmp(token, "xattr")) {       /*扩展属性*/
        opts->flags |= CGRP_ROOT_XATTR;
        continue;
}
if (!strncmp(token, "release_agent=", 14)) {/*释放代理，用户程序路径，在删除控制组时调用*/
        if (opts->release_agent)
                return -EINVAL;
        opts->release_agent =kstrndup(token + 14, PATH_MAX - 1, GFP_KERNEL);
        if (!opts->release_agent)
                return -ENOMEM;
        continue;
}
if (!strncmp(token, "name=", 5)) {       /*“name=***”参数，设置 hierarchy 名称*/
        const char *name = token + 5;
        if (!strlen(name))
                return -EINVAL;
        for (i = 0; i < strlen(name); i++) {       /*名称字符串合法性检查*/
                char c = name[i];
                if (isalnum(c))
                        continue;
                if ((c == '.') || (c == '-') || (c == '_'))
                        continue;
                return -EINVAL;
```

```
            }

            if (opts->name)       /*不能有 2 个“name=***”参数*/
                return -EINVAL;
            opts->name = kstrndup(name,MAX_CGROUP_ROOT_NAMELEN - 1,GFP_KERNEL);
            if (!opts->name)
                return -ENOMEM;
            continue;
        }      /*处理“name=***”参数结束*/

        for_each_subsys(ss, i) {       /*遍历子系统，其它参数表示绑定子系统的名称*/
            if (strcmp(token, ss->name))        /*参数名称与子系统名称不同，跳过*/
                continue;
            if (ss->disabled)       /*参数名称与子系统名称相同，但子系统关闭，跳过*/
                continue;

            /*参数名称与子系统名称相同，且子系统使能*/
            /*具有“all”参数，返回错误码，“all”参数不能与其它子系统名称参数同时出现*/
            if (all_ss)
                return -EINVAL;
            opts->subsys_mask |= (1 << i);        /*设置子系统位掩码*/
            one_ss = true;
            break;
        }      /*遍历子系统结束*/
        if (i == CGROUP_SUBSYS_COUNT)
            return -ENOENT;
    }     /*while 循环遍历参数结束*/

    if (opts->flags & CGRP_ROOT_SANE_BEHAVIOR) {
        ...
        if (nr_opts != 1) {         /*不能有其它参数*/
            ...   /*返回错误码*/
        }
        return 0;
    }

    if (all_ss || (!one_ss && !opts->none && !opts->name))
        for_each_subsys(ss, i)
            if (!ss->disabled)
                opts->subsys_mask |= (1 << i);       /*设置绑定子系统位掩码*/

    if (!opts->subsys_mask && !opts->name)
        return -EINVAL;
```

```
    if ((opts->flags & CGRP_ROOT_NOPREFIX) && (opts->subsys_mask & mask))
        return -EINVAL;

    if (opts->subsys_mask && opts->none)
        return -EINVAL;

    return 0;
}
```

parse_cgroupfs_options()函数的作用就是将由字符串表示的挂载参数转换成 cgroup_sb_opts 实例，以便于在挂载函数中使用。

■挂载函数

cgroup 文件系统类型挂载函数 cgroup_mount()定义如下：

```
static struct dentry *cgroup_mount(struct file_system_type *fs_type,
                                   int flags, const char *unused_dev_name,void *data)
/*data：指向挂载参数字符串*/
{
    struct super_block *pinned_sb = NULL;
    struct cgroup_subsys *ss;
    struct cgroup_root *root;
    struct cgroup_sb_opts opts;      /*用于保存挂载参数*/
    struct dentry *dentry;
    int ret;
    int i;
    bool new_sb;      /*是否创建新的超级块实例*/

    if (!use_task_css_set_links)   /*初始值为 false，执行 cgroup_enable_task_cg_lists()函数后设为 true*/
        cgroup_enable_task_cg_lists();        /*只在系统第一次挂载操作中调用*/
                                    /*将内核当前所有进程/线程添加到 init_css_set.tasks 双链表*/

    mutex_lock(&cgroup_mutex);

    ret = parse_cgroupfs_options(data, &opts);       /*解析挂载参数，见上文*/
    ...
    /*查找已经存在的根节点*/
    if (opts.flags & CGRP_ROOT_SANE_BEHAVIOR) {
        cgrp_dfl_root_visible = true;
        root = &cgrp_dfl_root;        /*要挂载默认 hierarchy？ */
        cgroup_get(&root->cgrp);
        ret = 0;
        goto out_unlock;
    }
```

```
for_each_subsys(ss, i) {          /*遍历各子系统*/
    if (!(opts.subsys_mask & (1 << i)) ||ss->root == &cgrp_dfl_root)
        continue;

    if (!percpu_ref_tryget_live(&ss->root->cgrp.self.refcnt)) {     /*增加绑定子系统的引用计数*/
        mutex_unlock(&cgroup_mutex);
        msleep(10);
        ret = restart_syscall();
        goto out_free;
    }
    cgroup_put(&ss->root->cgrp);
}       /*遍历各子系统结束*/

for_each_root(root) {       /*遍历已有 cgroup_root 实例，检查是否已有匹配的实例*/
    bool name_match = false;

    if (root == &cgrp_dfl_root)         /*跳过默认 cgroup_root 实例*/
        continue;
    if (opts.name) {         /*名称匹配*/
        if (strcmp(opts.name, root->name))
            continue;
        name_match = true;       /*查找到名称相同的 cgroup_root 实例*/
    }

    /*查找到名称相同的 cgroup_root 实例，或者 opts 中没有指定名称*/
    /*绑定子系统必须相同，才表示是与 opts 匹配的 cgroup_root 实例*/
    if ((opts.subsys_mask || opts.none) &&(opts.subsys_mask != root->subsys_mask)) {
        if (!name_match)
            continue;
        ret = -EBUSY;
        goto out_unlock;
    }
    /*找到了与 opts 匹配的现有 cgroup_root 实例*/
    if (root->flags ^ opts.flags)         /*标记必须相同*/
        pr_warn("new mount options do not match the existing superblock, will be ignored\n");

    pinned_sb = kernfs_pin_sb(root->kf_root, NULL);         /*锁定根节点关联的超级块实例*/
    ...       /*错误处理*/

    ret = 0;
    goto out_unlock;
}    /*遍历已有 cgroup_root 实例结束*/
```

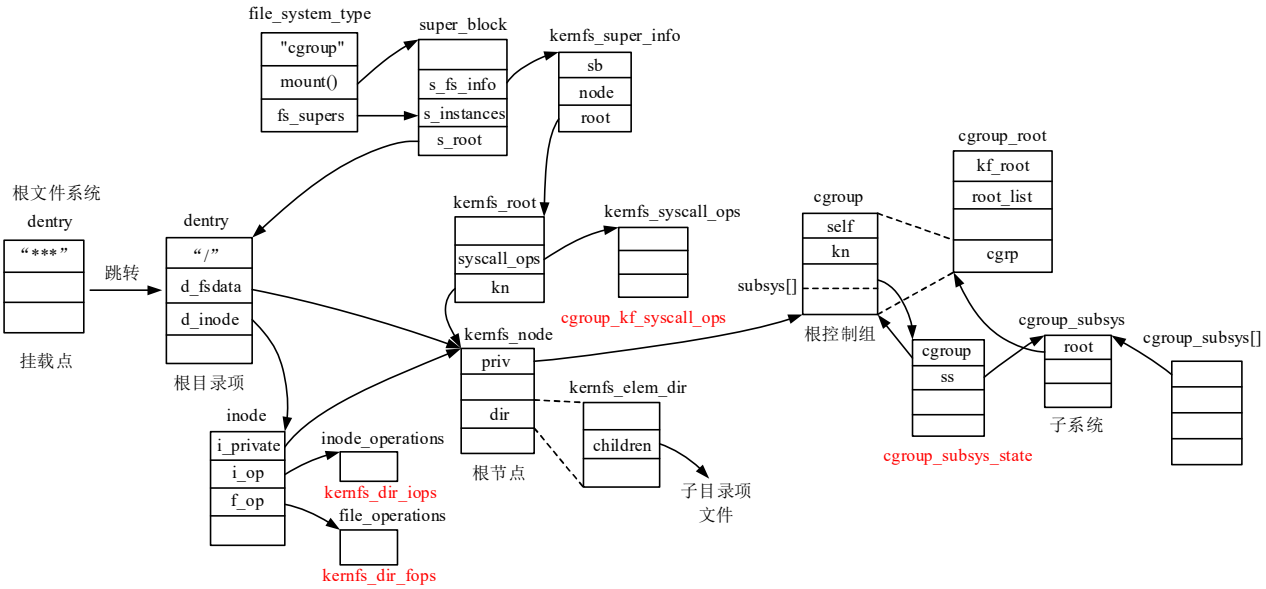
```
        /*没有找到匹配的 cgroup_root 实例，继续往下执行*/
        if (!opts.subsys_mask && !opts.none) {      /*必须传递 none 参数或子系统名称参数*/
            ...     /*错误处理*/
        }
        root = kzalloc(sizeof(*root), GFP_KERNEL);      /*分配 cgroup_root 实例*/
        ...
        init_cgroup_root(root, &opts);          /*初始化根节点 cgroup_root 实例*/

        ret = cgroup_setup_root(root, opts.subsys_mask);        /*激活 hierarchy 根节点，见上文*/
        ...     /*错误处理*/

    out_unlock:
        mutex_unlock(&cgroup_mutex);
    out_free:
        kfree(opts.release_agent);
        kfree(opts.name);
        ...
        dentry = kernfs_mount(fs_type, flags, root->kf_root,CGROUP_SUPER_MAGIC, &new_sb);
                                                        /*kernfs 文件系统挂载函数*/
        ...
        return dentry;      /*返回 cgroup 文件系统根目录项*/
    }
```

cgroup_mount()函数首先调用 parse_cgroupfs_options(data, &opts)函数解析挂载参数；然后检查内核是否已存在与挂载参数匹配的 cgroup_root 实例，如果已经存在则不需要创建 cgroup_root 实例，直接进行挂载操作，如果不存在匹配的 cgroup_root 实例，则先创建、初始化、激活 cgroup_root 实例，然后再执行挂载操作。挂载操作同 kernfs 文件系统的挂载操作。

需要注意的是，在初始化阶段所有的子系统都绑定到默认的 hierarchy。在 cgroup_mount()函数中需要将挂载参数指示绑定的子系统迁移至本 hierarchy，不再绑定到默认的 hierarchy。

迁移操作包括 cgroup_subsys_state 实例的迁移，删除子系统在原绑定 hierarchy 下控制组目录项下的文件，在新绑定 hierarchy 下控制组目录项下创建文件等（详见 cgroup_setup_root()函数）。

cgroup_mount()函数执行结果如下图所示：



在上图中，kernfs 文件系统根节点 kernfs_root 实例关联的 kernfs_syscall_ops 结构体实例为 cgroup_kf_syscall_ops（见 cgroup_setup_root()函数），此结构体中函数将用于创建/删除控制组 cgroup 实例（创建/删除目录项）等操作。挂载操作中将创建一个控制组（内嵌在 cgroup_root 实例，文件系统根目录项关联此控制组），控制组默认 cftype 实例数组以及绑定子系统关联的 cftype 实例数组，将在此目录项下创建相应的文件（图中未画出）。

由 kernfs 文件系统可知，目录项对应 inode 实例关联的节点操作结构实例为 kernfs_dir_iops，此实例中 mkdir()/rmdir()函数将用于创建/删除控制组（详见下文）。文件对应 inode 实例关联的文件操作结构实例为 kernfs_file_fops，此实例中的读写函数将调用 cftype 实例关联 kernfs_ops 实例中的函数，并最终调用 cftype 实例中定义的函数完成文件读写操作，后面将介绍 cgroup 文件系统中文件的读写操作。

2 创建控制组

挂载操作中创建了一个根控制组，用户可通过 mkdir()系统调用（mkdir 命令）在此控制组下创建子控制组。在 mkdir()系统调用中将调用普通目录项对应 inode 实例关联的节点操作结构实例中的 mkdir()函数创建子目录项。kernfs 文件系统中普通目录项关联的节点操作结构实例为 kernfs_dir_iops。

kernfs_dir_iops 实例简列如下（/fs/kernfs/dir.c）：

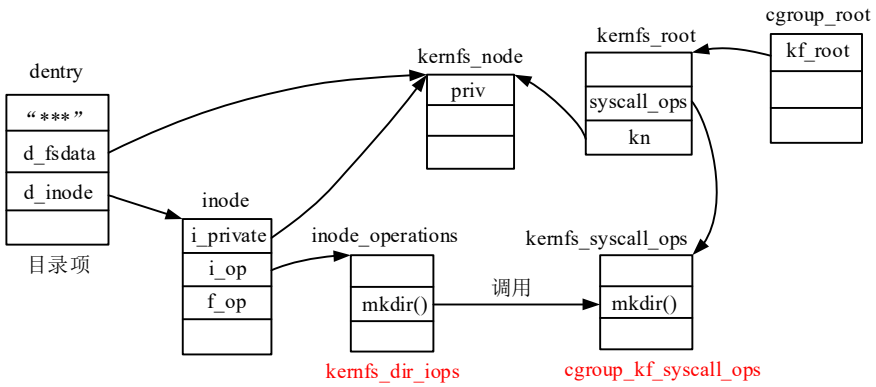
```
const struct inode_operations kernfs_dir_iops = {
    ...
    .mkdir      = kernfs_iop_mkdir,     /*创建子目录项*/
    .rmdir      = kernfs_iop_rmdir,     /*删除子目录项*/
    .rename     = kernfs_iop_rename,    /*重命名*/
};
```

kernfs_dir_iops 实例中创建子目录项的函数为 kernfs_iop_mkdir()，定义如下：

```
static int kernfs_iop_mkdir(struct inode *dir, struct dentry *dentry,umode_t mode)
{
    struct kernfs_node *parent = dir->i_private;
    struct kernfs_syscall_ops *scops = kernfs_root(parent)->syscall_ops;
                                        /*指向根节点关联的 kernfs_syscall_ops 实例*/
    int ret;
```

```
    if (!scops || !scops->mkdir)
        return -EPERM;

    if (!kernfs_get_active(parent))
        return -ENODEV;

    ret = scops->mkdir(parent, dentry->d_name.name, mode);
                                        /*调用 kernfs_syscall_ops 实例中 mkdir()函数*/
    kernfs_put_active(parent);
    return ret;
}
```

kernfs_iop_mkdir()函数将调用 kernfs 文件系统根节点关联 kernfs_syscall_ops 实例中 mkdir()函数，创建子目录项，如下图所示：



在 cgroup_setup_root()函数中将为 cgroup 文件系统 cgroup_root 根节点创建 kernfs 文件系统根节点 kernfs_root 实例，并设置其关联的 kernfs_syscall_ops 实例为 cgroup_kf_syscall_ops，定义如下：

```
static struct kernfs_syscall_ops cgroup_kf_syscall_ops = {
    .remount_fs         = cgroup_remount,
    .show_options       = cgroup_show_options,
    .mkdir              = cgroup_mkdir,         /*创建目录项，创建控制组*/
    .rmdir              = cgroup_rmdir,         /*删除目录项，删除控制组*/
    .rename             = cgroup_rename,
};
```

cgroup_mkdir()函数用于在 cgroup 文件系统中创建目录项，也就是创建子控制组，函数定义如下：

```
static int cgroup_mkdir(struct kernfs_node *parent_kn, const char *name,umode_t mode)
/*parent_kn：指向父目录项 kernfs_node 实例，name：名称，mode：访问模式*/
{
    struct cgroup *parent, *cgrp;
    struct cgroup_root *root;
    struct cgroup_subsys *ss;
    struct kernfs_node *kn;
    struct cftype *base_files;
    int ssid, ret;
```

```
if (strchr(name, '\n'))
        return -EINVAL;

parent = cgroup_kn_lock_live(parent_kn);        /*父目录项对应的控制组*/
...
root = parent->root;        /*cgroup 文件系统根节点*/

cgrp = kzalloc(sizeof(*cgrp), GFP_KERNEL);        /*分配控制组 cgroup 实例*/
...

ret = percpu_ref_init(&cgrp->self.refcnt, css_release, 0, GFP_KERNEL);        /*初始化引用计数*/
...
cgrp->id = cgroup_idr_alloc(&root->cgroup_idr, NULL, 2, 0, GFP_NOWAIT);     /*分配 ID 值*/
...
init_cgroup_housekeeping(cgrp);        /*初始化控制组*/

cgrp->self.parent = &parent->self;        /*设置父控制组*/
cgrp->root = root;        /*设置 cgroup 文件系统中根节点*/

if (notify_on_release(parent))
        set_bit(CGRP_NOTIFY_ON_RELEASE, &cgrp->flags);

if (test_bit(CGRP_CPUSET_CLONE_CHILDREN, &parent->flags))
        set_bit(CGRP_CPUSET_CLONE_CHILDREN, &cgrp->flags);

/*创建目录项 kernfs_node 实例，kernfs_node.priv 指向控制组 cgroup 实例*/
kn = kernfs_create_dir(parent->kn, name, mode, cgrp);
...
cgrp->kn = kn;

kernfs_get(kn);

cgrp->self.serial_nr = css_serial_nr_next++;

/*cgroup 实例添加到父子层次结构*/
list_add_tail_rcu(&cgrp->self.sibling, &cgroup_parent(cgrp)->self.children);
atomic_inc(&root->nr_cgrps);        /*增加控制组数量计数*/
cgroup_get(parent);

cgroup_idr_replace(&root->cgroup_idr, cgrp, cgrp->id);

ret = cgroup_kn_set_ugid(kn);        /*设置目录项的 uid、gid*/
...
```

```
    if (cgroup_on_dfl(cgrp))          /*设置公共 cftype 实例数组*/
        base_files = cgroup_dfl_base_files;
    else
        base_files = cgroup_legacy_base_files;

    ret = cgroup_addrm_files(cgrp, base_files, true);
                        /*在控制组目录项下添加公共 cftype 实例数组表示的文件*/
    ...
    /*绑定子系统*/
    for_each_subsys(ss, ssid) {
        if (parent->child_subsys_mask & (1 << ssid)) {
            ret = create_css(cgrp, ss,parent->subtree_control & (1 << ssid));
                                        /*为绑定子系统创建 cgroup_subsys_state 实例，添加文件*/
            ...
        }
    }

    if (!cgroup_on_dfl(cgrp)) {          /*非默认 hierarchy*/
        cgrp->subtree_control = parent->subtree_control;
        cgroup_refresh_child_subsys_mask(cgrp);
                    /*重置 cgrp->child_subsys_mask 值，只有在默认 hierarchy 中才会计算此值，
                    *否则直接赋值为 cgrp->subtree_control。*/
    }

    kernfs_activate(kn);

    ret = 0;
    goto out_unlock;
    ...
out_unlock:
    cgroup_kn_unlock(parent_kn);
    return ret;
    ...
}
```

cgroup_mkdir()函数主要完成以下工作：

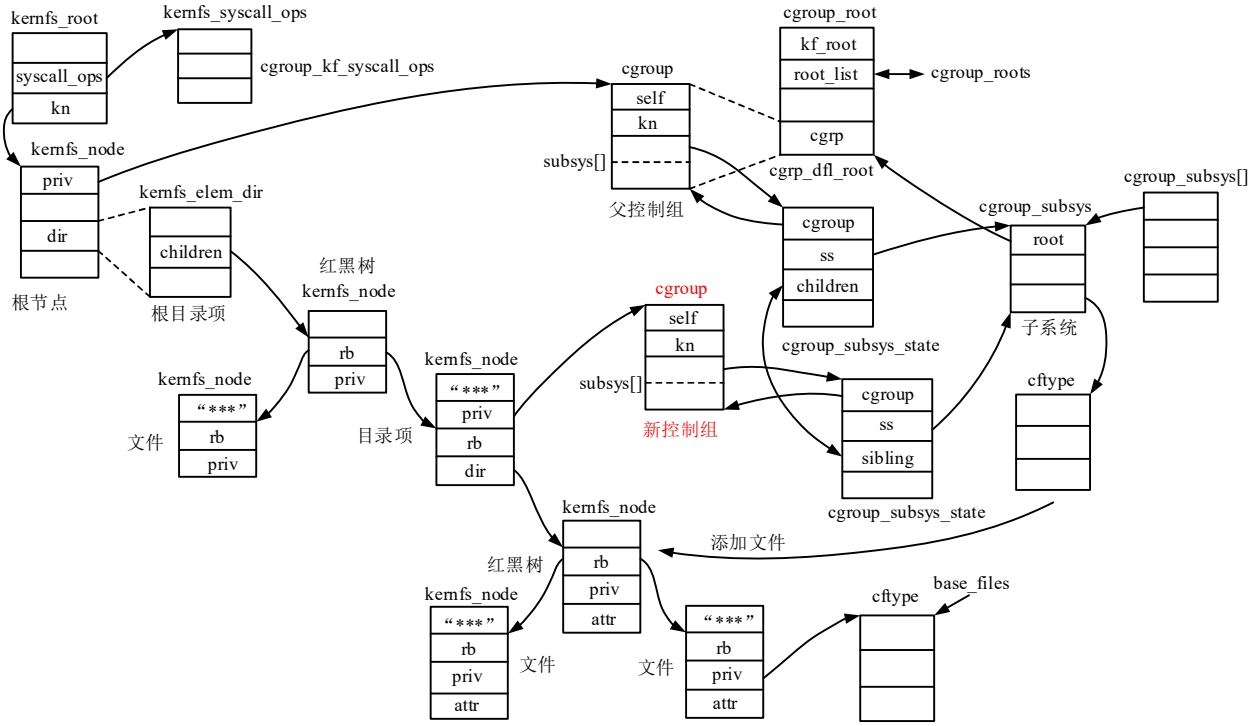
（1）创建并设置表示控制组的 cgroup 实例。

（2）创建并设置表示 kernfs 文件系统目录项的 kernfs_node 实例。

（3）在控制组目录项下添加默认 cftype 实例数组表示的文件。

（4）为新控制组绑定的子系统创建并设置 cgroup_subsys_state 实例，添加子系统关联 cftype 实例数组表示的文件。

cgroup_mkdir()函数执行结果如下图所示：



这里需要说明的是新控制组关联的子系统如何确定。在初始化时，所有子系统绑定至默认 hierarchy，而默认 hierarchy 对用户是不可见的。用户需要通过挂载操作创建新的 hierarchy，在挂载参数中会指定绑定的子系统，此时指定的子系统会从默认的 hierarchy 中迁移绑定至新创建的 hierarchy，这项工作由函数 rebind_subsystems()完成。绑定至 hierarchy 的子系统，也将绑定至 hierarchy 下的控制组。

rebind_subsystems()函数中将设置绑定子系统在 cgroup->subtree_control 和 cgroup->child_subsys_mask 位图成员中的标记位。在创建子控制组时，这两个成员值会传递给子控制组。

在 cgroup_mkdir()函数中将检测父控制组 child_subsys_mask 和 subtree_control 成员值，为子控制组绑定的子系统创建 cgroup_subsys_state 实例，并在子控制组目录项下添加文件，这项工作由 create_css()函数完成。

下面介绍一下 cgroup_addrm_files()和 create_css()函数的实现。

■添加文件

在 cgroup_mkdir()函数中将调用 **cgroup_addrm_files()**函数，通过 cftype 实例数组在控制组目录项下添加文件（也可以用于删除文件），函数定义如下：

```
static int cgroup_addrm_files(struct cgroup *cgrp, struct cftype cfts[],bool is_add)
/*cgrp：控制组指针，cfts[]：cftype 实例数组，is_add：是添加文件还是删除文件*/
{
    struct cftype *cft;
    int ret;

    lockdep_assert_held(&cgroup_mutex);

    for (cft = cfts; cft->name[0] != '\0'; cft++) {        /*遍历 cftype 实例数组，逐个添加文件*/
        /*根据标记值，跳过不适用于本控制组的 cftype 实例*/
        if ((cft->flags & __CFTYPE_ONLY_ON_DFL) && !cgroup_on_dfl(cgrp))
            continue;
```

```
            if ((cft->flags & __CFTYPE_NOT_ON_DFL) && cgroup_on_dfl(cgrp))
                continue;
            if ((cft->flags & CFTYPE_NOT_ON_ROOT) && !cgroup_parent(cgrp))
                continue;
            if ((cft->flags & CFTYPE_ONLY_ON_ROOT) && cgroup_parent(cgrp))
                continue;

            if (is_add) {
                ret = cgroup_add_file(cgrp, cft);       /*添加文件*/
                ...
            } else {
                cgroup_rm_file(cgrp, cft);       /*删除文件*/
            }
        }       /*遍历 cftype 实例数组结束*/
        return 0;
    }
```

cgroup_addrm_files()函数调用 **cgroup_add_file()**函数为 cftype 实例在控制组目录项下创建文件，函数定义如下：

```
    static int cgroup_add_file(struct cgroup *cgrp, struct cftype *cft)
    {
        char name[CGROUP_FILE_NAME_MAX];
        struct kernfs_node *kn;
        struct lock_class_key *key = NULL;
        int ret;

    #ifdef CONFIG_DEBUG_LOCK_ALLOC
        key = &cft->lockdep_key;
    #endif
        kn = __kernfs_create_file(cgrp->kn, cgroup_file_name(cgrp, cft, name),
                                  cgroup_file_mode(cft), 0, cft->kf_ops, cft,NULL, key);    /*在 kernfs 中添加文件*/
        /*文件名称为“子系统名称.cftype 名称”（没关联子系统直接为 cftype 名称），
        *关联 kernfs_ops 实例为 cftype 关联的实例，kernfs_node.priv 指向 cftype 实例。
        */
        ...
        ret = cgroup_kn_set_ugid(kn);       /*设置文件 uid、gid*/
        ...
        if (cft->write == cgroup_procs_write)
            cgrp->procs_kn = kn;
        else if (cft->seq_show == cgroup_populated_show)
            cgrp->populated_kn = kn;
        return 0;
    }
```

■创建 cgroup_subsys_state 实例

在 cgroup_mkdir()函数中将为子控制组绑定的子系统创建 cgroup_subsys_state 实例，并在子控制组目录项下添加文件，这项工作由 create_css()函数完成。

create_css()函数定义如下：

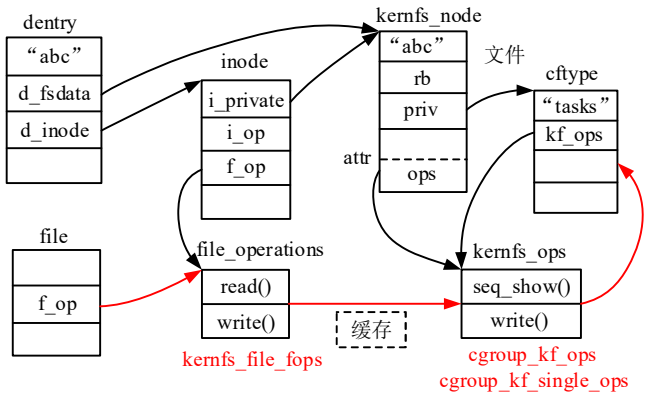
```
static int create_css(struct cgroup *cgrp, struct cgroup_subsys *ss,bool visible)
/*visible：其值表示父控制组 subtree_control 成员中是否设置了子系统标记位*/
{
    struct cgroup *parent = cgroup_parent(cgrp);
    struct cgroup_subsys_state *parent_css = cgroup_css(parent, ss);
    struct cgroup_subsys_state *css;
    int err;

    lockdep_assert_held(&cgroup_mutex);

    css = ss->css_alloc(parent_css);     /*分配并初始化 cgroup_subsys_state 实例*/
    ...
    init_and_link_css(css, ss, cgrp); /*建立 cgroup_subsys_state 与 cgroup_subsys、cgroup 之间关联*/

    err = percpu_ref_init(&css->refcnt, css_release, 0, GFP_KERNEL);        /*初始化引用计数*/
    ...
    err = cgroup_idr_alloc(&ss->css_idr, NULL, 2, 0, GFP_NOWAIT);       /*分配 ID 值*/
    ...
    css->id = err;       /*赋值 ID 值*/

    if (visible) {        /*父控制组 subtree_control 成员子系统标记位是否设置*/
        err = cgroup_populate_dir(cgrp, 1 << ss->id);
                                    /*在控制组下创建子系统关联 cftype 实例数组项对应的文件*/
        ...
    }
    list_add_tail_rcu(&css->sibling, &parent_css->children);       /*将实例添加到父子层次结构*/
    cgroup_idr_replace(&ss->css_idr, css, css->id);

    err = online_css(css);     /*调用 ss->css_online(css)，设置 cgroup->subsys[ss->id]=css*/
    ...
    return 0;
    ...
}
```

用户进程可通过 rmdir()系统调用（rmdir 命令）删除控制组。rmdir()系统调用的实现与 mkdir()系统调用类似，最终将调用 kernfs_syscall_ops 结构体 cgroup_kf_syscall_ops 实例中的 rmdir()函数，即 cgroup_rmdir()函数完成删除操作。删除操作主要是销毁各数据结构实例，函数源代码请读者自行阅读。

3 读写文件

内核默认的公共 cftype 实例以及控制组绑定子系统下的 cftype 实例，在控制组目录项下将创建对应的文件。进程迁入控制组，进程对子系统参数的获取/设置都是通过对文件的读写来实现的。

由前面介绍的 kernfs 文件系统可知，文件 inode 实例关联的 file_operations 实例为 kernfs_file_fops，此实例中读写函数将调用 kernfs_node 实例关联的 kernfs_ops 实例中的函数，如下图所示。



在由 cftype 实例创建文件时，kernfs_node 实例关联的 kernfs_ops 实例来自于 cftype 实例，而 cftype 关联的 kernfs_ops 实例在初始化 cftype 实例（数组）的 cgroup_init_cftypes()函数中赋值，函数代码简列如下。

```
static int cgroup_init_cftypes(struct cgroup_subsys *ss, struct cftype *cfts)
/*ss：关联的子系统，cfts：指向 cftype 实例数组*/
{
    struct cftype *cft;

    for (cft = cfts; cft->name[0] != '\0'; cft++) {        /*遍历 cftype 数组项*/
        struct kernfs_ops *kf_ops;
        if (cft->seq_start)       /*设置 cftype 实例关联的 kernfs_ops 实例*/
            kf_ops = &cgroup_kf_ops;        /*顺序文件，迭代读*/
        else
            kf_ops = &cgroup_kf_single_ops;     /*顺序文件，单次读*/
        ...
    }
    ...
}
```

cgroup_init_cftypes()函数中将根据 cftype 实例中是否定义了 seq_start()函数，赋予不同的 kernfs_ops 实例。cgroup_kf_single_ops 和 cgroup_kf_ops 实例定义如下：

```
static struct kernfs_ops cgroup_kf_single_ops = {     /*单次读，不需要迭代*/
    .atomic_write_len   = PAGE_SIZE,
    .write              = cgroup_file_write,     /*调用 cft->write()或 write_u64()或 write_s64()函数*/
    .seq_show           = cgroup_seqfile_show,   /*调用 cft->seq_show()或 read_u64()或 read_s64()函数*/
};

static struct kernfs_ops cgroup_kf_ops = {     /*迭代读，如果一次读的数据不够需要迭代进行*/
    .atomic_write_len   = PAGE_SIZE,
```

```
    .write          = cgroup_file_write,
    .seq_start      = cgroup_seqfile_start,      /*调用 cftype 实例中的相应迭代器函数*/
    .seq_next       = cgroup_seqfile_next,
    .seq_stop       = cgroup_seqfile_stop,
    .seq_show       = cgroup_seqfile_show,
};
```

7.15.5 进程与控制组

前面介绍了控制组、子系统的结构以及 cgroup 文件系统。控制组、子系统最终是用来管理、控制进程行为的。本小节将介绍进程与控制组的关系，主要是控制组对进程的管理和进程迁入控制组的实现。

1 管理结构初始化

进程 task_struct 结构体中与控制组相关的成员如下所示：

```
task_struct{
    ...
 ifdef CONFIG_CGROUPS              /*如果配置支持控制组*/
    struct css_set __rcu    *cgroups;      /*指向 css_set 结构体实例*/
    struct list_head    cg_list;           /*将 task_struct 实例添加到 css_set 实例中双链表*/
 #endif
     ...
}
```

task_struct 结构体中 cgroups 成员指向 css_set 结构体，定义如下（/include/linux/cgroup-defs.h）：

```
struct css_set {
    atomic_t refcount;          /*引用计数，使用此实例的进程（线程）数量*/
    struct hlist_node hlist;    /*散列链表节点成员*/

    struct list_head tasks;         /*task_struct 实例通过 task_struct.cg_list 成员添加到此双链表*/
    struct list_head mg_tasks;      /*链接使用同一 css_set 实例的进程/线程 task_struct 实例*/
    struct list_head cgrp_links;        /*链接 cgrp_cset_link 实例，用于管理控制组*/

    struct cgroup *dfl_cgrp;      /*指向默认的控制组*/
    struct cgroup_subsys_state *subsys[CGROUP_SUBSYS_COUNT];    /*指向绑定子系统状态*/

    struct list_head mg_preload_node;
    struct list_head mg_node;

    struct cgroup *mg_src_cgrp;
    struct css_set *mg_dst_cset;
    struct list_head e_cset_node[CGROUP_SUBSYS_COUNT];
                                        /*添加到 cgroup 实例 e_csets[]双链表中*/
    struct rcu_head rcu_head;
```

};

css_set 结构体包含一个 cgroup_subsys_state 指针数组，表示进程受哪些子系统控制。

cgrp_links 双链表成员链接 cgrp_cset_link 实例，实例关联了进程加入的控制组。

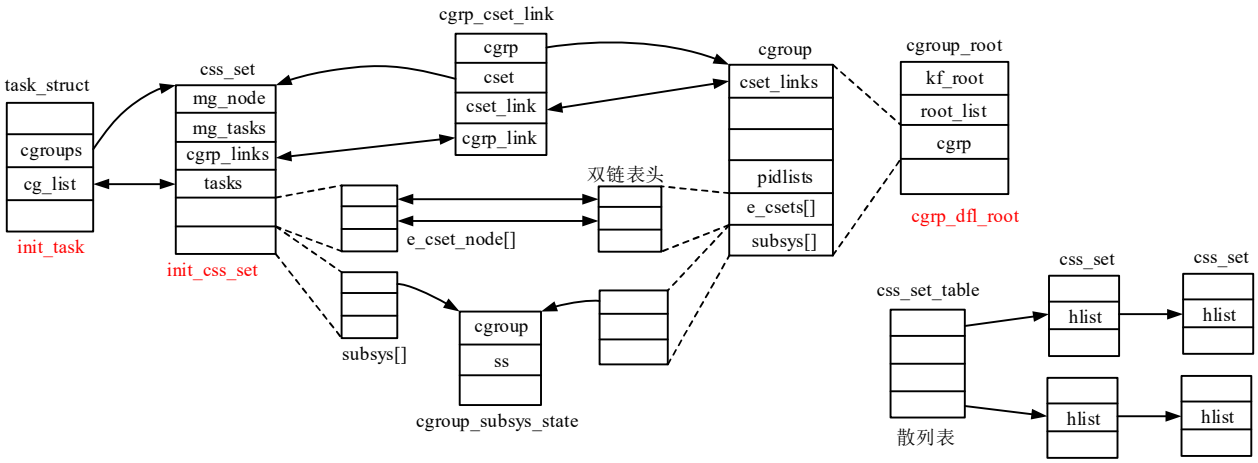
内核定义了全局散列表 css_set_table 用于管理 css_set 实例，如下所示：

```
#define CSS_SET_HASH_BITS   7
static DEFINE_HASHTABLE(css_set_table, CSS_SET_HASH_BITS);
```

内核静态定义了初始的 css_set 结构体实例 **init_css_set**，并赋予内核自身 task_struct 实例 init_task：

```
struct css_set init_css_set = {
    .refcount       = ATOMIC_INIT(1),
    .cgrp_links         = LIST_HEAD_INIT(init_css_set.cgrp_links),
    .tasks              = LIST_HEAD_INIT(init_css_set.tasks),
    .mg_tasks           = LIST_HEAD_INIT(init_css_set.mg_tasks),
    .mg_preload_node  = LIST_HEAD_INIT(init_css_set.mg_preload_node),
    .mg_node        = LIST_HEAD_INIT(init_css_set.mg_node),
};
```

在前面介绍的 cgroup 初始化过程中，包含对 css_set 等实例的初始化，请读者先看下图所示的初始化结果，随后将介绍相关的初始化在哪个函数中实现：



●在 cgroup_init_early()中，将 init_task.cgroups 成员指向 init_css_set 实例。

●在初始化子系统 cgroup_init_subsys()函数中，init_css_set.subsys[]成员指向各子系统为默认 hierarchy 根节点控制组创建的 cgroup_subsys_state 实例。

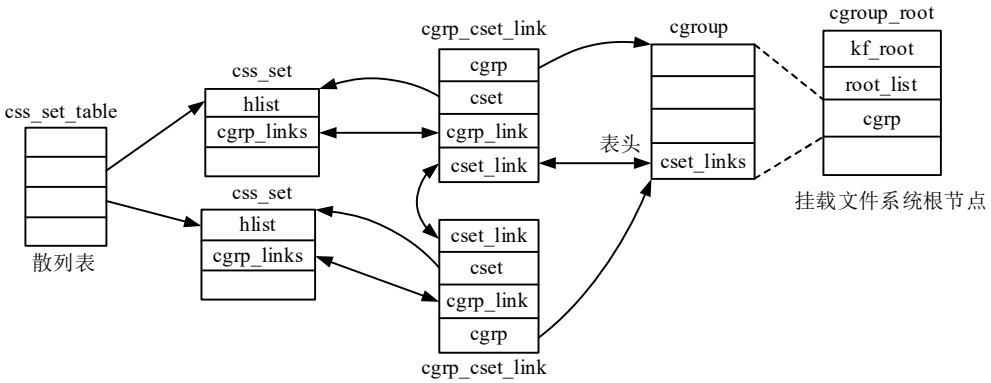
●在 cgroup_init()函数中，将 init_css_set 实例添加到全局散列表 css_set_table。

●在激活默认 hierarchy 根节点的 cgroup_setup_root()函数中（由 cgroup_init()函数调用），为散列表中现有的各 css_set 实例分别创建并设置 cgrp_cset_link 实例，cgrp_cset_link 实例添加到 css_set.cgrp_links 双链表和根节点控制组 cgroup.cset_links 双链表，即建立现有各 css_set 实例与默认 hierarchy 根节点控制组的关联。

●在 cgroup_init()函数中，将 init_css_set.e_cset_node[]各数组项表示的双链表节点添加到默认 hierarchy 根节点控制组中 e_csets[]数组项表示的双链表中。

●在第一次挂载 cgroup 文件系统时，挂载函数将调用 cgroup_enable_task_cg_lists()函数，将内核当前所有进程/线程添加到 init_css_set.tasks 双链表（链接 task_struct.cg_list 成员）。

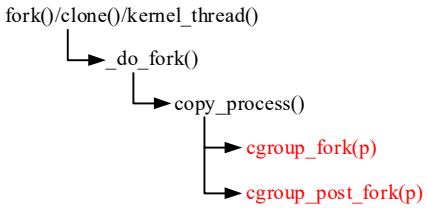
另外，在每次执行挂载 cgroup 文件系统时，挂载函数中将调用 cgroup_setup_root()函数为内核现有的所有 css_set 实例分别创建并设置 cgrp_cset_link 实例，cgrp_cset_link 实例添加到 css_set.cgrp_links 双链表和挂载 cgroup 文件系统根节点控制组 cgroup.cset_links 双链表，即建立现有各 css_set 实例与新创建 hierarchy 根节点控制组的关联，如下图所示。



2 控制组继承

内核在刚启动时只有内核自身一个进程，其 css_set 实例是静态定义并设置的。内核在创建进程/线程时，css_set 实例如何继承呢？

创建进程/线程的 fork()/clone()系统调用，以及内核创建内核线程的函数最终都要调用 copy_process()函数来产生新的进程/线程。copy_process()函数中包含父子进程/线程 css_set 实例的继承，函数调用关系如下图所示：



cgroup_fork()函数定义如下：

```
void cgroup_fork(struct task_struct *child)
/*child：子进程 task_struct 实例指针*/
{
    RCU_INIT_POINTER(child->cgroups, &init_css_set);      /*指向 init_css_set 实例*/
    INIT_LIST_HEAD(&child->cg_list);                /*初始化双链表节点成员*/
}
```

cgroup_post_fork()函数定义如下：

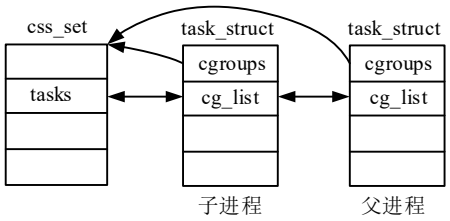
```
void cgroup_post_fork(struct task_struct *child)
{
    struct cgroup_subsys *ss;
    int i;
    if (use_task_css_set_links) {          /*系统已挂载了 cgroup 文件系统*/
        struct css_set *cset;
        down_write(&css_set_rwsem);
        cset = task_css_set(current);        /*当前（父进程）关联的 css_set 实例*/
```

```
        if (list_empty(&child->cg_list)) {
            rcu_assign_pointer(child->cgroups, cset);      /*指向父进程关联的 css_set 实例*/
            list_add(&child->cg_list, &cset->tasks);       /*添加到 css_set 实例中双链表*/
            get_css_set(cset);        /*增加 css_set 实例引用计数*/
        }
        up_write(&css_set_rwsem);
    }

    /*调用子系统定义的 fork()函数，如果有的话*/
    for_each_subsys_which(ss, i, &have_fork_callback)
        ss->fork(child);
}
```

由以上函数可知，要分两种情况讨论子进程关联的 css_set 实例，一是系统没有挂载任何 cgroup 文件系统的情况，二是系统挂载了 cgroup 文件系统的情况。

（1）系统没有挂载 cgroup 文件系统：子进程关联内核定义的 init_css_set 实例，并且在第一次挂载 cgroup 文件系统时，所有进程 task_struct 实例通过 cg_list 双链表节点成员添加到 init_css_set.tasks 双链表。

（2）系统挂载了 cgroup 文件系统：子进程关联父进程关联的 css_set 实例，并添加到此实例中的双链表中，如下图所示。



最开始所有进程/线程都关联 init_css_set 实例，在挂载 cgroup 文件系统后，用户进程可迁入到某个控制组，迁入后将为进程创建新的 css_set 实例，详见下文。

进程退出时（exit()系统调用），将调用 cgroup_exit()函数（由 do_exit()函数调用）执行 css_set 等实例的清理工作，函数源代码请读者自行阅读。

3 迁移进程

内核静态定义的默认 hierarchy 对用户是不可见的，用户需要通过挂载操作创建新的可见的 hierarchy，同时创建控制组。创建控制组后，用户可以将进程迁入控制组，这项工作是通过将进程 PID 值写入控制组目录项下"**tasks**"文件来实现的。

内核静态定义了 cftype 实例数组 cgroup_legacy_base_files 数组，此数组实例将在非默认 hierarchy 下的控制组目录项下创建相应的文件。cgroup_legacy_base_files 数组简列下：

```
static struct cftype cgroup_legacy_base_files[] = {
    {
        .name = "cgroup.procs",          /*文件名称*/
        .seq_start = cgroup_pidlist_start,
        .seq_next = cgroup_pidlist_next,
        .seq_stop = cgroup_pidlist_stop,
```

```
            .seq_show = cgroup_pidlist_show,
            .private = CGROUP_FILE_PROCS,
            .write = cgroup_procs_write,
                                /*将线程组 TGID 定入此文件，表示将线程组中所有线程迁入控制组*/
            .mode = S_IRUGO | S_IWUSR,
        },
        ...
        {
            .name = "tasks",
            .seq_start = cgroup_pidlist_start,      /*读文件内容函数*/
            .seq_next = cgroup_pidlist_next,
            .seq_stop = cgroup_pidlist_stop,
            .seq_show = cgroup_pidlist_show,
            .private = CGROUP_FILE_TASKS,
            .write = cgroup_tasks_write, /*将进程/线程 PID 写入此文件，表示将进程/线程迁入控制组*/
            .mode = S_IRUGO | S_IWUSR,        /*访问权限*/
        },
        ...
        { }   /* terminate */
    };
```

cgroup_legacy_base_files 数组名称为"**tasks**"的 cftype 实例对应的文件（文件名称也为"tasks"）表示控制组下的进程/线程，读文件将获取控制组下进程/线程 PID 值，写文件表示将指定 PID 值的进程/线程迁入控制组。

名称为"cgroup.procs"的 cftype 实例对应的文件表示线程组及其下线程的 ID 值，将线程组 TGID 值写入此文件表示将线程组中所有线程迁入控制组。

下面以名称为"**tasks**"的 cftype 实例为例，通过其文件读写函数说明进程/线程如何迁入控制组，以及查询控制组下进程/线程 ID 值。

由前一小节的介绍可知，"**tasks**"文件 kernfs_node 实例关联的 kernfs_ops 实例应为 cgroup_kf_ops，定义如下：

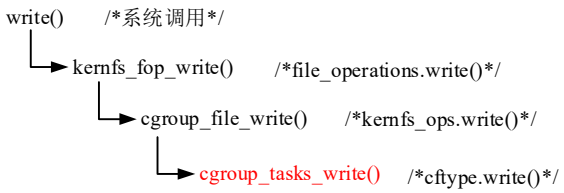
```
static struct kernfs_ops cgroup_kf_ops = {     /*迭代读，如果一次读的数据不够需要迭代进行*/
    .atomic_write_len   = PAGE_SIZE,
    .write              = cgroup_file_write,       /*写文件函数，调用 cftype 实例中 write()函数*/
    .seq_start          = cgroup_seqfile_start,    /*调用 cftype 实例中的相应函数*/
    .seq_next           = cgroup_seqfile_next,
    .seq_stop           = cgroup_seqfile_stop,
    .seq_show           = cgroup_seqfile_show,
};
```

下面将介绍写"**tasks**"文件函数的实现，即将进程迁入控制组。

■写"tasks"文件

用户进程对控制组下"**tasks**"文件的写操作，即将某个进程/线程的 PID 值写入此文件，表示将进程/线程迁入控制组。

由 kernfs 文件系统文件写操作可知，写操作函数将用户数据写入缓存区，然后调用 kernfs_ops 实例中的 write()函数，而 cgroup_kf_ops 实例中的 write()函数 **cgroup_file_write()**将调用 cftype 实例中的 write()函数执行写入操作，函数调用关系如下图所示。



"**tasks**"名称 cftype 实例中定义的 write()函数 **cgroup_tasks_write()**如下所示：

```
static ssize_t cgroup_tasks_write(struct kernfs_open_file *of,char *buf, size_t nbytes, loff_t off)
/*of：用于获取 cgroup 实例，buf：写入的是进程/线程 PID 值*/
{
    return    __cgroup_procs_write(of, buf, nbytes, off, false);
}

static ssize_t __cgroup_procs_write(struct kernfs_open_file *of, char *buf,
                                    size_t nbytes, loff_t off, bool threadgroup)
/*threadgroup：是否是线程组*/
{
    struct task_struct *tsk;
    struct cgroup *cgrp;
    pid_t pid;
    int ret;

    if (kstrtoint(strstrip(buf), 0, &pid) || pid < 0)        /*字符串转 PID 值*/
        return -EINVAL;

    cgrp = cgroup_kn_lock_live(of->kn);        /*获取控制组 cgroup 实例*/
    ...
retry_find_task:
    rcu_read_lock();
    if (pid) {
        tsk = find_task_by_vpid(pid);        /*由 PID 值查找 task_struct 实例*/
        ...
    } else {
        tsk = current;
    }
    ...
    get_task_struct(tsk);
    rcu_read_unlock();
    threadgroup_lock(tsk);
    ...
    ret = cgroup_procs_write_permission(tsk, cgrp, of);        /*权限检查*/
```

```
        if (!ret)
            ret = cgroup_attach_task(cgrp, tsk, threadgroup);        /*绑定进程（线程组）至控制组*/

        threadgroup_unlock(tsk);
        put_task_struct(tsk);
    out_unlock_cgroup:
        cgroup_kn_unlock(of->kn);
        return ret ?: nbytes;
    }
```

__cgroup_procs_write()函数首先将用户传递的字符串转换成整型数，表示进程（线程）PID 值，然后依 PID 值查找 task_struct 实例，执行权限检查，最后调用 cgroup_attach_task()函数绑定进程至控制组。下面主要介绍绑定函数 cgroup_attach_task()的实现。

●绑定进程

cgroup_attach_task()函数可以将进程/线程绑定至控制组，如果进程是线程组长，则其下所有线程也将绑定至控制组，这里为了便于理解只考虑绑定单个进程时的情形。

```
static int cgroup_attach_task(struct cgroup *dst_cgrp,struct task_struct *leader, bool threadgroup)
/*dst_cgrp：目的控制组，leader：绑定进程的 task_struct 实例*/
{
    LIST_HEAD(preloaded_csets);       /*临时双链表*/
    struct task_struct *task;
    int ret;

    /* look up all src csets */
    down_read(&css_set_rwsem);
    rcu_read_lock();
    task = leader;
    do {
        cgroup_migrate_add_src(task_css_set(task), dst_cgrp,&preloaded_csets);
                                        /*添加原 css_set 实例至 preloaded_csets 临时双链表*/
        if (!threadgroup)
            break;
    } while_each_thread(leader, task);
    rcu_read_unlock();
    up_read(&css_set_rwsem);

    ret = cgroup_migrate_prepare_dst(dst_cgrp, &preloaded_csets);
                                        /*为进程创建并设置 css_set、cgrp_cset_link 实例*/
    if (!ret)
        ret = cgroup_migrate(dst_cgrp, leader, threadgroup); /*执行迁移，进程关联新 css_set 实例等*/

    cgroup_migrate_finish(&preloaded_csets);
                                        /*完成迁移，释放 preloaded_csets 双链表中 css_set 实例*/
    return ret;
```

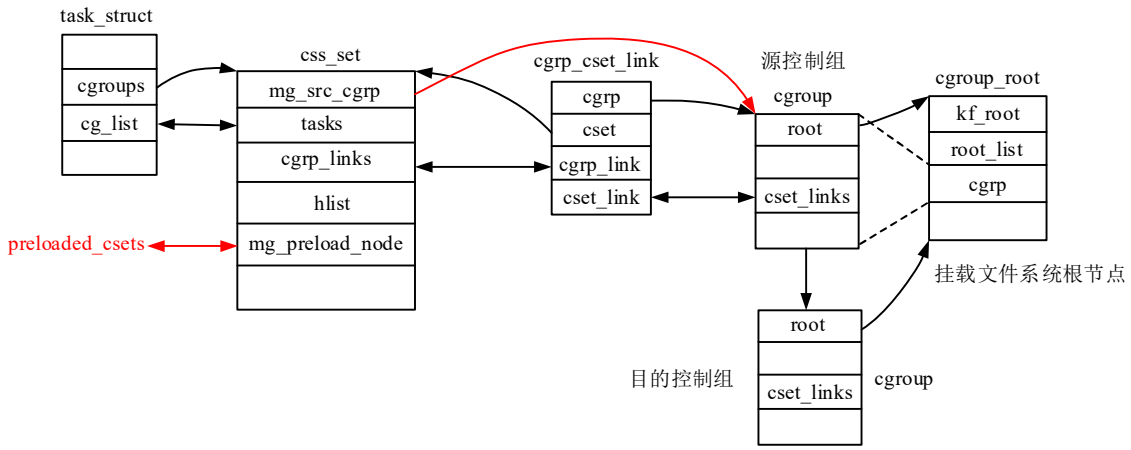
}

下面以图示的方式介绍 cgroup_attach_task()函数的执行流程，函数源代码请读者自行阅读。

（1）添加 css_set 至临时链表

由 cgroup 文件系统挂载操作可知，在激活根节点函数中将为现有的所有 css_set 实例创建 cgrp_cset_link 实例，以建立 css_set 实例与新挂载文件系统根节点（hierarchy）之间的关联。也就是说，每个 css_set 实例都与每个 hierarchy 都建立了关联，初始默认关联根节点中控制组。

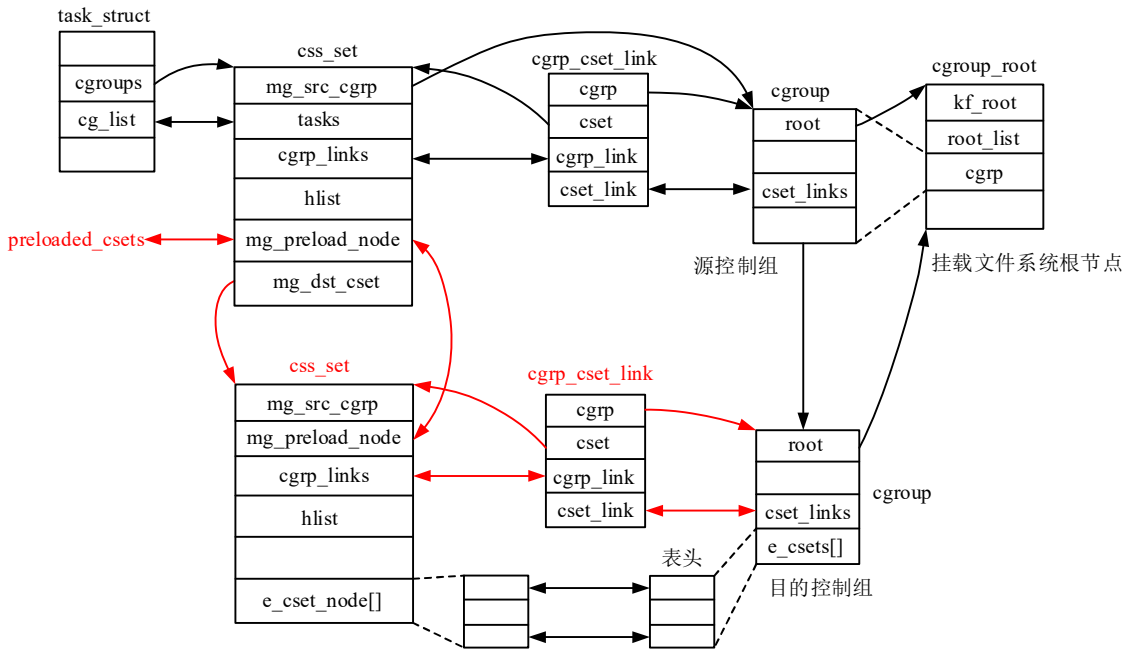
在 cgroup_migrate_add_src()函数中，将设置原 css_set 实例中的 mg_src_cgrp 成员。如果原 css_set 实例为 init_css_set，mg_src_cgrp 成员将指向目的控制组所在 hierarchy 根节点内嵌的控制组，如下图所示。否则，指向原 css_set 实例在目的控制组所在的 hierarchy 当前关联的控制组（可能是根节点内嵌控制组，也可能是上一次绑定的控制组）。



cgroup_migrate_add_src()函数中将确定目的控制组关联的根节点，搜索进程原关联的 css_set 实例 cgrp_links 双链表中 cgrp_cset_link 实例，检查其关联的控制组根节点，看是否与目的控制组根节点相同。若相同则将 css_set 实例中 mg_src_cgrp 成员指向 cgrp_cset_link 实例关联的控制组（不一定是目的控制组），将 css_set 实例通过 mg_preload_node 成员添加到 preloaded_csets 临时双链表，原 css_set 实例引用计数值加 1，如上图中红色箭头所示。

（2）准备目的 css_sets 实例

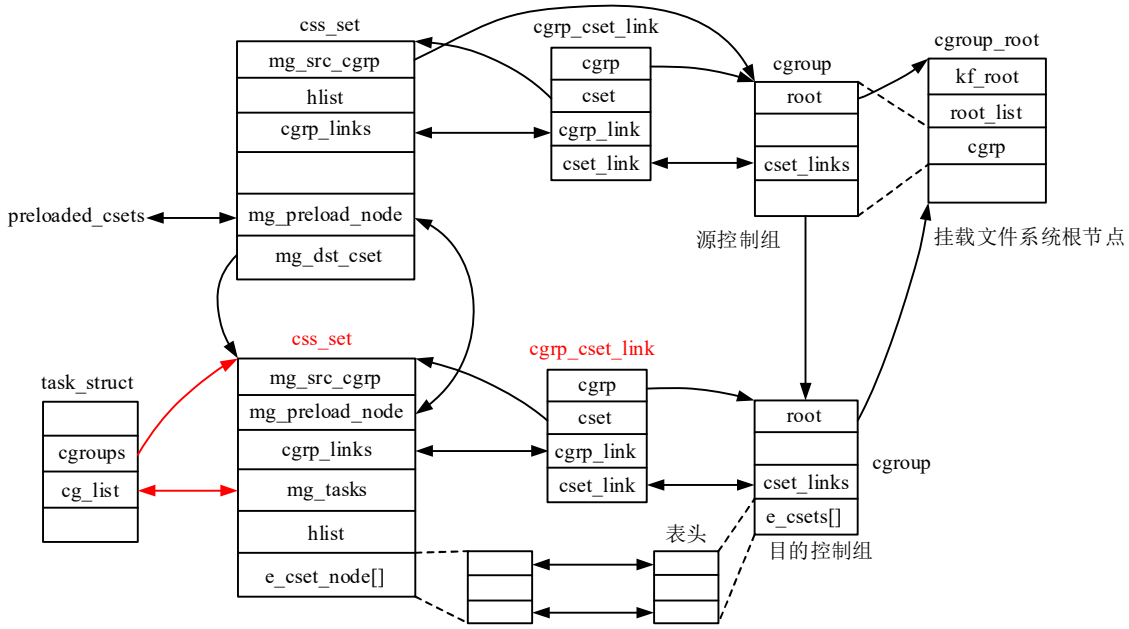
cgroup_attach_task()函数随后调用 cgroup_migrate_prepare_dst()函数为进程创建并设置新的 css_set 实例（引用计数值为 1），如下图所示：



cgroup_migrate_prepare_dst()函数除了创建 css_set 实例外，还将为其在每个 hierarchy 创建 cgrp_cset_link 实例，关联到目的控制组所在 hierarchy 的 cgrp_cset_link 实例将关联目的控制组 cgroup 实例，其它 hierarchy 的 cgrp_cset_link 实例内容与原 css_set 实例关联 cgrp_cset_link 实例的内容相同。新创建 css_set 实例将添加到 preloaded_csets 临时双链表末尾，并添加到全局散列表。此时进程还是关联原 css_set 实例，而不是关联新的 css_set 实例。

（3）执行迁移

cgroup_migrate()函数用于执行迁移操作，主要工作是将进程 task_struct 实例关联到新 css_set 实例（原实例引用计数减 1，新实例引用计数加 1），将 task_struct 实例通过 cg_list 双链表节点成员添加到新 css_set 实例中 **mg_tasks** 双链表，如下图所示。



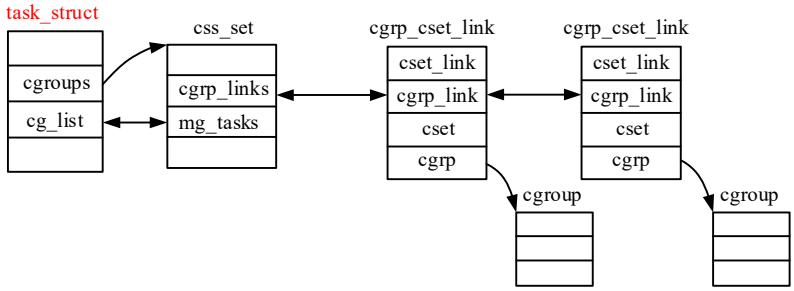
cgroup_attach_task()函数最后将调用 cgroup_migrate_finish()函数，主要工作是对 preloaded_csets 临时双链表中 css_set 实例引用计数值减 1，当计数值为 0 时，将释放 css_set 实例。

由上可知，将进程 PID 写入某个控制组下“tasks”文件，即代表将进程迁移入此控制组。这种迁移可能是将进程从默认 hierarchy 根节点控制组迁移到其它 hierarchy 下控制组，或者是将进程在同一 hierarchy 下控制组之间迁移。

在进程退出的系统调用中将调用 cgroup_exit()，此函数将进程迁出其所在控制组，并释放 css_set、cgrp_cset_link 实例，函数源代码请读者自行阅读。

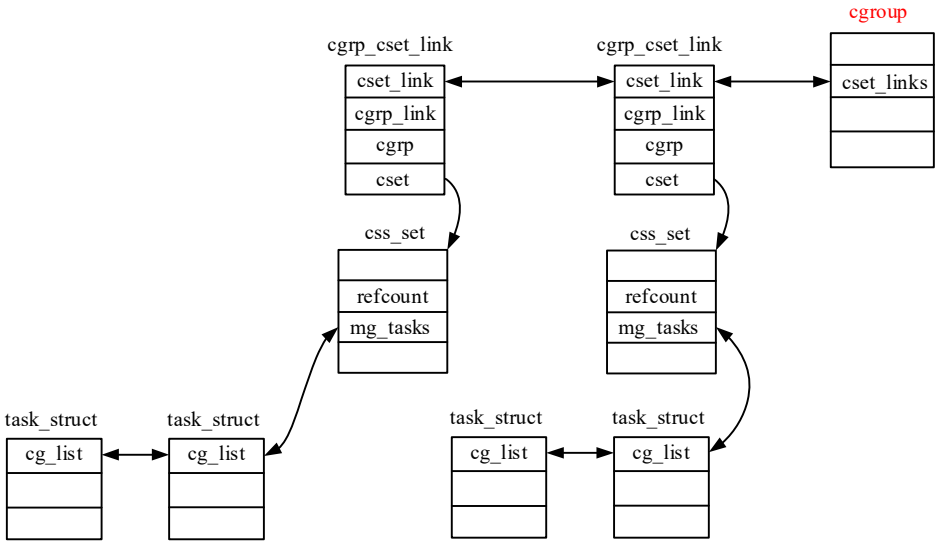
4 获取控制组进程 PID

进程通过其关联的 css_set 实例可获知进程绑定到哪些控制组，如下图所示：



css_set 实例 cgrp_links 双链成中链接的 cgrp_cset_link 实例的 cgrp 成员指向的是进程绑定的控制组。

控制组也可以通过其中的 cgrp_cset_link 实例双链表获知绑定了哪些进程，如下图所示：



以上两个图中的 cgrp_cset_link 实例双链表并不是同一个双链表，而是不同的双链表。cgrp_cset_link 实例同时添加到 css_set 实例和 cgroup 实例中的双链表。

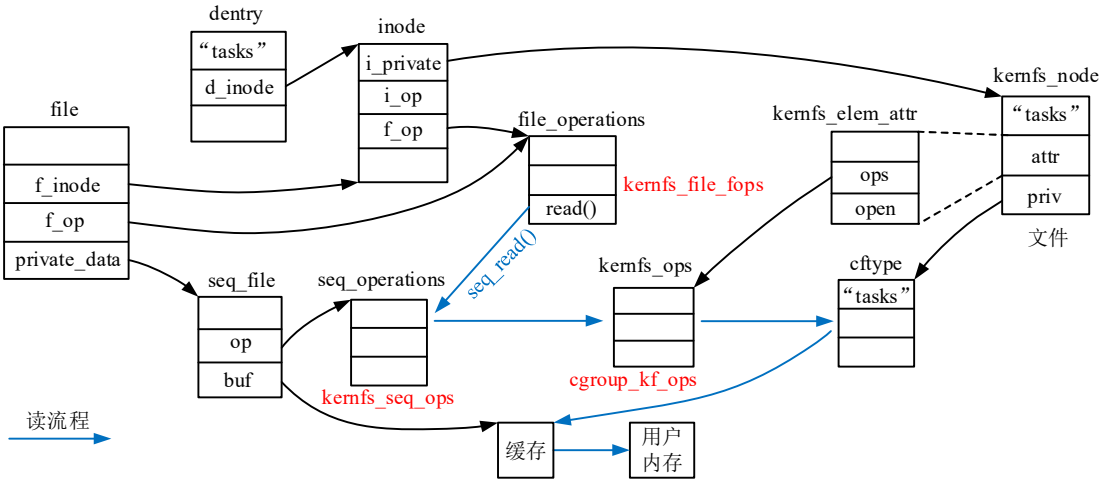
由控制组查找绑定的进程要稍微复杂一些，cgrp_cset_link 实例关联到 css_set 实例，而 css_set 实例的 mg_tasks 双链表管理的是绑定到控制组的进程/线程。refcount 成员记录了关联 css_set 实例的进程/线程数量。

用户进程通过读控制组下"**tasks**"文件的内容可获取控制组下绑定进程的 PID 值。由前一小节的介绍可知，"**tasks**"文件 kernfs_node 实例关联的 kernfs_ops 实例应为 cgroup_kf_ops，定义如下：

```
static struct kernfs_ops cgroup_kf_ops = {      /*迭代读，如果一次读的数据不够需要迭代进行*/
        .atomic_write_len   = PAGE_SIZE,
        .write              = cgroup_file_write,        /*写文件函数，调用 cftype 实例中 write()函数*/
        .seq_start          = cgroup_seqfile_start,     /*直接调用 cftype 实例中的 seq_start()函数，同下*/
        .seq_next           = cgroup_seqfile_next,
```

```
    .seq_stop       = cgroup_seqfile_stop,
    .seq_show       = cgroup_seqfile_show,
};
```

"**tasks**"文件关联的 file_operations 实例为 kernfs_file_fops，其 read()函数为 kernfs_fop_read()。由前面介绍的 kernfs 文件系统可知，由于 cgroup_kf_ops 实例中定义了 seq_show()函数，kernfs_fop_read()函数内将调用通用函数 seq_read()读取文件内容。而 seq_read()函数中又将调用 cgroup_kf_ops 实例中的迭代函数获取文件内容，如下图所示。



cgroup_kf_ops 实例中的迭代函数将直接调用 cftype 实例中的对应函数，"**tasks**"名称的 cftype 实例定义如下：

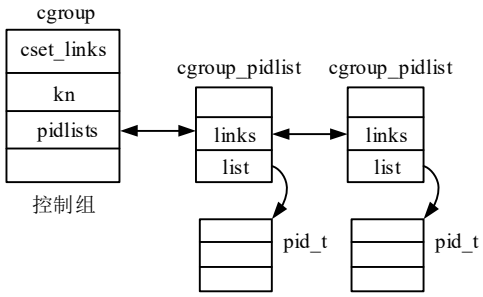
```
static struct cftype cgroup_legacy_base_files[] = {
    ...
    {
        .name = "tasks",
        .seq_start = cgroup_pidlist_start,      /*创建或查找 cgroup_pidlist 实例*/
        .seq_next = cgroup_pidlist_next,        /*返回下一个 PID 值指针*/
        .seq_stop = cgroup_pidlist_stop,        /*释放 cgroup_pidlist 实例*/
        .seq_show = cgroup_pidlist_show,
                                    /*从 cgroup_pidlist 实例 list 指向缓存中读取 PID 值填至缓存*/
        .private = CGROUP_FILE_TASKS,           /*文件类型*/
        .write = cgroup_tasks_write,
        .mode = S_IRUGO | S_IWUSR,              /*访问权限*/
    },
    ...
    { }   /* terminate */
};
```

在介绍以上迭代器函数前，先看 cgroup_pidlist 结构体的定义（/kernel/cgroup.c）：

```
struct cgroup_pidlist {
    struct { enum cgroup_filetype type; struct pid_namespace *ns; } key;      /*文件类型，命名空间*/
    pid_t *list;      /*指向保存 PID 值的数组*/
```

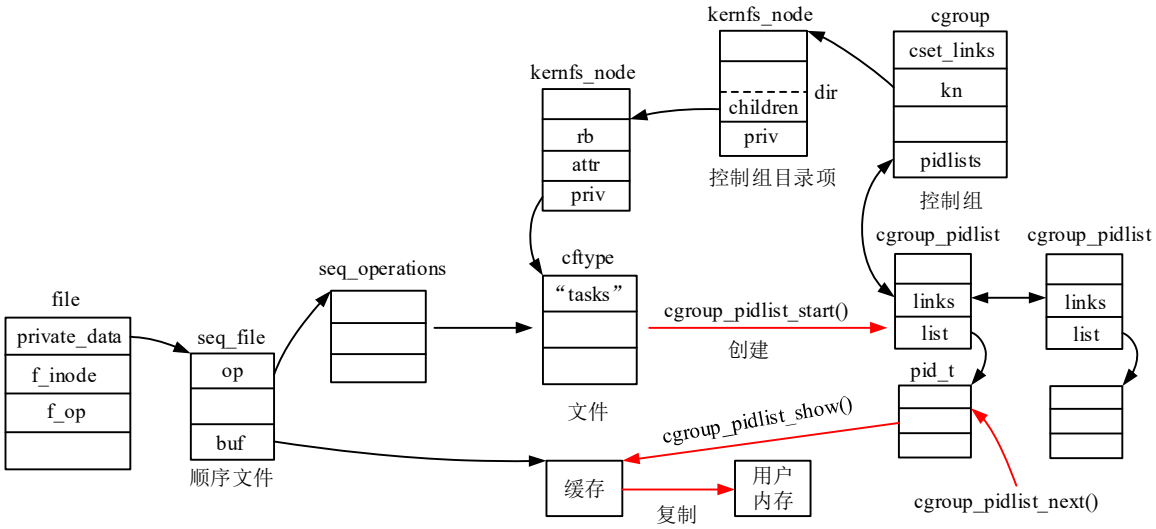
```
    int length;        /*pid_t 数组项数（PID 值数量）*/
    struct list_head links;      /*将 cgroup_pidlist 实例添加到 cgroup 实例 pidlists 双链表*/
    struct cgroup *owner;
    struct delayed_work destroy_dwork;      /*延时工作，释放 cgroup_pidlist 实例*/
};
```

cgroup_pidlist 实例管理结构如下图所示：



用户进程读取"**tasks**"文件时，将创建 cgroup_pidlist 实例及进程 PID 缓存数组（list 成员指向的数组），cgroup_pidlist 实例关联到 PID 数组。由控制组 cset_links 双链表实例关联的 css_set 实例的 refcount 成员可计算出控制组共绑定了多少个进程，以此分配 PID 数组。cgroup_pidlist 实例添加到 cgroup 实例 pidlists 双链表。

下图示意了"**tasks**"名称的 cftype 实例中迭代器函数的功能，下面只简述函数功能，源代码请读者自行阅读：



开始迭代函数 cgroup_pidlist_start()中将查找或创建 cgroup_pidlist 实例，添加到控制组 pidlists 双链表。cgroup_pidlist 实例 list 指向的数组保存了当前控制组下的进程 PID 值。此函数通过 cgroup 实例 cset_links 双链表查找到 css_set 实例，从 css_set 实例中获取进程 task_struct 实例，并从中读取进程 PID 值。

控制组 pidlists 双链表中之所以有多个实例，是因为 "cgroup.procs"文件管理的线程 PID 值也由 cgroup_pidlist 实例管理。另外，如果支持 PID 命名空间的话，每个命名空间都对应不同的 cgroup_pidlist 实例。

cgroup_pidlist 实例 list 成员指向的 PID 数组，是按 PID 值从小到大排列的。cgroup_pidlist_start()函数中的文件位置 pos 参数，实际保存的是当前读取的 PID 值。

cgroup_pidlist_next()函数返回下一个 PID 值指针，即指向 list 指向的 PID 数组项。cgroup_pidlist_show()函数将 PID 数组项中 PID 值写入顺序文件缓存。seq_read()函数最后会将 PID 值复制到用户内存。

控制组下其它文件的读写操作与"**tasks**"文件类似，请读者自行阅读源代码。

控制组只是提供了对进程分组的管理，进程资源的管理（限制）由具体使用控制组的子系统实现。

7.16 小结

Linux 奉行“万物皆文件”的哲学，文件不仅包括保存在外部存储介质中的普通文件，还可以表示外部设备、网络、设备参数、系统控制参数等。

文件必须位于某个文件系统中。文件系统可以是位于磁盘等外部存储介质中的普通文件系统，也可以是基于内存盘的文件系统，也可以是由内核数据结构组成的文件系统。文件系统是将文件以某种形式（文件系统类型）组织起来的一个实体。

通常我们所见的文件系统，是以目录项组织而成的树状层次结构。内核在虚拟文件系统中实现了一个由目录项构成的单一的树状层次结构，称为根文件系统。实际文件系统需要挂载到根文件系统某个目录项下，以导入到内核单一的树状层次结构。

虚拟文件系统中不仅包含根文件系统，还为文件系统、目录项、文件等定义了统一的操作接口。具体文件系统除了需要挂载到根文件系统外，还需要实现以上操作接口。内核通过这些统一的接口实现对文件系统、目录项、文件的操作。

虚拟文件系统中的根文件系统，相当于内核在内存中管理的一个文件库。进程操作文件前，需要建立进程与内核根文件系统中文件的联系，最终进程通过一个文件描述符（整数）来标识打开的文件。

本章前半部分介绍了虚拟文件系统的结构、文件系统的挂载，以及进程对文件的操作等，本章后半部分介绍了几种实际的文件系统类型。ext2 是基于外部存储介质的文件系统，它曾经是 Linux 系统的标准文件系统。proc、sysfs（kernfs）文件系统是基于内核数据结构（对象）的文件系统，用于导出内核（进程）信息，cgroup（kernfs）文件系统用于实现控制组的用户操作接口。