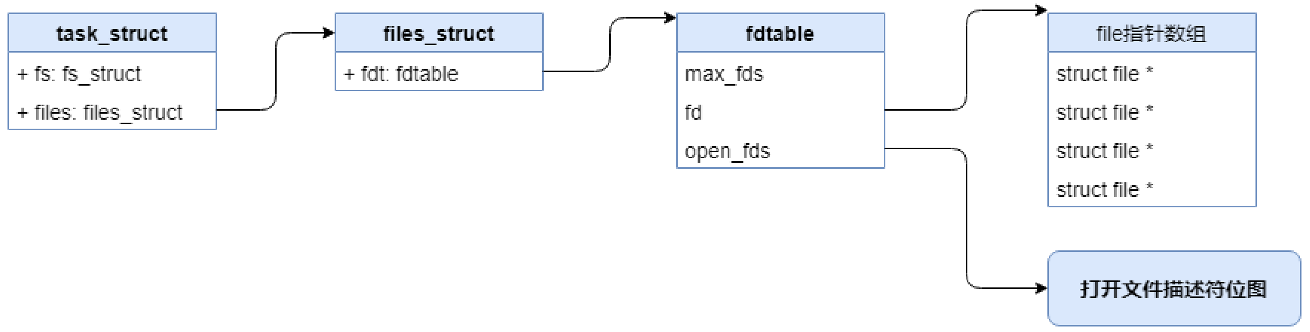
open flags 调研（已实现）

open接口原型

```
# man page
open, openat, creat - open and possibly create a file

#include <fcntl.h>
int open(const char *pathname, int flags);
int open(const char *pathname, int flags, mode_t mode);

int creat(const char *pathname, mode_t mode);

int openat(int dirfd, const char *pathname, int flags);
int openat(int dirfd, const char *pathname, int flags, mode_t mode);
int openat2(int dirfd, const char *pathname, const struct open_how *how, size_t size);
```

open系统调用会打开pathname指定的文件（如果不存在，如果携带O_CREAT
flag则会创建），返回一个文件描述符fd（该fd是进程打开文件描述符表的index），在后续系统调用（read(2)、write(2)、lseek(2)、fcntl(2)
etc.）中指向这个打开的文件。打开的文件描述符记录中保存着文件的offset 和 文件status。

每个进程都有个 task_struct 描述符用来描述进程相关的信息，其中有个 files_struct 类型的 files 字段，里面有个保存了当前进程所有已打开文件 描述符的数组，而通过 fd 就可以找到具体的文件描述符:



open & openat
系统调用的区别：如果pathname是绝对路径，则dirfd参数没用。如果pathname是相对路径，并且dirfd的值不是AT_FDCWD，则pathname的参照物是相对于dirfd指向的目录，而不是进程的当前工作目录；反之，如果dirfd的值是AT_FDCWD，pathname则是相对于进程当前工作目录的相对路径，此时等同于open。

open flags

flags定义

flags通过宏定义实现，定义见fcntl.h，主要包括如下flag

# 红色是不支持且会执行结果错误；橙色是暂不确定但不影响写入结果；紫色为暂时无法测试；黑色是已经支持

```
#define O_RDONLY 00000000
#define O_WRONLY 00000001
#define O_RDWR 00000002
#define O_CREAT 00000100
#define O_EXCL 00000200
#define O_NOCTTY 00000400
#define O_TRUNC 00001000
#define O_APPEND 00002000
#define O_NONBLOCK 00004000
#define O_SYNC 00010000(before linux 2.6.33) 04010000(after linux 2.6.33)
#define O_DSYNC 00010000(after linux 2.6.33)
#define FASYNC 00020000
#define O_DIRECT 00040000
#define O_LARGEFILE 00100000
#define O_DIRECTORY 00200000
#define O_NOFOLLOW 00400000
#define O_NOATIME 01000000
#define O_CLOEXEC 02000000
#define O_PATH 010000000(since linux 2.6.39)
#define O_TMPFILE 020000000|O_DIRECTORY
#define O_NDELAY O_NONBLOCK(O_NDELAY是在System V的早期版本引入的，后改进为O_NONBLOCK)
```

flags中必须access mode：O_RDONLY, O_WRONLY, O_RDWR其中之一；

文件创建标志只影响打开操作，文件状态标志影响后面的读写操作
file creation flags: O_CLOEXEC, O_CREAT, O_DIRECTORY, O_EXCL, O_NOCTTY, O_NOFOLLOW, O_TMPFILE, and O_TRUNC
file status flags: O_APPEND, FASYNC, O_DIRECT, O_SYNC(O_DSYNC), O_LARGEFILE, O_NOATIME, O_NONBLOCK, O_PATH

flags的含义

O_RDONLY: 只读
O_WRONLY: 只写
O_RDWR: 读写
O_CREAT: 当pathname对应的文件不存在时则创建它，文件uid为进程uid，gid为进程gid或父目录gid（取决于SGID是否置位）；当flags中出现O_CREAT 或 O_TMPFILE时，mode参数必须提供，否则会使用栈中随机字节填充；通常在没有ACL的情况下，有效的mode是经过与进程mask作用后的结果（mode & ~mask）。

```
# symbolic constants
S_IRWXU  00700
S_IRUSR  00400
S_IWUSR  00200
S_IXUSR  00100
S_IRWXG  00070
S_IRGRP  00040
S_IWGRP  00020
S_IXGRP  00010
S_IRWXO  00007
S_IROTH  00004
S_IWOTH  00002
S_IXOTH  00001

S_ISUID  0004000
S_ISGID  0002000
S_ISVTX  0001000

S_IFMT   0170000
S_IFSOCK 0140000     socket
S_IFLNK  0120000     (symbolic link)
S_IFREG  0100000
S_IFBLK  0060000     (block device)
S_IFDIR  0040000
S_IFCHR  0020000     (character device)
S_IFIFO  0010000     (fifo)
```

O_EXCL：与O_CREATE一起使用，如果pathname已经存在则返回失败(EEXIST)，否则创建文件成功。
O_NOCTTY：该参数不会使打开的文件成为该进程的控制终端。如果没有指定这个标志，那么任何一个 输入都将会影响用户的进程。
O_TRUNC：如果文件存在，且是个普通文件，具有对该文件的写权限，该flag会将文件长度截断为0。
O_APPEND：追加写，每次write都会将file offset 指向文件尾（file offset的修改和write操作在一个原子操作中完成）。
O_NONBLOCK O_NDELAY:
O_NONBLOCK和O_NDELAY所产生的结果都是使I/O变成非阻塞模式(non-blocking)，在读取不到数据或是写入缓冲区已满会马上return，而不会阻塞等待。差别在于：在读操作时，如果读不到数据，O_NDELAY会使I/O函数马上返回0，但这又衍生出一个问题，因为读取到文件末尾(EOF)时返回的也是0，这样无法区分是哪种情况。因此O_NONBLOCK就产生出来，它在读取不到数据时会回传-1，并且设置errno为EAGAIN。
O_SYNC：每次write都等到物理I/O完成，包括write引起的文件属性的更新。
O_DSYNC：每次write都等待物理I/O完成，但是如果写操作不影响读取刚写入的数据，则不等待文件属性更新（在linux 2.6.33之前只有O_SYNC flag，但是在绝大多数文件系统中对O_SYNC的实现都是O_DSYNC的含义，在2.6.33版本支持了O_DSYNC flag，且值使用原O_SYNC的值，但为了兼容老版本的O_SYNC，现在O_SYNC=O_DSYNC|04000000）。
FASYNC：异步的，启用signal-driven I/O。
O_DIRECT：直接I/O，执行磁盘I/O时绕过缓冲区高速缓存，从用户空间直接将数据传递到文件或磁盘设备。
O_LARGEFILE：使得32位操作系统对大文件支持（_FILE_OFFSET_BITS=64）。
O_DIRECTORY：以目录形式打开，如果pathname不是一个目录则会打开失败。
O_NOFOLLOW：如果pathname是一个符号链接，则会打开失败（ELOOP）。
O_NOATIME：不更新Inode中的last access time（进程uid=文件uid或者进程在它的user namespace有CAP_FOWNER，而文件的uid在这个namespace中有一个映射）。
O_CLOEXEC：在进程执行exec系统调用时关闭此打开的文件描述符，防止父进程泄露打开的文件给子进程。
O_PATH：使用 O_PATH 将不会真正打开一个文件，而只是准备好该文件的文件描述符，而且如果使用该标志位的话系统会忽略大部分其他的标志位(除了O_CLOEXEC，O_DIRECTORY，O_NOFOLLOW)。特别是如果配合使用 O_NOFOLLOW，那么遇到符号链接的时候将会返回这个符号链接本身的文件描述符，而非符号链接所指的对象。
O_TMPFILE：用于生产临时的无名的普通文件，pathname指定一个目录。

libfuse open

void(* fuse_lowlevel_ops::open)(fuse_req_t req, fuse_ino_t ino, struct fuse_file_info *fi)

```
Open a file

Open flags are available in fi->flags. The following rules apply.

Creation (O_CREAT, O_EXCL, O_NOCTTY) flags will be filtered out / handled by the kernel.
Access modes (O_RDONLY, O_WRONLY, O_RDWR) should be used by the filesystem to check if the operation is
permitted.
If the -o default_permissions mount option is given, this check is already done by the kernel before calling
open() and may thus be omitted by the filesystem.
When writeback caching is enabled, the kernel may send read requests even for files opened with O_WRONLY. The
filesystem should be prepared to handle this.
When writeback caching is disabled, the filesystem is expected to properly handle the O_APPEND flag and ensure
that each write is appending to the end of the file.
When writeback caching is enabled, the kernel will handle O_APPEND. However, unless all changes to the file come
through the kernel this will not work reliably.
The filesystem should thus either ignore the O_APPEND flag (and let the kernel handle it), or return an error
(indicating that reliably O_APPEND is not available).
Filesystem may store an arbitrary file handle (pointer, index, etc) in fi->fh, and use this in other all other
file operations (read, write, flush, release, fsync).

Filesystem may also implement stateless file I/O and not store anything in fi->fh.

There are also some flags (direct_io, keep_cache) which the filesystem may set in fi, to change the way the file
is opened. See fuse_file_info structure in <fuse_common.h> for more details.

If this request is answered with an error code of ENOSYS and FUSE_CAP_NO_OPEN_SUPPORT is set in
fuse_conn_info.capable,
this is treated as success and future calls to open and release will also succeed without being sent to the
filesystem process.

Valid replies: fuse_reply_open fuse_reply_err

Parameters
req request handle
ino the inode number
fi file information
```

open flags 在curvefs上的测试

在现在的curvefs上进行open相关flag测试，发现已经支持部分open flags（mount 参数指定了default_permissions），VFS做了部分工作（如上open brief所述）：

```
O_CREAT
# O_WRONLY|O_CREAT, S_IWUSR
root@pubbeta1-nostest2:/tmp/fsmount# strace ./main
...
open("in.txt", O_WRONLY|O_CREAT, 0200)  = 3

O_EXCL
# O_WRONLY|O_CREAT|O_EXCL
# in.txt 0200
root@pubbeta1-nostest2:/tmp/fsmount# strace ./main
...
open("in.txt", O_WRONLY|O_CREAT|O_EXCL, 0200) = -1 EEXIST (File exists)

O_RDONLYO_WRONLYO_RDWR
# O_RDONLY
# in.txt 0200
nbs@pubbeta1-nostest2:/tmp/fsmount$ strace ./main
...
open("in.txt", O_RDONLY)                = -1 EACCES (Permission denied)

# O_WRONLY|O_APPEND
# in.txt 0400
nbs@pubbeta1-nostest2:/tmp/fsmount$ strace ./main
...
open("in.txt", O_WRONLY|O_APPEND)       = -1 EACCES (Permission denied)

# O_RDWR
# in.txt 0400
nbs@pubbeta1-nostest2:/tmp/fsmount$ strace ./main
...
open("in.txt", O_RDWR)                  = -1 EACCES (Permission denied)

# O_RDWR
# in.txt 0777
```

```
nbs@pubbeta1-nostest2:/tmp/fsmount$ strace ./main
...
open("in.txt", O_RDWR)                  = 3

O_APPEND
# O_WRONLY|O_APPEND
# in.txt 600
root@pubbeta1-nostest2:/tmp/fsmount$ strace ./main
...
open("in.txt", O_WRONLY|O_APPEND)       = 3

O_NOCTTY(open brief
# O_WRONLY|O_NOCTTY
# in.txt 0644
root@pubbeta1-nostest2:/tmp/fsmount# strace ./main
...
open("in.txt", O_WRONLY|O_NOCTTY)       = 3

O_DIRECTORY
# O_RDONLY|O_DIRECTORY
# in.txt 0644
root@pubbeta1-nostest2:/tmp/fsmount$ strace ./main
...
open("in.txt", O_RDONLY|O_DIRECTORY)    = -1 ENOTDIR (Not a directory)
# dir 755
root@pubbeta1-nostest2:/tmp/fsmount$ strace ./main
...
open("dir", O_RDONLY|O_DIRECTORY)       = 3

curvefsO_NOFOLLOW
# O_NOFOLLOW

curvefs inode mtime/atime/ctimeO_NOATIME
# O_NOATIME

O_PATH
root@pubbeta1-nostest2:/tmp/fsmount$ strace ./main
...
```

```
open("in.txt", O_RDWR|O_CREAT|O_PATH, 0200) = 3
write(3, "\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0"..., 4095) = -1 EBADF (Bad file
descriptor)
```

```
root@pubbeta1-nostest2:/tmp/fsmount$ strace ./main
open("in.txt", O_RDONLY|O_PATH)         = 3
```

open flags 实现方式

cephfs处理方式是用Fh的结构体保存文件打开的状态和上下文信息，并不该Fh保存在Inode中，在后续读写等操作中依据该状态进行处理。

```
 // cephfs Inode Fh.flags(cephfsFh
struct Inode : RefCountedObject {
...
 std::set<Fh*> fhs;
}
// file handle for any open file state
struct Fh {
  InodeRef  inode;
  int     _ref;
  loff_t    pos;
  int       mds;        // have to talk to mds we opened with (for now)
  int       mode;       // the mode i opened the file with
  uint64_t  gen;

  int flags;
  bool pos_locked;           // pos is currently in use
  std::list<ceph::condition_variable*> pos_waiters;   // waiters for pos

  UserPerm actor_perms; // perms I opened the file with

  Readahead readahead;

  // file lock
  std::unique_ptr<ceph_lock_state_t> fcntl_locks;
  std::unique_ptr<ceph_lock_state_t> flock_locks;

  bool has_any_filelocks() {
    return
```

```
        (fcntl_locks && !fcntl_locks->empty()) ||
        (flock_locks && !flock_locks->empty());
    }

    // IO error encountered by any writeback on this Inode while
    // this Fh existed (i.e. an fsync on another Fh will still show
    // up as an async_err here because it could have been the same
    // bytes we wrote via this Fh).
    int async_err = {0};

    int take_async_err()
    {
        int e = async_err;
        async_err = 0;
        return e;
    }

    Fh() = delete;
    Fh(InodeRef in, int flags, int cmode, uint64_t gen, const UserPerm &perms);
    ~Fh();
    void get() { ++_ref; }
    int put() { return --_ref; }
  };
```

FastCFS处理方式是自定义FileInfo保存文件打开的状态信息，在create()、open()、opendir()操作时填充进 fuse_file_info结构中，在后续操作中直接使用：

```
struct fuse_file_info {
 /** Open flags.  Available in open() and release() */
 int flags;

 /** In case of a write operation indicates if this was caused
     by a delayed write from the page cache. If so, then the
     context's pid, uid, and gid fields will not be valid, and
     the *fh* value may not match the *fh* value that would
     have been sent with the corresponding individual write
     requests if write caching had been disabled. */
 unsigned int writepage : 1;

 /** Can be filled in by open, to use direct I/O on this file. */
 unsigned int direct_io : 1;

 /** Can be filled in by open. It signals the kernel that any
     currently cached file data (ie., data that the filesystem
     provided the last time the file was open) need not be
     invalidated. Has no effect when set in other contexts (in
     particular it does nothing when set by opendir()). */
 unsigned int keep_cache : 1;

 /** Indicates a flush operation.  Set in flush operation, also
     maybe set in highlevel lock operation and lowlevel release
     operation. */
 unsigned int flush : 1;

 /** Can be filled in by open, to indicate that the file is not
     seekable. */
 unsigned int nonseekable : 1;
 /* Indicates that flock locks for this file should be
    released.  If set, lock_owner shall contain a valid value.
    May only be set in ->release(). */
 unsigned int flock_release : 1;

 /** Can be filled in by opendir. It signals the kernel to
     enable caching of entries returned by readdir().  Has no
     effect when set in other contexts (in particular it does
     nothing when set by open()). */
 unsigned int cache_readdir : 1;

 /** Padding.  Reserved for future use*/
 unsigned int padding : 25;
 unsigned int padding2 : 32;

 /** File handle id.  May be filled in by filesystem in create,
  * open, and opendir().  Available in most other file operations on the
  * same file handle. */
 uint64_t fh;

 /** Lock owner id.  Available in locking operations and flush */
 uint64_t lock_owner;

 /** Requested poll events.  Available in ->poll.  Only set on kernels
     which support it.  If unsupported, this field is set to zero. */
 uint32_t poll_events;
};

// fastcfs
typedef struct fcfs_api_file_info {
    FCFSAPIContext *ctx;
    int64_t tid;
    struct {
        FDIRClientSession flock;
        FCFSAPIOpendirSession *opendir;
    } sessions;
    FDIRDEntryInfo dentry;
    int flags;
    int magic;
    struct {
        int last_modified_time;
    } write_notify;
    int64_t offset;  //current offset
} FCFSAPIFileInfo;

static int do_open(fuse_req_t req, FDIRDEntryInfo *dentry,
        struct fuse_file_info *fi, const FCFSAPIFileContext *fctx)
{
    int result;
    FCFSAPIFileInfo *fh;
   ...
```

```
    fi->fh = (long)fh;
    return 0;
}
```

整体flags支持方案

目前倾向于使用类似fastcfs的方式，自定义结构FileHandle，在create()、open()、opendir()时将上下文信息保存到fuse_file_info中，在后续文件操作时判断相关flags进行具体操作。简单的FileHandle如下：

```
struct FileHandle {
    int flags;
 int mod;
 uint64_t pos;
 Inode *inode;
}
```

具体flag的实现方案

O_TRUNC

需要实现file_truncate接口，接口中对目标文件进行内容删除，length置0。

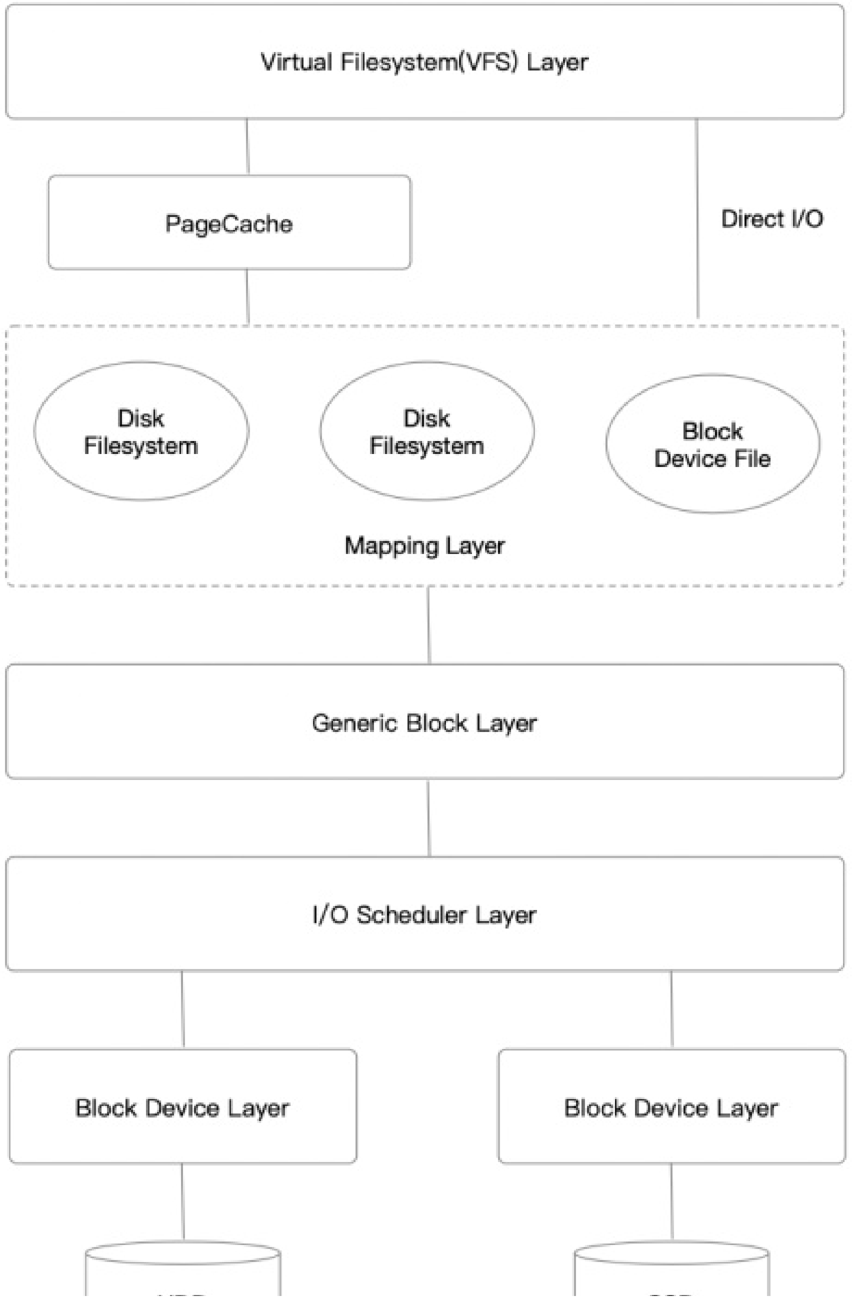
```
#
root@pubbeta1-nostest2:/tmp/fsmount# echo "1111111111111111" > f
root@pubbeta1-nostest2:/tmp/fsmount# cat f
1111111111111111
root@pubbeta1-nostest2:/tmp/fsmount# echo "aaa" > f
root@pubbeta1-nostest2:/tmp/fsmount# cat f
aaa
111111111111
# length=0
CURVEFS_ERROR FuseClient::FuseOpOpen(fuse_req_t req, fuse_ino_t ino,
          struct fuse_file_info *fi) {
...
if (fi->flags & O_TRUNC) {
        inode.set_length(0);
        ret = inodeManager_->UpdateInode(inode);
        if (ret != CURVEFS_ERROR::OK) {
            LOG(ERROR) << "inodeManager update inode fail, ret = " << ret
                       << ", inodeid = " << ino;
            return ret;
        }
    }
}

root@pubbeta1-nostest2:/tmp/fsmount# echo "1111111111111111" > f
root@pubbeta1-nostest2:/tmp/fsmount# cat f
1111111111111111
root@pubbeta1-nostest2:/tmp/fsmount# echo "aaa" > f
root@pubbeta1-nostest2:/tmp/fsmount# cat f
aaa
```

I/O模式类

O_DIRECT, O_SYNC, O_DSYNC, FASYNC, O_NONBLOCK(O_NDELAY), 这类flags应该是内核进行了支持，在用户态文件系统中进行相应设置，例如O_DIRECT有具体的文档描述说明了这一点，其他flag暂时查阅资料和代码还未发现正面说明。

O_DIRECT

一般来说，当调用 open() 系统调用打开文件时，如果不指定 O_DIRECT
标志，那么就是使用缓存I/O来对文件进行读写操作。系统缓存位于VFS和真实文件系统之间，当虚拟文件系统读文件时，首先从缓存中查找要读取的文件内容是否存在缓存中，如果存在就直接从缓存中读取。对文件进行写操作时也一样，首先写入到缓存中，然后由操作系统同步到块设备（如磁盘）中。对于通用块设备层来说要求io请求是块设备blocksize对齐的，对应buffered
io在pagecache层做了对齐，对应direct_io需要用户层来保证。


Virtual Filesystem(VFS) Layer
PageCache
Direct I/O
Disk Filesystem
Disk Filesystem
Block Device File
Mapping Layer
Generic Block Layer
I/O Scheduler Layer
Block Device Layer
Block Device Layer

实现：direct_io功能实现由VFS层提供，fuse也进行了支持，用户态文件系统要支持该flag需要在open中对flag进行解析，填充进fuse_file_info→direct_io，通过fuse_reply_open(req, fi)返回给内核处理。

```
// curvefs
void curve_ll_open(fuse_req_t req, fuse_ino_t ino, struct fuse_file_info *fi) {
    fi->direct_io = fi->flags & O_DIRECT
    fuse_reply_open(req, fi);
}

// libfuse
int fuse_reply_open(fuse_req_t req, const struct fuse_file_info *f)
{
 ...
 fill_open(&arg, f);
 return send_reply_ok(req, &arg, sizeof(arg));
}
static void fill_open(struct fuse_open_out *arg,
        const struct fuse_file_info *f)
{
 arg->fh = f->fh;
 if (f->direct_io)
  arg->open_flags |= FOPEN_DIRECT_IO;
 ...
}
```

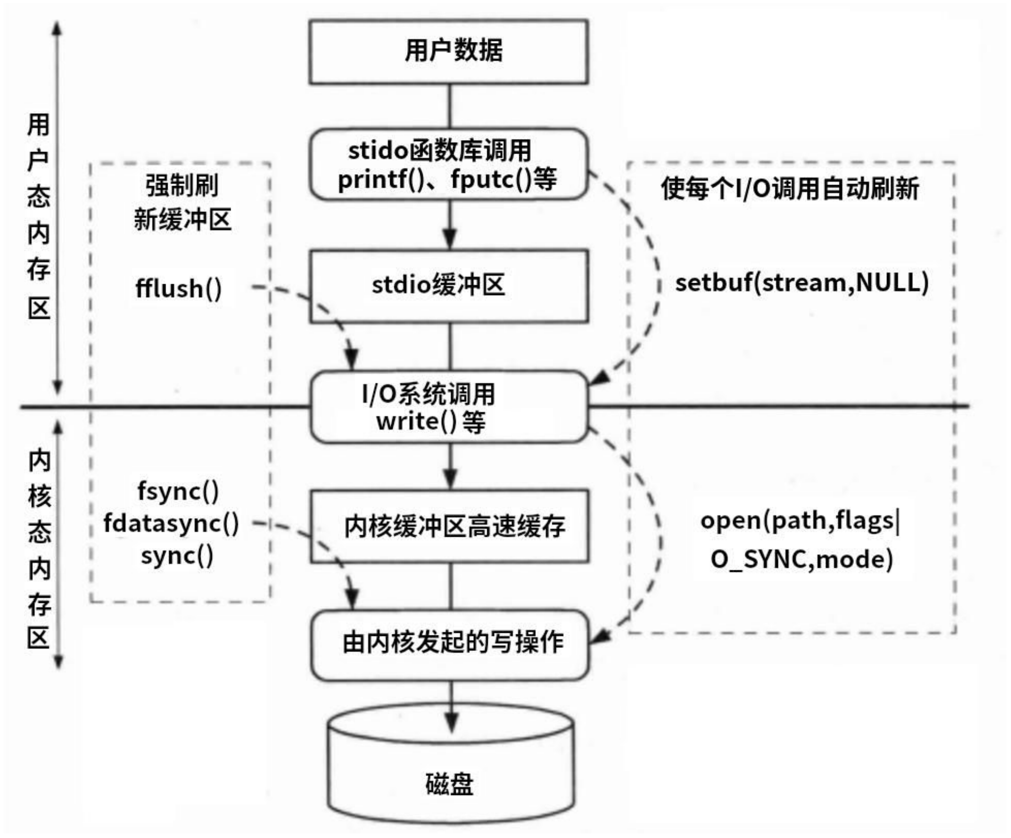
测试结果：基于curvefs测试结果不符合预期：在fuse_reply_open中设置fi→direct_io，从日志也看到设置成功，但是在不进行对齐的情况下本地文件系统会报错，但是curvefs没有报错，所以猜测要么没有真正启用成功，要么fuse做了处理。

分析结果：用户态文件系统在fuse_reply_open中设置fuse_file_info→direct_io后，内核感知该flag避免使用pagecahe，直接将数据在用户态和文件系统之间传递。从对本地文件系统和vfs分析发现，对齐的校验在VFS do_blockdev_direct_IO中实现，各文件系统根据自己direct_io的实现调用该函数，例如ext4就进行了调用，如果需要做对齐处理可能需要在用户态文件系统中做判断。

```
//
root@pubbeta1-nostest2:/tmp# strace ./main
...
open("in.txt", O_RDWR|O_CREAT|O_DIRECT, 0200) = 3
write(3, "\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0"..., 4095) = -1 EINVAL (Invalid
argument)

// curvefs
root@pubbeta1-nostest2:/tmp/fsmount# strace ./main
...
open("in.txt", O_RDWR|O_CREAT|O_DIRECT, 0200) = 3
write(3, "\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0"..., 4095) = 4095

// curvefsopenfidirect_iowritefi->direct_io=0, 16384=00040000
I 2021-08-09T15:02:50.754868+0800 870011 fuse_client.cpp:462] open fi->flags&O_DIRECT = 16384
I 2021-08-09T15:02:50.754873+0800 870011 fuse_client.cpp:466] open fi->fdirect_io = 1
I 2021-08-09T15:02:50.754935+0800 870010 fuse_client.cpp:303] write fi->flags&O_DIRECT = 16384
I 2021-08-09T15:02:50.754941+0800 870010 fuse_client.cpp:304] write fi->direct_io = 0
```

O_SYNC, O_DSYNC

同步I/O：强制刷新内核缓冲区到输出文件



对chubaofs和cephfs代码调研中发现在write中判断如果是直接IO则调用flush操作，但是对具体flush内容主要是对文件系统自己缓存的内容进行刷盘，没有发现对应内核缓冲区flush的相关设置或调用等。

```
 // chubaofs writeflush
func cfs_write(id C.int64_t, fd C.int, buf unsafe.Pointer, size C.size_t, off C.off_t) C.ssize_t {
 ...
 var wait bool

 if f.flags&uint32(C.O_DIRECT) != 0 || f.flags&uint32(C.O_SYNC) != 0 || f.flags&uint32(C.O_DSYNC) != 0 {
  wait = true
 }
 ...
 if wait {
  if err = c.flush(f); err != nil {
   return C.ssize_t(statusEIO)
  }
 }
}

// cephfs writeflush
int64_t Client::_write(Fh *f, int64_t offset, uint64_t size, const char *buf,
                 const struct iovec *iov, int iovcnt)
{
 ...
 if ((f->flags & O_SYNC) || (f->flags & O_DSYNC)) {
      _flush_range(in, offset, size);
    }
  } else {
    if (f->flags & O_DIRECT)
      _flush_range(in, offset, size);
   ...
  }
}
```

O_NONBLOCK(O_NDELAY), FASYNC, O_TMPFILE

查看的几个分布式系统都没有进行实现包括cephfs、chubaofs、moosefs、fastcfs。具体实现后续可以再深入看看。

结论

1，需要实现file_truncate接口来支持O_TRUNC flag（优先级高）。

2，待curvefs支持链接和支持对inode中atime、ctime、mtime的修改后，对O_NOFOLLOW和O_NOATIME进行测试（优先级中）。

3，目前I/O模式类flag（O_SYNC, O_DSYNC, O_NONBLOCK）经测试不会“影响“结果的正确性，后面继续对其实现方式进行研究（优先级中）。

参考文献

https://man7.org/linux/man-pages/man2/open.2.html

https://www.cnblogs.com/BinBinStory/p/7400993.html

https://juejin.cn/post/6844903923048792078

https://www.gnu.org/software/libc/manual/html_node/File-Status-Flags.html

https://android.googlesource.com/platform/prebuilts/gcc/linux-x86/host/x86_64-linux-glibc2.7-4.6/+/refs/heads/jb-dev/sysroot/usr/include/asm-generic/fcntl.h

https://www.gnu.org/software/libc/manual/html_node/Permission-Bits.html

https://xinqiu.gitbooks.io/linux-insides-cn/content/SysCall/linux-syscall-5.html

https://zhuanlan.zhihu.com/p/330515575

https://www.kernel.org/doc/html/latest/filesystems/fuse-io.html

https://stackoverflow.com/questions/27087912/write-back-vs-write-through-caching