# 文件系统 FAQ

发布版本：1.0

作者邮箱：cmc@rock-chips.com

日期：2018.07

文件密级：公开资料

---

**前言**

**概述**

**产品版本**

| 芯片名称 | 内核版本 |
| --- | --- |
| 全系列 | 通用 |

**读者对象**

本文档（本指南）主要适用于以下工程师：

技术支持工程师

软件开发工程师

**修订记录**

| 日期 | 版本 | 作者 | 修改说明 |
| --- | --- | --- | --- |
| 2018-07-30 | V1.0 | 陈谋春 | 初始版本 |
| 2019-04-23 | V1.1 | 陈谋春 | 增加性能测试和 IO 高性能编程 |

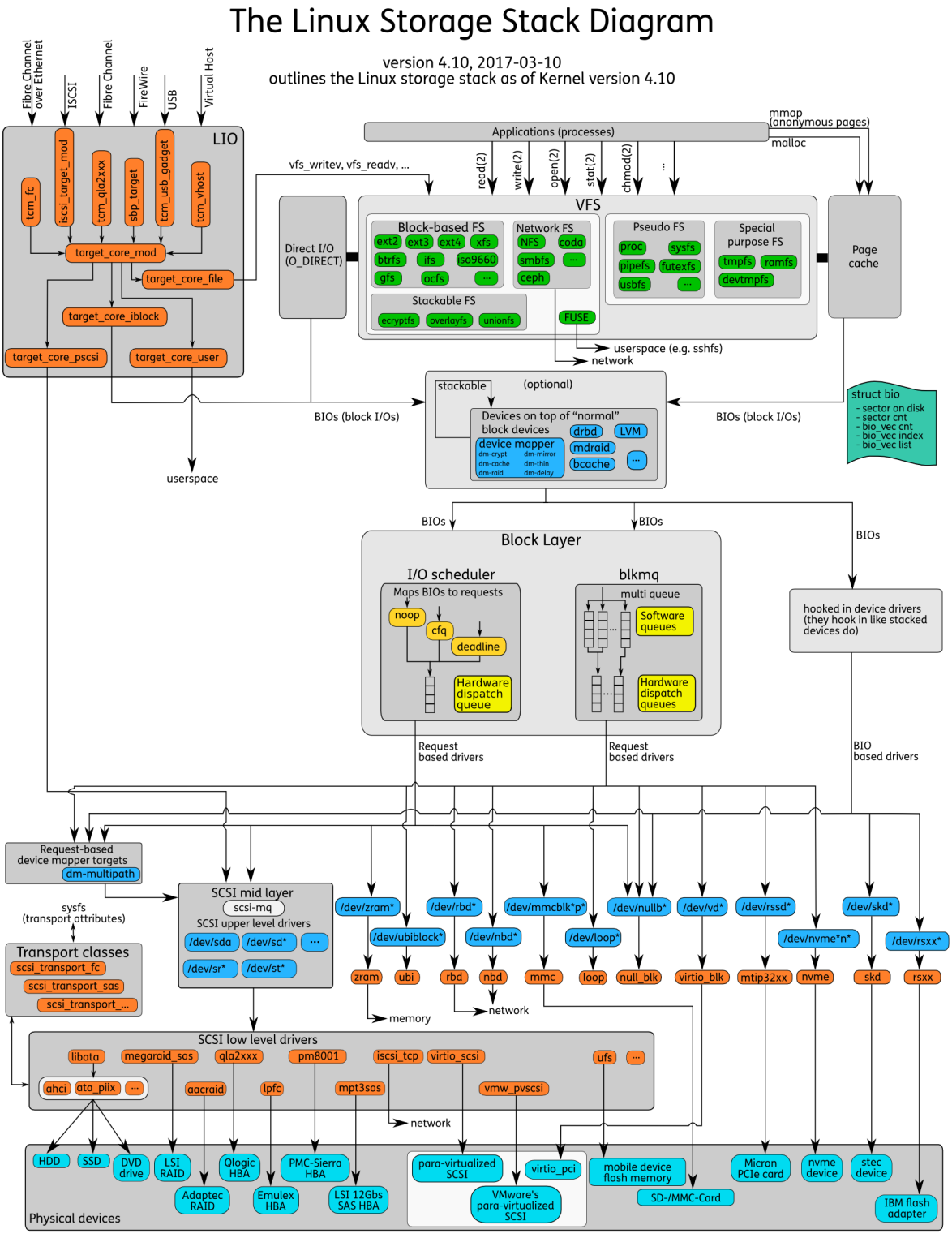
---



---

# 1 Linux Storage Stack

下图是一张 Linux 的存储栈的图解，通过它我们可以对 Linux 的存储子系统有个大致的了解：




我们从用户态发起一个系统调用，一般会进过这样的流程：VFS -> FS(ext4/f2fs) -> Block Layer -> Physical Devices。

# 2 系统调用与 C 库函数

fopen 和 open，read 和 fread，write 和 fwrite 有什么区别，很多人都会弄混了，而这经常会带来一些问题。所以在这里理清他们的关系是很有必要的。

如上图所示，open/read/write 是 Linux 提供的系统调用，用户态的程序只能通过这些接口来访问文件系统层。而 fopen/fread/fwrite 是 C 库提供的文件读写接口，其核心实现是基于 open/read/write 这一类的系统调用。说到这有些人可能会说，那我在写代码的时候应该用哪一套？其实都是可以的，主要看你的需求，C 库函数的目的是为了方便编程，所以 C 库的文件接口提供了一些额外的处理，例如字符串和文本的处理，比如 fpus/fgets，所以如果你在做文本数据的输入输出，无疑 C 库会是更好的选择。

==Note: 绝大部分的 C 库都为文件接口提供一层缓存，所以你调用 fwrite 操作的时候，实际上数据是先放到这一层缓存的，在编程的时候必须注意，在下一节会重点说明。==

# 3 Linux 数据回写

这一部分是问的最多的问题，很多对刚接触文件接口（包括前面的系统调用和 C 库函数）的人都会觉得很奇怪：为什么我 fwrite/write 函数已经返回了，此时掉电或重启后数据会丢失？问题的根源在于缓存的存在，由于存储设备属于低速设备，直接操作的话会有严重的延迟，所以通常会在 DRAM 上先缓存一部分数据。而 DRAM 是易失性存储设备，掉电数据就丢了，所以要确保数据固化就要把数据回写到存储设备上。

数据回写有两种方式：1.主动同步回写；2.异步后台回写；在了解数据如何回写前，有一点必须要注意，你的缓存有可能有好多层，所以你必须从上到下把每一层的缓存都刷出去（即写到下一层）。这一点我们下面会举几个具体的例子，先介绍怎么主动同步回写。

先来看一个系统调用的例子：

```
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

int fd;
ssize_t wr;
char szBuf[] = "hello world";

fd = open("/sdcard/test.txt", O_CREAT|O_WRONLY|O_TRUNC, S_IRWXU | S_IRWXG);
wr = write(fd, szBuf, strlen(szBuf));
close(fd);
// 在这里掉电，test.txt将会是空文件，close并不能保证数据回写
```

write 只是把数据提交到内核的 Page Cache（假设没有启用 DIO)，要想写到存储设备，必须通过 fdatasync 或 fsync 系统调用。下面看一下修正后的代码：

```
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

int fd;
ssize_t wr;
char szBuf[] = "hello world";

fd = open("/sdcard/test.txt", O_CREAT|O_WRONLY|O_TRUNC, S_IRWXU | S_IRWXG);
wr = write(fd, szBuf, strlen(szBuf));
```

```
#if 0
fdatasync(fd);  // 只回写数据
#else
fsync(fd);  // 回写数据和元数据（文件大小、最后修改时间等）
#endif
close(fd);
// 这里掉电是安全的
```

看完系统调用的例子，再来看一下 C 库的例子：

```
#include <stdio.h>

FILE *fp = null;

fp = fopen("/sdcard/test.txt", "w+");
fputs("hello world", fp);
fclose(fp);
// 在这里掉电，test.txt将会是空文件
```

我们知道 C 库也有一层文件缓存，所以掉电是因为数据还在这一层缓存中，熟悉 C 库的同学可能知道这里要加一个 fflush，其实并不对，fflush 只能保证把缓存回写到下一层，即内核的 Page Cache，依然需要调用 fsync 再刷到物理存储设备。正确写法如下：

```
#include <stdio.h>

FILE *fp = null;
int fd;

fp = fopen("/sdcard/test.txt", "w+");
fputs("hello world", fp);
fflush(fp); // c库缓存写回到内核page cache
fd = fileno(fp);
fsync(fd); // page cache回写到物理设备
fclose(fp);
// 这里掉电数据是安全的
```

最后再来看看 Java 的例子，下面是一个最常见的 Java 写文件 Demo：

```
public static void writeFile(String filePath, String conent) {
        BufferedWriter out = null;
        try {
            out = new BufferedWriter(new OutputStreamWriter(new
FileOutputStream(filePath, true)));
            out.write(conent);
            // 这里掉电会丢数据
        } catch (Exception e) {
            e.printStackTrace();
        } finally {
            try {
                if (out != null) {
                    out.close();
                }
            } catch (IOException e) {
                e.printStackTrace();
            }
```

```
        }
    }
```

这个例子仍然是缓存没处理好，BufferedWriter 从名字来看就知道是有缓存的，通过手册可以看到调用 flush 可以写回，那是不是加上这个就够了？答案是否定的，这个函数也只能做到把缓存写到 Page Cache，同样还要想办法触发 fsync 系统调用，正确的写法如下：

```
public static void writeFile(String filePath, String conent) {
        BufferedWriter out = null;
        try {
            FileOutputStream fos = new FileOutputStream(filePath, true);
            out = new BufferedWriter(new OutputStreamWriter(fos));
            out.write(conent);
            out.flush();   // 数据写到page cache
            FileUtils.sync(fos); // 数据写到物理设备，方法1
            //FileDescriptor fd = fos.getFD();
            //fd.sync();  // 数据写到物理设备，方法2
            // 这里掉电是安全的
        } catch (Exception e) {
            e.printStackTrace();
        } finally {
            try {
                if (out != null) {
                    out.close();
                }
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
    }
```

总结，如果你的程序需要确保数据立即写回，就需要考虑每一层的缓存，确保每一层都按顺序写回。讲到这里，那有些人就会有疑问，如果我不需要数据马上写回到物理设备，但是我需要知道数据什么时候会写回，这就涉及到我前面说的数据异步写回了。

Linux 内核会定时触发，把==已经提交到 Page Cache 的脏数据==[1]写回到物理设备，需要特别注意如果你的数据还在上层的缓存中，例如 Java 或 C 库的缓存中没有刷下来，那自然不会被内核的异步回写机制写回。大致的异步回写机制步骤如下：

- step 1: 内核按 dirty_writeback_centisecs 的时间间隔唤醒回写线程
- step 2: 回写线程会遍历 Page Cache 寻找那些被标记为脏的时间超过 dirty_expire_centisecs 的页面，并全部回写
- step 3: 回写线程接着会判断脏数据总量是否超过 dirty_background_ratio（单位是百分比）或 dirty_background_bytes，如果超过则回写所有脏数据
- step 4: 回写线程等待下次唤醒周期

  需要注意，在脏数据超过 dirty_ratio 和 dirty_bytes 以后如果继续写数据会自动触发同步回写，即这一次的 write 会把之前和本次的数据都写回到物理设备再返回。

  最后再列一下 Linux 内核回写机制的几个可调阈值，它们都位于/proc/sys/vm 目录：
- dirty_writeback_centisecs: 控制内核回写线程的唤醒周期，单位是 10ms，默认值是 500，即 5s 唤醒一次

- dirty_expire_centisecs：脏数据的过期时间，单位是 10ms，默认值是 3000，从 Page 被标记为脏算起，超过这个时间会被认为是过期数据，所以默认情况下，脏数据量没有超过阈值时，数据要等 30s 以上才会回写，实际上要考虑唤醒周期的影响，数据最长要等 35s 才会写回到存储设备
- dirty_background_ratio & dirty_background_bytes：二者功能一样，前者是百分比（==这里的基数不是总内存大小，而是可用内存大小，包括可回收的内存==），后者单位是字节，脏数据总量要超过这个阈值才会全部回写，否则只会写过期数据。这两个是互斥的，写其中一个，另一个会自动被清 0
- dirty_ratio & dirty_bytes：二者功能一样，前者是百分比（这里的基数不是总内存大小，而是可用内存大小，包括可回收的内存），后者单位是字节，脏数据量超过这个阈值以后会阻塞 write，确保数据同步写回到存储设备

  以上配置参数都可以根据自己的实际需求来调整，例如对于那些视频监控的产品，默认的配置可能会导致数据累积到非常多才会回写，给存储设备的压力就非常大了，这时候可以减小 dirty_background_ratio，dirty_writebac-k_centisecs，让数据写入更平滑。但是需要注意，考虑到掉电数据安全，这两个值不能无限减小，太小会导致后台一直在做数据回写，无形中增加了掉电丢数据的风险。最后再强调这些改动是全局的，即对所有的存储设备都生效，所以一定要慎重。

  在 Linux 下可以通过 bashrc 中加入 echo 命令来实现，而 Android 可以再 init.rc 脚本里搜一下其他写 proc 目录节点的位置，在附近加一下你的 vm 参数的调整，例如：

```
write /proc/sys/vm/dirty_background_ratio 5
```

# 4 Linux 数据预读

在 Linux 系统中，默认情况下不管是用户态调用 read，还是内核态调用 vfs_read，都会触发数据预读，即都会多度一部分数据到 Page Cache 中，这在顺序读的场景下对性能的提升是非常明显的。而高性能的存储设备可以通过加大预读窗口大小来大幅提升顺序读性能。在 Android 平台有两个方法来修改预读窗口大小：全局控制和文件单独控制。

全局控制是以存储设备为单位的，建议低速设备用 128KB，高速可以用 2048KB，例如：

```
write /sys/block/mmcblk0/queue/read_ahead_kb 2048
write /sys/block/dm-0/queue/read_ahead_kb 2048
```

==Note: 现在很多设备都开启了 verity 和 encrypt，最终物理存储设备会被映射成 dm-x 这样的逻辑设备，这些逻辑设备的窗口也要一起改掉才会生效==

文件单独控制是以文件为单位的，我们可以通过系统调用给文件一个调整预读窗口的 Hint，而不是直接设置预读大小，例子如下：

```
#include <fcntl.h>

posix_fadvise(fd, start, len, POSIX_FADV_SEQUENTIAL);  // 暗示内核，上层会在start开始的len范围内顺序访问，内核会在这个范围内加大预读窗口
posix_fadvise(fd, start, len, POSIX_FADV_RANDOM); // 暗示内核，上层会在start开始的len范围内随机访问，内核会在这个范围内禁止预读
posix_fadvise(fd, start, len, POSIX_FADV_NOREUSE); // 暗示内核，上层在指定范围内只会访问一次
posix_fadvise(fd, start, len, POSIX_FADV_WILLNEED); // 暗示内核，上层短时间内会访问这个范围的数据
posix_fadvise(fd, start, len, POSIX_FADV_DONTNEED); // 暗示内核，上层短时间内不会访问某段数据
```

# 5 文件掉电保护

每个文件系统都有一套自己的掉电保护机制，但是这个机制只保证文件系统本身的完整性，而无法保证文件的完整性。在本节开始前，有必要对这两者做一下解释：

- 文件系统的完整性：文件系统可正常挂载，所有的文件和目录都可正常访问，可以正常完成所有已实现的文件操作
- 文件的完整性：文件可正常读写，并且功能正常，例如媒体文件要能正常播放，XML 文件要能正常解析，压缩文件要能正常解压

  举一个具体的例子会更直观一些，假设有一个应用程序在写一个 XML 文件来保存一些配置信息，而就在写数据的过程中出现掉电，代码如下：

```
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

int fd, len;
ssize_t wr;
char *szBuf;

// 假设sdcard分区上已经有了A和B两个文件，并且已经写回到物理设备
fd = open("/sdcard/test.xml", O_CREAT|O_WRONLY|O_TRUNC, S_IRWXU | S_IRWXG);
{
    // 在这个区间发生随机掉电
    wr = write(fd, szBuf, 8192);
    fsync(fd);
    close(fd);
}
```

文件系统的完整性可以保证，在重新上电后 A 和 B 可以正常访问，并且数据是完整的，而 test.xml 这个文件可能存在也可能不存在（不启用 dirsync 的情况下）。如果存在的话其数据读写是正常的，但是不能保证数据是完整的，即其大小有可能是[0-8192]的任意值，应用程序很可能无法解析这个 XML。

导致这个问题的原因是 write 和 fsync 这些函数不是原子的，实际上大部分文件系统的系统调用都无法保证原子性，所以文件的完整性需要应用自己来保证。例如 Android 实现的 AtomicFile，其原理是利用 rename 函数的原子性来解决问题的，关键函数如下：

```
public FileOutputStream startWrite() throws IOException {
        // Rename the current file so it may be used as a backup during the next
read
        if (mBaseName.exists()) {
            if (!mBackupName.exists()) {
                if (!mBaseName.renameTo(mBackupName)) {
                    Log.w("AtomicFile", "Couldn't rename file " + mBaseName
                            + " to backup file " + mBackupName);
                }
            } else {
                mBaseName.delete();
            }
        }
        FileOutputStream str = null;
        try {
            str = new FileOutputStream(mBaseName);
        } catch (FileNotFoundException e) {
```

```
            File parent = mBaseName.getParentFile();
            if (!parent.mkdirs()) {
                throw new IOException("Couldn't create directory " + mBaseName);
            }
            FileUtils.setPermissions(
                parent.getPath(),
                FileUtils.S_IRWXU|FileUtils.S_IRWXG|FileUtils.S_IXOTH,
                -1, -1);
            try {
                str = new FileOutputStream(mBaseName);
            } catch (FileNotFoundException e2) {
                throw new IOException("Couldn't create " + mBaseName);
            }
        }
        return str;
    }

 public void finishWrite(FileOutputStream str) {
        if (str != null) {
            FileUtils.sync(str);
            try {
                str.close();
                mBackupName.delete();
            } catch (IOException e) {
                Log.w("AtomicFile", "finishWrite: Got exception:", e);
            }
        }
    }

public FileInputStream openRead() throws FileNotFoundException {
        if (mBackupName.exists()) {
            mBaseName.delete();
            mBackupName.renameTo(mBaseName);
        }
        return new FileInputStream(mBaseName);
    }
```

流程很清晰，写数据前先把文件改名成备份文件，再创建一个新文件来写，写完成以后把备份文件删除，读文件前先判断备份文件是否存在，有存在说明发生掉电，则用备份文件来恢复。有兴趣读完整代码可以参考AtomicFile.java。下面是一段 AtomicFile 的使用 Demo：

```
 public static void write(AtomicFile file, IntervalStats stats) throws
IOException {
        FileOutputStream fos = file.startWrite();
        try {
            // startWrite和finishWrite中间的写操作可以确保原子性
            write(fos, stats);
            file.finishWrite(fos);
            // 假设掉电位置
            fos = null;
        } finally {
            // When fos is null (successful write), this will no-op
            file.failWrite(fos);
        }
    }
```

C 语言也可以参考这里例子，这里就不详解了。在这里需要强调一点，android 的这个 AtomicFile 不能保证 finishWrite 操作完成后掉电，再开机数据一定是全部写完后的，以上面的例子来讲：在假设掉电位置掉电，则开机后这个文件的 stats 还是可能完全没写入。原因是 finishWrite 中的删除备份文件操作并不能保证写回磁盘，导致重新开机后进 openRead 操作会发现备份文件还在，就会把实际已经写好的新文件删掉，重新用备份文件来覆盖。所以说==android 的 AtomicFile 只保证这个文件 startWrite 和 finishWrite 之间的写数据原子性，即要么全部完成，要么全部失败==。

对于 android 来说，AtomicFile 符合它的设计目标，但是如果有时候你确实想要在 finishWrite 完成后掉电能看到新数据，其实也是有办法的，目前主要有两种方式：在文件系统挂载的时候加上 MS_DIRSYNC 参数，通过 ioctl 命令配置文件的父目录位 DIRSYNC 模式。前者更方便，缺点是这个挂载点下的所有目录都会进 DIRSYNC 模式，在文件创建和删除非常频繁的情况下，会有明显的性能损失；后者通过精确控制需要保护的目录来减少性能损失，但是缺点是要非常清楚哪些目录需要保护，并且不同文件系统的命令和标志都不一样，需要把你可能用到的所有文件系统的 ioctl 都写一遍，并且有些文件系统可能不支持这种方式(f2fs 和 ext4 都是两种方式都支持，而 fat 则不支持 ioctl 方式)，所以这里就不给出这个方式的代码了。

默认情况下 android 会把所有外部存储都默认配上 MS_DIRSYNC 选项，所以如果你的数据是放在外部存储，不需要任何修改。而内部存储则没有加这个选项，毕竟对于 android 来说绝大部分都是带电池的设备，内部存储基本不需要为掉电烦恼，性能才是第一优先（google 的亲儿子甚至加了 fsync_mode=nobarrier）。要为内部存储加这个选项也很简单，按如下修改方式即可：

```
# 打到system/core目录下
diff --git a/fs_mgr/fs_mgr_fstab.c b/fs_mgr/fs_mgr_fstab.c
index 9225d34..0785ace 100644
--- a/fs_mgr/fs_mgr_fstab.c
+++ b/fs_mgr/fs_mgr_fstab.c
@@ -56,6 +56,7 @@
     { "slave",      MS_SLAVE },
     { "shared",     MS_SHARED },
     { "defaults",   0 },
+    { "dirsync",    MS_DIRSYNC},
     { 0,            0 },
 };

# 修改device目录下，你的设备用的fstab文件，在data目录的最后一列加上dirsync选项
--- a/fstab.rk30board.bootmode.forceencrypt.emmc
+++ b/fstab.rk30board.bootmode.forceencrypt.emmc
@@ -8,7 +8,7 @@
 #/dev/block/platform/fe330000.sdhci/by-name/system         /system
ext4      ro,noatime,nodiratime,noauto_da_alloc
wait,check,verify
 /dev/block/platform/fe330000.sdhci/by-name/cache          /cache
ext4      noatime,nodiratime,nosuid,nodev,noauto_da_alloc,discard
wait,check
 /dev/block/platform/fe330000.sdhci/by-name/metadata       /metadata
ext4      noatime,nodiratime,nosuid,nodev,noauto_da_alloc,discard
wait,check
-/dev/block/platform/fe330000.sdhci/by-name/userdata       /data
f2fs      noatime,nodiratime,nosuid,nodev,discard,inline_xattr
wait,check,notrim,forceencrypt=/metadata/key_file
+/dev/block/platform/fe330000.sdhci/by-name/userdata       /data
f2fs      noatime,nodiratime,nosuid,nodev,discard,inline_xattr,dirsync
        wait,check,notrim,forceencrypt=/metadata/key_file
 #data for f2fs nobarrier
```

```
 #/dev/block/platform/fe330000.sdhci/by-name/userdata       /data
f2fs      noatime,nodiratime,nosuid,nodev,discard,inline_xattr,nobarrier
wait,check,notrim,forceencrypt=/metadata/key_file
```

# 6 性能测试

常见的文件系统测试有很多，这里介绍最简单的两种：dd 和 iozone，前者用于顺序性能测试，后者更全面一些，包括了一些随机性能测试。

几乎所有的 linux 系统都会包含 dd 命令，用它来测顺序读写非常方便，但是要注意清 cache 和回写，下面是一个简单的例子，可以根据需要自己修改：

```
#!/system/bin/sh

# 删除上一次测试数据，文件路径指向你要测试的文件系统下任意目录，sync是为了触发回写，避免测试过程中回写影响性能一致性
rm -f /data/local/2g
sync
# 在启用discard的文件系统下，删除会触发discard，休眠是为了避免discard对性能的影响
sleep 30s
# 写性能测试：写的总数据量不能太小，不然都会在内存中，不会触发回写
busybox dd if=/dev/zero of=/data/local/2g bs=4K count=512K
# 触发回写，清cache，防止影响后续测试，在对比写性能的时候不要忽略这次sync的时间，因为有可能vm回写参数配置有差异（前面有介绍），导致这里sync的数据量有差异，下面命令可以打出sync的数据量=Dirty+Writeback
cat /proc/meminfo | grep 'Dirty' -A 1
time sync
echo 3 > /proc/sys/vm/drop_caches
# 读性能测试：
busybox dd if=/data/local/2g of=/dev/null bs=4K count=512K
```

在排查性能问题的时候，还可以用 dd 测试存储设备本身的性能，正常情况下，二者的性能差距很小，如果文件系统的测试结果比存储节点明显慢，那就要查一下文件系统。测试的时候只要把文件路径，换成存储设备的节点就可以了，下面是例子：

```
#!/system/bin/sh

sync
echo 3 > /proc/sys/vm/drop_caches

# 先通过df命令，找到文件系统对应的设备节点，这里假设节点是/dev/block/dm-0
# 读性能测试：
busybox dd if=/dev/block/dm-0 of=/dev/null bs=4K count=512K
# 写性能测试：注意，这个测试会破坏文件系统，要慎重
busybox dd if=/dev/zero of=/dev/block/dm-0 bs=4K count=512K
# 触发回写，清cache，防止影响后续测试
time sync
```

iozone 是开源的文件系统性能测试套件，支持 direct 和 buffer 两套接口，移植到 android 也很简单，可以自己移植，也可以参考开源工程。下面给出几个常规的测试组合，如果想进一步了解 iozone，可以通过'-h'选项看具体帮助，也可以上网搜索，资料还是非常多的。先看看常见的选项：

```
 -a  Auto mode
 -f filename  to use
 -b Filename  Create Excel worksheet file
 -I  Use VxFS VX_DIRECT, O_DIRECT,or O_DIRECTIO for all file operations
 -L #  Set processor cache line size to value (in bytes)
 -q #  Set maximum record size (in Kbytes) for auto mode (or #m or #g)
 -R  Generate Excel report
 -s #  file size in Kb
              or -s #k .. size in Kb
              or -s #m .. size in Mb
              or -s #g .. size in Gb
 -S #  Set processor cache size to value (in Kbytes)
```

我们前面论述的都是常规的文件接口，一般称为 buffer io，下面是一个简单的测试命令：

```
# adb push iozone /data/local/
rk3399:/ # cd /data/local/
rk3399:/data/local # ./iozone -a -s 1g -q 256 -S 512 -L 64 -f /data/iozone.dat -
R -b ./iozone.xls
```

另外一种文件接口叫做 direct io，顾名思义是直接绕过内核的 page cache，直接读写存储设备，一般用在应用程序自己管理文件的缓存，自己控制回写时间的场景，比如数据库，具体测试命令可以参考如下：

```
rk3399:/data/local # ./iozone -a -I -s 1g -q 256 -S 512 -L 64 -f
/data/iozone.dat -R -b ./iozone.xls
```

最后，新版本的 android 还提供了 fio 工具，这也是一个开源的文件系统 benchmark，一般底层文件系统和存储驱动开发会更关心一些，有兴趣可以看一下 external/fio/HOWTO 的介绍，和 external/fio/examples 下的例子，默认情况下 fio 是没有编译的，下面是一个简单的多线程并发例子：

先修改一下测试用例 external/fio/examples/tiobench-example.fio：

```
[global]
direct=1     ; 启用direct io
size=64m     ; 文件大小
bsrange=4k-4k  ; 每次读取的数据块大小范围
timeout=60   ; 超时时间
numjobs=4    ; 每个job并发4个线程，下面定义了4个job，总共16个线程

[f1]
rw=write

[f2]
stonewall
rw=randwrite

[f3]
stonewall
rw=read

[f4]
stonewall
rw=randread
```

现在可以编译 fio，开始测试：

```
cmc@cmc-B15O-D3A:~/workspace/rockchip/rk_3399_8.1$ mmm external/fio/
cmc@cmc-B15O-D3A:~/workspace/rockchip/rk_3399_8.1$ adb root
cmc@cmc-B15O-D3A:~/workspace/rockchip/rk_3399_8.1$ adb push
out/target/product/rk3399/system/bin/fio /data/local/
cmc@cmc-B15O-D3A:~/workspace/rockchip/rk_3399_8.1$ adb push
external/fio/examples/tiobench-example.fio /data/local/
rk3399:/ # cd /data/local/
rk3399:/data/local # ./fio tiobench-example.fio
```

# 7 IO 高性能编程

## 7.1 direct io

前面说过这个接口适用于应用程序要自己控制缓存和回写的场景，操作和普通的 buffer io 差异不大，在文件打开的时候传 O_DIRECT 标志即可，但是需要注意的是：direct io 模式下要求 write 的 buffer 和 count 都必须 block 对齐，这里的 block 大小可以通过 sys 文件系统查询到，一般为 512。下面是一个 demo：

```
#include <string.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/time.h>
#include <fcntl.h>
#include <unistd.h>
#include <iostream>
#include <string>
#include <errno.h>
#include <sys/wait.h>

// buf和bs都要求block对齐
bool writeFileInODirectMode(char *fileName, unsigned char *buf, int bs)
{
    int fd;
    int ret;
    int time_ms, avg, min, max;
    struct timeval t1, t2;
    int write_count = FILE_SIZE / bs;
    int interval = (STREAM_TIME * 1000)/write_count;

    printf("interval=%d, count=%d\n", interval, write_count);
    // 这里需要传入O_DIRECT标志才能进入direct io模式
    fd = open(fileName, O_DIRECT | O_RDWR, S_IRWXU);
    if (fd < 0){
        printf("open %s failed", fileName);
        free(buf);
        return false;
    }

    avg = min = max = 0;
    for (int i = 0; i < write_count; i++) {
        gettimeofday(&t1, NULL);
```

```
        ret = write(fd, buf, bs);
        if (ret < 0) {
            perror("write ./direct_io.data failed");
            return false;
        }
        gettimeofday(&t2, NULL);
        time_ms = (t2.tv_sec - t1.tv_sec) * 1000 + (t2.tv_usec - t1.tv_usec) /
1000;
        if (time_ms >= interval) {
            struct timespec ts;
            int time_s;
            clock_gettime(CLOCK_MONOTONIC, &ts);
            time_s = ts.tv_sec + ts.tv_nsec/1000000000;
            printf("%ds: direct write %dk times=%dms\n", time_s, bs>>10,
time_ms);
        }
        else {
            int time_us = (interval - time_ms) * 1000;
            usleep(time_us);
        }

        min = min < time_ms ? min : time_ms;
        max = max > time_ms ? max : time_ms;
        avg += time_ms;
    }
    close(fd);
    printf("min=%dms, max=%dms, avg=%dms\n", min, max, avg/WRITE_COUNT);
    return true;
}

int main(int argc, char * argv[])
{
    unsigned char *buf_a, *buf_b;
    int i = 0, j=0;
    int bs = BUF_SIZE;
    int test_case = 0;
    pid_t pid;
    int status;
    char *dev_path;

    if (argc != 4) {
    printf("usage: sd_test test_case block_size dev_path\n");
    exit(1);
    }

    test_case = atoi(argv[1]);
    // block大小，这里是通过参数传入，实际场景可以通过读sys下的节点来得到，
    // 例如：cat /sys/block/mmcblk1/queue/physical_block_size，其中mmcblk1即文件系
统所在设备
    bs = atoi(argv[2])*1024;
    dev_path = argv[3];
    printf("test_case=%d,bs=%d,dev_path=%s\n", test_case,bs,dev_path);

    // 不能通过malloc分配buffer，要通过下面的接口分配来确保block对齐
    int ret = posix_memalign((void **)&buf_a, 512, bs);
    if (ret) {
        perror("posix_memalign failed");
        exit(1);
```

```
    }
    memset(buf_a, 'c', bs);

    ret = posix_memalign((void **)&buf_b, 512, bs);
    if (ret) {
        perror("posix_memalign failed");
        exit(1);
    }
    memset(buf_b, 'c', bs);

    if (test_case == 0) {
    system("dd if=/dev/zero of=./direct_a bs=65536 count=2048");
    system("dd if=/dev/zero of=./direct_b bs=65536 count=2048");
    system("echo 3 > /proc/sys/vm/drop_caches");
    pid = fork();
    if (pid > 0) {
      while (1) {
          printf("a run %d ", ++i);
          if(!writeFileInODirectMode("direct_a", buf_a, bs))
          {
              exit(1);
          }
      }
    }
     else {
        while (1) {
        printf("b run %d ", ++j);
        if(!writeFileInODirectMode("direct_b", buf_b, bs))
        {
            exit(1);
        }
    }
    }
    }
    else if (test_case == 1) {
        if(!seqWriteDev(dev_path, buf_a, bs))
        {
        exit(1);
        }
    }
    wait(&status);
    free(buf_a);
    free(buf_b);
}
```

## 7.2 async io

async io 即异步 io，后面都简称 aio，实际上对于内核来说，同步 io 的本质实现也是 aio+blocking，即内核阻塞调用者直到 aio 完成通知到来。目前 linux 平台下有两套 aio：glibc 和 linux native，前者更简洁易懂，但在 android 平台上不支持；后者会更通用更节省 cpu，一般还通过 libaio 来辅助。

### 7.2.1 glibc aio

先来看一下 glibc 的 aio 接口：

```
int aio_read(struct aiocb *aiocbp);  /* 提交一个异步读 */
int aio_write(struct aiocb *aiocbp); /* 提交一个异步写 */
int aio_cancel(int fildes, struct aiocb *aiocbp); /* 取消一个异步请求（或基于一个fd的所有异步请求，aiocbp==NULL） */
int aio_error(const struct aiocb *aiocbp);        /* 查看一个异步请求的状态（进行中EINPROGRESS？还是已经结束或出错？） */
ssize_t aio_return(struct aiocb *aiocbp);         /* 查看一个异步请求的返回值（跟同步读写定义的一样） */
int aio_suspend(const struct aiocb * const list[], int nent, const struct timespec *timeout); /* 阻塞等待请求完成 */
```

下面是 glibc 的 aio 读的最简单例子，写操作也可以参考：

```
#include<stdio.h>
#include<sys/types.h>
#include<sys/stat.h>
#include<errno.h>
#include<fcntl.h>
#include<aio.h>
#include<stdlib.h>
#include<strings.h>

#define BUFSIZE   256

int main()
{
    struct aiocb    cbp;
    int             fd,ret;
    int             i = 0;

    fd = open("test.txt",O_RDONLY);

    if(fd < 0)
    {
        perror("open error\n");
    }

    //填充struct aiocb 结构体
    bzero(&cbp,sizeof(cbp));
    //指定缓冲区
    cbp.aio_buf = (volatile void*)malloc(BUFSIZE+1);
    //请求读取的字节数
    cbp.aio_nbytes = BUFSIZE;
    //文件偏移
    cbp.aio_offset = 0;
    //读取的文件描述符
    cbp.aio_fildes = fd;
    //发起读请求
    ret = aio_read(&cbp);
    if(ret < 0)
    {
        perror("aio_read error\n");
        exit(1);
    }

    // 到这里就可以去做其他事情，做完再回来查询aio的状态
```

```
    //查看异步读取的状态，直到读取请求完成
    for(i = 1;aio_error(&cbp) == EINPROGRESS;i++)
    {
        printf("No.%3d\n",i);
    }
    //读取返回值
    ret = aio_return(&cbp);
    printf("return %d\n",ret);

    //    sleep(1);
    printf("%s\n",(char*)cbp.aio_buf);
    close(fd);
    return 0;
}
```

再举一个多路复用的例子：

```
#include<stdio.h>
#include<sys/types.h>
#include<sys/stat.h>
#include<errno.h>
#include<fcntl.h>
#include<aio.h>
#include<stdlib.h>
#include<strings.h>

#define BUFSIZE   1024
#define MAX       2

//异步读请求
int aio_read_file(struct aiocb *cbp,int fd,int size)
{
    int               ret;
    bzero(cbp,sizeof(struct aiocb));

    cbp->aio_buf = (volatile void*)malloc(size+1);
    cbp->aio_nbytes = size;
    cbp->aio_offset = 0;
    cbp->aio_fildes = fd;

    ret = aio_read(cbp);
    if(ret < 0)
    {
        perror("aio_read error\n");
        exit(1);
    }
}

int main()
{
    struct aiocb    cbp1,cbp2;
    int             fd1,fd2,ret;
    int             i = 0;
    //异步阻塞列表
    struct aiocb*   aiocb_list[2];
```

```
    fd1 = open("test.txt",O_RDONLY);
    if(fd1 < 0)
    {
        perror("open error\n");
    }
    aio_read_file(&cbp1,fd1,BUFSIZE);

    fd2 = open("test.txt",O_RDONLY);
    if(fd2 < 0)
    {
        perror("open error\n");
    }
    aio_read_file(&cbp2,fd2,BUFSIZE*4);

    //这里可以去做其他事情，完成后再回头查询aio状态

    //向列表加入两个请求
    aiocb_list[0] = &cbp1;
    aiocb_list[1] = &cbp2;
    //阻塞，直到请求完成才会继续执行后面的语句
    aio_suspend((const struct aiocb* const*)aiocb_list,MAX,NULL);
    printf("read1:%s\n",(char*)cbp1.aio_buf);
    printf("read2:%s\n",(char*)cbp2.aio_buf);

    close(fd1);
    close(fd2);
    return 0;
}
```

上面两个例子都是上层主动查询等待完成，还有一种方式是底层 aio 完成以后再通知上层，有两种方式通知：信号和线程回调，这里以信号为例如下：

```
#include<stdio.h>
#include<sys/types.h>
#include<sys/stat.h>
#include<errno.h>
#include<fcntl.h>
#include<aio.h>
#include<stdlib.h>
#include<strings.h>
#include<signal.h>

#define BUFSIZE   256

//信号处理函数,参数signo接收的对应的信号值
void aio_handler(int signo)
{
    int           ret;
    printf("异步操作完成，收到通知\n");
}

int main()
{
    struct aiocb    cbp;
    int             fd,ret;
    int             i = 0;
```

```
    fd = open("test.txt",O_RDONLY);

    if(fd < 0)
    {
        perror("open error\n");
    }

    //填充struct aiocb 结构体
    bzero(&cbp,sizeof(cbp));
    //指定缓冲区
    cbp.aio_buf = (volatile void*)malloc(BUFSIZE+1);
    //请求读取的字节数
    cbp.aio_nbytes = BUFSIZE;
    //文件偏移
    cbp.aio_offset = 0;
    //读取的文件描述符
    cbp.aio_fildes = fd;
    //发起读请求

    //设置异步通知方式: SIGEV_SIGNAL, or SIGEV_THREAD
    //用信号通知
    cbp.aio_sigevent.sigev_notify = SIGEV_SIGNAL;
    //发送异步信号
    cbp.aio_sigevent.sigev_signo  = SIGIO;
    //传入aiocb 结构体
    cbp.aio_sigevent.sigev_value.sival_ptr = &cbp;

    //安装信号
    signal(SIGIO,aio_handler);
    //发起异步读请求
    ret = aio_read(&cbp);
    if(ret < 0)
    {
        perror("aio_read error\n");
        exit(1);
    }
    //暂停4秒，保证异步请求完成
    sleep(4);
    close(fd);
    return 0;
}
```

显然从节省 cpu 角度来看，应该后面两种方式更优，所以推荐用后两种方式。想了解更多 glibc 的 aio 例子，可以看这篇文章;

### 7.2.2 linux native aio

linux native aio 则是内核提供了一套 aio 接口，所以在所有 linux 发行版上都能用，包括 android。先来看一下具体接口：

```
int io_setup(int maxevents, io_context_t *ctxp);  /* 创建一个异步IO上下文（io_context_t是一个句柄） */
int io_destroy(io_context_t ctx);  /* 销毁一个异步IO上下文（如果有正在进行的异步IO，取消并等待它们完成） */
long io_submit(aio_context_t ctx_id, long nr, struct iocb **iocbpp);  /* 提交异步IO请求 */
long io_cancel(aio_context_t ctx_id, struct iocb *iocb, struct io_event *result);  /* 取消一个异步IO请求 */
long io_getevents(aio_context_t ctx_id, long min_nr, long nr, struct io_event *events, struct timespec *timeout)  /* 等待并获取异步IO请求的事件（也就是异步请求的处理结果） */
```

通常为了简化编程，会通过 libaio 来辅助，下面是一个具体例子：

```
#include<stdio.h>
#include<fcntl.h>
#include<string.h>
#include<stdlib.h>
#include<libaio.h>
#include<errno.h>
#include<unistd.h>
#include<unistd.h>

#define MAX_COUNT 1024
#define BUF_SIZE  1 * 1024 *1024

#ifndef O_DIRECT
#define O_DIRECT         040000 /* direct disk access hint */
#endif

int main(int args, void *argv[]){
    int fd;

    void * buf = NULL;

    //获取页面大小
    int pagesize = sysconf(_SC_PAGESIZE);
    //处理对齐
    posix_memalign(&buf, pagesize, BUF_SIZE);

    memset(buf,'A',BUF_SIZE);

    io_context_t ctx;
    struct iocb io,*p=&io;
    struct io_event e[10];
    struct timespec timeout;

    memset(&ctx,0,sizeof(ctx));
    //创建并初始化context
    if(io_setup(MAX_COUNT,&ctx)!=0){
        printf("io_setup error\n");
        return -1;
    }

    if((fd = open("./test.txt", O_WRONLY | O_CREAT | O_APPEND | O_DIRECT, 0644))<0) {
```

```
        perror("open error");
        io_destroy(ctx);
        return -1;
    }

    int n = MAX_COUNT;

    while(n > 0) {
        //调用libaio初始化iocb，如果没有libaio可以自己初始化iocb
        io_prep_pwrite(&io, fd, buf, BUF_SIZE, 0);
        //提交请求
        if(io_submit(ctx, 1, &p)!=1) {
            io_destroy(ctx);
            printf("io_submit error\n");
            return -1;
        }
        // 这里可以跳出去做其他事情，回来再查询aio状态

        //获取完成事件
        int ret = io_getevents(ctx, 1, 10, e, NULL);
        if (ret != 1) {
            perror("ret != 1");
            break;
        }
        n--;
    }

    close(fd);
    //销毁context
    io_destroy(ctx);
    return 0;
}
```

大部分 linux 平台都有 libaio，实在没有也可以自己移植，也比较简单，可以从这里下载，对于 android 平台来说，也有一个简化的 libaio（system/core/libasyncio），低版本如果没有，可以从高版本去拷贝。

通过阅读 glibc 的源码我们可以知道，glibc 的 aio 实际上是新创建一个线程去做实际的 io 操作，以解放主线程，而 linux native aio 则利用 cpu 和 io 可以并行工作的原理，在 io 处理过程中调用者可以完成其他工作。前者多了线程的创建和同步通信开销，所以对于 cpu 资源非常紧张的场景，后者应该是更好的选择。

## 7.3 通过 ioctl 控制数据回写

文件系统的很多高级功能都是通过 ioctl 来实现，这里介绍 f2fs 的两个新特性：atomic write 和 volatile write。前者实现原子写，可以实现异常掉电下文件的数据一致性；后者则是强制数据缓存在 page cache 中，减少 writeback 对性能的影响，二者都可以应用于数据库的优化。下面是两个具体例子：

```
static int unixFileControl(sqlite3_file *id, int op, void *pArg){
  unixFile *pFile = (unixFile*)id;
  switch( op ){
#if defined(__linux__) && defined(SQLITE_ENABLE_BATCH_ATOMIC_WRITE)
    case SQLITE_FCNTL_BEGIN_ATOMIC_WRITE: {
      int rc = osIoctl(pFile->h, F2FS_IOC_START_ATOMIC_WRITE);
      return rc ? SQLITE_IOERR_BEGIN_ATOMIC : SQLITE_OK;
```

```
    }
    case SQLITE_FCNTL_COMMIT_ATOMIC_WRITE: {
      int rc = osIoctl(pFile->h, F2FS_IOC_COMMIT_ATOMIC_WRITE);
      return rc ? SQLITE_IOERR_COMMIT_ATOMIC : SQLITE_OK;
    }
    case SQLITE_FCNTL_ROLLBACK_ATOMIC_WRITE: {
      int rc = osIoctl(pFile->h, F2FS_IOC_ABORT_VOLATILE_WRITE);
      return rc ? SQLITE_IOERR_ROLLBACK_ATOMIC : SQLITE_OK;
    }
// 这是sqlite的一段源码，可以看到其数据原子性是直接依赖于底层实现，所以sqlite在f2fs下比ext4性能好很多
```

```
static int keepFileInMemory(int fd) {
  ioctl(fd, F2FS_IOC_START_VOLATILE_WRITE);
  // 这段区间对这个文件的写操作全部都会在内存中，不会发生回写
  ...
  ioctl(fd, F2FS_IOC_RELEASE_VOLATILE_WRITE);
}
```

# 8 开机LOG分析

## 8.1 挂载失败

Android 7.0 以后大部分厂商都是默认启用加密，这时候对 DATA 分区来说，整个开机过程会有两次 Mount 动作，第一次的时候直接挂载原始设备（即加密过的分区），如果失败就说明分区加密过了，Vold 会去配置 dm-crypt 设备节点，配置成功后会多一个 dm-x（ x 是从0开始顺序递增，比如你前面已经映射了3个 dm 设备，那这一次映射 x = 3，设备节点名字就是 dm-3 ），配置成功后会进行第二次挂载，这次挂载的就是 dm-x 设备（即解密后的分区）。

所以，在看到开机LOG中如果看到 DATA 分区挂载失败，并不一定是文件系统出错了，也有可能启用了加密，请看如下日志：

```
.48104] random: fsck.f2fs: uninitialized urandom read (40 bytes read, 79 bits of entropy available)
.58287] random: fsck.f2fs: uninitialized urandom read (40 bytes read, 79 bits of entropy available)
.63103] fsck.f2fs: Info: Fix the reported corruption.
.63103]
.63301] fsck.f2fs: Info: Segments per section = 1
.63301]
.63410] fsck.f2fs: Info: Sections per zone = 1
.63410]
.63450] fsck.f2fs: Info: sector size = 512
.63450]
.63491] fsck.f2fs: Info: total sectors = 6263808 (3058 MB)
.63491]
.63570] fsck.f2fs:       Can't find a valid F2FS superblock at 0x0
.63570]
.63610] fsck.f2fs:       Can't find a valid F2FS superblock at 0x1
.63610]
.63659] fsck.f2fs: fsck.f2fs terminated by exit(255)
.63659]
.65794] F2FS-fs (mmcblk1p17): Magic Mismatch, valid(0xf2f52010) - read(0xd0ba4f8)
.65862] F2FS-fs (mmcblk1p17): Can't find valid F2FS filesystem in 1th superblock
.66089] F2FS-fs (mmcblk1p17): Magic Mismatch, valid(0xf2f52010) - read(0x6704fab9)
.66153] F2FS-fs (mmcblk1p17): Can't find valid F2FS filesystem in 2th superblock
.66204] F2FS-fs (mmcblk1p17): Magic Mismatch, valid(0xf2f52010) - read(0xd0ba4f8)
.66263] F2FS-fs (mmcblk1p17): Can't find valid F2FS filesystem in 1th superblock
.66286] F2FS-fs (mmcblk1p17): Magic Mismatch, valid(0xf2f52010) - read(0x6704fab9)
.66305] F2FS-fs (mmcblk1p17): Can't find valid F2FS filesystem in 2th superblock
.66588] init: [libfs_mgr]__mount(source=/dev/block/by-name/userdata,target=/data,type=f2fs)=-1: Invalid argument
.66733] init: [libfs_mgr]Running /system/bin/fsck.f2fs -a /dev/block/by-name/userdata
.85048] fsck.f2fs: Info: Fix the reported corruption.
.85048]
.85226] fsck.f2fs: Info: Segments per section = 1
.85226]
```

==启用加密后的首次挂载失败，有两个特征：失败日志中有"Magic Mismatch"提示；并且设备节点是'"userdata"。只要符合这两个特征，基本就可以认为是启用加密后的正常日志，可以继续找一下后面挂载 dm-x 设备的日志，看是否成功。==

1. Page Cache 中被修改过的 Page 会被标记为脏，其中的数据被叫做脏数据↩