第十四篇文件系统操作一网打尽

日期：2014-04-08

✧ 文件系统简介

文件系统是文件存放在磁盘等存储设备上的组织方式。RT-Thread 的文件系统组件提供了对多种文件系统类型的支持，如：

Fatfs（适用于 SPI Nor Flash、SD 卡等）；

Romfs（存在于 MCU 片内 flash，只读）；

Ramfs（存在于 MCU 片内 ram 或外扩 ram、sdram 中）；

NFS（网络文件系统，可将 PC 端目录挂载到目标板的文件系统下）；

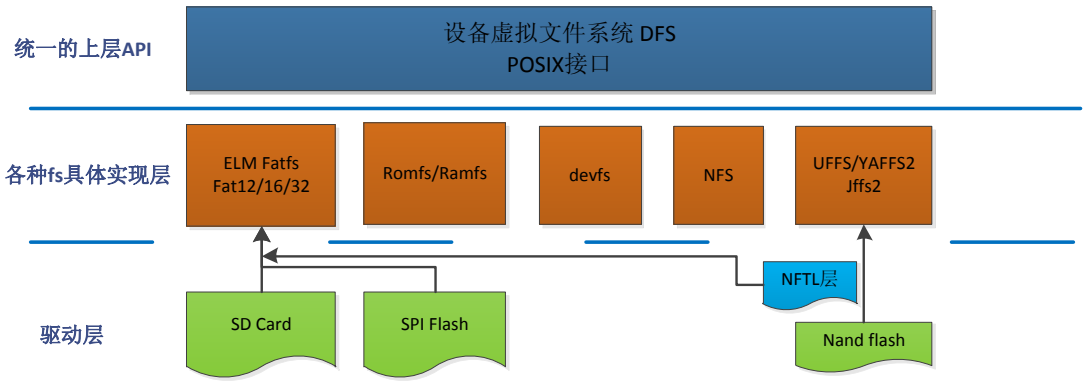
UFFS（Ultra-low-cost Flash File System，适用于 nand flash，和 yaffs2 比，ram 需求较少）；

Devfs（Device File system，可以用文件系统 API 操作注册的各个硬件设备）；

Yaffs2（Yet Another Flash File System2，适用于 nand flash，和 uffs 比，ram 需求较多）；

Jffs2（Journalling Flash File System Version2 适用于 nand flash，其功能就是管理在 MTD 设备上实现的日志型文件系统）。

为了能够适配上面说所的各种类型的文件系统，RT-Thread 在顶层设计了一套设备虚拟文件系统（DFS）并使用 POSIX 文件接口作为访问文件时的统一 API。下图展示了 RT-Thread 的文件系统结构：



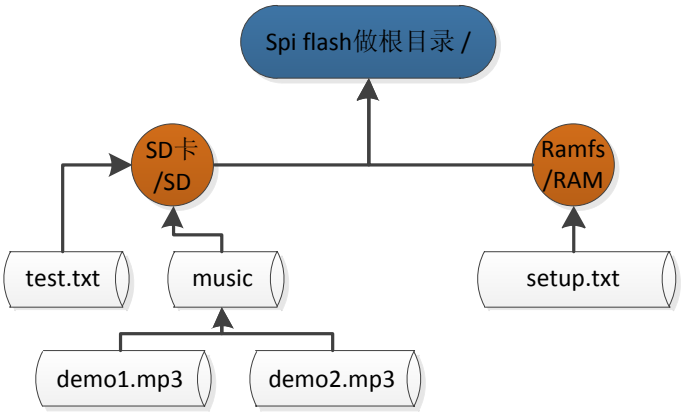
最上面的 DFS 层为我们提供了访问文件的各种 API：open、close、read、write 等等；中间一层是 RT-Thread 目前支持的各类 fs，我们可以在 rtconfig.h 中选择我们要使用哪个或哪几个具体的 fs；最底层是设备驱动层，我们需要为自己使用的硬件存储设备提供相应的驱动程序。上图中的 NFTL（nand flash transformer　layer）层使得 nand flash 可以支持 Fatfs。

✧ RT-Thread 下目录挂载结构

RT-Thread 下的文件系统节点采用挂载的方式。文件系统的根目录用“/”表示，根目录以下的子目录可以再挂载其他存储设备。比如我们可以将 spi flash 作为 Fatfs 挂载到根目录“/”下，sd 卡作为 Fatfs 挂载到根目录下的子目录“/SD”下，ramfs 挂载到根目录下的子目录“/RAM 下，等等。

注意：在根目录下挂载其他文件系统节点时需要先创建相应的子目录，当这个子目录不存在时，是挂载不成功的。我们可在 finsh 命令行下使用命令 mkdir(“/abc”)来创建子目录，其中 abc 为需要创建的子目录，如 mkdir(“/SD”)、mkdir(“/RAM”)等。

以上面的例子为基础，我们做一个树形结构的话，如下图：



当我们要访问某个文件时，用户需要提供所要访问的文件的路径，在 RT-Thread 中为了节省 ram 开销，我们往往不开启相对路径支持，所以需要使用这个文件的绝对路径来访问，比如要访问 sd 卡下 music 目录下的 demo1.mp3 文件时，它的绝对路径是“/SD/music/demo1.mp3”。

✧ 文件系统下的系统命令

RT-Thread 下的文件系统组件为用户提供了很多常用的命令，在工程调试中这些命令会给我们提供极大的方便，在这里我来列一下这些命令，并说明下用法。

我们运行这一篇的第一个例子《例程 1-elm_fatfs》，在 finsh 下使用 list()查看系统所提供的命令：

```
list()
--Function List:
device_test       -- e.g: device_test("sd0")
fs_test           -- file system R/W test. e.g: fs_test(3)
list_dir          -- list directory
reset             -- reset
led               -- set led[0 - 3] on[1] or off[0].
list_mem          -- list memory usage information
exec              -- exec module from a file
mkfs              -- make a file system
df                -- get disk free
ls                -- list directory contents
rm                -- remove files or directories
cat               -- print file
copy              -- copy source file to destination file
mkdir             -- create a directory
cd                -- change current working directory
list_date         -- show date and time.
set_date          -- set date. e.g: set_date(2010,2,28)
set_time          -- set time. e.g: set_time(23,59,59)
hello             -- say hello world
version           -- show RT-Thread version information
list_thread       -- list thread
list_sem          -- list semaphone in system
list_event        -- list event in system
list_mutex        -- list mutex in system
list_mailbox      -- list mail box in system
list_msgqueue     -- list message queue in system
list_memheap      -- list memory heap in system
list_mempool      -- list memory pool in system
list_timer        -- list timer in system
list_device       -- list device in system
list_module       -- list module in system
list_mod_detail   -- list module objects in system
list              -- list all symbol in system
--Variable List:
dummy             -- dummy variable for finsh
        0, 0x00000000
finsh />
```

上图中红色框中的命令就是系统为我们提供的和 fs 相关的命令，具体使用方法见下表：

命令	使用举例	说明	备注
mkfs	dfs_mkfs("elm", "flash0")	相当于格式化，将 flash0 设备按照 elm fatfs 的格式进行格式化	注册为 Fatfs 类型的存储设备必须先进行格式化
df	df("/SD")	查看挂载在 SD 目录下的 sd 卡还有多少剩余空间	即 disk free
ls	ls("/ABC")	列出/ABC 目录下的文件	ls("/")表示列出根目录下的文件
rm	rm("/ABC")	删除根目录下的 ABC 文件	如果 ABC 为目录则要保证目录为空才能删除成功

cat	cat("/demo.txt")	打印出 demo.txt 文件	用于查看文件内容
copy	copy("/SD/dem.txt","/dem.txt")	将/SD 目录下的 dem.txt 文件拷贝到根目录/下	用于不同存储设备间互相拷贝文件
mkdir	mkdir("/ts")	在根目录下创建 ts 子目录	用于新建目录
cd	切换目录命令，仅当需要支持工作目录时才需要，我们一般不用		

✧ 文件的基本操作 API 详解

1、打开(创建)文件 - open

在对文件进行读、写之前，必须先打开（创建）文件，这个动作所对应的 API 为 open 函数：

```
int open(const char *file, int flags, int mode)
```

参数 file：要打开或创建的文件名（使用文件的绝对路径，如“/SD/demo.txt“）；

参数 flags：指定文件打开的方式：

O_RDONLY	以只读方式打开
O_WRONLY	以只写方式打开
O_RDWR	以读写方式打开
O_APPEND	以追加的方式打开，即写入的内容添加在在文件末尾
O_CREAT	如果文件不存在，则创建该文件
O_TRUNC	如果文件已经存在，则在写入数据之前先删除其原有数据

参数 mode：这个参数为了符合 POSIX 接口而设置的，目前无实际意义，传入 0 即可；

返回值：如果文件成功打开或创建，则返回一个文件描述符，以后对该文件的操作就可以通过这个文件描述符来进行了，否则返回-1。

2、关闭文件 – close

使用完文件后可以用 close 函数来关闭该文件：

```
int close(int fd);
```

参数 fd：要关闭的文件描述符；

返回值：成功返回 0，否则返回-1。

调用 close 函数后系统会将数据写入到磁盘，然后释放该文件所占用的资源。

3、读数据 – read

read 函数用于从指定的文件中读取数据，形式如下：

```
int read(int fd, void *buf, size_t len)
```

参数 fd：要读取数据的文件描述符；

参数 buf：内存缓冲区地址，读出的数据会放在 buf 中；

参数 len：要读取的字节数，文件的读写位置随读取到的字节移动；

返回值：实际读到的字节数有，4 种可能的情况：

a、 返回值等于要读取的长度 len；

b、 返回值小于要读取的长度 len，表示文件剩余不多，只剩下不足 len 所要求的长度了；

c、 返回值等于 0，表示已经到达了文件的结尾；

d、 返回值为-1，表示读取中发生错误。

4、写数据 – write

write 函数用于向指定的文件中写入数据，形式如下：

```
int write(int fd, const void *buf, size_t len)
```

参数 fd：要写入数据的文件描述符；

参数 buf：指向要写入的数据缓冲区地址；

参数 len：要写入的字节数，文件的读写位置随之移动；

返回值：实际写入的字节数有，如果写入过程中发生错误则返回-1。

5、读写位置移动 – lseek

lseek 函数用于控制文件的读写位置，当文件被打开时，当前读写位置在文件的开头（另外的是使用 O_APPEND 方式打开时，当前读写位置在文件的末尾），随着读写操作的进行，当前读写位置不断变化，lseek 函数形式如下：

```
off_t lseek(int fd, off_t offset, int whence)
```

参数 fd：文件描述符；

参数 offset：偏移量；

参数 whence：用来设定进行读写位置偏移的参考点，取值如下：

SEEK_SET	参考点为文件的开头，新的读写位置即为偏移量
SEEK_CUR	参考点为当前位置，新的读写位置为当前位置加上偏移量
SEEK_END	参考点为文件的末尾，新的读写位置为当前位置加上偏移量（这时偏移量一般是负数）

返回值：返回当前的读写位置，如果操作过程中发生错误则返回-1。

lseek 函数使用技巧：

a、将读写位置移到文件的开头：`lseek(fd, 0, SEEK_SET)`

b、将读写位置移到文件的结尾：`lseek(fd, 0, SEEK_END)`

c、获取当前的读写位置：`lseek(fd, 0, SEEK_CUR)`

d、获得文件大小：`filesize = lseek(fd, 0, SEEK_END)`

lseek 函数使用注意点：

a、只有当参数 whence 为 SEEK_CUR 或 SEEK_END 时，参数 offset 才允许设为负数；

b、如果文件不是以 O_RDONLY 方式打开的，则新的读写位置可以超过文件的结尾；

c、devfs 下不支持此函数。

6、文件重命名 – rename

对一个文件重命名可用 rename 函数，它的形式如下：

```
int rename(const char *old, const char *new)
```

参数 old：原文件名（使用文件的绝对路径）；

参数 new：重新命名的文件名（使用文件的绝对路径）；

返回值：正常返回 0，如果调用过程中发生错误则返回-1。

7、删除文件、目录 – unlink

文件、目录的删除使用 unlink 函数，它的形式如下：

```
int unlink(const char *pathname)
```

参数 pathname：需要删除的文件名或目录名（使用文件的绝对路径）；

返回值：正常返回 0，如果调用过程中发生错误则返回-1。

注：如果需要删除目录，需要先保证目录为空；此函数和前面说过的 **rm** 命令功能一样。

8、获取文件属性 – stat、fstat

在 RT_Thread 中，文件的状态用如下的结构体记录：

```
struct stat
{
    rt_device_t st_dev;     /* 设备编号*/
    rt_uint16_t st_mode;    /* 文件类型*/
    rt_uint32_t st_size;    /* 文件总大小 */
    rt_time_t   st_mtime;    /* 文件修改时间*/
    rt_uint32_t st_blksize; /* 扇区大小*/
};
```

其中只有红色标记的域有实际意义，文件类型 st_mode 的可取值主要有如下几个：

```
#define DFS_S_IFSOCK           0140000  套接字
#define DFS_S_IFREG            0100000  常规文件
#define DFS_S_IFBLK            0060000  块设备文件
#define DFS_S_IFDIR            0040000  目录文件
#define DFS_S_IFCHR            0020000  字符设备文件
```

用户使用 st_mode 于就可以得到文件的大小和类型，这对于文件操作是很有用的（-^-还记得如何用 lseek 函数获得文件大小吗？-^-）

获取文件的属性状态可使用的函数有 stat 和 fstat，它们的形式如下:

```
int stat(const char *file, struct stat *buf)
```

参数 file：文件名（使用文件的绝对路径）；

参数 buf：stat 结构的指针；

返回值：正常返回 0，如果调用过程中发生错误则返回-1。

```
int fstat(int fildes, struct stat *buf)
```

参数 filedes：文件描述符；

参数 buf：stat 结构的指针；

返回值：正常返回 0，如果调用过程中发生错误则返回-1。

注意：使用 stat、fstat 函数获取目录（即文件夹）属性时，st_size 域是无效的，st_size 域只对普通文件有效。

✧ **目录的基本操作 API 详解**

1、创建目录 – mkdir

mkdir 函数用来创建一个目录文件（文件夹），它的一般形式如下：

```
int mkdir(const char *path, mode_t mode)
```

参数 path：要创建的目录名；

参数 mode：目录的访问权限设置，这里无意义，传入 0；

返回值：正常返回 0，如果创建失败则返回-1。

2、删除目录 – rmdir

和 mkdir 相反，rmdir 函数用来删除一个目录文件（文件夹），它的一般形式如下：

```
int rmdir(const char *pathname)
```

参数 pathname：要删除的目录名；

返回值：正常返回 0，如果删除失败则返回-1。

注意：只有目录为空时，才能正确删除，所以我们在使用此函数删除目录时必须先保证目录为空，如果不为空需要先删除此目录下的所有文件。

3、打开目录 – opendir

打开目录的函数是 opendir，它的一般形式如下：

```
DIR *opendir(const char *name)
```

参数 name：要打开的目录名；

返回值：正常返回 DIR 结构的指针，如果打开失败则返回 NULL。

4、关闭目录 – closedir

关闭目录的函数是 closedir，它的一般形式如下：

```
int closedir(DIR *d)
```

参数 d：指向 DIR 结构的指针；

返回值：正常返回 0，否则返回-1。

5、读取目录 – readdir

readdir 函数用来读取一个目录文件，它的一般形式如下：

```
struct dirent *readdir(DIR *d)
```

参数 d：指向 DIR 结构的指针；

返回值：返回一个指向 dirent 结构的指针，反复调用此函数读取就可以遍历目录中所有的文件，读到结尾将返回 NULL。

dirent 结构体定义如下：

```
struct dirent
{
    rt_uint8_t d_type;          /* 文件类型 */
    rt_uint8_t d_namlen;        /* 文件名长度 */
    rt_uint16_t d_reclen;       /* 记录的长度 */
    char d_name[DFS_PATH_MAX];   /* 文件名*/
};
```

其中 d_type 域 的可能值如下：

```
#define DFS_DT_UNKNOWN          0x00
#define DFS_DT_REG              0x01 /* 普通文件 */
#define DFS_DT_DIR              0x02 /* 目录 */
```

和 read 函数一样 readdir 函数

6、获取当前读取位置 – telldir

调用 readdir 时，每读取一个条目，当前读取位置就会向后移动一个，我们可以调用 telldir 函数来获得当前的读取位置，它的一般形式如下：

```
long telldir(DIR *d)
```

参数 d：指向 DIR 结构的指针；

返回值：返回当前的读取位置。

7、设置目录读取位置 – seekdir

我们可以用函数 seekdir 来设置即将要读取的目录位置，它的一般形式如下：

```
void seekdir(DIR *d, off_t offset)
```

参数 d：指向 DIR 结构的指针；

参数 offset：偏移位置；

返回值：无。

8、设置目录读取位置到开头 –rewinddir

rewinddir 函数设置读取目录的位置到开头，它的一般形式如下：

```
void rewinddir(DIR *d)
```

参数 d：指向 DIR 结构的指针；

返回值：无。

✧ 文件、目录操作实例

1、 文件读写

本篇例子《第 14 篇-文件系统例程\例程 1-elm_fatfs》中的 applications/readwrite.c 是一个文件读写的演示，例子将一个文件打开，分两次写入相同的数据，然后读出文件中的数据和原始写入相比较，看是否相同。例子源码如下：

```
#include <rtthread.h>
#include <dfs_posix.h>

#define TEST_FN     "/test.dat"

static char test_data[120], buffer[120];

/* 文件读写测试 */
void readwrite(void)
{
    int fd;
    int index, length;

    /* 以只写方式打开，如果文件不存在则创建，如果文件中原来有数据，则清空 */
    fd = open(TEST_FN, O_WRONLY | O_CREAT | O_TRUNC, 0);
    if (fd < 0)
    {  /* 我们需要对打开操作的结果做判断，如果失败了，则不能再进一步操作了 */
        rt_kprintf("open file for write failed\n");
        return;
    }

    /* 将写入缓冲区填充 */
    for (index = 0; index < sizeof(test_data); index ++)
    {
        test_data[index] = index + 27;
    }

    /* 这里 写入数据  将 testdata 数组中的数据写入文件，写入数量为数组大小*/
    length = write(fd, test_data, sizeof(test_data));
    if (length != sizeof(test_data))
    {
        rt_kprintf("write data failed\n");
        close(fd);
        return;
```

```
}

/* 关闭文件    关闭文件后 缓冲区中的数据才会 copy 到磁盘*/
close(fd);

/* 再次以只写的方式打开文件，O_APPEND 的意思是后续的操作都从文件末尾开始 */
fd = open(TEST_FN, O_WRONLY | O_CREAT | O_APPEND, 0);
if (fd < 0)
{
    rt_kprintf("open file for append write failed\n");
    return;
}
/* 再次写入数据   将 testdata 数组中的数据写入文件，写入数量为数组大小*/
length = write(fd, test_data, sizeof(test_data));
if (length != sizeof(test_data))
{
    rt_kprintf("append write data failed\n");
    close(fd);
    return;
}
/* 关闭文件 */
close(fd);

/* 只读打开文件  */
fd = open(TEST_FN, O_RDONLY, 0);
if (fd < 0)
{
    rt_kprintf("check: open file for read failed\n");
    return;
}

/* 读取数据(应该为第一次写入的数据) 读出后的数据在 buffer 中存储*/
length = read(fd, buffer, sizeof(buffer));
if (length != sizeof(buffer))
{
    rt_kprintf("check: read file failed\n");
    close(fd);
    return;
}

/* 检查数据是否正确 */
for (index = 0; index < sizeof(test_data); index ++)
{
    if (test_data[index] != buffer[index])
```

```
        {
            rt_kprintf("check: check data failed at %d\n", index);
            close(fd);
            return;
        }
    }

    /* 读取数据(应该为第二次写入的数据) 读出后的数据在 buffer 中存储*/
    length = read(fd, buffer, sizeof(buffer));
    if (length != sizeof(buffer))
    {
        rt_kprintf("check: read file failed\n");
        close(fd);
        return;
    }

    /* 检查数据是否正确 */
    for (index = 0; index < sizeof(test_data); index ++)
    {
        if (test_data[index] != buffer[index])
        {
            rt_kprintf("check: check data failed at %d\n", index);
            close(fd);
            return;
        }
    }

    /* 检查数据完毕，关闭文件 */
    close(fd);
    /* 打印结果 */
    rt_kprintf("read/write done.\n");
}

#ifdef RT_USING_FINSH
#include <finsh.h>
/* 输出函数到 finsh shell 命令行中 */
FINSH_FUNCTION_EXPORT(readwrite, perform file read and write test);
#endif
```

运行例程，在 finsh 命令行下输入 readwrite()命令，进行测试

```
finsh />readwrite()
read/write done.
        0, 0x00000000
```

还记得前面提到的 cat 命令吗？我们这里用 cat 命令打印出刚才写入的数据来看看：

```
finsh />cat("/test.dat")
 !"#$%&'()*+,-./0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmno
pqrstuvwxyz{|}~€岂儎厍喒塮媽嵠弻慷 !"#$%&'()*+,-./0123456789:;<=>?@ABCDEFGHIJKLM
NOPQRSTUVWXYZ[\]^_`abcdefghijklmnopqrstuvwxyz{|}~€岂儎厍喒塮媽嵠弻慷      0, 0x000
00000
```

2、目录操作

本篇例子《第 14 篇-文件系统例程\例程 1-elm_fatfs》中的 applications/ listdir.c 是一个目录操作的演示，例子会打开一个文件目录，并遍历这个目录下的所有文件，将这些条目都列出来。例子源码如下：

```
#include <rtthread.h>
#include <dfs_posix.h>

static char fullpath[256];

/* 例子会打印出 path 目录中所有的文件 */
void list_dir(const char *path)
{
    DIR *dir;

    /* 打开目录 */
    dir = opendir(path);
    if (dir != RT_NULL)
    {
        struct dirent *dirent;
        struct stat s;

        do
        {
            /* 读取目录条目 */
            dirent = readdir(dir);
            if (dirent == RT_NULL)
            {
                /* NULL 表示目录为空 */
                break;
            }
            rt_memset(&s, 0, sizeof(struct stat));

            /* 合并路径名，因为需要绝对路径 */
            rt_sprintf(fullpath, "%s/%s", path, dirent->d_name);
            /* 获取文件状态 */
            stat(fullpath, &s);

            /* 判断是目录还是普通文件 */
            if ( s.st_mode & DFS_S_IFDIR )
```

```
            {
                rt_kprintf("%s\t\t<DIR>\n", dirent->d_name);
            }
            else
            {
                /* 如果是普通文件，打印出文件大小 */
                rt_kprintf("%s\t\t%lu\n", dirent->d_name, s.st_size);
            }
        }
        /* 循环读取目录条目，直到最后一个条目读取完 */
        while (dirent != RT_NULL);

        /* 关闭目录 */
        closedir(dir);
    }
    else
    {
        rt_kprintf("open %s directory failed\n", path);
    }
}

#ifdef RT_USING_FINSH
#include <finsh.h>
/* 导出函数到 finsh 下 */
FINSH_FUNCTION_EXPORT(list_dir, list directory);
#endif
```

运行例程，在 finsh 命令行下输入 list_dir(“/”)命令，进行测试：

```
finsh />list_dir("/")
1.txt              0
resource                    <DIR>
SD                 <DIR>
firmware                    <DIR>
setup.ini                   79
wlan.nvm                    82
test.dat                    240
        0, 0x00000000
```

上面结果中出现 DIR 的表示此文件是目录，其他普通文件条目后面的数字表示文件的大小。

✧ ELM Fatfs 的使用

RT-Thread 中的 Fatfs 部分移植的是 elm fat（http://elm-chan.org/fsw/ff/00index_e.html，一个日本人写的 fatfs，想必大家在玩裸机文件系统时都有接触过这个吧-^-）