



下载APP



34 | 仓库管理：如何实现文件的六大基本操作？

2021-07-26 LMOS

《操作系统实战45讲》

课程介绍 >



讲述：陈晨

时长 20:29 大小 18.77M



你好，我是 LMOS。

我们在上一节课中，已经建立了仓库，并对仓库进行了划分，就是文件系统的格式化。有了仓库就需要往里面存取东西，对于我们的仓库来说，就是存取应用程序的文件。

所以今天我们要给仓库增加一些相关的操作，这些操作主要用于新建、打开、关闭、读写文件，它们也是文件系统的标准功能，自然即使我们这个最小的文件系统，也必须要支持。



好了，话不多说，我们开始吧。这节课的配套代码，你可以从 [这里](#) 下载。

辅助操作


通过上一节课的学习，我们了解了文件系统格式化操作，不难发现文件系统格式化并不复杂，但是它们需要大量的辅助函数。同样的，完成文件相关的操作，我们也需要大量的辅助函数。为了让你更加清楚每个实现细节，这里我们先来实现文件操作相关的辅助函数。

操作根目录文件

根据我们文件系统的设计，不管是新建、删除、打开一个文件，首先都要找到与该文件对应的 `rfmdir_t` 结构。

在我们的文件系统中，一个文件的 `rfmdir_t` 结构就储存在根目录文件中，所以想要读取文件对应的 `rfmdir_t` 结构，首先就要获取和释放根目录文件。

下面我们来实现获取和释放根目录文件的函数，代码如下所示。

 复制代码

```
1 //获取根目录文件
2 void* get_rootdirfile_blk(device_t* devp)
3 {
4     void* retptr = NULL;
5     rfmdir_t* rtdir = get_rootdir(devp); //获取根目录文件的rfmdir_t结构
6     //分配4KB大小的缓冲区并清零
7     void* buf = new_buf(FSYS_ALCBLKSZ);
8     hal_memset(buf, FSYS_ALCBLKSZ, 0);
9     //读取根目录文件的逻辑储存块到缓冲区中
10    read_rfsdevblk(devp, buf, rtdir->rdr_blknr)
11    retptr = buf; //设置缓冲区的首地址为返回值
12    goto errl1;
13 errl:
14    del_buf(buf, FSYS_ALCBLKSZ);
15 errl1:
16    del_rootdir(devp, rtdir); //释放根目录文件的rfmdir_t结构
17    return retptr;
18 }
19 //释放根目录文件
20 void del_rootdirfile_blk(device_t* devp, void* blkp)
21 {
22     //因为逻辑储存块的头512字节的空间中，保存的就是fimg_rhd_t结构
23     fimg_rhd_t* fmp = (fimg_rhd_t*)blkp;
24     //把根目录文件回写到储存设备中去，块号为fimg_rhd_t结构自身所在的块号
25     write_rfsdevblk(devp, blkp, fmp->fmd_sfbk)
26     //释放缓冲区
27     del_buf(blkp, FSYS_ALCBLKSZ);
28     return;
29 }
```

上述代码中，`get_rootdir` 函数的作用就是读取文件系统超级块中 `rfsdir_t` 结构到一个缓冲区中，`del_rootdir` 函数则是用来释放这个缓冲区，其代码非常简单，我已经帮你写好了。

获取根目录文件的方法也很容易，根据超级块中的 `rfsdir_t` 结构中的信息，读取根目录文件的逻辑储存块就行了。而释放根目录文件，就是把根目录文件的储存块回写到储存设备中去，最后释放对应的缓冲区就可以了。

获取文件名

下面我们来实现获取文件名，在我们的印象中，一个完整的文件名应该是这样的 `"/cosmos/drivers/drvrfs.c"`，这样的文件名包含了完整目录路径。

除了第一个 `"/` 是根目录外，其它的 `"/` 只是一个目录路径分隔符。然而，在很多情况下，我们通常需要把目录路径分隔符去除，提取其中的目录名称或者文件名称。为了简化问题，我们对文件系统来点限制，我们的文件名只能是 `"/xxxx"` 这种类型的。

下面我们就来实现去除路径分隔符提取文件名称的函数，代码如下所示。

 复制代码

```
1 //检查文件路径名
2 sint_t rfs_chkfilepath(char_t* fname)
3 {
4     char_t* chp = fname;
5     //检查文件路径名的第一个字符是否为“/”，不是则返回2
6     if(chp[0] != '/') { return 2; }
7     for(uint_t i = 1; ; i++)
8     {
9         //检查除第1个字符外其它字符中还有没有为“/”的，有就返回3
10        if(chp[i] == '/') { return 3; }
11        //如果这里i大于等于文件名称的最大长度，就返回4
12        if(i >= DR_NM_MAX) { return 4; }
13        //到文件路径字符串的末尾就跳出循环
14        if(chp[i] == 0 && i > 1) { break; }
15    }
16    //返回0表示正确
17    return 0;
18 }
19 //提取纯文件名
20 sint_t rfs_ret_fname(char_t* buf, char_t* fpath)
21 {
22     //检查文件路径名是不是“/xxxx”的形式
23     sint_t stus = rfs_chkfilepath(fpath);
24     //如果不为0就直接返回这个状态值表示错误
```

```

25     if(stus != 0) { return stus; }
26     //从路径名字符串的第2个字符开始复制字符到buf中
27     rfs_strcpy(&fpath[1], buf);
28     return 0;
29 }


```

上述代码中，完成获取文件名的是 `rfs_ret_fname` 函数，这个函数可以把 `fpath` 指向的路径名中的文件名提取出来，放到 `buf` 指向的缓冲区中，但在这之前，需要先调用 `rfs_chkfilepath` 函数检查路径名是不是 `"/xxxx"` 的形式，这是这个功能正常实现的必要条件。

判断文件是否存在

获取了文件名称，我们还需要实现这样一个功能：判断一个文件是否存在。因为新建和删除文件，要先判断储存设备里是不是存在着这个文件。具体来说，新建文件时，无法新建相同文件名的文件；删除文件时，不能删除不存在的文件。

我们一起通过后面这个函数还完成这个功能，代码如下所示。

 复制代码

```

1  sint_t rfs_chkfileisindex(device_t* devp, char_t* fname)
2  {
3      sint_t rets = 6;
4      sint_t ch = rfs_strlen(fname); //获取文件名的长度，注意不是文件路径名
5      //检查文件名的长度是不是合乎要求
6      if(ch < 1 || ch >= (sint_t)DR_NM_MAX) { return 4; }
7      void* rdblkp = get_rootdirfile_blk(devp);
8      fimgrhd_t* fmp = (fimgrhd_t*)rdblkp;
9      //检查该fimgrhd_t结构的类型是不是FMD_DIR_TYPE，即这个文件是不是目录文件
10     if(fmp->fmd_type != FMD_DIR_TYPE) { rets = 3; goto err; }
11     //检查根目录文件是不是为空，即没有写入任何数据，所以返回0，表示根目录下没有对应的文件
12     if(fmp->fmd_curfwritembk == fmp->fmd_fleblk[0].fb_blkstart &&
13     fmp->fmd_curfinwbkoff == fmp->fmd_fileifstbkoff) {
14         rets = 0; goto err;
15     }
16     rfsdir_t* dirp = (rfsdir_t*)((uint_t)(fmp) + fmp->fmd_fileifstbkoff); //指向
17     //指向根目录文件的结束地址
18     void* maxchkp = (void*)((uint_t)rdblkp + FSYS_ALCBLKSZ - 1);
19     //当前的rfsdir_t结构的指针比根目录文件的结束地址小，就继续循环
20     for(;;(void*)dirp < maxchkp;) {
21         //如果这个rfsdir_t结构的类型是RDR_FIL_TYPE，说明它对应的是文件而不是目录，所以下
22         if(dirp->rdr_type == RDR_FIL_TYPE) {
23             if(rfs_strcmp(dirp->rdr_name, fname) == 1) { //比较其文件名
24                 rets = 1; goto err;

```

```
25         }
26     }
27     dirp++;
28 }
29     rets = 0; //到了这里说明没有找到相同的文件
30 err:
31     del_rootdirfile_blk(devp,rdblkp); //释放根目录文件
32     return rets;
33 }
```

上述代码中，`rfs_chkfileisindex` 函数逻辑很简单。首先是检查文件名的长度，接着获取了根目录文件，然后遍历根其中的所有 `rfsdir_t` 结构并比较文件名是否相同，相同就返回 1，不同就返回其它值，最后释放了根目录文件。

因为 `get_rootdirfile_blk` 函数已经把根目录文件读取到内存里了，所以可以用 `dirp` 指针和 `maxchkp` 指针操作其中的数据。

好了，操作根目录文件、获取文件名、判断一个文件是否存在的三大函数就实现了，有了它们，再去实现文件相关的其它操作就方便多了，我们接着探索。

文件相关的操作

直到现在，我们还没对任何文件进行操作，而我们实现文件系统，就是为了应用程序更好地存放自己的“劳动成果”——文件，因此一个文件系统必须要支持一些文件操作。

下面我们将依次实现新建、删除、打开、读写以及关闭文件，这几大文件操作，这也是文件系统需要提供的最基本的功能。

新建文件


在没有文件之前，对任何文件本身的操作都是无效的，所以我们首先就要实现新建文件这个功能。

在写代码之前，我们还是先来看一看如何新建一个文件，一共可以分成后面这 4 步。

1. 从文件路径名中提取出纯文件名，检查储存设备上是否已经存在这个文件。

2. 分配一个空闲的逻辑储存块，并在根目录文件的末尾写入这个新建文件对应的 `rfsdir_t` 结构。
3. 在一个新的 4KB 大小的缓冲区中，初始化新建文件对应的 `fimgrhd_t` 结构。
4. 把第 3 步对应的缓冲区里的数据，写入到先前分配的空闲逻辑储存块中。


下面我们先来写好新建文件的接口函数。

 复制代码

```

1 //新建文件的接口函数
2 drvstus_t rfs_new_file(device_t* devp, char_t* fname, uint_t flg)
3 {
4     //在栈中分配一个字符缓冲区并清零
5     char_t fne[DR_NM_MAX];
6     hal_memset((void*)fne, DR_NM_MAX, 0);
7     //从文件路径名中提取出纯文件名
8     if(rfs_ret_fname(fne, fname) != 0) { return DFCERRSTUS; }
9     //检查储存介质上是否已经存在这个新建的文件，如果是则返回错误
10    if(rfs_chkfileisindex(devp, fne) != 0) {return DFCERRSTUS; }
11    //调用实际建立文件的函数
12    return rfs_new_dirfileblk(devp, fne, RDR_FIL_TYPE, 0);
13 }
```

我们在新建文件的接口函数中，就实现了前面第一步，完成了提取文件名和检查文件是否在储存设备中存在的的功能。接着我们来实现真正新建文件的函数，就是上述代码中 `rfs_new_dirfileblk` 函数，代码如下所示。

 复制代码

```

1 drvstus_t rfs_new_dirfileblk(device_t* devp, char_t* fname, uint_t flgtype, uint_t
2 {
3     drvstus_t rets = DFCERRSTUS;
4     void* buf = new_buf(FSYS_ALCBLKSZ); //分配一个4KB大小的缓冲区
5     hal_memset(buf, FSYS_ALCBLKSZ, 0); //清零该缓冲区
6     uint_t fblk = rfs_new_blk(devp); //分配一个新的空闲逻辑储存块
7     void* rdirblk = get_rootdirfile_blk(devp); //获取根目录文件
8     fimgrhd_t* fmp = (fimgrhd_t*)rdirblk;
9     //指向文件当前的写入地址，因为根目录文件已经被读取到内存中了
10    rfsdir_t* wrdirp = (rfsdir_t*)((uint_t)rdirblk + fmp->fmd_curfinwbkoff);
11    //对文件当前的写入地址进行检查
12    if(((uint_t)wrdirp) >= ((uint_t)rdirblk + FSYS_ALCBLKSZ)) {
13        rets=DFCERRSTUS; goto err;
```

```

14     }
15     wrdirp->rdr_stus = 0;
16     wrdirp->rdr_type = flgtype; // 设为文件类型
17     wrdirp->rdr_blknr = fblk; // 设为刚刚分配的空闲逻辑储存块
18     rfs_strcpy(fname, wrdirp->rdr_name); // 把文件名复制到 rfsdir_t 结构
19     fmp->fmd_filesz += (uint_t)(sizeof(rfsdir_t)); // 增加根目录文件的大小
20     // 增加根目录文件当前的写入地址，保证下次不被覆盖
21     fmp->fmd_curfinwbkoff += (uint_t)(sizeof(rfsdir_t));
22     fimgrhd_t* ftmp = (fimgrhd_t*)buf; // 指向新分配的缓冲区
23     fimgrhd_t_init(ftmp); // 调用 fimgrhd_t 结构默认的初始化函数
24     ftmp->fmd_type = FMD_FIL_TYPE; // 因为建立的是文件，所以设为文件类型
25     ftmp->fmd_sfbk = fblk; // 把自身所在的块，设为分配的逻辑储存块
26     ftmp->fmd_curfwritebk = fblk; // 把当前写入的块，设为分配的逻辑储存块
27     ftmp->fmd_curfinwbkoff = 0x200; // 把当前写入块的写入偏移量设为 512
28     // 把文件储存块数组的第 1 个元素的开始块，设为刚刚分配的空闲逻辑储存块
29     ftmp->fmd_fleblk[0].fb_blkstart = fblk;
30     // 因为只分配了一个逻辑储存块，所以设为 1
31     ftmp->fmd_fleblk[0].fb_blknr = 1;
32     // 把缓冲区中的数据写入到刚刚分配的空闲逻辑储存块中
33     if(write_rfsdevblk(devp, buf, fblk) == DFCERRSTUS) {
34         rets = DFCERRSTUS; goto err;
35     }
36     rets = DFCOKSTUS;
37 err:
38     del_rootdirfile_blk(devp, rdirblk); // 释放根目录文件
39 err1:
40     del_buf(buf, FSYS_ALCBLKSZ); // 释放缓冲区
41     return rets;
42 }

```

看完上述代码，我想提醒你，在 `rfs_new_dirfileblk` 函数中有两点很关键。

第一，前面反复提到的目录文件中存放的就是一系列的 `rfsdir_t` 结构。

第二，`fmp` 和 `ffmp` 这两个指针很重要。`fmp` 指针指向的是根目录文件的 `fimgrhd_t` 结构，因为要写入一个新的 `rfsdir_t` 结构，所以要获取并改写根目录文件的 `fimgrhd_t` 结构中的数据。而 `ffmp` 指针指向的是新建文件的 `fimgrhd_t` 结构，并且初始化了其中的一些数据。最后，该函数把这个缓冲区中的数据写入到分配的空闲逻辑储存块中，同时释放了根目录文件和缓冲区。

删除文件

新建文件的操作完成了，下面我们来实现删除文件的操作。

如果只能新建文件而不能删除文件，那么储存设备的空间最终会耗尽，所以文件系统就必须支持删除文件的操作。

同样的，还是先来了解删除文件的方法。删除文件可以通过后面这 4 步来实现。

1. 从文件路径名中提取出纯文件名。
2. 获取根目录文件，从根目录文件中查找待删除文件的 `rfsdir_t` 结构，然后释放该文件占用的逻辑储存块。
3. 初始化与待删除文件相对应的 `rfsdir_t` 结构，并设置 `rfsdir_t` 结构的类型为 `RDR_DEL_TYPE`。
4. 释放根目录文件。

这次我们用三个函数来实现这些步骤，删除文件的接口函数的代码如下。

[复制代码](#)

```
1 //文件删除的接口函数
2 drvstus_t rfs_del_file(device_t* devp, char_t* fname, uint_t flg)
3 {
4     if(flg != 0) {
5         return DFCERRSTUS;
6     }
7     return rfs_del_dirfileblk(devp, fname, RDR_FIL_TYPE, 0);
8 }
```

删除文件的接口函数非常之简单，就是判断一下标志，接着调用了 `rfs_del_dirfileblk` 函数，下面我们就来写好这个 `rfs_del_dirfileblk` 函数。

[复制代码](#)

```
1 drvstus_t rfs_del_dirfileblk(device_t* devp, char_t* fname, uint_t flgtype, ui
2 {
3     if(flgtype != RDR_FIL_TYPE || val != 0) { return DFCERRSTUS; }
4     char_t fne[DR_NM_MAX];
5     hal_memset((void*)fne, DR_NM_MAX, 0);
6     //提取纯文件名
7     if(rfs_ret_fname(fne, fname) != 0) { return DFCERRSTUS; }
8     //调用删除文件的核心函数
```



```

9     if(del_dirfileblk_core(devp, fne) != 0) { return DFCERRSTUS; }
10    return DFCOKSTUS;
11 }

```

rfs_del_dirfileblk 函数只是提取了文件名，然后调用了一个删除文件的核心函数，这个核心函数就是 del_dirfileblk_core 函数，它的实现代码如下所示。

[复制代码](#)

```

1 //删除文件的核心函数
2 sint_t del_dirfileblk_core(device_t* devp, char_t* fname)
3 {
4     sint_t rets = 6;
5     void* rblkp=get_rootdirfile_blk(devp); //获取根目录文件
6     fimgrhd_t* fmp = (fimgrhd_t*)rblkp;
7     if(fmp->fmd_type!=FMD_DIR_TYPE) { //检查根目录文件的类型
8         rets=4; goto err;
9     }
10    if(fmp->fmd_curfwritebk == fmp->fmd_fleblk[0].fb_blkstart && fmp->fmd_curf
11        rets = 3; goto err;
12    }
13    rfssdir_t* dirp = (rfssdir_t*)((uint_t)(fmp) + fmp->fmd_fileifstbkoff);
14    void* maxchkp = (void*)((uint_t)rblkp + FSYS_ALCBLKSZ-1);
15    for(;(void*)dirp < maxchkp;) {
16        if(dirp->rdr_type == RDR_FIL_TYPE) { //检查其类型是否为文件类型
17            //如果文件名相同，就执行以下删除动作
18            if(rfs_strcmp(dirp->rdr_name, fname) == 1) {
19                //释放rfssdir_t结构的rdr_blknr中指向的逻辑储存块
20                rfs_del_blk(devp, dirp->rdr_blknr);
21                //初始化rfssdir_t结构，实际上是清除其中的数据
22                rfssdir_t_init(dirp);
23                //设置rfssdir_t结构的类型为删除类型，表示它已经删除
24                dirp->rdr_type = RDR_DEL_TYPE;
25                rets = 0; goto err;
26            }
27        }
28        dirp++; //下一个rfssdir_t
29    }
30    rets=1;
31 err:
32    del_rootdirfile_blk(devp,rblkp); //释放根目录文件
33    return rets;
34 }

```

上述代码中的 del_dirfileblk_core 函数，它主要是遍历根目录文件中所有的 rfssdir_t 结构，并比较其文件名，看看删除的文件名称是否相同，相同就释放该 rfssdir_t 结构的

rdr_blknr 字段对应的逻辑储存块，清除该 rfsdir_t 结构中的数据，同时设置该 rfsdir_t 结构的类型为删除类型。

你可以这样理解：删除一个文件，就是把这个文件对应的 rfsdir_t 结构中的数据清空，这样就无法查找到这个文件了。同时，也要释放该文件占用的逻辑储存块。因为没有清空文件数据，所以可以通过反删除软件找回文件。

打开文件

接下来，我们就要实现打开文件操作了。一个已经存在的文件，要对它进行读写操作，首先就应该打开这个文件。

在实现这个打开文件操作之前，我们不妨先回忆一下前面课程里提到的 [objnode_t 结构](#)。

Cosmos 内核上层组件调用设备驱动程序时，都需要建立一个相应的 objnode_t 结构，把这个 I/O 包发送给相应的驱动程序，但是 objnode_t 结构不仅仅是用于驱动程序，它还用于表示进程使用了哪些资源，例如打开了哪些设备或者文件，而每打开一个设备或者文件就建立一个 objnode_t 结构，放在特定进程的资源表中。

为了适应文件系统设备驱动程序，在 cosmos/include/krlinc/krlobjnode_t.h 文件中，需要在 objnode_t 结构中增加一些东西，代码如下所示。

 复制代码

```
1 #define OBJN_TY_DEV 1//设备类型
2 #define OBJN_TY_FIL 2//文件类型
3 #define OBJN_TY_NUL 0//默认类型
4 typedef struct s_OBJNODE
5 {
6     spinlock_t  on_lock;
7     list_h_t    on_list;
8     sem_t       on_complesem;
9     uint_t      on_flg;
10    uint_t      on_stus;
11    //.....
12    void*        on_fname; //文件路径名指针
13    void*        on_finode; //文件对应的fimgrhd_t结构指针
14    void*        on_extp; //扩展所用
15 }objnode_t;
```

上述代码中 `objnode_t` 结构里增加了两个字段，一个是指向文件路径名的指针，表示打开哪个文件。因为要知道一个文件的所有信息，所以增加了指向对应文件的 `fimgrhd_t` 结构指针，也就是我们增加的第二个字段。

现在来看看打开一个文件的流程。一共也是 4 步。

1. 从 `objnode_t` 结构的文件路径提取文件名。
2. 获取根目录文件，在该文件中搜索对应的 `rfsdir_t` 结构，看看文件是否存在。
3. 分配一个 4KB 缓存区，把该文件对应的 `rfsdir_t` 结构中指向的逻辑储存块读取到缓存区中，然后释放根目录文件。
4. 把缓冲区中的 `fimgrhd_t` 结构的地址，保存到 `objnode_t` 结构的 `on_finode` 域中。

下面来写两个函数实现这些流程，同样我们需要先写好接口函数，代码如下所示。

[复制代码](#)

```
1 //打开文件的接口函数
2 drvstus_t rfs_open_file(device_t* devp, void* iopack)
3 {
4     objnode_t* obp = (objnode_t*)iopack;
5     //检查objnode_t中的文件路径名
6     if(obp->on_fname == NULL) {
7         return DFCERRSTUS;
8     }
9     //调用打开文件的核心函数
10    void* fmdp = rfs_openfileblk(devp, (char_t*)obp->on_fname);
11    if(fmdp == NULL) {
12        return DFCERRSTUS;
13    }
14    //把返回的fimgrhd_t结构的地址保存到objnode_t中的on_finode字段中
15    obp->on_finode = fmdp;
16    return DFCOKSTUS;
17 }
```

接口函数 `rfs_open_file` 中只是对参数进行了检查。然后调用了核心函数，这个函数就是 `rfs_openfileblk`，它的代码实现如下所示。

```

1 //打开文件的核心函数
2 void* rfs_openfileblk(device_t *devp, char_t* fname)
3 {
4     char_t fne[DR_NM_MAX]; void* rets = NULL,*buf = NULL;
5     hal_memset((void*)fne,DR_NM_MAX,0);
6     if(rfs_ret_fname(fne, fname) != 0) {从文件路径名中提取纯文件名
7         return NULL;
8     }
9     void* rblkp = get_rootdirfile_blk(devp); //获取根目录文件
10    fimgrhd_t* fmp = (fimgrhd_t*)rblkp;
11    if(fmp->fmd_type != FMD_DIR_TYPE) {判断根目录文件的类型是否合理
12        rets = NULL; goto err;
13    }
14    //判断根目录文件里有没有数据
15    if(fmp->fmd_curfwritebk == fmp->fmd_fleblk[0].fb_blkstart &&
16    fmp->fmd_curfinwbkoff == fmp->fmd_fileifstbkoff) {
17        rets = NULL; goto err;
18    }
19    rfsdir_t* dirp = (rfsdir_t*)((uint_t)(fmp) + fmp->fmd_fileifstbkoff);
20    void* maxchkp = (void*)((uint_t)rblkp + FSYS_ALCBLKSZ - 1);
21    for(;(void*)dirp < maxchkp;) {开始遍历文件对应的rfsdir_t结构
22        if(dirp->rdr_type == RDR_FIL_TYPE) {
23            //如果文件名相同就跳转到opfblk标号处运行
24            if(rfs_strcmp(dirp->rdr_name, fne) == 1) {
25                goto opfblk;
26            }
27        }
28        dirp++;
29    }
30    //如果到这里说明没有找到该文件对应的rfsdir_t结构，所以设置返回值为NULL
31    rets = NULL; goto err;
32 opfblk:
33    buf = new_buf(FSYS_ALCBLKSZ); //分配4KB大小的缓冲区
34    //读取该文件占用的逻辑储存块
35    if(read_rfsdevblk(devp, buf, dirp->rdr_blknr) == DFCERRSTUS) {
36        rets = NULL; goto err1;
37    }
38    fimgrhd_t* ffmp = (fimgrhd_t*)buf;
39    if(ffmp->fmd_type == FMD_NUL_TYPE || ffmp->fmd_fileifstbkoff != 0x200) {
40        rets = NULL; goto err1;
41    }
42    rets = buf; goto err; //设置缓冲区首地址为返回值
43 err1:
44    del_buf(buf, FSYS_ALCBLKSZ); //上面的步骤若出现问题就要释放缓冲区
45 err:
46    del_rootdirfile_blk(devp, rblkp); //释放根目录文件
47    return rets;
48 }

```

结合上面的代码我们能够看到，通过 `rfs_openfileblk` 函数中的 `for` 循环，可以遍历要打开的文件在根目录文件中对应的 `rfmdir_t` 结构，然后把对应文件占用的逻辑储存块读取到缓冲区中，最后返回这个缓冲区的首地址。

因为这个缓冲区开始的空间中，就存放着其文件对应的 `fimgrhd_t` 结构，所以返回 `fimgrhd_t` 结构的地址，整个打开文件的流程就结束了。

读写文件


刚才我们已经实现了打开文件，而打开一个文件，就是为对这个文件进行读写。

其实对文件的读写包含两个操作，一个是从储存设备中读取文件的数据，另一个是把文件的数据写入到储存设备中。

咱们先来看看如何读取已经打开的文件中的数据，大致的流程如下。

1. 检查 `objnode_t` 结构中用于存放文件数据的缓冲区及其大小。
2. 检查 `imgrhd_t` 结构中文件相关的信息。
3. 把文件的数据读取到 `objnode_t` 结构中指向的缓冲区中。

通过后面的代码，我们把读文件的接口函数跟核心函数一起实现。

 复制代码

```
1 //读取文件数据的接口函数
2 drvstus_t rfs_read_file(device_t* devp,void* iopack)
3 {
4     objnode_t* obp = (objnode_t*)iopack;
5     //检查文件是否已经打开，以及用于存放文件数据的缓冲区和它的大小是否合理
6     if(obp->on_finode == NULL || obp->on_buf == NULL || obp->on_bufsz != FSYS_
7         return DFCERRSTUS;
8     }
9     return rfs_readfileblk(devp, (fimgrhd_t*)obp->on_finode, obp->on_buf, obp-
10 }
11 //实际读取文件数据的函数
12 drvstus_t rfs_readfileblk(device_t* devp, fimgrhd_t* fmp, void* buf, uint_t le
13 {
14     //检查文件的相关信息是否合理
15     if(fmp->fmd_sfbk != fmp->fmd_curfwbk || fmp->fmd_curfwbk != fmp->
```



```

16         return DFCERRSTUS;
17     }
18     //检查读取文件数据的长度是否大于 ( 4096-512 )
19     if(len > (FSYS_ALCBLKSZ - fmp->fmd_fileifstbkoff)) {
20         return DFCERRSTUS;
21     }
22     //指向文件数据的开始地址
23     void* wrp = (void*)((uint_t)fmp + fmp->fmd_fileifstbkoff);
24     //把文件开始处的数据复制len个字节到buf指向的缓冲区中
25     hal_memcpy(wrp, buf, len);
26     return DFCOKSTUS;
27 }

```

上述代码中读取文件数据的函数很简单，关键是要明白前面那个打开文件的函数，因为在那里它已经把文件数据复制到一个缓冲区中了，`rfs_readfileblk` 函数中的参数 `buf`、`len` 都是接口函数 `rfs_read_file` 从 `objnode_t` 结构中提取出来的，其它的部分我已经通过注释已经说明了。

好了，我们下面就来实现怎么向文件中写入数据，和读取文件的流程一样，只不过要将要写入的数据复制到打开文件时为其分配的缓冲区中，最后还要把打开文件时为其分配的缓冲区中的数据，写入到相应的逻辑储存块中。

我们还是把写文件的接口函数和核心函数一起实现，代码如下所示。

[复制代码](#)

```

1  //写入文件数据的接口函数
2  drvstus_t rfs_write_file(device_t* devp, void* iopack)
3  {
4      objnode_t* obp = (objnode_t*)iopack;
5      //检查文件是否已经打开，以及用于存放文件数据的缓冲区和它的大小是否合理
6      if(obp->on_finode == NULL || obp->on_buf == NULL || obp->on_bufsz != FSYS_
7          return DFCERRSTUS;
8      }
9      return rfs_writefileblk(devp, (fimgrhd_t*)obp->on_finode, obp->on_buf, obp
10 }
11 //实际写入文件数据的函数
12 drvstus_t rfs_writefileblk(device_t* devp, fimgrhd_t* fmp, void* buf, uint_t l
13 {
14     //检查文件的相关信息是否合理
15     if(fmp->fmd_sfbk != fmp->fmd_curfwritebk || fmp->fmd_curfwritebk != fmp->
16         return DFCERRSTUS;
17     }
18     //检查当前将要写入数据的偏移量加上写入数据的长度，是否大于等于4KB
19     if((fmp->fmd_curfinwbkoff + len) >= FSYS_ALCBLKSZ) {

```

```

20         return DFCERRSTUS;
21     }
22     //指向将要写入数据的内存空间
23     void* wrp = (void*)((uint_t)fmp + fmp->fmd_curfinwbkoff);
24     //把buf缓冲区中的数据复制len个字节到wrp指向的内存空间中去
25     hal_memcpy(buf, wrp, len);
26     fmp->fmd_filesz += len; //增加文件大小
27     //使fmd_curfinwbkoff指向下一次将要写入数据的位置
28     fmp->fmd_curfinwbkoff += len;
29     //把文件数据写入到相应的逻辑储存块中，完成数据同步
30     write_rfsdevblk(devp, (void*)fmp, fmp->fmd_curfwritebk);
31     return DFCOKSTUS;
32 }

```

上述代码中，你要注意的，**rfs_writefileblk 函数永远都是从 fimgrhd_t 结构的 fmd_curfinwbkoff 字段中的偏移量开始写入文件数据的**，比如向空文件中写入 2 个字节，那么其 fmd_curfinwbkoff 字段的值就是 2，因为第 0、1 个字节空间已经被占用了，这就是**追加写入数据**的方式。

rfs_writefileblk 函数最后调用 write_rfsdevblk 函数把文件数据写入到相应的逻辑储存块中，完成数据同步。我们发现只要打开文件了，读写文件还是很简单的，最后还要实现关闭文件的操作。

关闭文件

有打开文件的操作，就需要有关闭文件的操作，因为打开一个文件，会为此分配一个缓冲区，这些都是系统资源，所以需要有一个关闭文件的操作来释放这些资源，以防止系统资源泄漏。

关闭文件的流程很简单，首先检查文件是否已经打开。然后把文件写入到对应的逻辑储存块中，完成数据的同步。最后释放文件数据占用的缓冲区。下面我们开始写代码实现，我们依然把接口和核心函数放在一起实现，代码如下所示。

 复制代码

```

1 //关闭文件的接口函数
2 drvstus_t rfs_close_file(device_t* devp, void* iopack)
3 {
4     objnode_t* obp = (objnode_t*)iopack;
5     //检查文件是否已经打开了
6     if(obp->on_finode == NULL) {
7         return DFCERRSTUS;
8     }

```


```
9     return rfs_closefileblk(devp, obp->on_finode);
10 }
11 //关闭文件的核心函数
12 drvstus_t rfs_closefileblk(device_t *devp, void* fblkp)
13 {
14     //指向文件的fimgrhd_t结构
15     fimgrhd_t* fmp = (fimgrhd_t*)fblkp;
16     //完成文件数据的同步
17     write_rfsdevblk(devp, fblkp, fmp->fmd_sfbk);
18     //释放缓冲区
19     del_buf(fblkp, FSYS_ALCBLKSZ);
20     return DFCOKSTUS;
21 }
```

上述代码是非常简单的，但在目前的情况下，`rfs_closefileblk` 函数中是没有必要调用 `write_rfsdevblk` 函数的，因为前面在写入文件数据的同时，就已经把文件的数据写入到逻辑储存块中去了。最后释放了先前打开文件时分配的缓冲区，而 `objnode_t` 结构不应该在此释放，它是由 Cosmos 内核上层组件进行释放的。

串联整合

到目前为止，我们实现了文件相关的操作，并且提供了接口函数，但是我们的文件系统是以设备的形式存在的，所以文件操作的接口，必须要串联整合到文件系统设备驱动程序之中，文件系统才能真正工作。

下面我们就去整合联串文件系统设备驱动程序。首先来串联整合文件系统的打开文件操作和新建文件操作，代码如下所示。

 复制代码

```
1  drvstus_t rfs_open(device_t* devp, void* iopack)
2  {
3      objnode_t* obp=(objnode_t*)iopack;
4      //根据objnode_t结构中的访问标志进行判断
5      if(obp->on_acsflgs == FSDEV_OPENFLG_OPEFILE) {
6          return rfs_open_file(devp, iopack);
7      }
8      if(obp->on_acsflgs == FSDEV_OPENFLG_NEWFILE) {
9          return rfs_new_file(devp, obp->on_fname, 0);
10     }
11     return DFCERRSTUS;
12 }
```

上述代码中 `rfs_open` 函数对应于设备驱动程序的打开功能派发函数，但没有相应的新建功能派发函数，于是我们就根据 `objnode_t` 结构中访问标志域设置不同的编码，来进行判断。

接着我们来串联整合关闭文件的操作。这次要简单一些，因为设备驱动程序有对应的关闭功能派发函数，直接调用关闭文件操作的接口函数就可以了，代码如下所示。

[复制代码](#)

```
1 drvstus_t rfs_close(device_t* devp, void* iopack)
2 {
3     return rfs_close_file(devp, iopack);
4 }
```

然后是文件读写操作的串联整合，设备驱动程序也有对应的读写功能派发函数，同样也是直接调用文件读写操作的接口函数即可，代码如下所示。

[复制代码](#)

```
1 drvstus_t rfs_read(device_t* devp, void* iopack)
2 {
3     //调用读文件操作的接口函数
4     return rfs_read_file(devp, iopack);
5 }
6 drvstus_t rfs_write(device_t* devp, void* iopack)
7 {
8     //调用写文件操作的接口函数
9     return rfs_write_file(devp, iopack);
10 }
```

最后，来串联整合稍微有点复杂的删除文件操作，这是因为设备驱动程序没有对应的功能派发函数，所以我们需要用到设备驱动程序的控制功能派发函数，代码如下所示。

[复制代码](#)

```
1 drvstus_t rfs_ioctl(device_t* devp, void* iopack)
2 {
3     objnode_t* obp = (objnode_t*)iopack;
4     //根据objnode_t结构中的控制码进行判断
5     if(obp->on_ioctrd == FSDEV_IOCTLRCD_DELFIL)
6     {
7         //调用删除文件操作的接口函数
8         return rfs_del_file(devp, obp->on_fname, 0);
9     }
10 }
```

```
9     }
10     return DFCERRSTUS;
11 }
```

上述代码中，我们给文件系统设备分配了一个 `FSDEV_IOCTLCD_DELFILE`（一个整数）控制码，Cosmos 内核上层组件的代码就可以根据需要，设置 `objnode_t` 结构中的控制码就能达到相应的目的了。


现在，文件相关的操作已经串联整合好了。

测试

前面实现了文件系统的 6 种最常用的文件操作，并且已经整合到文件系统设备驱动程序框架代码中去了，可是这些代码究竟对不对，测试运行了才知道。

下面来写好测试代码。要注意的是，Cosmos 下的任何设备驱动程序**都必须要有 `objnode_t` 结构才能运行**。所以，在这里我们需要手动建立一个 `objnode_t` 结构并设置好其中的字段，模拟一下 Cosmos 上层组件调用设备驱动程序的过程。

这一过程我们可以写个 `test_fsys` 函数来实现，代码如下所示。

 复制代码

```
1 void test_fsys(device_t *devp)
2 {
3     kprint("开始文件操作测试\n");
4     void *rwbuf = new_buf(FSYS_ALCBLKSZ); //分配缓冲区
5     //把缓冲区中的所有字节都置为0xff
6     hal_memset(rwbuf, 0xff, FSYS_ALCBLKSZ);
7     objnode_t *ondp = krlnew_objnode(); //新建一个objnode_t结构
8     ondp->on_acsflgs = FSDEV_OPENFLG_NEWFILE; //设置新建文件标志
9     ondp->on_fname = "/testfile"; //设置新建文件名
10    ondp->on_buf = rwbuf; //设置缓冲区
11    ondp->on_bufsz = FSYS_ALCBLKSZ; //设置缓冲区大小
12    ondp->on_len = 512; //设置读写多少字节
13    ondp->on_ioctlrd = FSDEV_IOCTLCD_DELFILE; //设置控制码
14    if (rfs_open(devp, ondp) == DFCERRSTUS) { //新建文件
15        hal_sysdie("新建文件错误");
16    }
17    ondp->on_acsflgs = FSDEV_OPENFLG_OPEFILE; //设置打开文件标志
18    if (rfs_open(devp, ondp) == DFCERRSTUS) { //打开文件
19        hal_sysdie("打开文件错误");
20    }
```

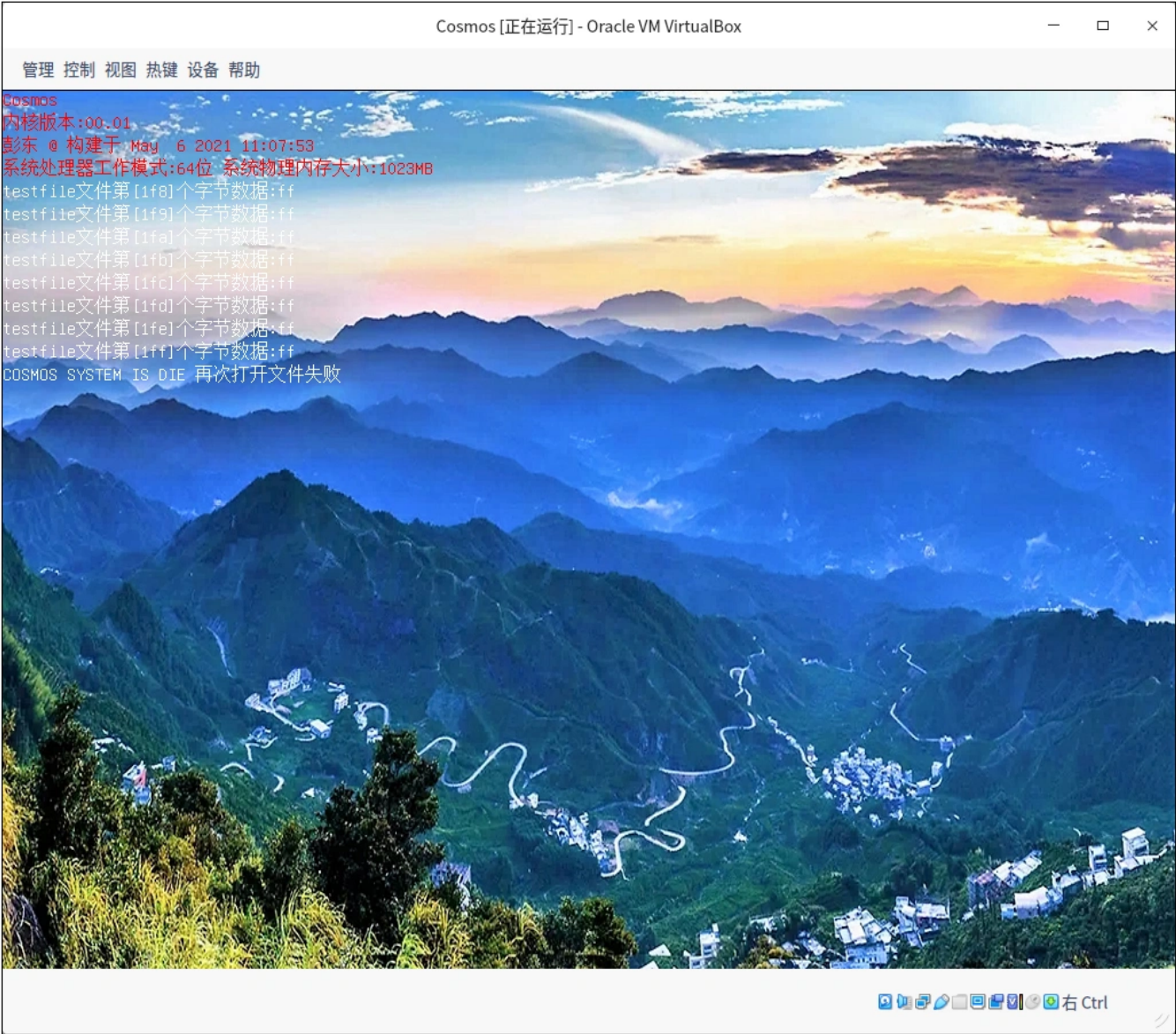


```
21     if (rfs_write(devp, ondp) == DFCERRSTUS) { //把数据写入文件
22         hal_sysdie("写入文件错误");
23     }
24     hal_memset(rwbuf, 0, FSYS_ALCBLKSZ); //清零缓冲区
25     if (rfs_read(devp, ondp) == DFCERRSTUS) { //读取文件数据
26         hal_sysdie("读取文件错误");
27     }
28     if (rfs_close(devp, ondp) == DFCERRSTUS) { //关闭文件
29         hal_sysdie("关闭文件错误");
30     }
31     u8_t *cb = (u8_t *)rwbuf; //指向缓冲区
32     for (uint_t i = 0; i < 512; i++) { //检查缓冲区空间中的头512个字节的数据，是否为0x
33         if (cb[i] != 0xff) { //如果不等于0xff就死机
34             hal_sysdie("检查文件内容错误");
35         }
36         kprint("testfile文件第[%x]个字节数据:%x\n", i, (uint_t)cb[i]); //打印文件内
37     }
38     if (rfs_ioctl(devp, ondp) == DFCERRSTUS) { //删除文件
39         hal_sysdie("删除文件错误");
40     }
41     ondp->on_acsflgs = FSDEV_OPENFLG_OPEFILE; //再次设置打开文件标志
42     if (rfs_open(devp, ondp) == DFCERRSTUS) { //再次打开文件
43         hal_sysdie("再次打开文件失败");
44     }
45     hal_sysdie("结束文件操作测试");
46     return;
47 }
```

上述代码虽然有点长，因为我们一下子测试了关于文件的 6 大操作。每个文件操作失败后都会死机，不会继续向下运行。

测试逻辑很简单：开始会建立并打开一个文件，接着写入数据，然后读取文件中数据进行比较，看看是不是和之前写入的数据相等，最后删除这个文件并再次打开，看是否会出错。因为文件已经删除了，打开一个已经删除的文件自然要出错，出错就说明测试成功。

现在我们把 test_fsys 函数放在 rfs_entry 函数的最后调用，然后打开终端切换到 cosmos 目录下执行 make vboxtest 命令，最后不出意外的话，你会看到如下图所示的情况。



文件操作测试示意图

从图里我们能看到，文件中的数据 and 最后重新打开已经删除文件时出现的错误，这说明了我们的代码是正确无误的。

至此，测试了文件相关的 6 大操作的代码，代码质量都是相当高的，都达到了我们的预期，一个简单、有诸多限制但却五脏俱全的文件系统就实现了。

重点回顾

这节课告一段落，恭喜你坚持到这里。

文件系统虽然复杂，但我们发现只要做得足够“小”，就能大大降低了实现的难度。虽然降低了实现的难度，但我们的 rfs 文件系统依然包含了一个正常文件系统所具有的功能特性，现在我来为你梳理一下本节课的重点：

1. 首先是文件系统的辅助操作，因为文件系统的复杂性，所以必须要实现一些如获取与释放根目录文件、获取文件名、判断文件是否存在等基础辅助操作函数。
2. 然后实现了文件系统必须要提供的 6 大文件操作：**新建文件、删除文件、打开文件、读写文件、关闭文件**。
3. 最后把这些文件操作全部串联整合到文件系统设备驱动程序之中，并且进行了测试，确认代码正确无误。

今天这节课，我们又实现了 Cosmos 内核的一个基础组件，即文件系统，不过它是以**设备的形式**存在的，这样做是为了方便以后的扩展和移植。

现在文件系统是实现了，不过还不够完善。你可能在想，我们文件系统在内存中，一断电数据就全完了。是的，不过你可以尝试写好硬盘驱动，然后把内存中的逻辑储存块写入到硬盘中就行了，期待你的实现。

思考题

请你想一想，我们这个简单的、小的，却五脏俱全的文件系统有哪些限制？

欢迎你在留言区记录你的收获或疑问，也鼓励你边学边练，多多动手实践。同时我推荐你把这节课分享给身边的朋友，跟他一起学习进步。

好，我是 LMOS，我们下节课见。

分享给需要的人，Ta订阅后你可得 **20** 元现金奖励

 赞 1  提建议

© 版权归极客邦科技所有，未经许可不得传播售卖。页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 33 | 仓库划分：文件系统的格式化操作

下一篇 35 | 瞧一瞧Linux：虚拟文件系统如何管理文件？

更多学习推荐

极客时间 | 训练营

大厂面试必考 100 题

2021 最新版 | 算法篇

限量免费领取  仅限前 99 名



精选留言 (2)

 写留言



LDxy

2021-07-26

好像还缺少seek操作

展开 

作者回复: 对，没有实现



 2



pedro

2021-07-26

限制1：不可持久化，不支持crash恢复，应加入磁盘块的写入，内存中有一定文件块的缓存，支持日志，防止系统崩溃，文件数据丢失。

限制2：缺少抽象层，无法支持多种格式的文件。

限制3：小量内存式文件系统，没有使用磁盘，不支持 mount 等骚操作。

等等.....

展开 

作者回复: 哈哈是的，你有能力可以写个硬盘驱动，在rfs这个驱动中，将IO包继续下发给硬盘驱动，让硬盘驱动写入到硬盘，由此，有层层驱动堆叠的IO栈就形成了



1