FatFS Version 0.01 源码注释

2014.6.17

作者：云鹤

QQ:　348597140

博客：**http://www.cnblogs.com/amanlikethis/**

概　述

笔者在学习 FatFs 文件系统的时候，对文件系统有很多的困惑。于是，在网上搜寻资料，逛了很多论坛，看到一位网友说：“现在的人太浮躁，不能沉下心来，有问题就发帖求助。FatFs0.01 的源码才一千多行，都懒得去看。”诚然，我被这位网友说中，所以开始了源码注释之旅。

本文写作有两个目的：其一，是对自己学习的总结，其二是希望本文能给其他朋友一些帮助。这些帮助不仅仅是对 FatFs 源码的理解，而且还有其他的方面。比如说写文档作总结，如何利用 Microsoft Visio 画图，如何对源码规范的注释。当然笔者水平有限，无论是 FatFs 源码剖析，还是文档编辑、注释都还有缺陷，但笔者希望能给各位带来有益的收获。笔者将附上本文的 Word、Pdf 以及文中用到的 Microsoft Visio 图表，最重要的还有源码注释文件。

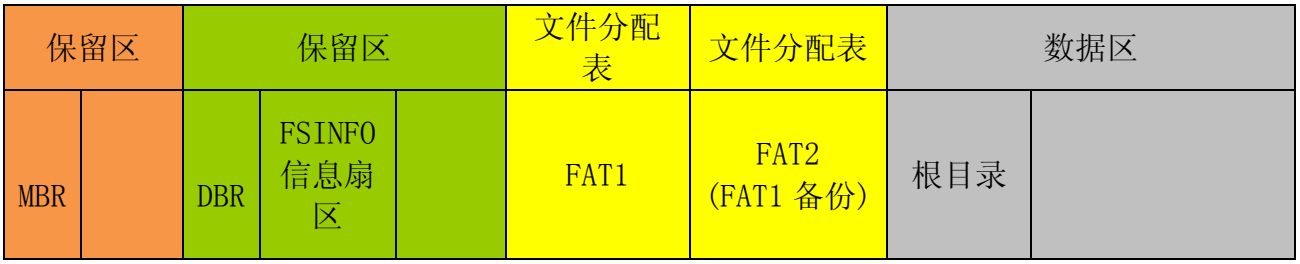
本文的写作参考了博客 http://blog.csdn.net/hexiaolong2009 的博主何小龙网友的文章，很感谢他的《FatFs 0.01 学习笔记》给我提供的启发。

一、实现的函数功能简述

1、 FRESULT f_open (FIL*, const char*, BYTE);
函数功能：打开或者创建一个文件
2、FRESULT f_read (FIL*, BYTE*, WORD, WORD*);
函数功能：读一个文件
3、FRESULT f_close (FIL*);
函数功能：关闭一个文件
4、FRESULT f_lseek (FIL*, DWORD);
函数功能：移动文件的指针
5、FRESULT f_opendir (DIR*, const char*);
函数功能：读一个目录中的目录项
6、FRESULT f_readdir (DIR*, FILINFO*);
函数功能：读取目录的内容
7、FRESULT f_stat (const char*, FILINFO*);
函数功能：获取文件的状态
8、FRESULT f_getfree (DWORD*);
函数功能：获得可用簇的数量
9、FRESULT f_mountdrv ();
函数功能：初始化文件系统
10、FRESULT f_write (FIL*, const BYTE*, WORD, WORD*);
函数功能：写文件
11、FRESULT f_sync (FIL*);
函数功能：同步文件缓冲区的内容到磁盘中
12、FRESULT f_unlink (const char*);
函数功能：删除一个文件或者目录
13、FRESULT f_mkdir (const char*);
函数功能：创建一个目录
14、FRESULT f_chmod (const char*, BYTE, BYTE);
函数功能：更改文件的属性

二、FATFS 主要数据结构

1、FAT32 文件系统的结构



概念与分区功能简述：

MBR:

保存了磁盘的分区信息（分区的起始地址、分区大小、分区结束地址）

DBR：

保存了当前分区的详细参数（比如 FAT 表的位置、FAT 表的的大小、簇大小、扇区大小、根目录中最大目录项数等等）

FAT 表：

以簇的形式对数据区重新划分空间，在 FAT 表中建立了簇的使用情况，哪些已经被占用，哪些没有被占用；簇链的结构，也即是簇与簇之间的连接关系。

目录：

在目录中，存在众多的目录项，目录项记录了文件名、大小、起始地址等等。

目录项：

目录中的存储单位，记录了每一个文件或者目录的信息。

数据区：

数据区中纯粹的文件数据

扇区：

SD 卡最小的读写单位，512 字节，也就是说一次最少读取或者写入的数据是 512 字节。

☆ **说明**

<1> 有些 SD 卡格式化后，并没有 MBR 部分，而且 SD 卡格式化后磁盘上只存在一个磁盘分区。也就是说，即使 SD 卡上有 MBR 部分，在 MBR 的 DPT（硬盘分区表）中也只有一个分区记录。

<2> 位于数据区之前的可以称之为文件系统管理区，此区域是以物理扇区为单位进行管理的，数据区则是以簇为单位进行管理的。

笔者认为 1：

狭义上的文件专指普通文件，目录则是位于普通文件的上层，用于管理处在其中的文件或者目录栏。笔者认为广义上的文件包含”目录和普通文件”。这样说目录，表示目录实际上也是一个文件，只不过是一个管理文件信息的特殊文件。

可以说文件与目录既有区别，又有联系。普通文件在文件管理的时候，给文件设计了一个管理变量—文件指针，用于指示文件当前读取或者写入的位置，它是以字节为单位进行计算的；而目录其实也有一个管理读取或者写入的位置的变量—目录指针，它则是以目录项（32 字节）为单位进行计算的。

笔者认为 2：

文件 = 目录中的目录项、FAT 表、文件对应的数据区内容

<1> 查看磁盘信息就是查看 DBR 和 FAT 表

<2> 读取文件就是查看目录项、FAT 表和文件对应的数据区内容

<3> 写文件就是修改目录项、FAT 表、文件对应的数据区内容

2、FATFS 主要数据结构

① FATFS

功能：保存了 SD 卡和文件系统的信息，主要是记录了 DBR 中的信息

```
/* File system object structure */
typedef struct _FATFS {
    BYTE  fs_type;          // FAT 文件系统的类型
    BYTE  files;            // 当前操作的文件数
    BYTE  sects_clust;      // 每簇扇区数
    BYTEn_fats;             // FAT 表个数
    WORD    n_rootdir;      // 根目录中的目录项项数
    BYTE  dirtyflag;        // 当前存储在 win[]中的内容是否修改过

    BYTE  pad1;
    DWORD  sects_fat;       // 每个 FAT 表的扇区数
    DWORD  max_clust;       // 数据区最大簇数
    DWORD  fatbase;         //  FAT 起始扇区
    DWORD  dirbase;         // 根目录起始扇区
    DWORD  database;        // 数据区起始扇区
    DWORD  winsect;         // 保存在 win[]中的当前扇区地址
    BYTE  win[512];         // 目录和 FAT 分配表的缓冲区
} FATFS;
```

拥有参数 fats、sects_fat、fatbase、dirbase、database、sects_clust 就知道了文件系统的整个布局，就可以方便的访问文件系统的每一部分。

② DIR

功能：作为目录项的指针，既可以用于记录一个特定文件在目录中的位置，又可以用于记录在目录中当前目录项指针的位置（类似与文件指针）。

```
typedef struct _DIR {
    DWORD  sclust;      // 目录起始簇号
    DWORD  clust;       // 当前簇号
    DWORD  sect;        // 当前扇区地址（物理扇区地址）

    WORD    index;      // 当前索引（目录中的逻辑索引）
                        //需要强调一点：索引是从目录的开始地址算
                        //起，每 32Bytes 加 1，而且即使切换扇区和簇，
                        //index 也不会从 0 开始重新计数；只有当切换
                        //目录时，才会重新清零。
} DIR;
```

☆ **说明**

<1 > 记录特定文件在目录中的位置只需要 sect 和 index

<2> 用于记录目录的位置只需要 sclust;记录目录指针则需要 clust、sect、index。

□ **本文规定**

本文所说的目录指针指的是目录的当前目录项位置，有 clust、sect、index 构成完整的目录指针。

③ FIL

功能：记录普通文件（不是目录文件）的详细信息，比如文件对应的目录项位置，文件起始簇号，文件指针，文件大小等。

```
typedef struct _FIL {
    DWORD  fptr;            // 文件指针，从文件的起始地址开始，以字节为单位计算
    DWORD  fsize;           // 文件大小
    DWORD  org_clust;       // 文件起始簇号
    DWORD  curr_clust;      // 当前簇号
```

```
    DWORD   curr_sect;          // 当前扇区地址
#ifndef _FS_READONLY

    DWORD   dir_sect;           // 文件对应的目录项所在扇区号
    BYTE*   dir_ptr;            // 目录项在 win[]中的入口地址

#endif
    BYTE*   buffer;             // 指向文件读写缓冲区（512 字节）
    BYTE  flag;                 // 文件状态标识
    BYTE  sect_clust;           // 当前簇中剩余扇区数
} FIL;
```

☆ **说明**

<1> dir_sect、dir_ptr 记录了文件对应在目录中目录项的位置

<2> org_clust 记录了文件的起始簇号

<3> fptr 为文件指针，记录了文件当前读写的相对于开始处的偏移量（以字节为单位）

<4> curr_clust、curr_sect、sect_clust 实际上也是文件读写指针，只不过它记录的是物理偏移量，结合着 fptr 就可以在物理磁盘上确定文件指针的确切位置。

□ **本文规定**

本文所说的文件指针在不同的语境中有两种含义：广义的文件指针指 fptr、curr_clust、curr_sect、sect_clust，狭义的文件指针专指 fptr。

④ FILINFO

功能：常用于需要获取文件参数的函数中，该结构体用于保存文件的属性，比如说文件的大小、创建时间、文件名字等。

```
typedef struct _FILINFO {
    DWORD fsize;                // 文件大小
    WORD fdate;                 // 日期
    WORD ftime;                 // 时间
    BYTE fattrib;               // 属性
    char fname[8+1+3+1];        // 名字(8.3  格式)
} FILINFO;
```

☆ Fname 保存了文件的名字，但是是 8.3 格式的，这与它在目录项中存储的文件名不一样。

⑤ win[512]

位于 FATFS 结构体中，作为目录项或者 FAT 分配表的读写缓冲区。它不是某一个文件专有的缓冲区，而是整个文件系统的公共读写缓冲区。

<1>当读取 MBR、DBR 的内容时，就需要借助于这个系统缓冲区。

<2>当读写 FAT 表时，也需要将磁盘中 FAT 表的数据读取到该缓冲区中，或者将缓冲区中的内容写入到磁盘对应的扇区中。

<3>当读写某一文件的信息（而非文件的数据时），就需要在此缓冲区中操作。而读写另外一个文件的信息时，则需要将上一个文件在缓冲区中的内容视情况同步到磁盘中，然后加载此文件的目录项对应扇区内容到缓冲区 win[512]中。

<4> 文件包含对应的目录项和数据空间。目录项需要在 win[]中操作在第<3>点已经说明了，而文件的数据空间的操作，则是交给了用户缓冲区，用户通过用户缓冲区读写文件的内容。文件的数据空间，有时也会通过文件的 buffer 与用户空间打交道。这将在⑥中进行讲述。

<5> 目录的操作全部都是在这个缓冲区中操作的，应用程序层也不会为目录开辟数据空间，所以目录的数据缓冲空间就是在这个缓冲区。需要注意的是：目录（目录文件）包含目录项（记录于上一层目录中）和数据空间，它的数据空间又担当了保存目录项的功能。不管是目录的目录项，还是目录的数据空间全部使用 win[]缓冲区操作。

⑥ buffer

buffer 是一个指向 512 字节缓冲区的指针，位于 FIL 结构体中，也就相当于是 FIL 中有一个 512 字节缓冲区的成员。此 512 字节的缓冲区，是一个文件的专有缓冲区。用于当文件的读写没有按照 512 字节对齐的时候，作为架接在磁盘与用户读写缓冲区之间的临时缓冲区。

三、函数功能与实现详细分析

0、move_window

这个函数不是一个可供用户调用的函数，是一个静态函数，只能被文件系统其他函数所调用。之所以首先讨论这个函数是因为它是唯一能够操作 win[]（系统缓冲区）的函数，其他的函数要想操作 win[]，必须通过调用此函数实现。

<1> 函数原型

```
BOOL move_window (
    DWORD sector            /* Sector number to make apperance in the FatFs->win */
)
```

<2> 函数说明

@函数功能：win[]操作函数（DBR、FAT 表、目录项）

① 读取新的扇区内容到临时缓冲区 win[]

② 同步 win[]中的内容到磁盘

注意：

<1> 如果读取新的扇区号就是现在存储在 win[]中的扇区号，就什么也不操作

<2> 如果不同，则根据情况同步 win[]到磁盘中，并且将新扇区中的内容读取到 win[]中

<3> 如果 sector 为 0，则函数功能变为同步 win[]到磁盘中，不会读取 0 扇区的内容到 win[]

@输入参数：sector 要读取扇区的扇区号

@输出参数： 无

<3> 备注

此函数被下列函数所直接或者间接调用：

第一类：操作 FAT 表

① get_cluster

② put_cluster

③ remove_chain

④ create_chain

第二类：操作 MBR、DBR

⑤ check_fs

第三类：操作目录项所在扇区（目录的数据空间）

⑥ trace_path

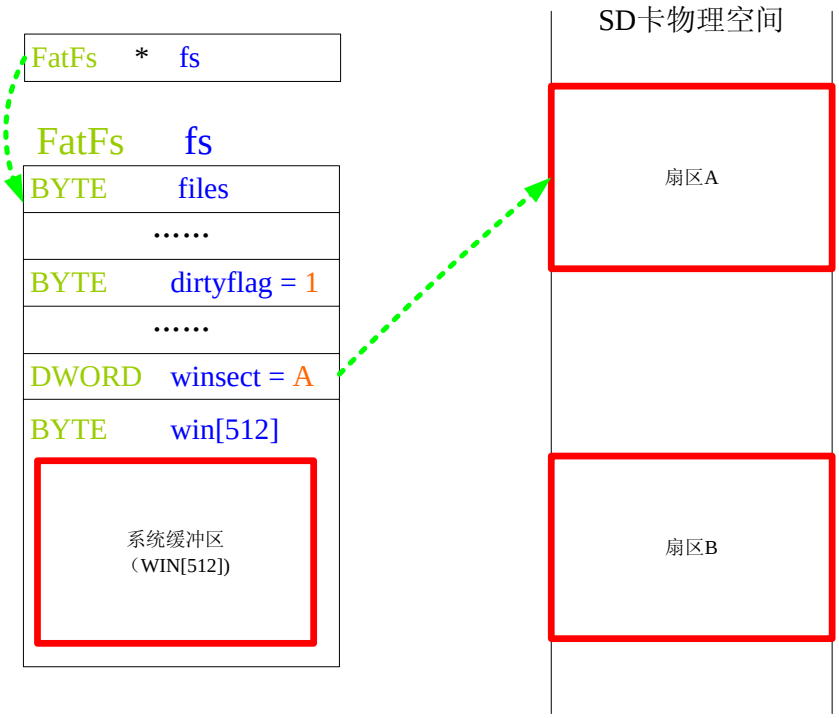
<4> 程序实现方法简述

首先判断要读取的扇区号是否与当前缓存在 win[]中的扇区号一致。倘若一致，则无需执行任何操作。倘若不一致，再判断缓存在 win[]中的内容是否被修改过，如果修改过，就需要更新到磁盘，最后还要把新扇区中的内容加载到 win[]中。

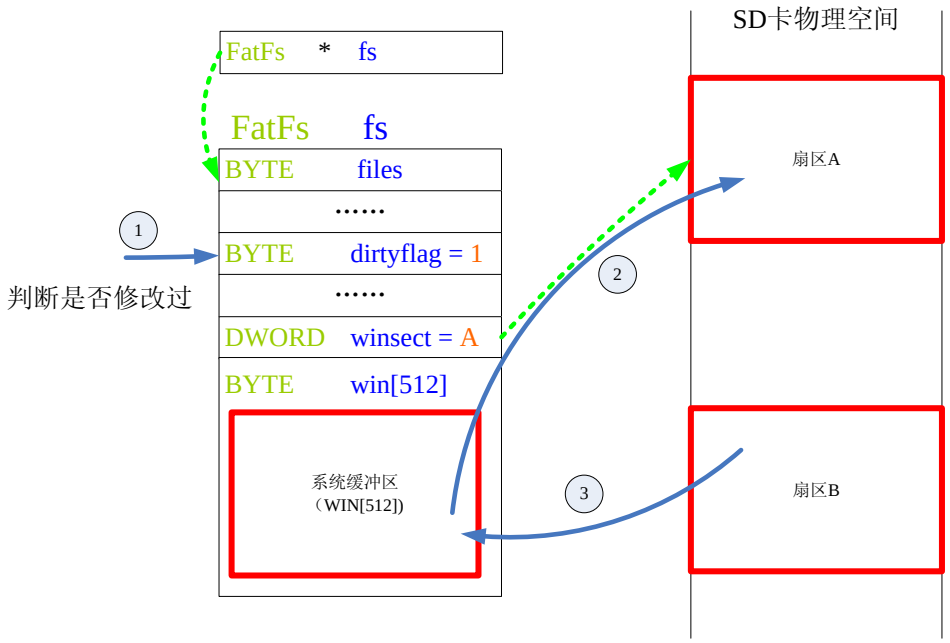
传入参数 0,0 与当前缓存在 win[]的扇区号肯定不一样，所以一定会同步 win[]内容到磁盘中。

<5> 程序执行示意图

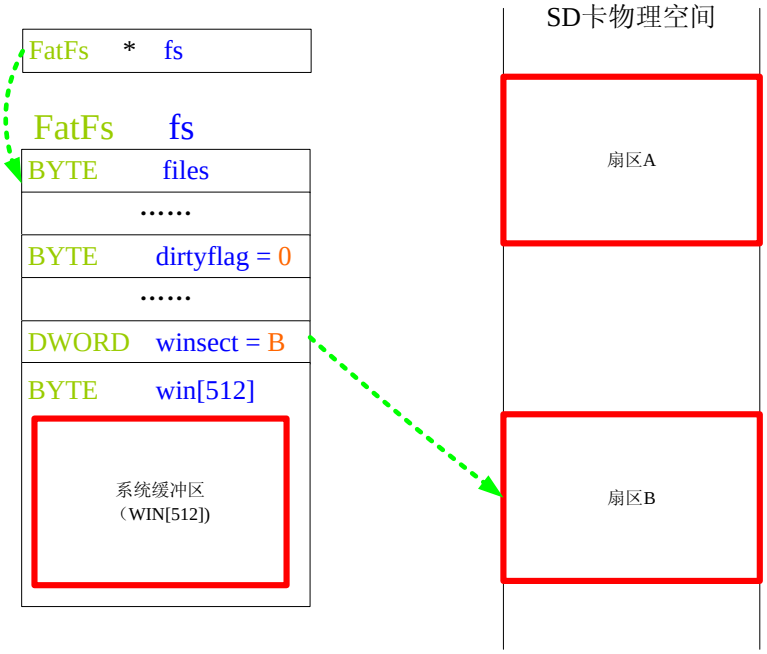
① 程序执行前



② 程序执行中



③ 程序执行后


FatFs * fs
FatFs fs
BYTE files
……
BYTE dirtyflag = 0
……
DWORD winsect = B
BYTE win[512]
系统缓冲区
（WIN[512])
SD卡物理空间
扇区A
扇区B

1、f_mountdrv

<1> 函数原型

FRESULT f_mountdrv ()

<2> 函数说明

@函数功能:

1.初始化 SD 卡

2、填充 FatFs 对象，即记录物理磁盘的相关参数

@输入参数 ：无

@输出参数： 无

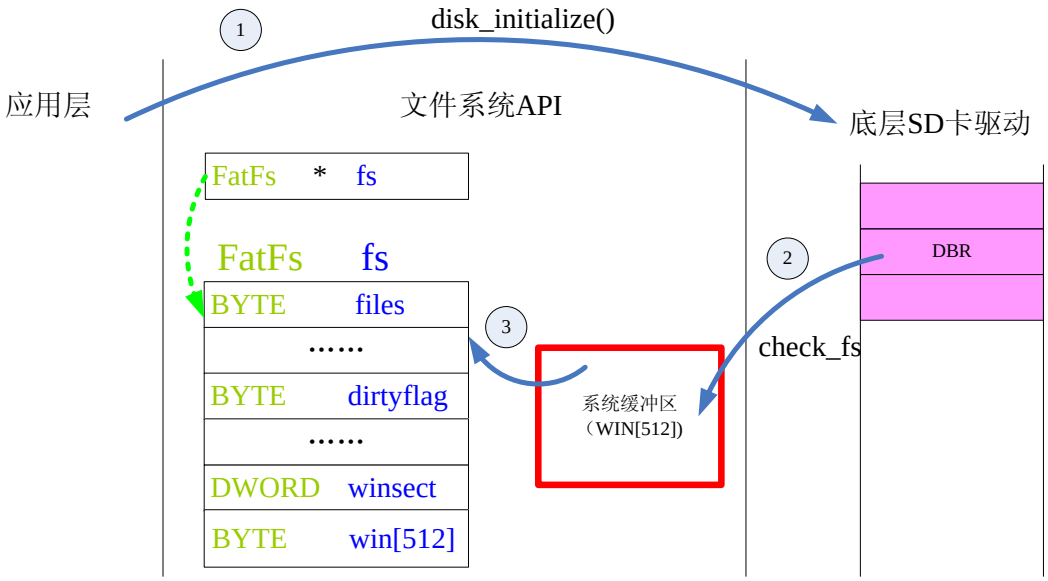
<3> 备注

<4> 程序实现方法简述

首先调用 SD 卡初始化函数，对 SD 卡进行初始化。然后读取物理磁盘 0 号扇区的内容，判断是否是 DBR 扇区。如果不是 DBR 扇区，那么肯定就是 MBR 扇区，再从 MBR 扇区中获取 DBR 扇区的地址，将 DBR 扇区的内容调取到 win[]中。

接下来从 win[]中，填充 FatFs 类型的系统对象，这样物理磁盘和文件系统的参数就被保存到了这个对象中。以后，程序就可以从全局变量--FatFs 类型的变量，访问文件系统的每一个区域。

<5> 程序执行示意图


1
disk_initialize()
应用层
文件系统API
底层SD卡驱动
FatFs * fs
FatFs fs
BYTE files
……
BYTE dirtyflag
……
DWORD winsect
BYTE win[512]
3
系统缓冲区
（WIN[512])
2
check_fs
DBR

2、f_open

<1> 函数原型

```
FRESULT f_open (
    FIL *fp,                /*  指向文件结构体变量  */
    const char *path,       /*  指向文件路径  */
    BYTE mode               /*  存取方式和打开方式  */
)
```

<2> 函数说明

@函数功能：以指定的方式打开或者新建一个文件。如果打开或者创建成功，
会填充 fp 指向的文件信息变量（包含文件的目录项确切位置和文件的信息）。

@输入参数：fp　　指向文件信息变量的指针
path　　指向文件的路径
mode　　打开方式

@输出参数：FR_OK　　打开或者创建成功
其他值　打开或者创建失败

<3> 备注

<4> 程序实现方法简述

① 以只读的方式打开一个已经存在的文件

首先调用函数 trace_path 搜索文件系统中是否存在目标文件，如果不存在就返回失败；如果存在就返回文件的目录项位置（dirscan、dir），并且将目录项所在扇区的内容加载到 win[]中。

接下来就是从 win[]中，将文件目录项的参数稍作转化后传入 FIL 类型的变量中。到此，一个文件就算完整的打开了。注意打开文件并不是打开文件的内容，而是文件的目录项，知道了文件的目录项就知道了如何去查看文件的内容。

以后，通过 FIL 类型的变量就可以操作对应的文件。

② 新建一个文件

首先调用函数 trace_path 搜索文件系统中是否存在目标文件，因为是新建文件肯定不存在。那么不存在的文件就返回新建文件当前文件夹的目录指针位置(dirscan、dir）--第一个空目录项所在位置，并且将当前目录指针所在扇区的内容加载到 win[]中。

首先给新建文件在当前文件夹中预定一个目录项位置，然后填入新建文件的目录项初始值（文件名、扩展名、属性、创建时间、更新时间）到 win[]中。注意这里并不会将新建文件目录项所在扇区同步到磁盘中，只有当调用 f_sync 函数时才会将文件的目录项所在扇区同步到磁盘。

创建一个新文件，只会在其上一层目录中添加对应的目录项并初始化，并不会给文件分配数据空间，当然文件的大小肯定是 0。

③ 重建一个文件

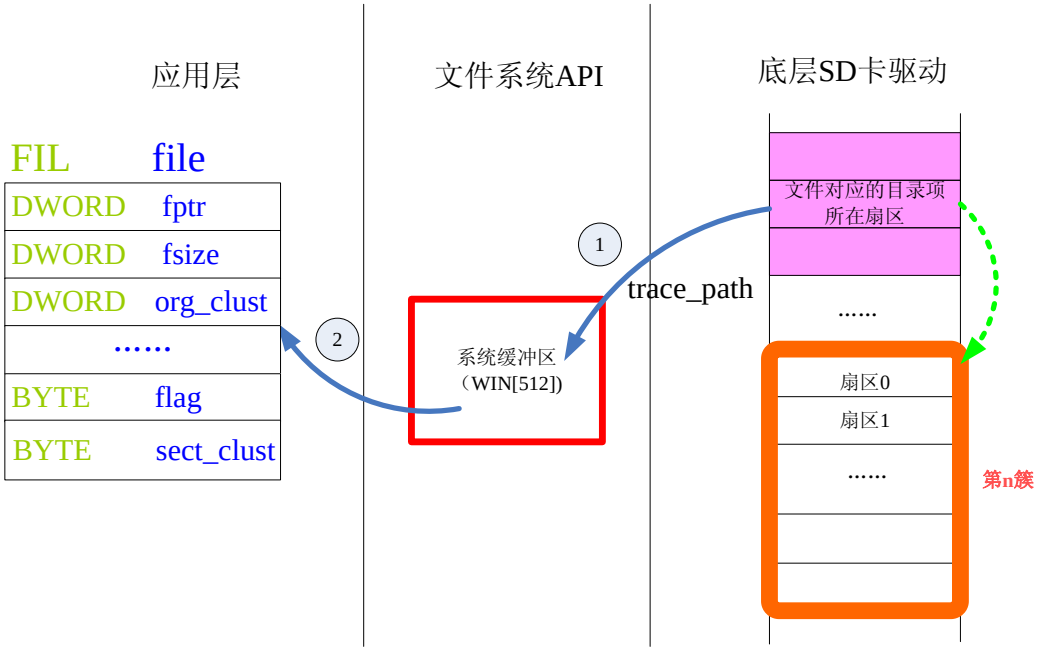
首先调用函数 trace_path 搜索文件系统中是否存在目标文件，因为是重建文件肯定存在。那么就返回文件的目录项位置（dirscan、dir），并且将目录项所在扇区的内容加载到 win[]中。

重建首先将文件的簇链删除，然后设置文件起始位置和文件大小为空，还需要初始化文件的属性、创建时间和修改时间。这里的修改都只是在 win[]中进行的，并没有同步到磁盘。只有当调用 f_sync 函数时才会将文件的目录项所在扇区同步到磁盘。

重建文件更改了原来文件在目录中的目录项信息，重建文件并没有分配簇，也就是没有分配数据空间。

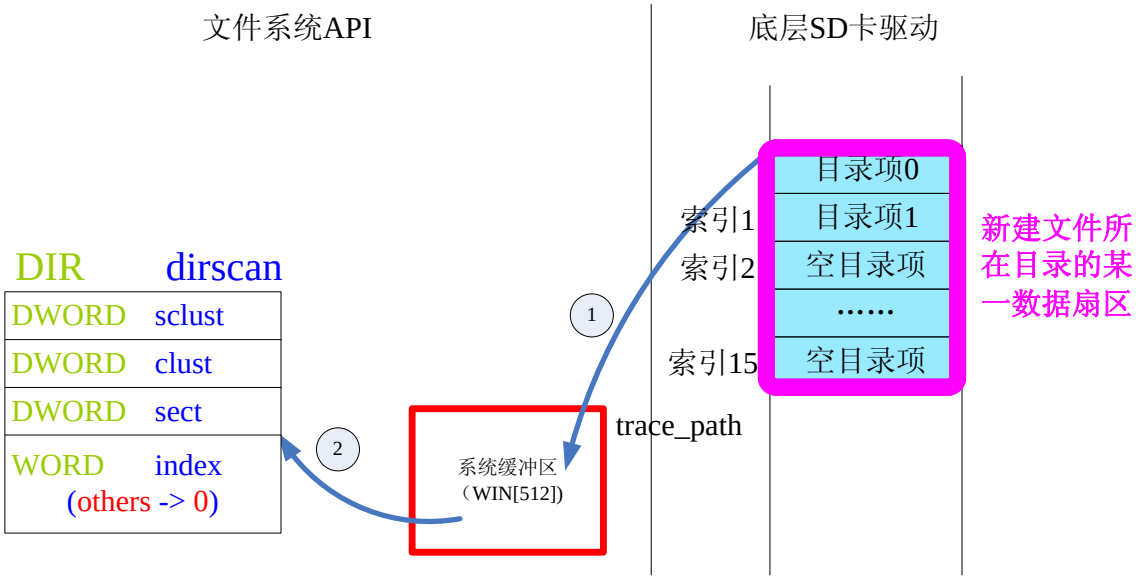
<5> 程序执行示意图

☆ 以只读的方式打开一个已经存在的文件

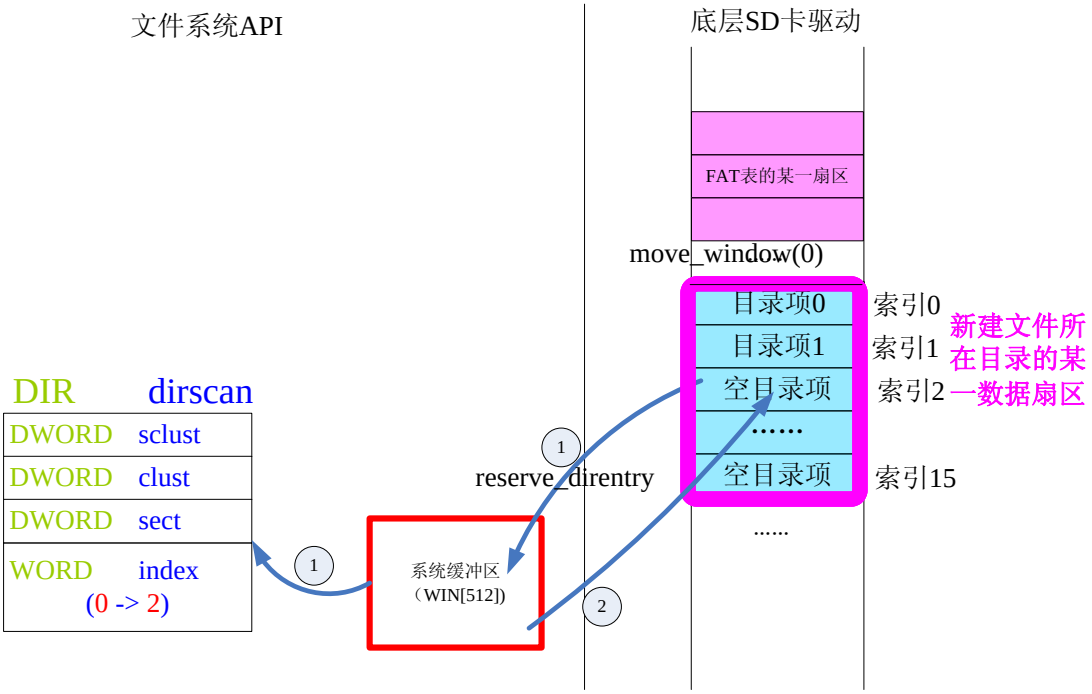


☆ 新建一个文件的过程

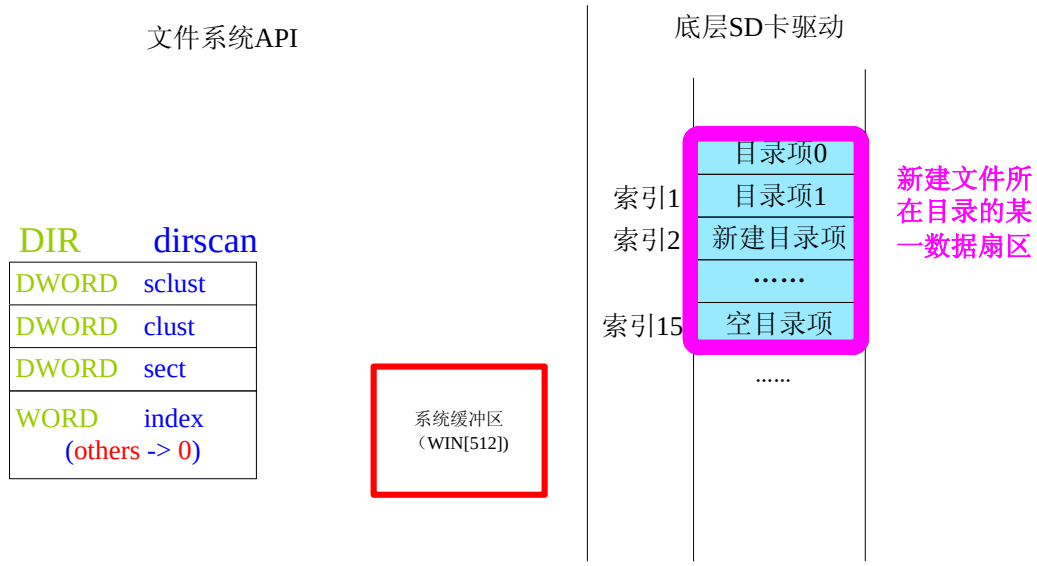
① 程序刚执行



② 程序执行中



③ 程序执行后


文件系统API
底层SD卡驱动
DIR dirscan
DWORD sclust
DWORD clust
DWORD sect
WORD index
(others -> 0)
系统缓冲区
（WIN[512])
目录项0
索引1
目录项1
索引2
新建目录项
……
索引15
空目录项
.....
新建文件所
在目录的某
一数据扇区

3、f_read

<1> 函数原型

```
FRESULT f_read (
    FIL *fp,        /* Pointer to the file object */
    BYTE *buff,         /* Pointer to data buffer */
    WORD btr,           /* Number of bytes to read */
    WORD *br            /* Pointer to number of bytes read */
)
```

<2> 函数说明

@函数功能 ：文件读操作
@输入参数：fp　　文件信息指针
　　buff 指向用户缓冲区
　　btr　　准备读取的字节数
　　br　　指向实际读取字节数的变量
@输出参数：FRESULT 成功与否

<3> 备注

此函数在读取文件内容后，还会移动文件指针到下一此读写操作的起点。

<4> 程序实现方法简述

读文件的情况有些复杂，不同的情况有不同的处理方法。在“<5>程序执行示意图”中，我展示了一种还算全面的情况，就以这种情况为例进行说明。开始读的时候，文件指针并没有位于扇区边界上（512 字节对齐），读取的跨度为 3 个簇。

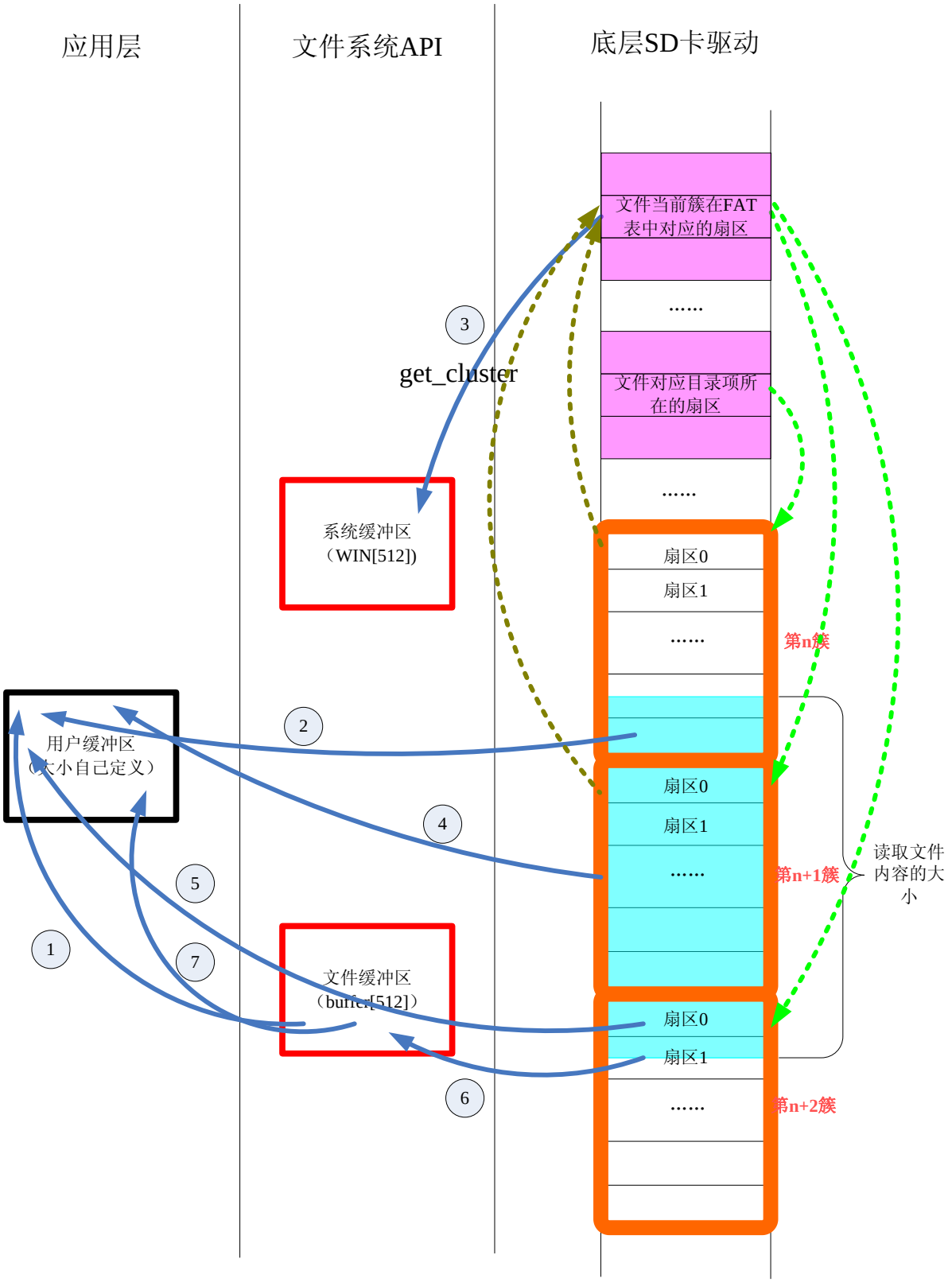
首先读没有对齐扇区的剩余内容，其实这个内容在以前的函数（以前的函数移动了文件指针）已经将这个扇区的内容加载到了 buffer 中。所以，直接从缓冲区 buffer 中读取此扇区文件指针以后的剩余内容到用户缓冲区。

接下来，读取第一个簇的剩余一个扇区的内容到用户缓冲区。通过 get_cluster 函数从 FAT 表中，获取第二个簇链的位置。然后一次性的将一个簇链的所有扇区内容读取到用户缓冲区中。再通过 get_cluster 函数从 FAT 表中，获取第三个簇链的位置。然后将第三个簇链的第一个扇区内容读取到用户缓冲区中。

最后，将最后所需要读取剩余内容所在的扇区(剩余部分不够一个扇区)读取到 buffer 中，然后再从 buffer 中读取所需要的剩余内容到用户缓冲区中。到这里为止，整个读取操作已经完成。

由于 buffer 中还有一部分内容没读，假设继续调用函数 f_read 函数读取数据，那么肯定先从这个 buffer 缓冲区中将文件指针以后的扇区剩余内容读取到用户缓冲区。

<5> 程序执行示意图


应用层
文件系统API
底层SD卡驱动
文件当前簇在FAT表中对应的扇区
.....
文件对应目录项所在的扇区
.....
扇区0
扇区1
.....
第n簇
扇区0
扇区1
.....
第n+1簇
扇区0
扇区1
.....
第n+2簇
读取文件内容的大小
3
get_cluster
系统缓冲区
（WIN[512])
用户缓冲区
（大小自己定义）
2
4
5
1
7
文件缓冲区
（buffer[512]）
6

4、f_write

<1> 函数原型

```
FRESULT f_write (
    FIL *fp,            /* Pointer to the file object */
    const BYTE *buff,   /* Pointer to the data to be written */
    WORD btw,           /* Number of bytes to write */
    WORD *bw            /* Pointer to number of bytes written */
)
```

<2> 函数说明

@函数功能 ：文件写操作，只对文件的数据区进行写入，并没有更新对应的目录项。
如果写入时，最后写入的数据字节没有完美的扇区对齐，那么肯定会将需要写入磁盘的一个扇区在文件缓冲区中进行缓存

@输入参数：fp 文件信息指针
buff 指向读取的用户缓冲区
btw 准备写入的字节数
bw 返回实际写入的字节数

@输出参数：FRESULT 成功与否

<3> 备注

此函数在写完文件内容后，还会移动文件指针到下一此读写操作的起点。

<4> 程序实现方法简述

写文件的情况与读取文件内容类似，不同的情况有不同的处理方法。在“<5>程序执行示意图”中，我展示了一个全面的情况，就以这种情况为例进行说明。开始写的时候，文件指针并没有位于扇区边界上（512 字节对齐），写入数据的跨度为 3 个簇。

首先写入没有对齐扇区的剩余内容，其实这个内容在以前的函数（以前的函数移动了文件指针）已经将这个扇区的内容加载到了 buffer 中。所以，将用户缓冲区中对应的内容写入到 buffer 中（从文件指针开始到 buffer 结束的这部分空间）。然后再将 buffer 中的内容写入到磁盘对应的扇区。

接下来，将用户缓冲区写入到第一个簇的剩余一个扇区中。通过 creat_chain 函数从 FAT 表中，获取第二个簇链的位置（如果是文件有剩余簇链则使用文件的剩余簇链，如果已经用完则重新从 FAT 表中搜索一个空的簇链连接到此文件中，也就是更改了文件的大小）。然后一次性的将用户缓冲区写入到第二个簇链的所有扇区中。再通过 get_cluster 函数从 FAT 表中，获取第三个簇链的位置。然后将用户缓冲区写入到第三个簇链的第一个扇区中。

最后，将最后所需要写入剩余内容所在的扇区(剩余部分不够一个扇区)读取到 buffer 中，然后再将用户缓冲区中剩余内容写入到 buffer 中。到这里为止，整个读取操作已经完成。注意这里并没有将 buffer 的内容写入到磁盘中。当调用 f_sync 函数的时候才会将 buffer 的内容同步到磁盘。

在函数返回之前，还需要判断文件大小是否更改了，如果大小更改了则要更新文件的大小，并将 FA__WRITTEN 记录到文件的 flag 中。这样做的目的是为了当执行 f_sync 时，可以根据 FA__WRITTEN 判断出文件修改过，从而更新文件的目录项。

☆ 假设

由于 buffer 中还有一部分内容没操作，

假设 1：继续调用函数 f_write 函数写入数据

那么肯定先将用户缓冲区的内容写入到这个 buffer 缓冲区中。只有超出了 buffer 缓冲区的范围，才会将这个 buffer 缓冲区的内容同步到磁盘，并且读取下一个扇区的内容到 buffer 中（假设文件指针仍然没有对齐）。

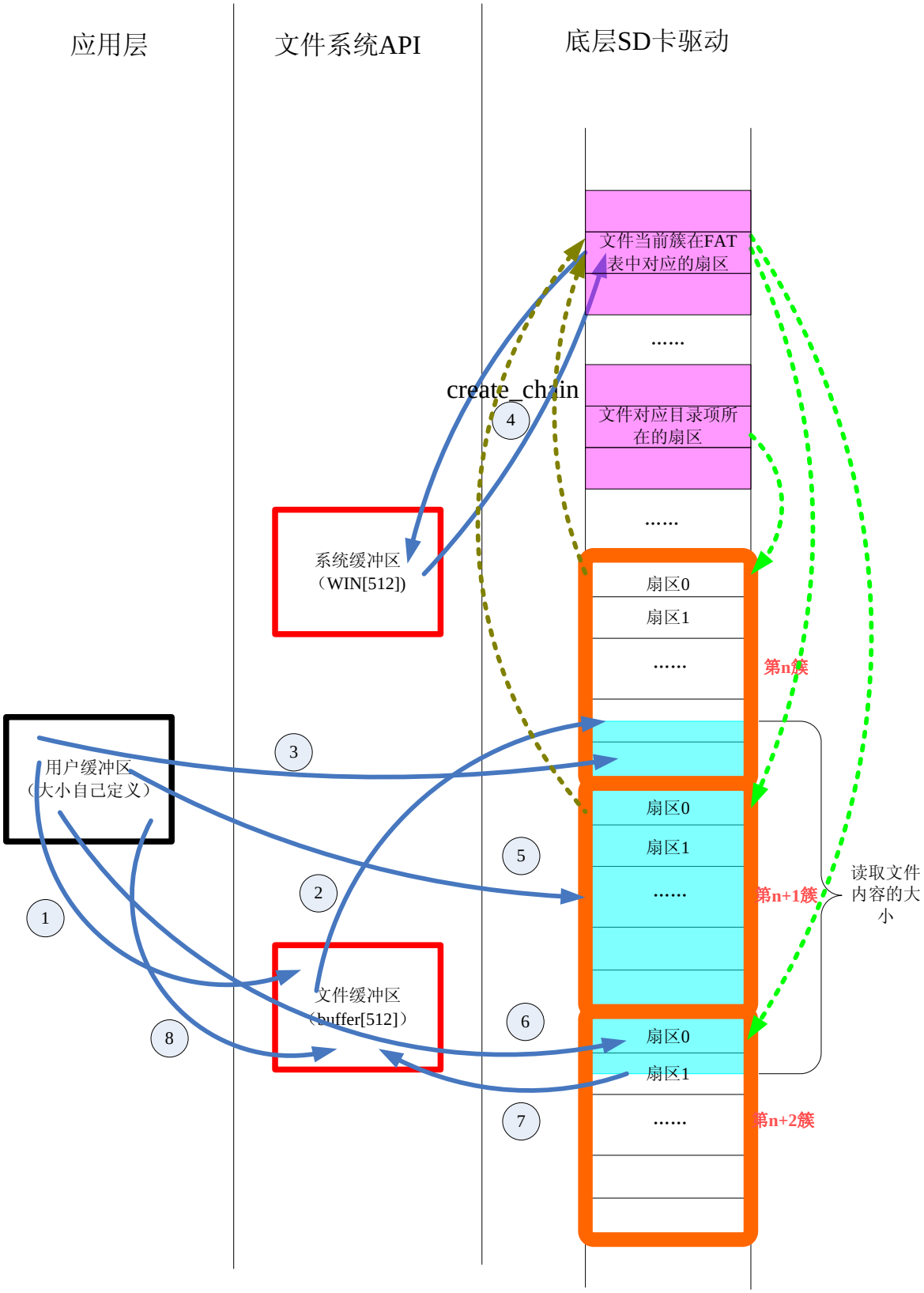
假设 2：调用函数 f_read 函数读取数据

先从这个 buffer 缓冲区中将文件指针以后的扇区剩余内容读取到用户缓冲区，而不会从磁盘中读取。

☆ 总结

buffer 的妙处，提高了读写的效率，避免了重复读写磁盘。

<5> 程序执行示意图



5、f_sync

<1> 函数原型

```
FRESULT f_sync (
    FIL *fp          /* Pointer to the file object */
)
```

<2> 函数说明

@函数功能 ：在关闭文件之前，同步文件缓冲区中的内容到磁盘，同步文件目录项信息到磁盘
@输入参数：fp 文件信息指针
@输出参数：FRESULT 成功与否

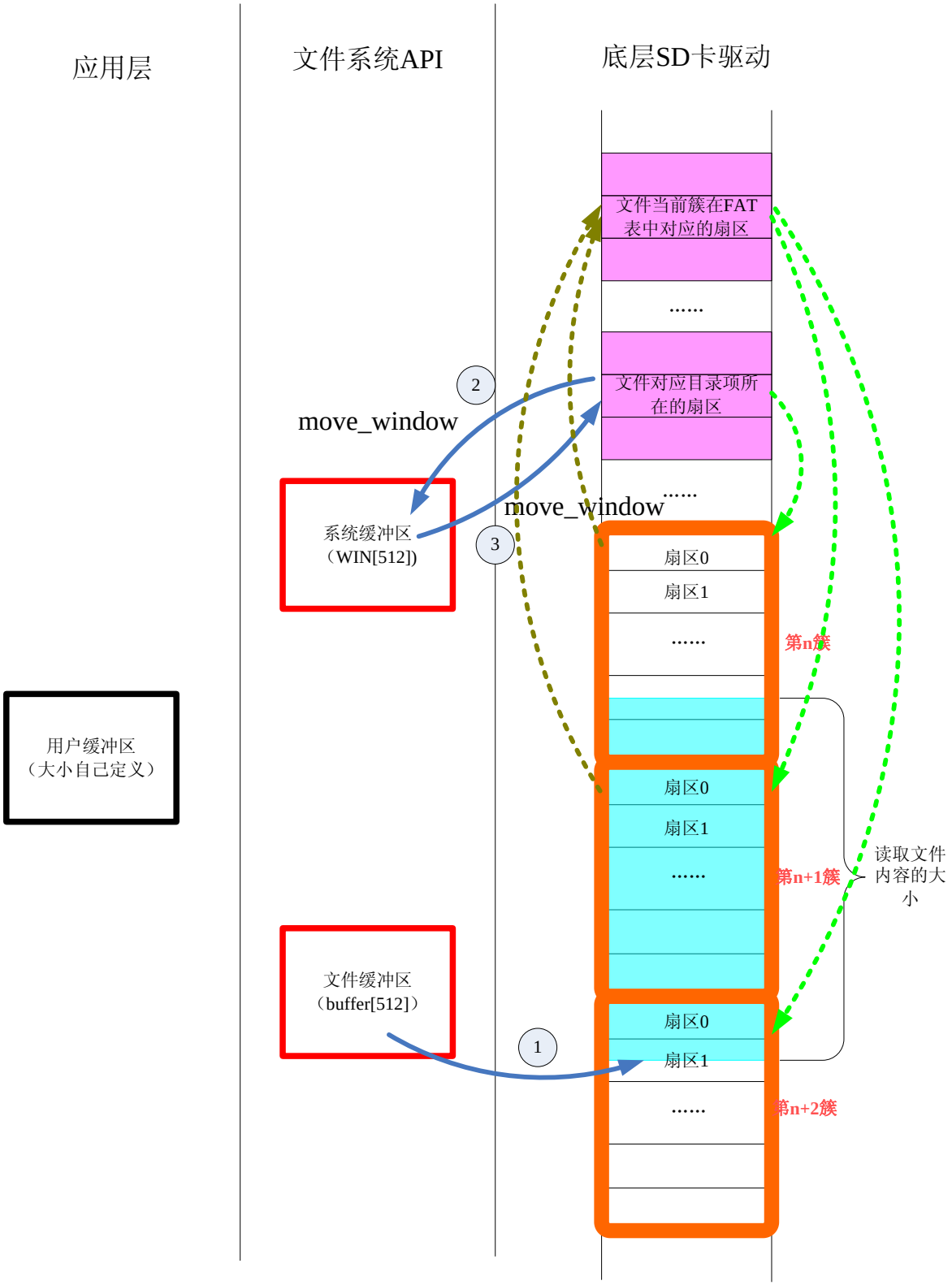
<3> 备注

<4> 程序实现方法简述

判断文件是否修改过，如果修改过再判断文件 buffer 缓冲区是否修改过，如果修改过则同步到磁盘中文件对应的数据空间中。如果文件修改过，还要更新文件的目录项，这时的修改也是在 win[]中的。

最后通过调用 move_window(0)，将文件目录项信息同步到磁盘中。

<5> 程序执行示意图


应用层
文件系统API
底层SD卡驱动
文件当前簇在FAT表中对应的扇区
.....
文件对应目录项所在的扇区
.....
2
move_window
move_window
系统缓冲区
（WIN[512])
3
扇区0
扇区1
.....
第n簇
扇区0
扇区1
.....
第n+1簇
读取文件内容的大小
扇区0
扇区1
.....
第n+2簇
用户缓冲区
（大小自己定义）
文件缓冲区
（buffer[512]）
1

6、f_opendir

<1> 函数原型

```
FRESULT f_opendir (
    DIR *scan,                  /* Pointer to directory object to initialize */
    const char *path    /* Pointer to the directory path, null str means the root */
)
```

<2> 函数说明

@函数功能 ：打开一个目录
@输入参数：scan：指向返回找到的目录项结构体
　　path　指向路径
@输出参数：FRESULT 成功与否

<3> 备注

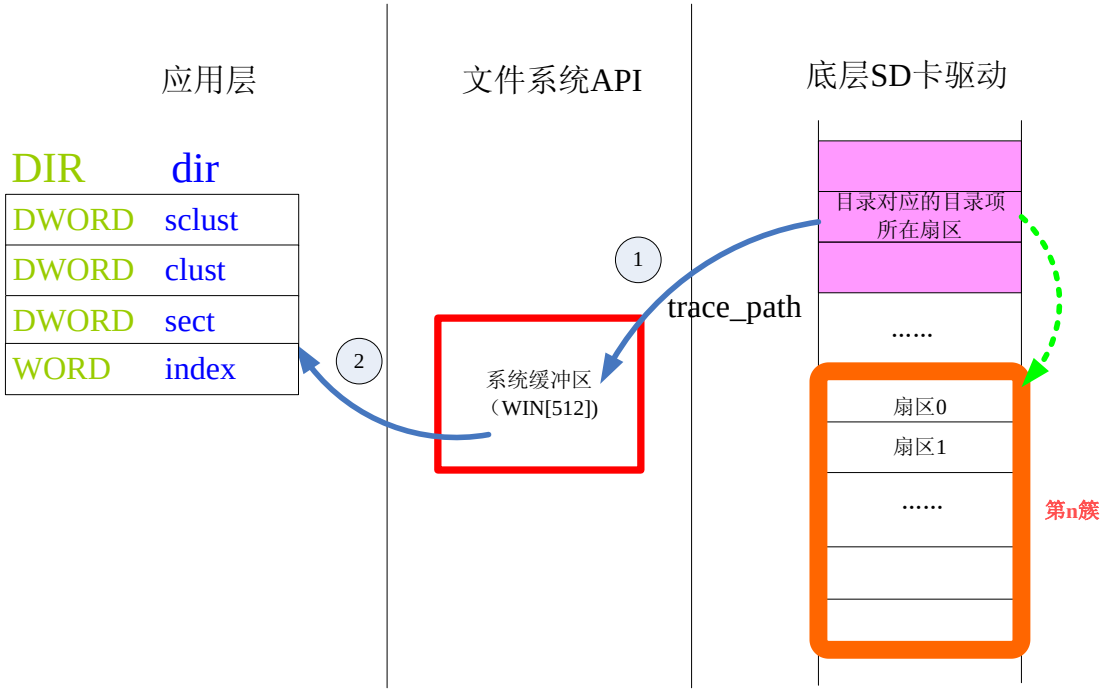
<4> 程序实现方法简述

首先调用函数 trace_path 搜索文件系统中是否存在所要打开的目录，如果不存在就返回失败；如果存在就返回目录对应目录项的位置（dirscan、dir），并且将目录对应目录项所在扇区的内容加载到 win[]中。

接下来判断找到的是不是一个目录。如果就是一个目录的话，就从 win[]中将目录对应目录项的参数稍作转化后传入 DIR 类型的变量中。到此，一个目录就算完整的打开了。注意打开目录并不是打开目录的内容，而是目录对应的目录项，知道了目录对应的目录项就知道了如何去查看目录的内容。

以后，通过 DIR 类型的变量就可以操作对应的目录。

<5> 程序执行示意图


应用层
文件系统API
底层SD卡驱动
DIR dir
DWORD sclust
DWORD clust
DWORD sect
WORD index
1
2
trace_path
系统缓冲区
（WIN[512])
目录对应的目录项
所在扇区
.....
扇区0
扇区1
.....
第n簇

7、f_mkdir

<1> 函数原型

```
FRESULT f_mkdir (
    const char *path        /* Pointer to the directory path */
)
```

<2> 函数说明

@函数功能 ：创建一个目录

注意：新建一个目录，它虽然是一个空目录（有效存储内容为 0），但是系统已经为它分配了一个簇的数据空间，用于保存它的目录项。这是与新建一个普通文件区别很大的地方。

另外，新建一个目录时，对新建目录在上一层目录的目录项以及新建目录中的目录项的初始化，全部都在 win[]中进行操作。

@输入参数：path 指向路径的指针

@输出参数：FRESULT 成功与否

<3> 备注

<4> 程序实现方法简述

首先调用函数 trace_path 搜索文件系统中是否存在目标目录，因为是新建目录肯定不存在。那么不存在目录时就返回新建目录所在当前文件夹的目录指针(dirscan、dir）--第一个空目录项位置，并且将当前目录指针所在扇区的内容加载到 win[]中。

接下来给新建目录在当前文件夹中预定一个目录项位置。然后调用 creat_chain 函数在 FAT 表中为新建目录找到一个可用的数据簇，再调用 move_window(0)同步 FAT 表到磁盘中。为新建目录的数据簇初始化，并且初始化第一个目录项。最后，填入新建目录的目录项初始值（目录名、属性、创建时间 、数据簇起始位置）到 win[]中。然后同步到磁盘中，完成整个新建目录的工作。

☆ 注意

<1> 创建一个新目录，不仅会在其上一层目录中添加对应的目录项并初始化，并且会给新建目录分配一个簇的数据空间，并进行初始化。

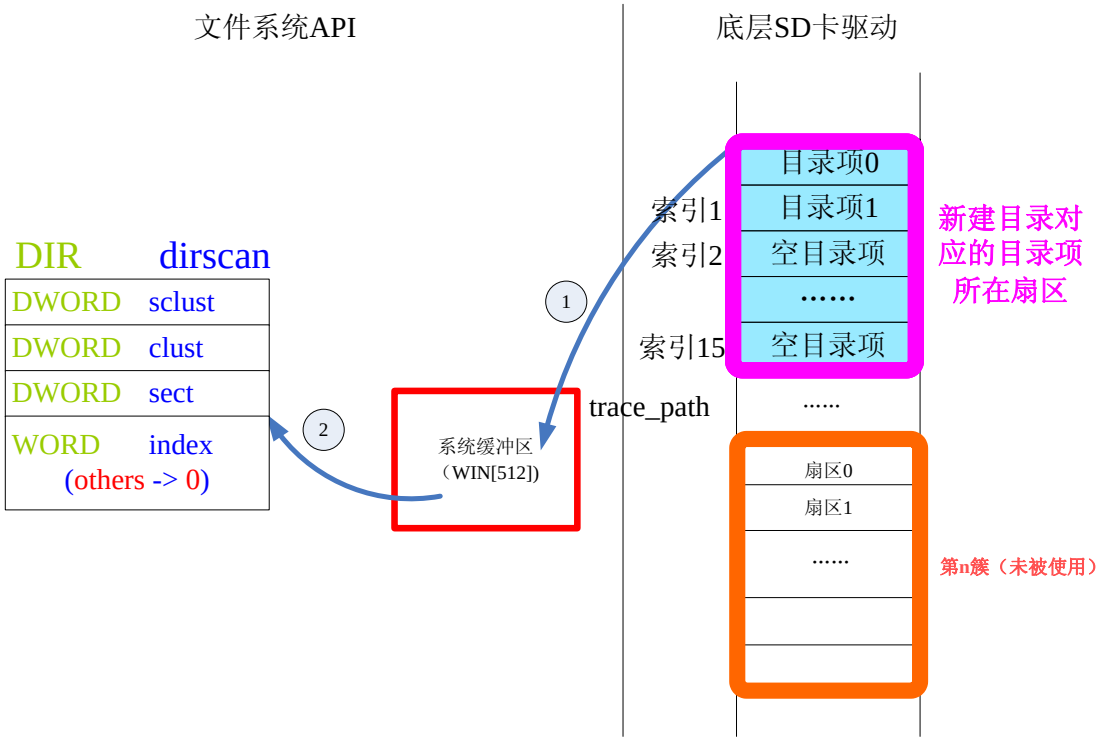
<2> 新建一个目录时，会将新建目录的数据簇和对应目录项所在扇区都同步到磁盘中，这与文件必须通过调用 f_sync 才能同步是不一样的。

<3> 新建一个目录会给目录分配数据空间，而新建文件则是没有的，这也是一个巨大的差别。

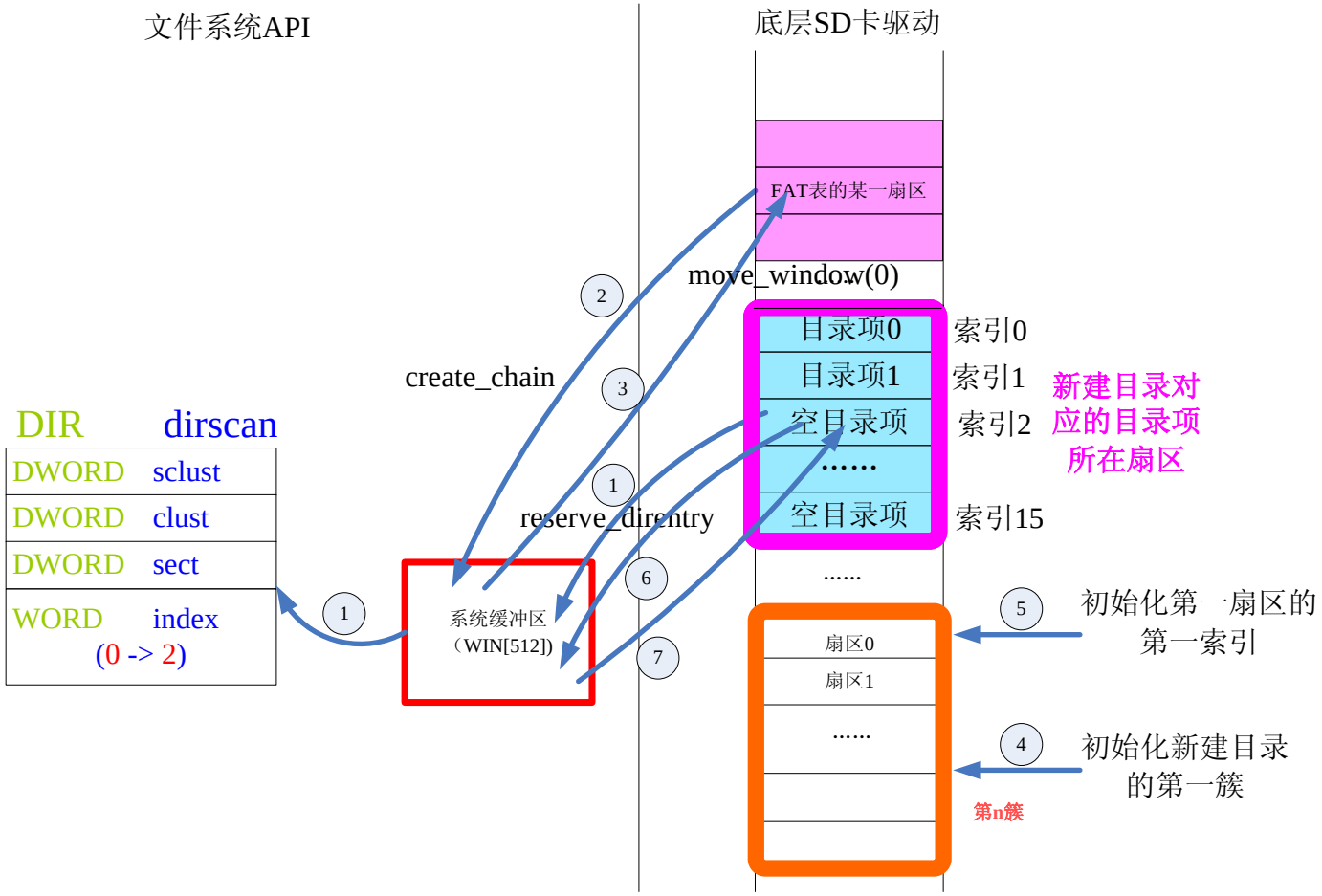
<4> 新建一个目录的所有操作都是在 win[]中进行的，不管是新建目录的对应目录项，还是新建目录的数据空间都是在 win[]中进行的。

<5> 程序执行示意图

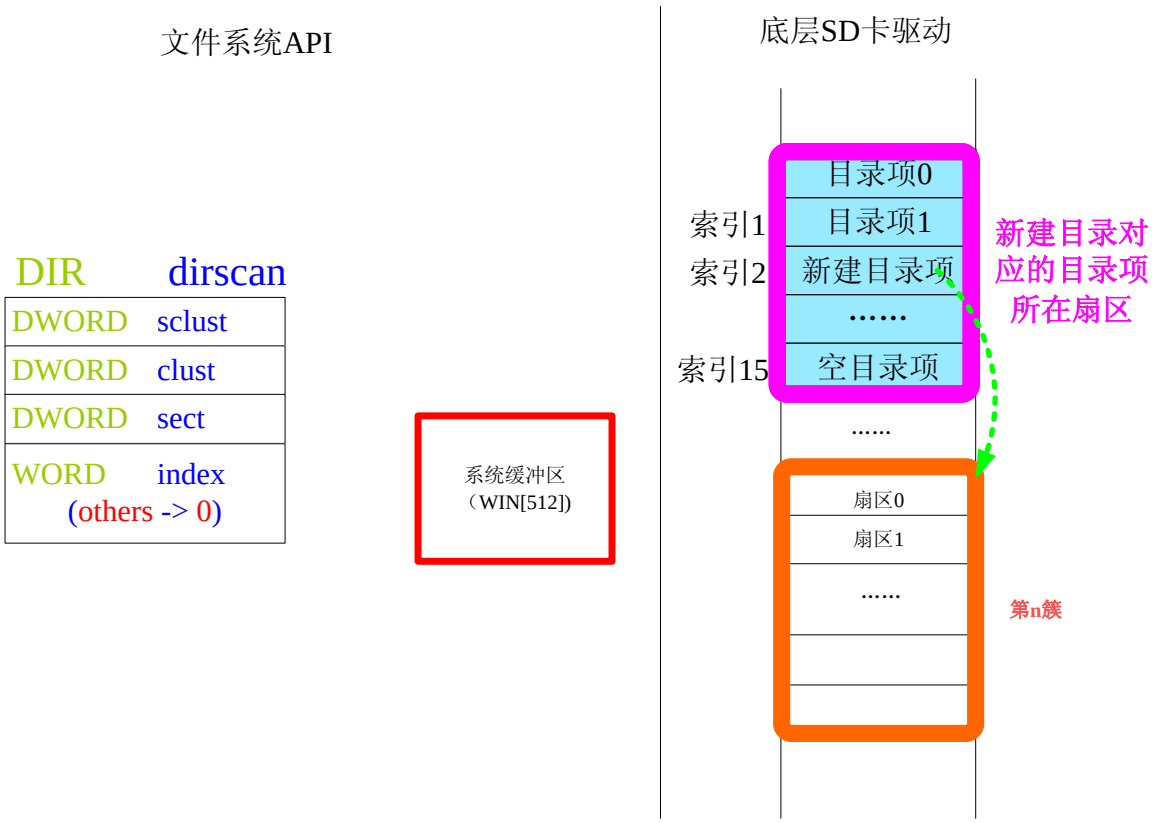
① 程序刚执行



②程序执行中



③ 程序执行后


文件系统API
底层SD卡驱动
DIR dirscan
DWORD sclust
DWORD clust
DWORD sect
WORD index
(others -> 0)
系统缓冲区
（WIN[512])
目录项0
索引1
目录项1
索引2
新建目录项
……
索引15
空目录项
……
新建目录对应的目录项所在扇区
扇区0
扇区1
……
第n簇

8、f_unlink

<1> 函数原型

```
FRESULT f_unlink (
    const char *path                /* Pointer to the file or directory path */
)
```

<2> 函数说明

@函数功能 ：删除一个文件或者目录

1、删除目录或者文件的簇链（回收数据空间）

2、文件或者目录的目录项被设置成为删除（0xE5），注意目录项并没有回收，只是标记为删除

@输入参数：path 指向路径的指针

@输出参数：FRESULT 成功与否

<3> 备注

<4> 程序实现方法简述

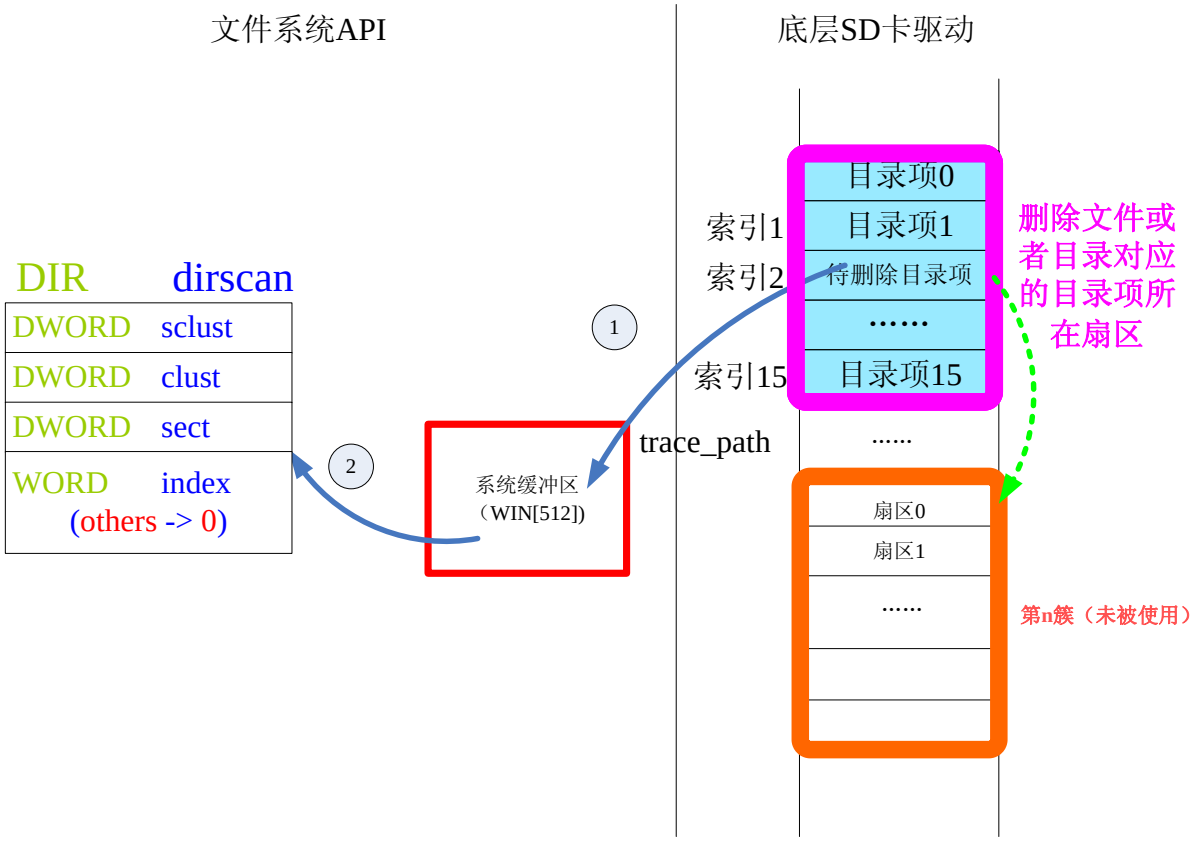
首先调用函数 trace_path 搜索文件系统中是否存在所要删除的目录或者文件，如果不存在就返回失败；如果存在就返回对应目录项的位置（dirscan、dir），并且将对应目录项所在扇区的内容加载到 win[]中。

判断要删除的是不是目录，如果是目录还要判断是不是非空目录，如果是非空目录则不允许删除。如果是空目录，那么就可以删除。

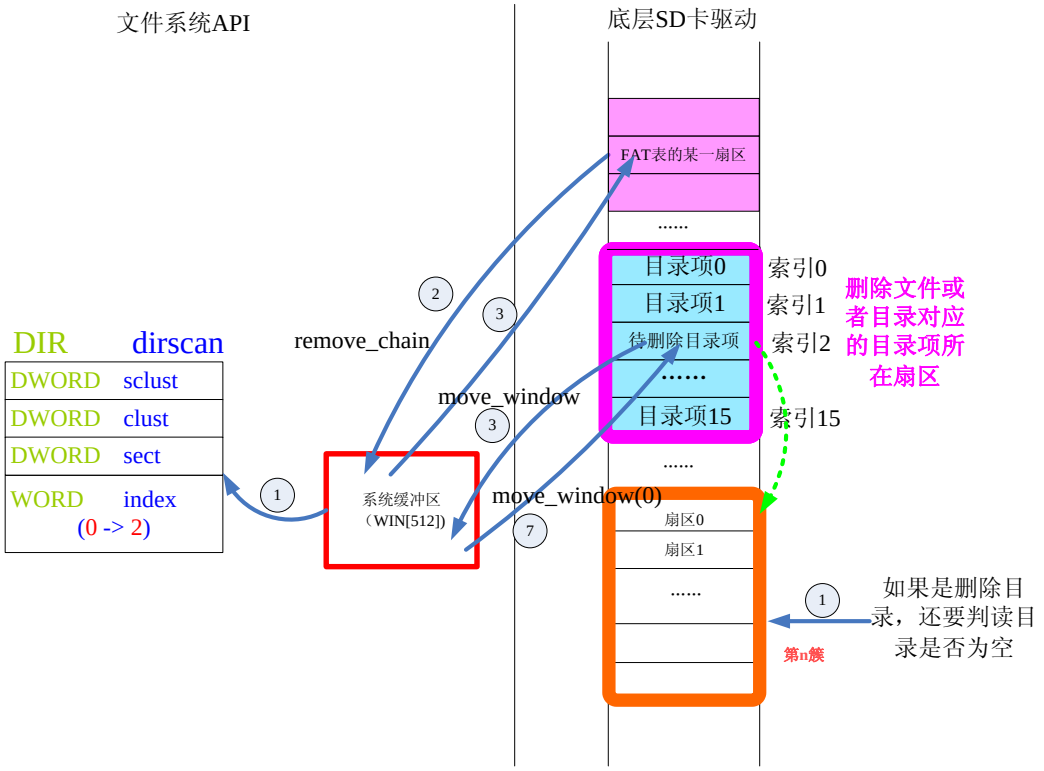
删除文件或者目录时，首先删除簇链（数据空间），然后修改目录项为删除状态（0xE5），最后同步目录项所在扇区 win[]缓冲区到磁盘中，完成删除。

<5> 程序执行示意图

① 程序刚执行



② 程序执行中



③ 程序执行后


文件系统API
底层SD卡驱动
DIR dirscan
DWORD sclust
DWORD clust
DWORD sect
WORD index
(others -> 0)
系统缓冲区
（WIN[512])
目录项0
索引1
目录项1
索引2
已删除目录项
……
索引15
目录项15
……
扇区0
扇区1
……
第n簇
删除文件或者目录对应的目录项所在扇区

9、f_lseek

<1> 函数原型

```
FRESULT f_lseek (
    FIL *fp,        /* Pointer to the file object */
    DWORD ofs       /* File pointer from top of file */
)
```

<2> 函数说明

@函数功能 ：移动文件指针，实际上就是修改文件指针（当前簇号、当前扇区号、文件指针 fptr）
@输入参数：fp 文件信息指针
ofs 定位文件指针的位置（从文件头部开始的偏移量）
@输出参数： fp 返回重新定位后的文件信息（包含文件指针）
FRESULT 成功与否

<3> 备注

<4> 程序实现方法简述

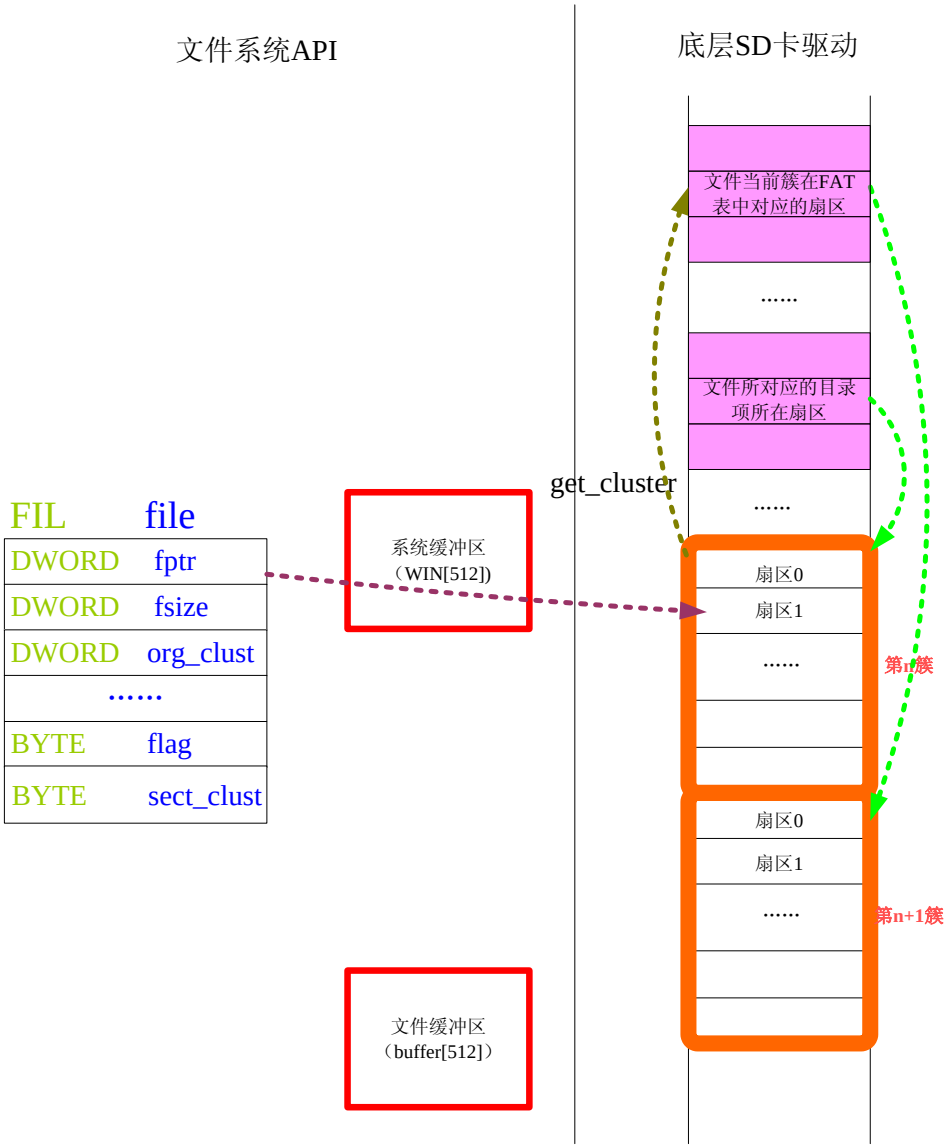
首先要对缓冲区 buffer 进行同步，因为文件指针中的 curr_sect 代表的是当前处在 buffer 中物理扇区号。现在要移动指针，也就是要移动当前扇区号 curr_sect，所以要先将 buffer 进行同步。

偏移量进行修正，因为可能偏移量超过了文件的大小，修正后的偏移量直接赋给 fptr。

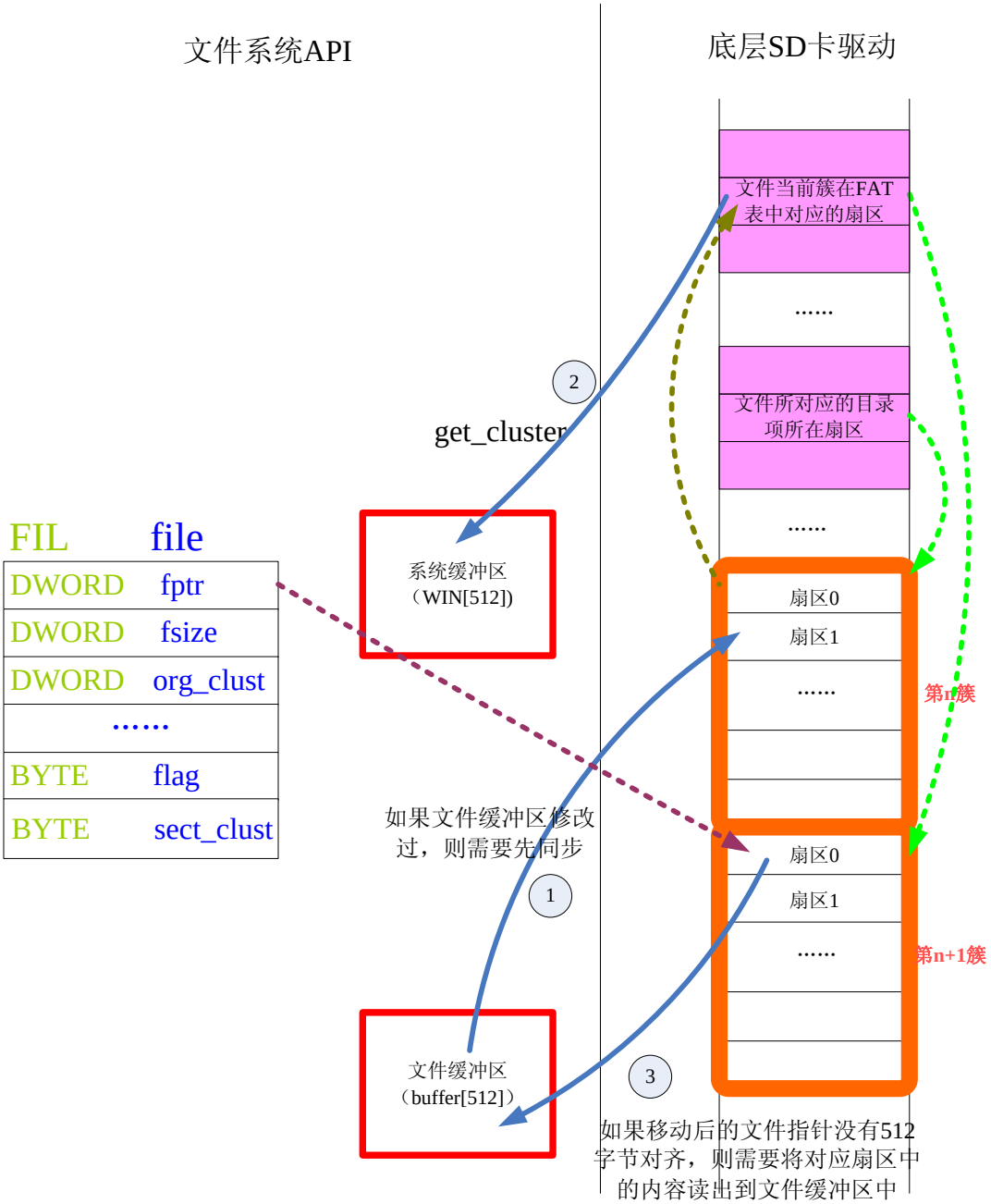
接下来根据偏移量，结合着当前的文件信息 FIL 类型的对象计算出移动指针后的簇号、扇区号。倘若移动后的文件指针没有 512 字节对齐，则还需要将 curr_sect 指向的物理扇区内容读取到 buffer 中。这样接下来的文件读写操作才不会出错。

<5> 程序执行示意图

① 程序执行前



② 程序执行中和程序执行后



10、f_readdir

<1> 函数原型

```
FRESULT    f_readdir (
      DIR *scan,                    /* Pointer to the directory object */
      FILINFO *finfo        /* Pointer to file information to return */
)
```

<2> 函数说明

@函数功能 ：从当前目录项指针处读取一个目录项，并且移动目录指针到下一个索引
@输入参数：scan：要读取的目录
finfo 目录的信息
finfo->fname[0] = 0 ：这是一个空目录项
finfo->fname[0] = others ：这是一个非空目录项
@输出参数：FRESULT 成功与否

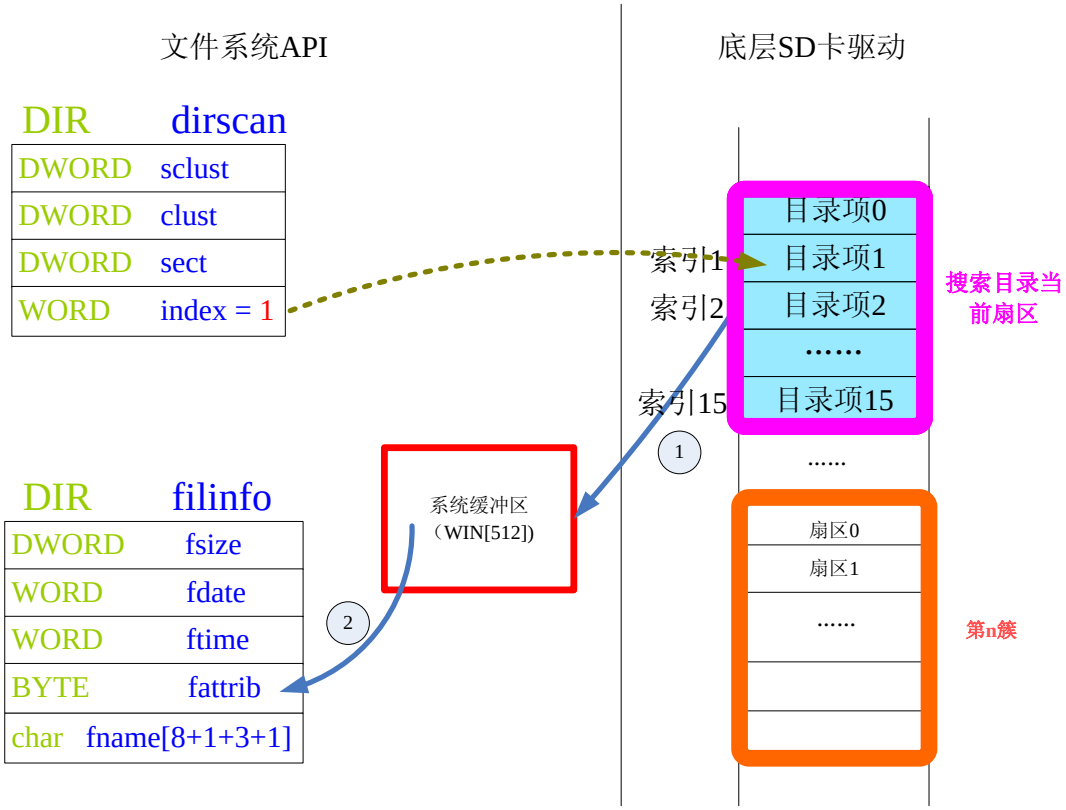
<3> 备注

<4> 程序实现方法简述

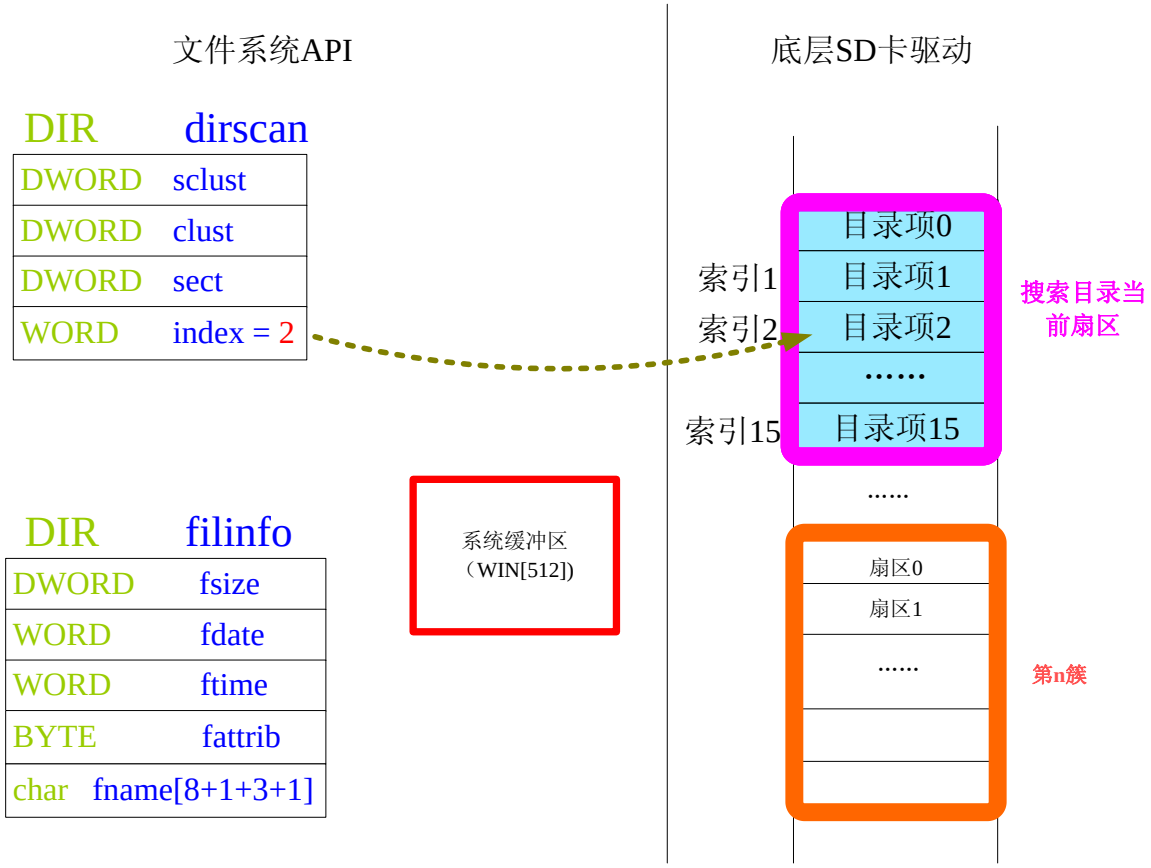
首先将目录指针当前所在物理扇区读取到 win[]中，然后调用 get_fileinifo 函数从当前目录指针处读取当前目录项并处理后存入 finfo 中。最后，还要移动目录项指针到下一个索引位置。

<5> 程序执行示意图

① 程序执行



② 程序执行后



四、FATFS 文件系统解惑

1、win[]和 buffer

文件系统用了两种重要的缓冲区 win[]和 buffer。这两种缓冲区的用途在第二章和第三章中已经阐述。但是深层次的思考，它们对应磁盘扇区的读写时机是什么？也即是说什么时候需要从磁盘中读取数据以更新缓冲区，又什么时候需要将缓冲区中的内容同步到磁盘。

常见的磁盘比如说 SD 卡、硬盘，它们的扇区大小 512 字节，对磁盘进行一次读写的字节数要是 512 的倍数，而非随意的几个字节都可以。

buffer 的魅力：

如果文件系统设计成这样：从当前文件指针处，读取 2 个字节到用户缓冲区，需要将磁盘对应的扇区读一次；再次读 2 个字节到用户缓冲区，又从磁盘的对应扇区读一次，那么效率肯定很慢，频繁的读取相同的扇区是一件很蠢的事。

在第二次读 2 个字节到用户缓冲区中的时候，可以看看需要的数据是否就处于 buffer 缓冲区中（第一次已经将一个扇区读如 buffer），如果在就直接从 buffer 中读，如果不在再从磁盘中读。显然，这样做可以避免重复读取相同的扇区，只有到了迫不得已的时候才会读取磁盘其他的扇区。这样做提高了系统的效率。

对于写文件时，buffer 的作用也是类似的：第一次写入 2 个字节，需要先将文件指针对应磁盘的扇区读入 buffer，然后通过用户缓冲区修改对应 2 个字节的内容；第二次写入 2 个字节的时候，如果写入的位置仍然在当前扇区的话，可以直接将用户缓冲区的内容替换掉接下来的 2 个字节的内容……如果迫不得已需要更换扇区，那将缓冲区中的内容同步到磁盘中，然后读入下一个扇区的内容到缓冲区中。

win[]的魅力：

操作 win[]最底层的函数 move_window，如果读取的扇区号不变那么没必要重新读取磁盘。如果读取扇区号改变了，先将 win[]同步到磁盘，然后读取另外一个扇区的内容到 win[]中。

这样的好处就是：假设我们如果不需要切换扇区，只对一个扇区进行读写操作。只在第一次读写的时候，将磁盘的内容读取到缓冲区中，接下来的读写操作实际上只是在缓冲区中进行的，而并没有实时的同步到磁盘中。一旦当我们切换扇区号的时候，就将 win[]中的数据同步到磁盘中。显然，这样的效率很高，要比每一次都同步到磁盘中快的多。

缓冲区在常见情况的功效：

通常我们读写一个文件，都不是一下很大范围的操作看，经常都是局部范围的读写。局部范围的读写只在第一次读写时，将磁盘中的数据读取到缓冲区中，接下来的操作全部都在缓冲区中进行。最后同步文件时，才将缓冲区的内容写入到磁盘。

总结：

win[]和 buffer 作为磁盘到用户空间之间的过渡桥梁，使得用户的零碎读写操作，变成了磁盘的整存整取操作，提高了文件系统的效率。

缓冲区算法实现关键：命中缓冲区就用缓冲区的操作，没命中替换缓冲区。

2、无缓冲区模式

FatFs 可裁剪成无缓冲区模式的文件系统，也就是阉割掉 buffer，但 win[]仍然需要。这样对于内存小的 MCU 来说，是一件非常有益的事情，当然也会为之付出代价—--不能零存领取。

没有 buffer，这样用户空间和文件数据空间是直通的，读写一次至少 512 字节，所以用户的操作都必须是 512 字节对齐的，也就是说文件指针要 512 字节对齐，而不像有 buffer 那样的任意数值。

由于 DBR/FAT/目录项这些参数的修改，必须是零存领取，所以 win[]就必须需要了。

3、_FS_READONLY 模式

Readonly 模式，也就是用户对于磁盘只有读取数据的需要，而没有写磁盘的要求。这种模式，阉割了代码量，对于 ROM 比较小的 MCU 来说也是一件非常有益的事情。但是这种模式对于减少 RAM 消耗量却起不到大的作用，要想减少 RAM 只能使用无缓冲 buffer 的模式。

五、FATFS 文件系统函数使用注意事项

1、不使用一个文件的时候，要调用 f_close 或者 f_sync 函数将文件同步到磁盘中。

2、f_read、f_write、f_lseek、f_sync、f_close 在使用前要先打开文件，也即是调用 f_open 函数。

3、f_chmod、f_stat 无需事先打开文件，可以直接使用

3、f_readdir 使用前要先打开目录，也就是调用函数 f_opendir