深圳信盈达科技有限公司

Fatfs-0.11 API 使用手册

陈潮辉

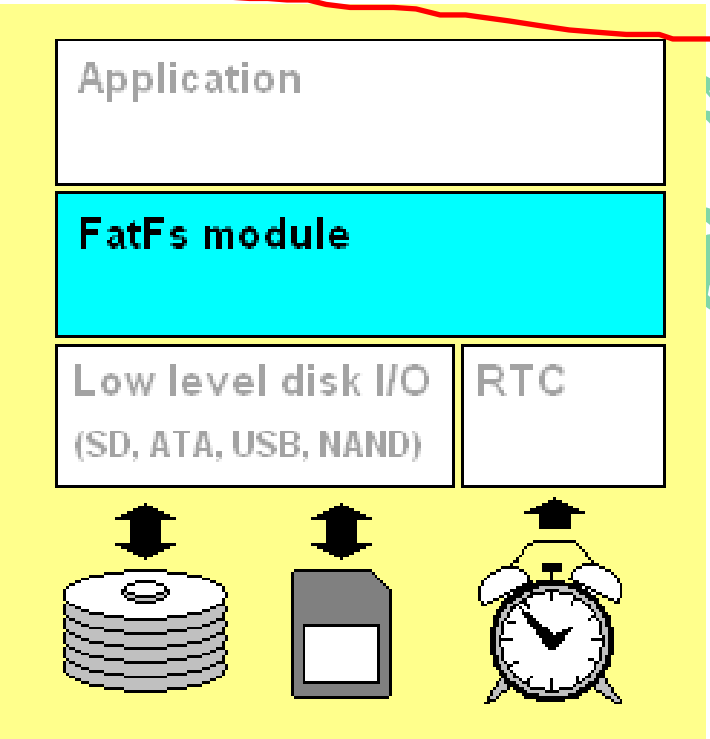
2015-7-10

目录

1. Fatfs 简介

FatFs 是一个为小型嵌入式系统设计的通用 FAT(File Allocation Table)文件系统模块。

FatFs 的编写遵循 ANSI C，并且完全与磁盘 I/O 层分开。因此，它独立(不依赖)于硬件架构。它可以被嵌入到低成本的微控制器中，如 AVR、8051、PIC、ARM、Z80、68K 等等，而不需要做任何修改。



1.1 特点

- Windows 兼容的 FAT 文件系统。
- 不依赖于平台，易于移植。
- 代码和工作区占用空间非常小。
- 多种配置选项：
  - 多卷(物理驱动器和分区)。
  - 多 ANSI/OEM 代码页，包括 DBCS。
  - 在 ANSI/OEM 或 Unicode 中长文件名的支持。
  - RTOS 的支持。
  - 多扇区大小的支持。
  - 只读模式时使用最少的 API，I/O 缓冲区等等。

官方网站：http://elm-chan.org/fsw/ff/00index_e.html

2. 应用程序接口

FatFs 模块为应用程序提供了下列 API 函数，这些函数描述了 FatFs 能对 FAT 卷执行相应的操作。

函数名	描述
f_mount	注册/注销一个工作区
f_open	打开/创建一个文件
f_close	关闭一个文件
f_read	读取文件
f_write	写文件
f_lseek	移动读/写指针，扩展文件大小
f_truncate	截断文件大小
f_sync	清空缓冲数据
f_opendir	打开一个目录
f_closedir	关闭一个目录
f_readdir	读取一个目录项
f_findfirst	在指定路径搜索指定类型的文件
f_findnext	搜索下一个匹配的文件
f_getfree	获取空闲簇
f_stat	获取文件状态
f_mkdir	创建一个目录
f_unlink	删除一个文件或目录
f_chmod	修改属性
f_utime	修改日间戳
f_rename	删除/移动一个文件或目录
f_chdir	修改当前目录
f_chdrive	修改当前驱动器
f_getcwd	恢复当前目录
f_forward	直接输出文件数据流
f_getlable	获取磁盘卷标
f_setlable	设置磁盘卷标
f_mkfs	在驱动器上创建一个文件系统
f_fdisk	划分一个物理驱动器
f_gets	读取一个字符串
f_putc	写一个字符
f_puts	写一个字符串
f_printf	写一个格式化的字符串
f_tell	获取当前读/写指针
f_eof	测试一个文件是否到达文件末尾
f_size	获取一个文件的大小
f_error	测试一个文件是否出错

2.1 f_mount

在 FatFs 模块上注册/注销一个工作区(文件系统对象)。

```
FRESULT f_mount (
    FATFS* fs,          /* Pointer to the file system object (NULL:unmount)*/
    const TCHAR* path,  /* Logical drive number to be mounted/unmounted */
    BYTE opt            /* 0:Do not mount (delayed mount), 1:Mount immediately */
)
```

参数

fs

指向文件系统对象注册和清除。空指针注销注册的文件系统对象。

path

指针使用指定的字符串逻辑驱动器。字符串没有驱动数字意味着默认驱动器。

opt

初始化选项。 0：现在不要安装（稍后安装），1：强制安装。

返回值

FR_OK (0) 操作成功。
FR_INVALID_DRIVE 驱动器号无效。

描述

f_mount 函数在 FatFs 模块上注册/注销一个工作区。 在使用任何其他文件函数之前，必须使用该函数为每个卷注册一个工作区。要注销一个工作区，只要指定 fs 为 NULL 即可，然后该工作区可以被丢弃。

范例 2.1：注册/注销一个工作区

```
int main(void)
{
   FATFS Sdfatfs;              /*定义一个逻辑磁盘工作区*/
   ……..  /*初始化相关外设*/
   f_mount(&Sdfatfs,"0:",1); /*为逻辑驱动器注册工作区*/
   ……..  /*用户代码*/
   f_mount(NULL,"0:",1);/*注销工作区，在废弃前*/
}
```

2.2 f_open

创建/打开一个用于访问文件的文件对象

```
FRESULT f_open (
    FIL* fp,            /* Pointer to the blank file object */
    const TCHAR* path,  /* Pointer to the file name */
    BYTE mode           /* Access mode and file open mode flags */
)
```

参数

fp 将被创建的文件对象结构的指针。

Path 带路径的文件名（如： 0：/test/xyd.txt）。

mode 指定文件的访问类型和打开方法。它是由下列标志的一个组合指定的。

模式	描述
FA_READ	指定读访问对象。可以从文件中读取数据。 与 FA_WRITE 结合可以进行读写访问。
FA_WRITE	指定写访问对象。可以向文件中写入数据。 与 FA_READ 结合可以进行读写访问。
FA_OPEN_EXISTING	打开文件。如果文件不存在，则打开失败。(默认)
FA_OPEN_ALWAYS	如果文件存在，则打开；否则，创建一个新文件。
FA_CREATE_NEW	创建一个新文件。如果文件已存在，则创建失败。
FA_CREATE_ALWAYS	创建一个新文件。如果文件已存在，则它将被截断并覆盖。

注意：当 _FS_READONLY == 1 时，模式标志 FA_WRITE, FA_CREATE_ALWAYS, FA_CREATE_NEW, FA_OPEN_ALWAYS 是无效的。

返回值

FR_OK (0) 操作成功。

FR_NO_FILE 找不到该文件。

FR_NO_PATH 找不到该路径。

FR_INVALID_NAME 文件名无效。

FR_INVALID_DRIVE 驱动器号无效。

FR_EXIST 该文件已存在。

FR_DENIED 由于下列原因，所需的访问被拒绝：

- 以写模式打开一个只读文件。
- 由于存在一个同名的只读文件或目录，而导致文件无法被创建。
- 由于目录表或磁盘已满，而导致文件无法被创建。

FR_NOT_READY 由于驱动器中没有存储介质或任何其他原因，而导致磁盘驱动器无法工作。

FR_WRITE_PROTECTED 在存储介质被写保护的情况下，以写模式打开或创建文件对象。

FR_DISK_ERR 由于底层磁盘 I/O 接口函数中的一个错误，而导致该函数失败。

FR_INT_ERR 由于一个错误的 FAT 结构或一个内部错误，而导致该函数失败。

FR_NOT_ENABLED 逻辑驱动器没有工作区。

FR_NO_FILESYSTEM 磁盘上没有有效的 FAT 卷。

描述

如果函数成功，则创建或者打开一个文件对象。该文件对象被后续的读/写函数用来访问文件。如果想要关闭一个打开的文件对象，则使用 f_close 函数。如果不关闭修改后的文件，那么文件有可能会崩溃。

在使用任何文件函数之前，必须使用 f_mount 函数为驱动器注册一个工作区。只有这样，其他文件函数才能正常工作。

范例 2.2：文件拷贝

```
void main (void)
{
	FATFS fs[2];		/*	逻辑驱动器的工作区(文件系统对象) */
	FIL fsrc, fdst;		/*	文件对象 */
	BYTE buffer[4096]; /*	文件拷贝缓冲区 */
	FRESULT res;		/* FatFs	函数公共结果代码 */
	UINT br, bw;		/*	文件读/写字节计数 */

	/*	为逻辑驱动器注册工作区 */
	f_mount(&fs[0],"0:",1);
	f_mount(&fs[1],"1:",1);
	/*	打开驱动器 1	上的源文件 */
	res = f_open(&fsrc, "1:srcfile.dat", FA_OPEN_EXISTING | FA_READ);
	if (res == FR_OK)
	{
	  /*	在驱动器 0	上创建目标文件 */
		res = f_open(&fdst, "0:dstfile.dat", FA_CREATE_ALWAYS | FA_WRITE);
		if (res == FR_OK)
		{
			/*	拷贝源文件到目标文件 */
			for (;;)
			{
				res = f_read(&fsrc, buffer, sizeof(buffer), &br);
				if (res || br == 0) break;	/*	文件结束了或者错误 */
				res = f_write(&fdst, buffer, br, &bw);
				if (res || bw < br) break;	/* 磁盘满了或者错误 */
			}
		}
	}

	/*	关闭打开的文件 */
	f_close(&fsrc);
	f_close(&fdst);

	/*	注销工作区(在废弃前) */
	f_mount(NULL,"0:",1);
	f_mount(NULL,"1:",1);
	while(1)
	{
	  ;
	}
}
```

2.3 f_close

关闭一个打开的文件

```
FRESULT f_close (
    FIL *fp         /* Pointer to the file object to be closed */
)
```

参数

fp 指向将被关闭的已打开的文件对象结构的指针。

返回值

FR_OK (0) 文件对象已被成功关闭。

FR_DISK_ERR 由于底层磁盘 I/O 函数中的错误，而导致该函数失败。

FR_INT_ERR 由于一个错误的 FAT 结构或一个内部错误，而导致该函数失败。

FR_NOT_READY 由于驱动器中没有存储介质或任何其他原因，而导致磁盘驱动器无法工作。

FR_INVALID_OBJECT 文件对象无效。

描述

f_close 函数关闭一个打开的文件对象。无论向文件写入任何数据，文件的缓存信息都将被写回到磁盘。**在写文件后一定要调用该函数，才能真正把数据写到文件中去。**该函数成功后，文件对象不再有效，并且可以被丢弃。如果文件对象是在只读模式下打开的，不需要使用该函数，也能被丢弃。

范例 2.3：关闭一个打开的文件

```
int main(void)
{
   FATFS Sdfatfs; /*定义一个逻辑磁盘工作区*/
   FIL file;    /*   文件对象 */
   FRESULT res;/* FatFs   函数公共结果代码 */
   UINT brw;   /*   文件读/写字节计数 */
   u8 datbuf[4096];/*   文件拷贝缓冲区 */
   ……..
   f_mount(&Sdfatfs,"0:",1);/*   为逻辑驱动器注册工作区 */
   res = f_open(&file,"0:/xyd.txt",FA_READ);     /*   打开驱动器 0 上的源文件 */
   if(res == FR_OK)/*打开成功*/
   {
        res = f_read(&file,datbuf,4096,&brw);//读取文件数据
        if(res == FR_OK)
        {
           ……..
        }
     }
     f_close(&file);
   ……..
   ……..
}
```

2.4 f_read

从一个文件读取数据

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

参数

fp 指向将被读取的已打开的文件对象结构的指针。

Buffer 指向存储读取数据的缓冲区的指针。

btr 要读取的字节数，UINT 范围内。

br 指向返回已读取字节数的 UINT 变量的指针。在调用该函数后，无论结果如何，数值都是有效的。

返回值

FR_OK (0) 读取数据成功。

FR_DISK_ERR 由于底层磁盘 I/O 函数中的错误，而导致该函数失败。

FR_INT_ERR 由于一个错误的 FAT 结构或一个内部错误，而导致该函数失败。

FR_NOT_READY 由于驱动器中没有存储介质或任何其他原因，而导致磁盘驱动器无法工作。

FR_INVALID_OBJECT 文件对象无效。

描述

文件对象中的读/写指针以已读取字节数增加。该函数成功后，应该检查*br 来检测文件是否结束。在读操作过程中，一旦*br < btr，则读/写指针到达了文件结束位置。

范例 2.4：读文件

```
int main(void)
{
  FATFS Sdfatfs; /*定义一个逻辑磁盘工作区*/
  FIL file;   /*   文件对象 */
  FRESULT res;/* FatFs 函数公共结果代码 */
  UINT brw;   /*   文件读/写字节计数 */
  u8 datbuf[4096];/*   文件拷贝缓冲区 */
  ……..
  f_mount(&Sdfatfs,"0:",1);/*   为逻辑驱动器注册工作区 */
  res = f_open(&file,"0:/xyd.txt",FA_READ);   /*   打开驱动器 0 上的源文件 */
  if(res == FR_OK)/*打开成功*/
  {
      res = f_read(&file,datbuf,4096,&brw);//读取文件数据
      if(res == FR_OK)
      {
        ……..
      }
      f_close(&file);
  }
}
```

2.5 f_write

写入数据到一个文件

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

参数

fp 指向将被写入的已打开的文件对象结构的指针。

buff 指向存储写入数据的缓冲区的指针。

btw 要写入的字节数，UINT 范围内。

bw 指向返回已写入字节数的 UINT 变量的指针。在调用该函数后，无论结果如何，数值都是有效的。

返回值

FR_OK (0) 写入成功。

FR_DENIED 由于文件是以非写模式打开的，而导致该函数被拒绝。

FR_DISK_ERR 由于底层磁盘 I/O 函数中的错误，而导致该函数失败。

FR_INT_ERR 由于一个错误的 FAT 结构或一个内部错误，而导致该函数失败。

FR_NOT_READY 由于驱动器中没有存储介质或任何其他原因，而导致磁盘驱动器无法工作。

FR_INVALID_OBJECT 文件对象无效。

描述

文件对象中的读/写指针以已写入字节数增加。该函数成功后，应该检查*bw 来检测磁盘是否已满。在写操作过程中，一旦 *bw < btw，则意味着该卷已满。

范例 2.5：写文件

```
u8 *str = "信盈达 Fatfs 测试！^v^ $"; /*定义一个字符串*/
int main(void)
{
   FATFS Sdfatfs; /*定义一个逻辑磁盘工作区*/
   FIL file;   /*  文件对象 */
   FRESULT res;/* FatFs  函数公共结果代码 */
   UINT brw;   /*  文件读/写字节计数 */
   f_mount(&Sdfatfs,"0:",1);/*  为逻辑驱动器注册工作区 */
   res = f_open(&file,"0:/xyd.txt",FA_WRITE);  /*打开驱动器 0 上的源文件 */
   if(res == FR_OK)/*打开成功*/
   {
        res = f_write(&file,str,strlen(str)+1,&brw);//往 txt 文件写一个字符串
        if(res == FR_OK)
        {
          ……
        }
        f_close(&file);
     }
}
```

2.6 f_lseek

移动一个打开的文件对象的文件读/写指针。也可以被用来扩展文件大小(簇预分配)。

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

参数

fp 打开的文件对象的指针 Offset 相对于文件起始处的字节数

返回值

FR_OK (0) 移动成功。

FR_DISK_ERR 由于底层磁盘 I/O 函数中的错误，而导致该函数失败。

FR_INT_ERR 由于一个错误的 FAT 结构或一个内部错误，而导致该函数失败。

FR_NOT_READY 由于驱动器中没有存储介质或任何其他原因，而导致磁盘驱动器无法工作。

FR_INVALID_OBJECT 文件对象无效。

描述

f_lseek 函数当 FS_MINIMIZE <= 2 时可用。

offset 只能被指定为相对于文件起始处的字节数。当在写模式下指定了一个超过文件大小的 offset 时，文件的大小将被扩展，并且该扩展的区域中的数据是未定义的。这适用于为快速写操作迅速地创建一个大的文件。f_lseek 函数成功后，为了确保读/写指针已被正确地移动，必须检查文件对象中的成员 fptr。如果 fptr 不是所期望的值，则发生了下列情况之一。

1) 文件结束。指定的 offset 被钳在文件大小，因为文件已被以只读模式打开。
2) 磁盘满。卷上没有足够的空闲空间去扩展文件大小。

范例 2.6：

```
/*  移动文件读/写指针到相对于文件起始处偏移为 5000 字节处 */
res = f_lseek(file, 5000);
/*  移动文件读/写指针到文件结束处，以便添加数据 */
res = f_lseek(file, file->fsize);
/*  向前 3000 字节 */
res = f_lseek(file, file->fptr + 3000);
/*  向后(倒带)2000 字节(注意溢出) */
res = f_lseek(file, file->fptr - 2000);
/*  簇预分配(为了防止在流写时缓冲区上溢 */
res = f_open(file, recfile, FA_CREATE_NEW | FA_WRITE); /* 创建一个文件 */
res = f_lseek(file, PRE_SIZE);      /*  预分配簇 */
if (res || file->fptr != PRE_SIZE) ... /*  检查文件大小是否已被正确扩展 */
res = f_lseek(file, DATA_START);    /*  没有簇分配延迟地记录数据流 */
...
res = f_truncate(file);     /*  截断未使用的区域 */
res = f_lseek(file, 0);     /*  移动到文件起始处 */
...
res = f_close(file);
```

2.7 f_truncate

截断文件大小

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

参数

fp 待截断的打开的文件对象的指针。

返回值

FR_OK (0) 操作成功。

FR_DENIED 由于文件是以非写模式打开的，而导致该函数被拒绝。

FR_DISK_ERR 由于底层磁盘 I/O 函数中的错误，而导致该函数失败。

FR_INT_ERR 由于一个错误的 FAT 结构或一个内部错误，而导致该函数失败。

FR_NOT_READY 由于驱动器中没有存储介质或任何其他原因，而导致磁盘驱动器无法工作。

FR_INVALID_OBJECT 文件对象无效。

描述

f_truncate 函数当_FS_READONLY == 0 并且 _FS_MINIMIZE == 0 时可用。

f_truncate 函数截断文件到当前的文件读/写指针。当文件读/写指针已经指向文件结束时，该函数不起作用。

范例 2.7：

```
res = f_open(file, recfile, FA_CREATE_NEW | FA_WRITE); /* 创建一个文件 */
res = f_lseek(file, PRE_SIZE);      /*  预分配簇 */
if (res || file->fptr != PRE_SIZE) ... /*  检查文件大小是否已被正确扩展 */
res = f_lseek(file, DATA_START);      /*  没有簇分配延迟地记录数据流 */
...
res = f_truncate(file);  /*  截断未使用的区域 */
res = f_lseek(file, 0);     /*  移动到文件起始处 */
...
res = f_close(file);
```

2.8 f_sync

冲洗一个写文件的缓冲信息。

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

参数

fp 待冲洗的打开的文件对象的指针。

返回值

FR_OK (0) 操作成功。
FR_DISK_ERR 由于底层磁盘 I/O 函数中的错误，而导致该函数失败。
FR_INT_ERR 由于一个错误的 FAT 结构或一个内部错误，而导致该函数失败。
FR_NOT_READY 由于驱动器中没有存储介质或任何其他原因，而导致磁盘驱动器无法工作。
FR_INVALID_OBJECT 文件对象无效。

描述

f_sync 函数当_FS_READONLY == 0 时可用。

f_sync 函数和 f_close 函数执行同样的过程，但是文件仍处于打开状态，并且可以继续对文件执行读/写/移动指针操作。这适用于以写模式长时间打开文件，比如数据记录器。定期的或 f_write 后立即执行 f_sync 可以将由于突然断电或移去磁盘而导致数据丢失的风险最小化。在 f_close 前立即执行 f_sync 没有作用，因 为在 f_close 中执行了 f_sync。换句话说，这两个函数的差异就是文件对象是不是无效的。

2.9 f_opendir

打开一个目录（文件夹）

```
FRESULT f_opendir (
    DIR* dp,              /* Pointer to directory object to create */
    const TCHAR* path     /* Pointer to the directory path */
)
```

参数

dp 待创建的空白目录对象的指针。

path '\0'结尾的字符串指针，该字符串指定了将被打开的目录名。

返回值

FR_OK (0) 操作成功，目录对象被创建。该目录对象被后续调用，用来读取目录项。

FR_NO_PATH 找不到路径。

FR_INVALID_NAME 路径名无效。

FR_INVALID_DRIVE 驱动器号无效。

FR_NOT_READY 由于驱动器中没有存储介质或任何其他原因，而导致磁盘驱动器无法工作。

FR_DISK_ERR 由于底层磁盘 I/O 函数中的错误，而导致该函数失败。

FR_INT_ERR 由于一个错误的 FAT 结构或一个内部错误，而导致该函数失败。

FR_NOT_ENABLED 逻辑驱动器没有工作区。

FR_NO_FILESYSTEM 磁盘上没有有效的 FAT 卷。

描述

f_opendir 函数当_FS_MINIMIZE <= 1 时可用。

f_opendir 函数打开一个已存在的目录，并为后续的调用创建一个目录对象。该目录对象结构可以在任何时候不经任何步骤而被丢弃。

2.10 f_closedir

关闭一个已打开的目录（文件夹）

```
FRESULT f_closedir (
  DIR* dp/* [IN] Pointer to the directory object */
);
```

参数

dp 指向将被关闭的已打开的目录对象结构的指针。

返回值

FR_OK 关闭成功。

FR_INT_ERR 由于一个错误的 FAT 结构或一个内部错误，而导致该函数失败。

FR_TIMEOUT 该功能被取消或操作超时

FR_INVALID_OBJECT 文件/目录对象是无效的或空指针的位置。

- 有一些原因如下：
  - 它已被关闭，则不会打开，或者它可以被折叠。
  - 已经无效由电压快速安装过程。 卷的所有打开的对象是无效的为好。
  - 相应的物理驱动器没有准备好因媒体去除。

描述

该 f_closedir 函数关闭打开的目录对象。 函数成功后，该目录对象不再有效，并且它可以被丢弃。当然目录对象也可以没有此过程丢弃，如果未启用_FS_LOCK 选项的话。 然而，为了与未来的代码相兼容，建议使用。

2.11 f_readdir

读取目录项

```
FRESULT f_readdir (
    DIR* dp,            /* Pointer to the open directory object */
    FILINFO* fno        /* Pointer to file information to return */
)
```

参数

dp 打开的目录对象的指针。

fno 存储已读取项的文件信息结构指针。

返回值

FR_OK (0) 读取成功。

FR_DISK_ERR 由于底层磁盘 I/O 函数中的错误，而导致该函数失败。

FR_INT_ERR 由于一个错误的 FAT 结构或一个内部错误，而导致该函数失败。

FR_NOT_READY 由于驱动器中没有存储介质或任何其他原因，而导致磁盘驱动器无法工作。

FR_INVALID_OBJECT 文件对象无效。

描述

f_readdir 函数当_FS_MINIMIZE <= 1 时可用。

f_readdir 函数顺序读取目录项。目录中的所有项可以通过重复调用 f_readdir 函数被读取。当所有目录项已被读取并且没有项要读取时，该函数没有任何错误地返回一个空字符串到 f_name[]成员中。当 FileInfo 给定一个空指针时，目录对象的读索引将被回绕。

当 LFN 功能被使能时，在使用 f_readdir 函数之前，文件信息结构中的 lfname 和 lfsize 必须被初始化为 有效数值。lfname 是一个返回长文件名的字符串缓冲区指针。lfsize 是以字符为单位的字符串缓冲区的大小。 如果读缓冲区或 LFN 工作缓冲区的大小(对于 LFN)不足，或者对象没有 LFN，则一个空字符串将被返回到 LFN 读缓冲区。如果 LFN 包含任何不能被转换为 OEM 代码的字符，则一个空字符串将被返回，但是这不 是 Unicode API 配置的情况。当 lfname 是一个空字符串时，没有 LFN 的任何数据被返回。当对象没有 LFN 时，任何小型大写字母可以被包含在 SFN 中。

当相对路径功能被使能(_FS_RPATH == 1)时，"."和".."目录项不会被过滤掉，并且它将出现在读目录项中。

范例 2.10：

```
FRESULT scan_files (
    char* path      /* Start node to be scanned (also used as work area) */
)
{
    FRESULT res;
    FILINFO fno;
    DIR dir;
    int i;
    char *fn; /* This function is assuming non-Unicode cfg. */
#if _USE_LFN
    static char lfn[_MAX_LFN + 1];
    fno.lfname = lfn;
    fno.lfsize = sizeof lfn;
#endif
    res = f_opendir(&dir, path);   /* Open the directory */
    if (res == FR_OK) {
        i = strlen(path);
        for (;;) {
            res = f_readdir(&dir, &fno);   /* Read a directory item */
            if (res != FR_OK || fno.fname[0] == 0) break;      /* Break on error or end of dir */
            if (fno.fname[0] == '.') continue;   /* Ignore dot entry */
#if _USE_LFN
            fn = *fno.lfname ? fno.lfname : fno.fname;

#else
            fn = fno.fname;
#endif

            if (fno.fattrib & AM_DIR) {   /* It is a directory */
                sprintf(&path[i], "/%s", fn);
                res = scan_files(path);
                if (res != FR_OK) break;
                path[i] = 0;
            } else {   /* It is a file. */
                printf("%s/%s\n", path, fn);
            }
        }
    }
    return res;
}
```

2.12 f_findfirst

在指定路径搜索指定类型的文件

```
FRESULT f_findfirst (
    DIR* dp,                /* Pointer to the blank directory object */
    FILINFO* fno,           /* Pointer to the file information structure */
    const TCHAR* path,      /* Pointer to the directory to open */
    const TCHAR* pattern    /* Pointer to the matching pattern */
)
```

参数

dp 打开的目录对象的指针。
fno 存储已读取项的文件信息结构指针。
Path 路径
Pattern 文件类型

返回值

FR_OK	搜索成功。
FR_DISK_ERR	由于底层磁盘 I/O 函数中的错误，而导致该函数失败。
FR_INT_ERR	由于一个错误的 FAT 结构或一个内部错误，而导致该函数失败。
FR_NOT_READY	由于驱动器中没有存储介质或任何其他原因，而导致磁盘驱动器无法工作。
FR_NO_PATH	无法找到路径。
FR_INVALID_NAME	给定的字符串是无效的路径名。
FR_INVALID_OBJECT	文件/目录对象是无效的或空指针的位置。
FR_INVALID_DRIVE	无效的驱动器号。
FR_NOT_ENABLED	工作区的逻辑驱动器未登记。
FR_NO_FILESYSTEM	驱动器上没有有效 FAT 卷。
FR_TIMEOUT	操作超时。
FR_NOT_ENOUGH_CORE	没有足够的内存来运行。
FR_TOO_MANY_OPEN_FILES	打开对象的数量已经达到了最大值，并没有更多的对象可以打开。

描述

由 path 指定的目录中可以打开后，它开始在目录中搜索指定文件类型(如“*.mp3”、“*.txt”等)的文件。 如果找到，关于文件的信息被存储到该文件的信息结构体对应成员里面。

这是一个 f_opendir 和 f_readdir 函数的封装。 在_USE_FIND == 1 和_FS_MINIMIZE <= 1 时可用。匹配模式可以包含通配符（？和*）。 一个？匹配一个任意字符和 * 匹配任何字符串。 “*.*”永远不匹配任何扩展名。

范例 2.12：

```
/* Search a directory for objects and display it */

void find_image (void)
{
    FRESULT fr;         /* Return value */
    DIR dj;             /* Directory search object */
    FILINFO fno;        /* File information */
#if _USE_LFN
    char lfn[_MAX_LFN + 1];
    fno.lfname = lfn;
    fno.lfsize = sizeof lfn;
#endif

    fr = f_findfirst(&dj, &fno, "", "dsc*.jpg");    /* Start to search for JPEG files with the name started by
"dsc" */

    while (fr == FR_OK && fno.fname[0]) {           /* Repeat while an item is found */
#if _USE_LFN
        printf("%-12s  %s\n", fno.fname, fno.lfname);/* Display the item name */
#else
        printf("%s\n", fno.fname);
#endif
        fr = f_findnext(&dj, &fno);                 /* Search for next item */
    }
    f_closedir(&dj);
}
```

2.13 f_findnext

搜索下一个匹配的对象

```
FRESULT f_findnext (
    DIR* dp,        /* Pointer to the open directory object */
    FILINFO* fno/* Pointer to the file information structure */
)
```

参数

dp 打开的目录对象的指针。

fno 存储已读取项的文件信息结构指针。

返回值

FR_OK	搜索成功。
FR_DISK_ERR	由于底层磁盘 I/O 函数中的错误，而导致该函数失败。
FR_INT_ERR	由于一个错误的 FAT 结构或一个内部错误，而导致该函数失败。
FR_NOT_READY	由于驱动器中没有存储介质或任何其他原因，而导致磁盘驱动器无法工作。
FR_INVALID_OBJECT	文件/目录对象是无效的或空指针的位置。
FR_TIMEOUT	操作超时。
FR_NOT_ENOUGH_CORE	没有足够的内存来运行。

描述

它从先前的调用 f_findfirst 或 f_findnext 继续搜索文件。如果匹配,关于对象的信息存储在文件信息结构。如果没有读取条目,将返回一个空字符串到 fno->fname[ ]。

2.14 f_getfree

获取空闲簇的数目

```
FRESULT f_getfree (
    const TCHAR* path,      /* Path name of the logical drive number */
    DWORD* nclst,           /* Pointer to a variable to return number of free clusters */
    FATFS** fatfs           /* Pointer to return pointer to corresponding file system object */
)
```

参数

path '\0'结尾的字符串指针，该字符串指定了逻辑驱动器的目录。
nclst 存储空闲簇数目的 DWORD 变量的指针。
fstfs 相应文件系统对象指针的指针。

返回值

FR_OK (0) 获取成功。*nclst 表示空闲簇的数目，并且*fatfs 指向文件系统对象。
FR_INVALID_DRIVE 驱动器号无效。
FR_NOT_READY 由于驱动器中没有存储介质或任何其他原因，而导致磁盘驱动器无法工作。
FR_DISK_ERR 由于底层磁盘 I/O 函数中的错误，而导致该函数失败。
FR_INT_ERR 由于一个错误的 FAT 结构或一个内部错误，而导致该函数失败。
FR_NOT_ENABLED 逻辑驱动器没有工作区。
FR_NO_FILESYSTEM 磁盘上没有有效的 FAT 卷。

描述

f_getfree 函数当_FS_READONLY == 0 并且_FS_MINIMIZE == 0 时有效。
f_getfree 函数获取驱动器上空闲簇的数目。文件系统对象中的成员 csize 是每簇中的扇区数，因此，以扇区为单位的空闲空间可以被计算出来。当 FAT32 卷上的 FSInfo 结构不同步时，该函数返回一个错误的空 闲簇计数。

范例 2.14：

```
ATFS *fs;
DWORD fre_clust, fre_sect, tot_sect;
/* Get volume information and free clusters of drive 1 */
res = f_getfree("1:", &fre_clust, &fs);
if (res) die(res);

/* Get total sectors and free sectors */
tot_sect = (fs->n_fatent - 2) * fs->csize;
fre_sect = fre_clust * fs->csize;

/* Print the free space (assuming 512 bytes/sector) */
printf("%10lu KiB total drive space.\n%10lu KiB available.\n",
                tot_sect / 2, fre_sect / 2);
```

2.15 f_stat

获取文件状态

```
FRESULT f_stat (
    const TCHAR* path,    /* Pointer to the file path */
    FILINFO* fno          /* Pointer to file information to return */
)
```

参数

path '\0'结尾的字符串指针，该字符串指定了待获取其信息的文件或目录。
fno 存储信息的空白 FILINFO 结构的指针。

返回值

FR_OK (0) 获取成功。
FR_NO_FILE 找不到文件或目录。
FR_NO_PATH 找不到路径。
FR_INVALID_NAME 路径名无效。
FR_INVALID_DRIVE 驱动器号无效。
FR_NOT_READY 由于驱动器中没有存储介质或任何其他原因，而导致磁盘驱动器无法工作。
FR_DISK_ERR 由于底层磁盘 I/O 函数中的错误，而导致该函数失败。
FR_INT_ERR 由于一个错误的 FAT 结构或一个内部错误，而导致该函数失败。
FR_NOT_ENABLED 逻辑驱动器没有工作区。
FR_NO_FILESYSTEM 磁盘上没有有效的 FAT 卷。

描述

f_stat 函数当_FS_MINIMIZE == 0 时可用。
f_stat 函数获取一个文件或目录的信息。信息的详情，请参考 FILINFO 结构和 f_readdir 函数。

2.16 f_mkdir

创建一个目录

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

参数

path '\0'结尾的字符串指针，该字符串指定了待创建的目录。

返回值

FR_OK 创建成功。
FR_NO_PATH 找不到路径。
FR_INVALID_NAME 路径名无效。
FR_INVALID_DRIVE 驱动器号无效。
FR_DENIED 由于目录表或磁盘满，而导致目录不能被创建。
FR_EXIST 已经存在同名的文件或目录。
FR_NOT_READY 由于驱动器中没有存储介质或任何其他原因，而导致磁盘驱动器无法工作。
FR_WRITE_PROTECTED 存储介质被写保护。
FR_DISK_ERR 由于底层磁盘 I/O 函数中的错误，而导致该函数失败。
FR_INT_ERR 由于一个错误的 FAT 结构或一个内部错误，而导致该函数失败。
FR_NOT_ENABLED 逻辑驱动器没有工作区。
FR_NO_FILESYSTEM 磁盘上没有有效的 FAT 卷。

描述

f_mkdir 函数当_FS_READONLY == 0 并且_FS_MINIMIZE == 0 时可用。
f_mkdir 函数创建一个新目录。

范例 2.16：

```
Fres = f_mkdir("sub1");
if (res) die(res);
res = f_mkdir("sub1/sub2");
if (res) die(res);
res = f_mkdir("sub1/sub2/sub3");
if (res) die(res);
```

2.17 f_unlink

删除一个文件或文件夹

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

参数

path '\0'结尾的字符串指针，该字符串指定了一个待移除的对象。

返回值

FR_OK (0) 删除成功。
FR_NO_FILE 找不到文件或目录。
FR_NO_PATH 找不到路径。
FR_INVALID_NAME 路径名无效。
FR_INVALID_DRIVE 驱动器号无效。
FR_DENIED 由于下列原因之一，而导致该函数被拒绝：
1）对象具有只读属性；
2）目录不是空的。
FR_NOT_READY 由于驱动器中没有存储介质或任何其他原因，而导致磁盘驱动器无法工作。
FR_WRITE_PROTECTED 存储介质被写保护。
FR_DISK_ERR 由于底层磁盘 I/O 函数中的错误，而导致该函数失败。
FR_INT_ERR 由于一个错误的 FAT 结构或一个内部错误，而导致该函数失败。
FR_NOT_ENABLED 逻辑驱动器没有工作区。
FR_NO_FILESYSTEM 磁盘上没有有效的 FAT 卷。

描述

f_unlink 函数当_FS_READONLY == 0 并且_FS_MINIMIZE == 0 时可用。
f_unlink 函数移除一个对象。不要移除打开的对象或当前目录。

2.18 f_chmod

修改一个文件或者目录的属性。

```
FRESULT f_chmod (
    const TCHAR* path,    /* Pointer to the file path */
    BYTE attr,            /* Attribute bits */
    BYTE mask             /* Attribute mask to change */
)
```

参数

path '\0'结尾的字符串指针，该字符串指定了一个待被修改属性的文件或目录。
attr 待被设置的属性标志，可以是下列标志的一个或任意组合。指定的标志被设置，其他的被清除。
mask 属性掩码，指定修改哪个属性，指定的属性被设置或清除。如下表：

属性	描述
AM_RDO	只读
AM_ARC	存档
AM_SYS	系统
AM_HID	隐藏

返回值

FR_OK (0) 修改成功。
FR_NO_FILE 找不到文件或目录。
FR_NO_PATH 找不到路径。
FR_INVALID_NAME 路径名无效。
FR_INVALID_DRIVE 驱动器号无效。
FR_NOT_READY 由于驱动器中没有存储介质或任何其他原因，而导致磁盘驱动器无法工作。
FR_WRITE_PROTECTED 存储介质被写保护。
FR_DISK_ERR 由于底层磁盘 I/O 函数中的错误，而导致该函数失败。
FR_INT_ERR 由于一个错误的 FAT 结构或一个内部错误，而导致该函数失败。
FR_NOT_ENABLED 逻辑驱动器没有工作区。
FR_NO_FILESYSTEM 磁盘上没有有效的 FAT 卷。

描述

f_chmod 函数当_FS_READONLY == 0 并且_FS_MINIMIZE == 0 时可用。
f_chmod 函数修改一个文件或目录的属性。

范例 2.18：

```
//  设置只读标志，清除存档标志，其他不变
f_chmod("file.txt", AR_RDO, AR_RDO |AR_ARC);
```

2.19 f_utime

修改一个文件或目录的时间戳。

```
FRESULT f_utime (
    const TCHAR* path,    /* Pointer to the file/directory name */
    const FILINFO* fno    /* Pointer to the time stamp to be set */
)
```

参数

path '\0'结尾的字符串的指针，该字符串指定了一个待修改时间戳的文件或目录。

fno 文件信息结构指针，其中成员 ftime 和 fdata 存储了一个待被设置的的时间戳。不关心任何其他成员。

返回值

FR_OK (0) 修改成功。
FR_NO_FILE 找不到文件或目录。
FR_NO_PATH 找不到路径。
FR_INVALID_NAME 路径名无效。
FR_INVALID_DRIVE 驱动器号无效。
FR_NOT_READY 由于驱动器中没有存储介质或任何其他原因，而导致磁盘驱动器无法工作。
FR_WRITE_PROTECTED 存储介质被写保护。
FR_DISK_ERR 由于底层磁盘 I/O 函数中的错误，而导致该函数失败。
FR_INT_ERR 由于一个错误的 FAT 结构或一个内部错误，而导致该函数失败。
FR_NOT_ENABLED 逻辑驱动器没有工作区。
FR_NO_FILESYSTEM 磁盘上没有有效的 FAT 卷。

描述

f_utime 函数当_FS_READONLY == 0 并且_FS_MINIMIZE == 0 时可用。
f_utime 函数修改一个文件或目录的时间戳。

2.20 f_rename

重命名一个对象

```
FRESULT f_rename (
    const TCHAR* path_old,    /* Pointer to the object to be renamed */
    const TCHAR* path_new     /* Pointer to the new name */
)
```

参数

path_old '\0'结尾的字符串的指针，该字符串指定了待被重命名的原对象名。

path_new '\0'结尾的字符串的指针，该字符串指定了重命名后的新对象名，不能包含驱动器号。

返回值

FR_OK (0) 重命名成功。
FR_NO_FILE 找不到原名。
FR_NO_PATH 找不到路径。
FR_INVALID_NAME 文件名无效。
FR_INVALID_DRIVE 驱动器号无效。
FR_NOT_READY 由于驱动器中没有存储介质或任何其他原因，而导致磁盘驱动器无法工作。
FR_EXIST 新名和一个已存在的对象名冲突。
FR_DENIED 由于任何原因，而导致新名不能被创建。
FR_WRITE_PROTECTED 存储介质被写保护。
FR_DISK_ERR 由于底层磁盘 I/O 函数中的错误，而导致该函数失败。
FR_INT_ERR 由于一个错误的 FAT 结构或一个内部错误，而导致该函数失败。
FR_NOT_ENABLED 逻辑驱动器没有工作区。
FR_NO_FILESYSTEM 磁盘上没有有效的 FAT 卷。

描述

f_rename 函数当_FS_READONLY == 0 并且_FS_MINIMIZE == 0 时可用。
f_rename 函数重命名一个对象，并且也可以将对象移动到其他目录。逻辑驱动器号由原名决定，新名不能包含一个逻辑驱动器号。不要重命名打开的对象。

范例 2.20：

```
    /*  重命名一个对象 */
    f_rename("oldname.txt", "newname.txt");

    /*  重命名并且移动一个对象到另一个目录 */
    f_rename("oldname.txt", "dir1/newname.txt");
```

2.21 f_chdir

改变一个驱动器的当前目录

```
FRESULT f_chdir (
    const TCHAR* path      /* Pointer to the directory path */
)
```

参数

path '\0'结尾的字符串的指针，该字符串指定了将要进去的目录。

返回值

FR_OK (0)函数成功。
FR_NO_PATH 找不到路径。
FR_INVALID_NAME 路径名无效。
FR_INVALID_DRIVE 驱动器号无效。
FR_NOT_READY 由于驱动器中没有存储介质或任何其他原因，而导致磁盘驱动器无法工作。
FR_DISK_ERR 由于底层磁盘 I/O 函数中的错误，而导致该函数失败。
FR_INT_ERR 由于一个错误的 FAT 结构或一个内部错误，而导致该函数失败。
FR_NOT_ENABLED 逻辑驱动器没有工作区。
FR_NO_FILESYSTEM 磁盘上没有有效的 FAT 卷。

描述

f_chdir 函数当_FS_RPATH == 1 时可用。
f_chdir 函数改变一个逻辑驱动器的当前目录。当一个逻辑驱动器被自动挂载时，它的当前目录被初始化为根目录。注意：当前目录被保存在每个文件系统对象中，因此它也影响使用同一逻辑驱动器的其它任务。

范例 2.21：

```
//改变当前驱动器的当前目录(根目录下的 dir1)
f_chdir("/dir1");
//改变驱动器 2 的当前目录(父目录)
f_chdir("2:..");
```

2.22 f_chdrive

改变当前驱动器

```
FRESULT f_chdrive (
    const TCHAR* path          /* Drive number */
)
```

参数

path 指定将被设置为当前驱动器的逻辑驱动器号。

返回值

FR_OK (0) 改变成功。

FR_INVALID_DRIVE 驱动器号无效。

描述

f_chdrive 函数当_FS_RPATH == 1 时可用。

f_chdrive 函数改变当前驱动器。当前驱动器号初始值为 0，注意：当前驱动器被保存为一个静态变量，因此它也影响使用文件函数的其它任务。

2.23 f_getcwd

恢复当前目录

```
FRESULT f_getcwd (
    TCHAR* buff,    /* Pointer to the directory path */
    UINT len        /* Size of path */
)
```

参数

buf 指向接收当前目录字符串的缓冲区

len 缓冲区的大小，单位为 TCHAR

返回值

FR_OK (0) 恢复成功。

FR_NOT_READY 由于驱动器中没有存储介质或任何其他原因，而导致磁盘驱动器无法工作。

FR_DISK_ERR 由于底层磁盘 I/O 函数中的错误，而导致该函数失败。

FR_INT_ERR 由于一个错误的 FAT 结构或一个内部错误，而导致该函数失败。

FR_NOT_ENABLED 逻辑驱动器没有工作区。

FR_NO_FILESYSTEM 磁盘上没有有效的 FAT 卷。

FR_TIMEOUT 函数由于线程安全控制超时而退出。（相关选项：_TIMEOUT）

FR_NOT_ENOUGH_CORE 没有足够的内存进行操作。原因有：

- 不能为 LFN 工作缓冲区分配内存（相关选项：_USE_LFN）。
- 得到的表大小不能满足要求的大小。

描述

f_getcwd 函数用完整的路径字符串（包括驱动器号）来恢复当前驱动器上的当前目录。

提示：在_FS_RPATH == 2 时可用。

2.24 f_forward

读取文件并将其转发到数据流设备。

```
FRESULT f_forward (
    FIL* fp,                        /* Pointer to the file object */
    UINT (*func)(const BYTE*,UINT),    /* Pointer to the streaming function */
    UINT btf,                       /* Number of bytes to forward */
    UINT* bf                        /* Pointer to number of bytes forwarded */
)
```

参数

fp 打开的文件对象的指针。

func 用户定义的数据流函数的指针。详情参考示例代码。

btf 要转发的字节数，UINT 范围内。

bf 返回已转发的字节数的 UINT 变量的指针。

返回值

FR_OK (0) 操作成功。

FR_DENIED 由于文件已经以非读模式打开，而导致函数失败。

FR_DISK_ERR 由于底层磁盘 I/O 函数中的错误，而导致该函数失败。

FR_INT_ERR 由于一个错误的 FAT 结构或一个内部错误，而导致该函数失败。

FR_NOT_READY 由于驱动器中没有存储介质或任何其他原因，而导致磁盘驱动器无法工作。

FR_INVALID_OBJECT 文件对象无效。

描述

f_forward 函数当_USE_FORWARD == 1 并且_FS_TINY == 1 时可用。

f_forward 函数从文件中读取数据并将数据转发到输出流，而不使用数据缓冲区。这适用于小存储系统因为它在应用模块中不需要任何数据缓冲区。文件对象的文件指针以转发的字节数增加。

如果*ByteFwd < ByteToFwd 并且没有错误，则意味着由于文件结束或在数据传输过程中流忙，请求的字节不能被传输。

范例 2.24：

```
/*-----------------------------------------------------------------*/
/*  示例代码：数据传输函数，将被 f_forward 函数调用 */
/*-----------------------------------------------------------------*/
UINT out_stream ( /*  返回已发送字节数或流状态 */
    const BYTE *p,    /*  将被发送的数据块的指针 */
    UINT btf /* >0:  传输调用(将被发送的字节数)。0:  检测调用 */
)
{
    UINT cnt = 0;
    if (btf == 0) {  /* 检测调用 */
        /*  当检测调用时，一旦它返回就绪，那么在后续的传输调用时，它必须接收至少一个字节，或者
        f_forward 将以 FR_INT_ERROR 而失败。 */
        if (FIFO_READY)  cnt = 1; /*  返回流状态(0:  忙，1:  就绪) */
    }
    else
    {   /*  传输调用 */
        do
        {   /*  当有数据要发送并且流就绪时重复 */
            cnt++;
            FIFO_PORT = *p++;
        } while (cnt < btf && FIFO_READY);
    }
    return cnt;
}
/*-----------------------------------------------------------------*/
/*  示例代码：使用 f_forward 函数 */
/*-----------------------------------------------------------------*/
FRESULT play_file (
    char *fn  /*  待播放的音频文件名的指针 */
)
{
    FRESULT rc;
    FIL fil;
    UINT dmy;
    rc = f_open(&fil, fn, FA_READ); /*  以只读模式打开音频文件 */
    while (rc == FR_OK && fil.fptr < fil.fsize) /*  重复，直到文件指针到达文件结束位置 */
    {
        /*  任何其他处理... */
        rc = f_forward(&fil, out_stream, 1000, &dmy); /*  定期或请求式填充输出流 */
    }
    /*  该只读的音频文件对象不需要关闭就可以被丢弃 */
    return rc;
}
```

2.25 f_getlabel

获取磁盘卷标和驱动器的卷序列号。

```
FRESULT f_getlabel (
    const TCHAR* path,    /* Path name of the logical drive number */
    TCHAR* label,         /* Pointer to a buffer to return the volume label */
    DWORD* vsn            /* Pointer to a variable to return the volume serial number */
)
```

参数

path 指向指定的字符串逻辑驱动器。空串指定默认驱动器。

lable 指向缓冲区的指针存储卷标。缓冲区的大小必须至少 12byte。如果缓冲区太小,将返回一个空串。如果不需要这些信息可以设置为空指针。

vsn 双字指针变量来存储卷序列号。如果不需要这些信息可设置为空指针。

返回值

FR_OK	获取成功
FR_DISK_ERR	由于底层磁盘 I/O 函数中的错误，而导致该函数失败。
FR_INT_ERR	由于一个错误的 FAT 结构或一个内部错误，而导致该函数失败。
FR_NOT_READY	由于驱动器中没有存储介质或任何其他原因，而导致磁盘驱动器无法工作。
FR_INVALID_DRIVE	无效的驱动器号。
FR_NOT_ENABLED	工作区的逻辑驱动器未登记。
FR_NO_FILESYSTEM	驱动器上没有有效 FAT 卷。
FR_TIMEOUT	操作超时。

描述

当_USE_LABEL == 1 时，本函数才有效。

范例 2.25：

```
char str[12];

/* Get volume label of the default drive */
f_getlabel("", str, 0);

/* Get volume label of the drive 2 */
f_getlabel("2:", str, 0);
```

2.26 f_setlabel

设置/删除一个卷的标签。

```
FRESULT f_setlabel (
    const TCHAR* label      /* Pointer to the volume label to set */
)
```

参数

label 以 null 结尾的字符串指针,指定的卷标。

返回值

FR_OK	操作成功。
FR_DISK_ERR	由于底层磁盘 I/O 函数中的错误，而导致该函数失败。
FR_INT_ERR	由于一个错误的 FAT 结构或一个内部错误，而导致该函数失败。
FR_NOT_READY	由于驱动器中没有存储介质或任何其他原因，而导致磁盘驱动器无法工作。
FR_INVALID_NAME	给定的字符串是无效的路径名。
FR_WRITE_PROTECTED	驱动器被写保护。
FR_INVALID_DRIVE	无效的驱动器号。
FR_NOT_ENABLED	工作区的逻辑驱动器未登记。
FR_NO_FILESYSTEM	驱动器上没有有效 FAT 卷。
FR_TIMEOUT	操作超时。

描述

字符串有一个驱动数字时,卷标将被设置为指定的卷驱动器号。如果没有,卷标将被设置为默认的驱动。如果给定的卷标长度为零,卷上的卷标将被删除。卷标签的格式类似于短文件名但有一些差异如下所示：：

- 11 字节或更少的长度作为转换成 OEM 代码页。LFN 扩展不应用于卷标。
- 不能包含时间
- 可以在卷标的任何地方包含空格。但尾部的空格被截断了。

备注：这是参考系统（Windows）中对处理用标题\ xE5 卷标的问题。为了避免这个问题，这个功能拒绝这些卷标为无效的名称。当_FS_READONLY == 0 and _USE_LABEL == 1 时可用。

范例 2.26：

```
/* Set volume label to the default drive */
f_setlabel("DATA DISK");

/* Set volume label to the drive 2 */
f_setlabel("2:DISK 3 OF 4");

/* Remove volume label of the drive 2 */
f_setlabel("2:");
```

2.27 f_mkfs

在驱动器上创建一个文件系统（俗称“格式化”）

```
FRESULT f_mkfs (
    const TCHAR* path,    /* Logical drive number */
    BYTE sfd,             /* Partitioning rule 0:FDISK, 1:SFD */
    UINT au               /* Size of allocation unit in unit of byte or sector */
)
```

参数

path 待格式化的逻辑驱动器号(0-9)。

sfd 当给定 0 时，首先在驱动器上的第一个扇区创建一个分区表，然后文件系统被创建在分区上。这被称为 FDISK 格式化，用于硬盘和存储卡。当给定 1 时，文件系统从第一个扇区开始创建，而没 有分区表。这被称为超级软盘(SFD)格式化，用于软盘和可移动磁盘。

au 指定每簇中以字节为单位的分配单元大小。数值必须是 0 或从 512 到 32K 之间 2 的幂。当指 定 0 时，簇大小取决于卷大小。

返回值

FR_OK (0)函数成功。

FR_INVALID_DRIVE 驱动器号无效。

FR_NOT_READY 由于驱动器中没有存储介质或任何其他原因，而导致磁盘驱动器无法工作。

FR_WRITE_PROTECTED 驱动器被写保护。

FR_NOT_ENABLED 逻辑驱动器没有工作区。

FR_DISK_ERR 由于底层磁盘 I/O 函数中的错误，而导致该函数失败。

FR_MKFS_ABORTED 由于下列原因之一，而导致函数在开始格式化前终止：

a) 磁盘容量太小

b) 参数无效

c) 该驱动器不允许的簇大小。

描述

f_mkfs 函数当_FS_READONLY == 0 并且_USE_MKFS == 1 时可用。

f_mkfs 函数在驱动器中创建一个 FAT 文件系统。对于可移动媒介，有两种分区规则：FDISK 和 SFD，通过参数 PartitioningRule 选择。FDISK 格式在大多数情况下被推荐使用。该函数当前不支持多分区，因此，物理驱动器上已存在的分区将被删除，并且重新创建一个占据全部磁盘空间的新分区。

根据 Microsoft 发布的 FAT 规范，FAT 分类：FAT12/FAT16/FAT32，由驱动器上的簇数决定。因此，选 择哪种 FAT 分类，取决于卷大小和指定的簇大小。簇大小影响文件系统的性能，并且大簇会提高性能。

2.28 f_fdisk

划分一个物理驱动器（即分区）

```
FRESULT f_fdisk (
    BYTE pdrv,          /* Physical drive number */
    const DWORD szt[],  /* Pointer to the size table for each partitions */
    void* work          /* Pointer to the working buffer */
)
```

参数

pdrv——指定要划分的物理驱动器

szt[]——分区映象表，必须有四个项目。

Work——指向函数工作区的指针。其大小必须至少为_MAX_SS 字节。

返回值

FR_OK (0) 划分成功。

FR_NOT_READY 由于驱动器中没有存储介质或任何其他原因，而导致磁盘驱动器无法工作。

FR_WRITE_PROTECTED 驱动器被写保护。

FR_DISK_ERR 由于底层磁盘 I/O 函数中的错误，而导致该函数失败。

FR_INVALID_PARAMETER 所给参数无效或不一致。

描述

f_fdisk 函数创建一个分区表到物理驱动器的 MBR。分区规则为通用 FDISK 格式，所以可以创建多达四个主分区。不支持扩展分区。szt[]指定了如何划分物理驱动器。第一个项目指定第一个主分区的大小，第四个项目指定第四个主分区。如果其值小于或等于 100，表示分区占整个磁盘空间的百分比。如果大于 100，则表示以扇区为单位的分区大小。

提示

在_FS_READONLY == 0、_USE_MKFS == 1 并且_MULTI_PARTITION == 2 时可用。

范例 2.28：

```
/* Volume management table defined by user (required when _MULTI_PARTITION == 1) */
PARTITION VolToPart[] =
{
    {0, 1},     /* Logical drive 0 ==> Physical drive 0, 1st partition */
    {0, 2},     /* Logical drive 1 ==> Physical drive 0, 2nd partition */
    {1, 0}      /* Logical drive 2 ==> Physical drive 1, auto detection */
};
```

```
/* Initialize  a brand-new disk drive mapped to physical drive 0 */
FATFS fs;
DWORD plist[] = {50, 50, 0, 0};    /* Divide drive into two partitions */
BYTE work[_MAX_SS];

f_fdisk(0, plist, work);       /* Divide physical drive 0 */

f_mount(&fs, "0:", 0);          /* Register work area to the logical drive 0 */
f_mkfs("0:", 0, 0);             /* Create FAT volume on the logical drive 0. 2nd argument is ignored. */
f_mount(0, "0:", 0);            /* Unregister work area from the logical drive 0 */

f_mount(&fs, "1:", 0);          /* Register a work area to the logical drive 1 */
f_mkfs("1:", 0, 0);             /* Create FAT volume on the logical drive 1. 2nd argument is ignored. */
f_mount(0, "1:", 0);            /* Unregister work area from the logical drive 1 */
```

2.29 f_gets

从文件中读取一个字符串

```
TCHAR* f_gets (
    TCHAR* buff,      /* Pointer to the string buffer to read */
    int len,          /* Size of string buffer (characters) */
    FIL* fp           /* Pointer to the file object */
)
```

参数

buff 存储读取字符串的读缓冲区指针。

len 读缓冲区大小

fp 打开的文件对象结构指针。

返回值

当函数成功后，buff 将被返回。

描述

f_gets 函数当_USE_STRFUNC == 1 或者_USE_STRFUNC == 2 时可用。如果_USE_STRFUNC == 2，文件中包含的'\r'则被去除。

f_gets 函数是 f_read 的一个封装函数。当读取到'\n'、文件结束或缓冲区被填冲了 len - 1 个字符时，读 操作结束。读取的字符串以'\0'结束。当文件结束或读操作中发生了任何错误，f_gets()返回一个空字符串。 可以使用宏 f_eof()和 f_error()检查 EOF 和错误状态。

2.30 f_putc

向文件中写入一个字符。

```
int f_putc (
    TCHAR  c,      /* A character to be output */
    FIL* fp        /* Pointer to the file  object */
)
```

参数

c 待写入的字符。

fp 打开的文件对象结构的指针。

返回值

当字符被成功地写入后，函数返回该字符。由于磁盘满或任何错误而导致函数失败，将返回 EOF。

描述

f_putc 函数当(_FS_READONLY == 0)&&(_USE_STRFUNC == 1 || _USE_STRFUNC == 2)时可用。

当_USE_STRFUNC == 2 时，字符'\n'被转换为"\r\n"写入文件中。

f_putc 函数是 f_write 的一个封装函数。

2.31 f_puts

向文件中写入一个字符串。

```
int f_puts (
    const TCHAR* str, /* Pointer to the string to be output */
    FIL* fp           /* Pointer to the file object */
)
```

参数

Str 待写入的'\0'结尾的字符串的指针。'\0'字符不会被写入。

fp 打开的文件对象结构的指针。

返回值

函数成功后，将返回写入的字符数。由于磁盘满或任何错误而导致函数失败，将返回 EOF。

描述

f_puts()当(_FS_READONLY == 0)&&(_USE_STRFUNC == 1 || _USE_STRFUNC == 2)时可用。

当_USE_STRFUNC == 2 时，字符串中的'\n'被转换为"\r\n"写入文件中。

f_puts()是 f_putc()的一个封装函数。

2.32 f_printf

向文件中写入一个格式化字符串。

```
int f_printf (
    FIL* fp,              /* Pointer to the file object */
    const TCHAR* fmt,     /* Pointer to the format string */
    ...                   /* Optional arguments... */
)
```

参数

fp 已打开的文件对象结构的指针。

fmt '\0'结尾的格式化字符串指针。

... 可选参数。

返回值

函数成功后，将返回写入的字符数。由于磁盘满或任何错误而导致函数失败，将返回 EOF。

描述

f_printf 函数当(_FS_READONLY == 0)&&(_USE_STRFUNC == 1 || _USE_STRFUNC == 2)时可用。
当_USE_STRFUNC == 2 时，包含在格式化字符串中的'\n'将被转换成"\r\n"写入文件中。
f_printf 函数是 f_putc 和 f_puts 的一个封装函数。

范例 32：

```
    f_printf(&fil, "%d", 1234);      /* "1234" */
    f_printf(&fil, "%6d,%3d%%", -200, 5);  /* "-200,  5%" */
    f_printf(&fil, "%-6u", 100);   /* "100   " */
    f_printf(&fil, "%ld", 12345678L);   /* "12345678" */
    f_printf(&fil, "%04x", 0xA3);       /* "00a3" */
    f_printf(&fil, "%08LX", 0x123ABC);/* "00123ABC" */
    f_printf(&fil, "%016b", 0x550F);  /* "0101010100001111" */
    f_printf(&fil, "%s", "String"); /* "String" */
    f_printf(&fil, "%-4s", "abc");   /* "abc " */
    f_printf(&fil, "%4s", "abc");       /* " abc" */
    f_printf(&fil, "%c", 'a');   /* "a" */
    f_printf(&fil, "%f", 10.0);      /* f_printf lacks floating point support */
```

2.33 f_tell

获取一个文件的当前读写指针。

```
DWORD f_tell (
   FIL* fp     /* [IN] File object */
);
```

参数

fp 指向打开文件对象结构的指针。

返回值

返回文件的当前读/写指针。

描述

在这个版本里，f_tell 函数是以一个宏来实现的。

```
#define f_tell(fp) ((fp)->fptr)
```

2.34 f_eof

测试一个文件的文件末尾。

```
int f_eof (
   FIL* fp     /* [IN] File object */
);
```

参数

fp 指向打开文件对象结构的指针。

返回值

如果读/写指针到达文件末尾，f_eof 函数返回一个非零值；否则返回 0。

描述

在这个版本里，f_eof 函数是以一个宏来实现的。

```
#define f_eof(fp) ((int)((fp)->fptr == (fp)->fsize))
```

2.35 f_size

获取一个文件的大小。

```
DWORD f_size (
   FIL* fp     /* [IN] File object */
);
```

参数

fp 指向打开文件对象结构的指针。

返回值

返回文件的大小，单位为字节。

描述

在这个版本里，f_size 函数是以一个宏来实现的。

```
#define f_size(fp) ((fp)->fsize)
```

2.36 f_error

测试一个文件是否出错。

```
int f_error (
   FIL* fp     /* [IN] File object */
);
```

参数

fp 指向打开文件对象结构的指针。

返回值

如果有错误返回非零值；否则返回 0。

描述

在这个版本里，f_error 函数是以一个宏来实现的。

```
#define f_error(fp) ((fp)->flag)
```

3. 磁盘 IO 接口函数

由于 FatFs 模块完全与磁盘 I/O 层分开，因此底层磁盘 I/O 需要下列函数去读/写物理磁盘以及获取当前时间。由于底层磁盘 I/O 模块并不是 FatFs 的一部分，因此它必须由用户提供。

3.1 disk_status

获取当前磁盘的状态。

```
DSTATUS disk_status (
  BYTE pdrv       /* [IN] Physical drive number */
);
```

参数

pdev 指定待确认的物理驱动器号。

返回值

磁盘状态，是下列标志的组合：

STA_NOINIT 指示磁盘驱动器还没有被初始化。当系统复位、磁盘移除和 disk_initialize 函数失败时，该标志被设置；当 disk_initialize 函数成功时，该标志被清除。

STA_NODISK 指示驱动器中没有存储介质。当安装了磁盘驱动器后，该标志始终被清除。

STA_PROTECTED 指示存储介质被写保护。在不支持写保护缺口的驱动器上，该标志始终被清除。当 STA_NODISK 被设置时，该标志无效。

范例 3.1：移植时的修改

```
DSTATUS disk_status (
    BYTE pdrv          /* Physical drive nmuber to identify the drive */
)
{
    return 0;/*暂时不需要，直接返回 0*/
}
```

3.2 disk_initialize

初始化磁盘驱动器。

```
DSTATUS disk_initialize (
  BYTE pdrv              /* [IN] Physical drive number */
);
```

参数

pdrv 指定待初始化的物理驱动器号。

返回值

disk_initialize 函数返回一个磁盘状态作为结果。磁盘状态的详情，参考 disk_status 函数。

描述

disk_initialize 函数初始化一个物理驱动器。函数成功后，返回值中的 STA_NOINIT 标志被清除。

disk_initialize 函数被 FatFs 模块在卷挂载过程中调用，去管理存储介质的改变。当 FatFs 模块起作用时，或卷上的 FAT 结构可以被瓦解时，应用程序不能调用该函数。可以使用 f_mount 函数去重新初始化文件系统。

范例 3.2：移植时的修改

```
DSTATUS disk_initialize (
    BYTE pdrv                   /* Physical drive nmuber to identify the drive */
)
{
    u8 result;
    switch (pdrv)
    {
        case SD : result = SD_Initialize();          /*SD 卡初始化*/
                                break;
        case ATA :
                                break;
        case MMC :
                                break;
        case USB :
                                break;
    }
    if(!result)return RES_OK;
    return STA_NOINIT;
}
```

3.3 disk_read

从磁盘驱动器中读取扇区数据。

```
DRESULT disk_read (
    BYTE pdrv,        /* [IN] Physical drive number */
    BYTE* buff,       /* [OUT] Pointer to the read data buffer */
    DWORD sector,     /* [IN] Start sector number */
    UINT count        /* [IN] Number of sectros to read */
);
```

参数

pdrv 指定物理驱动器号。

buff 存储读取数据的缓冲区的指针。该缓冲区大小需要满足要读取的字节数(扇区大小*扇区总数)。由上层指定的存储器地址可能会也可能不会以字边界对齐。

Sector 指定在逻辑块地址(LBA)中的起始扇区号。

Count 指定要读取的扇区数(1-128)。

返回值

RES_OK (0)函数成功

RES_ERROR 在读操作过程中发生了不能恢复的硬错误。

RES_PARERR 无效的参数。

RES_NOTRDY 磁盘驱动器还没被初始化。

范例 3.3: 移植时的修改

```
DRESULT disk_read (
    BYTE pdrv,        /* Physical drive nmuber to identify the drive */
    BYTE *buff,       /* Data buffer to store read data */
    DWORD sector,     /* Sector address in LBA */
    UINT count        /* Number of sectors to read */
)
{
    u8 result;
    switch (pdrv)
    {
        case SD :   result = SD_ReadDisk(buff,sector,count);        /*读扇区*/
                    break;
        case ATA :
                    break;
        case MMC :
                    break;
        case USB :
                    break;
    }
    if(!result)return RES_OK;
    return RES_PARERR;
}
```

3.4 disk_write

向磁盘驱动器中写入扇区。在只读配置中，不需要此函数。

```
DRESULT disk_write (
    BYTE drv,             /* [IN] Physical drive number */
    const BYTE* buff, /* [IN] Pointer to the data to be written */
    DWORD sector,         /* [IN] Sector number to write from */
    UINT count            /* [IN] Number of sectors to write */
);
```

参数

drv 指定物理驱动器号。

buff 存储写入数据的缓冲区的指针。由上层指定的存储器地址可能会也可能不会以字边界对齐。

sector 指定在逻辑块地址(LBA)中的起始扇区号。

count 指定要写入的扇区数(1-255)。

返回值

RES_OK (0)函数成功

RES_ERROR 在读操作过程中发生了不能恢复的硬错误。

RES_WRPRT 存储介质被写保护。

RES_PARERR 无效的参数。

RES_NOTRDY 磁盘驱动器还没被初始化。

范例 3.4：移植时的修改

```
#if _USE_WRITE
DRESULT disk_write (
    BYTE pdrv,              /* Physical drive nmuber to identify the drive */
    const BYTE *buff, /* Data to be written */
    DWORD sector,           /* Sector address in LBA */
    UINT count              /* Number of sectors to write */
)
{
    u8 result;
    switch (pdrv)
    {
        case SD : result = SD_WriteDisk((u8 *)buff,sector,count);  /*写扇区*/
                            break;
        case ATA :          break;
        case MMC :          break;
        case USB :          break;
    }
    if(!result)return RES_OK;
    return RES_PARERR;
}
#endif
```

3.5 disk_ioctl

控制设备特定的功能以及磁盘读写以外的其它功能。

```
DRESULT disk_ioctl (
    BYTE pdrv,      /* Physical drive nmuber (0..) */
    BYTE cmd,       /* Control code */
    void *buff      /* Buffer to send/receive control data */
)
```

参数

pdrv 指定驱动器号(1-9)。

cmd 指定命令代码。

buff 取决于命令代码的参数缓冲区的指针。当不使用时，指定一个 NULL 指针。

返回值

RES_OK (0)函数成功。

RES_ERROR 发生错误。

RES_PARERR 无效的命令代码。

RES_NOTRDY 磁盘驱动器还没被初始化。

描述

FatFs 模块只使用下述与设备无关的命令，没有使用任何设备相关功能。

命令	描述
CTRL_SYNC	确保磁盘驱动器已经完成等待写过程。当磁盘 I/O 模块有一个写回高速缓存时，立即冲洗脏扇区。在只读配置中，不需要该命令。
GET_SECTOR_SIZE	返回驱动器的扇区大小赋给 Buffer 指向的 WORD 变量。在单个扇区大小配置中(_MAX_SS 为 512)，不需要该命令。
GET_SECTOR_COUNT	返回总扇区数赋给 Buffer 指向的 DWORD 变量。只在 f_mkfs 函数中，使用了该命令。
GET_BLOCK_SIZE	返回以扇区 为单位的存 储阵列的擦 除块大小赋 给 Buffer 指向的 DWORD 变量。当擦除块大小未知或是磁盘设备时，返回 1。只在 f_mkfs 函数中，使用了该命令。

范例 3.5：移植时的修改

```
#if _USE_IOCTL
DRESULT disk_ioctl (
    BYTE pdrv,        /* Physical drive nmuber (0..) */
    BYTE cmd,         /* Control code */
    void *buff        /* Buffer to send/receive control data */
)
{
    u8 result= RES_OK;
    switch (pdrv)
    {
        case SD : switch(cmd)
                  {
                      case CTRL_SYNC:
                                  SD_CS=0;
                                  if(SD_WaitReady())result = RES_ERROR;/*等待卡准备好*/
                                  SD_CS=1;
                                  break;
                      case GET_SECTOR_SIZE:
                                  *(WORD*)buff = 512;/*sd 卡扇区字节数*/
                                  break;
                      case GET_BLOCK_SIZE:
                                  *(WORD*)buff = 8;/*块大小*/
                                  break;
                      case GET_SECTOR_COUNT:
                                  *(DWORD*)buff = SD_GetSectorCount();/*获取 SD 卡的总扇区数*/
                                  break;
                      default:    result = RES_PARERR;
                                  break;
                  }
                                  break;
        case ATA :
                                  break;
        case MMC :
                                  break;
        case USB :
                                  break;
    }
    if(!result)return RES_OK;
    return RES_PARERR;
}
#endif
```

3.6 get_fattime

获取当前时间。

```
DWORD get_fattime (void);
```

参数

void

返回值

返回的当前时间被打包进一个 DWORD 数值。各位域定义如下：

bit31:25 年，从 1980 年开始算起(0..127)

bit24:21 月(1..12)

bit20:16 日(1..31)

bit15:11 时(0..23)

bit10:5 分(0..59)

bit4:0 秒/2(0..29)，由此可见 FatFs 的时间分辨率为 2 秒

描述

get_fattime 函数必须返回任何有效的时间，即使系统不支持实时时钟。如果返回一个 0，则文件将没有一个有效的时间。在只读配置中，不需要此函数。

范例 3.6：移植时的修改

```
/*31-25: Year(0-127 org.1980), 24-21: Month(1-12), 20-16: Day(1-31) */
/*15-11: Hour(0-23), 10-5: Minute(0-59), 4-0: Second(0-29 *2) */
DWORD get_fattime (void)
{
        u32 date;
        RTC_Get();          /*读 RTC 得到当前时间*/
        date =
        (
           ((RTC.year - 1980) << 25)     |
             (RTC.month <<    21 )       |
             (RTC.day <<    16 )         |
             (RTC.hour << 11 )           |
             (RTC.min << 5    )          |
             ( RTC.sec )
        );

    return date;
}
```

4. fatfs 几个常用结构体及函数返回代码分析

4.1 FATFS 结构体

FATFS 结构体（文件系统对象）是逻辑驱动器的动态工作区。它的相关信息是由 f_mount 函数创建。应用程序不能修改该结构体任何成员。

```
/* File system object structure (FATFS) */
typedef struct {
    BYTE    fs_type;        /* 系统类型(为 0 时系统没有被挂载) */
    BYTE    drv;            /* 对应实际设备驱动号*/
    BYTE    csize;          /* 每个簇的扇区数目(1,2,4...128) */ //（簇：文件数据分配的基本单位）
    BYTE    n_fats;         /* 文件分配表的数目(1 or 2) */
    BYTE    wflag;          /* win[] flag (b0:dirty) */ //文件是否改动的标志，为 1 时要回写。
    BYTE    fsi_flag;       /* 文件信息回写标志 (b7:disabled, b0:dirty) */
    WORD    id;             /* 文件系统加载 ID */
    WORD    n_rootdir;      /* 根目录区目录项的数目 (FAT12/16) */
#if _MAX_SS != _MIN_SS
    WORD    ssize;          /*每扇区多少字节 (512, 1024, 2048 or 4096) */
#endif
#if _FS_REENTRANT
    _SYNC_t sobj;           /* 允许重入，则定义同步对象 */
#endif
#if !_FS_READONLY
    DWORD last_clust;       /* 最新分配的簇 */
    DWORD free_clust;       /* 空闲簇 */
#endif
#if _FS_RPATH
    DWORD cdir;             /*  使用相对路径，则要存储文件到系统当前目录(0:root) */
#endif
    DWORD n_fatent;         /* 文件分配表占用的扇区，n_fatent=数据簇数目+2*/
    DWORD fsize;            /* 每 FAT 表有多少个扇区 */
    DWORD volbase;          /* Volume start sector */
    DWORD fatbase;          /* 文件分配表开始扇区号*/
    DWORD dirbase;          /* 根目录开始扇区(FAT32:Cluster#) */
    DWORD database;         /* 数据起始扇区 */
    DWORD winsect;          /*  win[]中当前指定的扇区 */
    BYTE    win[_MAX_SS];   /* 扇区操作缓存(and file data at tiny cfg) */
} FATFS;
```

4.2 FIL 结构体

本结构体（文件对象）用保存一个已打开文件的相关信息。它是由 f_open 函数创建并通过 f_close 函数丢弃。应用程序不能修改该结构体任何成员。

```
/* File object structure (FIL) */
typedef struct {
    FATFS*  fs;             /* 指向相应文件系统对象 (**do not change order**) */
    WORD    id;             /*  自身文件系统挂载 id 号，即 fs->id*/
    BYTE    flag;           /* 文件状态标志 */
    BYTE    err;            /* 错误代码 */
    DWORD fptr;             /*  读写指针(当文件刚打开时为 0) */
    DWORD fsize;            /* 文件大小（按字节计算） */
    DWORD sclust;           /* 文件起始簇 (0:没有簇链，始终为 0 时，FSIZE 为 0) */
    DWORD clust;            /* 文件当前操作的簇 (not valid when fprt is 0)*/
    DWORD dsect;            /* 文件当前操作的扇区(0:invalid) */
#if !_FS_READONLY
    DWORD dir_sect;         /* 包含路径入口的扇区号 */
    BYTE*   dir_ptr;        /* 目录入口指针*/
#endif
#if _USE_FASTSEEK
    DWORD*      cltbl;      /* 指向查找映射表的簇(Nulled on file open) */
#endif
#if _FS_LOCK
    UINT    lockid;         /*文件锁 ID 号(index of file semaphore table Files[]) */
#endif
#if !_FS_TINY
    BYTE    buf[_MAX_SS];   /*文件读写缓冲区 */
#endif
} FIL;
```

4.3 DIR 结构体

DIR 结构体用于工作区由 f_oepndir，f_readdir，f_findfirst 和 f_findnext 函数来读取一个目录。应用程序不能修改该结构体任何成员。

```
/*Directory object structure (DIR)*/
typedef struct {
    FATFS*  fs;             /*  指向相应文件系统对象  (**do not change order**)*/
    WORD    id;             /*  文件系统加载 ID(**do not change order**) */
    WORD    index;          /*  目前读写索引代码*/
    DWORD sclust;           /*  文件数据区开始簇(0:Root dir) */
    DWORD clust;            /*  目前处理的簇  */
    DWORD sect;             /*  目前簇里对应的扇区  */
    BYTE*   dir;            /*  指向当前在 win 中的短文件名入口项*/
    BYTE*   fn;             /*  指向短文件名{file[8],ext[3],status[1]} */
#if _FS_LOCK
    UINT    lockid;         /*  文件锁  ID 号  (index of file semaphore table Files[]) */
#endif
#if _USE_LFN
    WCHAR*      lfn;        /*  指向长文件名缓冲*/
    WORD    lfn_idx;        /* Last matched LFN index number (0xFFFF:No LFN) */
#endif
#if _USE_FIND
    const TCHAR*    pat;    /* Pointer to the name matching pattern */
#endif
} DIR;
```

4.4 FILINFO 结构体

这个结构主要描述文件的状态信息，包括文件名 13 个字符（8+.+3+\0）、属性、修改时间等。应用程序不能修改该结构体任何成员。

```
typedef struct {
    DWORD fsize;            /*  文件大小*/
    WORD    fdate;          /*  最后修改日期*/
    WORD    ftime;          /*  最后修改时间*/
    BYTE    fattrib;        /*文件属性  */
    TCHAR   fname[13];      /*  短文件名(8.3 format) */
#if _USE_LFN
    TCHAR* lfname;          /*  指向长文件名缓冲区*/
    UINT    lfsize;         /*  长文件名缓冲区大小*/
#endif
} FILINFO;
```

4.5 fatfs_API 函数返回代码

在 FATFS API 中，大部分返回代码为枚举类型 FRESULT。当一个函数操作成功，则返回零，否则返回非零值，指示错误的类型。

```
/* File function return code (FRESULT) */
typedef enum {
    FR_OK = 0,                  /* (0)  操作成功*/
    FR_DISK_ERR,                /* (1)  由于底层磁盘 I/O 函数中的错误，而导致该函数失败*/
    FR_INT_ERR,                 /* (2)  由于一个错误的 FAT 结构或一个内部错误，而导致该函数失败 */
    FR_NOT_READY,               /* (3)  由于驱动器中没有存储介质或任何其他原因，而导致磁盘驱动器无法工作*/
    FR_NO_FILE,                 /* (4)  找不到该文件 */
    FR_NO_PATH,                 /* (5)  找不到该路径 */
    FR_INVALID_NAME,            /* (6)  文件名无效 */
    FR_DENIED,                  /* (7)  访问被拒绝 */
    FR_EXIST,                   /* (8)  该文件已存在，在创建文件和文件夹时才有可能被返回 */
    FR_INVALID_OBJECT,          /* (9)  文件对象无效 */
    FR_WRITE_PROTECTED,         /* (10) 在存储介质被写保护的情况下，以写模式打开或创建文件对象*/
    FR_INVALID_DRIVE,           /* (11) 驱动器号无效 */
    FR_NOT_ENABLED,             /* (12) 逻辑驱动器没有工作区*/
    FR_NO_FILESYSTEM,           /* (13) 磁盘上没有有效的 FAT 卷 */
    FR_MKFS_ABORTED,            /* (14) f_mkfs()由于任何参数错误而终止运行 */
    FR_TIMEOUT,                 /* (15) 操作超时 */
    FR_LOCKED,                  /* (16) The operation is rejected according to the file sharing policy */
    FR_NOT_ENOUGH_CORE,         /* (17) 没有足够的内存来运行 */
    FR_TOO_MANY_OPEN_FILES,     /* (18) 打开对象的数量已经达到了最大值 */
    FR_INVALID_PARAMETER        /* (19) 给定参数无效 */
} FRESULT;
```