《振南 **znFAT--**嵌入式 **FAT32** 文件系统设计与实现》一书

【上下册】已正式出版发行

全国各渠道全面发售

（在当当、京东、亚马逊、淘宝等网络平台上搜索

关键字"**znFAT**"即可购买，各地实体书店也有售）

此书是市面上 唯一 一套详细全面而深入讲解嵌入式存储技术、**FAT32** 文件系统、**SD** 卡驱动与应用方面的专著。全套书一共 **25** 章，近 **70** 万字。从基础、提高、实践、剖析、创新、应用等很多方面进行阐述，力求通俗，振南用十年磨一剑的精神编著此书，希望对广大工程师与爱好者产生参考与积极意义。

此书在各大电子技术论坛均有**长期的**「抢楼送书活动」，如 21IC、elecfans 等等。

振南的【ZN-X 开发板】是市面上唯一全模块化、多元化的开发板，可支持 51、AVR、STM32(M0/M3/M4)

详情请关注 **www. znmcu. cn** （振南个人主页!!）

第2章

更及核心，文件创建：修改FAT表实现文件创建功能

本章将实现文件与目录的创建，其中涉及大量构造性的工作，比如对文件目录项、功能扇区数据，乃至FAT32的核心——FAT表与簇链的构造。如果说上册只是对FAT32的“观摩”的话，那么从这一章开始就要对它“动刀”了。这是危险的行为，因为只要稍有不慎就可能对数据造成破坏，甚至导致整个文件系统的崩溃。同时，在实现上也会比读取操作更难一些，因为我们要操作的数据对象不再是现成地存放在扇区中，而需要对其进行创造。

2.1 文件的创建

文件能够呈现在我们面前的根本原因就是目录簇中的文件目录项。而文件如何创建则主要包括了两部分工作：文件目录项的构造以及将它放到目录簇中去（即文件目录项的“落定”），如图2.1所示。

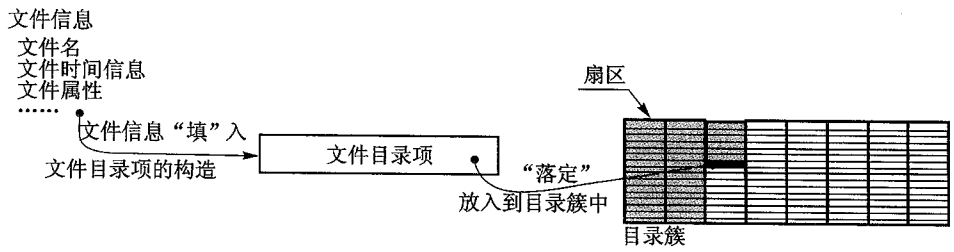

图2.1 文件创建的实现过程

2.1.1 文件目录项的构造

其实文件目录项的构造就像是一个“填表”的过程，把文件的相关信息填到文件目录项的字段中去，当然要遵循文件目录项的结构定义，实例如图2.2所示。图中完成了一个文件目录项的构造（日期和时间的计算方法其实就是上册第6章中时间参数解析的逆过程，即一个位段的合并过程）。接下来，将这个构造好的文件目录项写

入首目录簇中去，如图 2.3 所示。

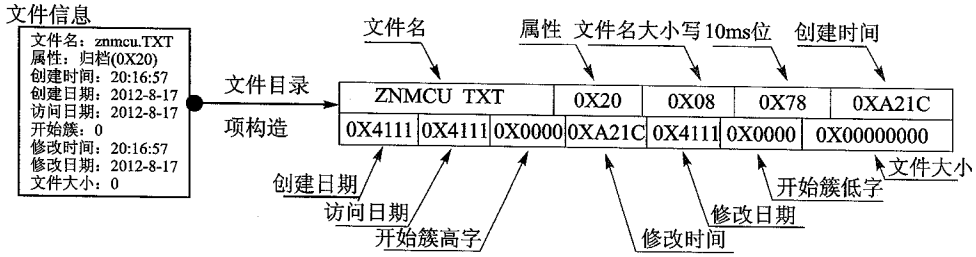

图 2.2　文件目录项的构造实例

```
5A 4E 4D 43 55 20 20 20  20 20 20 08 00 00 00 00  ZNMCU       .....
00 00 00 00 00 00 65 59  54 40 00 00 00 00 00 00  ......eYT@......
41 42 43 20 20 20 20 20  54 58 54 20 18 B5 CF 70  ABC     TXT .迪p
51 40 54 40 00 00 CC 70  51 40 03 00 84 00 00 00  Q@T@..蘧Q@..?...
5A 4E 4D 43 55 20 20 20  54 58 54 20 00 78 1C A2  ZNMCU   TXT .x.
11 41 11 41 00 00 1C A2  11 41 00 00 00 00 00 00  .A.A...?A.......
```

图 2.3　向首目录簇写入构造好的文件目录项

来看看首目录下是不是产生了一个新文件，如图 2.4 所示。果然有一个新的文件 ZNMCU. TXT，而且它的属性也与前面给定的文件信息参数是一致的。这就是文件创建的基础原型了。

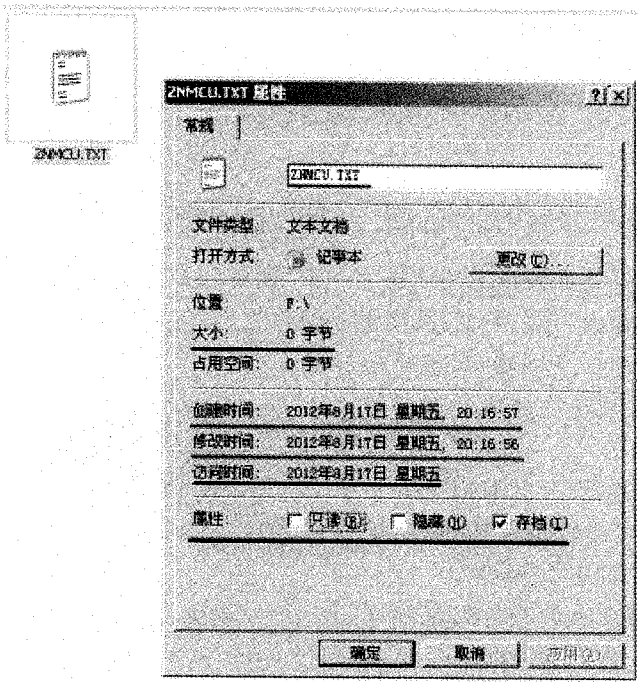

图 2.4　首目录中产生的新文件及其属性

具体的代码实现(znFAT.h)如下：

```
#define MAKE_TIME(h,m,s) ((((UINT16)h)<<11) + (((UINT16)m)<<5)
                          + (((UINT16)s)>>1))  //按时间位段定义合成时间字
#define MAKE_DATE(y,m,d) (((((UINT16)(y%100)) + 20)<<9) + (((UINT16)m)<<5)
                          + ((UINT16)d))  //按日期位段定义合成日期字
```

znFAT.c 代码如下：

```
UINT8 Fill_FDI(struct FDI *pfdi,INT8 *pfn,struct DateTime *pdt)
{
 //将 8·3 短文件名转为 11 字节的文件名字段数据
 Memory_Copy(((UINT8 *)(pfdi->Name)),....); //文件名字段数据填入文件目录项
 pfdi->Attributes=0X20; //将属性字节填入文件目录项(固定为归档,即 0X20)
 //判断主文件名与扩展名的大小写情况
 pfdi->LowerCase=....; 将大小写字节填入文件目录项
 pfdi->CTime10ms=(((pdt->time).sec)%2)? 0X78:0X00; //填入创建时间 10 ms 位
 time=MAKE_TIME((pdt->time).hour,(pdt->time).min,(pdt->time).sec);
 (pfdi->CTime)[0]=(UINT8)time; //填入创建时间
 (pfdi->CTime)[1]=(UINT8)(time>>8);
 date=MAKE_DATE((pdt->date).year,(pdt->date).month,(pdt->date).day);
 (pfdi->CDate)[0]=(UINT8)date; //填入创建日期
 (pfdi->CDate)[1]=(UINT8)(date>>8);
 //填入访问日期、修改时间与日期
 //开始簇高字、低字与文件大小字段暂置为 0
 return 0;
}
```

2.1.2 文件目录项的“落定”:写入目录簇

如果把构造好的文件目录项比喻为一个“便签”,那么将它写入目录簇的过程就如同将便签钉到墙上一样,所以,振南形象地称之为“落定”,如图 2.5 所示。

此时必须考虑一个问题:文件目录项具体应该落于何处?有两个重要的原则：

① 它不能覆盖现有的有效文件目录项,我们要寻找那些全空的位置,即数据为全 0,振南称之为“空位”;

② 它不能与现有的文件目录项同名,所以会涉及文件名的匹配。

具体的实现过程如图 2.6 所示。

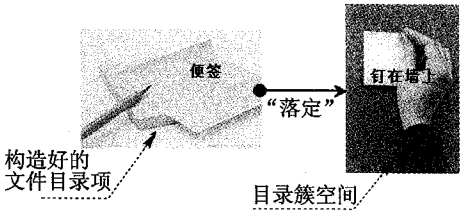

图 2.5 文件目录项的落定

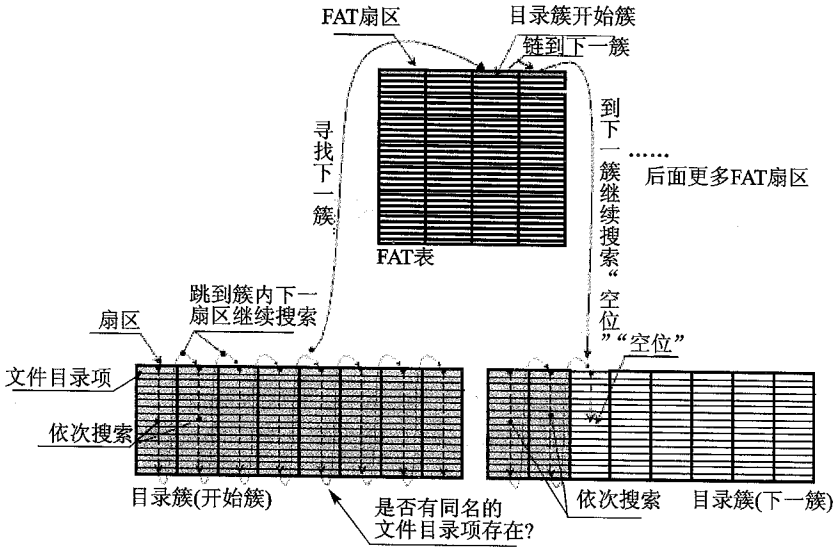

图 2.6　在目录簇中进行“空位搜索”与“同名检测”

它似乎和上册中实现打开文件函数(znFAT_Open_File)的时候，对目标文件目录项进行搜索的实现过程差不多。确实，它们本质上都是在搜索文件目录项，只不过这里搜索的是“空位”，同时也会对文件目录项进行简单的匹配。Settle_FDI 的具体代码实现(znFAT.c)如下：

```
UINT8 Settle_FDI(UINT32 dirclust,struct FDI * pfdi,UINT32 * psec,UINT8 * pn)
{
 UINT32 Cur_Clust = dirclust; //用于记录当前目录簇
 UINT32 sSec = 0; //用于记录簇的开始扇区
 struct FDIesInSEC * pfdis; //目录扇区结构指针
 UINT8 iSec = 0,iFDI = 0;
 //检查是否已有同名的文件目录项存在，搜索空位
 do
 {
  sSec = SOC(Cur_Clust); //计算簇的开始扇区
  for(iSec = 0;iSec<(Init_Args.SectorsPerClust);iSec ++ )
  {
   znFAT_Device_Read_Sector(nsec + iSec,znFAT_Buffer);
   pfdis = ((struct FDIesInSEC * )znFAT_Buffer);
   for(iFDI = 0;iFDI<16;iFDI ++ )
   {
    if(Memory_Compare((UINT8 * )((pfdis->FDIes) + iFDI),(UINT8 * )pfdi,11))
                                                    //比较文件名字段
    {
```

```
            (*psec) = sSec + iSec;  //★记录"空位"所在扇区
            (*pn) = iFDI;  //★记录"空位"在扇区中的位置
           return ERR_FDI_ALREADY_EXISTING; //已有同名文件存在
          }
          else
          {
           if(0 == ((((pfdis->FDIes)[iFDI]).Name)[0])) //如果是"空位"
           {
            (*psec) = sSec + iSec;  //★记录"空位"所在扇区
            (*pn) = iFDI;  //★记录"空位"在扇区中的位置
            Memory_Copy((UINT8 *)pfdi,
                  (UINT8 *)((((struct FDIesInSEC *)znFAT_Buffer)->FDIes) + iFDI),32);
            znFAT_Device_Write_Sector(sSec + iSec,znFAT_Buffer);  //回写扇区
            return 0;
           }
          }
         }
        }
        Cur_Clust = znFAT_GetNextCluter(Cur_Clust); //获取下一簇
       }while(!IS_END_CLU(Cur_Clust)); //如果不是最后一个簇,则继续循环
       //如果运行到这里,则说明当然簇中无空位
       return ERR_FDI_NO_SPARE_SPACE;
      }
```

这个函数运行之后就把文件目录项"落定"到"空位"上。同时,还将"空位"或同名文件目录项所在的扇区以及位置记录到了 psec 与 pn 指向的变量中。为什么要这么做,到后面就知道了。

有了上面的这些函数,我们就可以初步实现文件创建功能了,具体(znFAT.c)实现如下:

```
UINT8 znFAT_Create_File(struct FileInfo *pfi,INT8 *path,struct DateTime *pdt)
{
 UINT32 Cur_Cluster = 0,pos = 0;
 UINT8 res = 0;
 struct FDI fdi;
 INT8 *pfn;
 res = znFAT_Enter_Dir(path,&Cur_Cluster,&pos); //进入目录
 if(res)
 {
  return res; //进入目录失败,返回错误码
 }
 pfn = path + pos; //pfn 指向路径中的文件名
```

```
//文件名合法性检测
Fill_FDI(&fdi,pfn,pdt);  //构造文件目录项
res = Settle_FDI(Cur_Cluster,&fdi); //在当前目录簇中对文件目录项进行"落定"
if(!res)
{
 //将新建文件的信息装入 pfi 所指向的文件信息集合
 return 0;
}
else
{
 if(res == ERR_FDI_ALREADY_EXISTING) //如果有同名文件目录项
 {
  //将同名的文件信息装入 pfi 所指向的文件信息集合
 }
 return  res; //"落定"失败,返回错误码
}
}
```

这个函数使用了前面所实现的进入目录函数(znFAT_Enter_Dir),从而使其可以在指定的深层目录下创建文件。同时,在文件创建成功或者遇到同名文件时,又会将文件的相关信息装入到文件信息集合中。所以,此时文件无须再用 znFAT_Open_File 打开即可直接进行操作。

2.2 为自己开路:簇链的构造

2.2.1 目录簇的拓展

前面"空位搜索"(Prepare_To_Settle_FDI)的实现过程中,如果发现已无空位,则直接返回"无空闲空间"(ERR_FDI_NO_SPARE_SPACE)的错误,文件目录项的"落定"也随之流产,最终造成文件创建的失败。一个疑问随即而生:"一个目录下可以创建的文件难道还有数量限制?"当然不是。既然在目录的现有簇中已找不到空位来放置文件目录项,那我们就开辟一条新的道路。如何开辟?当然还是 FAT 簇链,它是 FAT32 中进行数据扩展和延伸的核心机制,如图 2.7 所示。

图 2.7 中,空簇找到之后(空簇在 FAT 表中的簇项值为 0)随即纳入到簇链之中,这样,空簇就与现有目录簇融为一体了。我们将文件目录项写到这个空簇中,在目录中便能看到新文件了。这样一来,FAT32 的目录中能够创建的文件数量岂不是就没有限制了?确实!只要有空簇就可以链上去,除非没有空簇可用了,这种时候就说明磁盘容量已满。

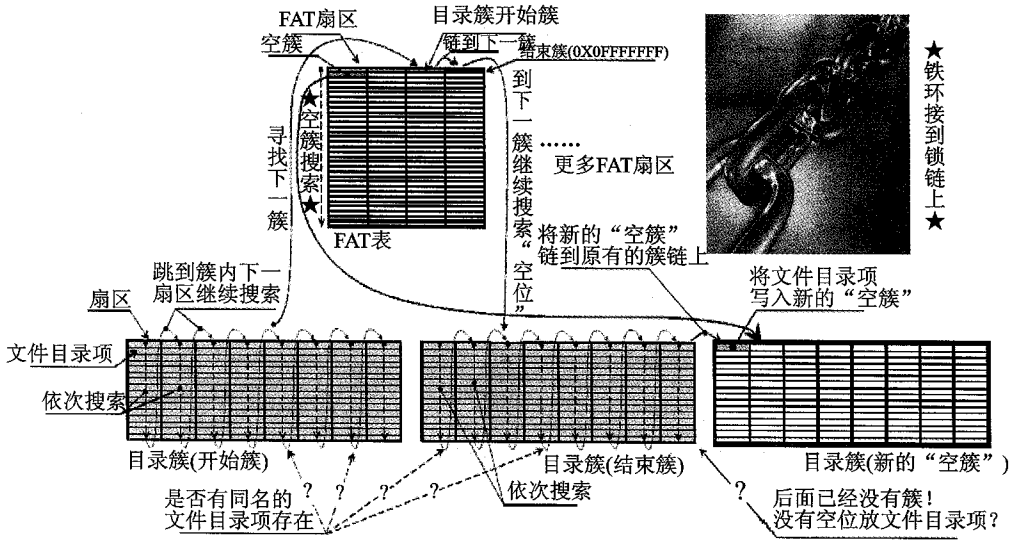

图 2.7　对目录簇进行拓展

2.2.2　寻找“路石”:空簇的查找

其实上面介绍的就是簇链的构造,它是 FAT32 中写操作的关键环节。这其中有一个非常重要的步骤,那就是对于空簇的查找。因为查找空簇所耗费的时间直接决定了簇链构造的速度,最终将影响写操作的整体效率。后面介绍“数据写入”和“数据狂飙”的时候,读者就能够更深刻地体会到空簇查找与簇链构造的效率对数据写入速度所产生的决定性作用了。关于空簇的查找方法,振南也进行了深入的研究,提出了几个方案,下面一一介绍给大家。

(1) 遍历查找

这种方法可以说是最原始、最简单、也是最容易想到的方法,同时效率也是比较低的。它从 FAT 表的最开始,逐一地对各个簇项进行遍历,直到某一个簇项的值为 0 为止。实现过程如图 2.8 所示。它实现起来也很简单,代码(znFAT.c)如下:

```
UINT8 Search_Free_Cluster_From_Start(UINT32 * nFreeCluster)
{
 UINT32 iSec = 0;
 UINT8   iItem = 0;
 struct FAT_Sec * pFAT_Sec;
 for(iSec = 0;iSec<Init_Args.FATsectors;iSec ++ )
 {
  znFAT_Device_Read_Sector(Init_Args.FirstFATSector + iSec,znFAT_Buffer);
                                                          //读取 FAT 扇区
  pFAT_Sec = (struct FAT_Sec * )znFAT_Buffer;
```

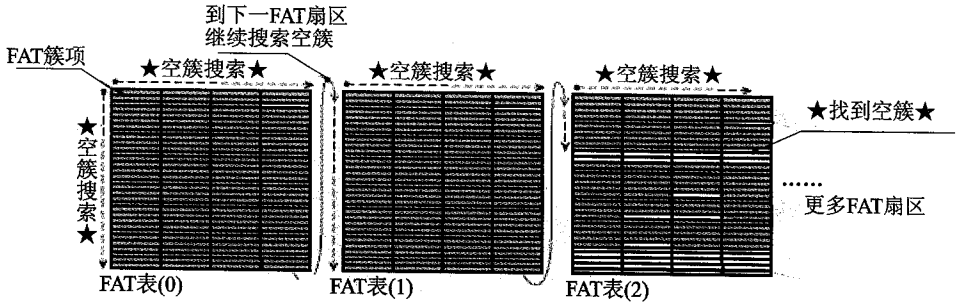

图 2.8 对 FAT 表进行遍历查找

```
for(iItem = 0;iItem<128;iItem ++ ) //遍历所有簇项,寻找空闲簇
{
 if(     (0 == (((pFAT_Sec - >items[iItem]). Item)[0]))
   && (0 == (((pFAT_Sec - >items[iItem]). Item)[1]))
   &&(0 == (((pFAT_Sec - >items[iItem]). Item)[2]))
   && (0 == (((pFAT_Sec - >items[iItem]). Item)[3])) ) //如果找到空簇
 {
  * nFreeCluster = ((iSec * 128) + iItem); //记录簇值
    return 0;
 }
}
}
return     1;
}
```

(2) 接力式遍历查找

上面介绍的遍历查找是比较慢的,而且每次都要从 FAT 表开头进行查找。为了使速度和效率有所提升,振南在这个基础上提出了"接力式遍历查找"的思想,就是每次查找都以上一次的位置为起点继续查找,减少每次从头查找所作的无用功。我们通过图 2.9 来详细说明。

在文件系统的初始化阶段(znFAT_Init)进行第一次空簇查找,它是对 FAT 表从头开始的(调用上面实现的 Search_Free_Cluster_From_Start 函数),最终可得到第一个可用的空簇(如图 2.9 中的 A),把它记录在一个变量中。代码(znFAT.h)实现如下:

```
struct znFAT_Init_Arg
{
 //文件系统核心参数
 UINT32 Free_Cluster; //空簇标记
};
```

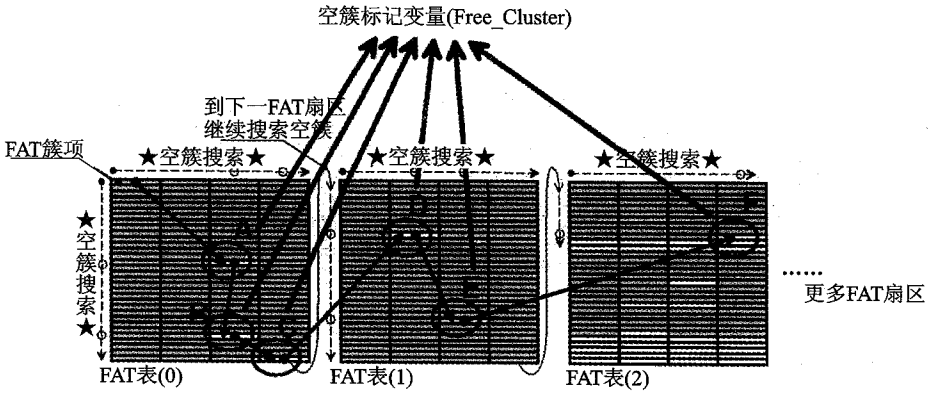

图 2.9　对 FAT 表进行接力式遍历查找

znFAT.c 代码如下：

```
UIN8 znFAT_Init(void)
{
 //如果有 MBR,则解析得到 DBR 扇区地址,否则 DBR 扇区地址为 0
 //读取 DBR 扇区,解析计算得到各核心参数
 if(Search_Free_Cluster_From_Start(&(Init_Args.Free_Cluster)))   //从头查找空簇
 {
  return 1; //如果空簇查找失败,则返回错误
 }
 return 0;
}
```

上面代码中用于记录空簇的变量 Free_Cluster 称之为“空簇标记”，是一个较为关键的参数，我们也将它纳入到了初始化参数集合(znFAT_Init_Arg)中。它提供了当前可用的空簇，同时又是查找下一个空簇的起点(比如要查找 D，就可以从当前 Free_Cluster 所记录的 C 开始查找)。如果有哪一个操作需要使用到空簇，那么它就可以把 Free_Cluster 中记录的簇直接拿来用，同时调用函数 Update_Free_Cluster 将其更新为下一个空簇(函数具体实现参见 znFAT 的完整源代码)。

有读者会说：“在 znFAT_Init 函数中对 FAT 表从头做遍历查找，不会让初始化过程变得很慢吗?”确实！在空簇的位置离 FAT 表开头比较远的时候就会导致这样的问题。那有没有办法可以避免这一问题的产生呢？振南提出过一种方法：在更新空簇标记(Free_Cluster)的同时也将其记录到一个特定的扇区中去，比如 DBR 与 FAT 之间的保留扇区。这样，在下一次 znFAT 初始化的时候，直接从这个扇区中把上次记录的空簇读出来即可，这样似乎就免去了每次初始化都从头遍历的步骤，形象的说明如图 2.10 所示。

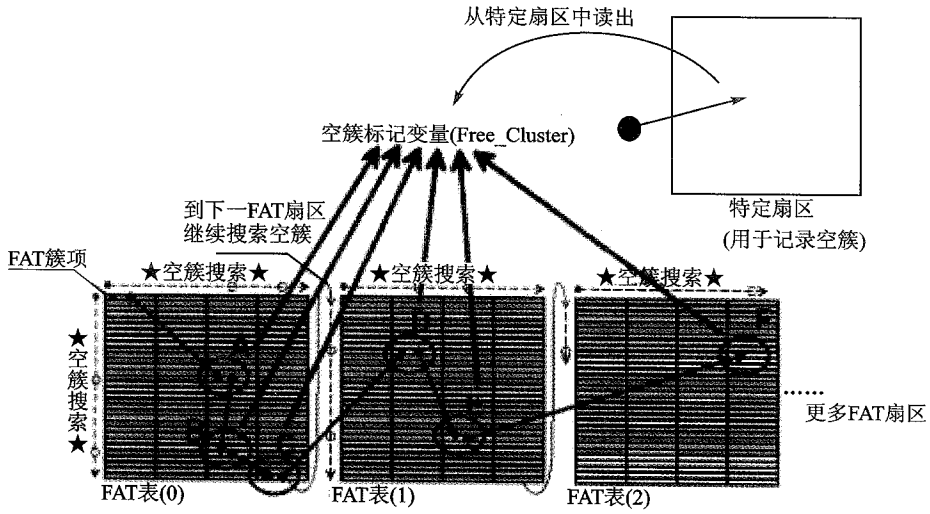

图 2.10　避免每次对 FAT 表从头查找的方法

但实际上，这种方法根本就行不通。为什么？原因就是我们自己约定的用于记录空簇的扇区在标准 FAT32 文件系统中是不认识或者说是不支持的。如果磁盘一直在 znFAT 方案下进行文件操作，那么不会有任何问题，我们会时刻维护这个扇区；但如果把它放到计算机上，使用带有标准 FAT32 文件系统的操作系统，比如 Windows，对它进行文件操作，那么这个特定扇区中记录的空簇是不会被更新的。再把它拿回到 znFAT 方案下，从扇区中所获取的空簇可能实际上已经被计算机使用过了，这将造成数据的覆盖和破坏。

此时，你一定会想："如果 FAT32 能够与我们共同维护一个用于记录空簇的扇区就好了！"实际上，FAT32 最初的设计者好像也在寻求一种快速查找空簇的方法，并且与振南碰撞出了"巧合的智慧火花"，具体是怎么回事？请看下文。

2.2.3　形同虚设的 FSINFO 扇区

关于 FAT32 中的 FSINFO 扇区，其实很多参考资料上都没有提及。一开始，振南也不知道它的存在。后来无意中看到了一篇文章，说它里面记录了空簇以及剩余的空簇数，它的性质和用途就如同上面介绍的"特定扇区"一样，而且受到 FAT32 的支持和维护。通常，FSINFO 扇区位于 DBR 的下一个扇区，具体如图 2.11 所示。

图 2.11 就是实际 FSINFO 扇区中的数据，显著特点就是有两个标记"RRaA"与"rrAa"，最后以"55AA"结束。我们关心的是空簇和剩余空簇数这两个参数，图中分别是 0X000069AD 与 0X000215BD。可以简单验证一下：使用剩余空簇数来计算磁盘的剩余存储空间，136 637×4 096＝559 665 152 字节。与 Windows 中的磁盘属性对比，如图 2.12 所示。

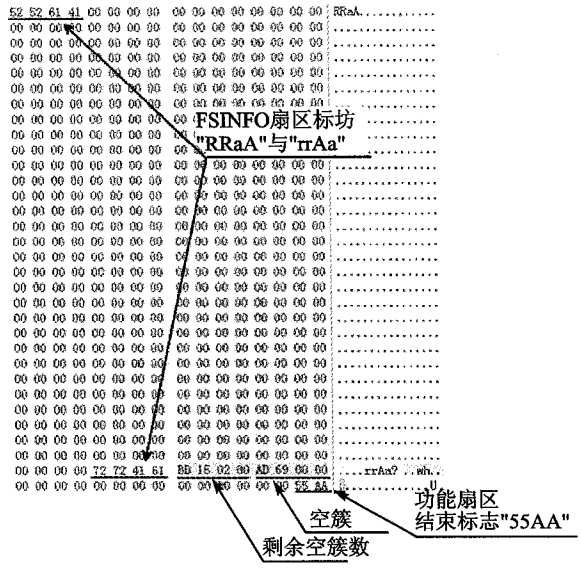

图 2.11　FSINFO 扇区中的数据

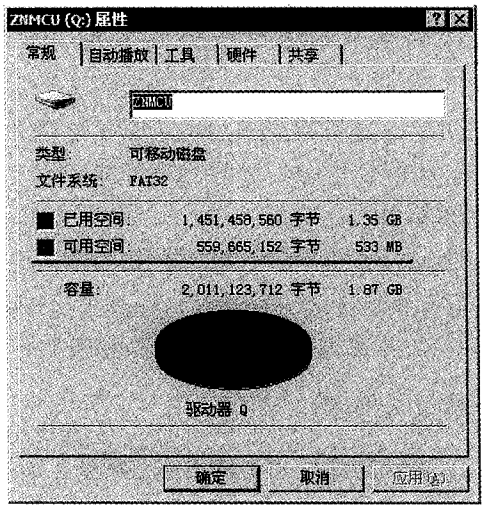

图 2.12　Windows 中的磁盘属性对话框

可以看到，计算结果与 Windows 中显示的剩余存储空间是完全吻合的。这样一个计算磁盘剩余存储空间的操作，在振南还不知道 FAT32 中存在 FSINFO 扇区之前，一直认为是很耗费时间的。振南认为要遍历整个 FAT 表、统计空簇的数量，然后才能计算得到磁盘剩余存储空间。

FSINFO 扇区其实是 FAT32 文件系统独有的一个功能部分，在 FAT 文件系统的其他早期版本中是没有的，比如 FAT12、FAT16。在这些文件系统中要完成计算

磁盘剩余空间的操作，确实要通过遍历FAT表来实现。但对于FAT32来说，这只是“转瞬之间”的事，这就得益于FSINFO扇区。当时振南认为：既然FAT32中有FSINFO扇区，那就直接读取其中记录的空簇即可。但后来一个“怪异”的现象显示FSINFO扇区中记录的空簇根本就是不可信的，如图2.13所示。

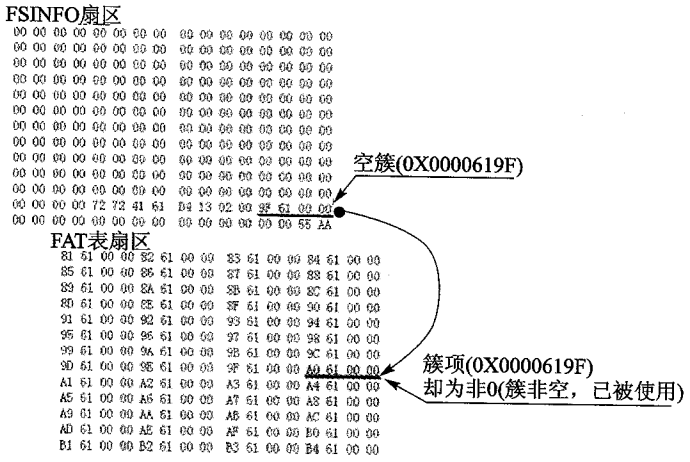

图2.13　FSINFO记录的“空簇”实际“非空”

图2.13中FSINFO扇区记录的空簇为0X0000619F，按理说对应的FAT簇项应该为0，但实则不然。如果直接拿来用，则必然会造成对现有数据的破坏，这到底是怎么回事呢？难道FSINFO扇区中记录的空簇是形同虚设的？后来，网上的一句话印证了这一猜测：“FAT32中虽然在FSINFO扇区中记录了空簇，但并不保证它一直是正确的！”

实际上，Windows等操作系统上的FAT32根本就不依赖于FSINFO扇区。在空簇的查找方面，它们有自己的一套算法。这种算法又有别于振南这里使用的“从头遍历，接力查找”的方法。

最终，振南放弃了使用FSINFO扇区来查找空簇的方案，znFAT还是无法逃脱在初始化中对FAT表从头遍历的“噩运”。但FSINFO扇区也并非一无是处，因为其中所记录的剩余空簇数是可以保证正确的。znFAT对于FSINFO扇区全力维护，尽量保证其参数的正确性。当有空簇被使用，或是有簇被回收的时候，初始化参数集合中的Free_Cluster与Free_nCluster(用于记录剩余空簇数，其初值在znFAT_Init中从FSINFO扇区中读回)都会随之修改，并同步更新FSINFO扇区中的参数字段。

至于具体如何实现对FSINFO扇区的更新，我们通过下面这个函数来完成(znFAT.h)：

```
struct FSInfo   //znFAT 中对 FSINFO 扇区结构的定义
{
 UINT8 Head[4];  //"RRaA"
```

```
 UINT8 Resv1[480];
 UINT8 Sign[4];  //"rrAa"
 UINT8 Free_Cluster[4];  //剩余空簇数
 UINT8 Next_Free_Cluster[4];  //空簇(实际无意义)
 UINT8 Resv2[14];
 UINT8 Tail[2]; //"55 AA"
};
```

znFAT.c 代码如下：

```
UINT8 Update_FSINFO()  //更新 FSINFO 扇区
{
 struct FSInfo * pfsinfo;
 znFAT_Device_Read_Sector(Init_Args.DBR_Sector_No + 1,znFAT_Buffer);
 pfsinfo = ((struct FSInfo * )znFAT_Buffer);
 //写入剩余空簇数
 pfsinfo->Free_Cluster[0] = Init_Args.Free_nCluster;
 pfsinfo->Free_Cluster[1] = Init_Args.Free_nCluster>>8;
 pfsinfo->Free_Cluster[2] = Init_Args.Free_nCluster>>16;
 pfsinfo->Free_Cluster[3] = Init_Args.Free_nCluster>>24;
 //Free_Cluster 更新无意义
 znFAT_Device_Write_Sector(Init_Args.DBR_Sector_No + 1,znFAT_Buffer);
 return 0;
}
```

2.2.4 簇链构造的实现

簇链构造的具体实现主要分 3 步：把原簇链“打开”，把空簇“放进去”，把簇链“关上”，实例请看图 2.14。

簇链的构造必然涉及对 FAT 表项的修改，我们通过函数 Modify_FAT 来完成这项操作，具体实现如下(znFAT.c)：

```
UINT8 Modify_FAT(UINT32 cluster,UINT32 next_cluster)
{
 UINT32 temp1 = 0,temp2 = 0;
 if(0 == cluster || 1 == cluster) return 1; //簇项 0 与 1 是不能修改的
 temp1 = Init_Args.FirstFATSector + (cluster * 4/Init_Args.BytesPerSector);
                                                    //计算簇项所在的 FAT 扇区
 temp2 = ((cluster * 4) % Init_Args.BytesPerSector);    //计算簇项在 FAT 扇区中的位置
 znFAT_Device_Read_Sector(temp1,znFAT_Buffer);          //读取 FAT1 扇区
 znFAT_Buffer[temp2 + 0] = next_cluster;                //修改簇项的值
 znFAT_Buffer[temp2 + 1] = next_cluster>>8;
 znFAT_Buffer[temp2 + 2] = next_cluster>>16;
 znFAT_Buffer[temp2 + 3] = next_cluster>>24;
```

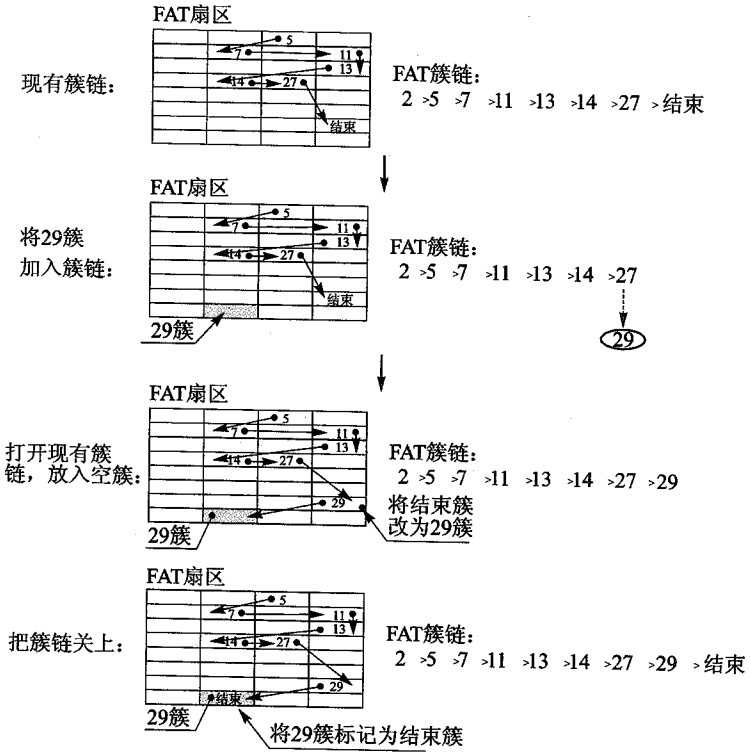

图 2.14　簇链构造的实现过程

```
    znFAT_Device_Write_Sector(temp1,znFAT_Buffer); //回写
    znFAT_Device_Read_Sector(temp1 + Init_Args.FATsectors,znFAT_Buffer); //读取 FAT2 扇区
    znFAT_Buffer[temp2 + 0] = next_cluster;  //修改簇项的值
    znFAT_Buffer[temp2 + 1] = next_cluster>>8;
    znFAT_Buffer[temp2 + 2] = next_cluster>>16;
    znFAT_Buffer[temp2 + 3] = next_cluster>>24;
    znFAT_Device_Write_Sector(temp1 + Init_Args.FATsectors,znFAT_Buffer);  //回写
    return 0;
}
```

我们知道 FAT32 是有两个 FAT 表的，所以上面的程序对 FAT1 与 FAT2 进行了同步修改。其实 FAT2 是 FAT1 的一个备份，通常来说它们的内容是完全相同的，这也为数据恢复提供了一个重要依据。

接下来将簇链构造加入到 Settle_FDI 函数之中，代码如下(znFAT.c)：

```
UINT8 Settle_FDI(UINT32 dirclust,struct FDI * pfdi,UINT32 * psec,UINT8 * pn)
{
 //其他变量定义
 UINT32 old_clu = 0; //用于记录上一簇
```

```
    do
    {
     //检查是否已有同名的文件目录项存在,搜索空位
     old_clu = Cur_Clust;
     Cur_Clust = znFAT_GetNextCluter(Cur_Clust); //获取下一簇
    }while(!IS_END_CLU(Cur_Clust)); //如果不是最后一个簇,则继续循环
    //如果运行到这里,则说明当然簇中无空位
    if(0! = Init_Args.Free_nCluster) //如果剩余空簇数不为 0,说明磁盘还有空间
    {
     Modify_FAT(old_clu,Init_Args.Free_Cluster); //将空簇接到原簇链上
     Modify_FAT(Init_Args.Free_Cluster,0X0FFFFFFF); //构造 FAT 簇链
     Clear_Cluster(Init_Args.Free_Cluster); //清空空闲簇
     (*psec) = SOC(Init_Args.Free_Cluster); //记录空位的位置
     (*pn) = 0;
     znFAT_Device_Read_Sector((*psec),znFAT_Buffer);
     Memory_Copy((UINT8*)((((struct FDIesInSEC *)znFAT_Buffer)->FDIes)), (UINT8
*)pfdi,32);
     znFAT_Device_Write_Sector((*psec),znFAT_Buffer);
     Update_Free_Cluster(); //更新空簇
     return 0;
    }
    else //磁盘已无空间
    {
     return ERR_NO_SPACE;
    }
   }
```

2.3 目录的创建

文件的创建已经大体完成,接下来自然会想到目录的创建,其实它们在实现上是基本相同的,这里仅针对一些差异来简单介绍。

2.3.1 目录项的构造

为了有别于前面所说的文件目录项,这里把目录的文件目录项暂称为目录项。目录项的构造仍然使用 Fill_FDI 函数来实现,不过要进行一些改进,代码(znFAT.c)如下:

```
UINT8 Fill_FDI(struct FDI *pfdi,INT8 *pfn,struct DateTime *pdt,UINT8 is_file)
{
 //文件目录项字段填充
```

```
pfdi->Attributes=(is_file? 0X20:0X30); //设置属性
//目录与文件不同,目录在创建之初就要为其分配空簇
//为的是写入.与..这两个特殊的目录项
if(!is_file) //如果是目录,则填充空簇
{
 pfdi->HighClust[0]=(Init_Args.Free_Cluster)>>16;
 pfdi->HighClust[1]=(Init_Args.Free_Cluster)>>24;
 pfdi->LowClust [0]=(Init_Args.Free_Cluster);
 pfdi->LowClust [1]=(Init_Args.Free_Cluster)>>8;
}
return 0;
}
```

其中,函数的形参中引入了is_file,用于区别文件与目录。除了设置不同的属性值以外,还有一个为新建的目录分配空簇的过程。对于文件而言,因为在创建之初数据为空,所以它并不占用空簇(开始簇为0)。那么目录为什么要事先分配一个空簇呢(其实它就是目录的开始簇)?我们要向这个空簇中写入什么呢?请往下看。

2.3.2 两个特殊的目录项

对DOS(或Windows命令行)比较熟悉的读者一定知道这样一个命令"CD..",功能就是返回上一级目录。CD是用来进入某一个目录的,后面一般跟的是目录名或路径。那".."难道也是目录名?是的,".."真的是一个目录名,其目录项真实地记录在目录开始簇中。这个目录簇中记录了上一级目录的开始簇(使得目录结构可回溯)。基本上每一个目录中都会有".."这个子目录(除了首目录以外);另外还有一个"."目录,记录了当前目录的开始簇。

我们可以使用DIR命令来查看当前目录下的所有子目录及文件,其中就有"."与"..",如图2.15所示。这两个文件目录项到底是怎么样的呢?用WinHex来看一下,如图2.16所示。

```
C:\Debug>dir
 驱动器 C 中的卷没有标签。
 卷的序列号是 0C5F-14CF

 C:\Debug 的目录

2012-01-07  23:57    <DIR>          .
2012-01-07  23:57    <DIR>          ..
2012-01-08  15:27            33,792 vc60.idb
2012-01-08  15:27            53,248 vc60.pdb
2012-01-07  23:57           213,628 a.pch
2012-01-08  15:27           196,468 a.ilk
2012-01-08  15:27           155,679 a.exe
2012-01-08  15:27           402,432 a.pdb
2012-01-08  15:27             3,239 a.obj
2012-03-31  22:03    <DIR>          abc
               7 个文件      1,058,486 字节
               3 个目录     49,561,600 可用字节
```

图2.15 使用DIR命令看到的目录下所有内容

```
2E 20 20 20 20 20 20 20  20 20 20 10 00 45 30 BF | .           ..E0 f
27 40 27 40 1B 00 31 BF  27 40 61 AD 00 00 00 00 | '@'@..1?@a?.....
2E 2E 20 20 20 20 20 20  20 20 20 10 00 45 30 BF | ..          ..E0
27 40 27 40 00 00 31 BF  27 40 00 00 00 00 00 00 | '@'@..1?@.......
56 43 36 30 20 20 20 20  49 44 42 20 18 A9 31 BF | VC60    IDB .?
27 40 64 40 1B 00 70 7B  28 40 85 AD 00 84 00 00 | '@d@..p{(@艮.?.
56 43 36 30 20 20 20 20  50 44 42 20 18 5C 32 BF | VC60    PDB .\2 
27 40 64 40 1B 00 70 7B  28 40 4C B0 00 D0 00 00 | '@d@..p{(@L??.a.
41 20 20 20 20 20 20 20  50 43 48 20 18 4B 32 BF | A       PCH .K2
27 40 64 40 1B 00 33 BF  27 40 4A B0 7C 42 03 00 | '@d@..3?@J晴B...
```

图 2.16　目录开始簇中的.与..目录项

这两个特殊的目录项总是位于目录开始簇的最前面，于是就知道为什么要在目录创建之初就给他分配空簇了。至于目录项的“落定”，我们仍然使用函数 Settle_FDI，代码如下(znFAT.c)：

```
UINT8 Create_Dir_In_Cluster(UINT32 * cluster,INT8 * pdn,struct DateTime * pdt)
{
 //变量定义
 UINT32 dummy = 0;
 struct FDI fdi;
 //目录名合法性检验
 Fill_FDI(&fdi,pdn,pdt,0); //构造目录项
 Settle_FDI( * cluster,&fdi,&dummy,&pos);//在当前簇中进行目录项的“落定”
 //向目录簇中写入.与..
 Modify_FAT(Init_Args.Free_Cluster,0X0FFFFFFF); //构造 FAT 簇链
 Clear_Cluster(Init_Args.Free_Cluster); //清空空簇
 //把 fdi 中的名字替换为. 名为.的目录项记录了当前目录开始簇
 fdi.Name[0] = '.';
 for(i = 1;i<11;i++ ) fdi.Name[i] = ' ';
 Memory_Copy(znFAT_Buffer,((UINT8 * )(&fdi)),32); //将目录项.装入到内部缓冲区中
 //把 fdi 中的名字替换为.. 名为..的目录项记录了上一层目录开始簇
 fdi.Name[1] = '.';
 fdi.HighClust[0] = ( * cluster)>>16;
 fdi.HighClust[1] = ( * cluster)>>24;
 fdi.LowClust [0] = ( * cluster);
 fdi.LowClust [1] = ( * cluster)>>8;
 Memory_Copy(znFAT_Buffer + 32,((UINT8 * )(&fdi)),32);
                                                    //将目录项..装入到内部缓冲区中
 znFAT_Device_Write_Sector(SOC(Init_Args.Free_Cluster),znFAT_Buffer); //回读扇区
 ( * cluster) = (Init_Args.Free_Cluster); //通过形参返回新创建的目录开始簇
 Update_Free_Cluster(); //更新空簇
 return 0;
}
```

这个函数可以在指定的目录中创建一个子目录。* cluster 是目录的开始簇，pdn 是子目录名，pdt 指向时间信息。最后，它将新创建的子目录的开始簇再记录到 * cluster 中。这当然只是一个中间函数，最终的目录创建函数还是要依据目录路径来进行。具体定义如下：

```
UINT8 znFAT_Create_Dir(INT8 * pdp,struct DateTime * pdt)
```

这个函数的具体实现可以参见 znFAT 的源代码，思路如图 2.17 所示。

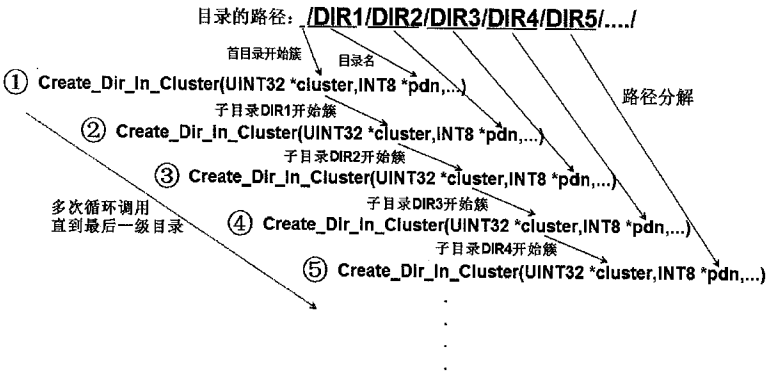

图 2.17　最终目录创建函数的主要实现过程

这里给读者留一个极限实例对文件与目录创建函数进行测试：创建一个 20 级的目录，然后在此目录下创建 1 000 个文件（这样艰巨的测试任务要借助于上册中介绍过的 PC 平台＋RAMDISK 的方案来完成，如果直接在 ZN－X 硬件平台上测试会需要太长的时间）。

感谢对振南及 znFAT 的关注与支持，希望振南在嵌入式 FAT32 文件系统方面的研究对您有所帮助！

更多内容请关注 振南电子网站

www.znmcu.cn