《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4章

巧策良方，数据狂飙：独特算法实现数据高速写入

第3章实现了数据的写入功能，但是最后却暴露出一个很严重的问题——数据的写入效率低下。导致这一问题的症结到底在哪？哪些因素会影响数据的写入效率？如何改善？这就是本章将要考虑的问题。振南独创性地提出了几种巧妙的策略和方案，比如簇链预建、CCCB算法、EXB算法等。它们到底是什么？且听振南细细讲解。

4.1 迫出硬件性能

4.1.1 连续多扇区驱动

我们知道，一个簇是由多个扇区组成的，这些扇区在物理结构上一定是连续的。前面在实现数据写入时是如何来处理这些连续扇区的呢？请看图4.1。

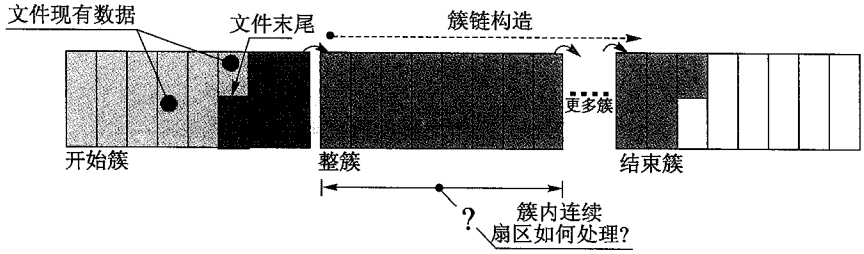


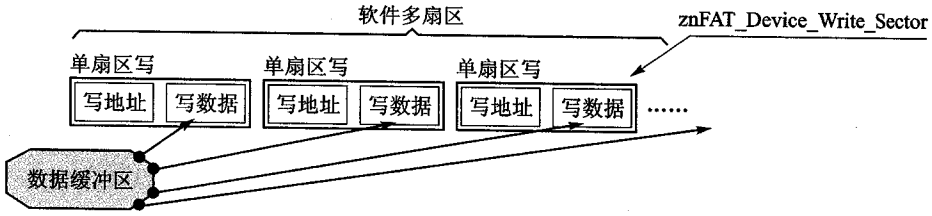
图4.1 数据写入过程中簇内连续扇区的处理

因为现在只有一个物理扇区写函数(znFAT_Device_Write_Sector)，它所实现的是对存储设备单一扇区进行写入操作。所以，对于簇内的连续扇区是这样处理的，代码如下：

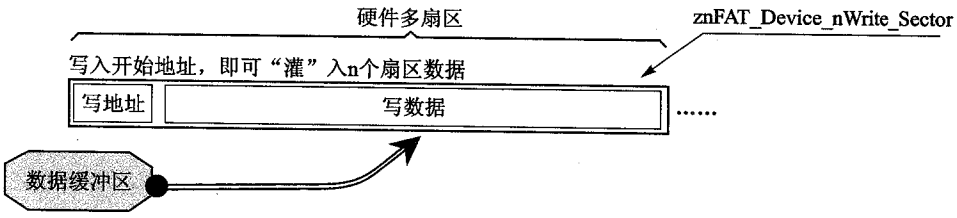
```
for(i = 0;i<(Init_Args.SectorsPerClust);i++) //向簇内连续扇区写入数据
{
 znFAT_Device_Write_Sector(pfi->File_CurSec + i,pbuf);
 pbuf + = 512;
}
```

这种实现方式就是单扇区写＋循环，可以称之为“软件多扇区”。当然，与之相对的就是“硬件多扇区”，这正是振南在这里要引出并着重讲解的。

大多数的存储设备都支持硬件多扇区操作，由硬件完成，因此在性能和速度上都有着软件多扇区无法比拟的绝对优势。图 4.2 体现了软硬两种方式在实现上的差异。可以看到，软件多扇区要多次调用单扇区写函数，而每调用一次都会引发底层驱动对存储设备的一系列操作：写扇区地址、写数据……。读者也许意识到了：“对于一段连续的扇区来说，每次都写入地址似乎有点多余，如果当前地址是 n，那么下一次地址肯定是 n＋1！”确实，所以就有了硬件多扇区。我们首先向存储设备写入开始扇区和要操作的总扇区数，随后就是纯粹的数据写入的过程了，这就注定了硬件多扇区的数据效率是软件多扇区无法比拟的。



(a) 软件多扇区方式



(b) 硬件多扇区方式

图 4.2　多扇区写入操作的软硬两种实现方式的差异

有人还是心存疑虑：“硬件多扇区到底能把数据效率提升多少？”振南就用实例来说明问题：使用软硬两种方式向 SD 卡中的连续扇区写入数据，看看它们分别会花费多少时间。测试代码如下(_main.c)：

```
struct _Time time;
unsigned char xdata buf[4096]; //在外部 RAM 中定义数据缓冲区
void main(void)
{
 unsigned long start_time = 0,end_time = 0;      //记录始末时间
 unsigned int i = 0,j = 0;
 UART_Init();
 UART_Send_Str("串口初始化完成\r\n");
 SD_Init();
```

```
UART_Send_Str("SD卡初始化完成\r\n");
UART_Send_Str("软件多扇区写操作开始\r\n");
P8563_Read_Time(); //读取时间
start_time = (((unsigned long)time.minute) * 60) + (time.second);
for(j = 0;j<100;j++) //以"软件多扇区"方式写 SD卡的 0～7 扇区 100 遍
{
 for(i = 0;i<8;i++) SD_Write_Sector(i,buf + i * 512);
}
P8563_Read_Time();
end_time = (((unsigned long)time.minute) * 60) + (time.second);
UART_Send_Str("软件多扇区写操作结束\r\n");
UART_Put_Inf("使用时间(秒):",end_time - start_time);
UART_Send_Str("硬件多扇区写操作开始\r\n");
P8563_Read_Time();
start_time = (((unsigned long)time.minute) * 60) + (time.second);
for(j = 0;j<100;j++) //以"硬件多扇区"方式写 SD卡的 0～7 扇区 100 遍
{
 SD_Write_nSector(8,0,buf);
}
P8563_Read_Time();
end_time = (((unsigned long)time.minute) * 60) + (time.second);
UART_Send_Str("硬件多扇区写操作结束\r\n");
UART_Put_Inf("使用时间(秒):",end_time - start_time);
while(1);
}
```

这个实验使用 PCF8563 实时钟芯片提供时间信息，通过计算多扇区写操作前后的时间差来获取其花费的时间。另外，为了使测试结果的差异更加明显，这里将多扇区写操作重复了 100 遍。最终的实验结果如图 4.3 所示。很显然，硬件多扇区比软件多扇区在数据效率上要高出一倍还要多。

```
串口初始化完成
SD卡初始化完成
软件多扇区写操作开始
软件多扇区写操作结束
使用时间（秒）：7
硬件多扇区写操作开始
硬件多扇区写操作结束
使用时间（秒）：3
```

图 4.3 软硬两种方式的多扇区写操作效率对比实验结果

4.1.2 多扇区抽象驱动接口

既然硬件多扇区的效率比软件多扇区要高，那我们就为 znFAT 引入多扇区抽象驱动接口，定义如下：

```
UINT8 znFAT_Device_Write_nSector(UINT32 nsec,UINT32 addr,UINT8 * buffer)
```

其中，形参中的 nsec 是要写入的总扇区数，addr 是开始扇区地址，buffer 是指向数据缓冲区的指针。

我们将原来程序中处理簇内连续扇区的代码替换为这个函数，就可以使数据的写入效率得以提升了。当然，前提是开发者必须能够提供多扇区驱动。这也许会造成一个问题：难道没有硬件多扇区驱动，znFAT 就没法使用了吗？这当然不行，所以振南对于多扇区抽象驱动接口的实现做了如下处理：

```
UINT8 znFAT_Device_Write_nSector(UINT32 nsec,UINT32 addr,UINT8 * buffer)
{
 UINT32 i = 0;
 if(0 == nsec) return 0; //如果要写的扇区数 0,则直接返回
 #ifndef USE_MULTISEC_W //此宏决定了是否使用硬件多扇区写入函数
 for(i = 0;i<nsec;i ++ ) //软件多扇区
 {
  SD_Write_Sector(addr + i,buffer); //单扇区写
  buffer + = 512;
 }
 #else
 SD_Write_nSector(nsec,addr,buffer); //硬件多扇区
 #endif
 return 0;
}
```

可以看到，代码中使用编译宏控制来选择使用哪种多扇区驱动的实现方式。像这种编译宏控制我们在后面将会看到更多，它可以控制代码选择性地编译，从而实现对 znFAT 功能的裁减和工作模式的切换与配置。

其实，硬件多扇区同样可以应用于数据读取，对于连续扇区的读取操作可以替换为 znFAT_Device_Read_nSector 来进行实现。

4.2 为数据作“巢”

使用了硬件多扇区之后，振南一度认为 znFAT 的数据写入效率已经够高了，但是后来发现并不是这样。在将 znFAT 与国际上现有的优秀方案对比之后发现，比如 FATFS、EFSL、ucFS 等，事实告诉我们，znFAT 与它们仍然有着较大的差距。深思之后，振南最终提出了一些算法，使得 znFAT 的效率得到了进一步的提升。到底是怎样的算法呢？下面就一一向读者介绍。

4.2.1 预建簇链思想的提出

振南一直相信，凡事只要多加思考，必定会有巧方可用或捷径可走。那我们就来想想，向文件中写入数据有什么更好更快的方法？现在 znFAT_WriteData 函数的实现策略是怎样的？简言之就是不停地写数据、构造簇链、写数据、构造簇链……如此

往复，最后更新文件大小与 FSINFO 扇区，如图 4.4 所示。

在这个过程中，数据写入与簇链构造是同步交替进行的，其实这就是造成数据写入效率不高的根源。因为这已经不是单纯的数据写入了，而是伴随着较为频繁的 FAT 扇区读/写操作。可以这样比喻：一辆好车本可以风驰电掣，但车手却偏偏要每行驶一会儿就停下来，检查检查车子、瞭望前方，这就导致这辆好车不能一往无前的“飙”，而总要牵绊太多，如图 4.5 所示。

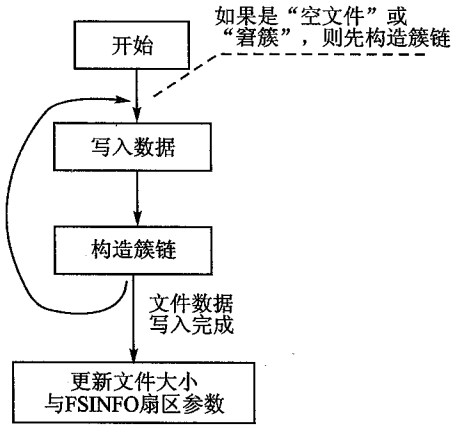
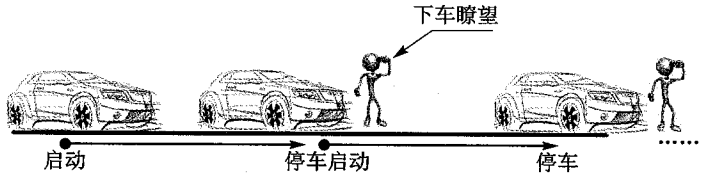


图 4.4　数据写入的大体过程

看来现在的这种实现方式只适用于数据存储速度不高的应用场合，那又有什么更好的实现方式呢？这就要说到“预建簇链”了。顾名思义，它就是在写数据之前，先把整条簇链一次性构建好，随后只管写数据就可以了。实际过程如图 4.6 所示。



注：行驶中“下车瞭望”，犹如数据写入过程中的 FAT 操作

图 4.5　数据写入过程中的 FAT 操作犹如行车时停

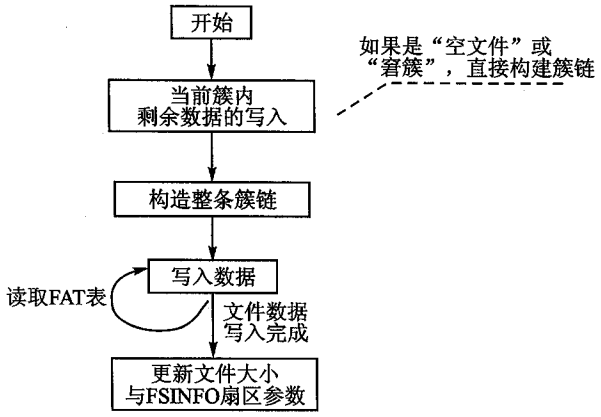


图 4.6　使用“预建簇链”方式实现数据写入的流程

其实预建簇链的策略和思想在前文中就有所应用。回想一下第 1 章：我们首先创建了一个大文件，然后“移花接木”，将数据写入其中。其实创建大文件的过程实质上就是在预先构造簇链。这使得我们在数据写入的过程中可以对 FAT 表“撒手不管”，而只管顺序地向扇区中写数据即可。（其实是应该按照簇链来向簇中写入数据的，只不过因为我们知道大文件的簇链是连续的，所以才省去了读取 FAT 表的步骤。）更形象的描述请看图 4.7。

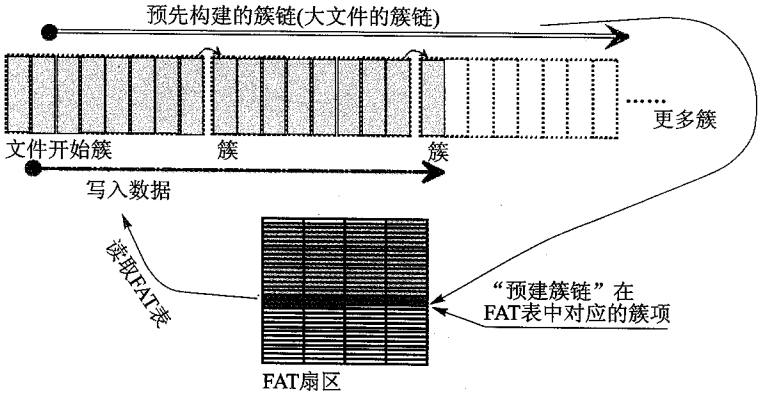


图 4.7　以预建簇链方式写入数据

再举一个例子来说明。在使用迅雷或其他下载软件下载文件的时候，你是否发现它们在下载之初就已经建好了一个临时文件呢？这个文件的体积恰好就是要下载的文件的大小。这些下载软件其实就在使用预建簇链的策略，原因主要有两点：① 提高数据的写入速度；② 尽量保证数据的连续性（一个文件可能会分很多次进行下载，比如断点续传，如果不是预先把文件创建好，那么最终将导致它的簇链支离破碎）。

预建簇链的目的是让数据在这条建好的簇链上“肆无忌惮”地“狂奔”。当然，只有在一次性写入数据量较多的时候，才更能够体现它的优势。

4.2.2　簇链预建的实现

前面我们在构造簇链的时候每次都只给它扩展一个簇，现在要实现整条簇链的构建，就要一次性扩展多个簇，这该如何编程来实现呢？其实很简单，代码如下：

```
for(i = 0;i<n;i++)
{
 Modify_FAT(cur_cluster,Init_Args.Free_Cluster); //将当前簇链到下一空簇
 cur_cluster = Init_Args.Free_Cluster;
 Update_Free_Cluster(); //更新空簇
}
Modify_FAT(cur_cluster,0X0FFFFFFF); //把簇链“关上”
```

振南起初就是这样做的，但后来发现它创建簇链的效率并不高，尤其是簇链比较长的时候，简直让人难以忍受，这主要是因为 Modify_FAT 函数对 FAT 扇区的频繁读/写。用这种方式来实现簇链的创建实际上跟老方法相比是“换汤不换药”。那有什么办法可以快速地构建簇链呢？答：尽量减少对 FAT 扇区的读/写次数。

我们仔细想想上面的这种实现方法，其实它做了很多的“无用功”：每调用一次 Modify_FAT 都会引发对 FAT 扇区的读/写，但实际上根本无需如此。因为要修改的簇项很多情况下都位于同一个 FAT 扇区中，我们可以一次性全部修改好，然后再一起回写到 FAT 扇区中去。实例如图 4.8 所示。

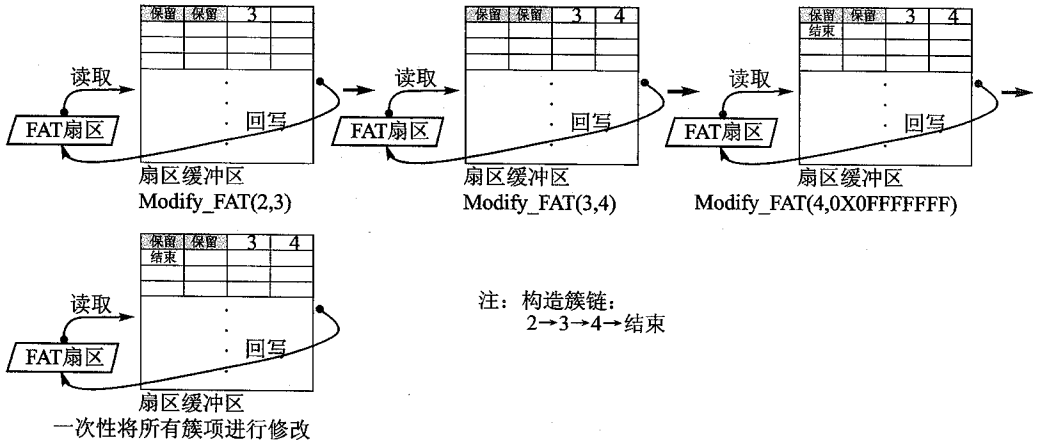


图 4.8　两种簇链构建方法的对比

接下来就来实现预建簇链函数(Create_Cluster_Chain)，其功能就是以 cluster 簇为起点，为后续将要写入的长度为 len 的数据预先构建簇链。也就是说，要在现有簇链的基础上继续扩展(len+CluSize−1)/CluSize 个簇，如图 4.9 所示。具体的实现代码如下(完整代码请参见 znFAT 源代码)(ZnFAT.c)：

```
UINT8 Create_Cluster_Chain(UINT32 cluster,UINT32 len)
{
 UINT32 iSec = 0,clu_sec = 0,old_clu = 0,ncluster = 0;
 UINT8 iItem = 0,temp = 0;
 struct FAT_Sec * pFAT_Sec = (struct FAT_Sec * )znFAT_Buffer;//FAT 扇区结构指针
 UINT32 Clu_Size = (Init_Args.SectorsPerClust * Init_Args.BytesPerSector);
                                                          //计算簇大小
 ncluster = (len + Clu_Size - 1)/Clu_Size; //计算预建簇链包含的簇数
 Init_Args.Free_nCluster - = ncluster; //更新剩余空簇数
 //以下为图 4.9 中的①
 if(0! = cluster) //簇链构建的开始簇不为 0
 {
  clu_sec = cluster/128;
```

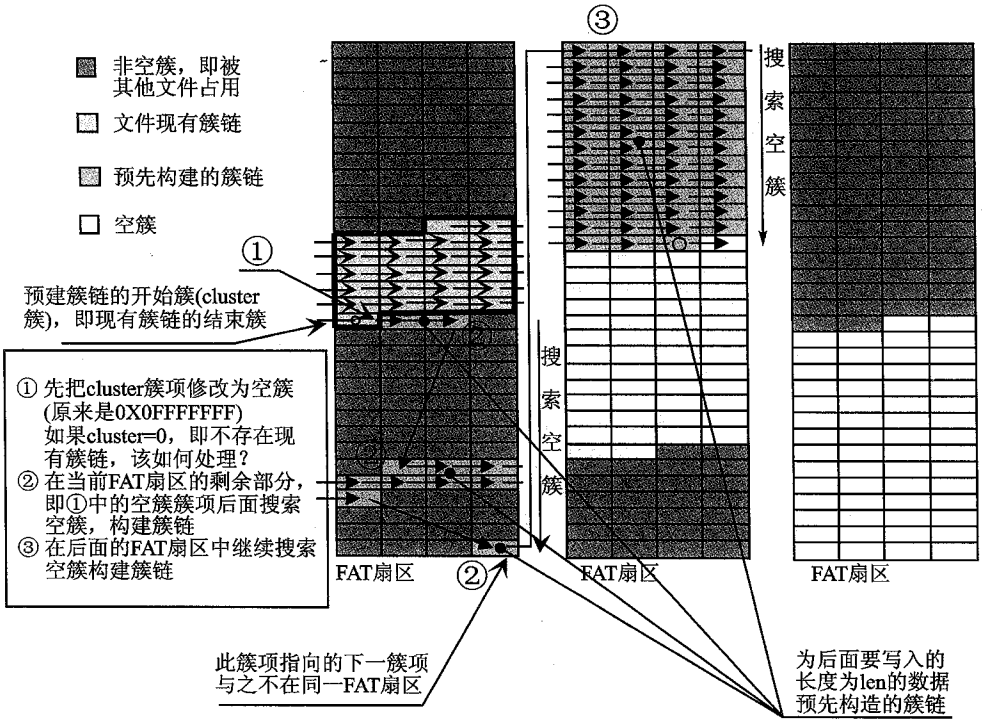


图 4.9　为后面要写入的数据预先构建簇链

```
znFAT_Device_Read_Sector(clu_sec + (Init_Args.FirstFATSector),
                                    znFAT_Buffer); //读取 cluster 簇项所在的 FAT 扇区
temp = cluster % 128; //计算 cluster 簇项在 FAT 扇区中的位置
 //将空簇链到 cluster 簇上
(((pFAT_Sec->items)[temp]).Item)[0] = (Init_Args.Free_Cluster);
(((pFAT_Sec->items)[temp]).Item)[1] = (Init_Args.Free_Cluster)>>8 ;
(((pFAT_Sec->items)[temp]).Item)[2] = (Init_Args.Free_Cluster)>>16;
(((pFAT_Sec->items)[temp]).Item)[3] = (Init_Args.Free_Cluster)>>24;
}
else //为 0,通常说明要给空文件构建簇链
{
 clu_sec = (Init_Args.Free_Cluster/128); //计算空簇项所在的 FAT 扇区
 znFAT_Device_Read_Sector(clu_sec + (Init_Args.FirstFATSector), znFAT_Buffer);
}
ncluster--; //已经为现有簇链扩展了一个簇,预建簇链的簇数减 1
if(0 == ncluster) //如果簇链已构建完成
{
 if(clu_sec == (Init_Args.Free_Cluster/128))
                                    //如果开始簇与它的下一簇的簇项在同一 FAT 扇区
 {
```

```
  //把簇链"关上"
  //FAT 扇区回写:FAT1 与 FAT2
  znFAT_Device_Write_Sector(clu_sec + (Init_Args.FirstFATSector), znFAT_Buffer);
  znFAT_Device_Write_Sector(clu_sec + (Init_Args.FirstFATSector
                                    + Init_Args.FATsectors),znFAT_Buffer);
 }
 else //不在同一 FAT 扇区
 {
  //先将当前 FAT 扇区回写
  clu_sec = (Init_Args.Free_Cluster/128); //计算空簇簇项所在的 FAT 扇区
  znFAT_Device_Read_Sector(clu_sec + (Init_Args.FirstFATSector), znFAT_Buffer);
  //把簇链"关上"
  //FAT 扇区回写
 }
 //更新空簇、更新 FSINFO
 return 0;
}
cluster = Init_Args.Free_Cluster;
old_clu = cluster;
clu_sec = (old_clu/128);
//以下为图 4.9 中的②
if((cluster % 128) + 1) != 128)//如果当前簇项不是其所在 FAT 扇区中的最后一个簇项
                    //也就是说要在当前 FAT 扇区中对剩余部分进行搜索,构建簇链
{
 znFAT_Device_Read_Sector(clu_sec + (Init_Args.FirstFATSector), znFAT_Buffer);
 for(iItem = ((cluster % 128) + 1);iItem<128;iItem ++ )//检测当前 FAT 扇区剩余部分
 {
  cluster ++ ; //簇号自增
  if(簇项为 0) //如果发现空簇
  {
   //将其链在前面的簇项上
   ncluster - - ;
   old_clu = cluster;
  }
  if(0 == ncluster) //如果簇链构建完成
  {
   //把 FAT 簇链"关上"
   //FAT 扇区回写
   Init_Args.Free_Cluster = cluster;
         //更新空簇、更新 FSINFO
   return 0;
  }
```

```
 }
}
//以下是图 4.9 中的③
for(iSec = (clu_sec + 1);iSec<(Init_Args.FATsectors);iSec ++ )
                                                    //在后面的 FAT 扇区中继续查找
{
 znFAT_Device_Read_Sector(iSec + (Init_Args.FirstFATSector),znFAT_Buffer);
 for(iItem = 0;iItem<128;iItem ++ ) //检测当前 FAT 扇区中的空簇
 {
  cluster ++ ;
  if(簇项为 0) //发现空簇
  {
   clu_sec = (old_clu/128);
   temp = (old_clu % 128);
   if(iSec! = clu_sec)//如果要更新的簇项所在 FAT 扇区与当前 FAT 扇区非同一扇区
   {
    znFAT_Device_Read_Sector(clu_sec + (Init_Args.FirstFATSector), znFAT_Buffer);
    //将其链在前面的簇项上
    //FAT 扇区回写
    znFAT_Device_Read_Sector(iSec + (Init_Args.FirstFATSector), znFAT_Buffer);
   }
   else //是同一扇区,则只需要在缓冲区中进行更新
   {
    //将空簇链在前面的簇上
   }
   ncluster - - ;
   old_clu = cluster;
  }
  if(0 == ncluster)
  {
   //把 FAT 簇链"关上"
   //FAT 扇区回写
   Init_Args.Free_Cluster = cluster;
   //更新空簇、更新 FSINFO
   return 0;
  }
 }
 //FAT 扇区回写
}
return 1;
}
```

好，有了预建簇链函数，接下来就可以完成对数据写入的改进了，如图 4.10 所示。篇幅限制，改进后的数据写入函数的具体代码就不再贴出了，读者可以参见 znFAT 源代码。

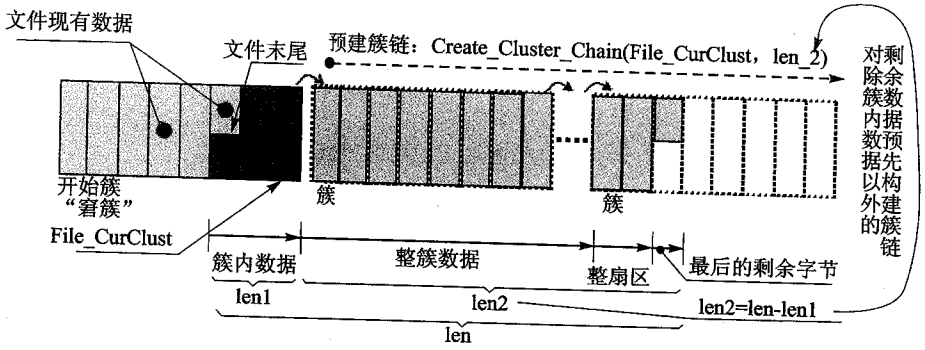


图 4.10　加入预建簇链后的数据写入示意图

4.2.3　将多扇区用到极致

硬件多扇区比软件多扇区效率要高得多，并且还把簇内连续扇区的写入操作替换为硬件多扇区接口函数(znFAT_Device_Write_nSector)，从而提高了数据写入的效率。现在我们已经实现了预建簇链，那回过头来想一想：它们两者合力，是否有把数据写入效率进一步提升的可能呢？答案是肯定的，硬件多扇区将因为簇链预建而使其优势发挥到极致。到底是怎么回事？下面振南就细细道来。

仅将硬件多扇区应用于簇内连续扇区上，主要是因为每次我们只为簇链扩展一个簇，这使得我们的目光只会放在这一个簇上，只能看到簇内扇区的“小连续”，如图 4.11 所示。

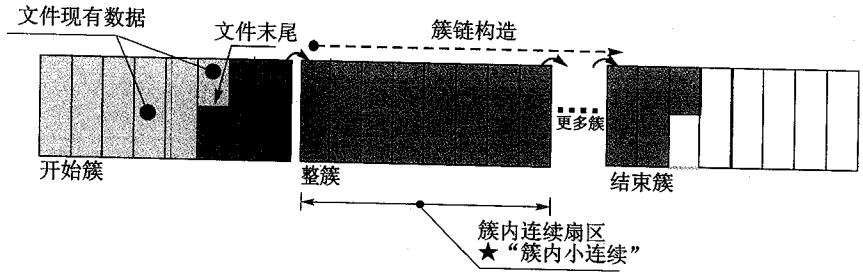


图 4.11　簇内扇区的“小连续”

但是预建簇链函数(Create_Cluster_Chain)可以一次性构建整条簇链，新构建的簇链上自然包含了多个簇。如果这些簇之间是连续的，那将看到一大段的连续扇区，振南称之为“大连续”，如图 4.12 所示。

这种大连续将使硬件多扇区的优势发挥得淋漓尽致。但是此时，可能有人也已

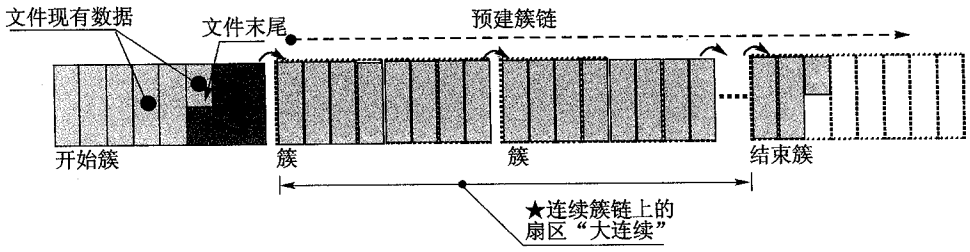


图 4.12　连续簇链上的扇区“大连续”

经看出了问题:“如果簇链不连续该怎么办呢?”,如图 4.13 所示。

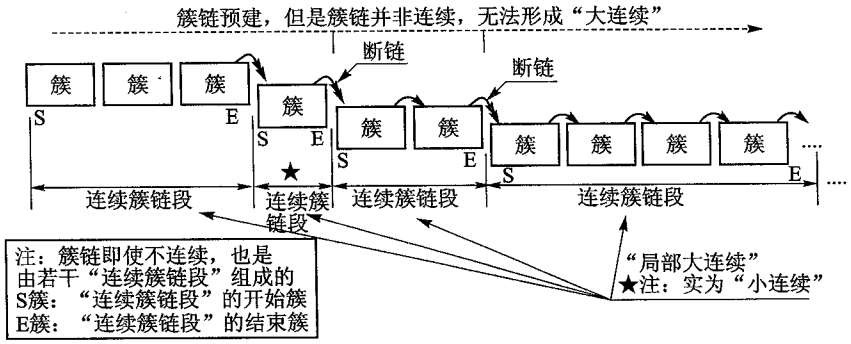


图 4.13　簇链由若干个连续簇链段组成

一个簇链,即便不完全连续,也一定是由若干个连续簇链段组成的。无法实现大连续,但是却可以针对连续簇链段实现局部大连续。当然,如果连续簇链段只包含一个簇,那它其实就是小连续了(如图 4.13 中的★)。

基于连续簇链段思想,我们可以对数据写入函数的实现进行改进,代码如下(znFAT.c):

```
//检测连续簇链段,尽可能使用多扇区驱动,提高数据写入效率
//start_clu 与 end_clu 用于记录连续簇链段的始末,对应于图中的 S 与 E
start_clu = end_clu = 簇链的开始簇;
for(iClu = 1;iClu<簇链包含的总簇数;iClu ++ )
{
 next_clu = Get_Next_Cluster(end_clu); //获取下一簇
 if((next_clu - 1) == end_clu) //如果两个簇连续
 {
  end_clu = next_clu;
 }
 else //如果两个簇不连续,即遇到断链
 {
  znFAT_Device_Write_nSector(((end_clu - start_clu + 1)
```

```
                    * (Init_Args.SectorsPerClust)),SOC(start_clu),pbuf);
                                        //对连续簇链段进行多扇区写操作
    pbuf += ((end_clu - start_clu + 1) * CluSize);
    start_clu = end_clu = next_clu;
  }
}
znFAT_Device_Write_nSector(((end_clu - start_clu + 1)
                    * (Init_Args.SectorsPerClust)),SOC(start_clu),pbuf);
                        //对最后一个连续簇链段进行多扇区写操作
pbuf += ((end_clu - start_clu + 1) * CluSize);
```

到这里，我们基本上已经快把硬件多扇区用到了极致。“基本上快到极致？可我认为我们已经找出了所有连续扇区的可能，难道还有更多的连续扇区可以发掘吗？”是的！试想一下，如果文件结束簇（用于存储最后不足整簇的剩余数据）与最后一个连续簇链段也连续的话，那么结束簇中的整扇区部分就与前面的扇区又构成了“更大的连续”，如图 4.14 所示。

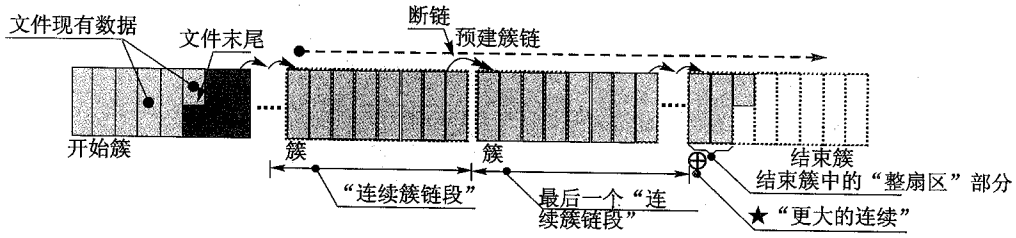


图 4.14　文件结束簇中的整扇区与最后连续簇链段构成的“更大的连续”

4.3　CCCB（压缩簇链缓冲）

4.3.1　CCCB 的提出

前面讲了基于预建簇链的数据写入，并且针对硬件多扇区以及扇区连续性对其进行了优化。但是，有没有考虑过这样一个问题：如果我们向一个文件写入 10 000 次数据，那么整体的数据写入效率将会如何呢？这其实是一个很实际的问题，在很多时候人们都是在周期性或分多次地向文件写入数据，振南称之为“间歇性频繁数据写入”。伴随着这种数据写入方式，将产生一个比较棘手的问题，如图 4.15 所示。

每一次预建簇链都会产生对 FAT 表的更新，那么写 10 000 次数据就会更新 10 000 次。也许你并不觉得这个问题有多么严重，每更新一次 FAT 表，都可以一次性构建很长的一条簇链出来。与大量数据高效率地写入相比，更新 FAT 表所花费的时间似乎不算什么。但是别忘了，只有在每次向文件写入的数据量比较大的时候

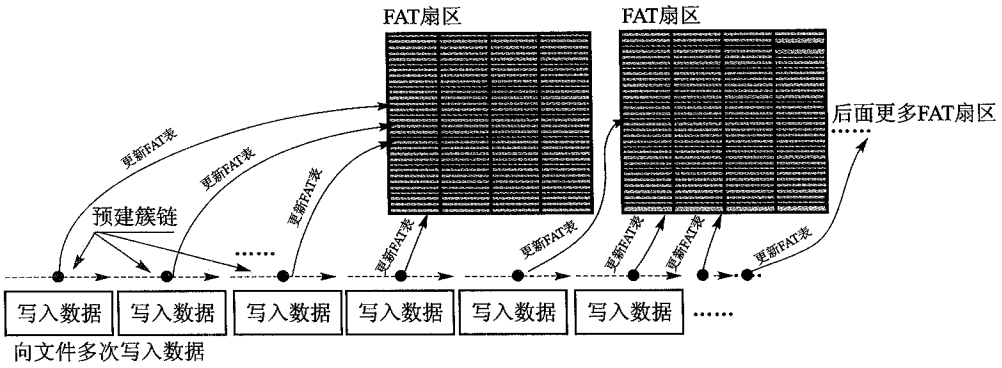


图 4.15　间歇性频繁数据写入产生对 FAT 表的频繁更新

才是这样。如果每次写入的数据量比较小(间歇性频繁小数据量写入),那么大部分的时间岂不是都浪费在更新 FAT 表上了吗?实际的情况其实可能会比这更加糟糕,如图 4.16 所示。

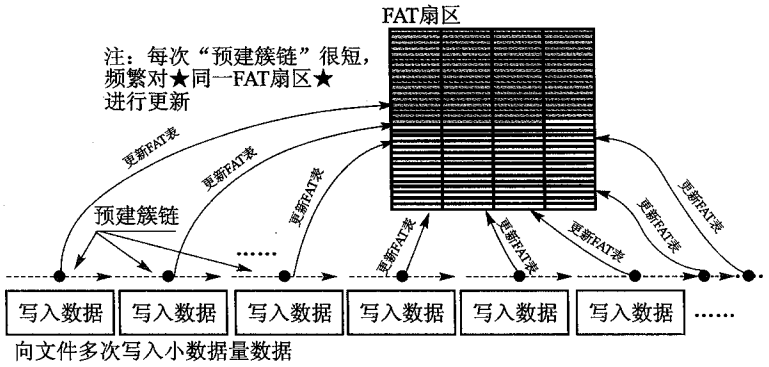


图 4.16　间歇性频繁数据写入产生对 FAT 表的频繁更新

可以看到,因为每次写入的数据量比较小,所以预建簇链也最多只能为现有簇链扩展出一个簇而已,而无法实现较长簇链的构建,这将产生对同一 FAT 扇区的非常频繁的读/写操作。可能读者对存储设备的特性还不太了解:通常如果对同一扇区进行多次读/写,那么就会发现它越来越慢,这主要归咎于存储设备内部控制器的自我保护机制(对同一扇区频繁操作将影响其使用寿命)。所以,在这种情况下,数据的写入效率将会比较低,甚至让人难以忍受。

那又有什么更好的办法呢?毕竟 FAT 表是必须要更新的。“是否可以把簇链暂存在内存中,等数据全部写完之后再一起更新到 FAT 表物理扇区中呢?”确实如此,但是要把文件的整条簇链放入内存又谈何容易?最大的“瓶颈”就是内存容量。要解决这一问题,我们还要从一则笑话说起:燕子和青蛙比赛嘴快,方法是从 1 数到 10,看谁数得快。燕子用极快的速度数完了这 10 个数,用了 2 s;青蛙不屑地看了看它,懒散地说道:“1 到 10”,只用了半秒。这笑话似乎有点冷,但是却为我们提供了一

个思路，如图 4.17 所示。

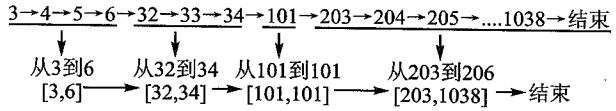


图 4.17　对簇链采用区间式表示

一个簇链是由若干个连续簇链段组成的，那么就可以使用“区间”方式来对簇链进行表达，如图 4.17 所示。每一个区间只记录了连续簇链段的始末，从而大大降低了内存的使用量。比如在图 4.17 中，原本这个簇链包含了 844 个簇，如果用原始方式，那么就需要 844 个存储单元。但如果用区间方式，却只需要 8 个存储单元即可。可见，区间方式所占用的存储单元数只与簇链的连续性有关。在通常情况下，文件的簇链都是比较连续的，一个支离破碎的簇链一般还是比较少见的。所以，无需多少内存即可对簇链进行记录。如果簇链的连续性比较好，那甚至可以仅用两个存储单元即可对文件整条簇链进行记录。这种区间方式很好地解决了簇链记录与内存容量之间的矛盾。其实，这个过程就是在对原始簇链进行压缩，使其占用更少的存储空间。于是，振南就给它取名为“CCC”(Compressed Cluster Chain)，即压缩簇链。在实际的实现过程中，用于存储这些区间的缓冲区就是 CCCB(CCC Buffer)，也就是压缩簇链缓冲。

我们对预建簇链的实现进行改进，如图 4.18 所示。可以看到，构建出来的簇链不再直接更新到 FAT 表的物理扇区中，而是以压缩簇链的方式暂存于 CCCB 缓冲区中。这样就避免了对 FAT 扇区的频繁读/写，从而使数据的写入效率进一步得以提升。这确实是一种非常巧妙的机制或者说策略。每当振南向别人介绍 znFAT 的独特之处时必然少不了 CCCB，“znFAT 可以用区区几个字节的存储空间对整个文件进行缓冲！”闻者无不惊叹。其实这些都要归功于 CCCB。

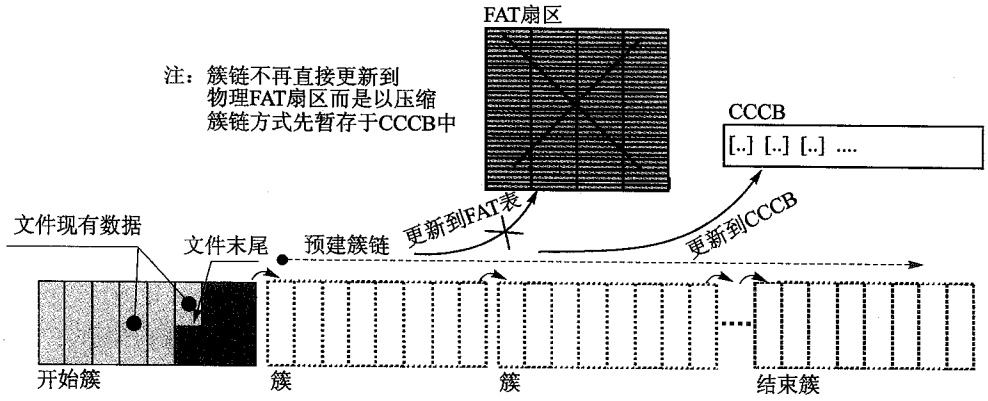


图 4.18　数据写入过程中预建簇链暂存于 CCCB 中

4.3.2 CCCB 的实现

前面这些关于 CCCB 的设计思想其实都还是比较好理解的，但是在具体的实现上也许会有些繁琐，因为它将涉及一些比较细节、比较麻烦的问题。限于篇幅，振南只能进行一个大体的介绍，让读者对其中的内容和问题有一个基本的了解，有兴趣的读者可以去细细研读 znFAT 源代码。

CCCB 的实现主要包含以下 3 个基本的操作：构造、回写与寻簇，如图 4.19 所示。这 3 个基本操作具体是什么意思呢？举个例子：向一个文件中写入数据，数据写入函数(znFAT_WriteData)首先会预建一条簇链。将簇链以压缩簇链方式存入簇链缓冲的过程就是“CCCB 的构造”(即图中的①)；簇链被构建起来之后，我们就要依照这条簇链来向各个簇写入数据了，这就涉及从 CCCB 中获取簇链关系的问题，就像是从 FAT 表中获取下一簇的函数 Get_Next_Cluster 一样，这就是“CCCB 的寻簇”(即图中的③)；CCCB 是位于内存中的，但是它不能一直驻留于内存之中，终归还是要落实于 FAT 表的物理扇区中的，这就是“CCCB 的回写”(即图中的②)。下面就来对这 3 个基本操作的实现方法进行逐一进行介绍。

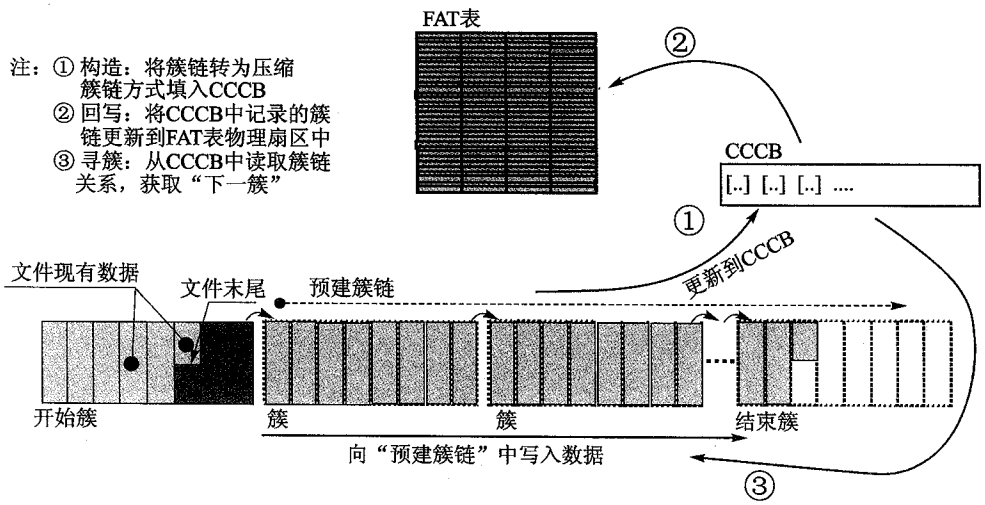


图 4.19 CCCB 在实现过程中所包含的 3 个基本操作

1. CCCB 的定义

在讲这 3 个基本操作之前，我们首先要把 CCCB 建立起来。也就是对 CCCB 的数据结构，比如数组、结构体等，还有相关的一些变量进行定义。具体代码如下(znFAT.c)：

```
#define CCCB_LEN   (8) //压缩簇链缓冲长度，一定是不小于 4 的偶数
UINT32 cccb_buf[CCCB_LEN]; //压缩簇链缓冲
```

```
UINT8 cccb_index;
UINT32 cccb_curclu;
```

这些定义都是全局的。cccb_buf 用于记录压缩簇链的数组；cccb_index 用于记录当前指向的数组元素下标，以方便将连续簇链段的始末填入其中；cccb_curclu 用于记录连续簇链段的当前簇。也许这样说还是有些抽象，我们还是在 3 个基本操作的实现中深入去理解它们的含义吧。

2. CCCB 的构造

CCCB 的构造其实很简单。首先将预建簇链的第一个簇赋给 cccb_curclu（其实它就是 CCCB 中第一个区间的开始簇），以后在簇链构建过程中得到的空簇均与 cccb_curclu 进行比较，看其是否连续。如果是则更新 cccb_curclu 为此空簇，如果不是就对当前区间进行“封口”，并开始新的区间。这一过程如图 4.20 所示。

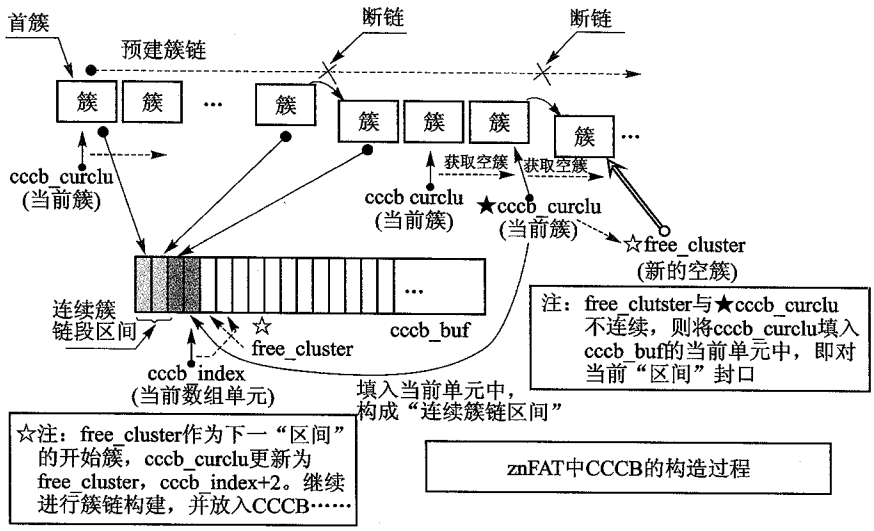


图 4.20　CCCB 构造的具体实现过程

前面振南说过 CCCB 中会有一些比较麻烦的情况，下面要讲到的内容也许就“可见一斑”了。我们想想，如果簇链的连续性确实不太好，有比较多的“断链”，那么就有可能出现 cccb_buf 不够用的情况，从而造成缓冲区的溢出。我们必须对其进行处理，尽量避免溢出错误的发生。那具体该如何处理呢？答：将 CCCB 回写、清空、再利用，如图 4.21 所示。详细代码参见 znFAT 源码中的 Create_Cluster_Chain 函数的具体实现。

3. CCCB 的回写

其实上面的内容就已经涉及 CCCB 的回写了，实际上它就是对压缩簇链的“解压”，将它还原为簇链，再写入到 FAT 扇区中去。这一操作对于数据写入功能是非

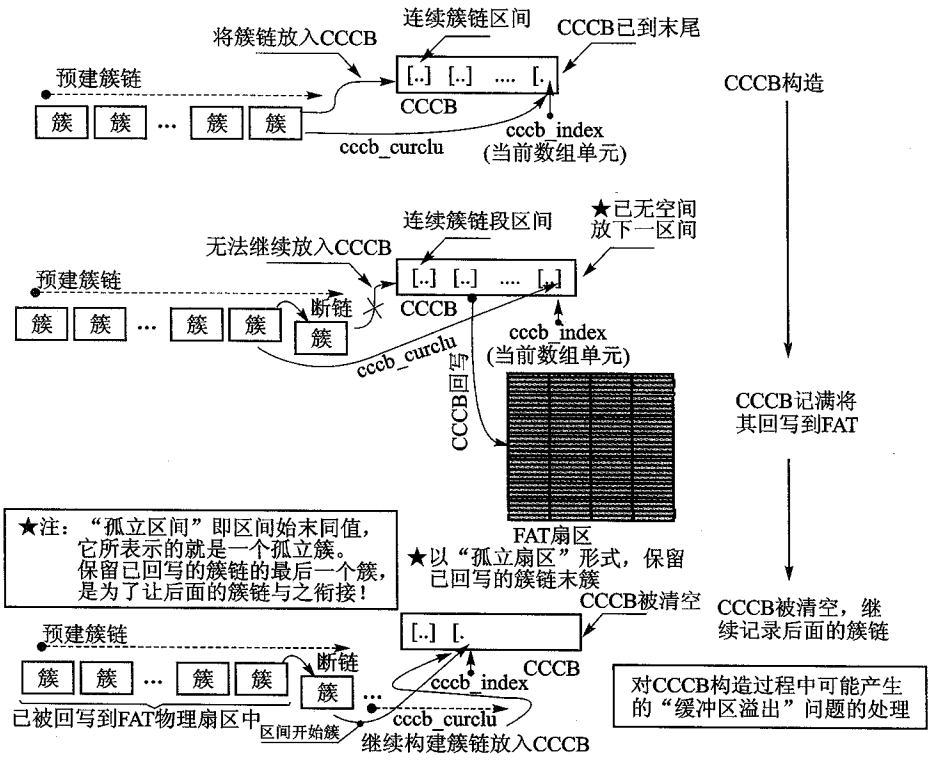


图 4.21　对 CCCB 构造过程中的缓冲区溢出问题进行处理

常重要的。试想，如果向文件写入了数据，但是却没做 CCCB 的回写，那会如何呢？数据将全部丢失！其实 CCCB 回写的具体实现很简单，振南通过下面这个实例来进行讲解：将图 4.17 中的压缩簇链回写到 FAT 中，如图 4.22 所示。znFAT 中使用函数 CCCB_Update_FAT 来完成这一操作。

4. CCCB 的寻簇

CCCB 的寻簇其实很简单，就是在 CCCB 中去查找某个簇的下一簇。但是有一点一定要注意到：在引入 CCCB，尤其是溢出回写机制之后，一条簇链就不光只存在于 CCCB 中了，可能有一部分已经被回写到 FAT 表中了。所以，在寻簇的时候就要二者兼顾，如图 4.23 所示。

实现时，首先在 CCCB 中查找，然后再在 FAT 表的物理扇区中查找。这样做的原因很明显，就是因为前者位于内存中，它的查找效率要比后者高得多。其实，通常情况下文件的连续性都会比较好，断链的数量不会超过 4 个（除非磁盘上的碎片太多），也就是说产生 CCCB 溢出回写的机率比较小，簇链一般全部存在于 CCCB 中。所以，CCCB 对数据写入效率的提高还是会起到很大作用的。

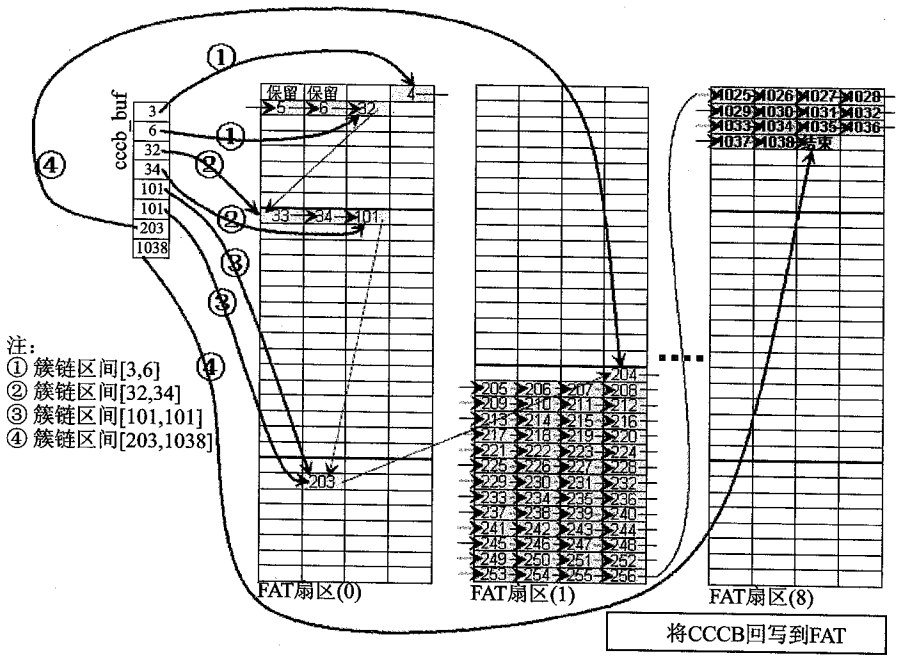


图 4.22　将 CCCB 中的压缩簇链回写到 FAT 表物理扇区

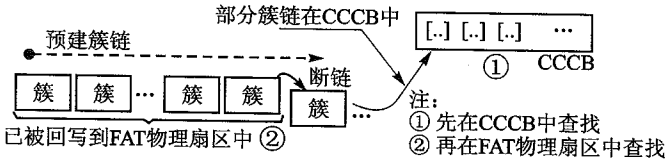


图 4.23　CCCB 寻簇时要二者兼顾

4.3.3　CCCB 的争抢与独立

前面我们所讲的其实只涉及了单个文件的数据写入，此时 CCCB 的相关操作确实还比较简单。但我们要知道，znFAT 是可以支持多文件的，也就是可以同时对多个文件进行操作，这种情况下，CCCB 又会变得如何呢？如图 4.24 所示。

CCCB 缓冲区是以全局变量的形式定义的，这就注定了在某一时刻它只能属于一个文件。如果像图 4.24 这样同时有多个文件都要用到 CCCB，那势必造成 CCCB 的争抢。说白了就是：如果有其他文件也要使用 CCCB，那就先把当前的 CCCB 回写，然后再将 CCCB 进行移交。振南把这个过程形象地称为“CCCB 的轮转”，即各个文件轮流作庄，逐个占用 CCCB，如图 4.25 所示。

其实造成这一问题的根本原因在于整个 znFAT 系统只定义了唯一的全局 CCCB，振南称之为“共享 CCCB”（Shared CCCB，简称 SCCCB）。如果每一个文件都

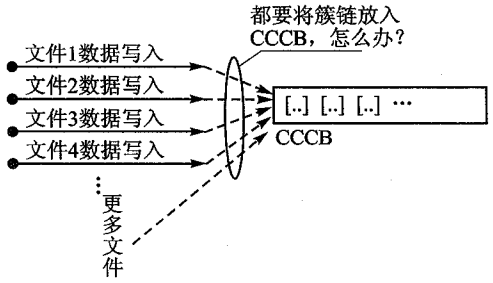


图 4.24　CCCB 在多文件情况下所产生的问题

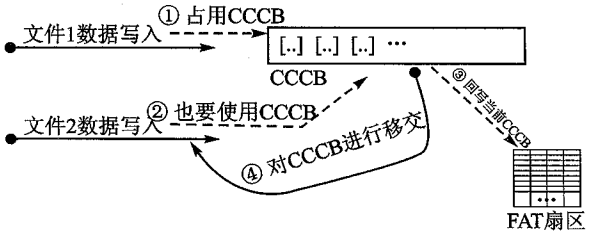


图 4.25　多文件情况下对 CCCB 的轮转

有自己专属的 CCCB，那么就不会再造成争抢的问题了，这就是“独立 CCCB”(Alone CCCB，简称 ACCCB)。CCCB 的争抢与独立的具体实现有点复杂，感兴趣的读者请参见 znFAT 源代码。

4.4　EXB(扇区交换缓冲)

EXB 是振南继 CCCB 之后提出的另一种独特方案，同样也是为了提高数据的写入效率。如果说 CCCB 是专注于簇链的话，那么 EXB 就是针对于数据本身进行的优化。

4.4.1　EXB 的提出

EXB(Sector Exchange Buffer)，即扇区交换缓冲。为了让读者认识到 EXB 所要解决的具体问题，我们还是通过一个实例来说明：向一个文件中写入 10 000 次数据，每次仅写入 10 个字节，数据写入的效率会如何？聪明的读者应该已经意识到了问题的所在：少量数据向同一扇区进行多次写入时，因数据拼接将产生对扇区频繁地“读-改-写”操作，如图 4.26 所示。

其实在多次向文件写入数据时，只要存在最后不足扇区的、若干个字节的剩余数据，那么在下一次数据写入时就必定会出现扇区内的数据拼接。如何解决这一问题呢？其实很简单：定义一个 512 字节的缓冲区，让它作为扇区的“映像”。在出现不足

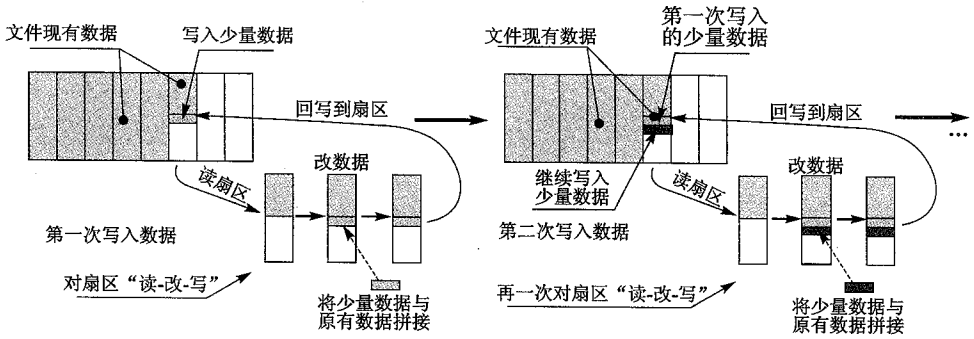


图 4.26　多次小数据量写入时产生扇区的频繁“读-改-写”操作

扇区的数据时，我们不再将它直接写入到扇区，而是先暂时存放在这个缓冲区中。当“攒”够了一个扇区的数据时，再将其一次性写入到扇区之中。这个缓冲区就是振南所说的 EXB，如图 4.27 所示。

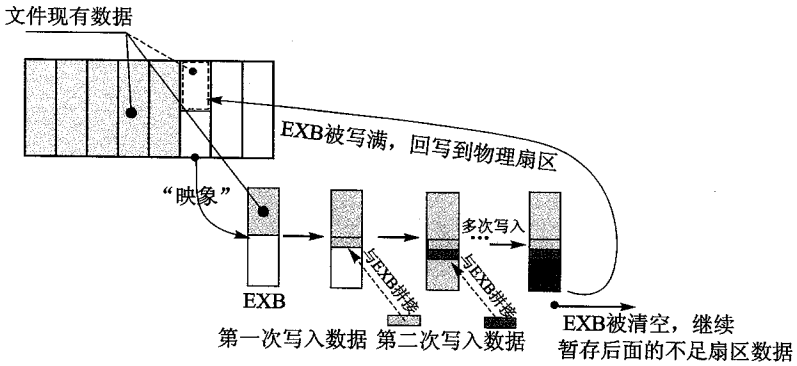


图 4.27　EXB 对不足扇区数据的暂存

4.4.2　EXB 的实现

EXB 其实与 CCCB 类似，都是一种缓冲机制，所以在实现上也有相似之处。但是，EXB 的实现要比 CCCB 简单多了，定义代码如下(znFAT.c)：

```
UINT8 exb_buf[512];
UINT32 exb_sec;
```

其中，exb_buf 是用于存储扇区数据的缓冲区；exb_sec 用于记录当前缓冲区中的数据所属扇区地址，以方便对数据进行回写。

EXB 的具体实现过程相对简单。不过，就像 CCCB 一样，在多文件的情况下，EXB 同样会产生争抢问题，同样会有“共享 EXB”(SEXB)与“独立 EXB”(AEXB)之分，详细参见 znFAT 源代码。

至此，CCCB 与 EXB 就讲完了。最后还有一个很重要的问题我们一定要注意

到:这些缓冲机制的引入使得在数据写入操作最终结束之后,可能还会有一部分簇链或者扇区数据驻留于内存之中。因此,我们一定要做最后一次回写。为了防止读者忘记这一重要步骤,振南为 znFAT 引入了文件关闭函数(znFAT_Close_File),以完成最后的回写操作。所以,在实际应用中,当我们完成了所有的文件数据操作之后,一定要记得调用这个函数(函数具体实现请参见 znFAT 源代码)。

本章讲的内容比较多,包括了多扇区、预建簇链、连续扇区优化、CCCB 与 EXB。这些内容每一个部分都是 znFAT 的精华,都是振南经过长期的研究、创造和实践而提出的。不夸张地说,在写此书的过程中,这一章花费的时间是其他章的 4 倍还要多。曾经有读者建议把本章分散扩展成几章来写,但是振南认为它们是一个有机的整体,是一个完整的创新知识体系,互为依存,互相促进,不可分割。

有人问:“你费了这么大劲,搞了这么多的‘创新’策略,文件数据的写入效率到底能提升到什么水平?”答:“提高了 4 倍多,如果再加上下一章将要讲到的‘非实时模式’方案,数据的写入效率将进一步提升 3～4 倍,最终达到极限,即基本与直接对物理扇区进行写入的效率持平(无文件系统的纯物理层)”。“空口无凭,何以为证?”振南会在后文中用实验来进行验证,敬请翻篇。

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