C语言中的文件

大家对于“文件”这个名词，可能已经不再陌生，但是，如何回答“到底什么是文件”，却可能感到困惑。

实际上，计算机中“文件”的定义，本身就比较困难，因为“文件”本身就是一个抽象的概念。

从计算机如何长时间存储数据说起。

我们知道，内存中的数据是不能长时间存储的（断点之后，数据就丢失了）。而像磁盘、光盘、U盘等存储介质，是可以长时间存储数据（断电后不丢失）。

操作系统为了方便管理那些存储介质，就发明了**文件系统**，文件系统中，就有以**文件**为单位的管理概念，从这个角度看，所谓文件，其实就是以某种方式组织起的信息的最小单位。

以上的文件系统，属于操作系统的范畴，不同操作系统的文件组织方式是不一样的。

C语言，作为源码级别的跨平台语言，它封装不同操作系统之间的的不同，为大家提供了统一的接口。

在C语言中，存在文件结构体以及对应的函数的。

```
struct _iobuf {
        char *_ptr;
        int   _cnt;
        char *_base;
        int   _flag;
        int   _file;
        int   _charbuf;
        int   _bufsiz;
        char *_tmpfname;
        };
typedef struct _iobuf FILE;
```

如以上，我们通过查看标准头文件，找到了FILE的定义，不过事实上，我们不需要太过关心FILE结构体的成员，因为C语言将其封装好了，就是不想让我们手动的操作内部的成员（高内聚、低耦合），在后续的学习中，我们可以发现，对于文件的操作，我们统统是通过相关函数，对FILE*指针，达到间接影响来操作。

文件的打开和关闭

C库函数中，提供了fopen函数用于打开文件，它的函数原型如下：

```
FILE *fopen(char *filename, char *mode );
```

可以看到，它有两个参数：

- filename：需要打开的文件的文件名
- mode：打开的属性（只读、只写、二进制……）

被打开的文件，被封装为了一个FILE结构体，但是我们不用关系其细节，该结构体的地址会被返回（FILE*）。

```
int main(int argc, char* argv[])
{
    FILE* pFile = fopen("shellmad.txt", "w"/*写方式，创造文件*/);
    return 0;
}
```

返回的文件结构体指针，就代表了文件本身。我们后续将通过操作这个指针，达到操作文件的目的。
有打开就有关闭，fclose的原型：

```
int fclose( FILE *stream );
```

它只有一个参数，就是需要被关闭的文件所对应的指针。

```
#include <stdio.h>



int main(int argc, char* argv[])
{
    FILE* pFile = fopen("shellmad.txt", "w"/*写方式，创造文件*/);

    if (pFile == NULL)
    {
        printf("打开文件出错\r\n");
    }

    //功能代码...

    if (pFile != NULL)
    {
        fclose(pFile);
```

```
        pFile = NULL;
    }

    return 0;
}
```

文件读写举例

首先，我们要思考：程序中对文件的写入、读取操作，**本质上**在解决什么问题？
实际上，它在解决信息在不同介质之间的传递问题。

- 读文件：将磁盘中的数据，转移到内存中
- 写文件：将内存中的数据，转移到磁盘中

如果能够像明白以上的问题，就能够很自然的理解C库函数中如何设计读、写文件的接口（fread，fwrite），因为他们都要解决以下几个问题：

- 在内存中的哪里
- 在磁盘（文件）中的哪里
- 要写多少字节

比如：

```
size_t fwrite( void *buffer, size_t size, size_t count, FILE *stream );
```

它的各个参数的含义：

- buffer：将内存中的哪个地址处的内容写文件
- stream：写入到哪个文件中
- size与count的乘积，代表需要写入的字节数目

其返回值，表示成功写入的字节数。

比如一下的实例代码，向文件中写入字符串：

```
#include <stdio.h>
#include <string.h>

```

```
4
5
6  int main(int argc, char* argv[])
7  {
8      FILE* pFile = fopen("shellmad.txt", "w"/*写方式，创造文件*/);
9
10     if (pFile == NULL)
11     {
12         printf("打开文件出错\r\n");
13     }
14
15     //功能代码...
16     char szBuffer[] = "helloworld";
17     int iRet = fwrite(szBuffer, 1, strlen(szBuffer), pFile);
18     printf("成功写入:%d字节\r\n", iRet);
19
20     if (pFile != NULL)
21     {
22         fclose(pFile);
23         pFile = NULL;
24     }
25
26     return 0;
27 }
```

类似的读文件的接口：

```
1  size_t fread( void *buffer, size_t size, size_t count, FILE *stream );
```

- stream：读取哪个文件
- buffer：读取到内存的那个位置
- size与count乘积：读取多少个字节

实例代码：

```
1  #include <stdio.h>
2  #include <string.h>
3
```

```
4
5
6  int main(int argc, char* argv[])
7  {
8      FILE* pFile = fopen("shellmad.txt", "r"/*读方式，打开文件*/);
9
10     if (pFile == NULL)
11     {
12         printf("打开文件出错\r\n");
13     }
14
15     //功能代码...
16     char szBuff[50] = { 0 };
17     int iRet = fread(szBuff, 1, 10, pFile);
18     szBuff[iRet] = 0;
19
20     printf("读取到的字符串:%s\r\n", szBuff);
21
22     if (pFile != NULL)
23     {
24         fclose(pFile);
25         pFile = NULL;
26     }
27
28     return 0;
29 }
```

fopen的属性说明

我们已经知道，fopen的第二个参数是打开的属性，他是字符串，其实有以下几个选项选择：

- "r"：只读方式打开，文件必须存在，如果不存在，则打开失败
- "w"：只写方式打开，文件如果不存在，则会创建新文件；如果文件已经存在，则会删除原文件，再创建新文件
- "r+"：可读可写方式打开，文件必须存在，如果不存在，则打开失败
- "w+"：可读可写方式打开，文件如果不存在，则会创建新文件；如果文件已经存在，则会删除原文件，再创建新文件

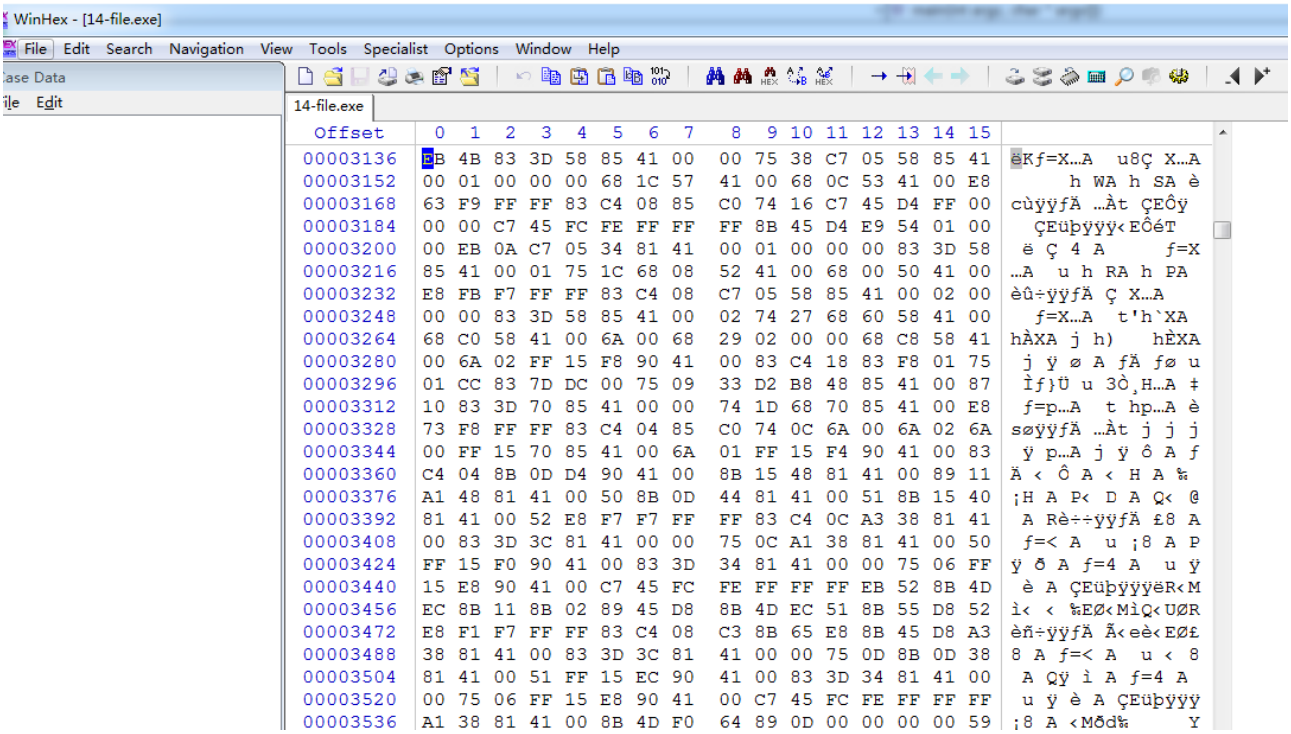
- "a"：附加方式打开，写入的数据在文件末尾。不会覆盖掉原有已经存在的数据
- "a+"：类似，可读可写方式打开

以上所有的属性，还可以与"b"组合使用，比如"rb", "rb+", "ab+"……

- 不与b组合，那么是以文本方式打开
- 与b组合，那么是以“二进制方式”打开

十六进制编辑器

文件中的数据与内存中的数据是类似的，本质上都是2进制（16进制）。我们在调试时，通过内存窗口，可以查看内存中的十六进制数据。
但是，文件不在内存中，我们无法通过调试直接查看它，而可以依赖**十六进制编辑器查看他。**
常见的十六进制编辑器有：WinHex、010editor、ultraEdit。



文本方式与二进制方式的区别

请看以下实例，会出现奇怪的bug：

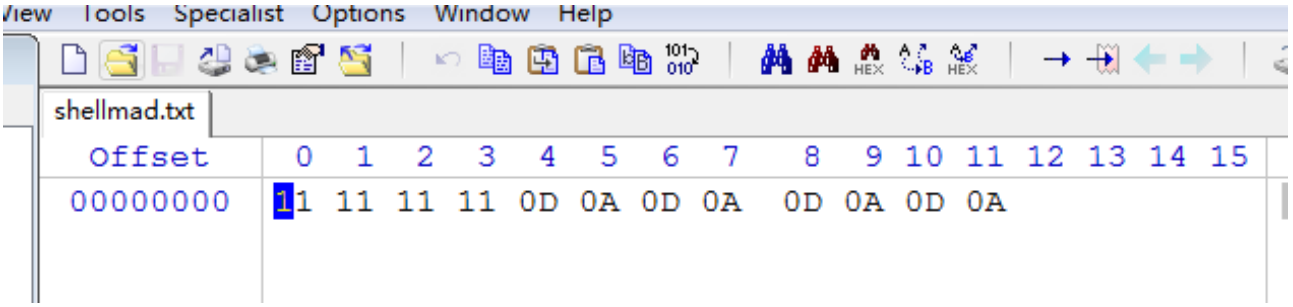
```
#include <stdio.h>
#include <string.h>


```

```
6  int main(int argc, char* argv[])
7  {
8      FILE* pFile = fopen("shellmad.txt", "w"/*读方式，打开文件*/);
9
10     if (pFile == NULL)
11     {
12         printf("打开文件出错\r\n");
13     }
14
15     //功能代码...
16     int iValueAry[2] = { 0x11111111, 0x0A0A0A0A };
17
18     fwrite(iValueAry, sizeof(int), 2, pFile);
19
20     if (pFile != NULL)
21     {
22         fclose(pFile);
23         pFile = NULL;
24     }
25
26     return 0;
27 }
```

按道理，我们像文件中写入了8字节的数据。

我们现在使用十六进制编辑器查看下：



会发现，一共写入了11个字节。经过对比发现，前4个字节是没有错误的。

实际上，这就是因为**文本模式**自作聪明带来的结果。

还记得我们之前有介绍过，不同的操作平台下，对换行的约定是不一致的：

- Windows： \r\n
- Linux：\n

  查阅ASCII码表，\n对应的就是0x0A，\r对应的是0x0D。

  所以，在我们刚刚写入0x0a0a0a0a数据时，因为文件是以**文本方式**打开的，所

以C库函数会自作聪明的，将"\n"替换为"\r\n"。
而是用二进制方式打开，就不会有这个问题。

```c
#include <stdio.h>
#include <string.h>



int main(int argc, char* argv[])
{
    FILE* pFile = fopen("shellmad.txt", "wb"/*读方式，打开文件*/);

    if (pFile == NULL)
    {
        printf("打开文件出错\r\n");
    }

    //功能代码...
    int iValueAry[2] = { 0x11111111, 0x0A0A0A0A };

    fwrite(iValueAry, sizeof(int), 2, pFile);

    if (pFile != NULL)
    {
        fclose(pFile);
        pFile = NULL;
    }

    return 0;
}
```

我们强烈推荐，在一般的文件操作时，使用二进制方式读写文件。

其他文件库函数的使用

fseek, ftell, fflush, feof

```c
int fseek( FILE *stream, long offset, int origin );
```

fseek用于移动文件的偏移，在我们之前的代码中，我们读取文件和写入文件，都是从0偏移处开始的。实际上，我们是可以通过fseek改变它默认偏移。
它的参数含义：

- stream：文件
- offset：偏移
- origin：从哪里开始移动（SEEK_CUR-当前位置，SEEK_END-文件末尾往前，SEEK_SET-文件初始往后移动）

以下代码，先将文件偏移移动到+4的位置，然后覆盖掉相关数据

```
#include <stdio.h>
#include <string.h>



int main(int argc, char* argv[])
{
    FILE* pFile = fopen("shellmad.txt", "rb+"/*读方式，打开文件*/);

    if (pFile == NULL)
    {
        printf("打开文件出错\r\n");
    }

    //功能代码...
    int iValue = 0x33333333;

    fseek(pFile, 4, SEEK_SET);

    fwrite(&iValue, sizeof(int), 1, pFile);

    if (pFile != NULL)
    {
        fclose(pFile);
        pFile = NULL;
    }

    return 0;
```

```
}
```

使用fseek配合fopen、fwrite得到空白大文件

```
#include <stdio.h>
#include <string.h>

int main(int argc, char* argv[])
{
    FILE* pFile = fopen("blankfile.txt", "w"/*读方式，打开文件*/);

    if (pFile == NULL)
    {
        printf("打开文件出错\r\n");
    }

    //功能代码...
    int iValue = 0x00;

    fseek(pFile, 1000, SEEK_SET);

    fwrite(&iValue, sizeof(int), 1, pFile);

    if (pFile != NULL)
    {
        fclose(pFile);
        pFile = NULL;
    }

    return 0;
}
```

fseek和ftell配合得到文件大小

ftell函数，可以获取文件当前所处的偏移。

```
#include <stdio.h>
#include <string.h>

```

```
int main(int argc, char* argv[])
{
    FILE* pFile = fopen("blankfile.txt", "rb"/*读方式，打开文件*/);

    if (pFile == NULL)
    {
        printf("打开文件出错\r\n");
    }

    fseek(pFile, 0, SEEK_END);

    long lOffset = ftell(pFile);
    printf("该文件大小:%d字节", lOffset);

    if (pFile != NULL)
    {
        fclose(pFile);
        pFile = NULL;
    }

    return 0;
}
```

feof可以用于判断是否达到文件末尾

文件缓存及fflush

我们单步调试来查看以下代码，注意观察，是否fwrite执行完毕后，文件的内容就发生了变化。

```
#include <stdio.h>
#include <string.h>

int main(int argc, char* argv[])
{
    FILE* pFile = fopen("testfflush.txt", "wb"/*读方式，打开文件*/);

    if (pFile == NULL)
    {
```

```
        printf("打开文件出错\r\n");
    }

    char szBuff[] = "hello";
    fwrite(szBuff, 5, 1, pFile);

    if (pFile != NULL)
    {
        fclose(pFile);
        pFile = NULL;
    }

    return 0;
}
```

可以发现，当单步时，fwrite调用完毕，数据并**没有被马上写入文件**，但是等到整个程序结束，数据却又写入了文件。
这背后的原因，是C库函数关于文件的**缓冲机制**。
我们知道，所谓的写入文件，就是将信息从内存转移到磁盘中。如果内存中有数据就马上写入到外部设备（磁盘中），很可能付出较大的时间待见。
为了缓解这个矛盾，就提出了“缓冲机制”，其实就是，将需要写入磁盘中的数据，先暂时在缓冲区待命，等**攒到**一定数目，再一口气写入磁盘。
也正因为如此，我们会发现，fwrite调用完毕后，数据并没有直接写入文件中。
如果想提前刷新缓冲区（将缓冲区中的内容，马上写入文件），可以调用fflush函数：

```
fflush(pFile);
```

加入这句代码后，单步可以发现：

```
#include <stdio.h>
#include <string.h>

int main(int argc, char* argv[])
{
    FILE* pFile = fopen("testfflush.txt", "wb"/*读方式，打开文件*/);

    if (pFile == NULL)
    {
```

```
        printf("打开文件出错\r\n");
    }

    char szBuff[] = "hello";
    fwrite(szBuff, 5, 1, pFile);
    fflush(pFile);

    if (pFile != NULL)
    {
        fclose(pFile);
        pFile = NULL;
    }

    return 0;
}
```

文件读写示例

语法高亮软件的实现