这个东西的来由:

在我大学的时候，我开始接触 Blizzard 公司的游戏．当然是喜欢玩他们设计的游戏，最开始的时候是玩 Diablo，后来是星际争霸！不知怎么搞的，我开始想要对他的游戏进行一番剖析，不过因为 Blizzard 公司的东西一般都封装在 MPQ 格式的文件中，我一直没有机会．后来在一个偶然的机会，我得到了一个叫做 Storm 的东西，可以浏览 MPQ 文件，并且附带核心的原代码，在 Windows98 下运行正常，但是在我的 WindowsNT 下就没有菜单，相当于不能运行，所以我想将他的核心代码加一点用户接口代码，但是由于对具体的用法不熟，所以没有成功．我十分的不甘心，所以决定重新破解 MPQ 文件格式．那时正是毕业设计的时候，因为我对于我的毕业设计胸有成竹，所以将我的 80% 的时间放在对 MPQ 文件的剖析之上，到我离校的时候，已经写出 MPQ 浏览器的可以使用的版本，并且写出了一个可以生成带压缩格式的 MPQ 文件编译器！一切都在 3 个月里完成了！工作后，因为我不是在软件行业，所以没有多时间继续解剖，所以决定公布原代码，并且在空闲的时候写一点帮助文件．这不，MPQ 文件格式剖析不就这样出来了?

什么是 MPQ 文件?

不用我讲，关心这个东西的人都对它有一定的了解！MPQ 文件相当于一个文件包！它存储着 Blizzard 游戏中需要的极大部分的数据：图象，声音，动画,电影，还有其他一些和游戏剧情有关的数据和脚本．大家可以使用 MPQ 文件浏览器查看一下(我的主页上有下载的 :) ．Blizzard 公司在他的游戏中使用了一个库 Storm.dll，其中有用于读取 MPQ 包的函数，但是不知什么原因，这个库文件中确没有写文件的函数．只想读取 MPQ 文件的朋友可以 到那里去看看，可以节省很多的时间的．当然，这篇文章是对 MPQ 文件的结构进行剖析的，对于它的应用我们可以撇开不管．下面涉及....

MPQ 包的结构!

MPQ 是一种文件管理包，它可以独立成为一个文件，也可以"寄生"在其他的文件里面，但是，一个文件中只能有一个压缩包．并且是在文件的最后部分．例如，MPQ 包可以放在可执行文件的结尾，形成一个新的带 MPQ 包的可执行文件(例如星际争霸光盘上的 install.exe，战网升级文件 BW_107.exe 等等)．但是，MPQ 在文件中的起始位置是有一定的原则的：MPQ 包只能在距离文件起始位置 512 字节的整数倍位置起始．其他位置都不认为是一个合法的 MPQ 包！MPQ 包内的所有文件都不保存文件名，文件的管理采用 hash table - block table 管理方式．hash table 负责文件的识别，block table 负责文件数据的具体定位！当然，这两个表格都加了密．

详细讲解 MPQ 包的结构!

在 MPQ 包的起始部分有一个包信息头，使用 C 语言语法是这样描述的:

```
typedef struct tagINFOHEAD{
```

```
    DWORD tag;                  // Must be "MPQ", '\0x1A'
    DWORD Version;              // Must be 0x20; Version?// It's the size of
                                // the InfoHead (32 Bytes).
    DWORD iWholeMpqSize;        // The Size of the MPQ package.
    char C1;                    // UNKOWN
    char C2;                    // UNKOWN
    char C3;                    // the size of Buffer2 is Based on This.
                                //Always 3 512<<3 = 4096
    char C4;                    // UNKOWN
    DWORD iHashTableBegin;      // The Beginning of the HashTable.
    DWORD iBlockTableBegin;     // The Beginning of the BlockTable.
    DWORD CountOfHashTable;     // the Count of the Hash table.
    DWORD CountOfBlockTable;    // the count of the Block table.
}TINFOHEAD, *PINFOHEAD;
```

tag:	这个地方必须等于"MPQ\x1A". 是 MPQ 包的起始标志, 相当于 BMP 位图文件的起始的 "BM" 标志.
Version:	我相信这个地方是 MPQ 包头信息的数据长度, 7 个 DWORD 加上 4 个 BYTE 应该是 32 个 BYTE. 由于在逆向工程开始的时候, 我以为是 MPQ 格式的版本信息, 所以将它命名为 Version.
C1, C2, C4:	这几个字节的用途我还是不大清除. 所以没有座出什么标识.
c3:	这个字节与 MPQ 中文件的分割有关, 一般等于 3. MPQ 包中, 有些文件是分块存储的, 它把一个完整的文件分割成一小块一小块的, 便于文件的定位(例如 SetFilePointer 等等). 这个包里面的文件每一块是多大, 由这个字节决定. 将 512 左移这个数(一般是 3), 就得到了文件分块的大小(512 << 3 = 4096 即每一块大小是 4k 字节).
iHashTableBegin:	Hash table 在 MPQ 包中的起始位置. 这个位置是相对于 MPQ 包的起始位置的.
iBlockTableBegin:	Block table 在 MPQ 包中的起始置. 这个位置也是相对于 MPQ 包的起始位置的.
CountOfHashTable:	Hash table 的项数! hash table 是一个数组, 包含很多的 hash item, 每一个 hash item 占用 16 字节(后面会具体分析的), 这里表示 hash table 中 hash item 的个数.
CountOfBlockTable:	Block table 的项数! (一般来说, 这个数就是 MPQ 包中所包含的文件的数目, 但是也有例外.) 和 hash table 差不多, block table 也包含很多的 block item, 每一个 block item 占用 16 字节(后面会具体分析的), 这里表示 block table 中 block item 的个数.

在上面的信息头结构中, 我提到了两个东西: hash table 和 block table. hash table 我称之为哈希表, block table 我称之为块表! 当然使用英语更确切一些! 这两个表格其实就是两个数组(当然是解密以后), 知道首地址, 一个下标就可以完全定位了! 确实很方便.

下面看看 hash table 中单项的结构:

```
typedef struct tagHASHTABLEITEM{
    DWORD dwHashValue1;          // 文件名加密后的第一个 32 bit 数据
    DWORD dwHashValue2;          // 文件名加密后的第二个 32 bit 数据
    DWORD dwFlag;                // 这个 hash table 的一些设置.
    DWORD iBlockIndex;           // 这个文件的数据在 block table 中的位置
(也就是数组的下标).
}HASHTABLEITEM,*PHASHTABLEITEM;
```

dwHashValue1, dwHashValue2:	这里面有两个加密了的数据(64 位), 这两个数据是将文件名处理后的数据, 一般来说不同的文件名产生的 64 位数据是不相同的. 这样用以代替文件名标识文件.
dwFlag:	这个数据表示这个 hash item 的一些属性! 如果是 0xFFFFFFFF (-1) 的话, 表示这个 hash item 是无效的.
iBlockIndex:	这个文件的 block item 在 block table 中的位置, 也就是说在 block table 这个数组中的下标. 用以对文件的数据进行定位.

我们再看看 block table 单项的结构:

```
typedef struct tagBLOCKTABLEITEM{
    DWORD FileStartAt;           // 文件数据在 MPQ 包中的起始位置
    DWORD nPackedSize;           // 文件压缩后的数据长度/文件数据在 MPQ
包中占据的长度.
    DWORD nFileSize;             // 文件本来的, 没有压缩前的数据长度
    DWORD dwFlag;                // 文件在 MPQ 包中的一些属性! 比如压缩方
法, 加密与否等等.
}BLOCKTABLEITEM, *PBLOCKTABLEITEM;
```

FileStartAt:	文件数据在 MPQ 包中的起始位置, 这个数据是相对于 MPQ 包的起始位置的.
nPackedSize:	文件在 MPQ 包中占据的数据长度. 对于压缩文件来说, 就是压缩后文件的长度, 对于没有压缩的文件来说, 一般就是文件的大小了.
nFileSize:	这个文件的真实大小. 也就是没有压缩/处理前的文件大小.
dwFlag:	这个文件在 MPQ 包中的一些属性. 包括文件的压缩方法, 加密方法, 还有文件有效与否等等.

文件的搜寻过程:

将文件名进行处理(也就是说:加密) 得到一个数, 将这个数作为起始下标, 在 hash table 中搜索整个表格, 如果其中某一项的 64 位加密位和与文件名处理后得到的 64 位数据相同, 那么就认为这个 hash item 项, 然后查看这个 hash item 项的标志位, 检查合法与否, 如果合法, 那么可以得到这个文件的 block item 的下标, 这样查找 block table, 检验 block item

的有效性，如果 Block item 有效，那么我们可以得到这个文件的很多的信息了，并且可以定位这个文件的数据在 MPQ 包中所占用的位置．如果知道具体的解压缩/解密方法，我们就可以读出文件的数据了．在搜寻的途中，如果有一个地方没有通过(比如 hash item 无效)，那么我们就必须认为所要求的文件不存在．

总之，搜寻文件的方法就是通过文件名在 hash table 之中找到 hash item 项，然后再在 block table 中找到 block item 项，通过 block item 定位文件数据．最后解密解压还原文件数据．大概如下图所示．知道了怎样读出 MPQ 文件，想要创建 MPQ 包就很容易了．:D

内部文件的存储方法

在 MPQ 包中，内部文件的存储是多种多样的，对于那些电影文件(.SMK, .BIK) 等等，这些文件是不经过任何处理，直接将数据复制进 MPQ 包之中的，但是对于绝大多数的文件来说，数据是经过压缩后存储在 MPQ 包之中的，还有一些文件还经过加密存储在文件中．文件具体采用什么方法存储在 MPQ 包中，这个由该文件的 Block item 项标明．

我们再来看看 block item 的数据结构:

```
typedef struct tagBLOCKTABLEITEM{
    DWORD FileStartAt;          // 文件数据在 MPQ 包中的起始位置
    DWORD nPackedSize;          // 文件压缩后的数据长度/文件数据在 MPQ
                                //包中占据的长度.
    DWORD nFileSize;            // 文件本来的，没有压缩前的数据长度
    DWORD dwFlag;               // 文件在 MPQ 包中的一些属性! 比如压缩方
                                //法，加密与否等等.
}BLOCKTABLEITEM, *PBLOCKTABLEITEM;
```

看到了最后一个成员变量 dwFlag 没有? 就是这个变量标明该文件的存储方法/加密方法．对于这个变量的详细说明如下．

- 0x00000100 - 文件使用 DCL 压缩方法分块压缩，分块存储在 MPQ 包中．
- 0x00000200 - 文件先进行压缩，然后整个压缩数据存储在 MPQ 包中．
- 0x00010000 - 文件加密
- 0x00020000 - 文件加密，密钥经过包大小和文件存储位置修正过!
- 0x80000000 - 标明这个文件有效!

上面这些数据经过逻辑"或"后，存储在 dwFlag 中．

可以看到，MPQ 包的文件存储方法是很复杂的．但是我们可以挑出几种简单的方法来说明一下．

最简单的情况

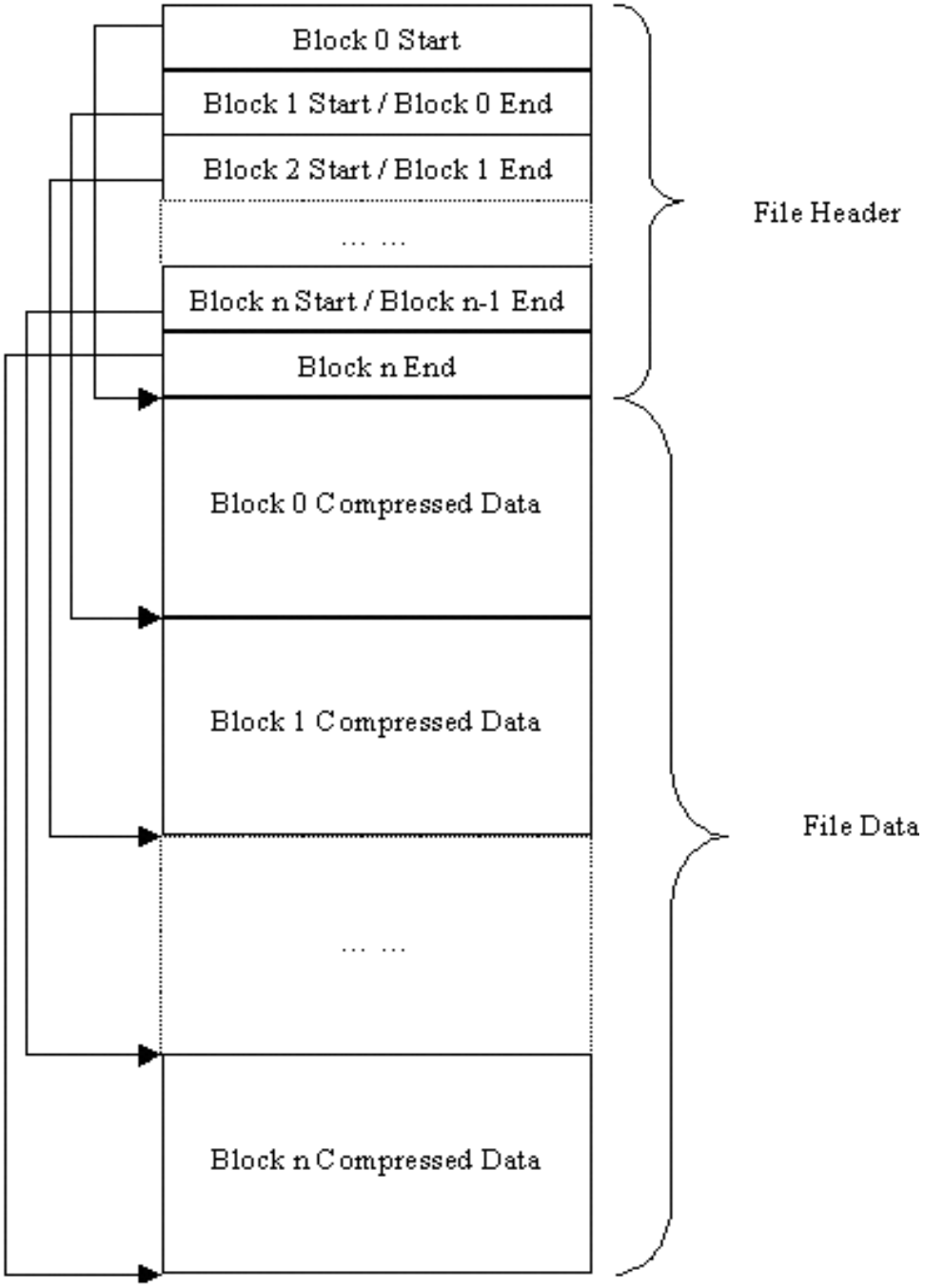
最简单的莫过于 0x80000000 了，文件数据直接 Copy 到 MPQ 包中，不经过任何的处理．对于这种文件，我们只要从 Block item 中读出 FileStartAt，将 MPQ 文件指针指向 MPQ 包的这个位置，后面 nPackedSize 个字节就是这个文件的内容．直接读出来就是了!

稍微复杂一点的存储方法

再来介绍一下标志位是 0x80000100 (文件有效，使用 DCL 方法分块压缩，分块存储在 MPQ 包中!

对于这种格式的存储方法．需要使用 Packware 公司的 DCL 数据压缩库才能解压数据.这种文件的的数据先被分割成一些数据块(一般是 4k 大小)．然后每一块数据经过 DCL 压缩，存储在 MPQ 包中间．

我们看看这种文件的存储结构:
首先在文件数据开始的地方有一个偏移量表格(数组)，记录了每一个文件数据块的起始偏移位置，这个表格大小就是 (fiesize + blocksize - 1)/blocksize +1 个 DWORD．第 0 个 DWORD 标明第 0 块从什么地方开始，第 1 个 DWORD 标明第 1 块数据的起始位置(同时也是第 0 块数据的结束位置) ... 依此类推，但是最后一个 DWORD 标明最后一个数据块的结束位置．注意：这里的偏移量是相对于文件数据开始的地方而言的．下面这幅图表示的就是这个意思:



知道了这个表格的结构，我们可以试着读出文件每一块经过压缩后的数据，然后使用 DCL 压缩库解压，最后综合而成这个文件的整个数据了．可以看看我的 MPQ complier 的原代码! SFile.cpp 中有个函数 CompressFile 就是创建这种格式文件数据的，值得参考!

DCL (Data Compress Lib)数据的解压缩

从上面可以看出，MPQ 包中一些文件是经过 DCL 数据压缩库压缩了的．我们必须有 DCL 才能对数据进行解压缩．DCL 是由 Packware 公司开发的一种数据压缩函数库，有版权的．一般由 3 个函数组成:

- implode 负责压缩数据
- explode 负责解压数据
- crc32 负责数据校验

对于这个函数库，我不想讲得太多，大家可以到 Packware 公司的主页去看看．我们只有使用这个函数库中的 explode 函数，才能将 MPQ 包中的一些文件的数据还原．

其他的文件存储方法/加密方法

对不起，因为时间不是很多，我剖析出了其他文件的加密方法和压缩存储的方法，但是没有时间验证，所以不想写出来．请大家见谅!

一些加密解密算法!

注意，如果需要了解这个地方提到的算法，请先 下载 MPQ 文件编译器的原代码．原代码里面有一个叫做 codec.cpp 的文件，所做的事情就是 MPQ 文件的加密解密工作．

在 MPQ 包中，最重要的数据结构 hashtable 和 block table 都是经过加密后才写入文件的，并且文件名称也是经过加密后的 64 位数据．所以，如果不知道 MPQ 的具体加密方法，并且按照相应的方法解密，我们无论如何也不能将需要的数据读出来．这里我们介绍一下 MPQ 包中的基本加密方法．

密码表格

在加密 MPQ 包的时候(加密文件名，hash table，和 block table)，他们使用了一个密码表，这个表格当然是有一定规律的，后面将要介绍这个表格产生的方法．这里先来说一下这个表格的结构．

MPQ 加密使用的是一张有 5 个页面的表格，分别编号为：页面 0，页面 1，页面 2，页面 3 和页面 4，每一个页面有 256 个 DWORD 的元素．当然每个页面都有他们自己的用途．例如：

页面 0：用来加密文件名，他产生查找这个文件时在 hash table 中搜索所用的初始索引值；

页面 1：也是用来加密文件名，产生文件加密位的第一个 32 bit 值；

页面 2：还是加密文件名称，产生文件加密位的第二个 32 bit 值

页面 3：用来产生加密解密 hash table /block table 时使用的密钥．

页面 4：是用来加密解密 hash table 和 block table 的

这个表格在我的原代码中是这样定义的.

```
DWORD MyCodeTable[0x5][0x100];
```

在读取/创建 MPQ 文件的时候，我们先需要创建这个表格，才能作其他的加密解密工作．下面我们看看怎么来创建这个表格.

还原密码表格

其实，我这个表格还是很有规律可循的．就是从某个数据开始，进行一系列的变换，其中产生的结果依次填入表格就是的了！这个初始数据就是 0x100001．具体算法请查看我的 MPQ 编译器的原代码 codec.cpp | InitCodeTable() 函数.

```
void InitCodeTable()
{
    if( MyCodeTable != NULL )
    {
        DWORD dwSeed = 0x100001;
        DWORD p2 = 0x2AAAAB;
        DWORD dwHeight;
        DWORD dwLow;

        for( int i = 0; i < 0x100; i++)
        {
            for(int j = 0; j<5; j++)
            {
                dwSeed = (dwSeed*125+3)%p2;
                dwHeight = (dwSeed&0xFFFF)<<16;
                dwSeed = (dwSeed*125+3)%p2;
                dwLow  = (dwSeed&0xFFFF);
                MyCodeTable[j][i] = dwHeight | dwLow ;
            }
        }
    }
    bCodeTableInited = TRUE;
}
```

文件名加密 / 字符串加密

文件名加密，我使用的函数原形是这样的：

```
DWORD Encode(char* csString, DWORD nCodePage);
```

- csString 就是要加密的文件名/字符串了；
- nCodePage 是选择的密码表的页面号码(0,1,2,3,4)；

函数返回加密后的 32 bit 数据．这里要注意一点就是，当使用第 0 页密码加密文件名后，我们要将得到的 32 位密码和 MPQ 文件头 INFOHEADER 中的 CountOfHashTable 变量尽心逻辑"与"，才能得到初始的 hash table 搜索下标．否则，在程序中就会出现数组越界情况了．(这个 32 bit 数据一般大于 CountOfHashTable.)

加密还原 hash table/ block table.

- 在我的原代码中，有两个函数使用来做这些工作的．原形如下

```
void DecryptTable(DWORD nTableSize, void* lpTable, DWORD
dwHashValue);
void EncryptTable(DWORD nTableSize, void* lpTable, DWORD
dwHashValue);
```

- nTableSize：这个参数是 hash table /block table 的大小，用 BYTE 单位，所以必须使用 CountOfHashTable/ CountOfBlockTable 乘以 16 (每一个 item 的大小是 16 BYTE)．
- lpTable：这个参数就是指向需要加密/解密的 hash table / block table 的首地址了，并且，结果也保存在这里．
- dwHashValue：加密解密所用的密钥．对于 hash table 这个密钥是将字符串"(hash table)" 使用密码表格的页面 3 进行加密后的 32 bit 值；对于 block table，这个密钥是将字符串"(block table)" 使用密码表格的页面 3 进行加密后的 32 bit 值；
- 可以在我的原代码中看到这样的代码，这些代码所作的事情就是加密/解密这两个表格．代码段如下所示．

```
// 解密 hash table .
DecryptTable( HashTableSize, lpHashTable, Encode(szHashTable, 3 ));
```

```
EncryptTable(    // 加密临时缓冲区中的 hash table.
    hMPQ->InfoHead->CountOfHashTable* sizeof(HASHTABLEITEM),
    HashBuffer,
    Encode(szHashTable, 3)
);
```

```
// 加密 BlockTable
EncryptTable(
    hMPQ->InfoHead->CountOfBlockTable*
sizeof(BLOCKTABLEITEM),
    BlockBuffer,
```

```
        Encode(szBlockTable, 3)
    );
```

```
    // 解密 Block table .
    DecryptTable( BlockTableSize, lpBlockTable, Encode(szBlockTable,
3 ));
```

- 上面写了一些关于 MPQ 文件中基本的加密解密方法和函数的使用方法．最好的办法还是查看我的 MPQ 编译器的原代码，点击这里下载．当然，在 MPQ 文件中还使用了其他的加密方法，但是这些加密算法都是建立在那个 5 页面的密码表格上的．其他的加密解密方法我还没有验证，所以不在这里胡说八道．大家也不要问我．