

MDX / MDD 文件格式

1 简介

最近在做一个词典 CLI 工具，所以我需要解析 .mdx 这种常见的词典格式。

它分为 header, key 以及 record 三个部分。其中 key 和 record 都有自己的 header，并且分为多个块。下面我们先考虑一下每个块具体是怎么回事。

2 Header

```
struct Header {  
    // payload 的总长度。  
    [[be]] uint32_t len;  
    // UTF-16 编码的 XML。  
    [[le]] uint8_t payload[len];  
    // XML 的校验码。  
    [[le]] uint32_t crc;  
};
```

这里的校验码是 Adler-32：

```
a = 1  
b = 0  
for x in bytes(data):  
    a = (a + x) % 65521  
    b = (b + a) % 65521  
result = (b << 16) | a
```

2.1 XML 属性

MDX 的根元素通常是 <Dictionary .../>，MDD 的根元素是 <Library_Data .../>。属性顺序不固定，值使用双引号，可能包含 & < > 之类的转义字符。

下面是一些常见的键。有一些只和写入有关，我们可以不用管。

属性	常见值	作用
GeneratedByVersion	1.2、2.0	版本号，决定布局。
RequiredEngineVersion	1.2、2.0	写入： 所需的版本，只读时可以忽略。
Encrypted	0、1、2、3、No、Yes	加密形态，见小节 5
Encoding	UTF-8、UTF-16、GBK、GB2312、Big5	key 和 record 的字符编码
Format	Html、Text	正文格式
KeyCaseSensitive	Yes、No	写入： 是否区分大小写
Stripkey	Yes、No	写入： 是否忽略标点和空白
Title	文本	词典标题
Description	文本	描述
CreationDate	2026-9-18	创建日期
DataSourceFormat	106	可忽略
StyleSheet	文本	CSS，可能需要渲染
RegisterBy	Email、DeviceID	注册 key 的类型
RegCode	十六进制串	注册 key 的加密形式

我们只处理 2.0 版本。1.x 版本主要的区别是所有的长度都是 32 位而不是 64 位。

MDX 和 MDD 文件虽然格式相似，但是对 Encoding 的处理不同。MDX 就是按照 Encoding 字段编码的，默认 UTF-8，而 MDD 的 key 全部都是 UTF-16LE，而 record 不是文本。

3 Key

这个部分记录每个词条所在的 record 的字节偏移。偏移量是从文件最开头开始算的。它分为 header, block metadata 以及 block 列表三个部分。每个 block 可能装有很多词条。

```
struct KeySectionHeader {
    // Block 总数
    [[be]] uint64_t blockCnt;
    // 词条总数
    [[be]] uint64_t entryCnt;
    // metadata 解压后的字节数
    [[be]] uint64_t dMetaSize;
    // metadata 长度
    [[be]] uint64_t metaSize;
    // block 总长度
    [[be]] uint64_t blockSize;
    // header 本身的校验码
    [[be]] uint32_t crc;
};
```

Meta 部分是一个压缩块。每个压缩块形如

```
struct CompressedBlock {
    // 压缩类型。0 为不压缩, 1 为 LZ0, 2 为 zlib。
    [[le]] uint32_t cType;
    // 解压后数据的校验码
    [[be]] uint32_t crc;
    [[le]] void *payload;
};
```

大部分内容都是 cType = 2 的。

解压之后, meta 的字段如下:

```
struct KeyMeta {
    struct Blockmeta {
        // 词条数
        [[be]] uint64_t entryCnt;
        // 第一个 key。最后一个 codepoint 是 0
        [[be]] uint16_t key1len;
        [[be]] char      key1[(key1len + 1) * sizeof(Codepoint)];
        // 同理, 最后一个key
        [[be]] uint16_t lastKeyLen;
        [[be]] char      lastKey[(lastKeyLen + 1) * sizeof(Codepoint)];
        // block 本体压缩/解压的长度
        [[be]] uint64_t blockSize;
        [[be]] uint64_t dBlockSize;
    } meta[blockCnt];
};
```

Block 本体自然也被压缩。它解压后的形态则是

```
struct KeyBlock {
    [[be]] uint64_t offset;
    [[be]] char      key[? * sizeof(CodePoint)];
};
```

这里的 key 没有长度, 纯粹以 codepoint 0 结尾。

4 Record

这是词条正文。

```
struct RecordHeader {
    // 和 KeyHeader 一样。
    [[be]] uint64_t blockCnt;
```

```

[[be]] uint64_t entryCnt;
// 这次不包括压缩前的大小了, 因为 meta 根本没有压缩。
[[be]] uint64_t metaSize;
[[be]] uint64_t blockSize;
};

struct RecordMeta {
    struct Blockmeta {
        // 每个 block 压缩和解压时的大小。
        [[be]] uint64_t cSize;
        [[be]] uint64_t dSize;
    } meta[blockCnt];
};

```

记录块比较特别, 是全局的。我们需要将解压后的字节按顺序拼起来, 而词条的正文就是对应 key 的 offset 到下一个 key 的 offset 之间的这段内容。

5 加密

5.1 Encrypted

Encrypted 可能是数字, 也可能是 No/Yes。常见解释:

值	含义	
0, No, 空	不加密	1, Yes
需要密钥	2	key metadata 混淆
3	两者都有	

5.2 Metadata 混淆

解码:

```

seed = [...wrapper[4:8], 0x95, 0x36, 0x00, 0x00] # 8 字节
key  = RIPEMD128(seed)                          # 16 字节

prev = 0x36
for i in 0 .. len(payload)-1:
    x = payload[i]
    t = ((x >> 4) | (x << 4)) & 0xff
    payload[i] = t ^ prev ^ (i & 0xff) ^ key[i % 16]
    prev = x

```

反混淆后的 payload 就是正常的压缩流 (通常是 zlib)。

我们尚不知道具体如何根据用户密钥加密/解密, 也不准备支持这一点。