# 模型IO二进制格式方案

程序时常需要保存自身的文档数据，更应该优先考虑通用的文本格式，比如 XML, JSON, Lua 等。这些通用的文本格式已经存在大量的工具和库，可以省下很多功夫。

文本格式读取慢，并且文件尺寸也比较大（就算经过 zip 压缩)，大多数情况下这都不是什么问题。但一些场合，要求更快读取速度，更小文件尺寸，这时就需要设计一种二进制文件格式。

## **整体文件结构**

常见二进制文件格式，时常采用 “文件头 + 分区” 的结构：

file header

section 0

section 1

section 2

section 3

....

section N

文件头描述了文件的整体信息，常见的字段有魔数、版本号、检验码、文件大小等等。文件头根据文件的具体用途会有额外的字段，比如一张图片，文件头当中就可以含有表示图片尺寸的字段。

文件的整体信息通常放在文件的最前面，所以才叫文件头。但少数情况下可以放到文件的最后面，变成文件尾，但基本上不会放在文件中间。分区的结构通常会是：

tag + length

section data

tag 和 length 合起来是分区头部，后面紧跟着分区的具体数据。

tag 可以是一个整数，也可以是一个字符串。tag 用来标识分区，不同的 tag 表示不同的分区种类，不同的分区种类有各自不同的读取方式。比如：

**#define kPicShapeTag 1**

**#define kPreivewTag 2**

当 tag 为 1 时，就表示是这个分区存放的是图元数据，当为 2 是表示这个分区存放一张预览图。

length 是个整数，表示分区数据的具体长度（不包括分区头部）或者表示整个分区的长度（包括分区头部）。

这种分区结构使得文件格式容易扩展，有新需求时就定义一个新的分区类型，原来的文件结构不需要修改。也容易「向上兼容」。

这里需要解释什么叫文件的「向上兼容」。「向上兼容」跟「向下兼容」对应。「向下兼容」指旧版本程序产生的旧版本文件格式，可以使用新版本的程序打开。比如程序 1.0 生成文件格式 1.0，新版本程序 1.2 可打开文件格式 1.0。「向上兼容」指新版本程序生成的新版本格式，可以使用旧版本程序打开。比如新版本程序 1.2 最新定义了文件格式 1.1，旧版本程序 1.0 虽然早就发布了，但还是可以打开新版本文件格式 1.1，虽然可能会缺少一些新版本的功能。

一个应用程序升级，「向下兼容」是最起码的要求，但「向上兼容」就并非所有程序都可以做到的。使用分区的结构，旧版本程序读取二进制格式时，一旦遇到不认识的分区类型可以直接忽略掉，这样就更容易实现「向上兼容」。

注意程序的版本号跟文件格式的版本号可以是不对应的。比如程序版本是 1.2，但文件格式可以还是停留在 1.1。

有时候分区数据本身再细分，再次采用这种分区结构。比如一个矢量绘图程序，需要记录下图元。可以设计成：

绘图 Header

绘图分区

预览图分区

绘图分区再划分成具体的图元分区，仍然采用 tag + length 的结构。

**#define kShapeCircle 1**

**#define kShapeRect 2**

**#define kShapeBezier 3**

这样就可以不断地增加各种图元。

## **文件头魔数（magic number)**

文件头当中，会有一个数字作为文件格式的标识。这个数字可以随意选定任何值，也可以占据任何字节（通常是 4 字节或者 8 字节），但这个数字选定后就会固定下来，基本上不会再有变化。在编程领域，一些说不清来历比较任意的数字会被称呼为魔数（ magic number）。因此这个随意选定用于标识文件格式的数字，就叫文件格式魔数；这个数字通常放在文件头当中，有时也就称为文件头魔数。

文件格式魔数可以随意选取，看设计者自身的喜好。有些设计者会取自己名字的缩写，有些会取自己的生日，有些会取当前日期，有些取文件格式的后缀名，有些仅仅是抛色子得到的随机数字。

比如 zip 格式的魔数在字节序是小端机器上是 0x04034b50，这个整数表示成 char[4] 就是 "PK\x03\x04"，其中 PK 就是设计者 Philip Katz 的名字首字母。

为了方便处理，避免数字在不同字节顺序的机器上有所区别，有时文件头魔数会定义成多字节格式，比如：

**struct** **Header**{

**uint8\_t** md5[16]; *// md5 作为 检验码，*

**char** magic[8]; *// 魔数*

}

Header header;

memcpy(header.magic, "vecpaint", 8);

文件头魔数无论被当成整数还是多个字节处理，它的作用都是相同的，只是作为一个文件格式的标识。

比如一个矢量绘图软件定义的文件格式魔数为 "vecpaint"，位于文件开始的第 16 个字节处（最前面 16 个字节放 md5）。这个软件保存数据输出文件 example.vecpaint，重新读取这个文件时，就首先判断文件对应位置，对应的魔数是否为 "vecpaint"，假如是的话，就进一步读取解析数据；假如并非这个魔数，就读取失败。

这个 example.vecpaint 文件既然是软件产生的，从后缀名就可以关联到对应的程序，当然一定会读取正确，为什么还需要先判断魔数呢？那我举个反面例子，我有一张 example.png 图片，之后我将它的后缀名改成 vecpaint，再用这个矢量绘图软件打开。假如不判断魔数当成正常文件处理，就很有可能出问题，甚至会引起程序崩溃。

另外一个软件可以处理多种文件格式，比如 Photoshop 可以处理 png 图片格式，也可以处理 jpg 图片格式。这样就通过魔数判断出各种文件格式，再进入对应的读取流程。

常见的文件格式 png, jpg，exe，swf，psd，pdf，gif，zip 等等，魔数的位置和值都有所不同，这样就可以区分出各类不同的二进制格式。而单纯采用后缀名来判断是不准确的。

需要注意的是，假如魔数正确，文件格式并非一定能够读取正确，还要进一步判断。但假如魔数错误的话，这个文件就一定会读取失败。

## **检验码**

文件头通常还会有个检验码，用于检验文件是否完整且没有经过修改。这个检验码可以使用 crc, 可以使用 md5，也可以使用其它算法。只要达到这个目的就行。

假如文件都是在本机写入和读取，这个检验码没有什么大作用。假如文件格式经过网络传输，这个检验码就十分有用。网络传输经常会发生数据不全，或者某些字节被改变了，导致文件数据不完整。通过这个检验码可以检测出这种问题，以便再做进一步处理（比如重新下载一次）。

现在的网络这样可靠，为什么还需要检验？举个例子，一个涂鸦文件格式，最初的文件格式是不带检验码的。这样当用户将涂鸦文件上传到服务器时，上传到一半，用户断网了。这时服务器上的涂鸦文件就只有一半。以后下载读取时，因为文件不完整就一直读取失败，有时甚至会引起客户端崩溃。设计这个涂鸦格式的第二版时，放一个 md5 作为检验码，就算服务器出错，客户端也可以一开始就识别出来进行异常处理。假如没有这个检验字段，客户端是没有办法知道数据是否完整的。

设计二进制文件格式时，通常会在最开头放 16 个字节的 md5。

**char** md5[16];*// 文件的剩余数据*

用文件的剩余数据计算出它的 md5, 存放在文件最开头。这个 md5 一方面可作为这个文件的检验码，另一方面可以作为这个文件的 key。读取文件格式的时候，先判断魔数是否正确，再重新计算出 md5 进行比较。md5 出错，表示文件不完整或者经过改动。

## **版本号**

文件头通常还会包含版本号。版本号不同的文件格式，读取方式可能会有所不同。不支持「向上兼容」的软件，碰到比它可以支持的更高版本的文件格式，直接读取失败，并返回一个错误信息。

版本号有时只是单独一个数字，不断往上递增。有时也会拆分成两个数字，为主版本号和次版本号。主版本号修改，通常表示文件格式发生大变动。而次版本号修改，通常只是表示添加了一些小功能。

软件的版本号制定方式是多种多样的。有些软件会直接采用发布年份作为版本号，比如 Windows 98，Office 2013。大部分软件的版本号采用三个数字，用小数点分隔，格式为：

主版本号.次版本号.补丁版本号

主版本号通常表示功能有很大改动，甚至界面都改掉了；次版本号用于表示添加了一些小功能；补丁版本号只是用了 fix bugs。iOS 系统的采用这种版本表示方式。

## **字节顺序**

字节顺序有大端字节序和小端字节序。不同机器字节序有可能不同，设计文件格式时需要考虑文件用什么字节序保存数据的。不然有可能在这一台机器上生成的文件，传输到另一台机器上就打开失败了。

若不注意字节顺序，用类似下面的代码去读写数据：

**static** **void** **writeI32**(std**::**vector**<uint8\_t>&** data, **int32\_t** val){

**uint8\_t\*** ptr **=** (**uint8\_t\***)**&**val;

data.insert(data.end(), ptr, ptr **+** 4);}

**static** **uint32\_t** **readI32**(**uint8\_t\*&** ptr){

**int32\_t** val **=** **\***((**int32\_t\***)ptr);

ptr **+=** 4;

**return** val;}

这样的代码初看起来没有什么错误，但这些代码时依赖机器字节序的。同样的代码，保存一个 4 字节整数 0x01020304，在大端机器，会保存为：

0x01 0x02 0x03 0x04

但在小端机就会被保存成:

0x04 0x03 0x02 0x01

这样就引起读写不一致。设计自己的文件格式或者去分析其它的文件格式，一定要注意字节序。有些文件格式，可以同时支持大端和小端字节序。它有文件头中有个字段指明文件保存的时候是采用什么方式保存。

但现在绝大多数情况下，我们用到的机器都是小端字节序的。文件格式倾向于小端存储。这样读写的时候，会更加方便些。

## **字节或者位**

现在的计算机会将 8 bit 划分成一个字节，平时计算和处理数据都用字节作为最小的单位，有些语言甚至不支持位运算。文件格式也倾向采用字节的存储方式。一个整数存储成 1 字节，2 字节，4 字节或者 8 字节。

但情况总有例外，有些场合需要更高效地减少文件大小。保存数字时用字节为单位存储也觉得浪费，这是可以将数字按位存储，读入或者写入的时候，都使用位运算。比如保存一串数字：

10, 11, 12, 14, 15, 16, 18, 19, 14, 30

这串数字都不会大于 32，可以 5 位表示。先保存这个数字 5，表示之后的数字以多少位保存。再依次用位操作，将每个数字用 5 位保存。

比如典型的是 swf 文件格式的设计。它大量使用了位存储，使得 swf 文件的格式压缩得很小。但这样写入和读取都会有点麻烦，更加耗费计算资源。

根据设计目的，假如需要大量压缩尺寸，可以采用位存储，但读取速度就慢。采用字节保存，读取速度会快，但文件尺寸通常会比按位保存要大些。其实就是空间和时间的取舍，空间换时间或者是时间换空间。

swf 的一个目标设计是方便经过网络传输，当时的网络条件比较差，它这样使用位存储是合理的。

## **字节对齐**

假设机器是 32 位（也就是 4 字节），当数据的地址为 4 的倍数时，计算机读取速度会更快。可以将计算机的地址，以 4 字节为单位，从 0 开始，将地址划分成一个个小格子，每个格式都可以放 4 字节，格子开始地址都是 4 的倍数。计算机读取一个格子的东西是最快的。假如一个整数是 4 字节，它处的地址是 4 字节对齐的话，就刚好占据一个格子。但假如这个整数非 4 字节对齐，就跨越了格子边界，占据了两个格子，计算机就需要读取两个格子的信息，再将这个整数的值拼出来，这样读取就会慢了。假如机器是 64 位（也就是 8 字节），这是它就用 8 字节来划分成格子，需要 8 字节对齐。

C/C++ 编译器编译代码时，也会尽量使得数据字节对齐，比如下面结构：

**struct** **Test**{

**int** a;

**double** b;

**int** c;

**double** d;};

编译器为使得数据不跨越格子边界，整个结构在 64 位机上占据 32 个字节。但若稍微调整一下：

**struct** **Test**{

**int** a;

**int** c;

**double** b;

**double** d;};

这样交换一些数据定义顺序，整个结构在 64 位机上占据 24 个字节，比原来节省 8 个字节。

文件格式的数据最终需要载入到内存中读取，为加快读取速度，设计的时候应考虑字节对齐。比如当采用 4 对齐时，当写入字符串只有 31 个字符，可以在最后面再写入一个字节 0，这样使得下一个数字是 4 字节对齐。

分析已有的文件格式时，也需要注意字节对齐。很多旧文件格式，就经常会在字符串后面填充一些 0 字节的。有些文件格式，甚至可以调整字节对齐方式。在文件头中有个对齐字段，指定读取写入时对齐字节数。可以是 1 字节对齐（1 字节对齐其实就是不对齐），有时是 2 字节对齐，有时是 4 字节对齐。

## **回写和流写**

「回写」是指数据写入之后，可以回头再修改。比如分区数据最通常以 tag + length 开头，但最开始不知道最终的分区数据长度。这样当写入分区头的 length 字段时，就只能先随意写些临时值。当写完最后的分区数据，知道数据长度了，再回头修改长度。

但有些情况下，数据写入之后就不能回头修改了。如将数据写入网络当中，不可能回头修改网络上的数据。这种一直写，不能回头修改的姑且就叫「流写」。「有些约束条件下，二进制文件格式就需要支持「流写」，比如在网络一端生成数据，在网络另一边读取数据。这种情况下，可以将 “文件头 + 分区数据” 稍微地调整一下变成 “分区数据 + 文件尾”。当按顺序写完所有分区数据，也就知道文件的整体信息，就可以依次写入文件尾的各字段。

另外分区格式，就不能采用 tag + length 方式了，长度不太可能预先知道。这种情况下，，可以在分区开始时先写入一个特殊的开始符号，分区结束之后再重复这个特殊符号。读取的时候，遇到这个特殊符号，表示分区要开始了，再次遇到这个符号，就表示分区结束了。

比如 http 的 multipart/form-data 的 post 方式，就使用特殊的符号标记数据的开始和结束。

------WebKitFormBoundaryZL8FggWNCK9cO6Bi

Content-Disposition: form-data; name="name"

Adam

------WebKitFormBoundaryZL8FggWNCK9cO6Bi

Content-Disposition: form-data; name="password"

HelloWorld

------WebKitFormBoundaryZL8FggWNCK9cO6Bi

## **代码的坑**

上面已经讨论了二进制设计中的常见问题，这些讨论只针对普遍情况，更详细具体的二进制格式需要根据用途来设计。二进制读写的代码通常会使用 C 或 C++ 来编写，最后稍微提两个常见的坑。

### **坑 1**

读写二进制时，应该先将读写字节的代码封装起来，在 C++ 中可以封装成一个类。有些人使用模板来设计读写接口：

**class** **ByteWriter**{**public:**

**template** **<typename** T**>**

**void** write(T val);};

**class** **ByteReader**{**public:**

**template** **<typename** T**>**

T read();};

之后就使用：

writer.write(header.majorVersion);

writer.write(header.minorVersion);

这种模板接口看似很统一，其实是不好的。二进制格式的读写要严格控制字节数目，应该从接口当中就看出读写了多少个字节，不然很容易出问题。比如将 majorVersion 的类型从 uint16\_t 修改成 uint8\_t，就会出问题了。这种读写接口，宁愿笨一点，麻烦一点。写成：

**class** **ByteWriter**{**public:**

**void** writeUI16(**uint16\_t** val);

**void** **writeUI32**(**uint32\_t** val);

**void** **writeBytes**(**void\*** bytes, size\_t val);};

**class** **ByteReader**{**public:**

**uint16\_t** readUInt16();

**uint32\_t** **readUInt32**();

**void** **readBytes**(**void\*** dest, size\_t len);};

writer.writeUI16(header.majorVersion);

writer.writeUI16(header.minorVersion);

ByteWriter 和 ByteReader 内部实现应该考虑到字节顺序。

### **坑 2**

读写二进制数据，千万不要为了方便而将一个结构整体写入，比如：

**struct** **Point**{

**int16\_t** x;

**int16\_t** y;};

Point pt;

writer.writeBytes(**&**pt, **sizeof**(pt));

就算不考虑字节顺序，这种代码也是很不好的。一方面 Point 中修改了字节的顺序，或者添加了字节，甚至源码完全不变仅仅是换了机器编译（比如从 32 位机器换到 64 位机器），这样的代码都有可能会出问题。二进制的读写应拆分成一个个字段分别读写。比如：

**inline** **void** **writePoint**(ByteWriter**&** writer, **const** Point**&** pt){

writer.writeI16(pt.x);

writer.writeI16(pt.y);}