介绍

当前版本 v1.1.2

PAG（Portable Animated Graphics）是一种针对高性能渲染场景而设计的专用动效文件格式，能够以极高的压缩比存储矢量图形，文本，位图，以及序列帧等动效元素，并充分利用各平台的硬件加速能力快速解码，高速渲染到各种尺寸的屏幕。PAG 文件格式的主要特点如下：

- **屏幕渲染** 主要针对屏幕渲染的场景而设计，而非用于多种动效创作工具之间的可编辑的动效内容交换，所以优化的重点方向是渲染性能和文件大小。
- **可扩展性** 基本数据结构基于标签数据块描述，可做到不断增加新的动效特性支持的同时，持续保证对旧文件格式的向后兼容性。
- **高压缩率** 纯二进制数据结构，采用极高压缩率的动态比特位存储，以及相似区块集中压缩等技术，对于相同动效内容的存储平均可以做到其他格式的 10% ~ 50% 文件大小。
- **文件独立** 单个文件可以集成任意资源，如矢量，图片，文本，序列帧，甚至音视频，单文件交付能力可以实现更加简洁的工作流。
- **高速渲染** 文件格式简单，不含有任何字符串匹配过程，二进制流相比文本类配置文件的解码速度有显著优势。编码时优化，提前转换好针对渲染的场景需要的直接数据，动效内容能够更快渲染上屏。

PAG 文件的后缀为 .pag

第一章：基础数据类型

本章节介绍基础的数据类型以及由此组成的更复杂的数据结构。在 PAG 文件格式中的所有其他数据结构都由这些基本类型组成。

整型和字节顺序

PAG 文件格式使用 8 位，16 位，32 位，64 位有符号和无符号的整数类型。所有的整数数值都以二进制的形式存储在 PAG 文件中。PAG 的字节存储使用小端字节顺序：最低位的字节存储在最低的内存地址，高位的字节后面存储。所有的整数类型都是字节对齐。一个整数值的第一个比特必须被存储在 PAG 文件中的字节的第一个比特。

整型表

类型	备注
Int8	8 位整数
Int16	16 位整数
Int32	32 位整数
Int64	64 位整数
Uint8	无符号 8 位整数
Uint16	无符号 16 位整数
Uint32	无符号 32 位整数
Uint64	无符号 64 位整数

布尔类型

PAG 文件中使用一个 byte 位标识布尔类型。

布尔类类型表

类型	备注
Bool	0：false， 1： true

浮点数类型

PAG 文件支持使用 IEEE 754 标准的单精度浮点数类型。

浮点数类型表

类型	备注
Float	单精度 (32 位)IEEE 标准 754

数组

对于连续存储的相同数据类型，我们在数据类型后面加 **[n]** 符号来表示，其中 **n** 表示数组长度。例如: **Uint8[10]** 表示 **Uint8** 类型的数组，长度为 10。若连续加两个中括号，如 **Int8[n][m]**, 则表示二维数组，第一维数组长度 **m**, 其中每个值类型是又是一个 **Int8[n]** 数组。

编码整型

PAG 支持可变字节长度的编码整数。支持四种编码整型。

编码整型表

类型	备注
EncodeInt32	可变长度编码的 32 位整数
EncodeUint32	可变长度编码的 32 位无符号整数
EncodeInt64	可变长度编码的 64 位整数
EncodeUint64	可变长度编码的 64 位无符号整数

可变长度编码是以字节作为最小储存单位，以字节的前七位 bit 存储数值，第八位来标识后面是否还有数值。如果第 8 位 bit 是 0，那么表明数值已经读完。如果第 8 位 bit 是 1，那么再向下读取一个字节，直到把长度都读完为止（32 位或是 64 位）。

以下为无符号 32 位编码整型的解析为例：

```
uint32_t ByteBuffer::readEncodedUint32() {
    static const uint32_t valueMask = 127;
    static const uint8_t hasNext = 128;
    uint32_t value = 0;
    uint32_t byte = 0;
    for (int i = 0; i < 32; i += 7) {
        if (_position >= _length) {
            Throw(context, "End of file was encountered.");
            break;
        }
        byte = bytes[_position++];
        value |= (byte & valueMask) << i;
        if ((byte & hasNext) == 0) {
            break;
        }
    }
    return value;
}
```

有符号 32 位编码整型，则是先读取无符号 32 位编码整型的值， 再来判断高位标识位。

以下为有符号 32 位编码整型的解析：

```
int32_t ByteBuffer::readEncodedInt32() {
    auto data = readEncodedUint32();
    int32_t value = data >> 1;
    return (data & 1) > 0 ? -value : value;
}
```

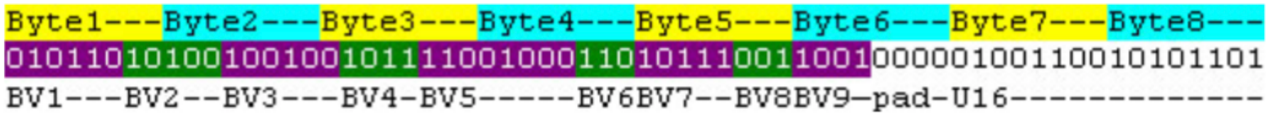
Bit 类型

Bit 类型值是可变长的比特字段，可以表示两种类型数字：

1. 无符号整数
2. 有符号整数

Bit 类型值没有被字节对齐。其他类型的值（如 Uint8 和 Uinit16）总是字节对齐。如果一个字节对齐的类型紧跟在 Bit 类型后面，那么在最后一个字节中，除了 Bit 类型值外，多出的其他比特位使用零补位。

下面的例子是一个 64 位的数据流。64 位表示不同位长的 9 个值，最后一个是 Uint16 的值。



比特流的第一个数值是一个 6 位的值（BV1），其次是一个 5 位值（BV2），BV2 这个数值跨越了 Byte1 与 Byte2 两个字节。BV3 这个数值跨了 Byte2 与 Byte3 两个字节。BV4 全部在 Byte3 字节内。BV9 之后是一个字节对齐的数值，所以最后一个 Byte 内的其余 4 个比特位使用零补齐。

Bit 值类型

类型	备注
SB[nBits]	整数（nBits 为指定位数值）
UB[nBits]	无符号整数（nBits 为指定位数值）

Bit 类型使用

Bit 数据类型一般使用所需要的最小比特位存储。大多数 Bit 类型字段使用固定的比特数。一般对于一组 Bit 类型数据，会计算出这组数据最小需要多少比特位来存储。这个最小比特位的值会存储在另一个数据结构里面。在这个最小比特位的范围类，多出的比特位，高位补零。

连续数据编码

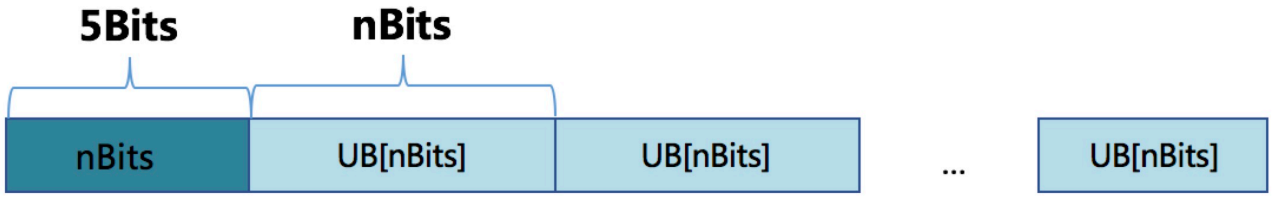
针对一组相同类型的连续数据，我们使用连续数据编码方法存储，以达到数据存储位最少。

连续无符号整数编码

对于连续无符号整数的存储，我们采用在连续无符号整型数据的前面加入头部区域，头部区域数据用来标识连续无符号整形数据存储的比特位数，后面连续无符号整形数据均按照此比特位数进行存储。

以连续的 32 位无符号整数的存储为例，头部区域为 5 个比特（32 位无符号整数最大占用 4 个字节，32 个比特，5 个比特位可以表示的数值区间为 0～31，去除 0 这种情况，预置加 1 可以表示数值区间为 1～32），计算出连续无符号整形数据的最大值和最小值，计算出两个数值存储需要占用的比特位数，取两者的最大值存储到头部区域中，连续无符号整形数据区域按照这个数值依次存储无符号整形数据。

其结构如下：



连续有符号整数编码

连续的有符号整数的存储与无符号整型存储方式一致，区别是无符号整数读取出的 nBits 位数据，全都表示数值内容，而有符号整型读取的 nBits 位之后，第一个 bit 位表示符号位（0 正数，1 负数），后连续的 nBits-1 位表示数值内容。

单个有符号整型读取过程可以先按照无符号整型读取之后，再分别转换为有符号整型：

```
int32_t ByteBuffer::readBits(uint8_t numBits) {
    auto value = readUBits(numBits);
    value <<= (32 - numBits);
    auto data = static_cast<int32_t>(value);
    return data >> (32 - numBits);
}
```

连续浮点数编码

对于连续的浮点数据，通常情况下我们保留原始精度，按照 IEEE 754 标准连续存储，不进行特殊压缩。但是对于表示某些特殊类别的，允许精度损失的连续浮点数据，编码的方式是通过 **浮点数 / 精度** 的方式将浮点数有损转换为整形数据，然后按照上面连续整形数据的方式进行编码。目前在 PAG 文件内表示以下类别的浮点数据可以按照对应精度转换为整型存储：

类型	精度	备注
SPATIAL_PRECISION	0.05	当浮点数表示空间上的坐标点时的精度
BEZIER_PRECISION	0.005	当浮点数表示塞尔曲线缓动参数的精度
GRADIENT_PRECISION	0.00002	当浮点数表示渐变插值位置参数的精度

具体到解码的过程中，解析出的整形数据再和精度相乘的到具体的浮点型数据。

连续布尔类型编码

在计算机存储方式中，Bool 类型数据占用 1 个字节，8 比特位，但有效的数据位只有 1 位，冗余存储数据占用 7 比特位。我们对于连续的的 Bool 类型数据，只使用一个比特位来标识，从而减少冗余数据存储。

Time

PAG 文件内 Time 统一使用 Int64 来描述。为了提高渲染时缓存的效率，最小的时间单位是 1 帧，帧数除以帧率可以转换为以秒为单位的外部时间。但是在存储文件时，所有的时间值都是按照 **EncodeUint64** 进行存储，而不是 EncodeInt64。因为在描述动效时，绝大部分情况时间都是正数，只有渲染计算过程中才会出现负数时间。而用无符号整型存储可以比有符号少占 1 位空间，因此在存储时间到文件时统一采用了无符号整型的方式。另外，即使小概率遇到了负数的时间，按照无符号存储和读取，也能正常还原出负数时间，区别只是负数情况转为无符号整数是一个巨大的数字，使用编码整型这种压缩方式可能会多占据几个字节的空间。但总体上出现负数时间存储到文件的概率非常低。

ID

PAG 文件内的 ID 统一使用 Uint32 来描述，存储时使用 EncodeUint32。通常 Composition， Layer 和 Mask 会具有 ID 类属性。

Enum

PAG 文件内使用 Uint8 来存储枚举类型。具体的枚举类型参考第二章：枚举

String

字符串类型使用 null 字符来标识结束。

stirng 类型

字段	类型	备注
String	Uint8[0~n]	非空字符数组
StringEnd	Uint8	标识结束，始终为 0

Point

Point 用来记录 x 轴，y 轴的位置。

Point 类型

字段	类型	备注
x	Float	x 轴坐标
y	Float	y 轴坐标

Ratio

Ratio 用来存储比率。

Ratio 类型

字段	类型	备注
numerator	EncodedInt32	分子
denominator	EncodedUint32	分母

Color

Color 代表一种颜色值，通常由 24 bit 位的红，绿，蓝三种颜色组成。

Color 类型

字段	类型	备注
Red	Uint8	红色值（0 ~ 255）
Green	Uint8	绿色值（0 ~ 255）
Blue	Uint8	蓝色值（0 ~ 255）

FontData

FontData 标识字体

FontData 类型

字段	类型	备注
fontFamily	String	字体描述
fontStyle	String	字体样式

AlphaStop

AlphaStop 描述透明度的渐变信息

AlphaStop 类型

字段	类型	备注
position	Uint16	开始点，浮点数类型使用 Uint16 存储，真实值需要乘以精度 GRADIENT_PRECISION，Uint16() * 0.00002f
midpoint	Uint16	中间点，浮点数类型使用 Uint16 存储，真实值需要乘以精度 GRADIENT_PRECISION，Uint16() * 0.00002f
opacity	Uint8	透明度（0 ~ 255）

ColorStop

ColorStop 描述颜色的渐变信息

AlphaStop 类型

字段	类型	备注
position	Uint16	开始点， 浮点数类型使用 Uint16 存储，真实值需要乘以精度 GRADIENT_PRECISION，Uint16() * 0.00002f
midpoint	Uint16	中间点，浮点数类型使用 Uint16 存储，真实值需要乘以精度 GRADIENT_PRECISION，Uint16() * 0.00002f
color	Color	颜色值

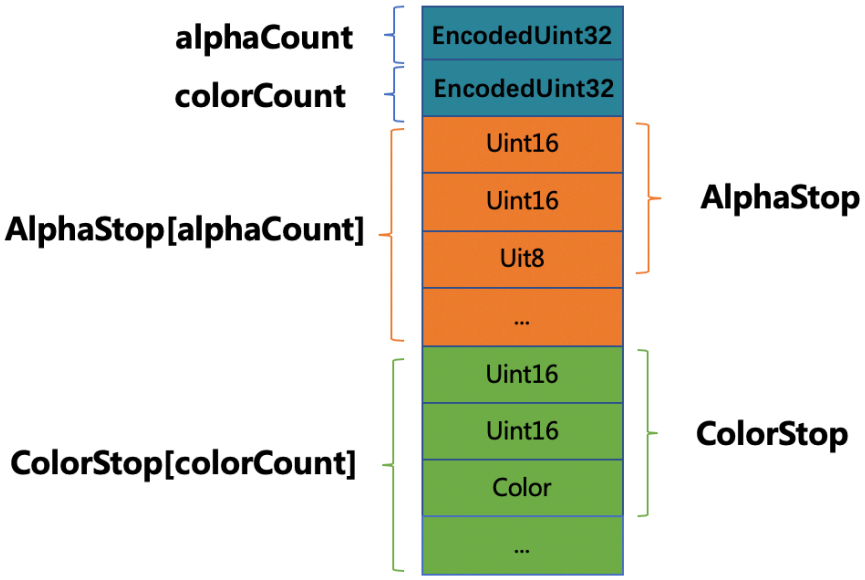
GradientColor

GradientColor 用来描述颜色与透明度的渐变信息。

GradientColor 类型

字段	类型	备注
alphaCount	EncodedUint32	透明度渐变信息数组长度，一个透明度渐变信息会包含，开始点：postion，中间点: midpostion，以及透明度值 opacity
colorCount	EncodedUint32	颜色渐变信息数组长度，一个颜色渐变信息包含，开始点：postion，中间点: midpostion，以及颜色值 color
alphaStopList	AlphaStop[alphaCount]	连续 alphaCount 个 AlphaStop
colorStopList	ColorStop[colorCount]	连续 colorCount 个 ColorStop

其对应的存储结构如下图：



TextDocument

TextDocument 文本信息，包含：文本，字体，大小，颜色等基础信息。

TextDocument 类型

字段	字段类型	备注
applyFillFlag	UB[1]	是否应用填充标识

applyStrokeFlag	UB[1]	是否应用描边标识
boxTextFlag	UB[1]	
fauxBoldFlag	UB[1]	是否粗体标识
fauxItalicFlag	UB[1]	是否斜体标识
strokeOverFillFlag	UB[1]	
baselineShiftFlag	UB[1]	
firstBaseLineFlag	UB[1]	
boxTextPosFlag	UB[1]	
boxTextSizeFlag	UB[1]	
fillColorFlag	UB[1]	是否有填充颜色信息
fontSizeFlag	UB[1]	是否有字体大小
strokeColorFlag	UB[1]	是否有描边颜色
strokeWidthFlag	UB[1]	是否有描边宽度
textFlag	UB[1]	是否有文本
justificationFlag	UB[1]	是否有对齐方式信息
leadingFlag	UB[1]	
trackingFlag	UB[1]	
hasFontDataFlag	UB[1]	是否包含字体信息
	UB[5]	全部为 0，字节对齐
baselineShift	Float	if baselineShiftFlag == 1
firstBaseLine	Float	if firstBaseLineFlag == 1
boxTextPosFlag	Point	if boxTextPosFlag == 1
boxTextSizeFlag	Point	if boxTextSizeFlag == 1
fillColor	Color	if fillColorFlag == 1
fontSize	Float	if fontSizeFlag == 1
strokeColor	Color	if strokeColorFlag == 1
strokeWidth	Float	if strokeWidthFlag == 1
text	String	if textFlag == 1
justificationFlag	Uint8	if justificationFlag == 1
leadingFlag	Float	if leadingFlag == 1
trackingFlag	Float	if trackingFlag == 1
fontID	EncodedUint32	if hasFontDataFlag == 1

Path

Path 用来标识路径等信息，主要的信息包含：动作列表与坐标列表。

PathVerb 类型

字段	类型	属性值	坐标数据	备注
Close	UB[3]	0	不需要	闭合当前的路径到路径起点
Move	UB[3]	1	**Point**	移动坐标点到指定位置，**Point** 表示要移动到的目标点
Line	UB[3]	2	**Point**	绘制一条直线，**Point** 表示要画直线到的目标点
HLine	UB[3]	3	**Float**	绘制一条水平直线，X 轴移动，Y 轴不变为上一次值，**Float** 数据表示 X 轴移动的目标位置
VLine	UB[3]	4	**Float**	绘制一条垂直直线，Y 轴移动，X 轴不变为上一次值，**Float** 数据表示 Y 轴移动的目标位置
Curve01	UB[3]	5	**Point**, **Point**	绘制一条三次贝塞尔曲线，第一个控制点跟上个动作的结束点值相同，两个 **Point** 分别表示第二个控制点和结束点
Curve10	UB[3]	6	**Point**, **Point**	绘制一条三次贝塞尔曲线，两个 **Point** 分别表示第一个和第二个控制点，结束点跟第二个控制点相同
Curve11	UB[3]	7	**Point**, **Point**, **Point**	绘制一条三次贝塞尔曲线，三个 **Point** 依次表示第一个控制点，第二个控制点，和结束点

Path 类型

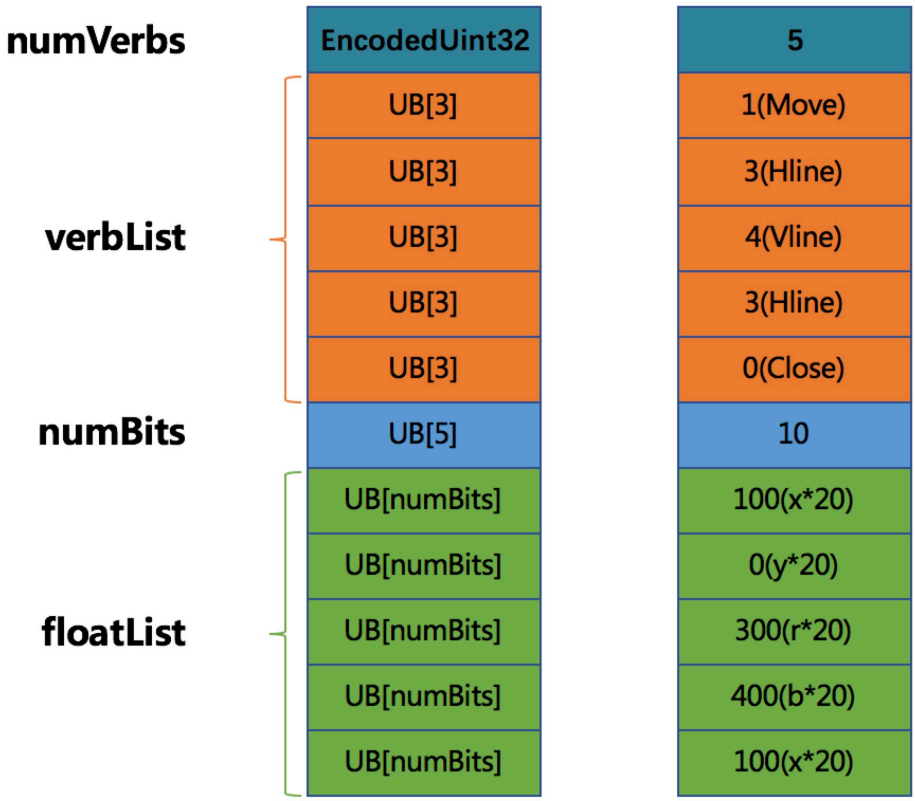
字段	类型	备注
numVerbs	EncodedUint32	动作列表的长度
verbList	UB[3][numVerbs]	动作列表数组，长度为 numVerbs，数组的每个值由一个 UB[3] 来标识 PathVerb 中的一个枚举值
numBits	UB[5]	表示接下来 floatList 里每个值所占的 bit 位数
floatList	SB[numBits][floatNum]	用于提供坐标点的浮点值数组，长度为 floatNum，数组的每个值由一个 SB[numBits] 存储

floatNum 的计算可以参考 PathVerb 的表格中 **坐标数据** 一列，其中一个 **Point** 可以等价为两个连续的 **Float**，分别表示 **Point** 的 **x** 和 **y** 坐标值。根据 verbList 的每个动作类型需要的 **Float** 个数，累加得到一共需要的 **Float** 数据总个数，即为 **floatNum** 的值。floatList 数组的每个值，需要先读取一个 **SB[numBits] 的值**，然后乘以 **SPATIAL_PRECISION** 最后得到一个 **Float** 值。

例如一个 **(x: 5, y: 0, r: 15, b: 20)** 的矩形, 用一个 Path 的数据结构描述如下：

- 执行 Move 到 (5, 0) 点， 需要记录下两个 Float 值：5，0
- 执行 HLine 到 (15, 0) 点，只需要记录一个 Float 值： 15
- 执行 VLine 到 (15, 20) 点，只需要记录一个 Float 值：20
- 执行 HLine 到 (5, 20) 点，只需要记录一个 Float 值：5
- 执行 Close 闭合矩形，回到起点 (5, 0)，不需要记录任何 Float 值。

一共需要记录 **5** 个动作：Move，HLine，VLine，HLine，Close，和 **5** 个 **Float** 坐标数据：5，0，15，20，5。对应的 Path 存储结构如下图：



floatList 中每个 Float 值的存储都是先乘以 **20 (1 / SPATIAL_PRECISION)**, 转换为整型后再存储为 **SB[numBits]**。其中 **numBits** 是根据坐标数据中存储的最大值 **400** 计算得出，**400** 的二进制表示为 **110010000** 需要 9 位，加上符号位最短一共需要 **10** 位，最终得出 **numBits** 为 **10** 长度时足够放下 **floatList** 中的每个数据。

字节流 ByteData

ByteData 标识了字节流的长度以及内容。

ByteData 类型

字段	类型	备注
length	EncodedUint32	字节长度
data	Byte[length]	读取length个字节

BitmapRect

位图信息。

BitmapRect 类型

字段	字段类型	备注
x	EncodedInt32	
y	EncodedInt32	
fileBytes	ByteData	图片数据

VideoFrame

视频帧信息。

VideoFrame 类型

字段	字段类型	备注
frame	Time	
fileBytes	ByteData	视频帧数据，字节流不包含 (0, 0, 0, 1) 四字节的 Start Code

第二章：Enum 枚举

本章主要介绍 PAG 文件内使用到的各个枚举类型以及其具体含义。

BlendMode

BlendMode 类型

类型	值	备注
Normal	0	
Multiply	1	
Screen	2	
Overlay	3	
Darken	4	
Lighten	5	
ColorDodge	6	
ColorBurn	7	
HardLight	8	
SoftLight	9	
Difference	10	
Exclusion	11	
Hue	12	
Saturation	13	
Color	14	
Luminosity	15	
Add	16	
DestinationIn	21	
DestinationOut	22	
DestinationATop	23	
SourceIn	24	
SourceOut	25	
Xor	26	

TrackMatteType

类型	值	备注
None	0	
Alpha	1	
AlphaInverted	2	
Luma	3	
LumaInverted	4	

MaskMode

遮罩蒙层的覆盖类型

MaskMode 类型

类型	值	备注
None	0	
Add	1	
Subtract	2	
Intersect	3	
Lighten	4	
Darken	5	
Difference	6	
Accum	7	

PolyStarType

PolyStarType 类型

类型	值	备注
Star	0	
Polygon	1	

CompositeOrder

CompositeOrder 类型

类型	值	备注
BelowPreviousInSameGroup	0	
AbovePreviousInSameGroup	1	

FillRule

FillRule 类型

类型	值	备注
NonZeroWinding	0	
EvenOdd	1	

LineCap

LineCap 类型

类型	值	备注
Butt	0	
Round	1	
Square	2	

LineJoin

LineJoin 类型

类型	值	备注
Miter	0	
Round	1	
Bevel	2	

GradientFillType

GradientFillType 类型

类型	值	备注
Linear	0	
Radial	1	

MergePathsMode

MergePathsMode 类型

类型	值	备注
Merge	0	
Add	1	
Subtract	2	
Intersect	3	
ExcludeIntersections	4	

TrimPathsType

TrimPathsType 类型

类型	值	备注
Simultaneously	0	
Individually	1	

ParagraphJustification

ParagraphJustification 类型

类型	值	备注
LeftJustify	0	
CenterJustify	1	
RightJustify	2	
FullJustifyLastLineLeft	3	
FullJustifyLastLineRight	4	
FullJustifyLastLineCenter	5	
FullJustifyLastLineFull	6	

第三章：PAG 整体结构

本章主要介绍一个 PAG 文件的结构，以及各个元素概要。

PAG 文件结构

PAG 文件主要有 FileHeader 以及 TagBlock 组成。TagBlock 表示一个数据块，由Tag列表组成。所有的 Tag 都有相同的结构，所以如果在解析一个 PAG 文件的时候遇见不理解的可以直接跳过当前的 Tag。

PAG 文件结构如图：

FileHeader

所有的 PAG 文件，文件开头的部分都是如下结构。类型字段可以参考第一章中的定义。

FileHeader 结构

字段	类型	备注
签名	Uint8	签名字节，总是'P'
签名	Uint8	签名字节，总是'A'
签名	Uint8	签名字节，总是'G'
版本号	Uint8	文件版本号，例如：0x04 表示 PAG 版本 4
文件长度	Uint32	文件长度 (整个文件，包含 FileHeader 长度)
压缩方式	Int8	标识文件的压缩方式，预留

PAG 文件头部都是以'P' 'A' 'G' 这三个字符开始。

然后记录的是文件的版本号，注意，文件的版本号是数值，不是字符。例如版本 4 存储的是 Ox04，而不是 ASCII 字符 '4'（Ox34）。当前文件版本号为：**1**。

文件长度是 PAG 文件的总长度，包含 FileHeader 部分。

压缩方式记录 PAG 文件的内部压缩方法，现在还没开始使用，预留。

TagBlock 结构

TagBlock 表示一个数据块，由Tag列表组成。

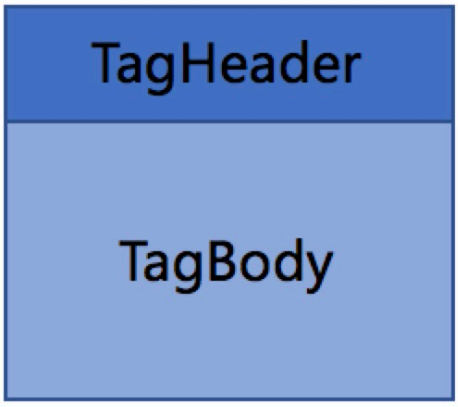
TagBlock 的结构如图：

其中 TagEnd 是一个特殊的 Tag, 通常用于标识本层次的循环结束，没有更多的 Tag 结构需要读取。某些特定的 Tag 内部会自行定义是否还包含有其他子 Tag 列表，同样也会在结尾使用 TagEnd 标识子 Tag 列表循环结束，没有更多的 Tag 需要读取。

Tag 结构

Tag 主要由两部分组成：**TagHeader** 和 **TagBody**。

Tag 的结构如下图：



TagHeader

TagHeader 记录了 TagCode (Tag 的类型 ID) 与 TagBody 的字节流长度。由于 TagBody 有可能非常短也可能非常长，出于压缩率考虑，我们将 TagHeader 又具体分为 short 和 long 两种类型的格式进行存储。short 型 TagHeader 用来记录最大 62 字节的 TagBody 数据。long 型 TagHeader，使用 32 位的无符号整型记录 TagBody 长度，可以最大存储 4G 的数据。

TagHeader(short) 类型

字段	类型	备注
TagCodeAndLength	Uint16	前 10 bit 为 TagCode，后 6 bit 为 TagBody 的长度

注意： TagCodeAndLength 字段是一次读取两个字节，而不是一个 10 bit，再加一个 6 bit。PAG 文件遵守小端字节序存储，所以

这两种存储方式是有区别的。

如果 TagBody 是 63 字节或是更长，它被存储在一个 long 型 TagHeader 里。long 型 TagHeader 包含了一个 short 型 TagHeader 的结构, 其中这个 short 型 TagHeader 记录的 TagBody 长度是一个固定值：0x3f（63），紧跟着是一个 32 字节的无符号整型表示 TagBody 真正的长度。

TagHeader(long) 类型

字段	类型	备注
TagCodeAndLength	Uint16	前 10 bit 作为 TaCode，后 6 bit 固定为 0x3f 来标识这个 TagHeader 为 long 型
Length	Uint32	TagBody 的实际长度

TagBody

TagBody 只定义了一个字节流的区块，具体解析内容的规则，是根据不同的 TagCode 类别来自行定义的。TagBody 也允许不存在，例如之前提到的特殊 TagEnd，从它的 TagHeader 读取出来的 TagBody 长度就是为 0。由于 TagBody 内部的字节流是可以完全自定义的，因此也可以定义为继续包含子 Tag 列表，即存在一个 Tag 中还可以包含一个或是多个子 Tag 的情况，从而实现嵌套。后面的章节将会详细讲解不同的 TagCode 类别对应的 TagBody 具体是如何存储的，总体上 TagBody 内部存储的就是一个属性列表的具体数据，存储方式可以划分为两大类：

- 如果 TagBody 内描述的属性列表，从种类到数量都是固定的，那么 TagBody 内部的结构会直接使用第一章节提供的那些基础数据类型进行排列组合定义。类似之前 **Path** 数据结构的定义方式，利用已有的数据结构排列组合出新的数据结构。解码时将 TagBody 直接交给对应的类别的解码模块即可。对于这类的 Tag， 我们在后面的章节会直接用表格列出其 TagBody 的数据结构。
- 如果 TagBody 内描述的属性列表，从种类到数量都是不确定的，就会需要大量额外的标识符字段，这时候用上述的的方式去定义数据结构将会浪费大量的文件空间。对这类的 Tag 我们会使用 **AttributeBlock** 这种动态数据结构来描述整个 TagBody 的内容。后面章节将会详细介绍 **AttributeBlock**。

第四章：AttributeBlock

前面章节中介绍了 Tag 的数据结构，在具体讲解每种 Tag 类别对应的 TagBody 是如何存储的之前，我们先介绍一个新的动态数据结构：**AttributeBlock**。当要存储的属性列表，从种类到数量都是不确定的，会需要额外大量的状态标识符字段，这种动态数据结构主要用于最大化压缩这些标识符，并根据标识符动态匹配不同的解码模块。

Value 和 Property 属性

AttributeBlock 通常就是用来描述一组属性列表的数据具体如何存储的，这里我们以 **Mask** 遮罩的数据结构为例，它需要存储的属性列表如下：

字段	类型	备注
id	Uint32	遮罩 ID
inverted	Bool	标识遮罩是否反转
maskMode	Uint8	遮罩的混合模式
maskPath	Property<Path>	遮罩的矢量路径
maskOpacity	Property<Uint8>	遮罩的透明度
maskExpansion	Property<Float>	遮罩的边缘扩展参数

到目前为止，我们介绍过的都是基础数据类型，可以看到 Mask 的属性列表里，前 3 个属性的类型也都是这种数据类型。这些属性的特点是只包含一个数据值，我可以把这类属性统称为 **Value** 属性。这里我们再引入另一种叫时间轴属性的类型。可以用

Property<T> 来泛指任意一种时间轴属性， **T** 可以为 Bool, String, Int8, Point, Path... 等任意一种基础数据类型。如果我们将 **T** 替换为具体类型，如： **Property<Point>**，那它就表示 **Point** 类的时间属性。

Value 属性跟 Property 属性的主要区别在于：Value 属性只包含一个数据值，而 Property 属性通常包含一整个时间轴的多个数据值，根据关键帧数量不同，它可以包含 1 个或多个关键点的数据值。可以简单理解为 Property 一系列 Value 在时间轴上的集合。Property 属性还含有时间缓动参数和空间缓动参数，用于控制关键帧之间的数据值如何产生瞬时插值。后面的章节会详细介绍 Property 属性的存储结构，这里我们只需要了解它的时间轴概念即可。

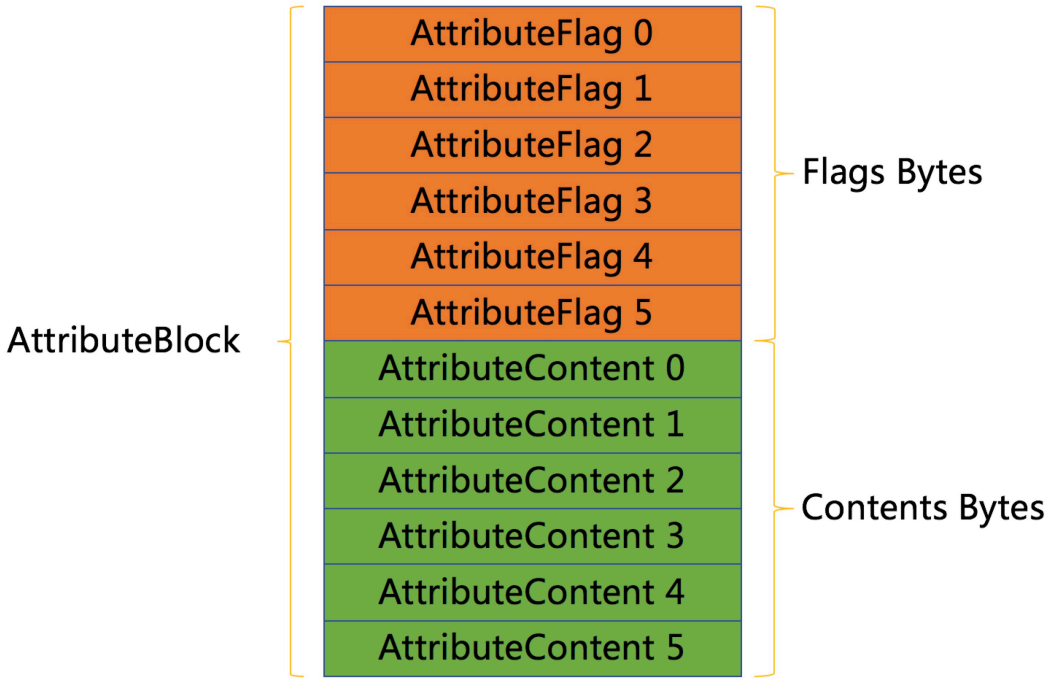
对于 Mask 的多个属性，最简单存储方式就是按每个类型完整结构都存储一遍，但显然这样是非常浪费空间的，我们观察可以发现很多存在冗余数据的情况：

- inverted 属性是一个 Bool 值，需要占 1 个字节，但实际可以只使用 1 个 bit 来标识。
- maskMode 通常都是等于默认值，这种情况不需要存储实际的值，直接使用 1 个 bit 来标识等于默认值即可。
- 所有的 Property 属性：1. 如果不存在空间缓动参数，可以不存储相关数据。2. 如果不存在关键帧，也就是只有一个数据值时，可以节省整个关键帧的存储结构，只存储一个值即可。3. 如果这个值还等于默认值，可以只用 1 个 bit 来标识等于默认值即可。

根据以上特点可以发现，每个属性都有可能存在很多特殊的状态标识，充分利用好这些状态标识可以在大部分情况下显著减少文件大小。尤其是 Property 属性，要完整存储一个时间轴属性内部需要非常复杂的数据结构来记录，但是大部分情况下，Property 属性可能都并不存在关键帧，即可以退化为 Value 属性来存储，这样不必要占据的空间可以显著减少。因此我们使用 **AttributeBlock** 的数据结构来描述这种需要大量状态标识的存储场景。

AttributeBlock 结构

AttributeBlock 主要由两部分组成：由 **AttributeFlag** 列表组成的标志位区域以及由 **AttributeContent** 列表组成的内容区域。具体结构如下图：



其中每个 **AttributeFlag** 的排列顺序是跟 **AttributeContent** 的顺序一一对应的。每次给定一个属性列表，我们就将每个属性的标志位信息与数据内容信息拆分成一对的 **AttributeFlag** 和 **AttributeContent** 结构，然后按顺序排列存储。例如这里的 **AttributeFlag 0** 和 **AttributeContent 0** 即对应 Mask 属性列表的第一项 **ID** 属性。从图中可以看出，存储结构是先存储完所有的 Flag 列表再开始存储所有的 Content 列表。这样做是因为 Flag 区每项通常都是以 bit 位表示的，集中存储在一起可以避免频繁发生字节对齐，从而产生大量空间浪费。Flag 区域读取结束后会进行一次统一的字节对齐，由于字节对齐产生的额外比特位都会置为 0，再从整字节数位置开始继续存储 **AttributeContent** 区域。

注意：为了节省文件空间，属性列表的数量并不会写入文件，而是硬编码在解析代码中，因为每个特定的 **AttributeBlock** 的属性数量是固定不会变化的，解析的过程中已经可以提前知道要解析的属性数量。

AttributeFlag

通过对 Value 属性和 Property 属性特征的总结，我们抽象出如下的 **AttributeFlag** 数据结构。

AttributeFlag 类型

字段	类型	备注
exist	UB[1]	标识对应的 AttributeContent 是否存在
animatable	UB[1]	是否有关键帧信息，仅当 exist 为 1 时存在此字段
hasSpatial	UB[1]	是否有空间缓动信息，仅当 exist 和 animatable 均为 1 时存在此字段

每个属性都能解析出这样的一个描述标志位信息的数据对象，用于辅助对后续 **AttributeContent** 区域的解码。但每个属性产生的 **AttributeFlag** 具体存储长度是动态的，取值范围是 **0 ~ 3** 比特位。例如 Value 属性最多只可能占 1 个比特位，解码时只会读取 **exist** 标志位。而 Property 属性会用到最多 3 个比特位来存储标志位信息。例如当一个 Property 属性不含有关键帧且属性值等于默认值时，它只会用到 1 个比特位 （0） 来存储 **AttributeContent**。解码时读取到 **exist** 标志位为 0，就会放弃后续的标志位读取。在极限情况下，整个属性列表都不含有关键帧而且都等于默认值，最终只会占用列表数量个比特位存储所有的 **exist** 标志为 0，而整个 Content 区域都为空。这样可以显著降低需要记录的文件大小。

AttributeType

前面提到 **AttributeFlag** 的实际存储大小可能是 **0 ~ 3** 位，并且 Value 类型属性只会固定读取 1 比特位。所以在解码时，还需要能知道具体的属性类型，才能确定 **AttributeFlag** 的读取规则。前面我们将属性大体上分为了 Value 和 Property 类别，实际上还可以细分为多种子类别，根据属性特征进一步压缩空间。以下是 PAG 文件中使用到所有属性类别，和它们对应在 Flag 区占的位数：

AttributeType 类型

类型	Flag 区	备注
Value	固定占 1 位	普通 Value 属性
FixedValue	不占用	固定存在的 Value 属性
BitFlag	固定占 1 位	Bool 值类型的 Value 属性
SimpleProperty	占 1 ~ 2 位	简单动效属性
DiscreteProperty	占 1 ~ 2 位	离散动效属性（不存在插值）
MultiDimensionProperty	占 1 ~ 2 位	多维时间缓动动效属性
SpatialProperty	占 1 ~ 3 位	空间缓动动效属性
Custom	固定占 1 位	扩展类型，数据读取规则自定义

根据不同 AttributeType 读取 AttributeFlag 的示例代码如下：

```
AttributeFlag ReadAttributeFlag(ByteBuffer* byteArray, const AttributeBase* config) {
    AttributeFlag flag = {};
    auto attributeType = config->attributeType;
    if (attributeType == AttributeType::FixedValue) {
        flag.exist = true;
        return flag;
    }
    flag.exist = byteArray->readBitBoolean();
    if (!flag.exist ||
        attributeType == AttributeType::Value ||
        attributeType == AttributeType::BitFlag ||
        attributeType == AttributeType::Custom) {
        return flag;
    }
    flag.animatable = byteArray->readBitBoolean();
    if (!flag.animatable || attributeType != AttributeType::SpatialProperty) {
        return flag;
    }
    flag.hasSpatial = byteArray->readBitBoolean();
    return flag;
}
```

其中可以看到，FixedValue 这种类别是不从 Flag 区读取数据的，直接返回 exist 为 true 的情况，标识这种类别属性的数据永远存在。

注意：为了节省文件空间，AttributeType 并不写入文件，而是硬编码配置在解析代码中，因为具体每个属性的 AttributeType 并不可能发生变化，在解析的过程中已经可以提前知道要解析的属性类型具体是什么。

AttributeContent

根据前面的步骤，我们已经能够解码出 AttributeFlag 对象，再结合已知的 AttributeType， 就能开始解码对应的每个 AttributeContent 的数据。以下是每种 AttributeType 对应的解析规则：

- Value : 如果 exist 为 true，则读取 AttributeContent 数据值，否则使用默认值。
- FixedValue : 忽略所有 flag，直接读取 AttributeContent 数据值。
- BitFlag : 直接使用 exist 的值作为 Bool 数据返回，不读取 AttributeContent 数据。
- SimpleProperty，DiscreteProperty，MultiDimensionProperty，SpatialProperty： 若 animatable 为 false， 直接走 Value 这种 AttributeType 的读取规则，根据 exist 判断从 AttributeContent 读取一个数据值还是使用默认值。若 animatable 为 true，走 Property 的解码流程。这些细分的不同 Property 类型以及 hasSpatial 标识位，都只在解码 Property 内部的数据结构内部才会用到。后面在详细介绍 Property 内部的解码规则。
- Custom： 之前的所有类别都可以走通用的解码模块，但是某些 AttributeContent 内容可能并不是一个属性，将类型配置为 Custom 后，可以由每个 Tag 具体定义这个 AttributeContent 应该使用什么模块解析。而 exist 标志就是用来判断是否存在对应的 AttributeContent。这样可以将自定义的数据块插入属性列表中整体存储。

注意：为了节省文件空间，每个属性的默认值并不写入文件，而是硬编码配置在解析代码中，因为具体每个属性的默认值不可能发生变化，在解析的过程中已经可以提前知道要解析的属性的默认值具体是什么。

经过上面的分析，我们已经能够完整解码出来 AttributeBlock 这种数据结构。任意给定一组属性列表，只需要列出属性排序，每个属性的基础数据类型，属性类型，以及默认值这些必须信息，即可参照之前 AttributeBlock 的规则进行解码。之后的数据结构表格中如果出现 **属性类型** 和 **默认值** 俩列，均表示需要按照 AttributeBlock 数据结构进行解析。之前的 Mask 对应的 AttributeBlock 结构表就可以定义为如下：

Mask 的 AttributeBlock 结构表

字段	数据类型	属性类型	默认值	备注
id	EncodedUint32	FixedValue	0	遮罩 ID
inverted	Bool	BitFlag	false	标识遮罩是否反转
maskMode	Uint8	Value	1	遮罩的混合模式
maskPath	Path	SimpleProperty	空 Path 对象	遮罩的矢量路径
maskOpacity	Uint8	SimpleProperty	255	遮罩的透明度
maskExpansion	Float	SimpleProperty	0	遮罩的边缘扩展参数

第五章: Property 属性

Property 是动效属性的基本单位，大部分对象上的属性都是 Property。因为使用广泛，对单个 Property 的存储结构进行优化压缩后，可以显著减少整个文件的大小。Property 一般不单独存储，一般跟其他的 Property 属性或者 Value 属性作为属性列表一起存储在 AttributeBlock 动态数据结构中，作为一个 AttributeContent。关于 AttributeBlock 的解析请参考上一个章节。所以根据之前的章节描述，当我们解析到 Property 属性对应的 AttributeContent 区域时，已经能够访问到 **数据类型**， **AttributeType**，**默认值** 这些配置参数，以及之前读取出来的 **AttributeFlag** 数据对象。通过这些辅助参数我们将开始对 Property 的内容进行解码。

Property 结构

Property 结构主要是由 **Keyframe** 结构的关键帧列表组成，列表长度可以是 0 到 多个。当 **Keyframe** 长度为 0 时（即 AttributeFlag.animatable 为 false），整个 Property 属性只包含一个有效值，所以可以退化为 Value 属性进行存储，继续根据 AttributeFlag.exist 判断是否从 AttributeContent 读取一个数据值还是使用默认值。这部分读取规则之前已经描述过，本章节的内容继续描述当 AttributeFlag.animatable 为 true 时，一个 Property 属性对应的 AttributeContent 应该如何解析。解析 Property 的结构实际上就是解析一组 Keframe 列表的结构。

Keyframe 数据结构

Property 通常包含若干个关键帧信息组成。一个关键帧包含了此帧的开始和结束时间，还有开始结束的属性值，以及标识属性值计算方法的差值器类型，时间缓动参数等。对于复杂的 Property 属性，它的关键帧还有可能包含多维度的时间缓动参数，或者额外包含空间缓动参数。我们先来看一个 **Keyframe** 需要记录那些数据字段：

KeyFrame 数据列表

字段	类型	备注
startValue	泛型值（任意一种基础数据类型）	起始值
endValue	泛型值（任意一种基础数据类型）	结束值
startTime	Time (Int64)	关键帧的起始时间值
endTime	Time (Int64)	关键帧的结束时间值
interpolationType	枚举（Uint8）	插值器类型
bezierOut	Point[dimensionality]	时间缓动参数数组（贝塞尔曲线第 1 个控制点）
bezierIn	Point[dimensionality]	时间缓动参数数组（贝塞尔曲线第 2 个控制点）
spatialOut	Point	空间缓动参数（贝塞尔曲线第 1 个控制点）
spatialIn	Point	空间缓动参数（贝塞尔曲线第 2 个控制点）

startValue 和 endValue 表示这个关键帧区间的起始值和结束值，对应的 startTime 和 endTime 表示这个关键帧区间的起始时刻和

结束时刻，所以当取值的时刻等于 startTime 时，会返回 startValue，取值的时刻等于 endTime 时会返回 endValue，而在 startTime 和 endTime 之间的时刻，取值是由 interpolationType 插值器类型来决定的。插值器类型如下：

KeyframeInterpolationType 类型

类型	值	备注
None	0	无效
Linear	1	线性插值
Bezier	2	根据时间缓动参数使用贝塞尔曲线插值
Hold	3	整个区间除了 endTime 时刻，都等于 startValue， endTime 瞬间切换到 endValue

Keyframe 压缩方式

结合以上所有数据结构，可以总结出 Keyframe 的数据字段并不是都需要完整存储的，主要有以下几种场景可以节省存储空间：

- 当插值类型不等于 **Bezier** 时可以不需要存储时间缓动参数 bezierOut 和 bezierIn。
- 当属性类型是 **DiscreteProperty** 时，表示插值器类型只可能是 **Hold**，不需要存储 **interpolationType**。这种情况通常出现在属性的基础数据类型是 Bool 值或者枚举时，它的数据是离散的，true 和 false 之间本质上不可能出现中间插值。
- 当属性类型是 **MultiDimensionProperty** 时，表示时间缓动参数是多条贝塞尔曲线组成，每条单独控制数据值的一个分量缓动，通常出现在表示缩放的时间轴属性上。具体是多少条贝塞尔曲线，是根据 startValue 和 endValue 的数据类型来确认。例如当数据类型是 Point 时间缓动参数就是 2 维数组，两条贝塞尔曲线分别控制 x，和 y 轴的独立缓动。对于大部分不是 **MultiDimensionProperty** 属性的情况，我们不需要根据基础数据类型判断维度，默认只存储一维时间缓动参数。
- 当属性类型是 **SpatialProperty** 时，表示关键帧可能存在空间缓动参数，这时候具体要根据 AttributeFlag.hasSpatial 标志来判断实际当前关键帧是否存在这部分参数。只有这种属性需要在 AttributeFlag 上用到第三个 hasSpaital 标志位。对于其他的属性类型，都不需要判断或者存储空间缓动参数 spatialOut 和 spatialIn。

除了以上各种场景可以通过判断可以节省存储空间外，我们还采用了其他的压缩策略：实际存储关键帧列表时是依次存储每个关键帧的一类数据后再集中存储下一类数据，而不是一个关键帧存储结束再存储下一个关键帧。这样做的好处是相似的数据可以集中压缩。例如 interpolationType 通常只会占用 2 个比特位，集中存储可以减少因字节对齐产生的额外空间浪费。在比如下一个关键字的 startValue 和 startTime 总等于上一个关键帧的 endValue 和 endTime，集中存储也可以跳过不需要存储的帧间重复数据。

Property 存储结构

Property 的具体存储结构如下，注意：这里仅描述在 AttributeFlag.animatable 为 true 的情况下的读取存储结构，若 AttributeFlag.animatable 为 false，参考之前的方式根据 AttributeFlag.exist 判断是否从 AttributeContent 读取一个数据值还是使用默认值即可。

Property 类型

字段	类型	备注
numFrames	EncodedUint32	Keyframe 数组的长度。
interpolationTypeList	UB[2][numFrames]	每个 Keyframe 对应的插值器类型，一共读取 numFrames 次。若属性类型是 **DiscreteProperty**，则跳过此区块。
timeList	EncodedUint64[numFrames+1]	每个 Keyframe 的起始和结束时间，后一个关键帧的起始时间等于前一个关键帧的结束时间，所以只读取 numFrames+1 次。
valueList	PropertyValueList	每个 Keyframe 的起始值和结束值，不同的基础数据类型有不同的存储方式，后面会详细介绍 PropertyValueList 的读取规则。
timeEaseNumBits	UB[5]	接下来每个时间缓动参数分量占的存储比特位数。
timeEaseList	TimeEaseValueList	时间缓动参数数组, 后面会详细介绍 TimeEaseValueList 的读取规则。
spatialFlagList	UB[numFrames * 2]	标志数组，标识接下来每个 Keyframe 是否分别包含 spatialIn 和 spatialOut 参数。
spatialEaseNumBits	UB[5]	接下来每个空间缓动参数分量占的存储比特位数。
spatialEaseList	SpatialEaseValueList	空间缓动参数数组，后面会详细介绍 SpatialEaseValueList 的读取规则。若属性类型不是 **SpatialProperty** 或 **AttributeFlag.hasSpatial** 为 false，跳过此区块。

注意：每个区块读取结束要进行一次字节对齐，跳过没读取完的剩余比特位，从下一个整数字节位置开始，再读取下一个区块。

PropertyValueList

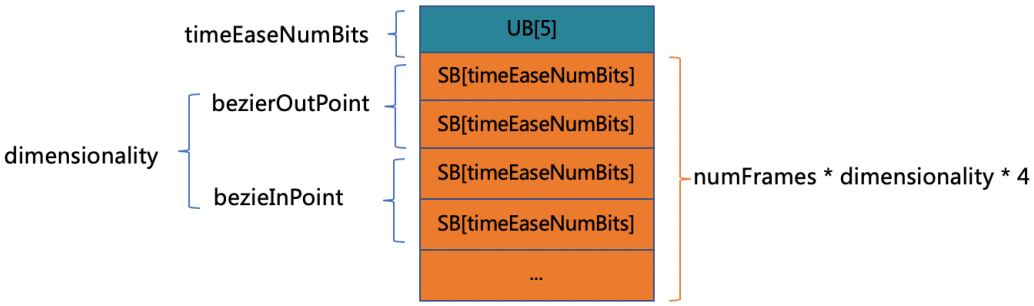
PropertyValue 区块存储了一个基础数据类型的列表，列表内容表示每个 Keyframe 的 起始值和结束值，由于后一个关键帧的起始值始终等于前一个关键帧的结束值，因此列表总长度是 numFrames + 1。这个数据列表的具体存储规则根据基础数据类型的不同而不同，具体如下表：

PropertyValue 存储方式

类型	存储方式
Float[numFrames + 1]	依次分别存储 numFrames + 1 个 Float 数据。
Bool[numFrames + 1]	依次分别存储 numFrames + 1 个 比特位，每个比特位代表一个 Bool 值。
Uint8[numFrames + 1]	按 32 位连续无符号整型进行压缩存储，先存储 UB[5] 的 numBits， 然后依次分别存储 numFrames + 1 个 UB[numBits] 数据，每个数据代表原始的 Uint8 值。
Uint32[numFrames + 1]	按 32 位连续无符号整型进行压缩存储，先存储 UB[5] 的 numBits， 然后依次分别存储 numFrames + 1 个 UB[numBits] 数据，每个数据代表原始的 Uint32 值。
Point[numFrames + 1]	通常情况下依次分别储存 numFrames + 1 个 Point 数据。若属性类型是 **SpatialProperty**, 则将每个 Point 数据的两个 Float 除以 SPATIAL_PRECISION 转为长度为 (numFrames+1)*2 的 Uint32 列表后，按照 32 为连续无符号整型进行压缩存储，具体存储规则同上。
Time[numFrames + 1]	依次存储 numFrames + 1 个 EncodedUint64 数据，每个数据代表原始的 Time 数据。
ID[numFrames + 1]	依次存储 numFrames + 1 个 EncodedUint32 数据，每个数据代表原始的 ID 数据。
Color[numFrames + 1]	依次分别存储 numFrames + 1 个 Color 数据。
Ratio[numFrames + 1]	依次分别存储 numFrames + 1 个 Ratio 数据。
String[numFrames + 1]	依次分别存储 numFrames + 1 个 String 数据。
Path[numFrames + 1]	依次分别存储 numFrames + 1 个 Path 数据。
TextDocument[numFrames + 1]	依次分别存储 numFrames + 1 个 TextDocument 数据。
GradientColor[numFrames + 1]	依次分别存储 numFrames + 1 个 GradientColor 数据。

TimeEaseValueList

TimeEaseValueList 存储结构如下图：

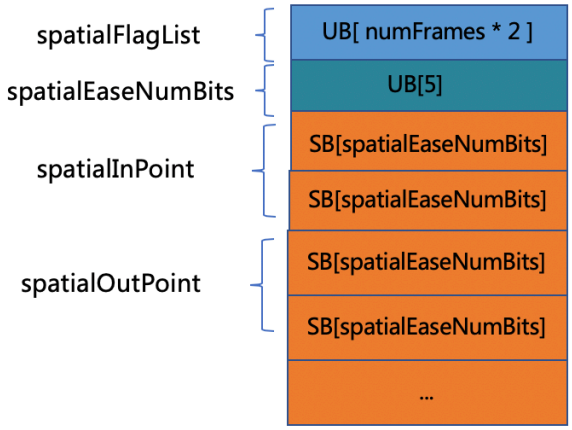


时间缓动参数的读取，除了依赖之前 **timeEaseNumBits** 之外，还依赖一个 **dimensionality** 参数。dimensionality 表示每个关键帧的 bezierIn 和 bezierOut 数组是几维的。可以从属性类型和基础数据类型分量的数量推导出来：例如当 属性类型是 **MultiDimensionProperty** 时，对于数据类型是 Point 的属性，dimensionality 就是 2 ，若属性类型不是 **MultiDimensionProperty** 属性， dimensionality 始终为 1。 timeEaseList 列表里的每一项表示时间缓动参数坐标点的一个 Float 分量，两个 Float 组成一个 Point，通常一维时间缓动属性的每个关键帧有 bezierIn 和 bezierOut 两个 Point，也就是需要 timeEaseList 里的 4 个 Float 分量依次表示。多维的情况是依次读取完本帧的所有维度时间缓动参数数组，然后再读取下一帧的数据。另外，如果当前的关键帧插值器类型不是 **Bezier**，会跳过这一帧的时间缓动参数读取流程。具体解析代码如下：

```
        int dimensionality = config.attributeType == AttributeType::MultiDimensionProperty ?
                             config.dimensionality() : 1;
        auto numBits = byteArray->readNumBits();
        for (auto& keyframe:keyframes) {
            if (keyframe->interpolationType != KeyframeInterpolationType::Bezier) {
                continue;
            }
            float x, y;
            for (int i = 0; i < dimensionality; i++) {
                x = byteArray->readBits(numBits) * BEZIER_PRECISION;
                y = byteArray->readBits(numBits) * BEZIER_PRECISION;
                keyframe->bezierOut.emplace_back(x, y);
                x = byteArray->readBits(numBits) * BEZIER_PRECISION;
                y = byteArray->readBits(numBits) * BEZIER_PRECISION;
                keyframe->bezierIn.emplace_back(x, y);
            }
        }
```

SpatialEaseValueList

SpatialEaseValueList 存储结构如下图：



注意：若属性类型不是 **SpatialProperty** 或 **AttributeFlag.hasSpatial** 为 false，此区块不需要读取。空间缓动参数的读取需要依赖之前的读取出来的 **spatialFlagList** 列表以及 **spatialEaseNumBits**。spatialFlagList 是两倍关键帧数量的长度，因为每个关键帧的空间缓动参数含有两个 Point 数据。spatialFlagList 列表里的值依次表示每个关键帧的 spatialIn 和 spatialOut 是否存在。如果不存在，使用默认值 （0，0）点。另外，读取出来的数据为整型，需要乘以 **SPATIAL_PRECISION** 转换为 Float 后作为 Point 的 x 和 y 数据分量。 具体解析代码如下：

```
        int index = 0;
        for (auto& keyframe:keyframes) {
            auto hasSpatialIn = spatialFlagList[index++];
            auto hasSpatialOut = spatialFlagList[index++];
            if (hasSpatialIn) {
                keyframe->spatialIn.x = byteArray->readBits(spatialEaseNumBits) * SPATIAL_PRECISION;
                keyframe->spatialIn.y = byteArray->readBits(spatialEaseNumBits) * SPATIAL_PRECISION;
            }
            if (hasSpatialOut) {
                keyframe->spatialOut.x = byteArray->readBits(spatialEaseNumBits) * SPATIAL_PRECISION
                keyframe->spatialOut.y = byteArray->readBits(spatialEaseNumBits) * SPATIAL_PRECISION
            }
        }
```

第六章：标签列表

PAG 使用 10 比特位来存储 TagCode，最多可以存储 1024 种 Tag. 其中现在已经使用了 52 种标签，其列表如下：

TagCode 类型

名称	数值 (十进制)	备注
End	0	Tag 结束标识
FontTables	1	字体集合，包含多个字体
VectorCompositionBlock	2	矢量组合信息
CompositionAttributes	3	组合基本属性信息
ImageTables	4	图片合集信息
LayerBlock	5	图层信息
LayerAttributes	6	图层基本属性信息
SolidColor	7	边框颜色
TextSource	8	文本信息，包含：文本，字体，大小，颜色等基础信息
TextPathOption	9	文本绘制信息，包含：绘制路径，前后左右间距等
TextMoreOption	10	文本其他信息
ImageReference	11	图片引用，指向一个图片
CompositionReference	12	组合引用，指向一个组合
Transform2D	13	2D 变换信息
MaskBlock	14	遮罩信息
ShapeGroup	15	Shape 信息
Rectangle	16	矩形信息
Ellipse	17	椭圆信息
PolyStar	18	多边星形
ShapePath	19	Shape 路径信息
Fill	20	填充规则信息
Stroke	21	描边
GradientFill	22	渐变填充
GradientStroke	23	渐变描边
MergePaths	24	合并路径
TrimPaths	25	裁剪路径
Repeater	26	中继器
RoundCorners	27	圆角
Performance	28	文件性能信息，主要用来校验 PAG 文件性能是否达标
DropShadowStyle	29	投影
BitmapCompositionBlock	45	位图序列帧

BitmapSequence	46	位图序列
ImageBytes	47	图片字节流
ImageBytes2	48	图片字节流（带缩放）
ImageBytes3	49	图片字节流（带透明通道）
VideoCompositionBlock	50	视频序列帧
VideoSequence	51	视频序列

End

End 标签标识 TAG 的结束，当解码器读取到这个标签时，标记这个 TAG 的内容都已经读取完了。如果 Tag 包含嵌套，那么遇见 End 标识需要跳出当前 Tag 读取，调到外层的 Tag 读取逻辑。

End 结构表

Type	类型	备注
End	Uint16	Tag 结束标识

FontTables

FontTables 是字体信息的合集。

FontTables 结构表

字段	类型	备注
count	EncodedUint32	字体数目
fontData	FontData[]	字体数组

VectorCompositionBlock

VectorCompositionBlock 是矢量图形的合集。里面可以包含简单的矢量图形，也可以再包含一个或是多个 VectorComposition。

VectorCompositionBlock 结构表

字段	类型	备注
id	EncodedUint32	唯一标识
TagBlock	TagBlock	TagBlock数据块，参考第三章TagBlock。 包含：CompositionAttributes(组合基本属性信息), LayerBlock(图层信息) 这两种TAG

CompositionAttributes

CompositionAttribute 存储了 Composition 基本属性信息。里面可以包含简单的矢量图形，也可以再包含一个或是多个 VectorComposition。

CompositionAttributes 结构表

字段	类型	备注
width	EncodedInt32	图层的宽
height	EncodedInt32	图层的高
duration	EncodedUint64	持续时间
frameRate	Float	帧率
backgroundColor	Color	背景颜色

ImageTables

ImageTables 是图片信息的合集。

ImageTables 结构表

字段	类型	备注
count	EncodedInt32	图片数目
images	ImageBytes[count]	图片数组

LayerBlock

LayerBlock 是图层信息的合集。

LayerBlock 结构表

字段	类型	备注
type	Uint8	Layer 类型
id	EncodedUint32	Layer 的唯一标识
TagBlock	TagBlock	TagBlock数据块，参考第三章TagBlock。 包含：LayerAttributes, LayerAttributes2, MaskBlock, Transform2D, SolidColor, TextSource, TextPathOption, TextMoreOption, CompositionReference, ImageReference等这几种TAG

LayerAttributes

LayerAttributes 是图层的属性信息。

LayerAttributesBlock 的 AttributeBlock 结构表

字段	字段类型	属性类型	默认值	备注
isActive	Bool	BitFlag	true	如果为 false，则不会被渲染
autoOrientation	Bool	BitFlag	false	自适应屏占比
parent	EncodedUint32	Value	0	Layer的ID
stretch	Ratio	Value	（1,1）	拉伸比例
startTime	Time	Value	0	开始时间
blendMode	枚举（Uint8）	Value	BlendMode::Normal	图层的混合模式
trackMatteType	枚举（Uint8）	Value	TrackMatteType::None	轨道蒙版
timeRemap	Float	SimpleProperty	0.0f	
duration	Time	FixedValue	0	时间区间

SolidColor

SolidColor 标识边框宽高以及颜色属性信息。

SolidColor 结构表

字段	字段类型	备注
solidColor	Color	颜色值
width	EncodedInt32	宽
height	EncodedInt32	高

TextSource

TextSource 文本信息，包含：文本，字体，大小，颜色等基础信息。

TextSource 的 AttributeBlock 结构表

字段	字段类型	属性类型	默认值	备注
sourceText	TextDocument	DiscreteProperty	TextDocument	

TextPathOption

TextPathOption 文本绘制信息，包含：绘制路径，前后左右间距等。

TextPathOption 的 AttributeBlock 结构表

字段	字段类型	属性类型	默认值	备注
path	EncodedUint32	Value	0	Mask ID
reversedPath	Bool	DiscreteProperty	false	
perpendicularToPath	Bool	DiscreteProperty	false	
forceAlignment	Bool	DiscreteProperty	false	
firstMargin	Float	SimpleProperty	0.0f	
lastMargin	Float	SimpleProperty	0.0f	

TextMoreOption

TextMoreOption 文本其他信息。

TextMoreOption 的 AttributeBlock 结构表

字段	字段类型	属性类型	默认值	备注
anchorPointGrouping	枚举（Uint8）	Value	ParagraphJustification::LeftJustify	
groupingAlignment	Point	MultiDimensionProperty	（0.0）	

ImageReference

ImageReference 图片引用标签，存储的是一个图片的唯一 ID，通过 ID 索引真正的图片信息。

ImageReference 的 AttributeBlock 结构表

字段	字段类型	属性类型	默认值	备注
id	EncodedUint32			

CompositionReference

CompositionReference 图层组合索引标签，存储的是一个图层组合的唯一 ID，通过 ID 索引真正的图层组合。

CompositionReference 的 AttributeBlock 结构表

字段	字段类型	属性类型	默认值	备注
id	EncodedUint32			
compositionStartTime	Time			开始时间

Transform2D

Transform2D 2D 变换信息，包含：锚点，缩放，旋转，x 轴偏移，y 轴偏移等信息。

Transform2D 的 AttributeBlock 结构表

字段	字段类型	属性类型	默认值	备注
anchorPoint	Point	Value	(0.0)	锚点
position	Point	Value	(0.0)	位置信息
xPosition	Float	Value	0.0	x 轴偏移
yPosition	Float	Value	0.0	y 轴偏移
scale	Point	Value	(0.0)	缩放
rotation	Float	Value	0.0	旋转
opacity	Uint8	Value	255	透明度（0 ~ 255）

Mask

Mask 遮罩标签。

Mask 的 AttributeBlock 结构表

字段	字段类型	属性类型	默认值	备注
id	EncodedUint32	FixedValue	0	
inverted	Bool	BitFlag	false	
maskMode	枚举（Uint8）	Value	MaskMode::Add	
maskPath	Path	SimpleProperty		
maskOpacity	Uint8	SimpleProperty	255	透明度（0 ~ 255）
maskExpansion	Float	SimpleProperty	0.0f	

ShapeGroup

ShapeGroup 投影标签。

ShapeGroup 的 AttributeBlock 结构表

字段	字段类型	属性类型	默认值	备注
blendMode	枚举（Uint8）	Value	BlendMode::Normal	
anchorPoint	Point	SpatialProperty	(0.0)	锚点
position	Point	SpatialProperty	(0.0)	位置信息
scale	Point	MultiDimensionProperty	(1, 1)	缩放
skew	Float	SimpleProperty	0.0	
skewAxis	Float	SimpleProperty	0.0	y 轴偏移
rotation	Float	SimpleProperty	0.0	旋转
opacity	Uint8	SimpleProperty	255	透明度（0 ~ 255）
TagBlock	TagBlock			TagBlock数据块，参考第三章 TagBlock。

Rectangle

矩形标签。

Rectangle 的 AttributeBlock 结构表

字段	字段类型	属性类型	默认值	备注
reversed	Bool	BitFlag	false	
size	Point	MultiDimensionProperty	（100,100）	宽高
position	Point	SpatialProperty	（0,0）	位置
roundness	Float	SimpleProperty	0.0f	

Ellipse

Ellipse 标签。

Ellipse 的 AttributeBlock 结构表

字段	字段类型	属性类型	默认值	备注
reversed	Bool	BitFlag	false	
size	Point	MultiDimensionProperty	（100,100）	宽高
position	Point	SpatialProperty	（0,0）	位置

PolyStar

多边星形标签。

PolyStar 的 AttributeBlock 结构表

字段	字段类型	属性类型	默认值	备注
reversed	Bool	BitFlag	false	
polyType	枚举（Uint8）	Value	PolyStarType::Star	
points	Float	SimpleProperty	5.0f	
position	Point	SpatialProperty	（0,0）	位置
rotation	Float	SimpleProperty	0.0f	
innerRadius	Float	SimpleProperty	50.0f	
outerRadius	Float	SimpleProperty	100.0f	
innerRoundness	Float	SimpleProperty	0.0f	
outerRoundness	Float	SimpleProperty	0.0f	

ShapePath

ShapePath 标签。

ShapePath 的 AttributeBlock 结构表

字段	字段类型	属性类型	默认值	备注
shapePath	Path	SimpleProperty		

Fill 标签

Fill 标签。

Fill 的 AttributeBlock 结构表

字段	字段类型	属性类型	默认值	备注
blendMode	枚举（Uint8）	Value	BlendMode::Normal	
composite	枚举（Uint8）	Value	CompositeOrder::BelowPreviousInSameGroup	
fillRule	枚举（Uint8）	Value	FillRule::NonZeroWinding	
color	Color	SimpleProperty	Red	
opacity	Uint8	SimpleProperty	255	透明度（0～255）

Stroke

Stroke 标签。

Stroke 的 AttributeBlock 结构表

字段	字段类型	属性类型	默认值	备注
blendMode	枚举（Uint8）	Value	BlendMode::Normal	
composite	枚举（Uint8）	Value	CompositeOrder::BelowPreviousInSameGroup	
lineCap	枚举（Uint8）	Value	LineCap::Butt	
lineJoin	枚举（Uint8）	Value	LineJoin::Miter	
miterLimit	Float	SimpleProperty	4.0f	
color	Color	SimpleProperty	White	
opacity	Uint8	SimpleProperty	255	透明度（0～255）
strokeWidth	Float	SimpleProperty	2.0f	

GradientFill

GradientFill 标签。

GradientFill 的 AttributeBlock 结构表

字段	字段类型	属性类型	默认值	备注
blendMode	枚举（Uint8）	Value	BlendMode::Normal	
composite	枚举（Uint8）	Value	CompositeOrder::BelowPreviousInSameGroup	
fillRule	枚举（Uint8）	Value	FillRule::NonZeroWinding	
fillType	枚举（Uint8）	Value	GradientFillType::Linear	
startPoint	Point	SpatialProperty	（0,0）	开始位置
endPoint	Point	SpatialProperty	（100,0）	结束位置
colors	Color[]	SimpleProperty		
opacity	Uint8	SimpleProperty	255	透明度（0～255）

GradientStroke

GradientStroke 标签。

GradientStroke 的 AttributeBlock 结构表

字段	字段类型	属性类型	默认值	备注
blendMode	枚举（Uint8）	Value	BlendMode::Normal	
composite	枚举（Uint8）	Value	CompositeOrder::BelowPreviousInSameGroup	
fillType	枚举（Uint8）	Value	GradientFillType::Linear	
startPoint	Point	SpatialProperty	（0,0）	开始位置
endPoint	Point	SpatialProperty	（100,0）	结束位置
color	Color	SimpleProperty	White	
opacity	Uint8	SimpleProperty	255	透明度（0 ~ 255）
strokeWidth	Float	SimpleProperty	2.0f	
lineCap	枚举（Uint8）	Value	LineCap::Butt	
lineJoin	枚举（Uint8）	Value	LineJoin::Miter	
miterLimit	Float	SimpleProperty	4.0f	
dashLength	UB[3]			dashLength = UB[3] + 1
dashOffsetFlag.exist	UB[1]			是否含有dashOffset标识
dashOffsetFlag.animatable	UB[1]			if dashOffsetFlag.exist = 1

MergePaths

MergePaths 标签。

MergePaths 的 AttributeBlock 结构表

字段	字段类型	属性类型	默认值	备注
mode	枚举（Uint8）	Value	MergePathsMode::Add	

TrimPaths

TrimPaths 标签。

TrimPaths 的 AttributeBlock 结构表

字段	字段类型	属性类型	默认值	备注
start	Float	SimpleProperty	0.0f	
end	Float	SimpleProperty	100.0f	
offset	Float	SimpleProperty	4.0f	
trimType	枚举（Uint8）	Value	TrimPathsType::Simultaneously	

Repeater

Repeater 标签。

Repeater 的 AttributeBlock 结构表

字段	字段类型	属性类型	默认值	备注
composite	枚举（Uint8）	Value	CompositeOrder::BelowPreviousInSameGroup	
copies	Float	SimpleProperty	3.0f	
offset	Float	SimpleProperty	0.0f	
anchorPoint	Point	SpatialProperty	（0,0）	
position	Point	SpatialProperty	（100,0）	
scale	Point	MultiDimensionProperty	(1.1)	缩放
rotation	Float	SimpleProperty	0.0f	
startOpacity	Uint8	SimpleProperty	255	透明度（0 ~ 255）
endOpacity	Uint8	SimpleProperty	255	透明度（0 ~ 255）

RoundCorners

RoundCorners 标签。

RoundCorners 的 AttributeBlock 结构表

字段	字段类型	属性类型	默认值	备注
radius	Float	SimpleProperty	10.0f	

Performance

Performance 标签主要存储 PAG 的性能指标数据。

Performance 结构表

字段	字段类型	备注
renderingTime	EncodedInt64	渲染耗时
imageDecodingTime	EncodedInt64	解压缩耗时
presentingTime	EncodedInt64	
graphicsMemory	EncodedInt64	渲染内存

DropShadowStyle

DropShadowStyle 标签。

DropShadowStyle 的 AttributeBlock 结构表

字段	字段类型	属性类型	默认值	备注
blendMode	枚举（Uint8）	DiscreteProperty	BlendMode::Normal	
color	Color	SimpleProperty	Black	
opacity	Uint8	SimpleProperty	191	透明度（0 ~ 255）
angle	Float	SimpleProperty	120.0f	
distance	Float	SimpleProperty	5.0f	
size	Float	DiscreteProperty	5.0f	

BitmapCompositionBlock

BitmapCompositionBlock 位图序列帧标签。

BitmapCompositionBlock 的 AttributeBlock 结构表

字段	字段类型	备注
id	EncodedUint32	唯一标识
TagBlock	TagBlock	TagBlock数据块，参考第三章TagBlock。 包含：CompositionAttributes(组合基本属性信息), BitmapSequence(位图信息) 这两种TAG

BitmapSequence

BitmapSequence 标签。

BitmapSequence 结构表

字段	字段类型	备注
width	EncodedUint32	
height	EncodedUint32	
frameRate	Float	
frameCount	EncodedUint32	位图帧数目
isKeyFrameFlag[frameCount]	UB[frameCount]	frameCount 个是否为关键帧的标识
bitmapRect[frameCount]	BitmapRect[frameCount]	frameCount 个 BitmapRect 数据

ImageBytes

ImageBytes 图片标签，存储了压缩后的图片相关属性信息。

ImageBytes 结构表

字段	字段类型	备注
id	EncodedUint32	
fileBytes	ByteData	图片字节流

ImageBytes2

ImageBytes2 图片标签版本 2，除了存储 ImageBytes 的信息外，还允许记录图片的缩放参数，通常根据实际最大用到的大小来存储图片，而不是按原始大小。

ImageBytes2 结构表

字段	字段类型	备注
id	EncodedUint32	
fileBytes	ByteData	图片字节流
scaleFactor	Float	缩放比例 (0 ~ 1.0)

ImageBytes3

ImageBytes3 图片标签版本 3， 除了包含 ImageBytes2 的信息外，还允许记录剔除透明边框后的图片。

字段	字段类型	备注
id	EncodedUint32	
fileBytes	ByteData	图片字节流
scaleFactor	Float	缩放比例 (0 ~ 1.0)
width	EncodedInt32	原始图片宽
width	EncodedInt32	原始图片宽
anchorX	EncodedInt32	原始图片中的不透明区域起点 x 轴坐标
anchorY	EncodedInt32	原始图片中的透明区域起点 y 轴坐标

VideoCompositionBlock

VideoCompositionBlock 存储了 1 个或多个不同尺寸的视频序列帧。

VideoCompositionBlock 结构表

字段	字段类型	备注
id	EncodedUint32	唯一标识
hasAlpha	Bool	是否有 Alpha 通道
CompositionAttributes	CompositionAttributes Tag	
TagBlock	TagBlock	TagBlock数据块，参考第三章TagBlock。 包含：CompositionAttributes(组合基本属性信息), VideoSequence(位图信息) 这两种TAG

VideoSequence

VideoSequence 存储了 1 个版本的视频序列帧的结构。

VideoSequence 结构表

字段	字段类型	备注
width	EncodedUint32	
height	EncodedUint32	
frameRate	Float	
alphaStartX	EncodedInt32	如果 VideoCompositionBlock 读取出来的 hasAlpha 为 1，才会有这个值
alphaStartY	EncodedInt32	如果 VideoCompositionBlock 读取出来的 hasAlpha 为 1，才会有这个值
spsData	ByteData	读取出来的字节流不包含 (0, 0, 0, 1) 四字节的 Start Code
ppsData	ByteData	读取出来的字节流不包含 (0, 0, 0, 1) 四字节的 Start Code
frameCount	EncodedUint32	位图帧数目
isKeyFrameFlag [frameCount]	UB[frameCount]	frameCount 个是否为关键帧的标识
videoFrames	VideoFrame[frameCount]	视频帧数目