公告

语言：c/c++ c# java

领域：各平台流媒体系统、直播监控、视频会议、音视频处理、网络通信

口号：构建全平台的通用流媒体系统

联系方式

商务合作：

QQ 229375788

淘宝店铺：

haibindev.taobao.com

技术交流：
haibindev@gmail.com

昵称： haibindev
园龄： 10年7个月
荣誉： 推荐博客
粉丝： 578
关注： 0
+加关注

< 2011年10月 >

日	一	二	三	四	五	六
25	26	27	28	29	30	1
2	3	4	5	6	7	8
9	10	11	12	13	14	15
16	17	18	19	20	21	22
23	24	25	26	27	28	29
30	31	1	2	3	4	5

MP4文件格式的解析，以及MP4文件的分割算法

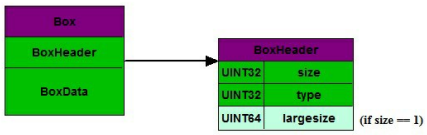
mp4应该算是一种比较复杂的媒体格式了，起源于QuickTime。以前研究的时候就花了一番的功夫，尤其是如何把它完美的融入到**视频点播**应用中，更是费尽了心思，主要问题是处理mp4文件庞大的“媒体头”。当然，流媒体点播也可以采用flv格式来做，flv也可以封装H.264视频数据的，不过Adobe却不推荐这么做，人家说毕竟mp4才是H.264最佳的存储格式嘛。

这几天整理并重构了一下mp4文件的解析程序，融合了分解与合并的程序，以前是c语言写的，应用在linux上运行的服务器程序上，现在改成c++，方便我在其他项目中使用它，至于用不用移植一份c#的，暂时用不到，等有必要了再说吧。这篇文章先简单介绍一下mp4文件的大体结构，以及它的分割算法，之后再写文章介绍如何把mp4完美应用在点播项目中。

一、MP4格式分析

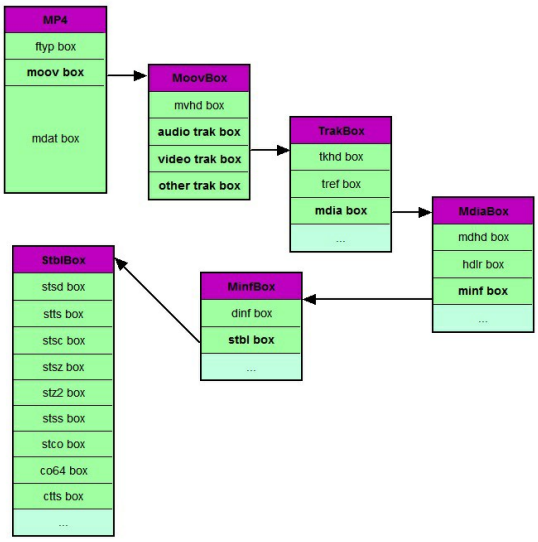
MP4(MPEG-4 Part 14)是一种常见的多媒体容器格式，它是在“ISO/IEC 14496-14”标准文件中定义的，属于MPEG-4的一部分，是“ISO/IEC 14496-12(MPEG-4 Part 12 ISO base media file format)”标准中所定义的媒体格式的一种实现，后者定义了一种通用的媒体文件结构标准。MP4是一种描述较为全面的容器格式，被认为可以在其中嵌入任何形式的数据，各种编码的视频、音频等都不在话下，不过我们常见的大部分的MP4文件存放的**AVC(H.264)**或**MPEG-4(Part 2)**编码的视频和**AAC**编码的音频。MP4格式的官方文件后缀名是“.mp4”，还有其他的以mp4为基础进行的扩展或者是缩水版本的格式，包括：**M4V**，**3GP**，**F4V**等。

mp4是由一个个“box”组成的，大box中存放小box，一级嵌套一级来存放媒体信息。box的基本结构是：



其中，size指明了整个box所占用的大小，包括header部分。如果box很大(例如存放具体视频数据的mdat box)，超过了uint32的最大数值，size就被设置为1，并用接下来的8位uint64来存放大小。

一个mp4文件有可能包含非常多的box，在很大程度上增加了解析的复杂性，这个网页上http://mp4ra.org/atoms.html记录了一些当前注册过的box类型。看到这么多box，如果要全部支持，一个个解析，怕是头都要爆了。还好，大部分mp4文件没有那么多的box类型，下图就是一个简化了的，常见的mp4文件结构：



一般来说，解析媒体文件，最关心的部分是视频文件的宽高、时长、码率、编码格式、帧列表、关键帧列表，以及所对应的时戳和在文件中的位置，这些信息，在mp4中，是以特定的算法分开存放在stbl box下属的几个box中的，需要解析stbl下面所有的box，来还原媒体信息。下表是对于以上几个重要的box存放信息的说明：

box类型	说明
ftyp	file type， 表明文件类型
moov	metadata container，存放媒体信息的地方
mvhd	movie header，文件的总体信息，如时长，创建时间等
trak	track or stream container，存放视频/音频流的容器
tkhd	track header，track的总体信息，如时长，宽高等
mdia	trak media information container，不解释
mdhd	media header，定义TimeScale，trak需要通过TimeScale换算成真实时间
hdlr	handler，表明本trak类型，指明是video/audio/还是hint
minf	media information container，数据在子box中
stbl	sample table box，存放时间/偏移的映射关系表，数据在子box中
stsd	sample descriptions
stts	(decoding) time-to-sample, "时戳-sample序号"的映射表
stsc	sample-to-chunk，sample和chunk的映射表，这里的算法比较巧妙
stsz	sample size，每个sample的大小
stz2	sample size，另一种sample size的存储算法，更节省空间
stss	sync sample table，可随机访问的sample列表（关键帧列表）
stco	chunk offset，每个chunk的偏移，sample的偏移可根据其他box推算出来
co64	64-bit chunk offset
mdat	media data container，具体的媒体数据

看吧，要获取到mp4文件的帧列表，还挺不容易的，需要一层层解析，然后综合stts stsc stsz stss stco等这几个box的信息，才能还原出帧列表，每一帧的时戳和偏移量。而且，你要照顾可能出现或者可能不出现的那些box。。。可以看的出来，mp4把帧sample进行了分组，也就是chunk，需要间接的通过chunk来描述帧，这样做的理由是可以压缩存储空间，缩小媒体信息所占用的文件大小。这里面，stsc box的解析相对来说比较复杂，它用了一种巧妙的方式来说明sample和chunk的映射关系，特别介绍一下。

```
BoxHeader header;
UInt8 version;
UInt24 flags;
UInt32 entry_count;
for (i = 0; i < entry_count; i++) {
   UInt32 first_chunk;
   UInt32 samples_per_chunk;
   UInt32 sample_description_index;
}
```

这是stsc box的结构，前几项的意义就不解释了，可以看到stsc box里每个entry结构体都存有三项数据，它们的意思是：“从**first_chunk**这个chunk序号开始，每个chunk都有**samples_per_chunk**个数的sample，而且每个sample都可以通过**sample_description_index**这个索引，在stsd box中找到描述信息”。也就是说，每个entry结构体描述的是一组chunk，它们有相同的特点，那就是每个chunk包含**samples_per_chunk**个sample，好，那你要问，这组相同特点的chunk有多少个？请通过下一个entry结构体来推算，用下一个entry的**first_chunk**减去本次的**first_chunk**，就得到了这组chunk的个数。最后一个entry结构体则表明从该**first_chunk**到最后一个chunk，每个chunk都有**sampls_per_chunk**个sample。很拗口吧，不过，就是这个意思:)。由于这种算法无法得知文件所有chunk的个数，所以你必须借助于stco或co64。直接上代码可能会清楚些：

1. 首先直接分析entry

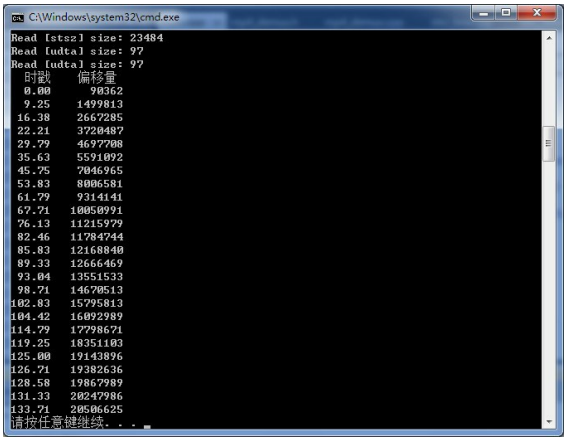
```
34 const char* VPMp4StscBox::FromBuf(const char* buf)
35 {
36     buf = ReadBoxHeader(&header_, buf);
37     buf = ReadBoxVF(&version_, &flags_, buf);
38
39     vp_uint32_t entry_count = BytesToUI32(buf); buf += 4;
40     for (int i = 0; i != entry_count; ++i)
41     {
```

```
        StscEntry stsc_entry;
        stsc_entry.first_chk = BytesToUI32(buf); buf += 4;
        stsc_entry.sams_per_chk = BytesToUI32(buf); buf += 4;
        stsc_entry.sdi = BytesToUI32(buf); buf += 4;
        entrys_.push_back(stsc_entry);
    }
    return buf;
}
```

2. 然后，通过stco或co64获知chunk总个数之后，开始还原映射表

```
void VPMp4StscBox::SetTotalChunkCount(vp_uint32_t totalChunkCount)
{
    chk_entrys_.resize(totalChunkCount);

    // 还原 chunk --> samples 列表
    vp_uint32_t last_real_chkno = totalChunkCount + 1;
    for (int i = entrys_.size() - 1; i >= 0; --i)
    {
        vp_uint32_t beg_real_chkno = entrys_[i].first_chk;
        for (vp_uint32_t chk_i = beg_real_chkno - 1; chk_i < last_real_chkno - 1; ++chk_i)
        {
            chk_entrys_[chk_i].sam_count = entrys_[i].sams_per_chk; // 每个chunk的sample个数
            chk_entrys_[chk_i].sdi = entrys_[i].sdi;
        }
        last_real_chkno = beg_real_chkno;
    }

    // 还原 sample --> chunk 列表
    vp_uint32_t sam_index = 0;
    for (vp_uint32_t i = 0; i != chk_entrys_.size(); ++i)
    {
        // chunk包含的第一个sample序号
        chk_entrys_[i].first_sam_index = sam_index;

        vp_uint32_t index_in_chunk = 0;
        for (vp_uint32_t sam_i = 0; sam_i != chk_entrys_[i].sam_count; ++ sam_i)
        {
            StscSampleEntry stsc_sam_entry;
            stsc_sam_entry.chunk_index = i;                     // sample对应的chunk序号
            stsc_sam_entry.index_in_chunk = index_in_chunk;     // sample在chunk内部的序号
            sam_entrys_.push_back(stsc_sam_entry);

            sam_index++;
            index_in_chunk++;
        }
    }

    assert(sam_index == sam_entrys_.size());
}
```

读出stsc之后，就可以综合stbl下的所有box，推算出视频和音频帧列表，时戳和偏移量等数据。下面截图展示获取到的关键帧列表：



有了关键帧列表之后，就可以继续我们一下个题目，就是**mp4文件的分割。**实现mp4的分割，是把mp4应用到点播系统中最关键的技术环节，做不到这个，就无法实现点播播放mp4影片的“**拖动**”。

二、MP4文件的分割算法

所谓“分割”，就是把大文件切成小文件，要实现mp4的分割，

- 首先，需要获取到关键帧列表
- 然后，选择要分割的时间段（比如从关键帧开始）
- 接着，重新生成moov box（注意所有相关的box 以及 box size都需要改变）
- 最后，拷贝对应的数据，生成新文件

第一点，上面已经介绍了，第二点，只需要遍历关键帧列表，就能找到离你想要分割的时间段最接近的关键帧，第四点就是“copy-paste”的工作，关键在于第三点。因为这一步涉及到stbl下的所有box，必须重新生成entrys，同样的，其他的box都还好，只需要保留关键帧所对应的sample和chunk，其余的删掉即可，只是stsc box的比较麻烦，说起来比较啰嗦，还是直接看代码吧：

```
void VPMp4StscBox::Cut(vp_uint32_t samIndexFrom, vp_uint32_t samIndexTo)
{
    // 起始sample对应的chunk index
    vp_uint32_t chunk_from_index = sam_entrys_[samIndexFrom].chunk_index;
    vp_uint32_t chunk_from_first_sam_index = chk_entrys_[chunk_from_index].first_sam_index;

    chk_entrys_[chunk_from_index].first_sam_index = samIndexFrom;
    chk_entrys_[chunk_from_index].sam_count -= (samIndexFrom - chunk_from_first_sam_index);

    // 结尾sample对应的chunk index
    vp_uint32_t chunk_to_index = sam_entrys_[samIndexTo].chunk_index;
    vp_uint32_t chunk_to_first_sam_index = chk_entrys_[chunk_to_index].first_sam_index;

    chk_entrys_[chunk_to_index].sam_count = samIndexTo - chunk_to_first_sam_index + 1;

    // 生成新的 sample --> chunk 映射表
    std::vector<StscSampleEntry> new_sam_entrys;
    for (vp_uint32_t sam_i = samIndexFrom; sam_i <= samIndexTo; ++sam_i)
    {
        StscSampleEntry sam_entry;
        sam_entry.chunk_index = sam_entrys_[sam_i].chunk_index - chunk_from_index;
        if (sam_entrys_[sam_i].chunk_index == chunk_from_index)
        {
            sam_entry.index_in_chunk = sam_entrys_[sam_i].index_in_chunk - sam_entrys_[samIndexFrom].index_in_chunk;
        }
        else
        {
            sam_entry.index_in_chunk = sam_entrys_[sam_i].index_in_chunk;
        }
        new_sam_entrys.push_back(sam_entry);
    }

    // 生成新的 chunk --> samples 映射表
    std::vector<StscChunkEntry> new_chk_entrys;
    for (vp_uint32_t chk_i = chunk_from_index; chk_i <= chunk_to_index; ++chk_i)
    {
        StscChunkEntry chk_entry;
        chk_entry.first_sam_index = chk_entrys_[chk_i].first_sam_index - samIndexFrom;
        chk_entry.sam_count = chk_entrys_[chk_i].sam_count;
        chk_entry.sdi = chk_entrys_[chk_i].sdi;
        new_chk_entrys.push_back(chk_entry);
    }

    sam_entrys_ = new_sam_entrys;
    chk_entrys_ = new_chk_entrys;
}
```

修改完box之后，需要重新生成moov box，由于moov box的大小以及时长等信息都发生了改变，所以需要box的大小做相应的修改，这点千万不能忘记，否则播放器会解析错误。重新生成box之后，还要计算一下分割后的数据的长度，由于数据长度也发生了改变，所以修改mdat box的大小的同时，要同时修改stbl下所有box的chunk offset，切记！

以下是整个的逻辑过程：

```
        FILE* fpo = fopen("copy.mp4", "wb");

        VPMp4Descriptor mp4_desc;
        mp4_desc.ParseFile(argv[1]);

        vp_uint64_t offset_from, offset_to;
        mp4_desc.Cut(0, 30, offset_from, offset_to);
        mp4_desc.MinusOffset(offset_from);
```

```
        vp_uint32_t moov_size = mp4_desc.CheckMoovSize(mp4_desc.MoovSize() + 1024);
        mp4_desc.PlusOffset(mp4_desc.FtypSize() + moov_size + mp4_desc.MdatHeaderSize());

        vp_uint64_t new_data_size = offset_to - offset_from;
        mp4_desc.SetDataSize(new_data_size);

        mp4_desc.WriteToFile(fpo);

        FILE* fpi = fopen(argv[1], "rb");
        vp_fseek_64(fpi, offset_from, SEEK_SET);
        char* buf = (char*)malloc(mp4_desc.DataSize());
        fread(buf, mp4_desc.DataSize(), 1, fpi);
        fclose(fpi);

        fwrite(buf, mp4_desc.DataSize(), 1, fpo);
        fclose(fpo);
```

好了，所有这些都实现之后，就具备了做mp4点播系统的条件了。不过，要做mp4点播，还有一些其他的问题需要解决，我将在下一篇文章中介绍。

++

HBStream

QQ:229375788 **haibindev.cnblogs.com，合作请联系QQ。（转载请注明作者和出处）**

++

标签: 视频, c++, mp4, 流媒体, 点播

haibindev
关注 - 0
粉丝 - 578
推荐博客
+加关注

« 上一篇： 做了一个电驴 p2p资源搜索小软件
» 下一篇： DirectShow捕获+mencoder+ffmpeg+sox 打造小巧的音视频制作、加工软件

posted on 2011-10-17 12:08 haibindev 阅读(112448) 评论(30) 编辑 收藏 举报