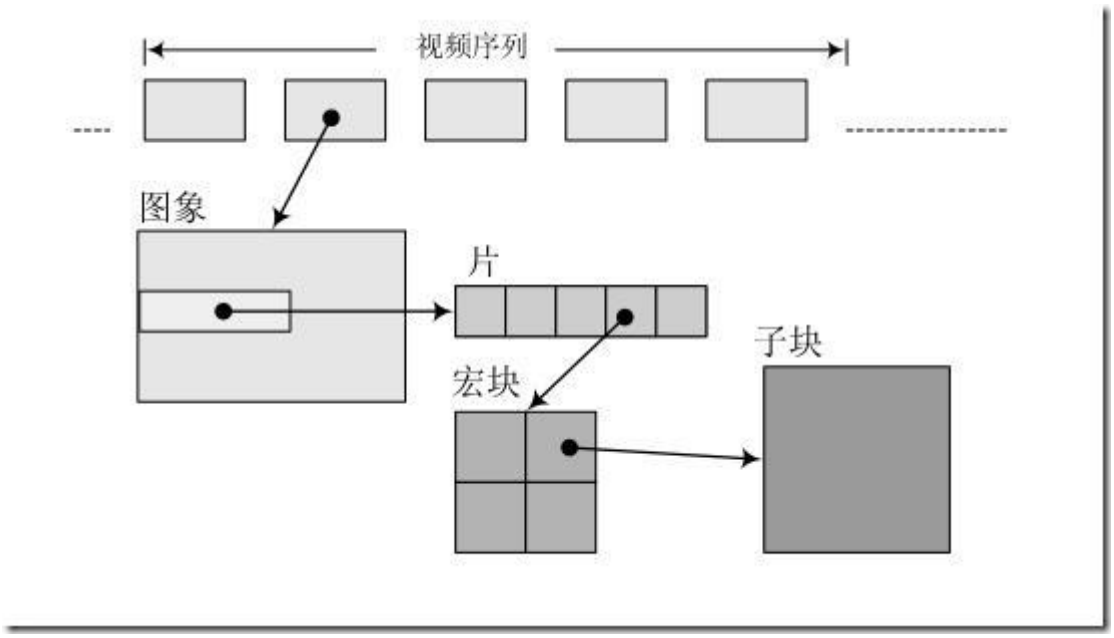
http://blog.csdn.net/xfding/article/details/5467952

# H.264 句法和语法总结（一）句法元素的分层结构

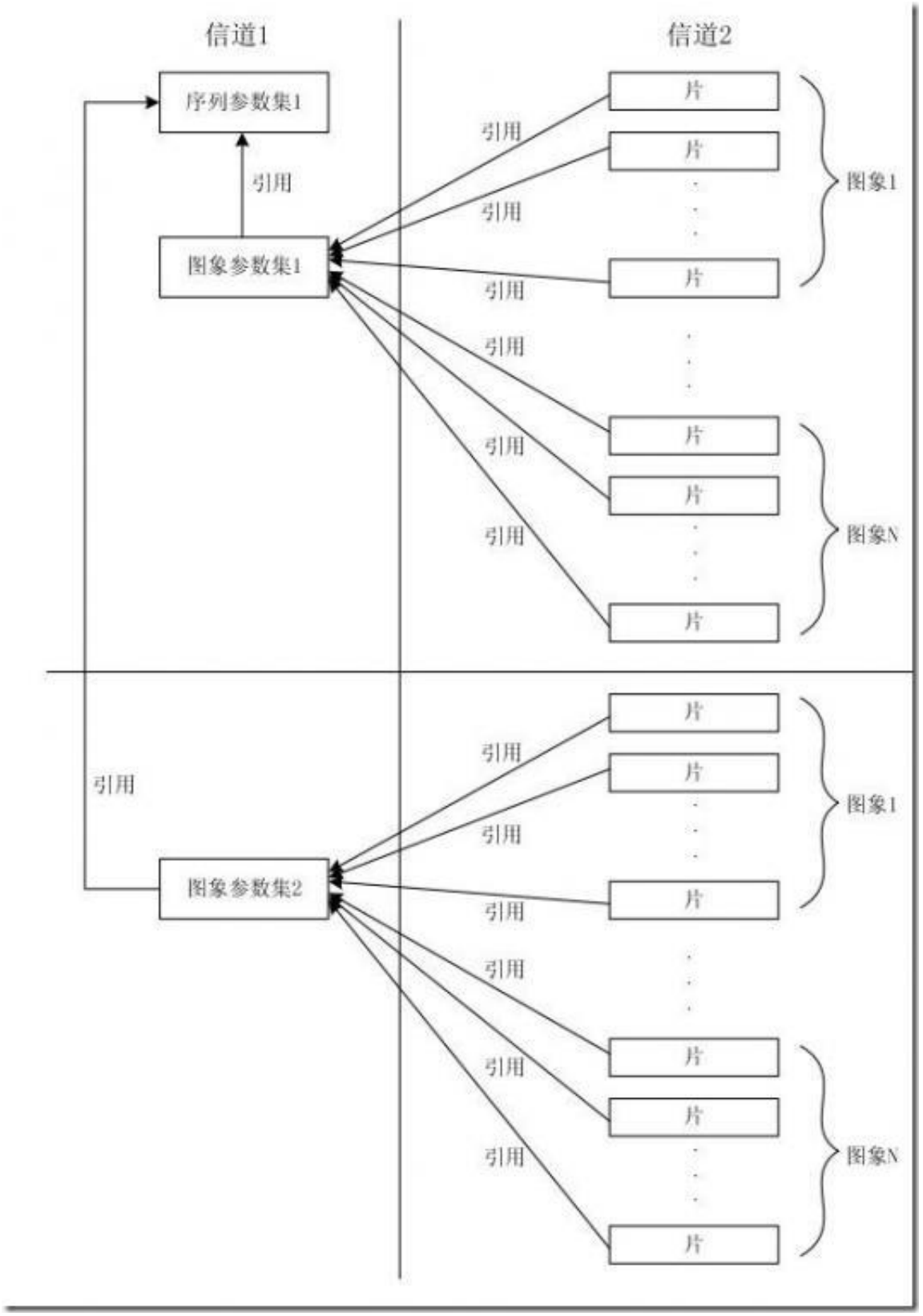


在 H.264 定义的码流中，句法元素被组织成有层次的结构，分别描述各个层次的信息，如下图所示



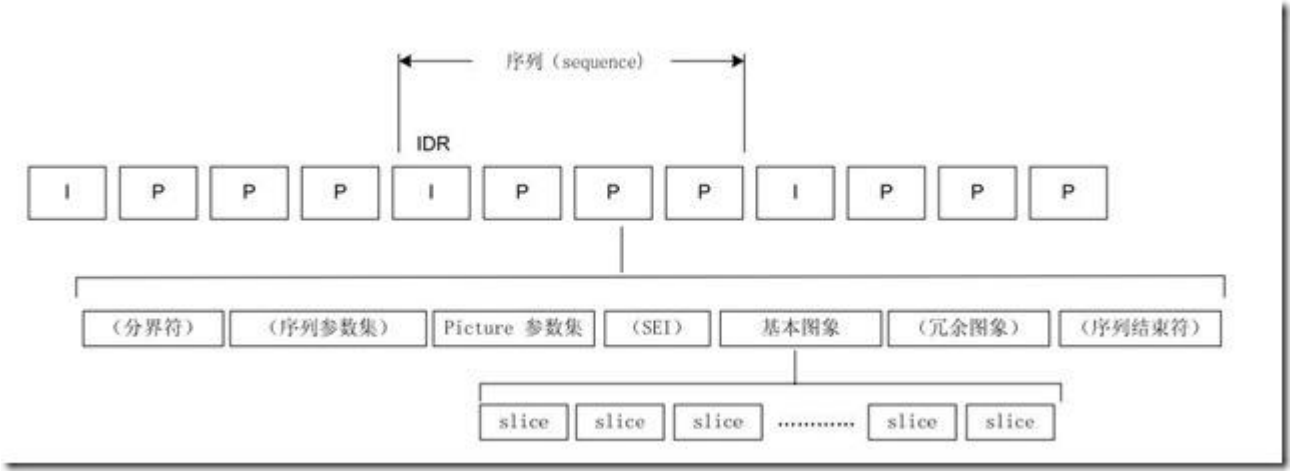

在 H.264 中，句法元素共被组织成 序列、图像、片、宏块、子宏块五个层次。

在这样的结构中，每一层的头部和它的数据部分形成管理与被管理的强依赖关系，头部的句法元素是该层数据的核心，而一旦头部丢失，数据部分的信息几乎不可能再被正确解码出来，尤其在序列层及图像层。


信道1
信道2
序列参数集1
引用
图象参数集1
片
图象1
图象N
图象参数集2

在 H.264 中，分层结构最大的不同是取消了序列层和图像层，并将原本属于序列和图像头部的大部分句法元素游离出来形成序列和图像两级参数集，其余的部分则放入片层。参数集是一个独立的数据单位，不依赖于参数集外的其他句法元素。由于参数集是独立的，可以被多次重发或者采用特殊技术加以保护。

复杂通信中的码流中可能出现的数据单位：




**IDR:** 一个序列的第一个图像叫做 IDR 图像（立即刷新图像），IDR 图像都是 I 图像。H.264 引入 IDR 图像是为了解码的重同步，当解码器解码到 IDR 图像时，立即将参考帧队列清空，将已解码的数据全部输出或抛弃，重新查找参数集，开始一个新的序列。IDR 图像一定是 I 图像，但 I 图像不一定是 IDR 图像。

http://blog.csdn.net/xfding/article/details/5468195

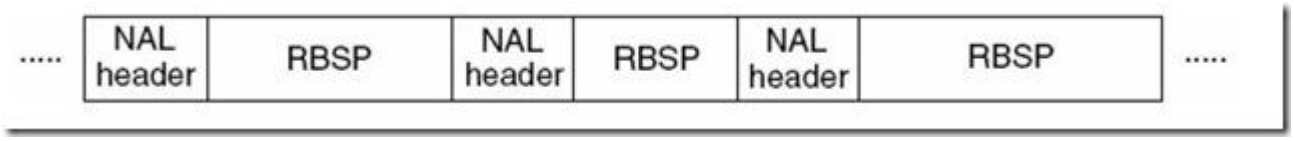
# H.264 句法和语法总结（二）NAL 层句法



h.264bytenetworklayer 存储网络

**NAL&VCL:** H.264 的功能分为两层，即视频编码层（VCL）和网络提取层（NAL，Network Abstraction Layer）。VCL 数据即编码处理的输出，它表示被压缩编码后的视频数据序列。在 VCL 数据传输或存储之前，这些编码的 VCL 数据，先被映射或封装进 NAL 单元中。

每个 NAL 单元包括一个原始字节序列负荷（RBSP）、一组对应于视频编码数据的 NAL 头信息。NAL 单元序列的结构如下：




**RBSP 的类型：**

| RBSP 类型 | 描　　述 |
|---|---|
| 参数集 PS | 序列的全局参数，如图像尺寸、视频格式等等 |
| 增强信息 SEI | 视频序列解码的增强信息 |
| 图像定界符 PD | 视频图像的边界 |
| 编码片 | 片的头信息和数据 |
| 数据分割 | DP 片层的数据，用于错误恢复解码 |
| 序列结束符 | 表明下一图像为 IDR 图像 |
| 流结束符 | 表明该流中已没有图像 |
| 填充数据 | 哑元数据，用于填充字节 |

**PS:** 包括**序列参数集 SPS** 和**图像参数集 PPS**

**SPS** 包含的是针对一连续编码视频序列的参数，如标识符 seq_parameter_set_id、帧数及 POC 的约束、参考帧数目、解码图像尺寸和帧场编码模式选择标识等等。

**PPS** 对应的是一个序列中某一幅图像或者某几幅图像，其参数如标识符 pic_parameter_set_id、可选的 seq_parameter_set_id、熵编码模式选择标识、片组数目、初始量化参数和去方块滤波系数调整标识等等。

**数据分割：** 组成片的编码数据存放在 3 个独立的 DP（数据分割，A、B、C）中，各自包含一个编码片的子集。分割 A 包含片头和片中每个宏块头数据。分割 B 包含帧内和 SI 片宏块的编码残差数据。分割 C 包含帧间宏块的编码残差数据。每个分割可放在独立的 NAL 单元并独立传输。

**NAL 层句法：**

```
nal_unit( NumBytesInNALunit ) {
  // forbidden_zero_bit 等于 0
  forbidden_zero_bit
  // nal_ref_idc 指示当前 NAL 的优先级。取值范围为 0-3，值越高,表示当前 NAL 越重要,需要优先受到保护。H.264 规定如果当前 NAL 是属于参考帧的片，或是序列参数集，或是图像参数集这些重要的数据单位时，本句法元素必须大于 0。
  nal_ref_idc
```

// nal_unit_type 指明当前 NAL unit 的类型

**nal_unit_type**

NumBytesInRBSP = 0

/* rbsp_byte[i] RBSP 的第 i 个字节。RBSP 指原始字节载荷，它是 NAL 单元的数据部分的封装格式，封装的数据来自 SODB（原始数据比特流）。SODB 是编码后的原始数据，SODB 经封装为 RBSP 后放入 NAL 的数据部分。下面介绍一个 RBSP 的生成顺序。

从 SODB 到 RBSP 的生成过程：

- 如果 SODB 内容是空的，生成的 RBSP 也是空的
- 否则，RBSP 由如下的方式生成：

1） RBSP 的第一个字节直接取自 SODB 的第 1 到 8 个比特，（RBSP 字节内的比特按照从左到右对应为从高到低的顺序排列，most significant）,以此类推，RBSP 其余的每个字节都直接取自 SODB 的相应比特。RBSP 的最后一个字节包含 SODB 的最后几个比特，及如下的 rbsp_trailing_bits()

2） rbsp_trailing_bits()的第一个比特是 1,接下来填充 0，直到字节对齐。（填充 0 的目的也是为了字节对齐）

3） 最后添加若干个 cabac_zero_word(其值等于 0x0000)

*/

```
for( i = 1; i < NumBytesInNALunit; i++ ) {
    if( i + 2 < NumBytesInNALunit && next_bits( 24 ) == 0x000003 ) {
        rbsp_byte[ NumBytesInRBSP++ ]
        rbsp_byte[ NumBytesInRBSP++ ]
        i += 2
        //emulation_prevention_three_byte NAL 内部为防止与起始码竞争而引入的填充字节 ,值为 0x03。
        emulation_prevention_three_byte
    } else
        rbsp_byte[ NumBytesInRBSP++ ]
}
}
```

http://blog.csdn.net/xfding/article/details/5468347

# H.264 句法和语法总结（三）序列参数集层（SPS)句法



h.264parameters 存储扩展

## seq_parameter_set_rbsp( ) {

// profile_idc level_idc 指明所用 profile、level

**profile_idc**

// constraint_set0_flag 等于 1 时表示必须遵从附录 A.2.1 所指明的所有制约条件。等于 0 时表示不必遵从所有条件。

**constraint_set0_flag**

// constraint_set1_flag 等于 1 时表示必须遵从附录 A.2.2 所指明的所有制约条件。等于 0 时表示不必遵从所有条件。

**constraint_set1_flag**

// constraint_set2_flag 等于 1 时表示必须遵从附录 A.2.3 所指明的所有制约条件。等于 0 时表示不必遵从所有条件。

**constraint_set2_flag**

// reserved_zero_5bits 在目前的标准中本句法元素必须等于 0，其他的值保留做将来用，解码器应该忽略本句法元素的值。

**reserved_zero_5bits** /* equal to 0 */

**level_idc**

// seq_parameter_set_id 指明本序列参数集的 id 号，这个 id 号将被 picture 参数集引用，本句法元素的值应该在[0，31]。

**seq_parameter_set_id**

// log2_max_frame_num_minus4 这个句法元素主要是为读取另一个句法元素 frame_num 服务的，frame_num 是最重要的句法元素之一，它标识所属图像的解码顺序 。这个句法元素同时也指明了 frame_num 的所能达到的最大值： MaxFrameNum = 2*exp( log2_max_frame_num_minus4 + 4 )

**log2_max_frame_num_minus4**

// pic_order_cnt_type 指明了 poc (picture order count) 的编码方法, poc 标识图像的播放顺序。由 poc 可以由 frame-num 通过映射关系计算得来，也可以索性由编码器显式地传送。

**pic_order_cnt_type**

if( **pic_order_cnt_type** == 0 )

// log2_max_pic_order_cnt_lsb_minus4 指明了变量 MaxPicOrderCntLsb 的值: MaxPicOrderCntLsb = pow(2, (log2_max_pic_order_cnt_lsb_minus4 + 4) )

**log2_max_pic_order_cnt_lsb_minus4**

else if( **pic_order_cnt_type** == 1 ) {

// delta_pic_order_always_zero_flag 等于 1 时,句法元素 delta_pic_order_cnt[0]和 delta_pic_order_cnt[1]
不在片头出现,并且它们的值默认为 0; 本句法元素等于 0 时,上述的两个句法元素将在片头出现。

**delta_pic_order_always_zero_flag**

// offset_for_non_ref_pic 被用来计算非参考帧或场的 POC,本句法元素的值应该在[pow(-2, 31) , pow(2, 31) – 1]。

**offset_for_non_ref_pic**

// offset_for_top_to_bottom_field 被用来计算帧的底场的 POC, 本句法元素的值应该在[pow(-2, 31) , pow(2, 31) – 1]。

**offset_for_top_to_bottom_field**

// num_ref_frames_in_pic_order_cnt_cycle 被用来解码 POC, 本句法元素的值应该在[0,255]。

**num_ref_frames_in_pic_order_cnt_cycle**

// offset_for_ref__frame[i] 用于解码 POC, 本句法元素对循环 num_ref_frames_in_pic_order_cycle 中的每一个元素指定一个偏移。

for( i = 0; i < num_ref_frames_in_pic_order_cnt_cycle; i++ )

**offset_for_ref_frame**[ i ]

}

// num_ref_frames 指定参考帧队列可能达到的最大长度, 解码器依照这个句法元素的值开辟存储区, 这个存储区用于存放已解码的参考帧, H.264 规定最多可用 16 个参考帧, 本句法元素的值最大为 16。值得注意的是这个长度以帧为单位, 如果在场模式下, 应该相应地扩展一倍。

**num_ref_frames**

// gaps_in_frame_num_value_allowed_flag 这个句法元素等于 1 时, 表示允许句法元素 frame_num 可以不连续。当传输信道堵塞严重时, 编码器来不及将编码后的图像全部发出, 这时允许丢弃若干帧图像。

**gaps_in_frame_num_value_allowed_flag**

// pic_width_in_mbs_minus1 本句法元素加 1 后指明图像宽度, 以宏块为单位: PicWidthInMbs = pic_width_in_mbs_minus1 + 1 通过这个句法元素解码器可以计算得到亮度分量以像素为单位的图像宽度: PicWidthInSamplesL = PicWidthInMbs * 16

**pic_width_in_mbs_minus1**

// pic_height_in_map_units_minus1 本句法元素加 1 后指明图像高度:

PicHeightInMapUnits = pic_height_in_map_units_minus1 + 1

**pic_height_in_map_units_minus1**

// frame_mbs_only_flag 本句法元素等于 0 时表示本序列中所有图像的编码模式都是帧，没有其他编码模式存在；本句法元素等于 1 时 ，表示本序列中图像的编码模式可能是帧，也可能是场或帧场自适应，某个图像具体是哪一种要由其他句法元素决定。

**frame_mbs_only_flag**

// mb_adaptive_frame_field_flag 指明本序列是否属于帧场自适应模式。mb_adaptive_frame_field_flag 等于 1 时表明在本序列中的图像如果不是场模式就是帧场自适应模式，等于 0 时表示本序列中的图像如果不是场模式就是帧模式。。表 列举了一个序列中可能出现的编码模式：

if( !**frame_mbs_only_flag** )

**mb_adaptive_frame_field_flag**

// direct_8x8_inference_flag 用于指明 B 片的直接和 skip 模式下运动矢量的预测方法。

**direct_8x8_inference_flag**

// frame_cropping_flag 用于指明解码器是否要将图像裁剪后输出，如果是的话，后面紧跟着的四个句法元素分别指出左右、上下裁剪的宽度。

**frame_cropping_flag**

if( **frame_cropping_flag** ) {

**frame_crop_left_offset**

**frame_crop_right_offset**

**frame_crop_top_offset**

**frame_crop_bottom_offset**

}

// vui_parameters_present_flag 指明 vui 子结构是否出现在码流中，vui 用以表征视频格式等额外信息。

**vui_parameters_present_flag**

if( **vui_parameters_present_flag** )

**vui_parameters**( )

**rbsp_trailing_bits**( )

}

http://blog.csdn.net/xfding/article/details/5476036

# H.264 句法和语法总结（四）图像参数集语义



h.264filter 工作

pic_parameter_set_rbsp( ) {

// pic_parameter_set_id 用以指定本参数集的序号，该序号在各片的片头被引用。

**pic_parameter_set_id**

// seq_parameter_set_id 指明本图像参数集所引用的序列参数集的序号。

**seq_parameter_set_id**

// entropy_coding_mode_flag 指明熵编码的选择，本句法元素为 0 时，表示熵编码使用 CAVLC，本句法元素为 1 时表示熵编码使用 CABAC

**entropy_coding_mode_flag**

// pic_order_present_flag POC 的三种计算方法在片层还各需要用一些句法元素作为参数, 本句法元素等于 1 时表示在片头会有句法元素指明这些参数; 本句法元素等于 0 时, 表示片头不会给出这些参数，这些参数使用默认值

**pic_order_present_flag**

// num_slice_groups_minus1 本句法元素加 1 后指明图像中片组的个数。H.264 中没有专门的句法元素用于指明是否使用片组模式，当本句法元素等于 0 （即只有一个片组），表示不使用片组模式，后面也不会跟有用于计算片组映射的句法元素。

**num_slice_groups_minus1**

if( **num_slice_groups_minus1** > 0 ) {

/* slice_group_map_type 用以指明片组分割类型。

map_units 的定义：

- 当 frame_mbs_only_flag 等于 1 时，map_units 指的就是宏块
- 当 frame_mbs_only_flag 等于 0 时
  - 帧场自适应模式时，map_units 指的是宏块对
  - 场模式时，map_units 指的是宏块
  - 帧模式时，map_units 指的是与宏块对相类似的，上下两个连续宏块的组合体。 */

**slice_group_map_type**

if( slice_group_map_type = = 0 )

for( iGroup = 0; iGroup <= num_slice_groups_minus1; iGroup++ )

// run_length_minus1[i] 用以指明当片组类型等于 0 时，每个片组连续的 map_units 个数

```
            run_length_minus1[ iGroup ]
        else if( slice_group_map_type    = =    2 )
          for( iGroup = 0; iGroup < num_slice_groups_minus1; iGroup++ ) {
            // top_left[i],bottom_right[i]   用以指明当片组类型等于 2 时, 矩形区域的左上及右下位置。
            top_left[ iGroup ]
            bottom_right[ iGroup ]
          }
        else if(   slice_group_map_type    = =    3    | |
                   slice_group_map_type    = =    4    | |
                   slice_group_map_type    = =    5 ) {
          // slice_group_change_direction_flag  与下一个句法元素一起指明确切的片组分割方法。
          slice_group_change_direction_flag
          // slice_group_change_rate_minus1       用以指明变量 SliceGroupChangeRAte
          slice_group_change_rate_minus1
        } else if( slice_group_map_type    = =    6 ) {
          // pic_size_in_map_units_minus1   在片组类型等于 6 时，用以指明图像以 map_units 为单位的大小。
          pic_size_in_map_units_minus1
          for( i = 0; i <= pic_size_in_map_units_minus1; i++ )
            // slice_group_id[i]          在片组类型等于 6 时，用以指明某个 map_units 属于哪个片组。
            slice_group_id[ i ]
        }
    }
  // num_ref_idx_l0_active_minus1  加 1 后指明目前参考帧队列的长度，即有多少个参考帧（包括短期和长期）。值得注意的是，当目前解码图像是场模式下，参考帧队列的长度应该是本句法元素再乘以 2，因为场模式下各帧必须被分解以场对形式存在。（这里所说的场模式包括图像的场及帧场自适应下的处于场模式的宏块对） 本句法元素的值有可能在片头被重载。
```

在序列参数集中有句法元素 num_ref_frames 也是跟参考帧队列有关，它们的区别是 num_ref_frames 指明参考帧队列的最大值，解码 器用它的值来分配内存空 间；num_ref_idx_l0_active_minus1 指明在这个队列中当前实际的、已存在的参考帧数目，这从它的名字“active”中也可以看出来。图像时，并不是直接传送该图像的编号，而是传送该

图像在参考帧队列中的序号。这个序号并不是在码流中传送的，这个句法元素是 H.264 中最重要的句法元素之一, 编码器要通知解码器某个运动矢量所指向的是哪个参考而是编码器和解码器同步地、用相同的方法将参考图像放入队列，从而获得一个序号。这个队列在每解一个图像，甚至是每个片后都会动态地更新。维护参考帧队列是编解码器十分重要的工作，而本句法元素是维护参考帧队列的重要依据。参考帧队列的复杂的维护机制是 H.264 重要也是很有特色的组成部分

**num_ref_idx_l0_active_minus1**

**num_ref_idx_l1_active_minus1**

// weighted_pred_flag 用以指明是否允许 P 和 S P 片的加权预测，如果允许，在片头会出现用以计算加权预测的句法元素。

**weighted_pred_flag**

// weighted_bipred_flag 用以指明是否允许 B 片的加权预测，本句法元素等于 0 时表示使用默认加权预测模式，等于 1 时表示使用显式加权预测模式，等于 2 时表示使用隐式加权预测模式。

**weighted_bipred_idc**

// pic_init_qp_minus26 加 26 后用以指明亮度分量的量化参数的初始值。在 H.264 中，量化参数分三个级别给出：图像参数集、片头、宏块。在图像参数集给出的是一个初始值。

**pic_init_qp_minus26** /* relative to 26 */

**pic_init_qs_minus26** /* relative to 26 */

// chroma_qp_index_offset 色度分量的量化参数是根据亮度分量的量化参数计算出来的，本句法元素用以指明计算时用到的参数。

**chroma_qp_index_offset**

// deblocking_filter_control_present_flag 编码器可以通过句法元素显式地控制去块滤波的强度，本句法元素指明是在片头是否会有句法元素传递这个控制信息。如果本句法元素等于 0，那些用于传递滤波强度的句法元素不会出现，解码器将独立地计算出滤波强度。

**deblocking_filter_control_present_flag**

// constrained_intra_pred_flag 在 P 和 B 片中，帧内编码的宏块的邻近宏块可能是采用的帧间编码。当本句法元素等于 1 时，表示帧内编码的宏块不能用帧间编码的宏块的像素作为自己的预测，即帧内编码的宏块只能用邻近帧内编码的宏块的像素作为自己的预测；而本句法元素等于 0 时，表示不存在这种限制。

**constrained_intra_pred_flag**

// redundant_pic_cnt_present_flag 指明是否会出现 redundant_pic_cnt 句法元素。

**redundant_pic_cnt_present_flag**

**rbsp_trailing_bits**( )

}

http://blog.csdn.net/xfding/article/details/5476663

# H.264 句法和语法总结（五）片头句法



h.264 算法 filterdivheadertable

## slice_header( ) {

// first_mb_in_slice 片中的第一个宏块的地址，片通过这个句法元素来标定它自己的地址。 要注意的是在帧场自适应模式下，宏块都是成对出现，这时本句法元素表示的是第几个宏块对，对应的第一个宏块的真实地址应该是 2 * first_mb_in_slice

**first_mb_in_slice**

/* slice_type 指明片的类型

| slice_type | Name of slice_type |
|---|---|
| 0 | P (P slice) |
| 1 | B (B slice) |
| 2 | I (I slice) |
| 3 | SP (SP slice) |
| 4 | SI (SI slice) |
| 5 | P (P slice) |
| 6 | B (B slice) |
| 7 | I (I slice) |
| 8 | SP (SP slice) |
| 9 | SI (SI slice) */ |

**slice_type**

// pic_parameter_set_id 图像参数集的索引号. 范围 0 到 255。

**pic_parameter_set_id**

// frame_num 每个参考帧都有一个依次连续的 frame_num 作为它们的标识,这指明了各图像的解码顺序。但事实上我们可以看到，frame_num 的出现没有 if 语句限定条件，这表明非参考帧的片头也会出现 frame_num。只是当该个图像是参考帧时，它所携带的这个句法元素在解码时才有意义。

H.264 对 frame_num 的值作了如下规定：当参数集中的句法元素

gaps_in_frame_num_value_allowed_flag 不为 1 时，每个图像的 frame_num 值是它前一个参考帧的 frame_num 值增加 1。这句话包含有两层意思：

1） 当 gaps_in_frame_num_value_allowed_flag 不为 1，即 frame_num 连续的情况下，每个图像的 frame_num 由前一个参考帧图像对应的值加 1，着重点是“前一个参考帧”。

前面我们曾经提到，对于非参考帧来说，它的 frame_num 值在解码过程中是没有意义的，因为 frame_num 值是参考帧特有的，它的主要作用是在该图像被其他图像引用作运动补偿的参考时提供一个标识。但 H.264 并没有在非参考帧图像中取消这一句法元素，原因是在 POC 的第二种和第三种解码方法中可以通过非参考帧的 frame_num 值计算出他们的 POC 值。

2） 当 gaps_in_frame_num_value_allowed_flag 等于 1，前文已经提到，这时若网络阻塞，编码器可以将编码后的若干图像丢弃，而不用另行通知解码器。在这种情况下，解码器必须有机制将缺失的 frame_num 及所对应的图像填补，否则后续图像若将运动矢量指向缺失的图像将会产生解码错误。

**frame_num**

if( !**frame_mbs_only_flag** ) {

// field_pic_flag 这是在片层标识图像编码模式的唯一一个句法元素。所谓的编码模式是指的帧编码、场编码、帧场自适应编码。当这个句法元素取值为 1 时 属于场编码； 0 时为非场编码。

**field_pic_flag**

if( **field_pic_flag** )

// bottom_field_flag 等于 1 时表示当前图像是属于底场；等于 0 时表示当前图像是属于顶场。

**bottom_field_flag**

}

if( **nal_unit_type** == 5 )

// idr_pic_id IDR 图像的标识。不同的 IDR 图像有不同的 idr_pic_id 值。值得注意的是，IDR 图像有不等价于 I 图像，只有在作为 IDR 图像的 I 帧才有这个句法元素，在场模式下，IDR 帧的两个场有相同的 idr_pic_id 值。idr_pic_id 的取值范围是 [0, 65535]，和 frame_num 类似，当它的值超出这个范围时，它会以循环的方式重新开始计数。

**idr_pic_id**

if( **pic_order_cnt_type** == 0 ) {

// pic_order_cnt_lsb 在 POC 的第一种算法中本句法元素来计算 POC 值，在 POC 的第一种算法中是显式地传递 POC 的值，而其他两种算法是通过 frame_num 来映

射 POC 的值。

**pic_order_cnt_lsb**

if( **pic_order_present_flag** && !**field_pic_flag** )

// delta_pic_order_cnt_bottom 如果是在场模式下，场对中的两个场都各自被构造为一个图像，它们有各自的 POC 算法来分别计算两个场的 POC 值，也就是一个场对拥有一对 POC 值；而在是帧模式或是帧场自适应模式下，一个图像只能根据片头的句法元素计算出一个 POC 值。根据 H.264 的规定，在序列中有可能出现场的情况，即 frame_mbs_only_flag 不为 1 时，每个帧或帧场自适应的图像在解码完后必须分解为两个场，以供后续图像中的场作为参考图像。所以当 frame_mb_only_flag 不为 1 时，帧或帧场自适应中包含的两个场也必须有各自的 POC 值。通过本句法元素，可以在已经解开的帧或帧场自适应图像的 POC 基础上新映射一个 POC 值，并把它赋给底场。当然，象句法表指出的那样，这个句法元素只用在 POC 的第一个算法中。

**delta_pic_order_cnt_bottom**

}

if( **pic_order_cnt_type** = = 1 && !**delta_pic_order_always_zero_flag** ) {

// delta_pic_order_cnt[0], delta_pic_order_cnt[1]：POC 的第二和第三种算法是从 frame_num 映射得来，这两个句法元素用于映射算法。delta_pic_order_cnt[0]用于帧编码方式下的底场和场编码方式的场，delta_pic_order_cnt[1] 用于帧编码方式下的顶场。

**delta_pic_order_cnt**[ 0 ]

if( **pic_order_present_flag** && !**field_pic_flag** )

**delta_pic_order_cnt**[ 1 ]

}

if( **redundant_pic_cnt_present_flag** )

// redundant_pic_cnt 冗余片的 id 号。

**redundant_pic_cnt**

if( **slice_type** == B )

// direct_spatial_mv_pred_flag 指出在 B 图像的直接预测的模式下，用时间预测还是用空间预测。1：空间预测；0：时间预测。

**direct_spatial_mv_pred_flag**

if( **slice_type** = = P | | **slice_type** = = SP | | **slice_type** = = B ) {

// num_ref_idx_active_override_flag 在图像参数集中我们看到已经出现句法元素 num_ref_idx_l0_active_minus1 和 num_ref_idx_l1_active_minus1 指定当前参考帧队列中实际可用的参考帧的数目。在片头可以重载这对句法元素，以给某特定图像更大的灵活度。这个句法元素就是指明片头是否会重载，如果该句法元素等于 1，下面会出现新的 num_ref_idx_l0_active_minus1 和 num_ref_idx_l1_active_minus1 值。

```
    num_ref_idx_active_override_flag
    if( num_ref_idx_active_override_flag ) {
      num_ref_idx_l0_active_minus1
      if( slice_type == B )
        num_ref_idx_l1_active_minus1
    }
  }
  // 参考帧队列重排序（reordering）句法
  ref_pic_list_reordering( )
  if( ( weighted_pred_flag && ( slice_type == P || slice_type == SP ) ) ||
    ( weighted_bipred_idc == 1 && slice_type == B ) )
    // 加权预测句法
    pred_weight_table( )
  if( nal_ref_idc != 0 )
    // 参考帧队列标记(marking)句法
    dec_ref_pic_marking( )
  if( entropy_coding_mode_flag && slice_type != I && slice_type != SI
)
    // cabac_init_idc 给出 cabac 初始化时表格的选择，范围 0 到 2。
    cabac_init_idc
  // slice_qp_delta 指出在用于当前片的所有宏块的量化参数的初始值。SliceQPY = 26+
pic_init_qp_minus26 + slice_qp_delta 范围是 0 to 51。 H.264 中量化参数是分图像参
数集、片头、宏块头三层给出的，前两层各自给出一个偏移值，这个句法元素就是片层的偏
移。
  slice_qp_delta
  if( slice_type == SP || slice_type == SI ) {
    if( slice_type == SP )
      // sp_for_switch_flag 指出 SP 帧中的 p 宏块的解码方式是否是 switching 模式
      sp_for_switch_flag
      // slice_qs_delta 与 slice_qp_delta 的与语义相似，用在 SI 和 SP 中的
      slice_qs_delta
  }
  if( deblocking_filter_control_present_flag ) {
    // disable_deblocking_filter_idc H.264 指定了一套算法可以在解码器端独立地计算
图像中各边界的滤波强度进行滤波。除了解码器独立计算之外，编码器也可以传递句法元素
```

来干涉滤波强度，当这个句法元素指定了在块的边界是否要用滤波，同时指明那个块的边界不用块滤波

```
        disable_deblocking_filter_idc
        if( disable_deblocking_filter_idc    !=    1 ) {
          // slice_alpha/beta_c0_offset_div2  给出用于增强  α/beta  和  t C0  的偏移值
          slice_alpha_c0_offset_div2
          slice_beta_offset_div2
        }
    }
    if( num_slice_groups_minus1 > 0    &&
        slice_group_map_type >= 3    &&    slice_group_map_type <= 5)
        // slice_group_change_cycle  当片组的类型是  3, 4,    5，由句法元素可获得片组中  映射单元的数目：
        slice_group_change_cycle
}
```

http://blog.csdn.net/xfding/article/details/5476763

# H.264 句法和语法总结（六）参考帧队列重排序（reordering）句法



h.264list

## ref_pic_list_reordering( ) {

```
    if( slice_type    !=    I    &&    slice_type    !=    SI ) {
      // ref_pic_list_reordering_flag_l0  指明是否进行重排序操作,这个句法元素等于 1 时表明紧跟着会有一系列句法元素用于参考帧队列的重排序。
      ref_pic_list_reordering_flag_l0
      if( ref_pic_list_reordering_flag_l0 )
        do {
          // reordering_of_pic_nums_idc      指明执行哪种重排序操作
            reordering_of_pic_nums_idc      操作
            0                               短期参考帧重排序，
```

abs_diff_pic_num_minus1 会出现在码流中，从当
前图像的 PicNum 减
去 (abs_diff_pic_num_minus1 + 1) 后指明需要重
排序的图像。
1 短期参考帧重排序，
abs_diff_pic_num_minus1 会出现在码流中，从当
前图像的 PicNum 加
上 (abs_diff_pic_num_minus1 + 1) 后指明需要重
排序的图像。
2 长期参考帧重排序，long_term_pic_num 会
出现在码流中，指明需要重
排序的图像。
3 结束循环，退出重排序操作。

```
reordering_of_pic_nums_idc
 if( reordering_of_pic_nums_idc == 0 | |
   reordering_of_pic_nums_idc == 1 )
   // abs_diff_pic_num_minus1 在对短期参考帧重排序时指明重排序图像与当
前的差
   abs_diff_pic_num_minus1
 else if( reordering_of_pic_nums_idc = = 2 )
   // long_term_pic_num 在对长期参考帧重排序时指明重排序图像
   long_term_pic_num
 } while( reordering_of_pic_nums_idc != 3 )
}
if( slice_type == B ) {
 ref_pic_list_reordering_flag_l1
  if( ref_pic_list_reordering_flag_l1 )
   do {
    reordering_of_pic_nums_idc
     if( reordering_of_pic_nums_idc = = 0 | |
       reordering_of_pic_nums_idc = = 1 )
      abs_diff_pic_num_minus1
     else if( reordering_of_pic_nums_idc = = 2 )
       long_term_pic_num
   } while( reordering_of_pic_nums_idc != 3 )
```

```
    }
}
```

http://blog.csdn.net/xfding/article/details/5476851

# H.264 句法和语法总结（七）加权预测句法



h.264table

## pred_weight_table( ) {

// luma_log2_weight_denom 给出参考帧列表中参考图像所有亮度的加权系数, 是个初始值 luma_log2_weight_denom 值的范围是 0 to 7。

**luma_log2_weight_denom**

// chroma_log2_weight_denom 给出参考帧列表中参考图像所有色度的加权系数，是个初始值 chroma_log2_weight_denom 值的范围是 0 to 7。

**chroma_log2_weight_denom**

```
for( i = 0; i <= num_ref_idx_l0_active_minus1; i++ ) {
```

// luma_weight_l0_flag 等于 1 时，指的是在参考序列 0 中的亮度的加权系数存在；等于 0 时，在参考序列 0 中的亮度的加权系数不存在。

**luma_weight_l0_flag**

```
if( luma_weight_l0_flag ) {
```

// luma_weight_l0[ i ] 用参考序列 0 预测亮度值时，所用的加权系数。如果 luma_weight_l0_flag is = 0, luma_weight_l0[ i ] = pow(2, luma_log2_weight_denom )

**luma_weight_l0**[ i ]

// luma_offset_l0[ i ] 用参考序列 0 预测亮度值时，所用的加权系数的额外的偏移。luma_offset_l0[ i ] 值的范围–128 to 127。如果 luma_weight_l0_flag is = 0, luma_offset_l0[ i ] = 0

**luma_offset_l0**[ i ]

```
}
```

**chroma_weight_l0_flag**

```
if( chroma_weight_l0_flag )
    for( j =0; j < 2; j++ ) {
        chroma_weight_l0[ i ][ j ]
        chroma_offset_l0[ i ][ j ]
    }
}
```

```
    if( slice_type    = =    B )
      for( i = 0; i <= num_ref_idx_l1_active_minus1; i++ ) {
        luma_weight_l1_flag
        if( luma_weight_l1_flag ) {
          luma_weight_l1[ i ]
          luma_offset_l1[ i ]
        }
        chroma_weight_l1_flag
        if( chroma_weight_l1_flag )
          for( j = 0; j < 2; j++ ) {
            chroma_weight_l1[ i ][ j ]
            chroma_offset_l1[ i ][ j ]
          }
      }
}
```

http://blog.csdn.net/xfding/article/details/5476906

# H.264 句法和语法总结（八）参考图像序列标记（marking)操作的语义

2010-04-12 15:54 1263 人阅读 评论(0) 收藏 举报

h.264referenceoutput

重排序（reordering）操作是对参考帧队列重新排序，而标记（marking）操作负责将参考图像移入或移出参考帧队列。

## dec_ref_pic_marking( ) {

```
    if( nal_unit_type    = =    5 ) {
```

// no_output_of_prior_pics_flag 仅在当前图像是 IDR 图像时出现这个句法元素，指明是否要将前面已解码的图像全部输出。

**no_output_of_prior_pics_flag**

// long_term_reference_flag 与上个图像一样，仅在当前图像是 IDR 图像时出现这一句法元素。这个句法元素指明是否使用长期参考这个机制。如果取值为 1，表明使用长

期参考，并且每个 IDR 图像被解码后自动成为长期参考帧，否则（取值为 0），IDR 图像被解码后自动成为短期参考帧。

**long_term_reference_flag**

} else {

// adaptive_ref_pic_marking_mode_flag 指明标记（marking）操作的模式，

adaptive_ref_pic_marking_mode_flag 标记（marking）模式

0 先入先出（FIFO）：使用滑动窗的机制，先入先出，在这种模式下没有办法对长期参考帧进行操作。

1 自适应标记（marking）：后续码流中会有一系列句法元素显式指明操作的步骤。自适应是指编码器可根据情况随机灵活地作出决策。

**adaptive_ref_pic_marking_mode_flag**

if( **adaptive_ref_pic_marking_mode_flag** )

do {

/* memory_management_control_operation 在自适应标记（marking）模式中，指明本次操作的具体内容

memory_management_control_operation 标记（marking）操作

0 结束循环，退出标记（marding）操作。

1 将一个短期参考图像标记为非参考图像，也即将一个短期参考图像移出参考帧队列。

2 将一个长期参考图像标记为非参考图像，也即将一个长期参考图像移出参考帧队列。

3 将一个短期参考图像转为长期参考图像。

4 指明长期参考帧的最大数目。

5 清空参考帧队列，将所有参考图像移出参考帧队列，并禁用长期参考机制

```
            6                                   将当前图像存为一个长期参考
帧。 */
            memory_management_control_operation
            if( memory_management_control_operation  = =  1  | |
              memory_management_control_operation  = =  3 )
              // difference_of_pic_nums_minus1  当
memory_management_control_operation 等于 3 或 1 时，由 这个
句法元素可以计算得到需要操作的图像在短期参考队列中的序号。参考帧队列中必须存在这
个图像。
              difference_of_pic_nums_minus1
            if(memory_management_control_operation  = =  2  )
              // long_term_pic_num  当 memory_management_control_operation 等于
2 时， 从此句法元素得到所要
操作的长期参考图像的序号。
              long_term_pic_num
            if( memory_management_control_operation  = =  3  | |
              memory_management_control_operation  = =  6 )
              // long_term_frame_idx  当 memory_management_control_operation 等
于 3 或 6 ，分配一个长期参考
帧的序号给一个图像。
              long_term_frame_idx
            if( memory_management_control_operation  = =  4 )
              // max_long_term_frame_idx_plus1  此句法元素减 1，指明长期参考队列的最
大数目 。max_long_term_frame_idx_plus1 值的范围 0 to num_ref_frames。
              max_long_term_frame_idx_plus1
          } while( memory_management_control_operation  !=  0 )
    }
}
```

http://blog.csdn.net/xfding/article/details/5477026

# H.264 句法和语法总结（九）片层数据句法

2010-04-12 16:11 1295 人阅读 评论(0) 收藏 举报

h.264alignmentbytelayer

## slice_data( ) {

```
    if( entropy_coding_mode_flag )
```

while( !byte_aligned( ) )

// cabac_alignment_one_bit 当熵编码模式是 CABAC 时,此时要求数据字节对齐,即数据从下一个字节的第一个比特开始,如果还没有字节对齐将出现若干个 cabac_alignment_one_bit 作为填充。

**cabac_alignment_one_bit**

CurrMbAddr = first_mb_in_slice * ( 1 + MbaffFrameFlag )

moreDataFlag = 1

prevMbSkipped = 0

do {

if( slice_type != I && slice_type != SI )

if( !entropy_coding_mode_flag ) {

// mb_skip_run 当图像采用帧间预测编码时，H.264 允许在图像平坦的区域使用“跳跃”块，“跳跃”块本身不携带任何数据，解码器通过周围已重建的宏块的数据来恢复“跳跃”块。当熵编码为 CAVLC 或 CABAC 时，“跳跃”块的表示方法不同。当 entropy_coding_mode_flag 为 1，即熵编码为 CABAC 时，是每个“跳跃”块都会有句法元素 mb_skip_flag 指明，而 entropy_coding_mode_flag 等于 0，即熵编码为 CAVLC 时，用一种行程的方法给出紧连着的“跳跃”块的数目，即句法元素 mb_skip_run。mb_skip_run 值的范围 0 to PicSizeInMbs – CurrMbAddr 。

**mb_skip_run**

prevMbSkipped = ( mb_skip_run > 0 )

for( i=0; i

CurrMbAddr = NextMbAddress( CurrMbAddr )

moreDataFlag = more_rbsp_data( )

} else {

// mb_skip_flag 指明当前宏块是否是跳跃编码模式的宏块

**mb_skip_flag**

moreDataFlag = !mb_skip_flag

}

if( moreDataFlag ) {

if( MbaffFrameFlag && ( CurrMbAddr % 2 = = 0 | |

( CurrMbAddr % 2 = = 1 && prevMbSkipped ) ) )

// mb_field_decoding_flag 在帧场自适应图像中，指明当前宏块所属的宏块对是帧模式还是场模式。0 帧模式；1 场模式。如果一个宏块对的两个宏块句法结构中都没有出现这个句法元素，即它们都是“跳跃”块时，本句法元素由以下决定：

- 如果这个宏块对与相邻的、左边的宏块对属于同一个片时，这个宏块对的

mb_field_decoding_flag 的值等于左边的宏块对的 mb_field_decoding_flag 的值。

- 否则，这个宏块对的 mb_field_decoding_flag 的值等于上边同属于一个片的宏块对的 mb_field_decoding_flag 的值。 - 如果这个宏块对既没有相邻的、上边同属于一个片的宏块对；也没有相邻的、左边同属于一个片的宏块对，这个宏块对的 mb_field_decoding_flag 的值等于 0，即帧模式。 end_of_slice_flag 指明是否到了片的结尾。

**mb_field_decoding_flag**

**macroblock_layer( )**

```
    }
    if( !entropy_coding_mode_flag )
      moreDataFlag = more_rbsp_data( )
    else {
      if( slice_type != I && slice_type != SI )
        prevMbSkipped = mb_skip_flag
      if( MbaffFrameFlag && CurrMbAddr % 2 = = 0 )
        moreDataFlag = 1
      else {
        end_of_slice_flag
        moreDataFlag = !end_of_slice_flag
      }
    }
    CurrMbAddr = NextMbAddress( CurrMbAddr )
  } while( moreDataFlag )
}
```

http://blog.csdn.net/xfding/article/details/5477351

# H.264 句法和语法总结（十一）宏块层预测句法

2010-04-12 17:01 913 人阅读 评论(0) 收藏 举报

h.264

```
mb_pred( mb_type ) {
  if( MbPartPredMode( mb_type, 0 ) = = Intra_4x4 | |
    MbPartPredMode( mb_type, 0 ) = = Intra_16x16 ) {
```

if( MbPartPredMode( mb_type, 0 ) == Intra_4x4 )

for( luma4x4BlkIdx=0; luma4x4BlkIdx<16; luma4x4BlkIdx++ ) {

// prev_intra4x4_pred_mode_flag[ luma4x4BlkIdx ] rem_intra4x4_pred_mode[ luma4x4BlkIdx ] 帧内预测的模式也是需要预测的， prev_intra4x4_pred_mode_flag 用来指明帧内预测时，亮度分量的预测模式的预测值是否就是真实预测模式，如果是，就不需另外再传预测模式。如果不是，就由 rem_intra4x4_pred_mode 指定真实预测模式。

**prev_intra4x4_pred_mode_flag**[ luma4x4BlkIdx ]

if( !**prev_intra4x4_pred_mode_flag**[ luma4x4BlkIdx ] )

**rem_intra4x4_pred_mode**[ luma4x4BlkIdx ]

}

// intra_chroma_pred_mode 在帧内预测时指定色度的预测模式，

| intra_chroma_pred_mode | 预测模式 |
|---|---|
| 0 | DC |
| 1 | Horizontal |
| 2 | Vertical |
| 3 | Plane |

**intra_chroma_pred_mode**

} else if( MbPartPredMode( mb_type, 0 ) != Direct ) {

for( mbPartIdx = 0; mbPartIdx < NumMbPart( mb_type ); mbPartIdx++)

if( ( num_ref_idx_l0_active_minus1 > 0 | |

mb_field_decoding_flag ) &&

MbPartPredMode( mb_type, mbPartIdx ) != Pred_L1 )

// ref_idx_l0[ mbPartIdx]用参考帧队列 L0 进行预测，即前向预测时，参考图像在参考帧队列中的序号。其中 mbPartIdx 是宏块分区的序号。 如 果 当 前 宏 块 是非场宏块 ， 则 ref_idx_l0[ mbPartIdx ] 值的范围是 0 到 num_ref_idx_l0_active_minus1。否则，如果当前宏块是场宏块，（宏块所在图像是场，当图像是帧场自适应时当前宏块处于场编码的宏块对），ref_idx_l0[ mbPartIdx]值的范围是 0 到 2*num_ref_idx_l0_active_minus1 + 1，如前所述，此时参考帧队列的帧都将拆成场，故参考队列长度加倍。

**ref_idx_l0**[ mbPartIdx ]

for( mbPartIdx = 0; mbPartIdx < NumMbPart( mb_type ); mbPartIdx++)

if( ( num_ref_idx_l1_active_minus1 > 0 | |

mb_field_decoding_flag ) &&

MbPartPredMode( mb_type, mbPartIdx ) != Pred_L0 )

**ref_idx_l1**[ mbPartIdx ]

for( mbPartIdx = 0; mbPartIdx < NumMbPart( mb_type ); mbPartIdx++)

if( MbPartPredMode ( mb_type, mbPartIdx ) != Pred_L1 )

for( compIdx = 0; compIdx < 2; compIdx++ )

// mvd_l0[ mbPartIdx ][ 0 ][ compIdx ] 运动矢量的预测值和实际值之间的差。mbPartIdx 是宏块分区的序号。CompIdx = 0 时水平运动矢量； CompIdx = 1 垂直运动矢量。

**mvd_l0**[ mbPartIdx ][ 0 ][ compIdx ]

for( mbPartIdx = 0; mbPartIdx < NumMbPart( mb_type ); mbPartIdx++)

if( MbPartPredMode( mb_type, mbPartIdx ) != Pred_L0 )

for( compIdx = 0; compIdx < 2; compIdx++ )

**mvd_l1**[ mbPartIdx ][ 0 ][ compIdx ]

}

}

http://blog.csdn.net/xfding/article/details/5477437

# H.264 句法和语法总结（十二）子宏块预测句法



h.264

## sub_mb_pred( mb_type ) {

for( mbPartIdx = 0; mbPartIdx < 4; mbPartIdx++ )

// sub_mb_type[ mbPartIdx ] 指明子宏块的预测类型，在不同的宏块类型中这个句法元素的语义不一样。

**sub_mb_type**[ mbPartIdx ]

for( mbPartIdx = 0; mbPartIdx < 4; mbPartIdx++ )

if( ( num_ref_idx_l0_active_minus1 > 0 | | mb_field_decoding_flag ) &&

mb_type != P_8x8ref0 &&

sub_mb_type[ mbPartIdx ] != B_Direct_8x8 &&

SubMbPredMode( sub_mb_type[ mbPartIdx ] ) != Pred_L1 )

**ref_idx_l0**[ mbPartIdx ]

for( mbPartIdx = 0; mbPartIdx < 4; mbPartIdx++ )

if( (num_ref_idx_l1_active_minus1 > 0 | | mb_field_decoding_flag )

```
&&
            sub_mb_type[ mbPartIdx ]    !=    B_Direct_8x8    &&
            SubMbPredMode( sub_mb_type[ mbPartIdx ] )    !=    Pred_L0 )
        ref_idx_l1[ mbPartIdx ]
    for( mbPartIdx = 0; mbPartIdx < 4; mbPartIdx++ )
      if( sub_mb_type[ mbPartIdx ]    !=    B_Direct_8x8    &&
        SubMbPredMode( sub_mb_type[ mbPartIdx ] )    !=    Pred_L1 )
        for( subMbPartIdx = 0;
              subMbPartIdx < NumSubMbPart( sub_mb_type[ mbPartIdx ] );
              subMbPartIdx++)
          for( compIdx = 0; compIdx < 2; compIdx++ )
            mvd_l0[ mbPartIdx ][ subMbPartIdx ][ compIdx ]
    for( mbPartIdx = 0; mbPartIdx < 4; mbPartIdx++ )
      if( sub_mb_type[ mbPartIdx ]    !=    B_Direct_8x8    &&
        SubMbPredMode( sub_mb_type[ mbPartIdx ] )    !=    Pred_L0 )
        for( subMbPartIdx = 0;
              subMbPartIdx < NumSubMbPart( sub_mb_type[ mbPartIdx ] );
              subMbPartIdx++)
          for( compIdx = 0; compIdx < 2; compIdx++ )
            mvd_l1[ mbPartIdx ][ subMbPartIdx ][ compIdx ]
}
```

http://blog.csdn.net/xfding/article/details/5477442

# H.264 句法和语法总结（十三）残差句法



h.264

```
residual( ) {
    if( !entropy_coding_mode_flag )
      residual_block = residual_block_cavlc
    else
      residual_block = residual_block_cabac
    if( MbPartPredMode( mb_type, 0 )    = =    Intra_16x16 )
      residual_block( Intra16x16DCLevel, 16 )
```

```
    for( i8x8 = 0; i8x8 < 4; i8x8++ )     /* each luma 8x8 block */
      for( i4x4 = 0; i4x4 < 4; i4x4++ )     /* each 4x4 sub-block of block */
        if( CodedBlockPatternLuma & ( 1 << i8x8 ) ) {
          if( MbPartPredMode( mb_type, 0 )     = =     Intra_16x16 )
            residual_block( Intra16x16ACLevel[ i8x8 * 4 + i4x4 ], 15 )
          else
            residual_block( LumaLevel[ i8x8 * 4 + i4x4 ], 16 )
        } else {
          if( MbPartPredMode( mb_type, 0 )     = =     Intra_16x16 )
            for( i = 0; i < 15; i++ )
              Intra16x16ACLevel[ i8x8 * 4 + i4x4 ][ i ] = 0
          else
            for( i = 0; i < 16; i++ )
              LumaLevel[ i8x8 * 4 + i4x4 ][ i ] = 0
        }
    for( iCbCr = 0; iCbCr < 2; iCbCr++ )
      if( CodedBlockPatternChroma & 3 )     /* chroma DC residual present */
        residual_block( ChromaDCLevel[ iCbCr ], 4 )
      else
        for( i = 0; i < 4; i++ )
          ChromaDCLevel[ iCbCr ][ i ] = 0
    for( iCbCr = 0; iCbCr < 2; iCbCr++ )
      for( i4x4 = 0; i4x4 < 4; i4x4++ )
        if( CodedBlockPatternChroma & 2 )
          /* chroma AC residual present */
          residual_block( ChromaACLevel[ iCbCr ][ i4x4 ], 15 )
        else
          for( i = 0; i < 15; i++ )
            ChromaACLevel[ iCbCr ][ i4x4 ][ i ] = 0
}
```

http://blog.csdn.net/xfding/article/details/5477464

# H.264 句法和语法总结（十四）CAVLC 残差句法

2010-04-12 17:19 695 人阅读 评论(0) 收藏 举报

h.264tokendescriptor

```
residual_block_cavlc( coeffLevel, maxNumCoeff ) {   C      Descriptor
    for( i = 0; i < maxNumCoeff; i++ )
        coeffLevel[ i ] = 0
   // coeff_token      指明了非零系数的个数，拖尾系数的个数。
   coeff_token
    if( TotalCoeff( coeff_token ) > 0 ) {
        if( TotalCoeff( coeff_token ) > 10    &&    TrailingOnes( coeff_token ) <
3 )
            suffixLength = 1
        else
            suffixLength = 0
        for( i = 0; i < TotalCoeff( coeff_token ); i++ )
            if( i < TrailingOnes( coeff_token ) ) {
                // trailing_ones_sign_flag   拖尾系数的符号
                    -    如果 trailing_ones_sign_flag = 0，相应的拖尾系数是+1。
                    -    否则，trailing_ones_sign_flag =1，相应的拖尾系数是-1。
                trailing_ones_sign_flag
                level[ i ] = 1 – 2 * trailing_ones_sign_flag
            } else {
                // level_prefix and level_suffix   非零系数值的前缀和后缀。
                level_prefix
                levelCode = ( level_prefix << suffixLength )
                if( suffixLength > 0    | |    level_prefix >= 14 ) {
                    level_suffix
                    levelCode += level_suffix
                }
                if( level_prefix    = =    15    &&    suffixLength    = =    0 )
                    levelCode += 15
                if( i    = =    TrailingOnes( coeff_token )    &&
                    TrailingOnes( coeff_token ) < 3 )
                    levelCode += 2
```

```
            if( levelCode % 2     = =     0 )
               level[ i ] = ( levelCode + 2 ) >> 1
            else
               level[ i ] = ( –levelCode – 1 ) >> 1
            if( suffixLength     = =     0 )
               suffixLength = 1
            if( Abs( level[ i ] )     >     ( 3 << ( suffixLength – 1 ) )     &&
                suffixLength < 6 )
               suffixLength++
         }
      if( TotalCoeff( coeff_token ) < maxNumCoeff ) {
         // total_zeros     系数中 0 的总个数。
         total_zeros
         zerosLeft = total_zeros
      } else
         zerosLeft = 0
      for( i = 0; i < TotalCoeff( coeff_token ) – 1; i++ ) {
         if( zerosLeft > 0 ) {

            run_before
            run[ i ] = run_before
         } else
            run[ i ] = 0
         zerosLeft = zerosLeft – run[ i ]
      }
      run[ TotalCoeff( coeff_token ) – 1 ] = zerosLeft
      coeffNum = -1
      for( i = TotalCoeff( coeff_token ) – 1; i >= 0; i-- ) {
         coeffNum += run[ i ] + 1
         coeffLevel[ coeffNum ] = level[ i ]
      }
   }
}
```