原 x264源代码简单分析：编码器主干部分-1

===

H.264源代码分析文章列表：

【编码 - x264】

【解码 - libavcodec H.264 解码器】

===

本文分析x264编码器主干部分的源代码。“主干部分”指的就是libx264中最核心的接口函数——x264_encoder_encode()，以及相关的几个接口函数x264_encoder_open()，x264_encoder_headers()，和x264_encoder_close()。这一部分源代码比较复杂，现在看了半天依然感觉很多地方不太清晰，暂且把已经理解的地方整理出来，以后再慢慢补充还不太清晰的地方。由于主干部分内容比较多，因此打算分成两篇文章来记录：第一篇文章记录x264_encoder_open()，x264_encoder_headers()，和x264_encoder_close()这三个函数，第二篇文章记录x264_encoder_encode()函数。

函数调用关系图

X264编码器主干部分的源代码在整个x264中的位置如下图所示。

单击查看更清晰的图片

X264编码器主干部分的函数调用关系如下图所示。

单击查看更清晰的图片

从图中可以看出，x264主干部分最复杂的函数就是x264_encoder_encode()，该函数完成了编码一帧YUV为H.264码流的工作。与之配合的还有打开编码器的函数x264_encoder_open()，关闭编码器的函数x264_encoder_close()，以及输出SPS/PPS/SEI这样的头信息的x264_encoder_headers()。

x264_encoder_open()用于打开编码器，其中初始化了libx264编码所需要的各种变量。它调用了下面的函数：
x264_validate_parameters()：检查输入参数（例如输入图像的宽高是否为正数）。
x264_predict_16x16_init()：初始化Intra16x16帧内预测汇编函数。
x264_predict_4x4_init()：初始化Intra4x4帧内预测汇编函数。
x264_pixel_init()：初始化像素值计算相关的汇编函数（包括SAD、SATD、SSD等）。
x264_dct_init()：初始化DCT变换和DCT反变换相关的汇编函数。
x264_mc_init()：初始化运动补偿相关的汇编函数。
x264_quant_init()：初始化量化和反量化相关的汇编函数。
x264_deblock_init()：初始化去块效应滤波器相关的汇编函数。
x264_lookahead_init()：初始化Lookahead相关的变量。
x264_ratecontrol_new()：初始化码率控制相关的变量。

x264_encoder_headers()输出SPS/PPS/SEI这些H.264码流的头信息。它调用了下面的函数：
x264_sps_write()：输出SPS
x264_pps_write()：输出PPS
x264_sei_version_write()：输出SEI

x264_encoder_encode()编码一帧YUV为H.264码流。它调用了下面的函数：
x264_frame_pop_unused()：获取1个x264_frame_t类型结构体fenc。如果frames.unused[]队列不为空，就调用x264_frame_pop()从unused[]队列取1个现成的；否则就调用x264_frame_new()创建一个新的。
x264_frame_copy_picture()：将输入的图像数据拷贝至fenc。
x264_lookahead_put_frame()：将fenc放入lookahead.next.list[]队列，等待确定帧类型。
x264_lookahead_get_frames()：通过lookahead分析帧类型。该函数调用了x264_slicetype_decide()，x264_slicetype_analyse()和x264_slicetype_frame_cost()等函数。经过一些列分析之后，最终确定了帧类型信息，并且将帧放入frames.current[]队列。
x264_frame_shift()：从frames.current[]队列取出1帧用于编码。
x264_reference_update()：更新参考帧列表。
x264_reference_reset()：如果为IDR帧，调用该函数清空参考帧列表。
x264_reference_hierarchy_reset()：如果是I（非IDR帧）、P帧、B帧（可做为参考帧），调用该函数。
x264_reference_build_list()：创建参考帧列表list0和list1。
x264_ratecontrol_start()：开启码率控制。
x264_slice_init()：创建 Slice Header。
x264_slices_write()：编码数据（最关键的步骤）。其中调用了x264_slice_write()完成了编码的工作（注意“x264_slices_write()”和“x264_slice_write()”名字差了一个“s”）。
x264_encoder_frame_end()：编码结束后做一些后续处理，例如记录一些统计信息。其中调用了x264_frame_push_unused()将fenc重新放回frames.unused[]队列，并且调用x264_ratecontrol_end()关闭码率控制。

x264_encoder_close()用于关闭解码器，同时输出一些统计信息。它调用了下面的函数：
x264_lookahead_delete()：释放Lookahead相关的变量。
x264_ratecontrol_summary()：汇总码率控制信息。
x264_ratecontrol_delete()：关闭码率控制。

本文将会记录x264_encoder_open()，x264_encoder_headers()，和x264_encoder_close()这三个函数的源代码。下一篇文章记录x264_encoder_encode()函数。

x264_encoder_open()

x264_encoder_open()是一个libx264的API。该函数用于打开编码器，其中初始化了libx264编码所需要的各种变量。该函数的声明如下所示。

[cpp]

```cpp
/* x264_encoder_open:
 *      create a new encoder handler, all parameters from x264_param_t are copied */
x264_t *x264_encoder_open( x264_param_t * );
```

x264_encoder_open()的定义位于encoder\encoder.c，如下所示。

[cpp]

```cpp
/******************************************************************************
 * x264_encoder_open:
```

```
 * 注释和处理：雷霄骅
 * http://blog.csdn.net/leixiaohua1020
 * leixiaohua1020@126.com
 ******************************************************************************/
//打开编码器
x264_t *x264_encoder_open( x264_param_t *param )
{
    x264_t *h;
    char buf[1000], *p;
    int qp, i_slicetype_length;

    CHECKED_MALLOCZERO( h, sizeof(x264_t) );

    /* Create a copy of param */
    //将参数拷贝进来
    memcpy( &h->param, param, sizeof(x264_param_t) );

    if( param->param_free )
        param->param_free( param );

    if( x264_threading_init() )
    {
        x264_log( h, X264_LOG_ERROR, "unable to initialize threading\n" );
        goto fail;
    }
    //检查输入参数
    if( x264_validate_parameters( h, 1 ) < 0 )
        goto fail;

    if( h->param.psz_cqm_file )
        if( x264_cqm_parse_file( h, h->param.psz_cqm_file ) < 0 )
            goto fail;

    if( h->param.rc.psz_stat_out )
        h->param.rc.psz_stat_out = strdup( h->param.rc.psz_stat_out );
    if( h->param.rc.psz_stat_in )
        h->param.rc.psz_stat_in = strdup( h->param.rc.psz_stat_in );

    x264_reduce_fraction( &h->param.i_fps_num, &h->param.i_fps_den );
    x264_reduce_fraction( &h->param.i_timebase_num, &h->param.i_timebase_den );

    /* Init x264_t */
    h->i_frame = -1;
    h->i_frame_num = 0;

    if( h->param.i_avcintra_class )
        h->i_idr_pic_id = 5;
    else
        h->i_idr_pic_id = 0;

    if( (uint64_t)h->param.i_timebase_den * 2 > UINT32_MAX )
    {
        x264_log( h, X264_LOG_ERROR, "Effective timebase denominator %u exceeds H.264 maximum\n", h->param.i_timebase_den );
        goto fail;
    }

    x264_set_aspect_ratio( h, &h->param, 1 );
    //初始化SPS和PPS
    x264_sps_init( h->sps, h->param.i_sps_id, &h->param );
    x264_pps_init( h->pps, h->param.i_sps_id, &h->param, h->sps );
    //检查级Level-通过宏块个数等等
    x264_validate_levels( h, 1 );

    h->chroma_qp_table = i_chroma_qp_table + 12 + h->pps->i_chroma_qp_index_offset;

    if( x264_cqm_init( h ) < 0 )
        goto fail;
    //各种赋值
    h->mb.i_mb_width = h->sps->i_mb_width;
    h->mb.i_mb_height = h->sps->i_mb_height;
    h->mb.i_mb_count = h->mb.i_mb_width * h->mb.i_mb_height;

    h->mb.chroma_h_shift = CHROMA_FORMAT == CHROMA_420 || CHROMA_FORMAT == CHROMA_422;
    h->mb.chroma_v_shift = CHROMA_FORMAT == CHROMA_420;

    /* Adaptive MBAFF and subme 0 are not supported as we require halving motion
     * vectors during prediction, resulting in hpel mvs.
     * The chosen solution is to make MBAFF non-adaptive in this case. */
    h->mb.b_adaptive_mbaff = PARAM_INTERLACED && h->param.analyse.i_subpel_refine;

    /* Init frames. */
    if( h->param.i_bframe_adaptive == X264_B_ADAPT_TRELLIS && !h->param.rc.b_stat_read )
        h->frames.i_delay = X264_MAX(h->param.i_bframe,3)*4;
    else
        h->frames.i_delay = h->param.i_bframe;
    if( h->param.rc.b_mb_tree || h->param.rc.i_vbv_buffer_size )
        h->frames.i_delay = X264_MAX( h->frames.i_delay, h->param.rc.i_lookahead );
    i_slicetype_length = h->frames.i_delay;
    h->frames.i_delay += h->i_thread_frames - 1;
    h->frames.i_delay += h->param.i_sync_lookahead;
    h->frames.i_delay += h->param.b_vfr_input;
```

```
    h->frames.i_bframe_delay = h->param.i_bframe ? (h->param.i_bframe_pyramid ? 2 : 1) : 0;

    h->frames.i_max_ref0 = h->param.i_frame_reference;
    h->frames.i_max_ref1 = X264_MIN( h->sps->vui.i_num_reorder_frames, h->param.i_frame_reference );
    h->frames.i_max_dpb  = h->sps->vui.i_max_dec_frame_buffering;
    h->frames.b_have_lowres = !h->param.rc.b_stat_read
        && ( h->param.rc.i_rc_method == X264_RC_ABR
          || h->param.rc.i_rc_method == X264_RC_CRF
          || h->param.i_bframe_adaptive
          || h->param.i_scenecut_threshold
          || h->param.rc.b_mb_tree
          || h->param.analyse.i_weighted_pred );
    h->frames.b_have_lowres |= h->param.rc.b_stat_read && h->param.rc.i_vbv_buffer_size > 0;
    h->frames.b_have_sub8x8_esa = !!(h->param.analyse.inter & X264_ANALYSE_PSUB8x8);

    h->frames.i_last_idr =
    h->frames.i_last_keyframe = - h->param.i_keyint_max;
    h->frames.i_input    = 0;
    h->frames.i_largest_pts = h->frames.i_second_largest_pts = -1;
    h->frames.i_poc_last_open_gop = -1;
    //CHECKED_MALLOCZERO(var, size)
    //调用malloc()分配内存,然后调用memset()置零
    CHECKED_MALLOCZERO( h->frames.unused[0], (h->frames.i_delay + 3) * sizeof(x264_frame_t *) );
    /* Allocate room for max refs plus a few extra just in case. */
    CHECKED_MALLOCZERO( h->frames.unused[1], (h->i_thread_frames + X264_REF_MAX + 4) * sizeof(x264_frame_t *) );
    CHECKED_MALLOCZERO( h->frames.current, (h->param.i_sync_lookahead + h->param.i_bframe
                        + h->i_thread_frames + 3) * sizeof(x264_frame_t *) );
    if( h->param.analyse.i_weighted_pred > 0 )
        CHECKED_MALLOCZERO( h->frames.blank_unused, h->i_thread_frames * 4 * sizeof(x264_frame_t *) );
    h->i_ref[0] = h->i_ref[1] = 0;
    h->i_cpb_delay = h->i_coded_fields = h->i_disp_fields = 0;
    h->i_prev_duration = ((uint64_t)h->param.i_fps_den * h->sps->vui.i_time_scale) / ((uint64_t)h->param.i_fps_num * h->sps->vui.i_num_units_in_tick);
    h->i_disp_fields_last_frame = -1;
    //RDO初始化
    x264_rdo_init();

    /* init CPU functions */
    //初始化包含汇编优化的函数
    //帧内预测
    x264_predict_16x16_init( h->param.cpu, h->predict_16x16 );
    x264_predict_8x8c_init( h->param.cpu, h->predict_8x8c );
    x264_predict_8x16c_init( h->param.cpu, h->predict_8x16c );
    x264_predict_8x8_init( h->param.cpu, h->predict_8x8, &h->predict_8x8_filter );
    x264_predict_4x4_init( h->param.cpu, h->predict_4x4 );
    //SAD等和像素计算有关的函数
    x264_pixel_init( h->param.cpu, &h->pixf );
    //DCT
    x264_dct_init( h->param.cpu, &h->dctf );
    //“之”字扫描
    x264_zigzag_init( h->param.cpu, &h->zigzagf_progressive, &h->zigzagf_interlaced );
    memcpy( &h->zigzagf, PARAM_INTERLACED ? &h->zigzagf_interlaced : &h->zigzagf_progressive, sizeof(h->zigzagf) );
    //运动补偿
    x264_mc_init( h->param.cpu, &h->mc, h->param.b_cpu_independent );
    //量化
    x264_quant_init( h, h->param.cpu, &h->quantf );
    //去块效应滤波
    x264_deblock_init( h->param.cpu, &h->loopf, PARAM_INTERLACED );
    x264_bitstream_init( h->param.cpu, &h->bsf );
    //初始化CABAC或者是CAVLC
    if( h->param.b_cabac )
        x264_cabac_init( h );
    else
        x264_stack_align( x264_cavlc_init, h );

    //决定了像素比较的时候用SAD还是SATD
    mbcmp_init( h );
    chroma_dsp_init( h );
    //CPU属性
    p = buf + sprintf( buf, "using cpu capabilities:" );
    for( int i = 0; x264_cpu_names[i].flags; i++ )
    {
        if( !strcmp(x264_cpu_names[i].name, "SSE")
            && h->param.cpu & (X264_CPU_SSE2) )
            continue;
        if( !strcmp(x264_cpu_names[i].name, "SSE2")
            && h->param.cpu & (X264_CPU_SSE2_IS_FAST|X264_CPU_SSE2_IS_SLOW) )
            continue;
        if( !strcmp(x264_cpu_names[i].name, "SSE3")
            && (h->param.cpu & X264_CPU_SSSE3 || !(h->param.cpu & X264_CPU_CACHELINE_64)) )
            continue;
        if( !strcmp(x264_cpu_names[i].name, "SSE4.1")
            && (h->param.cpu & X264_CPU_SSE42) )
            continue;
        if( !strcmp(x264_cpu_names[i].name, "BMI1")
            && (h->param.cpu & X264_CPU_BMI2) )
            continue;
        if( (h->param.cpu & x264_cpu_names[i].flags) == x264_cpu_names[i].flags
            && (!i || x264_cpu_names[i].flags != x264_cpu_names[i-1].flags) )
            p += sprintf( p, " %s", x264_cpu_names[i].name );
    }
    if( !h->param.cpu )
```

```
    if( !h->param.cpu )
        p += sprintf( p, " none!" );
    x264_log( h, X264_LOG_INFO, "%s\n", buf );

    float *logs = x264_analyse_prepare_costs( h );
    if( !logs )
        goto fail;
    for( qp = X264_MIN( h->param.rc.i_qp_min, QP_MAX_SPEC ); qp <= h->param.rc.i_qp_max; qp++ )
        if( x264_analyse_init_costs( h, logs, qp ) )
            goto fail;
    if( x264_analyse_init_costs( h, logs, X264_LOOKAHEAD_QP ) )
        goto fail;
    x264_free( logs );

    static const uint16_t cost_mv_correct[7] = { 24, 47, 95, 189, 379, 757, 1515 };
    /* Checks for known miscompilation issues. */
    if( h->cost_mv[X264_LOOKAHEAD_QP][2013] != cost_mv_correct[BIT_DEPTH-8] )
    {
        x264_log( h, X264_LOG_ERROR, "MV cost test failed: x264 has been miscompiled!\n" );
        goto fail;
    }

    /* Must be volatile or else GCC will optimize it out. */
    volatile int temp = 392;
    if( x264_clz( temp ) != 23 )
    {
        x264_log( h, X264_LOG_ERROR, "CLZ test failed: x264 has been miscompiled!\n" );
#if ARCH_X86 || ARCH_X86_64
        x264_log( h, X264_LOG_ERROR, "Are you attempting to run an SSE4a/LZCNT-targeted build on a CPU that\n" );
        x264_log( h, X264_LOG_ERROR, "doesn't support it?\n" );
#endif
        goto fail;
    }

    h->out.i_nal = 0;
    h->out.i_bitstream = X264_MAX( 1000000, h->param.i_width * h->param.i_height * 4
        * ( h->param.rc.i_rc_method == X264_RC_ABR ? pow( 0.95, h->param.rc.i_qp_min )
          : pow( 0.95, h->param.rc.i_qp_constant ) * X264_MAX( 1, h->param.rc.f_ip_factor )));

    h->nal_buffer_size = h->out.i_bitstream * 3/2 + 4 + 64; /* +4 for startcode, +64 for nal_escape assembly padding */
    CHECKED_MALLOC( h->nal_buffer, h->nal_buffer_size );

    CHECKED_MALLOC( h->reconfig_h, sizeof(x264_t) );

    if( h->param.i_threads > 1 &&
        x264_threadpool_init( &h->threadpool, h->param.i_threads, (void*)x264_encoder_thread_init, h ) )
        goto fail;
    if( h->param.i_lookahead_threads > 1 &&
        x264_threadpool_init( &h->lookaheadpool, h->param.i_lookahead_threads, NULL, NULL ) )
        goto fail;

#if HAVE_OPENCL
    if( h->param.b_opencl )
    {
        h->opencl.ocl = x264_opencl_load_library();
        if( !h->opencl.ocl )
        {
            x264_log( h, X264_LOG_WARNING, "failed to load OpenCL\n" );
            h->param.b_opencl = 0;
        }
    }
#endif

    h->thread[0] = h;
    for( int i = 1; i < h->param.i_threads + !!h->param.i_sync_lookahead; i++ )
        CHECKED_MALLOC( h->thread[i], sizeof(x264_t) );
    if( h->param.i_lookahead_threads > 1 )
        for( int i = 0; i < h->param.i_lookahead_threads; i++ )
        {
            CHECKED_MALLOC( h->lookahead_thread[i], sizeof(x264_t) );
            *h->lookahead_thread[i] = *h;
        }
    *h->reconfig_h = *h;

    for( int i = 0; i < h->param.i_threads; i++ )
    {
        int init_nal_count = h->param.i_slice_count + 3;
        int allocate_threadlocal_data = !h->param.b_sliced_threads || !i;
        if( i > 0 )
            *h->thread[i] = *h;

        if( x264_pthread_mutex_init( &h->thread[i]->mutex, NULL ) )
            goto fail;
        if( x264_pthread_cond_init( &h->thread[i]->cv, NULL ) )
            goto fail;

        if( allocate_threadlocal_data )
        {
            h->thread[i]->fdec = x264_frame_pop_unused( h, 1 );
            if( !h->thread[i]->fdec )
                goto fail;
        }
```

```
        }
        else
            h->thread[i]->fdec = h->thread[0]->fdec;

        CHECKED_MALLOC( h->thread[i]->out.p_bitstream, h->out.i_bitstream );
        /* Start each thread with room for init_nal_count NAL units; it'll realloc later if needed. */
        CHECKED_MALLOC( h->thread[i]->out.nal, init_nal_count*sizeof(x264_nal_t) );
        h->thread[i]->out.i_nals_allocated = init_nal_count;

        if( allocate_threadlocal_data && x264_macroblock_cache_allocate( h->thread[i] ) < 0 )
            goto fail;
    }

#if HAVE_OPENCL
    if( h->param.b_opencl && x264_opencl_lookahead_init( h ) < 0 )
        h->param.b_opencl = 0;
#endif
    //初始化lookahead
    if( x264_lookahead_init( h, i_slicetype_length ) )
        goto fail;

    for( int i = 0; i < h->param.i_threads; i++ )
        if( x264_macroblock_thread_allocate( h->thread[i], 0 ) < 0 )
            goto fail;
    //创建码率控制
    if( x264_ratecontrol_new( h ) < 0 )
        goto fail;

    if( h->param.i_nal_hrd )
    {
        x264_log( h, X264_LOG_DEBUG, "HRD bitrate: %i bits/sec\n", h->sps->vui.hrd.i_bit_rate_unscaled );
        x264_log( h, X264_LOG_DEBUG, "CPB size: %i bits\n", h->sps->vui.hrd.i_cpb_size_unscaled );
    }

    if( h->param.psz_dump_yuv )
    {
        /* create or truncate the reconstructed video file */
        FILE *f = x264_fopen( h->param.psz_dump_yuv, "w" );
        if( !f )
        {
            x264_log( h, X264_LOG_ERROR, "dump_yuv: can't write to %s\n", h->param.psz_dump_yuv );
            goto fail;
        }
        else if( !x264_is_regular_file( f ) )
        {
            x264_log( h, X264_LOG_ERROR, "dump_yuv: incompatible with non-regular file %s\n", h->param.psz_dump_yuv );
            goto fail;
        }
        fclose( f );
    }
    //这写法......
    const char *profile = h->sps->i_profile_idc == PROFILE_BASELINE ? "Constrained Baseline" :
                          h->sps->i_profile_idc == PROFILE_MAIN ? "Main" :
                          h->sps->i_profile_idc == PROFILE_HIGH ? "High" :
                          h->sps->i_profile_idc == PROFILE_HIGH10 ? (h->sps->b_constraint_set3 == 1 ? "High 10 Intra" : "High 10") :
                          h->sps->i_profile_idc == PROFILE_HIGH422 ? (h->sps->b_constraint_set3 == 1 ? "High 4:2:2 Intra" : "High 4:2:2") :
                          h->sps->b_constraint_set3 == 1 ? "High 4:4:4 Intra" : "High 4:4:4 Predictive";
    char level[4];
    snprintf( level, sizeof(level), "%d.%d", h->sps->i_level_idc/10, h->sps->i_level_idc%10 );
    if( h->sps->i_level_idc == 9 || ( h->sps->i_level_idc == 11 && h->sps->b_constraint_set3 &&
        (h->sps->i_profile_idc == PROFILE_BASELINE || h->sps->i_profile_idc == PROFILE_MAIN) ) )
        strcpy( level, "1b" );
    //输出型和级
    if( h->sps->i_profile_idc < PROFILE_HIGH10 )
    {
        x264_log( h, X264_LOG_INFO, "profile %s, level %s\n",
            profile, level );
    }
    else
    {
        static const char * const subsampling[4] = { "4:0:0", "4:2:0", "4:2:2", "4:4:4" };
        x264_log( h, X264_LOG_INFO, "profile %s, level %s, %s %d-bit\n",
            profile, level, subsampling[CHROMA_FORMAT], BIT_DEPTH );
    }

    return h;
fail:
    //释放
    x264_free( h );
    return NULL;
}
```

由于源代码中已经做了比较详细的注释，在这里就不重复叙述了。下面根据函数调用的顺序，看一下x264_encoder_open()调用的下面几个函数：

x264_sps_init()：根据输入参数生成H.264码流的SPS信息。

x264_pps_init()：根据输入参数生成H.264码流的PPS信息。

x264_predict_16x16_init()：初始化Intra16x16帧内预测汇编函数。

x264_predict_4x4_init()：初始化Intra4x4帧内预测汇编函数。

x264_pixel_init()：初始化像素值计算相关的汇编函数（包括SAD、SATD、SSD等）。

x264_dct_init()：初始化DCT变换和DCT反变换相关的汇编函数。

x264_mc_init()：初始化运动补偿相关的汇编函数。

x264_quant_init()：初始化量化和反量化相关的汇编函数。

x264_deblock_init()：初始化去块效应滤波器相关的汇编函数。

mbcmp_init()：决定像素比较的时候使用SAD还是SATD。

x264_sps_init()

x264_sps_init()根据输入参数生成H.264码流的SPS （Sequence Parameter Set，序列参数集）信息。该函数的定义位于encoder\set.c，如下所示。

[cpp]

```cpp
//初始化SPS
void x264_sps_init( x264_sps_t *sps, int i_id, x264_param_t *param )
{
    int csp = param->i_csp & X264_CSP_MASK;

    sps->i_id = i_id;
    //以宏块为单位的宽度
    sps->i_mb_width = ( param->i_width + 15 ) / 16;
    //以宏块为单位的高度
    sps->i_mb_height= ( param->i_height + 15 ) / 16;
    //色度取样格式
    sps->i_chroma_format_idc = csp >= X264_CSP_I444 ? CHROMA_444 :
                               csp >= X264_CSP_I422 ? CHROMA_422 : CHROMA_420;

    sps->b_qpprime_y_zero_transform_bypass = param->rc.i_rc_method == X264_RC_CQP && param->rc.i_qp_constant == 0;
    //型profile
    if( sps->b_qpprime_y_zero_transform_bypass || sps->i_chroma_format_idc == CHROMA_444 )
        sps->i_profile_idc  = PROFILE_HIGH444_PREDICTIVE;//YUV444的时候
    else if( sps->i_chroma_format_idc == CHROMA_422 )
        sps->i_profile_idc  = PROFILE_HIGH422;
    else if( BIT_DEPTH > 8 )
        sps->i_profile_idc  = PROFILE_HIGH10;
    else if( param->analyse.b_transform_8x8 || param->i_cqm_preset != X264_CQM_FLAT )
        sps->i_profile_idc  = PROFILE_HIGH;//高型 High Profile 目前最常见
    else if( param->b_cabac || param->i_bframe > 0 || param->b_interlaced || param->b_fake_interlaced || param->analyse.i_weighted_pred > 0 )
        sps->i_profile_idc  = PROFILE_MAIN;//主型
    else
        sps->i_profile_idc  = PROFILE_BASELINE;//基本型

    sps->b_constraint_set0  = sps->i_profile_idc == PROFILE_BASELINE;
    /* x264 doesn't support the features that are in Baseline and not in Main,
     * namely arbitrary_slice_order and slice_groups. */
    sps->b_constraint_set1  = sps->i_profile_idc <= PROFILE_MAIN;
    /* Never set constraint_set2, it is not necessary and not used in real world. */
    sps->b_constraint_set2  = 0;
    sps->b_constraint_set3  = 0;
    //级level
    sps->i_level_idc = param->i_level_idc;
    if( param->i_level_idc == 9 && ( sps->i_profile_idc == PROFILE_BASELINE || sps->i_profile_idc == PROFILE_MAIN ) )
    {
        sps->b_constraint_set3 = 1; /* level 1b with Baseline or Main profile is signalled via constraint_set3 */
        sps->i_level_idc      = 11;
    }
    /* Intra profiles */
    if( param->i_keyint_max == 1 && sps->i_profile_idc > PROFILE_HIGH )
        sps->b_constraint_set3 = 1;

    sps->vui.i_num_reorder_frames = param->i_bframe_pyramid ? 2 : param->i_bframe ? 1 : 0;
    /* extra slot with pyramid so that we don't have to override the
     * order of forgetting old pictures */
    //参考帧数量
    sps->vui.i_max_dec_frame_buffering =
    sps->i_num_ref_frames = X264_MIN(X264_REF_MAX, X264_MAX4(param->i_frame_reference, 1 + sps->vui.i_num_reorder_frames,
                            param->i_bframe_pyramid ? 4 : 1, param->i_dpb_size));
    sps->i_num_ref_frames -= param->i_bframe_pyramid == X264_B_PYRAMID_STRICT;
    if( param->i_keyint_max == 1 )
    {
        sps->i_num_ref_frames = 0;
        sps->vui.i_max_dec_frame_buffering = 0;
    }

    /* number of refs + current frame */
    int max_frame_num = sps->vui.i_max_dec_frame_buffering * (!!param->i_bframe_pyramid+1) + 1;
    /* Intra refresh cannot write a recovery time greater than max frame num-1 */
    if( param->b_intra_refresh )
    {
        int time_to_recovery = X264_MIN( sps->i_mb_width - 1, param->i_keyint_max ) + param->i_bframe - 1;
        max_frame_num = X264_MAX( max_frame_num, time_to_recovery+1 );
    }

    sps->i_log2_max_frame_num = 4;
    while( (1 << sps->i_log2_max_frame_num) <= max_frame_num )
```

```c
        sps->i_log2_max_frame_num++;
    //POC类型
    sps->i_poc_type = param->i_bframe || param->b_interlaced ? 0 : 2;
    if( sps->i_poc_type == 0 )
    {
        int max_delta_poc = (param->i_bframe + 2) * (!!param->i_bframe_pyramid + 1) * 2;
        sps->i_log2_max_poc_lsb = 4;
        while( (1 << sps->i_log2_max_poc_lsb) <= max_delta_poc * 2 )
            sps->i_log2_max_poc_lsb++;
    }

    sps->b_vui = 1;

    sps->b_gaps_in_frame_num_value_allowed = 0;
    sps->b_frame_mbs_only = !(param->b_interlaced || param->b_fake_interlaced);
    if( !sps->b_frame_mbs_only )
        sps->i_mb_height = ( sps->i_mb_height + 1 ) & ~1;
    sps->b_mb_adaptive_frame_field = param->b_interlaced;
    sps->b_direct8x8_inference = 1;

    sps->crop.i_left   = param->crop_rect.i_left;
    sps->crop.i_top    = param->crop_rect.i_top;
    sps->crop.i_right  = param->crop_rect.i_right + sps->i_mb_width*16 - param->i_width;
    sps->crop.i_bottom = (param->crop_rect.i_bottom + sps->i_mb_height*16 - param->i_height) >> !sps->b_frame_mbs_only;
    sps->b_crop = sps->crop.i_left  || sps->crop.i_top ||
                  sps->crop.i_right || sps->crop.i_bottom;

    sps->vui.b_aspect_ratio_info_present = 0;
    if( param->vui.i_sar_width > 0 && param->vui.i_sar_height > 0 )
    {
        sps->vui.b_aspect_ratio_info_present = 1;
        sps->vui.i_sar_width = param->vui.i_sar_width;
        sps->vui.i_sar_height= param->vui.i_sar_height;
    }

    sps->vui.b_overscan_info_present = param->vui.i_overscan > 0 && param->vui.i_overscan <= 2;
    if( sps->vui.b_overscan_info_present )
        sps->vui.b_overscan_info = ( param->vui.i_overscan == 2 ? 1 : 0 );

    sps->vui.b_signal_type_present = 0;
    sps->vui.i_vidformat = ( param->vui.i_vidformat >= 0 && param->vui.i_vidformat <= 5 ? param->vui.i_vidformat : 5 );
    sps->vui.b_fullrange = ( param->vui.b_fullrange >= 0 && param->vui.b_fullrange <= 1 ? param->vui.b_fullrange :
                           ( csp >= X264_CSP_BGR ? 1 : 0 ) );
    sps->vui.b_color_description_present = 0;

    sps->vui.i_colorprim = ( param->vui.i_colorprim >= 0 && param->vui.i_colorprim <=  9 ? param->vui.i_colorprim : 2 );
    sps->vui.i_transfer  = ( param->vui.i_transfer  >= 0 && param->vui.i_transfer  <= 15 ? param->vui.i_transfer  : 2 );
    sps->vui.i_colmatrix = ( param->vui.i_colmatrix >= 0 && param->vui.i_colmatrix <= 10 ? param->vui.i_colmatrix :
                           ( csp >= X264_CSP_BGR ? 0 : 2 ) );
    if( sps->vui.i_colorprim != 2 ||
        sps->vui.i_transfer  != 2 ||
        sps->vui.i_colmatrix != 2 )
    {
        sps->vui.b_color_description_present = 1;
    }

    if( sps->vui.i_vidformat != 5 ||
        sps->vui.b_fullrange ||
        sps->vui.b_color_description_present )
    {
        sps->vui.b_signal_type_present = 1;
    }

    /* FIXME: not sufficient for interlaced video */
    sps->vui.b_chroma_loc_info_present = param->vui.i_chroma_loc > 0 && param->vui.i_chroma_loc <= 5 &&
                                         sps->i_chroma_format_idc == CHROMA_420;
    if( sps->vui.b_chroma_loc_info_present )
    {
        sps->vui.i_chroma_loc_top = param->vui.i_chroma_loc;
        sps->vui.i_chroma_loc_bottom = param->vui.i_chroma_loc;
    }

    sps->vui.b_timing_info_present = param->i_timebase_num > 0 && param->i_timebase_den > 0;

    if( sps->vui.b_timing_info_present )
    {
        sps->vui.i_num_units_in_tick = param->i_timebase_num;
        sps->vui.i_time_scale = param->i_timebase_den * 2;
        sps->vui.b_fixed_frame_rate = !param->b_vfr_input;
    }

    sps->vui.b_vcl_hrd_parameters_present = 0; // we don't support VCL HRD
    sps->vui.b_nal_hrd_parameters_present = !!param->i_nal_hrd;
    sps->vui.b_pic_struct_present = param->b_pic_struct;

    // NOTE: HRD related parts of the SPS are initialised in x264_ratecontrol_init_reconfigurable

    sps->vui.b_bitstream_restriction = param->i_keyint_max > 1;
    if( sps->vui.b_bitstream_restriction )
    {
        sps->vui.b_motion_vectors_over_pic_boundaries = 1;
```

```
        sps->vui.i_max_bytes_per_pic_denom = 0;
        sps->vui.i_max_bits_per_mb_denom = 0;
        sps->vui.i_log2_max_mv_length_horizontal =
        sps->vui.i_log2_max_mv_length_vertical = (int)log2f( X264_MAX( 1, param->analyse.i_mv_range*4-1 ) ) + 1;
    }
}
```

从源代码可以看出，x264_sps_init()根据输入参数集x264_param_t中的信息，初始化了SPS结构体中的成员变量。有关这些成员变量的具体信息，可以参考《H.264标准》。

x264_pps_init()

x264_pps_init()根据输入参数生成H.264码流的PPS（Picture Parameter Set，图像参数集）信息。该函数的定义位于encoder\set.c，如下所示。

[cpp]

```cpp
//初始化PPS
void x264_pps_init( x264_pps_t *pps, int i_id, x264_param_t *param, x264_sps_t *sps )
{
    pps->i_id = i_id;
    //所属的SPS
    pps->i_sps_id = sps->i_id;
    //是否使用CABAC？
    pps->b_cabac = param->b_cabac;

    pps->b_pic_order = !param->i_avcintra_class && param->b_interlaced;
    pps->i_num_slice_groups = 1;
    //目前参考帧队列的长度
    //注意是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。
    pps->i_num_ref_idx_l0_default_active = param->i_frame_reference;
    pps->i_num_ref_idx_l1_default_active = 1;
    //加权预测
    pps->b_weighted_pred = param->analyse.i_weighted_pred > 0;
    pps->b_weighted_bipred = param->analyse.b_weighted_bipred ? 2 : 0;
    //量化参数QP的初始值
    pps->i_pic_init_qp = param->rc.i_rc_method == X264_RC_ABR || param->b_stitchable ? 26 + QP_BD_OFFSET : SPEC_QP( param->rc.i_qp_constant );
    pps->i_pic_init_qs = 26 + QP_BD_OFFSET;

    pps->i_chroma_qp_index_offset = param->analyse.i_chroma_qp_offset;
    pps->b_deblocking_filter_control = 1;
    pps->b_constrained_intra_pred = param->b_constrained_intra;
    pps->b_redundant_pic_cnt = 0;

    pps->b_transform_8x8_mode = param->analyse.b_transform_8x8 ? 1 : 0;

    pps->i_cqm_preset = param->i_cqm_preset;

    switch( pps->i_cqm_preset )
    {
    case X264_CQM_FLAT:
        for( int i = 0; i < 8; i++ )
            pps->scaling_list[i] = x264_cqm_flat16;
        break;
    case X264_CQM_JVT:
        for( int i = 0; i < 8; i++ )
            pps->scaling_list[i] = x264_cqm_jvt[i];
        break;
    case X264_CQM_CUSTOM:
        /* match the transposed DCT & zigzag */
        transpose( param->cqm_4iy, 4 );
        transpose( param->cqm_4py, 4 );
        transpose( param->cqm_4ic, 4 );
        transpose( param->cqm_4pc, 4 );
        transpose( param->cqm_8iy, 8 );
        transpose( param->cqm_8py, 8 );
        transpose( param->cqm_8ic, 8 );
        transpose( param->cqm_8pc, 8 );
        pps->scaling_list[CQM_4IY] = param->cqm_4iy;
        pps->scaling_list[CQM_4PY] = param->cqm_4py;
        pps->scaling_list[CQM_4IC] = param->cqm_4ic;
        pps->scaling_list[CQM_4PC] = param->cqm_4pc;
        pps->scaling_list[CQM_8IY+4] = param->cqm_8iy;
        pps->scaling_list[CQM_8PY+4] = param->cqm_8py;
        pps->scaling_list[CQM_8IC+4] = param->cqm_8ic;
        pps->scaling_list[CQM_8PC+4] = param->cqm_8pc;
        for( int i = 0; i < 8; i++ )
            for( int j = 0; j < (i < 4 ? 16 : 64); j++ )
                if( pps->scaling_list[i][j] == 0 )
                    pps->scaling_list[i] = x264_cqm_jvt[i];
        break;
    }
}
```

从源代码可以看出，x264_pps_init()根据输入参数集x264_param_t中的信息，初始化了PPS结构体中的成员变量。有关这些成员变量的具体信息，可以参考《H.264标准》。

x264_predict_16x16_init()

x264_predict_16x16_init()用于初始化Intra16x16帧内预测汇编函数。该函数的定义位于x264\common\predict.c，如下所示。

[cpp]

```cpp
//Intra16x16帧内预测汇编函数初始化
void x264_predict_16x16_init( int cpu, x264_predict_t pf[7] )
{
    //C语言版本
    //================================================
    //垂直 Vertical
    pf[I_PRED_16x16_V ]     = x264_predict_16x16_v_c;
    //水平 Horizontal
    pf[I_PRED_16x16_H ]     = x264_predict_16x16_h_c;
    //DC
    pf[I_PRED_16x16_DC]     = x264_predict_16x16_dc_c;
    //Plane
    pf[I_PRED_16x16_P ]     = x264_predict_16x16_p_c;
    //这几种是啥？
    pf[I_PRED_16x16_DC_LEFT]= x264_predict_16x16_dc_left_c;
    pf[I_PRED_16x16_DC_TOP ]= x264_predict_16x16_dc_top_c;
    pf[I_PRED_16x16_DC_128 ]= x264_predict_16x16_dc_128_c;
    //================================================
    //MMX版本
#if HAVE_MMX
    x264_predict_16x16_init_mmx( cpu, pf );
#endif
    //ALTIVEC版本
#if HAVE_ALTIVEC
    if( cpu&X264_CPU_ALTIVEC )
        x264_predict_16x16_init_altivec( pf );
#endif
    //ARMV6版本
#if HAVE_ARMV6
    x264_predict_16x16_init_arm( cpu, pf );
#endif
    //AARCH64版本
#if ARCH_AARCH64
    x264_predict_16x16_init_aarch64( cpu, pf );
#endif
}
```

从源代码可看出，x264_predict_16x16_init()首先对帧内预测函数指针数组x264_predict_t[]中的元素赋值了C语言版本的函数x264_predict_16x16_v_c()，x264_predict_16x16_h_c()，x264_predict_16x16_dc_c()，x264_predict_16x16_p_c()；然后会判断系统平台的特性，如果平台支持的话，会调用x264_predict_16x16_init_mmx()，x264_predict_16x16_init_arm()等给x264_predict_t[]中的元素赋值经过汇编优化的函数。下文将会简单看几个其中的函数。

相关知识简述

简单记录一下帧内预测的方法。帧内预测根据宏块左边和上边的边界像素值推算宏块内部的像素值，帧内预测的效果如下图所示。其中左边的图为图像原始画面，右边的图为经过帧内预测后没有叠加残差的画面。

H.264中有两种帧内预测模式：16x16亮度帧内预测模式和4x4亮度帧内预测模式。其中16x16帧内预测模式一共有4种，如下图所示。

这4种模式列表如下。

模式	描述
Vertical	由上边像素推出相应像素值
Horizontal	由左边像素推出相应像素值
DC	由上边和左边像素平均值推出相应像素值
Plane	由上边和左边像素推出相应像素值

4x4帧内预测模式一共有9种，如下图所示。

有关Intra4x4的帧内预测模式的代码将在后文中进行记录。下面举例看一下Intra16x16的Vertical预测模式的实现函数x264_predict_16x16_v_c()。

x264_predict_16x16_v_c()

x264_predict_16x16_v_c()实现了Intra16x16的Vertical预测模式。该函数的定义位于common\predict.c，如下所示。

```cpp
//16x16帧内预测
//垂直预测（Vertical）
void x264_predict_16x16_v_c( pixel *src )
{
    /*
     * Vertical预测方式
     *   |X1 X2 X3 X4
     * --+-----------
     *   |X1 X2 X3 X4
     *   |X1 X2 X3 X4
     *   |X1 X2 X3 X4
     *   |X1 X2 X3 X4
     *
     */
    /*
     * 【展开宏定义】
     * uint32_t v0 = ((x264_union32_t*)(&src[ 0-FDEC_STRIDE]))->i;
     * uint32_t v1 = ((x264_union32_t*)(&src[ 4-FDEC_STRIDE]))->i;
     * uint32_t v2 = ((x264_union32_t*)(&src[ 8-FDEC_STRIDE]))->i;
     * uint32_t v3 = ((x264_union32_t*)(&src[12-FDEC_STRIDE]))->i;
     * 在这里，上述代码实际上相当于：
     * uint32_t v0 = *((uint32_t*)(&src[ 0-FDEC_STRIDE]));
     * uint32_t v1 = *((uint32_t*)(&src[ 4-FDEC_STRIDE]));
     * uint32_t v2 = *((uint32_t*)(&src[ 8-FDEC_STRIDE]));
     * uint32_t v3 = *((uint32_t*)(&src[12-FDEC_STRIDE]));
     * 即分成4次，每次取出4个像素（一共16个像素），分别赋值给v0，v1，v2，v3
     * 取出的值源自于16x16块上面的一行像素
     *    0|          4          8          12         16
     *    ||    v0    |    v1    |    v2    |    v3    |
     * ---++==========+==========+==========+==========+
     *    ||
     *    ||
     *    ||
     *    ||
     *    ||
     *    ||
     *
     */
    //pixel4实际上是uint32_t（占用32bit），存储4个像素的值（每个像素占用8bit）

    pixel4 v0 = MPIXEL_X4( &src[ 0-FDEC_STRIDE] );
    pixel4 v1 = MPIXEL_X4( &src[ 4-FDEC_STRIDE] );
    pixel4 v2 = MPIXEL_X4( &src[ 8-FDEC_STRIDE] );
    pixel4 v3 = MPIXEL_X4( &src[12-FDEC_STRIDE] );

    //循环赋值16行
    for( int i = 0; i < 16; i++ )
    {
        //【展开宏定义】
        //(((x264_union32_t*)(src+ 0))->i) = v0;
        //(((x264_union32_t*)(src+ 4))->i) = v1;
        //(((x264_union32_t*)(src+ 8))->i) = v2;
        //(((x264_union32_t*)(src+12))->i) = v3;
        //即分成4次，每次赋值4个像素
        //
        MPIXEL_X4( src+ 0 ) = v0;
        MPIXEL_X4( src+ 4 ) = v1;
        MPIXEL_X4( src+ 8 ) = v2;
        MPIXEL_X4( src+12 ) = v3;
        //下一行
        //FDEC_STRIDE=32,是重建宏块缓存fdec_buf一行的数据量
        src += FDEC_STRIDE;
    }
}
```

从源代码可以看出，x264_predict_16x16_v_c()首先取出了16x16图像块上面一行16个像素的值存储在v0，v1，v2，v3四个变量中（每个变量存储4个像素），然后循环16次将v0，v1，v2，v3赋值给16x16图像块的16行。

看完C语言版本Intra16x16的Vertical预测模式的实现函数之后，我们可以继续看一下该预测模式汇编语言版本的实现函数。从前面的初始化函数中已经可以看出，当系统支持X86汇编的时候，会调用x264_predict_16x16_init_mmx()初始化x86汇编优化过的函数；当系统支持ARM的时候，会调用x264_predict_16x16_init_arm()初始化ARM汇编优化过的函数。

x264_predict_16x16_init_mmx()

x264_predict_16x16_init_mmx()用于初始化经过x86汇编优化过的Intra16x16的帧内预测函数。该函数的定义位于common\x86\predict-c.c（在“x86”子文件夹下），如下所示。

[cpp]

```cpp
//Intra16x16帧内预测汇编函数-MMX版本
void x264_predict_16x16_init_mmx( int cpu, x264_predict_t pf[7] )
{
    if( !(cpu&X264_CPU_MMX2) )
        return;
    pf[I_PRED_16x16_DC]      = x264_predict_16x16_dc_mmx2;
    pf[I_PRED_16x16_DC_TOP]  = x264_predict_16x16_dc_top_mmx2;
    pf[I_PRED_16x16_DC_LEFT] = x264_predict_16x16_dc_left_mmx2;
    pf[I_PRED_16x16_V]       = x264_predict_16x16_v_mmx2;
    pf[I_PRED_16x16_H]       = x264_predict_16x16_h_mmx2;
#if HIGH_BIT_DEPTH
    if( !(cpu&X264_CPU_SSE) )
        return;
    pf[I_PRED_16x16_V]       = x264_predict_16x16_v_sse;
    if( !(cpu&X264_CPU_SSE2) )
        return;
    pf[I_PRED_16x16_DC]      = x264_predict_16x16_dc_sse2;
    pf[I_PRED_16x16_DC_TOP]  = x264_predict_16x16_dc_top_sse2;
    pf[I_PRED_16x16_DC_LEFT] = x264_predict_16x16_dc_left_sse2;
    pf[I_PRED_16x16_H]       = x264_predict_16x16_h_sse2;
    pf[I_PRED_16x16_P]       = x264_predict_16x16_p_sse2;
    if( !(cpu&X264_CPU_AVX) )
        return;
    pf[I_PRED_16x16_V]       = x264_predict_16x16_v_avx;
    if( !(cpu&X264_CPU_AVX2) )
        return;
    pf[I_PRED_16x16_H]       = x264_predict_16x16_h_avx2;
#else
#if !ARCH_X86_64
    pf[I_PRED_16x16_P]       = x264_predict_16x16_p_mmx2;
#endif
    if( !(cpu&X264_CPU_SSE) )
        return;
    pf[I_PRED_16x16_V]       = x264_predict_16x16_v_sse;
    if( !(cpu&X264_CPU_SSE2) )
        return;
    pf[I_PRED_16x16_DC]      = x264_predict_16x16_dc_sse2;
    if( cpu&X264_CPU_SSE2_IS_SLOW )
        return;
    pf[I_PRED_16x16_DC_TOP]  = x264_predict_16x16_dc_top_sse2;
    pf[I_PRED_16x16_DC_LEFT] = x264_predict_16x16_dc_left_sse2;
    pf[I_PRED_16x16_P]       = x264_predict_16x16_p_sse2;
    if( !(cpu&X264_CPU_SSSE3) )
        return;
    if( !(cpu&X264_CPU_SLOW_PSHUFB) )
        pf[I_PRED_16x16_H]       = x264_predict_16x16_h_ssse3;
#if HAVE_X86_INLINE_ASM
    pf[I_PRED_16x16_P]       = x264_predict_16x16_p_ssse3;
#endif
    if( !(cpu&X264_CPU_AVX) )
        return;
    pf[I_PRED_16x16_P]       = x264_predict_16x16_p_avx;
#endif // HIGH_BIT_DEPTH

    if( cpu&X264_CPU_AVX2 )
    {
        pf[I_PRED_16x16_P]       = x264_predict_16x16_p_avx2;
        pf[I_PRED_16x16_DC]      = x264_predict_16x16_dc_avx2;
        pf[I_PRED_16x16_DC_TOP]  = x264_predict_16x16_dc_top_avx2;
        pf[I_PRED_16x16_DC_LEFT] = x264_predict_16x16_dc_left_avx2;
    }
}
```

可以看出，针对Intra16x16的Vertical帧内预测模式，x264_predict_16x16_init_mmx()会根据系统的特型初始化2个函数：如果系统仅支持MMX指令集，就会初始化x264_predict_16x16_v_mmx2()；如果系统还支持SSE指令集，就会初始化x264_predict_16x16_v_sse()。下面看一下这2个函数的代码。

x264_predict_16x16_v_mmx2()

x264_predict_16x16_v_sse()

在x264中，x264_predict_16x16_v_mmx2()和x264_predict_16x16_v_sse()这两个函数的定义是写到一起的。它们的定义位于common\x86\predict-a.asm，如下所示。

[plain]

```
;-----------------------------------------------------------------------------
; void predict_16x16_v( pixel *src )
; Intra16x16帧内预测Vertical模式
;-----------------------------------------------------------------------------
;SIZEOF_PIXEL取值为1
;FDEC_STRIDEB为重建宏块缓存fdec_buf一行像素的大小，取值为32
;
;平台相关的信息位于x86inc.asm
;INIT_MMX中
;  mmsize为8
;  mova为movq
;INIT_XMM中：
;  mmsize为16
;  mova为movdqa
;
;STORE16的定义在前面，用于循环16行存储数据

%macro PREDICT_16x16_V 0
cglobal predict_16x16_v, 1,2
%assign %%i 0
%rep 16*SIZEOF_PIXEL/mmsize                         ;rep循环执行，拷贝16x16块上方的1行像素数据至m0,m1...
                                                    ;mmssize为指令1次处理比特数
    mova m %+ %%i, [r0-FDEC_STRIDEB+%%i*mmsize]     ;移入m0,m1...
%assign %%i %%i+1
%endrep
%if 16*SIZEOF_PIXEL/mmsize == 4                     ;1行需要处理4次
    STORE16 m0, m1, m2, m3                          ;循环存储16行，每次存储4个寄存器
%elif 16*SIZEOF_PIXEL/mmsize == 2                   ;1行需要处理2次
    STORE16 m0, m1                                  ;循环存储16行，每次存储2个寄存器
%else                                               ;1行需要处理1次
    STORE16 m0                                      ;循环存储16行，每次存储1个寄存器
%endif
    RET
%endmacro

INIT_MMX mmx2
PREDICT_16x16_V
INIT_XMM sse
PREDICT_16x16_V
```

从汇编代码可以看出，x264_predict_16x16_v_mmx2()和x264_predict_16x16_v_sse()的逻辑是一模一样的。它们之间的不同主要在于一条指令处理的数据量：MMX指令的MOVA对应的是MOVQ，一次处理8Byte（8个像素）；SSE指令的MOVA对应的是MOVDQA，一次处理16Byte（16个像素，正好是16x16块中的一行像素）。
作为对比，我们可以看一下ARM平台下汇编优化过的Intra16x16的帧内预测函数。这些汇编函数的初始化函数是x264_predict_16x16_init_arm()。

x264_predict_16x16_init_arm()

x264_predict_16x16_init_arm()用于初始化ARM平台下汇编优化过的Intra16x16的帧内预测函数。该函数的定义位于common\arm\predict-c.c（“arm”文件夹下），如下所示。

[cpp]

```cpp
void x264_predict_16x16_init_arm( int cpu, x264_predict_t pf[7] )
{
    if (!(cpu&X264_CPU_NEON))
        return;

#if !HIGH_BIT_DEPTH
    pf[I_PRED_16x16_DC ]    = x264_predict_16x16_dc_neon;
    pf[I_PRED_16x16_DC_TOP] = x264_predict_16x16_dc_top_neon;
    pf[I_PRED_16x16_DC_LEFT]= x264_predict_16x16_dc_left_neon;
    pf[I_PRED_16x16_H ]     = x264_predict_16x16_h_neon;
    pf[I_PRED_16x16_V ]     = x264_predict_16x16_v_neon;
    pf[I_PRED_16x16_P ]     = x264_predict_16x16_p_neon;
#endif // !HIGH_BIT_DEPTH
}
```

从源代码可以看出，针对Vertical预测模式，x264_predict_16x16_init_arm()初始化了经过NEON指令集优化的函数x264_predict_16x16_v_neon()。

x264_predict_16x16_v_neon()

x264_predict_16x16_v_neon()的定义位于common\arm\predict-a.S，如下所示。

```
/*
 * Intra16x16帧内预测Vertical模式-NEON
 *
 */
 /* FDEC_STRIDE=32Bytes，为重建宏块一行像素的大小 */
 /* R0存储16x16像素块地址 */
function x264_predict_16x16_v_neon
    sub         r0, r0, #FDEC_STRIDE     /* r0=r0-FDEC_STRIDE */
    mov         ip, #FDEC_STRIDE         /* ip=32 */
                                         /* VLD向量加载：内存->NEON寄存器 */
                                         /* d0,d1为64bit双字寄存器，共16Byte，在这里存储16x16块上方一行像素 */
    vld1.64     {d0-d1}, [r0,:128], ip   /* 将R0指向的数据从内存加载到d0和d1寄存器（64bit） */
                                         /* r0=r0+ip */
.rept 16                                 /* 循环16次，一次处理1行 */
                                         /* VST向量存储：NEON寄存器->内存 */
    vst1.64     {d0-d1}, [r0,:128], ip   /* 将d0和d1寄存器中的数据传递给R0指向的内存 */
                                         /* r0=r0+ip */
.endr
    bx          lr                       /* 子程序返回 */
endfunc
```

可以看出，x264_predict_16x16_v_neon()使用vld1.64指令载入16x16块上方的一行像素，然后在一个16次的循环中，使用vst1.64指令将该行像素值赋值给16x16块的每一行。

至此有关Intra16x16的Vertical帧内预测方式的源代码就分析完了。后文为了简便，都只讨论C语言版本汇编函数。

x264_predict_4x4_init()

x264_predict_4x4_init()用于初始化Intra4x4帧内预测汇编函数。该函数的定义位于common\predict.c，如下所示。

```cpp
//Intra4x4帧内预测汇编函数初始化
void x264_predict_4x4_init( int cpu, x264_predict_t pf[12] )
{
    //9种Intra4x4预测方式
    pf[I_PRED_4x4_V]      = x264_predict_4x4_v_c;
    pf[I_PRED_4x4_H]      = x264_predict_4x4_h_c;
    pf[I_PRED_4x4_DC]     = x264_predict_4x4_dc_c;
    pf[I_PRED_4x4_DDL]    = x264_predict_4x4_ddl_c;
    pf[I_PRED_4x4_DDR]    = x264_predict_4x4_ddr_c;
    pf[I_PRED_4x4_VR]     = x264_predict_4x4_vr_c;
    pf[I_PRED_4x4_HD]     = x264_predict_4x4_hd_c;
    pf[I_PRED_4x4_VL]     = x264_predict_4x4_vl_c;
    pf[I_PRED_4x4_HU]     = x264_predict_4x4_hu_c;
    //这些是？
    pf[I_PRED_4x4_DC_LEFT]= x264_predict_4x4_dc_left_c;
    pf[I_PRED_4x4_DC_TOP] = x264_predict_4x4_dc_top_c;
    pf[I_PRED_4x4_DC_128] = x264_predict_4x4_dc_128_c;

#if HAVE_MMX
    x264_predict_4x4_init_mmx( cpu, pf );
#endif

#if HAVE_ARMV6
    x264_predict_4x4_init_arm( cpu, pf );
#endif

#if ARCH_AARCH64
    x264_predict_4x4_init_aarch64( cpu, pf );
#endif
}
```

从源代码可看出，x264_predict_4x4_init()首先对帧内预测函数指针数组x264_predict_t[]中的元素赋值了C语言版本的函数x264_predict_4x4_v_c()，x264_predict_4x4_h_c()，x264_predict_4x4_dc_c()，x264_predict_4x4_p_c()等一系列函数（Intra4x4有9种，后面那几种是怎么回事？）；然后会判断系统平台的特性，如果平台支持的话，会调用x264_predict_4x4_init_mmx()，x264_predict_4x4_init_arm()等给x264_predict_t[]中的元素赋值经过汇编优化的函数。作为例子，下文看一个Intra4x4的Vertical帧内预测模式的C语言函数。

相关知识简述

Intra4x4的帧内预测模式一共有9种。如下图所示。

可以看出，Intra4x4帧内预测模式中前4种和Intra16x16是一样的。后面多增加了几种预测箭头不是45度角的方式——前面的箭头位于“口”中，而后面的箭头位于“日”中。

x264_predict_4x4_v_c()

x264_predict_4x4_v_c()实现了Intra4x4的Vertical帧内预测方式。该函数的定义位于common\predict.c，如下所示。

```cpp
void x264_predict_4x4_v_c( pixel *src )
{
    /*
     * Vertical预测方式
     *   |X1 X2 X3 X4
     * --+-----------
     *   |X1 X2 X3 X4
     *   |X1 X2 X3 X4
     *   |X1 X2 X3 X4
     *   |X1 X2 X3 X4
     *
     */

    /*
     * 宏展开后的结果如下所示
     * 注：重建宏块缓存fdec_buf一行的数据量为32Byte
     *
     * (((x264_union32_t*)(&src[(0)+(0)*32]))->i) =
     * (((x264_union32_t*)(&src[(0)+(1)*32]))->i) =
     * (((x264_union32_t*)(&src[(0)+(2)*32]))->i) =
     * (((x264_union32_t*)(&src[(0)+(3)*32]))->i) = (((x264_union32_t*)(&src[(0)+(-1)*32]))->i);
     */
    PREDICT_4x4_DC(SRC_X4(0,-1));
}
```

x264_predict_4x4_v_c()函数的函数体极其简单，只有一个宏定义“PREDICT_4x4_DC(SRC_X4(0,-1));”。如果把该宏展开后，可以看出它取了4x4块上面一行4个像素的值，然后分别赋值给4x4块的4行像素。

x264_pixel_init()

x264_pixel_init()初始化像素值计算相关的汇编函数（包括SAD、SATD、SSD等）。该函数的定义位于common\pixel.c，如下所示。

```cpp
/****************************************************************************
 * x264_pixel_init:
 ****************************************************************************/
//SAD等和像素计算有关的函数
void x264_pixel_init( int cpu, x264_pixel_function_t *pixf )
{
    memset( pixf, 0, sizeof(*pixf) );

    //初始化2个函数-16x16,16x8
#define INIT2_NAME( name1, name2, cpu ) \
    pixf->name1[PIXEL_16x16] = x264_pixel_##name2##_16x16##cpu;\
    pixf->name1[PIXEL_16x8]  = x264_pixel_##name2##_16x8##cpu;
    //初始化4个函数-(16x16,16x8),8x16,8x8
#define INIT4_NAME( name1, name2, cpu ) \
    INIT2_NAME( name1, name2, cpu ) \
    pixf->name1[PIXEL_8x16]  = x264_pixel_##name2##_8x16##cpu;\
    pixf->name1[PIXEL_8x8]   = x264_pixel_##name2##_8x8##cpu;
    //初始化5个函数-(16x16,16x8,8x16,8x8),8x4
#define INIT5_NAME( name1, name2, cpu ) \
    INIT4_NAME( name1, name2, cpu ) \
    pixf->name1[PIXEL_8x4]   = x264_pixel_##name2##_8x4##cpu;
    //初始化6个函数-(16x16,16x8,8x16,8x8,8x4),4x8
#define INIT6_NAME( name1, name2, cpu ) \
    INIT5_NAME( name1, name2, cpu ) \
    pixf->name1[PIXEL_4x8]   = x264_pixel_##name2##_4x8##cpu;
    //初始化7个函数-(16x16,16x8,8x16,8x8,8x4,4x8),4x4
#define INIT7_NAME( name1, name2, cpu ) \
    INIT6_NAME( name1, name2, cpu ) \
    pixf->name1[PIXEL_4x4]   = x264_pixel_##name2##_4x4##cpu;
#define INIT8_NAME( name1, name2, cpu ) \
    INIT7_NAME( name1, name2, cpu ) \
    pixf->name1[PIXEL_4x16]  = x264_pixel_##name2##_4x16##cpu;

    //重新起个名字
#define INIT2( name, cpu ) INIT2_NAME( name, name, cpu )
#define INIT4( name, cpu ) INIT4_NAME( name, name, cpu )
#define INIT5( name, cpu ) INIT5_NAME( name, name, cpu )
#define INIT6( name, cpu ) INIT6_NAME( name, name, cpu )
#define INIT7( name, cpu ) INIT7_NAME( name, name, cpu )
#define INIT8( name, cpu ) INIT8_NAME( name, name, cpu )

#define INIT_ADS( cpu ) \
    pixf->ads[PIXEL_16x16] = x264_pixel_ads4##cpu;\
    pixf->ads[PIXEL_16x8] = x264_pixel_ads2##cpu;\
    pixf->ads[PIXEL_8x8] = x264_pixel_ads1##cpu;
    //8个sad函数
    INIT8( sad, );
    INIT8_NAME( sad_aligned, sad, );
    //7个sad函数-一次性计算3次
    INIT7( sad_x3, );
    //7个sad函数-一次性计算4次
    INIT7( sad_x4, );
    //8个ssd函数
```

```
    //ssd可以用来计算PSNR
    INIT8( ssd, );
    //8个satd函数
    //satd计算的是经过Hadamard变换后的值
    INIT8( satd, );
    //8个satd函数-一次性计算3次
    INIT7( satd_x3, );
    //8个satd函数-一次性计算4次
    INIT7( satd_x4, );
    INIT4( hadamard_ac, );
    INIT_ADS( );

    pixf->sa8d[PIXEL_16x16] = x264_pixel_sa8d_16x16;
    pixf->sa8d[PIXEL_8x8]   = x264_pixel_sa8d_8x8;
    pixf->var[PIXEL_16x16] = x264_pixel_var_16x16;
    pixf->var[PIXEL_8x16]  = x264_pixel_var_8x16;
    pixf->var[PIXEL_8x8]   = x264_pixel_var_8x8;
    pixf->var2[PIXEL_8x16]  = x264_pixel_var2_8x16;
    pixf->var2[PIXEL_8x8]   = x264_pixel_var2_8x8;
    //计算UV的
    pixf->ssd_nv12_core = pixel_ssd_nv12_core;
    //计算SSIM
    pixf->ssim_4x4x2_core = ssim_4x4x2_core;
    pixf->ssim_end4 = ssim_end4;
    pixf->vsad = pixel_vsad;
    pixf->asd8 = pixel_asd8;

    pixf->intra_sad_x3_4x4    = x264_intra_sad_x3_4x4;
    pixf->intra_satd_x3_4x4   = x264_intra_satd_x3_4x4;
    pixf->intra_sad_x3_8x8    = x264_intra_sad_x3_8x8;
    pixf->intra_sa8d_x3_8x8   = x264_intra_sa8d_x3_8x8;
    pixf->intra_sad_x3_8x8c   = x264_intra_sad_x3_8x8c;
    pixf->intra_satd_x3_8x8c  = x264_intra_satd_x3_8x8c;
    pixf->intra_sad_x3_8x16c  = x264_intra_sad_x3_8x16c;
    pixf->intra_satd_x3_8x16c = x264_intra_satd_x3_8x16c;
    pixf->intra_sad_x3_16x16  = x264_intra_sad_x3_16x16;
    pixf->intra_satd_x3_16x16 = x264_intra_satd_x3_16x16;

    //后面的初始化基本上都是汇编优化过的函数

#if HIGH_BIT_DEPTH
#if HAVE_MMX
    if( cpu&X264_CPU_MMX2 )
    {
        INIT7( sad, _mmx2 );
        INIT7_NAME( sad_aligned, sad, _mmx2 );
        INIT7( sad_x3, _mmx2 );
        INIT7( sad_x4, _mmx2 );
        INIT8( satd, _mmx2 );
        INIT7( satd_x3, _mmx2 );
        INIT7( satd_x4, _mmx2 );
        INIT4( hadamard_ac, _mmx2 );
        INIT8( ssd, _mmx2 );
        INIT_ADS( _mmx2 );

        pixf->ssd_nv12_core = x264_pixel_ssd_nv12_core_mmx2;
        pixf->var[PIXEL_16x16] = x264_pixel_var_16x16_mmx2;
        pixf->var[PIXEL_8x8]   = x264_pixel_var_8x8_mmx2;
#if ARCH_X86
        pixf->var2[PIXEL_8x8]  = x264_pixel_var2_8x8_mmx2;
        pixf->var2[PIXEL_8x16] = x264_pixel_var2_8x16_mmx2;
#endif

        pixf->intra_sad_x3_4x4    = x264_intra_sad_x3_4x4_mmx2;
        pixf->intra_satd_x3_4x4   = x264_intra_satd_x3_4x4_mmx2;
        pixf->intra_sad_x3_8x8    = x264_intra_sad_x3_8x8_mmx2;
        pixf->intra_sad_x3_8x8c   = x264_intra_sad_x3_8x8c_mmx2;
        pixf->intra_satd_x3_8x8c  = x264_intra_satd_x3_8x8c_mmx2;
        pixf->intra_sad_x3_8x16c  = x264_intra_sad_x3_8x16c_mmx2;
        pixf->intra_satd_x3_8x16c = x264_intra_satd_x3_8x16c_mmx2;
        pixf->intra_sad_x3_16x16  = x264_intra_sad_x3_16x16_mmx2;
        pixf->intra_satd_x3_16x16 = x264_intra_satd_x3_16x16_mmx2;
    }
    if( cpu&X264_CPU_SSE2 )
    {
        INIT4_NAME( sad_aligned, sad, _sse2_aligned );
        INIT5( ssd, _sse2 );
        INIT6( satd, _sse2 );
        pixf->satd[PIXEL_4x16] = x264_pixel_satd_4x16_sse2;

        pixf->sa8d[PIXEL_16x16] = x264_pixel_sa8d_16x16_sse2;
        pixf->sa8d[PIXEL_8x8]   = x264_pixel_sa8d_8x8_sse2;
#if ARCH_X86_64
        pixf->intra_sa8d_x3_8x8 = x264_intra_sa8d_x3_8x8_sse2;
        pixf->sa8d_satd[PIXEL_16x16] = x264_pixel_sa8d_satd_16x16_sse2;
#endif
        pixf->intra_sad_x3_4x4  = x264_intra_sad_x3_4x4_sse2;
        pixf->ssd_nv12_core = x264_pixel_ssd_nv12_core_sse2;
        pixf->ssim_4x4x2_core  = x264_pixel_ssim_4x4x2_core_sse2;
        pixf->ssim_end4        = x264_pixel_ssim_end4_sse2;
        pixf->var[PIXEL_16x16] = x264_pixel_var_16x16_sse2;
```

```
        pixf->var[PIXEL_8x8]   = x264_pixel_var_8x8_sse2;
        pixf->var2[PIXEL_8x8]  = x264_pixel_var2_8x8_sse2;
        pixf->var2[PIXEL_8x16] = x264_pixel_var2_8x16_sse2;
        pixf->intra_sad_x3_8x8 = x264_intra_sad_x3_8x8_sse2;
}
//此处省略大量的X86、ARM等平台的汇编函数初始化代码
}
```

x264_pixel_init()的源代码非常的长，主要原因在于它把C语言版本的函数以及各种平台的汇编函数都写到一块了（不知道现在最新的版本是不是还是这样）。x264_pixel_init()包含了大量和像素计算有关的函数，包括SAD、SATD、SSD、SSIM等等。它的输入参数x264_pixel_function_t是一个结构体，其中包含了各种像素计算的函数接口。x264_pixel_function_t的定义如下所示。

```cpp
typedef struct
{
    x264_pixel_cmp_t  sad[8];
    x264_pixel_cmp_t  ssd[8];
    x264_pixel_cmp_t satd[8];
    x264_pixel_cmp_t ssim[7];
    x264_pixel_cmp_t sa8d[4];
    x264_pixel_cmp_t mbcmp[8]; /* either satd or sad for subpel refine and mode decision */
    x264_pixel_cmp_t mbcmp_unaligned[8]; /* unaligned mbcmp for subpel */
    x264_pixel_cmp_t fpelcmp[8]; /* either satd or sad for fullpel motion search */
    x264_pixel_cmp_x3_t fpelcmp_x3[7];
    x264_pixel_cmp_x4_t fpelcmp_x4[7];
    x264_pixel_cmp_t sad_aligned[8]; /* Aligned SAD for mbcmp */
    int (*vsad)( pixel *, intptr_t, int );
    int (*asd8)( pixel *pix1, intptr_t stride1, pixel *pix2, intptr_t stride2, int height );
    uint64_t (*sa8d_satd[1])( pixel *pix1, intptr_t stride1, pixel *pix2, intptr_t stride2 );

    uint64_t (*var[4])( pixel *pix, intptr_t stride );
    int (*var2[4])( pixel *pix1, intptr_t stride1,
                    pixel *pix2, intptr_t stride2, int *ssd );
    uint64_t (*hadamard_ac[4])( pixel *pix, intptr_t stride );

    void (*ssd_nv12_core)( pixel *pixuv1, intptr_t stride1,
                           pixel *pixuv2, intptr_t stride2, int width, int height,
                           uint64_t *ssd_u, uint64_t *ssd_v );
    void (*ssim_4x4x2_core)( const pixel *pix1, intptr_t stride1,
                             const pixel *pix2, intptr_t stride2, int sums[2][4] );
    float (*ssim_end4)( int sum0[5][4], int sum1[5][4], int width );

    /* multiple parallel calls to cmp. */
    x264_pixel_cmp_x3_t sad_x3[7];
    x264_pixel_cmp_x4_t sad_x4[7];
    x264_pixel_cmp_x3_t satd_x3[7];
    x264_pixel_cmp_x4_t satd_x4[7];

    /* abs-diff-sum for successive elimination.
     * may round width up to a multiple of 16. */
    int (*ads[7])( int enc_dc[4], uint16_t *sums, int delta,
                   uint16_t *cost_mvx, int16_t *mvs, int width, int thresh );

    /* calculate satd or sad of V, H, and DC modes. */
    void (*intra_mbcmp_x3_16x16)( pixel *fenc, pixel *fdec, int res[3] );
    void (*intra_satd_x3_16x16) ( pixel *fenc, pixel *fdec, int res[3] );
    void (*intra_sad_x3_16x16)  ( pixel *fenc, pixel *fdec, int res[3] );
    void (*intra_mbcmp_x3_4x4)  ( pixel *fenc, pixel *fdec, int res[3] );
    void (*intra_satd_x3_4x4)   ( pixel *fenc, pixel *fdec, int res[3] );
    void (*intra_sad_x3_4x4)    ( pixel *fenc, pixel *fdec, int res[3] );
    void (*intra_mbcmp_x3_chroma)( pixel *fenc, pixel *fdec, int res[3] );
    void (*intra_satd_x3_chroma) ( pixel *fenc, pixel *fdec, int res[3] );
    void (*intra_sad_x3_chroma)  ( pixel *fenc, pixel *fdec, int res[3] );
    void (*intra_mbcmp_x3_8x16c) ( pixel *fenc, pixel *fdec, int res[3] );
    void (*intra_satd_x3_8x16c)  ( pixel *fenc, pixel *fdec, int res[3] );
    void (*intra_sad_x3_8x16c)   ( pixel *fenc, pixel *fdec, int res[3] );
    void (*intra_mbcmp_x3_8x8c)  ( pixel *fenc, pixel *fdec, int res[3] );
    void (*intra_satd_x3_8x8c)   ( pixel *fenc, pixel *fdec, int res[3] );
    void (*intra_sad_x3_8x8c)    ( pixel *fenc, pixel *fdec, int res[3] );
    void (*intra_mbcmp_x3_8x8)  ( pixel *fenc, pixel edge[36], int res[3] );
    void (*intra_sa8d_x3_8x8)   ( pixel *fenc, pixel edge[36], int res[3] );
    void (*intra_sad_x3_8x8)    ( pixel *fenc, pixel edge[36], int res[3] );
    /* find minimum satd or sad of all modes, and set fdec.
     * may be NULL, in which case just use pred+satd instead. */
    int (*intra_mbcmp_x9_4x4)( pixel *fenc, pixel *fdec, uint16_t *bitcosts );
    int (*intra_satd_x9_4x4) ( pixel *fenc, pixel *fdec, uint16_t *bitcosts );
    int (*intra_sad_x9_4x4)  ( pixel *fenc, pixel *fdec, uint16_t *bitcosts );
    int (*intra_mbcmp_x9_8x8)( pixel *fenc, pixel *fdec, pixel edge[36], uint16_t *bitcosts, uint16_t *satds );
    int (*intra_sa8d_x9_8x8) ( pixel *fenc, pixel *fdec, pixel edge[36], uint16_t *bitcosts, uint16_t *satds );
    int (*intra_sad_x9_8x8)  ( pixel *fenc, pixel *fdec, pixel edge[36], uint16_t *bitcosts, uint16_t *satds );
} x264_pixel_function_t;
```

在x264_pixel_init()中定义了好几个宏，用于给x264_pixel_function_t结构体中的函数接口赋值。例如“INIT8(sad,)”用于给x264_pixel_function_t中的sad[8]赋值。该宏展开后的代码如下。

```cpp
pixf->sad[PIXEL_16x16] = x264_pixel_sad_16x16;
pixf->sad[PIXEL_16x8]  = x264_pixel_sad_16x8;
pixf->sad[PIXEL_8x16]  = x264_pixel_sad_8x16;
pixf->sad[PIXEL_8x8]   = x264_pixel_sad_8x8;
pixf->sad[PIXEL_8x4]   = x264_pixel_sad_8x4;
pixf->sad[PIXEL_4x8]   = x264_pixel_sad_4x8;
pixf->sad[PIXEL_4x4]   = x264_pixel_sad_4x4;
pixf->sad[PIXEL_4x16]  = x264_pixel_sad_4x16;
```

“INIT8(ssd,)” 用于给x264_pixel_function_t中的ssd[8]赋值。该宏展开后的代码如下。

```cpp
pixf->ssd[PIXEL_16x16] = x264_pixel_ssd_16x16;
pixf->ssd[PIXEL_16x8]  = x264_pixel_ssd_16x8;
pixf->ssd[PIXEL_8x16]  = x264_pixel_ssd_8x16;
pixf->ssd[PIXEL_8x8]   = x264_pixel_ssd_8x8;
pixf->ssd[PIXEL_8x4]   = x264_pixel_ssd_8x4;
pixf->ssd[PIXEL_4x8]   = x264_pixel_ssd_4x8;
pixf->ssd[PIXEL_4x4]   = x264_pixel_ssd_4x4;
pixf->ssd[PIXEL_4x16]  = x264_pixel_ssd_4x16;
```

“INIT8(satd,)” 用于给x264_pixel_function_t中的satd[8]赋值。该宏展开后的代码如下。

```cpp
pixf->satd[PIXEL_16x16] = x264_pixel_satd_16x16;
pixf->satd[PIXEL_16x8]  = x264_pixel_satd_16x8;
pixf->satd[PIXEL_8x16]  = x264_pixel_satd_8x16;
pixf->satd[PIXEL_8x8]   = x264_pixel_satd_8x8;
pixf->satd[PIXEL_8x4]   = x264_pixel_satd_8x4;
pixf->satd[PIXEL_4x8]   = x264_pixel_satd_4x8;
pixf->satd[PIXEL_4x4]   = x264_pixel_satd_4x4;
pixf->satd[PIXEL_4x16]  = x264_pixel_satd_4x16;
```

下文打算分别记录SAD、SSD和SATD计算的函数x264_pixel_sad_4x4()，x264_pixel_ssd_4x4()，和x264_pixel_satd_4x4()。此外再记录一个一次性“批量”计算4个点的函数x264_pixel_sad_x4_4x4()。

相关知识简述

简单记录几个像素计算中的概念。SAD和SATD主要用于帧内预测模式以及帧间预测模式的判断。有关SAD、SATD、SSD的定义如下：

SAD（Sum of Absolute Difference）也可以称为SAE（Sum of Absolute Error），即绝对误差和。它的计算方法就是求出两个像素块对应像素点的差值，将这些差值分别求绝对值之后再进行累加。

SATD（Sum of Absolute Transformed Difference）即Hadamard变换后再绝对值求和。它和SAD的区别在于多了一个“变换”。

SSD（Sum of Squared Difference）也可以称为SSE（Sum of Squared Error），即差值的平方和。它和SAD的区别在于多了一个“平方”。

H.264中使用SAD和SATD进行宏块预测模式的判断。早期的编码器使用SAD进行计算，近期的编码器多使用SATD进行计算。为什么使用SATD而不使用SAD呢？关键原因在于编码之后码流的大小是和图像块DCT变换后频域信息紧密相关的，而和变换前的时域信息关联性小一些。SAD只能反应时域信息；SATD却可以反映频域信息，而且计算复杂度也低于DCT变换，因此是比较合适的模式选择的依据。

使用SAD进行模式选择的示例如下所示。下面这张图代表了一个普通的Intra16x16的宏块的像素。它的下方包含了使用Vertical，Horizontal，DC和Plane四种帧内预测模式预测的像素。通过计算可以得到这几种预测像素和原始像素之间的SAD（SAE）分别为3985，5097，4991，2539。由于Plane模式的SAD取值最小，由此可以断定Plane模式对于这个宏块来说是最好的帧内预测模式。

x264_pixel_sad_4x4()

x264_pixel_sad_4x4()用于计算4x4块的SAD。该函数的定义位于common\pixel.c，如下所示。

[cpp]

```cpp
  static int x264_pixel_sad_4x4( pixel *pix1, intptr_t i_stride_pix1,
                 pixel *pix2, intptr_t i_stride_pix2 )
{
    int i_sum = 0;
    for( int y = 0; y < 4; y++ ) //4个像素
    {
        for( int x = 0; x < 4; x++ ) //4个像素
        {
            i_sum += abs( pix1[x] - pix2[x] );//相减之后求绝对值，然后累加
        }
        pix1 += i_stride_pix1;
        pix2 += i_stride_pix2;
    }
    return i_sum;
}
```

可以看出x264_pixel_sad_4x4()将两个4x4图像块对应点相减之后，调用abs()求出绝对值，然后累加到i_sum变量上。

x264_pixel_sad_x4_4x4()

x264_pixel_sad_4x4()用于计算4个4x4块的SAD。该函数的定义位于common\pixel.c，如下所示。

[cpp]

```cpp
static void x264_pixel_sad_x4_4x4( pixel *fenc, pixel *pix0, pixel *pix1,pixel *pix2, pixel *pix3,
                                      intptr_t i_stride, int scores[4] )
{
    scores[0] = x264_pixel_sad_4x4( fenc, 16, pix0, i_stride );
    scores[1] = x264_pixel_sad_4x4( fenc, 16, pix1, i_stride );
    scores[2] = x264_pixel_sad_4x4( fenc, 16, pix2, i_stride );
    scores[3] = x264_pixel_sad_4x4( fenc, 16, pix3, i_stride );
}
```

可以看出，x264_pixel_sad_4x4()计算了起始点在pix0，pix1，pix2，pix3四个4x4的图像块和fenc之间的SAD，并将结果存储于scores[4]数组中。

x264_pixel_ssd_4x4()

x264_pixel_ssd_4x4()用于计算4x4块的SSD。该函数的定义位于common\pixel.c，如下所示。

[cpp]

```cpp
static int x264_pixel_ssd_4x4( pixel *pix1, intptr_t i_stride_pix1,
                 pixel *pix2, intptr_t i_stride_pix2 )
{
    int i_sum = 0;
    for( int y = 0; y < 4; y++ ) //4个像素
    {
        for( int x = 0; x < 4; x++ ) //4个像素
        {
            int d = pix1[x] - pix2[x]; //相减
            i_sum += d*d;              //平方之后，累加
        }
        pix1 += i_stride_pix1;
        pix2 += i_stride_pix2;
    }
    return i_sum;
}
```

可以看出x264_pixel_ssd_4x4()将两个4x4图像块对应点相减之后，取了平方值，然后累加到i_sum变量上。

x264_pixel_satd_4x4()

x264_pixel_satd_4x4()用于计算4x4块的SATD。该函数的定义位于common\pixel.c，如下所示。

[cpp]

```cpp
//SAD（Sum of Absolute Difference）=SAE（Sum of Absolute Error)即绝对误差和
//SATD（Sum of Absolute Transformed Difference）即hadamard变换后再绝对值求和
//
//为什么帧内模式选择要用SATD？
//SAD即绝对误差和，仅反映残差时域差异，影响PSNR值，不能有效反映码流的大小。
//SATD即将残差经哈德曼变换的4x4块的预测残差绝对值总和，可以将其看作简单的时频变换，其值在一定程度上可以反映生成码流的大小。
//4x4的SATD
static NOINLINE int x264_pixel_satd_4x4( pixel *pix1, intptr_t i_pix1, pixel *pix2, intptr_t i_pix2 )
{
    sum2_t tmp[4][2];
    sum2_t a0, a1, a2, a3, b0, b1;
    sum2_t sum = 0;

    for( int i = 0; i < 4; i++, pix1 += i_pix1, pix2 += i_pix2 )
    {
        a0 = pix1[0] - pix2[0];
        a1 = pix1[1] - pix2[1];
        b0 = (a0+a1) + ((a0-a1)<<BITS_PER_SUM);
        a2 = pix1[2] - pix2[2];
        a3 = pix1[3] - pix2[3];
        b1 = (a2+a3) + ((a2-a3)<<BITS_PER_SUM);
        tmp[i][0] = b0 + b1;
        tmp[i][1] = b0 - b1;
    }
    for( int i = 0; i < 2; i++ )
    {
        HADAMARD4( a0, a1, a2, a3, tmp[0][i], tmp[1][i], tmp[2][i], tmp[3][i] );
        a0 = abs2(a0) + abs2(a1) + abs2(a2) + abs2(a3);
        sum += ((sum_t)a0) + (a0>>BITS_PER_SUM);
    }
    return sum >> 1;
}
```

有关x264_pixel_satd_4x4()中的Hadamard变换在下面的DCT变换中再进行分析。可以看出该函数调用了一个宏HADAMARD4()用于Hadamard变换的计算，并最终将两个像素块Hadamard变换后对应元素求差的绝对值之后，累加到sum变量上。

x264_dct_init()

x264_dct_init()用于初始化DCT变换和DCT反变换相关的汇编函数。该函数的定义位于common\dct.c，如下所示。

[cpp]

```cpp
/******************************************************************************
 * x264_dct_init:
 ******************************************************************************/
void x264_dct_init( int cpu, x264_dct_function_t *dctf )
{
    //C语言版本
    //4x4DCT变换
    dctf->sub4x4_dct    = sub4x4_dct;
    dctf->add4x4_idct   = add4x4_idct;
    //8x8块：分解成4个4x4DCT变换，调用4次sub4x4_dct()
    dctf->sub8x8_dct    = sub8x8_dct;
    dctf->sub8x8_dct_dc = sub8x8_dct_dc;
    dctf->add8x8_idct   = add8x8_idct;
    dctf->add8x8_idct_dc = add8x8_idct_dc;

    dctf->sub8x16_dct_dc = sub8x16_dct_dc;
    //16x16块：分解成4个8x8块，调用4次sub8x8_dct()
    //实际上每个sub8x8_dct()又分解成4个4x4DCT变换，调用4次sub4x4_dct()
    dctf->sub16x16_dct  = sub16x16_dct;
    dctf->add16x16_idct = add16x16_idct;
    dctf->add16x16_idct_dc = add16x16_idct_dc;
    //8x8DCT，注意：后缀是_dct8
    dctf->sub8x8_dct8   = sub8x8_dct8;
    dctf->add8x8_idct8  = add8x8_idct8;

    dctf->sub16x16_dct8  = sub16x16_dct8;
    dctf->add16x16_idct8 = add16x16_idct8;
    //Hadamard变换
    dctf->dct4x4dc  = dct4x4dc;
    dctf->idct4x4dc = idct4x4dc;

    dctf->dct2x4dc = dct2x4dc;

#if HIGH_BIT_DEPTH
#if HAVE_MMX
    if( cpu&X264_CPU_MMX )
    {
        dctf->sub4x4_dct    = x264_sub4x4_dct_mmx;
        dctf->sub8x8_dct    = x264_sub8x8_dct_mmx;
        dctf->sub16x16_dct  = x264_sub16x16_dct_mmx;
    }
    if( cpu&X264_CPU_SSE2 )
    {
        dctf->add4x4_idct     = x264_add4x4_idct_sse2;
        dctf->dct4x4dc        = x264_dct4x4dc_sse2;
        dctf->idct4x4dc       = x264_idct4x4dc_sse2;
        dctf->sub8x8_dct8     = x264_sub8x8_dct8_sse2;
        dctf->sub16x16_dct8   = x264_sub16x16_dct8_sse2;
        dctf->add8x8_idct     = x264_add8x8_idct_sse2;
        dctf->add16x16_idct   = x264_add16x16_idct_sse2;
        dctf->add8x8_idct8    = x264_add8x8_idct8_sse2;
        dctf->add16x16_idct8    = x264_add16x16_idct8_sse2;
        dctf->sub8x8_dct_dc   = x264_sub8x8_dct_dc_sse2;
        dctf->add8x8_idct_dc  = x264_add8x8_idct_dc_sse2;
        dctf->sub8x16_dct_dc  = x264_sub8x16_dct_dc_sse2;
        dctf->add16x16_idct_dc= x264_add16x16_idct_dc_sse2;
    }
    if( cpu&X264_CPU_SSE4 )
    {
        dctf->sub8x8_dct8     = x264_sub8x8_dct8_sse4;
        dctf->sub16x16_dct8   = x264_sub16x16_dct8_sse4;
    }
    if( cpu&X264_CPU_AVX )
    {
        dctf->add4x4_idct     = x264_add4x4_idct_avx;
        dctf->dct4x4dc        = x264_dct4x4dc_avx;
        dctf->idct4x4dc       = x264_idct4x4dc_avx;
        dctf->sub8x8_dct8     = x264_sub8x8_dct8_avx;
        dctf->sub16x16_dct8   = x264_sub16x16_dct8_avx;
        dctf->add8x8_idct     = x264_add8x8_idct_avx;
        dctf->add16x16_idct   = x264_add16x16_idct_avx;
        dctf->add8x8_idct8    = x264_add8x8_idct8_avx;
        dctf->add16x16_idct8  = x264_add16x16_idct8_avx;
        dctf->add8x8_idct_dc  = x264_add8x8_idct_dc_avx;
        dctf->sub8x16_dct_dc  = x264_sub8x16_dct_dc_avx;
        dctf->add16x16_idct_dc= x264_add16x16_idct_dc_avx;
    }
#endif // HAVE_MMX
#else // !HIGH_BIT_DEPTH
    //MMX版本
#if HAVE_MMX
    if( cpu&X264_CPU_MMX )
    {
        dctf->sub4x4_dct    = x264_sub4x4_dct_mmx;
        dctf->add4x4_idct   = x264_add4x4_idct_mmx;
        dctf->idct4x4dc     = x264_idct4x4dc_mmx;
        dctf->sub8x8_dct_dc = x264_sub8x8_dct_dc_mmx2;
    //此处省略大量的X86、ARM等平台的汇编函数初始化代码
}
```

从源代码可以看出，x264_dct_init()初始化了一系列的DCT变换的函数，这些DCT函数名称有如下规律：

（1）DCT函数名称前面有“sub”，代表对两块像素相减得到残差之后，再进行DCT变换。

（2）DCT反变换函数名称前面有“add”，代表将DCT反变换之后的残差数据叠加到预测数据上。

（3）以“dct8”为结尾的函数使用了8x8DCT，其余函数是用的都是4x4DCT。

x264_dct_init()的输入参数x264_dct_function_t是一个结构体，其中包含了各种DCT函数的接口。x264_dct_function_t的定义如下所示。

[cpp]

```cpp
typedef struct
{
    // pix1  stride = FENC_STRIDE
    // pix2  stride = FDEC_STRIDE
    // p_dst stride = FDEC_STRIDE
    void (*sub4x4_dct)   ( dctcoef dct[16], pixel *pix1, pixel *pix2 );
    void (*add4x4_idct)  ( pixel *p_dst, dctcoef dct[16] );

    void (*sub8x8_dct)   ( dctcoef dct[4][16], pixel *pix1, pixel *pix2 );
    void (*sub8x8_dct_dc)( dctcoef dct[4], pixel *pix1, pixel *pix2 );
    void (*add8x8_idct)  ( pixel *p_dst, dctcoef dct[4][16] );
    void (*add8x8_idct_dc) ( pixel *p_dst, dctcoef dct[4] );

    void (*sub8x16_dct_dc)( dctcoef dct[8], pixel *pix1, pixel *pix2 );

    void (*sub16x16_dct) ( dctcoef dct[16][16], pixel *pix1, pixel *pix2 );
    void (*add16x16_idct)( pixel *p_dst, dctcoef dct[16][16] );
    void (*add16x16_idct_dc) ( pixel *p_dst, dctcoef dct[16] );

    void (*sub8x8_dct8)  ( dctcoef dct[64], pixel *pix1, pixel *pix2 );
    void (*add8x8_idct8) ( pixel *p_dst, dctcoef dct[64] );

    void (*sub16x16_dct8) ( dctcoef dct[4][64], pixel *pix1, pixel *pix2 );
    void (*add16x16_idct8)( pixel *p_dst, dctcoef dct[4][64] );

    void (*dct4x4dc) ( dctcoef d[16] );
    void (*idct4x4dc)( dctcoef d[16] );

    void (*dct2x4dc)( dctcoef dct[8], dctcoef dct4x4[8][16] );

} x264_dct_function_t;
```

x264_dct_init()的工作就是对x264_dct_function_t中的函数指针进行赋值。由于DCT函数很多，不便于一一研究，下文仅举例分析几个典型的4x4DCT函数：4x4DCT变换函数sub4x4_dct()，4x4IDCT变换函数add4x4_idct()，8x8块的4x4DCT变换函数sub8x8_dct()，16x16块的4x4DCT变换函数sub16x16_dct()，4x4Hadamard变换函数dct4x4dc()。

相关知识简述

简单记录一下DCT相关的知识。DCT变换的核心理念就是把图像的低频信息（对应大面积平坦区域）变换到系数矩阵的左上角，而把高频信息变换到系数矩阵的右下角，这样就可以在压缩的时候（量化）去除掉人眼不敏感的高频信息（位于矩阵右下角的系数）从而达到压缩数据的目的。二维8x8DCT变换常见的示意图如下所示。

早期的DCT变换都使用了8x8的矩阵（变换系数为小数）。在H.264标准中新提出了一种4x4的矩阵。这种4x4 DCT变换的系数都是整数，一方面提高了运算的准确性，一方面也利于代码的优化。4x4整数DCT变换的示意图如下所示（作为对比，右侧为4x4块的Hadamard变换的示意图）。

4x4整数DCT变换的公式如下所示。

对该公式中的矩阵乘法可以转换为2次一维DCT变换：首先对4x4块中的每行像素进行一维DCT变换，然后再对4x4块中的每列像素进行一维DCT变换。而一维的DCT变换是可以改造成为蝶形快速算法的，如下所示。

同理，DCT反变换就是DCT变换的逆变换。DCT反变换的公式如下所示。

同理，DCT反变换的矩阵乘法也可以改造成为2次一维IDCT变换：首先对4x4块中的每行像素进行一维IDCT变换，然后再对4x4块中的每列像素进行一维IDCT变换。而一维的IDCT变换也可以改造成为蝶形快速算法，如下所示。

除了4x4DCT变换之外，新版本的H.264标准中还引入了一种8x8DCT。目前针对这种8x8DCT我还没有做研究，暂时不做记录。

sub4x4_dct()

sub4x4_dct()可以将两块4x4的图像相减求残差后，进行DCT变换。该函数的定义位于common\dct.c，如下所示。

```cpp
/*
 * 求残差用
 * 注意求的是一个“方块”形像素
 *
 * 参数的含义如下：
 * diff：输出的残差数据
 * i_size：方块的大小
 * pix1：输入数据1
 * i_pix1：输入数据1一行像素大小（stride）
 * pix2：输入数据2
 * i_pix2：输入数据2一行像素大小（stride）
 *
 */
static inline void pixel_sub_wxh( dctcoef *diff, int i_size,
                                  pixel *pix1, int i_pix1, pixel *pix2, int i_pix2 )
{
    for( int y = 0; y < i_size; y++ )
    {
        for( int x = 0; x < i_size; x++ )
            diff[x + y*i_size] = pix1[x] - pix2[x];//求残差
        pix1 += i_pix1;//前进到下一行
        pix2 += i_pix2;
    }
}
//4x4DCT变换
//注意首先获取pix1和pix2两块数据的残差，然后再进行变换
//返回dct[16]
static void sub4x4_dct( dctcoef dct[16], pixel *pix1, pixel *pix2 )
{
    dctcoef d[16];
    dctcoef tmp[16];
    //获取残差数据，存入d[16]
    //pix1一般为编码帧（enc）
    //pix2一般为重建帧（dec）
    pixel_sub_wxh( d, 4, pix1, FENC_STRIDE, pix2, FDEC_STRIDE );

    //处理残差d[16]
    //蝶形算法：横向4个像素
    for( int i = 0; i < 4; i++ )
    {
        int s03 = d[i*4+0] + d[i*4+3];
        int s12 = d[i*4+1] + d[i*4+2];
        int d03 = d[i*4+0] - d[i*4+3];
        int d12 = d[i*4+1] - d[i*4+2];

        tmp[0*4+i] =   s03 +   s12;
        tmp[1*4+i] = 2*d03 +   d12;
        tmp[2*4+i] =   s03 -   s12;
        tmp[3*4+i] =   d03 - 2*d12;
    }
    //蝶形算法：纵向
    for( int i = 0; i < 4; i++ )
    {
        int s03 = tmp[i*4+0] + tmp[i*4+3];
        int s12 = tmp[i*4+1] + tmp[i*4+2];
        int d03 = tmp[i*4+0] - tmp[i*4+3];
        int d12 = tmp[i*4+1] - tmp[i*4+2];

        dct[i*4+0] =   s03 +   s12;
        dct[i*4+1] = 2*d03 +   d12;
        dct[i*4+2] =   s03 -   s12;
        dct[i*4+3] =   d03 - 2*d12;
    }
}
```

从源代码可以看出，sub4x4_dct()首先调用pixel_sub_wxh()求出两个输入图像块的残差，然后使用蝶形快速算法计算残差图像的DCT系数。

add4x4_idct()

add4x4_idct()可以将残差数据进行DCT反变换，并将变换后得到的残差像素数据叠加到预测数据上。该函数的定义位于common\dct.c，如下所示。

```cpp
//4x4DCT反变换（“add”代表叠加到已有的像素上）
static void add4x4_idct( pixel *p_dst, dctcoef dct[16] )
{
    dctcoef d[16];
    dctcoef tmp[16];

    for( int i = 0; i < 4; i++ )
    {
        int s02 =  dct[0*4+i]     +  dct[2*4+i];
        int d02 =  dct[0*4+i]     -  dct[2*4+i];
        int s13 =  dct[1*4+i]     + (dct[3*4+i]>>1);
        int d13 = (dct[1*4+i]>>1) -  dct[3*4+i];

        tmp[i*4+0] = s02 + s13;
        tmp[i*4+1] = d02 + d13;
        tmp[i*4+2] = d02 - d13;
        tmp[i*4+3] = s02 - s13;
    }

    for( int i = 0; i < 4; i++ )
    {
        int s02 =  tmp[0*4+i]     +  tmp[2*4+i];
        int d02 =  tmp[0*4+i]     -  tmp[2*4+i];
        int s13 =  tmp[1*4+i]     + (tmp[3*4+i]>>1);
        int d13 = (tmp[1*4+i]>>1) -  tmp[3*4+i];

        d[0*4+i] = ( s02 + s13 + 32 ) >> 6;
        d[1*4+i] = ( d02 + d13 + 32 ) >> 6;
        d[2*4+i] = ( d02 - d13 + 32 ) >> 6;
        d[3*4+i] = ( s02 - s13 + 32 ) >> 6;
    }


    for( int y = 0; y < 4; y++ )
    {
        for( int x = 0; x < 4; x++ )
            p_dst[x] = x264_clip_pixel( p_dst[x] + d[y*4+x] );
        p_dst += FDEC_STRIDE;
    }
}
```

从源代码可以看出，add4x4_idct()首先采用快速蝶形算法对DCT系数进行DCT反变换后得到残差像素数据，然后再将残差数据叠加到p_dst指向的像素上。需要注意这里是“叠加”而不是“赋值”。

sub8x8_dct()

sub8x8_dct()可以将两块8x8的图像相减求残差后，进行4x4DCT变换。该函数的定义位于common\dct.c，如下所示。

```cpp
//8x8块：分解成4个4x4DCT变换，调用4次sub4x4_dct()
//返回dct[4][16]
static void sub8x8_dct( dctcoef dct[4][16], pixel *pix1, pixel *pix2 )
{
    /*
     * 8x8 宏块被划分为4个4x4子块
     *
     * +---+---+
     * | 0 | 1 |
     * +---+---+
     * | 2 | 3 |
     * +---+---+
     *
     */
    sub4x4_dct( dct[0], &pix1[0], &pix2[0] );
    sub4x4_dct( dct[1], &pix1[4], &pix2[4] );
    sub4x4_dct( dct[2], &pix1[4*FENC_STRIDE+0], &pix2[4*FDEC_STRIDE+0] );
    sub4x4_dct( dct[3], &pix1[4*FENC_STRIDE+4], &pix2[4*FDEC_STRIDE+4] );
}
```

从源代码可以看出， sub8x8_dct()将8x8的图像块分成4个4x4的图像块，分别调用了sub4x4_dct()。

sub16x16_dct()

sub16x16_dct()可以将两块16x16的图像相减求残差后，进行4x4DCT变换。该函数的定义位于common\dct.c，如下所示。

[cpp]

```cpp
//16x16块：分解成4个8x8的块做DCT变换，调用4次sub8x8_dct()
//返回dct[16][16]
static void sub16x16_dct( dctcoef dct[16][16], pixel *pix1, pixel *pix2 )
{
    /*
     * 16x16 宏块被划分为4个8x8子块
     *
     * +--------+--------+
     * |        |        |
     * |   0    |   1    |
     * |        |        |
     * +--------+--------+
     * |        |        |
     * |   2    |   3    |
     * |        |        |
     * +--------+--------+
     *
     */
    sub8x8_dct( &dct[ 0], &pix1[0], &pix2[0] );  //0
    sub8x8_dct( &dct[ 4], &pix1[8], &pix2[8] );  //1
    sub8x8_dct( &dct[ 8], &pix1[8*FENC_STRIDE+0], &pix2[8*FDEC_STRIDE+0] );  //2
    sub8x8_dct( &dct[12], &pix1[8*FENC_STRIDE+8], &pix2[8*FDEC_STRIDE+8] );  //3
}
```

从源代码可以看出， sub8x8_dct()将16x16的图像块分成4个8x8的图像块，分别调用了sub8x8_dct()。而sub8x8_dct()实际上又调用了4次sub4x4_dct()。所以可以得知，不论sub16x16_dct()，sub8x8_dct()还是sub4x4_dct()，本质都是进行4x4DCT。

dct4x4dc()

dct4x4dc()可以将输入的4x4图像块进行Hadamard变换。该函数的定义位于common\dct.c，如下所示。

[cpp]

```cpp
//Hadamard变换
static void dct4x4dc( dctcoef d[16] )
{
    dctcoef tmp[16];

    //蝶形算法：横向的4个像素
    for( int i = 0; i < 4; i++ )
    {

        int s01 = d[i*4+0] + d[i*4+1];
        int d01 = d[i*4+0] - d[i*4+1];
        int s23 = d[i*4+2] + d[i*4+3];
        int d23 = d[i*4+2] - d[i*4+3];

        tmp[0*4+i] = s01 + s23;
        tmp[1*4+i] = s01 - s23;
        tmp[2*4+i] = d01 - d23;
        tmp[3*4+i] = d01 + d23;
    }
    //蝶形算法：纵向
    for( int i = 0; i < 4; i++ )
    {
        int s01 = tmp[i*4+0] + tmp[i*4+1];
        int d01 = tmp[i*4+0] - tmp[i*4+1];
        int s23 = tmp[i*4+2] + tmp[i*4+3];
        int d23 = tmp[i*4+2] - tmp[i*4+3];

        d[i*4+0] = ( s01 + s23 + 1 ) >> 1;
        d[i*4+1] = ( s01 - s23 + 1 ) >> 1;
        d[i*4+2] = ( d01 - d23 + 1 ) >> 1;
        d[i*4+3] = ( d01 + d23 + 1 ) >> 1;
    }
}
```

从源代码可以看出，dct4x4dc()实现了Hadamard快速蝶形算法。

x264_mc_init()

x264_mc_init()用于初始化运动补偿相关的汇编函数。该函数的定义位于common\mc.c，如下所示。

[cpp]

```cpp
//运动补偿
void x264_mc_init( int cpu, x264_mc_functions_t *pf, int cpu_independent )
{
    //亮度运动补偿
    pf->mc_luma   = mc_luma;
    //获得匹配块
    pf->get_ref   = get_ref;

    pf->mc_chroma = mc_chroma;
    //求平均
    pf->avg[PIXEL_16x16]= pixel_avg_16x16;
    pf->avg[PIXEL_16x8] = pixel_avg_16x8;
    pf->avg[PIXEL_8x16] = pixel_avg_8x16;
    pf->avg[PIXEL_8x8]  = pixel_avg_8x8;
    pf->avg[PIXEL_8x4]  = pixel_avg_8x4;
    pf->avg[PIXEL_4x16] = pixel_avg_4x16;
    pf->avg[PIXEL_4x8]  = pixel_avg_4x8;
    pf->avg[PIXEL_4x4]  = pixel_avg_4x4;
    pf->avg[PIXEL_4x2]  = pixel_avg_4x2;
    pf->avg[PIXEL_2x8]  = pixel_avg_2x8;
    pf->avg[PIXEL_2x4]  = pixel_avg_2x4;
    pf->avg[PIXEL_2x2]  = pixel_avg_2x2;
    //加权相关
    pf->weight    = x264_mc_weight_wtab;
    pf->offsetadd = x264_mc_weight_wtab;
    pf->offsetsub = x264_mc_weight_wtab;
    pf->weight_cache = x264_weight_cache;
    //赋值-只包含了方形的
    pf->copy_16x16_unaligned = mc_copy_w16;
    pf->copy[PIXEL_16x16] = mc_copy_w16;
    pf->copy[PIXEL_8x8]   = mc_copy_w8;
    pf->copy[PIXEL_4x4]   = mc_copy_w4;

    pf->store_interleave_chroma       = store_interleave_chroma;
    pf->load_deinterleave_chroma_fenc = load_deinterleave_chroma_fenc;
    pf->load_deinterleave_chroma_fdec = load_deinterleave_chroma_fdec;
    //拷贝像素-不论像素块大小
    pf->plane_copy = x264_plane_copy_c;
    pf->plane_copy_interleave = x264_plane_copy_interleave_c;
    pf->plane_copy_deinterleave = x264_plane_copy_deinterleave_c;
    pf->plane_copy_deinterleave_rgb = x264_plane_copy_deinterleave_rgb_c;
    pf->plane_copy_deinterleave_v210 = x264_plane_copy_deinterleave_v210_c;
    //关键：半像素内插
    pf->hpel_filter = hpel_filter;
    //几个空函数
    pf->prefetch_fenc_420 = prefetch_fenc_null;
    pf->prefetch_fenc_422 = prefetch_fenc_null;
    pf->prefetch_ref  = prefetch_ref_null;
    pf->memcpy_aligned = memcpy;
    pf->memzero_aligned = memzero_aligned;
    //降低分辨率-线性内插（不是半像素内插）
    pf->frame_init_lowres_core = frame_init_lowres_core;

    pf->integral_init4h = integral_init4h;
    pf->integral_init8h = integral_init8h;
    pf->integral_init4v = integral_init4v;
    pf->integral_init8v = integral_init8v;

    pf->mbtree_propagate_cost = mbtree_propagate_cost;
    pf->mbtree_propagate_list = mbtree_propagate_list;
    //各种汇编版本
#if HAVE_MMX
    x264_mc_init_mmx( cpu, pf );
#endif
#if HAVE_ALTIVEC
    if( cpu&X264_CPU_ALTIVEC )
        x264_mc_altivec_init( pf );
#endif
#if HAVE_ARMV6
    x264_mc_init_arm( cpu, pf );
#endif
#if ARCH_AARCH64
    x264_mc_init_aarch64( cpu, pf );
#endif

    if( cpu_independent )
    {
        pf->mbtree_propagate_cost = mbtree_propagate_cost;
        pf->mbtree_propagate_list = mbtree_propagate_list;
    }
}
```

从源代码可以看出，x264_mc_init()中包含了大量的像素内插、拷贝、求平均的函数。这些函数都是用于在H.264编码过程中进行运动估计和运动补偿的。x264_mc_init()的参数x264_mc_functions_t是一个结构体，其中包含了运动补偿函数相关的函数接口。x264_mc_functions_t的定义如下。

```cpp
typedef struct
{
    void (*mc_luma)( pixel *dst, intptr_t i_dst, pixel **src, intptr_t i_src,
                     int mvx, int mvy, int i_width, int i_height, const x264_weight_t *weight );

    /* may round up the dimensions if they're not a power of 2 */
    pixel* (*get_ref)( pixel *dst, intptr_t *i_dst, pixel **src, intptr_t i_src,
                       int mvx, int mvy, int i_width, int i_height, const x264_weight_t *weight );

    /* mc_chroma may write up to 2 bytes of garbage to the right of dst,
     * so it must be run from left to right. */
    void (*mc_chroma)( pixel *dstu, pixel *dstv, intptr_t i_dst, pixel *src, intptr_t i_src,
                       int mvx, int mvy, int i_width, int i_height );

    void (*avg[12])( pixel *dst,  intptr_t dst_stride, pixel *src1, intptr_t src1_stride,
                     pixel *src2, intptr_t src2_stride, int i_weight );

    /* only 16x16, 8x8, and 4x4 defined */
    void (*copy[7])( pixel *dst, intptr_t dst_stride, pixel *src, intptr_t src_stride, int i_height );
    void (*copy_16x16_unaligned)( pixel *dst, intptr_t dst_stride, pixel *src, intptr_t src_stride, int i_height );

    void (*store_interleave_chroma)( pixel *dst, intptr_t i_dst, pixel *srcu, pixel *srcv, int height );
    void (*load_deinterleave_chroma_fenc)( pixel *dst, pixel *src, intptr_t i_src, int height );
    void (*load_deinterleave_chroma_fdec)( pixel *dst, pixel *src, intptr_t i_src, int height );

    void (*plane_copy)( pixel *dst, intptr_t i_dst, pixel *src, intptr_t i_src, int w, int h );
    void (*plane_copy_interleave)( pixel *dst,  intptr_t i_dst, pixel *srcu, intptr_t i_srcu,
                                   pixel *srcv, intptr_t i_srcv, int w, int h );
    /* may write up to 15 pixels off the end of each plane */
    void (*plane_copy_deinterleave)( pixel *dstu, intptr_t i_dstu, pixel *dstv, intptr_t i_dstv,
                                     pixel *src,  intptr_t i_src, int w, int h );
    void (*plane_copy_deinterleave_rgb)( pixel *dsta, intptr_t i_dsta, pixel *dstb, intptr_t i_dstb,
                                         pixel *dstc, intptr_t i_dstc, pixel *src,  intptr_t i_src, int pw, int w, int h );
    void (*plane_copy_deinterleave_v210)( pixel *dsty, intptr_t i_dsty,
                                          pixel *dstc, intptr_t i_dstc,
                                          uint32_t *src, intptr_t i_src, int w, int h );
    void (*hpel_filter)( pixel *dsth, pixel *dstv, pixel *dstc, pixel *src,
                         intptr_t i_stride, int i_width, int i_height, int16_t *buf );

    /* prefetch the next few macroblocks of fenc or fdec */
    void (*prefetch_fenc)    ( pixel *pix_y, intptr_t stride_y, pixel *pix_uv, intptr_t stride_uv, int mb_x );
    void (*prefetch_fenc_420)( pixel *pix_y, intptr_t stride_y, pixel *pix_uv, intptr_t stride_uv, int mb_x );
    void (*prefetch_fenc_422)( pixel *pix_y, intptr_t stride_y, pixel *pix_uv, intptr_t stride_uv, int mb_x );
    /* prefetch the next few macroblocks of a hpel reference frame */
    void (*prefetch_ref)( pixel *pix, intptr_t stride, int parity );

    void *(*memcpy_aligned)( void *dst, const void *src, size_t n );
    void (*memzero_aligned)( void *dst, size_t n );

    /* successive elimination prefilter */
    void (*integral_init4h)( uint16_t *sum, pixel *pix, intptr_t stride );
    void (*integral_init8h)( uint16_t *sum, pixel *pix, intptr_t stride );
    void (*integral_init4v)( uint16_t *sum8, uint16_t *sum4, intptr_t stride );
    void (*integral_init8v)( uint16_t *sum8, intptr_t stride );

    void (*frame_init_lowres_core)( pixel *src0, pixel *dst0, pixel *dsth, pixel *dstv, pixel *dstc,
                                    intptr_t src_stride, intptr_t dst_stride, int width, int height );
    weight_fn_t *weight;
    weight_fn_t *offsetadd;
    weight_fn_t *offsetsub;
    void (*weight_cache)( x264_t *, x264_weight_t * );

    void (*mbtree_propagate_cost)( int16_t *dst, uint16_t *propagate_in, uint16_t *intra_costs,
                                   uint16_t *inter_costs, uint16_t *inv_qscales, float *fps_factor, int len );

    void (*mbtree_propagate_list)( x264_t *h, uint16_t *ref_costs, int16_t (*mvs)[2],
                                   int16_t *propagate_amount, uint16_t *lowres_costs,
                                   int bipred_weight, int mb_y, int len, int list );
} x264_mc_functions_t;
```

x264_mc_init()的工作就是对x264_mc_functions_t中的函数指针进行赋值。由于运动估计和运动补偿在x264中属于相对复杂的环节，其中许多函数的作用很难三言两语表述出来，因此只举一个相对简单的例子——半像素内插函数hpel_filter()。

相关知识简述

简单记录一下半像素插值的知识。《H.264标准》中规定，运动估计为1/4像素精度。因此在H.264编码和解码的过程中，需要将画面中的像素进行插值——简单地说就是把原先的1个像素点拓展成4x4一共16个点。下图显示了H.264编码和解码过程中像素插值情况。可以看出原先的G点的右下方通过插值的方式产生了a、b、c、d等一共16个点。

如图所示，1/4像素内插一般分成两步：

（1）半像素内插。这一步通过6抽头滤波器获得5个半像素点。

（2）线性内插。这一步通过简单的线性内插获得剩余的1/4像素点。

图中半像素内插点为b、m、h、s、j五个点。半像素内插方法是对整像素点进行6 抽头滤波得出，滤波器的权重为(1/32, -5/32, 5/8, 5/8, -5/32, 1/32)。例如b的计算公式为：

b=round((E - 5F + 20G + 20H - 5I + J) / 32)

剩下几个半像素点的计算关系如下：

m：由B、D、H、N、S、U计算

h：由A、C、G、M、R、T计算

s：由K、L、M、N、P、Q计算

j：由cc、dd、h、m、ee、ff计算。需要注意j点的运算量比较大，因为cc、dd、ee、ff都需要通过半像素内插方法进行计算。

在获得半像素点之后，就可以通过简单的线性内插获得1/4像素内插点了。1/4像素内插的方式如下图所示。例如图中a点的计算公式如下：

A=round((G+b)/2)

在这里有一点需要注意：位于4个角的e、g、p、r四个点并不是通过j点计算计算的，而是通过b、h、s、m四个半像素点计算的。

hpel_filter()

hpel_filter()用于进行半像素插值。该函数的定义位于common\mc.c，如下所示。

[cpp]

```cpp
//半像素插值公式
//b= (E - 5F + 20G + 20H - 5I + J)/32
//              x
//d取1，水平滤波器；d取stride，垂直滤波器（这里没有除以32）
#define TAPFILTER(pix, d) ((pix)[x-2*d] + (pix)[x+3*d] - 5*((pix)[x-d] + (pix)[x+2*d]) + 20*((pix)[x] + (pix)[x+d]))

/*
 * 半像素插值
 * dsth：水平滤波得到的半像素点(aa,bb,b,s,gg,hh)
 * dstv：垂直滤波的到的半像素点(cc,dd,h,m,ee,ff)
 * dstc：“水平+垂直”滤波得到的位于4个像素中间的半像素点（j）
 *
 * 半像素插值示意图如下：
 *
 *         A aa B
 *
 *         C bb D
 *
 * E   F   G  b H   I   J
 *
 * cc  dd  h  j m  ee  ff
 *
 * K   L   M  s N   P   Q
 *
 *         R gg S
 *
 *         T hh U
 *
 * 计算公式如下：
 * b=round( (E - 5F + 20G + 20H - 5I + J ) / 32)
 *
 * 剩下几个半像素点的计算关系如下：
 * m：由B、D、H、N、S、U计算
 * h：由A、C、G、M、R、T计算
 * s：由K、L、M、N、P、Q计算
 * j：由cc、dd、h、m、ee、ff计算。需要注意j点的运算量比较大，因为cc、dd、ee、ff都需要通过半像素内插方法进行计算。
 *
 */
static void hpel_filter( pixel *dsth, pixel *dstv, pixel *dstc, pixel *src,
                         intptr_t stride, int width, int height, int16_t *buf )
{
    const int pad = (BIT_DEPTH > 9) ? (-10 * PIXEL_MAX) : 0;
    /*
     * 几种半像素点之间的位置关系
     *
     * X： 像素点
     * H：水平滤波半像素点
     * V：垂直滤波半像素点
     * C： 中间位置半像素点
     *
     * X   H   X       X       X
     *
     * V   C
     *
     * X       X       X       X
     *
     *
     *
     * X       X       X       X
     *
     */
    //一行一行处理
    for( int y = 0; y < height; y++ )
    {
        //一个一个点处理
        //每个整像素点都对应h，v，c三个半像素点
        //v
        for( int x = -2; x < width+3; x++ )//(aa,bb,b,s,gg,hh),结果存入buf
        {
            //垂直滤波半像素点
            int v = TAPFILTER(src,stride);
            dstv[x] = x264_clip_pixel( (v + 16) >> 5 );
            /* transform v for storage in a 16-bit integer */
            //这应该是给dstc计算使用的？
            buf[x+2] = v + pad;
        }
        //c
        for( int x = 0; x < width; x++ )
            dstc[x] = x264_clip_pixel( (TAPFILTER(buf+2,1) - 32*pad + 512) >> 10 );//四个相邻像素中间的半像素点
        //h
        for( int x = 0; x < width; x++ )
            dsth[x] = x264_clip_pixel( (TAPFILTER(src,1) + 16) >> 5 );//水平滤波半像素点
        dsth += stride;
        dstv += stride;
        dstc += stride;
        src += stride;
    }
}
```

从源代码可以看出，hpel_filter()中包含了一个宏TAPFILTER()用来完成半像素点像素值的计算。在完成半像素插值工作后，dsth中存储的是经过水平插值后的半像素点，dstv中存储的是经过垂直插值后的半像素点，dstc中存储的是位于4个相邻像素点中间位置的半像素点。这三块内存中的点的位置关系如下图所示（灰色的点是整像素点）。

x264_quant_init()

x264_quant_init()初始化量化和反量化相关的汇编函数。该函数的定义位于common\quant.c，如下所示。

[cpp]

```cpp
//量化
void x264_quant_init( x264_t *h, int cpu, x264_quant_function_t *pf )
{
    //这个好像是针对8x8DCT的
    pf->quant_8x8 = quant_8x8;

    //量化4x4=16个
    pf->quant_4x4 = quant_4x4;
    //注意：处理4个4x4的块
    pf->quant_4x4x4 = quant_4x4x4;
    //Intra16x16中，16个DC系数Hadamard变换后对的它们量化
    pf->quant_4x4_dc = quant_4x4_dc;
    pf->quant_2x2_dc = quant_2x2_dc;
    //反量化4x4=16个
    pf->dequant_4x4 = dequant_4x4;
    pf->dequant_4x4_dc = dequant_4x4_dc;
    pf->dequant_8x8 = dequant_8x8;

    pf->idct_dequant_2x4_dc = idct_dequant_2x4_dc;
    pf->idct_dequant_2x4_dconly = idct_dequant_2x4_dconly;

    pf->optimize_chroma_2x2_dc = optimize_chroma_2x2_dc;
    pf->optimize_chroma_2x4_dc = optimize_chroma_2x4_dc;

    pf->denoise_dct = x264_denoise_dct;
    pf->decimate_score15 = x264_decimate_score15;
    pf->decimate_score16 = x264_decimate_score16;
    pf->decimate_score64 = x264_decimate_score64;

    pf->coeff_last4 = x264_coeff_last4;
    pf->coeff_last8 = x264_coeff_last8;
    pf->coeff_last[  DCT_LUMA_AC] = x264_coeff_last15;
    pf->coeff_last[ DCT_LUMA_4x4] = x264_coeff_last16;
    pf->coeff_last[ DCT_LUMA_8x8] = x264_coeff_last64;
    pf->coeff_level_run4 = x264_coeff_level_run4;
    pf->coeff_level_run8 = x264_coeff_level_run8;
    pf->coeff_level_run[  DCT_LUMA_AC] = x264_coeff_level_run15;
    pf->coeff_level_run[ DCT_LUMA_4x4] = x264_coeff_level_run16;

#if HIGH_BIT_DEPTH
#if HAVE_MMX
    INIT_TRELLIS( sse2 );
    if( cpu&X264_CPU_MMX2 )
    {
#if ARCH_X86
        pf->denoise_dct = x264_denoise_dct_mmx;
        pf->decimate_score15 = x264_decimate_score15_mmx2;
        pf->decimate_score16 = x264_decimate_score16_mmx2;
        pf->decimate_score64 = x264_decimate_score64_mmx2;
        pf->coeff_last8 = x264_coeff_last8_mmx2;
        pf->coeff_last[  DCT_LUMA_AC] = x264_coeff_last15_mmx2;
        pf->coeff_last[ DCT_LUMA_4x4] = x264_coeff_last16_mmx2;
        pf->coeff_last[ DCT_LUMA_8x8] = x264_coeff_last64_mmx2;
        pf->coeff_level_run8 = x264_coeff_level_run8_mmx2;
        pf->coeff_level_run[  DCT_LUMA_AC] = x264_coeff_level_run15_mmx2;
        pf->coeff_level_run[ DCT_LUMA_4x4] = x264_coeff_level_run16_mmx2;
#endif
        pf->coeff_last4 = x264_coeff_last4_mmx2;
        pf->coeff_level_run4 = x264_coeff_level_run4_mmx2;
        if( cpu&X264_CPU_LZCNT )
            pf->coeff_level_run4 = x264_coeff_level_run4_mmx2_lzcnt;
    }
    //此处省略大量的X86、ARM等平台的汇编函数初始化代码
}
```

从源代码可以看出，x264_quant_init ()初始化了一系列的量化相关的函数。它的输入参数x264_quant_function_t是一个结构体，其中包含了和量化相关各种函数指针。x264_quant_function_t的定义如下所示。

[cpp]

```cpp
typedef struct
{
    int (*quant_8x8)  ( dctcoef dct[64], udctcoef mf[64], udctcoef bias[64] );
    int (*quant_4x4)  ( dctcoef dct[16], udctcoef mf[16], udctcoef bias[16] );
    int (*quant_4x4x4)( dctcoef dct[4][16], udctcoef mf[16], udctcoef bias[16] );
    int (*quant_4x4_dc)( dctcoef dct[16], int mf, int bias );
    int (*quant_2x2_dc)( dctcoef dct[4], int mf, int bias );

    void (*dequant_8x8)( dctcoef dct[64], int dequant_mf[6][64], int i_qp );
    void (*dequant_4x4)( dctcoef dct[16], int dequant_mf[6][16], int i_qp );
    void (*dequant_4x4_dc)( dctcoef dct[16], int dequant_mf[6][16], int i_qp );

    void (*idct_dequant_2x4_dc)( dctcoef dct[8], dctcoef dct4x4[8][16], int dequant_mf[6][16], int i_qp );
    void (*idct_dequant_2x4_dconly)( dctcoef dct[8], int dequant_mf[6][16], int i_qp );

    int (*optimize_chroma_2x2_dc)( dctcoef dct[4], int dequant_mf );
    int (*optimize_chroma_2x4_dc)( dctcoef dct[8], int dequant_mf );

    void (*denoise_dct)( dctcoef *dct, uint32_t *sum, udctcoef *offset, int size );

    int (*decimate_score15)( dctcoef *dct );
    int (*decimate_score16)( dctcoef *dct );
    int (*decimate_score64)( dctcoef *dct );
    int (*coeff_last[14])( dctcoef *dct );
    int (*coeff_last4)( dctcoef *dct );
    int (*coeff_last8)( dctcoef *dct );
    int (*coeff_level_run[13])( dctcoef *dct, x264_run_level_t *runlevel );
    int (*coeff_level_run4)( dctcoef *dct, x264_run_level_t *runlevel );
    int (*coeff_level_run8)( dctcoef *dct, x264_run_level_t *runlevel );

#define TRELLIS_PARAMS const int *unquant_mf, const uint8_t *zigzag, int lambda2,\
                       int last_nnz, dctcoef *coefs, dctcoef *quant_coefs, dctcoef *dct,\
                       uint8_t *cabac_state_sig, uint8_t *cabac_state_last,\
                       uint64_t level_state0, uint16_t level_state1
    int (*trellis_cabac_4x4)( TRELLIS_PARAMS, int b_ac );
    int (*trellis_cabac_8x8)( TRELLIS_PARAMS, int b_interlaced );
    int (*trellis_cabac_4x4_psy)( TRELLIS_PARAMS, int b_ac, dctcoef *fenc_dct, int psy_trellis );
    int (*trellis_cabac_8x8_psy)( TRELLIS_PARAMS, int b_interlaced, dctcoef *fenc_dct, int psy_trellis );
    int (*trellis_cabac_dc)( TRELLIS_PARAMS, int num_coefs );
    int (*trellis_cabac_chroma_422_dc)( TRELLIS_PARAMS );
} x264_quant_function_t;
```

x264_quant_init ()的工作就是对x264_quant_function_t中的函数指针进行赋值。下文举例分析其中2个函数：4x4矩阵量化函数quant_4x4()，4个4x4矩阵量化函数quant_4x4x4()。

相关知识简述

简单记录一下量化的概念。量化是H.264视频压缩编码中对视频质量影响最大的地方，也是会导致“信息丢失”的地方。量化的原理可以表示为下面公式：

FQ=round(y/Qstep)

其中，y 为输入样本点编码，Qstep为量化步长，FQ 为y 的量化值，round()为取整函数（其输出为与输入实数最近的整数）。其相反过程，即反量化为：

y'＝FQ*Qstep

如果Qstep较大，则量化值FQ取值较小，其相应的编码长度较小，但是但反量化时损失较多的图像细节信息。简而言之，Qstep越大，视频压缩编码后体积越小，视频质量越差。

在H.264 中，量化步长Qstep 共有52 个值，如下表所示。其中QP 是量化参数，是量化步长的序号。当QP 取最小值0 时代表最精细的量化，当QP 取最大值51 时代表最粗糙的量化。QP 每增加6，Qstep 增加一倍。

《H.264标准》中规定，量化过程除了完成本职工作外，还需要完成它前一步DCT变换中“系数相乘”的工作。这一步骤的推导过程不再记录，直接给出最终的公式（这个公式完全为整数运算，同时避免了除法的使用）：

|Zij| = (|Wij|*MF + f)>>qbits

sign(Zij) = sign (Wij)

其中：

sign()为符号函数。

Wij为DCT变换后的系数。

MF的值如下表所示。表中只列出对应QP 值为0 到5 的MF 值。QP大于6之后，将QP实行对6取余数操作，再找到MF的值。

qbits计算公式为“qbits = 15 + floor(QP/6)”。即它的值随QP 值每增加6 而增加1。

f 是偏移量（用于改善恢复图像的视觉效果）。对帧内预测图像块取2^qbits/3，对帧间预测图像块取2^qbits/6。

为了更形象的显示MF的取值，做了下面一张示意图。图中深蓝色代表MF取值较大的点，而浅蓝色代表MF取值较小的点。

quant_4x4()

quant_4x4()用于对4x4的DCT残差矩阵进行量化。该函数的定义位于common\quant.c，如下所示。

```cpp
//4x4量化
//输入输出都是dct[16]
static int quant_4x4( dctcoef dct[16], udctcoef mf[16], udctcoef bias[16] )
{
    int nz = 0;
    //循环16个元素
    for( int i = 0; i < 16; i++ )
        QUANT_ONE( dct[i], mf[i], bias[i] );
    return !!nz;
}
```

可以看出quant_4x4()循环16次调用了QUANT_ONE()完成了量化工作。并且将DCT系数值，MF值，bias偏移值直接传递给了该宏。

QUANT_ONE()

QUANT_ONE()完成了一个DCT系数的量化工作，它的定义如下。

```cpp
//量化1个元素
#define QUANT_ONE( coef, mf, f ) \
{ \
    if( (coef) > 0 ) \
        (coef) = (f + (coef)) * (mf) >> 16; \
    else \
        (coef) = - ((f - (coef)) * (mf) >> 16); \
    nz |= (coef); \
}
```

从QUANT_ONE()的定义可以看出，它实现了上文提到的H.264标准中的量化公式。

quant_4x4x4()

quant_4x4x4()用于对4个4x4的DCT残差矩阵进行量化。该函数的定义位于common\quant.c，如下所示。

```cpp
//处理4个4x4量化
//输入输出都是dct[4][16]
static int quant_4x4x4( dctcoef dct[4][16], udctcoef mf[16], udctcoef bias[16] )
{
    int nza = 0;
    //处理4个
    for( int j = 0; j < 4; j++ )
    {
        int nz = 0;
        //量化
        for( int i = 0; i < 16; i++ )
            QUANT_ONE( dct[j][i], mf[i], bias[i] );
        nza |= (!!nz)<<j;
    }
    return nza;
}
```

从quant_4x4x4()的定义可以看出，该函数相当于调用了4次quant_4x4()函数。

x264_deblock_init()

x264_deblock_init()用于初始化去块效应滤波器相关的汇编函数。该函数的定义位于common\deblock.c，如下所示。

[cpp]

```cpp
//去块效应滤波
void x264_deblock_init( int cpu, x264_deblock_function_t *pf, int b_mbaff )
{
    //注意：标记"v"的垂直滤波器是处理水平边界用的
    //亮度-普通滤波器-边界强度Bs=1,2,3
    pf->deblock_luma[1] = deblock_v_luma_c;
    pf->deblock_luma[0] = deblock_h_luma_c;
    //色度的
    pf->deblock_chroma[1] = deblock_v_chroma_c;
    pf->deblock_h_chroma_420 = deblock_h_chroma_c;
    pf->deblock_h_chroma_422 = deblock_h_chroma_422_c;
    //亮度-强滤波器-边界强度Bs=4
    pf->deblock_luma_intra[1] = deblock_v_luma_intra_c;
    pf->deblock_luma_intra[0] = deblock_h_luma_intra_c;
    pf->deblock_chroma_intra[1] = deblock_v_chroma_intra_c;
    pf->deblock_h_chroma_420_intra = deblock_h_chroma_intra_c;
    pf->deblock_h_chroma_422_intra = deblock_h_chroma_422_intra_c;
    pf->deblock_luma_mbaff = deblock_h_luma_mbaff_c;
    pf->deblock_chroma_420_mbaff = deblock_h_chroma_mbaff_c;
    pf->deblock_luma_intra_mbaff = deblock_h_luma_intra_mbaff_c;
    pf->deblock_chroma_420_intra_mbaff = deblock_h_chroma_intra_mbaff_c;
    pf->deblock_strength = deblock_strength_c;

#if HAVE_MMX
    if( cpu&X264_CPU_MMX2 )
    {
#if ARCH_X86
        pf->deblock_luma[1] = x264_deblock_v_luma_mmx2;
        pf->deblock_luma[0] = x264_deblock_h_luma_mmx2;
        pf->deblock_chroma[1] = x264_deblock_v_chroma_mmx2;
        pf->deblock_h_chroma_420 = x264_deblock_h_chroma_mmx2;
        pf->deblock_chroma_420_mbaff = x264_deblock_h_chroma_mbaff_mmx2;
        pf->deblock_h_chroma_422 = x264_deblock_h_chroma_422_mmx2;
        pf->deblock_h_chroma_422_intra = x264_deblock_h_chroma_422_intra_mmx2;
        pf->deblock_luma_intra[1] = x264_deblock_v_luma_intra_mmx2;
        pf->deblock_luma_intra[0] = x264_deblock_h_luma_intra_mmx2;
        pf->deblock_chroma_intra[1] = x264_deblock_v_chroma_intra_mmx2;
        pf->deblock_h_chroma_420_intra = x264_deblock_h_chroma_intra_mmx2;
        pf->deblock_chroma_420_intra_mbaff = x264_deblock_h_chroma_intra_mbaff_mmx2;
#endif
    //此处省略大量的X86、ARM等平台的汇编函数初始化代码
}
```

从源代码可以看出，x264_deblock_init()中初始化了一系列环路滤波函数。这些函数名称的规则如下：

（1）包含"v"的是垂直滤波器，用于处理水平边界；包含"h"的是水平滤波器，用于处理垂直边界。

（2）包含"luma"的是亮度滤波器，包含"chroma"的是色度滤波器。

（3）包含"intra"的是处理边界强度Bs为4的强滤波器，不包含"intra"的是普通滤波器。

x264_deblock_init()的输入参数x264_deblock_function_t是一个结构体，其中包含了环路滤波器相关的函数指针。x264_deblock_function_t的定义如下所示。

[cpp]

```cpp
typedef struct
{
    x264_deblock_inter_t deblock_luma[2];
    x264_deblock_inter_t deblock_chroma[2];
    x264_deblock_inter_t deblock_h_chroma_420;
    x264_deblock_inter_t deblock_h_chroma_422;
    x264_deblock_intra_t deblock_luma_intra[2];
    x264_deblock_intra_t deblock_chroma_intra[2];
    x264_deblock_intra_t deblock_h_chroma_420_intra;
    x264_deblock_intra_t deblock_h_chroma_422_intra;
    x264_deblock_inter_t deblock_luma_mbaff;
    x264_deblock_inter_t deblock_chroma_mbaff;
    x264_deblock_inter_t deblock_chroma_420_mbaff;
    x264_deblock_inter_t deblock_chroma_422_mbaff;
    x264_deblock_intra_t deblock_luma_intra_mbaff;
    x264_deblock_intra_t deblock_chroma_intra_mbaff;
    x264_deblock_intra_t deblock_chroma_420_intra_mbaff;
    x264_deblock_intra_t deblock_chroma_422_intra_mbaff;
    void (*deblock_strength) ( uint8_t nnz[X264_SCAN8_SIZE], int8_t ref[2][X264_SCAN8_LUMA_SIZE],
                               int16_t mv[2][X264_SCAN8_LUMA_SIZE][2], uint8_t bs[2][8][4], int mvy_limit,
                               int bframe );
} x264_deblock_function_t;
```

x264_deblock_init()的工作就是对x264_deblock_function_t中的函数指针进行赋值。可以看出x264_deblock_function_t中很多的元素是一个包含2个元素的数组，例如deblock_luma[2]，deblock_luma_intra[2]等。这些数组中的元素[0]一般是水平滤波器，而元素[1]是垂直滤波器。下文将会举例分析一个普通边界的亮度垂直滤波器函数deblock_v_luma_c()。

相关知识简述

简单记录一下环路滤波（去块效应滤波）的知识。X264的重建帧（通过解码得到）一般情况下会出现方块效应。产生这种效应的原因主要有两个：

（1）DCT变换后的量化造成误差（主要原因）。
（2）运动补偿

正是由于这种块效应的存在，才需要添加环路滤波器调整相邻的“块”边缘上的像素值以减轻这种视觉上的不连续感。下面一张图显示了环路滤波的效果。图中左边的图没有使用环路滤波，而右边的图使用了环路滤波。

环路滤波分类

环路滤波器根据滤波的强度可以分为两种：

（1）普通滤波器。针对边界的Bs（边界强度）为1、2、3的滤波器。此时环路滤波涉及到方块边界周围的6个点（边界两边各3个点）：p2，p1，p0，q0，q1，q2。需要处理4个点（边界两边各2个点，只以p点为例）：

p0’ = p0 + (((q0 - p0) << 2) + (p1 - q1) + 4) >> 3
p1’ = (p2 + ((p0 + q0 + 1) >> 1) – 2p1) >> 1

（2）强滤波器。针对边界的Bs（边界强度）为4的滤波器。此时环路滤波涉及到方块边界周围的8个点（边界两边各4个点）：p3，p2，p1，p0，q0，q1，q2，q3。需要处理6个点（边界两边各3个点，只以p点为例）：

p0’ = (p2 + 2*p1 + 2*p0 + 2*q0 + q1 + 4) >> 3
p1’ = (p2 + p1 + p0 + q0 + 2) >> 2
p2’ = (2*p3 + 3*p2 + p1 + p0 + q0 + 4) >> 3

其中上文中提到的边界强度Bs的判定方式如下。

条件（针对两边的图像块）	Bs
有一个块为帧内预测 + 边界为宏块边界	4
有一个块为帧内预测	3
有一个块对残差编码	2
运动矢量差不小于1像素	1
运动补偿参考帧不同	1
其它	0

总体说来，与帧内预测相关的图像块（帧内预测块）的边界强度比较大，取值为3或者4；与运动补偿相关的图像块（帧间预测块）的边界强度比较小，取值为1。

环路滤波的门限

并不是所有的块的边界处都需要环路滤波。例如画面中物体的边界正好和块的边界重合的话，就不能进行滤波，否则会使画面中物体的边界变模糊。因此需要区别开物体边界和块效应边界。一般情况下，物体边界两边的像素值差别很大，而块效应边界两边像素值差别比较小。《H.264标准》以这个特点定义了2个变量alpha和beta来判决边界是否需要进行环路滤波。只有满足下面三个条件的时候才能进行环路滤波：

| p0 - q0 | < alpha
| p1 – p0 | < beta
| q1 - q0 | < beta

简而言之，就是边界两边的两个点的像素值不能太大，即不能超过alpha；边界一边的前两个点之间的像素值也不能太大，即不能超过beta。其中alpha和beta是根据量化参数QP推算出来（具体方法不再记录）。总体说来QP越大，alpha和beta的值也越大，也就越容易触发环路滤波。由于QP越大表明压缩的程度越大，所以也可以得知高压缩比的情况下更需要进行环路滤波。

deblock_v_luma_c()

deblock_v_luma_c()是一个普通强度的垂直滤波器，用于处理边界强度Bs为1，2，3的水平边界。该函数的定义位于common\deblock.c，如下所示。

[cpp]

```cpp
//去块效应滤波-普通滤波，Bs为1,2,3
//垂直（Vertical）滤波器
//      边界
//         x
//         x
// 边界----------
//         x
//         x
//
//
static void deblock_v_luma_c( pixel *pix, intptr_t stride, int alpha, int beta, int8_t *tc0 )
{
    //xstride=stride（用于选择滤波的像素）
    //ystride=1
    deblock_luma_c( pix, stride, 1, alpha, beta, tc0 );
}
```

可以看出deblock_v_luma_c()调用了另一个函数deblock_luma_c()。需要注意传递给deblock_luma_c()是一个水平滤波器和垂直滤波器都会调用的"通用"滤波器函数。在这里传递给deblock_luma_c()第二个参数xstride的值为stride，第三个参数ystride的值为1。

deblock_luma_c()

deblock_luma_c()是一个通用的滤波器函数，定义如下所示。

[cpp]

```cpp
//去块效应滤波-普通滤波，Bs为1,2,3
static inline void deblock_luma_c( pixel *pix, intptr_t xstride, intptr_t ystride, int alpha, int beta, int8_t *tc0 )
{
    for( int i = 0; i < 4; i++ )
    {
        if( tc0[i] < 0 )
        {
            pix += 4*ystride;
            continue;
        }
        //滤4个像素
        for( int d = 0; d < 4; d++, pix += ystride )
            deblock_edge_luma_c( pix, xstride, alpha, beta, tc0[i] );
    }
}
```

从源代码中可以看出，具体的滤波在deblock_edge_luma_c()中完成。处理完一个像素后，会继续处理与当前像素距离为ystride的像素。

deblock_edge_luma_c()

deblock_edge_luma_c()用于完成具体的滤波工作。该函数的定义如下所示。

```cpp
/* From ffmpeg */
//去块效应滤波-普通滤波，Bs为1,2,3
//从FFmpeg复制过来的？
static ALWAYS_INLINE void deblock_edge_luma_c( pixel *pix, intptr_t xstride, int alpha, int beta, int8_t tc0 )
{
    //p和q
    //如果xstride=stride，ystride=1
    //就是处理纵向的6个像素
    //对应的是方块的横向边界的滤波，即如下所示：
    //        p2
    //        p1
    //        p0
    //=====图像边界=====
    //        q0
    //        q1
    //        q2
    //
    //如果xstride=1，ystride=stride
    //就是处理纵向的6个像素
    //对应的是方块的横向边界的滤波，即如下所示：
    //          ||
    // p2 p1 p0 || q0 q1 q2
    //          ||
    //          边界

    //注意：这里乘的是xstride

    int p2 = pix[-3*xstride];
    int p1 = pix[-2*xstride];
    int p0 = pix[-1*xstride];
    int q0 = pix[ 0*xstride];
    int q1 = pix[ 1*xstride];
    int q2 = pix[ 2*xstride];
    //计算方法参考相关的标准
    //alpha和beta是用于检查图像内容的2个参数
    //只有满足if()里面3个取值条件的时候（只涉及边界旁边的4个点），才会滤波
    if( abs( p0 - q0 ) < alpha && abs( p1 - p0 ) < beta && abs( q1 - q0 ) < beta )
    {
        int tc = tc0;
        int delta;
        //上面2个点（p0，p2）满足条件的时候，滤波p1
        //int x264_clip3( int v, int i_min, int i_max )用于限幅
        if( abs( p2 - p0 ) < beta )
        {
            if( tc0 )
                pix[-2*xstride] = p1 + x264_clip3( (( p2 + ((p0 + q0 + 1) >> 1)) >> 1) - p1, -tc0, tc0 );
            tc++;
        }
        //下面2个点（q0，q2）满足条件的时候，滤波q1
        if( abs( q2 - q0 ) < beta )
        {
            if( tc0 )
                pix[ 1*xstride] = q1 + x264_clip3( (( q2 + ((p0 + q0 + 1) >> 1)) >> 1) - q1, -tc0, tc0 );
            tc++;
        }

        delta = x264_clip3( (((q0 - p0 ) << 2) + (p1 - q1) + 4) >> 3, -tc, tc );
        //p0
        pix[-1*xstride] = x264_clip_pixel( p0 + delta );    /* p0' */
        //q0
        pix[ 0*xstride] = x264_clip_pixel( q0 - delta );    /* q0' */
    }
}
```

从源代码可以看出，deblock_edge_luma_c()实现了前文记录的滤波公式。

deblock_h_luma_c()

deblock_h_luma_c()是一个普通强度的水平滤波器，用于处理边界强度Bs为1，2，3的垂直边界。该函数的定义如下所示。

```cpp
//去块效应滤波-普通滤波，Bs为1,2,3
//水平（Horizontal）滤波器
//      边界
//        |
// x x x | x x x
//        |
static void deblock_h_luma_c( pixel *pix, intptr_t stride, int alpha, int beta, int8_t *tc0 )
{
    //xstride=1（用于选择滤波的像素）
    //ystride=stride
    deblock_luma_c( pix, 1, stride, alpha, beta, tc0 );
}
```

从源代码可以看出，和deblock_v_luma_c()类似，deblock_h_luma_c()同样调用了deblock_luma_c()函数。唯一的不同在于它传递给deblock_luma_c()的第2个参数xstride为1，第3个参数ystride为stride。

mbcmp_init()

mbcmp_init()函数决定了x264_pixel_function_t中的像素比较的一系列函数（mbcmp[]）使用SAD还是SATD。该函数的定义位于encoder\encoder.c，如下所示。

```cpp
//决定了像素比较的时候用SAD还是SATD
static void mbcmp_init( x264_t *h )
{
    //b_lossless一般为0
    //主要看i_subpel_refine，大于1的话就使用SATD
    int satd = !h->mb.b_lossless && h->param.analyse.i_subpel_refine > 1;

    //sad或者satd赋值给mbcmp
    memcpy( h->pixf.mbcmp, satd ? h->pixf.satd : h->pixf.sad_aligned, sizeof(h->pixf.mbcmp) );
    memcpy( h->pixf.mbcmp_unaligned, satd ? h->pixf.satd : h->pixf.sad, sizeof(h->pixf.mbcmp_unaligned) );
    h->pixf.intra_mbcmp_x3_16x16 = satd ? h->pixf.intra_satd_x3_16x16 : h->pixf.intra_sad_x3_16x16;
    h->pixf.intra_mbcmp_x3_8x16c = satd ? h->pixf.intra_satd_x3_8x16c : h->pixf.intra_sad_x3_8x16c;
    h->pixf.intra_mbcmp_x3_8x8c  = satd ? h->pixf.intra_satd_x3_8x8c  : h->pixf.intra_sad_x3_8x8c;
    h->pixf.intra_mbcmp_x3_8x8 = satd ? h->pixf.intra_sa8d_x3_8x8 : h->pixf.intra_sad_x3_8x8;
    h->pixf.intra_mbcmp_x3_4x4 = satd ? h->pixf.intra_satd_x3_4x4 : h->pixf.intra_sad_x3_4x4;
    h->pixf.intra_mbcmp_x9_4x4 = h->param.b_cpu_independent || h->mb.b_lossless ? NULL
                               : satd ? h->pixf.intra_satd_x9_4x4 : h->pixf.intra_sad_x9_4x4;
    h->pixf.intra_mbcmp_x9_8x8 = h->param.b_cpu_independent || h->mb.b_lossless ? NULL
                               : satd ? h->pixf.intra_sa8d_x9_8x8 : h->pixf.intra_sad_x9_8x8;
    satd &= h->param.analyse.i_me_method == X264_ME_TESA;
    memcpy( h->pixf.fpelcmp, satd ? h->pixf.satd : h->pixf.sad, sizeof(h->pixf.fpelcmp) );
    memcpy( h->pixf.fpelcmp_x3, satd ? h->pixf.satd_x3 : h->pixf.sad_x3, sizeof(h->pixf.fpelcmp_x3) );
    memcpy( h->pixf.fpelcmp_x4, satd ? h->pixf.satd_x4 : h->pixf.sad_x4, sizeof(h->pixf.fpelcmp_x4) );
}
```

从mbcmp_init()的源代码可以看出，当i_subpel_refine取值大于1的时候，satd变量为1，此时后续代码中赋值给mbcmp[]相关的一系列函数指针的函数就是SATD函数；当i_subpel_refine取值小于等于1的时候，satd变量为0，此时后续代码中赋值给mbcmp[]相关的一系列函数指针的函数就是SAD函数。

至此x264_encoder_open()的源代码就分析完毕了。下文继续分析x264_encoder_headers()和x264_encoder_close()函数。

x264_encoder_headers()

x264_encoder_headers()是libx264的一个API函数，用于输出SPS/PPS/SEI这些H.264码流的头信息。该函数的声明如下。

```cpp
/* x264_encoder_headers:
 *      return the SPS and PPS that will be used for the whole stream.
 *      *pi_nal is the number of NAL units outputted in pp_nal.
 *      returns the number of bytes in the returned NALs.
 *      returns negative on error.
 *      the payloads of all output NALs are guaranteed to be sequential in memory. */
int     x264_encoder_headers( x264_t *, x264_nal_t **pp_nal, int *pi_nal );
```

x264_encoder_headers()的定义位于encoder\encoder.c，如下所示。

```cpp
/*******************************************************************************
 * x264_encoder_headers:
 * 注释和处理：雷霄骅
 * http://blog.csdn.net/leixiaohua1020
 * leixiaohua1020@126.com
 ******************************************************************************/
//输出文件头（SPS、PPS、SEI）
int x264_encoder_headers( x264_t *h, x264_nal_t **pp_nal, int *pi_nal )
{
    int frame_size = 0;
    /* init bitstream context */
    h->out.i_nal = 0;
    bs_init( &h->out.bs, h->out.p_bitstream, h->out.i_bitstream );

    /* Write SEI, SPS and PPS. */

    /* generate sequence parameters */
    //输出SPS
    x264_nal_start( h, NAL_SPS, NAL_PRIORITY_HIGHEST );
    x264_sps_write( &h->out.bs, h->sps );
    if( x264_nal_end( h ) )
        return -1;

    /* generate picture parameters */
    x264_nal_start( h, NAL_PPS, NAL_PRIORITY_HIGHEST );
    //输出PPS
    x264_pps_write( &h->out.bs, h->sps, h->pps );
    if( x264_nal_end( h ) )
        return -1;

    /* identify ourselves */
    x264_nal_start( h, NAL_SEI, NAL_PRIORITY_DISPOSABLE );
    //输出SEI（其中包含了配置信息）
    if( x264_sei_version_write( h, &h->out.bs ) )
        return -1;
    if( x264_nal_end( h ) )
        return -1;

    frame_size = x264_encoder_encapsulate_nals( h, 0 );
    if( frame_size < 0 )
        return -1;

    /* now set output*/
    *pi_nal = h->out.i_nal;
    *pp_nal = &h->out.nal[0];
    h->out.i_nal = 0;

    return frame_size;
}
```

从源代码可以看出，x264_encoder_headers()分别调用了x264_sps_write()，x264_pps_write()，x264_sei_version_write()输出了SPS，PPS，和SEI信息。在输出每个NALU之前，需要调用x264_nal_start()，在输出NALU之后，需要调用x264_nal_end()。下文继续分析上述三个函数。

x264_sps_write()

x264_sps_write()用于输出SPS。该函数的定义位于encoder\set.c，如下所示。

```cpp
//输出SPS
void x264_sps_write( bs_t *s, x264_sps_t *sps )
{
    bs_realign( s );
    //型profile，8bit
    bs_write( s, 8, sps->i_profile_idc );
    bs_write1( s, sps->b_constraint_set0 );
    bs_write1( s, sps->b_constraint_set1 );
    bs_write1( s, sps->b_constraint_set2 );
    bs_write1( s, sps->b_constraint_set3 );

    bs_write( s, 4, 0 );    /* reserved */
    //级level，8bit
    bs_write( s, 8, sps->i_level_idc );
    //本SPS的 id号
    bs_write_ue( s, sps->i_id );

    if( sps->i_profile_idc >= PROFILE_HIGH )
    {
        //色度取样格式
        //0代表单色
        //1代表4:2:0
        //2代表4:2:2
        //3代表4:4:4
        bs_write_ue( s, sps->i_chroma_format_idc );
        if( sps->i_chroma_format_idc == CHROMA_444 )
            bs_write1( s, 0 ); // separate_colour_plane_flag
        //高度
```

```
        //亮度
        //颜色位深=bit_depth_luma_minus8+8
        bs_write_ue( s, BIT_DEPTH-8 ); // bit_depth_luma_minus8
        //色度与亮度一样
        bs_write_ue( s, BIT_DEPTH-8 ); // bit_depth_chroma_minus8
        bs_write1( s, sps->b_qpprime_y_zero_transform_bypass );
        bs_write1( s, 0 ); // seq_scaling_matrix_present_flag
    }
    //log2_max_frame_num_minus4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
    //这个句法元素指明了frame_num的所能达到的最大值：
    //MaxFrameNum = 2^( log2_max_frame_num_minus4 + 4 )
    bs_write_ue( s, sps->i_log2_max_frame_num - 4 );
    //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。
    //由于H.264使用了B帧预测，使得图像的解码顺序并不一定等于播放顺序，但它们之间存在一定的映射关系
    //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。
    //H.264 中一共定义了三种poc 的编码方法
    bs_write_ue( s, sps->i_poc_type );
    if( sps->i_poc_type == 0 )
        bs_write_ue( s, sps->i_log2_max_poc_lsb - 4 );
    //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，因此最大值为16。
    bs_write_ue( s, sps->i_num_ref_frames );
    bs_write1( s, sps->b_gaps_in_frame_num_value_allowed );
    //pic_width_in_mbs_minus1加1后为图像宽（以宏块为单位）：
    //            PicWidthInMbs = pic_width_in_mbs_minus1 + 1
    //以像素为单位图像宽度（亮度）：width=PicWidthInMbs*16
    bs_write_ue( s, sps->i_mb_width - 1 );
    //pic_height_in_map_units_minus1加1后指明图像高度（以宏块为单位）
    bs_write_ue( s, (sps->i_mb_height >> !sps->b_frame_mbs_only) - 1);
    bs_write1( s, sps->b_frame_mbs_only );
    if( !sps->b_frame_mbs_only )
        bs_write1( s, sps->b_mb_adaptive_frame_field );
    bs_write1( s, sps->b_direct8x8_inference );

    bs_write1( s, sps->b_crop );
    if( sps->b_crop )
    {
        int h_shift = sps->i_chroma_format_idc == CHROMA_420 || sps->i_chroma_format_idc == CHROMA_422;
        int v_shift = sps->i_chroma_format_idc == CHROMA_420;
        bs_write_ue( s, sps->crop.i_left   >> h_shift );
        bs_write_ue( s, sps->crop.i_right  >> h_shift );
        bs_write_ue( s, sps->crop.i_top    >> v_shift );
        bs_write_ue( s, sps->crop.i_bottom >> v_shift );
    }

    bs_write1( s, sps->b_vui );
    if( sps->b_vui )
    {
        bs_write1( s, sps->vui.b_aspect_ratio_info_present );
        if( sps->vui.b_aspect_ratio_info_present )
        {
            int i;
            static const struct { uint8_t w, h, sar; } sar[] =
            {
                // aspect_ratio_idc = 0 -> unspecified
                {  1,  1, 1 }, { 12, 11, 2 }, { 10, 11, 3 }, { 16, 11, 4 },
                { 40, 33, 5 }, { 24, 11, 6 }, { 20, 11, 7 }, { 32, 11, 8 },
                { 80, 33, 9 }, { 18, 11, 10}, { 15, 11, 11}, { 64, 33, 12},
                {160, 99, 13}, {  4,  3, 14}, {  3,  2, 15}, {  2,  1, 16},
                // aspect_ratio_idc = [17..254] -> reserved
                { 0, 0, 255 }
            };
            for( i = 0; sar[i].sar != 255; i++ )
            {
                if( sar[i].w == sps->vui.i_sar_width &&
                    sar[i].h == sps->vui.i_sar_height )
                    break;
            }
            bs_write( s, 8, sar[i].sar );
            if( sar[i].sar == 255 ) /* aspect_ratio_idc (extended) */
            {
                bs_write( s, 16, sps->vui.i_sar_width );
                bs_write( s, 16, sps->vui.i_sar_height );
            }
        }

        bs_write1( s, sps->vui.b_overscan_info_present );
        if( sps->vui.b_overscan_info_present )
            bs_write1( s, sps->vui.b_overscan_info );

        bs_write1( s, sps->vui.b_signal_type_present );
        if( sps->vui.b_signal_type_present )
        {
            bs_write( s, 3, sps->vui.i_vidformat );
            bs_write1( s, sps->vui.b_fullrange );
            bs_write1( s, sps->vui.b_color_description_present );
            if( sps->vui.b_color_description_present )
            {
                bs_write( s, 8, sps->vui.i_colorprim );
                bs_write( s, 8, sps->vui.i_transfer );
```

```
                bs_write( s, 8, sps->vui.i_colmatrix );
            }
        }

        bs_write1( s, sps->vui.b_chroma_loc_info_present );
        if( sps->vui.b_chroma_loc_info_present )
        {
            bs_write_ue( s, sps->vui.i_chroma_loc_top );
            bs_write_ue( s, sps->vui.i_chroma_loc_bottom );
        }

        bs_write1( s, sps->vui.b_timing_info_present );
        if( sps->vui.b_timing_info_present )
        {
            bs_write32( s, sps->vui.i_num_units_in_tick );
            bs_write32( s, sps->vui.i_time_scale );
            bs_write1( s, sps->vui.b_fixed_frame_rate );
        }

        bs_write1( s, sps->vui.b_nal_hrd_parameters_present );
        if( sps->vui.b_nal_hrd_parameters_present )
        {
            bs_write_ue( s, sps->vui.hrd.i_cpb_cnt - 1 );
            bs_write( s, 4, sps->vui.hrd.i_bit_rate_scale );
            bs_write( s, 4, sps->vui.hrd.i_cpb_size_scale );

            bs_write_ue( s, sps->vui.hrd.i_bit_rate_value - 1 );
            bs_write_ue( s, sps->vui.hrd.i_cpb_size_value - 1 );

            bs_write1( s, sps->vui.hrd.b_cbr_hrd );

            bs_write( s, 5, sps->vui.hrd.i_initial_cpb_removal_delay_length - 1 );
            bs_write( s, 5, sps->vui.hrd.i_cpb_removal_delay_length - 1 );
            bs_write( s, 5, sps->vui.hrd.i_dpb_output_delay_length - 1 );
            bs_write( s, 5, sps->vui.hrd.i_time_offset_length );
        }

        bs_write1( s, sps->vui.b_vcl_hrd_parameters_present );

        if( sps->vui.b_nal_hrd_parameters_present || sps->vui.b_vcl_hrd_parameters_present )
            bs_write1( s, 0 );   /* low_delay_hrd_flag */

        bs_write1( s, sps->vui.b_pic_struct_present );
        bs_write1( s, sps->vui.b_bitstream_restriction );
        if( sps->vui.b_bitstream_restriction )
        {
            bs_write1( s, sps->vui.b_motion_vectors_over_pic_boundaries );
            bs_write_ue( s, sps->vui.i_max_bytes_per_pic_denom );
            bs_write_ue( s, sps->vui.i_max_bits_per_mb_denom );
            bs_write_ue( s, sps->vui.i_log2_max_mv_length_horizontal );
            bs_write_ue( s, sps->vui.i_log2_max_mv_length_vertical );
            bs_write_ue( s, sps->vui.i_num_reorder_frames );
            bs_write_ue( s, sps->vui.i_max_dec_frame_buffering );
        }
    }

    //RBSP拖尾
    //无论比特流当前位置是否字节对齐 ， 都向其中写入一个比特1及若干个（0~7个）比特0 ， 使其字节对齐
    bs_rbsp_trailing( s );
    bs_flush( s );
}
```

可以看出x264_sps_write()将x264_sps_t结构体中的信息输出出来形成了一个NALU。有关SPS相关的知识可以参考《H.264标准》。

x264_pps_write()

x264_pps_write()用于输出PPS。该函数的定义位于encoder\set.c，如下所示。

```cpp
//输出PPS
void x264_pps_write( bs_t *s, x264_sps_t *sps, x264_pps_t *pps )
{
    bs_realign( s );
    //PPS的ID
    bs_write_ue( s, pps->i_id );
    //该PPS引用的SPS的ID
    bs_write_ue( s, pps->i_sps_id );
    //entropy_coding_mode_flag
    //0表示熵编码使用CAVLC，1表示熵编码使用CABAC
    bs_write1( s, pps->b_cabac );
    bs_write1( s, pps->b_pic_order );
    bs_write_ue( s, pps->i_num_slice_groups - 1 );

    bs_write_ue( s, pps->i_num_ref_idx_l0_default_active - 1 );
    bs_write_ue( s, pps->i_num_ref_idx_l1_default_active - 1 );
    //P Slice 是否使用加权预测？
    bs_write1( s, pps->b_weighted_pred );
    //B Slice 是否使用加权预测？
    bs_write( s, 2, pps->b_weighted_bipred );
    //pic_init_qp_minus26加26后用以指明亮度分量的QP的初始值。
    bs_write_se( s, pps->i_pic_init_qp - 26 - QP_BD_OFFSET );
    bs_write_se( s, pps->i_pic_init_qs - 26 - QP_BD_OFFSET );
    bs_write_se( s, pps->i_chroma_qp_index_offset );

    bs_write1( s, pps->b_deblocking_filter_control );
    bs_write1( s, pps->b_constrained_intra_pred );
    bs_write1( s, pps->b_redundant_pic_cnt );

    if( pps->b_transform_8x8_mode || pps->i_cqm_preset != X264_CQM_FLAT )
    {
        bs_write1( s, pps->b_transform_8x8_mode );
        bs_write1( s, (pps->i_cqm_preset != X264_CQM_FLAT) );
        if( pps->i_cqm_preset != X264_CQM_FLAT )
        {
            scaling_list_write( s, pps, CQM_4IY );
            scaling_list_write( s, pps, CQM_4IC );
            bs_write1( s, 0 ); // Cr = Cb
            scaling_list_write( s, pps, CQM_4PY );
            scaling_list_write( s, pps, CQM_4PC );
            bs_write1( s, 0 ); // Cr = Cb
            if( pps->b_transform_8x8_mode )
            {
                if( sps->i_chroma_format_idc == CHROMA_444 )
                {
                    scaling_list_write( s, pps, CQM_8IY+4 );
                    scaling_list_write( s, pps, CQM_8IC+4 );
                    bs_write1( s, 0 ); // Cr = Cb
                    scaling_list_write( s, pps, CQM_8PY+4 );
                    scaling_list_write( s, pps, CQM_8PC+4 );
                    bs_write1( s, 0 ); // Cr = Cb
                }
                else
                {
                    scaling_list_write( s, pps, CQM_8IY+4 );
                    scaling_list_write( s, pps, CQM_8PY+4 );
                }
            }
        }
        bs_write_se( s, pps->i_chroma_qp_index_offset );
    }

    //RBSP拖尾
    //无论比特流当前位置是否字节对齐 ， 都向其中写入一个比特1及若干个（0~7个）比特0 ， 使其字节对齐
    bs_rbsp_trailing( s );
    bs_flush( s );
}
```

可以看出x264_pps_write()将x264_pps_t结构体中的信息输出出来形成了一个NALU。

x264_sei_version_write()

x264_sei_version_write()用于输出SEI。SEI中一般存储了H.264中的一些附加信息，例如下图中红色方框中的文字就是x264存储在SEI中的中的信息。

x264_sei_version_write()的定义位于encoder\set.c，如下所示。

[cpp]

```cpp
//输出SEI（其中包含了配置信息）
int x264_sei_version_write( x264_t *h, bs_t *s )
{
    // random ID number generated according to ISO-11578
    static const uint8_t uuid[16] =
    {
        0xdc, 0x45, 0xe9, 0xbd, 0xe6, 0xd9, 0x48, 0xb7,
        0x96, 0x2c, 0xd8, 0x20, 0xd9, 0x23, 0xee, 0xef
    };
    //把设置信息转换为字符串
    char *opts = x264_param2string( &h->param, 0 );
    char *payload;
    int length;

    if( !opts )
        return -1;
    CHECKED_MALLOC( payload, 200 + strlen( opts ) );

    memcpy( payload, uuid, 16 );
    //配置信息的内容
    //opts字符串内容还是挺多的
    sprintf( payload+16, "x264 - core %d%s - H.264/MPEG-4 AVC codec - "
             "Copy%s 2003-2014 - http://www.videolan.org/x264.html - options: %s",
             X264_BUILD, X264_VERSION, HAVE_GPL?"left":"right", opts );
    length = strlen(payload)+1;
    //输出SEI
    //数据类型为USER_DATA_UNREGISTERED
    x264_sei_write( s, (uint8_t *)payload, length, SEI_USER_DATA_UNREGISTERED );

    x264_free( opts );
    x264_free( payload );
    return 0;
fail:
    x264_free( opts );
    return -1;
}
```

从源代码可以看出，x264_sei_version_write()首先调用了x264_param2string()将当前的配置参数保存到字符串opts[]中，然后调用sprintf()结合opt[]生成完整的SEI信息，最后调用x264_sei_write()输出SEI信息。在这个过程中涉及到一个libx264的API函数x264_param2string()。

x264_param2string()

x264_param2string()用于将当前设置转换为字符串输出出来。该函数的声明如下。

[cpp]

```cpp
/* x264_param2string: return a (malloced) string containing most of
 * the encoding options */
char *x264_param2string( x264_param_t *p, int b_res );
```

x264_param2string()的定义位于common\common.c，如下所示。

[cpp]

```cpp
/****************************************************************************
 * x264_param2string:
 ****************************************************************************/
//把设置信息转换为字符串
char *x264_param2string( x264_param_t *p, int b_res )
{
    int len = 1000;
    char *buf, *s;
    if( p->rc.psz_zones )
        len += strlen(p->rc.psz_zones);
    //1000字节？
    buf = s = x264_malloc( len );
    if( !buf )
        return NULL;

    if( b_res )
    {
        s += sprintf( s, "%dx%d ", p->i_width, p->i_height );
        s += sprintf( s, "fps=%u/%u ", p->i_fps_num, p->i_fps_den );
        s += sprintf( s, "timebase=%u/%u ", p->i_timebase_num, p->i_timebase_den );
        s += sprintf( s, "bitdepth=%d ", BIT_DEPTH );
    }

    if( p->b_opencl )
        s += sprintf( s, "opencl=%d ", p->b_opencl );
    s += sprintf( s, "cabac=%d", p->b_cabac );
    s += sprintf( s, " ref=%d", p->i_frame_reference );
    s += sprintf( s, " deblock=%d:%d:%d", p->b_deblocking_filter,
                  p->i_deblocking_filter_alphac0, p->i_deblocking_filter_beta );
    s += sprintf( s, " analyse=%#x:%#x", p->analyse.intra, p->analyse.inter );
    s += sprintf( s, " me=%s", x264_motion_est_names[ p->analyse.i_me_method ] );
    s += sprintf( s, " subme=%d", p->analyse.i_subpel_refine );
    s += sprintf( s, " psy=%d", p->analyse.b_psy );
```

```c
    if( p->analyse.b_psy )
        s += sprintf( s, " psy_rd=%.2f:%.2f", p->analyse.f_psy_rd, p->analyse.f_psy_trellis );
    s += sprintf( s, " mixed_ref=%d", p->analyse.b_mixed_references );
    s += sprintf( s, " me_range=%d", p->analyse.i_me_range );
    s += sprintf( s, " chroma_me=%d", p->analyse.b_chroma_me );
    s += sprintf( s, " trellis=%d", p->analyse.i_trellis );
    s += sprintf( s, " 8x8dct=%d", p->analyse.b_transform_8x8 );
    s += sprintf( s, " cqm=%d", p->i_cqm_preset );
    s += sprintf( s, " deadzone=%d,%d", p->analyse.i_luma_deadzone[0], p->analyse.i_luma_deadzone[1] );
    s += sprintf( s, " fast_pskip=%d", p->analyse.b_fast_pskip );
    s += sprintf( s, " chroma_qp_offset=%d", p->analyse.i_chroma_qp_offset );
    s += sprintf( s, " threads=%d", p->i_threads );
    s += sprintf( s, " lookahead_threads=%d", p->i_lookahead_threads );
    s += sprintf( s, " sliced_threads=%d", p->b_sliced_threads );
    if( p->i_slice_count )
        s += sprintf( s, " slices=%d", p->i_slice_count );
    if( p->i_slice_count_max )
        s += sprintf( s, " slices_max=%d", p->i_slice_count_max );
    if( p->i_slice_max_size )
        s += sprintf( s, " slice_max_size=%d", p->i_slice_max_size );
    if( p->i_slice_max_mbs )
        s += sprintf( s, " slice_max_mbs=%d", p->i_slice_max_mbs );
    if( p->i_slice_min_mbs )
        s += sprintf( s, " slice_min_mbs=%d", p->i_slice_min_mbs );
    s += sprintf( s, " nr=%d", p->analyse.i_noise_reduction );
    s += sprintf( s, " decimate=%d", p->analyse.b_dct_decimate );
    s += sprintf( s, " interlaced=%s", p->b_interlaced ? p->b_tff ? "tff" : "bff" : p->b_fake_interlaced ? "fake" : "0" );
    s += sprintf( s, " bluray_compat=%d", p->b_bluray_compat );
    if( p->b_stitchable )
        s += sprintf( s, " stitchable=%d", p->b_stitchable );

    s += sprintf( s, " constrained_intra=%d", p->b_constrained_intra );

    s += sprintf( s, " bframes=%d", p->i_bframe );
    if( p->i_bframe )
    {
        s += sprintf( s, " b_pyramid=%d b_adapt=%d b_bias=%d direct=%d weightb=%d open_gop=%d",
                      p->i_bframe_pyramid, p->i_bframe_adaptive, p->i_bframe_bias,
                      p->analyse.i_direct_mv_pred, p->analyse.b_weighted_bipred, p->b_open_gop );
    }
    s += sprintf( s, " weightp=%d", p->analyse.i_weighted_pred > 0 ? p->analyse.i_weighted_pred : 0 );

    if( p->i_keyint_max == X264_KEYINT_MAX_INFINITE )
        s += sprintf( s, " keyint=infinite" );
    else
        s += sprintf( s, " keyint=%d", p->i_keyint_max );
    s += sprintf( s, " keyint_min=%d scenecut=%d intra_refresh=%d",
                  p->i_keyint_min, p->i_scenecut_threshold, p->b_intra_refresh );

    if( p->rc.b_mb_tree || p->rc.i_vbv_buffer_size )
        s += sprintf( s, " rc_lookahead=%d", p->rc.i_lookahead );

    s += sprintf( s, " rc=%s mbtree=%d", p->rc.i_rc_method == X264_RC_ABR ?
                               ( p->rc.b_stat_read ? "2pass" : p->rc.i_vbv_max_bitrate == p->rc.i_bitrate ? "cbr" : "abr" )
                               : p->rc.i_rc_method == X264_RC_CRF ? "crf" : "cqp", p->rc.b_mb_tree );
    if( p->rc.i_rc_method == X264_RC_ABR || p->rc.i_rc_method == X264_RC_CRF )
    {
        if( p->rc.i_rc_method == X264_RC_CRF )
            s += sprintf( s, " crf=%.1f", p->rc.f_rf_constant );
        else
            s += sprintf( s, " bitrate=%d ratetol=%.1f",
                          p->rc.i_bitrate, p->rc.f_rate_tolerance );
        s += sprintf( s, " qcomp=%.2f qpmin=%d qpmax=%d qpstep=%d",
                      p->rc.f_qcompress, p->rc.i_qp_min, p->rc.i_qp_max, p->rc.i_qp_step );
        if( p->rc.b_stat_read )
            s += sprintf( s, " cplxblur=%.1f qblur=%.1f",
                          p->rc.f_complexity_blur, p->rc.f_qblur );
        if( p->rc.i_vbv_buffer_size )
        {
            s += sprintf( s, " vbv_maxrate=%d vbv_bufsize=%d",
                          p->rc.i_vbv_max_bitrate, p->rc.i_vbv_buffer_size );
            if( p->rc.i_rc_method == X264_RC_CRF )
                s += sprintf( s, " crf_max=%.1f", p->rc.f_rf_constant_max );
        }
    }
    else if( p->rc.i_rc_method == X264_RC_CQP )
        s += sprintf( s, " qp=%d", p->rc.i_qp_constant );

    if( p->rc.i_vbv_buffer_size )
        s += sprintf( s, " nal_hrd=%s filler=%d", x264_nal_hrd_names[p->i_nal_hrd], p->rc.b_filler );
    if( p->crop_rect.i_left | p->crop_rect.i_top | p->crop_rect.i_right | p->crop_rect.i_bottom )
        s += sprintf( s, " crop_rect=%u,%u,%u,%u", p->crop_rect.i_left, p->crop_rect.i_top,
                                                   p->crop_rect.i_right, p->crop_rect.i_bottom );
    if( p->i_frame_packing >= 0 )
        s += sprintf( s, " frame-packing=%d", p->i_frame_packing );

    if( !(p->rc.i_rc_method == X264_RC_CQP && p->rc.i_qp_constant == 0) )
    {
        s += sprintf( s, " ip_ratio=%.2f", p->rc.f_ip_factor );
        if( p->i_bframe && !p->rc.b_mb_tree )
            s += sprintf( s, " pb_ratio=%.2f", p->rc.f_pb_factor );
```

```
        s += sprintf( s, " aq=%d", p->rc.i_aq_mode );
        if( p->rc.i_aq_mode )
            s += sprintf( s, ":%.2f", p->rc.f_aq_strength );
        if( p->rc.psz_zones )
            s += sprintf( s, " zones=%s", p->rc.psz_zones );
        else if( p->rc.i_zones )
            s += sprintf( s, " zones" );
    }

    return buf;
}
```

可以看出x264_param2string()几乎遍历了libx264的所有设置选项，使用“s += sprintf()”的形式将它们连接成一个很长的字符串，并最终将该字符串返回。

x264_encoder_close()

x264_encoder_close()是libx264的一个API函数。该函数用于关闭编码器，同时输出一些统计信息。该函数执行的时候输出的统计信息如下图所示。

x264_encoder_close()的声明如下所示。

[cpp]

```cpp
/* x264_encoder_close:
 *      close an encoder handler */
void    x264_encoder_close  ( x264_t * );
```

x264_encoder_close()的定义位于encoder\encoder.c，如下所示。

[cpp]

```cpp
/******************************************************************************
 * x264_encoder_close:
 * 注释和处理：雷霄骅
 * http://blog.csdn.net/leixiaohua1020
 * leixiaohua1020@126.com
 ******************************************************************************/
void    x264_encoder_close  ( x264_t *h )
{
    int64_t i_yuv_size = FRAME_SIZE( h->param.i_width * h->param.i_height );
    int64_t i_mb_count_size[2][7] = {{0}};
    char buf[200];
    int b_print_pcm = h->stat.i_mb_count[SLICE_TYPE_I][I_PCM]
                   || h->stat.i_mb_count[SLICE_TYPE_P][I_PCM]
                   || h->stat.i_mb_count[SLICE_TYPE_B][I_PCM];

    x264_lookahead_delete( h );

#if HAVE_OPENCL
    x264_opencl_lookahead_delete( h );
    x264_opencl_function_t *ocl = h->opencl.ocl;
#endif

    if( h->param.b_sliced_threads )
        x264_threadpool_wait_all( h );
    if( h->param.i_threads > 1 )
        x264_threadpool_delete( h->threadpool );
    if( h->param.i_lookahead_threads > 1 )
        x264_threadpool_delete( h->lookaheadpool );
    if( h->i_thread_frames > 1 )
    {
        for( int i = 0; i < h->i_thread_frames; i++ )
            if( h->thread[i]->b_thread_active )
            {
                assert( h->thread[i]->fenc->i_reference_count == 1 );
                x264_frame_delete( h->thread[i]->fenc );
            }

        x264_t *thread_prev = h->thread[h->i_thread_phase];
        x264_thread_sync_ratecontrol( h, thread_prev, h );
        x264_thread_sync_ratecontrol( thread_prev, thread_prev, h );
        h->i_frame = thread_prev->i_frame + 1 - h->i_thread_frames;
    }
    h->i_frame++;

    /*
     * x264控制台输出示例
     *
     * x264 [info]: using cpu capabilities: MMX2 SSE2Fast SSSE3 SSE4.2 AVX
     * x264 [info]: profile High, level 2.1
     * x264 [info]: frame I:2     Avg QP:20.51  size: 20184  PSNR Mean Y:45.32 U:47.54 V:47.62 Avg:45.94 Global:45.52
     * x264 [info]: frame P:33    Avg QP:23.08  size:  3230  PSNR Mean Y:43.23 U:47.06 V:46.87 Avg:44.15 Global:44.00
     * x264 [info]: frame B:65    Avg QP:27.87  size:   352  PSNR Mean Y:42.76 U:47.21 V:47.05 Avg:43.79 Global:43.65
```

```
     * x264 [info]: frame B:65    Avg QP:27.87  size:   352  PSNR Mean Y:42.76 U:47.21 V:47.05 Avg:43.79 Global:43.65
     * x264 [info]: consecutive B-frames:  3.0% 10.0% 63.0% 24.0%
     * x264 [info]: mb I  I16..4: 15.3% 37.5% 47.3%
     * x264 [info]: mb P  I16..4:  0.6%  0.4%  0.2%  P16..4: 34.6% 21.2% 12.7%  0.0%  0.0%    skip:30.4%
     * x264 [info]: mb B  I16..4:  0.0%  0.0%  0.0%  B16..8: 21.2%  4.1%  0.7%  direct: 0.8%  skip:73.1%  L0:28.7% L1:53.0% BI:18.3%

     * x264 [info]: 8x8 transform intra:37.1% inter:51.0%
     * x264 [info]: coded y,uvDC,uvAC intra: 74.1% 83.3% 58.9% inter: 10.4% 6.6% 0.4%
     * x264 [info]: i16 v,h,dc,p: 21% 25%  7% 48%
     * x264 [info]: i8 v,h,dc,ddl,ddr,vr,hd,vl,hu: 25% 23% 13%  6%  5%  5%  6%  8% 10%
     * x264 [info]: i4 v,h,dc,ddl,ddr,vr,hd,vl,hu: 22% 20%  9%  7%  7%  8%  8%  7% 12%
     * x264 [info]: i8c dc,h,v,p: 43% 20% 27% 10%
     * x264 [info]: Weighted P-Frames: Y:0.0% UV:0.0%
     * x264 [info]: ref P L0: 62.5% 19.7% 13.8%  4.0%
     * x264 [info]: ref B L0: 88.8%  9.4%  1.9%
     * x264 [info]: ref B L1: 92.6%  7.4%
     * x264 [info]: PSNR Mean Y:42.967 U:47.163 V:47.000 Avg:43.950 Global:43.796 kb/s:339.67
     *
     * encoded 100 frames, 178.25 fps, 339.67 kb/s
     *
     */

    /* Slices used and PSNR */
    /* 示例
     * x264 [info]: frame I:2     Avg QP:20.51  size: 20184  PSNR Mean Y:45.32 U:47.54 V:47.62 Avg:45.94 Global:45.52
     * x264 [info]: frame P:33    Avg QP:23.08  size:  3230  PSNR Mean Y:43.23 U:47.06 V:46.87 Avg:44.15 Global:44.00
     * x264 [info]: frame B:65    Avg QP:27.87  size:   352  PSNR Mean Y:42.76 U:47.21 V:47.05 Avg:43.79 Global:43.65
     */
    for( int i = 0; i < 3; i++ )
    {
        static const uint8_t slice_order[] = { SLICE_TYPE_I, SLICE_TYPE_P, SLICE_TYPE_B };
        int i_slice = slice_order[i];

        if( h->stat.i_frame_count[i_slice] > 0 )
        {
            int i_count = h->stat.i_frame_count[i_slice];
            double dur =  h->stat.f_frame_duration[i_slice];
            if( h->param.analyse.b_psnr )
            {
                //输出统计信息-包含PSNR
                //注意PSNR都是通过SSD换算过来的，换算方法就是调用x264_psnr()方法
                x264_log( h, X264_LOG_INFO,
                          "frame %c:%-5d Avg QP:%5.2f  size:%6.0f  PSNR Mean Y:%5.2f U:%5.2f V:%5.2f Avg:%5.2f Global:%5.2f\n",
                          slice_type_to_char[i_slice],
                          i_count,
                          h->stat.f_frame_qp[i_slice] / i_count,
                          (double)h->stat.i_frame_size[i_slice] / i_count,
                          h->stat.f_psnr_mean_y[i_slice] / dur, h->stat.f_psnr_mean_u[i_slice] / dur, h->stat.f_psnr_mean_v[i_slice] / dur,
                          h->stat.f_psnr_average[i_slice] / dur,
                          x264_psnr( h->stat.f_ssd_global[i_slice], dur * i_yuv_size ) );
            }
            else
            {
                //输出统计信息-不包含PSNR
                x264_log( h, X264_LOG_INFO,
                          "frame %c:%-5d Avg QP:%5.2f  size:%6.0f\n",
                          slice_type_to_char[i_slice],
                          i_count,
                          h->stat.f_frame_qp[i_slice] / i_count,
                          (double)h->stat.i_frame_size[i_slice] / i_count );
            }
        }
    }
    /* 示例
     * x264 [info]: consecutive B-frames:  3.0% 10.0% 63.0% 24.0%
     *
     */
    if( h->param.i_bframe && h->stat.i_frame_count[SLICE_TYPE_B] )
    {
        //B帧相关信息
        char *p = buf;
        int den = 0;
        // weight by number of frames (including the I/P-frames) that are in a sequence of N B-frames
        for( int i = 0; i <= h->param.i_bframe; i++ )
            den += (i+1) * h->stat.i_consecutive_bframes[i];
        for( int i = 0; i <= h->param.i_bframe; i++ )
            p += sprintf( p, " %4.1f%%", 100. * (i+1) * h->stat.i_consecutive_bframes[i] / den );
        x264_log( h, X264_LOG_INFO, "consecutive B-frames:%s\n", buf );
    }

    for( int i_type = 0; i_type < 2; i_type++ )
        for( int i = 0; i < X264_PARTTYPE_MAX; i++ )
        {
            if( i == D_DIRECT_8x8 ) continue; /* direct is counted as its own type */
            i_mb_count_size[i_type][x264_mb_partition_pixel_table[i]] += h->stat.i_mb_partition[i_type][i];
        }

    /* MB types used */
    /* 示例
     * x264 [info]: mb I  I16..4: 15.3% 37.5% 47.3%
     * x264 [info]: mb P  I16..4:  0.6%  0.4%  0.2%  P16..4: 34.6% 21.2% 12.7%  0.0%  0.0%    skip:30.4%
```

```
     * x264 [info]: mb B  I16..4:  0.0%  0.0%  0.0%  B16..8: 21.2%  4.1%  0.7%  direct: 0.8%  skip:73.1%  L0:28.7% L1:53.0% BI:18.3%

     */
    if( h->stat.i_frame_count[SLICE_TYPE_I] > 0 )
    {
        int64_t *i_mb_count = h->stat.i_mb_count[SLICE_TYPE_I];
        double i_count = h->stat.i_frame_count[SLICE_TYPE_I] * h->mb.i_mb_count / 100.0;
        //Intra宏块信息-存于buf
        //从左到右3个信息，依次为I16x16,I8x8,I4x4
        x264_print_intra( i_mb_count, i_count, b_print_pcm, buf );
        x264_log( h, X264_LOG_INFO, "mb I  %s\n", buf );
    }
    if( h->stat.i_frame_count[SLICE_TYPE_P] > 0 )
    {
        int64_t *i_mb_count = h->stat.i_mb_count[SLICE_TYPE_P];
        double i_count = h->stat.i_frame_count[SLICE_TYPE_P] * h->mb.i_mb_count / 100.0;
        int64_t *i_mb_size = i_mb_count_size[SLICE_TYPE_P];
        //Intra宏块信息-存于buf
        x264_print_intra( i_mb_count, i_count, b_print_pcm, buf );
        //Intra宏块信息-放在最前面
        //后面添加P宏块信息
        //从左到右6个信息，依次为P16x16, P16x8+P8x16, P8x8, P8x4+P4x8, P4x4, PSKIP
        x264_log( h, X264_LOG_INFO,
                  "mb P  %s  P16..4: %4.1f%% %4.1f%% %4.1f%% %4.1f%% %4.1f%%    skip:%4.1f%%\n",
                  buf,
                  i_mb_size[PIXEL_16x16] / (i_count*4),
                  (i_mb_size[PIXEL_16x8] + i_mb_size[PIXEL_8x16]) / (i_count*4),
                  i_mb_size[PIXEL_8x8] / (i_count*4),
                  (i_mb_size[PIXEL_8x4] + i_mb_size[PIXEL_4x8]) / (i_count*4),
                  i_mb_size[PIXEL_4x4] / (i_count*4),
                  i_mb_count[P_SKIP] / i_count );
    }
    if( h->stat.i_frame_count[SLICE_TYPE_B] > 0 )
    {
        int64_t *i_mb_count = h->stat.i_mb_count[SLICE_TYPE_B];
        double i_count = h->stat.i_frame_count[SLICE_TYPE_B] * h->mb.i_mb_count / 100.0;
        double i_mb_list_count;
        int64_t *i_mb_size = i_mb_count_size[SLICE_TYPE_B];
        int64_t list_count[3] = {0}; /* 0 == L0, 1 == L1, 2 == BI */
        //Intra宏块信息
        x264_print_intra( i_mb_count, i_count, b_print_pcm, buf );
        for( int i = 0; i < X264_PARTTYPE_MAX; i++ )
            for( int j = 0; j < 2; j++ )
            {
                int l0 = x264_mb_type_list_table[i][0][j];
                int l1 = x264_mb_type_list_table[i][1][j];
                if( l0 || l1 )
                    list_count[l1+l0*l1] += h->stat.i_mb_count[SLICE_TYPE_B][i] * 2;
            }
        list_count[0] += h->stat.i_mb_partition[SLICE_TYPE_B][D_L0_8x8];
        list_count[1] += h->stat.i_mb_partition[SLICE_TYPE_B][D_L1_8x8];
        list_count[2] += h->stat.i_mb_partition[SLICE_TYPE_B][D_BI_8x8];
        i_mb_count[B_DIRECT] += (h->stat.i_mb_partition[SLICE_TYPE_B][D_DIRECT_8x8]+2)/4;
        i_mb_list_count = (list_count[0] + list_count[1] + list_count[2]) / 100.0;
        //Intra宏块信息-放在最前面
        //后面添加B宏块信息
        //从左到右5个信息，依次为B16x16, B16x8+B8x16, B8x8, BDIRECT, BSKIP
        //
        //SKIP和DIRECT区别
        //P_SKIP的CBP为0,无像素残差，无运动矢量残
        //B_SKIP宏块的模式为B_DIRECT且CBP为0,无像素残差，无运动矢量残
        //B_DIRECT的CBP不为0,有像素残差，无运动矢量残
        sprintf( buf + strlen(buf), "  B16..8: %4.1f%% %4.1f%% %4.1f%%  direct:%4.1f%%  skip:%4.1f%%",
                 i_mb_size[PIXEL_16x16] / (i_count*4),
                 (i_mb_size[PIXEL_16x8] + i_mb_size[PIXEL_8x16]) / (i_count*4),
                 i_mb_size[PIXEL_8x8] / (i_count*4),
                 i_mb_count[B_DIRECT] / i_count,
                 i_mb_count[B_SKIP]   / i_count );
        if( i_mb_list_count != 0 )
            sprintf( buf + strlen(buf), "  L0:%4.1f%% L1:%4.1f%% BI:%4.1f%%",
                     list_count[0] / i_mb_list_count,
                     list_count[1] / i_mb_list_count,
                     list_count[2] / i_mb_list_count );
        x264_log( h, X264_LOG_INFO, "mb B  %s\n", buf );
    }
    //码率控制信息
    /* 示例
     * x264 [info]: final ratefactor: 20.01
     */
    x264_ratecontrol_summary( h );

    if( h->stat.i_frame_count[SLICE_TYPE_I] + h->stat.i_frame_count[SLICE_TYPE_P] + h->stat.i_frame_count[SLICE_TYPE_B] > 0 )
    {
#define SUM3(p) (p[SLICE_TYPE_I] + p[SLICE_TYPE_P] + p[SLICE_TYPE_B])
#define SUM3b(p,o) (p[SLICE_TYPE_I][o] + p[SLICE_TYPE_P][o] + p[SLICE_TYPE_B][o])
        int64_t i_i8x8 = SUM3b( h->stat.i_mb_count, I_8x8 );
        int64_t i_intra = i_i8x8 + SUM3b( h->stat.i_mb_count, I_4x4 )
                                 + SUM3b( h->stat.i_mb_count, I_16x16 );
        int64_t i_all_intra = i_intra + SUM3b( h->stat.i_mb_count, I_PCM);
        int64_t i_skip = SUM3b( h->stat.i_mb_count, P_SKIP )
                       + SUM3b( h->stat.i_mb_count, B_SKIP );
```

```c
        const int i_count = h->stat.i_frame_count[SLICE_TYPE_I] +
                            h->stat.i_frame_count[SLICE_TYPE_P] +
                            h->stat.i_frame_count[SLICE_TYPE_B];
        int64_t i_mb_count = (int64_t)i_count * h->mb.i_mb_count;
        int64_t i_inter = i_mb_count - i_skip - i_intra;
        const double duration = h->stat.f_frame_duration[SLICE_TYPE_I] +
                                h->stat.f_frame_duration[SLICE_TYPE_P] +
                                h->stat.f_frame_duration[SLICE_TYPE_B];
        float f_bitrate = SUM3(h->stat.i_frame_size) / duration / 125;
        //隔行
        if( PARAM_INTERLACED )
        {
            char *fieldstats = buf;
            fieldstats[0] = 0;
            if( i_inter )
                fieldstats += sprintf( fieldstats, " inter:%.1f%%", h->stat.i_mb_field[1] * 100.0 / i_inter );
            if( i_skip )
                fieldstats += sprintf( fieldstats, " skip:%.1f%%", h->stat.i_mb_field[2] * 100.0 / i_skip );
            x264_log( h, X264_LOG_INFO, "field mbs: intra: %.1f%%%s\n",
                      h->stat.i_mb_field[0] * 100.0 / i_intra, buf );
        }
        //8x8DCT信息
        if( h->pps->b_transform_8x8_mode )
        {
            buf[0] = 0;
            if( h->stat.i_mb_count_8x8dct[0] )
                sprintf( buf, " inter:%.1f%%", 100. * h->stat.i_mb_count_8x8dct[1] / h->stat.i_mb_count_8x8dct[0] );
            x264_log( h, X264_LOG_INFO, "8x8 transform intra:%.1f%%%s\n", 100. * i_i8x8 / i_intra, buf );
        }

        if( (h->param.analyse.i_direct_mv_pred == X264_DIRECT_PRED_AUTO ||
            (h->stat.i_direct_frames[0] && h->stat.i_direct_frames[1]))
            && h->stat.i_frame_count[SLICE_TYPE_B] )
        {
            x264_log( h, X264_LOG_INFO, "direct mvs  spatial:%.1f%% temporal:%.1f%%\n",
                      h->stat.i_direct_frames[1] * 100. / h->stat.i_frame_count[SLICE_TYPE_B],
                      h->stat.i_direct_frames[0] * 100. / h->stat.i_frame_count[SLICE_TYPE_B] );
        }

        buf[0] = 0;
        int csize = CHROMA444 ? 4 : 1;
        if( i_mb_count != i_all_intra )
            sprintf( buf, " inter: %.1f%% %.1f%% %.1f%%",
                     h->stat.i_mb_cbp[1] * 100.0 / ((i_mb_count - i_all_intra)*4),
                     h->stat.i_mb_cbp[3] * 100.0 / ((i_mb_count - i_all_intra)*csize),
                     h->stat.i_mb_cbp[5] * 100.0 / ((i_mb_count - i_all_intra)*csize) );
        /*
         * 示例
         * x264 [info]: coded y,uvDC,uvAC intra: 74.1% 83.3% 58.9% inter: 10.4% 6.6% 0.4%
         */
        x264_log( h, X264_LOG_INFO, "coded y,%s,%s intra: %.1f%% %.1f%% %.1f%%%s\n",
                  CHROMA444?"u":"uvDC", CHROMA444?"v":"uvAC",
                  h->stat.i_mb_cbp[0] * 100.0 / (i_all_intra*4),
                  h->stat.i_mb_cbp[2] * 100.0 / (i_all_intra*csize),
                  h->stat.i_mb_cbp[4] * 100.0 / (i_all_intra*csize), buf );

        /*
         * 帧内预测信息
         * 从上到下分别为I16x16,I8x8,I4x4
         * 从左到右顺序为Vertical, Horizontal, DC, Plane ....
         *
         * 示例
         *
         * x264 [info]: i16 v,h,dc,p: 21% 25%  7% 48%
         * x264 [info]: i8 v,h,dc,ddl,ddr,vr,hd,vl,hu: 25% 23% 13%  6%  5%  5%  6%  8% 10%
         * x264 [info]: i4 v,h,dc,ddl,ddr,vr,hd,vl,hu: 22% 20%  9%  7%  7%  8%  8%  7% 12%
         * x264 [info]: i8c dc,h,v,p: 43% 20% 27% 10%
         *
         */
        int64_t fixed_pred_modes[4][9] = {{0}};
        int64_t sum_pred_modes[4] = {0};
        for( int i = 0; i <= I_PRED_16x16_DC_128; i++ )
        {
            fixed_pred_modes[0][x264_mb_pred_mode16x16_fix[i]] += h->stat.i_mb_pred_mode[0][i];
            sum_pred_modes[0] += h->stat.i_mb_pred_mode[0][i];
        }
        if( sum_pred_modes[0] )
            x264_log( h, X264_LOG_INFO, "i16 v,h,dc,p: %2.0f%% %2.0f%% %2.0f%% %2.0f%%\n",
                      fixed_pred_modes[0][0] * 100.0 / sum_pred_modes[0],
                      fixed_pred_modes[0][1] * 100.0 / sum_pred_modes[0],
                      fixed_pred_modes[0][2] * 100.0 / sum_pred_modes[0],
                      fixed_pred_modes[0][3] * 100.0 / sum_pred_modes[0] );

        for( int i = 1; i <= 2; i++ )
        {
            for( int j = 0; j <= I_PRED_8x8_DC_128; j++ )
            {
                fixed_pred_modes[i][x264_mb_pred_mode4x4_fix(j)] += h->stat.i_mb_pred_mode[i][j];
                sum_pred_modes[i] += h->stat.i_mb_pred_mode[i][j];
            }
            if( sum_pred_modes[i] )
```

```
                x264_log( h, X264_LOG_INFO, "i%d v,h,dc,ddl,ddr,vr,hd,vl,hu: %2.0f%% %2.0f%% %2.0f%% %2.0f%% %2.0f%% %2.0f%% %2.0f%% %2.0f%% %2.0f%%\n", (3-i)*4,
                          fixed_pred_modes[i][0] * 100.0 / sum_pred_modes[i],
                          fixed_pred_modes[i][1] * 100.0 / sum_pred_modes[i],
                          fixed_pred_modes[i][2] * 100.0 / sum_pred_modes[i],
                          fixed_pred_modes[i][3] * 100.0 / sum_pred_modes[i],
                          fixed_pred_modes[i][4] * 100.0 / sum_pred_modes[i],
                          fixed_pred_modes[i][5] * 100.0 / sum_pred_modes[i],
                          fixed_pred_modes[i][6] * 100.0 / sum_pred_modes[i],
                          fixed_pred_modes[i][7] * 100.0 / sum_pred_modes[i],
                          fixed_pred_modes[i][8] * 100.0 / sum_pred_modes[i] );
        }
        for( int i = 0; i <= I_PRED_CHROMA_DC_128; i++ )
        {
            fixed_pred_modes[3][x264_mb_chroma_pred_mode_fix[i]] += h->stat.i_mb_pred_mode[3][i];
            sum_pred_modes[3] += h->stat.i_mb_pred_mode[3][i];
        }
        if( sum_pred_modes[3] && !CHROMA444 )
            x264_log( h, X264_LOG_INFO, "i8c dc,h,v,p: %2.0f%% %2.0f%% %2.0f%% %2.0f%%\n",
                      fixed_pred_modes[3][0] * 100.0 / sum_pred_modes[3],
                      fixed_pred_modes[3][1] * 100.0 / sum_pred_modes[3],
                      fixed_pred_modes[3][2] * 100.0 / sum_pred_modes[3],
                      fixed_pred_modes[3][3] * 100.0 / sum_pred_modes[3] );

        if( h->param.analyse.i_weighted_pred >= X264_WEIGHTP_SIMPLE && h->stat.i_frame_count[SLICE_TYPE_P] > 0 )
            x264_log( h, X264_LOG_INFO, "Weighted P-Frames: Y:%.1f%% UV:%.1f%%\n",
                      h->stat.i_wpred[0] * 100.0 / h->stat.i_frame_count[SLICE_TYPE_P],
                      h->stat.i_wpred[1] * 100.0 / h->stat.i_frame_count[SLICE_TYPE_P] );

        /*
         * 参考帧信息
         * 从左到右依次为不同序号的参考帧
         *
         * 示例
         *
         * x264 [info]: ref P L0: 62.5% 19.7% 13.8%  4.0%
         * x264 [info]: ref B L0: 88.8%  9.4%  1.9%
         * x264 [info]: ref B L1: 92.6%  7.4%
         *
         */
        for( int i_list = 0; i_list < 2; i_list++ )
            for( int i_slice = 0; i_slice < 2; i_slice++ )
            {
                char *p = buf;
                int64_t i_den = 0;
                int i_max = 0;
                for( int i = 0; i < X264_REF_MAX*2; i++ )
                    if( h->stat.i_mb_count_ref[i_slice][i_list][i] )
                    {
                        i_den += h->stat.i_mb_count_ref[i_slice][i_list][i];
                        i_max = i;
                    }
                if( i_max == 0 )
                    continue;
                for( int i = 0; i <= i_max; i++ )
                    p += sprintf( p, " %4.1f%%", 100. * h->stat.i_mb_count_ref[i_slice][i_list][i] / i_den );
                x264_log( h, X264_LOG_INFO, "ref %c L%d:%s\n", "PB"[i_slice], i_list, buf );
            }

        if( h->param.analyse.b_ssim )
        {
            float ssim = SUM3( h->stat.f_ssim_mean_y ) / duration;
            x264_log( h, X264_LOG_INFO, "SSIM Mean Y:%.7f (%6.3fdb)\n", ssim, x264_ssim( ssim ) );
        }
        /*
         * 示例
         *
         * x264 [info]: PSNR Mean Y:42.967 U:47.163 V:47.000 Avg:43.950 Global:43.796 kb/s:339.67
         *
         */
        if( h->param.analyse.b_psnr )
        {
            x264_log( h, X264_LOG_INFO,
                      "PSNR Mean Y:%6.3f U:%6.3f V:%6.3f Avg:%6.3f Global:%6.3f kb/s:%.2f\n",
                      SUM3( h->stat.f_psnr_mean_y ) / duration,
                      SUM3( h->stat.f_psnr_mean_u ) / duration,
                      SUM3( h->stat.f_psnr_mean_v ) / duration,
                      SUM3( h->stat.f_psnr_average ) / duration,
                      x264_psnr( SUM3( h->stat.f_ssd_global ), duration * i_yuv_size ),
                      f_bitrate );
        }
        else
            x264_log( h, X264_LOG_INFO, "kb/s:%.2f\n", f_bitrate );
    }

    //各种释放

    /* rc */
    x264_ratecontrol_delete( h );

    /* param */
```

```
    if( h->param.rc.psz_stat_out )
        free( h->param.rc.psz_stat_out );
    if( h->param.rc.psz_stat_in )
        free( h->param.rc.psz_stat_in );

    x264_cqm_delete( h );
    x264_free( h->nal_buffer );
    x264_free( h->reconfig_h );
    x264_analyse_free_costs( h );

    if( h->i_thread_frames > 1 )
        h = h->thread[h->i_thread_phase];

    /* frames */
    x264_frame_delete_list( h->frames.unused[0] );
    x264_frame_delete_list( h->frames.unused[1] );
    x264_frame_delete_list( h->frames.current );
    x264_frame_delete_list( h->frames.blank_unused );

    h = h->thread[0];

    for( int i = 0; i < h->i_thread_frames; i++ )
        if( h->thread[i]->b_thread_active )
            for( int j = 0; j < h->thread[i]->i_ref[0]; j++ )
                if( h->thread[i]->fref[0][j] && h->thread[i]->fref[0][j]->b_duplicate )
                    x264_frame_delete( h->thread[i]->fref[0][j] );

    if( h->param.i_lookahead_threads > 1 )
        for( int i = 0; i < h->param.i_lookahead_threads; i++ )
            x264_free( h->lookahead_thread[i] );

    for( int i = h->param.i_threads - 1; i >= 0; i-- )
    {
        x264_frame_t **frame;

        if( !h->param.b_sliced_threads || i == 0 )
        {
            for( frame = h->thread[i]->frames.reference; *frame; frame++ )
            {
                assert( (*frame)->i_reference_count > 0 );
                (*frame)->i_reference_count--;
                if( (*frame)->i_reference_count == 0 )
                    x264_frame_delete( *frame );
            }
            frame = &h->thread[i]->fdec;
            if( *frame )
            {
                assert( (*frame)->i_reference_count > 0 );
                (*frame)->i_reference_count--;
                if( (*frame)->i_reference_count == 0 )
                    x264_frame_delete( *frame );
            }
            x264_macroblock_cache_free( h->thread[i] );
        }
        x264_macroblock_thread_free( h->thread[i], 0 );
        x264_free( h->thread[i]->out.p_bitstream );
        x264_free( h->thread[i]->out.nal );
        x264_pthread_mutex_destroy( &h->thread[i]->mutex );
        x264_pthread_cond_destroy( &h->thread[i]->cv );
        x264_free( h->thread[i] );
    }
#if HAVE_OPENCL
    x264_opencl_close_library( ocl );
#endif
}
```

从源代码可以看出，x264_encoder_close()主要用于输出编码的统计信息。源代码中已经做了比较充分的注释，就不再详细叙述了。其中输出日志的时候用到了libx264中输出日志的API函数libx264()，下面记录一下。

x264_log()

x264_log()用于输出日志。该函数的定义位于common\common.c，如下所示。

```cpp
/*******************************************************************************
 * x264_log:
 *******************************************************************************/
//日志输出函数
void x264_log( x264_t *h, int i_level, const char *psz_fmt, ... )
{
    if( !h || i_level <= h->param.i_log_level )
    {
        va_list arg;
        va_start( arg, psz_fmt );
        if( !h )
            x264_log_default( NULL, i_level, psz_fmt, arg );//默认日志输出函数
        else
            h->param.pf_log( h->param.p_log_private, i_level, psz_fmt, arg );
        va_end( arg );
    }
}
```

可以看出x264_log()再开始的时候做了一个判断：只有该条日志级别i_level小于当前系统的日志级别param.i_log_level的时候，才会输出日志。libx264中定义了下面几种日志级别，数值越小，代表日志越紧急。

```cpp
/* Log level */
#define X264_LOG_NONE          (-1)
#define X264_LOG_ERROR          0
#define X264_LOG_WARNING        1
#define X264_LOG_INFO           2
#define X264_LOG_DEBUG          3
```

接下来x264_log()会根据输入的结构体x264_t是否为空来决定是调用x264_log_default()或者是x264_t中的param.pf_log()函数。假如都使用默认配置的话，param.pf_log()在x264_param_default()函数中也会被设置为指向x264_log_default()。因此可以继续看一下x264_log_default()函数。

x264_log_default()

x264_log_default()是libx264默认的日志输出函数。该函数的定义如下所示。

```cpp
//默认日志输出函数
static void x264_log_default( void *p_unused, int i_level, const char *psz_fmt, va_list arg )
{
    char *psz_prefix;
    //日志级别
    switch( i_level )
    {
        case X264_LOG_ERROR:
            psz_prefix = "error";
            break;
        case X264_LOG_WARNING:
            psz_prefix = "warning";
            break;
        case X264_LOG_INFO:
            psz_prefix = "info";
            break;
        case X264_LOG_DEBUG:
            psz_prefix = "debug";
            break;
        default:
            psz_prefix = "unknown";
            break;
    }
    //日志级别两边加上“[]”
    //输出到stderr
    fprintf( stderr, "x264 [%s]: ", psz_prefix );
    x264_vfprintf( stderr, psz_fmt, arg );
}
```

从源代码可以看出，x264_log_default()会在日志信息前面加上形如“x264 [日志级别]”的信息，然后将处理后的日志输出到stderr。

至此，对x264中x264_encoder_open()，x264_encoder_headers()，和x264_encoder_close()这三个函数的分析就完成了。下一篇文章继续记录x264编码器主干部分的x264_encoder_encode()函数。

雷霄骅
leixiaohua1020@126.com
http://blog.csdn.net/leixiaohua1020

文章标签： x264 libx264 视频 源代码 初始化

个人分类： x264

所属专栏： 开源多媒体项目源代码分析