# main() // 主函数

//给H264AVCEncoderTest\* pcH264AVCEncoderTest分配空间，后面函数用它调用

## H264AVCEncoderTest::create()

//初始化pcH264AVCEncoderTest的成员，主要是编码器的参数配置

## pcH264AVCEncoderTest->init()

//给EncoderCodingParameter\* m\_pcEncoderCodingParameter分配空间

//它是H264AVCEncoderTest \*pcH264AVCEncoderTest的成员

EncoderCodingParameter::create( m\_pcEncoderCodingParameter ) //编码参数

//解析输入命令行，并把相应参数写入EncoderCodingParameter的基类CodingParameter的成员中

m\_pcEncoderCodingParameter->init()

//读取配置文件

xReadFromFile( cFilename, rcBitstreamFile )

|  |  |
| --- | --- |
| fopen( rcFilename.c\_str(), "r"); encoder.cfg  //先创建每个参数  m\_pEncoderLines[uiParLnCount++] = new EncoderConfigLineXXX  //从配置文件里读一行，重复直到文件末尾  xReadLine( f, acTags )  //查找读到的参数对应的m\_pEncoderLines，更新值  fclose( f );  //从上面的配置文件读到的参数中得到有几层，和每层配置文件路径  //对每一层  setDependencyId(ui); //设置Did  //读取每一层的配置文件 读到如下类中  //EncoderCodingParameter::LayerParameters m\_acLayerParameters[n]  xReadLayerFromFile( acLayerConfigName[ui], getLayerParameters(ui) )   |  | | --- | | //打开配置文件 layerX.cfg  //分配空间并初始化参数m\_pLayerLines[] ,也是CodingParameter成员  //读取配置文件每行，并更新对应m\_pLayerLines的值  //初始化ScalingList  xReadScalMat()  //设置输入输出文件路径  rcLayer.setInputFilename ( (Char\*)cInputFilename.c\_str() );  rcLayer.setOutputFilename ( (Char\*)cOutputFilename.c\_str() );  //初始化每个LayerParameters::ResizeParameters m\_cResizeParameters  //读取SliceGroup的配置文件  xReadSliceGroupCfg( rcLayer)  //读取ROI配置文件  xReadROICfg( rcLayer)  //关闭配置文件 |   //设置CGS/MGS的Dependency\_Id和Quality\_id  //由BaseLayer更新当前Layer的ResizeParameter----------ESS |

//对每一层 rcLayer = m\_pcEncoderCodingParameter->getLayerParameters( uiLayer );

//创建并打开重建输出文件

WriteYuvToFile::create( m\_apcWriteYuv[uiLayer] )

m\_apcWriteYuv[uiLayer]->init( rcLayer.getOutputFilename() )

//创建并打开读入的视频文件

ReadYuvFile ::create( m\_apcReadYuv [uiLayer] )

m\_apcReadYuv[uiLayer]->init()

//创建并打开码流文件

WriteBitstreamToFile::create ( m\_pcWriteBitstreamToFile )

m\_pcWriteBitstreamToFile->init ( m\_cEncoderIoParameter.cBitstreamFilename )

|  |
| --- |
| 注意：  以上三个文件都是H264AVCEncoderTest类的成员  WriteBitstreamToFile\* m\_pcWriteBitstreamToFile;  WriteYuvToFile\* m\_apcWriteYuv [MAX\_LAYERS];  ReadYuvFile\* m\_apcReadYuv [MAX\_LAYERS];  也用到了辅助IO的成员  EncoderIoParameter m\_cEncoderIoParameter; |

//给编码器分配空间：它是H264AVCEncoderTest \*pcH264AVCEncoderTest的成员

h264::CreaterH264AVCEncoder::create( m\_pcH264AVCEncoder )

|  |
| --- |
| ParameterSetMng ::create( m\_pcParameterSetMng )  BitWriteBuffer ::create( m\_pcBitWriteBuffer )  BitCounter ::create( m\_pcBitCounter )  NalUnitEncoder ::create( m\_pcNalUnitEncoder)  SliceEncoder ::create( m\_pcSliceEncoder )  UvlcWriter ::create( m\_pcUvlcWriter )  UvlcWriter ::create( m\_pcUvlcTester, false )  CabacWriter ::create( m\_pcCabacWriter )  MbCoder ::create( m\_pcMbCoder )  MbEncoder ::create( m\_pcMbEncoder )  LoopFilter ::create( m\_pcLoopFilter )  IntraPredictionSearch ::create( m\_pcIntraPrediction )  MotionEstimationQuarterPel ::create( m\_pcMotionEstimation )  H264AVCEncoder ::create( m\_pcH264AVCEncoder )  ControlMngH264AVCEncoder ::create( m\_pcControlMng )  ReconstructionBypass ::create( m\_pcReconstructionBypass )  QuarterPelFilter ::create( m\_pcQuarterPelFilter )  Transform ::create( m\_pcTransform )  SampleWeighting ::create( m\_pcSampleWeighting )  XDistortion ::create( m\_pcXDistortion )  PicEncoder ::create( m\_pcPicEncoder )  //每一层的编码器  for( UInt uiLayer = 0; uiLayer < MAX\_LAYERS; uiLayer++ )  {  LayerEncoder ::create( m\_apcLayerEncoder [uiLayer] )  PocCalculator ::create( m\_apcPocCalculator [uiLayer] )  YuvBufferCtrl ::create( m\_apcYuvFullPelBufferCtrl[uiLayer] )  YuvBufferCtrl ::create( m\_apcYuvHalfPelBufferCtrl[uiLayer] )  } |

//设置NAL UNIT起始码m\_aucStartCodeBuffer[],绑定到H264AVCEncoderTest::BinData m\_cBinDataStartCode;

//进行编码

## pcH264AVCEncoderTest->go ()

//初始化编码器参数 h264::CreaterH264AVCEncoder\* m\_pcH264AVCEncoder; H264AVCEncoderTest成员

m\_pcH264AVCEncoder->init( m\_pcEncoderCodingParameter )

|  |
| --- |
| /\*  几个参数关系   1. H264AVCEncoderTest的成员CreaterH264AVCEncoder\* m\_pcH264AVCEncoder调用init   2. 初始化H264AVCEncoderTest的成员EncoderCodingParameter \* m\_pcEncoderCodingParameter  3. 而CreaterH264AVCEncoder的成员CodingParameter\* m\_pcCodingParameter  被赋值传入的参数m\_pcEncoderCodingParameter  \*/  RateDistortion::create( m\_pcRateDistortion ) //给率失真计算类分配空间 RateDistortion  m\_pcBitWriteBuffer ->init() //初始化写Packet/比特流的类 BitWriteBuffer  m\_pcBitCounter ->init() //初始化计算比特数类 BitCounter  m\_pcXDistortion ->init() //初始化计算失真所需的参数 （块行数/列数/Dist计算函数指针）XDistortion  m\_pcSampleWeighting ->init() //初始化加权函数 SampleWeighting  //初始化写NalUnit的类 NalUnitEncoder  m\_pcNalUnitEncoder->init( m\_pcBitWriteBuffer, m\_pcUvlcWriter, m\_pcUvlcTester )  m\_pcUvlcWriter ->init( m\_pcBitWriteBuffer ) //关联比特流缓存和写Uvlc的类 UvlcWriter  m\_pcUvlcTester ->init( m\_pcBitCounter ) //关联比特计数和写Uvlc的类 UvlcWriter  m\_pcCabacWriter ->init( m\_pcBitWriteBuffer ) //初始化Cabac的缓存 CabacWriter CabacEncoder  m\_pcParameterSetMng ->init() //空函数  m\_pcSliceEncoder->init() //关联MbEncoder、MbCoder、ControlMngH264AVCEncoder、CodingParameter、  PocCalculator和Transform 给Slice编码器 SliceEncoder  m\_pcReconstructionBypass ->init() //空函数  m\_pcLoopFilter->init( m\_pcControlMng, m\_pcReconstructionBypass, true ) //关联 ControlMngH264AVCEncoder和  ReconstructionBypass给去块滤波器 LoopFilter  m\_pcQuarterPelFilter->init() //分配1/4像素插值滤波器 QuarterPelFilter  m\_pcMbEncoder->init() //关联Transform、IntraPredictionSearch、MotionEstimation、CodingParameter、  RateDistortion和XDistortion给宏块编码器 MbEncoder  m\_pcMotionEstimation ->init() MotionEstimation  MotionCompensation::init() //初始化MC相关—1/4插值滤波、变换、加权  //设置计算失真、MV搜索参数  MotionEstimationCost::xInit() //设置RD相关  //设置跟搜索范围搜索深度有关参数  // spiral search参数  m\_pcControlMng->init() //分配上面所有的编码过程类到ControlMngH264AVCEncoder类中，集中管理  //包括每层的LayerEncoder、PocCalculator和YuvBufferCtrl  m\_pcPicEncoder->init() //初始化AVC编码器  m\_pcH264AVCEncoder ->init() //初始化SVC编码器 分配LayerEncoder、ParameterSetMng、PocCalculator、  // NalUnitEncoder、ControlMngIf和CodingParameter  //初始化LayerEncoder\* m\_apcLayerEncoder[MAX\_LAYERS]  和AccessUnitDataList m\_cAccessUnitDataList; H264AVCEncoder  **//对每层**  m\_apcLayerEncoder[uiLayer]->init() //初始化每层编码所需的参数  //把配置文件读入的参数LayerParameter复制过来  xInitCodingOrder() // 初始化编码顺序和分层的对应关系  //初始化如ESS、LARDO等参数  //计算每层采用LARDO的情况 |

//Rate Control相关初始化，与GOP有关

//写SPS和PPS

|  |
| --- |
| 参数：  UChar aucParameterSetBuffer[1000] 作为缓存  BinData cBinData 绑定上面的缓存  ExtBinDataAccessor cExtBinDataAccessor 操纵上面的cBinData |
| m\_pcH264AVCEncoder->writeParameterSets( &cExtBinDataAccessor, pcAVCSPS, bMoreSets ) CreaterH264AVCEncoder  m\_pcH264AVCEncoder->writeParameterSets( pcExtBinDataAccessor, rpcAVCSPS, rbMoreSets ) H264AVCEncoder  //如果是第一次调用  xInitParameterSets()  //对每一层  //设置基本参数  SequenceParameterSet::create( pcSPS ) // 分配SPS空间  PictureParameterSet ::create( pcPPS ) // 分配PPS空间  //设置SPS和PPS的ID和NalUnitType     |  | | --- | | ParameterSetMng\* m\_pcParameterSetMng有成员  StatBuf<SequenceParameterSet\*,2\*NUM\_SPS\_IDS> m\_cSPSBuf; 当前对应SPS\_ID的SPS  StatBuf<PictureParameterSet\*,256> m\_cPPSBuf; 当前对应PPS\_ID的PPS  std::list<SequenceParameterSet\*> m\_cSPSList; 历史上的SPS存在此(和当前SPS\_ID重合的)  std::list<PictureParameterSet\*> m\_cPPSList; 历史上的PPS存在此和当前PPS\_ID重合的)  调用store()函数将(按ID)  PPS =>m\_cPPSBuf 而从m\_cPPSBuf取出的PPS => m\_cPPSList  SPS =>m\_cSPSBuf 而从m\_cSPSBuf取出的SPS => m\_cSPSList |   m\_pcParameterSetMng->store( pcPPS )  m\_pcParameterSetMng->store( pcSPS )   |  | | --- | | 设置m\_auiActiveSPSId[uiDQId] = SPS\_ID  每个QualityLayer都有一个m\_auiActiveSPSId[uiDQId] |   m\_pcParameterSetMng->setActiveSPS( pcSPS->getSeqParameterSetId(), uiDQId + ui );  //设置SPS的内容  //设置subset SPS的内容  //设置PPS内容  // 初始化每层与帧分辨率有关的参数  m\_pcControlMng->initParameterSets( \*pcSPS, \*pcPPS ) ControlMngH264AVCEncoder  m\_apcYuvFullPelBufferCtrl[uiLayer]->initSPS( uiAllocY, uiAllocX, YUV\_Y\_MARGIN, YUV\_X\_MARGIN)  m\_apcYuvHalfPelBufferCtrl[uiLayer]->initSPS( uiAllocY, uiAllocX, YUV\_Y\_MARGIN, YUV\_X\_MARGIN, 1 )  m\_apcLayerEncoder [uiLayer]->initParameterSets( rcSPS, rcPPS )  //从ParameterSetMng类中的SPS和PPS缓存m\_cSPSBuf和m\_cPPSBuf读取并保存到  //std::list<SequenceParameterSet\*> m\_cUnWrittenSPS 和 std::list<PictureParameterSet\*> m\_cUnWrittenPPS;  m\_pcParameterSetMng->setParamterSetList( m\_cUnWrittenSPS, m\_cUnWrittenPPS )    // 写SEI相关信息  //m\_cUnWrittenSPS和m\_cUnWrittenPPS非空，则  m\_pcNalUnitEncoder->initNalUnit( pcExtBinDataAccessor ) //初始化写Nalunit的类  m\_pcBitWriteBuffer->initPacket( (UInt\*)(m\_pucTempBuffer), m\_uiPacketLength-1 ) //初始化写比特流的类  //读取SPS/PPS  m\_pcNalUnitEncoder->write( rcSPS )  rcSPS.write( m\_pcHeaderSymbolWriteIf ) //由m\_pcHeaderSymbolWriteIf写NalUnit  m\_pcNalUnitEncoder->closeNalUnit( uiBits )  xWriteTrailingBits() //写1000…补齐  m\_pcBitWriteBuffer->flushBuffer() //把拖尾等最后的数据写入并计算bit  convertRBSPToPayload(…) //加上0x03  m\_pcWriteBitstreamToFile->writePacket ( &m\_cBinDataStartCode ) //先写starcode 0x00000001 区分NALUNIT  m\_pcWriteBitstreamToFile->writePacket ( &cExtBinDataAccessor ) //再写实际NAL内容 |

//计算每层图像宽高、Crop信息和luma与chroma的Offset等

**//对每个Frame ==============================START===============================**

//对每一层 XXXXXXXXXXXXXXX START

|  |
| --- |
| 本函数的重要参数：在H264AVCEndoderTest::go()函数里  PicBuffer\* apcOriginalPicBuffer [MAX\_LAYERS]; //原始图像list  PicBuffer\* apcReconstructPicBuffer [MAX\_LAYERS]; //重建图像list  PicBufferList acPicBufferOutputList [MAX\_LAYERS]; //输出图像list  PicBufferList acPicBufferUnusedList [MAX\_LAYERS];  ExtBinDataAccessorList cOutExtBinDataAccessorList; //输出NalUnit的list |

//从PicBufferList m\_acUnusedPicBufferList [MAX\_LAYERS] 中弹出一个PicBuffer

//并存入PicBufferList m\_acActivePicBufferList [MAX\_LAYERS];

xGetNewPicBuffer( apcReconstructPicBuffer [uiLayer], uiLayer, auiPicSize[uiLayer] ) //重建帧

xGetNewPicBuffer( apcOriginalPicBuffer [uiLayer], uiLayer, auiPicSize[uiLayer] ) //原始帧

m\_apcReadYuv[uiLayer]->readFrame(…) // 读取一帧到apcOriginalPicBuffer [uiLayer]

XXXXXXXXXXXXXXX END

//处理一帧，所有Layer都处理，数据写入cOutExtBinDataAccessorList CreaterH264AVCEncoder调用

m\_pcH264AVCEncoder->process(cOutExtBinDataAccessorList,apcOriginalPicBuffer,

apcReconstructPicBuffer, acPicBufferOutputList, acPicBufferUnusedList )

🡪m\_pcH264AVCEncoder->process( rcExtBinDataAccessorList, H264AVCEncoder调用

apcOriginalPicBuffer,

apcReconstructPicBuffer,

apcPicBufferOutputList,

apcPicBufferUnusedList )

xWrite ( cOutExtBinDataAccessorList, uiBytesUsed ) //把编码一帧后的NALUNIT写到比特流中

//重复，直到cOutExtBinDataAccessorList为空

m\_pcWriteBitstreamToFile->writePacket( &m\_cBinDataStartCode ) //写起始码

m\_pcWriteBitstreamToFile->writePacket( rcList.front() ) //写NalUnit

**//对每层**

xWrite ( acPicBufferOutputList[uiLayer], uiLayer ) //把输出图像（重建文件）写到文件中

//从list中弹出一个图像

m\_apcWriteYuv[uiLayer]->writeFrame()

m\_cFile.write( pucSrc, uiWidth ) //重复调用此函数写重建图像

xRelease( acPicBufferUnusedList[uiLayer], uiLayer )

xRemovePicBuffer( rcPicBufferList, uiLayer ) //删除未使用的图像

//从m\_acActivePicBufferList中找到acPicBufferUnusedList中每一帧

//从m\_acActivePicBufferList中删除，添加到m\_acUnusedPicBufferList中

**==============================END===============================**

m\_pcH264AVCEncoder->finish(…)

m\_pcH264AVCEncoder->finish(…)

**//把不足GOP数量剩下的帧进行编码**

xProcessGOP( apcPicBufferOutputList, apcPicBufferUnusedList )

//从AccessUnitDataList m\_cAccessUnitDataList的成员std::list<AccessUnitData>m\_cAccessUnitDataList

//取出编码后的NalUnit到cOutExtBinDataAccessorList

m\_cAccessUnitDataList.emptyNALULists( rcExtBinDataAccessorList )

m\_apcLayerEncoder[uiLayer]->finish() //输出每层的编码信息，如比特数和PSNR等

xWrite ( cOutExtBinDataAccessorList, uiWrittenBytes ) //把cOutExtBinDataAccessorList中的Nal写到比特流

//对每一层

xWrite ( acPicBufferOutputList[uiLayer], uiLayer ) //把输出图像（重建文件）写到文件中

xRelease( acPicBufferUnusedList[uiLayer], uiLayer ) //删除未使用的图像

//写SEI信息

m\_pcH264AVCEncoder->writeParameterSets( &cExtBinDataAccessor, pcAVCSPS, bMoreSets )

m\_pcWriteBitstreamToFile->writePacket ( &m\_cBinDataStartCode )

m\_pcWriteBitstreamToFile->writePacket ( &cExtBinDataAccessor )

m\_pcWriteBitstreamToFile->uninit() //关闭比特流文件

m\_pcWriteBitstreamToFile->destroy() //释放空间

ScalableDealing() //把临时的比特流文件转成Scalable比特流

//结束函数

## pcH264AVCEncoderTest->destroy()

//释放m\_cBinDataStartCode、m\_pcH264AVCEncoder、读写文件、m\_pcEncoderCodingParameter和一些List的空间

# H264AVCEncoder::process( ExtBinDataAccessorList& rcExtBinDataAccessorList,

PicBuffer\* apcOriginalPicBuffer [MAX\_LAYERS], //每层的原始帧

PicBuffer\* apcReconstructPicBuffer [MAX\_LAYERS], //每层的重建帧

PicBufferList\* apcPicBufferOutputList, //输出图像List

PicBufferList\* apcPicBufferUnusedList ) //未使用图像List

这个H264AVCEncoder是H264AVCEncoderTest的CreaterH264AVCEncoder的SVC编码器H264AVCEncoder

//每一层的原始帧apcOriginalPicBuffer🡺H264AVCEncoder::PicBufferList m\_acOrgPicBufferList[MAX\_LAYERS]

//每一层的重建帧apcReconstructPicBuffer 🡺H264AVCEncoder::PicBufferList m\_acRecPicBufferList[MAX\_LAYERS]

关联Org和Rec图像List！！！！！！！！！！！！！！！！！！！！！！

**//当读到了一个GOP数量的帧时，进行编码------------------并不是读完所有帧才编码**

xProcessGOP( apcPicBufferOutputList, apcPicBufferUnusedList )

|  |  |  |
| --- | --- | --- |
| **//对每一层 这里都是每一层！！！！！！！！！！！！！！！！！！！！！！！！！！！**  //设置D\_id和Q\_id  //从m\_cAccessUnitDataList弄出一个AU，读取对应层的原始帧  m\_apcLayerEncoder[uiLayer]->initGOP( m\_cAccessUnitDataList.getAccessUnitData( MSYS\_UINT\_MAX ),  m\_acOrgPicBufferList[uiLayer] ) 注意 这是LayerEncoder   |  | | --- | | LayerEncoder的成员  Frame\* m\_apcFrameTemp[NUM\_TMP\_FRAMES]; // auxiliary frame memories  Frame\*\* m\_papcFrame; //读取每层的原始帧 编码后的重建帧  Frame\*\* m\_papcELFrame; // GOP最高EL层的参考帧  Frame\* m\_pcResidualLF; // frame stores for residual data  Frame\* m\_pcResidualILPred; // 层间预测的值(像素域)  Frame\* m\_pcSubband; //去块滤波之前的重建帧  Frame\* m\_apcBaseFrame[2]; // RefBasePic，GOP第一和最后一帧  Frame\* m\_pcAnchorFrameOriginal; // GOP最后一针的原始数据  Frame\* m\_pcAnchorFrameReconstructed; // GOP最后一帧的重建数据  Frame\* m\_pcBaseLayerFrame; // BL的m\_pcSubband  Frame\* m\_pcBaseLayerResidual; // BL的 m\_pcResidualILPred |   xInitGOP ( rcPicBufferInputList ) ---------------------------initGOP----------START-------------------   |  | | --- | | //如果第一个GOP已经编码  LayerEncoder成员Frame\*\* m\_papcFrame的第一帧拷贝LayerEncoder成员  Frame\* m\_pcAnchorFrameOriginal的内容和参数  m\_papcELFrame只设置参数，没读取内容  //对GOP中每一帧  m\_papcFrame读取帧，设置POC  m\_papcFrame [ uiFrame ]->setPicParameters ( \*m\_pcResizeParameters ) //设置ReSize 帧和场的信息  setPicParameters( cPP, ePicType )  xUpdatePicParameters()  m\_papcELFrame不读取帧，但设置POC和PicParameters  //拷贝GOP最后一帧作为Anchor （第一个I帧不算GOP的）  m\_pcAnchorFrameOriginal ->copyAll ( m\_papcFrame[ m\_uiGOPSize ] )  m\_pcAnchorFrameOriginal ->copyPicParameters (\*m\_papcFrame[ m\_uiGOPSize ] ); //拷贝Resize信息  m\_pcAnchorFrameReconstructed ->copyPicParameters (\*m\_papcFrame[ m\_uiGOPSize ] )  //LayerEncoder的成员ControlData\* m\_pacControlData用来控制GOP每一帧，其成员MbDataCtrl\* //getMbDataCtrl 用来控制所有Mb----------作为数组存在，关联MbData的MvD和Motion和TCoeff  为每一帧初始化m\_pacControlData和pcMbDataCtrl  //设置SliceHeader和参考队列(marking reordering MMCO)  xInitSliceHeadersAndRefLists()  清空按帧播放顺序的参考队列m\_acRefPicListFrameId\_L0 和m\_acRefPicListFrameId\_L1  初始化播放顺序、编码顺序、时间分层顺序数组  xInitRefFrameLists( uiFrameIdInGOP, uiTemporalId ) //设置前后向队列  修正参考队列，并计算队列长度  **//对GOP每一帧**  xInitSliceHeader( uiFrameIdInGOP, FRAME )  //SH存在m\_pacControlData[ uiFrameIdInGOP ]  xSetSliceHeaderBase ( \*pcSliceHeader, uiFrameIdInGOP, ePicType ) //设置SH基本参数  xSetSliceTypeAndRefLists ( \*pcSliceHeader, uiFrameIdInGOP, ePicType )  从m\_acRefPicListFrameId\_LX将播放顺序转成FrameNum顺序  设置SliceType和ref\_idx等等  xSetReorderingCommandsST() //参考队列重排序  xSetMMCOAndUpdateParameters ( \*pcSliceHeader, uiFrameIdInGOP, ePicType ) //进行MMCO  //如果有去块滤波，初始化滤波参数  pcSliceHeader->getDeblockingFilterParameter().setDisableDeblockingFilterIdc ( uiFilterIdc);  pcSliceHeader->getDeblockingFilterParameter().setSliceAlphaC0Offset ( 2 \* m\_iAlphaOffset );  pcSliceHeader->getDeblockingFilterParameter().setSliceBetaOffset ( 2 \* m\_iBetaOffset );  //计算并在SH里写POC  m\_pcPocCalculator->setPoc( \*pcSliceHeader, m\_papcFrame[uiFrameIdInGOP]->getPoc() + i2ndFieldOffset ) );  rcSliceHeader.setBotFieldPoc( );  rcSliceHeader.setDeltaPicOrderCntBottom ()  rcSliceHeader.setPicOrderCntLsb ( iCurrPoc & ~( -1 << m\_iBitsLsb ) );  m\_papcFrame[uiFrameIdInGOP]->setPoc( \*pcSliceHeader )    //初始化SH里的ESS参数    //初始化SliceGroup  pcSliceHeader->setSliceGroupChangeCycle(1);  pcSliceHeader->FMOInit(); |   xSetScalingFactors ()  //对每个时间层  xSetScalingFactors( uiLevel ) //设定每帧的ScalingFactor    xClearELPics () //让所有EL的参考层不可用  m\_papcELFrame[uiIndex]->setUnvalid()  //如果已经编码了GOP，则从前面的GOP中拷贝出重建的最后一帧  xEncodePicture( bPicCoded, 0, 0, rcAccessUnitData, rcPicBufferInputList )  m\_pcAnchorFrameReconstructed 🡪 m\_papcFrame[ uiFrameIdInGOP ]  m\_apcBaseFrame[1]🡪 m\_apcBaseFrame[0]    ---------------------------initGOP----------END-----------------------------    //一个GOP中最多允许的65个AU，**对每一个AU**  **//对每一层**  m\_apcLayerEncoder[uiLayer]->process( uiAUIndex,  m\_cAccessUnitDataList.getAccessUnitData( uiAUIndex ), //没找到对应号就新建  m\_acOrgPicBufferList [uiLayer], m\_acRecPicBufferList [uiLayer],  apcPicBufferUnusedList[uiLayer], m\_pcParameterSetMng ) ); 后面分析 GOP中一个AU的一层  //如果是第一个GOP，还要设置Profile和LevelIdc及ConstraintXFlag  **//对每一层**  把重建队列m\_acRecPicBufferList[ uiLayer ]里的帧给输出队列apcPicBufferOutputList[ uiLayer ] (go()里)  m\_acOrgPicBufferList[ uiLayer ]和m\_acRecPicBufferList[ uiLayer ]都给apcPicBufferUnusedList[ uiLayer ] (go()里)  清空原始和重建队列 |

//把编码得到的NalUnit写到传入参数rcExtBinDataAccessorList中

m\_cAccessUnitDataList.emptyNALULists( rcExtBinDataAccessorList );

GOP中一个AU的一层 调用函数

# LayerEncoder::process( UInt uiAUIndex, //这是CodingIndex

AccessUnitData& rcAccessUnitData,

PicBufferList& rcPicBufferInputList, // H264AVCEncoder::m\_acOrgPicBufferList [uiLayer],

PicBufferList& rcPicBufferOutputList, // H264AVCEncoder::m\_acRecPicBufferList [uiLayer],

PicBufferList& rcPicBufferUnusedList, // H264AVCEncoder::apcPicBufferUnusedList[uiLayer],

ParameterSetMng\* pcParameterSetMng )

|  |
| --- |
| xInitBitCounts() //统计lower layer编码的比特数  xUpdateELPics()  **//对GOP每一帧，如果m\_papcELFrame[uiIndex]是Unvalid**  m\_pcH264AVCEncoder->getELRefPic( m\_uiDependencyId, uiTemporalId, uiFIdInTId );  m\_apcLayerEncoder[uiELId]->getRefPic( uiTemporalId, uiFrameIdInTId );  //得到相同分辨率的最高EL层图像  m\_papcELFrame[uiIndex]->copy( const\_cast<Frame\*>( pcELPic ), FRAME ) //拷贝到m\_papcELFrame  xFillAndUpsampleFrame( m\_papcELFrame[uiIndex], FRAME, m\_bFrameMbsOnlyFlag ) //对最高层EL参考图像进行处理  m\_pcYuvFullPelBufferCtrl->initMb() //初始化整像素的宏块大小  m\_pcYuvHalfPelBufferCtrl->initMb() //初始化1/2像素的宏块大小  pcFrame->initHalfPel( pHPData ) // Frame类  getHalfPelYuvBuffer()->init( rpucYuvBuffer )  m\_rcYuvBufferCtrl.initMb(); //进行1/2像素的扩充  pcFrame->extendFrame( m\_pcQuarterPelFilter, ePicType, bFrameMbsOnlyFlag )  getFullPelYuvBuffer()->fillMargin( ) //边缘填充  m\_rcYuvBufferCtrl.initMb();  xFillPlaneMargin()  pcQuarterPelFilter->filterFrame(getFullPelYuvBuffer(), getHalfPelYuvBuffer() ) //进行int->1/2像素的插值滤波  xEncodePicture( bPicCoded, uiTemporalId, uiFrameIdInGOP, rcAccessUnitData, rcPicBufferInputList ) //后面分析  //当一个GOP编码完成  xStoreReconstruction( rcPicBufferOutputList )  //对GOP中每一帧，从输出队列H264AVCEncoder::m\_acRecPicBufferList中取出一个PicBuffer  m\_papcFrame[uiIndex<<m\_uiNotCodedStages]->store( pcPicBuffer )  getFullPelYuvBuffer()->storeToPicBuffer ( pcPicBuffer )  xFinishGOP ( rcPicBufferInputList, rcPicBufferOutputList, rcPicBufferUnusedList) //把output队列中过多的图像放到Unused队列 |

ErrVal // ×××××对每一个Layer GOP中的每一个AU ×××××

# LayerEncoder::xEncodePicture( Bool& rbPictureCoded, //是否编码，输出用

UInt uiTemporalId, //时间层ID

UInt uiFrameIdInGOP, //GOP中播放顺序

AccessUnitData& rcAccessUnitData, //该帧的AU

PicBufferList& rcPicBufferInputList ) // H264AVCEncoder::m\_acOrgPicBufferList [uiLayer],

//如果是IDR帧

把LayerEncoder的成员Bool m\_abCoded[(1<<MAX\_DSTAGES)+1]全部置0 （用来指示GOP中对应帧是否编码）

//如果是非第一个GOP的第一个帧

当前帧m\_papcFrame从m\_pcAnchorFrameReconstructed拷贝而来

m\_apcBaseFrame[0]从m\_apcBaseFrame[1]拷贝而来 //RefBasePic

重要参数

|  |  |  |
| --- | --- | --- |
| rcControlData | m\_pacControlData[ uiFrameIdInGOP ] | 帧控制类 |
| pcMbDataCtrl | rcControlData.getMbDataCtrl() | 宏块控制类 |
| pcFrame | m\_papcFrame [ uiFrameIdInGOP] | 当前帧 |
| pcResidualLF | m\_pcResidualLF | xStoreEstimation ，去块滤波计算CBP |
| pcResidualILPred | m\_pcResidualILPred | xStoreEstimation |
| pcOrgFrame | m\_apcFrameTemp [ 2 ] | pcFrame的编码前内容，原始帧像素 |
| pcPredSignal | m\_apcFrameTemp [ 3 ] | xStoreEstimation,没任何作用 |
| rcOutputList | rcAccessUnitData.getNalUnitList() | AU的NalUnit队列 |
|  | m\_pcSubband | 一层编码完成后的重建帧（去块滤波之前） |
|  | m\_apcBaseFrame[ 0/1 ] | GOP的两个RefBasePic |
| pcTempBaseFrame | m\_apcFrameTemp[0] | 辅助INTRA上采样用 （在xInitControlData里） |
| pcTempFrame | m\_apcFrameTemp[1] | 辅助INTRA上采样用 （在xInitControlData里） |

//RateControl相关，未关注

xInitControlData( uiFrameIdInGOP, uiTemporalId, ePicType )

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| xSetBaseLayerData( uiFrameIdInGOP, ePicType ) //设置BaseLayer信息  m\_pcH264AVCEncoder->getBaseLayerStatus( uiBaseLayerId, m\_uiDependencyId, ePicType, uiTemporalId )  m\_apcLayerEncoder[ruiBaseLayerId]->getBaseLayerStatus( bExists, ePicType, uiTemporalId ) //检查BaseLayerId  //由BaseLayer的Size信息更新本层的ESS信息  m\_pcResizeParameters->updatePicParameters( m\_papcFrame[ uiFrameIdInGOP ]->getPicParameters( ePicType ) );  设置Inter-layer 预测的参数 （BaseMode、Motion、Residual）  检查Level的约束  rcControlData.setBaseLayer( uiBaseLayerId ); //最后才确定BaseLayer的Id  设置QP & lambda  pcSliceHeader->setTCoeffLevelPredictionFlag( m\_pcLayerParameters->getTCoeffLevelPredictionFlag() );  xGetAndSetPredictionLists( rcControlData, uiFrameIdInGOP, ePicType, true ) //设置参考帧队列  //设置rcRefListStruct = rcControlData.getRefListStruct();  //当前为RC队列：rcRefListStruct.acRefFrameListRC[ X ]  获得当前帧状态eRLU： RLU\_REF\_BASE\_PIC (KeyPicture)或者 RLU\_RECONSTRUCTION  xGetPredictionLists( rcRefList0, rcRefList1, uiFrameIdCol, uiFrameIdInGOP, ePicType, eRLU, bHalfPel )  //对LayerEncoder的m\_acRefPicListFrameId\_LX[ uiFrameIdInGOP ]队列**每一帧**  pcFrame = xGetRefFrame( uiFrmId, eRefListUsage ); //获得对应index的参考帧  //如果是RefBasePic：m\_apcBaseFrame[ 0/1 ] ，否则m\_papcELFrame（MGS）或m\_papcFrame  //如果没有extend，还要xFillAndUpsampleFrame() / xFillAndExtendFrame()  rcRefListX.add( pcFrame->getPic( FRAME ) //加入队列  pcFrame->setLongTerm( m\_bUseLongTermPics );  // uiFrameIdCol是最后加入RCList1的FrameId，即List1的第一帧（因为是栈）     |  | | --- | | 如果是KeyPicture或者使用Std(MGScontrol=0)：ME和MC队列都用RC队列------同层的参考帧  MGScontrol=1：ME用EL的队列，MC用RC(std)的队列  MGScontrol=2：ME和MC都用EL的队列 |   SliceHeader写的是Std的RC队列  //当是B片时pcMbDataCtrl0L1 = m\_pacControlData[ rcRefListStruct.uiFrameIdCol ].getMbDataCtrl()，否则为空  rcControlData.setMbDataCtrl0L1( pcMbDataCtrl0L1 ); //这个是co-located MB的控制类  //更新ESS信息  m\_pcResizeParameters->updatePicParameters( m\_papcFrame[ uiFrameIdInGOP ]->getPicParameters( ePicType ) );  xInitBaseLayerData( rcControlData, ePicType )  先将m\_pcBaseLayerRec 、m\_pcBaseLayerSbb、m\_pcBaseLayerCtrl 、m\_pcBaseLayerCtrlField 设置成0  **//如果BaseLayer存在**  m\_pcH264AVCEncoder->getBaseLayerDataAvlb(pcBaseFrame, pcBaseResidual, pcBaseDataCtrl,  rcControlData.getBaseLayerId(),bBaseDataAvailable,  ePicType, pcSliceHeader->getTemporalId() )  🡪m\_apcLayerEncoder[uiBaseLayerId]->getBaseLayerDataAvlb(pcFrame, pcResidual, pcMbDataCtrl,  ePicType, uiTemporalId ) //获得BaseLayer信息   |  |  | | --- | --- | | LayerEncoder::xInitBaseLayerData()参数 | BaseLayer 在LayerEncoder里 | | pcBaseFrame | m\_pcSubband | | pcBaseResidual | m\_pcResidualILPred | | pcBaseDataCtrl | m\_pacControlData[m\_uiLastCodedFrameIdInGOP].getMbDataCtrl() |   由BaseLayer的信息pcBaseDataCtrl更新ESS内容m\_pcResizeParameters  //判断分辨率是否改变并设置  rcControlData.setSpatialScalability( m\_pcResizeParameters->getSpatialResolutionChangeFlag() );  m\_pcH264AVCEncoder->getBaseLayerData( \*pcSliceHeader, pcBaseFrame, pcBaseResidual, pcBaseDataCtrl,  m\_pcResizeParameters->getSpatialResolutionChangeFlag(),  rcControlData.getBaseLayerId(),ePicType,  pcSliceHeader->getTemporalId() )  🡪 m\_apcLayerEncoder[uiBaseLayerId]->getBaseLayerData( rcELSH, pcFrame, pcResidual, pcMbDataCtrl,  bSpatialScalability, ePicType, uiTemporalId ) );  //再设置一遍，觉得多余   |  |  | | --- | --- | | LayerEncoder::xInitBaseLayerData()参数 | BaseLayer 在LayerEncoder里，先复制给左边 | | pcBaseFrame，  最后🡺m\_pcBaseLayerFrame | m\_pcSubband 去块滤波前的重建帧 | | pcBaseResidual  最后🡺m\_pcBaseLayerResidual | m\_pcResidualILPred | | pcBaseDataCtrl | m\_pacControlData[m\_uiLastCodedFrameIdInGOP].getMbDataCtrl() |   //如果分辨率改变bSpatialScalability则  pcFrame = m\_pcSubband = LayerEncoder::Frame m\_apcFrameTemp[0] (数据一样，同一地址)  m\_pcLoopFilter->process(\*pcSliceHeader, pcFrame, pcResidual, m\_pacControlData[uiPos].getMbDataCtrl(),  &rcELSH.getInterLayerDeblockingFilterParameter(),  m\_pacControlData[uiPos].getSpatialScalability() ) //因为是去块滤波前的  **//如果BaseLayer的控制器pcBaseDataCtrl存在**  pcSliceHeader->setSCoeffResidualPredFlag( m\_pcResizeParameters ); //同分辨率的EL，TCoeff=FALSE则为真  m\_pcBaseLayerCtrl->initSlice( \*pcSliceHeader, PRE\_PROCESS, false, NULL )  把当前SliceHeader给了m\_pcBaseLayerCtrl的m\_pcSliceHeader  m\_pcBaseLayerCtrl->upsampleMotion(…)  **//对BaseLayer每个宏块**  cMotionUpsampling.resample( iMbX, iMbY ) MotionUpsampling   |  |  |  |  |  | | --- | --- | --- | --- | --- | | xInitMb ( iMbXCurr, iMbYCurr ) //设置宏块大小、块模式、每块ref\_idx等信息  xSetPartIdcArray()  **//对16个4x4块**  xGetRefLayerPartIdc( ( iX << 2 ) + 1, ( iY << 2 ) + 1, m\_aaiPartIdc[iX][iY] ) //得到该块在参考层的4x4块id  xGetRefLayerMb(iXInsideCurrMb,iYInsideCurrMb,iBaseMbIdx,iXInsideBaseMb, iYInsideBaseMb )  //标准G6.1过程 得到该4x4块在参考层的宏块id和在宏块内的坐标  //如果是InCropWindow且不是INTRA\_BL，**对每个List**  xGetRefIdxAndInitialMvPred( ListIdx( iListIdx ) )  //对每个4x4块  xGetInitialBaseRefIdxAndMv() //获得BaseLayer的MotionData和Mv，修正Mv  //ref\_idx和Mv保存在m\_aaaiRefIdx / m\_aaacMv    //对每个8x8块  xDeriveBlockModeAndUpdateMv( iB8x8Idx ) //根据宏块分区更新Mv    xDeriveMbMode () //由BaseMbMode检查ref\_idx和Mv，并得到MbMode  xDeriveFwdBwd ()  xSetInterIntraIdc () //设置m\_aabBaseIntra[][]  //如果是InCropWindow，则检查分区方式、ref\_idx和Mv是否正确，设置m\_bResPredSafe  xSetResPredSafeFlag ()  //从BaseLayer得到MbMode、分区方式、ref\_idx和Mv  xSetPredMbData()  //如果是INTRA\_BL或非InCropWindow，则不得到Motion信息  //否则拷贝每个List的每个8x8块的ref\_idx和每个4x4块的Mv  InCropWindow的话，MbMode、FwdBwd等等也拷贝  //SNR伸缩------SCoeff/TCoeff 且是InCropWindow  rcMbData.copyTCoeffs ( rcMbDataBase ); //拷贝BL的变换系数、CBP、QP、TransformSize  拷贝BLSkipFlag   |  |  | | --- | --- | | IntraBL | TCoeffPred | | 拷贝BL的MbMode | 拷贝IntraPredMode  CBP、QP、TransformSize | |   //如果BL有加权预测，直接拷贝BL的权值和加权表  pcSliceHeaderCurr->setLumaLog2WeightDenom( pcSliceHeaderBase->getLumaLog2WeightDenom() );  pcSliceHeaderCurr->setChromaLog2WeightDenom( pcSliceHeaderBase->getChromaLog2WeightDenom() );  pcSliceHeaderCurr->getPredWeightTable( LIST\_0 ).copy( pcSliceHeaderBase->getPredWeightTable( LIST\_0 ) )  pcSliceHeaderCurr->getPredWeightTable( LIST\_1 ).copy( pcSliceHeaderBase->getPredWeightTable( LIST\_1 ) );  **//如果BaseLayer可用(**bBaseDataAvailable**)**  m\_pcBaseLayerResidual->residualUpsampling( pcBaseResidual, m\_cDownConvert, m\_pcResizeParameters, pcBaseDataCtrl )  rcDownConvert.residualUpsampling( this, pcBaseFrame, pcParameters, pcMbDataCtrlBase ); //像素帧Resize  //BL的LayerEncoder的m\_pcResidualILPred进过残差上采样赋值给了m\_pcBaseLayerResidual  rcControlData.setBaseLayerSbb( m\_pcBaseLayerResidual );  **//如果BaseLayer可用(**bBaseDataAvailable**)**  m\_pcBaseLayerFrame->intraUpsampling( pcBaseFrame, pcTempBaseFrame, pcTempFrame,m\_cDownConvert,  m\_pcResizeParameters, pcBaseDataCtrl, m\_pcBaseLayerCtrl,  m\_pcBaseLayerCtrlField, m\_pcReconstructionBypass, m\_bCIUFlag,  m\_apabBaseModeFlagAllowedArrays[0], m\_apabBaseModeFlagAllowedArrays[1] )  //进行Intra上采样，从pcBaseFrame采样到m\_pcBaseLayerFrame pcTempBaseFrame和pcTempFrame辅助采样  **rcControlData保存了m\_pcBaseLayerResidual和m\_pcBaseLayerFrame**  //如果是EL，用BaseLayer的加权预测：  pcSliceHeader->setBasePredWeightTableFlag( true );  //否则，自己计算  m\_pcSliceEncoder->xSetPredWeights( \*pcSliceHeader, m\_papcFrame[uiFrameIdInGOP], rcControlData.getRefListStruct() );  pcSliceHeader->setBasePredWeightTableFlag( false ); |

//SEI、HRD、RedundantPicture、LARDO、SIP，未关注

xEncodeLayerRepresentation( rcOutputList, rcControlData, pcOrgFrame, pcFrame, pcResidualLF, pcResidualILPred, pcPredSignal,

uiBits, cPicOutputDataList, uiFrameIdInGOP, ePicType ) //后面分析

m\_pcSubband->copy( pcFrame, ePicType ) ) //保存去块滤波之前的重建帧

//SEI、RateControl、RedPic相关，未关注

//如果是增强层

m\_pcSliceEncoder->updateBaseLayerResidual( rcControlData, m\_uiFrameWidthInMb );

pcMbDataCtrl->initSlice( rcSliceHeader, DECODE\_PROCESS, false, NULL ) //初始化宏块控制器

//对每个宏块

pcMbDataCtrl ->initMb( pcMbDataAccess, uiMbY, uiMbX )

pcBaseLayerCtrl ->initMb ( pcMbDataAccessBase, uiMbY, uiMbX )

//如果该宏块不用ResidualPred，清除该BL对应的Residual宏块 (rcControl的m\_pcBaseLayerSbb，即LayerEncoder的**m\_pcBaseLayerResidual**)

//去块滤波

m\_pcLoopFilter->process( \*pcSliceHeader, pcFrame, pcResidualLF, pcMbDataCtrl, 0, rcControlData.getSpatialScalability() )

**注意：滤波的仍然是pcFrame，pcResidualLF只是用来计算CBP！！！！**

输出PSNR等信息

xClearBufferExtensions() //清除m\_papcFrame[]、m\_papcELFrame[]、m\_pcSubband、m\_pcResidualLF、

//m\_pcResidualILPred和m\_apcBaseFrame[X]的1/2像素缓存 ，**注意，是整个GOP的**

更新m\_uiLastCodedFrameIdInGOP = uiFrameIdInGOP和 m\_uiLastCodedTemporalId = uiTemporalId;

//如果是GOP最后一帧

m\_pcAnchorFrameReconstructed拷贝该帧内容🡺m\_papcFrame[ m\_uiGOPSize ]

ErrVal

# LayerEncoder::xEncodeLayerRepresentation(ExtBinDataAccessorList& rcOutExtBinDataAccessorList, // rcAccessUnitData.getNalUnitList()

ControlData& rcControlData, // m\_pacControlData[ uiFrameIdInGOP ];

Frame\* pcOrgFrame, // m\_apcFrameTemp[ 2 ]

Frame\* pcFrame, // m\_papcFrame[ uiFrameIdInGOP ]

Frame\* pcResidualLF, // m\_pcResidualLF

Frame\* pcResidualILPred, // m\_pcResidualILPred

Frame\* pcPredSignal, // m\_apcFrameTemp[ 3 ];

UInt& ruiBits, // 计算编码比特数

PicOutputDataList& rcPicOutputDataList, // 输出图像list ,仅仅输出屏幕用

UInt uiFrameIdInGOP, // GOP内第几帧

PicType ePicType ) // FRAME

参数：

ExtBinDataAccessorList cTmpExtBinDataAccessorList; // 给closeAndAppendNalUnits 函数用的临时NALUNIT list

ExtBinDataAccessorList acExtBinDataAccessorList[16]; // 每个MGSVector 的NalUnit的list

PicOutputDataList acPicOutputDataList [16];

//对每个SliceGroup

//对每个Slice

|  |
| --- |
| 设置每个Slice的起止宏块  //当发nal\_unit\_type=1/5的NAL单元时，必然会先发一个nal\_unit\_type=14的prefix NAL unit  xWritePrefixUnit( acExtBinDataAccessorList[0], \*pcSliceHeader, uiBits )  xInitExtBinDataAccessor (m\_cExtBinDataAccessor ) //与LayerEncoder 中的BinData m\_cBinData关联起来  m\_pcNalUnitEncoder->initNalUnit( &m\_cExtBinDataAccessor ) //m\_cExtBinDataAccessor 是操纵LayerEncoder的比特流用的  现在与NalUnit关联了  m\_pcBitWriteBuffer->initPacket( (UInt\*)(m\_pucTempBuffer), m\_uiPacketLength-1 ) //初始化packet 对齐比特  //BitWriteBuffer 的m\_pulStreamPacket 与NalUnitEncoder 的m\_pucTempBuffer 联系起来了    m\_pcNalUnitEncoder->writePrefix( rcSH )  rcSH.writePrefix( \*m\_pcHeaderSymbolWriteIf ) ); //写到了m\_pcHeaderSymbolWriteIf变量中  m\_pcNalUnitEncoder->closeNalUnit( uiBit ) ); //与NalUnitEncoder 脱离关系 数据仍在 m\_cExtBinDataAccessor 中  xWriteTrailingBits() //写1000…补齐  m\_pcBitWriteBuffer->flushBuffer() //把拖尾等最后的数据写入并计算bit  convertRBSPToPayload( uiBits, uiHeaderBytes, m\_pucBuffer, m\_pucTempBuffer, m\_uiPacketLength ) ); //加上0x03  xAppendNewExtBinDataAccessor( rcOutExtBinDataAccessorList, &m\_cExtBinDataAccessor ) ); //把NalUnit数据存入List中  //初始化Slice数据的NalUnit  xInitExtBinDataAccessor ( m\_cExtBinDataAccessor ) //与LayerEncoder 中的BinData m\_cBinData关联起来  m\_pcNalUnitEncoder->initNalUnit( &m\_cExtBinDataAccessor ) //同前面的PrefixNalUnit  m\_pcNalUnitEncoder->write( \*pcSliceHeader ) // 写SliceHeader  m\_pcSliceEncoder->encodeSliceSVC( rcControlData, \*pcOrgFrame, \*pcFrame, pcResidualLF, pcResidualILPred, pcPredSignal,  ePicType,m\_uiNumMaxIter, m\_uiIterSearchRange, m\_bBiPred8x8Disable,  m\_bMCBlks8x8Disable,m\_uiMaxDeltaQp, uiBits ) 后面分析  m\_pcNalUnitEncoder->closeAndAppendNalUnits( auiBits, cTmpExtBinDataAccessorList, &m\_cExtBinDataAccessor,  m\_cBinData, m\_pcH264AVCEncoder, m\_uiQualityLevelCGSSNR,m\_uiLayerCGSSNR )  xWriteTrailingBits() //拖尾0  m\_pcBitWriteBuffer->flushBuffer() //刷新比特流  convertRBSPToPayload( uiBytes, uiHeaderBytes, pucPayload, pucRBSP, uiPayloadBufferSize )  //如果有MGSVector，需要递归调用pcCurrentWriteBuffer去写比特流  m\_pcBitWriteBuffer->uninit()  把临时的NALUNIT的list cTmpExtBinDataAccessorList分别存入不同的MGSVector的NALUNIT list  acExtBinDataAccessorList[iMGSIdx]中 |

**//所有SliceGroup编码完毕后**

每个MGSVector的list连接起来到输出list =rcAccessUnitData.getNalUnitList() = 中

|  |  |
| --- | --- |
| EL层 SCoeff | TCoeff |
| m\_pcSliceEncoder->updatePictureResTransform( rcControlData, m\_uiFrameWidthInMb )  根据当前宏块和BL宏块的关系（CBP、MbMode等）更新QP和TransformSize等 | m\_pcSliceEncoder->updatePictureAVCRewrite( rcControlData, m\_uiFrameWidthInMb )  从BaseLayer继承MbMode\IntraPredMode,修改TransformSize和Qp等 |

ErrVal

# SliceEncoder::encodeSliceSVC( ControlData& rcControlData, // control data

Frame& rcOrgFrame, // original frame

Frame& rcFrame, // reconstructed frame

Frame\* pcResidualFrameLF, // reconstructed residual for loop filter Frame\* pcResidualFrameILPred,// reconstructed residual for inter-layer prediction

Frame\* pcPredFrame, // prediction signal

PicType ePicType, // picture type

UInt uiNumMaxIter, // maximum number of iteration for bi-predictive search

UInt uiIterSearchRange, // search range for iterative search

Bool bBiPred8x8Disable, // if bi-prediction for blocks smaller than 8x8 is allowed

Bool bMCBlks8x8Disable, // if blocks smaller than 8x8 are disabled

UInt uiMaxDeltaQp, // maximum delta QP

UInt& ruiBits // size of coded data

)

参数

|  |  |  |
| --- | --- | --- |
| pcBaseIntraRecFrame | rcControlData.getBaseLayerRec() | LayerEncoder的  **m\_pcBaseLayerFrame** |
| pcBaseResidualFrame | rcControlData.getBaseLayerSbb() | LayerEncoder的  **m\_pcBaseLayerResidual** |
| rcOrgPic | rcOrgFrame.getPic( ePicType ) | 原始帧 |
| rcPic | rcFrame.getPic( ePicType ) | 原始帧/编码帧/重建帧 |
| pcResidualPicLF | pcResidualFrameLF ->getPic( ePicType ) | xStoreEstimation |
| pcResidualPicILPred | pcResidualFrameILPred ->getPic( ePicType ) | xStoreEstimation |
| pcPredPic | pcPredFrame->getPic( ePicType ) | xStoreEstimation |
| pcBaseIntraRecPic | pcBaseIntraRecFrame ->getPic( ePicType ) |  |
| pcBaseResidualPic | pcBaseResidualFrame ->getPic( ePicType ) |  |

pcMbDataCtrl->initSlice ( rcSliceHeader, ENCODE\_PROCESS, false, pcMbDataCtrl0L1 )

//B-slice设置co-located宏块控制

//设置去块滤波器

在MbDataCtrl的DynBuf<DBFilterParameter\*> m\_cDBFPBuffer中加入该Slice的滤波参数

//设置高宽信息和QP

pcBaseMbDataCtrl->initSlice ( rcSliceHeader, PRE\_PROCESS, false, NULL )

**//相比上面的，没有去块滤波的设置**

m\_pcControlMng->initSliceForCoding( rcSliceHeader ) ControlMngH264AVCEncoder

//选择熵编码模式

ControlMngH264AVCEncoder的MbSymbolWriteIf\* m\_pcMbSymbolWriteIf指向m\_pcCabacWriter/m\_pcUvlcWriter;

m\_pcMbSymbolWriteIf ->startSlice( rcSH ) //只关注CABAC

xInitContextModels( rcSliceHeader ) //调用每个模型的initBuffer()函数初始化上下文

CabaEncoder::start() //初始化cabac编码器参数

m\_pcBitWriteBufferIf->writeAlignOne() //前面的0补齐

//每个MGSVector层都要嵌套调用pcCurrentWriter->startSlice( rcSH )初始化cabac上下文

m\_pcMbEncoder ->initSlice ( rcSH ) //初始化成员MbEncoder\* m\_pcMbEncoder; **它只支持Uvlc！！**

MbCoder::initSlice( rcSH, this, MbEncoder::m\_pcRateDistortionIf ) //关联MbCoder和RD计算类的指针

UvlcWriter::init( m\_BitCounter ) // 关联Uvlc编码器和比特计数成员

UvlcWriter::startSlice( rcSH ) // 嵌套初始化MGSVector的Uvlc编码器状态

|  |  |
| --- | --- |
| m\_pcIntMbBestData | m\_acIntMbTempData[0] |
| m\_pcIntMbTempData | m\_acIntMbTempData[1] |
| m\_pcIntMbBest8x8Data | m\_acIntMbTempData[2] |
| m\_pcIntMbTemp8x8Data | m\_acIntMbTempData[3] |

//初始化成员 MbCoder\* m\_pcMbCoder;

m\_pcMbCoder ->initSlice ( rcSH, m\_pcMbSymbolWriteIf, m\_pcRateDistortion ) //关联实际熵编码器和RD计算类

m\_pcMotionEstimation ->initSlice ( rcSH )

m\_pcSampleWeighting->initSliceForWeighting(rcSH); //初始化加权预测参数

MotionVectorCalculation::initSlice( rcSH ) //初始化Mv计算 （MaxMv个数和SpatialDirect）

m\_pcSampleWeighting ->initSlice ( rcSH ) //初始化加权预测参数，与前面有一些不同

//设置FMO

m\_pcMbCoder->getBitCount() //得到已编码比特数

**//对每个宏块**

//RateControl部分，未看

pcMbDataCtrl ->initMb( pcMbDataAccess, uiMbY, uiMbX ) )

//宏块数据在MbDataCtrl的成员 MbData\* m\_pcMbData中

rcMbDataCurr = m\_pcMbData[ uiCurrIdx ];

rcMbDataCurr.getMbTCoeffs().clear(); //清空变换系数

rcMbDataCurr.initMbData( … ); //设置上个Mb的Qp和这个Mb的id、地址等信息

rcMbDataCurr.clear(); //清空Mb数据信息

给MbDataCtrl成员MbDataAccess\* m\_pcMbDataAccess开辟空间并初始化

|  |
| --- |
| 调用了：  xGetRefMbData() //获得宏块周围宏块 MbData  xGetOutMbData() //返回不可用的宏块m\_pcMbData[m\_uiSize]  xGetColMbData( uiIdxColTop ), //返回List1的第一帧的相同位置宏块 |

//设置宏块ForceQP

pcBaseMbDataCtrl->initMb ( pcMbDataAccessBase, uiMbY, uiMbX ) // 同上面的函数

m\_pcControlMng ->initMbForCoding ( \*pcMbDataAccess, uiMbY, uiMbX, false, false) ControlMngH264AVCEncoder

m\_apcYuvFullPelBufferCtrl[m\_uiCurrLayer]->initMb( uiMbY, uiMbX, bMbAff ) //设置该宏块的Y/U/V的offset

m\_apcYuvHalfPelBufferCtrl[m\_uiCurrLayer]->initMb( uiMbY, uiMbX, bMbAff )

//参数保存在YuvBufferCtrl的成员YuvBufferParameter m\_acBufferParam[ MAX\_FRAME\_TYPE ]中

//每次都要重新计算，由传入的宏块x，y号转换成posX和posY再计算宏块的offset

// int-pel 和 1/2-pel都计算

m\_pcMotionEstimation->initMb( uiMbY, uiMbX, rcMbDataAccess )

MotionCompensation::initMb( uiMbPosY, uiMbPosX, rcMbDataAccess) //设置Mv最大和最小范围

//真正设置宏块Qp，写入pcMbDataAccess->getMbData()

m\_pcMbEncoder->encodeMacroblockSVC(\*pcMbDataAccess, pcMbDataAccessBase, rcOrgPic, rcPic, pcResidualPicLF,

pcResidualPicILPred, pcPredPic, pcBaseIntraRecPic, pcBaseResidualPic, rcRefListStruct, uiMaxMvPerMb,

uiNumMaxIter, uiIterSearchRange, bBiPred8x8Disable, bMCBlks8x8Disable, true, uiDeltaQp, dLambda, dCost ) );

后面分析

//用Cabac/Cavlc进行熵编码，用m\_pcMbSymbolWriteIf进行写码流

m\_pcMbCoder->encode( \*pcMbDataAccess, pcMbDataAccessBase, ( uiMbAddress == uiLastMbAddress ), true )

//计算slice data的编码数据

ErrVal

# MbEncoder::encodeMacroblockSVC( MbDataAccess& rcMbDataAccess,

// current macroblock data

MbDataAccess\* pcMbDataAccessBase, // inferred macroblock data (from base layer)

const Frame& rcOrgFrame, // original frame

Frame& rcFrame, // reconstructed frame

Frame\* pcResidualLF, // reconstructed residual for loop filter

Frame\* pcResidualILPred, // reconstructed residual for inter-layer prediction

Frame\* pcPredSignal, // prediction signal

const Frame\* pcBaseLayerIntraRec, // base layer intra reconstruction

const Frame\* pcBaseLayerResidual, // base layer residual reconstruction

RefListStruct& rcRefListStruct, // reference picture lists

UInt uiMaxNumMv, // maximum number of MVs for current macroblock

UInt uiNumMaxIter, // number of iteration for bi-predictive search

UInt uiIterSearchRange, // search range for iterative search

Bool bBiPred8x8Disable, // if bi-prediction for blocks smaller than 8x8 is allowed

Bool bMCBlks8x8Disable, // if blocks smaller than 8x8 are disabled

Bool bSkipModeAllowed, // if skip mode is allowed

UInt uiMaxDeltaQp, // maximum delta QP

Double dLambda, // Langrangian multiplier

Double& rdCost // r-d cost for coded macroblock

)

m\_pcRateDistortionIf->setMbQpLambda( rcMbDataAccess, uiCurrQP, dLambda ) //设置计算RD的qp和lambda

//宏块数据初始化并清空

m\_pcIntMbBestData ->init( rcMbDataAccess );

m\_pcIntMbTempData ->init( rcMbDataAccess );

m\_pcIntMbBest8x8Data->init( rcMbDataAccess );

m\_pcIntMbTemp8x8Data->init( rcMbDataAccess );

m\_pcXDistortion->loadOrgMbPelData( rcOrgFrame.getFullPelYuvBuffer(), m\_pcIntOrgMbPelData );

//把原始帧的该宏块内容复制到XDistortion的成员YuvMbBuffer m\_cOrgData; 中，地址给m\_pcIntOrgMbPelData

参数

|  |  |
| --- | --- |
| IsEnhLayer | 通过no\_inter\_layer\_pred\_flag判断是不是增强层 |
| InCropWindow | 参考层的InCropWindowFlag为true |
| pcMbDataAccessBase | 只有InCropWindow为true， 该值才不为NULL |
| bSNRMode | InCropWindow为true，且TCoeff或者SCoeff的一种 |
| bTCoeffPredFlag | Slice Header的TCoeffLevelPredFlag |
| bSCoeffPredFlag | Slice Header的SCoeffResidualPredFlag, BL是INTER或者INTRA\_BL |
| bARP | Slice Header的AdaptiveResidualPredictionFlag |
| bDRP | Slice Header的DefaultResidualPredictionFlag |
| bELMbRefInter | InCropWindow为true，为P/B slice，且参考层不是INTRA预测 |
| bCheckWithResPred | 为P/B slice，InCropWindow为true，ARP（bELMbRefInter）或者DRP |
| bCheckWithoutResPred | 为P/B slice，ARP或者DRP=false或者InCropWindow=false |
| bWithAndWithoutRP | P/B slice，InCropWindow为true，ARP |
| bPreferResPred | 1是CABAC 0是CAVLC |
| bABM | Slice Header的AdaptiveBaseModeFlag |
| bDBM | Slice Header的DefaultBaseModeFlag |
| bCheckBaseMode | InCropWindow=true，ABM或者DBM |
| bNonBaseModeOk | DBM=false或者InCropWindow=false或者ABM=true |
| bCheckStdInter | 不是InCropWindow，或者AdaptiveBaseMode，或者不是BaseMode |
| bCheckIntraBL | InCropWindow，允许INTRA，参考层是INTRA预测，ABM或DBM |
| bCheckSpatialIntra | 允许INTRA，非InCropWindow或者ABM或者不是DBM |

注：

|  |  |  |  |
| --- | --- | --- | --- |
| **SCoeff** | | **TCoeff** | |
| no\_inter\_layer\_pred\_flag = 0  SpatialResolutionChangeFlag = 0  tcoeff\_level\_prediction\_flag = 0 | | no\_inter\_layer\_pred\_flag = 0  SpatialResolutionChangeFlag = 0  tcoeff\_level\_prediction\_flag = 1 | |
| 1 m\_rcSliceHeader.getSCoeffResidualPredFlag() = true  2 m\_pcMbDataAccessBase ！= NULL  3 m\_rcMbCurr.isIntra() = false  m\_rcMbCurr.getResidualPredFlag() = true  m\_pcMbDataAccessBase->getMbData().isIntra() = false  或者  m\_rcMbCurr.isIntraBL () = true  m\_pcMbDataAccessBase->getMbData().isIntraBL() = true | | 1 m\_rcSliceHeader.getTCoeffLevelPredictionFlag() = true  2 m\_pcMbDataAccessBase ！= NULL  3 m\_rcMbCurr.isIntra() = false  m\_rcMbCurr.getResidualPredFlag() = true  m\_pcMbDataAccessBase->getMbData().isIntra() = false  或者  m\_rcMbCurr.isIntra() = true  m\_rcMbCurr.getBLSkipFlag() = true  // m\_eMbMode >= INTRA\_4X4 m\_bBLSkipFlag | |
| 当前INTER预测  支持Residual预测  参考层不是INTRA预测 | **当前宏块和参考层宏块都是INTRA\_BL** | 当前INTER预测  支持Residual预测  参考层不是INTRA预测 | 当前INTRA预测  当前宏块是base mode |

如果bZeroBaseLayerResFlag = true，

|  |  |  |
| --- | --- | --- |
| bPreferResPred | 1 cabac | 0 cavlc |
| bCheckWithoutResPred | 0 | 1 |
| bCheckWithResPred | 1 | 0 |

我的思考：如果BL的残差系数为0且两者都要计算，则用cabac时ResidualPred效率高些（EL很可能也全0，那么省比特，且省变换时的计算量）

|  |  |  |  |
| --- | --- | --- | --- |
|  | bCheckWithResPred | | |
| bTCoeffPredFlag | bSCoeffPredFlag | 其他 |
| cBaseLayerBuffer  原始宏块减去它作为后面的原始参照，在过程中不改变！！  **这是像素域的残差预测** | 全零 | 参考层的预测值  XPel m\_sPred  (BL是INTER or INTRA\_BL) | 参考层的残差  m\_ResidualILPred) |
| bZeroBaseLayerResFlag | 参考层量化系数 | 参考层反量化系数和预测值 | m\_ResidualILPred |

先看看xStoreEstimation函数

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| rcMbBestData.getMbTCoeffs()的m\_sPred 拷入   |  |  |  |  | | --- | --- | --- | --- | | INTRA\_BL且非TCoeff | INTRA | Residual Pred | 其他(TCoeff / INTER且非Residual Pred) | | rcMbBestData.getTempYuvMbBuffer() | rcMbBestData | pcBaseLayerBuffer | 0 | | INTRA预测值  但这里是INTRA\_BL  应该是其BL的INTRA重建值(去块滤波之前) | INTRA预测重建的(去块滤波之前) | 就是上面的表格中的cBaseLayerBuffer  预测的残差 | 全零 | |

//流程

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | **bCheckWithResPred** | | | **bCheckWithoutResPred** | |
|  | subtract( cBaseLayerBuffer ) | | |  | |
| **bCheckBaseMode**  参考层不是INTRA | xEstimateMbBLSkip TRUE | | | xEstimateMbBLSkip FALSE | |
| **bCheckStdInter** | xEstimateMbDirect TRUE  xEstimateMb16x16 TRUE  xEstimateMb16x8 TRUE  xEstimateMb8x16 TRUE  xEstimateMb8x8 TRUE  xEstimateMb8x8Frext TRUE | | | xEstimateMbDirect FALSE  xEstimateMb16x16 FALSE  xEstimateMb16x8 FALSE  xEstimateMb8x16 FALSE  xEstimateMb8x8 FALSE  xEstimateMb8x8Frext FALSE | |
| P Slice  DRP=true | xEstimateMbSkip TRUE | | P Slice | xEstimateMbSkip FALSE |
| P Slice  DRP=false  bCheckWithoutResPred=false | xEstimateMbSkip FALSE | |
| bWithAndWithoutRP=true  bCheckWithoutResPred = false | xEstimateMbDirect FALSE | |  | |
| 参考层是INTER  未使用上面的所有 | xEstimateMb16x16 TRUE | | | xEstimateMb16x16 FALSE | |
|  | add( cBaseLayerBuffer ) | | |  | |
| **下面是全新的开始** | | | | | |
| **bCheckSpatialIntra** | | | **bCheckIntraBL** | | |
| xEstimateMbIntra16  xEstimateMbIntra8  xEstimateMbIntra4  xEstimateMbPCM | | | xEstimateMbIntraBL | | |

xStoreEstimation(rcMbDataAccess, \*m\_pcIntMbBestData, pcResidualLF, pcResidualILPred, pcPredSignal,

rcRefListStruct, &cBaseLayerBuffer )

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| //INTRA预测清除MotionData和Mvd，SKIP模式清除Mvd  //设置FwdBwd,可见是宏块编码完成后使用  【8x8块\_3】【8x8块\_2】【8x8块\_1】【0 0 Use\_List1 Use\_List0】  rcMbBestData.getMbTCoeffs()的m\_sPred 拷入下面的内容   |  |  |  |  | | --- | --- | --- | --- | | INTRA\_BL且非TCoeff | INTRA | Residual Pred | 其他(TCoeff / INTER且非Residual Pred) | | rcMbBestData.getTempYuvMbBuffer() | rcMbBestData | pcBaseLayerBuffer | 0 | | INTRA预测值  但这里是INTRA\_BL  应该是其BL的INTRA重建值(去块滤波之前) | INTRA预测重建的(去块滤波之前) | 就是上面的表格中的cBaseLayerBuffer | 全零啊 |   rcMbBestData.copyTo ( rcMbDataAccess ); // rcMbBestData 的一些信息（MbData/Coeff/Mv）存到 rcMbDataAccess  把RC参考队列中对于ref\_idx的参考帧读出存入m\_rcMbCurr.getMbMotionData ( eListIdx )的m\_acRefPicIdc [4]中   |  |  |  | | --- | --- | --- | | **INTRA** | pcPredSignal = rcMbBestData.getTempYuvMbBuffer()  Intra预测样点 | | | pcResidualLF = cResidual.setAllSamplesToZero();  全零 | | | pcResidualILPred = cResidual.setAllSamplesToZero();  全零 | | | **Residual Pred** | **Base Mode** | pcPredSignal = rcMbBestData.getTempYuvMbBuffer() - rcMbBestData.getTempBLSkipResBuffer () +\*pcBaseLayerBuffer  RC预测 - BLSkip的BL的残差+ BaseLayerBuffer(全零)  **赋值给 rcMbBestData** | | pcResidualLF = rcMbBestData - rcMbBestData.getTempYuvMbBuffer() + rcMbBestData.getTempBLSkipResBuffer ()  MC重建 - RC预测 + BLSkip的BL的残差(由cBaseLayerBuffer得来) | | pcResidualILPred = rcMbBestData - rcMbBestData.getTempYuvMbBuffer() + pcBaseLayerBuffer  MC重建 - RC预测 + BaseLayerBuffer(全零) | | **Others** | pcPredSignal = rcMbBestData.getTempYuvMbBuffer();  RC预测  **赋值给rcMbBestData** | | pcResidualLF = rcMbBestData - rcMbBestData.getTempYuvMbBuffer() + pcBaseLayerBuffer  MC重建 - RC预测 + BaseLayerBuffer(全零) | | pcResidualILPred = rcMbBestData - rcMbBestData.getTempYuvMbBuffer() + pcBaseLayerBuffer  MC重建 - RC预测 + BaseLayerBuffer(全零) | | **Others** | pcPredSignal = rcMbBestData.getTempYuvMbBuffer()  RC预测(非ResidualPred的INTER) | BL的intra预测(INTRA\_BL) | | | pcResidualLF = rcMbBestData - rcMbBestData.getTempYuvMbBuffer()  MC重建 -- RC预测(非ResidualPred的INTER) | BL的intra预测(INTRA\_BL) | | | pcResidualILPred = rcMbBestData - rcMbBestData.getTempYuvMbBuffer()  MC重建 -- RC预测(非ResidualPred的INTER) | BL的intra预测(INTRA\_BL) | | |

//计算RD

rdCost = m\_pcIntMbBestData->rdCost(); //最终的RD

//无函数体

m\_pcIntMbBestData ->uninit();

m\_pcIntMbTempData ->uninit();

m\_pcIntMbBest8x8Data->uninit();

m\_pcIntMbTemp8x8Data->uninit();

ErrVal

## MbEncoder::xEstimateMb16x16( IntMbTempData\*& rpcMbTempData,

IntMbTempData\*& rpcMbBestData,

RefListStruct& rcRefListStruct,

UInt uiMinQP,

UInt uiMaxQP,

UInt uiMaxNumMv,

UInt uiNumMaxIter,

UInt uiIterSearchRange,

MbDataAccess\* pcMbDataAccessBase,

Bool bResidualPred )

//运动估计队列用的是MEList

//如果允许MotionPred

iBLRefIdx [0] = pcMbDataAccessBase->getMbMotionData( LIST\_0 ).getRefIdx (); //ref\_idx

cBLMvPred [0] = pcMbDataAccessBase->getMbMotionData( LIST\_0 ).getMv (); // BL的mv 作为EL运动估计的MVP

rpcMbTempData->clear(); // 清空

//设置几个参数

rpcMbTempData->setMbMode( MODE\_16x16 );

rpcMbTempData->setBLSkipFlag( false );

rpcMbTempData->setResidualPredFlag( bResidualPred );

//对List0、List1中**每个**ref\_idx，进行ME

|  |
| --- |
| pcRefFrame = rcRefFrameList0[iRefIdxTest]; //读取参考帧  // 根据当前块的ABC块得到MVP 存入cMvPred[0][iRefIdxTest]  rpcMbTempData->getMbDataAccess().getMvPredictor ( cMvPred[0][iRefIdxTest], iRefIdxTest, LIST\_0 );  m\_pcMotionEstimation->estimateBlockWithStart(…) // 运动估计出mv  //注意：这里**MVP**一定是周围块得到的MVP，**搜索起点**可能是MVP，也可能是BL的mv(当BL的ref\_idx与当前的ref\_idx相等时就以BL的Mv为搜索起点，但是MVP仍是周围块得到的)  //如果这个ref\_idx与BL的相同，则  **MotionPrediction**  m\_pcMotionEstimation->estimateBlockWithStart(…) // 运动估计出mv  //注意：这里MVP是BL的mv，搜索起点可能是BL的mv  //如果cost比前面的最小cost(org-ref的失真 和Mv与ref\_idx的比特数)  选用该ref\_idx和mv  //如果是List1预测，还要补偿出双向预测用的块  m\_pcMotionEstimation->compensateBlock ( &cYuvMbBuffer[1], PART\_16x16, MODE\_16x16 ) |

//如果是B帧，还要进行迭代的Bi-Pred，步骤与上面相同

//注意：MVP用对应LIST的MVP，起始MV也用上次对应LIST运动估计出的MV，包括Std的ME和BL的ME

//保存ref\_idx、mv和mvd

rpcMbTempData->getMbMotionData( LIST\_0 ).setRefIdx ( iRefIdx [0] );

rpcMbTempData->getMbMotionData( LIST\_0 ).setAllMv ( cMv [0] );

rpcMbTempData->getMbMvdData ( LIST\_0 ).setAllMv ( cMvd [0] );

//设置每个4x4块的MotionPredFlag

rpcMbTempData->getMbMotionData( LIST\_0 ).setMotPredFlag( bBLPred [0] );

rpcMbTempData->getMbMotionData( LIST\_1 ).setMotPredFlag( bBLPred [1] );

//计算4x4变换的RDCost

xSetRdCostInterMb(\*rpcMbTempData, pcMbDataAccessBase, rcRefListStruct, uiMinQP, uiMaxQP, bLowComplexMbEnable ) 后面分析

xCheckBestEstimation( rpcMbTempData, rpcMbBestData )

//如果当前4x4变换的RdCost比BestData的RdCost大，则函数退出

//交换rpcMbTempData和 rpcMbBestData

xCheckInterMbMode8x8( rpcMbTempData, rpcMbBestData, pcMbRefData, rcRefListStruct, uiMinQP, uiMaxQP,

false, pcMbDataAccessBase )

//计算当前8x8变换的RdCost

xSetRdCost8x8InterMb( \*rpcMbTempData, pcMbDataAccessBaseMotion, rcRefListStruct,

uiMinQP, uiMaxQP, bBLSkip, 0, pcBaseLayerRec, pcBaseLayerResidual )

xCheckBestEstimation( rpcMbTempData, rpcMbBestData )

ErrVal

## MbEncoder::xEstimateMbIntra16( IntMbTempData\*& rpcMbTempData,

IntMbTempData\*& rpcMbBestData,

UInt uiQp,

Bool bBSlice,

Bool bBLSkip

) *在xEstimateIntraBL()里会调用这个函数，且bBLSkip = true(只有这种情况为true，这个时候一定是TCoeff = 1，反过来不成立，因为可能TCoeff = 1，但是BLSkip不是true)*

rpcMbTempData->clear(); //清空MbData 、YuvMbBuffer、 MbDataStruct 、CostData 、MbTransformCoeffs数据

//读取预测需要的样点 从m\_pcIntPicBuffer (pic)= > rpcMbTempData (mb)

rpcMbTempData->loadIntraPredictors( m\_pcIntPicBuffer );

*如果是BLSkip且avcRewrite：读取BaseLayer的预测模式，下面的遍历只用这一种模式*

遍历四种INTRA16预测模式

//进行INTRA预测：

m\_pcIntraPrediction->predictSLumaMb( rpcMbTempData, n, bValid )

//进行变换/量化/反量化/反变换：

m\_pcTransform->transformMb16x16（）

//计算distortion：

m\_pcXDistortion->getLum16x16( pPel, iStride );

*如果是BLSkip且avcRewrite：m\_pcTransform->predictMb16x16()*

*xScanLumaBlock()*

//用Uvlc进行熵编码：

xScanLumaIntra16x16()

//计算RD：

m\_pcRateDistortionIf->getCost( uiBits, uiDist )

//根据上面选择的模式进行INTRA16预测：

m\_pcIntraPrediction->predictSLumaMb()

rpcMbTempData->getTempYuvMbBuffer().loadLuma( \*rpcMbTempData ); //保存Intra预测值

//进行变换/量化/反量化/反变换：系数存在rpcMbTempData->get( B4x4Idx(0) ) （TCoeff m\_aaiLevel）

m\_pcTransform->transformMb16x16（）

*如果是BLSkip且avcRewrite：m\_pcTransform->predictMb16x16()*

*计算新CBP*

//计算LUMA 的 RD：

m\_pcXDistortion->getLum16x16( pPel, iStride, rcDFunc );

m\_pcMotionEstimation->getRateCost( uiBestBits + uiMBits, rcDFunc == DF\_SAD )

//编码色度部分：

xEncodeChromaIntra()

*如果是BLSkip且avcRewrite：MbMode = INTRA\_BL*

否则，是INTRA16， INTRA\_4x4+1 +0/1/2/3代表四种INTRA预测还要加上CBP等构成MbMode

*avcRewrite（INTRA\_BL或者普通INTRA）和不用avcRewrite的CBP要计算*

MotionPred = false ResidualPred = false BaseMode与BLSkip有关

//计算总RD ：

xSetRdCostIntraMb（）

xCheckBestEstimation( rpcMbTempData, rpcMbBestData )

ErrVal

## MbEncoder::xEstimateMbIntraBL( IntMbTempData\*& rpcMbTempData,

IntMbTempData\*& rpcMbBestData,

UInt uiMinQP,

UInt uiMaxQP,

const Frame\* pcBaseLayerRec,

Bool bBSlice,

MbDataAccess\* pcMbDataAccessBase )

rpcMbTempData->clear(); // 清空数据

rpcMbTempData->setMbMode( INTRA\_BL ); //MbMode

rpcMbTempData->setBLSkipFlag( bBLSkip ) //只要BL是intra

rpcMbTempData->setTransformSize8x8( false );

rpcMbTempData->setResidualPredFlag( false );

|  |  |
| --- | --- |
| TCoeffLevelPredictionFlag = 1 （**SliceHeader**）  Based on the MbMode of BL  MODE\_PCM  --xEstimateMbPCMRewrite  INTRA\_4x4 (TransformSize8x8 decide INTRA\_4x4 or INTRA\_8x8)  --xEstimateMbIntra4 | xEstimateMbIntra8  others( INTRA\_16x16)  --xEstimateMbIntra16  Return | |
| SCoeffResidualPredFlag = 1（**SliceHeader**）（MGS CGS）  **rcYuvMbBuffer**拷贝BaseLayer的预测数据m\_sPred  (就是BL的intra预测样点====最底层BL的intra重建) | Others （**Spatial**）  **rcYuvMbBuffer**拷贝pcBaseLayerRec->getFullPelYuvBuffer()  BL的m\_Subband |
| rcTempYuvMbBuffer.loadLuma = rcYuvMbBuffer 都是BL的样点值 | |
| 对每个luma 8x8块 的每个4x4块  --xEncode4x4InterBlock ()     |  |  | | --- | --- | | isSCoeffPred() | else | | transform4x4BlkCGS ()  对当前4x4变换系数和baselayer反量化系数进行做差；对差值进行量化、反量化；得到的值加上baselayer反量化系数得到该层反量化值；最后进行反变换重建 | transform4x4Blk ()  正常的4x4变换 重建 | | isTCoeffPred() | | | predict4x4Blk (当前系数，baselayer系数) ---------接着上面else继续的  当前块4x4变换量化后系数和baselayer量化后系数进行做差（直接在变换域预测）  用Uvlc编码计算bits | |   //对chroma  --xEncodeChromaTexture ()  流程与Luma几乎相同  //计算RD  --xSetRdCostIntraMb ()  xCheckBestEstimation ( rpcMbTempData, rpcMbBestData )  //用8x8变换再计算一次  --xEstimateMbIntraBL8x8 () | |

INTRA\_BL：在不rewrite的情况下

预测值拷贝BL的预测值（分辨率相同）/BL重建值（分辨率不同）

**注意：**

**INTRA\_BL不可能用residual prediction和motion prediction**

**因此函数的bCheckWithResPred = 0，SCoeff = 0（虽然SH里面为1）**

**那么SH的SCoeff = 1有什么作用呢？**

**---------作用是用BL的预测值作为该EL层的预测值！！否则用BL的重建值**

**进行变换时**

|  |
| --- |
| **EL** transform4x4BlkCGS |
| **EL** transform4x4BlkCGS |
| **EL** transform4x4Blk |
| **BL** transform4x4Blk |

ErrVal

## MbEncoder::xEstimateMbBLSkip( IntMbTempData\*& rpcMbTempData,

IntMbTempData\*& rpcMbBestData,

RefListStruct& rcRefListStruct,

UInt uiMinQP,

UInt uiMaxQP,

const Frame\* pcBaseLayerRec, // 参考层的重建(去块滤波之前)

UInt uiMaxNumMv,

Bool bBiPred8x8Disable,

Bool bBSlice,

MbDataAccess\* pcMbDataAccessBase,

MbDataAccess& rcMbDataAccess,

Bool bResidualPred,

const YuvMbBuffer\* pcBLResidual ) //传入cBaseLayerBuffer

*参考层是INTER继续下去，否则，如果当前宏块是InCropWindow，进入xEstimateMbIntraBL()*

rpcMbTempData->clear ();

// 直接copy motion （ref\_idx） 也copy了mbMode 和分区方式

rpcMbTempData->copyMotion ( pcMbDataAccessBase->getMbData() );

//设置BLSkipFlag = true，ResidualPredFlag(可true可false

//Mvd清零

rpcMbTempData->getMbMvdData ( LIST\_0 ).setAllMv( Mv::ZeroMv() );

//计算RD

xSetRdCostInterMb()

xCheckBestEstimation( rpcMbTempData, rpcMbBestData ) );

xCheckInterMbMode8x8( rpcMbTempData, rpcMbBestData, pcMbRefData, rcRefListStruct, uiMinQP, uiMaxQP, true,

pcMbDataAccessBase, (Frame\*)pcBaseLayerRec, pcBLResidual )

计算RD的函数调用参数

|  |  |
| --- | --- |
| xEstimateMbBLSkip | xSetRdCostInterMb( \*rpcMbTempData, pcMbDataAccessBase, rcRefListStruct, uiMinQP, uiMaxQP,  bLowComplexMbEnable, true, 0, (Frame\*)pcBaseLayerRec, pcBLResidual(就是cBaseLayerBuffer) ) |
| xEstimateMb16x16 | xSetRdCostInterMb( \*rpcMbTempData, pcMbDataAccessBase, rcRefListStruct, uiMinQP, uiMaxQP,  bLowComplexMbEnable ) |
| xEstimateMbIntraBL | xSetRdCostIntraMb ( \*rpcMbTempData, uiCoeffBits, bBLSkip ) |
| xEstimateMbIntra16 | xSetRdCostIntraMb ( \*rpcMbTempData, uiBestBits, false ) |

ErrVal

## MbEncoder::xSetRdCostInterMb( IntMbTempData& rcMbTempData,

MbDataAccess\* pcMbDataAccessBase,

RefListStruct& rcRefListStruct,

UInt uiMinQP,

UInt uiMaxQP,

Bool bLowComplexity, // JVT-V079

Bool bBLSkip,

UInt uiAdditionalBits,

Frame\* pcBaseLayerRec,

const YuvMbBuffer\* pcBaseLayerResidual ) // cBaseLayerBuffer

参数

|  |  |  |
| --- | --- | --- |
| cTempPredBuffer |  | 空->MC预测值 |
| rcYuvMbBuffer | rcMbTempData | 空🡪MC预测值🡪MC重建值 |
| rcTempYuvMbBuffer | rcMbTempData .getTempYuvMbBuffer (); | RC的预测值 |
| rcTempBLSkipBaseRes | rcMbTempData .getTempBLSkipResBuffer () | 一开始为空，后来是cBaseLayerBuffer的更新后的值 |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 设置FwdBwd----------每个8x8块的LIST0/1参考情况  get prediction signal (for residual coding)---------运动补偿MC  读取MC的list  compensateMb()/compensateSubMb()： //补偿出预测宏块  xGetMbPredData() //读取ref\_idx/MV/MVD/参考帧buffer到变量cMC8x8D中  xPredLuma() xPredChroma() //根据mv和ref\_idx得到预测的宏块🡺 rcYuvMbBuffer  //BL的intra预测，该层用base mode 进行补偿出预测宏块  m\_pcMotionEstimation->compensateMbBLSkipIntra( rcMbDataAccess, &rcYuvMbBuffer, pcBaseLayerRec )  rcYuvMbBuffer（预测的宏块）赋值给cTempPredBuffer（MC的预测值）  get prediction signal (for reconstruction)---------重建RC  如果RC和MC队列不同，读取RC的list  compensateMb()/compensateSubMb()/compensateMbBLSkipIntra()，  预测值🡪 rcTempYuvMbBuffer（RC的预测值）  （如果MC和RC队列相同，rcTempYuvMbBuffer== rcYuvMbBuffer）  //是BaseMode且Residual Pred  m\_pcMotionEstimation->updateMbBLSkipResidual( rcMbDataAccess, rcTempBLSkipBaseRes )  // rcTempBLSkipBaseRes是pcBaseLayerResidual，对该值进行更新  rcYuvMbBuffer + = rcTempBLSkipBaseRes – pcBaseLayerResidual  *(如果BL是用的预测rcYuvMbBuffer=0；或者INTER的BaseMode，cBaseLayerBuffer=0)*  cTempPredBuffer和rcTempYuvMbBuffer都是rcYuvMbBuffer(MC预测值)  *注意：rcYuvMbBuffer是rpcMbTempData的宏块数据*  对每个8x8块的每个4x4块，进行变换   |  |  | | --- | --- | | SCoeff | 普通 | | transform4x4BlkCGS  1 (原始-MC预测)的差值进行变换  2 新变换系数 =前面的变换系数 – 参考层反量化后系数 就是变换系数差  3 新变换系数进行量化反量化  4 新反量化后的值 = 参考层反量化后 + 变换系数差量化反量化后  5 对新反量化后的值进行反变换  **变换域—变换系数的差！！！！**  **传出去的是变换系数的差，重建的仍是完整的块** | transform4x4Blk  1 (原始-MC预测)的差值进行变换  2 量化和反量化  3 反变换 | | TCoeff：SVC to AVC rewrite | |   **反变换后的残差会与预测值相加进行重建！！！**  CAVLC熵编码，计算Bits  //保存最佳重建的宏块  rcMbTempData.getMbTCoeffs ().copyFrom( cBestCoeffs );  rcMbTempData.loadLuma ( cBestRec );  rcMbTempData.loadChroma ( cBestRec );  //如果RC和MC队列不同 原来rcYuvMbBuffer是MC的，现在改成了RC的  rcYuvMbBuffer.subtract( cTempPredBuffer );  rcYuvMbBuffer.add ( rcTempYuvMbBuffer ); |

# NalUnit比特流写入

|  |
| --- |
| LayerEncoder::xEncodeLayerRepresentation()  传入  ExtBinDataAccessorList& rcOutExtBinDataAccessorList // 输出的NalUnit 每个由一个BinDataAccessor表示  声明  ExtBinDataAccessorList cTmpExtBinDataAccessorList; // 给 closeAndAppendNalUnits 函数用的list  ExtBinDataAccessorList acExtBinDataAccessorList[16]; // 每个 MGS层用的 list  // PrefixNal 用的是 acExtBinDataAccessorList[0]  PicOutputDataList acPicOutputDataList [16];  LayerEncoder::m\_cBinData 每次使用一个ExtBinDataAccessor，使用 m\_pucWriteBuffer暂存数据  LayerEncoder::m\_pcNalUnitEncoder 每次与对应LayerEncoder使用同一个ExtBinDataAccessor  BitWriteBuffer  UInt m\_uiDWordsLeft; // =（ m\_uiInitPacketLength+3）/ 4 剩余能写入的双字  UInt m\_uiBitsWritten; //记录写了多少bit  Int m\_iValidBits; //1～32的可写空间  UInt m\_ulCurrentBits; //当前写入的32比特  UInt\* m\_pulStreamPacket; //保存写的比特流 \*\*\*\*\* 实际写入处 由NalUnitEncoder成员  //m\_pcHeaderSymbolWriteIf调用write函数写入  UInt m\_uiInitPacketLength; //初始化开辟的缓存长度  LayerEncoder  m\_pucWriteBuffer //刚开始写入数据的缓存 与m\_BinData关联  m\_cBinData  m\_pcNalUnitEncoder // 当前调用的 NalUnitEncoder  m\_cExtBinDataAccessor // 当前使用的 BinDataAccessor  NalUnitEncoder  Bool m\_bIsUnitActive; //是否可用  BitWriteBuffer\* m\_pcBitWriteBuffer; //真正的packet写在此处的m\_pulStreamPacket  // 它用来调用initPacket（）  //在closeNalUnit()的xWriteTrailingBits()中直接写  //在MbCoder里有它  HeaderSymbolWriteIf\* m\_pcHeaderSymbolWriteIf; //用来召唤UvlcWriter和CabacWriter写比特流  //它有成员 BitWriteBufferIf\* m\_pcBitWriteBufferIf;  HeaderSymbolWriteIf\* m\_pcHeaderSymbolTestIf;  BinDataAccessor\* m\_pcBinDataAccessor; //当前使用的 BinDataAccessor  UChar\* m\_pucBuffer; //m\_pcBinDataAccessor的data() Payload  //由RBSP(m\_pucTempBuffer)加0x03得到  UChar\* m\_pucTempBuffer; //临时写入的缓存  // 就是BitWriteBuffer 的 m\_pulStreamPacket  UChar\* m\_pucTempBufferBackup;  UInt m\_uiPacketLength; //开辟的缓存长度  NalUnitType m\_eNalUnitType; //nal\_unit\_type  NalRefIdc m\_eNalRefIdc; //nal\_ref\_idc  MbCoder  MbSymbolWriteIf\* m\_pcMbSymbolWriteIf; //宏块级 cabac  RateDistortionIf\* m\_pcRateDistortionIf;  BitWriteBuffer\* m\_pcBitWriteBufferCabac;  BitWriteBuffer\* m\_pcBitWriteBufferUvlc; |