**景嘉微杯**

**图像压缩/解压模型设计文档**

|  |  |  |
| --- | --- | --- |
| **队员** | **：** | **万季风、陈渡、刘冰** |
| **日期** | **：** | **2023年5月30日** |

目 录

[目 录 I](#_Toc136350968)

[1 模型设计 1](#_Toc136350969)

[1.1 模型架构 1](#_Toc136350970)

[1.2 块格式 2](#_Toc136350971)

[2 算法设计 3](#_Toc136350972)

[2.1 颜色变换 3](#_Toc136350973)

[2.2 JPEG-LS 3](#_Toc136350974)

[2.3 LZMA 3](#_Toc136350975)

[2.4 混合算法 3](#_Toc136350976)

[3 实现函数说明 5](#_Toc136350977)

[3.1 算法混合器 5](#_Toc136350978)

[3.2 算法容器 5](#_Toc136350979)

[3.3 颜色变换器 7](#_Toc136350980)

[3.4 JPEG-LS压缩器 9](#_Toc136350981)

[3.5 LZMA 10](#_Toc136350982)

[4 测试结果 12](#_Toc136350983)

[4.1 在单一算法下的压缩率 12](#_Toc136350984)

[4.2 在混合算法下的结果 14](#_Toc136350985)

# 模型设计

## 模型架构

本模式采用颜色变换、JPEG-LS和LZMA算法混合执行，取长补短，尽可能地提升压缩率。压缩模型的设计架构如图1-1所示，未压缩的图像块经过颜色变化器（含YUV、颜色变换1、颜色变换2）、JPEG-LS压缩器、LZMA压缩器，最后进入算法混合器，选取最优算法。最后将结果送入算法容器中储存，作为最终的压缩图像块。

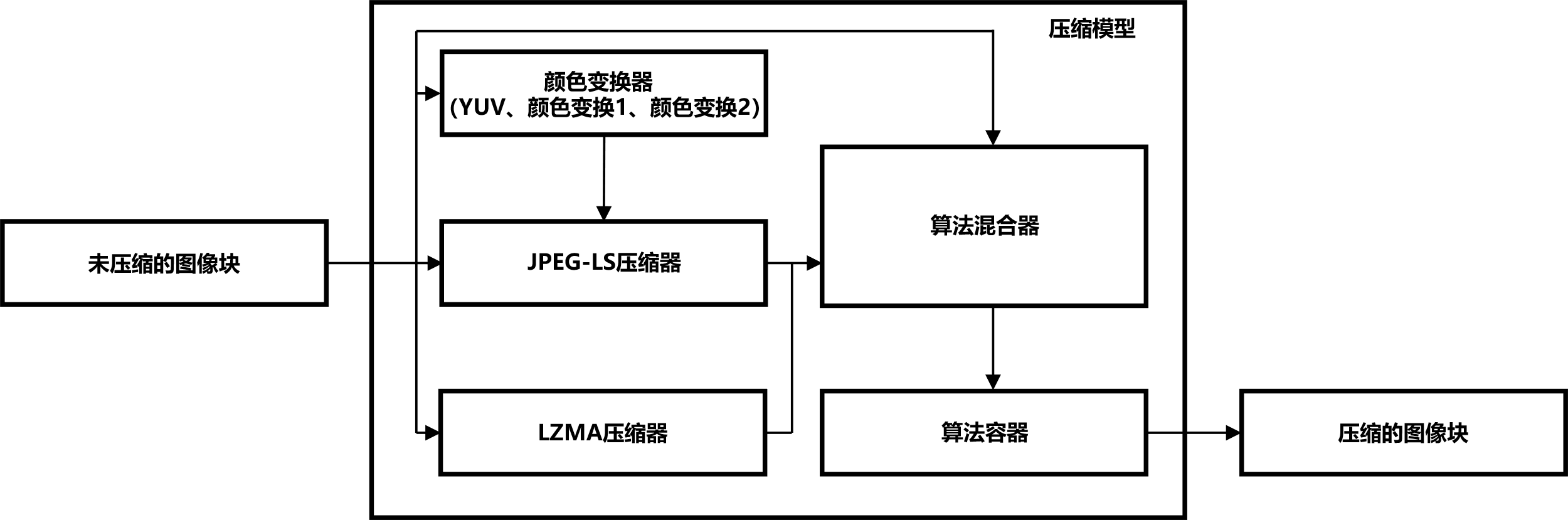


图 1‑1压缩模型架构

如图1-2所示，在算法混合器中包含了7中算法，分别为算法1（原块储存）、算法2（普通的 JPEG-LS）、算法3（LZMA）、算法4（空白块）、算法5（YUV变换的JPEG-LS）、算法6（颜色变换1的 JPEG-LS）、算法7（颜色变换2的 JPEG-LS）、并包含了一个混合器将这些算法混合。

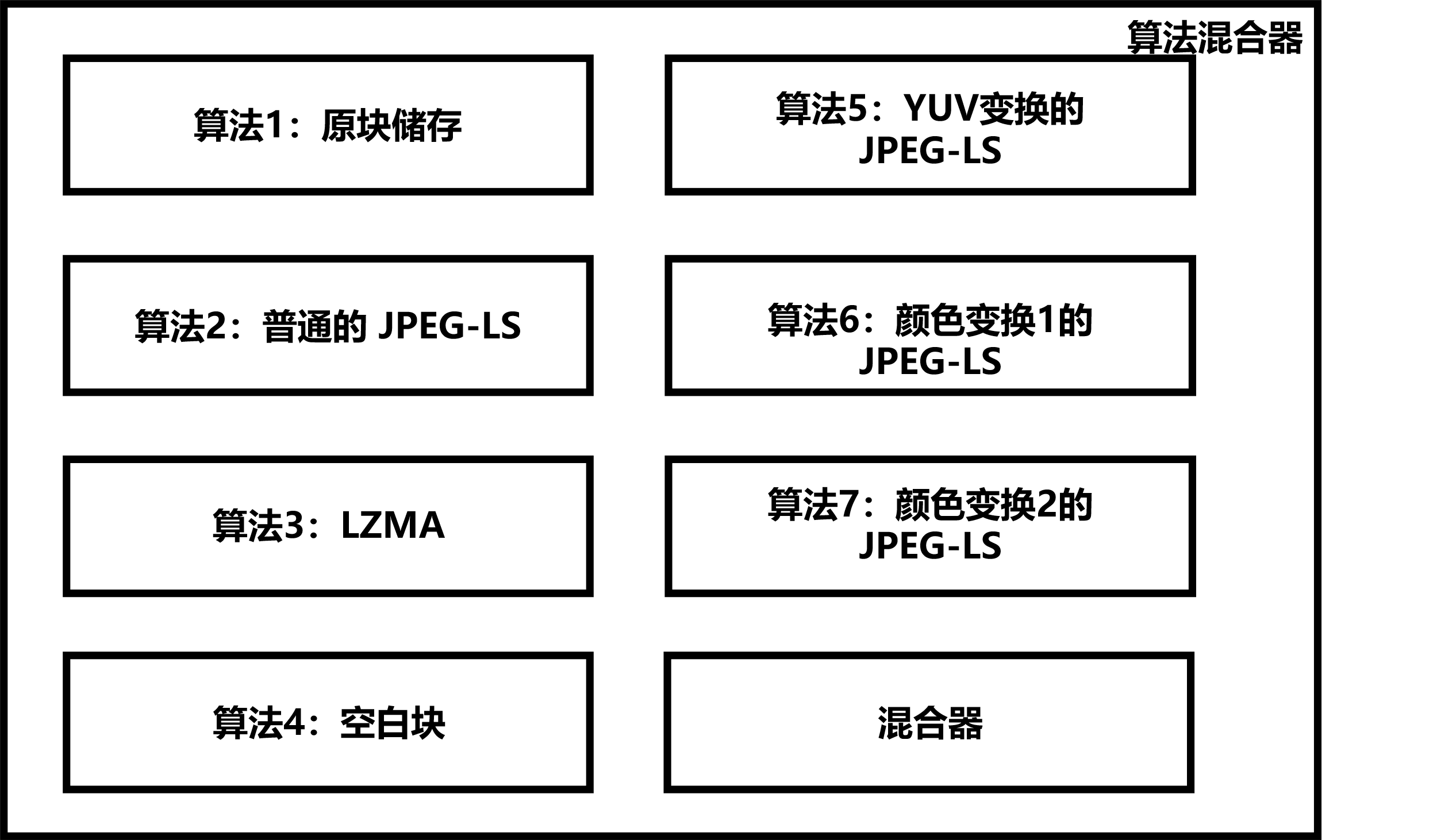


图 1‑2压缩模型架构

解压模型与压缩模型类似，近乎于是压缩的逆过程，如图1-3所示。压缩后的图像块会首先被算法容器解析器解析，随后送入算法混合器中执行对应解压算法，最终得到原始的图像块。

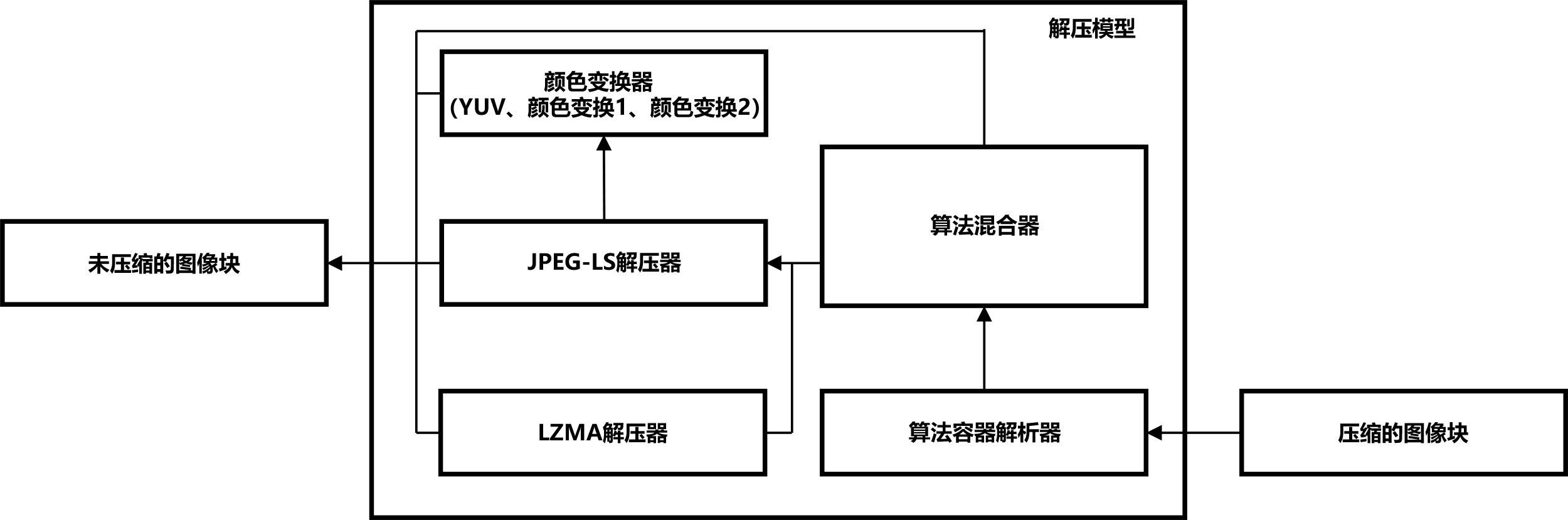


图 1‑3解压模型架构

## 块格式

为了能够正常读取算法容器中的特定参数，设计了如图1-4所示的块格式用于实际储存压缩块。块的长度是由主办方给出的图像参数得到的。整个块的长度不超过256。若块的长度小于256，则块的开头第一个字节为算法标识符，随后跟着不定长的对应算法压缩块，若块的长度为256则按原始图像储存，即算法1。

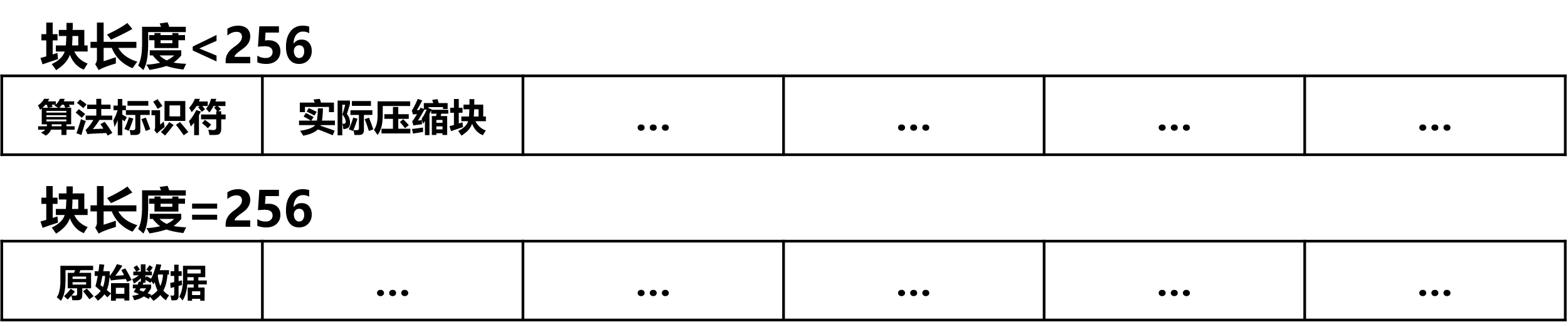


图 1‑4 块格式

# 算法设计

## 颜色变换

### YUV变换

### 颜色变换1

### 颜色变换2

## JPEG-LS

## LZMA

## 混合算法

### 算法1

即不作任何压缩，直接储存原始图像块。

### 算法2

即普通JPEG-LS，直接调用JPEG-LS算法压缩。

### 算法3

即LZMA，直接调用LZMA算法压缩。

### 算法4

压缩空白块，若图像块为空白则标识为空白块。

### 算法5

即先进行YUV变换再JPEG-LS压缩，虽然YUV变换不会压缩任何数据，但得益于YUV变换，JPEG-LS的压缩大小可能更小。

### 算法6

即先进行颜色变换1再JPEG-LS压缩，虽然颜色变换1不会压缩任何数据，但得益于颜色变换1，JPEG-LS的压缩大小可能更小。

### 算法7

即先进行颜色变换2再JPEG-LS压缩，虽然颜色变换2不会压缩任何数据，但得益于颜色变换2，JPEG-LS的压缩大小可能更小。

# 实现函数说明

## 算法混合器

算法混合器主要实现在compressAlgorithm中。在算法混合器中，每一个算法都储存在一个结构体中，这个结构体中包含两个函数，一个用于解压（decompress），一个用于压缩（compress），参数列表为输入流指针、输入流的长度、输出流指针、输出流的长度。对于压缩而言输入流就为待压缩的内容，输出流就为已压缩的内容，对于解压输入流就为已压缩的内容，输出流就为未压缩的内容。

typedef struct {

void (\*compress)(const uint8\_t \*inputStream, const int32\_t inputStreamLength, uint8\_t \*outputStream, int32\_t \*outputStreamLength);

void (\*decompress)(const uint8\_t \*inputStream, const int32\_t inputStreamLength,uint8\_t \*outputStream, int32\_t \*outputStreamLength);

} algorithmTypeDef;

通过这个结构体生成了算法混合器的核心算法表，在算法表中储存着算法混合器的所有算法。

algorithmTypeDef algorithmTable[] = {

{NULL, NULL},

{charlsCompress, charlsDecompress},

{lzipCompress, lzipDecompress},

{blankCompress, blankDecompress},

{charlsAndYUVCompress, charlsAndYUVDecompress},

{charlsAndColor1Compress, charlsAndColor1Decompress},

{charlsAndColor2Compress, charlsAndColor2Decompress}

};

## 算法容器

算法容器在rgbTileProc中实现，实际实现了argb2tile、tile2argb两个函数。对于压缩，会首先执行所有的压缩算法，并得到压缩率最高的算法。若最大压缩大小小于254，则在输出的第一个字节写入算法标识符，并在之后写入算法压缩后的结果，若最大压缩结果大于254则按原块储存不做压缩，即算法1。

int argb2tile(const unsigned char\* pClrBlk, unsigned char\* pTile, int\* pTileSize)

{

uint8\_t tempStream[64 \* 4 \* 4 \* 2];

int32\_t tempStreamLength;

int32\_t minSize = g\_nTileWidth \* g\_nTileHeight \* 4 - 1;

uint8\_t minType = 0;

for(uint8\_t type = 1; type < 7; type++)

{

algorithmTable[type].compress((unsigned char\*)pClrBlk, g\_nTileWidth \* g\_nTileHeight \* 4, tempStream, &tempStreamLength);

if(tempStreamLength < minSize)

{

minSize = tempStreamLength;

minType = type;

}

printf("%3d,", tempStreamLength);

}

if(minType == 0)

{

\*pTileSize = g\_nTileWidth \* g\_nTileHeight \* 4;

memcpy(pTile, pClrBlk, \*pTileSize);

}

else

{

\*pTile = minType;

algorithmTable[minType].compress((unsigned char\*)pClrBlk, g\_nTileWidth \* g\_nTileHeight \* 4, pTile + 1, pTileSize);

\*pTileSize += 1;

}

printf("%2d,%3d\n", minType, minSize);

return 0;

}

对于解压，首先判断长度是否为256，若为256则说明这个块是按原始储存的，若不为256则读取算法标识符，并执行对应的解压算法，最终将结果储存在输出中。

int tile2argb(const unsigned char\* pTile, int nTileSize, unsigned char\* pClrBlk)

{

uint8\_t type = \*pTile;

if(nTileSize >= 256)

{

memcpy(pClrBlk, pTile, nTileSize);

}

else

{

algorithmTable[type].decompress(pTile + 1, nTileSize - 1, (unsigned char\*)pClrBlk, &nClrBlkSize);

}

## 颜色变换器

颜色变换器主要实现在YUVConvert中。颜色变换器内置核心变换有三种，分别为YUV、颜色变换1，颜色变换2。这些核心变换均在通用颜色变换器中生成。通过指定colorTransform函数指针以使用不同的核心变换，用颜色变换器会不停调用变换核心来完成整个流的颜色变换。

void colorTransform(const uint8\_t \*inputStream, const int32\_t inputStreamLength, uint8\_t \*outputStream, int32\_t \*outputStreamLength, void(\*colorTransform)(BGRAStructDef \*, VUYAStructDef \*));

void colorInverseTransform(const uint8\_t \*inputStream, const int32\_t inputStreamLength, uint8\_t \*outputStream, int32\_t \*outputStreamLength, void(\*colorInverseTransform)(VUYAStructDef \*, BGRAStructDef \*));

### YUV变换核心

YUV变换实现如下，具体原理参见算法设计一章：

void YUVTransform(BGRAStructDef \*BGRA, VUYAStructDef \*VUYA)

{

VUYA->V = BGRA->B - BGRA->G + 128;

VUYA->U = BGRA->R - BGRA->G + 128;

VUYA->Y = BGRA->G + (VUYA->V + VUYA->U) / 4 - 64;

VUYA->A = BGRA->A;

}

YUV逆变换实现如下：

void YUVInverseTransform(VUYAStructDef \*VUYA, BGRAStructDef \*BGRA)

{

BGRA->G = VUYA->Y - (VUYA->U + VUYA->V) / 4 + 64;

BGRA->R = VUYA->U + BGRA->G - 128;

BGRA->B = VUYA->V + BGRA->G - 128;

BGRA->A = VUYA->A;

}

### 颜色变换1核心

与YUV变换类似，颜色变换1实现如下，具体原理参见算法设计一章：

void colorTransform1(BGRAStructDef \*BGRA, VUYAStructDef \*VUYA)

{

VUYA->V = BGRA->G;

VUYA->U = BGRA->B - BGRA->G + 128;

VUYA->Y = BGRA->R - BGRA->G + 128;

VUYA->A = BGRA->A;

}

颜色变换1实现如下：

void colorInverseTransform1(VUYAStructDef \*VUYA, BGRAStructDef \*BGRA)

{

BGRA->G = VUYA->V;

BGRA->R = VUYA->Y + VUYA->V - 128;

BGRA->B = VUYA->U + VUYA->V - 128;

BGRA->A = VUYA->A;

}

### 颜色变换2核心

与YUV变换类似，颜色变换2实现如下，具体原理参见算法设计一章：

void colorTransform2(BGRAStructDef \*BGRA, VUYAStructDef \*VUYA)

{

VUYA->V = BGRA->G;

VUYA->U = BGRA->B - (BGRA->R + BGRA->G) / 2 - 128;

VUYA->Y = BGRA->R - BGRA->G + 128;

VUYA->A = BGRA->A;

}

颜色变换2实现如下：

void colorInverseTransform2(VUYAStructDef \*VUYA, BGRAStructDef \*BGRA)

{

BGRA->G = VUYA->V;

BGRA->R = VUYA->Y + VUYA->V - 128;

BGRA->B = VUYA->U + (BGRA->R + VUYA->V) / 2 - 128;

BGRA->A = VUYA->A;

}

## JPEG-LS压缩器

JPEG-LS的实现在charls文件夹下，并在charlsWapper中提供了一个标准接口

void charlsCompress(const uint8\_t \*inputStream, int32\_t inputStreamLength,uint8\_t \*outputStream, int32\_t \*outputStreamLength);

void charlsDecompress(const uint8\_t \*inputStream, int32\_t inputStreamLength,uint8\_t \*outputStream, int32\_t \*outputStreamLength);

对于JPEG-LS的压缩部分，首先创建一个JPEG-LS压缩器，并对这个压缩器进行配置。将压缩器设置为sample模式，也就是按BGRABGRA…格式排列，将压缩模式设置为无损。最后将数据传入，进行压缩。

void charlsCompress(const uint8\_t \*inputStream, const int32\_t inputStreamLength, uint8\_t \*outputStream, int32\_t \*outputStreamLength)

{

charls::jpegls\_encoder encoder;

encoder.frame\_info({ 8, 8, 8, 4 })

.interleave\_mode(charls::interleave\_mode::sample)

.near\_lossless(0);

std::vector<uint8\_t> encoded\_buffer(encoder.estimated\_destination\_size());

encoder.destination(encoded\_buffer);

size\_t encoded\_size;

encoded\_size = encoder.encode(inputStream, 4);

encoded\_buffer.resize(encoded\_size);

\*outputStreamLength = encoded\_buffer.size() - CHARLS\_HEAD\_LENGTH - CHARLS\_TAIL\_LENGTH;

memcpy(outputStream, (uint8\_t \*)encoded\_buffer.data() + CHARLS\_HEAD\_LENGTH, \*outputStreamLength);

}

对于JPEG-LS解压部分，首先会将对应的头尾参数加入，送入解析器中一并解析，并解压得到最终结果。

void charlsDecompress(const uint8\_t \*inputStream, const int32\_t inputStreamLength, uint8\_t \*outputStream, int32\_t \*outputStreamLength)

{

uint8\_t tempBuffer[500];

memcpy(tempBuffer, charlsHead, CHARLS\_HEAD\_LENGTH);

memcpy(tempBuffer + CHARLS\_HEAD\_LENGTH, inputStream, inputStreamLength);

memcpy(tempBuffer + CHARLS\_HEAD\_LENGTH + inputStreamLength, charlsTail, CHARLS\_TAIL\_LENGTH);

charls::jpegls\_decoder decoder{ tempBuffer , (size\_t)(CHARLS\_HEAD\_LENGTH + inputStreamLength + CHARLS\_TAIL\_LENGTH) , true };

auto buffer = decoder.decode<std::vector<uint8\_t>>();

memcpy(outputStream, (void\*)buffer.data(), (size\_t)buffer.size());

\*outputStreamLength = buffer.size();

}

## LZMA

LZMA的实现在lzip文件夹下，并在lzipWapper中提供了一个标准接口。

void lzipCompress(const uint8\_t \*inputStream, int32\_t inputStreamLength,

uint8\_t \*outputStream, int32\_t \*outputStreamLength);

void lzipDecompress(const uint8\_t \*inputStream, int32\_t inputStreamLength, uint8\_t \*outputStream, int32\_t \*outputStreamLength);

对于LZMA压缩部分，初始化模块后创建一个压缩器，并设置压缩器的参数。为达到最佳压缩效果，字典与匹配长度选择尽量大，并将原始数据传入进行压缩。

void lzipCompress(const uint8\_t \*inputStream, const int32\_t inputStreamLength, uint8\_t \*outputStream, int32\_t \*outputStreamLength)

{

dis\_slots.init();

prob\_prices.init();

Lzma\_options encoder\_options = option\_mapping[6];

LZ\_encoder\_base \*encoder = 0; // polymorphic encoder

encoder = new LZ\_encoder(encoder\_options.dictionary\_size,

encoder\_options.match\_len\_limit, inputStream, inputStreamLength, outputStream, outputStreamLength);

while (true) // encode one member per iteration

{

encoder->encode\_member(0x0008000000000000ULL);

if (encoder->data\_finished())

break;

encoder->reset();

}

delete encoder;

}

对于LZMA解压部分，只需将输入数据传入LZMA解压器即可。

void lzipDecompress(const uint8\_t \*inputStream, const int32\_t inputStreamLength, uint8\_t \*outputStream, int32\_t \*outputStreamLength)

{

Range\_decoder rdec(inputStream, inputStreamLength);

rdec.reset\_member\_position();

LZ\_decoder decoder(rdec, 4096, outputStream, outputStreamLength);

decoder.decode\_member();

}

# 测试结果

## 在单一算法下的压缩率

首先测试了单一算法的表现。

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **样本** | **LZMA** | **JPEG-LS** | **YUV**  **JPEG-LS** | **color1**  **JPEG-LS** | **color2**  **JPEG-LS** |
| sample01.bmp | 7.08% | 5.46% | 8.30% | 8.32% | 8.23% |
| sample02.bmp | 16.75% | 17.19% | 17.84% | 17.44% | 17.52% |
| sample03.bmp | 53.90% | 32.86% | 31.68% | 31.68% | 32.23% |
| sample04.bmp | 53.77% | 38.48% | 32.87% | 32.97% | 32.78% |
| sample05.bmp | 51.70% | 33.35% | 31.29% | 31.41% | 31.36% |
| sample06.bmp | 77.38% | 46.44% | 42.96% | 42.74% | 42.56% |
| sample07.bmp | 74.38% | 43.47% | 38.27% | 38.51% | 38.30% |
| sample08.bmp | 63.43% | 48.45% | 38.54% | 38.65% | 38.55% |
| sample09.bmp | 73.39% | 47.92% | 39.58% | 39.62% | 39.59% |
| sample10.bmp | 35.90% | 25.38% | 24.80% | 24.71% | 24.63% |
| sample11.bmp | 33.66% | 22.41% | 22.17% | 22.16% | 22.16% |
| sample12.bmp | 60.14% | 35.40% | 32.97% | 33.17% | 33.10% |
| sample13.bmp | 65.19% | 46.83% | 38.65% | 38.70% | 38.89% |
| sample14.bmp | 35.14% | 25.38% | 23.19% | 23.30% | 23.17% |
| sample15.bmp | 63.73% | 43.99% | 37.36% | 37.48% | 37.35% |
| sample16.bmp | 37.21% | 25.76% | 23.33% | 23.42% | 23.33% |
| sample17.bmp | 44.48% | 32.52% | 27.92% | 27.95% | 27.83% |
| sample18.bmp | 28.99% | 20.00% | 18.88% | 18.92% | 18.84% |
| sample19.bmp | 49.40% | 37.91% | 30.97% | 31.11% | 31.04% |
| sample20.bmp | 49.54% | 32.48% | 29.25% | 29.25% | 29.12% |
| sample21.bmp | 13.79% | 17.13% | 16.46% | 16.68% | 16.70% |
| sample22.bmp | 9.55% | 10.81% | 11.18% | 10.99% | 11.02% |
| sample23.bmp | 20.60% | 17.32% | 17.36% | 17.09% | 17.03% |
| sample24.bmp | 16.10% | 14.90% | 14.10% | 13.83% | 13.91% |
| sample25.bmp | 60.93% | 40.10% | 33.94% | 33.96% | 33.80% |
| sample26.bmp | 79.57% | 56.15% | 48.63% | 48.59% | 48.39% |
| sample27.bmp | 62.11% | 43.21% | 35.43% | 35.50% | 35.34% |
| sample28.bmp | 54.81% | 39.32% | 39.81% | 39.72% | 40.50% |
| sample29.bmp | 73.39% | 49.94% | 48.08% | 47.99% | 49.15% |
| sample30.bmp | 79.24% | 48.62% | 44.94% | 45.42% | 45.45% |
| sample31.bmp | 73.50% | 52.16% | 49.04% | 49.49% | 49.81% |
| sample32.bmp | 25.06% | 23.09% | 25.79% | 25.30% | 26.27% |
| sample33.bmp | 79.31% | 61.00% | 57.07% | 57.79% | 58.24% |
| **平均压缩率** | **49.19%** | **34.41%** | **31.29%** | **31.33%** | **31.40%** |
| 最大压缩率 | 7.08% | 5.46% | 8.30% | 8.32% | 8.23% |
| 最小压缩率 | 79.57% | 61.00% | 57.07% | 57.79% | 58.24% |

表 4.1在单一算法下的压缩率

## 在混合算法下的结果

随后测试了本模型所使用的混合算法的表现。

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 样本 | sample01 | sample02 | sample03 | sample04 | sample05 |
| 压缩率 | 4.03% | 14.80% | 30.99% | 32.51% | 30.74% |
| 样本 | sample06 | sample07 | sample08 | sample09 | sample10 |
| 压缩率 | 42.45% | 38.18% | 38.21% | 39.31% | 23.61% |
| 样本 | sample11 | sample12 | sample13 | sample14 | sample15 |
| 压缩率 | 21.02% | 32.57% | 38.35% | 22.46% | 37.05% |
| 样本 | sample16 | sample17 | sample18 | sample19 | sample20 |
| 压缩率 | 22.77% | 27.35% | 18.02% | 30.65% | 28.84% |
| 样本 | sample21 | sample22 | sample23 | sample24 | sample25 |
| 压缩率 | 13.55% | 9.03% | 16.04% | 13.15% | 33.68% |
| 样本 | sample26 | sample27 | sample28 | sample29 | sample30 |
| 压缩率 | 47.92% | 35.14% | 37.50% | 47.11% | 44.71% |
| 样本 | sample31 | sample32 | sample33 |  |  |
| 压缩率 | 48.24% | 21.12% | 56.12% |  |  |

表 4.2各图片在混合算法下的压缩率

|  |  |  |
| --- | --- | --- |
| 平均压缩率 | 最大压缩率 | 最小压缩率 |
| 30.22% | 4.03% | 56.12% |

表 4.2在混合算法下的综合压缩率

## 综合比较

最后将各个算法进行比较，可以看到本模型使用的混合算法达到了最大的平均压缩率30.22%。

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **指标** | **LZMA** | **JPEG-LS** | **YUV**  **JPEG-LS** | **color1**  **JPEG-LS** | **color2**  **JPEG-LS** | **混合算法** |
| **平均压缩率** | 49.19% | 34.41% | 31.29% | 31.33% | 31.40% | **30.22%** |
| **最大压缩率** | 7.08% | 5.46% | 8.30% | 8.32% | 8.23% | **4.03%** |
| **最小压缩率** | 79.57% | 61.00% | 57.07% | 57.79% | 58.24% | **56.12%** |

表 4.2各个算法的综合压缩率