访问像素值

- 若要访问矩阵中的每个独立元素，只需要指定它的行号和列号即可。

  1.返回的对应元素可以是单个数值（灰色图像）

  2.也可以是多通道图像的数值向量（彩色图像）

- 椒盐噪声：　椒盐噪声是一个专门的噪声类型，它随机选择一些像素，把它们的颜色替换成白色或黑色。如果通信时出错，部分像素的值在传输时丢失，就会产生这种噪声。

```
//在图像中加入椒盐噪声（salt-and-pepper noise）
void salt(cv::Mat image, int n)
{
    // C++11 的默认随机数生成器
    // 调用默认的构造函数，因此会用默认种子来设置初始状态
    // 在不同时刻执行从generator产生的随机数字序列总是相同的，因为种子保持不变
    // 也可以自己提供种子
    std::default_random_engine generator; // 定义一个默认随机数生成器

    // 离散均匀分布的随机数范围（上边界和下边界）
    // 返回均匀分布在闭合范围 [a, b] 内的随机整数
    std::uniform_int_distribution<int> randomRow(0, image.rows - 1); // 椒盐噪声的y坐标的随机数范围
    std::uniform_int_distribution<int> randomCol(0, image.cols - 1); // 椒盐噪声的x坐标的随机数范围

    int i, j;
    for (int k = 0; k < n; k++) // 产生n个椒盐噪声
    {
        // 随机生成图像位置
        i = randomCol(generator); // 将随机数生成器对象传给随机数范围对象，生成椒盐噪声的x坐标
        j = randomRow(generator); // 生成椒盐噪声的x坐标

        if (image.type() == CV_8UC1) // 如果是灰度图像
        {
            // 单通道8位图像
            image.at<uchar>(j, i) = 255; // 颜色替换成白色，增加椒盐噪声
        }
        else if (image.type() == CV_8UC3)
        {
            // 方式一
            // 3 通道图像，下标0，1，2是通道
            // Vec3b: 含3个无符号字符（uchar）类型的数据，即Vec<uchar, 3>
```

```
            // image.at<cv::Vec3b>(j, i)[0] = 255; // B
            // image.at<cv::Vec3b>(j, i)[1] = 255; // G
            // image.at<cv::Vec3b>(j, i)[2] = 255; // R

            // 方式二
            // 直接使用短向量: 颜色替换成白色，增加椒盐噪声
            image.at<cv::Vec3b>(j, i) = cv::Vec3b(255, 255, 255); // Vec<uchar,
3>
        }
    }
}


int main()
{
#pragma region 访问像素值

    // 读入图像（彩色）
    cv::Mat image = cv::imread("1.JPG", cv::IMREAD_COLOR);

    // 调用函数以添加噪声
    salt(image, 3000); // 白色的像素（噪声）数量为3000个

    // 显示图像
    cv::namedWindow("Image");
    cv::imshow("Image", image);
    cv::waitKey();

#pragma endregion

    return 0;
}
```

cv::Mat_

```
// 用Mat_模板操作图像（cv::Mat -> cv::Mat_）
    cv::Mat_<uchar> img(image); // 将cv::Mat_类的对象img初始化为图像image，数据类型为
uchar

// 或使用引用：
// 创建一个cv::Mat_类的引用，但需要进行数据类型的转换，将cv::Mat类型转换为
cv::Mat_<uchar>&引用类型
    cv::Mat_<uchar>& im2 = reinterpret_cast<cv::Mat_<uchar>&>(image);
```

扫描图像

三种方法：

- 用指针扫描图像
- 用迭代器扫描图像
- 用at方法扫描图像

减色算法

> 单通道：256个元素
>
> 三通道：256×256×256个元素
>
> 假设N 是减色因子，将图像中每个像素的值除以N（这里假定使用整数除法，不保留余数）。然后将结果乘以N，得到N 的倍数，并且刚好不超过原始像素值。加上N / 2，就得到相邻的N 倍数之间的中间值。对所有8 位通道值重复这个过程，就会得到(256 / N) × (256 / N) × (256 / N)种可能的颜色值。

$$\left[\frac{I(x,y)}{N}\right] * N + \frac{N}{2}$$

如果N = 64 $\left[\frac{I(x,y)}{N}\right] * N$ ⟹

[0, 64)	=>	0	= 32
[64, 128)	=>	64	= 96
[128, 192)	=>	128	= 160
[192, 255)	=>	192	= 224

$+ \frac{N}{2}$

用指针扫描图像

```
// 通常创建uchar型指针，用ptr获取指针
    uchar* data = image.ptr(j)
// 三个通道一起处理，真正的列数为cols*channels
```

```
void colorReduce(cv::Mat image, int div = 64) // 在原始图像上修改，div-->减色因子
{
    // 图像的行数
    int nl = image.rows;
    // 每行的元素数量
    int nc = image.cols * image.channels();

    for (int j = 0; j < nl; j++) // 行循环
    {
        // 获取行j的地址
        uchar* data = image.ptr<uchar>(j); // ptr模板方法可以直接访问图像中一行的起始地址

        for (int i = 0; i < nc; i++)
        {
            // 处理每个像素 ------------------------

            // 减色算法，三个通道一起处理，指向数组的指针和数组名用法互通
            data[i] = data[i] / div * div + div / 2; // 三个通道同时被处理

            // 像素处理结束 ------------------------
        } // 一行结束
    }
}
```

用迭代器扫描图像

```
// 迭代器只要获取起始位置和终止位置便可循环遍历一次图像（从begin开始一个元素一个元素遍历）
// 初始位置是右上角，结束位置是左下角
// 处理元素时，需要对每个通道单独处理
```

```
void colorReduce2(cv::Mat image, int div = 64)
{
    // 获取迭代器（两种写法）
    cv::Mat_<cv::Vec3b>::iterator it = image.begin<cv::Vec3b>();     // 使用
cv::Mat_类获取迭代器  开始
    cv::MatIterator_<cv::Vec3b> itend = image.end<cv::Vec3b>();      // 使用
cv::MatIterator_类获取迭代器  结束

    // 不需要知道行数和列数
    for (; it != itend; ++it) // 迭代器就类似于一个指针，遍历整个图像
    {
        // 处理每个像素 ------------------------

        // 减色算法，每个通道单独处理
        // it指针指向三个元素（每个位置的三个通道）
        (*it)[0] = (*it)[0] / div * div + div / 2;
        (*it)[1] = (*it)[1] / div * div + div / 2;
        (*it)[2] = (*it)[2] / div * div + div / 2;

        // 像素处理结束 ------------------------
    }
}
```

用at方法扫描图像

```
// 每个通道单独处理
// 效率低，不推荐
```

```
void colorReduce2(cv::Mat image, int div = 64)
{
    // 获取迭代器（两种写法）
    cv::Mat_<cv::Vec3b>::iterator it = image.begin<cv::Vec3b>();     // 使用
cv::Mat_类获取迭代器  开始
    cv::MatIterator_<cv::Vec3b> itend = image.end<cv::Vec3b>();      // 使用
cv::MatIterator_类获取迭代器  结束

    // 不需要知道行数和列数
    for (; it != itend; ++it) // 迭代器就类似于一个指针，遍历整个图像
    {
        // 处理每个像素 ------------------------

        // 减色算法，每个通道单独处理
        // it指针指向三个元素（每个位置的三个通道）
        (*it)[0] = (*it)[0] / div * div + div / 2;
        (*it)[1] = (*it)[1] / div * div + div / 2;
        (*it)[2] = (*it)[2] / div * div + div / 2;

        // 像素处理结束 ------------------------
    }
}
```

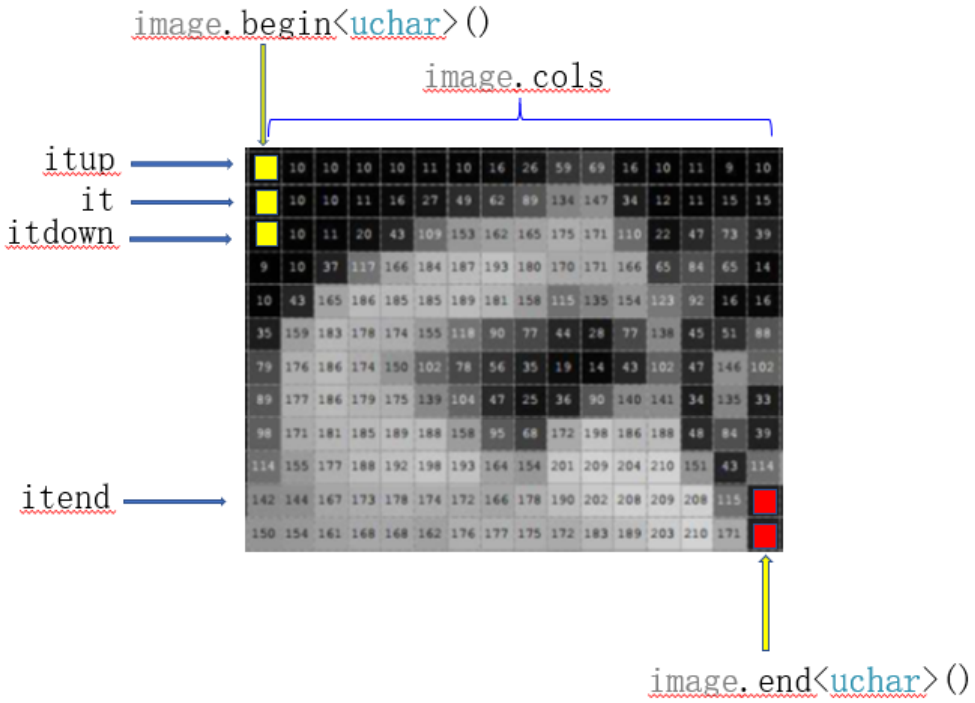
使用输入和输出参数

```
// 使用输入和输出参数
// 有的程序不希望对原始图像进行修改
void colorReduceIO(const cv::Mat& image,     // 输入图像，const表示这幅图像不会在函数中修改
    cv::Mat& result,                         // 输出图像
    int div = 64)
{
    int nl = image.rows;
    int nc = image.cols;
    int nchannels = image.channels();

    // 创建一个新图像，如果新图像大小类型与原图像相同，例如：调用时，colorReduceIO(image, image)
    // 这个方法就不会执行任何操作，也不会修改实例，只是直接返回，相当于在原图上修改
    result.create(image.rows, image.cols, image.type());

    for (int j = 0; j < nl; j++)
    {
        // 获取输入图像的行j的地址
        const uchar* data_in = image.ptr<uchar>(j);
        // 获取输出图像的行j的地址
        uchar* data_out = result.ptr<uchar>(j);

        for (int i = 0; i < nc * nchannels; i++)
        {
            // 处理每个像素 ------------------------

            data_out[i] = data_in[i] / div * div + div / 2;

            // 像素处理结束 ------------------------
        } // 一行结束
    }
}
```

扫描图像并访问相邻像素

锐化图像算法

> - 它基于拉普拉斯算子（将在后面章节讨论）。在图像处理领域有一个众所周知的结论：如果从图像中减去拉普拉斯算子部分，图像的边缘就会放大，因而图像会变得更加尖锐。
> - 可以用以下方法计算锐化的数值：
>
> sharpened_pixel= 5*current – left – right – up - down;
>
> 这里的left/right是与当前像素相邻的左侧/右侧像素，up/down是上一行/下一行的相邻像素



注：

- 不遍历第一行、最后一行，第一列、最后一列，因为 第一行没有上一行，最后一行没有下一行（列同理）
- 使用指针的效率较高
- 计算锐化时，相邻元素时用的是原图像像素的值；如果在原图像上修改，其计算时会用到修改后的值
- 未计算锐化值的点的像素设为0

使用指针

```
void sharpen(const cv::Mat& image, cv::Mat& result)
{
    // 这里不适合调用sharpen(image, image);
    // 因为下面计算锐化时，其相邻元素用的是原始图像像素的值
    // 如果在原始图像上修改，计算时会用到修改后的值，不符合预期
    result.create(image.size(), image.type());

    // 获得通道数
    int nchannels = image.channels();

    // 遍历所有行（除第一行和最后一行）
    // 因为第一行没有上一行，最后一行没有下一行，所以要跳过
    for (int j = 1; j < image.rows - 1; j++)
    {
        const uchar* previous = image.ptr<const uchar>(j - 1);  // 上一行
        const uchar* current = image.ptr<const uchar>(j);       // 当前行
        const uchar* next = image.ptr<const uchar>(j + 1);      // 下一行

        uchar* output = result.ptr<uchar>(j); // 输出图像的当前行

        // 遍历所有列（除第一列和最后一列）
        // 因为第一列没有左边一列，最后一列没有右边一列，所以要跳过
        for (int i = nchannels; i < (image.cols - 1) * nchannels; i++)
        {
```

```
            // 应用锐化算法 sharpened_pixel=5*current-left-right-up-down;
            // cv::saturate_cast<uchar>()函数在类型转换的同时，会把小于0的数值调整为0，
大于255的数值调整为255
            output[i] = cv::saturate_cast<uchar>(5 * current[i]
                - current[i - nchannels]    // left
                - current[i + nchannels]    // right
                - previous[i]               // up
                - next[i]);                 // down
        }
    }

    // 把未处理的像素设为0
    result.row(0).setTo(cv::Scalar(0));                 // 第一行
    result.row(result.rows - 1).setTo(cv::Scalar(0));   // 最后一行
    result.col(0).setTo(cv::Scalar(0));                 // 第一列
    result.col(result.cols - 1).setTo(cv::Scalar(0));   // 最后一列
}
```

使用迭代器

只考虑灰度图

```
void sharpenIterator(const cv::Mat& image, cv::Mat& result)
{
    // 只在灰度图像下工作
    CV_Assert(image.type() == CV_8UC1); // CV_Assert() 若括号中的表达式为false，则返
回一个错误信息，终止程序 执行

    // 初始化迭代器
    cv::Mat_<uchar>::const_iterator it = image.begin<uchar>() + image.cols;
// 当前行从第二行开始
    cv::Mat_<uchar>::const_iterator itend = image.end<uchar>() - image.cols;
// 倒数第二行结束
    cv::Mat_<uchar>::const_iterator itup = image.begin<uchar>();
// 当前行的上一行
    cv::Mat_<uchar>::const_iterator itdown = image.begin<uchar>() + image.cols;
// 当前行的下一行

    // 创建输出图像
    result.create(image.size(), image.type());
    // 初始化输出图像的迭代器
    cv::Mat_<uchar>::iterator itout = result.begin<uchar>() + result.cols;

    // 遍历整个图像
    for (; it != itend; ++it, ++itout, ++itup, ++itdown)
    {
        *itout = cv::saturate_cast<uchar>(*it * 5 - *(it - 1) - *(it + 1) -
*itup - *itdown);
    }

    // 把未处理的像素设为0
    result.row(0).setTo(cv::Scalar(0));                 // 第一行
    result.row(result.rows - 1).setTo(cv::Scalar(0));   // 最后一行
    result.col(0).setTo(cv::Scalar(0));                 // 第一列
    result.col(result.cols - 1).setTo(cv::Scalar(0));   // 最后一列
```

```
}
```

滤波器

```
void sharpen2D(const cv::Mat& image, cv::Mat& result)
{
    // Construct kernel (all entries initialize to 0)
    cv::Mat kernel(3, 3, CV_32F, cv::Scalar(0));
    // assigns kernel values
    kernel.at<float>(1, 1) = 5.0;
    kernel.at<float>(0, 1) = -1.0;
    kernel.at<float>(2, 1) = -1.0;
    kernel.at<float>(1, 0) = -1.0;
    kernel.at<float>(1, 2) = -1.0;

    // filter the image
    cv::filter2D(image, result, image.depth(), kernel);
}
```

图像的运算

图像相加

```
// 两幅图像相加1（使用函数）
    cv::addWeighted(image1, 0.7, image2, 0.9, 0., result); // 加权相加 result =
0.7 * image1 + 0.9 * image2 + 0.
    // 类似的还有 cv::add(), cv::subtract, cv::multiply(), cv::divide()

// 两幅图像相加2（使用重载运算符）
    result = 0.7 * image1 + 0.9 * image2;
```

分割图像通道

将彩色图像的三个通道分割为三张图片并分开处理

```
image2 = cv::imread("rain.jpg", cv::IMREAD_GRAYSCALE);

// 创建三幅图像的向量
std::vector<cv::Mat> planes; // 每幅图像对应一个通道

// 将一个三通道的图像分割为三个单通道图像
cv::split(image1, planes);

// 将下雨图像（灰度，单通道）加到boldt图像的蓝色通道上去
planes[0] += image2;

// 将三个通道图像合并为一个三通道图像
cv::merge(planes, result);
```

图像重映射

```
void wave(const cv::Mat& image, cv::Mat& result)
{
    // 映射矩阵（浮点数型cv::Mat）
```

```
    cv::Mat srcX(image.rows, image.cols, CV_32F);
    cv::Mat srcY(image.rows, image.cols, CV_32F);

    // 创建映射参数
    for (int i = 0; i < image.rows; i++)    // 行循环
    {
        for (int j = 0; j < image.cols; j++)    // 列循环
        {
            /// <第(i, j)个像素的新位置>

            // 保持在同一列，原来在第j列的像素，现在仍在第j列
            srcX.at<float>(i, j) = j;   // 第j列

            // 原来在第i行的像素，现在根据一个正弦曲线移动
            srcY.at<float>(i, j) = i + 3 * sin(j / 6.0);    // 第i行加上第j列的一个
正弦函数
        }
    }

    // 应用映射参数
    cv::remap(
        image,              // 输入图像
        result,             // 输出图像
        srcX,               // x方向的映射规则
        srcY,               // y方向的映射规则
        cv::INTER_LINEAR);  // 插值方法
}
```