OpenCV C++ 计算机视觉

Ch1 图像编程入门

1.1 装载、显示和存储图像

```
#include <iostream>

//引入定义了所需的类和函数的头文件
#include <opencv2/core.hpp>    //定义图像数据结构的核心
#include <opencv2/highgui.hpp>//包含所有图形接口函数
#include <opencv2/imgproc.hpp>

//响应鼠标与图像窗口交互的事件，注册回调函数 onMouse
//第一个参数表示触发回调函数的鼠标事件的类型；后面两个参数是事件发生时鼠标的位置，用像素坐标表示
//flags 表示事件发生时按下了鼠标的哪个键；最后一个参数是指向任意对象的指针，作为附加的参数发送
给函数
void onMouse( int event, int x, int y, int flags, void* param)  {

    //reinterpret_cast是C++强制类型转换符，用于指针类型转换
    //此处 void* 指针强制转换为 cv::Mat* 把所显示图像的地址作为附加参数
    cv::Mat *im= reinterpret_cast<cv::Mat*>(param);

    switch (event) { // 调度事件

        case cv::EVENT_LBUTTONDOWN: // 鼠标左键按下事件
        // 鼠标事件的回调函数可能收到的事件还有 cv::EVENT_MOUSEMOVE、
cv::EVENT_LBUTTONUP、                  //
cv::EVENT_RBUTTONDOWN 和 cv::EVENT_RBUTTONUP

            // 显示像素值(x,y)
            // 用 cv::Mat 对象的 at 方法来获取(x, y)的像素值
            std::cout << "at (" << x << "," << y << ") value is: "
                      << static_cast<int>(im->at<uchar>(cv::Point(x,y))) <<
std::endl;
            // static_cast<type>( expression ) 类型转换运算符，把expression转换为
type类型
            // 此处将 8 位无符号字符型(unsigned char)转换为整型
            break;
    }
}

int main() {

    cv::Mat image; //定义一个表示图像的变量，创建一个尺寸为 0×0 的图像
    std::cout << "This image is " << image.rows << " x "
              << image.cols << std::endl;

    // 从文件读入一个图像，解码，然后分配内存
    image=  cv::imread("puppy.bmp",
                       cv::IMREAD_GRAYSCALE);
                       // CV::IMREAD_GRAYSCALE 以单通道灰度图像方式读入，相当于输入0
```

```
                        // CV::IMREAD_COLOR      以三通道彩色图像方式读入，相当于输入1
                        // CV::IMREAD_UNCHANGED 以文件本身的格式读入，相当于输入-1

    // 检查图像的读取是否正确（如果找不到文件、文件 被破坏或者文件格式无法识别，就会发生错误）
    if (image.empty()) {  // 错误处理，如果没有分配图像数据，empty 方法将返回 true
        // 未创建图像……
        // 可能显示一个错误消息
        // 并退出程序
        std::cout << "Error reading image..." << std::endl;
        return 0;
    }

    // 检查图像的大小、通道数
    std::cout << "This image is " << image.rows << " x "
              << image.cols << std::endl;
    std::cout << "This image has "
              << image.channels() << " channel(s)" << std::endl;

    // 定义窗口，显示图像在此窗口中
    cv::namedWindow("Original Image");
    cv::imshow("Original Image", image);

    // cv::Mat* 指针强制转换为void*，把所显示图像的地址作为附加参数传给回调函数onMouse
    // 每当遇到鼠标点击事件时，控制台显示对应像素的值
    cv::setMouseCallback("Original Image", onMouse, reinterpret_cast<void*>
(&image));

    cv::Mat result; // 创建另一个空的图像
    // 翻转输入图像并创建新的矩阵来存放输出结果
    cv::flip(image,result,1); // 正数表示水平
                              // 0 表示垂直
                              // 负数表示水平和垂直
    //cv::flip(image,image,1); // 就地处理，不创建新的图像

    cv::namedWindow("Output Image"); // 输出窗口
    cv::imshow("Output Image", result);

    cv::waitKey(0); // 0 (默认值)表示永远地等待按键
                    // 键入的正数表示等待的毫秒数

    cv::imwrite("output.bmp", result); // 保存结果

    cv::namedWindow("Drawing on an Image");

    cv::circle(image,               // 目标图像
               cv::Point(155,110), // 中心点像素坐标
               65,                 // 半径
               0,                  // 颜色(在灰度图像上进行绘制用单个整数来表示颜色)
               3);                 // 厚度

    cv::putText(image,                   // 目标图像
                "This is a dog.",        // 文本
                cv::Point(40,200),       // 文本位置
                cv::FONT_HERSHEY_PLAIN,  // 字体类型
                2.0,                     // 字体大小
                255,                     // 字体颜色
                2);                      // 文本厚度
```

```
    cv::imshow("Drawing on an Image", image);

    cv::waitKey(0);

    return 0;
}
```

图像类型

U：无符号整数 unsigned int

S：有符号整数 signed int

F：浮点数 float

C：通道数 channel

无C/C1 单通道，灰度图像

C3 三通道，彩色图像（BGR）

> imshow 显示由整数（CV_16U 表示 16位无符号整数，CV_32S 表示 32位有符号整数）构成的图像时，图像每个像素的值会被除以 256，以便能够在 256级灰度中显示；在显示由浮点数构成的图像时，值的范围会被假设为 0.0（显示黑色）~1.0（显示白色）。超出这个 范围的值会显示为白色（大于 1.0的值）或黑色（小于 0.0的值）。

1.2 cv::Mat

cv::Mat 类是用来存放图像（以及其他矩阵数据）的数据结构。

```
#include <iostream>

#include <opencv2/core.hpp>
#include <opencv2/highgui.hpp>

// 创建一幅图像的测试函数
cv::Mat function() {

   // 创建由无符号字符（unsigned char）构成的灰度图像
   cv::Mat ima(500,500,CV_8U,50);
   // 返回图像
   return ima;
}

int main() {

    // 创建一个 240 行×320 列的新图像
    cv::Mat image1(240,320,CV_8U,100);
    // cv::Mat image1(240,320,CV_8U,cv::Scalar(100));
    // cv::Scalar 用于在调用函数时传递像素值,该结构通常包含一个或三个值

    cv::imshow("Image", image1);
    cv::waitKey(0);

    // 重新分配一个新图像（仅在大小或类型不同时）
```

```
    // 用 create 方法分配或重新分配图像的数据块
    // 如果图像已被分配，其原来的内容会先被释放，如果新的尺寸和类型与原来的相同，就不会重新分配内存
    image1.create(200,200,CV_8U);
    image1= 200;

    cv::imshow("Image", image1);
    cv::waitKey(0);

    // 创建一个红色的图像
    // 通道次序为 BGR （蓝、绿、红）
    cv::Mat image2(240,320,CV_8UC3,cv::Scalar(0,0,255));
    // cv::Size 结构包含了矩阵高度和宽度提供图像的尺寸信息;另外，可以用 size()方法得到当前矩阵的大小
    // cv::Mat image2(cv::Size(320,240),CV_8UC3);
    // image2= cv::Scalar(0,0,255);

    cv::imshow("Image", image2);
    cv::waitKey(0);

    // 读入一幅图像
    cv::Mat image3=  cv::imread("puppy.bmp");

    // 在两幅图像之间赋值时，图像数据（即像素）并不会被复制，两幅图像都指向同一个内存块
    // 任何一个进行转换都会影响到其他图像(浅复制)
    cv::Mat image4(image3);
    image1= image3;

    // 这些图像是源图像的副本图像(深复制)
    image3.copyTo(image2);
    cv::Mat image5= image3.clone();

    // 转换图像进行测试
    cv::flip(image3,image3,1);

    // 检查哪些图像在处理过程中受到了影响
    // 结果为image1\3\4发生翻转，image1\4为image3的浅拷贝，指向同一数据存储区域一变全变
    cv::imshow("Image 3", image3);
    cv::imshow("Image 1", image1);
    cv::imshow("Image 2", image2);
    cv::imshow("Image 4", image4);
    cv::imshow("Image 5", image5);
    cv::waitKey(0);

    // 从函数中获取一个灰度图像
    // 可以用变量 gray 操作由 function 函数创建的图像，而不需要额外分配内存
    // 对由 function 函数创建的灰度图像进行了一次浅复制，当局部变量 ima 超出作用范围后，ima 会被释放
    // 变量 gray 引用了 ima 内部的图像数据，因此 ima 的内存块不会被释
    cv::Mat gray= function();

    cv::imshow("Image", gray);
    cv::waitKey(0);

    // 作为灰度图像读入
    image1=  cv::imread("puppy.bmp", CV_LOAD_IMAGE_GRAYSCALE);

    // 转换成32位浮点型图像[0,1]
```

```
    // 用 convertTo 方法要把一幅图像复制到另一幅图像中，且两者的数据类型不相同
    // 此方法包含两个可选参数：缩放比例和偏移量，两幅图像的通道数量必须相同
    image1.convertTo(image2,CV_32F,1/255.0,0.0);

    cv::imshow("Image", image2);

    // 使用模板类 cv::Matx 处理矩阵

    // 3×3 双精度型矩阵
    cv::Matx33d matrix(3.0, 2.0, 1.0, //cv::Matx<double, 3, 3>
                       2.0, 1.0, 3.0,
                       1.0, 2.0, 3.0);
    // 3×1 列向量
    cv::Matx31d vector(5.0, 1.0, 3.0);//cv::Matx<double, 3, 1>
    // 相乘
    cv::Matx31d result = matrix*vector;

    std::cout << result;

    cv::waitKey(0);
    return 0;
}
```

在使用类的时候要特别小心，不要返回图像的类属性。下面的实现方法很容易引发错误：

```
class Test {
     // 图像属性
     cv::Mat ima;
   public:
     // 在构造函数中创建一幅灰度图像
     Test() : ima(240,320,CV_8U,cv::Scalar(100)) {}

     // 用这种方法回送一个类属性，这是一种不好的做法
     cv::Mat method() { return ima; }
};
```

如果某个函数调用了这个类的 method，就会对图像属性进行一次浅复制。副本一旦被修改， class 属性也被“偷偷地”修改，这会影响这个类的后续行为（反之亦然）。这违反了面向对象编程中重要的封装性原理。为了避免这种类型的错误，需要将其改成返回属性的一个副本。

1.3 ROI (Region of Interest)

```
#include <iostream>

#include <opencv2/core.hpp>
#include <opencv2/highgui.hpp>

int main() {

    cv::namedWindow("Image");

    cv::Mat image=  cv::imread("puppy.bmp");

    // 读入 logo
    cv::Mat logo=  cv::imread("smalllogo.png");
```

```
    // 在图像的右下角定义一个 ROI
    // 图像和 ROI 共享同一块图像数据
    // 使用 cv::Rect 方法描述一个矩形区域，前两个参数表示矩形的位置，后两个参数表示矩形的宽度和高度
    cv::Mat imageROI(image,
                     cv::Rect(image.cols-logo.cols, // ROI 坐标
                              image.rows-logo.rows,
                              logo.cols,logo.rows));// ROI 大小

    // 插入标志
    logo.copyTo(imageROI);

    cv::imshow("Image", image);
    cv::waitKey(0);


    image=  cv::imread("puppy.bmp");

    // cv::Mat 的 operator()函数返回另一个 cv::Mat 实例
    // 在定义 ROI 时，数据没有被复制，因此它的执行时间是固定的，不受 ROI 尺寸的影响
    imageROI= image(cv::Rect(image.cols-logo.cols,
                             image.rows-logo.rows,
                             logo.cols,logo.rows));

    // 用行和列的值域定义 ROI
    // imageROI= image(cv::Range(image.rows-logo.rows,image.rows),
    //                 cv::Range(image.cols-logo.cols,image.cols));
    // 定义由图像中一些行组成的 ROI： cv::Mat imageROI= image.rowRange(start,end);
    // 定义由图像中一些列组成的 ROI： cv::Mat imageROI= image.colRange(start,end);

    // 把标志作为掩码（必须是灰度图像）
    // 掩码是一个 8 位图像，如果掩码中某个位置的值不为 0，在这个位置上的操作就会起作用；
    // 如果掩码中某些像素位置的值为 0，对图像中相应位置的操作将不起作用
    cv::Mat mask(logo);

    // 插入标志，只复制掩码不为 0 的位置
    logo.copyTo(imageROI,mask);

    cv::imshow("Image", image);
    cv::waitKey(0);

    return 0;
}
```

Ch2 操作像素

2.1 访问像素值

```
#include <opencv2/core/core.hpp>
#include <opencv2/highgui/highgui.hpp>
#include <opencv2/imgproc/imgproc.hpp>
#include <random>

// 向图像中增加椒盐噪声
void salt(cv::Mat image, int n) {
```

```
    // 随机数生成器
    std::default_random_engine generator;
    std::uniform_int_distribution<int> randomRow(0, image.rows - 1);
    std::uniform_int_distribution<int> randomCol(0, image.cols - 1);

    int i,j;
    for (int k=0; k<n; k++) {

        // 随机生成图形位置
        i= randomCol(generator);
        j= randomRow(generator);

        // 使用 type 方法区分灰度图像和彩色图像
        if (image.type() == CV_8UC1)
        {
            // 单通道 8 位灰度图像
            // 调用 at 方法时须指定图像元素的类型，指定的类型与矩阵内的类型一致
            // 因为 cv::Mat 可以接受任何类型的元素，该方法不会进行任何类型转
            image.at<uchar>(j,i)= 255;

        }
        else if (image.type() == CV_8UC3)
        {
            // 3 通道彩色图像
            // 彩色图像的每个像素对应红、绿、蓝三个通道，包含彩色图像的 cv::Mat 类会返回一个
向量
            // 向量中包含3个无符号字符型（unsigned char）的数据，OpenCV定义这样的短向量为
cv::Vec3b
            image.at<cv::Vec3b>(j,i)[0]= 255;
            image.at<cv::Vec3b>(j,i)[1]= 255;
            image.at<cv::Vec3b>(j,i)[2]= 255;
            // image.at<cv::Vec3b>(j, i) = cv::Vec3b(255, 255, 255);
        }
    }
}

// 使用 cv::Mat_类（cv::Mat 类的模板子类）添加噪声
// cv::Mat_类定义了一些新的方法，但没有定义新的数据属性，因此这两个类的指针或引用可以直接互相
转换
void salt2(cv::Mat image, int n) {

    // 必须是一张灰度图像
    // CV_Assert() 若括号中的表达式值为false，则返回一个错误信息，终止程序执行
    CV_Assert(image.type() == CV_8UC1);

    std::default_random_engine generator;
    std::uniform_int_distribution<int> randomRow(0, image.rows - 1);
    std::uniform_int_distribution<int> randomCol(0, image.cols - 1);

    // 用 Mat 模板操作图像
    // 创建 cv::Mat_变量定义它的元素类型
    cv::Mat_<uchar> img(image);
    //  cv::Mat_<uchar>& im2= reinterpret_cast<cv::Mat_<uchar>&>(image);

    int i,j;
    for (int k=0; k<n; k++) {

        i = randomCol(generator);
```

```
            j = randomRow(generator);

            // 访问相应位置处那个像素值
            // 使用 cv::Mat_类新方法 operator()（重载操作符），可直接访问矩阵的元素（返回类型已知）
            img(j,i)= 255;
        }
    }


    int main()
    {
        cv::Mat image= cv::imread("boldt.jpg",1);

        // 调用函数以添加噪声
        salt(image,3000);

        // 显示结果
        cv::namedWindow("Image");
        cv::imshow("Image",image);

        cv::imwrite("salted.bmp",image);

        cv::waitKey();

        // 测试函数salt2
        image= cv::imread("boldt.jpg",0);

        salt2(image, 500);

        cv::namedWindow("Image");
        cv::imshow("Image",image);

        cv::waitKey();

        return 0;
    }
```

> 还有类似的向量类型用来表示二元素向量和四元素向量（cv::Vec2b 和 cv::Vec4b）。此 外还有针对其他元素类型的向量。例如，表示二元素浮点数类型的向量就是把类型名称的后一个字母换成 f，即 cv::Vec2f。对于短整型，后的字母换成 s；对于整型，后的字母换成 i； 对于双精度浮点数向量，后的字母换成 d。所有这些类型都用 cv::Vec<T,N>模板类定义，其 中 T 是类型，N 是向量元素的数量。
>
> 这些修改图像的函数在使用图像作为参数时都 采用了值传递的方式，是因为它们在复制图像时仍共享了同一块图像数据。因此在需要修改图像内容时，图像参数没必要采用引用传递的方式。

2.2 用指针扫描图像

> 彩色图像由三通道像素组成，每个通道表示红、绿、蓝三原色中一种颜色的亮度值，每个数 值都是 8位无符号字符类型，因此颜色总数为 256×256×256，即超过 1600万种颜色。因此，为了降低分析的复杂性，有时需要减少图像中颜色的数量。一种实现方法是把 RGB 空间细分到大 小相等的方块中。例如，如果把每种颜色数量减少到 1/8，那么颜色总数就变为 32×32×32。将旧 图像中的每个颜色值划分到一个方块，该方块的中间值就是新的颜色值；新图像使用新的颜色值，颜色数就减少了。
> 因此，基本的减色算法很简单。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。
>
> 一个宽 W 高 H 的图像所需的内存块大小为 W×H×3 uchars。不过出于性能上的考虑用几个额外的像素来填补行的长度。这是因为，如果行数是某个数字（例如 8）的整数倍，图像处理的性能可能会提高，因此好根据内存配置情况将数据对齐。当然，这些额外的像素既不会显示也不被保存，它们的额外数据会被忽略。OpenCV 把经过填充的行的长度指定为有效宽度。如果图像没有用额外的像素填充，那么有效宽度就等于实际的图像宽度。用 step 数据属性可得到单位是字节的有效宽度。即使图像的类型不是 uchar，step 仍然能提供行的字节数。我们可以通过 elemSize 方法（例如一个三通道短整型的矩阵 CV_16SC3，elemSize 会返回 6）获得像素的大小，通过 channels 方法（灰度图像为 1，彩色图像为 3）获得图像中通道的数量，后用 total 方法返回矩阵中的像素（即矩阵的条目）总数。

2.3 用迭代器扫描图像

> 对于彩色图像的 cv::Mat 实例，可以使用 image.begincv::Vec3b()。还可以在迭代器上使用数学计算，例如若要从图像的第二行开始，可以用 image.begincv::Vec3b()+image.cols 初始 化 cv::Mat 迭代器。获取集合结束位置的方法也类似，只是改用 end 方法。但是，用 end 方法得 到的迭代器已经超出了集合范围，因此必须在结束位置停止迭代过程。结束的迭代器也能使用数学计算，例如在后一行前就结束迭代，可使用 image.endcv::Vec3b()-image.cols
>
> 可以创建常量迭代器，用作对常量 cv::Mat 的引用， 或者表示当前循环不修改 cv::Mat 实例。常量迭代器的定义如下所示：
> cv::MatConstIteratorcv::Vec3b *it; 或者:*
> *cv::Mat*cv::Vec3b::const_iterator it;

2.4 编写高效的图像扫描循环

```
#include <iostream>

#include <opencv2/core/core.hpp>
#include <opencv2/highgui/highgui.hpp>

// 减色函数：减少图像中颜色的数量
// 参数为图像和每个颜色通道的减色因子，在原图像上处理
// 创建一个二重循环遍历所有像素值
void colorReduce(cv::Mat image, int div=64) {
      // div 必须是 2 的幂
      int nl= image.rows; // 行数
      int nc= image.cols * image.channels(); // 每行的元素数量

      for (int j=0; j<nl; j++) {

          //  使用 ptr 方法返回第 j 行的起始地址
          uchar* data= image.ptr<uchar>(j);

          for (int i=0; i<nc; i++) {

            // 处理每个像素
            data[i]= data[i]/div*div + div/2;
```

```
            } // 一行结束
        }
}

// 输入、输出为不同图像
void colorReduceIO(const cv::Mat &image, // 输入图像
                   cv::Mat &result,      // 输出图像
                   int div = 64) {

    int nl = image.rows; // 行数
    int nc = image.cols; // 列数
    int nchannels = image.channels(); // 通道数

    //检查输出图像，验证是否分配了一定大小的数据缓冲区，以及像素类型与输入图像是否相符
    // cv::Mat 的 create 方法中包含这个检查过程
    // 当用新的大小和像素类型重新分配矩阵时，调用 create 方法
    // 如果矩阵已有的大小和类型刚好与指定的大小和类型相同，这个方法就不会执行任何操作，直接返回。
    result.create(image.rows, image.cols, image.type());

    for (int j = 0; j<nl; j++) {

        // 获得第 j 行的输入和输出的地址
        const uchar* data_in = image.ptr<uchar>(j);
        uchar* data_out = result.ptr<uchar>(j);

        for (int i = 0; i<nc*nchannels; i++) {

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

        }
    }
}

// Test 1
// 利用整数除法的特性，取不超过又接近结果的整数
void colorReduce1(cv::Mat image, int div=64) {

      int nl= image.rows;
      int nc= image.cols * image.channels();
      uchar div2 = div >> 1; // div2 = div/2

      for (int j=0; j<nl; j++) {

          uchar* data= image.ptr<uchar>(j);

          for (int i=0; i<nc; i++) {

              // 利用指针运算从一列移到下一列
              *data++= *data/div*div + div2;

          }
      }
}

// Test 2
// 减色计算使用取模运算符，直接得到 div 的倍数
void colorReduce2(cv::Mat image, int div=64) {
```

```
        int nl= image.rows;
        int nc= image.cols * image.channels();
        uchar div2 = div >> 1; // div2 = div/2

        for (int j=0; j<nl; j++) {

            uchar* data= image.ptr<uchar>(j);

            for (int i=0; i<nc; i++) {

                    int v= *data;
                    *data++= v - v%div + div2;

            }
        }
}

// Test 3
// 使用位运算符，把像素值的前 n 位掩码后就得到最接近的 div 的倍数
void colorReduce3(cv::Mat image, int div=64) {

        int nl= image.rows;
        int nc= image.cols * image.channels();
        int n= static_cast<int>(log(static_cast<double>(div))/log(2.0) + 0.5);
        // 用来截取像素值的掩码
        uchar mask= 0xFF<<n;
        uchar div2= 1<<(n-1); // div2 = div/2

        for (int j=0; j<nl; j++) {

            uchar* data= image.ptr<uchar>(j);

            for (int i = 0; i < nc; i++) {

                *data &= mask;      // 掩码
                *data++ |= div2;    // 加 div/2

            }
        }
}


// Test 4
// 使用低层次指针算法
void colorReduce4(cv::Mat image, int div=64) {

        int nl= image.rows;
        int nc= image.cols * image.channels();
        int n= static_cast<int>(log(static_cast<double>(div))/log(2.0) + 0.5);
        int step= image.step; // 可得到一行的总字节数（包括填充像素）

        uchar mask= 0xFF<<n;
        uchar div2 = div >> 1;

        // cv::Mat 类的 data 属性表示内存块第一个元素的地址，它会返回一个无符号字符型的指针
        uchar *data= image.data;
```

```
      for (int j=0; j<nl; j++) {

          for (int i=0; i<nc; i++) {

            *(data+i) &= mask;
            *(data+i) += div2;

          }
          // step 数据属性可得到单位是字节的有效宽度（包括填充像素）
          // 即使图像的类型不是 uchar，step 仍然能提供行的字节数
          data+= step;  // 利用有效宽度来移动行指针，从一行移到下一行
      }
}

// Test 5
// 每次重新计算行的字节数
void colorReduce5(cv::Mat image, int div=64) {

      int nl= image.rows;
      int n= static_cast<int>(log(static_cast<double>(div))/log(2.0) + 0.5);
      uchar mask= 0xFF<<n;

      for (int j=0; j<nl; j++) {

          uchar* data= image.ptr<uchar>(j);

          for (int i=0; i<image.cols * image.channels(); i++) {

            *data &= mask;
            *data++ += div/2;

          }
      }
}

// Test 6
// 对连续图像的扫描
void colorReduce6(cv::Mat image, int div=64) {

      int nl= image.rows;  // 行数
      int nc= image.cols * image.channels();  // 每行的元素总数

      // 通常情况内存足够大的话图像的每一行是连续存放的，也就是在内存上图像的所有数据存放成一行
      // cv::Mat 的 isContinuous()方法判断图像数组是否为连续的，是则返回 true
      if (image.isContinuous())  {
      // 条件可替换为 image.step == image.cols*image.channels();
      // 检查行的长度（字节数）与“列的个数x单个像素”的字节数是否相等
          // 没有填充的像素
          nc= nc*nl;
          nl= 1;  // 它现在成了一个一维数组
       }

      int n= static_cast<int>(log(static_cast<double>(div))/log(2.0) + 0.5);
      // 用来截取像素值的掩码
      uchar mask= 0xFF<<n;
      uchar div2 = div >> 1; // div2 = div/2

      // 把宽度设为 1，高度设为 W×H，从而去除外层的循环
```

```
      // 对于连续图像，在单个（更长）循环中处理图像
      for (int j=0; j<nl; j++) {

          uchar* data= image.ptr<uchar>(j);

          for (int i=0; i<nc; i++) {

            *data &= mask;
            *data++ += div2;

          }
      }
}

// Test 7
// 对连续图像使用 reshape 方法
void colorReduce7(cv::Mat image, int div=64) {

      if (image.isContinuous()) {
        // 没有填充的像素
        image.reshape(1,   // 新的通道数
                      1) ; // 新的行数
      }
      // 用 reshape 方法修改矩阵的维数，不需要复制内存或重新分配内存了
      //第一个参数是新的通道数，第二个参数是新的行数，列数会进行相应的修改

      int nl= image.rows;
      int nc= image.cols*image.channels() ;

      int n= static_cast<int>(log(static_cast<double>(div))/log(2.0) + 0.5);
      uchar mask= 0xFF<<n;
      uchar div2 = div >> 1;

      for (int j=0; j<nl; j++) {

          uchar* data= image.ptr<uchar>(j);

          for (int i=0; i<nc; i++) {

            *data &= mask;
            *data++ += div2;

          }
      }
}

// Test 8
// 使用Mat_迭代器分别处理3个通道
void colorReduce8(cv::Mat image, int div=64) {

      // 使用在 Mat_模板类内部定义的 iterator 类型
      cv::Mat_<cv::Vec3b>::iterator it= image.begin<cv::Vec3b>();
      cv::Mat_<cv::Vec3b>::iterator itend= image.end<cv::Vec3b>();
      uchar div2 = div >> 1; // div2 = div/2

      // 使用常规的迭代器方法 begin 和 end 对像素进行循环遍历
      for ( ; it!= itend; ++it) {
```

```
        (*it)[0]= (*it)[0]/div*div + div2;
        (*it)[1]= (*it)[1]/div*div + div2;
        (*it)[2]= (*it)[2]/div*div + div2;

      }
}

// Test 9
// 使用cv::MatIterator_<cv::Vec3b>创建迭代器对象
void colorReduce9(cv::Mat image, int div=64) {

      // 创建一个 cv::MatIterator_对象
      // 图像迭代器用来访问图像元素，须在编译时明确返回值的类型
      // 使用常规的迭代器方法 begin 和 end 对像素进行循环遍历
      cv::MatIterator_<cv::Vec3b> it= image.begin<cv::Vec3b>();
      cv::MatIterator_<cv::Vec3b> itend= image.end<cv::Vec3b>();

      // 常量向量
      const cv::Vec3b offset(div/2,div/2,div/2);

      for ( ; it!= itend; ++it) {

        *it= *it/div*div + offset;//使用 cv::Vec3b 的重载运算符

      }
}

// Test 10
// 使用掩码的cv::Mat_<cv::Vec3b>::iterator迭代器
void colorReduce10(cv::Mat image, int div=64) {

      int n= static_cast<int>(log(static_cast<double>(div))/log(2.0) + 0.5);

      uchar mask= 0xFF<<n;
      uchar div2 = div >> 1;

      // 使用在 Mat_模板类内部定义的 iterator 类型
      // 创建 cv::Mat_引用时迭代器类型已被指定
      cv::Mat_<cv::Vec3b>::iterator it= image.begin<cv::Vec3b>();
      cv::Mat_<cv::Vec3b>::iterator itend= image.end<cv::Vec3b>();

      // 扫描全部像素
      for ( ; it!= itend; ++it) {

        (*it)[0]&= mask;
        (*it)[0]+= div2;
        (*it)[1]&= mask;
        (*it)[1]+= div2;
        (*it)[2]&= mask;
        (*it)[2]+= div2;

      }
}

// Test 11
// 使用cv::Mat_<cv::Vec3b>::iterator 迭代器
void colorReduce11(cv::Mat image, int div=64) {
```

```
        // 创建迭代器
        cv::Mat_<cv::Vec3b> cimage= image;
        cv::Mat_<cv::Vec3b>::iterator it=cimage.begin();
        cv::Mat_<cv::Vec3b>::iterator itend=cimage.end();
        uchar div2 = div >> 1;

        for ( ; it!= itend; it++) {

          (*it)[0]= (*it)[0]/div*div + div2;
          (*it)[1]= (*it)[1]/div*div + div2;
          (*it)[2]= (*it)[2]/div*div + div2;

        }
}


// Test 12
// 使用at方法
void colorReduce12(cv::Mat image, int div=64) {

        int nl= image.rows; // 行数
        int nc= image.cols; // 列数
        uchar div2 = div >> 1; // div2 = div/2

        for (int j=0; j<nl; j++) {
            for (int i=0; i<nc; i++) {

             image.at<cv::Vec3b>(j,i)[0]=  image.at<cv::Vec3b>(j,i)[0]/div*div +
div2;
             image.at<cv::Vec3b>(j,i)[1]=  image.at<cv::Vec3b>(j,i)[1]/div*div +
div2;
             image.at<cv::Vec3b>(j,i)[2]=  image.at<cv::Vec3b>(j,i)[2]/div*div +
div2;

            }
        }
}


// Test 13
// 使用cv::Scalar及Mat重载运算符
void colorReduce13(cv::Mat image, int div=64) {

        int n= static_cast<int>(log(static_cast<double>(div))/log(2.0) + 0.5);

        uchar mask= 0xFF<<n;

        // 执行减色
        image=(image&cv::Scalar(mask,mask,mask))+cv::Scalar(div/2,div/2,div/2);
}

// Test 14
// 使用查找表
void colorReduce14(cv::Mat image, int div=64) {

      // 创建LUT(Look-Up Table)像素灰度值的映射表
        cv::Mat lookup(1,256,CV_8U);
```

```
       for (int i=0; i<256; i++) {

         lookup.at<uchar>(i)= i/div*div + div/2;
       }

       cv::LUT(image,lookup,image);
      // void cv::LUT(InputArray src, InputArray lut, OutputArray dst);
      // src表示的是输入图像（可以是单通道也可是3通道）
      // lut表示查找表
      // dst表示输出图像

      //查找表也可以是单通道，也可以是3通道，如果输入图像为单通道，那查找表必须为单通道
      //若输入图像为3通道，查找表可以为单通道，也可以为3通道
      //若为单通道则表示对图像3个通道都应用这个表，若为3通道则分别应用
}

#define NTESTS 15        //测试函数个数
#define NITERATIONS 10   //测试次数

int main()
{
    cv::Mat image = cv::imread("boldt.jpg");

    // 图像处理时间计时
    //  cv::get TickCount()函数返回从近一次计算机开机到当前的时钟周期数
    const int64 start = cv::getTickCount();
    colorReduce(image, 64);

    // 经过的时间（单位：秒）
    // cv::getTickFrequency()返回每秒的时钟周期数
    double duration = (cv::getTickCount() - start) / cv::getTickFrequency();
    std::cout << "Duration= " << duration << "secs" << std::endl;

    cv::namedWindow("Image");
    cv::imshow("Image", image);

    cv::waitKey();

    // 测试函数的不同版本

    int64 t[NTESTS], tinit;
    // 计时器值设置为0
    for (int i = 0; i<NTESTS; i++)
        t[i] = 0;

    cv::Mat images[NTESTS];
    cv::Mat result;

    //定义指向函数的指针变量
    typedef void(*FunctionPointer)(cv::Mat, int);
    FunctionPointer functions[NTESTS] = {
         colorReduce , colorReduce1, colorReduce2, colorReduce3, colorReduce4,
         colorReduce5, colorReduce6, colorReduce7, colorReduce8, colorReduce9,
         colorReduce10, colorReduce11, colorReduce12, colorReduce13,
colorReduce14
    };

    // 重复测试10次
```

```
    int n = NITERATIONS;
    for (int k = 0; k<n; k++) {

        std::cout << k << " of " << n << std::endl;

        // 测试每个版本
        for (int c = 0; c < NTESTS; c++) {

            images[c] = cv::imread("boldt.jpg");

            tinit = cv::getTickCount();
            functions[c](images[c], 64);
            t[c] += cv::getTickCount() - tinit; //同一函数运行10次时间的和

            std::cout << ".";
        }

        std::cout << std::endl;
    }

    // 函数说明
    std::string descriptions[NTESTS] = {
        "original version:",
        "with dereference operator:",
        "using modulo operator:",
        "using a binary mask:",
        "direct ptr arithmetic:",
        "row size recomputation:",
        "continuous image:",
        "reshape continuous image:",
        "with iterators:",
        "Vec3b iterators:",
        "iterators and mask:",
        "iterators from Mat_:",
        "at method:",
        "overloaded operators:",
        "look-up table:",
    };

    for (int i = 0; i < NTESTS; i++) {

        cv::namedWindow(descriptions[i]);
        cv::imshow(descriptions[i], images[i]);
    }

    // 打印平均执行时间
    std::cout << std::endl << "-------------------------------------------" <<
std::endl << std::endl;
    for (int i = 0; i < NTESTS; i++) {

        std::cout << i << ". " << descriptions[i] << 1000.*t[i] /
cv::getTickFrequency() / n << "ms" << std::endl;
    }

    cv::waitKey();
    return 0;
}
```

2.5 扫描图像并访问相邻像素

```
#include <iostream>

#include <opencv2/core/core.hpp>
#include <opencv2/highgui/highgui.hpp>
#include <opencv2/imgproc/imgproc.hpp>

// 锐化图像的处理函数
// 从图像中减去拉普拉斯算子部分，图像的边缘放大，图像会变得更加尖锐
void sharpen(const cv::Mat &image, cv::Mat &result) {
    // 判断是否需要分配图像数据，如果需要，就分配
    result.create(image.size(), image.type());
    int nchannels= image.channels();

    for (int j= 1; j<image.rows-1; j++) { // 处理所有行（除了第一行和最后一行）

        // 访问上一行和下一行的相邻像素，需定义额外的指针，并与当前行的指针一起递增
        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);    // 输出行

        for (int i=nchannels; i<(image.cols-1)*nchannels; i++) {

            // 应用锐化算子
            // 计算锐化的数值 sharpened_pixel= 5*current-left-right-up-down
            *output++= cv::saturate_cast<uchar>(5*current[i]-current[i-
nchannels]-current[i+nchannels]-previous[i]-next[i]);
        }
    }

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

// 使用迭代器对灰度图像进行锐化
void sharpenIterator(const cv::Mat &image, cv::Mat &result) {

    CV_Assert(image.type() == CV_8UC1);

    // 初始化迭代器
    cv::Mat_<uchar>::const_iterator it= image.begin<uchar>()+image.cols;
    cv::Mat_<uchar>::const_iterator itend= image.end<uchar>()-image.cols;
    cv::Mat_<uchar>::const_iterator itup= image.begin<uchar>();
    cv::Mat_<uchar>::const_iterator itdown= image.begin<uchar>()+2*image.cols;

    // 设置输出图像和迭代器
    result.create(image.size(), image.type()); // 必要时分配
    cv::Mat_<uchar>::iterator itout= result.begin<uchar>()+result.cols;
```

```
    for ( ; it!= itend; ++it, ++itout, ++itup, ++itdown) {
            // 调用 cv::saturate_cast 模板函数，并传入运算结果
            // 计算像素的数学表达式的结果经常超出允许的范围（即小于 0 或大于 255）
            // 使用这个函数可把结果调整到8位无符号字符型的范围内
            // 具体做法是把小于 0的数值调整为 0，大于 255的 数值调整为 255
            *itout= cv::saturate_cast<uchar>(*it *5 - *(it-1)- *(it+1)- *itup -
*itdown);
    }

    // 边框上的像素没有完整的相邻像素，把未处理的像素设为 0
    // 使用 setTo 方法对矩阵中的所有元素赋值
    result.row(0).setTo(cv::Scalar(0));
    result.row(result.rows-1).setTo(cv::Scalar(0));
    result.col(0).setTo(cv::Scalar(0));
    result.col(result.cols-1).setTo(cv::Scalar(0));
}

// 使用卷积
void sharpen2D(const cv::Mat &image, cv::Mat &result) {

    // 构造内核（所有入口都初始化为 0），定义一个用于图像的滤波器
    cv::Mat kernel(3,3,CV_32F,cv::Scalar(0));
    // 对内核赋值，当前像素用核心矩阵中心单元格表示
    // 核心矩阵中的每个单元格表示相关像素的乘法系数，核心矩阵的大小就是邻域的大小
    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;

    // 对图像滤波
    // 使用 cv::filter2D 函数，传入图像和内核，即可返回滤波后的图像
    // 使用depth方法，图像深度是指存储每个像素所用的位数
    cv::filter2D(image,result,image.depth(),kernel);
}

int main()
{
    // sharpen

    cv::Mat image= cv::imread("boldt.jpg");
    if (!image.data)
        return 0;

    cv::Mat result;

    double time= static_cast<double>(cv::getTickCount());
    sharpen(image, result);
    time= (static_cast<double>(cv::getTickCount())-time)/cv::getTickFrequency();
    std::cout << "time= " << time << std::endl;

    cv::namedWindow("Image");
    cv::imshow("Image",result);

    // sharpenIterator

    // 以灰度图像载入
    image= cv::imread("boldt.jpg",0);
```

```
    time = static_cast<double>(cv::getTickCount());
    sharpenIterator(image, result);
    time= (static_cast<double>(cv::getTickCount())-time)/cv::getTickFrequency();
    std::cout << "time 3= " << time << std::endl;

    cv::namedWindow("Sharpened Image");
    cv::imshow("Sharpened Image",result);

    // sharpen2D

    image= cv::imread("boldt.jpg");

    time = static_cast<double>(cv::getTickCount());
    sharpen2D(image, result);
    time= (static_cast<double>(cv::getTickCount())-time)/cv::getTickFrequency();
    std::cout << "time 2D= " << time << std::endl;

    cv::namedWindow("Image 2D");
    cv::imshow("Image 2D",result);

    cv::waitKey();

    return 0;
}
```

2.6 图像运算

大部分 C++运算符都已被重载，其中包括位运算符&、 |、 ^、~和函数 min、max、abs。比较运算符 <、 <=、 ==、 !=、>和>=也已被重载，它们返回一个 8位的二值图像。此外还有矩 阵乘法 m1*m2（其中 m1 和 m2 都是 cv::Mat 实例）、矩阵求逆 m1.inv()、变位 m1.t()、行列 式 m1.determinant()、求范数 v1.norm()、叉乘 v1.cross(v2)、点乘 v1.dot(v2)，等等。 在理解这点后，你就会使用相应的组合赋值符了（例如+=运算符）。

```
#include <vector>
#include <opencv2/core.hpp>
#include <opencv2/highgui.hpp>
#include <opencv2/imgproc.hpp>

int main()
{
    cv::Mat image1;
    cv::Mat image2;

    image1= cv::imread("boldt.jpg");
    image2= cv::imread("rain.jpg");
    if (!image1.data)
        return 0;
    if (!image2.data)
        return 0;

    cv::namedWindow("Image 1");
    cv::imshow("Image 1",image1);
    cv::namedWindow("Image 2");
    cv::imshow("Image 2",image2);
```

```
    cv::Mat result;
    // 两个图象相加，得到加权和
    // c[i]= k1*a[i]+k2*b[i]+k3;
    // cv::addWeighted(imageA,k1,imageB,k2,k3,resultC);
    cv::addWeighted(image1,0.7,image2,0.9,0.,result);

    cv::namedWindow("result");
    cv::imshow("result",result);

    // 使用重载运算符operator+
    result= 0.7*image1+0.9*image2;

    cv::namedWindow("result with operators");
    cv::imshow("result with operators",result);

    image2= cv::imread("rain.jpg",0);

    // 创建三幅图像的向量
    std::vector<cv::Mat> planes;
    // 将一个三通道图像分割为三个单通道图像
    // 使用 cv::split 函数，将图像的三个通道分别复制到 三个 cv::Mat 实例中
    cv::split(image1,planes);
    // 加到蓝色通道上
    planes[0]+= image2;
    // cv::merge 函数执行反向操作，用三个单通道图像创建一个彩色图像
    cv::merge(planes,result);

    cv::namedWindow("Result on blue channel");
    cv::imshow("Result on blue channel",result);

    cv::waitKey();

    return 0;
}
```

2.7 图像重映射

```
#include <opencv2/core/core.hpp>
#include <opencv2/highgui/highgui.hpp>
#include <opencv2/imgproc/imgproc.hpp>

#include <math.h>

// 重映射图像，创建波浪形效果
void wave(const cv::Mat &image, cv::Mat &result) {

    // 映射参数
    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++) {
        for (int j=0; j<image.cols; j++) {

            // (i,j)像素的新位置
            srcX.at<float>(i,j)= j;// 保持在同一列,记录列坐标
```

```
            srcY.at<float>(i,j)= i+3*sin(j/6.0);// 原来在第 i 行的像素，根据一个正弦曲线移动
                                                // 记录行坐标

            // 水平翻转
            // srcX.at<float>(i, j)= static_cast<float>(image.cols-j-1);
            // srcY.at<float>(i, j)= static_cast<float>(i);
            // 上下翻转
            // srcX.at<float>(i, j) = static_cast<float>(j);
            // srcY.at<float>(i, j) = static_cast<float>(image.rows-i-1);
        }
    }

    // 应用映射参数
    // 目标图像中的像素可以映射回一个非整数的值（即处在两个像素之间的位置）
    cv::remap(image,  // 源图像
              result, // 目标图像
              srcX,   // x 映射,(x, y)的第一个映射或者是CV_16SC2、CV_32FC1或CV_32FC2的x值
              srcY,   // y 映射,第二个映射,表示类型为CV_16UC1、CV_32FC1的y值
                      //                  或空值(如果map1是用(x,y)进行表示)
              cv::INTER_LINEAR); // 像素插值法
}

int main()
{
    cv::Mat image= cv::imread("boldt.jpg",0);

    cv::namedWindow("Image");
    cv::imshow("Image",image);

    // 重映射图像
    cv::Mat result;
    wave(image,result);

    cv::namedWindow("Remapped image");
    cv::imshow("Remapped image",result);

    cv::waitKey();
    return 0;
}
```

Ch3 处理图像的颜色

3.1 用策略设计模式比较颜色

```
#if !defined COLORDETECT
#define COLORDETECT

#include <opencv2/core/core.hpp>
#include <opencv2/imgproc/imgproc.hpp>

class ColorDetector {
```

```
  private:

      // 允许的最小差距
      int maxDist;

      // 目标颜色
      cv::Vec3b target;

      // 颜色转换后的图像
      cv::Mat converted;
      bool useLab;

      // 存储二值映像结果的图像
      cv::Mat result;

  public:

      // 空构造函数
      // 在此初始化全部默认参数，设置为默认值
      ColorDetector() : maxDist(100), target(0,0,0), useLab(false) {}

      // 输入对象为Lab色彩空间的构造函数
      ColorDetector(bool useLab) : maxDist(100), target(0,0,0), useLab(useLab)
{}

      // 完整的构造函数，要求用户输入目标颜色和颜色距离（要确保输入值可预测并且有效）
      ColorDetector(uchar blue, uchar green, uchar red, int mxDist=100, bool
useLab=false): maxDist(mxDist), useLab(useLab) {

          // 目标颜色
          setTargetColor(blue, green, red);
      }

      // 计算与目标颜色向量间的距离
      int getDistanceToTargetColor(const cv::Vec3b& color) const {
          return getColorDistance(color, target);
      }

      // 计算两个颜色向量间的距离
      // 把 RGB 值差距的绝对值（也称为城区距离）进行累加
      int getColorDistance(const cv::Vec3b& color1, const cv::Vec3b& color2)
const {

          return abs(color1[0]-color2[0])+
                     abs(color1[1]-color2[1])+
                     abs(color1[2]-color2[2]);

          // 计算向量的欧几里得范数的函数
          // return static_cast<int>(
          //        cv::norm<int,3>(cv::Vec3i(color[0]-color2[0],
          //                                  color[1]-color2[1],
          //                                  color[2]-color2[2])));

          // 确保结果在输入数据类型的范围之内
          // cv::Vec3b dist;
          // cv::absdiff(color,color2,dist);  // cv::absdiff 计算两个数组差的绝对值
          // return cv::sum(dist)[0]; //使用重载运算符()
      }
```

```
// 处理图像，返回一个单通道的二进制图像
// 在类的内部声明函数，外部进行定义
cv::Mat process(const cv::Mat &image);

// 重载运算符()
cv::Mat operator()(const cv::Mat &image) {

    cv::Mat input;

    if (useLab) { // 转换为Lab色彩空间
        cv::cvtColor(image, input, CV_BGR2Lab);
    }
    else {
        input = image;
    }

    cv::Mat output;
    // 计算与目标颜色的距离的绝对值
    cv::absdiff(input,cv::Scalar(target),output);
    // 把通道分割进 3 幅图像
    std::vector<cv::Mat> images;
    cv::split(output,images);
    // 3 个通道相加（这里可能出现饱和的情况）
    output= images[0]+images[1]+images[2];
    // 应用阈值
    cv::threshold(output,                // 相同的输入/输出图像
                  output,
                  maxDist,               // 阈值（必须<256）
                  255,                   // 最大值
                  cv::THRESH_BINARY_INV); // 阈值化模式

    return output;
}

// Getters and setters

// 设置颜色差距的阈值，实现 color 公差参数的定制
// 阈值必须是正数，否则就设为 0
void setColorDistanceThreshold(int distance) {

    if (distance<0)
        distance=0;
    maxDist= distance;
}

// 取得颜色差距的阈值
int getColorDistanceThreshold() const {

    return maxDist;
}

// 设置需要检测的颜色，用三个参数表示三个颜色组件
void setTargetColor(uchar blue, uchar green, uchar red) {

    // 次序为 BGR
    target = cv::Vec3b(blue, green, red);
```

```
            if (useLab) {
                // 临时的单像素图像
                cv::Mat tmp(1, 1, CV_8UC3);
                tmp.at<cv::Vec3b>(0, 0) = cv::Vec3b(blue, green, red);

                // 将目标颜色转换成 Lab色彩空间
                cv::cvtColor(tmp, tmp, CV_BGR2Lab);

                target = tmp.at<cv::Vec3b>(0, 0);
            }
        }

        // 设置需要检测的颜色，用 cv::Vec3b 保存颜色值
        void setTargetColor(cv::Vec3b color) {

            target= color;
        }

        // 取得需要检测的颜色
        cv::Vec3b getTargetColor() const {

            return target;
        }
};


#endif
```

```
#include <opencv2/core/core.hpp>
#include <opencv2/highgui/highgui.hpp>

#include "colordetector.h"

int main()
{
    // 1. 创建图像处理器对象（调用空构造函数）
    ColorDetector cdetect;

    // 2. 读取输入的图像
    cv::Mat image= cv::imread("boldt.jpg");
    if (image.empty())
        return 0;
    cv::namedWindow("Original Image");
    cv::imshow("Original Image", image);

    // 3. 设置输入参数
    cdetect.setTargetColor(230,190,130);

    // 4. 处理图像并显示结果
    cv::namedWindow("result");
    cv::Mat result = cdetect.process(image);
    cv::imshow("result",result);

    // 用仿函数方法检测指定的颜色 (Lab)
    ColorDetector colordetector(230, 190, 130,  // 颜色
```

```
                                    45, true); // Lab色彩空间
cv::namedWindow("result (functor)");
result = colordetector(image);  //使用重载运算符()
cv::imshow("result (functor)",result);

// cv::floodFill 函数在判断一个像素时，检查附近像素的状态，识别某种颜色的相关区域
// 用户指定一个起始位置和允许的误差，就可以找出颜色接近的连续区域
// 根据亚像素确定搜寻的颜色，并检查它旁边的像素，判断它们是否为颜色接近的像素；
// 然后，继续检查它们旁边的像素，并持续操作，从图像中提取出特定颜色的区域
cv::floodFill(image,              // 输入/输出图像
    cv::Point(100, 50),           // 起始点
    cv::Scalar(255, 255, 255),    // 填充颜色
    (cv::Rect*)0,                 // 填充区域的边界矩形
    cv::Scalar(35, 35, 35),       // 偏差的最小/最大阈值
    cv::Scalar(35, 35, 35),       // 正差阈值，两个阈值通常相等
    cv::FLOODFILL_FIXED_RANGE); // 与起始点像素比较

cv::namedWindow("Flood Fill result");
result = colordetector(image);
cv::imshow("Flood Fill result", image);

// 创建图像来显示颜色空间属性
cv::Mat colors(100, 300, CV_8UC3, cv::Scalar(100, 200, 150));
cv::Mat range = colors.colRange(0, 100);
range = range + cv::Scalar(10, 10, 10);  //使用重载运算符+
range = colors.colRange(200, 300);
range = range + cv::Scalar(-10, -10, 10);

cv::namedWindow("3 colors");
cv::imshow("3 colors", colors);

cv::Mat labImage(100, 300, CV_8UC3, cv::Scalar(100, 200, 150));
cv::cvtColor(labImage, labImage, CV_BGR2Lab); //转换至Lab颜色空间
range = colors.colRange(0, 100);
range = range + cv::Scalar(10, 10, 10);
range = colors.colRange(200, 300);
range = range + cv::Scalar(-10, -10, 10);
cv::cvtColor(labImage, labImage, CV_Lab2BGR);

cv::namedWindow("3 colors (Lab)");
cv::imshow("3 colors (Lab)", labImage);

// 亮度与照度对比
cv::Mat grayLevels(100, 256, CV_8UC3);
for (int i = 0; i < 256; i++) {
    // 按列对像素进行赋值
    grayLevels.col(i) = cv::Scalar(i, i, i);
}

range = grayLevels.rowRange(50, 100);
cv::Mat channels[3];
cv::split(range, channels);
channels[1] = 128;
channels[2] = 128;
cv::merge(channels, 3, range);
cv::cvtColor(range, range, CV_Lab2BGR);
```

```
    cv::namedWindow("Luminance vs Brightness");
    cv::imshow("Luminance vs Brightness", grayLevels);

    cv::waitKey();

    return 0;
}
```

3.2 用GrabCut算法分割图像

```
#include <iostream>
#include <opencv2/core/core.hpp>
#include <opencv2/imgproc/imgproc.hpp>
#include <opencv2/highgui/highgui.hpp>


int main()
{
    cv::Mat image= cv::imread("boldt.jpg");
    if (!image.data)
        return 0;

    cv::namedWindow("Original Image");
    cv::imshow("Original Image",image);

    // 创建矩形边框
    cv::Rect rectangle(50,25,210,180);
    // 模型（内部使用）
    cv::Mat bgModel,fgModel;
    // 分割结果（掩膜）
    cv::Mat result;

    // GrabCut 分割算法
    // bgModel:背景模型，如果为null，函数内部会自动创建一个bgModel；
    // fgModel:前景模型，如果为null，函数内部会自动创建一个fgModel；
    // bgModel,fgModel必须是单通道浮点型（CV_32FC1）图像，且行数只能为1，列数只能为13x5；
    cv::grabCut(image,                    // 输入图像
                result,                   // 分割结果
                rectangle,                // 包含前景的矩形
                bgModel, fgModel,         // 背景/前景模型
                5,                        // 迭代次数
                cv::GC_INIT_WITH_RECT); // 使用矩形初始化grabCut

    // 取得标记为“可能属于前景”的像素
    // cv::compare 用于静止背景下移动物体的检测
    // cv::GC_BGD == 0 表示是背景；cv::GC_FGD == 1 表示是前景；
    // cv::GC_PR_BGD == 2 表示可能是背景；cv::GC_PR_FGD == 3 表示可能是前景
    // CMP_EQ=0 相等；CMP_GT=1 大于；CMP_GE=2 大于等于；
    // CMP_LT=3 小于；CMP_LE=4 小于等于；CMP_NE=5 不相等
    cv::compare(result,cv::GC_PR_FGD,result,cv::CMP_EQ);
    // 用“按位与”运算检查第一位
    //  result= result&1; 如果是前景像素，结果为 1

    // 生成输出图像
    cv::Mat foreground(image.size(), CV_8UC3,cv::Scalar(255, 255, 255));
```

```
    image.copyTo(foreground,result); // 不复制背景像素

    // 在原图中画一个矩形
    cv::rectangle(image, rectangle, cv::Scalar(255,255,255),1);
    cv::namedWindow("Image with rectangle");
    cv::imshow("Image with rectangle",image);

    // 显示结果
    cv::namedWindow("Foreground object");
    cv::imshow("Foreground object",foreground);

    cv::waitKey();
    return 0;
}
```

```
#include "colordetector.h"
#include <vector>

cv::Mat ColorDetector::process(const cv::Mat &image) {

      // 必要时重新分配二值图像
      // 与输入图像的尺寸相同，但用单通道
      result.create(image.size(),CV_8U);

      // 转换成 Lab色彩空间
      if (useLab)
          cv::cvtColor(image, converted, CV_BGR2Lab);

      // 取得转换图像的迭代器
      // 输入图像迭代器具有常量属性，无法修改像素值
      cv::Mat_<cv::Vec3b>::const_iterator it= image.begin<cv::Vec3b>();
      cv::Mat_<cv::Vec3b>::const_iterator itend= image.end<cv::Vec3b>();
      cv::Mat_<uchar>::iterator itout= result.begin<uchar>();

      // 取得色彩空间转换输出图像的迭代器
      if (useLab) {
          it = converted.begin<cv::Vec3b>();
          itend = converted.end<cv::Vec3b>();
      }

      // 针对每个像素
      for ( ; it!= itend; ++it, ++itout) {

          // 在每个迭代步骤中计算当前像素的颜色与目标颜色的差距，检查它是否在公差（maxDist）
范围之内
          // 如果是，就在输出图像中赋值 255（白色），否则就赋值 0（黑色）
          if (getDistanceToTargetColor(*it)<maxDist) {

              *itout= 255;

          } else {

              *itout= 0;
          }
```

```
    }

    return result;
}
```

3.3 转换颜色表示法

3.4 用色调、饱和度和亮度表示颜色

```
#include <opencv2/core/core.hpp>
#include <opencv2/highgui/highgui.hpp>
#include <opencv2/imgproc/imgproc.hpp>

#include <iostream>
#include <vector>

void detectHScolor(const cv::Mat& image,        // 输入图像
    double minHue, double maxHue,   // 色调区间
    double minSat, double maxSat,   // 饱和度区间
    cv::Mat& mask) {                // 输出掩码

    // 转换成 HSV 色彩空间
    cv::Mat hsv;
    cv::cvtColor(image, hsv, CV_BGR2HSV);

    // 把 3 个通道分割进 3 幅图像中
    std::vector<cv::Mat> channels;
    cv::split(hsv, channels);
    // channels[0]是色调
    // channels[1]是饱和度
    // channels[2]是亮度

    // 色调掩码
    cv::Mat mask1; // 小于 maxHue
    cv::threshold(channels[0], mask1, maxHue, 255, cv::THRESH_BINARY_INV);
    cv::Mat mask2; // 大于 minHue
    cv::threshold(channels[0], mask2, minHue, 255, cv::THRESH_BINARY);

    cv::Mat hueMask; // 色调掩码
    if (minHue < maxHue)
        hueMask = mask1 & mask2;
    else             // 如果区间穿越 0 度中轴线
        hueMask = mask1 | mask2;

    // 饱和度掩码
    // 从 minSat 到 maxSat
    cv::threshold(channels[1], mask1, maxSat, 255, cv::THRESH_BINARY_INV);
    cv::threshold(channels[1], mask2, minSat, 255, cv::THRESH_BINARY);

    cv::Mat satMask; // 饱和度掩码
```

```
    if (minSat < maxSat)
        satMask = mask1 & mask2;
    else             // 如果区间穿越 0 度中轴线
        satMask = mask1 | mask2;

    // 组合掩码
    mask = hueMask&satMask;
}

int main()
{

    cv::Mat image= cv::imread("boldt.jpg");
    if (!image.data)
        return 0;

    cv::namedWindow("Original image");
    cv::imshow("Original image",image);

    // 转换到 HSV 空间
    cv::Mat hsv;
    cv::cvtColor(image, hsv, CV_BGR2HSV);

    // 将 3 个通道分割到 3 幅图像
    std::vector<cv::Mat> channels;
    cv::split(hsv,channels);
    // channels[0]是色调
    // channels[1]是饱和度
    // channels[2]是亮度

    // 分别展示三个通道
    cv::namedWindow("Value");
    cv::imshow("Value",channels[2]);

    cv::namedWindow("Saturation");
    cv::imshow("Saturation",channels[1]);

    cv::namedWindow("Hue");
    cv::imshow("Hue",channels[0]);

    // 最高亮度图像
    cv::Mat newImage;
    cv::Mat tmp(channels[2].clone());
    // 所有像素的颜色亮度通道将变成 255
    channels[2]= 255;
    // 重新合并通道
    cv::merge(channels,hsv);
    // 转换回 BGR
    cv::cvtColor(hsv,newImage,CV_HSV2BGR);

    cv::namedWindow("Fixed Value Image");
    cv::imshow("Fixed Value Image",newImage);

    // 最大饱和度图像
    channels[1]= 255;
    channels[2]= tmp;
    cv::merge(channels,hsv);
    cv::cvtColor(hsv,newImage,CV_HSV2BGR);
```

```
cv::namedWindow("Fixed saturation");
cv::imshow("Fixed saturation",newImage);

// 最高亮度、最大饱和度图像
channels[1]= 255;
channels[2]= 255;
cv::merge(channels,hsv);
cv::cvtColor(hsv,newImage,CV_HSV2BGR);

cv::namedWindow("Fixed saturation/value");
cv::imshow("Fixed saturation/value",newImage);

// 以人为生成图像说明各种色调/饱和度组合
cv::Mat hs(128, 360, CV_8UC3);
for (int h = 0; h < 360; h++) {
    for (int s = 0; s < 128; s++) {
        hs.at<cv::Vec3b>(s, h)[0] = h/2;      // 所有色调角度
        hs.at<cv::Vec3b>(s, h)[1] = 255-s*2; // 饱和度从高到低
        hs.at<cv::Vec3b>(s, h)[2] = 255;      // 常数
    }
}

cv::cvtColor(hs, newImage, CV_HSV2BGR);

cv::namedWindow("Hue/Saturation");
cv::imshow("Hue/Saturation", newImage);

// 测试皮肤检测

// 读入图片
image= cv::imread("girl.jpg");
if (!image.data)
    return 0;

// 显示原图
cv::namedWindow("Original image");
cv::imshow("Original image",image);

// 检测肤色
cv::Mat mask;
detectHScolor(image,
    160, 10, // 色调为 320 度~20 度
    25, 166, // 饱和度为~0.1~0.65
    mask);

// 显示使用掩码后的图像
cv::Mat detected(image.size(), CV_8UC3, cv::Scalar(0, 0, 0));
image.copyTo(detected, mask);
cv::imshow("Detection result",detected);

// 比较照度和亮度的测试

// 创建线性强度图像
cv::Mat linear(100,256,CV_8U);
for (int i=0; i<256; i++) {

    linear.col(i)= i;
```

```
    }

    // 创建 Lab 色彩空间图像
    linear.copyTo(channels[0]);
    cv::Mat constante(100,256,CV_8U,cv::Scalar(128));
    constante.copyTo(channels[1]);
    constante.copyTo(channels[2]);
    cv::merge(channels,image);

    // 转换回 BGR 色彩空间
    cv::Mat brightness;
    cv::cvtColor(image,brightness, CV_Lab2BGR);
    cv::split(brightness, channels);

    // 创建组合图片
    cv::Mat combined(200,256, CV_8U);
    // 定义 ROI
    cv::Mat half1(combined,cv::Rect(0,0,256,100));
    linear.copyTo(half1);
    cv::Mat half2(combined,cv::Rect(0,100,256,100));
    channels[0].copyTo(half2);

    cv::namedWindow("Luminance vs Brightness");
    cv::imshow("Luminance vs Brightness",combined);

    cv::waitKey();
}
```