第2章 图像的基本操作

2.1 数字图像的表示

我们在电子设备上看到的图像，都可以称为数字图像，例如图 2.1 所示的 Lena。对计算机来说，这幅图像只是一些亮度不同的点。放大图 2.1 中箭头所指的圆形区域可得到图 2.2 所示的效果，图中灰度深浅不一的小方格每一个小方格就是一个点，一个点称为一个像素。对于一幅大小为 MxN 的图像，可以用 MxN 大小的矩阵表示，每个矩阵元素代表一个像素，元素的值表示这个位置图像的亮度，一般来说值越大该点就越亮。（在后面的章节中，对与一幅图像，或称为图像或称为矩阵；对于图像的一个像素，或称为矩阵元素或称为图像像素）

图 2.1Lena 的照片

图 2.2 图 2.1 中圆形区域的放大效果

通常，使用 2 维矩阵 MxN 表示灰度图像，彩色（多通道）图像用 3 维矩阵 MxNx3 表示。对于显图像显示，目前大部分设备使用无符号 8 位整数表示像素的值。

图 2.3 Lena 的灰度图（左）和彩色图（右）

图像数据在计算机内存的存储顺序为自左向右、自上向下，即图像的左上角为原点（也有自左向右、自下向上的顺序，即以左下角为原点），如图 2.4 和图 2.5 所示。

图 2.4 表示的是单通道灰度图像数据的存储，Iij 代表第 i 行第 j 列的像素值。图 2.5 表示的是 RGB 三通道彩色图像数据从存储，每个像素用三个字节表示 BijGijRij（注意，OpenCV 中 RGB 图像的通道顺序是 BGR）。

$I_{0\,0}$	$I_{0\,1}$	...	$I_{0\,N-1}$
$I_{1\,0}$	$I_{1\,1}$	...	$I_{1\,N-1}$
...	...	...	...
$I_{M-1\,0}$	$I_{M-1\,1}$	...	$I_{M-1\,N-1}$

图 2.4 灰度图像数据存储顺序示意图

B_{00}	G_{00}	R_{00}	B_{01}	G_{01}	R_{01}	...	B_{0N-1}	G_{0-1}	R_{0N-1}
B_{10}	G_{10}	R_{10}	B_{11}	G_{11}	R_{11}	...	B_{1N-1}	G_{1N-1}	R_{1N-1}
...	...	...	...	...	...	...	...	...	...
B_{M-10}	G_{M-10}	R_{M-10}	B_{M-11}	G_{M-11}	R_{M-11}	...	B_{M-1N-1}	G_{M-1N-1}	R_{M-1N-1}

图 2.5 彩色图像数据存储顺序示意图

2.2 Mat 类

上面提到，数字图像可以用矩阵来表示。OpenCV 的 Mat 类是一个优秀的表示图像的类，同时也是一个通用的表示矩阵的类。

Mat 类的定义和关键属性如下所示：

```cpp
class CV_EXPORTS Mat
{
public:
    //! 一系列函数
    ...

    /*! flags参数包含一些关于矩阵的信息:
        - 矩阵标识
        - 数据是否连续
        - 深度
        - 通道数目
     */
    int flags;
    //! 矩阵的维数，取值大于或等于2
    int dims;
    //! 矩阵的行数和列数，如果矩阵超过2维，这两个变量的值都为-1
    int rows, cols;
    //! 指向数据的指针
    uchar* data;

    //! helper fields used in locateROI and adjustROI
    const uchar* datastart;
    const uchar* dataend;
    const uchar* datalimit;

    //! 其他成员变量和成员函数
    ...

    //! 与UMat的交互
    UMatData* u;
}
```

2.2.1 创建 Mat 对象

Mat 类提供了一系列创建 Mat 对象的方法，可以根据需要选择使用。

使用构造函数

- 无参数构造函数
  Mat::Mat()

- 创建指定大小、类型为 type 的图像
  Mat::Mat(int rows, int cols, int type)
  Mat::Mat(Size size, int type)
  Mat::Mat(intndims, constint* sizes, int type)
  Mat::Mat(conststd::vector<int>& sizes, int type)
  有四种方式指定图像大小：
  1. 通过设定 int rows 和 int cols 指定行数和列数
  2. 通过 Size(cols, rows)指定行数和列数
  3. 通过 intndims 和 int* sizes 设定维数和每个维度的形状指定图像尺寸
  4. 通过容器 vector<int>& sizes 设定维数和每个维度的形状指定图像尺寸

- 创建指定大小、类型为 type、所有元素都初始化为值 s 的图像
  Mat::Mat(int rows, int cols, int type, const Scalar&s)
  Mat::Mat(Size size, int type, const Scalar&s)
  Mat::Mat(intndims, constint* sizes, int type, const Scalar& s)
  Mat::Mat(conststd::vector<int>& sizes, int type, const Scalar& s)

- 创建指定大小、类型为 type、行步长为 step 的图像
  Mat::Mat(int rows, int cols, int type, void* data, size_t step=AUTO_STEP)
  Mat::Mat(Size size, int type, void* data, size_t step=AUTO_STEP)
  Mat::Mat(intndims, constint* sizes,int type, void* data, constsize_t* steps=0)
  Mat::Mat(conststd::vector<int>& sizes, int type, void* data, constsize_t* stpes=0)
  行步长是指矩阵每行所占的字节数，包含在每行末为对齐而添加的字节。如果 step 设定为 AUTO_STEP 或 0 则表示不计算补齐的字节数。此构造函数不创建图像数据所需的内存，而是直接使用 data 所指的内存。

- 创建大小、类型与 m 相同的图像
  Mat::Mat(const Mat& m)
  此构造函数不会复制图像数据，新的图像与 m 共享图像数据，引用计数增加。所以，如果修改新创建的这个矩阵，m 也被修改了。

- 创建大小、类型与指定 m 部分相同的图像
  Mat::Mat(const Mat& m, const Range&rowRange, const Range&colRange=Range::all())
  Mat::Mat(const Mat& m, constRect&roi)
  Mat::Mat(const Mat& m, const Range* ranges)
  Mat::Mat(const Mat& m, conststd::vector<Range>& ranges)
  此构造函数不会复制图像数据，新图像与指定的 m 部分共享图像数据，引用计数增加。所以，如果修改新创建的这个矩阵，m 的内容也被修改了。
  有四种方式指定 m 区域：
  1. 通过 rowRange 和 colRange 指定
  2. 通过 Rect 指定
  3. 通过 Range 的数组指定
  4. 通过 Range 的容器指定

上面的构造函数参数中，很多都涉及到类型 type。Type 可以是 CV_8UC1, CV_16SC1, …, CV_64FC4 等。其中，8U 表示 8 位无符号整数，16S 表示 16 位有符号整数，64F 表示 64 位浮点数（即 double 类型）。CX 表示通道数，例如 C1 表示单通道图像，C4 表示 4 通道图像，以此类推。如果 type 中没有 CX，则默认 C1 单通道图像。

下面的代码为创建一个 Mat 对象并输出其内容：

```
01.  // 创建行数为3、列数为2、类型为8位无符号整型、所有元素值都
02.  // 被初始化为(0, 0, 255)的三通道图像
03.  Mat M(3, 2, CV_8UC3, Scalar(0, 0, 255));
04.  // 输出M
05.  cout << "M = " << endl << " " << M << endl;
```

```
M =
 [  0,   0, 255,   0,   0, 255;
    0,   0, 255,   0,   0, 255;
    0,   0, 255,   0,   0, 255]
```

另外，如果需要创建更多通道数的图像，可以使用宏 CV_8UC(n)可以定义，如：

```
01.  // 创建行数为3、列数为2、通道数为5的图像
02.  Mat M(3, 2, CV_8UC(5));
```

使用 create()函数

Mat 类的 create()函数也可以创建图像，它是 Mat 类的一个重要方法。

Mat::create(int rows, int cols, int type)

Mat::create(Size size, int type)

Mat::create(intndims, constint* sizes, int type)

Mat::create(conststd::vector<int>& sizes, int type)

OpenCV 的很多函数和方法均调用 create()创建输出矩阵。如果 create()函数指定的参数与当前的图像参数相同，则不进行内存申请，如果参数不同，则会减少当前图像数据内存的索引，并重新申请内存。例如：

```
01. // 构造函数创建图像
02. Mat M(3, 2, CV_8UC3);
03. // 释放内存重新创建图像
04. M.create(2, 2, CV_8UC2);
```

需要注意的是，create()函数不能设置图像像素的初始值。

使用 Matlab 风格的函数

OpenCV 也提供了如下一些 Matlab 风格的静态成员函数，使用很方便，也可以使代码简洁：

Mat::zeros(int rows, int cols, int type),

Mat::zeros(Size size, int type)

Mat::zeros(intndims, constint* sz, int type)

Mat::ones(int rows, int cols, int type),

Mat::ones(Size size, int type),

Mat::ones(intndims, constint* sz, int type)

Mat::eye(int rows, int cols, int type)

Mat::eye(Size size, int type)

使用方法可以为：

```
01. Mat Z = Mat::zeros(2, 3, CV_8UC1);
02. cout << "Z = " << endl << " " << Z << endl;
```

```
Z =
 [  0,   0,   0;
   0,   0,   0]
```

```
01. Mat O = Mat::ones(2, 3, CV_32F);
02. cout << "O = " << endl << " " << O << endl;
```

```
O =
 [1, 1, 1;
 1, 1, 1]
```

```
01. Mat E = Mat::eye(2, 3, CV_64F);
02. cout << "E = " << endl << " " << E << endl;
```

```
E =
 [1, 0, 0;
 0, 1, 0]
```

2.2.2 矩阵元素的表达

对于单通道图像，其元素类型一般为 8U（即 8 位无符号整数），也可以为 16S、32F 等，这些类型可以用 uchar、short、float 等 C/C++语言中的基本数据类型表达，如表 2.1 所示。

对于多通道图像，如 RGB 彩色图像，也可以将图像看作一个二维矩阵，但此时矩阵的元素不再是基本的数据类型。OpenCV 中有一个模板类 Vec，可以用来表示一个向量。OpenCV 使用 Vec 类预定义了如下一些向量，这些向量可以用于表达多通道图像的元素类型。

```
typedef Vec<uchar, 2> Vec2b;
typedef Vec<uchar, 3> Vec3b;
typedef Vec<uchar, 4> Vec4b;

typedef Vec<short, 2> Vec2s;
typedef Vec<short, 3> Vec3s;
typedef Vec<short, 4> Vec4s;

typedef Vec<int, 2> Vec2i;
typedef Vec<int, 3> Vec3i;
typedef Vec<int, 4> Vec4i;

typedef Vec<float, 2> Vec2f;
typedef Vec<float, 3> Vec3f;
typedef Vec<float, 4> Vec4f;
typedef Vec<float, 6> Vec6f;

typedef Vec<double, 2> Vec2d;
typedef Vec<double, 3> Vec3d;
typedef Vec<double, 4> Vec4d;
typedef Vec<double, 6> Vec6d;
```

例如，CV_8UC3 图像的元素类型可以使用 Vec3b，CV_32FC4 图像的素类型可以使用 Vec4f。对于 Vec 对象，可以使用[]符号读写其元素，类似操作数组：

```
Vec3b color;     // 描述RGB颜色的color变量
color[0] = 255;  // B分量
color[1] = 0;    // G分量
color[2] = 0;    // R分量
```

2.2.3 像素的读写

进行图像处理时可需要读取或者设置某个像素的值，也可能需要对图像所有像素进行遍历。OpenCV 提供了不同的方法实现图像遍历。

使用 at() 函数

at()函数是一个模板函数，它返回指定矩阵元素的引用。at() 的调用方式如表 2.1 所示。

表 2.1 at()函数的调用方式

元素类型	调用方式
CV_8U	Mat.at<uchar>(i, j)
CV_8S	Mat.at<schar>(i, j)
CV_16U	Mat.at<ushort>(i, j)
CV_16S	Mat.at<short>(i, j)
CV_32S	Mat.at<int>(i, j)
CV_32F	Mat.at<float>(i, j)
CV_64F	Mat.at<double>(i, j)

下面两行代码演示了如何使用 at()函数。

```
// 获取灰度图第i行第j列像素值
uchar value = grayImg.at<uchar>(i, j);
// 设置图像第i行第j列像素值为128
grayImg.at<uchar>(i, j) = 128;
```

如果要对整幅图像进行遍历，可以参考下面的例程。这个例程先创建两幅图像，单通道的 grayImg 和 3 通道的 colorImg，然后对这两幅图像所有的像素值进行赋值。

```
#include <iostream>
#include "opencv2/opencv.hpp"

using namespace std;
using namespace cv;

int main(int argc, char** argv)
{
    Mat grayImg(480, 640, CV_8UC1);
    Mat colorImg(480, 640, CV_8UC3);

    // 遍历灰度图像所有像素，并设置像素值
    for (int i = 0; i < grayImg.rows; i++)
        for (int j = 0; j < grayImg.cols; j++)
            grayImg.at<uchar>(i, j) = (i + j) % 255;

    // 遍历彩色图像所有像素，并设置像素值
    for (int i = 0; i < colorImg.rows; i++)
        for (int j = 0; j < colorImg.cols; j++)
        {
            Vec3b pixel;
            pixel[0] = i % 255; // Blue
            pixel[1] = j % 255; // Green
            pixel[2] = 0;       // Red
            colorImg.at<Vec3b>(i, j) = pixel;
        }

    // 显示结果
    imshow("gray image", grayImg);
    imshow("color image", colorImg);
    waitKey(0);

    return 0;
}
```

运行结果如图 2.6 所示。

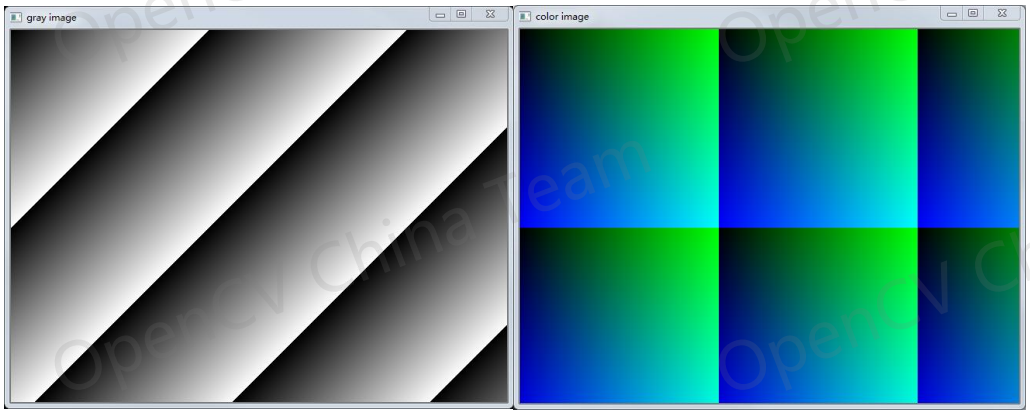

图 2.6 使用 at()函数遍历图像例程运行结果

需要注意的是，使用 at()函数的优点是代码的可读性高，但是函数效率并不是很高。所以不推荐使用 at()函数遍历图像。另外，at()函数只有在 Debug 模式下才进行越界检查。

使用迭代器

Mat 类支持使用 C++ STL 库的迭代器（iterator）进行元素遍历。以下代码演示了如何使用迭代器遍历图像像素。

```
#include <iostream>
#include "opencv2/opencv.hpp"

using namespace std;
using namespace cv;

int main(int argc, char** argv)
{
    Mat grayImg(480, 640, CV_8UC1);
    Mat colorImg(480, 640, CV_8UC3);

    // 遍历灰度图像所有像素，并设置像素值
    MatIterator_<uchar> grayItr, grayEnd;
    for (grayItr = grayImg.begin<uchar>(),
        grayEnd = grayImg.end<uchar>();
        grayItr != grayEnd; grayItr++)
        *grayItr = rand() % 255;

    // 遍历彩色图像所有像素，并设置像素值
    MatIterator_<Vec3b> colorItr, colorEnd;
    for (colorItr = colorImg.begin<Vec3b>(),
         colorEnd = colorImg.end<Vec3b>();
         colorItr != colorEnd; colorItr++)
    {
        (*colorItr)[0] = rand() % 255; // Blue
        (*colorItr)[1] = rand() % 255; // Green
        (*colorItr)[2] = rand() % 255; // Red
    }

    // 显示结果
    imshow("gray image", grayImg);
    imshow("color image", colorImg);
    waitKey(0);

    return 0;
}
```

运行结果如图 2.7 所示。

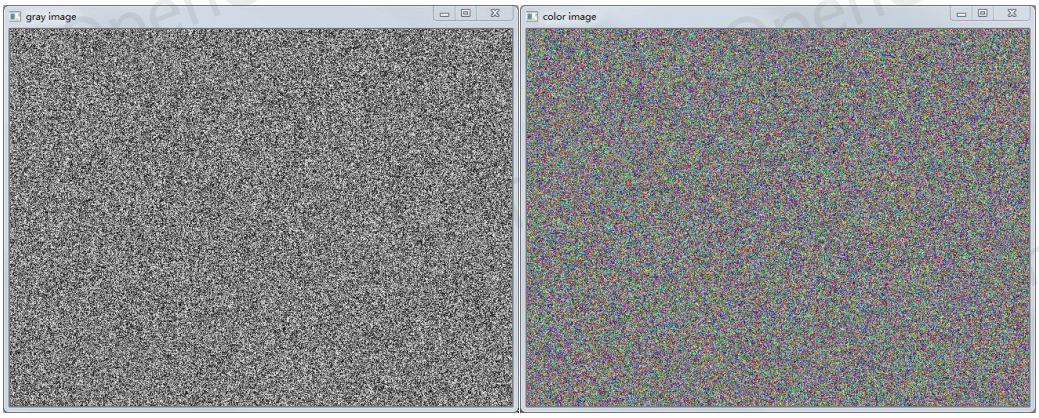

图 2.7 使用迭代器遍历图像例程运行结果

使用指针

遍历矩阵元素还可以使用指针访问。例如：

```cpp
#include <iostream>
#include "opencv2/opencv.hpp"

using namespace std;
using namespace cv;

int main(int argc, char** argv)
{
    Mat grayImg(480, 640, CV_8UC1);
    Mat colorImg(480, 640, CV_8UC3);

    // 遍历灰度图像所有像素，并设置像素值
    for (int i = 0; i < grayImg.rows; i++)
    {
        // 获取第i行首像素指针
        uchar* p = grayImg.ptr<uchar>(i);
        // 对第i行的每个像素（一个字节）赋值
        for (int j = 0; j < grayImg.cols; j++)
            p[j] = (i + j) % 255;
    }

    // 遍历彩色图像所有像素，并设置像素值
    for (int i = 0; i < grayImg.rows; i++)
    {
        // 获取第i行首像素指针
        Vec3b* p = colorImg.ptr<Vec3b>(i);
        // 对第i行的每个像素（三个字节）赋值
        for (int j = 0; j < grayImg.cols; j++)
        {
            p[j][0] = i % 255; // Blue
            p[j][1] = j % 255; // Green
            p[j][2] = 0;       // Red
        }
    }

    // 显示结果
    imshow("gray image", grayImg);
    imshow("color image", colorImg);
    waitKey(0);

    return 0;
}
```

运行结果如图 2.8 所示。

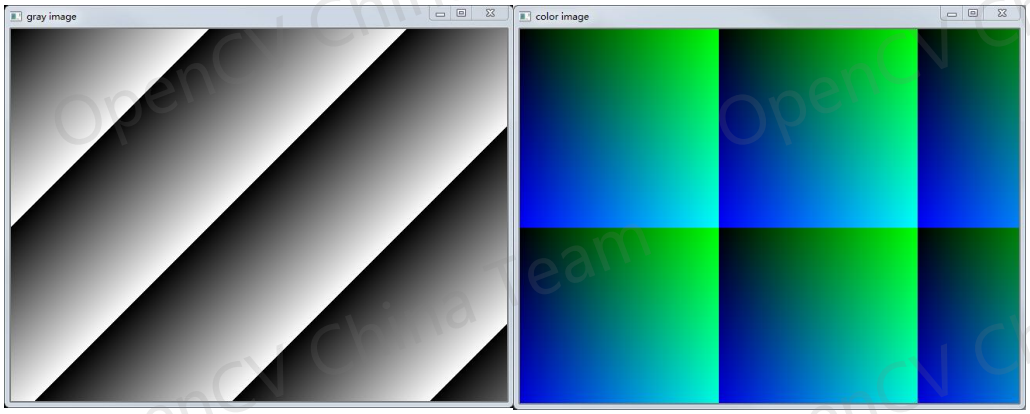

图 2.8 使用指针遍历图像例程运行结果

注意，C/C++中的指针操作并不进行类型和越界检查。如果指针访问出错，程序运行时有时候可能看上去正常，有时候可能突然出现“段错误”（segment fault）。当程序规模比较大逻辑比较复杂时，查找指针错误会非常困难。如果对指针使用不熟悉，并不建议直接通过指针操作访问像素。但是，如果程序的运行速度非常重要，还是建议使用指针进行遍历。

2.2.4 选取图像局部区域

Mat 类提供了多种方法选取图像局部区域。需要注意的是，这些方法并不进行内存的复制操作，即将局部区域赋值给新的 Mat 对象时，新对象只是与原始对象共享相同的数据区域，而不新申请内存。这也正是这些方法执行速度比较快的原因。

选取单行或单列

Mat::row(int y) const

Mat::col(int x) const

参数 y 和 x 分别是行索引和列索引。例如：

```
// 取出A矩阵的第i行
Mat line = A.row(i);

// 取出A矩阵的第i行，将这一行所有元素乘以2后，
// 赋值给第j行
A.row(j) A.row(i) * 2;
```

选取多行或多列

通过使用 Range 类，可以选取矩阵的多行或多列。Range 类是 OpenCV 的一个基本数据类型，其定义如下：

```
class CV_EXPORTS Range
{
public:
    ...

    static Range all();

    int start, end;
};
```

它有两个重要变量 star 和 end，表示的范围为从 start 到 end，包含 start 但不包含 end。Range 类还有一个静态方法 all()，可直接选取所有行或者所有列，作用类似 Matlab 中的“：”。

下面的代码演示了如何使用 Range 选取矩阵的多行或多列：

```
// 创建一个单位阵
Mat A = Mat::eye(10, 10, CV_32S);
// 提取第1到3列（不包括第3列）
Mat B = A(Range::all(), Range(1, 3));
// 提取B的第5到9行（不包括第9行）
Mat C = B(Range(5, 9), Range::all());
```

上面创建 C 的代码其实等价于：

```
Mat A = Mat::eye(10, 10, CV_32S);
Mat C = A(Range(5, 9), Range(1, 3));
```

选取感兴趣区域

有两种方法可以从图像中选取感兴趣区域（ROI，Region of Interest）

1. 使用构造函数，如以下代码所示：

```
// 创建宽度为320、高度为240的3通道图像
Mat img(Size(320, 240), CV_8UC3);
// 选取img中Rect(10, 10, 100, 100)的区域为ROI
// 使用构造函数
Mat roi(img, Rect(10, 10, 100, 100));
Mat roi2(img. Range(10, 100), Range(10, 100));
```

2. 使用括号运算符，如以下代码所示：

```
// 使用括号运算符
Mat roi3 = img(Rect(10, 10, 100, 100));
Mat roi4 = img(Range(10, 100), Range(10, 100));
```

选取对角线

Mat 类的 diag()函数由于获取矩阵的对角线元素。

Mat::diag(int d = 0) const

参数 d 的取值与所选取的对角线的对应关系如表 2.2 所示：

表 2.2 d 值与对角线位置

参数 d	对角线
d=0	取主对角线
d>0	取主对角线上方的其它对角线。例如，d=1 为取主对角线上方紧挨主对角线的对角线，以此类推。
d<0	取主对角线下方的其它对角线。例如，d=-1 为取主对角线下方紧挨主对角线的对角线，以此类推。

下面的代码和输出演示了使用 diag()函数选取矩阵的对角线：

```
// 创建一个4x4的矩阵，并初始化其元素
Mat M = (Mat_<int>(4, 4) <<
    1, 2, 3, 0,
    4, 5, 6, 0,
    7, 8, 9, 0,
    10,11,12,0);

// 取主对角线
Mat d0 = M.diag(0);
cout << "d0 = " << endl << " " << d0 << endl;
// 取主对角线上方第2条对角线
Mat d1 = M.diag(2);
cout << "d1 = " << endl << " " << d1 << endl;
// 取主对角线下方第1条对角线
Mat d2 = M.diag(-1);
cout << "d2 = " << endl << " " << d2 << endl;
```

```
d0 =
 [1;
 5;
 9;
 0]
```

```
d1 =
 [3;
 0]
```

```
d2 =
 [4;
 8;
 12]
```

此外，如同之前的选取区域的函数和方法，diag()函数也不进行内存复制操作，其复杂度是 O(1)。

2.2.5 输出

Mat 类重载了<<操作符，使得可以使用流操作输出矩阵内容，并支持不同的输出格式。

默认格式

```
// 创建一个3x2的矩阵，并随机初始化其元素
Mat R = Mat(3, 2, CV_8UC3);
randu(R, Scalar::all(0), Scalar::all(255));

// 默认输出格式
cout << "R (default) = " << endl << R << endl;
```

```
R (default) =
[ 91,   2,  79, 179,  52, 205;
 236,   8, 181, 239,  26, 248;
 207, 218,  45, 183, 158, 101]
```

Python 格式

```
// Python输出格式
cout << "R (python) = " << endl << format(R, Formatter::FMT_PYTHON) << endl;
```

```
R (python) =
[[[ 91,   2,  79], [179,  52, 205]],
 [[236,   8, 181], [239,  26, 248]],
 [[207, 218,  45], [183, 158, 101]]]
```

Numpy 格式

```
// Numpy输出格式
cout << "R (numpy) = " << endl << format(R, Formatter::FMT_NUMPY) << endl;
```

```
R (numpy) =
array([[[ 91,   2,  79], [179,  52, 205]],
       [[236,   8, 181], [239,  26, 248]],
       [[207, 218,  45], [183, 158, 101]]], dtype='uint8')
```

C 语言格式

```
// C语言输出格式
cout << "R (c) = " << endl << format(R, Formatter::FMT_C) << endl;
```

```
R (c) =
{ 91,   2,  79, 179,  52, 205,
 236,   8, 181, 239,  26, 248,
 207, 218,  45, 183, 158, 101}
```

CSV 格式

```
// CSV输出格式
cout << "R (csv) = " << endl << format(R, Formatter::FMT_CSV) << endl;
```

```
R (csv) =
 91,   2,  79, 179,  52, 205
236,   8, 181, 239,  26, 248
207, 218,  45, 183, 158, 101
```

除了 Mat 对象，OpenCV 也支持其它类型使用“<<”流操作输出内容。如：

```
// 二维点
Point2f P(5, 1);
cout << "Point (2d) = " << P << endl;
```

```
Point (2d) = [5, 1]
```

```
// 三维点
Point3f P(5, 1, 8);
cout << "Point (3d) = " << P << endl;
```

```
Point (3d) = [5, 1, 8]
```

2.2.6 Mat 表达式

利用 C++中的运算符重载，OpenCV 提供了 Mat 运算表达式，这使得在使用 C++编程时如同写 Matlab 脚本，简洁易懂，便于维护。

Mat 表达式支持如下运算（A、B 是 Mat 类型对象，s 表示 Scalar 对象，alpha 表示 double 值）：

- 加法、减法、取负：A+B，A-B，A+s，A-s，s+A，s-A，-A
- 缩放取值范围：A*alpha
- 对应元素的乘法和除法：A.mul(B)，A/B，alpha/A
- 矩阵乘法：A*B
- 转置：A.t()
- 求逆和求伪逆：A.inv()
- 比较运算：A cmpop B，A cmpop alpha，alpha compop A （cmpop 可以是>，>=，==，!=，<）。如果条件成立，结果矩阵的对应元素被置为 255，否则置为 0。
- 位逻辑运算：A logicop B，A logicop s，s logicop A，~A （logicop 可以是&，|和^）
- 对应元素的最大值和最小值：min(A, B)，min(A, alpha)，max(A, B)，max(A, alpha)
- 元素的绝对值：abs(A)
- 叉乘和点乘：A.cross(B)，A.dot(B)

下面的例程展示了 Mat 表达式的使用方法：

```
01.  #include <iostream>
02.  #include "opencv2/opencv.hpp"
03.
04.  using namespace std;
05.  using namespace cv;
06.
07.  int main(int argc, char** argv)
08.  {
09.      Mat A = Mat::eye(4, 4, CV_32SC1);
10.      Mat B = A * 3 + 1;
11.      Mat C = B.diag(0) + B.col(1);
12.
13.      cout << "A = " << A << endl << endl;
14.      cout << "B = " << B << endl << endl;
15.      cout << "C = " << C << endl << endl;
16.      cout << "C .* diag(B) = " << C.dot(B.diag(0)) << endl;
17.
18.      return 0;
19.  }
```

上面例程的输出结果如下：

```
A = [1, 0, 0, 0;
 0, 1, 0, 0;
 0, 0, 1, 0;
 0, 0, 0, 1]

B = [4, 1, 1, 1;
 1, 4, 1, 1;
 1, 1, 4, 1;
 1, 1, 1, 4]

C = [5;
 8;
 5;
 5]

C .* diag(B) = 92
```

2.3 Mat_类

从前面的章节中可以看到，读写矩阵元素时需要指定数据类型，例如 Mat.at<uchar>(i, j)。

这样需要不停地写<uchar>，很繁琐，甚至有时还可能出错，如下面的代码：

```
    Mat grayImg(480, 640, CV_8UC1);
    for (int i = 0; i < grayImg.rows; i++)
    {
        // 获取第i行首像素指针
        // 需要指定类型
        uchar* p = grayImg.ptr<uchar>(i);
        for (int j = 0; j < grayImg.cols; j++)
        {
            double d1 = (double)((i + j) % 255);
            // 用at()读写像素时，需要指定类型
            grayImg.at<uchar>(i, j) = d1;

            // 下面的代码错误，应该使用at<uchar>()
            // 但编译时不会提醒错误
            // 运行时结果不正确，d2不等于d1
            double d2 = grayImg.at<double>(i, j);
        }
    }
```

这种错误不是语法错误，编译时编译器不会提醒。程序运行时，at()函数获取到的不是期望的(i,j)位置上的元素，因为数据已经越界，但程序未必会报错。这类的错误使得程序忽而看上去正常，忽而弹出“段错误”，当代码规模很大时，会难以查错。

Mat_类是对 Mat 类的一个轻量级包装，它是一个模板类，使得访问元素时可以不需要指定元素类型，即使得代码简洁，又减少了出错的可能性。下面的例程展示了如何使用 Mat_ 类简化代码，减少出错。

```
#include <iostream>
#include "opencv2/opencv.hpp"

using namespace std;
using namespace cv;

int main(int argc, char** argv)
{
    /********** 使用Mat **********/
    Mat grayImg(480, 640, CV_8UC1);
    for (int i = 0; i < grayImg.rows; i++)
    {
        // 获取第i行首像素指针
        // 需要指定类型
        uchar* p = grayImg.ptr<uchar>(i);
        for (int j = 0; j < grayImg.cols; j++)
        {
            double d1 = (double)((i + j) % 255);
            // 用at()读写像素时，需要指定类型
            grayImg.at<uchar>(i, j) = d1;

            // 下面的代码错误，应该使用at<uchar>()
            // 但编译时不会提醒错误
            // 运行时结果不正确，d2不等于d1
            double d2 = grayImg.at<double>(i, j);
        }
    }

    /********** 使用Mat_ **********/
    // 在变量声明时指定矩阵元素类型
    Mat_<uchar> grayImg1 = (Mat_<uchar>&)grayImg;
    for (int i = 0; i < grayImg1.rows; i++)
    {
        // 获取第i行首像素指针
        // 不需要指定元素类型，语句简洁
        uchar* p = grayImg1.ptr(i);
        for (int j = 0; j < grayImg1.cols; j++)
        {
            double d1 = (double)((i + j) % 255);
            // 直接使用Matlab风格的矩阵元素读写，简洁
            grayImg1(i, j) = d1;
            double d2 = grayImg1(i, j);
        }
    }

    return 0;
}
```

2.4 Mat 类的内存管理

Mat 类由两个数据部分组成：矩阵头和一个指向存储所有像素值的内存的指针 uchar* data，如图 2.9 所示。矩阵头的尺寸是常数值，但是存储矩阵元素的内存会随图像的不同而不同，通常比矩阵头的尺寸大数个数量级。

复制矩阵数据往往花费较多时间，尤其是大矩阵。为了解决矩阵数据的传递，OpenCV 使用了引用计数机制。其思路是让每个 Mat 对象有自己的矩阵头信息，但多个 Mat 对象可以共享同一个矩阵数据，即让矩阵指针 data 指向同一地址，同时引用计数增加。OpenCV 中很多函数及很多操作（如函数参数传值）只复制矩阵头信息，而不复制矩阵数据。

…	…
int	flags
int	dims
int	rows
int	cols
uchar*	data
…	…

图 2.9 Mat 类的构成

关于多个矩阵对象共享同一矩阵数据，可以看下面的简单例子：

```
01. Mat A(100, 100, CV_8UC1);
02.
03. Mat B = A;
04.
05. Mat C = A(Rect(50, 50, 30, 30));
```

上面代码中有三个 Mat 对象，分别是 A，B 和 C。这三个矩阵共享同一矩阵数据，示意图如图 2.10 所示。

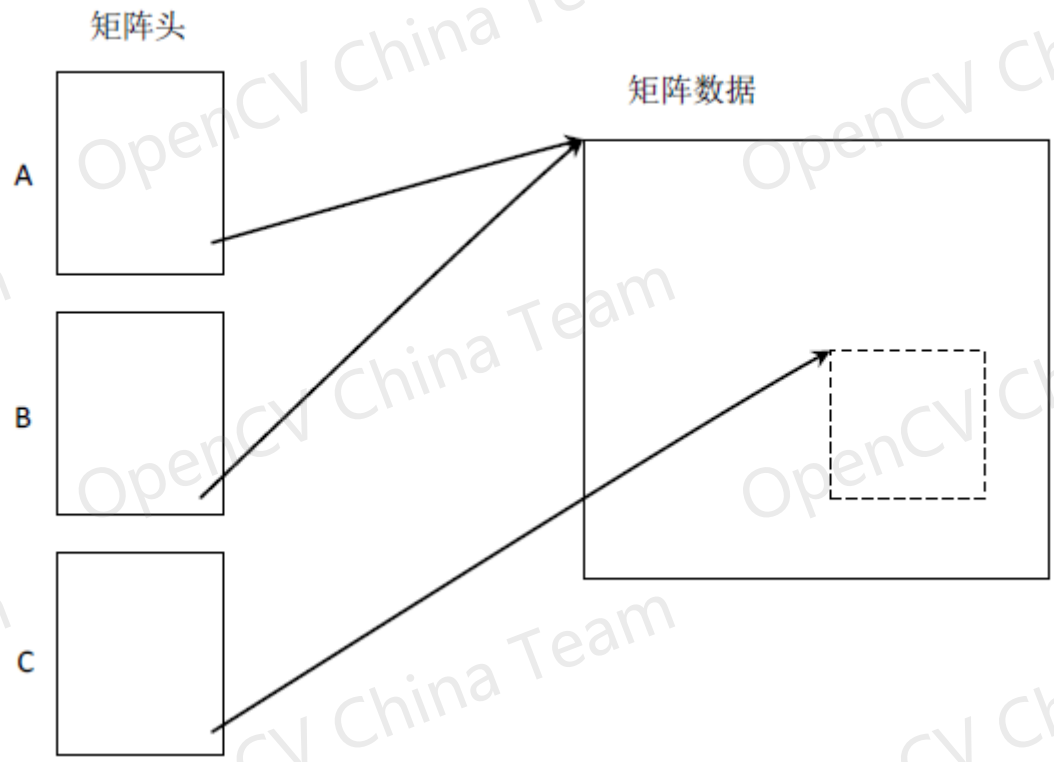

图 2.10 三个矩阵头共用同一矩阵数据

输出 A、B、C 的引用计数：

```
01.  Mat A(100, 100, CV_8UC1);
02.  cout << "A: " << A.u->refcount << endl;
03.
04.  Mat B = A;
05.  cout << "A: " << A.u->refcount << endl;
06.
07.  Mat C = A(Rect(50, 50, 30, 30));
08.  cout << "A: " << A.u->refcount << endl;
09.  cout << "B: " << B.u->refcount << endl;
10.  cout << "C: " << C.u->refcount << endl;
```

```
A: 1
A: 2
A: 3
B: 3
C: 3
```

2.5 标注图像

OpenCV 提供了一些简单但重要的绘制函数，在图像上进行标记或标注，可以方便的显示算法的结果。如图 2.11，进行目标检测后，通常需要在图像上用矩形框框出物体显示物体位置并表示出物体的类别及其他信息。

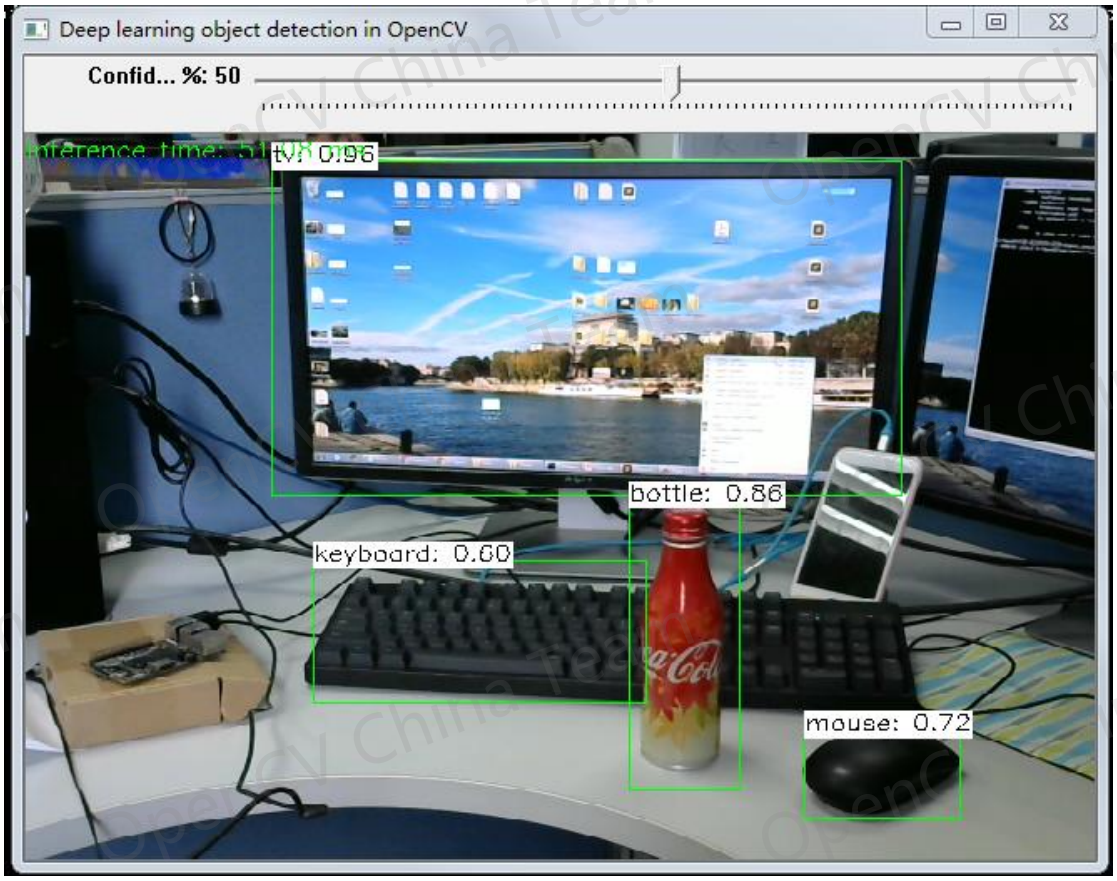

图 2.11 在图像上绘制显示目标检测结果

OpenCV 中常用的绘制函数有：

绘制直线

voidcv::line(InputOutputArrayimg, Point pt1, Point pt2, const Scalar& color, int thickness=1, intlineType=LINE_8, int shift=0)

img 为需要绘制标记的图像；pt1 和 pt2 分别为直线起点和终点的坐标；color 为直线的颜色；thickness 为直线的宽度；lineType 为直线的类型；shift 为 Number of fractional bits in the point coordinates。

绘制圆

void cv::circle(InputOutputArrayimg, Point center, int radius, const Scalar& color, int thickness=1, intlineType=LINE_8, int shift=0)

img 为需要绘制标记的图像；center 为圆心坐标；radius 为圆半径；color 为圆的颜色；thickness 为圆的线条宽度；lineType 为圆的线条类型；shift 为 Number of fractional bits in the point coordinates。

绘制矩形

voidcv::rectangle(InputOutputArrayimg, Point pt1, Point pt2, const Scalar& color, int thickness=1, intlineType=LINE_8, int shift=0)

img 为需要绘制标记的图像；pt1 为矩形左上角坐标， pt2 为矩形右下角坐标；color 为矩形边颜色；thickness 为矩形边的宽度；lineType 为矩形边的类型；shift 为坐标小数位数。

void cv::rectangle(InputOutputArrayimg, Rectrect, const Scalar& color, int thickness=1,

intlineType=LINE_8, int shift=0)

img 为需要绘制标记的图像；rect 为所要绘制的矩形；color 为矩形边颜色；thickness 为矩形边的宽度；lineType 为矩形边的类型；shift 为 Number of fractional bits in the point coordinates。

绘制文字

void cv::putText(InputOutputArrayimg, const String& text, Point org, intfontFace, double fontScale, Scalar color, int thickness=1, intlineType=LINE_8, bool bottomLeftOrigin=false)

img 为需要绘制标记的图像；text 为要显示的文字；org 为文字左下角的坐标；fontFace 为字体；fontScale 为相对于字体基础大小的缩放比例；color 为文字的颜色；thickness 为文字线条的宽度；lineType 为文字线条的类型；bottomLeftOrigin 为 true 时，图像坐标的原点在左下角，bottomLeftOrigin 为 false 时，图像坐标的原点在左上角。

下面的代码演示了如何使用这些绘制函数在图像上做标记：

```
#include <iostream>
#include "opencv2/opencv.hpp"

using namespace std;
using namespace cv;

int main(int argc, char** argv)
{
    Mat img = imread("flower.jpg");

    // 绘制绿色直线
    line(img, Point(300, 300), Point(450, 100), Scalar(0, 255, 0), 3);

    // 绘制黄色圆
    circle(img, Point(800, 400), 40, Scalar(0, 255, 255), 3);

    // 绘制红色矩形
    rectangle(img, Point(20, 20), Point(60, 100), Scalar(0, 0, 255), 3);
    // 绘制蓝色矩形
    rectangle(img, Rect(30, 30, 40, 80), Scalar(255, 0, 0), 3);

    // 绘制白色文字
    putText(img, "This is rose.", Point(50, 50), FONT_HERSHEY_COMPLEX,
            1.0, Scalar(255, 255, 255), 3);

    //显示图像
    imshow("Anno", img);
    waitKey(0);

    return 0;
}
```

绘制的结果如图 2.12 所示。

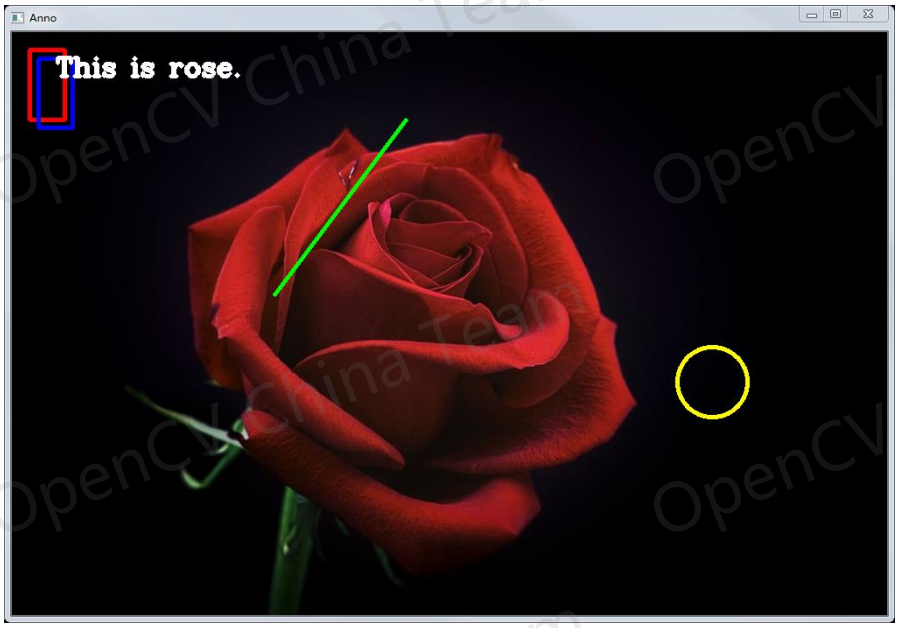

图 2.12 使用 OpenCV 的绘制函数在图像上做标记

2.6 应用

戴墨镜

给出一幅人脸图像 A 图 2.13 和一幅墨镜图像 B 图 2.14，利用前面介绍的知识，给 A 中的人脸带上 B 中的墨镜。

图 2.13 人脸图像

图 2.14 墨镜图像

- 一种简单直接的方法是用墨镜图像直接替换人脸图像的眼睛区域，下面的代码演示了如何实现：

```
#include <iostream>
#include "opencv2/opencv.hpp"

using namespace std;
using namespace cv;

int main(int argc, char** argv)
{
    // 读取人脸图像
    Mat faceImage = imread("musk.jpg");
    // 将图像类型由CV_8UC3转为CV_32FC3
    faceImage.convertTo(faceImage, CV_32FC3);
    // 归一化
    faceImage = faceImage / 255.0;
    imshow("face", faceImage);

    // 读取墨镜图像
    Mat glassPNG = imread("sunglass.png", -1);
    glassPNG.convertTo(glassPNG, CV_32F);
    glassPNG = glassPNG / 255.0;

    // 根据眼睛区域大小缩放墨镜图像
    resize(glassPNG, glassPNG, Size(), 0.5, 0.5);

    // 分离图像的颜色通道和alpha通道
    Mat glassRGBAChannels[4];
    Mat glassRGBChannels[3];
    split(glassPNG, glassRGBAChannels);

    for (int i = 0; i < 3; i++)
        // 复制B,G,R通道
        glassRGBChannels[i] = glassRGBAChannels[i];

    Mat glassBGR, glassMask1;
    // 将复制的分离的3通道图像合为彩色图
    merge(glassRGBChannels, 3, glassBGR);
    imshow("RGB sunglass", glassBGR);

    // Alpha通道图像
    glassMask1 = glassRGBAChannels[3];
    imshow("Alpha sunglass", glassMask1);

    int glassHeight = glassPNG.rows;
    int glassWidth = glassPNG.cols;
    int topLeftRow = 130;
    int topLeftCol = 130;
    int bottomRightRow = topLeftRow + glassHeight;
    int bottomRightCol = topLeftCol + glassWidth;

    // 复制人脸图像
    Mat faceWithGlassesNaive = faceImage.clone();
    // 选取眼睛区域
    Mat roiFace = faceWithGlassesNaive(Range(topLeftRow, bottomRightRow),
                  Range(topLeftCol, bottomRightCol));
    // 用墨镜图像替换眼睛区域图像
    glassBGR.copyTo(roiFace);
    imshow("face with sunglass naive", faceWithGlassesNaive);

    waitKey(0);

    return 0;
}
```

代码中的墨镜的彩色图像和 alpha 通道图像如图 2.15 所示。

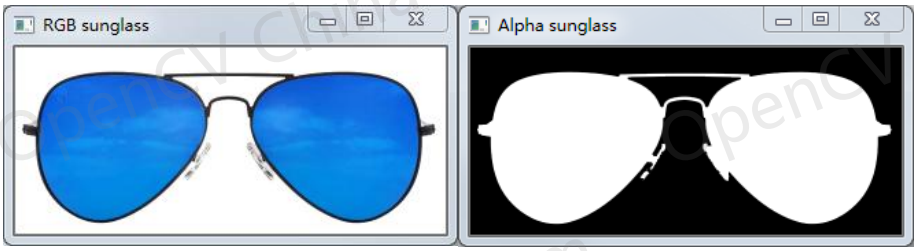

图 2.15(a)墨镜彩色图像　　(b)墨镜 alpha 通道图像

用墨镜图像直接替换眼睛区域的效果如图 2.16 所示。

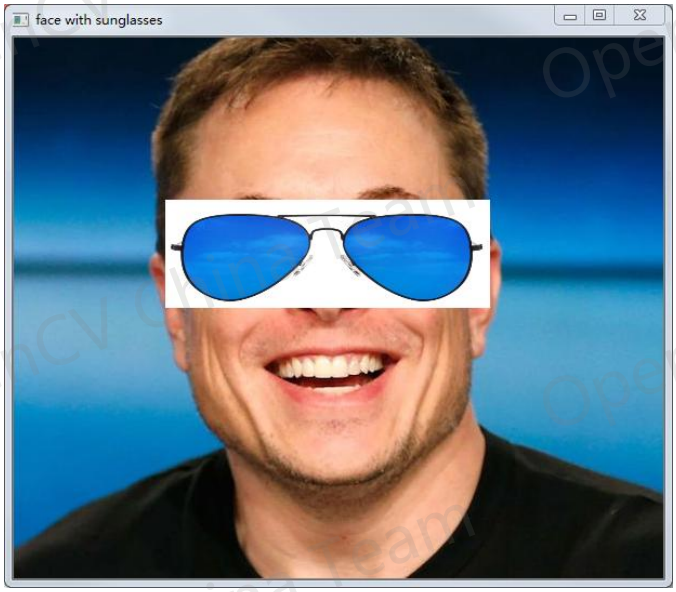

图 2.16 戴墨镜效果图

- 另一种自然效果的实现方法如下：

```cpp
#include <iostream>
#include "opencv2/opencv.hpp"

using namespace std;
using namespace cv;

int main(int argc, char** argv)
{
    ...
    // 之前的代码与第一种方法相同
    // 因为人脸图像是3通道图像，也生成3通道的mask图像
    Mat glassMask;
    Mat glassMaskChannels[] = { glassMask1,glassMask1,glassMask1 };
    merge(glassMaskChannels, 3, glassMask);

    // 复制人脸图像
    Mat faceWithGlassesArithmetic = faceImage.clone();
    // 选取眼睛区域
    Mat roiFace = faceWithGlassesArithmetic(Range(topLeftRow, bottomRightRow),
                  Range(topLeftCol, bottomRightCol));

    Mat eyeROIChannels[3];
    split(roiFace, eyeROIChannels);
    Mat maskedEyeChannels[3];
    Mat maskedEye;

    for (int i = 0; i < 3; i++)
        // 根据墨镜mask图像生成眼睛区域的mask图像
        multiply(eyeROIChannels[i], (1 - glassMaskChannels[i]), maskedEyeChannels[i]);
    merge(maskedEyeChannels, 3, maskedEye);
    imshow("masked eye region", maskedEye);

    Mat maskedGlass;
    // 根据mask图像生成掩膜的墨镜区域
    multiply(glassBGR, glassMask, maskedGlass);
    imshow("masked glass", maskedGlass);

    Mat eyeRoiFinal;
    // 将墨镜和眼睛区域合并获得最终效果
    add(maskedEye, maskedGlass, eyeRoiFinal);
    imshow("eye with sunglass", eyeRoiFinal);

    // 将戴墨镜的眼睛图像复制到人脸图像上
    eyeRoiFinal.copyTo(roiFace);
    imshow("face with sunglass", faceWithGlassesArithmetic);

    waitKey(0);

    return 0;
}
```

代码的中间结果图如图 2.17 所示。

(a) 眼睛区域镜片 mask 图像　(b) 镜片外区域 mask 图像　(c) a + b

图 2.17 中间结果图示

最后的戴墨镜效果如图 2.18 所示。

图 2.18 戴墨镜效果图