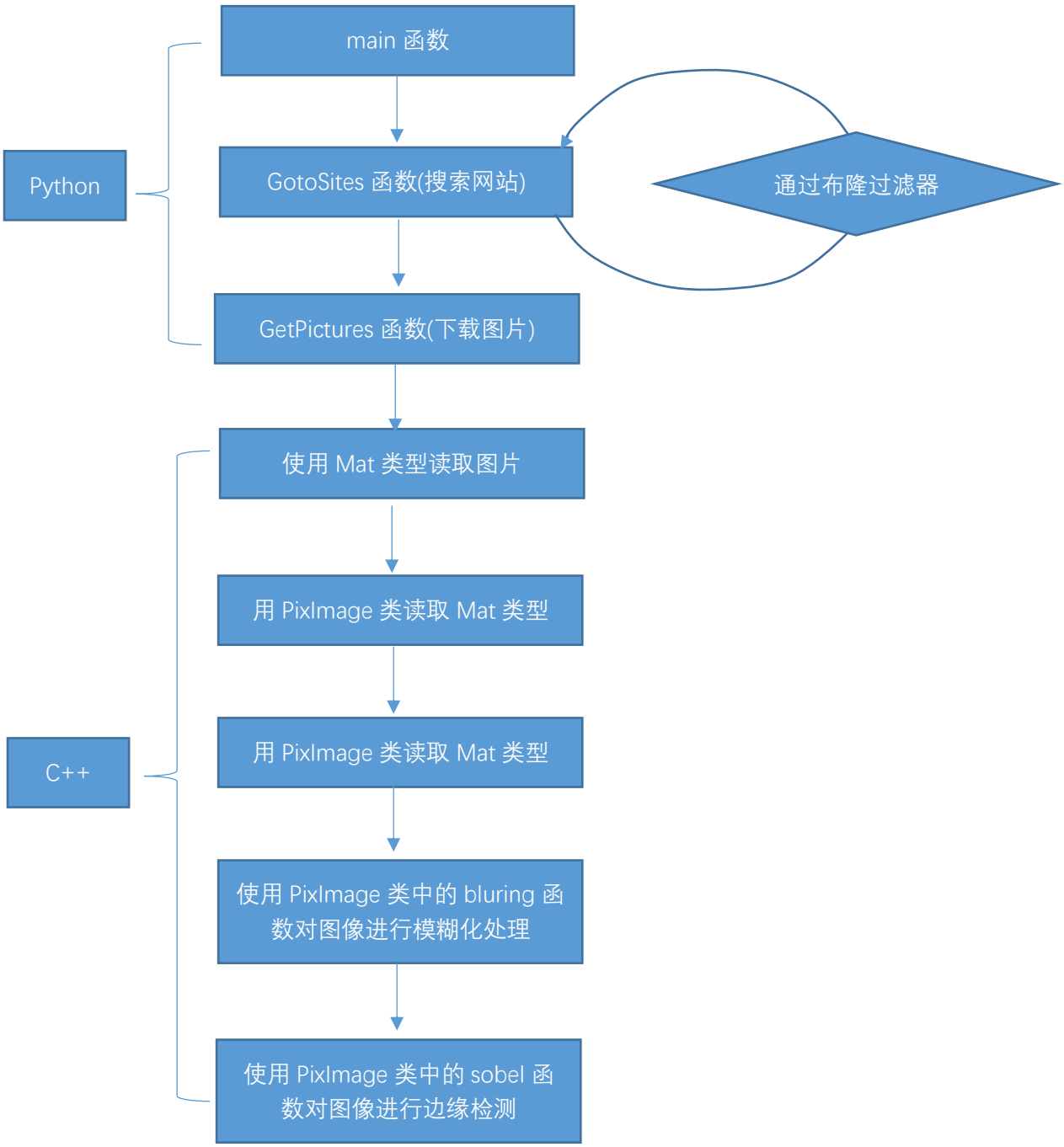
数据结构实验——图片网络爬虫设计与图像处理

161220017 陈翔

1 总体设计框架



2 实现细节

2.1 python 部分

2.1.1 概述

作者使用的 python 版本为 3.6，显式调用的 python 库有两个，分别为 re 和 requests。

在本实验中，作者一共定义了三个全局函数和一个类，三个函数名分别为 main，Goto_Sites 和 Get_Picture，定义的类名为 Bfilter。下面具体阐述它们的功能。

2.1.2main

该函数为主函数，程序从该函数开始执行。函数对爬取的初始网站进行了设置，然后以此为参数调用 Goto_Sites 函数。

```
def main():
    url = 'http://image.baidu.com/search/flip?tn=baiduimage&ipr
    j = 0
    Goto_Sites(url, j)
```

2.1.3Goto_Sites

该函数将传入的参数中的文档内容进行了读取，然后搜索其中所有可能含有网页地址的部分（通过正则表达式），并将其中的地址读取之。对于每一个新地址，我们会首先对其是否已经访问过进行判断，方法是通过布隆过滤器，这个我们下面会详细讲述。如果通过布隆过滤器，说明该地址没有被访问过，则调用 Get_Picture 函数，参数为当前地址，紧跟着继续调用（体现深度优先）Goto_Sites 函数，参数为当前地址。

```
def Goto_Sites(web_url, j):
    new_html = requests.get(web_url).text
    new_url_behind = re.findall('/search/.*height=0', new_html)
    for each in new_url_behind:
        new_url = "http://image.baidu.com"+each
        if BFilter.BF(new_url):
            Get_Picture(new_url, j)
            BFilter.ctr += 1
            if BFilter.ctr < 1000:
                Goto_Sites(new_url, j+1)
```

注：这里的查找网站所使用的正则表达式是针对百度图片的特例。

2.1.4Get_Picture

该函数将传入的参数中的文档内容进行了读取，然后搜索其中所有可能含有图片的部分，并将其中的图片地址打印之，接着对其进行下载操作。因为可能有无法下载的情况，为了避免程序进入假死状态，将下载语句放在 try 中，并规定如果响应时间超过 2 秒，则输出错误信息，继续下一张图片的下载。

```
def Get_Picture(web_url, j):
    i = j*60
    new_html = requests.get(web_url).text
    pic_url = re.findall('"objURL":"(.*?)",', new_html, re.S)
    for each in pic_url:
        print(each)
        try:
            pic = requests.get(each, timeout=2)
        except:
            print('错误，当前图片无法下载')
            continue
        string = 'pictures\\'+str(i)+'.jpg'
        fp = open(string, 'wb')
        fp.write(pic.content)
        fp.close()
        i += 1
```

注：这里的查找图片所使用的正则表达式是针对百度图片的特例。

2.1.5BFilter

这是一个类，其中的成员包括 8 个 hash 函数，8 个用来存放不同 hash 表的容量为 2500bit 的数组，以及一个判断字符串以在过滤器中的函数（这也是函数的接口）。八个 hash 函数已在讲义中给出，这里不重复。下面具体介绍一下程序接口的实现。

该函数的参数为一个字符串。在函数体中，首先声明一个值为 false 的布尔量 flag，然后依次调用八个 hash 函数，如果有一个 hash 函数返回值为 True，则说明该字符串不在原有的集合中，则将 flag 的值改为 true 然后该字符串的映射位置在对应的 hash 表中置一。最后返回值为非 flag。

```
def BF(str):
    flag = True
    if (BFilter.RSHashList[abs(BFilter.RSHash(str)) % 2500] == 0):
        flag = False
        BFilter.RSHashList[abs(BFilter.RSHash(str)) % 2500] = 1
    if (BFilter.JSHashlist[abs(BFilter.JSHash(str)) % 2500] == 0):...
    if (BFilter.PJWHashlist[abs(BFilter.PJWHash(str)) % 2500] == 0):...
    if (BFilter.ELFHashlist[abs(BFilter.ELFHash(str)) % 2500] == 0):...
    if (BFilter.BKDRHashlist[abs(BFilter.BKDRHash(str)) % 2500] == 0):...
    if (BFilter.SDBMHashlist[abs(BFilter.SDBMHash(str)) % 2500] == 0):...
    if (BFilter.DJBHashlist[ab                                        ):...
    if (BFilter.DEKHashlist[abs(BFilter.DEKHash(str)) % 2500] == 0):...
    return ~flag
```

Remove redundant parentheses more... (Ctrl+F1)

假设集合中已经存在 1000 个元素了，那么它们在任意 hash 函数的映射下的地址数一定小于等于 1000。假设它们最大都可以映射到 1000 个地址，那么接下来第 1001 个元素在每个 hash 函数下映射到一个非空地址的平均概率为 0.4，则八个 hash 函数都认为它已经被访问过的概率为 $1/(0.4)^8$=0.0655%，当然这还包括他的确已被访问过的概率，故假阳性概率远远小于 0.1%。

2.2 C++部分

2.2.1 概述

作者使用了 opencv 库用来读取图片格式和显示图片效果，其余操作都由自

定义的 PixImage 类来完成。接下来我们介绍一下这个类。

2.2.2PixImage

```
class PixImage {
private:
    Matrix R;//红色通道像素矩阵
    Matrix G;//绿色通道像素矩阵
    Matrix B;//蓝色通道像素矩阵
    Matrix Grey;//灰度值
    string name;//图像名称
    int len;//图像长度
    int height;//图像高度
    int size;//图像大小
public:
    PixImage(Mat img);//通过传入的Mat类构造自己的P
    PixImage(Mat img, string my_name); //传入带名
    void display();//显示图像信息
    void display(int x, int y);//显示(x,y)坐标处的
    void bluring(int n);//对图像进行均值过滤模糊化
    void sobel();//图像边缘检测算法
    Mat ret_value();//返回一个Mat对象
};
```

该类有八个成员变量和七个成员函数，其中 R,G,B，Grey 的类型为自定义的 Matrix 类，这是作者以前实现的，包括了一些矩阵常用的功能。七个成员函数中实际上只有 5 个不同名的函数。PixImage 函数为构造函数，通过一个传入的 Mat 类型来初始化成员变量。display 函数用来显示图像像素信息。bluring 函数用来对图像进行均值过滤。sobel 用来对图像进行边缘检测。ret_value 用来返回一个 Mat 对象。

- PixImage():用来根据传入的 Mat 对象初始化各成员变量

```
PixImage::PixImage(Mat img, string my_name) {//通过传入的Mat类构造自己的PixImage类
    len = img.cols;
    height = img.rows;
    size = len * height;
    name = my_name;
    Matrix R_temp(height, len);
    Matrix G_temp(height, len);
    Matrix B_temp(height, len);
    Matrix temp_Grey(height, len);
    Grey = temp_Grey;
    for (int i = 0; i<height; i++)
        for (int j = 0; j < len; j++) {
            int b = img.at<Vec3b>(i, j)[0];
            int g = img.at<Vec3b>(i, j)[1];
            int r = img.at<Vec3b>(i, j)[2];
            R_temp[i][j] = r;
            G_temp[i][j] = g;
            B_temp[i][j] = b;
        }
    R = R_temp;
    G = G_temp;
    B = B_temp;
}
```

- display():

```
void PixImage::display() {
    cout << "图片大小为" << size;
    cout << ",其中长为" << len;
    cout << ",高为" << height;
    cout << endl;
    for (int i = 0; i<height; i++)
        for (int j = 0; j < len; j++) {
            cout << "[" << B[i][j] << "," << G[i][j] << "," << R[i][j] << "]" << " ";
        }
    cout << endl;
}
```

- bluring(int n):均值过滤模糊化处理

```
void PixImage::bluring(int n) {//如果参数为偶数，自动转为奇数进行处理
    if (n <= 0) { ... }

    if (n == 1)//参数为1，不做任何处理
        return;
    int r = (int)(n / 2);


    //中间部分
    for(int i=r;i<height-r;i++)//对周围拥有足够多的元素的元素进行处理
        for (int j = r; j < len - r; j++) { ... }


    //左上方
    for (int i = 1; i < r; i++)//对其他元素进行处理,认为新半径为1
        for (int j = 1; j < r; j++) { ... }

    //右上方
    for (int i = 1; i < r; i++)//对其他元素进行处理,认为新半径为1
        for (int j = len -r; j < len-1; j++) { ... }

    //左下方
    for (int i = height - r; i < height - 1; i++)//对其他元素进行处理,认为新半径为1
        for (int j = 1; j < r; j++) { ... }

    //右下方
    for (int i = height - r; i < height - 1; i++)//对其他元素进行处理,认为新半径为1
        for (int j = len - r; j < len - 1; j++) { ... }
}
```

其中中间部分的代码为为将某一元素周围半径 r 的元素平均值赋予该元素：

```
//中间部分
for(int i=r;i<height-r;i++)//对周围拥有足够多的元素的元素进行处理
    for (int j = r; j < len - r; j++) {
        int sum_R = 0;//存储周围n*n矩阵的各元素之和
        int sum_G = 0;
        int sum_B = 0;
        for (int x = i - r; x <= i + r; x++)
            for (int y = j - r; y <= j + r; y++) {
                sum_R += R[x][y];
                sum_G += G[x][y];
                sum_B += B[x][y];
            }
        R[i][j] = sum_R / ((2 * r + 1)*(2 * r + 1));//赋予平均值
        G[i][j] = sum_G / ((2 * r + 1)*(2 * r + 1));
        B[i][j] = sum_B / ((2 * r + 1)*(2 * r + 1));
    }
```

- sobel():用来进行边缘检测

```
void PixImage::sobel() {
    Mat Greyimg; //声明一个灰度图
    Mat img = ret_value();//返回当前对象对应的Mat对象
    cvtColor(img, Greyimg, CV_BGR2GRAY); //将该Mat对象转为灰度图
    imshow("Greyimage", Greyimg); //展示灰度图
    for (int i = 0; i<height; i++)
        for (int j = 0; j < len; j++) {
            Grey[i][j] = Greyimg.at<uchar>(i, j);
        }
    Matrix Gx(height, len);
    Matrix Gy(height, len);
    Matrix G(height, len);
    for (int i = 1; i<height - 1; i++)//对周围拥有足够多的元素的元素进行处理
        for (int j = 1; j < len - 1; j++) {
            Gx[i][j] = (-1)*Grey[i - 1][j - 1] + 0 * Grey[i][j - 1] + 1 * Grey[i + 1][j - 1]
                + (-2)*Grey[i- 1][j] + 0 * Grey[i][j] + 2 * Grey[i + 1][j]
                + (-1)*Grey[i - 1][j + 1] + 0 * Grey[i][j + 1] + 1 * Grey[i + 1][j + 1];
            Gy[i][j] = 1*Grey[i - 1][j - 1] + 2* Grey[i][j - 1] + 1 * Grey[i + 1][j - 1]
                + 0*Grey[i - 1][j] + 0 * Grey[i][j] + 0 * Grey[i + 1][j]
                + (-1)*Grey[i - 1][j + 1] + (-2) * Grey[i][j + 1] + (-1) * Grey[i + 1][j + 1];
            G[i][j] = (int)sqrt(pow(Gx[i][j], 2) + pow(Gy[i][j], 2));
            //cout << G[i][j] << "  ";
        }
    long long sum=0;
    for (int i = 1; i<height - 1; i++)//对周围拥有不足量的元素的元素进行处理
        for (int j = 1; j < len - 1; j++) {
            sum += Grey[i][j];
        }
    int mean = (int)(sum / ((height - 2)*(len - 2)));
    int scale = 30; //可以修改的值
    int cutoff = scale * mean;
    int thresh = (int)sqrt(cutoff);//阈值
    //将灰度图转为黑白图
    for (int i = 1; i<height - 1; i++)
        for (int j = 1; j < len - 1; j++) {
            if (G[i][j] > thresh)
                Greyimg.at<uchar>(i, j) = 0xFF;
            else
                Greyimg.at<uchar>(i, j) = 0x00;
        }
    imshow("BAWimage", Greyimg);
}
```

- ret_value():用来返回一个 Mat 对象，方便显示操作

```
Mat PixImage::ret_value() {
    Mat temp(height, len, CV_8UC3);
    uchar *pxvec = temp.ptr<uchar>(0);//取像素数据首地址
    for (int i = 0; i < height; i++) {
        pxvec = temp.ptr<uchar>(i);//取每一行的像素首地址
        for (int j = 0; j < len * 3; j++) {
            uchar x = 0xFF;
            if (j % 3 == 0) {
                pxvec[j] = (unsigned char)B[i][(int)(j / 3)];
                //cout << (int)pxvec[j] << " ";
            }
            else if (j % 3 == 1) {
                pxvec[j] = (unsigned char)G[i][(int)(j / 3)];
                //cout << (int)pxvec[j] << " ";
            }
            else {
                pxvec[j] = (unsigned char)R[i][(int)(j / 3)];
                //cout << (int)pxvec[j] << " ";
            }
        }
    }
    return temp;
}
```

3 最终效果

3.1 爬到的图片：


10011


10012


10013

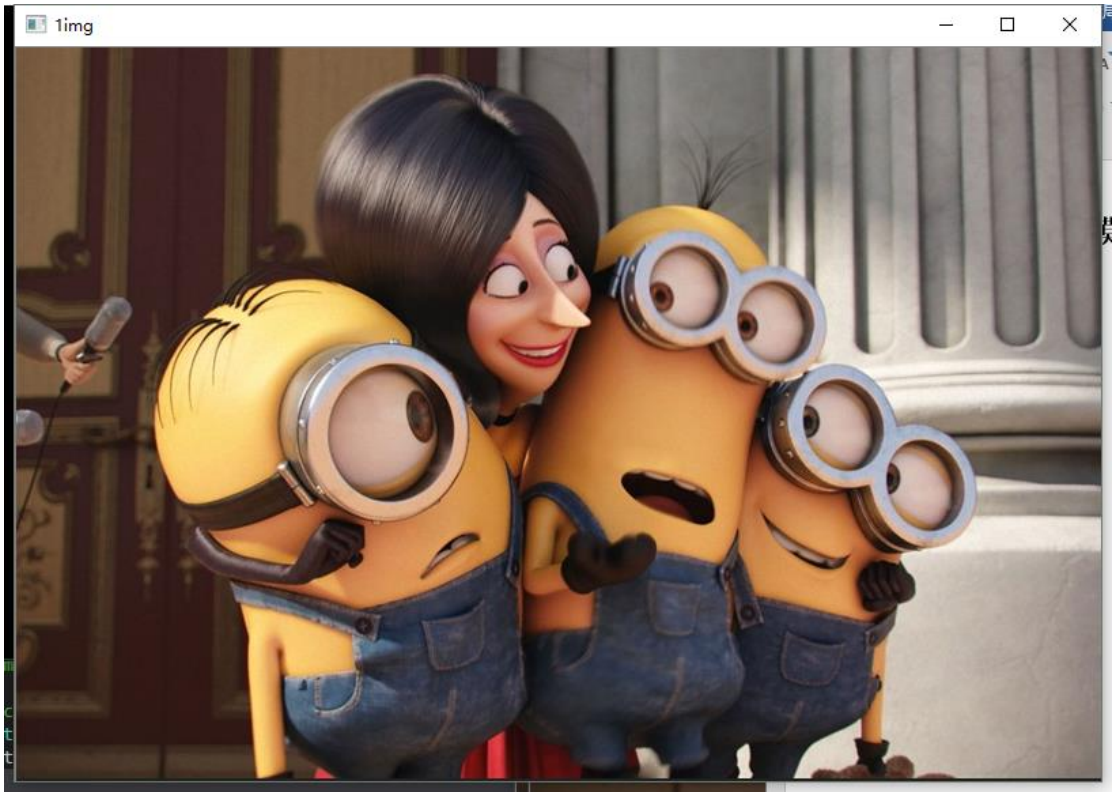

10014

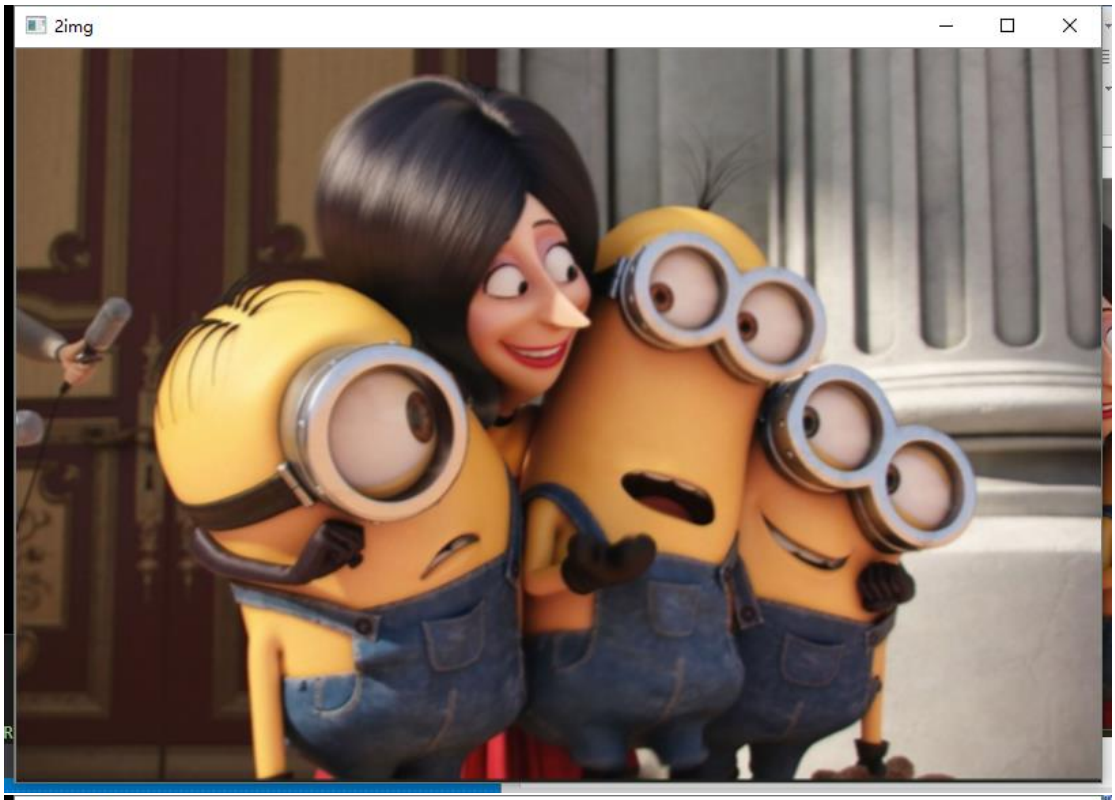

10015

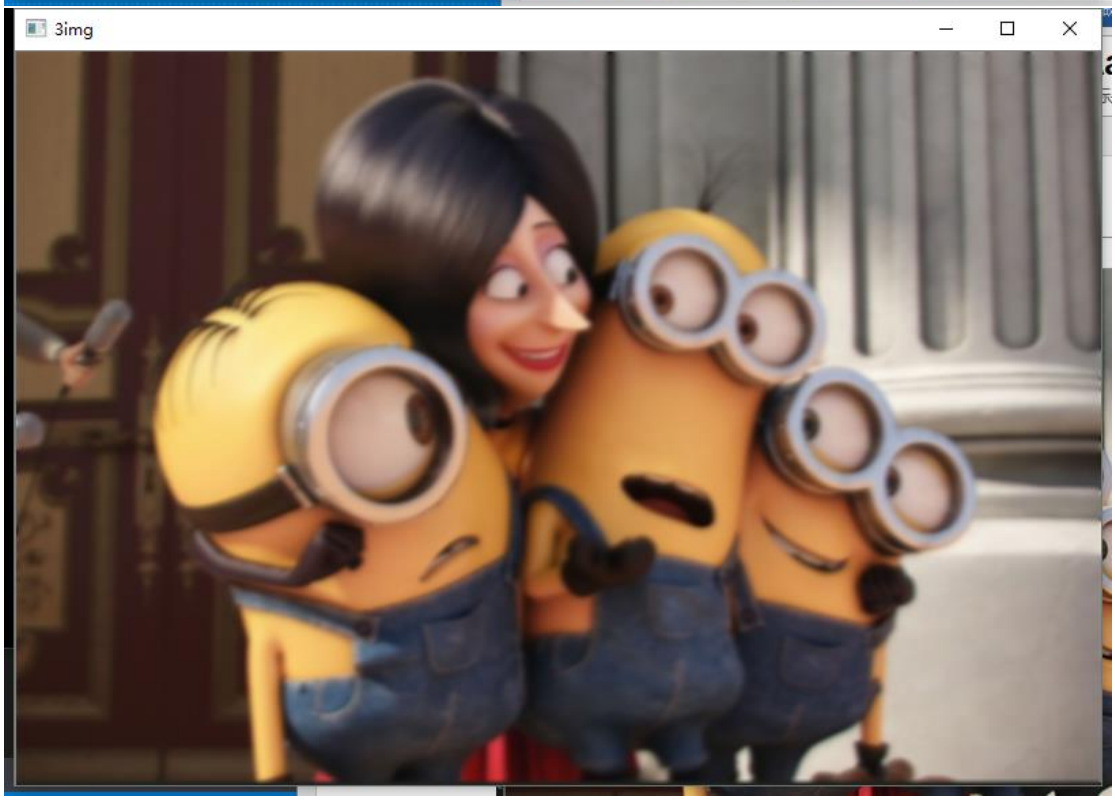

10016

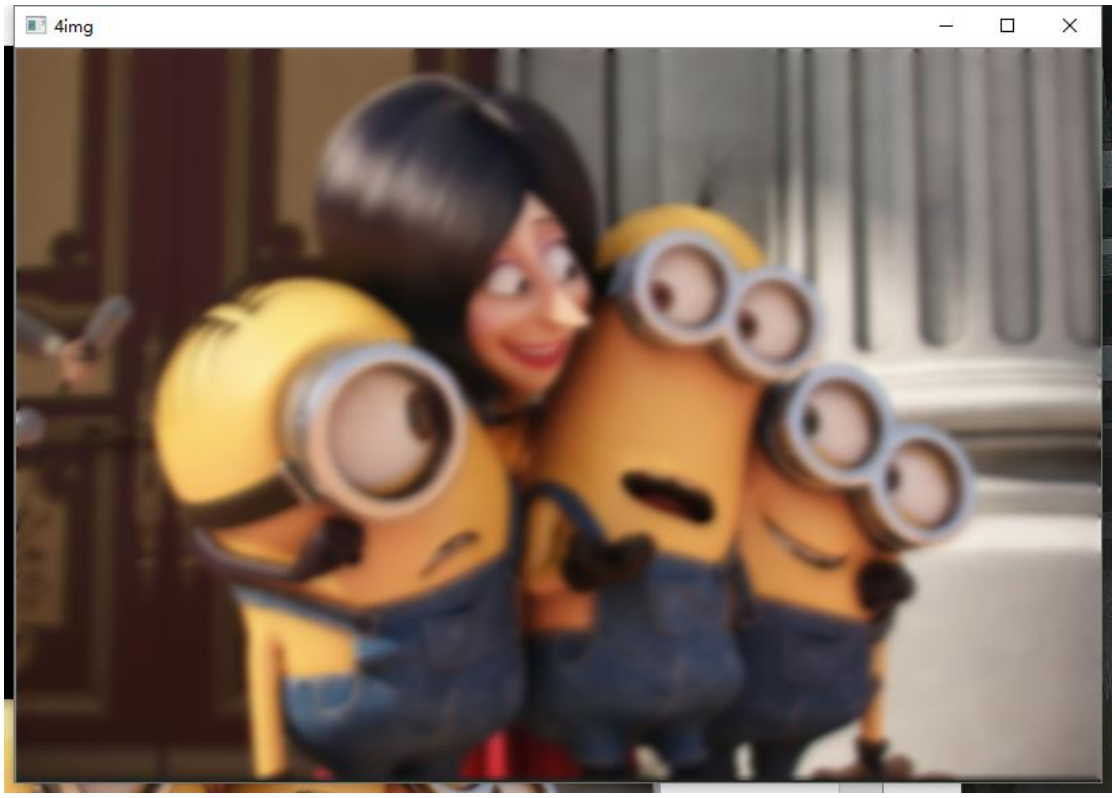

pictures
类型: 文件夹
位置: E:\Study\DataStructure
大小: 1.53 GB (1,646,815,349 字节)
占用空间: 1.54 GB (1,663,942,656 字节)

3.2 图片模糊化：


1img

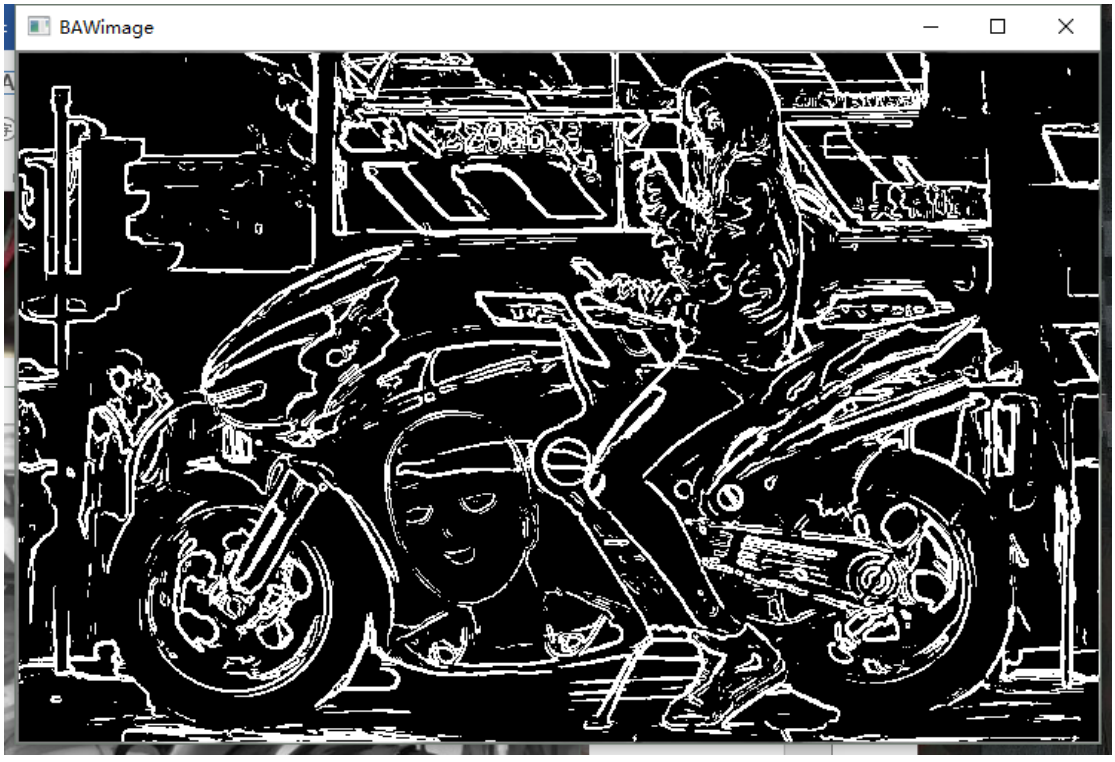

2img


3img



3.3 图片边缘检测：


Greyimage


BAWimage