图片网络爬虫设计与图片处理实验报告

姓名：王明
学号：161220124
班级：16 级李武军老师班
邮箱：1090616897@qq.com
时间：2018.1.4

爬虫部分流程

一、 网页选取

首先选取一个爬取的主网页，根据实验要求需选择一个网页多、图片多、易于下载和保存图片的网站，那么中国国家地理网站当然是不错的选择，图片也很正规。

网址：http://www.dili360.com/cng/pic/index.htm

网站首页：

二、 获取网页的文本

爬虫的库有很多，这里我使用的是 python 里使用的 requests 库。主要用到函数是 requests.get(url),url 是网址，该函数向该网址发起访问请求，并将结果保存为一个 response 对象。这个 responses 对象有很多的属性，我用到了其中的 raise_for_status()，判断访问的状态，如果是 200 则表示访问成功。还用到了 encoding 和 apparent_encoding,和网页的编码有关，还有 text,得到的是网页源代码文本的内容。

代码如下：

```
###根据网页的url，得到网页的HTML文本
def get_html_text(url):
    try:
        r = requests.get(url, timeout=30)          #使用get请求将网页提取为一个response对象
        r.raise_for_status()                       #查看是否访问成功
        r.encoding = r.apparent_encoding           #改变头的编码，使得整个源代码可读
        return r.text                              #返回网页的源代码文本
    except:
        return ""
```

如果运行成功的话，就得到了网页的源代码文本。（其格式是 html）

三、 匹配图片链接

接下来的一个问题自然是如何在文本中找到图片的网址链接。如果在文本匹配的问题，很容易就想到了正则表达式，而恰好 python 自带 re 库。需要使

用的是 re 库中的 re.findall(express,text),功能是在 text 文本中，匹配所有符合正则表达式 express 模式的表达式，并且作为一个列表返回。

函数观察网站的源代码：

```
<div class="c"></div>
<div class="wpr">
    <div class="detail" style="position:relative;">
        <img src="http://s0.dili360.com/www/images/loglo-footer.jpg">
      <!-- <img class="second" src="/images/logo-dili360.png">-->
      <div style="
          width: 95px;
          text-align: center;
          color: #fff;
          position: absolute;
```

发现其中的图片链接都是.jpg 结尾，于是就有了图片的正则表达式：

'http://.*?\.jpg'

代码如下：

```
###正则表达式匹配HTML文本中的图片
def find_picture(text):
    return re.findall(r'http://.*?\.jpg',text)  #在文本中匹配所有图片url，返回一个列表
```

如果运行成功，就得到了本网站所有图片的链接。

四、 将图片保存到本地

图片链接继续用 requests 爬取，返回一个 response 对象，这里需要使用另一个 response 的属性 content，即链接的内容，对于图片链接来说就是这张图片具体的二进制数据。

保存到本地图片需要用到 python 自带的系统库即 os 库，我们需要设定图片保存路径，然后对于文件夹是否存在、图片是否存在进行判断，然后使用文件操作，将图片的 content 写到文件里，那么图片就被保存下来了，当然为了计数的方便，可以用数字给爬取的图片命名。

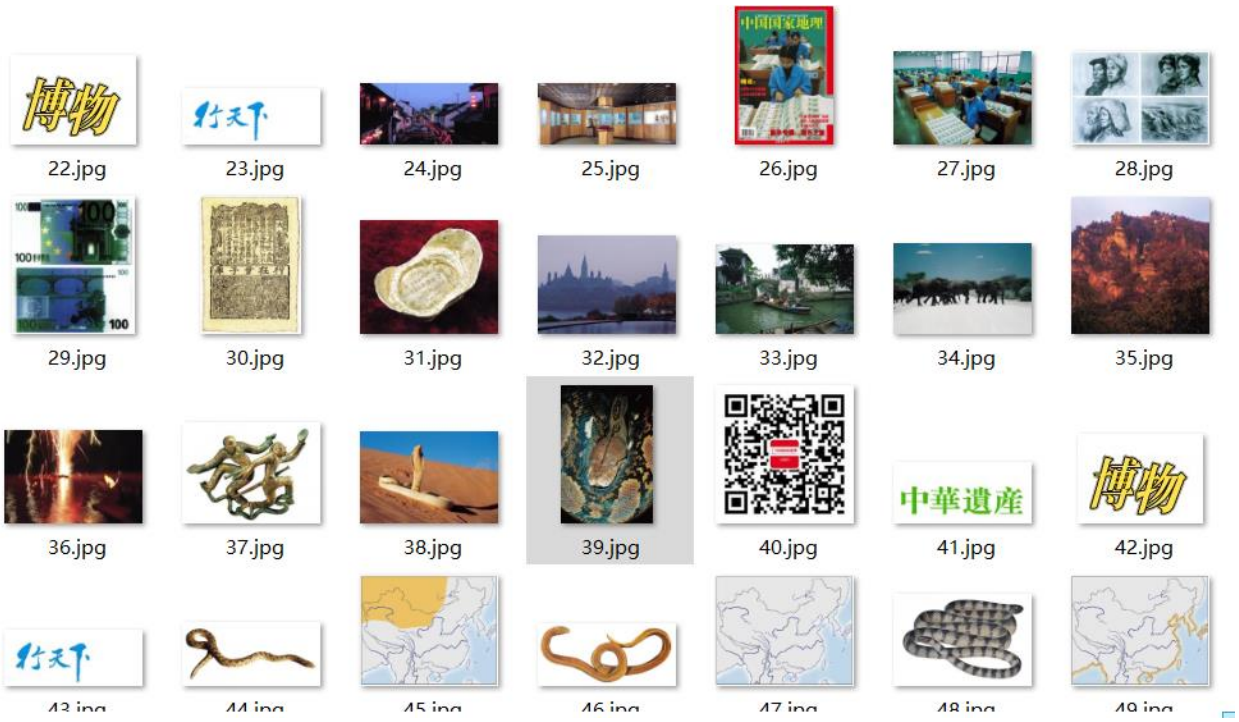
代码如下：

```
###将图片保存到本地文件夹
def save_picture(url):
    global num_now
    global num_need
    root = "C://Users//HP//Desktop//picture//"          #指定图片保存的目录路径
    path = root + str(num_now) + ".jpg"                 #对图片重命名，方便查看
    try:
        if num_now <= num_need:                         #检查是否已爬到足够的图片
            num_now = num_now + 1
            if not os.path.exists(root):                #检查文件夹是否存在
                os.mkdir(root)
            if not os.path.exists(path):                #检查图片是否已存在
                r = requests.get(url)
                with open(path, 'wb') as file:          #打开文件夹并写入图片
                    file.write(r.content)
                    file.close()
                    print("文件保存成功")
            else:
                print("文件已存在")
        else:
            return
    except:
        print("爬取失败")
```

合并上边的工作，完整地爬取图片：

```
###根据网页的url,爬下来一个网页的需要的图片
def crawl_picture(url):
    global counter
    global per_need
    picture_list = []                                   #新建一个列表来保存图片的url
    html_text = get_html_text(url)
    picture_list = find_picture(html_text)
    for i in picture_list:                              #循环的方式来保存图片，控制保存的数量
        counter = counter + 1
        if counter > per_need:
            counter = 0
            return
        save_picture(i)
```

爬取图片的效果：



五、 获取网页的子网页

和爬取图片的链接一样，只需使用正则表达式库来匹配所有的子网页的链接。

观察网页源代码：

```
lass= part-two >
iv class="wpr">
  <div class="menu">
      <ul>
          <a href="/cng/index/index.htm"><li class="active">首页</li></a>
          <a href="/cng/tag/index.htm"> <li>杂志内容</li></a>
          <a href="/cng/pic/index.htm"> <li>杂志图片</li></a>
          <!-- <a href="/cng/map/index.htm"> <li>地理地图</li></a> -->
          <a href="/cng/mag/index.htm"><li>往期回顾</li></a>
      </ul>
  </div>
div>
```

可以发现，都是/cng/开头，htm 结尾，所有正则表达式写为：

```
###正则表达式匹配HTML文本中的子网页
def find_childweb(text):
    return re.findall(r'/cng/.*?\.htm',text)    #在文本中匹配所有子网页url，返回一个列表
```

这样就得到了所有子网页的链接。

结合前面爬取网页源代码，可得到爬取子网页链接的代码：

```
###根据网页的url,爬取所有的子网页的url
def crawl_childweb(url):
    child_list = []                             #新建一个列表来保存子网页的url
    html_text = get_html_text(url)
    child_list = find_childweb(html_text)
    return child_list                           #返回这个爬取到的所有子网页的url
```

这里的 child_list 是一个的列表类型。

六、 爬取整个网站的图片

上面已经实现了将一个网页的子网页保存为一个列表 child_list 以及将一个网页的所有图片爬取到本地。那么如何爬取整个网站的图片内，基本的思路肯定是对于当前网页，先把所有图片爬下来，然后按照得到的子网页的顺序，去访问子网页，然后重复上述步骤。显然，用递归做比较方便，然后题目的要求是使用深度优先，那么需要借助一个结构——栈。每次 pop 出来一个网页，爬取图片，然后将子网页的链接 push 进栈，然后重复操作，直到图片数量爬够或者栈为空，爬取就结束。

需要设置一个栈结构，即一个全局的列表类型，定义为 html_list。

递归调用的代码如下：

```
###爬取整个网站所需的图片，通过栈方法，本质上是深度优先算法
def crawl_all():
    global html_list
    global num_now
    global num_need
    if num_now <= num_need:
        if html_list != []:                          #判断栈是否空，即是否爬玩所有网页
            now = html_list.pop()                    #取出栈顶元素。
            print(now)
            crawl_picture(now)                       #爬取当前网页的图片
            child_list = crawl_childweb(now)         #爬取当前网页的子网页，保存为列表
            for i in child_list:                     #通过BF的Hash来过滤已经爬过的网页，将没有爬过的进栈
                s = 'http://www.dili360.com' + i
                if Hash(s) == 0:
                    print(Hash(s))
                    html_list.append(s)
            crawl_all()                              #递归调用继续爬取
        else:
            print("爬取结束")
    else:
        return
```

这样就利用栈结构实现了深度优先爬取整个网站图片的功能。

七、 网页去重：

题目规定使用布隆过滤器，即 bloom filter。布隆过滤器的原理就不在此谈了，下面就是解决如何在指定爬取 1000 个网站，即 n = 1000 的情况下，计算最优的 m(数组长度)和 k(hash 函数的个数)。

确定的方法为：

系统首先要计算需要的内存大小m bits:

$$P = 2^{-ln2\cdot\frac{m}{n}} \Rightarrow \ln p = \ln 2\cdot(-\ln 2)\cdot\frac{m}{n} \Rightarrow m = -\frac{n\cdot lnp}{(ln2)^2}$$

再由m，n得到hash function的个数:

$$k = ln2\cdot\frac{m}{n} = 0.7\cdot\frac{m}{n}$$

在 n = 1000 时，m = 14378，k = 10。

实际操作时，由于希望取整数，以及提供的 hash 函数没那么多，于是查表：

Table 3: False positive rate under various *m*/*n* and *k* combinations.

m/*n*	*k*	*k*=1	*k*=2	*k*=3	*k*=4	*k*=5	*k*=6	*k*=7	*k*=8
2	1.39	0.393	0.400						
3	2.08	0.283	0.237	0.253					
4	2.77	0.221	0.155	0.147	0.160				
5	3.46	0.181	0.109	0.092	0.092	0.101			
6	4.16	0.154	0.0804	0.0609	0.0561	0.0578	0.0638		
7	4.85	0.133	0.0618	0.0423	0.0359	0.0347	0.0364		
8	5.55	0.118	0.0489	0.0306	0.024	0.0217	0.0216	0.0229	
9	6.24	0.105	0.0397	0.0228	0.0166	0.0141	0.0133	0.0135	0.0145
10	6.93	0.0952	0.0329	0.0174	0.0118	0.00943	0.00844	0.00819	0.00846
11	7.62	0.0869	0.0276	0.0136	0.00864	0.0065	0.00552	0.00513	0.00509
12	8.32	0.08	0.0236	0.0108	0.00646	0.00459	0.00371	0.00329	0.00314
13	9.01	0.074	0.0203	0.00875	0.00492	0.00332	0.00255	0.00217	0.00199
14	9.7	0.0689	0.0177	0.00718	0.00381	0.00244	0.00179	0.00146	0.00129
15	10.4	0.0645	0.0156	0.00596	0.003	0.00183	0.00128	0.001	0.000852
16	11.1	0.0606	0.0138	0.005	0.00239	0.00139	0.000935	0.000702	0.000574

最终取的 m = 16 * n = 16000，取的 k = 6。hash 函数定义的代码如下：

```
###6个Hash函数定义
def RSHash(str):
    global m
    b = 378551
    a = 63689
    hashing = 0
    for i in range(len(str)):
        hashing = hashing * a + ord(str[i])
        a = a * b
    return hashing%m

def JSHash(str):
    global m
    hashing = 1315423911
    for i in range(len(str)):
        hashing ^= (hashing << 5) + ord(str[i]) + (hashing >> 2)
    return hashing%m

def ELFHash(str):
    global m
    hashing = 0
    x = 0
    for i in range(len(str)):
        hashing = (hashing << 4) + ord(str[i])
        x = hashing & 0xf0000000
        if x != 0:
            hashing ^= (x >> 24)
        hashing &= ~x
    return hashing%m

def BKDRHash(str):
    global m
    seed = 131
    hashing = 0
    for i in range(len(str)):
        hashing = (hashing * seed) + ord(str[i])
    return hashing%m

def SDBMHash(str):
    global m
    hashing = 0
    for i in range(len(str)):
        hashing = ord(str[i]) + (hashing << 6) + (hashing << 16) - hashing
    return hashing%m

def DJBHash(str):
    global m
    hashing = 5381
    for i in range(len(str)):
        hashing = ((hashing << 5) + hashing) + ord(str[i])
    return hashing%m
```

每个 hash 函数参数是网址，返回值是在布隆过滤器中的数组下标。

检查是否是已有网址的代码如下：

```
###对一个url进行Hash，修改BF数组并返回结果
def Hash(str):
    global bf
    result = 0
    if bf[RSHash(str)] == 1:          #如果映射全是1，则说明已经出现过
        if bf[JSHash(str)] == 1:
            if bf[ELFHash(str)] == 1:
                if bf[BKDRHash(str)] == 1:
                    if bf[SDBMHash(str)] == 1:
                        if bf[DJBHash(str)] == 1:
                            result = 1
    if result == 0:                   #修改BF数组
        bf[RSHash(str)] = 1
        bf[JSHash(str)] = 1
        bf[ELFHash(str)] = 1
        bf[BKDRHash(str)] = 1
        bf[SDBMHash(str)] = 1
        bf[DJBHash(str)] = 1
    return result                     #返回判断结果

###初始化BF数组
def init_bf():
    global bf
    global m
    for i in range(m):                #通过循环置0
        bf.append(0)
```

初始化 BF 数组，然后对于每个网址就行检查，如果 hash 下来全是 1，那么就说明该网址在之前已经出现了。

八、 main 函数完成爬虫

main 函数设定起始网页的网址，并且调用以上的函数，完整实现爬虫功能

代码如下：

```
###main函数设置爬取的主网站链接
def main():
    global html_list
    init_bf()                         #初始化BF
    url = 'http://www.dili360.com/cng/pic/index.htm' #指定主网页
    html_list.append(url)             #将主网页进栈
    crawl_all()                       #开始爬取

main()
```

爬取的图片效果已经在上面的部分中截图展示过了。

该爬虫的使用方法我会放在程序使用说明里。

经过爬虫实践，该网站一个有 1007 个网页，超过 10000 张图片，符合实验的要求，网页去重效果也很好。

图片处理部分流程

一、 图片解析

要对图片进行处理，首先需要将图片保存在数据结构里并且能表达每个像素点的 RGB 值，我使用的是使用度较广的开源库 OPENCV，保存图片用的是 Mat 类，其可以解析图片，并且提取出每个像素点的 RGB 值。

使用 OPENCV 库，需要将其加入到 VS 中，代码如下：

```
#include <iostream>
#include "opencv2/opencv.hpp"          //使用第三方库opencv
#include <cmath>

using namespace std;
using namespace cv;

```

将图片保存为一个 Mat 类用的是 imread(),代码如下：

```
{
    Mat picture1 = imread("star.jpg");  //读入图片，保存为Mat类的一个对象
    Mat picture2 = imread("star.jpg");
```

将一个 Mat 类的对象保存为本地图片用的是 imwrite(),代码如下：

```
    imwrite("star3.jpg", ret);                    //保存最终二值化后的图片
}
```

提取每个像素点使用的是 Mat 的 at()方法，代码如下：

```
        }
    im.at<Vec3b>(i, j)[0] = blue / num;                /
    im.at<Vec3b>(i, j)[1] = green / num;
    im.at<Vec3b>(i, j)[2] = red / num;
}
```

这是一个三通道的图片的像素点提取方法。

二、 模糊化

模糊化就是以每个点为中心，取一个方阵，计算里面的 RGB 的平均值，然后赋给该像素点。

代码如下：

```cpp
void Bluring(Mat im, int n)   //im为要处理的图片，n为方阵的边长
{
    int t = (n - 1) / 2;
    for (int i = 0; i < im.rows; i++)
        for (int j = 0; j < im.cols; j++)
        {
            int blue = 0, green = 0, red = 0;
            int row_begin = i - t, row_end = i + t, col_begin = j - t, col_end = j + t;
            if (row_begin < 0)              //边缘处理
                row_begin = 0;
            if (row_end > im.rows - 1)
                row_end = im.rows - 1;
            if (col_begin < 0)
                col_begin = 0;
            if (col_end > im.cols - 1)
                col_end = im.cols - 1;
            int num = (row_end - row_begin + 1) * (col_end - col_begin + 1);
            for (int a = row_begin; a <= row_end; a++)
                for (int b = col_begin; b <= col_end; b++)
                {
                    blue += im.at<Vec3b>(a, b)[0];          //计算方阵内的所有颜色值得和
                    green += im.at<Vec3b>(a, b)[1];
                    red += im.at<Vec3b>(a, b)[2];
                }
            im.at<Vec3b>(i, j)[0] = blue / num;             //将计算结果取平均
            im.at<Vec3b>(i, j)[1] = green / num;
            im.at<Vec3b>(i, j)[2] = red / num;
        }
    imwrite("star1.jpg", im);                               //保存为图片
}
```

效果如下，分别是原图和模糊化后的图片：

三、 边缘检测

需要完成 3 步工作，首先是将三通道的 RGB 彩色图片变成单通道的灰度图。这步操作可以由 OpenCV 库提供的函数实现，代码如下：

```
Mat im;
cvtColor(image, im, COLOR_BGR2GRAY, 1);      //将彩色的三通道图灰度化保存为单通道的灰度图
imwrite("star2.jpg", im);                    //将灰度图保存下来方便查看
```

然后是通过 sobel 算子来计算阈值，代码如下：

```
Mat IM = im.clone();                         //克隆一个im,方便保存sobel计算后的结果
for (int i = 0; i < im.rows; i++)
    for (int j = 0; j < im.cols; j++)
    {
        if (i == 0 || i == im.rows - 1 || j == 0 || j == im.cols - 1)   //对于边缘无需处理，直接赋值
            IM.at<uchar>(i, j) = im.at<uchar>(i, j);
        else
        {
            int Gx = 0, Gy = 0;                  //sobel算子来计算某个像素的灰度值
            double G = 0;
            Gx = (im.at<uchar>(i - 1, j + 1) + 2 * im.at<uchar>(i, j + 1) + im.at<uchar>(i + 1, j + 1
            Gx = (im.at<uchar>(i - 1, j - 1) + 2 * im.at<uchar>(i - 1, j) + im.at<uchar>(i - 1, j + 1
            G = sqrt(Gx * Gx + Gy * Gy);
            IM.at<uchar>(i, j) = G;
        }
    }

double sum = 0;
double scale = 12;
double average;
for (int i = 0; i < im.rows; i++)
    for (int j = 0; j < im.cols; j++)
    {
        sum += im.at<uchar>(i, j);           //灰度值的和
    }
average = sum / (im.rows * im.cols);         //取平均值
cout << average << endl;
double cutoff = scale * average;
double thresh = sqrt(cutoff);
```

计算完阈值，还得添加一个系数 scale，这个数值的大小可以根据图片处理的实际效果来确定，对于这张图片，我取的是 12，此时相对来说效果要好。

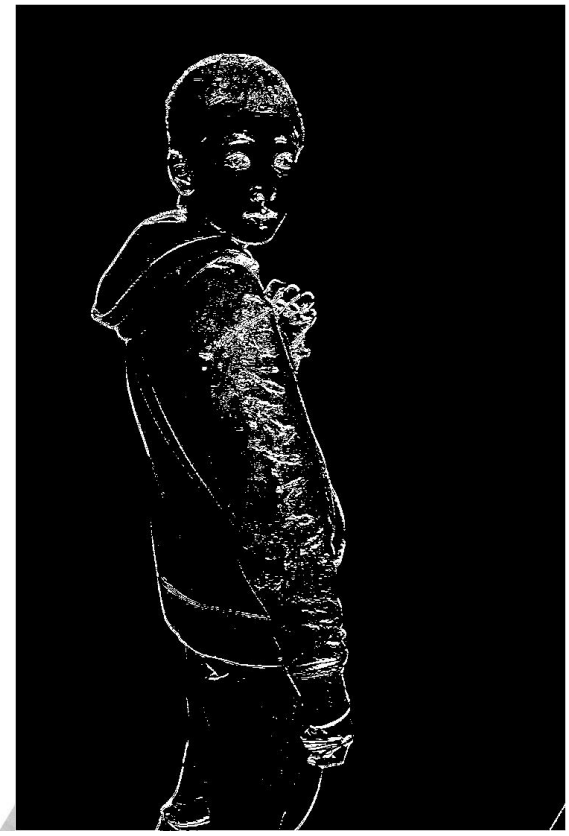
最后一步根据计算好的阈值来将图片二值化，这一步也是通过库提供的函数来实现的，这样就可以清晰地解析出图片的边缘，代码如下：

```
Mat ret;
threshold(IM, ret, thresh, 255, THRESH_BINARY);   //图像的二值化

imwrite("star3.jpg", ret);                        //保存最终二值化后的图片
```

这样，图片的边缘提取工作就完成了。

效果如下，左边是灰度化后的图，右边是二值化后的图：



从效果上看，还是可以的，边缘提取的还算准确与清晰。
所以，图片处理的功能也完整实现了。
程序使用方法我会放在程序使用说明里。