数据结构大实验
图片网络爬虫设计和图像处理

姓名：孙治
学号：161220114
email: imagecser@gmail.com
时间：2018 年 1 月 2 日星期二
项目：https://github.com/imagecser/dsimage

目录

一、实验目的

1、 实现 python Bloom Filter

2、 实现爬虫，使用正则表达式等工具，按照深度遍历优先爬取足量图片。

3、 实现图像处理，可以进行图像平滑降噪和 Sobel 边缘检测。

二、实验环境

实验环境

g++ (Ubuntu 5.4.0-6ubuntu1~16.04.5) 5.4.0 20160609

python 3.5.2

依赖项

python.requests

python.bs4

python.bitstring

stb: single-file public domain (or MIT licensed) libraries for C/C++

swipebox

三、Bloom Filter

1、 概述

本 Bloom Filter 使用 python 实现类，设置了 add 和__contains__内置函数，参数为 n 即存储的数量和 error 即错误率，其中，hash 函数使用简单的几个哈希函数，若不足，则通过修改参数增加足量的哈希函数以供使用。

2、 hash 函数

我们在 scrawler/hashf.py 中定义了数个 hash 函数，均来自网上的一些简单且高效的方案。

其中，我们都传入了 key 和 capacity 参数，key 即需要处理的数据，而 capacity 即 hash 函数返回值的最大值，我们通过这一参数使得返回值在范围内且避免过大数字的计算。

最后，我们有一个函数 hashf(key, capacity, n)，传入 n 参数，调整 hash 函数的种子，得到不同的结果。

3、 初始化

我们首先初始化

self.bits 存储 01 序列

self.capacity 表示 bits 的位数

self.k 表示需要的函数数量

self.n 表示需要存入的数据数量

self.error 表示错误率用于计算 capacity

self.count 表示已存入的数据数量

通过对 Bloom Filter 的学习，我们知道以下公式（出自维基百科）：

Optimal number of hash functions [edit]

The number of hash functions, k, must be a positive integer. Putting this constraint aside, for a given m and n, the value of k that minimizes the false positive probability is

$$k = \frac{m}{n} \ln 2.$$

The required number of bits, m, given n (the number of inserted elements) and a desired false positive probability p (and assuming the optimal value of k is used) can be computed by substituting the optimal value of k in the probability expression above:

$$p = \left(1 - e^{-\left(\frac{m}{n}\ln 2\right)\frac{n}{m}}\right)^{\frac{m}{n}\ln 2}$$

which can be simplified to:

$$\ln p = -\frac{m}{n}(\ln 2)^2.$$

This results in:

$$m = -\frac{n \ln p}{(\ln 2)^2}$$

So the optimal number of bits per element is

$$\frac{m}{n} = -\frac{\log_2 p}{\ln 2} \approx -1.44 \log_2 p$$

with the corresponding number of hash functions k (ignoring integrality):

$$k = -\frac{\ln p}{\ln 2} = -\log_2 p.$$

其中，使用 m 代替 self.capacity，n 代替 self.n，k 即 self.k，p 即 self.error。
通过以上公式，我们可以由参数 self.n 和 self.error 得到其他参数。
即：

```
self.capacity = int(- self.n * math.log(self.error) / (math.log(2) ** 2)) + 1
self.k = int(- math.log(self.error, 2)) + 1
```

其中 int(x) + 1 即向上取整。
至于 self.hashes，我们直接 import hashf 并从其中的函数直接 append 到 self.hashes 这一 list 数据结构存储函数。
self.bits 则调用 bitstring.Array 得到 self.capacity 大小的 01 序列并初始化为 0。

当然我们需要对初始化进行限制：
如 n>0 且 0<error<1。

4、add 函数

```
"""
>>> bf = BloomFilter()
>>> bf.add(1)
True
>>> bf.add(1)
False
:param key: add key to bloom filter
:return: if key has been in bloom filter, return False, else True
"""
```

add 函数实现了向 self.bits 中添加了一个数据。
首先判断 self.count 不得超过 self.n。
其次，我们通过 self.hashes 中存储的 k 个 hash 函数，得出需要置 1 的 self.bits 位。
同时，我们需要判断是否已经添加这一元素，如果所有 hash 函数得到的 bits 位均已置 1，则认为已添加这一函数，return False，否则 return True

5、__contains__内置函数

```
"""
>>> import pybloom
>>> bf = pybloom.BloomFilter()
>>> bf.add(1)
True
>>> 1 in bf
True
:param key: check if key is in Bloom Filter
:return: if key in Bloom Filter, return True, else False
"""
```

这个函数的过程非常类似 add 函数，甚至已经被包括在 add 函数中。

我们只需设置一个 bool 变量，通过 k 个 hash 函数，查看是否所有位均已为 1，若为 1 则 return True 否则 return False。

6、 打包

我把这一 Bloom Filter 按照 python 规范打包为了 module: pybloom 方便调用。

在 init.py 中，声明

from .pybloom import BloomFilter

只允许调用 BloomFilter 这个类。

四、网页图片爬虫

1、 概述

对于爬虫，我们的数据结构有如下：

```
count = sum([len(x) for _, _, x in os.walk(os.path.dirname("../dsimage/file/"))])  # index of file
MAX_SIZE = 15000  # stop when the count reaches MAX_SIZE
INIT_URL = "http://www.mm4000.com/"
lock_count = threading.Lock()
pageset = pybloom.BloomFilter(100000)
imgset = pybloom.BloomFilter(100000)
cache = queue.Queue(0)
stack = queue.LifoQueue(0)
stack.put(INIT_URL)
```

其中

MAX_SIZE 规定了我们需要爬取的图片数量：15000

count 即现在已爬取的图片数量，在这里，我们通过 os.path,dirname 函数，遍历并统计了已有的图片数。

INIT_URL 即我们爬取的起始网页

lock_count，因为我们需要使用多线程，对于上边的 count 我们设置了进程锁

pageset 即使用 Bloom Filter 存储已爬取的网址，进行去重

imgset 即使用 Bloom Filter 存储已下载的图片，进行去重

cache 为一个内置有锁的队列，存储我们需要下载的图片

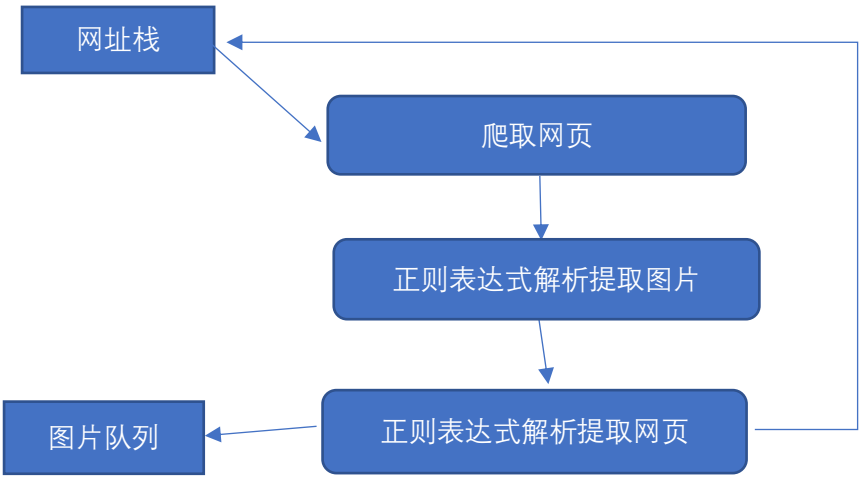
stack 为一个内置有锁的栈，存储我们需要解析的网页，其中初始有 INIT_URL。

除此之外，为了提高爬虫的效率，把整个框架分为三个部分：总体多线程、解析网页、下载图片，下面进行分别介绍。

2、 解析网页

这一步骤对应 mm4.py 文件的 parse_page()函数。

解析网页是该爬虫的最重要部分，我们需要使用到的全局变量是 pageset 即去重网址，imgset 即去重图片和 lock_count 这个是多线程需求而使用到的锁，cache 和 stack。
这个模块的框架比较简单，逻辑为：



① 在其中，为了提高效率，使用了 bs4 模块，写出如下的正则表达式：
soup.find_all('img', attrs={'data-original': re.compile("http://.+\.(jpg|png|jpeg)$")})
提取图片
soup.find_all('a', href=re.compile("http://www.mm4000.com/.+"))
这样的正则表达式提取网址。

② 在访问网页中，为了实现伪装，仅采用了伪装浏览器的方式，但已足够：

```
urlheader = {
    'User-Agent': 'Mozilla/5.0 (compatible; Googlebot/2.1; +http://www.google.com/bot.html)',
    'Proxy-Connection': 'keep-alive',
    'Pragma': 'no-cache',
    'Cache-Control': 'no-cache',
    'Upgrade-Insecure-Requests': '1',
    'DNT': '1',
    'Accept-Encoding': 'gzip, deflate',
    'Accept-Language': 'zh-SG,zh;q=0.9,zh-CN;q=0.8,en;q=0.7,zh-TW;q=0.6'
}
```

③ 由于调试过程中的失败，发现在 headers 中，需要指明 host，进行了处理：

```
base_url = urllib.parse.urlparse(url).netloc
urlheader['Host'] = base_url
```

④ 由于之后图片还需要展示在网页中，若图片过小不合适，故进行了正则表达式的筛选：

```
for line in soup.find_all('img', attrs={'data-original': re.compile("http://.+\.(jpg|png|jpeg)$")}):
    # width = int(line.get('width'))
    # height = int(line.get('height'))
    src = str(line.get('data-original'))
    # if width > 150 and height > 150:
    tiny_pattern = re.compile(r'http://.+_(60|160)\.jpg')
    if tiny_pattern.match(src):
        continue
```

⑤ 由于网址中有大量图片高度相似的网址，浪费性能，所以进行了筛选：如把.../1234_1.html 和.../1234.html 和.../1234_2.html 均替换为.../1234.html

```
num_pattern = re.compile(r'(http://.+)_\d+\.html')
href = num_pattern.sub(r'\1.html', href)
```

⑥ 由于网址中有使用#号进行定位的访问，故将此类网址进行合并，如：
将.../index.html 和.../index.html#head 合并

```
loc_pattern = re.compile(r'(http://.+)#.+')
href = loc_pattern.sub(r'\1', href)
```

⑦ 由于一些网址为无效网址，故设置了无效时间 timeout=3，并进行了 execption 的处理

```
try:
    context = requests.get(url, headers=urlheader, timeout=3).content.decode('utf-8', 'ignore')
except Exception as e:
    print(url + " " + str(e))
    continue
```

⑧ 运用了 Bloom Filter 进行了网址和图片的筛选：

```
if src not in imgset:
    cache.put([url, src])
    imgset.add(src)
```

```
if href not in pageset:
    pageset.add(href)
    stack.put(href)
```

3、 爬取图片

这一步骤对应 mm4.py 中的 save_img()函数。

在上一函数中，我们已经把 cache 队列和 imgset 这一过滤器中填充好了图片网址，下面我们基于它进行处理。

其实也就是，将 cache 队列中的元素弹出进行爬取。

① 针对一些共性问题和 parse_page()函数的处理策略相同，如在 headers 中指定 referer 和 Host:

```
imgheader = urlheader
imgheader['Referer'] = url
imgheader['Host'] = urllib.parse.urlparse(src).netloc
```

② 如设置 timeout 和 exception 捕捉错误：

```
try:
    img = requests.get(src, headers=imgheader, timeout=3).content
except Exception as e:
    print(src + " " + str(e))
    continue
```

③ 当然为了避免爬取的图片中会有一些无效链接，404 跳转至一些网页，进行了对网页的筛选，我们知道，这样的网页他必然会有链接指向主页，故：

```
if str(img).count('mm4000.com'):
    continue
```

④ 因为爬虫有数量限制，当数量达到上限，由于多线程的使用，普通的 return 并没有效果，所以进行操作系统级别的停止程序：

```
if count > MAX_SIZE:
    print(time.time() - start)
    os._exit(0)
```

并，记录了所用时间

4、 多线程

由于我的程序在一开始就面向多线程进行了良好的设计，转向多线程只需要进行 threading 的使用.

```
thread_save = []
thread_parse = []
for i in range(20):
    thread_parse.append(threading.Thread(target=parse_page, name="parse"))
for i in range(10):
    thread_save.append(threading.Thread(target=save_img, name="save" + str(i)))
for t in thread_parse:
    t.start()
for t in thread_save:
    t.start()
for t in thread_parse:
    t.join()
for t in thread_save:
    t.join()
```

但是，由于多线程设计到了公用变量的问题，使用了 threading.Lock()这一类进行了限制：如对 count 的计数，而我们使用的栈和队列都已经内置支持多线程，所以不需要进行这样的设计。

以上即爬虫的实现细则。

五、图片处理

1、 库的选择

图片处理对于这个工程来说，是相对简单的一个部分。对于这个部分，足够引起我们考虑的一个问题是图片处理库的选择，大多数同学选择了 opencv，但经过我的查找，发现 opencv 对于这个工程来说过于厚重，根本不适合，所以，经过我在 github 上搜寻 image processing 发现了 nothings/stb 这个库，轻量化且非常合适，数据结构也符合直接的需求，所以选择了 nothings/stb 这个项目进行处理。

2、 对该程序的认识基础

在我看来，图片处理到这个步骤仍然过于简单，所以我打算将其展示在我的个人网页上，一方面是原图，另一方面是处理后的图片，为了便于展示，我将展示原图和处理后的图片和原图拼接在一起的图片，从此出发点，整个 PixImage 分为了一下几个考量部分。

基本的读写图片和读写像素，bluring，sobel，拼接图片。

3、 基本操作

基于 stb 这个库，我对它进行了进一步的包装便于使用。

首先定义了私有变量：

```
private:
    int width;
    int height;
    int channel; // rgb=3, grayscale=1
    unsigned char* data; //image information, saved line by line
```

然后是基本的读写操作：

```
bool readFile(const char* filename, const int pchannel = 0) {
    if(data)
        stbi_image_free(data);
    data = stbi_load(filename, &width, &height, &channel, pchannel); // read pic from file and get its width, height and channel.
    return data ? true : false;
}

void writeFile(const char* filename) {
    stbi_write_jpg(filename, width, height, channel, data, 70); //write pic to file by given width, height and channel. the last para means quality and 70 is enough.
}
```

由于 stb 这个库过于好用，所以不需要 opencv 过于复杂的配置，只需要 include 头文件并使用函数即可立即使用，需要指出：为了避免内存溢出，进行了很多避免内存溢出的判断。

```
void setPixel(unsigned char* d, const int x, const int y, const int c, const int _width, const int pchannel, const unsigned char pix) {
    d[y * _width * pchannel + x * pchannel + c] = pix;
}

void setPixel(const int x, const int y, const int c, const unsigned char pix) {
    data[y * width * channel + x * channel + c] = pix;
}
```

这两个 setPixel 均为设置像素，分别为设置自己的像素和其他类的像素。

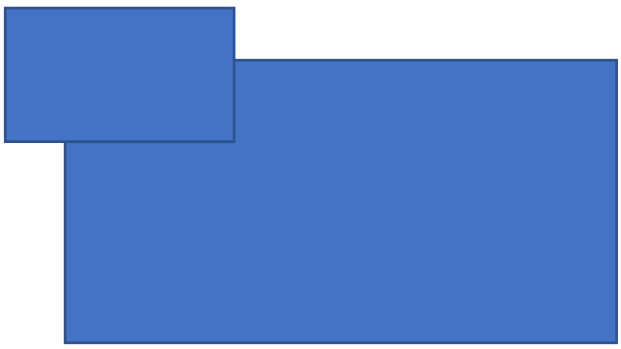
```
unsigned int getPixel(unsigned char* d, const int x, const int y, const int c, int pchannel) { //get pixel from any data, pchannel means the source's channel
    return d[y*width*pchannel + x * pchannel + c];
}
```

```
unsigned int getPixel(const int x, const int y, const int c) {
    return data[y*width*channel + x * channel + c];
} // different from the private member, giePixel, the function can only get pixel from its data.

unsigned int getIndex(const int i) {
    return data[i];
} // get pixel by its index
```

getPixel 和 getIndex 提供了多种多样的取像素的操作。

正是因为这一步的充分准备，所以接下来的步骤都并不难实现。

4、bluring



bluring 即对周围的像素进行取平均值，较为简单，唯一比较复杂的就是在边缘的判断，为了避免这样的问题，

```
int xstart = x - n / 2 < 0 ? 0 : x - n / 2;
int xend = x + n / 2 + 1 > width ? width : x + n / 2 + 1;
int ystart = y - n / 2 < 0 ? 0 : y - n / 2;
int yend = y + n / 2 + 1 > height ? height : y + n / 2 + 1;
int num = (xend - xstart) * (yend - ystart);
```

进行了平均区域的限制

```
for(int i = xstart; i < xend; ++i)
    for(int j = ystart; j < yend; ++j)
        sum += getPixel(data, i, j, c, channel);
ave = sum / num;
setPixel(output, x, y, c, width, pchannel, ave);
```

再通过以上的程序进行取平均值，并 setPixel 在新的画布上即可。
并且，根据我们学的的 cache 的原理，对程序的访问顺序进行了优化。

5、 sobel
sobel 边缘化处理的第一步是转化为灰度图。
只需将通道数变为 1，创建新的画布，用通过优化的公式（而非大家常用的乘那些小数点的公式），计算得到新的，并释放掉旧的内存即可。

```
void PixImage::grayscale() {
        unsigned char* output = create(1);
        for(int x = 0; x <width; ++x)
                for(int y = 0; y <height; ++y) {
                        unsigned char pix = (getPixel(x, y, 0) * 76 + getPixel(x, y, 1) * 150 + getPixel(x, y, 2) * 30) >> 8;
                        setPixel(output, x, y, 0, width, 1, pix);
                }
        stbi_image_free(data);
        data = output;
        channel = 1;
}
```

之后，通过给定的算法算出 gx,gy,g，在新的画布 sob 中依次带入。

```
void PixImage::sobel() {
        grayscale();
        unsigned int sum = 0;
        unsigned char *sob = new unsigned char[width * height], ave;
        for(int y = 1; y < height - 1; ++y)
                for(int x = 1; x < width - 1; ++x) {
                        unsigned int gx = getPixel(x + 1, y - 1, 0) + 2 * getPixel(x + 1, y, 0) + getPixel(x + 1, y + 1, 0) - getPixel(x -
                        unsigned int gy = getPixel(x - 1, y - 1, 0) + 2 * getPixel(x, y - 1, 0) + getPixel(x + 1, y - 1, 0) - getPixel(x -
                        unsigned char g = sqrt(gx * gx + gy * gy);
                        sob[y * width + x] = g;
                }
```

此后，算出像素的平均值，再依次对像素极端化即可。

① 对于一些边缘化，同样需要特殊处理，在这里和 bluring 不同，边缘值直接采取相邻像素即可，这样实现比较简单，也会和结果完全一致。

```
for(int i = 1; i < width - 1; ++i) {
        sob[i] = sob[width + i];
        sob[(height - 1) * width + i] = sob[(height - 2) * width + i];
}
for(int i = 1; i < height - 1; ++i) {
        sob[i * width] = sob[i * width + 1];
        sob[i * width + width - 1] = sob[i * width + width - 2];
}
sob[0] = sob[width + 1];
sob[width - 1] = sob[width + width - 2];
sob[(height - 1) * width] = sob[(height - 2) * width + 1];
sob[(height - 1) * width + width - 1] = sob[(height - 2) * width + width - 2];
```

② 根据对 scale 的理解，当 scale 为 4 时，mean 仅为 20 多，这个临界值并不是很理想，通过调试，认为 scale=6 时，效果比较明显且清晰。

③ 在求和过程中，由于像素的类型均为 unsigned char，会发生越界，所以对于 getPixel 和 getIndex 函数取像素，返回值均为 unsigned，避免溢出。

6、 拼接图片

为了可以再网站上直观展示，我添加了这一模块

在这一模块中，传入的第一个变量是一个 PixImage 数组，把这个数组中的图片和原图片水平拼接。

```
int _width = width;
for(int i = 0; i <size; ++i) {
        if(src[i]->getHeight() != height)
                src[i]->resize(src[i]->getWidth() * height / src[i]->getHeight(), height);
        if(src[i]->getChannel() == 1)
                src[i]->grayChannel();
        _width += src[i]->getWidth();
}
```

首先，把所有图片的高度调节到和 this 图片的高度相同，其次相加所有的宽度，设置新的画布。

之后，把 channel 为 1 的都统一为 3，这样才可以统一拼接，这一步骤只是简单的赋值，不做说明。

```
unsigned char* output = new unsigned char[_width*height*channel];
for(int y = 0; y <height; ++y) {
        int offset = width;
        for(int x = 0; x <width; ++x)
                for(int c = 0; c < 3; ++c)
                        setPixel(output, x, y, c, _width, 3, getPixel(x, y, c));
        for(int i = 0; i <size; ++i) {
                for(int x = 0; x <src[i]->getWidth(); ++x)
                        for(int c = 0; c < 3; ++c)
                                setPixel(output, offset + x, y, c, _width, 3, src[i]->getPixel(x, y, c));
                offset += src[i]->getWidth();
        }
}
```

在这之后，我们即可把三个图片拼接，即依次赋值。

如果有需求，我们可以把图片 resize，在这里用到了 stb 的 resize 相关模块，进行简单封装后，便可直接使用。

7、 主函数

```
void process(int start, int step) {
        PixImage src, blu, sob;
        char filename[20];
        for(int i = start; i <start + step; ++i) {
                printf("%d\n", i);
                sprintf(filename, "file/%d.jpg", i);
                src.readFile(filename);
                blu.copy(src);
                sob.copy(src);
                blu.bluring(5);
                sob.sobel();
                PixImage* a[2] = {&blu, &sob};
                src.combineHorizontal(a, 2);
                sprintf(filename, "ds/%dds.jpg", i);
                src.writeFile(filename);
        }
}

int main() {
        process(1, 4);
        return 0;
}
```

在主函数中，我们创建了三个 PixImage，分别为原图，模糊化和索贝尔，后将这三个图片拼接，并释放内存，存储图片。
文件相对路径和项目统一性很高，不需要重新设置。

六、网页展示
由于我有自己的个人网页，出于个人兴趣，把这些图片进行了网页展示。
在这里，我打算随机生成 32 个图片通过瀑布流的形式展示，并实现比较人性化的查看设计。
1、 图片的部署
首先，我把图片都部署在了 img.icser.me 服务器上，程序当然也基于路径问题进行了一些适配，最后，图片均位于 img.icser.me/ds 文件夹下，分别命名为 1.jpg ... 10000.jpg...和 1ds.jpg ... 10000ds.jpg 等等，分别为原图和处理后的合并的图片。
2、 服务器后端程序
首先是生成随机数的代码

```python
# coding: utf-8
import random

def image_random(maxv):
    l_num = random.sample(range(1, maxv + 1), 32)
    l_num = [str(i) for i in l_num]
    return l_num
```

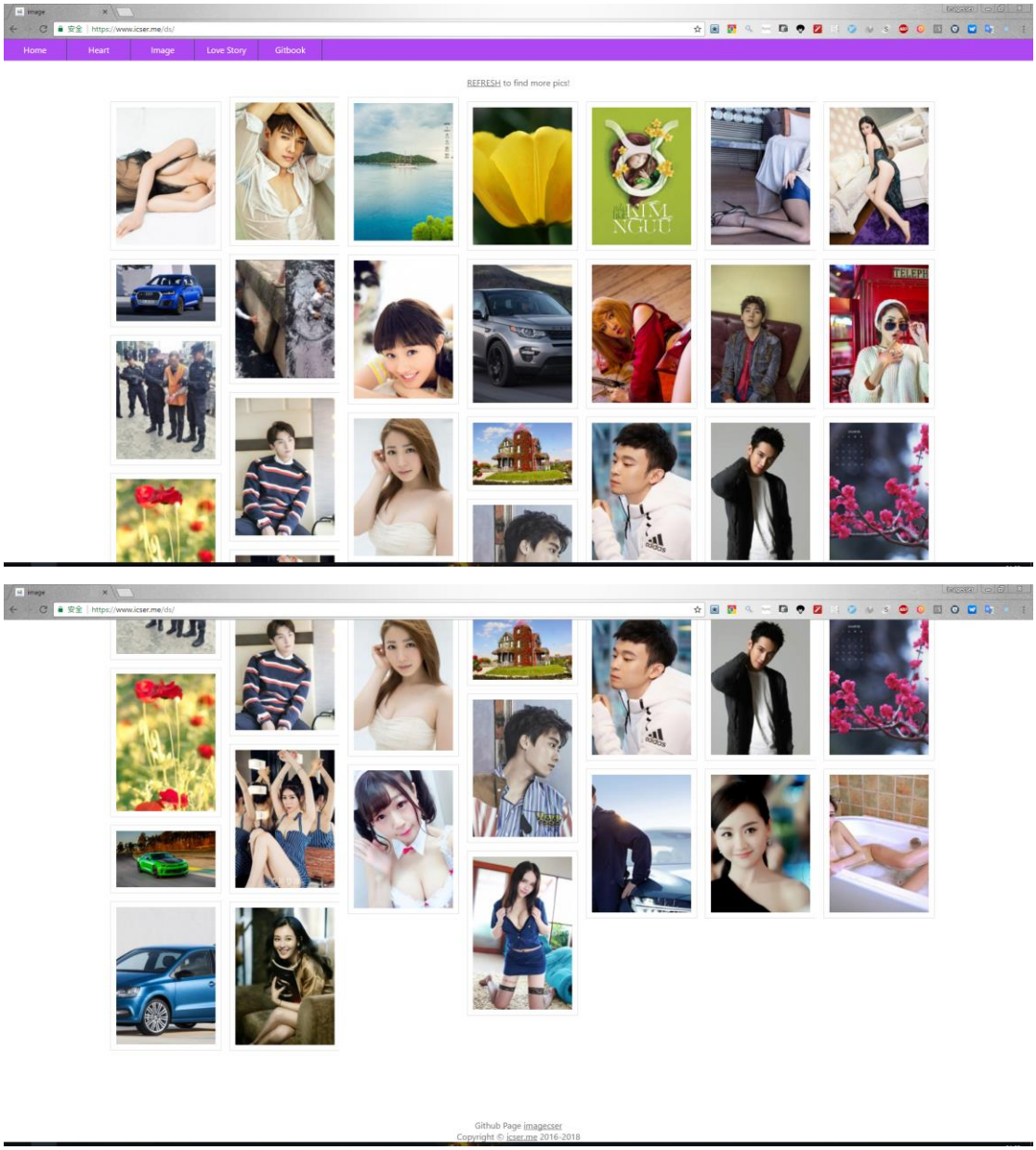
不需要做出解释，随机生成 32 个随机数

```python
@app.route('/ds/')
def ds_page():
    return render_template('image/index.html', l_src=image.image_random(9392), dirc="ds", suffix="ds")
```

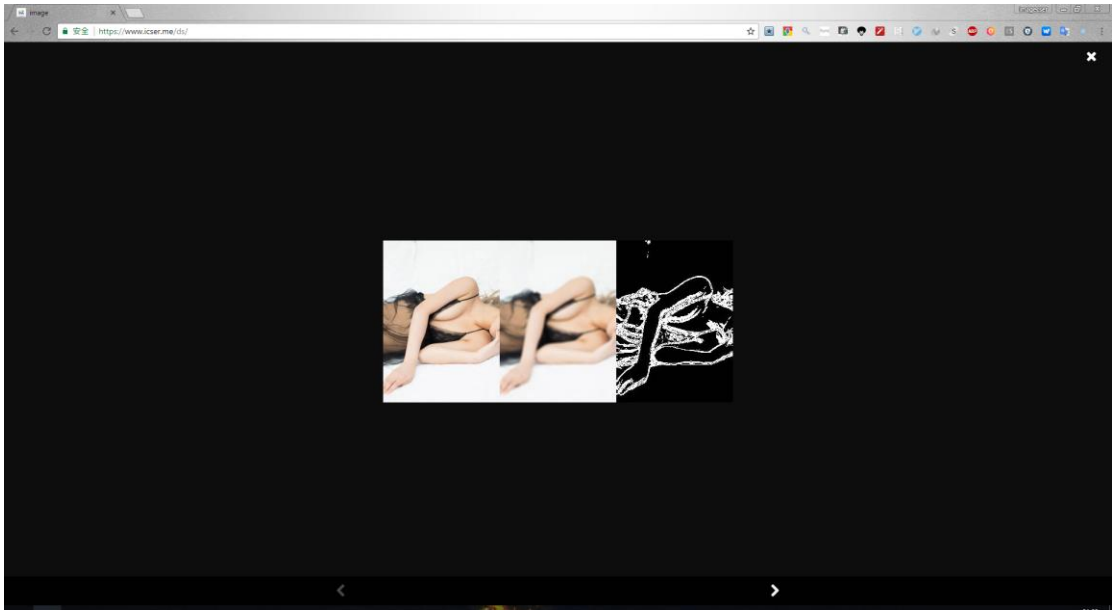
而后，在总的服务器代码中部署如上
向 index.html jinja 模板中传入 l_src, dirc, suffix 参数。

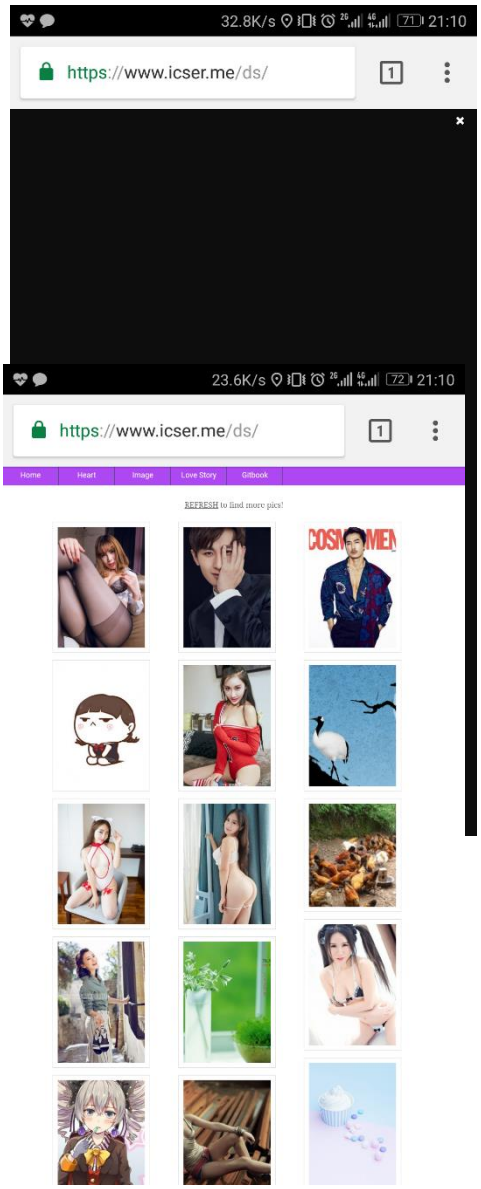
```
{% for i in l_src %}
{% set aurl = "https://img.icser.me/" + dirc + "/" + i + suffix + ".jpg" %}
{% set burl = "https://img.icser.me/" + dirc + "/" + i + ".jpg" %}
<a href="{{aurl}}" class="swipebox" title="">
<div class="box">
<div class="pic">
<img src="{{burl}}" alt="image">
</div>
</div>
</a>

{% endfor %}
```

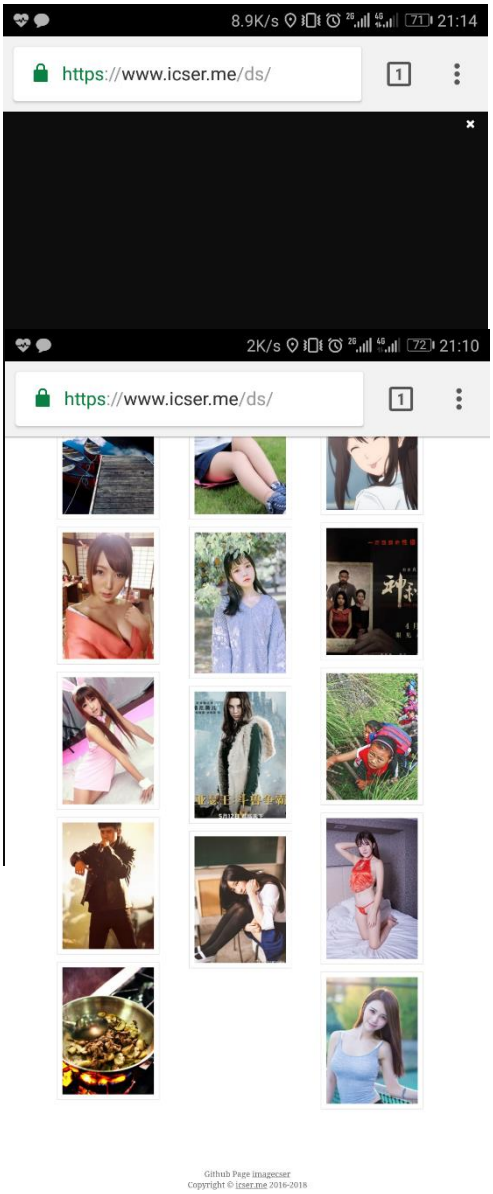
通过这样的代码分别指向 xxx.jpg 和 xxxds.jpg 即可指向相应图片。
在设计这一网页时，我使用了 swipebox 这一模板，使得网页同时适配电脑端和移动端的使用，并且查看图片比较流畅平滑。
3、 展示效果
具体展示效果：



下图为点击某一图片的展示，下方有翻页键。




32.8K/s 21:10
https://www.icser.me/ds/
Home
Heart
Image
Love Story
Gitbook
REFRESH to find more pics!


8.9K/s 21:14
https://www.icser.me/ds/
2K/s 21:10
https://www.icser.me/ds/
Github Page imagecser
Copyright © icser.me 2016-2018

在移动端，点击图片即可打开图片查看器查看处理后的图片，可以通过左滑右滑翻动图片，也可以通过向上滑动和向下滑动关闭图片查看器。
以上，完成了网页展示。

七、实验总结

在这次实验中，我们充分运用了数据结构课程学习的知识，如线性表，栈，递归，队列，图的生成树，Bloom Filter，hash 函数等等，巩固了知识。

在我已有 python，C++，前端知识的基础上，迅速上手，并完成了爬虫，图片处理，前端展示的代码实现。