链滴

Java 实现图片相似度比较 - 颜色分布法

作者：MingGH

原文链接：https://ld246.com/article/1658567125512

来源网站：链滴

文章首发于：Java 实现图片相似度比较 - 颜色分布法

0. 本文中实现的思路来自于以下博客：

相似图片搜索的原理（二）

相似图片搜索的原理

1. 如何判断两张图片的相似性有多少

本文的代码实现已经上传到github，需要的可以自行获取，地址：calculate-pic-looklike

想看详细版的，请看这篇文章：相似图片搜索的原理。

太长不看的简略版如下：

人类比较两张图片是否类似的时候，我们其实会依据图片中图形的轮廓是否在脑海中已经存在过进行想，然后大脑自动帮我们关联到看到的记忆上。

而对于计算机来说，判断两张图片的相似度靠的是 “感知哈希算法”。它会给每一张图片生成一个指纹”，然后比较两张图片的指纹，结果越接近就说明图片越相似。

怎么比较两个指纹的相似度，上面的文章中提到有：皮尔逊相关系数或者余弦相似度

Pearson相关系数（Pearson Correlation Coefficient）是用来衡量两个数据集合是否在一条线上面它用来衡量定距变量间的线性关系。

pearson相关系数衡量的是线性相关关系。若r=0，只能说x与y之间无线性相关关系，不能说无相关系。相关系数的绝对值越大，相关性越强：相关系数越接近于1或-1，相关度越强，相关系数越接近于，相关度越弱。

通常情况下通过以下取值范围判断变量的相关强度：

相关系数 0.8-1.0 极强相关

0.6-0.8 强相关

0.4-0.6 中等程度相关

0.2-0.4 弱相关

0.0-0.2 极弱相关或无相关

对于x,y之间的相关系数r ：

当r大于0小于1时表示x和y正相关关系

当r大于-1小于0时表示x和y负相关关系

当r=1时表示x和y完全正相关，r=-1表示x和y完全负相关

当r=0时表示x和y不相关

所以，比较两个图片的相似度在于得出图片指纹（数据集合），用这两个数据集合代入皮尔逊相关系就可以得到。

那么怎么从图片得到图片指纹呢?

1.1 图片到图片指纹

我们都知道任何图片在电子屏幕上显示都由光学三原色构成（红绿蓝），在计算机中我们对三原色用GB来进行表示，比如：

(255 ,180, 0) 表示的颜色是：

那么我们可以通过代码获取到一张图片的所有的像素点，然后把每个像素点进行拆分都可以得到RGB值，把这些RGB数值计数，那么就会得到：

(0,0,0) 出现次数10000次
(0,0,1) 出现次数25次
...
(0,0,1) 出现次数90次

把这些用次数收集起来，得到集合 {10000,25,...,90}就是**图片指纹**。

分别对两个图片都这样操作，然后得到这个数据集合，再通过上面说到的皮尔逊相关系数或者余弦相度 进行对比，就知道这两张图片的相似度了。

但是如果我们对R,G,B都采用0-255的范围进行比较，那么产生的可能性就有：$256^{3}=16777216$

为了我们简化计算，一般采用的是这样的方式：将0～255分成四个区：0～63为第0区，64～127为第区，128～191为第2区，192～255为第3区。这意味着红绿蓝分别有4个区，总共可以构成64种组合4的3次方）。

如下图所示：

红	绿	蓝	像素数量	红	绿	蓝	像素数量	红	绿	蓝	像素数量	红	绿	蓝	像素数量
0	0	0	7414	1	0	0	891	2	0	0	1146	3	0	0	11
0	0	1	230	1	0	1	13	2	0	1	0	3	0	1	0
0	0	2	0	1	0	2	0	2	0	2	0	3	0	2	0
0	0	3	0	1	0	3	0	2	0	3	0	3	0	3	0
0	1	0	8	1	1	0	592	2	1	0	2552	3	1	0	856
0	1	1	372	1	1	1	3462	2	1	1	9040	3	1	1	1376
0	1	2	88	1	1	2	355	2	1	2	47	3	1	2	0
0	1	3	0	1	1	3	0	2	1	3	0	3	1	3	0
0	2	0	0	1	2	0	0	2	2	0	0	3	2	0	0
0	2	1	0	1	2	1	101	2	2	1	8808	3	2	1	3650
0	2	2	10	1	2	2	882	2	2	2	53110	3	2	2	6260
0	2	3	1	1	2	3	16	2	2	3	11053	3	2	3	109
0	3	0	0	1	3	0	0	2	3	0	0	3	3	0	0
0	3	1	0	1	3	1	0	2	3	1	0	3	3	1	0
0	3	2	0	1	3	2	0	2	3	2	170	3	3	2	3415
0	3	3	0	1	3	3	0	2	3	3	17533	3	3	3	53929

将表中最后一栏提取出来，组成一个64维向量(7414, 230, 0, 0, 8, ..., 109, 0, 0, 3415, 53929)。这个量就是这张图片的特征值或者叫"指纹"。

然后代入皮尔逊相关系数进行计算

2. 代码实现部分

对原理我们了解时候，通过Java代码实现有X部分：

- 获取一张图片所有的像素点，并且获取到这个像素点的RGB数值
- 对所有的像素点进行统计，得到图片指纹
- 代入皮尔逊相关系数

2.1 获取一张图片的所有像素点以及RGB数值

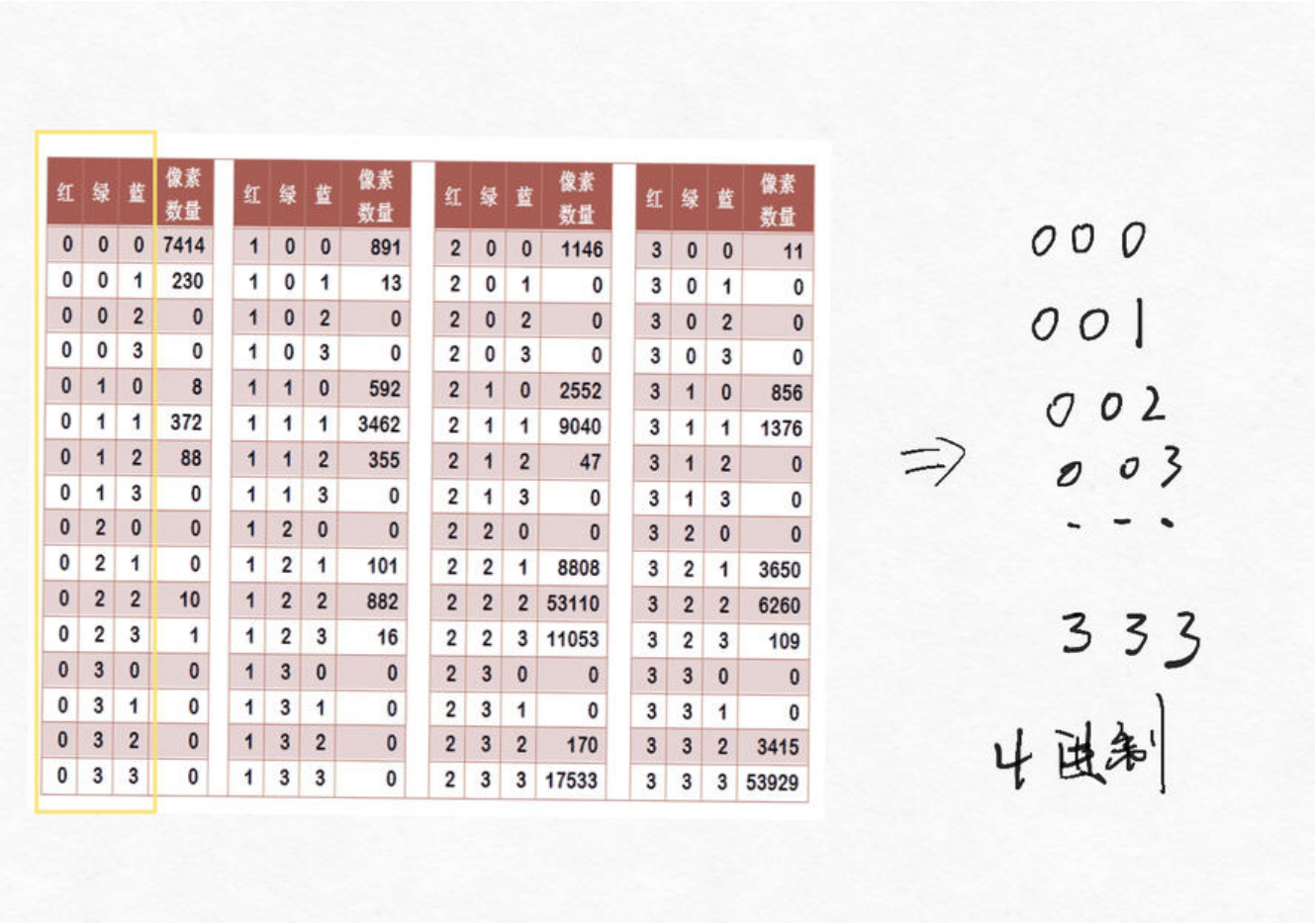
```
BufferedImage bimg = ImageIO.read(new File(path));
for(int i = 0; i < bimg.getWidth(); i++){
    for(int j = 0; j < bimg.getHeight(); j++){
        Color color = new Color( bimg.getRGB(i, j));
        int r = color.getRed();
        int g = color.getGreen();
        int b = color.getBlue();

    }
}
```

2.2 对像素点统计，得到图片指纹

这一步就比较关键了，我们拿到像素点之后，观察下图标黄的这三列，可以看到红绿蓝组成的数字正是从 001～333，符合四进制的规律。

那就意味着我们在上面对像素点进行遍历的时候，计算一个像素点属于哪个区（0-3），就把值放入进制的容量大小的数组。



红	绿	蓝	像素数量
0	0	0	7414
0	0	1	230
0	0	2	0
0	0	3	0
0	1	0	8
0	1	1	372
0	1	2	88
0	1	3	0
0	2	0	0
0	2	1	0
0	2	2	10
0	2	3	1
0	3	0	0
0	3	1	0
0	3	2	0
0	3	3	0

红	绿	蓝	像素数量
1	0	0	891
1	0	1	13
1	0	2	0
1	0	3	0
1	1	0	592
1	1	1	3462
1	1	2	355
1	1	3	0
1	2	0	0
1	2	1	101
1	2	2	882
1	2	3	16
1	3	0	0
1	3	1	0
1	3	2	0
1	3	3	0

红	绿	蓝	像素数量
2	0	0	1146
2	0	1	0
2	0	2	0
2	0	3	0
2	1	0	2552
2	1	1	9040
2	1	2	47
2	1	3	0
2	2	0	0
2	2	1	8808
2	2	2	53110
2	2	3	11053
2	3	0	0
2	3	1	0
2	3	2	170
2	3	3	17533

红	绿	蓝	像素数量
3	0	0	11
3	0	1	0
3	0	2	0
3	0	3	0
3	1	0	856
3	1	1	1376
3	1	2	0
3	1	3	0
3	2	0	0
3	2	1	3650
3	2	2	6260
3	2	3	109
3	3	0	0
3	3	1	0
3	3	2	3415
3	3	3	53929

所以我们在对图片进行遍历前，创建一个分区数^{3}=List容量数 的集合，然后对集合进行初始化。

```
//比较等级，也就是256个像素点分区后的区块数
public static int compareLevel = 4;

public static void main(String[] args) throws IOException {
    final String pic1Path = Objects.requireNonNull(Calculate.class.getClassLoader().getResource
"pic1.jpeg")).getPath();
    final String pic2Path = Objects.requireNonNull(Calculate.class.getClassLoader().getResource
"pic2.jpeg")).getPath();
    final List<Double> origin = getPicArrayData(pic1Path);
    System.out.println(origin);
    final List<Double> after = getPicArrayData(pic2Path);
    System.out.println(after);
    System.out.println(PearsonDemo.getPearsonBydim(origin, after));
}


public static List<Double> getPicArrayData(String path) throws IOException {
    BufferedImage image = ImageIO.read(new File(path));
```

```
    //初始化集合
    final List<Double> picFingerprint = new ArrayList<>(compareLevel*compareLevel*compar
Level);
    IntStream.range(0, compareLevel*compareLevel*compareLevel).forEach(i->{
        picFingerprint.add(i, 0.0);
    });
    //遍历像素点
    for(int i = 0; i < image.getWidth(); i++){
        for(int j = 0; j < image.getHeight(); j++){
            Color color = new Color(image.getRGB(i, j));
            //对像素点进行计算
            putIntoFingerprintList(picFingerprint, color.getRed(), color.getGreen(), color.getBlue());

        }
    }

    return picFingerprint;
}

/**
 * 放入像素的三原色进行计算，得到List的位置
 * @param picFingerprintList picFingerprintList
 * @param r r
 * @param g g
 * @param b b
 * @return
 */
public static List<Double> putIntoFingerprintList(List<Double> picFingerprintList, int r, int g,
nt b){
    //比如r g b是126, 153, 200 且 compareLevel为16进制，得到字符串：79c ,然后转10进制，这
数字就是List的位置
    final Integer index = Integer.valueOf(getBlockLocation(r) + getBlockLocation(g) + getBlock
ocation(b), compareLevel);
    final Double origin = picFingerprintList.get(index);
    picFingerprintList.set(index, origin + 1);
    return picFingerprintList;
}

/**w
 * 计算 当前原色应该分在哪个区块
 * @param colorPoint colorPoint
 * @return
 */
public static String getBlockLocation(int colorPoint){
    return IntStream.range(0, compareLevel)
            //以10进制计算分在哪个区块
            .filter(i -> {
                int areaStart = (256 / compareLevel) * i;
                int areaEnd = (256 / compareLevel) * (i + 1) - 1;
                return colorPoint >= areaStart && colorPoint <= areaEnd;
            })
            //如果compareLevel大于10则转为对应的进制的字符串
            .mapToObj(location -> compareLevel > 10 ? Integer.toString(location, compareLevel) :
ocation+"")
```

```
            .findFirst()
            .orElseThrow();
}
```

其中代码中 compareLevel就是256的因数，能够使用的数字有：1,2,4,8,16,32,64,128,256。每个数代表的是256个像素会被均分成多少份。

在方法 putIntoFingerprintList中，分别对这个像素点的R G B原色进行计算，计算过程便是方法 getBockLocation，计算的原理如下:

2.2.1 获取应该放入区块的位置

假设原色的值为：x，区块数为：n

第一个区块的范围为：$\left [0,\frac{256}{n} *1 -1\right]$

第二个区块的范围为：$\left [\frac{256}{n},\frac{256}{n} *2 -1\right]$

那么第 i 次的区块范围为：$\left [\frac{256}{n} * i,\frac{256}{n} *(i+1) -1\right]$

那么：$x\in \left [\frac{256}{n} * i,\frac{256}{n} *(i+1) -1\right]$

此时这个 i 就是x应该放入区块的位置。

这里举个例子：

假设原色的值为235

那么带入上面的公式：

0～63为第0区，64～127为第1区，128～191为第2区，192～255为第3区

那么235属于第3区

转变为代码的话就是：

```
public static String getBlockLocation(int colorPoint){
    return IntStream.range(0, compareLevel)
            //以10进制计算分在哪个区块
            .filter(i -> {
                int areaStart = (256 / compareLevel) * i;
                int areaEnd = (256 / compareLevel) * (i + 1) - 1;
                return colorPoint >= areaStart && colorPoint <= areaEnd;
            })
            //如果compareLevel大于10则转为对应的进制的字符串
            .mapToObj(location -> compareLevel > 10 ? Integer.toString(location, compareLevel) :
ocation+"")
            .findFirst()
            .orElseThrow();
}
```

在这个方法中：return colorPoint >= areaStart && colorPoint <= areaEnd;

这一行代码代表的意思就是：

如果：$colorPoint \geq \frac{256}{n} * i$ 且 $colorPoint\leq \frac{256}{n} *(i+1) -1$

那么为true，表示符合我们要的数据

最后放入List中，此时得到的这个List就是图片指纹

2.3 代入皮尔逊相关系数

这一步网上已经有很多Java实现的代码，就不重复造轮子了。贴一下网上找到的代码：

```
package run.runnable.calculatepiclooklike.utils;

import java.util.ArrayList;
import java.util.List;

/**
 * 皮尔逊相关系数
 */
public class PearsonDemo {
    public static Double getPearsonBydim(List<Double> ratingOne, List<Double> ratingTwo) {

        try {
            if(ratingOne.size() != ratingTwo.size()) {//两个变量的观测值是成对的，每对观测值之间相
独立。
                if(ratingOne.size() > ratingTwo.size()) {//保留小的处理大
                    List<Double> temp = ratingOne;
                    ratingOne = new ArrayList<>();
                    for(int i=0;i<ratingTwo.size();i++) {
                        ratingOne.add(temp.get(i));
                    }
                }else {
                    List<Double> temp = ratingTwo;
                    ratingTwo = new ArrayList<>();
                    for(int i=0;i<ratingOne.size();i++) {
                        ratingTwo.add(temp.get(i));
                    }
                }
            }
            double sim = 0D;//最后的皮尔逊相关度系数
            double commonItemsLen = ratingOne.size();//操作数的个数
            double oneSum = 0D;//第一个相关数的和
            double twoSum = 0D;//第二个相关数的和
            double oneSqSum = 0D;//第一个相关数的平方和
            double twoSqSum = 0D;//第二个相关数的平方和
            double oneTwoSum = 0D;//两个相关数的乘积和
            for(int i=0;i<ratingOne.size();i++) {//计算
                double oneTemp = ratingOne.get(i);
                double twoTemp = ratingTwo.get(i);
                //求和
                oneSum += oneTemp;
                twoSum += twoTemp;
                oneSqSum += Math.pow(oneTemp, 2);
                twoSqSum += Math.pow(twoTemp, 2);
                oneTwoSum += oneTemp*twoTemp;
            }
            double num = (commonItemsLen*oneTwoSum) - (oneSum*twoSum);
            double den = Math.sqrt((commonItemsLen * oneSqSum - Math.pow(oneSum, 2)) * (c
mmonItemsLen * twoSqSum - Math.pow(twoSum, 2)));
```

```
            sim = (den == 0) ? 1 : num / den;
            return sim;
        } catch (Exception e) {
            return null;
        }
    }

    public static double getPearsonCorrelationScore(List<Double> x, List<Double> y) {
        if (x.size() != y.size())
            throw new RuntimeException("数据不正确！");
        double[] xData = new double[x.size()];
        double[] yData = new double[x.size()];
        for (int i = 0; i < x.size(); i++) {
            xData[i] = x.get(i);
            yData[i] = y.get(i);
        }
        return getPearsonCorrelationScore(xData,yData);
    }

    public static double getPearsonCorrelationScore(double[] xData, double[] yData) {
        if (xData.length != yData.length)
            throw new RuntimeException("数据不正确！");
        double xMeans;
        double yMeans;
        double numerator = 0;// 求解皮尔逊的分子
        double denominator = 0;// 求解皮尔逊系数的分母

        double result = 0;
        // 拿到两个数据的平均值
        xMeans = getMeans(xData);
        yMeans = getMeans(yData);
        // 计算皮尔逊系数的分子
        numerator = generateNumerator(xData, xMeans, yData, yMeans);
        // 计算皮尔逊系数的分母
        denominator = generateDenomiator(xData, xMeans, yData, yMeans);
        // 计算皮尔逊系数
        result = numerator / denominator;
        return result;
    }

    /**
     * 计算分子
     *
     * @param xData
     * @param xMeans
     * @param yData
     * @param yMeans
     * @return
     */
    private static double generateNumerator(double[] xData, double xMeans, double[] yData,
ouble yMeans) {
        double numerator = 0.0;
        for (int i = 0; i < xData.length; i++) {
            numerator += (xData[i] - xMeans) * (yData[i] - yMeans);
```

```
        }
        return numerator;
    }

    /**
     * 生成分母
     *
     * @param yMeans
     * @param yData
     * @param xMeans
     * @param xData
     * @return 分母
     */
    private static double generateDenomiator(double[] xData, double xMeans, double[] yData,
ouble yMeans) {
        double xSum = 0.0;
        for (int i = 0; i < xData.length; i++) {
            xSum += (xData[i] - xMeans) * (xData[i] - xMeans);
        }
        double ySum = 0.0;
        for (int i = 0; i < yData.length; i++) {
            ySum += (yData[i] - yMeans) * (yData[i] - yMeans);
        }
        return Math.sqrt(xSum) * Math.sqrt(ySum);
    }

    /**
     * 根据给定的数据集进行平均值计算
     *
     * @param
     * @return 给定数据集的平均值
     */
    private static double getMeans(double[] datas) {
        double sum = 0.0;
        for (int i = 0; i < datas.length; i++) {
            sum += datas[i];
        }
        return sum / datas.length;
    }
}
```

2.4 实现效果

计算得出的结果我们可以参考这个：

通常情况下通过以下取值范围判断变量的相关强度：

相关系数 0.8-1.0 极强相关

0.6-0.8 强相关

0.4-0.6 中等程度相关

0.2-0.4 弱相关

0.0-0.2 极弱相关或无相关

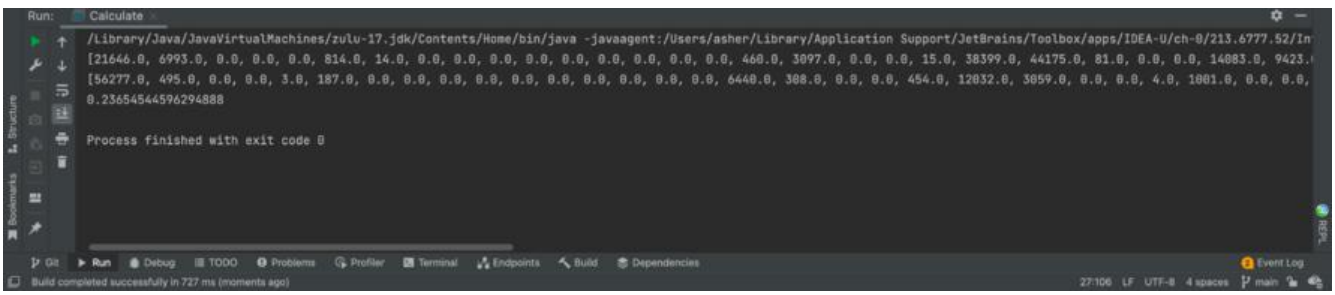
不相似图片计算

使用图片1，如下：

图片2，如下：

原文链接：Java 实现图片相似度比较 - 颜色分布法

compareLevel设置为4，进行相似度比较，得到的值为：0.23654544596294888



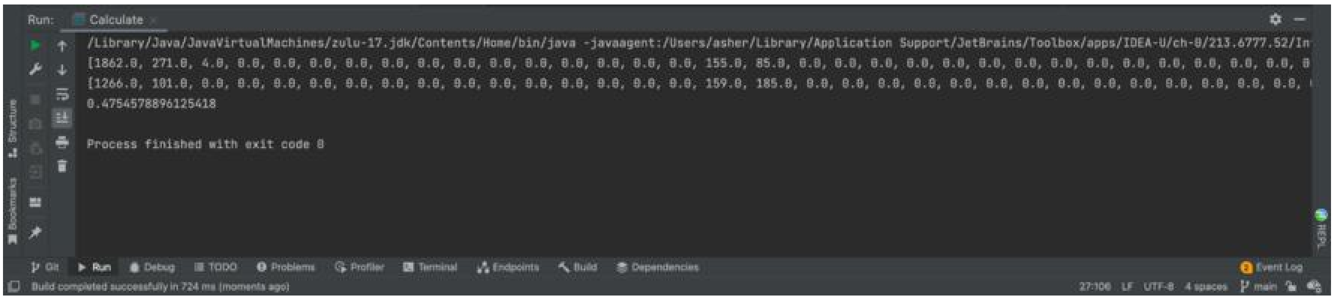
原文链接：Java 实现图片相似度比较 - 颜色分布法

相似图片计算

使用图片3，如下：

使用图片4，如下：

compareLevel设置为16，进行相似度比较，得到的值为：0.4754578896125418



代码实现已经上传到github，需要的可以自行获取，地址：calculate-pic-looklike

2.5 能不能不用list，而是用Map进行实现

在完成了上述的内容之后，这是我脑海中冒出来的一个想法。

在Map中这样进行存放 Map<像素计算得到的下标，总计>

实现后发现是不行的，因为一张图片里面经常不会存在所有颜色像素点，如果不存在这个像素点，那Map中的key是不完整的，那么得到的这个图片指纹也是不完整的。

比如身份证：

号码组成

可以通过冗余校验的方式来核查身份号码最后的校验和是否属实。

1	1	0	1	0	2	Y	Y	Y	Y	M	M	D	D	8	8	8	X
地址码						出生日期码								顺序及性别码			校验和

比如要有**地址码+出生日期码+(顺序及性别码)+校验和** 才组成一个完整的身份证

而使用Map的实现就相当于拿取的身份证是 **地址码+(顺序及性别码)+校验和** ，缺失了其中一部分。

附上Map部分的实现代码：

```
public static void putIntoFingerprintMap(Map<Integer, Integer> picFingerprintMap, int r, int g
 int b){
    final Integer picFingerprint = Integer.valueOf(getBlockLocation(r) + getBlockLocation(g) +
etBlockLocation(b), compareLevel);
    Integer value = picFingerprintMap.containsKey(Integer.valueOf(picFingerprint)) ? picFinger
rintMap.get(Integer.valueOf(picFingerprint)) + 1 : 1;
    picFingerprintMap.put(Integer.valueOf(picFingerprint), value);
}


public static List<Double> getPicArrayDataByMap(String path) throws IOException {
    BufferedImage bimg = ImageIO.read(new File(path));

    final Map<Integer, Integer> picFingerprintMap = new HashMap<>();

    for(int i = 0; i < bimg.getWidth(); i++){
        for(int j = 0; j < bimg.getHeight(); j++){
            //输出一列数据比对
            Color color = new Color( bimg.getRGB(i, j));
            int r = color.getRed();
            int g = color.getGreen();
            int b = color.getBlue();
            putIntoFingerprintMap(picFingerprintMap, r, g, b);
        }
    }

    final List<Integer> keys = picFingerprintMap.keySet().stream().sorted().collect(Collectors.toL
st());
    final ArrayList<Double> picFingerprintList = new ArrayList<>(keys.size());
    keys.forEach(key->{
        picFingerprintList.add(Double.valueOf(picFingerprintMap.get(key)));
    });
    return picFingerprintList;
}
```

3. 博客中信息的来源

相似图片搜索的原理（二）

相似图片搜索的原理

协同过滤算法