用js实现简单的验证码识别

后续会更新：将js代码等打包成chrome扩展程序，这样就可以让浏览器自动识别，完全傻瓜式使用啦～！

其实整篇文章难度不高，网上也有很多java、c等的代码。只是当时我写代码的时候，没有找到纯js可以用的代码和库，不能打包成chrome扩展，用起来还是不太方便的。所以在验证了思路的可行性后，我就大致写下来，给他人以方便吧。

目前有多种验证码识别思路，限于能力有限，我只好采用了最简单的机器学习。目标验证码也比较简单，如：yzm/img/4.jpg（含字母也一样）。识别控制速度在0.1秒以内的话，正确率在99.99999%（因为一直是识别正确哈哈哈）。

在动手之前，先梳理一下大致思路，方便比较独立的同学自己尝试完成代码：

1、先分析网页DOM结构，载入验证码图片。

2、将图片画到canvas上，拿到图片的像素数据。

3、先后对图片进行二值化、腐蚀膨胀、切割、旋转、缩放处理。

4、记录处理后的单个数字的二值化数据，并人工录入真实数字。

5、重复训练后。

6、识别时，用处理后的图像与库中数据对比，取得最相近的数据，得到真实数字。

（以下优化）

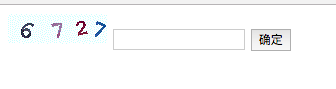
7、数据量大时，可以取前几个相似数据，并按权重从中选出最可能的数字，以提高准确度。

8、也可查找到相似度足够高时停止搜索，取其作为最后识别结果，以提升效率。

大神们可以直接去写了，我这低级简单的代码会遭你们嘲笑的。。。比较急于求成的同学也可以不用看了，回头直接拿demo去修改吧！

好吧既然你看到这里了，我就尽量说的清楚明白一点。

在动手之前，我简单模拟一下需要输入验证码的网站，效果如下：



好吧，是真的简单…点击图片可以更换验证码，输入框用来输入，按钮模拟提交，如下：

我们就假装他作为我们要自动识别的目标。

一、分析网页DOM结构，载入验证码图片。

我们可以看到，验证码的url是：img/0.jpg。我这里的url会变化，是为了模拟更换验证码的过程。但实际上，由于验证码绝大多数为后台生成的，所以地址是固定的。那么我们很容易就可以拿到图片数据：new一个Image，赋值url即可（直接get到img元素也行）。参考代码：

**var *img*** = **document**.getElementById(**"img"**);

二、将图片画到canvas上，拿到图片的像素数据。

要将图片画到canvas上，首先要创建一个canvas并初始化。参考代码：

**var *canvas1*** = document.createElement(**"canvas"**); **document**.getElementsByTagName(**"body"**)[0].appendChild(canvas1);  
***canvas1***.**style**.**backgroundColor** = **"cornsilk"**;  
**var *ctx1*** = ***canvas1***.getContext(**"2d"**);

随后，将图片绘制上去。参考代码：

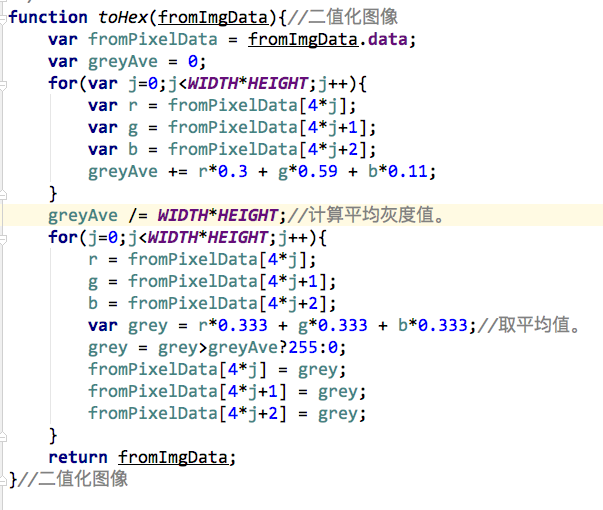
***ctx1***.drawImage(***img***,0,0,***img***.**width**,***img***.**height**);

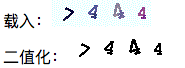
然后我们就可以利用canvas，拿到图片的像素数据。参考代码：

**var** imgData = ***ctx1***.getImageData(0,0,***WIDTH***,***HEIGHT***);

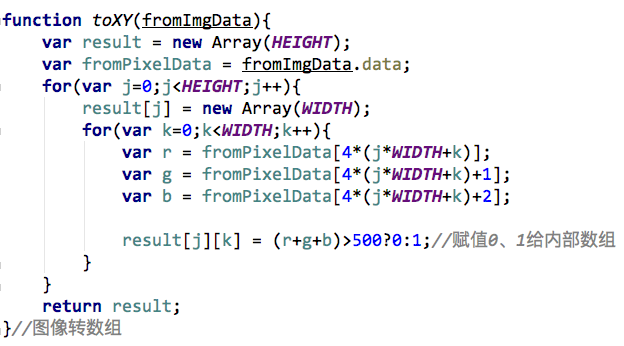
三、先后对图片进行二值化、腐蚀膨胀、切割、旋转、缩放处理。

这部分是图像识别的重点，直接影响到识别准确率和速度。复杂的验证码还应加上去躁等处理过程。比如可以检测贯穿的横线并消除，或者将颜色高度统一的背景去掉等等。我们的图片几乎没有干扰，只有简单的旋转和缩放，故直接进行二值化操作（二值化也能去掉少量的干扰）。

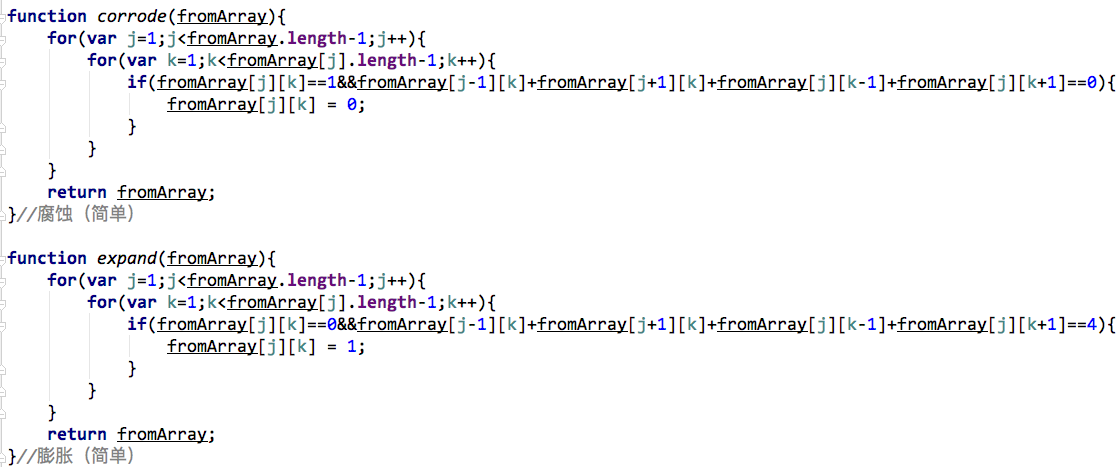
1、二值化操作的思路是：计算图片的平均灰度作为阈值，比阈值大的置为纯黑，反之纯白。参考代码：

二值化后，效果如图：可以发现，简单的背景色是可以去掉的。

二值化处理之后，就可以将图片转换成数组（存0或1）来保存了。参考代码如下：



2、接下来是腐蚀、膨胀。腐蚀的基本思路在于，将所有白色周围的像素都置成白色，以此来消除游离的个别黑色像素点噪声。膨胀正好相反，将黑色周围置成黑色，消除数字内部的个别白色。同时，腐蚀、膨胀的操作可以让图片更加平滑。参考代码：



由于我们的图片背景干扰不是很强烈，所以基本看不出差别。不过对于计算机来说，还是有不同的哟～尤其是背景复杂的图片，这一步很好用。

3、切割。

由于我们的图片内各数字没有粘连，所以切割时只从上至下，从左至右需要扫描图片，发现图片某一竖行均为白色，就切一刀。有粘连的验证码比较困难，暂时不讨论了。参考代码：



切割后，左右的空白因为都被切了，就没有了。但是上下仍然存在空白，所以进行处理。这里比较简单，就不放代码了，思路和切割类似，但简单很多。

4、旋转、缩放。

其实旋转不是必要的。没有旋转的步骤，可以用更多的数据量训练来弥补。同理，缩放也不是必须的。先大致讲一下思路：旋转和缩放都再次利用了canvas，将图片画上去之后，利用canvas的方法操作图片旋转或缩放，之后再把数据拿下来，就像我们最开始读图片时做的一样。旋转时，取顺时针逆时针各90度，取左右宽度最窄的角度，当作数字站立的旋转角度。缩放时，直接按预设长宽画图即可。这里我就只写了缩放。处理后再转换回数组形式。

处理后效果如图：屏幕快照%202017-08-12%20下午4.09.15.png

四、记录处理后的单个数字的二值化数据，并人工录入真实数字。

到这里，图像处理就搞定了，后面的工作就比较简单了。我们把上一步得到的数组和真实的数字一起保存起来。这个过程可以有很多方法。我当时采取了大家一起录入的方式，所以搭建了PHP+MySQL的服务器，用数据库存储。这块就不详述了，大家各显神威。

五、重复训练

为了方便训练，我直接在页面里增加了手动输入的地方，提交后刷新验证码，继续提交。提交20个验证码（20\*4=80个数字）后，便经常可以正确识别出4位验证码，在单个数字的数据量在300左右时（大约需要300/4=75个验证码），识别效率已经在95%以上。在500左右时已经基本见不到错误识别的情况了，这时候已经可以写代码实现自我训练了。此时识别一次大约需要0.06秒。

六、识别时，用处理后的图像与库中数据对比，取得最相近的数据，得到真实数字。

这块也比较简单。训练完成后，我将数据库数据导出，保存成了一个大的数组，直接用js就可以读了。识别时遍历所有的数据，按像素点逐一比较。由于尺寸做了归一化，所以直接数有多少像素匹配即可。匹配数量最多的即为识别出的结果。我只找到了最开始写的PHP代码，先放一下吧，有点懒得再写js了…：

七、优化部分

这块就大家自己看着来吧，因为我的图片不是很复杂，数据量也不是很大（千条级别），所以也没啥优化的必要，每次识别大约0.1秒吧。所以我只是没事干，做了之前大纲里写了那两个优化。其实我感觉主要的优化方向还是图像处理那块，尽量减少干扰，才能提高效率，也能检测更复杂的验证码。

好累，先休息下，看看有没有人看吧…（我感觉应该没多少人= =）