首先7位车牌的二维码内含数字，方便进行校验

并且以后必定是生产7位以后的车牌，而不会再生产6位

当然利用二维码来识别车牌上的数字是完全可以的，但是并不是数据挖掘课的目标。

二维码内包含的内容：（https://common.ofo.so/app/?utm\_source=bikeplate&utm\_medium=1201149　（其中1201149是车牌号码））

首先先对7位ofo车牌做个调查

车牌宽12.1cm



车牌高8cm



数字部分 长4.8cm，数字1的宽度为0.4cm



数字高1.4cm



除1外数字宽度为0.6cm



二维码宽 4.5cm



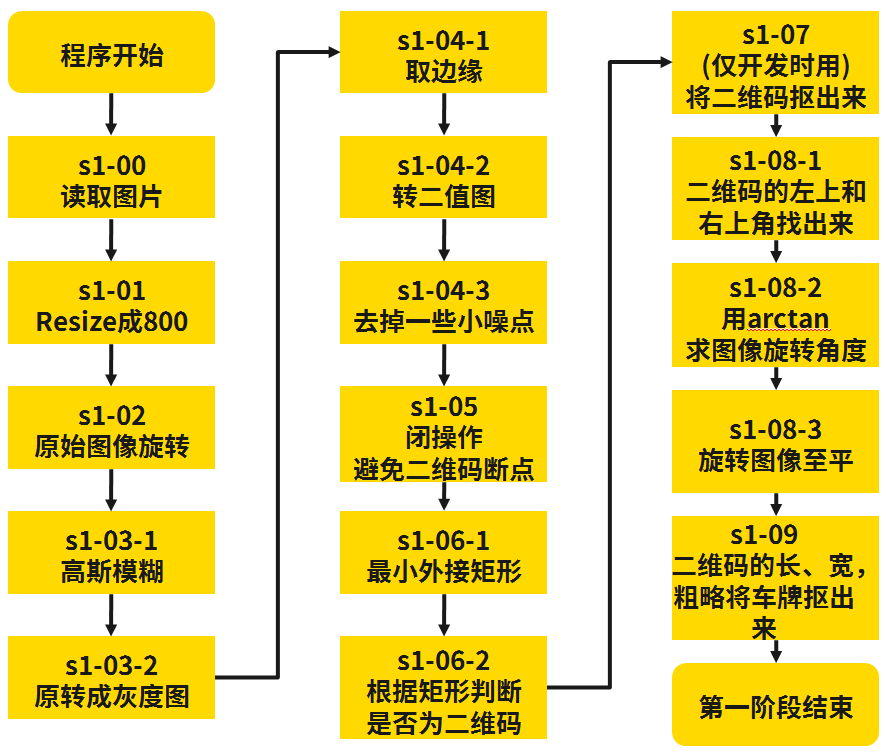
二维码 高4.5cm



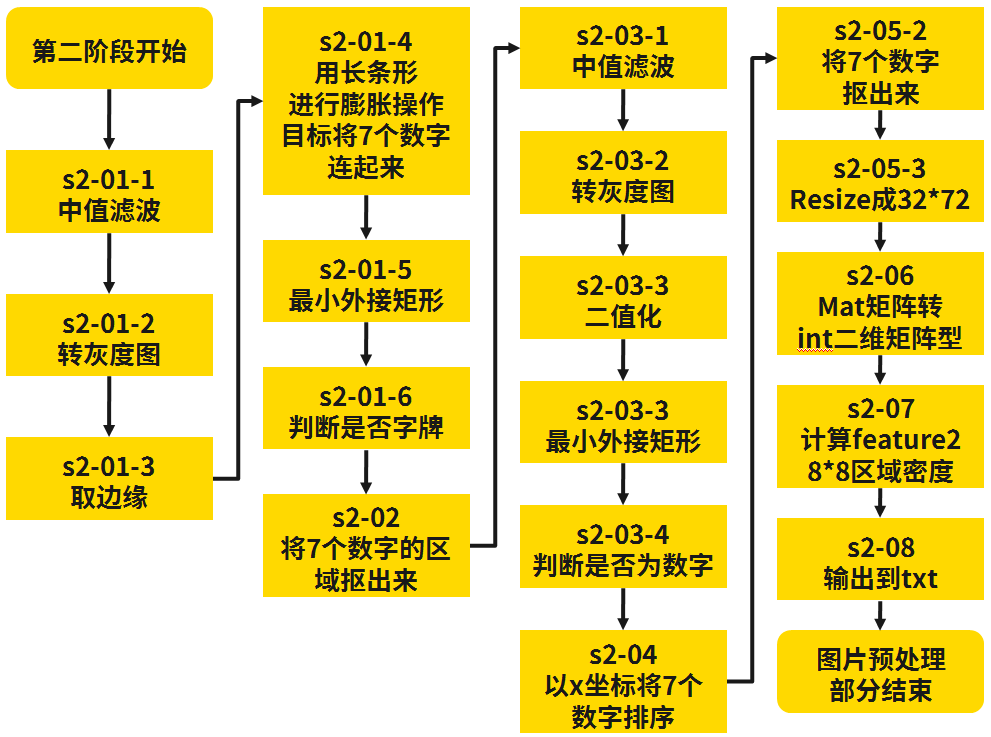
每一张包含ofo车牌的图片，都需要进行预处理，才能进行数据挖掘。

整个图片预处理，共有2个大阶段。

第1阶段的目标是将二维码找出来，并利用二维码偏转角度将7位数字的部分粗略抠出来。



第2阶段，分为二个部分：第一部分将7个数字的区域抠出来。第二部分将7个数字抠出来并提取特征



那为什么不直接在第一阶段就将7位数字的区域抠出来？因为ofo车牌上的“ofo bicycle”与7位数字的区域大小相当接近，如果直接抠出来的业务逻辑是可以，但是我当时没有想到。(可以看谁离二维码的y坐轴太近的话，就去掉)



下面就开始step by step讲解

s1-00，读取图片

这步没有什么好说的，确认能读取并显示出来。相当于是hello world。但要注意路径用“/”而不用“\”(“\”是转义符)

|  |
| --- |
| //s1-00:imread  cv::Mat srcImg=cv::imread(path+filename); //绝对路径。路径用/而不用\，否则会报错  cv::namedWindow("s1-00");  cv::imshow("s1-00",srcImg); |

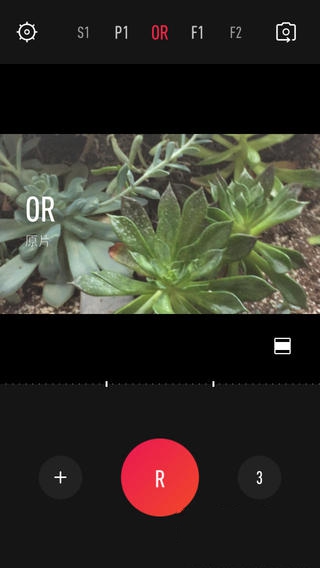
s1-01，resize成800px

为什么要选800px呢？

因为7位ofo车牌的数字比6位的小太多，如果 640px以下时，区分度不高。而太高分辨率处理又特别耗时。经过多次试验选择了800x800px。

为什么要选择正方形？

因为Android的后置摄像头默认是横向，而一开始是想做成像vue(一款视频拍摄app)那样相纵屏，但横向拍摄。但是相应的代码没有找到......



**毕竟不能反人类的让别人横着拿手机去拍照识别**，而查看自己的手机，意外地发现支持正方形拍照。所以，就立即决定用正方形来布局。

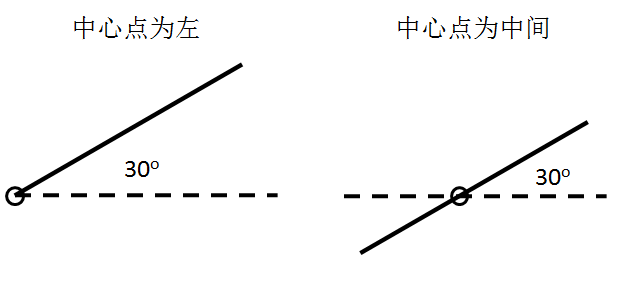
|  |
| --- |
| cv::resize(lastImg,resizeImg,Size(800,800)); |

s1-02，原始图像旋转

为什么要旋转？

因为Android手机竖着拿时，后置摄像头拍出来的图片是旋转了90度。所以要旋转回来。

注意：OpenCV与Matlab不同的是，OpenCV要先设置旋转的中心点，通常是图片的中间。



|  |
| --- |
| //先设置中心点  cv::Mat M = cv::getRotationMatrix2D(Point2f(lastImg.cols/2,lastImg.rows/2),270,1);  //再用warpAffine旋转  cv::warpAffine(lastImg,rotateImg,M,rotateImg.size()); |



s1-03，转灰度图

为什么要先用高斯模糊？

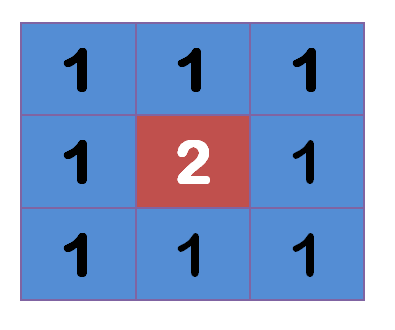
当时受到EasyPR一文的影响，作者提到基于色彩的高斯模糊过程比灰度后的高斯模糊过程更容易检测到边缘点。不过我没有ofo使用高斯模糊前后的对比。

什么是高斯模糊？（节选自http://www.ruanyifeng.com/blog/2012/11/gaussian\_blur.html）

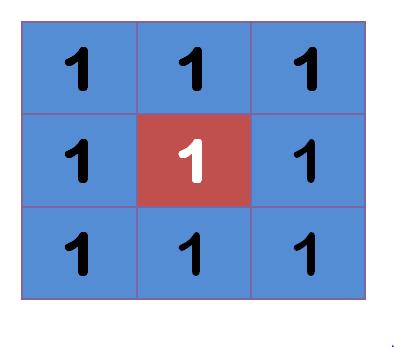
"模糊"，使图片产生模糊的效果。



所谓"模糊"，可以理解成每一个像素都取周边像素的平均值。

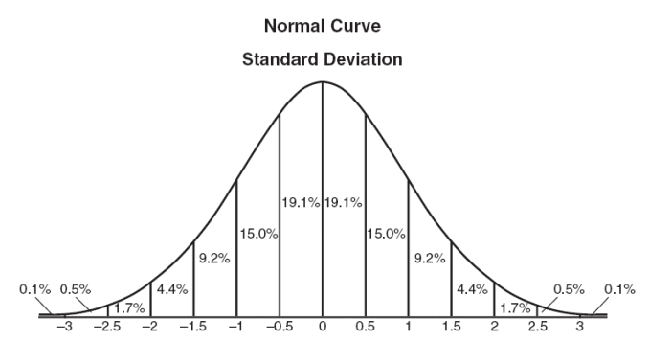


上图中，2是中间点，周边点都是1。



"中间点"取"周围点"的平均值，就会变成1。在数值上，这是一种"平滑化"。在图形上，就相当于产生"模糊"效果，"中间点"失去细节。

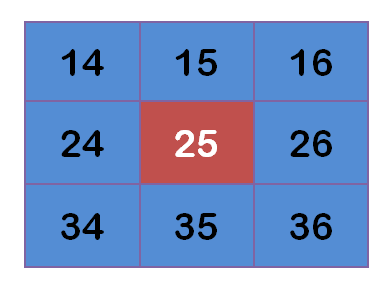
"高斯模糊"利用正态分布（又名"高斯分布"）来进行图像的模糊处理



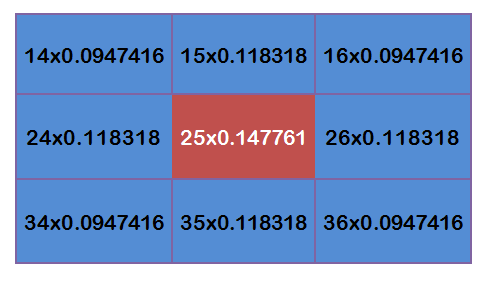
在图形上，正态分布是一种钟形曲线，越接近中心，取值越大，越远离中心，取值越小。

计算平均值的时候，我们只需要将"中心点"作为原点，其他点按照其在正态曲线上的位置，分配权重，就可以得到一个加权平均值。

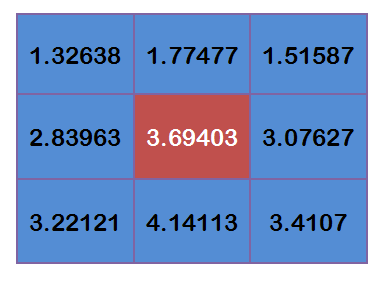
假设现有9个像素点，灰度值（0-255）如下：



每个点乘以自己的权重值：



得到



将这9个值加起来，就是中心点的高斯模糊的值。

对所有点重复这个过程，就得到了高斯模糊后的图像。如果原图是彩色图片，可以对RGB三个通道分别做高斯模糊。

为什么要转灰度图？

因为要取边缘

|  |
| --- |
| cv::GaussianBlur(lastImg,grayImg,Size(11,11),0,0); //高斯  cv::cvtColor(grayImg,grayImg,COLOR\_RGB2GRAY); |

****

s1-04，转二值图

为什么边缘化时要用Laplacian算子而不像EasyPR中使用Sobel算子？

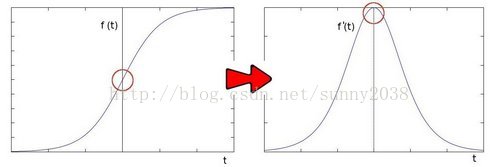
实际测试ofo车牌时Laplacian的效果远好于Sobel。并且OpenCV的Sobel算子用起来比Matlab麻烦得多。但是Laplacian算子对噪声比较敏感，所以后面要做降噪。

Laplacian与Sobel有什么区别？(节选自 http://blog.csdn.net/garfielder007/article/details/51326218 )

Laplacian算子是求图像的二阶导数，Sobel算子求图像的一阶导数。

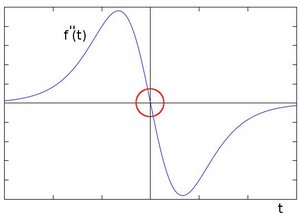
Sobel算子的原理：

图像中的边缘区域，像素值会发生“跳跃”，对这些像素求导，在其一阶导数在边缘位置为极值，这就是Sobel算子使用的原理——极值处就是边缘



Laplacian实现的方法是先用Sobel 算子计算二阶x和y导数，再求和

如果对像素值求二阶导数，会发现边缘处的导数值为0。如下图



为什么要转二值图？

因为只有黑与白，为下面外接矩形做准备。而Laplacion出来的依然是灰度图

为什么要去燥

去掉一些小点，避免对后面的操作有干扰

|  |
| --- |
| cv::Laplacian(lastImg,bwImg,-1,3); //边缘化cv::threshold(bwImg,bwImg,0,255,CV\_THRESH\_OTSU+CV\_THRESH\_BINARY);  cv::fastNlMeansDenoising(bwImg,bwImg,30,15,31); //去燥 |

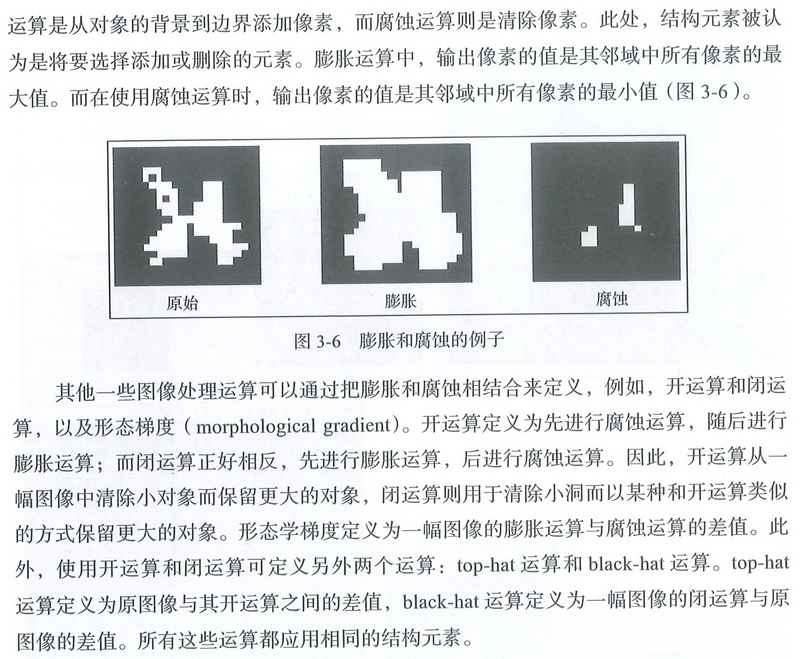


s1-05，闭操作

开运算：先腐蚀，再膨胀。用于去除较小的明亮区域。

闭运算：先膨胀，再腐蚀。用于消除低亮度值的孤立点。

膨胀和腐蚀的示意图：(来自《OpenCV图像处理》)

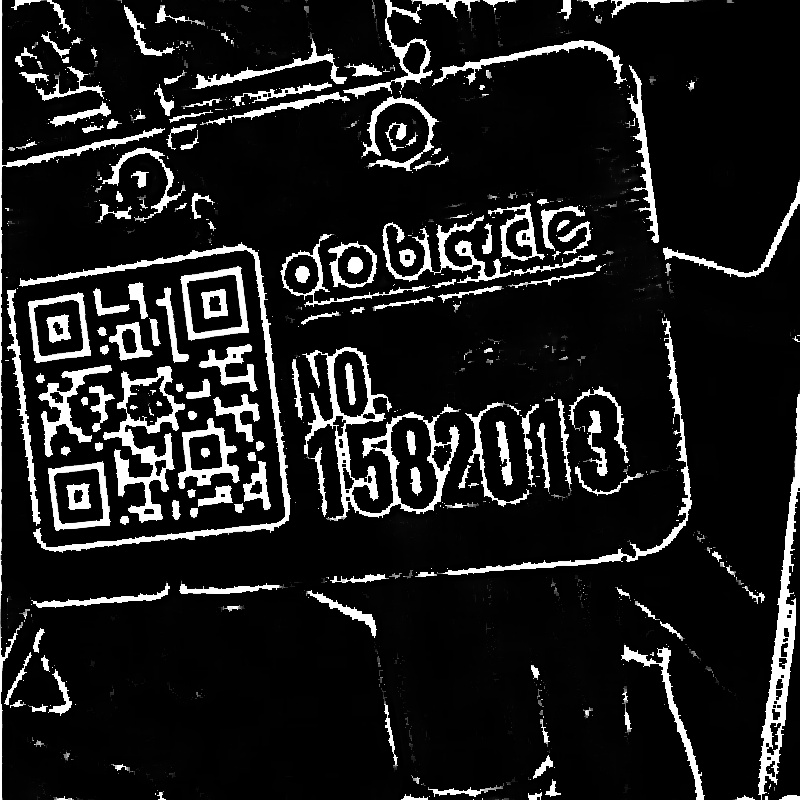


这一步闭操作主要的目的是，用一个小方块进行填补字符、二维码的裂缝，以免下一步取最小外接矩形时失败。(其实这一步在6位ofo中最明显，而7位中二维码的裂缝问题好像没有遇到)



|  |
| --- |
| Mat element1 = cv::getStructuringElement(MORPH\_RECT,Size(3,3));  cv::dilate(lastImg,dilateImg1,element1);  cv::erode(dilateImg1,dilateImg1,element1); |

闭操作后，图像明显



s1-06最小外接矩形

终于来到最激动人心的一步，对所有连在一起的点，做一个外接垂直矩形。

|  |
| --- |
| vector< vector< Point> > contoursQRFind; cv::findContours(rectQRFindImg,contoursQRFind,CV\_RETR\_EXTERNAL,CV\_CHAIN\_APPROX\_NONE);  for (int i=0;i<contoursQRFind.size();i++)  {  //绘制最小外接矩形  cv::Rect rectQRFind=cv::boundingRect(contoursQRFind[i]);  cv::rectangle(rectQRFindImg,rectQRFind,Scalar(255,255,255));  } |

当然做外接矩形时，顺便判断是否为二维码(长宽比为1:1，但是因为人拍照时会拍弯，所以相应的比例也要放松)

|  |
| --- |
| if(  ((float(rectQRFind.width)/rectQRFind.height)>0.7)&&  ((float(rectQRFind.width)/rectQRFind.height)<1.3)&&  ((rectQRFind.width)>200)&&  ((rectQRFind.width)<400)&&  ((rectQRFind.x>10))&&  ((rectQRFind.x<200))&&  ((rectQRFind.y>50))&&  ((rectQRFind.y<400))&&  ((contoursQRFind[i].size())>800)  ) |

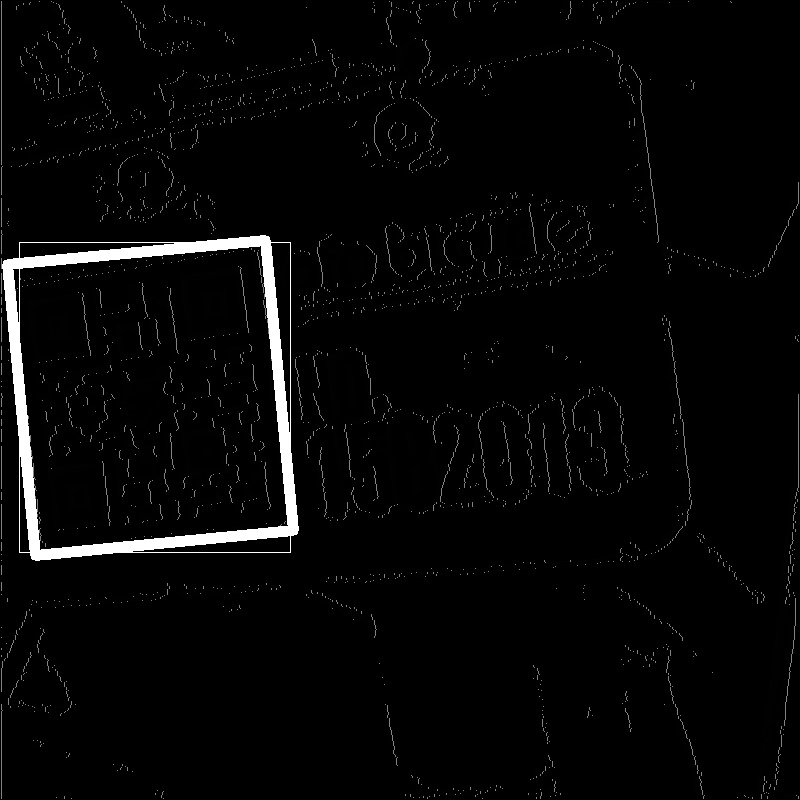
而确定了是二维码后，再找这个二维码的外接矩形（注意与外接垂直矩形的区别，见下图），以便下一步用来做旋转



|  |
| --- |
| cv::RotatedRect rectQRFindRotate=cv::minAreaRect(contoursQRFind[i]);  rectQRFindRotate.points(QRFindRotatePoint);  for (int j=0; j<=3; j++)  {  line(rectQRFindImg,QRFindRotatePoint[j],QRFindRotatePoint[(j+1)%4],Scalar(255,255,255),10);  // cout<<"rotate:"<<QRFindRotatePoint[j]<<endl;  } |

由于整个图片二维码是唯一，如果超过，或者找不到，就直接中止处理

|  |
| --- |
| if(QRFinded!=1)  {  cout<<"ERROR: Can't Find correct QR"<<endl;  continue;  } |



s1-08利用QR旋转

由于上面找到的向量集合中，4个角点坐标不一定是按左上、右上那样排。所以要先将左上角和右上角的点找出来。

|  |
| --- |
| //需要补一个寻找左上角和右上角点。想到的思路是x+y最小的为左上角，而接下来的是右上角  int findLeft,findLeftXY=800\*800;  for (int j=0; j<=3; j++)  {  if ((QRFindRotatePoint[j].x+QRFindRotatePoint[j].y)<findLeftXY)  {  findLeftXY=QRFindRotatePoint[j].x+QRFindRotatePoint[j].y;  findLeft=j;  }  } |

找到左上角的点后，就用arctan求旋转角度。之后就可以轻微旋转

|  |
| --- |
| // 用arctan来求旋转角度，公式(y/x)\*180/3.14  angelOfo = atan ((QRFindRotatePoint[findLeft+1].y-QRFindRotatePoint[findLeft].y)/(QRFindRotatePoint[findLeft+1].x-QRFindRotatePoint[findLeft].x))\*180/3.14;  //先设置中心点  cv::Mat M2 = cv::getRotationMatrix2D(Point2f(rotateOfoImg.cols/2,rotateOfoImg.rows/2),angelOfo,1);  //再用warpAffine旋转  //参数 源，目标，角度，尺寸  cv::warpAffine(rotateOfoImg,rotateOfoImg,M2,rotateOfoImg.size()); |



s1-09利用二维码的长、宽，粗略将车牌抠出来

先用二点间距离公式计算二维码的长、宽

|  |
| --- |
| //二点间距离公式  // sqrt(pow((x2-x1),2)+pow((y2-y1),2))  float distanceQRx= sqrt(pow((QRFindRotatePoint[findLeft+1].x-QRFindRotatePoint[findLeft].x),2) +pow((QRFindRotatePoint[findLeft+1].y-QRFindRotatePoint[findLeft].y),2));  float distanceQRy= sqrt(pow((QRFindRotatePoint[findLeft].x-QRFindRotatePoint[findLeft-1].x),2) +pow((QRFindRotatePoint[findLeft].y-QRFindRotatePoint[findLeft-1].y),2)); |

万分注意，抠出来之前，要做个判断，不要超过图像的x右界，否则会crash

|  |
| --- |
| int maskOfo7CharX=distanceQRx\*1.4;  if((maskOfo7CharX+QRx+distanceQRx)>800)  {  maskOfo7CharX=800-(QRx+distanceQRx);  } |

接下来就按照x初始点是QR的右边界，y初始点为QR的0.6位置，将QR长的1.4倍，QR宽的0.5倍的位置抠出来

|  |
| --- |
| cv::Rect maskOfo7Char(QRx+distanceQRx, QRy+(distanceQRy\*0.6), maskOfo7CharX, distanceQRy\*0.5);  cv::Mat croppedOfo7CharImg(cropOfo7CharImg, maskOfo7Char); |



第一阶段就到此结束。第二阶段的步骤与第一阶段比较类似。原理就不再细述

canny

http://www.cnblogs.com/tiandsp/archive/2013/04/20/3032860.html

老师说一天搞定，我觉得除非对计算机图形学的知识十分熟悉，否则挺难。前前后后坑了一个月才弄出来。

最大的难点是调好了一张图，而放到一批图的时候又会有概率问题