Android 笔记

简介

名称：Android 笔记
作者：zhaoqp

简介：Android 完全学习笔记
资料的来源包括书籍，网络，力求准确，全面，实用，经过我的部分修改和调整，一定会对你有帮助，由于个人能力有限，如有问题敬请批评，欢迎大家一起纠正错误，谢谢。

支持论坛：http：//immortal.5d6d.com

（Android 学习园地，打造资料最全面的学习社区）

欢迎大家共同学习，新的信息和大量的资料请进**论坛下载**。
QQ：396196516
Email：zhaoqp2010@gmail.com

章节会陆续更新。。。。。
其他章节正在修正中，敬请关注。
大家如果有好的资料可以联系我，一起把这本笔记做好，谢谢。

Android 笔记 1
简介 1
第七章 数学及算法 2
点在直线上 2
点在线段上 2
两条直线相交 3
两条线段相交 3
线段直线相交 4
矩形包含点 5
线段，折线，多边形在矩形中 5
矩形在矩形中 5
圆在矩形中 6
深度优先算法 6
广度优先算法 13
广度优先 A*算法 15
Dijkstra 算法 17
Dijkstra A*算法 21
有限状态机 23
有限状态机的 OO 实现 24
冒泡排序法 24
选择排序法 25

插入排序法.....25
快速排序法.....25

第七章 数学及算法

这一章将介绍下开发中常用的几何公式，公式都是大家学过的，只是写成了方法,以及常用的算法。
大家如果喜欢也可以自己实现一些复杂的放到论坛上与大家分享。

点在直线上

直线方程两点式： $\frac{y-y_1}{y_2-y_1}=\frac{x-x_1}{x_2-x_1}$（$P_1(x_1,y_1)$、$P_2(x_2,y_2)$ $x_1\neq x_2$， $y_1\neq y_2$）.

```
点 A（x，y）,B(x1,y1),C(x2,y2)    点 A 在直线 BC 上吗?
public  boolean pointAtSLine
           (double x,double y,double x1,double y1,double x2,double y2){
       int result = ( x - x1 ) * ( y2 - y1 ) - ( y - y1 ) * ( x2 - x1 );
        if(result==0){
           //System.out.println("点在直线上");
           return true;
        }else{
           //System.out.println("点不在直线上");
           return false;
        }
    }
```

点在线段上

```
点 A（x，y）,B(x1,y1),C(x2,y2)    点 A 在线段 BC 上吗?
public static boolean pointAtELine
     (double x,double y,double x1,double y1,double x2,double y2){
       int result = ( x - x1 ) * ( y2 - y1 ) - ( y - y1 ) * ( x2 - x1 );
        if(result==0){
           //System.out.println("点（"+x+","+y+"）在直线上");
           if(x >= Math.min(x1, x2) && x <= Math.max(x1,x2)
               && y >= Math.min(y1, y2) && y <= Math.max(y1,y2)){
              //System.out.println("点（"+x+","+y+"）在线段上");
               return true;
           }else{
```

```
                System.out.println("点（"+x+","+y+"）不在线段上");
                return false;
            }
        }else{
            System.out.println("点（"+x+","+y+"）在不在直线上");
            return false;
        }
    }
```

两条直线相交

斜率公式： $k=\frac{y_2-y_1}{x_2-x_1}$，其中 $P_1(x_1,y_1)$、$P_2(x_2,y_2)$.

```
点A（x1，y1）,B(x2,y2),C(x3,y3),D(x4,y4)    直线AB与直线CD相交吗?
public  boolean LineAtLine(double x1,double y1,double x2,
                double y2,double x3,double y3,double x4,double y4){
        double k1 = ( y2-y1 )/(x2-x1);
        double k2 = ( y4-y3 )/(x4-x3);
        if(k1==k2){
            //System.out.println("平行线");
            return false;
        }else{
          double x = ((x1*y2-y1*x2)*(x3-x4)-(x3*y4-y3*x4)*(x1-x2))
                        /((y2-y1)*(x3-x4)-(y4-y3)*(x1-x2));
          double y = ( x1*y2-y1*x2 - x*(y2-y1) ) / (x1-x2);
          //System.out.println("直线的交点("+x+","+y+")");
          return true;
        }
    }
```

两条线段相交

```
点A（x1，y1）,B(x2,y2),C(x3,y3),D(x4,y4)    线段AB与线段CD相交吗?
public boolean ELineAtELine(double x1,double y1,double x2,
                    double y2,double x3,double y3,double x4,double y4){
        double k1 = ( y2-y1 )/(x2-x1);
        double k2 = ( y4-y3 )/(x4-x3);
        if(k1==k2){
            System.out.println("平行线");
            return false;
        }else{
```

```
            double x = ((x1*y2-y1*x2)*(x3-x4)-(x3*y4-y3*x4)*(x1-x2))
                        /((y2-y1)*(x3-x4)-(y4-y3)*(x1-x2));
            double y = ( x1*y2-y1*x2 - x*(y2-y1) ) / (x1-x2);
            System.out.println("直线的交点("+x+","+y+")");
            if(x >= Math.min(x1, x2) && x <= Math.max(x1,x2)
                    && y >= Math.min(y1, y2) && y <= Math.max(y1,y2)
                    && x >= Math.min(x3, x4) && x <= Math.max(x3,x4)
                    && y >= Math.min(y3, y4) && y <= Math.max(y3,y4) ){
                //System.out.println("交点（"+x+","+y+"）在线段上");
                return true;
            }else{
                System.out.println("交点（"+x+","+y+"）不在线段上");
                return false;
            }
        }
}
```

线段直线相交

点 A（x1，y1）,B(x2,y2),C(x3,y3),D(x4,y4)　线段 AB 与直线 CD 相交吗?

```
public boolean ELineAtELine(double x1,double y1,double x2,
                    double y2,double x3,double y3,double x4,double y4){
        double k1 = ( y2-y1 )/(x2-x1);
        double k2 = ( y4-y3 )/(x4-x3);
        if(k1==k2){
            System.out.println("平行线");
            return false;
        }else{
            double x = ((x1*y2-y1*x2)*(x3-x4)-(x3*y4-y3*x4)*(x1-x2))
                        /((y2-y1)*(x3-x4)-(y4-y3)*(x1-x2));
            double y = ( x1*y2-y1*x2 - x*(y2-y1) ) / (x1-x2);
            System.out.println("交点("+x+","+y+")");
            if(x >= Math.min(x1, x2) && x <= Math.max(x1,x2)
                    && y >= Math.min(y1, y2) && y <= Math.max(y1,y2)){
                //System.out.println("交点（"+x+","+y+"）在线段上");
                return true;
            }else{
                System.out.println("交点（"+x+","+y+"）不在线段上");
                return false;
            }
        }
}
```

矩形包含点

矩形的边都是与坐标系平行或垂直的。
只要判断该点的横坐标和纵坐标是否夹在矩形的左右边和上下边之间。
点 A（x，y），B(x1,y1),C(x2,y2)　点 A 在以直线 BC 为对角线的矩形中吗?

```
public boolean pointAtRect(
        double x,double y,double x1,double y1,double x2,double y2){
        if(x >= Math.min(x1, x2) && x <= Math.max(x1,x2)
                && y >= Math.min(y1, y2) && y <= Math.max(y1,y2)){
           //System.out.println("点（"+x+","+y+"）在矩形内上");
           return true;
        }else{
          //System.out.println("点（"+x+","+y+"）不在矩形内上");
        return false;
        }
    }
```

线段，折线，多边形在矩形中

因为矩形是个凸集，所以只要判断所有端点是否都在矩形中就可以了。

矩形在矩形中

只要对角线的两点都在另一个矩形中就可以了。
点 A(x1,y1),B(x2,y2)，C(x1,y1),D(x2,y2) 以直线 AB 为对角线的矩形在以直线 BC 为对角线的矩形中吗?

```
public boolean RectAtRect(double x1,double y1,double x2,double y2,
                    double x3,double y3,double x4,double y4){
        if(x1 >= Math.min(x3, x4) && x1 <= Math.max(x3,x4)
                && y1 >= Math.min(y3, y4) && y1 <= Math.max(y3,y4)
                && x2 >= Math.min(x3, x4) && x2 <= Math.max(x3,x4)
                && y2 >= Math.min(y3, y4) && y2 <= Math.max(y3,y4)){
           //System.out.println("矩形在矩形内");
           return true;
        }else{
           //System.out.println("矩形不在矩形内");
           return false;
        }
    }
```

圆在矩形中

圆心在矩形中且圆的半径小于等于圆心到矩形四边的距离的最小值。

圆心(x,y) 半径 r 矩形对角点 A（x1，y1)，B(x2，y2)

```
public boolean circleAtRect(double x,double y,double r,double x1,
                    double y1,double x2,double y2){
        //圆心在矩形内
        if(x >= Math.min(x1, x2) && x <= Math.max(x1,x2)
                  && y >= Math.min(y1, y2) && y <= Math.max(y1,y2)){
        //圆心到4条边的距离
        double l1= Math.abs(x-x1);
        double l2= Math.abs(y-y2);
        double l3= Math.abs(x-x2);
        double l4= Math.abs(y-y2);
        if(r<=l1 && r<=l2 && r<=l3 && r<=l4){
              //System.out.println("圆在矩形内");
              return true;
          }else{
              //System.out.println("圆不在矩形内");
              return false;
          }

          }else{
             //System.out.println("圆不在矩形内");
            return false;
          }
    }
```

深度优先算法

目的：产生两点之间的路线，从一个节点开始，然后尽可能的沿着一条边深入，当遇到死胡同时进行回溯，然后继续搜索，直到找到目标为止。

思路:从出发点向8个方向延伸出线,再从延伸线最后一个方向点继续向8个方向延伸出线，特点总是 8 个中取一个分支延伸到底，再开始另外的分支，直到找到目标点。并不是最短路线。

我们的出发点和目标点：

```
static int[] source={2,2};      //出发点坐标
static int[] target= {13,20};  //目标点坐标
```

我们先类似这样的先定义一个栈：

```
Stack<int[][]> stack=new Stack<int[][]>();
```

栈里记录的是所有可到达的线段，而 int 数组的数据代表的两个点。

寻路的原理：首先我们在起点的位置，然后到 8 个方向探测点，然后以探测到的点为始点再向 8 个方向探测，直到找到为止。

用栈的原因：我们需要从栈里取出顶的数据做为始点寻找 8 个方向的所有点然后把这个始点删除，在取刚取的的 8 个方向的点做为始点寻找。

地图的数组：
每个表示的地图的一个图元，图元的值如果为 1 说明不能通过， 0 可以通过
这个是 3 维的数组：
第一维用来表示哪张图，现在这里只有一张图
第二维用来表示这张图的列
第三维用来表示这张图的行
如 ： map[0][0][0]==1,表示第一张图的第一行第一列，不能通过。
map[0][1][1]==0, 表示第一张图的第二行第二列，可以通过。

定义如下：

```
static int[][][] map = new int[][][]{
        {
            {1,1,1,1,1,1,1,1,0,0,0,0,0,0,0,0,0,0,0,0,0},
            {1,0,0,0,0,0,1,1,0,0,1,1,0,0,0,0,0,0,0,0,0},
            {1,0,0,0,0,0,1,1,0,0,1,1,0,0,0,0,1,1,0,0,0},
            {1,0,0,0,0,0,1,1,1,1,1,1,0,0,0,0,1,1,0,0,0},
            {1,0,0,0,0,0,0,0,0,0,1,1,0,0,0,1,1,1,1,0,0},
            {1,0,0,0,0,0,1,1,1,0,1,1,0,0,0,0,1,1,0,0,0},
            {1,0,0,0,0,0,1,1,1,0,1,1,0,0,0,0,1,1,0,0,0},
            {1,0,0,0,0,0,1,1,1,0,1,1,0,0,0,0,0,0,0,0,0},
            {1,1,1,0,0,1,1,1,1,0,1,1,0,0,0,0,0,0,0,0,0},
            {1,1,1,0,0,1,1,1,1,0,1,1,0,0,0,0,0,1,1,0,0},
            {0,0,0,0,0,0,0,0,0,0,1,1,0,0,0,0,0,1,1,0,0},
            {0,0,1,0,0,1,1,1,1,1,1,1,0,0,0,0,1,1,1,1,0},
            {1,1,1,0,0,1,1,1,1,1,1,1,1,1,0,0,1,1,1,1,0},
            {1,1,1,0,0,1,1,1,1,1,1,1,1,1,0,0,0,1,1,0,0},
            {1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0},
            {1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0},
            {1,1,1,0,0,1,1,1,1,1,1,1,0,0,1,1,1,1,1,0,0},
            {0,0,1,0,0,0,0,0,0,0,0,0,0,0,1,1,1,1,0,0,0},
            {0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1,1,1,0,0,0},
            {0,0,0,0,0,1,1,1,1,1,1,0,0,0,1,1,1,1,0,0,0},
            {1,1,1,0,0,0,0,0,1,1,1,0,0,0,0,0,0,0,0,0,0},
            {1,1,1,0,0,0,0,0,1,1,1,0,0,0,0,0,0,0,0,0,0},
            {1,1,1,0,0,0,0,0,1,1,1,0,0,0,0,0,0,0,0,0,0}
        }
```

```
    };
```

那我们有了起点和终点，也知道地图的边界了，那我们怎么寻路呢？

我们检测始点周围的 8 个方格是否有目标点 同时检查是否是边界或不能通过的区域。

定义：

```
//周围的8个方格的所有组合sequence
```

可见顺序是 ：上 下 左 右 左上 左下 右下 右上 逆时针方向检测，这个值的设置会决定路线的方向。你也可以更改下看下效果，合适的方向能更快的找到目标。

```
int[][] sequence={
        {0,1},{0,-1},
        {-1,0},{1,0},
        {-1,1},{-1,-1},
        {1,-1},{1,1}
};
```

下面我们只要把所有可能的寻路的线段都放入栈中，然后循环绘制线段就可以达到我们所要的结果了。

然后我们看下这个方法的实现：

```
ArrayList<int[][]> searchProcess=new ArrayList<int[][]>();//搜索过程
HashMap<String,int[][]> hm=new HashMap<String,int[][]>();//结果路径记录
 //0 未去过 1 去过
int[][] visited =
          new int[MapList.map[0].length][MapList.map[0][0].length];
public void DFS(){
    new Thread(){
        public void run(){
            boolean flag=true;
            //开始点表示为一条起点和终点重合的直线 目的是我们只有知道起点和终点才能画出寻路的线段
            int[][] start={
                    {source[0],source[1]},
                    {source[0],source[1]}
            };
            //把起点放入栈
            stack.push(start);
            int count=0;//步数计数器
                while(flag){
                    //从栈顶取出边取出后就被移除了
                    int[][] currentEdge=stack.pop();
                    int[] tempTarget=currentEdge[1];//取出此边的目的点
                    //判断目的点是否去过，若去过则直接进入下次循环
                    if(visited[tempTarget[1]][tempTarget[0]]==1){
                        continue;
                    }
                    count++;
```

```
            //标识目的点为访问过
            visited[tempTarget[1]][tempTarget[0]]=1;
            //将临时目的点加入搜索过程中
            searchProcess.add(currentEdge);
            //记录此临时目的点的父节点
            hm.put(tempTarget[0]+":"+tempTarget[1],new
             int[][]{currentEdge[1],currentEdge[0]});
            gameView.postInvalidate();
            try{
                Thread.sleep(timeSpan);
                }catch(Exception e){
                 e.printStackTrace();
            }
         //判断有否找到目的点
         if(tempTarget[0]==target[0]
                  &&tempTarget[1]==target[1]){
                  break;
          }
        //将所有可能的边入栈
        int currCol=tempTarget[0];
        int currRow=tempTarget[1];
            //周围的8个方格所有组合sequence
            for(int[] rc:sequence){
                int i=rc[1];
                int j=rc[0];
                //0，0 等于没动
                if(i==0&&j==0){continue;}
                    //边界和不通过的检测
                    if(currRow+i>=0
                        && currRow+i<MapList.map[mapId].length
                        && currCol+j>=0
                        &&currCol+j<MapList.map[mapId][0].length
                            //!=1 说明可以通过
                        && map[currRow+i][currCol+j]!=1){
                            int[][] tempEdge={
                                {tempTarget[0],tempTarget[1]},
                                {currCol+j,currRow+i}
                            };
                            stack.push(tempEdge);
                    }
                }
        }//循环结束
        pathFlag=true;  //标记可以画出结果路线了
        gameView.postInvalidate();
```

```
            }
        }.start();
    }
```

下面画出寻找的路线：

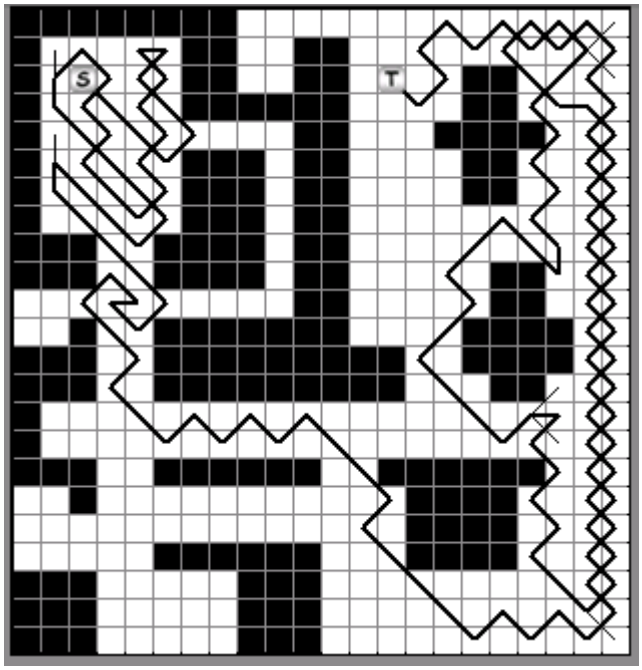
```
    int span = 13;  //每个方格的宽高这里是正方格
    for(int k=0;k<searchProcess.size();k++){//绘制寻找过程
            int[][] edge=searchProcess.get(k);
            paint.setColor(Color.BLACK);
            paint.setStrokeWidth(1);
            canvas.drawLine(
        edge[0][0]*(span+1)+span/2+6,edge[0][1]*(span+1)+span/2+6,
        edge[1][0]*(span+1)+span/2+6,edge[1][1]*(span+1)+span/2+6,
                paint
            );
        }
```

画出结果路线：和寻路的路线是一样的 但这个是反向画的，先从目标点开始，在 Map 中寻找能到这个目标点的线段，然后已那个起点为目标点继续查找，加粗显示这些线段。这个会永远只有一条，在 map 中的 key 表示的终点只有一个，也就是说到达某一点的线是唯一的。

```
if(pathFlag){


                int count=0;//路径长度计数器
                while(true){
                    //目标点的坐标
                   int[] temp=game.target;
                    int[][] tempA=hm.get(temp[0]+":"+ temp[1]);
                    paint.setColor(Color.BLACK);
                    paint.setStyle(Style.STROKE);//加粗
                    paint.setStrokeWidth(2);      //设置画笔粗度为2px
                    canvas.drawLine(
        tempA[0][0]*(span+1)+span/2+6,tempA[0][1]*(span+1)+span/2+6,
        tempA[1][0]*(span+1)+span/2+6,tempA[1][1]*(span+1)+span/2+6,
                        paint
                    );
                    count++;
                //判断有否到出发点 因为是从目标点开始的 到出发点就可以停了
                if(tempA[1][0]==source[0]
                                        &&tempA[1][1]==source[1]){
                        break;
                }
                temp=tempA[1];   //以这个点作为目标点继续反向查找
        }
```

如图：

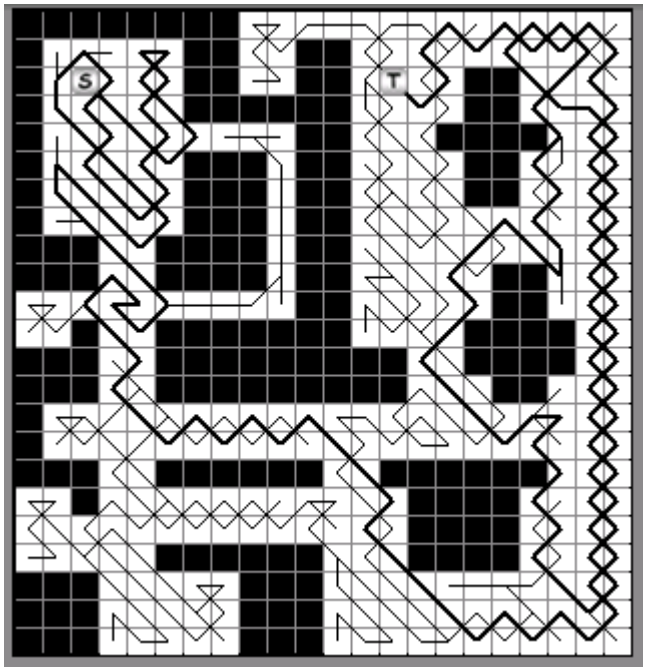



```
(2,2)
push;2,2,2,3
push;2,2,2,1
push;2,2,1,2
push;2,2,3,2
push;2,2,1,3
push;2,2,1,1
push;2,2,3,1
push;2,2,3,3
************************
(3,3)
push;3,3,3,4
push;3,3,3,2
push;3,3,2,3
push;3,3,4,3
push;3,3,2,4
push;3,3,2,2
push;3,3,4,2
push;3,3,4,4
************************
(4,4)
push;4,4,4,5
push;4,4,4,3
push;4,4,3,4
push;4,4,5,4
push;4,4,3,5
push;4,4,3,3
push;4,4,5,3
push;4,4,5,5
************************
```

比较下路线发现：

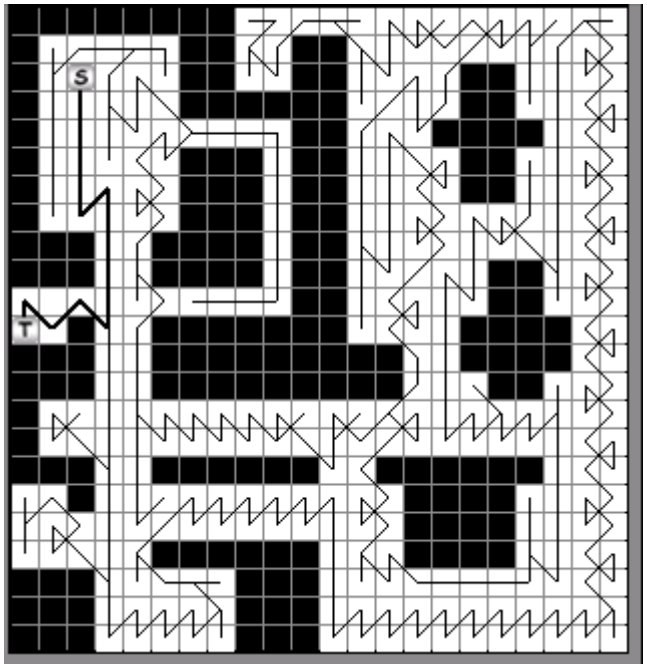
首先以（2，2）为起点，向 8 个方向找到了 8 个点都可以到达，压入栈中后，再取位于栈顶

的点（3，3）再去寻找，然后压入栈，依次类推，最后地图上所有的可到达的点都入栈中了，同时控制了栈中的点不重复，找到目标点就停止了，所以看到的会有空白，如果让他继续的话结果会是这样的：



可见是把所有的点都跑了个遍，大家看到了返回的只有一条路线，原以为会有多路线的，是什么原因呢?
原因在于：如果方格访问过，就会忽略这个方格，导致了我们不能产生多条路线。

下面这个图是为了说明：取得的条路线并不是一下就取得的 要看目标的位置，下图中的就是查找了大部分地图才找到的，效率很低。



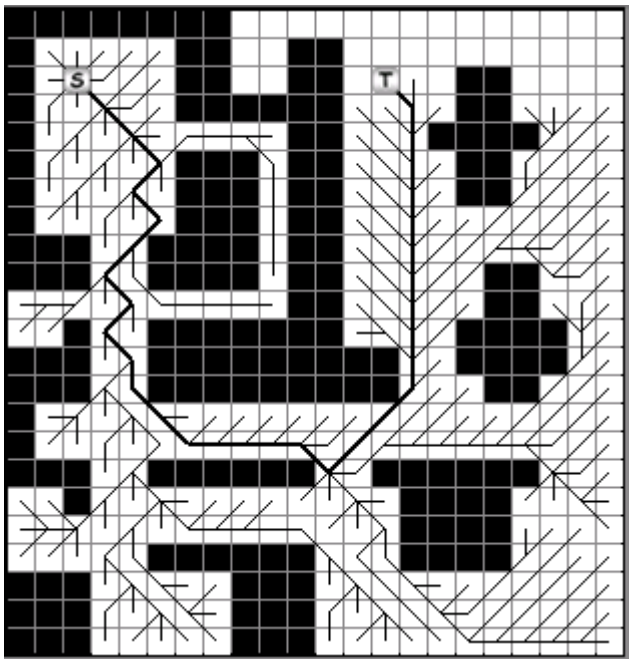
广度优先算法

目的是：产生两点之间的路线，从一个节点开始，先将所有连接到该节点的节点访问到，然后再继续访问下一层，直到找到目标为止。
思路：从出发点向 8 个方向延伸出线，找到点后从每个点都继续延伸出线，直到找到目标点。并不是最短路线，给人感觉是从出发点点开始像撒网一样，附近的点一个不漏，才向下延伸。和深度优先的区别是：广度优先算法采用的 queue 队列 而不是栈。实现基本相同。

```
LinkedList<int[][]> queue=new LinkedList<int[][]>();//广度优先所用队列
public void BFS(){//广度优先算法
        new Thread(){
            public void run(){
                int count=0;//步数计数器
                boolean flag=true;
                int[][] start={//开始状态
                    {source[0],source[1]},
                    {source[0],source[1]}
                };
                queue.offer(start);
                while(flag){
                    int[][] currentEdge=queue.poll();//从队首取出边
                    int[] tempTarget=currentEdge[1];//取出此边的目的点
                    //判断目的点是否去过，若去过则直接进入下次循环
                    if(visited[tempTarget[1]][tempTarget[0]]==1){
                        continue;
                    }
                    count++;
                    //标识目的点为访问过
                    visited[tempTarget[1]][tempTarget[0]]=1;
                    //将临时目的点加入搜索过程中
                    searchProcess.add(currentEdge);
                    //记录此临时目的点的父节点
                    hm.put(tempTarget[0]+":"+tempTarget[1],
                            new int[][]{currentEdge[1],currentEdge[0]});
                    gameView.postInvalidate();
                    try{
                        Thread.sleep(timeSpan);
                    }catch(Exception e){
                       e.printStackTrace();
                    }
                //判断有否找到目的点
                if(tempTarget[0]==target[0]&&tempTarget[1]==target[1]){
                        break;
                    }
```

```
                //将所有可能的边入队列
                int currCol=tempTarget[0];
                int currRow=tempTarget[1];
                for(int[] rc:sequence){
                    int i=rc[1];
                    int j=rc[0];
                    if(i==0&&j==0){continue;}
                       if(currRow+i>=0
                        && currRow+i<MapList.map[mapId].length
                        && currCol+j>=0
                        && currCol+j<MapList.map[mapId][0].length
                        && map[currRow+i][currCol+j]!=1){
                            int[][] tempEdge={
                                {tempTarget[0],tempTarget[1]},
                                {currCol+j,currRow+i}
                            };
                            queue.offer(tempEdge);
                        }
                    }
                }
                pathFlag=true;
                gameView.postInvalidate();
            }
        }.start();
    }
```

如图：



广度优先 A*算法

A*算法是一种启发式搜索，就是利用一个启发因子评估每次的寻找的路线的优劣，再决定往哪个节点走。实际上，A*只是一种思想，我们可以运用这种思想对之前实现的搜索算法进行优化。
这个的效率比较高，基本上是最优路径但不能完全保证，游戏中应用较多。和广度优先的区别：
用优先级队列比较了队列中线段的距离， 近的优先， 所以效率高了，路线也变得短了。

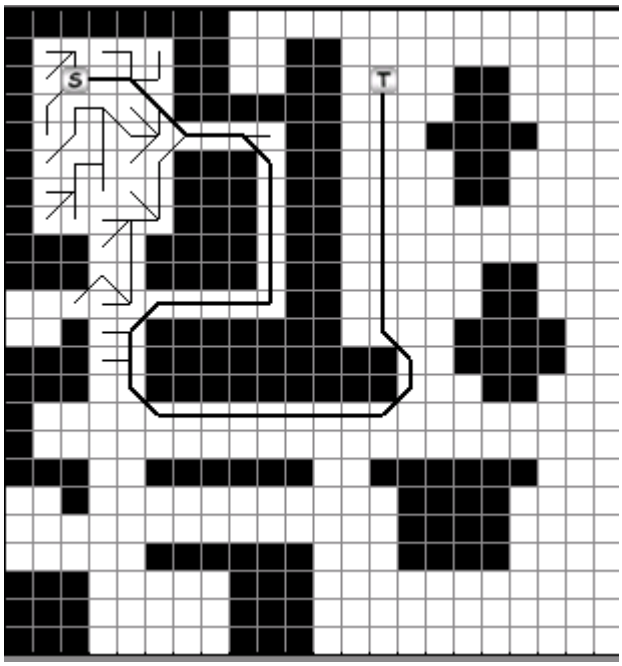
```
//A*用优先级队列
PriorityQueue<int[][]> astarQueue=
     new PriorityQueue<int[][]>(100,new AStarComparator(this));
//线段的距离比较
public class AStarComparator implements Comparator<int[][]>{
    Game game;
    public AStarComparator(Game game){
        this.game=game;
    }
    public int compare(int[][] o1,int[][] o2){
        int[] t1=o1[1];
        int[] t2=o2[1];
        int[] target=game.target;
        //直线物理距离
        int a=(t1[0]-target[0])*(t1[0]-target[0])
              +(t1[1]-target[1])*(t1[1]-target[1]);
        int b=(t2[0]-target[0])*(t2[0]-target[0])
        +(t2[1]-target[1])*(t2[1]-target[1]);
        //门特卡罗距离
        //int a=game.visited[o2[0][1]][o2[0][0]]
            +Math.abs(t1[0]-target[0])+Math.abs(t1[1]-target[1]);
        //int b=game.visited[o2[0][1]][o2[0][0]]
            +Math.abs(t2[0]-target[0])+Math.abs(t2[1]-target[1]);
        return a-b;
    }
    public boolean equals(Object obj){
        return false;
    }
    }

 public void BFSAStar(){//广度优先 A*算法
        new Thread(){
            public void run(){
                boolean flag=true;
                int[][] start={//开始状态
                    {source[0],source[1]},
```

```
        {source[0],source[1]}
    };
    astarQueue.offer(start);
    int count=0;
    while(flag){
        int[][] currentEdge=astarQueue.poll();//从队首取出边
        int[] tempTarget=currentEdge[1];//取出此边的目的点
        //判断目的点是否去过，若去过则直接进入下次循环
        if(visited[tempTarget[1]][tempTarget[0]]!=0){
            continue;
        }
        count++;
    //标识目的点为访问过
    visited[tempTarget[1]][tempTarget[0]]=
         visited[currentEdge[0][1]][currentEdge[0][0]]+1;
       //将临时目的点加入搜索过程中
       searchProcess.add(currentEdge);
        //记录此临时目的点的父节点
        hm.put(tempTarget[0]+":"+tempTarget[1],
             new int[][]{currentEdge[1],currentEdge[0]});
        gameView.postInvalidate();
        try{Thread.sleep(timeSpan);
        }catch(Exception e){
           e.printStackTrace();}
        //判断有否找到目的点
        if(tempTarget[0]==target[0]
            &&tempTarget[1]==target[1]){
            break;
        }
        int currCol=tempTarget[0];//将所有可能的边入优先级队列
        int currRow=tempTarget[1];
        for(int[] rc:sequence){
            int i=rc[1];
            int j=rc[0];
            if(i==0&&j==0){continue;}
              if(currRow+i>=0
                 && currRow+i<MapList.map[mapId].length
                  && currCol+j>=0
                 && currCol+j<MapList.map[mapId][0].length
                  && map[currRow+i][currCol+j]!=1){
                int[][] tempEdge={
                    {tempTarget[0],tempTarget[1]},
                    {currCol+j,currRow+i}
                };
```

```
                        astarQueue.offer(tempEdge);
                    }
                }
            }
            pathFlag=true;
            gameView.postInvalidate();
        }
    }.start();
}
```

如图：



Dijkstra 算法

Dijkstra 算法是典型的最短路径算法，一般用于求出从一点出发到达另一点的最优路径，但是因为其遍历运算的节点较多，所以运行效率较低。
其基本原理是：假设一个最短路径树，先将起点加到树中，并将从它发出的所有边加到搜索边界中，然后查找所有在计算搜索边界中的各个边所指向的节点中到源节点距离最近的点，将这个点加入树中，然后从这个节点继续查找。

从开始点取得 8 个方向最近的一个点，把 length[row][col]置为 1,然后把这个点加入到最短路径 hmPath 中，在从这个点开始找 8 个方向到这个点最近的点，把 length[row][col]置为 2，依次类推。

```
//记录到每个点的最短路径
HashMap<String,ArrayList<int[][]>> hmPath=
                        new HashMap<String,ArrayList<int[][]>>();
//记录路径长度
int[][] length=
  new int[MapList.map[mapId].length][MapList.map[mapId][0].length];
```

```
  //初始路径长度为最大距离都不可能的那么大
 for(int i=0;i<length.length;i++){
           for(int j=0;j<length[0].length;j++){
               length[i][j]=9999;
           }
  }
public void Dijkstra(){//Dijkstra算法
       new Thread(){
           public void run(){
               int count=0;//步数计数器
               boolean flag=true;//搜索循环控制
               int[] start={source[0],source[1]};//开始点col,row
               visited[source[1]][source[0]]=1;
              //计算此点所有可以到达点的路径及长度
               for(int[] rowcol:sequence){
           int trow=start[1]+rowcol[1];
           int tcol=start[0]+rowcol[0];
           if(trow<0||trow>18||tcol<0||tcol>18)continue;
           if(map[trow][tcol]!=0)continue;
           //记录路径长度
           length[trow][tcol]=1;
           //计算路径
           String key=tcol+":"+trow;
           ArrayList<int[][]> al = new ArrayList<int[][]>();
           al.add(new int[][]{{start[0],start[1]},{tcol,trow}});
           hmPath.put(key,al);
           //将去过的点记录
           searchProcess.add(
                   new int[][]{{start[0],start[1]},{tcol,trow}});
                  count++;
               }
           gameView.postInvalidate();
       //现在length中除了那8个点是1  其他的都是9999
       outer:while(flag){
       //找到当前扩展点K 要求扩展点K为从开始点到此点目前路径最短，且此点未考察过
                  int[] k = new int[2];
                  int minLen=9999;
                  for(int i=0;i<visited.length;i++){
                      for(int j=0;j<visited[0].length;j++){
                          //点未考察过
                          if(visited[i][j]==0){
                              //最开始都是9999除了延伸的那8个点 会取到第一个
                              if(minLen>length[i][j]){
                                  minLen=length[i][j];
```

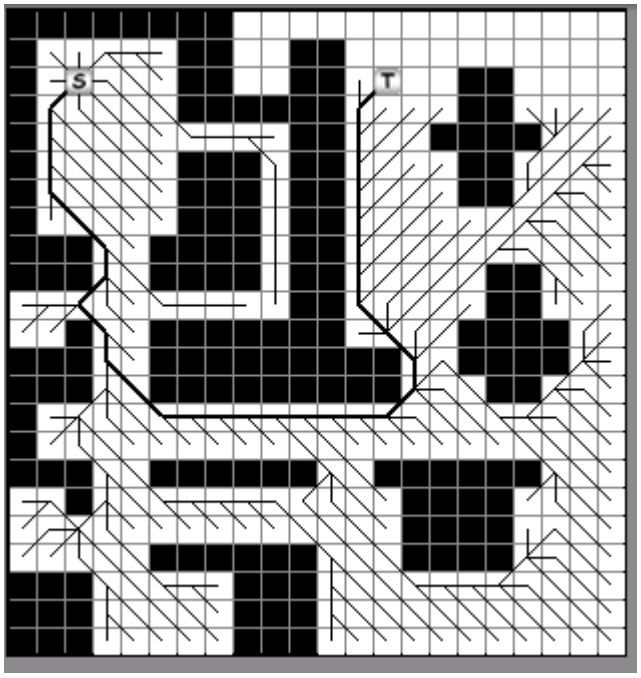
```
                            k[0]=j;//col
                            k[1]=i;//row
                        }
                    }
                }
            }
            visited[k[1]][k[0]]=1;//设置去过的点
            gameView.postInvalidate();//重绘
            int dk = length[k[1]][k[0]];//取出开始点到K的路径长度
            //取出开始点到K的路径
            ArrayList<int[][]> al=hmPath.get(k[0]+":"+k[1]);
            //循环计算所有K点能直接到的点到开始点的路径长度
            for(int[] rowcol:sequence){
                int trow=k[1]+rowcol[1];//计算出新的要计算的点的坐标
                int tcol=k[0]+rowcol[0];
          //若要计算的点超出地图边界或地图上此位置为障碍物则舍弃考察此点
          if(trow<0||trow>MapList.map[mapId].length-1
                ||tcol<0||tcol>MapList.map[mapId][0].length-1)
     continue;
if(map[trow][tcol]!=0)continue;
    int dj=length[trow][tcol];//取出开始点到此点的路径长度
    int dkPluskj=dk+1;//计算经K点到此点的路径长度
    //若经K点到此点的路径长度比原来的小则修改到此点的路径
    if(dj>dkPluskj){
        String key=tcol+":"+trow;
        //克隆开始点到K的路径
        ArrayList<int[][]> tempal= (ArrayList<int[][]>)al.clone();
        //将路径中加上一步从K到此点
        tempal.add(new int[][]{{k[0],k[1]},{tcol,trow}});
                    //将此路径设置为从开始点到此点的路径
                    hmPath.put(key,tempal);
                    //修改到从开始点到此点的路径长度
                    length[trow][tcol]=dkPluskj;
        //若此点从未计算过路径长度则将此点加入考察过程记录
        if(dj==9999){//将去过的点记录
        searchProcess.add(new int[][]{{k[0],k[1]},{tcol,trow}});
                    count++;
        }
        }
          //看是否找到目的点
          if(tcol==target[0]&&trow==target[1]){
                    pathFlag=true;
                    break outer;
          }
```

```
            }
             try{
                    Thread.sleep(timeSpan);
                }catch(Exception e){
                  e.printStackTrace();
              }
            }
         }
        }.start();
    }
```

路线的绘制：

```
   if(game.pathFlag){
         HashMap<String,ArrayList<int[][]>> hmPath=game.hmPath;
         ArrayList<int[][]> alPath=
         hmPath.get(game.target[0]+":"+game.target[1]);
        for(int[][] tempA:alPath){
            paint.setColor(Color.BLACK);
            paint.setStyle(Style.STROKE);//加粗
            paint.setStrokeWidth(2);//设置画笔粗度为2px
     canvas.drawLine(
        tempA[0][0]*(span+1)+span/2+6,tempA[0][1]*(span+1)+span/2+6,
        tempA[1][0]*(span+1)+span/2+6,tempA[1][1]*(span+1)+span/2+6,
                      paint
                  );
        }}
```

如图：



Dijkstra A*算法

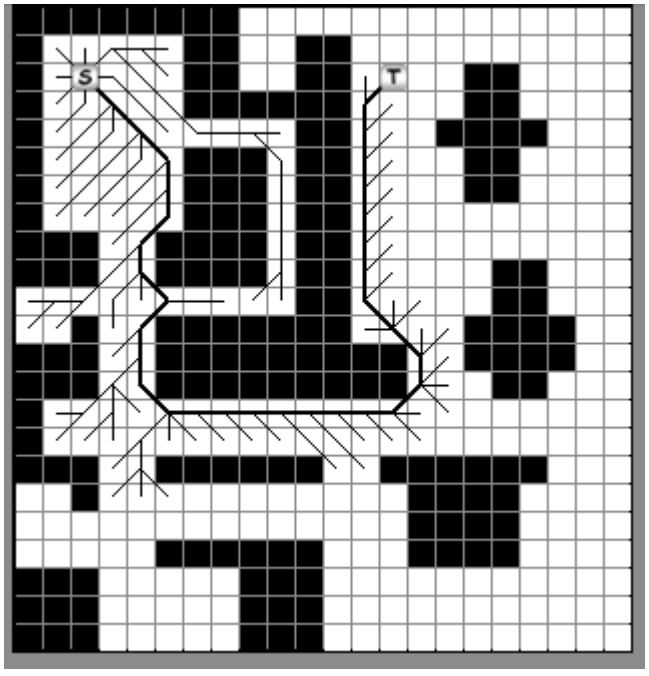
用 A*优化的 dijkstra 搜索算法得到的路径一定是最优路径，在真正游戏开发中这个应用比较多。总是在比较要路过的点与终点的距离最短。

```
public void DijkstraAStar(){//Dijkstra A*算法
    new Thread(){
        public void run(){
            int count=0;//步数计数器
            boolean flag=true;//搜索循环控制
        int[] start={source[0],source[1]};//开始点col,row
        visited[source[1]][source[0]]=1;
        //计算此点所有可以到达点的路径及长度
        for(int[] rowcol:sequence){
            int trow=start[1]+rowcol[1];
            int tcol=start[0]+rowcol[0];
            if(trow<0||trow>MapList.map[mapId].length-1
                      ||tcol<0||tcol>MapList.map[mapId][0].length-1)
               continue;
            if(map[trow][tcol]!=0)continue;
            //记录路径长度
            length[trow][tcol]=1;
            String key=tcol+":"+trow;//计算路径
            ArrayList<int[][]> al=new ArrayList<int[][]>();
            al.add(new int[][]{{start[0],start[1]},{tcol,trow}});
            hmPath.put(key,al);
            //将去过的点记录
            searchProcess.add(
                   new int[][]{{start[0],start[1]},{tcol,trow}});
            count++;
        }
        gameView.postInvalidate();
        outer:while(flag){
            int[] k=new int[2];
            int minLen=9999;
            boolean iniFlag=true;
            for(int i=0;i<visited.length;i++){
                for(int j=0;j<visited[0].length;j++){
                    if(visited[i][j]==0){
                     //与普通Dijkstra算法的区别部分=====begin===
                     if(length[i][j]!=9999){
                        if(iniFlag){//第一个找到的可能点
                           minLen=length[i][j]+
                     (int)Math.sqrt((j-target[0])*(j-target[0])
                     +(i-target[1])*(i-target[1]));
```

```
                    k[0]=j;//col
                    k[1]=i;//row
                    iniFlag=!iniFlag;
                }else{
                    int tempLen=length[i][j]+
                      (int)Math.sqrt((j-target[0])*(j-target[0])
                      +(i-target[1])*(i-target[1]));
                    if(minLen>tempLen){
                        minLen=tempLen;
                        k[0]=j;//col
                        k[1]=i;//row
                    }
                }
            }
        //与普通Dijkstra算法的区别部分==========end=============
                    }
                }
            }
        //设置去过的点
        visited[k[1]][k[0]]=1;
        //重绘
        gameView.postInvalidate();
        int dk=length[k[1]][k[0]];
        ArrayList<int[][]> al=hmPath.get(k[0]+":"+k[1]);
        for(int[] rowcol:sequence){
            int trow=k[1]+rowcol[1];
            int tcol=k[0]+rowcol[0];
            if(trow<0||trow>MapList.map[mapId].length-1
             ||tcol<0||tcol>MapList.map[mapId][0].length-1)
             continue;
            if(map[trow][tcol]!=0)continue;
                    int dj=length[trow][tcol];
                    int dkPluskj=dk+1;
        if(dj>dkPluskj){
          String key=tcol+":"+trow;
          ArrayList<int[][]> tempal=(ArrayList<int[][]>)al.clone();
          tempal.add(new int[][]{{k[0],k[1]},{tcol,trow}});
          hmPath.put(key,tempal);
          length[trow][tcol]=dkPluskj;
          if(dj==9999){
             //将去过的点记录
            searchProcess.add(new int[][]{{k[0],k[1]},{tcol,trow}});
             count++;
        }
```

```
            }
            //看是否找到目的点
            if(tcol==target[0]&&trow==target[1]){
                    pathFlag=true;
                    break outer;
            }
        }
         try{
             Thread.sleep(timeSpan);
              }catch(Exception e){
                  e.printStackTrace();
              }
             }
           }
          }.start();
        }
```

如图：



有限状态机

有限状态机就是一个具有有限数状态，并且能够根据相应的操作从一个状态变换到另一个状态，而在同一时刻只能处于一种状态下的智能体。
也就是使用状态来控制程序：

```
public boolean updateState(int state){//接收条件，更新状态的方法
        boolean result=true;
        switch(currentState){
```

```
            case 2:
            switch(state){
              case 0:
                    currentState=1;//切换状态
                    break;
               default:
                    result=false;//返回false表示状态切换出错
                }
            break;
        }
        return result;//返回true表示状态切换成功
    }
```

有限状态机的 OO 实现

当状态很多的时候，我们可以把状态封装成一个类，这样结构清晰，更好管理。

冒泡排序法

先说下交换的例子：

```
public class Swap {
        public static void swap(int[] data, int i, int j) {
                int temp = data[i];
                data[i] = data[j];
                data[j] = temp;
        }
}
```

下面是排序：

一次让相邻两个数值排序，每排一次最后出现一个最大值或是最小值排在序列的第一个或最后一个位置。

```
public void sort(int[] data) {
        for (int i = 0; i < data.length; i++) {
                for (int j = data.length - 1; j > i; j--) {
                        if (data[j] < data[j - 1]) {
                                swap(data, j, j - 1);
                        }
                }
        }
}
```

选择排序法

每一次比较的时候都找出一个最小或是最大的排在第一或是最后一个位置上。

```
public void sort(int[] data) {
    int temp;
    for (int i = 0; i < data.length; i++) {
        int lowIndex = i;
        for (int j = data.length - 1; j > i; j--) {
            if (data[j] < data[lowIndex]) {
                lowIndex = j;
            }
        }
        Swap.swap(data,i,lowIndex);
    }
}
```

插入排序法

就是从第二个元素开始一次和前面的数相比，比较 N-1 次，这样每一次都会把前面部分给排好顺序。

```
public void sort(int[] data) {
        int temp;
        for(int i=1;i<data.length;i++){
            for(int j=i;(j>0)&&(data[j]<data[j-1]);j--){
                Swap.swap(data,j,j-1);
            }
        }
}
```

快速排序法

通过一次分割，将无序序列分成两部分，其中一部分的元素值均不大于后一部分的元素值。然后用同样的方法对每一部分进行分割，一直到每一个子序列的长度小于或等于 1 为止。

```
public void sort(int[] data) {
    quickSort(data,0,data.length-1);
}

private void quickSort(int[] data,int i,int j){
    int pivotIndex=(i+j)/2;
```

```
        Swap.swap(data,pivotIndex,j);
        int k=partition(data,i-1,j,data[j]);
        Swap.swap(data,k,j);
        if((k-i)>1) quickSort(data,i,k-1);
        if((j-k)>1) quickSort(data,k+1,j);

}
```

分成部分；分割

```
private int partition(int[] data, int l, int r,int pivot) {
        do{
            while(data[++l]<pivot);
            while((r!=0)&&data[--r]>pivot);
            Swap.swap(data,l,r);
        }
        while(l<r);
        Swap.swap(data,l,r);
        return l;
}
```