# Java 2D学习笔记

2D图形系统在2D空间中对虚拟世界进行建模。与3D图形相比，2D图形在模型和绘制方面都更简单，2D对象的生成和操作更容易，而且，2D图形的绘制过程不需要像3D图形那样，通过使用复杂的投影来实现。虽然2D模型不能完全表现3D空间的特性，但是由于2D计算机图形具有简单与高效的特性，所以得到了广泛的应用。2D计算机图形是现代基于GUI程序的一个重要部分。

与2D图形相关的基本概念包括：

|- 绘制流水线； |- 对象空间；

|- 世界空间； |- 设备空间；

|- 坐标系； |- 图元；

|- 几何变换； |- 颜色；

|- 剪裁； |- 合成规则

Java 2D为2D图形提供了全面的支持。

## 一、初识Java 2D

### 1.1、Java 2D图形系统的坐标系

2D图形系统中，虚拟世界空间和视图空间都是2D的。绘制是指通过一些相对简单的变换来形成各种对象。

在数学中，我们在绘制一个图形对象的时候（例如：绘制一个圆），我们需要知道它在坐标系中的位置才能绘制（例如：圆的圆心）。但是在图形系统中，不是这样的，图形系统中，每个图形对象都有它自己的对象空间（可能你对“对象空间”感到比较迷茫，没关系，继续看下去），**我们先在它自己的对象空间中定义这个图像，然后经过某些变换之后把这个图形对象放到世界空间中去**，才能完成绘制。所以在此之前，我们需要介绍一下“对象空间”和“世界空间”。

先介绍**世界空间**：世界空间实际上就是一个坐标系，又可以把它称之为**世界坐标系（又被称为用户坐标系）**，世界坐标系在图形系统中是唯一的，不会有第二个世界坐标系。单独介绍世界空间是不能说明任何问题的，请清醒一下大脑，注意理解对象空间；

**对象空间**：对象空间也是一个坐标系，但这个坐标系的坐标轴可以旋转、平移、放缩，我又把它称为**模型坐标系**（官方名字叫做**本地坐标系**或**建模坐标系**），它只属于一个图形对象，初始的模型坐标系与世界坐标系是重合的。所以一个图形系统中可以有多个模型坐标系，因为一个图形系统中会有许多图形对象，每一个图形对象都有一个自己的模型坐标系。那么这个模型坐标系有什么用呢？上面有这样一句话“我们先在它自己的对象空间中定义这个图像，然后经过某些变换之后把这个图形对象放到世界空间中去”。意思就是说我们在定义图形对象的时候，会首先把图形“画”在它自己的模型坐标系中，然后经过某些变换（平移、旋转、缩放）最终会在世界坐标系将这个对象绘制出来，而模型坐标系在这之中扮演的角色就是完成图形的变换操作。

可能你还没有理解，那我们拿一个很简单的例子说说：在世界坐标系中以（3，4）为圆心画一个半径为4的圆。首先，我们创建一个单位圆对象，这个圆的圆心在它自己的模型坐标系的原点，半径为1，此时的模型坐标系是与世界坐标系重合的，我们需要将模型坐标系的单位长度放大四倍，然后将模型坐标系像x轴正向平移3个单位，向y轴正向平移4个单位。

你没有看错，我说的就是放大模型坐标系的单位长度，平移模型坐标系。所以我们在对图形对象进行变换的时候，图形对象相对于它的模型坐标系是完全不变的，进行变换的是它的模型坐标系，而变换完之后，图像对象在世界坐标系中的样子就完全变了，这就是模型坐标系的作用。

除了上述两类坐标系，还有一个“**设备坐标系**”。设备坐标系表示显示屏幕或打印机等输出设备的显示空间。如果学习过awt或swing的同学应该很容易理解，我们在JFrame的面板上放置按钮的时候，要给按钮设置location，这时的坐标系的原点在窗口面板的左上角，x轴正向向右，y轴正向向下。这个坐标系就是设备坐标系，设备坐标系的坐标值通常只能是整数，虽然这种情况明显不同于通常的数学表示形式，但是这种表示形式对计算机的显示设备来说更为自然。

默认情况下，Java 2D的世界坐标系与设备坐标系是一致的。

### 1.2、Java 2D的图形绘制过程

Java 2D中完成一个图形对象的绘制一般需要如下几个过程：  
 |- 构建2D图形对象

|- 对构建的图形对象进行几何变换

|- 应用颜色和其他绘图属性

|- 在图形设备上绘制场景

2D图形对象除了直线、多边形、椭圆等基本几何图形外，还可以是文本和图片等对象。

2D图形涉及到的变换，通常是仿射变换。对象变换可以改变对象的形状和位置，但是不能改变虚拟世界模型；观察变换虽然也不能改变虚拟世界模型，但是能改变世界空间中的整个场景视图，简单点讲，就是改变了世界模型中的观察点的位置，自然而然就改变了整个场景。

除了几何属性之外，很多其他属性对场景的绘制也有一定影响，这类属性中典型的例子就是颜色、透明度、纹理与线型等。

### 1.3、Graphics2D类

单独将这个类拿出来，是因为Java 2D中，是通过该类来访问绘图引擎的。在早期的Java版本中，图形的绘制是通过Java.awt.Graphics完成的。Java 2D使用了功能更加齐全的Graphics2D类来绘制图形，它继承了Graphics类，以兼容早期的AWT包中的GUI组件绘制功能。

Graphics类和Graphics2D类都是抽象类，原因在于它们的实现与平台有关。所以不能直接实例化该类对象。有两种方法可以获得Graphics2D类对象实例：①从paintComponent()方法的参数中取得；②通过调用getGraphics()方法取得。

在JComponent对象上绘制图形的标准方法是覆写paintComponent()方法：

void paintComponent(Graphics g)；

参数是一个Graphics类对象，他能够被强制转换为Graphics2D类对象。只要窗口中显示的内容需要重绘的时候（例如最小化的窗口复原时，窗口被遮挡的部分没被遮挡时），Java虚拟机会自动调用这个方法。因此这个方法所实现的绘制效果通常是持久的。

另一种获得Graphics2D对象的方法是利用getGraphics()方法：

Graphics getGraphics()；

这个方法返回的也是Graphics对象，同样也可以强制转换，它也能用来进行图形的绘制，但通常绘制效果是不持久的。

#### 1.3.1、Graphics类中的方法

在AWT中，Graphics类提供的方法可以控制绘制的各个方面。包括设置颜色、字体、平移坐标、设置XOR模式，以及直接画线条和椭圆等基本图形。

我在这里列出一些方法名称，具体作用可以根据方法名称理解，也可以自己去查文档。

（设置颜色、字体、XOR模式、平移坐标）

|- void setColor(Color c)； |- void setFont(Font f)；

|- void setXORMode(Color c)； |- void translate(int x, int y)；

（void drawXxx()）

|- void drawLine(int x1, int y1, int x2, int y2)；

|- void drawRect(int x, int y, int width, int height)；

|- void drawOval(int x, int y, int width, int height)；

|- void drawArc(int x, int y, int width, int height, int start, int arc)；

|- void drawPolygon(int[] xPoints, int[] yPoints, int nPoints)；

|- void drawString(String text, int x, int y)；

还有一个fillXxx()系列的方法，就不写了，很容易就能在文档中查到

上述方法都是早期AWT中的提供的绘图方法，那时并没有将建模-绘制区分开，也就是说当我们执行drawLine()这个方法时，既定义（建模）了一条线，也绘制了一条线。

而Java 2D中需要处理的图形问题要比这复杂，所以并没有将所有的绘制方法封装到一个类，而是将建模和几何变换的功能交由其他类来实现，Graphics2D类提供一些通用的方法来绘图。

**定义图形**对象由Java.awt.geom包中的许多类完成：

|- Point2D； |- Line2D；

|- Rectangle2D； |- Ellipse2D；

|- Arc2D； |- RoundRectangle2D；

这些类都是抽象类，需要借助它的内部类实例化对象。

**几何变换**由Java.awt.geom.AffineTransform类负责，有这样几个方法：  
 |- void rotate(double fovy)； 设置旋转角度

|- void transform(int tx, int ty)； 设置平移距离

|- void scale(double sx, double sy)； 设置放缩比例

最后的**绘制**过程由Graphics2D对象完成，有这样的方法：

|- void draw(Shape s)； |- void fill(Shape s)；

|- void transform(int tx, int ty)； |- void rotate(double fovy)；

|- void scale(double sx, double sy)；

|- void setTransform(AffineTransform Tx)；

我们可以看到AffineTransform类和Graphics2D类中都有几何变换的方法，那么两者有什么区别呢？还记得之前讲到过的“对象变换”和“观察变换”吗？AffineTransform类实现的就是“对象变换”，Graphics2D类实现的就是“观察变换”。

#### 1.3.2、体会“对象变化”和“观察变换”

**范例：**画一条（0,0）到（100,100）的直线，再画一条（100,100）到（200,100）的直线

|  |
| --- |
| package fuhao.demo;  import java.awt.Color;  import java.awt.Dimension;  import java.awt.Graphics;  import java.awt.Graphics2D;  import java.awt.geom.Line2D;  import javax.swing.JFrame;  import javax.swing.JPanel;  public class Demo extends JPanel{  public Demo() {  this.setPreferredSize(new Dimension(640, 480));  }  @Override  public void paintComponent(Graphics g) {  super.paintComponent(g);  Graphics2D g2d = (Graphics2D)g;  g2d.setColor(Color.blue);    **// 1、创建图形对象**  **Line2D line1 = new Line2D.Double(0, 0, 100, 100);**  **Line2D line2 = new Line2D.Double(100, 100, 200, 100);**  **// 2、绘制**  **g2d.draw(line1);**  **g2d.draw(line2);**  }    public static void main(String[] args) {  JFrame frame = new JFrame();  frame.setDefaultCloseOperation(JFrame.EXIT\_ON\_CLOSE);  frame.setLocation(300, 100);    Demo demo = new Demo();  frame.getContentPane().add(demo);  frame.setSize(frame.getContentPane().getPreferredSize());    frame.setVisible(true);  }  } |
|  |

这是没有经过任何几何变换的图形，那么现在我们用一下AffineTransform类进行几何变换（对象变换）。

|  |
| --- |
| // 2、创建几何变换  AffineTransform tr = new AffineTransform();  tr.rotate(Math.PI/6.0); // 顺时针旋转30° |

绘图部分的代码修改为如下

|  |
| --- |
| // 1、创建图形对象  Line2D line1 = new Line2D.Double(0, 0, 100, 100);  Line2D line2 = new Line2D.Double(100, 100, 200, 100);  **// 2、创建几何变换**  **AffineTransform tr = new AffineTransform();**  **tr.rotate(Math.PI/6.0); // 顺时针旋转30°**  **Shape s = tr.createTransformedShape(line1);**  // 3、绘制  g2d.draw(s);  g2d.draw(line2); |
|  |

我们创建了一个几何变换的对象，并设置了顺时针旋转30°，但是我们怎么知道到底是line1还是line2还是line1和line2一起执行这个几何变换呢。这里就又出现了一个方法：  
 Shape createTransformShape(Shape p)；

这个方法是AffineTransform中的方法，它接受一个Shape对象，并让这个Shape对象执行刚才设置的几何变换，然后将几何变换完成之后的Shape对象返回。我们这里传进去的是line1，所以只有line1执行了这个旋转的几何变换，line2没有影响。

那这个Shape到底是个什么呢？其实它是一个接口，所有的几何对象都实现了这个接口，也就是说这里的line1和line2都是Shape类对象。

上面这个例子讲的是“对象变换”，现在来看看“观察变换”，也就是用Graphics2D类对象进行几何变换

|  |
| --- |
| // 1、创建图形对象  Line2D line1 = new Line2D.Double(0, 0, 100, 100);  Line2D line2 = new Line2D.Double(100, 100, 200, 100);  // 2、进行几何变换  g2d.rotate(Math.PI/6.0);  // 3、绘制  g2d.draw(line1);  g2d.draw(line2); |
|  |

虽然“观察变换”的代码很简单，但是通过这张图我想大家都能看出问题了吧。没错，观察变换改变了观察图形的视角，简单地说就是将世界坐标系进行了改变，我们执行旋转30°的操作实际上是对世界坐标系进行了旋转，所以我们看到的原来平行的直线变成了斜的。那为什么对象变换只对该图形对象产生影响，而不对其他图形对象产生影响，或者说对象变换为什么对世界坐标系没有影响？这就是前面所讲到的模型坐标系（建模坐标系）的好处，我们在进行对象变换的时候，改变的仅仅是这个对象的模型坐标系，而不会对其它对象的模型坐标系产生任何影响，更不用说对世界坐标系产生影响。**而且有一点我再次强调，进行变换的时候，不是图形在变换，而是坐标系在变换**。你试一试先旋转后平移与先平移后旋转的差别，就会明白了。

### 1.4、绘图方程

根据数学方程作图是一种简单的图形学应用。给出一个图形方程，为其做图的最简单的方法，就是求解出一系列满足方程的坐标，然后绘制这些点。

对于形如y = f(x)的方程，最直接的方式就是选择一组x坐标，计算出对应的y坐标。而对于隐函数F(x, y) = 0它的计算就比较复杂了，我们可以将这些隐函数转化为参数方程，以便于计算。

**范例**：以画圆为例

|  |
| --- |
| super.paintComponent(g);  Graphics2D g2d = (Graphics2D)g;  g2d.setColor(Color.blue);  g2d.translate(200, 200);    // x = rcos(t); y = rsin(t)  int nPoints = 1000;  int r = 100;    int x1 = r;  int y1 = 0;  for(int i=0;i<nPoints;i++) {  double t = i\*2\*Math.PI/nPoints;  int x2 = (int) (r\*Math.cos(t));  int y2 = (int) (r\*Math.sin(t));  g2d.drawLine(x1, y1, x2, y2);  x1 = x2;  y1 = y2;  } |

这里我要更正上面的一个错误，上面我说到了计算一系列满足方程的点，绘制这些点，而这里我们实际上是绘制的直线：将这一系列点计算出来，每相邻的两个点连成直线，达到以直代曲的目的，这里我们用的就是圆的参数方程（当然这个圆圆心再原点）。

所以只要知道了参数方程，我们都可以用以直代曲的方式去绘制这些图形。

### 1.5、几何模型

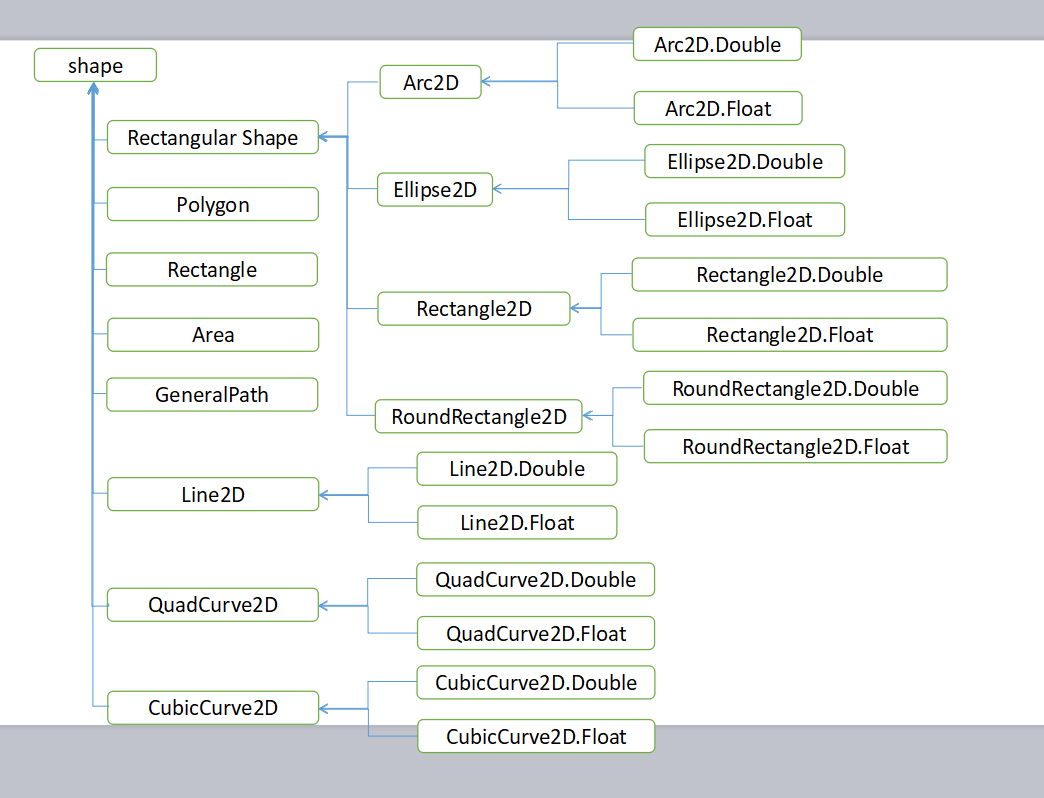
在2.3节里面，我就已经介绍过一部分图形对象，这里一是回顾，而是丰富。

组成虚拟世界的2D图形对象，除了基本的几何图形对象外，还可以是文本对象和图像，这里只讲几何图形对象。

#### 1.5.1、形状

在Java 2D中，如果实现了Shape接口，那么这个几何对象就可以由Graphics2D类对象绘制，我们已经知道了，就是void draw(Shape s)方法以及void fill(Shape s)方法。

Java 2D提供的工具可以用来构建基本形状，并且可以通过形状组合，构建更为复杂的形状。Shape接口（类）的层次结构如图所示：



所有的Xxx2D类都是抽象类（又说了一遍），它们只有借助自己的内部类才能够实例化，一般内部类就是两个，分别是坐标类型为double和坐标类型为float的两个类。而后面没有2D的类是可以直接实例化的。

这里面，我们不熟悉应该只有Area、GeneralPath、QuadCurve2D、CubicCurve2D这四个类。

Area类，这个类的作用之前提到过，将基本几何图形组合，构建更复杂的图形，具体怎么用，可以自行百度学习，只要掌握这个类中的一些方法就可以很容易的组合。

GeneralPath类，用来构造自定义形状的类。

QuadCurve2D类是二次曲线（一元二次函数）类，用这个类画二次曲线需要三个点作为参数，二次曲线的起点、终点、以及最值点。

CubicCurve2D是三次曲线（一元三次函数）类，用这个类画三次曲线需要四个点作为参数，三次曲线的起点、终点、两个最值点。

**范例**：定义一个窗口，窗口又菜单栏，用户通过菜单栏可以选择自己想要绘制的形状，然后通过拖动鼠标在屏幕上绘制图形。

|  |
| --- |
| package fuhao.demo;  import java.awt.Dimension;  import java.awt.Graphics;  import java.awt.Graphics2D;  import java.awt.Shape;  import java.awt.event.ActionEvent;  import java.awt.event.ActionListener;  import java.awt.event.MouseEvent;  import java.awt.event.MouseListener;  import java.awt.event.MouseMotionListener;  import java.awt.geom.Ellipse2D;  import java.awt.geom.Line2D;  import java.awt.geom.Rectangle2D;  import java.util.LinkedList;  import javax.swing.JFrame;  import javax.swing.JMenu;  import javax.swing.JMenuBar;  import javax.swing.JMenuItem;  import javax.swing.JPanel;  public class Demo extends JPanel{  public static final int LINE = -1;  public static final int RECT = -2;  public static final int ELLIPSE = -3;    private int kind;  private LinkedList<Shape> shapes = new LinkedList<>();  private int x1,y1,x2,y2;    public Demo() {  this.setPreferredSize(new Dimension(700, 700));  this.addMouseListener(new MouseListener() {  @Override  public void mouseReleased(MouseEvent e) {  Demo.this.x2 = e.getX();  Demo.this.y2 = e.getY();  Shape shape = null;  switch(Demo.this.kind) {  case LINE:  Line2D line = new Line2D.Double(Demo.this.x1, Demo.this.y1, Demo.this.x2, Demo.this.y2);  shape = line;  break;  case RECT:  Rectangle2D rect = new Rectangle2D.Double(Demo.this.x1, Demo.this.y1, Demo.this.x2-Demo.this.x1, Demo.this.y2-Demo.this.y1);  shape = rect;  break;  case ELLIPSE:  Ellipse2D e1 = new Ellipse2D.Double(Demo.this.x1, Demo.this.y1, Demo.this.x2-Demo.this.x1, Demo.this.y2-Demo.this.y1);  shape = e1;  break;  }  Demo.this.shapes.add(shape);  Graphics2D g = (Graphics2D)getGraphics();  g.draw(shape);  }  @Override  public void mousePressed(MouseEvent e) {  Demo.this.x1 = e.getX();  Demo.this.y1 = e.getY();  }  @Override  public void mouseExited(MouseEvent e) {  }  @Override  public void mouseEntered(MouseEvent e) {  }  @Override  public void mouseClicked(MouseEvent e) {  }  });  this.init();  }    private void init() {  JFrame frame = this.getFrame();  // 创建菜单并加到窗口上  JMenuBar mb = new JMenuBar();  frame.setJMenuBar(mb);    // 创建一个菜单栏，并加到菜单栏之中  JMenu menu = new JMenu("绘图");  mb.add(menu);    // 为上面的菜单栏创建多个菜单项，并为菜单项设置响应事件  // 1、直线  JMenuItem line = new JMenuItem("直线");  menu.add(line);  line.addActionListener(new ActionListener() {  @Override  public void actionPerformed(ActionEvent e) {  Demo.this.kind = LINE;  }  });    // 2、矩形  JMenuItem rect = new JMenuItem("矩形");  menu.add(rect);  rect.addActionListener(new ActionListener() {  @Override  public void actionPerformed(ActionEvent e) {  Demo.this.kind = RECT;  }  });    // 3、椭圆  JMenuItem ellipse = new JMenuItem("椭圆");  menu.add(ellipse);  ellipse.addActionListener(new ActionListener() {  @Override  public void actionPerformed(ActionEvent e) {  Demo.this.kind = ELLIPSE;  }  });    frame.setVisible(true);  }    @Override  public void paintComponent(Graphics g) {  super.paintComponent(g);  Graphics2D g2 = (Graphics2D)g;  for(int i=0;i<this.shapes.size();i++) {  g2.draw(this.shapes.get(i));  }  }    public JFrame getFrame() {  JFrame frame = new JFrame();  frame.setContentPane(this);  frame.setDefaultCloseOperation(JFrame.EXIT\_ON\_CLOSE);  frame.setSize(frame.getPreferredSize());  return frame;  }    public static void main(String[] args) {  Demo demo = new Demo();  }  } |

鼠标释放的一瞬间将图形绘制出来，并且保存在链表之中，然后再paintComponent方法中对绘制的图形进行重绘，让绘制效果具有持久性。

这个简单的程序并没有支持所有的特性，如不支持I/O操作，但它体现了一个很重要的特性：交互。程序允许用户通过鼠标绘制任意数量的图形。

#### 1.5.2、几何图形的组合

前面简单的介绍了一下Area类，如果当时你自己尝试过或者去百度上查过，其功能你应该是了解的比较清楚了，如果当时你没有尝试或去查资料，没关系，这节将会讲到。

生成更复杂的形状的一种方式，就是通过组合已经存在的几种形状而得到新的形状，这就是**构造区域几何模型**。Area类就是为了完成构造区域几何模型而设计的，它支持四个操作：合并、交叉、差分、对称差分。这都是集合之中的操作，在这里是**针对两个区域的操作**，并生成一个新的区域。

合并操作的结果是，将两个区域的所有点包括进来形成的一个整体；交叉操作的结果是，将同时属于两个区域的点放在一起形成的整体；差分操作的结果是，将属于第一个区域，但是不属于第二个区域的点保留下来形成的整体；对称差分的结果是，将两个区域合并的点去除掉两个区域交叉的点形成的整体（类似集合中的对称差）。

这四类操作分别由Area类中的四个方法完成。按顺序是：void add(Shape s)、void intersect(Shape s)、void subtract(Shape s)、void exclusiveOr(Shape s)。

上面讲到这四类操作必须有两个区域，而这四个方法都只有一个参数，这四个方法的参数都表示的是组合时的第二个区域，还有一个参数，或者说第一个区域去哪了呢？第一个区域的设置方式：通过Area的构造方法，传进去一个Shape类对象。但是如果在构造方法中没有传参数时，在执行这四个方法的时候，会将第一个区域当作没有包含任何点的区域与第二个区域执行对应的四类操作。

这样组合的好处有哪些呢？

**1、不会对传进来的Shape对象有任何影响；**

**2、可以无限制的传入Shape类对象进行组合。**

**3、Area对象实现了Shape接口，可以直接由Graphics2D类对象绘制**

#### 1.5.3、一般路径

Graphics2D引擎内部通过五种基本的曲线段**或**操作，完成任意形状的边界绘制。Shape接口提供的方法可以或者一个PathIterator接口，它用五种类型的曲线段来描述边界的路径。PathIterator定义了五个曲线段常量：SEG\_MOVETO、SEG\_LINETO、SEG\_QUADTO、SEG\_CUBICTO、SEG\_CLOSE。

构造自定义形状的一种强大的方法是使用GeneralPath类，它直接支持这五种曲线段来构成路径。GeneralPath中有五个方法，分别实现对这五类曲线段类型的构建：

void moveTo(float x, float y)；

void lineTo(float x, float y)；

void quadTo(float x1, float y1, float x2, float y2)；

void curveTo(float x1, float y1. float x2, float y2, float x3, float y3)；

void closePath()；

路径的构造过程可以视为是通过一支“笔”进行绘制的过程。在任意时刻，这个“笔”都有一个“当前位置”。通过moveTo()方法将笔移动到一个新的位置，而且不会绘制任何东西；通过lineTo()方法将笔移动到一个新的位置，并且会在原来的当前位置与新的位置之间绘制一条直线；通过quadTo()方法会以笔的当前位置为起点，以（x2, y2）为终点，（x1, y1）为最值点绘制一条二次曲线，并且将笔的当前位置修改为（x2,y2）；通过curveTo()方法也会以笔的当前位置为起点，（x3, y3）为终点，（x1, y1）（x2, y2）为两个最值点绘制一条三次曲线，并将笔的当前位置修改为（x3, y3）；通过调用closePath()方法会将笔的当前位置的点与上一次调用moveTo()方法设置的点之间连接一条线段。

GeneralPath定义的仅仅是路径，或者说是边界（轮廓），但是一个完整的形状，还需要为其指定形状的内部与外部，特别是对这个形状进行填充（fill）的时候。**缠绕规则**用来定义在什么条件下某个区域属于形状的内部。PathIterator接口定义了两种缠绕规则：WIND\_EVEN\_ODD（奇偶规则）、WIND\_NON\_ZERO（非零规则）。

**奇偶规则**：要确定某一块区域是否属于形状的内部，可以画一条穿过该区域的直线，考察从外部到此区域这条直线与路径相交的次数。奇偶规则规定，如果相交次数为奇数，则该区域在内部，否则在外部。实际上路径是用来区分内部与外部的分界线。

**非零规则**：非零规则的定义比较难想清楚，这里直接讲简单的。非零规则规定，沿着路径的方向移动时，路径的左侧被定义为内部，右侧被定义为外部（学过多重积分的同学应该很熟悉吧）。

**范例**：观察并理解两种缠绕规则

|  |
| --- |
| GeneralPath path1 = new GeneralPath();  path1.moveTo(100, 100);  path1.lineTo(300, 100);  path1.lineTo(300, 300);  path1.lineTo(100, 300);  path1.closePath();  path1.moveTo(150, 150);  path1.lineTo(250, 150);  path1.lineTo(250, 250);  path1.lineTo(150, 250);  path1.closePath();  GeneralPath path2 = new GeneralPath();  path2.moveTo(400, 100);  path2.lineTo(600, 100);  path2.lineTo(600, 300);  path2.lineTo(400, 300);  path2.closePath();  path2.moveTo(450, 150);  path2.lineTo(550, 150);  path2.lineTo(550, 250);  path2.lineTo(450, 250);  path2.closePath();  path2.setWindingRule(GeneralPath.WIND\_EVEN\_ODD);  GeneralPath path3 = new GeneralPath();  path3.moveTo(100, 400);  path3.lineTo(300, 400);  path3.lineTo(300, 600);  path3.lineTo(100, 600);  path3.closePath();  path3.moveTo(150, 450);  path3.lineTo(250, 450);  path3.lineTo(250, 550);  path3.lineTo(150, 550);  path3.closePath();  path3.setWindingRule(GeneralPath.WIND\_NON\_ZERO);  g2.draw(path1);  g2.fill(path2);  g2.fill(path3); |
|  |

GeneralPath类中还有一个方法reset()方法，重置路径，也就是删除之前“笔”绘制的所有路径，尝试一下就知道它的作用了。

### 1.6、总结

本节中所学到的主要的类与方法：

Graphics2D类、Shape接口、GeneralPath和Area类

Jpanel.paintComponent(Graphics g)；

Graphics2D.draw(Shape s)；

Graphics2D.fill(Shape s)；

主要概念：

三类坐标系、参数方程与方程的图像、构造区域几何模型、路径、缠绕规则

通过本章的学习，我们能在画布上绘制简单的几何图形，以及通过简单的几何图形组合而成的复杂的几何图形，我们能绘制由参数方程确定的曲线图像

## 二、2D图形学：绘制细节

上一节介绍了如何绘制形状，但是一个图形对象的属性和操作并不仅仅局限于它的几何形状，还有诸如颜色、笔划类型、变换、复合规则、剪裁路径和绘制提示，这些也都是影响绘制效果的重要因素。

### 2.1、颜色和涂色

#### 2.1.1、颜色空间

颜色是图形系统的重要属性。

**颜色空间：**系统通常以数字的形式定义可见的颜色，不同的颜色空间，描述颜色的精确和量化程度不一样。一般方法是选择一部分确定的基色（如红、绿、蓝），并用基色的组合表示任意颜色。

CIEXYZ是一种颜色标准，使用X、Y、Z三种基色代替红、绿、蓝三种基色，任何可见的颜色都可以表示成XYZ三种基色的线性组合。然而，物理设备上直接实现CIEXYZ很困难。

大多数显示器使用的是RGB（红绿蓝）颜色空间，打印机使用的CMYK颜色空间。

sRGB是一种标准的RGB颜色空间，与设备无关，我们学习中就在这种颜色空间下。

#### 2.1.2、颜色

构建一个几何形状后，可以使用fill或draw方法绘制，为了给对象设置绘制的颜色，用到Graphics类中这个方法：void setColor(Color c)；

一个Color对象定义一种颜色，而且Color类默认情况下使用sRGB颜色空间，一个颜色包含四个分量：red、green、blue、alpha。其中alpha是指颜色的透明度。

Color类由许多构造方法，可以自行查看文档或源代码，一般选择的构造方法是：  
 Color(int r, int green, int blue)；

这三个参数都是0到255之间的任何整数，如果想设置颜色的透明度，可以加上第四个参数，并且还可以在构造方法中设置新的颜色空间（一般用不到）。

**范例：**设置透明度

|  |
| --- |
| g2.setColor(new Color(255, 0, 0, 155));  g2.fill(new Ellipse2D.Float(100, 100, 100, 100));  g2.setColor(new Color(0, 255, 0, 155));  g2.fill(new Ellipse2D.Double(150, 150, 100, 100));  g2.setColor(new Color(0, 0, 255, 155));  g2.fill(new Ellipse2D.Double(100, 150, 100, 100)); |
|  |

**范例**：通过用户设置颜色分量绘制这三个圆

|  |
| --- |
| package fuhao.demo;  import java.awt.Color;  import java.awt.Dimension;  import java.awt.Graphics;  import java.awt.Graphics2D;  import java.awt.geom.Area;  import java.awt.geom.Ellipse2D;  import javax.swing.JFrame;  import javax.swing.JPanel;  import javax.swing.JSlider;  import javax.swing.event.ChangeEvent;  import javax.swing.event.ChangeListener;  public class Demo extends JPanel{  private int red = 200;  private int green = 200;  private int blue = 200;    public Demo() {  this.setPreferredSize(new Dimension(700, 700));  this.setLayout(null);  JSlider redSlider = new JSlider(JSlider.VERTICAL, 0, 255, 200);  redSlider.setSize(20, 255);  redSlider.setLocation(550, 200);  this.add(redSlider);  redSlider.addChangeListener(new ChangeListener() {  @Override  public void stateChanged(ChangeEvent e) {  Demo.this.red = ((JSlider) e.getSource()).getValue();  Demo.this.repaint();  }  });    JSlider greenSlider = new JSlider(JSlider.VERTICAL, 0, 255, 200);  greenSlider.setSize(20, 255);  greenSlider.setLocation(600, 200);  this.add(greenSlider);  greenSlider.addChangeListener(new ChangeListener() {  @Override  public void stateChanged(ChangeEvent e) {  Demo.this.green = ((JSlider) e.getSource()).getValue();  Demo.this.repaint();  }  });    JSlider blueSlider = new JSlider(JSlider.VERTICAL, 0, 255, 200);  blueSlider.setSize(20, 255);  blueSlider.setLocation(650, 200);  this.add(blueSlider);  blueSlider.addChangeListener(new ChangeListener() {  @Override  public void stateChanged(ChangeEvent e) {  Demo.this.blue = ((JSlider) e.getSource()).getValue();  Demo.this.repaint();  }  });  }    @Override  public void paintComponent(Graphics g) {  super.paintComponent(g);  Graphics2D g2 = (Graphics2D)g;    Ellipse2D redCircle = new Ellipse2D.Double(200, 200, 200, 200);  Ellipse2D greenCircle = new Ellipse2D.Double(150, 300, 200, 200);  Ellipse2D blueCircle = new Ellipse2D.Double(250, 300, 200, 200);    Area red = new Area(redCircle);  Area green = new Area(greenCircle);  Area blue = new Area(blueCircle);    Area rg = new Area(red);  rg.intersect(green);    Area rb = new Area(red);  rb.intersect(blue);    Area gb = new Area(green);  gb.intersect(blue);    Area rgb = new Area(red);  rgb.intersect(gb);    red.subtract(rg);  red.subtract(rb);  green.subtract(rg);  green.subtract(gb);  blue.subtract(rb);  blue.subtract(gb);    g2.setColor(new Color(this.red, 0, 0)); g2.fill(red);  g2.setColor(new Color(0, this.green, 0)); g2.fill(green);  g2.setColor(new Color(0, 0, this.blue)); g2.fill(blue);  g2.setColor(new Color(this.red, this.green, 0)); g2.fill(rg);  g2.setColor(new Color(this.red, 0, this.blue)); g2.fill(rb);  g2.setColor(new Color(0, this.green, this.blue)); g2.fill(gb);  g2.setColor(new Color(this.red, this.green, this.blue)); g2.fill(rgb);  }    public static void main(String[] args) {  Demo demo = new Demo();  JFrame frame = new JFrame();  frame.setContentPane(demo);  frame.setDefaultCloseOperation(JFrame.EXIT\_ON\_CLOSE);  frame.setSize(frame.getPreferredSize());  frame.setVisible(true);  }  } |
|  |

这是一个用户**交互式程序**，用户通过拖动滚动条来改变颜色分量，三个滚动条分别代表RGB三个分量，每次拖动滚动条，都会重新绘制整个图形。

#### 2.1.3、涂色

setColor()方法是Graphics类中的旧方法，它仅使用纯色进行绘制。Java 2D中的Graphics2D类使用一个更强大的setPaint(Paint p)方法来控制颜色的绘制。Paint是一种颜色的抽象，由Color类和其他一些类实现Paint接口，与简单的纯色相比，它能够表述更多的属性。

GradientPaint类是实现了Paint接口的一个类，它定义了一种颜色变化的涂色方式：**渐变涂色**。渐变涂色由两个点和两种颜色定义，当位置从第一个点向第二个点移动的时候，颜色会逐渐从第一种颜色变为第二种颜色。渐变涂色可以是循环的，也可以是非循环的，循环的渐变涂色周期性地重复同样地模式。

为了创建一种非循环的渐变涂色效果，使用如下构造方法：  
 GradientPaint(float x1, float y1, Color c1, float x2, float y2, Color c2)；

GradientPaint(Point2D p1, Color c1, Point2D p2, Color c2)；

为了创建一种循环的渐变涂色效果，使用如下构造方法：

GradientPaint(float x1, float y1, Color c1, float x2, float y2, Color c2, boolean cycl)；

GradientPaint(Point2D p1, Color c1, Point2D p2, Color c2, boolean cycl)；

将最后一个参数设置为true即可。

渐变模式中并不是连接这两点间的直线才会有渐变效果，而是从第一个点向第二个点移动的方向上就会产生渐变效果。如果是非循环的话，只有在这两点间才会有渐变效果（注意并不只是连接两点的直线），如果是循环的话，它会每一长度产生一个渐变。

例如：我将两点坐标分别设置为（100, 100）、（200, 100）。如果是非循环渐变，则只有（100, 100）到（200, 100）之间绘制的图形（直线、矩形、椭圆均可）才会有渐变效果；而如果是循环渐变，则会在（100, 100）到（200,100），（200, 100）到（300, 100）....之间都会有渐变效果。

TexturePaint类也是实现了Paint接口的类，它允许使用纹理模式填充对象。一幅图像和描点矩阵用于定义一个纹理涂色方式。构造方法：

TexturePaint(BufferedImage image, Rectangle2D anchor)；

其中第一个参数就是纹理，第二个参数就是描点矩形。描点矩形指的是只有在设备坐标系上的这个矩形范围内绘制的图形，才能正常的使用纹理涂色模式，超出这个矩形范围会将纹理以循环的方式绘制。

**范例**：体会渐变涂色和纹理涂色

|  |
| --- |
| BufferedImage img = null;  try {  img = ImageIO.read(new FileInputStream(new File("C:\\Users\\fuhao\\Desktop\\纹理.png")));  } catch (Exception e) {  e.printStackTrace();  }  TexturePaint tp = new TexturePaint(img, new Rectangle2D.Double(100, 100, img.getWidth(), img.getHeight()));  g2.setPaint(tp);  g2.fill(new Rectangle2D.Float(50, 50, 200, 200));  GradientPaint gp1 = new GradientPaint(300, 400, Color.red, 400, 400, Color.blue, false);  g2.setPaint(gp1);  g2.setStroke(new BasicStroke(20));  g2.draw(new Line2D.Double(300, 400, 600, 400));  GradientPaint gp2 = new GradientPaint(300, 400, Color.red, 400, 400, Color.blue, true);  g2.setPaint(gp2);  g2.setStroke(new BasicStroke(20));  g2.draw(new Line2D.Double(300, 500, 600, 500)); |
|  |

左上角是利用纹理涂色模式绘制的矩形，在生成TexturePaint对象时，我将描点矩形设置为（100, 100, width, height），而我这个矩形的左上角坐标是（50, 50）,很明显可以看到那个窗口关闭的“×”，这个本应该是图片最右上角的，因为我们绘制的矩形超过了它描点矩形范围，所以会以循环的方式，将最右边的纹理绘制在这里。

右下角是两个渐变模式，一个是循环渐变，一个是非循环渐变，我想区别一目了然吧。

### 2.2、笔划

计算机图形学中绘制的直线并不是宽度为0的理想直线。在实际应用中，线必须有特定的形状，定义这些形状细节的属性称为**笔划**。一个笔划可以包含线的宽度、虚线风格、线帽的形状和线的连接风格等属性。

Java 2D中使用Stroke接口表示笔划。BasicStroke类实现了Stroke接口，它允许设置线的宽度、线帽、连线风格和虚线，其构造方法如下：

BasicStroke()；

BasicStroke(float width)；

BasicStroke(float width, int cap, int join)；

BasicStroke(float width, int cap, int join, float miterlimit)；

BasicStroke(float width, int cap, int join, float miterlimit, float[] dash, float dashphase)；

其中width定义了线的宽度；

参数cap定义了线帽风格，有如下几种取值：CAP\_BUTT、CAP\_ROUND、CAP\_SQUARE，自己去尝试一下就知道有什么不同了；

参数join定义了连接风格，有如下几种取值：JOIN\_BEVEL、JOIN\_MITER、JOIN\_ROUND，参数miterlimit对JOIN\_MITER进行限制，防止一个很小的夹角的两边出现过长的连接；

dash数组通过设置ON/OFF段的长度定义了一个虚线状态，参数dashphase定义了虚线的起始点，例如：dash数组为{20, 20, 40, 40}，dashphase为0，意思就是说在绘制的线段中从线段的起点开始，按照实线长度20虚线长度20实线长度40虚线长度40的样式进行虚线绘制，至于当dashphase不为0时，自己尝试便知。

Graphics2D中用void setStroke(Stroke s)方法设置当前笔划。

### 2.3、仿射变换

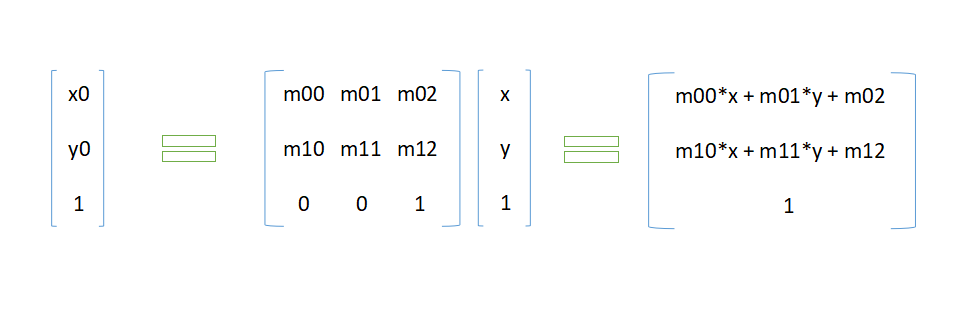
几何对象在绘制之前，需要经过一系列的变换。在计算机图形学中一般使用的变换称为**仿射变换**。仿射变换保留线的平行性质。常见的仿射变换包括：平移、旋转、放缩、反射、错切。

平移、旋转、放缩三种变换应该不需要讲了。我们着重讲剩下两种我们不熟悉的变换。

**反射变换**：沿着某一条直线建立该几何对象的镜像。反射变换的结果由变换直线决定

**错切变换**：错切变换是关于某条直线的，是将几何对象上的点已动工一定距离，该距离与点到变换直线的有符号距离成正比。简单地说，几何对象上所有的点都要平行于变换直线移动，在变换直线上的点不需要移动，在变换直线反方向上的点以相反的方向运动，而点移动的距离由错切因子决定。

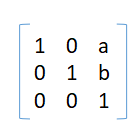
在数学上，一个二维仿射变换可以用一个三维矩阵表示。已知一个点（x, y）经过如下的仿射矩阵变换之后的坐标为（x0, y0），则三者的关系如图所示



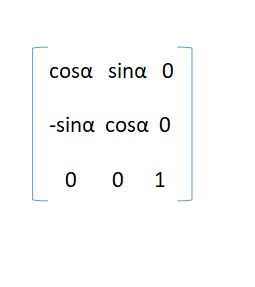
也就是说 x0 = m00\*x + m01\*y + m02；y0 = m10\*x + m11\*y + m12；

那我们可以将上述的几种基本变换的变换矩阵给计算出来。

**平移变换**：已知点（x, y）平移（a, b），平移后点的坐标为（x+a, y+b）。那么这个变换矩阵怎么计算呢？在上面这张图中，x0 = x+a，而x0 = m00\*x + m01\*y + m02，很明显，m00=1，而m01=0，m02=a，同样的道理可以得m10=0，m11=1，m12=b。也就是说变换矩阵是这个

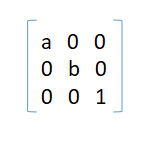


**旋转变换**：已知点（x, y）绕（0, 0）点顺时针旋转α角度后得坐标为（x\*cosα+y\*sinα, -x\*sinα+y\*cosα），所以它的变换矩阵为

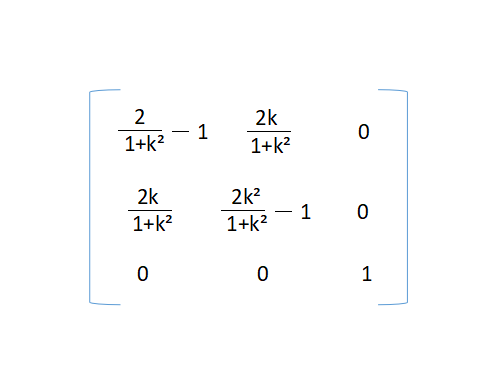


由于任意点得旋转太过复杂，这里就不讨论了。

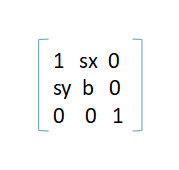
**放缩变换**：已知点坐标（x, y），x方向上放缩因子为a，y方向上放缩因子为b，则变换后得坐标（x\*a, y\*b），则它的变换矩阵为



**反射变化**：一个关于直线 y=kx的反射变换矩阵为



错切变化：sx和sy分别是x方向和y方向上的两个错切因子



Java 2D中有一个类专门负责进行各种仿射变换，这个类之前我们接触过，就是AffineTransform类，这个类既对上面的五类特殊的变换进行了实现，也对一般的变换进行了实现，也就是说，通过这个类，我们可以自己设置变换的仿射矩阵。首先看**构造方法**：

|- 无参构造：AffineTransform()；仅仅是一个变换对象，没有任何变换操作，也可以理解为是单位矩阵。

|- public AffineTransform(double m00, double m10, double m01, double m11, double m02, double m12)；这六个参数设置的就是仿射矩阵的前两行，由于第三行是0、0、1不会改变，所以不用对其进行设置。参数类型也可以是float类型

|- public AffineTransform(double[] flatmatrix)；和第二个构造方法相同，只不过把六个参数放进了数组，同样参数类型也可以是float[]型。

构造方法可以构造一个单位变换矩阵，也可以构造自己的变换矩阵，接下来看看用什么方法实现了上面五类基本仿射变换：

|- public void setToIdentity()；将当前变换矩阵设置为单位矩阵。

|- public void setToTranslation(double tx, double ty)；将上面讲到的平移变换矩阵中的a、b替换为这里的tx、ty。即进行平移量为（tx, ty）的平移变化。

|- public void setToRotation(double theta)；绕原点顺时针旋转theta角度的旋转变换。

|- public void setToRotation(double theta, double anchorx, double anchory)；绕点（anchorx, anchory）顺时针旋转theta角度的旋转变换

|- public void setToScale(double sx, double sy)；将上面讲到的放缩变换矩阵中的a、b替换为sx、sy，进行放缩变换。

|- public void setToShear(double shx, double shy)；进行错切变换，我这里把错切变换矩阵给出来，这两个参数的含义，我现在也没弄明白，但是并不妨碍计算变换后的坐标。

这里并没有给出反射变换的相关方法，但是我们可以根据自己设置仿射矩阵达到反射变换的目的，利用构造方法即可。

当我们设置好各种变换矩阵之后，要将其应用到特定的图形对象上（即其模型坐标系上），或应用在世界坐标系中。应用在图形对象上的方法我们用过：

Shape createTransformedShape(Shape s)；这是AffineTransform中的方法。

应用在世界坐标系上的方法我们也用过：

void setTransform(AffineTransform tx)；这是Graphics2D中的方法。

### 2.4、复合变换

简单变换组合起来就能形成复合变换。例如：可以先旋转，再平移。

复合变换的变换矩阵是多个简单变换的变换矩阵的乘积，假设M1、M2、M3分别是三个变换，它们的变换矩阵分别为T1、T2、T3，相应的复合变换T1\*T2\*T3的变换矩阵就是T1\*T2\*T3。**注意：复合变换的运算顺序是不可交换的，因此复合顺序很重要。复合变换是从右至左结合的。**例如：T1\*T2\*T3复合时，变换顺序是T3、T2、T1。

Java 2D中AffineTransform类提供下面几个方法支持复合变换：  
 |- public void rotate(double theta)；

|- public void rotate(double theta, double anchorx, double anchory)；

|- public void scale(double sx, double sy)；

|- public void shear(double shx, double shy)；

|- public void translate(double tx, double ty)；

这和前面讲到的setTo\*方法不同，setTo\*方法是将当前变换矩阵设置为参数矩阵，而这几个方法是在当前变换矩阵的基础上右乘参数矩阵构成复合矩阵。

除了上面五个方法外，还可以直接与另一个AffineTransform对象进行复合：  
 |- public void preConcatenate(AffineTransform Tx)；

|- public void concatenate(AffineTransform Tx)；

第一个方法是在当前变换矩阵的基础上左乘一个变换矩阵进行复合，第二个则是右乘。**而真正进行变换的时候，是从最右边的矩阵开始变换**。

### 2.5、透明度和合成规则

**合成规则**决定了绘制重叠图像的结果。通过选择合成规则，可以获得不同透明度的视觉效果。

要建立合成规则，需要用到α通道的概念。α通道可以视为颜色属性的一部分，用它来描述透明度。一个α值是介于0.0和1.0之间的数值，0.0表示完全透明，1.0表示完全不透明。

当两种颜色进行合成的时候，颜色α值表示该颜色在合成颜色中占的比例，合成颜色为最终显示的颜色。Porter-Duff规则用源颜色和目标颜色的线性组合定义合成颜色

α合 \*C合 = F源 \*α源\*C源 + F目\*α目\*C目

其中C代表颜色分量，F是系数，它的选取与合成规则有关。目前总共有12种Porter-Duff规则可用，它们在Java.awt.AlphaComposite类中定义了，这12个常量具体含义课通过文档了解。

对一个Graphics2D对象使用合成规则，调用：setComposite(合成规则)方法即可。

### 2.6、裁剪

**裁剪路径**定义了一个区域，只有在这个区域中绘制的图形对象才是可见的。

所有的Shape对象都可用来设置裁剪区域。下面这段代码将一个椭圆用作裁剪形状，并在椭圆上绘制一个矩形。

|  |
| --- |
| Graphics2D g2 = (Graphics2D) g;  Ellipse2D e1 = new Ellipse2D.Double(200, 200, 200, 200);  g2.setColor(Color.black); g2.fill(e1);  g2.setClip(e1);  Rectangle2D rect = new Rectangle2D.Double(220, 180, 160, 300);  g2.setStroke(new BasicStroke(10)); g2.setColor(Color.red); g2.draw(rect); |

只有在这个椭圆范围内的图形对象才能显示，就像这个矩形只能显示左右两条边的一部分。

Graphics2D类中还有一个方法改变裁剪区域：

void clip(Shape path)；

用路径去定义裁剪区域，这就需要我们前面所学的GeneralPath类了，利用这个类，我们可以自定义路径来定义裁剪区域。

### 2.7、文本和字体

在计算机图形学中，文本表示一类特殊的几何对象。文本字符串可以通过具有标准编码格式的字符序列紧凑地表示，编码格式有我们熟悉地ASCII码和Unicode等格式。字符的绘制由字体决定。

Java 2D提供了丰富地字体和文本操作功能。Font类表示一个字体对象，这个对象定义了一种字体，Graphics2D类中有setFont()方法设置字体以及drawString()方法绘制文字。

Font类：

**构造方法**：

|- public Font(String name, int style, int size)；第一个参数表示字体地名字，我们最常用的有“楷体”、“宋体”等；第二个参数表示字体的风格，在Font中定义了三个常量：PLAIN、ITALIC、BOLD，可以通过位运算“|”连在一起；最后一个参数表示字体的大小（size）

**普通方法**：

|- Font deriveFont(AffineTransform tx)；这个方法可以结合变换形成新的字体，例如我们需要斜着绘制文字的时候，可以先声明一个旋转变换，然后利用这个旋转变换生成新的字体，再绘制文字。

|- Rectangle2D getStringBounds(String str, FontRenderContext frc)；返回一段文字的边界矩形。

|- LineMetrics getLineMetrics(String str, FontRenderContext frc)；返回一段文字的细节性标度数据，利用LineMetrics类对象我们可以获取到字体的“基线”、“上升量”、“下降量”、“行间距”。有了这些属性我们能更科学的绘制文本，不至于让两个文本重叠。

上述两个方法中的参数FontRenderContext对象可以由Graphics2D对象的getFontRenderContext()方法获得。

Java 2D还提供了关于字体的相关操作的高级方法，将字体的字形作为Shape对象提取出来，这样一来字形进行复杂的处理和应用，就可以得到各种各样的视觉效果。Font类代表一种字体，GlyphVector类封装了一系列字形的几何描述，使用Font中的方法可以为某种字体的字符串获取一个GlyphVector对象：GlyphVector createGlyphVector(FontRenderContext frc, String str)；前面讲到第一个参数可由Graphics2D中的方法获得。一旦创建了GlyphVector对象，就可以获得字形的Shape类对象：

Shape getOutline()；

Shape getOutline(float x, float y)；

既然这样，文本也是可以作为裁剪区域的，我们直接看例子

**范例**：将字形作为裁剪区域

|  |
| --- |
| Graphics2D g2 = (Graphics2D) g;  String java = "Java";  Font font = new Font("宋体", Font.PLAIN, 144);  FontRenderContext frc = g2.getFontRenderContext();  GlyphVector gv = font.createGlyphVector(frc, java);  g2.setClip(gv.getOutline(200, 200));  g2.setColor(Color.red);  for(int i=0;i<2000;i++) { // 随机绘制2000个椭圆，只有裁剪区域的椭圆才会显示  Shape s = new Ellipse2D.Double(Math.random()\*500, Math.random()\*400, 30, 20);  g2.draw(s);  } |

我们先声明了一个字体对象font，然后获得了FontRenderContext对象，然后由这个font对象和FontRenderContext实例获得了一个GlyphVector对象，这个对象的getOutline(int x, int y)方法可以获得一个Shape对象，这个方法的两个参数表示在这个坐标处生成这个shape类对象，然后用setClip()方法即可将这个字符串作为裁剪区域了。

### 2.8、总结

这一节我们学到的东西有这样几个：  
 ·颜色、涂色

|- Color类

|- GradientPaint类（渐变涂色）

|- 循环涂色

|- 非循环涂色

|- TexturePaint类（纹理涂色）

·笔划

|- BasicStroke类

·变换（简单基本变换、复合变换）

|- AffineTransform类

·透明度和颜色合成规则

|- AlphaComposite类

·裁剪区域、裁剪路径

|- Shape类

|- GeneralPath类

·文本和字体

|- Font类

|- FontRenderContext类

|- GlyphVector类

在绘图中用上这些属性将会使显示效果更好，要多练习以熟悉这些类中的常用方法。

## 三、Java 2D高级话题

### 3.1、自定义基本图形类

我们目前为止可以绘制Java.awt.geom包中提供的各种简单形状，并且能将这些简单形状进行组合形成新的形状。但是这还不够，有一些复杂图形需要很多简单图形经过很复杂的组合才能实现，我们不能在每次需要就去组合一次，这样就显得代码重复。

我们之所以能绘制这些简单图形和组合图形，是因为它们都实现了Shape接口，看来关键之处在这个Shape接口，我们只要实现了Shape接口，那就相当于我们自己构建了一个基本图形类，在以后要用到的时候不必重新编写。

先看看Shape接口中的方法：  
 |- public boolean contains(Rectangle2D r)；

|- public boolean contains(double x, double y, double w, double h)；

|- public boolean contains(Point2D p)；

|- public boolean contains(double x, double y)；

|- public Rectangle2D getBounds2D()；

|- public Rectangle getBounds()；

|- public boolean intersects(Rectangle2D r)；

|- public boolean intersects(double x, double y, double w, double h)；

|- public PathIterator getPathIterator(AffineTransform at)；

|- public PathIterator getPathIterator(AffineTransform at, double flatness)；

前四个contains方法用来判断一个点或一个矩形是否全部在该图形内部；中间两个getBounds方法用来获取该图形的边界矩形；intersects方法测试相交的情况；最后两个方法返回一个PathIterator对象，该对象描述了图形的边界路径。

似乎自己实现这10个方法有点麻烦，而我们不是有现成的GeneralPath类吗？似乎我们继承这个类就可以自己定义基本图形类了，因为这个类也实现了Shape接口。是这样吗？并不是，GeneralPath这个类在声明的时候用final修饰了，是不能作为其他类的父类的。那怎么办呢？还有一招，我们在自己定义的类中声明一个GeneralPath对象，这样就可以很容易的重新实现这10个方法了。

**范例：**自己定义一个三角形的类

|  |
| --- |
| package fuhao.demo;  import java.awt.Rectangle;  import java.awt.Shape;  import java.awt.geom.AffineTransform;  import java.awt.geom.GeneralPath;  import java.awt.geom.PathIterator;  import java.awt.geom.Point2D;  import java.awt.geom.Rectangle2D;  public class sanjiaoxing implements Shape{  private GeneralPath path;    public sanjiaoxing(double x1, double y1, double x2, double y2, double x3, double y3) {  path = new GeneralPath();  path.moveTo(x1, y1);  path.lineTo(x2, y2);  path.lineTo(x3, y3);  path.closePath();  path.setWindingRule(GeneralPath.WIND\_NON\_ZERO);  }  @Override  public Rectangle getBounds() {  return this.path.getBounds();  }  @Override  public Rectangle2D getBounds2D() {  return this.path.getBounds2D();  }  @Override  public boolean contains(double x, double y) {  return this.path.contains(x, y);  }  @Override  public boolean contains(Point2D p) {  return this.path.contains(p);  }  @Override  public boolean intersects(double x, double y, double w, double h) {  return this.path.intersects(x, y, w, h);  }  @Override  public boolean intersects(Rectangle2D r) {  return this.path.intersects(r);  }  @Override  public boolean contains(double x, double y, double w, double h) {  return this.contains(x, y, w, h);  }  @Override  public boolean contains(Rectangle2D r) {  return this.path.contains(r);  }  @Override  public PathIterator getPathIterator(AffineTransform at) {  return this.path.getPathIterator(at);  }  @Override  public PathIterator getPathIterator(AffineTransform at, double flatness) {  return this.path.getPathIterator(at, flatness);  }  } |

看出来了吧，很简单，我们声明一个GeneralPath类对象，然后将我们自己定义的基本图形的路径定义出来，然后利用这个GeneralPath类对象就可以轻松的实现Shape接口的10个方法。

自定义基本图形类的时候唯一需要自己做的事就是，想好用哪几个点来确定这个图形，这是最重要的，这里用到的就是三角形的三个顶点坐标。

### 3.2、数字图像处理

Java中图像读取和处理的方式很多，也提供了很多类，我们只讲Java 2D中处理图像提供的支持。

Java 2D中通常使用BufferedImage类来表示图像。一个BufferedImage对象可由它的构造方法来实例化。

**|- public BufferedImage(int width, int height, int imageType)；**创建一个大小为width\*height的空图像。

还可以利用ImageIO类的静态方法获得一个图像对象

**|- public static Image read(File file)；**

得到BufferedImage对象后，我们就可以在这个图像上绘图了，在图像上绘图我们也需要得到一个Graphics2D类对象，这个在BufferedImage类中有方法来获取

**|- public Graphics createGraphics()；**

得到一个Graphics对象后，可以强制类型装换为Graphics2D对象，得到这个对象之后就可以在image对象上绘制图形了，和上一章中讲的绘制图形所用到的方法是一样的，只不过现在的画布不是窗口了，而是图像；我们不仅可以在图像上绘制几何图形，还可以绘制其他的图像，这就要用到Graphics2D中的drawImage()方法了

**|- public boolean drawImage(Image img, int x, int y, ImageObserver observer)；**

第一个参数是要绘制到图像对象上的另一个图像，第二个和第三个参数表示将这个图像绘制在目标图像的哪个坐标处，第四个参数设置为“new Component(){}”这个匿名内部类即可。

在绘制完图像之后，我们可以将其保存为一个外部图像文件，利用ImageIO中的另一个静态方法

**|- public static boolean write(BufferedImage img, String formatName, File file)；**

第一个参数表示要保存的图像对象，第二个参数表示保存图像的文件格式，如“png”，第三个参数表示保存的目标外部文件。

**范例：**创建一个空图像，在图像上画一个矩形，从外部读取一个图像文件，绘制到空图像上，然后保存图像到外部文件

|  |
| --- |
| BufferedImage img = new BufferedImage(1700, 1060, BufferedImage.TYPE\_INT\_RGB);  Graphics2D g2 = (Graphics2D)(img.createGraphics());  g2.setColor(Color.red); g2.fill(new Rectangle2D.Double(0, 0, 100, 100));  try {  BufferedImage writedImg = ImageIO.read(new File("C:\\Users\\fuhao\\Desktop\\纹理.png"));  g2.drawImage(writedImg, 100, 100, new Component() {});  ImageIO.write(img, "png", new File("C:\\Users\\fuhao\\Desktop\\test.png"));  } catch (IOException e1) {  e1.printStackTrace();  } |

### 3.3、动画

动画即动态的图画，它给人建立起一种动态的视觉效果。动画产生一系列图像（帧），这些图像显示了场景中的变化。当这些帧以某一个速率被连续的显示出来的时候（如每秒60帧），我们可以感受到场景中的连贯运动，而不是离散的图像。动画在图形模型中加入了时间维度，不同时间点上的每一帧，本质上是一幅普通的静止的图像。但是帧的内容会随时间改变，更高的帧速率表现出更平滑的动画效果，但是一个动画的帧速率受到图像绘制系统能力的限制。

在Java中，实现动画通常需要另外的线程来处理与时间相关的变化。但是Swing组件并不是线程安全的，我们不能在其他线程中用getGraphics方法来描绘图形。

在Swing组件上创建动画的合适方法是，讲图形绘制与图形变化分离开。绘图的代码只放在Swing组件的重绘方法中，而图形的变化也就是动画逻辑放到其他线程中处理。

一个典型的多线程动画框架如下所示：

|  |
| --- |
| public void paintComponent(Graphics g){  <绘制图形的代码>  }  public void run(){  while(true){  <更新图形数据>  repaint();  Sleep(time);  }  } |

paintComponent方法是绘制某一帧的代码，run方法中会对帧进行修改然后调用重绘方法将修改后的帧绘制出来。

**范例：**模拟下雨场景，许多垂直的线段在往下移动，线段的位置和长度随机

|  |
| --- |
| package fuhao.demo;  import java.awt.Dimension;  import java.awt.Graphics;  import java.awt.Graphics2D;  import java.awt.Shape;  import java.awt.geom.AffineTransform;  import java.awt.geom.Line2D;  import java.awt.geom.Rectangle2D;  import java.util.LinkedList;  import javax.swing.JFrame;  import javax.swing.JPanel;  public class Demo extends JPanel {  private LinkedList<Line2D> list = new LinkedList<>();    public Demo() {  this.setPreferredSize(new Dimension(700, 700));  new Thread(new Runnable() {  @Override  public void run() {  while(true) {  LinkedList<Integer> index = new LinkedList<>();    for(int i=0;i<Demo.this.list.size();i++) {  if(Demo.this.list.get(i).getY2() > 700) {  index.add(i);  } else {  Line2D line = Demo.this.list.get(i);  line.setLine(line.getX1(), line.getY1()+1, line.getX2(), line.getY2()+1);  }  }  for(int i=0;i<index.size();i++) {  Demo.this.list.remove((int)index.get(i));  }    int x = (int) (Math.random()\*700);  int y = 0;  int length = (int)(Math.random()\*20+5);  Line2D line = new Line2D.Double(x, y, x, y+length);  Demo.this.list.add(line);    try {  Thread.sleep(100);  Demo.this.repaint();  } catch (InterruptedException e) {  e.printStackTrace();  }  }  }  }).start();  }  @Override  public void paintComponent(Graphics g) {  super.paintComponent(g);  Graphics2D g2 = (Graphics2D)g;    for(int i=0;i<list.size();i++) {  g2.draw(list.get(i));  }  }    public static void main(String[] args) {  Demo demo = new Demo();  JFrame frame = new JFrame();  frame.setContentPane(demo);  frame.setDefaultCloseOperation(JFrame.EXIT\_ON\_CLOSE);  frame.setSize(frame.getPreferredSize());  frame.setVisible(true);  }  } |

创建了一个链表保存所有的线段，在绘图部分（paintComponent）将这个链表中保存的所有线段绘制出来，在动画逻辑部分（run）生成一条新的线段，更新线段的坐标，如果线段已经超出屏幕，将其从链表中删除，由于这是一个死循环，所以可以一直进行下去，而我们看到的就是许多线段在往下落这种动画效果。

除了创建新的线程，还可以使用Swing的Timer类。一个Timer类对象会以事先设置好的速率周期产生一个动作事件，该事件会触发它的监听器，监听器可以执行帧的更新操作。

Timer的构造方法：

|- Timer(int delay, ActionListener listener)；

第一个参数表示隔多久产生一个动作事件，第二个参数为监听动作事件的监听器。在实例化Timer类对象后，调用它的start方法就可以开始执行“产生事件-监听事件”的循环过程了。

对于动画，Timer类提供了比直接创建线程更为方便的方法。

**范例：**制作一个时钟，实时显示当前时间

|  |
| --- |
| package fuhao.demo;  import java.awt.BasicStroke;  import java.awt.Color;  import java.awt.Dimension;  import java.awt.GradientPaint;  import java.awt.Graphics;  import java.awt.Graphics2D;  import java.awt.Shape;  import java.awt.event.ActionEvent;  import java.awt.event.ActionListener;  import java.awt.geom.AffineTransform;  import java.awt.geom.Ellipse2D;  import java.awt.geom.Line2D;  import java.util.Calendar;  import javax.swing.JFrame;  import javax.swing.JPanel;  import javax.swing.Timer;  public class Demo extends JPanel {  private int hour;  private int minute = 15;  private int second = 40;    public Demo() {  this.hour = Calendar.getInstance().get(Calendar.HOUR);  this.minute = Calendar.getInstance().get(Calendar.MINUTE);  this.second = Calendar.getInstance().get(Calendar.SECOND);    this.setPreferredSize(new Dimension(700, 700));  Timer timer = new Timer(1000, new ActionListener() {  @Override  public void actionPerformed(ActionEvent e) {  Demo.this.second ++;  if(Demo.this.second == 60) {  Demo.this.second = 0;  Demo.this.minute++;  if(Demo.this.minute == 60) {  Demo.this.minute = 0;  Demo.this.hour ++;  if(Demo.this.hour == 12) {  Demo.this.hour = 0;  }  }  }  Demo.this.repaint();  }  });  timer.start();  }  @Override  public void paintComponent(Graphics g) {  super.paintComponent(g);  Graphics2D g2 = (Graphics2D)g;  this.setBackground(Color.WHITE);    // 绘制表盘  Ellipse2D circle = new Ellipse2D.Double(200, 200, 300, 300);  GradientPaint gp = new GradientPaint(200, 200, Color.WHITE, 500, 500, Color.GRAY);  g2.setPaint(gp); g2.fill(circle);    Shape line = new Line2D.Double(350, 205, 350, 225);  AffineTransform tx = new AffineTransform();  tx.setToRotation(Math.PI/6, 350, 350);  g2.setPaint(null);  g2.setColor(Color.BLACK);  g2.setStroke(new BasicStroke(3));  for(int i=0;i<12;i++) {  g2.draw(line);  line = tx.createTransformedShape(line);  }    // 绘制指针  Shape hourLine = new Line2D.Double(350, 350, 350, 280);  Shape minuteLine = new Line2D.Double(350, 350, 350, 260);  Shape secondLine = new Line2D.Double(350, 350, 350, 240);    AffineTransform hourTx = new AffineTransform();  AffineTransform minuteTx = new AffineTransform();  AffineTransform secondTx = new AffineTransform();    secondTx.setToRotation(Math.PI\*2/60\*this.second, 350, 350);  minuteTx.setToRotation(Math.PI\*2/60\*this.minute+Math.PI\*2/3600\*this.second, 350, 350);  hourTx.setToRotation(Math.PI\*2/12\*this.hour + Math.PI\*2/12/60\*this.minute + Math.PI\*2/12/3600\*this.second, 350, 350);    g2.setStroke(new BasicStroke(5)); g2.draw(hourTx.createTransformedShape(hourLine));  g2.setStroke(new BasicStroke(3)); g2.draw(minuteTx.createTransformedShape(minuteLine));  g2.setStroke(new BasicStroke(1)); g2.draw(secondTx.createTransformedShape(secondLine));  }    public static void main(String[] args) {  Demo demo = new Demo();  JFrame frame = new JFrame();  frame.setContentPane(demo);  frame.setDefaultCloseOperation(JFrame.EXIT\_ON\_CLOSE);  frame.setSize(frame.getPreferredSize());  frame.setVisible(true);  }  } |

我们将Timer的周期设置为1秒，每过1秒产生一个动作事件，而我们正好需要每过1秒更新时钟，在监听事件中写好逻辑，再到重绘方法中绘制即可。

**题外话**：细胞自动机

所谓细胞自动机，是一个简单的网格迭代系统，这个系统通过一定的固定的规则集进行进化，每个细胞只有两种状态：生或死，一个细胞有八个邻居。系统的迭代基于系统的上一个状态为每个细胞分配下一个状态，每个细胞都遵循相同的规则，且新的状态仅仅取决于该细胞的邻居们的当前状态。

细胞自动机的一个著名的例子就是生命游戏，它的规则是：

·如果一个死亡的细胞有三个活着的邻居，它就变活了；

·如果一个活着的细胞有两个或三个活的邻居，它就仍然活着；

·其他情况下，该细胞死亡。

**范例：**

定义细胞

|  |
| --- |
| package fuhao.demo;  import java.awt.Point;  public class Cell {  public static final int LEFT\_UP = 0;  public static final int UP = 1;  public static final int RIGHT\_UP = 2;  public static final int LEFT = 3;  public static final int RIGHT = 4;  public static final int LEFT\_DOWN = 5;  public static final int DOWN = 6;  public static final int RIGHT\_DOWN = 7;  public static final Point OUT = new Point(-1, -1);    private boolean color;  private Point point;  private Point[] other;    public Cell() {  this.color = false;  this.other = new Point[] {OUT, OUT, OUT, OUT, OUT, OUT, OUT, OUT};  }    public void setColor(boolean color) {  this.color = color;  }    public boolean isColor() {  return color;  }    public void setPoint(Point point) {  this.point = point;  }    public Point getPoint() {  return point;  }    public void setOther(int index, Point other) {  this.other[index] = other;  }    public Point getOther(int index) {  return other[index];  }  } |

编写细胞自动机

|  |
| --- |
| package fuhao.demo;  import java.awt.Color;  import java.awt.Dimension;  import java.awt.Graphics;  import java.awt.Graphics2D;  import java.awt.Point;  import java.awt.event.ActionEvent;  import java.awt.event.ActionListener;  import java.awt.geom.Ellipse2D;  import java.awt.geom.Line2D;  import java.util.Hashtable;  import javax.swing.JFrame;  import javax.swing.JPanel;  import javax.swing.Timer;  public class Demo extends JPanel {  private Hashtable<Point, Cell> hash = new Hashtable<>();    public Demo() {  this.setPreferredSize(new Dimension(800, 800));  Timer timer = new Timer(1000, new ActionListener() {  @Override  public void actionPerformed(ActionEvent e) {  Demo.this.game();  Demo.this.repaint();  }  });  timer.start();    for(int i=0;i<10;i++) {  for(int j=0;j<10;j++) {  Point p = new Point(i, j);  Cell c = new Cell();  c.setPoint(new Point(i\*70+10, j\*70+10));  double rand = Math.random()\*10;  if(rand <= 2) {  c.setColor(true);  }  c.setOther(Cell.LEFT\_UP, new Point(i-1, j-1));  c.setOther(Cell.UP, new Point(i, j-1));  c.setOther(Cell.RIGHT\_UP, new Point(i+1, j-1));  c.setOther(Cell.LEFT, new Point(i-1, j));  c.setOther(Cell.RIGHT, new Point(i+1, j));  c.setOther(Cell.LEFT\_DOWN, new Point(i-1, j+1));  c.setOther(Cell.DOWN, new Point(i, j+1));  c.setOther(Cell.RIGHT\_DOWN, new Point(i+1, j+1));    if(i==0) {  c.setOther(Cell.LEFT\_UP, Cell.OUT);  c.setOther(Cell.LEFT, Cell.OUT);  c.setOther(Cell.LEFT\_DOWN, Cell.OUT);  }  if(i==9) {  c.setOther(Cell.RIGHT\_UP, Cell.OUT);  c.setOther(Cell.RIGHT, Cell.OUT);  c.setOther(Cell.RIGHT\_DOWN, Cell.OUT);  }  if(j==0) {  c.setOther(Cell.LEFT\_UP, Cell.OUT);  c.setOther(Cell.UP, Cell.OUT);  c.setOther(Cell.RIGHT\_UP, Cell.OUT);  }  if(j==9) {  c.setOther(Cell.LEFT\_DOWN, Cell.OUT);  c.setOther(Cell.DOWN, Cell.OUT);  c.setOther(Cell.RIGHT\_DOWN, Cell.OUT);  }  this.hash.put(p, c);  }  }  }  @Override  public void paintComponent(Graphics g) {  super.paintComponent(g);  Graphics2D g2 = (Graphics2D)g;  this.setBackground(Color.WHITE);    for(int i=0;i<=10;i++) {  Line2D line = new Line2D.Double(0, i\*70, 700, i\*70);  g2.setColor(Color.BLACK); g2.draw(line);  }  for(int i=0;i<=10;i++) {  Line2D line = new Line2D.Double(i\*70, 0, i\*70, 700);  g2.setColor(Color.BLACK); g2.draw(line);  }    for(int i=0;i<10;i++) {  for(int j=0;j<10;j++) {  Cell c = this.hash.get(new Point(i, j));  if(c.isColor()) {  g2.setColor(Color.black);  g2.fill(new Ellipse2D.Double(c.getPoint().getX(), c.getPoint().getY(), 50, 50));  }  }  }  }    private void game() {  for(int i=0;i<10;i++) {  for(int j=0;j<10;j++) {  Cell c = this.hash.get(new Point(i, j));  int num = 0;  for(int k=0;k<8;k++) {  if(c.getOther(k) != Cell.OUT && this.hash.get(c.getOther(k)).isColor()) {  num++;  }  }  if(!c.isColor() && num==3) {  c.setColor(true);  } else if(c.isColor() && (num==2 || num==3)) {  c.setColor(true);  } else {  c.setColor(false);  }  }  }  }    public static void main(String[] args) {  Demo demo = new Demo();  JFrame frame = new JFrame();  frame.setContentPane(demo);  frame.setDefaultCloseOperation(JFrame.EXIT\_ON\_CLOSE);  frame.setSize(frame.getPreferredSize());  frame.setVisible(true);  }  } |

我们随机初始化了一些活着的细胞，然后按照这个规则一直更新。