18.2 开发游戏界面

连连看的游戏界面十分简单，大致上可分为两个区域：

- 游戏主界面区。
- 控制按钮与数据显示区。

18.2.1 开发界面布局

本程序将会使用一个 RelativeLayout 作为整体的界面布局元素，界面布局的上面是一个自定义组件，下面是一个水平排列的 LinearLayout。

程序清单：codes\18\Link\res\layout\main.xml

```
<?xml version="1.0" encoding="utf-8"?>
<RelativeLayout
      xmlns:android="http://schemas.android.com/apk/res/android"
      android:layout_width="fill_parent"
      android:layout_height="fill_parent"
      android:background="@drawable/room">
<!-- 游戏主界面的自定义组件 -->
<org.crazyit.link.view.GameView
      android:id="@+id/gameView"
      android:layout_width="fill_parent"
      android:layout_height="fill_parent" />
<!-- 水平排列的 LinearLayout -->
<LinearLayout
      android:layout_width="fill_parent"
      android:layout_height="fill_parent"
      android:orientation="horizontal"
      android:layout_marginTop="380px"
      android:background="#1e72bb"
      android:gravity="center">
<!-- 控制游戏开始的按钮 -->
<Button
      android:id="@+id/startButton"
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:background="@drawable/button_selector" />
<!-- 显示游戏剩余时间的文本框 -->
<TextView
      android:id="@+id/timeText"
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:gravity="center"
      android:textSize="20dip"
      android:width="150px"
      android:textColor="#ff9" />
</LinearLayout>
</RelativeLayout>
```

这个界面布局很简单，指定按钮的背景色时使用了@drawable/button_selector，这是一个

在 res\drawable 目录下配置的 StateListDrawable 对象，配置文件代码如下。

程序清单：codes\18\Link\res\drawable-mdpi\button_selector.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<selector xmlns:android="http://schemas.android.com/apk/res/android">
      <!-- 指定按钮按下时的图片 -->
      <item android:state_pressed="true"
            android:drawable="@drawable/start_down"
      />
      <!-- 指定按钮松开时的图片 -->
      <item android:state_pressed="false"
            android:drawable="@drawable/start"
      />
</selector>
```

其中 GameView 只是一个 View 的普通子类，开发了上面的界面布局文件之后，运行该程序将可以看到如图 18.3 所示的界面。

➤➤18.2.2 开发游戏界面组件

本游戏的界面组件采用了一个自定义 View：GameView，它从 View 基类派生而出，这个自定义 View 的功能就是根据游戏状态来绘制游戏界面上的全部方块。

为了开发这个 GameView，本程序还提供了一个 Piece 类，一个 Piece 对象代表游戏界面上的一个方块，它除了封装方块上的图片之外，还需要封装该方块代表二维数组中的哪个元素；也需要封装它的左上角在游戏界面中 *X*、*Y* 坐标。图 18.4 示意了方块左上角的 *X*、*Y* 坐标的作用。

图 18.3 游戏布局

图 18.4 方块的左上角

方块左上角的 *X*、*Y* 坐标可决定它的绘制位置，GameView 根据这两个坐标值绘制全部方块即可。下面是该程序中 Piece 类的代码。

程序清单：codes\18\Link\src\org\crazyit\link\view\Piece.java

```
public class Piece
{
      // 保存方块对象的所对应的图片
```

```
        private PieceImage image;
        // 该方块的左上角的 x 坐标
        private int beginX;
        // 该方块的左上角的 y 坐标
        private int beginY;
        // 该对象在 Piece[][]数组中第一维的索引值
        private int indexX;
        // 该对象在 Piece[][]数组中第二维的索引值
        private int indexY;
        // 只设置该 Piece 对象在棋盘数组中的位置
        public Piece(int indexX , int indexY)
        {
              this.indexX = indexX;
              this.indexY = indexY;
        }
        public int getBeginX()
        {
              return beginX;
        }
        public void setBeginX(int beginX)
        {
              this.beginX = beginX;
        }
        // 下面省略了各属性的 setter 和 getter 方法
        ...
        // 判断两个 Piece 上的图片是否相同
        public boolean isSameImage(Piece other)
        {
              if (image == null)
              {
                    if (other.image != null)
                          return false;
              }
              // 只要 Piece 封装图片 ID 相同，即可认为两个 Piece 相等
              return image.getImageId() == other.image.getImageId();
        }
  }
```

上面的Piece类中封装的PieceImage代表了该方块上的图片，但此处并未直接使用Bitmap对象来代表方块上的图片——因为我们需要使用 PieceImage 来封装两个信息：

- Bitmap 对象。
- 图片资源的 ID。

其中 Bitmap 对象用于在游戏界面上绘制方块；而图片资源的 ID 则代表了该 Piece 对象的标识，当两个 Piece 所封装的图片资源的 ID 相等时，即可认为这两个 Piece 上的图片相同。如以上程序中粗体字代码所示。

下面是 PieceImage 类的代码。

程序清单：codes\18\Link\src\org\crazyit\link\view\PieceImage.java

```
  public class PieceImage
  {
        private Bitmap image;
        private int imageId;
        // 有参数的构造器
```

```
        public PieceImage(Bitmap image, int imageId)
        {
              super();
              this.image = image;
              this.imageId = imageId;
        }
        // 省略了各属性的 setter 和 getter 方法
        ...
  }
```

GameView 主要就是根据游戏的状态数据来绘制界面上的方块，GameView 继承了 View 组件，重写了 View 组件上 onDraw(Canvas canvas)方法，重写该方法主要就是绘制游戏里剩余的方块；除此之外，它还会负责绘制连接方块的连接线。

GamaView 的代码如下。

程序清单：codes\18\Link\src\org\crazyit\link\view\GameView.java

```
public class GameView extends View
{
      // 游戏逻辑的实现类
      private GameService gameService;          //①
      // 保存当前已经被选中的方块
      private Piece selectedPiece;
      // 连接信息对象
      private LinkInfo linkInfo;
      private Paint paint;
      // 选中标识的图片对象
      private Bitmap selectImage;
      public GameView(Context context, AttributeSet attrs)
      {
            super(context, attrs);
            this.paint = new Paint();
            // 设置连接线的颜色
            this.paint.setColor(Color.RED);
            // 设置连接线的粗细
            this.paint.setStrokeWidth(3);
            this.selectImage = ImageUtil.getSelectImage(context);
      }
      public void setLinkInfo(LinkInfo linkInfo)
      {
            this.linkInfo = linkInfo;
      }
      public void setGameService(GameService gameService)
      {
            this.gameService = gameService;
      }
      @Override
      protected void onDraw(Canvas canvas)
      {
            super.onDraw(canvas);
            if (this.gameService == null)
                  return;
            Piece[][] pieces = gameService.getPieces();          //②
            if (pieces != null)
            {
                  // 遍历 pieces 二维数组
```

```
                for (int i = 0; i < pieces.length; i++)
                {
                    for (int j = 0; j < pieces[i].length; j++)
                    {
                        // 如果二维数组中该元素不为空（即有方块），将这个方块的图片画出来
                        if (pieces[i][j] != null)
                        {
                            // 得到这个 Piece 对象
                            Piece piece = pieces[i][j];
                            // 根据方块左上角 X、Y 坐标绘制方块
                            canvas.drawBitmap(piece.getImage().getImage(),
                                piece.getBeginX(), piece.getBeginY(), null);
                        }
                    }
                }
            }
            // 如果当前对象中有 linkInfo 对象，即连接信息
            if (this.linkInfo != null)
            {
                // 绘制连接线
                drawLine(this.linkInfo, canvas);
                // 处理完后清空 linkInfo 对象
                this.linkInfo = null;
            }
            // 画选中标识的图片
            if (this.selectedPiece != null)
            {
                canvas.drawBitmap(this.selectImage, this.selectedPiece.
                getBeginX(),
                    this.selectedPiece.getBeginY(), null);
            }
        }
        // 根据 LinkInfo 绘制连接线的方法
        private void drawLine(LinkInfo linkInfo, Canvas canvas)
        {
            // 获取 LinkInfo 中封装的所有连接点
            List<Point> points = linkInfo.getLinkPoints();
            // 依次遍历 linkInfo 中的每个连接点
            for (int i = 0; i < points.size() - 1; i++)
            {
                // 获取当前连接点与下一个连接点
                Point currentPoint = points.get(i);
                Point nextPoint = points.get(i + 1);
                // 绘制连线
                canvas.drawLine(currentPoint.x, currentPoint.y,
                    nextPoint.x, nextPoint.y, this.paint);
            }
        }
        // 设置当前选中方块的方法
        public void setSelectedPiece(Piece piece)
        {
            this.selectedPiece = piece;
        }
        // 开始游戏方法
        public void startGame()
        {
```

```
            this.gameService.start();
            this.postInvalidate();
        }
    }
```

上面的 GameView 中第一段粗体字代码用于根据游戏的状态数据来绘制界面中的所有方块，第二段粗体字代码则用于根据 LinkInfo 来绘制两个方块之间的连接线。

上面的程序中①号代码处定义了 GameService 对象，②号代码则调用了 GameService 的 getPieces()方法来获取游戏中剩余的方块，GameService 是游戏的业务逻辑实现类。后面会详细介绍该类的实现，此处暂不讲解。

➤➤18.2.3 处理方块之间的连接线

LinkInfo 是一个非常简单的工具类，它用于封装两个方块之间的连接信息——其实就是封装一个 List，List 里保存了连接线需要经过的点。

在实现 LinkInfo 对象之前，先来分析两个方块可以相连的情形。连连看游戏的规则约定：两个方块之间最多只能用 3 条线段相连，也就是说最多只能有 2 个“拐点”，加上两个方块的中心，方块的连接信息最多只需要 4 个连接点。图 18.5 显示了允许出现的连接情况。

图 18.5 方块的连接

考虑到 LinkInfo 最多需要封装 4 个连接点，最少需要封装 2 个连接点，因此程序定义如下 LinkInfo 类。

程序清单：codes\18\Link\src\org\crazyit\link\object\LinkInfo.java

```
public class LinkInfo
{
    // 创建一个集合用于保存连接点
    private List<Point> points = new ArrayList<Point>();
    // 提供第一个构造器，表示两个 Point 可以直接相连，没有转折点
    public LinkInfo(Point p1, Point p2)
    {
        // 加到集合中去
        points.add(p1);
        points.add(p2);
    }
    // 提供第二个构造器，表示三个 Point 可以相连，p2 是 p1 与 p3 之间的转折点
    public LinkInfo(Point p1, Point p2, Point p3)
    {
        points.add(p1);
        points.add(p2);
        points.add(p3);
    }
    // 提供第三个构造器，表示四个 Point 可以相连，p2，p3 是 p1 与 p4 的转折点
    public LinkInfo(Point p1, Point p2, Point p3, Point p4)
    {
        points.add(p1);
        points.add(p2);
        points.add(p3);
```

```
            points.add(p4);
        }
        // 返回连接集合
        public List<Point> getLinkPoints()
        {
            return points;
        }
    }
```

LinkInfo 中所用的 Point 代表一个点，程序直接使用了 android.graphics.Point 类，每个 Point 封装了该点的 *X*、*Y* 坐标。

18.3 连连看的状态数据模型

对于游戏玩家而言，游戏界面上看到“元素”千差万别、变化多端；但对于游戏开发者而言，游戏界面上的元素在底层都是一些数据，不同数据所绘制的图片有差异而已。因此建立游戏的状态数据模型是实现游戏逻辑的重要步骤。

➤➤18.3.1 定义数据模型

连连看的游戏界面是一个 $N \times M$ 的“网格”，每个网格上显示一张图片。但对于游戏开发者来说，这个网格只需要用一个二维数据来定义即可，而每个网格上所显示的图片，对于底层的数据模型来说，不同的图片对应于不同的数值即可。图 18.6 显示了数据模型的示意。

0	1	2					
	1	1					
		3					

图 18.6 连连看的数据模型

对于图 18.6 所示的数据模型，只要让数值为 0 的网络上不绘制图片，其他数值的网格则绘制相应的图片，就可显示出连连看的游戏界面了。

本程序实际上并不是直接使用 int[][]数组来保存游戏的状态数据，而是采用 Piece[][]来保存游戏的状态模型——因为 Piece 对象封装的信息更多，不仅包含了该方块的左上角的 *X*、*Y* 坐标，而且还包含了该 Piece 所显示的图片、图片 ID——这个图片 ID 就可作为该 Piece 的数据。

➤➤18.3.2 初始化游戏状态数据

为了初始化游戏状态，程序需要创建一个 Piece[][]数组，为此程序定义一个 AbstractBoard 抽象类，该抽象类的代码如下。

程序清单：codes\18\Link\src\org\crazyit\link\board\AbstractBoard.java

```
public abstract class AbstractBoard
{
    // 定义一个抽象方法，让子类去实现
    protected abstract List<Piece> createPieces(GameConf config,
        Piece[][] pieces);
    public Piece[][] create(GameConf config)
    {
        // 创建 Piece[][]数组
        Piece[][] pieces = new Piece[config.getXSize()][config.getYSize()];
        // 返回非空的 Piece 集合，该集合由子类去创建
        List<Piece> notNullPieces = createPieces(config, pieces);     //①
        // 根据非空 Piece 对象的集合的大小来取图片
        List<PieceImage> playImages = ImageUtil.getPlayImages(config.get
        Context(),
            notNullPieces.size());
        // 所有图片的宽、高都是相同的
        int imageWidth = playImages.get(0).getImage().getWidth();
        int imageHeight = playImages.get(0).getImage().getHeight();
        // 遍历非空的 Piece 集合
        for (int i = 0; i < notNullPieces.size(); i++)
        {
            // 依次获取每个 Piece 对象
            Piece piece = notNullPieces.get(i);
            piece.setImage(playImages.get(i));
            // 计算每个方块左上角的 X、Y 坐标
            piece.setBeginX(piece.getIndexX() * imageWidth
                + config.getBeginImageX());
            piece.setBeginY(piece.getIndexY() * imageHeight
                + config.getBeginImageY());
            // 将该方块对象放入方块数组的相应位置处
            pieces[piece.getIndexX()][piece.getIndexY()] = piece;
        }
        return pieces;
    }
}
```

上面的程序中粗体字代码块用于初始化 Piece[][]数组，初始化代码负责为各非空的 Piece 元素的 beginX、beginY、image 属性赋值，其中 beginX、beginY 根据该方块在二维数组中的位置动态计算得到。

上面的程序中①号代码调用了 createPieces(config, pieces)抽象方法来创建一个 List<Piece>集合，该抽象方法将会交给其子类去实现，这里是典型的“模板模式”的应用。AbstractBoard 抽象基类完全可以根据 Piece 对象在二维数组中的位置动态地计算它的 beginX、beginY，但 AbstractBoard 不确定 Piece[][]数组的哪些元素是非空的。

由于连连看游戏的初始状态可能有很多种——比如横向分布的方块、竖向分布的方块、矩阵排列的方块、随机分布的方块等，该程序为了考虑以后的扩展性，此处只是采用了模板模式：定义 AbstractBoard 抽象基类来完成通用的代码，而暂时无法确定、需要子类实现的方法定义成 createPieces(GameConf config, Piece[][] pieces)抽象方法。

上面的程序中还用到了一个 ImageUtil 工具类，它的作用是自动搜寻/res/drawable-mdpi 目录下的图片，并根据需要随机地读取该目录下的图片。后面会详细介绍该工具类的用法。

下面为该 AbstractBoard 实现 3 个子类。

1．矩阵排列的方块

矩阵排列的方块会填满二维数组的每个数组元素，只是把四周留空即可，该子类的代码如下。

程序清单：codes\18\Link\src\org\crazyit\link\board\impl\FullBoard.java

```
public class FullBoard extends AbstractBoard
{
	@Override
	protected List<Piece> createPieces(GameConf config,
		Piece[][] pieces)
	{
		// 创建一个 Piece 集合，该集合里面存放初始化游戏时所需的 Piece 对象
		List<Piece> notNullPieces = new ArrayList<Piece>();
		for (int i = 1; i < pieces.length - 1; i++)
		{
			for (int j = 1; j < pieces[i].length - 1; j++)
			{
				// 先构造一个 Piece 对象，只设置它在 Piece[][]数组中的索引值
				// 所需要的 PieceImage 由其父类负责设置
				Piece piece = new Piece(i, j);
				// 添加到 Piece 集合中
				notNullPieces.add(piece);
			}
		}
		return notNullPieces;
	}
}
```

该子类初始化的游戏界面如图 18.7 所示。

图 18.7　矩阵排列的方块

2．竖向排列的方块

竖向排列的方块以垂直的空列分隔开，该子类的代码如下。

程序清单：codes\18\Link\src\org\crazyit\link\board\impl\VerticalBoard. java

```
public class VerticalBoard extends AbstractBoard
{
      protected List<Piece> createPieces(GameConf config,
            Piece[][] pieces)
      {
            // 创建一个 Piece 集合，该集合里面存放初始化游戏时所需的 Piece 对象
            List<Piece> notNullPieces = new ArrayList<Piece>();
            for (int i = 0; i < pieces.length; i++)
            {
                  for (int j = 0; j < pieces[i].length; j++)
                  {
                        // 加入判断，符合一定条件才去构造 Piece 对象，并加到集合中
                        if (i % 2 == 0)
                        {
                              // 如果 x 能被 2 整除，即单数列不会创建方块
                              // 先构造一个 Piece 对象，只设置它在 Piece[][]数组中的索引值
                              // 所需要的 PieceImage 由其父类负责设置
                              Piece piece = new Piece(i, j);
                              // 添加到 Piece 集合中
                              notNullPieces.add(piece);
                        }
                  }
            }
            return notNullPieces;
      }
}
```

上面的程序中粗体字代码控制了只设置 i % 2 == 0 的列，也就是只设置索引为偶数的列，该子类初始化的游戏界面如图 18.8 所示。

图 18.8　竖向排列的方块

3．横向排列的方块

竖向排列的方块以水平的空行分隔开，该子类的代码如下。

程序清单：codes\18\Link\src\org\crazyit\link\board\impl\HorizontalBoard.java

```
public class HorizontalBoard extends AbstractBoard
{
```

```
        protected List<Piece> createPieces(GameConf config,
                Piece[][] pieces)
        {
                // 创建一个 Piece 集合，该集合里面存放初始化游戏时所需的 Piece 对象
                List<Piece> notNullPieces = new ArrayList<Piece>();
                for (int i = 0; i < pieces.length; i++)
                {
                        for (int j = 0; j < pieces[i].length; j++)
                        {
                                // 加入判断，符合一定条件才去构造 Piece 对象，并加到集合中
                                if (j % 2 == 0)
                                {
                                        // 如果 j 能被 2 整除，即单数行不会创建方块
                                        // 先构造一个 Piece 对象，只设置它在 Piece[][]数组中的索引值
                                        // 所需要的 PieceImage 由其父类负责设置
                                        Piece piece = new Piece(i, j);
                                        // 添加到 Piece 集合中
                                        notNullPieces.add(piece);
                                }
                        }
                }
                return notNullPieces;
        }
}
```

上面的程序中粗体字代码控制了只设置 j % 2 == 0 的行，也就是只设置索引为偶数的行，该子类初始化的游戏界面如图 18.9 所示。

图 18.9　横向分布的方块

18.4　加载界面的图片

正如前面 AbstractBoard 类的代码中看到的，当程序需要创建 *N* 个 Piece 对象时，程序会直接调用 ImageUtil 的 getPlayImages()方法去获取图片，该方法将会随机从 res\drawable-mdpi 目录下取得 *N* 张图片。

为了让 getPlayImages()方法从 res\drawable-mdpi 目录下随机取得 *N* 张图片，程序的实现思路可分为如下几步：

① 通过反射来获取 R.drawable 的所有 Field（Android 的每张图片资源都会自动转换为 R.drawable 的静态 Field），并将这些 Field 值添加到一个 List 集合中。

② 从第一步得到的 List 集合中随机“抽取”*N*/2 个图片 ID。

③ 将第二步得到的 *N*/2 个图片 ID 全部复制一份，这样就得到了 *N* 个图片 ID，而且每个图片 ID 都可以找到与之配对的。

④ 将第三步得到的 *N* 个图片 ID 再次“随机打乱”，并根据图片 ID 加载相应的 Bitmap 对象，最后把图片 ID 及对应的 Bitmap 封装成 PieceImage 后返回。

下面是 ImageUtil 类的代码。

程序清单：codes\18\Link\src\org\crazyit\link\util\ImageUtil.java

```
public class ImageUtil
```

```
{
	// 保存所有连连看图片资源值(int 类型)
	private static List<Integer> imageValues = getImageValues();
	//获取连连看所有图片的 ID（约定所有图片 ID 以 p_开头）
	public static List<Integer> getImageValues()
	{
		try
		{
			// 得到 R.drawable 所有的属性，即获取 drawable 目录下的所有图片
			Field[] drawableFields = R.drawable.class.getFields();
			List<Integer> resourceValues = new ArrayList<Integer>();
			for (Field field : drawableFields)
			{
				// 如果该 Field 的名称以 p_开头
				if (field.getName().indexOf("p_") != -1)
				{
					resourceValues.add(field.getInt(R.drawable.class));
				}
			}
			return resourceValues;
		}
		catch (Exception e)
		{
			return null;
		}
	}
	/**
	 * 随机从 sourceValues 的集合中获取 size 个图片 ID，返回结果为图片 ID 的集合
	 * @param sourceValues 从中获取的集合
	 * @param size 需要获取的个数
	 * @return size 个图片 ID 的集合
	 */
	public static List<Integer> getRandomValues(List<Integer> sourceValues,
		int size)
	{
		// 创建一个随机数生成器
		Random random = new Random();
		// 创建结果集合
		List<Integer> result = new ArrayList<Integer>();
		for (int i = 0; i < size; i++)
		{
			try
			{
				// 随机获取一个数字，大于、小于 sourceValues.size()的数值
				int index = random.nextInt(sourceValues.size());
				// 从图片 ID 集合中获取该图片对象
				Integer image = sourceValues.get(index);
				// 添加到结果集中
				result.add(image);
			}
			catch (IndexOutOfBoundsException e)
			{
				return result;
			}
		}
		return result;
```

```
        }
        /**
         * 从 drawable 目录中获取 size 个图片资源 ID，其中 size 为游戏数量
         * @param size 需要获取的图片 ID 的数量
         * @return size 个图片 ID 的集合
         */
        public static List<Integer> getPlayValues(int size)
        {
              if (size % 2 != 0)
              {
                    // 如果该数除以 2 有余数，将 size 加 1
                    size += 1;
              }
              // 再从所有的图片值中随机获取 size 的一半数量
              List<Integer> playImageValues = getRandomValues(imageValues, size / 2);
              // 将 playImageValues 集合的元素增加一倍（保证所有图片都有与之配对的图片）
              playImageValues.addAll(playImageValues);
              // 将所有图片 ID 随机“洗牌”
              Collections.shuffle(playImageValues);
              return playImageValues;
        }
        /**
         * 将图片 ID 集合转换 PieceImage 对象集合，PieceImage 封装了图片 ID 与图片本身
         * @param context
         * @param resourceValues
         * @return size 个 PieceImage 对象的集合
         */
        public static List<PieceImage> getPlayImages(Context context, int size)
        {
              // 获取图片 ID 组成的集合
              List<Integer> resourceValues = getPlayValues(size);
              List<PieceImage> result = new ArrayList<PieceImage>();
              // 遍历每个图片 ID
              for (Integer value : resourceValues)
              {
                    // 加载图片
                    Bitmap bm = BitmapFactory.decodeResource(
                          context.getResources(),  value);
                    // 封装图片 ID 与图片本身
                    PieceImage pieceImage = new PieceImage(bm, value);
                    result.add(pieceImage);
              }
              return result;
        }
        // 获取选中标识的图片
        public static Bitmap getSelectImage(Context context)
        {
              Bitmap bm = BitmapFactory.decodeResource(context.getResources(),
                    R.drawable.selected);
              return bm;
        }
  }
```