华清远见 FARSIGHT 始于2004 | 嵌入式培训专家

《Android游戏案例开发与关键技术》

作者：华清远见

专业始于专注 卓识源于远见

第2章 图层

本章简介

游戏场景所呈现的游戏画面主要包括游戏主角、游戏地图、游戏道具等， 这些元素可能具有不同的属性，如动画效果、主要功能等，而各个元素之间也存在相互的遮挡、碰撞以及其他事件，由此，对游戏元素的定义和操作至关重要。这里，我们引入图层的概念，来实现以上的处理。我们把相同属性、相同功能的多个元素放置到同一个场景的同一个图层上，这样一个场景中就可能包含多个图层，比如我们在一个游戏场景中，将属于游戏地图的元素——天空、陆地等——置于同一图层，将游戏道具中的工具、对话框等设为另一个图层，游戏主角、怪物、NPC 和障碍物就可以是另外的一些图层，这样处理起来就方便很多。

2.1 图层结构

2.1.1 图层的组成元素

我们将各个图层的游戏元素抽象成某种数据结构，为每一组图层的数据结构分配一个图层 ID，并定义或选择合适的数据来组织这些图层。

游戏元素的数据结构在 ImageModel.java 中定义，包括两个属性：Bitmap 类型的元素图像 img 和 Matrix 类型的元素处理矩阵 matrix；程序在绘制图层的时候，通过 ImageModel 的对象来获取相应的元素图像、变换矩阵和图像大小，这些方法通常被封装在游戏引擎当中，这里我们使用的游戏引擎是自定义的，因此需要自己定义可绘制接口来封装这些方法。程序中的可绘制接口是 Drawable.java，用 ImageModel.java 来继承。代码清单 2-1、2-2 分别表示自定义游戏引擎的可绘制接口类 Drawable.java 和游戏元素实体类 ImageModel.java。

代码清单 2-1 自定义游戏引擎中的可绘制接口类

```java
public interface Drawable {
  public Matrix getPicMatrix();     // 获取图片旋转的矩阵表示
  public Bitmap getCurrentPic();    // 获取当前动作图片的资源
  public int getPicWidth();         // 返回图片的宽度
  public int getPicHeight ();       // 返回图片的高度
}
```

代码清单 2-2 游戏元素实体类

```java
public class ImageModel implements Drawable{
  private Bitmap img;
  private Matrix matrix = new Matrix();
  public void setPicMatrix(Matrix matrix){
       this.matrix = matrix;
  }
  @Override
  public Matrix getPicMatrix() {
       // TODO Auto-generated method stub
       return matrix;
  }
  public void setCurrentPic(Bitmap img){
       this.img = img;
  }
  @Override
  public Bitmap getCurrentPic() {
       // TODO Auto-generated method stub
       return img;
  }

  @Override
  public int getPicWidth() {
       // TODO Auto-generated method stub
       return img.getWidth();
  }

  @Override
  public int getPicHeight() {
       // TODO Auto-generated method stub
       return img.getHeight();
  }
}
```

2.1.2 图层的组织

1．数据结构：Hashmap

组织图层的数据结构使用哈希表（Hashmap）。Hashmap 是一种 T-Value 的形式，我们用 T 代表每个图层的 ID，将 Value 定义为另外一种数据结构，以存储属于同一个图层的元素。值得注意的是，我们同一个图层里 ArrayList 中的元素在绘制完成后仍然呈现层次效果，这种层次效果通过元素在 ArrayList 中的顺序来实现。MainSurface 类中定义了这种组织形式，如代码清单 2-3 所示。

代码清单 2-3　定义组织图层的数据结构

```
// 图片的图层分布
private HashMap<Integer, ArrayList<Drawable>> picLayer =new HashMap<Integer, ArrayList<Drawable>>();
// 修改后的图片的图层分布,这里根据操作分为两个图层，分别是添加的元素和删除的元素
private HashMap<Integer, ArrayList<Drawable>> addPicLayer = new HashMap<Integer, ArrayList<Drawable>>(),removePicLayer = new HashMap<Integer, ArrayList<Drawable>>();
```

注意：
图层之间、同一图层的元素之间都存在层次关系，区别在于，不同的图层属性不同，而同一图层的各个元素属性都是相同的。

2．图层绘制

确定了图层的组成元素和组织结构之后，我们通过 onDraw()方法来绘制每一帧的画面。游戏的动态效果是通过屏幕的不断绘制来实现的，在短暂的时间内呈现不同的画面，使我们看到的场景好像是动起来的。在程序中实现这种效果是通过线程来控制的，比如将 onDraw()方法的绘制频率设为 1 秒 30 次，即 1 秒 30 帧，每一帧当中重复绘制图层上所有的内容，屏幕上就能连续呈现出一张一张的画面。

onDraw()方法的具体实现分为以下两个步骤：

（1）更新图层内容——updatePicLayer（CHANGE_MODE_UPDATE,0,null）;

（2）遍历所有图层，按图层先后顺序绘制。在绘制每个图层时，遍历 ArrayList 中的所有元素，用 getCurrentPic()和 getPicMatrix()分别得到元素的图像和矩阵，通过画布的 drawBitmap 方法来实现——canvas.drawBitmap(drawable.getCurrentPic(),drawable.getPicMatrix(), Paint)。

绘制流程如图 2-1 所示。

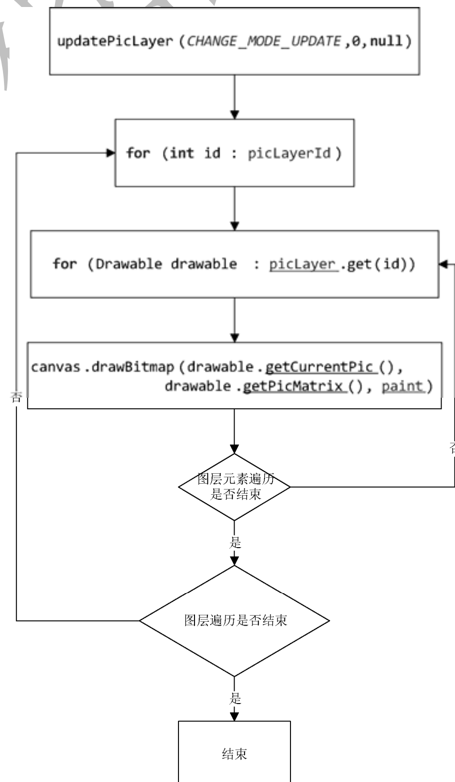


图 2-1　绘制流程

代码清单 2-4 为 onDraw()方法的实现。

代码清单 2-4　图层内容的绘制

```
/**
 * 绘图方法，这个方法是由线程控制，周期性调用的
 */
public void onDraw(Canvas canvas) {
	// 更新图层内容
	updatePicLayer(CHANGE_MODE_UPDATE,0,null);
	// 遍历所有图层，按图层先后顺序绘制
	for (int id : picLayerId) {
			for (Drawable drawable : picLayer.get(id)) {
				canvas.drawBitmap(drawable.getCurrentPic(),
						drawable.getPicMatrix(), paint);
			}
	}
}
```

2.2　图层调用

这一小节主要讲解在工程中实现图层的过程，包括 SurfaceView 介绍、绘图类 MainSurface 对象的创建、绘图线程的启动以及 MainSurface 类的内容。

2.2.1　界面的视图——SurfaceView 类

- 开发复杂游戏，而且对程序的执行效率要求更高时用此类，因本身就是双缓冲机制的。
- SurfaceView 可以直接访问一个画布。
- SurfaceView 是提供给需要直接画像素而不是使用窗体部件的应用使用的。
- View 及其子类（如 TextVie,Button）要画在 Surface 上。每个 Surface 创建一个 Canvas 对象（属性时常改变）用来管理 View 在 Surface 上绘制操作。
- 使用 SurfaceView 绘图时，一般都是出现在最顶层。在使用时要对其创建、销毁情况改变进行监视，这就需要实现 SurfaceHolder.Callback 接口，如果要对被绘制的画布进行裁剪，控制其大小时都需要使用 SurfaceHolder 来完成处理。
- 在程序中，SurfaceHolder 对象需要通过 getHolder 方法来获得，同时还需要 addCallback 方法来添加“回调函数”。
- SurfaceView 与 View 不同之处在于，SurfaceView 不需要通过线程来更新视图，在绘制前必须使用 lockCanvas 方法锁定画布，并得到画布，然后在画布上绘制。
- 绘制完成后，使用 unlockCanvasAndPost 方法来解锁画布。
- addCallback:给 SurfaceView 添加一个回调函数。
- removeCallback:从 SurfaceView 移除回调函数。

2.2.2　创建 MainSurface

我们在程序中定义一个继承 SurfaceView 的类 MainSurface。程序启动时，创建一个 MainSurface 对象，在 MainSurface 类的构造方法中，创建一个绘图线程 odt。

MainSurface 对象的声明和创建在 GameActivity 类中实现，MainSurface 对象声明及对象创建如代码清单 2-5 及 2-6 所示。

代码清单 2-5　MainSurface 对象声明

```
private MainSurface surface;
```

代码清单 2-6　MainSurface 对象创建

华清远见

```
surface = new MainSurface(this);
```

2.2.3 绘制 MainSurface

MainSurface 的绘制方法 onDraw()由线程控制，采用周期性调用。首先介绍控制 onDraw 方法的线程类 onDrawTread，然后介绍启动该线程的过程。

1．绘制线程类

绘制线程类代码如代码清单 2-7 所示。

代码清单 2-7 onDrawTread.java

```
public class OnDrawThread extends Thread{
  private MainSurface surface;
  private SurfaceHolder sh;
  private int drawSpeed;
  // 每次绘制后的休息毫秒数，这个值是根据常量中的绘制帧数决定的
  public OnDrawThread(MainSurface surface){
      super();
      this.surface = surface;
      sh = surface.getHolder();
      drawSpeed = 1000/Constant.ON_DRAW_SLEEP;
  }

  public void run(){
      super.run();
      Canvas canvas = null;
      while(GamingInfo.getGamingInfo().isGaming()){
          try{
              canvas = sh.lockCanvas(null);
                  if(canvas!=null){
                      surface.onDraw(canvas);
                  }
          }catch(Exception e){
              Log.e(this.getName(), e.toString());
              e.printStackTrace();
          }finally{
              try{
                  if(sh!=null){
                      sh.unlockCanvasAndPost(canvas);
                  }
              }catch(Exception e){
                  Log.e(this.getName(), e.toString());
              }
          }
          try{
              Thread.sleep(drawSpeed);
          }catch(Exception e){

          }
      }
  }
}
```

2．启动绘制线程类

MainSurface 被创建时，在 MainSurface 的构造方法里面创建一个绘图线程 odt，同时 MainSurface 所继承的 SurfaceHolder.Callback 监听到 MainSurface 的创建时自动调用 surfaceCreate 方法，在 surfaceCreate 中开启线程，从而以双缓冲模式绘制图层。

MainSurface 类的代码实现如代码清单 2-8 所示。

代码清单 2-8 图层内容的绘制

```
/**
```

```
 * 绘图类
 * @author Xiloerfan
 *
 */
public class MainSurface extends SurfaceView implements SurfaceHolder.Callback {
    /**
     * 修改图层的操作定义
     */
    // 更新图层
    private final static int CHANGE_MODE_UPDATE = 0;
    // 添加元素到图层
    private final static int CHANGE_MODE_ADD = 1;
    // 从图层删除元素
    private final static int CHANGE_MODE_REMOVE = 2;

    // 图片的图层分布
    private HashMap<Integer, ArrayList<Drawable>> picLayer =new HashMap<Integer, ArrayList <Drawable>>();
    // 修改后的图片的图层分布,这里根据操作分为两个图层，分别是添加的元素和删除的元素
    private HashMap<Integer, ArrayList<Drawable>> addPicLayer = new HashMap<Integer,
ArrayList<Drawable>>(),removePicLayer = new HashMap<Integer, ArrayList<Drawable>>();
    // 是否修改过图层
    private boolean changeLayer = false;
    private int picLayerId[];
    // 定义一个图层 ID，加速获取图层绘制（省去了从 map 中获取各个图层排序问题）
    private Paint paint;
    // 画笔
    private OnDrawThread odt;
    // 屏幕绘制线程，用于控制绘制帧数，周期性调用 onDraw 方法
    public MainSurface(Context context) {
        super(context);
        this.getHolder().addCallback(this);
        paint = new Paint();
        paint.setColor(Color.WHITE);
        odt = new OnDrawThread(this);

    }

    public void surfaceChanged(SurfaceHolder arg0, int arg1, int arg2, int arg3) {
        // TODO Auto-generated method stub

    }

    public void surfaceCreated(SurfaceHolder arg0) {
        // TODO Auto-generated method stub
        odt.start();
    }

    public void surfaceDestroyed(SurfaceHolder arg0) {
        // TODO Auto-generated method stub
        paint = null;
        picLayerId = null;
        picLayer = null;

    }

    @Override
    /**
     * 绘图方法，这个方法是由线程控制，周期性调用的
     */
    public void onDraw(Canvas canvas) {
        // 更新图层内容
        updatePicLayer(CHANGE_MODE_UPDATE,0,null);
        // 遍历所有图层，按图层先后顺序绘制
        for (int id : picLayerId) {
                for (Drawable drawable : picLayer.get(id)) {
                    canvas.drawBitmap(drawable.getCurrentPic(),
```

```
                                    drawable.getPicMatrix(), paint);
                    }
            }
    }
    /**
     * 更新图层，这里分为三种操作方法，分别是更新临时图层中的内容到绘制图层中，删除绘制图层中的元素，添加绘制图层中的元素
     * 这里加了个线程锁，保证多线程下操作图层的安全性
     * @param mode对绘制图层的操作类型，对应当前类的 CHANGE_MODE 常量
     * @param layerId  操作的图层 ID
     * @param draw     操作的图层元素
     */
    private synchronized void updatePicLayer(int mode,int layerId,Drawable draw){
        switch(mode){
        // 将临时图层中的内容更新至绘制图层中
        case CHANGE_MODE_UPDATE:
            // 如果有修改
            if(changeLayer){
                // 向图层添加新的元素
                for(Integer id:addPicLayer.keySet()){
                    for(Drawable d:addPicLayer.get(id)){
                        // 如果要添加的元素所处图层不存在，则创建这个图层，并更新图层 ID 数组
                        if(this.picLayer.get(id)==null){
                            this.picLayer.put(id, new ArrayList<Drawable>());
                            updateLayerIds(id);
                        }
                        this.picLayer.get(id).add(d);
                    }
                }
                addPicLayer.clear();
                // 删除图层中的元素
                for(Integer id:removePicLayer.keySet()){
                    for(Drawable d:removePicLayer.get(id)){
                        this.picLayer.get(id).remove(d);
                    }
                }
                removePicLayer.clear();
                changeLayer = false;
            }
            break;
        /**
         * 无论是向绘图图层中添加还是删除元素，都不是直接操作绘制图层，都是存放在对应的临时图层中，等待绘制方法将绘制周期中变化的内容更新到绘制图层中
         * 保证多线程操作情况下的安全性
         */
        // 添加一个元素
        case CHANGE_MODE_ADD:
            ArrayList<Drawable> al = addPicLayer.get(layerId);
            if(al==null){
                al = new ArrayList<Drawable>();
                addPicLayer.put(layerId, al);
            }
            al.add(draw);
            changeLayer = true;
            break;
        // 删除一个元素
        case CHANGE_MODE_REMOVE:
            ArrayList<Drawable> al1 = removePicLayer.get(layerId);
            if(al1==null){
                al1 = new ArrayList<Drawable>();
                removePicLayer.put(layerId, al1);
            }
            al1.add(draw);
            changeLayer = true;
            break;
        }
```

```
    }

    /**
     * 将一个可绘制的图放入图层中
     *
     * @param layer
     * 图层号虽然是 int，但是实际上只支持到 byte，原因是图层没有必要那么多
     * @param pic
     * 可绘制的图
     */
    public void putDrawablePic(int layer, Drawable pic) {
        updatePicLayer(CHANGE_MODE_ADD,layer,pic);
    }

    /**
     * 将一个可绘制的图从图层中移除
     *
     * @param layer
     * @param pic
     */
    public void removeDrawablePic(int layer, Drawable pic) {
        updatePicLayer(CHANGE_MODE_REMOVE,layer,pic);
    }

    /**
     * 更新图层 Id
     *
     * @param newLayerId
     */
    private void updateLayerIds(int newLayerId) {
        // 初始化图层
        if (picLayerId == null) {
            picLayerId = new int[1];
            picLayerId[0] = newLayerId;
            // 将新的图层 ID 添加到初始化的图层 ID 数组中
        } else {
            // 创建一个新的图层数组，长度比原来的大 1 位
            int picLayerIdFlag[] = new int[picLayerId.length + 1];
            for (int i = 0; i < picLayerId.length; i++) {
                // 排序操作，如果新的图层 ID 小于当前图层 ID，将新的图层 ID 插入其中
                if (picLayerId[i] > newLayerId) {
                    for (int f = picLayerIdFlag.length - 1; f > i; f--) {
                        picLayerIdFlag[f] = picLayerId[f - 1];
                    }
                    picLayerIdFlag[i] = newLayerId;
                    break;
                } else {
                    picLayerIdFlag[i] = picLayerId[i];
                }
                // 如果到了最后，都没有比新图层 ID 大的，就将新的图层 ID 存入最后
                if (i == picLayerId.length - 1) {
                    picLayerIdFlag[picLayerIdFlag.length - 1] = newLayerId;
                }
            }
            // 将新的图层 ID 数组覆盖原有的
            this.picLayerId = picLayerIdFlag;
        }
    }
    /**
     * 这个方法用于停止绘制线程，一旦停止，Surface 的 onDraw 方法将不会再调用
     */
    public void stopDraw(){
        odt.stopDraw();
    }
}
```

2.3 图层示例

下面我们举一个图层的例子，以更深入地掌握以上理论，如代码清单 2-9 所示。

代码清单 2-9 main.xml

```
view sourceprint?01 <?xml version="1.0" encoding="utf-8"?>

 <LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"

    android:layout_width="fill_parent" android:layout_height="fill_parent"

    android:orientation="vertical">


    <LinearLayout android:id="@+id/LinearLayout01"

        android:layout_width="wrap_content" android:layout_height="wrap_content">

        <Button android:id="@+id/Button01" android:layout_width="wrap_content"

            android:layout_height="wrap_content" android:text="简单绘画"></Button>

        <Button android:id="@+id/Button02" android:layout_width="wrap_content"

            android:layout_height="wrap_content" android:text="定时器绘画"></Button>

    </LinearLayout>

    <SurfaceView android:id="@+id/SurfaceView01"

        android:layout_width="fill_parent" android:layout_height="fill_parent"></SurfaceView>

 </LinearLayout>
```

代码清单 2-10 TestSurfaceView.java

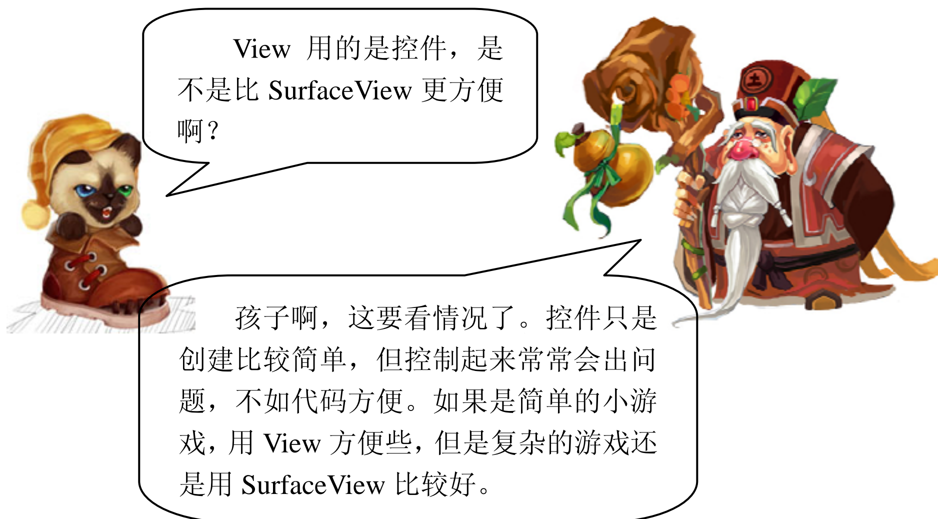
```
import java.util.Timer;
import java.util.TimerTask;

import android.app.Activity;
import android.graphics.Canvas;
import android.graphics.Color;
import android.graphics.Paint;
import android.graphics.Rect;
import android.os.Bundle;
import android.util.Log;
import android.view.SurfaceHolder;
import android.view.SurfaceView;
import android.view.View;
import android.widget.Button;

public class TestSurfaceView extends Activity {
    /** Called when the activity is first created. */
    Button btnSimpleDraw, btnTimerDraw;
    SurfaceView sfv;
    SurfaceHolder sfh;

    private Timer mTimer;
    private MyTimerTask mTimerTask;
    int Y_axis[],// 保存正弦波的 Y 轴上的点
    centerY,// 中心线
    oldX,oldY,// 上一个 XY 点
    currentX;// 当前绘制到的 X 轴上的点
```

```
@Override
public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.main);

    btnSimpleDraw = (Button) this.findViewById(R.id.Button01);
    btnTimerDraw = (Button) this.findViewById(R.id.Button02);
    btnSimpleDraw.setOnClickListener(new ClickEvent());
    btnTimerDraw.setOnClickListener(new ClickEvent());
    sfv = (SurfaceView) this.findViewById(R.id.SurfaceView01);
    sfh = sfv.getHolder();

    // 动态绘制正弦波的定时器
    mTimer = new Timer();
    mTimerTask = new MyTimerTask();

    // 初始化 y 轴数据
    centerY = (getWindowManager().getDefaultDisplay().getHeight() - sfv
            .getTop()) / 2;
    Y_axis = new int[getWindowManager().getDefaultDisplay().getWidth()];
    for (int i = 1; i < Y_axis.length; i++) {         // 计算正弦波
        Y_axis[i - 1] = centerY
                - (int) (100 * Math.sin(i * 2 * Math.PI / 180));
    }
}

class ClickEvent implements View.OnClickListener {

    @Override
    public void onClick(View v) {

        if (v == btnSimpleDraw) {
            SimpleDraw(Y_axis.length-1);          // 直接绘制正弦波

        } else if (v == btnTimerDraw) {
            oldY = centerY;
            mTimer.schedule(mTimerTask, 0, 5);    // 动态绘制正弦波
        }

    }

}

class MyTimerTask extends TimerTask {
    @Override
    public void run() {

        SimpleDraw(currentX);
        currentX++;                                  // 往前进
        if (currentX == Y_axis.length - 1) {// 如果到了终点，则清屏重来
            ClearDraw();
            currentX = 0;
            oldY = centerY;
        }
    }

}

/**
 * 绘制指定区域
 */
void SimpleDraw(int length) {
    if (length == 0)
        oldX = 0;
    Canvas canvas = sfh.lockCanvas(new Rect(oldX, 0, oldX + length,
```

```
            getWindowManager().getDefaultDisplay().getHeight()));// 关键:获取画布
        Log.i("Canvas:",
                String.valueOf(oldX) + "," + String.valueOf(oldX + length));

        Paint mPaint = new Paint();
        mPaint.setColor(Color.GREEN);                // 画笔为绿色
        mPaint.setStrokeWidth(2);                    // 设置画笔粗细

        int y;
        for (int i = oldX + 1; i < length; i++) {    // 绘制正弦波
            y = Y_axis[i - 1];
            canvas.drawLine(oldX, oldY, i, y, mPaint);
            oldX = i;
            oldY = y;
        }
        sfh.unlockCanvasAndPost(canvas);          // 解锁画布，提交画好的图像
    }

    void ClearDraw() {
        Canvas canvas = sfh.lockCanvas(null);
        canvas.drawColor(Color.BLACK);            // 清除画布
        sfh.unlockCanvasAndPost(canvas);

    }
}
```

如图 2-2 所示为程序运行截图。

图 2-2　程序运行截图



2.4 本章小结

本章主要介绍了图层的结构以及在工程中的调用。讲解了图层组成元素的数据结构、组织图层的数据结构，以及控制图层绘制的线程、线程的创建和启动。

华清远见

联系方式

集团官网：www.hqyj.com 嵌入式学院：www.embedu.org 移动互联网学院：www.3g-edu.org

企业学院：www.farsight.com.cn 物联网学院：www.topsight.cn 研发中心：dev.hqyj.com

集团总部地址：北京市海淀区西三旗悦秀路北京明园大学校内 华清远见教育集团

北京地址：北京市海淀区西三旗悦秀路北京明园大学校区，电话：010-82600386/5

上海地址：上海市徐汇区漕溪路 250 号银海大厦 11 层 B 区，电话：021-54485127

深圳地址：深圳市龙华新区人民北路美丽 AAA 大厦 15 层，电话：0755-25590506

成都地址：成都市武侯区科华北路 99 号科华大厦 6 层，电话：028-85405115

南京地址：南京市白下区汉中路 185 号鸿运大厦 10 层，电话：025-86551900

武汉地址：武汉市工程大学卓刀泉校区科技孵化器大楼 8 层，电话：027-87804688

西安地址：西安市高新区高新一路 12 号创业大厦 D3 楼 5 层，电话：029-68785218

广州地址：广州市天河区中山大道 268 号天河广场 3 层，电话：020-28916067