# SurfaceView

## 一、 SurfaceView主要作用

1. 视频输出的屏幕。（这个用得比较多）
2. 绘制图像。
3. 视频画面，游戏画面，相机画面。

## 二、注意事项

surfaceCreated有特定的生命周期，注意在callback中进行操作。（具体请看例子）

## 三、生命周期

**当Activity完全显示之后，SurfaceView才会被创建**

**只要Activity不是在前台，SurfaceView就会销毁**

## 四、SurfaceView与View的区别

- View的绘图效率不高，主要用于动画变化较少的程序
- SurfaceView 绘图效率较高，用于界面更新频繁的程序
- SurfaceView拥有独立的Surface（绘图表面），即它不与其宿主窗口共享同一个Surface。
  一般来说，每一个窗口在SurfaceFlinger服务中都对应有一个Layer，用来描述它的绘图表面。对于那些具有SurfaceView的窗口来说，每一个SurfaceView在SurfaceFlinger服务中还对应有一个独立的Layer或者LayerBuffer，用来单独描述它的绘图表面，以区别于它的宿主窗口的绘图表面。因此SurfaceView的UI就可以在一个独立的线程中进行绘制，可以不会占用主线程资源。
- SurfaceView使用双缓冲机制，播放视频时画面更流畅。

### 什么是双缓冲机制

在运用时可以理解为：SurfaceView在更新视图时用到了两张 Canvas，一张 frontCanvas 和一张 backCanvas ，每次实际显示的是 frontCanvas ，backCanvas 存储的是上一次更改前的视图。当你在播放这一帧的时候，它已经提前帮你加载好后面一帧了，所以播放起视频很流畅。
当使用lockCanvas（）获取画布时，得到的实际上是backCanvas 而不是正在显示的 frontCanvas ，之后你在获取到的 backCanvas 上绘制新视图，再 unlockCanvasAndPost（canvas）此视图，那么上传的这张 canvas 将替换原来的 frontCanvas 作为新的frontCanvas ，原来的 frontCanvas 将切换到后台作为 backCanvas 。例如，如果你已经先后两次绘制了视图A和B，那么你再调用 lockCanvas（）获取视图，获得的将是A而不是正在显示的B，之后你将重绘的 A 视图上传，那么 A 将取代 B 作为新的 frontCanvas 显示在 SurfaceView 上，原来的B则转换为backCanvas。
相当与多个线程，交替解析和渲染每一帧视频数据。

### 使用场景

所以SurfaceView一方面可以实现复杂而高效的UI，另一方面又不会导致用户输入得不到及时响应。常用于画面内容更新频繁的场景，比如游戏、视频播放和相机预览。

# 五、使用步骤

1.继承SurfaceView
2.实现SurfaceHolder.Callback接口
3.使用getHolder().addCallback(this);
4.重写surfaceChanged(...),surfaceCreated(..),surfaceDestroyed(..)
5.画图holder.lockCanvas()->画图ing->holder.unlockCanvasAndPost(..);

### （1）创建SurfaceView

创建自定义的SurfaceView继承自SurfaceView,并实现两个接口：SurfaceHolder.Callback和Runnable.代码如下：

```
public class MySurfaceView extends SurfaceView implements
SurfaceHolder.Callback,Runnable
```

通过实现这两个接口，就需要在自定义的SurfaceView中实现接口的方法，对于SurfaceHolder.Callback方法，需要实现如下方法，其实就是SurfaceView的生命周期：

```
 @Override
    public void surfaceCreated(SurfaceHolder holder) {


    }
   @Override
    public void surfaceChanged(SurfaceHolder holder, int format, int width, int
height) {

    }

    @Override
    public void surfaceDestroyed(SurfaceHolder holder) {

    }
```

对于Runnable接口，需要实现run()方法，

```
 @Override
    public void run() {

}
```

### （2）初始化SurfaceView

在自定义的MySurfaceView的构造方法中，**需要对SurfaceView进行初始化，包括SurfaceHolder的初始化、画笔的初始化等。**在自定义的SurfaceView中，通常需要定义以下三个成员变量:

```
    private SurfaceHolder mHolder;
    private Canvas mCanvas;//绘图的画布
    private boolean mIsDrawing;//控制绘画线程的标志位
```

SurfaceHolder，顾名思义，它里面保存了一个对Surface对象的引用，而我们执行绘制方法本质上就是操控Surface。SurfaceHolder因为保存了对Surface的引用，所以使用它来处理Surface的生命周期。（说到底 SurfaceView的生命周期其实就是Surface的生命周期）例如使用 SurfaceHolder来处理生命周期的初始化。

**初始化代码如下：**

```
 public MySurfaceView(Context context, AttributeSet attrs) {
        super(context, attrs);
        initView();
    }

    public MySurfaceView(Context context, AttributeSet attrs, int defStyleAttr)
{
        super(context, attrs, defStyleAttr);
        initView();
    }

    public MySurfaceView(Context context, AttributeSet attrs, int defStyleAttr,
int defStyleRes) {
        super(context, attrs, defStyleAttr);
    }

    private void initView() {
        mHolder = getHolder();//获取SurfaceHolder对象
        mHolder.addCallback(this);//注册SurfaceHolder的回调方法
        setFocusable(true);
        setFocusableInTouchMode(true);
        this.setKeepScreenOn(true);
    }
```

**（3）使用SurfaceView**

通过SurfaceHolder对象的lockCanvans()方法，我们可以获取当前的Canvas绘图对象。接下来的操作就和自定义View中的绘图操作一样了。需要注意的是这里获取到的Canvas对象还是继续上次的Canvas对象，而不是一个新的对象。因此，之前的绘图操作都会被保留，如果需要擦除，则可以在绘制前，通过drawColor()方法来进行清屏操作。

绘制的时候，在surfaceCreated()方法中开启子线程进行绘制，而子线程使用一个while(mIsDrawing)的循环来不停的进行绘制，在绘制的逻辑中通过lockCanvas()方法获取Canvas对象进行绘制，通过unlockCanvasAndPost(mCanvas)方法对画布内容进行提交。整体代码模板如下：

```
import android.content.Context;
import android.graphics.Canvas;
import android.util.AttributeSet;
import android.view.SurfaceHolder;
import android.view.SurfaceView;

public class MySurfaceView  extends SurfaceView
        implements SurfaceHolder.Callback, Runnable {

    // SurfaceHolder
    private SurfaceHolder mHolder;
    // 用于绘图的Canvas
    private Canvas mCanvas;
    // 子线程标志位
```

```
    private boolean mIsDrawing;

    public MySurfaceView(Context context) {
        super(context);
        initView();
    }

    public MySurfaceView(Context context, AttributeSet attrs) {
        super(context, attrs);
        initView();
    }

    public MySurfaceView(Context context, AttributeSet attrs, int defStyle) {
        super(context, attrs, defStyle);
        initView();
    }

    private void initView() {
        mHolder = getHolder();
        mHolder.addCallback(this);
        setFocusable(true);
        setFocusableInTouchMode(true);
        this.setKeepScreenOn(true);
        //mHolder.setFormat(PixelFormat.OPAQUE);
    }

    @Override
    public void surfaceCreated(SurfaceHolder holder) {
        mIsDrawing = true;
        new Thread(this).start();
    }

    @Override
    public void surfaceChanged(SurfaceHolder holder,
                               int format, int width, int height) {
    }

    @Override
    public void surfaceDestroyed(SurfaceHolder holder) {
        mIsDrawing = false;
    }

    @Override
    public void run() {
        while (mIsDrawing) {
            draw();
        }
    }
    //绘图操作
    private void draw() {
        try {
            mCanvas = mHolder.lockCanvas();
            // draw sth绘制过程
        } catch (Exception e) {
        } finally {
            if (mCanvas != null)
                mHolder.unlockCanvasAndPost(mCanvas);//保证每次都将绘图的内容提交
        }
```

```
    }
}
```

这里说一个优化的地方，这就是在run方法中。
在我们的draw()方法每一次更新所耗费的时间是不确定的。举个例子 比如第一次循环draw() 耗费了1000毫秒 ， 第二次循环draw() 耗时2000毫秒。很明显这样就会造成运行刷新时间时快时慢,可能出现卡顿现象。为此最好保证每次刷新的时间是相同的，这样可以保证整体画面过渡流畅。

```
    /**每30帧刷新一次屏幕**/
        public static final int TIME_IN_FRAME = 30;
    @Override
    public void run() {
        while (mIsRunning) {

        /**取得更新之前的时间**/
        long startTime = System.currentTimeMillis();

        /**在这里加上线程安全锁**/
        synchronized (mSurfaceHolder) {
            /**拿到当前画布 然后锁定**/
            mCanvas =mSurfaceHolder.lockCanvas();
            draw();
            /**绘制结束后解锁显示在屏幕上**/
            mSurfaceHolder.unlockCanvasAndPost(mCanvas);
        }

        /**取得更新结束的时间**/
        long endTime = System.currentTimeMillis();

        /**计算出一次更新的毫秒数**/
        int diffTime  = (int)(endTime - startTime);

        /**确保每次更新时间为30帧**/
        while(diffTime <=TIME_IN_FRAME) {
            diffTime = (int)(System.currentTimeMillis() - startTime);
            /**线程等待**/
            Thread.yield();
        }

        }
    }
```