

目录

目录

一、进程与线程

- 1.1 进程、线程
- 1.2 Android中的主线程
- 1.3 ANR 拓展

二、Handler机制

- 2.1 Handler机制
- 2.2 Handler的使用场景
- 2.3 Handler机制简介
- 2.4 Handler原理：UI线程与消息队列机制
- 2.5 Handler常用方法
- 2.6 Handler的使用
- 2.7 Handler的使用举例
 - 2.7.1 发送Runnable对象
 - 2.7.2 发送Message对象
 - 2.7.3 辨析Runnable与Message
- 2.8 Handler总结

三、Android中的多线程

- 3.1 Thread
- 3.2 ThreadPool
 - 3.2.1 为什么要使用线程池
 - 3.2.2 几种常用的线程池
 - 3.2.3 使用示例
- 3.3 AsyncTask(已弃用)
- 3.4 HandlerThread
- 3.5 IntentService(不常用, 自学)
- 3.6 Android多线程总结

四、自定义View

- 4.1 View绘制的三个重要步骤
- 4.2 绘制流程
- 4.3 自定义View：重写onDraw
 - 4.3.1 画点
 - 4.3.2 画线
 - 4.3.3 画圆
 - 4.3.4 填充
 - 4.3.5 不规则图形
 - 4.3.6 画文本
- 4.4 自定义View总结

一、进程与线程

1.1 进程、线程

1. 进程：资源分配的最小单位 ==> 一个软件
 1. 如一辆列车
2. 线程：CPU调度的最小单位 ==> 一个软件的各个功能
 1. 如一辆列车的列车长
3. 主要区别：

1. 一个进程可以有多个线程
2. 同一个进程的多个线程，共享进程的资源

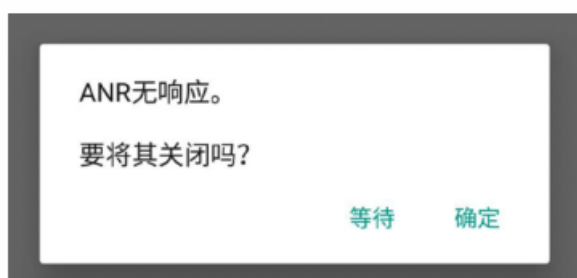
1.2 Android中的主线程

1. 启动应用时，系统会为该应用创建一个称为“main”（主线程）的执行线程。这个线程负责所有和UI界面有关的显示、以及响应UI事件监听任务，因此又称座UI线程。
2. 划重点：所有跟**ui相关的操作**都必须放在**主线程**

1.3 ANR 拓展

ANR: Application Not Responding

1. 程序中所有的组件都会运行在UI线程中，所以必须保证该线程的工作效率
2. UI线程一旦出现问题，就会降低用户体验
3. 如在UI线程中进行耗时操作，如下载文件、查询数据库等就会阻塞UI线程，长时间无法响应UI交互操作，给用户带来“卡屏”、“死机”的感觉



二、Handler机制

2.1 Handler机制

Handler机制为Android系统解决了以下两个问题：

1. 任务调度
2. 线程通信

2.2 Handler的使用场景

先看这样两个例子：

1. 启动今日头条app的时候，展示了一个开屏广告，默认播放10秒；在10秒后，需跳转到主界面。
2. 用户在抖音App中，点击下载视频，下载过程中需要弹出Loading弹窗，下载结束后提示用户下载成功/失败。

2.3 Handler机制简介

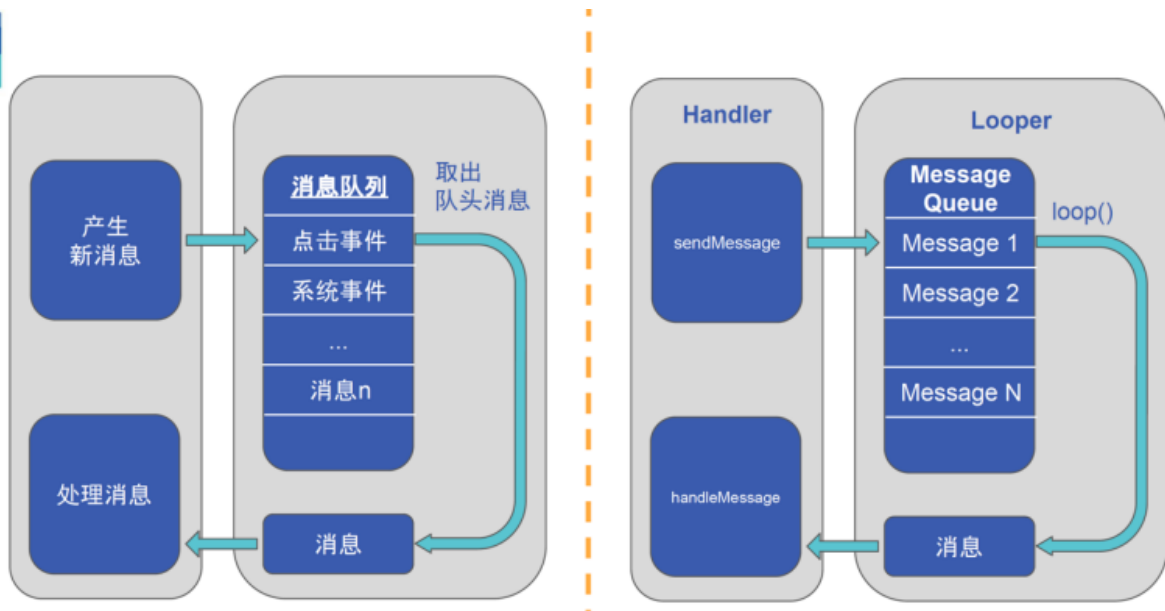
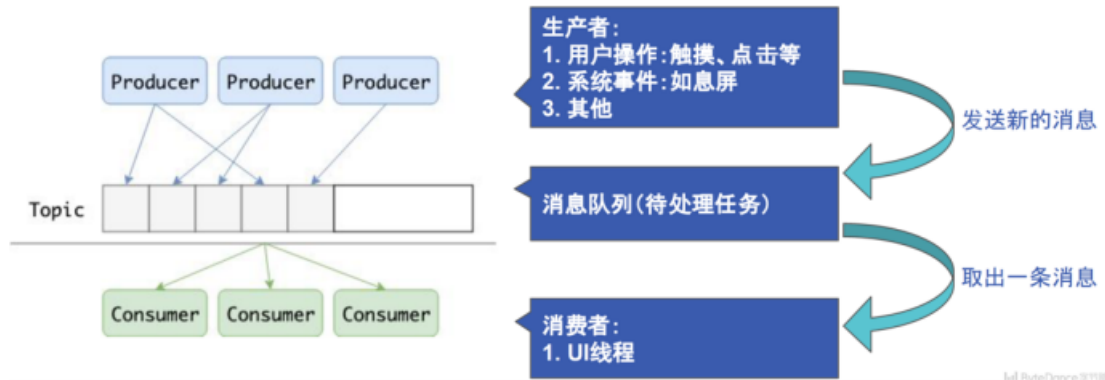
本质是消息机制，负责消息的分发以及处理

1. 通俗点来说，每个线程都有一个“流水线”，我们可往这条流水线上放“消息”，流水线的末端有工作人员会去处理这些消息。因为流水线是单线的，所有消息都必须按照先来后到的形式依次处理
2. 放什么消息以及怎么处理消息，是需要我们去自定义的。Handler机制相当于提供了这样的一套模式，我们只需要“放消息到流水线上”，“编写这些消息的处理逻辑”就可以了，流水线会源源不断把消息运送到末端处理。

- 最后注意重点：每个线程只有一个“流水线”，他的基本范围是线程，负责线程内的通信以及线程间的通信。
- 每个线程可以看成是一个厂房，每个厂房只有一个生产线。

2.4 Handler原理：UI线程与消息队列机制

Android中，UI线程负责处理界面的展示，响应用户的操作：



- Message**：消息，由MessageQueue统一队列，然后交由Handler处理
- MessageQueue**：消息队列，用来存放Handler发送过来Message，并且按照先入先出的规则执行
- Handler**：处理者，负责发送和处理Message每个Message必须有一个对应的Handler
- Looper**：消息轮询器，不断的从MessageQueue中抽取Message并执行

2.5 Handler常用方法

```
// 立即发送消息
public final boolean sendMessage(Message msg)
public final boolean post(Runnable r);

// 延时发送消息：马上发送消息，但是会延迟处理
public final boolean sendMessageDelayed(Message msg, long delayMillis)
public final boolean postDelayed(Runnable r, long delayMillis);

// 定时发送消息
public boolean sendMessageAtTime(Message msg, long uptimeMillis);
public final boolean postAtTime(Runnable r, long uptimeMillis);
```

```
public final boolean postAtTime(Runnable r, Object token, long uptimeMillis);

// 移除消息
public final void removeCallbacks(Runnable r);
public final void removeMessages(int what); // what字段:给消息的命名
public final void removeCallbacksAndMessages(Object token); // token=null: 表示移除所有消息
```

2.6 Handler的使用

1. 调度Message:

1. 建一个Handler, 实现handleMessage()方法
2. 在适当的时候给上面的Handler发送消息

2. 调度Runnable:

1. 建一个Handler, 然后直接调度Runnable即可

3. 取消调度:

1. 通过Handler取消已经发送过的Message/Runnable

2.7 Handler的使用举例

2.7.1 发送Runnable对象

1. 新建一个Handler对象: new Handler()
2. 新建一个Runnable对象
 1. 重写run()方法, 表示该Runnable对象要做什么事情
3. 调用Handler对象的post()方法, 将Runnable对象发送出去

启动今日头条app的时候, 展示了一个开屏广告, 默认播放3秒; 在3秒后, 需跳转到主界面

```
Handler handler = new Handler();
Runnable loadingDismissRunnable = new Runnable() {
    @Override
    public void run() {
        dismissLoading(); 该Runnable启动时, 令开屏广告消失
    }
};
handler.postDelayed(loadingDismissRunnable, delayMillis: 10000);
```

10s后执行该Runnable, 令开屏广告消失

如果用户点击了跳过, 则应该直接进入主界面

```

Handler handler = new Handler();
Runnable loadingDismissRunnable = new Runnable() {
    @Override
    public void run() {
        dismissLoading();
    }
};
handler.postDelayed(loadingDismissRunnable, delayMillis: 10000);
Button skipButton = findViewById(R.id.skip_button);
skipButton.setOnClickListener(new View.OnClickListener() {
    @Override
    public void onClick(View v) {
        handler.removeCallbacks(loadingDismissRunnable);
        dismissLoading();
    }
});

```

点击了跳过, 需要将之前负责延迟的Runnable取消, 然后再将广告删除

2.7.2 发送Message对象

1. 新建一个Handler对象: `new Handler(Looper.getMainLopper())`
 1. 重写`handlerMessage()`方法, 表示收到不同Message时做出的反应
2. 调用Handler对象的`post()`方法, 将Message对象发送出去
 1. 可以直接发送`msg.what`, 来代表一个msg

用户在抖音App中, 点击下载视频, 下载过程中需要弹出Loading窗, 下载结束后提示用户下载成功/失败

```

private Handler mHandler = new Handler(Looper.getMainLooper()) {
    @Override
    public void handleMessage(@NonNull Message msg) {
        Log.i(tag: "MainActivity", msg: "what = " + msg.what);
        switch (msg.what) {
            case MSG_START_DOWNLOAD:
                showToast("开始下载");
                showDownloadingUI();
                break;
            case MSG_DOWNLOAD_SUCCESS:
                showToast("下载成功");
                dismissDownloadingUI();
                break;
            case MSG_DOWNLOAD_FAIL:
                showToast("下载失败");
                dismissDownloadingUI();
                break;
            default:
                break;
        }
    }
};

```

```

        downloadingButton.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View v) {
                new DownloadThread().start();
            }
        });
    }

    private class DownloadThread extends Thread {
        @Override
        public void run() {
            mHandler.sendMessage(MSG_START_DOWNLOAD); 给主线程发消息
            try { 通过try执行下载操作
                downloadVideo();
                mHandler.sendMessage(MSG_DOWNLOAD_SUCCESS);
            } catch (Exception e) { 下载失败, 发送失败消息
                mHandler.sendMessage(MSG_DOWNLOAD_FAIL);
            }
        }

        private void downloadVideo() throws InterruptedException {
            // 具体下载操作
            Log.i( tag: "MainActivity", msg: "downloading");
            Thread.sleep( millis: 3000);
        }
    }
}

```

2.7.3 辨析Runnable与Message

1. Runnable会被打包成Message，所以实际上Runnable也是Message
2. 没有明确的界限，取决于使用的方便程度

以下两段代码等价

```

Handler handler = new Handler();
Runnable loadingDismissRunnable = new Runnable() {
    @Override
    public void run() {
        dismissLoading();
    }
};
handler.postDelayed(loadingDismissRunnable, delayMillis: 10000);

```

```

Handler handler = new Handler() {
    @Override
    public void handleMessage(@NonNull Message msg) {
        if (msg.what == 1) {
            dismissLoading();
        }
    }
};
handler.sendMessageDelayed( what: 1, delayMillis: 10000);

```

2.8 Handler总结

1. Handler就是Android中的消息队列机制的一个应用，可理解为是一种生产者消费者的模型，解决了Android中的线程内&线程间的任务调度问题

2. **Handler**机制的本质就是一个死循环，待处理的**Message**加到队列里面，**Looper**负责轮询执行
3. 掌握**Handler**的基本用法：立即/延时/定时发送消息、取消消息

三、Android中的多线程

3.1 Thread

1. 新建一个类，继承**Thread**，重写其**run()**方法
2. 调用时，先新建一个实例
 1. 可以传入一个String参数，表示线程的名字
3. 调用**thread.start()**方法，开启线程

```
private class MyThread extends Thread {
    public MyThread(String name) {
        super(name);
    }
    @Override
    public void run() {
        for (int i = 0; i < 100 ; i++) {
            Log.i(TAG, msg: "current thread:" + Thread.currentThread().getName() + ", i = " + i);
        }
    }
}

/**
 * 创建一个Thread
 */
private void threadTest() {
    setContentView(R.layout.thread_test_1);
    MyThread thread = new MyThread( name: "mythread");
    Button start = findViewById(R.id.start_button);
    start.setOnClickListener(new View.OnClickListener() {
        @Override
        public void onClick(View v) {
            // 开启线程
            thread.start();
        }
    });
}
```

3.2 ThreadPool

3.2.1 为什么要使用线程池

1. 线程的创建和销毁的开销都比较大，降低资源消耗
2. 线程是可复用的，提高响应速度
3. 对多任务多线程进行管理，提高线程的可管理性

3.2.2 几种常用的线程池

1. 单个任务处理时间比较短且任务数量很大（多个线程的线程池）：
 1. **FixedThreadPool** 定长线程池
 2. **CachedThreadPool** 可缓存线程池
2. 执行定时任务（定时线程池）：
 1. **ScheduledThreadPool** 定时任务线程池
3. 特定单项任务（单线程线程池）：
 1. **SingleThreadPool** 只有一个线程的线程池

3.2.3 使用示例

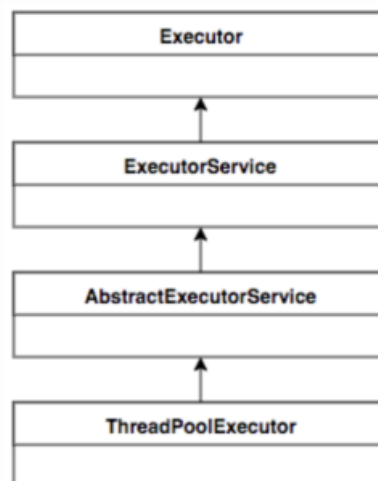
1. 接口`java.util.concurrent.ExecutorService`表述了异步执行的机制，并且可以让任务在一组线程内执行

2. 重要函数：

1. `execute(Runnable)`：向线程池提交一个任务
2. `submit(Runnable/Callable)`：有返回值（Future），可以查询任务的执行状态和执行结果
3. `shutdown()`：关闭线程池

1. 创建一个线程池`ExecutorService`的示例
2. 创建一个`Runnable`对象，并编写其业务逻辑
3. 通过`service.execute()`方法，向线程池提交任务

```
private void threadPoolTest() {  
    // 创建固定大小（线程数量为3）的线程池  
    ExecutorService service = Executors.newFixedThreadPool( nThreads: 3);  
    // 创建任务  
    Runnable task = new Runnable() {  
        @Override  
        public void run() {  
            Log.i(TAG, msg: "test thread pool");  
        }  
    };  
    // 向线程池提交任务  
    service.execute(task);  
}
```



3.3 AsyncTask(已弃用)

回到之前的例子：

用户在抖音App中，点击下载视频，下载过程中需要弹出Loading窗，下载结束后提示用户下载成功/失败。

Handler模式来实现的异步操作，代码相对臃肿，在多个任务同时执行时，不易对线程进行精确的控制。

Android提供了工具类AsyncTask，它使创建异步任务变得更加简单，不再需要编写任务线程和Handler实例即可完成相同的任务

AsyncTask

AsyncTask的定义及重要函数：

1. AsyncTask<Params, Progress, Result>：UI线程
2. onPreExecute：UI线程
3. doInBackground：非UI线程
4. publishProgress：非UI线程
5. onProgressUpdate：UI线程
6. onPostExecute：UI线程

使用方法：创建实例后调用execute方法，传入params

```
private class DownloadTask extends AsyncTask<String, Integer, String> {
    public static final String DOWNLOAD_FAIL = "download_fail";

    @Override protected void onPreExecute() {
        super.onPreExecute();
        toast(msg: "开始下载");
        showLoading();
    }

    @Override protected String doInBackground(String... strings) {
        String url = strings[0];
        try {
            return downloadVideo(url);
        } catch (Exception e) {
            return DOWNLOAD_FAIL;
        }
    }

    private String downloadVideo(String videoId) {
        int progress = 0;
        while (progress < 100) {
            publishProgress(progress);
            progress++;
        }
        return "local_url";
    }

    @Override protected void onProgressUpdate(Integer... values) {
        super.onProgressUpdate(values);
        Log.d(tag: "download", msg: "下载进度: " + values[0]);
    }

    @Override protected void onPostExecute(String s) {
        super.onPostExecute(s);
        if (DOWNLOAD_FAIL.equals(s)) {
            hideLoading();
            toast(msg: "下载失败");
        } else {
            hideLoading();
            toast(msg: "下载成功: " + s);
        }
    }
}
```

3.4 HandlerThread

1. HandlerThread的本质：继承Thread类 & 封装Handler类

1. 试想一款股票交易App：

1. 由于因为股票的行情数据都是实时变化的
2. 所以我们软件需要每隔一定时间向服务器请求行情数据

2. 该轮询调度需要放到子线程，由Handler + Looper去处理和调度

2. HandlerThread是Android API提供的一个方便、便捷的类，使用它可以快速的创建一个带有Looper的线程。Looper可以用来创建Handler实例

1. 创建一个HandlerThread对象
2. 使用handlerThread.start()方法，运行线程
3. 通过handlerThread.getLooper()方法，获取该线程的Looper
4. 通过Looper实例创建Handler，将Handler与该线程关联

```
private void handlerThreadTest() {
    // 创建一个HandlerThread
    HandlerThread handlerThread = new HandlerThread( name: "myhandlerthread");
    // 启动HandlerThread
    handlerThread.start();
    // 创建与该线程绑定的Handler
    Handler handler = new Handler(handlerThread.getLooper());
    // 使用handler向线程发送消息
    handler.post(new Runnable() {
        @Override
        public void run() {
            Log.i(TAG, msg: "current thread:" + Thread.currentThread().getName());
        }
    });
}
```

HandlerThread的源码

1. onLooperPrepared():
2. run(): 运行该线程
3. getThreadHandler():
4. quit()和quitSafely(): 停止该线程

```
public class HandlerThread extends Thread {
    int mPriority;
    int mTid = -1;
    Looper mLooper;
    private @Nullable Handler mHandler;

    public HandlerThread(String name) {...}

    /** Constructs a HandlerThread. ...*/
    public HandlerThread(String name, int priority) {...}

    /** Call back method that can be explicitly overridden if needed to execute some ...*/
    protected void onLooperPrepared() {}

    @Override
    public void run() {...}

    /** This method returns the Looper associated with this thread. If this thread not been started ...*/
    public Looper getLooper() {...}

    /** @return a shared {@link Handler} associated with this thread ...*/
    @NonNull
    public Handler getThreadHandler() {...}

    /** Quits the handler thread's looper. ...*/
    public boolean quit() {...}

    /** Quits the handler thread's looper safely. ...*/
    public boolean quitSafely() {...}

    /** Returns the identifier of this thread. See Process.myTid(). ...*/
    public int getThreadId() { return mTid; }
}
```

3.5 IntentService(不常用, 自学)

Service是执行在主线程的。
而很多情况下，我们需要做的事情可能并不希望在主线程执行，那么就应该用IntentService。
比如：用Service下载文件

那什么是IntentService？

IntentService 是 Service 的子类，它使用工作线程逐一处理所有启动请求。如果您不要求服务同时处理多个请求，这是最好的选择。

```
public class DownloadService extends IntentService {
    /**
     * Creates an IntentService.  Invoked by your subclass's constructor.
     *
     * @param name Used to name the worker thread, important only for debugging.
     */
    public DownloadService(String name) {
        super(name, "DownloadService");
    }

    @Override
    protected void onHandleIntent(@Nullable Intent intent) {
        if (intent != null) {
            try {
                String url = intent.getStringExtra("url");
                // download file from url
            } catch (Exception e) {
                e.printStackTrace();
            }
        }
    }
}
```

3.6 Android多线程总结

Thread	多线程的基础
ThreadPool	对线程进行更好的管理
AsyncTask	Android中为了简化多线程的使用，而设计的默认封装
HandlerThread	开启一个线程，就可以处理多个耗时任务
IntentService	Android中无界面异步操作的默认实现

四、自定义View

4.1 View绘制的三个重要步骤

1. Measure：测量宽高

2. **Layout**: 确定位置

3. **Draw**: 绘制形状

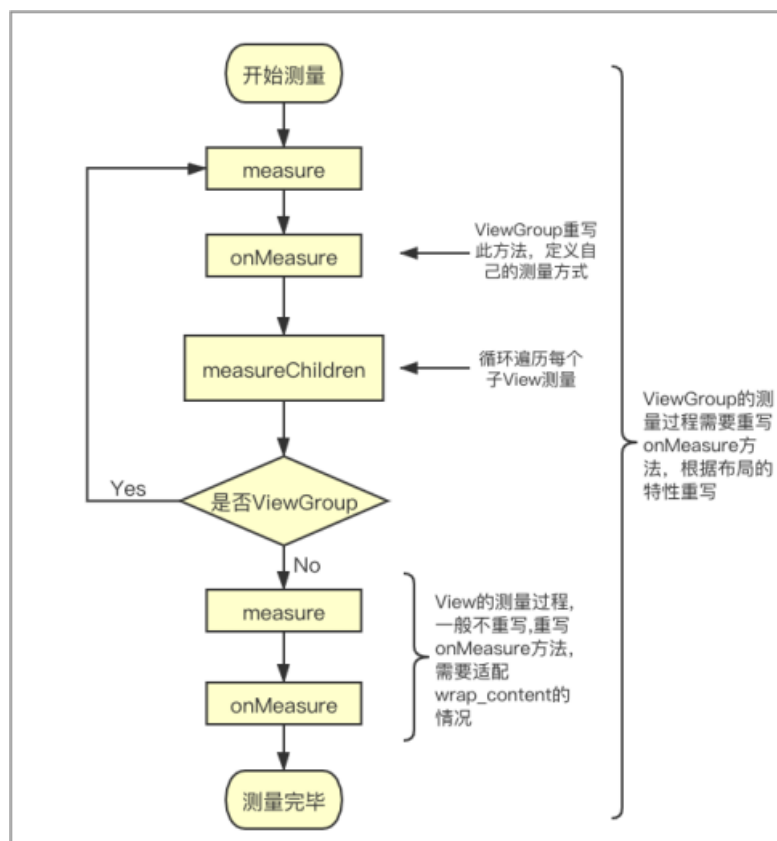
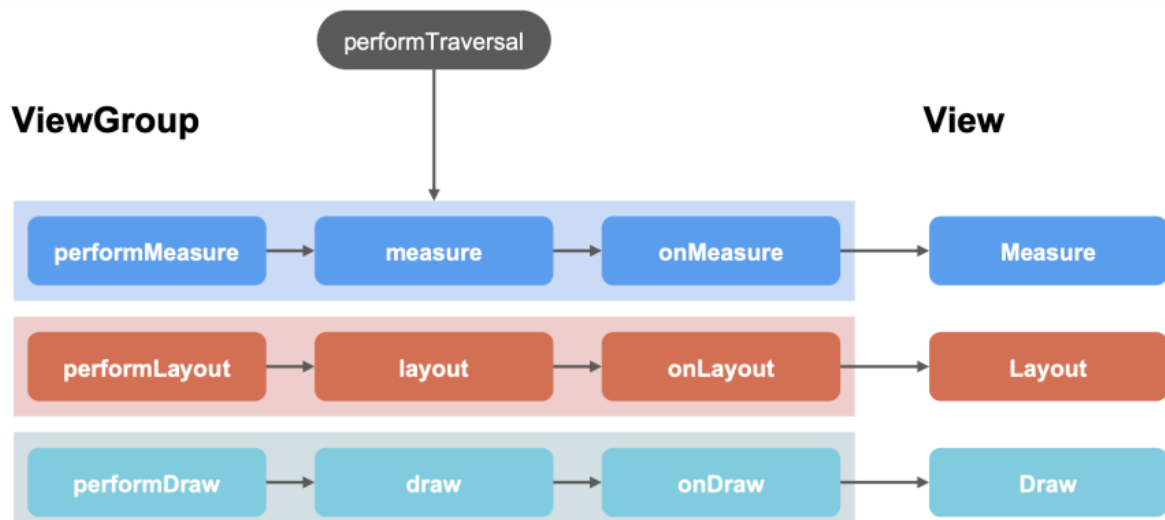
4. 举例说明:

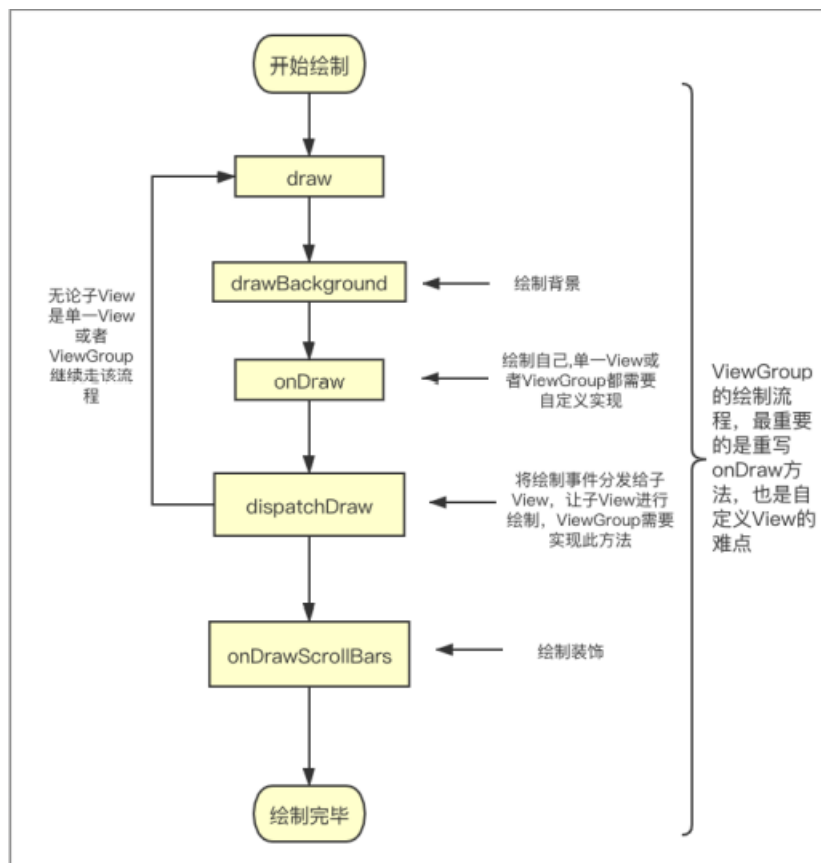
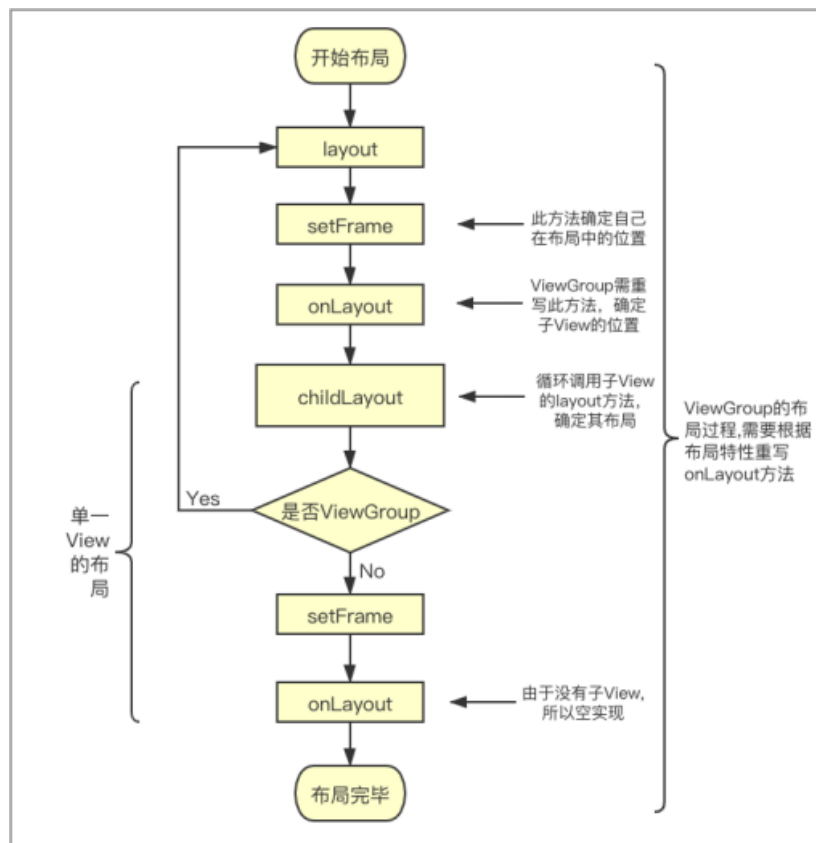
1. 首先画一个100 x 100的照片框, 需要尺子测量出宽高的长度 (measure过程)

2. 然后确定照片框在屏幕中的位置 (layout过程)

3. 最后借助尺子用手画出我们的照片框 (draw过程)

4.2 绘制流程

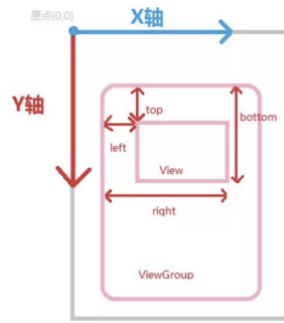




4.3 自定义View: 重写onDraw

1. Canvas: 画布
2. Paint: 画笔

3. 坐标轴:

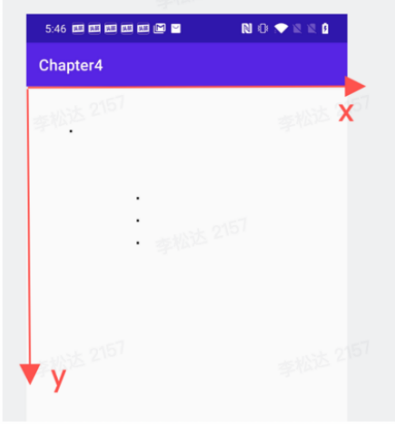


```
public class ClockView extends View {
    public ClockView(Context context) { super(context); }
    public ClockView(Context context, @Nullable AttributeSet attrs) { super(context, attrs); }
    public ClockView(Context context, @Nullable AttributeSet attrs, int defStyleAttr) {...}
    @Override
    protected void onDraw(Canvas canvas) {
        super.onDraw(canvas);
        // 自己的绘制代码
        // ...
    }
}
```



4.3.1 画点

View绘制-点



```
public ExampleView(Context context) {
    super(context);
    init();
}

public ExampleView(Context context, @Nullable AttributeSet attrs) {
    super(context, attrs);
    init();
}

public ExampleView(Context context, @Nullable AttributeSet attrs, int defStyleAttr) {
    super(context, attrs, defStyleAttr);
    init();
}


private void init() {
    mPaint = new Paint();
    mPaint.setColor(Color.BLACK);
    mPaint.setAntiAlias(true);
    mPaint.setStyle(Paint.Style.FILL);
    mPaint.setStrokeWidth(10f);
}

@Override
protected void onDraw(Canvas canvas) {
    drawPoints(canvas);
}

private void drawPoints(Canvas canvas) {
    canvas.drawPoint(x: 100, y: 100, mPaint);
    canvas.drawPoints(new float[] {
        500, 500,
        500, 600,
        500, 700
    }, mPaint);
}
```

4.3.2 画线

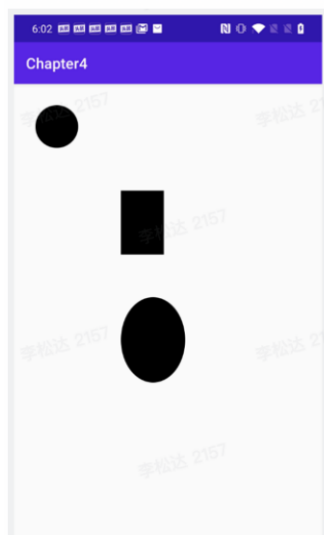
View绘制-线



```
private void drawLines(Canvas canvas) {
    canvas.drawLine( startX: 300, startY: 300, stopX: 500, stopY: 600, mPaint);
    canvas.drawLines(new float[] {
        100, 200, 200, 200,
        100, 300, 200, 300
    }, mPaint);
}
```

4.3.3 画圆

View绘制-圆



```
private void drawGraphics(Canvas canvas) {  
    canvas.drawCircle( cx: 200, cy: 200, radius: 100, mPaint);  
    canvas.drawRect( left: 500, top: 500, right: 700, bottom: 800, mPaint);  
    canvas.drawOval( left: 500, top: 1000, right: 800, bottom: 1400, mPaint);  
}
```



4.3.4 填充

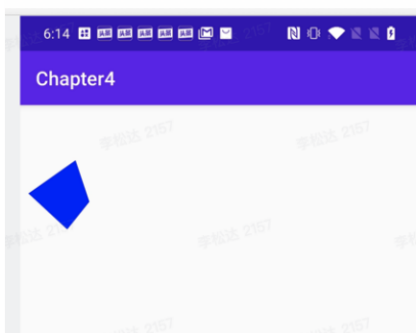
View绘制-填充



```
private void drawCircle(Canvas canvas) {  
    Paint temp = new Paint();  
    temp.setColor(Color.BLUE);  
    temp.setAntiAlias(true);  
    temp.setStyle(Paint.Style.FILL);  
    temp.setStrokeWidth(50f);  
  
    canvas.drawCircle( cx: 200, cy: 200, radius: 100, temp);  
    temp.setStyle(Paint.Style.STROKE);  
    canvas.drawCircle( cx: 200, cy: 500, radius: 100, temp);  
    temp.setStyle(Paint.Style.FILL_AND_STROKE);  
    canvas.drawCircle( cx: 200, cy: 800, radius: 100, temp);  
}
```

4.3.5 不规则图形

View绘制-不规则图形



```
private void drawPath(Canvas canvas) {  
    mPaint.setColor(Color.BLUE);  
    // 绘制多边形的类  
    Path path = new Path();  
    // 起始点  
    path.moveTo( x: 200, y: 200);  
    path.lineTo( x: 250, y: 350);  
    path.lineTo( x: 170, y: 450);  
    path.lineTo( x: 30, y: 320);  
    // 闭合图形  
    path.close();  
    canvas.drawPath(path, mPaint);  
}
```


4.3.6 画文本

View绘制-画文本



```
private void drawPathWithText(Canvas canvas) {
    mPaint.setColor(Color.BLUE);
    mPaint.setTextSize(50f);
    canvas.drawText(text: "这是一段测试文本", x: 100, y: 100, mPaint);
    // 绘制多边形的类
    Path path = new Path();
    // 起始点
    path.moveTo(x: 200, y: 200);
    path.lineTo(x: 250, y: 350);
    path.lineTo(x: 170, y: 450);
    path.lineTo(x: 30, y: 320);
    // 闭合图形
    path.close();
    mPaint.setTextSize(25f);
    canvas.drawTextOnPath(text: "这是第二段测试文本, 测试的内容是使用canvas画出一段文本",
        path, hOffset: 0, vOffset: 0, mPaint);
}
```

View绘制-画文本



```
private void drawText(Canvas canvas) {
    mPaint.setColor(Color.BLUE);
    mPaint.setTextSize(50f);
    mPaint.setTextAlign(Paint.Align.LEFT);
    canvas.drawText(text: "这是一段测试文本", x: 500, y: 500, mPaint);
    mPaint.setTextAlign(Paint.Align.CENTER);
    canvas.drawText(text: "这是一段测试文本", x: 500, y: 700, mPaint);
    mPaint.setTextAlign(Paint.Align.RIGHT);
    canvas.drawText(text: "这是一段测试文本", x: 500, y: 900, mPaint);
}
```

4.4 自定义View总结

1. 重要绘制流程:

1. **Measure**: 测量
2. **Layout**: 布局
3. **Draw**: 绘制

2. 以及几个重要函数:

1. **invalidate**
2. **requestLayout**

3. 理解ViewTree 及 ViewGroup 的Measure / Layout / Draw的流程

4. View自定义绘制:

1. 绘制图形: 点、线、圆形、椭圆、矩形、圆角矩形
2. 绘制文字