

# 复杂应用组件

何睿 [herui.ryon@bytedance.com](mailto:herui.ryon@bytedance.com)

字节跳动Android工程师



## 提纲

- Handler机制（Android的消息队列机制）
- Android中的多线程
- 自定义View

# Handler机制



# 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x秒；在x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。

# Handler概念

Handler机制为Android系统解决了以下两个问题:

1. 调度 (Schedule) Android系统在某个时间点执行特定的任务
  - a. `Message(android.os.Message)`
  - b. `Runnable(java.lang.Runnable)`
2. 将需要执行的任务加入到用户创建的线程的任务队列中

From Android Developer Website:

There are two main uses for a Handler:

- (1) to schedule messages and runnables to be executed at some point in the future;
- (2) to enqueue an action to be performed on a different thread than your own.

# 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

```
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);
```

// 取消消息

```
public final void removeCallbacks(Runnable r);
public final void removeMessages(int what);
public final void removeCallbacksAndMessages(Object token);
```

# Handler的使用

- 调度Message
  - ✓ 新建一个Handler, 实现handleMessage()方法
  - ✓ 在适当的时候给上面的Handler发送消息
- 调度Runnable
  - ✓ 新建一个Handler, 然后直接调度Runnable即可
- 取消调度
  - ✓ 通过Handler取消已经发送过的Message/Runnable

# Handler的使用举例

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 runnable = new Runnable() {
    @Override
    public void run() {
        // 跳转首页
        jumpToMainActivity();
    }
};
handler.postDelayed(runnable, delayMillis: 3000);
```



# Handler的使用举例

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。如果用户点击了跳过，则应该直接进入主界面。

```
final Handler handler = new Handler();
final Runnable runnable = new Runnable() {
    @Override
    public void run() {
        // 跳转首页
        jumpToMainActivity();
    }
};
handler.postDelayed(runnable, delayMillis: 3000);
mSkipButton.setOnClickListener(new View.OnClickListener() {
    @Override
    public void onClick(View v) {
        handler.removeCallbacks(runnable);
        jumpToMainActivity();
    }
});
```

# Handler的使用举例

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。

补充知识点：

Android中，UI控件并非是线程安全的，只能在主线程内调用，所以所有对于UI控件的调用，必须在主线程。

因此，通常我们也把主线程也叫做UI线程。

# 进程与线程

进程（Process）是一个具有一定独立功能的程序关于某个数据集合的一次运行活动。它是操作系统动态执行的基本单元，在传统的操作系统中，进程既是基本的分配（资源）单元，也是基本的执行（调度）单元。

一般情况下，android中的一个app是一个进程，如果需要多进程，需要手动开启。

线程（Thread）是操作系统能够进行运算调度的最小单位。它被包含在进程之中，是进程中的实际运作单位。

# Handler的使用举例

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。

```
View mDownloadButton = findViewById(R.id.mSkipButton);
mDownloadButton.setOnClickListener((v) -> {
    new DownloadThread(mVideoId).start();
});
}

private class DownloadThread extends Thread {
    String mVideoId;
    public DownloadThread(String videoId) { this.mVideoId = videoId; }
    @Override
    public void run() {
        mHandler.sendMessage(Message.obtain(mHandler, MSG_START_DOWNLOAD));
        try {
            String videoPath = downloadVideo(mVideoId);
            mHandler.sendMessage(Message.obtain(mHandler, MSG_DOWNLOAD_SUCCESS, videoPath));
        } catch (Exception e) {
            mHandler.sendMessage(Message.obtain(mHandler, MSG_DOWNLOAD_FAIL));
        }
    }
}

private String downloadVideo(String videoId) {}
}
```

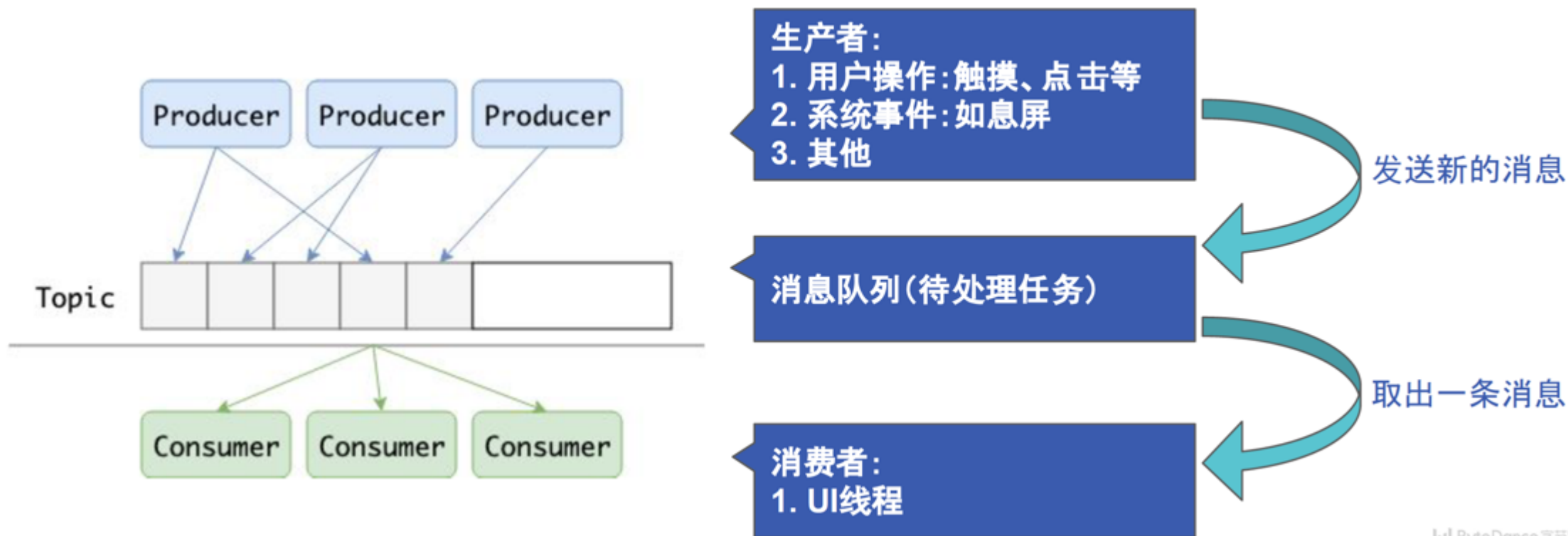
# Handler的使用举例

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。

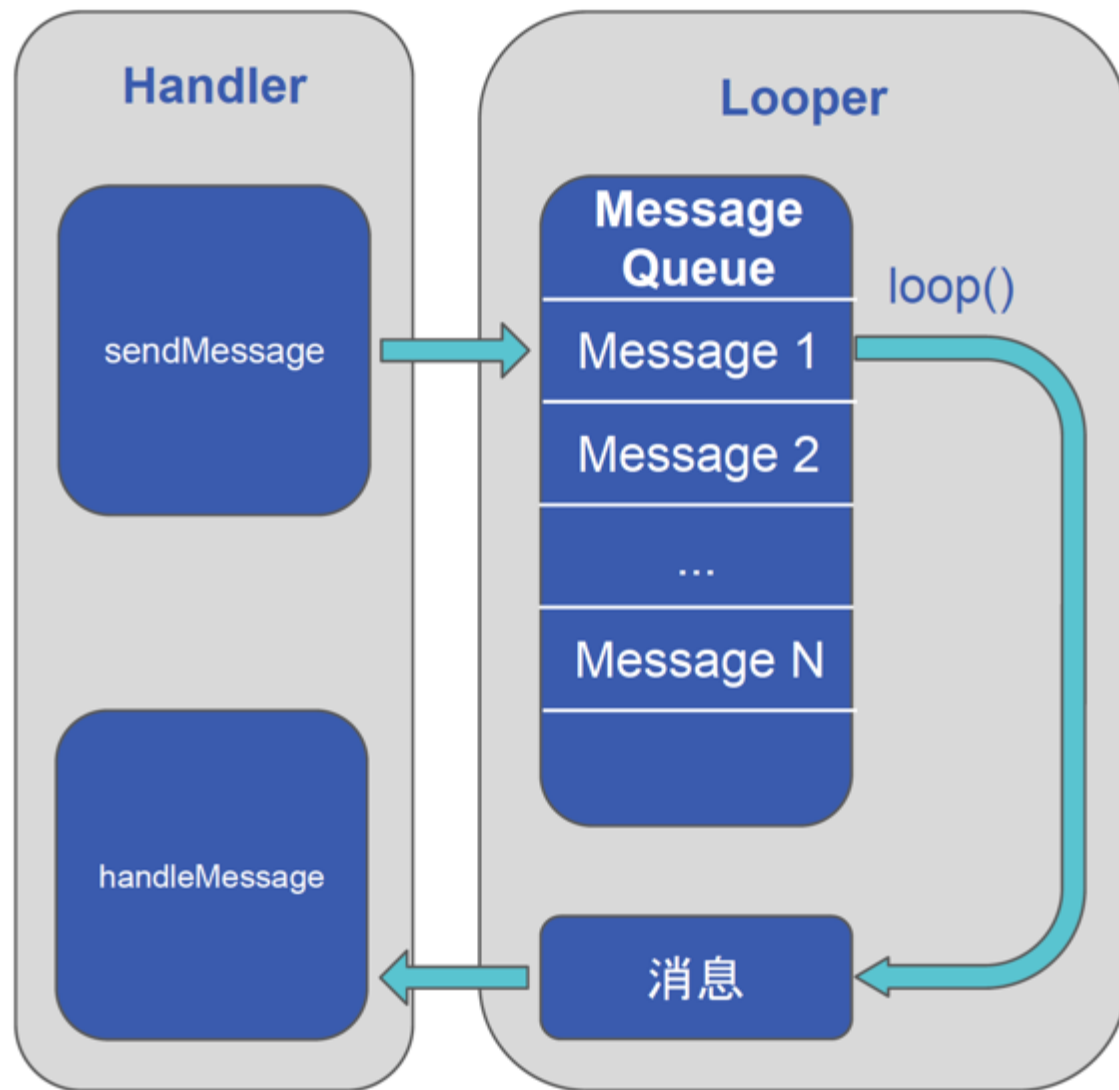
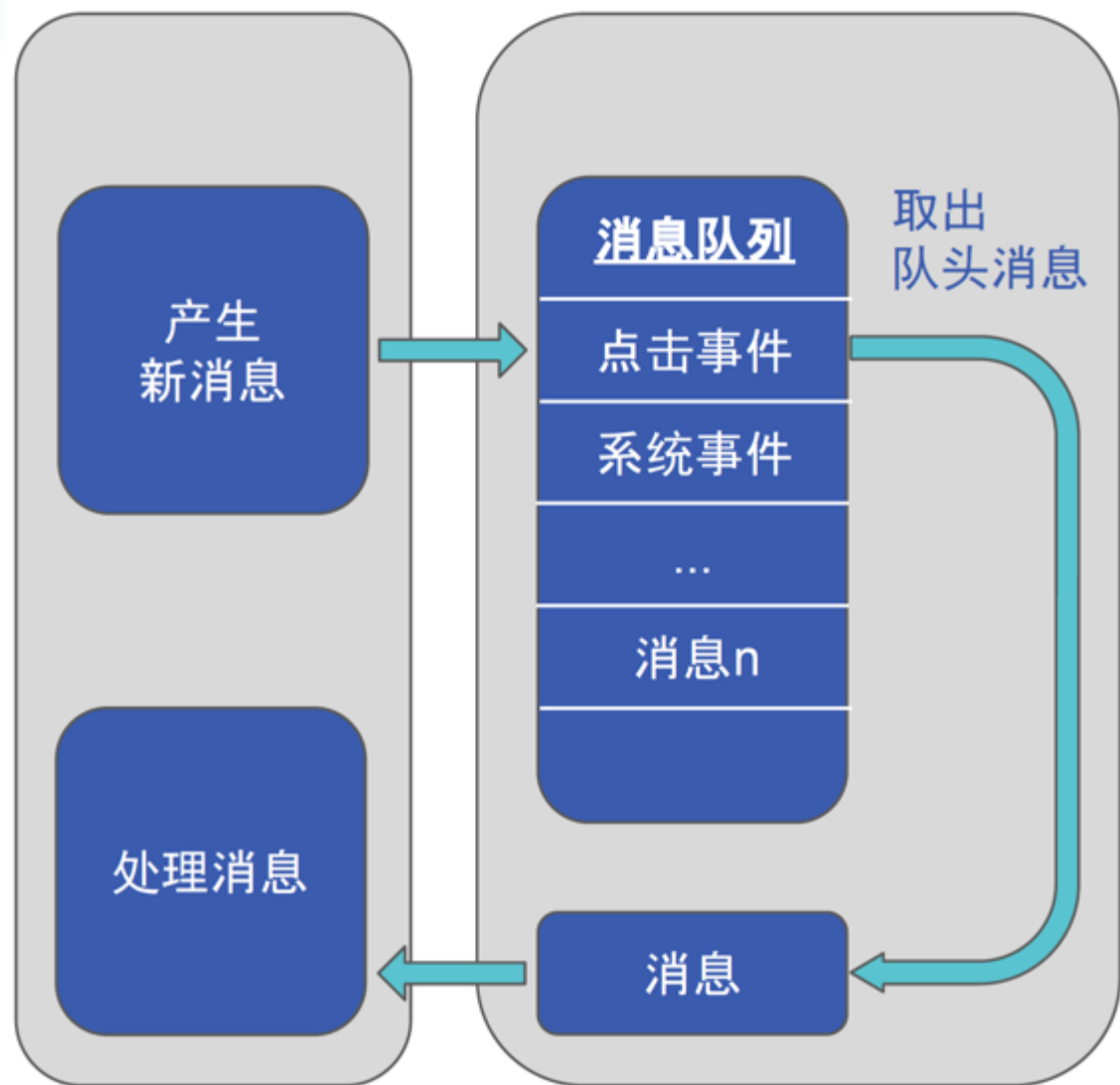
```
private static final int MSG_START_DOWNLOAD = 1;
private static final int MSG_DOWNLOAD_SUCCESS = 2;
private static final int MSG_DOWNLOAD_FAIL = 3;
private Handler mHandler = new Handler() {
    @Override
    public void handleMessage(@NonNull Message msg) {
        super.handleMessage(msg);
        switch (msg.what) {
            case MSG_START_DOWNLOAD:
                toast(msg: "开始下载");
                showLoading();
                break;
            case MSG_DOWNLOAD_SUCCESS:
                toast(msg: "下载成功");
                hideLoading();
                break;
            case MSG_DOWNLOAD_FAIL:
                toast(msg: "下载失败");
                hideLoading();
                break;
            default:
                break;
        }
    }
};
```

# Handler原理：UI线程与消息队列机制

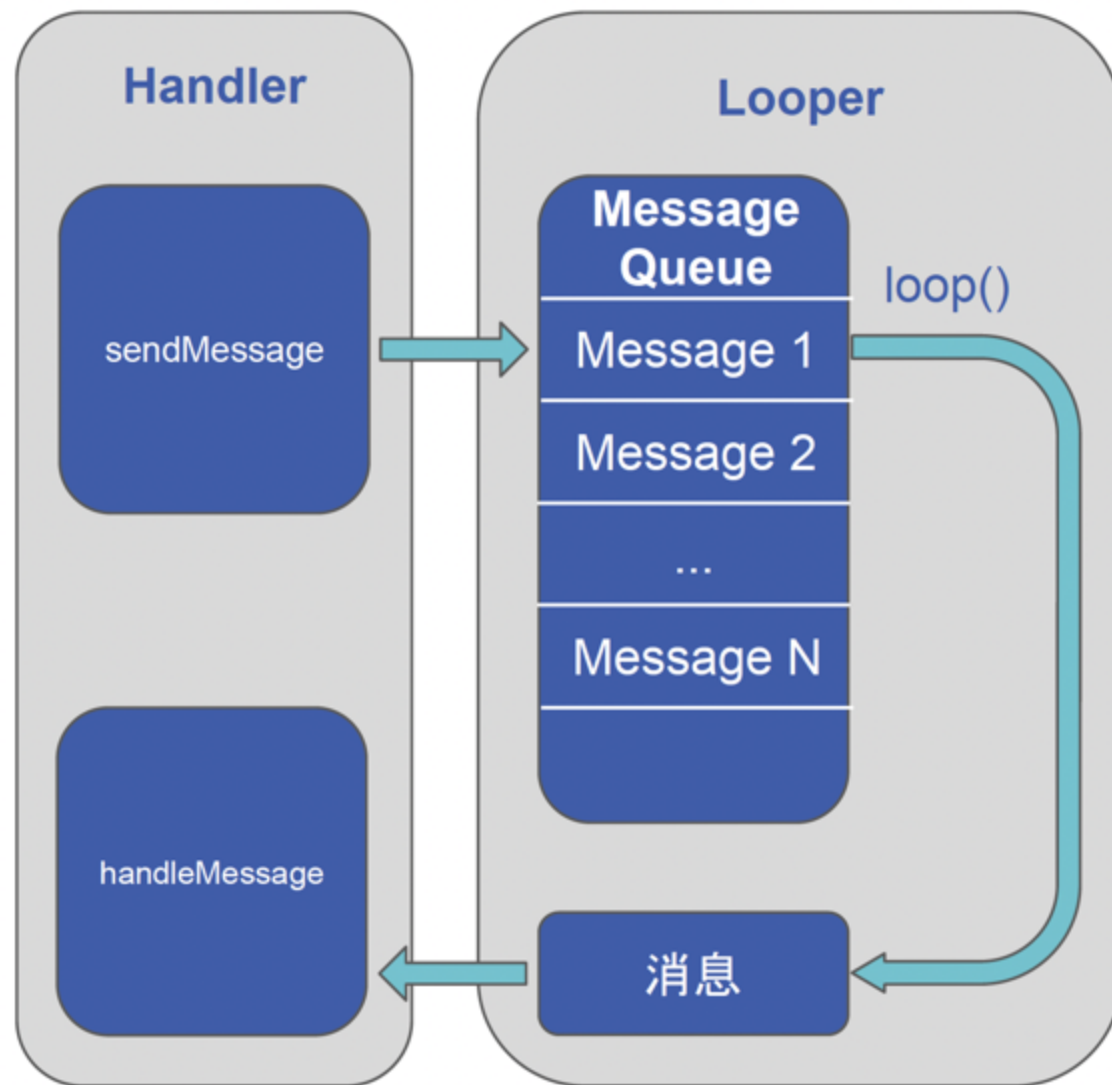
Android中，UI线程负责处理界面的展示，响应用户的操作：







- 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。





# 辨析Runnable/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

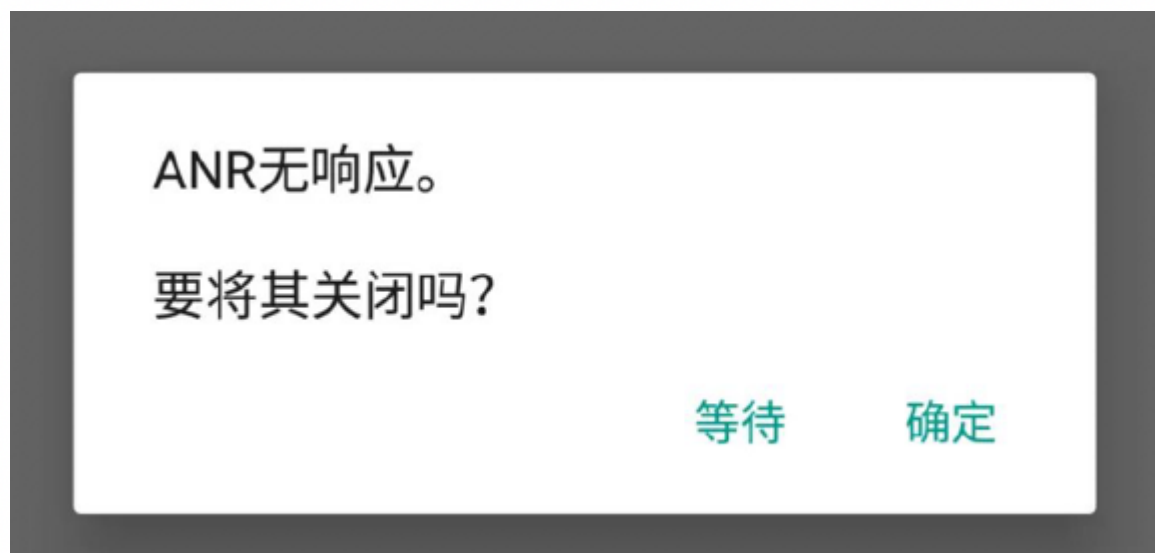
```
Handler handler = new Handler();
Runnable runnable = new Runnable() {
    @Override
    public void run() {
        // 跳转首页
        jumpToMainActivity();
    }
};
handler.postDelayed(runnable, delayMillis: 3000);
```

=

```
Handler handler = new Handler() {
    @Override
    public void handleMessage(@NonNull Message msg) {
        super.handleMessage(msg);
        if (msg.what == MSG_GO_MAIN_ACTIVITY) {
            // 跳转首页
            jumpToMainActivity();
        }
    }
};
handler.sendMessageDelayed(
    Message.obtain(handler, MSG_GO_MAIN_ACTIVITY),
    delayMillis: 3000);
```

## 扩展：ANR

主线程（UI线程）不能执行耗时操作，否则会出现 ANR (Application Not Responding)



## PROCESS

### Main Thread (UI Thread)

System  
Events

Input  
Events

Application

Service

Alarm

UI Drawing

( 主线程执行了太多任务，可能比你预想的要忙很多 )

## PROCESS

### Main Thread (UI Thread)

App  
Events

UI  
Drawing

UI  
Drawing

Input  
Event

UI  
Drawing

16ms

16ms

16ms

( 其中每一帧内容的绘制其实只有16ms )

# Handler总结

- ✓ 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；
- ✓ Handler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；
- ✓ 掌握Handler的基本用法：立即/延时/定时发送消息、取消消息；



# Android多线程



# Android中的常用线程

✓ Thread

✓ ThreadPool

✓ HandlerThread \*

✓ IntentService \*

✓ AsyncTask \*(deprecated)

# Thread

```
class MyThread extends Thread{  
    @Override  
    public void run() {  
        super.run();  
        // do something  
    }  
}
```

一个简单的Thread的例子

```
Thread thread = new Thread() {  
    @Override  
    public void run() {  
        super.run();  
        while(!isInterrupted()){  
            // do something  
        }  
    }  
};  
thread.start();  
thread.interrupt();
```

怎样优雅的启动和停止一个Thread





# ThreadPool

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。

重要函数：

- `execute(Runnable)`
- `submit(Runnbale)`: 有返回值（Future），可以cancel，更方便进行错误处理
- `shutdown()`





# ThreadPool

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

# ThreadPool

介绍几种常用的线程池：

- 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）：
  - 网络库：FixedThreadPool 定长线程池
  - DB操作：CachedThreadPool 可缓存线程池
- 执行定时任务（定时线程池）：
  - 定时上报性能日志数据：ScheduledThreadPool 定时任务线程池
- 特定单项任务（单线程线程池）：
  - 日志写入：SingleThreadPool 只有一个线程的线程池

# HandlerThread

HandlerThread的本质：继承Thread类 & 封装Handler类

试想一款股票交易App：

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

这个轮询的请求的调度是否可以放到非主线程，由Handler + Looper去处理和调度？

# HandlerThread

```
public class StockHandlerThread extends HandlerThread implements Handler.Callback {
    public static final int MSG_QUERY_STOCK = 100;
    // 与工作线程相关的Handler
    private Handler mHandler;

    public StockHandlerThread(String name) {
        super(name);
    }

    public StockHandlerThread(String name, int priority) {
        super(name);
    }

    @Override
    protected void onLooperPrepared() {
        mHandler = new Handler(getLooper(), callback: this);
        // 首次请求
        mHandler.sendMessage(MSG_QUERY_STOCK);
    }

    @Override
    public boolean handleMessage(@NonNull Message msg) {
        if (msg.what == MSG_QUERY_STOCK) {
            // 请求股票数据
            // ...
            // 回调主线程或者写入数据库
            // ...
            // 10s后再次请求
            mHandler.sendMessageDelayed(MSG_QUERY_STOCK, delayMillis: 10 * 1000);
        }
        return true;
    }
}
```

# HandlerThread

源码：

```
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; }
}
```

# 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

# Android自定义view



# View绘制的三个重要步骤

Measure: 测量宽高

Layout: 确定位置

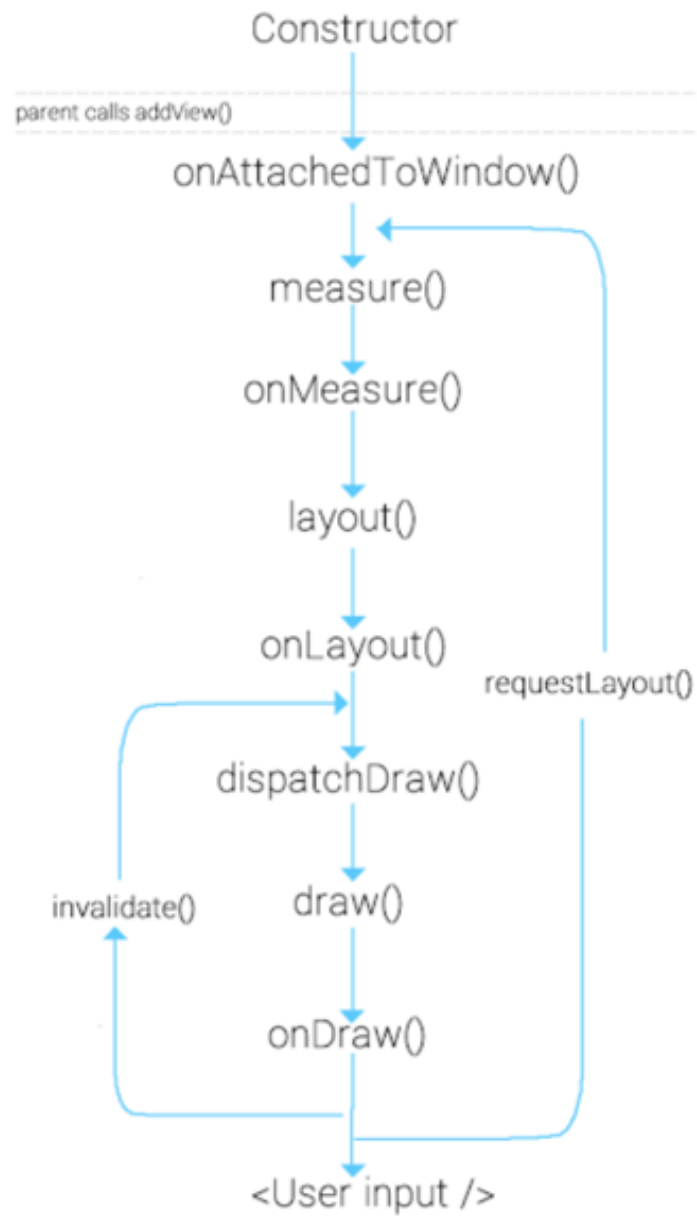
Draw : 绘制形状

举例说明:

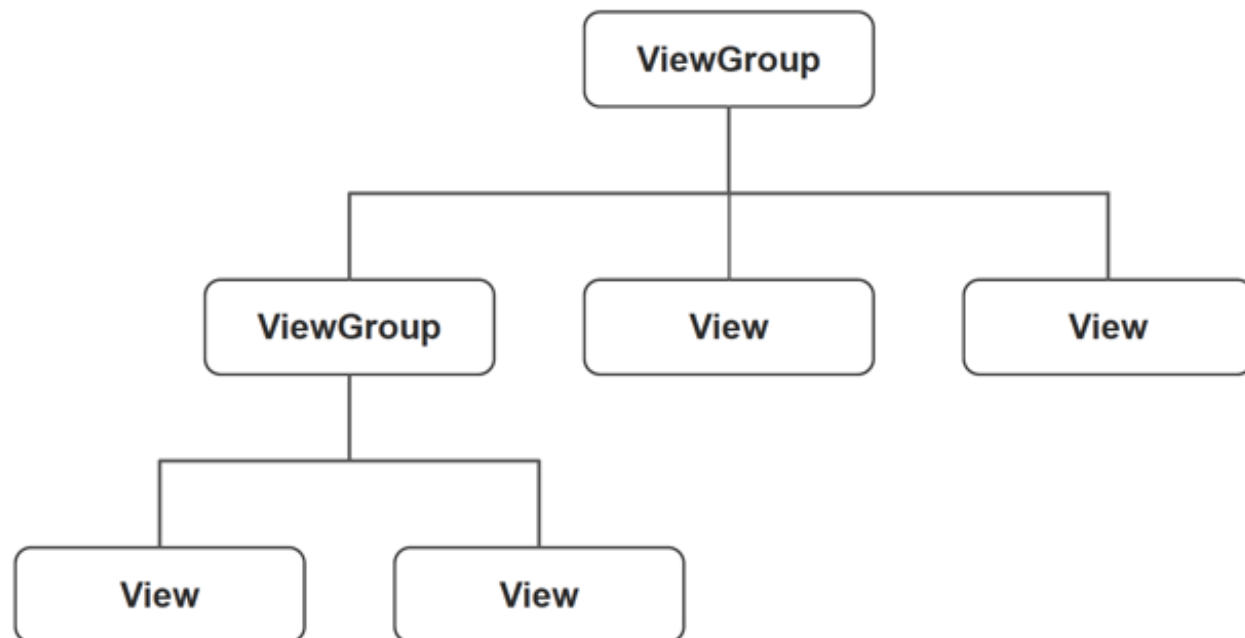
1. 首先画一个100 x 100的照片框, 需要尺子测量出宽高的长度 (measure过程)
2. 然后确定照片框在屏幕中的位置 (layout过程)
3. 最后借助尺子用手画出我们的照片框 (draw过程)



# View绘制的三个重要步骤

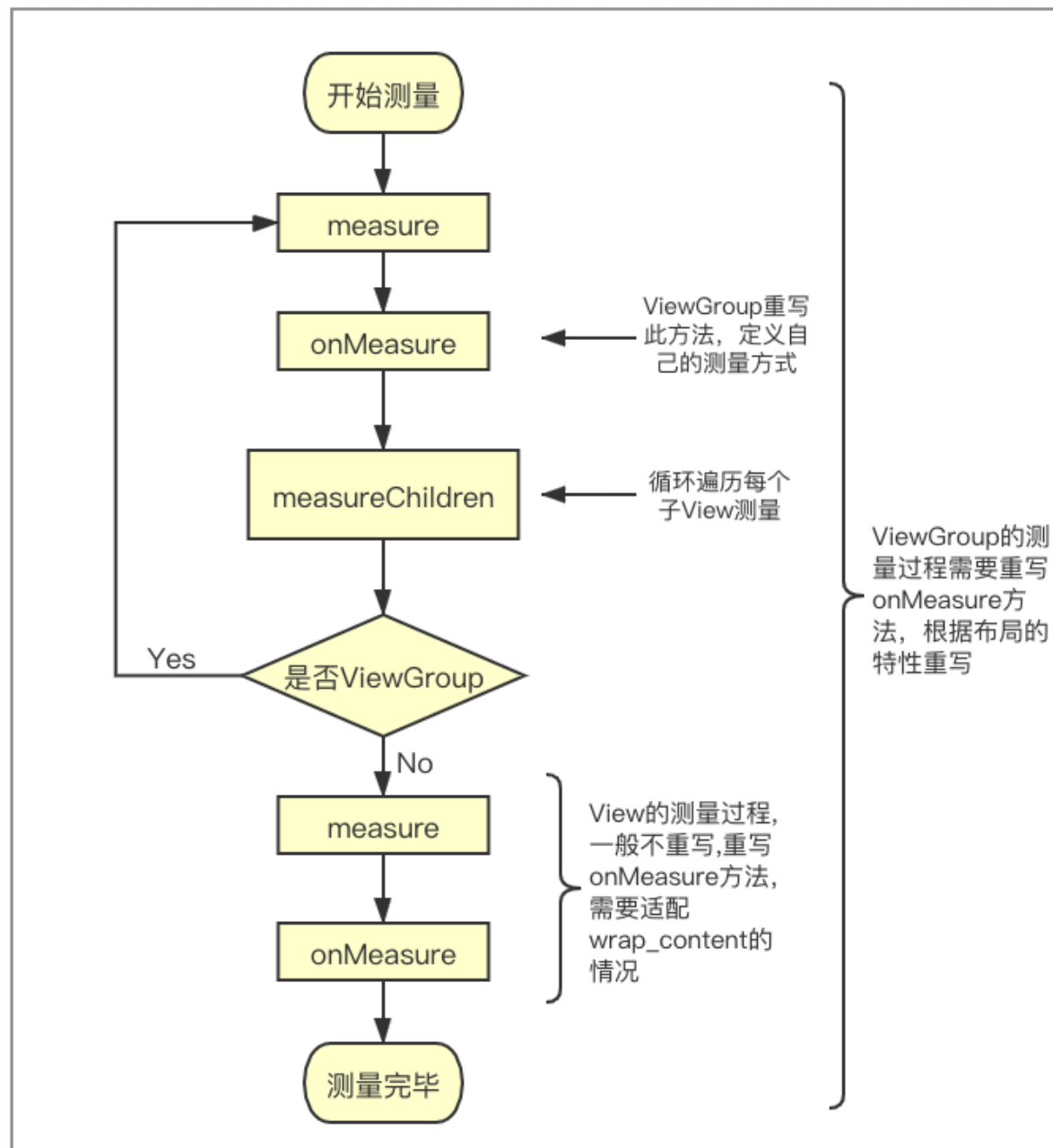


## 扩展：详解 ViewTree 及 View / ViewGroup 绘制流程

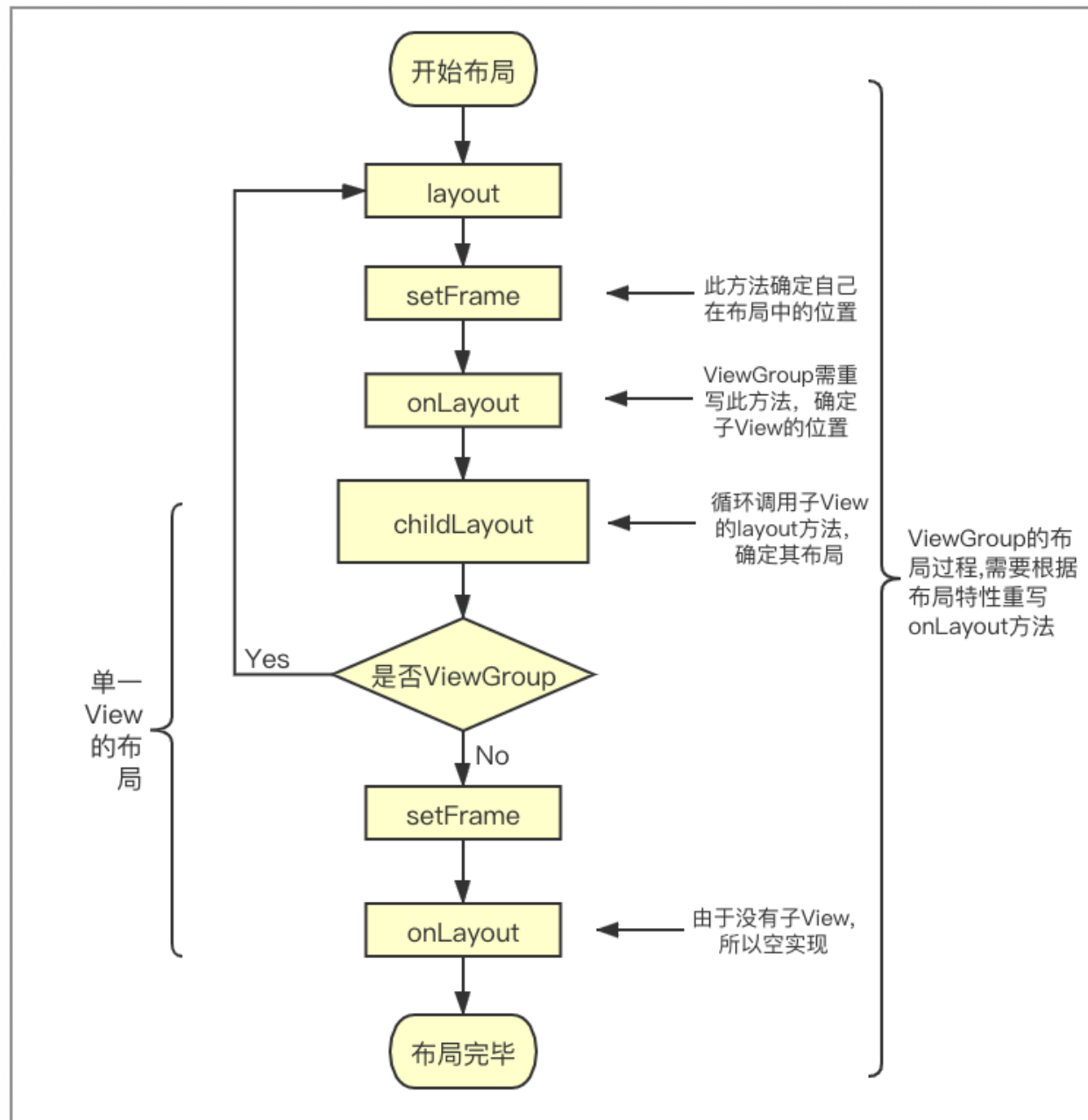


自上而下递归进行：  
Measure  
Layout  
Draw

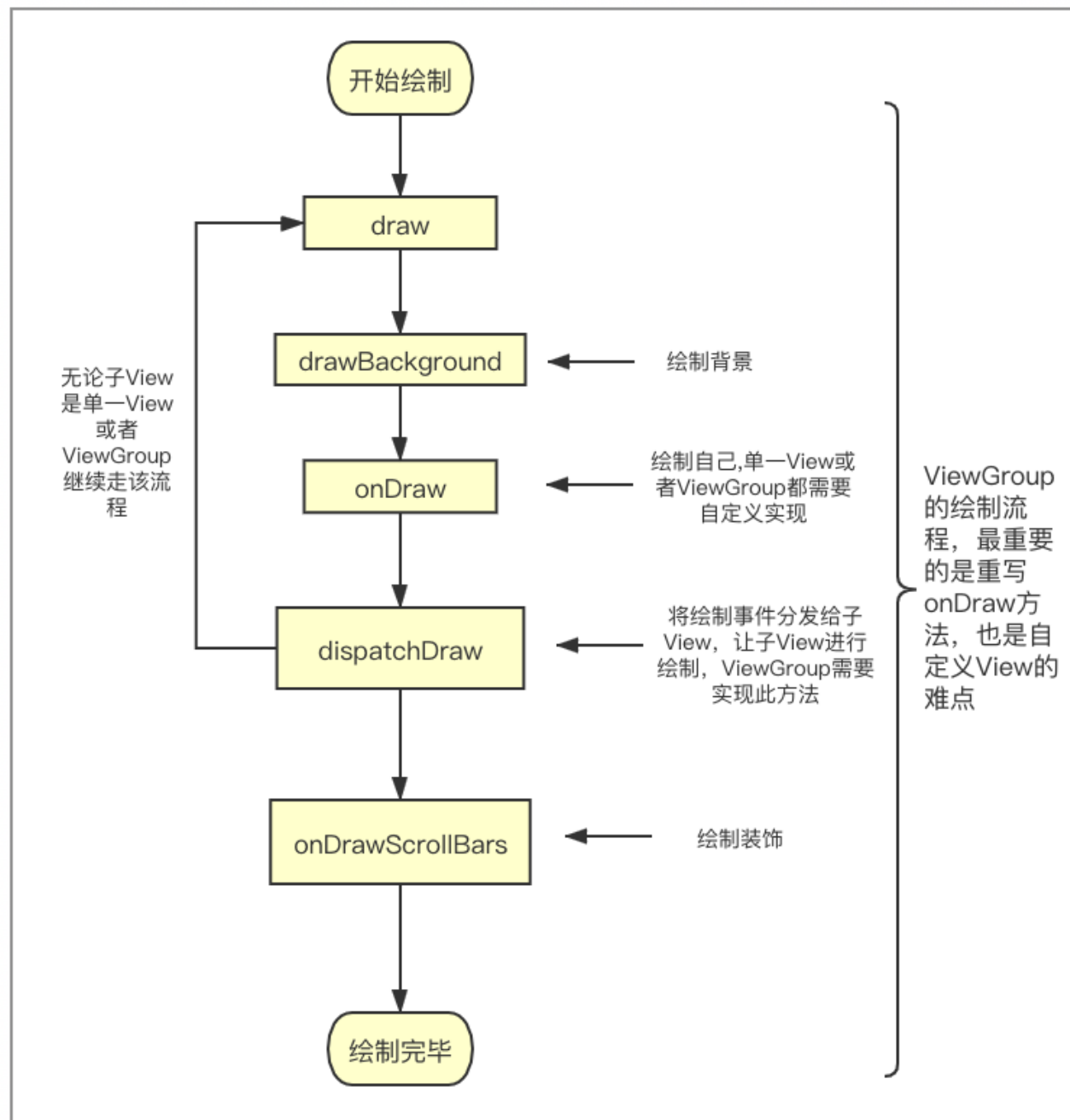
## 扩展： ViewGroup 绘制



## 扩展： ViewGroup 绘制



# ViewGroup绘制



# 自定义View-重写onDraw

自定义View最常见操作 - 重写onDraw

```
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  
        // ...  
    }  
}
```

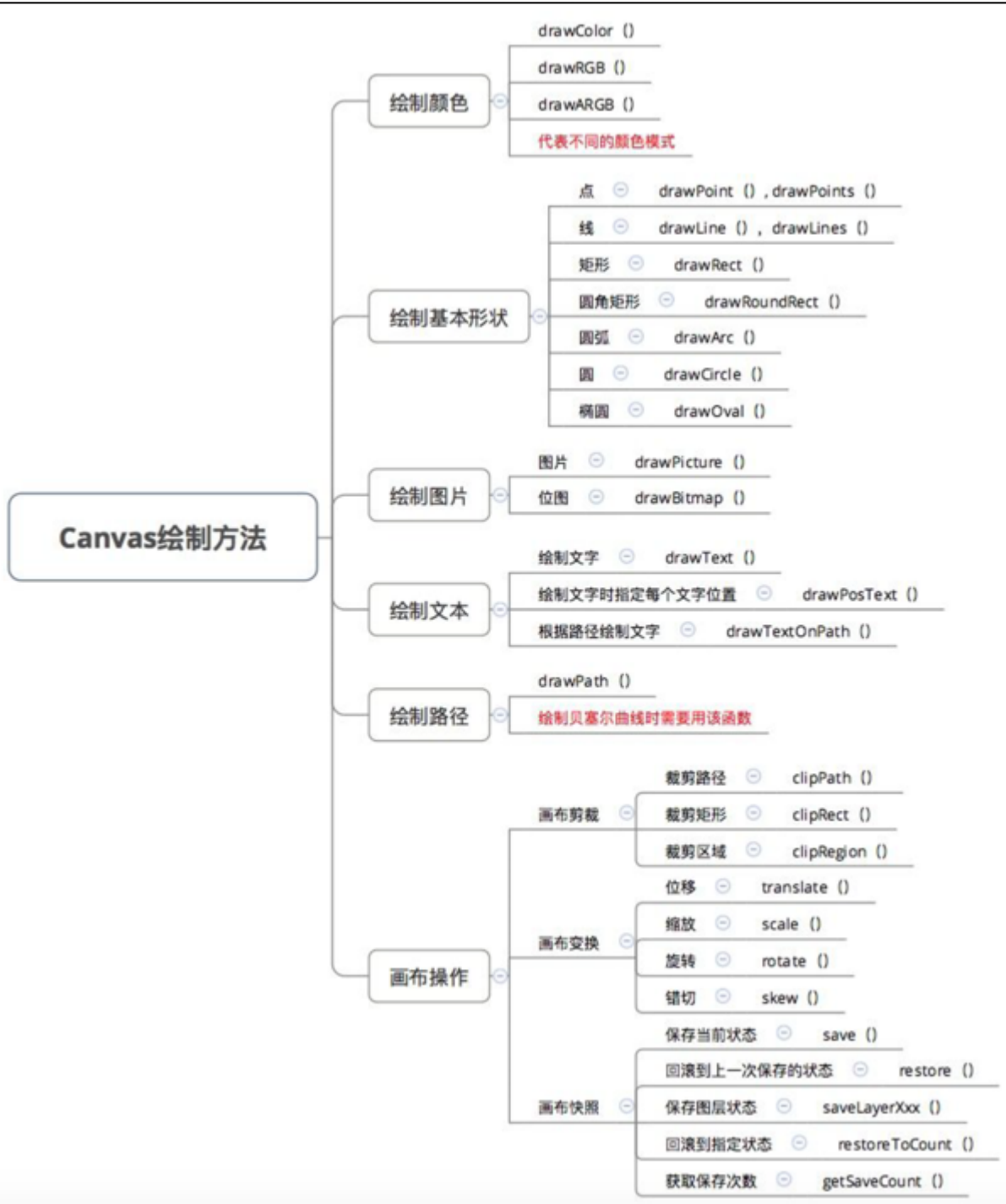
# 自定义View-重写onDraw

自定义View最常见操作 - 重写onDraw

概念解析：

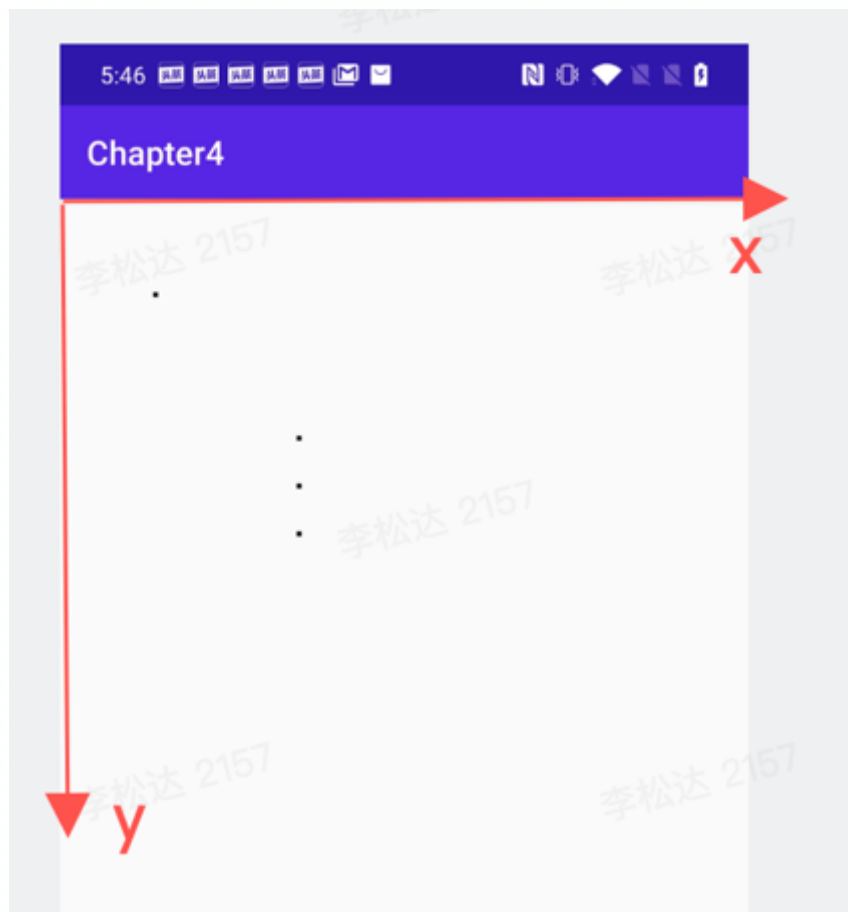
1. Canvas：画布

2. Paint：画笔





# View绘制-点



```
public class CustomView extends View {  
    private Paint mPaint;  
    public CustomView(Context context) {  
        super(context);  
        init();  
    }  
    public CustomView(Context context, @Nullable AttributeSet attrs) {  
        super(context, attrs);  
        init();  
    }  
    public CustomView(Context context, @Nullable AttributeSet attrs, int defStyleAttr) {  
        super(context, attrs, defStyleAttr);  
        init();  
    }  
    private void init() {  
        mPaint = new Paint();  
        mPaint.setColor(Color.BLACK);  
        mPaint.setStyle(Paint.Style.FILL);  
        mPaint.setAntiAlias(true);  
        mPaint.setStrokeWidth(10f);  
    }  
    @Override  
    protected void onDraw(Canvas canvas) {  
        super.onDraw(canvas);  
        canvas.drawPoint(x: 200, y: 200, mPaint);  
        canvas.drawPoints(new float[]{  
            500, 500,  
            500, 600,  
            500, 700  
        }, mPaint);  
    }  
}
```

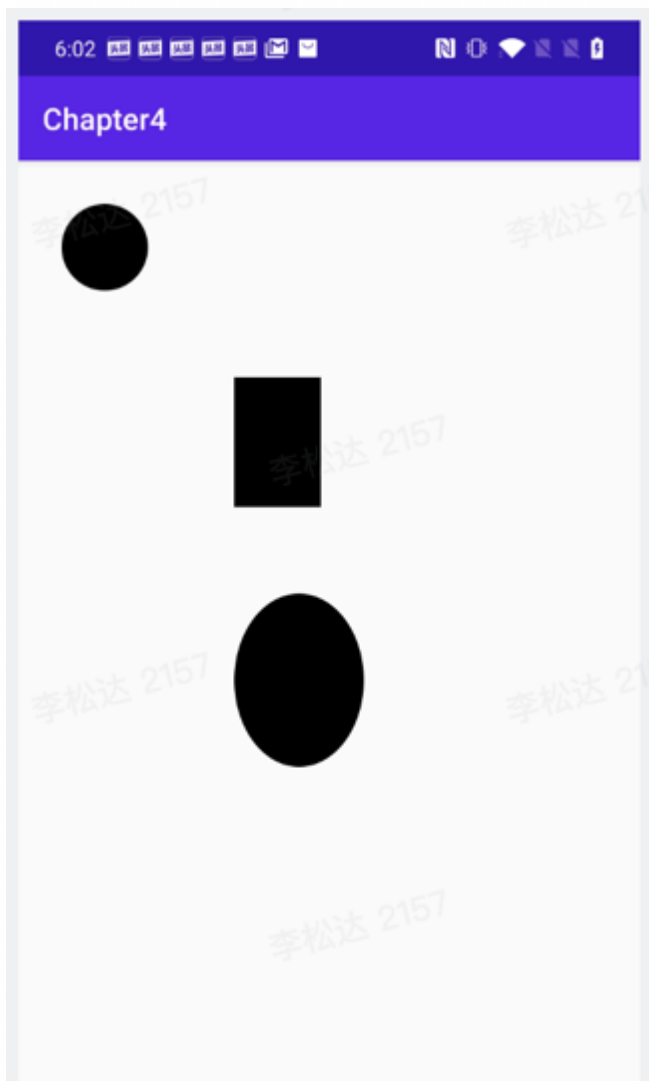


# View绘制-线



```
@Override
protected void onDraw(Canvas canvas) {
    super.onDraw(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);
}
```

# View绘制-圆



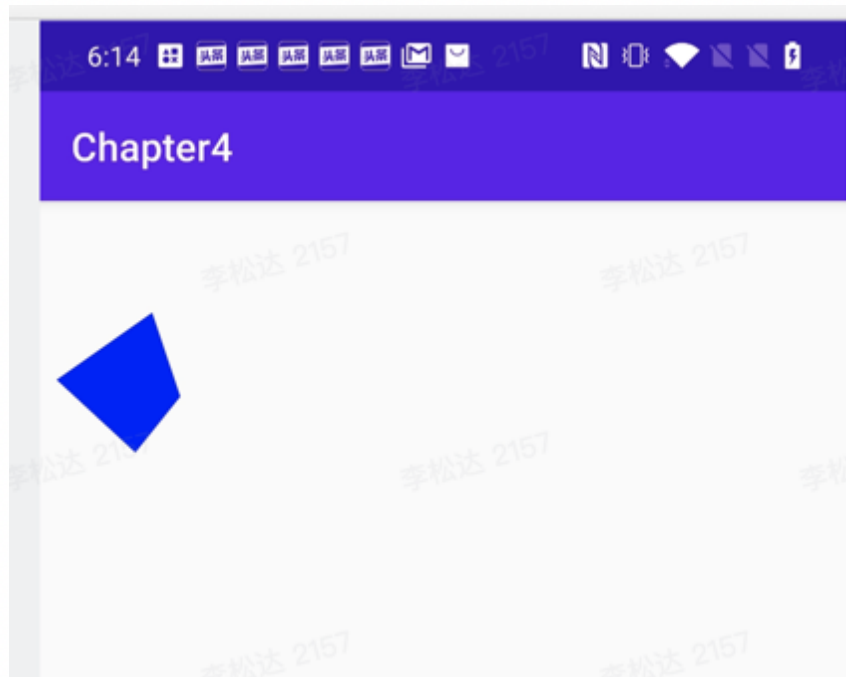
```
@Override
protected void onDraw(Canvas canvas) {
    super.onDraw(canvas);
    canvas.drawCircle(cx: 200, cy: 200, radius: 100, mPaint);
    canvas.drawRect(left: 500, top: 500, right: 700, bottom: 800, mPaint);
    canvas.drawOval(left: 500, top: 1000, right: 800, bottom: 1400, mPaint);
}
```

# View绘制-填充



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

# View绘制-不规则图形



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

# View绘制-画文本



```
@Override
protected void onDraw(Canvas canvas) {
    super.onDraw(canvas);
    mPaint.setTextSize(50f);
    canvas.drawText(text: "这是一段测试文本", x: 100, y: 100, mPaint);
    Path path = new Path();//绘制多边形的类
    path.moveTo(x: 200, y: 200);//起始点
    path.lineTo(x: 250, y: 350);
    path.lineTo(x: 170, y: 450);
    path.lineTo(x: 30, y: 320);
    path.close();//闭合图形
    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



```
@Override
protected void onDraw(Canvas canvas) {
    super.onDraw(canvas);
    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);
}
```



# 自定义view总结

View的绘制流程：

- 重要绘制流程：
  - ✓ Measure：测量
  - ✓ Layout：布局
  - ✓ Draw：绘制
- 以及几个重要函数：
  - ✓ onSizeChanged
  - ✓ invalidate
  - ✓ requestLayout
- 理解 ViewTree 及 ViewGroup 的Measure / Layout / Draw的流程
- View自定义绘制：
  - ✓ 绘制图形：点、线、圆形、椭圆、矩形、圆角矩形
  - ✓ 绘制文字：文字的测量



# 课堂作业

# 时钟App

作业：

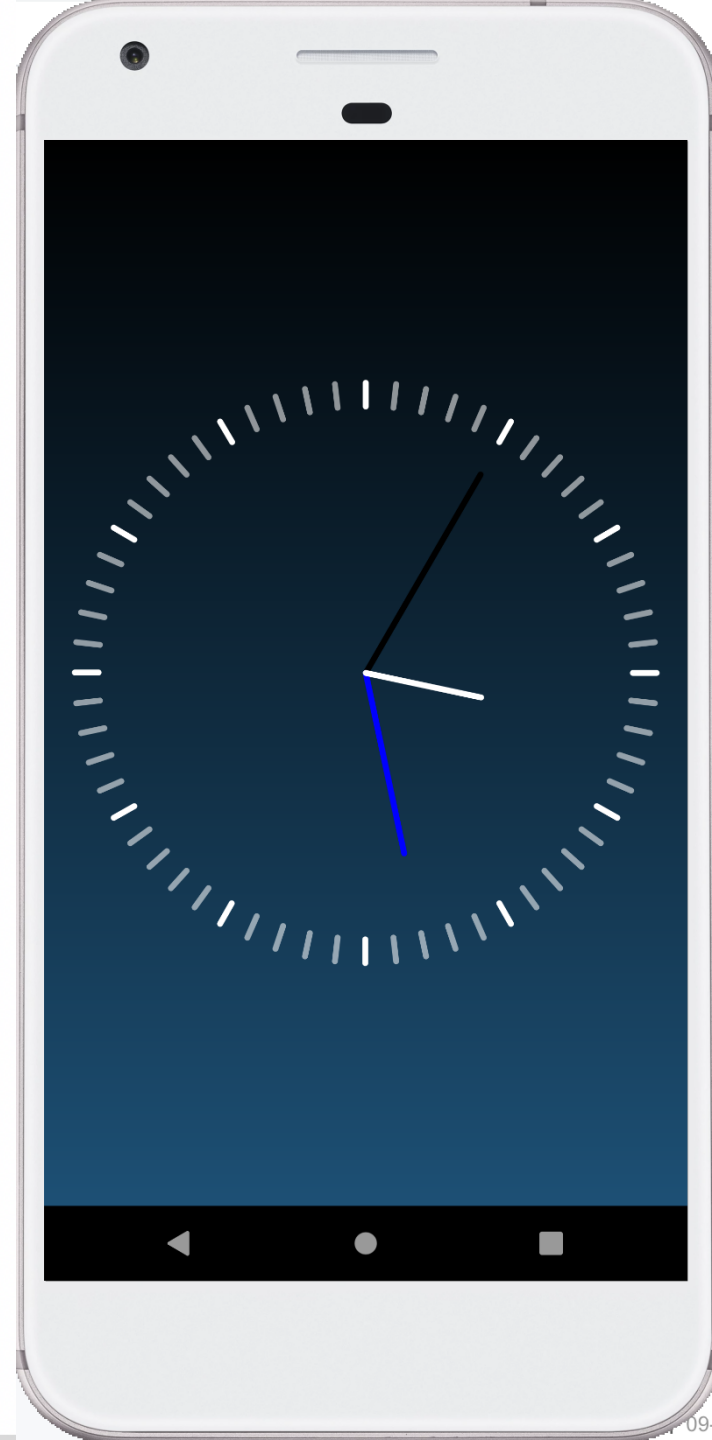
1. 绘制时钟界面，包括表盘、时针、分针、秒针
2. 时针、分针、秒针需要跳动

减分项：

1. 程序会在某些情况下崩溃
2. 逻辑过于复杂

<https://github.com/bytedance-android-camp-sjtu-2021/chapter-4>

在Clock.java类中完成todo部分





THANKS.