

复杂应用组件

Handler机制、多线程 与自定义View

邢雨辰 字节跳动Android工程师



●● 提纲

01 | Handler机制(Android的消息队列机制)

02 | Android中的多线程

03 | 自定义View

Handler机制 (Android的消息队列机制)



Handler 是做什么的？

先看这样两个例子：

1. 今日头条App启动时，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窗，下载结束后提示用户下载成功/失败。

你需要使用Handler !

Handler机制

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; and (2) to enqueue an action to be performed on a different thread than your own.

Handler的使用举例

今日头条App启动时, 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

```
mHandler.postDelayed(new Runnable() {  
    @Override  
    public void run() {  
        goMainActivity();  
    }  
}, delayMillis: 1000);
```

Handler的使用举例

今日头条App启动时, 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; **如果用户点击了跳过, 则应该直接进入主界面。**

```
mHandler.postDelayed(new Runnable() {  
    @Override  
    public void run() {  
        goMainActivity();  
    }  
}, delayMillis: 1000);  
  
mSkipView.setOnClickListener(new View.OnClickListener() {  
    @Override  
    public void onClick(View v) {  
        mHandler.removeCallbacksAndMessages( token: null);  
        goMainActivity();  
    }  
});
```

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。

```
public final int MSG_DOWN_FAIL = 1;
public final int MSG_DOWN_SUCCESS = 2;
public final int MSG_DOWN_START = 3;

private Handler mHandler = new Handler() {
    public void handleMessage(Message msg) {
        switch (msg.what) {
            case MSG_DOWN_FAIL:
                hideLoading();
                toast( msg: "下载失败");
                break;
            case MSG_DOWN_SUCCESS:
                hideLoading();
                toast( msg: "下载成功\n文件已保存在: " + msg.obj);
                break;
            case MSG_DOWN_START:
                toast( msg: "开始下载");
                showLoading();
                break;
        }
    }
};

private void initView() {
    mDownloadButton.setOnClickListener(new View.OnClickListener() {
        @Override
        public void onClick(View v) {
            new DownloadVideoThread(mVideoId).start();
        }
    });
}

public class DownloadVideoThread extends Thread {

    private String mVideoId;

    public DownloadVideoThread(String videoId) {...}

    @Override
    public void run() {
        //发送消息给 mHandler
        mHandler.sendMessage(MSG_DOWN_START);
        try {
            String localPath = downloadVideo(mVideoId);
            mHandler.sendMessage(Message.obtain(mHandler, MSG_DOWN_SUCCESS, localPath));
        } catch (Throwable t) {
            mHandler.sendMessage(Message.obtain(mHandler, MSG_DOWN_FAIL));
        }
    }

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




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

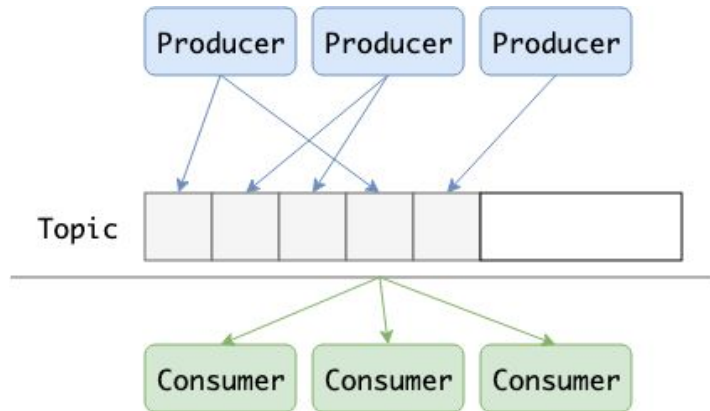
```
public final void removeCallbacks(Runnable r);
public final void removeMessages(int what);
public final void removeCallbacksAndMessages(Object token);
```

Handler原理:消息队列机制

- 在计算机科学中,消息队列(英语:Message Queue)是一种进程间通信或同一进程的不同线程间的**通信方式**。

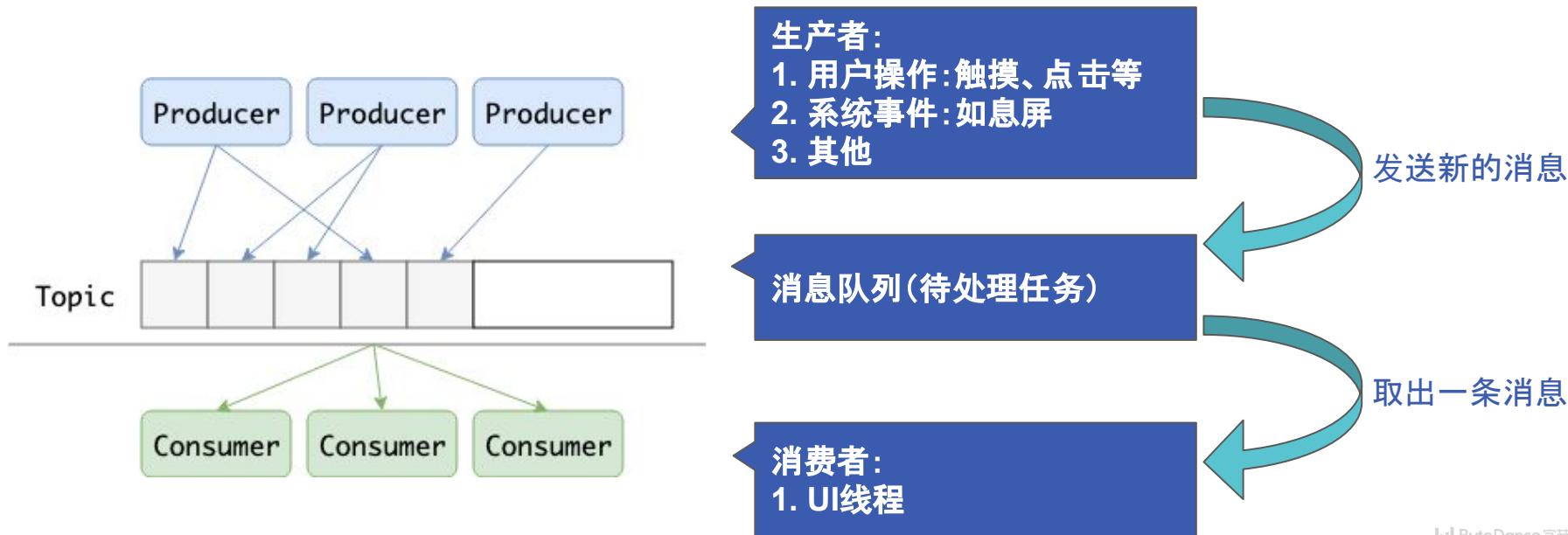
消息队列实际应用:

- Kafka分布式消息处理系统
- JS线程池模型
- Windows/Android UI线程消息处理



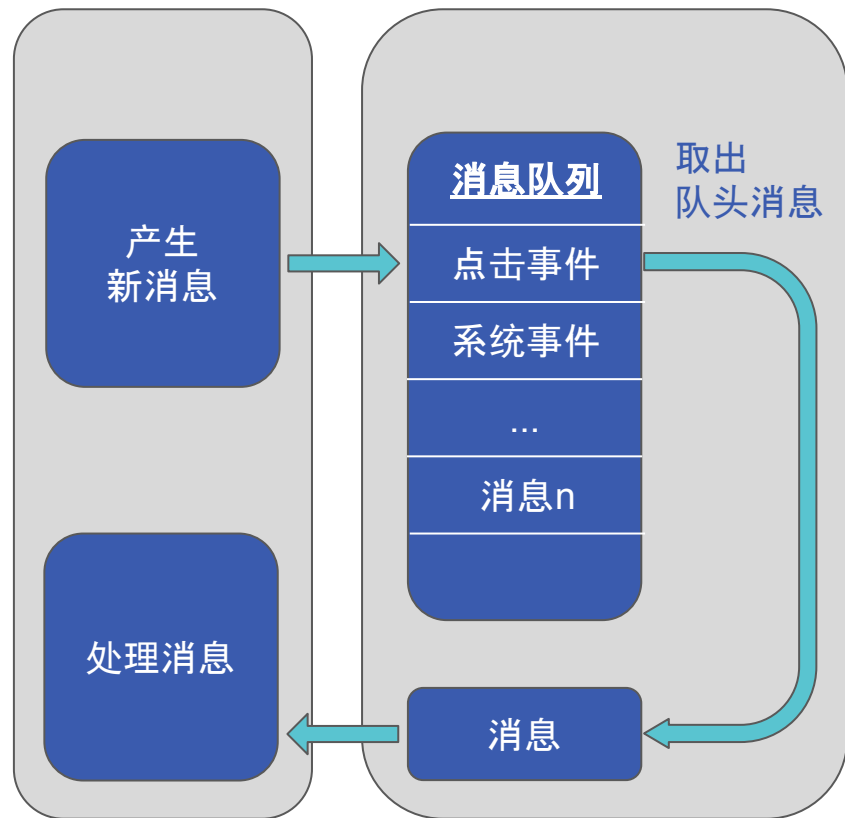
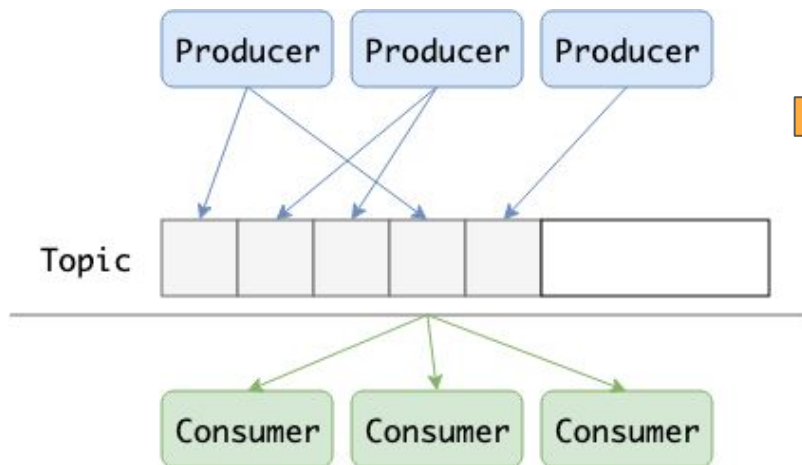
Handler原理: UI线程与消息队列机制

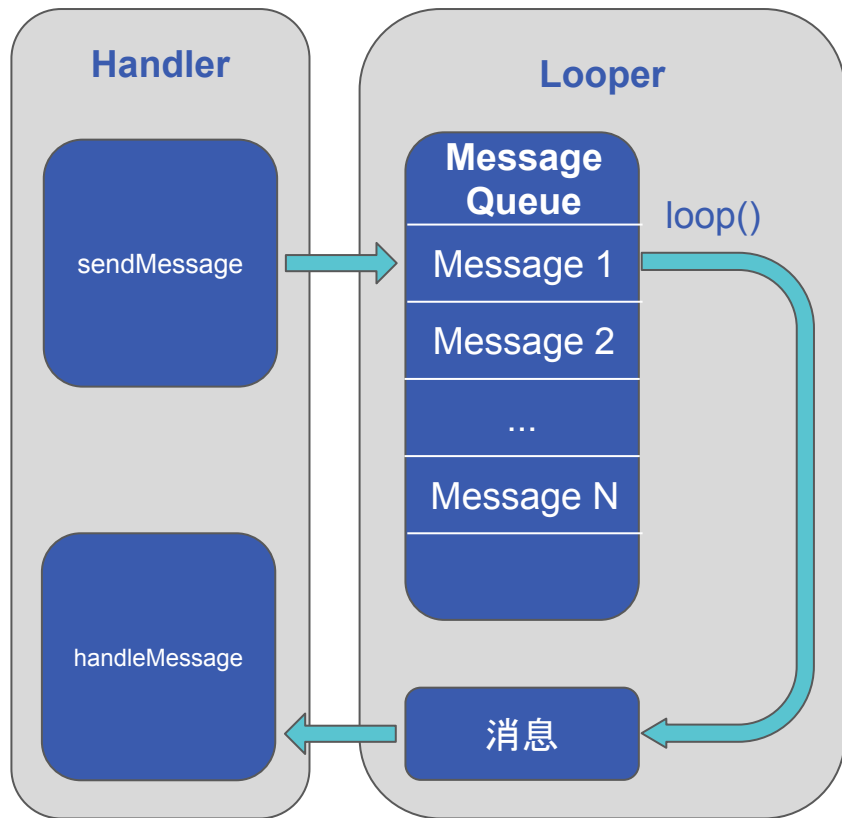
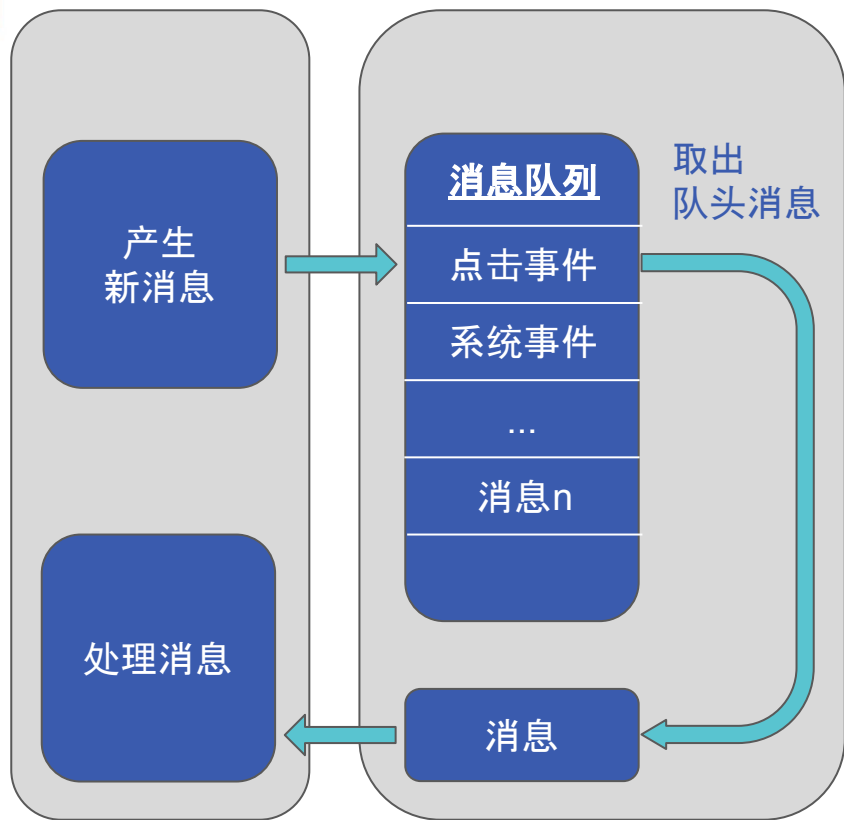
- Windows/Android中,
UI线程负责处理界面的展示, 响应用户的操作:



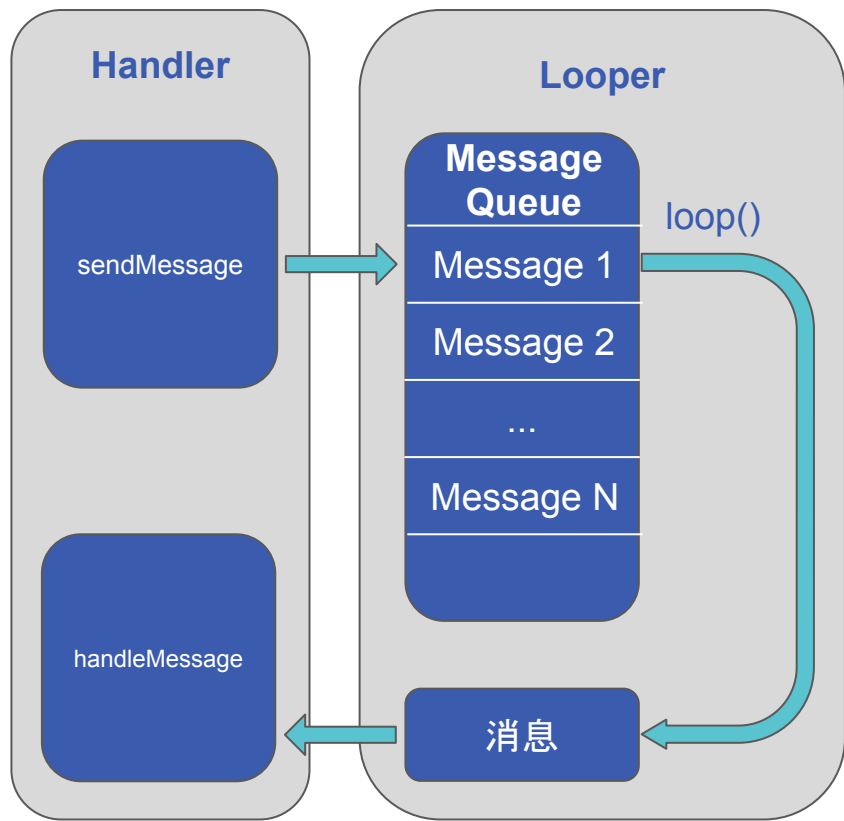
Handler原理: UI线程与消息队列机制

- Windows/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发送过来的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

```
mHandler.postDelayed(new Runnable() {  
    @Override  
    public void run() {  
        goMainActivity();  
    }  
}, delayMillis: 1000);
```

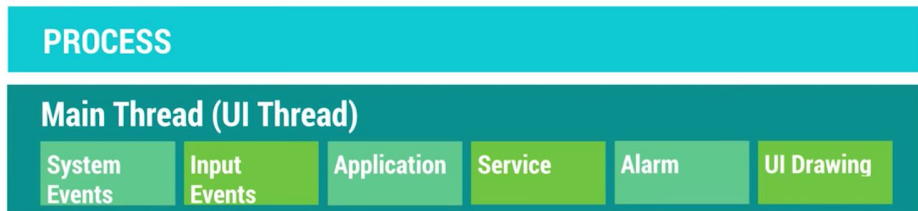
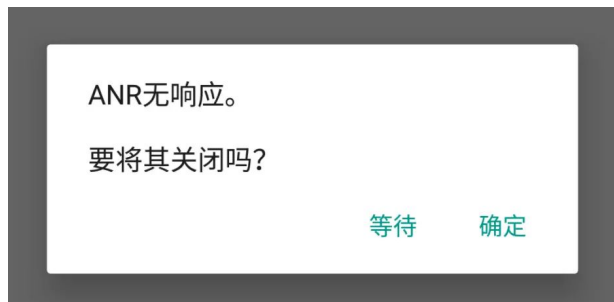
=

```
protected final Handler mHandler = new Handler() {  
    @Override  
    public void handleMessage(Message msg) {  
        super.handleMessage(msg);  
        switch (msg.what) {  
            case MSG_GO_MAIN_ACTIVITY:  
                goMainActivity();  
                break;  
        }  
    }  
};
```

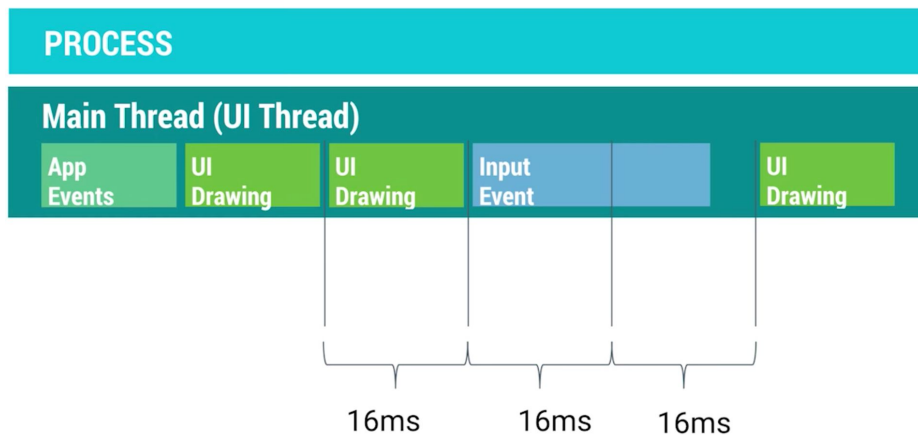
```
mHandler.sendMessageDelayed(  
    Message.obtain(mHandler, MSG_GO_MAIN_ACTIVITY),  
    delayMillis: 1000);
```


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)



(主线程执行了太多任务, 可能比你预想的要忙很多)



(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中的多线程



回顾进程的产生





进程(Process)

- **进程**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。

线程的产生及定义



- 线程的定义：
 - 线程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。



Android提供了五种常用的操作多线程的方式

01 | Thread(线程)

02 | ThreadPool(线程池)

03 | AsyncTask

04 | IntentService

05 | RxJava-Schedulers

Thread

- Thread (java.lang.Thread)

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

一个简单的Thread的例子

```
public class InterruptAThread extends Thread {  
  
    @Override  
    public void run() {  
        super.run();  
        // 判断状态, 如果被打断则跳出并将线程置空  
        while (!isInterrupted()){  
            // do something  
        }  
    }  
}  
  
private void howToStopAThread() {  
    InterruptAThread thread = new InterruptAThread();  
    // Start Thread  
    thread.start();  
    // Stop thread  
    thread.interrupt();  
}
```

怎样优雅的启动和停止一个 Thread

扩展: HandlerThread (Android特有)

- 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 ?

这时可以使用HandlerThread

```
public class StockHandlerThread extends HandlerThread implements Handler.Callback {

    public static final int MSG_QUERY_STOCK = 100;

    private Handler mWorkerHandler; //与工作线程相关联的Handler

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

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

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

    @Override
    public boolean handleMessage(Message msg) {
        switch (msg.what) {
            case MSG_QUERY_STOCK:
                // 请求股票数据
                // ...
                // 回调到主线程或写入DB
                // ...
                // 10s后再次请求
                mWorkerHandler.sendMessageDelayed(MSG_QUERY_STOCK, delayMillis: 10 * 1000);
                break;
        }
        return true;
    }
}
```

扩展: HandlerThread

(Handler的实现如右图所示)

```
/**
 * Handy class for starting a new thread that has a looper. The looper can then be
 * used to create handler classes. Note that start() must still be called.
 */
public class HandlerThread extends Thread {
    int mPriority;
    int mTid = -1;
    Looper mLooper;
    private @Nullable Handler mHandler;

    public HandlerThread(String name) {...}

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

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

    @Override
    public void run() {
        mTid = Process.myTid();
        Looper.prepare();
        synchronized (this) {
            mLooper = Looper.myLooper();
            notifyAll();
        }
        Process.setThreadPriority(mPriority);
        onLooperPrepared();
        Looper.loop();
        mTid = -1;
    }

    /**...*/
    public Looper getLooper() {...}

    /**...*/
    @NonNull
    public Handler getThreadHandler() {...}

    /**...*/
    public boolean quit() {...}

    /**...*/
    public boolean quitSafely() {...}

    /**...*/
    public int getThreadId() { return mTid; }
}
```



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. 新建和销毁线程，如此一来会大大降低系统的效率
2. 而线程是可以重用的



ThreadPool的使用

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: ScheduledThreadPoolExecutor 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

AsyncTask

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

```
public final int MSG_DOWN_FAIL = 1;
public final int MSG_DOWN_SUCCESS = 2;
public final int MSG_DOWN_START = 3;

private Handler mHandler = new Handler() {
    public void handleMessage(Message msg) {
        switch (msg.what) {
            case MSG_DOWN_FAIL:
                hideLoading();
                toast( msg: "下载失败");
                break;
            case MSG_DOWN_SUCCESS:
                hideLoading();
                toast( msg: "下载成功\n文件已保存在: " + msg.obj);
                break;
            case MSG_DOWN_START:
                toast( msg: "开始下载");
                showLoading();
                break;
        }
    }
};

private void initView() {
    mDownloadButton.setOnClickListener(new View.OnClickListener() {
        @Override
        public void onClick(View v) {
            new DownloadVideoThread(mVideoId).start();
        }
    });
}

public class DownloadVideoThread extends Thread {
    private String mVideoId;

    public DownloadVideoThread(String videoId) {...}

    @Override
    public void run() {
        //发送消息给 mHandler
        mHandler.sendMessage(Message.obtain(mHandler, MSG_DOWN_START));
        try {
            String localPath = downloadVideo(mVideoId);
            mHandler.sendMessage(Message.obtain(mHandler, MSG_DOWN_SUCCESS, localPath));
        } catch (Throwable t) {
            mHandler.sendMessage(Message.obtain(mHandler, MSG_DOWN_FAIL));
        }
    }

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

```

private void initView2() {
    mDownloadButton.setOnClickListener(new View.OnClickListener() {
        @Override
        public void onClick(View v) {
            new DownloadAsyncTask().execute(mVideoId);
        }
    });
}

private class DownloadAsyncTask extends AsyncTask<String, Integer, String> {

    final static String DOWNLOAD_FAILED = "DOWNLOAD_FAILED";

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

    @Override
    protected String doInBackground(String... args) {
        String videoId = args[0];
        try {
            return downloadVideo(videoId);
        } catch (Throwable t) {
            return DOWNLOAD_FAILED;
        }
    }

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

    @Override
    protected void onPostExecute(String result) {
        super.onPostExecute(result);
        if (DOWNLOAD_FAILED.equals(DOWNLOAD_FAILED)) {
            hideLoading();
            toast( msg: "下载失败");
        } else {
            hideLoading();
            toast( msg: "下载成功\n文件已保存在: " + result);
        }
    }
}

```

```

public final int MSG_DOWN_FAIL = 1;
public final int MSG_DOWN_SUCCESS = 2;
public final int MSG_DOWN_START = 3;

```

```

private Handler mHandler = new Handler() {
    public void handleMessage(Message msg) {
        switch (msg.what) {
            case MSG_DOWN_FAIL:
                hideLoading();
                toast( msg: "下载失败");
                break;
            case MSG_DOWN_SUCCESS:
                hideLoading();
                toast( msg: "下载成功\n文件已保存在: " + msg.obj);
                break;
            case MSG_DOWN_START:
                toast( msg: "开始下载");
                showLoading();
                break;
        }
    }
};

```

```

private void initView() {
    mDownloadButton.setOnClickListener(new View.OnClickListener() {
        @Override
        public void onClick(View v) {
            new DownloadVideoThread(mVideoId).start();
        }
    });
}

```

```

public class DownloadVideoThread extends Thread {

    private String mVideoId;

    public DownloadVideoThread(String videoId) {...}

    @Override
    public void run() {
        //发送消息给 mHandler
        mHandler.sendMessage(MSG_DOWN_START);
        try {
            String localPath = downloadVideo(mVideoId);
            mHandler.sendMessage(Message.obtain(mHandler, MSG_DOWN_SUCCESS, localPath));
        } catch (Throwable t) {
            mHandler.sendMessage(Message.obtain(mHandler, MSG_DOWN_FAIL));
        }
    }

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

```


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

```
private class DownloadAsyncTask extends AsyncTask<String, Integer, String> {  
  
    final static String DOWNLOAD_FAILED = "DOWNLOAD_FAILED";  
  
    @Override  
    protected void onPreExecute() {  
        super.onPreExecute();  
        toast( msg: "开始下载");  
        showLoading();  
    }  
  
    @Override  
    protected String doInBackground(String... args) {  
        String videoId = args[0];  
        try {  
            return downloadVideo(videoId);  
        } catch (Throwable t) {  
            return DOWNLOAD_FAILED;  
        }  
    }  
  
    private String downloadVideo(String videoId) {  
        int progress = 0;  
        while(progress < 100) {  
            publishProgress( ...values: ++progress);  
        }  
        return "local_url";  
    }  
  
    @Override  
    protected void onProgressUpdate(Integer... values) {  
        super.onProgressUpdate(values);  
    }  
  
    @Override  
    protected void onPostExecute(String result) {  
        super.onPostExecute(result);  
        if (DOWNLOAD_FAILED.equals(DOWNLOAD_FAILED)) {  
            hideLoading();  
            toast( msg: "下载失败");  
        } else {  
            hideLoading();  
            toast( msg: "下载成功\n文件已保存在: " + result);  
        }  
    }  
}
```

IntentService

回顾一下Service:

Service 是一个可以在后台执行长时间运行操作而不提供用户界面的应用组件。

常见Service:

- 音乐播放
- Push



IntentService

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。

更通俗的讲：

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

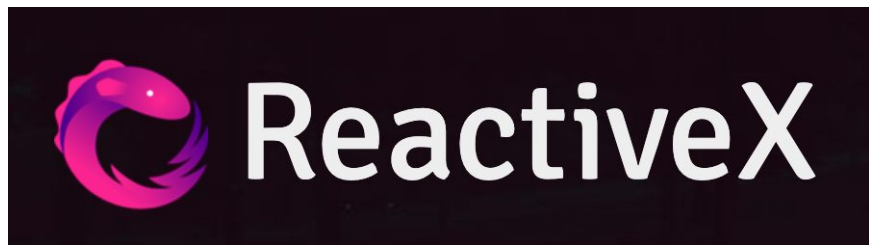
```
class DownloadIntentService extends IntentService {

    /**
     * A constructor is required, and must call the super IntentService(String)
     * constructor with a name for the worker thread.
     */
    public DownloadIntentService() {
        super( name: "DownloadIntentService");
    }

    /**
     * The IntentService calls this method from the default worker thread with
     * the intent that started the service. When this method returns, IntentService
     * stops the service, as appropriate.
     */
    @Override
    protected void onHandleIntent(Intent intent) {
        try {
            String url = intent.getStringExtra( name: "URL");
            // Download file from url
        } catch (Throwable t) {
            t.printStackTrace();
        }
    }
}
```

扩展: RxJava - 简单介绍

1. Rx = ReactiveX = Reactive Extensions = Observables + LINQ + Schedulers
2. Rx是一个使用可观察数据流进行异步编程的编程接口, ReactiveX结合了观察者模式、迭代器模式和函数式编程的精华
3. ReactiveX不仅仅是一个编程接口, 它是一种编程思想的突破, 它影响了许多其它的程序库和框架以及编程语言。



原始实现:

```
private void originImpl(List<File> folders) {  
    new Thread() {  
        @Override  
        public void run() {  
            super.run();  
            for (File folder : folders) {  
                File[] files = folder.listFiles();  
                for (File file : files) {  
                    if (file.getName().endsWith(".png")) {  
                        final Bitmap bitmap = getBitmapFromFile(file);  
                        getActivity().runOnUiThread(new Runnable() {  
                            @Override  
                            public void run() {  
                                addToBitmapPool(bitmap);  
                            }  
                        });  
                    }  
                }  
            }  
        }  
    }.start();  
}
```

RxJava2实现:

```
@SuppressWarnings("CheckResult")  
private void rxjava2Impl(List<File> folders) {  
    Flowable.fromIterable(folders) Flowable<File>  
        .flatMap(file -> Flowable.fromArray(file.listFiles())) Flowable<File>  
        .filter(file -> file.getName().endsWith(".png")) Flowable<File>  
        .map(this::getBitmapFromFile) Flowable<Bitmap>  
        .subscribeOn(Schedulers.io()) Flowable<Bitmap>  
        .observeOn(AndroidSchedulers.mainThread()) Flowable<Bitmap>  
        .subscribe(this::addToBitmapPool);  
}
```

扩展: RxJava - Schedulers

调度器类型	效果
<code>Schedulers.computation()</code>	用于计算任务, 如事件循环或和回调处理, 不要用于IO操作(IO操作请使用 <code>Schedulers.io()</code>); 默认线程数等于处理器的数量
<code>Schedulers.io()</code>	用于IO密集型任务, 如异步阻塞IO操作, 这个调度器的线程池会根据需要增长; 对于普通的计算任务, 请使用 <code>Schedulers.computation()</code> ; <code>Schedulers.io()</code> 默认是一个 <code>CachedThreadScheduler</code> , 很像一个有线程缓存的新线程调度器
<code>Schedulers.newThread()</code>	为每个任务创建一个新线程
<code>Schedulers.immediate()</code>	在当前线程立即开始执行任务
<code>Schedulers.trampoline()</code>	当其它排队的任务完成后, 在当前线程排队开始执行
<code>AndroidSchedulers.mainThread()</code>	Android中主线程; 来自于RxAndroid

Android多线程总结

01 | Thread(线程)

多线程的基础

02 | ThreadPool(线程池)

对线程进行更好的管理

03 | AsyncTask

Android中为了简化多线程的使用,
而设计的默认封装

04 | IntentService

Android中无界面异步操作的默认实现

05 | RxJava-Schedulers

当下流行的开发框架下的线程调度方式

自定义View



View绘制的三个重要步骤

Measure: 测量宽高

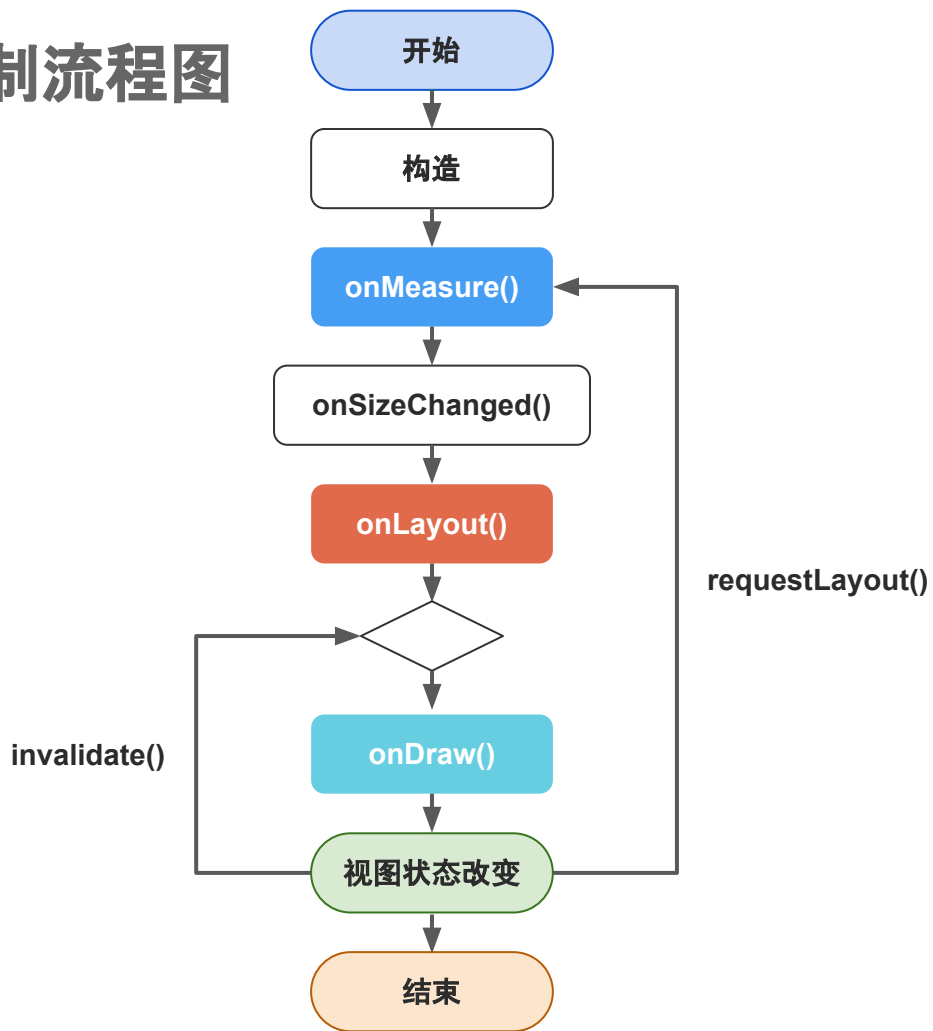
Layout: 测量宽高

Draw: 绘制形状

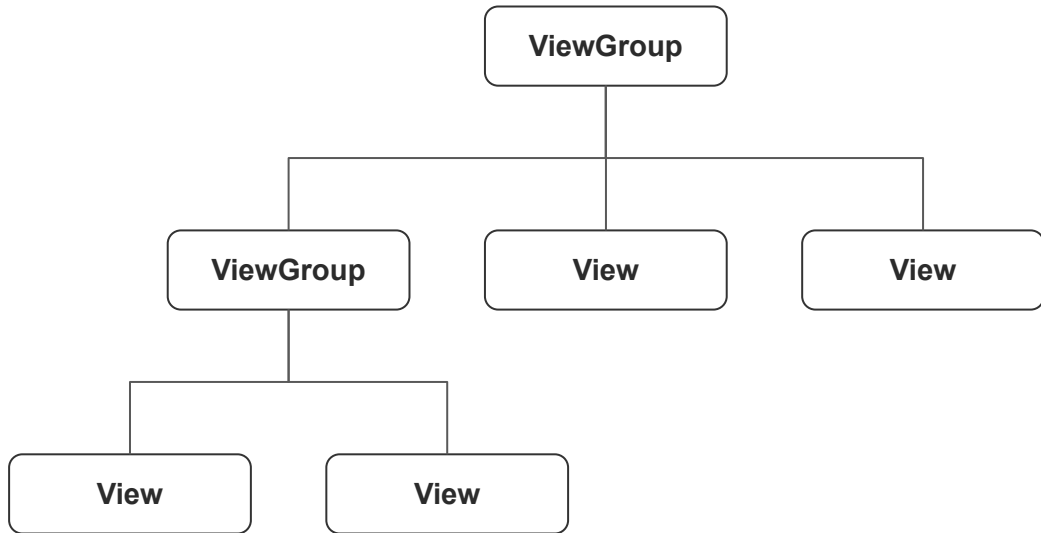
举例说明:

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)

完整的View绘制流程图

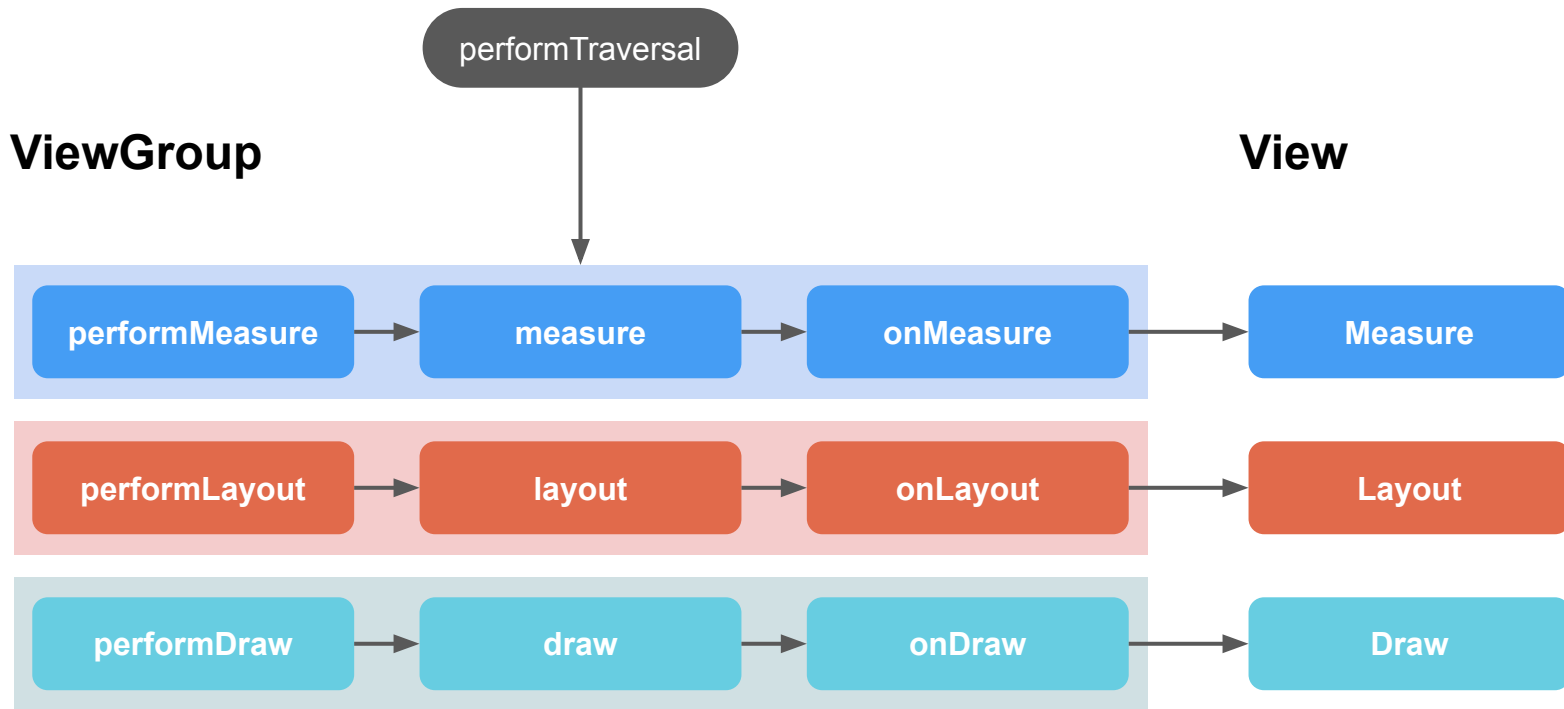


扩展:详解 ViewTree 及 View / ViewGroup 绘制流程

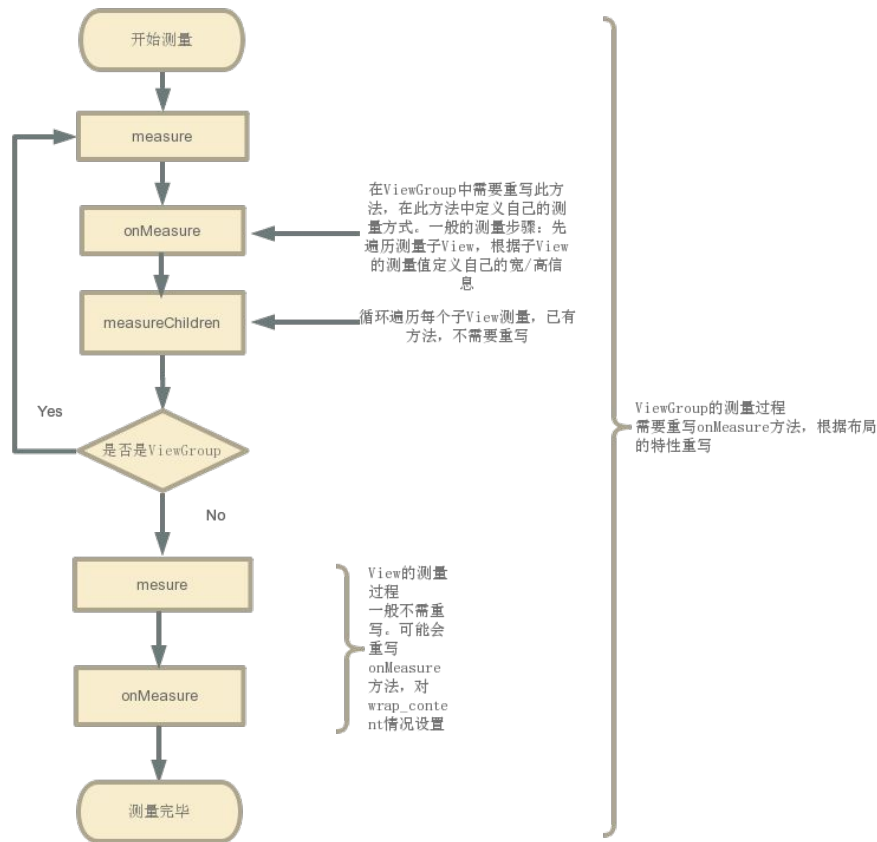


自上而下递归进行:
Measure
Layout
Draw

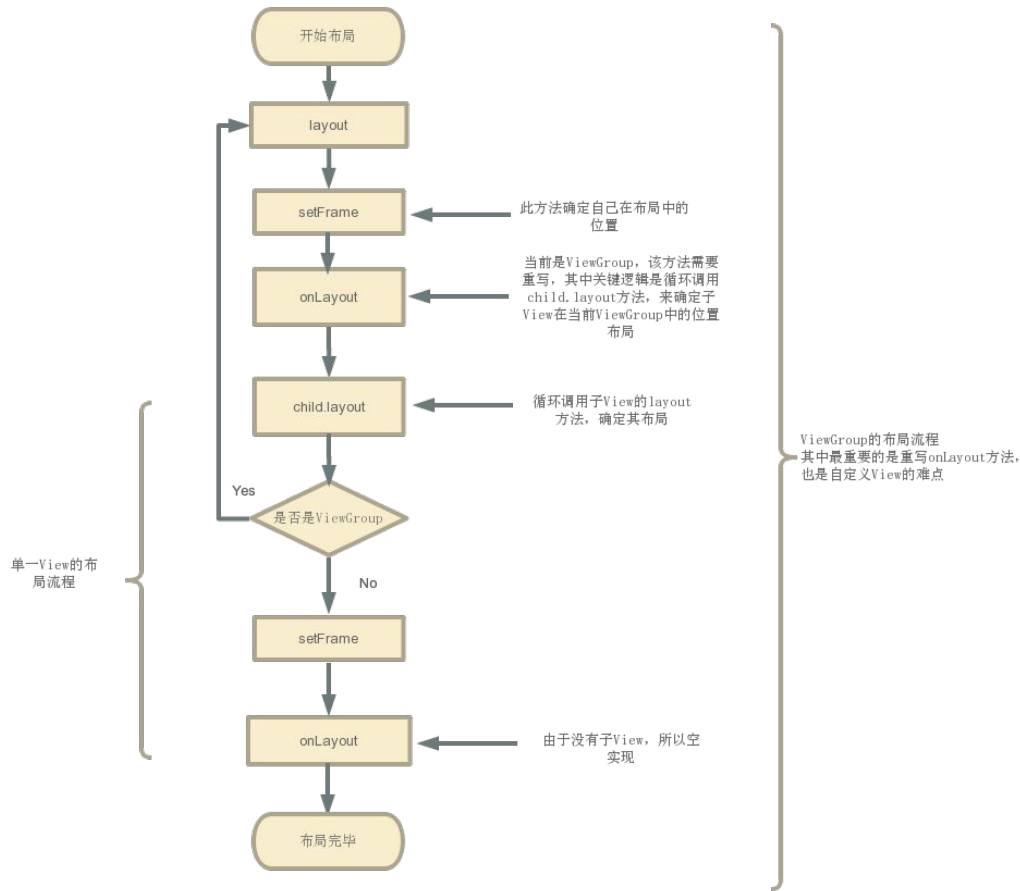
扩展: ViewGroup的绘制流程



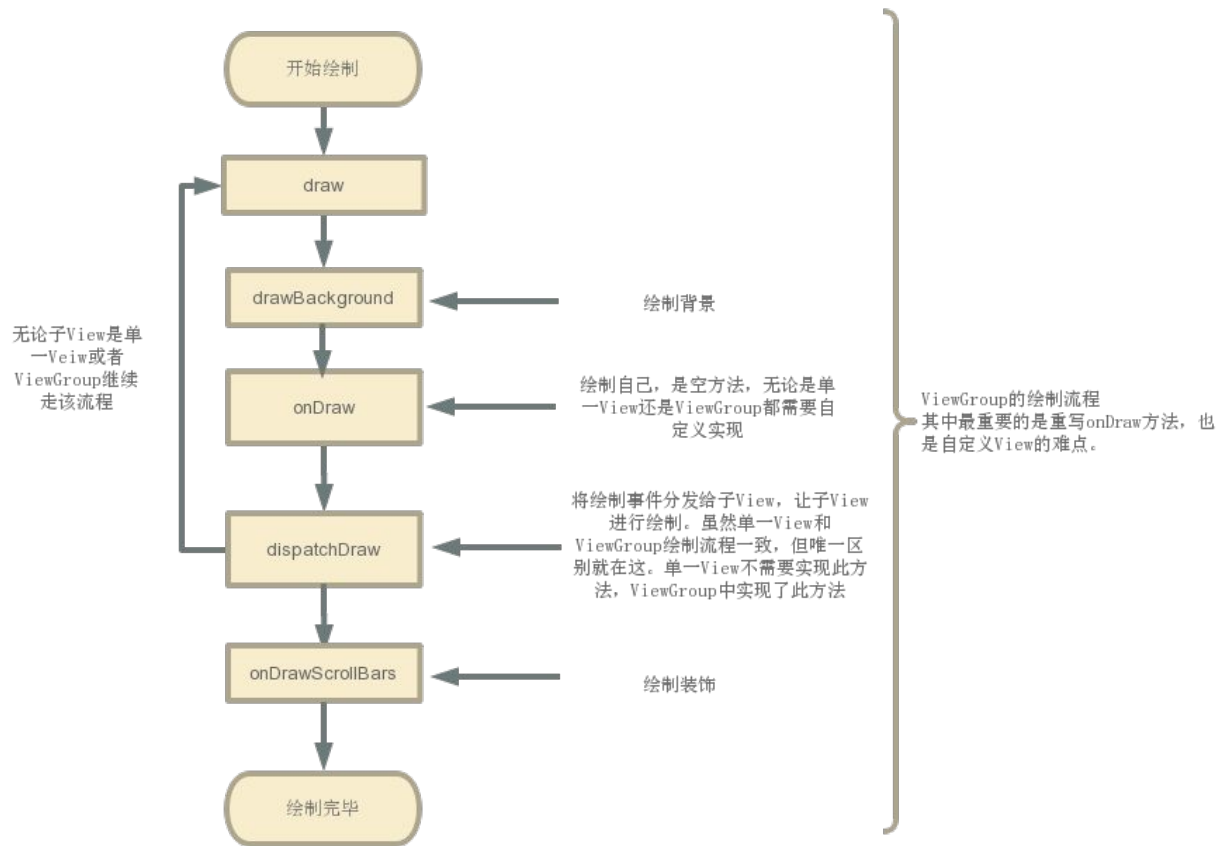
扩展: 详解ViewTree及View/ViewGroup绘制流程



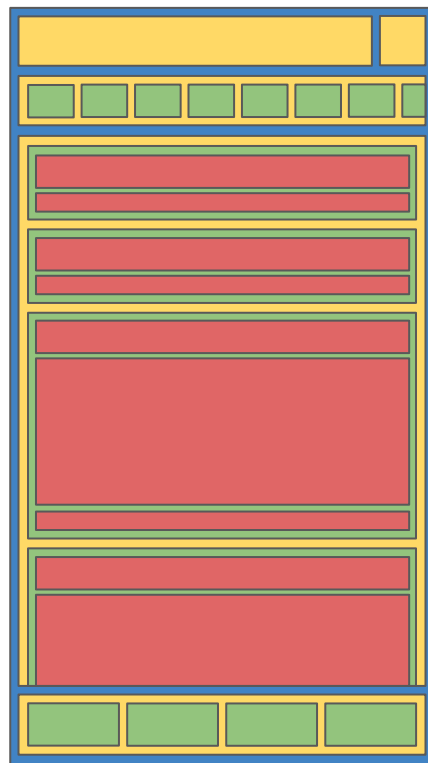
扩展: 详解ViewTree及View/ViewGroup绘制流程



扩展: 详解ViewTree及View/ViewGroup绘制流程



扩展: 详解ViewTree及View/ViewGroup绘制流程



- 第一层
- 第二层
- 第三层
- 第四层



自定义View-重写onDraw

自定义View最常见操作 - 重写onDraw

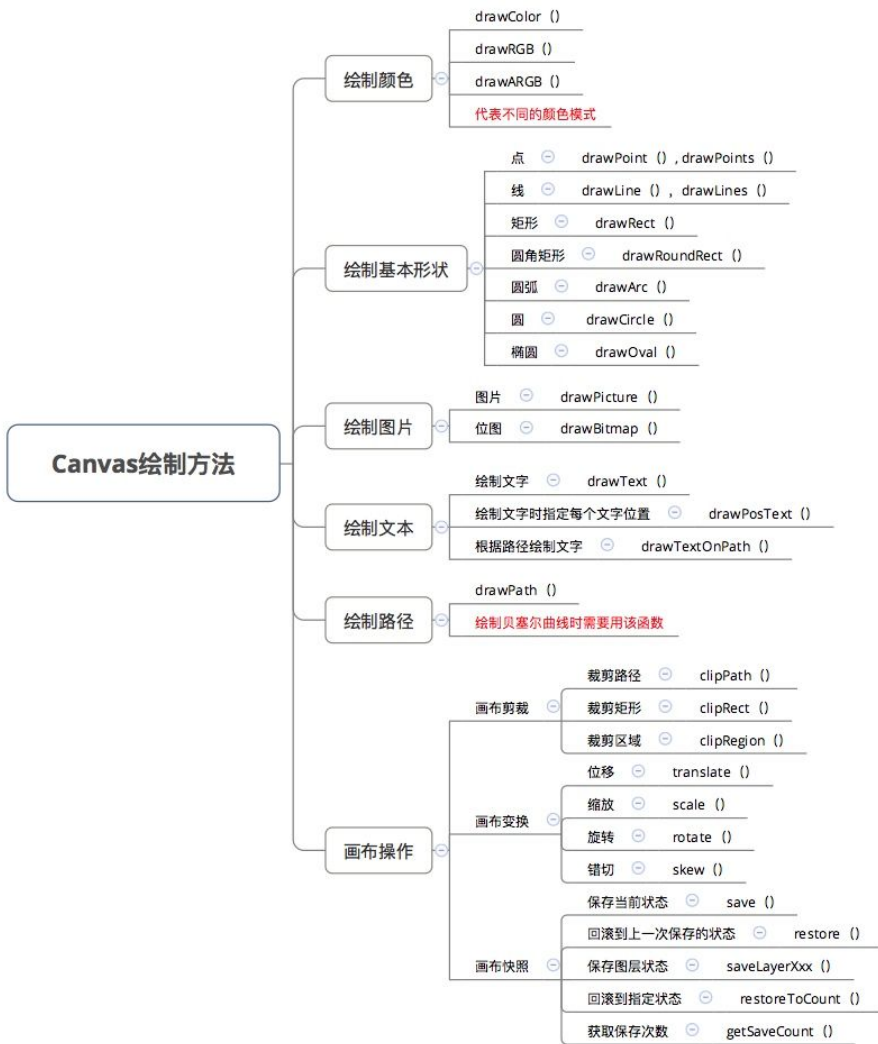
```
1 @Override
2 protected void onDraw(Canvas canvas) {
3     super.onDraw(canvas);
4     // 绘制代码
5 }
```

自定义View-重写onDraw

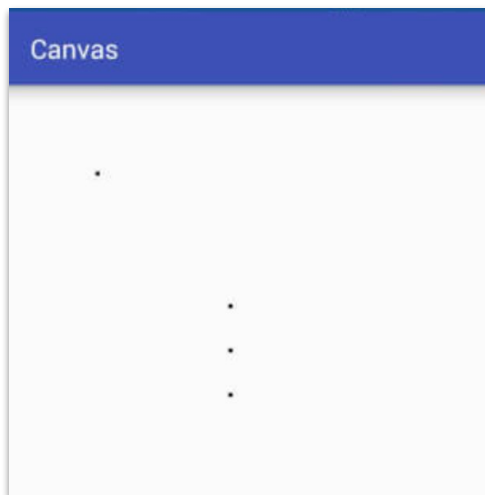
概念解析:

1. Canvas: 画布

2. Paint: 画笔



基本绘制-点 (Point)



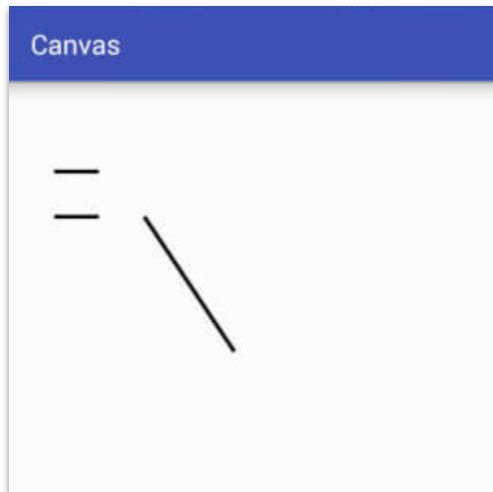
```
private Paint pointPaint;

private void initPaint() {
    pointPaint = new Paint();
    pointPaint.setColor(Color.BLACK);           //设置画笔颜色
    pointPaint.setStyle(Paint.Style.FILL);       //设置画笔模式为填充
    pointPaint.setStrokeWidth(10f);             //设置画笔宽度为10px
}

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

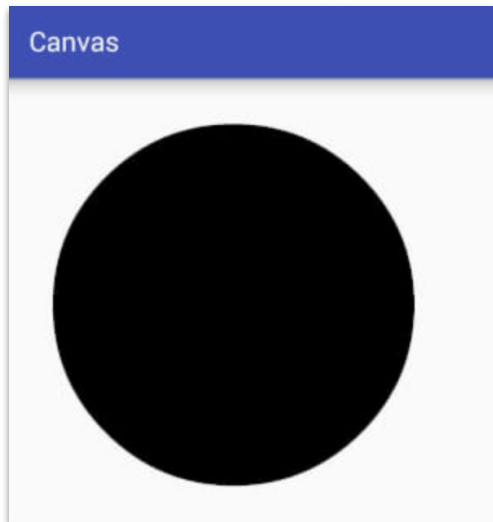
    canvas.drawPoint(x: 200, y: 200, pointPaint); //在坐标(200,200)位置绘制一个点
    canvas.drawPoints(new float[]{               //绘制一组点，坐标位置由float数组指定
        500, 500,
        500, 600,
        500, 700
    }, pointPaint);
}
```

基本绘制-线 (Line)



```
private void initPaint() {  
    linePaint = new Paint();  
    linePaint.setColor(Color.BLACK);           //设置画笔颜色  
    linePaint.setStyle(Paint.Style.FILL);      //设置画笔模式为填充  
    linePaint.setStrokeWidth(10f);            //设置画笔宽度为10px  
}  
  
@Override  
protected void onDraw(Canvas canvas) {  
    super.onDraw(canvas);  
  
    // 在坐标(300,300)(500,600)之间绘制一条直线  
    canvas.drawLine( startX: 300, startY: 300, stopX: 500, stopY: 600, linePaint);  
    // 绘制一组线 每四数字(两个点的坐标)确定一条线  
    canvas.drawLines(new float[]{  
        100, 200, 200, 200,  
        100, 300, 200, 300  
    }, linePaint);  
}
```

基本绘制-圆形 (Circle)



```
private Paint circlePaint;

private void initPaint() {
    circlePaint = new Paint();
    circlePaint.setColor(Color.BLACK);           //设置画笔颜色
    circlePaint.setStyle(Paint.Style.FILL);      //设置画笔模式为填充
}

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

    // 绘制一个圆心坐标在(500,500), 半径为400 的圆
    canvas.drawCircle(cx: 500, cy: 500, radius: 400, circlePaint);
}
```

基本绘制-矩形/圆角矩形/椭圆 (Rect / RoundRect / Oval)

```
private Paint paint;

private void initPaint() {
    paint = new Paint();
    paint.setColor(Color.BLACK);    //设置画笔颜色
    paint.setStyle(Paint.Style.FILL); //设置画笔模式为填充
}

@TargetApi(Build.VERSION_CODES.LOLLIPOP)
@Override
protected void onDraw(Canvas canvas) {
    super.onDraw(canvas);

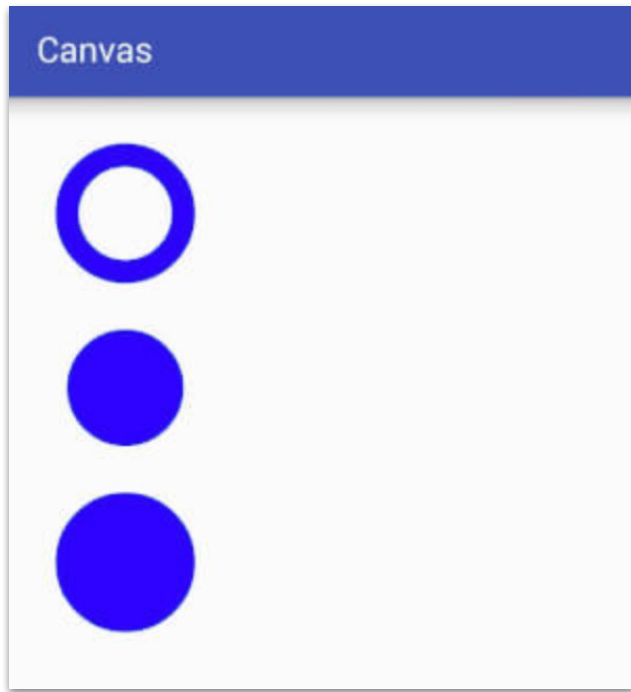
    // 绘制矩形
    canvas.drawRect( left: 100, top: 100, right: 800, bottom: 400, paint);

    // 绘制圆角矩形
    canvas.drawRoundRect( left: 100, top: 100, right: 800, bottom: 400, rx: 30, ry: 30, paint);

    // 绘制椭圆
    canvas.drawOval( left: 100, top: 100, right: 800, bottom: 400, paint);
}
```

基本绘制-填充

(代码举例)



```
private Paint paint;

private void initPaint() {
    paint = new Paint();
    paint.setColor(Color.BLUE);
    paint.setStrokeWidth(40);
}

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

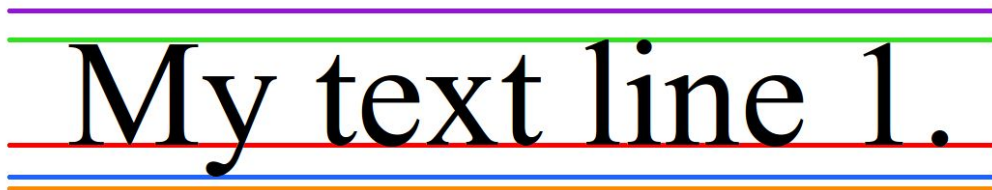
    // 描边
    paint.setStyle(Paint.Style.STROKE);
    canvas.drawCircle( cx: 200, cy: 200, radius: 100, paint);

    // 填充
    paint.setStyle(Paint.Style.FILL);
    canvas.drawCircle( cx: 200, cy: 500, radius: 100, paint);

    // 描边加填充
    paint.setStyle(Paint.Style.FILL_AND_STROKE);
    canvas.drawCircle( cx: 200, cy: 800, radius: 100, paint);
}
```


基本绘制-文字

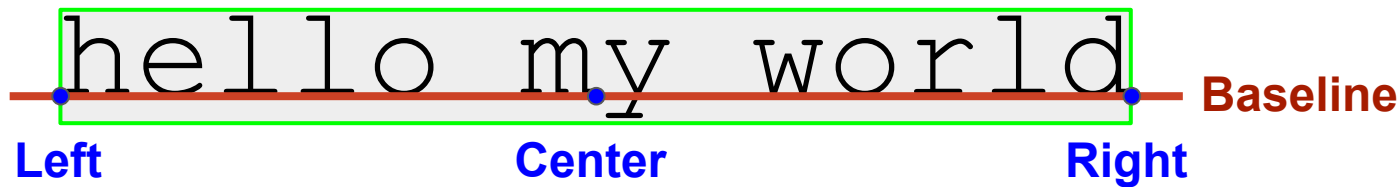
- Top
- Ascent
- Baseline
- Descent
- Bottom



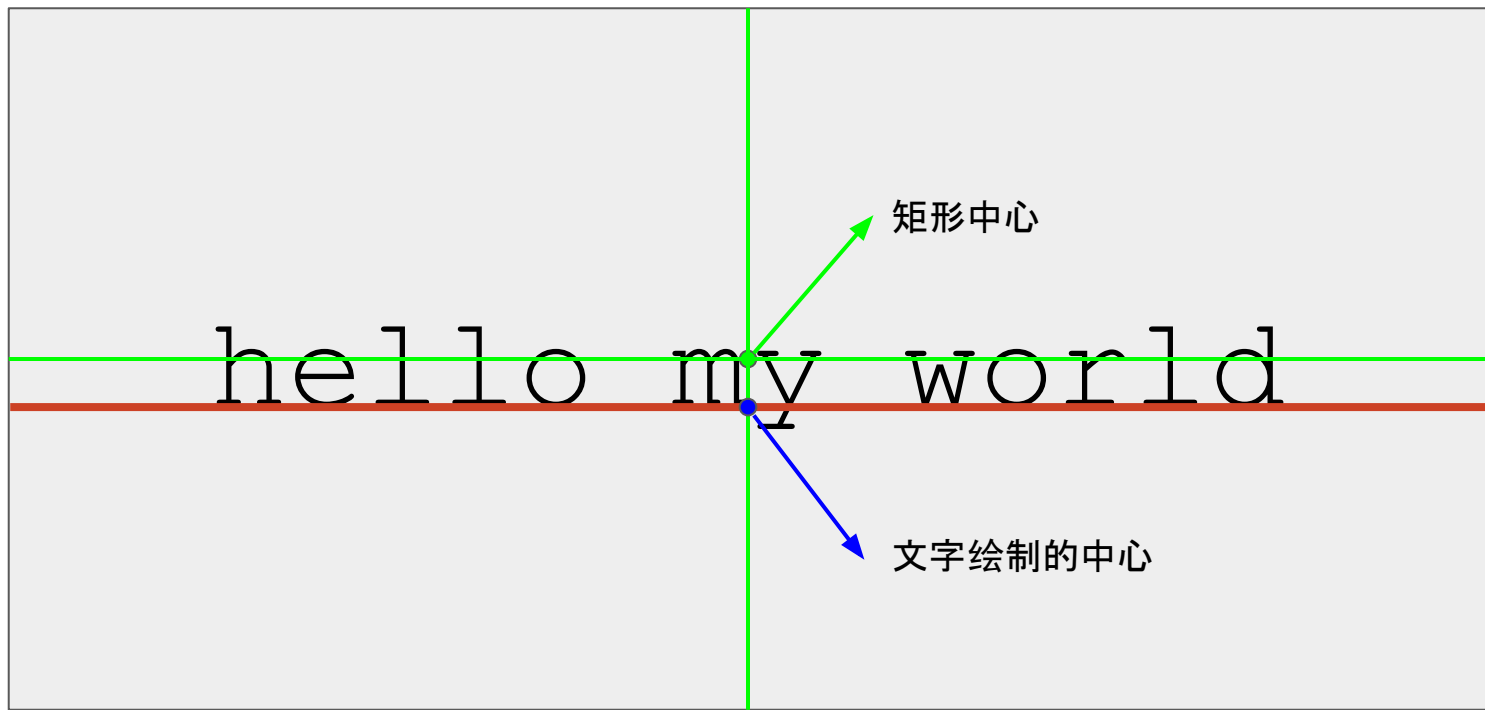
My text line 1.

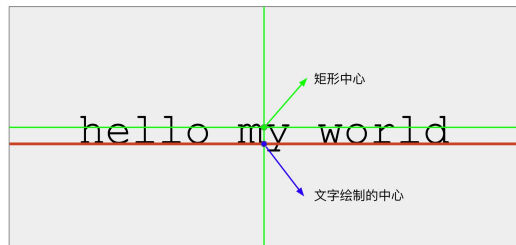
The diagram illustrates the vertical alignment of the text "My text line 1." using five horizontal lines: a purple line at the top of the capital 'M', a green line at the top of the lowercase 'y', a red line at the baseline of the text, a blue line at the bottom of the lowercase 'y', and an orange line at the bottom of the text. The text is positioned between the green and red lines.

基本绘制-文字



基本绘制-文字





```
private void drawTextCenter(Canvas canvas, float centerX, float centerY, int color) {  
  
    Paint textPaint = new Paint();  
    textPaint.setColor(Color.WHITE);  
    textPaint.setTextSize(50);  
    textPaint.setStyle(Paint.Style.FILL);  
    //该方法即为设置基线上那个点究竟是left,center,还是right 这里我设置为center  
    textPaint.setTextAlign(Paint.Align.CENTER);  
  
    Paint.FontMetrics fontMetrics = textPaint.getFontMetrics();  
    float top = fontMetrics.top;//为基线到字体上边框的距离,即上图中的top  
    float bottom = fontMetrics.bottom;//为基线到字体下边框的距离,即上图中的bottom  
  
    int baseLineY = (int) (rect.centerY() - top / 2 - bottom / 2);//基线中间点的y轴计算公式  
  
    canvas.drawText("hello my world", rect.centerX(), baseLineY, textPaint);  
}
```



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
3. 有内存泄露(什么是内存泄露?)



10:46:18_{AM}

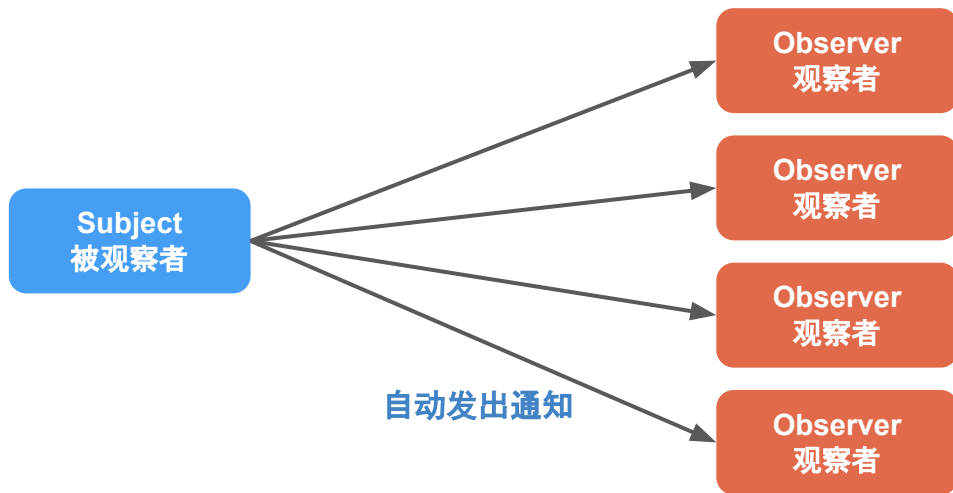


附：内存泄露 - Memory Leak

- Java的内存回收机制
- 一般内存泄漏(traditional memory leak)的原因是：由忘记释放分配的内存导致的。
 - 举例：Cursor的泄露
- 逻辑内存泄漏(logical memory leak)的原因是：当应用不再需要这个对象，当仍未释放该对象的所有引用。
 - 举例：Activity的泄露
 - 最常见原因：内部类引用外部变量

附：观察者模式

- 生动的例子：
 - 西游记里面悟空请求菩萨降服红孩儿，菩萨洒了一地水招来一个老乌龟：
 - 这个乌龟就是观察者，菩萨通过洒水通知他。



附: 观察者模式

优点:

- 解耦, 被观察者只知道观察者列表「抽象接口」, 被观察者不知道具体的观察者
- 被观察者发送通知, 所有注册的观察者都会收到信息「可以实现广播机制」

Android中的例子:

- `View.setOnClickListener(...);`



THANKS



ByteDance 字节跳动