

# 复杂应用组件

Handler机制、多线程与自定义View

全亮 [liang.tong@bytedance.com](mailto:liang.tong@bytedance.com)

字节跳动Android工程师



## ●●提纲

**01** | 进程与线程

**02** | Handler机制

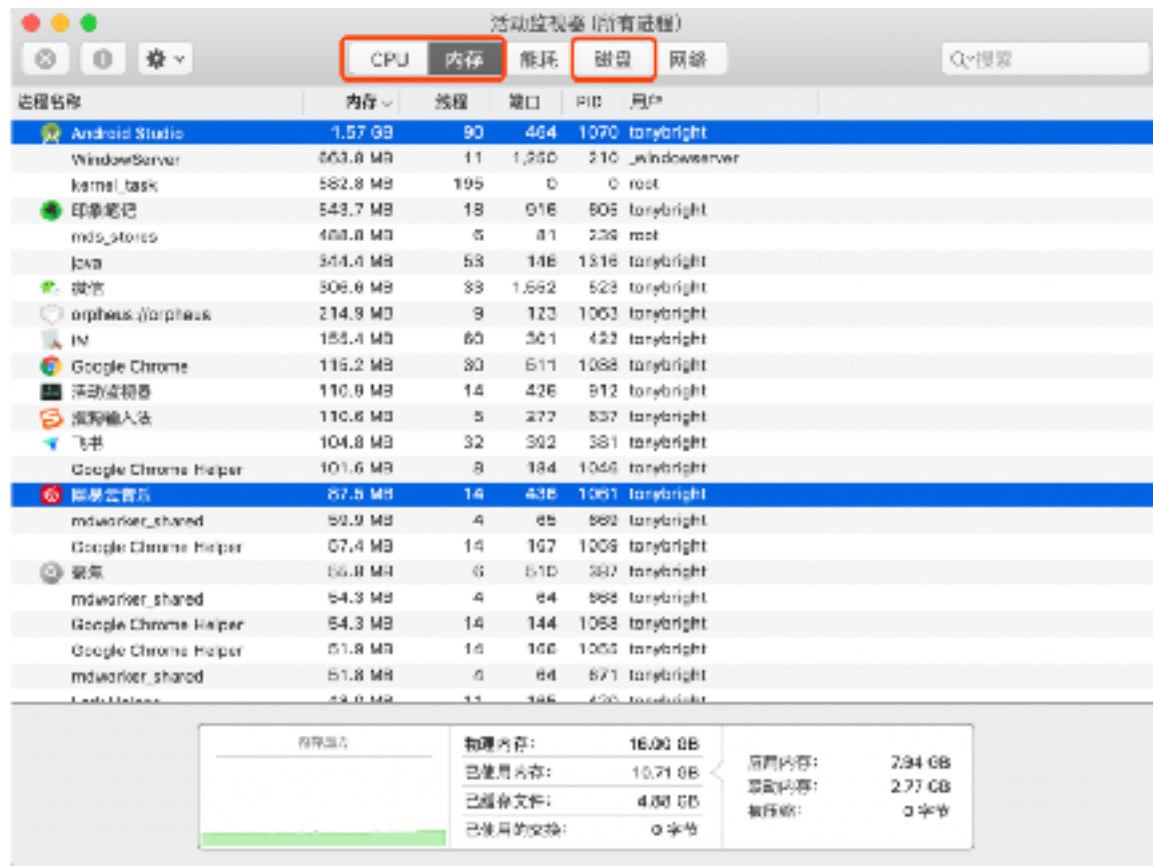
**03** | Android中的多线程

**04** | 自定义View

# 进程与线程



# 进程



# 进程

## 抖音app创建的进程

```
tonybright@tonybright ~ $ adb shell "ps|grep com.ss.android.ugc.aweme"
u0_a613      22342    621 2228160 246680 0      0 S com.ss.android.ugc.aweme
u0_a613      22590    621 1842112 70468 0      0 S com.ss.android.ugc.aweme:bm
u0_a613      22660    621 1816440 64644 0      0 S com.ss.android.ugc.aweme:push
u0_a613      22772    621 1834996 71856 0      0 S com.ss.android.ugc.aweme:pushservice
```

# 线程

## 抖音主进程的部分线程

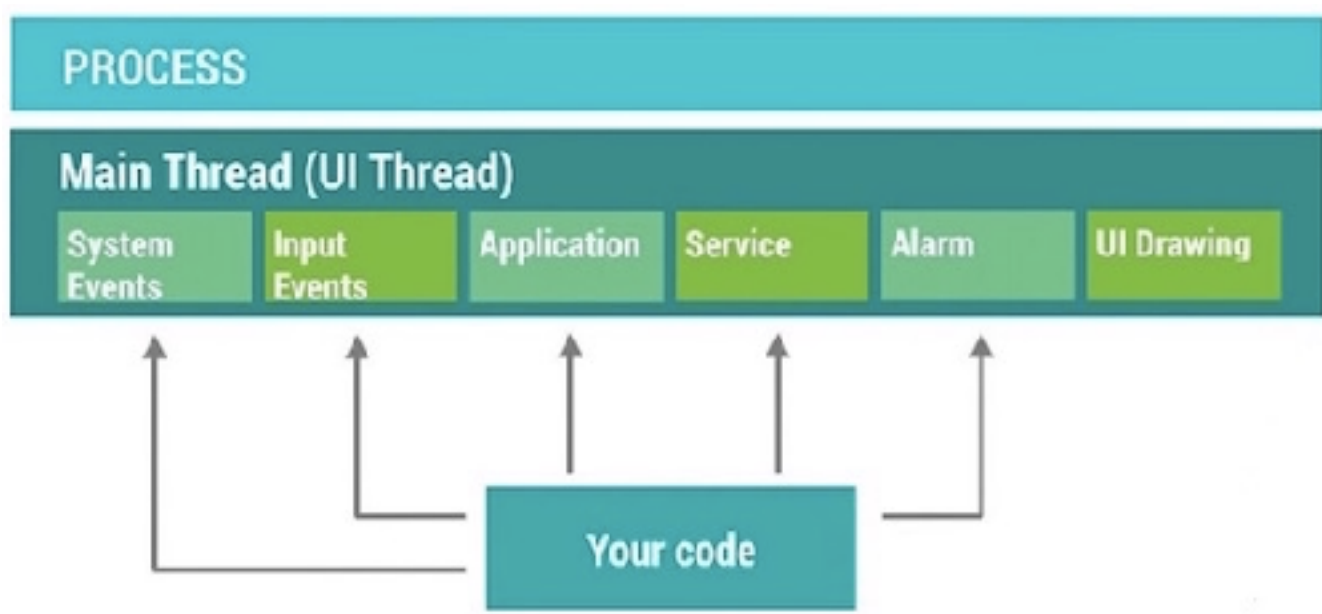
```
tonybright@tonybright ~ $ adb shell ps -T | grep "22342"
u0_a613      22342 22342   621 2171240 218380 0      0 S droid.ugc.awene
u0_a613      22342 22349   621 2171240 218380 0      0 S Jit thread pool
u0_a613      22342 22350   621 2171240 218380 0      0 S Signal Catcher
u0_a613      22342 22351   621 2171240 218380 0      0 S ReferenceQueueD
u0_a613      22342 22352   621 2171240 218380 0      0 S FinalizerDaemon
u0_a613      22342 22354   621 2171240 218380 0      0 S HeapTaskDaemon
u0_a613      22342 22355   621 2171240 218380 0      0 S Binder:22342_1
u0_a613      22342 22356   621 2171240 218380 0      0 S Binder:22342_2
u0_a613      22342 22361   621 2171240 218380 0      0 S Profile Saver
u0_a613      22342 22369   621 2171240 218380 0      0 S awene-thread-po
u0_a613      22342 22370   621 2171240 218380 0      0 S awene-thread-po
u0_a613      22342 22371   621 2171240 218380 0      0 S awene-thread-po
u0_a613      22342 22372   621 2171240 218380 0      0 S awene-thread-po
u0_a613      22342 22374   621 2171240 218380 0      0 S Legollander
u0_a613      22342 22409   621 2171240 218380 0      0 S Binder:22342_3
u0_a613      22342 22429   621 2171240 218380 0      0 S TaskMonitor-10
u0_a613      22342 22439   621 2171240 218380 0      0 S queued-work-loa
u0_a613      22342 22446   621 2171240 218380 0      0 S ActionReaper
u0_a613      22342 22449   621 2171240 218380 0      0 S Legollander
u0_a613      22342 22460   621 2171240 218380 0      0 S RxSchedulerPurg
u0_a613      22342 22462   621 2171240 218380 0      0 S RxCachedMarkerP
u0_a613      22342 22468   621 2171240 218380 0      0 S CronetInit
u0_a613      22342 22469   621 2171240 218380 0      0 S ChromiumNet10
u0_a613      22342 22477   621 2171240 218380 0      0 S DeviceRegisterT
u0_a613      22342 22482   621 2171240 218380 0      0 S Queue
```



## 进程、线程总结

- 进程是资源分配的基本单位
- 线程是CPU调度的基本单位
- 进程可以有多个线程，同一进程的线程共享进程的资源

# Android主进程&UI线程





# 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系统解决了以下问题:

1. 任务调度
2. 线程通信



## 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_DOWNLOAD_FAIL = 1;
public final int MSG_DOWNLOAD_SUCCESS = 2;
public final int MSG_DOWNLOAD_START = 3;

private Handler mHandler = new Handler() {
    public void handleMessage(Message msg) {
        switch (msg.what) {
            case MSG_DOWNLOAD_FAIL:
                hideLoading();
                toast(msg, "下载失败");
                break;
            case MSG_DOWNLOAD_SUCCESS:
                hideLoading();
                toast(msg, "下载成功\n文件已保存: " + msg.obj);
                break;
            case MSG_DOWNLOAD_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.sendEmptyMessage(MSG_DOWNLOAD_START);
        try {
            String localPath = downloadVideo(mVideoId);
            mHandler.sendMessage(Message.obtain(mHandler, MSG_DOWNLOAD_SUCCESS, localPath));
        } catch (Throwable t) {
            mHandler.sendMessage(Message.obtain(mHandler, MSG_DOWNLOAD_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);
```

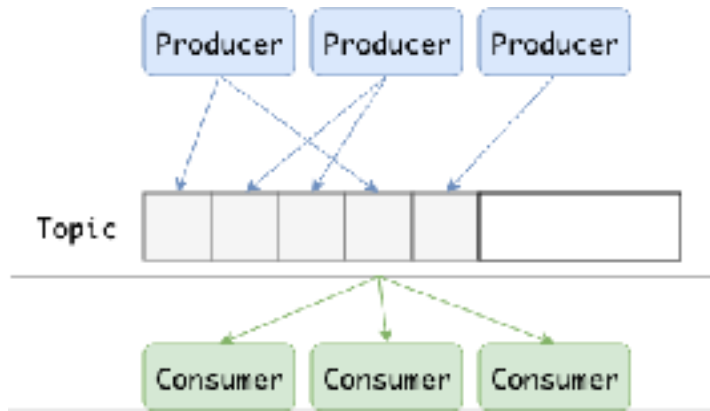


# Handler原理：消息队列机制

- 消息队列机制（英语：Message Queue）常用于进程间、线程间的通信

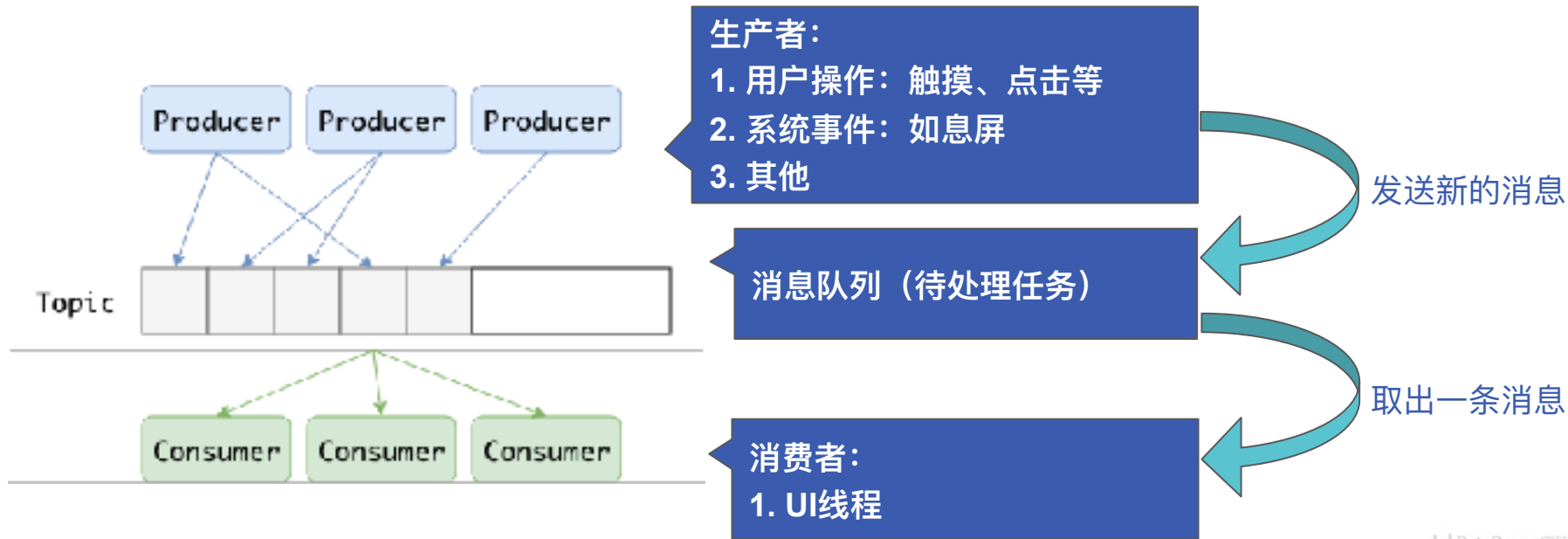
## 消息队列实际应用：

- Kafka分布式消息处理系统
- Java线程池模型
- Android UI线程消息处理模型



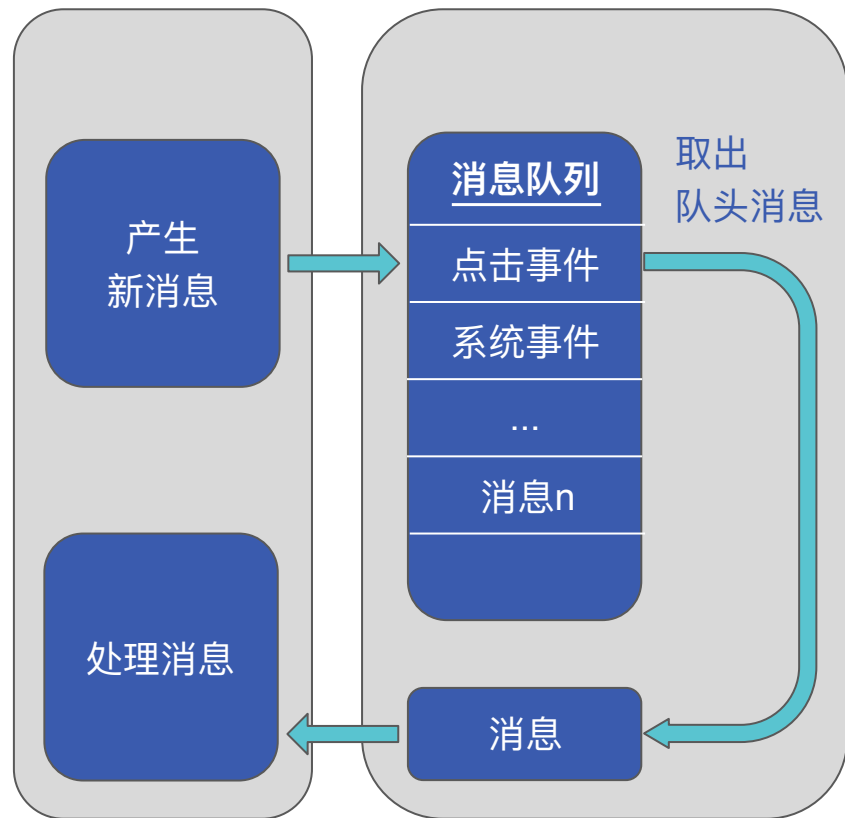
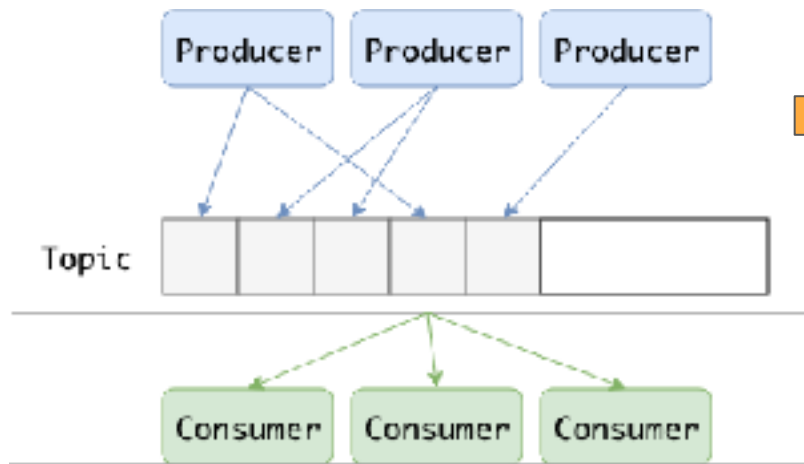
# Handler原理：UI线程与消息队列机制

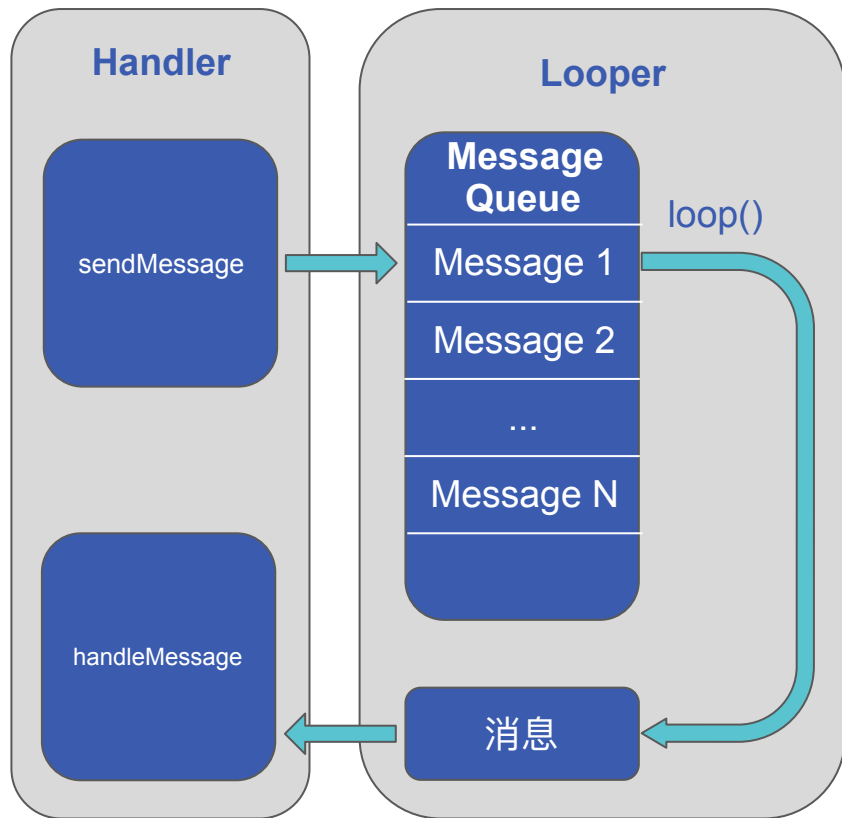
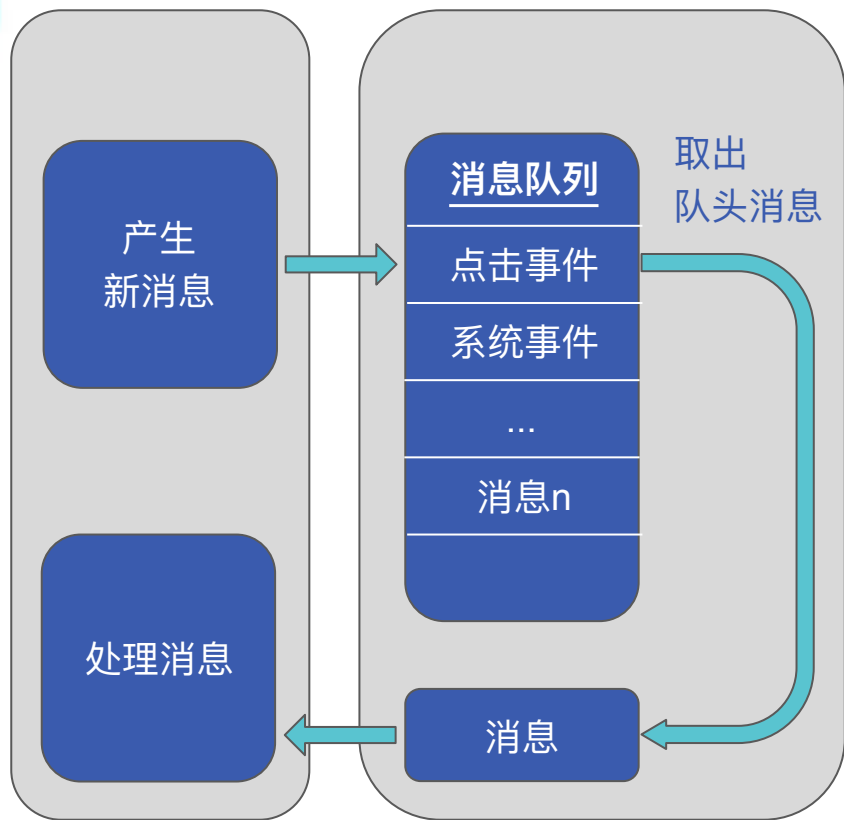
- Android中UI线程负责处理界面的展示，响应用户的操作：



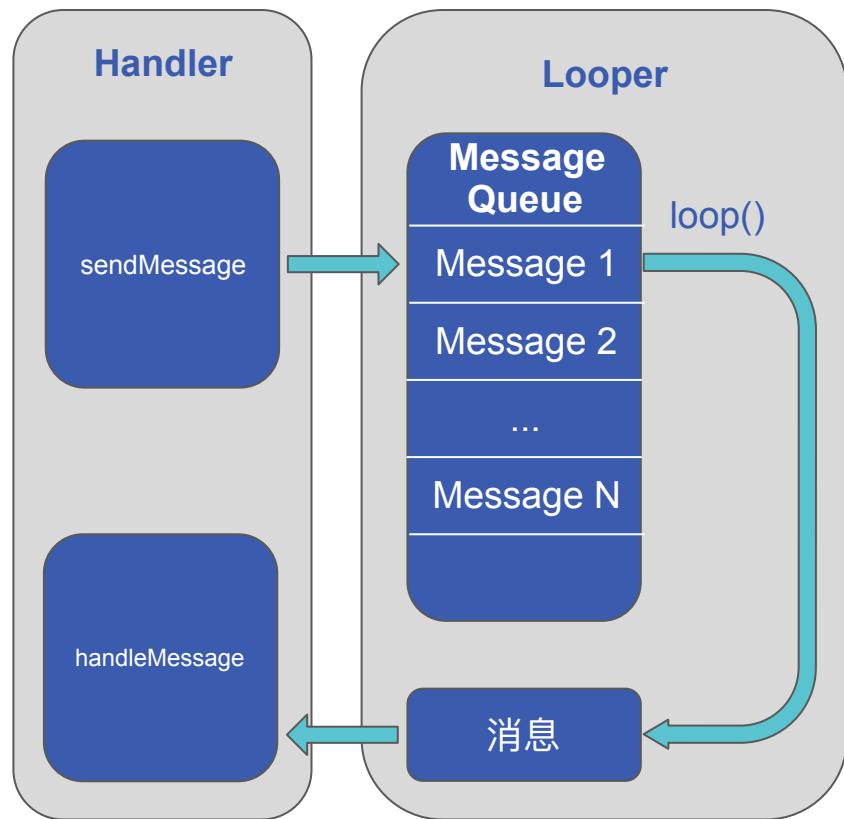
# Handler原理：UI线程与消息队列机制

- 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

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

=

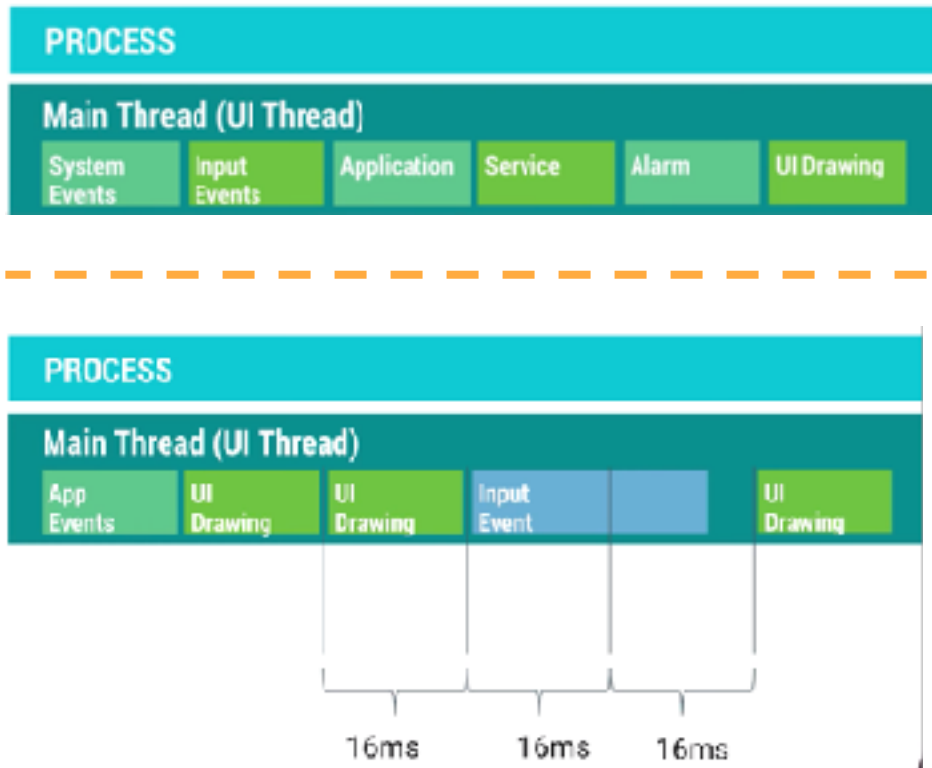
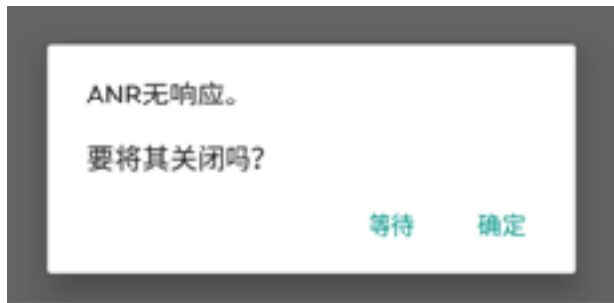
```
protected final Handler mHandler = new Handler() {  
    @Override  
    public void handleMessage(Message msg) {  
        super.handleMessage(msg);  
        switch (msg.what) {  
            case MSG_GO_MAIN_ACTIVITY:  
                goMainActivity();  
                break;  
        }  
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

```
private static Message getPostMessage(Runnable r) {  
    Message m = Message.obtain();  
    m.callback = r;  
    return m;  
}
```

```
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)



(其中每一帧内容的绘制其实只有16ms)



# Handler总结

- Handler机制是消息队列机制在Android上的应用，解决Android中的任务调度和线程通信问题
- Handler负责向消息队列里添加消息，Looper维持一个循环，从消息队列取消息，派发给Handler处理，队列为空时阻塞等待
- 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





# Android里常用的操作多线程方式

**01** | Thread (线程)

**02** | ThreadPool (线程池)

**03** | AsyncTask

**04** | IntentService



# 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，通过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 mId = -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
    // work before looper loop.
    //
    protected void onLooperPrepared() {
    }

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

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

    //...
    @Nullable
    public Handler getThreadHandler() {...}

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

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

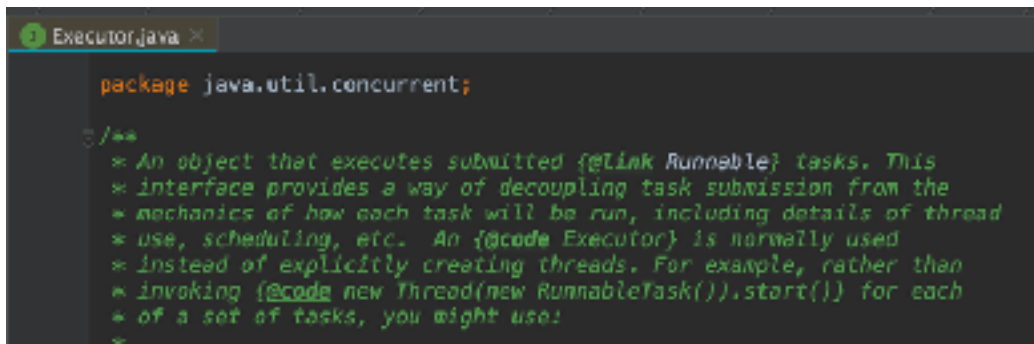
    //...
    public int getThreadId() { return mId; }
```



# ThreadPool

- 线程池的作用

- 执行提交的任务
- 解耦任务提交、执行
- 封装线程使用、调度细节



```
Executor.java X
package java.util.concurrent;

/**
 * An object that executes submitted {@link Runnable} tasks. This
 * interface provides a way of decoupling task submission from the
 * mechanics of how each task will be run, including details of thread
 * use, scheduling, etc. An {@code Executor} is normally used
 * instead of explicitly creating threads. For example, rather than
 * invoking {@code new Thread(new RunnableTask()).start()} for each
 * of a set of tasks, you might use:
 */
```

- 重要函数:

- execute(Runnable)
- submit(Runnable): 有返回值 (Future) , 可以cancel, 线程池处理异常
- shutdown()



# ThreadPool

为什么要使用线程池？

1. 频繁地执行线程创建、销毁，性能开销很大，线程池的线程复用可以有效降低性能开销
2. 基于线程池更便于做线程任务监控和性能优化





# 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）：
  - newFixedThreadPool 定长线程池
  - newCachedThreadPool 可缓存线程池
- 执行定时任务（定时线程池）：
  - newScheduledThreadPool 定时任务线程池
- 特定单项任务（单线程线程池）：
  - newSingleThreadExecutor 单线程线程池

```
ExecutorService service = Executors.newSingleThreadScheduledExecutor();
((ScheduledExecutorService) service).scheduleAtFixedRate(new Runnable() {
    @Override public void run() {
        // 请求股票数据
        // ...
        // 写入DB、回调到主线程显示行情数据
        // ..
    }
}, initialDelay: 1, period: 10, TimeUnit.SECONDS);
```



# 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。

```

private void initView() {
    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(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));
        }
    }

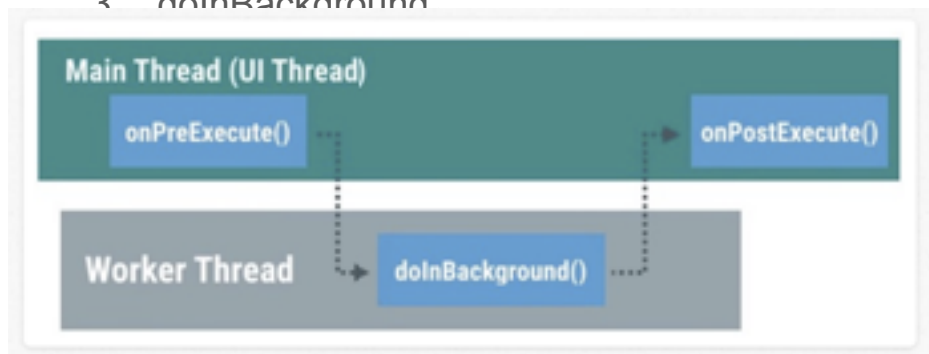
    private String downloadVideo(String videoId) {...}
}

```

# AsyncTask

AsyncTask的定义及重要函数:

1. AsyncTask<Params, Progress, Result>
2. onPreExecute:
3. doInBackground:



```
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：

- 不需要展示用户界面
- 执行时间比较长
- 后台运行

常见Service：

- 音乐播放
- Push推送



正在运行		
头条	今日头条	12 MB
	1 个进程和 2 个服务	1:55:17
头条	今日头条	12 MB
	1 个进程和 1 个服务	1:55:20
头条	今日头条	126 MB
	2 个进程和 2 个服务	1:55:15
微信	微信	27 MB
	1 个进程和 1 个服务	31:05:11
微信	微信	179 MB
	1 个进程和 1 个服务	28:04:14
QQ	QQ	80 MB
	2 个进程和 2 个服务	07:40
钉钉	钉钉	76 MB
	1 个进程和 1 个服务	24:46:12
钉钉	钉钉	11 MB
	1 个进程和 2 个服务	24:46:10
钉钉	钉钉	173 MB
	2 个进程和 1 个服务	3:24:25
钉钉	钉钉	62 MB
	2 个进程和 1 个服务	13:00



# IntentService

那什么是IntentService?

*Service*是执行在主线程的。

而很多情况下，我们做的事情非常耗时，需要在单独的线程执行，那么就应该用*IntentService*。

比如：用*Service*下载文件

# IntentService示例

```
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("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("URL");  
            // Download file from url  
        } catch (Throwable t) {  
            t.printStackTrace();  
        }  
    }  
}
```

# IntentService源码

```
IntentService.java
public abstract class IntentService extends Service {
    private volatile Looper mServiceLooper;
    private volatile ServiceHandler mServiceHandler;
    private String mName;
    private boolean mRedelivery;

    private final class ServiceHandler extends Handler {
        public ServiceHandler(Looper looper) { super(looper); }

        @Override
        public void handleMessage(Message msg) {
            onHandleIntent((Intent)msg.obj);
            stopSelf(msg.arg1);
        }
    }

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

    /**
     * @see {@link}
     */
    public void setIntentRedelivery(boolean enabled) { mRedelivery = enabled; }

    @Override
    public void onCreate() {
        // TODO: It would be nice to have an option to hold a partial wakelock
        // during processing, and to have a static startService(Context, Intent)
        // method that would launch the service & hand off a wakelock.

        super.onCreate();
        HandlerThread thread = new HandlerThread("IntentService!" + mName + "!");
        thread.start();

        mServiceLooper = thread.getLooper();
        mServiceHandler = new ServiceHandler(mServiceLooper);
    }
}
```



## 扩展：RxJava - 简单介绍



ReactiveX

Reactive Extensions for Async Programming

<https://github.com/ReactiveX>

ReactiveX / RxJava

Watch

2,380

★ Unstar

39,617

Fork

6,686

Code

Issues 10

Pull requests 1

Projects 0

Wiki

Security

Insights

RxJava – Reactive Extensions for the JVM – a library for composing asynchronous and event-based programs using observable sequences for the Java VM.

java

rxjava

flow

reactive-streams

5,543 commits

4 branches

215 releases

235 contributors

Apache-2.0

Branch: 3.x

New pull request

Create new file

Upload files

Find File

Clone or download

## Thread:

```
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() {  
                            imageCollectorView.addImage(bitmap);  
                        }  
                    });  
                }  
            }  
        }  
    }  
}.start();
```

---

## RxJava:

```
Observable.from(folders)  
    .flatMap((Func1) (folder) -> { Observable.from(file.listFiles()) })  
    .filter((Func1) (file) -> { file.getName().endsWith(".png") })  
    .map((Func1) (file) -> { getBitmapFromFile(file) })  
    .subscribeOn(Schedulers.io())  
    .observeOn(AndroidSchedulers.mainThread())  
    .subscribe((Action1) (bitmap) -> { imageCollectorView.addImage(bitmap) });
```

# 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

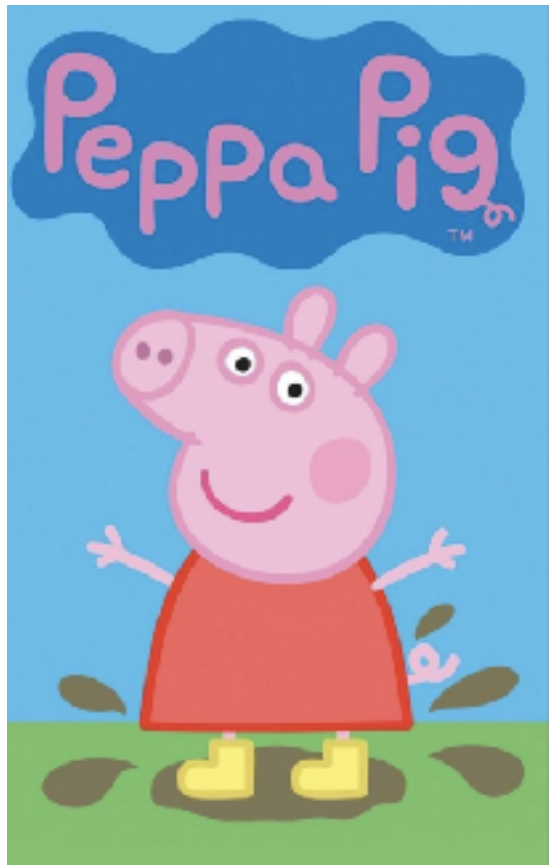
**05** | RxJava

当下流行的开发框架下的线程调度方式

# 自定义View



## 怎么画一个佩奇?

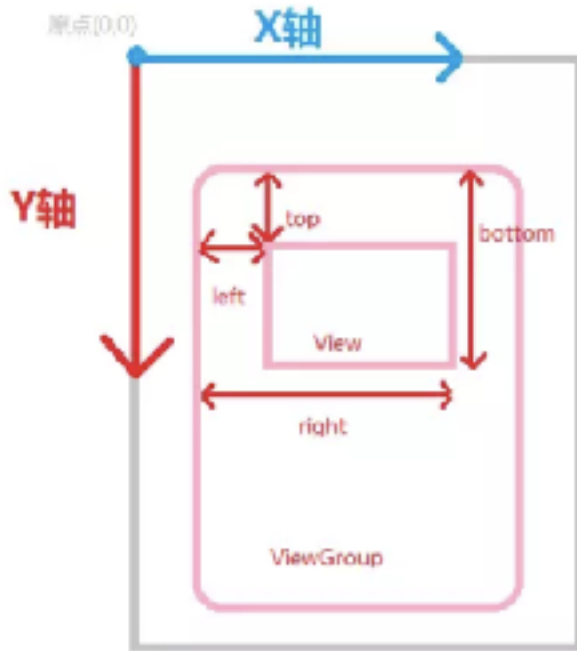
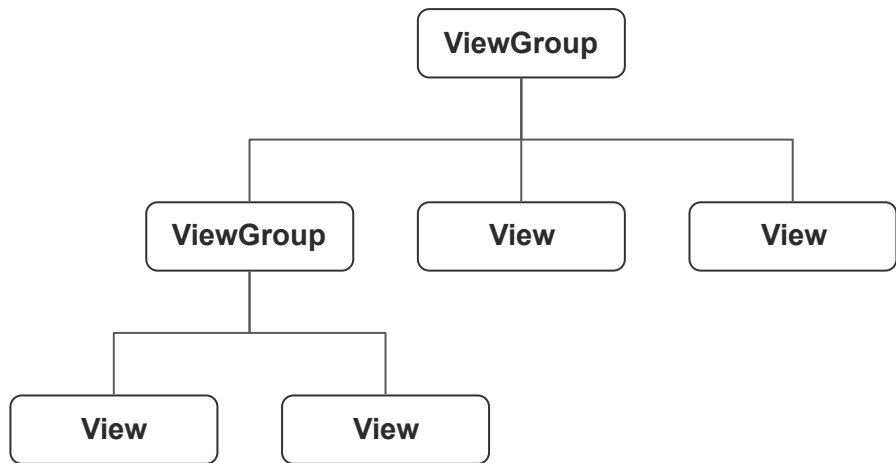


Measure: 测量宽高

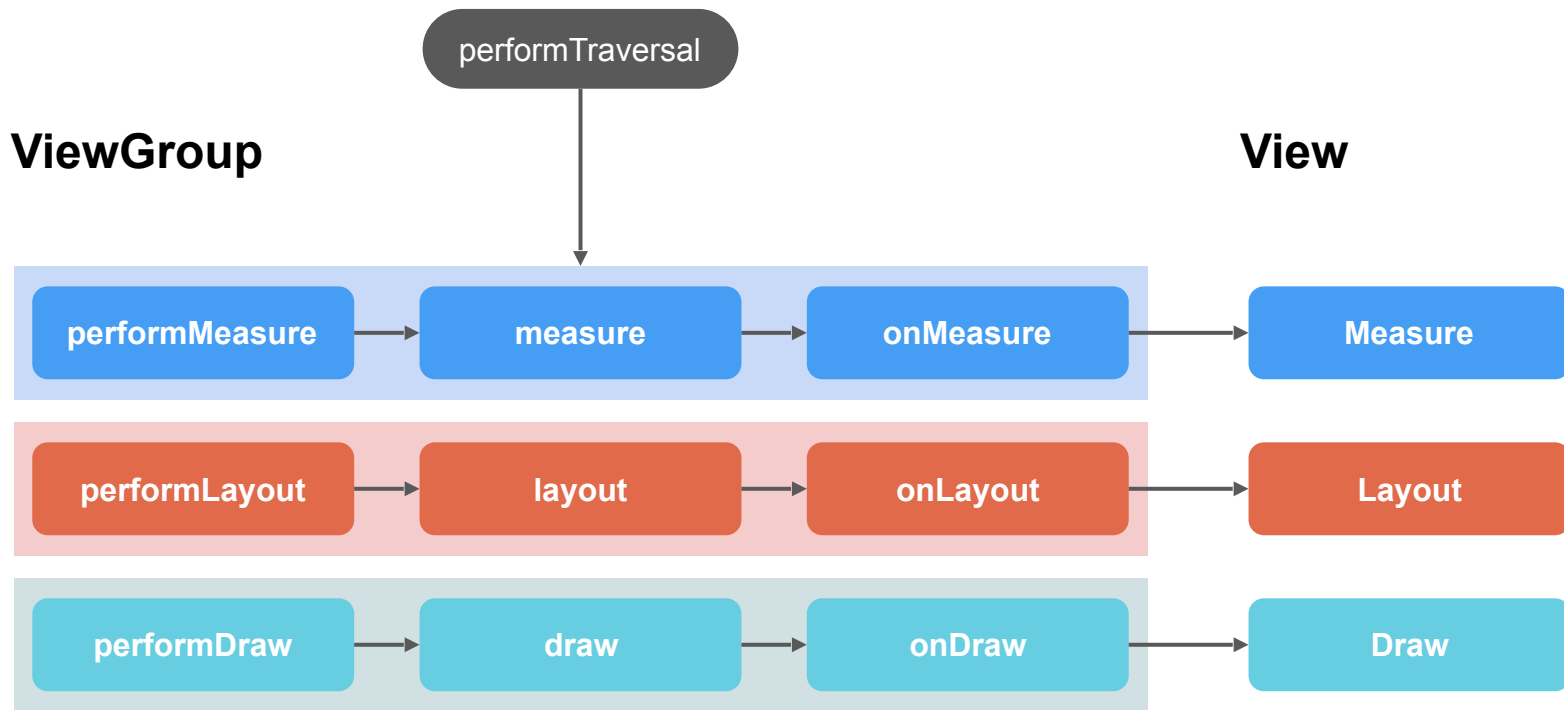
Layout: 计算布局

Draw: 绘制形状

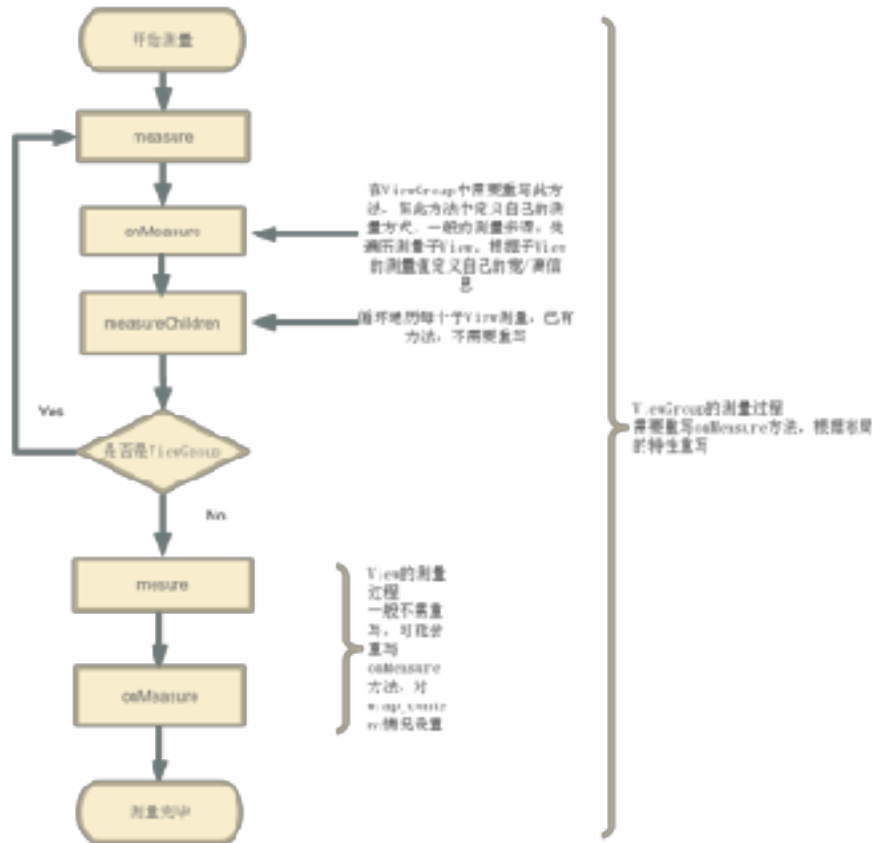
## 扩展：详解 ViewTree 及 View / ViewGroup 绘制流程



## 扩展：ViewGroup的绘制流程



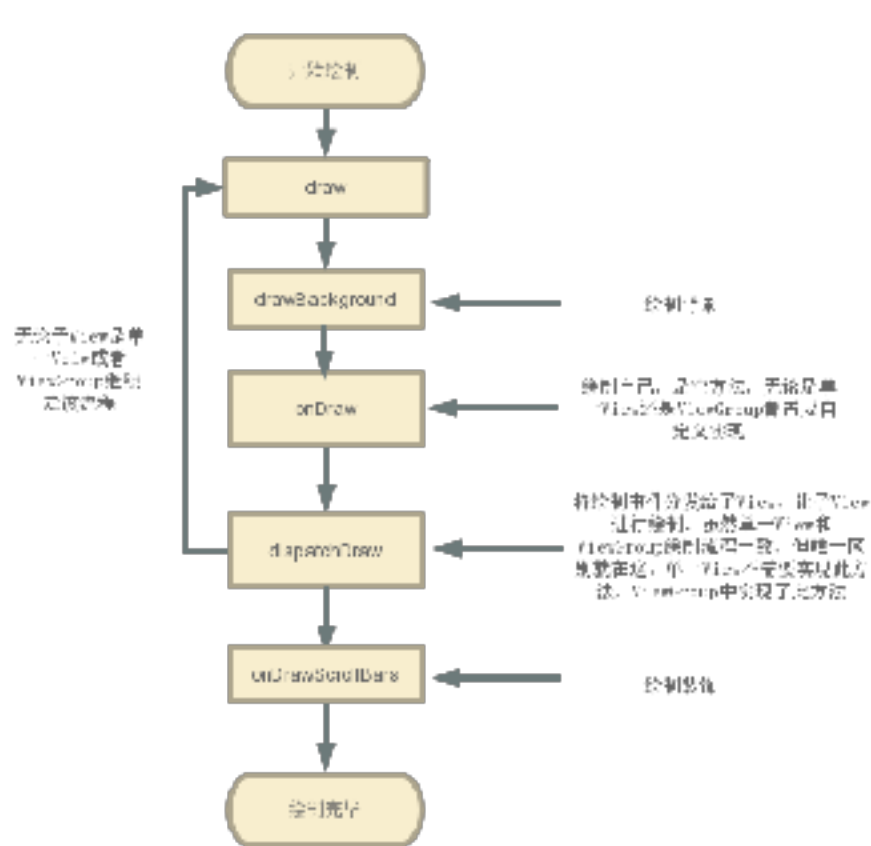
## 扩展：详解ViewTree及View/ViewGroup绘制流程







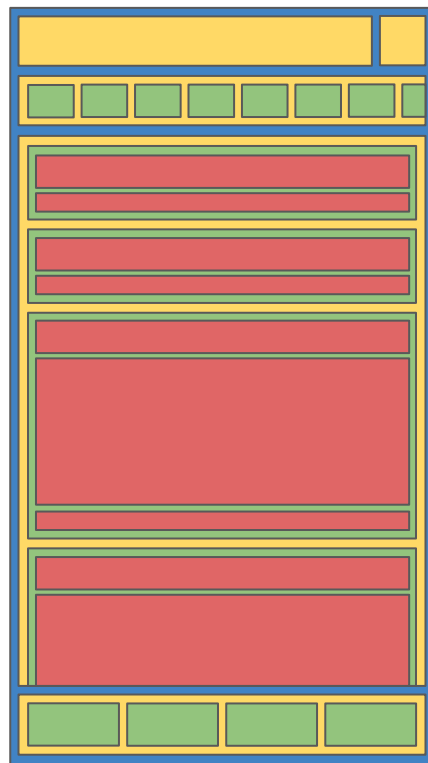
## 扩展：详解ViewTree及View/ViewGroup绘制流程



```
16
17 // Draw traversal performs several drawing steps which must be executed
18 // in the appropriate order:
19 //
20 // 1. Draw the background
21 // 2. If necessary, save the canvas' layers to prepare for fading
22 // 3. Draw view's content
23 // 4. Draw children
24 // 5. If necessary, draw the fading edges and restore layers
25 // 6. Draw decorates (scrollbars for instance)
26
```

ViewGroup的绘制流程  
其中最关键的是 `dispatchDraw` 方法，也是父类 `View` 的实现。

## 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

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

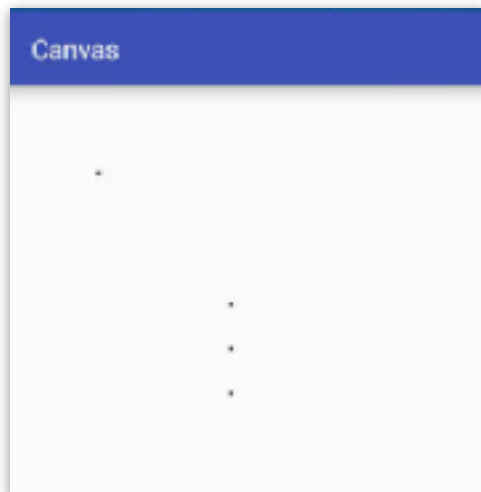
# 自定义View-重写onDraw

概念解析：

1. Canvas：画布
2. Paint：画笔



## 基本绘制-点 (Point)



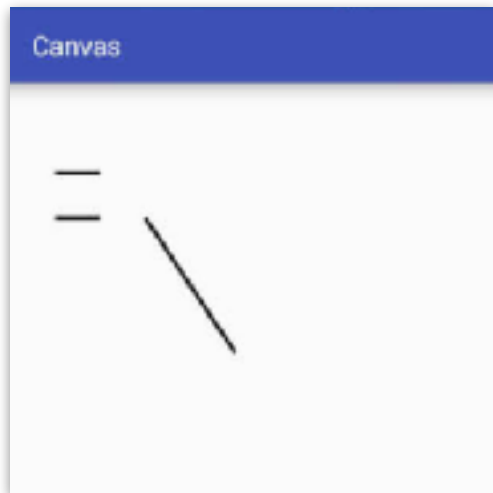
```
private Paint pointPaint;

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);

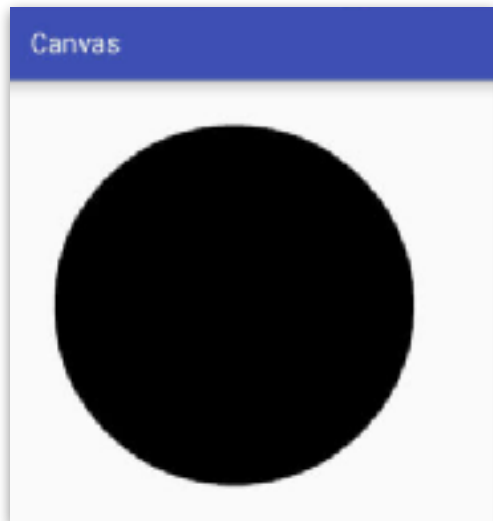
    canvas.drawPoint(x: 200, y: 200, pointPaint); //在坐标[200,200]位置绘制一个点
    canvas.drawPoints(new float[] {
        500, 500,
        500, 600,
        500, 700
    }, pointPaint);
}
```

## 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);

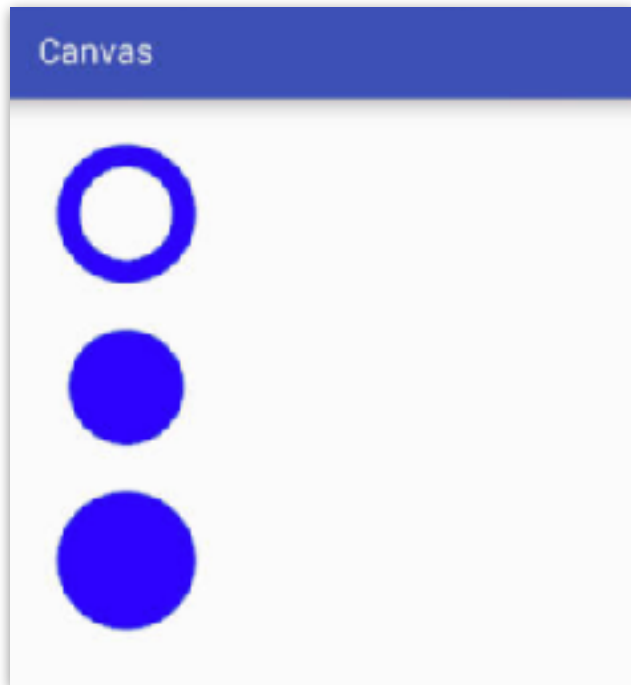
    // 绘制矩形
    canvas.drawRect( left: 100, top: 100, right: 800, bottom: 400, paint);

    // 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);
}

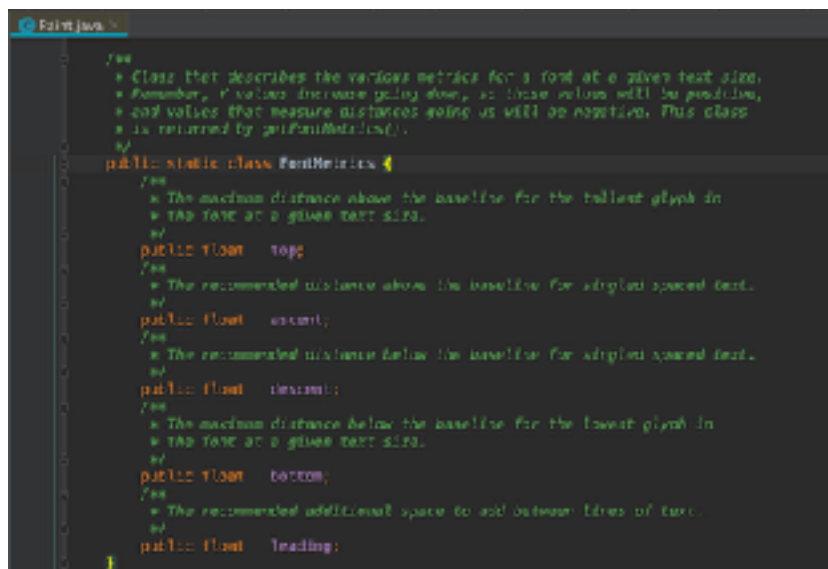
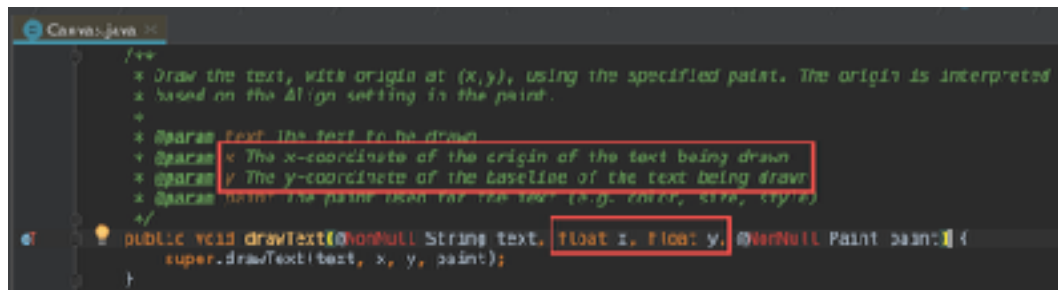
@Override
protected void onDraw(Canvas canvas) {
    super.onDraw(canvas);

    // 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);
}
```

## 绘制文字-基线

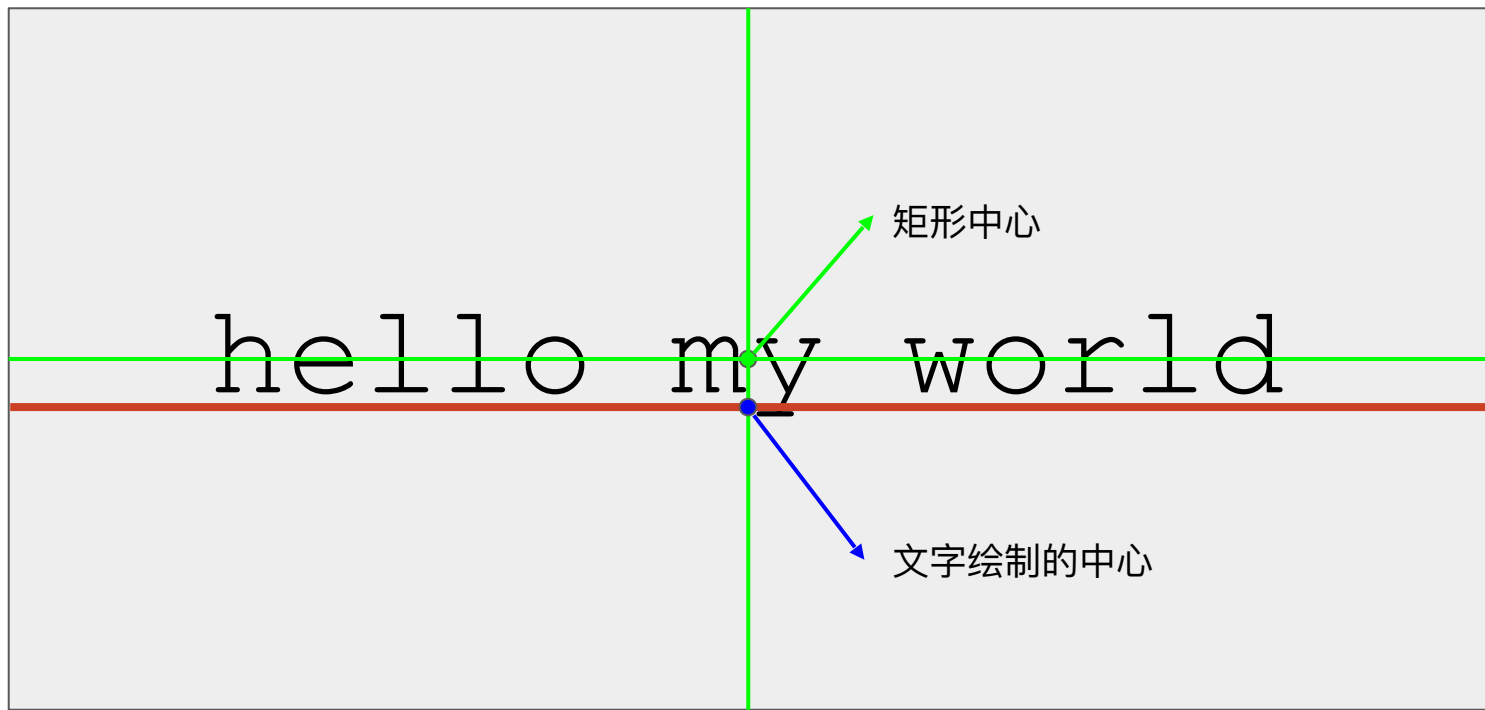


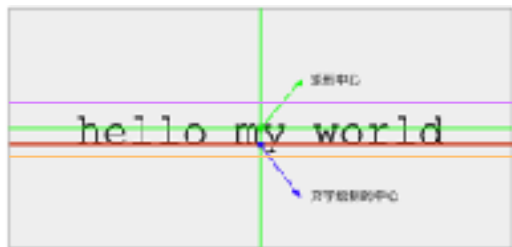
Top  
Ascent  
Baseline  
Descent  
Bottom

My text line 1.



## 基本绘制-文字





Top

Bottom

```
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);  
    textPaint.setTextAlign(Paint.Align.CENTER);  
  
    Paint.FontMetrics fontMetrics = textPaint.getFontMetrics();  
    float top = fontMetrics.top; // 上图中的top  
    float bottom = fontMetrics.bottom; // 上图中的bottom  
    int baseLineY = (int) (centerY + ((bottom - top) / 2 - bottom));  
    canvas.drawText("hello my world", 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：
    - invalidate(如果布局没变化，只触发draw)
    - requestLayout(触发layout、measure)
- 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





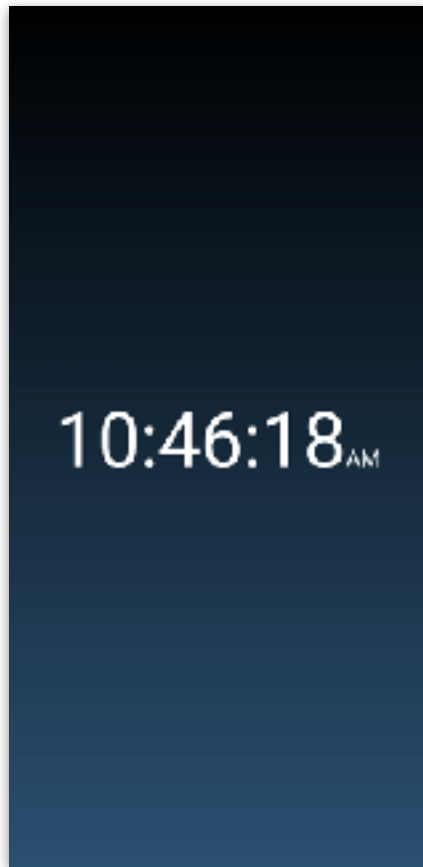
# 时钟App

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
3. 有表盘模式和数字模式，点击页面切换
4. 支持横竖屏切换

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？）





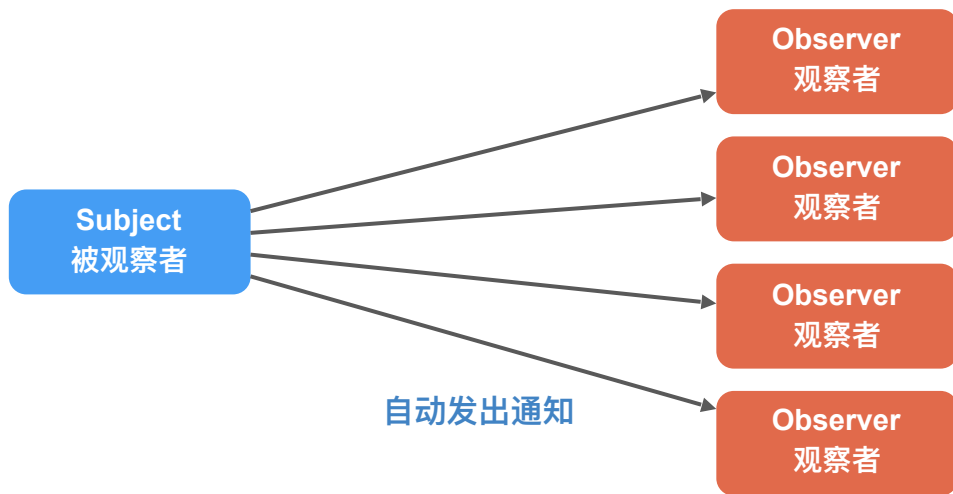


## 附：内存泄露 - Memory Leak

- 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：
  - 我们在抖音上关注了姚晨，姚晨发布新视频时，我们会收到push通知





## 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