复杂应用组件

Handler机制、多线程与自定义View

全亮　liang.tong@bytedance.com
字节跳动Android工程师

••提纲

01 | 进程与线程

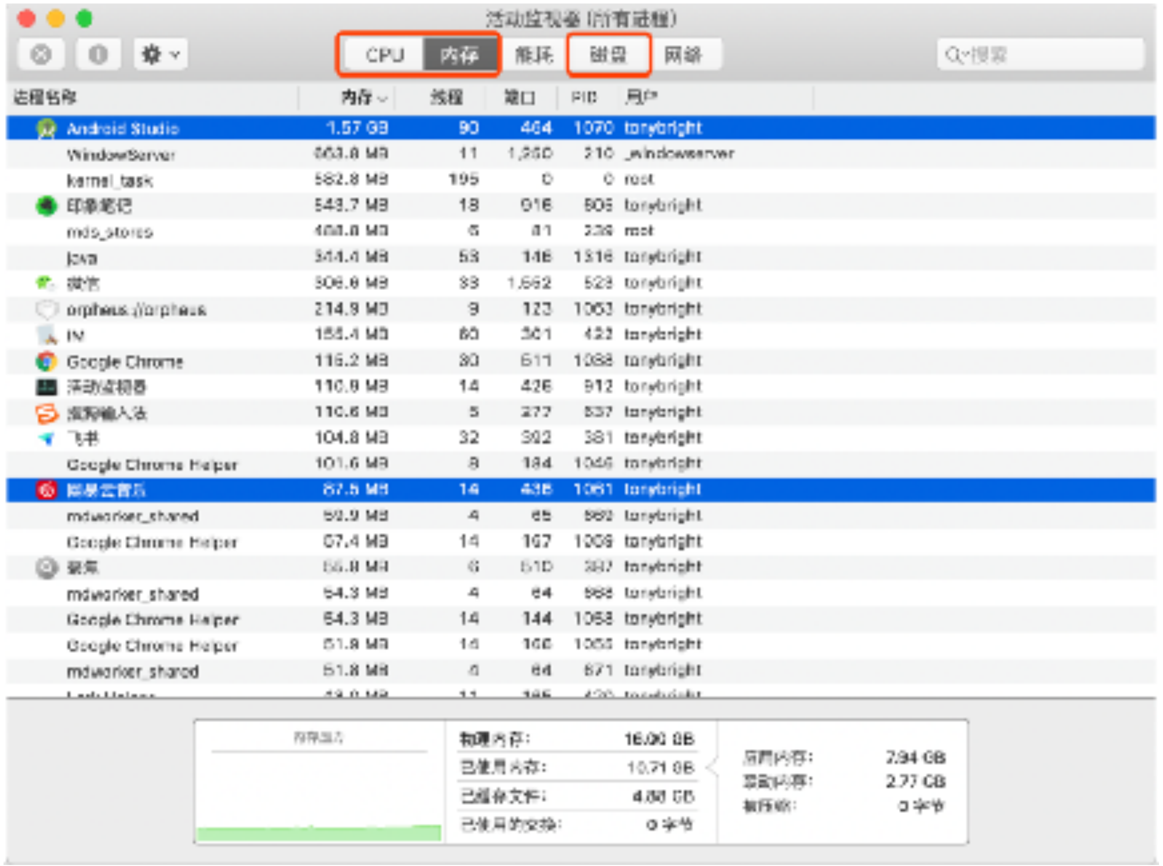
02 | Handler机制

03 | Android中的多线程

04 | 自定义View

进程与线程

进程

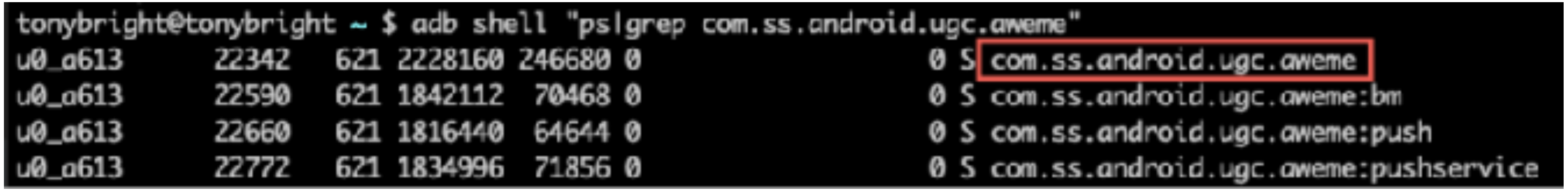


进程名称	内存	线程	端口	PID	用户
Android Studio	1.57 GB	90	464	1070	tonybright
WindowServer	663.8 MB	11	1,260	210	_windowserver
kernel_task	582.8 MB	195	0	0	root
印象笔记	543.7 MB	18	916	805	tonybright
mds_stores	408.8 MB	5	81	228	root
java	344.4 MB	53	146	1316	tonybright
微信	306.8 MB	33	1,662	523	tonybright
orpheus://orpheus	214.9 MB	9	123	1063	tonybright
IN	155.4 MB	60	301	422	tonybright
Google Chrome	116.2 MB	30	611	1088	tonybright
活动监视器	110.8 MB	14	426	912	tonybright
搜狗输入法	110.6 MB	5	277	537	tonybright
飞书	104.8 MB	32	302	381	tonybright
Google Chrome Helper	101.6 MB	8	184	1046	tonybright
网易云音乐	87.5 MB	14	436	1061	tonybright
mdworker_shared	69.9 MB	4	65	869	tonybright
Google Chrome Helper	67.4 MB	14	167	1009	tonybright
聚焦	66.8 MB	6	510	387	tonybright
mdworker_shared	64.3 MB	4	64	868	tonybright
Google Chrome Helper	64.3 MB	14	144	1058	tonybright
Google Chrome Helper	61.8 MB	14	166	1055	tonybright
mdworker_shared	61.8 MB	4	64	871	tonybright

物理内存：	16.00 GB
已使用内存：	10.71 GB
已缓存文件：	4.80 GB
已使用的交换：	0 字节
应用内存：	7.94 GB
联动内存：	2.77 GB
被压缩：	0 字节

进程

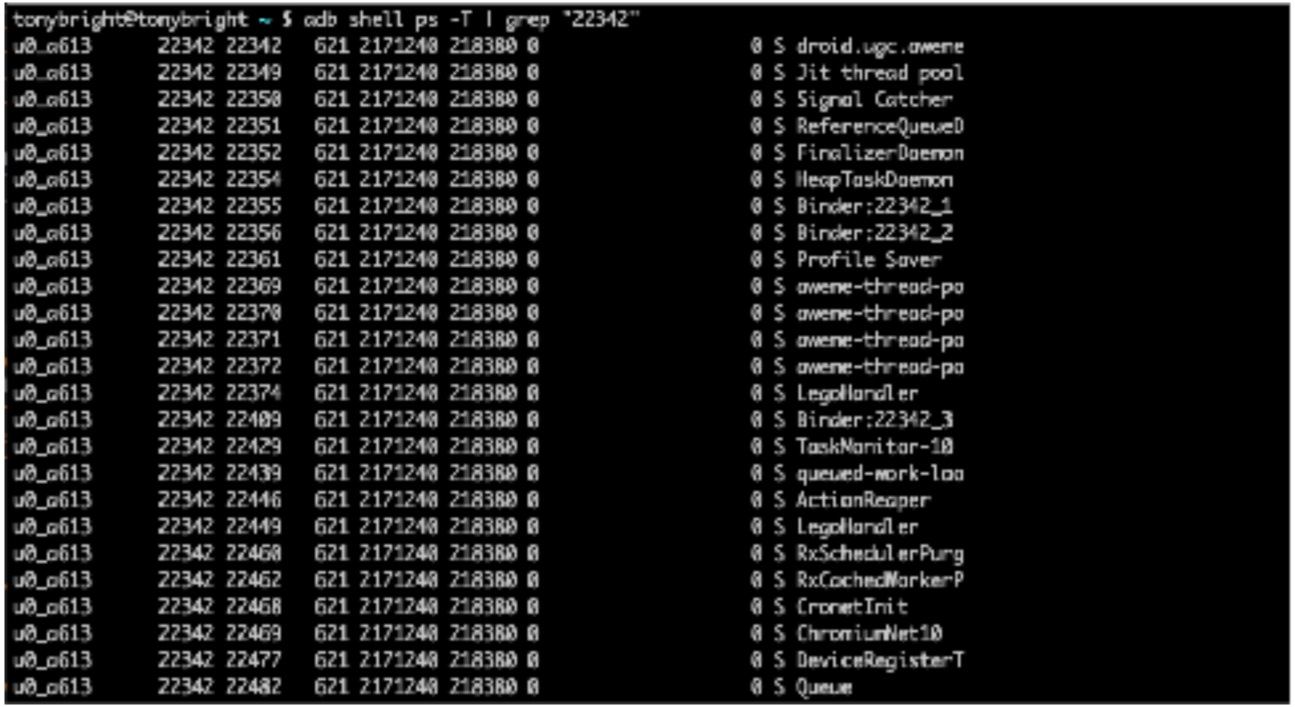
抖音app创建的进程



```
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

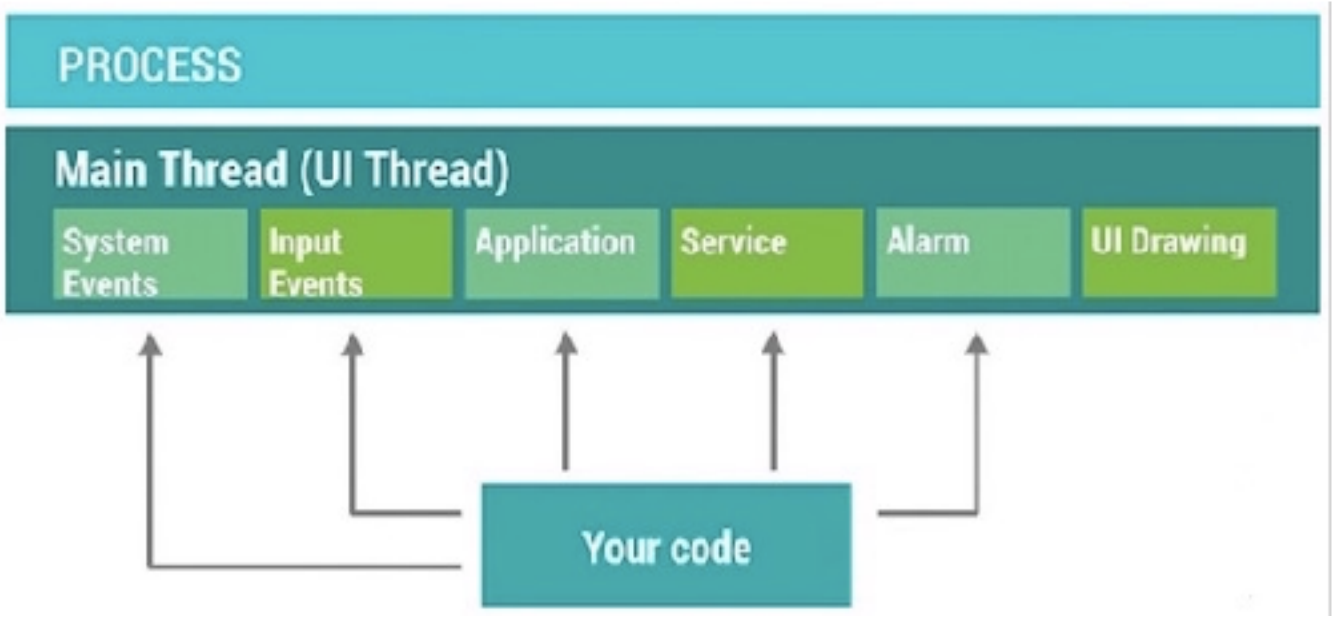


```
tonybright@tonybright ~ $ adb shell ps -T | grep "22342"
u0_a613       22342 22342   621 2171240 218380 0                   0 S droid.ugc.aweme
u0_a613       22342 22349   621 2171240 218380 0                   0 S Jit thread pool
u0_a613       22342 22350   621 2171240 218380 0                   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 aweme-thread-po
u0_a613       22342 22370   621 2171240 218380 0                   0 S aweme-thread-po
u0_a613       22342 22371   621 2171240 218380 0                   0 S aweme-thread-po
u0_a613       22342 22372   621 2171240 218380 0                   0 S aweme-thread-po
u0_a613       22342 22374   621 2171240 218380 0                   0 S LegoHandler
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-loo
u0_a613       22342 22446   621 2171240 218380 0                   0 S ActionReaper
u0_a613       22342 22449   621 2171240 218380 0                   0 S LegoHandler
u0_a613       22342 22460   621 2171240 218380 0                   0 S RxSchedulerPurg
u0_a613       22342 22462   621 2171240 218380 0                   0 S RxCachedWorkerP
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*!

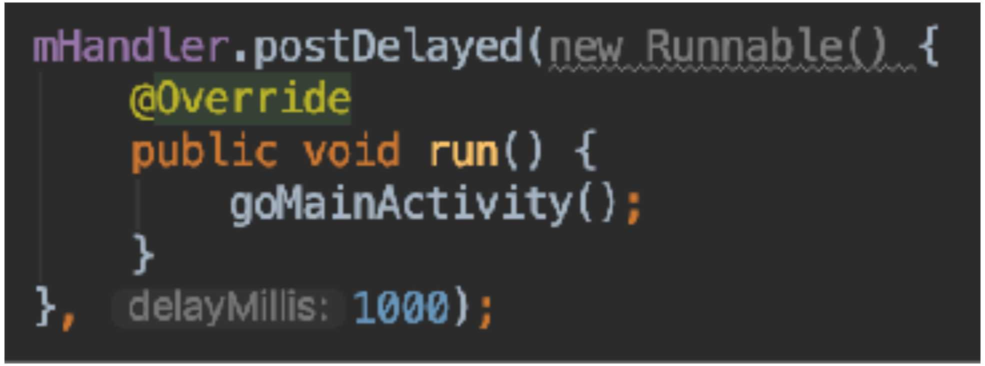
Handler机制

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

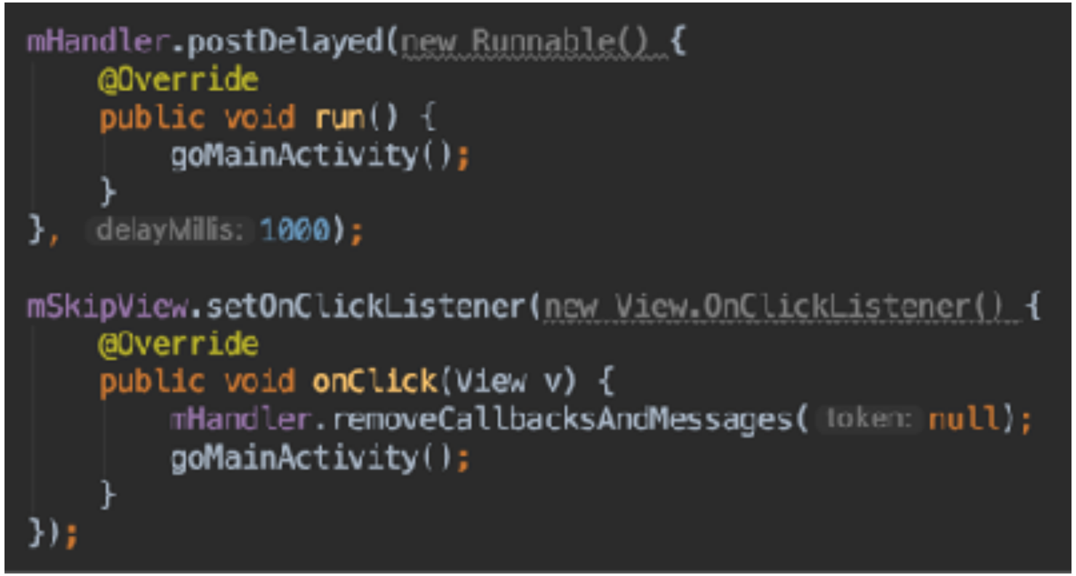
今日头条App启动时，展示了一个开屏广告，默认播放x秒；在x秒后，需跳转到主界面



```java
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();
    }
});
```

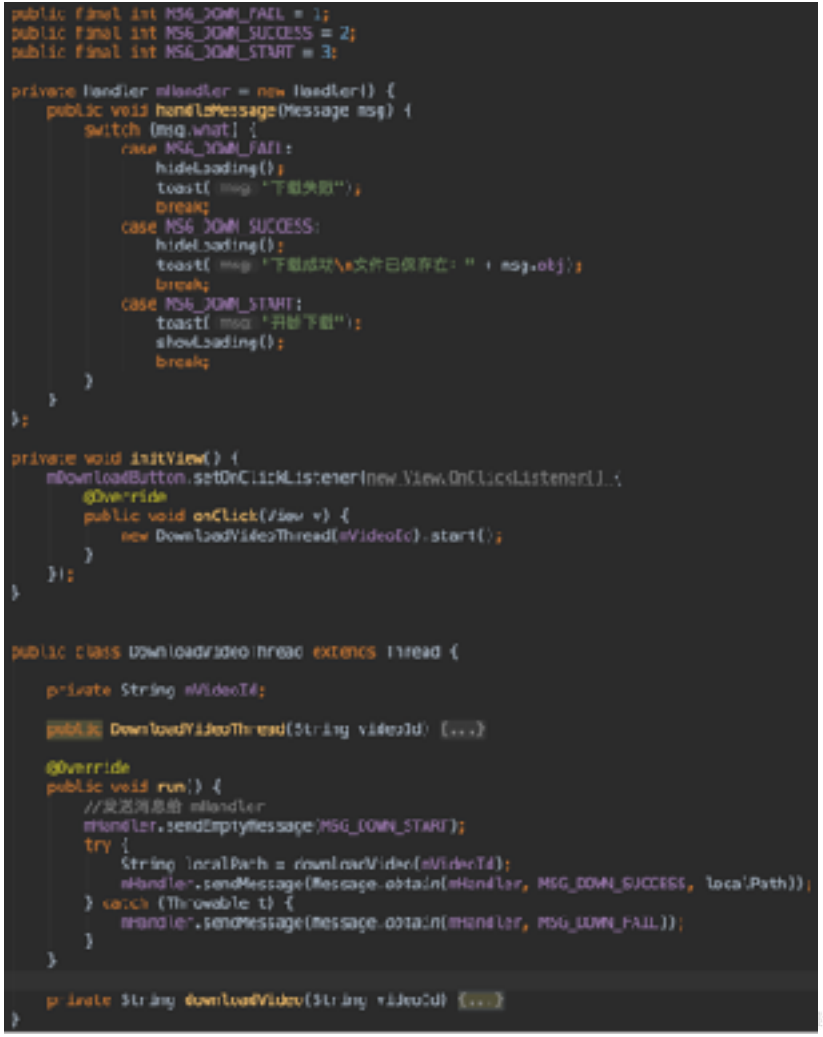
Handler的使用举例

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。



```java
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.sendEmptyMessage(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

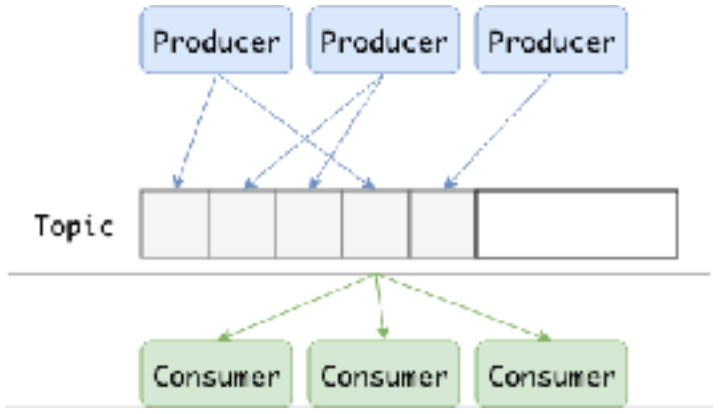
```
// 立即发送消息
public final boolean sendMessage(Message msg)
public final boolean post(Runnable r);

// 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

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

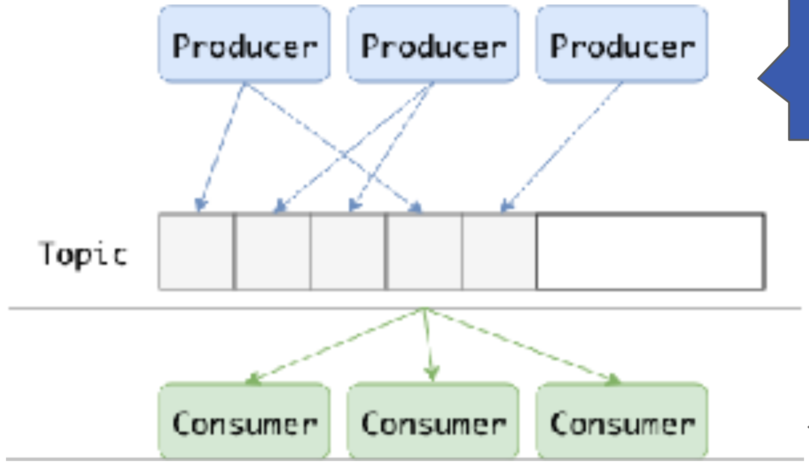
消息队列实际应用：

- 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：



生产者：
1. 用户操作：触摸、点击等
2. 系统事件：如息屏
3. 其他

发送新的消息

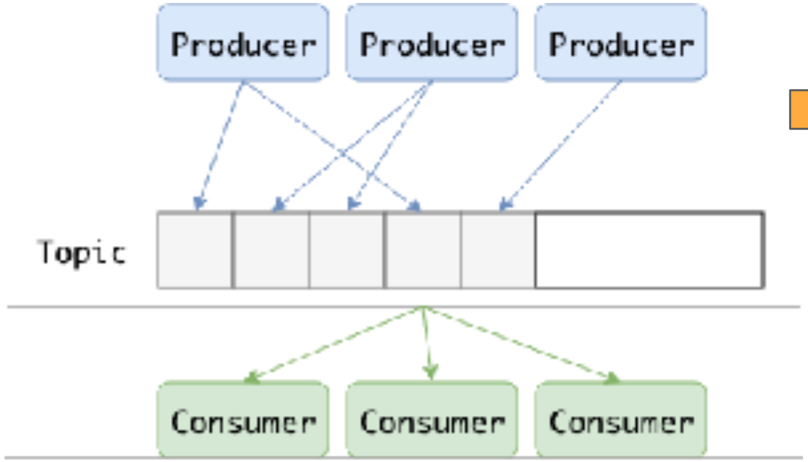
消息队列（待处理任务）

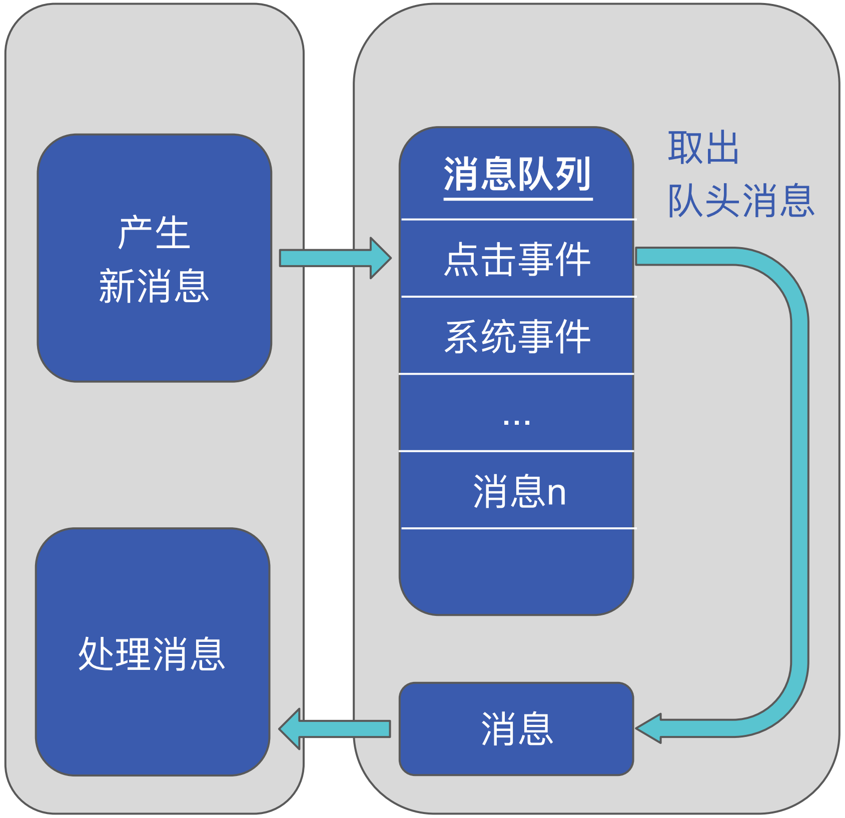
取出一条消息

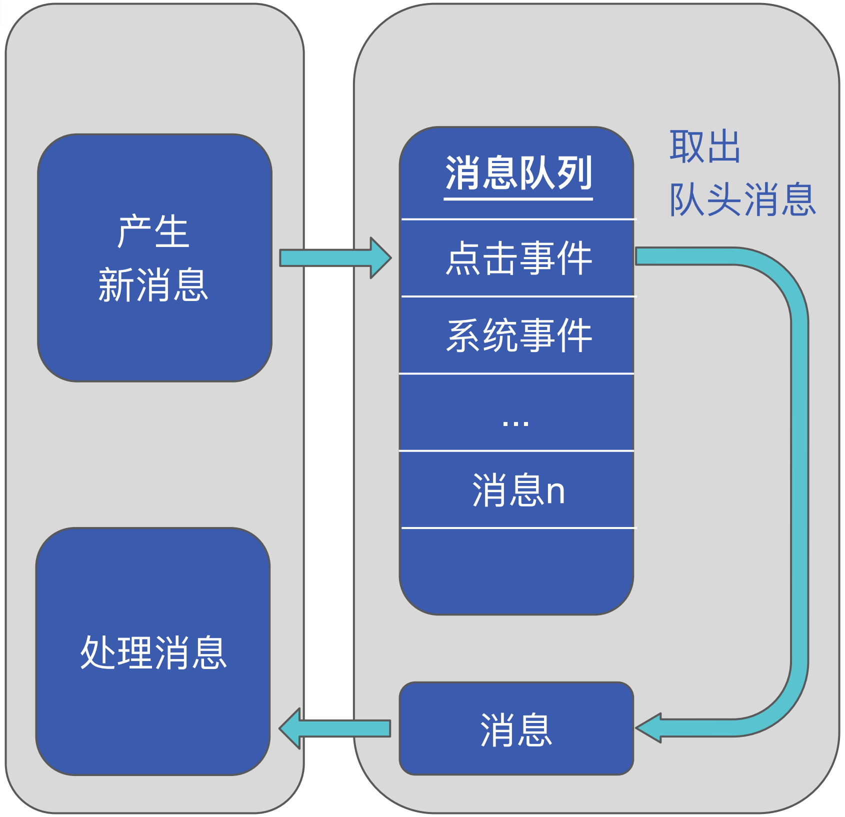
消费者：
1. UI线程

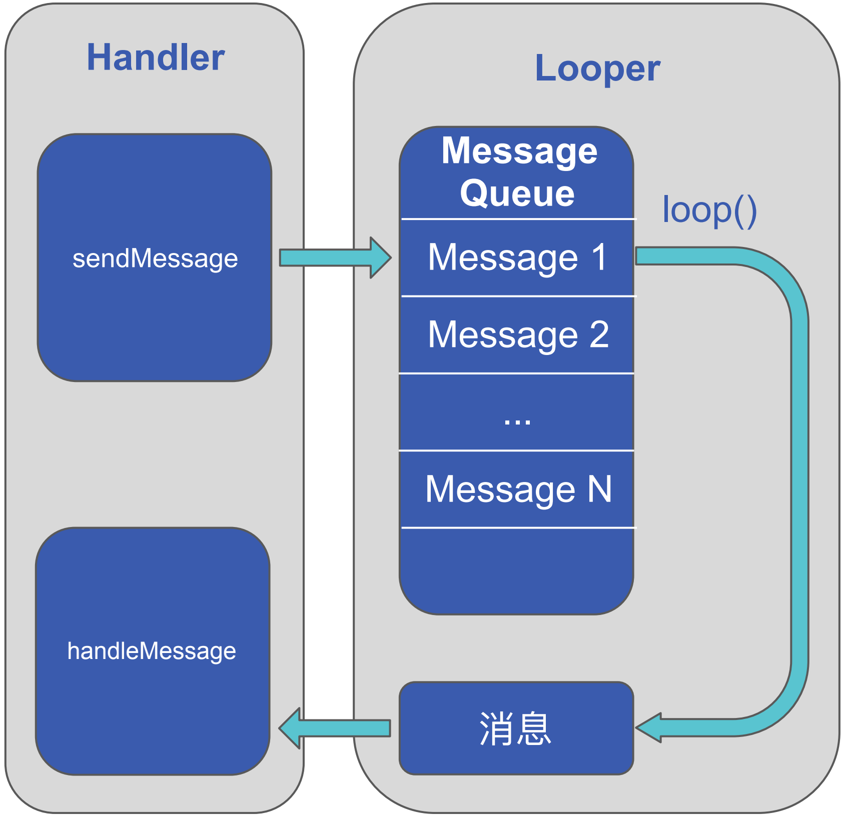
Handler原理：UI线程与消息队列机制

- Android中UI线程负责处理界面的展示，响应用户的操作：

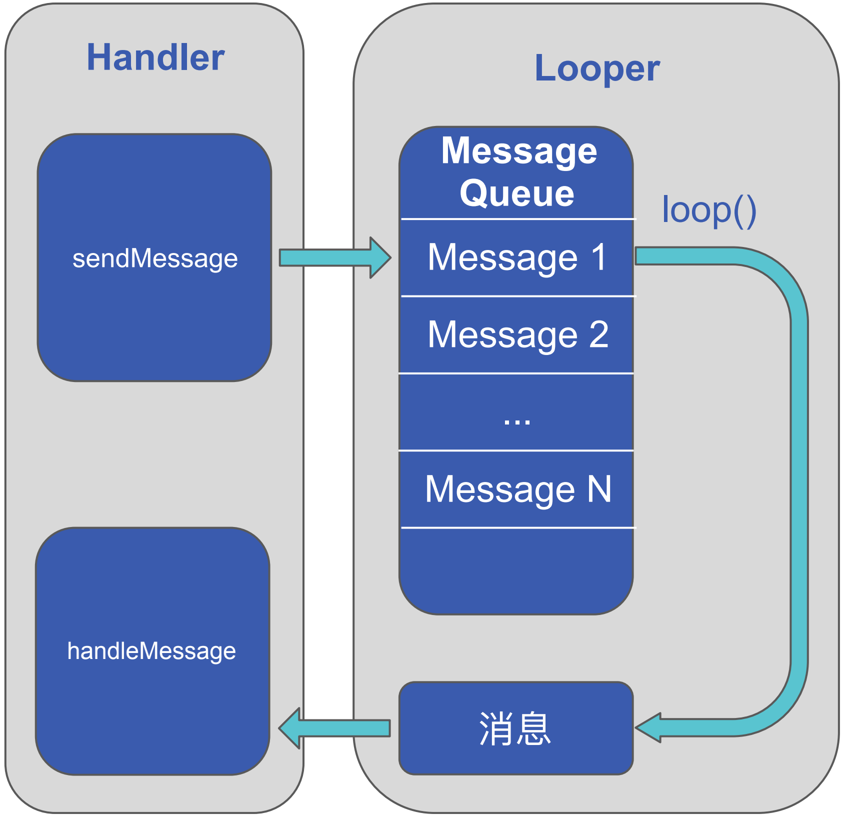





产生
新消息
处理消息
消息队列
点击事件
系统事件
...
消息n
消息
取出
队头消息

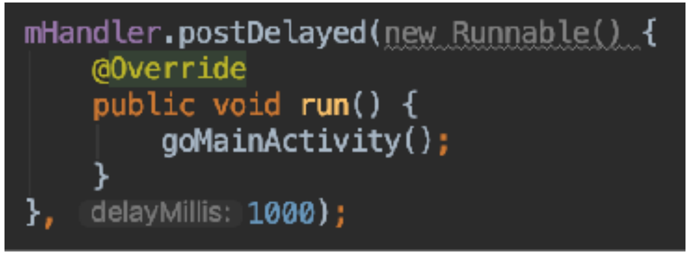

Handler
sendMessage
handleMessage
Looper
Message Queue
Message 1
Message 2
...
Message N
消息
loop()

- 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。



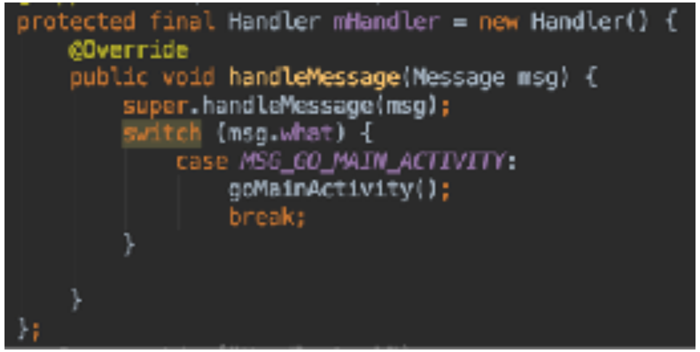
辨析Runnable/Message

1. Runnable会被打包成Message

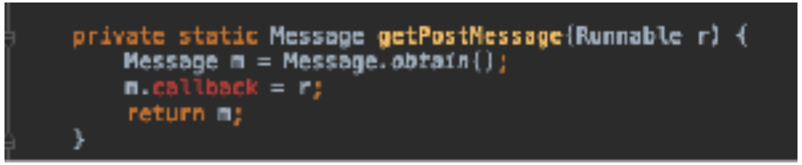


```java
mHandler.postDelayed(new Runnable() {
    @Override
    public void run() {
        goMainActivity();
    }
}, delayMillis: 1000);
```

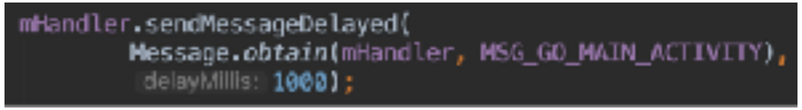
=



```java
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;
        }

    }
};
```



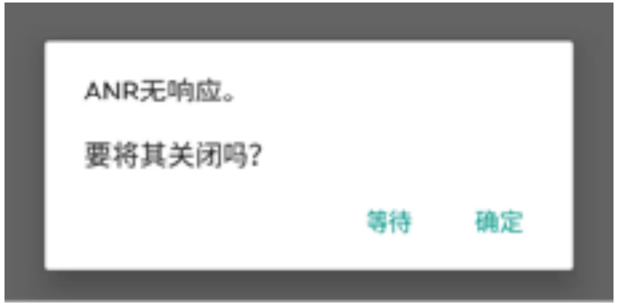
```java
    private static Message getPostMessage(Runnable r) {
        Message m = Message.obtain();
        m.callback = r;
        return m;
    }
```



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

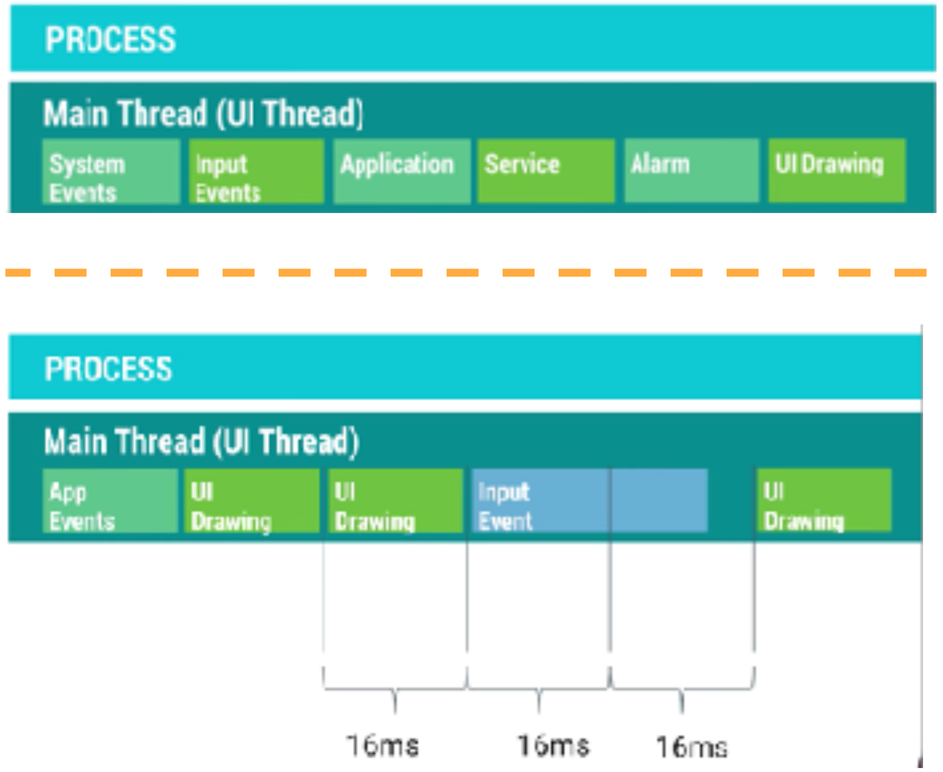
扩展：ANR

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


ANR无响应。

要将其关闭吗?

等待 确定



（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

ByteDance字节跳动

Android里常用的操作多线程方式

01 | Thread（线程）

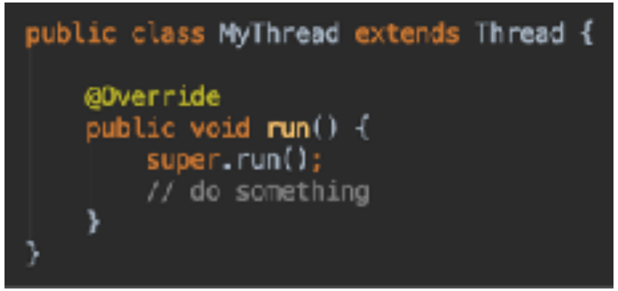
02 | ThreadPool（线程池）

03 | AsyncTask

04 | IntentService

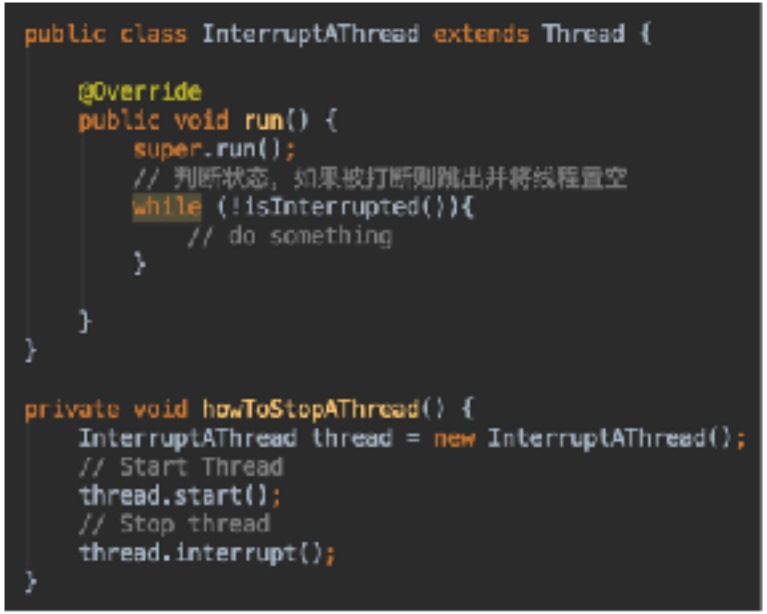
Thread

- Thread（java.lang.Thread）



```java
public class MyThread extends Thread {

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

一个简单的Thread的例子



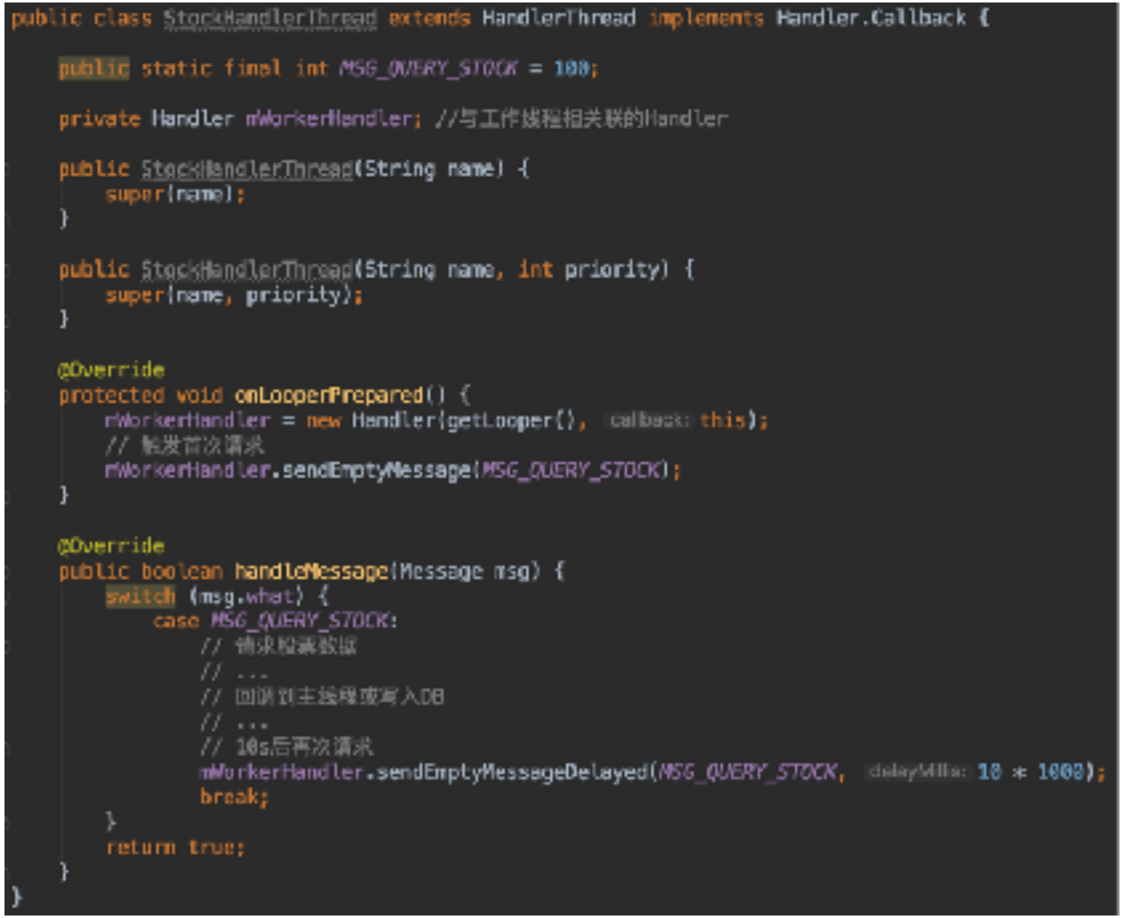
```java
public class InterruptAThread extends Thread {

    @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？

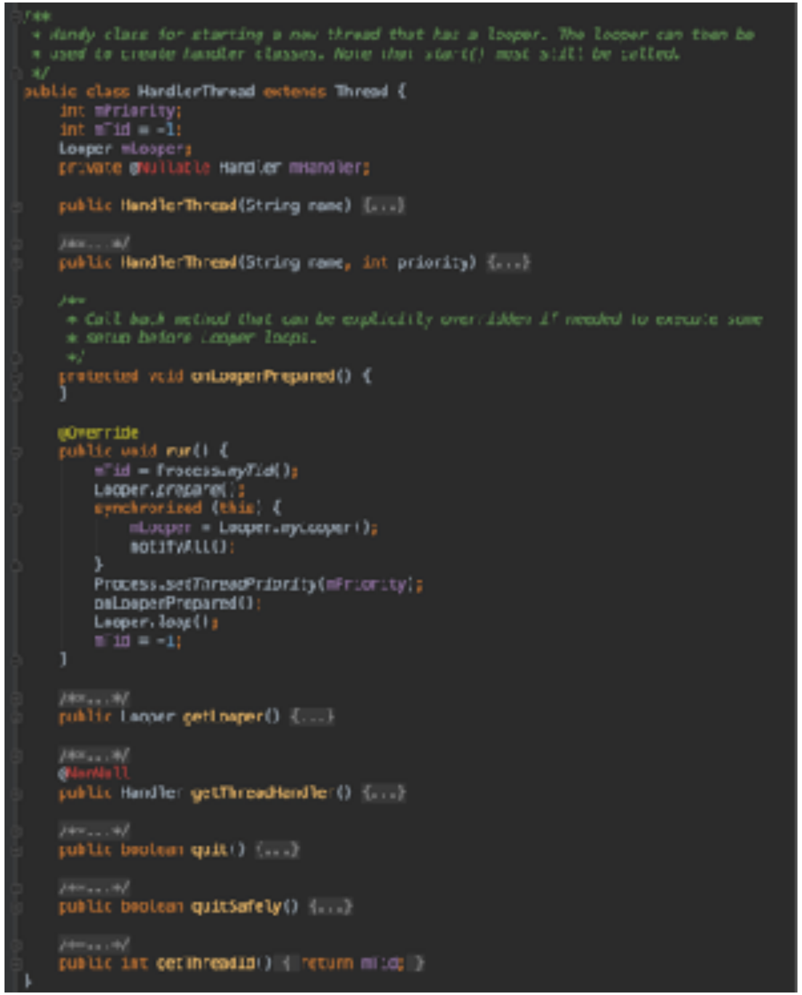
这时可以使用*HandlerThread*



```java
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.sendEmptyMessage(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.sendEmptyMessageDelayed(MSG_QUERY_STOCK, delayMillis: 10 * 1000);
                break;
        }
        return true;
    }
}
```

扩展：HandlerThread

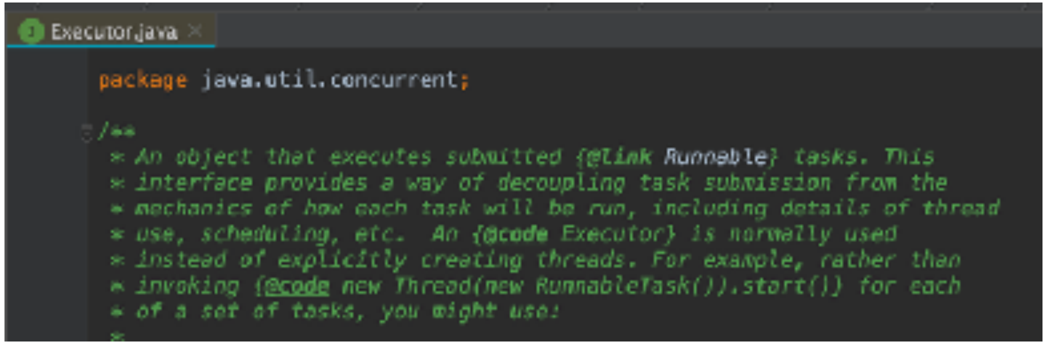
（Handler的实现如右图所示）



```java
/**
 * 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

- 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

```
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?

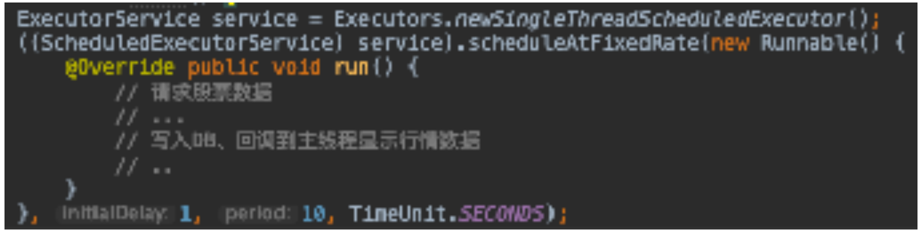
1. 频繁地执行线程创建、销毁，性能开销很大，线程池的线程复用可以有效降低性能开销

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

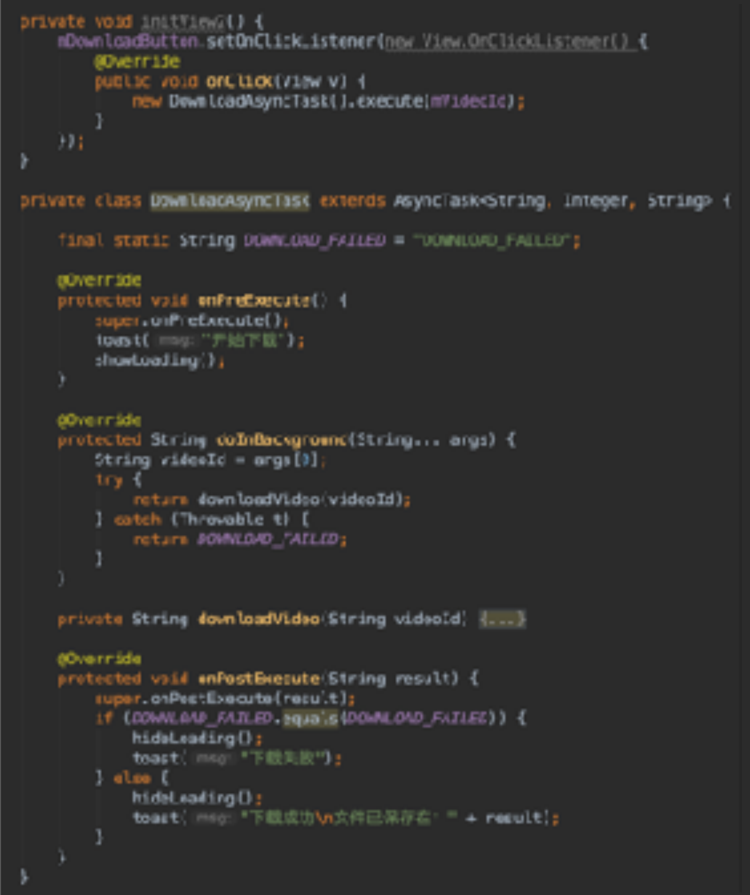


```
ExecutorService service = Executors.newSingleThreadScheduledExecutor();
((ScheduledExecutorService) service).scheduleAtFixedRate(new Runnable() {
    @Override public void run() {
        // 请求股票数据
        // ...
        // 写入DB、回调到主线程显示行情数据
        // ..
    }
}, initialDelay: 1, period: 10, TimeUnit.SECONDS);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

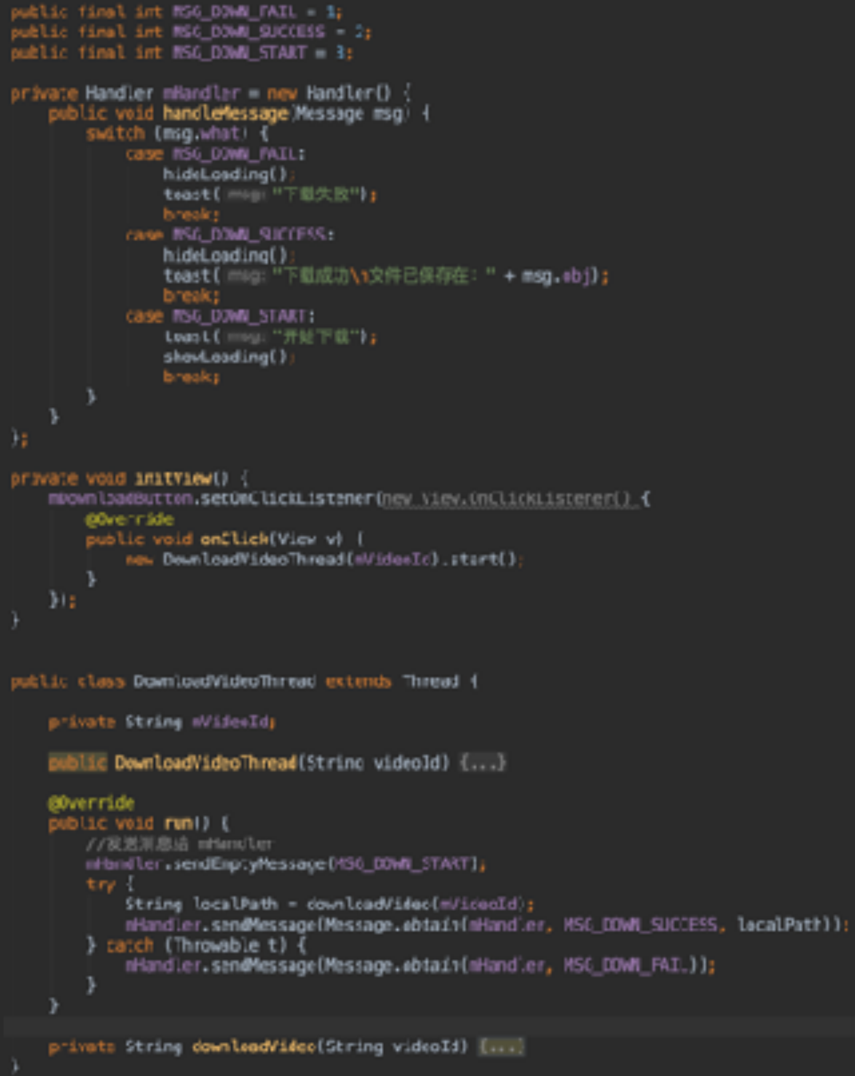
AsyncTask

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。



```java
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);
        }
    }
}
```

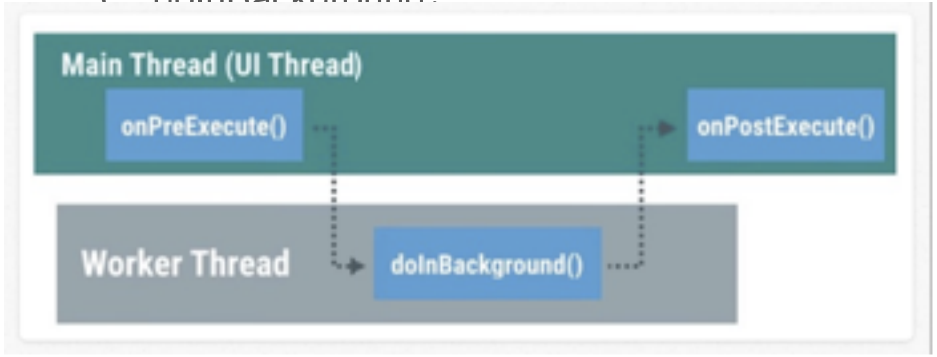


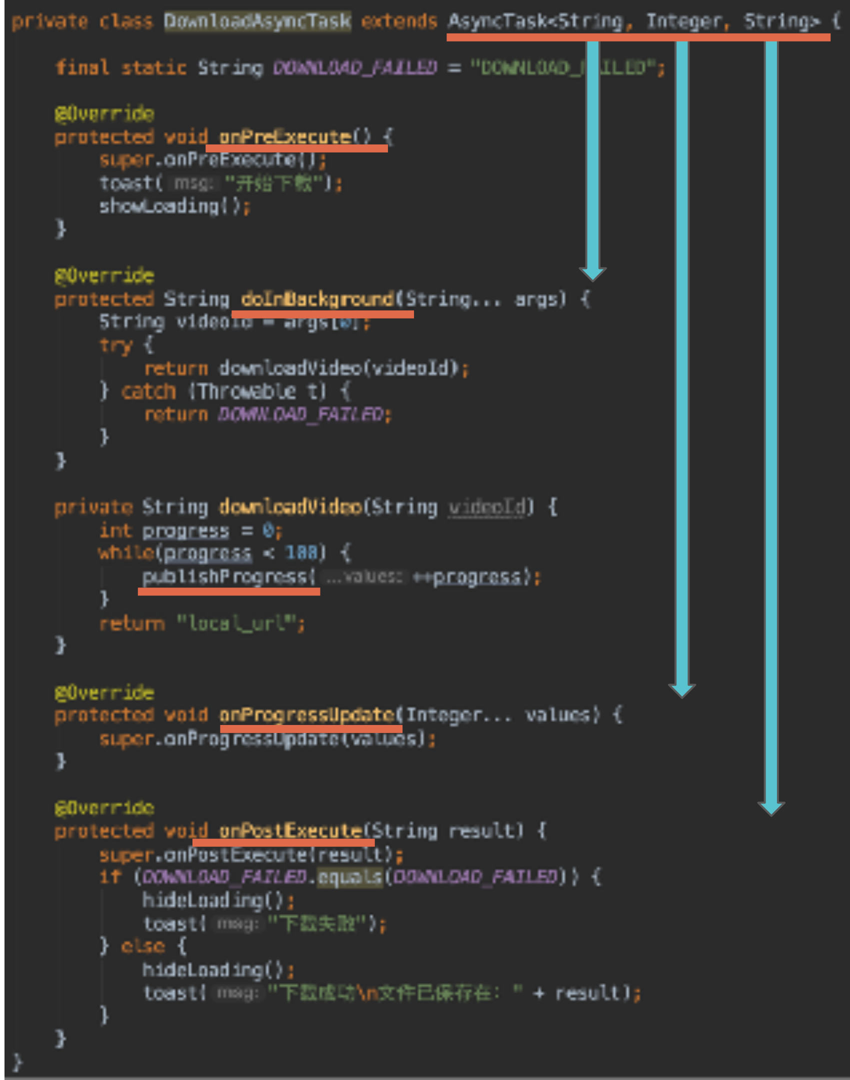
```java
public final int MSG_DOWN_FAIL = 1;
public final int MSG_DOWN_SUCCESS = 2;
public final int MSG_DOWN_START = 3;

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.sendEmptyMessage(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);
        }
    }
}
```

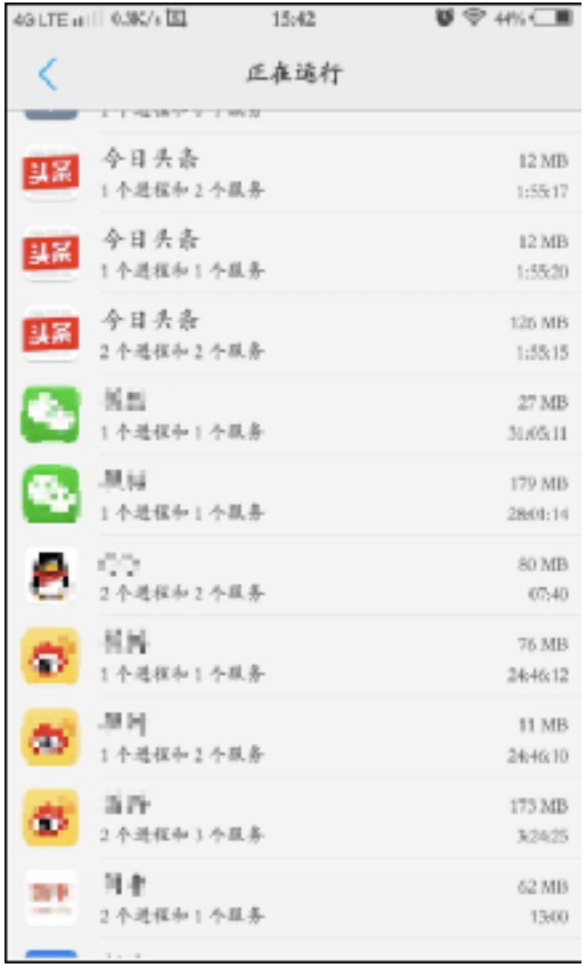
IntentService

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



4G LTE 0.3K/s 15:42 44%

正在运行

今日头条 12 MB
1个进程和2个服务 1:55:17

今日头条 12 MB
1个进程和1个服务 1:55:20

今日头条 126 MB
2个进程和2个服务 1:55:15

[illegible] 27 MB
1个进程和1个服务 31:05:11

[illegible] 179 MB
1个进程和1个服务 28:01:14

[illegible] 80 MB
2个进程和2个服务 07:40

[illegible] 76 MB
1个进程和1个服务 24:46:12

[illegible] 11 MB
1个进程和2个服务 24:46:10

[illegible] 173 MB
2个进程和3个服务 32:25

[illegible] 62 MB
2个进程和1个服务 13:00

IntentService

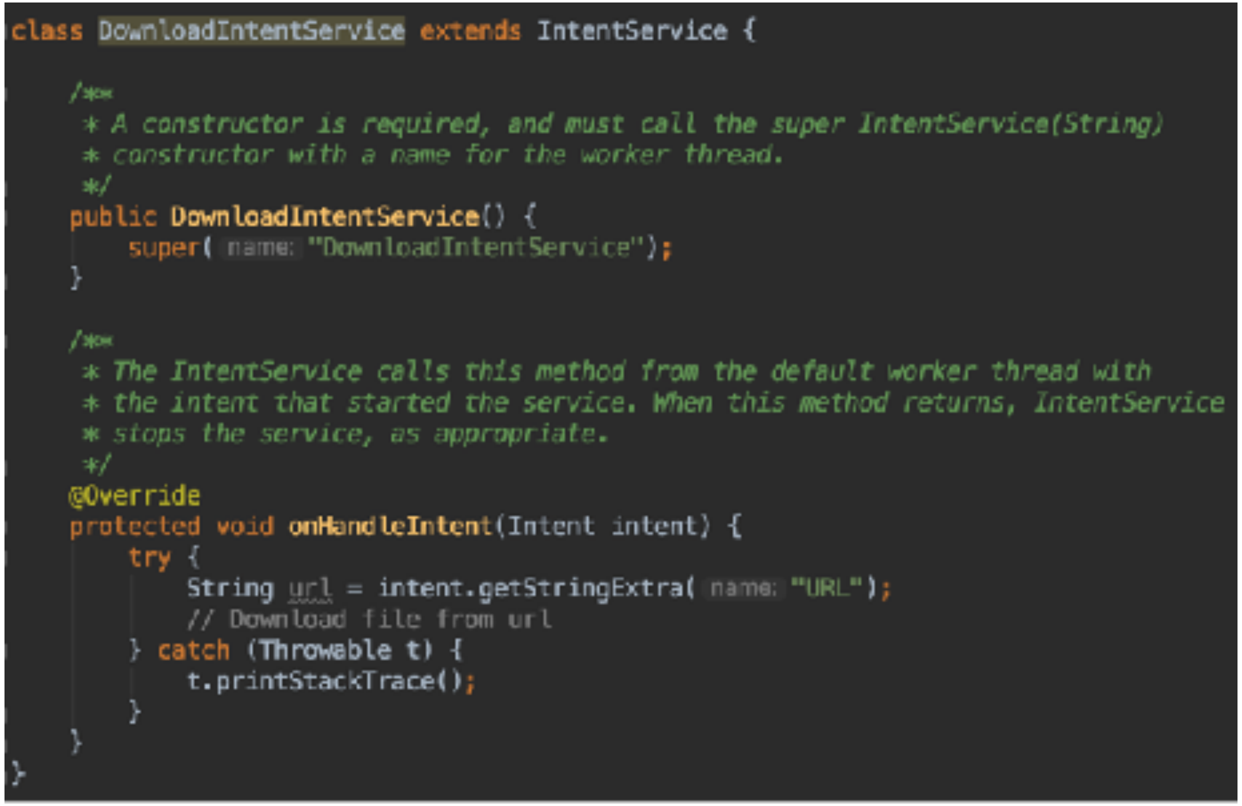
那什么是IntentService?

　　*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*。

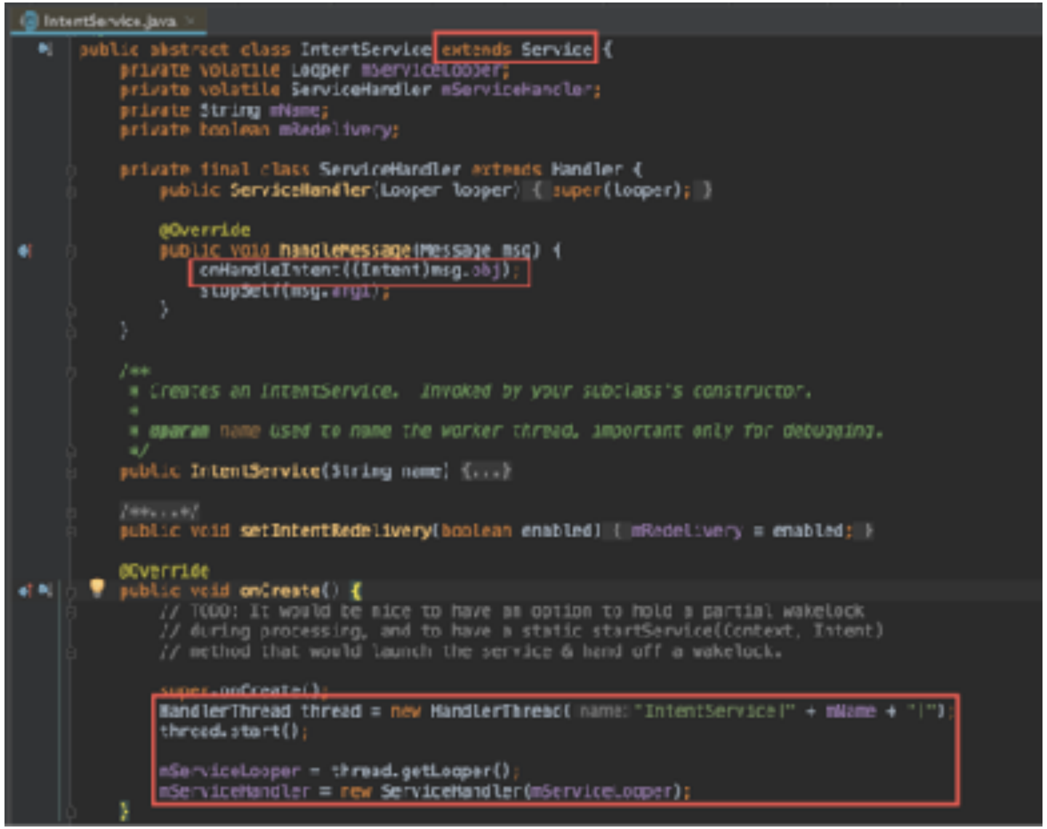
　　比如：用*Service*下载文件

IntentService示例



```java
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();
        }
    }
}
```

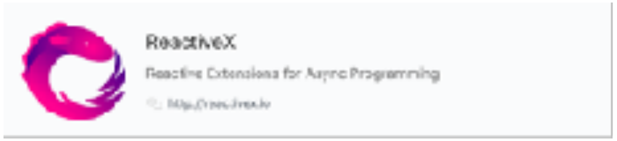
IntentService源码



IntentService.java

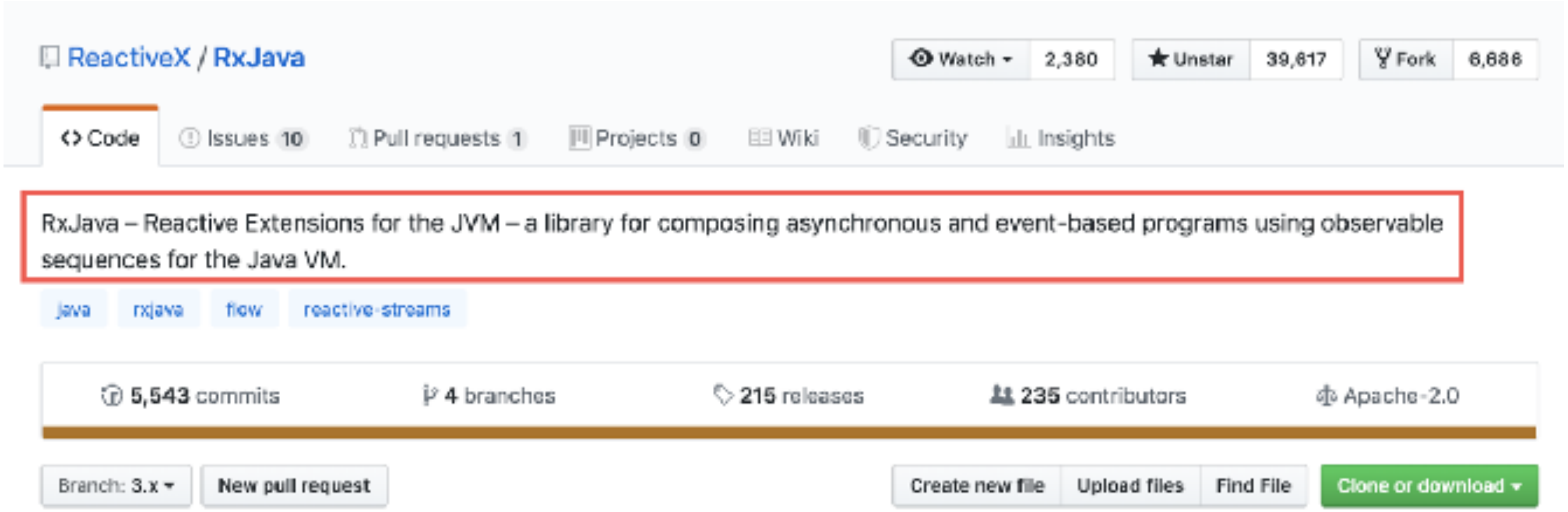
```java
public abstract class IntentService extends Service {
    private volatile Looper mServiceLooper;
    private volatile ServiceHandler mServiceHandler;
    private String mName;
    private boolean mRedelivery;

    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) {...}

    /**...*/
    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( name: "IntentService[" + mName + "]");
        thread.start();

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

扩展：RxJava - 简单介绍


ReactiveX

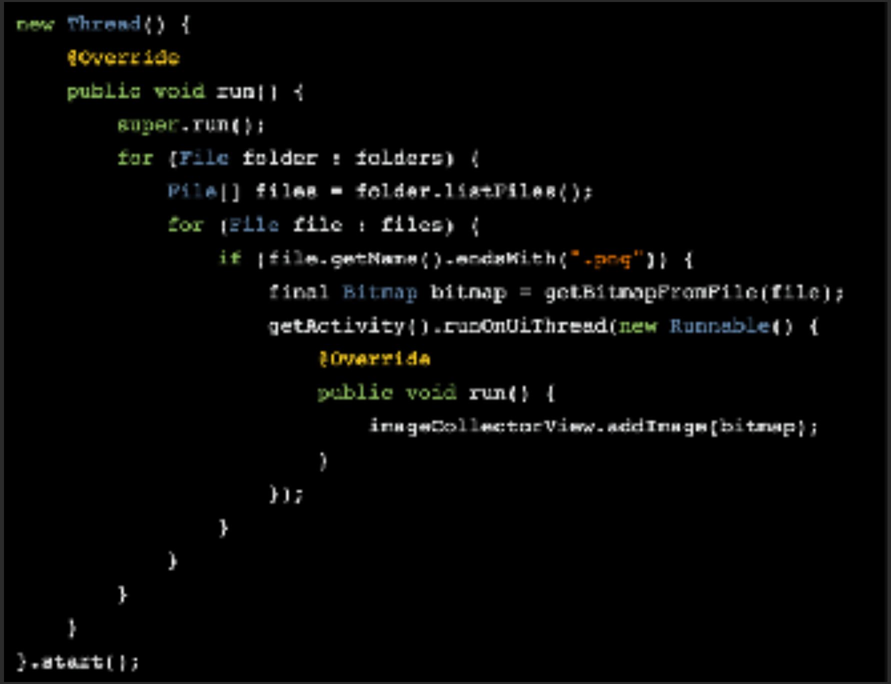
Reactive Extensions for Async Programming

http://reactivex.io



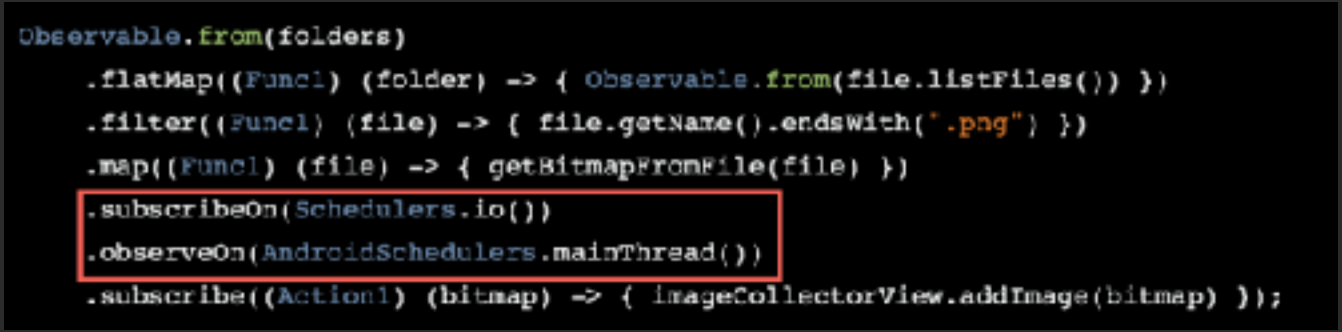
RxJava – Reactive Extensions for the JVM – a library for composing asynchronous and event-based programs using observable sequences for the Java VM.

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();
```

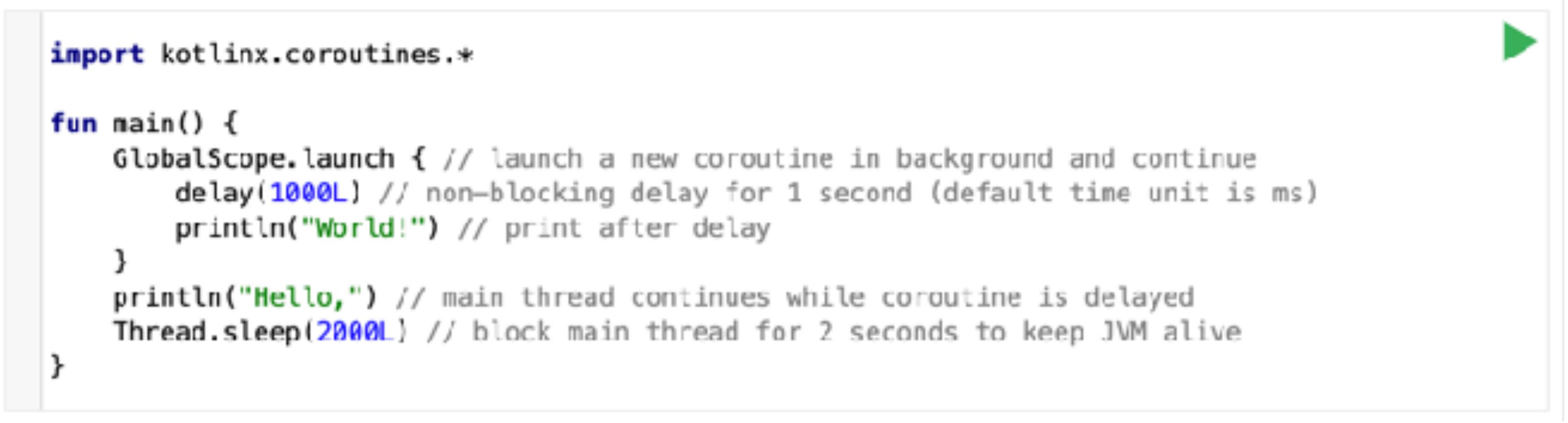
RxJava：



```
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) });
```

扩展：Kotlin - Coroutines(协程)

Essentially, coroutines are light-weight threads.



```kotlin
import kotlinx.coroutines.*

fun main() {
    GlobalScope.launch { // launch a new coroutine in background and continue
        delay(1000L) // non-blocking delay for 1 second (default time unit is ms)
        println("World!") // print after delay
    }
    println("Hello,") // main thread continues while coroutine is delayed
    Thread.sleep(2000L) // block main thread for 2 seconds to keep JVM alive
}
```

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、Coroutine	当下流行的开发框架下的线程调度方式

自定义View

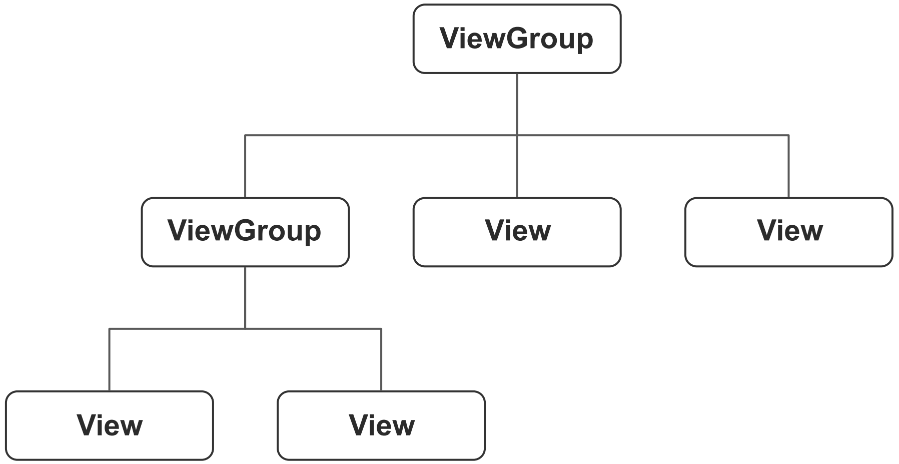
怎么画一个佩奇?

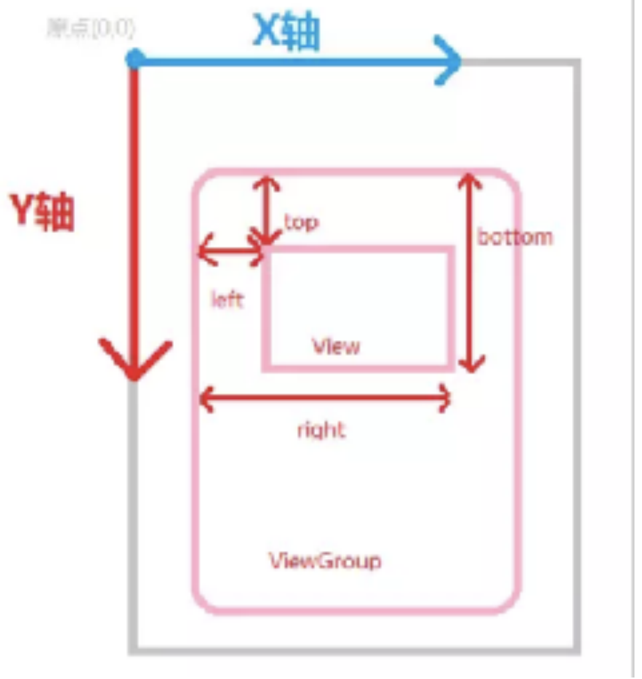
Measure: 测量宽高

Layout: 计算布局

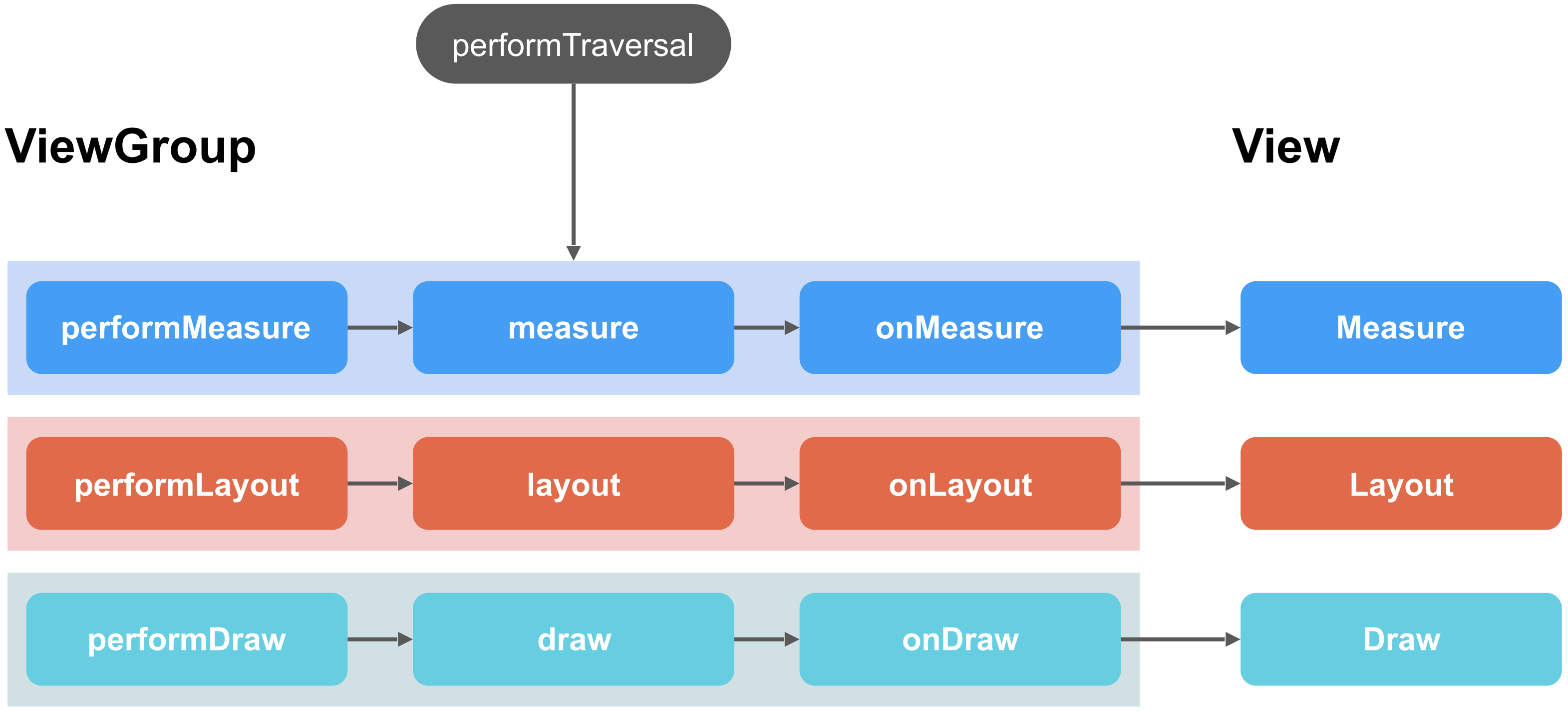
Draw：绘制形状

扩展：详解 ViewTree 及 View / ViewGroup 绘制流程

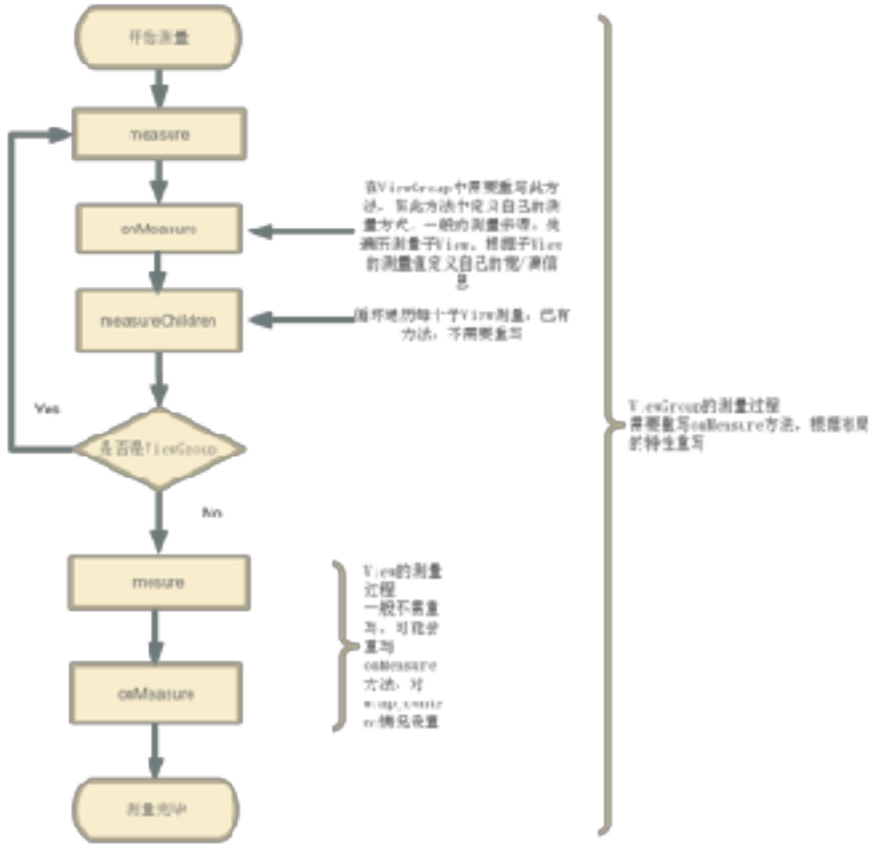




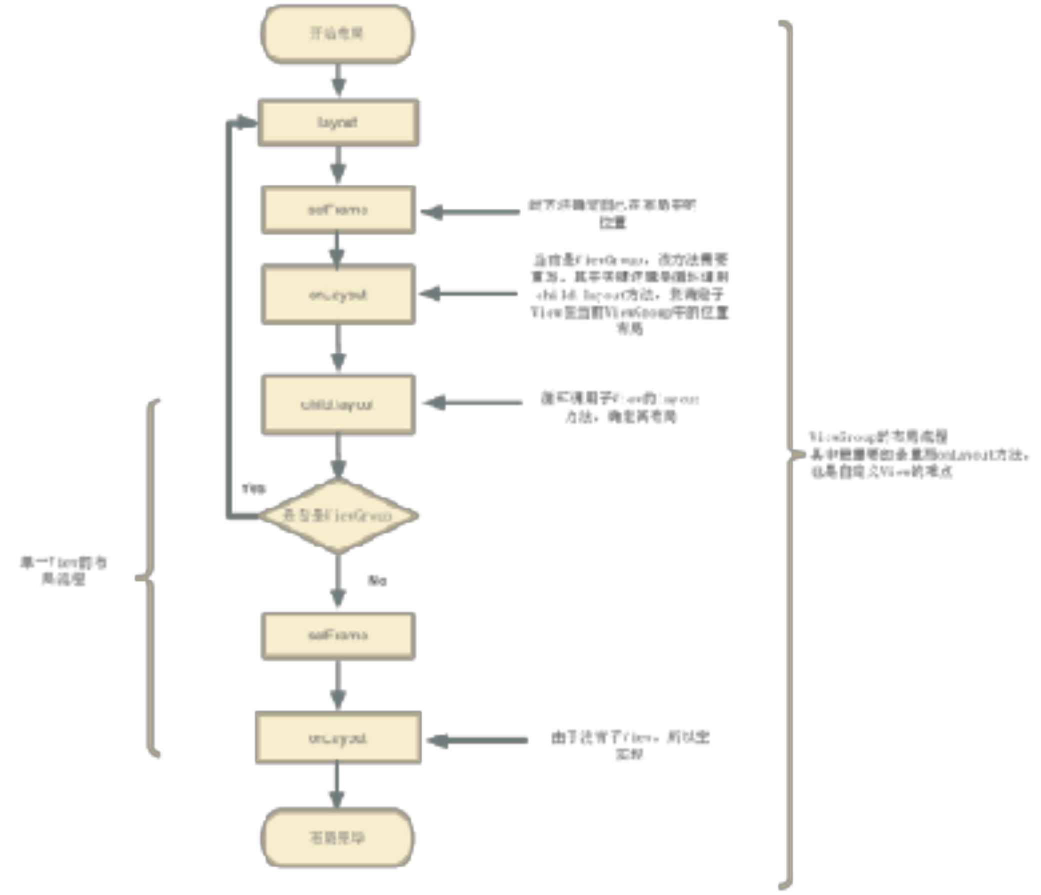
扩展：ViewGroup的绘制流程



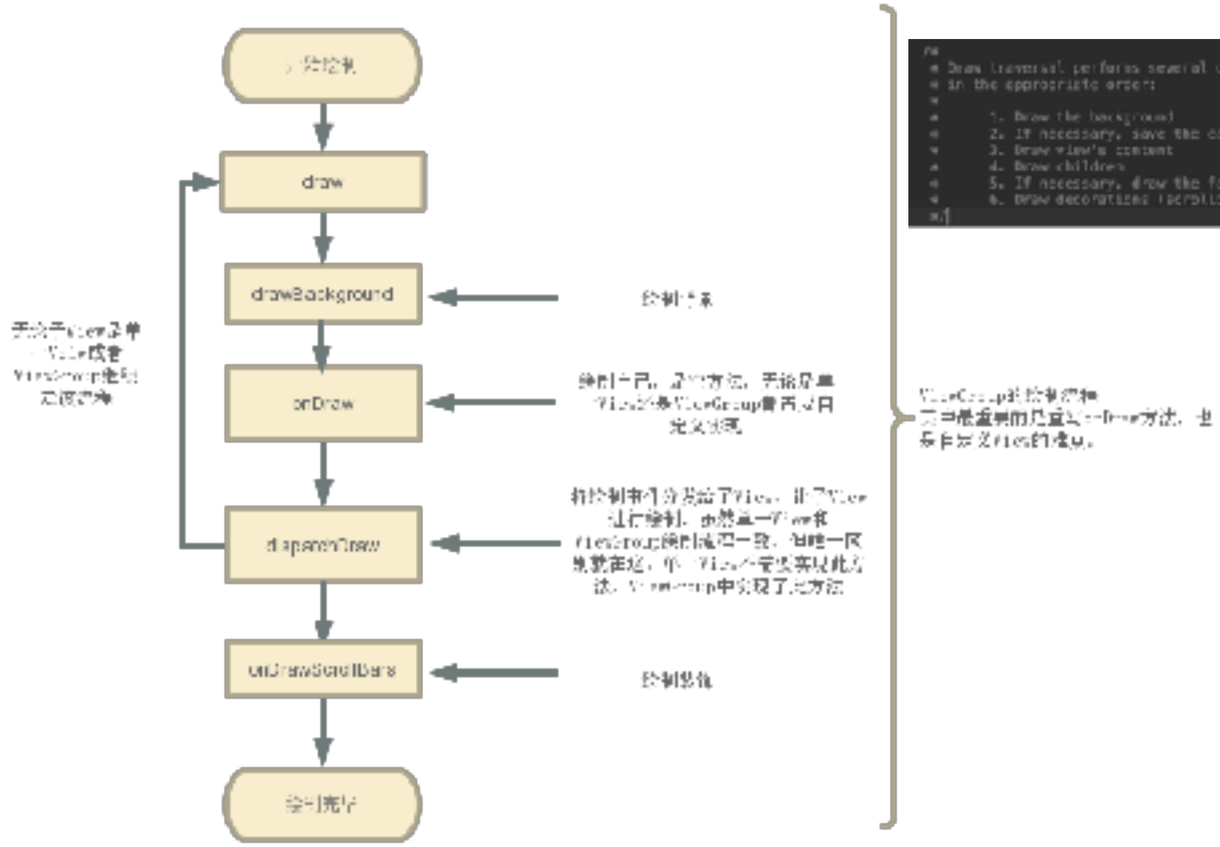
扩展：详解ViewTree及View/ViewGroup绘制流程

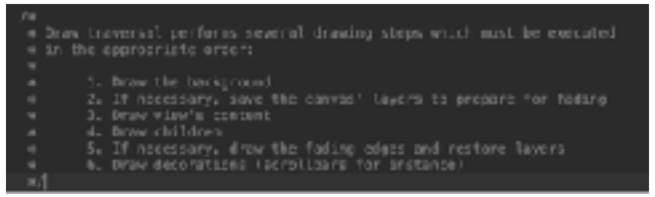

开始测量
measure
onMeasure
在ViewGroup中需要重写此方法，在此方法中定义自己的测量方式，一般的测量步骤，先遍历测量子View，根据子View的测量值定义自己的宽/高信息
measureChildren
循环遍历每个子View测量，已有方法，不需要重写
Yes
是否是ViewGroup
No
mesure
onMeasure
View的测量过程
一般不需重写，可能会重写onMeasure方法，对wrap_content情况设置
测量完毕
ViewGroup的测量过程
需要重写onMeasure方法，根据布局的特性重写

扩展：详解ViewTree及View/ViewGroup绘制流程


开始布局
layout
setFrame
根据下传参数设置自己在布局中的位置
onLayout
当前是ViewGroup，该方法需要重写。其中需要遍历调用所有子元素的child.layout方法，来确定子View在当前ViewGroup中的位置布局
child.layout
遍历调用子View的layout方法，确定其布局
Yes
是否是ViewGroup
No
单一View的布局流程
setFrame
onLayout
由于没有子View，所以空实现
布局完毕
ViewGroup的布局流程
其中最重要的就是重写onLayout方法，也是自定义View的难点

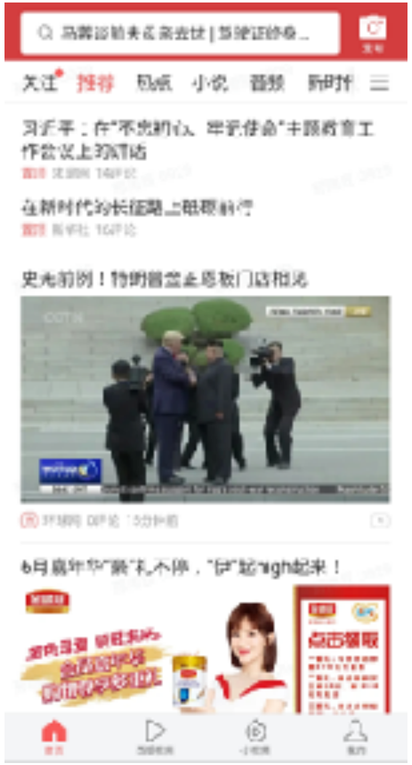
扩展：详解ViewTree及View/ViewGroup绘制流程


开始绘制
draw
drawBackground
onDraw
dispatchDraw
onDrawScrollBars
绘制完毕
无论是View还单一View或者ViewGroup绘制流程都一样
绘制背景
绘制自己，onDraw方法，无论是单一View还是ViewGroup都需要自己完成实现
将绘制事件分发给子View，由子View进行绘制，当然单一View和ViewGroup绘制流程一致，但唯一区别就在这，单一View不需要实现此方法，ViewGroup中实现了此方法
绘制装饰
ViewGroup的绘制流程
其中最重要的是重写onDraw方法，也是自定义View的重点。



```
/*
 * Draw traversal performs several drawing steps which must be executed
 * in the appropriate order:
 *
 *      1. Draw the background
 *      2. If necessary, save the canvas' layers to prepare for fading
 *      3. Draw view's content
 *      4. Draw children
 *      5. If necessary, draw the fading edges and restore layers
 *      6. Draw decorations (scrollbars for instance)
 */
```

扩展：详解ViewTree及View/ViewGroup绘制流程

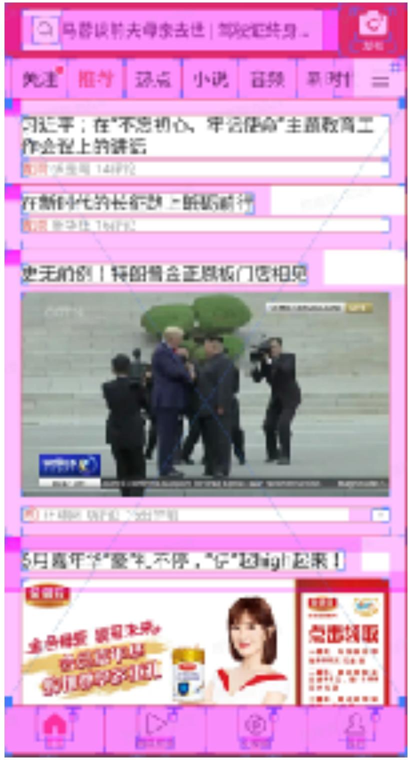


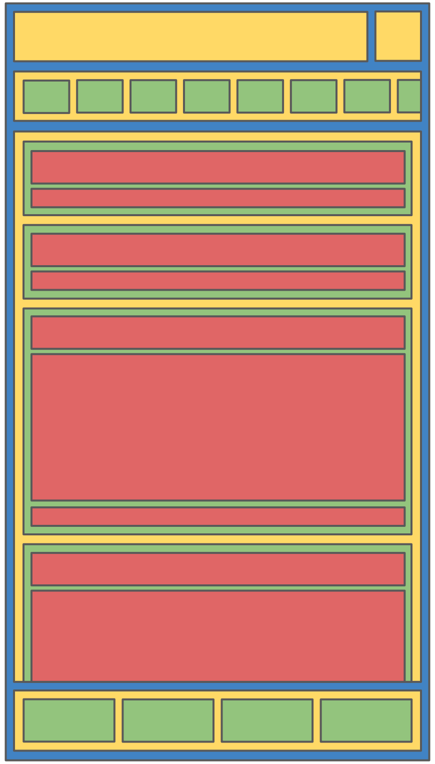
习近平：在"不忘初心、牢记使命"主题教育工作会议上的讲话

在新时代的长征路上砥砺前行

史无前例！特朗普金正恩板门店相见

6月嘉年华"惠"不停，"乐"起high起来！





🟦 第一层

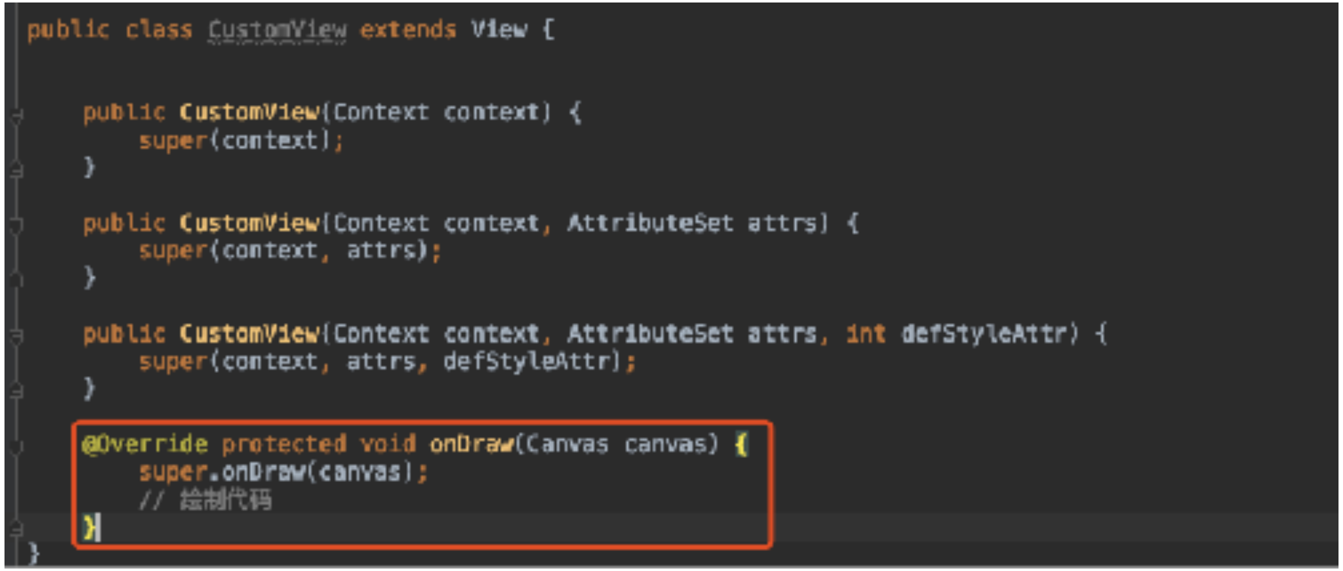
🟨 第二层

🟩 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

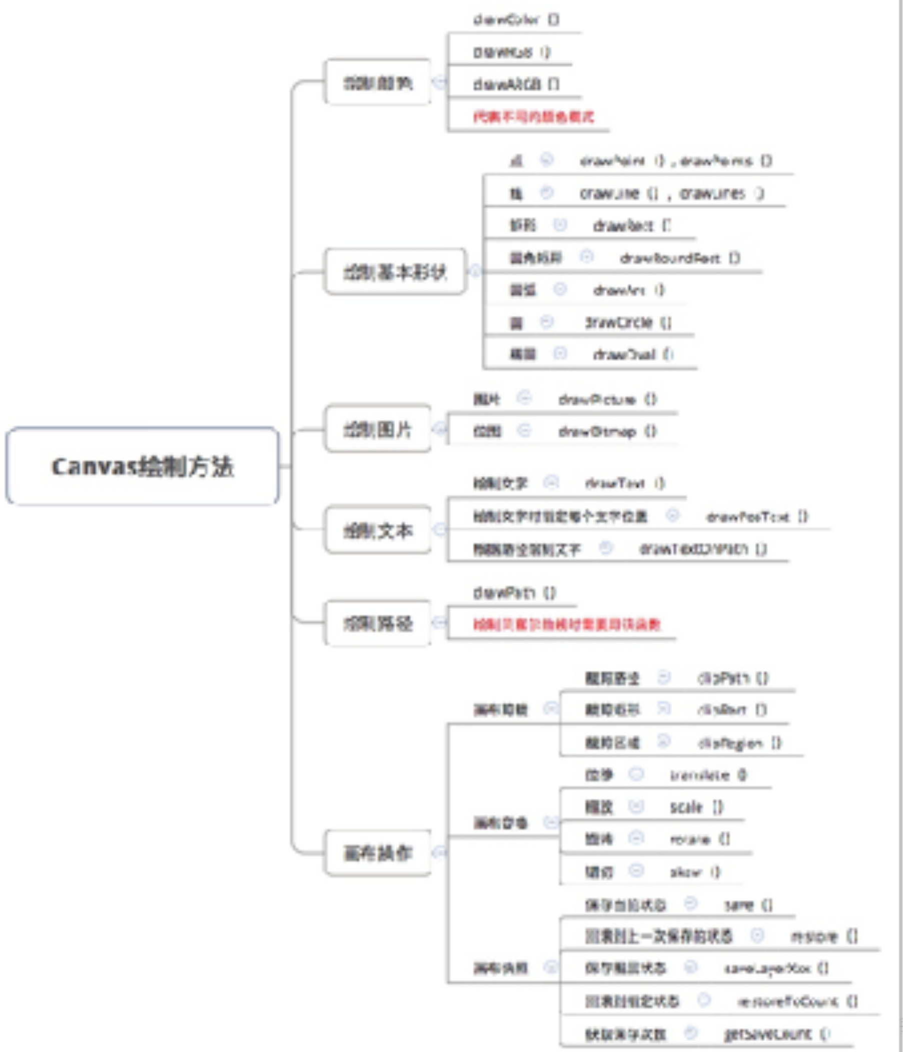


```
public class CustomView extends View {

    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
    }
}
```

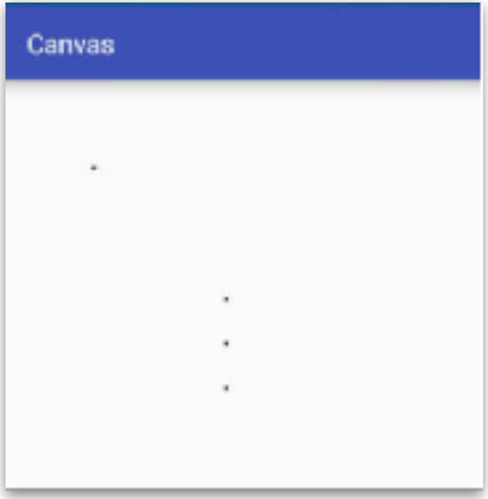
自定义View-重写onDraw

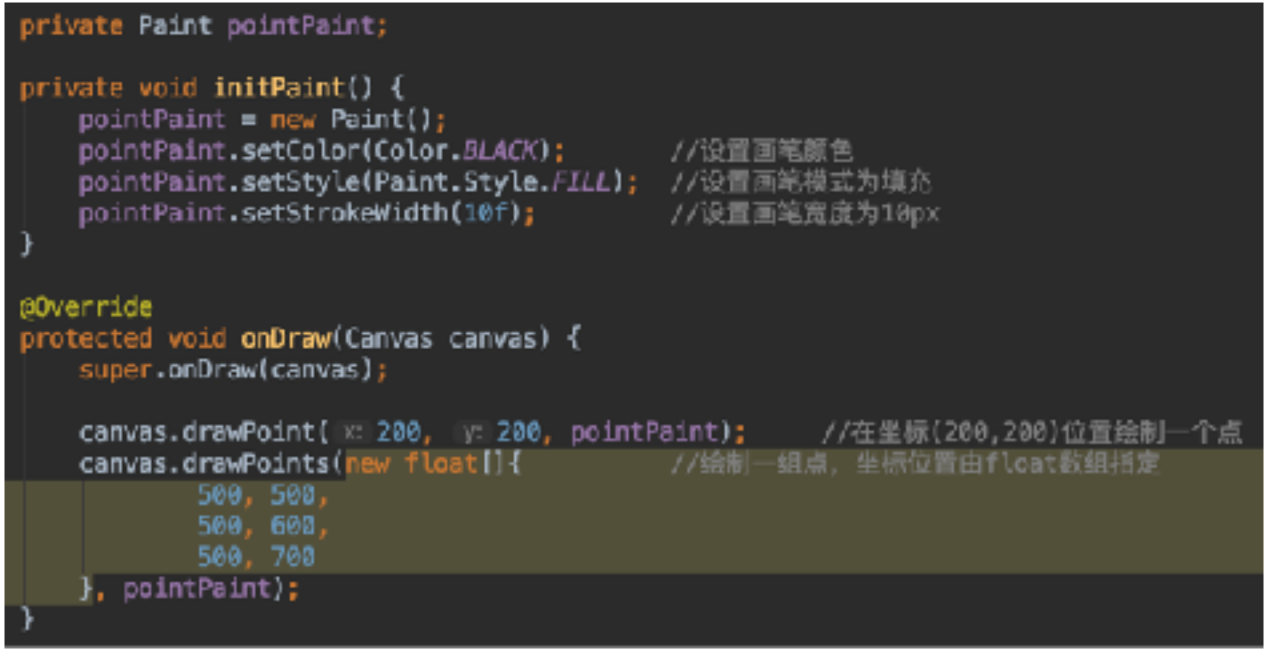
概念解析：

1. Canvas：画布
2. Paint：画笔



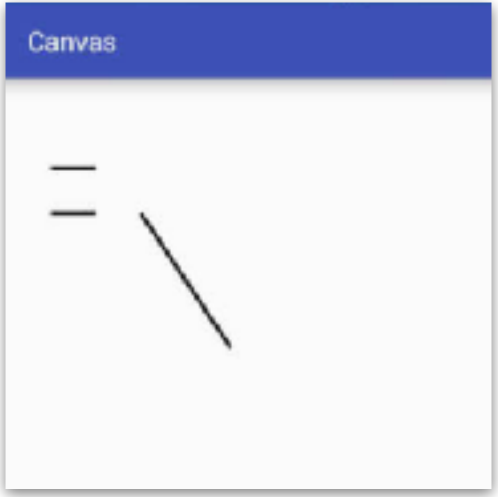
基本绘制-点 (Point)

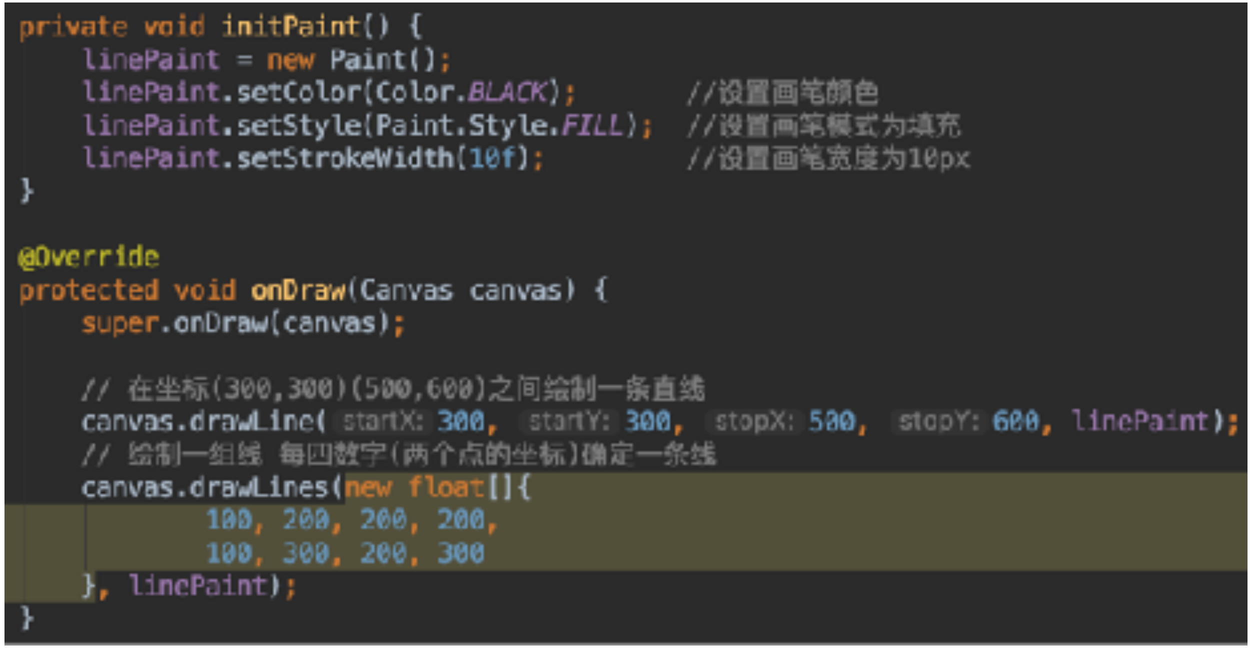




```
private Paint pointPaint;

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[]{               //绘制一组点，坐标位置由float数组指定
            500, 500,
            500, 600,
            500, 700
    }, pointPaint);
}
```

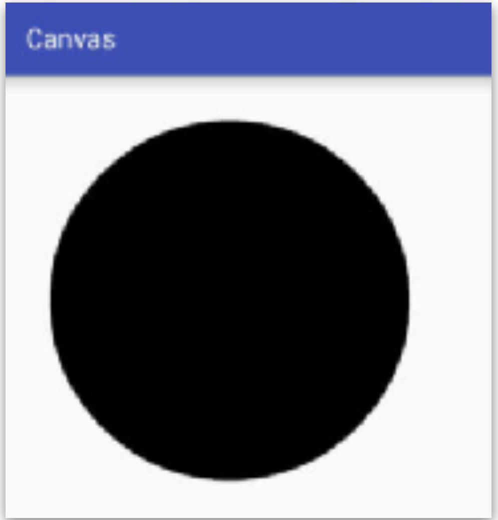
基本绘制-线 (Line)

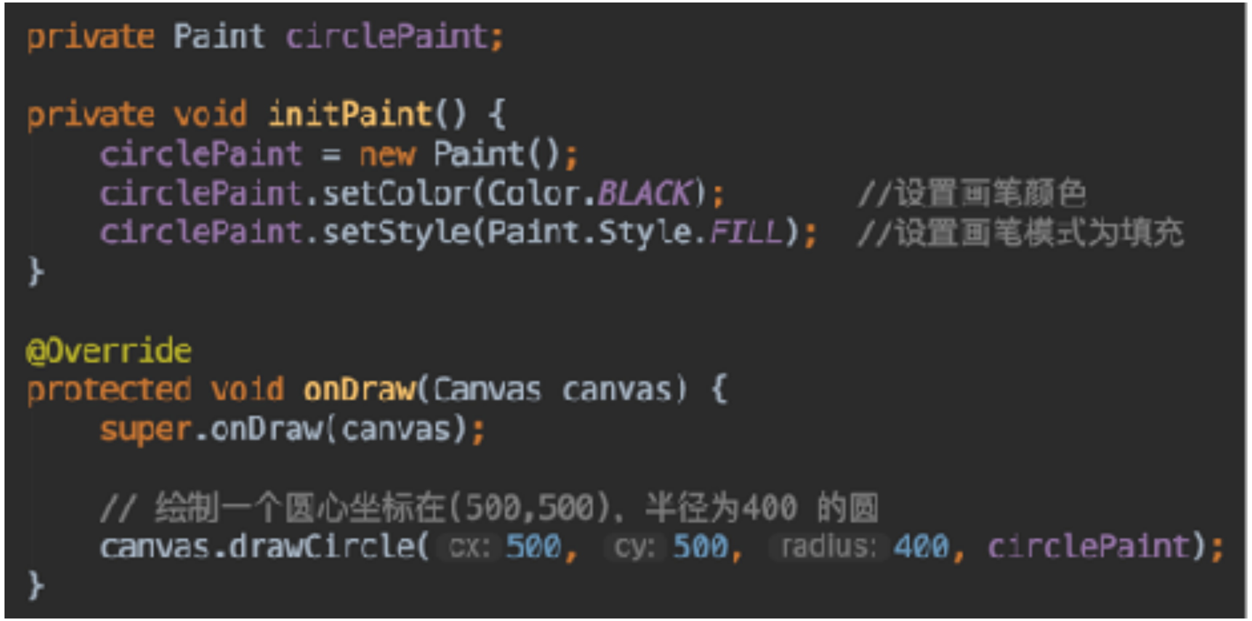




```java
private void initPaint() {
    linePaint = new Paint();
    linePaint.setColor(Color.BLACK);          //设置画笔颜色
    linePaint.setStyle(Paint.Style.FILL);     //设置画笔模式为填充
    linePaint.setStrokeWidth(10f);            //设置画笔宽度为10px
}

@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);
}
```

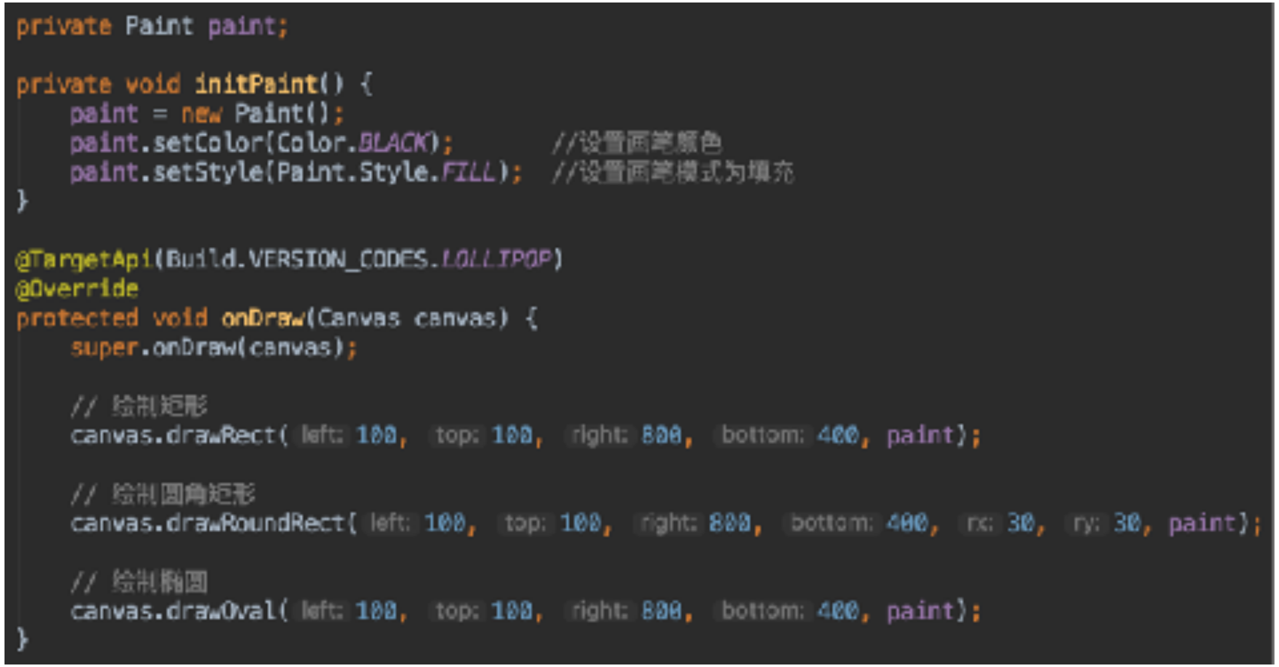
基本绘制-圆形 (Circle)





```
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);
}
```

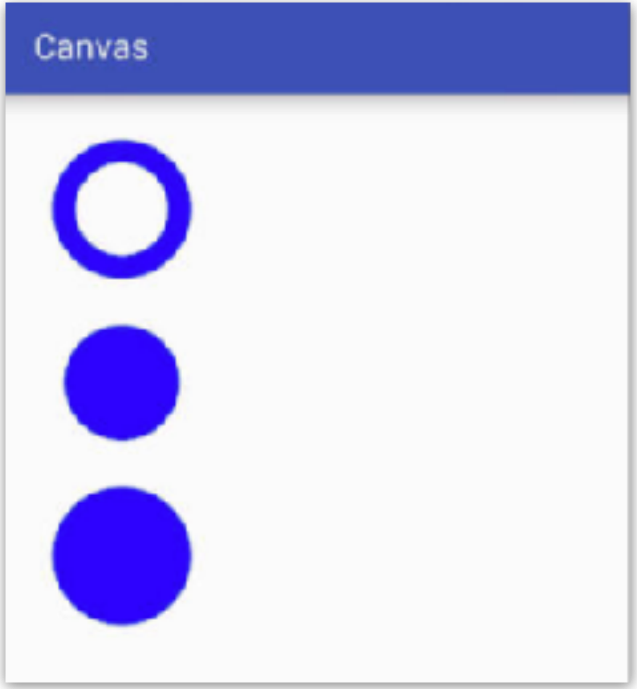
基本绘制-矩形/圆角矩形/椭圆 (Rect / RoundRect / Oval)

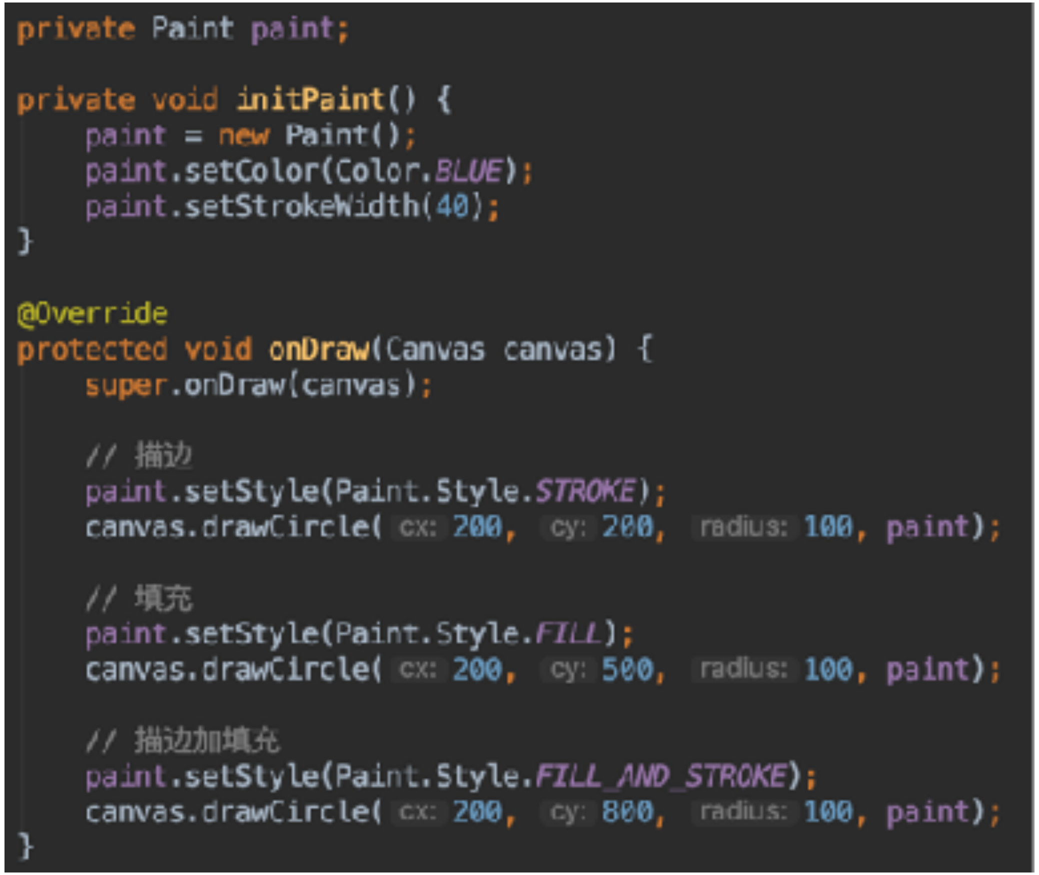


```
private Paint paint;

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

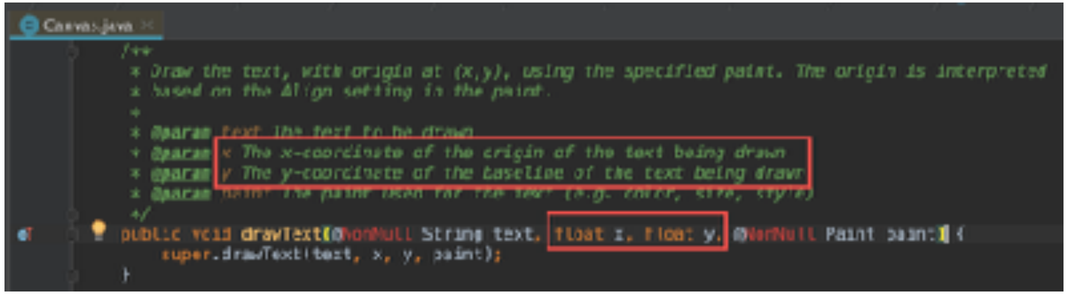
（代码举例）





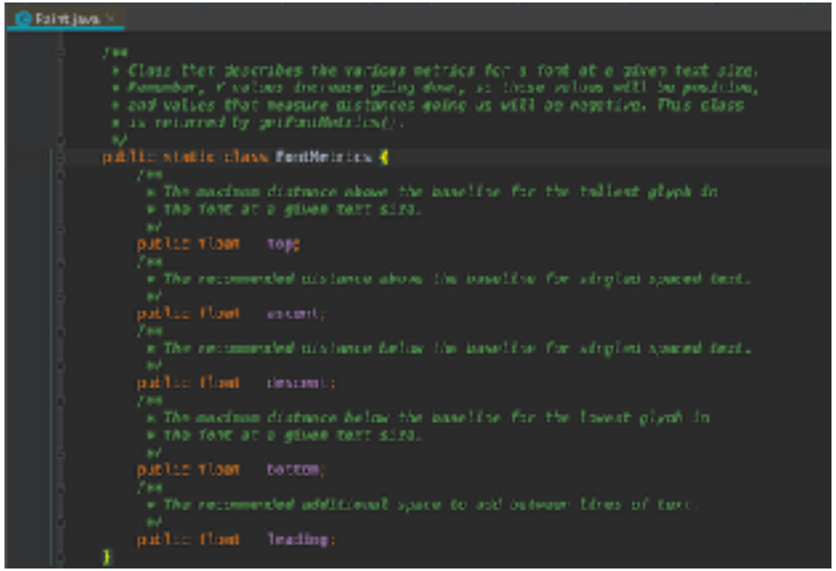
```
private Paint paint;

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



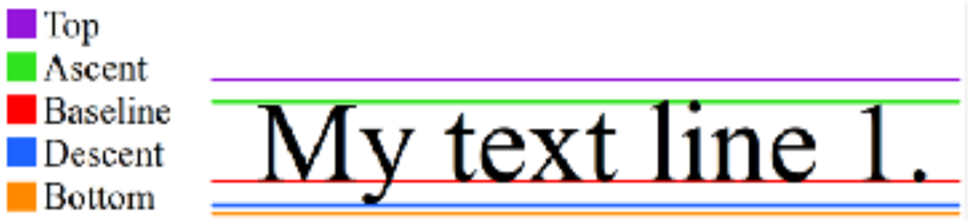
Canvas.java

```
/**
 * Draw the text, with origin at (x,y), using the specified paint. The origin is interpreted
 * based on the Align setting in the paint.
 *
 * @param text The text to be drawn
 * @param x The x-coordinate of the origin of the text being drawn
 * @param y The y-coordinate of the baseline of the text being drawn
 * @param paint The paint used for the text (e.g. color, size, style)
 */
public void drawText(@NonNull String text, float x, float y, @NonNull Paint paint) {
    super.drawText(text, x, y, paint);
}
```

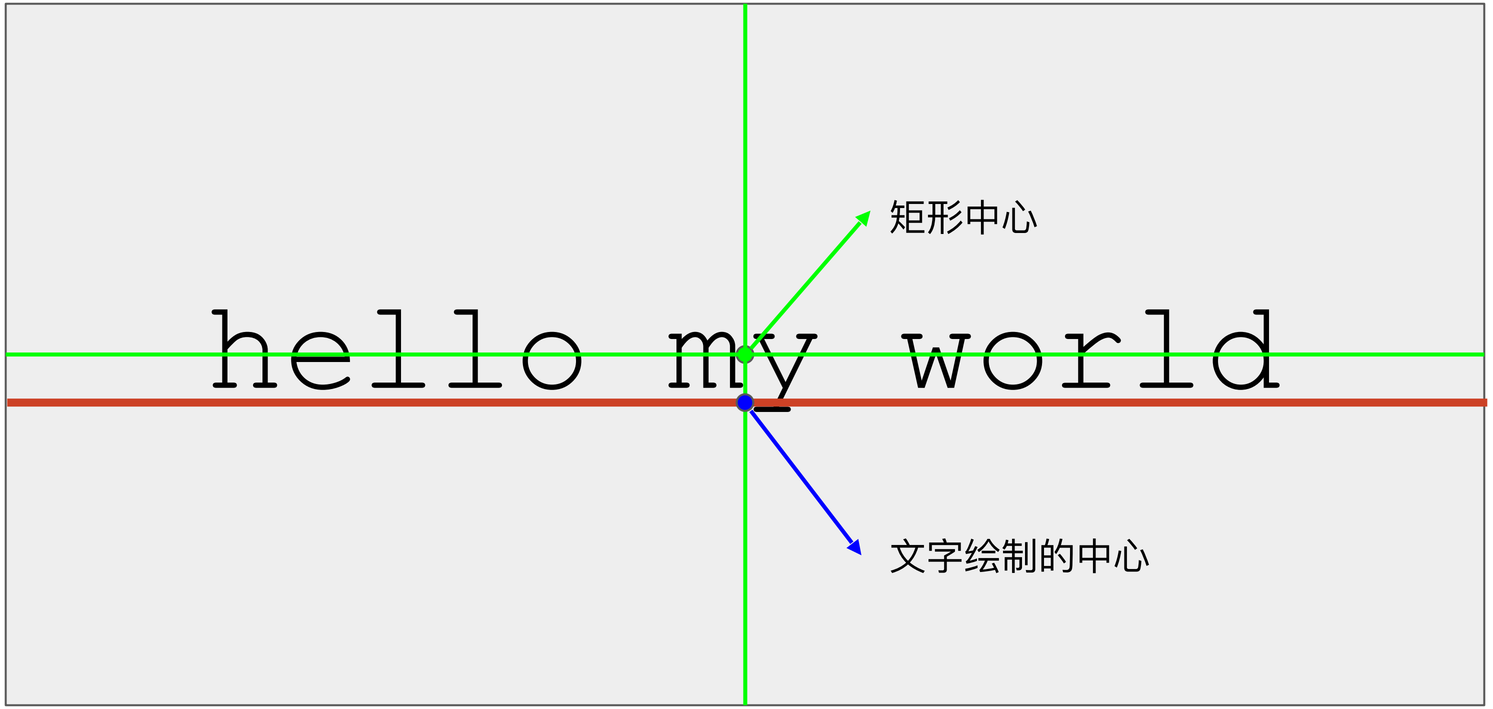


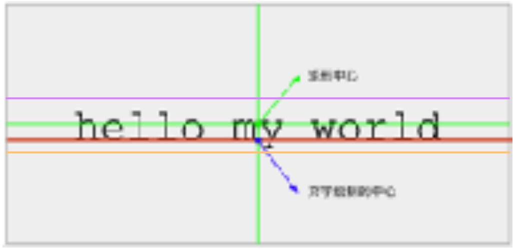
Paint.java

```
/**
 * Class that describes the various metrics for a font at a given text size.
 * Remember, Y values increase going down, so those values will be positive,
 * and values that measure distances going up will be negative. This class
 * is returned by getFontMetrics().
 */
public static class FontMetrics {
    /**
     * The maximum distance above the baseline for the tallest glyph in
     * the font at a given text size.
     */
    public float   top;
    /**
     * The recommended distance above the baseline for singled spaced text.
     */
    public float   ascent;
    /**
     * The recommended distance below the baseline for singled spaced text.
     */
    public float   descent;
    /**
     * The maximum distance below the baseline for the lowest glyph in
     * the font at a given text size.
     */
    public float   bottom;
    /**
     * The recommended additional space to add between lines of text.
     */
    public float   leading;
}
```



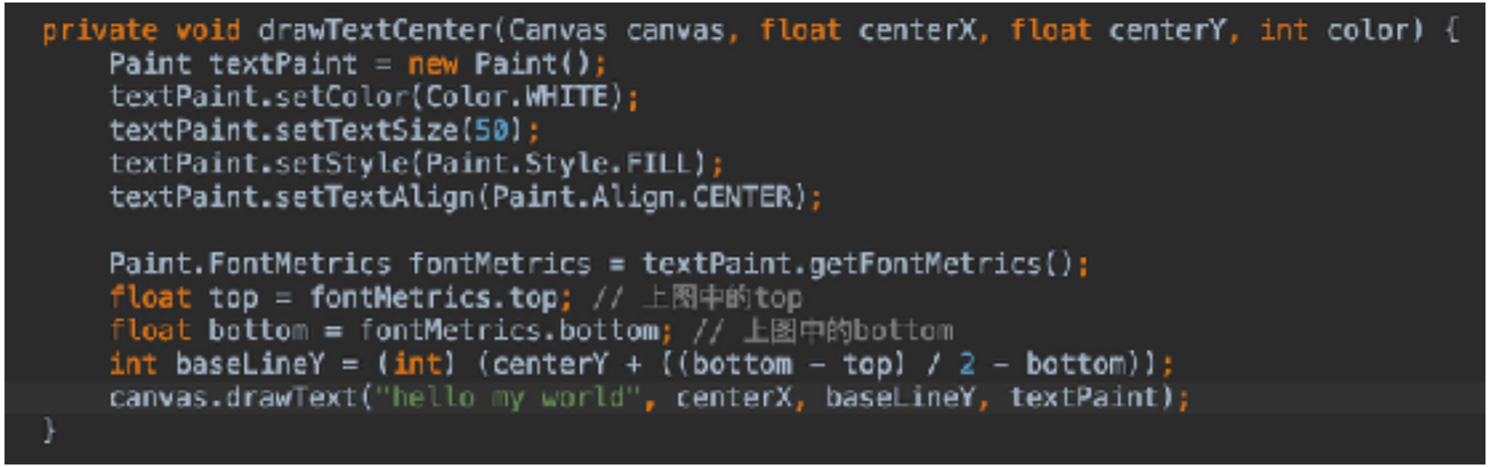
基本绘制-文字





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

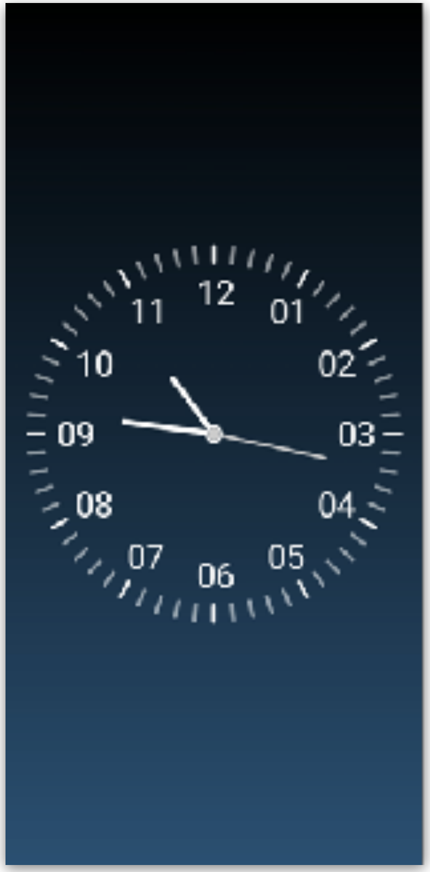
时钟App

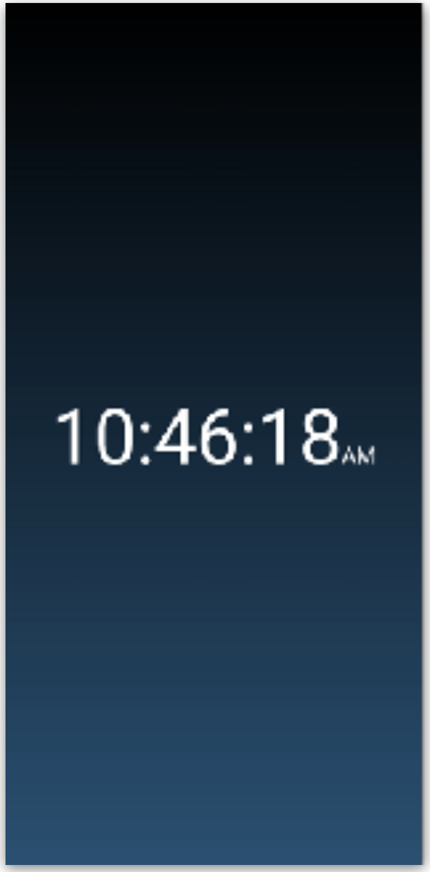
作业：

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？）



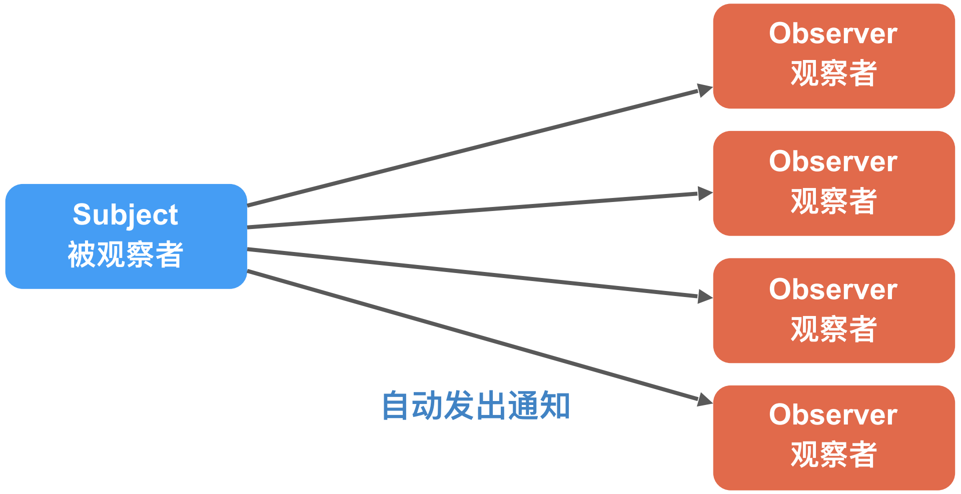


附：内存泄露 - Memory Leak

- 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