1. 如何对 Android 应用进行性能分析

android 性能主要之响应速度 和 UI 刷新速度。

首先从函数的耗时来说，有一个工具 TraceView 这是androidsdk 自带的工作，用于测量函数耗时的。

UI 布局的分析，可以有 2 块，一块就是 Hierarchy Viewer 可以看到 View 的布局层次，以及每个 View 刷新加载的时间。

这样可以很快定位到那块 layout & View 耗时最长。

还有就是通过自定义 View 来减少 view 的层次。

2. 什么情况下会导致内存泄露

内存泄露是个折腾的问题。

什么时候会发生内存泄露？内存泄露的根本原因：长生命周期的对象持有短生命周期的对象。短周期对象就无法及时释放。

I. 静态集合类引起内存泄露

主要是hashmap，Vector 等，如果是静态集合 这些集合没有及时 setnull 的话，就会一直持有这些对象。

II.remove 方法无法删除 set 集　Objects.hash(firstName, lastName);

经过测试，hashcode 修改后，就没有办法 remove 了。

III. observer 我们在使用监听器的时候，往往是 addxxxlistener，但是当我们不需要的时候，忘记 removexxxlistener，就容易内存 leak。

广播没有 unregisterrecevier

IV.各种数据链接没有关闭，数据库 contentprovider，io，sokect 等。cursor

V.内部类：

java 中的内部类（匿名内部类），会持有宿主类的强引用 this。

所以如果是 new Thread 这种，后台线程的操作，当线程没有执行结束时，activity 不会被回收。

Context 的引用，当 TextView 等等都会持有上下文的引用。如果有 static drawable，就会导致该内存无法释放。

VI.单例

单例 是一个全局的静态对象，当持有某个复制的类 A 是，A 无法被释放，内存 leak。

3. 如何避免 OOM 异常

首先 OOM 是什么?

当程序需要申请一段"大"内存，但是虚拟机没有办法及时的给到，即使做了 GC 操作以后

这就会抛出 OutOfMemoryException 也就是 OOM

Android 的 OOM 怎么样?

为了减少单个 APP 对整个系统的影响，android 为每个 app 设置了一个内存上限。

```
    public void getMemoryLimited(Activity context)
    {
        ActivityManager activityManager
=(ActivityManager)context.getSystemService(Context.ACTIVITY_SERVIC E);
        System.out.println(activityManager.getMemoryClass());
        System.out.println(activityManager.getLargeMemoryClass());
        System.out.println(Runtime.getRuntime().maxMemory()/(1024*1024));
    }


09-10 10:20:00.477 4153-4153/com.joyfulmath.samples I/System.out: 192
09-10 10:20:00.477 4153-4153/com.joyfulmath.samples I/System.out: 512
09-10 10:20:00.477 4153-4153/com.joyfulmath.samples I/System.out: 192
```

HTC M7 实测，192M 上限。512M 一般情况下，192M 就是上限，但是由于某些特殊情况，android 允许使用一个更大的 RAM。

如何避免 OOM

减少内存对象的占用

I.ArrayMap/SparseArray 代替 hashmap

II.避免在 android 里面使用 Enum

III. 减少bitmap 的内存占用

- inSampleSize：缩放比例，在把图片载入内存之前，我们需要先计算出一个合适的缩放比例，避免不必要的大图载入。
- decode format：解码格式，选择 ARGB_8888/RBG_565/ARGB_4444/ALPHA_8，存在很大差异。

IV. 减少资源图片的大小，过大的图片可以考虑分段加载

内存对象的重复利用

大多数对象的复用，都是利用对象池的技术。

I.listview/gridview/recycleview contentview 的复用

II.inBitmap 属性对于内存对象的复用
ARGB_8888/RBG_565/ARGB_4444/ALPHA_8

这个方法在某些条件下非常有用，比如要加载上千张图片的时候。

III.避免在ondraw 方法里面 new 对象

IV.StringBuilder 代替+

4. Android 中如何捕获未捕获的异常

CrashHandler

关键是实现 Thread.UncaughtExceptionHandler

然后是在 application 的 oncreate 里面注册。

5. ANR 是什么？怎样避免和解决 ANR（重要）

ANR->Application Not Responding

也就是在规定的时间内，没有响应。

三种类型：

1）. KeyDispatchTimeout(5 seconds) --主要类型按键或触摸事件在特定时间内无响应

2）. BroadcastTimeout(10 seconds) --BroadcastReceiver 在特定时间内无法处理完成

3）. ServiceTimeout(20 seconds) --小概率类型 Service 在特定的时间内无法处理完成

为什么会超时：事件没有机会处理 & 事件处理超时

怎么避免 ANR

ANR 的关键

是处理超时，所以应该避免在 UI 线程，BroadcastReceiver 还有service 主线程中，处理复杂的逻辑和计算

而交给 work thread 操作。

1）避免在 activity 里面做耗时操作，oncreate & onresume

2）避免在 onReceiver 里面做过多操作

3） 避免在Intent Receiver 里启动一个 Activity，因为它会创建一个新的画面，并从当前用户正在运行的程序上抢夺焦点。

4） 尽量使用 handler 来处理 UI thread & workthread 的交互。

如何解决 ANR

首先定位 ANR 发生的 log：

```
04-01 13:12:11.572 I/InputDispatcher( 220): Application is not
responding:Window{2b263310com.android.email/com.android.email.acti
vity.SplitScreenActivitypaused=false}.....5009.8ms sinceevent,
5009.5ms since waitstarted
CPUusage from 4361ms to 699ms ago---------- CPU在ANR 发生前的使用情况
04-0113:12:15.872 E/ActivityManager( 220): 100%TOTAL: 4.8% user +
7.6% kernel + 87% iowait

04-0113:12:15.872 E/ActivityManager( 220): CPUusage from 3697ms to 4223ms
later:-- ANR 后 CPU 的使用量
```

从log 可以看出，cpu 在做大量的 io 操作。

所以可以查看 io 操作的地方。

当然，也有可能 cpu 占用不高，那就是 主线程被 block 住了。

6. Android 线程间通信有哪几种方式

1）共享变量（内存）

2）管道3）

handle 机制

runOnUiThread(Runnable)

view.post(Runnable)

7. Devik 进程，linux 进程，线程的区别

Dalvik 进程。

每一个android app 都会独立占用一个 dvm 虚拟机，运行在 linux 系统中。

所以dalvik 进程和 linux 进程是可以理解为一个概念。

8. 描述一下 android 的系统架构

从小到上就是：

linux kernel,lib dalvik vm ,application framework, app

9. android 应用对内存是如何限制的?我们应该如何合理使用内存？

activitymanager.getMemoryClass() 获取内存限制。

关于合理使用内存，其实就是避免 OOM & 内存泄露中已经说明。

10. 简述 android 应用程序结构是哪些

1）main code

2) unit test

3）mianifest

4）res->drawable,drawable-xxhdpi,layout,value,mipmap

mipmap 是一种很早就有的技术了，翻译过来就是**纹理映射技术.**

google 建议只把启动图片放入。

5）lib

6）color

11. 请解释下 Android 程序运行时权限与文件系统权限的区别

文件的系统权限是由 linux 系统规定的，只读，读写等。

运行时权限，是对于某个系统上的 app 的访问权限，允许，拒绝，询问。该功能可以防止非法的程序访问敏感的信息。

12. Framework 工作方式及原理，Activity 是如何生成一个 view 的，机制是什么

Framework 是 android 系统对 linux kernel，lib 库等封装，提供 WMS，AMS，bind 机制，handler-message 机制等方式，供 app 使用。

简单来说 framework 就是提供 app 生存的环境。

1）Activity 在attch 方法的时候，会创建一个 phonewindow（window 的子类）

2）onCreate 中的 setContentView 方法，会创建 DecorView

3）DecorView 的addview 方法，会把 layout 中的布局加载进来。

13. 多线程间通信和多进程之间通信有什么不同，分别怎么实现

线程间的通信可以参考第 6 点。

进程间的通信：bind 机制（IPC->AIDL），linux 级共享内存，boradcast，

Activity 之间，activity & serview 之间的通信，无论他们是否在一个进程内。

14. Android 屏幕适配

屏幕适配的方式：xxxdpi， wrap_content,match_parent. 获取屏幕大小，做处理。

dp 来适配屏幕，sp 来确定字体大小

drawable-xxdpi, values-1280*1920 等 这些就是资源的适配。

wrap_content,match_parent，这些是 view 的自适应weight，

这是权重的适配。

15. 什么是 AIDL 以及如何使用

Android Interface Definition Language

AIDL 是使用 bind 机制来工作。

参数：

java 原生参数

String

parcelable

list & map 元素 需要支持 AIDL

16. android 进程内的消息驱动机制---Handler,MessageQueue,Runnable,Looper

1.Runnable & MessageQueue:

Runnable 和Message 是消息的 2 种载体。

消息的行为本质上就是一段操作 Runnable，或者是一段数据 Message，包含这操作内容，由 handlemessage 来判断处理。

他们的操作方式就是：

```
public final boolean post(Runnable r)
{
    return sendMessageDelayed(getPostMessage(r), 0);
}

public final boolean postAtTime(Runnable r, long uptimeMillis)

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

上面就是 Runnable 的方法，可以看到 Runnable 会被分装成 Message 的形式发送。

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

所以本质上，都是以 Message 的封装方式处理。

最终所有的消息都会放入 MessageQueue 里面。

MessageQueue 并不是一个真正的队列，而是链表。

Looper 就是循环在某件事情，类似于 while（true）干的事情。

Handler 就是真正做事情的。

Looper 不断的从 MessageQueue 从取出数据，然后交给 handler 来处理。

2.Handler:

framework/base/core/android/os/Handler.java

其实 handler 的作用，它的注释已经解释的非常清楚。


```
/**
 * A Handler allows you to send and process {@link Message} and Runnable
 * objects associated with a thread's {@link MessageQueue}.  Each Handler
 * instance is associated with a single thread and that thread's message
 * queue.  When you create a new Handler, it is bound to the thread /
 * message queue of the thread that is creating it -- from that point on,
 * it will deliver messages and runnables to that message queue and execute
 * them as they come out of the message queue.
*
 * <p>There are two main uses for a Handler: (1) to schedule messages and
 * runnables to be executed as some point in the future; and (2) to enqueue
 * an action to be performed on a different thread than your own.
*
 * <p>When posting or sending to a Handler, you can either
 * allow the item to be processed as soon as the message queue is ready
 * to do so, or specify a delay before it gets processed or absolute time for
 * it to be processed.  The latter two allow you to implement timeouts,
 * ticks, and other timing-based behavior.
*/
```

这个一共三段内容，大意是：

1）handler 使用 runnable 或者 message 的方式传递，存储在一个 thread 的 messagequeue 里面。

当你创建一个新的 handler 的时候，他会与这个创建它的线程绑定。

对于一个Thread 来说MessageQueue,和Looper 只有一个。

2）使用handler 一般有2 种场景。

希望do runnable 或者某种Message 在in the future.

或者把一个 action（Runnable or Message）传递到其他线程进行操作。

常见的操作就是在工作线程中使用主线程 handler 来操作 UI。

3）你可以让handler 直接操作message 内容，或者等待一段时间，这个时间是可以配置的。

handle 的2 大功能

处理message：

```
public void dispatchMessage(Message msg) 分发消息
public void handleMessage(Message msg)　处理消息，该方法通常情况下，须由子类继承。
```

Looper.loop()方法会调用 dispatchMessage 来处理消息。

```
public void dispatchMessage(Message msg)
    { if (msg.callback != null) {
        handleCallback(msg);
    } else {
        if (mCallback != null) {
            if (mCallback.handleMessage(msg))
                { return;
            }
        }
        handleMessage(msg);
    }
}
```

handler 的子类通过重载该方法，可以修改 handler 的消息派发方式。

handler 的第二个作用是把 message & Runnable 分装到MessageQueue 里面。

handler，messagequeue，looper 目的是什么，目的就是启动消息机制。

MessageQueue:

MessageQueue 从哪里得到，从 Handler 源码看到，是从 Looper 里面来的。

```
public Handler(Looper looper, Callback callback, boolean async)
    { mLooper = looper;
    mQueue = looper.mQueue;
    mCallback = callback;
    mAsynchronous = async;
}
```

Looper：

```
    private Looper(boolean quitAllowed) {
        mQueue = new MessageQueue(quitAllowed);
        mThread = Thread.currentThread();
    }
```

Looper 构造函数就干了 2 件事。

创建Messagequeue，所以每个 Looper 都有唯一的一个 MessageQueue 与之对应。得到运行thread。

```
    // sThreadLocal.get() will return null unless you've called prepare().
    static final ThreadLocal<Looper> sThreadLocal = new ThreadLocal<Looper>();
```

Looper 有个特殊的变量，ThreadLocal，这个对象只对自己所在的线程全局，其他的线程无法看到它。

Looper 提供了很多 static 的方法，所以肯定还有一些能都识别"身份"的方法。

这些方法在我们使用 looper 的时候，最重要的是如下 2 个：

```
    private static void prepare(boolean quitAllowed)
        { if (sThreadLocal.get() != null) {
            throw new RuntimeException("Only one Looper may be created per
thread");
        }
        sThreadLocal.set(new Looper(quitAllowed));
    }
/**
     * Run the message queue in this thread. Be sure to call
     * {@link #quit()} to end the loop.
     */
    public static void loop() {
        final Looper me = myLooper();
        if (me == null) {
            throw new RuntimeException("No Looper; Looper.prepare() wasn't called
on this thread.");
        }
        final MessageQueue queue = me.mQueue;

        // Make sure the identity of this thread is that of the local process,
        // and keep track of what that identity token actually is.
        Binder.clearCallingIdentity();
        final long ident = Binder.clearCallingIdentity();

        for (;;) {
            Message msg = queue.next(); // might block
```

```
                if (msg == null) {
                    // No message indicates that the message queue is quitting.
                    return;
                }

                // This must be in a local variable, in case a UI event sets the logger
                Printer logging = me.mLogging;
                if (logging != null) {
                    logging.println(">>>>> Dispatching to " + msg.target + " " +
                            msg.callback + ": " + msg.what);
                }

                msg.target.dispatchMessage(msg);

                if (logging != null) {
                    logging.println("<<<<< Finished to " + msg.target + " " +
msg.callback);
                }

                // Make sure that during the course of dispatching the
                // identity of the thread wasn't corrupted.
                final long newIdent = Binder.clearCallingIdentity();
                if (ident != newIdent) {
                    Log.wtf(TAG, "Thread identity changed from 0x"
                            + Long.toHexString(ident) + " to 0x"
                            + Long.toHexString(newIdent) + " while dispatching to "
                            + msg.target.getClass().getName() + " "
                            + msg.callback + " what=" + msg.what);
                }

                msg.recycle();
            }
        }
```

prepare 才是looper 创建以及和thread 绑定的地方。

looper.loop()方法是整个 looper 机制启动的地方。从

此thread 就会接受消息和处理消息了。

这里有个小问题：

```
                Message msg = queue.next(); // might block
                if (msg == null) {
                    // No message indicates that the message queue is quitting.
                    return;
                }
```

一开始的时候，MessageQueue handler 没有传递消息进队列，按理说取到的消息是 null，这样 looper 就直接退出了。

这个问题等到分析源码的时候，在解决。

这样handler，messaqequeue,looper，和thread 都关联起来了。

下面还有一个 mainlooper 的问题。

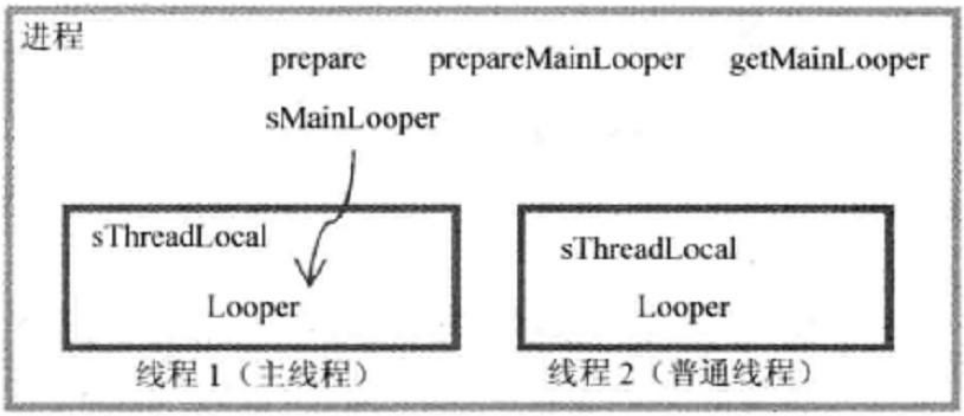
```
public static void main(String[] args) {
    ...

    Looper.prepareMainLooper();

    if (sMainThreadHandler == null)
            { sMainThreadHandler =
            thread.getHandler();
        }

    Looper.loop();
```

以上是ActivityThread 的部分入口函数 main 的源码： 可见prepareMainLooper()的方法，是给主线程使用的。而 looper 那边的

```
private static Looper sMainLooper;  // guarded by Looper.class
```

是为了给其他线程应用使用。

这样其他线程可以给主线程发消息。



如图所示：主线程的 looper 将由sMainLooper 作为应用，保存在 static 空间中，其他工作线程可以访问它

至此，整个消息机制的框架已经驱动起来。

17. 事件分发机制

1. View的事件分发机制

一个button，简单一点就是onTouch，还有onclick 事件，我们一个一个来分析

首先响应的是dispatchTouchEvent

```
public boolean dispatchTouchEvent(MotionEvent event) {
    if (mOnTouchListener != null && (mViewFlags & ENABLED_MASK) ==
ENABLED &&
            mOnTouchListener.onTouch(this, event))
        { return true;
    }
    return onTouchEvent(event);
}
```

其实，在 android 源码的命名还是很有规律的，dispatchXXX，也就是分发机制，往往就是第一个需要响应的地方。

我们来分析下：touchlistener 不为空，也就是 view 的使用者设置了回调。

第二个条件就是 View 必须是 enable 的。第三：onTouch 返回 false，就说明 onTouch 不消费该事件，由 OnTouchEvent 响应。

如果返回True，那么就会直接 return。所

以onClick 事件一定会被调到。

```
⊞⊟
onTouchEvent
```

最终会走到 performClick 这个方法。

```
public boolean performClick()
    { final boolean result;
    final ListenerInfo li = mListenerInfo;
    if (li != null && li.mOnClickListener != null)
        { playSoundEffect(SoundEffectConstants.CLIC
        K); li.mOnClickListener.onClick(this);
        result = true;
    } else {
        result = false;
    }

    sendAccessibilityEvent(AccessibilityEvent.TYPE_VIEW_CLICKED);
    return result;
}
```

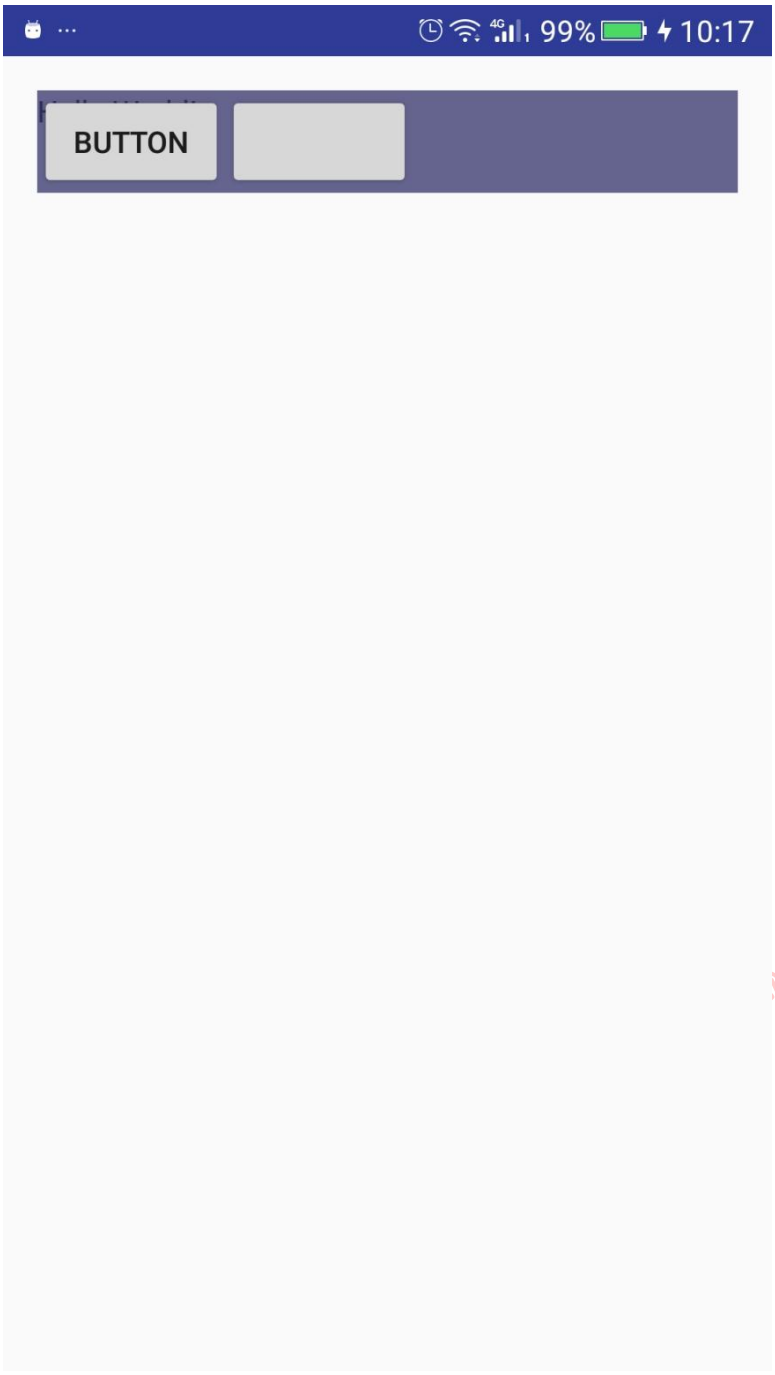
可以看到，如果 setOnClickListener, onClick 就会走到。

2. ViewGroup 的事件分发机制

```
<com.joyfulmath.frameworksample.viewdemo.MyLayout
        android:id="@+id/my_layout"
        android:background="#99000044"
        android:layout_width="match_parent"
        android:layout_height="wrap_content">
        <Button
            android:id="@+id/button_id"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:text="button"/>
        <Button
            android:id="@+id/imageId"
            android:layout_centerInParent="true"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:src="@android:drawable/ic_lock_power_off"/>
    </com.joyfulmath.frameworksample.viewdemo.MyLayout>
```

一个 layout 里面有 2 个 button,

```
⊞⊟
TestViewAction
```



分别点击 button1 & button2 & 灰色部分

等到 log 如下：

```
08-27 10:19:26.799 2120-2120/com.joyfulmath.frameworksample I/TestViewAction: onTouch: button_id [at (TestViewAction.java:55)]
08-27 10:19:26.880 2120-2120/com.joyfulmath.frameworksample I/TestViewAction: onTouch: button_id [at (TestViewAction.java:55)]
08-27 10:19:26.896 2120-2120/com.joyfulmath.frameworksample I/TestViewAction: onTouch: button_id [at (TestViewAction.java:55)]
08-27 10:19:26.913 2120-2120/com.joyfulmath.frameworksample I/TestViewAction: onTouch: button_id [at (TestViewAction.java:55)]
```

```
08-27 10:19:26.926 2120-2120/com.joyfulmath.frameworksample I/TestViewAction:
onTouch: button_id [at (TestViewAction.java:55)]
08-27 10:19:26.926 2120-2120/com.joyfulmath.frameworksample I/TestViewAction:
onClick: button_id [at (TestViewAction.java:38)]
08-27 10:19:27.434 2120-2120/com.joyfulmath.frameworksample I/TestViewAction:
onTouch: imageId [at (TestViewAction.java:58)]
08-27 10:19:27.535 2120-2120/com.joyfulmath.frameworksample I/TestViewAction:
onTouch: imageId [at (TestViewAction.java:58)]
08-27 10:19:27.543 2120-2120/com.joyfulmath.frameworksample I/TestViewAction:
onTouch: imageId [at (TestViewAction.java:58)]
08-27 10:19:27.544 2120-2120/com.joyfulmath.frameworksample I/TestViewAction:
onClick: imageId [at (TestViewAction.java:41)]
08-27 10:19:28.111 2120-2120/com.joyfulmath.frameworksample I/TestViewAction:
onTouch: my_layout [at (TestViewAction.java:61)]
08-27 10:19:28.156 2120-2120/com.joyfulmath.frameworksample I/TestViewAction:
onTouch: my_layout [at (TestViewAction.java:61)]
08-27 10:19:28.173 2120-2120/com.joyfulmath.frameworksample I/TestViewAction:
onTouch: my_layout [at (TestViewAction.java:61)]
08-27 10:19:28.190 2120-2120/com.joyfulmath.frameworksample I/TestViewAction:
onTouch: my_layout [at (TestViewAction.java:61)]
08-27 10:19:28.237 2120-2120/com.joyfulmath.frameworksample I/TestViewAction:
onTouch: my_layout [at (TestViewAction.java:61)]
08-27 10:19:28.237 2120-2120/com.joyfulmath.frameworksample I/TestViewAction:
onClick: my_layout [at (TestViewAction.java:44)]
```

也就是点击 button1 以后，不会传递都 layout

But，如果 layout 里面有一个函数

```
public boolean onInterceptTouchEvent(MotionEvent ev)
```

这个函数就是截断对 button 的分发处理，默认是 return false。

至此，我们有了一个大概的流程。

Activtiy->ViewGroup->View

如果仔细分析就会发现，在 Activity 里面有一个 getDocView。所以 Activity 里面有个 RootView 的概念。

言归正传，ViewGroup 本质上也是一个 View，所以，可以把模型简单的定性为 Activtiy->ViewGroup->View 三层。

首先 Activity 里面有 2 个函数，我们分析看看：

```
@Override
public boolean dispatchTouchEvent(MotionEvent ev)
    { TraceLog.i();
    return super.dispatchTouchEvent(ev);
}
```

```
@Override
public boolean onTouchEvent(MotionEvent event)
    { TraceLog.i();
    return super.onTouchEvent(event);
}
```

所以大体流程如下：

1. @Activty.diapatchTouchEvent ->@Layout.dispatchTouchEvent->@layout.onInterceptTouchEvent return true/false

2. return true->@layout.onTouchEvent 后面部分同 view

3. return false->@view.dispatchTouchEvent View 的分发见上一片流程。

18. 子线程发消息到主线程进行更新 UI，除了 handler 和 AsyncTask，还有什么

EventBus，广播，view.post, runinUiThread

但是无论各种花样，本质上就 2 种：handler 机制 + 广播

19. 子线程中能不能 new handler？为什么

必须可以。子线程 可以 new 一个 mainHandler，然后发送消息到 UI Thread。

20. Android 中的动画有哪几类，它们的特点和区别是什么

视图动画，或者说补间动画。只是视觉上的一个效果，实际 view 属性没有变化，性能好，但是支持方式少。

属性动画，通过变化属性来达到动画的效果，性能略差，支持点击等事件。android 3.0

帧动画，通过 drawable 一帧帧画出来。

Gif 动画，原理同上，canvas 画出来。

具体可参考：https://i.cnblogs.com/posts?categoryid=672052

21. 如何修改 Activity 进入和退出动画

overridePendingTransition

22. SurfaceView & View 的区别

view 的更新必须在 UI thread 中进行

surfaceview 会单独有一个线程做 ui 的更新。

surfaceview 支持open GL 绘制。