Android 0x06

> 这节课主要介绍Android的消息机制和事件分发

Android消息机制

> 消息机制各位应该都不陌生，早在上学期学习网络请求的时候应该都简单的接触过Handler和Message，使用这两个工具实现了跨线程通信，这就是一个简单的消息机制的使用。其实消息机制的使用远不止此，下面就一起进一步了解一下Android消息机制

消息机制简述

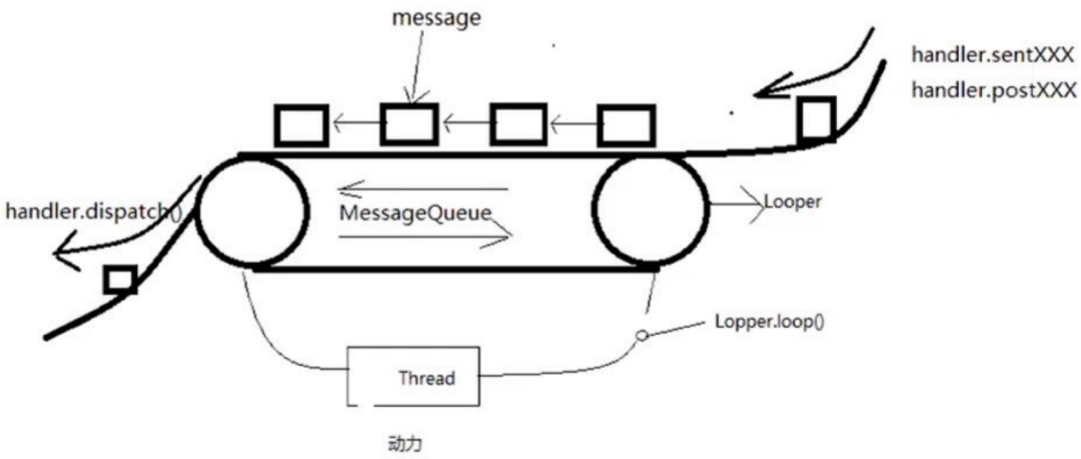
在开始之前还是最好先对消息机制有一个简单的了解，在我的理解中，消息机制就是Message的收发机制，Handler发送Message，并最终回调接收到自己发送的Message，这个功能的实现过程就是消息机制。

先看看典中典《Android开发艺术探索》是怎么描述消息机制的

> Android的消息机制主要是指**Handler**的运行机制，Handler的运行需要底层的**MessageQueue**和**Looper**的支持

这段话简单的说明了消息机制中的御三家，Handler，MessageQueue，Looper，这三者共同支撑起了Android的消息机制。

下面是一张非常经典的消息机制各个部分的合作关系



Handler，MessageQueue，Looper的合作关系

让我们从发送一个Message开始

Handler

Handler自然是分析发送Message的起始点，我们都知道，message是通过Handler的`sendMessage(Message msg)`方法发送出去的，所以不妨将`sendMessage(Message msg)`作为探索的入口，从这里开始一探消息机制

sendMessage(Message msg)

啥都不说了，上源码

```
public final boolean sendMessage(Message msg)
{
    return sendMessageDelayed(msg, 0);
}
```

调用了`sendMessageDelayed`方法，即延时发送策略，由于延时时间设置为0，可以理解为要求立刻发送

继续往下走两步，将会来到`sendMessageAtTime`方法，可以理解为在`uptimeMillis`时刻发送message

```
public boolean sendMessageAtTime(Message msg, long uptimeMillis) {
    MessageQueue queue = mQueue;
    if (queue == null) {
        RuntimeException e = new RuntimeException(
            this + " sendMessageAtTime() called with no mQueue");
        Log.w("Looper", e.getMessage(), e);
        return false;
    }
    return enqueueMessage(queue, msg, uptimeMillis);
}
```

划重点，这里MessageQueue开始正式登场，即消息队列，首先Handler会试图获取mQueue，之后利用这个MessageQueue实现Message的入队。

在继续深究消息如何入队之前，先来看看这个MessageQueue是如何获取的。

MessageQueue的获取最终会定位到下面这行代码

```
public Handler(Callback callback, boolean async) {
    .......无所谓の代码........
    mLooper = Looper.myLooper();
    if (mLooper == null) {
        throw new RuntimeException(
            "Can't create handler inside thread " + Thread.currentThread()
            + " that has not called Looper.prepare()");
    }
    mQueue = mLooper.mQueue;
    .......无所谓の代码........
}
```

可见，MessageQueue来自Looper，这里先简单提一嘴，到Looper的时候会详细来看这里MessageQueue的出处

下面回到主题，来看看message是怎样enqueue的

```
private boolean enqueueMessage(MessageQueue queue, Message msg, long
uptimeMillis) {
    msg.target = this;
    if (mAsynchronous) {
        msg.setAsynchronous(true);
    }
    return queue.enqueueMessage(msg, uptimeMillis);
}
```

这里首先会给message设置target属性，这个属性可以理解为对发送message的Handler的存储，将来在message出队的时候，就可以通过这个target属性获得究竟是哪一个Handler发送的此message，也就可以回调到对应Handler的`handleMessage()`方法，当然这都是后话了。

在设置完target之后，设置`isAsynchronous`的过程将会涉及到一些Handler**同步屏障**的知识，这里先放一下，后面再说

进行完message的基础配置之后，Handler将会正式的将message加入MessageQueue之中，接下来的代码就要到MessageQueue中去看了。

MessageQueue

Message入队流程

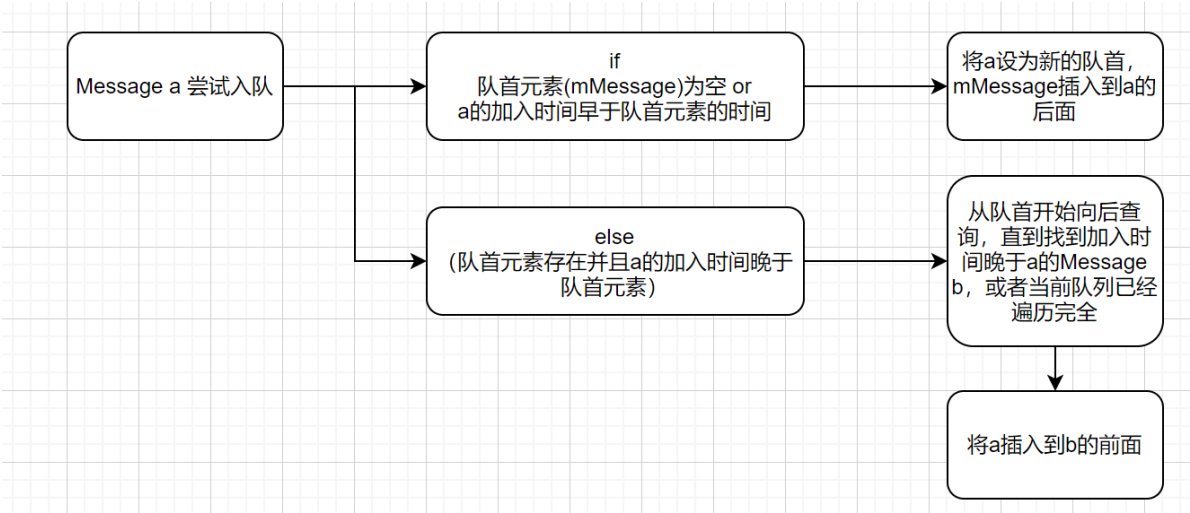
消息队列是这场Message发送之旅的下一站，从Handler的`enqueueMessage()`方法中可以很简单的看到，接下来的消息传递会调用到MessageQueue的`enqueueMessage()`方法，顺着这行调用，可以定位到下面这段代码

```
boolean enqueueMessage(Message msg, long when) {
    if (msg.target == null) {
        throw new IllegalArgumentException("Message must have a target.");
    }
    if (msg.isInUse()) {//isInUse()方法返回的是当前Message对象是否正在使用，正在使用的
Message不饿能入队
        throw new IllegalStateException(msg + " This message is already in
use.");
    }

    synchronized (this) {
        if (mQuitting) {//如果已经终止了looper的消息循环（线程已死 or 手动停止），则报错
            IllegalStateException e = new IllegalStateException(
                msg.target + " sending message to a Handler on a dead thread");
            Log.w(TAG, e.getMessage(), e);
            msg.recycle();
            return false;
        }

        msg.markInUse();//标志当前Message已经在使用
        msg.when = when;//标记当前Message的加入时间
        Message p = mMessages;//获取队首元素
        boolean needWake;//是否需要唤醒线程
        if (p == null || when == 0 || when < p.when) {//Mark 1 <------
            //如果队首元素为空或者当前元素的时间早于队首元素的时间，则将这个Message设为新的队
首元素
            // New head, wake up the event queue if blocked.
            msg.next = p;
            mMessages = msg;
            needWake = mBlocked;
```

```
        } else {
            // Inserted within the middle of the queue.  Usually we don't have
to wake
            // up the event queue unless there is a barrier at the head of the
queue
            // and the message is the earliest asynchronous message in the
queue.
            needWake = mBlocked && p.target == null && msg.isAsynchronous();
            Message prev;
            //轮询检索到合适的时间插入新的Message，保持MessageQueue的时间单调性
            for (;;) {
                prev = p;
                p = p.next;
                if (p == null || when < p.when) {
                    break;
                }
                if (needWake && p.isAsynchronous()) {
                    needWake = false;
                }
            }
            msg.next = p; // invariant: p == prev.next
            prev.next = msg;
        }

        // We can assume mPtr != 0 because mQuitting is false.
        if (needWake) {
            nativeWake(mPtr);
        }
    }
    return true;
}
```

mark 1之前的代码内容可以参考注释，让我们从mark 1开始详细看一下入队的逻辑

学一下郭神，我也给一个流程图



简而言之，就是按照when的时间顺序插入到MessageQueue的单链表之中，保持MessageQueue的时间上的单调性。

到现在为止Message入对的流程就已经基本完成了，让我们梳理一下，Message入队的大体流程如下

- Handler调用sendMessage方法
- sendMessage方法中获得该Thread的Looper的MessageQueue

- 将Message塞入MessageQueue

接下来看看Message是如何出队的

Message的出队流程

真正的Message的出队流程需要Looper的驱动，这里先抛开Looper，单纯的从MessageQueue的Message出口来理解一下Message的出队流程。

Message从MessageQueue走出的出口是`next()`方法，可以很轻松的定位到下面这个方法，我就切出和Message出队相关的重点部分来写一下注释。

```
Message next() {
    .......无所谓の代码........
    for (;;) {
        .......无所谓の代码........
        synchronized (this) {
            // Try to retrieve the next message.  Return if found.
            final long now = SystemClock.uptimeMillis();
            Message prevMsg = null;
            Message msg = mMessages;
            .......无所谓の代码........
            if (msg != null) {
                if (now < msg.when) {
                    // 这时候的队首元素时间还尚早，还不能出队，所以需要计算距离出队还剩余多
少时间
                    nextPollTimeoutMillis = (int) Math.min(msg.when - now,
Integer.MAX_VALUE);
                } else {
                    //时间合适，允许出队
                    mBlocked = false;//关闭阻塞
                    if (prevMsg != null) {//这个不为null的情况和同步屏障有关，暂时可以
跳过这里的逻辑
                        prevMsg.next = msg.next;
                    } else {
                        mMessages = msg.next;//将MessageQueue的队首元素后移
                    }
                    msg.next = null;//取出的队首元素清除掉后面的引用
                    if (DEBUG) Log.v(TAG, "Returning message: " + msg);
                    msg.markInUse();
                    return msg;
                }
            } else {
                // No more messages.
                nextPollTimeoutMillis = -1;
            }

            .......无所谓の代码........
        }
        .......无所谓の代码........
    }
}
```

简单的说，这就是一个从MessageQueue中获取下一个Message的方法，如果队列中没有消息，next方法就会一直阻塞，如果有，就返回当前队首Message

现在尚且生疏的部分估计就是如何从Thread的Looper中获取MessageQueue，下一部分就会详细介绍Looper，但是在开启Looper之前，还是先填一下坑：MessageQueue的同步屏障

同步屏障

Message的分类

> Handler中的Message可以分为三类：**同步消息**、**异步消息**、**同步屏障**。前两者可以通过Message中的`isAsynchronous()`方法加以判断，而同步屏障则是一种没有target的Message

```
//Message.java
public boolean isAsynchronous() {
 return (flags & FLAG_ASYNCHRONOUS) != 0;
}
```

在建立Handler时，其构造函数中的async属性就会确定这个handler发送的是什么Message，比如，默认的无参Handler构造方法如下

```
public Handler() {
    this(null, false);
}
```

其令行调用的另一个双参构造函数的第二个参数（也就是async）就会被设置为false，即这个Handler只会发送同步消息，但是如果在在构造函数中传入true，这个Handler就将发送异步消息，同步消息和异步消息在Message层面的唯一不同就是最上面提到的

`isAsynchronous()`返回值，同步消息是false，异步消息是true，而Message的这一属性在被Handler发送的时候就已经设置了。

那么什么是同步消息屏障呢?

同步屏障

前面已经提到了，同步屏障是一种特殊的Message，它并没有target（target为null），看到这里是不是有同学会感到疑惑，不是enqueueMessage的时候会判断target是否为null的吗，如果target为空不是会报错的吗？的确是这样，同步屏障的插入方式不是靠enqueueMessage，而是靠`postSyncBarrier()`方法强行插入到MessageQueue里面的，接下来康康这行代码

```
private int postSyncBarrier(long when) {
    // Enqueue a new sync barrier token.
    // We don't need to wake the queue because the purpose of a barrier is to
stall it.
    synchronized (this) {
        final int token = mNextBarrierToken++;//给这个同步屏障签一个token
        final Message msg = Message.obtain();//利用obtain方法更有效率的生成一个
Message
        msg.markInUse();
        msg.when = when;
        msg.arg1 = token;

        Message prev = null;
        Message p = mMessages;
        //下面的逻辑依旧是按照时间顺序将同步屏障插入到消息队列中
        if (when != 0) {
            while (p != null && p.when <= when) {
                prev = p;
                p = p.next;
            }
        }
        if (prev != null) { // invariant: p == prev.next
```

```
            msg.next = p;
            prev.next = msg;
        } else {
            msg.next = p;
            mMessages = msg;
        }
        return token;
    }
}
```

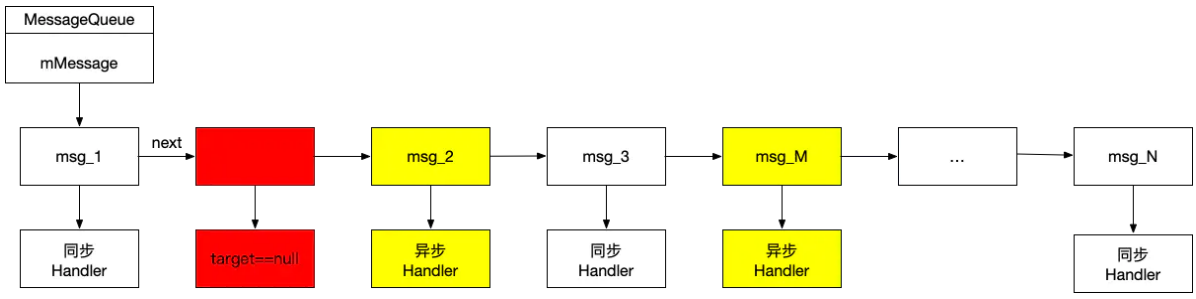
到上面为止，我们就已经在消息队列中插入了一个同步屏障，那么它会在哪里发挥作用呢?

Message出队时

根据上面出队的相关知识，我们可以很快定位到下面的这段代码

```
@UnsupportedAppUsage
Message next() {
    .......无所谓の代码........
    for (; ; ) {
        .......无所谓の代码........
        synchronized (this) {
            // Try to retrieve the next message.  Return if found.
            final long now = SystemClock.uptimeMillis();
            Message prevMsg = null;
            Message msg = mMessages;
            //TAG 1: 这里是同步消息屏障开启的重要代码
            if (msg != null && msg.target == null) {
                // Stalled by a barrier.  Find the next asynchronous message in
the queue.
                do {
                    prevMsg = msg;
                    msg = msg.next;
                } while (msg != null && !msg.isAsynchronous());
            }
            i.......无所谓の代码........
        }
        .......无所谓の代码........
    }
}
```

可以看到，如果当前Message是没有target的，就代表达到了当前的MessageQueue的同步消息屏障，队列会不断向后查询，**同步消息**将会被掠过，直到队列中出现了一个**非同步消息**或者消息队列已经为空，即将next的逻辑变为：弹出下一个异步消息。整体逻辑可以描述为：如果队列没有遍历到同步消息屏障，则正常按照时间弹出队首，如果遇到同步屏障，则只处理异步消息。就如同下图所示



并且可以明显的看到，如果next递归到同步消息屏障，出队的将不再是队首元素，也就是说同步消息屏障将一直横亘在队首，导致之后出队时处理的都是异步消息。

那么怎样清除同步消息队列呢？调用`removeSyncBarrier()`即可

使用场景

显然同步屏障可以提升异步消息的处理优先级，事实上这就是同步消息屏障的最大用处之一。Android的UI刷新很多地方都要用到消息机制，如果这些UI刷新的Message得不到优先处理，就有可能会引起页面卡顿，而适时的使用同步消息屏障就可以有效解决视图长时间不会得到刷新的问题

好，接下来就可以尽情的看Looper了

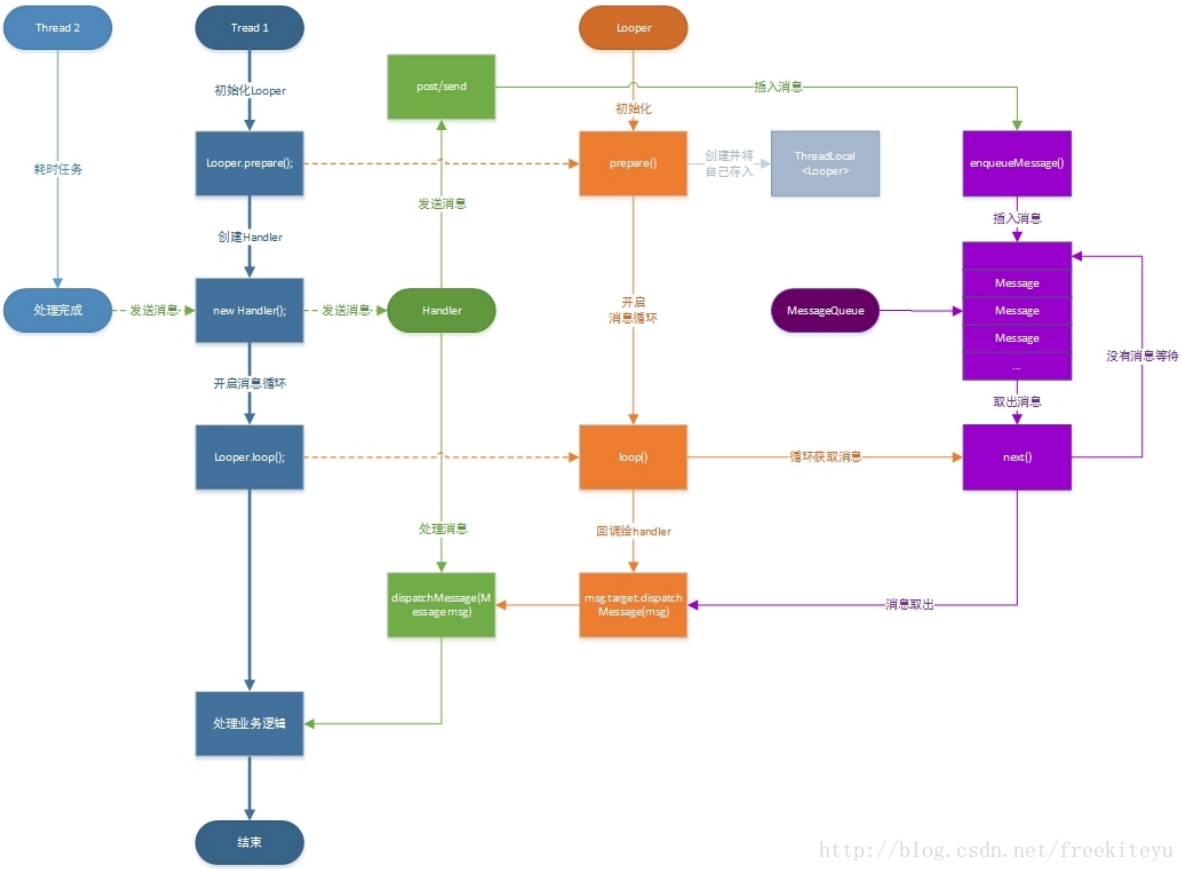
Looper

什么是Looper，看看百度百科的解释

> ~~张亨硕，生于1993年2月28日，游戏ID：looper，韩国《英雄联盟》职业选手，2016赛季RNG战队上单选手。~~
>
> ~~2013年夏季赛作为练习生被选入MVPO（SSW前身）战队，在2014年取得全球总决赛冠军[1]，2016年取得LPL春季赛冠军[2]，夏季赛亚军[3]。~~
>
> ~~2018年4月28日，Looper宣布退役[4]。~~
>
> 
>

不皮了

从最上面我给的那张不同组件之间的合作关系图也可以看出，Looper充当的是传动带的功能，用于输送Message，下面这张图也可以很清楚的说明Looper的作用，即不断地从MessageQueue中取出Message并dispatch给对应的Handler。



万事讲究一个源头，先来看看Looper是怎么形成的

万恶之源，Zygote


20-Android-邱天
我觉得不是移动的锅 <——是Zygote的锅

Zygote就是孵化器，它在Android系统中负责孵化新的进程，App的启动就需要Zygote去fork一个新的进程去给这个App启动，可以理解为App启动的源头。（如果你要说Launcher是启动源头，那么Launcher的启动者是谁呢，如果你要说BootLoader是启动源头，那还是你比较底层）在Zygote的驱动下，Android系统将启动虚拟机，fork新的进程，并开启**主线程Looper**。

为了更直观的体验Android的入口，这里就放一下RuntimeInit的查询主函数入口的方法

```
protected static Runnable findStaticMain(String className, String[] argv,
ClassLoader classLoader) {
    Class<?> cl;

    try {
        cl = Class.forName(className, true, classLoader);
    } catch (ClassNotFoundException ex) {
        throw new RuntimeException(
                "Missing class when invoking static main " + className,
        ex);
    }

    Method m;
    try {
        //剑指main函数
        m = cl.getMethod("main", new Class[] { String[].class });
    } catch (NoSuchMethodException ex) {
        throw new RuntimeException(
                "Missing static main on " + className, ex);
```

```
    } catch (SecurityException ex) {
        throw new RuntimeException(
                "Problem getting static main on " + className, ex);
    }

    int modifiers = m.getModifiers();
    if (! (Modifier.isStatic(modifiers) && Modifier.isPublic(modifiers))) {
        throw new RuntimeException(
                "Main method is not public and static on " + className);
    }

    /*
     * This throw gets caught in ZygoteInit.main(), which responds
     * by invoking the exception's run() method. This arrangement
     * clears up all the stack frames that were required in setting
     * up the process.
     */
    return new MethodAndArgsCaller(m, argv);
}
```

这里的main函数位于ActivityThread里面

```
public static void main(String[] args) {
    .......无所谓の代码........
    //Looperの现
    Looper.prepareMainLooper();
    .......无所谓の代码........
    ActivityThread thread = new ActivityThread();
    thread.attach(false, startSeq);

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

    if (false) {
        Looper.myLooper().setMessageLogging(new
                LogPrinter(Log.DEBUG, "ActivityThread"));
    }

    // End of event ActivityThreadMain.
    Trace.traceEnd(Trace.TRACE_TAG_ACTIVITY_MANAGER);
    Looper.loop();

    throw new RuntimeException("Main thread loop unexpectedly exited");
}
```

可以看到，一旦App被启动，就会立刻在主线程里面建立一个Looper并投入loop。

不如定位到prepareMainLooper中，来看一下Looper是怎么形成的

```
public static void prepareMainLooper() {
    prepare(false);
    synchronized (Looper.class) {
        if (sMainLooper != null) {
            throw new IllegalStateException("The main Looper has already been
prepared.");
        }
        sMainLooper = myLooper();
    }
}
```

可以看到，主线程Looper的真正形成是调用了`prepare()`方法，接下来就看一下这个`prepare()`方法是怎么样的

Looper.prepare()

```
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));
}
```

在继续下去之前，先介绍一个东西：ThreadLocal

ThreadLocal

首先推荐一个博客，就是最近郭霖公众号上推荐的那篇ThreadLocal的源码解析，课下有时间的同学可以去了解一下

[源码篇：ThreadLocal的奇思妙想（万字图文）]

先从宏观上解释一下什么是ThreadLocal

> ThreadLocal是一个关于创建线程局部变量的类。
>
> 通常情况下，我们创建的变量是可以被任何一个线程访问并修改的。而使用ThreadLocal创建的变量只能被当前线程访问，其他线程则无法访问和修改。

简单的说，ThreadLocal创建了一个线程隔离的映射关系，就是从`ThreadLocal<T>`到T的一个映射，通过ThreadLocal可以在本线程中访问到T，而在其他线程中无法访问。

> 其实这里非常有趣，当调用ThreadLocal的set方法赋值的时候，其实ThreadLocal是将自己当作key，吧自己和被set进去的对象作为键值对存储到ThreadLoalMap之中的，后者为Thread所持有，可以通过Thread的threadLocals属性访问到

Looper中的ThreadLocal（sThreadLocal）是存储Looper的，在prepare的时候，如果sThreadLocal映射到的Looper不为空，就代表

当前线程已经产生过一次Looper，这是不允许的，故会抛出一个线程只能产生一个Looper的错误。

如果尚未产生Looper，就会生成一个新的Looper并且通过ThreadLocal存储起来。

Looper形成之后，接下来就是要让Looper循环起来，也就是调用Looper.prepare方法

Looper.loop()

废话不多说，直接上源码

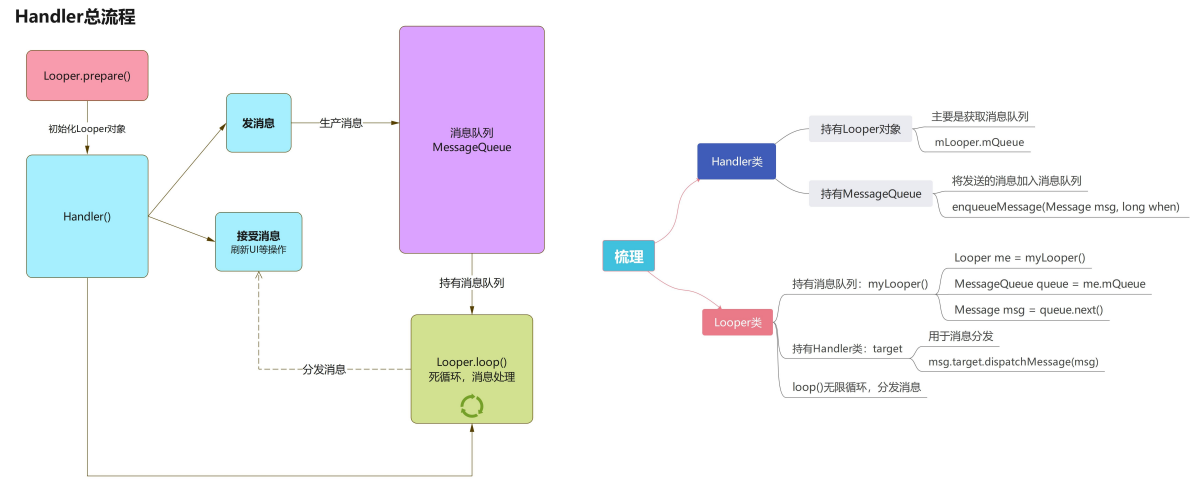
```
public static void loop() {
    final Looper me = myLooper();
    if (me == null) {
        throw new RuntimeException("No Looper; Looper.prepare() wasn't called on
this thread.");
    }
    final MessageQueue queue = me.mQueue;
    .......无所谓の代码........
    for (;;) {
        Message msg = queue.next(); // might block
        if (msg == null) {
            // No message indicates that the message queue is quitting.
            return;
        }
        .......无所谓の代码........
        final long dispatchStart = needStartTime ? SystemClock.uptimeMillis() :
0;
        final long dispatchEnd;
        try {
            msg.target.dispatchMessage(msg);
            dispatchEnd = needEndTime ? SystemClock.uptimeMillis() : 0;
        } finally {
            if (traceTag != 0) {
                Trace.traceEnd(traceTag);
            }
        }
        .......无所谓の代码........
        msg.recycleUnchecked();
    }
}
```

可以看到，当Looper开始loop时，将会开启一个**死循环**，退出条件为当前Looper的MessageQueue内部不再有message

> - 这个MessageQueue是在Looper的构造函数里面同步生成的
> - 如果MessageQueue没有Message了，其实也就是调用了quit方法，也就是说如果外部不干预（调用Looper.quit()方法），这个循环将永远都不会终止

简单的来看，loop的作用就是不断地从MessageQueue中取出Message，然后调用dispatch方法将Message分发给各自的Handler。

到现在为止，我们实现了一个逻辑上的回环，也就是从Handler发送Message到Handler dispatchMessage，**如果不严谨的说**，调用了这个方法之后接下来就会执行到Handler的handleMessage方法来在Handler创建的线程去处理message



不知道看到这里你是否有些疑惑，主线程中Looper的loop是死循环方法，那它为什么没有导致ANR（Application Not Responding，应用未响应）呢?

ActivityThread的main方法主要作用就是做消息循环，一旦退出消息循环，主线程运行完毕，那么你的应用也就退出了。

Android是事件驱动的，在Loop.loop()中不断接收事件、处理事件，而Activity的生命周期都依靠于主线程的Loop.loop()来调度，所以可想而知它的存活周期和Activity也是一致的。当没有事件需要处理时，主线程就会阻塞；当子线程往消息队列发送消息，并且往管道文件写数据时，主线程就被唤醒。

也就是说，其实App的运转靠的就是Looper的死循环和不断拿出Message，这个死循环就是App的运作本身，自然不会导致ANR

下面总结一下几点要注意的事项

> - 如果要发送Message到子线程，你需要先Looper.prepare()并Looper.loop()一下
> - 如果你要使用ThreadLocal，记得要及时清理引用，因为ThreadLocalMap拿到的是value的强引用，可能会导致内存泄漏

> 其实正常的来说，消息机制应该介绍到这里就结束了，但是不知道你是否有这样的疑惑虽然Handler的Message机制可以实现Message的收发，但这和线程调度有什么关系呢？Message究竟是如何做到在线程a里面发送，最后到线程b里面接收的呢?
>
> 我感觉如果想要更好的理解消息机制，回答这个问题是必须的

到底是在哪里实现的线程调度

为什么直接在子线程中回调不能实现线程调度

去年差不多寒假的时候，我写下来这样几行脑溢血代码，把当时带我的学长（丁神）折磨的不行

```
new Thread(
    () -> {
        Message msg = new Message();
        mHandler.handleMessage(msg);//没有十年脑溢血写不出这行代码
    }
).start();
```

为什么这种直接回调不能实现线程调度呢?

在JVM中，虚拟机栈是线程隔离的，如果方法a是在UI线程中调用，那么方法a的局部变量、对象引用等就会被打包成栈帧存储在UI线程的虚拟机栈中，相反如果是在子线程中调用，就会存储在子线程的虚拟机栈中，存储在不同的栈中意味着调用他的方法和他方法的返回值会归属于不同的栈中，而想要实现线程调度，就是要想办法把把信息传递给主线程，最后在主线程中调用方法，将方法加入主线程的虚拟机栈中才可以。直接进行接口回调显然会将回到的方法加入到子线程的虚拟机栈帧中。

现在的瓶颈

首先，JVM的堆内存是线程共享的，如果我将子线程中的Message放在堆内存中，然后在主线程中访问这个堆内存中的Message，就可实现信息的交互。但这样做的问题在于子线程无法通知主线程何时Message已经被写入堆内存，因为通过直接回调的方法将会导致方法的栈帧被加入到子线程中去。简单的说，实现子线程和主线程之间的数据互通并不困难，真正困难的是如何互相通知你想要的Message已经准备好并且写入了堆内存。

不能回调，那就轮询

其实消息机制就是实现了这个通知的过程，子线程的将Message写入MessageQueue，Looper轮询获取Message，如果有就回调到Handler进行下发处理，这样的回调属于主线程内的回调，也就实现了线程调度的目的

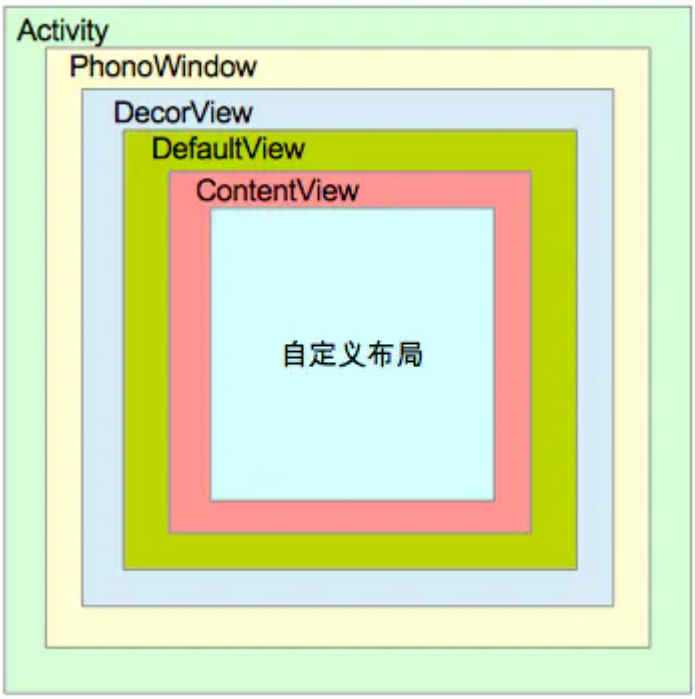
好接下来进入郭神主场，事件分发的介绍

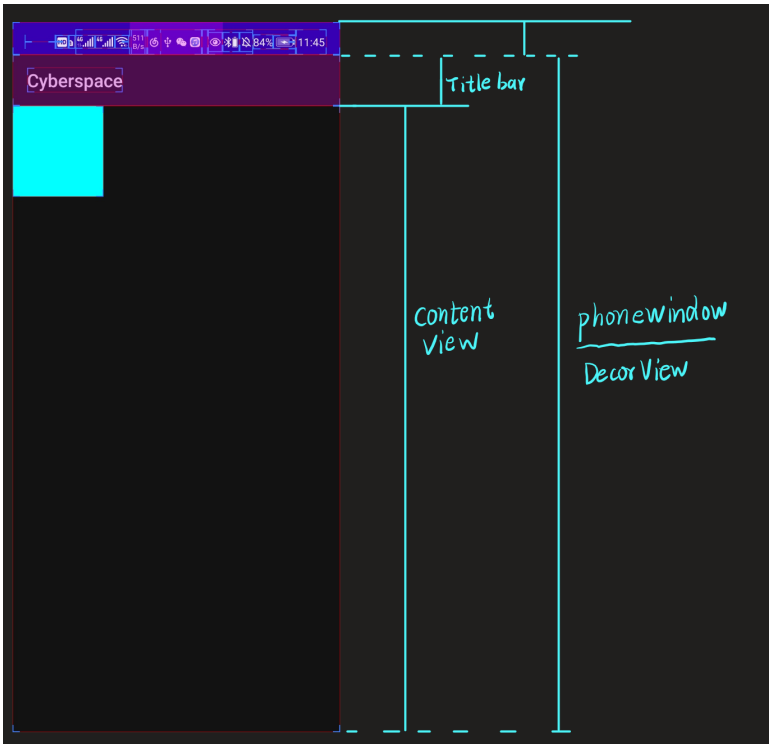
Android事件分发机制

App的视图层级结构

虽然上上节课我有提到这个东西，思来想去还是决定再贫嘴一次

下左图为Android Activity的UI构成，Activity将持有一个Window对象，而Window对象一般是由PhoneWindow实现的，这些组件在手机页面中表示一般如右所示





`DecorView`就是应用窗口的`Root View`，`DecorView`将要呈现的具体内容展现在了`PhoneWindow`上，而`DecorView`又分为两部分，一部分是顶部的`Title（ActionBar）`，另一部分是`ContentView`（本身为`FrameLayout`），这两者被放置在一个`LinearLayout`之中。大名鼎鼎的`setContentView`方法就是在为`ContentView`设置内容

MotionEvent

> 用户每次对屏幕的操作都会被封装成事件（MotionEvent）

总共分为四种事件类型

- ACTION_DOWN: 按下view (所有事件的开始)
- ACTION_UP: 抬起View (一般情况下标志着事件的结束)
- ACTION_MOVE: 滑动View
- ACTION_CANCEL: 事件结束 (非人为原因)

每一次点击屏幕并不只会产生一个事件, 而是会产生一系列事件: 事件序列

事件序列是如何传递的呢？这里先举一个和事件分发很相似的例子

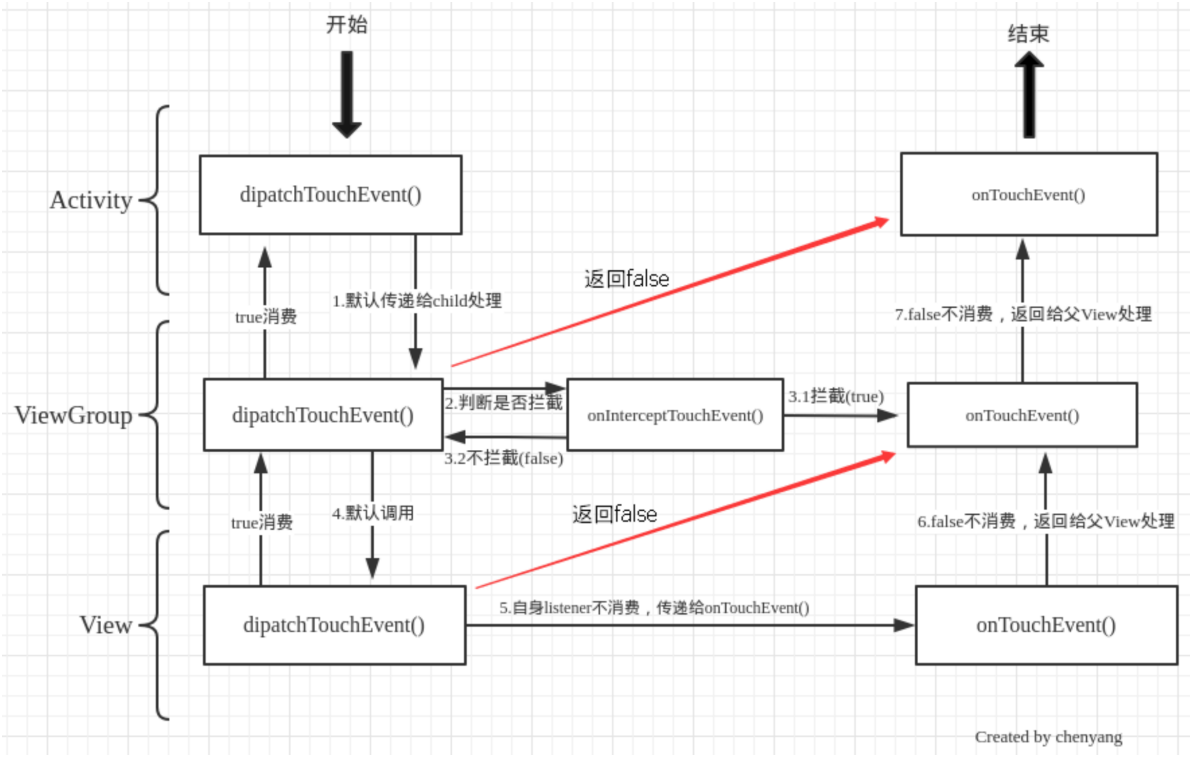
> 公司里面有一个需求要写，假设这个公司有5个段位的程序员，老板先把工作交给最顶的那一个程序员（傲世宗师），这个程序员感觉这个工作别人也可以处理，就把这个工作交给比他段位低一点的程序员处理（超凡大师），大师段位程序员也是优先将任务丢给段位更低一点的程序员，这样锅不断下发，直到最底层程序员张煜（坚韧黑铁）接到了这个需求，他看了一下，发现自己并不能写这个需求，就告诉自己上司：“这玩意爷不会写”，之后上司就会判断一下自己是否可以写这个需求，如果不行，就会告诉比自己更高一级的程序员，把锅丢给他，然后循环往复。最坏的情况就是傲视宗师的程序员也不能解决这个问题，锅又回到了老板手里，然后老板（最强王者段位，其实老板曾是Google核心开发者）拿出键盘自己写需求去了。

- 当手指接触到屏幕的时候，将会产生一系列点击事件
- 这些点击事件首先到了Activity手里，Activity将事件交付到PhoneWindow，之后PhoneWindow会将点击事件交付到DecorView加以处理，如果位置正确，DecorView会将事件交付给ContentView
- 在ContentView中，如果事件传递到一个ViewGroup中，如果ViewGroup感觉自己需要拦截这个事件，就将拦截这个事件并交由自己处理，这个事件也就将就此消失。相反，如果ViewGroup不准备拦截这个事件，就会继续下发给自己的子view处理
- 如果子View也是一个ViewGroup，它会执行和上面相同的逻辑。如果是一个View，那它没有继续下发的可能性了，就只能选择自己是否要消费这个事件，如果消费，这个事件就将消失，反之则会向上返回给自己的上一层级的ViewGroup，后者判断自己是否要消费。

说了这么多，还是从代码层面看一下事件分发是如何实现的

事件分发有三大核心方法

`dispatchTouchEvent` 事件分发，返回值为是否消耗这个事件
`onInterceptTouchEvent` 事件拦截，返回值为是否拦截这个事件
`onTouchEvent` 事件消耗，返回值为是否消耗这个事件



dispatchTouchEvent

此方法用来进行事件的分发，如果事件能够传递到当前view，就一定会调用这个方法，如果这个方法的返回值为true，就代表将要在这个层级消费这些事件，如果为false，就代表要把锅退给自己的上一级

> 在上面写需求的例子中，这个方法的返回值代表的就是在目前这个层级，需求是否已经被写完了，不论是自己的下级写完的，还是自己写完的。

onInterceptTouchEvent

仅仅ViewGroup具有这个方法，表示View是否拦截某个事件，如果拦截，则不会将事件继续下发，即将自己这一层级当作最底层处理，要么自己消费点击事件，要么把事件退给上一级。

> 在上面的例子中，就代表当前层级是否**决定**要处理这个需求，不再丢给自己的下属，当然，这里只是决定要处理，能不能处理还要另说。

onTouchEvent

处理事件的方法，当事件到达View或者ViewGroup决定拦截事件之后，会优先检查当前View/ViewGroup是否有设置OnTouchLinstner，如果有就会优先给到它去处理，这时候如何处理还要看onTouch的返回值，如果返回为false，就会调用**onTouchEvent**方法，如果走到**onTouchEvent**方法中，假设有**OnClickListener**，就将调用到onClick方法。

伪代码展示

- Activity

```
//Activity的核心伪代码
public boolean dispatchTouchEvent(MotionEvent ev) {
    if (child.dispatchTouchEvent(ev)) { //先看子view是否消费事件
        return true;
    } else {
        return onTouchEvent(ev); //如果子View没有消费该事件，则调用自身的
        onTouchEvent()处理。
    }
}
```

- ViewGroup

```
//ViewGroup的核心伪代码
public boolean dispatchTouchEvent(MotionEvent ev) {
    if (!onInterceptTouchEvent(ev)) {
        return child.dispatchTouchEvent(ev); //不拦截，则传给子View进行分发理
    } else {
        return onTouchEvent(ev); //拦截事件，交由自身对象的onTouchEvent()方法处理
    }
}
```

- View

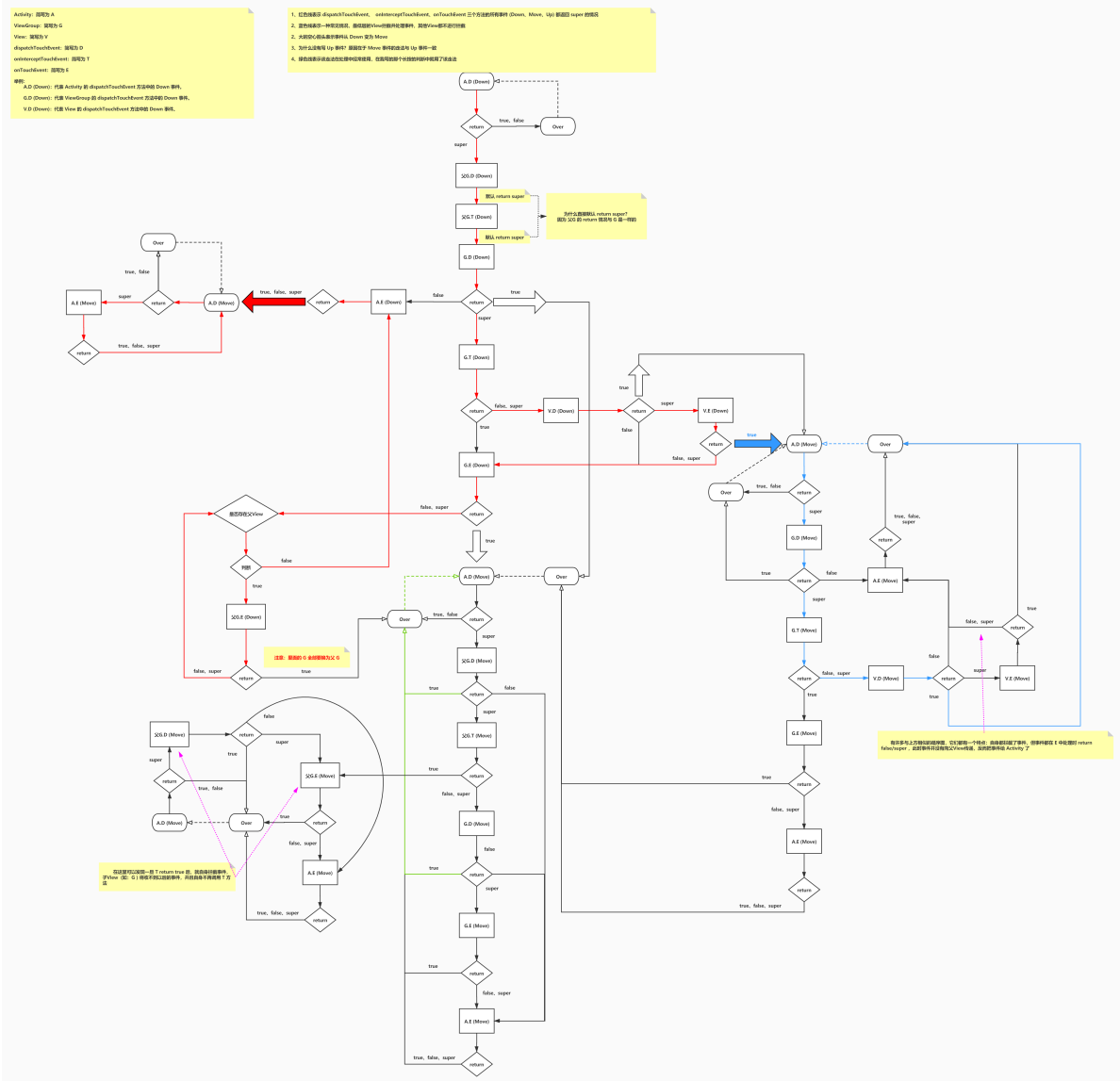
```
//ViewGroup的核心伪代码
public boolean dispatchTouchEvent(MotionEvent ev) {
    if (!onInterceptTouchEvent(ev)) {
        return child.dispatchTouchEvent(ev); //不拦截，则传给子View进行分发处理
    } else {
        return onTouchEvent(ev); //拦截事件，交由自身对象的onTouchEvent()方法处理
    }
}
```

其他一些东西

- 在dispatch的时候会先判断当前view是否可见，如果不可见就直接返回为false
- 一旦View决定拦截事件，那么同一个时间序列的所有时间啊都会交由它处理
- 如果一个View拦截了事件但是却没有消耗掉DOWN事件，那么余下的事件都不会再交付给它处理，会直接返还给上一层级的View，交由它的onTouchEvent处理
- 如果当前view消耗了DOWN事件，但是没有消耗除了DOWN之外的其他事件，这些事件会直接冒泡到Activity层级加以处理

源码中的处理

郭神的神图贴在这里，代表源码描述的逻辑。



滑动冲突

产生的场景

- 父ViewGroup和子View的滑动方向一致

  如果父VIewGroup和子View的滑动方向一致，如果我们需要让两者都滑动。当ViewGroup接收到事件以后，由于不会拦截事件，就会将事件传给子View，一旦有子View处理了这个ACTION_DOWN事件，在默认情况下，这个事件列的后续事件都将交给这个子View处理，这个时候由于子View是可以滑动的，但是父ViewGroup始终滑动不了

- 父ViewGroup和子View的滑动方向不一致

解决方案

- 父View拦截

  可以通过ViewGroup的`onInterceptTouchEvent()`方法去有选择的拦截，把事件不给子view

- 子View反馈

  通过`getParent().requestDisallowInterceptTouchEvent()`请求不进行拦截