

当我们按下屏幕后，事件是怎么传递到应用的？



易保山
移动端开发

6 人赞同了该文章

赞同 6

分享

Android 是一个有用户界面（GUI）的操作系统，在它诞生之初，就是为带有触摸屏的手持设备准备的。作为提供给用户最重要的交互方式之一，了解触摸系统是怎么工作的，对于实际的项目开发有着非常大的帮助

本篇是图形系列的第五篇文章，在之前的几篇文章中，我们分别了解了 Android 系统[渲染/合成的底层原理]和[自定义 View / ViewGroup 的流程]

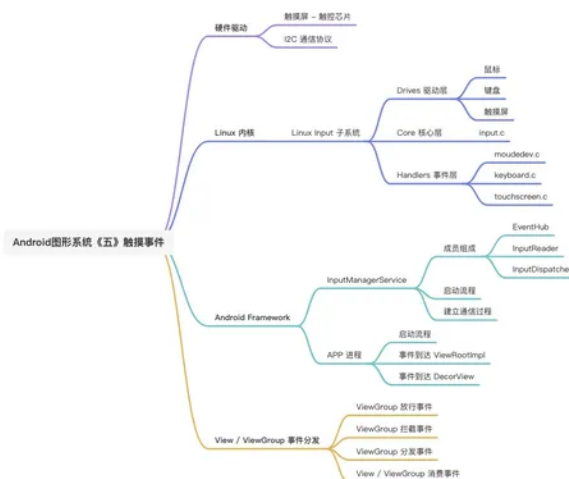
今天我们来聊聊图形系统中，另一个老生常谈的话题：**事件分发**

和以往相比，今天的文章会稍微有那么一点点不一样

我们会从认识硬件驱动开始，自底向上，一步步的来了解，事件是怎么到达的系统内核，内核又是怎么传递到应用，以及应用最终是如何消费掉事件的

废话不多说，我们直接进入正题，let's go

前排提醒：全文 1.5w 字，建议阅读时长 30 分钟



知乎 @易保山
@稀土掘金技术社区

android_graphic_v5_overview.png

一、触摸事件的起源（Linux Kernel）

Android 输入事件的类型，和分发的流程都比较复杂，除了触摸事件外，系统还有来自 鼠标、键盘、音量键、电源键 等其他 Input 设备的事件需要处理

我们日常开发接触比较多的是 ‘触摸事件’，因此，本文主要讨论的是 ‘触摸事件’ 的分发流程，其他类型的输入事件顺带会提一嘴，不是本文的重点

本文一共分为三大部分：

第一部分介绍 ‘触摸事件的起源’，主要讲的是驱动上报原始事件，内核解析原始事件并保存到设备文件中，以供 Framework 读取分发

第二部分介绍 ‘触摸事件的传递’，主要讲的是 InputManagerService 怎么把事件传递到应用进程，并分发给目标 Window

赞同 6

添加评论

分享

喜欢

在文章的开头，我们先来聊聊第一部分的内容，触摸事件的起源

从硬件到内核

在《当我们点击“微信”应用后，它是怎么显示出来的？》这篇文章中，为了搞清楚 Android 设备的绘图硬件是什么，我们拆了一台小米11 手机

今天，我们继续来拆小米



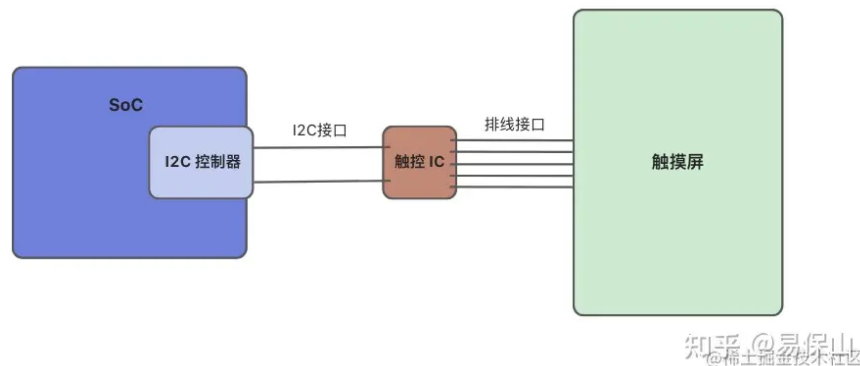
android_graphic_v5_mi10.png

图片来源：【集微拆评】小米10拆解：内部布局与iPhone相似，1亿像素主摄吸睛

如上图，这是拆解后小米10的内部布局，图中左边黄色箭头所指的部分，是触摸屏的触控芯片

小米10 触控芯片使用的是，来自意法半导体的 "FJABH"，这块芯片是用来干嘛的呢？

用来和 CPU 进行通信的



android_graphic_v5_ic_touch.png

图片是重绘版，参考自：blog.csdn.net/qq_397979...

我们知道，当我们按下触摸屏后，屏幕的电压/电流大小会发生变化（不展开讨论触摸屏工作原理）

I²C 是硬件之间常用的一种通信协议，它规定了什么表示起始、停止、应答和非应答等一系列信号

当然，作为应用开发，我们无需关心他们的通信细节

我们只需要知道：“一旦触摸屏的信号发生变化，触控芯片就能通过 I²C 总线通知到 CPU”。了解这一点就够了

好了，现在触摸信号已经能被 CPU 读取了，接下来我们看 CPU，也就是操作系统如何处理触摸信号

内核创建设备文件

我们都知道，Google 使用 Linux 作为 Android 系统的内核，管理着主板上的内存、网卡、硬盘等硬件设备，其中也包括 CPU

在上一小节中，触摸屏已经和 CPU 建立了通信。也就是说，操作系统可以读取触摸屏发送过来的信号了

接下来的工作重点分为两个部分，一是制定触摸屏具体的上报规则；二是想办法把设备发生的事件报告给应用程序

先来看设备的上报规则，我们以键盘事件举例，同样都是按下 'A' 按键

达尔优 键盘上报的是：0010

罗技 键盘上报的是：0001

同一个按键事件，两个键盘厂商上报的按键值却不相同，这显然是不行的。

所以，只有上一小节的通信规则（I²C）还不够，我们还需要制定一个内容规则，来规范各个厂家发送的数据内容

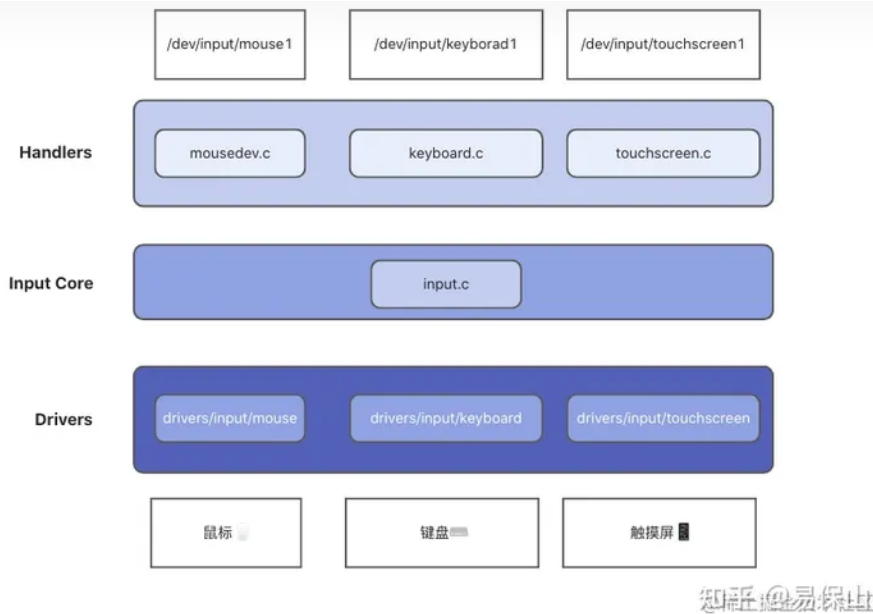
说到输入规范，这就不能不提 Linux 的 Input 子系统了

在 2001 年发布的 `f="elixir.bootlin.com/linu...">[2.4.0]` 版本，Linux 首次加入了 Input 子系统的代码，为的就是将输入设备的共性抽象出来，制定统一的输入规则

首发版本只支持手柄、鼠标和键盘这三种硬件，在随后 2002 年发布的 `= "elixir.bootlin.com/linu...">[2.5.25]` 版本中，加入了对触摸屏的支持

这样一来，厂商只需要按照 Linux 制定的规范，来上报按键值、屏幕坐标等信息即可，上报规则的问题就解决了

除了规范输入内容，Input 子系统还为应用程序提供了操作/读取输入设备的接口，来看框架图：



android_graphic_v5_linux_input_system.png

图片来源：自己画的

如图，Input 子系统分为三层：

- 最下层：输入设备驱动层，`drivers/input/xxx`，这里就是各大厂商需要遵循的协议规范，向内核层报告输入的内容
- 中间层：输入核心层，`input.c` 属于这一层。这是 Linux 核心逻辑，用来管理设备添加、卸载等操作，事件提供给应用前的准备工作
- 最上层：输入事件驱动层，到这里硬件驱动已经抽象为设备文件了，对应 `/dev/input/xxx`，硬件驱动发送的数据就保存在该路径下的各个设备文件中，等待应用读取

最下面的 Drivers 层，是 Linux 平台对各种输入设备的规范，各大厂商都需要去遵循该协议，否则 Linux 内核无法识别，设备也就无法正常工作

然后是中间的核心层，它是输入设备驱动的管理层，在输入框架中起着承上启下的作用：向下提供驱动层的接口，向上提供事件处理层的接口

我们来看一眼 `input.c` 中的几个关键方法，也就大概知道它提供了哪些功能

```
/drivers/input/input.c
class input { //Linux input 框架的核心层，为驱动层提供设备注册和操作接口

    /* 设备注册 */
    int input_register_device(input_dev *dev); // 注册一个 input 设备到内核
    void input_unregister_device(input_dev *dev); // 从内核注销掉一个 input 设备

    /* 设备连接 */
    int input_attach_handler(input_dev *dev, input_handler *handler);
    void input_disconnect_device(input_dev *dev);

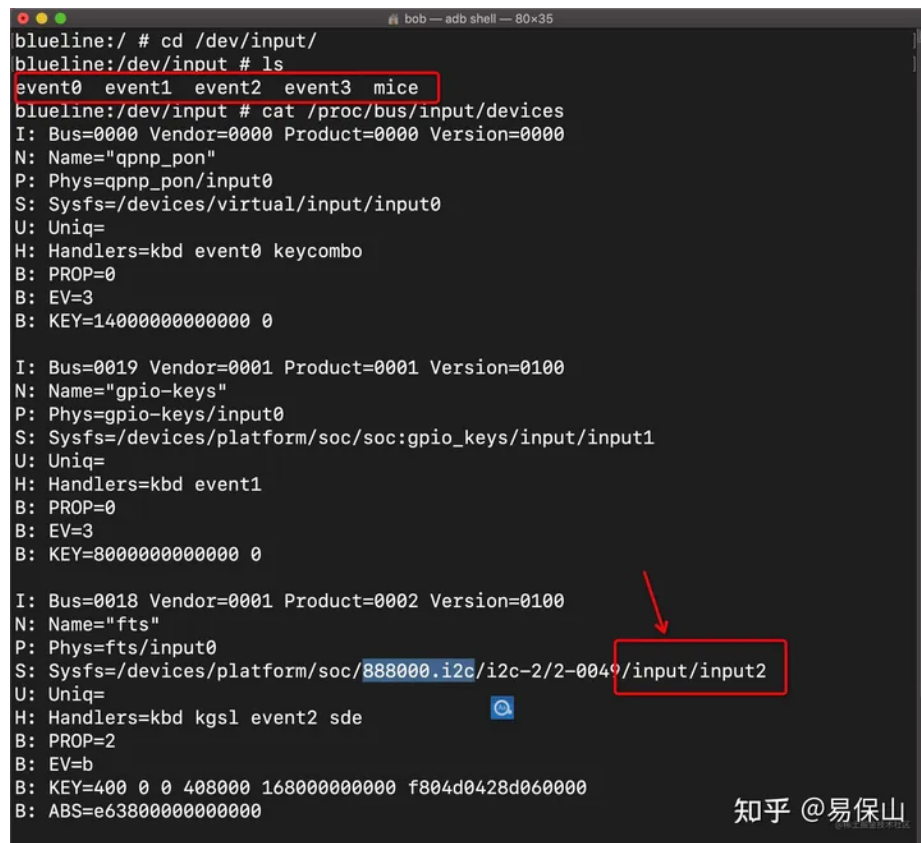
    /* 事件上报 */
    void input_handle_event(input_dev *dev, type, code, value);

    /* 应用程序数据读取 */
    int input_event_to_user(input_dev *dev, type, code, value);

}
```

从代码来看，核心层负责 设备的注册、设备的连接、事件上报 和 数据读取。这些设备、事件的具体实现逻辑我们这里不展开讨论，太长了。

我们可以在 `/dev/input/xxx` 找到所有已加载成功的输入设备，它们就是事件驱动层创建的设备文件



```

blueline:/ # cd /dev/input/
blueline:/dev/input # ls
event0 event1 event2 event3 mice
blueline:/dev/input # cat /proc/bus/input/devices
I: Bus=0000 Vendor=0000 Product=0000 Version=0000
N: Name="qnpn_pon"
P: Phys=qnpn_pon/input0
S: Sysfs=/devices/virtual/input/input0
U: Uniq=
H: Handlers=kbd event0 keycombo
B: PROP=0
B: EV=3
B: KEY=14000000000000 0

I: Bus=0019 Vendor=0001 Product=0001 Version=0100
N: Name="gpio-keys"
P: Phys=gpio-keys/input0
S: Sysfs=/devices/platform/soc/soc:gpio_keys/input/input1
U: Uniq=
H: Handlers=kbd event1
B: PROP=0
B: EV=3
B: KEY=8000000000000 0

I: Bus=0018 Vendor=0001 Product=0002 Version=0100
N: Name="fts"
P: Phys=fts/input0
S: Sysfs=/devices/platform/soc/888000.i2c/i2c-2/2-0049/input/input2
U: Uniq=
H: Handlers=kbd kgs1 event2 sde
B: PROP=2
B: EV=b
B: KEY=400 0 0 408000 168000000000 f804d0428d060000
B: ABS=e638000000000000
  
```

知乎 @易保山

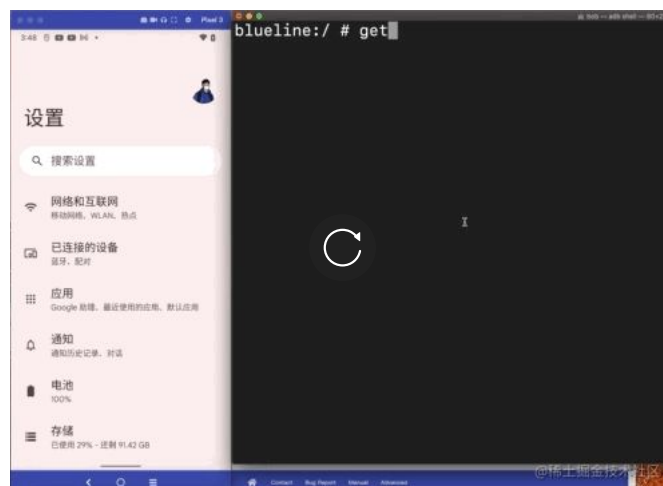
android_graphic_v5_pixel3_devices.jpg

图片来源：自己截的

如图所示，我手里的 Pixel 3 在 `/dev/input/` 这个路径下，发现了4个输入设备，从 `event0` 到 `event3`

我们可以用 `'cat /proc/bus/input/devices'` 命令，查看每个输入设备的信息，我这台手机 `event2` 节点是触摸屏的设备文件（`'* name = fts'` 表示的触控驱动厂商是 '敦泰 '*'）

接着，我们还可以用 `'getevent'` 命令打开这个设备文件，获取它发送的原始数据



android_graphic_v5_pixel3_getevent.GIF

图片来源：自己录的

你看，当我们滑动屏幕时，终端窗口会不停的打印来自 `event2` 的触摸

首先，按下触摸屏后，`触控芯片` 捕获到电压/电流的变化，计算出位置坐标后，通过 `I2C` 总线汇报给 CPU

接着，我们需要统一通信内容的规则，在 Linux 平台下，触控芯片需要实现 `input.c` 协议，事件按照规定的协议上报给内核系统

最后，等到设备开机，内核加载触摸屏驱动，再然后是设备的 `注册/连接/上报` 的过程

到这里，内核已经为我们收集好触摸屏的输入事件，并存放在了 `eventX` 设备文件中，Linux Input 子系统的任务已经完成

接下来我们看 Android 系统的框架层 (Framework) 是如何处理触摸事件的

二、触摸事件的传递 (Android Framework)

上回书说到，触摸屏上报的事件已经保存到 `/dev/input/xxx` 设备文件中，那么系统接下来的任务是：

读取触摸事件，并封装成 `MotionEvent` / `KeyEvent` 对象，最后分发给正在运行的 APP 使用

“读取”和“分发”，是 Android Framework 的主线任务

在接下来的章节中，我们将主要围绕着这两件事展开

ps：本章的“事件分发”探讨的是，如何将触摸事件从设备文件传递到 APP 进程，和 View 的事件分发不是一回事儿，注意别搞混了

初识 InputManagerService

我们都知道，在 Framework 中，输入事件是由 InputManagerService (后续简称IMS) 来管理

我们又知道，InputManagerService 是 Java 层代码，不可能直接调用到 Linux 内核层的 Input 框架来获取输入事件

因此，IMS 必然需要 native 层的支持，才能实现对事件的读取与分发

实际的 InputManagerService 实现一共分为三层

1. **native 层，这是 IMS 的核心层，负责 读取/分发 事件，** `EventHub.cpp`、`InputReader.cpp`、`InputDispatcher.cpp` 三员大将都在这
2. **jni 层，主要是对 native 做转发，另外负责创建对象啥的，不怎么需要关注**
3. **Java 层，主要负责通信部分，和 WMS 同步窗口数据啦，和 APP 跨进程通信啦等等**

在这其中，只有 native 层稍微有那么一点点复杂，因为数据的读取和分发都发生在 native 层

好，接下来，我们来认识 native 层的主力人员

注意，我们本小节只关注 native 中各个角色有哪些方法功能，做了哪些事情

至于对象什么时候创建，运行在哪个进程，哪个线程，这个后面会讲到，不是本小节关注的重点

1、EventHub

EventHub 的作用是监听、读取 `/dev/input` 目录下产生的新事件，并封装成 `RawEvent` 结构体供其他人使用

文件在 `frameworks/native/services/inputflinger/EventHub.cpp`

```
//frameworks/native/services/inputflinger/EventHub.cpp
class EventHub {

    EventHub::EventHub(void) {
        mEpollFd = epoll_create(EPOLL_SIZE_HINT); // 创建 epoll，用于监听设备文件是否有可
        mInotifyFd = inotify_init(); // 创建 inotify，用于监听文件系统是否变化，有变化说明
    }

    size_t EventHub::getEvents( timeoutMillis, buffer, bufferSize) {
        //getEvents() 是 IMS 的核心，该方法一共做了两件事
        // 1. 监听设备插拔动作，执行对应的设备的打开/卸载操作，并生成 RawEvent 结构通知调用者
        // 2. 监听输入设备文件的事件变化
        //如果没有任何事件发生，调用 epoll_wait() 函数执行等待
        return event - buffer; // 返回读取到的事件数量
    }

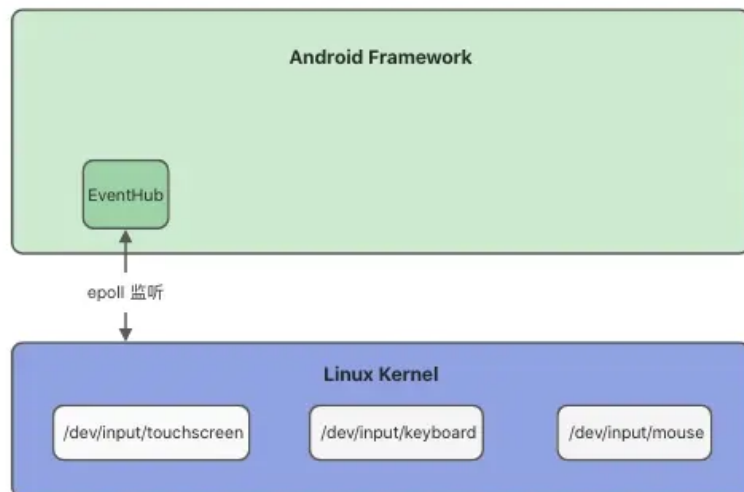
}
```

EventHub 在初始化时，会创建 mEpollFd 和 mInotifyFd 两个 fd，利用 inotify 机制监听设备增删事件，利用 epoll 监听设备文件读写状态变化

然后是 getEvents() 方法，这个方法用来获取当前的设备事件，包括设备数量变化，和设备上报的事件，是 IMS 获取消息的“核心方法”

如果没有任何事件发生，getEvents() 会发生阻塞，直到有事件发生才返回

EventHub 整个过程如下图



知乎@易保山

android_graphic_v5_native_1.png

图片来源：自己画的

2、InputReader

从类的命名就能看出来，InputReader 的职责是读取输入消息

不过，它可不是只会读取事件，拿到原始事件以后，还需要解析事件，拆解为 按键、触摸屏、鼠标等，根据不同的输入，封装成不同的对象，提交到消息队列等待分发


```

/frameworks/native/services/inputflinger/InputReader.cpp
class InputReader {

    class InputReaderThread : Thread { //内部类

        /*InputReaderThread 线程启动后，循环将不断地执行 InputReader#loopOnce()函数*/
        bool InputReaderThread::threadLoop() {
            mReader->loopOnce();
            return true;
        }
    }

    void InputReader::loopOnce(); // 核心方法，负责读取事件，解析事件
}

```

InputReader 类的核心方法是 `InputReader#loopOnce()`，它负责读取事件和解析事件

`loopOnce()` 方法被 `InputReaderThread` 线程的 `threadLoop()` 所调用，`InputReaderThread` 是 `InputReader` 的内部线程类

和 Java 不同，在 C++ 中，我们只需要将 `threadLoop()` 返回值设置为 `true`，当 `InputReaderThread` 线程启动后，该方法就会被循环调用，不用手动写 `while(true)`

`InputReaderThread` 线程启动时机我们后面会讲到，先回过头来看 `InputReader#loopOnce()` 的工作

```

/frameworks/native/services/inputflinger/InputReader.cpp
class InputReader {

    // 读取事件，解析事件
    void InputReader::loopOnce() {
        int count = mEventHub->getEvents(); // 读消息，有消息返回，没消息阻塞到 epoll()。E
        if(count) processEventsLocked();//解析原始事件、提交到队列等待分发
    }
}

```

关键代码只有两行

第一行是调用了 `EventHub#getEvents()` 获取事件，我们在上一节已经介绍过这个方法了，有消息返回，没消息阻塞到 `epoll()`

并且，因为执行事件分发需要时间，在上一次分发还没有执行结束之前，如果多个设备都发生了事件，或者一个设备发送了多次事件，都会造成数据的积压，`getEvents()` 返回的事件数量可能是多条

第二行代码 `processEventsLocked()` 是获取到事件以后，对原始事件进行解析，然后提交到队列等待分发

```

/frameworks/native/services/inputflinger/InputReader.cpp
class InputReader {

    void InputReader::processEventsLocked(rawEvents, count) {
        // 遍历所有事件，解析、分发
        for (const RawEvent* rawEvent = rawEvents; count;) {
            int32_t type = rawEvent->type; // 获取事件类型
            switch (type){ // 源码不包含此 switch 逻辑，这是 InputDevice 中的内容，为了方便
                case EV_KEY; // 按键类型的事件。能够上报这类事件的设备有键盘、鼠标、手柄、手
                case EV_ABS; // 绝对坐标类型的事件。这类事件描述了在空间中的一个点，触控板、角
                case EV_REL; // 相对坐标类型的事件。这类事件描述了
                case EV_SW; // 开关类型的事件。这类事件描述了若干I
            }
        }
    }
}

```



```

// 我们只专注 touch 触摸事件
dispatchTouches(when, policyFlags);
}
}

void dispatchTouches(){
    // 判断是否只是单指事件，或是多指触摸等等
    // 解析完成后，调用 dispatchMotion() 分发
    dispatchMotion();
}

void dispatchMotion(){
    // 最终生成 NotifyMotionArgs 结构，交给 InputDispatcher 执行最后的分发
    NotifyMotionArgs args;
    InputDispatcher::notifyMotion(args); // 提交到 InputDispatcher
}
}

```

ps: 为了方便理解，我把其他分支整合到主方法来，中间还省略了许多代码。所以建议不要对照源码看这篇文章，不然你可能会因为找不到某个方法回来骂我的~

解析事件的业务逻辑几乎都放在了 `processEventsLocked()` 方法

在 `processEventsLocked()` 方法中，首先遍历事件集合，根据不同的事件类型，调用不同的解析方法

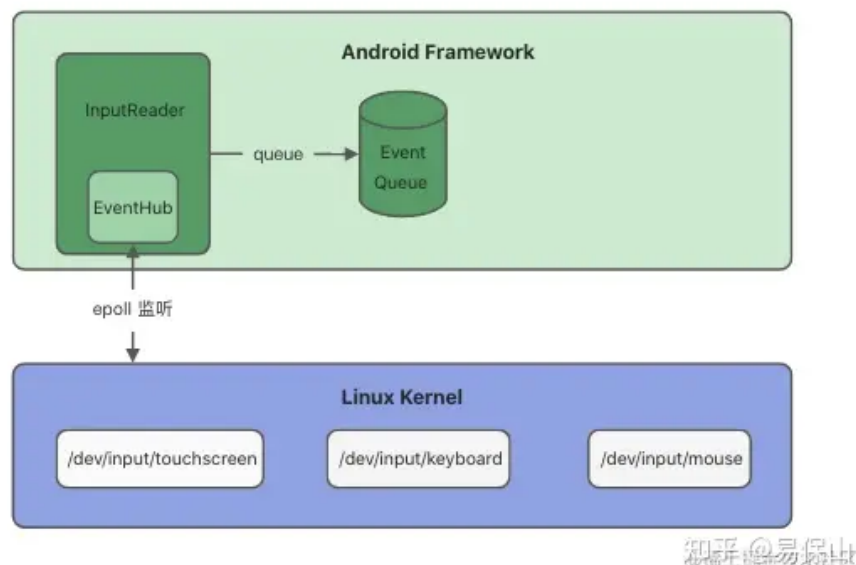
我们只专注 touch 触摸事件，也就是 `dispatchTouches()`

在 `dispatchTouches()` 方法中，根据触摸点信息来决定，是单指还是多指事件

解析完成后，调用 `dispatchMotion()` 生成 `NotifyMotionArgs` 对象，通知 `InputDispatcher#notifyMotion()` 方法

当 `InputDispatcher#notifyMotion()` 方法被调用，也就代表着，一次读取事件，解析事件的过程就结束了

接下来就看事件是怎么分发的了，`InputReader` 整个过程如下图



android_graphic_v5_native_2.png

图片来源：自己画的

接下来的 InputDispatcher 应该不用再介绍了，就是用来分发输入事件的

上一小节结束时放图片，左边的 InputReader 向中间的 EventQueue 队列提交了事件消息，我先来解释一下这是什么时候发生的



```
/frameworks/native/services/inputflinger/InputDispatcher.cpp
class InputDispatcher {

    void notifyMotion(const NotifyMotionArgs* args) {
        MotionEntry* newEntry = new MotionEntry(args); // 封装成 entry
        enqueueInboundEventLocked(newEntry); // 入列一个节点，等待分发被执行，逻辑在 dispatch
    }

}


```

呐，看代码

上一小节最后一行调用的 InputDispatcher#notifyMotion() 方法，作用就是把事件消息提交到消息队列，等待分发

好，尾收完了我们继续来看 InputDispatcher 类

到了 InputDispatcher 这里，原始的输入事件已经被封装成 Key 、 Motion 等对象，InputDispatcher 的任务是：找到合适的 Window，并把数据传递过去

```
/frameworks/native/services/inputflinger/InputDispatcher.cpp
class InputDispatcher {

    class InputDispatcherThread : Thread {

        bool InputDispatcherThread::threadLoop() {
            mDispatcher->dispatchOnce();
            return true;
        }

    }

    void dispatchOnce() {
        if(!queue.isEmpty()) dispatchOnceInnerLocked(); // 有消息就执行分发
    }

}


```

首先来看 InputDispatcher 的 dispatchOnce() 主方法，和 InputReader 设计思路相同，InputDispatcher 也是内部有个线程类，然后循环调用 dispatchOnce() 执行分发

如果消息队列中有消息，调用 dispatchOnceInnerLocked() 执行分发

```
/frameworks/native/services/inputflinger/InputDispatcher.cpp
class InputDispatcher {

    void dispatchOnceInnerLocked() {
        mPendingEvent = queue.dequeue(); // 从派发队列取出一个事件，简略写法
        switch (mPendingEvent->type) { // 判断消息的类型：配置更改、插拔消息、key事件、触摸
            case EventEntry::TYPE_MOTION:
                dispatchMotionLocked(); // 我们只专注 触摸事件
            }
    }

    void dispatchMotionLocked() {
        int32_t injectionResult = findTouchedWindowTargetsLocked(); // 为 Motion 事件寻
        if (injectionResult) dispatchEventLocked(); // 如果成功
    }

}


```

}

负责分发的 `dispatchOnceInnerLocked()` 方法需要处理不同输入类型的事件，我们这里还是只关注触摸消息

如果是触摸事件，那么触摸消息的分发是由 `dispatchMotionLocked()` 方法来完成的

`dispatchMotionLocked()` 触摸消息的分发，分为两步执行：

第一步，为 Motion 事件寻找合适的目标窗口，这个任务交给 `findTouchedWindowTargetsLocked()` 函数去完成

第二步，如果成功地找到了可以接收事件的目标窗口，交给 `dispatchEventLocked()` 函数完成实际的派发工作

接下来我们来看，`findTouchedWindowTargetsLocked()` 和 `dispatchEventLocked()` 这两个方法的实现

```
/frameworks/native/services/inputflinger/InputDispatcher.cpp
class InputDispatcher {

    int findTouchedWindowTargetsLocked(){
        size_t numWindows = mWindowHandles.size(); // 获取窗口集合
        for (size_t i = 0; i < numWindows; i++){//从前向后遍历所有的window以找出触摸的win
        }

        // 合适的目标窗口被确定下来之后，便可以开始将实际的事件发送给窗口了
        void dispatchEventLocked(Vector<InputTarget>& inputTargets) {
            InputChannel channel = inputTarget.inputChannel;//删减过的流程
            channel->sendMessage(&msg);//给能够被触摸的window发送跨进程消息
        }

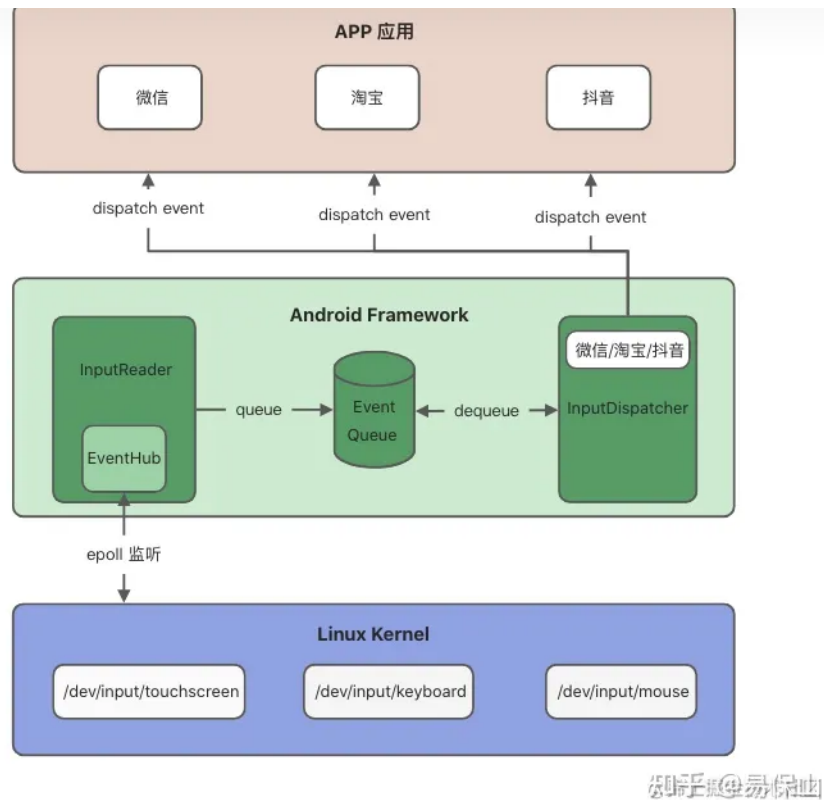
    }
}
```

`findTouchedWindowTargetsLocked()` 方法的主要逻辑，是根据窗口的点击区域与事件发生的坐标点选取合适的目标窗口

代码稍微有点长，这里就不展开讨论了，感兴趣的朋友可以阅读 [《Window Touchable Region》](#) 这篇文章

再来看负责分发的 `dispatchEventLocked()` 方法，代码实现也很简单，根据窗口查找该窗口对应的 `channel`，然后通过 `channel` 跨进程把事件传递到 APP

到这里，`InputDispatcher` 所有的分发工作就全部结束了，整个过程如下图



android_graphic_v5_native_3.png

图片来源：自己画的

哎，还没完，回头来看负责分发触摸消息的 `InputDispatcher#dispatchMotionLocked()` 函数

```
/frameworks/native/services/inputflinger/InputDispatcher.cpp
class InputDispatcher {

    void dispatchMotionLocked() {
        int32_t injectionResult = findTouchedWindowTargetsLocked(); // 为 Motion 事件寻
        if (injectionResult) dispatchEventLocked(); // 如果成功地找到了可以接收事件的目标
    }

    int findTouchedWindowTargetsLocked(){
        size_t numWindows = mWindowHandles.size(); // 获取窗口集合
        for (size_t i = 0; i < numWindows; i++)
            ...
    }

    void dispatchEventLocked(Vector<InputTarget>& inputTargets) {
        InputChannel channel = inputTarget.inputChannel;
        channel->sendMessage(&msg); // 给能够被触摸的window发送跨进程消息
    }

}
```

再看一遍源码，我们来思考两个问题

1. 负责查找窗口的 `findTouchedWindowTargetsLocked()` 方法中，`mWindowHandles` 所持有的窗口集合，是从哪里来的？
2. 负责通信的 `dispatchEventLocked()` 方法中，`InputChannel` 是什么？IMS 是什么时候和 APP 建立通信的？

回想一下，在 `InputDispatcher` 之前，不管是 `EventHub`，还是 `InputReader`，它们都是和 Linux 内核打交道，读取事件、解析事件啥的

`findTouchedWindowTargetsLocked()` 的窗口集合从哪里来的?

`dispatchEventLocked()` 又是如何把触摸事件通知到 APP 进程的?

带着这两个疑问, 我们开始进入 `InputManagerService` 的启动流程环节, 这里面一定有我们要寻找的答案

启动 `InputManagerService`

我们知道, `InputManagerService` 作为运行在 `SystemServer` 进程中的服务, 启动顺序是排在 `Zygote` 进程之后的

Java 虚拟机初始化完成后, 再由 `Zygote` 进程 fork 而来

本小节我们将要来跟踪 `InputManagerService` 的启动流程, 中间会涉及到一些没那么重要的类 (重要的都在上面介绍过了)

比如, 同为在 `/frameworks/native/services/inputflinger` 包下面的 `InputManager` 类, 我们在介绍 native 层成员的时候就没有带上它

因为 `InputManager` 只是负责管理 `reader` 和 `dispatcher` 线程, 没有业务逻辑, 不是很重要

在整个 IMS 的启动流程中, 我们时刻要谨记, 本节的重点是:

- 了解 `InputManagerService` 大致的启动流程
- 了解 `InputDispatcher` 的窗口集合从哪里来, 以及 IMS 如何建立跟 APP 通信的?

搞清楚这两个关键问题, `IMS` 启动流程这 part 就可以翻篇了, 千万别被陷入到源码中, 很难出来的~

1、IMS 窗口集合从哪里来?

在前两节我们了解到, `InputManagerService` 可以分为 native、jni、Java 三层

这三层大致的启动顺序是: 先从 Java 层的初始化开始, 再调用到 jni 层, 由 jni 拉起 native 的各个类, 进而完成 native 部分的初始化, 最后返回到 Java 层, `IMS` 开始为各个 APP 提供服务

我们先来看 Java 层的初始化工作

```
/frameworks/base/services/java/com/android/server/SystemServer.java
class SystemServer {

    private void startOtherServices() {
        inputManager = new InputManagerService(context); // 创建 IMS 对象
        ...
        //将 InputMonitor 对象保存到 IMS 对象
        inputManager.setWindowManagerCallbacks(wm.getInputMonitor());
        inputManager.start();
    }
}
```

在 `SystemServer#startOtherServices()` 启动服务的方法中, 首先创建了 `InputManagerService` 对象

然后, 将 `WindowManagerService` 中的 `InputMonitor` 对象保存到 `IMS` 中

这是 `InputMonitor` 类是干嘛的? 之前好像没见过

简单来说, 它是连接 `WMS` 和 `IMS` 的枢纽。WMS 通过 `InputMonitor.java` 挂有了 `IMS` 的引用, 当窗口信息发生变化后, 通过 `InputMonitor#updateInputWindows`

我们本小节的目标之一，是为了搞清楚 InputDispatcher 持有的窗口集合从哪里来？

现在，答案有了，是 WMS 通过调用 InputMonitor#updateInputWindowsLw() 函数，最终同步到 InputDispatcher 类中

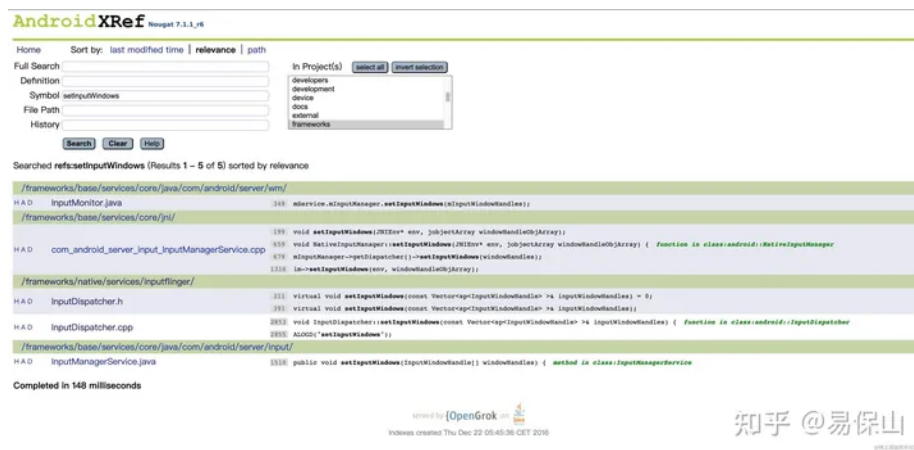
简单说一下查找过程

先回到 InputDispatcher 源码，搜索 mWindowHandles 关键字，很容易就能发现了
mWindowHandles 集合是在 setInputWindows() 函数中被赋值

```
/frameworks/native/services/inputflinger/InputDispatcher.cpp
class InputDispatcher {

    void InputDispatcher::setInputWindows(inputWindowHandles) {
        mWindowHandles = inputWindowHandles;
    }
}
```

那么 setInputWindows() 是谁调用的？接着搜索 Framework，结果如图



android_graphic_v5_method_setInputWindows.jpg

图片来源：aosp.xref.com/android-7.1.2

我们发现，Java 层的 IMS 也有一个 setInputWindows() 方法，并通过 JNI 指向了 native 中的 InputDispatcher#setInputWindows()

最终的调用动作就发生在 InputMonitor 类的 updateInputWindowsLw() 方法中！

好了，InputDispatcher 持有的窗口集合来源，这个疑问已经解决了。我们继续来跟启动流程

```
/frameworks/base/services/java/com/android/server/SystemServer.java
class SystemServer {

    private void startOtherServices() {
        inputManager = new InputManagerService(context);
        inputManager.start();
    }
}

/frameworks/base/services/core/java/com/android/server/input/InputManagerService.java
class InputManagerService {
    // 【step 1.0】初始化流程
    public InputManagerService(Context context) {
        mPtr = nativeInit(this, mContext, mHandler.getLooper().getQueue());
        LocalServices.addService(InputManagerInternal.class, new LocalService());
    }
}
```

```

        nativeStart(mPtr); // 详见 【2.1】
        Watchdog.getInstance().addMonitor(this);
    }
}

```

在 SystemServer 创建完 InputManagerService 对象后，紧接着就调用了 `inputManager#start()` 启动了服务

所以我们需要把启动流程分为两个步骤来看，一个是初始化流程做了什么，第二个才是启动流程

在 InputManagerService 的构造函数中，调用了 `nativeInit()` 执行了初始化工作，这是个 jni 方法，我们一起去看看源码实现

2、IMS 的初始化工作

```

/frameworks/base/services/core/jni/com_android_server_input_InputManagerService.cpp
class NativeInputManager {

    static jlong nativeInit(env, jclass, serviceObj, contextObj, messageQueueObj) {
        NativeInputManager* im = new NativeInputManager(contextObj, serviceObj, messageQueueObj);
    }

    NativeInputManager::NativeInputManager(contextObj, serviceObj, messageQueueObj) {
        sp<EventHub> eventHub = new EventHub(); // 创建 EventHub 对象
        mInputManager = new InputManager(eventHub, this, this); // 创建 InputManager 对象
    }

}

```

`nativeInit()` 方法中创建了 NativeInputManager 对象

在 NativeInputManager 的构造函数中，又创建了两个我们熟悉的对象，EventHub 和 InputManager

EventHub 在前面已经介绍过了，负责监听事件变化，并对外提供获取事件变化的接口

InputManager 也提到过，负责创建 Reader 和 Dispatcher 两线程，没什么逻辑

```

/frameworks/native/services/inputflinger/EventHub.cpp
class EventHub {

    EventHub::EventHub(void) {
        mEpollFd = epoll_create(EPOLL_SIZE_HINT); // 创建 epoll，用于监听设备文件是否有可读事件
        mInotifyFd = inotify_init(); // 创建 inotify，用于监听文件系统是否变化，有变化说明
    }

}

/frameworks/native/services/inputflinger/InputManager.cpp
class InputManager {

    InputManager::InputManager(eventHub, readerPolicy, dispatcherPolicy) {
        mDispatcher = new InputDispatcher(dispatcherPolicy);
        mReader = new InputReader(eventHub, readerPolicy, mDispatcher);
        initialize();
    }

    void InputManager::initialize() {
        mReaderThread = new InputReaderThread(mReader); // 创建线程 "InputReader"
        mDispatcherThread = new InputDispatcherThread(mDispatcher); // 创建线程 "InputDispatcher"
    }

}

```


呐，你看

EventHub 只是创建了 mEpollFd 和 mINotifyFd 两个 fd 对象

InputManager 也只是创建了 InputReader 和 InputDispatcher 两个对象，然后创建了 InputReaderThread 和 InputDispatcherThread 两个线程

好了，现在 IMS 底层的三员大将：EventHub、InputReader、InputDispatcher 全部成功创建，IMS 类的初始化工作就全部结束了

3、IMS 的启动流程

我们把 SystemServer 的代码再拿出来看看，看看接下来应该做什么

```
/frameworks/base/services/java/com/android/server/SystemServer.java
class SystemServer {

    private void startOtherServices() {
        inputManager.start();
    }
}

/frameworks/base/services/core/java/com/android/server/input/InputManagerService.java
class InputManagerService {

    public InputManagerService(Context context); // 初始化工作已完成

    public void start() {
        nativeStart(mPtr); // 启动服务
        Watchdog.getInstance().addMonitor(this);
    }
}
```

初始化工作完成以后，紧接着调用了 start() 方法启动了服务

start() 内部调用了 nativeStart() 方法，这又是个 jni 函数，我们继续向下跟

```
/frameworks/base/services/core/jni/com_android_server_input_InputManagerService.cpp
class NativeInputManager {

    static void nativeStart(env, jclass, ptr) {
        getInputManager()->start(); // 调用 InputManager 的 start() 方法
    }
}

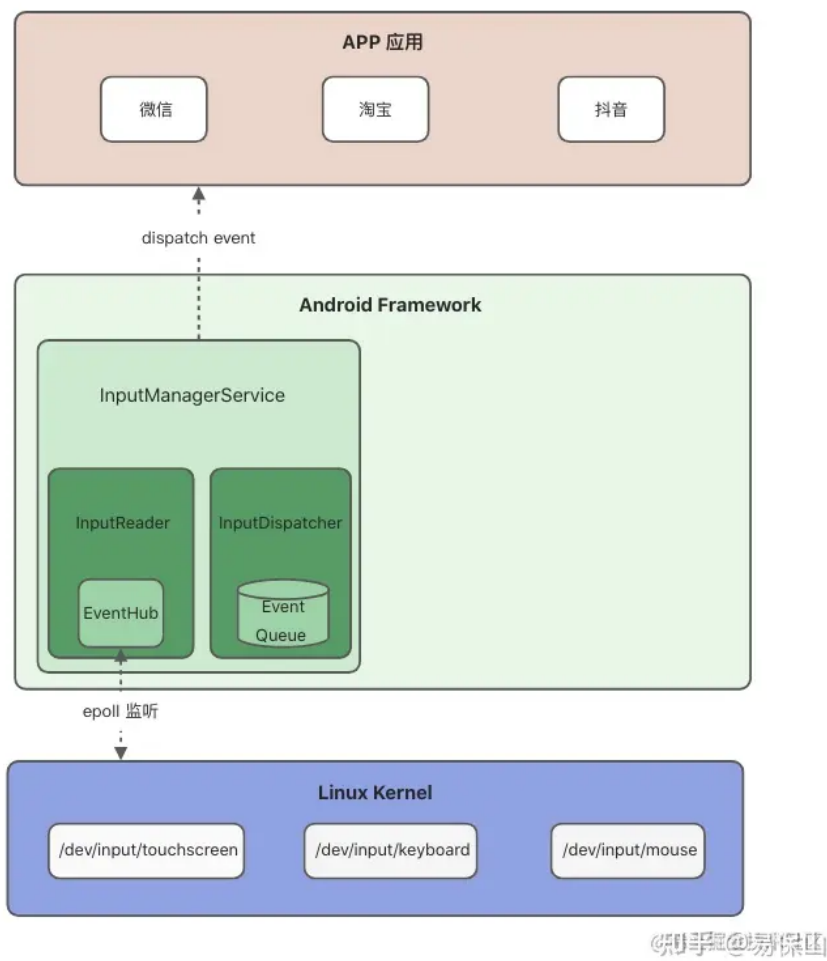
/frameworks/native/services/inputflinger/InputManager.cpp
class InputManager {

    status_t InputManager::start() {
        result = mDispatcherThread->run("InputDispatcher", PRIORITY_URGENT_DISPLAY); //
        result = mReaderThread->run("InputReader", PRIORITY_URGENT_DISPLAY); // 启动线程
        ...
        return OK;
    }
}
```

nativeStart() 内部调用了 InputManager#start()，启动了 InputReaderThread 和 InputDispatcher 线程

InputDispatcher 循环派发消息，一直从事件队列中取 Event 消息，派发给合适的窗口

到这里，IMS 的启动工作就全部结束了，整个过程如下图



android_graphic_v5_ims_process.png

图片来源：自己画的

IMS 启动流程稍微有那么一点点长，完整的启动分析我放在了 GitHub，点击[这里](#)跳转查看

启动 APP 进程

在上一小节 InputManagerService 的启动流程中，我们解决了 “InputDispatcher 的窗口集合从哪里来” 的问题

现在还剩下 “APP 是如何和 IMS 建立通信的 ” 这个问题还没有解决

系统服务全部准备就绪后，接下来是 APP 应用的启动流程，在跟踪启动应用进程的过程中，我们会找到 “APP 是如何和 IMS 建立通信的 ” 这个问题的答案

1、为 Activity 分配窗口

APP 的启动过程我们应该都多少有点了解，和 AMS 通信怎么创建进程这部分我们就跳过了，本篇重点是触摸事件，我们直接快进到为 Activity 设置视图，分配窗口这部分内容

```
/frameworks/base/core/java/android/view/ViewRootImpl.java
class ViewRootImpl {
```

```
void setView(View view){
    mInputChannel = new InputChannel(); // 创建了空的 InputChannel，下面代码将会生成
    Session.addToDisplay(view,mInputChannel); // 向wms添加窗口

    if(mInputChannel!=null) mInputEventReceiver = new WindowInputEventReceiver(mIn
}

}
```

我们在 Activity 设置的视图文件，最终会调用到 ViewRootImpl#setView() 方法

在 setView() 方法中，一共有三行关键代码

1. 创建了属于该视图的 InputChannel 空对象，先不用管
2. 向 WMS 添加窗口，并将刚刚创建的 InputChannel 对象一并传递过去，重要逻辑
3. 创建了 WindowInputEventReceiver，将该视图的 InputChannel 保存起来，也不用管

在这三行代码中，第2行是关键代码，第1行和第3行，以现有的信息没办法解释它们内部做了什么，等到后面有机会在介绍

好，我们来看第2行代码发生了什么

```
/frameworks/base/services/core/java/com/android/server/wm/Session.java
class Session {

    void addToDisplay(InputChannel inputChannel){
        WindowManagerService.addWindow(inputChannel);
    }

}

/frameworks/base/services/core/java/com/android/server/wm/WindowManagerService.java
class WindowManagerService {

    int addWindow(InputChannel outInputChannel){
        WindowState win = new WindowState(); // 首次添加视图时创建，用于描述一个window
        win.openInputChannel(outInputChannel); // 创建通信的关键代码，打开一对已连接的 soc
    }

}

}
```

Session#addToDisplay() 方法只是做了转发，实际创建窗口的工作还是由
WindowManagerService#addWindow() 来完成的

在 addWindow() 方法中，首先创建了 WindowState 对象，用于描述该窗口信息

随后调用了 WindowState 对象中的 openInputChannel() 方法，它是创建通信的关键代码，内部是创建了一对已连接的 socket

我们来重点关注 WindowState#openInputChannel() 方法

```
/frameworks/base/services/core/java/com/android/server/wm/WindowState.java
class WindowState {

    InputChannel mInputChannel;
    InputChannel mClientChannel;

    void openInputChannel(InputChannel outInputChannel) {
        InputChannel[] inputChannels = InputChannel.openInputC
        // 这是一对已连接的管道，将 socket 两端分别保存到服务端和客
```

```

// 1. Client 端 InputChannel 调用 transferTo() 方法传给 ViewRootImpl 的 mInputChannel
mClientChannel.transferTo(outInputChannel);
// 2. Server 端 InputChannel 存在 WindowState 的 mInputChannel 变量
InputManagerService.registerInputChannel(mInputChannel);
}
}

/frameworks/native/libs/input/InputTransport.cpp
status_t InputChannel::openInputChannelPair(name, outServerChannel, outClientChannel) {
    int sockets[2] = socketpair(sockets);
    serverChannelName.append(" (server)");
    outServerChannel = new InputChannel(serverChannelName, sockets[0]);
    clientChannelName.append(" (client)");
    outClientChannel = new InputChannel(clientChannelName, sockets[1]);
    return OK;
}

```

`WindowState#openInputChannel()` 中，首先调用了 `InputChannel#openInputChannelPair()` 函数创建两个 `InputChannel`，其内部调用 Linux 的 `socketpair()` 函数

`socketpair()` 是 Linux 提供了一种进程间通信的方式，跟 `pipe()` 函数是类似的。但 `pipe()` 创建的匿名管道是半双工的，而 `socketpair()` 可以认为是创建一个全双工的管道：

我向我持有的 `socket` 中写数据，你在你持有的 `socket` 中能够读到数据，反过来也一样，即两端都可以对自己持有的 `socket` 进行读写

在触摸事件的传递中，Google 团队使用了 `socketpair()` 创建一对已连接的 `socket`，用于 IMS 和 APP 间跨进程通信

其中，IMS 会持有名为 `server` 的一端（`sockets[0]`）进行读写；APP 持有名为 `client` 的一端（`sockets[1]`）进行读写

提个醒，我们现在看的代码是：设置 Activity 视图时，经过 `binder` 通信后，跑在了 `system_server` 进程中的 WMS 服务里

因此，我们接下来的任务，是把 `sockets[0]` 的 `server` 端，传递给 IMS，让触摸事件发生后，IMS 能够通知到 APP

然后，把 `sockets[1]` 的 `client` 端通过 `binder` 跨进程回传给 APP 进程保存，让 APP 能够接收到来自 IMS 的消息

好，开始干活

2、sockets[1] 回传给 APP

源码里是先将 Client 的回传给 APP 进程，那我们就按照源码的顺序来看，先把属于 APP 进程的 `socket` / `InputChannel` 回传过去

```

/frameworks/base/services/core/java/com/android/server/wm/WindowState.java
class WindowState {

    void openInputChannel(InputChannel outInputChannel) {
        // 1. Client 端 InputChannel 调用 transferTo() 方法传给 ViewRootImpl 的 mInputChannel
        mClientChannel.transferTo(outInputChannel);
        // 2. Server 端 InputChannel 存在 WindowState 的 mInputChannel 变量
        InputManagerService.registerInputChannel(mInputChannel);
    }
}

/frameworks/base/core/java/android/view/ViewRootImpl.java
class ViewRootImpl {

```

```

void setView(View view){
    mInputChannel = new InputChannel(); // 创建了空的 InputChannel
    try {
        Session.addToDisplay(view,mInputChannel);
    } catch (RemoteException e) {
        mInputChannel = null;
    }
    ...
    if(mInputChannel != null ) mInputEventReceiver = new InputEventReceiver(mInput
}

}

```

看代码，`Session#addToDisplay()` 是将我们设置的视图，和刚刚创建的空的 `mInputChannel` 传递到 `WindowManagerService`

如果 WMS 成功添加了视图，没有发生异常，表示属于 APP 端的 `socket` / `InputChannel` 已经创建成功并通过 `binder` 跨进程传递回来了，此时 `mInputChannel` 变量就不是刚刚创建的空对象了，里面已经包含了 APP 端的 `socket`

监听这个 `socket`，我们可以收到来自 IMS 的消息；往这个 `socket` 写数据，IMS 也可以收到我们发送的消息，美滋滋

如果 WMS 添加视图失败了，会抛出 `RemoteException` 远程连接异常，`mInputChannel` 变量将被清空。

总之，只要 `mInputChannel` 变量不为空，就表示属于 APP 端的 `socket` 已经传回来了，WMS 和 APP 中间的传递过程我们先不管

好，现在 APP 端的 `InputChannel` 已经回传成功，下一步的代码是创建了 `InputEventReceiver` 对象，并将 APP 端的 `InputChannel` 和 APP 端的 `Looper` 一同传递过去

一起来看 `InputEventReceiver` 的代码

```

/frameworks/base/core/java/android/view/InputEventReceiver.java
class InputEventReceiver {

    public InputEventReceiver(InputChannel inputChannel, Looper looper) {
        mReceiverPtr = nativeInit(new WeakReference<InputEventReceiver>(this),inputCha
    }

}

```

java 层的 `InputEventReceiver` 只是个空壳子，实际的实现在 native 层，继续向下跟

```

/frameworks/base/core/jni/android_view_InputEventReceiver.cpp
class NativeInputEventReceiver {

    static jlong nativeInit(env, clazz, receiverWeak, inputChannelObj, messageQueueOb
        int fd = inputChannelObj->getFd();
        messageQueueObj->getLooper()->addFd(fd, 0, events, this, NULL);
        ...
    }

}

```

代码我又合并过，关键代码就两行

第二行是利用 `messageQueueObj` 对象获取里面的 `Looper`，把上一步拿到的 `socketfd` 丢进去监听

代码虽然不多，但理解这两行代码，需要对 `Handler` 机制比较熟悉，包括 `native` 层

我来简单解释一下：

在《[Android组件系列：再谈Handler机制（Native篇）](#)》这篇文章中，我们了解到：在 `native` 层同样拥有一套 `Looper` 机制。这套 `Looper` 不但可以处理 `native` 层消息，还支持监听 自定义 `fd`，这是本小节的重点

Java 层的 `MessageQueue` 在初始化时，会调用 `nativeInit()` 方法，同步创建 `naive` 层的 `NativeMessageQueue` 对象，并将返回的 `native` 引用保存到 `mPtr` 变量中

注意看，我们现在跟踪的 `NativeInputEventReceiver` 构造函数的入参，传递过来的参数一个是 `InputChannel`，里面包含了属于 APP 端的 `socket`，还有一个参数是来自 Java 层的 `MessageQueue` 对象

那么，`NativeInputEventReceiver#nativeInit()` 方法里完成的事情是：利用 Java `MessageQueue` 的 `mPtr` 变量持有的 `NativeMessageQueue` 对象，找到 `native Looper`

然后，利用 `native` 层的 `Looper` 对象来监听一同传递过来的 `InputChannel` 中的 `socketfd`。

这一段听起来可能会有点绕，暂时没理解某个点问题也不大，我们只需要知道，在设置视图的时候，顺便创建了和 `IMS` 通信的通道就行了

好了，APP 端的 `socket (*sockets[1] *)` 回传工作算是结束了，接下来我们看怎么把 `sockets[0]` 传递给 `IMS`

3、sockets[0] 传递到 IMS

`IMS` 端的 `socket` 最终是由 `InputDispatcher` 来保管，因为分发事件时，是 `InputDispatcher` 直接使用 `socket` 通知 APP 的

所以，我们本小节的目标是：搞清楚 `IMS` 端的 `socket` 是如何传递到 `InputDispatcher` 类中的？

`IMS` 传递的起点，依旧是在 `WindowState#openInputChannel()` 方法中：

```
/frameworks/base/services/core/java/com/android/server/wm/WindowState.java
class WindowState {

    void openInputChannel(InputChannel outInputChannel) {
        // 1. Client 端 InputChanel 调用 transferTo() 方法传给 ViewRootImpl 的 mInputCh
        mClientChannel.transferTo(outInputChannel);
        // 2. Server 端 InputChannel 存在 WindowState 的 mInputChannel 变量
        InputManagerService.registerInputChannel(mInputChannel);
    }
}

/frameworks/base/services/core/java/com/android/server/input/InputManagerService.java
class InputManagerService {

    void registerInputChannel(){
        nativeRegisterInputChannel(mPtr, inputChannel, inputWindowHandle, false);
    }
}
```

`InputManagerService#registerInputChannel()` 又是一个空壳，具体实现在 `native`，继续向下跟

```

void nativeRegisterInputChannel(){
    InputManager->getDispatcher()->registerInputChannel(inputChannel, inputWindowH
}
}

/frameworks/native/services/inputflinger/InputDispatcher.cpp
class InputDispatcher {

    status_t InputDispatcher::registerInputChannel(inputChannel,inputWindowHandle,moni
    // 将代表窗口通信的 inputChannel , 以及代表窗口信息的 inputWindowHandle 封装成 Conn
    sp<Connection> connection = new Connection(inputChannel, inputWindowHandle, mo
    mConnectionsByFd.add(fd, connection);
    ... // 省略 IMS 监听来自 client 的代码, 这部分是处理 ANR 的逻辑
}
}

```

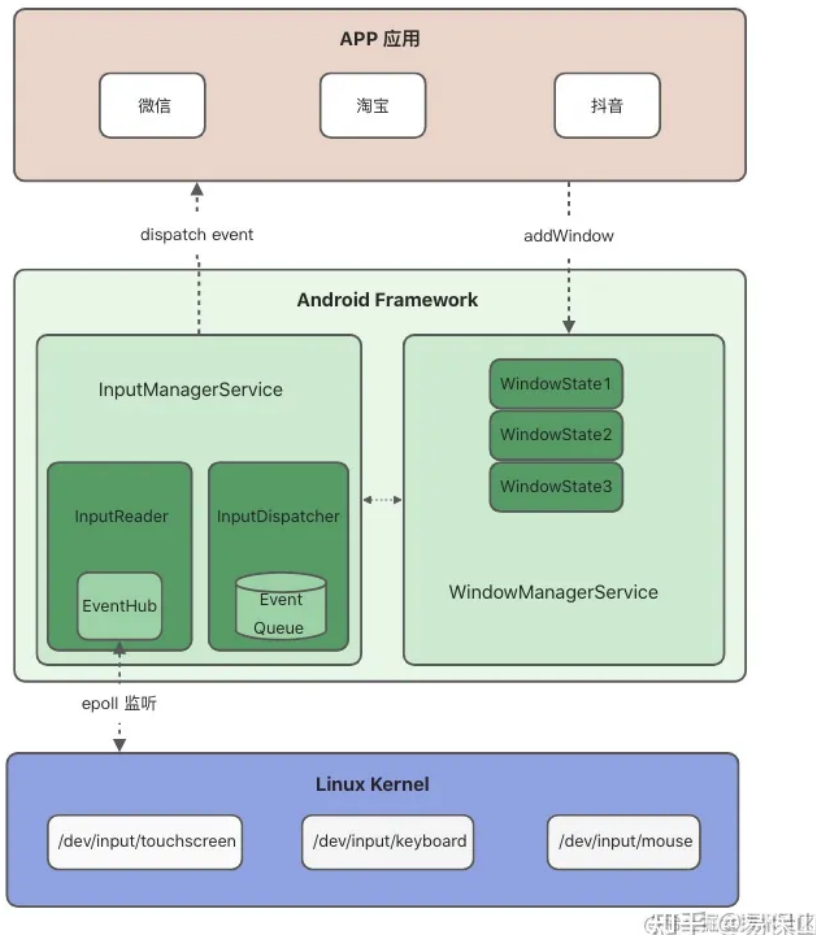
NativeInputManager#nativeRegisterInputChannel() 随手又是一个转发

在 InputDispatcher#registerInputChannel() 方法中, 经过一番长途跋涉, 最终将 socket 注册到 InputDispatcher, 一起被注册过来的还有代表该窗口信息的 inputWindowHandle

代表窗口通信的 inputChannel , 以及代表窗口信息的 inputWindowHandle 被封装成 Connection 对象, 保存到 mConnectionsByFd 集合中

至此, IMS 端的 socket 传递工作结束, APP 和 IMS 可以愉快的通信了

整个过程如下



触摸事件到达 ViewRootImpl

好了，万事俱备，只欠东风

现在 APP 进程起来了，APP 和 IMS 也成功建立了通信，接下来我们按下屏幕，看看触摸事件是怎么到达我们熟悉的 View 的

```
/frameworks/base/core/jni/android_view_InputEventReceiver.cpp
class NativeInputEventReceiver {

    static jlong nativeInit(env, clazz, receiverWeak, inputChannelObj, messageQueueObj) {
        int fd = inputChannelObj->getFd();
        messageQueueObj->getLooper()->addFd(fd, 0, events, this, NULL);
        ...
    }
}
```

在介绍 APP 端的 socket 回传一节中，最后一步停在了

NativeInputEventReceiver#nativeInit() 方法，利用 Looper 监听了 socketfd

现在我们来聊聊：当 socketfd 有消息时，APP 端会怎么处理，事件是怎么分发到根 View 的？

```
/frameworks/base/core/jni/android_view_InputEventReceiver.cpp
class NativeInputEventReceiver {

    int NativeInputEventReceiver::handleEvent( receiveFd, events, data) {
        switch (type) { // 简略写法，这是 consumeEvents() 方法中的逻辑
            case AINPUT_EVENT_TYPE_KEY: {
                inputEventObj = factory->createKeyEvent(); // 转换为按键类型事件 KeyEvent
            }
            case AINPUT_EVENT_TYPE_MOTION: {
                inputEventObj = factory->createMotionEvent(); // 转换为触摸类型事件 MotionEvent
            }
        }
        env->CallVoidMethod(receiverObj.get(),dispatchInputEvent, seq, inputEventObj);
    }
}
```

APP 端的 socketfd 发生事件变化时，Looper 根据指定的回调地址，调用到

NativeInputEventReceiver#handleEvent() 方法

在 NativeInputEventReceiver#handleEvent() 方法中，根据事件类型，创建不同的事件对象

最后，利用 CallVoidMethod 回调 Java 方法，将事件传递到 Java 层的

InputEventReceiver#dispatchInputEvent() 方法

```
/frameworks/base/core/java/android/view/InputEventReceiver.java
abstract class InputEventReceiver {

    // Called from native code.
    private void dispatchInputEvent(int seq, InputEvent event) {
        onInputEvent(event);
    }
}

/frameworks/base/core/java/android/view/ViewRootImpl.java
class ViewRootImpl extends InputEventReceiver {

    class WindowInputEventReceiver extends InputEventReceiver
```

```

    public void onInputEvent(InputEvent event) {
        enqueueInputEvent(event, this, 0, true);
    }
}

void enqueueInputEvent(InputEvent event, InputEventReceiver receiver, int flags, boolean
...
// 执行消息入列以后，接着还有一个比较复杂的流水线过程，我们这里先不关心，直接来看 process
// input 消息到达 ViewRootImpl 后，Google 使用责任链的模式，将输入事件拆分为 KeyEvent
//
if(event.getType == input) processPointerEvent(event);
if(event.getType == key) processKeyEvent(event);
}

// 责任链模式，每个 InputStage 负责不同的功能，链中的某个 InputStage 的结果会影响对下一节
// 在 ViewRootImpl#setView() 函数中指定责任链的前后顺序，这里不展开讨论，请查看参考资料列表
class InputStage {

    // 在 ViewRootImpl 中有好几个同名 processPointerEvent() 方法，eventTarget 通常是
    private int processPointerEvent(QueuedInputEvent q) {
        MotionEvent event = (MotionEvent)q.mEvent;
        final View eventTarget = mView; // 通常是 DecorView
        boolean handled = eventTarget.dispatchPointerEvent(event);
        ...
        return handled ? FINISH_HANDLED : FORWARD;
    }

    private int processKeyEvent(QueuedInputEvent q) {
        KeyEvent event = (KeyEvent)q.mEvent;
        mView.dispatchKeyEvent(event);
        final View eventTarget = mView; // 通常是 DecorView
        boolean handled = eventTarget.dispatchPointerEvent(event);
        ...
        return handled ? FINISH_HANDLED : FORWARD;
    }

}

}
}

```

`InputEventReceiver` 是抽象类，在 `dispatchInputEvent()` 方法中回调 `onInputEvent()` 方法。

而 `ViewRootImpl` 的内部类 `WindowInputEventReceiver` 实现了 `InputEventReceiver` 类，并且在 `onInputEvent()` 内部又调用了 `enqueueInputEvent()` 入列输入消息。

所以，接下来的分发逻辑全部都发生在 `ViewRootImpl#enqueueInputEvent()` 方法中。

胜利的曙光就在前方，继续冲。

```

/frameworks/base/core/java/android/view/ViewRootImpl.java
class ViewRootImpl extends InputEventReceiver {

    void enqueueInputEvent(InputEvent event, InputEventReceiver receiver, int flags, boolean
        if(event.getType == input) processPointerEvent(event);
        if(event.getType == key) processKeyEvent(event);
    }

    class InputStage {

        private int processPointerEvent(QueuedInputEvent q) {
            MotionEvent event = (MotionEvent)q.mEvent;
            final View eventTarget = mView; // 通常是 DecorView
            boolean handled = eventTarget.dispatchPointerEvent
            ...
        }
    }
}

```

```
private int processKeyEvent(QueuedInputEvent q);// 处理 key 事件，忽略
}

}
```

input 消息到达 ViewRootImpl 后，将输入事件拆分为 KeyEvent 和 MotionEvent 两种类型，做进一步的处理

Google 团队使用了 责任链模式 来处理事件消息，每个 InputStage 负责不同的功能，链中的某个 InputStage 的结果会影响对下一节点的执行，或停止分发等

ViewRootImpl#setView() 函数中指定了责任链执行的前后顺序，我们这里不展开讨论，感兴趣的同学可以查看参考资料列表中[《这一次，带你彻底弄懂 Android 事件分发机制\(外/内层责任链\)》](#)

为了省事，我们直接看 processPointerEvent() 处理触摸事件的方法

在 processPointerEvent() 方法中，先是将 InputEvent 强转为 MotionEvent，然后，调用 mView 的 dispatchPointerEvent() 方法执行分发

触摸事件到达 DecorView

了解 Window 创建流程的朋友肯定知道，mView 就是 DecorView，那这里其实调用的是 DecorView#dispatchPointerEvent() 执行分发，接着来跟踪

```
/frameworks/base/core/java/android/view/View.java
class View {

    public final boolean dispatchPointerEvent(MotionEvent event) {
        if (event.isTouchEvent()) {
            return dispatchTouchEvent(event);
        } else {
            return dispatchGenericMotionEvent(event);
        }
    }
}
```

DecorView 继承自 FrameLayout，FrameLayout 继承自 ViewGroup，ViewGroup 继承自 View

View 的 dispatchPointerEvent() 是 final 关键字修饰的，不允许子类重写

所以，调用 DecorView#dispatchPointerEvent() 实际的执行者是 View

在 View#dispatchPointerEvent() 方法中，首先判断是不是触摸事件，那肯定是啊

接着调用 dispatchTouchEvent() 方法执行分发

dispatchTouchEvent() 没有被 final 修饰，可以被重写，所以我们现在回到 DecorView 的 dispatchTouchEvent() 方法中

```
/frameworks/base/core/java/com/android/internal/policy/DecorView.java
class DecorView extends FrameLayout {

    @Override
    public boolean dispatchTouchEvent(MotionEvent ev) {
        Window.Callback cb = mWindow.getCallback(); // 给 Activity 和 Dialog 拦截事件的
        return cb != null ? cb.dispatchTouchEvent(ev) : super.dispatchTouchEvent(ev);
    }
}
```

在 `DecorView#dispatchTouchEvent()` 方法中，先是判断了 `mWindow` 持有的 `Callback` 是否为空

这里临时补充两个小细节

1. **Window.Callback** 是个接口，而 **Activity** 和 **Dialog** 都实现了这个接口
2. **DecorView** 持有的 `mWindow` 的赋值路径是这样的：`PhoneWindow#setContentView()` -> `installDecor()` -> `generateDecor()` -> `DecorView#setWindow()`，不展开讨论了

接着看代码，如果 `mWindow` 的 `Callback` 不为空，则优先调用 `Callback` 的 `dispatchTouchEvent()` 函数执行分发

我们以 **Activity** 来举例

```
/frameworks/base/core/java/android/app/Activity.java
class Activity {

    public boolean dispatchTouchEvent(MotionEvent ev) {
        if (getWindow().superDispatchTouchEvent(ev)) {
            return true;
        }
        return onTouchEvent(ev);
    }

    public boolean onTouchEvent(MotionEvent event) {
        ...
    }
}

/frameworks/base/core/java/com/android/internal/policy/PhoneWindow.java
class PhoneWindow {
    public boolean superDispatchTouchEvent(MotionEvent event) {
        return mDecor.superDispatchTouchEvent(event);
    }
}

/frameworks/base/core/java/com/android/internal/policy/DecorView.java
class DecorView extends FrameLayout {

    public boolean superDispatchTouchEvent(MotionEvent event) {
        return super.dispatchTouchEvent(event);
    }
}
```

Activity 的 `dispatchTouchEvent()` 方法被调用后，先调用了 `getWindow().superDispatchTouchEvent(ev)` 执行分发，没人处理再调用自身的 `onTouchEvent()` 方法

而 `getWindow().superDispatchTouchEvent()` 方法兜兜转转一圈，还是调用到 `DecorView#superDispatchTouchEvent()` 中

前面说过了，**DecorView** 继承自 **FrameLayout**，**FrameLayout** 是没有重写 `dispatchTouchEvent()` 方法的，**FrameLayout** 继承自 **ViewGroup**，**ViewGroup** 重写了 `dispatchTouchEvent()`

所以，最终执行分发的还是 `ViewGroup#dispatchTouchEvent()` 方法

合着绕了一圈，**Activity** 是啥也没做是吧？

ummmm~ 是的

好，现在触摸事件分发的起点到了我们非常熟悉的 `ViewGroup#dispatchTouchEvent()` 这里

接下来的一整章，我们来复习 View / ViewGroup 事件分发的流程



三、触摸事件的消费 (Application)

在之前的两节内容中，一个 `input` 事件一路从硬件驱动，成功的到达应用的 `DecorView`

我们接下来的任务是，把这个事件分发给某个具体的 View 或者是 ViewGroup

正文开始前，我们先来聊聊触摸事件的本体：`MotionEvent`

`MotionEvent` 表示一个 触摸事件，里面包含了事件的类型，触摸的位置信息等，对分发者来说，事件的类型非常重要，来简单认识一下

- `ACTION_DOWN`：按下屏幕
- `ACTION_MOVE`：手指滑动
- `ACTION_UP`：抬起手指，离开屏幕
- `ACTION_CANCEL`：非正常抬起，几乎等同于 `ACTION_UP`，通常是父视图拦截
- `ACTION_POINTER_DOWN`：多指触摸，表示已经有一只手指按下时，另有一只手指再次按下
- `ACTION_POINTER_UP`：多指触摸，表示屏幕上已经有多个手指，抬起其中一只手指后触发的事件

几种常用触摸的类型就这么多，最后两种是多指触摸的情况下才会收到的事件类型，本文我们不打算讨论多指触摸（包括 `TouchTarget`），所以后面会忽略掉

在一次事件分发中，以每个 `DOWN` 事件为开始，`UP` / `CANCEL` 事件为停止，在 `DOWN -> MOVE -> MOVE -> UP / CANCEL` 整个过程看做是一个事件序列

ViewGroup 的消费、分发、拦截与放行

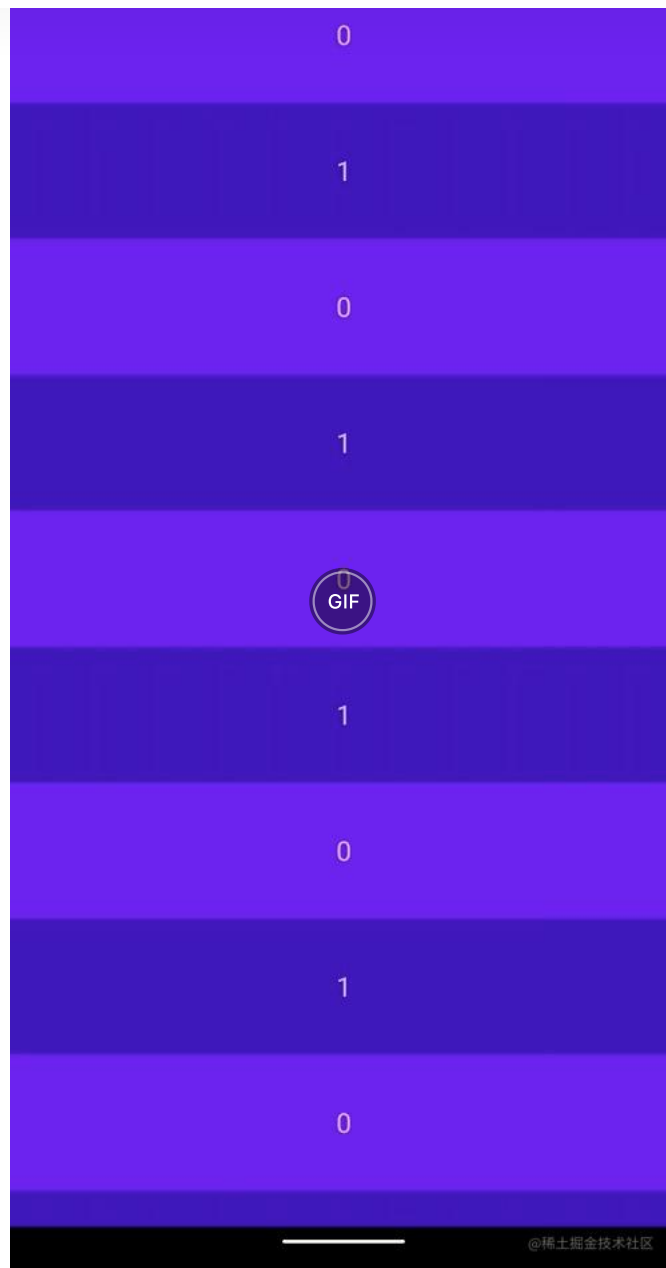
在进入 ViewGroup 的源码之前，我们先来思考一个问题：什么是事件分发？

我们都知道，在 Android 图形系统中，主要绘图的任务是交给 View 去完成的，ViewGroup 是作为管理视图的容器，它的任务是按照一定的规则摆放子 View，自身则基本不参与绘图操作

但是在触摸事件的分发中，ViewGroup 对触摸事件的态度就变了，因为它和子 View 一样，都需要响应触摸事件

1、ViewGroup 四种场景

举个例子，有一个可以纵向滑动的 ViewGroup，里面摆满了 `TextView`，用户在滑动屏幕时，肯定是期望展示更多内容的。那么这时候，就需要 ViewGroup 对子 View 重新布局摆放，将隐藏在屏幕底部的子 View 显示出来



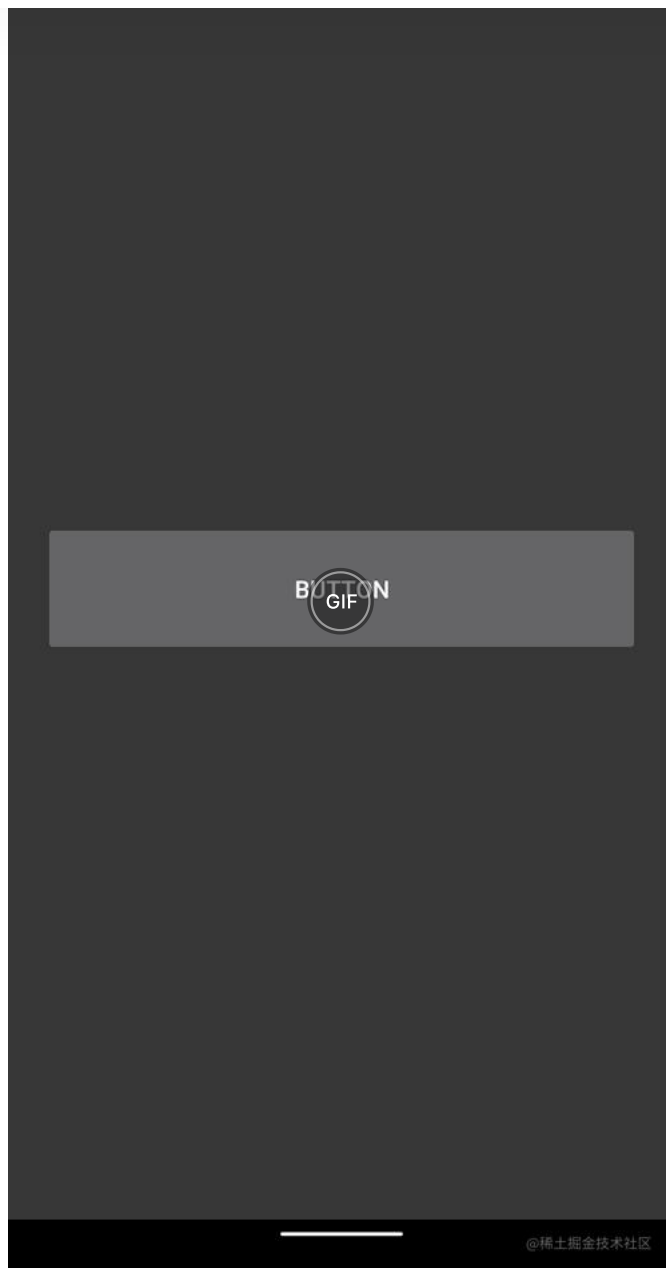
android_graphic_v5_viewgroup_consumed.GIF

图片来源：自己录的

上面的场景中，ViewGroup 必须要拿到用户滑动屏幕的数据，才能计算展示多少视图才算合适

即，ViewGroup 需要消费触摸事件

再举个例子，在一个不支持滑动的 ViewGroup 中，有个居中显示的 Button 按钮



android_graphic_v5_viewgroup_dispatch.GIF

图片来源：自己录的

用户按下屏幕后，ViewGroup 肯定比自己的子 View 要优先拿到触摸事件

而 ViewGroup 自身又不需要这个事件，那么，它就需要根据触摸位置去查找，有没有子 View 需要该事件

经过计算，如果发现用户按偏了，没点到 Button，那就不管了，dispatchTouchEvent() 返回 false，爱谁消费谁消费，反正我不要

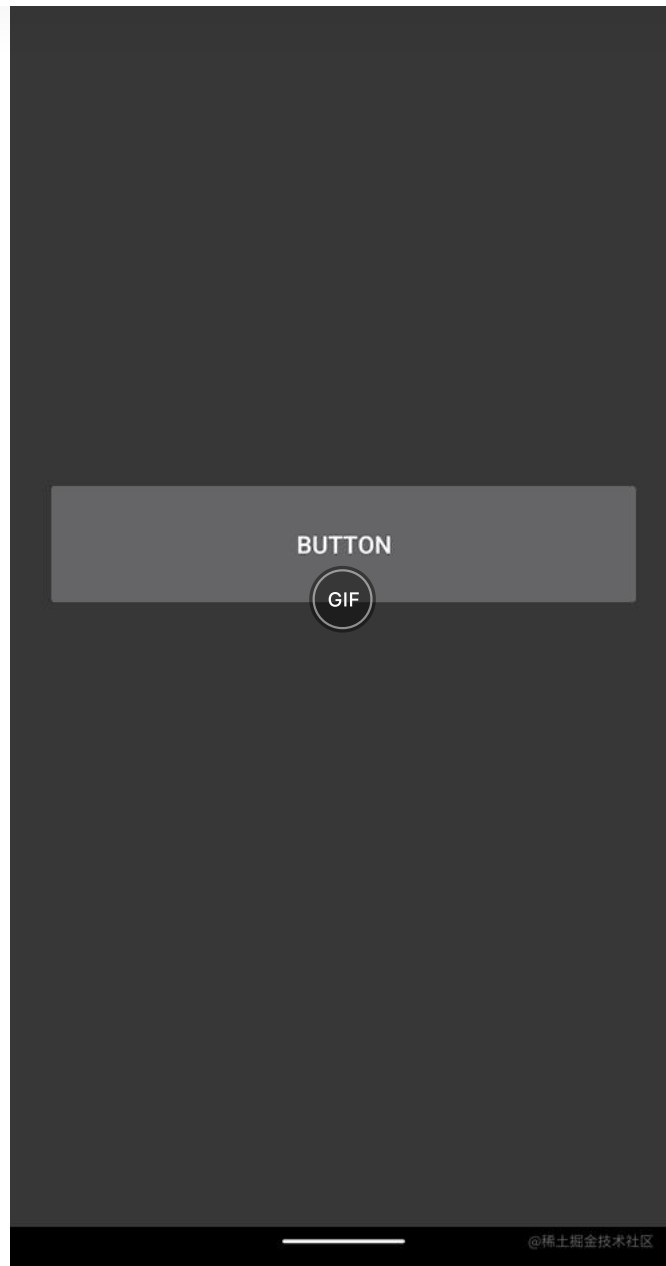
如果发现用户按到了中间的 Button，这时候 ViewGroup 就需要把这个事件交给 Button，询问子 View 要不要消费

即，ViewGroup 需要分发事件

再再举个例子，在一个支持纵向滑动的 ViewGroup 中，有个居中显示的 Button 按钮

现在，用户按下了屏幕中的 Button，ViewGroup 觉得不是滑动事件，就把这个事件分发给了 Button

但是用户在按下 Button 后，接着又开始滑动屏幕了



android_graphic_v5_viewgroup_intercept.GIF

图片来源：自己录的

此时的 ViewGroup 需要计算滑动距离，所以需要这个触摸事件的，那只能对不起 Button 了，ViewGroup 会把本来准备分发给 Button 的事件拦截消费掉

即，ViewGroup 需要拦截事件

另外，我们还需要考虑一种极端情况：如果子 View 和 ViewGroup 都需要触摸事件，应该怎么处理？

按正常的拦截逻辑，ViewGroup 先拿到事件，自己又有消费的需求，肯定是紧着自己用

但是，总会有场景需要将事件优先分发给子视图，比如：

在一个支持纵向滚动的 ViewGroup，它的两个子 View 同样是支持纵向滚动的 ViewGroup，两个子 View 分别都包含若干 TextView

现在的需求是：长按某一个 TextView 时，允许该 TextView 上下自由拖动



android_graphic_v5_viewgroup_request.GIF

图片来源：自己录的

当用户长按 `TextView` 移动时，预期是移动这个 `TextView`，理论上这个移动事件应该被 `TextView` 的爸爸消费掉，因为需要重新布局绘制；

但实际上是 `TextView` 的爷爷消费掉的，因为爷爷觉得用户是在滑动屏幕，要把屏幕下方更多的视图显示出来

ViewGroup 和 ViewGroup 中的子 View 都想要消费事件（滑动冲突），这时候该怎么办？

很简单，我们需要创建一种机制，让 ViewGroup 知道子 View 也需要这个事件

我们暂且把这套机制称之为“请求放行”好了

ViewGroup 为子 View 开放一个请求放行的接口，当 ViewGroup 收到来自子 View 的放行请求时，优先将事件分发给子 View，这样，问题就解决了

触摸事件的消费、拦截、放行与分发，这四种情况几乎覆盖了 ViewGroup 对触摸事件处理（单指）的所有场景

2、事件的放行和拦截

第二章结束时，最终调用停在了 `ViewGroup#dispatchTouchEvent()` 方法，这也是整个 View / ViewGroup 事件分发的起源

```
/frameworks/base/core/java/android/view/ViewGroup.java
class ViewGroup extends View {

    public boolean dispatchTouchEvent(MotionEvent ev) {
        int actionMasked = ev.getAction() & MotionEvent.ACTION_MASK;
        TouchTarget newTouchTarget = null;
        boolean intercepted;
        boolean handled = false;
        if (actionMasked == MotionEvent.ACTION_DOWN || mFirstTouchTarget != null) {
            // 检查子视图是否调用了 requestDisallowInterceptTouchEvent(true) 请求放行
            boolean disallowIntercept = (mGroupFlags & FLAG_DISALLOW_INTERCEPT) != 0;
            if (!disallowIntercept) {
                intercepted = onInterceptTouchEvent(ev); // 子视图未请求放行，询问 ViewGro
            } else {
                intercepted = false;
            }
        } else {
            intercepted = true; // 如果 mFirstTouchTarget 为空，并且事件类型不为 DOWN，表
        }
        return handled;
    }

    public boolean onInterceptTouchEvent(MotionEvent ev) {
        // 询问 ViewGroup 自身是否需要处理事件
        return false;
    }

    public void requestDisallowInterceptTouchEvent(boolean disallowIntercept) {
        if (disallowIntercept) {
            mGroupFlags |= FLAG_DISALLOW_INTERCEPT;
        } else {
            mGroupFlags &= ~FLAG_DISALLOW_INTERCEPT;
        }
    }
}
```

ViewGroup 的几种场景我们已经在上面介绍过了，接下来就是对照源码解释的过程，比较轻松

在事件分发的开始，先是判断了事件是否是 DOWN 类型，或者 `mFirstTouchTarget` 变量是否不为空 (* `mFirstTouchTarget` 记录的是消费上一次 DOWN 事件的是谁*)

满足条件则进入“检查放行”和“检查是否要拦截”的逻辑

`disallowIntercept` 为 true，表示子视图是否调用了 `requestDisallowInterceptTouchEvent(true)` 方法请求放行，将 `intercepted` 标识置为 false，并且，本次事件将不询问 ViewGroup 自身是否要消费

ViewGroup#requestDisallowInterceptTouchEvent() 方法就是之前介绍过的，开放给子 View 请求放行的一套机制

如果子视图没有请求放行，那么调用 `onInterceptTouchEvent()` 询问自己要不要拦截消费，不需要消费事件会继续分发，我们后面会讲到

如果 `mFirstTouchTarget` 为空，并且事件类型不为 DOWN，表示先前的事件也是 ViewGroup 自己消费的，无需执行分发，再次交给自己执行即可，将 `intercepted` 置为 true

只要子 View 没有调用 `requestDisallowInterceptTouchEvent()` 方法请求放行, ViewGroup 有权在任何情况下, 通过调用 `onInterceptTouchEvent()` 返回 `true` 的方式, 拦截任一触摸事件

我们在继承 ViewGroup 以后, 首先要重写 `onInterceptTouchEvent()` 方法, 然后我们根据自己的需求, 判断要不要消费某个事件

`true` 表示自身需要消费, 该事件将会被拦截, 并会在下一步发送到 ViewGroup 自身的 `onTouchEvent()` 方法中, 不会继续向下分发

`false` 表示不消费, 继续向下分发事件

ViewGroup 的放行机制和拦截就结束了, 我们继续看代码, 下一步该执行事件的分发了

3、事件的分发

如果子视图没有请求放行, ViewGroup 自身也不消费, 那么 `intercepted` 标识为 `false`, 进入分发流程

```
/frameworks/base/core/java/android/view/ViewGroup.java
class ViewGroup extends View {

    public boolean dispatchTouchEvent(MotionEvent ev) {
        ...
        // 子视图未请求放行, ViewGroup 自身也不消费, 进入分发流程
        if (!intercepted) {
            if (actionMasked == MotionEvent.ACTION_DOWN) {
                for (int i = mChildrenCount - 1; i >= 0; i--) {
                    ...// 检查子 View 是否可触摸, 是否在触摸区域内等等, 过程略
                    // 找到合适的子 View 后, 调用 dispatchTransformedTouchEvent() 执行事件分发
                    if (dispatchTransformedTouchEvent(ev, false, child, idBitsToAssign,
                        newTouchTarget = addTouchTarget(child, idBitsToAssign); // 生成新的 TouchTarget
                        break;
                    }
                }
                // 没有新的需要触摸事件的视图, 那么, 把链表尾部的 TouchTarget 拿出来, 在下一步分发
                if (newTouchTarget == null && mFirstTouchTarget != null) {
                    newTouchTarget = mFirstTouchTarget;
                    while (newTouchTarget.next != null) {
                        newTouchTarget = newTouchTarget.next;
                    }
                }
            }
        }
        return handled;
    }

    private boolean dispatchTransformedTouchEvent(MotionEvent event, boolean cancel, View child,
        boolean handled;
        if (child == null) {
            handled = super.dispatchTouchEvent(event); // ViewGroup 继承自 View, 这里调用 super 方法
        } else {
            handled = child.dispatchTouchEvent(event);
        }
        return handled;
    }
}
```

分发的过程是将所有符合条件的子 View 都询问一遍

在上面这段代码中, 使用 `for` 循环遍历所有的子 View, 检查每个子 View 是否可被触摸等等

所有的子 View 遍历一遍后，如果发现没有消费的视图，并且，当前 `mFirstTouchTarget` 不为空，那直接将保留着上一次消费的 View 最近的一个 `TouchTarget` 拿出来，等待下一步执行

4、事件的消费

事件的消费代码比较简单：

```
//frameworks/base/core/java/android/view/ViewGroup.java
class ViewGroup extends View {

    public boolean dispatchTouchEvent(MotionEvent ev) {
        ...
        // 跑到这，如果 mFirstTouchTarget 还是为空，表示这个事件 ViewGroup 自身要消费
        if (mFirstTouchTarget == null) {
            handled = dispatchTransformedTouchEvent(ev, canceled, null, TouchTarget.ALL_P
        } else {
            TouchTarget target = mFirstTouchTarget;
            // 遍历 TouchTarget 链表，执行事件分发，代码有去重操作，被我删了，即之前分发过的新
            while (target != null) {
                final TouchTarget next = target.next;
                if (dispatchTransformedTouchEvent(ev, cancelChild, target.child, target
                    handled = true;
                }
                target = next;
            }
        }
        return handled;
    }

    private boolean dispatchTransformedTouchEvent(MotionEvent event, boolean cancel, Vi
        boolean handled;
        if (child == null) {
            handled = super.dispatchTouchEvent(event); // ViewGroup 继承自 View，这里调
        } else {
            handled = child.dispatchTouchEvent(event);
        }
        return handled;
    }
}
```

如果 `mFirstTouchTarget` 为空，表示这个事件 ViewGroup 自身要消费，调用 `dispatchTransformedTouchEvent()` 方法

在 `dispatchTransformedTouchEvent()` 方法中，如果 `child` 参数为空，表示是 ViewGroup 要消费，那么调用 ViewGroup 的父类：`View#dispatchTouchEvent()`

`dispatchTouchEvent()` 方法最终把事件传递给 `onTouchEvent()`，我们马上就能看到 View 的消费流程了

如果 `mFirstTouchTarget` 对象不为空，说明有其他子 View 消费了事件（可能有多），依旧调用 `dispatchTransformedTouchEvent()` 执行分发

好了，ViewGroup 的 消费、分发、拦截与放行，到这里就结束了，接下来我们看 `View#dispatchTouchEvent()` 的流程

View 的消费

聊完了 ViewGroup 对触摸事件的处理，我们接着来聊 View 这边是怎：

都是调用到 `View#dispatchTouchEvent()` 中

`View#dispatchTouchEvent()` 逻辑比较少，重要代码只有一句，调用了 `onTouchEvent()` 消费事件，除此之外 `View` 的事件分发就没什么好聊的了

```
//frameworks/base/core/java/android/view/View.java
class View {

    public boolean dispatchTouchEvent(MotionEvent event) {
        boolean result = onTouchEvent(event);
        return result;
    }

    public boolean onTouchEvent(MotionEvent event) {
        return false;
    }
}
```

好了，`View / ViewGroup` 的事件分发到这里就结束了，当然我们看到的是非常精简的版本了，只有三两个关键函数，并且几乎没有任何细节

因为 `View / ViewGroup` 的事件分发比较简单，不像 Framework 的逻辑，绕来绕去的，事件分发的逻辑大部分时间都在内部做跳转，感兴趣的朋友自己去跟一遍源码很快就了解清楚了

四、结语

本篇文章稍微有点长，从硬件驱动，系统内核，到 Framework 都有涉及。其中，了解 IMS 的实现原理，APP 和 IMS 通信的建立，以及 `ViewGroup` 的 `dispatchTouchEvent()` 方法的事件派发逻辑，是本篇文章比较重要的内容

在文章的最后，我们用张大伟老师《[深入理解Android 卷III](#)》书中的一段话作为本文总结：

简单来说，内核将原始事件写入到设备文件中，`InputReader` 不断地通过 `EventHub` 将原始事件取出来，解析加工成 `KeyEvent`、`MotionEvent` 事件，然后交给 `InputDispatcher`

`InputDispatcher` 根据 `WMS` 提供的窗口信息将事件交给合适的窗口

接着，窗口的 `ViewRootImpl` 对象再沿着控件树将事件派发给感兴趣的控件，控件对其收到的事件作出响应，更新自己的画面、执行特定的动作

所有这些参与者以 `IMS` 为核心，构建了 `Android` 庞大而复杂的输入体系。

好了，本篇文章到这里就全部结束了。文中提到的硬件驱动和系统内核这两块暂时还不是我擅长的领域，所以如果各位大佬发现本文有写的不对的地方，还望及时指出，我会第一时间改正，感谢

另外，欢迎各位大佬给我留言，我们一起来探讨技术问题

全文完

五、参考资料

- [《深入理解Android 卷III - 张大伟》](#)
- [电阻屏已经被智能手机抛弃，还有哪些应用场景？](#)
- [手机全贴合屏幕技术解析](#)
- [【Linux驱动】I2C子系统与触摸屏驱动 - @hongZ](#)
- [【Linux驱动】input子系统与按键驱动 - @hongZ](#)
- [Linux驱动开发|input子系统 - 安迪西](#)
- [从 0 开始学 Linux 驱动开发 - Hcamael](#)
- [Android\(Linux\) 输入子系统解析 - Andy Lee](#)

- [Android 触摸事件分发机制（一）从内核到应用 一切的开始 - 吴迪](#)
- [Android 触摸事件分发机制（二）原始事件消息传递与分发的开始 - 吴迪](#)
- [Android事件分发机制二：核心分发逻辑源码解析 - 一只修仙的猿](#)
- [InputChannel and InputDispatcher in Android](#)



发布于 2022-11-29 10:07 · IP 属地浙江

Android 应用

写下你的评论...



还没有评论，发表第一个评论吧

推荐阅读

分享 | 屏幕调节软件,F.lux和 Nocturne

<http://weixin.qq.com/r/GDkwKAC> (二维码自动识别) Hello 大家好我是 BreakNg 今天分享给大家的是两个屏幕调节软件 | 1 f.lux ▶ f.lux创意十足很贴心的一款自动屏幕亮度色彩...

Break...

发表于年轻人x科...

Scrcpy-在电脑无缝操作手机 (投屏/录屏/免Root)

工作中会遇到想把手机投放到电脑上进行演示，还有可能想在电脑上使用Android 应用/玩游戏等。除了使用一些虚拟机软件之后，还可以应用一款开源免费的安卓手机屏幕投屏+控制软件-Scrcpy。 01...

芒果小西米



手
哪
E