Android Source code notes

# Handler

一个信息从handler发送，然后到接收，这个过程到底是什么样的？

场景：

一个倒计时，发送方handler里的handler发送信息到接受方handler里，接受方handler里的handler发送信息到主线程，主线程显示结果

1. 分别创建两个handler，handler1作用在thread2和1里，handler2作用在主线程和handler2中。
2. handler在创建时会获取当前线程的looper，由于Thread2中没有Looper，因此我们需要调用Looper.prepare方法创建thread2的looper，在prepare方法中，实际我们是调用了looper的构造函数，创建了一个looper，并创建了messagequeue，然后将这个looper放到了threadlocal中。
3. 接下来就可以创建handler了，handler创建时会实现callback或者实现handlemessage方法，两种方法都可以，在创建时，会获取当前线程的looper，和looper中的messagequeue。
4. 创建完之后我们就可以调用Looper.loop方法阻塞式获取messagequeue中的message了，注意要在创建完handler之后调用，否则这个线程会被阻塞，loop之后的代码都不会执行了.
5. 在Thread1中创建通过Message.obtain()方法创建message
6. Thread1里的handler1在循环里调用sendMessage(message)，循环中包含Thread.sleep, 发送信息到Thread1。sendmessage最终会将信息发送到handler所绑定的messagequeue里，同时message.target值等于发送他的handler
7. 这时thread2的looper会获取到thread1种传来的message，通过调用message.target.dispatchmessage方法,在这个方法里调用了handler的handlemessage方法。
8. 由此实现了handler跨线程交流
9. 以上步骤都可以通过handlerthread简化,使用handlerthread其实也就是为了获取里面的Looper，从而可以在创建handler的时候，将这个looper传进去

## Message

**It contains three fields to store info:**

**public int what**;**public int arg1**;**public int arg2**;

**long when**;  
Bundle **data**;  
Handler **target**;  
Runnable **callback**;  
*// sometimes we store linked lists of these things*Message **next**;

**private static** Message *sPool*;

**There is a field**

*/\*\* If set message is in use.  
 \* This flag is set when the message is enqueued and remains set while it  
 \* is delivered and afterwards when it is recycled. The flag is only cleared  
 \* when a new message is created or obtained since that is the only time that  
 \* applications are allowed to modify the contents of the message.  
 \*  
 \* It is an error to attempt to enqueue or recycle a message that is already in use.  
 \*/*

**static final int *FLAG\_IN\_USE*** = 1 << 0;

**The best way to get a message is to call obtain() method**

**public static** Message obtain() {  
 **synchronized** (***sPoolSync***) {  
 **if** (*sPool* != **null**) {  
 Message m = *sPool*;  
 *sPool* = m.next;  
 m.next = **null**;  
 m.flags = 0; *// clear in-use flag  
 sPoolSize*--;  
 **return** m;  
 }  
 }  
 **return new** Message();  
}

**void** recycleUnchecked() {  
 *// Mark the message as in use while it remains in the recycled object pool.  
 // Clear out all other details.* **flags** = ***FLAG\_IN\_USE***;  
 **what** = 0;  
 **arg1** = 0;  
 **arg2** = 0;   
 **...**  
  
 **synchronized** (***sPoolSync***) {  
 **if** (*sPoolSize* < ***MAX\_POOL\_SIZE***) {  
 **next** = *sPool*;  
 *sPool* = **this**;  
 *sPoolSize*++;  
 }  
 }  
}

It implements a pool to avoid allocating new objects in many cases.

这里的recycle机制还是挺复杂的，首先里面有个静态Message变量sPool，由于是静态的，所以是被所有Message共享。

1. 通过obtain方法第一次创建一个Message，由于sPool是空的，所以直接return一个new message
2. 这个message方法调用了recycle()，这时这个message会被赋给sPool，代表这个被回收了，下次调用就可以直接用sPool了
3. 又有新的Message被回收，被赋给sPool，这时候旧的Message会被新的sPool的next指向

总的来说，其实是从Message中的static变量sPool所形成的链表中取Message，如果被recycle了，会挂在sPool的头部。

## Looper

*// sThreadLocal.get() will return null unless you've called prepare().***static final** ThreadLocal<Looper> ***sThreadLocal*** = **new** ThreadLocal<Looper>();  
**private static** Looper *sMainLooper*; *// guarded by Looper.class***final** MessageQueue **mQueue**;  
**final** Thread **mThread**;

首先看一下Looper怎么用

**class** LooperThread **extends** Thread {  
 **public** Handler **mHandler**;  
 **public void** run() {  
 Looper.prepare();  
 **mHandler** = **new** Handler() {  
 **public void** handleMessage(Message msg) {  
 *// process incoming messages here* }  
 };  
 Looper.loop();  
 }  
}

看一下prepare方法

**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));  
}

*/\*\*  
 \* Return the Looper object associated with the current thread. Returns  
 \* null if the calling thread is not associated with a Looper.  
 \*/***public static** @Nullable Looper myLooper() {  
 **return *sThreadLocal***.get();  
}

所以Looper里维护了一个ThreadLocal，就是为了存储每个线程对应的Looper

loop()

*/\*\*  
 \* 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;  
  
 **for** (;;) {  
 Message msg = queue.next(); *// might block* **if** (msg == **null**) {  
 *// No message indicates that the message queue is quitting.* **return**;  
 }  
 **try** {  
 msg.target.dispatchMessage(msg); } **finally** {  
 }  
 msg.recycleUnchecked();  
 }

一个无限循环的loop，读取message，之后调用message里target（handler）的dispatch方法。

## Handler

*/\*\*  
 \* Callback interface you can use when instantiating a Handler to avoid  
 \* having to implement your own subclass of Handler.  
 \*/***public interface** Callback {  
 */\*\*  
 \** ***@param msg*** *A {****@link*** *android.os.Message Message} object  
 \** ***@return*** *True if no further handling is desired  
 \*/* **public boolean** handleMessage(Message msg);  
}

*/\*\*  
 \* Handle system messages here.  
 \*/***public void** dispatchMessage(Message msg) {  
 **if** (msg.callback != **null**) {  
 *handleCallback*(msg);  
 } **else** {  
 **if** (**mCallback** != **null**) {  
 **if** (**mCallback**.handleMessage(msg)) {  
 **return**;  
 }  
 }  
 handleMessage(msg);  
 }  
}

**public final** Message obtainMessage()  
{  
 **return** Message.*obtain*(**this**);  
}

**public boolean** sendMessageAtTime(Message msg, **long** uptimeMillis) {  
 MessageQueue queue = **mQueue**;  
 **if** (queue == **null**) {  
 **return false**;  
 }  
 **return** enqueueMessage(queue, msg, uptimeMillis);  
}

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

## Messagequeue

**boolean** enqueueMessage(Message msg, **long** when) {  
 **synchronized** (**this**) {  
 **if** (**mQuitting**) {  
 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();  
 msg.when = when;  
 Message p = **mMessages**;  
 **boolean** needWake;  
 **if** (p == **null** || when == 0 || when < p.when) {  
 *// 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;  
 **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**;  
}

如果队列为空，或者当前处理的时间点为0（when的数值，when表示Message将要执行的时间点），或者当前Message需要处理的时间点先于队列中的首节点，那么就将Message放入队列首部，否则进行第v2步。

遍历队列中Message，找到when比当前Message的when大的Message，将Message插入到该Message之前，如果没找到则将Message插入到队列最后。

判断是否需要唤醒，一般是当前队列为空的情况下，next那边会进入睡眠，需要enqueue这边唤醒next函数。后面会详细介绍

Message next() {  
 *// Return here if the message loop has already quit and been disposed.  
 // This can happen if the application tries to restart a looper after quit  
 // which is not supported.* **final long** ptr = **mPtr**;  
 **if** (ptr == 0) {  
 **return null**;  
 }  
  
 **int** pendingIdleHandlerCount = -1; *// -1 only during first iteration* **int** nextPollTimeoutMillis = 0;  
 **for** (;;) {  
 **if** (nextPollTimeoutMillis != 0) {  
 Binder.*flushPendingCommands*();  
 }  
  
 nativePollOnce(ptr, nextPollTimeoutMillis);  
  
 **synchronized** (**this**) {  
 *// Try to retrieve the next message. Return if found.* **final long** now = SystemClock.*uptimeMillis*();  
 Message prevMsg = **null**;  
 Message msg = **mMessages**;  
 **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());  
 }  
 **if** (msg != **null**) {  
 **if** (now < msg.when) {  
 *// Next message is not ready. Set a timeout to wake up when it is ready.* nextPollTimeoutMillis = (**int**) Math.min(msg.when - now, Integer.***MAX\_VALUE***);  
 } **else** {  
 *// Got a message.* **mBlocked** = **false**;  
 **if** (prevMsg != **null**) {  
 prevMsg.next = msg.next;  
 } **else** {  
 **mMessages** = msg.next;  
 }  
 msg.next = **null**;  
 **if** (***DEBUG***) Log.*v*(***TAG***, **"Returning message: "** + msg);  
 msg.markInUse();  
 **return** msg;  
 }  
 } **else** {  
 *// No more messages.* nextPollTimeoutMillis = -1;  
 }  
  
 *// Process the quit message now that all pending messages have been handled.* **if** (**mQuitting**) {  
 dispose();  
 **return null**;  
 }  
  
 *// If first time idle, then get the number of idlers to run.  
 // Idle handles only run if the queue is empty or if the first message  
 // in the queue (possibly a barrier) is due to be handled in the future.* **if** (pendingIdleHandlerCount < 0  
 && (**mMessages** == **null** || now < **mMessages**.when)) {  
 pendingIdleHandlerCount = **mIdleHandlers**.size();  
 }  
 **if** (pendingIdleHandlerCount <= 0) {  
 *// No idle handlers to run. Loop and wait some more.* **mBlocked** = **true**;  
 **continue**;  
 }  
  
 **if** (**mPendingIdleHandlers** == **null**) {  
 **mPendingIdleHandlers** = **new** IdleHandler[Math.*max*(pendingIdleHandlerCount, 4)];  
 }  
 **mPendingIdleHandlers** = **mIdleHandlers**.toArray(**mPendingIdleHandlers**);  
 }  
  
 *// Run the idle handlers.  
 // We only ever reach this code block during the first iteration.* **for** (**int** i = 0; i < pendingIdleHandlerCount; i++) {  
 **final** IdleHandler idler = **mPendingIdleHandlers**[i];  
 **mPendingIdleHandlers**[i] = **null**; *// release the reference to the handler* **boolean** keep = **false**;  
 **try** {  
 keep = idler.queueIdle();  
 } **catch** (Throwable t) {  
 Log.*wtf*(***TAG***, **"IdleHandler threw exception"**, t);  
 }  
  
 **if** (!keep) {  
 **synchronized** (**this**) {  
 **mIdleHandlers**.remove(idler);  
 }  
 }  
 }  
  
 *// Reset the idle handler count to 0 so we do not run them again.* pendingIdleHandlerCount = 0;  
  
 *// While calling an idle handler, a new message could have been delivered  
 // so go back and look again for a pending message without waiting.* nextPollTimeoutMillis = 0;  
 }  
}

整个next函数的主要是执行步骤是:

step1: 初始化操作，如果mPtr为null，则直接返回null，设置nextPollTimeoutMillis为0，进入下一步。

step2: 调用nativePollOnce, nativePollOnce有两个参数,第一个为mPtr表示native层MessageQueue的指针，nextPollTimeoutMillis表示超时返回时间，调用这个nativePollOnce会等待wake，如果超过nextPollTimeoutMillis时间，则不管有没有被唤醒都会返回。-1表示一直等待，0表示立刻返回。下一小节单独介绍这个函数。

step3: 获取队列的头Message(msg)，如果头Message的target为null，则查找一个异步Message来进行下一步处理。当队列中添加了同步Barrier的时候target会为null。

step4: 判断上一步获取的msg是否为null，为null说明当前队列中没有msg，设置等待时间nextPollTimeoutMillis为-1。实际上是等待enqueueMessage的nativeWake来唤醒，执行step4。如果非null，则下一步

step5: 判断msg的执行时间(when)是否比当前时间(now)的大，如果小，则将msg从队列中移除，并且返回msg，结束。如果大则设置等待时间nextPollTimeoutMillis为(int) Math.min(msg.when - now, Integer.MAX\_VALUE)，执行时间与当前时间的差与MAX\_VALUE的较小值。执行下一步

step6: 判断是否MessageQueue是否已经取消，如果取消的话则返回null，否则下一步

step7: 运行idle Handle，idle表示当前有空闲时间的时候执行，而运行到这一步的时候，表示消息队列处理已经是出于空闲时间了（队列中没有Message，或者头部Message的执行时间(when)在当前时间之后）。如果没有idle，则继续step2，如果有则执行idleHandler的queueIdle方法，我们可以自己添加IdleHandler到MessageQueue里面（addIdleHandler方法），执行完后，回到step2。

需要说的时候，我们平常只是使用Message，但是实际上IdleHandler如果使用的好，应该会达到意想不到的效果，它表示MessageQueue有空闲时间的时候执行一下。

**SyncBarrier**

**public void** dispatchInputEvent(InputEvent event, InputEventReceiver receiver) {  
 SomeArgs args = SomeArgs.obtain();  
 args.arg1 = event;  
 args.arg2 = receiver;  
 Message msg = mHandler.obtainMessage(MSG\_DISPATCH\_INPUT\_EVENT, args);  
 msg.setAsynchronous(**true**);  
 mHandler.sendMessage(msg);  
}

注意它这里就是使用的异步Message，使用了msg.setAsyncronous(true)。 而SyncBarrier有什么用处呢？我们刚刚介绍的时候，当消息队列的第一个Message的target的时候，表示它是一个SyncBarrier，它会阻拦同步消息，而选择队列中第一个异步消息处理，如果没有则会阻塞。这表示什么呢？这是表示第一个Message是SyncBarrier的时候，会只处理异步消息。而我们前面介绍了InputEvent的时候，它就是异步消息，在有SyncBarrier的时候就会被优先处理。所以在调用了scheduleTraversal的时候，就会只处理触摸事件这些消息了，保证用户体验。保证了触摸事件及时处理，实际上这也能减少ANR。如果这个时候MessageQueue中有很多Message，也能够及时处理那些触摸事件的Message了。

## Context

*/\*\*  
 \* Interface to global information about an application environment. It  
 \* allows access to application-specific resources and classes, as well as  
 \* up-calls for application-level operations such as launching activities,  
 \* broadcasting and receiving intents, etc.  
 \*/*

Some important methods:

getResourses()

getSharedPreferences()

getDir()

startActivity()

sendBroadcast()

registerReceiver()

bindService()

# Rxjava

## 前言：

最近阅读了Rxjava源码，也尝试了单独构造一个简单的Rxjava。但是，很多时候我只知道这段代码是什么意思，rxjava是如何工作的，但并不知道为什么代码会这么写，因此我决定从头到尾，由浅入深，再重新研究一次Rxjava。

本文的目的有两个：

1. 深入了解RxJava
2. 培养能独立写出Rxjava的能力

## 什么是Rxjava？

根据官方的定义：RxJava is a Java VM implementation of [Reactive Extensions](http://reactivex.io/): a library for composing asynchronous and event-based programs by using observable sequences.

通俗来讲就是基于响应式能够异步处理事件的库

那什么是响应式？

有篇文章很好的解释了响应式编程：

(English)<https://gist.github.com/staltz/868e7e9bc2a7b8c1f754>

(中文) <http://wiki.jikexueyuan.com/project/android-weekly/issue-145/introduction-to-RP.html>

例子中使用了Rxjava，其实Rxjava就是响应式编程的一种实现。

## 从头造一个Rxjava

好了，如果看完以上的文章，我们应该知道RxJava是一个事件源产生事件，然后由接受者接受并且处理。

好的，按照上面的说法，那我们先创建一个事件吧，根据事件的定义：

*Stream 就是一个按时间排序的 Events 序列,它可以放射三种不同的 Events：(某种类型的)Value、Error 或者一个" Completed" Signal。*

我们也不清楚到底发送的是那种Event，于是干脆全部写在一起算了：

**public class** Event {  
 */\*  
 1 : value  
 2 : Error  
 3 : Completed  
 \*/* **int type**;  
 **int value**;  
  
 **public** Event(**int** type, **int** value) {  
 **this**.**type** = type;  
 **this**.**value** = value;  
 }  
}

有了事件之后，那我们需要写一个接收事件的接收者，接收者会根据接收到的Event的signal来确定执行哪个事件：

**public class** Reciever {  
  
 Event **event**;  
 **void** onNext(**int** value){  
 System.***out***.println(value);  
 }  
  
 **void** onError(){  
 System.***out***.println(**"error"**);  
 }  
  
 **void** onCompleted(){  
 System.***out***.println(**"completed"**);  
 }  
  
 **public void** setEvent(Event e) {  
 **this**.**event** = e;  
 }  
  
 **void** execute(){  
 **if**(**event**.**type**==1){  
 onNext(**event**.**value**);  
 } **else if**(**event**.**type**==2) {  
 onError();  
 } **else if**(**event**.**type**==3){  
 onCompleted();  
 }  
 }  
  
}

同样我们也需要一个事件的发送者，发送给接收者

**public class** Sender {  
 Reciever<Integer> **reciever**;  
  
 Sender() {  
 }  
  
 **public void** setReciever(Reciever<Integer> reciever) {  
 **this**.**reciever** = reciever;  
 }  
  
 **void** sendEvent(Event e) {  
 **reciever**.setEvent(e);  
 **reciever**.execute();  
 }  
}

同时写一个Main函数测试一下：

**public class** Main {  
 **public static void** main(String[] args) {  
 Sender sender=**new** Sender();

sender.setReciever(new Reciever());  
 sender.sendEvent(**new** Event(1,2,3));  
 }  
}

100

可以看到接收者接收到了信息并且打印了出来

这里接收者的行为我们已经定义好了，就是将接收的值或者类型打印出来，如果我们不想打印，而是直接显示出来呢，或者我们想将其保存下来。那么显然，我们需要将这个类抽象出来。另外一点，Event发送过来的信息可能不仅经是int类型，也可能是string类型，所以我们需要用到泛型

**public interface** Reciever<T> {  
 **void** onNext(T t);  
 **void** onError();  
 **void** onCompleted();  
}

**public class** OutputReciever<T> **implements** Reciever<T> {  
  
 **private** Event<Integer> **event**;  
  
 @Override  
 **public void** onNext(T t) {  
 System.***out***.println(t);  
 }  
  
 @Override  
 **public void** onError() {  
 System.***out***.println(**"error"**);  
 }  
  
 @Override  
 **public void** onCompleted() {  
 System.***out***.println(**"completed"**);  
 }  
  
 **public void** setEvent(Event e) {  
 **this**.**event** = e;  
 }  
  
 **void** execute(){  
 **if**(**event**.**type**==1){  
 onNext((T)**event**.**value**);  
 } **else if**(**event**.**type**==2) {  
 onError();  
 } **else if**(**event**.**type**==3){  
 onCompleted();  
 }  
 }  
  
}

假设我们现在要发送多个Event，并且记录下Event发送的时间，那么该如何写呢:

**public class** Main {  
 **public static void** main(String[] args) {  
 Sender sender=**new** Sender();  
 sender.sendEvent(**new** Event<Integer>(1,100));  
 System.***out***.println(java.time.LocalDate.*now*());  
 sender.sendEvent(**new** Event<Integer>(1,101));  
 System.***out***.println(java.time.LocalDate.*now*());  
 sender.sendEvent(**new** Event<Integer>(1,102));  
 System.***out***.println(java.time.LocalDate.*now*());  
 sender.sendEvent(**new** Event<Integer>(1,103));  
 System.***out***.println(java.time.LocalDate.*now*());  
 }  
}

那这样岂不是我每发送一个事件都要写一个记录时间的？ 幸运的是，命令模式为我们解决了这个问题，我们新加一个client类：

**public class** Client {  
 Sender **sender**;  
 Event<Integer> **event**;  
  
 **public** Client() {  
 }  
  
 **public** Client(Sender sender) {  
 **this**.**sender** = sender;  
 }  
  
 **void** setEvent(Event e){  
 **this**.**event**=e;  
 }  
  
 **void** excute(){  
 **sender**.sendEvent(**event**);  
 System.***out***.println(java.time.LocalDate.*now*());  
 }  
}

**public class** Main {  
 **public static void** main(String[] args) {  
 Client client=**new** Client(**new** Sender());  
 client.setEvent(**new** Event<Integer>(1,100));  
 client.excute();  
 client.setEvent(**new** Event<Integer>(1,101));  
 client.excute();  
 client.setEvent(**new** Event<Integer>(1,102));  
 client.excute();  
 client.setEvent(**new** Event<Integer>(1,103));  
 client.excute();  
 }  
}

这样虽然我们每次都要执行一次excute，但是同样如果我们想一次发送多个事件的话，可以在client里实现一个queue，加入后一次执行完，但在Rxjava里，我们目前先不考虑这个问题

好了，接下来我们来考虑另一件事，就是如何定义一个接受事件后的行为，在例子中，我们直接在outputReciever中输出了传过来的值，那么如果我不想输出，而是想做一些其他处理那，一个很直接的方法就是重新创建一个类，然后实现Reciever方法，然后在sender类中传入reciever的值, 但这样岂不是每次都要重新创建一个新类，能不能直接以参数的形式传入，类似于这样：

Client client=**new** Client(**new** Sender().setReciever(**new** Reciever<Integer>() {  
 @Override  
 **public void** onNext(Integer integer) {  
   
 }  
  
 @Override  
 **public void** onError() {  
  
 }  
  
 @Override  
 **public void** onCompleted() {  
  
 }  
}));  
client.setEvent(**new** Event<Integer>(1,100));  
client.excute();

于是我们把sender改成这样：

**public class** Sender {  
 Reciever<Integer> **reciever**;  
  
 **public** Sender setReciever(Reciever<Integer> reciever) {  
 **this**.**reciever** = reciever;  
 **return this**;  
 }  
  
 **void** sendEvent(Event e) {  
 **reciever**.setEvent(e);  
 **reciever**.execute();  
 }  
}

但这样一来，

**reciever**.setEvent(e);  
**reciever**.execute();

这两个方法就实现不了了，显然Reciever是不能包含这两个方法的，我们对Reciever的定义就是接收到signal之后根据signal改如何反应的方法，那么，我们就直接在sendEvent中来判断吧。

**public class** Sender<T> {  
 Reciever<T> **reciever**;  
  
 Sender() {  
 }  
  
 **public** Sender setReciever(Reciever<T> reciever) {  
 **this**.**reciever** = reciever;  
 **return this**;  
 }  
  
 **void** sendEvent(Event e) {  
 **if**(e.**type**==1){  
 **reciever**.onNext((T)e.**value**);  
 } **else if**(e.**type**==2) {  
 **reciever**.onError();  
 } **else if**(e.**type**==3){  
 **reciever**.onCompleted();  
 }  
 }  
}

Client client = **new** Client(**new** Sender().setReciever(**new** Reciever<Integer>() {  
 @Override  
 **public void** onNext(Integer integer) {  
 System.***out***.println(integer);  
 }  
  
 @Override  
 **public void** onError() {  
  
 }  
  
 @Override  
 **public void** onCompleted() {  
  
 }  
}));  
client.setEvent(**new** Event<Integer>(1, 100));  
client.excute();

Ok，执行一遍，没有问题

在精细一点，Sender这个类似乎和Client类紧紧绑定在一起，干脆把两个类合并在一起：

**public class** Client<T> {  
 Sender **sender**;  
 Event<T> **event**;  
  
 **public** Client(Sender sender) {  
 **this**.**sender** = sender;  
 }  
  
 **void** setEvent(Event e){  
 **this**.**event**=e;  
 }  
  
 **void** excute(){  
 **sender**.sendEvent(**event**);  
 System.***out***.println(java.time.LocalDate.*now*());  
 }  
  
 **public static class** Sender<T>{  
 Reciever<T> **reciever**;  
  
 Sender() {  
 }  
  
 **public** Sender setReciever(Reciever<T> reciever) {  
 **this**.**reciever** = reciever;  
 **return this**;  
 }  
  
 **void** sendEvent(Event e) {  
 **if**(e.**type**==1){  
 **reciever**.onNext((T)e.**value**);  
 } **else if**(e.**type**==2) {  
 **reciever**.onError();  
 } **else if**(e.**type**==3){  
 **reciever**.onCompleted();  
 }  
 }  
 }  
  
}

好了，这样我们成功简化了一个类

**继续优化（观察者模式）**

在响应式编程中，一个stream是自然而来的，我们需要定义的是对这个流的操作和响应，也就是说stream是事件源，操作和响应是对象。stream不需要关心到底有多少操作和响应，那么我们定义的模式应该是由stream去添加操作和响应。在之前的代码中，我们创建client的时候就已经定义好了响应，之后再添加事件的。现在应该反过来：

Execute实在有点多余，sender现在似乎也已经没什么用了

**public class** Client<T> {  
 Event<T> **event**;  
  
 **public** Client(Event e) {  
 **this**.**event**=e;  
  
 }  
 **void** addReciever(Reciever reciever){  
 **if**(**event**.**type**==1){  
 reciever.onNext((T)**event**.**value**);  
 } **else if**(**event**.**type**==2) {  
 reciever.onError();  
 } **else if**(**event**.**type**==3){  
 reciever.onCompleted();  
 }  
 }  
}

Client client = **new** Client(**new** Event<Integer>(1, 100));  
client.addReciever(**new** Reciever<Integer>(){  
 @Override  
 **public void** onNext(Integer t) {  
 System.***out***.println(t);  
 }  
  
 @Override  
 **public void** onError() {  
  
 }  
  
 @Override  
 **public void** onCompleted() {  
  
 }  
});

client创建加订阅写了两步，可以通过建造者模式和工厂模式进行优化

Client.*create*(**new** Event<Integer>(1, 100))  
.addReciever(**new** Reciever<Integer>(){  
 @Override  
 **public void** onNext(Integer t) {  
 System.***out***.println(t);  
 }  
  
 @Override  
 **public void** onError() {  
  
 }  
  
 @Override  
 **public void** onCompleted() {  
  
 }  
});

**public class** Client<T> {  
 Event<T> **event**;  
  
 **public** Client(Event<T> event) {  
 **this**.**event** = event;  
 }  
  
 **public static** <T> Client<T> create(Event<T> e){  
 **return new** Client<T>(e);  
 }  
  
 **void** addReciever(Reciever reciever){  
 **if**(**event**.**type**==1){  
 reciever.onNext((T)**event**.**value**);  
 } **else if**(**event**.**type**==2) {  
 reciever.onError();  
 } **else if**(**event**.**type**==3){  
 reciever.onCompleted();  
 }  
 }  
}

好了，现在看起来已经非常接近Rxjava的源码了

接下来我们修改一下stream的定义，之前stream是一个signal，Reciever会根据signal来做出相应的反应，这次stream将其定义为需要Reciever做出的反应，我们将这种流定义为ActionEven

看一下Event类，里面添加了一个reciever

**public class** Event<T> {  
 */\*  
 1 : value  
 2 : Error  
 3 : Completed  
 \*/* **int type**;  
 T **value**;  
  
 **public** Event(**int** type, T value) {  
 **this**.**type** = type;  
 **this**.**value** = value;  
 }  
  
 **public void** call(Reciever<T> reciever){  
   
 }  
}

显然，这个Reciever必须是我们之后添加的Reciever类，那这个类既然还没被创建，那怎么添加呢，这时候我们可以通过回调的方式，一旦Reciever被创建了，我们就可以把这个值传递给Event了e.

**public class** Client<T> {  
 ActionEvent<T> **actionEvent**;  
  
 **public** Client(ActionEvent<T> actionEvent) {  
 **this**.**actionEvent** = actionEvent;  
 }  
  
 **public static** <T> Client<T> create(ActionEvent<T> e){  
 **return new** Client<T>(e);  
 }  
  
 **void** addReciever(Reciever reciever){  
 **actionEvent**.call(reciever);  
 }  
  
 **public interface** ActionEvent<T>{  
 **void** call(Reciever<T> reciever);  
 }  
}

Client.*create*(**new** Client.ActionEvent<Integer>() {  
 @Override  
 **public void** call(Reciever<Integer> reciever) {  
 reciever.onNext(100);  
 reciever.onNext(101);  
 }  
})  
.addReciever(**new** Reciever<Integer>(){  
 @Override  
 **public void** onNext(Integer t) {  
 System.***out***.println(t);  
 }  
  
 @Override  
 **public void** onError() {  
  
 }  
  
 @Override  
 **public void** onCompleted() {  
  
 }  
});

好了，我们已经近乎于相同Rxjava了。

接下来学习一下如何实现操作符，比如我们想要将发送过来的Integer转化为String，这里我们需要拿到之前onNext()方法里的参数，然后将其变为string，然后再发送出去。那到底怎么拿到onNext方法的参数呢？ 看一下我们之前的Reciever是怎么拿到参数的，其实还是在onNext方法里，那么我们就可以在onNext方法里拿到参数，然后再重新创建一个Reciever和ActionEvent发送出去, 其实就相当于两个Reciever，一个Reciever接收到信息后，处理一番再交给下一个Reciever，这个处理一番我们就给他叫做Transformer，转化的方式有很多种，因此依旧设为接口，里面包含一个方法即如何将接收到的信息转化为另一种形式：

**public interface** Transformer<T,R>{  
 R call(T from);  
}

添加完这个接口之后，我们写一个addTransformer，由于是建造者模式，我们需要重新创建一个Client对象：

**public** <R> Client<R> addTransformer(Transformer<T,R> transformer){  
 **return** *create*(**new** ActionEvent<R>() {  
 @Override  
 **public void** call(Reciever<R> reciever) {  
   
 }  
 })  
}

写到这里我们需要整理一下了，这里的ActionEvent是哪个Event，里面的Reciever又是哪个Reciever。这个Event是在addTransformer里的，所以应该是我们需要发送给下一个Reciever的Event，这个Reciever下一个接收的Reciever。那么我们在发送之前需要将其转化好：

**另一个问题来了，上一个发送的Event我们怎么接收到？**

我们直接在这里面创建一个Reciever来接收上一个Event, 这是完整的代码：

**public class** Client<T> {  
 ActionEvent<T> **actionEvent**;  
  
 **public** Client(ActionEvent<T> actionEvent) {  
 **this**.**actionEvent** = actionEvent;  
 }  
  
 **public static** <T> Client<T> create(ActionEvent<T> e) {  
 **return new** Client<T>(e);  
 }  
  
 **void** addReciever(Reciever reciever) {  
 **actionEvent**.call(reciever);  
 }  
  
 **public** <R> Client<R> addTransformer(Transformer<T, R> transformer) {  
 **return** *create*(**new** ActionEvent<R>() {  
 @Override  
 **public void** call(Reciever<R> reciever) {  
 addReciever(**new** Reciever<T>() {  
 @Override  
 **public void** onNext(T t) {  
 reciever.onNext(transformer.call(t));  
 }  
  
 @Override  
 **public void** onError() {  
  
 }  
  
 @Override  
 **public void** onCompleted() {  
  
 }  
 });  
 }  
 });  
 }  
  
 **public interface** ActionEvent<T> {  
 **void** call(Reciever<T> reciever);  
 }  
  
 **public interface** Transformer<T, R> {  
 R call(T from);  
 }  
}

**public class** Main {  
 **public static void** main(String[] args) {  
  
  
 Client.*create*(**new** Client.ActionEvent<Integer>() {  
 @Override  
 **public void** call(Reciever<Integer> reciever) {  
 reciever.onNext(100);  
 reciever.onNext(101);  
 }  
 }).addTransformer(**new** Client.Transformer<Integer,String>(){  
 @Override  
 **public** String call(Integer from) {  
 **return** String.*valueOf*(**"string"**+from);  
 }  
 })  
 .addReciever(**new** Reciever<String >(){  
 @Override  
 **public void** onNext(String t) {  
 System.***out***.println(t);  
 }  
  
 @Override  
 **public void** onError() {  
  
 }  
  
 @Override  
 **public void** onCompleted() {  
  
 }  
 });  
 }  
}

我们来走一遍流程：

首先是Client.create()方法创建了一个事件，这个事件指明了下一个创建的Reciever要做的事情

接着我们调用了addTransformer方法，并定义了tranform的形式，即将Integer转化为String，在这个方法里，我们添加了一个新的Reciever，这个Reciever的onNext的方法会调用call将Integer转化为String，这个过程是立刻执行的，一旦addReciever我们就会立刻执行Event的事件。

转化完成之后，我们又定义了一个新的Event，这个Event会得到上一步所转化的String，然后定义下一个Reciever的行为。

最后添加新的Reciever执行第二个Event里的内容。

## Scheduler

接下来该实现一个能进行线程切换的功能了，来看一下最终我们要实现的效果：

Client.*create*(**new** Client.ActionEvent<Integer>() {  
 @Override  
 **public void** call(Reciever<Integer> reciever) {  
 reciever.onNext(100);  
 reciever.onNext(101);  
 }  
**}).subscribeOn(Schedulers.io())**.addReciever(**new** Reciever<String >(){  
 @Override  
 **public void** onNext(String t) {  
 System.***out***.println(t);  
 }  
  
 @Override  
 **public void** onError() {  
  
 }  
  
 @Override  
 **public void** onCompleted() {  
  
 }  
});

通过subscribeOn(Schedulers.io())来使得Reviever在IO线程来执行，那么同样我们也可以使其在其他种类线程中执行。

怎么实现呢?

这里我们需要引入一个新的类**Executor**

**为什么引入Executor线程池框架**

new Thread()的缺点

每次new Thread()耗费性能

调用new Thread()创建的线程缺乏管理，被称为野线程，而且可以无限制创建，之间相互竞争，会导致过多占用系统资源导致系统瘫痪。

不利于扩展，比如如定时执行、定期执行、线程中断

**采用线程池的优点**

重用存在的线程，减少对象创建、消亡的开销，性能佳

可有效控制最大并发线程数，提高系统资源的使用率，同时避免过多资源竞争，避免堵塞

提供定时执行、定期执行、单线程、并发数控制等功能

其内部使用了线程池机制，它在java.util.cocurrent 包下，通过该框架来控制线程的启动、执行和关闭，可以简化并发编程的操作，还有关键的一点：有助于避免this逃逸问题——如果我们在构造器中启动一个线程，因为另一个任务可能会在构造器结束之前开始执行，此时可能会访问到初始化了一半的对象用Executor在构造器中。

因此最终我们是通过Excuter.excute来执行任务

那么我们先写一个Scheduler来作为manager来管理线程吧

**package** rxjava;  
  
*/\*\*  
 \** ***@author*** *jikangwang  
 \*/***public class** Client<T> {  
 ActionEvent<T> **actionEvent**;  
  
 **public** Client(ActionEvent<T> actionEvent) {  
 **this**.**actionEvent** = actionEvent;  
 }  
  
 **public static** <T> Client<T> create(ActionEvent<T> e) {  
 **return new** Client<T>(e);  
 }  
  
 **void** addReciever(Reciever reciever) {  
 **actionEvent**.call(reciever);  
 }  
  
 **public** Client<T> observeOn(Scheduler scheduler){  
 **return** Client.*create*(**new** ActionEvent<T>() {  
 @Override  
 **public void** call(Reciever<T> reciever) {  
 Scheduler.Worker worker=scheduler.createWorker();  
 **actionEvent**.call(**new** Reciever<T>() {  
 @Override  
 **public void** onNext(T t) {  
 worker.schedule(**new** Runnable() {  
 @Override  
 **public void** run() {  
 reciever.onNext(t);  
 }  
 });  
 }  
  
 @Override  
 **public void** onError() {  
  
 }  
  
 @Override  
 **public void** onCompleted() {  
  
 }  
 });  
 }  
 });  
 }  
  
 **public** Client<T> subscribeOn(Scheduler scheduler) {  
 **return** Client.*create*(**new** ActionEvent<T>() {  
 @Override  
 **public void** call(Reciever<T> reciever) {  
 scheduler.createWorker().schedule(**new** Runnable() {  
 @Override  
 **public void** run() {  
 **actionEvent**.call(reciever);  
 }  
 });  
 }  
 });  
 }  
 **public** <R> Client<R> addTransformer(Transformer<T, R> transformer) {  
 **return** *create*(**new** ActionEvent<R>() {  
 @Override  
 **public void** call(Reciever<R> reciever) {  
 addReciever(**new** Reciever<T>() {  
 @Override  
 **public void** onNext(T t) {  
 reciever.onNext(transformer.call(t));  
 }  
  
 @Override  
 **public void** onError() {  
  
 }  
  
 @Override  
 **public void** onCompleted() {  
  
 }  
 });  
 }  
 });  
 }  
  
 **public interface** ActionEvent<T> {  
 **void** call(Reciever<T> reciever);  
 }  
  
 **public interface** Transformer<T, R> {  
 R call(T from);  
 }  
}

**package** rxjava;  
  
**import** java.util.concurrent.Executors;  
  
*/\*\*  
 \** ***@author*** *jikangwang  
 \*/***public class** Schedulers {  
 **private static final** Scheduler ***ioScheduler*** = **new** Scheduler(Executors.*newSingleThreadExecutor*());  
 **public static** Scheduler io() {  
 **return *ioScheduler***;  
 }  
  
}

**package** rxjava;  
  
**import** java.util.concurrent.Executor;  
  
*/\*\*  
 \** ***@author*** *jikangwang  
 \*/***public class** Scheduler {  
 **final** Executor **executor**;  
 **public** Scheduler(Executor executor) {  
 **this**.**executor** = executor;  
 }  
 **public** Worker createWorker() {  
 **return new** Worker(**executor**);  
 }  
 **public static class** Worker {  
 **final** Executor **executor**;  
 **public** Worker(Executor executor) {  
 **this**.**executor** = executor;  
 }  
 **public void** schedule(Runnable runnable) {  
 **executor**.execute(runnable);  
 }  
 }  
}

# Fragment & Activity & LifeCycle

Fragment真的算坑中之坑了，一堆bug，这次直接看源码，彻底搞懂fragment

## LifeCycle

抽象类

**public abstract void** addObserver(@NonNull LifecycleObserver observer);

**public enum** Event {  
 */\*\*  
 \* Constant for onCreate event of the {****@link*** *LifecycleOwner}.  
 \*/* ***ON\_CREATE***,  
 */\*\*  
 \* Constant for onStart event of the {****@link*** *LifecycleOwner}.  
 \*/* ***ON\_START***,  
 。。。

}

**public enum** State {***DESTROYED***,***INITIALIZED***,

算是观察者模式的一种体现吧，里面通过枚举定义了各种事件和状态，事件的发生会导致状态的改变

## LifecycleOwner

比较简单的类了，实现它的类就等于标记其有生命周期

**public interface** LifecycleOwner {  
 */\*\*  
 \* Returns the Lifecycle of the provider.  
 \*  
 \** ***@return*** *The lifecycle of the provider.  
 \*/* @NonNull  
 Lifecycle getLifecycle();  
}

## LifecycleRegistry

这个类是Lifecycle类的实现，主要处理了和lifecycle相关的事情，例如添加观察者，生命周期的调整

**private** FastSafeIterableMap<LifecycleObserver, LifecycleRegistry.ObserverWithState> mObserverMap = **new** FastSafeIterableMap();  
**private** State mState;  
**private final** WeakReference<LifecycleOwner> mLifecycleOwner;

**private** ArrayList<State> mParentStates = **new** ArrayList();

观察者的状态，当前状态，以及和Lifecycle绑定的owner，parentstate是用来记录历史state的

/\*\*

\* 添加LifecycleObserver观察者，并将之前的状态分发给这个Observer,例如我们在onResume之后注册这个Observer，

\* 该Observer依然能收到ON\_CREATE事件

\*/

**public void** addObserver(@NonNull LifecycleObserver observer) {  
 State initialState = **this**.mState == State.DESTROYED ? State.DESTROYED : State.INITIALIZED;   
 **if** (previous == **null**) {  
 LifecycleOwner lifecycleOwner = (LifecycleOwner)**this**.mLifecycleOwner.get();  
 **if** (lifecycleOwner != **null**) {  
 **while**(statefulObserver.mState.compareTo(targetState) < 0 && **this**.mObserverMap.contains(observer)) {  
 **this**.pushParentState(statefulObserver.mState);  
 statefulObserver.dispatchEvent(lifecycleOwner, upEvent(statefulObserver.mState));  
 **this**.popParentState();  
 targetState = **this**.calculateTargetState(observer);  
 }  
  
 **if** (!isReentrance) {  
 **this**.sync();  
 }  
  
 --**this**.mAddingObserverCounter;  
 }  
 }  
}

## Fragment

FragmentManager fragmentManager=getSupportFragmentManager();  
FragmentTransaction fragmentTransaction=fragmentManager.beginTransaction();  
fragmentTransaction.replace(R.id.***main\_container***,**fragment**);  
fragmentTransaction.addToBackStack(**null**);  
fragmentTransaction.commit();

先说一个大概的执行流程：首先创建一个fragmentmanager，这个manager是管理fragment的，然后创建一个Transaction，这个Transaction代表Fragment的行为，通过replace，addtobackstack都是改变transaction里的变量，最后commit会将transaction放到manager里的一个list里，之后会发送给Host里的handler，handler来负责执行，执行的具体是由manager里的execute方法实现的，大概就是根据transaction里的变量进行一些操作，例如根据最后会通过moveState方法来对fragment进行创建，走生命周期。

添加一个fragment的流程：

1. 首先调用getSupportFragmentManager()，会进入**FragmentActivity.Class**

**final** FragmentController mFragments = FragmentController.createController(**new** FragmentActivity.HostCallbacks());

**public** FragmentManager getSupportFragmentManager() {  
 **return this**.mFragments.getSupportFragmentManager();  
}

会调用FragmentActivity.class里的FragmentController里的方法，这个**FragmentController**是啥玩意？

Provides integration points with a [FragmentManager](https://developer.android.com/reference/android/support/v4/app/FragmentManager.html) for a fragment host.

It is the responsibility of the host to take care of the Fragment's lifecycle. The methods provided by [FragmentController](https://developer.android.com/reference/android/support/v4/app/FragmentController.html) are for that purpose.

意思就是Fragment的host会通过Controller来管理Fragment的生命周期, 果然方法里包含一堆dispatch方法，估计是Activity生命周期在走的时候，会通过这里面的方法通知Fragment

**private final** FragmentHostCallback<?> mHost;

**public** FragmentManager getSupportFragmentManager() {  
 **return this**.mHost.getFragmentManagerImpl();  
}

**public void** dispatchCreate() {  
 **this**.mHost.mFragmentManager.dispatchCreate();  
}  
  
**public void** dispatchActivityCreated() {  
 **this**.mHost.mFragmentManager.dispatchActivityCreated();  
}

1. 接下来会调用FragmentHostCallback里的getFragmentManagerImpl方法，这个**FragmentHostCallback**又是啥玩意？

Integration points with the Fragment host.

Fragments may be hosted by any object; such as an [Activity](https://developer.android.com/reference/android/app/Activity.html). In order to host fragments, implement [FragmentHostCallback](https://developer.android.com/reference/android/support/v4/app/FragmentHostCallback.html), overriding the methods applicable to the host.

这里其实和上面一样也只是个过渡，，上面的方法不过是通过mHost.mFragmentManager来实现，这里就是为其提供Fragmentmanager看一下里面具体的方法：

**private final** Activity mActivity;  
**private final** Context mContext;  
**private final** Handler mHandler;  
**final** FragmentManagerImpl mFragmentManager;

**final** FragmentManagerImpl **mFragmentManager** = **new** FragmentManagerImpl();

FragmentManagerImpl getFragmentManagerImpl() {  
 **return this**.mFragmentManager;  
}

**public** View onFindViewById(**int** id) {  
 **return null**;  
}//??

里面持有了Fragment host的引用，看来Fragment和activity就是最终在这产生了联系,

**FragmentManagerImpl.class**

好了，终于找到了熟悉的FragmentManager了，让我们细细剖析一下这里面到底写了啥

ArrayList<OpGenerator> **mPendingActions**;  
**boolean mExecutingActions**;

这个应该就是代表Actions还没执行的List

**final** ArrayList<Fragment> **mAdded** = **new** ArrayList<>();  
SparseArray<Fragment> **mActive**;  
ArrayList<BackStackRecord> **mBackStack**;

存放被管理的fragment和回退栈

ArrayList<OnBackStackChangedListener> **mBackStackChangeListeners**;

**int mCurState** = Fragment.***INITIALIZING***;  
FragmentHostCallback **mHost**;  
FragmentContainer **mContainer**;  
Fragment **mParent**;

一些listener，Fragment生命周期状态，与宿主关系

Runnable **mExecCommit** = **new** Runnable() {  
 @Override  
 **public void** run() {  
 execPendingActions();  
 }  
};

果然, transactions是被异步执行的

**public** FragmentTransaction beginTransaction() {  
 **return new** BackStackRecord(**this**);  
}

似乎看到了一个熟悉的方法☺

//大概就是检查这个FragmentManager能不能正常工作

**private void** checkStateLoss() {  
 **if** (isStateSaved()) {  
 **throw new** IllegalStateException(  
 **"Can not perform this action after onSaveInstanceState"**);  
 }  
 **if** (**mNoTransactionsBecause** != **null**) {  
 **throw new** IllegalStateException(  
 **"Can not perform this action inside of "** + **mNoTransactionsBecause**);  
 }  
}

@Override  
**public boolean** executePendingTransactions() {  
 **boolean** updates = execPendingActions();  
 forcePostponedTransactions();  
 **return** updates;  
}  
  
@Override  
**public void** popBackStack() {  
 enqueueAction(**new** PopBackStackState(**null**, -1, 0), **false**);  
}  
  
@Override  
**public boolean** popBackStackImmediate() {  
 checkStateLoss();  
 **return** popBackStackImmediate(**null**, -1, 0);  
}

//嗯。。原来是吧action放进arraylist里等待执行了啊

**public void** enqueueAction(OpGenerator action, **boolean** allowStateLoss) {  
 **if** (!allowStateLoss) {  
 checkStateLoss();  
 }  
 **synchronized** (**this**) {  
 **if** (**mDestroyed** || **mHost** == **null**) {  
 **if** (allowStateLoss) {  
 *// This FragmentManager isn't attached, so drop the entire transaction.* **return**;  
 }  
 **throw new** IllegalStateException(**"Activity has been destroyed"**);  
 }  
 **if** (**mPendingActions** == **null**) {  
 **mPendingActions** = **new** ArrayList<>();  
 }  
 **mPendingActions**.add(action);  
 scheduleCommit();  
 }  
}

*/\*\*  
 \* Schedules the execution when one hasn't been scheduled already. This should happen  
 \* the first time {****@link*** *#enqueueAction(OpGenerator, boolean)} is called or when  
 \* a postponed transaction has been started with  
 \* {****@link*** *Fragment#startPostponedEnterTransition()}  
 \*/***private void** scheduleCommit() {  
 **synchronized** (**this**) {  
 **boolean** postponeReady =  
 **mPostponedTransactions** != **null** && !**mPostponedTransactions**.isEmpty();  
 **boolean** pendingReady = **mPendingActions** != **null** && **mPendingActions**.size() == 1;  
 **if** (postponeReady || pendingReady) {  
 **mHost**.getHandler().removeCallbacks(**mExecCommit**);  
 **mHost**.getHandler().post(**mExecCommit**);  
 }  
 }  
}

嗯。。不是很懂，又remove又添加，啥意思？这里就是通过Host的handler来添加Action，但这里很重要，所有即将被执行的Action会被放到queue里，然后会通过handler由host执行，这里发送过去的mExecCommit就是执行命令，一旦Host所在线程的messagequeue执行到这了，fragment的一切action就会开始执行，然后会继续执行moveToState方法，就是显示Fragment之类的

**void** moveToState(Fragment f, **int** newState, **int** transit, **int** transitionStyle,  
 **boolean** keepActive) {

**public boolean** execPendingActions() {  
 ensureExecReady(**true**);  
  
 **boolean** didSomething = **false**;  
 **while** (generateOpsForPendingActions(**mTmpRecords**, **mTmpIsPop**)) {  
 **mExecutingActions** = **true**;  
 **try** {  
 removeRedundantOperationsAndExecute(**mTmpRecords**, **mTmpIsPop**);  
 } **finally** {  
 cleanupExec();  
 }  
 didSomething = **true**;  
 }  
  
 doPendingDeferredStart();  
 burpActive();  
  
 **return** didSomething;  
}

那么具体是怎么执行命令的呢

// add这么简单，pop这么麻烦？

**void** addBackStackState(BackStackRecord state) {  
 **if** (**mBackStack** == **null**) {  
 **mBackStack** = **new** ArrayList<BackStackRecord>();  
 }  
 **mBackStack**.add(state);  
}  
  
@SuppressWarnings(**"unused"**)  
**boolean** popBackStackState(ArrayList<BackStackRecord> records, ArrayList<Boolean> isRecordPop,  
 String name, **int** id, **int** flags) {  
 **if** (**mBackStack** == **null**) {  
 **return false**;  
 }  
 **if** (name == **null** && id < 0 && (flags & ***POP\_BACK\_STACK\_INCLUSIVE***) == 0) {  
 **int** last = **mBackStack**.size() - 1;  
 **if** (last < 0) {   
 **return false**;  
 }  
 records.add(**mBackStack**.remove(last)); //卧槽，这段真是鬼才写法  
 isRecordPop.add(**true**);  
 } **else** {  
 **int** index = -1;  
 **if** (name != **null** || id >= 0) {  
 *// If a name or ID is specified, look for that place in  
 // the stack.* index = **mBackStack**.size()-1;  
 **while** (index >= 0) {  
 BackStackRecord bss = **mBackStack**.get(index);  
 **if** (name != **null** && name.equals(bss.getName())) {  
 **break**;  
 }  
 **if** (id >= 0 && id == bss.**mIndex**) {  
 **break**;  
 }  
 index--;  
 }  
 **if** (index < 0) {  
 **return false**;  
 }  
 **if** ((flags&***POP\_BACK\_STACK\_INCLUSIVE***) != 0) {  
 index--;  
 *// Consume all following entries that match.* **while** (index >= 0) {  
 BackStackRecord bss = **mBackStack**.get(index);  
 **if** ((name != **null** && name.equals(bss.getName()))  
 || (id >= 0 && id == bss.**mIndex**)) {  
 index--;  
 **continue**;  
 }  
 **break**;  
 }  
 }  
 }  
 **if** (index == **mBackStack**.size()-1) {  
 **return false**;  
 }  
 **for** (**int** i = **mBackStack**.size() - 1; i > index; i--) {  
 records.add(**mBackStack**.remove(i));  
 isRecordPop.add(**true**);  
 }  
 }  
 **return true**;  
}

看下来还是比较简单的，就是根据提供的信息在backstack里找，找到就删，然后做记录，这里不管是backstack还是record都存放了BackStackRecord，这个BackStackRecord又大有看头了

*/\*\*  
 \* Entry of an operation on the fragment back stack.  
 \*/***final class** BackStackRecord **extends** FragmentTransaction **implements** FragmentManager.BackStackEntry, FragmentManagerImpl.OpGenerator {

原来BackStackRecord就是FragmentTransaction啊，就相当于一次对Fragment操作的记录

**final** FragmentManagerImpl **mManager**;  
  
**static final int *OP\_NULL*** = 0;  
**static final int *OP\_ADD*** = 1;  
**static final int *OP\_REPLACE*** = 2;  
**static final int *OP\_REMOVE*** = 3;  
**static final int *OP\_HIDE*** = 4;

**static final class** Op {  
 **int cmd**;  
 Fragment **fragment**;  
 **int enterAnim**;  
 **int exitAnim**;  
 **int popEnterAnim**;  
 **int popExitAnim**;  
  
 Op() {  
 }  
  
 Op(**int** cmd, Fragment fragment) {  
 **this**.**cmd** = cmd;  
 **this**.**fragment** = fragment;  
 }  
}

Op就是operation了

ArrayList<Op> **mOps** = **new** ArrayList<>();

@Override  
**public** FragmentTransaction add(**int** containerViewId, Fragment fragment, @Nullable String tag) {  
 doAddOp(containerViewId, fragment, tag, ***OP\_ADD***);  
 **return this**;  
}  
  
**private void** doAddOp(**int** containerViewId, Fragment fragment, @Nullable String tag, **int** opcmd) {  
  
 fragment.**mFragmentManager** = **mManager**;  
 fragment.**mTag** = tag;  
 fragment.**mContainerId** = fragment.**mFragmentId** = containerViewId;  
 addOp(**new** Op(opcmd, fragment));  
}

**void** addOp(Op op) {  
 **mOps**.add(op);  
 op.**enterAnim** = **mEnterAnim**;  
 op.**exitAnim** = **mExitAnim**;  
 op.**popEnterAnim** = **mPopEnterAnim**;  
 op.**popExitAnim** = **mPopExitAnim**;  
}

好了，顺便看一下transcation的add方法，其实就是给这个fragment赋上一些属性，然后将记录保存在Transaction的Op里

@Override  
**public** FragmentTransaction replace(**int** containerViewId, Fragment fragment) {  
 **return** replace(containerViewId, fragment, **null**);  
}  
  
@Override  
**public** FragmentTransaction replace(**int** containerViewId, Fragment fragment,  
 @Nullable String tag) {  
 **if** (containerViewId == 0) {  
 **throw new** IllegalArgumentException(**"Must use non-zero containerViewId"**);  
 }  
  
 doAddOp(containerViewId, fragment, tag, ***OP\_REPLACE***);  
 **return this**;  
}

Replace和add方法区别不大，主要是最后opcmd方法有区别，Op里的cmd就是表示到底这是add还是replace方法

@Override  
**public** FragmentTransaction remove(Fragment fragment) {  
 addOp(**new** Op(***OP\_REMOVE***, fragment));  
  
 **return this**;  
}  
  
@Override  
**public** FragmentTransaction hide(Fragment fragment) {  
 addOp(**new** Op(***OP\_HIDE***, fragment));  
  
 **return this**;  
}

同样，不管是add，replace，hide，show方法，其实都只是给Op里的cmd赋上了不同的值，并不是立刻执行，应该是后期某个方法需要读取Op里cmd，然后进行不同的操作

@Override  
**public** FragmentTransaction addToBackStack(@Nullable String name) {  
 **mAddToBackStack** = **true**;  
 **mName** = name;  
 **return this**;  
}

@Override  
**public int** commit() {  
 **return** commitInternal(**false**);  
}

@Override  
**public void** commitNow() {  
 disallowAddToBackStack();  
 **mManager**.execSingleAction(**this**, **false**);  
}

**int** commitInternal(**boolean** allowStateLoss) {  
 **if** (**mCommitted**) **throw new** IllegalStateException(**"commit already called"**);  
 **mCommitted** = **true**;  
 **if** (**mAddToBackStack**) {  
 **mIndex** = **mManager**.allocBackStackIndex(**this**);  
 } **else** {  
 **mIndex** = -1;  
 }  
 **mManager**.enqueueAction(**this**, allowStateLoss);  
 **return mIndex**;  
}

哦，commit方法，原来就是把这个transaction加入到manager里的list里等待执行，那么commit方法执行过后就真的不一定是立刻执行的了，并且这个方法还会返回这个transaction在list里的index，这里还有commitNow方法，可以立刻执行。好了Transaction方法基本上就是这些内容，我们回过头继续看Manager方法

**public void** addFragment(Fragment fragment, **boolean** moveToStateNow) {  
 **if** (*DEBUG*) Log.*v*(***TAG***, **"add: "** + fragment);  
 makeActive(fragment);  
 **if** (!fragment.**mDetached**) {  
 **if** (**mAdded**.contains(fragment)) {  
 **throw new** IllegalStateException(**"Fragment already added: "** + fragment);  
 }  
 **synchronized** (**mAdded**) {  
 **mAdded**.add(fragment);  
 }  
 fragment.**mAdded** = **true**;  
 fragment.**mRemoving** = **false**;  
 **if** (fragment.**mView** == **null**) {  
 fragment.**mHiddenChanged** = **false**;  
 }  
 **if** (fragment.**mHasMenu** && fragment.**mMenuVisible**) {  
 **mNeedMenuInvalidate** = **true**;  
 }  
 **if** (moveToStateNow) {  
 moveToState(fragment);  
 }  
 }  
}

会将当前fragment加入到mAdded里

@Override  
**public boolean** generateOps(ArrayList<BackStackRecord> records, ArrayList<Boolean> isRecordPop) {  
 **if** (FragmentManagerImpl.*DEBUG*) {  
 Log.*v*(***TAG***, **"Run: "** + **this**);  
 }  
  
 records.add(**this**);  
 isRecordPop.add(**false**);  
 **if** (**mAddToBackStack**) {  
 **mManager**.addBackStackState(**this**);  
 }  
 **return true**;  
}

这里是Action被具体执行的地方

Fragment遇到的一些问题：

1. **Fragment打开以后会显示在button下面：**

<**FrameLayout  
 android:id="@+id/fragment"  
   
 android:layout\_width="match\_parent"  
 android:layout\_height="match\_parent"**>  
 <**Button  
 android:layout\_gravity="center"  
 android:id="@+id/button"  
 android:layout\_width="wrap\_content"  
 android:layout\_height="wrap\_content"** />  
</**FrameLayout**>

## BroadCast