**1 构造函数分析**

首先我们来看构造函数，构造函数有很多个重载版本，无非**就是无参的调用有参的，少参的调用多参的**，这是很常用的一个套路，比如View的构造函数就是这样，这样的好处就是，只需要在最多参的构造函数中 调用一次初始化操作就可以，其他少参的或无参的也会初始化到。现在我们来看最多参的构造即可:

public Handler(@NonNull Looper looper, @Nullable Callback callback, boolean async) {

mLooper = looper;

mQueue = looper.mQueue;

mCallback = callback;

mAsynchronous = async;

}

mLooper: 一个Looper，用来对消息进行循环检测，有消息就取出处理，没有就等待 mQueue: 一个MessageQueue，是一个队列，用来保存Message mCallback: 一个callback，用于触发Handler的回调，下面是他的定义

public interface Callback {

boolean handleMessage(@NonNull Message msg);

}

isAsynchronous:一个boolean值，用来定义消息是否是异步的，这个后面再讲。 好，现在我们来看第一个参数looper从哪里来，点击Looper.getMainLooper()，可以看到:

public static Looper getMainLooper() {

synchronized (Looper.class) {

return sMainLooper;

}

}

然后看sMainLooper被赋值的地方:

public static void prepareMainLooper() {

prepare(false);

synchronized (Looper.class) {

if (sMainLooper != null) {

throw new IllegalStateException("The main Looper has already been prepared.");

}

sMainLooper = myLooper();

}

}

这里先调用perpare(false)，然后做个初始化判断，如果已经初始化过了，就抛出异常，然后对sMainLooper进行赋值，好，我们先不看prepare()，只看myLooper():

public static @Nullable Looper myLooper() {

return sThreadLocal.get();

}

就这一句，既然有get()，肯定有set()，我们找找这个sThreadLocal在哪里set()的:

static final ThreadLocal<Looper> sThreadLocal = new ThreadLocal<Looper>();

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));

}

哦，原来在prepare()里面set的，这就对上了，在prepareMainLooper()里面，先使用prepare(false)，创建一个Looper然后使用sThreadLocal.set(looper)，保存到sThreadLocal中， 然后使用sThreadLocal.get()赋值给sMainLooper。这里有个问题:为什么不这样写:

sMainLooper = new Looper(quitAllowed);

多简单！！！而非要:

sThreadLocal.set(new Looper(quitAllowed));

sMainLooper = sThreadLocal.get();

这个问题我们先记录一下：**问题1 为什么Android要把Looper存在ThreadLocal里面？**，先回到我们的主线，我们现在绕了一大圈，找到了第一个Handler的第一次个参数mLooper在这里被new出来了， 我们看看new Looper(quitAllowed);做了什么:

private Looper(boolean quitAllowed) {

mQueue = new MessageQueue(quitAllowed);

mThread = Thread.currentThread();

}

第一：创建了一个MessageQueue()；第二：保存了当前线程。 那这个线程是哪个线程啊？就是new Looper()的那个线程啊。 new Looper()是在prepare()中调用的，prepare()是在prepareMainLooper()中调用的，那就是说这个线程就是prepareMainLooper()被调用的线程，那prepareMainLooper()在哪里被调用的? 哦，在这里，ActivityThread的main方法中:

public static void main(String[] args) {

...

省略若干代码

...

Looper.prepareMainLooper();

...

省略若干代码

...

Looper.loop();

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

}

这个ActivityThread.main()又是在哪里被调用的？这个是在当前进程被创建的时候调用的，也就是Android应用进程的入口，这个后面再说，现在知道了 Looper中的线程就是这个线程，这个线程是谁呢？就是UI线程，也就是主线程！可以先留个问题：**问题2 ActivityThread.main()在哪里被调用?** 好，现在我们来总结一下：我们找到了Handler的第一个参数Looper被new出来了，然后他保存了当前的线程，然后还new出来一个MessageQueue()，那么我们看看MessageQueue()的构造吧:

MessageQueue(boolean quitAllowed) {

mQuitAllowed = quitAllowed;

mPtr = nativeInit();

}

就保存了一个参数，然后调用了一个nativeInit()方法，Handler的第一个参数mLooper已经初始化了，现在我们来看第二个参数mQueue:

mQueue= looper.mQueue;

就是把looper的mQueue拿过来保存一下，那第三个参数callback和第四个参数呢，都是保存一下ok了，好，Handler的构造函数完毕。说白了就是： 1 创建Looper并保存 2 创建MessageQueue并保存

**2 调用流程分析**

我们怎么使用Handler的呢？很多方法，要么: 1 handler.sendMessage(message);要么:2 handler.post(runnable)，当然还有其他的使用方法，其实道理都是一样的，我们来看最普遍的用法: handler.sendMessage(); 这里先放一个Message的简略类信息:

public final class Message implements Parcelable {

public int what; //不BB了，都知道的

public long when; //指定此message执行的时间，到了when这个时间就需要执行

int flags; //用于控制状态

Handler target; //被哪个Handler执行

Runnable callback; //回调

Message next; //指向下一个Message

\*\*\*省略其他成员变量\*\*\*

}

点进去看:

public final boolean sendMessage(@NonNull Message msg) {

return sendMessageDelayed(msg, 0); //这里直接把0当作delay传入

}

public final boolean sendMessageDelayed(@NonNull Message msg, long delayMillis) {

if (delayMillis < 0) {

delayMillis = 0;

}

return sendMessageAtTime(msg, SystemClock.uptimeMillis() + delayMillis); //这里用当前时间+需要延迟的时间传入

}

public boolean sendMessageAtTime(@NonNull Message msg, long uptimeMillis) {

MessageQueue queue = mQueue;

if (queue == null) {

RuntimeException e = new RuntimeException(this + " sendMessageAtTime() called with no mQueue");

Log.w("Looper", e.getMessage(), e);

return false;

}

return enqueueMessage(queue, msg, uptimeMillis); // 这个queue就是第一步初始化Looper时候的那个

}

我们只看sendMessageAtTime(msg, uptimeMillis)即可，其实就是一句话: enqueueMessage(queue, msg, uptimeMillis);继续看:

private boolean enqueueMessage(@NonNull MessageQueue queue, @NonNull Message msg,long uptimeMillis) {

msg.target = this; //将msg.target赋值为当前Handler

if (mAsynchronous) { //这个我知道，Handler构造函数的第四个参数

msg.setAsynchronous(true); // 这tm是啥意思，算了不管了

}

return queue.enqueueMessage(msg, uptimeMillis); //这queue是MessageQueue，就是Looper被new出来创建的那个

}

逻辑也很简单: msg.target = this，保存了当前Handler，然后又调用了MessageQueue的enqueueMessage()，好，我们看下这个函数:

boolean enqueueMessage (Message msg,long when){ //这里的when指的是要执行的时间

\*\*\*省略部分代码\*\*\*

synchronized (this) {

msg.markInUse();

msg.when = when; //保存要执行的时间，系统时间在when的时候，需要执行这个msg

Message p = mMessages; //这是一个成员变量: Message mMessages，保存当前的消息队列的队头

boolean needWake; //是否需要唤醒

if (p == null || when == 0 || when < p.when) {

//我们看下这个if条件:

//1 p == null就意味着 mMessages == null; 也就是说队头是null，也就是说本次enqueue的msg是第一个消息

//2 when == 0，这个不可能啊，when是怎么来的，我们回忆一下，是在sendMessageAtTime()里面，用当前时间加上delay的延迟时间，怎么会是0，除非故意传递0，很可能系统

//内部调用的，先不看

//3 when < p.when，那就是msg.when < mMessages.when，也就是说这个msg应该在队头执行之前执行的

//上面条件都说明：这个msg需要在队头之前执行，那是不是要插入到队头呢，看下面代码:

msg.next = p; //嗯，这个msg的next是p

mMessages = msg; //然后队头指向了这个msg，哦，明白了，就是说这个msg是第一个消息，就插到队头，ok了解

needWake = mBlocked; //如果当前阻塞了，需要唤醒，唤醒啥？后面再说，反正记住要唤醒就行

//\*\*FLAG1\*\* 看过MessageQueue.next()再回来看这里!!! 好，我们回来了，刚刚分析了一波如果没有消息需要处理就阻塞，现在这里明显有消息需要处理了，那么如果阻塞，就需要唤醒，

//所以needWake = mBlocked; 阻塞了就需要唤醒，否则不需要。好，去看FLAG2

} else {

//这里就是不需要插到队头的了

needWake = mBlocked && p.target == null && msg.isAsynchronous(); // 这个条件不用看了，系统使用的，就当false就行，以后分析

//此处往下是重点

Message prev;

for (; ; ) {

prev = p; //prev指向队头

p = p.next; //队头指向下一个

if (p == null || when < p.when) {

//1 p == null，除非是最后一个message了，也就是遍历完了

//2 when < p.when，那就是说这个msg在p之前执行了

break;

}

if (needWake && p.isAsynchronous()) { //这个先不用看了，系统使用的，就当false就行，以后分析

needWake = false;

}

}

msg.next = p; //遍历结束，msg在p之前执行，就将msg插入到p前面，所以msg.next = p;

prev.next = msg; //prev.next本来是p的，现在指向了msg，那就是将msg插入到prev和p之间了

//通过上面分析，可以看懂messageQueue是按照when的大小排序的，越早执行的越靠前

}

if (needWake) { //如果需要唤醒，就唤醒，目前我们看到的，只有在if里面才可能为true，也就是needWake = mBlocked，后面再分析mBlocked

nativeWake(mPtr);

//\*\*FLAG2\*\* ok我们回来了，如果next()里没有return msg；那么就会被阻塞，也就是会调用nativePollOnce()阻塞，

//那么此时有消息需要处理就使用nativeWake()来唤醒了，此时我们反复看enqueueMessage()这个函数和next()对比，就发现是成对的，一个存放msg，一个取出msg，

//然后通过Looper.loop()来检测，很好的机制

}

}

return true;

}

这里删除了部分代码，我们只看关键部分，仔细看注释，我们发现，整个流程就是将msg按照执行时间顺序插入到队列，越早执行的越靠前，如果msg是第一条消息(p == null)， 或者需要立刻执行(when == 0||when < p.when)，就会立刻唤醒，调用nativeWake(mPtr)去唤醒native执行，那么这个nativeWake(mPtr)是什么玩意呢，这里可以先简单的 理解为java层的notify()即可。那么我们把消息存入到MessageQueue里面了，从哪里取出来使用呢，我们回忆一下我们第一步的Looper的构造，是在ActivityTrhead.main() 函数里面，里面就两句:

public static void main(String[] args) {

...

省略若干代码

...

Looper.prepareMainLooper();

...

省略若干代码

...

Looper.loop();

这里有个异常，看来是跑不到，说明Looper.loop()是个死循环

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

}

我们已经知道第一句：Looper.prepareMainLooper()是创建了Looper，并且创建了MessageQueue，而我们刚刚的handler.sendMessage(message)也在这个messageQueue里面 添加了message，现在就差取出来message使用了，肯定在Looper.loop()里面，好，我们看看，这里只列出部分关键代码:

public static void loop() {

final Looper me = myLooper();

if (me == null) {

throw new RuntimeException("No Looper; Looper.prepare() wasn't called on this thread.");

}

final MessageQueue queue = me.mQueue;

boolean slowDeliveryDetected = false;

for (;;) { //这里果然是个死循环，否则上面就会抛出异常了

Message msg = queue.next(); //从messageQueue中取出message，终于找到了，在这里取出消息

if (msg == null) {

return; //没有消息就返回，就退出了循环，等等！那不就意味着上面那个异常会发生吗，app就crash了，除非msg==null不会发生，等会看看这个queue.next()会不会返回null

}

try {

//我们前面说过msg.target就是handler.sendMessage(msg)的这个handler，那就是调用了这个handler的dispatchMessage(msg)，好我们看看

msg.target.dispatchMessage(msg);

} catch (Exception exception) {

throw exception;

} finally {

\*\*\*省略其他\*\*\*

}

msg.recycleUnchecked(); //回收

}

}

Handler.dispatchMessage()如下：

public void dispatchMessage(@NonNull Message msg) {

if (msg.callback != null) {

handleCallback(msg); //嗯，如果message有callback就直接调用

} else {

if (mCallback != null) {

if (mCallback.handleMessage(msg)) { //message没有callback就调用自己的mCallback了，也就是我们第一步的第三个参数

return;

}

}

handleMessage(msg); //最后才调用我们构造函数重载的handlerMessage(msg)方法，

}

}

上面逻辑很简单，这里总结一下 1 如果msg有callback，那么只调用这个 2 如果msg没有callback，先调用自己的mCallback(如果有的话)，如果mCallback的handleMessage(msg)返回true，就结束，否则调用Handler自己的handleMessage(msg) 分析完事，那么queue.next()会不会返回null呢，我们看看:

Message next() {

final long ptr = mPtr;

if (ptr == 0) {

return null; //这里有返回null!!! 卧槽，什么情况下发生?

}

int nextPollTimeoutMillis = 0; //记录下一条消息还有多长时间执行

for (;;) { //这里又是个死循环

...省略部分代码...

//这个可以理解为java的wait()，表示让native层wait(nextPollTimeoutMillis)时间，嗯，等待这么长时间

//还记得之前MessageQueue.enqueueMessage(msg)吗，里面的nativeWake(mPtr)就等价于noticy()，用来唤醒

nativePollOnce(ptr, nextPollTimeoutMillis);

synchronized (this) {

final long now = SystemClock.uptimeMillis(); //记录当前时间

Message prevMsg = null;

Message msg = mMessages; //取出第一条消息，也就是队头消息

//这个if不用看了，系统使用的，以后再讲

if (msg != null && msg.target == null) {

do {

prevMsg = msg;

msg = msg.next;

} while (msg != null && !msg.isAsynchronous());

}

//我们直接看这里

if (msg != null) { //msg!=null，也就是队头消息不为null，也就是队列非空

if (now < msg.when) {

//跑到这里表示当前时间小于第一条消息要执行的时间，也就是还没到执行的时候，msg是第一条消息，第一条消息还不到执行的时候，

//后面的消息都不用看了，因为前面分析过他们是按照执行时间从小到大排序的

//计算下一次执行还要多久，用msg.when - now算出来要等多久再取消息(这就是Handler能延时的原理)，Integer.MAX\_VALUE是为了防止应用层恶意传参数

nextPollTimeoutMillis = (int) Math.min(msg.when - now, Integer.MAX\_VALUE);

} else {

//跑到这里说明now >= msg.when，也就是说有消息要处理了

mBlocked = false; //那就解除阻塞

if (prevMsg != null) {

//这里不用看了，系统使用的，直接看else

prevMsg.next = msg.next;

} else {

//直接看这里，队头指向下一个消息

mMessages = msg.next;

}

msg.next = null; //这个msg的next赋值为null，那就是从队头删除了

msg.markInUse(); //标记为已使用

return msg; //返回，嗯，这个时候就回到looper.loop()里面了，去使用这个msg的targert.dispatchMessage(msg)了

}

} else {

//msg==null，那就是队列为null，直接赋值为-1，表示无限等待，那就只能等待enqueueMessage(msg)里面的nativeWake(mPtr)来唤醒了，

//也就是有消息才唤醒了

nextPollTimeoutMillis = -1;

}

mBlocked = true; //跑到这里，表示没有return msg;那么肯定是阻塞了，标记为true，好，现在去看FLAG1

...省略部分代码...

}

...省略处理空闲消息的代码...

}

}

FLAG3，好现在我们从FLAG2回来了，我们总结一下: Looper.loop()是个死循环，从Message.next()获取msg，然后Message.next()也是死循环，循环取出当前队头的message，如果到了msg的执行时间就返回，并且使用mBlocked = false 标记为非阻塞，如果不到执行时间就先计算还有多久执行:int delay = msg.when - now， 然后调用nativePollOnce(delay)等待，等待结束就会执行，如果没有消息就无限等待，等待过程中会 把mBlocked = true标记为阻塞，等到有新消息入队，也就是enqueueMessage(msg,when)的时候，会判断此时是不是有消息需要执行: if (p == null || when == 0 || when < p.when)， 如果有消息需要执行，就判断是否阻塞，如果阻塞，就唤醒(needWake = mBlocked)，否则不唤醒。

**问题1:** MessageQueue不会无限休眠吗? 所以要么是限时等待，时间到了就会自动醒来，要么是无限等待，无限等待只发生在MessageQueue为空的情况下，而在此情况下，如果有新消息入队，一定满足p == null，也就是一定会唤醒，所以不存在 无限休眠的状态。

**问题2:** Handler怎么实现延时机制的? 仔细看注释的小伙伴已经知道了，Handler是获取当前时间，然后加上delay的时间，保存到message.when中，然后在looper.loop()中获取当前时间跟这个when对比，如果小于说明需要立刻执行，否则就做个减法: int waitTime = message.when - now;然后调用cpu等待waitTime再执行，从而达到了延时的效果

**问题3:** 里面有一些异步消息是什么情况下使用的? 这个是用于屏幕渲染时候的VSYNC机制的，可以这么说: VSYNC通过handler来实现同步，VSYNC的消息都是异步的，会优先被处理

Handler是通过nativePollOnce()和nativeWake()来等待和唤醒的，并且通过msg.when来实现延时的，同时也留下了几个问题:

**1 什么是异步消息AsyncMessage**

**2 为什么要把Looper放在ThreadLocal里面**

**3 ActivityThread.main()是在哪里被调用的**

**1 什么是异步消息AsyncMessage**

Message next() {

...省略部分代码...

for (; ; ) {

...省略部分代码...

synchronized (this) {

Message prevMsg = null;

Message msg = mMessages;

//这里循环找异步消息 msg.isAsynchronous

if (msg != null && msg.target == null) {

do {

prevMsg = msg;

msg = msg.next;

} while (msg != null && !msg.isAsynchronous());

}

...省略部分代码...

}

}

}

我们来看看msg.isAsynchronous代码:

public boolean isAsynchronous() {

return (flags & FLAG\_ASYNCHRONOUS) != 0;

}

static final int FLAG\_ASYNCHRONOUS = 1 << 1;

那么我们来看看在哪里添加了这个flag:

public void setAsynchronous(boolean async) {

if (async) {

flags |= FLAG\_ASYNCHRONOUS;

} else {

flags &= ~FLAG\_ASYNCHRONOUS;

}

}

现在我们看到Message.setAsynchronous(boolean async)这个方法用来控制是否是异步消息，在哪里调用了呢， 我们先来看View.requestLayout():

public void requestLayout() {

...省略部分代码...

//这里的mParent是自己的父View，如果view本身已经是最顶层view，则mParent就是ViewRootImpl

if (mParent != null && !mParent.isLayoutRequested()) {

mParent.requestLayout();

}

...省略部分代码...

}

这里会层层向上调用mParent.requestLayout()，直到调用到ViewRootImpl.requestLayout()为止

@Override

public void requestLayout() {

if (!mHandlingLayoutInLayoutRequest) {

checkThread(); //检查当前线程

mLayoutRequested = true; //更新标记

scheduleTraversals(); //开始遍历

}

}

void checkThread() {

//这里检查线程是否是mThread，不是的话就抛出异常，mThread一般是UI线程

if (mThread != Thread.currentThread()) {

throw new CalledFromWrongThreadException("Only the original thread that created a view hierarchy can touch its views.");

}

}

public boolean mTraversalScheduled;

int mTraversalBarrier;

Choreographer mChoreographer;

void scheduleTraversals() {

if (!mTraversalScheduled) { //不在执行中才执行

mTraversalScheduled = true; //更新标记为执行中

//向mHandler的MessageQueue中添加一个同步屏障，一个同步屏障也是一个Message，但是这个Message的target是null，并且后面会有一个异步消息AsynchronousMessage

mTraversalBarrier = mHandler.getLooper().getQueue().postSyncBarrier();

//这里是重点！！！，通过Choreographer来post一个消息，//TAG1: CALLBACK\_TRAVERSAL，要记住这个TAG

mChoreographer.postCallback(Choreographer.CALLBACK\_TRAVERSAL, mTraversalRunnable, null);

...省略部分代码...

}

}

我们先来看MessageQueue.postSyncBarrier():

public int postSyncBarrier() {

//这里使用当前时间作为参数传递进去

return postSyncBarrier(SystemClock.uptimeMillis());

}

private int postSyncBarrier(long when) {

synchronized (this) {

final int token = mNextBarrierToken++; //token增加1

final Message msg = Message.obtain();

msg.markInUse();

msg.when = when; //msg.when就是当前时间

msg.arg1 = token; //arg1就是token

//注意！！！这个msg没有target，也就是msg.target=null

//循环遍历，将这个同步屏障msg插入到队列

Message prev = null;

Message p = mMessages;

//遍历查找when之前的消息

if (when != 0) {

while (p != null && p.when <= when) {

prev = p;

p = p.next;

}

}

//插入到队列

if (prev != null) {

msg.next = p;

prev.next = msg;

} else {

msg.next = p;

mMessages = msg;

}

return token;

}

}

再来看看removeSyncBarrier():

//这个token就是我们上面保存的

public void removeSyncBarrier(int token) {

synchronized (this) {

//遍历删除这个token对应的msg

Message prev = null;

Message p = mMessages;

//只要p.target!=null || p.arg1!=token就一直找，因为上面我们分析了同步屏障的target=null，并且arg1=token

while (p != null && (p.target != null || p.arg1 != token)) {

prev = p;

p = p.next;

}

//检测队列

if (p == null) {

throw new IllegalStateException("The specified message queue synchronization "

+ " barrier token has not been posted or has already been removed.");

}

final boolean needWake;

//移除同步屏障消息

if (prev != null) {

prev.next = p.next;

needWake = false;

} else {

mMessages = p.next;

needWake = mMessages == null || mMessages.target != null;

}

//回收

p.recycleUnchecked();

//如果需要唤醒，则唤醒等待

if (needWake && !mQuitting) {

nativeWake(mPtr);

}

}

}

然后看mChoreographer.postCallback(Choreographer.CALLBACK\_TRAVERSAL, mTraversalRunnable, null);第一个参数是个int，第二个参数mTraversalRunnable是个Runnable，代码如下:

final TraversalRunnable mTraversalRunnable = new TraversalRunnable();

final class TraversalRunnable implements Runnable {

@Override

public void run() {

doTraversal();

}

}

void doTraversal() {

if (mTraversalScheduled) { //不在执行中才执行

mTraversalScheduled = false; //更新标记为执行中

//从mHandler的MesssageQueue中移除同步屏障，还记得刚刚添加同步屏障的代码吗

mHandler.getLooper().getQueue().removeSyncBarrier(mTraversalBarrier);

//开始遍历布局

performTraversals();

}

}

//这个方法对view进行了测量，布局和绘制，我们后面会细讲

private void performTraversals() {

......

performMeasure();

......

performLayout();

......

performDraw();

......

}

我们来看下Choreographer类的部分关键代码:

class Choreographer {

public static final int CALLBACK\_INPUT = 0;

public static final int CALLBACK\_ANIMATION = 1;

public static final int CALLBACK\_INSETS\_ANIMATION = 2;

public static final int CALLBACK\_TRAVERSAL = 3; //这是刚刚的第一个参数

public static final int CALLBACK\_COMMIT = 4;

private static final int CALLBACK\_LAST = CALLBACK\_COMMIT;

private final CallbackQueue[] mCallbackQueues;

private final FrameHandler mHandler;

private final FrameDisplayEventReceiver mDisplayEventReceiver;

private Choreographer(Looper looper, int vsyncSource) {

mLooper = looper;

mHandler = new FrameHandler(looper);

//这里初始化mDisplayEventReceiver

mDisplayEventReceiver = USE\_VSYNC

? new FrameDisplayEventReceiver(looper, vsyncSource)

: null;

mLastFrameTimeNanos = Long.MIN\_VALUE;

mFrameIntervalNanos = (long)(1000000000 / getRefreshRate());

//这里对mCallbackQueues进行初始化，直接new出来一个长度为5的数组，并且初始化5个元素，为什么长度是5？因为上面说了CALLBACK\_LAST=4

mCallbackQueues = new CallbackQueue[CALLBACK\_LAST + 1];

for (int i = 0; i <= CALLBACK\_LAST; i++) {

mCallbackQueues[i] = new CallbackQueue();

}

}

//这就是刚刚调用的方法

public void postCallback(int callbackType, Runnable action, Object token) {

postCallbackDelayed(callbackType, action, token, 0); //第四个参数是0

}

public void postCallbackDelayed(int callbackType,Runnable action, Object token, long delayMillis) {

if (action == null) {

throw new IllegalArgumentException("action must not be null");

}

if (callbackType < 0 || callbackType > CALLBACK\_LAST) {

throw new IllegalArgumentException("callbackType is invalid");

}

postCallbackDelayedInternal(callbackType, action, token, delayMillis);

}

//最终调到这里

private void postCallbackDelayedInternal(int callbackType, Object action, Object token, long delayMillis) {

synchronized (mLock) {

final long now = SystemClock.uptimeMillis(); //获取当前时间

final long dueTime = now + delayMillis; //delayMillis = 0，所以dueTime = now

//这是个数组 callbackType是CALLBACK\_TRAVERSAL，dueTime=0，action就是那个mTraversalRunnable

//这里就是将这个mTraversalRunnable保存在CALLBACK\_TRAVERSAL对应的集合里了

mCallbackQueues[callbackType].addCallbackLocked(dueTime, action, token);

if (dueTime <= now) { //true

scheduleFrameLocked(now); //跑这里

} else {

//否则通过Handler发送出去

Message msg = mHandler.obtainMessage(MSG\_DO\_SCHEDULE\_CALLBACK, action);

msg.arg1 = callbackType;

msg.setAsynchronous(true);

mHandler.sendMessageAtTime(msg, dueTime);

}

}

}

//接着看

private void scheduleFrameLocked(long now) {

if (!mFrameScheduled) {

mFrameScheduled = true;

if (USE\_VSYNC) {

//看这里

if (isRunningOnLooperThreadLocked()) {

scheduleVsyncLocked();

} else {

Message msg = mHandler.obtainMessage(MSG\_DO\_SCHEDULE\_VSYNC); //msg.what = MSG\_DO\_SCHEDULE\_VSYNC

msg.setAsynchronous(true); //标记为异步消息!! 前面我们加了个同步屏障了的

mHandler.sendMessageAtFrontOfQueue(msg); //发送，这个mHandler是FrameHandler

}

} else {

final long nextFrameTime = Math.max(mLastFrameTimeNanos / TimeUtils.NANOS\_PER\_MS + sFrameDelay, now);

Message msg = mHandler.obtainMessage(MSG\_DO\_FRAME);

msg.setAsynchronous(true);

mHandler.sendMessageAtTime(msg, nextFrameTime);

}

}

}

}

接着来看FrameHandler的代码:

private final class FrameHandler extends Handler {

public FrameHandler(Looper looper) {

super(looper);

}

@Override

public void handleMessage(Message msg) {

switch (msg.what) {

case MSG\_DO\_FRAME:

doFrame(System.nanoTime(), 0);

break;

case MSG\_DO\_SCHEDULE\_VSYNC: //跑到了这里

doScheduleVsync();

break;

case MSG\_DO\_SCHEDULE\_CALLBACK:

doScheduleCallback(msg.arg1);

break;

}

}

}

void doScheduleVsync() {

synchronized (mLock) {

if (mFrameScheduled) {

scheduleVsyncLocked();

}

}

}

private void scheduleVsyncLocked() {

mDisplayEventReceiver.scheduleVsync();

}

//来看下scheduleVsync方法:

public void scheduleVsync() {

if (mReceiverPtr == 0) {

Log.w(TAG, "Attempted to schedule a vertical sync pulse but the display event "

+ "receiver has already been disposed.");

} else {

//这里直接调用了native的方法，native调用完成会回到

nativeScheduleVsync(mReceiverPtr);

}

}

native完事后会回调到FrameDisplayEventReceiver.onVsync()里面：

private final class FrameDisplayEventReceiver extends DisplayEventReceiverimplements Runnable {

private boolean mHavePendingVsync;

private long mTimestampNanos;

private int mFrame;

public FrameDisplayEventReceiver(Looper looper, int vsyncSource) {

super(looper, vsyncSource, CONFIG\_CHANGED\_EVENT\_SUPPRESS);

}

@Override

public void onVsync(long timestampNanos, long physicalDisplayId, int frame) {

long now = System.nanoTime();

if (timestampNanos > now) {

timestampNanos = now;

}

if (mHavePendingVsync) {

Log.w(TAG, "Already have a pending vsync event. There should only be "

+ "one at a time.");

} else {

mHavePendingVsync = true;

}

mTimestampNanos = timestampNanos;

mFrame = frame;

Message msg = Message.obtain(mHandler, this); //这里的第二个参数是this，意味着msg.callback = this，那么就会跑到自己的run函数

msg.setAsynchronous(true);

mHandler.sendMessageAtTime(msg, timestampNanos / TimeUtils.NANOS\_PER\_MS);

}

@Override

public void run() {

//跑到这里了

mHavePendingVsync = false;

doFrame(mTimestampNanos, mFrame);

}

}

void doFrame(long frameTimeNanos, int frame) {

final long startNanos;

synchronized (mLock) {

if (!mFrameScheduled) {

return; // no work to do

}

...省略计算时间的代码...

//下面开始执行事件顺序为: 1 输入事件 2 动画 3 布局

try {

Trace.traceBegin(Trace.TRACE\_TAG\_VIEW, "Choreographer#doFrame");

AnimationUtils.lockAnimationClock(frameTimeNanos / TimeUtils.NANOS\_PER\_MS);

mFrameInfo.markInputHandlingStart();

doCallbacks(Choreographer.CALLBACK\_INPUT, frameTimeNanos);

mFrameInfo.markAnimationsStart();

doCallbacks(Choreographer.CALLBACK\_ANIMATION, frameTimeNanos);

doCallbacks(Choreographer.CALLBACK\_INSETS\_ANIMATION, frameTimeNanos);

mFrameInfo.markPerformTraversalsStart();

//TAG: CALLBACK\_TRAVERSAL，还记得这个TAG吗，我们看这里即可

doCallbacks(Choreographer.CALLBACK\_TRAVERSAL, frameTimeNanos);

doCallbacks(Choreographer.CALLBACK\_COMMIT, frameTimeNanos);

} finally {

AnimationUtils.unlockAnimationClock();

Trace.traceEnd(Trace.TRACE\_TAG\_VIEW);

}

}

void doCallbacks(int callbackType, long frameTimeNanos) {

CallbackRecord callbacks;

synchronized (mLock) {

//这里根据callbackType取出所有事件，这里的type是:CALLBACK\_TRAVERSAL

callbacks = mCallbackQueues[callbackType].extractDueCallbacksLocked(now / TimeUtils.NANOS\_PER\_MS);

if (callbacks == null) {

return;

}

mCallbacksRunning = true;

...省略部分代码...

}

try {

//遍历取出所有事件，执行，还记得我们之前的那个:mChoreographer.postCallback(Choreographer.CALLBACK\_TRAVERSAL, mTraversalRunnable, null)吗？

//第二个参数就是i一个i饿runnable，就会被执行，也就是执行到我们的traversal()里去了，最终就会去测量、布局、绘制

for (CallbackRecord c = callbacks; c != null; c = c.next) {

c.run(frameTimeNanos);

}

} finally {

...回收callback...

}

}

通过以上分析我们了解到: Android在布局的时候使用了异步消息，为什么要使用异步消息呢，有什么好处，答案就是：异步消息会被优先处理，比如当前MessageQueue中有100个消息，这时候有个布局消息来了，正常的话要等待100个执行完，如果是异步消息，则可以优先执行，这样UI就不会因为消息过多而卡顿，这就是优点，我们再来看看MessageQueue.next的代码:

Message next() {

...省略部分代码...

for (;;) {

...省略部分代码...

synchronized (this) {

final long now = SystemClock.uptimeMillis(); //记录当前时间

Message prevMsg = null;

Message msg = mMessages; //取出第一条消息，也就是队头消息

//msg.target == null! 上面分析过，这是一个同步屏障，而同步屏障后面一般会有一个人异步消息，同步屏障的目的就是为了标记后面有个异步消息的

if (msg != null && msg.target == null) { //如果有同步屏障，那么就找异步消息

do {

prevMsg = msg;

msg = msg.next;

} while (msg != null && !msg.isAsynchronous()); //果然在这里，只要不是异步消息，就一直找，直到找到为止

}

//开始处理消息

if (msg != null) {

...处理msg...

} else {

// 标记为-1无限等待

nextPollTimeoutMillis = -1;

}

...省略部分代码...

}

...省略处理空闲消息的代码...

}

}

**异步消息是被优先执行的，而且异步消息都跟在同步屏障后面的**

**2 为什么要把Looper放在ThreadLocal里面**

**我们先来看怎么把Looper放在ThreadLocal里面的?** **Looper.prepare()**

static final ThreadLocal<Looper> sThreadLocal = new ThreadLocal<Looper>();

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));

}

**ThreadLocal.set()**

public void set(T value) {

Thread t = Thread.currentThread();

ThreadLocalMap map = getMap(t);

if (map != null)

map.set(this, value);

else

createMap(t, value);

}

**ThreadLocalMap.getMap()** 可以看到Thread有一个成变量ThreadLocalMap

ThreadLocalMap getMap(Thread t) {

return t.threadLocals;

}

public class Thread implements Runnable {

ThreadLocal.ThreadLocalMap threadLocals = null;

}

**createMap()** 这里直接创建一个ThreadLocalMap

void createMap(Thread t, T firstValue) {

t.threadLocals = new ThreadLocalMap(this, firstValue);

}

***这里来总结一下存放Looper的逻辑*** 1 每个线程都有个ThreadLocalMap的成员变量threadLocals，这个变量可以理解为一个HashMap，其中key是ThreadLocal，value是任意类型 2 Looper内部有个ThreadLocal成员变量sThreadLocal，在prepare()的时候，会先创建一个Looper，然后获取当前线程的ThreadLocalMap，并且把自自己的成员变量sThreadLocal作为key，把这个looper作为值存放进去，可以简单理解为:

Looper looler = new Looper(); //创建一个looper

ThreadLocalMap map = Thread.currentThread().getMap(); //获取当前线程的ThreadLocalMap

map.set(sThreadLocal, looper); //将looper存入当前线程的ThreadLocalMap中

**我们再来看看是怎么从ThreadLocal里面获取Looper的?**

public static @Nullable Looper myLooper() {

return sThreadLocal.get();

}

很简单，直接使用ThreadLocal的get()函数，***这里可以缕一下：*** 因为ThreadLocal是每个线程都有的成员变量，所以在线程A里面使用Looper.myLooper()得到的就是线程A的looper，线程B得到的就是线程B的looper，UI线程肯定就是UI线程的looper，所以就达到了线程切换的目的。

**3 ActivityThread.main()是在哪里被调用的**

答: 是在app进程被创建的时候调用的，app进程被创建的时候，会通过zygote来fork一个新进程作为app进程，然后通过反射调用ActivityThread的main方法

**总结**

1 异步消息跟在同步屏障后面(不一定是紧临)，异步消息会被优先执行，寻找异步消息的过程是先找同步屏障，如果有，就遍历next来找异步消息，同步屏障的特点是msg.target=null，异步消息的特点是isAsynchronous = true。 eg: 同步屏障是50，异步消息是100，同步屏障前面的1-49会先执行，然后执行到50的时候发现是同步屏障，然后就遍历后续消息，找到异步消息100，取出来执行，然后移除同步屏障50，在继续执行后面的消息51，以此类推

2 使用ThreadLocal是为了将Looper保存在线程内部，从而达到跨线程的目的

# Handler Native层实现机制

我们知道Handler的构建需要先创建Looper，而创建Looper需要先创建MessageQueue，我们直接来看MessageQueue的构造，顺便放上native相关的变量:

package android.os;

public final class MessageQueue {

private long mPtr; //这个mPtr是native层返回的

private native static long nativeInit();

private native static void nativeDestroy(long ptr);

private native void nativePollOnce(long ptr, int timeoutMillis);

private native static void nativeWake(long ptr);

private native static boolean nativeIsPolling(long ptr);

private native static void nativeSetFileDescriptorEvents(long ptr, int fd, int events);

MessageQueue(boolean quitAllowed) {

mQuitAllowed = quitAllowed;

mPtr = nativeInit(); //这里调用了第一个native函数，对应native层的android\_os\_MessageQueue.cpp的android\_os\_MessageQueue\_nativeInit，记住这个返回值

}

}

我们来看下对应的native函数:

static jlong android\_os\_MessageQueue\_nativeInit(JNIEnv\* env, jclass clazz) {

//创建native层的MessageQueue

NativeMessageQueue\* nativeMessageQueue = new NativeMessageQueue();

//...

//计数器加1

nativeMessageQueue->incStrong(env);

//将native层的MessageQueue指针转换成long并返回给java层，也就是java层的mPtr

return reinterpret\_cast<jlong>(nativeMessageQueue);

}

//NativeMessateQueue的构造函数

NativeMessageQueue::NativeMessageQueue():mPollEnv(NULL), mPollObj(NULL), mExceptionObj(NULL) {

mLooper = Looper::getForThread(); //从当前Thread中获取Looper

if (mLooper == NULL) {

mLooper = new Looper(false); //没有就创建

Looper::setForThread(mLooper); //放入当前线程

}

}

接着我们来看native层Looper的构造函数

Looper::Looper(bool allowNonCallbacks): mAllowNonCallbacks(allowNonCallbacks),

//...建立文件描述符

mWakeEventFd.reset(eventfd(0, EFD\_NONBLOCK | EFD\_CLOEXEC));

//...建立epoll机制

rebuildEpollLocked();

}

**我们先来大致了解一下linux的epoll机制:**

#### 1 关于fd

在linux上，一切都是以文件的形式存在的，设备、驱动等都是文件，比如binder驱动，这些文件都是以文件描述符fd来描述的，fd是一个非负整数，代表一个文件的编号，linux上每一个进程控制块PCB都会有一个文件描述符映射表，fd就是这个表的索引，这个表的每个项有一个指向已打开的文件的指针。所以我们可以将fd理解为文件、驱动、设备等。

#### 2 epoll的相关方法

* epoll\_create: 创建一个epoll对象
* epoll\_ctl(epollfd, option, fd, event): 这个方法根据参数可以分为很多作用

***option可以是下面类型:***  
EPOLL\_CTL\_ADD 添加  
EPOLL\_CTL\_DEL 删除  
EPOLL\_CTL\_MOD 修改

***event可以是下面类型:***  
EPOLLIN 文件描述符可读  
EPOLLOUT 文件描述符可写  
EPOLLERR 文件描述符错误  
EPOLLHUP 文件描述符挂起

* epoll\_wait(epollfd,...): 等待直到epollfd对应的事情发生

#### 3 关于epoll机制

可以简单的理解为EPOLLIN和EPOLLOUT，EPOLLIN表示可读，EPOLLOUT表示可写，写入端写入数据后，读出端会立刻感应到，从而去读数据，当读出端读出数据后，写入端会立刻感应到，从而去写数据

接着我们来看一下epoll代码:

void Looper::rebuildEpollLocked() {

//如果文件描述符已经存在就关闭

if (mEpollFd >= 0) {

mEpollFd.reset();

}

//创建一个新的epoll文件描述符，并添加到wake管道

mEpollFd.reset(epoll\_create1(EPOLL\_CLOEXEC));

struct epoll\_event eventItem; //创建一个epoll事件eventItem

memset(& eventItem, 0, sizeof(epoll\_event)); //将eventItem清空

eventItem.events = EPOLLIN; //设置这个事件类型为EPOLLIN，也就是可读事件

eventItem.data.fd = mWakeEventFd.get(); 设置文件描述符为唤醒事件

//注册唤醒事件到epoll

int result = epoll\_ctl(mEpollFd.get(), EPOLL\_CTL\_ADD, mWakeEventFd.get(), &eventItem);

//注册其他事件

for (size\_t i = 0; i < mRequests.size(); i++) {

const Request& request = mRequests.valueAt(i);

struct epoll\_event eventItem;

request.initEventItem(&eventItem);

int epollResult = epoll\_ctl(mEpollFd.get(), EPOLL\_CTL\_ADD, request.fd, &eventItem);

}

}

到这里我们总结一下:

* 1 我们要创建Handler;
* 2 因为创建Handler需要Looper，所以我们去创建Looper;
* 3 因为创建Looper，需要创建MessageQueue，所以我们去创建MessageQueue;
* 4 在创建MessageQueue的时候，我们发现他调用了nativeInit();
* 5 于是我们进入native层的nativeInit();***这里开始进入native了***
* 6 我们在nativeInit中发现创建了native的MessageQueue
* 7 然后我们跟着进入MessageQueue的构造发现创建了native的Looper
* 8 然后我们进入Looper的构造发现 ***1 建立文件描述符 2 建立epoll***
* 9 然后我们进入建立epoll的代码发现注册了一个唤醒事件
* 10 逻辑结束

我们通过第一章直到发送消息最后是要进入MessageQueue的，直接看enqueue即可

boolean enqueueMessage(Message msg, long when) {

synchronized (this) {

//...

boolean needWake;

if (p == null || when == 0 || when < p.when) {

//...

needWake = mBlocked;

} else {

//是否需要唤醒

needWake = mBlocked && p.target == null && msg.isAsynchronous();

//...

}

//本次逻辑在这里，如果需要唤醒，就调用nativeWake(mPtr)，这个mPtr就是构造函数nativeInit()的那个

if (needWake) {

nativeWake(mPtr);

}

}

return true;

}

我们看下native层的nativeWake():

static void android\_os\_MessageQueue\_nativeWake(JNIEnv\* env, jclass clazz, jlong ptr) {

//根据ptr取出对应的MessageQueue

NativeMessageQueue\* nativeMessageQueue = reinterpret\_cast<NativeMessageQueue\*>(ptr);

//调用wake()函数

nativeMessageQueue->wake();

}

void NativeMessageQueue::wake() {

//直接调用了Looper的wake()

mLooper->wake();

}

void Looper::wake() {

uint64\_t inc = 1;

//将唤醒事件的文件描述符mWakeEventFd写入inc，我们通过epoll机制知道，当写入数据的时候，读出端会立刻感知到，

//也就是说，此时读出端已经知道有数据写入了，也就是有message消息需要处理了，那么在哪处理呢，对了，就在MessageQueue的next()里面，我们去看

ssize\_t nWrite = TEMP\_FAILURE\_RETRY(write(mWakeEventFd, &inc, sizeof(uint64\_t)));

//...

}

总结一下:

* 1 我们在Java层send一个message
* 2 最终会调用到java层的MessageQueue的enqueue中
* 3 如果需要唤醒就会去调用nativeWake() ***这里开始进入native***
* 4 nativeWake最终会调用到native Looper的wake()
* 5 这个wake()最终写入了一个唤醒事件，用于通知有消息写入
* 6 那么我们来看java层怎么收到这个消息

经过第一章我们知道消息是在Looper.loop()中处理的，又是从MessageQueue.next()中取出的，我们来看MessageQueue.next():

Message next() {

final long ptr = mPtr;

if (ptr == 0) {

return null;

}

int nextPollTimeoutMillis = 0;

for (; ; ) {

//...

//ptr就是上面的mPtr，nextPollTimeoutMillis就是休眠时间

nativePollOnce(ptr, nextPollTimeoutMillis);

//...

}

}

可以看到在开始就调用了nativePollOnce(ptr,time)这个函数，我们看下这个函数:

static void android\_os\_MessageQueue\_nativePollOnce(JNIEnv\* env, jobject obj, jlong ptr, jint timeoutMillis) {

//又是这一行，根据ptr取MessageQueue的

NativeMessageQueue\* nativeMessageQueue = reinterpret\_cast<NativeMessageQueue\*>(ptr);

//调用MessageQueue的pollOnce()

nativeMessageQueue->pollOnce(env, obj, timeoutMillis);

}

void NativeMessageQueue::pollOnce(JNIEnv\* env, jobject pollObj, int timeoutMillis) {

//...

//看这句就行，又是调用了Looper的pollOnce，看来native的MessageQueue就是个传话的，真正干事的是Looper，这难道就是传说中的代理模式?

mLooper->pollOnce(timeoutMillis);

//...

}

int Looper::pollOnce(int timeoutMillis, int\* outFd, int\* outEvents, void\*\* outData) {

int result = 0;

//死循环...

for (;;) {

//死循环结束的条件是result != 0，那就是说如果等于0就一直跑

if (result != 0) {

//...

return result;

}

//等于0就一直跑这个，那就来看这个，pollInner其实是不会返回0的，那也就是说，只要pollInner有返回，就能结束循环

result = pollInner(timeoutMillis);

}

}

来看下最核心的pollInner方法，参数就是java层传递的那个nextPollTimeoutMillis参数:

int Looper::pollInner(int timeoutMillis) {

//...

//result的取值有四种: POLL\_WAKE = -1 POLL\_CALLBACK = -2 POLL\_TIMEOUT = -3 POLL\_ERROR = -4

int result = POLL\_WAKE;

//创建事件集合eventItems，EPOLL\_MAX\_EVENTS=16

struct epoll\_event eventItems[EPOLL\_MAX\_EVENTS];

//调用epoll\_wait()来等待事件，如果有事件，就放入事件集合eventItems中，并返回事件数量，如果没有，就一直等，超时时间为我们传入的timeoutMillis

int eventCount = epoll\_wait(mEpollFd, eventItems, EPOLL\_MAX\_EVENTS, timeoutMillis);

//加锁

mLock.lock();

//如果发生的事件小于0，也就是说没有事件处理，就跳转到Done

if (eventCount <= 0) {

if (errno == EINTR) {

goto Done;

}

result = POLL\_ERROR;

goto Done;

}

//没有goto到Done，也就是有事件发生，就跑到这里

//遍历事件集合eventItems

for (int i = 0; i < eventCount; i++) {

//取出文件描述符

int fd = eventItems[i].data.fd;

//取出事件类型

uint32\_t epollEvents = eventItems[i].events;

if (fd == mWakeEventFd) {//如果文件描述符为mWakeEventFd

if (epollEvents & EPOLLIN) {//并且事件类型为EPOLLIN（可读事件）

//说明有数据可读，就调用awoken()读数据，直到读完为止，

awoken();

} else {

ALOGW("Ignoring unexpected epoll events 0x%x on wake event fd.", epollEvents);

}

} else {

//...

}

}

//这里就是Done，是从if(eventCount<=0)跳转过来的

Done:;

mNextMessageUptime = LLONG\_MAX;

//mMessageEnvelopes是一个Vector，存放native层的消息

while (mMessageEnvelopes.size() != 0) {

nsecs\_t now = systemTime(SYSTEM\_TIME\_MONOTONIC);

//取出第一个MessageEnvelope，MessageEnvelop有收件人Hanlder和消息内容Message，可以理解为jave层的Message

const MessageEnvelope& messageEnvelope = mMessageEnvelopes.itemAt(0);

//判断消息的执行时间，跟java层的那个 if(message.when <= now)类似

if (messageEnvelope.uptime <= now) { //表示消息已经到了执行时间

{

//取出Handler

sp<MessageHandler> handler = messageEnvelope.handler;

//取出Message

Message message = messageEnvelope.message;

mMessageEnvelopes.removeAt(0); //删除，因为接下来就处理了

mSendingMessage = true;

//释放锁

mLock.unlock();

//处理消息

handler->handleMessage(message);

}

mLock.lock();

mSendingMessage = false;

result = POLL\_CALLBACK;

} else {//消息还没到执行时间

mNextMessageUptime = messageEnvelope.uptime;

//跳出循环，进入下一次轮询，java层是计算等待时间nextPollTimeoutMillis

break;

}

}

//释放锁

mLock.unlock();

//...

return result;

}

 1 我们要处理消息，所以先取消息，所以进入了java层的MessageQueue

 2 如果当前消息还不到执行，怎么办，就计算还有多久执行，得出timeout，然后调用nativePollOnce(timeout);

 3 然后我们进入native的MessageQueue ***此处进入native***

 4 我们一路追踪，发现native的MessageQueue就是个靠嘴吃饭的，毛都没干，最终进入了native的Looper里

 5 我们发现Looper的pollOnce()是个死循环，内部调用了pollInner()，只要pollInner()返回，就跳出循环从而回到java层继续向下执行

 6 进入pollInner()，我们发现他有三个逻辑  
等待事件，超时时间就是java层传递过来的timeout，如果有事件发生，则会返回发生的事件，并去处理  
有事件发生，就取出事件并处理  
没事件发生，就处理native的Message  
返回result，且result!=0

 7 返回到jave层，继续向下处理

* 1 java层的Message如果不该处理，就会调用nativePollOnce()等待，在代码中哪里等待呢?

答: 就在native Looper的pollOnce里面的epoll\_wait(timeout)，只要时间不到timeout，就会一直等待，等价于卡在这里，不向下执行

* 2 如果我的timeout传递-1，就是一直等待，那不就卡死在这里了吗

答: 如果在等待期间，有事件发生，那么会立刻返回事件数目，此时向下执行，发现有事件，就会去读取事件并处理，比如java层enqueue事件的时候，就会跑到nativeWake()里面，此时写入一个唤醒事件，此时 epoll\_wait()就会监听到唤醒事件，就会立刻返回，向下执行。

* 3 如果一直没有事件处理呢？

答: 那么就在等待timeout后，直接返回0，此时就会去执行native的message，代码中的goto Done就是这个逻辑

**总结**: java层通过nativePollOnce(timeout)来实现等待，唤醒的方法要么是过了timeout时间后，超时了，要么是通过nativeWake()来写入一个事件，让native监听到这个事件从而返回。

1 现在我们调用handler.sendMessageDelay(msg,3000);让一个message在3秒后执行，

2 MessageQueue.next()查看此message，发现msg.when > now，还不该执行，于是就将nextPollTimeoutMillis = 3000,

并且将mBlockd = true,然后调用nativePollOnce(nextPollTimeoutMillis)去等待，此时native先跑到pollOnce()，

result被初始化为0，所以一直在for里面，然后进入pollOnce()，在epoll\_wait()死等着

3 此时我们又调用handler.sendMessage(msg)，让一个message立即执行

4 此时在MessageQueue.enqueueMessage()中，先将这个消息插入到延时3000ms的那个message前面，然后发现when==0，

于是就将needwake = mBlocked，也就是true，于是接着就调用了nativeWake();于是接着进入native层的Looper的wake()里面写入了一个唤醒事件，

此时Looper的epoll\_wait()就监听到了，于是立刻返回，处理监听事件后就return result，此时pollOnce()里面因为result!=0，就返回到了java层，

于是MessageQueue的next就立刻活跃起来，马上拿出刚刚sendMessage(msg)的这个msg开始处理，处理完了到下一个(也就是延时3000ms那个)，发现msg.when > now，

就继续第2步的等待，后面如果还有新消息，就重复前面的过程，没有新消息就等待epoll\_wait()超时返回，超时时间是3000ms，返回后跟上述步骤基本一样。