窗体底端

Android 子线程中更新UI的几种方法 及原理

在开发中我们经常会遇到在子线程中更新ui的操作，那我们今天就介绍下几种常用的方式

 1.使用 runOnUiThread方法更新ui

  2.view.post 方法

 3.view.postDelaged 方法

 4.handler.send...方法更新ui

1.使用 runOnUiThread方法更新ui

     这个方法在Activity 中 ，所以只有在他的子类或者拿到他实例时才能使用

runOnUiThread(new Runnable() {

@Override

public void run() {

}

});

我们看下runOnUiThread方法的源码

public final void runOnUiThread(Runnable action) {

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

mHandler.post(action);

} else {

action.run();

}

}

这里有个判断，如果是当前的ui线程那么就直接执行runnable 的run方法开始更新ui,如果不是就用handler发送posp 消息，我们先看下mHandler是咋回事，通过activity源码发现它本身new了一个mHandler 就是我们刚才使用的handrler，如果你了解过handrler 源码你就知道，handrler是会去绑定一个Looper,android应用在启动的时候就创建了looper 并且开始轮询，具体看ActiityThead main方法

public static void main(String[] args) {

Trace.traceBegin(Trace.TRACE\_TAG\_ACTIVITY\_MANAGER, "ActivityThreadMain");

SamplingProfilerIntegration.start();

// CloseGuard defaults to true and can be quite spammy. We

// disable it here, but selectively enable it later (via

// StrictMode) on debug builds, but using DropBox, not logs.

CloseGuard.setEnabled(false);

Environment.initForCurrentUser();

// Set the reporter for event logging in libcore

EventLogger.setReporter(new EventLoggingReporter());

// Make sure TrustedCertificateStore looks in the right place for CA certificates

final File configDir = Environment.getUserConfigDirectory(UserHandle.myUserId());

TrustedCertificateStore.setDefaultUserDirectory(configDir);

Process.setArgV0("<pre-initialized>");

//获取主线程的looper对象

Looper.prepareMainLooper();

//开启主线程

ActivityThread thread = new ActivityThread();

thread.attach(false);

if (sMainThreadHandler == null) {

sMainThreadHandler = thread.getHandler();

}

if (false) {

Looper.myLooper().setMessageLogging(new

LogPrinter(Log.DEBUG, "ActivityThread"));

}

// End of event ActivityThreadMain.

Trace.traceEnd(Trace.TRACE\_TAG\_ACTIVITY\_MANAGER);

//开始消息轮训

Looper.loop();

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

}

 到这里我们知道主线程已将存在， looper已经创建完毕，并开始轮询消息。现在处于主线程中handler绑定主线程中的looper。了解完这些，我们心中大概有个概念了。那继续看下post方法

public final boolean post(Runnable r)

{

return sendMessageDelayed(getPostMessage(r), 0);

}

private static Message getPostMessage(Runnable r) {

Message m = Message.obtain();

m.callback = r;

return m;

}

看到传进来的Runnable 方法赋值给了 一条消息的callback 接口，然后调用sendMessagDelayed方法，这个方法我们应该比较熟悉了，不就是我们平时使用handler发送消息时调用的方法么。这个方法最会调用到了sendMessageAtFrontOfQueue(Message msg) 方法，这个比较重要了。

public final boolean sendMessageAtFrontOfQueue(Message msg) {

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, 0);

}

首先拿到了当前线程也就是主线程的消息队列mQueue，如果不知咋回事建议看看下handler原理，这里就再唠叨几句，一个线程中会有一个looper 对象和一个MessageQueue对象 可以有多个handler 。然后调用了enqueueMessage(MessageQueue queue, Message msg, long uptimeMillis) 方法 。

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

msg.target = this;

if (mAsynchronous) {

msg.setAsynchronous(true);

}

return queue.enqueueMessage(msg, uptimeMillis);

}

然后调用消息队列的enqueueMessage 将消息放入消息对列中

boolean enqueueMessage(Message msg, long when) {

if (msg.target == null) {

throw new IllegalArgumentException("Message must have a target.");

}

if (msg.isInUse()) {

throw new IllegalStateException(msg + " This message is already in use.");

}

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;

//把刚进入消息队列的消息置位消息队列的第一条消息

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 {

//根据时间排序,为刚进入队列的消息寻找合适的位置

// 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) {

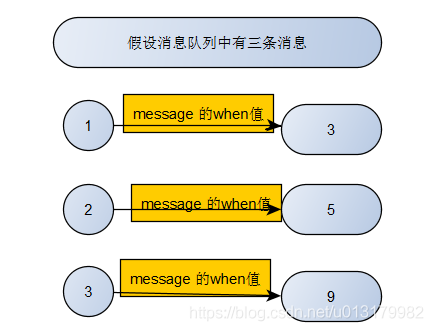
nativeWake(mPtr);

}

}

return true;

}

* 第一种情况比较好理解，就是把当前消息的next 指向一个空消息 将自身赋值为mMessages的值
* 重点理解下第二种情况，假设有三条消息，三条消息对应的when值分别为3,5,9
* 

1. 假设我们传进来的msg的when值为 4，那初次进入参数值为

1.  prev.when=p.when=3

2.   p.when=p.next.when= 5

3.   xxMessage.when= 9

现在msg.when的值小于p.when=5 ,when<p.when 所以执行了 break方法，那么msg.next=p   既msg的下一条指向了p,

prev.next=msg 既  prev --> msg--->p---->xxMessage   顺利将msg插入到队列中。

接着我们看下消息的出列操作，前边我们讲过Looper对象， ui线已经创建了looper对象 和队列 并且调用了Looper.loop()方法开始轮询消息出站操作。看下loop()方法。

public static void loop() {

//获取looper对象

final Looper me = myLooper();

if (me == null) {

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

}

//拿到了队列

final MessageQueue queue = me.mQueue;

// Make sure the identity of this thread is that of the local process,

// and keep track of what that identity token actually is.

Binder.clearCallingIdentity();

final long ident = Binder.clearCallingIdentity();

//无限循环取消息

for (;;) {

//获取when值最小的一条消息

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

if (msg == null) {

// No message indicates that the message queue is quitting.

return;

}

// This must be in a local variable, in case a UI event sets the logger

final Printer logging = me.mLogging;

if (logging != null) {

logging.println(">>>>> Dispatching to " + msg.target + " " +

msg.callback + ": " + msg.what);

}

final long slowDispatchThresholdMs = me.mSlowDispatchThresholdMs;

final long traceTag = me.mTraceTag;

if (traceTag != 0 && Trace.isTagEnabled(traceTag)) {

Trace.traceBegin(traceTag, msg.target.getTraceName(msg));

}

final long start = (slowDispatchThresholdMs == 0) ? 0 : SystemClock.uptimeMillis();

final long end;

try {

//获取当前绑定的handler 并且调用其中的dispatchMessage方法，执行handler handlerMessage回调

msg.target.dispatchMessage(msg);

end = (slowDispatchThresholdMs == 0) ? 0 : SystemClock.uptimeMillis();

} finally {

if (traceTag != 0) {

Trace.traceEnd(traceTag);

}

}

if (slowDispatchThresholdMs > 0) {

final long time = end - start;

if (time > slowDispatchThresholdMs) {

Slog.w(TAG, "Dispatch took " + time + "ms on "

+ Thread.currentThread().getName() + ", h=" +

msg.target + " cb=" + msg.callback + " msg=" + msg.what);

}

}

if (logging != null) {

logging.println("<<<<< Finished to " + msg.target + " " + msg.callback);

}

// Make sure that during the course of dispatching the

// identity of the thread wasn't corrupted.

final long newIdent = Binder.clearCallingIdentity();

if (ident != newIdent) {

Log.wtf(TAG, "Thread identity changed from 0x"

+ Long.toHexString(ident) + " to 0x"

+ Long.toHexString(newIdent) + " while dispatching to "

+ msg.target.getClass().getName() + " "

+ msg.callback + " what=" + msg.what);

}

//将当前message 初始化以便重复利用

msg.recycleUnchecked();

}

}

       这里主要有三个点注意，1.队列是如何获取when最小的一条消息，2.message是如何重复利用的 这两个我们今天不做探讨，我们主要看下 3.run 方法是如何执行或者handlerMessage如何执行的。进入handler 的dispatchMessage方法看下。

public void dispatchMessage(Message msg) {

if (msg.callback != null) {

handleCallback(msg);

} else {

if (mCallback != null) {

if (mCallback.handleMessage(msg)) {

return;

}

}

handleMessage(msg);

}

}

代码一目了然，如果message设置了callback 那么执行handlerCallback 方法，否则执行我们熟悉的handlerMesage方法。

这段我们讲的是runOnUiThread  传进来一个接口并且获取到了Message对象 when赋值为0 ，msg.callback =Runnable(传入的接口)

看下handleCallback(Message message) 方法。

private static void handleCallback(Message message) {

message.callback.run();

}

只有一行代码，那就是执行run方法。走了这么大一圈，终于完了。

总结下从runOnUiThread  到run方法执行过程：首先runOnUiThread  方法传入了一个接口，然后获取一个message对象，获取主线程的looper,对列，和handler,然后是消息的入栈操作，接着是消息出栈，获取当前消息的handler调用消息分发事件dispatchMessage，然后判断是接口为空，不为空的话执行run方法。

  2.view.post 方法

 接着看下post方法，上面讲了那么多，如果都理解了，下面的相对就非常简单了。

mPhotoView.post(new Runnable() {

@Override

public void run() {

}

});

这个和 runOnUiThread 基本类似，都是传入一个Runnable接口，看下post（Runnable action）方法。

public boolean post(Runnable action) {

final AttachInfo attachInfo = mAttachInfo;

if (attachInfo != null) {

return attachInfo.mHandler.post(action);

}

// Postpone the runnable until we know on which thread it needs to run.

// Assume that the runnable will be successfully placed after attach.

getRunQueue().post(action);

return true;

}

attachinfo 是当前view 的属性信息，如果为空调用队列的post方法。

attachInfo.mHandler.post(action) 这个是view 本身都会有一个handler 绑定的是主线程的looper ，再看post

public final boolean post(Runnable r)

{

return sendMessageDelayed(getPostMessage(r), 0);

}

调用到了handler 的sendMessageDelayed（Message msg,long delayMillis）方法我们上边讲过了，接下来他的调用流程和runOnUiThread   一样了。最后都会执行run方法。

现在看下attactInfo 为空的情况。

getRunQueue().post(action);

看下RunQueue是啥

static RunQueue getRunQueue() {

RunQueue rq = sRunQueues.get();

if (rq != null) {

return rq;

}

rq = new RunQueue();

sRunQueues.set(rq);

return rq;

}

先通过aRunQueues.get获取一个Runqueue对象,如果有就返回，没有就新建一个并调用sRunQueues的set方法,这个不细说，重点看下RunQueue 的post

void post(Runnable action) {

postDelayed(action, 0);

}

void postDelayed(Runnable action, long delayMillis) {

HandlerAction handlerAction = new HandlerAction();

handlerAction.action = action;

handlerAction.delay = delayMillis;

synchronized (mActions) {

mActions.add(handlerAction);

}

}

其实 是调用了postDelayed方法，我们看到新建了一个HandlerActivity 并且初始化参数， 并加入到一个 ArrayList 集合中。那他在哪里取出hadlerAction 呢。

void executeActions(Handler handler) {

synchronized (mActions) {

final ArrayList<HandlerAction> actions = mActions;

final int count = actions.size();

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

final HandlerAction handlerAction = actions.get(i);

handler.postDelayed(handlerAction.action, handlerAction.delay);

}

actions.clear();

}

}

在当前类中有这么一个方法，取出了handlerAction  并且调用了handler的postDelayed方法，这个我们比较熟悉，就是最后要执行run方法了。我们重点看下executeAction 何时调用的。在ViewRootlmp 中找到了。performTraversals方法，那么它又是哪里调用的呢。

向上追溯， doTraversal（） --->  TraversalRunnable  ---> scheduleTraversals 最后到了这里，这个方法是绘制view了

boolean cancelDraw = mAttachInfo.mTreeObserver.dispatchOnPreDraw() ||

viewVisibility != View.VISIBLE;

if (!cancelDraw && !newSurface) {

if (!skipDraw || mReportNextDraw) {

if (mPendingTransitions != null && mPendingTransitions.size() > 0) {

for (int i = 0; i < mPendingTransitions.size(); ++i) {

mPendingTransitions.get(i).startChangingAnimations();

}

mPendingTransitions.clear();

}

performDraw();

}

} else {

if (viewVisibility == View.VISIBLE) {

// Try again

scheduleTraversals();

} else if (mPendingTransitions != null && mPendingTransitions.size() > 0) {

for (int i = 0; i < mPendingTransitions.size(); ++i) {

mPendingTransitions.get(i).endChangingAnimations();

}

mPendingTransitions.clear();

}

}

说实话，这个方法调用的地方太多了，一时间还真的难以确认是哪里调用的，只能大胆猜测下，在view 属性获取成功，并且状态可见时调用。现在算是流程整个通了，梳理下：当没有获取到view相关属性时，就将action存起来（mActions.add(handlerAction)），等到获取到view相关属性并且状态可见时再取出（scheduleTraversals）。

现在看executeActions（方法）调用了handler的postDelayed方法 ，当然，这个handler也是主线程的，下面流程就是handler那套了，通过判断是否有接口，执行run方法。

3.view.postDelaged 方法

前面我们讲了view.post 方法，如果你仔细观察，这个方法其实都不用讲了。

public boolean postDelayed(Runnable action, long delayMillis) {

final AttachInfo attachInfo = mAttachInfo;

if (attachInfo != null) {

return attachInfo.mHandler.postDelayed(action, delayMillis);

}

// Assume that post will succeed later

ViewRootImpl.getRunQueue().postDelayed(action, delayMillis);

return true;

}

其实分别调用了handrle的postDelayed 和RunQueue 的postDelayed 方法。而view.post只是对postDelayed的简单封装。

attachInfo.mHandler.post(action);

public final boolean post(Runnable r)

{

return sendMessageDelayed(getPostMessage(r), 0);

}

attachInfo.mHandler.post(action);

void post(Runnable action) {

postDelayed(action, 0);

}

其他的流程都一样。

 4.handler.send...方法更新ui

上面我们讲了几种更新更新ui的方式，其实归根到底好是调用了handler的send...方法来更新数据。

但是归根到底还是调用了  sendMessageAtTime(Message msg, long uptimeMillis) 方法。

public boolean sendMessageAtTime(Message msg, long uptimeMillis) {

MessageQueue queue = mQueue;

if (queue == null) {

RuntimeException e = new RuntimeException(

this + " sendMessageAtTime() called with no mQueue");

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

return false;

}

return enqueueMessage(queue, msg, uptimeMillis);

}

最后通过enqueueMessage 将消息送入到消息队列中，然后消息队列轮行取出消息执行handler中的

public void dispatchMessage(Message msg) 方法，然后执行

private static void handleCallback(Message message) {

message.callback.run();

}

或者

handleMessage(msg)

方法，在run 中或在  handleMessage 执行我们的更新ui 的操做。

到这里更新ui 的几种方法就介绍完了，本来觉得一天就能搞定的结果搞了四天才写完，其中又不断的学习了新的知识，希望大家在看的时候尽量的结合源码去看，这样能够更好的理解。

 补充一点，在看资料的时候看到有人说IntentService 和AsyncTask  也可以更新ui 其实他们都是内置了一个子线程然后绑定了一个主线程的Handler实现的。

参考：

<https://blog.csdn.net/suma_sun/article/details/51584026>

<https://blog.csdn.net/lufeng20/article/details/24314381>

<https://blog.csdn.net/qq_23547831/article/details/50751687>

版权声明：本文为博主原创文章，遵循[CC 4.0 BY-SA](https://creativecommons.org/licenses/by-sa/4.0/)版权协议，转载请附上原文出处链接和本声明。

本文链接：<https://blog.csdn.net/u013179982/article/details/86215938>