# **Android 消息处理机制原理**

## Android 消息机制本质

使用Handler将子线程的Message放入主线的MessageQueue中，在主线程中使用。

## Android 消息机制

我们都知道Android UI线程是不安全的(因invalidate  
方法不安全)，如果在子线程中尝试进行UI更新可能会导致程序崩溃。

解决的办法也很简单：在UI线程中创建一个handler实例，在子线程中创建Message，使用handler将Message发送出去，之在handler的handleMessge()方法中捕获Handler发送的Message对象，然后再进行UI操作。

Android这种方法借鉴了Windows的消息处理机制。

## Android 消息机制原理之Handler的创建

探究如何创建一个Handler对象,以及创建Handler对象的注意事项。  
创建一个Handler对象非常简单,但是也有需要注意的地方。

分别在主线和子线程中创建Handler对象

public class HandlerActivity extends Activity {

Handler mHandler;

Handler sHandler;

@Override

public void onCreate(Bundle savedInstanceState) {

super.onCreate(savedInstanceState);

mHandler = new Handler();

new Thread(new Runnable() {

@Override

public void run() {

sHandler = new Handler();

}

}).start();

}

}

运行以上代码程序崩溃并报错：

Can't create handler inside thread that has not called Looper.prepare()  
**提示**：子线程没有调用Loopr.prepare()，不能创建Handler对象

在sHandler = new Handler();之前调用Loopr.prepare()方法试试。  
程序没有报错。  
究其原因，看起源码，搞清楚为什么不调用Looper.prepare()方法就会报错。Handler的无参数构造函数源码如下：

public Handler() {

if (FIND\_POTENTIAL\_LEAKS) {

final Class<? extends Handler> klass = getClass();

if ((klass.isAnonymousClass() || klass.isMemberClass() || klass.isLocalClass()) &&

(klass.getModifiers() & Modifier.STATIC) == 0) {

Log.w(TAG, "The following Handler class should be static or leaks might occur: " +

klass.getCanonicalName());

}

}

mLooper = Looper.myLooper();

if (mLooper == null) {

throw new RuntimeException(

"Can't create handler inside thread that has not called Looper.prepare()");

}

mQueue = mLooper.mQueue;

mCallback = null;

}

调用Looper.myLooper()方法获取Looper对象，如果Looper对象为null则抛出"Can't create handler inside thread that has not called Looper.prepare()"异常。那么什么时候Looper对象为Null呢？查看Looper.myLooper()源码：

public static Looper myLooper() {

return sThreadLocal.get();

}

源码非常简单，只是从sThreadLocal.get()中获取了Looper对象，结合抛出的异常，不难猜出sThreadLocal中的Looper对象是在Looper.prepare()方法中设置进去的，继续查看Looper.prepare()源码

public static void prepare() {

if (sThreadLocal.get() != null) {

throw new RuntimeException("Only one Looper may be created per thread");

}

sThreadLocal.set(new Looper());

}

通过Looper.prepare()源码可知：

1. 每个线程中只能有一个Looper对象。
2. 在Looper.prepare()中为这个线程创建并设置Loopr对象

那么问题来了，主线程没有调用Looper.prepare()方法，为什么就没有崩溃呢？细心的朋友我相信已经注意到这点，实际上在程序启动时，系统已经为我们调用了Looper.prepare()方法。  
查看ActivityThread的main()方法源码：

public static void main(String[] args) {

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);

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

Looper.prepareMainLooper();

if (sMainThreadHandler == null) {

sMainThreadHandler = new Handler();

}

ActivityThread thread = new ActivityThread();

thread.attach(false);

if (false) {

Looper.myLooper().setMessageLogging(new

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

}

Looper.loop();

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

}

查看Looper.prepareMainLooper()源码，发现会调用Looper.prepare()

public static void prepareMainLooper() {

prepare();

setMainLooper(myLooper());

myLooper().mQueue.mQuitAllowed = false;

}

所以在主线程中就不用再去手动调用Looper.prepare()方法。这样基本上把Handler在线程中的创建搞明白了。

**总结**：

* 在主线程中可以直接创建Handler对象。
* 在子线程中需要先调用Looper.prepare()再创建Handler对象。

## Android 消息机制原理之Message的发送与接收

发送消息这个流程相信大家已经非常熟悉，使用new创建新的Message对象或者通过handler的obtainMessage()方法获取(比较喜欢这种方式)，使用setDate()或者arg参数为Message携带一些数据，并通过Handler对象发送出去。以下示例代码用于发送与接收的讲解：

public class HandlerActivity extends Activity {

Handler mHandler = new Handler(){

@Override

public void handleMessage(Message msg) {

super.handleMessage(msg);

}

};

@Override

public void onCreate(Bundle savedInstanceState) {

super.onCreate(savedInstanceState);

new Thread(new Runnable() {

@Override

public void run() {

Message msg = mHandler.obtainMessage();

msg.arg1 = 14;

Bundle bundle = new Bundle();

bundle.putString("handler","handler");

mHandler.sendMessage(msg);

}

}).start();

}

}

可是这里mHandler到底是把Message发送到哪里去了呢？为什么之后又可以在Handler的handleMessage()方法中重新得到这条Message呢？看来又需要通过阅读源码才能解除我们心中的疑惑了，Handler中提供了很多个发送消息的方法，其中除sendMessageAtFrontOfQueue()方法之外，其它的发送消息方法最终都会辗转调sendMessageAtTime()方法，这个方法的源码如下所示：

public boolean sendMessageAtTime(Message msg, long uptimeMillis){

boolean sent = false;

MessageQueue queue = mQueue;

if (queue != null) {

msg.target = this;

sent = queue.enqueueMessage(msg, uptimeMillis);

}

else {

RuntimeException e = new RuntimeException(

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

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

}

return sent;

}

sendMessageAtTime()方法接收两个参数，其中msg参数就是我们发送的Message对象，而uptimeMillis参数则表示发送消息的时间，它的值等于自系统开机到当前时间的毫秒数再加上延迟时间，如果你调用的不是sendMessageDelayed()方法，延迟时间就为0，然后将这两个参数都传递到MessageQueue的enqueueMessage()方法中。这个MessageQueue又是什么东西呢？其实从名字上就可以看出了，它是一个消息队列(世纪并不是队列结构)，用于将所有收到的消息以队列的形式进行排列，并提供入队和出队的方法。这个类是在Looper的构造函数中创建的，因此一个Looper也就对应了一个MessageQueue。

那么enqueueMessage()方法毫无疑问就是入队的方法了，我们来看下这个方法的源码：

final boolean enqueueMessage(Message msg, long when) {

if (msg.isInUse()) {

throw new AndroidRuntimeException(msg

+ " This message is already in use.");

}

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

throw new RuntimeException("Main thread not allowed to quit");

}

final boolean needWake;

synchronized (this) {

if (mQuiting) {

RuntimeException e = new RuntimeException(

msg.target + " sending message to a Handler on a dead thread");

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

return false;

} else if (msg.target == null) {

mQuiting = true;

}

msg.when = when;

//Log.d("MessageQueue", "Enqueing: " + msg);

Message p = mMessages;

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

msg.next = p;

mMessages = msg;

needWake = mBlocked; // new head, might need to wake up

} else {

Message prev = null;

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

prev = p;

p = p.next;

}

msg.next = prev.next;

prev.next = msg;

needWake = false; // still waiting on head, no need to wake up

}

}

if (needWake) {

nativeWake(mPtr);

}

return true;

}

首先你要知道，MessageQueue并没有使用一个集合把所有的消息都保存起来，它只使用了一个mMessages对象表示当前待处理的消息。然后观察上面的代码的16~31行我们就可以看出，所谓的入队其实就是将所有的消息按时间来进行排序，这个时间当然就是我们刚才介绍的uptimeMillis参数。具体的操作方法就根据时间的顺序调用msg.next，从而为每一个消息指定它的下一个消息是什么。当然如果你是通过sendMessageAtFrontOfQueue()方法来发送消息的，它也会调用enqueueMessage()来让消息入队，只不过时间为0，这时会把新入队的这条消息赋值给mMessages，然后将这条消息的next指定为刚才的mMessages，这样也就完成了添加消息到队列头部的操作。现在入队操作我们就已经看明白了，那出队操作是在哪里进行的呢?这个就需要看一看Looper.loop()方法的源码了，如下所示：

public static void loop() {

Looper me = myLooper();

if (me == null) {

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

}

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();

while (true) {

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

if (msg != null) {

if (msg.target == null) {

// No target is a magic identifier for the quit message.

return;

}

long wallStart = 0;

long threadStart = 0;

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

Printer logging = me.mLogging;

if (logging != null) {

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

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

wallStart = SystemClock.currentTimeMicro();

threadStart = SystemClock.currentThreadTimeMicro();

}

msg.target.dispatchMessage(msg);

if (logging != null) {

long wallTime = SystemClock.currentTimeMicro() - wallStart;

long threadTime = SystemClock.currentThreadTimeMicro() - threadStart;

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

if (logging instanceof Profiler) {

((Profiler) logging).profile(msg, wallStart, wallTime,

threadStart, threadTime);

}

}

// 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);

}

msg.recycle();

}

}

}

可以看到，这个方法从第13行开始，进入了一个死循环，然后不断地调用的MessageQueue的next()方法，我想你已经猜到了，这个next()方法就是消息队列的出队方法。不过由于这个方法的代码稍微有点长，我就不贴出来了，它的简单逻辑就是如果当前MessageQueue中存在mMessages(即待处理消息)，就将这个消息出队，然后让下一条消息成为mMessages，否则就进入一个阻塞状态，一直等到有新的消息入队。继续看loop()方法的第14行，每当有一个消息出队，就将它传递到msg.target的dispatchMessage()方法中，那这里msg.target又是什么呢？其实就是Handler啦，你观察一下上面sendMessageAtTime()方法的第6行就可以看出来了。接下来当然就要看一看Handler中dispatchMessage()方法的源码了，如下所示：

public void dispatchMessage(Message msg) {

if (msg.callback != null) {

handleCallback(msg);

} else {

if (mCallback != null) {

if (mCallback.handleMessage(msg)) {

return;

}

}

handleMessage(msg);

}

}

在第5行进行判断，如果mCallback不为空，则调用mCallback的handleMessage()方法，否则直接调用Handler的handleMessage()方法，并将消息对象作为参数传递过去。这样我相信大家就都明白了为什么handleMessage()方法中可以获取到之前发送的消息了吧！  
因此，一个最标准的异步消息处理线程的写法应该是这样：

class LooperThread extends Thread {

public Handler mHandler;

public void run() {

Looper.prepare();

mHandler = new Handler() {

public void handleMessage(Message msg) {

// process incoming messages here

}

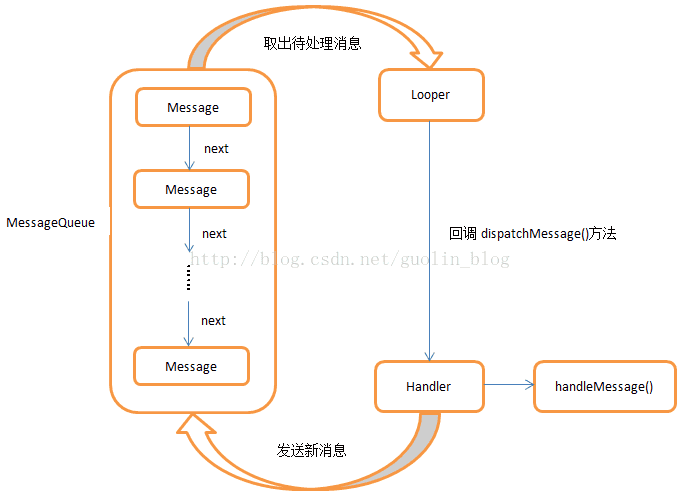
};

Looper.loop();

}

}

当然，这段代码是从Android官方文档上复制的，不过大家现在再来看这段代码，是不是理解的更加深刻了？  
那么我们还是要来继续分析一下，为什么使用异步消息处理的方式就可以对UI进行操作了呢？这是由于Handler总是依附于创建时所在的线程，比如我们的Handler是在主线程中创建的，而在子线程中又无法直接对UI进行操作，于是我们就通过一系列的发送消息、入队、出队等环节，最后调用到了Handler的handleMessage()方法中，这时的handleMessage()方法已经是在主线程中运行的，因而我们当然可以在这里进行UI操作了。整个异步消息处理流程的示意图如下图所示：



整个异步消息处理机制.png

另外除了发送消息之外，我们还有以下几种方法可以在子线程中进行UI操作：

Handler的post()方法

View的post()方法

Activity的runOnUiThread()方法

我们先来看下Handler中的post()方法，代码如下所示：

public final boolean post(Runnable r){

return sendMessageDelayed(getPostMessage(r), 0);

}

也太简单了！竟然就是直接调用了一开始传入的Runnable对象的run()方法。因此在子线程中通过Handler的post()方法进行UI操作就可以这么写：

public class MainActivity extends Activity {

private Handler handler;

@Override

protected void onCreate(Bundle savedInstanceState) {

super.onCreate(savedInstanceState);

setContentView(R.layout.activity\_main);

handler = new Handler();

new Thread(new Runnable() {

@Override

public void run() {

handler.post(new Runnable() {

@Override

public void run() {

// 在这里进行UI操作

}

});

}

}).start();

}

}

虽然写法上相差很多，但是原理是完全一样的，我们在Runnable对象的run()方法里更新UI，效果完全等同于在handleMessage()方法中更新UI。

然后再来看一下View中的post()方法，代码如下所示：

public boolean post(Runnable action) {

Handler handler;

if (mAttachInfo != null) {

handler = mAttachInfo.mHandler;

} else {

ViewRoot.getRunQueue().post(action);

return true;

}

return handler.post(action);

}

原来就是调用了Handler中的post()方法，我相信已经没有什么必要再做解释了。  
最后再来看一下Activity中的runOnUiThread()方法，代码如下所示：

public final void runOnUiThread(Runnable action) {

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

mHandler.post(action);

} else {

action.run();

}

}

如果当前的线程不等于UI线程(主线程)，就去调用Handler的post()方法，否则就直接调用Runnable对象的run()方法。还有什么会比这更清晰明了的吗？  
通过以上所有源码的分析，我们已经发现了，不管是使用哪种方法在子线程中更新UI，其实背后的原理都是相同的，必须都要借助异步消息处理的机制来实现，而我们又已经将这个机制的流程完全搞明白了，真是一件一本万利的事情啊。