[Android HandlerThread 详解](https://www.cnblogs.com/huansky/p/13828497.html)

**概述**

**HandlerThread** 相信大家都比较熟悉了，从名字上看是一个带有 Handler 消息循环机制的一个线程，比一般的线程多了消息循环的机制，可以说是 Handler + Thread 的结合，从源码上看也是如此的设计。

对 Handler 不熟悉的可以看 [Android Handler 源码分析（详细）](https://www.cnblogs.com/huansky/p/9427854.html) 一文，会教你一步步去认识 Handler 。

一般情况下如果需要子线程和主线程之间相互交互，可以用 HandlerThread 来设计，这比单纯的 Thread 要方便，而且更容易管理，因为大家都知道Thread 的生命周期在一些情况下是不可控制的，比如直接 new Thread().start() 这种方式在项目中是不推荐使用的，实际上 Android 的源码中也有很多地方用到了 HandlerThread，下面我将分析一下 HandlerThread 用法以及源码解析。

**使用示例**

[复制代码](javascript:void(0);)

// 实例对象，参数为线程名字

HandlerThread handlerThread = new HandlerThread("handlerThread");

// 启动线程

handlerThread.start();

// 参数为 HandlerThread 内部的一个 looper

Handler handler = new Handler(handlerThread.getLooper()) {

@Override

public void handleMessage(Message msg) {

super.handleMessage(msg);

}

};

[复制代码](javascript:void(0);)

**注意：**这个使用的顺序是不能更改的！！！，因为如果不先让子线程 start 起来，那么创建主线程的 handler 的参数 getLooper 是获取不到的，这一点可以看源码就清楚。

**Demo 详解**

这里模拟在子线程下载东西，然后和主线程之间进行通信。主线程知道了下载开始和下载结束的时间，也就能及时改变界面 UI。

首先是 DownloadThread 类，继承于 HandlerThread，用于下载。

[复制代码](javascript:void(0);)

public class DownloadThread extends HandlerThread{

private static final String TAG = "DownloadThread";

public static final int TYPE\_START = 2;//通知主线程任务开始

public static final int TYPE\_FINISHED = 3;//通知主线程任务结束

private Handler mUIHandler;//主线程的Handler

public DownloadThread(String name) {

super(name);

}

/\*

\* 执行初始化任务

\* \*/

@Override

protected void onLooperPrepared() {

Log.e(TAG, "onLooperPrepared: 1.Download线程开始准备");

super.onLooperPrepared();

}

//注入主线程Handler

public void setUIHandler(Handler UIhandler) {

mUIHandler = UIhandler;

Log.e(TAG, "setUIHandler: 2.主线程的handler传入到Download线程");

}

//Download线程开始下载

public void startDownload() {

Log.e(TAG, "startDownload: 3.通知主线程,此时Download线程开始下载");

mUIHandler.sendEmptyMessage(TYPE\_START);

//模拟下载

Log.e(TAG, "startDownload: 5.Download线程下载中...");

SystemClock.sleep(2000);

Log.e(TAG, "startDownload: 6.通知主线程,此时Download线程下载完成");

mUIHandler.sendEmptyMessage(TYPE\_FINISHED);

}

}

[复制代码](javascript:void(0);)

然后是 MainActivity 部分，UI 和处理消息。

[复制代码](javascript:void(0);)

public class MainActivity extends AppCompatActivity {

private static final String TAG = "MainActivity";

private DownloadThread mHandlerThread;//子线程

private Handler mUIhandler;//主线程的Handler

@Override

protected void onCreate(Bundle savedInstanceState) {

super.onCreate(savedInstanceState);

setContentView(R.layout.activity\_main);

//初始化，参数为线程的名字

mHandlerThread = new DownloadThread("mHandlerThread");

//调用start方法启动线程

mHandlerThread.start();

//初始化Handler，传递 mHandlerThread 内部的一个 looper

mUIhandler = new Handler(mHandlerThread.getLooper()) {

@Override

public void handleMessage(Message msg) {

//判断mHandlerThread里传来的msg，根据msg进行主页面的UI更改

switch (msg.what) {

case DownloadThread.TYPE\_START:

//不是在这里更改UI哦，只是说在这个时间，你可以去做更改UI这件事情，改UI还是得在主线程。

Log.e(TAG, "4.主线程知道Download线程开始下载了...这时候可以更改主界面UI");

break;

case DownloadThread.TYPE\_FINISHED:

Log.e(TAG, "7.主线程知道Download线程下载完成了...这时候可以更改主界面UI，收工");

break;

default:

break;

}

super.handleMessage(msg);

}

};

//子线程注入主线程的mUIhandler，可以在子线程执行任务的时候，随时发送消息回来主线程

mHandlerThread.setUIHandler(mUIhandler);

//子线程开始下载

mHandlerThread.startDownload();

}

@Override

protected void onDestroy() {

//有2种退出方式

mHandlerThread.quit();

//mHandlerThread.quitSafely(); 需要API>=18

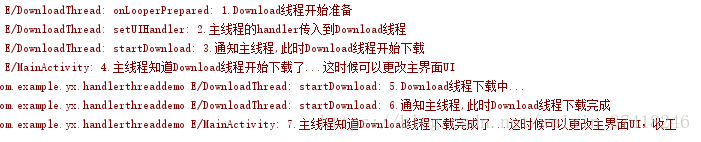
super.onDestroy();

}

}

[复制代码](javascript:void(0);)

运行的Log日志如下



**源码解析**

public class HandlerThread extends Thread {

 HandlerThread 本质是一个线程，只是其持有了 handler，所以可在子线程进行消息处理和分发。

接下去看下构造函数相关的：

[复制代码](javascript:void(0);)

int mPriority;//优先级

int mTid = -1;

Looper mLooper;//自带的Looper

private @Nullable Handler mHandler;

public HandlerThread(String name) {

super(name);

mPriority = Process.THREAD\_PRIORITY\_DEFAULT;

}

/\*\*

\* Constructs a HandlerThread.

\* @param name

\* @param priority The priority to run the thread at. The value supplied must be from

\* {@link android.os.Process} and not from java.lang.Thread.

\*/

public HandlerThread(String name, int priority) {

super(name);

mPriority = priority;

}

[复制代码](javascript:void(0);)

这里有两个构造方法，一个 HandlerThread(String name)，一个 HandlerThread(String name, int priority)，我们可以自己设定线程的名字以及优先级。注意！是 Process 里的优先级而不是Thread 的。

[复制代码](javascript:void(0);)

/\*\*

\* Call back method that can be explicitly overridden if needed to execute some

\* setup before Looper loops.

\*/

protected void onLooperPrepared() {

}

@Override

public void run() {

mTid = Process.myTid();

Looper.prepare();

synchronized (this) {

mLooper = Looper.myLooper();

notifyAll();

}

Process.setThreadPriority(mPriority);

onLooperPrepared();

Looper.loop();

mTid = -1;

}

[复制代码](javascript:void(0);)

这里面有一个方法 onLooperPrepared()，在实际中，我们可以重写这个方法做一些初始化的操作，这个 run() 是重点。

run 方法中首先获取线程 id,然后就调用了 Looper.prepare 方法创建一个 Looper,接着调用了 Looper.myLooper 方法获取到了当前线程的 Looper。

接着通过 notifyAll 通知等带唤醒，这里的等待是在 HandlerThread 的 getLooper 方法里调用的 wait 方法，getLooper 方法是为了获取该 HandlerThread 中的 Looper。

如果在没调用 HandlerThread 的 start 方法开启线程前就调用 getLooper 方法就通过 wait 方法暂时先进入等待，等到 run 方法运行后再进行唤醒。唤醒之后 run 方法中继续设置了构造函数中传入的优先级，接着调用了onLooperPrepared 方法，该方法是个空实现，该方法是为了在 Looper 开启轮询之前如果要进行某些设置，可以复写该方法。

最后调用Looper.loop开启轮询。退出的时候，将 mTid  = -1；

[复制代码](javascript:void(0);)

public Looper getLooper() {

if (!isAlive()) {

return null;

}

// If the thread has been started, wait until the looper has been created.

synchronized (this) {

while (isAlive() && mLooper == null) {

try {

wait();

} catch (InterruptedException e) {

}

}

}

return mLooper;

}

[复制代码](javascript:void(0);)

 这个方法是获取当前的 Looper，可以看到如果没有获取的时候就一直等待直到获取，而前面也提到了获取到了就唤醒了所有的线程，看来这是线程的等待-唤醒机制应用。

[复制代码](javascript:void(0);)

public Handler getThreadHandler() {

if (mHandler == null) {

mHandler = new Handler(getLooper());

}

return mHandler;

}

[复制代码](javascript:void(0);)

这个是获取 HandlerThread 绑定的 Looper 线程的 Handler

[复制代码](javascript:void(0);)

public boolean quit() {

Looper looper = getLooper();

if (looper != null) {

looper.quit();

return true;

}

return false;

}

public boolean quitSafely() {

Looper looper = getLooper();

if (looper != null) {

looper.quitSafely();

return true;

}

return false;

}

[复制代码](javascript:void(0);)

可以看到这两个方法去退出线程的 Looper 循环，那么这两个方法有什么区别呢,实际上都是调用了 MessageQueue 的 quit() 方法，源码如下:

[复制代码](javascript:void(0);)

void quit(boolean safe) {

if (!mQuitAllowed) {

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

}

synchronized (this) {

if (mQuitting) {

return;

}

mQuitting = true;

if (safe) {

removeAllFutureMessagesLocked();

} else {

removeAllMessagesLocked();

}

// We can assume mPtr != 0 because mQuitting was previously false.

nativeWake(mPtr);

}

}

[复制代码](javascript:void(0);)

可以看到: 当我们调用 quit 方法的时候，实际上执行了 MessageQueue 中的 removeAllMessagesLocked 方法，该方法的作用是把 MessageQueue 消息池中所有的消息全部清空，无论是延迟消息（延迟消息是指通过 sendMessageDelayed 或通过 postDelayed 等方法发送的需要延迟执行的消息，只要不是立即执行的消息都是延迟消息）还是非延迟消息。

而 quitSafely 方法时，实际上执行了 MessageQueue 中的 removeAllFutureMessagesLocked 方法，通过名字就可以看出，该方法只会清空 MessageQueue 消息池中所有的延迟消息，并将消息池中所有的非延迟消息派发出去让 Handler 去处理，quitSafely 相比于 quit 方法安全之处在于清空消息之前会派发所有的非延迟消息,一句话，就是清除未来需要执行的消息。

这两个方法有一个共同的特点就是：Looper 不再接收新的消息了,消息循环就此结束，此时通过 Handler 发送的消息也不会在放入消息杜队列了，因为消息队列已经退出了。应用这2个方法的时候需要注意的是：quit 方法从 API 1 就开始存在了，比较早，而 quitSafely 直到 API 18 才添加进来.

**总结**

* 如果经常要开启线程，接着又是销毁线程，这是很耗性能的，HandlerThread 很好的解决了这个问题；
* HandlerThread 由于异步操作是放在 Handler 的消息队列中的，所以是串行的，但只适合并发量较少的耗时操作。
* HandlerThread 用完记得调用退出方法。
* 注意使用 handler 避免出现内存泄露