**handler机制**

参考：<http://blog.csdn.net/lmj623565791/article/details/38377229/>

<https://blog.csdn.net/ashqal/article/details/32107099>

1. Handler

Handler.sendMessage(Message)发送消息

Handler.handleMessage(Message)处理消息

1. Looper：轮询器，通过Looper.loop()从MmessageQueue中取消息。
2. MessageQueue：消息队列
3. Message：消息实体，可以通过new Message()或者Message.obtain()，Message.obtain()可以维护一个消息池。
4. 主线程中的Handler
5. ActivityThread类中通过Looper.prepareMainLooper()方法创建Looper和MessageQueue

注：ActivityThread类的使用参考http://blog.csdn.net/luoshengyang/article/details/6747696

public static void prepareMainLooper() {

prepare(false);

synchronized (Looper.class) {

if (sMainLooper != null) {

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

}

sMainLooper = myLooper();

}

}

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));

}

private Looper(boolean quitAllowed) {

mQueue = new MessageQueue(quitAllowed);

mThread = Thread.currentThread();

}

1. Handler通过调用Looper.myLooper()获取当前线程的Looper对象，从而与Looper建立联系

public Handler(Callback callback, boolean async) {

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 = callback;

mAsynchronous = async;

}

1. Looper.prepare()方法中创建Looper并将其保存在ThreadLocal变量中，Looper.prepare()在每个线程中只允许调用一次，否则会报异常。

（每个Thread实例都有一个ThreadLocalMap，Looper.prepare()调用时，通过获取当前Thread实例，取出当前Thread中的ThreadLocalMap，并以当前ThreadLocal为key保存Looper； ThreadLocalMap通过数组保存键值对，数组初始长度为16，容量达到总长度的一半时，扩容为原来的两倍；ThreadLocalMap通过key的hash值对length取余，得到value保存初始位置，然后从该位置往后循环查找需要的键值对，知道找到或者数组对应位置是null为止）

1. 调用Looper.loop()方法，从MessageQueue队列中取出消息并通过Handler.dispatchMessage调用Handler.handleMessage()处理消息。

注：若消息队列没有消息，Looper.loop()阻塞（实际是调用MessageQueue#next()时阻塞）

public Handler() {

this(null, false);

}

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;

// 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();

for (;;) {

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

Printer logging = me.mLogging;

if (logging != null) {

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

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

}

msg.target.dispatchMessage(msg);

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);

}

msg.recycleUnchecked();

}

}

public void dispatchMessage(Message msg) {

if (msg.callback != null) {

handleCallback(msg);

} else {

if (mCallback != null) {

if (mCallback.handleMessage(msg)) {

return;

}

}

handleMessage(msg);

}

}

1. Handler.sendMessage()将消息发送到MessageQueue中，并调用enqueueMessage方法将msg.target赋为当前Handler

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

msg.target = this;

if (mAsynchronous) {

msg.setAsynchronous(true);

}

return queue.enqueueMessage(msg, uptimeMillis);

}

1. 子线程中的Handler（与主线程中的Handler类似，但是需要手动调用Looper.prepare()和Looper.loop()方法）

new Thread(new Runnable() {

@Override

public void run() {

String msg;

Looper.prepare();

childHandler = new Handler() {

@Override

public void handleMessage(Message msg) {

super.handleMessage(msg);

System.out.println("此消息来自-->>" + msg.obj+ "，在" + "btn的子线程当执行的")；

}

};

Looper.loop();//开始轮循

System.out.println("轮询结束"); // Looper.loop()之后的代码都不会被执行，因此这句语句得不到执行

}

}).start();

1. 每个Handler对象都会绑定一个Looper对象，每个Looper对象对应一个消息队列（MessageQueue）。如果在创建Handler时不指定与其绑定的Looper对象，系统默认会将通过Looper.myLooper()获取当前线程的Looper绑定到该Handler上。

注：向子线程通信前必须在该子线程调用Looper.prepare()，创建一个与该子线程绑定的Looper，否则会报运行时错误。

1. Looper对象与当前线程绑定，保证一个线程只会有一个Looper实例，同时一个Looper实例也只有一个MessageQueue。
2. Looper.prepare()每个线程只能调用一次，否则会抛出运行时错误。
3. Handler.post(Runnable)将Runnable封装成一个Message的callback，然后通过sendMessageDelayed()将该message发送出去。且并不会开启新线程，仍在Loop绑定的线程中执行。
4. Message.obtain()方法使用Message内部维护的一个Message池，用于Message的复用，避免使用new 重新分配内存。
5. Handler(Callback callback) 用来创建一个Handler的实例但并不需要派生Handler的子类。
6. Handler#dispatchMessage
7. 如果message的callback不为空，则执行message的callback并返回；
8. 反之判断构造函数传进来的成员变量mCallback是否为空，不为空则执行mCallback#run并返回
9. 否则直接执行Handler#handlerMessage
10. 如果消息队列没有消息，则Looper#loop()中的无限循环会被MessageQueue#next()阻塞，MessageQueue#next()判读没有消息时，会设置一个至少Integer.MAX\_VALUE毫秒的超时，在超时时间到了之后才会继续尝试取下一条数据。