**Android下的多线程（Looper的理解）**

在andorid中线程分为有消息循环的线程和没有消息循环的线程，有消息循环的线程一般都会有一个Looper，这个是andorid的新概念。我们的主线程（UI线程）就是一个消息循环线程，针对这种消息循环的机制，我们引入一个新的机制Handler，我们有消息循环，就要往消息循环里面发送相应的消息，自定义消息一般都会有自己对应的处理、消息的发送和清除、消息的处理都封装在Handle里面，主要Handler只是针对那些有Looper的线程，不管是UI线程还是子线程，只要你有Looper，我们就可以往你的消息队列里面添加东西并做相应的处理。注意：android中关于UI的不能放在子线程处理，因为子线程是不能操作UI的，只能进行数据、系统等其他非UI的操作。

我们在创建Handler时候，里面包含了一个Looper，我们的子线程要变成消息循环线程，需要调用Looper.prepare()函数：

**public** **static** **void** prepare() {

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

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

}

sThreadLocal.set(**new** Looper());

}

从上面的代码中我们可以看出，实际上是给该线程设置了一个Looper，同时保证在一个线程中只能存在一个Looper对象，再看看Looper的构造函数：

**private** **static** Looper mMainLooper = **null**; // guarded by Looper.class

**private** Looper() {

mQueue = **new** MessageQueue();

mRun = **true**;

mThread = Thread.currentThread();

}

实际上是创建一个消息队列，同时将该Looper与当前线程关联，同时我们发现构造函数是私有的，也就是说我们不能new一个Looper出来，我们要让自己的线程产生一个消息队列就必须要调用Looper.prepare()函数了。

**我们再看看Handler的构造函数：**

**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**) {//表示没有Looper，则没有消息队列

**throw** **new** RuntimeException(

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

}

mQueue = mLooper.mQueue;

mCallback = **null**;

}

**16 Handler消息传递机制**

http://www.cnblogs.com/codingmyworld/archive/2011/09/14/2174255.html

Android中的消息传递机制是另一种“事件处理”，这种机制主要是为了解决android中的多线程问题—android平台不允许Activity新启动的线程访问该Activity里的界面组件，这就会导致新启动的线程无法动态改变界面组件的属性值。

Handler类的主要作用有两个：

1. 在新启动的线程中发送消息
2. 在主线程中获取、处理消息

Android的消息处理有三个核心类：Looper、Handler和Message，其实还有一个Message Queue（消息队列），但是MQ被封装在Looper里面了。

**Handler消息的发送：**

通常我们在调用Handler的sendEmptyMessage时候调用了下面的函数：

**public** **boolean** sendMessageAtTime(Message msg, **long** uptimeMillis)

{

**boolean** sent = **false**;

MessageQueue queue = mQueue;

**if** (queue != **null**) {

**msg.target = this;//在这里定义了消息是由哪个Handler发出的**

sent = queue.enqueueMessage(msg, uptimeMillis);

}

**else** {

RuntimeException e = **new** RuntimeException(

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

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

}

**return** sent;

}

**Handler消息的处理：**

将普通线程变换为Looper线程需要调用Looper.prepare()与Looper.loop()

**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);//target实际上就是Handler，用于处理消息**

**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();

}

}

}

**Handler的dispatchMessage函数：（调用handleMessage函数，即我们需要重写的）**

**public** **void** dispatchMessage(Message msg) {

**if** (msg.callback != **null**) {

handleCallback(msg);

} **else** {

**if** (mCallback != **null**) {

**if** (mCallback.handleMessage(msg)) {

**return**;

}

}

handleMessage(msg);

}

}

**Message类：**

在整个消息处理机制中，message又叫task，封装了任务携带的信息和处理该任务的handler，使用需要注意下面几点：

1. 尽管Message有public方法，但是应该通过Message.oain来从消息池中获取，节省资源
2. 如果你的message比较简单，优先使用Message.arg1和Message.arg2，这逼用Bundle更节省内存
3. 用Message.what标识信息，以便用不同方式处理message