**Android应用程序的消息处理机制**

Android 应用程序是通过消息来驱动的，系统为每一个应用程序维护一个消息队列（MessageQueue），应用程序不断地从这个消息队列里拿消息（Looper），然后对这些消息处理（Handler），这样就实现了通过消息来驱动的应用程序的执行。它是由消息循环，消息发送，消息处理三部分组成。

参考http://blog.csdn.net/luoshengyang/article/details/6817933

知识延伸：为什么子线程中不能访问UI？

Android的UI控件不是线程安全的，如果在多线程中并发访问UI控件会出现不可预期的状态。为啥系统不对UI控件加上锁的机制？首先给UI控件加上锁的机制会让UI控件的访问逻辑变得很复杂，其次锁机制会降低UI控件的访问效率，因为锁机制会阻塞一些线程的执行。

* **消息循环**

在消息处理机制中，消息都是存放在一个消息队列中的，而应用程序的主线程（UI线程）则是对这个消息队列无限循环的遍历，直到应用退出。如果队列中有消息，应用程序的主线程会把它取出来，交给相应的Handler处理。如果队列中没有消息，应用程序的主线程会进入到空闲状态，等待下一个消息的到来。这个消息循环是由Looper来实现的。

**先看一下应用程序的主线程是如何进入到消息循环中去的**：

1. **public** **final** **class** ActivityThread {
2. ......
4. **public** **static** **final** **void** main(String[] args) {
5. ......
7. Looper.prepareMainLooper();
9. ......
11. ActivityThread thread = **new** ActivityThread();
12. thread.attach(**false**);
14. ......
16. Looper.loop();
18. ......
19. }
20. }

主线程做了两件事：第一，创建了一个ActivityThread实例。第二，创建了Looper的实例，并使主线程进入到无限循环中。我们只看第二件事：

**public final class** Looper {  
   
 *// sThreadLocal.get() will return null unless you've called prepare().* **static final** ThreadLocal<Looper> ***sThreadLocal*** = **new** ThreadLocal<Looper>();  
 **private static** Looper *sMainLooper*; *// guarded by Looper.class* **final** MessageQueue **mQueue**;  
 **final** Thread **mThread**;

**private** Looper(**boolean** quitAllowed) {  
 **mQueue** = **new** MessageQueue(quitAllowed);  
 **mThread** = Thread.*currentThread*();  
 }  
 */\*\* Initialize the current thread as a looper.  
 \* This gives you a chance to create handlers that then reference  
 \* this looper, before actually starting the loop. Be sure to call  
 \* {****@link*** *#loop()} after calling this method, and end it by calling  
 \* {****@link*** *#quit()}.  
 \*/* **public static void** prepare() {  
 *prepare*(**true**);  
 }  
  
 **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));  
 }  
  
 */\*\*  
 \* Initialize the current thread as a looper, marking it as an  
 \* application's main looper. The main looper for your application  
 \* is created by the Android environment, so you should never need  
 \* to call this function yourself. See also: {****@link*** *#prepare()}  
 \*/* **public static void** prepareMainLooper() {  
 *prepare*(**false**);  
 **synchronized** (Looper.**class**) {  
 **if** (*sMainLooper* != **null**) {  
 **throw new** IllegalStateException(**"The main Looper has already been prepared."**);  
 }  
 *sMainLooper* = *myLooper*();  
 }  
 }  
  
 */\*\*  
 \* Returns the application's main looper, which lives in the main thread of the application.  
 \*/* **public static** Looper getMainLooper() {  
 **synchronized** (Looper.**class**) {  
 **return** *sMainLooper*;  
 }  
 }

}

Looper.prepareMainLooper()函数主要做的事情是在主线程中创建一个Looper对象，并存放在sThreadLocal对象中。在创建Looper对象的同时会还会创建一个消息队列MessageQueue，保存在Looper的成员变量mQueue中，后续的消息都是存在这个队列中。

**public final class** MessageQueue {  
 Message **mMessages**;//**总是指向队列的第一个消息。**

@SuppressWarnings(**"unused"**)  
 **private long mPtr**; *// used by native code*  *...*  
 **private native static long** nativeInit();  
 ...  
 MessageQueue(**boolean** quitAllowed) {  
 **mQuitAllowed** = quitAllowed;  
 **mPtr** = *nativeInit*();  
 }

...

}

MessageQueue的初始化是由JNI层的nativeInit来实现的。我们这里不深入剖析。上述Looper的创建过程做了如下事情：

1. 在java层创建了Looper对象，这个Looper对象使用来进入消息循环的，并同时创建一个 MessageQueue对象mQueue。
2. 创建MessageQueue对象mQueue时，MessageQueue的构造方法中，在JNI层创建了一个 NativeMessageQueue对象，这个NativeMessageQueue保存在java层的消息队列对象mQueue的 成员变量mPtr中。
3. 创建NativeMessageQueue对象时，NativeMessageQueue内部，在c++层创建了一个Looper对象， 保存在JNI层的NativeMessageQueue对象的成员变量mLooper中，这个对象的作用是，当java 层的消息队列没有消息时，使应用程序主线程进入到 等待状态，而当java层的消息队列来了消 息之后，就唤醒应用程序主线程来处理这个消息。

回到ActivityThread类的main函数中，准备好上面的数据之后，就开始调用Looper类的loop函数进入到无限循环中去了：

**public static void** loop() {  
 **final** Looper me = *myLooper*();  
 **。。。**  
 **final** MessageQueue queue = me.mQueue;  
 。。。  
 **for** (;;) {  
 Message msg = queue.next(); *// might block* **if** (msg == **null**) {  
 *// No message indicates that the message queue is quitting.* **return**;  
 }  
 。。。  
 msg.target.dispatchMessage(msg);  
 。。。  
 msg.recycleUnchecked();  
 }  
}

进入到无限循环后，不断从消息队列里去获取下一个消息msg，如果msg为空的话，就要退出这个循环了，否则就调用msg的target成员变量的dispatchMessage（）方法来处理这个消息，这个target就是发送这条消息的Handler。这里最关键的就是获取消息队列的下一条消息这个方法MessageQueue.next函数了

说明：

MessageQueue内部是由一个单链表的数据结构来维护消息队列的，MessageQueue主要做两件事，插入一条消息和读取一条消息（读取的同时会删除这条消息），而单链表对于插入和删除是很有优势的。

Message next() {  
 *// Return here if the message loop has already quit and been disposed.  
 // This can happen if the application tries to restart a looper after quit  
 // which is not supported.* **final long** ptr = **mPtr**;  
 **if** (ptr == 0) {  
 **return null**;  
 }  
  
 **int** pendingIdleHandlerCount = -1; *// -1 only during first iteration* **int** nextPollTimeoutMillis = 0;  
 **for** (;;) {  
 **if** (nextPollTimeoutMillis != 0) {  
 Binder.*flushPendingCommands*();  
 }  
  
 nativePollOnce(ptr, nextPollTimeoutMillis);  
  
 **synchronized** (**this**) {  
 *// Try to retrieve the next message. Return if found.* **final long** now = SystemClock.*uptimeMillis*();  
 Message prevMsg = **null**;  
 Message msg = **mMessages**;  
 **if** (msg != **null** && msg.target == **null**) {  
 *// Stalled by a barrier. Find the next asynchronous message in the queue.* **do** {  
 prevMsg = msg;  
 msg = msg.next;  
 } **while** (msg != **null** && !msg.isAsynchronous());  
 }  
 **if** (msg != **null**) {  
 **if** (now < msg.when) {  
 *// Next message is not ready. Set a timeout to wake up when it is ready.* nextPollTimeoutMillis = (**int**) Math.min(msg.when - now, Integer.***MAX\_VALUE***);  
 } **else** {  
 *// Got a message.* **mBlocked** = **false**;  
 **if** (prevMsg != **null**) {  
 prevMsg.next = msg.next;  
 } **else** {  
 **mMessages** = msg.next;  
 }  
 msg.next = **null**;  
 **if** (***DEBUG***) Log.*v*(***TAG***, **"Returning message: "** + msg);  
 msg.markInUse();  
 **return** msg;  
 }  
 } **else** {  
 *// No more messages.* nextPollTimeoutMillis = -1;  
 }  
  
 *// Process the quit message now that all pending messages have been handled.* **if** (**mQuitting**) {  
 dispose();  
 **return null**;  
 }  
  
 *// If first time idle, then get the number of idlers to run.  
 // Idle handles only run if the queue is empty or if the first message  
 // in the queue (possibly a barrier) is due to be handled in the future.* **if** (pendingIdleHandlerCount < 0  
 && (**mMessages** == **null** || now < **mMessages**.when)) {  
 pendingIdleHandlerCount = **mIdleHandlers**.size();  
 }  
 **if** (pendingIdleHandlerCount <= 0) {  
 *// No idle handlers to run. Loop and wait some more.* **mBlocked** = **true**;  
 **continue**;  
 }  
  
 **if** (**mPendingIdleHandlers** == **null**) {  
 **mPendingIdleHandlers** = **new** IdleHandler[Math.*max*(pendingIdleHandlerCount, 4)];  
 }  
 **mPendingIdleHandlers** = **mIdleHandlers**.toArray(**mPendingIdleHandlers**);  
 }  
  
 *// Run the idle handlers.  
 // We only ever reach this code block during the first iteration.* **for** (**int** i = 0; i < pendingIdleHandlerCount; i++) {  
 **final** IdleHandler idler = **mPendingIdleHandlers**[i];  
 **mPendingIdleHandlers**[i] = **null**; *// release the reference to the handler* **boolean** keep = **false**;  
 **try** {  
 keep = idler.queueIdle();  
 } **catch** (Throwable t) {  
 Log.*wtf*(***TAG***, **"IdleHandler threw exception"**, t);  
 }  
  
 **if** (!keep) {  
 **synchronized** (**this**) {  
 **mIdleHandlers**.remove(idler);  
 }  
 }  
 }  
  
 *// Reset the idle handler count to 0 so we do not run them again.* pendingIdleHandlerCount = 0;  
  
 *// While calling an idle handler, a new message could have been delivered  
 // so go back and look again for a pending message without waiting.* nextPollTimeoutMillis = 0;  
 }  
}

首先会判断mPtr假如为0则返回一个空消息，当Looper的循环中收到这个空消息之后便会退出循环。

继续执行nativePollOnce(ptr,nextPollTimeoutMills);这是一个JNI方法，它能根据nextPollTimeoutMills的值使线程进入到阻塞状态：

ptr：指向在JNI层创建的NativeMessageQueue的对象。

nextPollTimeoutMills：表示当前线程要等待的时间，-1表示无限等待状态，0表示不等待。 当大于0时代表某个消息的延迟执行时间。for循环开始时传入0。

nativePollOnce方法带入参数ptr取回前面在JNI层创建的NativeMessageQueue对象，然后调用它的pollOnce函数，这个函数又会调用前面在C++层创建的mLooper对象的pollOnce函数，假如带入nextPollTimeoutMills参数不为0（-1或者大于0），那么这个函数会使线程睡眠，当java层的消息队列有新消息或者到达nextPollTimeoutMills的等待时间，那么这个函数就会唤醒应用程序主线程并返回，相应的nativePollOnce方法也会返回，就去查看消息队列中有没有消息。

如果有：

如果当前时候now大于等于消息设置的延迟执行时间，那么就把这个消息返回给Looper的loop方法，否则nextPollTimeoutMills记录最新消息延迟等待剩余时间。

说明：

延迟时间的计算的关键方法SystemClock.uptimeMillis()它返回的是从开机到当前时间的毫秒数（不包括手机睡眠时间），在Handler中发送消息时消息的延迟时间是这个函数返回的毫秒数加上设置的延迟毫秒数（如果没有设置延迟时间，则默认是0），这样上面那个延迟时间的判断就好理解了。

如果没有：将nextPollTimeoutMills设置为-1则进入无穷等待时间。

这里线程进入到等待状态，有两种情况，1、消息队列中没有消息。2、消息设置了延迟执行时间，并且还没有到这个延迟时间。**在进入到空闲等待前，如果注册了IdleHandler来处理一些事情，那么就会先执行这些IdleHandler**然后重置nextPollTimeoutMills为0，这样等到线程下次调用nativePollOnce方法时就不用设置超时时间了，这是因为当线程执行IdleHandler时很有可能有新的消息加入到消息队列中去了。如果没有注册IdleHandler即pendingIdleHandlerCount等于0通过continue语句直接进入到下一次循环。以此类推，直到消息被返回。这样next函数就分析完了。

* **消息发送**

应用程序的主线程准备就好消息队列并且进入到消息循环后，其它地方就可以往这个消息队列中发送消息了。但是ActivityThread还需要一个Handler来和消息队列进行交互，这个Handler就是ActivityThread.H，他继承自Handler，ActivityThread内部有一个H的实例mH，mH在main函数中ActivityThread初始化的时候被创建，由于mH继承自Handler，所以当创建mH时会调用Handler的构造函数：

**public** Handler(Callback callback, **boolean** async) {  
 。。。  
 **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;  
}  
**public** Handler(Looper looper, Callback callback, **boolean** async) {  
 **mLooper** = looper;  
 **mQueue** = looper.mQueue;  
 **mCallback** = callback;  
 **mAsynchronous** = async;  
}

Handler有两个构造函数，主要都是给mLooper和mQueue赋值，但是这两个是有区别的，第一个是通过Looper.myLooper()获取当前线程的Looper实例。第二个是可以传递任何一个线程的Looper实例来构造Handler，有了这个Looper对象之后就可以通过Looper.mQueue访问应用程序的消息队列了。

有了mH对象之后就可以通过sendXXX向消息队列列添加消息了：

注意：Handler有两种发送消息的方式：

1、sendXXX。

2、postXXX。

第二种发送方式是发送一个Runnable对象然后这个对象被封装到Message中，最终调用

sendXXX方法发送的，唯一不同点是处理消息的方式不同，具体见后面分析。

**public final boolean** sendMessageDelayed(Message msg, **long** delayMillis)  
 {  
 **if** (delayMillis < 0) {  
 delayMillis = 0;  
 }  
 **return** sendMessageAtTime(msg, SystemClock.*uptimeMillis*() + delayMillis);  
 }

发送消息时是可以指定消息的延迟处理时间段的，如果不指定延迟时间段，默认就是当前时间SystemClock.uptimeMillis()点，即表示马上要处理。假如设置了，则会获得当前时间然后加上设置的延迟时间段，即得到这个消息延迟的绝对时间点，在MessageQueue的next()方法中也会用这个函数来计算延迟时间的。最后把这个消息添加到队列中：

**private boolean** enqueueMessage(MessageQueue queue, Message msg, **long** uptimeMillis) {  
 msg.target = **this**;  
 **if** (**mAsynchronous**) {  
 msg.setAsynchronous(**true**);  
 }  
 **return** queue.enqueueMessage(msg, uptimeMillis);  
 }

首先设置这个这个消息的目标对象target，即这个消息最终是由谁来处理的。这里设置为this，表示这个消息由当前这个Handler对象来处理。随后调用MessageQueue对象的

enqueueMessage方法：

**boolean** enqueueMessage(Message msg, **long** when) {  
 **。。。**  
 **synchronized** (**this**) {  
 。。。  
 msg.markInUse();  
 msg.when = when;  
 Message p = **mMessages**;  
 **boolean** needWake;  
 **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** {  
 *// 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**;  
}

把消息添加到消息队列中分两种情况：

**第一种：**当前消息队列为空，这时候应用程序处于空闲等待状态，只要把消息添加到消息的头，然后唤醒线程就可以了：

。。。

msg.next = p;  
 **mMessages** = msg;  
 needWake = **mBlocked**;

。。。

*nativeWake*(**mPtr**);

这里通过nativeWake(mptr)方法唤醒线程，它是一个JNI方法，这个方法会根据传进的mptr参数找到JNI层的NativeMessageQueue对象，然后调用这个对象的wake函数，这个函数又调用c++层的mLooper成员变量的wake函数。

我们上面说到当消息队列没有消息时线程在MessageQueue的next()方法中调用的JNI方法nativePollOnce里阻塞住，nativePollOnce最终是调用c++层mLooper成员变量的pollOnce函数睡眠的，当c++层的mLooper成员变量的wake函数返回后，应用程序的主线程就会从这里的Looper类的pollInner函数返回到JNI层的nativePollOnce函数，最后返回到Java层中的MessageQueue.next函数中去，这里它就会发现消息队列中有新的消息需要处理了，于就会处理这个消息。

**第二种：**当前消息队列不为空，我们前面说过发送的消息可以指定它的延迟处理时间，而消息队列中的消息，就是按照这个时间从小到大来排序的，因此，当把消息加入到消息队列时，就要根据它的延迟时间来找合适的位置，然后在放到消息列表中。

* **消息处理**

前面分析消息循环时，说到应用程序的主线程是在Looper类的loop成员函数中进行消息循环过程的：

**public static void** loop() {  
 **final** Looper me = *myLooper*();  
 **。。。**  
 **final** MessageQueue queue = me.mQueue;  
 。。。  
 **for** (;;) {  
 Message msg = queue.next(); *// might block* **if** (msg == **null**) {  
 *// No message indicates that the message queue is quitting.* **return**;  
 }  
 。。。  
 msg.target.dispatchMessage(msg);  
 。。。  
 msg.recycleUnchecked();  
 }  
}

当msg返回后就调用它的target成员变量的的dispatchMessage()方法来处理这个消息，这个target代表发送消息的Handler，即哪个Handler发送的消息，这个消息就由哪个Handler的dispatchMessage()方法来处理，具体在哪个线程里处理不是根据Handler对象创建在哪个线程，就在哪个线程处理，而是这个Handler对象，内部的mLooper对象属于哪个线程，这个消息的最终处理就是在哪个线程：

**public void** dispatchMessage(Message msg) {  
 **if** (msg.callback != **null**) {  
 *handleCallback*(msg);  
 } **else** {  
 **if** (**mCallback** != **null**) {  
 **if** (**mCallback**.handleMessage(msg)) {  
 **return**;  
 }  
 }  
 handleMessage(msg);  
 }  
}

首先判断msg的callback是否为空，Message的callback是一个Runnable对象如果不为空则由这个Runnable来处理消息。这个参数是通过Handler.postXXX(Runnable run)获得的：

Runnable **run** =**new** Runnable() {  
 @Override  
 **public void** run() {  
 。。。  
 }  
 };

**handler**.post(**run**);

然后判断mCallback是否为空，mCallback是个接口，如果我们设置了这个接口程序执行到了这里，并且处理方法返回了true那么只有这个接口来处理这个消息，假如返回了false那么handleMessage（msg）也会处理这个消息，如下：

Handler **handler** = **new** Handler(**new** Handler.Callback() {  
 @Override  
 **public boolean** handleMessage(Message msg) {  
 **return false**;  
 }  
 });

Handler **handler** = **new** Handler() {  
 @Override  
 **public void** handleMessage(Message msg) {  
 **super**.handleMessage(msg);  
   
 }  
 };

* **补充**
* **如何创建一个常驻线程**：

**public class** MyThread **extends** Thread **implements** Handler.Callback {  
 **private** Handler **handler**;  
  
 @Override  
 **public void** run() {  
 **super**.run();  
 Looper.*prepare*();  
 **handler** = **new** Handler(**this**);  
 Looper.*loop*();  
 }  
  
 @Override  
 **public boolean** handleMessage(Message msg) {  
 **return false**;  
 }  
}

这个常驻线程也是可以退出的，Looper提供了以下两个方法来退出一个Looper：

quit:直接退出。

quitSafely:设定一个退出标记，然后把消息队列中已有的消息处理完毕后才安全 的退出。

当调用了以上两个方法后，MessageQueue的next()方法会返回null，也就是说Looper必须退出。

* **ThreadLocal**

ThreadLocal是线程本地存储区（Thread Local Storage，简称为TLS），每个线程都有自己的私有存储区，ThreadLocal在哪个线程存储的数据，那么这个数据就只能在那个线程中获取，其他线程获取不到：

**final** ThreadLocal<String> threadLocal = **new** ThreadLocal<>();  
Thread thread1 = **new** Thread(**new** Runnable() {  
 @Override  
 **public void** run() {  
 threadLocal.set(**"thread1"**);  
 Log.*i*(**"线程1获取data："**, **""** + threadLocal.get());  
 }  
});  
thread1.start();  
  
Thread thread2 = **new** Thread(**new** Runnable() {  
 @Override  
 **public void** run() {  
 threadLocal.set(**"thread2"**);  
 Log.*i*(**"线程2获取data："**, **""** + threadLocal.get());  
 }  
});  
thread2.start();  
  
Thread thread3 = **new** Thread(**new** Runnable() {  
 @Override  
 **public void** run() {  
 Log.*i*(**"线程3获取data："**, **""** + threadLocal.get());  
 }  
});  
thread3.start();

打印结果为：

**线程1获取data：thread1**

**线程2获取data：thread2**

**线程3获取data：null**