## Handler的使用和原理

### Handler经典使用:

**private** Handler **mhandler** = **new** Handler(){  
 @Override  
 **public void** handleMessage(Message msg) {  
 **super**.handleMessage(msg);  
 **switch** (msg.**what**) {  
 **case** 1:  
 *//主线程处理UI逻辑* **break**;  
 **default**:  
 **break**;  
  
 }  
 }  
};  
  
@Override  
**protected void** onCreate(Bundle savedInstanceState) {  
 **super**.onCreate(savedInstanceState);  
 //子线程发送消息  
 **new** Thread(**new** Runnable() {  
 @Override  
 **public void** run() {  
 Message message = Message.*obtain*();  
 message.**what** = 1;  
 message.**obj** = **""**;  
 **mhandler**.sendMessage(message);  
 }  
 }).start();

}

### Handler原理:

Android系统的事件驱动是通过Handler进行的.

//<http://androidxref.com/9.0.0_r3/xref/frameworks/base/core/java/android/app/ActivityThread.java>

**public** **static** **void** **[main](http://androidxref.com/9.0.0_r3/s?refs=main&project=frameworks)**([String](http://androidxref.com/9.0.0_r3/s?defs=String&project=frameworks)[] **[args](http://androidxref.com/9.0.0_r3/s?refs=args&project=frameworks)**) {

[6625](http://androidxref.com/9.0.0_r3/xref/frameworks/base/core/java/android/app/ActivityThread.java" \l "6625)

[6626](http://androidxref.com/9.0.0_r3/xref/frameworks/base/core/java/android/app/ActivityThread.java" \l "6626)

[6641](http://androidxref.com/9.0.0_r3/xref/frameworks/base/core/java/android/app/ActivityThread.java" \l "6641) [Looper](http://androidxref.com/9.0.0_r3/s?defs=Looper&project=frameworks).[prepareMainLooper](http://androidxref.com/9.0.0_r3/s?defs=prepareMainLooper&project=frameworks)();

[6643](http://androidxref.com/9.0.0_r3/xref/frameworks/base/core/java/android/app/ActivityThread.java" \l "6643)

[6655](http://androidxref.com/9.0.0_r3/xref/frameworks/base/core/java/android/app/ActivityThread.java" \l "6655) [ActivityThread](http://androidxref.com/9.0.0_r3/s?defs=ActivityThread&project=frameworks) [thread](http://androidxref.com/9.0.0_r3/s?defs=thread&project=frameworks) = **new** [ActivityThread](http://androidxref.com/9.0.0_r3/s?defs=ActivityThread&project=frameworks)();

[6656](http://androidxref.com/9.0.0_r3/xref/frameworks/base/core/java/android/app/ActivityThread.java" \l "6656) [thread](http://androidxref.com/9.0.0_r3/s?defs=thread&project=frameworks).[**attach**](http://androidxref.com/9.0.0_r3/xref/frameworks/base/core/java/android/app/ActivityThread.java#attach)(**false**, [**startSeq**](http://androidxref.com/9.0.0_r3/xref/frameworks/base/core/java/android/app/ActivityThread.java#startSeq));

[6657](http://androidxref.com/9.0.0_r3/xref/frameworks/base/core/java/android/app/ActivityThread.java" \l "6657)

[6658](http://androidxref.com/9.0.0_r3/xref/frameworks/base/core/java/android/app/ActivityThread.java" \l "6658) **if** (**[sMainThreadHandler](http://androidxref.com/9.0.0_r3/xref/frameworks/base/core/java/android/app/ActivityThread.java" \l "sMainThreadHandler)** == [null](http://androidxref.com/9.0.0_r3/s?defs=null&project=frameworks)) {

[6659](http://androidxref.com/9.0.0_r3/xref/frameworks/base/core/java/android/app/ActivityThread.java" \l "6659) [**sMainThreadHandler**](http://androidxref.com/9.0.0_r3/xref/frameworks/base/core/java/android/app/ActivityThread.java#sMainThreadHandler) = [thread](http://androidxref.com/9.0.0_r3/s?defs=thread&project=frameworks).[**getHandler**](http://androidxref.com/9.0.0_r3/xref/frameworks/base/core/java/android/app/ActivityThread.java#getHandler)();

[6660](http://androidxref.com/9.0.0_r3/xref/frameworks/base/core/java/android/app/ActivityThread.java" \l "6660) }

[6661](http://androidxref.com/9.0.0_r3/xref/frameworks/base/core/java/android/app/ActivityThread.java" \l "6661)

[6662](http://androidxref.com/9.0.0_r3/xref/frameworks/base/core/java/android/app/ActivityThread.java" \l "6662) **if** (**false**) {

[6663](http://androidxref.com/9.0.0_r3/xref/frameworks/base/core/java/android/app/ActivityThread.java" \l "6663) [Looper](http://androidxref.com/9.0.0_r3/s?defs=Looper&project=frameworks).[myLooper](http://androidxref.com/9.0.0_r3/s?defs=myLooper&project=frameworks)().[setMessageLogging](http://androidxref.com/9.0.0_r3/s?defs=setMessageLogging&project=frameworks)(**new**

[6664](http://androidxref.com/9.0.0_r3/xref/frameworks/base/core/java/android/app/ActivityThread.java" \l "6664) [LogPrinter](http://androidxref.com/9.0.0_r3/s?defs=LogPrinter&project=frameworks)([Log](http://androidxref.com/9.0.0_r3/s?defs=Log&project=frameworks).[DEBUG](http://androidxref.com/9.0.0_r3/s?defs=DEBUG&project=frameworks), "ActivityThread"));

[6665](http://androidxref.com/9.0.0_r3/xref/frameworks/base/core/java/android/app/ActivityThread.java" \l "6665) }

[6666](http://androidxref.com/9.0.0_r3/xref/frameworks/base/core/java/android/app/ActivityThread.java" \l "6666)

[6669](http://androidxref.com/9.0.0_r3/xref/frameworks/base/core/java/android/app/ActivityThread.java" \l "6669) [Looper](http://androidxref.com/9.0.0_r3/s?defs=Looper&project=frameworks).[loop](http://androidxref.com/9.0.0_r3/s?defs=loop&project=frameworks)();

[6670](http://androidxref.com/9.0.0_r3/xref/frameworks/base/core/java/android/app/ActivityThread.java" \l "6670)

[6671](http://androidxref.com/9.0.0_r3/xref/frameworks/base/core/java/android/app/ActivityThread.java" \l "6671) **throw** **new** [RuntimeException](http://androidxref.com/9.0.0_r3/s?defs=RuntimeException&project=frameworks)("Main thread loop unexpectedly exited");

[6672](http://androidxref.com/9.0.0_r3/xref/frameworks/base/core/java/android/app/ActivityThread.java" \l "6672) }

Handler的核心成员变量:

Looper:分发事件,与Handler所在的线程绑定

MessageQueue:事件队列对象,数据结构Message是一个链表, MessageQueue是Looper的成员变量,在Lopper的构造函数创建.,与MessageQueue绑定.

Handler所在的线程必须有一个Looper,否则会抛出异常,所以在创建Handler的时候一定要先保证创建Handler的线程里面已经存在一个Looper,这样创建一个Handler,Looper,MessageQueue就都创建出来了

//android/os/Handler.java

**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 "** + Thread.*currentThread*()  
 + **" that has not called Looper.prepare()"**);  
 }  
 **mQueue** = **mLooper**.mQueue;  
 **mCallback** = callback;  
 **mAsynchronous** = async;  
}

Handler发送消息事件到成员变量MessageQueue的数据结构Message队列中:

*/\*\*  
 \* Enqueue a message into the message queue after all pending messages  
 \* before the absolute time (in milliseconds) <var>uptimeMillis</var>.  
 \* <b>The time-base is {****@link*** *android.os.SystemClock#uptimeMillis}.</b>  
 \* Time spent in deep sleep will add an additional delay to execution.  
 \* You will receive it in {****@link*** *#handleMessage}, in the thread attached  
 \* to this handler.  
 \*   
 \** ***@param uptimeMillis*** *The absolute time at which the message should be  
 \* delivered, using the  
 \* {****@link*** *android.os.SystemClock#uptimeMillis} time-base.  
 \*   
 \** ***@return*** *Returns true if the message was successfully placed in to the   
 \* message queue. Returns false on failure, usually because the  
 \* looper processing the message queue is exiting. Note that a  
 \* result of true does not mean the message will be processed -- if  
 \* the looper is quit before the delivery time of the message  
 \* occurs then the message will be dropped.  
 \*/*

**public boolean** sendMessageAtTime(Message msg, **long** uptimeMillis) {  
 MessageQueue queue = **mQueue**;  
 **if** (queue == **null**) {  
 RuntimeException e = **new** RuntimeException(  
 **this** + **" sendMessageAtTime() called with no mQueue"**);  
 Log.*w*(**"Looper"**, e.getMessage(), e);  
 **return false**;  
 }  
 **return** enqueueMessage(queue, msg, uptimeMillis);  
}

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

//链表的按照时间从小到大排列,时间小的在表头

**boolean** enqueueMessage(Message msg, **long** when) {  
**synchronized** (**this**) {  
 msg.markInUse();  
 msg.when = when;  
 Message p = **mMessages**;  
 //如果消息队列为空或者时间为0即需要立即处理或者此消息小于表头的 消息时间,就直接把此消息放在表头,否则遍历第一个大于此消息需要处理的时间的消息然后插入这个消息的前面  
 **if** (p == **null** || when == 0 || when < p.when) {  
 *// New head, wake up the event queue if blocked.* msg.next = p;  
 **mMessages** = msg;  
   
 } **else** {  
   
 Message prev;  
 **for** (;;) {  
 prev = p;  
 p = p.next;  
 **if** (p == **null** || when < p.when) {  
 **break**;  
 }  
   
 }  
 msg.next = p; *// invariant: p == prev.next* prev.next = msg;  
 }  
  
 }  
 **return true**;  
}

Looper分发事件:

*//android/os/Looper.java*

*/\*\*  
 \* Run the message queue in this thread. Be sure to call  
 \* {****@link*** *#quit()} to end the loop.  
 \*/*

**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;  
 **for** (;;) {  
 Message msg = queue.next(); *// might block* **if** (msg == **null**) {  
 *// No message indicates that the message queue is quitting.* **return**;  
 }  
  
  
 **try** {  
 msg.target.dispatchMessage(msg);  
   
 } **finally** {  
   
 msg.recycleUnchecked();  
 }  
}

// android/os/MessageQueue.java

Message next() {  
**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;  
 }  
  
 }  
}

*/\*\*  
 \* Handle system messages here.  
 \*/*

**//处理消息****//android/os/Handler.java**

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

Looper与线程的绑定:

Looper是在Handler的构造方法中通过Looper.*myLooper*()创建的:

*/\*\*  
 \* Return the Looper object associated with the current thread. Returns  
 \* null if the calling thread is not associated with a Looper.  
 \*/*

*//* *android/os/Looper.java*

*静态对象只存在一个*

**static final** ThreadLocal<Looper> ***sThreadLocal*** = **new** ThreadLocal<Looper>();

**public static** @Nullable Looper myLooper() {  
 **return *sThreadLocal***.get();  
}

*/\*\*  
 \* Returns the value in the current thread's copy of this  
 \* thread-local variable. If the variable has no value for the  
 \* current thread, it is first initialized to the value returned  
 \* by an invocation of the {****@link*** *#initialValue} method.  
 \*  
 \** ***@return*** *the current thread's value of this thread-local  
 \*/*

*//* *java/lang/ThreadLocal.java***public** T get() {

//与当前线程绑定  
 Thread t = Thread.*currentThread*();

//实际存储数据在当前线程的ThreadLocalMap对象中, ThreadLocalMap是ThreadLocal的静态内部类,map的key是this指的是当前的***sThreadLocal对象,value是存储的具体数据,这里指的是Looper对象,因为sThreadLocal是静态的所以只存在一个,所以Looper在当前线程中只会存在一个***  
 ThreadLocalMap map = getMap(t);  
 **if** (map != **null**) {  
 ThreadLocalMap.Entry e = map.getEntry(**this**);  
 **if** (e != **null**) {  
 @SuppressWarnings(**"unchecked"**)  
 T result = (T)e.**value**;  
 **return** result;  
 }  
 }  
 **return** setInitialValue();  
}

*/\*\*  
 \* Get the map associated with a ThreadLocal. Overridden in  
 \* InheritableThreadLocal.  
 \*  
 \** ***@param t*** *the current thread  
 \** ***@return*** *the map  
 \*/*ThreadLocalMap getMap(Thread t) {  
 **return** t.threadLocals;  
}

*/\*\*  
 \* Variant of set() to establish initialValue. Used instead  
 \* of set() in case user has overridden the set() method.  
 \*  
 \** ***@return*** *the initial value  
 \*/*

**private** T setInitialValue() {  
 T value = initialValue();  
 Thread t = Thread.*currentThread*();  
 ThreadLocalMap map = getMap(t);  
 **if** (map != **null**)

//如果存在map就直接设置进去  
 map.set(**this**, value);  
 **else**

//不存在就创建一个当前线程的ThreadLocalMap对象createMap(t, value);  
 **return** value;  
}

*/\*\*  
 \* Create the map associated with a ThreadLocal. Overridden in  
 \* InheritableThreadLocal.  
 \*  
 \** ***@param t*** *the current thread  
 \** ***@param firstValue*** *value for the initial entry of the map  
 \*/***void** createMap(Thread t, T firstValue) {  
 t.threadLocals = **new** ThreadLocalMap(**this**, firstValue);  
}