# [TODO](http://gityuan.com/2015/12/26/handler-message-framework/)

同步栅栏概念和使用

<http://gityuan.com/2015/12/27/handler-message-native/>

可以用两个handler，为何可以了，性能如何影响

1.5.10.2. Looper.myQueue().addIdleHandler(new Idler());

<http://gityuan.com/2016/09/24/android-thread/>

isPolling原理

ProMS应用于ThreadLocal

Ctong的，借鉴

# Java层

## 概述

在整个Android的源码世界里，有两大通信利剑，其一是Binder IPC机制，另一个便是消息机制(由Handler/Looper/MessageQueue等构成的)

Android有大量的消息驱动方式来进行交互，比如Android的四剑客Activity, Service, Broadcast, ContentProvider的启动过程的交互，都离不开消息机制，Android某种意义上也可以说成是一个以消息驱动的系统。

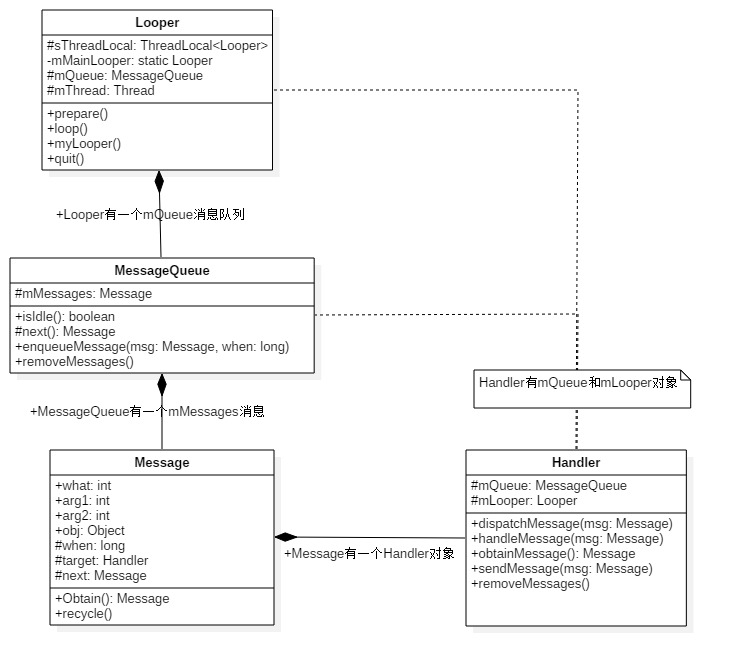
消息机制涉及Message/MessageQueue /Looper/Handler这4个类。

### 模型

消息机制主要包含：

* Message：消息分为硬件产生的消息(如按钮、触摸)和软件生成的消息；
* MessageQueue：消息队列的主要功能向消息池投递消息(MessageQueue.enqueueMessage)和取走消息池的消息(MessageQueue.next)；
* Handler：消息辅助类，主要功能向消息池发送各种消息事件(Handler.sendMessage)和处理相应消息事件(Handler.handleMessage)；
* Looper：不断循环执行(Looper.loop)，按分发机制将消息分发给目标处理者

### 架构图



Looper有一个MessageQueue消息队列；

MessageQueue有一组待处理的Message；

Message中有一个用于处理消息的Handler；

Handler中有Looper和MessageQueue

### 典型实例

典型关于Handler/Looper的线程如下, 接下来，围绕着这个实例展开详细分析

**class** LooperThread **extends** Thread {  
 **public** Handler **mHandler**;  
 **public void** run() {  
 Looper.*prepare*(); *//【见 2.1】* **mHandler** = **new** Handler() { *//【见 3.1】* **public void** handleMessage(Message msg) {  
 *//****TODO 定义消息处理逻辑. 【见 3.2】*** }  
 };  
 Looper.*loop*(); *//【见 2.2】* }  
}

## Looper

### prepare()

对于无参的情况，默认调用prepare(true)

**public static void** prepare() {  
 *prepare*(**true**);  
}

**private static void** prepare(**boolean** quitAllowed) {  
 *//每个线程只允许执行一次该方法，第二次执行时线程的TLS已有数据，则会抛出异常。* **if** (sThreadLocal.get() != **null**) {  
 **throw new** RuntimeException(**"Only one Looper may be created per thread"**);  
 }  
 *//创建Looper对象，并保存到当前线程的TLS区域* sThreadLocal.set(**new** Looper(quitAllowed));  
}

quitAllowed表示的是这个Looper运行退出，而对于false的情况则表示当前Looper不运行退出。具体可看MessageQueue

这里的sThreadLocal是ThreadLocal类型，下面，先说说ThreadLocal。

#### ThreadLocal

ThreadLocal： 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的本地存储区域，不同线程之间彼此不能访问对方的TLS区域。TLS常用的操作方法：

##### ThreadLocal.set(T value)

将value存储到当前线程的TLS区域，源码如下：

**public void** set(T value) {  
 Thread currentThread = **Thread.*currentThread*();** *//获取当前线程* Values values = values(currentThread); *//查找当前线程的本地储存区* **if** (values == **null**) {  
 *//当线程本地存储区，尚未存储该线程相关信息时，则创建Values对象* values = initializeValues(currentThread);  
 }  
 *//保存数据value到当前线程this* values.put(**this**, value);  
}

##### ThreadLocal.get()

ThreadLocal.get()：获取当前线程TLS区域的数据，源码如下：

**public** T get() {  
 Thread currentThread = Thread.*currentThread*(); *//获取当前线程* Values values = values(currentThread); *//查找当前线程的本地储存区* **if** (values != **null**) {  
 Object[] table = values.table;  
 **int** index = hash & values.mask;  
 **if** (**this**.reference == table[index]) {  
 **return** (T) table[index + 1]; *//返回当前线程储存区中的数据* }  
 } **else** {  
 *//创建Values对象* values = initializeValues(currentThread);  
 }  
 **return** (T) values.getAfterMiss(**this**); *//从目标线程存储区没有查询是则返回null*}

ThreadLocal的get()和set()方法操作的类型都是泛型.

##### ThreadLocal的工作原理

ThreadLocal是一个线程内部的数据存储类，通过它可以在指定的线程中存储数据，数据存储以后，只有在指定线程中可以获取到存储的数据，对于其他线程来说则无法获取到数据。

在日常开发中用到ThreadLocal的地方较少，但是在某些特殊的场景下，通过ThreadLocal可以轻松地实现一些看起来很复杂的功能，这一点在Android的源码中也有所体现，比如Looper、ActivityThread以及AMS中都用到了ThreadLocal。

具体到ThreadLocal的使用场景，这个不好统一来描述，一般来说，当某些数据是以**线程为作用域**并且不同线程具有不同的数据副本的时候，就可以考虑采用ThreadLocal。比如对于Handler来说，它需要获取当前线程的Looper，**很显然Looper的作用域就是线程并且不同线程具有不同的Loope**r，这个时候通过ThreadLocal就可以轻松实现Looper在线程中的存取。如果不采用ThreadLocal，那么系统就必须提供一个全局的哈希表供Handler查找指定线程的Looper，这样一来就必须提供一个类似于LooperManager的类了，但是系统并没有这么做而是选择了ThreadLocal，这就是ThreadLocal的好处。

ThreadLocal另一个使用场景是复杂逻辑下的对象传递，比如监听器的传递，有些时候一个线程中的任务过于复杂，这可能表现为函数调用栈比较深以及代码入口的多样性，在这种情况下，我们又需要监听器能够贯穿整个线程的执行过程，这个时候可以怎么做呢？其实这时就可以采用ThreadLocal，采用ThreadLocal可以让监听器作为线程内的全局对象而存在，**在线程内部只要通过get方法就可以获取到监听器**。如果不采用ThreadLocal，那么我们能想到的可能是如下两种方法：第一种方法是将监听器通过参数的形式在函数调用栈中进行传递，第二种方法就是将监听器作为静态变量供线程访问。上述这两种方法都是有局限性的。第一种方法的问题是当函数调用栈很深的时候，通过函数参数来传递监听器对象这几乎是不可接受的，这会让程序的设计看起来很糟糕。**第二种方法是可以接受的，但是这种状态是不具有可扩充性的**，比如同时有两个线程在执行，那么就需要提供两个静态的监听器对象，如果有10个线程在并发执行呢？提供10个静态的监听器对象？这显然是不可思议的，而采用ThreadLocal，每个监听器对象都在自己的线程内部存储，根据就不会有方法2的这种问题。

介绍了那么多ThreadLocal的知识，可能还是有点抽象，下面通过实例来演示ThreadLocal的真正含义。首先定义一个ThreadLocal对象，这里选择Boolean类型的，如下所示。代码如下，结果看注释部分：



在上面的代码中，在主线程中设置mBooleanThreadLocal的值为true，在子线程1中设置mBooleanThreadLocal的值为false，在子线程2中不设置mBooleanThreadLocal的值。然后分别在3个线程中通过get方法去mBooleanThreadLocal的值，根据前面对ThreadLocal的描述，这个时候，主线程中应该是true，子线程1中应该是false，而子线程2中由于没有设置值，所以应该是null。

从上面日志可以看出，虽然在不同线程中访问的是同一个ThreadLocal对象，但是它们通过ThreadLocal获取到的值却是不一样的，这就是ThreadLocal的奇妙之处。结合这这个例子然后再看一遍前面对ThreadLocal的两个使用场景的理论分析，我们应该就能比较好地理解ThreadLocal的使用方法了。ThreadLocal之所以有这么奇妙的效果，是因为不同线程访问同一个ThreadLocal的get方法，ThreadLocal内部会从各自的线程中取出一个数组，然后再从数组中根据当前ThreadLocal的索引去查找出对应的value值，很显然，不同线程中的数组是不同的，这就是为什么通过ThreadLocal可以在不同的线程中维护一套数据的副本并且彼此互不干扰。

对ThreadLocal的使用方法和工作过程做了介绍后，下面分析ThreadLocal的内部实现， ThreadLocal是一个泛型类，它的定义为public class ThreadLocal<T>，只要弄清楚ThreadLocal的get和set方法就可以明白它的工作原理。

首先看ThreadLocal的set方法，如下所示。

Get和set的源码如下，api24和其他版本的可能不一样，其实是内部维护了一个HashMap！！！

**public void** set(T value) {  
 Thread t = Thread.*currentThread*();  
 ThreadLocalMap map = getMap(t);  
 **if** (map != **null**)  
 map.set(**this**, value);  
 **else** createMap(t, value);  
}

**public** T get() {  
 Thread t = Thread.*currentThread*();  
 ThreadLocalMap map = getMap(t);  
 **if** (map != **null**) {  
 ThreadLocalMap.Entry e = map.getEntry(**this**);  
 **if** (e != **null**)  
 **return** (T)e.**value**;  
 }  
 **return** setInitialValue();  
}

接着回到前面提到的sThreadLocal变量，其定义如下：

static final ThreadLocal<Looper> sThreadLocal = new ThreadLocal<Looper>()，可见sThreadLocal的get()和set()操作的类型都是Looper类型。

#### Looper()

Looper.prepare()在每个线程只允许执行一次，该方法会创建Looper对象，Looper的构造方法中会创建一个MessageQueue对象，再将Looper对象保存到当前线程TLS。

对于Looper类型的构造方法如下：

private Looper(boolean quitAllowed) {

mQueue = new MessageQueue(quitAllowed); //创建MessageQueue对象. 【见4.1】

mThread = Thread.currentThread(); //记录当前线程.

}

另外，与prepare()相近功能的，还有一个prepareMainLooper()方法，该方法主要在ActivityThread类中使用

### prepareMainLooper

**public static void** prepareMainLooper() {  
 *prepare*(**false**); //设置不允许退出的Looper

//将当前的Looper保存为主Looper，每个线程只允许执行一次。  
 **synchronized** (Looper.**class**) {  
 **if** (*sMainLooper* != **null**) {  
 **throw new** IllegalStateException(**"The main Looper has already been prepared."**);  
 }  
 *sMainLooper* = *myLooper*();  
 }  
}

### loop()

**public static void** loop() {  
 **final** Looper me = *myLooper*();//获取TLS存储的Looper对象 【见2.4】  
 **if** (me == **null**) {  
 **throw new** RuntimeException(**"No Looper; Looper.prepare() wasn't called on this thread."**);  
 }  
 **final** MessageQueue queue = me.mQueue; //获取Looper对象中的消息队列  
  
 *// Make sure the identity of this thread is that of the local process,  
 // and keep track of what that identity token actually is.*

//确保在权限检查时基于本地进程，而不是基于最初调用进程。Binder.*clearCallingIdentity*();  
 **final long** ident = Binder.*clearCallingIdentity*();  
  
 **for** (;;) {//进入loop的主循环方法  
 Message msg = queue.next(); *// might block*//可能会阻塞 【见4.2】**if** (msg == **null**) {//没有消息，则退出循环  
 *// 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*

//默认为null，可通过setMessageLogging()方法来指定输出，用于debug功能**final** Printer logging = me.mLogging;  
 **if** (logging != **null**) {  
 logging.println(**">>>>> Dispatching to "** + msg.target + **" "** +  
 msg.callback + **": "** + msg.**what**);  
 }  
  
 **final long** traceTag = me.mTraceTag;  
 **if** (traceTag != 0 && Trace.*isTagEnabled*(traceTag)) {  
 Trace.*traceBegin*(traceTag, msg.target.getTraceName(msg));  
 }  
 **try** {

//用于分发Message 【见3.2】  
 msg.target.dispatchMessage(msg);  
 } **finally** {  
 **if** (traceTag != 0) {  
 Trace.*traceEnd*(traceTag);  
 }  
 }  
  
 **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.*

//确保分发过程中identity不会损坏**final long** newIdent = Binder.*clearCallingIdentity*();  
 **if** (ident != newIdent) {//打印identity改变的log，在分发消息过程中是不希望身份被改变的。  
 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**);  
 }  
//将Message放入消息池 【见5.2】  
 msg.recycleUnchecked();  
 }  
}

loop()进入循环模式，不断重复下面的操作，直到没有消息时退出循环

1. 读取MessageQueue的下一条Message；
2. 把Message分发给相应的target；
3. 再把分发后的Message回收到消息池，以便重复利用。

这是这个消息处理的核心部分。另外，上面代码中可以看到有logging方法，这是用于debug的，默认情况下logging == null，通过设置setMessageLogging()用来开启debug工作。

### quit()

public void quit() {

mQueue.quit(false); //消息移除

}

public void quitSafely() {

mQueue.quit(true); //安全地消息移除

}

Looper.quit()方法的实现最终调用的是MessageQueue.quit()方法

### 其他方法

#### myLooper

用于获取TLS存储的Looper对象

public static @Nullable Looper myLooper() {

return sThreadLocal.get();

}

## Handler

Handler的工作主要包含消息的发送和接收过程。消息的发送可以通过post的一系列方法以及send的一系列方法，post的一系列方法最终是通过send的一系列方法来实现的。发送一条消息的典型过程如下所示。sendMessage，sendMessageDelayed，sendMessageAtTime，enqueueMessage

### Handler()

#### 无参looper构造

下面看一下Handler的一个默认构造方法Handler()，这个构造方法会调用下面的构造方法。很明显，如果当前线程没有Looper的话，就会抛出“Can't create handler inside thread that has not called Looper.prepare()”这个异常，这也解释了在没有Looper的子线程中创建Handler会引发程序异常的原因。

**public** Handler() {  
 **this**(**null**, **false**);  
}

|  |
| --- |
| **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; } |

对于Handler的无参构造方法，默认采用当前线程TLS中的Looper对象，并且callback回调方法为null，且消息为同步处理方式。只要执行的Looper.prepare()方法，那么便可以获取有效的Looper对象

**private static final boolean *FIND\_POTENTIAL\_LEAKS*** = **false**;

**public** Handler(Callback callback, **boolean** async) {  
 **if** (***FIND\_POTENTIAL\_LEAKS***) {

//匿名类、内部类或本地类都必须申明为static，否则会警告可能出现内存泄露  
 **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());  
 }  
 }  
//必须先执行Looper.prepare()，才能获取Looper对象，否则为null.  
 **mLooper** = Looper.*myLooper*();//从当前线程的TLS中获取Looper对象【见2.1】  
 **if** (**mLooper** == **null**) {  
 **throw new** RuntimeException(  
 **"Can't create handler inside thread that has not called Looper.prepare()"**);  
 }  
 **mQueue** = **mLooper**.mQueue; //消息队列，来自Looper对象  
 **mCallback** = callback; //回调方法  
 **mAsynchronous** = async; //设置消息是否为异步处理方式  
}

#### 有looper构造

Handler还有一个特殊的构造方法，那就是通过一个特定的Looper来构造Handler，它的实现如下所示。通过这个构造方法可以实现一些特殊的功能，然后调用重载的多参数方法。

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

public Handler(Looper looper) {

this(looper, null, false);

}

public Handler(Looper looper, Callback callback, boolean async) {

mLooper = looper;

mQueue = looper.mQueue;

mCallback = callback;

mAsynchronous = async;

}

Handler类在构造方法中，可指定Looper，Callback回调方法以及消息的处理方式(同步或异步)，对于无参的handler，默认是当前线程的Looper

### dispatchMessage(消息分发机制)

可以发现，Handler发送消息的过程仅仅是向消息队列中插入了一条消息，MessageQueue的next方法就会返回这条消息给Looper，Looper收到消息后就开始处理了，最终消息由Looper交由Handler处理，即Handler的dispatchMessage方法会被调用，这时Handler就进入了处理消息的阶段。dispatchMessage的实现如下所示：

在Looper.loop()中，当发现有消息时，调用消息的目标handler，执行dispatchMessage()方法来分发消息。

|  |
| --- |
| **public void** dispatchMessage(Message msg) {  **if** (msg.callback != **null**) {  //当Message存在回调方法，回调msg.callback.run()方法；  *handleCallback*(msg);  } **else** {  **if** (**mCallback** != **null**) {  //当Handler存在Callback成员变量时，回调方法handleMessage()；  **if** (**mCallback**.handleMessage(msg)) {  **return**;  }  }  //Handler自身的回调方法handleMessage()  handleMessage(msg);  } } |

分发消息流程概要：

1. 当Message的回调方法不为空时，则回调方法msg.callback.run()，其中callBack数据类型为Runnable,否则进入步骤2；
2. 当Handler的mCallback成员变量不为空时，则回调方法mCallback.handleMessage(msg),否则进入步骤3；
3. 调用Handler自身的回调方法handleMessage()，该方法默认为空，**Handler子类通过覆写该方法来完成具体的逻辑。**

对于很多情况下，消息分发后的处理方法是第3种情况，即Handler.handleMessage()，一般地往往通过覆写该方法从而实现自己的业务逻辑。**mCallback用的很少，是在构造**Handler赋值的

Handler处理消息的过程如下：

1.首先检查Message的callback是否为null，不为null就通过handleCallback来处理消息。**Message的callback是一个Runnable对象**，**实际上就是Handler的**post(Runnable r)**方法所传递的Runnable参数**。handleCallback就是执行message.callback.run();

2.其次检查mCallback是否为null，不为null就调用mCallback的handleMessage方法来处理消息。Callback是个仅含handleMessage的接口。通过Callback可以采用如下方式来创建Handler对象：Handler handler = new Handler(callback)。那么Callback的意义是什么呢？源码里面的注释已经做了说明：可以用来创建一个Handler的实例但并不需要派生Handler的子类。在日常开发中，创建Handler最常见的方式就是派生一个Handler的子类并重写其handleMessage方法来处理具体的消息，而Callback给我们提供了另外一种使用Handler的方式，当我们不想派生子类时，就可以通过Callback来实现。 简称**匿名线程**

|  |
| --- |
| **new** Thread(**new** Runnable() {  @Override  **public void** run() {*//  /\* \*/* } }).start(); |

3.最后，调用Handler的handleMessage方法来处理消息。Handler处理消息的过程可以归纳为一个流程图，如图10-2所示。



### 消息发送

#### post

发送消息，并设置消息的callback，用于处理消息

**public final boolean** post(Runnable r)  
{  
 **return** sendMessageDelayed(*getPostMessage*(r), 0);  
}

**private static** Message getPostMessage(Runnable r) {  
 Message m = Message.*obtain*();  
 m.callback = r;  
 **return** m;  
}

**public final boolean** postAtFrontOfQueue(Runnable r)  
{  
 **return** sendMessageAtFrontOfQueue(*getPostMessage*(r));  
}

**public final boolean** sendMessageAtFrontOfQueue(Message msg) {  
 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, 0);  
}

#### send

最终条用enqueueMessage

sendEmptyMessage

public final boolean sendEmptyMessage(int what) {

return sendEmptyMessageDelayed(what, 0);

}

public final boolean sendEmptyMessageDelayed(int what, long delayMillis) {

Message msg = Message.obtain();

msg.what = what;

return sendMessageDelayed(msg, delayMillis);

}

public final boolean sendMessageDelayed(Message msg, long delayMillis) {

if (delayMillis < 0) {

delayMillis = 0;

}

return sendMessageAtTime(msg, SystemClock.uptimeMillis() + delayMillis);

}

public boolean sendMessageAtTime(Message msg, long uptimeMillis) {

MessageQueue queue = mQueue;

if (queue == null) {

return false;

}

return enqueueMessage(queue, msg, uptimeMillis);

}

sendMessageAtFrontOfQueue

public final boolean sendMessageAtFrontOfQueue(Message msg) {

MessageQueue queue = mQueue;

if (queue == null) {

return false;

}

return enqueueMessage(queue, msg, 0);

}

该方法通过设置消息的触发时间为0，从而使Message加入到消息队列的队头。

#### enqueueMessage

**private boolean** enqueueMessage(MessageQueue queue, Message msg, **long** uptimeMillis) {  
 msg.target = **this**;  
 **if** (**mAsynchronous**) {  
 msg.setAsynchronous(**true**);  
 }  
 **return** queue.enqueueMessage(msg, uptimeMillis); 【见4.3】  
}

#### 小节

Handler.sendEmptyMessage()等系列方法最终调用MessageQueue.enqueueMessage(msg, uptimeMillis)，将消息添加到消息队列中，其中uptimeMillis为系统当前的运行时间，不包括休眠时间。

### 其他方法

#### obtainMessage

获取消息

public final Message obtainMessage() {

return Message.obtain(this); 【见5.2】

}

Handler.obtainMessage()方法，最终调用Message.obtainMessage(this)，其中this为当前的Handler对象。

#### removeMessages

public final void removeMessages(int what) {

mQueue.removeMessages(this, what, null); 【见 4.5】

}

Handler是消息机制中非常重要的辅助类，更多的实现都是MessageQueue, Message中的方法，Handler的目的是为了更加方便的使用消息机制

## MessageQueue

MessageQueue是消息机制的Java层和C++层的连接纽带，大部分核心方法都交给native层来处理，其中MessageQueue类中涉及的native方法如下：

private native static long nativeInit();

private native static void nativeDestroy(long ptr);

private native void nativePollOnce(long ptr, int timeoutMillis);

private native static void nativeWake(long ptr);

private native static boolean nativeIsPolling(long ptr);

private native static void nativeSetFileDescriptorEvents(long ptr, int fd, int events);

关于这些native方法的介绍，见Android消息机制-native篇。

### 创建MessageQueue

MessageQueue(boolean quitAllowed) {

mQuitAllowed = quitAllowed;

//通过native方法初始化消息队列，其中mPtr是供native代码使用

mPtr = nativeInit();

}

### next()

提取下一条message

Message **next()** {

final long ptr = mPtr;

if (ptr == 0) { //当消息循环已经退出，则直接返回

return null;

}

int pendingIdleHandlerCount = -1; // 循环迭代的首次为-1

int nextPollTimeoutMillis = 0;

for (;;) {

if (nextPollTimeoutMillis != 0) {

Binder.flushPendingCommands();

}

//阻塞操作，当等待nextPollTimeoutMillis时长，或者消息队列被唤醒，都会返回

nativePollOnce(ptr, nextPollTimeoutMillis);

synchronized (this) {

final long now = SystemClock.uptimeMillis();

Message prevMsg = null;

Message msg = mMessages;

if (msg != null && msg.target == null) {

//当消息Handler为空时，查询MessageQueue中的下一条异步消息msg，则退出循环。

do {

prevMsg = msg;

msg = msg.next;

} while (msg != null && !msg.isAsynchronous());

}

if (msg != null) {

if (now < msg.when) {

//当异步消息触发时间大于当前时间，则设置下一次轮询的超时时长

nextPollTimeoutMillis = (int) Math.min(msg.when - now, Integer.MAX\_VALUE);

} else {

// 获取一条消息，并返回

mBlocked = false;

if (prevMsg != null) {

prevMsg.next = msg.next;

} else {

mMessages = msg.next;

}

msg.next = null;

//设置消息的使用状态，即flags |= FLAG\_IN\_USE

msg.markInUse();

return msg; //成功地获取MessageQueue中的下一条即将要执行的消息

}

} else {

//没有消息

nextPollTimeoutMillis = -1;

}

//消息正在退出，返回null

if (mQuitting) {

dispose();

return null;

}

//当消息队列为空，或者是消息队列的第一个消息时

if (pendingIdleHandlerCount < 0 && (mMessages == null || now < mMessages.when)) {

pendingIdleHandlerCount = mIdleHandlers.size();

}

if (pendingIdleHandlerCount <= 0) {

//没有idle handlers 需要运行，则循环并等待。

mBlocked = true;

continue;

}

if (mPendingIdleHandlers == null) {

mPendingIdleHandlers = new IdleHandler[Math.max(pendingIdleHandlerCount, 4)];

}

mPendingIdleHandlers = mIdleHandlers.toArray(mPendingIdleHandlers);

}

//只有第一次循环时，会运行idle handlers，执行完成后，重置pendingIdleHandlerCount为0.

for (int i = 0; i < pendingIdleHandlerCount; i++) {

final IdleHandler idler = mPendingIdleHandlers[i];

mPendingIdleHandlers[i] = null; //去掉handler的引用

boolean keep = false;

try {

keep = idler.queueIdle(); //idle时执行的方法

} catch (Throwable t) {

Log.wtf(TAG, "IdleHandler threw exception", t);

}

if (!keep) {

synchronized (this) {

mIdleHandlers.remove(idler);

}

}

}

//重置idle handler个数为0，以保证不会再次重复运行

pendingIdleHandlerCount = 0;

//当调用一个空闲handler时，一个新message能够被分发，因此无需等待可以直接查询pending message.

nextPollTimeoutMillis = 0;

}

}

nativePollOnce是阻塞操作，其中nextPollTimeoutMillis代表下一个消息到来前，还需要等待的时长；当nextPollTimeoutMillis = -1时，表示消息队列中无消息，会一直等待下去。

当处于空闲时，往往会执行IdleHandler中的方法。当nativePollOnce()返回后，next()从mMessages中提取一个消息。

nativePollOnce()在native做了大量的工作，想进一步了解可查看 Android消息机制2-Handler(native篇)。

这里可以解释如果native和java的消息处理时间一模一样，应该是先处理native的！

Message next() {

nativePollOnce(ptr, nextPollTimeoutMillis);

Message msg = mMessages;

mMessages = msg.next;

msg.next = null;

return msg;

}

### enqueueMessage

添加一条消息到消息队列

boolean enqueueMessage(Message **msg**, long when) {

// 每一个普通Message必须有一个target

if (msg.target == null) {

throw new IllegalArgumentException("Message must have a target.");

}

if (msg.isInUse()) {

throw new IllegalStateException(msg + " This message is already in use.");

}

synchronized (this) {

if (mQuitting) { //正在退出时，回收msg，加入到消息池

IllegalStateException e = new IllegalStateException(

msg.target + " sending message to a Handler on a dead thread");

Log.w(TAG, e.getMessage(), e);

msg.recycle();

return false;

}

msg.markInUse();

msg.when = when;

Message p = mMessages;

boolean needWake;

if (p == null || when == 0 || when < p.when) {

//p为null(代表MessageQueue没有消息） 或者msg的触发时间是队列中最早的， 则进入该该分支

msg.next = p;

mMessages = msg;

needWake = mBlocked; //当阻塞时需要唤醒

} else {

//将消息按时间顺序插入到MessageQueue。一般地，不需要唤醒事件队列，除非

//消息队头存在barrier，并且同时Message是队列中最早的异步消息。

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;

prev.next = msg;

}

//消息没有退出，我们认为此时mPtr != 0

if (needWake) {

nativeWake(mPtr);

}

}

return true;

}

MessageQueue是按照Message触发时间的先后顺序排列的，队头的消息是将要最早触发的消息。当有消息需要加入消息队列时，会从队列头开始遍历，直到找到消息应该插入的合适位置，以保证所有消息的时间顺序。

#### 案例分析

Fg

06-16 16:08:03.354 3098-3098/com.android.systemui W/MessageQueue: Handler (android.view.ViewRootImpl$ViewRootHandler) {e2ca046} sending message to a Handler on a dead thread

java.lang.IllegalStateException: Handler (android.view.ViewRootImpl$ViewRootHandler) {e2ca046} sending message to a Handler on a dead thread

at android.os.MessageQueue.enqueueMessage(MessageQueue.java:543)

at android.os.Handler.enqueueMessage(Handler.java:647)

at android.os.Handler.sendMessageAtTime(Handler.java:616)

at android.view.ViewRootImpl$ViewRootHandler.sendMessageAtTime(ViewRootImpl.java:3453)

at android.os.Handler.sendMessageDelayed(Handler.java:586)

at android.os.Handler.sendEmptyMessageDelayed(Handler.java:550)

at android.os.Handler.sendEmptyMessage(Handler.java:535)

at android.view.ViewRootImpl.die(ViewRootImpl.java:5904)

at android.view.WindowManagerGlobal.removeViewLocked(WindowManagerGlobal.java:446)

at android.view.WindowManagerGlobal.removeView(WindowManagerGlobal.java:384)

at android.view.WindowManagerImpl.removeView(WindowManagerImpl.java:119)

at com.kpad.systemui.recorder.ui.FloatingManager.removeView(FloatingManager.java:59)

at com.kpad.systemui.recorder.ui.FloatingView.hide(FloatingView.java:122)

at com.kpad.systemui.recorder.RecorderActivity2$4.onReceive(RecorderActivity2.java:323)

收到了录屏开始的广播，正准备显示页面，然后又立即关闭了

### removeMessages

oid removeMessages(Handler h, int what, Object object) {

if (h == null) {

return;

}

synchronized (this) {

Message p = mMessages;

//从消息队列的头部开始，移除所有符合条件的消息

while (p != null && p.target == h && p.what == what

&& (object == null || p.obj == object)) {

Message n = p.next;

mMessages = n;

p.recycleUnchecked();

p = n;

}

//移除剩余的符合要求的消息

while (p != null) {

Message n = p.next;

if (n != null) {

if (n.target == h && n.what == what

&& (object == null || n.obj == object)) {

Message nn = n.next;

n.recycleUnchecked();

p.next = nn;

continue;

}

}

p = n;

}

}

}

这个移除消息的方法，采用了两个while循环，第一个循环是从队头开始，移除符合条件的消息，第二个循环是从头部移除完连续的满足条件的消息之后，再从队列后面继续查询是否有满足条件的消息需要被移除

### postSyncBarrier

**public int** postSyncBarrier() {  
 **return** postSyncBarrier(SystemClock.*uptimeMillis*());  
}  
  
**private int** postSyncBarrier(**long** when) {  
 *// Enqueue a new sync barrier token.  
 // We don't need to wake the queue because the purpose of a barrier is to stall it.* **synchronized** (**this**) {  
 **final int** token = **mNextBarrierToken**++;  
 **final** Message msg = Message.*obtain*();  
 msg.markInUse();  
 msg.when = when;  
 msg.**arg1** = token;  
  
 Message prev = **null**;  
 Message p = **mMessages**;  
 **if** (when != 0) {  
 **while** (p != **null** && p.when <= when) {  
 prev = p;  
 p = p.next;  
 }  
 }  
 **if** (prev != **null**) { *// invariant: p == prev.next* msg.next = p;  
 prev.next = msg;  
 } **else** {  
 msg.next = p;  
 **mMessages** = msg;  
 }  
 **return** token;  
 }  
}

前面小节[4.3]已说明每一个普通Message必须有一个target，对于特殊的message是没有target，即**同步**barrier token。 这个消息的价值就是用于拦截同步消息，所以并不会唤醒Looper.

**public void** removeSyncBarrier(**int** token) {  
 *// Remove a sync barrier token from the queue.  
 // If the queue is no longer stalled by a barrier then wake it.* **synchronized** (**this**) {  
 Message prev = **null**;  
 Message p = **mMessages**;

//从消息队列找到 target为空,并且token相等的Message  
 **while** (p != **null** && (p.target != **null** || p.**arg1** != token)) {  
 prev = p;  
 p = p.next;  
 }  
 **if** (p == **null**) {  
 **throw new** IllegalStateException(**"The specified message queue synchronization "** + **" barrier token has not been posted or has already been removed."**);  
 }  
 **final boolean** needWake;  
 **if** (prev != **null**) {  
 prev.next = p.next;  
 needWake = **false**;  
 } **else** {  
 **mMessages** = p.next;  
 needWake = **mMessages** == **null** || **mMessages**.target != **null**;  
 }  
 p.recycleUnchecked();  
  
 *// If the loop is quitting then it is already awake.  
 // We can assume mPtr != 0 when mQuitting is false.* **if** (needWake && !**mQuitting**) {  
 *nativeWake*(**mPtr**);  
 }  
 }  
}

postSyncBarrier只对同步消息产生影响，对于异步消息没有任何差

### isPolling

当前的looper线程是否在polling工作来做，这个是个很好的用于检测loop是否存活的方法,看們狗 設計原理

**public boolean** isPolling() {  
 **synchronized** (**this**) {  
 **return** isPollingLocked();  
 }  
}

### IdleHandler

简而言之，就是在looper里面的message暂时处理完了，这个时候会回调这个接口，返回false，那么就会移除它，返回true就会在下次message处理完了的时候继续回调，

public static interface IdleHandler {

/\*\*

\* Called when the message queue has run out of messages and will now

\* wait for more. Return true to keep your idle handler active, false

\* to have it removed. This may be called if there are still messages

\* pending in the queue, but they are all scheduled to be dispatched

\* after the current time.

\*/

boolean queueIdle();

}

如果有这种需求，想要在某个activity绘制完成去做一些事情，那这个时机是什么时候呢？有同学可能觉得onResume()是一个合适的机会，不是可是这个onResume() 真的是各种绘制都已经完成才回调的吗？No, too naïve

<https://blog.csdn.net/tencent_bugly/article/details/78395717>

另外在ActivityThread也有IdleHandler的实现，时序图为

### quit()

void quit(boolean safe) {

// 当mQuitAllowed为false，表示不运行退出，强行调用quit()会抛出异常

if (!mQuitAllowed) {

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

}

synchronized (this) {

if (mQuitting) { //防止多次执行退出操作

return;

}

mQuitting = true;

if (safe) {

removeAllFutureMessagesLocked(); //移除尚未触发的所有消息

} else {

removeAllMessagesLocked(); //移除所有的消息

}

//mQuitting=false，那么认定为 mPtr != 0

nativeWake(mPtr);

}

}

消息退出的方式：

当safe =true时，只移除尚未触发的所有消息，对于正在触发的消息并不移除；

当safe =flase时，移除所有的消息

quitAllowed表示的是这个Looper运行退出，而对于false的情况则表示当前Looper不运行退出。具体可看MessageQueue

## Message

### 创建消息

每个消息用Message表示，Message主要包含以下内容：

| **数据类型** | **成员变量** | **解释** |
| --- | --- | --- |
| int | what | 消息类别 |
| long | when | 消息触发时间 |
| int | arg1 | 参数1 |
| int | arg2 | 参数2 |
| Object | obj | 消息内容 |
| Handler | target | 消息响应方 |
| Runnable | callback | 回调方法 |

创建消息的过程，就是填充消息的上述内容的一项或多项

### 消息池

在代码中，可能经常看到recycle()方法，咋一看，可能是在做虚拟机的gc()相关的工作，其实不然，这是用于把消息加入到消息池的作用。这样的好处是，当消息池不为空时，可以直接从消息池中获取Message对象，而不是直接创建，提高效率。

静态变量sPool的数据类型为Message，通过next成员变量，维护一个消息池；静态变量MAX\_POOL\_SIZE代表消息池的可用大小；消息池的默认大小为50。

消息池常用的操作方法是obtain()和recycle()。

#### obtain

从消息池中获取消息

public static Message obtain() {

synchronized (sPoolSync) {

if (sPool != null) {

Message m = sPool;

sPool = m.next;

m.next = null; //从sPool中取出一个Message对象，并消息链表断开

m.flags = 0; // 清除in-use flag

sPoolSize--; //消息池的可用大小进行减1操作

return m;

}

}

return new Message(); // 当消息池为空时，直接创建Message对象

}

obtain()，从消息池取Message，都是把消息池表头的Message取走，再把表头指向next;

#### recycle

把不再使用的消息加入消息池

public void recycle() {

if (isInUse()) { //判断消息是否正在使用

if (gCheckRecycle) { //Android 5.0以后的版本默认为true,之前的版本默认为false.

throw new IllegalStateException("This message cannot be recycled because it is still in use.");

}

return;

}

recycleUnchecked();

}

//对于不再使用的消息，加入到消息池

void recycleUnchecked() {

//将消息标示位置为IN\_USE，并清空消息所有的参数。

flags = FLAG\_IN\_USE;

what = 0;

arg1 = 0;

arg2 = 0;

obj = null;

replyTo = null;

sendingUid = -1;

when = 0;

target = null;

callback = null;

data = null;

synchronized (sPoolSync) {

if (sPoolSize < MAX\_POOL\_SIZE) { //当消息池没有满时，将Message对象加入消息池

next = sPool;

sPool = this;

sPoolSize++; //消息池的可用大小进行加1操作

}

}

}

recycle()，将Message加入到消息池的过程，都是把Message加到链表的表头

## 总结

最后用一张图，来表示整个消息机制handler\_java图解：



Handler通过sendMessage()发送Message到MessageQueue队列；

Looper通过loop()，不断提取出达到触发条件的Message，并将Message交给target来处理；

经过dispatchMessage()后，交回给Handler的handleMessage()来进行相应地处理。

将Message加入MessageQueue时，处往管道写入字符，可以会唤醒loop线程；如果MessageQueue中没有Message，并处于Idle状态，则会执行IdelHandler接口中的方法，往往用于做一些清理性地工作。

消息分发的优先级：

Message的回调方法：message.callback.run()，优先级最高；

Handler的回调方法：Handler.mCallback.handleMessage(msg)，优先级仅次于1；

Handler的默认方法：Handler.handleMessage(msg)，优先级最低。

### QA 线程和looper是一一对应关系的么？

1.线程和looper是一一对应关系，由sThreadLocal保证，只能通过prepare创建线程单例looper（构造函数为Looper）

private static void prepare(boolean quitAllowed) {

//每个线程只允许执行一次该方法，第二次执行时线程的TLS已有数据，则会抛出异常。

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

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

}

//创建Looper对象，并保存到当前线程的TLS区域

sThreadLocal.set(new Looper(quitAllowed));

}

private Looper(boolean quitAllowed) {

}

### new HandlerThread 得到的looper不是主线程的looper

2.HandlerThread 是一个子线程=>new HandlerThread 得到的looper不是主线程的looper

public void run() {

mTid = Process.myTid();

Looper.prepare();

synchronized (this) {

mLooper = Looper.myLooper();

notifyAll();

}

Process.setThreadPriority(mPriority);

onLooperPrepared();

Looper.loop();

mTid = -1;

}

### 3.主线程looper创建时间

android/app/ActivityThread.java

public static final void main(String[] args) {

Looper.prepareMainLooper();

ActivityThread thread = new ActivityThread();

thread.attach(false);

Looper.loop();

thread.detach();

}

# Native

相关源码

framework/base/core/java/andorid/os/MessageQueue.java

framework/base/core/jni/android\_os\_MessageQueue.cpp

framework/base/core/java/andorid/os/Looper.java （Java层）

system/core/libutils/Looper.cpp （Native层）

system/core/include/utils/Looper.h

system/core/libutils/RefBase.cpp

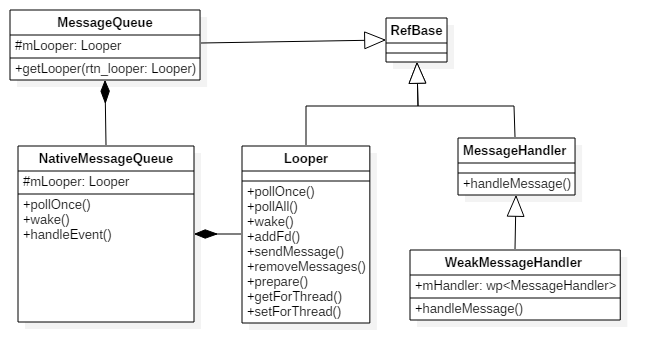
framework/base/native/android/looper.cpp （ALoop对象）

framework/native/include/android/looper.h

## 概述

在文章Android消息机制1-Handler(Java层)中讲解了Java层的消息处理机制，其中MessageQueue类里面涉及到多个native方法，除了MessageQueue的native方法，native层本身也有一套完整的消息机制，用于处理native的消息。在整个消息机制中，而MessageQueue是连接Java层和Native层的纽带，换言之，Java层可以向MessageQueue消息队列中添加消息，Native层也可以向MessageQueue消息队列中添加消息。

Native层的关系图



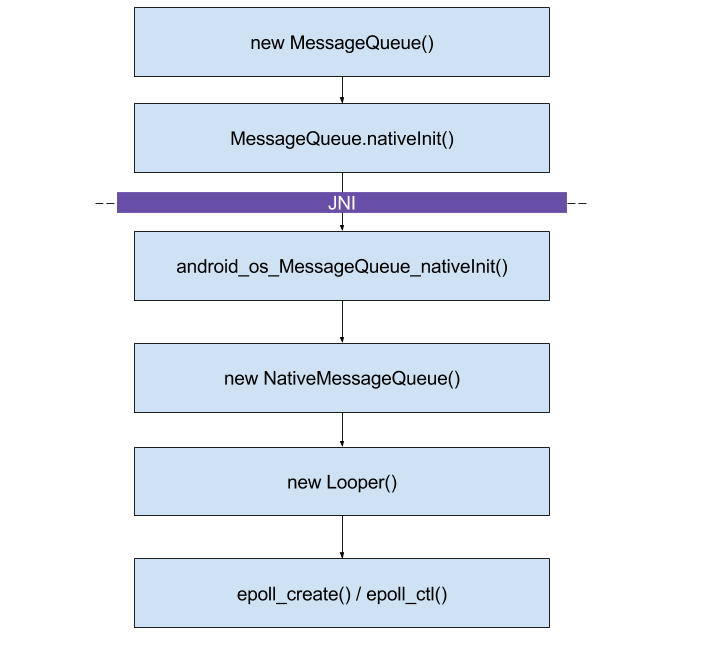
## MessageQueue

在MessageQueue中的native方法如下：

**private native static long** nativeInit();  
**private native static void** nativeDestroy(**long** ptr);  
**private native void** nativePollOnce(**long** ptr, **int** timeoutMillis); */\*non-static for callbacks\*/***private native static void** nativeWake(**long** ptr);  
**private native static boolean** nativeIsPolling(**long** ptr);  
**private native static void** nativeSetFileDescriptorEvents(**long** ptr, **int** fd, **int** events);

### nativeInit()

初始化过程的调用链如下



下面来进一步来看看调用链的过程：

**【1】 new MessageQueue()**

==> MessageQueue.java

MessageQueue(boolean quitAllowed) {

mQuitAllowed = quitAllowed;

mPtr = nativeInit(); //mPtr记录native消息队列的信息 【2】

}

**【2】android\_os\_MessageQueue\_nativeInit()**

==> android\_os\_MessageQueue.cpp

static jlong android\_os\_MessageQueue\_nativeInit(JNIEnv\* env, jclass clazz) {

//初始化native消息队列 【3】

NativeMessageQueue\* nativeMessageQueue = new NativeMessageQueue();

nativeMessageQueue->incStrong(env); //增加引用计数

return reinterpret\_cast<jlong>(nativeMessageQueue);

}

此处reinterpret\_cast是C++里的强制类型转换符

**【3】new NativeMessageQueue()**

==> android\_os\_MessageQueue.cpp

NativeMessageQueue::NativeMessageQueue()

: mPollEnv(NULL), mPollObj(NULL), mExceptionObj(NULL) {

mLooper = Looper::getForThread(); //获取TLS中的Looper对象

if (mLooper == NULL) {

mLooper = new Looper(false); //创建native层的Looper 【4】

Looper::setForThread(mLooper); //保存native层的Looper到TLS

}

}

* Looper::getForThread()，功能类比于Java层的Looper.myLooper();
* Looper::setForThread(mLooper)，功能类比于Java层的ThreadLocal.set();

此处Native层的Looper与Java层的Looper没有任何的关系，只是在Native层重实现了一套类似功能的逻辑。

**【4】new Looper()**

==> Looper.cpp

Looper::Looper(bool allowNonCallbacks) :

mAllowNonCallbacks(allowNonCallbacks), mSendingMessage(false),

mPolling(false), mEpollFd(-1), mEpollRebuildRequired(false),

mNextRequestSeq(0), mResponseIndex(0), mNextMessageUptime(LLONG\_MAX) {

mWakeEventFd = eventfd(0, EFD\_NONBLOCK); //构造唤醒事件的fd

AutoMutex \_l(mLock);

rebuildEpollLocked(); //重建Epoll事件【5】

}

**【5】epoll\_create/epoll\_ctl**

==> Looper.cpp

void Looper::rebuildEpollLocked() {

if (mEpollFd >= 0) {

close(mEpollFd); //关闭旧的epoll实例

}

mEpollFd = epoll\_create(EPOLL\_SIZE\_HINT); //创建新的epoll实例，并注册wake管道

struct epoll\_event eventItem;

memset(& eventItem, 0, sizeof(epoll\_event)); //把未使用的数据区域进行置0操作

eventItem.events = EPOLLIN; //可读事件

eventItem.data.fd = mWakeEventFd;

//将唤醒事件(mWakeEventFd)添加到epoll实例(mEpollFd)

int result = epoll\_ctl(mEpollFd, EPOLL\_CTL\_ADD, mWakeEventFd, & eventItem);

for (size\_t i = 0; i < mRequests.size(); i++) {

const Request& request = mRequests.valueAt(i);

struct epoll\_event eventItem;

request.initEventItem(&eventItem);

//将request队列的事件，分别添加到epoll实例

int epollResult = epoll\_ctl(mEpollFd, EPOLL\_CTL\_ADD, request.fd, & eventItem);

}

}

关于epoll的原理以及为什么选择epoll的方式，可查看文章[select/poll/epoll对比分析](http://gityuan.com/2015/12/06/linux_epoll/)。

*elect/poll/epoll都是IO多路复用机制，可以同时监控多个描述符，当某个描述符就绪(读或写就绪)，则立刻通知相应程序进行读或写操作。本质上select/poll/epoll都是同步I/O，即读写是阻塞的。*

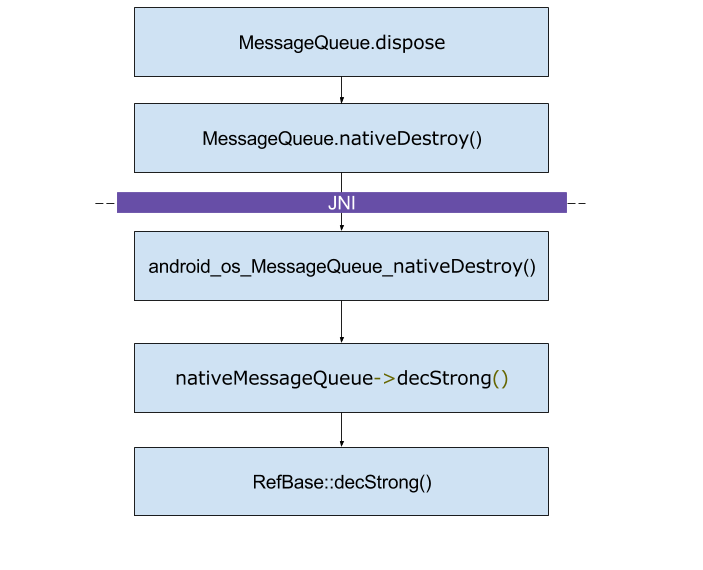
Epoll性能最高，回调函数，消息机制

Looper对象中的mWakeEventFd添加到epoll监控，以及mRequests也添加到epoll的监控范围内

### nativeDestroy()

ssd

清理回收的调用链如下：



下面来进一步来看看调用链的过程：

**【1】MessageQueue.dispose()**

==> MessageQueue.java

private void dispose() {

if (mPtr != 0) {

nativeDestroy(mPtr); 【2】

mPtr = 0;

}

}

**【2】android\_os\_MessageQueue\_nativeDestroy()**

==> android\_os\_MessageQueue.cpp

static void android\_os\_MessageQueue\_nativeDestroy(JNIEnv\* env, jclass clazz, jlong ptr) {

NativeMessageQueue\* nativeMessageQueue = reinterpret\_cast<NativeMessageQueue\*>(ptr);

nativeMessageQueue->decStrong(env); 【3】

}

nativeMessageQueue继承自RefBase类，所以decStrong最终调用的是RefBase.decStrong().

**【3】RefBase::decStrong()**

==> RefBase.cpp

void RefBase::decStrong(const void\* id) const

{

weakref\_impl\* const refs = mRefs;

refs->removeStrongRef(id); //移除强引用

const int32\_t c = android\_atomic\_dec(&refs->mStrong);

if (c == 1) {

refs->mBase->onLastStrongRef(id);

if ((refs->mFlags&OBJECT\_LIFETIME\_MASK) == OBJECT\_LIFETIME\_STRONG) {

delete this;

}

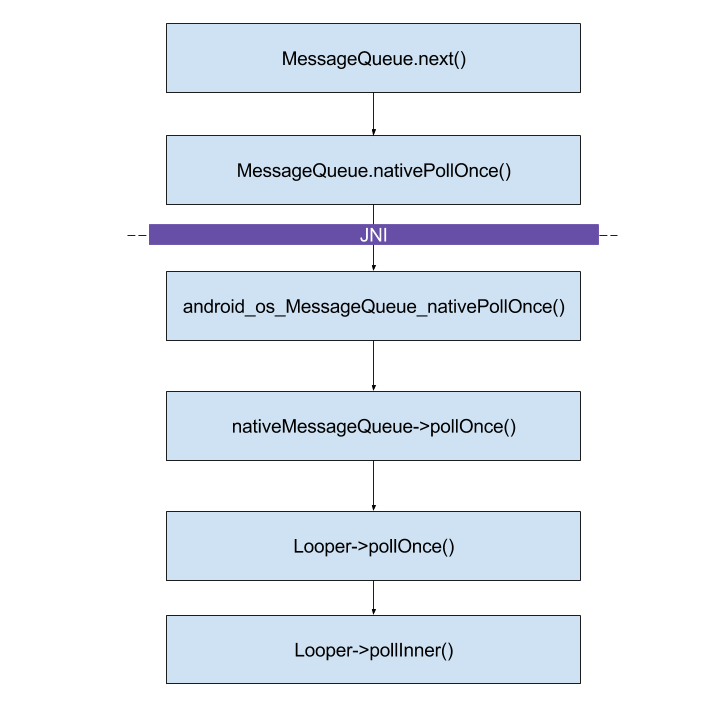
}

refs->decWeak(id); // 移除弱引用

}

### nativePollOnce()

nativePollOnce用于提取消息队列中的消息，提取消息的调用链，如下：



下面来进一步来看看调用链的过程：

**【1】MessageQueue.next()**

==> MessageQueue.java

Message next() {

final long ptr = mPtr;

if (ptr == 0) {

return null;

}

for (;;) {

...

nativePollOnce(ptr, nextPollTimeoutMillis); //阻塞操作 【2】

...

}

**【2】android\_os\_MessageQueue\_nativePollOnce()**

==> android\_os\_MessageQueue.cpp

static void android\_os\_MessageQueue\_nativePollOnce(JNIEnv\* env, jobject obj, jlong ptr, jint timeoutMillis) {

//将Java层传递下来的mPtr转换为nativeMessageQueue

NativeMessageQueue\* nativeMessageQueue = reinterpret\_cast<NativeMessageQueue\*>(ptr);

nativeMessageQueue->pollOnce(env, obj, timeoutMillis); 【3】

}

**【3】NativeMessageQueue::pollOnce()**

==> android\_os\_MessageQueue.cpp

void NativeMessageQueue::pollOnce(JNIEnv\* env, jobject pollObj, int timeoutMillis) {

mPollEnv = env;

mPollObj = pollObj;

mLooper->pollOnce(timeoutMillis); 【4】

mPollObj = NULL;

mPollEnv = NULL;

if (mExceptionObj) {

env->Throw(mExceptionObj);

env->DeleteLocalRef(mExceptionObj);

mExceptionObj = NULL;

}

}

**【4】Looper::pollOnce()**

==> Looper.h

inline int pollOnce(int timeoutMillis) {

return pollOnce(timeoutMillis, NULL, NULL, NULL); 【5】

}

**【5】 Looper::pollOnce()**

==> Looper.cpp

int Looper::pollOnce(int timeoutMillis, int\* outFd, int\* outEvents, void\*\* outData) {

int result = 0;

for (;;) {

// 先处理没有Callback方法的 Response事件

while (mResponseIndex < mResponses.size()) {

const Response& response = mResponses.itemAt(mResponseIndex++);

int ident = response.request.ident;

if (ident >= 0) { //ident大于0，则表示没有callback, 因为POLL\_CALLBACK = -2,

int fd = response.request.fd;

int events = response.events;

void\* data = response.request.data;

if (outFd != NULL) \*outFd = fd;

if (outEvents != NULL) \*outEvents = events;

if (outData != NULL) \*outData = data;

return ident;

}

}

if (result != 0) {

if (outFd != NULL) \*outFd = 0;

if (outEvents != NULL) \*outEvents = 0;

if (outData != NULL) \*outData = NULL;

return result;

}

// 再处理内部轮询

result = pollInner(timeoutMillis); 【6】

}

}

参数说明：

* timeoutMillis：超时时长
* outFd：发生事件的文件描述符
* outEvents：当前outFd上发生的事件，包含以下4类事件
  + EVENT\_INPUT 可读
  + EVENT\_OUTPUT 可写
  + EVENT\_ERROR 错误
  + EVENT\_HANGUP 中断
* outData：上下文数据

**【6】Looper::pollInner()**

==> Looper.cpp

int Looper::pollInner(int timeoutMillis) {

...

int result = POLL\_WAKE;

mResponses.clear();

mResponseIndex = 0;

mPolling = true; //即将处于idle状态

struct epoll\_event eventItems[EPOLL\_MAX\_EVENTS]; //fd最大个数为16

//等待事件发生或者超时，在nativeWake()方法，向管道写端写入字符，则该方法会返回；

int eventCount = epoll\_wait(mEpollFd, eventItems, EPOLL\_MAX\_EVENTS, timeoutMillis);

mPolling = false; //不再处于idle状态

mLock.lock(); //请求锁

if (mEpollRebuildRequired) {

mEpollRebuildRequired = false;

rebuildEpollLocked(); // epoll重建，直接跳转Done;

goto Done;

}

if (eventCount < 0) {

if (errno == EINTR) {

goto Done;

}

result = POLL\_ERROR; // epoll事件个数小于0，发生错误，直接跳转Done;

goto Done;

}

if (eventCount == 0) { //epoll事件个数等于0，发生超时，直接跳转Done;

result = POLL\_TIMEOUT;

goto Done;

}

//循环遍历，处理所有的事件

for (int i = 0; i < eventCount; i++) {

int fd = eventItems[i].data.fd;

uint32\_t epollEvents = eventItems[i].events;

if (fd == mWakeEventFd) {

if (epollEvents & EPOLLIN) {

awoken(); //已经唤醒了，则读取并清空管道数据【7】

}

} else {

ssize\_t requestIndex = mRequests.indexOfKey(fd);

if (requestIndex >= 0) {

int events = 0;

if (epollEvents & EPOLLIN) events |= EVENT\_INPUT;

if (epollEvents & EPOLLOUT) events |= EVENT\_OUTPUT;

if (epollEvents & EPOLLERR) events |= EVENT\_ERROR;

if (epollEvents & EPOLLHUP) events |= EVENT\_HANGUP;

//处理request，生成对应的reponse对象，push到响应数组

pushResponse(events, mRequests.valueAt(requestIndex));

}

}

}

Done: ;

//再处理Native的Message，调用相应回调方法

mNextMessageUptime = LLONG\_MAX;

while (mMessageEnvelopes.size() != 0) {

nsecs\_t now = systemTime(SYSTEM\_TIME\_MONOTONIC);

const MessageEnvelope& messageEnvelope = mMessageEnvelopes.itemAt(0);

if (messageEnvelope.uptime <= now) {

{

sp<MessageHandler> handler = messageEnvelope.handler;

Message message = messageEnvelope.message;

mMessageEnvelopes.removeAt(0);

mSendingMessage = true;

mLock.unlock(); //释放锁

handler->handleMessage(message); // 处理消息事件

}

mLock.lock(); //请求锁

mSendingMessage = false;

result = POLL\_CALLBACK; // 发生回调

} else {

mNextMessageUptime = messageEnvelope.uptime;

break;

}

}

mLock.unlock(); //释放锁

//处理带有Callback()方法的Response事件，执行Reponse相应的回调方法

for (size\_t i = 0; i < mResponses.size(); i++) {

Response& response = mResponses.editItemAt(i);

if (response.request.ident == POLL\_CALLBACK) {

int fd = response.request.fd;

int events = response.events;

void\* data = response.request.data;

// 处理请求的回调方法

int callbackResult = response.request.callback->handleEvent(fd, events, data);

if (callbackResult == 0) {

removeFd(fd, response.request.seq); //移除fd

}

response.request.callback.clear(); //清除reponse引用的回调方法

result = POLL\_CALLBACK; // 发生回调

}

}

return result;

}

pollOnce返回值说明：

* POLL\_WAKE： 表示由wake()触发，即pipe写端的write事件触发；
* POLL\_CALLBACK： 表示某个被监听fd被触发。
* POLL\_TIMEOUT： 表示等待超时；
* POLL\_ERROR：表示等待期间发生错误；

**【7】Looper::awoken()**

void Looper::awoken() {

uint64\_t counter;

//不断读取管道数据，目的就是为了清空管道内容

TEMP\_FAILURE\_RETRY(read(mWakeEventFd, &counter, sizeof(uint64\_t)));

}

**poll小结**

pollInner()方法的处理流程：

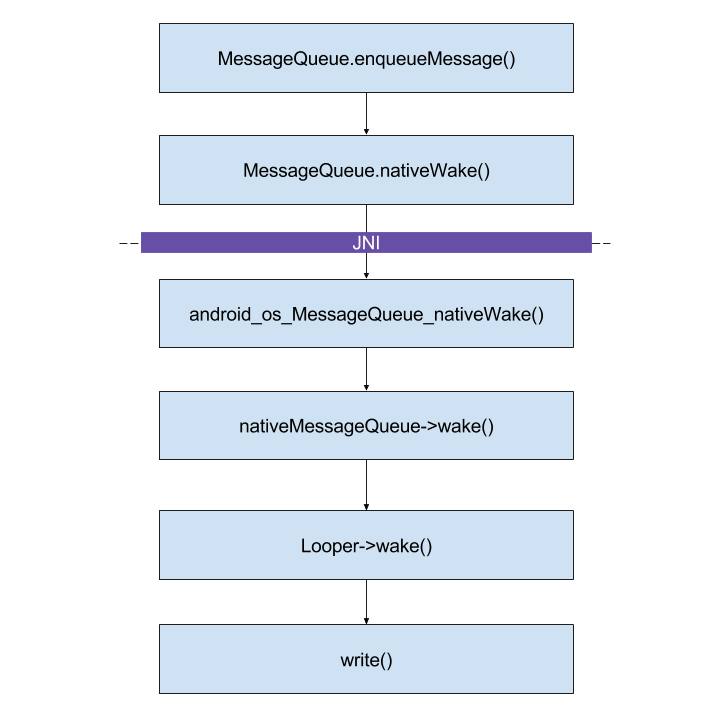
1. 先调用epoll\_wait()，这是阻塞方法，用于等待事件发生或者超时；
2. 对于epoll\_wait()返回，当且仅当以下3种情况出现：
   * POLL\_ERROR，发生错误，直接跳转到Done；
   * POLL\_TIMEOUT，发生超时，直接跳转到Done；
   * 检测到管道有事件发生，则再根据情况做相应处理：
     + 如果是管道读端产生事件，则直接读取管道的数据；
     + 如果是其他事件，则处理request，生成对应的reponse对象，push到reponse数组；
3. 进入Done标记位的代码段：
   * 先处理Native的Message，调用Native 的Handler来处理该Message;
   * 再处理Response数组，POLL\_CALLBACK类型的事件；

从上面的流程，可以发现对于Request先收集，一并放入reponse数组，而不是马上执行。真正在Done开始执行的时候，是先处理native Message，再处理Request，说明native Message的优先级高于Request请求的优先级。

另外pollOnce()方法中，先处理Response数组中不带Callback的事件，再调用了pollInner()方法。

### 2.4 nativeWake()

nativeWake用于唤醒功能，在添加消息到消息队列enqueueMessage(), 或者把消息从消息队列中全部移除quit()，再有需要时都会调用 nativeWake方法。包含唤醒过程的添加消息的调用链，如下：



下面来进一步来看看调用链的过程：

**【1】MessageQueue.enqueueMessage()**

==> MessageQueue.java

boolean enqueueMessage(Message msg, long when) {

... //将Message按时间顺序插入MessageQueue

if (needWake) {

nativeWake(mPtr); 【2】

}

}

往消息队列添加Message时，需要根据mBlocked情况来决定是否需要调用nativeWake。

**【2】android\_os\_MessageQueue\_nativeWake()**

==> android\_os\_MessageQueue.cpp

static void android\_os\_MessageQueue\_nativeWake(JNIEnv\* env, jclass clazz, jlong ptr) {

NativeMessageQueue\* nativeMessageQueue = reinterpret\_cast<NativeMessageQueue\*>(ptr);

nativeMessageQueue->wake(); 【3】

}

**【3】NativeMessageQueue::wake()**

==> android\_os\_MessageQueue.cpp

void NativeMessageQueue::wake() {

mLooper->wake(); 【4】

}

**【4】Looper::wake()**

==> Looper.cpp

void Looper::wake() {

uint64\_t inc = 1;

// 向管道mWakeEventFd写入字符1

ssize\_t nWrite = TEMP\_FAILURE\_RETRY(write(mWakeEventFd, &inc, sizeof(uint64\_t)));

if (nWrite != sizeof(uint64\_t)) {

if (errno != EAGAIN) {

ALOGW("Could not write wake signal, errno=%d", errno);

}

}

}

其中TEMP\_FAILURE\_RETRY 是一个宏定义， 当执行write失败后，会不断重复执行，直到执行成功为止。

### 2.5 sendMessage

在[Android消息机制1-Handler(Java层)](http://gityuan.com/2015/12/26/handler-message-framework/)文中，讲述了Java层如何向MessageQueue类中添加消息，那么接下来讲讲Native层如何向MessageQueue发送消息。

**【1】sendMessage**

void Looper::sendMessage(const sp<MessageHandler>& handler, const Message& message) {

nsecs\_t now = systemTime(SYSTEM\_TIME\_MONOTONIC);

sendMessageAtTime(now, handler, message);

}

**【2】sendMessageDelayed**

void Looper::sendMessageDelayed(nsecs\_t uptimeDelay, const sp<MessageHandler>& handler,

const Message& message) {

nsecs\_t now = systemTime(SYSTEM\_TIME\_MONOTONIC);

sendMessageAtTime(now + uptimeDelay, handler, message);

}

sendMessage(),sendMessageDelayed() 都是调用sendMessageAtTime()来完成消息插入。

**【3】sendMessageAtTime**

void Looper::sendMessageAtTime(nsecs\_t uptime, const sp<MessageHandler>& handler,

const Message& message) {

size\_t i = 0;

{ //请求锁

AutoMutex \_l(mLock);

size\_t messageCount = mMessageEnvelopes.size();

//找到message应该插入的位置i

while (i < messageCount && uptime >= mMessageEnvelopes.itemAt(i).uptime) {

i += 1;

}

MessageEnvelope messageEnvelope(uptime, handler, message);

mMessageEnvelopes.insertAt(messageEnvelope, i, 1);

//如果当前正在发送消息，那么不再调用wake()，直接返回。

if (mSendingMessage) {

return;

}

} //释放锁

//当把消息加入到消息队列的头部时，需要唤醒poll循环。

if (i == 0) {

wake();

}

}

### 2.6 小结

本节介绍MessageQueue的native()方法，经过层层调用：

* nativeInit()方法：
  + 创建了NativeMessageQueue对象，增加其引用计数，并将NativeMessageQueue指针mPtr保存在Java层的MessageQueue
  + 创建了Native Looper对象
  + 调用epoll的epoll\_create()/epoll\_ctl()来完成对mWakeEventFd和mRequests的可读事件监听
* nativeDestroy()方法
  + 调用RefBase::decStrong()来减少对象的引用计数
  + 当引用计数为0时，则删除NativeMessageQueue对象
* nativePollOnce()方法
  + 调用Looper::pollOnce()来完成，空闲时停留在epoll\_wait()方法，用于等待事件发生火灾超时
* nativeWake()方法
  + 调用Looper::wake()来完成，向管道mWakeEventfd写入字符；

## 三、Native结构体和类

Looper.h/ Looper.cpp文件中，定义了Message结构体，消息处理类，回调类，Looper类。

### 3.1 Message结构体

struct Message {

Message() : what(0) { }

Message(int what) : what(what) { }

int what; // 消息类型

};

### 3.2 消息处理类

MessageHandler类

class MessageHandler : public virtual RefBase {

protected:

virtual ~MessageHandler() { }

public:

virtual void handleMessage(const Message& message) = 0;

};

WeakMessageHandler类，继承于MessageHandler类

class WeakMessageHandler : public MessageHandler {

protected:

virtual ~WeakMessageHandler();

public:

WeakMessageHandler(const wp<MessageHandler>& handler);

virtual void handleMessage(const Message& message);

private:

wp<MessageHandler> mHandler;

};

void WeakMessageHandler::handleMessage(const Message& message) {

sp<MessageHandler> handler = mHandler.promote();

if (handler != NULL) {

handler->handleMessage(message); //调用MessageHandler类的处理方法()

}

}

### 3.3 回调类

LooperCallback类

class LooperCallback : public virtual RefBase {

protected:

virtual ~LooperCallback() { }

public:

//用于处理指定的文件描述符的poll事件

virtual int handleEvent(int fd, int events, void\* data) = 0;

};

SimpleLooperCallback类， 继承于LooperCallback类

class SimpleLooperCallback : public LooperCallback {

protected:

virtual ~SimpleLooperCallback();

public:

SimpleLooperCallback(Looper\_callbackFunc callback);

virtual int handleEvent(int fd, int events, void\* data);

private:

Looper\_callbackFunc mCallback;

};

int SimpleLooperCallback::handleEvent(int fd, int events, void\* data) {

return mCallback(fd, events, data); //调用回调方法

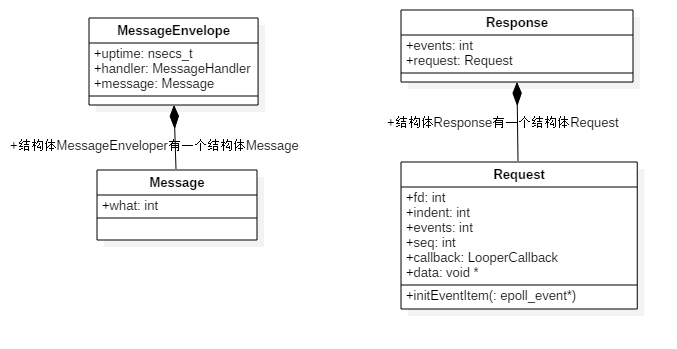
}

### 3.4 Looper类

static const int EPOLL\_SIZE\_HINT = 8; //每个epoll实例默认的文件描述符个数

static const int EPOLL\_MAX\_EVENTS = 16; //轮询事件的文件描述符的个数上限

其中Looper类的内部定义了Request，Response，MessageEnvelope这3个结构体，关系图如下：



代码如下：

struct Request { //请求结构体

int fd;

int ident;

int events;

int seq;

sp<LooperCallback> callback;

void\* data;

void initEventItem(struct epoll\_event\* eventItem) const;

};

struct Response { //响应结构体

int events;

Request request;

};

struct MessageEnvelope { //信封结构体

MessageEnvelope() : uptime(0) { }

MessageEnvelope(nsecs\_t uptime, const sp<MessageHandler> handler, const Message& message) : uptime(uptime), handler(handler), message(message) {

}

nsecs\_t uptime;

sp<MessageHandler> handler;

Message message;

};

MessageEnvelope正如其名字，信封。MessageEnvelope里面记录着收信人(handler)，发信时间(uptime)，信件内容(message)

#### addFd

Navite Looper 除了提供message机制之外，还提供了监听文件描述符的方式。

通过addFd()接口加入需要被监听的文件描述符。

int addFd(int fd, int ident, int events, Looper\_callbackFunc callback, void\* data);

int addFd(int fd, int ident, int events, const sp<LooperCallback>& callback, void\* data);

其中：

- fd：为所需要监听的文件描述符。

- ident：表示为当前发生事件的标识符，必须>=0，或者为POLL\_CALLBACK(-2)如果指定了callback。

- events：表示为要监听的文件类型，默认是EVENT\_INPUT。

- callback：当有事件发生时，会回调该callback函数。

- data： 用于传递指针，c就可以调用java的代码，jni都是static方法，如果一个class有静态方法，互相调用怎么办呢

两种使用方式：

指定callback来处理事件 : 当该文件描述符上有事件到来时，该callback会被执行，然后从该fd读取数据。这个时候ident是被忽略的。

通过指定的ident来处理事件: 当该文件描述符有数据到来时，pollOnce() 会返回一个ident，调用者会判断该ident是否等于自己需要处理的事件ident，如果是的话，则开始处理事件。

1. 通过指定的ident来处理事件:

使用场景：

在Android 4.0.4版本SensorEventQueue.cpp 有使用。

SensorEventQueue.cpp : getLooper() : 将所需奥监听的文件描述符增加到looper中。

sp<Looper> SensorEventQueue::getLooper() const

{

Mutex::Autolock \_l(mLock);

if (mLooper == 0) {

mLooper = new Looper(true);

// 将要监听的文件描述符作为事件的标识符穿进去，并且callback为null

mLooper->addFd(getFd(), getFd(), ALOOPER\_EVENT\_INPUT, NULL, NULL);

}

return mLooper;

}

1

2

3

4

5

6

7

8

9

10

SensorEventQueue.cpp : waitForEvent(): 通过looper的pollOnce轮询监听事件，然后返回事件的标识符，如果事件的标识符等于getFd()所监听的文件描述符时，则返回NO\_ERROR。

status\_t SensorEventQueue::waitForEvent() const

{

const int fd = getFd();

sp<Looper> looper(getLooper());

int events;

int32\_t result;

do {

result = looper->pollOnce(-1, NULL, &events, NULL);

if (result == ALOOPER\_POLL\_ERROR) {

ALOGE("SensorEventQueue::waitForEvent error (errno=%d)", errno);

result = -EPIPE; // unknown error, so we make up one

break;

}

if (events & ALOOPER\_EVENT\_HANGUP) {

// the other-side has died

ALOGE("SensorEventQueue::waitForEvent error HANGUP");

result = -EPIPE; // unknown error, so we make up one

break;

}

} while (result != fd);

return (result == fd) ? status\_t(NO\_ERROR) : result;

}

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

android/4.0.4/frameworks/base/core/jni/android\_hardware\_SensorManager.cpp : sensor\_data\_poll()

早期版本采用如下的消息驱动机制，在5.0版本已经没有这样使用了。

可以看到：当waitForEvent() 返回NO\_ERROR，就会去读取数据。

static jint

sensors\_data\_poll(JNIEnv \*env, jclass clazz, jint nativeQueue,

jfloatArray values, jintArray status, jlongArray timestamp)

{

sp<SensorEventQueue> queue(reinterpret\_cast<SensorEventQueue \*>(nativeQueue));

if (queue == 0) return -1;

status\_t res;

ASensorEvent event;

res = queue->read(&event, 1);

if (res == -EAGAIN) {

res = queue->waitForEvent();

if (res != NO\_ERROR)

return -1;

res = queue->read(&event, 1);

}

if (res < 0)

return -1;

jint accuracy = event.vector.status;

env->SetFloatArrayRegion(values, 0, 3, event.vector.v);

env->SetIntArrayRegion(status, 0, 1, &accuracy);

env->SetLongArrayRegion(timestamp, 0, 1, &event.timestamp);

return event.sensor;

}

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

2. 指定callback来处理事件：

callback 必须是继承子LooperCallback,或者是如下形式的回调函数。

typedef int (\*Looper\_callbackFunc)(int fd, int events, void\* data);

实际上这种形式也是封装成了SimpleLooperCallback对象，该对象同样是继承自LooperCallback。

int Looper::addFd(int fd, int ident, int events, Looper\_callbackFunc callback, void\* data) {

return addFd(fd, ident, events, callback ? new SimpleLooperCallback(callback) : NULL, data);

}

1

2

3

SimpleLooperCallback定义如下，在Looper.h 文件

/\*\*

\* Wraps a Looper\_callbackFunc function pointer.

\*/

class SimpleLooperCallback : public LooperCallback {

protected:

virtual ~SimpleLooperCallback();

public:

SimpleLooperCallback(Looper\_callbackFunc callback);

virtual int handleEvent(int fd, int events, void\* data);

private:

Looper\_callbackFunc mCallback;

};

使用场景1:

InputDispatcher.cpp : registerInputChannel()

mLooper->addFd(fd, 0, ALOOPER\_EVENT\_INPUT, handleReceiveCallback, this);

handleReceiveCallback() 的实现如下：

int InputDispatcher::handleReceiveCallback(int fd, int events, void\* data) {

InputDispatcher\* d = static\_cast<InputDispatcher\*>(data);

...

使用场景2:

android\_view\_InputEventReceiver.cpp : setFdEvents() 将fd加入looper 轮询监听，另外还需要实现一个handleEvent() 函数。

void NativeInputEventReceiver::setFdEvents(int events) {

if (mFdEvents != events) {

mFdEvents = events;

int fd = mInputConsumer.getChannel()->getFd();

if (events) {

mMessageQueue->getLooper()->addFd(fd, 0, events, this, NULL);

} else {

mMessageQueue->getLooper()->removeFd(fd);

}

}

}

int NativeInputEventReceiver::handleEvent(int receiveFd, int events, void\* data) {

if (events & (ALOOPER\_EVENT\_ERROR | ALOOPER\_EVENT\_HANGUP)) {

DisplayEventReceiver myDisplayEvent;  
sp<Looper> loop = **new** Looper(**false**);  
loop->addFd(myDisplayEvent.getFd(), 0, ALOOPER\_EVENT\_INPUT, receiver,  
 &myDisplayEvent);  
**do** {int32\_t ret = loop->pollOnce(-1);  
} **while** (1);

### 3.5 ALooper类

ALooper类定义在通过looper.cpp/looper.h（注意此文件是小写字母开头，与Looper.cpp不同，具体源码路径，可通过查看文章最开头的 相关源码）

static inline Looper\* ALooper\_to\_Looper(ALooper\* alooper) {

return reinterpret\_cast<Looper\*>(alooper);

}

static inline ALooper\* Looper\_to\_ALooper(Looper\* looper) {

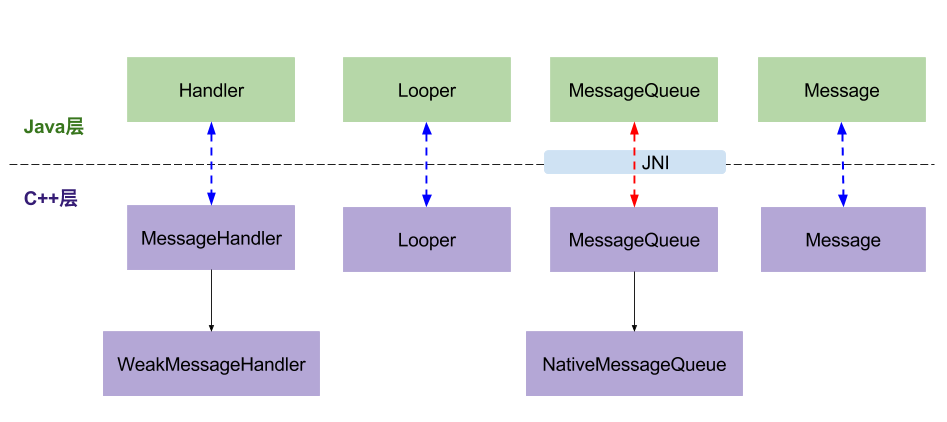
return reinterpret\_cast<ALooper\*>(looper);

}

ALooper类 与前面介绍的Looper类，更多的操作是通过ALooper\_to\_Looper(), Looper\_to\_ALooper()这两个方法转换完成的，也就是说ALooper类中定义的所有方法，都是通过转换为Looper类，再执行Looper中的方法。

## 总结

MessageQueue通过mPtr变量保存NativeMessageQueue对象，从而使得MessageQueue成为Java层和Native层的枢纽，既能处理上层消息，也能处理native层消息；下面列举Java层与Native层的对应图



图解：

* 红色虚线关系：Java层和Native层的MessageQueue通过JNI建立关联，彼此之间能相互调用，搞明白这个互调关系，也就搞明白了Java如何调用C++代码，C++代码又是如何调用Java代码。
* 蓝色虚线关系：Handler/Looper/Message这三大类Java层与Native层并没有任何的真正关联，只是分别在Java层和Native层的handler消息模型中具有相似的功能。都是彼此独立的，各自实现相应的逻辑。
* WeakMessageHandler继承于MessageHandler类，NativeMessageQueue继承于MessageQueue类

另外，消息处理流程是先处理Native Message，再处理Native Request，最后处理Java Message。理解了该流程，也就明白有时上层消息很少，但响应时间却较长的真正原因。

# REF

[Android消息机制1-Handler(Java层)](http://gityuan.com/2015/12/26/handler-message-framework/)

<http://gityuan.com/2015/12/26/handler-message-framework/>

http://gityuan.com/2015/12/27/handler-message-native/

# 实战分析

## 子线程不行绘制是如何设计的

在Android系统中在onresume之后，主线程可以进行UI绘制(通过ViewRootImpl.checkThread()来保证)，子线程可以么？SurfaceView可以在子线程中绘制么？

ViewRootImpl.checkThread() {

if (mThread != Thread.currentThread()) {

throw new CalledFromWrongThreadException(

"Only the original thread that created a view hierarchy can touch its views.");

}

}

在Android系统中在onresume之后，主线程可以进行UI绘制(通过ViewRootImpl.checkThread()来保证)。

子线程也可以（demo 见 https://blog.csdn.net/innost/article/details/47660193），只要view没有被添加到viewrootimpl的层级中，ondraw中canvas从Surface来

ViewRootImpl.checkThread() {

if (mThread != Thread.currentThread()) {

throw new CalledFromWrongThreadException(

"Only the original thread that created a view hierarchy can touch its views.");

}

}

## Android中为什么主线程不会因为Looper.loop()里的死循环卡死？

不卡死是因为按时水蕨了

这里涉及线程，先说说说进程/线程，**进程：**每个app运行时前首先创建一个进程，该进程是由Zygote fork出来的，用于承载App上运行的各种Activity/Service等组件。进程对于上层应用来说是完全透明的，这也是google有意为之，让App程序都是运行在Android Runtime。大多数情况一个App就运行在一个进程中，除非在AndroidManifest.xml中配置Android:process属性，或通过native代码fork进程。

**线程：**线程对应用来说非常常见，比如每次new Thread().start都会创建一个新的线程。该线程与App所在进程之间资源共享，从Linux角度来说进程与线程除了是否共享资源外，并没有本质的区别，都是一个task\_struct结构体**，在CPU看来进程或线程无非就是一段可执行的代码，CPU采用CFS调度算法，保证每个task都尽可能公平的享有CPU时间片**。

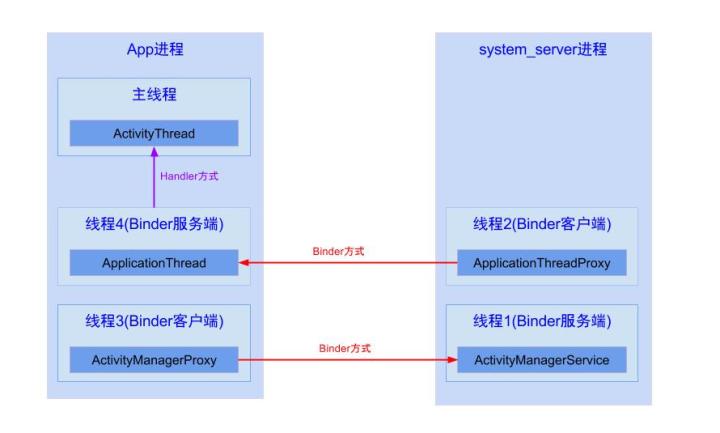
有了这么准备，再说说死循环问题：

对于线程既然是一段可执行的代码，当可执行代码执行完成后，线程生命周期便该终止了，线程退出。而对于主线程，我们是绝不希望会被运行一段时间，自己就退出，那么如何保证能一直存活呢？**简单做法就是可执行代码是能一直执行下去的，死循环便能保证不会被退出，**例如，binder线程也是采用死循环的方法，通过循环方式不同与Binder驱动进行读写操作，当然并非简单地死循环，无消息时会休眠。但这里可能又引发了另一个问题，既然是死循环又如何去处理其他事务呢？通过创建新线程的方式。

真正会卡死主线程的操作是在回调方法onCreate/onStart/onResume等操作时间过长，会导致掉帧，甚至发生ANR，looper.loop本身不会导致应用卡死。

### 进程与线程间通信的角度，通过一张图加深大家对App运行过程

Cf



作者：Gityuan  
链接：https://www.zhihu.com/question/34652589/answer/90344494  
来源：知乎  
著作权归作者所有。商业转载请联系作者获得授权，非商业转载请注明出处。

**system\_server进程是系统进程**，java framework框架的核心载体，里面运行了大量的系统服务，比如这里提供ApplicationThreadProxy（简称ATP），ActivityManagerService（简称AMS），这个两个服务都运行在system\_server进程的不同线程中，由于ATP和AMS都是基于IBinder接口，都是binder线程，binder线程的创建与销毁都是由binder驱动来决定的。

**App进程则是我们常说的应用程序**，主线程主要负责Activity/Service等组件的生命周期以及UI相关操作都运行在这个线程； 另外，每个App进程中至少会有两个binder线程 ApplicationThread(简称AT)和ActivityManagerProxy（简称AMP），除了图中画的线程，其中还有很多线程，比如signal catcher线程等，这里就不一一列举。

Binder用于不同进程之间通信，由一个进程的Binder客户端向另一个进程的服务端发送事务，比如图中线程2向线程4发送事务；而handler用于同一个进程中不同线程的通信，比如图中线程4向主线程发送消息。

**结合图说说Activity生命周期，比如暂停Activity，流程如下：**

1. 线程1的AMS中调用线程2的ATP；（由于同一个进程的线程间资源共享，可以相互直接调用，但需要注意多线程并发问题）
2. 线程2通过binder传输到App进程的线程4；
3. 线程4通过handler消息机制，将暂停Activity的消息发送给主线程；
4. 主线程在looper.loop()中循环遍历消息，当收到暂停Activity的消息时，便将消息分发给ActivityThread.H.handleMessage()方法，再经过方法的调用，最后便会调用到Activity.onPause()，当onPause()处理完后，继续循环loop下去。

## AsyncChannel

AsyncChannel 用于两个Handler 之间的通信，具体的通信方式为源Handler 通过sendMessage 向目标Handler 发送消息，而目标Handler 通过replyToMessage 回复源Handler 处理结果；这两个Handler 可位于同一个进程，也可分属于两个不同的进程

# Android的消息机制

本章所要讲述的内容是Android的消息机制。提到消息机制读者应该都不陌生，在日常开发中不可避免地要涉及这方面的内容。从开发的角度来说，Handler是Android消息机制的上层接口，这使得在开发过程中只需要和Handler交互即可。Handler的使用过程很简单，通过它可以轻松地将一个任务切换到Handler所在的线程中去执行。很多人认为Handler的作用是更新UI，这的确没错，但是更新UI仅仅是Handler的一个特殊的使用场景.具体来说是这样的：有时候需要在子线程中进行耗时的I/O操作，可能是读取文件或者访问网络等，当耗时操作完成以后可能需要在UI上做一些改变，由于Android开发规范的限制，我们并不能在子线程中访问UI控件，否则就会触发程序异常，**这个时候通过Handler就可以将更新UI的操作切换到主线程中执行**。因此，本质上来说，Handler并不是专门用于更新UI的，它只是常被开发者用来更新UI。

Android的消息机制主要是指Handler的运行机制，Handler的运行需要底层的MessageQueue和Looper的支撑。MessageQueue的中文翻译是消息队列，顾名思义，它的内部存储了一组消息，以队列的形式对外提供插入和删除的工作。虽然叫消息队列，但是它的内部存储结构并不是真正的队列，而是采用**单链表**的数据结构来存储消息列表。Looper的中文翻译为循环，在这里可以理解为消息循环。由于MessageQueue只是一个消息的存储单元，它不能去处理消息，而Looper就填补了这个功能，Looper会以无限循环的形式去查找是否有新消息，如果有的话就处理消息，否则就一直等待着。Looper中还有一个特殊的概念，那就是ThreadLocal，ThreadLocal并不是线程，它的作用是可以在每个线程中存储数据。我们知道，Handler创建的时候会采用当前线程的Looper来构造消息循环系统，那么Handler内部如何获取到当前线程的Looper呢？这就要使用ThreadLocal了，ThreadLocal可以在不同的线程中互不干扰地存储并提供数据，通过ThreadLocal可以轻松获取每个线程的Looper。当然需要注意的是，线程是默认没有Looper的，如果需要使用Handler就必须为线程创建Looper。我们经常提到的主线程，也叫UI线程，它就是ActivityThread，ActivityThread被创建时就会初始化Looper，这也是在主线程中默认可以使用Handler的原因。

## Android的消息机制概述

前面提到，Android的消息机制主要是指Handler的运行机制以及Handler所附带的MessageQueue和Looper的工作过程，这三者实际上是一个整体，只不过我们在开发过程中比较多地接触到Handler而已。**Handler的主要作用是将一个任务切换到某个指定的线程中去执行**，**那么Android为什么要提供这个功能呢？或者说Android为什么需要提供在某个具体的线程中执行任务这种功能呢？**这是因为Android规定访问UI只能在主线程中进行，如果在子线程中访问UI，那么程序就会抛出异常。 ViewRootImpl对UI操作做了验证，这个验证工作是由ViewRootImpl的checkThread方法来完成的，如下所示。



针对checkThread方法中抛出的异常信息，相信读者在开发中都曾经遇到过。由于这一点的限制，导致必须在主线程中访问UI，但是Android又建议不要在主线程中进行耗时操作，否则会导致程序无法响应即ANR。考虑一种情况，假如我们需要从服务端拉取一些信息并将其显示在UI上，这个时候必须在子线程中进行拉取工作，拉取完毕后又不能在子线程中直接访问UI，如果没有Handler，那么我们的确没有办法将访问UI的工作切换到主线程中去执行。因此，**系统之所以提供Handler，主要原因就是为了解决在子线程中无法访问UI的矛盾**。

这里再延伸一点，**系统为什么不允许在子线程中访问UI呢？这是因为Android的UI控件不是线程安全的**，如果在多线程中并发访问可能会导致UI控件处于不可预期的状态，那为什么系统不对UI控件的访问加上锁机制呢？缺点有两个：首先加上锁机制会让UI访问的逻辑变得复杂；其次锁机制会降低UI访问的效率，因为锁机制会阻塞某些线程的执行。鉴于这两个缺点，最简单且高效的方法就是采用**单线程模型来处理UI操作**，对于开发者来说也不是很麻烦，只是需要通过Handler切换一下UI访问的执行线程即可。

Handler的使用方法这里就不做介绍了，这里描述一下Handler的工作原理。Handler创建时会采用当前线程的Looper来构建内部的消息循环系统，如果当前线程没有Looper，那么就会报错，如下所示。

代码如下：

|  |
| --- |
| **new** Thread(**new** Runnable() {  @Override  **public void** run() {*//Can't create handler inside thread that has not called Looper.prepare()* Toast.*makeText*(KApplication.*getIns*(),**"toast in sub Thread"**,Toast.***LENGTH\_LONG***).show();  } }).start(); |

报错如下：



如果解决上述问题呢？其实很简单，只需要为当前线程创建Looper即可，或者在一个有Looper的线程中创建Handler也行，具体会在10.2.3节中进行介绍。

或者直接用主线程的Looper



Handler创建完毕后，这个时候其内部的Looper以及MessageQueue就可以和Handler一起协同工作了，然后通过Handler的post方法将一个Runnable投递到Handler内部的Looper中去处理，也可以通过Handler的send方法发送一个消息，这个消息同样会在Looper中去处理。其实post方法最终也是通过**send**方法来完成的，接下来主要来看uop下send方法的工作过程。当Handler的send方法被调用时，它会调用MessageQueue的enqueueMessage方法将这个消息放入消息队列中，然后Looper发现有新消息到来时，就会处理这个消息，最终消息中的Runnable或者Handler的 handleMessage方法就会被调用。注意Looper是运行在创建Handler所在的线程中的，这样一来Handler中的业务逻辑就被切换到创建Handler所在的线程中去执行了，这个过程可以用图10-1来表示。



图10-1  Handler的工作过程

## Android的消息机制分析

在10.1节中对Android的消息机制已经做了一个概括性的描述，通过图10-1也能够比较好地理解Handler的工作过程。本节将对Android消息机制的实现原理做一个全面的分析。由于Android的消息机制实际上就是Handler的运行机制，因此本节主要围绕着Handler的工作过程来分析Android的消息机制，主要包括Handler、MessageQueue和Looper。同时为了更好地理解Looper的工作原理，本节还会介绍ThreadLocal，通过本节的介绍可以让读者对Android的消息机制有一个深入的理解。

不做展开。

### 消息队列的工作原理

消息队列在Android中指的是MessageQueue，MessageQueue主要包含两个操作：插入和读取。读取操作本身会伴随着删除操作，插入和读取对应的方法分别为enqueueMessage和next，其中enqueueMessage的作用是往消息队列中插入一条消息，而next的作用是从消息队列中取出一条消息并将其从消息队列中移除。尽管MessageQueue叫消息队列，但是它的内部实现并不是用的队列，实际上它是通过一个单链表的数据结构为维护消息列表，单链表在插入和删除上比较有优势。下面主要看一下它的enqueueMessage和next方法的实现，enqueueMessage的源码的伪代码如下所示。

|  |
| --- |
| boolean enqueueMessage(Message msg, long when) {  msg.when = when;  Message p = mMessages;  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;  }  } |

就是一个按照时间排序的单链表的插入操作。下面看一下next方法的实现，next的主要逻辑如下所示。

|  |
| --- |
| **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()); } |

可以发现next方法是一个无限循环的方法，如果消息队列中没有消息，那么next方法会一直阻塞在这里。当有新消息到来时，next方法会返回这条消息并将其从单链表中移除。

### Looper的工作原理

在10.2.2节中介绍了消息队列的主要实现，本节将分析Looper的具体实现。Looper在Android的消息机制中扮演着消息循环的角色，具体来说就是它会不停地从MessageQueue中去查看是否有新消息，如果有新消息就会立刻处理，否则就一直阻塞在那里。首先看一下它的构造方法，在构造方法中它会创建一个MessageQueue即消息队列，然后将当前线程的对象保存起来，如下所示。



我们知道，Handler的工作需要Looper，没有Looper的线程就会报错，那么如何为一个线程创建Looper呢？其实很简单，通过Looper.prepare()即可为当前线程创建一个Looper，接着通过Looper.loop()来开启消息循环，如下所示。



Looper除了prepare方法外，还提供了prepareMainLooper方法，这个方法主要是给主线程也就是ActivityThread创建Looper所使用的，其本质也是通过prepare方法来实现的。由于主线程的Looper比较特殊，所以Looper提供了一个getMainLooper方法，通过它可以在任何地方获取到主线程的Looper。Looper也是可以退出的，Looper提供了quit和quitSafely来退出一个Looper，二者的区别是：quit会直接退出Looper，而quitSafely只是设定一个退出标记，然后把消息队列中的已有消息处理完毕后才安全地退出。Looper退出后，通过Handler发送的消息会失败，这个时候Handler的send方法会返回false。在子线程中，如果手动为其创建了Looper，那么在所有的事情完成以后应该调用quit方法来终止消息循环，否则这个子线程就会一直处于等待的状态，而如果退出Looper以后，这个线程就会立刻终止，因此建议不需要的时候终止Looper。

Looper最重要的一个方法是loop方法，只有调用了loop后，消息循环系统才会真正地起作用，它的实现如下所示。

|  |
| --- |
| public static void loop() {  for (;;) {  Message msg = queue.next(); // might block  if (msg == null) {  // No message indicates that the message queue is quitting.  return;  }  }  } |

Looper的loop方法的工作过程也比较好理解，loop方法是一个死循环，唯一跳出循环的方式是MessageQueue的next方法返回了null。当Looper的quit方法被调用时，Looper就会调用MessageQueue的quit或者quitSafely方法来通知消息队列退出，当消息队列被标记为退出状态时，它的next方法就会返回null。也就是说，Looper必须退出，否则loop方法就会无限循环下去。loop方法会调用MessageQueue的next方法来获取新消息，**而next是一个阻塞操作，当没有消息时，next方法会一直阻塞在那里**，这也导致loop方法一直阻塞在那里。如果MessageQueue的next方法返回了新消息，Looper就会处理这条消息：msg.target.dispatchMessage(msg)，这里的msg.target是发送这条消息的Handler对象，这样Handler发送的消息最终又交给它的dispatchMessage方法来处理了。但是这里不同的是，Handler的dispatchMessage方法是在创建Handler时所使用的Looper中执行的，这样就成功地将代码逻辑切换到指定的线程中去执行了。

## 主线程的消息循环

Android的主线程就是ActivityThread，主线程的入口方法为main，在main方法中系统会通过Looper.prepareMainLooper()来创建主线程的Looper以及MessageQueue，并通过Looper.loop()来开启主线程的消息循环，这个过程如下所示。



主线程的消息循环开始了以后，ActivityThread还需要一个Handler来和消息队列进行交互，这个Handler就是ActivityThread.H，它内部定义了一组消息类型，主要包含了四大组件的启动和停止等过程，如下所示。

|  |
| --- |
| **private class** H **extends** Handler {  **public static final int** LAUNCH\_ACTIVITY = 100;  **public static final int** PAUSE\_ACTIVITY = 101;  **public static final int** PAUSE\_ACTIVITY\_FINISHING = 102;  **public static final int** STOP\_ACTIVITY\_SHOW = 103;  **public static final int** STOP\_ACTIVITY\_HIDE = 104;  **public static final int** SHOW\_WINDOW = 105;  **public static final int** HIDE\_WINDOW = 106;  **public static final int** RESUME\_ACTIVITY = 107;  **public static final int** SEND\_RESULT = 108;  **public static final int** DESTROY\_ACTIVITY = 109;  **public static final int** BIND\_APPLICATION = 110;  **public static final int** EXIT\_APPLICATION = 111;  **public static final int** NEW\_INTENT = 112;  **public static final int** RECEIVER = 113;  **public static final int** CREATE\_SERVICE = 114;  **public static final int** SERVICE\_ARGS = 115;  **public static final int** STOP\_SERVICE = 116; |

ActivityThread通过ApplicationThread和AMS进行进程间通信，AMS以进程间通信的方式完成ActivityThread的请求后会回调ApplicationThread中的Binder方法，然后ApplicationThread会向H发送消息，H收到消息后会将ApplicationThread中的逻辑切换到ActivityThread中去执行，即切换到主线程中去执行，这个过程就是**主线程的消息循环模型。**

# Android的线程和线程池

除了最常使用的Thread之外, 在Android之中可以扮演线程角色的还有很多: 如AsyncTask和IntentService, 同时HandlerThread也是一种特殊的线程. 虽然这些线程的表现形式有别于基础线程. 但是本质上还是传统的线程. 例如AsyncTask它的底层使用了线程池. 而对于IntentService和HandlerThread来说, 他们的底层则直接使用了线程.

**根据不同的特性来实现不同的场景**

* AsyncTask: 封装了线程池和Handler, 它主要是为了方便开发者在子线程中更新UI.
* HandlerThread: 是一种具有消息循环的线程, 在它的内部可以使用Handler.
* IntentService: 是一个服务, 系统对其进行了封装使其可以更方便地执行后台任务, IntentService内部采用了HandlerThread来执行任务, 当任务执行完毕后IntentService会自动退出. 从执行任务的角度来看, 更像一个后台的线程. 但是因为其本身是一种服务, 所以导致不容易被系统杀死从而保证任务的执行. 而如果是一个后台线程, 由于这个时候进行中没有活动的四大组件, 那么这个进程的优先级会很低, 很容易被系统杀死, 这就是IntentService的优点.

**线程的简单概述**

在操作系统中, 线程是操作系统调度的最小单元, 同时线程又是一种受限的系统资源, 即线程不可能无限制的产生, 并且线程的创建和销毁都会有相应的开销. 当系统中存在大量的线程时, 系统会通过时间片转轮的方式调度每个线程, 因此线程不可能做到绝对的并行, 除非线程数量小于等于cpu的核心数. 但这种情况太少了, 所以线程池的概念就由此出现. 通过线程池就可以避免因为频繁创建和销毁线程所带来的系统开销。

## 主线程和子线程

主线程是指进程所拥有的线程, 在Java中默认情况下一个进程只有一个线程, 这个线程就是主线程. 主线程主要处理界面交互相关的逻辑, 因为用户随时会和界面发生交互, **因此主线程在任何时候都必须有较高的响应速度**, 否则就会产生一种界面卡顿的感觉. 为了保持较高的响应速度, 这就要求主线程中不能执行耗时的任务, 这个时候子线程就派上用场. 子线程也叫作工作线程, 除了主线程以外的线程都叫做子线程

Android沿用了Java的线程模型, 其中的线程也分为主线程和子线程, 其中主线程也叫UI线程. **主线程的作用是运行四大组件以及处理它们和用户的交互**. 而子线程的作用则是执行耗时任务, 比如网络请求, I/O操作等. 从Android 3.0 开始系统要求网络访问必须在子线程中进行, 否则网络访问将会失败并抛出NetworkOnMainThreadException这个异常, 这样做是为了避免主线程由于耗时操作所阻塞从而出现ANR现象.

## Android中的线程形态

### AsyncTask

*AsyncTask是一种轻量级的异步任务类, 他可以在线程池中执行后台任务, 然后把执行的进度和最终的结果传递给主线程并在主线程更新UI. 从实现上来说. AsyncTask封装了Thread和Handler, 通过AsyncTask可以更加方便地执行后台任务, 对于特别耗时的任务来说, 不建议用AsyncTask，建议使用线程池.*

public abstract class AsyncTask<Params, Progress, Result>

AsyncTask就是一个抽象的泛型类. 这三个泛型的意义.

* Params: 表示参数的类型
* Progress: 表示后台任务的执行进度的类型
* Result: 则表示后台任务的返回结果的类型

如果不需要传递具体的参数, 那么这三个泛型参数可以用Void来代替.

**AsyncTask提供了4个核心方法, 含义如下**

1. onPreExecute(): **在主线程执行**, 在异步任务执行之前, 此方法会被调用, 一般可以用于做一些准备工作
2. doInBackground(): **在线程池中执行**, 此方法用于执行异步任务, 参数params表示异步任务的输入参数. 在此方法中可以通过publishProgress()方法来更新任务的进度, publishProgress()方法会调用onProgressUpdate()方法. 另外此方法需要返回计算结果给onPostExecute()
3. onProgressUpdate(): **在主线程执行**,在异步任务执行之后, 此方法会被调用, 其中result参数是后台任务的返回值, 即doInBackground()的返回值.
4. onPostExecute(): **在主线程执行**, 在异步任务执行之后, 此方法会被调用, 其中result参数是后台任务的返回值, 即doInBackground的返回值.

除了上述的四种方法,还有onCancelled(), 它同样在主线程执行, 当异步任务被取消时, onCancelled()方法会被调用, 这个时候onPostExecute()则不会被调用.

**AsyncTask在使用过程中有一些条件限制**

1. AsyncTask的类必须在主线程被加载, 这就意味着第一次访问AsyncTask必须发生在主线程, 这个问题不是绝对, 因为在Android 4.1及以上的版本已经被系统自动完成. 在Android 5.0的源码中, 可以看到ActivityThread#main()会调用AsyncTask#init()方法.
2. AsyncTask的对象必须在主线程中创建.
3. execute方法必须在UI线程调用.
4. 不要在程序中直接调用onPreExecute(), onPostExecute(), doInBackground和onProgressUpdate()
5. 一个AsyncTask对象只能执行一次, 即只能调用一次execute()方法, 否则会报运行时异常.
6. 在Android 1.6之前, AsyncTask是串行执行任务的; Android 1.6的时候AsyncTask开始采用线程池里处理并行任务; 但是Android 3.0开始, 为了避免AsyncTask带来的并发错误, AsyncTask又采用了一个线程来串行的执行任务. 尽管如此在3.0以后, 仍然可以通过AsyncTask#executeOnExecutor()方法来并行执行任务.

### AsyncTask的工作原理

这里以源码5.0来分析, 不同的版本源码具体实现是不同的.

为了分析AsyncTask的工作原理, 可以从它的execute()方法开始分析, execute()方法又会调用executeOnExecutor()方法. 实际上这里调用进来后是一个串行的线程池, 一个进程中所有的AsyncTask全都在这个串行的线程池中排队执行, 然后会先调用AsyncTask#onPreExecute()方法, 然后线程池开始执行. 看看SerialExecutor()的源码实现:

|  |
| --- |
| public static final Executor SERIAL\_EXECUTOR = new SerialExecutor();  private static volatile Executor sDefaultExecutor = SERIAL\_EXECUTOR;  private static class SerialExecutor implements Executor {  final ArrayDeque<Runnable> mTasks = new ArrayDeque<Runnable>();  Runnable mActive;  public synchronized void execute(final Runnable r) {  mTasks.offer(new Runnable() {  public void run() {  try {  r.run();  } finally {  scheduleNext();  }  }  });  if (mActive == null) {  scheduleNext();  }  }  protected synchronized void scheduleNext() {  if ((mActive = mTasks.poll()) != null) {  THREAD\_POOL\_EXECUTOR.execute(mActive);  }  }  } |

在SerialExecutor的实现可以分析AsyncTask的排队执行的过程. 首先系统会把AsyncTask#Params参数封装成FutureTask对象, FutureTask是一个并发类, 在这里充当了Runnable的作用. 接着这个FutureTask会交给SerialExecutor#execute()方法去处理. 这个方法首先会把FutureTask对象插入到任务队列mTasks中, 如果这个时候没有正在活动AsyncTask任务, 那么就会调用SerialExecutor#scheduleNext()方法来执行下一个AsyncTask任务. 同时当一个AsyncTask任务执行完后, AsyncTask会继续执行其他任务直到所有的任务都执行完毕为止, 从这一点可以看出, 在默认情况下, **AsyncTask是串行执行的**

AsyncTask中有两个线程池(SerialExecutor和THREAD\_POOL\_EXECUTOR)和一个Handler(InternalHandler), 其中线程池SerialExecutor用于任务的排列, 而线程池THREAD\_POOL\_EXECUTOR用于真正的执行任务, 而InternalHandler用于将执行环境从线程切换到主线程, 其本质仍然是线程的调用过程.

在AsyncTask的构造方法中有如下这段代码, 由于FutureTask#run()方法会调用mWorker.call()方法, 因此mWorker的call方法最终会在线程池中执行.

|  |
| --- |
| mWorker = new WorkerRunnable<Params, Result>() {  public Result call() throws Exception {  mTaskInvoked.set(true);  Process.setThreadPriority(Process.THREAD\_PRIORITY\_BACKGROUND);  *//noinspection unchecked*  return postResult(doInBackground(mParams));  }  }; |

在mWorker.call()方法中, 首先将mTaskInvoked设为了true. 表示当前任务已经被调用过了. 然后执行AsyncTask#doInBackground()方法, 接着将其返回值传递给postResult(), 这个方法的实现:

|  |
| --- |
| private Result postResult(Result result) {  @SuppressWarnings("unchecked")  Message message = sHandler.obtainMessage(MESSAGE\_POST\_RESULT,  new AsyncTaskResult<Result>(this, result));  message.sendToTarget();  return result;  } |

这里, postResult()会通过sHandler发送一个MESSAGE\_POST\_RESULT消息, 而sHandler的定义如下:

|  |
| --- |
| private static final InternalHandler sHandler = new InternalHandler();  private static class InternalHandler extends Handler {  @SuppressWarnings({"unchecked", "RawUseOfParameterizedType"})  @Override  public void handleMessage(Message msg) {  AsyncTaskResult result = (AsyncTaskResult) msg.obj;  switch (msg.what) {  case MESSAGE\_POST\_RESULT:  *// There is only one result*  result.mTask.finish(result.mData[0]);  break;  case MESSAGE\_POST\_PROGRESS:  result.mTask.onProgressUpdate(result.mData);  break;  }  }  } |

可以看出sHandler是一个静态的Handler对象, 为了能够将执行环境切换到主线程, 这就要求sHandler这个对象必须在主线程中创建. 由于静态成员会在加载类的时候进行初始化, 因此这就变相要求AsyncTask的类必须在主线程中加载, 否则同一个进程中的AsyncTask都无法正常工作. sHandler收到了消息后会调用AsyncTask#finish()方法, 如下:

|  |
| --- |
| private void finish(Result result) {  if (isCancelled()) {  onCancelled(result);  } else {  onPostExecute(result);  }  mStatus = Status.FINISHED;  } |

这个finish()方法很简单, 如果AsyncTask被取消了, 那么就调用onCancelled()方法, 否则就会调用onPostExecute()方法, 可以看到doInBackground的反馈结果会传递给onPostExecute()方法.

通过源码分析, 可以确定从3.0开始, 默认情况下AsyncTask就是串行的. 通过一段代码测试.

|  |
| --- |
| @Override  public void onClick(View v) {  switch (v.getId()){  case R.id.btn\_main:  new MyAsync("任务\_1").execute("");  new MyAsync("任务\_2").execute("");  new MyAsync("任务\_3").execute("");  new MyAsync("任务\_4").execute("");  new MyAsync("任务\_5").execute("");  break;  }  }  private static class MyAsync extends AsyncTask<String, Integer, String>{  private static final String TAG = MyAsync.class.getSimpleName();  private final String mTaskName;  public MyAsync(String taskName){  mTaskName = taskName;  }    @Override  protected String doInBackground(String... params) {  SystemClock.sleep(3000);  return mTaskName;  }  @Override  protected void onPostExecute(String s) {  super.onPostExecute(s);  SimpleDateFormat df = new SimpleDateFormat("HH:mm:ss");  Log.e(TAG, s+" onPostExecute finish time: " +df.format(new Date()));  }  } |

给按钮添加一个点击触发, 点击的时候会触发五个AsyncTask的创建和执行 , 看一下结果



在5.0机器上测试所有的任务是串行执行的. 总共耗费了10秒. 而在2.x的版本所有的结束时间却都是一样的.

如果使用executeOnExecutor()那么结果看一下:

*/\*\**

\* 在版本3.0以上使用并行的方式开启

\*/

private void checkConcurrent() {

new MyAsync("任务\_1").executeOnExecutor(AsyncTask.THREAD\_POOL\_EXECUTOR,"");

new MyAsync("任务\_2").executeOnExecutor(AsyncTask.THREAD\_POOL\_EXECUTOR,"");

new MyAsync("任务\_3").executeOnExecutor(AsyncTask.THREAD\_POOL\_EXECUTOR,"");

new MyAsync("任务\_4").executeOnExecutor(AsyncTask.THREAD\_POOL\_EXECUTOR,"");

new MyAsync("任务\_5").executeOnExecutor(AsyncTask.THREAD\_POOL\_EXECUTOR,"");

}

ok, 按照你的需求可以选择使用哪种实现方式.

### HandlerThread

HandlerThread继承了Thread, 它是一种可以使用Handler的Thread, 它的实现也很简单, 就是run方法中通过Looper.prepare()来创建消息队列, 并通过Looper.loop()来开启消息循环, 这样在实际的使用中就允许在HandlerThread中创建Handler, 看一下run()方法.

|  |
| --- |
| @Override  public void run() {  mTid = Process.myTid();  Looper.prepare();  synchronized (this) {  mLooper = Looper.myLooper();  notifyAll();  }  Process.setThreadPriority(mPriority);  onLooperPrepared();  Looper.loop();  mTid = -1;  } |

从HandlerThread的实现来看, 它和普通的Thread有显著的不同之处. 普通的Thread主要用于在run方法中执行一个耗时任务; 而HandlerThread在内部创建了消息队列, 外界需要通过Handler的消息方式来通知HandlerThread执行一个具体的任务. HandlerThread是一个很有用的类, 在Android中一个具体使用场景就是IntentService.

由于HandlerThread#run()是一无限线循环方法, 因此当明确不需要再使用HandlerThread时, 最好通过quit()或者quitSafely()方法来终止线程的执行.

### IntentService

IntentService是一种特殊的Service, 它继承了Service并且它是一个抽象类, 因此需要创建子类才能使用. IntentService可以用于执行后台耗时任务, 当任务执行后会自动停止, 同时由于本质是服务的原因, 这导致了它的优先级比单纯的线程要高很多, 所以IntentService比较适合执行一些高优先级的后台任务.

IntentService封装了Handler和HandlerThread. 这是在onCreate()来实现的.

|  |
| --- |
| @Override  public void onCreate() {  *// TODO: It would be nice to have an option to hold a partial wakelock*  *// during processing, and to have a static startService(Context, Intent)*  *// method that would launch the service & hand off a wakelock.*  super.onCreate();  HandlerThread thread = new HandlerThread("IntentService[" + mName + "]");  thread.start();  mServiceLooper = thread.getLooper();  mServiceHandler = new ServiceHandler(mServiceLooper);  } |

当第一次启动的时候, onCreate()就会被调用, 内部就会创建一个HandlerThread, 然后使用它的Looper来构造一个Handler对象mServiceHandler. 这样通过 mServiceHandler发送的消息最终都会在HandlerThread中执行, 从这个角度看IntentService也可以用于执行后台任务.

每次启动IntentService的时候, 它的onStartCommand()方法就会被调用一次, IntentService在onStartCommand()中处理每个后台任务的Intent. 看看源码是如何处理外界的Intent的.

首先onStartCommand()调用了onStart()

|  |
| --- |
| @Override **public void** onStart(@Nullable Intent intent, **int** startId) {  Message msg = **mServiceHandler**.obtainMessage();  msg.**arg1** = startId;  msg.**obj** = intent;  **mServiceHandler**.sendMessage(msg); } |

这个方法只是通过mServiceHandler发送了一个消息, 所以这个消息会在HandlerThread中被处理. 消息收到后, 会将Intent对象传递给onHandlerIntent()方法去处理. 注意这个Intent对象和外界startService()参数传递内容是一样的. 通过Intent的参数就可以区分具体的后台任务, 这样在onHandlerIntent()方法中就可以对不同的后台任务做处理了.

当onHandlerIntent()方法执行完毕后, IntentService会通过stopSelf()方法来停止服务. 这里使用的方法是有参数的, 不使用无参数的stopSelf()是因为无参函数会立刻停止服务, 可能会导致还有没执行完的任务失效.

有参的stopSelf(int startId)在尝试 停止服务之前会判断最近启动的服务次数是否和startId这个参数值相等, 如果相等就立刻停止服务, 否则反之. 这个策略可以从AMS#stopServiceToken()方法的实现中找到依据.

IntentService#onHandleIntent()方法是一个抽象方法, 他需要我们在子类中实现, 它的作用是从Intent参数中区分具体的任务并执行这些任务. 如果目前只存在一个后台任务, 那么onHandleIntent()方法执行完成后, stopSelf(int startId)会立即停止服务; 如果目前存在多个后台任务, 那么当onHandleIntent()方法执行完最后一个任务时, stopSelf(int startId)才会停止任务. 另外由于每次执行一个后台任务都必须启动一次IntentService, 而其内部则通过消息的方式向HandlerThread请求执行任务, Handler中的Looper是顺序处理消息的, 这就意味着IntentService也是顺序执行后台任务的, 当有多个后台任务同时存在时, 这些后台任务会按照外界发起的顺序排队执行.

用代码来演示一遍:

可以看出, 三个后台任务是串行执行的, 他们的执行顺序就是发起请求的顺序. 当任务3完成了之后.KIntentService才真正的停止, 因为这是时候执行了onDestroy().

注意要在**manifest**声明该服务

|  |
| --- |
| **public class** KIntentService **extends** IntentService {  **private static final** String ***TAG*** = KIntentService.**class**.getSimpleName();  **public** KIntentService() {  **super**(***TAG***);  }  @Override  **protected void** onHandleIntent(@Nullable Intent intent) {  String task = intent.getStringExtra(**"task"**);  Log.*d*(***TAG***, **"receiver task :"**+task);  SystemClock.*sleep*(2000);  }  @Override  **public void** onDestroy() {  Log.*w*(***TAG***, **"onDestroy: 准备关闭"** );  **super**.onDestroy();  } }  *// 开启3个服务* Intent intent = **new** Intent(**this**, KIntentService.**class**); intent.putExtra(**"task"**, **"hi, 我是数据1"**); startService(intent); intent.putExtra(**"task"**, **"hi, 我是数据2"**); startService(intent); intent.putExtra(**"task"**, **"hi, 我是数据3"**); startService(intent); |

## Android中的线程池

概括一下线程池的优点:

1. 重用线程池中的线程, 避免因为线程的创建和销毁所带来的性能开销.
2. 能有效控制线程的最大并大数, 避免大量的线程之间因互相抢占系统资源而导致的阻塞现象.
3. 能够对线程进行简单的管理, 并提供定时执行以及指定间隔循环执行等能力.

Android中的线程池的概念来源于Java中的Executor, Executor是一个接口, 真正的线程池的实现为ThreadPoolExecutor. ThreadPoolExecutor提供了一系列参数来配制线程池, 通过不同的参数可以创建不同的线程池. 而从功能的特性来分的话可以分成四类. 下面说明.

### ThreadPoolExecutor

ThreadPoolExecutor是线程池的真正实现, 它的构造方法提供了一系列参数来配置线程池, 下面对构造方法中参数进行一下说明, 这些参数将会直接影响到线程池的功能特性.

|  |
| --- |
| public ThreadPoolExecutor(int corePoolSize,  int maximumPoolSize,  long keepAliveTime,  TimeUnit unit,  BlockingQueue<Runnable> workQueue) {  this(corePoolSize, maximumPoolSize, keepAliveTime, unit, workQueue,  Executors.defaultThreadFactory(), defaultHandler);  } |

* corePoolSize: 线程池的核心线程数, 默认情况下, 核心线程会在线程池中一直存活, 即使都处于闲置状态. 如果将ThreadPoolExecutor#allowCoreThreadTimeOut属性设置为true, 那么闲置的核心线程在等待新任务到来时会有超时的策略, 这个时间间隔由keepAliveTime属性来决定. 当等待时间超过了keepAliveTime设定的值那么核心线程将会终止.
* maximumPoolSize: 线程池所能容纳的最大线程数, 当活动线程数达到这个数值之后, 后续的任务将会被阻塞.
* keepAliveTime: 非核心线程闲置的超时时长, 超过这个时长, 非核心线程就会被回收.allowCoreThreadTimeOut这个属性为true的时候, 这个属性同样会作用于核心线程.
* unit: 用于指定keepAliveTime参数的时间单位, 这是一个枚举, 常用的有TimeUtil.MILLISECONDS(毫秒), TimeUtil.SECONDS(秒)以及TimeUtil.MINUTES(分)
* workQueue: 线程池中的任务队列, 通过线程池的execute方法提交的Runnable对象会存储在这个参数中.
* threadFactory: 线程工厂, 为线程池提供创建新线程的功能. ThreadFactory是一个接口.

**线程池执行任务时大致遵循如下规则:**

1. 如果线程池中的线程数量未达到核心线程的数量, 那么会直接启动一个核心线程来执行任务.
2. 如果线程池中的线程数量已经达到或者超过核心线程的数量, 那么任务会被插入到任务队列中排队等待执行.
3. 如果在步骤2中无法将任务插入到任务队列中, 这通常是因为任务队列已满, 这个时候如果线程数量未达到线程池的规定的最大值, 那么会立刻启动一个非核心线程来执行任务.
4. 如果步骤3中的线程数量已经达到最大值的时候, 那么会拒绝执行此任务, ThreadPoolExecutor会调用RejectedExecution方法来通知调用者.

ThreadPoolExecutor的参数配置在AsyncTask中有明显的体现, 下面是其配置情况



这个配置后的规格是这样的

* 核心线程数等于CPU核心数 + 1;
* 线程池的最大线程数为CPU核心数的2倍 + 1;
* 核心线程无超时机制, 非核心线程在闲置时有超时机制,超时时间为1秒.
* 任务队列的容量为128

### 线程池的分类

**1. FixedThreadPool**

**通过Executors#newFixedThreadPool()方法来创建.** 它是一种线程数量固定的线程池, 当线程处于空闲状态时, 它们并不会被回收, 除非线程池关闭了. 当所有的线程都处于活动状态时, 新任务都会处于等待状态, 直到有线程空闲出来. 由于FixedThreadPool只有核心线程并且这些核心线程不会被回收, **这意味着它能够更加快速地响应外界的请求.**

**2. CachedThreadPool**  
**通过Executors#newCachedThreadPool()方法来创建.** 它是一种线程数量不定的线程池, 它只有非核心线程, 并且其最大值线程数为Integer.MAX\_VALUE. 这就可以认为这个最大线程数为任意大了. 当线程池中的线程都处于活动的时候, 线程池会创建新的线程来处理新任务, 否则就会利用空闲的线程来处理新任务. 线程池中的空闲线程都有超时机制, 这个超时时长为60S, 超过这个时间那么空闲线程就会被回收.

和FixedThreadPool不同的是, CachedThreadPool的任务队列其实相当于一个空集合, 这将导致任何任务都会立即被执行, 因为在这种场景下SynchronousQueue是无法插入任务的. SynchronousQueue是一个非常特殊的队列, 在很多情况下可以把它简单理解为一个无法存储元素的队列. **在实际使用中很少使用.这类线程比较适合执行大量的耗时较少的任务**

**3. ScheduledThreadPool**

**通过Executors#newScheduledThreadPool()方法来创建.** 它的核心线程数量是固定的, 而非核心线程数是没有限制的, 并且当非核心线程闲置时会立刻被回收掉. **这类线程池用于执行定时任务和具有固定周期的重复任务**

**4. SingleThreadExecutor**

**通过Executors#newSingleThreadPool()方法来创建.** 这类线程池内部只有一个核心线程, 它确保所有的任务都在同一个线程中按顺序执行. **这类线程池意义在于统一所有的外界任务到一个线程中, 这使得在这些任务之间不需要处理线程同步的问题**

Android Handler 异步消息处理机制的妙用 创建强大的图片加载类：（这个很好！！！自己动手实现以下）

http://blog.csdn.net/lmj623565791/article/details/38476887

# Android 异步消息处理机制

Android 异步消息处理机制 让你深入理解 Looper、Handler、Message三者关系：

http://blog.csdn.net/lmj623565791/article/details/38377229

android：LruCache缓存小结

http://www.csdn123.com/html/topnews201408/73/10373.htm

LinkedHashMap

http://www.cnblogs.com/children/archive/2012/10/02/2710624.html

1.最简单的使用实例

http://blog.csdn.net/liuhe688/article/details/6407225

在分析Android消息机制之前，我们先来看一段代码：

[java] view plaincopy

public class MainActivity extends Activity implements View.OnClickListener {

private TextView stateText;

private Button btn;

@Override

public void onCreate(Bundle savedInstanceState) {

super.onCreate(savedInstanceState);

setContentView(R.layout.main);

stateText = (TextView) findViewById(R.id.tv);

btn = (Button) findViewById(R.id.btn);

btn.setOnClickListener(this);

}

@Override

public void onClick(View v) {

new WorkThread().start();

}

//工作线程

private class WorkThread extends Thread {

@Override

public void run() {

//......处理比较耗时的操作

//处理完成后改变状态

stateText.setText("completed");

}

}

}

这段代码似乎看上去很正常，但是当你运行时就会发现，它会报一个致命性的异常：

[java] view plaincopy

ERROR/AndroidRuntime(421): FATAL EXCEPTION: Thread-8

ERROR/AndroidRuntime(421): android.view.ViewRoot$CalledFromWrongThreadException:

Only the original thread that created a view hierarchy can touch its views.

到底是怎么回事呢？原因在于，Android系统中的视图组件并不是线程安全的，如果要更新视图，必须在主线程中更新，不可以在子线程中执行更新的操作。

既然这样，我们就在子线程中通知主线程，让主线程做更新操作吧。那么，我们如何通知主线程呢？我们需要使用到Handler对象。

我们稍微修改一下上面的代码：

[java] view plaincopy

public class MainActivity extends Activity implements View.OnClickListener {

private static final int COMPLETED = 0;

private TextView stateText;

private Button btn;

private Handler handler = new Handler() {

@Override

public void handleMessage(Message msg) {

if (msg.what == COMPLETED) {

stateText.setText("completed");

}

}

};

@Override

public void onCreate(Bundle savedInstanceState) {

super.onCreate(savedInstanceState);

setContentView(R.layout.main);

stateText = (TextView) findViewById(R.id.tv);

btn = (Button) findViewById(R.id.btn);

btn.setOnClickListener(this);

}

@Override

public void onClick(View v) {

new WorkThread().start();

}

//工作线程

private class WorkThread extends Thread {

@Override

public void run() {

//......处理比较耗时的操作

//处理完成后给handler发送消息

Message msg = new Message();

msg.what = COMPLETED;

handler.sendMessage(msg);

}

}

}

通过上面这种方式，我们就可以解决线程安全的问题，把复杂的任务处理工作交给子线程去完成，然后子线程通过handler对象告知主线程，由主线程更新视图，这个过程中消息机制起着重要的作用。

下面，我们就来分析一下Android中的消息机制。

熟悉Windows编程的朋友知道Windows程序是消息驱动的，并且有全局的消息循环系统。Google参考了Windows的消息循环机制，也在Android系统中实现了消息循环机制。Android通过Looper、Handler来实现消息循环机制。Android的消息循环是针对线程的，每个线程都可以有自己的消息队列和消息循环。

Android系统中的Looper负责管理线程的消息队列和消息循环。通过Looper.myLooper()得到当前线程的Looper对象，通过Looper.getMainLooper()得到当前进程的主线程的Looper对象。

前面提到，Android的消息队列和消息循环都是针对具体线程的，一个线程可以存在一个消息队列和消息循环，特定线程的消息只能分发给本线程，不能跨线程和跨进程通讯。但是创建的工作线程默认是没有消息队列和消息循环的，如果想让工作线程具有消息队列和消息循环，就需要在线程中先调用Looper.prepare()来创建消息队列，然后调用Looper.loop()进入消息循环。下面是我们创建的工作线程：

[java] view plaincopy

class WorkThread extends Thread {

public Handler mHandler;

public void run() {

Looper.prepare();

mHandler = new Handler() {

public void handleMessage(Message msg) {

// 处理收到的消息

}

};

Looper.loop();

}

}

这样一来，我们创建的工作线程就具有了消息处理机制了。

那么，为什么前边的示例中，我们怎么没有看到Looper.prepare()和Looper.loop()的调用呢？原因在于，我们的Activity是一个UI线程，运行在主线程中，Android系统会在Activity启动时为其创建一个消息队列和消息循环。

前面提到最多的是消息队列(MessageQueue)和消息循环(Looper)，但是我们看到每个消息处理的地方都有Handler的存在，它是做什么的呢？Handler的作用是把消息加入特定的Looper所管理的消息队列中，并分发和处理该消息队列中的消息。构造Handler的时候可以指定一个Looper对象，如果不指定则利用当前线程的Looper对象创建。下面是Handler的两个构造方法：

[java] view plaincopy

/\*\*

\* Default constructor associates this handler with the queue for the

\* current thread.

\*

\* If there isn't one, this handler won't be able to receive messages.

\*/

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) {

throw new RuntimeException(

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

}

mQueue = mLooper.mQueue;

mCallback = null;

}

/\*\*

\* Use the provided queue instead of the default one.

\*/

public Handler(Looper looper) {

mLooper = looper;

mQueue = looper.mQueue;

mCallback = null;

}

下面是消息机制中几个重要成员的关系图：

一个Activity中可以创建出多个工作线程，如果这些线程把他们消息放入Activity主线程的消息队列中，那么消息就会在主线程中处理了。因为主线程一般负责视图组件的更新操作，对于不是线程安全的视图组件来说，这种方式能够很好的实现视图的更新。

那么，子线程如何把消息放入主线程的消息队列中呢？只要Handler对象以主线程的Looper创建，那么当调用Handler的sendMessage方法，系统就会把消息主线程的消息队列，并且将会在调用handleMessage方法时处理主线程消息队列中的消息。

对于子线程访问主线程的Handler对象，你可能会问，多个子线程都访问主线程的Handler对象，发送消息和处理消息的过程中会不会出现数据的不一致呢？答案是Handler对象不会出现问题，因为Handler对象管理的Looper对象是线程安全的，不管是添加消息到消息队列还是从消息队列中读取消息都是同步保护的，所以不会出现数据不一致现象。

深入理解Android消息处理机制对于应用程序开发非常重要，也可以让我们对线程同步有更加深刻的认识，希望这篇文章可以对朋友们有所帮助。

2.android的消息处理机制（图+源码分析）——Looper,Handler,Message源码剖析

作为一个大三的预备程序员，我学习android的一大乐趣是可以通过源码学习google大牛们的设计思想。android源码中包含了大量的设计模式，除此以外，android sdk还精心为我们设计了各种helper类，对于和我一样渴望水平得到进阶的人来说，都太值得一读了。这不，前几天为了了解android的消息处理机制，我看了Looper，Handler，Message这几个类的源码，结果又一次被googler的设计震撼了，特与大家分享。

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被封装到Looper里面了，我们不会直接与MQ打交道，因此我没将其作为核心类。下面一一介绍：

线程的魔法师 Looper

Looper的字面意思是“循环者”，它被设计用来使一个普通线程变成Looper线程。所谓Looper线程就是循环工作的线程。在程序开发中（尤其是GUI开发中），我们经常会需要一个线程不断循环，一旦有新任务则执行，执行完继续等待下一个任务，这就是Looper线程。使用Looper类创建Looper线程很简单：

public class LooperThread extends Thread {

@Override

public void run() {

// 将当前线程初始化为Looper线程

Looper.prepare();

// ...其他处理，如实例化handler

// 开始循环处理消息队列

Looper.loop();

}

}

通过上面两行核心代码，你的线程就升级为Looper线程了！！！是不是很神奇？让我们放慢镜头，看看这两行代码各自做了什么。

1)Looper.prepare()

通过上图可以看到，现在你的线程中有一个Looper对象，它的内部维护了一个消息队列MQ。注意，一个Thread只能有一个Looper对象，为什么呢？咱们来看源码。

public class Looper {

// 每个线程中的Looper对象其实是一个ThreadLocal，即线程本地存储(TLS)对象

private static final ThreadLocal sThreadLocal = new ThreadLocal();

// Looper内的消息队列

final MessageQueue mQueue;

// 当前线程

Thread mThread;

// 。。。其他属性

// 每个Looper对象中有它的消息队列，和它所属的线程

private Looper() {

mQueue = new MessageQueue();

mRun = true;

mThread = Thread.currentThread();

}

// 我们调用该方法会在调用线程的TLS中创建Looper对象

public static final void prepare() {

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

// 试图在有Looper的线程中再次创建Looper将抛出异常

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

}

sThreadLocal.set(new Looper());

}

// 其他方法

}

通过源码，prepare()背后的工作方式一目了然，其核心就是将looper对象定义为ThreadLocal。如果你还不清楚什么是ThreadLocal，请参考《理解ThreadLocal》。

2）Looper.loop()

调用loop方法后，Looper线程就开始真正工作了，它不断从自己的MQ中取出队头的消息(也叫任务)执行。其源码分析如下：

public static final void loop() {

Looper me = myLooper(); //得到当前线程Looper

MessageQueue queue = me.mQueue; //得到当前looper的MQ

// 这两行没看懂= = 不过不影响理解

Binder.clearCallingIdentity();

final long ident = Binder.clearCallingIdentity();

// 开始循环

while (true) {

Message msg = queue.next(); // 取出message

if (msg != null) {

if (msg.target == null) {

// message没有target为结束信号，退出循环

return;

}

// 日志。。。

if (me.mLogging!= null) me.mLogging.println(

">>>>> Dispatching to " + msg.target + " "

+ msg.callback + ": " + msg.what

);

// 非常重要！将真正的处理工作交给message的target，即后面要讲的handler

msg.target.dispatchMessage(msg);

// 还是日志。。。

if (me.mLogging!= null) me.mLogging.println(

"<<<<< Finished to " + msg.target + " "

+ msg.callback);

// 下面没看懂，同样不影响理解

final long newIdent = Binder.clearCallingIdentity();

if (ident != newIdent) {

Log.wtf("Looper", "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);

}

// 回收message资源

msg.recycle();

}

}

}

除了prepare()和loop()方法，Looper类还提供了一些有用的方法，比如

Looper.myLooper()得到当前线程looper对象：

public static final Looper myLooper() {

// 在任意线程调用Looper.myLooper()返回的都是那个线程的looper

return (Looper)sThreadLocal.get();

}

getThread()得到looper对象所属线程：

public Thread getThread() {

return mThread;

}

quit()方法结束looper循环：

public void quit() {

// 创建一个空的message，它的target为NULL，表示结束循环消息

Message msg = Message.obtain();

// 发出消息

mQueue.enqueueMessage(msg, 0);

}

到此为止，你应该对Looper有了基本的了解，总结几点：

1.每个线程有且最多只能有一个Looper对象，它是一个ThreadLocal

2.Looper内部有一个消息队列，loop()方法调用后线程开始不断从队列中取出消息执行

3.Looper使一个线程变成Looper线程。

那么，我们如何往MQ上添加消息呢？下面有请Handler！（掌声~~~）

异步处理大师 Handler

什么是handler？handler扮演了往MQ上添加消息和处理消息的角色（只处理由自己发出的消息），即通知MQ它要执行一个任务(sendMessage)，并在loop到自己的时候执行该任务(handleMessage)，整个过程是异步的。handler创建时会关联一个looper，默认的构造方法将关联当前线程的looper，不过这也是可以set的。默认的构造方法：

public class handler {

final MessageQueue mQueue; // 关联的MQ

final Looper mLooper; // 关联的looper

final Callback mCallback;

// 其他属性

public Handler() {

// 没看懂，直接略过，，，

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());

}

}

// 默认将关联当前线程的looper

mLooper = Looper.myLooper();

// looper不能为空，即该默认的构造方法只能在looper线程中使用

if (mLooper == null) {

throw new RuntimeException(

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

}

// 重要！！！直接把关联looper的MQ作为自己的MQ，因此它的消息将发送到关联looper的MQ上

mQueue = mLooper.mQueue;

mCallback = null;

}

// 其他方法

}

下面我们就可以为之前的LooperThread类加入Handler：

public class LooperThread extends Thread {

private Handler handler1;

private Handler handler2;

@Override

public void run() {

// 将当前线程初始化为Looper线程

Looper.prepare();

// 实例化两个handler

handler1 = new Handler();

handler2 = new Handler();

// 开始循环处理消息队列

Looper.loop();

}

}

加入handler后的效果如下图：

可以看到，一个线程可以有多个Handler，但是只能有一个Looper！

Handler发送消息

有了handler之后，我们就可以使用 post(Runnable), postAtTime(Runnable, long), postDelayed(Runnable, long), sendEmptyMessage(int),sendMessage(Message), sendMessageAtTime(Message, long)和 sendMessageDelayed(Message, long)这些方法向MQ上发送消息了。光看这些API你可能会觉得handler能发两种消息，一种是Runnable对象，一种是message对象，这是直观的理解，但其实post发出的Runnable对象最后都被封装成message对象了，见源码：

// 此方法用于向关联的MQ上发送Runnable对象，它的run方法将在handler关联的looper线程中执行

public final boolean post(Runnable r)

{

// 注意getPostMessage(r)将runnable封装成message

return sendMessageDelayed(getPostMessage(r), 0);

}

private final Message getPostMessage(Runnable r) {

Message m = Message.obtain(); //得到空的message

m.callback = r; //将runnable设为message的callback，

return m;

}

public boolean sendMessageAtTime(Message msg, long uptimeMillis)

{

boolean sent = false;

MessageQueue queue = mQueue;

if (queue != null) {

msg.target = this; // message的target必须设为该handler！

sent = queue.enqueueMessage(msg, uptimeMillis);

}

else {

RuntimeException e = new RuntimeException(

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

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

}

return sent;

}

其他方法就不罗列了，总之通过handler发出的message有如下特点：

1.message.target为该handler对象，这确保了looper执行到该message时能找到处理它的handler，即loop()方法中的关键代码

msg.target.dispatchMessage(msg);

2.post发出的message，其callback为Runnable对象

Handler处理消息

说完了消息的发送，再来看下handler如何处理消息。消息的处理是通过核心方法dispatchMessage(Message msg)与钩子方法handleMessage(Message msg)完成的，见源码

// 处理消息，该方法由looper调用

public void dispatchMessage(Message msg) {

if (msg.callback != null) {

// 如果message设置了callback，即runnable消息，处理callback！

handleCallback(msg);

} else {

// 如果handler本身设置了callback，则执行callback

if (mCallback != null) {

/\* 这种方法允许让activity等来实现Handler.Callback接口，避免了自己编写handler重写handleMessage方法。见http://alex-yang-xiansoftware-com.iteye.com/blog/850865 \*/

if (mCallback.handleMessage(msg)) {

return;

}

}

// 如果message没有callback，则调用handler的钩子方法handleMessage

handleMessage(msg);

}

}

// 处理runnable消息

private final void handleCallback(Message message) {

message.callback.run(); //直接调用run方法！

}

// 由子类实现的钩子方法

public void handleMessage(Message msg) {

}

可以看到，除了handleMessage(Message msg)和Runnable对象的run方法由开发者实现外（实现具体逻辑），handler的内部工作机制对开发者是透明的。这正是handler API设计的精妙之处！

Handler的用处

我在小标题中将handler描述为“异步处理大师”，这归功于Handler拥有下面两个重要的特点：

1.handler可以在任意线程发送消息，这些消息会被添加到关联的MQ上。

2.handler是在它关联的looper线程中处理消息的。

这就解决了android最经典的不能在其他非主线程中更新UI的问题。android的主线程也是一个looper线程(looper在android中运用很广)，我们在其中创建的handler默认将关联主线程MQ。因此，利用handler的一个solution就是在activity中创建handler并将其引用传递给worker thread，worker thread执行完任务后使用handler发送消息通知activity更新UI。(过程如图)

下面给出sample代码，仅供参考：

public class TestDriverActivity extends Activity {

private TextView textview;

@Override

protected void onCreate(Bundle savedInstanceState) {

super.onCreate(savedInstanceState);

setContentView(R.layout.main);

textview = (TextView) findViewById(R.id.textview);

// 创建并启动工作线程

Thread workerThread = new Thread(new SampleTask(new MyHandler()));

workerThread.start();

}

public void appendText(String msg) {

textview.setText(textview.getText() + "\n" + msg);

}

class MyHandler extends Handler {

@Override

public void handleMessage(Message msg) {

String result = msg.getData().getString("message");

// 更新UI

appendText(result);

}

}

}

public class SampleTask implements Runnable {

private static final String TAG = SampleTask.class.getSimpleName();

Handler handler;

public SampleTask(Handler handler) {

super();

this.handler = handler;

}

@Override

public void run() {

try { // 模拟执行某项任务，下载等

Thread.sleep(5000);

// 任务完成后通知activity更新UI

Message msg = prepareMessage("task completed!");

// message将被添加到主线程的MQ中

handler.sendMessage(msg);

} catch (InterruptedException e) {

Log.d(TAG, "interrupted!");

}

}

private Message prepareMessage(String str) {

Message result = handler.obtainMessage();

Bundle data = new Bundle();

data.putString("message", str);

result.setData(data);

return result;

}

}

当然，handler能做的远远不仅如此，由于它能post Runnable对象，它还能与Looper配合实现经典的Pipeline Thread(流水线线程)模式。请参考此文《Android Guts: Intro to Loopers and Handlers》

封装任务 Message

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。message的用法比较简单，这里不做总结了。但是有这么几点需要注意（待补充）：

1.尽管Message有public的默认构造方法，但是你应该通过Message.obtain()来从消息池中获得空消息对象，以节省资源。

2.如果你的message只需要携带简单的int信息，请优先使用Message.arg1和Message.arg2来传递信息，这比用Bundle更省内存

3.擅用message.what来标识信息，以便用不同方式处理message。

（完） PS：写了好久啊，觉得还不错的话给个推荐哦亲

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

# 实际运用BlockCanary

消息机制的使用

<http://blog.zhaiyifan.cn/2016/01/16/BlockCanaryTransparentPerformanceMonitor/>