1、Handler的由来

当程序第一次启动的时候，Android会同时启动一条主线程（ Main Thread）来负责处理与UI相关的事件，我们叫做UI线程。

Android的UI操作并不是线程安全的（出于性能优化考虑），意味着如果多个线程并发操作UI线程，可能导致线程安全问题。

为了解决Android应用多线程问题—Android平台只允许UI线程修改Activity里的UI组建，就会导致新启动的线程无法改变界面组建的属性值。

简单的说：当主线程队列处理一个消息超过5秒,android 就会抛出一个 ANP(无响应)的异常,所以,我们需要把一些要处理比较长的消息,放在一个单独线程里面处理,把处理以后的结果,返回给主线程运行,就需要用的Handler来进行线程建的通信。

2 、Handler的作用

2.1　让线程延时执行

主要用到的两个方法：

- `final boolean postAtTime(Runnable r, long uptimeMillis)`
- `final boolean postDelayed(Runnable r, long delayMillis)`

2.2　让任务在其他线程中执行并返回结果

分为两个步骤：

- 在新启动的线程中发送消息

使用Handler对象的sendMessage()方法或者SendEmptyMessage()方法发送消息。

- 在主线程中获取处理消息

重写Handler类中处理消息的方法（void handleMessage(Message msg)），当新启动的线程发送消息时，消息发送到与之关联的MessageQueue。而Hanlder不断地从MessageQueue中获取并处理消息。

3 、Handler更新UI线程一般使用

- 首先要进行Handler 申明，复写handleMessage方法(放在主线程中)

```
private Handler handler = new Handler() {

        @Override
        public void handleMessage(Message msg) {
            // TODO 接收消息并且去更新UI线程上的控件内容
            if (msg.what == UPDATE) {
                // 更新界面上的textview
                tv.setText(String.valueOf(msg.obj));
            }
            super.handleMessage(msg);
        }
    };
```

- 子线程发送Message给ui线程表示自己任务已经执行完成，主线程可以做相应的操作了。

```
new Thread() {
            @Override
            public void run() {
                // TODO 子线程中通过handler发送消息给handler接收，由handler去更新TextView的值
                try {
                        //do something

                        Message msg = new Message();
                        msg.what = UPDATE;
                        msg.obj = "更新后的值" ;
                        handler.sendMessage(msg);
                    }
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        }.start();
```

4、Handler原理分析

4.1 Handler的构造函数

① public Handler()

② public Handler(Callbackcallback)

③ public Handler(Looperlooper)

④ public Handler(Looperlooper, Callbackcallback)

- 第①个和第②个构造函数都没有传递Looper，这两个构造函数都将通过调用Looper.myLooper()获取当前线程绑定的Looper对象，然后将该Looper对象保存到名为mLooper的成员字段中。 下面来看①②个函数源码：

```
113    public Handler() {
114        this(null, false);
115    }

127    public Handler(Callback callback) {
128        this(callback, false);
129    }

//他们会调用Handler的内部构造方法

188    public Handler(Callback callback, boolean async) {
189        if (FIND_POTENTIAL_LEAKS) {
190      final Class<? extends Handler> klass = getClass();
191      if ((klass.isAnonymousClass() ||klass.isMemberClass()
       || klass.isLocalClass()) &&
192                    (klass.getModifiers() & Modifier.STATIC) == 0) {
193                Log.w(TAG, "The following Handler class should be static or leaks might occur: " +
194                    klass.getCanonicalName());
195            }
196        }
197//**********************************
198        mLooper = Looper.myLooper();
199        if (mLooper == null) {
200            throw new RuntimeException(
201                "Can't create handler inside thread that has not called Looper.prepare()");
202        }
203        mQueue = mLooper.mQueue;
204        mCallback = callback;
205        mAsynchronous = async;
206    }
```

我们看到暗红色的重点部分：

通过Looper.myLooper()获取了当前线程保存的Looper实例，又通过这个Looper实例获取了其中保存的MessageQueue（消息队列）。**每个Handler 对应一个Looper对象，产生一个MessageQueue**

- 第③个和第④个构造函数传递了Looper对象，这两个构造函数会将该Looper保存到名为mLooper的成员字段中。 下面来看③④个函数源码：

```
136     public Handler(Looper looper) {
137         this(looper, null, false);
138     }

147     public Handler(Looper looper, Callback callback) {
148         this(looper, callback, false);
149     }
//他们会调用Handler的内部构造方法

227     public Handler(Looper looper, Callback callback, boolean async) {
228         mLooper = looper;
229         mQueue = looper.mQueue;
230         mCallback = callback;
231         mAsynchronous = async;
232     }
```

- 第②个和第④个构造函数还传递了Callback对象，Callback是Handler中的内部接口，需要实现其内部的handleMessage方法，Callback代码如下:

```
80      public interface Callback {
81          public boolean More ...handleMessage(Message msg);
82      }
```

Handler.Callback是用来处理Message的一种手段，如果没有传递该参数，那么就应该重写Handler的handleMessage方法，也就是说为了使得Handler能够处理Message，我们有两种办法：　　1. 向Hanlder的构造函数传入一个Handler.Callback对象，并实现Handler.Callback的handleMessage方法　　2. 无需向Hanlder的构造函数传入Handler.Callback对象，但是需要重写Handler本身的handleMessage方法　　也就是说无论哪种方式，我们都得通过某种方式实现handleMessage方法，这点与Java中对Thread的设计有异曲同工之处。

4.2　Handle发送消息的几个方法源码

```
    public final boolean sendMessage(Message msg)
     {
         return sendMessageDelayed(msg, 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) {
            RuntimeException e = new RuntimeException(
                    this + " sendMessageAtTime() called with no mQueue");
            Log.w("Looper", e.getMessage(), e);
            return false;
        }
        return enqueueMessage(queue, msg, uptimeMillis);
    }
```

我们可以看出他们最后都调用了sendMessageAtTime（），然后返回了enqueueMessage方法，下面看一下此方法源码：

```
626     private boolean enqueueMessage(MessageQueue queue, Message msg, long uptimeMillis) {
           //把当前的handler作为msg的target属性
627         msg.target = this;
628         if (mAsynchronous) {
629             msg.setAsynchronous(true);
630         }
631         return queue.enqueueMessage(msg, uptimeMillis);
632     }
```

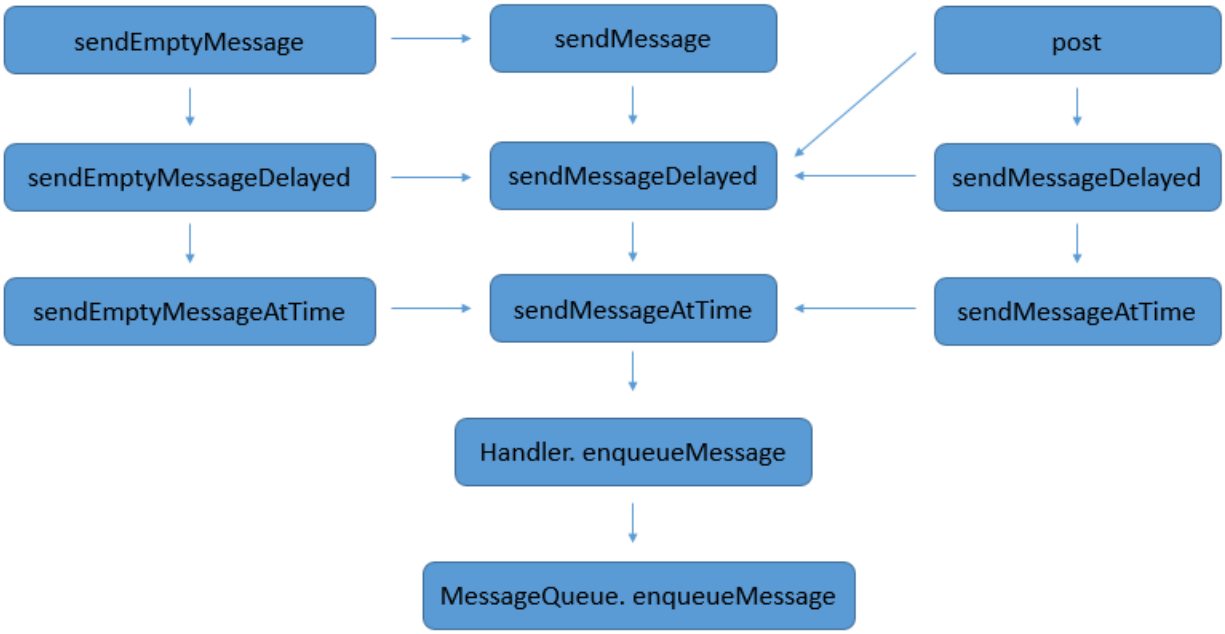
在该方法中有两件事需要注意：

1. msg.target = this

该代码将Message的target绑定为当前的Handler

1. queue.enqueueMessage 变量queue表示的是Handler所绑定的消息队列MessageQueue，通过调用queue.enqueueMessage(msg, uptimeMillis)我们将Message放入到消息队列中。

过下图可以看到完整的方法调用顺序：



5、Looper原理分析

我们一般在主线程申明Handler，有时我们需要继承Thread类实现自己的线程功能，当我们在里面申明Handler的时候会报错。其原因是主线程中已经实现了两个重要的Looper方法，下面看一看ActivityThread.java中main方法的源码：

```
public static void main(String[] args) {
            //......省略
5205        Looper.prepareMainLooper();//>
5206
5207        ActivityThread thread = new ActivityThread();
5208        thread.attach(false);
5209
5210        if (sMainThreadHandler == null) {
5211            sMainThreadHandler = thread.getHandler();
5212        }
5213
5214        AsyncTask.init();
5215
5216        if (false) {
5217            Looper.myLooper().setMessageLogging(new
5218    LogPrinter(Log.DEBUG, "ActivityThread"));
5219        }
5220
5221        Looper.loop();//>
5222
5223        throw new RuntimeException("Main thread loop unexpectedly exited");
5224    }
5225}
```

5.1 首先看prepare()方法

```
70      public static void prepare() {
71          prepare(true);
72      }
73
74      private static void prepare(boolean quitAllowed) {
          //证了一个线程中只有一个Looper实例
75          if (sThreadLocal.get() != null) {
76              throw new RuntimeException("Only one Looper may be created per thread");
77          }
78          sThreadLocal.set(new Looper(quitAllowed));
79      }
```

该方法会调用Looper构造函数同时实例化出MessageQueue和当前thread.

```
186     private Looper(boolean quitAllowed) {
187         mQueue = new MessageQueue(quitAllowed);
188         mThread = Thread.currentThread();
189     }

182     public static MessageQueue myQueue() {
183         return myLooper().mQueue;
184     }
```

prepare()方法中通过ThreadLocal对象实现Looper实例与线程的绑定。（不清楚的可以查看 ThreadLocal的使用规则和源码分析）

5.2 loop()方法

```
109     public static void loop() {
110         final Looper me = myLooper();
111         if (me == null) {
112             throw new RuntimeException("No Looper; Looper.prepare() wasn't called on this thread."
113         }
114         final MessageQueue queue = me.mQueue;
115
118         Binder.clearCallingIdentity();
119         final long ident = Binder.clearCallingIdentity();
120
121         for (;;) {
122             Message msg = queue.next(); // might block
123             if (msg == null) {
124
125                 return;
126             }
127
129             Printer logging = me.mLogging;
130             if (logging != null) {
131                 logging.println(">>>>> Dispatching to " + msg.target + " " +
132                         msg.callback + ": " + msg.what);
133             }
//重点****
135             msg.target.dispatchMessage(msg);
136
137             if (logging != null) {
138                 logging.println("<<<<< Finished to " + msg.target + " " + msg.callback);
139             }
140
142             // identity of the thread wasn't corrupted.
143             final long newIdent = Binder.clearCallingIdentity();
144             if (ident != newIdent) {
145                 Log.wtf(TAG, "Thread identity changed from 0x"
146                         + Long.toHexString(ident) + " to 0x"
147                         + Long.toHexString(newIdent) + " while dispatching to "
148                         + msg.target.getClass().getName() + " "
149                         + msg.callback + " what=" + msg.what);
150             }
151
152             msg.recycleUnchecked();
153         }
154     }
```

首先looper对象不能为空，就是说loop()方法调用必须在prepare()方法的后面。

Looper一直在不断的从消息队列中通过MessageQueue的next方法获取Message，然后通过代码msg.target.dispatchMessage(msg)让该msg所绑定的Handler（Message.target）执行dispatchMessage方法以实现对Message的处理。

Handler的dispatchMessage的源码如下：

```
93      public void dispatchMessage(Message msg) {
94          if (msg.callback != null) {
95              handleCallback(msg);
96          } else {
97              if (mCallback != null) {
98                  if (mCallback.handleMessage(msg)) {
99                      return;
100                 }
101             }
102             handleMessage(msg);
103         }
104     }
```

我们可以看到Handler提供了三种途径处理Message，而且处理有前后优先级之分：首先尝试让postXXX中传递的Runnable执行，其次尝试让Handler构造函数中传入的Callback的handleMessage方法处理，最后才是让Handler自身的handleMessage方法处理Message。

6、如何在子线程中使用Handler

Handler本质是从当前的线程中获取到Looper来监听和操作MessageQueue，当其他线程执行完成后回调当前线程。

子线程需要先prepare（）才能获取到Looper的，是因为在子线程只是一个普通的线程，其ThreadLoacl中没有设置过Looper，所以会抛出异常，而在Looper的prepare（）方法中sThreadLocal.set(new Looper())是设置了Looper的。

6.1 实例代码

定义一个类实现Runnable接口或继承Thread类（一般不继承）。

```
class Rub implements Runnable {

    public Handler myHandler;
    // 实现Runnable接口的线程体
    @Override
    public void run() {

      /*①、调用Looper的prepare()方法为当前线程创建Looper对象并，
       创建Looper对象时，它的构造器会自动的创建相对应的MessageQueue*/
        Looper.prepare();

        /*.②、创建Handler子类的实例，重写HandleMessage()方法，该方法处理除当前线程以外线程的消息*/
         myHandler = new Handler() {
            @Override
            public void handleMessage(Message msg) {
                String ms = "";
                if (msg.what == 0x777) {

                }
            }

        };
        //③、调用Looper的loop()方法来启动Looper让消息队列转动起来
        Looper.loop();
    }
}
```

注意分成三步：

1．调用Looper的prepare()方法为当前线程创建Looper对象，创建Looper对象时，它的构造器会创建与之配套的MessageQueue。

2．有了Looper之后，创建Handler子类实例，重写HanderMessage()方法，该方法负责处理来自于其他线程的消息。

3．调用Looper的looper()方法启动Looper。

然后使用这个handler实例在任何其他线程中发送消息，最终处理消息的代码都会在你创建Handler实例的线程中运行。

7、总结

Handler:

- 发送消息，它能把消息发送给Looper管理的MessageQueue。
- 处理消息，并负责处理Looper分给它的消息。
- Handler的构造方法，会首先得到当前线程中保存的Looper实例，进而与Looper实例中的MessageQueue想关联。
- Handler的sendMessage方法，会给msg的target赋值为handler自身，然后加入MessageQueue中。

Looper:

- 每个线程只有一个Looper，它负责管理对应的MessageQueue，会不断地从MessageQueue取出消息，并将消息分给对应的Hanlder处理。
- 主线程中，系统已经初始化了一个Looper对象，因此可以直接创建Handler即可，就可以通过Handler来发送消息、处理消息。程序自己启动的子线程，程序必须自己创建一个Looper对象，并启动它，调用 `Looper.prepare()` 方法。
- prapare()方法：保证每个线程最多只有一个Looper对象。
- looper()方法：启动Looper，使用一个死循环不断取出MessageQueue中的消息，并将取出的消息分给对应的Handler进行处理。

MessageQueue:

- 由Looper负责管理，它采用先进先出的方式来管理Message。

Message:

- Handler接收和处理的消息对象。