作為一個大三的預備程序員，我學習 android 的一大樂趣是可以通過源碼學習 google 大牛們的設計思想。android 源碼中包含了大量的設計模式，除此以外，android sdk 還精心為我們設計了各種 helper 類，對於和我一樣渴望水平得到進階的人來說，都太值得一讀了。這不，前幾天為了瞭解 android 的消息處理機制，我看了 **Looper，Handler，Message** 這幾個類的源碼，結果又一次被 googler 的設計震撼了，特與大家分享。

android 的消息處理有三個核心類：Looper,Handler 和 Message。其實還有一個 Message Queue（消息隊列），但是 MQ 被封裝到 Looper 裡面了，我們不會直接與 MQ 打交道，因此我沒將其作為核心類。下面一一介紹：

線程的魔法師 Looper

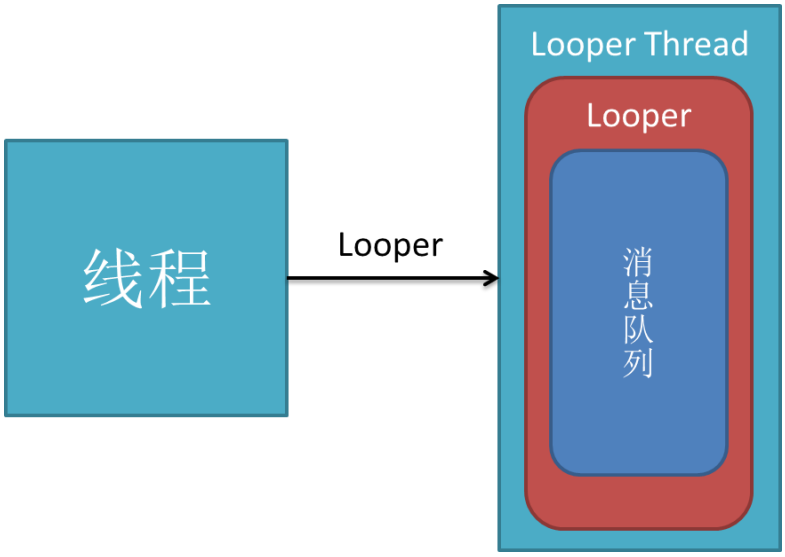
Looper 的字面意思是「循環者」,它被設計用來使一個普通線程變成 **Looper 線程**。所謂 Looper 線程就是循環工作的線程。在程序開發中（尤其是 GUI 開發中），我們經常會需要一個線程不斷循環，一旦有新任務則執行，執行完繼續等待下一個任務，這就是 Looper 線程。使用 Looper 類創建 Looper 線程很簡單：

⊞View Code

```
public class LooperThread extends Thread {
    @Override
    public void run() {
        // 將當前線程初始化為 Looper 線程
        Looper.prepare();
        // ...其他處理，如實例化 handle
        // 開始循環處理消息隊列
        Looper.loop();
    }
}
```

通過上面兩行核心代碼，你的線程就升級為 Looper 線程了！！！是不是很神奇？讓我們放慢鏡頭，看看這兩行代碼各自做了什麼。

1)Looper.prepare()



通過上圖可以看到，現在你的線程中有一個 Looper 對象，它的內部維護了一個消息隊列 MQ。

注意，**一個 Thread 只能有一個 Looper 對象**，為什麼呢？咱們來看源碼。

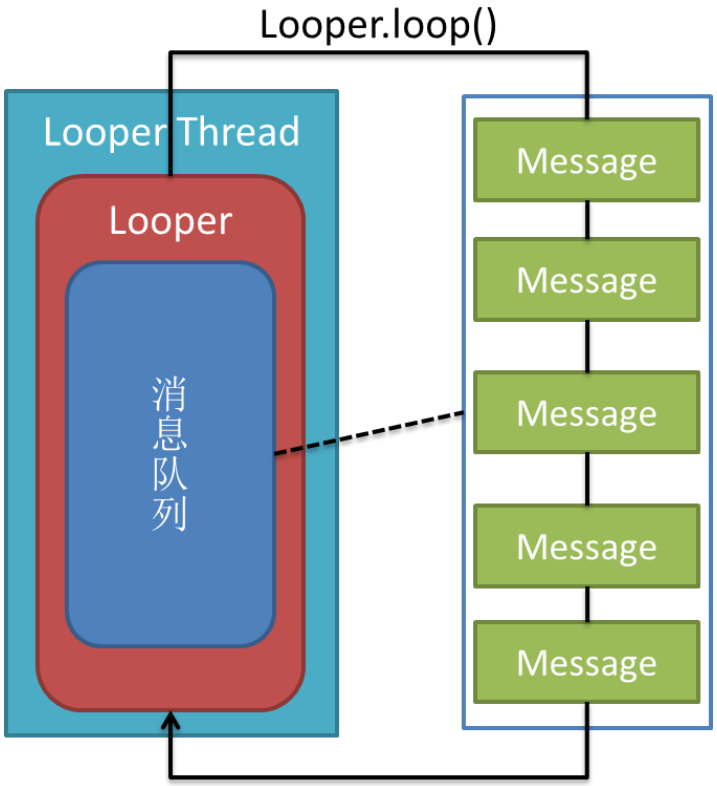
View Code

```
public class Looper {

    // 每個線程中的 Looper 對象其實是一個 ThreadLocal，即線程本地存儲(TLS)對象

    private static final ThreadLocal sThreadLocal = new ThreadLocal();

    // Looper 內的消息隊列

    final MessageQueue mQueue;

    // 當前線程

    Thread mThread;
    // 。。。其他屬性


    // 每個 Looper 對象中有它的消息隊列，和它所屬的線程

    private Looper() {

        mQueue = new MessageQueue();
```

```
        mRun = true;

        mThread = Thread.currentThread();

    }

    // 我們調用該方法會在調用線程的 TLS 中創建 Looper 對象

    public static final void prepare() {

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

            // 試圖在有 Looper 的線程中再次創建 Looper 將拋出異常

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

        }
        sThreadLocal.set(new Looper());

    }
    // 其他方法

}
```

通過源碼，prepare()背後的工作方式一目瞭然，其核心就是將 looper 對象定義為 ThreadLocal。如果你還不清楚什麼是 ThreadLocal，請參考《理解 ThreadLocal》。

2）Looper.loop()



調用 loop 方法後,Looper 線程就開始真正工作了,它不斷從自己的 MQ 中取出隊頭的消息(也叫任務)執行。其源碼分析如下：

⊞View Code

```
public static final void loop() {

        Looper me = myLooper();  //得到當前線程 Looper

        MessageQueue queue = me.mQueue;  //得到當前 looper 的 MQ


        // 這兩行沒看懂= = 不過不影響理解

        Binder.clearCallingIdentity();
        final long ident = Binder.clearCallingIdentity();

        // 開始循環

        while (true) {
```

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

        if (msg != null) {

            if (msg.target == null) {

                // message 沒有 target 為結束信號，退出循環

                return;

            }
            // 日誌。。。

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

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

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

                    );
            // 非常重要！將真正的處理工作交給 message 的 target，即後面要講的 handler

            msg.target.dispatchMessage(msg);
            // 還是日誌。。。

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

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

                    + msg.callback);


            // 下面沒看懂，同樣不影響理解

            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 資源

                msg.recycle();
            }

        }
    }
```

除了 prepare()和 loop()方法，Looper 類還提供了一些有用的方法，比如 Looper.myLooper()得到當前線程 looper 對象：

⊞View Code

```
    public static final Looper myLooper() {

        // 在任意線程調用 Looper.myLooper()返回的都是那個線程的 looper

        return (Looper)sThreadLocal.get();

    }
```

getThread()得到 looper 對象所屬線程：

⊞View Code

```
    public Thread getThread() {

        return mThread;

    }
```

quit()方法結束 looper 循環：

⊞View Code

```
    public void quit() {

        // 創建一個空的 message，它的 target 為 NULL，表示結束循環消息

        Message msg = Message.obtain();

        // 發出消息

        mQueue.enqueueMessage(msg, 0);

    }
```

到此為止，你應該對 Looper 有了基本的瞭解，總結幾點：

1.每個線程有且最多只能有一個 Looper 對象，它是一個 ThreadLocal

2.Looper 內部有一個消息隊列，loop()方法調用後線程開始不斷從隊列中取出消息執行

3.Looper 使一個線程變成 Looper 線程。

那麼，我們如何往 MQ 上添加消息呢？下面有請 Handler！（掌聲~~~）

異步處理大師 Handler

什麼是 handler？handler 扮演了往 MQ 上添加消息和處理消息的角色(只處理由自己發出的消息），即**通知 MQ 它要執行一個任務(sendMessage)，並在 loop 到自己的時候執行該任務(handleMessage)，整個過程是異步的**。handler 創建時會關聯一個 looper，默認的構造方法將關聯當前線程的 looper，不過這也是可以 set 的。默認的構造方法：

⊞View Code

```
public class handler {


    final MessageQueue mQueue;  // 關聯的 MQ

    final Looper mLooper;  // 關聯的 looper

    final Callback mCallback;

    // 其他屬性
```

```
    public Handler() {
        // 沒看懂，直接略過，，，
        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());
            }
        }
        // 默認將關聯當前線程的 looper
        mLooper = Looper.myLooper();
        // looper 不能為空，即該默認的構造方法只能在 looper 線程中使用
        if (mLooper == null) {
            throw new RuntimeException(
                "Can't create handler inside thread that has not called Looper.prepare()");
        }
        // 重要！！！直接把關聯 looper 的 MQ 作為自己的 MQ，因此它的消息將發送到關聯 looper 的 MQ
上
```

```
        mQueue = mLooper.mQueue;

        mCallback = null;

    }

    // 其他方法

}
```

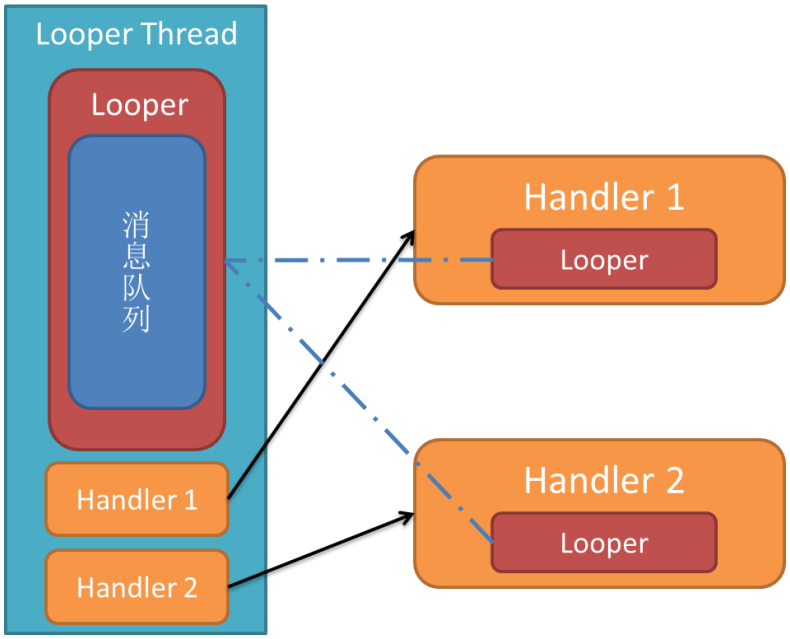
下面我們就可以為之前的 LooperThread 類加入 Handler：

⊞View Code

```
public class LooperThread extends Thread {

    private Handler handler1;

    private Handler handler2;


    @Override
    public void run() {

        // 將當前線程初始化為 Looper 線程

        Looper.prepare();


        // 實例化兩個 handler

        handler1 = new Handler();

        handler2 = new Handler();


        // 開始循環處理消息隊列
```

```
        Looper.loop();
    }

}
```

加入 handler 後的效果如下圖：



可以看到，**一個線程可以有多個 Handler，但是只能有一個 Looper！**

Handler 發送消息

有了 handler 之後，我們就可以使用 post(Runnable), postAtTime(Runnable, long), postDelayed(Runnable, long), sendEmptyMessage(int), sendMessage(Message), sendMessageAtTime(Message, long)和 sendMessageDelayed(Message, long)這些方法向 MQ 上發送消息了。光看這些 API 你可能會覺得 handler 能發兩種消息，一種是 Runnable 對象，一種是 message 對象，這是直觀的理解，但其實 post 發出的 Runnable 對象最後都被封裝成 message 對象了，見源碼：

⊞View Code

```
    // 此方法用於向關聯的 MQ 上發送 Runnable 對象，它的 run 方法將在 handler 關聯的 looper 線程中執行

    public final boolean post(Runnable r)

    {
       // 注意 getPostMessage(r)將 runnable 封裝成 message
```

```
   return  sendMessageDelayed(getPostMessage(r), 0);

}

private final Message getPostMessage(Runnable r) {

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

    m.callback = r;  //將 runnable 設為 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 必須設為該 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;

    }
```

其他方法就不羅列了，總之通過 handler 發出的 message 有如下特點：

1. message.target 為該 handler 對象，這確保了 looper 執行到該 message 時能找到處理它的 handler，即 loop()方法中的關鍵代碼

```
msg.target.dispatchMessage(msg);
```

2. post 發出的 message，其 callback 為 Runnable 對象

Handler 處理消息

說完了消息的發送，再來看下 handler 如何處理消息。消息的處理是通過核心方法 dispatchMessage(Message msg)與鉤子方法 handleMessage(Message msg)完成的，見源碼

⊞View Code

```
    // 處理消息，該方法由 looper 調用

    public void dispatchMessage(Message msg) {

        if (msg.callback != null) {

            // 如果 message 設置了 callback，即 runnable 消息，處理 callback！

            handleCallback(msg);
        } else {

            // 如果 handler 本身設置了 callback，則執行 callback

            if (mCallback != null) {

                /* 這種方法允許讓 activity 等來實現 Handler.Callback 接口，避免了自己編寫 handler 重寫 handleMessage 方法。見 http://alex-yang-xiansoftware-com.iteye.com/blog/850865 */

                if (mCallback.handleMessage(msg)) {

                    return;

                }
            }
```

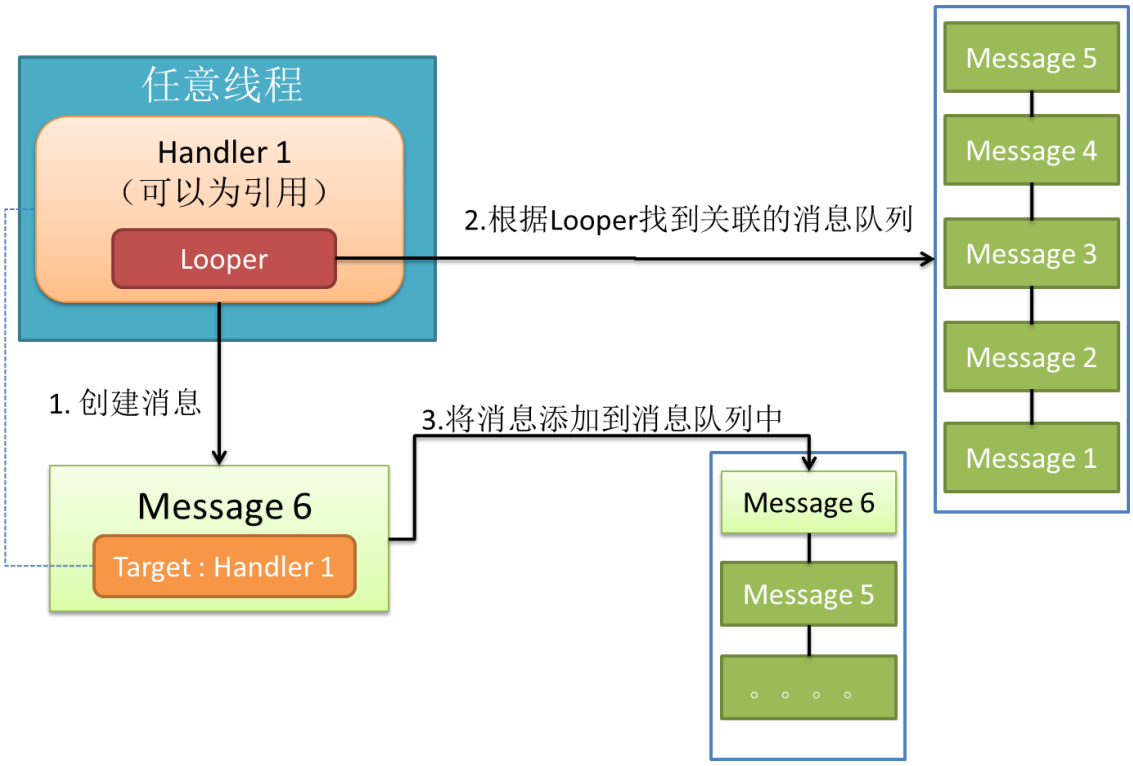
```
            // 如果 message 沒有 callback，則調用 handler 的鉤子方法 handleMessage

            handleMessage(msg);
        }

    }

    // 處理 runnable 消息

    private final void handleCallback(Message message) {

        message.callback.run();  //直接調用 run 方法！

    }
    // 由子類實現的鉤子方法

    public void handleMessage(Message msg) {

    }
```

可以看到，除了 handleMessage(Message msg)和 Runnable 對象的 run 方法由開發者實現外（實現具體邏輯），handler 的內部工作機制對開發者是透明的。這正是 handler API 設計的精妙之處！

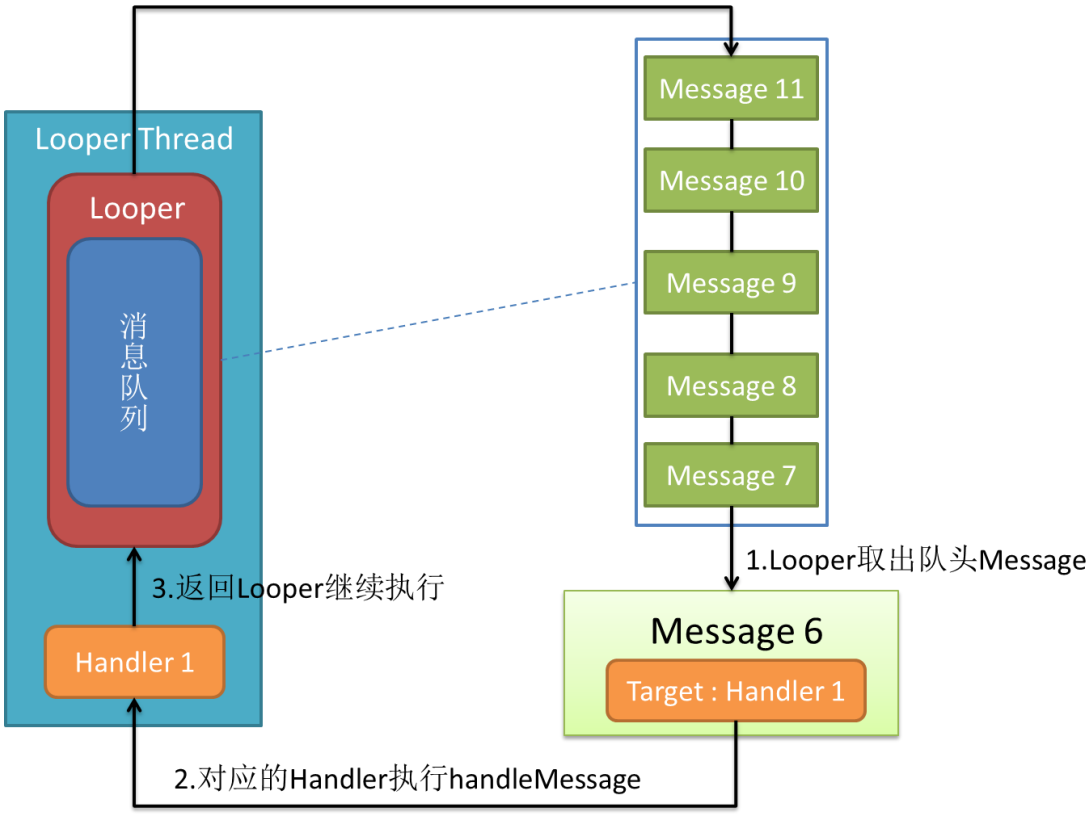
Handler 的用處

我在小標題中將 handler 描述為「異步處理大師」，這歸功於 Handler 擁有下面兩個重要的特點：

1. handler 可以在**任意線程發送消息**，這些消息會被添加到關聯的 MQ 上。

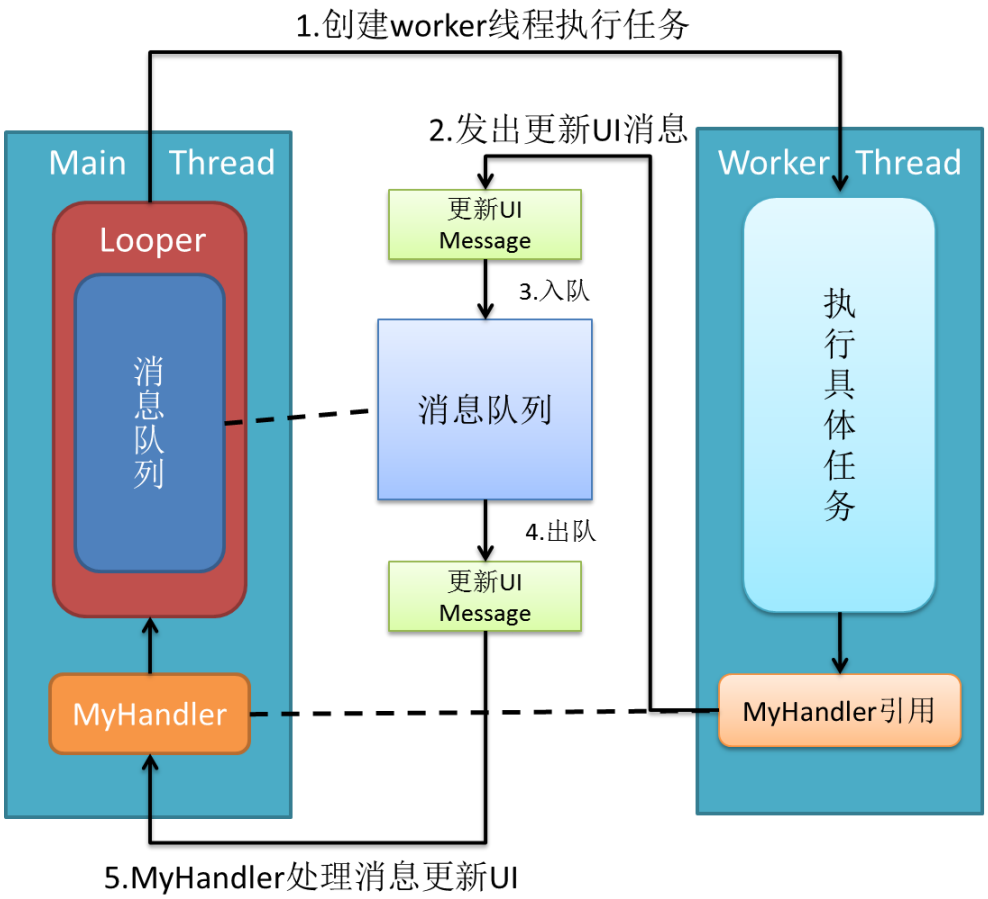


2. handler 是在它**關聯的 looper 線程中處理消息**的。



這就解決了 android 最經典的不能在其他非主線程中更新 UI 的問題。**android 的主線程也是一個 looper 線程**(looper 在 android 中運用很廣)，我們在其中創建的 handler 默認將關聯主線程 MQ。因此，利用 handler 的一個 solution 就是在 activity 中創建 handler 並將

其引用傳遞給 worker thread，worker thread 執行完任務後使用 handler 發送消息通知 activity 更新 UI。(過程如圖)



下面給出 sample 代碼，僅供參考：

⊞View Code

```
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);
```

```
        // 創建並啟動工作線程

        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);
        }

    }
}
```

View Code

```
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 {  // 模擬執行某項任務，下載等
            Thread.sleep(5000);
            // 任務完成後通知 activity 更新 UI
            Message msg = prepareMessage("task completed!");
            // message 將被添加到主線程的 MQ 中
            handler.sendMessage(msg);
        } catch (InterruptedException e) {
            Log.d(TAG, "interrupted!");
```