1. 基本概念
2. 工作原理
3. 使用方式
4. 源码分析：Handler.sendMessage（）方式
5. 源码分析：Handler.post（）方式
6. 解决内存泄漏

基本概念

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。所以说 Handler机制其实是一套 Android 消息传递机制 / 异步通信机制

在多线程的应用场景中，将工作线程中需更新UI的操作信息 传递到 UI主线程，从而实现 工作线程对UI的更新处理，最终实现异步消息的处理

Handler 机制中的相关概念如下：

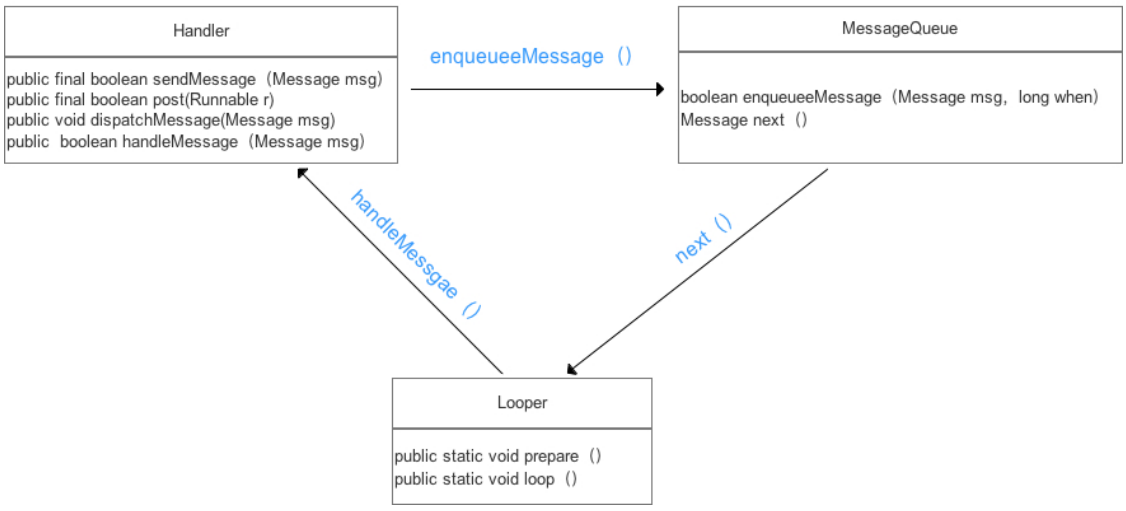
概念	定义	作用	备注
主线程 (UI线程、Main Thread)	当应用程序第1次启动时，会同时自动开启1条主线程	处理与 UI 相关的事件 (如更新、操作等)	主线程 与 子线程 通信媒介= Handler
子线程 (工作线程)	人为手动开启的线程	执行耗时操作 (如 网络请求、数据加载等)	
消息 (Message)	线程间通讯的数据单元 (即 Handler接受 & 处理的消息对象)	存储需操作的通信信息	/
消息队列 (Message Queue)	一种 数据结构 (存储特点：先进先出)	存储 Handler发送过来的消息（Message）	/
处理者 (Handler)	• 主线程 与 子线程 的通信媒介 • 线程消息的主要处理者	• 添加 消息（Message） 到消息队列（Message Queue） • 处理循环器（Looper）分派过来的消息（Message）	/
循环器 (Looper)	消息队列（Message Queue） 与 处理者（Handler） 的通信媒介	消息循环，即 • 消息获取：循环取出消息队列（Message Queue）的消息（Message） • 消息分发：将取出的消息（Message） 发送给 对应的处理者（Handler）	• 每个线程中只能拥有1个Looper • 1个Looper可 绑定 多个线程的Handler • 即 多个线程可往1个Looper所持有的MessageQueue中发送消息，提供了线程间通信的可能

Handler机制 中有3个重要的类：

- 处理器 类（Handler）
- 消息队列 类（MessageQueue）
- 循环器 类（Looper）

类名	核心方法	作用
处理器 类 (Handler)	sendMessage（Message msg）	将消息 发送 到消息队列中 (Message –>> MessageQueue)
	post（Runnable r）	
	dispatchMessage（Message msg）	将消息分发给对应的Handler
	handleMessage（Message msg）	根据某个消息进行相关处理& 操作 (如 UI操作)
消息队列 类 (MessageQueue)	enqueueeMessage(Message msg,long when)	入队：将消息 根据时间 放入到消息队列中
	Message next （）	出队：即 从 消息队列中 移出该消息
循环器 类 (Looper)	prepare （）	创建1个循环器对象（Looper） & 消息队列对象(MessageQueue) (属 当前线程)
	loop （）	消息循环 • 即 循环从消息队列中获取消息 & 发送给Handler • 无限循环 & 无消息时则阻塞

三者之间的关系



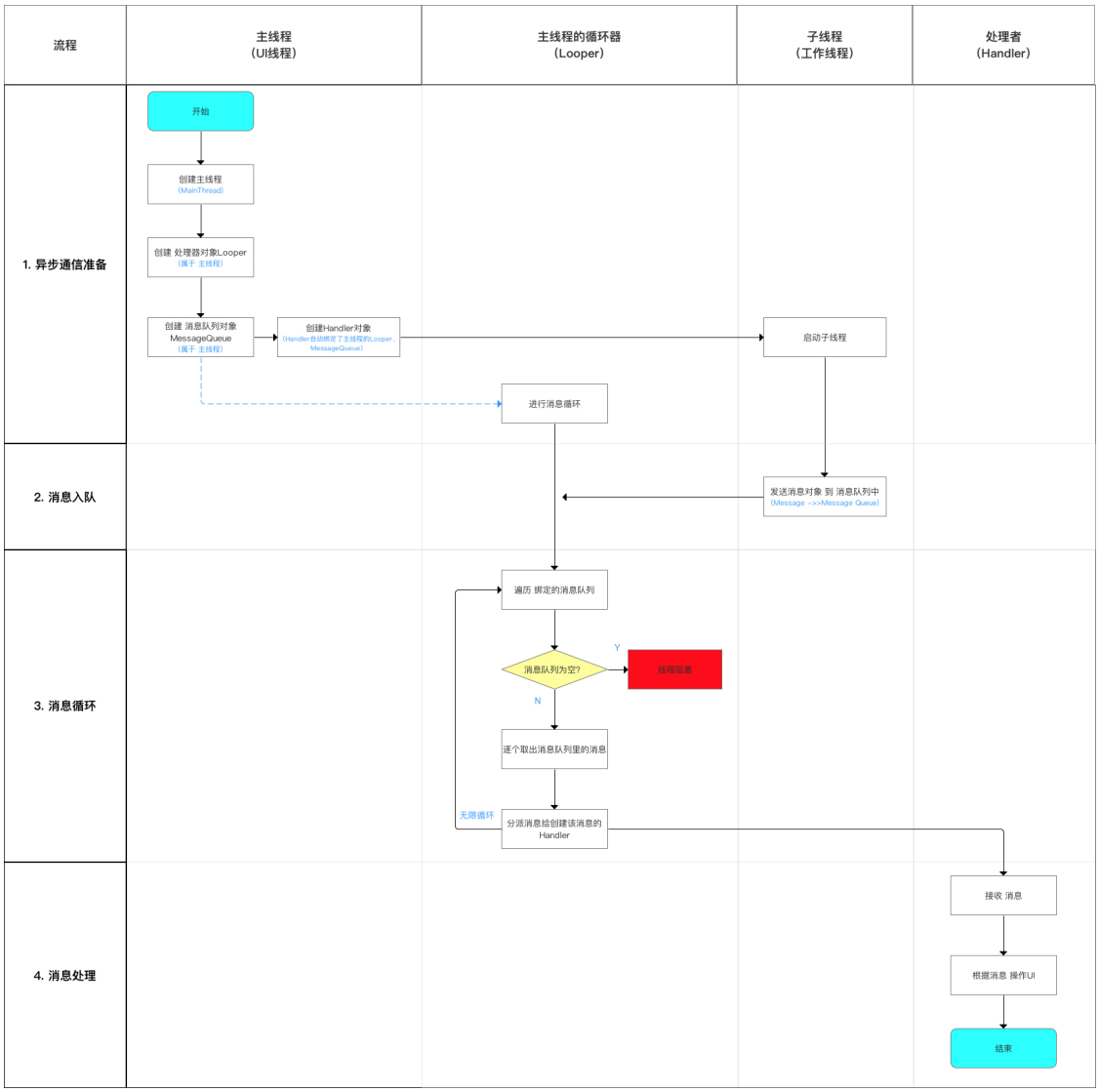
工作原理

Handler机制的工作流程主要包括4个步骤：

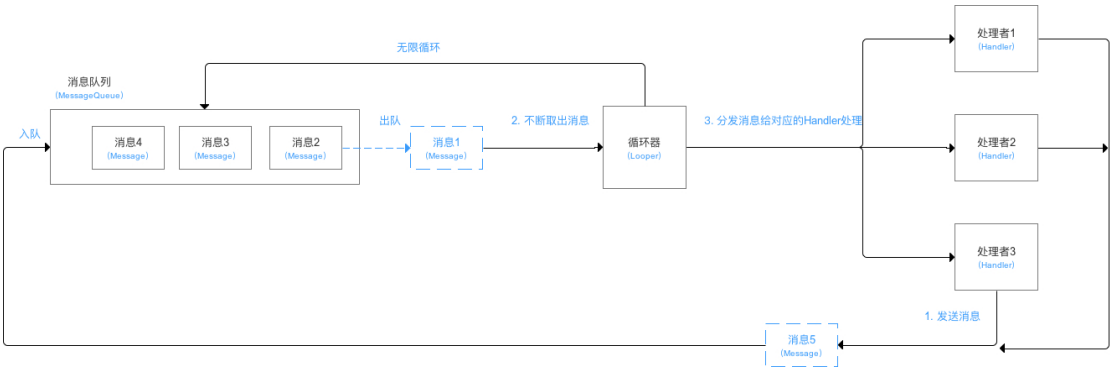
- 1.异步通信准备
- 2.消息发送
- 3.消息循环
- 4.消息处理

步骤	具体描述	备注
1. 异步通信准备	在主线程中创建： • 处理器 对象（Looper） • 消息队列 对象（Message Queue） • Handler 对象	• Looper、Message Queue均属于主线程 • 创建Message Queue后，Looper则自动进入消息循环 • 此时，Handler自动绑定了主线程的Looper、Message Queue
2. 消息入队	工作线程 通过 Handler 发送消息（Message）到消息队列（Message Queue）中	该消息内容 = 工作线程对UI的操作
3. 消息循环	• 消息出队：Looper循环取出 消息队列（Message Queue） 中的消息（Message） • 消息分发：Looper将取出的消息（Message） 发送给 创建该消息的处理者(Handler)	在消息循环过程中，若消息队列为空，则线程阻塞
4. 消息处理	• 处理者(Handler）接收 处理器（Looper）发送过来的消息（Message） • 处理者(Handler）根据消息（Message）进行UI操作	

流程图

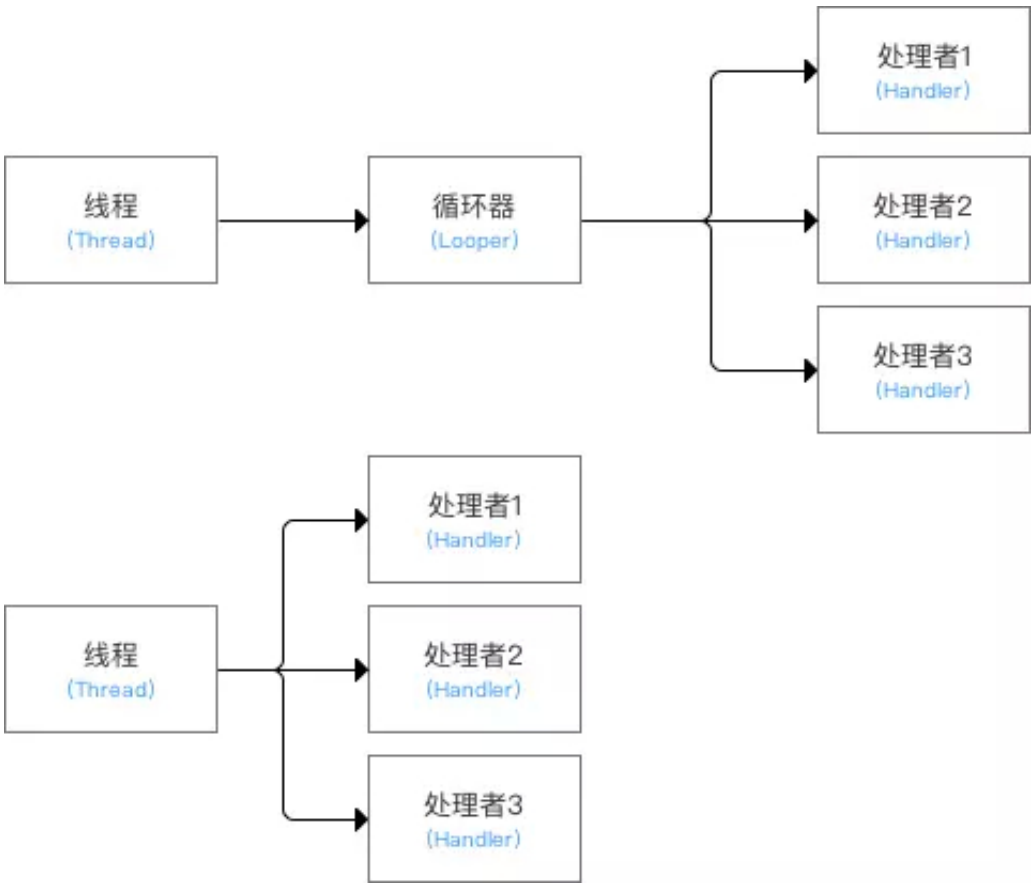


示意图



特别注意： 线程（Thread）、循环器（Looper）、处理者（Handler）之间的对应关系如下：

- 1个线程（Thread）只能绑定1个循环器（Looper）
- 1个线程（Thread）或 1个循环器（Looper） 可绑定多个处理者（Handler）
- 1个处理者（Handler） 只能绑定1个循环器（Looper）



使用方式

Handler使用方式 因发送消息到消息队列的方式不同共分为2种

- Handler.sendMessage（）
- Handler.post（）

方式1：使用 Handler.sendMessage（）

> 在该使用方式中，又分为2种：新建Handler子类（内部类）、匿名 Handler子类
>
> 但本质相同，即 继承了Handler类 & 创建了子类

```
/**
  * 方式1：新建Handler子类（内部类）
  */

// 步骤1：自定义Handler子类（继承Handler类） & 复写handleMessage（）方法
class mHandler extends Handler {

    // 通过复写handlerMessage() 从而确定更新UI的操作
    @Override
    public void handleMessage(Message msg) {
     ...// 需执行的UI操作

    }
}

// 步骤2：在主线程中创建Handler实例
private Handler mhandler = new mHandler();

// 步骤3：创建所需的消息对象
Message msg = Message.obtain(); // 实例化消息对象
msg.what = 1; // 消息标识
msg.obj = "AA"; // 消息内容存放

// 步骤4：在工作线程中 通过Handler发送消息到消息队列中
// 多线程可采用AsyncTask、继承Thread类、实现Runnable
mHandler.sendMessage(msg);

// 步骤5：开启工作线程（同时启动了Handler）
// 多线程可采用AsyncTask、继承Thread类、实现Runnable


/**
  * 方式2：匿名内部类
  */
// 步骤1：在主线程中 通过匿名内部类 创建Handler类对象
private Handler mhandler = new  Handler(){

    // 通过复写handlerMessage()从而确定更新UI的操作
    @Override
    public void handleMessage(Message msg) {
            ...// 需执行的UI操作
        }
};

// 步骤2：创建消息对象
Message msg = Message.obtain(); // 实例化消息对象
msg.what = 1; // 消息标识
```

```
msg.obj = "AA"; // 消息内容存放

// 步骤3：在工作线程中 通过Handler发送消息到消息队列中
// 多线程可采用AsyncTask、继承Thread类、实现Runnable
mHandler.sendMessage(msg);

// 步骤4：开启工作线程（同时启动了Handler）
// 多线程可采用AsyncTask、继承Thread类、实现Runnable
```

方式2：使用Handler.post（）

```
// 步骤1：在主线程中创建Handler实例
private Handler mhandler = new mHandler();

// 步骤2：在工作线程中 发送消息到消息队列中 & 指定操作UI内容
// 需传入1个Runnable对象
mHandler.post(new Runnable() {
        @Override
        public void run() {
            ... // 需执行的UI操作
        }

});

// 步骤3：开启工作线程（同时启动了Handler）
// 多线程可采用AsyncTask、继承Thread类、实现Runnable
```

源码分析：Handler.sendMessage（）方式

```
// 此处以 匿名内部类 的方式为例
private Handler mhandler = new  Handler(){
    @Override
    public void handleMessage(Message msg) {

    }
};

Message msg = Message.obtain();
msg.what = 1;
msg.obj = "AA";
mHandler.sendMessage(msg);
```

1.在主线程中 通过匿名内部类 创建Handler类对象

```
private Handler mhandler = new  Handler(){
    @Override
    public void handleMessage(Message msg) {

    }
};

/**
 * 源码分析：Handler的构造方法
 * 作用：初始化Handler对象 & 绑定线程
 * 注：
 *   a. Handler需绑定 线程才能使用；绑定后，Handler的消息处理会在绑定的线程中执行
 *   b. 绑定方式 = 先指定Looper对象，从而绑定了 Looper对象所绑定的线程（因为Looper对象本已绑定了对应线程）
 *   c. 即：指定了Handler对象的 Looper对象 = 绑定到了Looper对象所在的线程
 */
public Handler() {

    this(null, false);
    // ->>分析1
}

/**
  * 分析1：this(null, false) = Handler（null, false）
  */
public Handler(Callback callback, boolean async) {

    ...

    // 1. 指定Looper对象
    mLooper = Looper.myLooper();
    if (mLooper == null) {
        throw new RuntimeException("Can't create handler inside thread that has not called Looper.prepare()");
    }
    // Looper.myLooper()作用：获取当前线程的Looper对象；若线程无Looper对象则抛出异常
    // 即 ：若线程中无创建Looper对象，则也无法创建Handler对象
    // 故 若需在子线程中创建Handler对象，则需先创建Looper对象
    // 注：可通过Loop.getMainLooper()可以获得当前进程的主线程的Looper对象

    // 2. 绑定消息队列对象（MessageQueue）
    mQueue = mLooper.mQueue;
    // 获取该Looper对象中保存的消息队列对象（MessageQueue）
```

```
    // 至此，保证了handler对象 关联上 Looper对象中MessageQueue
}
```

从上面可看出：

当创建Handler对象时，则通过 构造方法 自动关联当前线程的Looper对象 & 对应的消息队列对象（MessageQueue），从而 自动绑定了 创建Handler对象的操作线程

在上述使用步骤中，并无 创建Looper对象 & 对应的消息队列对象（MessageQueue）这1步，那么当前线程的Looper对象 & 对应的消息队列对象（MessageQueue） 是什么时候创建的呢？

2.创建循环器对象（Looper） & 消息队列对象（MessageQueue）

- 创建Looper对象主要通过方法：Looper.prepareMainLooper()、Looper.prepare ()
- 创建消息队列对象（MessageQueue）方法：创建Looper对象时则会自动创建
（即：创建循环器对象（Looper）的同时，会自动创建消息队列对象（MessageQueue））

核心方法	作用	备注
Looper.prepareMainLooper()	为 主线程（UI线程） 创建1个循环器对象（Looper）	同时也会创建1个对应的消息队列对象（MessageQueue）
Looper.prepare ()	为当前线程（子线程） 创建1个循环器对象（Looper）	

```
/**
  * 源码分析1：Looper.prepare()
  * 作用：为当前线程（子线程） 创建1个循环器对象（Looper），同时也生成了1个消息队列对象（MessageQueue）
  * 注：需在子线程中手动调用该方法
  */
public static final void prepare() {

    if (sThreadLocal.get() != null) {
        throw new RuntimeException("Only one Looper may be created per thread");
    }
    // 1. 判断sThreadLocal是否为null，否则抛出异常
    //即 Looper.prepare()方法不能被调用两次 = 1个线程中只能对应1个Looper实例
    // 注：sThreadLocal = 1个ThreadLocal对象，用于存储线程的变量

    sThreadLocal.set(new Looper(true));
    // 2. 若为初次Looper.prepare()，则创建Looper对象 & 存放在ThreadLocal变量中
```

```
    // 注：Looper对象是存放在Thread线程里的
    // 源码分析Looper的构造方法->>分析a
}


// 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；不同线程对应不同的值。

//创建
ThreadLocal<String> mStringThreadLocal = new ThreadLocal<>();
//set方法
mStringThreadLocal.set("helloworld");
//get方法
mStringThreadLocal.get();


/**
 * 分析a：Looper的构造方法
 **/
private Looper(boolean quitAllowed) {

    mQueue = new MessageQueue(quitAllowed);
    // 1. 创建1个消息队列对象（MessageQueue）
    // 即 当创建1个Looper实例时，会自动创建一个与之配对的消息队列对象（MessageQueue）

    mThread = Thread.currentThread();
}

/**
  * 源码分析2：Looper.prepareMainLooper()
  * 作用：为 主线程（UI线程） 创建1个循环器对象（Looper），同时也生成了1个消息队列对象（MessageQueue）
  * 注：该方法在主线程（UI线程）创建时自动调用，即 主线程的Looper对象自动生成，不需手动生成
  */
// 在Android应用进程启动时，会默认创建1个主线程（ActivityThread，也叫UI线程）
// 创建时，会自动调用ActivityThread的1个静态的main（）方法 = 应用程序的入口
// main（）内则会调用Looper.prepareMainLooper()为主线程生成1个Looper对象

/**
 * 源码分析：main（）
 **/
public static void main(String[] args) {
    ...

    Looper.prepareMainLooper();
```

```
    // 1. 为主线程创建1个Looper对象，同时生成1个消息队列对象
 (MessageQueue)
    // 方法逻辑类似Looper.prepare()
    // 注：prepare()：为子线程中创建1个Looper对象


    ActivityThread thread = new ActivityThread();
    // 2. 创建主线程

    Looper.loop();
    // 3. 自动开启 消息循环 ->>下面将详细分析
}

public static void prepareMainLooper() {
    prepare(false);
    ...
}
```

创建主线程时，会自动调用ActivityThread的1个静态的main（）；而main（）内则会调用Looper.prepareMainLooper()为主线程生成1个Looper对象，同时也会生成其对应的MessageQueue对象

- 主线程的Looper对象自动生成，不需手动生成；而子线程的Looper对象则需手动通过Looper.prepare()创建
- 在子线程若不手动创建Looper对象 则无法生成Handler对象
- 根据Handler的作用（在主线程更新UI），故Handler实例的创建场景主要在 主线程

3.生成Looper & MessageQueue对象后，则会自动进入消息循环：Looper.loop（）

```
/**
  * 源码分析： Looper.loop()
  * 作用：消息循环，即从消息队列中获取消息、分发消息到Handler
  * 特别注意：
  *       a. 主线程的消息循环不允许退出，即无限循环
  *       b. 子线程的消息循环允许退出：调用消息队列MessageQueue的quit ()
  */
public static void loop() {

    ...

    // 1. 获取当前Looper的消息队列
    final Looper me = myLooper();
```

```
    if (me == null) {
        throw new RuntimeException("No Looper; Looper.prepare()
wasn't called on this thread.");
    }
    // myLooper()作用：返回sThreadLocal存储的Looper实例；若me为null 则抛
出异常
    // 即loop （）执行前必须执行prepare （），从而创建1个Looper实例

    final MessageQueue queue = me.mQueue;
    // 获取Looper实例中的消息队列对象（MessageQueue）

    // 2. 消息循环（通过for循环）
    for (;;) {

        // 2.1 从消息队列中取出消息
        Message msg = queue.next();
        if (msg == null) {
            return;
        }
        // next()：取出消息队列里的消息，若取出的消息为空，则线程阻塞

        // 2.2 派发消息到对应的Handler
        msg.target.dispatchMessage(msg);
        // 把消息Message派发给消息对象msg的target属性，target属性实际是1个
handler对象

        // 3. 释放消息占据的资源
        msg.recycle();
    }
}

--------------------分析 queue.next()

/**
  * 定义：属于消息队列类（MessageQueue）中的方法
  * 作用：出队消息，即从 消息队列中 移出该消息
  */
  Message next() {

    ...

    // 该参数用于确定消息队列中是否还有消息
    // 从而决定消息队列应处于出队消息状态 or 等待状态
    int nextPollTimeoutMillis = 0;

    for (;;) {
        if (nextPollTimeoutMillis != 0) {
            Binder.flushPendingCommands();
```

```
        }

        // nativePollOnce方法在native层，若是nextPollTimeoutMillis
为-1，此时消息队列处于等待状态
        nativePollOnce(ptr, nextPollTimeoutMillis);

        synchronized (this) {

            final long now = SystemClock.uptimeMillis();
            Message prevMsg = null;
            Message msg = mMessages;

            // 出队消息，即 从消息队列中取出消息：按创建Message对象的时间顺序
            if (msg != null) {
                if (now < msg.when) {
                    nextPollTimeoutMillis = (int) Math.min(msg.when
- now, Integer.MAX_VALUE);
                } else {
                    // 取出了消息
                    mBlocked = false;
                    if (prevMsg != null) {
                        prevMsg.next = msg.next;
                    } else {
                        mMessages = msg.next;
                    }
                    msg.next = null;
                    if (DEBUG) Log.v(TAG, "Returning message: " +
msg);

                    msg.markInUse();
                    return msg;
                }
            } else {

                // 若 消息队列中已无消息，则将nextPollTimeoutMillis参数设
为-1
                // 下次循环时，消息队列则处于等待状态
                nextPollTimeoutMillis = -1;
            }
            ...
        }
        ...
    }
}

--------------------分析 dispatchMessage(msg)

/**
  * 定义：属于处理者类（Handler）中的方法
```

```
  * 作用：派发消息到对应的 Handler实例 & 根据传入的 msg作出对应的操作
  */
public void dispatchMessage(Message msg) {

    // 1. 若 msg.callback属性不为空，则代表使用了post（Runnable r）发送消息
    // 则执行 handleCallback(msg)，即回调 Runnable对象里复写的run（）
    if (msg.callback != null) {
        handleCallback(msg);
    } else {
        if (mCallback != null) {
            if (mCallback.handleMessage(msg)) {
                return;
            }
        }

        // 2. 若msg.callback属性为空，则代表使用了sendMessage（Message msg）发送消息（即此处需讨论的）
        // 则执行 handleMessage(msg)，即回调复写的 handleMessage(msg)
        handleMessage(msg);
    }
}

_______________________

private static void handleCallback(Message message) {
    message.callback.run();
}

/**
 * 注：该方法 = 空方法，在创建Handler实例时复写 = 自定义消息处理方式
 **/
public void handleMessage(Message msg) {
}
```

- 消息循环的操作 = 消息出队 + 分发给对应的Handler实例
- 分发给对应的Handler的过程：根据出队消息的归属者通过dispatchMessage(msg)进行分发，最终回调复写的handleMessage(Message msg)，从而实现 消息处理 的操作
- 特别注意：在进行消息分发时（dispatchMessage(msg)），会进行1次发送方式的判断：
  - 若msg.callback属性不为空，则代表使用了post（Runnable r）发送消息，则直接回调Runnable对象里复写的run（）
  - 若msg.callback属性为空，则代表使用了sendMessage（Message msg）发送消息，则回调复写的handleMessage(msg)

步骤说明		核心方法	具体描述	备注
1. 主线程创建时	a. 创建Looper & MessageQueue对象	Looper.prepareMainLooper() / Looper.prepare()	• 为当前线程 / 主线程 创建1个循环器对象（Looper） • 同时也生成了1个消息队列对象（MessageQueue）	• Looper、MessageQueue对象属于线程 • 主线程的Looper对象自动生成：创建主线程时自动调用 • 子线程的Looper对象需手动生成：手动调用方法
	b. 进入消息循环	• Looper.loop（） • MessageQueue.next（） • Handler.dispatchMessage() • Handler.handleMessage()	1. 消息出队：通过 MessageQueue.next（）将消息移出消息队列 2. 分发消息：根据出队消息的归属者通过dispatchMessage()分发到对应的Handler，最终回调复写的handleMessage()，从而实现 消息处理 的操作	• 创建Looper & MessageQueue对象后，则自动进入消息循环 • 在进行消息分发时（dispatchMessage(msg)），会进行1次发送方式的判断： a. 若msg.callback属性不为空，则代表使用了post（Runnable r）发送消息，则直接回调Runnable对象里复写的run（） b. 若msg.callback属性为空，则代表使用了sendMessage（Message msg）发送消息，则回调复写的handleMessage(msg) • 此处是情况b
2. 创建Handler实例对象 （需复写handleMessage()）		• Handler类的构造方法 • Handler.handleMessage()	1. 创建Handler实例对象 2. 指定Looper对象（此时绑定了线程） 3. 绑定MessageQueue对象 4. 复写Handler.handleMessage() 以便消息处理回调	• Handler需绑定 线程才能使用；绑定后，Handler的消息处理会在绑定的线程中执行 • 绑定方式 = 先指定Looper对象，从而绑定了 Looper对象所绑定的线程（因Looper对象本已绑定了对应线程） • 即：指定了Handler对象的 Looper对象 = 绑定到了Looper对象所在的线程

4.创建消息对象

```
/**
 * 具体使用
 */
Message msg = Message.obtain(); // 实例化消息对象
msg.what = 1; // 消息标识
msg.obj = "AA"; // 消息内容存放

/**
 * 作用：创建消息对象
 * 注：创建Message对象可用关键字new 或 Message.obtain()
 */
public static Message obtain() {

    // Message内部维护了1个 Message池，用于 Message消息对象的复用
    // 使用obtain （）则是直接从池内获取
    synchronized (sPoolSync) {
        if (sPool != null) {
            Message m = sPool;
            sPool = m.next;
            m.next = null;
            m.flags = 0; // clear in-use flag
            sPoolSize--;
            return m;
        }
        // 建议：使用obtain （）”创建“消息对象，避免每次都使用new重新分配内存
    }
    // 若池内无消息对象可复用，则还是用关键字new创建
    return new Message();
}
```

步骤说明	核心方法	具体描述	备注
创建消息对象 （Message）	Message.obtain （）	• Message类内部维护了1个Message池，用于消息对象的复用 • 使用obtain （）则是直接从池内获取	• 创建Message对象可用关键字new 或 Message.obtain() • 使用建议：使用obtain （）”创建“消息对象，避免每次都使用new重新分配内存 • 注：若池内无消息对象可复用，则还是用关键字new创建

5.在工作线程中 发送消息到消息队列中

```
/**
* 具体使用
*/
mHandler.sendMessage(msg);

/**
* 定义：属于处理器类（Handler）的方法
* 作用：将消息 发送 到消息队列中（Message -----> MessageQueue）
*/
```

```
*/
public final boolean sendMessage(Message msg) {
    return sendMessageDelayed(msg, 0);
}


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) {

    // 1. 获取对应的消息队列对象（MessageQueue）
    MessageQueue queue = mQueue;

    ...

    // 2. 调用了 enqueueMessage方法
    return enqueueMessage(queue, msg, uptimeMillis);
}


private boolean enqueueMessage(MessageQueue queue, Message msg,
long uptimeMillis) {
     // 1. 将msg.target赋值为this，即 ：把 当前的Handler实例对象作为msg的
target属性
     msg.target = this;
     // 请回忆起上面说的Looper的loop()中消息循环时，会从消息队列中取出每个消
息msg，然后执行 msg.target.dispatchMessage(msg)去处理消息（实际上则是将该
消息派发给对应的Handler实例 ）

    // 2. 调用消息队列的 enqueueMessage（），即：Handler发送的消息，最终是
保存到消息队列
    return queue.enqueueMessage(msg, uptimeMillis) ;
}

------------------------以下代码是在 MessageQueue 消息队列类中

/**
  * 定义：属于消息队列类（MessageQueue）的方法
  * 作用：入队，即 将消息 根据时间 放入到消息队列中（Message ----->
MessageQueue）
```

```
MessageQueue)

  * 采用单链表实现：提高插入消息、删除消息的效率
  */
boolean enqueueMessage(Message msg, long when) {

    ...

    synchronized (this) {

        msg.markInUse();
        msg.when = when;
        Message p = mMessages;
        boolean needWake;

        // 判断消息队列里有无消息
        // a. 若无，则将当前插入的消息 作为队头 & 若此时消息队列处于等待状态，
则唤醒
        if (p == null || when == 0 || when < p.when) {
            msg.next = p;
            mMessages = msg;
            needWake = mBlocked;
        } else {
            needWake = mBlocked && p.target == null &&
msg.isAsynchronous();
            Message prev;

            // b. 判断消息队列里有消息，则根据 消息（Message）创建的时间 插
入到队列中
            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;
        }

        if (needWake) {
            nativeWake(mPtr);
        }
    }
    return true;
}
```

```
}


// 之后，随着Looper对象的无限消息循环
// 不断从消息队列中取出Handler发送的消息 & 分发到对应Handler
// 最终回调Handler.handleMessage()处理消息
```

Handler发送消息的本质 = 为该消息定义target属性（即本身实例对象） & 将消息入队到绑定线程的消息队列中。具体如下：

步骤说明	核心方法	具体描述	备注
通过Handler发送消息 到 消息队列中	• Handler.sendMessage（） • Handler.enqueueMessage（） • MessageQueue.enqueueMessage（）	1. 获取对应的消息队列对象（MessageQueue） 2. 将消息对象的target属性 为 当前Handler实例 3. 调用MessageQueue.enqueueMessage（），将Handler需发送的消息入队到消息队列中	之后，随着Looper对象的无限消息循环，不断从消息队列中取出Handler发送的消息 & 分发到对应Handler，最终回调Handler.handleMessage()处理消息

6.总结

根据操作步骤的源码分析总结

步骤说明		核心方法	具体描述	备注
1. 主线程创建时	a. 创建Looper & MessageQueue对象	Looper.prepareMainLooper() / Looper.prepare()	• 为当前线程 / 主线程 创建1个循环器对象（Looper） • 同时也生成了1个消息队列对象（MessageQueue）	• Looper、MessageQueue对象属于线程 • 主线程的Looper对象自动生成：创建主线程时自动调用 • 子线程的Looper对象需手动生成：手动调用方法
	b. 进入消息循环	• Looper.loop（） • MessageQueue.next（） • Handler.dispatchMessage() • Handler.handleMessage()	1. 消息出队：通过 MessageQueue.next（）将消息移出消息队列 2. 分发消息：根据出队消息的归属者通过dispatchMessage()分发到对应的Handler，最终回调复写的handleMessage()，从而实现 消息处理 的操作	• 创建Looper & MessageQueue对象后，则自动进入消息循环 • 在进行消息分发时（dispatchMessage(msg)），会进行1次发送方式的判断： a. 若msg.callback属性不为空，则代表使用了post（Runnable r）发送消息，则直接回调Runnable对象里复写的run（） b. 若msg.callback属性为空，则代表使用了sendMessage（Message msg）发送消息，则回调复写的handleMessage(msg) • 此处是情况b
2. 创建Handler实例对象 （需复写handleMessage()）		• Handler类的构造方法 • Handler.handleMessage()	1. 创建Handler实例对象 2. 指定Looper对象（此时绑定了线程） 3. 绑定MessageQueue对象 4. 复写Handler.handleMessage() 以便消息处理回调	• Handler需绑定 线程才能使用：绑定后，Handler的消息处理会在绑定的线程中执行 • 绑定方式 = 先指定Looper对象，从而绑定了 Looper对象所绑定的线程（因Looper对象本已绑定了对应线程） • 即：指定了Handler对象的 Looper对象 = 绑定到了Looper对象所在的线程
3. 创建消息对象 （Message）		Message.obtain（）	• Message类内部维护了1个Message池，用于消息对象的复用 • 使用obtain（）则是直接从池内获取	• 创建Message对象可用关键字new 或 Message.obtain() • 使用建议：使用obtain（）"创建"消息对象，避免每次都使用new重新分配内存 • 注：若池内无消息对象可复用，则还是用关键字new创建
4. 通过Handler发送消息 到 消息队列中		• Handler.sendMessage（） • Handler.enqueueMessage（） • MessageQueue.enqueueMessage（）	1. 获取对应的消息队列对象（MessageQueue） 2. 将消息对象的target属性 为 当前Handler实例 3. 调用MessageQueue.enqueueMessage（），将Handler需发送的消息入队到消息队列中	之后，随着Looper对象的无限消息循环，不断从消息队列中取出Handler发送的消息 & 分发到对应Handler，最终回调Handler.handleMessage()处理消息

源码分析：Handler.post（）方式

```
// 在主线程中创建Handler实例
private Handler mhandler = new mHandler();

// 在工作线程中 发送消息到消息队列中 & 指定操作UI内容，需传入1个Runnable对象
mHandler.post(new Runnable() {
    @Override
    public void run() {
        ...
    }
});
```

1.在主线程中创建Handler实例

```
/**
  * 具体使用
  * 与方式1的不同：此处无复写Handler.handleMessage()
  */
private Handler mhandler = new  Handler();

/**
  * 源码分析：Handler的构造方法
  * 作用：
  *     a. 在此之前，主线程创建时,隐式创建了Looper对象、MessageQueue对象
  *     b. 初始化Handler对象、绑定线程 & 进入消息循环
  * 此处的源码分析类似方式1，此处不作过多描述
  */
```

2.在工作线程中 发送消息到消息队列中

```
/**
  * 需传入1个Runnable对象、复写run()从而指定UI操作
  */
mHandler.post(new Runnable() {
    @Override
    public void run() {
        // 要执行的UI操作
    }
});

/**
  * 源码分析：Handler.post（Runnable r）
```

```
  * 定义：属于处理者类（Handler）中的方法
  * 作用：定义UI操作、将Runnable对象封装成消息对象 & 发送 到消息队列中
 （Message ->>> MessageQueue）
  * 注：
  *    a. 相比sendMessage()，post（）最大的不同在于，更新的UI操作可直接在重
写的run（）中定义
  *    b. 实际上，Runnable并无创建新线程，而是发送 消息 到消息队列中
  */
public final boolean post(Runnable r) {

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



/**
 * 将传入的Runable对象封装成1个消息对象
 **/
private static Message getPostMessage(Runnable r) {
    // 1. 创建1个消息对象（Message）
    Message m = Message.obtain();

    // 2. 将 Runable对象 赋值给消息对象（message）的callback属性
    m.callback = r;

    // 3. 返回该消息对象
    return m;
}



/**
 * 实际上，从此处开始，则类似方式 1 = 将消息入队到消息队列，
 * 即 最终是调用 MessageQueue.enqueueMessage（）
 **/
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) {
    // 1. 获取对应的消息队列对象（MessageQueue）
    MessageQueue queue = mQueue;

    // 2. 调用了enqueueMessage方法
```

```
    return enqueueMessage(queue, msg, uptimeMillis);
}


private boolean enqueueMessage(MessageQueue queue, Message msg,
long uptimeMillis) {
     // 1. 将msg.target赋值为this，即 ：把 当前的Handler实例对象作为msg的
target属性
     msg.target = this;

    // 2. 调用消息队列的 enqueueMessage（），即：Handler发送的消息，最终是
保存到消息队列
    return queue.enqueueMessage(msg, uptimeMillis) ;
}
```

从上面的分析可看出：

- 消息对象的创建 = 内部 根据Runnable对象而封装
- 发送到消息队列的逻辑 = 方式1中sendMessage（Message msg）

3.下面，我们直接看 消息循环，即Looper.loop()方法

```
/**
  * 源码分析： Looper.loop()
  * 作用：消息循环，即从消息队列中获取消息、分发消息到Handler
  * 特别注意：
  *       a. 主线程的消息循环不允许退出，即无限循环
  *       b. 子线程的消息循环允许退出：调用消息队列MessageQueue的quit（）
  */
public static void loop() {

    ...

    // 1. 获取当前Looper的消息队列
    final Looper me = myLooper();
    if (me == null) {
        throw new RuntimeException("No Looper; Looper.prepare()
wasn't called on this thread.");
    }
    // myLooper()作用：返回sThreadLocal存储的Looper实例；若me为null 则抛
出异常
    // 即loop （）执行前必须执行prepare （），从而创建1个Looper实例

    final MessageQueue queue = me.mQueue;
    // 获取Looper实例中的消息队列对象（MessageQueue）
```

```
    // 2. 消息循环（通过for循环）
    for (;;) {

        // 2.1 从消息队列中取出消息
        Message msg = queue.next();
        if (msg == null) {
            return;
        }
        // next()：取出消息队列里的消息，若取出的消息为空，则线程阻塞

        // 2.2 派发消息到对应的Handler
        msg.target.dispatchMessage(msg);
        // 把消息Message派发给消息对象msg的target属性，target属性实际是1个handler对象

        // 3. 释放消息占据的资源
        msg.recycle();
    }
}

/**
  * 定义：属于处理者类（Handler）中的方法
  * 作用：派发消息到对应的 Handler实例 & 根据传入的msg作出对应的操作
  */
public void dispatchMessage(Message msg) {

    // 1. 若msg.callback属性不为空，则代表使用了post（Runnable r）发送消息，则执行handleCallback(msg)，即回调Runnable对象里复写的run（）
    if (msg.callback != null) {
        handleCallback(msg);
    } else {
        if (mCallback != null) {
            if (mCallback.handleMessage(msg)) {
                return;
            }
        }

        // 2. 若msg.callback属性为空，则代表使用了sendMessage（Message msg）发送消息，则执行handleMessage(msg)，即回调复写的handleMessage(msg)
        handleMessage(msg);
    }
}


private static void handleCallback(Message message) {

    message.callback.run();
    // Message对象的callback属性 = 传入的Runnable对象
```

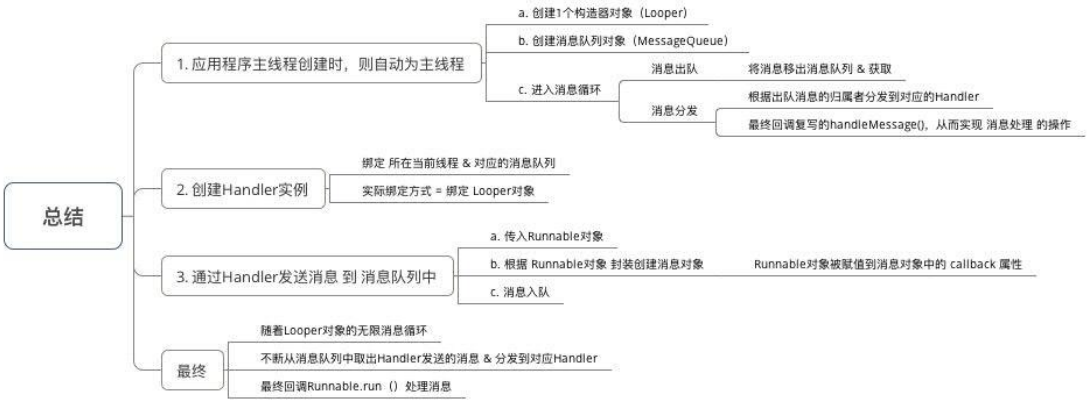
```
        // Message对象的callback属性 = 传入的Runnable对象
        // 即回调Runnable对象里复写的run()
    }
```

至此，已经明白了使用 Handler.post（）的工作流程：与方式1 Handler.sendMessage（）类似，区别在于：

- 不需外部创建消息对象，而是内部根据传入的Runnable对象 封装消息对象
- 回调的消息处理方法是：复写Runnable对象的run（）

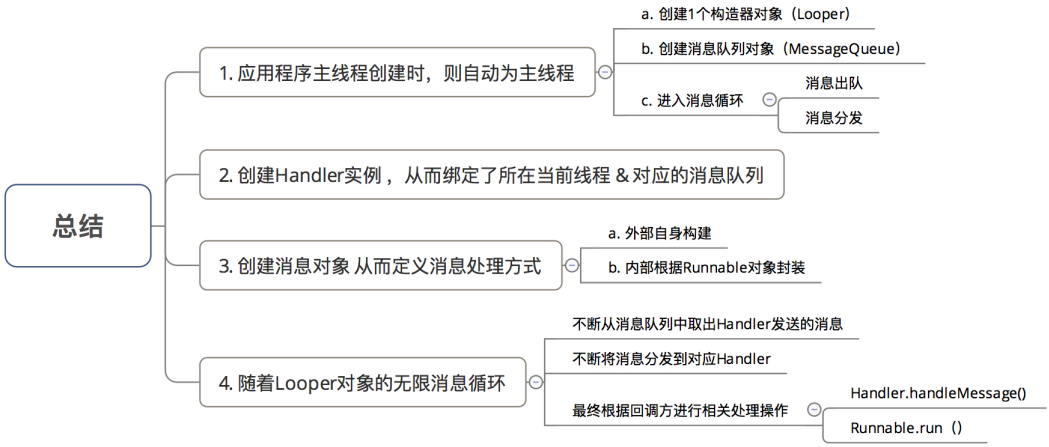
4.总结

步骤说明		核心方法	具体描述	备注
1. 主线程创建时	a. 创建Looper & MessageQueue对象	Looper.prepareMainLooper() / Looper.prepare()	• 为当前线程 / 主线程 创建1个循环器对象（Looper） • 同时也生成了1个消息队列对象（MessageQueue）	• Looper、MessageQueue对象属于线程 • 主线程的Looper对象自动生成：创建主线程时自动调用 • 子线程的Looper对象需手动生成：手动调用方法
	b. 进入消息循环	• Looper.loop（） • MessageQueue.next（） • Handler.dispatchMessage() • Handler.handleMessage()	1. 消息出队：通过 MessageQueue.next（）将消息移出消息队列 2. 分发消息：根据出队消息的归属者通过dispatchMessage()分发到对应的Handler，最终回调复写的handleMessage()，从而实现 消息处理 的操作	• 创建Looper & MessageQueue对象后，则自动进入消息循环 • 在进行消息分发时（dispatchMessage(msg)），会进行1次发送方式的判断： a. 若msg.callback属性不为空，则代表使用了post（Runnable r）发送消息，则直接回调Runnable对象里复写的run（） b. 若msg.callback属性为空，则代表使用了sendMessage（Message msg）发送消息，则回调复写的handleMessage(msg) • 此处是情况a
2. 创建Handler实例对象		Handler类的构造方法	1. 创建Handler实例对象 2. 指定Looper对象（此时绑定了线程） 3. 绑定MessageQueue对象	• Handler需绑定 线程才能使用；绑定后，Handler的消息处理会在绑定的线程中执行 • 绑定方式 = 先指定Looper对象，从而绑定了 Looper对象所绑定的线程（因Looper对象本已绑定了对应线程） • 即：指定了Handler对象的 Looper对象 = 绑定到了Looper对象所在的线程
3. 通过Handler发送消息 到 消息队列中 （需传入Ruunable对象 & 复写run（）））		• Handler.post（） • Handler.enqueueMessage（） • MessageQueue.enqueueMessage（） • Runnable.run（）	1. 内部根据传入的Runnable对象 封装成消息对象Message 2. 获取对应的消息队列对象（MessageQueue） 3. 将消息对象的target属性 为 当前Handler实例 4. 调用MessageQueue.enqueueMessage（），将Handler需发送的消息入队到消息队列中	将 Runable对象 赋值给消息对象（message）的callback属性



5.二者的具体异同

方式	相同		不相同点		
	目的	发送流程	复写消息处理方法的时机	回调消息处理的方法	消息对象创建（Message）
sendMessage（Message msg）	将消息发送到绑定线程的消息队列中	先获取消息队列对象；再 通过MessageQueue.enqueueMessage（）入队消息	创建Handler实例对象时	复写Handler.handleMessage（）	外部创建 & 直接传入到Handler
post（Runnable r）			Handler发送消息 到 消息队列时	复写Runnable.run （）	根据传入的Runnable对象进行内部封装



解决内存泄漏

Android中使用Handler造成内存泄露的原因

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

    public void handleMessage(android.os.Message msg) {
        if (msg.what == 1) {
            noteBookAdapter.notifyDataSetChanged();
        }
    }
};
```

上面是一段简单的Handler的使用。当使用内部类（包括匿名类）来创建Handler的时候，Handler对象会隐式地持有一个外部类对象（通常是一个Activity）的引用（不然你怎么可能通过Handler来操作Activity中的View？）。而Handler通常会伴随着一个耗时的后台线程（例如从网络拉取图片）一起出现，这个后台线程在任务执行完毕（例如图片下载完毕）之后，通过消息机制通知Handler，然后Handler把图片更新到界面。然而，如果用户在网络请求过程中关闭了Activity，正常情况下，Activity不再被使用，它就有可能在GC检查时被回收掉，但由于这时线程尚未执行完，而该线程持有Handler的引用（不然它怎么发消息给Handler？），这个Handler又持有Activity的引用，就导致该Activity无法被回收（即内存泄露）。

解决方案

方法一：通过程序逻辑来进行保护。

- 1.在关闭Activity的时候停掉你的后台线程。线程停掉了，就相当于切断了Handler和外部连接的线，Activity自然会在合适的时候被回收。
- 2.如果你的Handler是被delay的Message持有了引用，那么使用相应的Handler的removeCallbacks()方法，把消息对象从消息队列移除就行了。

方法二：将Handler声明为静态类。

> PS:在Java 中，非静态的内部类和匿名内部类都会隐式地持有其外部类的引用，静态的内部类不会持有外部类的引用。

- 静态类不持有外部类的对象，所以你的Activity可以随意被回收。由于Handler不再持有外部类对象的引用，导致程序不允许你在Handler中操作Activity中的对象了。所以你需要在Handler中增加一个对Activity的弱引用（WeakReference）。

```
static class MyHandler extends Handler{

    WeakReference<Activity> mWeakReference;

    public MyHandler(Activity activity){
        mWeakReference = new WeakReference<Activity>(activity);
    }

    @Override
    public void handleMessage(Message msg){

        final Activity activity = mWeakReference.get();
        if(activity != null){
            ...
        }
    }
}
```

对于上面的代码，用户在关闭Activity之后，就算后台线程还没结束，但由于仅有一条来自Handler的弱引用指向Activity，所以GC仍然会在检查的时候把Activity回收掉。这样，内存泄露的问题就不会出现了。