**Handler**：是一个消息分发对象，进行发送和处理消息，并且其 Runnable 对象与一个线程的 MessageQueue 关联。

**作用**：调度消息，将一个任务切换到某个指定的线程中去执行。

**为什么需要 Handler？**

**子线程不允许访问 UI**

假若子线程允许访问 UI，则在多线程并发访问情况下，会使得 UI 控件处于不可预期的状态。

传统解决办法：加锁，但会使得UI访问逻辑变的复杂，其次降低 UI 访问的效率。

### 引入 Handler

采用单线程模型处理 UI 操作，通过 Handler 切换到 UI 线程，解决子线程中无法访问 UI 的问题。

## Handler 使用

### 方式一： post(Runnable)

* 创建一个工作线程，实现 Runnable 接口，实现 run 方法，处理耗时操作
* 创建一个 handler，通过 handler.post/postDelay，投递创建的 Runnable，在 run 方法中进行更新 UI 操作。

new Thread(new Runnable() {

@Override

public void run() {

/\*\*

耗时操作

\*/

handler.post(new Runnable() {

@Override

public void run() {

/\*\*

更新UI

\*/

}

});

}

}).start();

### 方式二： sendMessage(Message)

* 创建一个工作线程，继承 Thread，重新 run 方法，处理耗时操作
* 创建一个 Message 对象，设置 what 标志及数据
* 通过 sendMessage 进行投递消息
* 创建一个handler，重写 handleMessage 方法，根据 msg.what 信息判断，接收对应的信息，再在这里更新 UI。

private Handler handler = new Handler(){

@Override

public void handleMessage(Message msg) {

super.handleMessage(msg);

switch (msg.what) { //判断标志位

case 1:

/\*\*

获取数据，更新UI

\*/

break;

}

}

};

public class WorkThread extends Thread {

@Override

public void run() {

super.run();

/\*\*

耗时操作

\*/

//从全局池中返回一个message实例，避免多次创建message（如new Message）

Message msg =Message.obtain();

msg.obj = data;

msg.what=1; //标志消息的标志

handler.sendMessage(msg);

}

}

new WorkThread().start();

Handler 存在的问题

内存方面

Handler 被作为 Activity 引用，如果为非静态内部类，则会引用外部类对象。当 Activity finish 时，Handler可能并未执行完，从而引起 Activity 的内存泄漏。**故而在所有调用 Handler 的地方，都用静态内部类。**

### 异常方面

当 Activity finish 时,在 onDestroy 方法中释放了一些资源。此时 Handler 执行到 handlerMessage 方法,但相关资源已经被释放,从而引起空指针的异常。

### 避免

* 如果是使用 handlerMessage，则在方法中加try catch。
* 如果是用 post 方法，则在Runnable方法中加try catch。

## Handler 的改进

* 内存方面：使用静态内部类创建 handler 对象，且对 Activity 持有弱引用
* 异常方面：不加 try catch，而是在 onDestory 中把消息队列 MessageQueue 中的消息给 remove 掉。

/\*\*

\* 为避免handler造成的内存泄漏

\* 1、使用静态的handler，对外部类不保持对象的引用

\* 2、但Handler需要与Activity通信，所以需要增加一个对Activity的弱引用

\*/

private static class MyHandler extends Handler {

private final WeakReference<Activity> mActivityReference;

//弱引用

MyHandler(Activity activity) {

this.mActivityReference = new WeakReference<Activity>(activity);

}

@Override

public void handleMessage(Message msg) {

super.handleMessage(msg);

MainActivity activity = (MainActivity) mActivityReference.get(); //获取弱引用队列中的activity

switch (msg.what) { //获取消息，更新UI

case 1:

byte[] data = (byte[]) msg.obj;

activity.threadIv.setImageBitmap(activity.getBitmap(data));

break;

}

}

}

@Override

protected void onDestroy() {

super.onDestroy();

//避免activity销毁时，messageQueue中的消息未处理完；故此时应把对应的message给清除出队列

handler.removeCallbacks(postRunnable); //清除runnable对应的message

//handler.removeMessage(what) 清除what对应的message

}

## Handler 的使用实现

* 耗时操作采用从网络加载一张图片
* 继承 Thread 或实现 Runnable 接口的线程，与 UI 线程进行分离，其中 Runnable 与主线程通过回调接口进行通信，降低耦合，提高代码复用性。

public class MainActivity extends AppCompatActivity {

ImageView threadIv;

ImageView runnableIv;

SendThread sendThread;

PostRunnable postRunnable;

private final MyHandler handler = new MyHandler(this);

@Override

protected void onCreate(Bundle savedInstanceState) {

super.onCreate(savedInstanceState);

setContentView(R.layout.activity\_main);

threadIv = (ImageView) findViewById(R.id.thread\_iv);

runnableIv = (ImageView) findViewById(R.id.runnable\_iv);

sendThread = new SendThread(handler);

sendThread.start(); //方法二

postRunnable = new PostRunnable(handler);

postRunnable.setRefreshUI(new PostRunnable.RefreshUI() {

@Override

public void setImage(byte[] data) {

runnableIv.setImageBitmap(getBitmap(data));

}

});

new Thread(postRunnable).start(); //方法一

}

、

/\*\*

为避免handler造成的内存泄漏

1、使用静态的handler，对外部类不保持对象的引用

2、但Handler需要与Activity通信，所以需要增加一个对Activity的弱引用

/

private static class MyHandler extends Handler {

private final WeakReference<Activity> mActivityReference;

MyHandler(Activity activity) {

this.mActivityReference = new WeakReference<Activity>(activity);

}

@Override

public void handleMessage(Message msg) {

super.handleMessage(msg);

MainActivity activity = (MainActivity) mActivityReference.get(); //获取弱引用队列中的activity

switch (msg.what) { //获取消息，更新UI

case 1:

byte[] data = (byte[]) msg.obj;

activity.threadIv.setImageBitmap(activity.getBitmap(data));

break;

}

}

}

private Bitmap getBitmap(byte[] data) {

return BitmapFactory.decodeByteArray(data, 0, data.length);

}

@Override

protected void onDestroy() {

super.onDestroy();

//避免activity销毁时，messageQueue中的消息未处理完；故此时应把对应的message给清除出队列

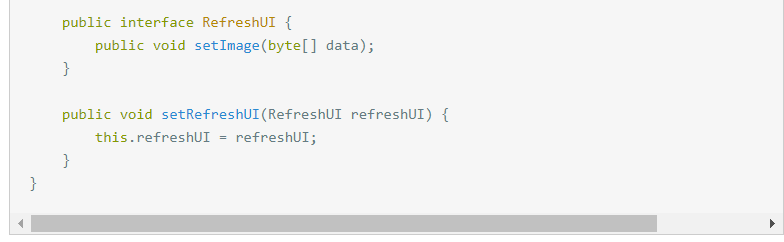
handler.removeCallbacks(postRunnable); //清除runnable对应的message

//handler.removeMessage(what) 清除what对应的message

}

}

#### 方式一：实现 runnable 接口，通过 post（Runnable）通信，并通过给定的回调接口通知 Activity 更新



#### 方式二:继承Thread，通过handler的sendMessage通信



## Handler 通信机制

## Handler 如何关联 Looper、MessageQueue

#### 1、Handler 发送消息

上一段很熟悉的代码：

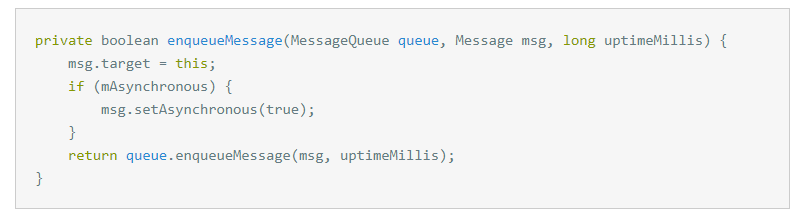


从sendMessageQueue开始追踪，函数调用关系：sendMessage -> sendMessageDelayed ->sendMessageAtTime，在sendMessageAtTime中，携带者传来的message与Handler的mQueue一起通过enqueueMessage进入队列了。

对于postRunnable而言，通过post投递该runnable，调用getPostMessage，通过该runnable构造一个message，再通过 sendMessageDelayed投递，接下来和sendMessage的流程一样了。

#### 2、消息入队列

在enqueueMessage中，通过MessageQueue入队列，并为该message的target赋值为当前的handler对象，记住msg.target很重要，之后Looper取出该消息时，还需要由msg.target.dispatchMessage回调到该handler中处理消息。

在MessageQueue中,由Message的消息链表进行入队列

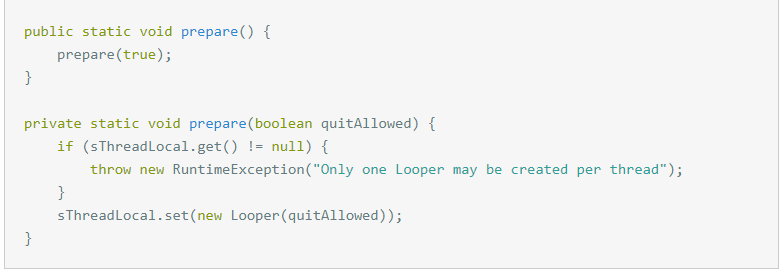


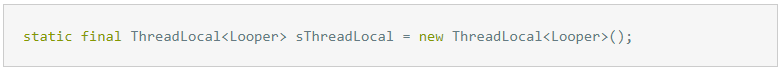
#### 3、Looper 处理消息

再说处理消息之前，先看Looper是如何构建与获取的：

* 构造Looper时，构建消息循环队列，并获取当前线程



* 但该函数是私有的，外界不能直接构造一个Looper，而是通过Looper.prepare来构造的：
* 这里创建Looper，并把Looper对象保存在sThreadLocal中，那sThreadLocal是什么呢？



它是一个保存Looper的TheadLocal实例，而ThreadLocal是线程私有的数据存储类，可以来保存线程的Looper对象，这样Handler就可以通过ThreadLocal来保存于获取Looper对象了。

* TheadLocal 如何保存与获取Looper？

 在 set 中都是通过 values.put 保存当前线程的 Looper 实例，通过 values.getAfterMiss(this)获取，其中put和getAfterMiss都有key和value，都是由Value对象的table数组保存的，那么在table数组里怎么存的呢？

很显然在数组中，前一个保存着ThreadLocal对象引用的索引，后一个存储传入的Looper实例。

* 接下来看Looper在loop中如何处理消息

在loop中，一个循环，通过next取出MessageQueue中的消息

若取出的消息为null，则结束循环，返回。

设置消息为空，可以通过MessageQueue的quit和quitSafely方法通知消息队列退出。

### 若取出的消息不为空，则通过msg.target.dispatchMessage回调到handler中去。4、handler处理消息

Looper把消息回调到handler的dispatchMessage中进行消息处理：

* 若该消息有callback，即通过Post(Runnable)的方式投递消息，因为在投递runnable时，把runnable对象赋值给了message的callback。
* 若handler的mCallback不为空，则交由通过callback创建handler方式去处理。
* 否则，由最常见创建handler对象的方式，在重写handlerMessage中处理。