#### 腾讯---Handler的原理是什么?能深入分析下 Handler的实现机制吗？

**面试官: Handler的原理是什么?能深入分析下 Handler的实现机制吗？**

**心理分析**：面试官问该问题是想问清楚handler的源码，handler机制如何实现，对消息泵Looper理不理解

**求职者:**应该从handler使用 Looper作用 源码分析 和最后总结讲起

# 目录

img

img

示意图

# 1. Handler 机制简介

* 定义 一套 Android 消息传递机制
* 作用

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

img

img

示意图

* 为什么要用 Handler消息传递机制 答：**多个线程并发更新UI的同时 保证线程安全**。具体描述如下

img

img

示意图

* 总结 使用Handler的原因：将工作线程需操作UI的消息 传递 到主线程，使得主线程可根据工作线程的需求 更新UI，**从而避免线程操作不安全的问题**

# 2. 储备知识

在阅读Handler机制的源码分析前，请务必了解Handler的一些储备知识：**相关概念、使用方式 & 工作原理**

### 2.1 相关概念

关于 Handler 机制中的相关概念如下：

在下面的讲解中，我将直接使用英文名讲解，即 Handler、Message、Message Queue、Looper，希望大家先熟悉相关概念

img

img

示意图

### 2.2 使用方式

* Handler使用方式 因**发送消息到消息队列的方式不同而不同**，共分为2种：使用Handler.sendMessage（）、使用Handler.post（）
* 下面的源码分析将依据使用步骤讲解

# 3. Handler机制的核心类

在源码分析前，先来了解Handler机制中的核心类

### 3.1 类说明

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）

### 3.2 类图

img

img

示意图

### 3.3 具体介绍

img

img

示意图

# 4. 源码分析

* 下面的源码分析将根据 Handler的使用步骤进行
* Handler使用方式 因**发送消息到消息队列的方式不同而不同**，共分为2种：使用Handler.sendMessage（）、使用Handler.post（）
* 下面的源码分析将依据上述2种使用方式进行

### 方式1：使用 Handler.sendMessage（）

* 使用步骤

/\*\*
 \* 此处以 匿名内部类 的使用方式为例
 \*/
 // 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

* 源码分析 下面，我将根据上述每个步骤进行源码分析

### 步骤1：在主线程中 通过匿名内部类 创建Handler类对象

/\*\*
 \* 具体使用
 \*/
 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
 }
 };

/\*\*
 \* 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) {

 ...// 仅贴出关键代码

 // 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 实现创建Handler对象操作的线程
* 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？

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

### 步骤1前的隐式操作1：创建循环器对象（Looper） & 消息队列对象（MessageQueue）

* 步骤介绍

img

img

示意图

* 源码分析

/\*\*
 \* 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
 }

 /\*\*
 \* 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）

 mRun = true;
 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) {
 ... // 仅贴出关键代码

 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

 }

总结：

* 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
1. 即 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
2. 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**
* 生成Looper & MessageQueue对象后，则会自动进入消息循环：Looper.loop（），即又是另外一个隐式操作。

### 步骤1前的隐式操作2：消息循环

此处主要分析的是Looper类中的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() {

 ...// 仅贴出关键代码

 // 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
 // ->> 分析1

 // 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
 // ->>分析2

 // 3. 释放消息占据的资源
 msg.recycle();
 }
}

/\*\*
 \* 分析1：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() {

 ...// 仅贴出关键代码

 // 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;
 }

 ......
 }
 .....
 }
}// 回到分析原处

/\*\*
 \* 分析2：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（）
 // 上述结论会在讲解使用“post（Runnable r）”方式时讲解
 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) ->> 分析3
 handleMessage(msg);

 }
 }

 /\*\*
 \* 分析3：handleMessage(msg)
 \* 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) {
 ... // 创建Handler实例时复写
 }

总结：

* 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：

1. 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（）
2. 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)

**至此，关于步骤1的源码分析讲解完毕**。总结如下

img

img

示意图

### 步骤2：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

/\*\*
 \* 源码分析：Message.obtain()
 \* 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();

 }

* 总结

img

img

示意图

### 步骤3：在工作线程中 发送消息到消息队列中

多线程的实现方式：AsyncTask、继承Thread类、实现Runnable

/\*\*
 \* 具体使用
 \*/

 mHandler.sendMessage(msg);

/\*\*
 \* 源码分析：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);
 // ->>分析1
 }

 /\*\*
 \* 分析1：sendMessageDelayed(msg, 0)
 \*\*/
 public final boolean sendMessageDelayed(Message msg, long delayMillis)
 {
 if (delayMillis < 0) {
 delayMillis = 0;
 }

 return sendMessageAtTime(msg, SystemClock.uptimeMillis() + delayMillis);
 // ->> 分析2
 }

 /\*\*
 \* 分析2：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 ->>分析3
 return enqueueMessage(queue, msg, uptimeMillis);
 }

 /\*\*
 \* 分析3：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->>分析4
 return queue.enqueueMessage(msg, uptimeMillis）;
 }

 /\*\*
 \* 分析4：queue.enqueueMessage(msg, uptimeMillis）
 \* 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）
 \* 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) {

 ...// 仅贴出关键代码

 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

* 总结 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：

img

img

示意图

至此，关于使用 Handler.sendMessage（）的源码解析完毕

### 总结

* 根据操作步骤的源码分析总结

img

img

示意图

* 工作流程总结

# 下面，将顺着文章：工作流程再理一次

img

img

示意图

img

img

示意图

### 方式2：使用 Handler.post（）

* 使用步骤

// 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 下面，我将根据上述每个步骤进行源码分析

实际上，该方式与方式1中的Handler.sendMessage（）工作原理相同、源码分析类似，下面将主要讲解不同之处

### 步骤1：在主线程中创建Handler实例

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

/\*\*
 \* 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
 \*/

### 步骤2：在工作线程中 发送消息到消息队列中

/\*\*
 \* 具体使用
 \* 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() {
 ... // 需执行的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);
 // getPostMessage(r) 的源码分析->>分析1
 // sendMessageDelayed（）的源码分析 ->>分析2

 }
 /\*\*
 \* 分析1：getPostMessage(r)
 \* 作用：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();
 // 注：创建Message对象可用关键字new 或 Message.obtain()
 // 建议：使用Message.obtain()创建，
 // 原因：因为Message内部维护了1个Message池，用于Message的复用，使用obtain（）直接从池内获取，从而避免使用new重新分配内存

 // 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;
 } // 回到调用原处

 /\*\*
 \* 分析2：sendMessageDelayed(msg, 0)
 \* 作用：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);
 // 请看分析3
 }

 /\*\*
 \* 分析3：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 ->>分析3
 return enqueueMessage(queue, msg, uptimeMillis);
 }

 /\*\*
 \* 分析4：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）;
 }

 // 注：实际上从分析2开始，源码 与 sendMessage（Message msg）发送方式相同

从上面的分析可看出：

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

下面，我们重新回到步骤1前的隐式操作2：消息循环，即Looper类中的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() {

 ...// 仅贴出关键代码

 // 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
 // ->>分析1

 // 3. 释放消息占据的资源
 msg.recycle();
 }
}

/\*\*
 \* 分析1：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（即此处需讨论的）
 // 则执行handleCallback(msg)，即回调Runnable对象里复写的run（）->> 分析2
 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);

 }
 }

 /\*\*
 \* 分析2：handleCallback(msg)
 \*\*/
 private static void handleCallback(Message message) {
 message.callback.run();
 // 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，你应该明白使用 Handler.post（）的工作流程：与方式1（Handler.sendMessage（））类似，区别在于：

1. 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
2. 回调的消息处理方法是：复写Runnable对象的run（）

二者的具体异同如下：

img

img

示意图

至此，关于使用 Handler.post（）的源码解析完毕

### 总结

* 根据操作步骤的源码分析总结

img

img

示意图

* 工作流程总结

# 下面，将顺着文章：工作流程再理一次

img

img

示意图

img

img

示意图

**至此，关于Handler机制的源码全部分析完毕。**

# 5. 总结

* 本文详细分析了Handler机制的源码，文字总结 & 流程图如下：

img

img

示意图

img

img

示意图

img

img

示意图

img

img