[深入理解 Handler 消息机制](https://segmentfault.com/a/1190000020304177)

记得很多年前的一次面试中，面试官问了这么一个问题，你在项目中一般如何实现线程切换？ 他的本意应该是考察 RxJava 的使用，只是我的答案是 Handler，他也就没有再追问下去了。在早期 Android 开发的荒芜时代，Handler 的确承担了项目中大部分的线程切换工作，通常包括子线程更新 UI 和消息传递。不光在我们自己的应用中，在整个 Android 体系中，Handler 消息机制也是极其重要的，不亚于 Binder 的地位。 ActivityThread.java 中的内部类 H 就是一个 Handler，它内部定义了几十种消息类型来处理一些系统事件。

Handler 的重要性毋庸置疑，今天就通过 AOSP 源码来深入学习 Handler。相关类的源码包含注释均已上传到我的 Github 仓库 [android\_9.0.0\_r45](https://link.segmentfault.com/?enc=w3ejnPjY6tyJfe9gmU7S7g%3D%3D.j6Tx3lrBrMDdJ1CyrcbZuhsHIPUI2GAWMQHVvy4HA8TLB1YjJ9bbOEJF15EmDe3z) :

[Handler.java](https://link.segmentfault.com/?enc=SVWr7HlKP7Q3EBDSZVRtRw%3D%3D.%2B7s%2FAaroqqfh4LvgoZ20j9qxJQ%2BMZBe0Zz2skA%2FNQ6889zfZQ9Ryk9rgo7yjj%2FWHY%2BQemE2BQfGHCMXQVGCbY0Hee%2FMPnjRpI7SnAecKQcFbN05f70l3wf1Qu76jqTIYE%2FSBK27rxwgo%2BC6BxVOO%2Fg%3D%3D)

[Looper.java](https://link.segmentfault.com/?enc=UdgXO9PiBF62WqEHr9ED1A%3D%3D.7OYY3mDyibef0mq1SGODsZ8aiD5GVQm1Ma9aRY3nrW2BuNVN4qLtw6%2FgVxXeaREz9uxDnQzef77RZ44RL4U%2BMk0vEbIwymn0wOjhbRn%2FosGlHU9y8yw%2FJkoXYjMV%2F1965rXpk84JFXZNqJV%2BRiYyyw%3D%3D)

[Message.java](https://link.segmentfault.com/?enc=yxYglDIjFp1778DZr2MkMQ%3D%3D.TytwRRrw1x3srn606eMQKWLJlLJGZDhzOnqUVBQ7%2BkTZXaF5HwIZoaD4fe13U8XK992o7n%2FlSdubIVLhKGczm7GWjSBjdqAq8FWhujDACcM7F5S0t6sjXDNaDgCLQFSFb0JXCb0vLoi5R8CSSQXPbw%3D%3D)

[MessageQueue.java](https://link.segmentfault.com/?enc=tyL6y1j%2Fg%2B0TIWMhECigDQ%3D%3D.%2BJpwLdTbP1xBD2Vftb79%2BssdlaukxhP1ZdUq7CogomATDKC2gpi08sldegfpfF0vRjW1eL80gOcFTbHguKRKJWU%2FlM5JcHIxVCnbErNkMirO8%2FVgF53KChZ%2BaOQVnmZyKtmN6muSFSibEinmXPAqgs252fWfSZ5WHiSjskO%2FKgE%3D)

Handler

Handler 用来发送和处理线程对应的消息队列 MessageQueue 中存储的 Message。每个 Handler 实例对应一个线程以及该线程的消息队列。当你创建一个新的 Handler，它会绑定创建它的线程和消息队列，然后它会向消息队列发送 Message 或者 Runnable，并且在它们离开消息队列时执行。

Handler 有两个主要用途：

1. 规划 Message 或者 Runnable 在未来的某个时间点执行
2. 在另一个线程上执行代码

以上翻译自官方注释。说白了，Handler 只是安卓提供给开发者用来发送和处理事件的，而消息如何存储，消息如何循环取出，这些逻辑则交给 MessageQueue 和 Looper 来处理，使用者并不需要关心。但要真正了解 Handler 消息机制，认真读一遍源码就必不可少了。

构造函数

Handler 的构造函数大致上可以分为两类，先来看第一类：

|  |
| --- |
| public Handler() {  this(null, false);  }  public Handler(Callback callback) {  this(callback, false);  }  public Handler(Callback callback, boolean async) {  // 如果是匿名类、内部类、本地类，且没有使用 static 修饰符，提示可能导致内存泄漏  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());  }  }  // 从当前线程的 ThreadLocal获取 Looper  mLooper = Looper.myLooper();  if (mLooper == null) { // 创建 Handler 之前一定要先创建 Looper。主线程已经自动为我们创建。  throw new RuntimeException(  "Can't create handler inside thread " + Thread.currentThread()  + " that has not called Looper.prepare()");  }  mQueue = mLooper.mQueue; // Looper 持有一个 MessageQueue  mCallback = callback; // handleMessage 回调  mAsynchronous = async; // 是否异步处理  } |

这一类构造函数最终调用的都是两个参数的方法，参数中不传递 Looper，所以要显式检查是否已经创建 Looper。创建 Handler 之前一定要先创建 Looper，否则会直接抛出异常。在主线程中 Looper 已经自动创建好，无需我们手动创建，在 ActivityThread.java 的 main() 方法中可以看到。Looper 持有一个消息队列 MessageQueue，并赋值给 Handler 中的 mQueue 变量。Callback 是一个接口，定义如下：

|  |
| --- |
| public interface Callback {  public boolean handleMessage(Message msg);  } |

通过构造器参数传入 CallBack 也是 Handler 处理消息的一种实现方式。

再回头看一下在上面的构造函数中是如何获取当前线程的 Looper 的？

|  |
| --- |
| mLooper = Looper.myLooper(); // 获取当前线程的 Looper |

这里先记着，回头看到 Looper 源码时再详细解析。

看过 Handler 的第一类构造函数，第二类其实就很简单了，只是多了 Looper 参数而已：

|  |
| --- |
| public Handler(Looper looper) {  this(looper, null, false);  }    public Handler(Looper looper, Callback callback) {  this(looper, callback, false);  }    public Handler(Looper looper, Callback callback, boolean async) {  mLooper = looper;  mQueue = looper.mQueue;  mCallback = callback;  mAsynchronous = async;  } |

直接赋值即可。

除此之外还有几个标记为 @hide 的构造函数就不作说明了。

### 发送消息

发送消息大家最熟悉的方法就是 sendMessage(Message msg) 了，可能有人不知道其实还有 post(Runnable r) 方法。虽然方法名称不一样，但最后调用的都是同一个方法。

|  |
| --- |
| sendMessage(Message msg)  sendEmptyMessage(int what)  sendEmptyMessageDelayed(int what, long delayMillis)  sendEmptyMessageAtTime(int what, long uptimeMillis)  sendMessageAtTime(Message msg, long uptimeMillis) |

几乎所有的 sendXXX() 最后调用的都是 sendMessageAtTime() 方法。

|  |
| --- |
| post(Runnable r)  postAtTime(Runnable r, long uptimeMillis)  postAtTime(Runnable r, Object token, long uptimeMillis)  postDelayed(Runnable r, long delayMillis)  postDelayed(Runnable r, Object token, long delayMillis) |

所有的 postXXX() 方法都是调用 getPostMessage() 将 参数中的 Runnable 包装成 Message，再调用对应的 sendXXX() 方法。看一下 getPostMessage() 的代码：

|  |
| --- |
| private static Message getPostMessage(Runnable r) {  Message m = Message.obtain();  m.callback = r;  return m;  }  private static Message getPostMessage(Runnable r, Object token) {  Message m = Message.obtain();  m.obj = token;  m.callback = r;  return m;  } |

主要是把参数中的 Runnable 赋给 Message 的 callback 属性。

殊途同归，发送消息的重任最后都落在了 sendMessageAtTime() 身上。

|  |
| --- |
| 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);  }    private boolean enqueueMessage(MessageQueue queue, Message msg, long uptimeMillis) {  msg.target = this;  if (mAsynchronous) {  msg.setAsynchronous(true);  }  return queue.enqueueMessage(msg, uptimeMillis); // 调用 Messagequeue 的 enqueueMessage() 方法  } |

Handler 就是一个撒手掌柜，发送消息的任务转手又交给了 MessageQueue 来处理。

再额外提一点，enqueueMessage() 方法中的参数 uptimeMillis 并不是我们传统意义上的时间戳，而是调用 SystemClock.updateMillis() 获取的，它表示自开机以来的毫秒数。

## MessageQueue

### enqueueMessage()

Message 的入队工作实际上是由 MessageQueue 通过 enqueueMessage() 函数来完成的。

|  |
| --- |
| boolean enqueueMessage(Message msg, long when) {  if (msg.target == null) { // msg 必须有 target  throw new IllegalArgumentException("Message must have a target.");  }  if (msg.isInUse()) { // msg 不能正在被使用  throw new IllegalStateException(msg + " This message is already in use.");  }  synchronized (this) {  if (mQuitting) { // 正在退出，回收消息并直接返回  IllegalStateException e = new IllegalStateException(  msg.target + " sending message to a Handler on a dead thread");  Log.w(TAG, e.getMessage(), e);  msg.recycle();  return false;  }  msg.markInUse();  msg.when = when;  Message p = mMessages;  boolean needWake;  if (p == null || when == 0 || when < p.when) {  // New head, wake up the event queue if blocked.  // 插入消息队列头部，需要唤醒队列  msg.next = p;  mMessages = msg;  needWake = mBlocked;  } else {  // Inserted within the middle of the queue. Usually we don't have to wake  // up the event queue unless there is a barrier at the head of the queue  // and the message is the earliest asynchronous message in the queue.  needWake = mBlocked && p.target == null && msg.isAsynchronous();  Message prev;  for (;;) {  prev = p;  p = p.next;  if (p == null || when < p.when) { // 按消息的触发时间顺序插入队列  break;  }  if (needWake && p.isAsynchronous()) {  needWake = false;  }  }  msg.next = p; // invariant: p == prev.next  prev.next = msg;  }  // We can assume mPtr != 0 because mQuitting is false.  if (needWake) {  nativeWake(mPtr);  }  }  return true;  } |

从源码中可以看出来，MessageQueue 是用链表结构来存储消息的，消息是按触发时间的顺序来插入的。

enqueueMessage() 方法是用来存消息的，既然存了，肯定就得取，这靠的是 next() 方法。

### next()

|  |
| --- |
| Message next() {  // Return here if the message loop has already quit and been disposed.  // This can happen if the application tries to restart a looper after quit  // which is not supported.  final long ptr = mPtr;  if (ptr == 0) {  return null;  }  int pendingIdleHandlerCount = -1; // -1 only during first iteration  int nextPollTimeoutMillis = 0;  for (;;) {  if (nextPollTimeoutMillis != 0) {  Binder.flushPendingCommands();  }  // 阻塞方法，主要是通过 native 层的 epoll 监听文件描述符的写入事件来实现的。  // 如果 nextPollTimeoutMillis = -1，一直阻塞不会超时。  // 如果 nextPollTimeoutMillis = 0，不会阻塞，立即返回。  // 如果 nextPollTimeoutMillis > 0，最长阻塞nextPollTimeoutMillis毫秒(超时)，如果期间有程序唤醒会立即返回。  nativePollOnce(ptr, nextPollTimeoutMillis);  synchronized (this) {  // Try to retrieve the next message. Return if found.  final long now = SystemClock.uptimeMillis();  Message prevMsg = null;  Message msg = mMessages;  if (msg != null && msg.target == null) {  // Stalled by a barrier. Find the next asynchronous message in the queue.  // msg.target == null表示此消息为消息屏障（通过postSyncBarrier方法发送来的）  // 如果发现了一个消息屏障，会循环找出第一个异步消息（如果有异步消息的话），  // 所有同步消息都将忽略（平常发送的一般都是同步消息）  do {  prevMsg = msg;  msg = msg.next;  } while (msg != null && !msg.isAsynchronous());  }  if (msg != null) {  if (now < msg.when) {  // 消息触发时间未到，设置下一次轮询的超时时间  // Next message is not ready. Set a timeout to wake up when it is ready.  nextPollTimeoutMillis = (int) Math.min(msg.when - now, Integer.MAX\_VALUE);  } else {  // Got a message.  // 得到 Message  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(); // 标记 FLAG\_IN\_USE  return msg;  }  } else {  // No more messages.  // 没有消息，会一直阻塞，直到被唤醒  nextPollTimeoutMillis = -1;  }  // Process the quit message now that all pending messages have been handled.  if (mQuitting) {  dispose();  return null;  }  // If first time idle, then get the number of idlers to run.  // Idle handles only run if the queue is empty or if the first message  // in the queue (possibly a barrier) is due to be handled in the future.  // Idle handle 仅当队列为空或者队列中的第一个消息将要执行时才会运行  if (pendingIdleHandlerCount < 0  && (mMessages == null || now < mMessages.when)) {  pendingIdleHandlerCount = mIdleHandlers.size();  }  if (pendingIdleHandlerCount <= 0) {  // No idle handlers to run. Loop and wait some more.  // 没有 idle handler 需要运行，继续循环  mBlocked = true;  continue;  }  if (mPendingIdleHandlers == null) {  mPendingIdleHandlers = new IdleHandler[Math.max(pendingIdleHandlerCount, 4)];  }  mPendingIdleHandlers = mIdleHandlers.toArray(mPendingIdleHandlers);  }  // Run the idle handlers.  // We only ever reach this code block during the first iteration.  // 只有第一次循环时才会执行下面的代码块  for (int i = 0; i < pendingIdleHandlerCount; i++) {  final IdleHandler idler = mPendingIdleHandlers[i];  mPendingIdleHandlers[i] = null; // release the reference to the handler  boolean keep = false;  try {  keep = idler.queueIdle();  } catch (Throwable t) {  Log.wtf(TAG, "IdleHandler threw exception", t);  }  if (!keep) {  synchronized (this) {  mIdleHandlers.remove(idler);  }  }  }  // Reset the idle handler count to 0 so we do not run them again.  // 将 pendingIdleHandlerCount 置零保证不再运行  pendingIdleHandlerCount = 0;  // While calling an idle handler, a new message could have been delivered  // so go back and look again for a pending message without waiting.  nextPollTimeoutMillis = 0;  }  } |

next() 方法是一个死循环，但是当没有消息的时候会阻塞，避免过度消耗 CPU。nextPollTimeoutMillis 大于 0 时表示等待下一条消息需要阻塞的时间。等于 -1 时表示没有消息了，一直阻塞到被唤醒。

这里的阻塞主要靠 native 函数 nativePollOnce() 来完成。其具体原理我并不了解，想深入学习的同学可以参考 Gityuan 的相关文 [Android消息机制2-Handler(Native层)](https://link.segmentfault.com/?enc=2q9pZOOcHVXb9b%2FX1eJF1Q%3D%3D.av18qZEbVaHS7PLWdVftntENA1vKninKiVIR94XxpyRzpNeI%2Faeh8nt0ljDnnPX5WS848k1C%2Bisw6rFSl3XznQ%3D%3D) 。

MessageQueue 提供了消息入队和出队的方法，但它自己并不是自动取消息。那么，谁来把消息取出来并执行呢？这就要靠 **Looper** 了。

## Looper

创建 Handler 之前必须先创建 Looper，而主线程已经为我们自动创建了 Looper，无需再手动创建，见 ActivityThread.java 的 main() 方法：

|  |
| --- |
| public static void main(String[] args) {  ...  Looper.prepareMainLooper(); // 创建主线程 Looper  ...  } |

### prepareMainLooper()

|  |
| --- |
| public static void prepareMainLooper() {  prepare(false);  synchronized (Looper.class) {  if (sMainLooper != null) {  throw new IllegalStateException("The main Looper has already been prepared.");  }  sMainLooper = myLooper();  }  } |

sMainLooper 只能被初始化一次，也就是说 prepareMainLooper() 只能调用一次，否则将直接抛出异常。

### prepare()

|  |
| --- |
| public static void prepare() {  prepare(true);  }  private static void prepare(boolean quitAllowed) {  // 每个线程只能执行一次 prepare()，否则会直接抛出异常  if (sThreadLocal.get() != null) {  throw new RuntimeException("Only one Looper may be created per thread");  }  // 将 Looper 存入 ThreadLocal  sThreadLocal.set(new Looper(quitAllowed));  } |

主线程中调用的是 prepare(false)，说明主线程 Looper 是不允许退出的。因为主线程需要源源不断的处理各种事件，一旦退出，系统也就瘫痪了。而我们在子线程调用 prepare() 来初始化 Looper时，默认调动的是 prepare(true)，子线程 Looper 是允许退出的。

每个线程的 Looper 是通过 ThreadLocal 来存储的，保证其线程私有。

再回到文章开头介绍的 Handler 的构造函数中 mLooper 变量的初始化：

|  |
| --- |
| mLooper = Looper.myLooper();  public static @Nullable Looper myLooper() {  return sThreadLocal.get();  } |

也是通过当前线程的 ThreadLocal 来获取的。

### 构造函数

private Looper(boolean quitAllowed) {

mQueue = new MessageQueue(quitAllowed); // 创建 MessageQueue

mThread = Thread.currentThread(); // 当前线程

}

再对照 Handler 的构造函数：

public Handler(Looper looper, Callback callback, boolean async) {

mLooper = looper;

mQueue = looper.mQueue;

mCallback = callback;

mAsynchronous = async;

}

其中的关系就很清晰了。

* Looper 持有 MessageQueue 对象的引用
* Handler 持有 Looper 对象的引用以及 Looper 对象的 MessageQueue 的引用

### loop()

class LooperThread extends Thread {

public Handler mHandler;

public void run() {

Looper.prepare();

mHandler = new Handler() {

public void handleMessage(Message msg) {

// process incoming messages here

}

};

Looper.loop();

}

}

让消息队列转起来的核心就是 Looper.loop()。

|  |
| --- |
| public static void loop() {  final Looper me = myLooper(); // 从 ThreadLocal 中获取当前线程的 Looper  if (me == null) {  throw new RuntimeException("No Looper; Looper.prepare() wasn't called on this thread.");  }  final MessageQueue queue = me.mQueue; // 获取当前线程的消息队列  ... // 省略部分代码  for (;;) { // 循环取出消息，没有消息的时候可能会阻塞  Message msg = queue.next(); // might block  if (msg == null) {  // No message indicates that the message queue is quitting.  return;  }  ... // 省略部分代码    try {  msg.target.dispatchMessage(msg); // 通过 Handler 分发 Message  dispatchEnd = needEndTime ? SystemClock.uptimeMillis() : 0;  } finally {  if (traceTag != 0) {  Trace.traceEnd(traceTag);  }  }  ... // 省略部分代码  msg.recycleUnchecked(); // 将消息放入消息池，以便重复利用  }  } |

简单说就是一个死循环不停的从 MessageQueue 中取消息，取到消息就通过 Handler 来进行分发，分发之后回收消息进入消息池，以便重复利用。

从消息队列中取消息调用的是 MessageQueue.next() 方法，之前已经分析过。在没有消息的时候可能会阻塞，避免死循环消耗 CPU。

取出消息之后进行分发调用的是 msg.target.dispatchMessage(msg)，msg.target 是 Handler 对象，最后再来看看 Handler 是如何分发消息的。

|  |
| --- |
| public void dispatchMessage(Message msg) {  if (msg.callback != null) { // callback 是 Runnable 类型，通过 post 方法发送  handleCallback(msg);  } else {  if (mCallback != null) { // Handler 的 mCallback参数 不为空时，进入此分支  if (mCallback.handleMessage(msg)) {  return;  }  }  handleMessage(msg); // Handler 子类实现的 handleMessage 逻辑  }  }  private static void handleCallback(Message message) {  message.callback.run();  } |

* Message 的 callback 属性不为空时，说明消息是通过 postXXX() 发送的，直接执行 Runnable 即可。
* Handler 的 mCallback 属性不为空，说明构造函数中传入了 Callback 实现，调用 mCallback.handleMessage(msg) 来处理消息
* 以上条件均不满足，只可能是 Handler 子类重写了 handleMessage() 方法。这好像也是我们最常用的一种形式。

## Message

之所以把 Message 放在最后说，因为我觉得对整个消息机制有了一个完整的深入认识之后，再来了解 Message 会更加深刻。首先来看一下它有哪些重要属性：

int what ：消息标识

int arg1 : 可携带的 int 值

int arg2 : 可携带的 int 值

Object obj : 可携带内容

long when : 超时时间

Handler target : 处理消息的 Handler

Runnable callback : 通过 post() 发送的消息会有此参数

Message 有 public 修饰的构造函数，但是一般不建议直接通过构造函数来构建 Message，而是通过 Message.obtain() 来获取消息。

### obtain()

|  |
| --- |
| public static Message obtain() {  synchronized (sPoolSync) {  if (sPool != null) {  Message m = sPool;  sPool = m.next;  m.next = null;  m.flags = 0; // clear in-use flag  sPoolSize--;  return m;  }  }  return new Message();  } |

sPool 是消息缓存池，链表结构，其最大容量 MAX\_POOL\_SIZE 为 50。obtain() 方法会直接从消息池中取消息，循环利用，节约资源。当消息池为空时，再去新建消息。

### recycleUnchecked()

还记得 Looper.loop() 方法中最后会调用 msg.recycleUnchecked() 方法吗？这个方法会回收已经分发处理的消息，并放入缓存池中。

void recycleUnchecked() {

// Mark the message as in use while it remains in the recycled object pool.

// Clear out all other details.

flags = FLAG\_IN\_USE;

what = 0;

arg1 = 0;

arg2 = 0;

obj = null;

replyTo = null;

sendingUid = -1;

when = 0;

target = null;

callback = null;

data = null;

synchronized (sPoolSync) {

if (sPoolSize < MAX\_POOL\_SIZE) {

next = sPool;

sPool = this;

sPoolSize++;

}

}

}

## 总结

说到这里，Handler 消息机制就全部分析完了，相信大家也对整个机制了然于心了。

* Handler 被用来发送消息，但并不是真正的自己去发送。它持有 MessageQueue 对象的引用，通过 MessageQueue 来将消息入队。
* Handler 也持有 Looper 对象的引用，通过 Looper.loop() 方法让消息队列循环起来。
* Looper 持有 MessageQueue 对象应用，在 loop() 方法中会调用 MessageQueue 的 next() 方法来不停的取消息。
* loop() 方法中取出来的消息最后还是会调用 Handler 的 dispatchMessage() 方法来进行分发和处理。

最后，关于 Handler 一直有一个很有意思的面试题：

Looper.loop() 是死循环为什么不会卡死主线程 ？

看起来问的好像有点道理，实则不然。你仔细思考一下，loop() 方法的死循环和卡死主线程有任何直接关联吗？其实并没有。

回想一下我们经常在测试代码时候写的 main() 函数：

public static void main(){

System.out.println("Hello World");

}

姑且就把这里当做主线程，它里面没有死循环，执行完就直接结束了，没有任何卡顿。但是问题是它就直接结束了啊。在一个 Android 应用的主线程上，你希望它直接就结束了吗？那肯定是不行的。所以这个死循环是必要的，保证程序可以一直运行下去。Android 是基于事件体系的，包括最基本的 Activity 的生命周期都是由事件触发的。主线程 Handler 必须保持永远可以相应消息和事件，程序才能正常运行。

另一方面，这并不是一个时时刻刻都在循环的死循环，当没有消息的时候，loop() 方法阻塞，并不会消耗大量 CPU 资源。

关于 Handler 就说到这里了。还记得文章说过线程的 Looper 对象是保存在 **ThreadLocal** 中的吗？下一篇文章就来说说 ThreadLocal 是如何保存 **线程局部变量** 的。