**Handler所有问题**

* **Handler 机制中，存在哪些角色？各自承担了什么功能？**

Handler：消息辅助类 & 对外的接口 & 向 MQ 投递消息 & 消息的目标处理者；  
Message：消息的载体 & 被 Handler 投递 & 自带 Handler 处理 & 自带消息池；  
Looper：循环器 & 持有 MQ & 循环从 MQ 中获取消息 & TLS 线程唯一；  
MessageQueue：基于时间的优先级队列 & 链表结构 & Java 与 C++ 层的纽带；

* **Looper死循环为什么不会导致应用卡死**

答： 因为当Looper处理完所有消息的时候会进入阻塞状态，当有新的Message进来的时候会打破阻塞继续执行。

这其实没理解好ANR这个概念。当我发送一个绘制UI 的消息到主线程Handler之后，经过一定的时间没有被执行，则抛出ANR异常。Looper的死循环，是循环执行各种事务，包括UI绘制事务。Looper死循环说明线程没有死亡，如果Looper停止循环，线程则结束退出了。Looper的死循环本身就是保证UI绘制任务可以被执行的原因之一。同时UI绘制任务有同步屏障，可以更加快速地保证绘制更快执行。

理解原因：  
1.界面的绘制本身就是这个循环内的一个事件  
2.界面的绘制是通过了同步屏障保护下发送的异步消息，会被主线程优先处理，因此使得界面绘制拥有了最高的优先级，不会因为 Handler 中事件太多而造成卡顿。

* **可以在子线程直接new一个Handler吗**

可以在子线程直接new一个Handler，不过需要在一个线程里需要先调用Looper.prepare()和Looper.loop()方法。

**Thread {**

**Looper.prepare()**

**object : Handler() {**

**override fun handleMessage(msg: Message) {**

**super.handleMessage(msg)**

**}**

**}**

**Looper.loop()**

**}.start()**

在主线程中为什么没看到Looper.prepare()？因为系统已经在应用启动的main方法里面调用Looper.prepareMainLooper()。Looper启动了handler的机制才能够正常运行，启动之前有需要prepare去创建Looper。如果不去调用loop()方法开启循环读取消息，你就算用handler传递了消息，没有去取消息的呀。

**public static void main(String[] args) {**

**Looper.prepareMainLooper();**

**Looper.loop();**

**}**

* **MessageQueue内部是什么样的数据结构？**

MessageQueue内部存有Message对象，Message内存存有一个next的Message对象，就形成了message->next->message->next ...。  
那么就形成了一个单链表。

**public final class MessageQueue {**

**Message mMessages;**

**Message next() {**

**if (msg != null && msg.target == null) {**

**// Stalled by a barrier. Find the next asynchronous message in the queue.**

**do {**

**prevMsg = msg;**

**msg = msg.next;**

**} while (msg != null && !msg.isAsynchronous());**

**}**

**}**

**}**

**public final class Message implements Parcelable {**

**@UnsupportedAppUsage**

**/\*package\*/ Message next;**

**}**

再看将消息入列的enqueueMessage方法：

**boolean enqueueMessage(Message msg, long when) {**

**synchronized (this) {**

**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.**

**msg.next = p;**

**mMessages = msg;**

**needWake = mBlocked;**

**} else {**

**needWake = mBlocked && p.target == null && msg.isAsynchronous();**

**Message prev;**

**for (;;) {**

**prev = p;**

**p = p.next;**

**if (p == null || when < p.when) {**

**break;**

**}**

**if (needWake && p.isAsynchronous()) {**

**needWake = false;**

**}**

**}**

**msg.next = p; // invariant: p == prev.next**

**prev.next = msg;**

**}**

**}**

**}**

enqueueMessage()方法里面，通过延迟时间when，去遍历消息队列的延迟时间，依次作比较，如果值小于某个消息的值，就会将该消息插入到这个消息前一个位置，实现一个节点的插入。为什么这里是一个队列呢？因为取消息的时候永远是从一个方向去取，永远是从头部开始取，那这不就是一个队列吗？

又因为发送消息最终是sendMessageAtTime()，可以控制传送的时间先后，将时间短的插入到先发送但是延迟时间长的节点前面去，所以MessageQueue是一个单链表的优先级队列。所以在loop方法中queue.next()能依次取出下一个消息，就是因为链表的结构。

* **Handler为什么会造成内存泄露？**

handler创建的时候，是生成了一个内部类，内部类能直接使用外部类的属性和方法，内部类是默认持有外部类的引用，即Activity。而在handler调用enqueueMessage方法时，将自己赋予了msg的target属性，所以msg是持有handler的引用的。如果某个消息因为延迟执行或者没有处理完成，消息会一直挂载。msg持有handler，handler持有activity导致activity无法回收，造成了内存泄露。

**private boolean enqueueMessage(@NonNull MessageQueue queue, @NonNull Message msg,**

**long uptimeMillis) {**

**msg.target = this;**

**return queue.enqueueMessage(msg, uptimeMillis);**

**}**

解决：将 Handler 定义成静态的内部类，在内部持有 Activity 的弱引用，并在  
Acitivity 的 onDestroy()中调用 handler.removeCallbacksAndMessages(null)及时  
移除所有消息。

* **Hander机制采用的是什么设计模式？**

采用的是生产者/消费者的设计模式。子线程生产消息，主线程消费消息，MessageQueue为生产仓库。

* **Handler是怎么实现切换线程的？**

**var handler = object: Handler() {**

**override fun handleMessage(msg: Message) {**

**Log.i("minfo", "从子线程收到消息 ${msg.what}")**

**}**

**}**

**object : Thread() {**

**override fun run() {**

**handler.sendEmptyMessage(200)**

**}**

**}**

在一个线程中创建handler，然后在另一个创建的线程中调用该handler的发送消息，该handler中能接收到消息，即实现了线程间通信，那么handler是如何实现线程切换的呢？

当在A线程中创建handler的时候，同时创建了Looper与MessageQueue，Looper在A线程中调用loop进入一个无限的for循环从MessageQueue中取消息。当B线程调用handler发送一个message的时候，会通过msg.target.dispatchMessage(msg);将message插入到handler对应的MessageQueue中，Looper发现有message插入到MessageQueue中，便取出message执行相应的逻辑，因为Looper.loop()是在A线程中启动的，所以则回到了A线程，达到了从B线程切换到A线程的目的。

* **handler.sendMessage()与handler.post()的区别？**

1、如果msg.callback不为空，也就是通过post方法发送消息的时候，会把消息交给这个msg.callback进行处理，然后就没有后续了。  
2、如果msg.callback为空，也就是通过sendMessage发送消息的时候，会判断Handler当前的mCallback是否为空，如果不为空就交给Handler.Callback.handleMessage处理。

所以post(Runnable) 与 sendMessage的区别就在于后续消息的处理方式，是交给msg.callback还是 Handler.Callback或者Handler.handleMessage。

**public void dispatchMessage(@NonNull Message msg) {**

**if (msg.callback != null) {**

**handleCallback(msg);**

**} else {**

**if (mCallback != null) {**

**if (mCallback.handleMessage(msg)) {**

**return;**

**}**

**}**

**handleMessage(msg);**

**}**

**}**

**private static void handleCallback(Message message) {**

**message.callback.run();**

**}**

* **MessageQueue是怎么增删消息的？**

**msg.next = p; // invariant: p == prev.next**

**prev.next = msg;**

添加消息：在enqueMessage方法中，将该条msg按照时间大小，遍历链表到正确的位置，将该msg插入到该节点中。

**msg.next = null;**

**msg.markInUse();**

**return msg;**

**void markInUse() {**

**flags |= FLAG\_IN\_USE;**

**}**

删除消息：在next方法中，将该msg对象的next指向null，并将flags标识为FLAG\_IN\_USE。

* **一个线程可以有几个Handler？几个Looper？几个MessageQueue？**

**一个线程可以有几个Handler？**

可以创建无数个Handler，但是他们使用的消息队列都是同一个，也就是同一个Looper。Handler在哪个线程创建的，就跟哪个线程的Looper关联，也可以在Handler的构造方法中传入指定的Looper，Looper.loop()循环读取消息。

那同一个Looper是怎么区分不同的Handler的？因为在msg入队列时，会将msg.target设置一个handler，处理消息的时候，也会调用msg对象的target去处理消息。

**一个线程有几个looper?如何保证的？**

Looper类中部分代码：

**final MessageQueue mQueue;**

**static final ThreadLocal<Looper> sThreadLocal = new ThreadLocal<Looper>();**

**private static void prepare(boolean quitAllowed) {**

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

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

**}**

**sThreadLocal.set(new Looper(quitAllowed));**

**}**

**public static @Nullable Looper myLooper() {**

**return sThreadLocal.get();**

**}**

从Looper的源码中可以看到，在一个线程中初始化Looper，是用ThreadLocal存储Looper对象。同时保证一个线程只拥有一个Looper，存入looper之前，从sThreadLocal.get()中取看是否存过looper， sThreadLocal中如果存在则会抛出异常，保证一个线程中的looper实例唯一，且一旦存入，不可修改不可新增。

**几个MessageQueue**

一个线程的looper只会创建一次，只有一个looper对象，一个looper下只有一个MessageQueue属性，所以一个线程只有一个MessageQueue。

**private Looper(boolean quitAllowed) {**

**mQueue = new MessageQueue(quitAllowed);**

**mThread = Thread.currentThread();**

**}**

* **A Handler发送的消息为什么不会跑到B Handler的handleMessage（）方法中？**

一个线程中包含多个Handler对象时，在消息的添加分发时，通过Message的target(Handler)标注，处理消息会去除对应的target去调用handleMessage，所以不会错乱。

* **ThreadLoacal的原理？**

这里见我的另一篇[ThreadLoacal的原理](https://www.jianshu.com/p/b1908246766a)

* **在UI中创建的Handler，通过post方式发送的消息在run方法中可以进行UI更新吗？**

在消息处理结束后，其回调至Runnable 的run方法中，需要注意的是这里的仍然是在UI线程中，因为我们创建的Handler是在UI线程中，且Handler将Runnable内部封住成Message的形式，所以最终调用的是Runable中的run()方法，因此还是回到UI线程中，可以更新UI界面。

* **主线程为什么不用初始化Looper？**

Android程序的入口在ActivityThread的main方法中，在main方法中已经调用了prepareMainLooper()去创建looper对象了，并且调用了 Looper.loop()。如果是自己创建的子线程中，需要自己初始化looper并调用loop方法。ActivityThread 不继承自 Thread，它只是一个运行在主线程上的对象。

**public static void main(String[] args) {**

**...**

**// 初始化主线程Looper**

**Looper.prepareMainLooper();**

**...**

**// 新建一个ActivityThread对象**

**ActivityThread thread = new ActivityThread();**

**thread.attach(false, startSeq);**

**// 获取ActivityThread的Handler，也是他的内部类H**

**if (sMainThreadHandler == null) {**

**sMainThreadHandler = thread.getHandler();**

**}**

**...**

**Looper.loop();**

**}**

* **Handler如何保证MessageQueue并发访问安全？**

各个线程都往一个messageQueue中存取msg，在存取数据的时候，是通过synchronized来保证了线程的安全性，使用messageQueue作为对象锁。各个子线程和主线程都是往用一个messageQueue存取消息，对调用同一个MessageQueue对象的线程来说，它们都是互斥的，所以保证了并发访问安全。

**boolean enqueueMessage(Message msg, long when) {**

**synchronized (this) {**

**......**

**}**

**}**

**Message next() {**

**synchronized (this) {**

**......**

**}**

**}**

* **能不能让一个Message加急被处理？**

可以 / 一种使得异步消息可以被更快处理的机制。  
而看了前面 MessageQueue::next 的代码我们知道，当 MessageQueue 中遇到了一个同步屏障，则它会不断地忽略后面的同步消息直到遇到一个异步的消息，这样设计的目的其实是为了使得当队列中遇到同步屏障时，则会使得异步的消息优先执行，这样就可以使得一些消息优先执行。

**Message next() {**

**for (;;) {**

**if (msg != null && msg.target == null) {**

**// 同步屏障，找到下一个异步消息**

**do {**

**prevMsg = msg;**

**msg = msg.next;**

**} while (msg != null && !msg.isAsynchronous());**

**}**

**}**

**}**

* **Message 的同步屏障有什么用？有什么意义？如何发送一个同步屏障？**

在 Handler 中还存在了一种特殊的消息，它的 target 为 null，并不会被消费，仅仅是作为一个标识处于 MessageQueue 中。它就是 SyncBarrier (同步屏障)这种特殊的消息。

**public int postSyncBarrier() {**

**return postSyncBarrier(SystemClock.uptimeMillis());**

**}**

**private int postSyncBarrier(long when) {**

**// Enqueue a new sync barrier token.**

**// We don't need to wake the queue because the purpose of a barrier is to stall it.**

**synchronized (this) {**

**final int token = mNextBarrierToken++;**

**final Message msg = Message.obtain();**

**msg.markInUse();**

**msg.when = when;**

**msg.arg1 = token;**

**Message prev = null;**

**Message p = mMessages;**

**if (when != 0) {**

**while (p != null && p.when <= when) {**

**prev = p;**

**p = p.next;**

**}**

**}**

**if (prev != null) { // invariant: p == prev.next**

**msg.next = p;**

**prev.next = msg;**

**} else {**

**msg.next = p;**

**mMessages = msg;**

**}**

**return token;**

**}**

**}**

移除同步屏障

**public void removeSyncBarrier(int token) {**

**...**

**}**

比如 View 的绘制过程中的 TraversalRunnable 消息就是异步消息，在放入队列之前先放入了一个消息屏障，从而使得界面绘制的消息会比其他消息优先执行，避免了因为 MessageQueue 中消息太多导致绘制消息被阻塞导致画面卡顿，当绘制完成后，就会将消息屏障移除。

* **同步屏障的使用场景**

貌似在开发的过程中，我们很少用到同步屏障，那么源码中在哪里用到了？

Android中的UI消息就是异步消息，需要优先处理 比如View更新，调用onDraw、requestLayout、invalidate等view最终都会调用到ViewRootImpl类中的scheduleTraversals方法。

**void scheduleTraversals(){**

**if（!mTraversalScheduled）{**

**mTraversalScheduled=true;**

**//开启同步屏障**

**mTraversalBarrier=mHandler.getLooper().getQueue().postSyncBarrier();**

**//发送异步消息**

**mChoreographer.postCallback();**

**...**

**}**

**}**

这里开启了同步屏障，并发送了异步消息，由于UI更显相关的消息是优先级最高的，这样系统就会优先处理这些异步消息了。

当然处理完消息后要移除同步屏障，这个时候就调用到了ViewRootImpl#unscheduleTraversals()。

* **什么是异步消息？如何发送**

意义：需配合同步屏障使用，否者与同步消息无区别；  
异步消息：setAsynchronous(true) → 向 flags 添加 FLAG\_ASYNCHRONOUS 标记  
发送方式 通过异步 Handler 发送 → 构造 Handler 时，async 传递 true 发送消息前，主动调用 setAsynchronous(true)  
安全起见，Android 9.0 普通开发者无法使用异步消息，所有发送方式被标记为 @hide

将handler标识标位异步handler，该handler就发送异步消息。

**public void setAsynchronous(boolean async) {**

**if (async) {**

**flags |= FLAG\_ASYNCHRONOUS;**

**} else {**

**flags &= ~FLAG\_ASYNCHRONOUS;**

**}**

**}**

* **Handler的阻塞唤醒机制是怎么回事？**

当消息不可用或者没有消息的时候就会阻塞在next方法，会进入nativePollOnce()方法，而阻塞的办法是通过pipe/epoll机制。

在enqueueMessage方法中，如果有新的消息进入，会根据needWeak字段，调用nativeWake()方法进行唤醒。

* **Handler 里藏着的 Callback 能干什么？**

Handler.Callback 有优先处理消息的权利 ，当一条消息被 Callback 处理并拦截  
（返回 true），那么 Handler 的 handleMessage(msg) 方法就不会被调用了；  
如果 Callback 处理了消息，但是并没有拦截，那么就意味着一个消息可以同时  
被 Callback 以及 Handler 处理。

* **什么是IdleHandler？**

当MessageQueue没有消息的时候，就会阻塞在next方法中，其实在阻塞之前，MessageQueue还会做一件事，就是检查是否存在IdleHandler，如果有，就会去执行它的queueIdle方法。

当没有消息处理的时候，就会去处理这个mIdleHandlers集合里面的每个IdleHandler对象，并调用其queueIdle方法。 最后根据queueIdle返回值判断是否用完删除当前的IdleHandler。

**private IdleHandler[] mPendingIdleHandlers;**

**Message next() {**

**int pendingIdleHandlerCount = -1;**

**for (;;) {**

**synchronized (this) {**

**//当消息执行完毕，就设置pendingIdleHandlerCount**

**if (pendingIdleHandlerCount < 0**

**&& (mMessages == null || now < mMessages.when)) {**

**pendingIdleHandlerCount = mIdleHandlers.size();**

**}**

**//初始化mPendingIdleHandlers**

**if (mPendingIdleHandlers == null) {**

**mPendingIdleHandlers = new IdleHandler[Math.max(pendingIdleHandlerCount, 4)];**

**}**

**//mIdleHandlers转为数组**

**mPendingIdleHandlers = mIdleHandlers.toArray(mPendingIdleHandlers);**

**}**

**// 遍历数组，处理每个IdleHandler**

**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);**

**}**

**//如果queueIdle方法返回false，则处理完就删除这个IdleHandler**

**if (!keep) {**

**synchronized (this) {**

**mIdleHandlers.remove(idler);**

**}**

**}**

**}**

**// Reset the idle handler count to 0 so we do not run them again.**

**pendingIdleHandlerCount = 0;**

**}**

**}**

* **IntentService是啥？有什么使用场景？**

**public abstract class IntentService extends Service {**

**private final class ServiceHandler extends Handler {**

**public ServiceHandler(Looper looper) {**

**super(looper);**

**}**

**@Override**

**public void handleMessage(Message msg) {**

**onHandleIntent((Intent)msg.obj);**

**stopSelf(msg.arg1);**

**}**

**}**

**@Override**

**public void onCreate() {**

**super.onCreate();**

**HandlerThread thread = new HandlerThread("IntentService[" + mName + "]");**

**thread.start();**

**mServiceLooper = thread.getLooper();**

**mServiceHandler = new ServiceHandler(mServiceLooper);**

**}**

**@Override**

**public void onStart(@Nullable Intent intent, int startId) {**

**Message msg = mServiceHandler.obtainMessage();**

**msg.arg1 = startId;**

**msg.obj = intent;**

**mServiceHandler.sendMessage(msg);**

**}**

这就是一个可以在子线程进行耗时任务，并且在任务执行后调用stopSelf()自动停止的Service。

* **HandlerThread是啥？有什么使用场景？**

**public class HandlerThread extends Thread {**

**@Override**

**public void run() {**

**mTid = Process.myTid();**

**Looper.prepare();**

**synchronized (this) {**

**mLooper = Looper.myLooper();**

**notifyAll();**

**}**

**Process.setThreadPriority(mPriority);**

**onLooperPrepared();**

**Looper.loop();**

**mTid = -1;**

**}**

**}**

HandlerThread就是一个封装了Looper的Thread类。就是为了让我们在子线程里面更方便的使用Handler。

* **Handler 分发事件优先级，是否可拦截？拦截的优先级如何？**

可以统一拦截消息，但无法拦截通过Runnable通过getPostMessage(Runnable r)生成的Message。  
因为它msg.callback不为空会优先处理msg.callback，不会经过统一的Hanlder的mCallback。

**public void dispatchMessage(@NonNull Message msg) {**

**if (msg.callback != null) {**

**handleCallback(msg);**

**} else {**

**if (mCallback != null) {**

**// mCallback 处理完如果返回 false，还是会继续往下走，再交给 Handler.handleMessage 处理的**

**// 所以这边可以通过反射去 hook 一个 Handler ，可以监听 Handler 处理的每个消息，也可以改 msg 里面的值**

**if (mCallback.handleMessage(msg)) {**

**return;**

**}**

**}**

**handleMessage(msg);**

**}**

**}**

* **主线程 Looper 何时运行？**

App 启动时，会调用到 ActivityThread 中，Looper 就在其 main() 方法中被启动；main() 中会主动调用 Looper.prepareMainLooper() 和 Looper.loop()

* **Handler 的 Message 可以分为那 3 类？分别有什么标识？**

1.发送普通消息  
2.发送异步消息  
给msg对象加入设置异步的标识。

**public void setAsynchronous(boolean async) {**

**if (async) {**

**flags |= FLAG\_ASYNCHRONOUS;**

**} else {**

**flags &= ~FLAG\_ASYNCHRONOUS;**

**}**

**}**

3.同步屏障  
使用postSyncBarrier方法发送同步屏障，使msg.target == null。

**private int postSyncBarrier(long when) {**

**...**

**}**

* **点击页面上的按钮后更新TextView的内容，谈谈你的理解？**

点击按钮的时候会发送消息到Handler，但是为了保证优先执行，会加一个标记异步，同时会发送一个target为null的消息，这样在使用消息队列的next获取消息的时候，如果发现消息的target为null，那么会遍历消息队列将有异步标记的消息获取出来优先执行，执行完之后会将target为null的消息移除。(同步屏障)

* **同一个 Message 对象能否重复 send？**

**关键在于如何定义同一个 Message。**

**角度一：Java 对象层面，可被复用；**

**原因：Message 由消息池维护，即同一个对象被回收后会被再次复用；｜ new Message & Message.obtain()**

**角度二：业务层面，不能复用；**

**原因：Message 通过 enqueueMessage() 入队时，会通过 markInUse() 标记，再次入队无法通过 isInUse() 检查，则抛出异常；**

* **Looper.loop 中，如果没有待处理的消息，为什么不会阻塞 UI？**

主线程在 MessageQueue 没有消息时，会阻塞在 loop 的 queue.next() 方法中的 nativePollOnce()方法里。

此时主线程会释放 CPU 资源进入休眠状态，直到下一个消息到达或者有事务发生，通过往 pipe 管道写端写入数据的方式，来唤醒主线程。这里采用的是 epoll 机制。

epoll 机制是一种 IO 多路复用机制，可以同时监控多个描述符，在有事件发生的时候，立即通知相应程序进行读或写操作，类似一种 callback 的回调机制。主线程在大多数时候是处于休眠状态，并不会消耗大量的 CPU 资源。当有新的消息或事务到达时，会立即唤醒主线程进行处理，所以对用户来说是无感知的。

* **Looper 的 Printer 输出的日志，有什么其他用途？依靠的原理是什么？有什么缺点？**

用途：性能监控；  
原理：通过筛选日志内存，区分 Message 的开始执行和结束执行的时间点，即可判断处理 Message 的耗时，即主线程卡顿耗时；  
缺点：Printer 存在大量字符串拼接，在消息量大时，会导致性能受损；｜ 实测数据：存在 Printer 时，在列表快速滑动时，平均帧率降低 5 帧；

* **Handler 可以 IPC 通信吗？**

不能；Handler是一种共享内存的通信方式，Handler 只能用于共享内存地址的 2 个线程通信，即同进程的 2 个线程通信；

* **为什么系统不建议在子线程访问UI？（为什么不能在子线程更新UI？）**

如果采用多线程访问UI会出现线程安全，那为什么不加锁呢？

加锁会降低UI访问的效率。本身UI控件就是离用户比较近的一个组件，加锁之后自然会发生阻塞，那么UI访问的效率会降低，最终反应到用户端就是这个手机有点卡。  
所以，Android设计出了单线程模型来处理UI操作，再搭配上Handler，是一个比较合适的解决方案。

* **子线程访问UI的 崩溃原因 和 解决办法？**

1.在ViewRootImpl创建之前进行子线程的UI更新，比如onCreate方法中进行子线程更新UI。  
2.子线程切换到主线程进行UI更新，比如Handler、view.post方法。  
3.给操作的view设置大小为matchparent或者

* **Message可以如何创建？哪种效果更好，为什么？**

Message.obtain来创建Message，这样会复用之前的Message的内存，不会频繁的创建对象，导致内存抖动。

* **Message消息被分发之后会怎么处理？消息怎么复用的？**

在一个message对象调用了dispatchMessage之后，会进行回收操作。  
在recycleUnchecked方法中，释放了所有资源，然后将当前的空消息插入到sPool表头。  
这里的sPool就是一个消息对象池，它也是一个链表结构的消息，最大长度为50。  
使用obtain来获取复用消息，直接复用消息池sPool中的第一条消息，然后sPool指向下一个节点，消息池数量减一。

**public void recycle() {**

**if (isInUse()) {**

**if (gCheckRecycle) {**

**throw new IllegalStateException("This message cannot be recycled because it "**

**+ "is still in use.");**

**}**

**return;**

**}**

**recycleUnchecked();**

**}**

**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 = UID\_NONE;**

**workSourceUid = UID\_NONE;**

**when = 0;**

**target = null;**

**callback = null;**

**data = null;**

**synchronized (sPoolSync) {**

**if (sPoolSize < MAX\_POOL\_SIZE) {**

**next = sPool;**

**sPool = this;**

**sPoolSize++;**

**}**

**}**

**}**

* **主线程中能否调用quit()方法？**

是不能的。它会抛出一个异常，让程序挂掉。

* **修改了手机系统时间，handler的延时消息收到也会发生改变吗**

发送延时消息，会调用sendMessageDelayed，然后调用sendMessageAtTime，将一个时间点传入，该时间为SystemClock.uptimeMillis()

**public final boolean sendMessageDelayed(@NonNull Message msg, long delayMillis) {**

**if (delayMillis < 0) {**

**delayMillis = 0;**

**}**

**return sendMessageAtTime(msg, SystemClock.uptimeMillis() + delayMillis);**

**}**

**/\*\***

**\* Returns milliseconds since boot, not counting time spent in deep sleep.**

**\***

**\* @return milliseconds of non-sleep uptime since boot.**

**\*/**

**@CriticalNative**

**native public static long uptimeMillis();**

SystemClock.uptimeMillis() 是一个表示当前时间的一个相对时间，它代表的是 自系统启动开始从0开始的到调用该方法时相差的毫秒数

System.currentTimeMillis() 代表的是从 1970-01-01 00:00:00 到当前时间的毫秒数,我们可以通过修改系统时间达到修改该值的目的，所以该值是不可靠的值

* **handler怎么移除一个消息**

通过调用：

**Handler().removeMessages(what: Int)**

调用了handler中：

**public final void removeMessages(int what) {**

**mQueue.removeMessages(this, what, null);**

**}**

进入removeMessages方法：

**void removeMessages(Handler h, Runnable r, Object object) {**

**if (h == null || r == null) {**

**return;**

**}**

**synchronized (this) {**

**Message p = mMessages;**

**// Remove all messages at front.**

**while (p != null && p.target == h && p.callback == r**

**&& (object == null || p.obj == object)) {**

**Message n = p.next;**

**mMessages = n;**

**p.recycleUnchecked();**

**p = n;**

**}**

**// Remove all messages after front.**

**while (p != null) {**

**Message n = p.next;**

**if (n != null) {**

**if (n.target == h && n.callback == r**

**&& (object == null || n.obj == object)) {**

**Message nn = n.next;**

**n.recycleUnchecked();**

**p.next = nn;**

**continue;**

**}**

**}**

**p = n;**

**}**

**}**

**}**

removeMessages会将handler对应message queue里的消息清空，如果带了int参数则是对应的消息清空。1、这个方法使用的前提是之前调用过sendEmptyMessageDelayed(0, time)，意思是延迟time执行handler中msg.what=0的方法；  
2、在延迟时间未到的前提下，执行removeMessages(0)，则上面的handler中msg.what=0的方法取消执行；  
3、在延迟时间已到，handler中msg.what=0的方法已执行，再执行removeMessages(0)，不起作用。

参考：  
[https://blog.csdn.net/javine/article/details/45953575](https://links.jianshu.com/go?to=https%3A%2F%2Fblog.csdn.net%2Fjavine%2Farticle%2Fdetails%2F45953575)  
[https://blog.csdn.net/bzlj2912009596/article/details/79736912](https://links.jianshu.com/go?to=https%3A%2F%2Fblog.csdn.net%2Fbzlj2912009596%2Farticle%2Fdetails%2F79736912)

* **handler.removeCallbacksAndMessages(null)是什么意思**

如果需要删除handler所有的消息和回调函数，那就需要使用handler.removeCallbacksAndMessages(null)。  
这样做的好处是在Acticity退出的时候，可以避免内存泄露，因为有延迟消息，msg持有handler，handler持有activity，activity退出需要移除所有消息。  
会while循环遍历msg消息链表，调用msg的recycleUnchecked()进行回收，方法会将msg所有属性置空和置0。

**参考：**

<https://www.jianshu.com/p/7f1c46fb55c8>

[https://blog.csdn.net/qq\_39477770/article/details/109331658?utm\_medium=distribute.pc\_relevant.none-task-blog-baidujs\_title-1&spm=1001.2101.3001.4242](https://links.jianshu.com/go?to=https%3A%2F%2Fblog.csdn.net%2Fqq_39477770%2Farticle%2Fdetails%2F109331658%3Futm_medium%3Ddistribute.pc_relevant.none-task-blog-baidujs_title-1%26spm%3D1001.2101.3001.4242)

[https://blog.csdn.net/u012165769/article/details/114681388?utm\_medium=distribute.pc\_relevant.none-task-blog-baidujs\_title-0&spm=1001.2101.3001.4242](https://links.jianshu.com/go?to=https%3A%2F%2Fblog.csdn.net%2Fu012165769%2Farticle%2Fdetails%2F114681388%3Futm_medium%3Ddistribute.pc_relevant.none-task-blog-baidujs_title-0%26spm%3D1001.2101.3001.4242)

[https://blog.csdn.net/u012165769/article/details/113531570](https://links.jianshu.com/go?to=https%3A%2F%2Fblog.csdn.net%2Fu012165769%2Farticle%2Fdetails%2F113531570)

[https://blog.csdn.net/weixin\_39952074/article/details/111249137](https://links.jianshu.com/go?to=https%3A%2F%2Fblog.csdn.net%2Fweixin_39952074%2Farticle%2Fdetails%2F111249137)

[说一下Handler的同步屏障机制？](https://www.jianshu.com/p/85f5c951da57?utm_campaign=haruki&utm_content=note&utm_medium=reader_share&utm_source=weixin)