Handler源码分析&相关问题汇总

1. 使用Handler进行线程间通信的特点

· 线程间如何通信

Handler通信实现的方案实际上是内存共享

· 为什么线程间不会干扰

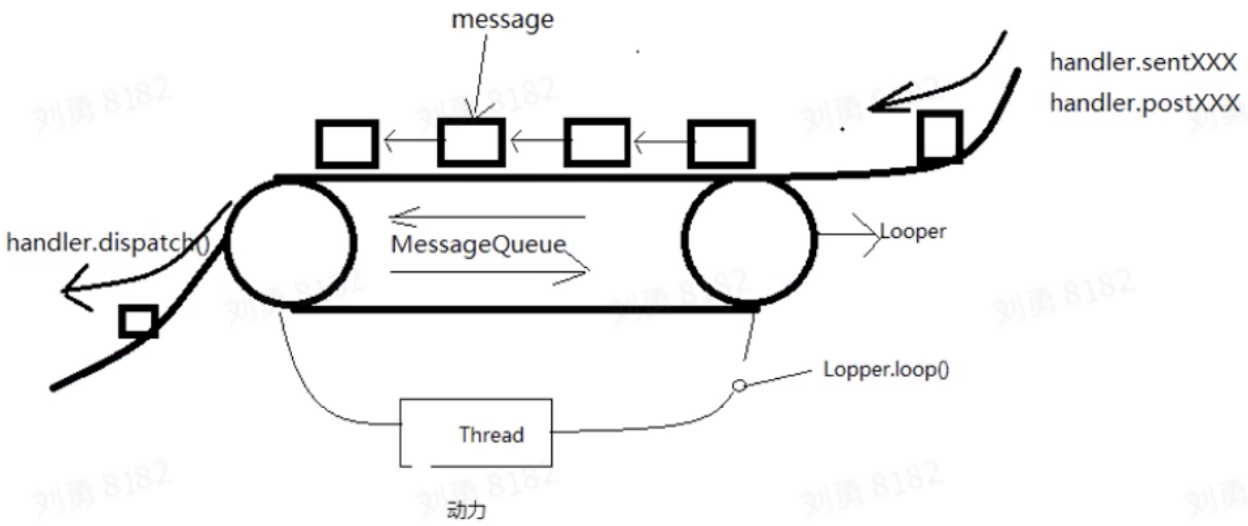
内存管理设计思路优秀

· 为什么不用wait/notify

因为handler已经将需要这部分功能进行Linux层的封装

2. Handler工作流程

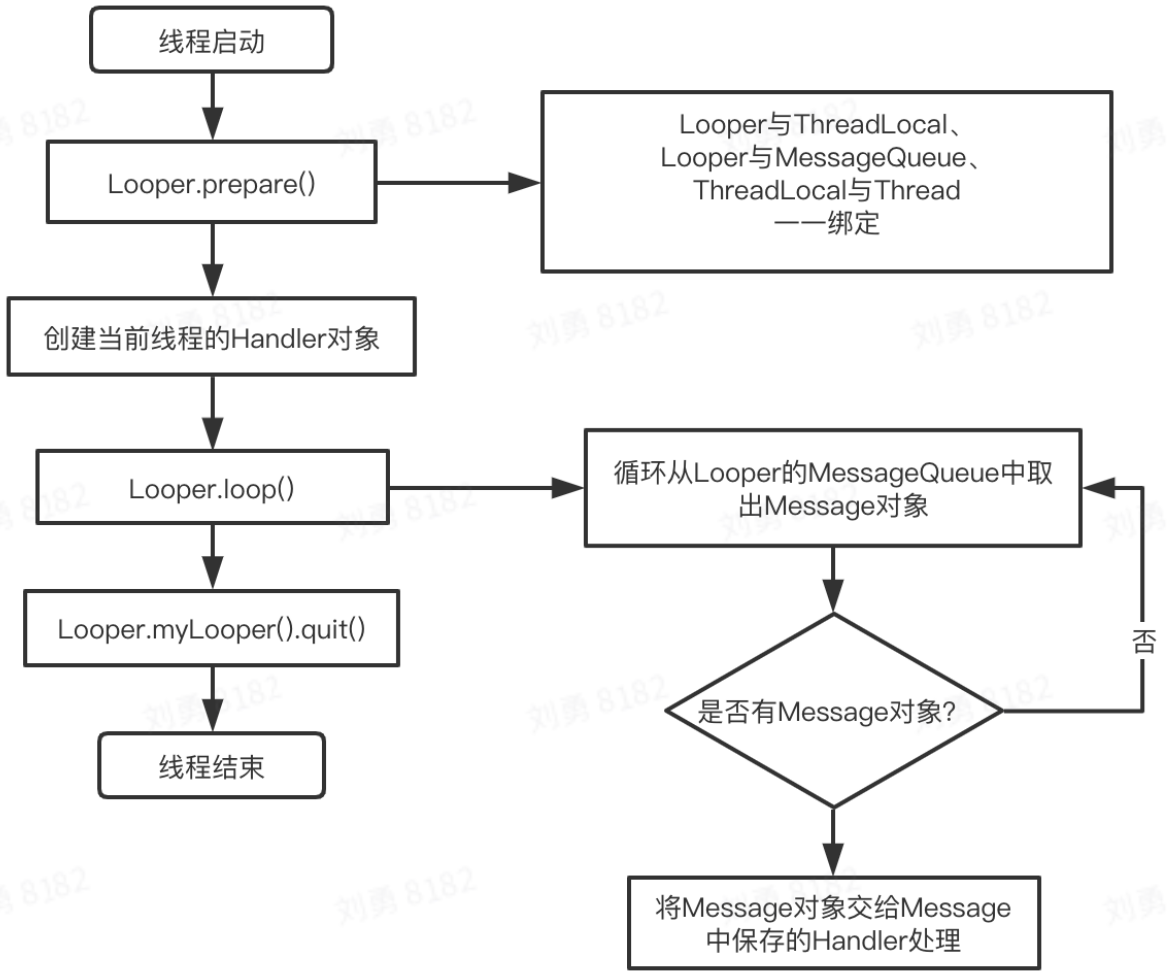
· 传送带机制



Handler消息机制类似传送带。
传送带的电源是线程Thread，当线程run起来的时候，传送带就有了动力；
Looper.loop()是打开传送带的开关，传送带开始运行；

Handler是人，不管在什么线程发送消息，最后都会把消息放入传送带中；
Message是人的物品，谁把物品放入传送带，最后物品也会通过传送带回到人的手中。

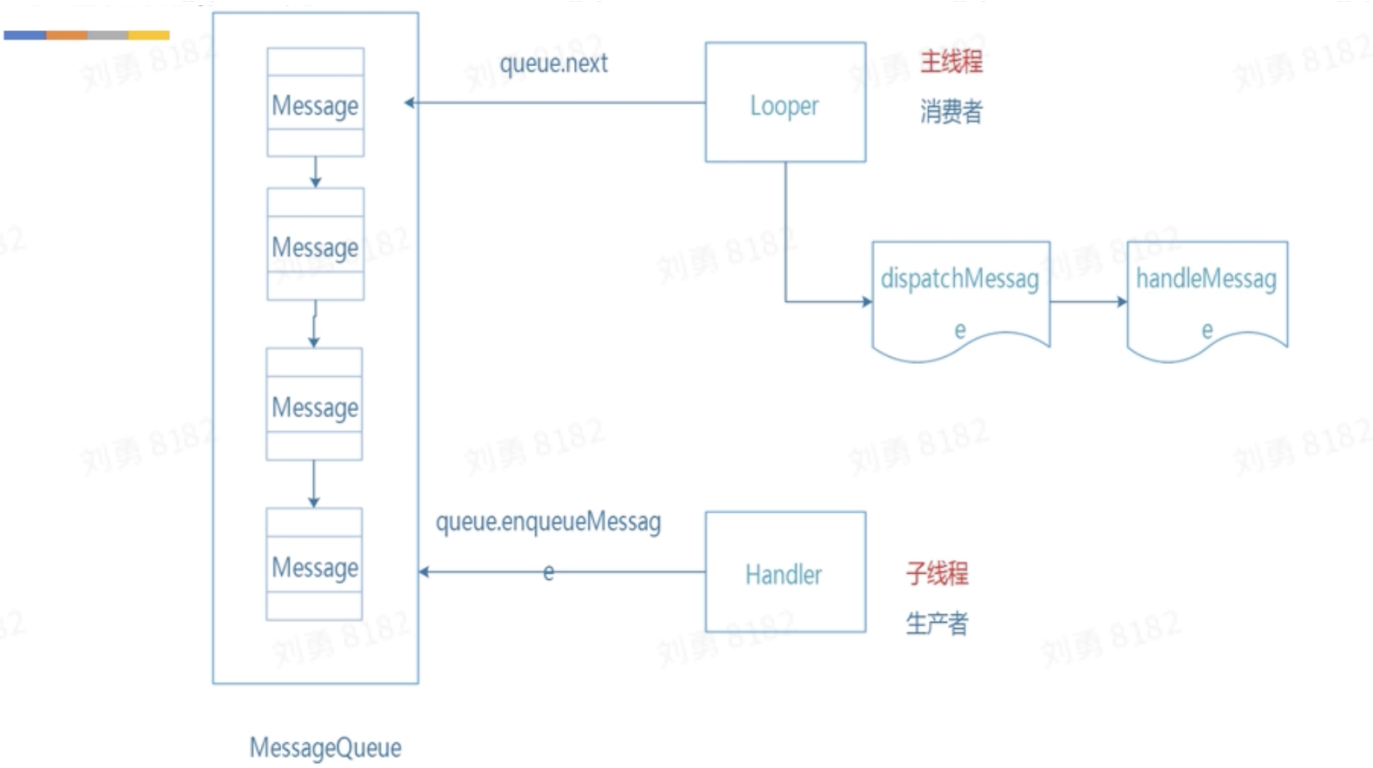
· 消息机制流程图



a. 线程启动，调用Looper.prepare()方法，创建Looper对象，并初始化Looper内部的MessageQueue成员。通过ThreadLocal使Looper对象与线程Thread一一对应。

互动问题一：
Looper的构造方法是private，那么Looper对象是不是一个单例？

b. 在该线程创建Handler对象，同时重写handleMessage(Message msg)方法来处理MessageQueue中的消息。
c. 由于线程间共享内存，新建子线程也可以使用该Handler对象发送消息到MessageQueue
d. 使用生产者—消费者模式，handler在子线程将消息插入队列。然后在该线程调用Looper.loop()，循环从Looper的MessageQueue种取消息给对应的Handler处理



互动问题二：
同一个looper去实例了两个handler，通过这两个handler去sendMessage的时候，最终哪个handler会处理message对象？

e. 如果在子线程中创建Handler对象，需要在消息处理结束时,调用Looper.myLooper().quit()结束Looper内的死循环，否则线程一直执行导致内存泄漏。

互动问题三：
为什么在Handler的构造方法和Looper.loop()方法中，需要通过myLooper()方法获取Looper对象？

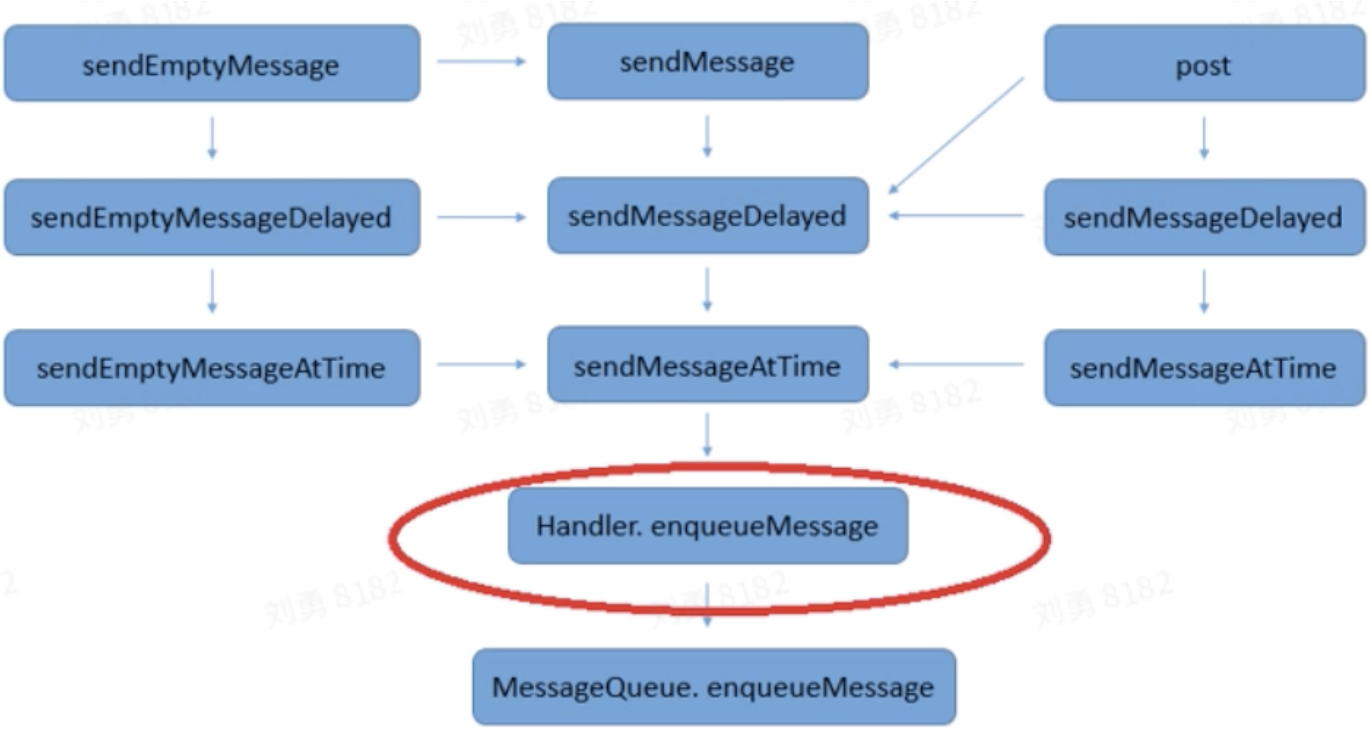
手写Handler Demo地址

https://github.com/sa16225193/handlerDemo

3. 主要类介绍

· Handler

主要函数



a. sendMessage()、post()——都是构造一个Message对象，并放入Looper的MessageQueue中
b. handleMessage(Message msg)——处理出队列的消息

· Looper

a. prepare()——构造线程唯一的Looper对象，并初始化MessageQueue成员
b. myLooper()——获取当前线程唯一的Looper对象
c. loop()——循环从MessageQueue中取出Message对象，并交给Message的Handler成员处理

· MessageQueue

一个以时间为优先级的线程阻塞队列。

a. enqueueMessage(Message msg, long when)——消息入队列
b. next()——消息出队列
c. quit(boolean safe)——终止next()死循环
d. nativePollOnce(long ptr, int timeoutMillis)——线程阻塞，等待消息入队列

· Message

线程间通信的数据载体。

a. target:Handler——保存发送消息的Handler对象
b. setAsynchronous(boolean async)——设置为异步消息

·ThreadLocal

ThreadLocal可以包装一个对象，使其成为线程私有的局部变量，通过 ThreadLocal 的 get 和 set 方法来访问这个线程局部变量，而不受到其他线程的影响。

a. set()——设置线程唯一的对象
b. get()——取出当前线程唯一的对象

Q&A

·ThreadLocal 是什么

ThreadLocal可以包装一个对象，使其成为线程私有的局部变量，通过 ThreadLocal 的 get 和 set 方法来访问这个线程局部变量，而不受到其他线程的影响。ThreadLocal 实现了线程之间的数据隔离，同时提升同一线程中该变量访问的便利性。

ThreadLocal是用来存储指定线程的数据的，当某些数据的作用域是该指定线程并且该数据需要贯穿该线程的所有执行过程时就可以使用ThreadnLocal存储数据，当某线程使用ThreadnLocal存储数据后，只有该线程可以读取到存储的数据，除此线程之外的其他线程是没办法读取到该数据的。

```
ThreadLocal<Boolean> local = new ThreadLocal<>();
// 设置初始值为true.
local.set(true);

Boolean bool = local.get();
Logger.i("MainThread读取的值为: " + bool);//true

new Thread() {
    @Override
    public void run() {
        Boolean bool = local.get();
        Logger.i("SubThread读取的值为: " + bool);//null

        // 设置值为false.
        local.set(false);
    }
}.start():

// 主线程睡1秒，确保上方子线程执行完毕再执行下面的代码。
Thread.sleep(1000);

```

```
Boolean newBool = local.get();
Logger.i("MainThread读取的新值为：" + newBool);//true
```

· Looper死循环为什么不会导致应用卡死

线程即是一段可执行的代码，当可执行代码执行完成后，线程生命周期便该终止了，线程退出。而对于主线程，我们是绝不希望会被运行一段时间，自己就退出。简单做法就是可执行代码是能一直执行下去的，死循环便能保证不会被退出。

真正会卡死主线程的操作是在回调方法onCreate/onStart/onResume等操作时间过长，会导致掉帧，甚至发生ANR，looper.loop本身不会导致应用卡死。

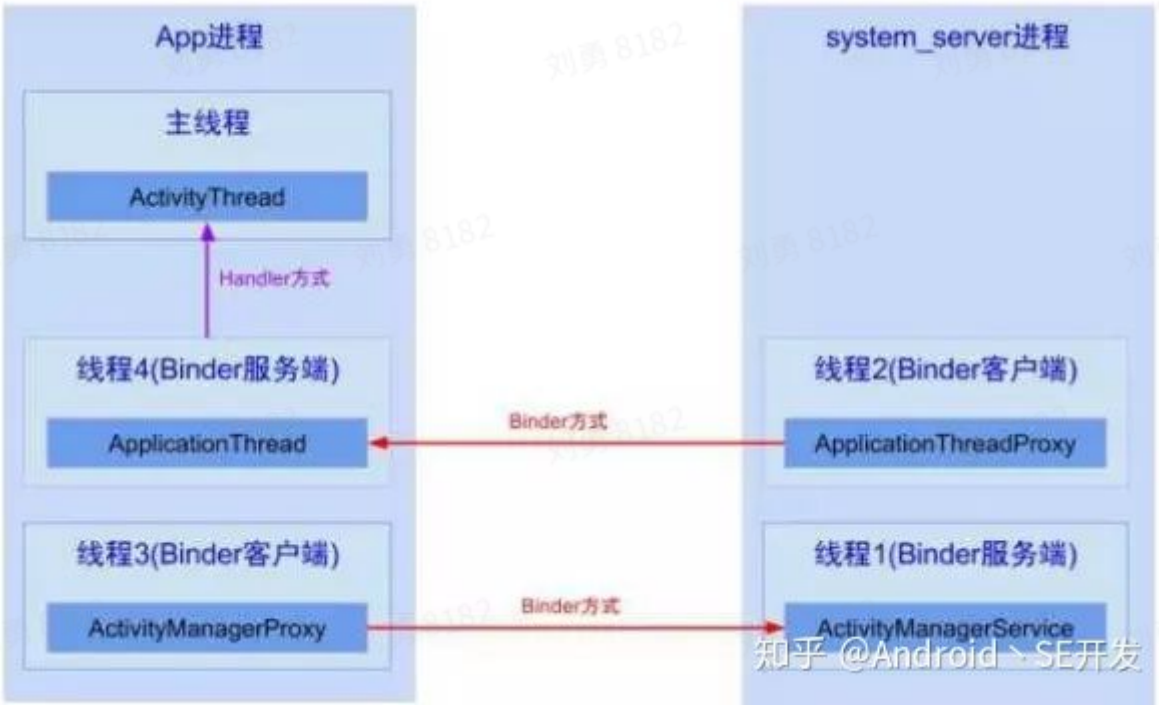
在主线程的MessageQueue没有消息时，便阻塞在loop的queue.next()中的nativePollOnce()方法里，此时主线程会释放CPU资源进入休眠状态，直到下个消息到达或者有事务发生，通过往pipe管道写端写入数据来唤醒主线程工作。

· 主线程的消息循环机制是什么

事实上，会在进入死循环之前便创建了新binder线程，在代码ActivityThread.main()中：

```
public static void main(String[] args) {
    //创建Looper和MessageQueue对象，用于处理主线程的消息
    Looper.prepareMainLooper();
     //创建ActivityThread对象
    ActivityThread thread = new ActivityThread()
    if (sMainThreadHandler == null) {
         sMainThreadHandler = thread.getHandler();
     }
     //建立Binder通道 (创建新线程)
    thread.attach(false);
     Looper.loop(); //消息循环运行
    throw new RuntimeException("Main thread loop unexpectedly exited");
}
```

Activity启动过程：



system_server进程

> system_server进程是系统进程，java framework框架的核心载体，里面运行了大量的系统服务，比如这里提供ApplicationThreadProxy（简称ATP），ActivityManagerService（简称AMS），这个两个服务都运行在system_server进程的不同线程中，由于ATP和AMS都是基于IBinder接口，都是binder线程，binder线程的创建与销毁都是由binder驱动来决定的。

App进程

> App进程则是我们常说的应用程序，主线程主要负责Activity/Service等组件的生命周期以及UI相关操作都运行在这个线程；另外，每个App进程中至少会有两个binder线程 ApplicationThread(简称AT)和ActivityManagerProxy（简称AMP），除了图中画的线程，其中还有很多线程

Binder

Binder用于不同进程之间通信，由一个进程的Binder客户端向另一个进程的服务端发送事务，比如图中线程2向线程4发送事务；而handler用于同一个进程中不同线程的通信，比如图中线程4向主线程发送消息。

Activity的生命周期都是依靠主线程的Looper.loop()

/ActivityThread

```
1 public final class ActivityThread {
2 final H mH = new H();
3 private void sendMessage(int what, ...) {
4     Message msg = Message.obtain();
5     msg.what = what;
```

```
    mH.sendMessage(msg);
}

private class H extends Handler {
    public static final int LAUNCH_ACTIVITY  = 100;
    public static final int PAUSE_ACTIVITY = 101;

    public void handleMessage(Message msg) {
        switch (msg.what) {
            case LAUNCH_ACTIVITY: {
                ...
                handleLaunchActivity(...);
                break;
            }
            case PAUSE_ACTIVITY: {
                ...
                handlePauseActivity(...);
                break;
            }
        }
    }
```

主线程的消息又是哪来的呢？当然是App进程中的其他线程通过Handler发送给主线程
thread.attach(false)方法函数中便会创建一个Binder线程（具体是指ApplicationThread，Binder的服务端，用于接收系统服务AMS发送来的事件），该Binder线程通过Handler将Message发送给主线程。
//ActivityThread.attach

```
private void attach(boolean system, long startSeq) {
            RuntimeInit.setApplicationObject(mAppThread.asBinder());
            final IActivityManager mgr = ActivityManager.getService();
            try {
                mgr.attachApplication(mAppThread, startSeq);
            } catch (RemoteException ex) {
                throw ex.rethrowFromSystemServer();
            }
    }
```

//ActivityManager
其中Singleton是帮助方便开发者实现的单例模式的类，ActivityManager#getService()最终拿到的是ActivityManagerService，既Binder的客户端（C），也就是我们常说的AMS，它在SystemServer#run()时被创建，并且运行一个独立的进程中。

```
public class ActivityManager {
...

public static IActivityManager getService() {
    return IActMngService.get();
}

private static final Singleton<IActivityManager> IActMngService
    = new Singleton<IActivityManager>() {

    @Override
    protected IActivityManager create() {
        IBinder b = ServiceManager.getService(Context.ACTIVITY_SERVICE);
        IActivityManager am = IActivityManager.Stub.asInterface(b);
        return am;
    }
};
```

//ApplicationThread

`ApplicationThread` 是 `Binder` 的服务端，也就是说这里远程方法被调用时，都运行在非主线程中，而是 `Binder` 的线程池的线程中。

```
private class ApplicationThread extends IApplicationThread.Stub {

    @Override
    public final void scheduleLaunchActivity(IBinder token, ...) {
        ...
        sendMessage(H.LAUNCH_ACTIVITY, token, ...);
    }

    public final void scheduleResumeActivity(IBinder token, ...) {
        ...
        sendMessage(H.RESUME_ACTIVITY, token,
    }

    public final void schedulePauseActivity(IBinder token, ...) {
        ...
        sendMessage(H.PAUSE_ACTIVITY, token, ...);
    }

    public final void scheduleStopActivity(IBinder token, ...)  {
        ...
        sendMessage(H.STOP_ACTIVITY, token, ...);
    }
```

```
23
24      public final void scheduleDestroyActivity(IBinder token, ...) {
25          ...
26          sendMessage(H.DESTROY_ACTIVITY, token, ...);
27      }
28
29      ...
30  }
```

> 结合图说说Activity生命周期，比如暂停Activity，流程如下：

1.线程1的AMS中调用线程2的ATP；（由于同一个进程的线程间资源共享，可以相互直接调用，但需要注意多线程并发问题）
2.线程2通过binder传输到App进程的线程4；
3.线程4通过handler消息机制，将暂停Activity的消息发送给主线程；
4.主线程在looper.loop()中循环遍历消息，当收到暂停Activity的消息时，便将消息分发给ActivityThread.H.handleMessage()方法，再经过方法的调用，
5.最后便会调用到Activity.onPause()，当onPause()处理完后，继续循环loop下去。

最后通过《Android开发艺术探索》的一段话总结：

> ActivityThread通过ApplicationThread和AMS进行进程间通讯，AMS以进程间通信的方式完成ActivityThread的请求后会回调ApplicationThread中的Binder方法，然后ApplicationThread会向H发送消息，H收到消息后会将ApplicationThread中的逻辑切换到ActivityThread中去执行，即切换到主线程中去执行，这个过程就是主线程的消息循环模。

ActivityThread的动力是什么

进程

每个app运行时前首先创建一个进程，该进程是由Zygote fork出来的，用于承载App上运行的各种Activity/Service等组件。进程对于上层应用来说是完全透明的，这也是google有意为之，让App程序都是运行在Android Runtime。大多数情况一个App就运行在一个进程中，除非在AndroidManifest.xml中配置Android:process属性，或通过native代码fork进程。

线程

线程对应用来说非常常见，比如每次new Thread().start都会创建一个新的线程。该线程与App所在进程之间资源共享，从Linux角度来说进程与线程除了是否共享资源外，并没有本质的区别，都是一个task_struct结构体，在CPU看来进程或线程无非就是一段可执行的代码，CPU采用CFS调度算法，保证每个task都尽可能公平的享有CPU时间片。

其实承载ActivityThread的主线程就是由Zygote fork而创建的进程。

Handler是如何能够线程切换

线程间是共享资源的，所以Handler处理不同线程问题就只要注意异步情况即可。

Handler创建的时候会采用当前线程的Looper来构造消息循环系统，Looper在哪个线程创建，就跟哪个线程绑定，并且Handler是在他关联的Looper对应的线程中处理消息的。

那么Handler内部如何获取到当前线程的Looper呢——ThreadLocal。ThreadLocal可以在不同的线程中互不干扰的存储并提供数据，通过ThreadLocal可以轻松获取每个线程的Looper。当然需要注意的是

①线程是默认没有Looper的，如果需要使用Handler，就必须为线程创建Looper。

②ActivityThread被创建时就会初始化Looper，这也是在主线程中默认可以使用Handler的原因。

系统为什么不允许多线程中访问UI

这是因为Android的UI控件不是线程安全的，如果在多线程中并发访问可能会导致UI控件处于不可预期的状态，那么为什么系统不对UI控件的访问加上锁机制呢？缺点有两个：

①首先加上锁机制会让UI访问的逻辑变得复杂

②锁机制会降低UI访问的效率，因为锁机制会阻塞某些线程的执行。

所以最简单且高效的方法就是采用单线程模型来处理UI操作。

子线程有哪些更新UI的方法

a. 主线程中定义Handler，子线程通过mHandler发送消息，主线程Handler的handleMessage更新UI。
b. 用Activity对象的runOnUiThread方法。
c. 创建Handler，传入getMainLooper。
d. View.post(Runnable r) 。也是通过Handler实现

```
public boolean post(Runnable action) {
    final AttachInfo attachInfo = mAttachInfo;
    if (attachInfo != null) {
        return attachInfo.mHandler.post(action); //一般情况走这里    }

    // Postpone the runnable until we know on which thread it needs to run.    // Assume that the runnable will be successfully placed after attach.
     getRunQueue().post(action);
    return true;}

```

Looper在哪个线程创建，就跟哪个线程绑定，并且Handler是在他关联的Looper对应的线程中处理消息的。

·子线程是否可以showToast，showDialog

可以，比如下面这样

```
new Thread(new Runnable() {
        @Override
        public void run() {
            Looper.prepare();
            Toast.makeText(MainActivity.this, "run on thread",
Toast.LENGTH_SHORT).show();
            Looper.loop();
        }
    }).start();

```

这样便能在子线程中Toast，不是说子线程…？老样子，我们追根到底看一下Toast内部执行方式。

//Toast

```
/** * Show the view for the specified duration. */public void show() {
    ``````

    INotificationManager service = getService();//从SMgr中获取名为notification的服务
   String pkg = mContext.getOpPackageName();
    TN tn = mTN;
    tn.mNextView = mNextView;

    try {
        service.enqueueToast(pkg, tn, mDuration);//enqueue? 难不成和Handler的队列有
关?   } catch (RemoteException e) {
        // Empty   }}
```

在show方法中，我们看到Toast的show方法和普通UI 控件不太一样，并且也是通过Binder进程间通讯方法执行Toast绘制。这其中的过程就不在多讨论了，有兴趣的可以在NotificationManagerService类中分析。

通过TN 类，可以了解到它是Binder的本地类。在Toast的show方法中，将这个TN对象传给NotificationManagerService就是为了通讯！并且我们也在TN中发现了它的show方法。

```
private static class TN extends ITransientNotification.Stub {//Binder服务端的具体实
现类
    /**   * schedule handleShow into the right thread   */
    @Override
    public void show(IBinder windowToken) {
        mHandler.obtainMessage(0, windowToken).sendToTarget();
    }
```
```

```
7
8
9       final Handler mHandler = new Handler() {
10          @Override
11          public void handleMessage(Message msg) {
12              IBinder token = (IBinder) msg.obj;
13              handleShow(token);
14          }
15      };
16  }
```

看完上面代码，就知道子线程中Toast报错的原因，因为在TN中使用Handler，所以需要创建Looper对象。 那么既然用Handler来发送消息，就可以在handleMessage中找到更新Toast的方法。 在handleMessage看到由handleShow处理。

//Toast的TN类

```
1  public void handleShow(IBinder windowToken) {
2
3              ``````
4              mWM = (WindowManager)context.getSystemService(Context.WINDOW_SERVICE);
5
6              mParams.x = mX;
7              mParams.y = mY;
8              mParams.verticalMargin = mVerticalMargin;
9              mParams.horizontalMargin = mHorizontalMargin;
10             mParams.packageName = packageName;
11             mParams.hideTimeoutMilliseconds = mDuration ==
12                 Toast.LENGTH_LONG ? LONG_DURATION_TIMEOUT :
   SHORT_DURATION_TIMEOUT;
13             mParams.token = windowToken;
14             if (mView.getParent() != null) {
15                 mWM.removeView(mView);
16             }
17             mWM.addView(mView, mParams);//使用WindowManager的addView方法
    trySendAccessibilityEvent();
18         }
19     }
```

看到这里就可以总结一下：

Toast本质是通过window显示和绘制的（操作的是window），而主线程不能更新UI 是因为ViewRootImpl的checkThread方法在Activity维护的View树的行为。

```
1  void checkThread() {
2          if (mThread != Thread.currentThread()) {
3              throw new CalledFromWrongThreadException(
```

```
                    "Only the original thread that created a view hierarchy can
touch its views.");
        }
    }
```

Toast中TN类使用Handler是为了用队列和时间控制排队显示Toast，所以为了防止在创建TN时抛出异常，需要在子线程中使用Looper.prepare();和Looper.loop();（但是不建议这么做，因为它会使线程无法执行结束，导致内存泄露）
Dialog亦是如此。

· 如何处理Handler 使用不当导致的内存泄露?

首先上文在子线程中为了节目效果，使用如下方式创建Looper

```
Looper.prepare();
``````
Looper.loop();
```

实际上这是非常危险的一种做法

在子线程中，如果手动为其创建Looper，那么在所有的事情完成以后应该调用quit方法来终止消息循环，否则这个子线程就会一直处于等待的状态。而如果调用Looper.quit()以后，这个线程就会立刻终止，因此建议不需要的时候终止Looper。(【Looper.myLooper().quit();】)

那么，如果在Handler的**handleMessage**方法中（或者是run方法）处理消息，如果这个是一个延时消息，会一直保存在主线程的消息队列里，并且会影响系统对Activity的回收，造成内存泄露。
总结一下，解决Handler内存泄露主要2点

a. 有延时消息，要在Activity销毁的时候移除Messages
b. 匿名内部类导致的泄露改为匿名静态内部类，并且对上下文或者Activity使用弱引用。

## · HandlerThread

HandlerThread继承Thread，它是一种可以使用Handler的Thread，它的实现也很简单，在run方法中也是通过Looper.prepare()来创建消息队列，并通过Looper.loop()来开启消息循环（与我们手动创建方法基本一致），这样在实际的使用中就允许在HandlerThread中创建Handler了。
由于HandlerThread的run方法是一个无限循环，因此当不需要使用的时候通过quit或者quitSafely方法来终止线程的执行。

HandlerThread的本质也是线程，所以切记关联的Handler中处理消息的handleMessage为子线程。

```
public class HandlerThread extends Thread {

```
```

```
    private Looper mLooper;
    private Handler mHandler;

    public HandlerThread() {
    }

    @Override
    public void run() {
        Looper.prepare();
        mLooper = Looper.myLooper();
        Looper.loop();
    }

    public Looper getLooper() {
        return mLooper;
    }

    public Handler getHandler() {
        if (mHandler == null) {
            mHandler = new Handler(getLooper());
        }
        return mHandler;
    }

    public void quit() {
        if (mLooper != null) {
            mLooper.quit();
        }
    }
}
```

· IntentService

`IntentService`的本质也是四大组件之一的`Service`，它与普通`Service`不同的是，`IntentService#onStart()`执行后会将操作发送到子线程去执行，其内部使用的就是我们上面提到的`HandleThread`

```
public abstract class IntentService extends Service {

    private volatile Looper mServiceLooper;
    private volatile ServiceHandler mServiceHandler;

    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();
        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);
    }

    @Override
    public void onDestroy() {
        mServiceLooper.quit();
    }

    protected abstract void onHandleIntent(@Nullable Intent intent);
}
```

该例中使用HandlerThread的方式与我们上面用的方式一致，先是启动HandlerThread线程，然后使用该子线程的Looper重新实例化了一个Handler，等到onStart()响应时，将操作通过新实例化的Handler发送到HandlerThread这个子线程去操作，由于消息队列在不设置优先级和执行时间时，遵循FIFO的原则，也就是说它是以单线程的形式一个个的执行队列中的任务。

因此，如果我们平常使用的Service中有耗时任务，不妨把它换成IntentService来试试。

· IdleHandler

```
/** * Callback interface for discovering when a thread is going to block * waiting for more messages. */public static interface IdleHandler {
    /**     * Called when the message queue has run out of messages and will now     * wait for more.  Return true to keep your idle handler active, false     * to have it removed.  This may be called if there are still messages     * pending in the queue, but they are all scheduled to be dispatched     * after the current time.     */
    boolean queueIdle();
}
```

根据注释可以了解到，这个接口方法是在消息队列全部处理完成后或者是在阻塞的过程中等待更多的消息的时候调用的，返回值false表示只回调一次，true表示可以接收多次回调。
具体使用如下代码

```
Looper.myQueue().addIdleHandler(new MessageQueue.IdleHandler() {
        @Override
        public boolean queueIdle() {
            return false;
        }
    });
```

· 同步屏障 SyncBarrier 是什么？有什么作用？

Message 分为两类：同步消息、异步消息。在一般情况下，两类消息处理起来没有什么不同。只有在设置了同步屏障后才会有差异。同步屏障从代码层面上看是一个 Message 对象，但是其 target 属性为 null，用以区分普通消息。在 `MessageQueue.next()` 中如果当前消息是一个同步屏障，则跳过后面所有的同步消息，找到第一个异步消息来处理。

发送异步消息有两种方式：

a. 使用Handler包含async参数的构造方法,只要async参数为true，所有的消息都将是异步消息。

```
public Handler(boolean async) {
    this(null, async);
}
```

b. 显示设置Message为异步消息

```
Message msg = mHandler.obtainMessage(...);
msg.setAsynchronous(true);
```

第一种方式最终也是调用msg.setAsynchronous(true);来设置为异步消息的。

我们可以通过 MessageQueue 对象的 `postSyncBarrier()` 发送一个同步屏障，通过 `removeSyncBarrier(token)` 移除同步屏障

```
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;
    }
}
```

可以看出，在实例化Message对象的时候并没有设置它的target成员变量的值，然后随即就根据执行时间把它放到链表的某个位置了，实际上就是链表的开始位置。也就是说，当在消息队列中放入一个target为空的Message的时候，当前Handler的这一套消息机制就开启了同步阻断。

MessageQueue.next()

```
Message next() {
         ...
        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.
                do {
                    prevMsg = msg;
                    msg = msg.next;
                } while (msg != null && !msg.isAsynchronous());
            }
        }
}
```

当开启了同步障碍时，Looper在获取下一个要执行的消息时，会在链表中寻找第一个要执行的异步消息，如果没有找到异步消息，就让当前线程沉睡。

Android 应用框架中为了更快的响应 UI 刷新事件在 `ViewRootImpl.scheduleTraversals()` 中使用了同步屏障

```
void scheduleTraversals() {
    if (!mTraversalScheduled) {
        mTraversalScheduled = true;
        // 设置同步障碍，确保mTraversalRunnable优先被执行
        mTraversalBarrier = mHandler.getLooper().getQueue().postSyncBarrier();
        // 内部通过Handler发送了一个异步消息
        mChoreographer.postCallback(
                Choreographer.CALLBACK_TRAVERSAL, mTraversalRunnable, null);
        if (!mUnbufferedInputDispatch) {
            scheduleConsumeBatchedInput();
        }
        notifyRendererOfFramePending();
        pokeDrawLockIfNeeded();
    }
}
```

上面源码中的任务 mTraversalRunnable 调用了 `performTraversals()` 执行 `measure()`、`layout()`、`draw()` 等方法。为了让 mTraversalRunnable 尽快被执行，在发消息之前调用 MessageQueue 对象的 `postSyncBarrier()` 设置了一个同步屏障。

参考文档

1. https://zhuanlan.zhihu.com/p/73690640
2. https://www.lagou.com/lgeduarticle/7093.html
3. https://yanzhenjie.blog.csdn.net/article/details/89218745

4. https://blog.csdn.net/hfy8971613/article/details/103881609