安卓中的线程、异步任务、Service 与 IntentService

deepwaterooo

2021 年 12 月 19 日

目录

1 Java 创建线程的三种方式总结

- https://blog.csdn.net/u011578734/article/details/110523825

1.1 继承 Thread 类

```
class MyThread extends Thread {
    @Override
        public void run() {
        super.run();
    }
}
private void testThread(){
    Thread thread = new MyThread();
    thread.start();
}
```

- 缺点: Java 的单继承限制，想通过 Thread 实现多线程，就只能继承 Thread 类，不可继承其他类。

1.2 实现 Runnable 接口

- 如果自己的类已经继承了其他类，这时就只能通过实现 Runnable 接口来实现多线程了。
- 不过，继承 Runnable 接口后，想要启动线程，需要把该类的对象作为参数，传递给 Thread 的构造函数，并使用 Thread 类的实例方法 start 来启动。

```
public class TestThread extends A implements Runnable {
    public void run() {
        // todo
    }
}
// 启动线程
TestThread testThread = new TestThread();
Thread thread = new Thread(testThread);
thread.start();
```

- Thread 内部的 run 方法我们可以看到它的实现原理：

```
private Runnable target;
public void run() {
    if (target != null) {
        target.run();
    }
}
```

- target 是我们传递进来的 Runnable 对象,当线程执行时,线程的 run 方法会直接调用 Runnable 对象的 run 方法。

1.3 实现 Callable 接口

- 如果想要执行的线程有返回，怎么处理呢？这时应该使用 Callable 接口了，与 Runnable 相比，Callable 可以有返回值，返回值通过 FutureTask 进行封装。

```
public class MyCallable implements Callable<Integer> {
    public Integer call() {
        return 111;
    }
}
public static void main(String[] args) throws ExecutionException, InterruptedException {
    MyCallable mc = new MyCallable();
    FutureTask<Integer> ft = new FutureTask<>(mc);
    Thread thread = new Thread(ft);
    thread.start();
    System.out.println(ft.get());
}
```

1.4 比较

- 这几种线程创建方式中，实现接口会更好一些，因为：
  - Java 不支持多重继承，因此继承了 Thread 类就无法继承其它类，但是可以实现多个接口。
  - 类可能只要求可执行就行，继承整个 Thread 类开销过大。
  - 另外，如果需要有返回值时，使用 Callable 接口是适合的。

2 Handler 的使用

- Android 中，不允许应用程序在子线程中更新 UI，UI 的处理必须在 UI 线程中进行，这样 Android 定制了一套完善的线程间通信机制——Handler 通信机制。Handler 作为 Android 线程通信方式，高频率的出现在我们的日常开发工作中，我们常用的场景包括：使用异步线程进行网络通信、后台任务处理等，Handler 则负责异步线程与 UI 线程（主线程）之间的交互。
- Android 为了确保 UI 操作的线程安全，规定所有的 UI 操作都必须在主线程（UI 线程）中执行，决定了 UI 线程中不能进行耗时任务，在开发过程中，需要将网络，IO 等耗时任务放在工作线程中执行，工作线程中执行完成后需要在 UI 线程中进行刷新，因此就有了 Handler 进程内线程通信机制，当然 Handler 并不是只能用在 UI 线程与工作线程间的切换，Android 中任何线程间通信都可以使用 Handler 机制。

2.1 UI 线程中使用 Handler

- UI 线程中使用 Handler 非常简单，因为框架已经帮我们初始化好了 Looper, 只需要创建一个 Handler 对象即可，之后便可以直接使用这个 Handler 实例向 UI 线程发消息（子线程—>UI 线程）

```
    private Handler handler = new Handler(){
        @Override
        public void handleMessage(@NonNull Message msg) {
            super.handleMessage(msg);
            //处理消息
        }
    };
    @Override
    protected void onCreate(@Nullable Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_six);
    }
```

- 这种方式会导致 内存泄露 。
- Handler 内存泄漏的两个原因:

- Java 中非静态内部类和匿名内部类会持有外部类的引用

```
// 这是一个外部类 Handler 不会持有外部类引用
// 显然 handleMessage 没地方写了
Handler handler = new Handler();

// 重写 handleMessage 后将得到一个内部类 Handler，以内 handleMessage 是在外部类中实现的
// 它持有外部类引用，可能会引起内存泄漏
Handler handler = new Handler() { // 这是重写了 handleMessage 后的内部类 Handler
  @Override public void handleMessage(Message msg) {
        super.handleMessage(msg);
        switch (msg.what) {
            case 0:
              MLog.i(msg.obj);
              break;
            case 1:
              break;
            default:

        }
    }
};

// 这里 Handler 是一个匿名类，但不是内部类
// Runnable 是一个匿名内部类，持有外部类引用，可能会引起内存泄漏
new Handler().post(new Runnable() {
    @Override public void run() {
      // ...
    }
});
```

- Handler 的生命周期比外部类长
- 我们通过 Handler 发送消息，在 Message 对象中会持有当前 Handler 对象的引用，在 Java 中非静态成员类、内部类、匿名类会持有外部对象的引用（这里在源码中有提到），而 Looper 是线程局部变量，其生命周期与 UI 线程相同，Looper 持有 MessageQueue 的引用，MessageQueue 持有 Message 的引用，当通过 Handler 发送一个延时消息未处理之前用户已经离开当前 Activity，会导致 Activity 不能及时释放而内存泄漏。
- 分析
  - 非静态的内部 Handler 子类、匿名 Handler 子类会持有外部类的引用 (Activity)，而 Handler 可能会因为要等待处理耗时操作导致存活时间超过 Activity，或者消息队列中存在未被 Looper 处理的 Message ，而 Message 会持有 Handler 的引用。于是，在 Activity 退出时，其引用还是被 Handler 持有，导致 Activity 无法被及时回收，造成内存泄露。
  - 非静态的内部 Runnable 子类、匿名 Runnable 子类 post 到任意 Handler 上时，Runnable 其实是 Massage 中的 Callback，持有 Message 引用，如果这个 Massage 在消息队列还没有被处理，那么就会造成 Runnable 一直持有外部类的引用而造成内存泄露。

2.1.1 解决思路

- 通过静态内部类或者外部类来声明 Handler 和 Runnable，然后使用弱引用来拿到外部类的变量。
- 在 Activity/Fragment 销毁的时候请空 MessageQueue 中的消息。

1. 官方推荐的一种

```
private Handler handler = new Handler(new Handler.Callback() {
        @Override
        public boolean handleMessage(@NonNull Message msg) {
            switch (msg.what){
```

```
            case 1:
            //处理子线程发过来的消息
            Toast.makeText(SixActivity.this,(String)msg.obj,Toast.LENGTH_LONG).show();
            Log.d("aa",(String) msg.obj);
            break;

            }
            return false;
        }
    });
```

2. 静态内部类

- 下面的例子实现了子线程（执行 run() 耗时函数的线程）向主线程发送消息

```
public static final int LOAD_COM = 1; // 加载任务的 id 标志

private Handler mHandler = new MyHandler(MainActivity.this); // 在 MainActivity 中，创建了一个 Handler 对象。

private static class MyHandler extends Handler { // MainActivity 中的静态 static 内部类
    private final WeakReference<MainActivity> mActivity; // 持有当前 MainActivity 的 WeakReference
    private MyHandler(MainActivity activity) {
        this.mActivity = new WeakReference(activity);
    }
    @Override public void handleMessage(@NonNull Message msg) { // ui 线程中，负责消息返回的处理逻辑
        super.handleMessage(msg);       // UI 线程中，Handler 对象的 handleMessage 方法负责处理消息的返回
        switch (msg.what){
        case LOAD_COM:
            Log.d("TestHandler", msg.obj.toString());
            MainActivity mainActivity = mActivity.get();
            if (mainActivity != null){
                mainActivity.mTextView.setText(msg.obj.toString());
            }
            break;
        }
    }
};
@Override public void onClick(View v) {
    switch (v.getId()) {
    case R.id.start_load: // 当按钮 start_load 点击时，启动一个后台线程，模拟一个后台加载过程（线程休眠 1 秒）
        new Thread() {
            @Override
            public void run() { // 后台线程中执行的逻辑：这里代码写定义在主线程 MainActivity 中，但实际 run() 函数的
                try {
                    Thread.sleep(1000);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
// 子线程发送消息
                // Message message = new Message();//可以使用 new Message 来创建消息，但是一般不这样使用？
                Message message = Message.obtain(); // 后台任务完成后，使用 Handler 对象的 sendMessage 方法发送消
                message.what = LOAD_COM;
                message.obj = " 我是子线程消息";
                mHandler.sendMessage(message); // 从后台线程中，发送消息给 UI 线程
            }
        }.start();
        break;
    }
}
```

- 主线程给子线程发送消息（UI 线程—> 子线程）

```
public class SixActivity extends AppCompatActivity {
    private Handler handler;
    private Button btn;
    @Override
        protected void onCreate(@Nullable Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_six);
        new MyOneThread().start();      // 子线程创建方式
        btn= findViewById(R.id.dian);
        btn.setOnClickListener(new View.OnClickListener() {
```

```
                    @Override
                        public void onClick(View v) {
                        Message message=Message.obtain();
                        message.what=1;
                        message.obj = " 我是主线程的消息发送给子线程";
                        handler.sendMessage(message); // 封装完数据发送给子线程
                    }
                });
        }
        class MyOneThread extends Thread {
            @Override public void run() {
                // 在子线程中处理消息, 子线程中处理消息, 没有默认的 Loop
                // 由于只有主线程成才默认的 Looper.prepare(), Looper.loop();
                Looper.prepare(); // 创建 Looper: 如果不添加会报错
                handler = new Handler() { // 在子线程中创建消息 Handler
                    @Override
                    public void handleMessage(@NonNull Message msg) {
                        switch (msg.what){
                        case 1:
                        Log.d("aa",(String) msg.obj);
                        break;
                        }
                    }
                };
                // 循环读取 messageQueue
                Looper.loop(); // 如果不添加读取不到消息
            }
        }
    }
```

- 子线程中，也可以使用这个方式来获取 Looper

```
handler = new Handler(Looper.getMainLooper()) {
    @Override
    public void handleMessage(@NonNull Message msg) {
        switch (msg.what) {
        case 1:
        Log.d("aa",(String) msg.obj);
        break;
        }
    }
};
```

- 子线程发送消息到子线程（子线程—> 子线程）

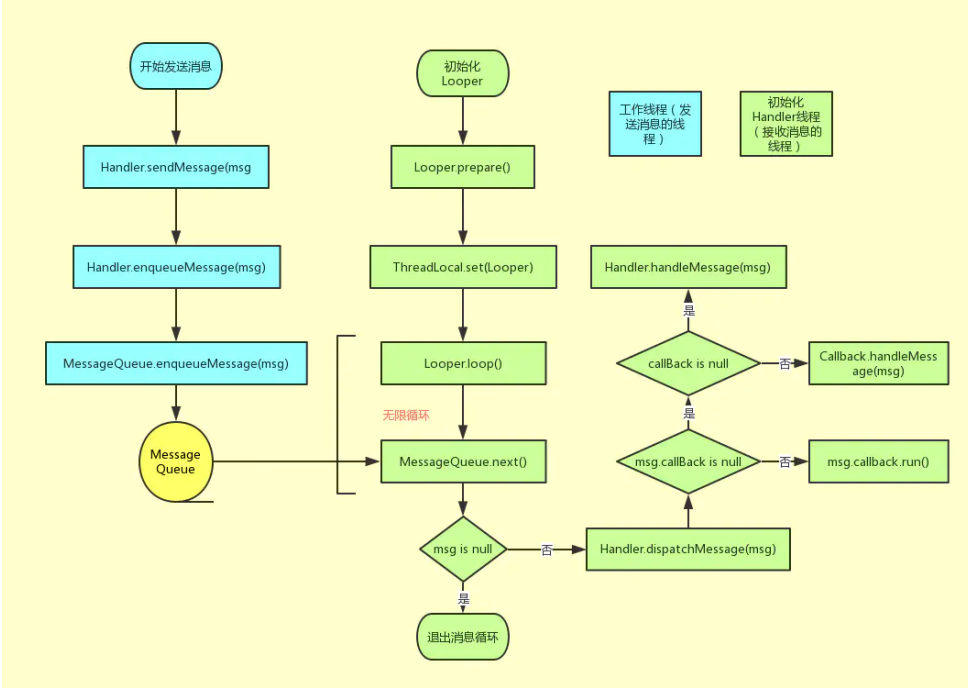
```
btn.setOnClickListener(new View.OnClickListener() {
        @Override public void onClick(View v) {
            new Thread(new Runnable() {
                    @Override
                    public void run() {
                        Message message = Message.obtain();
                        message.obj = " 我是子线程发送到子线消息";
                        message.what = 1;
                        handler.sendMessage(message); // 发送消息的子线程也是有 handler 的
                    }
                }).start();
        }
    });
class MyOneThread extends Thread {
    @Override public void run() {
        //在子线程中处理消息, 子线程中处理消息, 没有默认的 Loop
        //由于只有主线程成才默认的 Looper.prepare(), Looper.loop();
        // Looper.prepare(); // 创建 Looper: 效果一样, 换下面的方式
        handler = new Handler(Looper.getMainLooper()){
            @Override
            public void handleMessage(@NonNull Message msg) {
                switch (msg.what){
                case 1:
                Log.d("aa",(String) msg.obj);
                break;
                }
```

```
            }
        };
        // Looper.loop(); // 循环读取 messageQueue
    }
}
```

- 使用 Handler.post() 直接更新 ui

```
private Handler handler=new Handler();
@Override
protected void onCreate(@Nullable Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_six);
    btn = findViewById(R.id.dian);
    new Thread(new Runnable() {
            @Override
            public void run() {
                // Message message=Message.obtain();
                // message.obj=" 我是子线程静态消息";
                // message.what=1;
                // handler.sendMessage(message);
                handler.post(new Runnable() {
                        @Override
                        public void run() {
                            Log.d("aa"," 直接更新 Ui");
                            btn.setText(" 我是更新的消息");
                        }
                    });
            }
        }).start();
}
```

- post 和 sendMessage 本质上是没有区别的，只是实际用法中有一点差别
- post 也没有独特的作用，post 本质上还是用 sendMessage 实现的，post 只是一中更方便的用法而已



2.2 关于安卓 **handler** 的面试小问题

2.2.1 **Looper** 和 **Handler** 一定要处于一个线程吗？子线程中可以用 **MainLooper** 去创建 **Handler** 吗？

- （1）子线程中

```
Handler handler = new Handler(Looper.getMainLooper()); // 此时, 子线程的 handler 与 Looper.getMainLooper() 主线程 Looper, 两者
```

- 此时两者就不在一个线程中
- （2）子线程中可以用 MainLooper 去创建 Handler.

2.2.2 **Handler** 的 **post** 方法发送的是同步消息吗？可以发送异步消息吗？

- 用户层面发送的都是同步消息
- 不能发送异步消息
- 异步消息只能由系统发送。

2.2.3 **Handler.post** 的逻辑在哪个线程执行的，是由 **Looper** 所在线程还是 **Handler** 所在线程决定的？

- 由 Looper 所在线程决定的
- 最终逻辑是在 Looper.loop() 方法中，从 MsgQueue 中拿出 msg，并且执行其逻辑，这是在 Looper 中执行的，因此是由 Looper 所在的线程决定的。

2.2.4 **Handler** 构造方法中通过 **Looper.myLooper();** 是如何获取到当前线程的 **Looper** 的？

- myLooper() 内部使用 ThreadLocal 实现，因此能够获取各个线程自己的 Looper

2.2.5 **MessageQueue(**消息队列**)**

- 消息队列被封装到 Looper 里面了，我们一般不会直接与 MessageQueue 打交道。我们只需要记住它是用来存放消息的单链表结构。队列的顺序由 Message 的 next 属性来维护。MessageQueue 是整个 Handler 机制的核心，里面涉及很多特性我们这里都不展开讲述 (比如消息屏障机制)。

2.3 **handler** 工作原理总结：**Handler** 的工作原理

- Handler 机制整体流程；
  - IdHandler(闲时机制）；
  - postDelay() 的具体实现；
  - post() 与 sendMessage() 区别；
  - 使用 Handler 需要注意什么问题，怎么解决的?
- 问题很细，能准备多详细就准备多详细。人家自己封装了一套 Handler 来避免内存泄漏问题
- Handler 的消息传递机制涉及到四个部分：
  - 1. Message：线程间传递的对象。

  - 2. MessageQueue：消息队列，用来存放 Handler 发布的 Message.
  - 3. Handler：负责将 Message 插入到 MessageQueue 中以及对 MessageQueue 中的 Message 进行处理。
  - 4. Looper：负责从 MessageQueue 中取出 Message，并交给 Handler.
- 其中：
  - Looper 存储在 ThreadLocal 中，Looper 在创建时会同时创建 MessageQueue，作为其成员对象. 因此 Looper 和 MessageQueue 是属于创建者线程的，各线程之间的 Looper 和 MessageQueue 相互独立。
  - Handler 在创建时会从当前线程的 ThreadLocal 中取得 Looper.
  - 发送消息时，在发送线程中调用接收线程中的 Handler 的 sendMessage 方法，过程中，Handler 会将自身赋予到 Message 的 target 中，并将 Message 插入到 Handler 对应的 MessageQueue 中。
  - 而接收线程中的 Looper 在循环过程中会取出这个 Message，通过 Message.target 取出接收线程中的 Handler，并将消息交 Handler 对象处理。由此实现了跨线程通信。
  - 要注意的是：线程与 Looper 和 MessageQueue 是一对一的关系，即一个线程只维护一个 Looper 和一个 MessageQueue; 而线程与 Handler 的关系是一对多，即一个线程可以有很多 Handler，一个 Handler 只对应一个线程，这也是为什么 Handler 在发送消息时，为什么要将自身赋给 Message.target 的原因。
- Handler 内存泄露的解决方法
  - 方法 1：通过程序逻辑进行保护。
    - 关闭 Activity 的时候停掉后台线程，这样就相当于切断了 Handler 和外部连接的线，Activity 自然会在合适的时候被回收。
    - 如果你的 Handler 是被 delay 的 Message 持有了引用，那么在 Activity 销毁前使用相应的 Handler 的 removeCallbacksAndMessages() 方法，把消息对象从消息队列移除就行了。
  - 方法 2：将 Handler 声明为静态类
    - 静态类不持有外部类的对象，这样即使 Handler 在运行，Activity 也可以被回收。
    - 由于静态类的 Handler 不再持有外部类对象，如果要操作 Activity 需要增加一个 Activity 的弱引用。
- 优点：
  - 操作简单，无学习成本。
- 缺点：
  - 代码规范性较差，不易维护。
  - 每次操作都会开启一个匿名线程，系统开销较大。

2.3.1 Looper.loop() 为什么不会阻塞主线程?

- https://segmentfault.com/a/1190000037449196 这部分格式再整理一下

主线程 Looper 从消息队列读取消息，当读完所有消息时，主线程阻塞。子线程往消息队列发送消息，并且往管道文件写数据，主线程即被唤醒，从管道文件读取数据，主线程被唤醒只是为了读取消息，当消息读取完毕，再次睡眠。因此 loop 的循环并不会对 CPU 性能有过多的消耗。

主线程中如果没有 looper 进行循环，那么主线程一运行完毕就会退出。那么我们还能运行 APP 吗，显然，这是不可能的，Looper 主要就是做消息循环，然后由 Handler 进行消息分发处理，一旦退出消息循环，那么你的应用也就退出了。

总结：Looper 的无限循环必不可少。

补充说明：

我看有一部分人理解”Looper.loop() 的阻塞“和”UI 线程上执行耗时操作卡死“的区别时还一脸懵逼的状况，简单回答一波：

首先这两之间一点联系都没有，完全两码事。Looper 上的阻塞，前提是没有输入事件，MsgQ 为空，Looper 空闲状态，线程进入阻塞，释放 CPU 执行权，等待唤醒。UI 耗时导致卡死，前提是要有输入事件，MsgQ 不为空，Looper 正常轮询，线程并没有阻塞，但是该事件执行时间过长（5 秒?），而且与此期间其他的事件（按键按下，屏幕点击..）都没办法处理（卡死），然后就 ANR 异常了。

2.3.2 Android – Looper.prepare() 和 Looper.loop() —深入版

Android 中的 Looper 类，是用来封装消息循环和消息队列的一个类，用于在 android 线程中进行消息处理。handler 其实可以看做是一个工具类，用来向消息队列中插入消息的。

(1) Looper 类用来为一个线程开启一个消息循环。默认情况下 android 中新诞生的线程是没有开启消息循环的。（主线程除外，主线程系统会自动为其创建 Looper 对象，开启消息循环。）Looper 对象通过 MessageQueue 来存放消息和事件。一个线程只能有一个 Looper，对应一个 MessageQueue。

(2) 通常是通过 Handler 对象来与 Looper 进行交互的。Handler 可看做是 Looper 的一个接口，用来向指定的 Looper 发送消息及定义处理方法。默认情况下 Handler 会与其被定义时所在线程的 Looper 绑定，比如，Handler 在主线程中定义，那么它是与主线程的 Looper 绑定。mainHandler = new Handler() 等价于 new Handler（Looper.myLooper()）. Looper.myLooper()：获取当前进程的 looper 对象，类似的 Looper.getMainLooper() 用于获取主线程的 Looper 对象。

(3) 在非主线程中直接 new Handler() 会报如下的错误:

E/AndroidRuntime(6173): Uncaught handler: thread Thread-8 exiting due to uncaught exception E/AndroidRuntime(6173): java.lang.RuntimeException: Can’t create handler inside thread that has not called Looper.prepare()

原因是非主线程中默认没有创建 Looper 对象，需要先调用 Looper.prepare() 启用 Looper。

(4) Looper.loop();

让 Looper 开始工作，从消息队列里取消息，处理消息。

注意:写在 Looper.loop() 之后的代码不会被执行,这个函数内部应该是一个循环,当调用 mHandler.getLooper().quit() 后，loop 才会中止，其后的代码才能得以运行。(5) 基于以上知识，可实现主线程给子线程（非主线程）发送消息。

2.3.3 线程的切换又是怎么回事？

那么线程的切换又是怎么回事呢？很多人搞不懂这个原理，但是其实非常简单，我们将所涉及的方法调用栈画出来，如下：

Thread.foo(){ Looper.loop() -> MessageQueue.next() -> Message.target.dispatchMessage() -> Handler.handleMessage() } 显而易见,Handler.handleMessage() 所在的线程最终由调用 Looper.loop() 的线程所决定。

平时我们用的时候从异步线程发送消息到 Handler，这个 Handler 的 handleMessage() 方法是在主线程调用的，所以消息就从异步线程切换到了主线程。

2.3.4 Handler 是如何实现线程之间的切换的

Handler 是如何实现线程之间的切换的呢？例如现在有 A、B 两个线程，在 A 线程中有创建了 handler，然后在 B 线程中调用 handler 发送一个 message。

通过上面的分析我们可以知道，当在 A 线程中创建 handler 的时候，同时创建了 MessageQueue 与 Looper，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。

image

小结：

1.Handler 初始化之前，Looper 必须初始化完成。UI 线程之所以不用初始化，因为在 ActivityThread 已经初始化，其他子线程初始化 Handler 时，必须先调用 Looper.prepare()。

2. 通过 Handler 发送消息时，消息会回到 Handler 初始化的线程，而不一定是主线程。

3. 使用 ThreadLocal 时，需要注意内存泄漏的问题。

通俗点的说法 Handler 机制其实就是借助共享变量来进行线程切换的.

Handler 是如何实现线程之间的切换的

妙用 Looper 机制

我们可以利用 Looper 的机制来帮助我们做一些事情：

将 Runnable post 到主线程执行；利用 Looper 判断当前线程是否是主线程。完整示例代码如下：

public final class MainThread {

private MainThread() { }

private static final Handler HANDLER = new Handler(Looper.getMainLooper());

public static void run(@NonNull Runnable runnable) { if (isMainThread()) { runnable.run(); }else{ HANDLER.post(runnable); } }

public static boolean isMainThread() { return Looper.myLooper() == Looper.getMainLooper(); }

} 能够省去不少样板代码。先明确我们的问题：

Handler 是如何与线程关联的？Handler 发出去的消息是谁管理的？消息又是怎么回到 handleMessage() 方法的？线程的切换是怎么回事？回答：Handler 发送的消息由 MessageQueue 存储管理，并由 Loopler 负责回调消息到 handleMessage()。

线程的转换由 Looper 完成，handleMessage() 所在线程由 Looper.loop() 调用者所在线程决定。

2.3.5 为什么在子线程中创建 **Handler** 会抛异常

- Handler 的工作是依赖于 Looper 的，而 Looper(与消息队列) 又是属于某一个线程 (ThreadLocal 是线程内部的数据存储类，通过它可以在指定线程中存储数据，其他线程则无法获取到)，其他线程不能访问。因此 Handler 就是间接跟线程是绑定在一起了。因此要使用 Handler 必须要保证 Handler 所创建的线程中有 Looper 对象并且启动循环。因为子线程中默认是没有 Looper 的，所以会报错。正确的使用方法是：

```
public class WorkThread extends Thread {
    private Handler mHander;
    public Handler getHander() {
        return mHander;
    }
    public void quit() { // 这里是资源释放吗?
        mHander.getLooper().quit();
    }
    @Override
        public void run() {
        super.run();

        // 创建该线程对应的 Looper,
        // 内部实现
        //   1。new Looper()
```

```
        //   2。将 1 步中的 lopper 放在 ThreadLocal 里，ThreadLocal 是保存数据的，主要应用场景是：线程间数据互不影响的情况
        //   3。在 1 步中的 Looper 的构造函数中 new MessageQueue();
        // 对消息机制不懂得同学可以查阅资料，网上很多也讲的很不错。
        Looper.myLooper(); // 一般是 call Looper.prepare(); 吧，再查一下

        mHander = new Handler(){
            @SuppressLint("HandlerLeak")
            @Override
            public void handleMessage(Message msg) {
                super.handleMessage(msg);
                Log.d("WorkThread", (Looper.getMainLooper() == Looper.myLooper()) + "," + msg.what);
            }
        };

        // 注意这 3 个的顺序不能颠倒
        Looper.loop();
    }
}
```

2.3.6 Handler 发送延时消息

- handler 发送延时消息是通过 postDelayed() 方法将 Runnanle 对象封装成 Message，然后调用 sendMessageAtTime()，设置的时间是当时的时间 + 延时的时间。
- 发送延时消息实际上是往 messageQueue 中加入一条 Message。
- Message 在 MessageQueue 中实际是以单链表来存储的，且是按照时间顺序来插入的。时间顺序是以 Message 中的 when 属性来排序的。
- 重点：
  - postDelay 并不是等待 delayMillis 延时时常后再加入消息队列，而是加入消息队列后阻塞 (消息队列会按照阻塞时间排序) 等待 delayMillis 后唤醒消息队列再执行。
  - sleep 会阻塞线程
  - postDelayed 不会阻塞线程

2.3.7 Handler 线程间通信

- 作用：线程之间的消息通信
- 流程：主线程默认实现了 Looper （调用 loop.prepare 方法向 sThreadLocal 中 set 一个新的 looper 对象，looper 构造方法中又创建了 MsgQueue）手动创建 Handler，调用 sendMessage 或者 post (runable) 发送 Message 到 msgQueue ，如果没有 Msg 这添加到表头，有数据则判断 when 时间循环 next 放到合适的 msg 的 next 后。Looper.loop 不断轮训 Msg，将 msg 取出并分发到 Handler 或者 post 提交的 Runable 中处理，并重置 Msg 状态位。回到主线程中重写 Handler 的 handlerMessage 回调的 msg 进行主线程绘制逻辑。
- 问题：
  - Handler 同步屏障机制：通过发送异步消息，在 msg.next 中会优先处理异步消息，达到优先级的作用
  - Looper.loop 为什么不会卡死：为了 app 不挂掉，就要保证主线程一直运行存在，使用死循环代码阻塞在 msgQueue.next() 中的 nativePollOnce() 方法里，主线程就会挂起休眠释放 cpu，线程就不会退出。Looper 死循环之前，在 ActivityThread.main() 中就会创建一个 Binder 线程（ApplicationThread），接收系统服务 AMS 发送来的事件。当系统有消息产生（其实系统每 16ms 会发送一个刷新 UI 消息唤醒）会通过 epoll 机制向 pipe 管道写端写入数据就会发送消息给 looper 接收到消息后处理事件，保证主线程的一直存活。只有在主线程中处理超时才会让 app 崩溃也就是 ANR。

- Messaage 复用：将使用完的 Message 清除附带的数据后，添加到复用池中，当我们需要使用它时，直接在复用池中取出对象使用，而不需要重新 new 创建对象。复用池本质还是 Message 为 node 的单链表结构。所以推荐使用 Message.obation 获取对象。

3 AsyncTask

- 较为轻量级的异步类，封装了 FutureTask 的线程池、ArrayDeque 和 Handler 进行调度。AsyncTask 主要用于 后台与界面持续交互
- 我们来看看 AsyncTask 这个抽象类的定义，当我们定义一个类来继承 AsyncTask 这个类的时候，我们需要为其指定 3 个泛型参数：

```
AsyncTask   <Params, Progress, Result>
```

- Params: 这个泛型指定的是我们传递给异步任务执行时的参数的类型。
- Progress: 这个泛型指定的是我们的异步任务在执行的时候将执行的进度返回给 UI 线程的参数的类型。
- Result: 这个泛型指定的异步任务执行完后返回给 UI 线程的结果的类型。
- 我们在定义一个类继承 AsyncTask 类的时候，必须要指定好这三个泛型的类型，如果都不指定的话，则都将其写成 void。
- 我们来看一个官方给的例子：

```
private class DownloadFilesTask extends AsyncTask<URL, Integer, Long> {
     protected Long doInBackground(URL... urls) {
         int count = urls.length;
         long totalSize = 0;
         for (int i = 0; i < count; i++) {
             totalSize += Downloader.downloadFile(urls[i]);
             publishProgress((int) ((i / (float) count) * 100));
             // Escape early if cancel() is called
             if (isCancelled()) break;
         }
         return totalSize;
     }
     protected void onProgressUpdate(Integer... progress) {
         setProgressPercent(progress[0]);
     }
     protected void onPostExecute(Long result) {
         showDialog("Downloaded " + result + " bytes");
     }
}
```

- 使用时只需要集成 AsyncTask，创建对象并调用 execute 执行即可：

```
new DownloadFilesTask().execute(url1, url2, url3);
```

- doInBackground(Params…) 方法里执行耗时逻辑，然后在 onPostExecute(Result) 中将结果更新回 UI 组件
- AsyncTask 的几个主要方法中，doInBackground 方法运行在子线程，execute、onPreExecute、onProgressUpdate、onPostExecute 这几个方法都是在 UI 线程运行的。

3.1 使用 **AsyncTask** 的注意事项

- AsyncTask 的实例必须在 UI Thread 中创建。
- 只能在 UI 线程中调用 AsyncTask 的 execute 方法。
- AsyncTask 被重写的四个方法是系统自动调用的, 不应手动调用。
- 每个 AsyncTask 只能被执行一次，多次执行会引发异常。
- AsyncTask 的四个方法，只有 doInBackground 方法是运行在其他线程中, 其他三个方法都运行在 UI 线程中，也就说其他三个方法都可以进行 UI 的更新操作。
- AsyncTask 默认是串行执行，如果需要并行执行，使用接口 executeOnExecutor 方法。

3.2 优点：

- 结构清晰，使用简单，适合后台任务的交互。
- 异步线程的优先级已经被默认设置成了：$\text{THREAD}_{PRIORITY\,BACKGROUND}$ UI 线程抢占资源。

3.3 缺点：

- 结构略复杂，代码较多。
- 每个 AsyncTask 只能被执行一次，多次调用会发生异常。
- AsyncTask 在整个 Android 系统中维护一个线程池，有可能被其他进程的任务抢占而降低效率。

4 HandlerThread

- HandlerThread 是一个自带 Looper 消息循环的线程类。处理异步任务的方式和 Thread + Looper + Handler 方式相同。

4.1 优点：

- 简单，内部实现了普通线程的 Looper 消息循环。
- 可以串行执行多个任务。
- 内部拥有自己的消息队列，不会阻塞 UI 线程。

4.2 缺点：

- 没有结果返回接口，需要自行处理。
- 消息过多时，容易造成阻塞。
- 只有一个线程处理，效率较低。
- 线程优先级默认优先级为 $\text{THREAD}_{PRIORITY\,DEFAULT}$ UI 线程抢占资源。

5 IntentService

- IntentService 继承自 Service 类,用于启动一个异步服务任务,它的内部是通过 HandlerThread 来实现异步处理任务的。
- 我们来看下 IntentService 的主要方法：

```
// mServiceLooper;
// mServiceHandler;
@Override
public void onCreate() {
    // TODO: It would be nice to have an option to hold a partial wakelock during processing,
    // and to have a static startService(Context, Intent) method that would launch the service & hand off a wakelock.
    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);
}
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);
    }
}
```

5.1 优点：

- 只需要继承 IntentService，就可以在 onHandlerIntent 方法中异步处理 Intent 类型任务了。
- 任务结束后 IntentService 会自行停止，无需手动调用 stopService。
- 可以执行处理多个 Intent 请求，顺序执行多任务。
- IntentService 是继承自 Service，具有后台 Service 的优先级。

5.2 缺点：

- 需要启动服务来执行异步任务，不适合简单任务处理。
- 异步任务是由 HandlerThread 实现的，只能单线程、顺序处理任务。
- 没有返回 UI 线程的接口。

6 使用线程池来处理异步任务

- 利用 Executors 的静态方法 newCachedThreadPool()、newFixedThreadPool()、newSingleThreadExecutor() 及重载形式实例化 ExecutorService 接口即得到线程池对象。
- 动态线程池 newCachedThreadPool()：根据需求创建新线程的，需求多时，创建的就多，需求少时，JVM 自己会慢慢的释放掉多余的线程。

- 固定数量的线程池 newFixedThreadPool()：内部有个任务阻塞队列，假设线程池里有 2 个线程，提交了 4 个任务，那么后两个任务就放在任务阻塞队列了，即使前 2 个任务 sleep 或者堵塞了，也不会执行后两个任务，除非前 2 个任务有执行完的。
- 单线程 newSingleThreadExecutor()：单线程的线程池，这个线程池可以在线程死后（或发生异常时）重新启动一个线程来替代原来的线程继续执行下去。

6.1 优点：

- 线程的创建和销毁由线程池来维护，实现了线程的复用，从而减少了线程创建和销毁的开销。
- 适合执行大量异步任务，提高性能。
- 灵活性高，可以自由控制线程数量。
- 扩展性好，可以根据实际需要进行扩展。

6.2 缺点：

- 代码略显复杂。
- 线程池本身对系统资源有一定消耗。
- 当线程数过多时，线程之间的切换成本会有很大开销，从而使性能严重下降。
- 每个线程都会耗费至少 1040KB 内存，线程池的线程数量需要控制在一定范围内。
- 线程的优先级具有继承性，如果在 UI 线程中创建线程池，线程的默认优先级会和 UI 线程相同，从而对 UI 线程使用资源进行抢占。