安卓中的线程、异步任务、Service 与 IntentService

deepwaterooo

2021 年 12 月 17 日

目录

1 Java 创建线程的三种方式总结

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 。
  - 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。

2.1.1 解决思路

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）

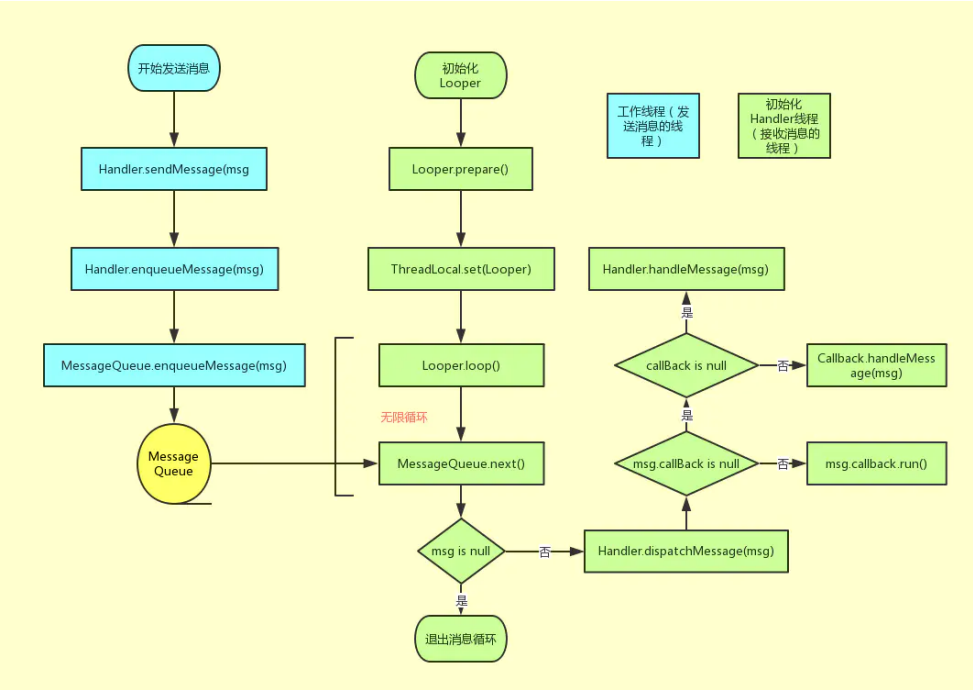
```
btn.setOnClickListener(new View.OnClickListener() {
        @Override public void onClick(View v) {
            new Thread(new Runnable() {
                    @Override
                    public void run() {
                        Message message = Message.obtain();
                        message.obj = " 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 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。