安卓中的线程、异步任务、Service 与 IntentService

deepwaterooo

September 18, 2022

Contents

1 Java 创建线程的三种方式总结

- 关于线程、异步任务，多进程等等：纸上得来终觉浅，绝知此事要躬行
- 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 提供了四种常用的操作多线程的方式，分别是：
  - 1. Handler+Thread
  - 2. AsyncTask
  - 3. ThreadPoolExecutor
  - 4. IntentService
- 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);
}
```

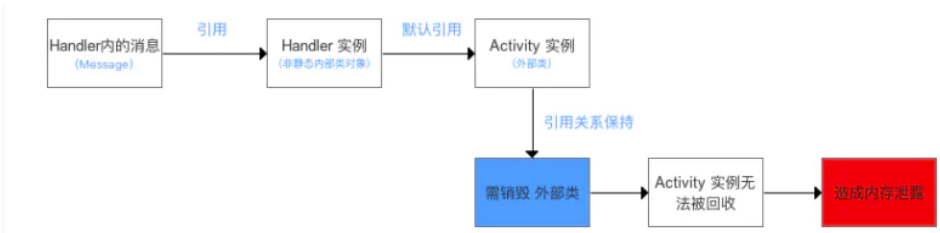
- 这种方式会导致内存泄露 。

2.1.1 泄露原因讲解

- 储备知识
  - 主线程的 Looper 对象的生命周期 = 该应用程序的生命周期
  - 在 Java 中，非静态内部类 & 匿名内部类都默认持有外部类的引用

1. 泄露原因描述
   - 上述的 Handler 实例的消息队列有 2 个分别来自线程 1、2 的消息（分别延迟 1s、6s）
   - 在 Handler 消息队列还有未处理的消息 / 正在处理消息时，消息队列中的 Message 持有 Handler 实例的引用
   - 由于 Handler = 非静态内部类 / 匿名内部类（2 种使用方式），故又默认持有外部类的引用（即 MainActivity 实例），引用关系如下图：

   - 上述的引用关系会一直保持，直到 Handler 消息队列中的所有消息被处理完毕。在 Handler 消息队列还有未处理的消息 / 正在处理消息时，此时若需销毁外部类 MainActivity，但由于上述引用关系，垃圾回收器（GC）无法回收 MainActivity，从而造成内存泄漏。如下图：



2. 总结
   - 当 Handler 消息队列还有未处理的消息 / 正在处理消息时，存在引用关系：“未被处理 / 正处理的消息 -> Handler 实例 -> 外部类”
   - 若出现 Handler 的生命周期 > 外部类的生命周期时（即 Handler 消息队列还有未处理的消息 / 正在处理消息而外部类需销毁时），将使得外部类无法被垃圾回收器（GC）回收，从而造成内存泄露

2.1.2 解决方案

- 从上面可看出，造成内存泄露的原因有 2 个关键条件：
  - 存在“未被处理 / 正处理的消息 -> Handler 实例 -> 外部类”的引用关系
  - Handler 的生命周期 > 外部类的生命周期
    - 即 Handler 消息队列还有未处理的消息 / 正在处理消息而外部类需销毁
- 解决方案的思路 = 使得上述任 1 条件不成立即可。

1. 解决方案 1：静态内部类
   - 原理：静态内部类不默认持有外部类的引用，从而使得“未被处理 / 正处理的消息 -> Handler 实例 -> 外部类”的引用关系不存在。

- 具体方案：将 Handler 的子类设置成静态内部类。此外，还可使用 WeakReference 弱引用持有外部类，保证外部类能被回收。因为：弱引用的对象拥有短暂的生命周期，在垃圾回收器线程扫描时，一旦发现了具有弱引用的对象，不管当前内存空间足够与否，都会回收它的内存

```
public class MainActivity extends AppCompatActivity {
    public static final String TAG = "carson: ";
    private Handler showhandler;
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        // 实例化自定义的 Handler 类对象->> 分析 1
        // 注：
            // a. 此处并无指定 Looper，故自动绑定当前线程 (主线程) 的 Looper、MessageQueue;
            // b. 定义时需传入持有的 Activity 实例 (弱引用)
        showhandler = new FHandler(this);
        new Thread() {
            @Override
            public void run() {
                try {
                    Thread.sleep(1000);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
                // a. 定义要发送的消息
                Message msg = Message.obtain();
                msg.what = 1;// 消息标识
                msg.obj = "AA";// 消息存放
                showhandler.sendMessage(msg);
            }
        }.start();
    }
    // 设置为：静态内部类
    private static class FHandler extends Handler{
        // 定义 弱引用实例
        private WeakReference<Activity> reference;
        // 在构造方法中传入需持有的 Activity 实例
        public FHandler(Activity activity) {
            // 使用 WeakReference 弱引用持有 Activity 实例
            reference = new WeakReference<Activity>(activity); }
        // 通过复写 handlerMessage() 从而确定更新 UI 的操作
        @Override
        public void handleMessage(Message msg) {
            switch (msg.what) {
                case 1:
                    Log.d(TAG, " 收到线程 1 的消息");
                    break;
                case 2:
                    Log.d(TAG, " 收到线程 2 的消息");
                    break;

            }
        }
    }
}
```

2. 解决方案 2：当外部类结束生命周期时，清空 Handler 内消息队列

- 原理：不仅使得“未被处理 / 正处理的消息 -> Handler 实例 -> 外部类”的引用关系不复存在，同时使得 Handler 的生命周期（即消息存在的时期）与外部类的生命周期同步
- 具体方案：当外部类（此处以 Activity 为例）结束生命周期时（此时系统会调用 onDestroy()），清除 Handler 消息队列里的所有消息（调用 removeCallbacksAndMessages(null)）

```
@Override
    protected void onDestroy() {
        super.onDestroy();
        mHandler.removeCallbacksAndMessages(null);
        // 外部类 Activity 生命周期结束时，同时清空消息队列 & 结束 Handler 生命周期
    }
```

- 使用建议

  - 为了保证 Handler 中消息队列中的所有消息都能被执行，此处推荐使用解决方案 1 解决内存泄露问题，即静态内部类 + 弱引用的方式

```
// 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。

2.1.3 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）

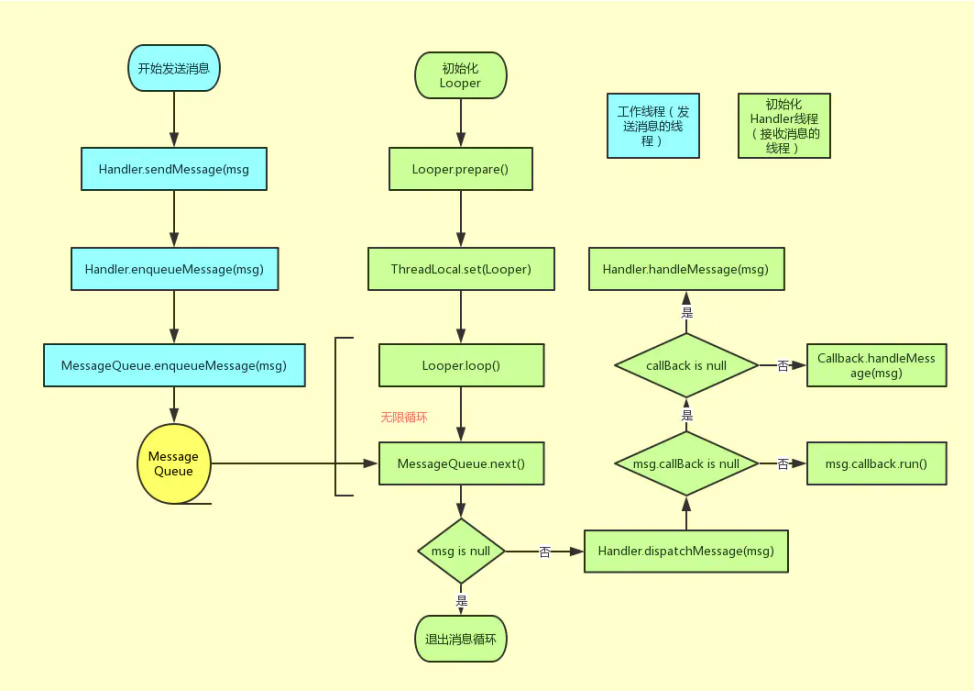
```
btn.setOnClickListener(new View.OnClickListener() {
        @Override public void onClick(View v) {
            new Thread(new Runnable() {
                    @Override
                    public void run() {
                        Message message = Message.obtain();
                        message.obj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

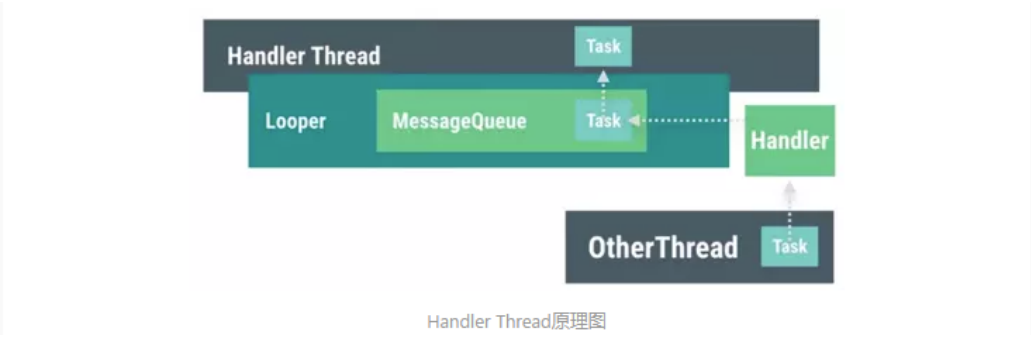
- sleep 会阻塞线程
- postDelayed 不会阻塞线程

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。

2.4 handler 稍带总结

- Android 主线程包含一个消息队列 (MessageQueue)，该消息队列里面可以存入一系列的 Message 或 Runnable 对象。通过一个 Handler 你可以往这个消息队列发送 Message 或者 Runnable 对象，并且处理这些对象。每次你新创建一个 Handle 对象，它会绑定于创建它的线程 (也就是 UI 线程) 以及该线程的消息队列，从这时起，这个 handler 就会开始把 Message 或 Runnable 对象传递到消息队列中，并在它们出队列的时候执行它们。



Handler Thread原理图

- Handler 可以把一个 Message 对象或者 Runnable 对象压入到消息队列中，进而在 UI 线程中获取 Message 或者执行 Runnable 对象，Handler 把压入消息队列有两类方式，Post 和 sendMessage：

2.4.1 Post 方式：

- Post 允许把一个 Runnable 对象入队到消息队列中。它的方法有：
- post(Runnable)/postAtTime(Runnable,long)/postDelayed(Runnable,long)
- 对于 Handler 的 Post 方式来说，它会传递一个 Runnable 对象到消息队列中，在这个 Runnable 对象中，重写 run() 方法。一般在这个 run() 方法中写入需要在 UI 线程上的操作。

```
public class MyThread implements Runnable {

    @Override
    public void run() {
        // 下载一个图片
        HttpClient httpClient = new DefaultHttpClient();
        HttpGet httpGet = new HttpGet(image_path);
        HttpResponse httpResponse = null;
        try {
            httpResponse = httpClient.execute(httpGet);
            if (httpResponse.getStatusLine().getStatusCode() == 200) {
                byte[] data = EntityUtils.toByteArray(httpResponse
                        .getEntity());
                // 得到一个Bitmap对象，并且为了使其在post内部可以访问，必须声明为final
                final Bitmap bmp=BitmapFactory.decodeByteArray(data, 0, data.length);
                handler.post(new Runnable() {
                    @Override
                    public void run() {
                        // 在Post中操作UI组件ImageView
                        ivImage.setImageBitmap(bmp);
                    }
                });
                // 隐藏对话框
                dialog.dismiss();
            }
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
```

2.4.2 sendMessage：

- sendMessage 允许把一个包含消息数据的 Message 对象压入到消息队列中。它的方法有：sendEmptyMessage(int)/sendMessage(Message)/sendMessageAtTime(Message,long)/sendMessa
- Handler 如果使用 sendMessage 的方式把消息入队到消息队列中，需要传递一个 Message 对象，而在 Handler 中，需要重写 handleMessage() 方法，用于获取工作线程传递过来的消息，此方法运行在 UI 线程上。Message 是一个 final 类，所以不可被继承。

```
private  Handler handler = new Handler() {
    // 在Handler中获取消息，重写handleMessage()方法
    @Override
    public void handleMessage(Message msg) {
        // 判断消息码是否为1
        if(msg.what==IS_FINISH){
            byte[] data=(byte[])msg.obj;
            Bitmap bmp=BitmapFactory.decodeByteArray(data, 0, data.length);
            ivImage.setImageBitmap(bmp);
            dialog.dismiss();
        }
    }
};
```

```
public class MyThread implements Runnable {

    @Override
    public void run() {
        HttpClient httpClient = new DefaultHttpClient();
        HttpGet httpGet = new HttpGet(image_path);
        HttpResponse httpResponse = null;
        try {
            httpResponse = httpClient.execute(httpGet);
            if (httpResponse.getStatusLine().getStatusCode() == 200) {
                byte[] data = EntityUtils.toByteArray(httpResponse
                        .getEntity());
                // 获取一个Message对象，设置what为1
                Message msg = Message.obtain();
                msg.obj = data;
                msg.what = IS_FINISH;
                // 发送这个消息到消息队列中
                handler.sendMessage(msg);
            }
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

- 优缺点
  - 1. Handler 用法简单明了，可以将多个异步任务更新 UI 的代码放在一起，清晰明了
  - 2. 处理单个异步任务代码略显多
- 适用范围
  - 1. 多个异步任务的更新 UI

3 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。

3.1 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。

3.2 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 THREAD_PRIORITY_DEFAULT，容易和 UI 线程抢占资源。

4 ThreadPoolExecutor

- ThreadPoolExecutor 提供了一组线程池，可以管理多个线程并行执行。这样一方面减少了每个并行任务独自建立线程的开销，另一方面可以管理多个并发线程的公共资源，从而提高了多线程的效率。所以 ThreadPoolExecutor 比较适合一组任务的执行。Executors 利用工厂模式对 ThreadPoolExecutor 进行了封装，使用起来更加方便。
- Executors 提供了四种创建 ExecutorService 的方法，他们的使用场景如下：

```
- 1. Executors.newFixedThreadPool()
  - 创建一个定长的线程池，每提交一个任务就创建一个线程，直到达到池的最大长度，这时线程池会保持长度不再变化
- 2. Executors.newCachedThreadPool()
  - 创建一个可缓存的线程池，如果当前线程池的长度超过了处理的需要时，它可以灵活的回收空闲的线程，当需要增加时，
  - 它可以灵活的添加新的线程，而不会对池的长度作任何限制
- 3. Executors.newScheduledThreadPool()
  - 创建一个定长的线程池，而且支持定时的以及周期性的任务执行，类似于Timer
- 4. Executors.newSingleThreadExecutor()
  - 创建一个单线程化的executor，它只创建唯一的worker线程来执行任务
```

- 适用范围
  - 批处理任务

4.1 HandlerThread 构造函数和继承关系：

```
public class HandlerThread extends Thread {
    int mPriority;
    int mTid = -1;
    Looper mLooper;
    private @Nullable Handler mHandler;

    public HandlerThread(String name) { // 构造函数 1
        super(name);
        mPriority = Process.THREAD_PRIORITY_DEFAULT;
    }

    /**
     * Constructs a HandlerThread.
     * @param name
     * @param priority The priority to run the thread at. The value supplied must be from
     * {@link android.os.Process} and not from java.lang.Thread.
     */
    public HandlerThread(String name, int priority) { // 构造函数 2
        super(name);
        mPriority = priority;
    }
}
```

- 逻辑解析：
  - HandlerThread 继承自 Thread 类，所以它本质上是一个线程类，可以实现线程相关的操作。

- 我们来看 HandlerThread 的构造函数，这里有 2 个重载版本。
- 第一个构造函数接收一个 name 参数，直接调用了 super 方法，为该线程命令，且线程优先级为 Process.THREAD_PRIORITY_DEFAULT。
- 第二个构造函数接收 2 个参数，可实现线程命名，和设置线程的优先级。

4.2 HandlerThread 的 run 方法：

```
@Override
public void run() {
    mTid = Process.myTid();
    Looper.prepare();
    synchronized (this) {
        mLooper = Looper.myLooper();
        notifyAll();
    }
    Process.setThreadPriority(mPriority);
    onLooperPrepared();
    Looper.loop();
    mTid = -1;
}
```

- 逻辑解析：
  - 我们在创建了 HandlerThread 实例之后，调用 start() 方法执行，ART 虚拟机会帮我们创建一个线程对象，然后在子线程中执行 run 方法。
  - run 方法中，执行了 Looper.prepare() 方法，创建了一个 Looper 对象并绑定到该线程。
  - 然后在同步块中，执行了 mLooper 的赋值，调用 notifyAll 通知 Looper 已创建完成。
  - 调用 Process.setThreadPriority 方法设置线程优先级。
  - 调用空方法 onLooperPrepared，通常用于回调使用。
  - 最后调用 Looper.loop() 方法，启动该线程的消息循环。

4.3 getLooper 方法：

```
public Looper getLooper() {
    if (!isAlive()) {//判断线程是否存活
        return null;
    }

    // If the thread has been started, wait until the looper has been created.
    synchronized (this) {
        //如果线程存活，但 mLooper 没有创建完成，则等待
        while (isAlive() && mLooper == null) {
            try {
                wait();
            } catch (InterruptedException e) {
            }
        }
    }
    return mLooper;//返回当前线程的 Looper 对象
}
```

- 逻辑解析：
- getLooper 方法可以返回当前线程绑定的 Looper 对象，我们可以使用该对象作为参数，创建一个该线程的 Handler 对象，用于线程交互。
- getLooper 方法首先会判断当前线程是否存活，如果存活，则继续。
- 如果线程存活，但 Looper 对象还没有创建完成，则调用 wait 方法进行等待（Looper 创建完成后，会在 HandlerThread 的 run 方法中，调用 notifyAll() 通知，以继续执行当前逻辑）。
- 最后返回 Looper 对象即可。

4.4 HandlerThread 的 getThreadHandler 方法

```
/**
 * @return a shared {@link Handler} associated with this thread
 * @hide
 */
public Handler getThreadHandler() {
    if (mHandler == null) {
        mHandler = new Handler(getLooper());
    }
    return mHandler;
}
```

- 如果我们不想自己创建 Handler 对象，也可以使用 HandlerThread 为他们提供的 Handler 对象。
- getThreadHandler 方法返回一个当前线程的 Handler 对象。
- 它是一个隐藏方法，我们在应用中不可调用。

4.5 HandlerThread 的退出

```
//立即执行退出
public boolean quit() {
    Looper looper = getLooper();
    if (looper != null) {
        looper.quit();
        return true;
    }
    return false;
}
//处理完成已到执行时间的消息后退出。
public boolean quitSafely() {
    Looper looper = getLooper();
    if (looper != null) {
        looper.quitSafely();
        return true;
    }
    return false;
}
```

- 当不需要 HandlerThread 时，需要调用 quit 方法或 quitSafely 方法结束线程管理的 Looper 消息循环。

4.6 HandlerThread 实战

- HandlerThread 的实现其实并不复杂，我们以一个简单 Demo 来看它的使用。

4.6.1 Demo

```
public class HandlerThreadTest {
    private final static String TAG = "budaye";
    public static final int HT_MSG = 1;
    private HandlerThread mHandlerThread = new HandlerThread("myHandlerThread", Process.THREAD_PRIORITY_BACKGROUND);
    private Handler mHandler = null;

    public void startHandlerthread(){
        mHandlerThread.start();
        if (mHandler == null){
            mHandler = new Handler(mHandlerThread.getLooper()){
                @Override
                public void handleMessage(@NonNull Message msg) {
                    super.handleMessage(msg);
                    switch (msg.what){
                        case HT_MSG:
                            Log.d(TAG, " 当前线程: " + Thread.currentThread());
                            Log.d(TAG, " 收到其他线程发送过来的消息了");
                            break;
                    }
                }
            };
```

```
        }
    }
    public void sendMessage() {
        Log.d(TAG, " 当前线程: " + Thread.currentThread());
        Message msg = Message.obtain();
        msg.what = HT_MSG;
        mHandler.sendMessage(msg);
    }
}
```

- 逻辑解析:
  - 在 HandlerThreadTest 里，创建了一个 HandlerThread 对象。
  - 调用 startHandlerthread 方法，开始了 HandlerThread 对象的线程消息循环，并且创建了一个 mHandler 对象，用于处理其他线程发送过来的消息。
  - 我们可以在 UI 线程中调用 sendMessage 方法来发送给 HandlerThread 线程一个消息，并在 mHandler 对象的 handleMessage 方法中进行处理。
- Demo 日志输出

```
25208-25208/com.example.simpledemo D/budaye: 当前线程: Thread[main,5,main]
25208-25284/com.example.simpledemo D/budaye: 当前线程: Thread[myHandlerThread,5,main]
25208-25284/com.example.simpledemo D/budaye: 收到其他线程发送过来的消息了
```

- 我们分别在 2 个线程中输出了日志。

4.7 最佳实践 & 总结

- HandlerThread 是一个异步处理的工具类，它可以帮助我们很轻松的实现异步线程处理。
- HandlerThread 继承自 Thread 类，它的本质是一个线程类。
- HandlerThread 实现原理非常简单，它利用了 Handler 原理，在内部了一个 Looper 循环，并绑定到当前线程中。
- 我们使用创建一个 Handler 对象，绑定到 HandlerTHread 对象所对应的 Looper，并处理其他线程发送过来的消息。
- HandlerThread 在构造方法中可以设置线程优先级,默认使用 Process.THREAD_PRIORITY_DEFAUL 作为默认优先级。
- 我们在应用过程中，不要大量使用 HandlerThread 来执行异步任务，这样会造成线程资源的浪费，大量的异步任务，建议使用线程池来进行操作。
- HandlerThread 的线程优先级设定一定要注意，如果任务优先级不高，则应该设置成后台任务优先级，避免和 UI 线程抢占系统资源。

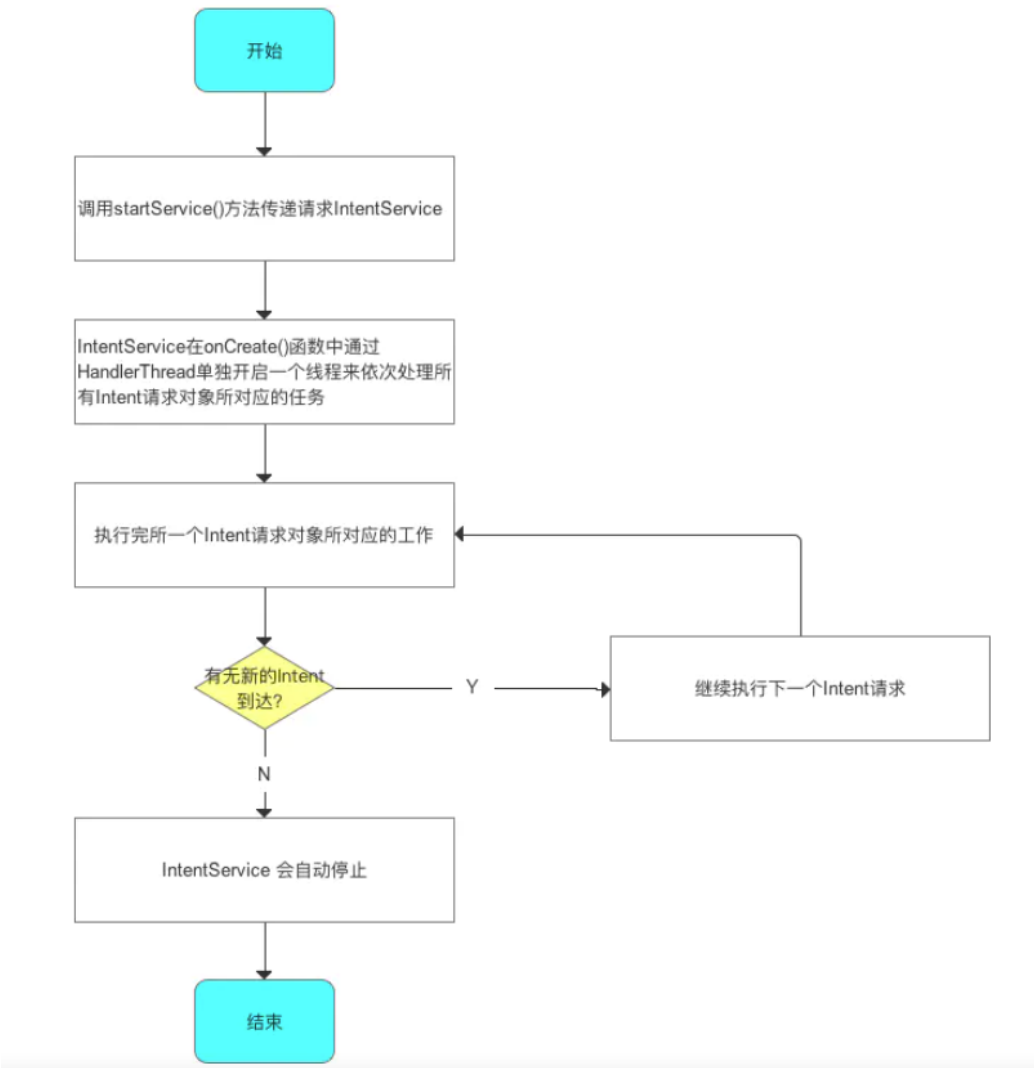
5 IntentService

- 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。
- 作用
  - 处理异步请求 & 实现多线程
- 使用场景: 线程任务需按顺序、在后台执行
  - 最常见的场景：离线下载
  - 不符合多个数据同时请求的场景：所有的任务都在同一个 Thread looper 里执行

- 使用步骤
  - 步骤 1：定义 IntentService 的子类，需复写 onHandleIntent() 方法
  - 步骤 2：在 Manifest.xml 中注册服务
  - 步骤 3：在 Activity 中开启 Service 服务
- 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 源码原理分析



- 特别注意
- 若启动 IntentService 多次，那么每个耗时操作则以队列的方式在 IntentService 的 onHandleIntent 回调方法中依次执行，执行完自动结束
- 接下来，我们将通过源码分析解决以下问题：
  - IntentService 如何单独开启 1 个新的工作线程
  - IntentService 如何通过 onStartCommand() 将 Intent 传递给服务 & 依次插入到工作队列中

5.1.1 源码分析

1. 问题 1：IntentService 如何单独开启 1 个新的工作线程
   - 主要分析内容 = IntentService 源码中的 onCreate() 方法

```
@Override public void onCreate() {
    super.onCreate();

    // 1. 通过实例化 HandlerThread 新建线程 & 启动；故使用 IntentService 时，不需额外新建线程
    // HandlerThread 继承自 Thread，内部封装了 Looper
    HandlerThread thread = new HandlerThread("IntentService[" + mName + "]");
    thread.start();

    // 2. 获得工作线程的 Looper & 维护自己的工作队列
    mServiceLooper = thread.getLooper();

    // 3. 新建 mServiceHandler & 绑定上述获得 Looper
    // 新建的 Handler 属于工作线程  ======>>>>> 分析 1
    mServiceHandler = new ServiceHandler(mServiceLooper);
}
/**
 * 分析 1：ServiceHandler 源码分析
 **/
private final class ServiceHandler extends Handler {
    // 构造函数
    public ServiceHandler(Looper looper) {
        super(looper);
    }
    // IntentService 的 handleMessage() 把接收的消息交给 onHandleIntent() 处理
    @Override
        public void handleMessage(Message msg) {
        // onHandleIntent 方法在 工作线程 中执行
        // onHandleIntent() = 抽象方法，使用时需重写  ======>>>>> 分析 2
        onHandleIntent((Intent)msg.obj);
        // 执行完调用 stopSelf() 结束服务
        stopSelf(msg.arg1);
    }
}
/**
 * 分析 2：  onHandleIntent() 源码分析
 * onHandleIntent() = 抽象方法，使用时需重写
 **/
@WorkerThread protected abstract void onHandleIntent(Intent intent);
```

2. 问题 2：IntentService 如何通过 onStartCommand() 将 Intent 传递给服务 & 依次插入到工作队列中

```
/**
 * onStartCommand () 源码分析
 * onHandleIntent() = 抽象方法，使用时需重写
 **/
public int onStartCommand(Intent intent, int flags, int startId) {
    // 调用 onStart() ======>>>>> 分析 1
    onStart(intent, startId);
    return mRedelivery ? START_REDELIVER_INTENT : START_NOT_STICKY;
}
/**
 * 分析 1：onStart(intent, startId)
 **/
public void onStart(Intent intent, int startId) {
    // 1. 获得 ServiceHandler 消息的引用
    Message msg = mServiceHandler.obtainMessage();
    msg.arg1 = startId;

    // 2. 把 Intent 参数包装到 message 的 obj 发送消息中，
    //这里的 Intent = 启动服务时 startService(Intent) 里传入的 Intent
    msg.obj = intent;

    // 3. 发送消息，即 添加到消息队列里
    mServiceHandler.sendMessage(msg);
}
```

- 至此，关于 IntentService 的源码分析讲解完毕。

5.1.2 源码总结

- 从上面源码可看出：IntentService 本质 = Handler + HandlerThread：
  - 通过 HandlerThread 单独开启 1 个工作线程：IntentService

  - 创建 1 个内部 Handler： ServiceHandler
  - 绑定 ServiceHandler 与 IntentService
  - 通过 onStartCommand() 传递服务 intent 到 ServiceHandler 、依次插入 Intent 到工作队列中 & 逐个发送给 onHandleIntent()
  - 通过 onHandleIntent() 依次处理所有 Intent 对象所对应的任务
- 因此我们通过复写 onHandleIntent() & 在里面根据 Intent 的不同进行不同线程操作即可

5.1.3 注意事项

- 此处，有两个注意事项需要关注的：
  - 工作任务队列 = 顺序执行
  - 不建议通过 bindService() 启动 IntentService
- 注意事项 1：工作任务队列 = 顺序执行
  - 即若一个任务正在 IntentService 中执行，此时你再发送 1 个新的任务请求，这个新的任务会一直等待直到前面一个任务执行完毕后才开始执行
- 原因：
  - 由于 onCreate() 只会调用一次 = 只会创建 1 个工作线程；
  - 当多次调用 startService(Intent) 时（即 onStartCommand （） 也会调用多次），其实不会创建新的工作线程，只是把消息加入消息队列中 & 等待执行。
  - 所以，多次启动 IntentService 会按顺序执行事件
  - 若服务停止，则会清除消息队列中的消息，后续的事件不执行
- 注意事项 2：不建议通过 bindService() 启动 IntentService
  - 原因：

```
// 在 IntentService 中，onBind()`默认返回 null
@Override
public IBinder onBind(Intent intent) {
    return null;
}
```

- 采用 bindService() 启动 IntentService 的生命周期如下：

```
onCreate() ->> onBind() ->> onunbind()->> onDestory()
```

- 即，并不会调用 onStart() 或 onStartcommand()，故不会将消息发送到消息队列，那么 onHandleIntent() 将不会回调，即无法实现多线程的操作
  - 此时，你应该使用 Service，而不是 IntentService

5.2 优缺点

5.2.1 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。

5.2.2 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。

5.3 与 Service 的区别

类型	运行线程	结束服务 操作	备注
IntentService（继承自Service类）	创建1个工作线程处理多线程任务	不需要（在所有intent被处理完后，系统会自动关闭服务）	• IntentService 为 Service的 onBingd() 提供了默认实现：返回null • IntentService 为 Service的 onStartCommand （）提供了默认实现：将请求的intent添加到队列中
Service	主线程（不能处理耗时操作，否则会出现ANR）	需主动调用stopService （）	

5.4 与其他线程的区别

类型	线程属性	作用	线程优先级
IntentService	类似 后台线程（采用HandlerThread实现）	后台服务（继承了 Service）	高（不容易被系统杀死）
其他线程	普通线程	普通多线程作用	低（若进程中无活动的四大组件，则该线程的优先级非常低，容易被系统杀死）

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。

7 Java 基础之多线程 (没那么复杂)

7.1 多线程的引入

- 1. 什么是多线程
  - 线程是程序执行的一条路径，一个进程中可以包含多条线程; 多线程并发执行可以提高程序的效率
- 2. 进程和线程之间的关系
  - 操作系统可以同时执行多个任务，每个任务就是进程; 进程可以同时执行多个任务，每个任务就是线程。
- 3. 多线程的应用场景
  - 红蜘蛛同时共享屏幕给多个电脑
  - 迅雷开启多条线程一起下载
  - QQ 开启多人聊天
  - 服务器同时处理多个客户的请求
- 多线程并行和并发的区别
  - 并行性和并发性是两个概念，并行性指在同一时刻，有多条指令在多个处理器上同时执行
  - 并发性指的是同一时刻只有一条指令被执行，但多个进程指令被快速切换执行是的在宏观上具有多个进程被同时执行的效果
- Java 程序运行原理和 JVM 的启动是多线程的吗?
  - java 程序运行原理
  - Java 命令启动 jvm，启动 jvm 等于启动一个应用程序，也就是启动了一个进程，该进程会自动启动一个“主线程”，然后主线程去调用某个类的 main 方法
  - Jvm 启动是多线程的的吗
  - JVM 启动至少启动了垃圾回收线程和主线程，所以是多线程

7.2 多线程的实现方式

7.2.1 1. 继承 Thread 类创建线程类

- (1) 定义 Thread 的子类，并重写该类的 run 方法，该 ru 的执行体就代表了线程需要完成的任务，因此 run() 方法被称为线程执行体
- (2) 创建 Thread 子类的实例，即创建了线程对象
- (3) 调用线程对象的 start() 方法启动该线程

```
public class Demo1__Thread {
    public static void main(String[] args) {
        ChThread t=new ChThread();
        t.start();
        for(int i=0;i<1000;i++) {
            System.out.println(" 我是主方法! ");
        }
    }
}
 class ChThread extends Thread {
    @Override
    public void run() {
        super.run();
        for(int i=0;i<1000;i++) {
            System.out.println(" 我是 run 方法");
        }
    }
 }
```

- 上述代码的执行验证了多线程，如果上述程序的执行过程是多线程的话，会发现屏幕中的”我是主方法”和“我是 run 方法”的字样是交替出现的，这说明了程序的的执行过程为并行执行 Thread 类的 Start() 方法启动 run() 方法的线程，和主方法中的执行同时进行。

7.2.2 2. 实现 Runnable 接口创建线程类

- (1) 定义一个实现了 Runnable 接口的实现类 (2) 创建 Runnable 实现类的实例 (3) 将创建的实例作为 Thread 类的 target 类创建 Thread 对象，该对象才是真正的线程对象 (4) 用创建的 Thread 对象启动线程

```
public class tmp implements Runnable { // Thread_Running
    private int i;
    public void run() {
        for(;i < 15;i++)
            System.out.println(Thread.currentThread().getName()+"  "+i);
    }
    public static void main(String[] args) {
        for(int i = 0;i < 15;i++) {
            System.out.println(Thread.currentThread().getName()+"  "+i);
            if(i == 7) {
                new Thread(new tmp(),"thread 1").start();
                new Thread(new tmp(),"thread 2").start();
            }
        }
    }
}
```

- 执行的结果如下

```
main  0
main  1
main  2
main  3
main  4
main  5
main  6
thread 1  0
thread 1  1
thread 1  2
thread 1  3
thread 1  4
```

```
thread 1  5
thread 2  0
thread 2  1
thread 2  2
thread 2  3
thread 2  4
thread 2  5
thread 2  6
thread 2  7
thread 2  8
thread 2  9
thread 2  10
thread 2  11
thread 2  12
thread 2  13
thread 2  14
main  7
thread 1  6
main  8
thread 1  7
main  9
thread 1  8
thread 1  9
thread 1  10
main  10
thread 1  11
thread 1  12
thread 1  13
thread 1  14
main  11
main  12
main  13
main  14
```

- 查看 API 文档，会发现 Runnable 接口只定义了 run() 方法这一个抽象类，所以实现 Runnable 接口的实现类只有 run() 方法，仅作为线程执行体，所以，Runnable 对象仅仅作为 Thread 对象的 target，而实际的线程对象依然是 Thread 实例，Thread 实例负责执行 target 的 run() 方法。

7.2.3 3. 实现 Callable 接口和 Future 接口创建多线程 (Callable 接口和 FutureTask 还有点儿生)

- （1）Callable 接口更像是 Runnable 接口的增强版，相比较 Runable 接口,Call() 方法新增捕获和抛出异常的功能;Call() 方法可以返回值
- （2）Future 接口提供了一个实现类 FutureTask 实现类，FutureTaks 类用来保存 Call() 方法的返回值，并作为 Thread 类的 target。
- （3）调用 FutureTask 的 get() 方法来获取返回值

```java
import java.util.concurrent.Callable;
import java.util.concurrent.ExecutionException;
import java.util.concurrent.FutureTask;
/**
 * (1) 创建 Callable 接口的实现类，并重写 call() 方法，该 call() 方法作为线程的执行体，且有返回值
 * (2) 创建了 Callable 接口的实现类的实例，并用 FutureTask() 方法包装对象，该 FutureTask() 对象实现了
 *     将对象的返回值包装的功能
 * (3) 使用 FutureTask 对象将 Thread 对象的 target，创建并启动线程
 * (4) 调用 FutureTask 对象的 get() 方法获得子线程执行结束后的返回值
 * */
public class tmp implements Callable < Integer> {    // tmp: Callable_Future
    @Override
        public Integer call() throws Exception {   // 重写 Callable 接口中的 call() 方法
        int i = 0;
        for(; i < 20; i++)
            System.out.println(Thread.currentThread().getName()+" "+i);
        return i;
    }
    public static void main(String[] args) throws InterruptedException, ExecutionException {
        // 创建 Callable 的对象
        tmp ca = new tmp();
        FutureTask < Integer> ft = new FutureTask < Integer>(ca);
```

```
        for(int i = 0; i < 20; i++) {
            // 返回值主线程的名称和执行代号
            System.out.println(Thread.currentThread().getName()+" "+i);
            if (i == 7) {
                new Thread(ft, "Callable thread").start();
                // 该方法将导致主线程被阻塞，直到 call() 方法结束并返回为止
                // System.out.println(" 子线程的返回值"+ft.get());
            }
        }
        try {
            System.out.println("sub thread return value: "+ft.get());
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

- 上面的程序中，FutufeTask 方法的 get() 方法将获得 Call() 方法的返回值，但是该方法将导致主线程受阻直到 Call() 方法结束并返回为止。
- 打印结果如下：

```
main 0
main 1
main 2
main 3
main 4
main 5
main 6
main 7
main 8
main 9
main 10
main 11
main 12
main 13
Callable thread 0
main 14
Callable thread 1
main 15
Callable thread 2
main 16
Callable thread 3
main 17
main 18
main 19
Callable thread 4
Callable thread 5
Callable thread 6
Callable thread 7
Callable thread 8
Callable thread 9
Callable thread 10
Callable thread 11
Callable thread 12
Callable thread 13
Callable thread 14
Callable thread 15
Callable thread 16
Callable thread 17
Callable thread 18
Callable thread 19
sub thread return value: 20
```

7.2.4 4. 三种实现多线程的方式对比

实现方式	重写方法	启动线程的方式	优点
继承 Thread 类	run()	Thread 对象.start()	编程简单，直接用 this 即可获取当前对象
实现 Runnable 接口	run()	Thread 对象.start()	1. 实现类可以继承其他类 2. 多个对象共享一个 target，形成了清晰的模型
实现 Callable 接口	Call()	Thread 对象.start()	1. 同 Runnable 接口 2.Call() 方法有返回值并能抛出异常

- 通过上面的对比发现，一般在项目中，我们使用 Runnable 接或者 Callable 接口来实现多线程。

7.2.5 5. 线程的生命周期

- 1. 线程并不是创建过之后就开始执行的，也不是一直处于执行状态，一个线程的生命周期一个又五个：新建、就绪、运行、阻塞、死亡。
- 2. 线程的这五种状态的原理参照《计算机组成原理 2》多线程部分

7.2.6 6. 控制线程

1. 1.join 线程
    - 1.Thread 类中的成员方法，让调用 join() 方法的线程处于等待状态，直到子线程执行结束
2.join() 方法通常由使用线程的程序调用，用于将一个线程拆分成若干小的线程执行，执行结束最后由主线程进行进一步的操作

```
public class tmp extends Thread { // tmp: tmp
    // 创建一个有参构造函数，用来为线程命名
    public tmp(String str) {
        super(str);
    }
    // 重写 run 方法
    public void run() {
        for(int i = 1; i < 20; i++) {
            System.out.println(getName()+" "+i);
        }
    }
    public static void main(String[] args)throws Exception {
        // 启动子线程
        new tmp("Sub thread").start();
        for(int i = 0; i < 20; i++) {
            if(i == 7) {
                tmp jt = new tmp("new thread's sub thread");
                jt.start();  // 启动子线程
                jt.join();   // 让主线程处于等待状态
            }
            System.out.println(Thread.currentThread().getName()+" "+i);
        }
    }
}
```

- mian 线程调用 join 之后，出去阻塞状态，两个子线程并发执行。直到两个子程序执行结束之后，main 线程才开始执行。
- join() 线程有如下三种重载形式
    - 1.join() :
    - 2.join(long millis) : 被 join 的线程的时间最长为 millis 毫秒，如果超过了这个 millis 则线程不被执行
    - 3.join(long millis,int nanos) : 被 join 的线程的时间最长为 millis 毫秒 +nanos 微秒，如果超过了这个时间则线程不被执行第三种重载行驶一般用不到，因为无论是计算机系统还是计算机硬件，还没发精确到微秒
- 打印结果如下：

```
main 0
main 1
main 2
main 3
main 4
main 5
main 6
Sub thread 1
Sub thread 2
Sub thread 3
new thread's sub thread 1
```

```
new thread's sub thread 2
Sub thread 4
new thread's sub thread 3
new thread's sub thread 4
Sub thread 5
new thread's sub thread 5
Sub thread 6
new thread's sub thread 6
Sub thread 7
new thread's sub thread 7
Sub thread 8
new thread's sub thread 8
Sub thread 9
new thread's sub thread 9
Sub thread 10
new thread's sub thread 10
Sub thread 11
new thread's sub thread 11
Sub thread 12
new thread's sub thread 12
Sub thread 13
new thread's sub thread 13
Sub thread 14
new thread's sub thread 14
Sub thread 15
new thread's sub thread 15
Sub thread 16
new thread's sub thread 16
Sub thread 17
new thread's sub thread 17
Sub thread 18
new thread's sub thread 18
Sub thread 19
new thread's sub thread 19
main 7
main 8
main 9
main 10
main 11
main 12
main 13
main 14
main 15
main 16
main 17
main 18
main 19
```

2. 2. 后台线程

- 后台线程运行于后台，任务是为其他线程提供服务，也被称为“守护线程”或“精灵线程”。JVM 的垃圾回收机制就是一个典型的后台线程。
 通过调用 Thread 类的 setDaemon(true) 方法将线程设置为后台线程

```java
public class tmp extends Thread { // DaemonThread
    @Override
        public void run() {
        for(int i = 0; i < 2000; i++) {
            System.out.println(getName()+" "+i);
        }
    }
    public static void main(String[] args) {
        tmp dt = new tmp();
        dt.setDaemon(true);
        dt.start();
        for(int i = 0; i < 7; i++) {
            System.out.println(Thread.currentThread().getName()+" "+i);  // 主线程：  10 个
        }
    }
}
```

- 运行结果：

```
Thread-0 0
Thread-0 1
```

```
Thread-0 2
main 0
Thread-0 3
main 1
main 2
main 3
main 4
main 5
main 6
Thread-0 4
Thread-0 5
Thread-0 6
```

- 1.Thread-0 的后台进程本该当 i=999 的时候才停止进行，但是程序中只进行到 114 次，这是因为所有的前台进程结束之后，后天进程的存在也就是去了意义，所以后台进程也跟着死亡。
- 2. 前台线程死亡之后，JVM 会通知后台线程死亡，但是从后台线程接收指令到做出反应需要一定的时间，这就是为什么上述程序中的后台进程在 main 线程死亡之后后台进程还进行到 114 的原因。

3. 3. 线程睡眠

- 如果需要线程停顿一段时间进入阻塞状态，可以调用 Thread 类的静态方法 sleep()，sleep （) 有两种重载形式
  - sleep(long millis)
    - 让当前正在执行的线程暂停 mili 毫秒，进入阻塞状态。该方法受到系统计时器和线程调度器精度的影响。
  - sleep(long millis,int nanos)
    - 让当前正在执行的线程暂停 milis 毫秒 +nanos 微秒，进入阻塞状态。该方法受到系统计时器和线程调度器精度的影响（不常用）

#+BEGIN_SRC java public class tmp { // SleepThread public static void main(String[] args) throws Exception { for(int i = 0; i < 7; i++) { System.out.println(Thread.currentThread().ge "+i); Thread.sleep(2000); } } } #+END_SRC (

- 观察程序想运行过程会发现，每一个进程 (_ 进程 _？) 之间相隔 1 秒。

4. 4. 线程让步

- 调用 Thrad 类的静态方法 yield() 方法可以实现线程让步，和 sleep() 方法类似，yield() 方法也是让当前正在运行的线程暂停，但是不会使线程阻塞，而是让线程进入就绪状态，让线程调度器重新安排一次线程调度，完全有可能出现的状况是，刚刚调用 yield() 方法进入就绪状态的线程就被线程调度器重新调度出来重新执行。

```java
public class tmp extends Thread { // YieldThread
    public tmp(String str) {
        super(str);
    }
    @Override
        public void run() {
        // TODO Auto-generated method stub
        for (int i = 0; i < 12; i++) {
            System.out.println(getName()+" "+i);
            if (i == 7)
                Thread.yield();                    // 线程让步
        }
    }
    public static void main(String[] args) {
        tmp yt1 = new tmp("Adv");
        yt1.setPriority(MAX_PRIORITY);             // 将次线程设置成最高优先级
        yt1.start();                               // 启动线程
        tmp yt2 = new tmp("Base");
        yt2.setPriority(MIN_PRIORITY);
        yt2.start();
    }
}
```

- 1. 线程调用 yield() 方法后将执行的机会让给优先级相同的线程
2. 高优先级的线程调用 yield() 方法暂停之后，系统中没有与之相同优先级和更高的优先级的线程，则线程调度器会将该线程重新调度出来，重新执行。
- sleep() 方法和 yield() 方法比较

方法	执行机会	线程状态	是否抛出
sleep()	线程暂停之后，会给其他线程执行的机会，不用理会线程的优先级	阻塞状态	Interrup
yield()	线程暂停之后，只会给优先级相同或更高优先级的线程执行的机会	就绪状态	否

- 打印结果：

```
Base 0
Base 1
Base 2
Base 3
Base 4
Base 5
Base 6
Base 7
Adv 0
Adv 1
Adv 2
Adv 3
Base 8
Adv 4
Base 9
Adv 5
Base 10
Adv 6
Base 11
Adv 7
Adv 8
Adv 9
Adv 10
Adv 11
```

5. 5. 改变线程优先级
    - 1. 一般线程具有优先级，更高优先级的线程比优先级低的线程能获得更多的执行机会
    - 2. 每个线程默认的优先级和创建他们的父线程的优先级相同。
    - 3.Thread 类提供了 setPriorit(int newPriority)、getPriority() 方法来设置和获取线程的优先级 4. 也可以使用 Thread 类的 3 个静态常量来设置线程的优先级

8 两种线程池：THREAD_POOL_EXECUTOR 和 SERIAL_EXECUTOR

- 线程池是管理异步任务 AsyncTask 所创建的队列，系统提供了两个线程池：THREAD_POOL_EXECUTO 和 SERIAL_EXECUTOR
    - 当一个 AsyncTask 执行 execute() 方法时，默认被加入 SERIAL_EXECUTOR 中，等待执行。SERIAL_EXECUTOR 中的任务按照“先进先执行，依次执行”的原则，后续任务要等先前的任务执行完毕后才会执行，顺次执行。
    - 而另一种线程池 THREAD_POOL_EXECUTOR 则不同，被加入线程池的任务会立刻执行，也就是所有任务并发执行。但是有个前提，如果 THREAD_POOL_EXECUTOR 中一起执行的任务超过了 5 个，那么新加入的任务会等这五个中有线程执行完毕了再执行。(这里再查验证一下，与他给出的代码不符合)
- 使用 AsyncTask.executeOnExecutor() 来指定使用哪个线程池来管理 AsyncTask

```
DownloadTask task = new DownloadTask(m_progressBar[m_prgBarIndex++], MainActivity.this);
m_taskList.add(task);
task.executeOnExecutor(AsyncTask.THREAD_POOL_EXECUTOR, "westworld");
```

9 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。

9.1 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。

9.2 补充总结

- AsyncTask 是 android 提供的轻量级的异步类，可以直接继承 AsyncTask，在类中实现异步操作，并提供接口反馈当前异步执行的程度 (可以通过接口实现 UI 进度更新)，最后反馈执行的结果给 UI 主线程。
- AsyncTask 通过一个阻塞队列 BlockingQuery<Runnable> 存储待执行的任务，利用静态线程池 THREAD_POOL_EXECUTOR 提供一定数量的线程，默认 128 个。在 Android 3.0 以前，默认采取的是并行任务执行器，3.0 以后改成了默认（by default，如果想要并行执行，需要设置配置）采用串行任务执行器，通过静态串行任务执行器 SERIAL_EXECUTOR 控制任务串行执行，循环取出任务交给 THREAD_POOL_EXECUTOR 中的线程执行，执行完一个，再执行下一个。(这里写的内容需要再校对一下)
- 用法举例：

```
class DownloadTask extends AsyncTask<Integer, Integer, String>{
    // AsyncTask<Params, Progress, Result>
    //后面尖括号内分别是参数 (例子里是线程休息时间), 进度 (publishProgress 用到), 返回值类型
    @Override
    protected void onPreExecute() {
        //第一个执行方法
        super.onPreExecute();
    }
    @Override
    protected String doInBackground(Integer... params) {
        //第二个执行方法,onPreExecute() 执行完后执行
        for(int i=0;i<=100;i++){
            publishProgress(i);
            try {
                Thread.sleep(params[0]);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
        return " 执行完毕";
    }
    @Override
    protected void onProgressUpdate(Integer... progress) {
        //这个函数在 doInBackground 调用 publishProgress 时触发, 虽然调用时只有一个参数
        //但是这里取到的是一个数组, 所以要用 progesss[0] 来取值
        //第 n 个参数就用 progress[n] 来取值
        tv.setText(progress[0]+"%");
        super.onProgressUpdate(progress);
    }
    @Override
    protected void onPostExecute(String result) {
        //doInBackground 返回时触发, 换句话说, 就是 doInBackground 执行完后触发
        //这里的 result 就是上面 doInBackground 执行后的返回值, 所以这里是" 执行完毕"
        setTitle(result);
        super.onPostExecute(result);
    }
}
```

9.3 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：THREAD_PRIORITY_BACKGROUND，不会与 UI 线程抢占资源。
  - 1. 处理单个异步任务简单，可以获取到异步任务的进度
  - 2. 可以通过 cancel 方法取消还没执行完的 AsyncTask
  - 3. 处理多个异步任务代码显得较多

9.4 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。
- 适用范围
  - 1. 单个异步任务的处理

10 AsyncTask 源码解析

- 我们接下来分析 AsyncTask 的源码，看下它的实现原理。
- 在 Demo 中，我们调用 AsyncTask 的实例对象的 execute 方法来实现调用的，那么我们就从 execute 方法作为入口点来分析。

10.1 AsyncTask 的 execute 方法：

```
@MainThread
public final AsyncTask<Params, Progress, Result> execute(Params... params) {
    return executeOnExecutor(sDefaultExecutor, params);
}
```

- 这里调用了 executeOnExecutor 方法，参数 sDefaultExecutor 是一个线程池，作为默认线程池。

10.2 executeOnExecutor 方法：

```
@MainThread
public final AsyncTask<Params, Progress, Result> executeOnExecutor(Executor exec,
        Params... params) {
    if (mStatus != Status.PENDING) {
        switch (mStatus) {
            case RUNNING:
                throw new IllegalStateException("Cannot execute task:"
                        + " the task is already running.");
            case FINISHED:
                throw new IllegalStateException("Cannot execute task:"
                        + " the task has already been executed "
                        + "(a task can be executed only once)");
        }
    }
    mStatus = Status.RUNNING;
    onPreExecute();
    mWorker.mParams = params;
    exec.execute(mFuture);
    return this;
}
```

- 逻辑解析
  - 该方法在 UI 线程中执行。
  - 属性 mStatus 用来判断当前任务的状态，当任务正在运行或已经结束了，会抛出 Error。
  - 设置 mStatus 的状态为 Status.RUNNING。
  - 调用 onPreExecute() 方法，该方法通常在使用 AsyncTask 时，需要实现。
  - 把参数 params 保存在 mWorker.mParams 中。
  - 调用线程池，执行 mFuture。
- 线程池执行的部分我们后面来分析，先来看这里有 2 个重要的属性 mWorker 和 mFuture，它们起到了非常重要的作用，我们来看。

10.3 mWorker、mFuture

- 属性 mWorker 和 mFuture 是在 AsyncTask 的构造方法中初始化的，它们是后台线程控制逻辑的核心所在。
- 我们来看 AsyncTask 的构造方法：

```
public AsyncTask() {//这里是在 UI 线程中执行的
    this((Looper) null);
}
public AsyncTask(@Nullable Handler handler) {
    this(handler != null ? handler.getLooper() : null);
}
public AsyncTask(@Nullable Looper callbackLooper) {
    mHandler = callbackLooper == null || callbackLooper == Looper.getMainLooper()
        ? getMainHandler()
        : new Handler(callbackLooper);
    mWorker = new WorkerRunnable<Params, Result>() {
        public Result call() throws Exception {//这里是在后台线程中执行的
            mTaskInvoked.set(true);
            Result result = null;
            try {
                Process.setThreadPriority(Process.THREAD_PRIORITY_BACKGROUND);//这里设置了线程的优先级为 THREAD_PRIORITY_
                //noinspection unchecked
                result = doInBackground(mParams);//这里调用了 doInBackground 方法
                Binder.flushPendingCommands();
            } catch (Throwable tr) {
                mCancelled.set(true);
                throw tr;
            } finally {
                postResult(result); //post 结果给 UI 线程
            }
            return result;//把后台线程处理结果返回
        }
    };
    mFuture = new FutureTask<Result>(mWorker) {
        @Override
        protected void done() { //异步任务执行完成后回调，这里的当前线程还是在后台线程中
            try {
                postResultIfNotInvoked(get());
            } catch (InterruptedException e) {
                android.util.Log.w(LOG_TAG, e);
            } catch (ExecutionException e) {
                throw new RuntimeException("An error occurred while executing doInBackground()",
                        e.getCause());
            } catch (CancellationException e) {
                postResultIfNotInvoked(null);
            }
        }
    };
}
private static abstract class WorkerRunnable<Params, Result> implements Callable<Result> {
    Params[] mParams;
}
```

- 逻辑解析：
  - 首先初始化 Handler 对象，因为 AsyncTask 涉及到后台线程和 UI 线程之间的通信，它内部实现使用了 Handler 通信机制。
  - mWorker 属性赋值，这里的 mWorker 是一个 WorkerRunnable 对象，它是一个抽象类，继承自 Callable 接口，用于执行异步任务并获取线程执行结果。WorkerRunnable 对象的 call() 方法是在后台线程中执行的。这里可以看到在 call() 方法中，调用了 doInBackground 方法。
  - call() 方法中设置了线程的优先级为 THREAD_PRIORITY_BACKGROUND（后台线程优先级）。
  - mFuture 属性赋值，这里的 mFuture 是一个 FutureTask 对象，FutureTask 可以用于线程任务的控制等逻辑，异步任务执行完成后，会回调 done() 方法。
  - done() 方法中，调用 postResultIfNotInvoked 方法，把任务结果返回给 UI 线程，这里执行的是非正常情况下的返回，正常执行结束会在 mWorker 的 call 方法中通过调用 postResult(result) 将结果返回给 UI 线程。

10.4 Handler 处理

- 我们来看，后台任务处理完成后，会在 WorkerRunnable 的 call 方法中，调用 postResult(result) 将结果返回给 UI 线程，但如果后台任务执行之前被终止，则会把结果传递给 postResultIfNotInvoked 方法。
- 我们来看这两个方法：

```
private void postResultIfNotInvoked(Result result) {
    final boolean wasTaskInvoked = mTaskInvoked.get();
    if (!wasTaskInvoked) { //同步属性的布尔值，表示后台任务是否已经开始执行
        postResult(result);//这里，后台任务并没有执行
    }
}
private Result postResult(Result result) {
    @SuppressWarnings("unchecked")
    Message message = getHandler().obtainMessage(MESSAGE_POST_RESULT,
            new AsyncTaskResult<Result>(this, result));
    message.sendToTarget();
    return result;
}
```

- postResult 方法，创建一个 message 对象，把执行结果封装成一个 AsyncTaskResult 对象，通过 Handler 机制，将后台执行结果传递给 UI 线程（这里只考虑 UI 线程一种情况）。

10.5 串行执行的线程池实现

- 到了这里，Asynctask 的执行部分、结果处理以及跨线程通信部分，都已经分析完成了，接下来我们来看后台线程是如何执行的，任务队列是如何实现的。
- 我们从上文知道，AsyncTask 的 execute 方法中，直接使用了一个默认线程池 sDefaultExecutor 来负责后台线程的创建及执行。
- 回顾一下：

```
@MainThread
public final AsyncTask<Params, Progress, Result> execute(Params... params) {
    return executeOnExecutor(sDefaultExecutor, params);
}
```

- sDefaultExecutor 的单线程队列的实现：

```
private static volatile Executor sDefaultExecutor = SERIAL_EXECUTOR;
public static final Executor SERIAL_EXECUTOR = new SerialExecutor();

private static class SerialExecutor implements Executor {
    final ArrayDeque<Runnable> mTasks = new ArrayDeque<Runnable>();
    Runnable mActive;
    public synchronized void execute(final Runnable r) {
        mTasks.offer(new Runnable() {
            public void run() {
                try {
                    r.run();
                } finally {
                    scheduleNext();
                }
            }
        });
        if (mActive == null) {
            scheduleNext();
        }
    }
    protected synchronized void scheduleNext() {
        if ((mActive = mTasks.poll()) != null) {
            THREAD_POOL_EXECUTOR.execute(mActive);
        }
    }
}
```

- 我们可以看到，sDefaultExecutor 是 SerialExecutor 的一个实例对象。

- SerialExecutor 的属性：
  - mTasks：是一个先进先出的队列，这里可以理解为容量无限的一个先进先出的队列（当然不可能无限，不过正常情况下我们是达不到它的最大容量的）。
  - mActive：表示当前正在执行的任务，它是一个 Runnable 对象。
- 后台任务队列的执行过程：
  - 当第一次有任务进来时，先把任务添加到 mTasks 队列中，这时 mActive == null，执行 scheduleNext 方法。
  - scheduleNext 方法中，从队列中取出队列头部的 Runnale 对象，并赋给 mActive，最后把它交由 THREAD_POOL_EXECUTOR 线程池执行。
  - 当第一个任务执行结束后，紧接着会调用 scheduleNext 方法执行下一个任务，以此类推……
- 串行执行的秘密：这里其实就实现了一个单线程的任务队列，所有 AnsyncTask 的任务，都会顺序的，单线程执行。它的实现原理其实就是，准备了一个可以扩容的先进先出的任务队列 mTasks，所有的后台任务执行时，先进入队列中，然后调用 scheduleNext 执行，当前任务结束后，继续执行下一个任务，这样也就实现了串行的任务处理。

10.6 线程池 THREAD_POOL_EXECUTOR

- 这里的线程池其实没有发挥线程池的作用，因为默认情况下，AsyncTask 只会执行单个任务调用，因为在 SerialExecutor 中已经实现了任务的队列管理了，SerialExecutor 会单个顺序的执行线程任务的调用。
- THREAD_POOL_EXECUTOR 的定义：

```
public static final Executor THREAD_POOL_EXECUTOR;
private static final int CORE_POOL_SIZE = 1;
private static final int MAXIMUM_POOL_SIZE = 20;
private static final int BACKUP_POOL_SIZE = 5;
private static final int KEEP_ALIVE_SECONDS = 3;
static {
    ThreadPoolExecutor threadPoolExecutor = new ThreadPoolExecutor(
            CORE_POOL_SIZE, MAXIMUM_POOL_SIZE, KEEP_ALIVE_SECONDS, TimeUnit.SECONDS,
            new SynchronousQueue<Runnable>(), sThreadFactory);
    threadPoolExecutor.setRejectedExecutionHandler(sRunOnSerialPolicy);
    THREAD_POOL_EXECUTOR = threadPoolExecutor;
}
```

- THREAD_POOL_EXECUTOR 其实是一个可并发的线程池，它的核心线程数是 1，最大线程数是 20。（注意，不同 Android 版本中的实现会有不同）
- 线程的创建：

```
private static final ThreadFactory sThreadFactory = new ThreadFactory() {
    private final AtomicInteger mCount = new AtomicInteger(1);
    public Thread newThread(Runnable r) {
        return new Thread(r, "AsyncTask #" + mCount.getAndIncrement());
    }
};
```

- 线程池是通过 ThreadFactory 对象来进行线程创建的，并且会为线程命名为"AsyncTask #" + mCount.getAndIncrement()。

10.7 AsyncTask 并发的实现

- 在 Android 3.0 时，AsyncTask 的任务执行改回了单一线程中顺序执行，那么我们如果想用 AsyncTask 实现并发任务，可以做到吗？答案是肯定的，我们来看如何实现。

- 我们来看 AsyncTask 的 executeOnExecutor 方法：

```
public final AsyncTask<Params, Progress, Result> executeOnExecutor(Executor exec,
        Params... params) {
    ……
    exec.execute(mFuture);
    return this;
}
```

- 源码解析：
  - executeOnExecutor 方法，可以接受一个线程池的参数，来让使用者实现自定义线程池的目的。
  - 我们可以自定义一个并发的线程池，通过调用 executeOnExecutor 方法，替换掉 AsyncTask 的默认线程池，即可实现并发处理。
- 注意：虽然这里可以实现线程池的并发处理，但是并不建议这么做，如果想要实现并发的后台任务，推荐使用 java.util.concurrent 并发工具包中的 Executor、ThreadPoolExecutor 和 FutureTask 等。

10.8 AsyncTask 的最佳实践

- 到了这里，我们已经深入了解了 AsyncTask 的使用及实现原理了，那么我们在开发时，有哪些是需要注意的呢?
- AsyncTask 使用时的一些要求：
- AsyncTask 类必须在 UI 线程中被加载（Android 4.1 之后自动执行）。
- AsyncTask 实例对象，必须在 UI 线程中创建。
- 必须在 UI 线程中调用 execute() 来运行任务。
- 必须创建 AsyncTask 的子类，以它的子类形式来使用。
- 必须在子类中至少实现 doInBackground() 回调方法，通常我们还会实现 onPostExecute() 方法。
- 不要手动调用 onPreExecute(), onPostExecute(Result), doInBackground(Params…), onProgressUpdate(Progress…) 方法。
- AsyncTask 后台任务，只能执行一次，否则会抛出异常。
- 整个进程中的 AsyncTask 线程池及任务队列都是同一个（自定义的除外），大量的调用 AsyncTask 执行后台任务会造成任务队列整体执行时间的变长，导致任务执行的时间不可控。
- AsyncTask 的默认线程优先级是 Process.THREAD_PRIORITY_BACKGROUND（后台线程级别），优先级较低，分配的 CPU 资源会较少，不适合执行优先级较高的任务。

10.9 关于执行顺序及 Android 历史版本中的变更

- AsyncTask 后台任务的执行顺序是不可靠的，因为它在 Android 历史版本中经历了多次变更。在刚刚引入 AsyncTask 时，AsyncTask 后台任务的执行，是在单一线程中顺序执行的；但是在 Android 1.6 中，AsyncTask 的任务执行改成了可多线程并发执行；在 Android 3.0 时，AsyncTask 的任务执行又改回了单一线程中顺序执行，以避免多线程造成的并发等问题。
- 如果我们想通过 AsyncTask 来并发执行后台任务，可以使用 executeOnExecutor(java.util.concurrent java.lang.Object[]) 方法，传递一个线程池来执行。
- 注意：AsyncTask 在 Android R（Android 11）中被标记为弃用状态，建议使用 java.util.concurrent 并发工具包来代替。

11 几种异步方式的再次比较

实现方式	优点	缺点
Thread+ Handler	实现简单	1.代码规范性较差，不易维护 2.每次操作都会开启一个匿名线程，系统开销较大
HandlerThread	1.内部已经实现了普通线程的Looper消息循环 2.可以串行执行多个任务 3.内部拥有自己的消息队列，不会阻塞UI线程	1.没有结果返回接口，需要自行处理 2.消息过多时，容易造成阻塞 3.只有一个线程处理，效率较低 4.线程优先级默认优先级为THREAD_PRIORITY_DEFAULT，容易和 UI 线程抢占资源
IntentService	1.只需要继承 IntentService，就可以在onHandlerIntent 方法中异步处理Intent 类型任务 2.任务结束后 IntentService 会自行停止，无需手动调用 stopService 3.可以执行处理多个 Intent 请求，顺序执行多任务 4.IntentService 是继承自Service，具有后台Service的优先级	1.需要启动服务来执行异步任务，不适合简单任务处理 2.异步任务是由 HandlerThread 实现的，只能单线程、顺序处理任务 3.没有返回 UI 线程的接口
AsyncTask	1.结构清晰，使用简单，适合与后台任务的交互 2.异步线程的优先级已经被默认设置成THREAD_PRIORITY_BACKGROUND，不会与 UI 线程抢占资源	1.结构略复杂，代码较多 2.每个 AsyncTask 只能被执行一次，多次调用会发生异常 3.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

- 以上就是 Android 系统中常用的异步任务的实现方式。