第5章 多线程与网络应用

思想引领

视频讲解

5.1 使用多线程与 Handler

5.1.1 任务说明

本任务的演示效果如图 5-1 所示。在该应用中，活动页面视图根节点为垂直的 LinearLayout，在布局中依次放置 1 个 TextView，用于显示个人信息；1 个 TextView(id 为 tv_result)，用于显示计数器的值；两个水平放置的 Button，分别为 START 和 STOP。单击 START 按钮，启动一个后台线程每隔 0.01s 计数一次，并在 tv_result 上更新计数值，同时 START 按钮使能禁止，变为灰色，STOP 按钮使能开启；单击 STOP 按钮，后台线程结束，停止计数，同时 STOP 按钮使能禁止，START 按钮使能开启。单击 START 按钮，重新开启线程，计数器从 0 开始计数。

图 5-1 多线程计数器演示效果

5.1.2 任务相关知识点

1. 线程在 Android 中的重要性

Android 的 UI 优化机制，使系统在 UI 线程中(即 Activity、Fragment 等，能修改 UI 的活动线程)连续更新 UI 时，只取最后一次更新的结果而忽略过程中 UI 的更新。因此，若在

Button 的单击事件中使用 for 循环对 TextView 进行文本内容更新,则只能更新最后一次的结果。例如,for 循环执行 100 次,每次调用 Thread.sleep()方法睡眠 10ms 后对计数器自增,并将其更新到 TextView 上,在视觉效果上,用户只能看到 for 循环最后一次的更新结果,并且系统检测到 UI 长时间不能响应还可能直接报错。

对于 Android 编程而言,最佳的实践方式是将耗时的数据获取与处理(如网络连接、网络数据获取、较复杂的数据计算等)交给后台 Thread(线程)或者 Service(服务),而 Activity 或者 Fragment 等 UI 线程中则负责接收处理后的数据,渲染更新 UI,Android 的后台线程以及 Service 是不允许直接更新 UI 的。

Thread 的使用,主要是改写 run()方法,并对 Thread 对象调用 start()方法启动线程,使之在后台执行 run()方法。当 run()方法运行结束,线程消亡,即使线程定义为 Activity 的全局变量,当线程运行结束后,线程对象也是不可靠的,因此线程对象往往将其视为局部变量,用完即释。如果在主 UI 线程中调用后台线程对象的 run()方法,能执行对应的代码,但是并不是在后台线程中执行,而是在 UI 线程中直接执行后台线程的 run()方法,后台线程角色失效。

当后台线程启动后,对 Android 编程而言,亟须解决的一个问题是,后台线程产生的数据与 Android UI 前端之间如何交互。在本任务中,后台线程每隔 0.01s 会更新一次计数值,那么,Activity 前端 UI 线程中如何获知?对于编写 C 语言的开发者,最"勤劳"的做法是对线程设置某个标记变量,在发布数据时,将标记变量置 1,而前端在 for 循环中一直读取该标记变量,若为 1,则取计数值并将标记重置为 0;"高级"一点的做法是设置中断,在中断中处理数据。显然这些做法都不适合 Android,前者将导致 Android 的 CPU 一直处于忙碌状态,耗能增大,且系统响应迟钝;后者没有对应 API,难以实现。

对于 Android 系统,目前常用的前后台多线程之间的数据交互有 3 种方法。

(1) 通过 Handler(句柄)实现事件传递和数据交互。

(2) 通过自定义接口,在后台线程中切换主 UI 线程,并产生接口回调,在主 UI 线程中实现接口并响应回调。

(3) 通过 Android 的 LiveData,利用生产者和观察者模式,在后台线程中对 LiveData 数据通过 postValue()方法修改值,在主 UI 线程中则对 LiveData 调用 Observer 接口侦听数据变动。

2. 句柄 Handler

本任务使用 Handler 实现后台线程与前端的交互。Handler(android.os.Handler,不是 Java 包中的 java.util.logging.Handler)是 Android 所提供的一个特殊类,负责消息传递。

Handler 工作方式如图 5-2 所示。Handler 对象在主 UI 中(Activity 或 Fragment 等)定义,并且一般是全局变量。后台线程需要用到主 UI 中定义的 Handler 对象,因此 Handler 对象往往会作为构造参数传递给后台线程,当后台线程需要发送数据给前端 UI 时,则通过 Handler 对象的 obtainMessage()方法获得消息对象 Message,Message 可以使用 Bundle 放置多个字段的数据,进而 Handler 使用 Message 在线程间传输数据。Bundle 是 Android 提供的数据封装类,通过 key-value 的方式放置数据,可将其理解为超级 HashMap,提供了诸如 putFloat()、getFloat()、putString()、getString()、putStringArray()、

getStringArray()等常用数据的存取方法，也提供了 putSerializable()、getSerializable()等方法支持自定义类数据(需要实现 Serializable 序列化接口)的存取。

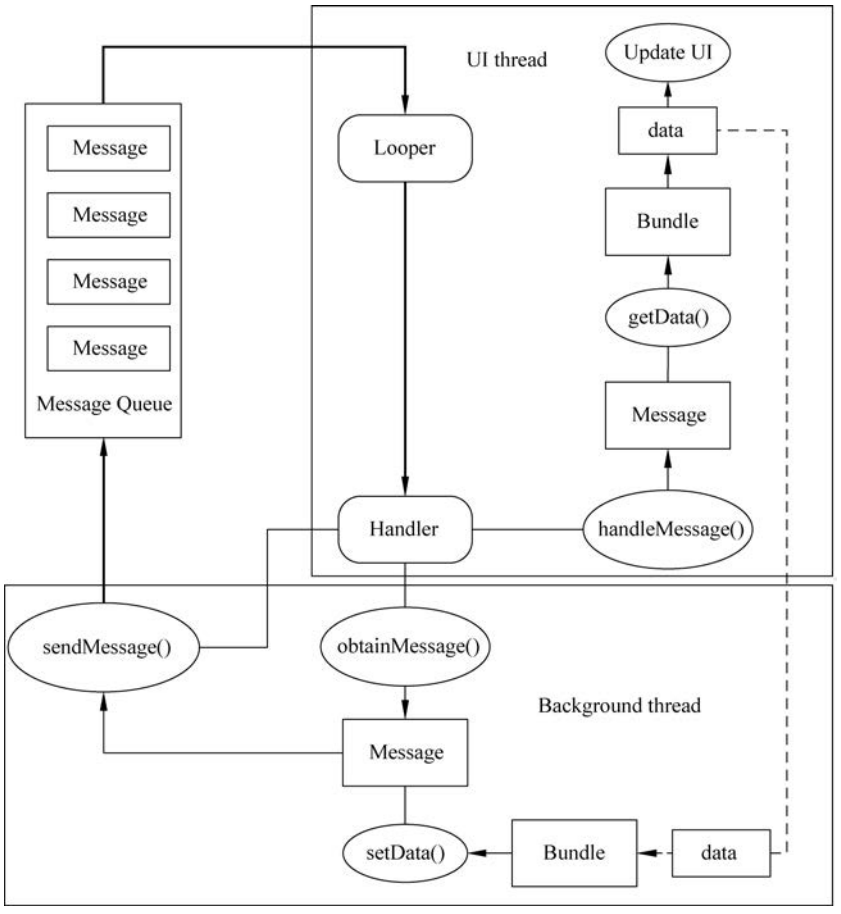

图 5-2　Handler 的工作方式

后台线程通过 Bundle 对象，按 key-value 的方式调用对应的方法存入数据后，使用主 UI 传进来的 Handler 对象获得消息对象 Message，对 Message 对象调用 setData()方法将 Bundle 对象放入消息中，最后通过 Handler 对象的 sendMessage()方法将消息发送出去。消息进入消息队列 MessageQueue 中，系统 Looper 会自动调度消息队列，当存在 Handler 对象对应的消息时，会从队列中取出消息发送给 Handler 对象，此时，Handler 对象会产生 handleMessage()回调。由于 Handler 是在前端 UI 线程中定义的，其 handleMessage()回调也在 UI 线程中执行。在 handleMessage()回调中会传入对应的 Message 对象，进而可从消息对象中通过 getData()方法获得 Bundle 对象，Bundle 对象则可通过 key 取出对应数据将之渲染到 UI 上。

总而言之，前端定义 Handler 对象，并处理 handleMessage()回调从消息中获得数据，用于更新 UI；后台线程通过 Handler 对象获得 Message 对象，将数据装入 Message，并通过 Handler 发送消息，两者配合实现了后台线程与前端 UI 的数据交互。当 Message 仅需要传递 int 数据时，可以不使用 Bundle，而是直接使用 Message 对象的 arg1 和 arg2 传递简单的 int 数据。

5.1.3 任务实现

1. 实现 UI 布局

MainActivity 的布局 my_main. xml 如代码 5-1 所示,其中 Button 可通过 android:enabled 属性控制初始使能状态,当属性值为 false 时,则 Button 使能被禁止,无法被单击。

代码 5-1 MainActivity 的布局文件 my_main. xml

```
<LinearLayout xmlns:android = "http://schemas.android.com/apk/res/android"
    xmlns:app = "http://schemas.android.com/apk/res - auto"
    android:orientation = "vertical"
    android:layout_width = "match_parent"
    android:layout_height = "match_parent">
    <TextView
        android:layout_width = "match_parent"
        android:layout_height = "wrap_content"
        android:text = "Your name and ID" />
    <TextView
        android:id = "@ + id/tv_result"
        android:layout_width = "match_parent"
        android:layout_height = "wrap_content"
        android:text = "0.00" />
    <LinearLayout
        android:layout_width = "match_parent"
        android:layout_height = "wrap_content"
        android:orientation = "horizontal">
    <Button
        android:id = "@ + id/bt_start"
        android:layout_width = "0dp"
        android:layout_height = "wrap_content"
        android:layout_weight = "1"
        android:text = "Start" />
    <Button
        android:id = "@ + id/bt_stop"
        android:layout_width = "0dp"
        android:layout_height = "wrap_content"
        android:layout_weight = "1"
        android:enabled = "false"
        android:text = "Stop" />
<!--    通过设置 android:enabled 属性可控制 Button 是否可用 -->
    </LinearLayout>
</LinearLayout>
```

2. 实现自定义线程 CounterThread

自定义后台线程 CounterThread 如代码 5-2 所示。在线程中通过构造方法传递主 UI 线程中定义的 Handler 对象,此外 Bundle 需要通过 key-value 方式存取 float 类型的计数值,建议将 key 定义为常量,本任务中,将计数值的 key 定义成私有常量 KEY_COUNTER。

线程的核心工作是改写 run()方法。在 run()方法中,对状态 isRunning 进行 while 循环判断,若 isRunning 为 true,则循环执行睡眠 10ms 后对计数值 counter 自增 0.01,并通过 Handler 对象将封装有 counter 的消息发送出去,从而主 UI 中的 Handler 对象会响应 handleMessage()回调,在回调方法中取出 counter 并更新 UI。Bundle 对象的取数涉及 key 操作,为了进一步减少耦合,可将取数作为后台线程的公开方法,即本任务中的

getMessageData()方法，供使用者调用，此时宜将 Handler 回调的 Message 对象作为 getMessageData()方法的传参，使 getMessageData()方法从 Message 对象中取出 Bundle，进而使用类中所定义的 KEY_COUNTER 取出 float 变量。考虑到线程可能消亡，因此 getMessageData()方法应该设计为静态的方法，使之不依赖于具体的线程对象。

在后台线程中，变量 isRunning 被定义为原子变量 AtomicBoolean，并且其取数和改值均调用原子变量所提供的方法，之所以这样做是因为在多线程中，如果多个线程共同访问(写值)某无锁变量是非线程安全的。在极端情况下，若多个线程试图同时修改同一个无锁变量，则会导致其出乎逻辑的错误，尤其是具有多字段的自定义类变量，变量的修改往往无法在一条机器指令周期内完成，会出现变量不同字段被不同线程同时修改，导致数据面目全非。而原子操作则能保证某线程访问该变量时，自动加锁，操作完后，释放锁，从而使该变量在加锁期间无法被其他线程修改，保证数据完整性安全。

按照 C 语言的逻辑，在 run()方法中，变量 isRunning 被设为 true 之后，有读者认为在 while 判断时，该循环是死循环。事实上，变量 isRunning 可被线程所提供的 stopCounter()方法修改为 false。只要外部的线程调用了 stopCounter()方法，随时可将变量 isRunning 修改为 false，使 while 循环结束，run()方法执行完毕，线程消亡。Thread.sleep()方法提供睡眠功能，使用时需要 try-catch 语句捕捉异常。注意，Thread.sleep()方法的睡眠时间并不是严格精确的，不能胜任精确定时的场合。

代码 5-2　自定义线程 CounterThread.java

```
1    import android.os.Bundle;
2    import android.os.Handler;
3    import android.os.Message;
4    import java.util.concurrent.atomic.AtomicBoolean;
5    public class CounterThread extends Thread{
6        private static final String KEY_COUNTER = "key_counter" ;
7        private AtomicBoolean isRunning;     //原子变量,使之多线程操作安全
8        private Handler handler;             //handler 由外部传入,在 UI 线程中定义 handler
9        //注意导入包是 android.os.Handler,而非 java.util.logging.Handler
10       public CounterThread(Handler handler) {
11           this.handler = handler;
12       }
13       @Override
14       public void run() {
15           isRunning = new AtomicBoolean(true);
16           float counter = 0.0f;
17           while (isRunning.get()){
18               try {
19                   Thread.sleep(10);          //睡眠 10ms,可能会有异常,用 try-catch 捕捉
20                   //非精准计时,与实际时间有误差
21               } catch (InterruptedException e) {
22                  e.printStackTrace();
23               }
24               counter += 0.01f;
25               Bundle bundle = new Bundle();
26               //Handler 对象利用 Bundle 打包数据
27               //Bundle 可理解为一个超级 HashMap,通过 key-value 存放数据
28               //Bundle 支持常用数据存取的指定方法,避免强制类型转换
29               bundle.putFloat(KEY_COUNTER,counter);
30               Message msg = handler.obtainMessage();  //从 Handler 对象中取出消息对象
```

```
                msg.setData(bundle);                    //将 Bundle 对象放入消息对象
                handler.sendMessage(msg);               //将消息通过 Handler 对象发送出去
                //句柄对象 Handler 在主 UI 中回调 handleMessage()处理消息,并更新 UI
            }
        }
        public void stopCounter(){
            isRunning.set(false);
            //isRunning 设置为 false,使得 run()方法中 while 条件不再成立,循环结束
        }
        public static float getMessageData(Message msg){
            //定义为 static 方法,使之不依赖于实例对象
            //解析消息数据直接在该类中实现有利于解耦,外部无需关心 Bundle 对象存放数据的 key
            Bundle bundle = msg.getData();
            float v = bundle.getFloat(KEY_COUNTER);
            return v;
        }
    }
```

3. 实现 MainActivity

MainActivity 的实现如代码 5-3 所示。线程在 run()方法执行完毕时即消亡,一般情况下被定义为局部变量,但是由于 MainActivity 类的 onCreate()方法中有匿名回调需要调用 Thread 对象的方法,而匿名回调要访问局部变量,就需要将该局部变量声明为 final,而 final 修饰的变量不能被多次重新创建,因此本任务中将线程对象设为成员变量,使之能被 MainActivity 类中的各方法访问。

Handler 对象创建时,可直接匿名实现 handleMessage()回调,在回调中调用 CounterThread 静态方法从 Message 对象中获得计数器值,并更新给 TextView 对象,从而实现了后台线程发送消息传递数据,UI 线程中回调 handleMessage()方法接收数据并更新 UI。

MainActivity 中,两个 Button 的使能状态须在单击事件中互相反转,其原因是,Button 对象 bt_start 负责创建并启动线程,Button 对象 bt_stop 负责停止线程,若线程还没有启动,单击 bt_stop 无意义,因此须在 bt_start 启动线程后才可以使能 bt_stop,使之具有可停止的线程对象。此外,本任务中只需要 1 个后台线程,因此 bt_start 启动线程后不能再启动新线程,需将 bt_start 自身使能禁止,避免反复单击创建新线程。

代码 5-3　MainActivity.java

```
import androidx.annotation.NonNull;
import androidx.appcompat.app.AppCompatActivity;
import android.os.Bundle;
import android.os.Handler;
import android.os.Message;
import android.view.View;
import android.widget.Button;
import android.widget.TextView;
import android.widget.Toast;
public class MainActivity extends AppCompatActivity {
    Handler handler;
    CounterThread thread;
    @Override
    protected void onCreate(Bundle savedInstanceState) {
```

```
        super.onCreate(savedInstanceState);
        setContentView(R.layout.my_main);
        TextView tv = findViewById(R.id.tv_result);
        Button bt_start = findViewById(R.id.bt_start);
        Button bt_stop = findViewById(R.id.bt_stop);
        handler = new Handler(new Handler.Callback() {
            @Override
            public boolean handleMessage(@NonNull Message msg) {
                //后台线程通过 Handler 对象调用 sendMessage()方法发送消息后产生的回调
                float f = CounterThread.getMessageData(msg);
                tv.setText(String.format("%.2f",f));     //更新计数器值
                return true;                             //Handler 事件不再往下传递
            }
        });
        bt_start.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View v) {
                bt_start.setEnabled(false);              //反转两个 Button 的使能状态
                bt_stop.setEnabled(true);
                thread = new CounterThread(handler);
                thread.start();        //启动线程 run()方法,run()结束时线程消亡
                //不能调用 thread.run(),该方法会在主 UI 中执行,而不是开辟后台线程执行
            }
        });
        bt_stop.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View v) {
                bt_start.setEnabled(true);
                bt_stop.setEnabled(false);
                thread.stopCounter();
                //修改 CounterThread 的 isRunning 为 false,循环结束,线程运行结束
            }
        });
    }
}
```

5.1.4　验证变量的线程安全性

1. 验证原子变量的线程安全性

为了进一步验证变量的线程安全与否,可自定义 1 个线程类 TestAtomicThread,如代码 5-4 所示。在线程中定义 1 个静态的原子变量计数器 counter 供多个后台线程共享,以及可控的增减方向变量 isAdd。当 isAdd 为 true 时,在循环中对 counter 自增 20,反之则自减 20,随后睡眠 1ms,循环程序执行 5000 次。TestAtomicThread 类提供 getCounter()方法获取计数器值,resetCounter()方法对计数器清零。

代码 5-4　自定义线程类 TestAtomicThread.java

```
import java.util.concurrent.atomic.AtomicLong;
public class TestAtomicThread extends Thread {
    private static AtomicLong counter = new AtomicLong(0);
    private boolean isAdd;
    public TestAtomicThread(boolean isAdd) {
        this.isAdd = isAdd;
    }
```

```
        @Override
        public void run() {
            for (int i = 0; i < 5000; i++) {
                if (isAdd) {
                    counter.getAndAdd(20l);
                } else {
                    counter.getAndAdd( - 20l);
                }
                try {
                    sleep(1);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        }
        public static AtomicLong getCounter() {
            return counter;
        }
        public static void resetCounter(){
            counter.set(0l);
        }
    }
```

在 MainActivity 的布局中增加 1 个 Button(id 为 bt_test),在 MainActivity 的 onCreate()方法中增加相应代码,所增加的内容如代码 5-5 所示。单击 bt_test 按钮,生成两个 TestAtomicThread 线程实例,在启动线程前,先对计数器清零。启动线程后,两个线程在各自的循环程序中对共享的计数器改值,其中,线程 t1 对计数器值每次自增 20,线程 t2 对计数器值每次自减 20,主 UI 线程中调用线程对象的 join()方法等待线程 t1 和线程 t2 运行结束。基于 join()方法是阻塞式操作,等待线程对象执行结束才能继续往下执行这一原理,在主 UI 中会一直等待线程 t1 和线程 t2 运行结束,并且在等待期间,主 UI 线程被阻塞,不能执行 UI 操作。当线程 t1 和线程 t2 执行结束时,主 UI 不被阻塞,此时 Toast 会显示 TestAtomicThread 的计数器值。鉴于原子变量是线程安全的,线程 t1 对计数器 5000 次的自增操作与线程 t2 对计数器的 5000 次自减操作的累计影响相抵消,因此,Toast 显示的计数器结果为 0。

代码 5-5　MainActivity 的 onCreate()方法对 TestAtomicThread 的验证代码

```
    Button bt_test = findViewById(R.id.bt_test);
    bt_test.setOnClickListener(new View.OnClickListener() {
        @Override
        public void onClick(View v) {
            TestAtomicThread t1 = new TestAtomicThread(true);
            TestAtomicThread t2 = new TestAtomicThread(false);
            TestAtomicThread.resetCounter();
            t1.start();
            t2.start();
            try {
                t1.join();       //阻塞操作,在主 UI 中等待线程 t1 运行结束
                t2.join();       //阻塞操作,在主 UI 中等待线程 t2 运行结束
                Toast.makeText(MainActivity.this,
                               "Counter = " + TestAtomicThread.getCounter(),
                               Toast.LENGTH_LONG).show();
```

```
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    });
```

2. 验证普通变量的非线程安全性

定义1个非线程安全类TestCommonThread，如代码5-6所示，该类中，将计数器counter更改为Long变量，并且在run()方法中对counter无加锁操作，因此线程的run()方法对counter操作是非线程安全的。

在TestCommonThread中还定义了resetCounter()静态方法，使用了synchronized关键字修饰，静态的synchronized方法相当于对TestCommonThread.class加锁，不管TestCommonThread生成多少个线程实例对象，在调用resetCounter()方法时，只有1个线程实例能对其操作，该实例完成resetCounter()方法调用后，其他线程实例才能调用该方法。

代码5-6　自定义线程TestCommonThread.java

```
public class TestCommonThread extends Thread{
    private static Long counter = 0l;
    private Boolean isAdd;
    public TestCommonThread(boolean isAdd) {
        this.isAdd = isAdd;
    }
    @Override
    public void run() {
        for (int i = 0; i < 5000; i++) {
            if (isAdd) {
                counter += 20l;
            } else {
                counter -= 20l;
            }
            try {
                sleep(1);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
    public static long getCounter() {
        return counter;
    }
    public static synchronized void resetCounter(){
        //静态方法加锁,等效于给 TestCommonThread.class 加锁
        //所有调用该方法的线程实例等待该方法解锁后才能继续调用 resetCounter()
        counter = 0l;
    }
}
```

MainActivity中的验证程序如代码5-7所示，在onCreate()方法中运行。当线程t1和线程t2运行结束后，Toast显示计数器值，运行结果验证了计数器值在多数情况下不为0。造成该结果的原因是：在线程的run()方法中，线程t1对counter自增操作尚未结束时，有

可能线程 t2 对 counter 同时进行自减操作,导致读写数据不一致,即非线程安全操作,导致最终结果与预期不一致。通俗地说,假设某时刻计数器 counter=100,线程 t1 在自增操作时读取了 counter 并在 CPU 中进行了加法运算,正要写回给内存(但还没有写入,此时 CPU 计算结果为 120,内存结果为 100)时,线程 t2 又访问了 counter,此时读取的值依然是 100,并做了减法运算(结果为 80),随后线程 t1 在内存中写回 120,紧接着线程 t2 写回内存 80。此时预期的计数器值为 100,由于没有对变量加锁的原因,导致计数器的真实值为 80,与预期不一致。在测试中,循环用了 5000 次是为了增加两个线程写回数据时产生冲突的概率。

代码 5-7　MainActivity 中对 TestCommonThread 的验证代码

```
Button bt_test = findViewById(R.id.bt_test);
bt_test.setOnClickListener(new View.OnClickListener() {
      @Override
      public void onClick(View v) {
          TestCommonThread t1 = new TestCommonThread(true);
          TestCommonThread t2 = new TestCommonThread(false);
          TestCommonThread.resetCounter();
          t1.start();
          t2.start();
          try {
             t1.join();     //阻塞操作,在主 UI 中等待线程 t1 运行结束
             t2.join();     //阻塞操作,在主 UI 中等待线程 t2 运行结束
             Toast.makeText(MainActivity.this,
                            "Counter = " + TestCommonThread.getCounter(),
                             Toast.LENGTH_LONG).show();
          } catch (InterruptedException e) {
             e.printStackTrace();
          }
      }
});
```

3. 使用 synchronized 修饰解决多线程访问冲突

解决多线程访问冲突问题,不一定都要使用原子变量。代码 5-8 对 TestCommonThread 进行修改,在 run()方法中对变量 counter 加了 synchronized 关键字修饰(见粗体代码),使对应的代码块具备加锁保护功能。当运行该代码块时,只有 1 个线程能运行,只有该线程运行完毕后,其他线程才能运行该代码块,此时对 counter 自增或自减操作是线程安全的,实际运行结果和预期结果均为 0。

代码 5-8　在 TestCommonThread 中给变量 counter 加锁

```
@Override
public void run() {
    for (int i = 0; i < 5000; i++) {
        synchronized (counter) {
            if (isAdd) {
                counter += 20l;
            } else {
               counter -= 20l;
            }
        }
        try {
            sleep(1);
        } catch (InterruptedException e) {
```

```
14                e.printStackTrace();
15            }
16        }
17    }
```

为了测量两个线程的运行时间，在 MainActivity 中修改了相关代码，具体见代码 5-9 的粗体内容。在线程启动之前，对 bt_start 按钮调用 performClick()方法，相当于对 bt_start 按钮产生单击操作，等到 t1 和 t2 两个线程结束后，调用 bt_stop 按钮的 performClick()方法，即结束定时，此时 TextView 上显示的即为两个线程跑完的时间(非精确测量)。

代码 5-9　MainActivity 中模拟 Button 的单击行为

```
1     bt_test.setOnClickListener(new View.OnClickListener() {
2         @Override
3         public void onClick(View v) {
4             TestCommonThread t1 = new TestCommonThread(true);
5             TestCommonThread t2 = new TestCommonThread(false);
6             TestCommonThread.resetCounter();
7             bt_start.performClick();     //用代码控制 bt_start 按钮产生单击行为
8             t1.start();
9             t2.start();
10            try {
11                t1.join();               //阻塞操作,在主 UI 中等待线程 t1 运行结束
12                t2.join();               //阻塞操作,在主 UI 中等待线程 t2 运行结束
13                bt_stop.performClick(); //用代码控制 bt_stop 按钮产生单击行为
14                //tv 上显示 t1 和 t2 两个线程运行结束时产生的后台计数时间
15                Toast.makeText(MainActivity.this,
16                               "Counter = " + TestCommonThread.getCounter(),
17                               Toast.LENGTH_LONG).show();
18            } catch (InterruptedException e) {
19               e.printStackTrace();
20            }
21        }
22    });
```

多次运行的结果是：计数器 counter 最终值为 0。因此代码 5-8 的操作可行，TextView 上显示的时间约为 10.80s。如果指令运算的时间忽略不计，线程 t1 和线程 t2 几乎同时运行，理论值应该在 5.00s 左右(5000 次循环，每次循环消耗 1ms)，但是由于线程的 sleep()方法为非精确定时，加上线程切换调动的开销，使得实际消耗时间远大于理论值。

作为比较，运行代码 5-6，软测量的时间约为 10.70s，与加了同步锁的代码 5-8 相差不大。若在 synchronized 代码块中，不小心将 sleep()方法的相关代码放在同步代码块，如代码 5-10 所示，则在计数器 counter 访问冲突时，sleep()方法的执行也被包含在同步锁中，需等当前线程 sleep()方法结束才能将代码块释放给另一个线程，此时会无形中增加线程之间的消耗时间。运行代码 5-10 的软测量结果为 12.20s，显然在访问冲突时由于 sleep()方法的执行，增加了总运行时间。作为参考，对 Atomic 变量的测量结果为 10.80s 左右，与代码 5-8 相当。

修改代码 5-10，使用 synchronized (TestCommonThread.class)，即对 TestCommonThread 类加锁，当该类的线程实例 t1 和 t2 运行同步代码块时，只能有 1 个线程运行，此时 counter 变量最终结果依然是 0，符合预期，但是 t1 和 t2 几乎全程不能并行执行，运行消耗的时间大

幅增加,软测量结果为 21.40s。

若将锁加在线程的变量 isAdd 上,即改成 synchronized (isAdd),由于 isAdd 不是静态变量,线程实例 t1 和 t2 都有各自的变量 isAdd,此时,同步锁针对的是两个不同的变量 isAdd,从而导致线程 t1 和线程 t2 相互独立运行,同步锁不起作用,计数器最终值不为 0,运行软测量结果为 10.76s。若将同步锁改成线程实例,即 synchronized (this),由于线程 t1 和线程 t2 不是同一个实例,同样,同步锁不起作用,计数器最终值不为 0,运行软测量结果为 10.76s。

由此可见,synchronized 同步代码的"陷阱"比较多,稍不注意,就可能发生预期之外的错误或者牺牲了性能。对于提供了原子操作的变量,尽量在多线程访问冲突场合使用原子变量。

代码 5-10　牺牲性能的同步代码

```
synchronized (counter) {
    if (isAdd) {
       counter += 20l;
    } else {
       counter -= 20l;
    }
    try {
        sleep(1);
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
}
```

5.2　使用多线程与自定义接口

5.2.1　任务说明

本任务功能同 5.1 节的任务,但在实现方式上,采用了自定义接口,利用接口回调方法实现后台线程向前端 UI 传递数据。

自定义接口回调避免了通过 Handler 传递数据,因此后台线程的写法更像传统 Java 的写法,但在细节上依然存在差异,后台线程自定义接口回调的方法虽然是在前端 UI 线程中实现的,但本质上,却是在后台线程中执行的,若前端实现的回调方法中涉及 UI 更新,则实际上是在后台线程中更新 UI,这在 Android 系统中是不允许的。

接口回调可以理解成一种特殊的方法占位符,定义和使用接口回调的地方占位了回调方法,并传递了所需要的数据,而实现接口回调的地方对回调方法所传递的数据进行处理。通过这种方式,就能实现代码的抽象与解耦。占位接口回调的只用关心什么时候需要触发该接口回调,并通过接口回调传递什么数据出去,而不必关心使用者如何实现该接口回调,对传递的数据做如何处理;实现接口回调的只用关心什么时候接收接口回调的数据,如何使用所传递的数据或更新 UI,而不必关心接口回调是怎么触发的、触发机制是什么、在具体的哪一行代码触发。因此,接口具有生产者和观察者特性,生产者产生数据,并占位接口回调传递数据,观察者实现接口回调,使用所传递的数据做事务逻辑处理或者更新 UI。

鉴于实现接口回调的代码本质上是在接口占位中执行的，若接口回调的实现方法中涉及UI更新，则后台线程接口占位不能直接写在线程的run()方法中，而是需要将接口占位放在主UI线程中。Android提供了对应的解决办法，可将Activity对象传递到后台线程，利用Activity对象所提供的runOnUiThread()方法将后台线程切换至Activity所在的UI线程，进而可在runOnUiThread()方法的匿名Runnable中占位自定义的接口回调，解决了后台线程不能更新UI的问题。

5.2.2　任务实现

1. 实现后台线程 CounterThread

MainActivity的布局文件参考5.1节。后台线程CounterThread如代码5-11所示，鉴于后台线程需要Activity对象的runOnUiThread()方法进行UI线程切换，因此将Activity对象作为类成员，并通过构造方法传递该对象。

在CounterThread中，自定义了OnUpdateListener接口和对应的onUpdate(float counter)方法，通过该方法将计数器值counter传递给使用线程的Activity。此外，由于Activity对象需要调用runOnUiThread()切换线程，并将匿名实现的Runnable接口作为切换线程方法的参数，涉及了匿名回调访问计数器值，若counter作为final的局部变量，则无法被重新赋值，因此counter只能作为类成员变量实现全局访问。CounterThread类提供了setOnUpdateListener()方法，使Activity在使用线程时，可通过该方法实现OnUpdateListener接口，并在接口回调方法中使用计数器值更新UI。

CounterThread的run()方法中，依然是睡眠10ms，更新一次计数器值，并通过Activity对象的runOnUiThread()切换线程。为了防止Activity实例因没有调用setOnUpdateListener()方法实现OnUpdateListener接口，而导致接口对象为null产生空对象错误，占位接口回调时，须先对接口对象判断是否为空，再调用接口对象所提供的onUpdate(counter)方法，将计数器值counter传递出去。当Activity的OnUpdateListener接口被触发时，本质上，在Activity中实现的OnUpdateListener接口和回调方法，在代码5-11第36行处执行，而这些代码是在runOnUiThread()方法中执行的，已从后台线程切换到Activity对象的UI线程，既保证了后台线程传递数据，又保证了在UI线程中接收数据更新UI。

代码5-11　使用自定义接口的线程 CounterThread.java

```
1    import android.app.Activity;
2    import java.util.concurrent.atomic.AtomicBoolean;
3    public class CounterThread extends Thread{
4        private AtomicBoolean isRunning;            //原子变量多线程操作是安全的
5        private Activity activity;                  //使用 Activity 对象切换 UI 线程
6        private OnUpdateListener onUpdateListener; //定义自定义接口对象
7        private float counter;     //全局变量 counter 在切换线程的匿名 run()方法中被访问
8        public CounterThread(Activity activity) {
9            this.activity = activity;               //Activity 对象从外部传入
10           //可直接传 MainActivity.this
11       }
12       public interface OnUpdateListener {
13           void onUpdate(float counter);           //利用自定义接口传递计数器值
14       }
15       public void setOnUpdateListener(OnUpdateListener onUpdateListener) {
```

```
            this.onUpdateListener = onUpdateListener;       //传递外部实现的接口
            //外部实现的接口得到计数器值,并更新 UI
        }
        @Override
        public void run() {
            isRunning = new AtomicBoolean(true);
            counter = 0.0f;
            while (isRunning.get()){
                try {
                    Thread.sleep(10);  //睡眠 10ms,可能会有异常,用 try - catch 捕捉异常
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
                counter += 0.01f;
                activity.runOnUiThread(new Runnable() {
                    //利用 Activity 对象切换到主 UI 线程
                    //后台线程 Thread 不能更新 UI
                    @Override
                    public void run() {
                        if(onUpdateListener!= null){
                            onUpdateListener.onUpdate(counter);
                            //通过自定义接口将计数器值传给接口实现者,并更新 UI
                        }
                    }
                });
            }
        }
        public void stopCounter(){
            isRunning.set(false);
            //将 isRunning 设置为 false,使得 run()循环结束
        }
    }
```

2. 实现 MainActivity

MainActivity 的实现如代码 5-12 所示。MainActivity 不需要使用 Handler 传递数据，其实现逻辑相比 5.1 节要简单，启动后台线程后，对线程对象调用 setOnUpdateListener()方法，并匿名实现 OnUpdateListener 接口，在接口回调中获得后台线程的计数器值，并更新到 TextView 对象上。

代码 5-12　MainActivity.java

```
import androidx.appcompat.app.AppCompatActivity;
import android.os.Bundle;
import android.view.View;
import android.widget.Button;
import android.widget.TextView;
public class MainActivity extends AppCompatActivity {
    CounterThread thread;
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.my_main);
        TextView tv = findViewById(R.id.tv_result);
        Button bt_start = findViewById(R.id.bt_start);
        Button bt_stop = findViewById(R.id.bt_stop);
```

```
15          bt_start.setOnClickListener(new View.OnClickListener() {
16              @Override
17              public void onClick(View v) {
18                  bt_start.setEnabled(false);       //反转两个 Button 的使能状态
19                  bt_stop.setEnabled(true);
20                  thread = new CounterThread(MainActivity.this);
21                  thread.setOnUpdateListener(new CounterThread.OnUpdateListener() {
22                      //设置线程的接口回调,在回调方法中得到计数器值,并更新 UI
23                      @Override
24                      public void onUpdate(float counter) {
25                          tv.setText(String.format("%.2f",counter));
26                      }
27                  });
28                  thread.start();                   //启动后台线程
29              }
30          });
31          bt_stop.setOnClickListener(new View.OnClickListener() {
32              @Override
33              public void onClick(View v) {
34                  bt_start.setEnabled(true);
35                  bt_stop.setEnabled(false);
36                  thread.stopCounter();
37              }
38          });
39      }
40  }
```

5.3　使用多线程与 LiveData

5.3.1　任务说明

本任务的演示效果如图 5-3 所示,相比 5.1 节的任务,在功能逻辑上要复杂一些。单击 START 按钮后,启动后台线程,每隔 0.01s 更新 1 次计数值,通过 LiveData 的实现类 MutableLiveData 感知数据并在 UI 中更新。START 按钮被单击后会变成 PAUSE 按钮,同时使能 STOP 按钮。PAUSE 按钮有两种行为,具体如下所述。

图 5-3　任务的演示效果

(1) 在 PAUSE 按钮状态下,单击 STOP 按钮,定时器结束,PAUSE 按钮变成 START 按钮,STOP 按钮禁止。

(2) 单击 PAUSE 按钮,该按钮变成 RESUME 按钮,并且 STOP 按钮禁止,定时器暂停。

单击 RESUME 按钮,定时器工作,并且继续在上一次计数值的基础上计数,RESUME 按钮变成 PAUSE 按钮,STOP 按钮使能。

相比 5.1 节的任务,START 按钮身兼数职,具有重新启动计数、暂停计数、继续计数的功能。在逻辑上,通过 STOP 按钮重置 START 按钮,而 START 按钮的单击行为则会进入 PAUSE 和 RESUME 的循环状态。

在实现上,若将后台线程计数器值设为静态变量,不管是通过 Handler 还是自定义接口,均能实现类似的功能,但是所有的计数器线程共享 1 个静态变量,无法做到多个 Activity 或者 Fragment 拥有各自独立的后台计数器。本任务使用了全新的方法,即 LiveData 来实现后台线程与 UI 线程的交互,LiveData 与视图模型绑定后,可让不同的 Activity 或者 Fragment 拥有独立的 LiveData 数据。

5.3.2 任务实现

1. LiveData 的特点与使用方法

LiveData 是 Jetpack 提供的一种响应式编程组件,它可以包含任何类型的数据,并在数据发生变化的时候通知观察者,适合与 ViewModel(视图模型)结合在一起使用,可以让 ViewModel 将数据的变化主动通知给 Activity 或者 Fragment。

LiveData 本身是抽象类,一般会使用实现类 MutableLiveData < T >定义 LiveData 数据,其中 T 是泛型,即将需要被观察的对象的数据类型作为 MutableLiveData 的泛型定义该数据,MutableLiveData 对象具有 setValue()和 postValue()两种常用的改值方法,其中 setValue()方法在 UI 线程中使用,postValue()方法在后台线程和 UI 线程中均可使用。UI 线程获得 MutableLiveData 对象,并对其实现 Observer 接口,则在接口回调 onChanged()方法中,会获得 MutableLiveData 对象 setValue()或者 postValue()的更新值,进而可将更新值用于 UI 渲染。

一般而言,MutableLiveData 不会单独使用,而是放在继承了 ViewModel 的自定义视图模型中,使之具有生命周期的管理功能。当视图模型的拥有者(Activity 或者 Fragment)没有消亡,则视图模型中的 MutableLiveData 也不会消亡。从另外一个角度理解,当后台线程和前端 UI 均使用相同的视图拥有者,则后台线程不管生成消亡多少次,其从视图模型中获得的 MutableLiveData 与前端 UI 的 MutableLiveData 始终是同一个对象,从后台线程的角度看,多次线程的创建和消亡不会导致 MutableLiveData 消亡,使之具有静态变量的特征。但是 MutableLiveData 比静态变量更强大,当多个 Activity 使用后台线程时,由于不同 Activity 对视图模型而言属于不同的拥有者,因此对应的 MutableLiveData 属于不同的对象,从而不同 Activity 对象调用后台线程,只要后台线程处理好与对应 Activity 的拥有者关系,就能实现不同 Activity 以及所对应的后台线程具有相互独立的 MutableLiveData 对象。对本任务而言,即不同的 Activity 拥有各自独立的后台线程计数器,每个 Activity 均可以对其实现重启、暂停、继续和停止操作,互不影响,若是使用后台线程的静态变量则难以实现该

功能，各个 Activity 只能共享同一个静态变量，无法做到互不干扰。

2. 创建自定义视图模型 CounterViewModel

自定义视图模型 CounterViewModel 如代码 5-13 所示，该类继承自 ViewModel。CounterViewModel 的设计比较简单，定义了 1 个私有变量计数器 counter，使用 MutableLiveData < Float >类型定义，即该数据是 Float 泛型的 MutableLiveData 数据，在 CounterViewModel 类中实现 getCounter()方法，使之返回该私有数据。泛型只接受类数据，float 类型的计数器在 MutableLiveData 中须使用 Float 泛型。在 MutableLiveData 构造方法中，对 counter 实例化，并且设置了计数器初始值 0。若采用无构造方法的视图模型，可在声明成员变量时直接对其实例化。

代码 5-13　自定义视图模型 CounterViewModel. java

```
1    import androidx.lifecycle.MutableLiveData;
2    import androidx.lifecycle.ViewModel;
3    public class CounterViewModel extends ViewModel {
4        //在自定义的 CounterViewModel 中定义 MutableLiveData
5        private MutableLiveData < Float > counter;
6        //MutableLiveData 数据需要泛型定义数据类型,泛型只接受类,float 对应类是 Float
7        public CounterViewModel() {
8            counter = new MutableLiveData <>();        //对 counter 实例化
9            counter.setValue(0.0f);                    //设置初始值
10       }
11       public MutableLiveData < Float > getCounter() {
12           return counter;
13       }
14   }
```

取视图模型中的变量 counter，需要先取得视图模型，可通过以下方式获得视图模型。

```
CounterViewModel counterViewModel = new ViewModelProvider(owner)
                                        .get(CounterViewModel.class);
```

其中，owner 是拥有者，可用 Activity(或者 Fragment)的实例，如 MainActivity. this。得到视图模型对象后，可调用 getCounter()方法，获得使用 MutableLiveData < Float >定义的计数器 counter。变量 counter 具有数据感知能力，可在后台线程中被改值，并在 UI 线程中被观测。

3. 实现后台线程 CounterThread

CounterThread 的实现如代码 5-14 所示，在构造方法中，传入视图拥有者 ViewModelStoreOwner 对象，以便于通过拥有者获得视图模型以及对应的 LiveData 数据。后台线程相比之前的任务，略有不同，即计数器的处理有两种模式。

(1) 清零模式，将计数器清零，重新开始计数。

(2) 暂停模式，计数器在原有基础上计数。

为了便于识别后台线程计数器的处理模式，在 CounterThread 中定义了计数模式变量 mode 和两个常量 MODE_RESTART(清零模式)以及 MODE_RESUME(暂停模式)。当构造方法使用单参数构造时，计数器默认为清零模式。

后台线程的 run()方法中，通过拥有者获得视图模型，进而调用视图模型的 getCounter()方法获得 LiveData 的计数器对象 counter。计数任务中，对线程模式进行判断，若是清零模

式,则计数器 counter 通过 postValue()方法重新设为 0;若是暂停模式,则利用视图模型数据不会消亡的特点,可在原有值基础上计数。在循环计数中,计数器的更新值使用 postValue()方法。在 UI 线程中,对通过视图模型获得的计数器 counter 实现了 Observer 侦听接口,当后台线程的计数器通过 postValue()方法更新值后,UI 线程中的计数器则能通过 Observer 接口感知数据变动,进而触发 onChanged()回调方法,回调方法的传参就是计数器的更新值,可被 UI 线程取出用于 UI 更新。

代码 5-14　使用 LiveData 的后台线程 CounterThread.java

```
import androidx.lifecycle.MutableLiveData;
import androidx.lifecycle.ViewModelProvider;
import androidx.lifecycle.ViewModelStoreOwner;
import java.util.concurrent.atomic.AtomicBoolean;
public class CounterThread extends Thread{
    private ViewModelStoreOwner owner;
    private AtomicBoolean isRunning;
    private MutableLiveData<Float> counter;
    private int mode;
    public static final int MODE_RESTART = 0;          //计数器清零模式
    public static final int MODE_RESUME = 1;           //计数器暂停模式
    public CounterThread(ViewModelStoreOwner owner,int mode) {
        this.owner = owner;
    //可以不传 owner,而是直接传 ViewModel 对象
    //使用传参 owner,验证后台线程与前端 UI 使用同一个 owner 获得的视图模型是否相同
    //相同视图模型实例对应的 LiveData 是同一个对象
        this.mode = mode;
    }
    public CounterThread(ViewModelStoreOwner owner) {
        this.owner = owner;
        mode = MODE_RESTART;
    }
    @Override
    public void run() {
        isRunning = new AtomicBoolean(true);
        CounterViewModel counterViewModel = new ViewModelProvider(owner)
                                            .get(CounterViewModel.class);
    //从 CounterViewModel 中获得计数器变量,该变量具有 Observer 接口
    //在主 UI 中,可对该变量实现 Observer 接口,感知变量的变动
       counter = counterViewModel.getCounter();
    //counter 不会随线程消亡,而是在 owner 的生命周期中一直存在
       if(mode == MODE_RESTART) {
       //在 MODE_RESTART 模式,计数器重置为 0
           counter.postValue(0.0f);
           //在线程中使用 postValue()改值,在主 UI 中,setValue()和 postValue()均可
       }
       while (isRunning.get()){
           try {
               Thread.sleep(10);
           } catch (InterruptedException e) {
               e.printStackTrace();
           }
           counter.postValue(counter.getValue() + 0.01f);
```

```
44                  //在主 UI 中通过 counter 的 Observer 接口和回调方法取出更新值,更新 UI
45              }
46          }
47          public void stopCounter(){
48              isRunning.set(false);
49          }
50      }
```

4. 实现 MainActivity

本任务 MainActivity 的布局文件同 5.1 节。MainActivity 的实现如代码 5-15 所示。在 onCreate()方法中,视图模型通过 ViewModelProvider 获得,进而通过视图模型得到 MutableLiveData<Float>类型的计数器对象 counter,此时,由于 MainActivity 和 CounterThread 使用的视图模型拥有者均是 MainActivity 实例,因此两者是相同的视图模型,对应的计数器 counter 也是相同的对象实例。后台线程的计数器通过 postValue()方法改值后,MainActivity 前端 UI 通过 Observer 侦听感知数据变动,并通过 onChanged()回调方法获得更新后的值,进而更新 UI。

按钮 bt_start 的行为逻辑相比之前的任务略显复杂,初始状态是 START,被单击后,按钮文本更新为 PAUSE,即可用该按钮暂停计数;当 bt_start 再次被单击,PAUSE 按钮变为 RESUME 按钮,并调用线程的 stopCounter()方法结束线程,此时视图模型中的计数器 counter 生命周期跟随拥有者 MainActivity,不会因线程消亡而消亡;当 RESUME 按钮被单击时,后台线程不对计数器 counter 清零,而是直接在上一次取值基础上改值,从而实现计数器继续计数的功能,此时线程并不是暂停,而是被重新创建并启动。单击 STOP 按钮,线程结束,bt_start 恢复为 START 按钮,计数器处于清理模式,若再次启动线程,计数器从 0 开始计数。

代码 5-15　使用 LiveData 的 MainActivity.java

```
1   import androidx.appcompat.app.AppCompatActivity;
2   import androidx.lifecycle.MutableLiveData;
3   import androidx.lifecycle.Observer;
4   import androidx.lifecycle.ViewModelProvider;
5   import android.content.Context;
6   import android.content.Intent;
7   import android.os.Bundle;
8   import android.view.View;
9   import android.widget.Button;
10  import android.widget.TextView;
11  public class MainActivity extends AppCompatActivity {
12      CounterThread thread;
13      @Override
14      protected void onCreate(Bundle savedInstanceState) {
15      super.onCreate(savedInstanceState);
16      setContentView(R.layout.my_main);
17      CounterViewModel counterViewModel = new ViewModelProvider(this)
18                                              .get(CounterViewModel.class);
19      //从 CounterViewModel 中获得 MutableLiveData 数据,生命周期同 MainActivity
20      MutableLiveData<Float> counter = counterViewModel.getCounter();
21      TextView tv = findViewById(R.id.tv_result);
22      //MutableLiveData 数据具有侦听功能,在 Observer 接口中感知数据的变动
```

```
        counter.observe(this, new Observer<Float>() {
            @Override
            public void onChanged(Float aFloat) {
                String s = String.format("%.2f", aFloat);
                tv.setText(s);
            }
        });
        Button bt_start = findViewById(R.id.bt_start);
        Button bt_stop = findViewById(R.id.bt_stop);
        bt_start.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View v) {
                //状态: Start->Pause->Resume->Pause->Resume...
                //或者 Start->(Stop)->Start
                String start_tag = bt_start.getText().toString();
                if(start_tag.equalsIgnoreCase("Start")) {
                   thread = new CounterThread(MainActivity.this);
                   bt_stop.setEnabled(true);
                   bt_start.setText("Pause");
                   thread.start();
                }else if(start_tag.equalsIgnoreCase("Pause")){
                   thread.stopCounter();
                   bt_stop.setEnabled(false);
                   bt_start.setText("Resume");
                }else {
                   thread = new CounterThread(MainActivity.this,
                                              CounterThread.MODE_RESUME);
                   //使用 MODE_RESUME,计数器值不清 0,在上一次取值基础上计数
                   bt_start.setText("Pause");
                   bt_stop.setEnabled(true);
                   thread.start();
                }
            }
        });
        bt_stop.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View v) {
                bt_start.setText("Start");        //bt_start 须恢复成 START 按钮
                bt_stop.setEnabled(false);
                thread.stopCounter();
            }
        });
    }
}
```

5. 使用两个 Activity 测试视图模型

感兴趣的读者可尝试创建两个不同的 Activity,各自拥有自身的视图模型,使之拥有各自独立的 LiveData 计数器,并观察两个计数器是否存在相互干扰(事实上没有干扰)。具体做法,可在 my_main.xml 布局文件上增加 1 个 Button(id 为 bt_jump),用于跳转至另一个 Activity,并在 MainActivity 的 onCreate()方法中增加如下代码:

```
findViewById(R.id.bt_jump).setOnClickListener(new View.OnClickListener() {
//R.id.bt_jump 为 my_main.xml 布局中跳转 Activity 所需的 Button
  @Override
  public void onClick(View v) {
```

```
        Context ctx = getApplicationContext();
        Intent i = new Intent(ctx,MainActivity2.class);
        startActivity(i);
    }
});
```

其中，getApplicationContext()方法可获得当前 Activity 的上下文，Intent 为意图，用于启动 Activity 或者 Service 对象。本任务中，使用 Intent 双参数构造方法，第 1 个参数为上下文，第 2 个参数为要启动的 Activity 类，得到 Intent 对象后，可通过 startActivity()方法，将该意图作为 Activity 启动对象，从而实现不同活动页面之间的跳转功能。

MainActivity2 的实现可直接复制自 MainActivity 文件，利用 Refactor 的 Rename 功能，修改为 MainActivity2，并将 Button 对象 bt_jump 的事件触发代码改为：

```
findViewById(R.id.bt_jump).setOnClickListener(new View.OnClickListener() {
    @Override
    public void onClick(View v) {
        Context ctx = getApplicationContext();
        Intent i = new Intent(ctx,MainActivity.class);
        startActivity(i);
    }
});
```

修改后，项目中有两个 Activity：MainActivity 和 MainActivity2，若要使两者均能被应用启动，则需要在配置文件中增加 activity 声明。打开 AndroidManifest.xml 文件，修改成如下配置。

```
<application
    android:allowBackup = "true"
    android:icon = "@mipmap/ic_launcher"
    android:label = "@string/app_name"
    android:roundIcon = "@mipmap/ic_launcher_round"
    android:supportsRtl = "true"
    android:theme = "@style/Theme.Tcf_task4_1V3">
    <activity
        android:name = ".MainActivity"
        android:launchMode = "singleInstance"
        android:exported = "true">
        <intent-filter>
            <action android:name = "android.intent.action.MAIN" />
            <category android:name = "android.intent.category.LAUNCHER" />
        </intent-filter>
    </activity>
    <activity android:name = ".MainActivity2"
        android:parentActivityName = ".MainActivity"
        android:launchMode = "singleInstance">
    </activity>
</application>
```

在 AndroidManifest.xml 文件中，activity 标签是活动页面(Activity)的申明标签，若应用中定义了若干个 Activity，每个 Activity 均需要使用 activity 标签进行申明，其中，android:name 指向的是 Activity 的类名称，android:launchMode="singleInstance"是启动单例模式，即应用中启动对应 Activity 时，若该 Activity 已存在，则切换到对应任务栈显示

该 Activity,而不是重新生成一个新的 Activity。

<action android:name="android.intent.action.MAIN"/>设置该 Activity 是应用的默认启动的活动页面,类似于 main()函数。"<category android:name="android.intent.category.LAUNCHER"/>"设置该应用在应用列表中可见。在 MainActivity2 的标签中,还增加了 android:parentActivityName=".MainActivity"属性,指明 MainActivity2 是由 MainActivity 跳转过来的,因此 MainActivity2 活动页面的动作栏上有左箭头,单击左箭头即可从当前活动页面跳转回 parentActivity 指向的 Activity,即 MainActivity。

5.4 使用 Okhttp 和 Gson 获取 Web API 数据

5.4.1 任务说明

本任务的演示效果如图 5-4 所示。在该应用中,活动页面视图根节点为垂直的 LinearLayout,在布局中依次放置 1 个 TextView,用于显示个人信息;1 个 EditText,用于显示获取数据的网址;1 个 ASYNC MODE 按钮,id 为 bt_async,以异步方式调用 Okhttp 获取数据;1 个 BLOCK MODE 按钮,id 为 bt_block,以同步(阻塞)方式调用 Okhttp 获取数据;1 个 ListView,用于显示 Web API 所获取的数据。

图 5-4 获取 Web API 数据演示效果

Web API 服务可通过运行本书所提供的基于 Node.js 所写的 Web 后端应用提供,网址需根据读者运行 Web 服务的设备 IP 地址进行更改,端口号默认为 8080。此外,读者还可以使用 Android 经典书籍《第一行代码 Android》(作者:郭霖)中提供的中国城市数据 Web API(网址请扫描前言中的二维码获取)。Web API 回传数据为 JSON 格式,通过该 API 可获得中国省份列表、省内地级市列表和地级市内各区县列表。本任务将使用第三方库 Okhttp 访问 Web API,并使用 Gson 解析 JSON 数据,最终将 JSON 数据转为列表数据在 ListView 中显示。

5.4.2 任务实现

1. 导入第三方库

本任务使用了第三方库 Okhttp 和 Gson,须在项目中导入后,方能使用。第三方库有多种导入方法,最常用的方式是在 Module Gradle 文件中导入对应资源。第三方库的导入地址和版本号可在 GitHub 官网(网址请扫描前言中的二维码获取)或者 MVN Repository 官网(网址请扫描前言中的二维码获取)等代码仓库中搜索。这里以 MVN Repository 为例,搜索 Gson,并选择 2.9.0 版本,即可进入对应仓库界面,如图 5-5 所示,资源包既可通过下载 jar 文件进行离线配置,也可通过 Maven、Gradle、SBT 和 Ivy 等方式进行在线配置,选择 Gradle(Short)选项,即可得到对应配置语句 implementation 'com.google.code.gson:gson:2.9.0',进

而可在 Gradle 文件中进行在线配置。

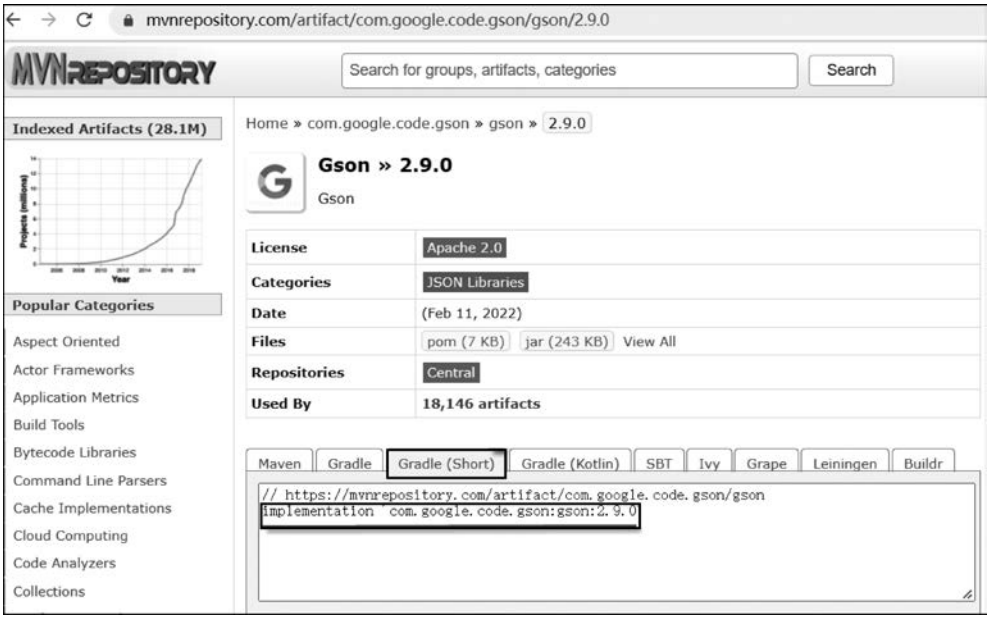

图 5-5　通过 MVN Repository 搜索 Gson 所获得的 Gradle 配置

本任务所需的 Okhttp 和 Gson 第三方依赖库，选择在 Module Gradle 文件中进行在线配置。如图 5-6 所示，在 Android 项目中有多个 Gradle 文件，依赖包以及 SDK 等配置主要在 Module Gradle 文件中完成。

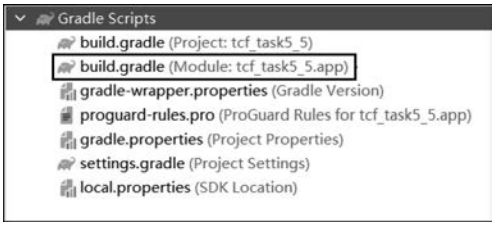

图 5-6　Android 项目中的 Gradle 文件

在 Module Gradle 文件的 dependencies 节点中，已存在若干依赖库，不同版本 Android Studio 所创建的项目，其依赖库的版本号甚至依赖库的名称均可能发生变化，这就会导致同一项目在不同版本 Android Studio 中打开时，会使其自动下载对应的 Gradle 文件和依赖库。本任务的依赖库配置如代码 5-16 所示，Gradle 文件一旦检测到依赖库发生了变化，会在 IDE 右上角出现 Sync Now 按钮，单击该按钮，即可在线下载相关依赖库。

代码 5-16　Module Gradle 文件添加 Okhttp 和 Gson 依赖

```
dependencies {
    implementation 'androidx.appcompat:appcompat:1.3.0'
    //appcompat 是项目默认创建的依赖库，不同 Android Studio 版本号会有区别
    …
    //以下是项目额外增加的第三方依赖库
    implementation 'com.squareup.okhttp3:okhttp:4.9.3'
    implementation 'com.google.code.gson:gson:2.9.0'
    //在线加载 Okhttp 和 Gson 第三方依赖库，单击 IDE 右上角 Sync Now 按钮进行更新
    …
}
```

2. 添加 Internet 上网权限

Android 系统会对应用限制访问数据或执行操作的权限，若应用没有在声明文件中声

明 Internet 权限，默认无法访问 Internet 资源。上网权限属于低级权限，只需要在 AndroidManifest.xml 文件中添加静态权限即可。若是读写系统通讯录则属于高级权限，对于 Android 6.0 及以上版本的系统，除了在 AndroidManifest 中配置权限外，还需要在代码中调用 Runtime Permission(运行时权限)。

在 AndroidManifest.xml 中添加静态权限，既可添加在 application 标签之前，也可在该标签之后。对于 Android 10.0 及以上系统，出于安全考虑，默认只能访问 Https 协议网络资源，不能访问 Http 协议网络资源，因此还需要在 application 标签中添加 android:usesCleartextTraffic="true"属性，使其能访问 Http 资源。

本任务的 AndroidManifest 配置文件修改部分如代码 5-17 所示，在 application 标签内添加了 android:usesCleartextTraffic 属性，使得 Android 10.0 及以上系统能访问 Http 资源，并在 application 标签之外通过 uses-permission 标签添加了 Internet 访问权限。

代码 5-17　AndroidManifest.xml 文件中配置 Internet 权限

```
<manifest …>
    <application
        …
        android:usesCleartextTraffic="true"
        …>
        <activity
            …
        </activity>
…
    </application>
    <uses-permission android:name="android.permission.INTERNET">
    </uses-permission>
</manifest>
```

3. 解析 JSON 数据

JSON(JavaScript Object Notation)是一种轻量级的数据交换格式，比 XML 更小、更快、更易于解析，常在 Web API 中使用。JSON 数据分为对象和数组，对象使用花括号({})、数组使用方括号([])包围。对象中各个字段以 key-value 的键值形式定义，key 与 value 之间用冒号(:)隔开，一个对象可拥有多个字段，不同字段间用逗号(,)隔开。对象字段中的 key 使用字符串数据，由双引号包围；value 可以是数值型数据(整数或浮点数，无双引号包围)，字符串数据(由双引号包围)，逻辑值数据(true 或 false)，数组数据(由方括号包围，元素间用逗号隔开)，null 以及对象数据(由花括号包围)，数组数据中的元素也可以是对象，从而使 JSON 能构成复杂的嵌套数据。需要注意的是，以上提到的符号都是在半角状态下输入的。

以本任务回传的 Web API 数据为例，有以下所述的两种数据格式。

(1) 查询全国省份(第 1 级查询)和查询某省内地级市(第 2 级查询)的数据如代码 5-18 所示。查询返回的 JSON 数据为数组数据，数组中每个元素为对象数据，对象数据具有相同的字段，其 id 字段为整型数据，name 字段为字符串数据。第 1 级查询的网址为“http://服务器 IP 地址:端口号/api/china”，假设运行 Web API 服务的 IP 地址为 10.5.14.14，则对应访问网址为“http://10.5.14.14:8080/api/china”，若端口号使用 80，则端口号可省略。第 2 级查询的网址为“http://服务器 IP 地址:端口号/api/china/{省份 id}”，本任务中，浙江省

的 id 为 17，则可使用“http://10.5.14.14:8080/api/china/17”查询浙江省的地级市列表数据。

(2) 查询某地级市内各区县(第 3 级查询)的数据如代码 5-19 所示。其结构同样为数组数据，相比第 1 级和第 2 级查询结果，其对象数据多了 1 个 weather_id 字段，值为字符串数据。第 3 级查询的网址为“http://服务器 IP 地址:端口号/api/china/{省份 id}/{地级市 id}”，例如，浙江省的 id 为 17，杭州市的 id 为 126，则可使用“http://10.5.14.14:8080/api/china/17/126”查询杭州市的区县列表数据。

注意，例中所用的 IP 地址须更改成读者运行 Web API 服务的实际 IP 地址。

代码 5-18　Web API 第 1 级和第 2 级查询返回的 JSON 数据

```
[
    {"id":1,"name":"北京"},
    …
    {"id":17,"name":"浙江"},
    …
]
```

代码 5-19　Web API 第 3 级查询返回的 JSON 数据

```
[
    {"id":999,"name":"杭州","weather_id":"CN101210101"},
    {"id":1000,"name":"萧山","weather_id":"CN101210102"},
    …
    {"id":1006,"name":"富阳","weather_id":"CN101210108"}
]
```

本任务对 JSON 返回的数据进行解析，并将解析结果封装成 City.class 类，类描述如代码 5-20 所示，类成员定义符合 Gson 解析要求。City 类中，成员变量 name 的变量名称和变量类型匹配了 JSON 对象的 name 字段，成员变量 id 则匹配了 JSON 对象的 id 字段。注意，成员变量 weatherId 与 JSON 对象的 weather_id 字段不匹配，需使用注解@SerializedName("weather_id")，使变量 weatherId 匹配由注解标注的 JSON 字段。这样设计的好处是，在设计类时可使类成员名称与 JSON 数据字段不一致，项目维护中，即使 Web API 的 JSON 对象字段发生了变化，项目解析端只需更改映射即可工作，提高了项目的可维护性。Gson 可使用所定义的 City 类解析 JSON 数据，并返回类对象，若 City 类中存在某些字段无法匹配 JSON 对象，默认会忽略，不影响解析结果。例如，用 City 类匹配代码 5-18 的 JSON 对象，类成员变量 weatherId 无匹配项，用 Gson 解析对象时，则会使用 weatherId 的默认值(null)生成 City 类对象。在本任务中，City 类没有使用 private 关键字修饰，事实上，将成员变量修改成 private 类型，并定义相关的 getter 和 setter 方法，并不影响解析结果。City 类改写了 toString()方法，使之支持字符串化，可直接用于 ArrayAdapter。

代码 5-20　JSON 解析结果的封装类 City.java

```
import com.google.gson.annotations.SerializedName;
public class City {
    public String name;
    public int id;
    @SerializedName("weather_id")
    public String weatherId;
```

```
7         //若 JSON 数据的字段"weather_id"与类成员 weatherId 不符,可用@SerializedName 映射
8         @Override
9         public String toString() {
10            return String.format(" % s, id = % d",name,id);
11        }
12        public City() {
13            name = "";
14            weatherId = "";
15        }
16        public City(String name, int id, String weatherId) {
17            this.name = name;
18            this.id = id;
19            this.weatherId = weatherId;
20        }
21    }
```

JSON 数据解析工具类 CityParsingUtils 如代码 5-21 所示,在该类中给出三个静态方法,对应了三种解析方法,用户可调用任何一种方法进行数据解析。三种方法均使用了 throws 关键字抛出异常,表示在处理数据过程中,若有异常,直接抛出,而调用者则需要使用 try-catch 捕捉异常。三者解析方法的使用和实现如下表述。

(1) json2ListByJsonObj()方法使用最原始的方式解析 JSON 数据。经分析可知,Web API 返回数据是 JSON 数组,因此使用源数据构造 JSONArray 对象,并使用 for 循环从数组中逐一取出 JSON 对象,再利用 JSON 对象的字段名称和字段类型调用对应方法进行取值。本任务第 3 级 Web API 回传的 JSON 对象有 weather_id 字段,而前两级则没有,因此,处理过程中,还需要将 JSON 对象源数据转成字符串,并判断字符串中是否包含了 weather_id 字段,再进行相应的取值处理。显然这种原生处理方法比较被动,一旦数据源或者模型发生变化,需要改动较多的代码,不利于后期维护。

(2) json2ListByGson()方法使用 Gson 解析数据(Gson 需要在 Gradle 中添加依赖)。在该方法中,首先使用源数据构造 JSONArray 对象,在遍历中取出 JSON 对象,再转换为字符串作为 Gson 源数据。Gson 可直接将源数据转换为预先定义的数据类并返回类对象,当源数据与类字段存在未匹配项或缺项,并不影响转换,一旦数据结构发生变动,只需要修改 City 类使之与新的数据匹配,大大提高了可维护性。

(3) json2ListByGsonList()方法使用了 TypeToken 反射,可直接将源数据转换为列表类对象,使得代码更简洁,也更适合复杂嵌套的结构化数据的解析。

代码 5-21　自定义 JSON 解析工具类 CityParsingUtils.java

```
1   import android.text.TextUtils;
2   import com.google.gson.Gson;
3   import com.google.gson.reflect.TypeToken;
4   import org.json.JSONArray;
5   import org.json.JSONException;
6   import org.json.JSONObject;
7   import java.lang.reflect.Type;
8   import java.util.ArrayList;
9   import java.util.List;
10  public class CityParsingUtils {
11      /** 需要在 Module Gradle 文件的 dependencies 中增加 Gson
12       implementation 'com.google.code.gson:gson:2.9.0'
```

```
13       * /
14      public static List < City > json2ListByJsonObj(String s) throws JSONException {
15          //若遇到错误,抛出异常,由调用者使用 try - catch 捕捉异常
16          //直接采用 JsonObject 解析
17          List < City > list = new ArrayList <>();
18          if(!TextUtils.isEmpty(s)){
19              JSONArray jsonArray = new JSONArray(s);
20              for (int i = 0; i < jsonArray.length(); i++) {
21                  JSONObject jsonObject = jsonArray.getJSONObject(i);
22                  int id = jsonObject.getInt("id");
23                  String name = jsonObject.getString("name");
24                  String weatherId = "";
25                  if(jsonObject.toString().toLowerCase().contains("weather_id")){
26                      //JSON 数据有些包含 weather_id 字段,有些不包含,处理比较被动
27                      //需要将 JSON 对象转换成 string,对字段进行包含判断
28                      weatherId = jsonObject.getString("weather_id");
29                  }
30                  City city = new City(name, id, weatherId);
31                  list.add(city);
32              }
33          }
34          return list;
35      }
36      public static List < City > json2ListByGson(String s) throws JSONException {
37          List < City > list = new ArrayList <>();
38          if(!TextUtils.isEmpty(s)){
39              JSONArray jsonArray = new JSONArray(s);
40              for (int i = 0; i < jsonArray.length(); i++) {
41                  //从 JSON 数组中遍历 JSON 对象,对 JSON 对象利用 Gson 转换成对应类
42                  String s1 = jsonArray.get(i).toString();
43                  //JSON 对象再转换成 JSON 字符串供 Gson 转换
44                  City city = new Gson().fromJson(s1, City.class);
45                  //若 City.class 中定义的某些字段和 JSON 字段没有匹配,只取能匹配的字段
46                  list.add(city);
47              }
48          }
49          return list;
50      }
51      public static List < City > json2ListByGsonList(String s) throws Exception {
52          //定义方法时,增加 throws Exception,以便于调用时捕捉异常
53          List < City > list = new ArrayList <>();
54          if(!TextUtils.isEmpty(s)){
55              Type type = new TypeToken < List < City >>(){}.getType();
56          //利用 com.google.gson.reflect.TypeToken 反射构造 List < City >列表对象类数据
57              list = new Gson().fromJson(s,type); //直接利用 TypeToken 转换为列表对象
58          }
59          return list;
60       }
61  }
```

4. 使用 Okhttp 获取 Internet 数据

Okhttp 是一套处理 Http 网络请求的依赖库,由 Square 公司设计研发并开源,目前可以在 Java 和 Kotlin 中使用。Okhttp 支持 Web 常用的 GET 和 POST 操作,并且有拦截器和鉴权功能。当 Web API 需要用户登录鉴权时,Okhttp 可利用 authentication()方法检测是否鉴权失效和自动登录鉴权,在前后端分离的应用场景,Okhttp 作为前端角色,负责与后

端进行数据交互。在本任务中,只使用了 Okhttp 的 GET 操作,并介绍了异步和同步两种操作方法。

使用 Okhttp 前,需要在 Gradle 中添加对应依赖。为了方便调用者使用,将 Okhttp 对 Web 的请求封装到一个自定义工具类 WebApiUtils 中,如代码 5-22 所示。Okhttp 的使用需要通过 new OkHttpClient()构造方法得到一个 OkHttpClient 客户端对象。考虑到调用者访问 Web,多次网络请求均会使用同一个客户端对象,因此在工具类中将 OkHttpClient 对象设计成静态成员变量,使之成为单例模式,即不管外部调用者调用多少次 WebApiUtils,均由同一个 OkHttpClient 客户端提供服务。当 OkHttpClient 对象为 null 时,为了避免有多个线程同时调用 getClient()方法生成对象,导致 OkHttpClient 对象不一致,在 getClient()方法中使用了 synchronized 修饰,并且锁的对象是工具类本身,从而保证了 getClient()方法的原子性,即在同一时刻只能有一个线程能访问该方法。在 WebApiUtils 类中,getClient()方法是 private 类型,外部类不能直接调用,只能通过 WebApiUtils 提供的其他 public 方法间接调用。

异步调用需要将获取的 Web 数据回传,可通过 Handler、自定义接口或者 LiveData 等方式实现,本任务中,采用自定义接口 OnReadFinishedListener 实现,其中 onFinished()方法传递解析后的 City 列表数据,onFail()方法则传递错误信息字符串,接口的两个方法由调用者实现。

Okhttp 支持异步调用和同步调用。异步调用时,Okhttp 会自动启动一个后台线程去访问 Web 资源,调用者的调用代码尚未执行结束就会继续往下执行其他代码,并通过回调事件告知调用者出错或者完成情况。Okhttp 后台线程处理过程中,若遇到异常,则会回调 onFailure()方法,在该方法中可接入自定义接口 OnReadFinishedListener 的 onFail()回调方法。考虑到调用者是在 UI 线程中实现自定义接口的 onFail()回调方法,并且涉及 UI 修改,因此,在 Okhttp 的 onFailure()方法中处理 onFail()回调时,需要切换到 UI 线程。鉴于此,异步调用方法需要传递 Activity 对象,利用 Activity 对象切换 UI 线程。

如代码 5-22 所示,getApiDataAsync()方法为自定义的 Okhttp 异步调用方法。在该方法中,首先通过 getClient()方法获得 OkHttpClient 对象,并根据网址传参 url 生成 Request 请求对象,在生成过程中使用 get()方法表明 Web 访问采用 GET 方式,最后调用 OkHttpClient 对象的 newCall()方法,将 Request 对象作为方法的参数,得到 Call 对象。Call 对象可理解为 OkHttpClient 对 Request 请求产生的调用对象,支持同步调用和异步调用,同步调用时,代码进入阻塞,执行完毕才会继续执行后续代码;异步调用则不等待调用结果,可直接执行后续代码,通过回调得到执行结果或者出错信息。

getApiDataAsync()方法通过 Call 对象的 enqueue()方法实现异步调用功能,在 enqueue()方法中需要实现 Callback 接口的两个回调:onFailure()方法,在程序异常时触发;onResponse()方法,在成功获取网络数据时触发。两个回调方法都在 OkHttpClient 自身管理的后台线程中执行,无法直接更新 UI,若用户在回调中涉及 UI 更新,则需要切换到 UI 线程中进行处理。getApiDataAsync()方法给出的解决方案是通过 Activity 对象的 runOnUiThread()方法进行线程切换,在 UI 线程中调用自定义接口的方法,其中 onFailure()回调负责调用自定义接口的 onFail()方法,将错误信息转换成字符串传递给外部调用者进行处理,onResponse()回调则获取网络响应的文本数据,并调用 JSON 数据解析工具类的相关方法将其转换成 City

类型的列表数据，进而切换到主 UI 线程通过自定义接口的 onFinished()方法将列表数据传递给调用者更新 UI。在 onResponse()回调中，原生方法有 throws 语句用于抛出异常，本任务中，删除了 throws 语句，直接在回调中通过 try-catch 处理异常，并且将 catch 中捕捉的异常通过自定义接口的 onFail()方法传递给调用者处理。该实现方式的好处是，调用者只需要实现自定义接口的 onFinished()方法和 onFail()方法，进行两类问题的处理即可，无须再额外处理各类异常抛出的问题。

同步调用使用 getApiDataBlock()方法，不同于 getApiDataAsync()方法，它阻塞网络请求代码并在所有代码执行结束后才返回结果，因此该方法定义了返回类型 List < City >。而异步调用是在接口回调中返回结果，使异步方法是 void 类型，无返回结果。同步和异步方法均声明成 static 静态方法，外部调用者可直接使用 WebApiUtils 类对应的方法，无须生成 WebApiUtils 对象。在 getApiDataBlock()方法中还使用了 throws Exception 语句抛出异常，使调用者能通过 try-catch 捕捉异常并处理异常。getApiDataBlock()方法中，如何生成 OkHttpClient 和 Request 对象，以及将两者关联成 Call 对象，与 getApiDataAsync()方法处理方式相同，区别在于 Call 对象的调用方式，异步调用使用 enqueue()方法加入队列，同步方法则直接调用 execute()方法执行请求并返回结果。CityParsingUtils 提供了 3 种解析 JSON 数据的方法，在同步调用中选择了不同于异步调用的解析方法，用于验证各种解析方法的有效性，并无特别用意，读者可任意更换解析方法。

WebApiUtils 工具类提供了异步调用和同步调用两种方式，通过 OkHttp 访问网址，得到结果，并解析成对应的列表数据。同步调用看似比异步调用使用了更少的代码，但是同步调用在 UI 线程中并不能直接使用，其原因是同步调用 Okhttp 的所有处理均会在调用者所在的线程中执行，而 UI 线程并不能直接执行访问网络等耗时操作。因此，在 UI 线程中，若要使用同步调用，依然需要生成一个后台线程去调用同步方法，而异步调用的 enqueue()方法会使 OkHttp 自动开启后台线程处理相关事务，对调用者而言，异步调用无须额外的后台线程。由以上分析可知，在 UI 线程中更适合调用 WebApiUtils 工具类的异步方法，而同步方法适用于后台线程或者 Service 服务中对网络资源进行遍历访问等无 UI 交互的场合。

代码 5-22　自定义 Okhttp 调用工具类 WebApiUtils. java

```
1    import android.app.Activity;
2    import androidx.annotation.NonNull;
3    import java.io.IOException;
4    import java.util.List;
5    import okhttp3.Call;
6    import okhttp3.Callback;
7    import okhttp3.OkHttpClient;
8    import okhttp3.Request;
9    import okhttp3.Response;
10   public class WebApiUtils {
11       /** 需要在 Module Gradle 文件的 dependencies 标签中增加 Okhttp 组件
12        implementation 'com.squareup.okhttp3:okhttp:4.9.3'
13        */
14       private static OkHttpClient client;
15       //client 在 WebApiUtils 中是单例模式,始终只有 1 个对象
16       private static OkHttpClient getClient() {
17           synchronized (WebApiUtils.class) {
18               //加锁避免多个线程同时调用 getClient(),在 client == null 时重复生成实例
```

```
            //对 WebApiUtils.class 加锁,多个线程只能有 1 个线程能访问此代码
            if (client == null) {
            client = new OkHttpClient();
          }
          return client;
        }
      }
      public interface OnReadFinishedListener{
        //自定义接口,定义两个回调,分别用于读取成功和出错处理
        public void onFinished(List<City> readOutList);
        //读取成功,将 Json 数据转换为类列表数据返回
        public void onFail(String e);
        //读取失败,回传错误信息
      }
      public static void getApiDataAsync(Activity activity,String url,
                                          OnReadFinishedListener l){
        //传入 Activity 对象,用于切换线程
        OkHttpClient c = getClient();     //通过调用 getClient()得到 OkHttpClient 单例
        Request request = new Request.Builder().url(url).get().build();
        //构造一个请求对象 Request,url()传 url 网址,get()是 GET 请求的方法
        //请求方法有 get()、post()、delete()、put()等方法
        Call call = client.newCall(request);           //利用 Request 生成一个 call 对象
        //每一次访问对应一个 call 对象,异步访问时将 call 对象加入队列
        call.enqueue(new Callback() {
          @Override
          public void onFailure(@NonNull Call call, @NonNull IOException e) {
            activity.runOnUiThread(new Runnable() {   //切换到 UI 线程
                @Override
                public void run() {
                    l.onFail(e.toString());
                    //通过自定义接口将错误信息通过 onFail()传递到 UI 线程
                }
            });
          }
          @Override
          public void onResponse(@NonNull Call call,
                                 @NonNull Response response){
            //修改原回调方法,去除 throws 异常语句,直接捕捉处理
            try {
                String s = response.body().string();         //得到响应的文本
                //注意是 string()方法,不是 toString()方法
                List<City> list = CityParsingUtils.json2ListByGsonList(s);
                //CityParsingUtils 提供了 3 种解析方法,可调用任何一种
                activity.runOnUiThread(new Runnable() {       //切换到 UI 线程
                    @Override
                    public void run() {
                        l.onFinished(list);
                    }
                });
            } catch (Exception e) {
                e.printStackTrace();
                activity.runOnUiThread(new Runnable() {       //切换到 UI 线程
                    @Override
                    public void run() {
                        l.onFail(e.toString());   //错误信息通过接口回调传给调用者
                    }
                });
```

```
            }
        }
    });
}
public static List<City> getApiDataBlock(String url) throws Exception{
    //在运行过程中遇到异常,将抛出异常给调用者处理
    //采用同步的方式,代码将阻塞,适合其他后台线程循环调用此方法获取批量请求结果
    OkHttpClient client = getClient();
    Request request = new Request.Builder().url(url).get().build();
    Call call = client.newCall(request);
    //生成 Request 和 Call 对象,与异步调用相同,区别的是 call 的处理方式
    Response response = call.execute();
    //同步方法,运行该代码将进入阻塞,直至 call 执行完毕
    String s = response.body().string();
    List<City> list = CityParsingUtils.json2ListByGson(s);
    //调用 CityParsingUtils 第二种解析方法,可尝试更换成其他方法
    return list;
}
}
```

5. 实现 MainActivity

MainActivity 的布局文件如代码 5-23 所示,内容比较简单,不再赘述。

代码 5-23　MainActivity 的布局文件 my_main.xml

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="match_parent"
    android:layout_height="match_parent">
    <TextView
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="Your name and ID" />
    <EditText
        android:id="@+id/et_url"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:ems="10"
        android:inputType="textPersonName"
        android:text="http://guolin.tech/api/china/" />
    <Button
        android:id="@+id/bt_async"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="Async mode" />
    <Button
        android:id="@+id/bt_block"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="Block mode" />
    <ListView
        android:id="@+id/listView"
        android:layout_width="match_parent"
        android:layout_height="match_parent" />
</LinearLayout>
```

MainActivity 的实现如代码 5-24 所示。应用在调用 WebApiUtils 类所提供的方法之

前,需要在 AndroidManifest.xml 文件中配置 Internet 权限,并且对于 Android 10.0 及以上版本的系统,访问 Http 资源需要在 application 标签内增加 android:usesCleartextTraffic 属性。在 MainActivity 页面中,用户可通过 EditText 更改网址参数,获取不同的城市列表。

MainActivity 布局文件中有 2 个 Button,分别用于调用 WebApiUtils 的异步方法和同步阻塞方法。在异步方法中,对 WebApiUtils.OnReadFinishedListener 接口匿名实现,分别处理 onFinished()回调和 onFail()回调。onFinished()回调方法回传城市列表数据,并通过 showList()方法生成适配器,进而在 ListView 中显示网络请求的回传数据;onFail()方法则将回传的出错信息使用 Toast 显示,告知用户程序异常的具体信息。同步方法不能直接在 UI 线程中调用,需要开启后台线程,并在子线程中调用。本任务中,直接匿名实现后台线程,在后台线程中获得同步方法的返回结果后,再利用 MainActivity 对象切换到 UI 线程,在 UI 线程中调用 showList()方法更新 ListView。WebApiUtils.getApiDataBlock()方法使用 throws 语句抛出异常,被调用时需要使用 try-catch 捕捉,并在 catch 中处理错误信息。注意,同步方法是在后台线程中调用的,因此,打印错误信息也需要切换到 UI 线程中执行。由此可见,对调用者而言,在 UI 线程中,使用异步调用比同步调用更简洁易用。

代码 5-24　MainActivity.java

```
import androidx.appcompat.app.AppCompatActivity;
import android.os.Bundle;
import android.view.View;
import android.widget.ArrayAdapter;
import android.widget.EditText;
import android.widget.ListView;
import android.widget.Toast;
import java.util.List;
public class MainActivity extends AppCompatActivity {
    /** manifests/AndroidManifest.xml 中需要增加上网权限相关配置
     * 在 application 标签之后增加 Internet 访问权限
     Android10.0+系统,需要在 AndroidManifest 的 application 标签内增加以下属性:
     android:usesCleartextTraffic="true"
     */
    ListView lv;
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.my_main);
        EditText et=findViewById(R.id.et_url);
        lv=findViewById(R.id.listView);
        findViewById(R.id.bt_async)
                .setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View view) {
                String url = et.getText().toString().trim();
                //trim()方法将字符串头尾多余空格去除
                //以下是异步调用的方法
                WebApiUtils.getApiDataAsync(MainActivity.this, url,
                        new WebApiUtils.OnReadFinishedListener() {
                    @Override
                    public void onFinished(List<City> readOutList) {
                        showList(readOutList);
                    }
```

```
35                  @Override
36                  public void onFail(String e) {
37                      showToast(e);
38                  }
39              });
40          }
41      });
42      findViewById(R.id.bt_block)
43              .setOnClickListener(new View.OnClickListener() {
44          @Override
45          public void onClick(View view) {
46              String url = et.getText().toString().trim();
47              //同步调用,在主UI中无法直接调用,需要启动一个线程来调用
48              new Thread(new Runnable() {
49                  @Override
50                  public void run() {
51                      try {
52                          List<City> list = WebApiUtils.getApiDataBlock(url);
53                          MainActivity.this.runOnUiThread(new Runnable() {
54                              @Override
55                              public void run() {
56                                  showList(list);
57                              }
58                          });
59                      } catch (Exception exception) {
60                          exception.printStackTrace();
61                          MainActivity.this.runOnUiThread(new Runnable() {
62                              @Override
63                              public void run() {
64                                  showToast(exception.toString());
65                              }
66                          });
67                       }
68                  }
69              }).start();        //直接启动所定义的线程
70          }
71      });
72  }
73  private void showList(List<City> readOutList) {
74      //将列表数据生成适配器,将适配器显示到ListView组件上
75      ArrayAdapter<City> adapter = new ArrayAdapter<>(this,
76              android.R.layout.simple_list_item_1,readOutList);
77      lv.setAdapter(adapter);
78  }
79  private void showToast(String e) {
80      Toast.makeText(this,e,Toast.LENGTH_LONG).show();
81  }
82  }
```

5.5　Activity 的页面跳转与数据传递

5.5.1　任务说明

本任务在 5.4 节基础上完成，具体效果如图 5-7 所示，应用通过 Activity 之间的跳转实

现分级城市列表的显示。应用由两个 Activity 页面构成,分别为 MainActivity 和 MainActivity2。应用的默认主页面为 MainActivity,通过访问 Web API 获取数据,在 ListView 中显示浙江省的地级市列表,单击列表项,获得所单击城市的 id,生成第 3 级城市列表网址,传递网址并跳转到 MainActivity2。MainActivity2 显示区县列表,单击列表项则结束当前页面,并将所单击的列表项数据传回给 MainActivity。MainActivity 接收到 MainActivity2 所传递的数据后,使用 SnackBar 显示 MainActivity2 回传的数据。此外,MainActivity2 可通过动作栏左上角的返回键返回到上一级的 Activity 页面,即 MainActivity 页面。

图 5-7　通过 Activity 跳转实现分级城市列表显示

5.5.2　任务实现

1. Activity 类在应用中的声明

根据模板创建的 Android 应用,默认只有 1 个 Activity 类,并且已在 AndroidManifest.xml 文件中设置了相关声明。若要在应用中跳转到同个项目的其他 Activity,则须在 AndroidManifest 中做相应的声明,否则由于权限原因无法跳转到对应的 Activity。本任务共创建了两个 Activity,分别为 MainActivity 和 MainActivity2,因此在 AndroidManifest 中须添加对应的声明,如代码 5-25 所示,每一个活动页面(Activity)使用一个 activity 标签,其 android:name 属性值为 Activity 对应的类名称。

代码 5-25　关于 Activity 的声明文件 AndroidManifest.xml

```
<activity
    android:name=".MainActivity"
    android:exported="true"
    android:launchMode="singleTask">
    <intent-filter>
        <action android:name="android.intent.action.MAIN" />
        <category android:name="android.intent.category.LAUNCHER" />
    </intent-filter>
```

```
    </activity>
    <activity
        android:name = ".MainActivity2"
        android:launchMode = "singleTask"
        android:parentActivityName = ".MainActivity">
    </activity>
```

2. Activity 类的启动模式

一个 Android 应用通常会被拆分成多个 Activity，各个 Activity 之间通过 Intent 实现跳转。在 Android 系统中，通过栈结构来保存整个应用的 Activity，当一个应用启动时，如果当前环境中不存在该应用的任务栈，系统就会创建一个任务栈。此后，这个应用所启动的 Activity 都将在这个任务栈中被管理，这个栈也被称为一个 Task，即若干个 Activity 的集合，他们组合在一起形成一个 Task。

在标准模式下，当一个 Activity 启动了另一个 Activity 的时候，新启动的 Activity 就会置于任务栈的顶部，而启动它的 Activity 则处于停止状态，保留在任务栈中。当用户按下返回键或者调用 finish()方法时，系统会移除 Task 顶部的 Activity，让后面的 Activity 恢复活动状态(回调 onResume()方法)。当然 Activity 也可以有不同的启动模式，可在声明文件 AndroidMainifest 中，对 activity 标签添加 android:launchMode 属性来设置启动模式，或者在代码中通过 Intent 对象的 flag 属性来设置启动模式。

在项目的声明文件中，activity 标签内的额外属性 android:launchMode 常用有如下四种选项，分别对应一种启动模型。

(1) 标准模式—standard。标准模式是默认模式，即没有添加 android:launchMode 属性时采用的模式。在该模式下，每次启动 Activity，均会在任务栈中创建新的 Activity 实例，因此，多次启动 Activity 后，任务栈中会有多个相同类名的 Activity 实例，并且各个 Activity 实例与调用者在同一个任务栈中。

(2) 栈顶单例—singleTop。在该模式下，若任务栈中没有被启动的 Activity，则创建 Activity 实例，并置于栈顶；若任务栈中已有 Activity 实例，并且在栈顶，则不会创建对应实例(standard 模式会继续创建 Activity 实例)，此时不会回调 onCreate()方法，但会调用 Activity 的 onNewIntent()方法；若任务栈中已有 Activity 实例，但不在栈顶，则会重新创建 Activity 实例于栈顶(与 standard 模式相同)。栈顶单例所创建的 Activity 实例与调用者在同一个任务栈中。

(3) 栈内单例—singleTask。在该模式下，若任务栈中没有被启动的 Activity，则创建 Activity 实例，并置于栈顶；若任务栈中已有 Activity 实例，并且在栈顶，则不会创建该实例，不会回调 onCreate()方法，但会调用 Activity 的 onNewIntent()方法；若任务栈中已有该 Activity 实例，但不在栈顶，则会将任务栈中该 Activity 之上的活动全部销毁，使被启动的 Activity 能处于栈顶，此时不会回调 onCreate()方法，但会回调 onNewIntent()方法。栈内单例模式，所创建的 Activity 与调用者在同一个任务栈中。

(4) 全局单例—singleInstance。在该模式下，若没有对应 Activity 实例，会重新创建 1 个新的任务栈，并在新的任务栈中创建 Activity 实例使之处于栈顶；若已有该 Activity 实例，则切换任务栈，将该 Activity 置于任务栈前台，使该 Activity 实例可见，此时 Activity 不会回调 onCreate()方法，但会回调 onNewIntent()方法。全局单例模式，所创建的 Activity

与调用者不在同一个任务栈中。

为了更形象地说明这4种启动模式的区别,设计4个Activity和对应的启动模式,A:standard;B:singleTop;C:singleTask;D:singleInstance。这4个Activity都能按指定的方式启动各个Activity。为了便于观察,每个Activity设计1个静态计数器,初始值为0,Activity在onCreate()回调方法中会将计数器自增1,在onNewIntent()回调方法中则不改变计数器值,使观测到的现象是:重建的Activity,其计数器自增1,复用原有Activity时则计数器值不变。

(1) 操作1:A→A→B→B→A→A→B→B,其中箭头方向表示Activity的启动顺序。此时共有1个任务栈,栈中的活动页面过程状态为:A(1)→A(2)→B(1)→B(1)→A(3)→A(4)→B(2)→B(2),任务栈最后的状态为:A(1)→A(2)→B(1)→A(3)→A(4)→B(2),共创建了6个Activity,其中括号内的值表示对应Activity的计数器值。若单击应用的返回键,则会按B(2)→A(4)→A(3)→B(1)→A(2)→A(1)的顺序依次关闭Activity。活动B不在栈顶时会被重新创建,在栈顶时则会被复用,活动A每次均被重新创建。

(2) 操作2:A→A→C→C→A→A→C→C。此时共有1个任务栈,栈中的活动页面过程状态为:A(1)→A(2)→C(1)→C(1)→A(3)→A(4)→C(1)→C(1),活动栈的最终状态为:A(1)→A(2)→C(1),共创建了5个Activity,并在第7个操作时,由于C是singleTask模式,在任务栈中找到实例C(1),此时C(1)之上有A(3)和A(4),会将其销毁,使C(1)处于任务栈顶。因此,任务栈最终只保留了3个活动:A(1)、A(2)和C(1)。若单击应用的返回键,则会按C(1)→A(2)→A(1)的顺序依次关闭Activity。操作2与操作1的最大区别是活动C被创建后,每次启动C时会将C之上的活动销毁,并且C在任务栈中是唯一的,不会被重复创建。

(3) 操作3:A→A→D→D→A→A→D→D。此时共有两个任务栈,所有活动A在同一个任务栈中,活动D则在另一个任务栈中保持单例。两个任务栈中的过程状态为:A(1)→A(2)→D(1)→D(1)→A(3)→A(4)→D(1)→D(1),任务栈1的最终状态为:A(1)→A(2)→A(3)→A(4),任务栈2的最终状态为:D(1),并且任务栈2在任务栈1之上。若单击应用的返回键,则会关闭任务栈2的D(1),此时任务栈2退出,将任务栈1置于前台,继续按返回键,会按A(4)→A(3)→A(2)→A(1)的顺序依次关闭Activity。由于活动A和活动D处在两个不同的任务栈中,当用户通过Home键或者其他方式进行任务切换后,并不能保证任务栈2在任务栈1之上,此时单击活动D的应用返回键,可能存在结束任务栈2时无法回到任务栈1的情况。

综上所述,本节任务的应用须使MainActivity和MainActivity2保持在同一个任务栈中,使MainActivity2的返回键能有效回退到MainActivity,此时应将声明文件中的所有Activity启动模式设置成栈内单例(singleTask)模式,使各Activity在同一个任务栈中保持单例。

3. Gradle依赖和权限

在Module Gradle文件dependencies节点中添加Okhttp和Gson的依赖。

```
implementation 'com.squareup.okhttp3:okhttp:4.9.3'
implementation 'com.google.code.gson:gson:2.9.0'
```

在声明文件 AndroidManifests.xml 中添加上网权限。

```
<uses-permission android:name="android.permission.INTERNET"></uses-permission>
```

对于 Android 10.0 及以上系统，在 AndroidManifests.xml 的 application 标签中添加 Http 明码访问模式。

```
android:usesCleartextTraffic="true"
```

本任务会复用 5.4 节项目中写好的封装类，将 5.4 节项目的 City.java、CityParsingUtils.java 和 WebApiUtils.java 复制到本项目 MainActivity.java 所在的文件夹中。

4. 实现第 1 个活动 MainActivity

本任务中，2 个 Activity 使用同一个布局文件 my_main.xml，如代码 5-26 所示，布局中仅有 1 个 TextView 和 1 个 ListView。

代码 5-26　活动页面的布局文件 my_main.xml

```
1   <LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
2         android:orientation="vertical"
3         android:layout_width="match_parent"
4         android:layout_height="match_parent">
5         <TextView
6               android:layout_width="match_parent"
7               android:layout_height="wrap_content"
8               android:text="Your name and ID" />
9         <ListView
10              android:id="@+id/listView"
11              android:layout_width="match_parent"
12              android:layout_height="match_parent" />
13  </LinearLayout>
```

MainActivity 的实现如代码 5-27 所示。在 MainActivity 中，变量 baseUrl 为获取浙江省城市列表的 Web API 网址，读者需要根据实际运行设备的 IP 进行更改。在 ListView 单击事件 onItemClick()方法中，取出被单击城市的 id，与变量 baseUrl 拼接成一个完整的资源网址 tempUrl，启动 MainActivity2 时，作为 Activity 之间的传递数据。Activity 之间传递数据，建议使用 Bundle，既能传递自定义类，也能传递常用的变量。

本任务创建一个自定义类 CommonValues，如代码 5-28 所示，专门用于定义应用中需要的常量。Bundle 的 key 可从 CommonValues 类中获得。

启动另一个 Activity 可使用 Intent(意图)实现。Intent 的构造方法有很多种，有动作＋数据资源的方式，让系统选择对应的应用启动；也有调用者＋类的方式，指定启动哪个类。本任务使用调用者＋类的方式定义 Intent，在构造方法中，第 1 个参数是调用者上下文，可用 MainActivity.this 明确指定，也可以通过 getApplicationContext()方法获得应用上下文，第 2 个参数即为需要启动的活动类名称。Intent 携带的数据可通过 putExtras()方法将 Bundle 对象传入，被启动的活动则可通过 getIntent().getExtras()方法取得 Bundle 对象，进而可通过 Bundle 解析所需数据。

创建 Intent 对象后，有两种方式启动 Activity，具体表述如下。

(1) 不带返回结果启动。通过 startActivity(Intent intent)方法调用，参数即为意图对

象 intent。假设活动 A 通过该方式启动了活动 B,则活动 B 不负责将处理结果返回给活动 A。

(2) 带返回结果启动。早期的 SDK 中通过 startActivityForResult(Intent intent,int requestCode)方法调用,除了 Intent 对象,还需要请求码 requestCode,请求码的值可由用户自定义。当前,用于启动 Intent 的 startActivityForResult()方法已被弃用,推荐使用 registerForActivityResult()方法生成 ActivityResultLauncher 对象用于启动 Intent。registerForActivityResult()方法直接在参数中设置回调接口,可匿名实现接口的回调方法,其好处是不需要用户定义请求码,并且该方法功能更强大,对 Kotlin 编程更简洁友好。

ActivityResultLauncher 作为带返回结果的 Intent 启动器,在 onCreate()方法中注册,然后在需要调用的地方调用 launch()方法启动 Intent,ActivityResultLauncher 对象在代码 5-27 中被定义为成员变量,并通过 iniActivityLauncher()方法初始化注册。ActivityResultLauncher 的构造方法中传入两个参数,第 1 个参数是 Contract(合约)对象,有多种类型;第 2 个参数是结果回调,可在回调方法中处理返回的数据。

常用的 Contract 如下所列。

(1) StartActivityForResult(),最常用的 Contract 合约,启动带返回结果的 Intent。

(2) RequestMultiplePermissions(),用于请求一组运行时权限。

(3) RequestPermission(),用于请求单个运行时权限。

(4) TakePicturePreview(),调用 MediaStore. ACTION_IMAGE_CAPTURE 拍照,返回值为 Bitmap 图片。

(5) TakePicture(),调用 MediaStore. ACTION_IMAGE_CAPTURE 拍照,并将图片保存到给定的 Uri 地址,返回 true 表示保存成功。

(6) TakeVideo(),调用 MediaStore. ACTION_VIDEO_CAPTURE 拍摄视频,保存到给定的 Uri 地址,返回一张缩略图。

(7) PickContact(),从系统通讯录应用中获取联系人。

(8) 文档内容操作合约,常见的有: CreateDocument(),OpenDocumentTree(),OpenMultipleDocuments(),OpenDocument(),GetMultipleContents(),GetContent()等。

在 MainActivity 中,ActivityResultLauncher 采用 StartActivityForResult()方法生成的合约,用于启动 MainActivity2,并处理 MainActivity2 的返回结果。ActivityResultLauncher 对象初始化时,第 2 个参数直接处理 MainActivity2 的返回结果,Bundle 对象携带的是 City 对象,City 类需要实现序列化接口(Serializable),才能在 Bundle 中通过 putSerializable()方法存入数据以及 getSerializable()方法取出数据。对 City 类的序列化,只需对类增加 implements Serializable 修饰即可实现,其他保持不变。

City 类的序列化实现如下。

```
1    import java.io.Serializable;
2    public class City implements Serializable {
3    …//类相关成员和方法保持不变
4    }
```

SnackBar 是高级版的 Toast,其使用方式与 Toast 类似,但是额外增加了按钮功能。SnackBar 构造方法的第 1 个参数是 View 对象,表示 SnackBar 在哪个视图上生成,即依赖

在哪个视图上，而 Toast 的第 1 个参数则是上下文。SnackBar 的按钮可通过 setAction()方法设置，该方法中第 1 个参数是按钮上显示的文本，第 2 个参数是按钮单击响应回调。

代码 5-27　活动页面 1——MainActivity.java

```
import android.content.Intent;
import android.os.Bundle;
import android.util.Log;
import android.view.View;
import android.widget.AdapterView;
import android.widget.ArrayAdapter;
import android.widget.ListView;
import android.widget.Toast;
import androidx.activity.result.ActivityResult;
import androidx.activity.result.ActivityResultCallback;
import androidx.activity.result.ActivityResultLauncher;
import androidx.activity.result.contract.ActivityResultContracts;
import androidx.appcompat.app.AppCompatActivity;
import com.google.android.material.snackbar.Snackbar;
import java.util.List;
public class MainActivity extends AppCompatActivity {
    String baseUrl = "http://10.5.14.14:8080/api/china/17";
    //baseUrl 须更换成运行 Web API 的实际设备地址
    ListView lv;
    ArrayAdapter<City> adapter;
    ActivityResultLauncher<Intent> launcher;//定义能回调返回结果的意图启动器
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.my_main);
        lv = findViewById(R.id.listView);
        WebApiUtils.getApiDataAsync(this, baseUrl,
                new WebApiUtils.OnReadFinishedListener() {
                    @Override
                    public void onFinished(List<City> readOutList) {
                        showList(readOutList);
                    }
                    @Override
                    public void onFail(String e) {
                        showToast(e);
                    }
                });
        //ActivityResultLauncher 取代 startActivityForResult()
        iniActivityLauncher();                //对 ActivityResultLauncher 初始化
        lv.setOnItemClickListener(new AdapterView.OnItemClickListener() {
            @Override
            public void onItemClick(AdapterView<?> adapterView, View view,
                                    int i, long l) {
                City item = adapter.getItem(i);
                String tempUrl = String.format("%s/%d", baseUrl, item.id);
                //打印新的网址，获取城市对应县城或行政区列表
                Bundle bundle = new Bundle();
                bundle.putString(CommonValues.KEY_URL,tempUrl);
                //利用 Bundle 封装需要传递的数据
                Intent intent = new Intent(getApplicationContext(),
                        MainActivity2.class);
```

```
52                  intent.putExtras(bundle);     //给待启动的 Intent 携带 Bundle 数据
53                  //registerForActivityResult()是推荐的带返回结果的意图启动方法
54                  //使用 ActivityResultLauncher 对象的 launch()方法启动意图
55                  launcher.launch(intent);
56              }
57          });
58      }
59      private void iniActivityLauncher() {
60          launcher = registerForActivityResult(
61                  new ActivityResultContracts.StartActivityForResult(),
62                  new ActivityResultCallback<ActivityResult>() {
63                      @Override
64                      public void onActivityResult(ActivityResult result) {
65                          Intent data = result.getData();
66                          //获得意图对象 data
67                          int resultCode = result.getResultCode();
68                          //获得结果码 resultCode
69                          if (resultCode == RESULT_OK) {
70                              //判断结果码是否为 RESULT_OK
71                              Bundle b = data.getExtras();     //获得 Intent 对象的 Bundle
72                              City city = (City) b.getSerializable(
73                                                  CommonValues.KEY_CITY);
74                              //从 Bundle 对象中取得自定义的 City 数据
75                              showSnackBar(city);     //调用 SnackBar 显示 city 信息
76                          }
77                  }
78              });
79      }
80      @Override
81      protected void onNewIntent(Intent intent) {
82          //解决跳转到 MainActivity 没有回调 onCreate()方法的问题
83          super.onNewIntent(intent);
84          Log.d("onNewIntent",this.getLocalClassName());  //打印日志观察回调
85          //this.getLocalClassName()获取当前类的名称
86      }
87      private void showList(List<City> readOutList) {
88          //将列表数据生成适配器,显示到 ListView 对象上
89          adapter = new ArrayAdapter<>(this,
90                  android.R.layout.simple_list_item_1,readOutList);
91          lv.setAdapter(adapter);
92      }
93      private void showToast(String e) {
94          Toast.makeText(this,e,Toast.LENGTH_LONG).show();
95      }
96      private void showSnackBar(City city) {
97          Snackbar.make(lv, "City = " + city.name, Snackbar.LENGTH_LONG)
98                  .setAction("WeatherId", new View.OnClickListener() {
99                      @Override
100                     public void onClick(View view) {
101                         showToast("Weather_id = " + city.weatherId);
102                     }
103                 }).show();
104         //使用 Snackbar 显示内容,Snackbar 至多可设置 1 个 Action(按钮)
105     }
106 }
```

代码 5-28　常量定义类 CommonValues.java

```
public class CommonValues {
    public static final String KEY_URL = "key_url";
    public static final String KEY_CITY = "key_city";
}
```

5. 实现第 2 个活动 MainActivity2

MainActivity2 的实现如代码 5-29 所示，其主要功能是根据 MainActivity 传过来的第 3 级城市列表网址，获取数据并显示在 ListView 上，当用户单击 ListView 时，将单击数据回传给 MainActivity。在 updateWebData()方法中，MainActivity2 通过 getIntent().getExtras()方法获得 MainActivity 所传递的 Bundle 对象，并从 Bundle 中解析出携带的网址，调用 WebApiUtils.getApiDataAsync()异步方法获取 Web API 数据用于更新 ListView。在 ListView 的列表项单击事件处理中，获取对应城市数据，并通过 putSerializable()方法将 City 数据放入 Bundle 对象。Intent 对象可通过 getIntent()方法取得，进而将所需回传的数据通过 Bundle 放入 Intent 对象上。回传数据可通过 setResult()方法实现，该方法的第 1 个参数为结果码，对确定性操作，使用 RESULT_OK，该常量由 Android SDK 提供；第 2 个参数为携带数据的 Intent 对象，将 Bundle 数据回传至启动该 Intent 的调用者。MainActivity2 调用 setResult()方法后，则 MainActivity 中 ActivityResultLauncher 初始化时第 2 个参数的接口回调 onActivityResult()方法得以响应，进而在响应中可通过 Intent 获得 Bundle 对象，从而获得 Bundle 携带的数据。MainActivity2 回传结果后，可通过 finish()方法销毁自身，此时任务栈中 MainActivity 处于栈顶，变为可见。

感兴趣的读者，可在本任务基础上，完善项目，设计 3 个 Activity，使之分别显示全国各省份、地级市和区县的城市信息，形成完整的城市分级列表应用。

代码 5-29　活动页面 2——MainActivity2.java

```
import android.content.Intent;
import android.os.Bundle;
import android.view.View;
import android.widget.AdapterView;
import android.widget.ArrayAdapter;
import android.widget.ListView;
import android.widget.Toast;
import androidx.appcompat.app.AppCompatActivity;
import java.util.List;
public class MainActivity2 extends AppCompatActivity {
    ListView lv;
    ArrayAdapter<City> adapter;
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.my_main);
        lv = findViewById(R.id.listView);
        updateWebData();
        lv.setOnItemClickListener(new AdapterView.OnItemClickListener() {
            @Override
            public void onItemClick(AdapterView<?> adapterView, View view,
                                    int i, long l) {
            City item = adapter.getItem(i);
```

```
            Intent intent = getIntent();     //获取启动本 Activity 的 Intent 对象
            Bundle b = new Bundle();
            b.putSerializable(CommonValues.KEY_CITY,item);
            //通过 putSerializable()放置 City 数据
            //City 类需要 implements Serializable 实现序列化
            //取数可通过 Bundle 对象的 getSerializable()方法并强制类型转换获得
            intent.putExtras(b);
            setResult(RESULT_OK,intent);
            //返回数据给 MainActivity,RESULT_OK 是结果码,表示一种状态
            finish();                        //结束 MainActivity2
        }
      });
    }
    private void updateWebData() {
      //从 Bundle 中取出网址,访问资源,更新 ListView
      Bundle bundle = getIntent().getExtras();
      String url = bundle.getString(CommonValues.KEY_URL);
      //获取上一个 Activity 传过来的 url 值
      WebApiUtils.getApiDataAsync(this, url,
          new WebApiUtils.OnReadFinishedListener() {
              @Override
              public void onFinished(List<City> readOutList) {
                  showList(readOutList);
              }
              @Override
              public void onFail(String e) {
                  showToast(e);
              }
          });
      }
      //showList(), showToast()等方法同 MainActivity.java
      …
}
```

5.6 使用 RxHttp 获取 Web API 数据

5.6.1 任务说明

RxHttp 是基于 RxJava+Retrofit+OkHttp 实现的轻量级,完美兼容 MVVM 架构的网络请求封装类库,小巧精致,简单易用。RxHttp 支持 GET、POST、PUT、DELETE 等请求方式,支持文件上传下载及进度侦听,与 RxJava 结合,在实现相同的功能时,其代码量更少,非常受开发者青睐。RxHttp 在 Kotlin 编程环境中使用较多,本着 Java 入门的初衷,在本任务中将介绍如何在 Java 开发环境下配置 RxHttp,以及使用 RxHttp 实现 Web API 数据的获取。本任务需要实现的功能以及运行效果与 5.4 节的任务类似,在此不赘述。

5.6.2 任务实现

1. Gradle 配置

使用 RxHttp 时,在 Module Gradle 文件中需要做较多的配置,具体按以下步骤完成。

步骤 1:在 android 节点的 defaultConfig 标签内增加以下内容。

```
javaCompileOptions {
    annotationProcessorOptions {
        arguments = [
                //使用 asXxx 方法时必须,传入所依赖的 RxJava 版本
                rxhttp_rxjava: '3.1.4',
                rxhttp_okhttp: '4.9.1',
                rxhttp_package: 'rxhttp', //指定 RxHttp 类包名,可随意指定
        ]
    }
}
```

步骤 2：检查 android 节点是否有 compileOptions 配置，若没有，增加以下编译选项配置。

```
compileOptions {
    sourceCompatibility JavaVersion.VERSION_1_8
    targetCompatibility JavaVersion.VERSION_1_8
}
```

步骤 3：在 dependencies 节点中增加以下依赖库。

```
//  以下是必要的 RxHttp 组件
implementation 'com.squareup.okhttp3:okhttp:4.9.1'
implementation 'com.github.liujingxing.rxhttp:rxhttp:2.7.3'
annotationProcessor 'com.github.liujingxing.rxhttp:rxhttp-compiler:2.7.3'

//  以下是 rxlife 管理所需的组件
implementation 'com.github.liujingxing.rxlife:rxlife-coroutine:2.1.0'
//rxlife 管理协程生命周期,页面销毁,关闭请求

//  以下是需要使用 as 方法时所需的组件
implementation 'io.reactivex.rxjava3:rxjava:3.1.4'
implementation 'io.reactivex.rxjava3:rxandroid:3.0.0'
implementation 'com.github.liujingxing.rxlife:rxlife-rxjava3:2.2.2'
//rxlife-rxjava3 管理 RxJava3 生命周期,页面销毁,关闭请求
```

在 settings.gradle 文件中增加 maven{url"https://jitpack.io"}代码仓库，以便依赖库能优先从指定仓库下载对应文件。具体配置如下所示。

```
pluginManagement {
    repositories {
        maven { url "https://jitpack.io" }
        gradlePluginPortal()
        google()
        mavenCentral()
    }
}
dependencyResolutionManagement {
    repositoriesMode.set(RepositoriesMode.FAIL_ON_PROJECT_REPOS)
    repositories {
        maven { url "https://jitpack.io" }
        google()
        mavenCentral()
    }
}
```

2. 上网权限配置

在 AndroidManifest 中增加上网权限如下所示。

```
<uses-permission android:name="android.permission.INTERNET"></uses-permission>
```

若是 Android 10.0 及以上版本,还须在 application 标签内增加访问 Http 资源属性,如下所示。

```
android:usesCleartextTraffic="true"
```

3. 实现 MainActivity

MainActivity 的布局文件 my_main.xml 和解析数据类 City.java 可参考 5.4 节。MainActivity 的实现如代码 5-30 所示。在 MainActivity 中,从网页获取数据并解析数据的核心代码由 getCityList()方法实现,与 5.4 节相比,使用 RxHttp 能用更少的代码实现相同的功能。

RxHttp 采用链式写法,get()方法是使用 GET 获取网络数据,asClass()方法和 asList()方法能将获取结果直接转换为对应类或者类列表,observeOn()方法指定观察者回调的线程。RxHttp 会在获取数据时自行创建后台线程,若无 observeOn()方法指定,默认使用 RxHttp 所在线程处理观察者回调,若 RxHttp 是在后台线程调用的,则需要 observeOn()方法指定 UI 线程,否则 RxHttp 的观察者回调不能直接用于更新 UI。观察者回调使用 subscribe()方法,在解析数据完成后,产生该回调,将所解析的数据回传。subscribe()方法的使用方式比较类似 JavaScript,若使用双参数,前者是解析返回的数据,后者是出错返回的错误对象,各参数均采用箭头函数匿名实现回传参数的处理。本任务中,subscribe()方法中的第 1 个箭头函数处理解析结果 cities(City 类的列表数据),第 2 个箭头函数处理出错信息,箭头前的变量为传递给箭头函数的传参。

代码 5-30　MainActivity.java

```
1   import androidx.appcompat.app.AppCompatActivity;
2   import android.os.Bundle;
3   import android.view.View;
4   import android.widget.ArrayAdapter;
5   import android.widget.EditText;
6   import android.widget.ListView;
7   import android.widget.Toast;
8   import java.util.List;
9   import io.reactivex.rxjava3.android.schedulers.AndroidSchedulers;
10  import rxhttp.RxHttp;
11  public class MainActivity extends AppCompatActivity {
12      ListView lv;
13       @Override
14      protected void onCreate(Bundle savedInstanceState) {
15          super.onCreate(savedInstanceState);
16          setContentView(R.layout.my_main);
17          lv = findViewById(R.id.listView);
18          EditText et = findViewById(R.id.et_url);
19          findViewById(R.id.bt_async)
20                      .setOnClickListener(new View.OnClickListener(){
21              @Override
22              public void onClick(View view) {
```

```
23                  String url = et.getText().toString();
24                  getCityList(url);
25              }
26          });
27      }
28      private void getCityList(String url) {
29          RxHttp.get(url)
30                  .asList(City.class)         //直接将 JSON 数组转换成类列表
31                  .observeOn(AndroidSchedulers.mainThread())     //在主线程中调用观察者
32                  .subscribe(cities -> {      //采用观察者订阅模式回调
33                      showList(cities);
34                  },e->{
35                      showToast(e.toString());
36                  });
37      }
38      private void showList(List<City> readOutList) {
39          //将列表数据生成适配器,显示到 ListView 对象上
40          ArrayAdapter<City> adapter = new ArrayAdapter<>(this,
41                  android.R.layout.simple_list_item_1,readOutList);
42          lv.setAdapter(adapter);
43      }
44      private void showToast(String e) {
45          Toast.makeText(this,e,Toast.LENGTH_LONG).show();
46      }
47  }
```

RxHttp 功能非常强大,这里仅仅是抛砖引玉,给出最基本的 GET 使用方法,读者可自行尝试下载文件以及 POST 操作等高级应用。

5.7　使用 Jsoup 实现网页数据提取

5.7.1　任务说明

本任务使用 Jsoup 实现网页数据提取,应用的演示效果如图 5-8 所示。在该应用中,活动页面视图根节点为垂直的 LinearLayout,布局中依次放置 1 个 TextView,用于显示个人信息;1 个 Button,用于开启后台线程获取网页数据;1 个 ScrollView(滚动视图),视图中内嵌 1 个 TextView,用于显示网页的解析数据。

本任务使用第三方库 Jsoup 解析清华大学出版社新书推荐网页接口(网址请扫描前言中的二维码获取)中的 HTML 数据,并提取出网页中的图书书名、图书作者、图书详情链接,以及图书图片链接等关键信息。如图 5-8 所示,在 ScrollView 内嵌的 TextView 中,显示了 Jsoup 的解析结果,每条图书信息用分隔线隔开,并且所解析的链接支持单击跳转。当 TextView 中显示的内容很长,超出一屏幕的显示内容时,推荐将 TextView 嵌入 ScrollView 视图中,用户则可通过滑动屏幕阅览

图 5-8　解析清华大学出版社新书推荐网页接口数据的演示效果

TextView 的剩余内容。

本任务需要后台线程访问网页,提取网页关键信息,并在 Activity 的 UI 线程中更新数据。后台线程与前端 UI 的数据交互根据之前的任务可知,主流有 3 种方式:①通过 Handler 传递;②通过自定义接口和 Activity 对象的线程切换实现;③通过 LiveData 和视图模型实现。本任务采用第 3 种方式实现,事实上选取上述任何一种方法均能实现相同功能。

5.7.2 任务实现

1. Gradle 和权限配置

在 Module Gradle 文件的 dependencies 节点中增加 Jsoup 依赖,如下所示。

```
implementation 'org.jsoup:jsoup:1.14.3'
```

在 AndroidManifest 的 application 标签之外增加上网权限,如下所示。

```
<uses-permission android:name="android.permission.INTERNET"></uses-permission>
```

对于 Android 10.0 及以上系统,还须在 AndroidManifest 的 application 标签中增加访问 Http 使能属性,如下所示。

```
android:usesCleartextTraffic="true"
```

2. 视图模型与图书数据封装类

使用 Jsoup 访问网页对 Android 系统而言是一项耗时的操作,因此 Jsoup 解析网页数据等相关操作宜放在后台线程中操作,不能直接在 UI 线程中操作。考虑到后台线程与前端 UI 是通过视图模型的 LiveData 交互数据的,本任务还需自定义相关视图模型。

自定义视图模型 MainViewModel 如代码 5-31 所示。在 MainViewModel 中,定义了两个 MutableLiveData 成员变量,其中变量 bookList 用于交互图书列表数据,变量 errMessage 用于交互错误信息。变量 errMessage 在成员变量声明时直接被初始化。变量 bookList 在构造方法中被初始化,初始化时,实际的数据载体 List<BookItem>对象依然是 null,此时可通过 setValue()方法设置 ArrayList 对象,为变量 bookList 赋值。

代码 5-31 自定义视图模型 MainViewModel.java

```
1   import androidx.lifecycle.MutableLiveData;
2   import androidx.lifecycle.ViewModel;
3   import java.util.ArrayList;
4   import java.util.List;
5   public class MainViewModel extends ViewModel {
6       private MutableLiveData<List<BookItem>> bookList;
7       private MutableLiveData<String> errMessage=new MutableLiveData<>();
8       //bookList 是后台线程解析出的图书列表数据
9       //errMessage 是后台线程工作过程中的出错信息
10      public MainViewModel() {
11          bookList =new MutableLiveData<>();
12          bookList.setValue(new ArrayList<>());      //为 bookList 设置初始化列表对象
13      }
14      public MutableLiveData<List<BookItem>> getBookList() {
```

```
15            return bookList;
16        }
17        public MutableLiveData < String > getErrMessage() {
18            return errMessage;
19        }
20    }
```

BookItem 是自定义的图书数据封装类，如代码 5-32 所示，该类有 4 个字段：图书书名 title、图书作者 author、图书详情链接 href、图书图片链接 imgSrc。为了方便 BookItem 类数据的字符串打印，在 BookItem 类中改写了 toString()方法，使用 String. format()方法将相关字段转换为所需字符串。BookItem 实现了 Serializable 接口，使之能用于 Bundle 序列化传数。

代码 5-32　自定义图书数据 BookItem. java

```
1     import java. io. Serializable;
2     public class BookItem implements Serializable{
3         private String title;
4         private String author;
5         private String href;
6         private String imgSrc;
7         public BookItem(String title, String author, String href, String imgSrc) {
8             this. title = title;
9             this. author = author;
10            this. href = href;
11            this. imgSrc = imgSrc;
12        }
13        @Override
14        public String toString() {       //BookItem 对象转字符串的方法,打印各属性值
15            return String. format("Title = % s\nAuthor = % s\nHref = % s\nImgSrc = % s\n",
16                    title,author,href,imgSrc);
17        }
18        public String getTitle() {
19            return title;
20        }
21        public void setTitle(String title) {
22            this. title = title;
23        }
24        public String getAuthor() {
25            return author;
26        }
27        public void setAuthor(String author) {
28            this. author = author;
29        }
30        public String getHref() {
31            return href;
32        }
33        public void setHref(String href) {
34            this. href = href;
35        }
36        public String getImgSrc() {
37            return imgSrc;
38        }
39        public void setImgSrc(String imgSrc) {
40            this. imgSrc = imgSrc;
41        }
42    }
```

3. 实现网页解析后台线程

网页解析后台线程 BookItemGetThread 如代码 5-33 所示,通过构造方法传递视图模型对象 ViewModel 和待解析网址 url。网页中解析的图书详情链接是相对链接,需要与网站前缀地址拼接成完整链接,因此定义了常量 BASE_URL 用于拼接相对链接网址。

BookItemGetThread 后台线程的 run()方法中,通过所传递的视图模型对象获得用于前端和后台交互的图书列表数据维持对象 bookList,注意,bookList 是 LiveData 数据对象,而对应的列表对象 list 则通过 bookList 的 getValue()方法获取。本任务中,视图模型的拥有者设成 MainActivity 对象,即使线程消亡,只要 MainActivity 没有消亡,变量 bookList 依然存在,所维持的列表对象 list 不会消亡,因此在后台线程启动后,须对 list 调用 clear()方法将之前保留的列表数据清除。变量 errMessage 为错误信息,当 Jsoup 解析过程中出错时,可将错误信息传给 errMessage,当 UI 线程中对 errMessage 设置了观察侦听,则可通过对应的回调方法及时处理错误信息。

Jsoup 访问网页可通过 connect()方法接受对应网址,注意网址字符串必须包含"http://"或"https://"等协议。若要设置访问超时,可对 Jsoup 链式操作增加 timeout()方法,参数是 ms 单位的超时时间,最后通过 get()方法获得 Document 文档对象。Jsoup 提取文档对象中的节点可通过 select(String tag)方法实现,传参 tag 为节点的标签类型,返回值为 Elements 对象。Elements 对象需要先定位后使用,常用的定位方式有:first()方法,取得首个元素;last()方法,取得最后一个元素;get(int index)方法,取得传参 index 指定索引位的元素。Elements 对象定位方法返回的元素为 Element,即节点对象。

为了更好地说明问题,现将待解析网页中关键 HTML 内容抽取出,如代码 5-34 所示,并使用该内容用于分析 Jsoup 解析网页的相关方法。由代码 5-34 可见,一条图书信息在 class=" n_b_product"的 dl 节点中,可用 Jsoup 提供的 select("dl[class * =product]")方法匹配 dl 节点进行提取,传参 dl[class * =product]表示只提取具有 class 属性,且属性值包含了 product 字符串的 dl 节点。select()方法返回的是 Elements 对象,获得所需的 dl 节点合集之后,需要对其进行 for 循环展开,对每个 dl 节点进一步提取相关信息。本任务中,对 dl 节点合集使用 for each 展开,从 Elements 对象中取得 Element 元素,用于进一步的解析处理。

由代码 5-34 所示的网页内容可知,图书详情链接在 a 节点的 href 属性中,可通过 dl 节点对象的 select("a").first().attr("href")方法获得 href 属性值,并且与前缀网址 BASE_URL 拼接成完整的链接地址。此外,href 属性值还可以通过 attr("abs:href")直接获得绝对网址,从而避免了与前缀网址的拼接操作。图书信息字段 imgSrc 和 title 可以使用类似的方法提取。图书作者、图书价格和图书简介在 p 节点中,每个 p 节点构成一个字段数据,其中图书作者在首个 p 节点中,可通过 select("p").first().text()方法提取,其中 text()方法为节点对象去标签后的纯文本。从 HTML 中提取的关键信息用于构造 BookItem 对象,并加到列表 list 中。遍历结束,通过 LiveData 对象 bookList 的 postValue()方法更新值,则 Activity 的 UI 线程中同一个 LiveData 对象可在 Observer 接口回调中取出更新后的值,用于更新 ListView。在处理过程中,若遇到异常,则通过 try-catch 捕捉异常,将错误信息转成字符串,并通过 LiveData 对象 errMessage 更新值,从而在 Activity 中可对 errMessage 调用 Observer 接口处理异常。

代码 5-33　网页解析后台线程 BookItemGetThread.java

```
import androidx.lifecycle.MutableLiveData;
import org.jsoup.Jsoup;
import org.jsoup.nodes.Document;
import org.jsoup.nodes.Element;
import org.jsoup.select.Elements;
import java.io.IOException;
import java.util.List;
public class BookItemGetThread extends Thread{
    private MainViewModel viewModel;
    private String url;
    public static final String BASE_URL
            = "http://www.tup.tsinghua.edu.cn/booksCenter/";
    //BASE_URL 为 href 提取链接地址的前缀地址,与 href 拼接成绝对网址才能被访问
    public BookItemGetThread(MainViewModel viewModel, String url) {
        this.viewModel = viewModel;
        this.url = url;
    }
    @Override
    public void run() {
        MutableLiveData<List<BookItem>> bookList = viewModel.getBookList();
        List<BookItem> list = bookList.getValue();
        list.clear();       //将列表对象 list 数据清空
        MutableLiveData<String> errMessage = viewModel.getErrMessage();
        try {
           Document doc = Jsoup.connect(url).timeout(10000).get();
           Elements dls = doc.select("dl[class*=product]");
           //取 dl 标签,标签具有属性 class="*product*"的才被匹配,*表示任意字符
           for (Element dl : dls) {
               String href0 = dl.select("a").first().attr("href");
               String href = BASE_URL + href0;     //拼接成的 href 是绝对网址
               //href 可以使用 attr("abs:href")获取绝对网址
               String imgSrc = dl.select("img").first().attr("abs:src");
               //abs:src 或者 abs:href 可以取得绝对网址
               String title = dl.select("span")
                           .first().attr("title");
               //取得 span 标签中 title 的属性值
               String author = dl.select("p").first().text();
               //取首个段落,对应的是作者信息
               list.add(new BookItem(title,author,href,imgSrc));
           }
           bookList.postValue(list);             //通过 ViewModel 传递数据
        } catch (IOException e) {
           e.printStackTrace();
           errMessage.postValue(e.toString());
           //将错误信息传递给观察者
        }
    }
}
```

代码 5-34　待提取的 HTML 关键内容

```
<dl class="n_b_product">
 <dt>
  <a href="book_08766201.html" target="_blank">
   <img src="../upload/smallbookimg/087662-01.jpg" width="100" height="146" />
```

```
    </a>
  </dt>
  <dd>
    <span title="Python 从菜鸟到高手(第 2 版)">Python 从菜鸟到高手(第 2...</span>
    <p>李宁</p>
    <p class="ft_purple">定价: 95 元</p>
    <p>本书从实战角度系统讲解了 Python 核心知识点以及 Python 在 We...</p>
    </dd>
  </dl>
  <dl class="n_b_product">
  <dt>
   <a href="book_09175301.html" target="_blank">
     <img src="../upload/smallbookimg/091753-01.jpg" width="100" height="146" />
   </a>
  </dt>
  <dd>
  <span title="动手学推荐系统——基于 PyTorch 的算法实现(微课视频版)">动手学推荐
  系统——基于 P...</span>
  <p>於方仁</p>
  <p class="ft_purple">定价: 79 元</p>
  <p>本书从理论结合实践编程来学习推荐系统.由浅入深,先基础...</p>
 </dd>
 </dl>
  …
```

4. 实现 MainActivity

MainActivity 的布局文件 my_main.xml 如代码 5-35 所示。在布局中,id 为 tv_result 的 TextView 组件所显示的内容将超过屏幕高度,若期望能通过上下滑动屏幕显示剩余内容,应将 tv_result 嵌在 ScrollView 中。同时,tv_result 显示的超链接期望能被系统自动识别为网址,当用户单击该链接时,会自动调用相应应用(浏览器)访问网址,该功能需要额外属性 android:autoLink 进行控制,属性值 web 用于识别网址链接、email 用于识别邮箱地址、map 用于识别地理位置、phone 用于识别电话号码、all 则识别所有类型,不同类型的数据会匹配相应的应用。

代码 5-35　MainActivity 的布局文件 my_main.xml

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
      android:orientation="vertical"
      android:layout_width="match_parent"
      android:layout_height="match_parent">
      <TextView
           android:layout_width="match_parent"
           android:layout_height="wrap_content"
           android:text="Your name and ID" />
      <Button
           android:id="@+id/button"
           android:layout_width="match_parent"
           android:layout_height="wrap_content"
           android:text="Get data" />
      <ScrollView
           android:layout_width="match_parent"
           android:layout_height="match_parent">
           <LinearLayout
```

```
                android:layout_width = "match_parent"
                android:layout_height = "wrap_content"
                android:orientation = "vertical" >
                < TextView
                      android:id = "@ + id/tv_result"
                      android:layout_width = "match_parent"
                      android:layout_height = "wrap_content"
                      android:autoLink = "web"
                      android:text = "TextView" />
            </LinearLayout >
        </ScrollView >
</LinearLayout >
```

MainActivity 的实现如代码 5-36 所示。在 MainActivity 中，通过 ViewModelProvider 获得视图模型对象 viewModel，并从中取得两个 LiveData 数据 bookList 和 errMessage，分别对其设置观察侦听。当后台线程对 LiveData 数据通过 postValue()方法更新后，MainActivity 的 UI 线程中对应数据会通过 Observer 接口响应 onChanged()回调，通过回调传参取得更新后的数据并进行处理。在 LiveData 数据 bookList 的 onChanged()回调中，取得图书列表数据 bookItems，并调用 printBookList()方法将其转换为所需格式的字符串显示到文本对象 tv 上。错误信息使用 Toast 显示，通过 LiveData 数据 errMessage 的 Observer 接口实现。网页解析后台线程的生成和启动在 Button 单击事件回调中实现。MainActivity 的成员变量 url 为获取图书信息的网址，网址中，参数 pageIndex 是页索引值，pageSize 是返回一页信息的图书数目，两者均可被修改。在 printBookList()方法中，通过遍历图书列表，将每个 BookItem 对象转为字符串，并在每条图书信息前后增加分隔线，该方法中涉及较多的字符串拼接操作，适合使用 StringBuilder 对象和 append()方法进行字符串拼接，以避免产生过多的字符串内存碎片。

代码 5-36　MainActivity.java

```
import androidx.appcompat.app.AppCompatActivity;
import androidx.lifecycle.MutableLiveData;
import androidx.lifecycle.Observer;
import androidx.lifecycle.ViewModelProvider;
import android.os.Bundle;
import android.view.View;
import android.widget.TextView;
import android.widget.Toast;
import java.util.List;
public class MainActivity extends AppCompatActivity{
    MainViewModel viewModel;
    String url = "http://www.tup.tsinghua.edu.cn/booksCenter/" +
                 "new_book.ashx?pageIndex = 0&pageSize = 15&id = 0&jcls = 0";
    //获取图书数据的网址,pageIndex 为页面索引值,pageSize 为一个页面的数据数目
    //可根据需要修改 pageIndex 和 pageSize 的参数值
    @Override
    protected void onCreate(Bundle savedInstanceState){
        super.onCreate(savedInstanceState);
        setContentView(R.layout.my_main);
        TextView tv = findViewById(R.id.tv_result);
        viewModel =  new ViewModelProvider(this).get(MainViewModel.class);
```

```
        MutableLiveData<List<BookItem>> bookList = viewModel.getBookList();
        //通过视图模型数据观察者模式响应数据动态更新
        bookList.observe(this, new Observer<List<BookItem>>(){
            @Override
            public void onChanged(List<BookItem> bookItems){
                String s = printBookList(bookItems);
                tv.setText(s);
            }
        });
        MutableLiveData<String> errMessage = viewModel.getErrMessage();
        //响应后台线程通过视图模型发送的错误信息
        errMessage.observe(this, new Observer<String>(){
            @Override
            public void onChanged(String s){
                showToast(s);
            }
        });
        findViewById(R.id.button).setOnClickListener(new View.OnClickListener(){
            @Override
            public void onClick(View view){
                new BookItemGetThread(viewModel,url).start();
                //生成并启动后台线程解析网页数据
            }
        });
    }
    private void showToast(String s){
        Toast.makeText(this,s,Toast.LENGTH_LONG).show();
    }
    private String printBookList(List<BookItem> bookItems){
        //将传参 bookItems 转换成字符串
        StringBuilder sb = new StringBuilder();
        for (int i = 0; i < bookItems.size(); i++) {
            BookItem bookItem = bookItems.get(i);
            sb.append("------------" + i + "-----------------\n");
            sb.append(bookItem.toString());
            sb.append("----------------------------------\n");
        }
        return sb.toString();
    }
}
```

5.8 使用 Jsoup 和 Glide 实现网页数据渲染

5.8.1 任务说明

本任务在 5.7 节的基础上完成,效果如图 5-9 所示。活动页面中,将 5.7 节 my_main.xml 布局文件中的 ScrollView 以及子视图更改为 ListView。ListView 每个行视图显示了图书书名、图书作者和图书封面图片。本节任务需要对图书数据类 BookItem 实现对应的自定义适配器,并使用第三方库 Glide 将图片网址的图片数据加载到适配器的 ImageView 组件中。

图 5-9　使用 ListView 显示图文并茂的图书信息

5.8.2　任务实现

1. 实现自定义适配器

复制 5.7 节项目的所有文件，包括 Gradle 设置和 AndroidManifest 的上网相关设置。在 Module Gradle 的 dependencies 节点中增加 Glide 依赖库，如下所示。

```
implementation 'com.github.bumptech.glide:glide:4.13.2'
```

自定义适配器的行视图 row_view.xml 如代码 5-37 所示。布局中，id 为 row_view_tv_title 的 TextView 用于显示图书书名，该组件不仅设置了文本颜色和字体大小，还增加了额外属性 android:ellipsize，用于控制文本内容过长时的处理方式，当属性值为 end 时，截断过长内容的尾部，并用三点省略号“…”取代剩余文本。TextView 可使用行数控制属性 android:maxLines 设置文本框的最大行数。

代码 5-37　适配器行视图 row_view.xml

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    android:layout_width="match_parent"
    android:gravity="center"
    android:padding="5dp"
    android:layout_height="wrap_content">
    <LinearLayout
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_weight="1"
        android:orientation="vertical">
        <TextView
            android:id="@+id/row_view_tv_title"
            android:layout_width="match_parent"
```

```
                android:layout_height = "wrap_content"
                android:textColor = "#000"
                android:textSize = "16sp"
                android:ellipsize = "end"
                android:maxLines = "3"
                android:text = "TextView" />
            < TextView
                android:id = "@ + id/row_view_tv_author"
                android:layout_width = "match_parent"
                android:layout_height = "wrap_content"
                android:text = "TextView" />
        </LinearLayout >
        < ImageView
            android:id = "@ + id/row_view_iv"
            android:layout_width = "80dp"
            android:layout_height = "80dp"
            app:srcCompat = "@drawable/ic_launcher_background" />
</LinearLayout >
```

自定义图书适配器 BookItemAdapter 如代码 5-38 所示。在适配器的 getView()方法中,取出图书数据的图片链接赋给变量 imgSrc,并使用 Uri. parse()方法将其转换为标准 Uri 资源,提供给 Glide 对象加载网络数据。Glide 的使用相对比较简单,可通过链式操作完成图片加载,其中,with()方法设置的是 Context 对象或者 Activity 对象; load()方法加载图片数据源,图片网址可通过 Uri. parse()方法转换为标准 Uri 资源,也可以直接使用 String 类型的网址; placeholder()方法指定加载过程中替换的图片,该方法是可选项; into()方法指定加载图片赋予的 ImageView 对象。Glide 会自动开启后台线程加载网络图片,无须用户干预,并且具有缓存功能,当加载同一个网址图片时,可利用缓存数据加速加载。

代码 5-38　自定义适配器 BookItemAdapter. java

```
import android.content.Context;
import android.net.Uri;
import android.view.LayoutInflater;
import android.view.View;
import android.view.ViewGroup;
import android.widget.ArrayAdapter;
import android.widget.ImageView;
import android.widget.TextView;
import androidx.annotation.NonNull;
import androidx.annotation.Nullable;
import com.bumptech.glide.Glide;
import java.util.List;
public class BookItemAdapter extends ArrayAdapter < BookItem > {
    private Context context;
    private List < BookItem > list;
    public BookItemAdapter(@NonNull Context context, List < BookItem > list) {
        super(context, android.R.layout.simple_list_item_1,list);
        this.context = context;
        this.list = list;
    }
    @NonNull
    @Override
    public View getView(int position, @Nullable View convertView,
                                    @NonNull ViewGroup parent) {
```

```
        View v = convertView;
        if(v == null){
            v =  LayoutInflater.from(context)
                    .inflate(R.layout.row_view,parent, false);
        }
        BookItem bookItem  =  list.get(position);
        //取出行位置对应的对象,填充给行视图各 UI
        TextView tv_title = v.findViewById(R.id.row_view_tv_title);
        TextView tv_author = v.findViewById(R.id.row_view_tv_author);
        ImageView iv = v.findViewById(R.id.row_view_iv);
        tv_title.setText(bookItem.getTitle());
        tv_author.setText(bookItem.getAuthor());
        String imgSrc  =  bookItem.getImgSrc();
        Glide.with(context).load(Uri.parse(imgSrc)).into(iv);
        //使用 Glide 加载图片网址,并把图片数据渲染到 ImageView 对象 iv 上
        return v;
    }
}
```

2. 实现 MainActivity

MainActivity 的布局文件 my_main.xml 如代码 5-39 所示,相比 5.7 节的项目,活动页面布局中,将 ScrollView 及子视图改成了 ListView。

代码 5-39　MainActivity 的布局文件 my_main.xml

```
<?xml version = "1.0" encoding = "utf - 8"?>
< LinearLayout xmlns:android = "http://schemas.android.com/apk/res/android"
      android:orientation = "vertical"
      android:layout_width = "match_parent"
      android:layout_height = "match_parent">
      < TextView
           android:layout_width = "match_parent"
           android:layout_height = "wrap_content"
           android:text = "Your name and ID" />
      < Button
           android:id = "@ + id/button"
           android:layout_width = "match_parent"
           android:layout_height = "wrap_content"
           android:text = "Get data" />
      < ListView
           android:id = "@ + id/listView"
           android:layout_width = "match_parent"
           android:layout_height = "match_parent" />
</LinearLayout >
```

MainActivity 的实现如代码 5-40 所示。在 onCreate()方法中,ListView 设置了列表项单击侦听,在侦听回调中,取得图书数据 item,并将 item 的 href 值转换为标准 Uri 资源,通过动作＋数据的方式定义了意图对象 intent,以 Activity 的方式启动该意图,使其调用默认浏览器访问对应网页,达到跳转到图书详情页面的目的。LiveData 对象 bookList 通过 observe()方法实现数据侦听,取得后台线程解析网页所获取的图书列表数据,生成对应适配器对象,用于更新 ListView。

代码 5-40　MainActivity. java

```
import androidx.appcompat.app.AppCompatActivity;
import androidx.lifecycle.MutableLiveData;
import androidx.lifecycle.Observer;
import androidx.lifecycle.ViewModelProvider;
import android.content.Intent;
import android.net.Uri;
import android.os.Bundle;
import android.view.View;
import android.widget.AdapterView;
import android.widget.ListView;
import android.widget.Toast;
import java.util.List;
public class MainActivity extends AppCompatActivity {
    MainViewModel viewModel;
    String url = "http://www.tup.tsinghua.edu.cn/booksCenter/" +
           "new_book.ashx?pageIndex = 0&pageSize = 15&id = 0&jcls = 0";
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.my_main);
        viewModel = new ViewModelProvider(this).get(MainViewModel.class);
        MutableLiveData<List<BookItem>> bookList = viewModel.getBookList();
        ListView lv = findViewById(R.id.listView);
        lv.setOnItemClickListener(new AdapterView.OnItemClickListener() {
            @Override
            public void onItemClick(AdapterView<?> adapterView, View view,
                                    int i, long l) {
                BookItem item = (BookItem) adapterView.getItemAtPosition(i);
                Intent intent = new Intent(Intent.ACTION_VIEW,
                       Uri.parse(item.getHref()));
                startActivity(intent);     //调用内置浏览器跳转到图书详情页面
            }
        });
        //通过视图模型数据观察者模式响应数据动态更新
        bookList.observe(this, new Observer<List<BookItem>>() {
            @Override
            public void onChanged(List<BookItem> bookItems) {
                BookItemAdapter adapter = new BookItemAdapter(MainActivity.this,
                        bookItems);
                lv.setAdapter(adapter);
            }
        });
        MutableLiveData<String> errMessage = viewModel.getErrMessage();
        //响应后台线程通过视图模型发送的错误信息
        errMessage.observe(this, new Observer<String>() {
            @Override
            public void onChanged(String s) {
                showToast(s);
            }
        });
        findViewById(R.id.button).setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View view) {
                new BookItemGetThread(viewModel,url).start();
                //生成并启动后台线程解析图书数据
```

```
56              }
57          });
58      }
59      private void showToast(String s) {
60          Toast.makeText(this,s,Toast.LENGTH_LONG).show();
61      }
62  }
```

5.9　使用 SwipeRefreshLayout 和 WebView

5.9.1　任务说明

本任务在 5.8 节的基础上完成，演示效果如图 5-10 所示。本任务的应用由两个 Activity 构成，分别为 MainActivity 和 BookItemActivity。MainActivity 用于显示图书列表，BookItemActivity 用于显示图书详情。

图 5-10　任务的演示效果

在 MainActivity 布局中，对 ListView 增加父视图 SwipeRefreshLayout，使其具有下拉刷新功能，在下拉过程中，将指向下一页的图书列表网址给待启动的后台线程，图书列表加载完成后通过 LiveData 更新 ListView，从而实现了图书列表换页的功能。MainActivity 布局中还增加了 SeekBar 组件，采用离散进度样式，拖动 SeekBar 组件，可更改网址中的页索引参数值，实现网址换页。SeekBar 和 SwipeRefreshLayout 联动，在 SwipeRefreshLayout 下拉刷新侦听接口中，同步更改 SeekBar 的当前进度值。ListView 实现了列表项单击事件侦听，在侦听回调中，获得对应图书数据，传递并跳转至 BookItemActivity。

BookItemActivity 取得 MainActivity 所传递的图书数据，使用 WebView 组件显示图书详情链接对应的 HTML 数据。BookItemActivity 的动作栏标题显示图书书名，并且有应用返回键，用户单击返回键可返回至 MainActivity。

5.9.2 任务实现

1. Gradle 配置

本任务中,Jsoup 和 Glide 的配置以及 AndroidManifest 上网权限等静态配置同 5.8 节。SwipeRefreshLayout 在多数 Android Studio 版本中不存在,需要以第三方库的形式导入。在 Module Gradle 文件的 dependencies 节点中导入 SwipeRefreshLayout,如下所示。

```
implementation 'androidx.swiperefreshlayout:swiperefreshlayout:1.1.0'
```

2. SwipeRefreshLayout 使用要点

SwipeRefreshLayout 作为 Android 的第三方库,使用方式较为简单。SwipeRefreshLayout 通过 setOnRefreshListener()方法设置下拉刷新侦听器,用于启动后台线程获取新数据。在刷新侦听接口中,通过 onRefresh()回调方法启动后台线程,此时 SwipeRefreshLayout 以转圈的方式悬浮在屏幕上方,若没有通过 setRefreshing(false)方法取消刷新状态,SwipeRefreshLayout 则会一直以刷新转圈的状态悬浮在应用视图上。基于异步工作的原理,SwipeRefreshLayout 的 setRefreshing(false)方法建议在后台线程数据回传的回调方法中执行。

3. 实现 MainActivity

MainActivity 的布局文件 my_main.xml 如代码 5-41 所示,在布局中,将 ListView 嵌入到 SwipeRefreshLayout 组件中,使用户下拉 ListView 时能触发 SwipeRefreshLayout 的刷新回调。在编辑 XML 时,UI 面板中没有 SwipeRefreshLayout 组件,需要用户通过 Gradle 加载对应库后,才能在 XML 节点中通过手动输入组件名称添加 SwipeRefreshLayout。SwipeRefreshLayout 的宽度与 ListView 相同,可用 match_parent 属性值设置;该组件的高度包围 ListView,可用 wrap_content 属性值设置;此外还需要添加 android:id 属性。布局文件中增加了 SeekBar 组件,用于显示和控制网址中的页索引参数 pageIndex,SeekBar 具有连续模式和离散模式,使用 style 属性控制,本任务中使用了离散模式,使 SeekBar 的拖动值只能是整数值。SeekBar 的 android:max 属性用于控制拖动条的最大进度值,android:progress 属性则控制拖动条的当前进度值。

代码 5-41 MainActivity 的布局文件 my_main.xml

```
<?xml version = "1.0" encoding = "utf - 8"?>
< LinearLayout xmlns:android = "http://schemas.android.com/apk/res/android"
      android:layout_width = "match_parent"
      android:layout_height = "match_parent"
      android:orientation = "vertical">
      < TextView
           android:layout_width = "match_parent"
           android:layout_height = "wrap_content"
           android:text = "Your name and ID" />
      < SeekBar
           android:id = "@ + id/seekBar"
           style = "@style/Widget.AppCompat.SeekBar.Discrete"
           android:layout_width = "match_parent"
           android:layout_height = "wrap_content"
           android:max = "10"
           android:progress = "0" />
```

```
<!-- android:max 设置 SeekBar 的最大进度值,android:progress 则设置当前进度值 -->
    <androidx.swiperefreshlayout.widget.SwipeRefreshLayout
        android:id="@+id/swipeRefreshLayout"
        android:layout_width="match_parent"
        android:layout_height="wrap_content">
        <ListView
            android:id="@+id/listView"
            android:layout_width="match_parent"
            android:layout_height="match_parent" />
    </androidx.swiperefreshlayout.widget.SwipeRefreshLayout>
</LinearLayout>
```

MainActivity 的实现如代码 5-42 所示。在 MainActivity 中，常量 KEY_DATA 作为 Bundle 对象存取数据的 key，通过静态方法 getIntentBookItem()从 Bundle 对象中取得所传输的 BookItem 对象，实现不同 Activity 之间对 Bundle 对象字段的解耦。变量 urlList 用于存储图书列表数据的网址，并使用常量 MAX_PAGE 控制图书列表网址参数 pageIndex 的最大值，变量 page_i 则为当前网址的页索引，与变量 urlList 配合使用。自定义方法 iniUrlList()用于初始化网址列表变量 urlList，在 for 循环中更改网址参数 pageIndex 的值，将生成的网址添加到变量 urlList 中。

SeekBar 对象通过 setMax()方法设置进度最大值，并且最大值由常量 MAX_PAGE 控制。SeekBar 对象通过 setProgress()方法设置 SeekBar 的当前进度值，该值由变量 page_i 控制。SeekBar 对象通过 setOnSeekBarChangeListener()方法设置进度值改变侦听，在侦听接口中共有 3 个回调方法，其中，onProgressChanged()方法在进度值发生改变时触发；onStartTrackingTouch()方法在用户刚开始拖拽拖动条时触发；onStopTrackingTouch()方法在用户释放拖动条时触发。显然，更新页面索引，并启动后台线程加载新的图书列表应该在 onStopTrackingTouch()方法中处理，若在 onProgressChanged()方法中处理，每改变一次进度值，将启动一次后台线程，若用户连续拖拽拖动条，将产生多次没有必要的后台线程调用。在 onStopTrackingTouch()方法中，对 SeekBar 对象调用 getProgress()方法取得拖动条当前值，并赋给变量 page_i，再调用 updateWebPage()方法，从网址列表 urlList 中取得 page_i 索引的网址，进而启动后台线程获取图书列表数据。

SwipeRefreshLayout 的下拉刷新侦听可通过 setOnRefreshListener()方法进行匿名实现，在刷新回调 onRefresh()方法中，页面索引值 page_i 自增，并判断是否越界，若越界则将 page_i 重置为 0，最后通过调用 updateWebPage()方法启动线程，实现图书列表数据的异步加载。

MainActivity 没有使用按钮触发后台线程获取图书列表，而是直接在 onCreate()方法中调用 updateWebPage()方法获取图书数据，此时 page_i 默认为 0，从而实现应用启动时，直接获取首页的图书列表数据。对 ListView 每做一次下拉刷新，变量 page_i 会自增 1 次，并获取 page_i 指向的图书数据用于更新 ListView，此时 SwipeRefreshLayout 会一直处于刷新状态，须在后台数据回调时结束刷新状态，因此，在变量 bookList 和 errMessage 的观察回调中，须将 SwipeRefreshLayout 的刷新状态设为 false，使刷新状态消失。

ListView 设置了列表项单击侦听，在 onItemClick()回调方法中，取得被单击的 BookItem 数据，将之封装在 Bundle 对象中，用于 Intent 数据传输。Bundle 对象没有直接

设置自定义类的存取数方法,可以使用 putSerializable()方法来存数,使用 getSerializable()方法来取数。这两个方法的使用前提是自定义类需要实现序列化(Serializable)接口,并且 getSerializable()方法返回的是 Serializable 对象,需要强制转换成用户指定的类对象。MainActivity 跳转至 BookItemActivity 通过 Intent 实现,并在 Intent 中通过 Bundle 携带了所需要传输的 BookItem 数据。MainActivity 实现了 getIntentBookItem()方法,可从传参 Intent 对象中取得 BookItem 数据,从而使 BookItemActivity 可以直接调用 getIntentBookItem()方法取得 Activity 之间所传递的数据。

代码 5-42　MainActivity. java

```
1   import androidx.appcompat.app.AppCompatActivity;
2   import androidx.lifecycle.MutableLiveData;
3   import androidx.lifecycle.Observer;
4   import androidx.lifecycle.ViewModelProvider;
5   import androidx.swiperefreshlayout.widget.SwipeRefreshLayout;
6   import android.content.Intent;
7   import android.os.Bundle;
8   import android.view.View;
9   import android.widget.AdapterView;
10  import android.widget.ListView;
11  import android.widget.SeekBar;
12  import android.widget.Toast;
13  import java.util.ArrayList;
14  import java.util.List;
15  public class MainActivity extends AppCompatActivity {
16      public static final String KEY_DATA = "key_data";
17      MainViewModel viewModel;
18      ArrayList<String> urlList;
19      int MAX_PAGE = 20;                          //控制网址参数 pageIndex 的最大值
20      int page_i = 0;
21      SwipeRefreshLayout refreshLayout;
22      @Override
23      protected void onCreate(Bundle savedInstanceState) {
24          super.onCreate(savedInstanceState);
25          setContentView(R.layout.my_main);
26          viewModel = new ViewModelProvider(this).get(MainViewModel.class);
27          MutableLiveData<List<BookItem>> bookList = viewModel.getBookList();
28          ListView lv = findViewById(R.id.listView);
29          iniUrlList();
30          SeekBar seekBar = findViewById(R.id.seekBar);
31          seekBar.setMax(MAX_PAGE);           //设置 SeekBar 对象进度最大值
32          seekBar.setProgress(page_i);        //设置 SeekBar 对象当前进度值
33          seekBar.setOnSeekBarChangeListener(
34                                      new SeekBar.OnSeekBarChangeListener() {
35              @Override
36              public void onProgressChanged(SeekBar seekBar, int progress,
37                                            boolean fromUser) {
38
39              }
40              @Override
41              public void onStartTrackingTouch(SeekBar seekBar) {
42
43              }
44              @Override
45              public void onStopTrackingTouch(SeekBar seekBar) {
```

```
46                  //用户释放拖动 SeekBar 后,再加载数据
47                  page_i =  seekBar.getProgress();
48                  //从 SeekBar 对象取得进度值,用于更新 page_i
49                  updateWebPage();
50              }
51          });
52          refreshLayout = findViewById(R.id.swipeRefreshLayout);
53          refreshLayout.setOnRefreshListener(
54                  new SwipeRefreshLayout.OnRefreshListener() {
55                      @Override
56                      public void onRefresh() {
57                          page_i++;
58                          if(page_i > MAX_PAGE){//防止页面索引值越界
59                              page_i = 0;
60                          }
61                          seekBar.setProgress(page_i);
62                          updateWebPage();     //根据 page_i 值取相应网址更新图书数据
63                      }
64                  }
65          );
66        updateWebPage();                       //更新 page_i 指向的页面
67        lv.setOnItemClickListener(new AdapterView.OnItemClickListener() {
68           @Override
69           public void onItemClick(AdapterView<?> adapterView, View view,
70                                   int i, long l) {
71              BookItem item =  (BookItem) adapterView.getItemAtPosition(i);
72              Intent intent  =  new Intent(MainActivity.this,
73                                      BookItemActivity.class);
74              Bundle b  =  new Bundle();
75              b.putSerializable(KEY_DATA,item);
76              //BookItem 类需要实现 Serializable 接口
77              intent.putExtras(b);             //对 Intent 对象添加需要传递的 Bundle 数据
78              startActivity(intent);           //跳转到 BookItemActivity
79           }
80        });
81        //通过视图模型数据观察者模式响应数据动态更新
82        bookList.observe(this, new Observer<List<BookItem>>() {
83           @Override
84           public void onChanged(List<BookItem> bookItems) {
85              refreshLayout.setRefreshing(false);       //取消刷新状态
86              BookItemAdapter adapter = new BookItemAdapter(MainActivity.this,
87                                                            bookItems);
88              lv.setAdapter(adapter);
89           }
90        });
91        MutableLiveData<String> errMessage  =  viewModel.getErrMessage();
92        //响应后台线程通过视图模型发送的错误信息
93        errMessage.observe(this, new Observer<String>() {
94           @Override
95           public void onChanged(String s) {
96              refreshLayout.setRefreshing(false);       //取消刷新状态
97              showToast(s);
98           }
99        });
100     }
101     private void updateWebPage() {
102        refreshLayout.setRefreshing(true);
```

```
        //设置 SwipeRefreshLayout 组件处于刷新状态
        String url = urlList.get(page_i);
        new BookItemGetThread(viewModel,url).start();
        //生成并启动后台线程解析图书数据
    }
    private void iniUrlList() {//生成图书信息网址列表
        urlList = new ArrayList<>();
        String url_p = "http://www.tup.tsinghua.edu.cn/booksCenter/" +
                "new_book.ashx?pageIndex= %d&pageSize=15&id=0&jcls=0";
        //url_p 为用于打印网址的临时变量,pageIndex 值由 %d 控制
        for (int i = 0; i <= MAX_PAGE ; i++) {
            String url = String.format(url_p, i);
            urlList.add(url);
        }
    }
    private void showToast(String s) {
        Toast.makeText(this,s,Toast.LENGTH_LONG).show();
    }
    public static BookItem getIntentBookItem(Intent intent){
        //传参为 Intent 对象,从 Intent 对象中获得 Bundle 对象,进而获得 BookItem 数据
        //通过静态方法解耦 Bundle 数据的 key
        Bundle b = intent.getExtras();
        return (BookItem) b.getSerializable(KEY_DATA);
    }
}
```

4. 实现 BookItemActivity

BookItemActivity 的布局文件 activity_book_item.xml 如代码 5-43 所示,在垂直的 LinearLayout 中嵌入 WebView 组件,用于显示图书详情的网页内容。

代码 5-43 BookItemActivity 的布局文件 activity_book_item.xml

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical">
    <WebView
        android:id="@+id/webView"
        android:layout_width="match_parent"
        android:layout_height="match_parent" />
</LinearLayout>
```

BookItemActivity 的实现如代码 5-44 所示。WebView 通过 getSettings()方法得到设置对象,并通过该对象使能 JavaScript 和缩放功能。WebView 默认没有开启 JavaScript 功能,若不通过 setJavaScriptEnabled()方法开启使能,WebView 将无法运行加载网页中的 JavaScript 代码,使部分功能无法正常工作。此外,WebView 显示的 HTML 页面,由于样式原因,会造成页面中文字过大,无法显示完整页面的情况,此时需要通过 setBuiltInZoomControls()方法使之支持缩放模式,以及 setUseWideViewPort()方法设置 WebView 处于 WideViewPort 模式,使页面匹配 WebView 的尺寸。BookItemActivity 的动作栏标题可通过 getSupportActionBar()方法得到 ActionBar 对象,进而通过 setTitle()方法设置标题。对于有些 Web 网站,会检测用户的浏览器类型,若是移动端的,访问网站时

则会发生重定向，重新加载专门针对移动端的网址。WebView 发生重定向会调用系统的浏览器加载重定向网址，此时 BookItemActivity 和系统浏览器是独立的两个 Activity，若要强制 WebView 加载重定向网址，则需要对 WebView 对象的 setWebViewClient()方法以及改写的 shouldOverrideUrlLoading()方法处理重定向问题。

代码 5-44　BookItemActivity.java

```
import androidx.appcompat.app.ActionBar;
import androidx.appcompat.app.AppCompatActivity;
import android.os.Bundle;
import android.webkit.WebSettings;
import android.webkit.WebView;
import android.webkit.WebViewClient;
public class BookItemActivity extends AppCompatActivity {
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_book_item);
        WebView webView = findViewById(R.id.webView);
        WebSettings settings = webView.getSettings();
        //得到 WebView 的设置对象
        settings.setJavaScriptEnabled(true);        //对 WebView 使能 JavaScript 功能
        settings.setBuiltInZoomControls(true);      //设置支持内置的缩放模式
        settings.setUseWideViewPort(true);
        //设置 WideViewPort 模式，使得网页匹配 WebView 宽度
        BookItem bookItem = MainActivity.getIntentBookItem(getIntent());
        String url = bookItem.getHref();
        String title = bookItem.getTitle();
        ActionBar actionBar = getSupportActionBar();
        actionBar.setTitle(title);                  //设置 Activity 的标题
        webView.loadUrl(url);
        //当页面发生重定向，默认调用系统浏览器加载 url，不会在 WebView 视图中加载网页
        webView.setWebViewClient(new WebViewClient(){
            //使用布局中的 WebView 组件加载重定向网页
            @Override
            public boolean shouldOverrideUrlLoading(WebView view, String url) {
                view.loadUrl(url);
                return super.shouldOverrideUrlLoading(view, url);
            }
        });
    }
}
```

5. 设置声明文件

本节的项目中具有两个 Activity，若不对项目的声明文件 AndroidManifest.xml 进行配置，MainActivity 无法启动 BookItemActivity。项目声明文件如代码 5-45 所示，代码中仅列出需要修改的内容。项目中每一个活动页面均需要 activity 标签进行声明，其中 BookItemActivity 的父活动页面是 MainActivity，通过 android:parentActivityName 属性进行设置，只有设置了该属性后，BookItemActivity 的动作栏上才会出现返回键，用户可通过单击返回键跳转至 MainActivity。BookItemActivity 没有设置启动模式，默认是 standard 模式，每启动一次 BookItemActivity 均会创建新的活动页面。与之相反的是，MainActivity 通过 android:launchMode 属性设置为 singleTask 模式，使之在任务栈中唯

一,BookItemActivity 通过返回键跳转至 MainActivity 时,MainActivity 不会被重建。

代码 5-45　项目的声明文件 AndroidManifest. xml

```
<application
…
      android:usesCleartextTraffic = "true">
      <activity
            android:name = ".BookItemActivity"
            android:parentActivityName = ".MainActivity"
            android:exported = "false" />
      <activity
            android:name = ".MainActivity"
            android:launchMode = "singleTask"
            android:exported = "true">
            <intent - filter>
                  <action android:name = "android.intent.action.MAIN" />
                  <category android:name = "android.intent.category.LAUNCHER" />
            </intent - filter>
      </activity>
</application>
```

5.10　本章综合作业

编写一个新闻 App,具有 2 个 Activity,分别用于显示新闻列表和新闻详情页面。新闻源可用 Jsoup 解析 HTML,或者使用 Web API 获取 JSON 新闻数据。

(1) 新闻列表 Activity 的页面布局中有 ListView 或者 RecyclerView,可显示新闻的图片、新闻标题和新闻发布时间等信息,并支持下拉刷新换页。新闻列表响应列表项单击事件,在回调方法中获得对应新闻的链接,并启动新闻详情页面 Activity,显示单击项的新闻详情。

(2) 新闻详情页面 Activity,可采用 WebView 组件或者自定义布局实现,显示新闻的标题、发布时间、分段新闻详情内容和新闻图片。单击新闻详情页面动作栏的返回键,可返回新闻列表 Activity。

读者可根据兴趣,自由发挥,完善新闻 App 的界面设计和功能开发。