第 5 章　Android 的并发处理

在移动应用中常常需要多个任务同时处理。对于访问在线资源、访问数据库、解析数据、加载音频视频等特别耗时的操作或者在移动应用中必须同时处理多任务时，可以使用 Android 系统的多线程、Handler 机制及异步任务来执行并发处理。本章将结合多线程、Handlder 消息处理机制、异步任务来解决多任务。由于 Android 11 及以后版本不支持 AsyncTask 来处理异步任务，而是使用 Kotlin 的协程处理异步任务，因此本章会介绍如何使用 Kotlin 协程解决异步任务处理。

5.1　多　线　程

线程是操作系统调度的最小单位，是依附进程而存在的。Kotlin 对 Java 的线程进行了封装，简化的线程的处理。

创建一个线程类，有两种形式，一种是定义类，让它实现 Runnable 接口，形式如下：

```
class MyThread: Runnable{
    override fun run(){
        println("定义实现 Runnable 的类")
    }
}
```

然后，利用这个线程体类对象创建线程对象并启动线程。代码的表现形式如下：

```
Thread(MyThread()).start()
```

另外一种自定义线程的定义方式是，定义一个 Thread 类的子类，然后创建这个线程类的对象，再调用 start()方法启动线程，形式如下：

```
Thread {
    override fun run() {
        println("使用对象表达式创建")
    }
}.start()
```

Kotlin 对线程对象的创建和启动进行简化，可以采用下列方式来创建和启动线程：

```
thread {
    println("running from thread(): ${Thread.currentThread()}")
}
```

每一个启动的 Android 移动应用都有一个单独的一个进程。这个进程中有多个线程。

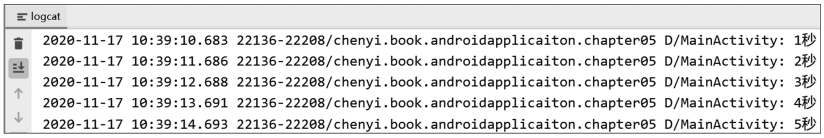
这些线程中仅有一个 UI 主线程。Android 应用程序运行时创建的 UI 主线程主要负责控制 UI 界面的显示、更新和控件交互。Android 程序创建之初，一个进程是单线程状态，所有的任务都在主线程运行，这会导致 UI 主线程的负担过重。一个应用中可通过定义多个线程来完成不同任务，分担主线程的责任，避免主线程阻塞。当主线程阻塞超过规定时间，会出现 ANR(Application Not Responding，应用程序无响应)问题，导致程序中断运行。

例 5-1 显示计时的应用实例 1，代码如下：

```
//模块 Ch05_01 主活动定义 MainActivity.kt
class MainActivity : AppCompatActivity() {
    private var running =true       //设置线程运行的控制变量
    private var time =0
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)
        startBtn.setOnClickListener {//定义按钮执行启动线程
            running =true
            thread{                 //创建并启动自定义线程
                while(running){
                    Thread.sleep(1000)
                    time++
                    Log.d("MainActivity","${time}秒")
                }
            }
        }
        endBtn.setOnClickListener {
                                    //定义按钮执行停止线程,设置线程运行的条件为 false
            running =false
        }
    }
}
```

运行效果如图 5-1 所示。

(a) 运行界面



(b) 运行的日志

图 5-1 计时运行

在 MainActivity 中定义并启动了一个自定义线程。这个线程的主要任务是每秒更新

一次日志并记录流逝的时间。再通过布尔值 running 来控制线程运行。单击“开始”按钮，running 为 true，观察日志可以发现，时间动态显示。单击“结束”按钮，设置 running 为 false，使得线程停止。观察日志，可以发现停止计时。

对上述例子进行修改，自定义线程体，让文本框 timeTxt 内容发生变化，代码如下：

```
class MainActivity : AppCompatActivity() {
    private var running =true
    private var time =0
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)
        startBtn.setOnClickListener {
            running =true
            thread{
                while(running){
                    Thread.sleep(1000)
                    time++
                    timeTxt.text ="${time}秒"
                    Log.d("MainActivity","${time}秒")
                }
            }
        }
        endBtn.setOnClickListener {
            running =false
        }
    }
}
```

执行上述代码，会出现 android.view.ViewRootImpl $CalledFromWrongThreadException 异常，导致程序中断闪退移动应用。这是因为自定义线程无法修改主线程定义的 UI 控件。只有 UI 主线程才能修改和变更 UI 控件。下一节 Handler 机制中来解决这个问题。

5.2 Handler 机制

用于消息(Message)处理的 Handler 机制是一种异步处理方式。通过 Handler 机制可以实现不同线程之间的通信。Handler 机制相关的类有 Looper、MessageQueue、Message 和 Handler 类。它们各司其职，彼此之间的关系如图 5-2 所示。

其中，Looper 在线程内部定义，每个线程只能定义一个 Looper。Looper 内部封装了 MessageQueue(消息队列)。Looper 对象消息队列进行循环管理。通常所说的 Looper 线程就是循环工作的线程。一个线程不断循环，一旦有新任务则执行，执行完毕，继续等待下一个任务，以这种方式执行任务的线程就是 Looper 线程。

Message 也称为任务。有时将 Runnable 线程体对象封装成 Message 对象来处理其中的任务。Handler 对象发送、接收并进行 Message 的处理。Message 封装了任务携带的信息和处理该任务的 handler。可以直接调用 Message 构造函数来创建 Message 对象，但是这种方法并不常用。最常见获取 Message 对象的方式如下。

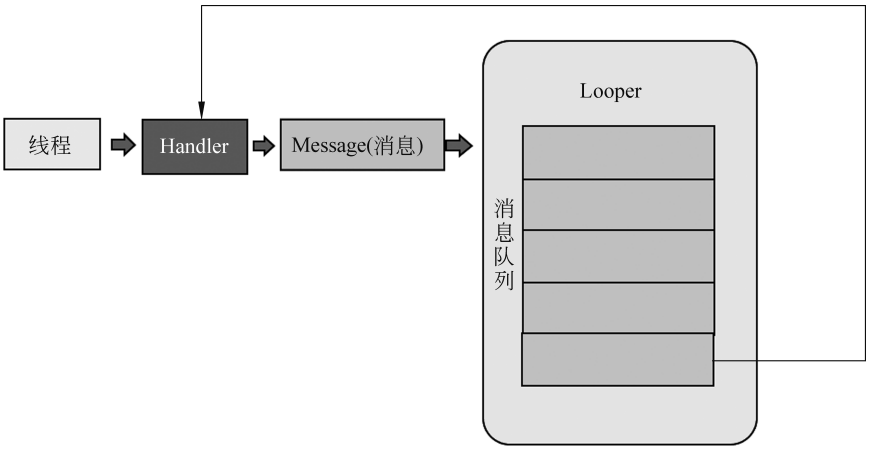


图 5-2　Handler 消息处理机制

(1) 通过 Message.obtain()函数从消息池中获得空消息对象,以节省资源。

(2) 通过 handler.obtainMessage()函数获得 Message 对象实例,即从全局的消息池中获得消息对象。

Message 对象具有属性 Message.arg1 和 Message.arg2,它们用来传递基本简单数据信息,例如数值等。这比用 Bundle 更省内存。如果需要传递更复杂的对象,可以通过消息 Message 对象的 obj 属性来实现。Message 对象的 what 属性来标识信息,以便用不同方式处理消息对象。

当产生一个消息时,关联 Looper 的 Handler 对象会将 Message 对象发送给 MessageQueue。Looper 对 MessageQueue 进行管理和控制,控制 Message 进出 MessageQueue。一旦 Message 从 MessageQueue 出列,可以使用 Handler 对这个 Message 对象进行处理。

Handler 既是处理器,也是调度器,用于调度和处理线程。它最主要的工作是发送和接收并处理 Message。Handler 往往与 Looper 关联。Handler 可以调度 Message,将一个任务切换到某个指定的线程中去执行。在 Handler 中常见的任务是用于更新 UI。在其他线程也称为工作线程(Work Thread),修改了数据,而这些数据会影响到 UI 的界面。因为这些工作线程自己并不能变更 UI,所以常见的处理方式是将数据封装成 Message,并通过 Handler 对象发送到 MessageQueue 中。最后在 UI 主线程中接收这些数据,对 UI 界面进行变更,如图 5-3 所示。

例 5-2　显示计时的应用实例 2,代码如下:

```
//模块 Ch05_02 MainActivity.kt
class MainActivity : AppCompatActivity() {
    var running =true
    val handler =object: Handler(Looper.get MainLooper()){
        override fun handleMessage(msg: Message) {
            if(msg.what ==0x123){                          //判断消息的来源
                timeTxt.text ="${msg.arg1}秒"
```

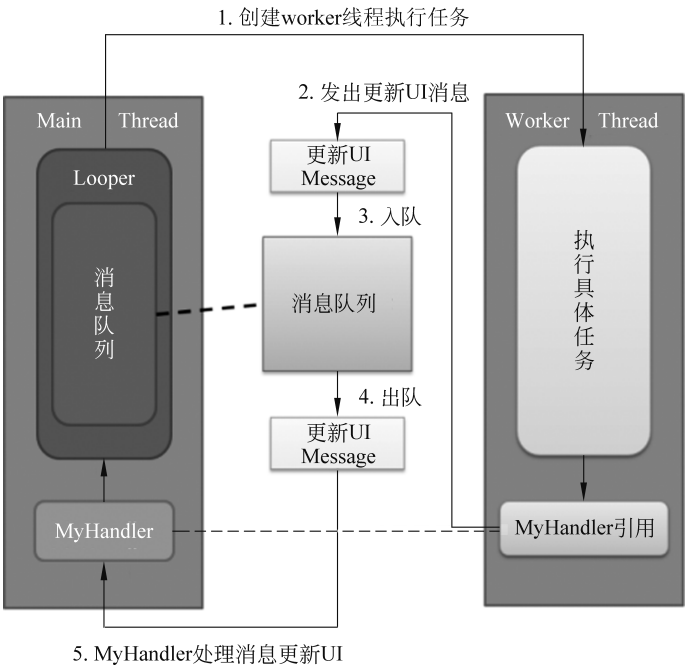


图 5-3　Handler 机制

```
            }
        }
    }
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)
        startBtn.setOnClickListener {                    //定义按钮处理启动线程
            running =true
            var time =0
            thread{                                      //创建并启动工作线程
                while(running){
                    Thread.sleep(1000)                   //当前线程休眠 1s
                    time++
                    var message =Message.obtain()
                    message.arg1 =time                   //设置参数
                    message.what =0x123                  //设置消息标识
                    handler.sendMessage(message)         //发送消息
                }
            }
        }
        endBtn.setOnClickListener {                      //让线程停止循环自动终结
            running =false
        }
    }
}
```

运行结果如图 5-4 所示。

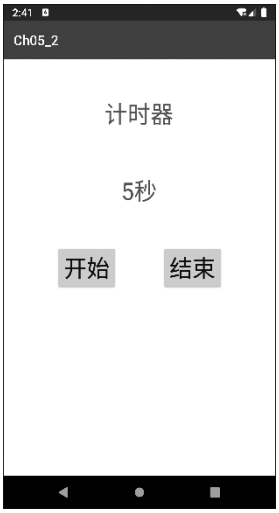


图 5-4　计时器运行结果

在上述的 MainActivity 中并没有创建 Looper 对象，而是通过 Looper.getMainLooper()函数调用 UI 主线程内置的 Looper 对象。在这个计数器的活动中，可直接利用 UI 主线程的 Looper 来控制 MessageQueue。上述代码中，object 表示生成一个匿名对象。Handler 对象在工作线程(自定义的线程)中，发送消息。在主线程中，handler 对象对出列的 Message 的 what 来源标识进行判断，确定其是否是从指定工作线程发出的，最后对这个消息进行处理。在本例中，实现了修改 TextView 控件的文本信息，从而实现了动态计时功能。

5.3　异步任务

在 Android 10 及之前的版本中，当处理耗时的任务时，会通过异步任务 AsyncTask 来实现。异步任务的执行步骤有 3 个步骤。

(1) 在 UI 线程接收事件。

(2) 在非 UI 线程中处理相应事件。

(3) UI 根据处理结果进行刷新。

通过这 3 个步骤，将一些耗时的操作在工作线程中完成，并实现了用户界面更新的处理。下面是一个异步任务的基本结构，代码如下：

```
class SomeTask(): AsyncTask<Unit, Int, Boolean>() {
    /** 任务执行前调用 */
    override fun onPreExecute() {
        …
    }
    /** 这个方法的所有方法会在子线程中被调用,用于处理耗时的任务 */
    override fun doInBackground(vararg params: Unit?): Boolean {
        …
```

```
            return true
        }
        /* * 修改执行的进度 * /
        override fun onProgressUpdate(vararg values: Int?) {
            …
        }
        /* * 完成所有的处理 * /
        override fun onPostExecute(result: Boolean?) {
            …
        }
    }
```

上面代码定义 SomeTask 是 AsyncTask 的子类，要求指定 3 个类型参数＜Unit，Int，Boolean＞，这 3 个类型参数也可以是别的类型。不管怎样，这 3 个类型参数按照出现的顺序，它们分别表示输入的数据类型、进度的数据类型、返回结果的数据类型。结合表 5-1 中 AsyncTask 常见的函数，可以更好地处理异步任务。

表 5-1　AsyncTask 常见函数

函　　数	说　　明
onPreExecute()	在执行实际的后台操作前被 Thread 调用。可以在该方法中做一些准备工作，如在界面上显示一个进度条
onProgressUpdate(Progress …)	在 publishProgress()函数后调用，用来在 UI 上更新进度
onPostExecute(Result)	在 doInBackground()函数执行完成后，被 Thread 调用，后台的计算结果将通过该方法传递到 Thread
doInBackground(Params …)	在 onPreExecute()函数执行后马上执行，该函数运行在后台线程中。这里将主要负责执行那些很耗时的后台计算工作。可以调用 publishProgress()函数来更新实时的任务进度。它是抽象方法，子类必须实现
onCancelled(Result)	在激活 cancel(boolean)函数以及 doInBackground(Object[])函数完成之后，在 Thread 中运行
onCancelled()	默认由它的实现进行激活

AsyncTask 执行流程如图 5-5 所示。首先由主线程调用异步任务的 onPreExecute()函数执行一些准备工作。然后由后台的工作线程调用 doInBackground()函数执行主要耗时的业务。在 doInBackground()函数中会调用 publishProgress()函数对发布并更新当选业务的执行进度。当 doInBackground()函数后台任务执行完毕。UI 主线程调用 onCancelled()函数处理任务取消业务。最后，UI 主线程执行 onPostExecute()函数来处理计算结果，将后台执行的计算结果传递给 UI 主线程。至此，异步任务执行完毕。

例 5-3　使用 AsyncTask 动态显示图片。

(1) 定义在 MainActivity。首先在 MainActivity 对应的布局文件中定义了 FrameLayout，用于包含 Fragment，代码如下：

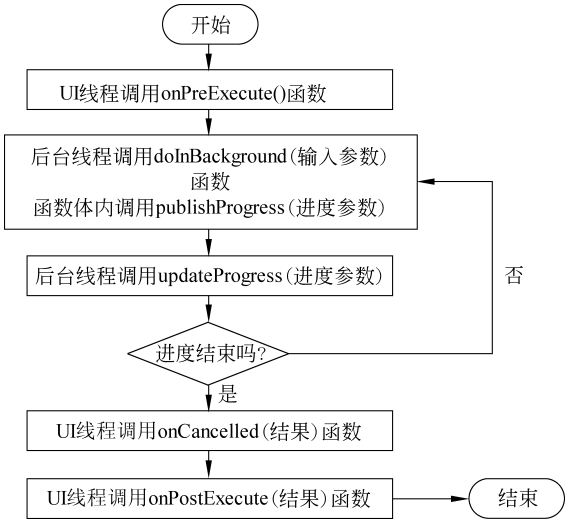


图 5-5　AsyncTask 异步任务执行流程

```
<!--模块 Ch05_03 定义主活动的布局 activity_main.xml -->
<?xml version="1.0" encoding="utf-8"?>
<FrameLayout xmlns: android="http://schemas.android.com/apk/res/android"
    xmlns: app="http://schemas.android.com/apk/res-auto"
    xmlns: tools="http://schemas.android.com/tools"
    android: layout_width="match_parent"
    android: layout_height="match_parent"
    tools: context=".MainActivity"
    android: id="@+id/mainFrag" />
```

MainActivity 用于加载 Fragment,代码如下:

```
//模块 Ch05_03 定义主活动 MainActivity.kt
class MainActivity : AppCompatActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)
        replaceFragment(MainFragment())
    }
    fun replaceFragment(fragment: Fragment){
        supportFragmentManager.beginTransaction().apply{
            replace(R.id.mainFrag,fragment)
            addToBackStack(null)
            commit()
        }
    }
}
```

(2) 定义 MainFragment 和定义异步任务。MainFragment 是嵌入在 MainActivity 中的 Fragment,它对应的布局包括了要显示的 ImageView 和显示进度相关的 View(视图)组件,代码如下:

```
<!--模块 Ch05_03 定义 MainFragment 的布局 fragment_main.xml -->
<?xml version="1.0" encoding="utf-8"?>
<androidx.constraintlayout.widget.ConstraintLayout
    xmlns: android="http://schemas.android.com/apk/res/android"
    xmlns: tools="http://schemas.android.com/tools"
    android: layout_width="match_parent"
    android: layout_height="match_parent"
    xmlns: app="http://schemas.android.com/apk/res-auto"
    android: background="@android: color/black"
    tools: context=".MainFragment">
    <!--定义图像视图设置默认的图片-->
    <ImageView android: id="@+id/imageView"
        android: layout_width="match_parent"
        android: layout_height="match_parent"
        app: layout_constraintEnd_toEndOf="parent"
        app: layout_constraintHorizontal_bias="0.0"
        app: layout_constraintStart_toStartOf="parent"
        app: layout_constraintTop_toTopOf="parent"
        app: srcCompat="@mipmap/scene1" />
    <ProgressBar android: id="@+id/progressBar"
        style="?android: attr/progressBarStyle"
        android: layout_width="wrap_content"
        android: layout_height="wrap_content"
        android: progress="0"
        app: layout_constraintBottom_toBottomOf="@+id/imageView"
        app: layout_constraintEnd_toEndOf="parent"
        app: layout_constraintHorizontal_bias="0.553"
        app: layout_constraintStart_toStartOf="parent"
        app: layout_constraintTop_toTopOf="@+id/imageView"
        app: layout_constraintVertical_bias="1.0" />
    <!--记录进度的文本标签 -->
    <TextView android: id="@+id/progressTxt"
        android: layout_width="wrap_content"
        android: layout_height="wrap_content"
        android: text="TextView" android: textSize="32sp"
        android: textColor="@android: color/white"
        app: layout_constraintBottom_toBottomOf="parent"
        app: layout_constraintEnd_toEndOf="parent"
        app: layout_constraintHorizontal_bias="0.539"
        app: layout_constraintStart_toStartOf="parent"
        app: layout_constraintTop_toTopOf="parent"
        app: layout_constraintVertical_bias="0.022" />
    <!--定义单选按钮组,包括的单选按钮表现为圆形 -->
    <RadioGroup android: id="@+id/group"
        android: layout_width="match_parent"
        android: layout_height="wrap_content"
        android: gravity="center_horizontal"
        android: orientation="horizontal"
        app: layout_constraintBottom_toBottomOf="parent"
```

```
        app: layout_constraintEnd_toEndOf="parent"
        app: layout_constraintStart_toStartOf="parent"
         app: layout_constraintTop_toTopOf="parent"
         app: layout_constraintVertical_bias="0.931">
         <RadioButton android: id="@+id/one"
             android: layout_width="wrap_content"
             android: layout_height="wrap_content"
             android: button="@drawable/radiobtn_style"
             android: checked="true" />
         <RadioButton android: id="@+id/two"
             android: layout_width="wrap_content"
             android: layout_height="wrap_content"
             android: button="@drawable/radiobtn_style" />
         <RadioButton android: id="@+id/three"
             android: layout_width="wrap_content"
             android: layout_height="wrap_content"
             android: button="@drawable/radiobtn_style" />
         <RadioButton android: id="@+id/four"
             android: layout_width="wrap_content"
             android: layout_height="wrap_content"
             android: button="@drawable/radiobtn_style" />
         <RadioButton android: id="@+id/five"
             android: layout_width="wrap_content"
             android: layout_height="wrap_content"
             android: button="@drawable/radiobtn_style" />
    </RadioGroup>
</androidx.constraintlayout.widget.ConstraintLayout>
```

下列的 strings.xml 定义了相关的字符资源和图片资源的整型数组，代码如下：

```
<!--模块 Ch05_03 定义用户界面使用的资源 res/values/strings.xml -->
<resources>
    <string name="app_name">AsyncTask01</string>
    <integer-array name="images">
        <item>@mipmap/scene1</item>
        <item>@mipmap/scene2</item>
        <item>@mipmap/scene3</item>
        <item>@mipmap/scene4</item>
        <item>@mipmap/scene5</item>
    </integer-array>
    <string name="title_hello">主页</string>
    <string name="hello_blank_fragment">欢迎来到 Android 世界!</string>
</resources>
```

在 MainFragment 中加载图片资源，并结合异步任务依次显示，代码如下：