

← Files

main ▾



ParameciaAndroidReverse / 教程 / android开发.md



TimeBeneficiary last week



12616 lines (9297 loc) · 424 KB

Preview

Code

Blame



- [Android 体系结构](#)
- [四大组件](#)
- [活动](#)
- [服务](#)
 - [服务的两种启动方式](#)
- [广播接受者](#)
- [内容提供者](#)
- [创建一个activity之activity_main](#)
- [用户交互_响应用户操作和更新应用界面](#)
- [@Override](#)
- [关于注解](#)
- [如何使用 Intents 在 Android 应用中进行页面跳转](#)
- [如何在 Android 应用中保存和读取数据](#)
- [Fragments](#)
- [Fragment 的交互](#)
- [Fragment 的交互_调用activity当中的函数](#)
- [使用 Fragment 与 Activity 之间的接口通信](#)
- [Fragment 的生命周期](#)

- 广播接受者 (BroadcastReceiver)
- 普通广播和有序广播
- 关于广播的案例_通过监听充电断电广播显示提示 (动态注册)
- 关于广播的案例_时区更改 (静态注册)
- 关于广播的案例_Intent.ACTION_BATTERY_CHANGED显示电量 (动态注册)
- 动态注册的广播再来一个例子
- 隐式广播和显式广播
- 显式_静态注册广播的例子
- 显式_动态注册的案例
- 隐式_静态广播
- 隐式_动态广播
- Activity 间通信广播:
- SecondActivity 通过广播修改 MainActivity 的展示效果
- 一个最简单的有序广播 (动态注册_隐式)
- 一个最简单的有序广播 (静态注册_显式)
- 阻断有序广播
- 有序广播 (修改广播单条信息_修改广播结果)
- 有序广播 (修改广播多条信息_修改广播额外信息)
- setResultExtras 和 setResultData
- 普通广播的数据传递 (Bundle&intent)
- 有序广播 (带最终接受者)
- 有序广播_指定finalReceiver的运行线程 (第四个参数)
- 带权限的广播
 - 权限的申请
 - 一个关于权限的代码案例
- 有序广播 (结果码)
- 有序广播 (最后三个参数)
- 普通广播 (将任务分发到其他线程执行)
- Activity之间的通信startActivityForResult_onActivityResult
- 广播的生命周期
- Activity 的生命周期

- 多线程开发
- 多线程开发 (Thread 类)
- 多线程开发 (Thread 类_Runnable的继承优势)
- 多线程开发 (Thread 类_Runnable的资源共享优势)
 - `runOnUiThread()`
 - 使用 Lambda 表达式简化 Runnable
 - 使用线程池执行 Runnable 任务_FixedThreadPool
 - 使用线程池执行 Runnable 任务_CachedThreadPool
 - 使用线程池执行 Runnable 任务_ScheduledThreadPool
 - 使用线程池执行 Runnable 任务_SingleThreadExecutor
- 多线程_Android 中与其他组件配合使用 Runnable
- Handler_通信
- 在其他的线程当中创建 looper
- `import android.os.Message`
- `import android.os.Message_arg1`和`arg2`
- `import android.os.Message_obj`
- `import android.os.Message_在message当中存放多个数据`
- `import android.os.Message_msg.when`
- Handle的线程之间的通信
- Handler looper 还有 Runnable 这三个东西的关系
 - Handler
 - Looper
 - Runnable
- Handle的任务调度
 - `post(Runnable r)`
 - `postDelayed(Runnable r, long delayMillis)`
 - `postAtTime(Runnable r, long uptimeMillis)`
 - SystemClock 的几种方法
- Handler.Callback
- Lambda表达式
 - 接口当中的默认方法
- `View.post()` 和 `View.postDelayed()`
- Timer 和 TimerTask

- 线程池
 - 线程池_fixedThreadPool (固定线程池)
 - 线程池_SingleThreadExecutor (单线程执行器)
 - 线程池_cachedThreadPool (缓存线程池)
 - 线程池_ScheduledThreadPool (定时任务)
- 自创建线程池
- 泛型 (Generics)
 - 泛型类和泛型接口
 - 泛型方法
 - 类型参数和类型参数约束
- 通配符
- AsyncTask 类
- AsyncTask 的替代方案
- ExecutorService_submit
- Callable 和 Runnable
- ViewModel
- 两个build.gradle
- LiveData
 - observe和observeForever
 - observeForever
- WorkManager
 - WorkManager_链式任务
 - WorkManager_并行任务
 - WorkManager_约束条件
 - WorkManager_输入输出数据
 - WorkManager_Result.failure
 - WorkManager_Result.retry()
 - WorkManager_Result.failure()
 - getWorkInfoById
- JobScheduler

🔗 Android 体系结构

Android系统采用分层架构，从高到低分为四层，分别是

1. 应用程序层 提供一些核心应用程序包，如短信客户端程序、电话拨号程序，Web浏览器、日历、闹钟，安装在上面的程序
2. 应用程序框架层 主要提供构建应用程序时用到的各种API。例如，活动管理器、窗体管理器、内容提供者、资源管理等。
3. 系统库与Android运行层

i. 系统库

是应用程序框架的支撑，主要通过C/C++库来为Android系统提供主要的特性支持。例如包含有SQLite嵌入式数据库引擎，WebKit提供浏览器内核的支持，Media Framework多媒体库，支持多种常用的音频、视频格式的录制和回放

2. Android运行时层



包含核心库和虚拟机，核心库兼容了大多数JAVA语言所需要调用的功能函数，还包含了Android的核心API

4. Linux内核层 作为硬件和软件之间的抽象层，为Android设备上的各种硬件提供底层驱动，如显示驱动、音频驱动、蓝牙驱动等
5. 我们使用Android最主要的就是使用应用，我们所使用的应用，就是应用层
6. 但是开发应用需要用到一些 API 这些 API 就是应用程序框架层
7. API 也不是凭空来的，他也需要支持，比方说数据库，多媒体等等，这个就是 Android 的系统库
8. 与此同时，还有一个 Android 系统的虚拟机，Android 的程序就是运行在这个地方。
9. 系统库也不是凭空来的，还需要 Linux 的内核，这个内核就是和硬件进行交流的驱动程序

🔗 四大组件

🔗 活动

Android 中，Activity是所有程序的根本，所有程序的流程都运行在Activity 之中，Activity可以算是开发者遇到的最频繁，也是Android 当中最基本的模块之一。在Android的程序当中，Activity 一般代表手机屏幕的一屏。如果把手机比作一个浏览器，那么Activity就相当于一个网页。在Activity 当中可以添加一些Button、Check box 等控件。可以看到Activity 概念和网页的概念相当类似。一般一个Android 应用是由多个Activity 组成的。这多个Activity 之间可以进行相互跳转，例如，按下一个Button按钮后，可能会跳转到其他的Activity。和网页跳转稍微有些不一样的是，Activity 之间的跳转有可能返回值，例如，从Activity A 跳转到Activity B，那么当Activity B 运行结束的时候，有可能会给Activity A 一个返回值。这样做在很多时候是相当方便的。当打开一个新的屏幕时，之前一个屏幕会被置为暂停状态，并且压入历史堆栈中。用户可以通过回退操作返回到以前打开过的屏幕。可以选择性的移除一些没有必要保留的屏幕，因为Android会把每个应用的开始到当前的每个屏幕保存在堆栈中

🔗 服务

Service 是android 系统中的一种组件，它跟Activity 的级别差不多，但是他不能自己运行，只能后台运行，并且可以和其他组件进行交互。Service 是没有界面的长生命周期的代码。Service是一种程序，它可以运行很长时间，但是它却没有用户界面。这么说有点枯燥，来看个例子。打开一个音乐播放器的程序，这个时候若想上网了，那么，打开Android浏览器，这个时候虽然已经进入了浏览器这个程序，但是，歌曲播放并没有停止，而是在后台继续一首接着一首的播放。其实这个播放就是由播放音乐的Service进行控制。当然这个播放音乐的Service也可以停止，例如，当播放列表里边的歌曲都结束，或者用户按下了停止音乐播放的快捷键等。Service 可以在和多场合的应用中使用，比如播放多媒体的时候用户启动了其他Activity这个时候程序要在后台继续播放，比如检测SD 卡上文件的变化，再或者在后台记录地理信息位置的改变等等，总之服务嘛，总是藏在后头的。

🔗 服务的两种启动方式

Context.startService(): Service会经历onCreate -> onStart (如果Service还没有运行, 则android先调用onCreate () 然后调用onStart()); 如果Service已经运行, 则只调用onStart (), 所以一个Service的onStart方法可能会重复调用多次)

StopService 的时候直接 onDestroy 如果是调用者自己直接退出而没有调用 StopService 的话, Service会一直在后台运行。该 Service 的调用者再启动起来后可以通过 stopService 关闭 Service 。注意, 多次调用 Context.startService() 不会嵌套 (即使会有相应的onStart()方法被调用), 所以无论同一个服务被启动了多少次, 一旦调用 Context.stopService() 或者 StopSelf(), 他都会被停止。补充说明: 传递给StartService (0的Intent对象会传递给onStart()方法。调用顺序为: onCreate --> onStart (可多次调用) --> onDestroy

Context.bindService(): Service会经历onCreate()-->onBind(), onBind将返回给客户端一个IBind接口实例, IBind允许客户端回调服务的方法, 比如得到 Service运行的状态或其他操作。这个时候把调用者 (Context, 例如 Activity) 会和Service绑定在一起, Context退出了, Service就会调用 onUnbind --> onDestroyed相应退出, 所谓绑定在一起就共存亡了。

🔗 广播接受者

在Android 中, Broadcast是一种广泛运用的在应用程序之间传输信息的机制。而BroadcastReceiver 是对发送出来的Broadcast进行过滤接受并响应的一类组件。可以使用BroadcastReceiver 来让应用对一个外部的的事件做出响应。这是非常有意思的, 例如, 当电话呼入这个外部事件到来的时候, 可以利用BroadcastReceiver 进行处理。例如, 当下载一个程序成功完成的时候, 仍然可以利用BroadcastReceiver 进行处理。BroadcastReceiver不能生成UI, 也就是说对于用户来说不是透明的, 用户是看不到的。BroadcastReceiver通过 NotificationManager 来通知用户这些事情发生了。BroadcastReceiver 既可以在AndroidManifest.xml 中注册, 也可以在运行时的代码中使用 Context.registerReceiver ()进行注册。只要是注册了, 当事件来临的时候, 即使程序没有启动, 系统也在需要的时候启动程序。各种应用还可以通过使用 Context.sendBroadcast() 将它们自己的Intent Broadcasts广播给其他应用程序。

🔗 内容提供者

Content Provider 是Android提供的第三方应用数据的访问方案。在Android中，对数据的保护是很严密的，除了放在SD卡中的数据，一个应用所持有的数据库、文件等内容，都是不允许其他直接访问的。Android当然不会真的把每个应用都做成一座孤岛，它为所有应用都准备了一扇窗，这就是Content Provider。应用想对外提供的数据，可以通过派生Content Provider类，封装成一枚Content Provider，每个Content Provider都用一个uri作为独立的标识，形如：content://com.xxxxx。所有东西看着像REST的样子，但实际上，它比REST 更为灵活。和REST类似，uri也可以有两种类型，一种是带id的，另一种是列表的，但实现者不需要按照这个模式来做，给id的uri也可以返回列表类型的数据，只要调用者明白，就无妨，不用苛求所谓的REST。

🔗 创建一个activity之activity_main

路径 app/res/layout

注意这个布局文件的文件名，只能够小写字母组成的

```
<?xml version="1.0" encoding="utf-8"?>
<androidx.constraintlayout.widget.ConstraintLayout xmlns:android="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">

    <TextView
        android:id="@+id/textView"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Hello, World!"
        android:textSize="24sp"
        app:layout_constraintBottom_toBottomOf="parent"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintTop_toTopOf="parent" />

</androidx.constraintlayout.widget.ConstraintLayout>
```


详细解读这个文件

`<?xml version="1.0" encoding="utf-8"?>` 表示的是，xml 的版本还有就是编码格式

```
<androidx.constraintlayout.widget.ConstraintLayout xmlns:android="http://schemas.android.com/apk/res-auto"
    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">
```

`ConstraintLayout` 表示的是 容器

`xmlns` 是命名空间，让我们可以使用 `Android`、`app`和`tools`命名空间的属性

`layout_width` 和 `layout_height` 属性设置为 `match_parent`，这意味着 `ConstraintLayout` 将填充其父容器的整个空间

`tools:context` 属性指定了与此布局相关联的Activity类

翻译

`ConstraintLayout` 约束_布局

widget 小部件

```
<TextView
    android:id="@+id/textView"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="Hello, World!"
    android:textSize="24sp"
    app:layout_constraintBottom_toBottomOf="parent"
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintTop_toTopOf="parent" />
```

- `android:id="@+id/textView"` 为 `TextView` 指定了一个唯一的ID，我们可以在代码中引用它
- `android:layout_width="wrap_content"` 和 `android:layout_height="wrap_content"` 表示 `TextView` 的宽度和高度将根据其内容自适应。
- `android:text="Hello, World!"` 设置 `TextView` 显示的文本为 "Hello, World!"
- `android:textSize="24sp"` 设置文本的大小为24sp（尺寸独立像素，适合用于设置字体大小）

这个主要是设置 `TextView` 的自己的属性

- `app:layout_constraintBottom_toBottomOf="parent"`：将 `TextView` 的底部与父容器的底部对齐。
- `app:layout_constraintEnd_toEndOf="parent"`：将 `TextView` 的末端（在英语环境中为右侧）与父容器的末端对齐。
- `app:layout_constraintStart_toStartOf="parent"`：将 `TextView` 的起始端（在英语环境中为左侧）与父容器的起始端对齐。
- `app:layout_constraintTop_toTopOf="parent"`：将 `TextView` 的顶部与父容器的顶部对齐

定义了 `TextView` 相对于其父容器 `ConstraintLayout` 的约束关系

翻译

wrap 包裹

wrap_content 自适应内容

`</androidx.constraintlayout.widget.ConstraintLayout>` 这是根元素 `ConstraintLayout` 的结束标签

父容器（也称为父视图或父元素）是一个包含其他视图（称为子视图或子元素）的视图。父容器负责组织、定位和管理其子视图。在我们刚才讨论的布局示例中，`ConstraintLayout` 是一个父容器，它包含了一个 `TextView` 子视图

父容器可以是任何类型的布局视图，例如 `LinearLayout`、`RelativeLayout` 或 `ConstraintLayout` 等。每种父容器都有其特定的布局规则和属性，用于控制子视图的位置和行为。

```
package com.example.helloworldapp;

import androidx.appcompat.app.AppCompatActivity;
import android.os.Bundle;

public class MainActivity extends AppCompatActivity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
    }
}
```

🔗 用户交互_响应用户操作和更新应用界面

```
<Button
    android:id="@id/my_button"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="一个按钮"
    tools:layout_editor_absoluteX="16dp"
    tools:layout_editor_absoluteY="149dp"
    tools:ignore="MissingConstraints" />
```

`tools:ignore="MissingConstraints"` 添加这个属性是因为，在这个按钮当中，有一个属性是比较重要的但是没有添加，`ignore="MissingConstraints"` 忽略缺少约束条件

```
public class MainActivity extends AppCompatActivity {  
    @Override  
    protected void onCreate(Bundle savedInstanceState) {  
        super.onCreate(savedInstanceState);  
        setContentView(R.layout.activity_main);  
  
        Button myButton = findViewById(R.id.my_button);  
        myButton.setOnClickListener(new View.OnClickListener()  
        {  
            @Override  
            public void onClick(View v) {  
                // 在这里处理点击事件  
                Toast.makeText(MainActivity.this, "Button c  
            }  
        });  
    }  
}
```

这里面主要是三个函数

`OnClickListener()` 他的作用是给 button 做一个监听器，他的作用是监听 button 的按键事件

`setOnClickListener()` 他的作用就是将我们的 监听器 附加到我们的按键，他接受一个参数就是 `OnClickListener()` 这个接口实现的类

`onClick()` 这个就是点击了 button 之后的一些操作，比方说在里面打开打一个 activity 或者是显示一个提示等等 ...

这个代码，我们能够知道，先是从布局信息当中找到我们想要操作的控件 (by id) 这个按键就有一个对象

对象.`setOnClickListener` 就能够关联上 `OnClickListener`

`OnClickListener` 就能够执行 `onClick` 里面的代码

如果将这个拆分来写可以是这个样子



```
public class MainActivity extends AppCompatActivity {
    // 创建一个实现了OnClickListener接口的类
    private class MyButtonClickListener implements View.OnC
        @Override
        public void onClick(View v) {
            // 在这里处理点击事件
            Toast.makeText(MainActivity.this, "Button click
        }
    }

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        // 找到按钮
        Button myButton = findViewById(R.id.my_button);

        // 创建一个MyButtonClickListener实例
        MyButtonClickListener listener = new MyButtonClickL

        // 使用setOnClickListener方法将监听器附加到按钮上
        myButton.setOnClickListener(listener);
    }
}
```

1. `Toast.makeText()`: 这是一个静态方法，用于创建一个 `Toast` 对象。
`Toast` 是 Android 中一种用于显示短暂消息的小型弹出窗口。它在屏幕上持续显示几秒钟，然后自动消失，不干扰用户操作。
2. `MainActivity.this`: 这是一个对当前 `MainActivity` 实例的引用。因为我们在 `MyButtonClickListener` 类中使用了这段代码，所以需要明确地引用外部类（即 `MainActivity`）的实例。`MainActivity.this` 表示我们希望使用 `MainActivity` 的当前实例作为上下文（Context）参数。上下文是一个抽象类，代表应用程序环境的信息，通常用于访问与应用程序相关的资源和操作。
3. `Toast.LENGTH_SHORT`: 这是一个预定义的常量，表示 `Toast` 消息显示的持续时间。`Toast.LENGTH_SHORT` 表示 `Toast` 消息显示的时间较短。另一个可选值是 `Toast.LENGTH_LONG`，表示 `Toast` 消息显示的时间较长。
4. `show()`: 这是一个 `Toast` 对象的方法，用于将 `Toast` 消息显示在屏幕上。在使用 `makeText()` 创建 `Toast` 对象后，必须调用 `show()` 方法才

能将其显示给用户。



```
package com.fu.tt;
import android.widget.Button;
import android.view.View;

import androidx.appcompat.app.AppCompatActivity;

import android.os.Bundle;
import android.widget.Toast;

public class MainActivity extends AppCompatActivity {
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        Button firstButton = findViewById(R.id.first_button);
        firstButton.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View v) {
                Toast.makeText(MainActivity.this, "o1111111"
            }
        });
        Toast.makeText(this, "o1111111", Toast.LENGTH_LONG).s
    }
}
```

这个代码，就是使用的匿名内部类的方式进行的，也是内部类，所以要通过 `MainActivity.this` 的方式对当前实例 context 引用

如果不是在里面，就可以直接使用 `this` 进行引用，比方说这样

```
Toast.makeText(this, "o1111111", Toast.LENGTH_LONG).show();
```

下面两个代码增加，对 `MainAcitivity.this` 的理解

```
public class MainActivity extends AppCompatActivity implements   
  
    @Override  
    protected void onCreate(Bundle savedInstanceState) {  
        super.onCreate(savedInstanceState);  
        setContentView(R.layout.activity_main);  
  
        Button firstButton = findViewById(R.id.first_button);  
        firstButton.setOnClickListener(this);  
    }  
  
    @Override  
    public void onClick(View v) {  
        Toast.makeText(this, "o1111111", Toast.LENGTH_LONG)  
    }  
}
```



```
package com.fu.tt;
import android.widget.Button;
import android.view.View;

import androidx.appcompat.app.AppCompatActivity;

import android.os.Bundle;
import android.widget.Toast;

public class MainActivity extends AppCompatActivity {
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        Button firstButton = findViewById(R.id.first_button);
        firstButton.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View v) {
                ##
                Toast.makeText(v.getContext(), "o1111111", To
                ##
            }
        });

        Toast.makeText(this, "? ? 、 、 、 ", Toast.LENGTH_LONG).s
    }
}
```

🔗 @Override

@Override 是一个 Java 注解，用于表示一个方法是重写了父类或接口中的方法。当你在一个子类或实现了某个接口的类中编写一个方法时，如果该方法是重写父类或接口中已有的方法，建议使用 `@Override` 注解

🔗 关于注解

android 当中有一些直接调用的注解，比方说 @Override 他表示这个函数是一个重写

- @Override

注解用于标记一个方法，该方法覆盖了其父类中的方法，如果你的覆盖了一个，父类当中不存在的方法，就会报错

```
@Override
public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);
}
```

- @Nullable 和 @NotNull

@Nullable 和 @NotNull 注解用于标记一个方法的参数、返回值或字段的可空性。如果你使用 @Nullable 注解标记了一个参数、返回值或字段，那么它可以为 null。如果你使用 @NotNull 注解标记了一个参数、返回值或字段，那么它不可以为 null。

```
@Nullable
public void setName(@Nullable String name) {
    this.name = name;
}

@NotNull
public String getName() {
    return name;
}

@NotNull
private String name;
```

- @SuppressWarnings

解用于禁用 Lint 检查器中的某些警告或错误。如果你使用 `@SuppressWarnings` 注解标记了一个类、方法或字段，那么 Lint 检查器将不会对该类、方法或字段进行检查

```
@SuppressWarnings("SetTextI18n")
private void initUI() {
    // Set the text of the button
    Button firstButton = findViewById(R.id.first_button);
    firstButton.setText("Click me!");
}
```



Lint 检查器是 Android 开发工具中的一个静态代码分析工具，它可以帮助开发者发现代码中的潜在问题

`@SuppressWarnings("SetTextI18n")` 是一个注解，它用于禁止 Lint 检查器中的 `SetTextI18n` 警告。这个警告是关于 Android 应用程序中的国际化问题，它表示应该使用字符串资源来设置 UI 元素

的文本，而不是直接在代码中设置文本。

类似的还有很多

```
@SuppressWarnings("UnusedCode")
```

该警告表示应用程序中存在未使用的代码，例如未使用的方法、变量等等

```
@SuppressWarnings("UnusedResources")
```

警告表示应用程序中存在未使用的资源，例如布局文件、字符串资源等等

翻译

Suppress 本身就是抑制，压制的意思，还通常指的是针对错误的抑制，让一些错误不出现。

🔗 如何使用 Intents 在 Android 应用中进行页面跳转

Intents 是 Android 系统中用于描述应用间或应用内组件间通信的一种机制，Intents 可以用于启动 Activity、服务或发送广播等

以下是一个简单的示例，我们将创建两个 Activity，MainActivity 和 SecondActivity，并在 MainActivity 中添加一个按钮，用于跳转到 SecondActivity

- 首先在 app/res/layout 当中新建一个 secondactivity 的布局界面，activity_second.xml

```
<?xml version="1.0" encoding="utf-8"?>
<androidx.constraintlayout.widget.ConstraintLayout xmlns:android="http://schemas.android.com/apk/res-auto"
    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=".SecondActivity">

    <TextView
        android:id="@+id/textView2"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Welcome to Second Activity!"
        android:textSize="24sp"
        app:layout_constraintBottom_toBottomOf="parent"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintTop_toTopOf="parent" />

</androidx.constraintlayout.widget.ConstraintLayout>
```

- 在代码当中新建一个 SecondActivity.java 的类文件

```
package com.fu.tt;

import androidx.appcompat.app.AppCompatActivity;

import android.os.Bundle;

public class SecondActivity extends AppCompatActivity{
    @Override
    protected void onCreate(Bundle savedInstanceState)
    {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_second);
    }
}
```

继承自 `AppCompatActivity`，并在 `onCreate` 方法中设置 `activity_second.xml` 为其布局

- 还需要将这个 activity 的这个类注册到 `androidmanifest.xml` 当中让系统知道

```
<manifest ...>
    ...
    <application ...>
        ...
        <activity android:name=".SecondActivity" />
    </application>
</manifest>
```

- 添加按钮，然后跳转

需要在 `MainActivity` 的布局文件 `activity_main.xml` 中添加一个按钮，用于跳转到 `SecondActivity`。添加如下代码：

```
<Button
    android:id="@+id/go_to_second_button"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="Go to Second Activity"
    app:layout_constraintBottom_toTopOf="@+id/textView"
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintTop_toTopOf="parent"
    app:layout_constraintVertical_bias="0.764" />
```

- 最后，在 `MainActivity` 类中，为按钮设置一个 `OnClickListener`，并使用 `Intent` 跳转到 `SecondActivity`

```
Button goToSecondActivityButton = findViewById(R.id
goToSecondActivityButton.setOnClickListener(new Vie
    @Override
    public void onClick(View v) {
        Intent intent = new Intent(MainActivity.this
            startActivity(intent);
    }
});
```

1. `new Intent(MainActivity.this, SecondActivity.class)`: 这是创建一个新的 `Intent` 对象的构造函数。它需要两个参数:
2. `MainActivity.this`: 这是当前 `Activity` 的上下文 (Context)。因为我们在 `MainActivity` 的内部类 (`OnClickListener`) 中创建了 `Intent`，所以需要使用 `MainActivity.this` 来引用外部类的实例。
3. `SecondActivity.class`: 这是我们要跳转到的目标 `Activity` 的类名。这里我们传递了 `SecondActivity` 的类对象 (`SecondActivity.class`)，告诉 `Intent` 我们希望启动这个 `Activity`。

🔗 如何在 Android 应用中保存和读取数据

翻译

Share_Preferences 共享偏好设置

可以在用户更改设置或输入内容时调用 `saveData` 方法，并在 `Activity` 启动或恢复时调用 `loadData` 方法来显示先前保存的数据

```
private void saveData(String key, String value) {  
    SharedPreferences sharedPreferences = getSharedPreferences(  
    SharedPreferences.Editor editor = sharedPreferences.edi  
    editor.putString(key, value);  
    editor.apply();  
}
```

```
private String loadData(String key) {  
    SharedPreferences sharedPreferences = getSharedPref  
    return sharedPreferences.getString(key, "default_va  
}
```

这两个函数，一个写入，一个读取

`SharedPreferences`，这是一种轻量级的数据存储方式，适用于存储少量简单的数据

- `getSharedPreferences` 方法获取 `SharedPreferences` 实例
- `SharedPreferences.Editor` 对象并使用 `putString` 方法将键值对添加到编辑器中
- 最后，调用 `apply` 方法将更改保存到 `SharedPreferences` 文件

可以在用户更改设置或输入内容时调用 `saveData` 方法，并在 `Activity` 启动或恢复时调用 `loadData` 方法来显示先前保存的数据。直到卸载软件或者是，手动删除，这些数据才会消失。

```
SharedPreferences sharedPreferences =  
getSharedPreferences("MySharedPreferences", MODE_PRIVATE);
```

 这里是打开/创建一个 SharedPreferences 的对象，并且设置这个文件的名字 MySharedPreferences，和他的访问属性 MODE_PRIVATE 这个表示，只能够本程序的代码能够访问这个存储的数据。进程间的通信多是使用

```
ContentProvider
```

```
SharedPreferences.Editor editor = sharedPreferences.edit();
```

 创建 SharedPreferences 对象的编辑器

```
editor.putString(key, value);
```

 就是写入键值对

还有 `putInt` `putBoolean` ...

```
editor.apply();
```

保存写入数据

当你调用 `apply()` 方法时，应用程序可以继续执行其他任务，而不需要等待数据保存操作完成。由于 `apply()` 方法是异步的，通常推荐使用它而不是 `commit()` 方法，后者是同步的并可能阻塞当前线程

我们可以看到，在访问的时候，和创建的时候，都设置了属性但是实际上，大多数的情况，MODE_PRIVATE，直接设置这个属性就行，无论是在读取的时候还是在

```
sharedPreferences.getString(key, "default_value");
```

 这个函数在执行的时候，如果没有找到对应的键值对，就会弹出 default_value（这个是自定的）

Fragments

翻译

Fragments 分段，块

在大屏的手机上，特别是平板上，我们竖着拿平板，和横着拿平板，我们的显示布局是不一样的

Fragments的作用就是将一个 activity 的界面，分为好几段，方便管理，他依附于 activity 而存在

1. **灵活性**：Fragments 可以在不同的 Activity 之间重用，这意味着可以在不同的屏幕和场景下共享相同的 UI 逻辑。
 2. **适应性**：Fragments 支持动态调整布局，可以根据屏幕尺寸和方向更好地适应不同的设备。例如，在手机上，您可以使用一个 Fragment 显示列表，然后在用户选择列表项时，启动另一个 Activity 并使用另一个 Fragment 显示详细信息。而在平板电脑上，您可以在同一个 Activity 中并排显示这两个 Fragment。
 3. **生命周期管理**：Fragments 有自己的生命周期，与宿主 Activity 相互关联但又独立于它。这使得在处理与 Fragment 相关的操作时更加灵活，例如在 Activity 运行时动态添加、删除或替换 Fragment。Fragments 的生命周期管理也有助于实现更高效的资源管理。
- 首先新建一个 fragments 的类



```
package com.fu.tt;

import androidx.fragment.app.Fragment;
import android.os.Bundle;
import android.view.LayoutInflater;
import android.view.ViewGroup;

import androidx.annotation.NonNull;
import androidx.annotation.Nullable;
import android.view.View;

public class OneFragment extends Fragment {

    @Override
    public void onCreate(@Nullable Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        // 在这里执行您需要在视图创建之前的操作，例如初始化成员变量或
        // 但是大部分的时候是不需要 Oncreate 函数的，因为大部分的初
    }

    @Nullable
    @Override
    public View onCreateView(@NonNull LayoutInflater inflater
        return inflater.inflate(R.layout.onefragment, conta
    }
}
```

`LayoutInflater inflater`：用于将 XML 布局文件转换为实际的 View 对象，`onCreateView` 这个函数一定会返回一个 View 类型的对象，所以需要转换一下

`inflater.inflate` 里面有三个参数

`R.layout.onefragment` 指定我的 fragment 的布局内容

`ViewGroup container`：承载 Fragment 视图的父容器。在将 Fragment 添加到 Activity 时，这通常是一个 `FrameLayout` 或其他容器，每个 Fragment 都必须要有有一个父容器，即它的视图必须要嵌入到另一个视图中。这个父容器在 Fragment 被添加到 Activity 中时被指定

`Bundle savedInstanceState`：如果 Fragment 在之前的配置更改（如屏幕旋转）中被销毁并重新创建，此参数将包含之前保存的 Fragment 状态。


```
return inflater.inflate(R.layout.fragment_example, container, false);
```

这行代码使用传入的 `LayoutInflater` 对象将 XML 布局文件（这里是 `R.layout.fragment_example`）转换为一个 `View` 对象。这个 `View` 对象随后被添加到 `ViewGroup container`

重点关注一下这个 `container`，他的类型是 `ViewGroup`

他就是，`Activity_main.xml` 当中添加的容器

还有就是关注一下 `false` 这个参数（通常情况之下，只需要，默认 `false` 就行了）

```
public View onCreateView(@NonNull LayoutInflater inflater,   
    if (container != null) {  
        inflater.inflate(R.layout.onefragment, container, t  
        return container.findViewById(R.id.onefragment_root  
    } else {  
        return inflater.inflate(R.layout.onefragment, conta  
    }  
}
```

注意添加了一个 `onefragment_root_view` 这个是 `OneFragment.xml` 的根布局的 `id`

就是如果你的这个第三个参数设置为 `true`，那么还需要添加一步，就是根视图的引用，但是设置为 `false` 这个步骤是自动的，

给 `OneFragment.xml`

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

    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Hello from One Fragment!"
        android:textSize="24sp" />
</LinearLayout>
```



OneFragment.xml

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

    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Hello from One Fragment!"
        android:textSize="24sp" />
</LinearLayout>
```



Activity_main.xml 当中添加

```
<androidx.fragment.app.FragmentContainerView
    android:id="@+id/fragment_container"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    app:layout_constraintBottom_toBottomOf="parent"
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintTop_toTopOf="parent" />
```



```
public class MainActivity extends AppCompatActivity {
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        OneFragment exampleFragment = new OneFragment();
        FragmentManager fragmentManager = getSupportFragmentManager()
        FragmentTransaction fragmentTransaction = fragmentManager
        fragmentTransaction.add(R.id.fragment_container, ex
        fragmentTransaction.commit();
    }
}
```



1. `OneFragment exampleFragment = new OneFragment();` 创建一个 `OneFragment` 类型的对象 `exampleFragment`，它是我们要添加到 Activity 中的 Fragment 实例。
2. `FragmentManager fragmentManager = getSupportFragmentManager();` 通过调用 `getSupportFragmentManager()` 方法来获取与当前 Activity 关联的 `FragmentManager` 实例。`FragmentManager` 是一个类，用于管理 Activity 中的 Fragments。
3. `FragmentTransaction fragmentTransaction = fragmentManager.beginTransaction();` 通过 `FragmentManager` 的 `beginTransaction()` 方法，开启一个新的 Fragment 事务。`FragmentTransaction` 类用于执行与 Fragments 相关的一系列操作（如添加、删除、替换等）。
4. `fragmentTransaction.add(R.id.fragment_container,`

`exampleFragment`); 使用 `add()` 方法将 `exampleFragment` 添加到 `fragment_container` 中。`R.id.fragment_container` 是一个容器（通常是一个 `FrameLayout`），用于承载 `Fragments`。这行代码的作用是将 `exampleFragment` 放入指定的容器中。

5. `fragmentTransaction.commit()`; 调用 `commit()` 方法以提交 `Fragment` 事务。这个方法告诉 `FragmentManager`，我们已完成所有的 `Fragment` 操作并希望执行这些操作。`FragmentManager` 会在合适的时机将 `Fragment` 添加到 `Activity` 中。

翻译

Transaction

事务通常由一系列的操作组成，如果其中任何一个操作失败，则整个事务都将回滚，即撤销之前的所有操作。

🔗 Fragment 的交互

- 首先在 `fragment` 里面添加按钮

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

    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Hello from One Fragment!"
        android:textSize="24sp" />

    <Button
        android:id="@+id/button_fragment"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Click me!" />

</LinearLayout>
```

Linear_Layout 线性布局

- class OneFragment

```
package com.fu.tt;

import androidx.fragment.app.Fragment;
import android.os.Bundle;
import android.view.LayoutInflater;
import android.view.ViewGroup;

import androidx.annotation.NonNull;
import androidx.annotation.Nullable;
import android.view.View;
import android.widget.Button;
import android.widget.Toast;

public class OneFragment extends Fragment {

    @Override
    public void onCreate(@Nullable Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        // 在这里执行您需要在视图创建之前的操作，例如初始化成员变量或
    }

    @Nullable
    @Override
    public View onCreateView(@NonNull LayoutInflater inflater,
        View view = inflater.inflate(R.layout.onefragment,
        Button button = view.findViewById(R.id.button_fragm
        button.setOnClickListener(new View.OnClickListener()
            @Override
            public void onClick(View v) {
                Toast.makeText(getActivity(), "我的fragment",
            }
        });
        return view;
    }
}
```

- MainActivity.class

```
package com.fu.tt;
import android.content.Intent;
import android.widget.Button;
import android.view.View;

import androidx.appcompat.app.AppCompatActivity;
import androidx.fragment.app.FragmentManager;
import androidx.fragment.app.FragmentTransaction;

import android.os.Bundle;
import android.widget.Toast;
import android.content.SharedPreferences;

public class MainActivity extends AppCompatActivity {
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        OneFragment exampleFragment = new OneFragment();
        FragmentManager fragmentManager = getSupportFragmentManager()
        FragmentTransaction fragmentTransaction = fragmentManager
        fragmentTransaction.add(R.id.fragment_container, ex
        fragmentTransaction.commit();
    }
}
}
```

需要注意的是，我们知道

```
button.setOnClickListener(new View.OnClickListener(
    @Override
    public void onClick(View v) {
        Toast.makeText(getActivity(), "我的fragment",
    }
});
```

这一段的代码，我们是通过匿名的内部类的方式实现的

```
Toast.makeText(getActivity(),"我的  
fragment",Toast.LENGTH_SHORT).show();
```

所以我们要通过这样的方式，指定 context

在前面的MainActivity我们有三种方式进行指定

1. MainActivity.this

2. v.getContext()

或者是第三种，通过直接继承的方式

```
public class MainActivity extends AppCompatActivity implements  
  
    @Override  
    public void onClick(View v)  
    {  
        Toast.makeText(this, "o1111111", Toast.LENGTH_L  
    }  
  
    @Override  
    protected void onCreate(Bundle savedInstanceState)  
    {  
        super.onCreate(savedInstanceState);  
        setContentView(R.layout.activity_main);  
  
        Button firstButton = findViewById(R.id.first_bu  
        firstButton.setOnClickListener(this);  
    }
```

那在这个地方能够使用 OneFragment.this 嘛，能够使用，第三种方式嘛？
(可以使用 v.getContext() , getActivity ())

都不行，这是因为 Toast.makeText() 方法的第一个参数需要一个 Context 对象，而不是 OneFragment 类的实例。由于 OneFragment 类扩展了 Fragment 类，Fragment 不是 Context 的子类

而由于 MainActivity 类继承自 AppCompatActivity ，它本身就是一个 Context 类的子类。因此，在这种情况下，您可以将 MainActivity.this 作为 Toast.makeText() 方法的第一个参数

那在 MainActivity 当中能够使用 `getActivity()` 嘛？

不能因为在 MainActivity 类中，您不能使用 `getActivity()` 方法来获取上下文（Context 对象）。`getActivity()` 方法只能在 Fragment 类中使用，因为它用于获取与 Fragment 关联的 Activity 实例。

🔗 Fragment 的交互_调用activity当中的函数

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

    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Hello from One Fragment!"
        android:textSize="24sp" />

    <Button
        android:id="@+id/button_fragment"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Click me!" />

</LinearLayout>
```

```
public class MainActivity extends AppCompatActivity {
    // ...

    public void onFragmentButtonClicked() {
        Toast.makeText(this, "Button clicked in Fragment!",
    }
}
```

```
public class ExampleFragment extends Fragment {  
    @Nullable  
    @Override  
    public View onCreateView(@NonNull LayoutInflater inflater,  
        View view = inflater.inflate(R.layout.fragment_exam  
  
        Button button = view.findViewById(R.id.button_fragm  
        button.setOnClickListener(new View.OnClickListener(  
            @Override  
            public void onClick(View v) {  
                ##  
                if (getActivity() instanceof MainActivity)  
                    ((MainActivity) getActivity()).onFragme  
                }  
                ##  
            }  
        });  
        return view;  
    }  
}
```



🔗 使用 Fragment 与 Activity 之间的接口通信

假设我们有一个名为 `MainActivity` 的 Activity，它包含一个名为 `ExampleFragment` 的 Fragment。我们希望在 Fragment 中的按钮被点击时，能够在 Activity 中执行一些操作。

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

    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Hello from One Fragment!"
        android:textSize="24sp" />

    <Button
        android:id="@+id/button_fragment"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Click me!" />

</LinearLayout>
```

```
public class MainActivity extends AppCompatActivity implements OnFragmentButtonClickListener {
    // ...

    public void onFragmentButtonClicked() {
        Toast.makeText(this, "在 Fragment 中的按钮被点击时，在
    }
}
```

一个名为 `listener` 的变量，以及重写 `onAttach` 和 `onDetach` 方法

在 `onAttach` 方法中检查是否实现了 `OnFragmentButtonClickListener` 接口，如果实现了，我们将其赋值给 `listener`。当 Fragment 脱离 Activity 时，我们将 `listener` 置为 null，以避免内存泄漏。



```

public class ExampleFragment extends Fragment {
    public interface OnFragmentButtonClickListener {
        void onFragmentButtonClicked();
    }

    private OnFragmentButtonClickListener listener;


    @Override
    public void onAttach(@NonNull Context context) {
        super.onAttach(context);
        if (context instanceof OnFragmentButtonClickListene
            listener = (OnFragmentButtonClickListener) cont
        } else {
            throw new RuntimeException(context.toString() +
        }
    }

    @Override
    public void onDetach() {
        super.onDetach();
        listener = null;
    }

    // ...
}

```

最后，修改 `ExampleFragment` 类中的按钮点击监听器，以调用 `listener.onFragmentButtonClicked()`：



```

package com.fu.tt;

import androidx.fragment.app.Fragment;
import android.os.Bundle;
import android.view.LayoutInflater;
import android.view.ViewGroup;

import androidx.annotation.NonNull;
import androidx.annotation.Nullable;
import android.view.View;
import android.widget.Button;
import android.widget.Toast;

```

```

import android.content.Context;

public class OneFragment extends Fragment {
    private OnFragmentButtonClickListener listener;

    public interface OnFragmentButtonClickListener {
        void onFragmentButtonClicked();
    }

    @Override
    public void onCreate(@Nullable Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        // 在这里执行您需要在视图创建之前的操作，例如初始化成员变量或
    }

    @Nullable
    @Override
    public View onCreateView(@NonNull LayoutInflater inflater,
        View view = inflater.inflate(R.layout.onefragment,
        Button button = view.findViewById(R.id.button_fragm
        button.setOnClickListener(new View.OnClickListener(
            @Override
            public void onClick(View v) {
                if (listener != null) {
                    listener.onFragmentButtonClicked();
                }
            }
        ));
    return view;
}

@Override
public void onAttach(@NonNull Context context) {
    super.onAttach(context);
    if (context instanceof OnFragmentButtonClickListene
        listener = (OnFragmentButtonClickListener) cont
    } else {
        throw new RuntimeException(context.toString() +
    }
}

@Override
public void onDetach() {
    super.onDetach();
    listener = null;
}

```

```
}  
}
```

需要注意的是，context 其实就是所属的 activity 的实例（MainActivity.this）

```
public void onAttach(@NonNull Context context) {  
    super.onAttach(context);  
    if (context instanceof OnFragmentButtonClickListene  
        listener = (OnFragmentButtonClickListener) cont  
    } else {  
        throw new RuntimeException(context.toString() +  
    }  
}
```

🔗 Fragment 的生命周期

1. **onAttach():** 当 Fragment 与 Activity 关联时，此方法被调用。在这里，您可以获取宿主 Activity 的引用并执行与之相关的初始化操作。
2. **onCreate():** 当 Fragment 的实例被创建时，此方法被调用。您可以在这里进行非视图相关的初始化操作，例如初始化变量。
3. **onCreateView():** 当 Fragment 需要创建其视图时，此方法被调用。您需要在这里加载 Fragment 的布局并返回视图对象。
4. **onViewCreated():** 当 Fragment 的视图被创建后，此方法被调用。在这里，您可以执行与视图相关的初始化操作，如设置监听器和更新视图内容。
5. **onActivityCreated():** 当宿主 Activity 的 `onCreate()` 方法执行完成后，此方法被调用。您可以在这里进行与 Activity 相关的初始化操作。
6. **onStart():** 当 Fragment 变得可见时，此方法被调用。您可以在这里执行与界面展示相关的操作。
7. **onResume():** 当 Fragment 变得可以与用户交互时，此方法被调用。这是 Fragment 生命周期中的最后一个阶段。

从这个阶段开始，Fragment 可能会进入暂停状态。当这种情况发生时，以下方法将被调用：

1. **onPause():** 当 Fragment 不再与用户交互时，此方法被调用。您可以在这里保存数据和执行其他与暂停状态相关的操作。
2. **onStop():** 当 Fragment 变得不可见时，此方法被调用。您可以在这里执行与界面隐藏相关的操作。

接下来，Fragment 可能会重新进入活动状态，或者它可能会被销毁。如果 Fragment 被销毁，以下方法将被调用：

1. **onDestroyView():** 当 Fragment 的视图被销毁时，此方法被调用。您可以在这里释放与视图相关的资源。
2. **onDestroy():** 当 Fragment 的实例被销毁时，此方法被调用。您可以在这里释放与 Fragment 相关的资源。
3. **onDetach():** 当 Fragment 与宿主 Activity 分离时，此方法被调用。您可以在这里执行与解除关联相关的操作。
4. **onAttach():** Fragment 和 Activity 建立关联时调用。可以获取 Activity 的引用。
5. **onCreate():** Fragment 创建时调用。进行非视图相关的初始化。
6. **onCreateView():** Fragment 需要创建视图时调用。加载布局并返回视图对象。
7. **onViewCreated():** 视图创建后调用。执行视图相关的初始化操作。
8. **onActivityCreated():** Activity 完成创建后调用。进行 Activity 相关的初始化操作。
9. **onStart():** Fragment 变得可见时调用。执行与展示相关的操作。
10. **onResume():** Fragment 可以与用户交互时调用。此时 Fragment 处于活动状态。

接下来，Fragment 可能暂停、停止或销毁。相应的方法依次为：

1. **onPause():** 当暂停时调用。保存数据，执行与暂停相关的操作。
2. **onStop():** 当不可见时调用。执行与界面隐藏相关的操作。
3. **onDestroyView():** 视图销毁时调用。释放视图资源。
4. **onDestroy():** 实例销毁时调用。释放 Fragment 资源。

5. `onDetach()`: 与 Activity 分离时调用。执行与解除关联相关的操作。

🔗 广播接受者 (BroadcastReceiver)

广播接收器 (BroadcastReceiver) 是 Android 系统中一种用于在应用程序之间发送和接收全局消息的组件。它允许你响应系统级事件，例如设备启动、网络状态变化、电池电量变化等。广播接收器还可以用于在应用程序内部发送和接收消息，以实现不同组件之间的通信

- 设置接收到之后的操作

```
public class MyBroadcastReceiver extends BroadcastReceiver {  
    @Override  
    public void onReceive(Context context, Intent intent) {  
        // 在这里处理接收到的广播  
    }  
}
```

- 注册广播

```
<application ...>  
    ...  
    <receiver android:name=".MyBroadcastReceiver">  
        <intent-filter>  
            <action android:name="com.example.MY_BROADCAST">  
        </intent-filter>  
    </receiver>  
    ...  
</application>
```

- 发送广播

```
Intent intent = new Intent("com.example.MY_BROADCAST");  
sendBroadcast(intent);
```


🔗 普通广播和有序广播

1. 普通广播 (Normal Broadcasts) :

普通广播是异步发送的，这意味着广播的接收者几乎同时收到广播。这种广播无法被中止，因此所有注册了的接收器都会收到该广播。普通广播使用 `sendBroadcast()` 方法发送。

```
Intent intent = new Intent("com.example.MY_BROADCAST");  
sendBroadcast(intent);
```



2. 有序广播 (Ordered Broadcasts) :

有序广播是同步发送的。这意味着在发送广播时，接收器按照优先级顺序依次处理广播。优先级较高的接收器可以拦截广播，阻止优先级较低的接收器接收到广播。有序广播使用 `sendOrderedBroadcast()` 方法发送。

```
Intent intent = new Intent("com.example.MY_ORDERED_BROADCAST");  
sendOrderedBroadcast(intent, null);
```



在发送有序广播时，你可以通过调用 `abortBroadcast()` 方法来阻止后续接收器接收广播。但请注意，这仅适用于具有相同或更低优先级的接收器。

```
public class MyHighPriorityReceiver extends BroadcastReceiver {  
    @Override  
    public void onReceive(Context context, Intent intent) {  
        // 处理广播  
        abortBroadcast(); // 阻止后续接收器接收广播  
    }  
}
```



在 `AndroidManifest.xml` 文件中，你可以使用 `android:priority` 属性为广播接收器设置优先级。优先级的范围为 -1000 到 1000，数值越高，优先级越高。

```
<receiver android:name=".MyHighPriorityReceiver">
    <intent-filter android:priority="1000">
        <action android:name="com.example.MY_ORDERED_BROADCAST" />
    </intent-filter>
</receiver>
```



🔗 关于广播的案例_通过监听充电断电广播显示提示（动态注册）

- 继承一个广播接收器的类 `PowerConnectionReceiver`

```
package com.fu.tt;

import android.content.BroadcastReceiver;
import android.content.Context;
import android.content.Intent;
import android.widget.Toast;

public class PowerConnectionReceiver extends BroadcastReceiver {
    @Override
    public void onReceive(Context context, Intent intent) {
        String action = intent.getAction();
        if (action != null) {
            if (action.equals(Intent.ACTION_POWER_CONNECTED)) {
                Toast.makeText(context, "充电器已连接", Toast.LENGTH_SHORT).show();
            } else if (action.equals(Intent.ACTION_POWER_DISCONNECTED)) {
                Toast.makeText(context, "充电器已断开", Toast.LENGTH_SHORT).show();
            }
        }
    }
}
```



- 在您的 `MainActivity` 中，创建一个 `PowerConnectionReceiver` 的实例：

```
private PowerConnectionReceiver powerConnectionReceiver;
```



注意，这里的 `PowerConnectionReceiver` 就是广播接收器的类

- 在 `onCreate` 方法中初始化并注册广播接收器：

```
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);

    powerConnectionReceiver = new PowerConnectionReceiver()
    IntentFilter filter = new IntentFilter();
    filter.addAction(Intent.ACTION_POWER_CONNECTED);
    filter.addAction(Intent.ACTION_POWER_DISCONNECTED);
    registerReceiver(powerConnectionReceiver, filter);
}
```



1. `IntentFilter filter = new IntentFilter();`

这里创建了一个 `IntentFilter` 实例。`IntentFilter` 类是用于指定广播接收器需要接收哪些类型的广播事件的工具。在创建 `IntentFilter` 之后，可以通过 `addAction()` 方法添加要监听的广播事件。

2. `filter.addAction(Intent.ACTION_POWER_CONNECTED);` 这一行将 `ACTION_POWER_CONNECTED` 动作添加到了 `filter` 中。`ACTION_POWER_CONNECTED` 是一个系统广播，当设备连接到电源时触发。这意味着 `PowerConnectionReceiver` 将会在设备开始充电时收到广播。

- 在 `onDestroy` 方法中注销广播接收器：

```
@Override
protected void onDestroy() {
    super.onDestroy();
    unregisterReceiver(powerConnectionReceiver);
}
```



🔗 关于广播的案例_时区更改（静态注册）

有的广播是不能够进行静态广播的注册的

比方说。。。

1. android.intent.action.SCREEN_ON
2. android.intent.action.SCREEN_OFF
3. android.intent.action.USER_PRESENT
4. android.intent.action.TIME_TICK
5. android.intent.action.PACKAGE_ADDED
6. android.intent.action.PACKAGE_REMOVED
7. android.intent.action.BATTERY_CHANGED
8. android.intent.action.ACTION_POWER_CONNECTED
9. android.intent.action.ACTION_POWER_DISCONNECTED
10. android.intent.action.ACTION_SHUTDOWN
11. android.intent.action.ACTION_PACKAGE_NEEDS_VERIFICATION
12. android.intent.action.ACTION_PACKAGE_VERIFIED
13. android.intent.action.ACTION_PACKAGE_FIRST_LAUNCH
14. android.intent.action.ACTION_PACKAGE_FULLY_REMOVED
15. android.intent.action.ACTION_PACKAGE_INSTALL
16. android.intent.action.ACTION_PACKAGE_REPLACED
17. android.intent.action.ACTION_PACKAGE_RESTARTED
18. android.intent.action.ACTION_PACKAGE_DATA_CLEARED
19. android.intent.action.ACTION_PACKAGE_CHANGED
20. android.intent.action.ACTION_PACKAGE_REMOVED
21. android.intent.action.ACTION_PACKAGE_ADDED

- 22. `android.intent.action.ACTION_PACKAGE_SUSPENDED`
- 23. `android.intent.action.ACTION_PACKAGE_UNSPENDED`
- 24. `android.intent.action.ACTION_MY_PACKAGE_SUSPENDED`
- 25. `android.intent.action.ACTION_MY_PACKAGE_UNSPENDED`
- 26. `android.intent.action.ACTION_MANAGE_APP_PERMISSION`
- 27. `android.intent.action.ACTION_MANAGE_APP_PERMISSIONS`
- 28. `android.intent.action.ACTION_LOCATION_SOURCE_SETTINGS`
- 29. `android.intent.action.ACTION_BUG_REPORT`

.....

1. 可以静态注册的广播：

- 显式广播（即指定了特定组件的广播）。
- 部分隐式广播（没有指定特定组件，而是基于 Intent Filter 的广播），这些广播通常与系统启动、设备启动、网络状态变化等事件有关。官方文档中会明确说明哪些广播可以静态注册。

2. 只能动态注册的广播：

- 隐式广播，从 Android 8.0（API 级别 26）开始，许多系统广播事件不能使用静态注册。这类广播主要是那些频繁发生且与应用程序的实时状态相关的广播，例如电池状态、Wi-Fi状态、屏幕状态等。

下面的这个

```
<application
    ...
>
...
<receiver
    android:name=".TimeZoneChangeReceiver"
    android:exported="false">
    <intent-filter>
        <action android:name="android.intent.action.TIM
    </intent-filter>
</receiver>
</application>
```



```
package com.example.yourapp;
```



```
import android.content.BroadcastReceiver;
```

```
import android.content.Context;
```

```
import android.content.Intent;
```

```
import android.widget.Toast;
```

```
public class TimeZoneChangeReceiver extends BroadcastReceiver
```

```
    @Override
```

```
    public void onReceive(Context context, Intent intent) {
```

```
        if (intent.getAction() != null && intent.getAction()
```

```
            Toast.makeText(context, "时区已更改", Toast.LENGTH
```

```
        }
```

```
    }
```

```
}
```

🔗 关于广播的案例_Intent.ACTION_BATTERY_CHANGED显示电量（动态注册）

`ACTION_BATTERY_CHANGED` 是一个系统广播，当设备的电池状态、电量、充电方式等发生变化时，它会被触发。这个广播会提供一系列有关电池信息的额外数据，包括电池状态、充电方式、电池剩余电量等。

```
public class MainActivity extends AppCompatActivity {  
    private BatteryStatusReceiver batteryStatusReceiver;  
  
    @Override  
    protected void onCreate(Bundle savedInstanceState) {  
        super.onCreate(savedInstanceState);  
        setContentView(R.layout.activity_main);  
  
        batteryStatusReceiver = new BatteryStatusReceiver()  
        IntentFilter filter = new IntentFilter(Intent.ACTION_BATTERY_CHANGED)  
        registerReceiver(batteryStatusReceiver, filter);  
    }  
  
    @Override  
    protected void onDestroy() {  
        super.onDestroy();  
        if (batteryStatusReceiver != null) {  
            unregisterReceiver(batteryStatusReceiver);  
        }  
    }  
}
```





```
package com.example.yourapp;

import android.content.BroadcastReceiver;
import android.content.Context;
import android.content.Intent;
import android.os.BatteryManager;
import android.widget.Toast;

public class BatteryStatusReceiver extends BroadcastReceiver
    @Override
    public void onReceive(Context context, Intent intent) {
        String action = intent.getAction();
        if (action != null && action.equals(Intent.ACTION_B
            int status = intent.getIntExtra(BatteryManager.E
            int level = intent.getIntExtra(BatteryManager.E
            int scale = intent.getIntExtra(BatteryManager.E

            Toast.makeText(context, "电池电量: " + level + ",
        }
    }
}
```

`status` 变量用于存储电池的当前状态。`BatteryManager.EXTRA_STATUS` 是从 `Intent` 中获取电池状态的键。`-1` 是默认值，当获取失败时返回。

- `BatteryManager.BATTERY_STATUS_UNKNOWN`: 表示电池状态未知（值为1）。
- `BatteryManager.BATTERY_STATUS_CHARGING`: 表示电池正在充电（值为2）。
- `BatteryManager.BATTERY_STATUS_DISCHARGING`: 表示电池正在放电（值为3）。
- `BatteryManager.BATTERY_STATUS_NOT_CHARGING`: 表示电池未充电（值为4）。
- `BatteryManager.BATTERY_STATUS_FULL`: 表示电池已充满（值为5）。

🔗 动态注册的广播再来一个例子

在Activity中注册广播，可以正常监听电量状态，但随着Activity生命周期变化，不能持续监听电量

常见的不能静态注册的

`android.intent.action.SCREEN_ON`

`android.intent.action.SCREEN_OFF`

`android.intent.action.BATTERY_CHANGED`

`android.intent.action.CONFIGURATION_CHANGED`

`android.intent.action.CONFIGURATION_CHANGED` 是一个系统广播，它在设备的配置发生变化时触发。例如，当屏幕方向发生变化、用户切换语言、更改系统字体大小等情况下，此广播会被触发。

`android.intent.action.TIME_TICK`（每一分钟发送一次，类似于打点计时器）

动态注册不能放到activity中，因为动态注册必须要在activity消亡的时候调用 `unregisterReceiver`，会随着activity的解锁消失而不能再接收广播。一般的办法是在activity起来后马上start一个service,这个service里动态注册一个 `broadcastreceiver`，service必须常驻在系统内，我在切换时区的广播当中添加，打开服务的代码

```
package com.fu.tt;
```



```
import android.content.BroadcastReceiver;
```

```
import android.content.Context;
```

```
import android.content.Intent;
```

```
import android.os.BatteryManager;
```

```
import android.widget.Toast;
```

```
public class BootBroadcastReceiver extends BroadcastReceiver
```

```
    @Override
```

```
    public void onReceive(Context context, Intent intent) {
```

```
        String timeZone_action = "android.intent.action.TIM
```

```
        if(timeZone_action.equals(intent.getAction())){
```

```
            Toast.makeText(context, "BOOT_COMPLETED", Toast.L
```

```
            Intent batteryServiceIntent = new Intent(context
```

```
            context.startService(batteryServiceIntent);
```

```
        }
```

```
    }
```

```
}
```

```
<?xml version="1.0" encoding="utf-8"?>
```



```
<manifest xmlns:android="http://schemas.android.com/apk/res
```

```
    xmlns:tools="http://schemas.android.com/tools">
```

```
    <uses-permission android:name="android.permission.ACCE
```

```
    <uses-permission android:name="android.permission.RECEI
```

```
    <uses-permission android:name="android.permission.READ_
```

```
    <application
```

```
        android:allowBackup="true"
```

```
        android:dataExtractionRules="@xml/data_extraction_r
```

```
        android:fullBackupContent="@xml/backup_rules"
```

```
        android:icon="@mipmap/ic_launcher"
```

```
        android:label="@string/app_name"
```

```
        android:roundIcon="@mipmap/ic_launcher_round"
```

```
        android:supportsRtl="true"
```

```
        android:theme="@style/Theme.TT"
```

```
        tools:targetApi="31">
```

```
...
```

```

        <receiver
            android:name=".BootBroadcastReceiver"
            android:exported="false">
            <intent-filter>
                <action android:name="android.intent.action
            </intent-filter>
        </receiver>

        <activity
            android:name=".MainActivity"
            android:exported="true">
            <intent-filter>
                <action android:name="android.intent.action
                <category android:name="android.intent.cate
            </intent-filter>

            <meta-data
                android:name="android.app.lib_name"
                android:value="" />
        </activity>
        ...
        <service android:name=".BatteryService" />
    </application>
</manifest>

```

```

package com.fu.tt;

import android.app.Service;
import android.content.BroadcastReceiver;
import android.content.Context;
import android.content.Intent;
import android.content.IntentFilter;
import android.os.BatteryManager;
import android.os.Environment;
import android.os.IBinder;
import android.util.Log;
import android.widget.Toast;
import android.os.Handler;
import android.os.Looper;

import java.io.BufferedWriter;
import java.io.File;
import java.io.FileWriter;
import java.io.IOException;

```



```

import java.text.SimpleDateFormat;
import java.util.Date;

public class BatteryService extends Service {

    private static final String TAG = "BatteryReceiver";
    private static final long LOG_INTERVAL = 5000; // 5秒
    private Handler logHandler;

    private Runnable logRunnable = new Runnable() {
        @Override
        public void run() {
            Log.i(TAG, "Service is running");
            logHandler.postDelayed(this, LOG_INTERVAL);
        }
    };

    @Override
    public IBinder onBind(Intent intent) {
        return null;
    }

    @Override
    public void onCreate() {
        super.onCreate();
        Log.i(TAG, "onCreate-----");
        IntentFilter batteryfilter = new IntentFilter();
        batteryfilter.addAction(Intent.ACTION_BATTERY_CHANG
            registerReceiver(batteryReceiver, batteryfilter);
    }

    @Override
    public void onStart(Intent intent, int startId) {
        super.onStart(intent, startId);
    }

    @Override
    public int onStartCommand(Intent intent, int flags, int
        Log.i(TAG, "onStartCommand-----");
        logHandler = new Handler(Looper.getMainLooper());
        logHandler.post(logRunnable);
        return Service.START_STICKY; //
    }
}

```

```

@Override
public void onDestroy() {
    Log.i(TAG, "onDestroy-----");
    super.onDestroy();
    logHandler.removeCallbacks(logRunnable);
    this.unregisterReceiver(batteryReceiver);
}

// 接收电池信息更新的广播
private BroadcastReceiver batteryReceiver = new Broadca
{
    @Override
    public void onReceive(Context context, Intent inten
    {
        Toast.makeText(context, "执行了", Toast.LENGTH_SHO
        Log.i(TAG, "BatteryReceiver-----");
        String action = intent.getAction();
        Log.i(TAG, " 0 action:" + action);
        Log.i(TAG, "ACTION_BATTERY_CHANGED");
        int status = intent.getIntExtra("status", 0);
        int health = intent.getIntExtra("health", 0);
        boolean present = intent.getBooleanExtra("prese
        int level = intent.getIntExtra("level", 0);
        int scale = intent.getIntExtra("scale", 0);
        int icon_small = intent.getIntExtra("icon-small
        int plugged = intent.getIntExtra("plugged", 0);
        int voltage = intent.getIntExtra("voltage", 0);
        int temperature = intent.getIntExtra("temperatu
        String technology = intent.getStringExtra("tech

        String statusString = "";
        switch (status)
        {
            case BatteryManager.BATTERY_STATUS_UNKNOWN:
                statusString = "unknown";
                break;
            case BatteryManager.BATTERY_STATUS_CHARGING
                statusString = "charging";
                break;
            case BatteryManager.BATTERY_STATUS_DISCHARG
                statusString = "discharging";
                break;
            case BatteryManager.BATTERY_STATUS_NOT_CHAR
                statusString = "not charging";
                break;
        }
    }
}

```

```

        case BatteryManager.BATTERY_STATUS_FULL:
            statusString = "full";
            break;
    }
    String acString = "";

    switch (plugged)
    {
        case BatteryManager.BATTERY_PLUGGED_AC:
            acString = "plugged ac";
            break;
        case BatteryManager.BATTERY_PLUGGED_USB:
            acString = "plugged usb";
            break;
    }

    SimpleDateFormat sDateFormat = new SimpleDateFormat(
    String date = sDateFormat.format(new java.util.

    Log.i(TAG, "battery: date=" + date + ",status "
        + ",level=" + level + ",scale=" + scale
        + ",voltage=" + voltage + ",acString=" +

    }
};
private static void writeLogToFile(String logTypename,
                                   String tag, String t
    Log.i("zjq", "mylog-----");
    File path = Environment.getExternalStorageDirectory
    Date nowtime = new Date();
    String needWriteMessage = text;
    File file = new File(path, "needWriteFiel" + logTyp
    try {
        FileWriter filerWriter = new FileWriter(file, t
        BufferedWriter bufWriter = new BufferedWriter(f
        bufWriter.write(needWriteMessage);
        bufWriter.newLine();
        bufWriter.close();
        filerWriter.close();
    } catch (IOException e) {
        e.printStackTrace();
    }
}
}
}

```

🔗 隐式广播和显式广播

在前面的广播当中，我们发现，发送的时候，都是系统的行为，我们在接受的时候，只能够匹配上就能够接受这个广播，需要注意的是，有些系统的广播，不单单要注册，还需要你的权限 `<uses-permission android:name="android.permission.RECEIVE_BOOT_COMPLETED" />` 比方说这样。这种在发送的时候没有指定接受的类的时候

但是我们可以主动的发送广播。发送广播的时候，如果我们指定了接受的 class，那么就是显式广播

如果我们发送的时候没有指定接受的 class 就是隐式的广播

🔗 显式_静态注册广播的例子

```
package com.fu.tt;

import android.content.BroadcastReceiver;
import android.content.Context;
import android.content.Intent;
import android.widget.Toast;

public class CustomBroadcastReceiver extends BroadcastReceiver {

    @Override
    public void onReceive(Context context, Intent intent) {
        Toast.makeText(context, "显式广播", Toast.LENGTH_SHORT)
    }
}
```

```
<receiver android:name=".CustomBroadcastReceiver"
    android:exported="false">
    <intent-filter>
        <action android:name="com.example.CUSTOM_BR
    </intent-filter>
</receiver>
```



```
package com.fu.tt;
import android.content.Intent;
import android.content.IntentFilter;
import android.widget.Button;
import android.view.View;

import androidx.appcompat.app.AppCompatActivity;
import androidx.fragment.app.FragmentManager;
import androidx.fragment.app.FragmentTransaction;

import android.os.Bundle;
import android.widget.Toast;
import android.content.SharedPreferences;

public class MainActivity extends AppCompatActivity implements

    private BatteryStatusReceiver batteryStatusReceiver;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        // 发送广播
        Button btnSendBroadcast = findViewById(R.id.btn_sen
        btnSendBroadcast.setOnClickListener(new View.OnClic
            @Override
            public void onClick(View v) {
                sendCustomBroadcast();
            }
        });
    }
```




```

@Override
protected void onDestroy()
{
    super.onDestroy();
    // unregisterReceiver(powerConnectionReceiver);
    if (batteryStatusReceiver != null) {
        unregisterReceiver(batteryStatusReceiver);
    }
}

private void sendCustomBroadcast() {
    Intent customBroadcast = new Intent(this, CustomBro
    customBroadcast.setAction("com.example.CUSTOM_BROAD
    sendBroadcast(customBroadcast);
}
}

```

🔗 显式_动态注册的案例

我们将创建两个 Activity: MainActivity 和 SecondActivity。MainActivity 将动态注册一个广播接收器, 而 SecondActivity 将发送一个显式广播, 通知 MainActivity 更新 UI

BroadcastReceiver.java

```

package com.fu.tt;
import android.content.BroadcastReceiver;
import android.content.Context;
import android.content.Intent;
import android.widget.Toast;

public class MyBroadcastReceiver extends BroadcastReceiver
@Override
public void onReceive(Context context, Intent intent) {
    String message = intent.getStringExtra("message");
    if (context instanceof MainActivity) {
        ((MainActivity) context).updateUI(message);
    }
}
}

```

在这个方法中，我们需要更新 `MainActivity` 的 UI，但是我们需要首先确保 `context` 是一个 `MainActivity` 实例。

1. 首先，我们将 `context` 强制转换为 `MainActivity` 类型。这是因为 `context` 是一个更通用的类型，而我们需要访问 `MainActivity` 中的 `updateUI` 方法。强制转换确保我们可以调用这个特定的方法。
2. 接下来，我们调用 `updateUI(message)` 方法。这个方法是在 `MainActivity` 类中定义的，并且接收一个字符串作为参数。在这个例子中，`message` 参数是从广播的 `Intent` 中获取的。

`MainActivity.java` 并且定义 `updateUI`



```
public class MainActivity extends AppCompatActivity {
    private MyBroadcastReceiver myBroadcastReceiver;
    private TextView textView;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        textView = findViewById(R.id.textView);
        myBroadcastReceiver = new MyBroadcastReceiver();

        IntentFilter filter = new IntentFilter("com.example
        registerReceiver(myBroadcastReceiver, filter);
    }

    @Override
    protected void onResume() {
        super.onResume();
    }

    @Override
    protected void onPause() {
        super.onPause();
        unregisterReceiver(myBroadcastReceiver);
    }

    public void updateUI(String message) {
        textView.setText(message);
    }

    public void goToSecondActivity(View view) {
        Intent intent = new Intent(this, SecondActivity.class);
        startActivity(intent);
    }
}
```

SecondActivity.java



```
public class SecondActivity extends AppCompatActivity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_second);
    }

    public void sendMessageToMainActivity(View view) {
        Intent intent = new Intent("com.example.UPDATE_UI")
        intent.setPackage(getPackageName());
        intent.putExtra("message", "Hello from SecondActivi
        sendBroadcast(intent);
    }
}
```

`intent.setPackage(getPackageName());` 这行代码设置了 Intent 的目标包名。这意味着，只有这个包名下的应用程序才能接收到这个 Intent。在这个例子中，`getPackageName()` 返回当前应用的包名。

这一行代码的主要目的是将 Intent 的范围限制在当前应用程序内。这样，你可以确保 Intent 只会发送给你自己的应用，而不会泄露到其他应用中。

SecondActivity.xml



```
<?xml version="1.0" encoding="utf-8"?>
<androidx.constraintlayout.widget.ConstraintLayout xmlns:android="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=".SecondActivity">

    <TextView
        android:id="@+id/textView2"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Welcome to Second Activity!"
        android:textSize="24sp"
        app:layout_constraintBottom_toBottomOf="parent"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintTop_toTopOf="parent" />

    <Button
        android:id="@+id/button_send_broadcast"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:onClick="sendBroadcast"
        android:text="send2 Broadcast"
        app:layout_constraintBottom_toTopOf="@+id/textView2"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintHorizontal_bias="0.498"
        app:layout_constraintStart_toStartOf="parent" />

    <Button
        android:id="@+id/button2_send_broadcast"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_marginBottom="60dp"
        android:onClick="sendMessageToMainActivity"
        android:text="send Broadcast"
        app:layout_constraintBottom_toTopOf="@+id/textView2"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintHorizontal_bias="0.497"
        app:layout_constraintStart_toStartOf="parent" />

</androidx.constraintlayout.widget.ConstraintLayout>
```

🔗 隐式_静态广播

已经关闭大部分这样的操作，能够写，不报错，但是不发送。

少量的系统广播，主要就是触发比较少的 比方说，开机广播，还有就是切换时区广播

AndroidManifest.xml

```
<manifest xmlns:android="http://schemas.android.com/apk/res-anim" package="com.example.timezonechange">

    <application
        ...>

        <receiver android:name=".TimeZoneChangeReceiver">
            <intent-filter>
                <action android:name="android.intent.action.TIMEZONE_CHANGED">
            </intent-filter>
        </receiver>

    </application>

</manifest>
```

MyBroadcastReceiverYin.java

```

package com.fu.tt;
import android.content.BroadcastReceiver;
import android.content.Context;
import android.content.Intent;
import android.widget.Toast;

public class MyBroadcastReceiverYin extends BroadcastReceiver
    @Override
    public void onReceive(Context context, Intent intent) {
        if ("android.intent.action.TIMEZONE_CHANGED".equals
            String message = "收到切换时区的，隐式静态广播";
            Toast.makeText(context, message, Toast.LENGTH_S
        }
    }
}

```

🔗 隐式_动态广播

Activity_main.xml

```

<?xml version="1.0" encoding="utf-8"?>
<androidx.constraintlayout.widget.ConstraintLayout xmlns:an
    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">

    <TextView
        android:id="@+id/textView"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Hello, World!"
        android:textSize="24sp"
        app:layout_constraintBottom_toBottomOf="parent"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintTop_toTopOf="parent" />

    <Button
        android:id="@+id/send_broadcast_yin_button"

```

```
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_marginTop="420dp"
        android:text=" 隐式静态广播 "
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintHorizontal_bias="0.498"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintTop_toTopOf="parent" />

<Button
    android:id="@+id/broadcast_button"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_marginTop="500dp"
    android:text=" 隐式动态广播 "
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintHorizontal_bias="0.498"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintTop_toTopOf="parent" />

</androidx.constraintlayout.widget.ConstraintLayout>
```

Mainactivity.java



```
private CustomBroadcastReceiver customBroadcastReceiver;


@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);

    Button broadcastButton = findViewById(R.id.broadcas
    broadcastButton.setOnClickListener(new View.OnClickListener
    @Override
    public void onClick(View view) {
        Intent intent = new Intent("com.example.CUSTOM_
        intent.putExtra("message", "Hello from MainActi
        sendBroadcast(intent);
    }
});

customBroadcastReceiver = new CustomBroadcastReceiver()
IntentFilter intentFilter = new IntentFilter("com.examp
registerReceiver(customBroadcastReceiver, intentFilter)
}

@Override
protected void onDestroy() {
    super.onDestroy();
    unregisterReceiver(customBroadcastReceiver);
}
```

CustomBroadcastReceiver.java



```
package com.fu.tt;

import android.content.BroadcastReceiver;
import android.content.Context;
import android.content.Intent;
import android.widget.Toast;

public class CustomBroadcastReceiver extends BroadcastReceiver

    @Override
    public void onReceive(Context context, Intent intent)
    {
        if (intent.getAction().equals("com.example.CUSTOM_A
        {
            Toast.makeText(context, "收到隐式动态广播:  ", To
        }
        else
        {
            Toast.makeText(context, "收到广播:  ", Toast.LEN
        }
    }
}
```

🔗 Activity 间通信广播：

简单来说就是一个 activity 进行注册（一般来说注册在前，至少同时），另一个 activity 进行发送，然后激活广播接收器。显示内容



```
package com.fu.tt;
import android.content.Intent;
import android.content.IntentFilter;
import android.widget.Button;
import android.view.View;

import androidx.appcompat.app.AppCompatActivity;
import androidx.fragment.app.FragmentManager;
import androidx.fragment.app.FragmentTransaction;

import android.os.Bundle;
import android.widget.Toast;
import android.content.SharedPreferences;

public class MainActivity extends AppCompatActivity implements

    // 客制化广播接收器
    private CustomBroadcastReceiver customBroadcastReceiver

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        // 通信, 广播注册
        customBroadcastReceiver = new CustomBroadcastReceiv
        IntentFilter intentFilter = new IntentFilter("com.o
        registerReceiver(customBroadcastReceiver, intentFil

    }

    @Override
    protected void onDestroy() {
        Toast.makeText(MainActivity.this, "已经执行了onDestory
        super.onDestroy();
        unregisterReceiver(customBroadcastReceiver);
    }
}
```



```
package com.fu.tt;

import androidx.appcompat.app.AppCompatActivity;

import android.content.Intent;
import android.content.IntentFilter;
import android.os.Bundle;
import android.util.Log;
import android.view.View;
import android.widget.Button;
import android.widget.Toast;

public class SecondActivity extends AppCompatActivity{
    @Override
    protected void onCreate(Bundle savedInstanceState)
    {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_second);

        // 发送广播
        Button btnSendBroadcast = findViewById(R.id.btn_sendBroadcast);
        btnSendBroadcast.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View v) {
                sendCustomBroadcast();
            }
        });
    }

    @Override
    protected void onDestroy() {
        super.onDestroy();
    }

    // 附带一个信息，发送了一个广播 com.out.CUSTOM_BROADCAST
    private void sendCustomBroadcast() {
        Intent customBroadcast = new Intent("com.out.CUSTOM_BROADCAST");
        customBroadcast.putExtra("message", "来自MainActivity");
        sendBroadcast(customBroadcast);
    }
}
```

```
package com.fu.tt;
```



```
import android.content.BroadcastReceiver;
```

```
import android.content.Context;
```

```
import android.content.Intent;
```

```
import android.widget.Toast;
```

```
public class CustomBroadcastReceiver extends BroadcastReceiver
```

```
    @Override
```

```
    public void onReceive(Context context, Intent intent)
```

```
    {
```

```
        String message = intent.getStringExtra("message");
```

```
        Toast.makeText(context, "收到广播: " + message, Toa
```

```
    }
```

```
}
```

🔗 **SecondActivity 通过广播修改 MainActivity 的展示效果**

1. MainActivity对应的 (activity_main.xml) :




```
<?xml version="1.0" encoding="utf-8"?>
<androidx.constraintlayout.widget.ConstraintLayout xmlns:an
    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">

    <TextView
        android:id="@+id/textView"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Hello, World!"
        app:layout_constraintBottom_toBottomOf="parent"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintTop_toTopOf="parent" />

    <Button
        android:id="@+id/go_to_second_button"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:onClick="goToSecondActivity" <! 设置了这个，就
        android:text="Go to Second Activity"
        app:layout_constraintBottom_toTopOf="@+id/textView"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintTop_toTopOf="parent" />
</androidx.constraintlayout.widget.ConstraintLayout>
```

MainActivity.java:



```
import androidx.appcompat.app.AppCompatActivity;
import android.content.BroadcastReceiver;
import android.content.Context;
import android.content.Intent;
import android.content.IntentFilter;
import android.os.Bundle;
import android.view.View;
import android.widget.TextView;
```

```

public class MainActivity extends AppCompatActivity {

    private TextView textView; // 全局变量，获得更新的数据

    // 声明在 MainActivity 当中的广播接收器，在里面能够接受来自 inter
    private BroadcastReceiver updateUIReceiver = new Broadc
        @Override
        public void onReceive(Context context, Intent inten
            String newText = intent.getStringExtra("new_tex
            textView.setText(newText); // 这个就是用来设置 te
        }
    };

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        textView = findViewById(R.id.textView);
    }


    @Override
    protected void onResume() {
        super.onResume();
        IntentFilter filter = new IntentFilter("com.example
        registerReceiver(updateUIReceiver, filter);
    }

    @Override
    protected void onDestroy() {
        super.onDestroy();
        unregisterReceiver(updateUIReceiver);
    }

    public void goToSecondActivity(View view) {
        Intent intent = new Intent(this, SecondActivity.class)
        startActivity(intent);
    }
}

```

SecondActivity (activity_second.xml)



```
<?xml version="1.0" encoding="utf-8"?>
<androidx.constraintlayout.widget.ConstraintLayout xmlns:an
    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=".SecondActivity">

    <Button
        android:id="@+id/button_send_broadcast"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:onClick="sendBroadcast"
        android:text="Send Broadcast"
        app:layout_constraintBottom_toBottomOf="parent"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintTop_toTopOf="parent" />
</androidx.constraintlayout.widget.ConstraintLayout>
```

SecondActivity.java:


```
import androidx.appcompat.app.AppCompatActivity;
import android.content.Intent;
import android.os.Bundle;
import android.view.View;

public class SecondActivity extends AppCompatActivity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_second);
    }

    // 发送一条广播，并且携带一条信息
    public void sendBroadcast(View view) {
        Intent intent = new Intent("com.example.UPDATE_UI")
        intent.putExtra("new_text", "Text updated from Seco
        sendBroadcast(intent);
    }
}
```

🔗 一个最简单的有序广播（动态注册_隐式）

有序广播（Ordered Broadcast）是一种特殊类型的广播，允许多个广播接收器按照优先级顺序接收和处理广播。在传统的广播中，所有注册的广播接收器几乎是同时收到广播，但在有序广播中，系统会按照接收器的优先级顺序依次发送广播。每个接收器在处理完广播后，可以决定是否传递给下一个接收器，还可以修改广播数据。

如果我们没有写优先级，那么这个优先级就是 0。

如果大家都是 0 那么顺序就是注册的顺序，接近同步。

翻译

priority：优先权

MainActivity.java

```
package com.fu.tt;
```

```
import android.content.BroadcastReceiver;
import android.content.Context;
import android.content.Intent;
import android.content.IntentFilter;
import android.util.Log;
import android.widget.Button;
import android.view.View;

import androidx.appcompat.app.AppCompatActivity;
import androidx.fragment.app.FragmentManager;
import androidx.fragment.app.FragmentTransaction;

import android.os.Bundle;
import android.widget.TextView;
import android.widget.Toast;
import android.content.SharedPreferences;

public class MainActivity extends AppCompatActivity {

    private HighPriorityReceiver highPriorityReceiver;
    private LowPriorityReceiver lowPriorityReceiver;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        // 按钮调用我发送广播
        Button broadcastButton = findViewById(R.id.broadcas
        broadcastButton.setOnClickListener(new View.OnClickListener()
        @Override
        public void onClick(View view) {
            sendOrderedBroadcast();
        }
        });

        // 调用注册广播
        highPriorityReceiver = new HighPriorityReceiver();
        lowPriorityReceiver = new LowPriorityReceiver();
        registerOrderedReceivers();
    }

    // 注销广播
    @Override
```

```

protected void onDestroy() {
    super.onDestroy();

    unregisterReceiver(highPriorityReceiver);
    unregisterReceiver(lowPriorityReceiver);
}

protected void onResume()
{
    super.onResume();
}

@Override
protected void onPause() {
    super.onPause();
}

// 注册广播，设置优先级
private void registerOrderedReceivers() {
    IntentFilter highPriorityFilter = new IntentFilter(
        highPriorityFilter.setPriority(100);
    registerReceiver(highPriorityReceiver, highPriority

    IntentFilter lowPriorityFilter = new IntentFilter("
        lowPriorityFilter.setPriority(999);
    registerReceiver(lowPriorityReceiver, lowPriorityFi
}

// 发送广播
private void sendOrderedBroadcast() {
    Intent intent = new Intent("com.example.ORDERED_BRO
    sendOrderedBroadcast(intent, null);
}
}

```

LowPriorityReceiver.java

```
package com.fu.tt;

import android.content.BroadcastReceiver;
import android.content.Context;
import android.content.Intent;
import android.util.Log;
import android.widget.Toast;

public class LowPriorityReceiver extends BroadcastReceiver
    @Override
    public void onReceive(Context context, Intent intent) {
        Toast.makeText(context, "LowPriorityReceiver 接收到广
    }
}
```

HighPriorityReceiver.java

```
package com.fu.tt;

import android.content.BroadcastReceiver;
import android.content.Context;
import android.content.Intent;
import android.util.Log;
import android.widget.Toast;

public class HighPriorityReceiver extends BroadcastReceiver
    @Override
    public void onReceive(Context context, Intent intent) {
        Toast.makeText(context, "HighPriorityReceiver 接收到
    }
}
```

🔗 一个最简单的有序广播（静态注册_显式）

AndroidManifest.xml

```
<receiver android:name=".LowPriorityReceiver" android:enabled="true" android:exported="true">
```

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res
    xmlns:tools="http://schemas.android.com/tools">

    <uses-permission android:name="android.permission.ACCESS
    <uses-permission android:name="android.permission.RECEI
    <uses-permission android:name="android.permission.READ_

    <uses-permission android:name="com.example.permission.M
    <uses-permission android:name="com.example.permission.M

    <permission
        android:name="com.example.permission.MY_BROADCAST_P
        android:protectionLevel="normal" />

    <uses-permission android:name="com.example.permission.M

    <uses-permission android:name="android.permission.READ_

    <application
        android:allowBackup="true"
        android:dataExtractionRules="@xml/data_extraction_r
        android:fullBackupContent="@xml/backup_rules"
        android:icon="@mipmap/ic_launcher"
        android:label="@string/app_name"
        android:roundIcon="@mipmap/ic_launcher_round"
        android:supportsRtl="true"
        android:theme="@style/Theme.TT"
        tools:targetApi="31">

        <receiver android:name=".MyOrderedBroadcastReceiver
            android:exported="false">
            <intent-filter android:priority="100"> <!-- 设
                <action android:name="com.example.ORDERED_E
            </intent-filter>
        </receiver>

        <activity
            android:name=".MainActivity"
            android:exported="true">
            <intent-filter>
                <action android:name="android.intent.action
```

```

        <category android:name="android.intent.cate
    </intent-filter>

    <meta-data
        android:name="android.app.lib_name"
        android:value="" />
    </activity>

</application>
</manifest>

```

MyOrderedBroadcastReceiver.java

```

package com.fu.tt;
import android.content.BroadcastReceiver;
import android.content.Context;
import android.content.Intent;
import android.util.Log;
import android.widget.Toast;

public class MyOrderedBroadcastReceiver extends BroadcastRe
    @Override
    public void onReceive(Context context, Intent intent) {
        String action = intent.getAction();
        if ("com.example.ORDERED_EXPLICIT_ACTION".equals(ac
            Toast.makeText(context, "接收到静态注册有序显式广播'
        }
    }
}

```

MainActivity.java

```

package com.fu.tt;
import android.app.Activity;
import android.content.BroadcastReceiver;
import android.content.ComponentName;
import android.content.Context;
import android.content.Intent;

```

```
import android.content.IntentFilter;
import android.content.pm.PackageManager;
import android.os.Handler;
import android.os.HandlerThread;
import android.util.Log;
import android.widget.Button;
import android.view.View;
import androidx.annotation.NonNull;
import androidx.appcompat.app.AppCompatActivity;
import androidx.core.app.ActivityCompat;
import androidx.core.content.ContextCompat;
import android.os.Bundle;
import android.widget.Toast;
import android.os.HandlerThread;

import android.Manifest;

public class MainActivity extends AppCompatActivity {

    private ReceiverA receiverA;
    private ReceiverB receiverB;
    private BroadcastReceiver finalReceiver;
    private Handler handler;
    private HandlerThread handlerThread;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        HandlerThread handlerThread = new HandlerThread("Fi
        handlerThread.start();
        handler = new Handler(handlerThread.getLooper());

        // 发送有序广播
        findViewById(R.id.send_ordered_broadcast_button).se
        @Override
        public void onClick(View v) {
            Log.i("guangbo", "发送按钮点击");
            Toast.makeText(MainActivity.this, "发送广播",
                sendOrderedExplicitBroadcast());
        }
    }
}
```

```

        }
    });
}

private void registerReceivers() {
    receiverA = new ReceiverA();
    IntentFilter filterA = new IntentFilter("com.exempl
    filterA.setPriority(100);
    registerReceiver(receiverA, filterA);

    receiverB = new ReceiverB();
    IntentFilter filterB = new IntentFilter("com.exempl
    filterB.setPriority(99);
    registerReceiver(receiverB, filterB);
}

private void sendOrderedBroadcastWithFinalReceiver() {
    Intent intent = new Intent("com.example.orderedbroa
    finalReceiver = new FinalReceiver();
    // 发送有序广播, 添加最终接收者
    sendOrderedBroadcast(intent, null, finalReceiver, h
}

@Override
protected void onDestroy() {
    super.onDestroy();
    unregisterReceiver(finalReceiver);
}

protected void onResume()
{
    super.onResume();
}

@Override
protected void onPause() {
    super.onPause();
}

private void sendOrderedExplicitBroadcast() {
    Intent intent = new Intent();
    intent.setComponent(new ComponentName(getPackageNam
    intent.setAction("com.example.ORDERED_EXPLICIT_ACTI
    sendOrderedBroadcast(intent, null);
}

```



```
}  
}
```

这个代码

```
private void sendOrderedExplicitBroadcast() {  
    Intent intent = new Intent();  
    intent.setComponent(new ComponentName(getPackageName()  
    intent.setAction("com.example.ORDERED_EXPLICIT_ACTI  
    sendOrderedBroadcast(intent, null);  
}
```



跟这个代码

```
private void sendOrderedExplicitBroadcast() {  
    Intent intent = new Intent(this, MyOrderedBroadcastR  
    intent.setAction("com.example.ORDERED_EXPLICIT_ACTI  
    sendOrderedBroadcast(intent, null);  
}
```



作用是一样的

🔗 阻断有序广播

```
abortBroadcast(); // 阻断广播
```

ReceiverA.java



```
package com.fu.tt;

import android.content.BroadcastReceiver;
import android.content.Context;
import android.content.Intent;
import android.util.Log;

public class ReceiverA extends BroadcastReceiver {
    @Override
    public void onReceive(Context context, Intent intent) {
        if ("com.example.orderedbroadcast.ACTION".equals(in
            Log.d("Receiver", "收到有序广播, 并且已经阻断了广播的
            abortBroadcast(); // 阻断广播
        }
    }
}
```

ReceiverB.java



```
package com.fu.tt;

import android.content.BroadcastReceiver;
import android.content.Context;
import android.content.Intent;
import android.util.Log;

public class ReceiverB extends BroadcastReceiver {
    @Override
    public void onReceive(Context context, Intent intent) {
        if ("com.example.orderedbroadcast.ACTION".equals(in
            Log.d("Receiver", "收到有序广播B");
        }
    }
}
```




```
<?xml version="1.0" encoding="utf-8"?>
<androidx.constraintlayout.widget.ConstraintLayout xmlns:an
    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">

    . . . . .

    <Button
        android:id="@+id/send_ordered_broadcast_button"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="发送有序广播"
        app:layout_constraintBottom_toBottomOf="parent"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintHorizontal_bias="0.501"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintTop_toTopOf="parent"
        app:layout_constraintVertical_bias="0.622" />

</androidx.constraintlayout.widget.ConstraintLayout>
```

MainActivity.java



```
package com.fu.tt;
import android.content.Intent;
import android.content.IntentFilter;
import android.content.pm.PackageManager;
import android.util.Log;
import android.widget.Button;
import android.view.View;
import androidx.annotation.NonNull;
import androidx.appcompat.app.AppCompatActivity;
import androidx.core.app.ActivityCompat;
import androidx.core.content.ContextCompat;
import android.os.Bundle;
```

```
import android.widget.Toast;

import android.Manifest;

public class MainActivity extends AppCompatActivity {

    private ReceiverA receiverA;
    private ReceiverB receiverB;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        // 动态注册广播接收器
        registerReceivers();

        // 发送有序广播
        findViewById(R.id.send_ordered_broadcast_button).setOnClickListener() {
            @Override
            public void onClick(View v) {
                Log.i("guangbo", "发送按钮点击");
                sendOrderedBroadcast();
            }
        });
    }

    private void registerReceivers() {
        receiverA = new ReceiverA();
        IntentFilter filterA = new IntentFilter("com.example")
        filterA.setPriority(100);
        registerReceiver(receiverA, filterA);

        receiverB = new ReceiverB();
        IntentFilter filterB = new IntentFilter("com.example")
        filterB.setPriority(999);
        registerReceiver(receiverB, filterB);
    }

    private void sendOrderedBroadcast() {
```

```

        Intent intent = new Intent("com.example.orderedbroa
sendOrderedBroadcast(intent, null);
        Log.i("guangbo", "发送函数执行完成");
    }

    @Override
    protected void onDestroy() {
        super.onDestroy();
        unregisterReceiver(receiverA);
        unregisterReceiver(receiverB);
    }

    protected void onResume()
    {
        super.onResume();
    }

    @Override
    protected void onPause() {
        super.onPause();
    }
}

```

🔗 有序广播（修改广播单条信息_修改广播结果）

MainActivity.java

```

package com.fu.tt;
import android.app.Activity;
import android.content.BroadcastReceiver;
import android.content.ComponentName;
import android.content.Context;
import android.content.Intent;
import android.content.IntentFilter;
import android.content.pm.PackageManager;
import android.os.Handler;
import android.os.HandlerThread;
import android.util.Log;
import android.widget.Button;
import android.view.View;

```



```
import androidx.annotation.NonNull;
import androidx.appcompat.app.AppCompatActivity;
import androidx.core.app.ActivityCompat;
import androidx.core.content.ContextCompat;
import android.os.Bundle;
import android.widget.Toast;
import android.os.HandlerThread;

import android.Manifest;

public class MainActivity extends AppCompatActivity {

    private BroadcastReceiver myOrderedBroadcastReceiver1;
    private BroadcastReceiver myOrderedBroadcastReceiver2;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        // 注册广播
        registerOrderedReceivers();

        // 发送有序广播
        findViewById(R.id.send_ordered_broadcast_button).setOnClickListener() {
            @Override
            public void onClick(View v) {
                Log.i("guangbo", "发送按钮点击");
                Toast.makeText(MainActivity.this, "发送广播",
                    sendOrderedBroadcastWithModifiedData());
            }
        });
    }

    // 注册广播
    private void registerOrderedReceivers() {
        IntentFilter filter = new IntentFilter("com.example
        filter.setPriority(100);
```

```

        myOrderedBroadcastReceiver1 = new MyOrderedBroadcas
        registerReceiver(myOrderedBroadcastReceiver1, filte

        filter.setPriority(50);
        myOrderedBroadcastReceiver2 = new MyOrderedBroadcas
        registerReceiver(myOrderedBroadcastReceiver2, filte
    }

// 注销广播
@Override
protected void onDestroy() {
    super.onDestroy();
    unregisterReceiver(myOrderedBroadcastReceiver1);
    unregisterReceiver(myOrderedBroadcastReceiver2);
}

// 发送广播，并且附带信息
private void sendOrderedBroadcastWithModifiedData() {
    Intent intent = new Intent("com.example.ORDERED_ACT
    intent.putExtra("message", "Hello, this is a messag

    // 发送有序广播
    sendOrderedBroadcast(intent, null);
}

protected void onResume()
{
    super.onResume();
}

@Override
protected void onPause() {
    super.onPause();
}
}

```

MyOrderedBroadcastReceiver1.java



```
package com.fu.tt;

import android.content.BroadcastReceiver;
import android.content.Context;
import android.content.Intent;
import android.util.Log;
import android.widget.Toast;

public class MyOrderedBroadcastReceiver1 extends BroadcastR
    @Override
    public void onReceive(Context context, Intent intent) {
        String originalMessage = intent.getStringExtra("mes
        String newMessage = originalMessage + " (Modified b

        Log.d("Receiver1", "Original Message: " + originalM
        Log.d("Receiver1", "New Message: " + newMessage);

        // 修改广播信息
        setResultData(newMessage);
    }
}
```

MyOrderedBroadcastReceiver2



```
package com.fu.tt;

import android.content.BroadcastReceiver;
import android.content.Context;
import android.content.Intent;
import android.util.Log;
import android.widget.Toast;

public class MyOrderedBroadcastReceiver2 extends BroadcastR
    @Override
    public void onReceive(Context context, Intent intent) {
        String modifiedMessage = getResultData();
        Log.d("Receiver2", "Modified Message: " + modifiedM
    }
}
```


🔗 有序广播（修改广播多条信息_修改广播额外信息）

MainActivity.java

```
package com.fu.tt;
import android.app.Activity;
import android.content.BroadcastReceiver;
import android.content.ComponentName;
import android.content.Context;
import android.content.Intent;
import android.content.IntentFilter;
import android.content.pm.PackageManager;
import android.os.Handler;
import android.os.HandlerThread;
import android.util.Log;
import android.widget.Button;
import android.view.View;
import androidx.annotation.NonNull;
import androidx.appcompat.app.AppCompatActivity;
import androidx.core.app.ActivityCompat;
import androidx.core.content.ContextCompat;
import android.os.Bundle;
import android.widget.Toast;
import android.os.HandlerThread;

import android.Manifest;

public class MainActivity extends AppCompatActivity {

    private BroadcastReceiver myOrderedBroadcastReceiver1;
    private BroadcastReceiver myOrderedBroadcastReceiver2;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        // 注册广播
        registerOrderedReceivers();
    }
}
```

```

        // 发送有序广播
        findViewById(R.id.send_ordered_broadcast_button).setOnClickListener(
            @Override
            public void onClick(View v) {
                Log.i("guangbo", "发送按钮点击");
                Toast.makeText(MainActivity.this, "发送广播",
                    sendOrderedBroadcastWithModifiedData());
            }
        });
    }

    // 注册广播
    private void registerOrderedReceivers() {
        IntentFilter filter = new IntentFilter("com.example
        filter.setPriority(100);
        myOrderedBroadcastReceiver1 = new MyOrderedBroadcas
        registerReceiver(myOrderedBroadcastReceiver1, filte

        filter.setPriority(50);
        myOrderedBroadcastReceiver2 = new MyOrderedBroadcas
        registerReceiver(myOrderedBroadcastReceiver2, filte
    }

    // 注销广播
    @Override
    protected void onDestroy() {
        super.onDestroy();
        unregisterReceiver(myOrderedBroadcastReceiver1);
        unregisterReceiver(myOrderedBroadcastReceiver2);
    }

    // 给我们的广播写入两条数据，两个键值对
    private void sendOrderedBroadcastWithModifiedData() {
        Intent intent = new Intent("com.example.ORDERED_ACT
        intent.putExtra("message1", "Hello, this is message
        intent.putExtra("message2", "Hello, this is message

        // 发送有序广播
        sendOrderedBroadcast(intent, null);
    }

```

```
protected void onResume()  
{  
    super.onResume();  
}  
  
@Override  
protected void onPause() {  
    super.onPause();  
}  
  
}
```

MyOrderedBroadcastReceiver1.java

```
package com.fu.tt;
```



```
import android.content.BroadcastReceiver;
```

```
import android.content.Context;
```

```
import android.content.Intent;
```

```
import android.os.Bundle;
```

```
import android.util.Log;
```

```
import android.widget.Toast;
```

```
public class MyOrderedBroadcastReceiver1 extends BroadcastR  
    @Override
```

```
    public void onReceive(Context context, Intent intent) {  
        String originalMessage1 = intent.getStringExtra("me  
        String newMessage1 = originalMessage1 + " (Modified  
        Log.d("Receiver1", "Original Message1: " + original  
        Log.d("Receiver1", "New Message1: " + newMessage1);
```

```
  
        String originalMessage2 = intent.getStringExtra("me  
        String newMessage2 = originalMessage2 + " (Modified  
        Log.d("Receiver1", "Original Message2: " + original  
        Log.d("Receiver1", "New Message2: " + newMessage2);
```

```
  
        // 获取当前的额外数据
```

```
        Bundle extras = getResultExtras(true);
```

```
  
        // 修改多个键值对的信息
```

```
        extras.putString("message1", "newValue1");
```

```
        extras.putString("message2", "newValue2");
```

```
  
        // 将修改后的 Bundle 设置为结果额外数据
```

```
        setResultExtras(extras);
```

```
    }
```

```
}
```

MyOrderedBroadcastReceiver2.java

```
package com.fu.tt;
```



```
import android.content.BroadcastReceiver;
```

```
import android.content.Context;
```

```
import android.content.Intent;
```

```
import android.os.Bundle;
```

```
import android.util.Log;
```

```
import android.widget.Toast;
```

```
public class MyOrderedBroadcastReceiver2 extends BroadcastReceiver  
    @Override
```

```
    public void onReceive(Context context, Intent intent) {
```

```
        Bundle extras = getResultExtras(false);
```

```
        String modifiedMessage1 = extras.getString("message
```

```
        String modifiedMessage2 = extras.getString("message
```

```
        Log.d("Receiver2", "Modified Message1: " + modified
```

```
        Log.d("Receiver2", "Modified Message2: " + modified
```

```
    }
```

```
}
```

🔗 setResultExtras 和 setResultData

有序广播当中的数据传递

`setResultExtras(Bundle)` 和 `setResultData(String)` 是在 `BroadcastReceiver` 的 `onReceive()` 方法中使用的两个方法，它们分别用于修改广播中的额外数据（键值对）和结果数据（字符串）。这两个方法通常在有序广播中使用，因为有序广播允许接收者之间传递修改后的数据。

1. `setResultExtras(Bundle)` :

这个方法用于设置或修改广播中的额外数据。额外数据包含在一个 `Bundle` 对象中，可以在广播中附带任意数量的键值对。你可以使用

`getResultExtras(boolean)` 获取当前的额外数据，然后对其进行修改，最后使用 `setResultExtras(Bundle)` 设置回广播。




```
@Override
public void onReceive(Context context, Intent intent) {
    // 获取当前的额外数据
    Bundle extras = getResultExtras(true);

    // 修改键值对的信息
    extras.putString("key1", "new value 1");
    extras.putString("key2", "new value 2");

    // 将修改后的 Bundle 设置为结果额外数据
    setResultExtras(extras);
}
```

2. setResultData(String) :

这个方法用于设置或修改广播中的结果数据。结果数据是一个字符串，它可以在广播中传递单个值。你可以使用 `getResultData()` 获取当前的结果数据，然后使用 `setResultData(String)` 设置回广播。



```
@Override
public void onReceive(Context context, Intent intent) {
    // 获取当前的结果数据
    String currentData = getResultData();

    // 修改结果数据
    String newData = currentData + " modified by Receiver";

    // 将修改后的数据设置为结果数据
    setResultData(newData);
}
```

要获取额外数据（键值对）和结果数据（字符串），你可以在 `BroadcastReceiver` 的 `onReceive()` 方法中使用 `getResultExtras(boolean)` 和 `getResultData()` 方法。

1. 获取额外数据（键值对）：

你可以使用 `getResultExtras(boolean)` 方法获取广播中的额外数据。这个方法返回一个 `Bundle` 对象，其中包含所有键值对。`boolean` 参数表示如果当前没有额外数据，是否创建一个新的空 `Bundle` 对象。

```
@Override
public void onReceive(Context context, Intent intent) {
    // 获取额外数据
    Bundle extras = getResultExtras(true);

    // 从额外数据中获取特定的键值对
    String value1 = extras.getString("key1");
    String value2 = extras.getString("key2");

    // 对获取到的数据进行操作
    // ...
}
```

2. 获取结果数据（字符串）：

你可以使用 `getResultData()` 方法获取广播中的结果数据。这个方法返回一个字符串，表示结果数据。

```
@Override
public void onReceive(Context context, Intent intent) {
    // 获取结果数据
    String resultData = getResultData();

    // 对获取到的结果数据进行操作
    // ...
}
```

🔗 普通广播的数据传递（Bundle&intent）

****intent ****

MainActivity.java

```
Intent intent = new Intent("com.example.myaction");  
intent.putExtra("key_string", "Hello World!");  
intent.putExtra("key_int", 42);  
intent.putExtra("key_bool", true);  
sendBroadcast(intent);
```



MyBroadcastReceiver.java

```
public class MyBroadcastReceiver extends BroadcastReceiver  
    @Override  
    public void onReceive(Context context, Intent intent) {  
        String valueString = intent.getStringExtra("key_str  
        int valueInt = intent.getIntExtra("key_int", 0);  
        boolean valueBool = intent.getBooleanExtra("key_boo  
  
        // 在这里使用获取到的值进行相应的操作  
    }  
}
```



1. `getStringExtra(String key)` - 获取字符串值
2. `getIntExtra(String key, int defaultValue)` - 获取整数值
3. `getBooleanExtra(String key, boolean defaultValue)` - 获取布尔值
4. `getLongExtra(String key, long defaultValue)` - 获取长整数值
5. `getFloatExtra(String key, float defaultValue)` - 获取浮点数值
6. `getDoubleExtra(String key, double defaultValue)` - 获取双精度浮点数值
7. `getByteExtra(String key, byte defaultValue)` - 获取字节值
8. `getShortExtra(String key, short defaultValue)` - 获取短整数值
9. `getCharExtra(String key, char defaultValue)` - 获取字符值
10. `getParcelableExtra(String key)` - 获取实现了 `Parcelable` 接口的对象

11. `getSerializableExtra(String key)` - 获取实现了 `Serializable` 接口的对象
12. `getBundleExtra(String key)` - 获取 `Bundle` 对象

注意：对于基本数据类型（如整数、布尔值、浮点数等），`getXXXExtra` 方法需要一个默认值参数，以便在找不到指定键的情况下返回默认值。对于引用类型（如字符串、`Parcelable`、`Serializable` 等），当找不到指定键时，方法会返回 `null`。

```
int intValue = intent.getIntExtra("key", 0); // 如果找不到 "k
boolean boolValue = intent.getBooleanExtra("key", false); /
float floatValue = intent.getFloatExtra("key", 0.0f); // 如!
```

Bundle

MainActivity.java

```
// 创建一个 Intent 对象
Intent intent = new Intent("com.example.MY_BROADCAST_ACTION

// 将数据添加到 Intent 对象
intent.putExtra("key1", "value1");
intent.putExtra("key2", "value2");

// 发送广播
sendBroadcast(intent);
```

MyBroadcastReceiver.java

```
public class MyBroadcastReceiver extends BroadcastReceiver   
    @Override  
    public void onReceive(Context context, Intent intent) {  
        // 从 Intent 对象中获取包含所有额外数据的 Bundle 对象  
        Bundle extras = intent.getExtras();  
  
        // 从 Bundle 对象中提取特定数据  
        String value1 = extras.getString("key1");  
        String value2 = extras.getString("key2");  
  
        // 对获取到的数据进行操作  
        // ...  
    }  
}
```

多种数据类型的数据

MainActivity.java

```
private void sendCustomBroadcast() {   
    Intent intent = new Intent("com.example.CUSTOM_ACTION")  
  
    Bundle extras = new Bundle();  
    extras.putInt("key_int", 42);  
    extras.putBoolean("key_boolean", true);  
    extras.putString("key_string", "Hello, Bundle!");  
    // ... 添加其他类型的值  
  
    intent.putExtras(extras);  
  
    sendBroadcast(intent);  
}
```

MyBroadcastReceiver.java

```

public class MyBroadcastReceiver extends BroadcastReceiver
    @Override
    public void onReceive(Context context, Intent intent) {
        Bundle extras = intent.getExtras();
        if (extras != null) {
            int intValue = extras.getInt("key_int", 0);
            boolean boolValue = extras.getBoolean("key_bool");
            String strValue = extras.getString("key_string");
            // ... 获取其他类型的值

            Log.d("MyBroadcastReceiver", "Received int: " + intValue);
            Log.d("MyBroadcastReceiver", "Received boolean: " + boolValue);
            Log.d("MyBroadcastReceiver", "Received string: " + strValue);
            // ... 打印其他类型的值
        }
    }
}

```

extras.getInt 这个的，默认值是可有可无的。如果有的话，就是指定默认值，没有的话，就是默认值

🔗 有序广播（带最终接受者）

FinalReceiver.java

```

import android.content.BroadcastReceiver;
import android.content.Context;
import android.content.Intent;
import android.util.Log;

public class FinalReceiver extends BroadcastReceiver {
    @Override
    public void onReceive(Context context, Intent intent) {
        Log.d("FinalReceiver", "收到有序广播");
    }
}

```

MainActivity.java



```
package com.fu.tt;

import android.app.Activity;
import android.content.BroadcastReceiver;
import android.content.Context;
import android.content.Intent;
import android.content.IntentFilter;
import android.content.pm.PackageManager;
import android.util.Log;
import android.widget.Button;
import android.view.View;
import androidx.annotation.NonNull;
import androidx.appcompat.app.AppCompatActivity;
import androidx.core.app.ActivityCompat;
import androidx.core.content.ContextCompat;
import android.os.Bundle;
import android.widget.Toast;

import android.Manifest;

public class MainActivity extends AppCompatActivity {

    private ReceiverA receiverA;
    private ReceiverB receiverB;
    private BroadcastReceiver finalReceiver;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        // 创建一个最终接收者
        finalReceiver = new FinalReceiver();

        // 动态注册广播接收器
        registerReceivers();

        // 发送有序广播
        findViewById(R.id.send_ordered_broadcast_button).setOnClickListener()
        @Override
        public void onClick(View v) {
            Log.i("guangbo", "发送按钮点击");
            sendOrderedBroadcastWithFinalReceiver();
        }
    }
}
```

```

        }
    });
}

private void registerReceivers() {
    receiverA = new ReceiverA();
    IntentFilter filterA = new IntentFilter("com.exempl
    filterA.setPriority(100);
    registerReceiver(receiverA, filterA);

    receiverB = new ReceiverB();
    IntentFilter filterB = new IntentFilter("com.exempl
    filterB.setPriority(99);
    registerReceiver(receiverB, filterB);
}

private void sendOrderedBroadcastWithFinalReceiver() {
    Intent intent = new Intent("com.example.orderedbroa

    // 发送有序广播，添加最终接收者
    sendOrderedBroadcast(intent, null, finalReceiver, n
}

@Override
protected void onDestroy() {
    super.onDestroy();
    unregisterReceiver(receiverA);
    unregisterReceiver(receiverB);
}

protected void onResume()
{
    super.onResume();
}

@Override
protected void onPause() {
    super.onPause();
}
}

```

注意这个注销对于 finalReceiver 是不需要的

详细分析 `sendOrderedBroadcast(intent, null, finalReceiver, null, Activity.RESULT_OK, null, null);` 这一行代码

1. `intent` - 要发送的广播的 `Intent`。它包含了广播的 `action`、数据、类别等信息。
2. `null` - 一个字符串，用于指定发送广播所需的权限。在这个例子中，我们使用 `null` 表示发送广播不需要任何特定权限。
3. `finalReceiver` - 最终的广播接收者，它是一个 `BroadcastReceiver` 实例。当有序广播完成时，这个接收者的 `onReceive()` 方法将被调用。在这个例子中，我们创建了一个匿名的 `BroadcastReceiver` 实例作为最终接收者。
4. `null` - 一个 `Handler`，用于指定哪个线程应该处理最终接收者的 `onReceive()` 方法。在这个例子中，我们使用 `null` 表示使用默认的线程。当你发送一个有序广播时，广播接收者会按顺序处理广播。这个 `Handler` 参数可以指定最终接收者的 `onReceive()` 方法应该在哪个线程上运行。如果你设置为 `null`，那么默认情况下，接收者的 `onReceive()` 方法将在主线程（UI线程）上运行。例如，如果你想让最终接收者在子线程上运行，你可以创建一个 `HandlerThread` 和一个与之关联的 `Handler`，然后将这个 `Handler` 传递给 `sendOrderedBroadcast()` 方法，在大多数情况下，您可以将此参数设置为 `null`，这意味着最终接收者将在当前线程（通常是主线程）中执行 `onReceive()` 方法。这对于大多数简单的场景已经足够了，因为广播接收者的处理通常不会耗费太多时间。

然而，在某些情况下，您可能希望最终接收者在一个不同的线程中处理广播，以避免阻塞主线程。例如，如果您的应用程序需要在收到广播时执行一些耗时的操作（如文件读写、网络请求等），那么在主线程中执行这些操作可能会导致应用程序界面卡顿或不响应。

5. `Activity.RESULT_OK` - 一个整数，表示广播的初始结果代码。接收者可以根据需要修改这个结果代码。
6. `null` - 一个字符串，表示广播的初始结果数据。接收者可以根据需要修改这个结果数据。

7. `null` - 一个 `Bundle`，用于传递广播的初始额外数据。接收者可以根据需要修改这个额外数据。（4~7都是用来广播之间进行通信的）

注意这些参数只存在有序广播当中

🔗 有序广播_指定finalReceiver的运行线程（第四个参数）

我们知道第四个参数的作用是，用于指定哪个线程应该处理最终接收者的 `onReceive()` 方法。

FinalReceiver.java

```
package com.fu.tt;
import android.content.BroadcastReceiver;
import android.content.Context;
import android.content.Intent;
import android.util.Log;

public class FinalReceiver extends BroadcastReceiver {
    @Override
    public void onReceive(Context context, Intent intent) {
        Log.d("FinalReceiver", "开始处理耗时操作");

        // 模拟耗时操作，比如网络请求或者文件操作
        try {
            Thread.sleep(5000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        Log.d("FinalReceiver", "耗时操作完成");
    }
}
```

MainActivity.java

```
package com.fu.tt;
import android.app.Activity;
import android.content.BroadcastReceiver;
import android.content.Context;
import android.content.Intent;
```

```
import android.content.IntentFilter;
import android.content.pm.PackageManager;
import android.os.Handler;
import android.os.HandlerThread;
import android.util.Log;
import android.widget.Button;
import android.view.View;
import androidx.annotation.NonNull;
import androidx.appcompat.app.AppCompatActivity;
import androidx.core.app.ActivityCompat;
import androidx.core.content.ContextCompat;
import android.os.Bundle;
import android.widget.Toast;
import android.os.HandlerThread;

import android.Manifest;

public class MainActivity extends AppCompatActivity {

    private BroadcastReceiver finalReceiver;
    private Handler handler;
    private HandlerThread handlerThread;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        HandlerThread handlerThread = new HandlerThread("Fi
        handlerThread.start();
        handler = new Handler(handlerThread.getLooper());

        // 动态注册广播接收器
        registerReceivers();

        // 发送有序广播
        findViewById(R.id.send_ordered_broadcast_button).se
        @Override
        public void onClick(View v) {
```



```

        Log.i("guangbo", "发送按钮点击");
        sendOrderedBroadcastWithFinalReceiver();
    }
});
}

private void registerReceivers() {
    receiverA = new ReceiverA();
    IntentFilter filterA = new IntentFilter("com.exempl
    filterA.setPriority(100);
    registerReceiver(receiverA, filterA);

    receiverB = new ReceiverB();
    IntentFilter filterB = new IntentFilter("com.exempl
    filterB.setPriority(99);
    registerReceiver(receiverB, filterB);
}

private void sendOrderedBroadcastWithFinalReceiver() {
    Intent intent = new Intent("com.example.orderedbroa
    finalReceiver = new FinalReceiver();

    // 发送有序广播，添加最终接收者
    sendOrderedBroadcast(intent, null, finalReceiver, h
}

// 注销广播，关闭线程，并且是安全关闭（也就是说，他会在关闭之前，将没有
// 如果使用的 handlerThread.quit(); 就是直接关闭线程
@Override
protected void onDestroy() {
    super.onDestroy();
    handlerThread.quitSafely();
    unregisterReceiver(finalReceiver);
}


protected void onResume()
{
    super.onResume();
}

@Override
protected void onPause() {
    super.onPause();
}
}

```

```
}
```

分析代码

```
HandlerThread handlerThread = new HandlerThread("Fi   
handlerThread.start();  
handler = new Handler(handlerThread.getLooper());
```

1. `HandlerThread handlerThread = new HandlerThread("FinalReceiverThread");`
 - i. 这行代码创建了一个名为 "FinalReceiverThread" 的新 `HandlerThread` 实例。`HandlerThread` 是一个特殊的线程，它可以处理和执行任务。
2. `handlerThread.start();`
 - i. 这行代码启动了我们创建的 `HandlerThread`。当线程启动后，它会准备好处理任务
3. `handler = new Handler(handlerThread.getLooper());`
 - i. 这行代码创建了一个 `Handler` 实例，它是用来将任务发送到 `HandlerThread` 上执行的工具。我们需要让 `Handler` 知道它应该将任务发送到哪个线程上，所以我们将刚刚创建的 `handlerThread` 的 `Looper` 传递给新的 `Handler`。`Looper` 是一个处理消息队列（任务队列）的组件，它将任务分发到关联的线程上。

```
// 发送有序广播，添加最终接收者   
sendOrderedBroadcast(intent, null, finalReceiver, h
```

通过这样的方式，将 `finalReceiver` 的任务队列添加进 `handlerThread` 线程当中

（一般来说，`Handler` 的 `post()` 和 `postDelayed()` 方法用于将任务发送到与之关联的 `HandlerThread` 上。这个我们后面介绍）

🔗 带权限的广播

首先我们要知道，我们的某一次的广播并不是想每一个都收到，在之前，我们的做法是显示广播，但是显式管理起来，在比较多的情况之下，就很难维护了

给我们的广播上加上权限也是为了安全。特别是比较高的权限，需要签名才能够执行广播接收器。

它能够限制广播发送者，我给广播接收器设置了权限，只有发送者跟我有一样的权限，才能够执行广播接收器

它能够限制广播接受者，我给广播发送这设置了权限，只有接受者跟我一样的权限才能够，接受我发送的广播

🔗 权限的申请

```
<permission
    android:name="com.example.permission.MY_BROADCAST_PERMI
    android:protectionLevel="normal" />

<uses-permission android:name="com.example.permission.MY_BR
```

`<permission>`：用于声明自定义权限。当您想创建一个自己的权限，以便其他应用程序使用时，您需要在 Manifest 文件中声明 `<permission>`。这里您可以设置权限的名称、保护级别等属性

`<uses-permission>`：用于声明您的应用程序需要使用的权限。当您的应用程序需要使用某个权限（无论是 Android 系统权限还是其他应用程序提供的自定义权限）时，需要在 Manifest 文件中使用 `<uses-permission>` 标签，标签用于告诉系统您的应用程序需要使用某个权限。这可以是您自己定义的权限（通过 `<permission>` 标签声明的权限），也可以是 Android 系统中的预定义权限（如访问网络、访问存储等）

`<permission>` 用于声明您的应用程序提供的自定义权限，而 `<uses-permission>` 用于声明您的应用程序要要用哪些权限

```
android:protectionLevel="normal"
```



`android:protectionLevel` 是用于定义一个自定义权限的安全级别。它决定了其他应用程序在请求使用该权限时需要满足的条件。对于

`android:protectionLevel="normal"`，这是最低的安全级别，表示任何应用程序只需在其 `AndroidManifest.xml` 文件中使用 `<uses-permission>` 标签请求这个权限，就可以获得这个权限。

除了 "normal" 之外，还有其他几种 `protectionLevel`：

1. `dangerous`：表示权限可能会影响用户隐私或设备操作。请求此类权限的应用程序需要在运行时向用户请求权限，用户可以选择允许或拒绝。
2. `signature`：表示只有与声明此权限的应用程序具有相同签名的应用程序才能获得此权限。这意味着只有由相同开发者签名的应用程序才能共享这个权限。
3. `signatureOrSystem`：表示只有具有相同签名的应用程序或系统级应用程序才能获得此权限。这意味着只有由相同开发者签名的应用程序或者已经预安装在设备上的系统应用才能使用这个权限。

简而言之，`android:protectionLevel="normal"` 意味着这个自定义权限相对容易获得，任何应用程序只要在其 `manifest` 文件中声明对该权限的需求

`AndroidManifest.xml`



```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res
    xmlns:tools="http://schemas.android.com/tools">

    <!-- 声明一个权限 -->
    <!-- 使用权限 -->
    <permission
        android:name="com.example.permission.MY_BROADCAST_P
        android:protectionLevel="normal" />


    <uses-permission android:name="com.example.permission.M

    <application
        android:allowBackup="true"
        android:dataExtractionRules="@xml/data_extraction_r
        android:fullBackupContent="@xml/backup_rules"
        android:icon="@mipmap/ic_launcher"
        android:label="@string/app_name"
        android:roundIcon="@mipmap/ic_launcher_round"
        android:supportsRtl="true"
        android:theme="@style/Theme.TT"
        tools:targetApi="31">

        ....

    </application>
</manifest>
```

MainActivity.java



```
package com.fu.tt;
import android.content.BroadcastReceiver;
import android.content.Context;
import android.content.Intent;
import android.content.IntentFilter;
import android.util.Log;
import android.widget.Button;
import android.view.View;
```

```
import androidx.appcompat.app.AppCompatActivity;
import androidx.fragment.app.FragmentManager;
import androidx.fragment.app.FragmentTransaction;

import android.os.Bundle;
import android.widget.TextView;
import android.widget.Toast;
import android.content.SharedPreferences;

import android.content.ComponentName;

public class MainActivity extends AppCompatActivity {

    private CustomBroadcastReceiver customBroadcastReceiver

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        // 注册广播，设置权限
        customBroadcastReceiver = new CustomBroadcastReceiv
        IntentFilter filter = new IntentFilter("com.example
        registerReceiver(customBroadcastReceiver, filter,"c

        Button broadcastButton = findViewById(R.id.broadcas
        broadcastButton.setOnClickListener(new View.OnClickListener
        @Override
        public void onClick(View view) {
            sendCustomBroadcast();
        }
    });
}

// 发送广播，设置权限
private void sendCustomBroadcast() {
    Intent intent = new Intent("com.example.CUSTOM_ACTI
    sendBroadcast(intent,"com.example.permission.MY_BRO
}

@Override
protected void onDestroy() {
    super.onDestroy();
}
```

```

        unregisterReceiver(customBroadcastReceiver);
    }

    protected void onResume()
    {
        super.onResume();
    }

    @Override
    protected void onPause() {
        super.onPause();
    }
}

```

CustomBroadcastReceiver.java

```

package com.fu.tt;

import android.content.BroadcastReceiver;
import android.content.Context;
import android.content.Intent;
import android.util.Log;
import android.widget.Toast;

public class CustomBroadcastReceiver extends BroadcastReceiver {
    private static final String TAG = "CustomBroadcastRecei

    @Override
    public void onReceive(Context context, Intent intent) {
        Toast.makeText(context, "接受到权限广播", Toast.LENGTH
        Log.i(TAG, "Received custom broadcast.");
    }
}

```

🔗 一个关于权限的code案例

这是一个请求联系人权限的demo

Androidmanifest.xml



```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res
    xmlns:tools="http://schemas.android.com/tools">

<!-- 这就是我们想要申请的权限 -->
    <uses-permission android:name="android.permission.READ_

    <application
        android:allowBackup="true"
        android:dataExtractionRules="@xml/data_extraction_r
        android:fullBackupContent="@xml/backup_rules"
        android:icon="@mipmap/ic_launcher"
        android:label="@string/app_name"
        android:roundIcon="@mipmap/ic_launcher_round"
        android:supportsRtl="true"
        android:theme="@style/Theme.TT"
        tools:targetApi="31">

    . . .

    </application>
</manifest>
```



```
package com.fu.tt;
import android.content.pm.PackageManager;
import android.widget.Button;
import android.view.View;
import androidx.annotation.NonNull;
import androidx.appcompat.app.AppCompatActivity;
import androidx.core.app.ActivityCompat;
import androidx.core.content.ContextCompat;
import android.os.Bundle;
import android.widget.Toast;

// Manifest.permission.READ_CONTACTS 这个需要的import
```



```

import android.Manifest;

public class MainActivity extends AppCompatActivity {

    private static final int PERMISSION_REQUEST_READ Contac

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        Button requestPermissionButton = findViewById(R.id.
            requestPermissionButton.setOnClickListener(new View
                @Override
                public void onClick(View view) {
                    requestReadContactsPermission();
                }
            });
    }

    private void requestReadContactsPermission() {
        // 检查是不是已经拥有了读取联系人的权限
        if (ContextCompat.checkSelfPermission(this, Manifest
            != PackageManager.PERMISSION_GRANTED) {

            if (ActivityCompat.shouldShowRequestPermissionR
                Toast.makeText(this, "需要读取联系人权限来显示联
            }

            ActivityCompat.requestPermissions(this, new Str
                PERMISSION_REQUEST_READ_CONTACTS);
        } else {
            displayContacts();
        }
    }

    @Override
    public void onRequestPermissionsResult(int requestCode,
        super.onRequestPermissionsResult(requestCode, permi
        if (requestCode == PERMISSION_REQUEST_READ_CONTACTS
            if (grantResults.length > 0 && grantResults[0]
                displayContacts();
            } else {

```

```

        Toast.makeText(this, "权限被拒绝，无法显示联系人",
            Toast.LENGTH_SHORT).show();
    }

    private void displayContacts() {
        Toast.makeText(this, "读取并显示联系人列表...", Toast.LENGTH_SHORT).show();
        // 实际应用中，在这里实现读取联系人并显示的逻辑
    }

    @Override
    protected void onDestroy() {
        super.onDestroy();
    }

    protected void onResume() {
        super.onResume();
    }

    @Override
    protected void onPause() {
        super.onPause();
    }
}

```

activity_main.xml

```
<?xml version="1.0" encoding="utf-8"?>
<androidx.constraintlayout.widget.ConstraintLayout xmlns:an
    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">

    <Button
        android:id="@+id/request_permission_button"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="请求联系人权限"
        app:layout_constraintBottom_toBottomOf="parent"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintTop_toTopOf="parent" />

</androidx.constraintlayout.widget.ConstraintLayout>
```



现在来逐句分析一下这个代码

1. `<uses-permission`
`android:name="android.permission.READ_CONTACTS" />`
 - i. 这个是我们申请的权限，不是我们声明的权限。所以是 `uses-permission`
2. `android:allowBackup="true"`
 - i. `<application>` 标签里设置。它的作用是指示应用的数据是否可以进行备份和恢复。值为 `true` 表示允许备份，值为 `false` 表示不允许备份。然后在用户更换设备或重新安装应用时恢复。这可以帮助用户保留应用的设置、数据等信息
3. `android:dataExtractionRules="@xml/data_extraction_rules"`

- i. 它指定了一个XML资源文件，该文件包含了应用数据提取规则（备份，数据转移，一键换机这样的功能）。这些规则决定了哪些应用数据应该被包含在设备的全局备份和恢复过程

中，`@xml/data_extraction_rules` 表示规则文件位于应用的 `res/xml` 目录下，文件名为 `data_extraction_rules.xml`。在这个文件中，通过 `<include>` 和 `<exclude>` 来包含和排除一些文件夹

```
<data-extraction-rules>
  <include domain="file" path="shared_prefs" />
  <exclude domain="file" path="shared_prefs/exclud
  <include domain="file" path="databases" />
  <include domain="file" path="app_webview" />
  <include domain="file" path="app_textures" />
</data-extraction-rules>
```



- 4. `android:fullBackupContent="@xml/backup_rules"`（Android 12 开始就已经弃用，改用 `dataExtractionRules` 属性）

- i. 全量备份是指将应用程序的所有数据备份到云端（例如，Google Drive），以便在需要时进行恢复。全量备份可以在用户更换设备或者重置设备后，帮助用户恢复应用程序的数据，文件在 `xml/backup_rules.xml`。所以这两个属性，一个是负责数据转移，也就是一键换机的时候，使用，另一个是，负责云端备份，至于本地备份是开发者自己的行为。在新的Android当中的（12及其以上），这个属性已经删掉了，全部变成了 `dataExtractionRules` 这个属性

- 5. `android:label="@string/app_name"`

- i. 个例子中，它表示应用程序的名称将从 `res/values/strings.xml` 资源文件中引用的字符串值中获取，该字符串的名称是 `app_name`

- 6. `android:roundIcon="@mipmap/ic_launcher_round"`

- i. 是一个属性，用于指定应用程序的圆形图标。在这个例子中，它表示应用程序的圆形图标将从 `mipmap` 资源文件夹中引用的图像文件中获取，该图像的名称是 `res/values/ic_launcher_round` 注意这不是一个文件，而是一个文件夹，下面放了各种分辨率下的，图标的文件。

- 7. `android:supportsRtl="true"`

i. 是一个属性，表示您的应用程序支持从右到左（RTL）的布局方向。
这主要用于支持阿拉伯语、希伯来语等从右到左书写的语言

8. `tools:targetApi="31"` 这个属性用于指定当前应用的目标 API 级别，在这个例子中是 31。API 级别表示应用程序所针对的 Android 平台版本。这个只显示信息，不参与编译

先是检测是不是已经获取到了读取联系人的权限，如果没有就请求联系人的权限

```
private void requestReadContactsPermission() {  
    // 检查是不是已经拥有了读取联系人的权限  
    if (ContextCompat.checkSelfPermission(this, Manifest.  
        != PackageManager.PERMISSION_GRANTED) {  
  
        if (ActivityCompat.shouldShowRequestPermissionR  
            Toast.makeText(this, "需要读取联系人权限来显示联  
        }  
  
        ActivityCompat.requestPermissions(this, new Str  
            PERMISSION_REQUEST_READ_CONTACTS);  
    } else {  
        displayContacts();  
    }  
}
```

翻译

GRANTED：已授予

```
if (ContextCompat.checkSelfPermission(this, Manifest.permis  
    != PackageManager.PERMISSION_GRANTED)
```

`ContextCompat.checkSelfPermission(this,`
`Manifest.permission.READ_CONTACTS` 是用来检查是不是已经获取了权限，
获取与否会返回一个值，如果这个值 等于
`PackageManager.PERMISSION_GRANTED` 就说明已经获取了权限，如果不等于
就说明没有获取权限

```
if (ActivityCompat.shouldShowRequestPermissionRationale(  
    Toast.makeText(this, "需要读取联系人权限来显示联  
})
```

翻译

Rationale : 原因

这个是，是否应该显示请求权限的原因

这个时候就应该有人奇怪，为什么我之前就已经判断是不是有权限，有权限就获取权限，没有权限，就说明用户拒绝了，为什么还要是否应该显示请求权限的原因

因为有一种情况，当用户第一次安装并运行应用时，应用可能会请求某些权限。对于用户来说，可能会觉得突然弹出权限请求不明所以。如果直接请求权限，用户可能会对此感到困惑，并拒绝授予权限。这对应用的功能和用户体验可能会产生负面影响。 `shouldShowRequestPermissionRationale()` 方法的作用在于，当用户之前已经拒绝过权限请求时，这个方法会返回 `true`。这时，开发者可以在再次请求权限之前，向用户展示解释，阐述为什么应用需要这个权限。这样可以提高用户对权限请求的理解，增加用户同意授权的可能性。

需要注意的是， `shouldShowRequestPermissionRationale()` 方法仅在用户已经拒绝过权限请求的情况下返回 `true`。对于首次请求权限，或者用户在拒绝权限请求时勾选了 "不再询问" 选项，这个方法会返回 `false`。这就是为什么我们需要在请求权限之前，额外判断是否需要向用户展示解释的原因。

`ActivityCompat.shouldShowRequestPermissionRationale()` 方法返回值如下：

- 返回 `true`：当用户之前已经拒绝过权限请求，但没有勾选 "不再询问" 选项。在这种情况下，应用应当在请求权限之前向用户提供解释，让用户了解为什么应用需要这个权限。
- 返回 `false`：在以下情况下，该方法会返回 `false`
 - i. 用户首次安装应用，还没有请求过权限。

- ii. 用户在拒绝权限请求时勾选了 "不再询问" 选项。
- iii. 应用已经获得了相应权限。

```
ActivityCompat.requestPermissions(this, new Str   
PERMISSION_REQUEST_READ_CONTACTS);
```

这段代码的作用是请求 READ_CONTACTS 权限。当应用调用这个方法时，系统会弹出一个对话框，询问用户是否允许应用访问联系人信息。用户做出决定后，系统会调用 `onRequestPermissionsResult()` 方法，通知应用请求的结果。

`new String[]{Manifest.permission.READ_CONTACTS}`：一个包含要请求的权限的字符串数组。在这个例子中，我们只请求一个权限：`Manifest.permission.READ_CONTACTS`，用于读取联系人信息。

`PERMISSION_REQUEST_READ_CONTACTS` 是我们的自定义的标识符，用于区别不同的权限。

```
public void onRequestPermissionsResult(int requestCode,   
    super.onRequestPermissionsResult(requestCode, permi  
    if (requestCode == PERMISSION_REQUEST_READ_CONTACTS  
        if (grantResults.length > 0 && grantResults[0]  
            displayContacts();  
        } else {  
            Toast.makeText(this, "权限被拒绝，无法显示联系人  
        }  
    }  
}
```

这个代码的作用是显示我们的联系人，但是我们的权限授予是异步的，所以，我们使用这个权限的时候，不能够确定就已经赋予上了。所以需要检查一下

🔗 有序广播（结果码）

Activity.RESULT_OK 这个就是一个常量他就是 -1，然后传进去

在广播接收器当中的结束的位置，通过 setResultCode(200)；类似于这样的操作。

我们在任何一个位置，比方说原来的Activity，或者是下一个 OnReceive 的方法当中就能够通过 getResultCode() 的函数获取到，当前的 结果码，如果，我们获取到的结果码是 200，就说明，我们的这个已经执行完毕了

MainActivity.java

```
package com.fu.tt;
import android.app.Activity;
import android.content.BroadcastReceiver;
import android.content.ComponentName;
import android.content.Context;
import android.content.Intent;
import android.content.IntentFilter;
import android.content.pm.PackageManager;
import android.os.Handler;
import android.os.HandlerThread;
import android.util.Log;
import android.widget.Button;
import android.view.View;
import androidx.annotation.NonNull;
import androidx.appcompat.app.AppCompatActivity;
import androidx.core.app.ActivityCompat;
import androidx.core.content.ContextCompat;
import android.os.Bundle;
import android.widget.Toast;
import android.os.HandlerThread;

import android.Manifest;

public class MainActivity extends AppCompatActivity {

    private BroadcastReceiver finalReceiver;
    private MyOrderedBroadcastReceiver1 myOrderedBroadcastR
```



```
private MyOrderedBroadcastReceiver2 myOrderedBroadcastR
```

```
@Override
```

```
protected void onCreate(Bundle savedInstanceState) {  
    super.onCreate(savedInstanceState);  
    setContentView(R.layout.activity_main);
```

```
    // 注册广播
```

```
    registerOrderedReceivers();
```

```
    // 发送有序广播
```

```
    findViewById(R.id.send_ordered_broadcast_button).se
```

```
    @Override
```

```
    public void onClick(View v) {
```

```
        Log.i("guangbo", "发送按钮点击");
```

```
        Toast.makeText(MainActivity.this, "发送广播",  
            sendOrderedBroadcast();
```

```
    }
```

```
});
```

```
}
```

```
// 注册广播
```

```
private void registerOrderedReceivers() {
```

```
    IntentFilter filter = new IntentFilter("com.example  
    filter.setPriority(100);
```

```
    myOrderedBroadcastReceiver1 = new MyOrderedBroadcas  
    registerReceiver(myOrderedBroadcastReceiver1, filte
```

```
    filter.setPriority(50);
```

```
    myOrderedBroadcastReceiver2 = new MyOrderedBroadcas  
    registerReceiver(myOrderedBroadcastReceiver2, filte
```

```
}
```

```
@Override
```

```
protected void onDestroy() {
```

```
    super.onDestroy();
```

```
    unregisterReceiver(myOrderedBroadcastReceiver1);
```

```

        unregisterReceiver(myOrderedBroadcastReceiver2);
    }

    protected void onResume()
    {
        super.onResume();
    }

    @Override
    protected void onPause() {
        super.onPause();
    }

    private void sendOrderedBroadcast() {
        Intent intent = new Intent("com.example.ORDERED_ACT
        intent.setPackage(getPackageName());
        sendOrderedBroadcast(intent, null, new BroadcastRec
            @Override
            public void onReceive(Context context, Intent i
                int resultCode = getResultCode();
                Log.d("OrderedBroadcast", "Final Receiver:
            }
        }, null, Activity.RESULT_OK, null, null);
    }
}

```

OrderedBroadcastReceiver1.java



```
package com.fu.tt;

import android.app.Activity;
import android.content.BroadcastReceiver;
import android.content.Context;
import android.content.Intent;
import android.util.Log;

public class OrderedBroadcastReceiver1 extends BroadcastReceiver
    @Override
    public void onReceive(Context context, Intent intent) {
        int resultCode = getResultCode(); // 他没有什么作用，他
        if (resultCode == Activity.RESULT_OK) {
            Log.d("OrderedBroadcast", "Receiver 1: Result c
                setResultCode(100); // 设置结果码
        }
    }
}
```

OrderedBroadcastReceiver2.java



```
package com.fu.tt;

import android.app.Activity;
import android.content.BroadcastReceiver;
import android.content.Context;
import android.content.Intent;
import android.util.Log;

public class OrderedBroadcastReceiver2 extends BroadcastReceiver
    @Override
    public void onReceive(Context context, Intent intent) {
        int resultCode = getResultCode();
        if (resultCode == 100) {
            Log.d("OrderedBroadcast", "Receiver 2: Result c
                setResultCode(200);
        }
    }
}
```

在普通广播（非有序广播）中，广播接收者是异步执行的，且接收者之间没有先后顺序，因此也没有结果代码。普通广播通常用于通知多个接收者某个事件发生，而不关心接收者处理的结果或顺序。

🔗 有序广播（最后三个参数）

最后三个参数分别是

1. 结果码（Result Code）：结果码是一个整数值，用于表示广播处理的状态或结果。你可以通过 `setResultCode(int)` 方法在 `BroadcastReceiver` 中设置结果码，然后在后续的接收者或最终接收者中使用 `getResultCode()` 方法获取这个值。结果码主要用于表示广播处理过程中的某种状态，例如表示操作成功、失败或其他状态。
2. 结果数据（Result Data）：结果数据是一个字符串值，用于在广播接收者之间传递简单的文本信息。你可以通过 `setResultData(String)` 方法在 `BroadcastReceiver` 中设置结果数据，然后在后续的接收者或最终接收者中使用 `getResultData()` 方法获取这个值。结果数据适用于传递较简单的文本信息，例如修改后的文本消息等。
3. 额外数据（Extras）：额外数据是一个 `Bundle` 对象，其中可以包含多种数据类型（如字符串、整数、布尔值等）。你可以在发送广播时将数据存储在 `Intent` 的 `Extras` 中，然后在 `BroadcastReceiver` 中使用 `intent.getExtras()` 方法获取这些数据。如果需要在广播接收者之间修改或添加额外数据，可以通过 `setResultExtras(Bundle)` 方法设置新的 `Bundle` 对象，然后在后续的接收者或最终接收者中使用 `getResultExtras(boolean)` 方法获取这个 `Bundle`。额外数据适用于传递复杂的数据结构，例如包含多个键值对的 `Bundle` 对象。

这三个参数的存在意义主要是为了在广播接收者之间传递不同类型和复杂度的数据。结果码用于传递状态信息，结果数据用于传递简单的文本信息，而额外数据用于传递更复杂的数据结构。它们在一定程度上具有互补性，可以根据你的需求选择使用哪个参数来传递数据。需要注意的是，这些参数在有序广播中才能发挥作用，因为有序广播允许接收者之间传递修改后的数据。在普通广播中，你需要使用 `Intent` 对象中的 `getExtras()` 和其他类似方法获取相应的数据。

🔗 普通广播（将任务分发到其他线程执行）

普通广播中指定运行的线程：默认情况下，广播接收者的 `onReceive()` 方法在主线程中执行。如果您需要在其他线程中执行广播接收者的代码，可以在 `onReceive()` 方法中使用 `Handler`、`HandlerThread` 或其他并发工具将代码分发到其他线程。然而，这种方式并不会影响广播接收者的执行顺序，仅仅是将接收者的处理逻辑移到其他线程执行。

BackgroundTaskManager.java

```
package com.fu.tt;

import android.os.Handler;
import android.os.HandlerThread;

public class BackgroundTaskManager {
    private static BackgroundTaskManager instance;
    private HandlerThread handlerThread;
    private Handler handler;

    private BackgroundTaskManager() {
        handlerThread = new HandlerThread("BackgroundTaskMa");
        handlerThread.start();
        handler = new Handler(handlerThread.getLooper());
    }

    public static BackgroundTaskManager getInstance() {
        if (instance == null) {
            instance = new BackgroundTaskManager();
        }
        return instance;
    }

    public void postTask(Runnable task) {
        handler.post(task);
    }
}
```

MyBroadcastReceiver.java




```
package com.fu.tt;
import android.content.BroadcastReceiver;
import android.content.Context;
import android.content.Intent;
import android.os.Handler;
import android.os.HandlerThread;
import android.util.Log;
import android.widget.Toast;

public class MyBroadcastReceiver extends BroadcastReceiver
    @Override
    public void onReceive(final Context context, final Intent intent) {
        Log.i("MyBroadcastReceiver", "收到广播");

        BackgroundTaskManager.getInstance().postTask(new Runnable() {
            @Override
            public void run() {
                // 在这里执行后台任务
                Log.d("MyBroadcastReceiver", "后台任务执行中...");

                // 模拟耗时任务
                try {
                    Thread.sleep(2000);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
                Log.d("MyBroadcastReceiver", "后台任务完成");
            }
        });
    }
}
```



```
package com.fu.tt;
import android.app.Activity;
import android.content.BroadcastReceiver;
import android.content.ComponentName;
import android.content.Context;
import android.content.Intent;
import android.content.IntentFilter;
```

```
import android.content.pm.PackageManager;
import android.os.Handler;
import android.os.HandlerThread;
import android.util.Log;
import android.widget.Button;
import android.view.View;
import androidx.annotation.NonNull;
import androidx.appcompat.app.AppCompatActivity;
import androidx.core.app.ActivityCompat;
import androidx.core.content.ContextCompat;
import android.os.Bundle;
import android.widget.Toast;
import android.os.HandlerThread;

import android.Manifest;

public class MainActivity extends AppCompatActivity {

    private MyBroadcastReceiver myBroadcastReceiver;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        myBroadcastReceiver = new MyBroadcastReceiver();
        IntentFilter intentFilter = new IntentFilter("com.e
        registerReceiver(myBroadcastReceiver, intentFilter)

        // 发送有序广播
        findViewById(R.id.send_ordered_broadcast_button).se
        @Override
        public void onClick(View v) {
            Log.i("guangbo", "发送按钮点击");
            Toast.makeText(MainActivity.this, "发送广播",
            Intent intent = new Intent("com.example.BAC
            sendBroadcast(intent);
        }
    });
}
```

```

    }

    @Override
    protected void onDestroy() {
        super.onDestroy();
        unregisterReceiver(myBroadcastReceiver);
        BackgroundTaskManager.getInstance().quitSafely();
    }

    protected void onResume()
    {
        super.onResume();
    }

    @Override
    protected void onPause()
    {
        super.onPause();
    }
}

```

分析代码

```

public static BackgroundTaskManager getInstance() {
    if (instance == null) {
        instance = new BackgroundTaskManager();
    }
    return instance;
}

```



这个 `getInstance()` 方法是一个典型的单例模式的实现。单例模式（Singleton Pattern）是一种设计模式，它确保一个类只有一个实例，并提供一个全局访问点。在这个例子中，`BackgroundTaskManager` 类是一个单例类。

可以把 `MyBroadcastReceiver.java` 这个类拆开写，方便理解

```

package com.fu.tt;
import android.content.BroadcastReceiver;
import android.content.Context;

```




```
import android.content.Intent;
import android.os.Handler;
import android.os.HandlerThread;
import android.util.Log;
import android.widget.Toast;

public class MyBroadcastReceiver extends BroadcastReceiver
    // 接收到广播
    @Override
    public void onReceive(final Context context, final Intent intent) {
        handleBroadcast(context, intent);
    }

    private void handleBroadcast(Context context, Intent intent) {
        Log.i("MyBroadcastReceiver", "收到广播");

        // 生成多线程的任务
        Runnable task = createBackgroundTask();
        // 将多线程的任务添加给线程
        BackgroundTaskManager backgroundTaskManager = BackgroundTaskManager.getInstance();
        backgroundTaskManager.postTask(task);
    }

    // 返回一个 Runnable 的任务，任务加在重写的 run 函数当中
    private Runnable createBackgroundTask() {
        return new Runnable() {
            @Override
            public void run() {
                performBackgroundTask();
            }
        };
    }

    private void performBackgroundTask() {
        Log.d("MyBroadcastReceiver", "后台任务执行中...");
        // 当前运行在哪一个线程当中
        Log.d("MyBroadcastReceiver", "正在 : " + Thread.currentThread().getName());

        // 模拟耗时任务
        try {
            Thread.sleep(2000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

```
        Log.d("MyBroadcastReceiver", "后台任务完成");  
    }  
}
```

🔗 Activity之间的通信startActivityResult_onActivityResult

在前面的介绍当中，我们知道，广播能够做到两个Activity之间的通信，onActivityResult和startActivityResult 同样的能够做到。

🔗 广播的生命周期

播接收器（BroadcastReceiver）没有像 Activity、Service 或 Fragment 那样的典型生命周期。广播接收器的生命周期非常简短，因为它们的主要目的是接收应用程序之间或系统组件之间发送的广播消息，并对其进行响应。

当广播接收器接收到匹配的 Intent 时，系统会调用其 `onReceive()` 方法。在 `onReceive()` 方法内，可以处理接收到的 Intent，例如，检查 Intent 的操作、读取携带的数据等。在处理完广播后，广播接收器的实例会被销毁。

广播接收器的生命周期可以总结为以下步骤：

1. 注册：通过在 AndroidManifest.xml 中静态注册或在代码中动态注册，订阅特定的广播事件。
2. 接收：当广播事件发生时，系统会创建广播接收器的实例，并调用其 `onReceive()` 方法。
3. 处理：在 `onReceive()` 方法内处理接收到的 Intent。
4. 销毁：处理完广播后，广播接收器实例会被销毁。

需要注意的是，广播接收器不应执行耗时操作，因为在 `onReceive()` 方法中，有一个很短的执行时间限制（约 10 秒）。如果需要执行耗时操作，可以考虑在广播接收器内启动一个 Service 来处理。

🔗 Activity 的生命周期

1. 创建 (onCreate) : 当 Activity 实例被创建时调用。在这个阶段, 您可以初始化组件、设置布局、注册事件监听器等。
2. 启动 (onStart) : 当 Activity 变为可见时调用。此时, Activity 还不能与用户进行交互, 但可以执行准备工作。
3. 恢复 (onResume) : 当 Activity 准备好与用户进行交互时调用。此时, Activity 处于前台并获取到焦点。
4. 暂停 (onPause) : 当 Activity 失去焦点时调用。此时, Activity 仍然可见, 但无法与用户进行交互。
5. 停止 (onStop) : 当 Activity 完全不可见时调用。例如, 用户切换
6. 重启 (onRestart) : 当 Activity 从完全不可见状态重新变为可见时调用。在 onStop 之后, onRestart 会被触发, 然后是 onStart。
7. 销毁 (onDestroy) : 当 Activity 实例被销毁时调用。此时, 您应该释放资源、注销广播接收器等。在 Activity 结束或系统回收资源时, onDestroy 会被触发。

以一个简单的应用为例, 该应用包含两个 Activity: Activity1 和 Activity2。我们将详细描述用户在这个应用中执行的操作以及对应的 Activity 生命周期阶段。

1. 用户打开应用: 这时, Activity1 的 onCreate、onStart 和 onResume 方法依次被调用。Activity1 处于运行状态。
2. 用户点击按钮跳转到 Activity2: 此时, Activity1 的 onPause 方法被调用, 因为它失去了焦点。接着, Activity2 的 onCreate、onStart 和 onResume 方法被调用, 进入运行状态。此外, Activity1 的 onStop 方法会被调用, 因为它已完全不可见。
3. 用户点击返回按钮, 返回到 Activity1: 这时, Activity2 的 onPause 方法被调用。然后, Activity1 的 onRestart、onStart 和 onResume 方法被调用, 重新回到运行状态。接下来, Activity2 的 onStop 和 onDestroy 方法被调用, 因为用户已经离开了 Activity2。
4. 用户按下 Home 键, 将应用放到后台: 此时, Activity1 的 onPause 和 onStop 方法被调用, 因为它失去了焦点并完全不可见。
5. 用户通过最近应用列表重新打开应用: 这时, Activity1 的 onRestart、onStart 和 onResume 方法被调用, 重新回到运行状态。

6. 用户按下返回键或通过其他方式退出应用：此时，Activity1 的 onPause、onStop 和 onDestroy 方法被调用，完成整个生命周期。

🔗 多线程开发

在 Android 中，有以下几种常用的多线程编程方法：

1. **Thread 类**：直接使用 Java 中的 `Thread` 类创建一个新的线程。你可以通过继承 `Thread` 类并重写 `run()` 方法，或者通过传递一个 `Runnable` 对象来实现。这是一种基本的多线程编程方法，但需要注意线程间的同步和通信问题。
2. **AsyncTask 类**：`AsyncTask` 是一个 Android 提供的轻量级异步任务类，适用于执行较短时间的后台任务。`AsyncTask` 可以让你在后台线程上执行任务，然后在主线程上更新 UI。需要注意的是，`AsyncTask` 自 Android 11（API 级别 30）开始已被弃用，建议使用其他替代方案。
3. **HandlerThread 类**：`HandlerThread` 是一个可以在新线程中运行 `Handler` 的预定义类。它创建了一个新的线程，其中包含一个与其关联的 `Looper`。这个 `Looper` 可以用于创建 `Handler` 对象，从而实现在子线程中处理消息和运行任务。`HandlerThread` 非常适合于需要执行周期性任务或需要处理消息队列的场景。
4. **IntentService 类**：`IntentService` 是一个用于处理异步请求的服务类。通过发送 `Intent` 来启动服务，`IntentService` 会在一个工作线程上执行该请求，然后在完成后自动停止服务。`IntentService` 适用于执行一次性的后台任务，例如文件下载。
5. **ThreadPoolExecutor 类**：`ThreadPoolExecutor` 是 Java 并发库中的一个类，用于创建和管理线程池。它可以用于执行大量的短时间任务，这些任务可以在一个固定数量的线程上并发执行。线程池可以减少线程创建和销毁的开销，提高系统性能。
6. **RxJava**：RxJava 是一个响应式编程库，它可以让你使用观察者模式来处理异步任务。RxJava 提供了一系列操作符，让你可以方便地管理多线程、数据流和回调。
7. **Kotlin 协程**：如果你使用 Kotlin 进行 Android 开发，可以使用 Kotlin 协程来处理异步任务。协程是一种轻量级的线程管理方式，它可以让你编写简洁的异步代码，同时提供了结构化的并发。

🔗 多线程开发 (Thread 类)

Thread 类是 Java 提供的基本多线程编程工具。使用 Thread 类创建线程有两种方法：

1. 继承 Thread 类并重写 `run()` 方法；
2. 实现 Runnable 接口并将 Runnable 对象传递给 Thread 构造函数。

方法一：继承 Thread 类

1. 创建一个新类，继承自 Thread 类，并重写 `run()` 方法。

```
public class MyThread extends Thread {  
    @Override  
    public void run() {  
        // 在这里执行线程任务  
        for (int i = 0; i < 5; i++) {  
            System.out.println("MyThread: " + i);  
            try {  
                Thread.sleep(1000); // 休眠 1 秒  
            } catch (InterruptedException e) {  
                e.printStackTrace();  
            }  
        }  
    }  
}
```

2. 在需要创建线程的地方创建 `MyThread` 实例，并调用 `start()` 方法启动线程。

```
MyThread myThread = new MyThread();  
myThread.start();
```

完整的代码



```
package com.fu.tt;
import androidx.appcompat.app.AppCompatActivity;

import android.os.Bundle;
import android.util.Log;

public class MainActivity extends AppCompatActivity {
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        MyThread myThread = new MyThread();
        myThread.start();
    }

    // 自定义线程类，继承自 Thread 类
    private static class MyThread extends Thread {
        @Override
        public void run() {
            super.run();

            // 在这里执行线程任务
            for (int i = 0; i < 5; i++) {
                Log.d("MyThread", "线程运行中: " + i);

                try {
                    // 模拟耗时任务，线程休眠1秒
                    Thread.sleep(1000);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        }
    }

    @Override
    protected void onDestroy() {
        super.onDestroy();
    }

    protected void onResume()
    {
        super.onResume();
    }
}
```

```

    }

    @Override
    protected void onPause()
    {
        super.onPause();
    }
}

```

在这个例子中，线程会在 `run()` 方法中的任务执行完毕后自动退出。你可以看到 `run()` 方法里有一个循环，循环执行5次打印日志和休眠操作。当循环执行完毕后，`run()` 方法结束，线程自然就退出了。

方法二：实现 Runnable 接口（更加推荐）

1. 创建一个新类，实现 Runnable 接口，并重写 `run()` 方法。

```

public class MyRunnable implements Runnable {
    @Override
    public void run() {
        // 在这里执行线程任务
        for (int i = 0; i < 5; i++) {
            System.out.println("MyRunnable: " + i);
            try {
                Thread.sleep(1000); // 休眠 1 秒
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}

```

2. 在需要创建线程的地方创建 Thread 实例，将 `MyRunnable` 对象传递给 Thread 构造函数，并调用 `start()` 方法启动线程。

```
MyRunnable myRunnable = new MyRunnable();
Thread thread = new Thread(myRunnable);
thread.start();
```



完整的代码（比较不正常的写法）

```
package com.fu.tt;
import androidx.appcompat.app.AppCompatActivity;

import android.os.Bundle;
import android.util.Log;

public class MainActivity extends AppCompatActivity implements
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        Thread myThread = new Thread(this);
        myThread.start();
    }

    // 自定义线程类，继承自 Thread 类
    private static class MyThread extends Thread {
        @Override
        public void run() {
            super.run();

            // 在这里执行线程任务
            for (int i = 0; i < 5; i++) {
                Log.d("MyThread", "线程运行中: " + i);

                try {
                    // 模拟耗时任务，线程休眠1秒
                    Thread.sleep(1000);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        }
    }
}
```




```

    }

}

@Override
protected void onDestroy() {
    super.onDestroy();
}

protected void onResume()
{
    super.onResume();
}

@Override
protected void onPause()
{
    super.onPause();
}

@Override
public void run() {
    for (int i = 0; i < 100; i++) {
        Log.d("MyRunnable", "MyRunnable is running: " +
            i);
        try {
            Thread.sleep(1000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
}
}

```

完整的代码（比较正常的写法）

MainActivity.java

```

package com.fu.tt;
import androidx.appcompat.app.AppCompatActivity;

import android.os.Bundle;
import android.util.Log;

```



```

public class MainActivity extends AppCompatActivity {
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        MyRunnable myRunnable = new MyRunnable();
        Thread myThread = new Thread(myRunnable);
        myThread.start();
    }

    // 自定义线程类，继承自 Thread 类
    private static class MyThread extends Thread {
        @Override
        public void run() {
            super.run();

            // 在这里执行线程任务
            for (int i = 0; i < 5; i++) {
                Log.d("MyThread", "线程运行中: " + i);

                try {
                    // 模拟耗时任务，线程休眠1秒
                    Thread.sleep(1000);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        }
    }

    @Override
    protected void onDestroy() {
        super.onDestroy();
    }

    protected void onResume()
    {
        super.onResume();
    }

    @Override
    protected void onPause()
    {

```

```
        super.onPause();  
    }  
}
```

MyRunnable.java

```
package com.fu.tt;  
  
import android.util.Log;  
  
public class MyRunnable implements Runnable {  
    @Override  
    public void run() {  
        for (int i = 0; i < 100; i++) {  
            Log.d("MyRunnable", "MyRunnable is running: " +  
                i);  
            try {  
                Thread.sleep(1000);  
            } catch (InterruptedException e) {  
                e.printStackTrace();  
            }  
        }  
    }  
}
```

这两种方式有什么区别，为什么要大费周章的使用第二种方式？

1. 继承关系

方法一是通过继承 Thread 类来实现多线程，方法二是通过实现 Runnable 接口来实现多线程。Java 语言不支持多重继承，因此，如果一个类已经继承了其他类，就无法再继承 Thread 类。在这种情况下，实现 Runnable 接口就成了实现多线程的唯一选择。

2. 资源共享

当使用继承 Thread 类的方法创建线程时，每个线程都拥有自己的实例变量。这意味着，每个线程之间是相互独立的，它们不共享实例变量。而使用 Runnable 接口的方法，由于多个线程共享同一个 Runnable 实例，因此，它们可以共享实例变量，更容易实现资源共享。

3. 代码复用

实现 Runnable 接口的方式更具代码复用性。当一个类实现了 Runnable 接口，它可以作为一个线程任务，也可以作为一个普通的类被其他类调用。而继承 Thread 类的方式则限制了代码复用性，因为它是一个线程类，只能用于线程任务。

4. 灵活性

实现 Runnable 接口允许您更灵活地组织代码。例如，可以创建一个工作类，实现 Runnable 接口，并在该类中实现多个任务。然后，可以通过创建多个线程，为每个线程分配一个任务，以实现并发执行。这样的代码组织更加清晰和灵活。

总结起来，使用实现 Runnable 接口的方式更具灵活性、代码复用性和资源共享性。在实际开发中，实现 Runnable 接口的方式比继承 Thread 类更受推荐。当然，根据具体需求和场景，您可以选择适合您的方法。

接下来，我们通过代码——来看他的优势

🔗 多线程开发（Thread 类_Runnable的继承优势）

以下是一个使用 Runnable 的例子，该例子展示了当一个类已经继承了另一个类时，为什么需要实现 Runnable 接口以实现多线程

首先，创建一个名为 BaseClass.java 的新文件，并定义 BaseClass 类：

```
public class BaseClass {  
    public void baseMethod() {  
        Log.d("BaseClass", "This is a method in the base cl  
    }  
}
```

然后创建一个名为 `MyExtendedClass.java` 的新文件，继承 `BaseClass` 类并实现 `Runnable` 接口：

```
public class MyExtendedClass extends BaseClass implements R  
    @Override  
    public void run() {  
        Log.d("MyExtendedClass", "MyExtendedClass is runnin  
        try {  
            Thread.sleep(2000);  
        } catch (InterruptedException e) {  
            e.printStackTrace();  
        }  
    }  
}
```

接下来，在你的主要活动或其他类中，你可以创建一个 `MyExtendedClass` 实例，启动一个新线程并调用 `baseMethod()`：

```
public class MainActivity extends AppCompatActivity {  
    @Override  
    protected void onCreate(Bundle savedInstanceState) {  
        super.onCreate(savedInstanceState);  
        setContentView(R.layout.activity_main);  
  
        MyExtendedClass myExtendedClass = new MyExtendedCla  
        Thread myThread = new Thread(myExtendedClass);  
        myThread.start();  
        myExtendedClass.baseMethod();  
    }  
}
```

在这个例子中，`MyExtendedClass` 类继承了 `BaseClass` 并实现了 `Runnable` 接口。由于 Java 不支持多重继承，我们不能同时继承 `BaseClass` 和 `Thread`。但是，我们可以实现 `Runnable` 接口，从而让 `MyExtendedClass` 具有多线程功能，同时保留对 `BaseClass` 的继承。这使得 `MyExtendedClass` 可以访问 `BaseClass` 中的方法，如 `baseMethod()`。

🔗 多线程开发（Thread 类_Runnable的资源共享优势）

MainActivity.java

```
package com.fu.tt;
import androidx.appcompat.app.AppCompatActivity;

import android.os.Bundle;
import android.util.Log;
import android.widget.TextView;

public class MainActivity extends AppCompatActivity {
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        // 他们对于同一个 sharedCounter 的counter 数值进行增加，一
        Counter sharedCounter = new Counter(0);

        Thread thread1 = new Thread(new IncrementTask(sharedCounter));
        Thread thread2 = new Thread(new IncrementTask(sharedCounter));

        thread1.start();
        thread2.start();

        // 他的作用是阻塞当前的主线程，也就是，UI线程，目的是，等待我们
        // 如果阻塞，可能两边线程各自加了50，总数就是100。还以为没有进
        try {
            thread1.join();
            thread2.join();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

```
// 这个作用是更新UI，加上这个的目的是确保我们的更新，UI的工作是在主线程
    runOnUiThread(new Runnable() {
        @Override
        public void run() {
            TextView textView = findViewById(R.id.textV
            textView.setText("Counter value: " + shared
        }
    });
}

@Override
protected void onDestroy() {
    super.onDestroy();
}

protected void onResume()
{
    super.onResume();
}

@Override
protected void onPause()
{
    super.onPause();
}
}
```

Counter.java

```
package com.fu.tt;

public class Counter {
    private int value;

    public Counter(int initialValue) {
        value = initialValue;
    }

    public void increment() {
        value++;
    }

    public int getValue() {
        return value;
    }
}
```



IncrementTask.java

```
package com.fu.tt;

public class IncrementTask implements Runnable {
    private Counter counter;

    public IncrementTask(Counter counter) {
        this.counter = counter;
    }

    @Override
    public void run() {
        for (int i = 0; i < 100; i++) {
            counter.increment();
        }
    }
}
```



这个是保证我们的更改 UI 的代码一定执行在 UI 主线程上面

MainActivity.java

```
package com.fu.tt;
import androidx.appcompat.app.AppCompatActivity;

import android.os.Bundle;
import android.util.Log;
import android.view.View;
import android.widget.Button;
import android.widget.TextView;

public class MainActivity extends AppCompatActivity {
    private TextView textView;
    private Button button;
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        button = findViewById(R.id.button);
        textView = findViewById(R.id.textView);

        button.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View v) {
                performLongRunningTaskAndUpdateTextView();
            }
        });
    }

    @Override
    protected void onDestroy() {
        super.onDestroy();
    }

    protected void onResume()
    {
        super.onResume();
    }
}
```

```
@Override
protected void onPause()
{
    super.onPause();
}

private void performLongRunningTask() {
    new Thread(new Runnable() {
        @Override
        public void run() {
            simulateLongRunningTask();
            updateUIAfterTaskCompletion();
            runOnUiThread(new Runnable() {
                @Override
                public void run() {

                }
            });
        }
    }).start();
}

private void simulateLongRunningTask() {
    // 模拟耗时任务（如网络请求）
    try {
        Thread.sleep(2000);
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
}

private void updateUIAfterTaskCompletion() {

    textView.setText("Task completed");
}

private void performLongRunningTaskAndUpdateTextView()
    new Thread(new Runnable() {
        @Override
        public void run() {
            // 模拟耗时操作
            try {
                Thread.sleep(2000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }).start();
}
```

```

    }

    // 耗时操作完成, 更新 TextView
    runOnUiThread(new Runnable() {
        @Override
        public void run() {
            textView.setText("Task completed!")
        }
    });
}
}).start();
}
}

```

在这个案例当中，我们把更新UI的操作移步到了，一个线程当中，但是，在执行的时候，又通过 `runOnUiThread` 这样的方式，对我们的，更新代码强制在主线程当中运行（但是你会发现，即便是将 `runOnUiThread` 这个部分的代码删掉，程序依然会成功的运行，并且没有任何的错误，这个错误，在不同的设备上的是否触发，有随机性）

🔗 使用 Lambda 表达式简化 Runnable

```

Runnable runnable = () -> {
    System.out.println("Hello from the Runnable implemented");
};

```

```

new Thread(() -> {
    // 这里执行任务
}).start();

```

其实就是省掉了

```
@Override
public void run() {
    ...
}
```



变成了

```
() -> textView.setText("Task completed!")
```



对比下面的两个代码

```
package com.fu.tt;
import androidx.appcompat.app.AppCompatActivity;

import android.os.Bundle;
import android.util.Log;
import android.view.View;
import android.widget.Button;
import android.widget.TextView;

public class MainActivity extends AppCompatActivity {
    private TextView textView;
    private Button button;
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        button = findViewById(R.id.button);
        textView = findViewById(R.id.textView);

        button.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View v) {
                performLongRunningTaskAndUpdateTextView();
            }
        });
    }
}
```



```
@Override
protected void onDestroy() {
    super.onDestroy();
}

protected void onResume()
{
    super.onResume();
}

@Override
protected void onPause()
{
    super.onPause();
}

private void performLongRunningTask() {
    new Thread(new Runnable() {
        @Override
        public void run() {
            simulateLongRunningTask();
            updateUIAfterTaskCompletion();
            runOnUiThread(new Runnable() {
                @Override
                public void run() {

                }
            });
        }
    }).start();
}

private void simulateLongRunningTask() {
    // 模拟耗时任务（如网络请求）
    try {
        Thread.sleep(2000);
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
}

private void updateUIAfterTaskCompletion() {
```

```
        textView.setText("Task completed");
    }

    private void performLongRunningTaskAndUpdateTextView()
    {
        new Thread(new Runnable() {
            @Override
            public void run() {
                // 模拟耗时操作
                try {
                    Thread.sleep(2000);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }

                // 耗时操作完成, 更新 TextView
                runOnUiThread(new Runnable() {
                    @Override
                    public void run() {
                        textView.setText("Task completed!")
                    }
                });
            }
        }).start();
    }
}
```



```
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 {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        Button button = findViewById(R.id.button);
        TextView textView = findViewById(R.id.textView);

        button.setOnClickListener(view -> {
            // 使用 Lambda 表达式实现 Runnable
            Runnable task = () -> {
                try {
                    // 模拟一个耗时任务
                    Thread.sleep(3000);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }

                // 更新 UI 必须在 UI 线程中进行
                runOnUiThread(() -> textView.setText("Task
            });

            // 创建并启动一个新线程来执行任务
            new Thread(task).start();
        });
    }
}
```

🔗 使用线程池执行 Runnable 任务_FixedThreadPool

线程池是一种管理线程的机制，它允许您事先创建多个线程并将它们保存在一个池中。当需要执行任务时，线程池会从池中获取一个空闲的线程来执行任务。任务完成后，线程返回池中，等待下一个任务。线程池的好处在于减少了线程的创建和销毁开销，同时能更有效地管理系统资源。

FixedThreadPool: 这是一个拥有固定数量线程的线程池。任务提交时，如果线程池中有空闲线程，它将被用于执行任务。如果没有空闲线程，任务将被排队等待。



```
package com.fu.tt;
import androidx.appcompat.app.AppCompatActivity;

import android.os.Bundle;
import android.util.Log;
import android.view.View;
import android.widget.Button;
import android.widget.TextView;

import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

public class MainActivity extends AppCompatActivity {
    private TextView textView;
    private Button button;
    private ExecutorService executorService;

    private static final String TAG = "ThreadPoolExample";

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        Button button = findViewById(R.id.button);
        button.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View v) {
                executeTasks();
            }
        });
    }

    private void executeTasks() {
```



```
ExecutorService = Executors.newFixedThreadPool(4);

Runnable task1 = new Runnable() {
    @Override
    public void run() {
        for (int i = 0; i < 10; i++) {
            Log.d(TAG, "Task 1 - Number: " + i);
        }
    }
};

Runnable task2 = new Runnable() {
    @Override
    public void run() {
        for (int i = 0; i < 10; i++) {
            Log.d(TAG, "Task 2 - Number: " + i);
        }
    }
};

Runnable task3 = new Runnable() {
    @Override
    public void run() {
        for (int i = 0; i < 10; i++) {
            Log.d(TAG, "Task 3 - Number: " + i);
        }
    }
};

Runnable task4 = new Runnable() {
    @Override
    public void run() {
        for (int i = 0; i < 10; i++) {
            Log.d(TAG, "Task 4 - Number: " + i);
        }
    }
};

// 提交任务到线程池
executorService.execute(task1);
executorService.execute(task2);
executorService.execute(task3);
executorService.execute(task4);

// 关闭线程池（在完成所有任务后）
```

```

        executorService.shutdown();
    }

    @Override
    protected void onDestroy() {
        super.onDestroy();
        // 关闭线程池，释放资源
        executorService.shutdown();
    }

    protected void onResume()
    {
        super.onResume();
    }

    @Override
    protected void onPause()
    {
        super.onPause();
    }
}

```

🔗 使用线程池执行 Runnable 任务_CachedThreadPool

CachedThreadPool 是 Java 提供了一种线程池实现，它具有动态调整线程数量的特点。当需要执行新任务时，**CachedThreadPool** 会尝试重用之前创建的空闲线程。如果没有空闲线程可用，**CachedThreadPool** 会创建一个新线程来执行任务。当线程完成任务并变为空闲时，如果在一定时间内（默认为 60 秒）没有新任务分配给该线程，线程将被终止并从线程池中移除。这种特性使得 **CachedThreadPool** 在执行大量短时任务时表现出较高的性能。

MainActivity.java

```

package com.fu.tt;
import androidx.appcompat.app.AppCompatActivity;

import android.os.Bundle;
import android.util.Log;

```



```
import android.view.View;
import android.widget.Button;
import android.widget.TextView;

import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

public class MainActivity extends AppCompatActivity {
    private TextView textView;
    private Button button;
    private ExecutorService executorService;

    private static final String TAG = "CachedThreadPoolExam

@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);

    executorService = Executors.newFixedThreadPool(4);

    Button button = findViewById(R.id.button);
    button.setOnClickListener(new View.OnClickListener() {
        @Override
        public void onClick(View v) {
            executeTasks();
        }
    });
}

private void executeTasks() {
    // 创建线程池
    ExecutorService executorService = Executors.newCach

    // 创建任务
    Runnable task1 = () -> {
        for (int i = 0; i < 10; i++) {
            Log.d(TAG, "Task 1 - Number: " + i);
        }
    };

    Runnable task2 = () -> {
        for (int i = 0; i < 10; i++) {
            Log.d(TAG, "Task 2 - Number: " + i);
```

```

        }
    };

    Runnable task3 = () -> {
        for (int i = 0; i < 10; i++) {
            Log.d(TAG, "Task 3 - Number: " + i);
        }
    };

    // 提交任务到线程池
    executorService.execute(task1);
    executorService.execute(task2);
    executorService.execute(task3);

    // 关闭线程池（在完成所有任务后）
    executorService.shutdown();
}

@Override
protected void onDestroy() {
    super.onDestroy();
    // 关闭线程池，释放资源
    executorService.shutdown();
}

protected void onResume()
{
    super.onResume();
}

@Override
protected void onPause()
{
    super.onPause();
}
}

```

`ScheduledThreadPool` 是 Java 提供的一种线程池实现，它允许您在指定的延迟后执行任务，或者按照固定的时间间隔定期执行任务。

```
import androidx.appcompat.app.AppCompatActivity;
import android.os.Bundle;
import android.util.Log;
import android.view.View;
import android.widget.Button;

import java.util.concurrent.Executors;
import java.util.concurrent.ScheduledExecutorService;
import java.util.concurrent.TimeUnit;

public class MainActivity extends AppCompatActivity {

    private static final String TAG = "ScheduledThreadPoolE

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        Button button = findViewById(R.id.button);
        button.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View v) {
                scheduleTasks();
            }
        });
    }

    private void scheduleTasks() {
        ScheduledExecutorService scheduledExecutorService =

        // 延迟任务
        Runnable delayedTask = () -> Log.d(TAG, "Delayed ta
        scheduledExecutorService.schedule(delayedTask, 3, T

        // 定期任务
        Runnable periodicTask = () -> Log.d(TAG, "Periodic
        scheduledExecutorService.scheduleAtFixedRate(period
    }
}
```

翻译

scheduled：计划中的

Fixed：固定的

Rate：频率

period：周期

```
Runnable delayedTask = () -> Log.d(TAG, "Delayed ta  
scheduledExecutorService.schedule(delayedTask, 3, T
```

delayedTask：任务

3 TimeUnit.SECONDS：延迟 3 秒钟

```
// 定期任务  
Runnable periodicTask = () -> Log.d(TAG, "Periodic  
scheduledExecutorService.scheduleAtFixedRate(period
```

1. `periodicTask`：这是要执行的任务，它实现了 `Runnable` 接口。
2. `5`：这是一个整数值，表示任务的**初始延迟**。这意味着任务将在 5 秒后开始执行。
3. `2`：这是一个整数值，表示任务执行之间的**固定周期**。这意味着任务开始执行后，每隔 2 秒将再次执行一次。
4. `TimeUnit.SECONDS`：这是一个枚举值，表示时间单位。在这个例子中，它表示初始延迟和固定周期的时间单位都是秒。

如果我们延迟 3 小时，但是每隔秒钟执行一次，应该怎么办呢？也就是说，延迟时间和固定周期的时间单位不一样

```
ScheduledExecutorService scheduledExecutorService = Executors.  
Runnable periodicTask = new Runnable() {  
    @Override  
    public void run() {  
        // 你的任务代码  
    }  
};  
  
long initialDelay = TimeUnit.HOURS.toSeconds(3); // 将 3 小时  
scheduledExecutorService.scheduleAtFixedRate(periodicTask,
```

🔗 使用线程池执行 Runnable 任务_SingleThreadExecutor

`SingleThreadExecutor` 是一个线程池实现，它只使用一个工作线程来执行任务。这个线程池的主要特点是任务会按照提交顺序依次执行。因为它只有一个线程，所以能保证任务之间的顺序性，不会出现多线程执行时的竞争问题。这种线程池适用于需要顺序执行的任务场景。

MainActivity.java

```
package com.fu.tt;  
import androidx.appcompat.app.AppCompatActivity;  
  
import android.os.Bundle;  
import android.util.Log;  
import android.view.View;  
import android.widget.Button;  
import android.widget.TextView;  
  
import java.util.concurrent.ExecutorService;  
import java.util.concurrent.Executors;  
  
public class MainActivity extends AppCompatActivity {  
    private TextView textView;  
    private Button button;  
    private ExecutorService executorService;
```

```

private ExecutorService singleThreadExecutor;

@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);

    singleThreadExecutor = Executors.newSingleThreadExe

    Button button = findViewById(R.id.button);
    button.setOnClickListener(new View.OnClickListener(
        @Override
        public void onClick(View v) {
            singleThreadExecutorTasks();
        }
    ));
}

private void singleThreadExecutorTasks() {
    Runnable task1 = new Runnable() {
        @Override
        public void run() {
            Log.i("SingleThreadExecutor", "Task 1 is ru
        }
    };

    Runnable task2 = new Runnable() {
        @Override
        public void run() {
            Log.i("SingleThreadExecutor", "Task 2 is ru
        }
    };

    Runnable task3 = new Runnable() {
        @Override
        public void run() {
            Log.i("SingleThreadExecutor", "Task 3 is ru
        }
    };

    singleThreadExecutor.execute(task1);
    singleThreadExecutor.execute(task2);
    singleThreadExecutor.execute(task3);
}

```



```
@Override
protected void onDestroy() {
    super.onDestroy();
    // 关闭线程池，释放资源
    singleThreadExecutor.shutdown();
}

protected void onResume()
{
    super.onResume();
}


@Override
protected void onPause()
{
    super.onPause();
}
}
```

这个是依次执行的

🔗 多线程_Android 中与其他组件配合使用 Runnable

Handler 是 Android 中一种非常重要的组件，它可以让你发送和处理消息或者 Runnable 对象到应用程序的主线程的 MessageQueue 中。每个 Handler 实例都与一个线程和这个线程的消息队列相关联。

你可以创建一个新的 Handler，然后提供一个 Runnable 对象给它执行。Handler 会将这个 Runnable 对象添加到消息队列中，等到轮到这个 Runnable 对象执行时，Handler 就会调用 Runnable 的 run() 方法。



```
package com.fu.tt;

import android.os.Bundle;
import android.os.Handler;
import android.util.Log;
import androidx.appcompat.app.AppCompatActivity;

public class MainActivity extends AppCompatActivity {
    private Handler mHandler;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        mHandler = new Handler();
        mHandler.postDelayed(mRunnable, 5000);
    }

    private final Runnable mRunnable = new Runnable() {
        @Override
        public void run() {
            Log.d("MainActivity", "This message is logged a
        }
    };
}
```

Handler 类提供了一些方法来管理消息和 Runnable 对象。以下是一些最常用的方法：

1. `post(Runnable r)` : 将 Runnable 对象加入到消息队列中，当轮到它执行时，就会调用 Runnable 的 `run()` 方法。
2. `postDelayed(Runnable r, long delayMillis)` : 这个方法和 `post()` 方法很类似，但是它允许你设置一个延迟时间。Runnable 对象会在延迟了 `delayMillis` 毫秒之后执行。
3. `postAtTime(Runnable r, long uptimeMillis)` : 这个方法让你能够设置一个特定的时间，当系统的“正常运行时间”达到这个时间时，Runnable 对象就会被执行。系统的“正常运行时间”是一种时间计数，从系统启动开始，包括深度睡眠时间。
4. `sendEmptyMessage(int what)` : 这个方法发送一个包含了一个整型值的

空消息。

5. `sendMessage(Message msg)`: 这个方法发送一个 `Message` 对象。
`Message` 对象包含了一个描述和任意的数据对象。
6. `sendMessageDelayed(Message msg, long delayMillis)`: 这个方法
和 `sendMessage()` 方法类似, 但是它允许你设置一个延迟时间。`Message`
对象会在延迟了 `delayMillis` 毫秒之后发送。
7. `sendMessageAtTime(Message msg, long uptimeMillis)`: 这个方法让
你能够设置一个特定的时间, 当系统的“正常运行时间”达到这个时间时,
`Message` 对象就会被发送。
8. `removeCallbacks(Runnable r)`: 这个方法移除所有挂起的 `posts` 的
`Runnable` 对象和 `associated` 的消息。
9. `removeMessages(int what)`: 这个方法移除所有的 `what` 值为 `what` 参数
值的消息。

`Handler` 类还有其他一些方法, 但这些是最常用的。不同的方法可以适用于不同的场景, 你可以根据你的需求选择合适的方法。

View 在 Android 中, `View` 是用户界面元素的基础类, 比如按钮和文本框等等。在主线程中, 我们经常需要更改 `View` 的状态, 比如更改文本框的文本或者改变按钮的颜色。然而, 这些操作如果在子线程中执行的话, 会抛出 `CalledFromWrongThreadException` 异常, 因为 Android 的 UI 组件不是线程安全的, 不允许在子线程中直接操作。

这时, 我们就可以用 `View.post(Runnable)` 方法来解决这个问题。这个方法可以保证 `Runnable` 中的代码在主线程中执行, 从而可以安全地操作 UI 组件。下面是一个使用 `View.post(Runnable)` 来更改文本框文本的例子



```
public class MainActivity extends AppCompatActivity {  
    private TextView textView;  
  
    @Override  
    protected void onCreate(Bundle savedInstanceState) {  
        super.onCreate(savedInstanceState);  
        setContentView(R.layout.activity_main);  
        textView = (TextView) findViewById(R.id.textview);  
  
        new Thread(new Runnable() {  
            @Override  
            public void run() {  
                // 模拟耗时操作  
                try {  
                    Thread.sleep(2000);  
                } catch (InterruptedException e) {  
                    e.printStackTrace();  
                }  
  
                // 使用 View.post(Runnable) 在主线程中更改文本机  
                textView.post(new Runnable() {  
                    @Override  
                    public void run() {  
                        textView.setText("Text updated.");  
                    }  
                });  
            }  
        }).start();  
    }  
}
```

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical"
    android:gravity="center"
    android:padding="16dp">

    <Button
        android:id="@+id/startWorkButton"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Start Work" />

    <TextView
        android:id="@+id/textview"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Hello World!" />

</LinearLayout>
```



🔗 Handler_通信

Handler 的作用是将任务投递到指定的线程中执行。它并不负责创建或管理线程，而是在已有的线程（如主线程或子线程）中调度任务。同时，Handler 可以携带一些信息，以便在任务执行过程中使用

Handler 是 Android 中一个非常有用的组件，它允许您在不同线程之间进行**通信和任务调度**。Handler 主要用于在后台线程执行任务并将结果传递回主线程（UI线程）。

翻译

Handler：处理程序

Looper：循环器

下面这个代码演示如何在后台线程执行任务并更新 UI。在这个示例中，我们将创建一个简单的计时器，在后台线程中每隔一秒更新计数器，并将其显示在 TextView 中

activity_layout.xml

```
<TextView
    android:id="@+id/textView"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="Count: 0" />
```



Mainactivity.java

```
package com.fu.tt;
import androidx.appcompat.app.AppCompatActivity;

import android.os.Bundle;
import android.os.Handler;
import android.os.Looper;
import android.util.Log;
import android.view.View;
import android.widget.Button;
import android.widget.TextView;

import java.util.concurrent.ExecutorService;

public class MainActivity extends AppCompatActivity {
    private TextView textView;
    private Button button;
    private Handler handler;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        handler = new Handler(Looper.getMainLooper());
```



```

        Button button = findViewById(R.id.button);
        button.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View v) {
                startCounter();
            }
        });
    }

    private void startCounter() {
        new Thread(new Runnable() {
            @Override
            public void run() {
                for (int i = 1; i <= 1000; i++) {
                    // 模拟耗时任务（例如计算或网络请求）
                    try {
                        Thread.sleep(100);
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    }

                    // 将计数值传递给主线程以更新 UI
                    int finalI = i;
                    handler.post(new Runnable() {
                        @Override
                        public void run() {
                            TextView textView = findViewById(R.id.textView);
                            textView.setText("Count: " + finalI);
                        }
                    });
                }
            }
        }).start();
    }

    @Override
    protected void onDestroy() {
        super.onDestroy();
    }

    protected void onResume()

```

```

{
    super.onResume();
}

@Override
protected void onPause()
{
    super.onPause();
}
}

```

分析代码

```
handler = new Handler(Looper.getMainLooper());
```



`Handler` 对象，它关联到主线程（UI线程）的 `Looper`

`Looper.getMainLooper()`： `Looper` 是一个消息循环处理器，它在一个线程中管理一个消息队列。每个线程只能有一个 `Looper`。 `getMainLooper()` 是一个静态方法，用于获取主线程（UI线程）的 `Looper` 对象。

```

        handler.post(new Runnable() {
            @Override
            public void run() {
                TextView textView = findViewById(R.id.textView);
                textView.setText("Count: " + fi
            }
        });

```



下面是 `handler.post()` 方法的简要说明：

- 参数：一个实现了 `Runnable` 接口的对象。这个对象的 `run()` 方法将在与 `Handler` 关联的线程中执行。
- 返回值：一个布尔值，表示 `Runnable` 对象是否成功添加到消息队列。

如果成功添加，返回 `true`；否则返回 `false`

将代码改成 `runOnUiThread` 也是可以的，能够保证修改 UI线程 的代码，运行在，UI线程上

```
package com.fu.tt;
import androidx.appcompat.app.AppCompatActivity;

import android.os.Bundle;
import android.os.Handler;
import android.os.Looper;
import android.util.Log;
import android.view.View;
import android.widget.Button;
import android.widget.TextView;

import java.util.concurrent.ExecutorService;

public class MainActivity extends AppCompatActivity {
    private TextView textView;
    private Button button;
    private Handler handler;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        handler = new Handler(Looper.getMainLooper());

        Button button = findViewById(R.id.button);
        button.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View v) {
                startCounter();
            }
        });
    }

    private void startCounter() {
        new Thread(new Runnable() {
            @Override
```

```

        public void run() {
            for (int i = 1; i <= 1000; i++) {
                // 模拟耗时任务（例如计算或网络请求）
                try {
                    Thread.sleep(100);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }

                // 将计数值传递给主线程以更新 UI
                int finalI = i;
                runOnUiThread(new Runnable() {
                    @Override
                    public void run() {
                        TextView textView = findViewById(R.id.textView);
                        textView.setText("Count: " + finalI);
                    }
                });
            }
        }
    }).start();
}

@Override
protected void onDestroy() {
    super.onDestroy();
}

protected void onResume() {
    super.onResume();
}

@Override
protected void onPause() {
    super.onPause();
}
}

```

🔗 在其他的线程当中创建 looper

在刚刚的代码当中，我们知道

```
handler = new Handler(Looper.getMainLooper());
```



1. **Handler**： **Handler** 是 Android 系统提供的一个组件，用于在不同线程之间传递和处理消息。通常，它用于将后台线程的计算结果传递给主线程以更新 UI。 **Handler** 的主要功能是将 **Runnable** 对象或 **Message** 对象发送到关联的 **Looper** 的消息队列中。
2. **Looper.getMainLooper()**： **Looper** 是一个消息循环处理器，它在一个线程中管理一个消息队列。每个线程只能有一个 **Looper**。 **getMainLooper()** 是一个静态方法，用于获取主线程（UI线程）的 **Looper** 对象。
3. **new Handler(Looper.getMainLooper())**： 这个构造函数创建一个新的 **Handler** 实例，并将其关联到传入的 **Looper** 对象。在这个例子中，我们传入了主线程的 **Looper**，这意味着我们创建的 **Handler** 对象与主线程关联。

那么 **Looper.getMainLooper()** **getMainLooper()** 是一个静态方法，用于获取主线程（UI线程）的 **Looper** 对象，那我们如何获取，其他线程的 **looper** 呢？

以下是一个完整的例子，演示了如何在一个新的线程中创建一个 **Looper**，并通过 **Handler** 在主线程和新线程之间发送消息。在这个例子中，我们将在新线程中计算斐波那契数列的第 n 项，并将结果显示在主线程的 **TextView** 上。

```
package com.fu.tt;  
import androidx.annotation.NonNull;  
import androidx.appcompat.app.AppCompatActivity;  
  
import android.os.Bundle;
```



```
import android.os.Handler;
import android.os.Looper;
import android.os.Message;
import android.util.Log;
import android.view.View;
import android.widget.Button;
import android.widget.TextView;

import java.util.concurrent.ExecutorService;

public class MainActivity extends AppCompatActivity {
    private TextView textView;
    private Handler mainHandler;
    private Handler workerHandler;

    private static final int MESSAGE_TYPE_FIBONACCI_RESULT

    @Override
    protected void onCreate(Bundle savedInstanceState) {

        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        textView = findViewById(R.id.textView);

        mainHandler = new Handler(Looper.getMainLooper()) {
            @Override
            public void handleMessage(Message msg)
            {
                if (msg.what == MESSAGE_TYPE_FIBONACCI_RESU
                    int result = msg.arg1;
                    textView.setText("Fibonacci result: " +
                }
            }
        };

        MyLooperThread myLooperThread = new MyLooperThread(
            myLooperThread.start());

        // 等待 workerHandler 初始化完成
        while ((workerHandler = myLooperThread.getHandler())
            try {
                Thread.sleep(100);
            } catch (InterruptedException e) {
```

```

        e.printStackTrace();
    }
}

// 计算斐波那契数列的第 10 项
Message workerMessage = Message.obtain(workerHandle
workerHandler.sendMessage(workerMessage);
}

private class MyLooperThread extends Thread {
    private Handler handler;

    @Override
    public void run() {
        Looper.prepare();
        handler = new Handler(Looper.myLooper()) {
            @Override
            public void handleMessage(Message msg) {
                int n = msg.what;
                int result = fibonacci(n);

                Message mainMessage = Message.obtain(ma
                mainMessage.arg1 = result;
                mainHandler.sendMessage(mainMessage);
            }
        };
        Looper.loop();
    }

    // 用于判断，是不是已经新建成功
    public Handler getHandler() {
        return handler;
    }

    private int fibonacci(int n) {
        if (n <= 1) {
            return n;
        }
        return fibonacci(n - 1) + fibonacci(n - 2);
    }
}

@Override
protected void onDestroy() {
    super.onDestroy();
}

```

```

    }

    protected void onResume()
    {
        super.onResume();
    }

    @Override
    protected void onPause()
    {
        super.onPause();
    }
}

```

这个示例中，我们创建了一个名为 `MyLooperThread` 的子类，它从 `Thread` 继承。在该线程的 `run` 方法中，我们调用 `Looper.prepare()` 和 `Looper.loop()` 以在新线程中创建一个 `Looper`。我们还创建了一个 `Handler` 对象，它用于处理从主线程发送过来的消息。

在 `MainActivity` 的 `onCreate` 方法中，我们启动了 `MyLooperThread`，等待它的 `Handler` 初始化完成。然后，我们通过发送一个消息来请求计算斐波那契数列的第 10 项。`MyLooperThread` 的 `Handler` 接收到消息后，计算结果并将其发送回主线程，主线程的 `Handler` 更新 `TextView` 以显示结果

`Looper.prepare()` 和 `Looper.loop()` 是 `Looper` 类的方法，它们用于在新线程中创建一个消息循环。下面是对这两个方法的详细解释：

1. `Looper.prepare()`：这个方法会在当前线程中初始化一个新的 `Looper` 实例。每个线程只能有一个 `Looper` 实例，所以 `prepare()` 方法只能在一个线程中调用一次。调用 `Looper.prepare()` 会为当前线程创建一个与之关联的消息队列。
2. `Looper.loop()`：这个方法会启动消息循环。它从与当前线程关联的消息队列中获取消息，并将它们分发给相应的 `Handler`。`Looper.loop()` 会一直运行，直到消息队列为空且没有活动的 `Handler` 时才会退出。当一个线程调用了 `Looper.loop()`，它会一直处理消息，直到循环结束。这就是为什么称之为消息循环。

当您想要在一个新线程中创建一个 `Looper` 时，您需要调用 `Looper.prepare()` 和 `Looper.loop()`。首先，调用 `Looper.prepare()` 来创建一个新的 `Looper` 实例并设置与当前线程关联的消息队列。接下来，创建一个或多个 `Handler`，它们将负责处理消息。最后，调用 `Looper.loop()` 启动消息循环。

🔗 import android.os.Message

`Message` 是 Android 中用于在线程之间传递信息的一个轻量级的数据结构。`Message` 通常与 `Handler` 一起使用，用于在线程之间发送消息，实现线程间通信。

以下是 `Message` 的一些关键概念和属性：

1. `what`：一个整型字段，用于表示消息的类型。您可以在发送和处理消息时使用这个字段来区分不同类型的消息。
2. `arg1` 和 `arg2`：这是两个整型字段，您可以用它们携带一些整数数据。这对于携带简单的数据非常有用，但是对于更复杂的数据结构，您可能需要使用下面提到的 `obj` 字段。
3. `obj`：一个 `Object` 字段，您可以用它携带任何类型的对象数据。请注意，您需要确保在发送和接收消息时正确地处理这个字段中的对象类型。
4. `when`：一个长整型字段，表示消息何时应该被处理。当您通过 `Handler` 发送延时消息时，`when` 字段会自动设置为系统时间加上延迟的毫秒数。

```
package com.fu.tt;
import androidx.appcompat.app.AppCompatActivity;

import android.os.Bundle;
import android.os.Handler;
import android.os.Looper;
import android.os.Message;
import android.util.Log;
import android.view.View;
import android.widget.Button;
import android.widget.TextView;
```



```

import java.util.concurrent.ExecutorService;

public class MainActivity extends AppCompatActivity {
    private TextView textView;
    private MyLooperThread myLooperThread;
    private Handler mainHandler;
    private Handler handler;

    private static final int MESSAGE_TYPE_UPDATE_TEXT = 1;

    @Override
    protected void onCreate(Bundle savedInstanceState) {

        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        textView = findViewById(R.id.textView);

        handler = new Handler(Looper.getMainLooper()) {
            @Override
            public void handleMessage(Message msg) {
                if (msg.what == MESSAGE_TYPE_UPDATE_TEXT) {
                    String text = (String) msg.obj;
                    textView.setText(text);
                }
            }
        };

        new Thread(new Runnable() {
            @Override
            public void run() {
                String newText = "Hello from background thr
                Message message = Message.obtain(handler, M
                message.obj = newText;
                handler.sendMessage(message);
            }
        }).start();
    }

    @Override
    protected void onDestroy() {
        super.onDestroy();
    }
}

```



```

protected void onResume()
{
    super.onResume();
}

@Override
protected void onPause()
{
    super.onPause();
}
}

```

```

new Thread(new Runnable() {
    @Override
    public void run() {
        String newText = "Hello from background thr
        Message message = Message.obtain(handler, M
        message.obj = newText;
        handler.sendMessage(message);
    }
}).start();

```



1. `new Thread(new Runnable() {...}).start();`：这里创建了一个新的线程，并传入一个 `Runnable` 对象，这个对象包含了线程要执行的代码。然后调用 `start()` 方法启动这个线程。
2. `@Override public void run() {...}`：这是 `Runnable` 接口的 `run()` 方法。这个方法包含了在线程中要执行的代码。当线程启动时，它将自动调用这个方法。
3. `String newText = "Hello from background thread!";`：这是一个简单的字符串变量，我们将在后续的代码中将其作为消息发送给主线程。
4. `Message message = Message.obtain(handler, MESSAGE_TYPE_UPDATE_TEXT);`：这里我们使用 `Message.obtain()` 方法创建一个新的 `Message` 实例。我们传入一个 `Handler` 对象（在本例中是主线程的 `handler`）和一个整数 `MESSAGE_TYPE_UPDATE_TEXT`（用于表示消息类型）。这样我们可以在 `handleMessage()` 方法中根据

`what` 字段区分消息类型。

5. `message.obj = newText;`：这里我们将 `newText` 字符串赋值给 `Message` 的 `obj` 字段。这允许我们将字符串作为消息内容发送给主线程。
6. `handler.sendMessage(message);`：这里我们使用 `handler` 的 `sendMessage()` 方法将 `Message` 实例发送给主线程。当主线程收到这个消息时，它将在 `handleMessage()` 方法中处理消息，并根据 `what` 字段更新 UI。

```
handler = new Handler(Looper.getMainLooper()) {  
    @Override  
    public void handleMessage(Message msg) {  
        if (msg.what == MESSAGE_TYPE_UPDATE_TEXT) {  
            String text = (String) msg.obj;  
            textView.setText(text);  
        }  
    }  
};
```

1. `handler = new Handler(Looper.getMainLooper());`：这行代码创建了一个 `Handler` 对象，并将其与主线程的 `Looper` 对象关联起来。通过将 `Looper.getMainLooper()` 作为参数传递给 `Handler` 构造函数，我们告诉这个 `Handler` 处理来自主线程 `Looper` 的消息队列。因此，这个 `Handler` 可以在主线程中处理从其他线程发送过来的消息。
2. `public void handleMessage(Message msg)`：这个方法是 `Handler` 类的一个重要方法，需要我们重写以处理消息。这个方法会在主线程中被调用，以处理从其他线程发送过来的消息。
3. `if (msg.what == MESSAGE_TYPE_UPDATE_TEXT)`：`msg.what` 是一个整数值，用于区分不同类型的消息。在这个例子中，我们检查 `msg.what` 是否等于预先定义的 `MESSAGE_TYPE_UPDATE_TEXT` 常量，以确定是否需要处理这条消息。
4. `String text = (String) msg.obj`：如果消息的类型是 `MESSAGE_TYPE_UPDATE_TEXT`，我们将 `msg.obj` 转换为 `String` 类型。在这个例子中，`msg.obj` 包含了我们要在 `TextView` 中显示的文本。
5. `textView.setText(text)`：最后，我们使用 `setText()` 方法

将 `TextView` 的文本设置为从 `msg.obj` 获取的字符串。这样，主线程就可以根据从其他线程发送过来的消息更新UI。

🔗 import android.os.Message_arg1和arg2

`Message` 对象中的 `arg1` 和 `arg2` 是两个整型字段（如果是其他的数据类型，就用 `message.obj`），它们可以用于携带整数类型的数据。这对于在消息中传递简单的整数数据非常有用。它们可以作为额外的信息与 `msg.what` 字段一起使用，以帮助接收方根据这些参数执行相应的操作。

```
package com.fu.tt;
import androidx.appcompat.app.AppCompatActivity;

import android.os.Bundle;
import android.os.Handler;
import android.os.Looper;
import android.os.Message;
import android.util.Log;
import android.view.View;
import android.widget.Button;
import android.widget.TextView;

import java.util.concurrent.ExecutorService;

public class MainActivity extends AppCompatActivity {
    private TextView textView;
    private Handler handler;

    public static final int OPERATION_ADD = 1;
    public static final int OPERATION_SUBTRACT = 2;

    @Override
    protected void onCreate(Bundle savedInstanceState) {

        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        textView = findViewById(R.id.textview);

        handler = new Handler(Looper.getMainLooper()) {
```

```

@Override
public void handleMessage(Message msg) {
    int result = -1;
    if (msg.what == OPERATION_ADD) {
        result = msg.arg1 + msg.arg2;
    } else if (msg.what == OPERATION_SUBTRACT)
        result = msg.arg1 - msg.arg2;
    }
    // 更新UI, 例如显示结果
    textView.setText(String.valueOf(result));
}

Message message = Message.obtain();
message.what = OPERATION_SUBTRACT;
message.arg1 = 3;
message.arg2 = 5;
handler.sendMessage(message);
}

@Override
protected void onDestroy() {
    super.onDestroy();
}

protected void onResume()
{
    super.onResume();
}

@Override
protected void onPause()
{
    super.onPause();
}
}

```

我将 Message 对象的创建和发送移到了 onCreate() 方法的末尾。这样在创建 Handler 对象后，才会发送 Message 对象。否则，Handler 对象可能尚未创建，而 Message 对象已经发送，这将导致无法处理消息

在这次的创建 message 的时候，我们使用的是

```
Message message = Message.obtain();
```



而不是使用的

```
Message message = Message.obtain(handler, MESSAGE_TYPE_UPDA
```



这是因为

```
Message message = Message.obtain();  
message.what = OPERATION_SUBTRACT;  
message.arg1 = 3;  
message.arg2 = 5;  
handler.sendMessage(message);
```



在后续的操作当中已经单独指定了 `message.what = OPERATION_SUBTRACT;`

至于 handler，在发送广播的时候 `handler.sendMessage(message);` 就已经指定了，在创建的时候指定，其实是有点画蛇添足的

🔗 import android.os.Message_obj

`Message.obj` 是一个用于携带任意类型对象数据的字段。在使用 `Message` 传递数据时，您可以将需要传递的对象赋值给 `obj` 字段。然后，在接收消息时，您需要根据发送的对象类型进行相应的处理。

Person.java

```
public class Person {  
    public String name;  
    public int age;  
  
    public Person(String name, int age) {  
        this.name = name;  
        this.age = age;  
    }  
}
```



MainActivity.java



```
public class MainActivity extends AppCompatActivity {
    private TextView textView;
    private Handler handler;

    public static final int MESSAGE_TYPE_PERSON = 1;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        textView = findViewById(R.id.textView);

        handler = new Handler(Looper.getMainLooper()) {
            @Override
            public void handleMessage(Message msg) {
                if (msg.what == MESSAGE_TYPE_PERSON) {
                    Person person = (Person) msg.obj;
                    String text = "Name: " + person.name +
                        textView.setText(text);
                }
            }
        };

        new Thread(new Runnable() {
            @Override
            public void run() {
                Person person = new Person("Alice", 30);
                Message message = Message.obtain();
                message.what = MESSAGE_TYPE_PERSON;
                message.obj = person;
                handler.sendMessage(message);
            }
        }).start();
    }
}
```

在这个例子中，我们创建了一个 `Person` 对象并将其发送给主线程的 `Handler`。`Handler` 根据 `message.what` 判断消息类型，并将 `message.obj` 转换为 `Person` 类型，然后在 UI 上显示其属性。

需要注意的是，在使用 `Message.obj` 传递对象时，要确保发送和接收时正确处理对象类型，避免类型转换异常。

🔗 import android.os.Message_在message当中存放多个数据

可以发现，无论是， arg1_arg2 还是 obj 都只能存放一个数据，想要发送多个信息，可以考虑 Bundle 的数据（或者是在 msg.obj 当中传入一个 Bundle）

```
package com.fu.tt;
import androidx.annotation.NonNull;
import androidx.appcompat.app.AppCompatActivity;

import android.os.Bundle;
import android.os.Handler;
import android.os.Looper;
import android.os.Message;
import android.util.Log;
import android.view.View;
import android.widget.Button;
import android.widget.TextView;

import java.util.concurrent.ExecutorService;

public class MainActivity extends AppCompatActivity {
    private TextView textView;
    private Handler handler;

    public static final int MESSAGE_TYPE_UPDATE_TEXT = 1;

    @Override
    protected void onCreate(Bundle savedInstanceState) {

        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        textView = findViewById(R.id.textview);

        handler = new Handler(Looper.getMainLooper()) {
            @Override
            public void handleMessage(@NonNull Message msg)
                if (msg.what == MESSAGE_TYPE_UPDATE_TEXT) {
                    Bundle data = msg.getData();
                    String newText = data.getString("newTex
```



```

        int someInt = data.getInt("someIntKey")
        double someDouble = data.getDouble("som

        // 使用获取到的数据执行操作
        textView.setText(newText);
        // ...
    }
}

};

new Thread(new Runnable() {
    @Override
    public void run() {
        String newText = "Hello from background thr
        int someInt = 42;
        double someDouble = 3.14;

        Message message = Message.obtain(handler, M

        // 创建一个 Bundle 对象，并将多个数据放入其中
        Bundle data = new Bundle();
        data.putString("newTextKey", newText);
        data.putInt("someIntKey", someInt);
        data.putDouble("someDoubleKey", someDouble)

        // 将 Bundle 对象与 Message 关联
        message.setData(data);

        handler.sendMessage(message);
    }
}).start();

}

@Override
protected void onDestroy() {
    super.onDestroy();
}

protected void onResume()
{
    super.onResume();
}

```

```
@Override
protected void onPause()
{
    super.onPause();
}
}
```

```
package com.fu.tt;
import androidx.annotation.NonNull;
import androidx.appcompat.app.AppCompatActivity;

import android.os.Bundle;
import android.os.Handler;
import android.os.Looper;
import android.os.Message;
import android.util.Log;
import android.view.View;
import android.widget.Button;
import android.widget.TextView;

import java.util.concurrent.ExecutorService;

public class MainActivity extends AppCompatActivity {
    private TextView textView;
    private Handler handler;

    public static final int MESSAGE_TYPE_UPDATE_TEXT = 1;

    @Override
    protected void onCreate(Bundle savedInstanceState) {

        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        textView = findViewById(R.id.textview);

        handler = new Handler(Looper.getMainLooper()) {
            @Override
            public void handleMessage(@NonNull Message msg)
                if (msg.what == MESSAGE_TYPE_UPDATE_TEXT) {
```



```

        Bundle data = (Bundle) msg.obj;
        String newText = data.getString("newText");
        int someInt = data.getInt("someIntKey");
        double someDouble = data.getDouble("someDoubleKey");

        // 使用获取到的数据执行操作
        textView.setText(String.valueOf(someInt));
        // ...
    }
}

};

new Thread(new Runnable() {
    @Override
    public void run() {
        String newText = "Hello from background thread";
        int someInt = 42;
        double someDouble = 3.14;

        Message message = Message.obtain(handler, "newText", newText, someInt, someDouble);

        // 创建一个 Bundle 对象，并将多个数据放入其中
        Bundle data = new Bundle();
        data.putString("newTextKey", newText);
        data.putInt("someIntKey", someInt);
        data.putDouble("someDoubleKey", someDouble);

        // 将 Bundle 对象与 Message 关联
        message.obj = data;

        handler.sendMessage(message);
    }
}).start();
}

@Override
protected void onDestroy() {
    super.onDestroy();
}

protected void onResume() {
    super.onResume();
}

```

```
    }

    @Override
    protected void onPause()
    {
        super.onPause();
    }
}
```

🔗 import android.os.Message_msg.when

`Message.when` 是一个长整型字段，用于表示消息应该在何时被处理。在使用 `Handler` 发送延时消息时，`when` 字段会自动设置为当前系统时间加上延迟的毫秒数。`Handler` 内部会根据这个值进行消息调度和处理。

当您需要一定时间后处理某个消息时，您可以使用 `Handler` 的 `sendMessageDelayed()` 方法，该方法需要指定一个延迟时间。例如，假设我们想要在 5 秒后更新 `TextView` 的文本：

```
handler.sendMessageDelayed(message, 5000);
```



```
public class MainActivity extends AppCompatActivity {
    private TextView textView;
    private Handler handler;

    public static final int MESSAGE_TYPE_UPDATE_TEXT = 1;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        textView = findViewById(R.id.textView);

        handler = new Handler(Looper.getMainLooper()) {
            @Override
            public void handleMessage(Message msg) {
                if (msg.what == MESSAGE_TYPE_UPDATE_TEXT) {
                    textView.setText("Text updated after 5
                }
            }
        };

        // 创建一个 Message 对象并设置其类型
        Message message = Message.obtain();
        message.what = MESSAGE_TYPE_UPDATE_TEXT;

        // 使用 sendMessageDelayed() 方法发送延时消息
        handler.sendMessageDelayed(message, 5000); // 延迟 5
    }
}
```

或者是

sendMessageAtTime



```
public class MainActivity extends AppCompatActivity {  
    private TextView textView;  
    private Handler handler;  
  
    public static final int MESSAGE_TYPE_UPDATE_TEXT = 1;  
  
    @Override  
    protected void onCreate(Bundle savedInstanceState) {  
        super.onCreate(savedInstanceState);  
        setContentView(R.layout.activity_main);  
  
        textView = findViewById(R.id.textView);  
  
        handler = new Handler(Looper.getMainLooper()) {  
            @Override  
            public void handleMessage(Message msg) {  
                if (msg.what == MESSAGE_TYPE_UPDATE_TEXT) {  
                    textView.setText("Text updated after 5  
                }  
            }  
        };  
  
        // 创建一个 Message 对象并设置其类型  
        Message message = Message.obtain();  
        message.what = MESSAGE_TYPE_UPDATE_TEXT;  
  
        // 设置 Message.when 字段为当前系统时间加上延迟的毫秒数  
        long currentTimeMillis = SystemClock.uptimeMillis();  
        long delayMillis = 5000;  
        message.when = currentTimeMillis + delayMillis;  
  
        // 使用 sendMessageAtTime() 方法发送延迟消息  
        handler.sendMessageAtTime(message, message.when);  
    }  
}
```

🔗 Handle的线程之间的通信

在之前，我们已经讲解了如何做到，线程之间的通信，现在我们再看看，如何做到，线程之间的通信



```
package com.fu.tt;
import androidx.annotation.NonNull;
import androidx.appcompat.app.AppCompatActivity;
import androidx.core.app.ActivityCompat;
import androidx.core.content.ContextCompat;

import android.content.pm.PackageManager;
import android.os.Bundle;
import android.os.Handler;
import android.os.Looper;
import android.os.Message;
import android.util.Log;
import android.view.View;
import android.widget.Button;
import android.widget.TextView;
import android.Manifest;

import java.util.concurrent.ExecutorService;

public class MainActivity extends AppCompatActivity {

    private static final int UPDATE_UI = 1;
    private Button button;
    private TextView textView;

    private Handler handler = new Handler(Looper.getMainLoo
        @Override
        public void handleMessage(Message msg) {
            if (msg.what == UPDATE_UI) {
                textView.setText("耗时操作已完成");
            }
        }
    };

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        button = findViewById(R.id.button);
        textView = findViewById(R.id.textView);

        button.setOnClickListener(new View.OnClickListener(
            @Override
```

```

        public void onClick(View v) {
            performTask();
        }
    });
}

// 在另一个线程，执行的耗时任务
private void performTask() {
    new Thread(new Runnable() {
        @Override
        public void run() {
            try {
                Thread.sleep(2000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }

            Message message = handler.obtainMessage(UPD
                handler.sendMessage(message);
        }
    }).start();
}

@Override
protected void onDestroy() {
    super.onDestroy();
}

protected void onResume()
{
    super.onResume();
}

@Override
protected void onPause()
{
    super.onPause();
}
}

```

🔗 Handler loop 还有 Runnable 这三个东西的关系

🔗 Handler

作用与功能：

1. 通信： `Handler` 用于在不同线程之间发送和处理消息。它可以将消息从一个线程发送到另一个线程，例如从后台线程发送消息到主线程（UI线程）。
2. 任务调度： `Handler` 还可以用于任务调度，即在指定的时间或时间间隔后执行特定任务。它提供了一系列方法（如 `post()`、`postDelayed()` 和 `postAtTime()`）来安排任务。

🔗 Looper

作用与功能：

1. 消息循环： `Looper` 是一个运行在特定线程的消息循环，负责从与其关联的消息队列中取出并处理消息。每个线程只能有一个关联的 `Looper`，默认情况下，仅主线程（UI线程）有一个 `Looper`。
2. 管理消息队列： `Looper` 管理一个消息队列，它将接收到的消息和 `Runnable` 对象按照优先级和预定执行时间排序。当执行完一个任务后，`Looper` 将继续从队列中取出并执行下一个任务。

🔗 Runnable

作用与功能：

1. 封装任务： `Runnable` 是一个接口，用于封装要在特定线程上执行的任务。您可以通过实现 `Runnable` 接口的 `run()` 方法来定义任务内容。
2. 灵活调度： `Runnable` 对象可以直接传递给 `Handler` 的 `post()` 方法或作为消息的一部分发送。这为在不同线程之间调度任务提供了灵活性。

简单来说，`Runnable` 的作用是，封装任务，`Looper` 的作用是，管理消息和任务，`Handler` 的作用是发送消息和任务

但是处理消息是在

```
private Handler handler = new Handler(Looper.getMainLoo   
    @Override  
    public void handleMessage(Message msg) {  
        if (msg.what == UPDATE_UI) {  
            textView.setText("耗时操作已完成");  
        }  
    }  
};
```

当中实现的

🔗 Handle的任务调度

`Handler` 除了用于线程间通信外，还可以用于任务调度。任务调度是指在某个特定的时间或时间间隔后执行特定的任务。`Handler` 提供了 `post()`, `postDelayed()`, `postAtTime()` 等方法来实现任务调度。

1. `post(Runnable r)`: 将一个 `Runnable` 对象加入到消息队列中，该队列会被关联到 `Handler` 的 `Looper`。`Runnable` 会在下一个空闲时间执行。

```
handler.post(new Runnable() {   
    @Override  
    public void run() {  
        // 在主线程执行的代码  
    }  
});
```

2. `postDelayed(Runnable r, long delayMillis)`: 将一个 `Runnable` 对象加入到消息队列中，该队列会被关联到 `Handler` 的 `Looper`。`Runnable` 会在指定的延迟时间（以毫秒为单位）后执行。

```
handler.postDelayed(new Runnable() {  
    @Override  
    public void run() {  
        // 在主线程执行的代码  
    }  
}, 3000); // 延迟 3 秒执行
```



3. `postAtTime(Runnable r, long uptimeMillis)`: 将一个 `Runnable` 对象加入到消息队列中，该队列会被关联到 `Handler` 的 `Looper`。 `Runnable` 会在指定的系统运行时间（以毫秒为单位）后执行。

```
long targetTime = SystemClock.uptimeMillis() + 5000; // 当前  
handler.postAtTime(new Runnable() {  
    @Override  
    public void run() {  
        // 在主线程执行的代码  
    }  
}, targetTime);
```



`post(Runnable r)`

`post(Runnable r)` 是 `Handler` 类中的一个方法，它用于将一个 `Runnable` 对象添加到与 `Handler` 关联的 `Looper` 的消息队列中。 `Runnable` 对象中的任务将在 `Looper` 所关联的线程中执行。

这里是 `post(Runnable r)` 方法的详细解释：

功能： 将一个 `Runnable` 对象添加到消息队列中，以便在 `Looper` 关联的线程上执行。

参数： `r` - 要执行的 `Runnable` 对象。

返回值： 如果成功将 `Runnable` 对象添加到消息队列，则返回 `true`；否则返回 `false`。



```
package com.fu.tt;
import androidx.annotation.NonNull;
import androidx.appcompat.app.AppCompatActivity;
import androidx.core.app.ActivityCompat;
import androidx.core.content.ContextCompat;

import android.content.pm.PackageManager;
import android.os.Bundle;
import android.os.Handler;
import android.os.Looper;
import android.os.Message;
import android.util.Log;
import android.view.View;
import android.widget.Button;
import android.widget.TextView;
import android.Manifest;

import java.util.concurrent.ExecutorService;

public class MainActivity extends AppCompatActivity {

    private static final int UPDATE_UI = 1;
    private Button button;
    private TextView textView;

    private Handler mainThreadHandler = new Handler(Looper.

@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);

    button = findViewById(R.id.button);
    textView = findViewById(R.id.textView);

    mainThreadHandler = new Handler(Looper.getMainLoope

    button.setOnClickListener(new View.OnClickListener()
        @Override
        public void onClick(View v) {
            updateUI();
        }
    });
}
```

```

private void updateUI() {
    mainThreadHandler.post(new Runnable() {
        @Override
        public void run() {
            textView.setText("UI has been updated!");
        }
    });
}

@Override
protected void onDestroy() {
    super.onDestroy();
}

protected void onResume()
{
    super.onResume();
}

@Override
protected void onPause()
{
    super.onPause();
}
}

```

🔗 postDelayed(Runnable r, long delayMillis)

参数：

- `r` - 要执行的 `Runnable` 对象。
- `delayMillis` - 延迟执行 `Runnable` 对象的时间，以毫秒为单位。

返回值： 如果成功将 `Runnable` 对象添加到消息队列，则返回 `true`；否则返回 `false`。

```

package com.fu.tt;
import androidx.annotation.NonNull;

```



```

import androidx.appcompat.app.AppCompatActivity;
import androidx.core.app.ActivityCompat;
import androidx.core.content.ContextCompat;

import android.content.pm.PackageManager;
import android.os.Bundle;
import android.os.Handler;
import android.os.Looper;
import android.os.Message;
import android.util.Log;
import android.view.View;
import android.widget.Button;
import android.widget.TextView;
import android.Manifest;

import java.util.concurrent.ExecutorService;

public class MainActivity extends AppCompatActivity {

    private static final int UPDATE_UI = 1;
    private Button button;
    private TextView textView;

    private Handler mainThreadHandler = new Handler(Looper.

@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);

    button = findViewById(R.id.button);
    textView = findViewById(R.id.textView);

    mainThreadHandler = new Handler(Looper.getMainLoope

    button.setOnClickListener(new View.OnClickListener(
        @Override
        public void onClick(View v) {
            updateUI();
        }
    ));
}

private void updateUI() {

```

```

        mainThreadHandler.postDelayed(new Runnable() {
            @Override
            public void run() {
                textView.setText("UI has been updated!");
            }
        }, 3000); //3秒钟
    }

    @Override
    protected void onDestroy() {
        super.onDestroy();
    }

    protected void onResume()
    {
        super.onResume();
    }

    @Override
    protected void onPause()
    {
        super.onPause();
    }
}

```

🔗 postAtTime(Runnable r, long uptimeMillis)

`postAtTime(Runnable r, long uptimeMillis)` 是 `Handler` 类中的一个方法，它用于将一个 `Runnable` 对象添加到与 `Handler` 关联的 `Looper` 的消息队列中，并在指定的系统运行时间（`uptimeMillis`）时执行。与 `post(Runnable r)` 和 `postDelayed(Runnable r, long delayMillis)` 类似，`Runnable` 对象中的任务将在 `Looper` 所关联的线程中执行。

功能： 将一个 `Runnable` 对象添加到消息队列中，在指定的系统运行时间（`uptimeMillis`）时在 `Looper` 关联的线程上执行。

参数：

- `r` - 要执行的 `Runnable` 对象。

- `uptimeMillis` - 指定的系统运行时间，以毫秒为单位。这个值是从系统启动开始计算的，不包括深度睡眠时间。

返回值： 如果成功将 `Runnable` 对象添加到消息队列，则返回 `true`；否则返回 `false`。

```
package com.fu.tt;

import androidx.annotation.NonNull;
import androidx.appcompat.app.AppCompatActivity;
import androidx.core.app.ActivityCompat;
import androidx.core.content.ContextCompat;

import android.content.pm.PackageManager;
import android.os.Bundle;
import android.os.Handler;
import android.os.Looper;
import android.os.Message;
import android.os.SystemClock;
import android.util.Log;
import android.view.View;
import android.widget.Button;
import android.widget.TextView;
import android.Manifest;

import java.util.concurrent.ExecutorService;

public class MainActivity extends AppCompatActivity {

    private static final int UPDATE_UI = 1;
    private Button button;
    private TextView textView;

    private Handler mainThreadHandler = new Handler(Looper.

@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);

    button = findViewById(R.id.button);
    textView = findViewById(R.id.textView);
```



```

        mainThreadHandler = new Handler(Looper.getMainLooper());

        button.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View v) {
                updateUI();
            }
        });
    }

    private void updateUI() {
        long currentTime = SystemClock.uptimeMillis();
        long targetTime = currentTime + 3000; // 在当前时间的
        mainThreadHandler.postAtTime(new Runnable() {
            @Override
            public void run() {
                // 在主线程（UI线程）中执行的代码，例如更新 UI
                textView.setText("定时更新的内容");
            }
        }, targetTime);
    }

    @Override
    protected void onDestroy() {
        super.onDestroy();
    }

    protected void onResume() {
        {
            super.onResume();
        }
    }

    @Override
    protected void onPause() {
        {
            super.onPause();
        }
    }
}

```

🔗 SystemClock 的几种方法

`SystemClock.uptimeMillis()` 是 Android 中 `SystemClock` 类的一个方法，它返回自系统启动以来的时间（不包括设备处于深度睡眠状态的时间），单位为毫秒。这个方法在计算延迟、设定定时任务或者测量时间间隔等场景下非常有用。

`SystemClock.uptimeMillis()` 的返回值表示设备从启动到现在经过的时间，但不包括设备进入深度睡眠（如 CPU 休眠）的时间。这意味着在设备处于活跃状态时，这个值会持续增加，但在设备进入睡眠状态时，这个值会暂停。

`SystemClock` 类中还有其他几个与时间相关的方法，例如：

- `SystemClock.elapsedRealtime()`：返回自系统启动以来的时间，包括设备处于深度睡眠状态的时间，单位为毫秒。
- `SystemClock.currentThreadTimeMillis()`：返回当前线程已运行的时间，单位为毫秒。
- `SystemClock.sleep(long ms)`：使当前线程睡眠指定的毫秒数。

🔗 `Handler.Callback`

`mainThreadHandler = new Handler(Looper.getMainLooper())` 的方式并没有完全被淘汰，但是在某些场景下，使用 `Handler.Callback` 接口来处理消息更加推荐，因为这样可以更好地管理和组织代码。

`Handler.Callback` 接口允许您更加灵活地处理 `Handler` 收到的消息。当您在创建 `Handler` 对象时提供一个实现了 `Handler.Callback` 接口的对象，`Handler` 会将接收到的消息传递给该 `Callback` 对象的 `handleMessage(Message msg)` 方法处理。这样，您可以将消息处理逻辑与 `Handler` 对象的创建分离，使代码更加模块化和易于维护。也就是说，可以在类里面重写消息/任务处理函数 `handleMessage`，而不是，在创建的时候，一起写 `handleMessage`

不使用 `Handler.Callback`

```
package com.fu.tt;

import androidx.appcompat.app.AppCompatActivity;

import android.os.Bundle;
import android.os.Handler;
import android.os.Looper;
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 {

    private static final int MESSAGE_TYPE_1 = 1;
    private static final int MESSAGE_TYPE_2 = 2;

    private Button buttonType1;
    private Button buttonType2;

    private Handler mainThreadHandler;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        buttonType1 = findViewById(R.id.buttonType1);
        buttonType2 = findViewById(R.id.buttonType2);

        mainThreadHandler = new Handler(Looper.getMainLooper())
        @Override
        public void handleMessage(Message msg) {
            switch (msg.what) {
                case MESSAGE_TYPE_1:
                    Toast.makeText(MainActivity.this, "
                    break;
                case MESSAGE_TYPE_2:
                    Toast.makeText(MainActivity.this, "
                    break;
                default:
                    super.handleMessage(msg);
            }
        }
    }
}
```

```

};

buttonType1.setOnClickListener(new View.OnClickListener()
    @Override
    public void onClick(View v) {
        mainThreadHandler.sendMessage(MESSAGE_
    }
});

buttonType2.setOnClickListener(new View.OnClickListener()
    @Override
    public void onClick(View v) {
        mainThreadHandler.sendMessage(MESSAGE_
    }
});
}
}

```

加了Handler.Callback

```

package com.fu.tt;

import androidx.appcompat.app.AppCompatActivity;

import android.os.Bundle;
import android.os.Handler;
import android.os.Looper;
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 implements

    private static final int MESSAGE_TYPE_1 = 1;
    private static final int MESSAGE_TYPE_2 = 2;

    private Button buttonType1;
    private Button buttonType2;

    private Handler mainThreadHandler;

```

```

@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);

    buttonType1 = findViewById(R.id.buttonType1);
    buttonType2 = findViewById(R.id.buttonType2);

    mainThreadHandler = new Handler(Looper.getMainLooper());

    buttonType1.setOnClickListener(new View.OnClickListener() {
        @Override
        public void onClick(View v) {
            mainThreadHandler.sendMessage(MESSAGE_1);
        }
    });

    buttonType2.setOnClickListener(new View.OnClickListener() {
        @Override
        public void onClick(View v) {
            mainThreadHandler.sendMessage(MESSAGE_2);
        }
    });
}

@Override
public boolean handleMessage(Message msg) {
    // 根据 msg.what 的值处理不同的消息
    switch (msg.what) {
        case MESSAGE_TYPE_1:
            // 处理类型为 MESSAGE_TYPE_1 的消息
            Toast.makeText(this, "Message type 1 received", Toast.LENGTH_SHORT).show();
            break;
        case MESSAGE_TYPE_2:
            // 处理类型为 MESSAGE_TYPE_2 的消息
            Toast.makeText(this, "Message type 2 received", Toast.LENGTH_SHORT).show();
            break;
        default:
            // 对于未知的消息类型, 返回 false, 表示消息未被处理
            return false;
    }
    // 返回 true, 表示消息已被处理
    return true;
}

```

```
}
```

通过Lambda表达式

```
package com.fu.tt;

import androidx.appcompat.app.AppCompatActivity;

import android.os.Bundle;
import android.os.Handler;
import android.os.Looper;
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 {

    private static final int MESSAGE_TYPE_1 = 1;
    private static final int MESSAGE_TYPE_2 = 2;

    private Button buttonType1;
    private Button buttonType2;

    private Handler mainThreadHandler;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        buttonType1 = findViewById(R.id.buttonType1);
        buttonType2 = findViewById(R.id.buttonType2);

        mainThreadHandler = new Handler(Looper.getMainLooper()
            switch (msg.what) {
                case MESSAGE_TYPE_1:
                    Toast.makeText(MainActivity.this, "Mess
                    break;
                case MESSAGE_TYPE_2:
                    Toast.makeText(MainActivity.this, "Mess
```

```

                break;
            default:
                return false;
        }
        return true;
    });

    buttonType1.setOnClickListener(new View.OnClickListener()
    @Override
    public void onClick(View v) {
        mainThreadHandler.sendMessage(MESSAGE_
    }
    });

    buttonType2.setOnClickListener(new View.OnClickListener()
    @Override
    public void onClick(View v) {
        mainThreadHandler.sendMessage(MESSAGE_
    }
    });
}
}

```

将lambda表达式展开

```

package com.fu.tt;

import androidx.appcompat.app.AppCompatActivity;

import android.os.Bundle;
import android.os.Handler;
import android.os.Looper;
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 {

    private static final int MESSAGE_TYPE_1 = 1;
    private static final int MESSAGE_TYPE_2 = 2;

```

```

private Button buttonType1;
private Button buttonType2;

private Handler mainThreadHandler;

@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);

    buttonType1 = findViewById(R.id.buttonType1);
    buttonType2 = findViewById(R.id.buttonType2);

    mainThreadHandler = new Handler(Looper.getMainLooper());
    @Override
    public boolean handleMessage(Message msg) {
        switch (msg.what) {
            case MESSAGE_TYPE_1:
                Toast.makeText(MainActivity.this, "
                break;
            case MESSAGE_TYPE_2:
                Toast.makeText(MainActivity.this, "
                break;
            default:
                return false;
        }
        return true;
    }
});

buttonType1.setOnClickListener(new View.OnClickListener()
@Override
public void onClick(View v) {
    mainThreadHandler.sendMessage(MESSAGE_
}
});

buttonType2.setOnClickListener(new View.OnClickListener()
@Override
public void onClick(View v) {
    mainThreadHandler.sendMessage(MESSAGE_
}

```



```
        });  
    }  
}
```

加了Handler.Callback

```
package com.fu.tt;  
  
import androidx.appcompat.app.AppCompatActivity;  
  
import android.os.Bundle;  
import android.os.Handler;  
import android.os.Looper;  
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 {  
  
    private static final int MESSAGE_TYPE_1 = 1;  
    private static final int MESSAGE_TYPE_2 = 2;  
  
    private Button buttonType1;  
    private Button buttonType2;  
  
    private Handler mainThreadHandler;  
  
    @Override  
    protected void onCreate(Bundle savedInstanceState) {  
        super.onCreate(savedInstanceState);  
        setContentView(R.layout.activity_main);  
  
        buttonType1 = findViewById(R.id.buttonType1);  
        buttonType2 = findViewById(R.id.buttonType2);  
  
        Handler.Callback callback = new Handler.Callback()  
        {  
            @Override  
            public boolean handleMessage(Message msg) {  
                switch (msg.what) {  
                    case MESSAGE_TYPE_1:  

```

```

        Toast.makeText(MainActivity.this, "
            break;
        case MESSAGE_TYPE_2:
            Toast.makeText(MainActivity.this, "
                break;
            default:
                return false;
        }
        return true;
    }
};

mainThreadHandler = new Handler(Looper.getMainLoope

buttonType1.setOnClickListener(new View.OnClickListener
    @Override
    public void onClick(View v) {
        mainThreadHandler.sendMessage(MESSAGE_
    }
});

buttonType2.setOnClickListener(new View.OnClickListener
    @Override
    public void onClick(View v) {
        mainThreadHandler.sendMessage(MESSAGE_
    }
});
}
}

```

🔗 Lambda表达式

ambda 表达式是 Java 8 中引入的一种新特性，它提供了一种简洁、函数式的方法来表示只有一个方法的接口（也称为**函数式接口**）的实例。Lambda 表达式可以使代码更简洁、易读。

```
(parameters) -> { body }
```



1. 参数列表 (parameters): 它定义了 Lambda 表达式所需要的输入参数。参数列表可以为空，有一个参数或多个参数。如果参数列表为空，您需要

使用一对空括号 `()`。如果只有一个参数，可以选择省略括号。如果有多个参数，需要用逗号分隔，用括号括起来。

2. 箭头符号 (`->`)：它用于分隔参数列表和 Lambda 表达式的主体部分。
3. 主体 (body)：它包含了 Lambda 表达式需要执行的代码。主体可以是一个表达式或一个代码块。如果主体是一个表达式，那么结果将被隐式返回。如果主体是一个代码块，您需要使用大括号 `{}` 将其括起来，并且可能需要显式使用 `return` 语句返回结果。

如果一个接口里只有一个抽象方法（不包括默认方法和静态方法），那么这个接口就被称为函数式接口（Functional Interface）。只有函数式接口才能使用 Lambda 表达式。

```
package com.fu.tt;

interface MyFunctionalInterface {
    void doSomething(String input);
}
```





```
package com.fu.tt;

import androidx.appcompat.app.AppCompatActivity;

import android.os.Bundle;
import android.os.Handler;
import android.os.Looper;
import android.os.Message;
import android.util.Log;
import android.view.View;
import android.widget.Button;
import android.widget.TextView;
import android.widget.Toast;

public class MainActivity extends AppCompatActivity {

    private static final int MESSAGE_TYPE_1 = 1;
    private static final int MESSAGE_TYPE_2 = 2;

    private Button buttonType1;
    private Button buttonType2;

    private Handler mainThreadHandler;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        MyFunctionalInterface myInstance1 = input -> Log.i(

        MyFunctionalInterface myInstance2 = new MyFunctiona
            @Override
            public void doSomething(String input) {
                Log.i("tag", "Hello, " + input);
            }
        };

        myInstance1.doSomething("heisenberg");
        myInstance2.doSomething("heisenberg");
    }
}
```

一个案例

Greeting.java

```
package com.fu.tt;

@FunctionalInterface
public interface Greeting {
    void sayHello(String name);
}
```

`@FunctionalInterface` 是一个注解，用于表示一个接口是一个函数式接口。函数式接口是一个具有单一抽象方法（SAM）的接口。由于它们只包含一个方法，它们可以用于 Lambda 表达式和方法引用。

请注意，`@FunctionalInterface` 注解不是强制性的。如果一个接口只包含一个抽象方法，它在语义上仍然是一个函数式接口，即使没有使用 `@FunctionalInterface` 注解。然而，建议使用这个注解来明确表示接口的用途并提高代码的可读性。

LambdaDemo.java

```
public class LambdaDemo {
    public static void main(String[] args) {
        // 使用带有大括号的 Lambda 表达式实现 Greeting 接口
        Greeting greeting = (name) -> {
            String greetingMessage = "Hello, " + name + "!";
            System.out.println(greetingMessage);
            return greetingMessage;
        };

        // 调用 sayHello 方法
        String result = greeting.sayHello("Alice");
        System.out.println("Greeting result: " + result);
    }
}
```

或者是，当只有一个的函数的时候，就可以不用 {}

```
package com.fu.tt;

import android.util.Log;

public class LambdaDemo {
    public static void main(String[] args) {
        // 使用 Lambda 表达式实现 Greeting 接口
        Greeting greeting = (name) -> Log.i("tag", "Hello "

        // 调用 sayHello 方法
        greeting.sayHello("Alice");
    }
}
```

🔗 接口当中的默认方法

接口可以包含默认方法（Default Method）。默认方法是一种特殊类型的方法，它在接口中提供了具体的实现，而不是一个抽象方法。默认方法允许接口的设计者在不破坏已有实现的情况下，向接口添加新的方法

默认方法使用 `default` 关键字修饰，并提供方法体。这样，实现接口的类可以选择覆盖默认方法，也可以直接使用默认实现。以下是一个带有默认方法的接口示例：

```
public interface MyInterface {
    void abstractMethod(String input);

    default void defaultMethod() {
        System.out.println("This is a default method in the
    }
}
```

🔗 View.post() 和 View.postDelayed()

`View.post()` 和 `View.postDelayed()` 是 Android View 类的方法，它们允许你将一个 Runnable 对象排队到 UI 线程的消息队列中。这对于更新 UI 或执行与 UI 相关的操作非常有用，因为这些操作必须在 UI 线程上执行。这两个方法都可以确保 Runnable 对象在 UI 线程上运行。

1. `View.post(Runnable action)`：这个方法将 Runnable 对象添加到 View 的消息队列中。当 View 的 UI 线程准备好执行 Runnable 时，它会执行。这是在不使用 Handler 的情况下，在 UI 线程上执行代码的简便方法

View.post()

```
textView.post(new Runnable() {  
    @Override  
    public void run() {  
        textView.setText("Text updated on the UI thread");  
    }  
});
```



View.postDelayed()

```
textView.postDelayed(new Runnable() {  
    @Override  
    public void run() {  
        textView.setText("Text updated after 2 seconds");  
    }  
}, 2000); // 更新文本将在 2 秒后发生
```



完整的demo

activity_main.xml

```
<androidx.constraintlayout.widget.ConstraintLayout xmlns:android="http://schemas.android.com/apk/res-auto"
    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">

    <Button
        android:id="@+id/updateButton"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Update Text"
        app:layout_constraintBottom_toBottomOf="parent"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintTop_toTopOf="parent" />

    <TextView
        android:id="@+id/textView"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Initial Text"
        app:layout_constraintBottom_toTopOf="@+id/updateButton"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintTop_toTopOf="parent" />

</androidx.constraintlayout.widget.ConstraintLayout>
```

MainActivity.java

```
package com.fu.tt;

import androidx.appcompat.app.AppCompatActivity;

import android.os.Bundle;
import android.os.Handler;
import android.os.Looper;
import android.os.Message;
```



```

import android.util.Log;
import android.view.View;
import android.widget.Button;
import android.widget.TextView;
import android.widget.Toast;

public class MainActivity extends AppCompatActivity {

    private Button updateButton;
    private TextView textView;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        updateButton = findViewById(R.id.updateButton);
        textView = findViewById(R.id.textView);

        updateButton.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View v) {
                textView.post(new Runnable() {
                    @Override
                    public void run() {
                        textView.setText("Updating...");
                    }
                });

                textView.postDelayed(new Runnable() {
                    @Override
                    public void run() {
                        textView.setText("Updated!");
                    }
                }, 2000);
            }
        });
    }
}

```

🔗 Timer 和 TimerTask

1. Timer 类

`Timer` 类是 Java 提供的一个实用工具类，用于在将来的某个时间点调度任务。您可以使用 `schedule()` 方法指定要执行的任务（`TimerTask` 类型）以及任务的执行时间。这里有一个简单的示例：

```
import java.util.Timer;

public class TimerExample {
    public static void main(String[] args) {
        Timer timer = new Timer();
        timer.schedule(new MyTimerTask(), 3000);
    }
}
```

2. TimerTask 类

`TimerTask` 是一个实现了 `Runnable` 接口的抽象类，它定义了一个 `run()` 方法，该方法将在指定时间由 `Timer` 类执行。以下是一个简单的 `MyTimerTask` 示例：

MyTimerTask.java

```
package com.fu.tt;

import android.util.Log;

import java.util.TimerTask;

public class MyTimerTask extends TimerTask {
    @Override
    public void run() {
        Log.i("MyTimerTask", "Task executed!");
    }
}
```

要让任务定期执行，可以使用 `Timer` 类的 `scheduleAtFixedRate()` 方法。以下是一个简单的示例

```
import java.util.Timer;

public class TimerExample {
    public static void main(String[] args) {
        Timer timer = new Timer();
        timer.scheduleAtFixedRate(new MyTimerTask(), 0, 3000)
    }
}
```

在这个示例中，我们使用 `scheduleAtFixedRate()` 方法让 `MyTimerTask` 每隔 3 秒执行一次。首次执行任务将在 0 毫秒后开始，即立即开始。

将这两个类放在一个项目中，运行 `TimerExample` 类，您会看到控制台每隔 3 秒输出一次 "Task executed!"。

一个案例

如何在 `TimerTask` 中更新 UI 线程的 UI



```
package com.fu.tt;


import java.util.TimerTask;

public class MyTimerTask extends TimerTask {

    private MainActivity mainActivity;

    public MyTimerTask(MainActivity mainActivity) {
        this.mainActivity = mainActivity;
    }

    @Override
    public void run() {
        mainActivity.runOnUiThread(new Runnable() {
            @Override
            public void run() {
                mainActivity.updateTextView();
            }
        });
    }
}
```



```
package com.fu.tt;

import androidx.appcompat.app.AppCompatActivity;

import android.os.Bundle;
import android.os.Handler;
import android.os.Looper;
import android.os.Message;
import android.util.Log;
import android.view.View;
import android.widget.Button;
import android.widget.TextView;
import android.widget.Toast;
import java.util.Timer;

public class MainActivity extends AppCompatActivity {

    private Button updateButton;
```

```

private TextView textView;
private int counter = 0;

@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);

    updateButton = findViewById(R.id.updateButton);
    textView = findViewById(R.id.textView);


    Timer timer = new Timer();

    updateButton.setOnClickListener(new View.OnClickListener() {
        @Override
        public void onClick(View v) {
            Timer timer = new Timer();
            timer.scheduleAtFixedRate(new MyTimerTask(M
        }
    });
}

public void updateTextView() {
    counter++;
    textView.setText("Counter: " + counter);
}
}

```

或者是写在一起



```
import androidx.appcompat.app.AppCompatActivity;

import android.os.Bundle;
import android.os.Handler;
import android.widget.TextView;

import java.util.Timer;
import java.util.TimerTask;

public class MainActivity extends AppCompatActivity {

    private TextView textView;
    private int counter = 0;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        textView = findViewById(R.id.textView);

        Timer timer = new Timer();
        timer.scheduleAtFixedRate(new MyTimerTask(), 0, 100)
    }

    class MyTimerTask extends TimerTask {
        @Override
        public void run() {
            runOnUiThread(new Runnable() {
                @Override
                public void run() {
                    counter++;
                    textView.setText("Counter: " + counter)
                }
            });
        }
    }
}
```

`java.util.concurrent.ExecutorService` 是一个 Java 接口，它表示一个可以执行提交的 `Runnable` 和 `Callable` 任务的线程池。

它提供了一些方法，如

`execute(Runnable)`、`submit(Callable)`、`submit(Runnable)` 等，用于提交任务，并提供了如 `shutdown()` 和 `awaitTermination(long, TimeUnit)` 等方法来关闭线程池。这个接口是 Java 并发框架的核心组件之一，它的实现类为 `ThreadPoolExecutor`。

`java.util.concurrent.Executors` 是一个 Java 类，提供了一些静态工厂方法，用于创建不同类型的 `ExecutorService` 实例。这些方法包括：

- `newFixedThreadPool(int nThreads)`：创建一个固定线程数量的线程池。
- `newCachedThreadPool()`：创建一个缓存线程池，根据需要自动增加线程数量。
- `newSingleThreadExecutor()`：创建一个单线程执行器。
- `newScheduledThreadPool(int corePoolSize)`：创建一个调度线程池，用于在给定延迟后执行任务或定期执行任务。

通过使用 `Executors` 类提供的工厂方法，你可以轻松地创建各种类型的线程池，而不需要自己手动实例化 `ThreadPoolExecutor` 类并配置各种参数。这大大简化了线程池的创建过程。

总结一下，`ExecutorService` 是一个接口，表示一个线程池，而 `Executors` 是一个类，提供了创建 `ExecutorService` 实例的工厂方法。两者结合起来，使得在 Java 中创建和使用线程池变得简单和方便。

在 Android 开发中，线程池是一种有效管理多个线程的方法，以便更高效地处理并发任务。线程池可以重用已经创建的线程，从而减少了创建和销毁线程的开销。在 Java 和 Android 中，`ExecutorService` 是管理线程池的主要类。

以下是关于线程池的一些关键知识点，以及与之相关的代码示例：

1. 创建线程池

使用 `Executors` 类创建线程池。以下是创建不同类型线程池的方法：

- 固定线程池（Fixed Thread Pool）：拥有固定数量的线程。

```
javaCopy code
int numberOfThreads = 4;
ExecutorService fixedThreadPool = Executors.newFixedThreadPool(4);
```

- 单线程执行器（Single Thread Executor）：仅有一个线程的线程池。

```
javaCopy code
ExecutorService singleThreadExecutor = Executors.newSingleThreadExecutor();
```

- 缓存线程池（Cached Thread Pool）：根据需要创建新线程的线程池，但会在可能的情况下重用之前构造的线程。

```
javaCopy code
ExecutorService cachedThreadPool = Executors.newCachedThreadPool();
```

- 定时线程池（Scheduled Thread Pool）：用于执行定时任务的线程池。

```
javaCopy code
int numberOfThreads = 4;
ScheduledExecutorService scheduledThreadPool = Executors.newScheduledThreadPool(4);
```

2. 提交任务

使用 `ExecutorService` 的 `submit()` 方法提交任务。任务通常是实现了 `Runnable` 接口的类。


```
javaCopy code
class MyTask implements Runnable {
    @Override
    public void run() {
        // 任务逻辑
    }
}

MyTask task = new MyTask();
fixedThreadPool.submit(task);
```

也可以使用 Lambda 表达式创建任务：

```
javaCopy code
fixedThreadPool.submit(() -> {
    // 任务逻辑
});
```

3. 关闭线程池

在不再需要线程池时，使用 `shutdown()` 方法关闭线程池。这将启动有序的关闭，在关闭之前允许提交的任务完成。可以使用 `awaitTermination()` 方法等待线程池关闭。

```
javaCopy code
fixedThreadPool.shutdown();

try {
    if (!fixedThreadPool.awaitTermination(60, TimeUnit.SECONDS))
        fixedThreadPool.shutdownNow();
} catch (InterruptedException e) {
    fixedThreadPool.shutdownNow();
}
```

4. 定时任务

对于定时任务，使用 `ScheduledExecutorService`。以下是如何使用 `schedule()`、`scheduleAtFixedRate()` 和 `scheduleWithFixedDelay()` 方法执行定时任务：

- `schedule()`：在指定的延迟之后运行一次任务。

```
javaCopy code
Runnable task = () -> {
    // 任务逻辑
};
long delay = 5;
TimeUnit timeUnit = TimeUnit.SECONDS;
scheduledThreadPool.schedule(task, delay,
timeUnit);
```

- `scheduleAtFixedRate()`：在指定的初始延迟之后开始运行任务，并以固定的速率连续执行任务。如果任务的执行时间超过指定的周期，则下一个任务将在当前任务完成后立即开始。


```
javaCopy code
Runnable task = () -> {
    // 任务逻辑
};
long initialDelay = 1;
long period = 5;
TimeUnit timeUnit = TimeUnit.SECONDS;
scheduledThreadPool.scheduleAtFixedRate(task, initialDelay, period, timeUnit);
```

- `scheduleWithFixedDelay()`：在指定的初始延迟之后开始运行任务，并在每次任务完成后以固定的延迟执行下一个任务。

```
javaCopy code
Runnable task = () -> {
    // 任务逻辑
};
long initialDelay = 1;
long delay = 5;
TimeUnit timeUnit = TimeUnit.SECONDS;
scheduledThreadPool.scheduleWithFixedDelay(task, initialDelay, delay, timeUnit);
```

5. 自定义线程池

在某些情况下，您可能需要自定义线程池，以便更好地控制线程的创建、管理和回收。可以使用 `ThreadPoolExecutor` 类创建自定义线程池。



```
javaCopy code
int corePoolSize = 2;
int maximumPoolSize = 4;
long keepAliveTime = 60;
TimeUnit timeUnit = TimeUnit.SECONDS;
BlockingQueue<Runnable> workQueue = new
LinkedBlockingQueue<>();
ThreadFactory threadFactory =
Executors.defaultThreadFactory();
RejectedExecutionHandler handler = new
ThreadPoolExecutor.AbortPolicy();

ThreadPoolExecutor customThreadPool = new
ThreadPoolExecutor(
    corePoolSize, maximumPoolSize, keepAliveTime,
    timeUnit, workQueue, threadFactory, handler
);
```


这些参数的含义如下：

- `corePoolSize`：线程池维护的核心线程数。
- `maximumPoolSize`：线程池允许的最大线程数。
- `keepAliveTime`：非核心线程空闲等待新任务的最长时间。超过这个时间，非核心线程将被终止。
- `timeUnit`：`keepAliveTime` 参数的时间单位。
- `workQueue`：用于存储等待执行的任务的阻塞队列。
- `threadFactory`：用于创建新线程的工厂。
- `handler`：当线程池已满且队列已满时，拒绝处理新任务的处理程序。

🔗 线程池_fixedThreadPool（固定线程池）

ExecutorService 是一个可管理线程池的接口，提供了线程管理和任务执行的方法。FixedThreadPool 是 ExecutorService 的一种实现，它创建了一个具有固定线程数的线程池。

```
package com.fu.tt;
```



```

import androidx.appcompat.app.AppCompatActivity;

import android.os.Bundle;
import android.os.Handler;
import android.os.Looper;
import android.os.Message;
import android.util.Log;
import android.view.View;
import android.widget.Button;
import android.widget.TextView;
import android.widget.Toast;
import java.util.Timer;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

public class MainActivity extends AppCompatActivity {
    private static final String TAG = "MainActivity";
    private ExecutorService fixedThreadPool;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        Button button = findViewById(R.id.start_button);
        fixedThreadPool = Executors.newFixedThreadPool(3);

        button.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View v) {
                for (int i = 0; i < 100; i++) {
                    final int taskNumber = i;
                    fixedThreadPool.execute(new Runnable() {
                        @Override
                        public void run() {
                            try {
                                Thread.sleep(2000);
                            } catch (InterruptedException e) {
                                e.printStackTrace();
                            }
                            Log.d(TAG, "Task " + taskNumber);
                        }
                    });
                }
            }
        });
    }
}

```

```

        }
    }
});
}

@Override
protected void onDestroy() {
    super.onDestroy();
    if (fixedThreadPool != null) {
        // 关闭线程池
        fixedThreadPool.shutdown();
    }
}
}
}

```

1. 线程复用：线程池中的线程在完成任务后不会立即销毁，而是等待新的任务分配给它。这种复用机制避免了频繁创建和销毁线程带来的性能开销。
2. 任务调度：线程池负责管理任务队列，根据线程的可用性来调度任务。在这个例子中，FixedThreadPool 的线程数量为3，因此线程池会并发执行3个任务。当一个任务完成时，空闲的线程会开始执行下一个任务。任务将在队列中等待，直到有可用的线程来执行它们。（一次性打印三个日志）
3. 资源控制：线程池可以控制同时运行的线程数量，从而避免过多的线程消耗系统资源。在这个例子中，我们创建了一个具有3个线程的FixedThreadPool，这意味着在任何时候，最多只会有3个线程同时执行任务。这有助于提高应用程序的性能，特别是在资源受限的设备上。

🔗 线程池_SingleThreadExecutor（单线程执行器）

单线程执行器（Single Thread Executor）是一种特殊类型的线程池，它只有一个工作线程。这意味着它同时只能执行一个任务，而其他任务将排队等待。单线程执行器的一个主要优点是它可以保证任务按照提交的顺序依次执行。这在某些场景下非常有用，比如你需要按顺序执行一系列任务，而不希望它们在多个线程中并发执行

```
package com.fu.tt;
```



```
import androidx.appcompat.app.AppCompatActivity;

import android.os.Bundle;
import android.os.Handler;
import android.os.Looper;
import android.os.Message;
import android.util.Log;
import android.view.View;
import android.widget.Button;
import android.widget.TextView;
import android.widget.Toast;
import java.util.Timer;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

public class MainActivity extends AppCompatActivity {
    private static final String TAG = "MainActivity";
    private ExecutorService executorService;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        Button button = findViewById(R.id.start_button);

        // 创建一个单线程执行器
        executorService = Executors.newSingleThreadExecutor()

        button.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View v) {
                for (int i = 0; i < 10; i++) {
                    final int taskNumber = i;
                    executorService.execute(new Runnable() {
                        @Override
                        public void run() {
                            try {
                                Thread.sleep(2000);
                            } catch (InterruptedException e) {
                                e.printStackTrace();
                            }
                            Log.d(TAG, "Task " + taskNumber
                        }
                    }
                }
            }
        })
    }
}
```

```

        });
    }
}

@Override
protected void onDestroy() {
    super.onDestroy();
    if (executorService != null) {
        executorService.shutdown();
    }
}
}

```

🔗 线程池_cachedThreadPool（缓存线程池）

缓存线程池（Cached Thread Pool）是一种特殊类型的线程池，它的工作线程数量可以根据需要动态调整。当有新任务提交到缓存线程池时，如果有空闲的工作线程，那么这个任务将立即被这个工作线程执行；如果没有空闲的工作线程，线程池会创建一个新的工作线程来执行这个任务。此外，当工作线程空闲一段时间后（默认为60秒），它将被自动销毁以减少资源占用。

缓存线程池适用于执行大量短时任务的场景，因为它可以根据任务数量动态调整线程数量，从而提高系统资源的利用率。

```

import androidx.appcompat.app.AppCompatActivity;

import android.os.Bundle;
import android.widget.TextView;

import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

public class MainActivity extends AppCompatActivity {

    private TextView textView;
    private ExecutorService cachedThreadPool;

```

```

@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);

    textView = findViewById(R.id.textView);

    // 创建一个缓存线程池
    cachedThreadPool = Executors.newCachedThreadPool();

    // 提交任务
    for (int i = 0; i < 10; i++) {
        int taskNumber = i + 1;
        cachedThreadPool.execute(() -> {
            try {
                Thread.sleep(1000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }

            // 更新 UI
            runOnUiThread(() -> textView.append("Task "

        ));
    }
}

@Override
protected void onDestroy() {
    super.onDestroy();
    // 关闭线程池
    if (cachedThreadPool != null) {
        cachedThreadPool.shutdown();
    }
}
}

```

🔗 线程池_ScheduledThreadPool (定时任务)

定时线程池（Scheduled Thread Pool）允许您在指定延迟之后运行任务，或者定期执行任务。在 Android 开发中，您可以使用

`java.util.concurrent.ScheduledThreadPoolExecutor` 类来创建和管理定时线程池。

scheduleAtFixedRate（周期）

```
import androidx.appcompat.app.AppCompatActivity;
import android.os.Bundle;
import android.view.View;
import android.widget.Button;
import android.widget.TextView;
import java.util.concurrent.Executors;
import java.util.concurrent.ScheduledExecutorService;
import java.util.concurrent.TimeUnit;

public class MainActivity extends AppCompatActivity {

    private TextView textView;
    private Button startButton;
    private int counter = 0;
    private ScheduledExecutorService scheduledExecutorService;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        textView = findViewById(R.id.text_view);
        startButton = findViewById(R.id.start_button);

        scheduledExecutorService = Executors.newScheduledThreadPool(1);

        startButton.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View v) {
                scheduledExecutorService.scheduleAtFixedRate(new Runnable() {
                    @Override
                    public void run() {
                        runOnUiThread(new Runnable() {
                            @Override
                            public void run() {
                                counter++;
                                textView.setText(counter + "");
                            }
                        });
                    }
                }, 0, 1, TimeUnit.SECONDS);
            }
        });
    }
}
```

```

        counter++;
        textView.setText("Counter:
    }
    });
    }
    }, 0, 1, TimeUnit.SECONDS);
}
});
}

@Override
protected void onDestroy() {
    super.onDestroy();
    scheduledExecutorService.shutdown();
}
}


```

`Executors.newScheduledThreadPool(1)` 创建一个包含一个线程的定时线程池，然后，当用户点击 `startButton` 时，我们使用 `scheduleAtFixedRate` 方法安排一个定期运行的任务，注意：**每次点击按钮时，你都会向线程池中提交一个新的定时任务。这些任务会在同一个线程中依次执行。因此，在你的示例中，你会有多个定时任务在同一个线程中同时运行，导致计数器加速。并不是因为点击一次 START 就新增一个线程，多个线程同时运行，导致的计时速度加快**

```
scheduleAtFixedRate(new Runnable, 0, 1, TimeUnit.SECONDS)
```




1. `Runnable command`: 要执行的任务。这是一个实现了 `Runnable` 接口的对象，表示要执行的代码。
2. `long initialDelay`: 初始延迟。这是任务首次执行前的等待时间。这个值的单位是由第 4 个参数指定的。
3. `long period`: 任务执行的间隔。这是连续任务执行之间的时间间隔。这个值的单位是由第 4 个参数指定的。
4. `TimeUnit unit`: 时间单位。这是一个枚举类型，用于指定初始延迟和执行间隔的时间单位。可用的单位包括 `NANOSECONDS`（纳秒）、`MICROSECONDS`（微秒）、`MILLISECONDS`（毫秒）、`SECONDS`（秒）、`MINUTES`（分钟）、`HOURS`（小时）和 `DAYS`（天）。

```
scheduledExecutorService.scheduleAtFixedRate(new Runnable()   
    @Override  
    public void run() {  
        // ... 更新 UI ...  
    }  
}, 0, 1, TimeUnit.SECONDS);
```

这意味着：

1. 我们创建了一个新的 `Runnable` 对象，该对象包含要执行的任务（在本例中为更新 UI）。
2. 初始延迟为 0，任务将立即开始执行。
3. 任务执行间隔为 1 秒，即每隔 1 秒执行一次任务。
4. 时间单位为 `TimeUnit.SECONDS`，表示初始延迟和执行间隔都以秒为单位。

如果你想将十个任务分配给十个线程，你只需要创建一个拥有十个线程的线程池，每一次 `scheduledExecutorService.scheduleAtFixedRate` 都会将新的任务分配空闲的线程，如果，你的没有空闲的，就会加在一个线程上面添加多个任务 通过点按按钮的方式，给十个线程分配任务

```
package com.fu.tt;   
  
import androidx.appcompat.app.AppCompatActivity;  
import android.os.Bundle;  
import android.view.View;  
import android.widget.Button;  
import android.widget.TextView;  
import java.util.concurrent.Executors;  
import java.util.concurrent.ScheduledExecutorService;  
import java.util.concurrent.TimeUnit;  
  
public class MainActivity extends AppCompatActivity {  
  
    private TextView textView;  
    private Button startButton;  
    private int counter = 0;
```

```

private ScheduledExecutorService scheduledExecutorServi

@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);

    textView = findViewById(R.id.textView);
    startButton = findViewById(R.id.startButton);

    scheduledExecutorService = Executors.newScheduledTh

    startButton.setOnClickListener(new View.OnClickListener()
        @Override
        public void onClick(View v) {
            scheduledExecutorService.scheduleAtFixedRate(
                @Override
                public void run() {
                    runOnUiThread(new Runnable() {
                        @Override
                        public void run() {
                            counter++;
                            textView.setText("Counter:
                        }
                    });
                }, 0, 1, TimeUnit.SECONDS);
        }
    });
}

@Override
protected void onDestroy() {
    super.onDestroy();
    scheduledExecutorService.shutdown();
}
}

```

一次性就执行十次，让我们的线程池满负荷运行

```
package com.fu.tt;
```



```

import androidx.appcompat.app.AppCompatActivity;
import android.os.Bundle;
import android.view.View;
import android.widget.Button;
import android.widget.TextView;
import java.util.concurrent.Executors;
import java.util.concurrent.ScheduledExecutorService;
import java.util.concurrent.TimeUnit;

public class MainActivity extends AppCompatActivity {

    private TextView textView;
    private Button startButton;
    private int counter = 0;
    private ScheduledExecutorService scheduledExecutorService;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        textView = findViewById(R.id.textView);
        startButton = findViewById(R.id.startButton);

        scheduledExecutorService = Executors.newScheduledThreadPool(1);

        startButton.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View v) {
                for (int i = 0; i < 10; i++) {
                    scheduledExecutorService.scheduleAtFixedRate(new Runnable() {
                        @Override
                        public void run() {
                            runOnUiThread(new Runnable() {
                                @Override
                                public void run() {
                                    counter++;
                                    textView.setText("Count: " + counter);
                                }
                            });
                        }
                    }, 0, 1, TimeUnit.SECONDS);
                }
            }
        });
    }
}

```

```

    }

    @Override
    protected void onDestroy() {
        super.onDestroy();
        scheduledExecutorService.shutdown();
    }
}

```

schedule (延迟)

`ScheduledExecutorService` 的 `schedule` 方法允许你在指定的延迟后执行一个一次性任务。

```

package com.fu.tt;

import androidx.appcompat.app.AppCompatActivity;
import android.os.Bundle;
import android.view.View;
import android.widget.Button;
import android.widget.TextView;

import java.util.concurrent.Executors;
import java.util.concurrent.ScheduledExecutorService;
import java.util.concurrent.TimeUnit;

public class MainActivity extends AppCompatActivity {

    private TextView textView;
    private Button startButton;
    private ScheduledExecutorService scheduledExecutorServi

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        textView = findViewById(R.id.textView);
        startButton = findViewById(R.id.startButton);
    }
}

```

```

        scheduledExecutorService = Executors.newScheduledTh

startButton.setOnClickListener(new View.OnClickListener()
    @Override
    public void onClick(View v) {
        scheduledExecutorService.schedule(new Runna
            @Override
            public void run() {
                runOnUiThread(new Runnable() {
                    @Override
                    public void run() {
                        textView.setText("Task exec
                    }
                });
            }
        }, 3, TimeUnit.SECONDS);
    }
});
}

@Override
protected void onDestroy() {
    super.onDestroy();
    scheduledExecutorService.shutdown();
}
}

```

在这个例子中，我们有一个 `TextView` 和一个 `Button`。当用户点击按钮时，将在延迟 3 秒后执行一个任务，该任务将更新 `TextView` 的文本。请注意，这个任务只会执行一次，不会周期性地重复。

`scheduledExecutorService.schedule()` 方法有三个参数：

1. `Runnable`：要执行的任务。
2. `long delay`：任务开始执行前的延迟时间。
3. `TimeUnit unit`：延迟时间的单位（如 `TimeUnit.SECONDS`、`TimeUnit.MILLISECONDS` 等）。

🔗 自创建线程池

`ThreadPoolExecutor` 是 `java.util.concurrent` 包中的一个类，它允许你创建自定义的线程池。与 `Executors` 类提供的预定义线程池不同，使用 `ThreadPoolExecutor`，你可以更好地控制线程池的行为，例如核心线程数、最大线程数、线程空闲时间等。

```
package com.fu.tt;

import androidx.appcompat.app.AppCompatActivity;
import android.os.Bundle;
import android.view.View;
import android.widget.Button;
import android.widget.TextView;

import java.util.concurrent.LinkedBlockingQueue;
import java.util.concurrent.ThreadPoolExecutor;
import java.util.concurrent.TimeUnit;

public class MainActivity extends AppCompatActivity {

    private TextView textView;
    private Button startButton;
    private ThreadPoolExecutor threadPoolExecutor;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        textView = findViewById(R.id.textView);
        startButton = findViewById(R.id.startButton);

        int corePoolSize = 2;
        int maximumPoolSize = 4;
        long keepAliveTime = 10;
        TimeUnit unit = TimeUnit.SECONDS;
        LinkedBlockingQueue<Runnable> workQueue = new Linke

        threadPoolExecutor = new ThreadPoolExecutor(corePoo

        startButton.setOnClickListener(new View.OnClickListener
```



```


@Override
public void onClick(View v) {
    threadPoolExecutor.execute(new Runnable() {
        @Override
        public void run() {
            try {
                Thread.sleep(3000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            runOnUiThread(new Runnable() {
                @Override
                public void run() {
                    textView.setText("Task exec
                });
            });
        }
    });
}

@Override
protected void onDestroy() {
    super.onDestroy();
    threadPoolExecutor.shutdown();
}
}

```

1. `corePoolSize`：线程池中的核心线程数。即使线程空闲，线程池也会保持这些核心线程。
2. `maximumPoolSize`：线程池中允许的最大线程数。当工作队列已满时，线程池会创建新线程，直到达到最大线程数。
3. `keepAliveTime`：非核心线程空闲时的最长保持时间。超过这个时间，非核心线程将被终止。
4. `unit`：`keepAliveTime` 的时间单位。
5. `workQueue`：用于存储等待执行的任务的阻塞队列。

当用户点击按钮时，我们将一个任务提交给线程池。这个任务会模拟耗时操作（通过 `Thread.sleep(3000)`），然后在主线程上更新 `TextView` 的文本。

```
LinkedBlockingQueue<Runnable> workQueue = new LinkedBlockin 
```

`LinkedBlockingQueue` 是 `java.util.concurrent` 包中的一个类，它实现了一个线程安全的阻塞队列。这个队列用于存储等待执行的任务。它是 `ThreadPoolExecutor` 构造函数的一个参数。

阻塞队列（`BlockingQueue`）是一种特殊的队列，它支持在多线程环境下的插入和提取操作。其中，“阻塞”的含义是指，当队列为空时，从队列中获取元素的操作会被阻塞，直到队列中有元素为止。同样，当队列已满时，向队列中添加元素的操作也会被阻塞，直到队列中有空位置为止。

`LinkedBlockingQueue` 是一种实现了阻塞队列接口的类，它使用链表作为底层数据结构。它是线程安全的，意味着在多线程环境下，你不需要担心多个线程同时访问队列时的同步问题。

在线程池中，`LinkedBlockingQueue` 用于存储等待执行的任务。当你向线程池提交任务时，任务会被添加到 `LinkedBlockingQueue` 中。线程池中的线程会从队列中获取任务并执行它们。如果队列为空，线程会等待，直到有新任务提交到队列为止。

举个例子，假设你有一个线程池，它有3个线程。同时，你向线程池提交了5个任务。线程池中的3个线程将立即开始执行前3个任务。而剩下的2个任务会被添加到 `LinkedBlockingQueue` 中等待执行。当线程池中的一个线程完成了它的任务，它将从队列中获取下一个任务并执行。这样，队列中的任务会按照先进先出（FIFO）的顺序被处理。

总之，`LinkedBlockingQueue` 是一种线程安全的、基于链表的阻塞队列。在线程池中，它用于存储等待执行的任务，并且支持多线程环境下的安全操作。

🔗 泛型（Generics）

泛型（Generics）是 Java 编程语言中的一个重要特性，它允许在类、接口和方法中使用类型参数。泛型的主要目的是提高代码的可重用性，减少代码重复，并增加类型安全性。

****就是不在意数据的类型****

在编译时，Java 编译器会将泛型信息擦除，因此在运行时泛型类型的信息是不可用的。这意味着泛型实例在运行时的类型实际上是它们的原始类型

🔗 泛型类和泛型接口

泛型类和泛型接口是带有一个或多个类型参数的类和接口。例如，Java 集合框架中的 `List<E>` 和 `Map<K, V>` 都是泛型接口。在这些接口中，`E`、`K` 和 `V` 分别表示元素类型、键类型和值类型。

一个例子


1. 首先，创建一个名为 `Storage` 的泛型接口，包含 `add`、`get` 和 `size` 方法。

```
public interface Storage<T> {  
    void add(T item);  
    T get(int index);  
    int size();  
}
```



在这个泛型接口中，我们使用了泛型参数 `T`，表示要存储的元素类型

创建一个实现了 `Storage` 接口的泛型类 `ArrayStorage`，用于将元素存储在一个数组中。



```
import java.util.Arrays;

public class ArrayStorage<T> implements Storage<T> {
    private Object[] items;
    private int size;

    public ArrayStorage() {
        items = new Object[10];
        size = 0;
    }

    @Override
    public void add(T item) {
        if (size == items.length) {
            items = Arrays.copyOf(items, size * 2);
        }
        items[size++] = item;
    }

    @SuppressWarnings("unchecked")
    @Override
    public T get(int index) {
        if (index < 0 || index >= size) {
            throw new IndexOutOfBoundsException("Index: " +
        }
        return (T) items[index];
    }

    @Override
    public int size() {
        return size;
    }
}
```

在这个泛型类中，我们使用了与 `Storage` 接口相同的泛型参数 `T`。注意，我们在 `ArrayStorage` 类中使用了一个 `Object[]` 数组来存储元素，并在获取元素时进行了类型转换。

在你的 `MainActivity` 类中，测试 `ArrayStorage` 类的使用



```
import androidx.appcompat.app.AppCompatActivity;
import android.os.Bundle;
import android.widget.TextView;

public class MainActivity extends AppCompatActivity {

    private TextView textView;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        textView = findViewById(R.id.textview);


        // 使用泛型类 ArrayStorage 存储 String 类型的元素
        Storage<String> stringStorage = new ArrayStorage<>(
            stringStorage.add("Hello");
            stringStorage.add("World");

        // 使用泛型类 ArrayStorage 存储 Integer 类型的元素
        Storage<Integer> integerStorage = new ArrayStorage<
            integerStorage.add(1);
            integerStorage.add(2);
            integerStorage.add(3);

        StringBuilder sb = new StringBuilder();
        sb.append("String Storage: ").append(stringStorage.
            sb.append("Integer Storage: ").append(integerStorag

        textView.setText(sb.toString());
    }
}
```

```
Storage<Integer> integerStorage = new ArrayStorage<>();
```



因为在声明类的时候，并没有声明他是什么类型的类，不知道里面的函数都是处理什么类型的数据，所以在创建的时候指定，他的数据类型

实际上，他的写法应该是

```
Storage<Integer> integerStorage = new ArrayStorage<Integer>
```

但是在java7以后就简化了这样的写法。

🔗 泛型方法

```
package com.fu.tt;

import java.util.List;

public class Util {
    public static <T> int findFirstOccurrence(List<T> list,
        for (int i = 0; i < list.size(); i++) {
            if (list.get(i).equals(target)) {
                return i;
            }
        }
        return -1;
    }
}
```



```
package com.fu.tt;

import androidx.appcompat.app.AppCompatActivity;
import android.os.Bundle;
import android.view.View;
import android.widget.Button;
import android.widget.TextView;

import java.util.Arrays;
import java.util.List;
import java.util.concurrent.LinkedBlockingQueue;
import java.util.concurrent.ThreadPoolExecutor;
import java.util.concurrent.TimeUnit;

public class MainActivity extends AppCompatActivity {

    private TextView textView;
    private Button startButton;
    private ThreadPoolExecutor threadPoolExecutor;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        textView = findViewById(R.id.textView);
        List<String> names = Arrays.asList("Alice", "Bob",
        int index_names = Util.findFirstOccurrence(names, "

        List<Integer> age = Arrays.asList(1,2, 3, 4, 5);
        int index_age = Util.findFirstOccurrence(names, "Ch

        textView.setText("First occurrence of 'Charlie' is
    }

    @Override
    protected void onDestroy() {
        super.onDestroy();
    }
}
```

🔗 类型参数和类型参数约束

类型参数约束是指对泛型中可用类型的限制。这可以通过使用 `extends` 关键字来实现，表示类型参数必须是指定类型的子类（或实现了指定接口）。

简单来说，不是一个泛型可以衍生为多个类型嘛，那么约束就是，只接受指定的类型类的衍生类型

```
// Shape 接口及其实现类
public interface Shape {
    double getArea();
}

public class Circle implements Shape {
    private double radius;

    public Circle(double radius) {
        this.radius = radius;
    }

    @Override
    public double getArea() {
        return Math.PI * radius * radius;
    }
}

public class Rectangle implements Shape {
    private double width;
    private double height;

    public Rectangle(double width, double height) {
        this.width = width;
        this.height = height;
    }

    @Override
    public double getArea() {
        return width * height;
    }
}

// Box 泛型类，使用类型参数约束
public class Box<T extends Shape> {
    private T shape;
```



```

    public Box(T shape) {
        this.shape = shape;
    }

    public T getShape() {
        return shape;
    }

    public void setShape(T shape) {
        this.shape = shape;
    }
}

// 使用 Box 泛型类的示例代码
public class Main {
    public static void main(String[] args) {
        Box<Circle> circleBox = new Box<>(new Circle(5));
        Box<Rectangle> rectangleBox = new Box<>(new Rectang

        // 下面这行代码将导致编译错误，因为 String 类型不符合 T ext
        // Box<String> stringBox = new Box<>("Error");

        System.out.println("Circle area: " + circleBox.getS
        System.out.println("Rectangle area: " + rectangleBo

    }
}

```

🔗 通配符

一种更加灵活的忽略数据类型

1. 无限制通配符 (<?>)

假设我们有一个方法，它需要接受一个 `List`，并打印其中的所有元素。我们可以使用无限制通配符 (<?>) 来表示未知的类型：

```
public void printListItems(List<?> list) {  
    for (Object item : list) {  
        System.out.println(item);  
    }  
}
```



那这个和我们的刚刚的泛型有什么区别呢，都是忽略

泛型是声明的时候，不知道自己的接受的是什么类型的数据，但是一旦建立，就知道自己创建的是什么类型的数据，就已经确定了。

通配符是，你在传输数据的时候都不知道，自己传进去的是什么类型的数据

详细的关注，for 循环的内容

```
public <T> void printListItems1(List<T> list) {  
    // 他非常确定，数据类型就是 T  
    for (T item : list) {  
        Log.i("tag",String.valueOf(item));  
    }  
}  
  
public void printListItems2(List<?> list) {  
    // for 循环的时候也不清楚自己是什么数据类型，只能用所有数据类型  
    for (Object item : list) {  
        Log.i("tag",String.valueOf(item));  
    }  
}
```



2. 上界通配符 (<? extends T>)

上界通配符用于限制未知类型的上限。例如，我们有一个 `Animal` 类和几个子类（如 `Dog` 和 `Cat`）。我们想创建一个接受 `Animal` 及其子类的列表的方法。这时，我们可以使用上界通配符：



```
public abstract class Animal {
    public abstract void makeSound();
}


public class Dog extends Animal {
    public void makeSound() {
        System.out.println("Woof!");
    }
}

public class Cat extends Animal {
    public void makeSound() {
        System.out.println("Meow!");
    }
}

public void playSounds(List<? extends Animal> animals) {
    // 不知道传进来的是什么类型，只能用父类来接受数据，因为，必定是 An
    for (Animal animal : animals) {
        animal.makeSound();
    }
}
```

3. 下界通配符 (<? super T>)

下界通配符用于限制未知类型的下限。例如，我们想创建一个将元素添加到列表的方法。这个方法应该接受任何 `Number` 类型及其超类的列表，例如 `List<Number>`、`List<Object>` 等。这时，我们可以使用下界通配符：



```
public void addNumberToList(List<? super Number> list, Number number) {
    list.add(number);
}
```

使用 `<? super Number>`，我们可以确保此方法接受 `Number` 及其超类的列表。

结合上界通配符（<? extends T>）来解释一下。上界通配符用于限制泛型参数的类型必须是某个类的子类（或其自身）。上界通配符在处理泛型集合中的对象时很有用，因为它们允许你处理泛型集合中的不同类型的对象

```
public class Animal {  
    // ...  
}  
  
public class Mammal extends Animal {  
    // ...  
}  
  
public class Dog extends Mammal {  
    // ...  
}
```

```
public static void processAnimals(List<? extends Animal> animals) {  
    for (Animal animal : animals) {  
        // 处理 animal 对象  
    }  
}
```

这里，我们使用了上界通配符 `<? extends Animal>` 来指定列表中的元素必须是 `Animal` 类型或其子类型。现在，我们可以使用这个方法处理不同类型的动物列表：

```
List<Animal> animalList = new ArrayList<>();  
List<Mammal> mammalList = new ArrayList<>();  
List<Dog> dogList = new ArrayList<>();
```



```
// 有效  
processAnimals(animalList);
```

```
// 有效  
processAnimals(mammalList);
```

```
// 有效  
processAnimals(dogList);
```

现在，我们可以将 `Animal`、`Mammal` 和 `Dog` 类型的对象列表传递给 `processAnimals` 方法，这是因为我们使用了上界通配符 `<? extends Animal>`。

现在，我们对比一下上界通配符和下界通配符：

- 上界通配符（`<? extends T>`）：限制泛型参数的类型必须是 `T` 或其子类。它在读取泛型集合中的对象时很有用，因为它们允许你处理泛型集合中的不同类型的对象。
- 下界通配符（`<? super T>`）：限制泛型参数的类型必须是 `T` 或其父类。它在将对象写入泛型集合时很有用，因为它们允许你将更具体的类型（子类）添加到泛型集合中。

```
public class Animal {
    // ...
}

public class Mammal extends Animal {
    // ...
}

public class Dog extends Mammal {
    // ...
}
```

假设我们想要实现一个方法，该方法可以将 `Mammal` 或其子类的对象添加到 `List` 中，该 `List` 可能包含 `Mammal` 或其父类的对象。为此，我们可以使用下界通配符 `<? super Mammal>`：

```
public static void addMammal(List<? super Mammal> mammals) {
    mammals.add(new Mammal());
    mammals.add(new Dog());
}
```

现在，我们可以使用这个方法将 `Mammal` 和 `Dog` 对象添加到 `Animal` 和 `Mammal` 类型的列表中：

```
List<Animal> animalList = new ArrayList<>();
List<Mammal> mammalList = new ArrayList<>();

// 有效
addMammal(animalList);

// 有效
addMammal(mammalList);

// 注意：以下代码是无效的，因为 List<Dog> 只能容纳 Dog 对象，而不能容纳 Mammal 对象
// List<Dog> dogList = new ArrayList<>();
// addMammal(dogList);
```

🔗 AsyncTask 类

自 Android 11（API 级别 30）开始已被弃用

AsyncTask 类是 Android 提供的一个便捷的类，用于在后台线程上执行异步任务，同时在 UI 线程上更新用户界面。AsyncTask 通常用于执行较短时间的后台任务，如从网络加载数据、文件操作或者数据库查询等。

AsyncTask 的工作原理是在后台线程上执行任务，然后在任务完成时，通知 UI 线程进行界面更新。这样可以确保用户界面保持响应，避免因耗时操作而卡住。

它允许你在 UI 线程之外执行耗时操作，然后将结果发布到 UI 线程，从而避免阻塞主线程。AsyncTask 类提供了一个简单的方法来实现异步操作。

他跟，Thread 单开一个线程，然后通过 `runOnUiThread` 或者是通过 `Handler` 发送任务的方式更新 UI 线程的 UI 有什么区别？

AsyncTask 存在的意义在于它提供了一个简化的、更易于理解和使用的方法来处理后台任务和 UI 线程之间的通信。与直接使用 `Thread` 和 `Handler` 或者 `runOnUiThread` 不同，AsyncTask 将这些步骤封装在一个类中，并提供了一个更加结构化的方法来处理后台任务和 UI 更新。

以下是 AsyncTask 和其他方法之间的一些区别：

1. 结构化：AsyncTask 将后台任务执行和 UI 更新分离到不同的方法中（`doInBackground`、`onProgressUpdate` 和 `onPostExecute`），这使得代码更加清晰和易于理解。
2. 线程池管理：AsyncTask 内部使用了线程池来处理并发任务，而不是为每个任务创建一个新的线程。这有助于提高性能并降低资源消耗。
3. 进度更新：AsyncTask 提供了一种简单的方法来更新任务进度。使用 `publishProgress` 方法，你可以直接从 `doInBackground` 方法中触发 `onProgressUpdate` 方法，而不需要单独处理 `Handler`。
4. 生命周期管理：AsyncTask 可以处理与 Activity 生命周期相关的问题，例如在旋转屏幕时正确取消任务以避免内存泄漏。

然而，AsyncTask 并非适用于所有场景。它主要适用于短暂的、与 UI 交互的后台任务。对于长时间运行的后台任务或需要更细粒度控制的场景，使用其他方法（如 Service、Thread、Handler 或其他异步处理库）可能更合适。总之，AsyncTask 是一个易于使用的工具，适用于处理简单的后台任务和 UI 更新。

```
import android.os.AsyncTask;
import android.widget.TextView;

// 注意这里的，参数就是下面的函数的参数，并且，这里的参数不是固定的，比
public class MyAsyncTask extends AsyncTask<Void, Integer, S

    private TextView textView;

    public MyAsyncTask(TextView textView) {
        this.textView = textView;
    }

// 这个方法在主线程上执行，在异步任务开始之前调用。通常用于进行一些
@Override
protected void onPreExecute() {
    super.onPreExecute();
    textView.setText("Starting...");
}

// 一个方法，你需要在其中编写在后台线程执行的任务。当你执行一个 Asy
@Override
protected String doInBackground(Void... voids) {
    for (int i = 1; i <= 5; i++) {
        try {
            Thread.sleep(1000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        // 是 AsyncTask 类中的一个方法，它用于在后台任务中更新 UI
        publishProgress(i);
    }
    return "Finished!";
}

// 这个主要是用来显示进度
@Override
protected void onProgressUpdate(Integer... values) {
```



```
        super.onProgressUpdate(values);
        textView.setText("Counter: " + values[0]);
    }

    @Override
    protected void onPostExecute(String result) {
        super.onPostExecute(result);
        textView.setText(result);
    }
}
```

`publishProgress` 和 `onProgressUpdate` 之间的关系是： `publishProgress` 负责从后台线程发布进度，而 `onProgressUpdate` 负责在主线程上处理进度更新。

`MyAsyncTask` 类有四个泛型参数：

1. Params - 传递给后台任务的参数类型（在此示例中为 Void）
2. Progress - 在任务执行期间发布进度更新的类型（在此示例中为 Integer）
3. Result - 后台任务完成后返回的结果类型（在此示例中为 String）

`doInBackground` 方法是在新线程中执行的，用于执行后台任务。在此示例中，我们将休眠 5 秒并更新计数器。 `publishProgress` 方法用于发布进度更新，将触发 `onProgressUpdate` 方法的调用。

`onProgressUpdate` 和 `onPostExecute` 方法将在 UI 线程上执行，因此可以直接更新 UI。

接下来，你可以在 Activity 中使用这个 AsyncTask：



```
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 {

    private TextView textView;
    private Button startButton;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        textView = findViewById(R.id.textView);
        startButton = findViewById(R.id.startButton);

        startButton.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View v) {
                new MyAsyncTask(textView).execute();
            }
        });
    }
}
```

体现可变参数的例子



```
package com.fu.tt;

import android.os.AsyncTask;
import android.widget.TextView;

public class CounterTask extends AsyncTask<Integer, Integer,
    private TextView counterTextView;

    public CounterTask(TextView textView) {
        this.counterTextView = textView;
    }

    @Override
    protected Void doInBackground(Integer... params) {
        int maxCount = params[0];
        for (int i = 1; i <= maxCount; i++) {
            try {
                Thread.sleep(1000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            publishProgress(i, maxCount);
        }
        return null;
    }

    @Override
    protected void onProgressUpdate(Integer... values) {
        int currentCount = values[0];
        int totalCount = values[1];
        counterTextView.setText("Counter: " + currentCount
    }
}
```



```
package com.fu.tt;

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 {

    private TextView textView;
    private Button startButton;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        textView = findViewById(R.id.textview);
        startButton = findViewById(R.id.startButton);

        startButton.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View v) {
                new CounterTask(textView).execute(10); // 他
            }
        });
    }
}
```



```
new CounterTask(textView).execute(10);
```

`new CounterTask().execute(10)` 这行代码的作用是创建一个新的 `CounterTask` 对象，并调用其 `execute` 方法来执行异步任务。这里的 `10` 是传递给 `execute` 方法的参数。

在我们的例子中，`10` 作为参数传递给了 `CounterTask` 类中的 `doInBackground` 方法。`AsyncTask` 类在执行 `doInBackground` 方法时，会将 `execute` 方法接收到的参数传递给它。

在这个示例中，`10` 表示我们希望计数器从 1 计数到 10。因此，当我们在 `doInBackground` 方法中通过 `params[0]` 获取参数时，我们得到的值就是 `10`，这就是我们的最大计数值。

`AsyncTask` 类的 `execute` 方法可以接受可变数量的参数，这些参数将作为输入传递给 `doInBackground` 方法。参数的类型需要与 `AsyncTask` 的第一个泛型参数相匹配。例如，在我们的 `CounterTask` 示例中，我们使用了 `Integer` 类型的泛型参数，因此可以传递一个或多个 `Integer` 类型的参数。

如果你想为 `doInBackground` 方法传递多个参数，你可以在调用 `execute` 方法时提供这些参数。例如：

```
new CounterTask().execute(10, 20, 30);  
  
// 在 doInBackground 方法中，你可以通过 params 数组访问这些参数：  
@Override  
protected String doInBackground(Integer... params) {  
    int firstParam = params[0]; // 10  
    int secondParam = params[1]; // 20  
    int thirdParam = params[2]; // 30  
    // ...  
}
```

`doInBackground` 方法可以接受任何类型的参数，而不仅仅是 `Void...` 或 `Integer...`。`AsyncTask` 类是泛型的，因此可以使用任何类型作为输入参数。例如，你可以使用 `String` 类型，`Double` 类型，自定义类等作为输入参数。泛型参数的类型与你在创建 `AsyncTask` 子类时指定的泛型参数类型相匹配。

例如，如果你想使用 `String` 类型的参数，你可以这样创建 `AsyncTask` 子类：

```
private class MyTask extends AsyncTask<String, Void, Void>
    @Override
    protected Void doInBackground(String... params) {
        // 在这里使用 String 类型的参数
    }

    // ...
}
```

```
new MyTask().execute("parameter1", "parameter2");
```

```
@Override
protected void onPostExecute(String result) {
    textView.setText(result);
}
```

`onPostExecute` 方法是 `AsyncTask` 的一个回调方法，它在**主线程（UI线程）上运行**。当 `doInBackground` 方法在后台线程执行完任务并返回结果时，`onPostExecute` 方法会被自动调用。在 `onPostExecute` 方法内，你可以使用 `doInBackground` 返回的结果来更新UI或执行其他与UI相关的操作。

`onPostExecute` 方法接收一个参数，这个参数的类型与 `AsyncTask` 的第三个泛型参数（`Result`）相同。这个参数就是 `doInBackground` 方法返回的结果。

在这个例子中，`onPostExecute` 方法接收一个 `String` 类型的参数 `result`，这个参数就是 `doInBackground` 方法返回的结果。在这个方法内，我们使用 `result` 更新 `textView` 的文本。

总之，`onPostExecute` 方法的主要目的是在后台任务完成后，在主线程上更新UI或执行其他与UI相关的操作。

注意，`public class CounterTask extends AsyncTask<Integer, Integer, String>` 如果最后一个参数是 `Void` 就没有必要写这个，也没有啊办法拿返回。

```
public class MyAsyncTask extends AsyncTask<Void, Integer, S 
```

在 `AsyncTask` 中，这三个泛型参数有如下含义：

1. 第一个参数（`Params`）：这是传递给 `doInBackground` 方法的参数类型。在这个例子中，它是 `Void` 类型。`doInBackground` 方法会接收这种类型的参数，然后在后台线程上执行任务。当你调用 `execute()` 方法并传递参数时，这些参数会传递给 `doInBackground` 方法。
2. 第二个参数（`Progress`）：这是用于报告任务进度的类型。在这个例子中，它是 `Integer` 类型。当任务需要报告进度时，可以调用 `publishProgress()` 方法，并传入相应类型的参数。这将触发 `onProgressUpdate()` 方法的调用，该方法在主线程上运行，允许你更新 UI 以显示进度信息。
3. 第三个参数（`Result`）：这是 `doInBackground` 方法返回的结果类型。在这个例子中，它是 `String` 类型。当 `doInBackground` 方法执行完毕后，它会返回一个值，该值将传递给 `onPostExecute()` 方法。`onPostExecute()` 方法也在主线程上运行，允许你根据结果更新 UI。

因此，在这个例子中：

- `Void` 是传递给 `doInBackground` 方法的参数类型。
- `Integer` 是用于报告任务进度的类型。
- `String` 是 `doInBackground` 方法返回的结果类型。

第一个参数 `Void` 对应的 `protected String doInBackground(Void... voids)` 的参数

第二个参数 Integer 对应的 `protected void onProgressUpdate(Integer... values)` 和 `publishProgress(i)` 的参数类型

第三个参数 String 对应的 `protected void onPostExecute(String result)` 的参数

🔗 AsyncTask 的替代方案

在最新的Android开发中，`AsyncTask` 已经不再推荐使用。Google现在推荐使用 `Kotlin coroutines`（协程）和 `LiveData` 来进行异步处理和UI更新。如果你仍然使用Java进行Android开发，可以使用 `java.util.concurrent` 包中的类，例如 `ExecutorService` 和 `Future` 来进行异步操作。

```
package com.fu.tt;

import androidx.appcompat.app.AppCompatActivity;
import android.os.Bundle;
import android.os.Handler;
import android.os.Looper;
import android.view.View;
import android.widget.Button;
import android.widget.TextView;

import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

public class MainActivity extends AppCompatActivity {

    private TextView textView;
    private Button button;
    private ExecutorService executorService;
    private Handler handler;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        textView = findViewById(R.id.textView);
        button = findViewById(R.id.button);
    }
}
```



```

        executorService = Executors.newSingleThreadExecutor();
        handler = new Handler(Looper.getMainLooper());

        button.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View v) {
                startAsyncTask();
            }
        });
    }

    private void startAsyncTask() {
        executorService.execute(new Runnable() {
            @Override
            public void run() {
                // Simulate a long-running operation
                try {
                    Thread.sleep(2000);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }

                final String result = "Updated text from As

                handler.post(new Runnable() {
                    @Override
                    public void run() {
                        textView.setText(result);
                    }
                });
            }
        });
    }


    @Override
    protected void onDestroy() {
        super.onDestroy();
        executorService.shutdown();
    }
}

```

`ExecutorService` 提供了一种管理线程执行的机制。它是 Java `java.util.concurrent` 包中的一个接口，提供了各种方法来管理和控制线程池中的线程。`submit` 是这个接口中的一个方法，用于提交一个可执行任务（`Runnable` 或 `Callable`）到线程池中进行异步执行。

`submit` 方法的主要特点：

1. 当你提交一个任务时，`ExecutorService` 将根据线程池中的配置来决定如何执行这个任务。如果线程池中有空闲线程，它会立即分配一个线程来执行任务。如果线程池已满，它会将任务加入到等待队列中，直到有可用线程。
2. `submit` 方法可以接受一个 `Runnable` 或 `Callable` 类型的任务。`Runnable` 类型的任务没有返回值，而 `Callable` 类型的任务可以返回一个值。当你提交一个 `Callable` 类型的任务时，`submit` 方法会返回一个 `Future` 对象，你可以用这个对象来获取任务的返回值。
3. `submit` 方法允许你提交任务后立即继续执行其他操作，而不需要等待任务完成。这使得你可以在应用程序中实现并发和异步操作。



```
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.Future;

public class Main {
    public static void main(String[] args) {
        ExecutorService executorService = Executors.newFixedThreadPool(10);

        // Using execute() method
        Runnable runnableTask = () -> {
            System.out.println("Running a Runnable task using execute()");
        };
        executorService.execute(runnableTask);

        // Using submit() method
        Runnable runnableTaskWithSubmit = () -> {
            System.out.println("Running a Runnable task using submit()");
        };
        Future<?> runnableFuture = executorService.submit(runnableTaskWithSubmit);

        // Using submit() method with Callable
        Callable<String> callableTask = () -> {
            System.out.println("Running a Callable task using submit()");
            return "Callable task result";
        };
        Future<String> callableFuture = executorService.submit(callableTask);

        executorService.shutdown();
    }
}
```

`submit` 和 `execute` 在不同的场景下有各自的优势。选择哪一个取决于你的具体需求。

如果你只需要执行一个简单的 `Runnable` 任务，并且不关心任务是否完成、任务的返回值或异常，那么使用 `execute` 就足够了。它更简单，因为你不需要处理返回的 `Future` 对象。

然而，如果你需要执行一个返回值的任务（如 `Callable`），或者你需要查询任务的执行状态、处理任务中的异常，那么 `submit` 更适合这种情况，因为它返回一个 `Future` 对象，可以用来获取任务的返回值、检查任务是否完成，以及捕获任务中的异常。

总之，`submit` 和 `execute` 都有各自的应用场景。根据你的需求选择合适的方法。在一些情况下，`submit` 可能是一个更优的选择，但在其他情况下，`execute` 可能更简单、更合适。

🔗 Callable 和 Runnable

`Callable` 和 `Runnable` 都是 Java 中表示任务的接口，它们主要的区别在于是否有返回值和是否可以抛出异常：

1. 返回值：

- `Runnable` 接口的 `run` 方法没有返回值，即它的返回类型是 `void`。
- `Callable` 接口的 `call` 方法有返回值，返回类型是泛型 `V`，即 `V call()`。

2. 异常处理：

- `Runnable` 接口的 `run` 方法不能抛出已检查的异常（checked exceptions），只能处理运行时异常（runtime exceptions）。
- `Callable` 接口的 `call` 方法可以抛出已检查的异常，这使得异常处理更加灵活。

根据你的需求选择 `Runnable` 或 `Callable`。如果你的任务需要返回一个结果，或者需要抛出已检查的异常，那么你应该使用 `Callable`。如果你的任务不需要返回结果，且只需处理运行时异常，那么 `Runnable` 更为简单。

🔗 ViewModel

当 Android 设备发生配置更改，例如屏幕旋转，系统默认会重新创建当前活动（Activity）或片段（Fragment）。这可能导致当前界面控制器中的数据丢失，因为重新创建活动或片段时，成员变量的状态会被重置。

ViewModel 可以解决这个问题，它是一种特殊的类，用于在配置更改期间存储和管理 UI 相关数据。这意味着在设备旋转时，ViewModel 会保持其状态，而不会像普通 Activity 那样重置。

为了更好地理解这个概念，让我们以一个计数器应用程序为例。在应用程序中，有一个按钮，用户可以点击它来增加计数器的值。如果我们将计数器值存储在 Activity 成员变量中，那么当屏幕旋转时，计数器值将丢失。然而，如果我们将计数器值存储在 ViewModel 中，即使在屏幕旋转时，计数器值也会保持不变。

除了保持数据生存之外，ViewModel 还有助于实现解耦和更易于测试的代码。因为 ViewModel 不直接依赖于 Activity 或 Fragment，所以它可以独立于界面控制器进行测试。这使得编写单元测试变得更简单，因为你不需要模拟整个界面控制器，而只需关注 ViewModel 中的逻辑。

简而言之，ViewModel 的主要优点是：

1. 在配置更改（如设备旋转）期间保持 UI 相关数据。
2. 实现代码解耦，使得 ViewModel 可以独立于界面控制器进行测试。

Mainactivit.java



```
package com.fu.tt;

import androidx.appcompat.app.AppCompatActivity;
import androidx.lifecycle.ViewModelProvider;
import android.os.Bundle;
import android.view.View;
import android.widget.Button;
import android.widget.TextView;

public class MainActivity extends AppCompatActivity {
    private CounterViewModel viewModel;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        viewModel = new ViewModelProvider(this).get(Counter

        TextView counterTextView = findViewById(R.id.counte
        Button incrementButton = findViewById(R.id.incremen

        incrementButton.setOnClickListener(new View.OnClickListener()
            @Override
            public void onClick(View view) {
                viewModel.incrementCounter();
                counterTextView.setText("Counter: " + viewM
            }
        });
    }
}
```

CounterViewModel.java

```
package com.fu.tt;
```



```
import androidx.lifecycle.ViewModel;
```

```
public class CounterViewModel extends ViewModel {  
    private int counter;  
  
    public CounterViewModel() {  
        counter = 0;  
    }  
  
    public int getCounter() {  
        return counter;  
    }  
  
    public void incrementCounter() {  
        counter++;  
    }  
}
```

Activity_main.xml

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical"
    android:gravity="center"
    android:padding="16dp">

    <TextView
        android:id="@+id/counterTextView"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Counter: 0" />

    <Button
        android:id="@+id/incrementButton"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_marginTop="16dp"
        android:text="Increment Counter" />

</LinearLayout>
```



🔗 两个build.gradle

在 Android 项目中，通常有两个 `build.gradle` 文件，分别为：

1. 项目级别的 `build.gradle` (Project-level build.gradle)：位于项目根目录下。此文件主要用于配置整个项目的构建设置，包括 Gradle 插件版本、gradle 版本等。这个文件还可以定义全局变量，例如通用的依赖库版本，这样在模块级别的 `build.gradle` 文件中就可以使用它们。
2. 模块级别的 `build.gradle` (Module-level build.gradle)：位于每个模块（如 app 模块）的根目录下。此文件主要用于配置模块相关的构建设置，包括应用 ID、编译 SDK 版本、目标 SDK 版本、依赖库等。每个模块都有一个单独的 `build.gradle` 文件，允许您为每个模块单独配置构建设置。

要区分这两个文件，您可以查看它们的路径和内容。项目级别的 `build.gradle` 位于项目根目录下，而模块级别的 `build.gradle` 位于模块目录下。另外，它们的内容也有所不同。项目级别的文件通常包含 `buildscript` 和 `allprojects` 块，而模块级别的文件包含 `android` 和 `dependencies` 块。

这里是一个简单的 **ViewModel** 示例，演示如何使用 ViewModel 存储和管理计数器数据。

1. 在模块 (app) 的 `build.gradle` 文件中添加 ViewModel 依赖：

```
dependencies {  
    // ...  
    implementation 'androidx.lifecycle:lifecycle-viewmodel:  
}
```

2. 创建一个名为 `CounterViewModel` 的 ViewModel 类，该类将存储计数器值并提供用于递增计数器的方法：

```
import androidx.lifecycle.ViewModel;  
  
public class CounterViewModel extends ViewModel {  
    private int counter;  
  
    public CounterViewModel() {  
        counter = 0;  
    }  
  
    public int getCounter() {  
        return counter;  
    }  
  
    public void incrementCounter() {  
        counter++;  
    }  
}
```

2. 在 `MainActivity` 类中，使用 `ViewModelProvider` 创建

CounterViewModel 的实例，并在按钮点击时递增计数器值：

```
import androidx.appcompat.app.AppCompatActivity;
import androidx.lifecycle.ViewModelProvider;
import android.os.Bundle;
import android.view.View;
import android.widget.Button;
import android.widget.TextView;

public class MainActivity extends AppCompatActivity {
    private CounterViewModel viewModel;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        viewModel = new ViewModelProvider(this).get(Counter

        TextView counterTextView = findViewById(R.id.counte
        Button incrementButton = findViewById(R.id.incremen

        incrementButton.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View view) {
                viewModel.incrementCounter();
                counterTextView.setText("Counter: " + viewM
            }
        });
    }
}
```

4. 在 activity_main.xml 布局文件中，添加一个 TextView 用于显示计数器值，以及一个 Button 用于递增计数器：

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk
    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"
    android:orientation="vertical"
    android:padding="16dp"
    tools:context=".MainActivity">

    <TextView
        android:id="@+id/counterTextView"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Counter: 0" />

    <Button
        android:id="@+id/incrementButton"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_marginTop="16dp"
        android:text="Increment Counter" />

</LinearLayout>
```

🔗 LiveData

(简而言之就是，一个能够在 activity 之间，Fragment 之间，ViewModel 之间，或者他们相互之间，实时更新，实时共享的数据，类似于多线程的，共享数据)

LiveData 是 Android Jetpack 构件库中的一个核心组件，它是一个可观察的数据持有者类，可用于在应用程序的不同组件（如 Activity、Fragment 或 ViewModel）之间共享和观察数据。LiveData 遵循观察者模式，使 UI 组件（如 Activity 或 Fragment）能够在数据更改时自动更新。这有助于实现更简洁、可维护和可测试的代码。

1. 确保 UI 与数据状态一致：因为 LiveData 遵循观察者模式，所以当数据更

改时，所有订阅的 UI 组件都会自动更新。

2. **生命周期感知**：LiveData 是生命周期感知的，这意味着它会自动管理观察者的订阅，确保只在活跃的生命周期（如 Activity 处于 started 或 resumed 状态）时更新 UI。这样可以避免在不活跃的生命周期（如 Activity 处于 paused 或 stopped 状态）中进行不必要的 UI 更新，从而提高性能。
3. **无内存泄漏**：由于 LiveData 会自动管理订阅，所以在组件（如 Activity 或 Fragment）销毁时不会造成内存泄漏。
4. **数据共享**：LiveData 可以在不同组件之间轻松共享数据，特别是与 ViewModel 一起使用时，可以实现 UI 逻辑与数据处理逻辑的解耦。

可以了解到 **LiveData** 的最主要的作用是，共享数据，并且是共享实时数据，但是我们在前面的学习当中就已经学习了，通过多线程的数据共享的方式进行共享数据，为什么还需要这个东西？

线程之间的数据共享是指多个线程同时访问和操作相同的数据。这种情况下，需要注意线程同步和互斥的问题，以避免数据不一致、数据竞争和其他潜在问题。线程之间的数据共享主要关注如何在多线程环境下正确处理数据。

而 LiveData 提供的数据共享是指在应用程序的不同组件（如 Activity、Fragment 或 ViewModel）之间共享和观察数据。LiveData 的数据共享关注的是组件间的数据传递，以及在数据变化时自动更新 UI。LiveData 的优势在于生命周期感知、自动管理观察者订阅、避免内存泄漏等。

LiveData 和线程间的数据共享有以下区别：

1. **目的**：线程间的数据共享关注的是多线程环境下数据的正确处理，而 LiveData 关注的是组件间的数据传递和自动更新 UI。
2. **生命周期感知**：LiveData 是生命周期感知的，这意味着它会根据组件的生命周期自动管理观察者订阅，从而避免在不活跃的生命周期中进行不必要的 UI 更新。线程间的数据共享不涉及生命周期感知。
3. **线程安全**：LiveData 保证了数据的线程安全，因为它内部使用了线程安全的数据结构（如 AtomicReference）和同步机制。线程间的数据共享需要手动处理同步和互斥问题，以确保数据的正确性。

总之，LiveData 的存在意义在于提供一种生命周期感知、线程安全且易于管理的数据共享方式，使组件间的数据传递更加简洁、可维护和可测试。而线程间的数据共享主要关注在多线程环境下如何正确处理数据。在实际应用中，可以根据需求和场景选择合适的方式进行数据共享。

CounterViewModel.java

```
package com.fu.tt;

import androidx.lifecycle.LiveData;
import androidx.lifecycle.MutableLiveData;
import androidx.lifecycle.ViewModel;

public class CounterViewModel extends ViewModel {
    private MutableLiveData<Integer> counter = new MutableLiveData<>();

    public CounterViewModel() {
        counter.setValue(0);
    }

    public LiveData<Integer> getCounter() {
        return counter;
    }

    public void incrementCounter() {
        counter.setValue(counter.getValue() + 1);
    }
}
```

`MutableLiveData` 是一个类，它可以包含或持有某种类型的值，这种类型就是尖括号里的 `Integer`。在这个例子中，`counter` 是一个可以持有 `Integer` 类型值的

`MutableLiveData` 实例。我们可以通过 `counter.setValue()` 来设置这个值，通过 `counter.getValue()` 来获取这个值。

MainActivity.java

```
package com.fu.tt;

import androidx.appcompat.app.AppCompatActivity;
import androidx.lifecycle.Observer;
import androidx.lifecycle.ViewModelProvider;
import android.os.Bundle;
import android.view.View;
import android.widget.Button;
import android.widget.TextView;

public class MainActivity extends AppCompatActivity {
    private CounterViewModel viewModel;
    private TextView counterTextView;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        counterTextView = findViewById(R.id.counterTextView)
        viewModel = new ViewModelProvider(this).get(Counter

        // 因为 viewModel.getCounter() 的这个函数，方法被设计为延迟
        // 因此它允许其他类（如你的 Activity 或 Fragment）观察并监听
        viewModel.getCounter().observe(this, new Observer<Integer>() {
            // 当观察的数据发生更改的时候，执行
            @Override
            public void onChanged(Integer counterValue) {
                counterTextView.setText("Counter: " + counterValue);
            }
        });

        Button incrementButton = findViewById(R.id.incrementButton)
        incrementButton.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View v) {
                viewModel.incrementCounter();
            }
        });
    }
}
```

多个 LiveData 的情况

ProductViewModel.java

```
package com.fu.tt;

import androidx.lifecycle.LiveData;
import androidx.lifecycle.MutableLiveData;
import androidx.lifecycle.ViewModel;

public class ProductViewModel extends ViewModel {
    private MutableLiveData<String> productName = new MutableLiveData<>();
    private MutableLiveData<Double> productPrice = new MutableLiveData<>();

    public ProductViewModel() {
        productName.setValue("Default Product");
        productPrice.setValue(0.0);
    }

    public LiveData<String> getProductName() {
        return productName;
    }

    public LiveData<Double> getProductPrice() {
        return productPrice;
    }

    public void changeProductName(String newName) {
        productName.setValue(newName);
    }

    public void changeProductPrice(Double newPrice) {
        productPrice.setValue(newPrice);
    }
}
```



```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical"
    android:gravity="center"
    android:padding="16dp">

    <TextView
        android:id="@+id/counterTextView"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Counter: 0" />

    <TextView
        android:id="@+id/productNameTextView"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Product Name: " />

    <TextView
        android:id="@+id/productPriceTextView"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Product Price: " />

    <Button
        android:id="@+id/changeNameButton"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Change Product Name" />

    <Button
        android:id="@+id/changePriceButton"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Change Product Price" />

</LinearLayout>
```


🔗 observe和observeForever

当 `LiveData` 对象的数据改变时，它会通知所有观察者。在 `LiveData` 中，你主要会看到 `observe` 和 `observeForever` 方法，用于注册观察者。

- `observe(LifecycleOwner owner, Observer<? super T> observer)`: 注册一个观察者，这个观察者会在 `LifecycleOwner`（如 `Activity` 或 `Fragment`）的生命周期在 `STARTED` 或 `RESUMED` 状态时被通知。这是确保只有在 `Activity` 或 `Fragment` 处于活跃状态时才更新 UI 的好方法。
- `observeForever(Observer<? super T> observer)`: 注册一个观察者，不受任何生命周期的限制，这个观察者会始终被通知数据变化。需要注意的是，使用 `observeForever` 需要更谨慎，因为如果你忘记了在不再需要观察者时取消注册，可能会导致内存泄漏。

🔗 observeForever

MainActivity.java



```
package com.fu.tt;

import androidx.appcompat.app.AppCompatActivity;
import androidx.lifecycle.Observer;
import androidx.lifecycle.ViewModelProvider;

import android.os.Bundle;
import android.util.Log;

public class MainActivity extends AppCompatActivity {
    private ExampleViewModel viewModel;
    private Observer<Integer> observer;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        viewModel = new ViewModelProvider(this).get(Example

        observer = new Observer<Integer>() {
            @Override
            public void onChanged(Integer counterValue) {
                Log.i("Example", "Counter: " + counterValue
            }
        };

        // observeForever
        viewModel.getCounter().observeForever(observer);

        viewModel.incrementCounter(); // This will trigger
    }

    @Override
    protected void onDestroy() {
        super.onDestroy();
        // 销毁 LiveData
        viewModel.getCounter().removeObserver(observer);
    }
}
```

ExampleViewModel.java

```
import androidx.lifecycle.LiveData;
import androidx.lifecycle.MutableLiveData;
import androidx.lifecycle.ViewModel;

public class ExampleViewModel extends ViewModel {
    private MutableLiveData<Integer> counter = new MutableLiveData<>();

    public ExampleViewModel() {
        counter.setValue(0);
    }

    public LiveData<Integer> getCounter() {
        return counter;
    }

    public void incrementCounter() {
        counter.setValue(counter.getValue() + 1);
    }
}
```

viewModel.getCounter().observeForever(observer); 为什么不是写成
viewModel.getCounter().observeForever(this, observer);

observeForever() 方法只需要一个参数：Observer。它不需要 LifecycleOwner 参数，因为这个方法被设计为在没有 LifecycleOwner 的情况下使用。这就是为什么你看到的调用形式是 observeForever(observer) 而不是 observeForever(this, observer)。

这与 observe() 方法不同，observe() 方法需要一个 LifecycleOwner 参数，例如 this（在 Activity 或 Fragment 中），以及一个 Observer。这允许 LiveData 知道何时在 LifecycleOwner 的生命周期状态改变时开始或停止向 Observer 发送更新。

所以，如果你使用 observeForever()，那么你必须在适当的时候手动调用 removeObserver() 以防止内存泄漏，例如在 Activity 或 Fragment 的 onDestroy() 方法中。这是因为 observeForever() 不会自动处理生命周期，因此 LiveData 不会知道何时停止向 Observer 发送更新。

WorkManager

它确保了即使应用退出或设备重启，任务也能得到执行。WorkManager 适用于那些对立即性要求不高，但必须确保执行的任务。例如，上传日志文件，同步应用数据等。

使用 WorkManager 执行后台任务主要包括以下步骤：

1. 创建一个 Worker 类：Worker 类是实际执行后台任务的地方。你需要继承 Worker 类，然后重写 doWork() 方法。
2. 创建 WorkRequest：WorkRequest 代表一个待执行的工作。
WorkManager 提供了两种类型的 WorkRequest：OneTimeWorkRequest（一次性任务）和 PeriodicWorkRequest（周期性任务）。
3. 将 WorkRequest 提交给 WorkManager：通过调用 WorkManager 的 enqueue() 方法，你可以将 WorkRequest 添加到 WorkManager 的执行队列中。

LoggingWorker.java



```
package com.fu.tt;

import android.content.Context;
import androidx.work.Worker;
import androidx.work.WorkerParameters;
import android.util.Log;

public class LoggingWorker extends Worker {
    public LoggingWorker(Context context, WorkerParameters
        super(context, params);
    }

    // 一般来说只需要重写 doWork
    @Override
    public Result doWork() {
        Log.d("LoggingWorker", "This is a message from Logg
        // 一定要返回的内容表示已经成功执行
        return Result.success();
    }
}
```

MainActivity.java



```
package com.fu.tt;

import androidx.appcompat.app.AppCompatActivity;
import androidx.work.OneTimeWorkRequest;
import androidx.work.PeriodicWorkRequest;
import androidx.work.WorkManager;

import android.os.Bundle;

import java.util.concurrent.TimeUnit;

public class MainActivity extends AppCompatActivity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        OneTimeWorkRequest loggingWorkRequest = new OneTime
            WorkManager.getInstance(this).enqueue(loggingWorkRe
        }
    }
}
```

WorkManager 的任务模式有两种

OneTimeWorkRequest 是一次性的任务，当任务被执行后，就不会再次执行。

PeriodicWorkRequest 是周期性的任务，任务会按照预设的间隔时间反复执行。（最小的执行周期是 15min）



```
import androidx.appcompat.app.AppCompatActivity;
import androidx.work.PeriodicWorkRequest;
import androidx.work.WorkManager;

import android.os.Bundle;

import java.util.concurrent.TimeUnit;

public class MainActivity extends AppCompatActivity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        PeriodicWorkRequest loggingWorkRequest = new PeriodicWorkRequest.Builder();

        WorkManager.getInstance(this).enqueue(loggingWorkRe
    }
}
```

WorkManager_链式任务

你可以设置一系列的任务，按照一定的顺序依次执行。例如，你可以先执行一个下载任务，然后再执行一个解析任务，最后再执行一个更新 UI 的任务

FirstWorker.java

```
package com.fu.tt;
```



```
// FirstWorker.java
```

```
import android.content.Context;
```

```
import android.util.Log;
```

```
import androidx.work.Worker;
```

```
import androidx.work.WorkerParameters;
```

```
public class FirstWorker extends Worker {
```

```
    public FirstWorker(Context context, WorkerParameters wo
```

```
        super(context, workerParams);
```

```
    }
```

```
    @Override
```

```
    public Result doWork() {
```

```
        // 执行任务...
```

```
        Log.i("tag", "First task is done.");
```

```
        return Result.success();
```

```
    }
```

```
}
```

SecondWorker.java


```
package com.fu.tt;
```



```
// SecondWorker.java
```

```
import android.content.Context;
```

```
import android.util.Log;
```

```
import androidx.work.Worker;
```

```
import androidx.work.WorkerParameters;
```

```
public class SecondWorker extends Worker {
```

```
    public SecondWorker(Context context, WorkerParameters w
```

```
        super(context, workerParams);
```

```
    }
```

```
    @Override
```

```
    public Result doWork() {
```

```
        // 执行任务...
```

```
        Log.i("tag", "Second task is done.");
```

```
        return Result.success();
```

```
    }
```

```
}
```

MainActivity.java



```
package com.fu.tt;

import androidx.appcompat.app.AppCompatActivity;
import androidx.work.OneTimeWorkRequest;
import androidx.work.WorkManager;

import android.os.Bundle;


public class MainActivity extends AppCompatActivity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        OneTimeWorkRequest firstWorkRequest = new OneTimeWo
        OneTimeWorkRequest secondWorkRequest = new OneTimeW

        WorkManager.getInstance(this)
            .beginWith(firstWorkRequest)
            .then(secondWorkRequest)
            .enqueue();
    }
}
```

如果有多个任务，就一直 then



```
OneTimeWorkRequest downloadWorkRequest = new OneTimeWorkReq
OneTimeWorkRequest compressWorkRequest = new OneTimeWorkReq
OneTimeWorkRequest processWorkRequest = new OneTimeWorkRequ

WorkManager.getInstance(this)
    .beginWith(downloadWorkRequest)
    .then(compressWorkRequest)
    .then(processWorkRequest)
    .enqueue();
```

就是开多条工作任务链

```
package com.fu.tt;

// FirstWorker.java
import android.content.Context;
import android.util.Log;

import androidx.work.Worker;
import androidx.work.WorkerParameters;

public class FirstWorker extends Worker {
    public FirstWorker(Context context, WorkerParameters wo
        super(context, workerParams);
    }

    @Override
    public Result doWork() {
        // 执行任务...
        Log.i("tag", "First task is done.");
        return Result.success();
    }
}
```





```
package com.fu.tt;

// SecondWorker.java
import android.content.Context;
import android.util.Log;

import androidx.work.Worker;
import androidx.work.WorkerParameters;

public class SecondWorker extends Worker {
    public SecondWorker(Context context, WorkerParameters w
        super(context, workerParams);
    }

    @Override
    public Result doWork() {
        // 执行任务...
        Log.i("tag", "Second task is done.");
        return Result.success();
    }
}
```

```
package com.fu.tt;
```



```
// ThirdWorker.java
```

```
import android.content.Context;
```

```
import android.util.Log;
```

```
import androidx.work.Worker;
```

```
import androidx.work.WorkerParameters;
```

```
public class ThirdWorker extends Worker {
```

```
    public ThirdWorker(Context context, WorkerParameters wo
```

```
        super(context, workerParams);
```

```
    }
```

```
    @Override
```

```
    public Result doWork() {
```

```
        // 执行任务...
```

```
        Log.i("tag", "Third task is done.");
```

```
        return Result.success();
```

```
    }
```

```
}
```

MainActivity.java



```
package com.fu.tt;

import androidx.appcompat.app.AppCompatActivity;
import androidx.work.OneTimeWorkRequest;
import androidx.work.WorkContinuation;
import androidx.work.WorkManager;

import android.os.Bundle;

import java.util.Arrays;

public class MainActivity extends AppCompatActivity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        // 新建三个任务
        OneTimeWorkRequest firstWorkRequest = new OneTimeWo
        OneTimeWorkRequest secondWorkRequest = new OneTimeW
        OneTimeWorkRequest thirdWorkRequest = new OneTimeWo

        // 开始两个任务链
        WorkContinuation chain1 = WorkManager.getInstance(t
        WorkContinuation chain2 = WorkManager.getInstance(t

        WorkContinuation combinedChain = WorkContinuation
            .combine(Arrays.asList(chain1, chain2))
            .then(thirdWorkRequest);

        combinedChain.enqueue();
    }
}
```

翻译

combine: 合并

[🔗](#) WorkManager_约束条件

你可以为你的工作请求设置一些约束条件，只有当这些条件满足时，你的工作请求才会被执行。这些约束条件可以是网络连接状态，设备充电状态，设备空闲状态等。

以下是一个例子，展示了如何设置约束条件，以便你的工作只在设备处于充电状态且网络连接可用时执行：

MyWorker.java

```
package com.fu.tt;

import android.content.Context;
import androidx.work.Worker;
import androidx.work.WorkerParameters;
import android.util.Log;

public class MyWorker extends Worker {
    public MyWorker(Context context, WorkerParameters param
        super(context, params);
    }

    @Override
    public Result doWork() {
        Log.i("MyWorker", "Work is done!ok");
        return Result.success();
    }
}
```

```
package com.fu.tt;

import androidx.appcompat.app.AppCompatActivity;
import androidx.work.Constraints;
import androidx.work.NetworkType;
import androidx.work.OneTimeWorkRequest;
import androidx.work.WorkContinuation;
import androidx.work.WorkManager;

import android.os.Bundle;
```

```
import android.view.View;
import android.widget.Button;

import java.util.Arrays;

public class MainActivity extends AppCompatActivity {
    private Button startWorkButton;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        startWorkButton = findViewById(R.id.startWorkButton)
        startWorkButton.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View v) {
                startWork();
            }
        });
    }

    private void startWork() {
        Constraints constraints = new Constraints.Builder()
            .setRequiredNetworkType(NetworkType.CONNECT)
            .build();

        OneTimeWorkRequest workRequest = new OneTimeWorkReq
            .setConstraints(constraints)
            .build();

        WorkManager.getInstance(this).enqueue(workRequest);
    }
}
```



```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical"
    android:gravity="center"
    android:padding="16dp">

    <Button
        android:id="@+id/startWorkButton"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Start Work" />

</LinearLayout>
```



WorkManager 提供了一系列的约束条件，你可以根据需要选择使用。以下是你可以设置的所有约束条件：

1. **NetworkType（网络类型）**：你可以指定工作需要的网络类型。例如，你可以指定工作只在 Wi-Fi 连接时执行，或者只在有任何类型的网络连接（包括蜂窝数据）时执行。可用的值有 `NetworkType.NOT_REQUIRED`、`NetworkType.CONNECTED`、`NetworkType.UNMETERED`、`NetworkType.NOT_ROAMING` 和 `NetworkType.METERED`。
2. **BatteryNotLow（电池电量不低）**：你可以指定只有当设备电池电量充足时，工作才会执行。如果设置为 `true`，那么工作只会在设备电池电量不低时执行。
3. **RequiresCharging（需要充电）**：你可以指定只有当设备处于充电状态时，工作才会执行。如果设置为 `true`，那么工作只会在设备正在充电时执行。
4. **StorageNotLow（存储空间不低）**：你可以指定只有当设备的存储空间充足时，工作才会执行。如果设置为 `true`，那么工作只会在设备的存储空间不低时执行。
5. **RequiresDeviceIdle（需要设备空闲）**：你可以指定只有当设备处于空闲状态时，工作才会执行。这通常用于那些不需要用户交互，并且可以在设备空闲时执行的工作。如果设置为 `true`，那么工作只会在设备处于空闲

状态时执行。

```
Constraints constraints = new Constraints.Builder()  
    .setRequiredNetworkType(NetworkType.UNMETERED) // 只在有线网络  
    .setRequiresCharging(true) // 只在充电  
    .setRequiresBatteryNotLow(true) // 只在电量充足  
    .setRequiresStorageNotLow(true) // 只在存储空间充足  
    .setRequiresDeviceIdle(true) // 只在设备空闲  
    .build();
```



并且需要注意的是

当你在构建约束条件时，使用的逻辑是 "AND" 逻辑，也就是说，所有的条件都需要满足，工作才会被执行。所以在你提供的例子中，网络必须是无线网络，设备必须正在充电，电池电量不能低，存储空间不能低，设备必须空闲，所有这些条件都满足时，任务才会执行。

WorkManager_输入输出数据



```
package com.fu.tt;

import androidx.appcompat.app.AppCompatActivity;
import androidx.work.OneTimeWorkRequest;
import androidx.work.WorkManager;
import android.os.Bundle;
import androidx.lifecycle.Observer;
import androidx.work.Data;
import androidx.work.WorkInfo;
import android.util.Log;

public class MainActivity extends AppCompatActivity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        OneTimeWorkRequest incrementWorkRequest = new OneTimeWorkRequest.Builder()
            .setInputData(new Data.Builder().putInt("increment", 1)
            .build());

        WorkManager.getInstance(this).enqueue(incrementWorkRequest);

        WorkManager.getInstance(this).getWorkInfoByIdLiveData("increment")
            .observe(this, new Observer<WorkInfo>() {
                @Override
                public void onChanged(WorkInfo workInfo) {
                    if (workInfo != null && workInfo.getState() == WorkInfo.State.SUCCEEDED) {
                        int output = workInfo.getOutput().getInt("output");
                        Log.d("MainActivity", "Received output: " + output);
                    }
                }
            });
    }
}
```

```
package com.fu.tt;
```



```
import android.content.Context;
import android.util.Log;
```

```
import androidx.annotation.NonNull;
import androidx.work.Data;
import androidx.work.Worker;
import androidx.work.WorkerParameters;
```

```
public class IncrementWorker extends Worker {
    public static final String KEY_INPUT = "input";
    public static final String KEY_OUTPUT = "output";

    public IncrementWorker(@NonNull Context context, @NonNu
        super(context, workerParams);
    }

    @NonNull
    @Override
    public Result doWork() {
        int input = getInputData().getInt(KEY_INPUT, 0);
        int output = input + 1;

        Data outputData = new Data.Builder()
            .putInt(KEY_OUTPUT, output)
            .build();

        Log.d("IncrementWorker", "Incremented value: " + ou
        return Result.success(outputData); // 并且输出内容
    }
}
```

[🔗](#) WorkManager_Result.failure



```
package com.fu.tt;

import android.content.Context;
import android.util.Log;

import androidx.annotation.NonNull;
import androidx.work.Data;
import androidx.work.Worker;
import androidx.work.WorkerParameters;

public class FailingWorker extends Worker {
    public static final String KEY_ERROR_MESSAGE = "error_m

    public FailingWorker(@NonNull Context context, @NonNull
        super(context, workerParams);
    }

    @NonNull
    @Override
    public Result doWork() {
        try{
            return Result.success();
        }catch (Exception e)
        {
            Data failureData = new Data.Builder()
                .putString(KEY_ERROR_MESSAGE, "Somethin
                .build();

            Log.d("FailingWorker", "Failing as expected");
            return Result.failure(failureData);
        }
    }
}
```

```

package com.fu.tt;

import androidx.appcompat.app.AppCompatActivity;
import androidx.lifecycle.Observer;
import androidx.work.OneTimeWorkRequest;
import androidx.work.WorkInfo;
import androidx.work.WorkManager;

import android.os.Bundle;
import android.util.Log;

public class MainActivity extends AppCompatActivity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        OneTimeWorkRequest failingWorkRequest = new OneTime
            .build();

        WorkManager.getInstance(this).enqueue(failingWorkRe

        WorkManager.getInstance(this).getWorkInfoByIdLiveDa
            .observe(this, new Observer<WorkInfo>() {
                @Override
                public void onChanged(WorkInfo workInfo) {
                    if (workInfo != null && workInfo.ge
                        String errorMessage = workInfo.
                        Log.d("MainActivity", "FailingW
                    }
                }
            }));
    }
}

```

🔗 [WorkManager_Result.retry\(\)](#)

当你的工作项可能因暂时的问题（例如网络连接问题）而失败时，你可能希望稍后再次尝试执行该工作项，而不是立即将其标记为失败。在这种情况下，你可以在 `doWork()` 方法中返回 `Result.retry()`，来告诉 `WorkManager` 你希望再次尝试执行这项工作

RetryWorker.java

```
import android.content.Context;
import androidx.annotation.NonNull;
import androidx.work.Worker;
import androidx.work.WorkerParameters;

public class RetryWorker extends Worker {

    public RetryWorker(@NonNull Context context, @NonNull WorkerParameters workerParams) {
        super(context, workerParams);
    }

    @NonNull
    @Override
    public Result doWork() {
        // 模拟一项需要网络连接的任务，但网络当前不可用
        if (checkNetworkConnection()) {
            // 网络连接正常，执行任务...
            return Result.success();
        } else {
            // 网络连接不可用，稍后重试
            return Result.retry();
        }
    }

    private boolean checkNetworkConnection() {
        // 这里只是一个示例，实际上你需要检查实际的网络连接状态
        return false;
    }
}
```

MainActivity.java



```
import androidx.appcompat.app.AppCompatActivity;
import androidx.lifecycle.Observer;
import androidx.work.OneTimeWorkRequest;
import androidx.work.WorkInfo;
import androidx.work.WorkManager;

import android.os.Bundle;
import android.util.Log;

public class MainActivity extends AppCompatActivity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        OneTimeWorkRequest retryWorkRequest = new OneTimeWo
            .build();

        WorkManager.getInstance(this).enqueue(retryWorkRequ

        WorkManager.getInstance(this).getWorkInfoByIdLiveDa
            .observe(this, new Observer<WorkInfo>() {
                @Override
                public void onChanged(WorkInfo workInfo) {
                    if (workInfo != null) {
                        Log.d("MainActivity", "Work state:
                    }
                }
            });
    }
}
```

🔗 `WorkManager.Result.failure()`



```
package com.fu.tt;

import android.content.Context;

import androidx.annotation.NonNull;
import androidx.work.Worker;
import androidx.work.WorkerParameters;

import android.util.Log;

public class FailureWorker extends Worker {
    public FailureWorker(@NonNull Context context, @NonNull
        super(context, workerParams);
    }

    @NonNull
    @Override
    public Result doWork() {
        Log.d("FailureWorker", "Doing work...");
        return Result.failure();
    }
}
```



```
package com.fu.tt;

import androidx.appcompat.app.AppCompatActivity;
import androidx.lifecycle.Observer;
import androidx.work.OneTimeWorkRequest;
import androidx.work.WorkInfo;
import androidx.work.WorkManager;

import android.os.Bundle;
import android.util.Log;

public class MainActivity extends AppCompatActivity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        OneTimeWorkRequest failureWorkRequest = new OneTime
            .build();

        // 将任务进行排队
        WorkManager.getInstance(this).enqueue(failureWorkRe

        // getWorkInfoByIdLiveData 只会返回一个包含工作状态的 Work
        WorkManager.getInstance(this).getWorkInfoByIdLiveDa
            .observe(this, new Observer<WorkInfo>() {
                @Override
                public void onChanged(WorkInfo workInfo)
                    if (workInfo != null) {
                        Log.d("FailureWorker", "Work st
                    }
                }
            }));
    }
}
```

在这个示例中，当 `FailureWorker` 的 `doWork()` 方法执行时，它将返回 `Result.failure()`，标识工作执行失败。在 `MainActivity` 中，我们观察这个工作的状态，当 `FailureWorker` 执行完毕，状态将会变为 `FAILED`，并在日志中输出这个状态。

他是如何做到监听的

在 `WorkManager` 中，每个 `WorkRequest`（无论是 `OneTimeWorkRequest` 还是 `PeriodicWorkRequest`）都有一个唯一的 ID，这个 ID 在创建 `WorkRequest` 时自动生成。当你将 `WorkRequest` 提交给 `WorkManager` 进行排队时，`WorkManager` 就会在内部数据库中为这个 `WorkRequest` 创建一个条目，并将其状态设置为 `ENQUEUED`。

每当 `WorkRequest` 的状态发生改变时，例如当 `WorkManager` 开始执行它时，或者当它完成、失败或被取消时，`WorkManager` 都会更新这个内部数据库条目的状态。这就是 `WorkManager` 是如何跟踪每个 `WorkRequest` 状态的方式。

你可以通过调用 `WorkManager` 的 `getWorkInfoById()` 或 `getWorkInfoByIdLiveData()` 方法，传入 `WorkRequest` 的 ID，来获取该 `WorkRequest` 的 `WorkInfo` 对象。`WorkInfo` 对象包含了 `WorkRequest` 的当前状态和其他信息。在你的代码中，你可以观察这个 `WorkInfo` 对象，以便在 `WorkRequest` 的状态发生改变时得到通知。

`getWorkInfoByIdLiveData()` 方法返回的是一个 `LiveData<WorkInfo>` 对象。`LiveData` 是一个在 `Android Architecture Components` 中引入的类，它遵循观察者模式，允许你添加一个观察者来监听数据的改变。当你观察 `WorkInfo` 的 `LiveData` 时，每当 `WorkRequest` 的状态在 `WorkManager` 的内部数据库中发生改变时，你的观察者就会收到通知。

这就是你如何能够监听 `WorkRequest` 的运行状态，包括它是否运行失败。

如何理解

`WorkManager.getInstance(this).getWorkInfoByIdLiveData(failureWorkRequest.getId())`

首先，`WorkManager` 是一个类，你可以把它想象成一个负责管理工作任务的“经理”。这个经理只有一个，所以我们称之为单例（Singleton）。这就是我们需要调用 `getInstance()` 的原因，因为我们希望获取这个唯一的“经理”。

然后，括号里的 `this` 是一个指向当前环境或当前活动的引用。在这个案例中，`this` 指的就是你当前的活动或应用。这个 `this` 是 `WorkManager` 所需要的，因为它需要了解当前的环境，以便能够正确地调度和执行工作任务。

接下来，`enqueue(failureWorkRequest)` 这个操作就像是告诉“经理”：“嘿，我有一个任务需要你去处理。”你把 `failureWorkRequest` 这个任务交给了“经理”，然后“经理”就会把它放到待办任务的列表里。

总的来

说，`WorkManager.getInstance(this).enqueue(failureWorkRequest)` 这行代码的意思就是：找到管理工作的“经理”，并给他一个任务去做。

在这个例子中，当 `WorkInfo` 的 `LiveData` 对象的数据发生变化时，就会调用这个 `onChanged(WorkInfo workInfo)` 方法，并把最新的 `WorkInfo` 数据作为参数传入。这就是如何实现监听 `WorkInfo` 的 `LiveData`

为什么要设计成单例模式？

在许多情况下，一个类确实可以生成很多个实例。但有时候，我们需要一个全局可访问的单一实例来协调系统中的各种操作，这就是单例（Singleton）设计模式的应用场景。

在Android中，`WorkManager` 就被设计成单例，这样不论在应用的任何地方，我们都可以通过 `WorkManager.getInstance()` 获取到同一个 `WorkManager` 实例，来进行工作的调度和管理。

这种设计主要有以下优点：

1. 资源共享：所有的地方都使用同一个 `WorkManager` 实例，意味着他们都共享同一份资源（比如内存、磁盘等），不会出现资源的浪费。
2. 全局控制：`WorkManager` 能够协调和管理应用中所有的工作任务，避免了因为存在多个实例而可能出现的任务冲突或不一致的问题。

总的来说，`WorkManager` 设计成单例主要是为了提高资源利用效率和保证工作任务的全局一致性。

`getWorkInfoById()` 是一个同步方法，它会直接返回 `WorkInfo` 对象，而不是返回 `LiveData<WorkInfo>`。这意味着，你需要在一个后台线程中调用此方法，因为在 Android 中，在主线程中进行耗时操作（例如从数据库中获取数据）是不被允许的。下面是一个使用 `getWorkInfoById()` 的例子：



```
package com.fu.tt;

import androidx.appcompat.app.AppCompatActivity;
import androidx.work.OneTimeWorkRequest;
import androidx.work.WorkInfo;
import androidx.work.WorkManager;

import android.os.AsyncTask;
import android.os.Bundle;
import android.util.Log;

import java.util.UUID;
import java.util.concurrent.ExecutionException;

public class MainActivity extends AppCompatActivity {

    private OneTimeWorkRequest failureWorkRequest;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        failureWorkRequest = new OneTimeWorkRequest.Builder
            .build();

        WorkManager.getInstance(this).enqueue(failureWorkRe
            new WorkInfoAsyncTask().execute(failureWorkRequest.
    }

    private class WorkInfoAsyncTask extends AsyncTask<UUID,

        @Override
        protected WorkInfo doInBackground(UUID... uuids) {
            try {
                return WorkManager.getInstance(getApplicati
```

```

        } catch (ExecutionException | InterruptedException e) {
            e.printStackTrace();
            return null;
        }
    }

    @Override
    protected void onPostExecute(WorkInfo workInfo) {
        if (workInfo != null) {
            Log.d("MainActivity", "Work state: " + workInfo.getState())
        }
    }
}

```

🔗 JobScheduler

JobService 是一种服务，它用来执行那些可以在任何时间点运行并且不需要用户交互的工作

JobScheduler 是用于在特定的条件下执行后台任务。这些条件可能包括设备的充电状态、网络连接状态、设备闲置状态等。

"设备闲置"是指设备处于空闲状态，即用户没有使用设备的时候。在 Android 设备中，这通常指的是屏幕熄灭且设备未插入充电器的情况。当设备闲置时，系统会尽可能地减少电池消耗，例如通过降低 CPU 的速度、关闭 Wi-Fi 扫描、限制应用的后台活动等方式。

对于后台任务，这个状态是很重要的，因为在设备闲置时运行任务可能会消耗电池并打扰用户。因此，Android 提供了一种机制（如 JobScheduler 和 WorkManager），允许你指定任务在设备闲置时是否可以运行，或者在设备不再闲置时自动开始运行。这样可以帮助你的应用在尽可能减少对用户的打扰和电池消耗的同时，完成必要的后台工作。

你可以为 **JobScheduler** 设置多个约束条件，比如，你可以设置只有当设备在充电并且连接到 Wi-Fi 时，才执行某个任务。当这些条件都满足时，**JobScheduler** 会执行这个任务。

JobScheduler 和 WorkManager 都提供了在满足特定约束条件时执行任务的功能，但它们之间存在一些重要的差异。

1. 兼容性：WorkManager 是兼容所有Android版本的一种解决方案。对于那些不支持 JobScheduler API的旧版Android（API版本小于 21），WorkManager 将会使用 AlarmManager 和 BroadcastReceiver 等旧的API来进行后台任务的调度。而 JobScheduler 只能在Android 5.0（API 21）及以上版本使用。
2. 任务持久性：WorkManager 的工作请求（WorkRequest）是持久性的，这意味着即使应用程序或设备重新启动，这些工作请求也会被保存并在条件满足时继续执行。而 JobScheduler 的工作（JobInfo）则不是持久性的，如果设备重新启动，你需要重新调度它们。
3. 任务链：WorkManager 提供了一个强大的功能，允许你创建任务链和复杂的任务依赖关系。例如，你可以创建一个任务链，其中一个任务的输出是下一个任务的输入。或者，你可以创建一个任务，只有当多个其他任务全部完成时才会执行。JobScheduler 没有提供这样的功能。

综上所述，虽然 JobScheduler 和 WorkManager 在一些方面有相似之处，但 WorkManager 提供了更高级的功能和更广泛的兼容性。因此，如果你正在开发一个需要在所有Android版本上运行的应用，或者你需要使用到任务链或持久性任务等高级功能，你应该使用 WorkManager。如果你只需要在Android 5.0及以上版本上执行简单的后台任务，JobScheduler 可能会是一个更轻量级的选择。

```
package com.fu.tt;

import android.app.job.JobInfo;
import android.app.job.JobScheduler;
import android.content.BroadcastReceiver;
import android.content.ComponentName;
import android.content.Context;
import android.content.Intent;
import android.content.IntentFilter;
import android.os.Bundle;
import android.widget.TextView;
```



```

import androidx.appcompat.app.AppCompatActivity;

public class MainActivity extends AppCompatActivity {
    private TextView textView;
    private BroadcastReceiver receiver = new BroadcastRecei
        @Override
        public void onReceive(Context context, Intent inten
            String message = intent.getStringExtra("message
            textView.setText(message);
        }
};

@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);

    textView = findViewById(R.id.textview);

    IntentFilter filter = new IntentFilter(MyJobService
        registerReceiver(receiver, filter);

    ComponentName serviceName = new ComponentName(this,
        JobInfo jobInfo = new JobInfo.Builder(1149248367, s
            .setOverrideDeadline(0)
            .build();

    JobScheduler jobScheduler = (JobScheduler) getSystem
        jobScheduler.schedule(jobInfo);
}

@Override
protected void onDestroy() {
    super.onDestroy();
    unregisterReceiver(receiver);
}
}

```

翻译

Component: 组件



```
package com.fu.tt;

import android.annotation.SuppressLint;
import android.app.job.JobParameters;
import android.app.job.JobService;
import android.content.Intent;
import android.util.Log;

@SuppressLint("SpecifyJobSchedulerIdRange")
public class MyJobService extends JobService {
    public static final String ACTION_JOB_EXECUTED = "com.e

    @Override
    public boolean onStartJob(JobParameters params) {
        new Thread(() -> {
            try {
                Thread.sleep(5000); // 线程睡眠5秒
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            Intent intent = new Intent(ACTION_JOB_EXECUTED)
            intent.putExtra("message", "Job executed!");
            sendBroadcast(intent);
        }).start();
        return true;
    }

    @Override
    public boolean onStopJob(JobParameters params) {
        return false;
    }
}
```



```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res
    xmlns:tools="http://schemas.android.com/tools">

    <uses-permission android:name="android.permission.ACCESS
    <uses-permission android:name="android.permission.RECEI
    <uses-permission android:name="android.permission.READ_

    <uses-permission android:name="com.example.permission.M
    <uses-permission android:name="com.example.permission.M

    <permission
        android:name="com.example.permission.MY_BROADCAST_P
        android:protectionLevel="normal" />

    <uses-permission android:name="com.example.permission.M

    <uses-permission android:name="android.permission.READ_

    <application
        android:allowBackup="true"
        android:dataExtractionRules="@xml/data_extraction_r
        android:fullBackupContent="@xml/backup_rules"
        android:icon="@mipmap/ic_launcher"
        android:label="@string/app_name"
        android:roundIcon="@mipmap/ic_launcher_round"
        android:supportsRtl="true"
        android:theme="@style/Theme.TT"
        tools:targetApi="31">

        <service
            android:name=".MyJobService"
            android:permission="android.permission.BIND_JOB
            android:exported="true">
        </service>


    </application>
</manifest>
```

`ComponentName` 是一个Android类，用于明确地指定应用程序的一个特定组件，比如一个服务(Service)。

在我们的例子中，我们正在创建一个任务(Job)，我们需要告诉Android系统，我们的这个任务应该由哪个服务来处理。为了这样做，我们需要创建一个 `ComponentName`，这个 `ComponentName` 就像是我们的服务的名字牌，可以告诉Android系统：“嘿，这个任务应该交给这个服务来处理”。

```
ComponentName serviceName = new ComponentName(this, MyJobSe 
```

其实就在做这样的事情：它创建了一个 `ComponentName`，这个 `ComponentName` 包含了当前应用（由 `this` 表示）和 `MyJobService` 服务的信息。这样，当我们在创建任务时，我们可以使用这个 `ComponentName` 来明确地告诉Android系统，这个任务应该由 `MyJobService` 来处理。

```
JobInfo jobInfo = new JobInfo.Builder(1149248367, serviceName   
    .setOverrideDeadline(0)  
    .build();
```

这段代码的目的是建立一个Job信息，这个信息将告诉Android系统关于我们希望执行的Job的详细信息。

`JobInfo` 是一个包含有关您希望系统执行的任务的信息的类。这些信息包括：

- 要执行的任务的ID
- 要执行任务的服务（我们之前讨论过的 `ComponentName`）
- 任务执行的条件（比如设备是否在充电，是否有网络连接等）

一个 `JobInfo.Builder` 是一个用来构建 `JobInfo` 的辅助类，我们给这个 `Builder` 提供了任务ID和服务名称，然后使用 `setOverrideDeadline(0)` 来设置一个执行任务的最后期限。在这个例子中，我们设置了0，意味着这个任务应该立即执行。

最后，我们调用 `build()` 方法，这将返回一个 `JobInfo` 对象，我们可以将这个对象传递给 `JobScheduler` 来安排任务。

总的来说，这段代码就是在创建一个 `JobInfo` 对象，这个对象包含了我们希望执行的任务的所有信息，然后我们可以将这个对象传递给 `JobScheduler` 来安排任务。

****这里还可以添加其他的限制条件 ****

`JobInfo.Builder` 提供了一系列的方法，用于设置任务的执行条件。以下是一些常用的设置条件的方法：

1. `setRequiredNetworkType(int networkType)`：设置任务执行需要的网络类型。例如，`JobInfo.NETWORK_TYPE_UNMETERED` 表示任务在 Wi-Fi 连接下执行。
2. `setRequiresCharging(boolean requiresCharging)`：设置设备是否需要在充电时才执行任务。
3. `setRequiresDeviceIdle(boolean requiresDeviceIdle)`：设置设备是否需要在空闲状态下才执行任务。
4. `setPersisted(boolean isPersisted)`：设置设备重启后，任务是否继续。
5. `setRequiresBatteryNotLow(boolean requiresBatteryNotLow)`：设置设备的电量是否需要不低于某个阈值才执行任务。
6. `setRequiresStorageNotLow(boolean requiresStorageNotLow)`：设置设备的存储空间是否需要不低于某个阈值才执行任务。
7. `setPeriodic(long intervalMillis)`：设置周期性执行的任务。
8. `setOverrideDeadline(long maxExecutionDelayMillis)`：设置任务的最大延迟执行时间。
9. `setMinimumLatency(long minLatencyMillis)`：设置任务的最小延迟执行时间。

`JobService` 是 Android 中用于执行后台任务的服务。在 Android 5.0 及更高版本中引入。`JobService` 提供了两个主要方法需要重写，以实现任务的执行逻辑和停止逻辑。

1. `onStartJob(JobParameters params)`: 当系统判断任务的执行条件满足时, 会调用此方法。在这个方法中, 你需要实现你的任务逻辑。这个方法运行在主线程中, 因此如果有耗时操作, 需要开启新线程来处理。这个方法需要返回一个布尔值, 如果返回 `true`, 表示任务还在执行, 当任务执行完毕时, 需要调用 `jobFinished()` 方法来通知系统; 如果返回 `false`, 表示任务已经执行完毕。
2. `onStopJob(JobParameters params)`: 当系统判断任务需要被停止时, 会调用此方法。在这个方法中, 你需要实现任务的停止逻辑。这个方法也需要返回一个布尔值, 如果返回 `true`, 表示任务需要重新调度; 如果返回 `false`, 表示任务不需要再次执行。



```
public class MyJobService extends JobService {

    private boolean jobCancelled = false;

    @Override
    public boolean onStartJob(JobParameters params) {
        doBackgroundWork(params);
        return true; // 表示任务还在执行
    }

    private void doBackgroundWork(final JobParameters param
        new Thread(new Runnable() {
            @Override
            public void run() {
                for (int i = 0; i < 10; i++) {
                    if (jobCancelled) {
                        return;
                    }

                    Log.d("JobService", "run: " + i);

                    try {
                        Thread.sleep(1000);
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    }
                }

                Log.d("JobService", "Job finished");
                jobFinished(params, false); // 任务执行完毕, i
            }
        }).start();
    }

    @Override
    public boolean onStopJob(JobParameters params) {
        Log.d("JobService", "Job cancelled before completio
        jobCancelled = true; // 任务被取消, 更新状态
        return true; // 表示任务需要重新调度
    }
}
```

`jobFinished(JobParameters params, boolean needsReschedule)` 是 `JobService` 中的一个方法，当你的任务执行完成时，你应该调用这个方法告诉系统你的任务已经完成。这个方法接收两个参数：

1. `params`：这是你在 `onStartJob(JobParameters params)` 方法中接收到的 `JobParameters` 对象，它包含了你的任务的一些参数和配置信息。
2. `needsReschedule`：这是一个布尔值，如果你的任务由于某种原因（比如设备重启）没有完成，你希望系统在条件满足时重新调度你的任务，你可以将这个值设为 `true`。如果你的任务已经成功完成，或者你不希望任务被重新调度，你应该将这个值设为 `false`。

```
@Override
public boolean onStartJob(JobParameters params) {
    // 在这里执行你的任务...
    // 当任务完成时，调用 jobFinished() 方法
    jobFinished(params, false); // 任务完成，不需要重新调度
    return true; // 任务正在执行
}
```

在这个示例中，当 `onStartJob()` 被调用时，任务开始执行。任务执行完成后，`jobFinished()` 被调用，告诉系统任务已经完成，不需要重新调度。

🔗 IntentService 类

`IntentService` 是 Android 中一种特殊的 `Service`，它用于处理异步请求并在后台线程中执行操作。`IntentService` 是一种简化了线程管理的服务，它会在一个单独的工作线程中处理传入的 `Intent` 请求。当任务完成后，`IntentService` 会自动停止。这使得 `IntentService` 成为处理后台任务的理想选择，特别是当任务不需要与用户界面交互时。

要使用 `IntentService`，你需要执行以下步骤：

1. 创建一个继承自 `IntentService` 的类。
2. 在类中实现 `onHandleIntent(Intent intent)` 方法。
3. 通过在应用中发送 `Intent` 启动你的 `IntentService`。

下面是一个简单的 `IntentService` 示例：

```
public class MyIntentService extends IntentService {  
  
    // 一个用于处理任务的构造函数  
    public MyIntentService() {  
        super("MyIntentService");  
    }  
  
    @Override  
    protected void onHandleIntent(@Nullable Intent intent)  
        // 这里执行后台任务，这个方法运行在一个单独的工作线程中  
        // 从 Intent 中获取你需要处理的数据  
        String data = intent.getStringExtra("data");  
  
        // 执行你的任务，例如：下载文件、上传数据等  
        // ...  
  
        // 当任务完成，IntentService 将自动停止  
    }  
}
```

要启动这个 `IntentService`，你可以在你的 `Activity` 或其他组件中发送一个 `Intent`，如下所示：

```
Intent intent = new Intent(this, MyIntentService.class);  
intent.putExtra("data", "Some data to process");  
startService(intent);
```

总结一下，`IntentService` 是一个用于处理后台任务的服务，它具有简化线程管理的优势，并在任务完成后自动停止。你可以通过继承 `IntentService` 并实现 `onHandleIntent(Intent intent)` 方法来创建自己的 `IntentService`。

首先，我来解释一下什么是异步请求。在编程中，异步操作意味着你可以在不等待一个任务完成的情况下继续执行其他任务。例如，假设你有一个程序需要下载一个大文件。如果你在主线程中进行这个操作，你的程序会被阻塞，也就是说，你不能进行其他操作直到文件下载完成。但如果你在一个单独的线程（或者使用异步方式）中进行下载，你的程序可以在文件下载的同时继续进行其他操作。这就是异步的概念。

现在，我来提供一个 `IntentService` 的简单示例。假设我们创建一个 `IntentService`，它在后台获取一个网页的内容。

首先，我们创建一个继承自 `IntentService` 的类：



```
package com.fu.tt;

import android.app.IntentService;
import android.content.Intent;
import android.os.Bundle;
import android.os.ResultReceiver;
import java.util.Locale;

public class MyIntentService extends IntentService {
    public static final String PARAM_INPUT = "input";
    public static final String PARAM_RESULT_RECEIVER = "res"
    public static final int RESULT_CODE = 100;

    // 这个构造函数调用了父类IntentService的构造函数，并传递了一个线程
    public MyIntentService() {
        super("MyIntentService");
    }

    @Override
    protected void onHandleIntent(Intent intent) {
        if (intent != null) {
            String input = intent.getStringExtra(PARAM_INPU
            ResultReceiver resultReceiver = intent.getParce

            // 模拟一个耗时任务
            try {
                Thread.sleep(3000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }

            String result = String.format(Locale.getDefault

            if (resultReceiver != null) {
                Bundle bundle = new Bundle();
                bundle.putString("result", result);
                resultReceiver.send(RESULT_CODE, bundle);
            }
        }
    }
}
```



```
package com.fu.tt;

import androidx.appcompat.app.AppCompatActivity;
import android.os.Bundle;
import android.os.Handler;
import android.os.ResultReceiver;
import android.view.View;
import android.widget.Button;
import android.widget.TextView;
import android.content.Intent;

public class MainActivity extends AppCompatActivity {
    private TextView textView;
    private Button button;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        textView = findViewById(R.id.textview);
        button = findViewById(R.id.button);

        button.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View v) {
                Intent intent = new Intent(MainActivity.this);
                intent.putExtra(MyIntentService.PARAM_INPUT);
                intent.putExtra(MyIntentService.PARAM_RESULT);
                startService(intent);
            }
        });
    }

    private class MyResultReceiver extends ResultReceiver {
        MyResultReceiver(Handler handler) {
            super(handler);
        }

        @Override
        protected void onReceiveResult(int resultCode, Bundle resultData) {
            if (resultCode == MyIntentService.RESULT_CODE) {
                String result = resultData.getString("result");
                textView.setText(result);
            }
        }
    }
}
```


```
    }  
  }  
}
```

```
// 这个构造函数调用了父类IntentService的构造函数，并传递了一个线程名  
public MyIntentService() {  
    super("MyIntentService");  
}
```

给线程命名的主要原因是为了便于调试和监控。

1. **调试**：当你的应用出现问题，需要查看错误或异常时，线程的名字可以帮助你快速定位问题发生在哪个线程。特别是在多线程并发的情况下，如果线程没有具体的名字，只有默认的线程名（如Thread-0、Thread-1等），在查看日志或异常堆栈时，将很难分辨出问题发生在哪个线程。给线程命名，可以使问题定位更加清晰。
2. **性能监控**：在对程序进行性能分析（如CPU使用率、线程阻塞等）时，线程名也非常重要。性能监控工具（如Android Studio的Profiler工具）会显示各个线程的状态和活动，如果线程有具体的名字，你可以更好地理解每个线程的行为和它们如何影响应用的性能。

所以，为线程提供一个有意义的名字是一个好的编程实践，可以提高代码的可读性和可维护性。在上述的 `IntentService` 中，线程的名字被用于创建后台处理任务的工作线程。



```
@Override
protected void onHandleIntent(Intent intent) {
    if (intent != null) {
        String input = intent.getStringExtra(PARAM_INPU
        ResultReceiver resultReceiver = intent.getParce


        // 模拟一个耗时任务
        try {
            Thread.sleep(3000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

        String result = String.format(Locale.getDefault

        if (resultReceiver != null) {
            Bundle bundle = new Bundle();
            bundle.putString("result", result);
            resultReceiver.send(RESET_CODE, bundle);
        }
    }
}
```

这个方法是处理接收到的 `Intent` 的地方。当你使用 `startService(Intent)` 方法启动 `IntentService` 时，这个方法会在后台线程上被调用。这意味着你可以在这里进行耗时的操作，而不会影响到主线程。

在 `onHandleIntent` 方法内部，我们首先检查传入的 `Intent` 是否为 `null`，然后从中取出需要处理的输入字符串和 `ResultReceiver` 对象。`ResultReceiver` 是一个可以接收异步结果的类。



```
if (resultReceiver != null) {
    Bundle bundle = new Bundle();
    bundle.putString("result", result);
    resultReceiver.send(RESET_CODE, bundle);
}
```

最后，我们检查 `ResultReceiver` 是否存在，并将处理后的结果发送回去

&&&&

🔗 ResultReceiver

ResultReceiver 是 Android 提供的一个类，它可以接收跨进程的结果。这个类非常适合在需要跨进程通信时使用，比如从 Service（可能在不同的进程中）返回结果到 Activity。

通过一个简单的 Activity-to-Activity 示例来理解 ResultReceiver。在这个示例中，我们有两个 Activity：MainActivity 和 SecondActivity。MainActivity 有一个按钮和一个 TextView。点击按钮会启动 SecondActivity，SecondActivity 做一些计算然后把结果发送回 MainActivity。

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res
    xmlns:tools="http://schemas.android.com/tools">

    <uses-permission android:name="android.permission.ACCE
    <uses-permission android:name="android.permission.RECEI
    <uses-permission android:name="android.permission.READ_

    <uses-permission android:name="com.example.permission.M
    <uses-permission android:name="com.example.permission.M

    <permission
        android:name="com.example.permission.MY_BROADCAST_P
        android:protectionLevel="normal" />

    <uses-permission android:name="com.example.permission.M

    <uses-permission android:name="android.permission.READ_

    <application
        android:allowBackup="true"
        android:dataExtractionRules="@xml/data_extraction_r
        android:fullBackupContent="@xml/backup_rules"
        android:icon="@mipmap/ic_launcher"
        android:label="@string/app_name"
```

```
        android:roundIcon="@mipmap/ic_launcher_round"
        android:supportsRtl="true"
        android:theme="@style/Theme.TT"
        tools:targetApi="31">

        <activity
            android:name=".MainActivity"
            android:exported="true">
            <intent-filter>
                <action android:name="android.intent.action
                <category android:name="android.intent.cate
            </intent-filter>

            <meta-data
                android:name="android.app.lib_name"
                android:value="" />
        </activity>
        <activity android:name=".SecondActivity" />

    </application>
</manifest>
```



```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical"
    android:gravity="center"
    android:padding="16dp">

    <TextView
        android:id="@+id/textView"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Hello World!" />

    <Button
        android:id="@+id/button"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="a button" />

</LinearLayout>
```



```
package com.fu.tt;

import androidx.appcompat.app.AppCompatActivity;

import android.content.Intent;
import android.os.Bundle;
import android.os.Handler;
import android.os.ResultReceiver;
import android.view.View;
import android.widget.Button;
import android.widget.TextView;

public class MainActivity extends AppCompatActivity {
    private TextView textView;
    private Button button;

    public static final String EXTRA_RECEIVER = "com.exempl
```



```

@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);

    textView = findViewById(R.id.textView);
    button = findViewById(R.id.button);

    button.setOnClickListener(new View.OnClickListener()
        @Override
        public void onClick(View view) {
            Intent intent = new Intent(MainActivity.this);
            intent.putExtra(EXTRA_RECEIVER, new MyResultReceiver());
            startActivity(intent);
        }
    });
}

private class MyResultReceiver extends ResultReceiver {
    public MyResultReceiver(Handler handler) {
        super(handler);
    }

    @Override
    protected void onReceiveResult(int resultCode, Bundle resultData) {
        if (resultCode == 200 && resultData != null) {
            final String result = resultData.getString("result");
            runOnUiThread(new Runnable() {
                @Override
                public void run() {
                    textView.setText(result);
                }
            });
        }
    }
}
}
}
}

```

```

package com.fu.tt;

import androidx.appcompat.app.AppCompatActivity;

import android.content.Intent;
import android.os.Bundle;
import android.os.ResultReceiver;

public class SecondActivity extends AppCompatActivity {
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_second);

        Intent intent = getIntent();
        ResultReceiver receiver = intent.getParcelableExtra

        if (receiver != null) {
            Bundle bundle = new Bundle();
            bundle.putString("result", "Here is the result");
            receiver.send(200, bundle);
            finish();
        }
    }
}

```

&&&&&&&&

🔗 **Bundle**



```
package com.fu.tt;

import androidx.appcompat.app.AppCompatActivity;

import android.content.Intent;
import android.os.Bundle;
import android.view.View;
import android.widget.Button;

public class MainActivity extends AppCompatActivity {
    private Button button;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        button = findViewById(R.id.button);

        button.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View view) {
                Intent intent = new Intent(MainActivity.this,
                    MainActivity.class);
                Bundle bundle = new Bundle();
                bundle.putString("message", "Hello from MainActivity");
                bundle.putInt("number", 123);
                intent.putExtras(bundle);
                startActivity(intent);
            }
        });
    }
}
```



```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical"
    android:gravity="center"
    android:padding="16dp">

    <TextView
        android:id="@+id/textView"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Hello World!" />

    <Button
        android:id="@+id/button"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="a button" />

</LinearLayout>
```



```
package com.fu.tt;

import androidx.appcompat.app.AppCompatActivity;

import android.content.Intent;
import android.os.Bundle;
import android.widget.TextView;

public class SecondActivity extends AppCompatActivity {
    private TextView textView;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_second);

        textView = findViewById(R.id.textView);

        Intent intent = getIntent();

        // intent.putExtras(bundle);
        Bundle bundle = intent.getExtras();

        if (bundle != null) {
            String message = bundle.getString("message");
            int number = bundle.getInt("number");
            textView.setText(message + " " + number);
        }
    }
}
```

1. `getIntent()`: 这是一个 Activity 类中的方法，它返回用来启动此 Activity 的 Intent。每个 Activity 都是由某个 Intent 触发并启动的，这个 Intent 可以包含一些附加的数据，例如我们刚刚在例子中通过 Bundle 传递的数据。当你在一个 Activity 中调用 `getIntent()` 方法时，它会返回启动该 Activity 的原始 Intent。
2. `getExtras()`: 这是 Intent 类中的方法，它返回此 Intent 中包含的所有附加数据，这些数据以 Bundle 的形式返回。在我们的例子中，我们将包含消息和数字的 Bundle 添加到了 Intent 中，然后通过 `getExtras()` 方法在 SecondActivity 中取回了这个 Bundle。

```
<?xml version="1.0" encoding="utf-8"?>
<androidx.constraintlayout.widget.ConstraintLayout xmlns:android="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=".SecondActivity">

    <TextView
        android:id="@+id/textView"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Welcome to Second Activity!"
        android:textSize="24sp"
        app:layout_constraintBottom_toBottomOf="parent"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintTop_toTopOf="parent" />

</androidx.constraintlayout.widget.ConstraintLayout>
```



🔗 Parcelable

翻译

Parcelable: 可打包的

在 Android 中，有时你可能需要在不同的组件之间传递复杂的对象数据，比如从一个 Activity 到另一个 Activity。但是，Intent 中只能携带基本的数据类型，所以为了传递复杂的对象，我们需要将它们转换为可以轻松传输的形式。这就是 `Parcelable` 接口的作用。

`Parcelable` 是 Android 提供的一个接口，你可以通过实现这个接口，然后定义如何将对象的状态写入和恢复出“Parcel（包裹）”对象，来实现对象的序列化和反序列化。这样，就可以将对象放入 Intent 或 Bundle 中，进行传递了。

Bundle 和 **Parcelable** 是两个用于不同场景的工具，但它们都是 Android 中数据传输的基本机制。在理解它们的区别之前，让我们先来看看它们的主要用途。

1. **Bundle** : Bundle 是一个 Android 类，用于将数据封装为键值对 (Key-Value Pairs) 的形式。Bundle 通常在 Activities、Fragments 和 Services 之间传递数据时使用，它可以容纳不同类型的数据，如 boolean、byte、char、int、long、float、String、Parcelable 等。
2. **Parcelable** : Parcelable 是一个 Android 接口，它定义了如何将复杂对象序列化为一种可以在内存中传递的格式。这使得对象可以被存储到磁盘或通过 IPC (进程间通信) 发送到另一个进程。

现在，让我们来看看它们的区别：

- **数据类型：** Bundle 可以存储各种基本类型和一些 Android 特定类型的数据，如 Bundle, CharSequence, Parcelable, Serializable 等。而 Parcelable 是一种序列化机制，允许自定义对象进行序列化，以便可以在内存中传输。
- **性能：** 由于 Parcelable 是在内存中进行数据传输，因此比使用 Serializable (Java 自带的序列化接口) 进行序列化的 Bundle 更高效。这就是为什么 Android 官方推荐使用 Parcelable 而不是 Serializable。
- **复杂性：** 虽然 Parcelable 性能更高，但使用起来相对复杂，需要编写更多的代码。而 Bundle 则简单易用，无需编写额外的代码。

总的来说，Bundle 与 Parcelable 是 Android 数据传输的两种方式，它们各有优势。通常来说，如果你只需要传输一些简单的数据类型（如 int、String 等），那么 Bundle 就足够了。但是，如果你需要传输的是自定义对象，那么你可能需要实现 Parcelable 接口。

⋈⋈⋈⋈⋈⋈⋈⋈⋈⋈⋈⋈⋈⋈⋈⋈