第5章 Android组件与事件

5.1 Android组件

Android应用程序在Android应用框架之上，由一些系统自带的应用程序和用户创建的应用程序组成。组件是可以调用的基本功能模块，Android应用程序就是由组件组成的。一个Android应用程序通常包含4个核心组件和一个Intent，4个核心组件分别是Activity、Service、BroadcastReceiver和ContentProvider。Intent是组件之间进行通信的载体，不仅可以在同一个应用中起传递信息的作用，还可以在不同的应用间传递信息，如图5-1所示。

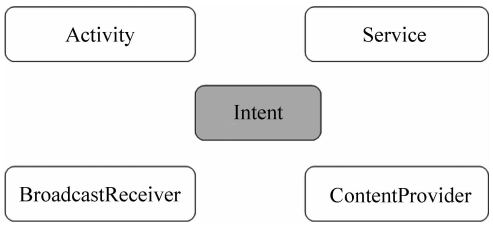

图5-1 Android应用程序组件

5.1.1 Android组件Activity

1. Activity的生命周期与创建

Activity是与用户交互的接口，提供了一个用户完成相关操作的窗口。当在开发中创建Activity后，通过调用setContentView(View)方法给该Activity指定一个布局界面，而这个界面就是提供给用户交互的接口。Android系统中是通过Activity栈的方式管理Activity的，而Activity自身则是通过生命周期的方法管理自己的创建与销毁。Activity生命周期流程图如图5-2所示。

Activity的形态如下。

(1) Active/Running：Activity处于活动状态，此时Activity处于栈顶，是可见状态，可与用户进行交互。

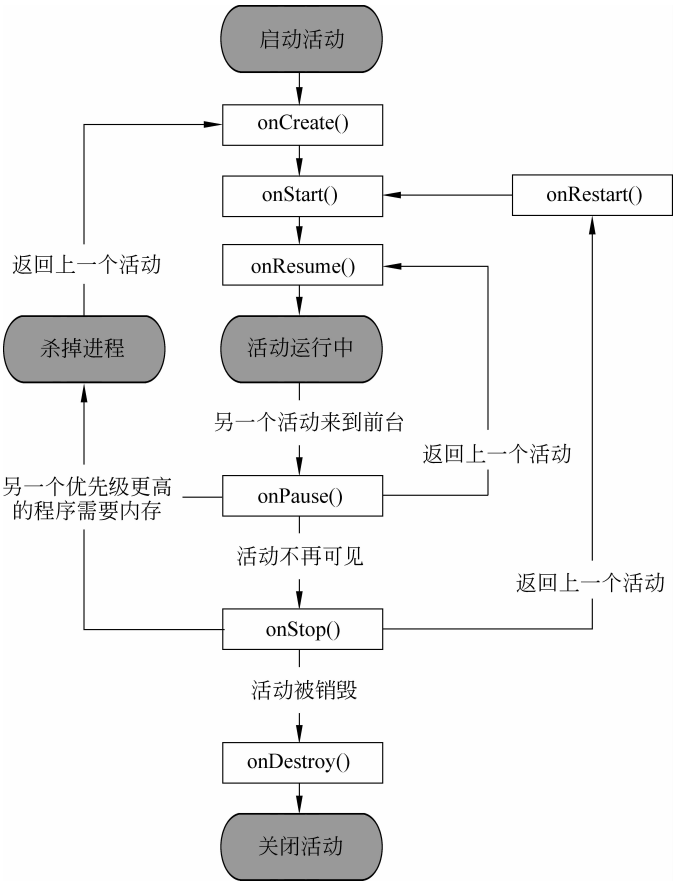

图 5-2　Activity 生命周期流程图

(2) Paused：当 Activity 失去焦点时，或被一个新的非全屏的 Activity，或被一个透明的 Activity 放置在栈顶时，Activity 就转化为 Paused 状态。但此时 Activity 只是失去了与用户交互的能力，其所有的状态信息及其成员变量都还存在，只在系统内存紧张的情况下，才有可能被系统回收。

(3) Stopped：当一个 Activity 被另一个 Activity 完全覆盖时，被覆盖的 Activity 就会进入 Stopped 状态，此时它不再可见，但是与 Paused 状态一样保持着其所有的状态信息及其成员变量。

(4) Killed：当 Activity 被系统回收时，Activity 就处于 Killed 状态。

所谓典型的生命周期，就是在有用户参与的情况下，Activity 经历创建、运行、停止、销毁等正常的生命周期过程。这里先介绍几个主要方法的调用时机，然后再通过代码层验证其调用流程。

(1) onCreate()：该方法在 Activity 被创建时调用，它是生命周期第一个调用的方法，创建 Activity 时一般都需要重写该方法，然后在该方法中做一些初始化的操作，如通过 setContentView()设置界面布局的资源，初始化所需要的组件信息等。

(2) onStart()：此方法被调用时表示 Activity 正在启动，此时 Activity 已处于可见状态，只是还没有在前台显示，因此无法与用户进行交互。可以简单理解为 Activity 已显示，

但我们无法看见。

(3) onResume()：当此方法被调用时，说明 Activity 已在前台可见，可与用户交互了(处于 Active/Running 形态)。onResume()方法与 onStart()的相同点是，两者都表示 Activity 可见，只不过 onStart()调用时 Activity 还是后台，无法与用户交互，而调用 onResume()时 Activity 已显示在前台，可与用户交互。当然，从流程图也可以看出，当 Activity 停止后(onPause()方法和 onStop()方法被调用)，重新回到前台时也会调用 onResume()方法，因此也可以在 onResume()方法中初始化一些资源，如重新初始化在 onPause()或者 onStop()方法中释放的资源。

(4) onPause()：此方法被调用时，表示 Activity 正在停止(处于 Paused 形态)。一般情况下，onStop()方法会紧接着被调用。通过流程图还可以看到的一种情况是：onPause()方法执行后直接执行了 onResume()方法，这属于比较极端的现象，这可能是用户操作使当前 Activity 退居后台后又迅速再回到当前的 Activity，此时 onResume()方法就会被调用。当然，在 onPause()方法中，我们可以做一些数据存储或者动画停止或者资源回收的操作，但是不能太耗时，因为这可能会影响到新的 Activity 的显示——onPause()方法执行完成后，新 Activity 的 onResume()方法才会被执行。

(5) onStop()：一般在 onPause()方法执行完成后直接执行，表示 Activity 即将停止或者完全被覆盖(处于 Stopped 形态)，此时 Activity 不可见，仅在后台运行。同样，在 onStop()方法可以做一些资源释放的操作(不能太耗时)。

(6) onRestart()：表示 Activity 正在重新启动，当 Activity 由不可见状态变为可见状态时，该方法被调用。这种情况一般是用户打开一个新的 Activity 时，当前的 Activity 就会被暂停(onPause()和 onStop()被执行了)，接着又回到当前 Activity 页面，onRestart()方法就会被调用。

(7) onDestroy()：此时 Activity 正在被销毁，也是生命周期最后一个执行的方法。一般地可以在此方法中做一些回收工作和最终的资源释放。

下面通过程序验证上面流程中的几种比较重要的情况，见 example5.1。

```
package com.cmcm.activitylifecycle;

import android.content.Intent;
import android.support.v7.app.AppCompatActivity;
import android.os.Bundle;
import android.view.View;
import android.widget.Button;
public class MainActivity extends AppCompatActivity {
    Button bt;
    //Activity 创建时被调用,@param savedInstanceState
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        LogUtils.e("onCreate is invoke!!!");
        bt=(Button) findViewById(R.id.bt);
        bt.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View v) {
```

```
                Intent i =new Intent(MainActivity.this,SecondActivity.class);
                startActivity(i);
            }
        });
    }
    //Activity从后台重新回到前台时被调用
    @Override
    protected void onRestart() {
        super.onRestart();
        LogUtils.e("onRestart is invoke!!!");
    }
    //Activity创建或者从后台重新回到前台时被调用
    @Override
    protected void onStart() {
        super.onStart();
        LogUtils.e("onStart is invoke!!!");
    }
    //Activity创建或者从被覆盖、后台重新回到前台时被调用
    @Override
    protected void onResume() {
        super.onResume();
        LogUtils.e("onResume is invoke!!!");
    }
    //Activity被覆盖到下面或者锁屏时被调用
    @Override
    protected void onPause() {
        super.onPause();
        LogUtils.e("onPause is invoke!!!");
    }
    //退出当前Activity或者跳转到新Activity时被调用
    @Override
    protected void onStop() {
        super.onStop();
        LogUtils.e("onStop is invoke!!!");
    }
    //退出当前Activity时被调用,调用之后Activity就结束了
    @Override
    protected void onDestroy() {
        super.onDestroy();
        LogUtils.e("onDestroy is invoke!!!");
    }
}
```

2. Activity 间的数据传递与交互

1）数据传递

假设有两个 Activity，即 MainActivity 与 SecondActivity，其中 MainActivity 是主活动。MainActivity 中有一个字符串，现在想把这个字符串传递到 SecondActivity，可以用 putExtra 与 getStringExtra 在 Activity 之间传递数据，见示例 example5.2。

```
public String getStringExtra(String name){
  return mExtras ==null?null:mExtras.getString(name);
}
MainActivity需要传递 data 给 SecondActivity,具体使用如下:
//MainActivity(sender)
        button.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View v) {
                String data =" hello world!";
                Intent intent =new Intent(MainActivity.this, SecondActivity.class);
                intent.putExtra("extra_data",data); //{"extra_data":data}
                startActivity(intent);
            }
        });
// SecondActivity(receiver)
public class SecondActivity extends Activity {
    @Override
    protected void onCreate(Bundle savedInstanceState){
        super.onCreate(savedInstanceState);
        requestWindowFeature(Window.FEATURE_NO_TITLE);
        setContentView(R.layout.second_layout);
        //通过 intent 获取数据
        Intent intent =getIntent();
        String data =intent.getStringExtra("extra_data");
        Toast.makeText(SecondActivity.this,data,Toast.LENGTH_SHORT).show();
    }
}
```

2）数据返回

因为数据返回一般是 Activity 的销毁，而不是调用，因此与 1）中的方式不一样。假设现在需要从 SecondActivity 返回数据给 MainActivity：

```
// SecondActivity
public class SecondActivity extends Activity {
    @Override
    protected void onCreate(Bundle savedInstanceState){
        super.onCreate(savedInstanceState);
        requestWindowFeature(Window.FEATURE_NO_TITLE);
        setContentView(R.layout.second_layout);
        Button button2 = (Button) findViewById(R.id.button_2);
        button2.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View v) {
                Intent intent =new Intent();
                intent.putExtra("data_return", "Hello MainActivity");
                //绑定 result_code 与 intent 的内容
                setResult(RESULT_OK,intent);
                finish();
```

```
            }
        });
    }
}
```

MainActivity 需要接收的参数有 request Code、result Code 和一个 intent。

```
public class MainActivity extends AppCompatActivity {
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        requestWindowFeature(Window.FEATURE_NO_TITLE);
        setContentView(R.layout.activity_main);
        final Button button = (Button) findViewById(R.id.button_1);
        button.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View v) {
                Intent intent =new Intent(MainActivity.this, SecondActivity.class);
                startActivityForResult(intent, 200);
            }
        });
    }
    @Override
    protected void onActivityResult(int requestCode, int resultCode, Intent data){
        switch (requestCode){
            case 200:
                if (requestCode ==RESULT_OK){
                    String returnedData =data.getStringExtra("data_return");
                    Log.d("MainActivity", returnedData);
                }
                break;
            default:
        }
    }
     ...
}
```

3）在 Bundle 中传递及保存数据

在 Bundle 中传递及保存数据是为了防止 Activity 调用时，当前 Activity 如果需要在 onDestroy()时保存一些临时数据，则可以用构造函数 onSaveInstanceState()。

保存数据示例 example5.3 如下。

```
@Override
    protected void onSaveInstanceState(Bundle outState){
        super.onSaveInstanceState(outState);
        String tempData ="Something you just typed";
        outState.putString("data_key",tempData);
    }
```

取出数据：

```
public class ActivityLifeCycleTest extends AppCompatActivity {
    public static final String TAG = "MainActivity";
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        Log.d(TAG, "onCreate");
        Toast.makeText(ActivityLifeCycleTest.this,TAG+"===onCreate==",Toast.
        LENGTH_SHORT).show();
        requestWindowFeature(Window.FEATURE_NO_TITLE);
        setContentView(R.layout.activity_activity_life_cycle_test);
        if (savedInstanceState !=null){
            String data =savedInstanceState.getString("data_key");
            Log.d(TAG, data);
        }
...
}
```

5.1.2 Android 组件 Service

Service 常用于没有用户界面，但需要长时间在后台运行的应用，与应用程序的其他模块(如 Activity)一同运行于主线程中。一般通过 startService()或 bindService()方法创建 Service，通过 stopService()或 stopSelf()方法终止 Service。通常，都在 Activity 中启动和终止 Service。

在 Android 应用中，Service 的典型应用是：音乐播放器，在一个媒体播放器程序中，大概要有一个或多个活动(Activity)供用户选择歌曲并播放。然而，音乐的回放就不能使用活动(Activity)了，因为用户希望能够切换到其他界面时音乐继续播放。这种情况下，媒体播放器活动(Activity)要用 Context. startService()启动一个服务在后台运行，保持音乐的播放。系统将保持这个音乐回放服务的运行，直到它结束。需要注意，要用 Context. bindService()方法连接服务(如果它没有运行，要先启动它)。当连接到服务后，可以通过服务暴露的一个接口和它通信。对于音乐服务，它支持暂停、倒带、重放等功能。

1. Service 与 Thread

Service(服务)：Android 四大组件之一，非常适合执行那些不需要与用户交互且要求长期执行的任务。需要注意：服务依赖于创建服务时所在的应用程序进程。

Thread(线程)：程序执行的最小单元，可以用 Thread 执行一些异步操作。

子进程和服务的使用及其区别如下。

(1) 子进程的使用：通常使用子线程完成耗时任务。但是，有时需要根据任务的执行结果更新显示相应的 UI 控件，要知道 Android 的 UI 与许多其他 GUI 库一样是线程不安全的，必须在主线程中操作，于是需要采用异步消息机制。为了方便起见，我们还是使用基于异步消息机制的 AsyncTask 抽象类：使用 AsyncTask 的诀窍在于，在 doInBackground(运行在子线程中)方法中执行耗时操作，在 onProgressUpdate(运行在主线程中)方法中进行 UI 操作，在 onPostExcute(运行在主线程中)方法中执行任务的收尾工作。

(2) 服务的使用：服务我们最初的理解是在后台处理一些耗时操作，但是不要被其所谓的后台概念迷惑，实际上服务不会自动开启线程，所有的代码都是默认运行在主线程中的，如果直接在服务中进行耗时操作，必定会阻塞主线程，出现 ANR 的情况。因此，需要在服务中手动创建子线程，在子线程中进行耗时操作。一个比较标准的服务如 example5.4 所示。

```
public class MyService extends Service {
    …
    @Override
    public int onStartCommand(Intent intent,int flags,int startId)
    {
        new Thread(new Runnable()
        {
            public void run()
            {
                //处理具体逻辑
            }
        }).start();
        return super.onStartCommand(intent,flags,startId);
        }
}
```

但是，这种服务一旦启动，必须调用 stopSelf()方法或 stopService()方法，才能停止该服务。为了简单地创建一个异步的、会自动停止的服务，Android 专门提供了一个 IntentService 类，只需要新建一个 MyIntentService，继承 IntentService 类，在 onHandlerIntent()方法中执行耗时操作即可，因为这个方法是在子线程中运行的，且这个服务在运行结束后会自动停止。

2. IntentService 的使用

Android 中的 IntentService 继承自 Service 类，我们在讨论 IntentService 之前，先想一下 Service 的特点：Service 的回调方法(onCreate、onStartCommand、onBind、onDestroy)都是运行在主线程中的。当通过 startService 启动 Service 之后，需要在 Service 的 onStartCommand()方法中写代码完成工作，但是 onStartCommand()是运行在主线程中的，如果需要在此处完成一些网络请求或 IO 等耗时操作，就会阻塞主线程 UI 无响应，从而出现 ANR 现象。为了解决这种问题，最好的办法是在 onStartCommand()中创建一个新的线程，并把耗时代码放到这个新线程中执行。在 onStartCommand()中开启新的线程作为工作线程执行网络请求，这样不会阻塞主线程。可见，创建一个带有工作线程的 Service 是一种很常见的需求(因为工作线程不会阻塞主线程)，所以 Android 为了简化开发带有工作线程的 Service，Android 额外开发了一个类——IntentService。

IntentService 的特点：

(1) IntentService 自带一个工作线程，当 Service 需要做一些可能会阻塞主线程的工作时，可以考虑使用 IntentService。

(2) 需要将要做的实际工作放入 IntentService 的 onHandleIntent()回调方法中，当通

过 startService(intent)启动 IntentService 之后，最终 Android Framework 会回调其 onHandleIntent()方法，并将 intent 传入该方法，这样就可以根据 intent 做实际工作，并且 onHandleIntent 运行在 IntentService 所持有的工作线程中，而非主线程。

(3) 当通过 startService 多次启动 IntentService，会产生多个 job，由于 IntentService 只有一个工作线程，所以每次 onHandleIntent 只能处理一个 job。面对多个 job，IntentService 会如何处理？处理方式是 one-by-one，也就是一个一个按照先后顺序处理，先将 intent1 传入 onHandleIntent，让其完成 job1，然后将 intent2 传入 onHandleIntent，让其完成 job2……，直至所有 job 完成，所以说 IntentService 不能并行执行多个 job，只能一个一个地按先后顺序完成，当所有 job 完成时，IntentService 就销毁了，会执行 onDestroy()回调方法。

IntentService 继承自 Service 类，并且 IntentService 重写了 onCreate()、onStartCommand()、onStart()、onDestroy()回调方法，IntentService 还添加了一个 onHandleIntent()回调方法。下面依次解释这几个方法在 IntentService 中的作用。

(1) onCreate：在 onCreate()回调方法中，利用 mName 作为线程名称，创建 HandlerThread，HandlerThread 是 IntentService 的工作线程。HandlerThread 执行了 start()方法后，其本身就关联了消息队列和 Looper，并且消息队列开始循环起来。

(2) onStartCommand：IntentService 重写了 onStartCommand()回调方法，即在内部调用 onStart()回调方法。

(3) onStart：在 onStart()方法中创建 Message 对象，并将 intent 作为 Message 的 obj 参数，这样 Message 与 Intent 就关联起来了，然后通过 Handler 的 sendMessage()方法将关联了 Intent 信息的 Message 发送给 Handler。

(4) onHandleIntent：在 onStart()方法中，通过 sendMessage()方法将 Message 放入 Handler 所关联的消息队列中后，Handler 所关联的 Looper 对象会从消息队列中取出一个 Message，然后将其传入 Handler 的 handleMessage()方法中，在 handleMessage()方法中首先通过 Message 的 obj 获取到原始的 Intent 对象，然后将其作为参数传给 onHandleIntent()方法让其执行。handleMessage()方法是运行在 HandlerThread 中的，所以 onHandleIntent()也是运行在工作线程中的。执行完 onHandleIntent()之后，需要调用 stopSelf(startId)声明某个 job 完成了。当所有 job 完成时，Android 会回调 onDestroy()方法，销毁 IntentService。

(5) onDestroy：当所有 job 完成时，Service 会销毁并执行其 onDestroy()回调方法。在该方法中调用了 Handler 的 quit()方法，该方法会终止消息循环。

总结：IntentService 可以在工作线程中完成工作，而不阻塞主线程，但是 IntentService 不能并行处理多个 job，只能依次处理，一个接一个，当所有 job 完成后，会自动执行 onDestroy()方法而无须自己调用 stopSelf()或 stopSelf(startId)方法。IntentService 并不神秘，只是 Android 对一种常见开发方式的封装，便于开发人员减少开发工作量。IntentService 是一个助手类，如果 Android 没有提供该类，也可以写一个类似的类。IntentService 之于 Service，类似于 HandlerThread 之于 Handler。

5.1.3 BroadcastReceiver 组件

1. BroadcastReceiver

在 Android 中，Broadcast 是一种广泛运用在应用程序之间传输信息的组件，而 BroadcastReceiver 是接收并响应广播消息的组件：对发送出来的 Broadcast 进行过滤接收并响应，它不包含任何用户界面，可以通过启动 Activity 或者 Notification 通知用户接收到重要消息，在 Notification 中有多种方法提示用户，如闪动背景灯、振动设备、发出声音或在状态栏上放置一个持久的图标。

BroadcastReceiver 过滤接收的过程如图 5-3 所示。

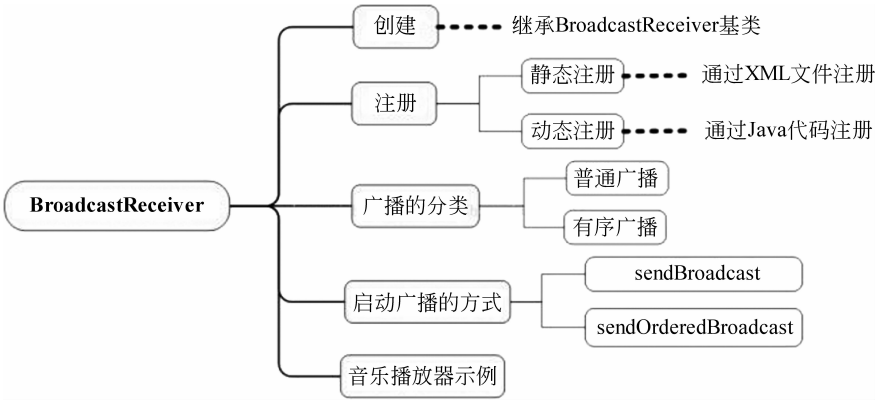

图 5-3 BroadcastReceiver 过滤接收的过程

需要发送消息时，把要发送的消息和用于过滤的信息（如 Action、Category）装入一个 Intent 对象，然后通过调用 Context. sendBroadcast ( )、sendOrderBroadcast ( ) 或 sendStickyBroadcast()方法，把 Intent 对象以广播方式发送出去。

当 Intent 发送后，所有已经注册的 BroadcastReceiver 会检查注册时的 Intent Filter 是否与发送的 Intent 相匹配，若匹配，则调用 BroadcastReceiver 的 onReceive()方法。因此，在定义一个 BroadcastReceiver 时，通常需要实现 onReceive()方法。

BroadcastReceiver 注册有以下两种方式。

（1）静态地在 AndroidManifest. xml 中用＜receiver＞标签声明注册，并在标签内用＜intent-filter＞标签设置过滤器。

（2）动态地在代码中先定义并设置好一个 Intent Filter 对象，然后在需要注册的地方调用 Context. registerReceiver()方法，如果取消，就调用 Context. unregisterReceiver()方法。

不管是用 XML 注册的，还是用代码注册的，程序退出时，一般需要注销，否则下次启动程序可能会有多个 BroadcastReceiver。另外，若在使用 sendBroadcast()方法时指定了接收权限，则只有在 AndroidManifest. xml 中用＜user-permission＞标签声明了拥有此权限的 BroadcastReceiver 时，才有可能接收到发送来的 Broadcast。

同样，若在注册 BroadcastReceiver 时指定了可接收的 Broadcast 的权限，则只有在包内的 AndroidManifest. xml 中用＜user-permission＞标签声明了拥有此权限的 Context 对