第3章　Android 应用程序组件

在进行深入开发之前，本章对开发的基本单元（程序组件）进行了详细而深入的介绍，包括各组件的使用方式、框架及配置，希望读者能够真切地了解各组件的特征和适用性，并在此基础上，能够就具体应用策划各组件的集成应用。

此外，本章还对组件之间的一些交互机制和方式进行了实例说明，通过这些实例希望读者能够深刻掌握这些机制的用法，为后面的应用集成奠定基础。

3.1　应用程序组件

有过软件项目开发经历的读者应该很了解，一个成型的项目往往需要由多个程序组件构成。一般软件项目中，比较常见的应用程序组件有：可执行文件（Windows 平台中以 exe 为后缀，Linux 平台中无后缀）和动态链接库（Windows 平台中以 dll 为后缀，Linux 平台中以 so 为后缀）。同为可执行程序，在用途表现方面可能存在较大的差异：有的可执行程序提供可视界面，与用户进行交互；有的不提供界面，而是作为后台服务。而对于动态链接库文件，有的仅包含资源定义，有的仅包含共享代码……其区分也是主要体现在功用方面。

应用程序组件的多样化，其目的就是为了适应各种不同场合的应用。例如：在 J2EE 平台，应用程序有 Applet、JSP、Servlet 等多种类型，各种应用程序有各自的适用场合。

3.2　Android 应用程序组件

同样的，在 Android 平台也存在多种类型的应用程序。通过第 1 章对平台的介绍以及第 2 章中对第一个应用程序开发过程的讲解，相信读者应该可以列举出 Android 平台的应用程序组件：Activity、Service、Broadcast Receiver 和 Content Provider，如图 3-1 是这些重要的组件的类结构层次。

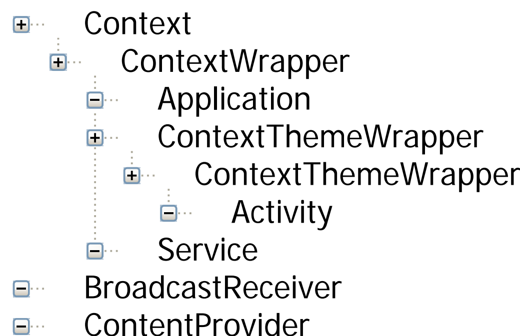

图 3-1 Android 平台应用程序组件类结构层次

表 3-1 是图 3-1 中应用程序组件的说明。

表 3-1　应用程序组件类/接口说明

类/接口	说明
android.app.Activity	Activity，关注于用户的行为
android.app.Service	服务，关注与后台事务的操作
android.content.Context	代表了应用程序所在环境的全局信息的接口
android.content.ContentResolver	用于访问应用程序的内容模型
android.content.BroadcastReceiver	广播接收器，用于接收广播消息
android.content.Intent	意图对象，用于描述将要执行的操作

3.2.1 Activity（活动）——形象大使

Activity 程序提供一组可视界面来与用户进行交互，主要用于处理前端事务，就像 Applet 程序或者 J2SE 平台中的 Swing 程序。作者将 Activity 组件定义为形象大使的角色，因为一款工具、游戏或者系统的用户界面的体验效果将会直接影响到该软件的形象。如果读者的程序中有很多“漂亮”的“形象大使”，那么一定会得到更多的关注。

图 3-2 是一个 Activity 程序运行的实例界面，如同其名字一样，只是一个简单的 Activity 程序。

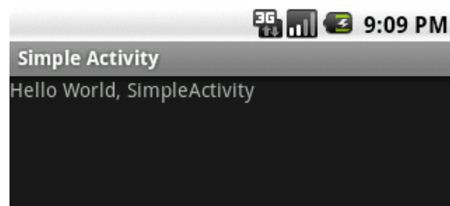

图 3-2 Activity 程序实例界面

代码 3-1 是图 3-1 所示的程序中主 Activity 的定义代码。

代码 3-1 Activity 定义代码

文件名：SimpleActivity.java

```
1	public class SimpleActivity extends Activity { //android.app.Activity
2	      private static final String TAG = "SIMPLE_ACT";
3	
4	    //Activity创建时回调
5	    @Override
6	    public void onCreate(Bundle savedInstanceState) {
7	        super.onCreate(savedInstanceState);
8	        //设置内容视图
9	        setContentView(R.layout.main);
10	
11	        Log.i(TAG, "Activity creating...");
12	    }
13	
14	    //Activity启动时回调
15	    @Override
16	    public void onStart() {
17	        super.onStart();
18	
19	        Log.i(TAG, "Activity starting...");
20	
21	        showInfo();
22	    }
23	
24	    //Activity停止时回调
25	    @Override
26	    public void onStop() {
27	        super.onStop();
28	
```

Download at Pin5i.Com

代码 3-1 Activity 定义代码

文件名：SimpleActivity.java

```
29            Log.i(TAG, "Activity stopping...");
30        }
31
32        //Activity销毁时回调
33        @Override
34        public void onDestroy() {
35            super.onDestroy();
36
37            Log.i(TAG, "Activity destroying...");
38        }
39
40        //显示Activity组件关联组件信息
41        private void showInfo() {
42            Log.i(TAG, "Title: " + this.getTitle().toString() );
43            Log.i(TAG, "Calling activity: " + this.getCallingPackage() );
44            //获取应用程序实例
45            Application app = this.getApplication();
46            Log.i(TAG, "Package:" + app.getPackageName() );
47            //获取应用程序上下文实例
48            Context appContext = this.getApplicationContext();
49            Log.i(TAG, "AppContext: " + appContext.toString() );
50            //获取资产管理器
51            AssetManager am = this.getAssets();
52            Log.i(TAG, "AssetManager: " + am.getLocales()[1]);
53            //获取基础上下文
54            Context baseContext = this.getBaseContext();
55            Log.i(TAG, "BaseContext: " + baseContext.toString() );
56            //内容解决者
57            ContentResolver resolver = this.getContentResolver();
58            Log.i(TAG, "Resolver: " + resolver.toString() );
59            //布局填充器
60            LayoutInflater inflater = this.getLayoutInflater();
61            Log.i(TAG, "LayoutInflater: " + inflater.toString() );
62            //菜单填充器
63            MenuInflater inflater2 = this.getMenuInflater();
64            Log.i(TAG, "MenuInflater: " + inflater2.toString() );
65            //资源管理器
66            Resources resources = this.getResources();
67            Log.i(TAG, "Resources: " + resources.toString() );
68            //墙纸
69            Drawable wallpaper = this.getWallpaper();
70            Log.i(TAG, "Wallpaper: " + wallpaper.toString() );
```

Download at Pin5i.Com

代码 3-1 Activity 定义代码

文件名：SimpleActivity.java

```
71              //获取窗体实例
72              Window window = this.getWindow();
73              LayoutParams layoutParams = window.getAttributes();
74              Log.i(TAG, "Layout height: " + layoutParams.height +
75                      ", width: " + layoutParams.width);
76              //获取窗体管理器
77              WindowManager wm = this.getWindowManager();
78              Display display = wm.getDefaultDisplay();
79              Log.i(TAG, "Window height: " + display.getHeight() +
80                      ", width: " + display.getWidth() );
81          }
82      };
```

对于代码 3-1，从结构上来分析：SimpleActivity 继承于父类 Activity（第 1 行），并重载了父类的“onCreate”、“onStart”、“onStop”和“onDestroy”方法，在“onStart”方法中输出了该 Activity 的有关信息（第 21 行中调用“showInfo”方法）。这种使用模式和 Java Applet 程序是一样的：读者所编写的 Applet 组件必须继承于父类 Applet 或 JApplet，并重载父类的部分方法，例如：“init”、“start”、“stop”、“destroy”和“paint”方法，在这些继承方法中进行当前 Applet 组件的初始、绘制和善后工作。

1. Activity 组件框架

简而言之，在 Android 平台，所有的 Activity 组件必须继承于其父类 Activity（在 android.app 包中），这一规则由 Android 平台的应用程序框架来约定，读者可以从 Applet 的程序框架获得启发。

提示：Activity 实际上和 Java Applet 一样都是一种程序框架，初学者没有必要一开始就深究它的由来。

日志 3-1 该 Activity 程序运行过程中在日志收集器（LogCat）中输出的内容。

日志 3-1 Activity 程序输出日志

文件名：LogCat

```
1    Activity creating...
2    Activity starting...
3    Title: Simple Activity
4    Calling activity: null
5    Package:foolstudio.demo
6    AppContext: android.app.Application@43735650
7    AssetManager: ja
8    BaseContext: android.app.ApplicationContext@43739788
9    Resolver: android.app.ApplicationContext$ApplicationContentResolver@43739810
10   LayoutInflater: com.android.internal.policy.impl.PhoneLayoutInflater@4359cf20
11   MenuInflater: android.view.MenuInflater@435a1cc0
12   Resources: android.content.res.Resources@435982c8
13   Wallpaper: android.graphics.drawable.BitmapDrawable@435989f8
14   Layout height: -1, width: -1
```

Download at Pin5i.Com

日志 3-1 Activity 程序输出日志

文件名：LogCat

```
15      Window height: 480, width: 320
16      Activity stopping...
17      Activity destroying...
```

2. Activity 程序生命周期

通过日志 3-1 读者可以看出，Activity 程序的生命周期是：创建→启动→停止→销毁，有关 Activity 程序生命周期的详细说明请参见 Android SDK 附带的文档，在此不在赘述。

3. Activity 程序界面资源绑定

在创建阶段，Activity 通过“setContentView”方法来设置可视界面（代码 3-1 第 10 行）。需要注意的是，“setContentView”方法的参数是一个资源 ID，对应布局资源文件夹中的“main.xml”文件，该文件中定义了一个布局资源，其内容如代码 3-2 所示。

代码 3-2 Activity 布局资源定义

文件名：main.xml

```
1   <?xml version="1.0" encoding="utf-8"?>
2   <LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
3       android:orientation="vertical"
4       android:layout_width="fill_parent"
5       android:layout_height="fill_parent">
6       <TextView
7           android:layout_width="fill_parent"
8           android:layout_height="wrap_content"
9           android:text="@string/hello"/>
10  </LinearLayout>
```

对布局的概念和布局资源的定义将在第 4 章中进行详细说明，这里只是让读者从框架上了解 Activity 程序与可视组件的绑定机制。

4. Activity 程序清单

通过日志 3-1 的第 3 行，读者可以知道该 Activity 程序的抬头是“Simple Activity”。但是，无论从代码 3-1 还是代码 3-2，读者都无法获知程序的抬头信息。实际上，当读者在 Eclipse 中通过 ADT 插件建一个 Android 程序时，该程序所有的相关信息都保存到一个名为“AndroidManifest.xml”的文件中。代码 3-3 是该文件的完整内容。

代码 3-3 程序清单文件定义

文件名：AndroidManifest.xml

```
1   <?xml version="1.0" encoding="utf-8"?>
2   <manifest xmlns:android="http://schemas.android.com/apk/res/android"
3         package="foolstudio.demo"
4         android:versionCode="1"
5         android:versionName="1.0">
6       <application android:icon="@drawable/icon" android:label="@string/app_name">
7           <activity android:name=".SimpleActivity" android:label="@string/app_name">
8               <intent-filter>
9                   <action android:name="android.intent.action.MAIN" />
10                  <category android:name="android.intent.category.LAUNCHER" />
```

Download at Pin5i.Com

代码 3-3　程序清单文件定义

文件名：AndroidManifest.xml

```
11                  </intent-filter>
12              </activity>
13          </application>
14          <uses-sdk android:minSdkVersion="3" />
15      </manifest>
```

通过代码 3-3，读者可以获知该 Activity 程序的包名、Android 版本代码、版本名称和 SDK 版本等与安装有关的信息。该应用程序的组成在<application>节点中进行描述，通过代码 3-3 读者可以了解，该应用程序中只包含一个 Activity。

提示：一个 Android 应用程序可以包含多个 Activity，但只能有一个 Activity 用于启动，多个 Activity 之间可以进行相互调用，形成一个 Activity 栈，Android 平台对 Activity 栈进行管理。

5. Activity 关联组件

从表现形式上，Activity 程序用来提供可视界面，但是在后台方面，Activity 几乎关联了 Android 应用程序框架中的大部分组件。从代码 3-1 中的“showInfo”方法（第 41 行）读者可以看出，Activity 的关联组件有：应用程序、应用程序上下文、资产管理器、基础上下文、内容解决者、布局填充器、菜单填充器、资源管理器、墙纸、窗体布局和窗体管理器，如图 3-3 所示。

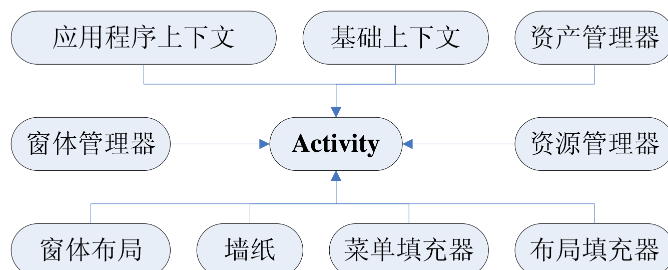

图 3-3 Activity 关联组件

图 3-3 中所反应的是大多数与 Activity 存在关联的组件，完整的内容请参考 Android SDK 参考文档。

3.2.2 Service（服务）——老黄牛

Android 平台对 Service 的定义和读者熟知的平台所描述的基本是一致的，Android 平台中的 Service 组件不提供可视界面，主要用于后台处理，例如：下载 Internet 文件、播放背景音乐等，其与用户的交互一般通过 Activity 组件进行桥接。作者将服务定义为老黄牛的角色，不言而喻，老黄牛勤勤恳恳，埋头苦干，总是很少抛头露面。

图 3-4 和图 3-5 分别是通过 Activity 的界面按钮启动和停止后台服务的运行界面。

Download at Pin5i.Com

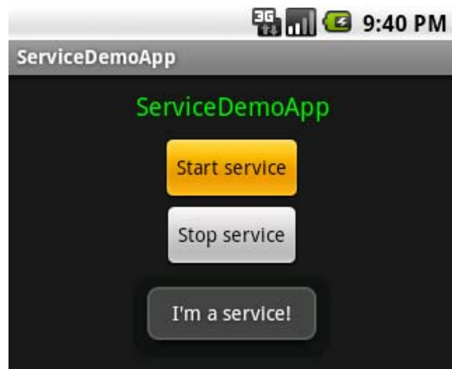

图 3-4 启动服务

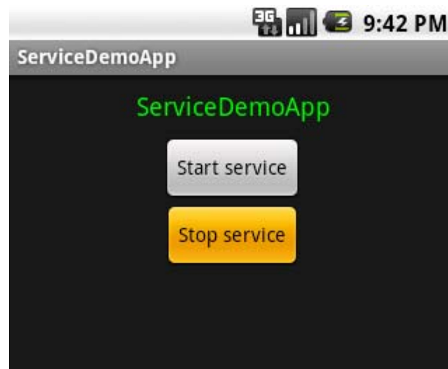

图 3-5 停止服务

1. 服务组件的使用方式

代码 3-4 是图 3-4 所示的用户界面对应的 Activity 的关键定义。代码中，服务组件是在 Activity 中进行启动和停止。

代码 3-4 启动服务的 Activity 关键代码

文件名：ServiceDemoAct.java

```
1   public class ServiceDemoAct extends Activity implements OnClickListener {
2       ……
3       //按钮被点击时回调
4       @Override
5       public void onClick(View v) {
6
7           switch(v.getId() ) {
8               case R.id.BTN_START: { //点击启动服务按钮
9                   doStart();
10                  break;
11              }
12              case R.id.BTN_STOP: { //点击停止服务按钮
13                  doStop();
14                  break;
15              }
16          }
17      }
18
19      //启动服务操作
20      private void doStart() {
21          Intent startService = new Intent(this, DummyService.class);
22          this.startService(startService);
23      }
24
25      //停止服务操作
26      private void doStop() {
27          Intent startService = new Intent(this, DummyService.class);
28          this.stopService(startService);
29      }
```

Download at Pin5i.Com

代码 3-4 启动服务的 Activity 关键代码

文件名：ServiceDemoAct.java

```
30      };
```

代码 3-4 中，通过 2 个按钮来分别开始和停止服务（第 8 行和第 12 行）。Activity 与服务通过 Intent 机制（在本章 3.3.1 节中作者将介绍该机制）来打交道，通过指定服务类名即可启动相关服务实例。

2. 服务组件框架

代码 3-5 是代码 3-4 中所参考的服务类（“DummyService”）的定义。

代码 3-5 服务类的定义

文件名：DummyService.java

```
1       public class DummyService extends Service {
2           //当（客户端）连接服务时回调
3           @Override
4           public IBinder onBind(Intent intent) {
5
6               return null;
7           }
8
9           //初始化时回调
10          @Override
11          public void onCreate() {
12
13              super.onCreate();
14
15              Log.d(getClass().getName(), "Service created.");
16          }
17
18          //服务启动时回调
19          @Override
20          public void onStart(Intent intent, int startId) {
21
22              super.onStart(intent, startId);
23
24              Toast.makeText(this, "I'm a service!", Toast.LENGTH_LONG).show();
25
26              Log.d(getClass().getName(), "Service starting...");
27          }
28
29          //服务实例销毁时回调
30          @Override
31          public void onDestroy() {
32
33              super.onDestroy();
34
```

Download at Pin5i.Com

代码 3-5 服务类的定义

文件名：DummyService.java

```
35                Log.d(getClass().getName(), "Service destroyed.");
36            }
37    };
```

和 Activity 一样，服务组件的定义也要遵循既定框架：所有的服务组件必须继承于其父类 Service（在 android.app 包中），子类服务通过在重载的方法（例如：代码 3-5 中“onBind”、“onCreate”、“onStart”和“onDestroy”方法）中实现其特定的任务。代码 3-5 中的服务组件只做了一件事情：弹出一个提示文本，告诉用户它是服务（第 24 行）。

3. 服务程序清单

代码 3-6 是该服务程序的清单文件内容。

代码 3-6 服务程序清单

文件名：AndroidManifest.xml

```
1     <?xml version="1.0" encoding="utf-8"?>
2     <manifest xmlns:android="http://schemas.android.com/apk/res/android"
3           package="foolstudio.demo"
4           android:versionCode="1"
5           android:versionName="1.0">
6         <application android:icon="@drawable/icon" android:label="@string/app_name">
7             <activity android:name=".ServiceDemoAct"
8                       android:label="@string/app_name">
9                 ……
10            </activity>
11             <service android:label="DummyService" android:name=".DummyService">
12                 <intent-filter>
13                     <action android:name="foolstudio.demo.DummyService" />
14                 </intent-filter>
15             </service>
16        </application>
17        <uses-sdk android:minSdkVersion="3" />
18    </manifest>
```

4. 服务程序界面资源绑定

代码 3-6 中表明，该服务程序由 1 个 Activity 和 1 个服务组件构成。服务组件不提供可视界面，所以它无需与界面资源（例如：布局、菜单等）进行绑定，但 Activity 还是需要定义布局资源。代码 3-7 是该服务程序中的 Activity 所用到的布局资源定义。

代码 3-7 服务示程序布局资源定义

文件名：main.xml

```
1     <?xml version="1.0" encoding="utf-8"?>
2     <LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
3         ……>
4         <TextView
5             ……/>
6         <Button android:id="@+id/BTN_START"
```

Download at Pin5i.Com

代码 3-7 服务示程序布局资源定义

文件名：main.xml

```
7               ……
8               android:text="Start service" />
9           <Button android:id="@+id/BTN_STOP"
10              ……
11              android:text="Stop service" />
12      </LinearLayout>
```

3.2.3 Broadcast Receiver（广播接收器）——倾听者

如果说 Activity 和服务都是实干派，那么将广播接收器组件定义为倾听者的角色是再恰当不过了。在 Android 平台，广播接收器组件用于接收和响应系统广播的消息。和 Service 一样，广播接收器也需要通过 Activity 与用户交互进行桥接。图 3-6 是注册广播接收器的执行界面，图 3-7 是广播接收器接收到消息并弹出的界面。

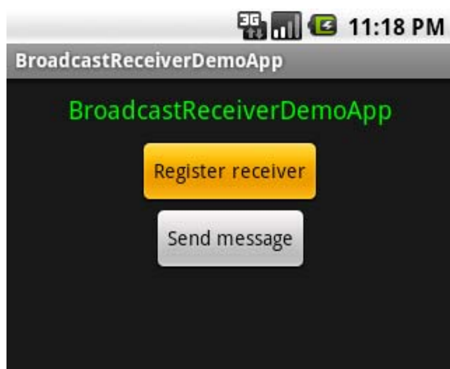

图 3-6 注册广播接收器组件

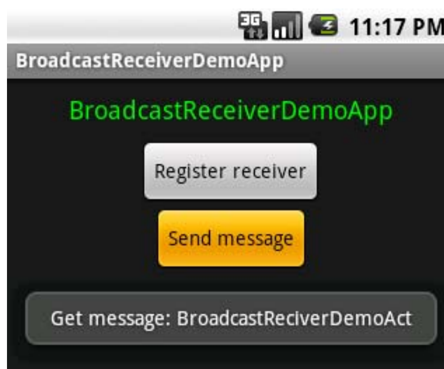

图 3-7 接收广播消息并弹出

1. 广播接收器的使用方式

代码 3-8 是图 3-6 所示的用户界面对应的 Activity 的关键定义。在 Activity 的定义中，首先对广播接收器实例进行注册，并向该广播接收器发送消息。

代码 3-8 广播接收器程序中 Activity 关键代码

文件名：BroadcastReceiverDemoAct.java

```
1   public class BroadcastReceiverDemoAct extends Activity implements OnClickListener {
2           //意图数据键值
3           public static final String INTENT_EXTRAS_NAME = "BroadcastReciverDemoAct";
4           //广播接收器实例
5           private DummyBroadcastReceiver mReceiver = null;
6           //意图过滤器
7           private IntentFilter mIntentFilter = null;
8
9          //Activity组件初始化时回调
10         @Override
11         public void onCreate(Bundle savedInstanceState) {
12              ……
13             //初始化广播接收器
14              mReceiver = new DummyBroadcastReceiver();
15             //使用广播接收器类名初始化意向过滤器
```

Download at Pin5i.Com

代码 3-8 广播接收器程序中 Activity 关键代码

文件名：BroadcastReceiverDemoAct.java

```
16            mIntentFilter = new IntentFilter(DummyBroadcastReceiver.class.getName() );
17        }
18
19        //Activity组件销毁时回调
20        @Override
21        protected void onDestroy() {
22
23            if(mReceiver != null) { //注销接收器
24                this.unregisterReceiver(mReceiver);
25            }
26
27            super.onDestroy();
28        }
29        ……
30        //注册接收器
31        private void doRegister() {
32            this.registerReceiver(mReceiver, mIntentFilter);
33        }
34
35        //发送消息
36        private void doSend() {
37            //为消息创建容器（意图对象实例）
38            Intent sendIntent = new Intent(DummyBroadcastReceiver.class.getName() );
39            //向意图附加容器中添加数据项
40            sendIntent.putExtra(INTENT_EXTRAS_NAME, INTENT_EXTRAS_NAME);
41            //发送消息"包裹"
42            this.sendBroadcast(sendIntent);
43        }
44    };
```

代码 3-8 中，首先分别定义了 1 个广播接收器实例（第 5 行）和 1 个过滤器（第 7 行），然后在第 14 行和第 16 行分别对广播接收器和过滤器进行初始化，其中，过滤器的初始化需要提供广播接收器的类名称。初始化完毕之后，在第 32 行，对广播接收器进行注册，让之开始“倾听”广播消息，而至于关注哪些消息，是由过滤器来“告诉”它。

并非只有 Android 平台才能进行“广播”，Activity 组件也可以发送广播消息，该过程也需要通过意向组件来实施（从第 38 行到第 42 行）。

2. 广播接收器组件框架

代码 3-9 是代码 3-8 中所参考的广播接收器对象的定义代码。

代码 3-9 广播接收器定义

文件名：DummyBroadcastReceiver.java

```
1    public class DummyBroadcastReceiver extends BroadcastReceiver {
2        //当接收消息时回调
3        @Override
```

Download at Pin5i.Com

代码 3-9 广播接收器定义

文件名：DummyBroadcastReceiver.java

```
4           public void onReceive(Context context, Intent intent) {
5
6                   String msg =
7                           intent.getStringExtra(BroadcastReceiverDemoAct.INTENT_EXTRAS_NAME);
8                   //弹出消息
9                   Toast.makeText(context, "Get message: " + msg, Toast.LENGTH_LONG).show();
10          }
11      };
```

与 Activity 和服务组件一样，所有广播接收器的定义必须继承父类 BroadcastReceiver（在 android.content 包中），在所重载的消息接收方法（代码 3-9 中第 4 行“onReceive”方法）中实现对消息的过滤判断和接收。

3. 广播接收器程序清单

在广播接收器示例程序中，广播接收器是在代码中进行注册，所以在程序清单中无需指明广播接收器，只需包含 1 个界面 Activity 组件即可，其内容结构和代码 3-3 是相同的。

实际上，在程序清单中也可以定义广播接收器，而且这种方式下，无需再在代码中进行定义广播接收器和注册操作。代码 3-10 是将广播接收器示例程序中的广播接收器定义在程序清单中的举例。

代码 3-10 广播接收器程序清单

文件名：AndroidManifest.xml

```
1       <application android:icon="@drawable/icon" android:label="@string/app_name">
2               <activity……>
3                       ……
4               </activity>
5               <receiver android:label="AlarmListener" android:name=".AlarmListener"/>
6       </application>
```

提示： 在代码中注册广播接收器和在程序清单中定义广播接收器这两种方式都可以实现接收广播消息。这两种使用方式的最大区别在于广播接收器的初始化方式：对于在代码中注册广播接收器的方式，用户可以定制该接收器的初始化方式（例如：传递初始化参数等，典型的是传递主线程消息队列处理器实例，由此实现将接收器接收内容传递到主线程界面中），而通过程序清单定义广播接收器的初始化过程由平台自动完成，用户无法干预，但是该方式无需显式地对广播接收器进行注册或注销。

4. 广播接收器程序界面资源绑定

和服务程序一样，虽然广播接收器组件不提供可视界面，所以其无需与界面资源进行绑定。但是广播接收器组件与用户的交互需要通过 Activity 来桥接，所以 Activity 的界面资源还是需要定义的。代码 3-11 是广播接收器示例程序中 Activity 的布局资源定义代码。

代码 3-11 广播接收器程序布局资源定义

文件名：main_view.xml

```
1       <?xml version="1.0" encoding="utf-8"?>
2       <LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
3               ……>
```

Download at Pin5i.Com

代码 3-11 广播接收器程序布局资源定义

文件名：main_view.xml

```
4           <TextView
5               ……
6               android:text="@string/app_name"/>
7           <Button android:id="@+id/BTN_REGISTER"
8               ……
9               android:text="Register receiver" />
10          <Button android:id="@+id/BTN_SEND"
11              ……
12              android:text="Send message" />
13      </LinearLayout>
```

3.2.4 Content Provider（内容提供者）

对内容提供者，通过名字就可以很好地理解该组件的角色。在 Android 平台，内容提供者用于将一个程序的数据通过约定手段提供给其他程序。内容提供者组件不提供可视组件，也无需直接与用户进行交互。通常情况下，需要数据的程序或组件按照约定方式从内容提供者那里获取数据，再通过可视界面显示数据。图 3-8 是一个从内容提供者读取并显示记录的程序界面。

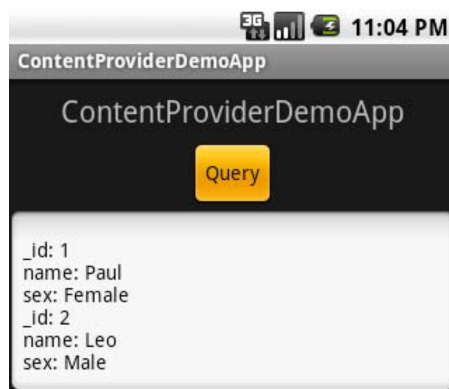

图 3-8 内容提供者示例程序界面

1. 内容提供者的使用方式

代码 3-12 是图 3-8 所示的用户界面对应的 Activity 的关键定义。在 Activity 的定义中，Activity 实例利用其关联组件 ContentResolver（内容解决者），通过对内容提供者约定的资源标识执行查询请求，进而得到数据记录。

代码 3-12 内容提供者程序 Activity 定义

文件名：ContentProviderDemoAct.java

```
1       public class ContentProviderDemoAct extends Activity implements OnClickListener{
2           //内容解决者实例
3           private ContentResolver mCR = null;
4           ……
5           @Override
6           public void onCreate(Bundle savedInstanceState) {
7               ……
8               //获取Activity组件的内容解决器对象实例
```

Download at Pin5i.Com

代码 3-12 内容提供者程序 Activity 定义

文件名：ContentProviderDemoAct.java

```
9               mCR = this.getContentResolver();
10          }
11          ……
12          //执行查询
13          private void doQuery() {
14              //生成资源全路径
15              Uri recUri = ContentUris.withAppendedId(MyDB.CONTENT_URI,
16                                                      MyDB.ALL_ROWS);
17              //不能进行类型转换，否则会抛出类转换异常
18              Cursor mc = mCR.query(recUri, null, null, null, null);
19
20              if(mc == null) {
21                  Toast.makeText(this, "获取游标失败！", Toast.LENGTH_LONG).show();
22                  return;
23              }
24              //游标复位
25              mc.moveToFirst();
26              //遍历游标
27              while(!mc.isAfterLast()) {
28                  printText("_id: " + mc.getInt(0) );
29                  printText("name: " + mc.getString(1) );
30                  printText("sex: " + mc.getString(2) );
31                  //移动到下一条记录
32                  mc.moveToNext();
33              }
34              mc.close();
35          }
36          ……
37      };
```

在代码 3-12 中，Activity 首先是通过其“getContentResolver”方法获取内容解决者组件（第 9 行），在通过内容解决者组件的查询方法“query”执行记录查询操作，并获取数据记录游标（第 18 行），最后通过游标的移动，获取所有记录集中的记录内容（第 32 行）。

读者需要注意的是，查询方法“query”所需的参数“resUri”描述的就是数据资源的 URI（Uniform Resource Identifier，统一资源标识），URI 的构成部分参考到数据提供者的定义属性（第 15 行和第 16 行）。

提示：URI 的规范定义于 RFC 2396：Uniform Resource Identifiers（URI），其定义语法为：[模式：]模式规范部分[#片段]，形如：http://localhost/url.html#10。有关 URI 的详细介绍请参见 RFC 2396 规范。

2. 内容提供者组件框架

代码 3-13 是代码 3-12 中所参考的内容提供者的定义。

代码 3-13 定制内容提供者定义

Download at Pin5i.Com

文件名：MyDB.java

```
1    public class MyDB extends ContentProvider {
2        //URI组成部分定义
3        public static final String URI_AUTHORITY = "foolstudio.demo.MyDB";
4        public static final String URI_PATH = "RecordSet";
5        public static final String URI_PATH2 = "RecordSet/#";
6        public static final int ALL_ROWS = 1;
7        public static final int SINGLE_ROW = 2;
8        //该URI的授权部分必须为有效类的全名
9        public static final Uri CONTENT_URI =
10           Uri.parse("content://foolstudio.demo.MyDB/RecordSet");
11       public static final UriMatcher uriMatcher;
12       //列名
13       public static final String _ID = "_id";
14       public static final String FIELD_NAME = "name";
15       public static final String FIELD_SEX = "sex";
16       //列名数组
17       public static final String COLUMN_NAMES[] = new String[] {
18           _ID, FIELD_NAME, FIELD_SEX
19       };
20       //初始化矩阵游标
21       public static MatrixCursor mCursor = new MatrixCursor(COLUMN_NAMES);
22       //定义列索引
23       public static final int INDEX_ID = 0;
24       public static final int INDEX_NAME = 1;
25       public static final int INDEX_SEX = 2;
26       //初始化URL匹配器
27       static {
28           uriMatcher = new UriMatcher(UriMatcher.NO_MATCH);
29           //添加授权、路径和片段部分
30           uriMatcher.addURI(URI_AUTHORITY, URI_PATH, ALL_ROWS);
31           uriMatcher.addURI(URI_AUTHORITY, URI_PATH2, SINGLE_ROW);
32       }
33
34       //接收删除请求时回调
35       @Override
36       public int delete(Uri uri, String selection, String[] selectionArgs) {
37           //先进行URL匹配判断
38           switch(uriMatcher.match(uri) ) {
39               case ALL_ROWS: { //执行删除所有行的操作
40                   break;
41               }
42               case SINGLE_ROW: { //执行删除指定行的操作
43                   break;
```

Download at Pin5i.Com

代码 3-13 定制内容提供者定义

文件名：MyDB.java

```
44                      }
45                }
46                return 0;
47          }
48          //接收查询请求时回调
49          @Override
50          public String getType(Uri uri) {
51                switch(uriMatcher.match(uri) ) { //先进行URL匹配判断
52                      case ALL_ROWS: {
53                            return ("vnd.android.cursor.dir/vnd.foolstudio.MyDB");
54                            //break;
55                      }
56                      case SINGLE_ROW: {
57                            return ("vnd.android.cursor.item/vnd.foolstudio.MyDB");
58                            //break;
59                      }
60                }
61                return null;
62          }
63
64          //接收记录插入请求时回调
65          @Override
66          public Uri insert(Uri uri, ContentValues values) {
67                ……
68                return null;
69          }
70
71          //初始化游标
72          @Override
73          public boolean onCreate() {
74                mCursor.addRow(new String[] {"1", "Paul", "Female"} );
75                mCursor.addRow(new String[] {"2", "Leo", "Male"} );
76
77                return true;
78          }
79
80          //接收查询请求时回调
81          @Override
82          public Cursor query(Uri uri, String[] projection, String selection,
83                      String[] selectionArgs, String sortOrder) {
84                ……
85                return (mCursor);
```

Download at Pin5i.Com

代码 3-13 定制内容提供者定义

文件名：MyDB.java

```
86          }
87
88          //接收更新请求时回调
89          @Override
90         public int update(Uri uri, ContentValues values, String selection, String[] selectionArgs) {
91                ……
92                return 0;
93          }
94     };
```

代码 3-13 中，与 Activity 和服务组件一样，所有内容提供者的定义必须继承父类 ContentProvider（android.content 包中），在所重载的方法中实现对数据记录的删除（第 36 行）、插入（第 66 行）、创建（第 73 行）、更新（第 90 行）和查询（第 82 行）等操作。

其中代码中所定义的一些静态成员都是内容提供者程序框架所要求定义的，更为详细约定请参考 Android SDK 文档。

3. 内容提供者程序清单

代码 3-14 是数据提供者示例程序的清单文件内容，其中“<provider>”标记（第 10 行）就是内容提供者组件的定义开始。

代码 3-14 数据提供者示例程序清单

文件名：AndroidManifest.xml

```
1     <?xml version="1.0" encoding="utf-8"?>
2     <manifest xmlns:android="http://schemas.android.com/apk/res/android"
3              package="foolstudio.demo"
4              ……>
5          <application android:icon="@drawable/icon" android:label="@string/app_name">
6               <activity android:name=".ContentProviderDemoAct"
7                           android:label="@string/app_name">
8                   ……
9               </activity>
10               <provider android:name=".MyDB" android:label="MyDB"
11                      android:authorities="foolstudio.demo.MyDB"/>
12          </application>
13         <uses-sdk android:minSdkVersion="3" />
14    </manifest>
```

4. 内容提供者程序界面资源绑定

从代码 3-14 中读者可以看到，该服务程序由 1 个 Activity 和 1 个内容提供者组件构成。内容提供者组件不提供可视界面，所以它也无需与界面资源进行绑定，而其中 Activity 还是需要定义布局资源。代码 3-15 是该内容提供者示例程序中的 Activity 所用到的布局资源定义。

代码 3-15 内容提供者示例程序界面资源定义

文件名：main.xml

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

Download at Pin5i.Com

代码 3-15 内容提供者示例程序界面资源定义

文件名：main.xml

```
2     <LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
3         ……>
4         <TextView
5             …… />
6         <Button android:id="@+id/BTN_QUERY"
7             ……
8             android:text="Query"/>
9         <EditText android:id="@+id/TXT_CONTENTS"
10            ……/>
11    </LinearLayout>
```

3.2.5 Android 应用程序组件小结

多种不同类型的应用程序造就了丰富多彩的 Android 平台，这些程序中，既有热衷于表现的“形象大使”（Activity），也有习惯于埋头苦干的“老黄牛”（服务）；有冷静的“倾听者”（广播接收者），也有热情的“奉献者”（内容提供者）。各种程序各有所长，能够满足 Android 平台所有的应用场合。

3.3 组件应用机制

在对 Android 平台 4 种应用程序组件的介绍中，作者提到了有关组件之间的相互调用、发送广播消息等应用。而这些应用机制正是关联所有应用程序的强韧纽带，有了这些纽带，就能够把各种不同类型、不同功能的应用程序“团结”在一起，从而帮助用户完成各种复合的应用。

在 Android 平台中，比较常用的应用机制有，组件与组件、组件与线程和组件与服务之间的交互机制。

3.3.1 组件与组件间的交互机制

在 Android 平台，组件与组件之间的交互通过 Intent（意向）组件来实现。在 SDK 的参考中，Android 平台把意向归纳为激活组件，就是用于激活其他组件的组件。作者在这里把它归纳到应用机制进行讲解，其主要目的是为了读者区分地理解应用程序组件特性和这些组件的使用方法，从程序框架的角度来理解 Android 平台中应用程序的使用模式。

图 3-9 和图 3-10 所示的应用程序中，主 Activity 启动一个新的 Activity，并将数据记录传递给该 Activity 组件进行显示。

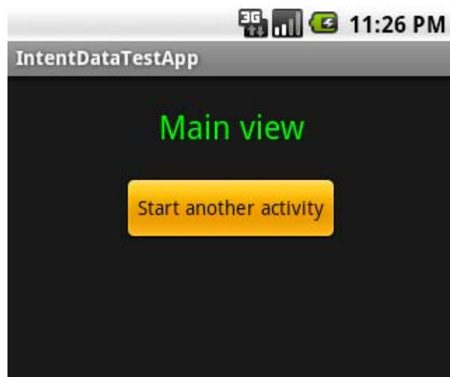

图 3-9 启动新的 Activity

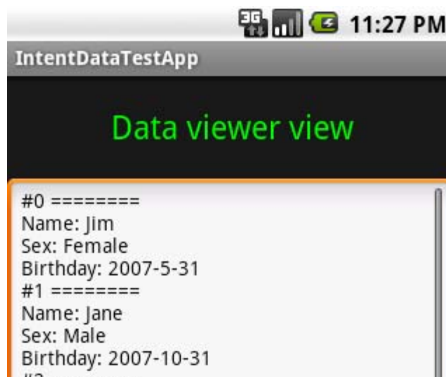

图 3-10 Activity 组件间数据传递

Download at Pin5i.Com

代码 3-16 是图 3-9 所对应的程序的主 Activity 定义代码，其主要功能是提供程序主界面，通过按钮启动创建意向对象，并启动另外一个 Activity（DataViewerAct）。

代码 3-16 意图示例程序主 Activity 定义

文件名：IntentDataTestAct.java

```
1	public class IntentDataTestAct extends Activity implements OnClickListener {
2		//请求识别码
3		public static final int REQ_CODE = 2012;
4		//附加数据键值
5		public static final String EXTRAS_KEY = "EXTRAS_DATA";
6		//数据记录容器
7		private ArrayList<Kid> mKids = new ArrayList<Kid>();
8	
9		/*Andoid平台禁止访问Activity的构造函数，否则抛出异常
10		public static IntentDataTestAct mInstance = new IntentDataTestAct();
11		private IntentDataTestAct() { //单例模式，构造函数无需向外部提供
12		}
13		//单例模式获取Activity类实例
14		public static IntentDataTestAct getInstance() {
15			return (mInstance);
16		}
17		*/
18	
19		@Override
20		public void onCreate(Bundle savedInstanceState) {
21			……
22		}
23	
24		@Override
25		public void onClick(View v) {
26			……
27		}
28	
29		//启动新的活动
30		private void doStart() {
31			//通过Activity类名创建意向对象实例
32			Intent startNew = new Intent(this, DataViewerAct.class);
33			//初始化数据集
34			ArrayList<Kid> kids = initArrayList();
35			//将数据集加入到意向对象的附加容器中
36			startNew.putParcelableArrayListExtra(EXTRAS_KEY, kids);
37			//通过意向参数启动新的Activity组件
38			this.startActivity(startNew);
39			//以要求反馈结果的方式启动新的Activity组件
40			//this.startActivityForResult(startNew, REQ_CODE);
```

Download at Pin5i.Com

代码 3-16 意图示例程序主 Activity 定义

文件名：IntentDataTestAct.java

```
41          }
42
43          //当接收调用Activity反馈结果时回调
44          @Override
45          protected void onActivityResult(int requestCode, int resultCode, Intent data) {
46              if(requestCode == REQ_CODE) { //判断是否合法的请求代码
47                  //获取反馈结果数据包
48                  Bundle bundle = data.getExtras();
49                  //从包中获取消息内容并显示
50                  String msg = bundle.getString("Msg");
51                  Toast.makeText(this, msg, Toast.LENGTH_LONG).show();
52              }
53
54              super.onActivityResult(requestCode, resultCode, data);
55          }
56
57          //初始化数据记录列表
58          private ArrayList<Kid> initArrayList() {
59              addKid(mKids, "Jim", Kid.SEX_FEMALE, "2007-5-31");
60              ……
61              return mKids;
62          }
63
64          //获取数据记录列表
65          public ArrayList<Kid> getArrayList() {
66              return (mKids);
67          }
68          ……
69      };
```

在代码 3-16 中有一段注释了的代码（第 9 行到第 17 行），其本意是：主 Activity 以单例（Singleton）的形式向外部组件提供获取类实例对象的接口（第 14 行），在显示数据的 Activity 组件中通过主 Activity 提供的接口获取主 Activity 类实例，继而通过其数据访问接口（第 65 行）获取数据并显示。

但是，通过单例模式来实现 Activity 之间的数据共享的想法在 Android 平台行不通，因为 Android 平台禁止应用程序访问 Activity 组件的构造函数，其异常输出如下所示。

```
java.lang.RuntimeException: Unable to instantiate activity ComponentInfo
java.lang.IllegalAccessException: access to constructor not allowed
```

实际上，Android 平台的这一禁止规则，正是用来规范应用程序组件之间的数据传递机制，保证组件的数据安全。

1. Activity 组件调用 Activity 组件的方式

在 Android 平台，Activity 组件可以通过“startActivity”方法来调用其他 Activity 组件，

Download at Pin5i.Com

该方法有且仅有的一个参数就是意向对象（代码 3-16 第 38 行），需要传递的数据存放到意向对象的附加容器中（代码 3-16 第 36 行）。

在使用“startActivity”方法调用新的 Activity 组件的方式中，新的 Activity 将不会反馈执行结果给调用它的 Activity 组件。而如果既需要调用新的 Activity 组件，而且还需要该组件将结果反馈给调用方 Activity，那么需要使用“startActivityForResult”方法（代码 3-16 第 40 行），该方法的第 1 个参数还是意向对象实例，第 2 个参数是请求代码，用来识别反馈结果是否为预期。

2. 意向对象的内涵

既然其他 Activity 组件通过意向对象来调用，那么在意向对象中，需要包含哪些内容：

（1）目标组件或者选择条件，告诉平台指定或者判断由哪个 Activity 组件来执行任务；

（2）行为方式，指明任务的行为方式。例如，对于记录工具，是执行增加还是删除操作。

（3）资源标识或数据，指明需要处理的内容。

有关意向对象的详细说明请参见 Android SDK 参考文档。

3. 意向对象的附加容器

在代码 3-16 的第 37 行，意图对象通过“putParcelableArrayListExtra”方法将包含多条小孩（Kid）记录的一个 ArrayList 对象添加到该意图对象的附加容器中，该数据项有一个字符串（常量“EXTRAS_KEY”）作为键，类似于 Map 容器操作。所以，读者可以初步推断出意向对象的附加容器 Bundle，类似于 Map，只是其键的类型是字符串。

4. 意向对象的附加容器中的记录

既然 Activity 与 Activity 之间的数据无法通过简单地内存共享来实现，只能通过意向对象的附加容器进行传递，那么意向对象的附加容器中的记录又是一种什么结构呢。代码 3-17 是代码 3-16 中数据记录（小孩信息）的定义。

代码 3-17 意图对象附加容器中记录的定义

文件名：Kid.java

```
1    public class Kid implements Parcelable {
2        public static final int SEX_FEMALE = 1;
3        public static final int SEX_MALE = 0;
4        //属性字段
5        private String name = null;
6        private int sex = 1;
7        private String birthday = null;
8
9        //必须要有一个名为CREATOR的成员对象，否则无法进行Parcelable对象通信
10       public static final Parcelable.Creator<Kid> CREATOR = new Parcelable.Creator<Kid>() {
11           public Kid createFromParcel(Parcel in) {
12               return new Kid(in);
13           }
14           public Kid[] newArray(int size) {
15               return new Kid[size];
16           }
17       };
18       public Kid(String _name, int _sex, String _birthday) {
19           this.birthday = _birthday;
```

Download at Pin5i.Com

代码 3-17 意图对象附加容器中记录的定义

文件名：Kid.java

```
20              this.sex = _sex;
21              this.name = _name;
22          }
23          //设置和获取姓名信息
24          public void setName(String name) {
25              this.name = name;
26          }
27          public String getName() {
28              return name;
29          }
30          //设置和获取生日信息
31          public void setBirthday(String birthday) {
32              this.birthday = birthday;
33          }
34          public String getBirthday() {
35              return birthday;
36          }
37          //设置和获取性别信息
38          public void setSex(int sex) {
39              this.sex = sex;
40          }
41          public int getSex() {
42              return sex;
43          }
44          ……
45          //实现Parcelable接口（从包裹中构造对象实例）
46          public Kid(Parcel in) {
47              this.name = in.readString();
48              this.sex = in.readInt();
49              this.birthday = in.readString();
50          }
51
52          @Override
53          public int describeContents() {
54              return 0;
55          }
56
57          //用于定义写对象到包裹中的方法
58          @Override
59          public void writeToParcel(Parcel dest, int flags) {
60              dest.writeString(this.name);
61              dest.writeInt(this.sex);
```

Download at Pin5i.Com

代码 3-17 意图对象附加容器中记录的定义

文件名：Kid.java

```
62                dest.writeString(this.birthday);
63           }
64      };
```

与通常类的定义代码相比，读者也许觉得代码 3-17 过于复杂，可能对属性“CREATOR”和方法“writeToParcel”的定义不知所云。这些“莫名其妙”属性和方法，却正是 Android 平台对于可以通过 IPC（Inter-process Communication，进程间通信）机制进行传递的数据类的定义进行的约定，Activity 与 Activity 之间可以传递的数据类必须满足该约定。否者将会抛出如下的异常：

```
android.os.BadParcelableException: Parcelable protocol requires a Parcelable.Creator object called
CREATOR on class foolstudio.demo.Kid
```

代码中第 2 行的 Parcelable 接口就是代表了可以通过 Parcel（包裹）进行数据传递的功能特性。只有实现了 Parcelable 接口的类对象才能通过意向对象的附加空间进行传递，实际上，意图类（Intent）本身也实现了 Parcelable 接口。

实现于 Parcelable 接口的“CREATOR”属性用于“告诉”平台如何创建该类的实例（第 10 行）；而 “writeToParcel”方法用于“告诉”平台如何将该类的数据存储到“包裹”中（第 59 行）。通过对属性名和方法名进行约定，平台可以获知该对象的数据的读取和写入的接口，从而可以进行对象的实例化（从包裹中创建类实例）和持久化（将类实例存储到包裹中）。

5. 被调用方 Activity 接收传递数据

在代码 3-16 中，主 Activity 将小孩记录数组通过意向对象的附加容器传递给了数据显示 Activity，代码 3-18 是显示数据的 Activity 组件的定义。

代码 3-18 记录显示 Activity 组件的定义

文件名：DataViewerAct.java

```
1    public class DataViewerAct extends Activity {
2         @Override
3         public void onCreate(Bundle savedInstanceState) {
4              ……
5              //获取父Activity所传递的附加数据
6              Intent intent = this.getIntent();
7    ArrayList<Kid> kids = intent.getParcelableArrayListExtra(IntentDataTestAct.EXTRAS_KEY);
8              for(int i = 0; i < kids.size(); ++i) {
9                   Kid kid = kids.get(i);
10                  ……
11             }
12
13             //通过意向组件将结果发送到调用方Activity
14             Intent result = new Intent();
15             //填充结果数据包
16             result.putExtra("Msg", "Get " + kids.size() + " record(s).");
17             //设置返回结果
```

Download at Pin5i.Com

代码 3-18 记录显示 Activity 组件的定义

文件名：DataViewerAct.java

```
18                this.setResult(IntentDataTestAct.REQ_CODE, result);
19            }
20     };
```

在代码 3-18 中，Activity 组件通过“getIntent”方法获取到与当前 Activity 组件关联的意向对象实例（第 6 行），然后通过意向对象实例的“getParcelableArrayListExtra”方法从意向对象实例的附加空间按照指定的键进行检索，获取到之前存储的、包含小孩记录的记录数组对象（第 7 行），然后再遍历数据，输出记录内容（第 8 行）。

6. 被调用方 Activity 反馈结果

在 Activity 组件调用 Activity 组件的方式介绍时，作者提到，通过“startActivityForResult”方法调用 Activity 组件，被调用的 Activity 还可以将结果反馈给调用方 Activity 组件。如图 3-11 所示，在被调用 Activity 组件关闭时，在主 Activity 中接收到被调用 Activity 组件反馈的结果信息。

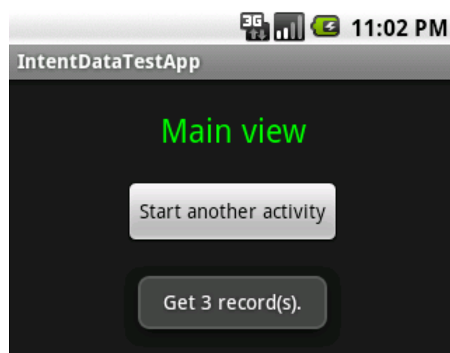

图 3-11 获取被调用 Activity 的返回结果

代码 3-18 中，从第 14 行到第 18 行就是被调用方将结果反馈给调用方 Activity 组件的核心代码。读者可以看出，被调用方组件反馈结果给调用方也需要使用意向组件（第 14 行），被调用方将需要反馈的数据项以“键—值”的方式添加到该意向对象的附加容器中（第 16 行），然后通过“setResult”方法反馈给调用方组件（第 18 行）。

调用方 Activity（主 Activity 组件）通过重载父类的“onActivityResult”方法来获取所收到的反馈结果（代码 3-16 中第 45 行）。

注意：调用方组件对结果的读取与被调用方发送结果的过程是逆向的，而且被调用方反馈结果时还附带了一个请求识别码（IntentDataTestAct.REQ_CODE，代码 3-18 中第 18 行），而该识别码正是调用方用于判断其所接收的结果是否为预期的判别码（代码 3-16 中第 54 行）。

7. 程序清单

通过意向组件，读者可以轻松地实现在一个 Activity 组件中调用另外一个组件。而实际上，被调用的 Activity 要事先“告诉”应用平台，否则运行时将会抛出目标 Activity 组件没有找到的异常，如下所示。

```
android.content.ActivityNotFoundException: Unable to find explicit activity class
have you declared this activity in your AndroidManifest.xml?
```

异常输出中提示，所有的 Activity 组件都必须在清单文件中进行声明。代码 3-19 是该

Download at Pin5i.Com

意向示例程序的清单文件内容。

代码 3-19 包含多个 Activity 组件的程序清单

文件名：AndroidManifest.xml

```
1    <?xml version="1.0" encoding="utf-8"?>
2    <manifest xmlns:android="http://schemas.android.com/apk/res/android"
3          package="foolstudio.demo"
4          android:versionCode="1"
5          android:versionName="1.0">
6        <application android:icon="@drawable/icon" android:label="@string/app_name">
7            <activity android:name=".IntentDataTestAct"
8                       android:label="@string/app_name">
9                <intent-filter>
10                   <action android:name="android.intent.action.MAIN" />
11                   <category android:name="android.intent.category.LAUNCHER" />
12               </intent-filter>
13           </activity>
14           <activity android:name=".DataViewerAct"/>
15       </application>
16       <uses-sdk android:minSdkVersion="3" />
17   </manifest>
```

代码 3-19 中，第 7 行和第 14 行分别声明了 1 个 Activity 组件，但是通过“<intent-filter>”标记读者可以看出，第 7 行定义的 Activity 组件是整个程序的主组件（第 10 行），且作为启动用（第 11 行）。

需要补充的是，无论是主 Activity 组件还是从 Activity，都需要绑定可视界面的（即图 3-9 和图 3-10 所示），那么还必须为这两个 Activity 组件分别定义界面布局资源。有关界面布局资源的定义将在第 4 章中进行说明。

3.3.2 未决意向对象

如果说意向对象所描述的是即将执行的动作（该动作会马上被执行），那么未决意向对象（PendingIntent）描述的却是稍后将要执行的动作（该动作可能会被取消），例如：闹钟设定、短信发送、任务通知等。

读者可以把未决意向理解为带有条件的意向（Intent），这样未决意向对象实例还需要依据意向对象来获取。通过 PendingIntent 类的静态方法“getActivity”、“getBroadcast”和“getService”可以分别获取用于启动 Activity、执行广播和启动服务的未决意向实例。未决对象实例不能用来执行操作，所以，还必须由操作主体通过指定未决对象实例来启动操作，Android 平台会从未决对象实例中获取该操作不是立即执行，而是需要满足某种条件（在指定时刻或超过延迟时段等）。

在通过未决意向对象启动 Activity 组件的过程中，还必须通过意向对象“告诉”Activity 管理器新 Activity 与调用方没有关系，新 Activity 是新任务（FLAG_ACTIVITY_NEW_TASK）。

有关未决意向对象的使用实例，可以参考短信发送（第 8.2.2 节）、系统通知（第 15.1.9 节）和闹钟设置（第 15.1.3 节）应用。

3.3.3 组件与线程间的交互机制

有些读者可能以这样的方式使用线程：通过用户界面（例如：按钮）启动线程，然后在线程的执行代码中将状态信息输出到用户界面（例如：文本框）。在 J2SE 平台，这样的使

Download at Pin5i.Com

用方式可能不会遇到什么问题；但是在 Android 平台，将会抛出以下的异常信息：

android.view.ViewRoot$CalledFromWrongThreadException: Only the original thread that created a view hierarchy can touch its views.

该异常的意思是，只有最初创建视图层次结构的线程才能接触该结构中的视图，其言外之意就是，不是最初创建界面的线程是不能接触界面元素的。那么，在不是创建界面的线程中，如何将内容输出到界面元素中呢？

1. 线程消息队列

Android 平台提供了一种称为线程消息队列（Message Queue）的机制来解决上述使用中遇到的问题。首先在界面线程中创建一个可以与界面线程的消息队列进行关联的接口实例，其它的线程通过这个接口实例就可以将消息发送到界面线程的消息队列中，最后由界面线程将消息内容输出到界面容器中，这个接口实例就是一个 Handler（处理者）类实例。

图 3-12 是一个 Activity 组件与外部线程交互的示例程序的运行界面。

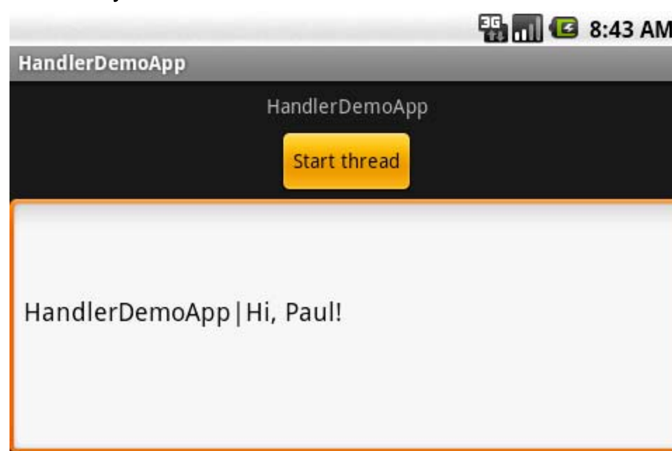

图 3-12 Activity 组件与外部线程交互的示例程序界面

代码 3-20 是图 3-11 所对应的 Activity 的定义代码。

代码 3-20 调用线程的 Activity 定义

文件名：HandlerDemoAct.java

```
1    public class HandlerDemoAct extends Activity implements OnClickListener {
2        ……
3        @Override
4        public void onCreate(Bundle savedInstanceState) {
5            ……
6            //初始化线程消息队列接口实例
7            mHandler = new Handler() {
8                //处理消息时回调
9                @Override
10               public void handleMessage(Message msg) {
11                   //获取消息数据
12                   Bundle bundle = msg.getData();
13                   //从消息数据集中获取数据项
14                   String sender = bundle.getCharSequence("Sender").toString();
15                   String data = bundle.getString("Msg");
16                   //将消息输出到可视界面
```

Download at Pin5i.Com

代码 3-20 调用线程的 Activity 定义

文件名：HandlerDemoAct.java

```
17                      addMsg(sender+"|"+data);
18
19                      super.handleMessage(msg);
20                  }
21              };
22          }
23          ……
24          //启动线程
25          private void doStart() {
26              //创建外部线程并启动
27              LocalThread t = new LocalThread(this, mHandler);
28              t.start();
29          }
30
31          //将指定文本输出到可视组件中该方法不能被其他本View之外的线程调用
32          public void addMsg(String msg) {
33              mTxtMsg.append(msg+"\n");
34          }
35      };
```

代码 3-20 中，从第 7 行到第 21 行，定义了一个线程消息队列处理器实例，该实例将会与主线程的消息队列进行绑定。在该实例的定义体中，“handleMessage”方法是通过重载而来，用于处理消息队列中的消息。

2. 线程消息队列的消息读取

每条消息所包含的数据被存放在一个 Bundle 类实例中（第 12 行），Bundle 类实例似于 Map 容器，该容器中的数据项目必须通过指定的键来进行获取（第 14 行中通过“Sender”来获取发送者的信息；第 15 行中通过“Msg”来获取发送信息的内容）。

3. 线程消息队列的消息添加

在代码 3-20 中第 27 行，创建了一个外部线程，并将处理器实例传递给该线程。代码 3-21 是该线程的定义代码。

代码 3-21 访问消息队列的线程的定义

文件名：LocalThread.java

```
1   public class LocalThread extends Thread {
2       //主线程消息队列接口实例
3       private Handler mHandler = null;
4       private HandlerDemoAct mContext = null;
5
6       public LocalThread(HandlerDemoAct handlerDemoAct) {
7           //传递主Activity接口
8           this.mContext = handlerDemoAct;
9       }
10
```

Download at Pin5i.Com

代码 3-21 访问消息队列的线程的定义

文件名：LocalThread.java

```
11          public LocalThread(HandlerDemoAct handlerDemoAct, Handler handler) {
12                //传递主线程消息队列处理器接口和主Activity接口
13                this.mHandler = handler;
14                this.mContext = handlerDemoAct;
15          }
16
17          @Override
18          public void run() {
19                //创建消息传递容器
20                Bundle bundle = new Bundle();
21                //添加数据项
22                bundle.putCharSequence("Sender", mContext.getTitle() );
23                bundle.putString("Msg", "Hi, Paul!");
24                //创建消息实例
25                Message msg = new Message();
26                //设置消息数据
27                msg.setData(bundle);
28                //发送消息
29                mHandler.sendMessage(msg);
30
31                super.run();
32          }
33    };
```

在外部线程的执行函数“run”中，需要先创建一个 Bundle 对象实例（第 20 行），然后将消息内容以“键—值”的形式添加到该实例中（第 22 行和第 23 行）。再创建一个消息对象实例（第 25 行），然后将前面已经填充好的 Bundle 实例设置为该消息实例的数据内容（第 27 行）。最后通过处理器将消息发送到主线程的消息队列中（第 29 行）。

3.3.4 组件与服务间的交互机制

在 3.2.2 小节中，作者对服务组件的使用方式进行了简要地说明，并列举了如何在 Activity 组件中启动或关闭服务组件。但这种应用方式存在一定的问题，对服务组件的控制只有开始和结束两种，如果还有其他的控制则无法做到。例如：作为一个播放音乐文件的服务组件，除了开始和结束控制，还应该有播放下一首或者播放上一首的功能。

1. AIDL IPC 机制

在 Android 平台，提供了一套称为 AIDL IPC 的机制，可以解决客户端组件与服务组件的接口访问。AIDL（Android Interface Definition Language，Android 接口定义语言）是经过扩展，适应 Android 平台的一种对接口定义语言（IDL），通过 AIDL ADT 插件可以自动生成对应的 Java 代码。

AIDL IPC 机制是一种基于接口、轻量级的，类似于 COM 或 Croba 机制。服务组件通过 Android 接口定义语言定义其需要向外界提供的接口，客户端可以通过连接到服务组件来获取这些服务接口，从而实现与服务组件进行交互的目的。

Download at Pin5i.Com

提示：Android 平台底层提供了一种轻量级的、用于 RPC（Remote Procedure Calls，远程过程调用）的机制。客户端组件在本地调用接口方法，但是该方法在远程组件中执行，并且将结果返回到客户端。详细的解释请参考 SDK 文档 “Processes and Threads” 章节。实际上，Android 平台的这一机制与 J2SE 平台中的 RMI（Remote Method Invoke，远程方法调用）机制类似，读者可以结合 RMI 机制来了解 Android 平台的 RPC 应用模式。

图 3-13 和图 3-14 分别是是客户端组件与服务组件进行连接和断开的界面，而图 3-15 是客户端组件调用服务组件所提供的接口进行交互的界面。

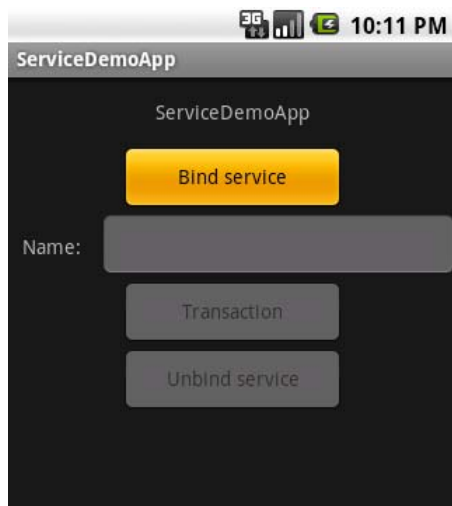

图 3-13 服务连接

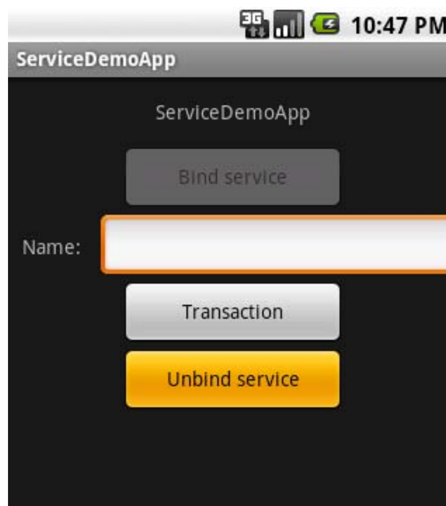

图 3-14 断开服务连接

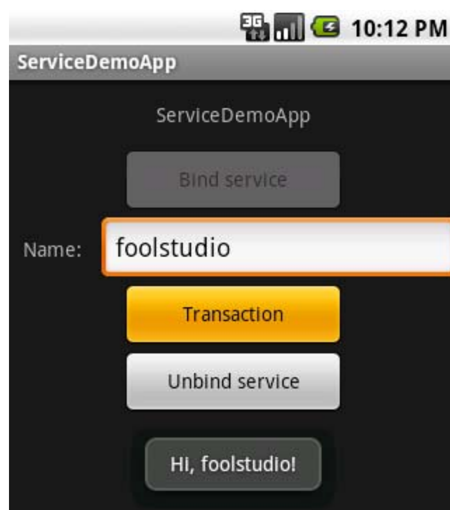

图 3-15 通过服务接口进行通信

代码 3-22 是客户端组件（Activity）的定义代码。

代码 3-22 客户端组件的定义代码

文件名：ServiceDemoAct.java

```
1    public class ServiceDemoAct extends Activity implements OnClickListener {
2        ……
3        //服务接口实例
4        private IEchoService mService = null;
```

Download at Pin5i.Com

代码 3-22 客户端组件的定义代码

文件名：ServiceDemoAct.java

```
5        //服务连接接口实例
6        private ServiceConnection mConnection = new ServiceConnection() {
7            //当连接服务时回调
8            @Override
9            public void onServiceConnected(ComponentName name, IBinder service) {
10               //通过存根获取服务接口实例
11               mService = IEchoService.Stub.asInterface(service);
12           }
13           //当连接断开时回调
14           @Override
15           public void onServiceDisconnected(ComponentName name) {
16               //销毁服务接口实例
17               mService = null;
18           }
19       };
20       ……
21       //连接到服务组件
22       private void doBind() {
23           //连接到指定服务组件（如果不存在则自动创建）
24           bindService(new Intent(EchoService.class.getName() ),
25                           mConnection,
26                           Context.BIND_AUTO_CREATE);
27           setButtons(true);
28       }
29
30       //开始通信
31       private void doTrasaction() {
32           String echo = null;
33
34           try {
35               //通过服务接口调用远程方法
36               echo = mService.getEcho(mTxtName.getText().toString().trim() );
37           } catch (RemoteException e) {
38               // TODO Auto-generated catch block
39               e.printStackTrace();
40           }
41           //本地显示远程服务返回的结果
42           Toast.makeText(this, echo, Toast.LENGTH_LONG).show();
43       }
44
45       //断开连接
46       private void doUnbind() {
```

Download at Pin5i.Com

代码 3-22 客户端组件的定义代码

文件名：ServiceDemoAct.java

```
47              //断开与远程服务的连接
48              unbindService(mConnection);
49              setButtons(false);
50          }
51          ……
52      };
```

代码 3-22 中，“ServiceConnection”成员用来管理客户端组件与服务组件的连接（第 6 行），在其定义体中重载了“onServiceConnected”（第 9 行）和“onServiceDisconnected”方法（第 15 行），用于连接服务和断开连接的回调。

在“onServiceConnected”方法中，通过服务接口存根的“asInterface”方法获取到“代表”服务实例的接口实例（第 11 行），通过该接口实例就可以在客户端组件（Activity）中使用服务组件所“暴露”的方法（第 36 行就是通过服务接口实例“mService”来调用服务组件所提供的“getEcho”方法）。

提示：有关存根（Stub）的定义，在 J2SE 平台的 RMI 机制中也有说明。存根是一种中间接口，用于客户端和服务端的通信数据解释。这就必须要求客户端和服务端的数据内容是可以通过“包裹”来传递的，即这些数据类都必须实现 Parcelable 接口。

在 J2SE RMI 机制中，远程接口定义中返回值的类型必须为支持序列化（Serializable）的实体类，用户自定义类型如果需要通过 RMI 机制进行传递，那么该类也必须实现 Serializable 接口。

从通信的角度而言，无论是 Parcelable 还是 Serialization 接口，都是视为一种协议规范，通过这种协议规范，客户端才能与远程服务端进行信息交换，如此而已。

Android 平台提供了 Activity 组件与服务组件进行连接或断开的接口，在代码 3-22 第 24 行，通过“bindService”方法将服务连接接口（mConnection）与服务组件进行绑定，其中的“Context.BIND_AUTO_CREATE”标志（第 26 行）用于指示自动创建服务组件并绑定。

在第 48 行，通过“unbindService”方法来断开与服务组件的连接。

2. 服务接口的定义

代码 3-23 使用 AIDL 语言定义的服务组件的接口，该接口中只定义一个方法（“getEcho”），该方法用于获取对指定呼叫者的“回话”。

代码 3-23 回声服务接口定义

文件名：IEchoService.aidl

```
1   package foolstudio.demo.service;
2
3   interface IEchoService {
4       String getEcho(String call);
5   }
```

ADT 插件会自动解释 aidl 文件，并生成一个接口定义的 Java 文件（在工程文件结构的“gen”目录中，如图 3-15 所示的“IEchoService.java”文件）。

Download at Pin5i.Com

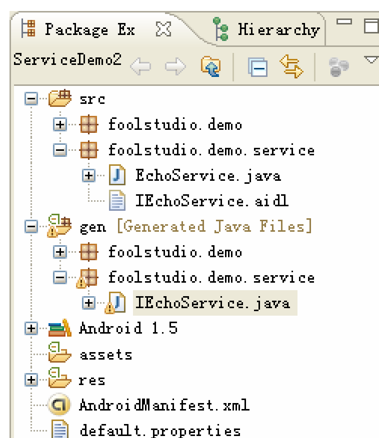

图 3-15 aidl 文件与自动生成文件路径

对于 AIDL 的详细语法，请参考有关 IDL 资料和 Android SDK 文档“Designing a Remote Interface Using AIDL”章节。

3. 服务组件的定义

服务接口定义好了，还要由服务组件来实施才行，代码 3-24 是服务组件的定义。

代码 3-24 远程服务组件的定义

文件名：EchoService.java

```
1   public class EchoService extends Service {
2       //服务接口存根 对象实例
3       private final IEchoService.Stub mBinder = new IEchoService.Stub() {
4           @Override
5           public String getEcho(String call) throws RemoteException {
6               //返回结果给客户端
7               return ("Hi, " + call + "!");
8           }
9       };
10
11      //存在服务连接时回调
12      @Override
13      public IBinder onBind(Intent intent) {
14          //判断请求的客户端是否为预期对象，如果是则返回服务接口存根对象实例
15          if(EchoService.class.getName().equals(intent.getAction())) {
16              return (mBinder);
17          }
18
19          return null;
20      }
21  };
```

在服务组件定义体中，首先定义了一个服务接口存根（Stub）（第 3 行），该存根类的定义在由 aidl 文件自动生成的接口文件定义中。该存根实例用于服务组件与客户端进行通信，将服务组件的输出内容通过协议规定的形式发送给客户端存根对象（第 7 行）。

服务组件的“onBind”方法（第 13 行）重载于系统服务组件框架（第 1 行），用于向客户端提供服务接口存根对象。

Download at Pin5i.Com

提示：代码 3-24 中"onBind"方法的返回值是一个"IBinder"实例，IBinder 接口是一个远程对象的基本接口，描述了与远程对象进行交互的基本协议。但是 Android 平台建议不通过直接实现该接口来定义远程对象，而是通过继承 Binder 类。Binder 类是对 IBinder 接口的实现，是远程对象的基类，该类是轻量级远程过程调用（RPC）机制的核心部分。大多数情况下，开发人员通过 aidl 来定义服务接口，然后由 aidl 工具生成对应的 Binder 子类。

4. 程序清单

代码 3-25 是该服务演示程序的清单文件内容，该程序包含 1 个 Activity 组件（第 7 行）和 1 个服务组件（第 10 行）。

代码 3-25 回声服务程序的工程清单文件

文件名：AndroidManifest.xml

```
1   <?xml version="1.0" encoding="utf-8"?>
2   <manifest xmlns:android="http://schemas.android.com/apk/res/android"
3           package="foolstudio.demo"
4           android:versionCode="1"
5           android:versionName="1.0">
6       <application android:icon="@drawable/icon" android:label="@string/app_name">
7           <activity android:name=".ServiceDemoAct" android:label="@string/app_name">
8               ……
9           </activity>
10            <service android:name=".service.EchoService" android:label="EchoService">
11                <intent-filter>
12                    <action android:name="foolstudio.demo.service.EchoService" />
13                </intent-filter>
14            </service>
15       </application>
16      <uses-sdk android:minSdkVersion="3" />
17  </manifest>
```

3.4 Android 平台应用程序组件小结

Android 平台中定义了 4 种重要的应用程序组件：Activity（活动）、服务、广播接收器和内容提供者。这些应用程序组件根据其适用场合的不同被作者赋予不同的角色：Activity 组件主要应用于提供用户界面，是非常注重界面表现的"形象大使"；而服务组件主要用于后台业务处理，不习惯表现的"老黄牛"；广播接收器主要用于接收系统或用户程序所发送的广播消息并做出响应，是一个积极的"倾听者"；内容提供者组件主要用于通过约定的方式将本组件的数据共享给外部组件。

通过这 4 种基本组件的组合和集成，开发人员就可以开发出满足各种应用的程序。Android 平台还为 4 种组件之间的过程调用、数据共享提供了一些辅助的机制。通过意图组件桥接机制，可以实现 Activity 组件与 Activity 组件以及 Activity 组件与服务组件之间的数据交互；通过线程消息队列的机制，Activity 组件可以与外部线程进行消息传递；

AIDL IPC 和 RPC 机制，是 Android 平台的核心机制，用于提供对远程对象的访问。Activity 组件可以在本地调用远程服务组件所"暴露"的接口，该方法在远程对象中进行执行，通过 PRC 机制进行过程参数和执行结果的传递。

Download at Pin5i.Com