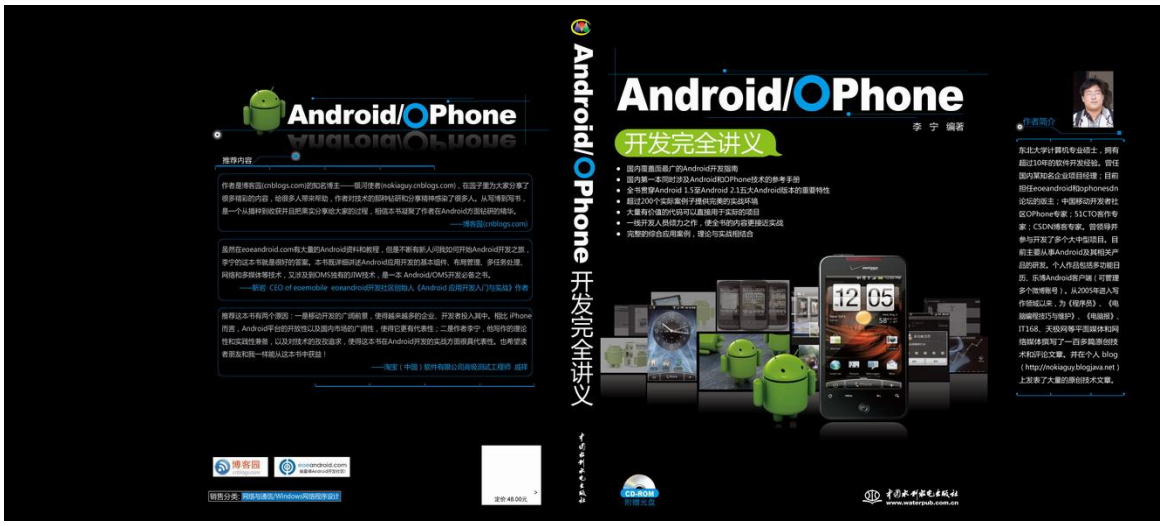
《Android/OPhone 开发完全讲义》样章

李宁　编著

http://nokiaguy.blogjava.net



内容简介

本书近 500 页，共 25 章，分为 5 篇，超过 200 个完整的例子、超过 2 万行代码。

第 1 篇：介绍了 Android 的相关内容，其中包括 Android 的基本概念、搭建 Android 开发环境、Android SDK 中常用命令行工具的使用方法、在 PC 上安装 Android，并测试程序、Android 的学习资源等，还会给出一个简单的 Android 程序，使读者可以了解开发 Android 程序的基本过程。

第 2 篇：主要讲解了 Android 的用户接口，包括 View、定制组件、对话框、Toast 和 Notification、各种菜单、布局；Android 中中的组件（android.widget 包中的组件）；移动存储解决方案，包括 SharedPreferences、 PreferenceActivity、文件存储、XML 存储、Sqlite 数据库及其应用、ContentProvider 等；应用程序之间的通讯，包括跨进程调用 Activity、接收和发送广播；Android 服务详细解，包括 Service 的生命周期，系统服务、时间服务和 AIDL 服务；网络，包括从网络上获得数据装载 Gallery、ListView 等组件。以及从 google 上获得图像，WebView 组件、访问 Http 资源以及 Webservice 等；多媒体技术，包括图形的基本操作、旋转图像、透明度、扭曲和拉伸图像、路径，音频和视频等技术。

第 3 篇：2D 和 OpenGL ES 动画，国际化，Android 中的 14 种资源，访问 Android 手机的硬件，包括通过 USB 驱动在手机上直接调试程序，录音，控制手机的摄像头、传感器（电子罗盘，计步器），GPS 和地图定位，wifi 等。App Widget、在桌面上添加快捷方式，实时文件夹（LiveFolder），NDK,脚本语言（Python、perl 等），手势输入、TTS、蓝牙等。

第 4 篇：介绍了 OPhone 的基础知识，以及 OPhone 与 Android 的差异，OPhone 的 API 扩展，并详细介绍了 JIL Widget 在多媒体，获得系统信息，控制硬件等方面的内容。

第 5 篇：给出了两个综合的例子：万年历，知道当前位置的 GTalk 机器人。

8

Android 服务

服务（Service）是 Android 系统中 4 个应用程序组件之一（其他的组件详见 3.2 节的内容）。服务主要用于两个目的：后台运行和跨进程访问。通过启动一个服务，可以在不显示界面的前提下在后台运行指定的任务，这样可以不影响用户做其他事情。通过 AIDL 服务可以实现不同进程之间的通信，这也是服务的重要用途之一。

本章内容

- Service 的生命周期
- 绑定 Activity 和 Service
- 在 BroadcastReceiver 中启动 Service
- 系统服务
- 时间服务
- 在线程中更新 GUI 组件
- AIDL 服务
- 在 AIDL 服务中传递复杂的数据

8.1　Service 起步

Service 并没有实际界面，而是一直在 Android 系统的后台运行。一般使用 Service 为应用程序提供一些服务，或不需要界面的功能，例如，从 Internet 下载文件、控制 Video 播放器等。本节主要介绍 Service 的启动和结束过程（Service 的生命周期）以及启动 Service 的各种方法。

8.1.1　Service 的生命周期

本节的例子代码所在的工程目录是 src\ch08\ch08_servicelifecycle

Service 与 Activity 一样，也有一个从启动到销毁的过程，但 Service 的这个过程比 Activity 简单得多。Service 启动到销毁的过程只会经历如下 3 个阶段：

- 创建服务
- 开始服务
- 销毁服务

一个服务实际上是一个继承 android.app.Service 的类，当服务经历上面 3 个阶段后，会分别调用 Service 类中的 3 个事件方法进行交互，这 3 个事件方法如下：

```
public void onCreate();                                    //  创建服务
public void onStart(Intent intent, int startId);           //  开始服务
public void onDestroy();                                   //  销毁服务
```

一个服务只会创建一次，销毁一次，但可以开始多次，因此，onCreate 和 onDestroy 方法只会被调用一次，而 onStart 方法会被调用多次。

下面编写一个服务类，具体看一下服务的生命周期由开始到销毁的过程。

```
package net.blogjava.mobile.service;

import android.app.Service;
import android.content.Intent;
import android.os.IBinder;
import android.util.Log;

//  MyService 是一个服务类，该类必须从 android.app.Service 类继承
public class MyService extends Service
{
    @Override
    public IBinder onBind(Intent intent)
    {
        return null;
    }
    //  当服务第 1 次创建时调用该方法
    @Override
    public void onCreate()
    {
        Log.d("MyService", "onCreate");
        super.onCreate();
    }
    //  当服务销毁时调用该方法
    @Override
    public void onDestroy()
    {
        Log.d("MyService", "onDestroy");
        super.onDestroy();
    }
    //  当开始服务时调用该方法
    @Override
    public void onStart(Intent intent, int startId)
    {
        Log.d("MyService", "onStart");
        super.onStart(intent, startId);
    }
}
```

在 MyService 中覆盖了 Service 类中 3 个生命周期方法，并在这些方法中输出了相应的日志信息，以便更容易地观察事件方法的调用情况。

读者在编写 Android 的应用组件时要注意，不管是编写什么组件（例如，Activity、Service 等），都需要在 AndroidManifest.xml 文件中进行配置。MyService 类也不例子。配置这个服务类很简单，只需要在 AndroidManifest.xml 文件的<application>标签中添加如下代码即可：

```
<service android:enabled="true" android:name=".MyService" />
```

其中 android:enabled 属性的值为 true，表示 MyService 服务处于激活状态。虽然目前 MyService 是激活的，但系统仍然不会启动 MyService，要想启动这个服务。必须显式地调用 startService 方法。如果想停止服务，需要显式地调用 stopService 方法，代码如下：

```
public void onClick(View view)
{
    switch (view.getId())
    {
        case R.id.btnStartService:
            startService(serviceIntent);          //  单击【Start Service】按钮启动服务
            break;
        case R.id.btnStopService:
            stopService(serviceIntent);           //  单击【Stop Service】按钮停止服务
            break;
    }
}
```

其中 serviceIntent 是一个 Intent 对象，用于指定 MyService 服务，创建该对象的代码如下：

```
serviceIntent = new Intent(this, MyService.class);
```

运行本节的例子后，会显示如图 8.1 所示的界面。

图 8.1　开始和停止服务

第 1 次单击【Start Service】按钮后，在 DDMS 透视图的 LogCat 视图的 Message 列会输出如下两行信息：

```
onCreate
onStart
```

然后单击【Stop Service】按钮，会在 Message 列中输出如下信息：

```
onDestroy
```

下面按如下的单击按钮顺序的重新测试一下本例。

```
【Start Service】→【Stop Service】→【Start Service】→【Start Service】→【Start Service】→【Stop Service】
```

测试完程序，就会看到如图 8.2 所示的输出信息。可以看出，只在第 1 次单击【Start Service】按钮后会调用 onCreate 方法，如果在未单击【Stop Service】按钮时多次单击【Start Service】按钮，系统只在第 1 次单击【Start Service】按钮时调用 onCreate 和 onStart 方法，再单击该按钮时，系统只会调用 onStart 方法，而不会再次调用 onCreate 方法。

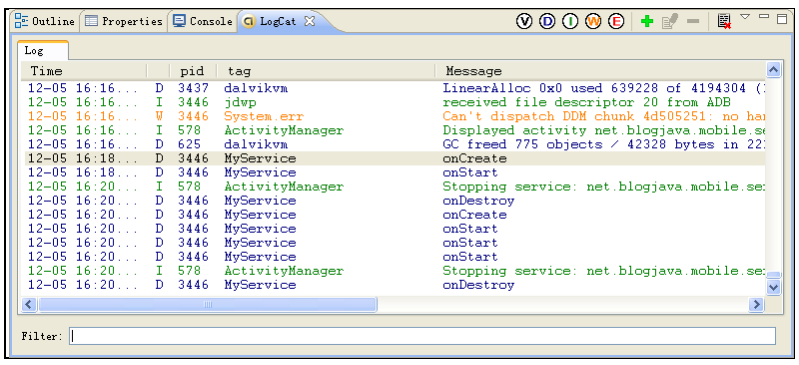


图 8.2　服务的生命周期方法的调用情况

在讨论完服务的生命周期后，再来总结一下创建和开始服务的步骤。创建和开始一个服

务需要如下 3 步：

（1）编写一个服务类，该类必须从 android.app.Service 继承。Service 类涉及到 3 个生命周期方法，但这 3 个方法并不一定在子类中覆盖，读者可根据不同需求来决定使用哪些生命周期方法。在 Service 类中有一个 onBind 方法，该方法是一个抽象方法，在 Service 的子类中必须覆盖。这个方法当 Activity 与 Service 绑定时被调用（将在 8.1.3 节详细介绍）。

（2）在 AndroidManifest.xml 文件中使用<service>标签来配置服务，一般需要将<service>标签的 android:enabled 属性值设为 true，并使用 android:name 属性指定在第 1 步建立的服务类名。

（3）如果要开始一个服务，使用 startService 方法，停止一个服务要使用 stopService 方法。

8.1.2　绑定 Activity 和 Service

本节的例子代码所在的工程目录是 src\ch08\ch08_serviceactivity

如果使用 8.1.1 节介绍的方法启动服务，并且未调用 stopService 来停止服务，这个服务就会随着 Android 系统的启动而启动，随着 Android 系统的关闭而关闭。也就是服务会在 Android 系统启动后一直在后台运行，直到 Android 系统关闭后服务才停止。但有时我们希望在启动服务的 Activity 关闭后服务自动关闭，这就需要将 Activity 和 Service 绑定。

通过 bindService 方法可以将 Activity 和 Service 绑定。bindService 方法的定义如下：

```
public boolean bindService(Intent service, ServiceConnection conn, int flags)
```

该方法的第 1 个参数表示与服务类相关联的 Intent 对象，第 2 个参数是一个 ServiceConnection 类型的变量，负责连接 Intent 对象指定的服务。通过 ServiceConnection 对象可以获得连接成功或失败的状态，并可以获得连接后的服务对象。第 3 个参数是一个标志位，一般设为 Context.BIND_AUTO_CREATE。

下面重新编写 8.1.1 节的 MyService 类，在该类中增加了几个与绑定相关的事件方法。

```
package net.blogjava.mobile.service;

import android.app.Service;
import android.content.Intent;
import android.os.Binder;
import android.os.IBinder;
import android.util.Log;

public class MyService extends Service
{
    private MyBinder myBinder = new MyBinder();
    //  成功绑定后调用该方法
    @Override
    public IBinder onBind(Intent intent)
    {
        Log.d("MyService", "onBind");
        return myBinder;
    }
    //  重新绑定时调用该方法
    @Override
    public void onRebind(Intent intent)
    {
        Log.d("MyService", "onRebind");
        super.onRebind(intent);
    }
    //  解除绑定时调用该方法
    @Override
    public boolean onUnbind(Intent intent)
    {
        Log.d("MyService", "onUnbind");
        return super.onUnbind(intent);
```

```
    }
    @Override
    public void onCreate()
    {
        Log.d("MyService", "onCreate");
        super.onCreate();
    }
    @Override
    public void onDestroy()
    {
        Log.d("MyService", "onDestroy");
        super.onDestroy();
    }
    @Override
    public void onStart(Intent intent, int startId)
    {
        Log.d("MyService", "onStart");
        super.onStart(intent, startId);
    }
    public class MyBinder extends Binder
    {
        MyService getService()
        {
            return MyService.this;
        }
    }
}
```

现在定义一个 MyService 变量和一个 ServiceConnection 变量，代码如下：

```
private MyService myService;
private ServiceConnection serviceConnection = new ServiceConnection()
{
    //  连接服务失败后，该方法被调用
    @Override
    public void onServiceDisconnected(ComponentName name)
    {
        myService = null;
        Toast.makeText(Main.this, "Service Failed.", Toast.LENGTH_LONG).show();
    }
    //  成功连接服务后，该方法被调用。在该方法中可以获得 MyService 对象
    @Override
    public void onServiceConnected(ComponentName name, IBinder service)
    {
        //  获得 MyService 对象
        myService = ((MyService.MyBinder) service).getService();
        Toast.makeText(Main.this, "Service Connected.", Toast.LENGTH_LONG).show();
    }
};
```

最后使用 bindService 方法来绑定 Activity 和 Service，代码如下：

```
bindService(serviceIntent, serviceConnection, Context.BIND_AUTO_CREATE);
```

如果想解除绑定，可以使用下面的代码：

```
unbindService(serviceConnection);
```

在 MyService 类中定义了一个 MyBinder 类，该类实际上是为了获得 MyService 的对象实例的。在 ServiceConnection 接口的 onServiceConnected 方法中的第 2 个参数是一个 IBinder 类型的变量，将该参数转换成 MyService.MyBinder 对象，并使用 MyBinder 类中的 getService 方法获得 MyService 对象。在获得 MyService 对象后，就可以在 Activity 中随意操作 MyService 了。

运行本节的例子后，单击【Bind Service】按钮，如果绑定成

图 8.3　绑定服务

功，会显示如图 8.3 所示的信息提示框。关闭应用程序后，会看到在 LogCat 视图中输出了 onUnbind 和 onDestroy 信息，表明在关闭 Activity 后，服务先被解除绑定，最后被销毁。如果先启动（调用 startService 方法）一个服务，然后再绑定（调用 bindService 方法）服务，会怎么样呢？在这种情况下，虽然服务仍然会成功绑定到 Activity 上，但在 Activity 关闭后，服务虽然会被解除绑定，但并不会被销毁，也就是说，MyService 类的 onDestroy 方法不会被调用。

8.1.3　在 BroadcastReceiver 中启动 Service

本节的例子代码所在的工程目录是 src\ch08\ch08_startupservice

在 8.1.1 节和 8.1.2 节都是先启动了一个 Activity，然后在 Activity 中启动服务。如果是这样，在启动服务时必须要先启动一个 Activity。在很多时候这样做有些多余，阅读完第 7 章的内容，会发现实例 43 可以利用 Broadcast Receiver 在 Android 系统启动时运行一个 Activity。也许我们会从中得到一些启发，既然可以在 Broadcast Receiver 中启动 Activity，为什么不能启动 Service 呢？说做就做，现在让我们来验证一下这个想法。

先编写一个服务类，这个服务类没什么特别的，仍然使用前面两节编写的 MyService 类即可。在 AndroidManifest.xml 文件中配置 MyService 类的代码也相同。

下面来完成最关键的一步，就是建立一个 BroadcastReceiver，代码如下：

```
package net.blogjava.mobile.startupservice;

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

public class StartupReceiver extends BroadcastReceiver
{
    @Override
    public void onReceive(Context context, Intent intent)
    {
        //  启动一个 Service
        Intent serviceIntent = new Intent(context, MyService.class);
        context.startService(serviceIntent);
        Intent activityIntent = new Intent(context, MessageActivity.class);
        //  要想在 Service 中启动 Activity，必须设置如下标志
        activityIntent.setFlags(Intent.FLAG_ACTIVITY_NEW_TASK);
        context.startActivity(activityIntent);
    }
}
```

在 StartupReceiver 类的 onReceive 方法中完成了两项工作：启动服务和显示一个 Activity 来提示服务启动成功。其中 MessageActivity 是一个普通的 Activity 类，只是该类在配置时使用了“@android:style/Theme.Dialog”主题，因此，如果服务启动成功，会显示如图 8.4 所示的信息。

图 8.4　在 BroadcastReceiver 中启动服务

如果安装本例后，在重新启动模拟器后并未出现如图 8.4 所示的信息提示框，最大的可能是没有在 AndroidManifest.xml 文件中配置 BroadcastReceiver 和 Service，下面来看一下 AndroidManifest.xml 文件的完整代码。

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="net.blogjava.mobile.startupservice" android:versionCode="1"
    android:versionName="1.0">
    <application android:icon="@drawable/icon" android:label="@string/app_name">
        <activity android:name=".MessageActivity"    android:theme="@android:style/Theme.Dialog">
            <intent-filter>
                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
        <receiver android:name="StartupReceiver">
            <intent-filter>
                <action android:name="android.intent.action.BOOT_COMPLETED" />
                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </receiver>
        <service android:enabled="true" android:name=".MyService" />
    </application>
    <uses-sdk android:minSdkVersion="3" />
    <uses-permission android:name="android.permission.RECEIVE_BOOT_COMPLETED" />
</manifest>
```

现在运行本例，然后重启一下模拟器，看看 LogCat 视图中是否输出了相应的日志信息。

8.2　系统服务

在 Android 系统中有很多内置的软件，例如，当手机接到来电时，会显示对方的电话号。也可以根据周围的环境将手机设置成震动或静音。如果想把这些功能加到自己的软件中应该怎么办呢？答案就是“系统服务”。在 Android 系统中提供了很多这种服务，通过这些服务，就可以像 Android 系统的内置软件一样随心所欲地控制 Android 系统了。本节将介绍几种常用的系统服务来帮助读者理解和使用这些技术。

8.2.1　获得系统服务

系统服务实际上可以看作是一个对象，通过 Activity 类的 getSystemService 方法可以获得指定的对象（系统服务）。getSystemService 方法只有一个 String 类型的参数，表示系统服务的 ID，这个 ID 在整个 Android 系统中是唯一的。例如，audio 表示音频服务，window 表示窗口服务，notification 表示通知服务。

为了便于记忆和管理，Android SDK 在 android.content.Context 类中定义了这些 ID，例如，下面的代码是一些 ID 的定义。

```
public static final String AUDIO_SERVICE = "audio";                    //  定义音频服务的 ID
public static final String WINDOW_SERVICE = "window";                  //  定义窗口服务的 ID
public static final String NOTIFICATION_SERVICE = "notification";      //  定义通知服务的 ID
```

下面的代码获得了剪贴板服务（android.text.ClipboardManager 对象）。

```
//  获得 ClipboardManager 对象
android.text.ClipboardManager clipboardManager=
            (android.text.ClipboardManager)getSystemService(Context.CLIPBOARD_SERVICE);
clipboardManager.setText("设置剪贴版中的内容");
```

在调用 ClipboardManager.setText 方法设置文本后，在 Android 系统中所有的文本输入框都可以从这个剪贴板对象中获得这段文本，读者不妨自己试一试！

窗口服务（WindowManager 对象）是最常用的系统服务之一，通过这个服务，可以获得很多与窗口相关的信息，例如，窗口的长度和宽度，如下面的代码所示：

```
//  获得 WindowManager 对象
android.view.WindowManager windowManager = (android.view.WindowManager)
                                        getSystemService(Context.WINDOW_SERVICE);
//  在窗口的标题栏输出当前窗口的宽度和高度，例如，320*480
setTitle(String.valueOf(windowManager.getDefaultDisplay().getWidth()) + "*"
                        + String.valueOf(windowManager.getDefaultDisplay().getHeight()));
```

本节简单介绍了如何获得系统服务以及两个常用的系统服务的使用方法，在接下来的实例 47 和实例 48 中将给出两个完整的关于获得和使用系统服务的例子以供读者参考。

实例 47：监听手机来电

工程目录：src\ch08\ch08_phonestate

当来电话时，手机会显示对方的电话号，当接听电话时，会显示当前的通话状态。在这期间存在两个状态：来电状态和接听状态。如果在应用程序中要监听这两个状态，并进行一些其他处理，就需要使用电话服务（TelephonyManager 对象）。

本例通过 TelephonyManager 对象监听来电状态和接听状态，并在相应的状态显示一个 Toast 提示信息框。如果是来电状态，会显示对方的电话号，如果是通话状态，会显示“正在通话...”信息。下面先来看看来电和接听时的效果，如图 8.5 和图 8.6 所示。

图 8.5　来电状态

图 8.6　接听状态

要想获得 TelephonyManager 对象，需要使用 Context.TELEPHONY_SERVICE 常量，代码如下：

```
TelephonyManager tm = (TelephonyManager) getSystemService(Context.TELEPHONY_SERVICE);
MyPhoneCallListener myPhoneCallListener = new MyPhoneCallListener();
//  设置电话状态监听器
tm.listen(myPhoneCallListener, PhoneStateListener.LISTEN_CALL_STATE);
```

其中 MyPhoneCallListener 类是一个电话状态监听器，该类是 PhoneStateListener 的子类，代码如下：

```
public class MyPhoneCallListener extends PhoneStateListener
{
    @Override
    public void onCallStateChanged(int state, String incomingNumber)
    {
        switch (state)
        {
            //  通话状态
            case TelephonyManager.CALL_STATE_OFFHOOK:
                Toast.makeText(Main.this, "正在通话...", Toast.LENGTH_SHORT).show();
                break;
            //  来电状态
            case TelephonyManager.CALL_STATE_RINGING:
                Toast.makeText(Main.this, incomingNumber,Toast.LENGTH_SHORT).show();
                break;
        }
        super.onCallStateChanged(state, incomingNumber);
    }
}
```

如果读者是在模拟器上测试本例，可以使用 DDMS 透视图的【Emulator Control】视图模拟打入电话。进入【Emulator Control】视图，会看到如图 8.7 所示的界面。在【Incoming number】文本框中输入一个电话号，选中【Voice】选项，单击【Call】按钮，这时模拟器就会接到来电。如果已经运行本例，在来电和接听状态就会显示如图 8.5 和图 8.6 所示的 Toast 提示信息。

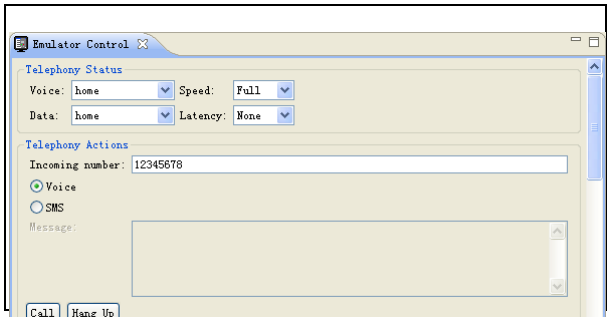


图 8.7　用【Emulator Control】视图模拟拨打电话

实例 48：来电黑名单

工程目录：src\ch08\ch08_phoneblacklist

虽然手机为我们带来了方便，但有时实在不想接听某人的电话，但又不好直接挂断电话，怎么办呢？很简单，如果发现是某人来的电话，直接将手机设成静音，这样就可以不予理睬了。

本例与实例 47 类似，也就是说，仍然需要获得 TelephonyManager 对象，并监听手机的来电状态。为了可以将手机静音，还需要获得一个音频服务（AudioManager 对象）。本例需要修改实例 47 中的手机接听状态方法 onCallStateChanged 中的代码，修改后的结果如下：

```
public class MyPhoneCallListener extends PhoneStateListener
{
    @Override
    public void onCallStateChanged(int state, String incomingNumber)
    {
        //  获得音频服务（AudioManager 对象）
        AudioManager audioManager = (AudioManager) getSystemService(Context.AUDIO_SERVICE);
        switch (state)
        {
            case TelephonyManager.CALL_STATE_IDLE:
                //  在手机空闲状态时，将手机音频设为正常状态
                audioManager.setRingerMode(AudioManager.RINGER_MODE_NORMAL);
                break;
            case TelephonyManager.CALL_STATE_RINGING:
                //  在来电状态时，判断打进来的是否为要静音的电话号，如果是，则静音
                if ("12345678".equals(incomingNumber))
                {
                    //  将电话静音
                    audioManager.setRingerMode(AudioManager.RINGER_MODE_SILENT);
                }
                break;
        }
        super.onCallStateChanged(state, incomingNumber);
    }
}
```

在上面的代码中，只设置了“12345678”为静音电话号，读者可以采用实例 47 的方法使用“12345678”打入电话，再使用其他的电话号打入，看看模拟器是否会响铃。

8.2.2　在模拟器上模拟重力感应

众所周知，Android 系统支持重力感应，通过这种技术，可以利用手机的移动、翻转来实现更为有趣的程序。但遗憾的是，在 Android 模拟器上是无法进行重力感应测试的。既然 Android 系统支持重力感应，但又在模拟器上无法测试，该怎么办呢？别着急，天无绝人之路，有一些第三方的工具可以帮助我们完成这个工作，本节将介绍一种在模拟器上模拟重力感应的工具（sensorsimulator）。这个工具分为服务端和客户端两部分。服务端是一个在 PC 上运行的 Java

Swing GUI 程序，客户端是一个手机程序（apk 文件），在运行时需要通过客户端程序连接到服务端程序上才可以在模拟器上模拟重力感应。

读者可以从下面的地址下载这个工具：

http://code.google.com/p/openintents/downloads/list

进入下载页面后，下载如图 8.8 所示的黑框中的 zip 文件。

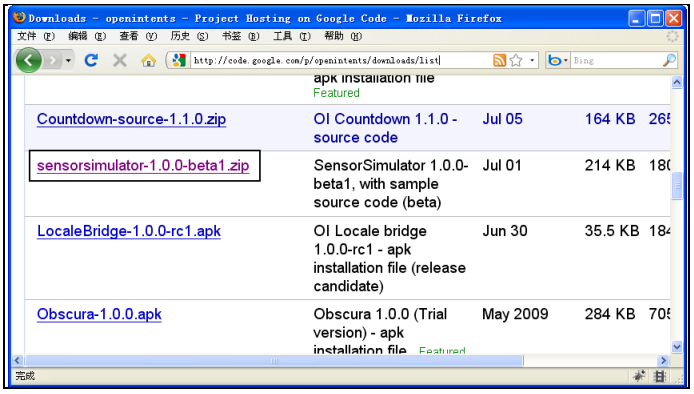


图 8.8　sensorsimulator 下载页面

将 zip 文件解压后，运行 bin 目录中的 sensorsimulator.jar 文件，会显示如图 8.9 所示的界面。界面的左上角是一个模拟手机位置的三维图形，右上角可以通过滑杆来模拟手机的翻转、移动等操作。

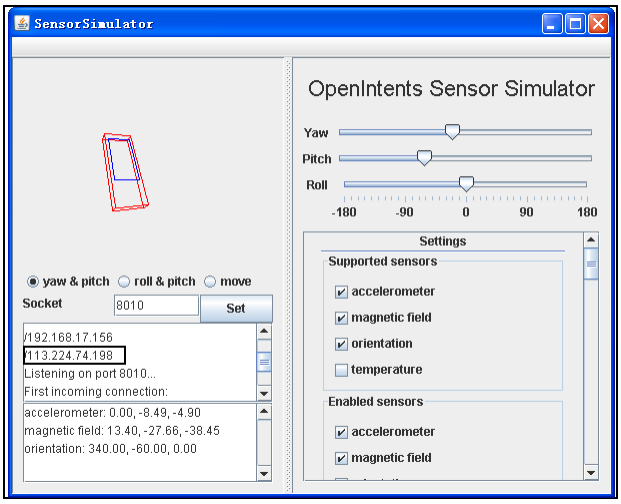


图 8.9　sensorsimulator 主界面

下面来安装客户端程序，先启动 Android 模拟器，然后使用下面的命令安装 bin 目录中的 SensorSimulatorSettings.apk 文件。

```
adb install SensorSimulatorSettings.apk
```

如果安装成功，会在模拟器中看到如图 8.10 所示黑框中的图标。运行这个程序，会进入如图 8.11 所示的界面。在 IP 地址中输入如图 8.9 所示黑框中的 IP（注意，每次启动服务端程序时这个 IP 可能不一样，应以每次启动服务端程序时的 IP 为准）。最后进入【Testing】页，单击【Connect】按钮，如果连接成功，会显示如图 8.12 所示的效果。

图 8.10　安装客户端设置软件

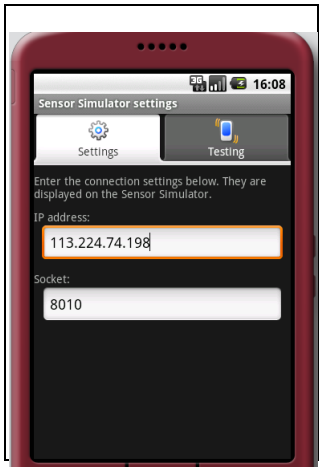


图 8.11　进行客户端设置

下面来测试一下 SensorSimulator 自带的一个 demo，在这个 demo 中输出了通过模拟重力感应获得的数据。

这个 demo 就在 samples 目录中，该目录有一个 SensorDemo 子目录，是一个 Eclipse 工程目录。读者可以直接使用 Eclipse 导入这个目录，并运行程序，如果显示的结果如图 8.13 所示，说明成功使用 SensorSimulator 在 Android 模拟器上模拟了重力感应。

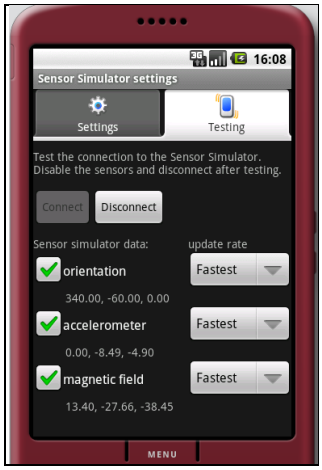


图 8.12　测试连接状态

图 8.13　测试重力感应 demo

在实例 49 中将给出一个完整的例子来演示如何利用重力感应的功能来实现手机翻转静音的效果。

实例 49：手机翻转静音

工程目录：src\ch08\ch08_phonereversal

与手机来电一样，手机翻转状态（重力感应）也由系统服务提供。重力感应服务（android.hardware.SensorManager 对象）可以通过如下代码获得：

```
SensorManager sensorManager = (SensorManager)getSystemService(Context.SENSOR_SERVICE);
```

本例需要在模拟器上模拟重力感应，因此，在本例中使用 SensorSimulator 中的一个类（SensorManagerSimulator）来获得重力感应服务，这个类封装了 SensorManager 对象，并负责与服务端进行通信，监听重力感应事件也需要一个监听器，该监听器需要实现 SensorListener 接口，并通过该接口的 onSensorChanged 事件方法获得重力感应数据。本例完整的代码如下：

```
package net.blogjava.mobile;
```

```
import org.openintents.sensorsimulator.hardware.SensorManagerSimulator;
import android.app.Activity;
import android.content.Context;
import android.hardware.SensorListener;
import android.hardware.SensorManager;
import android.media.AudioManager;
import android.os.Bundle;
import android.widget.TextView;

public class Main extends Activity implements SensorListener
{
    private TextView tvSensorState;
    private SensorManagerSimulator sensorManager;
    @Override
    public void onAccuracyChanged(int sensor, int accuracy)
    {
    }
    @Override
    public void onSensorChanged(int sensor, float[] values)
    {
        switch (sensor)
        {
            case SensorManager.SENSOR_ORIENTATION:
                //  获得声音服务
                AudioManager audioManager = (AudioManager)
                                        getSystemService(Context.AUDIO_SERVICE);
                //  在这里规定翻转角度小于-120 度时静音，values[2]表示翻转角度，也可以设置其他角度
                if (values[2] < -120)
                {
                    audioManager.setRingerMode(AudioManager.RINGER_MODE_SILENT);
                }
                else
                {
                    audioManager.setRingerMode(AudioManager.RINGER_MODE_NORMAL);
                }
                tvSensorState.setText("角度：" + String.valueOf(values[2]));
                break;
        }
    }
    @Override
    protected void onResume()
    {
        //  注册重力感应监听事件
        sensorManager.registerListener(this, SensorManager.SENSOR_ORIENTATION);
        super.onResume();
    }
    @Override
    protected void onStop()
    {
        //  取消对重力感应的监听
        sensorManager.unregisterListener(this);
        super.onStop();
    }
    @Override
    public void onCreate(Bundle savedInstanceState)
    {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);
        //  通过 SensorManagerSimulator 对象获得重力感应服务
        sensorManager = (SensorManagerSimulator) SensorManagerSimulator
                .getSystemService(this, Context.SENSOR_SERVICE);
        //  连接到服务端程序（必须执行下面的代码）
        sensorManager.connectSimulator();
```

```
        }
    }
```

在上面的代码中使用了一个 SensorManagerSimulator 类，该类在 SensorSimulator 工具包带的 sensorsimulator-lib.jar 文件中，可以在 lib 目录中找到这个 jar 文件。在使用 SensorManagerSimulator 类之前，必须在相应的 Eclipse 工程中引用这个 jar 文件。

现在运行本例，并通过服务端主界面右侧的【Roll】滑动杆移动到指定的角度，例如，-74.0 和-142.0，这时设置的角度会显示在屏幕上，如图 8.14 和图 8.15 所示。

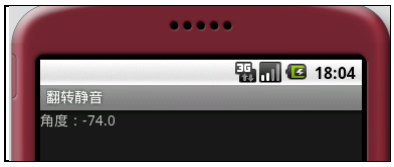


图 8.14　翻转角度大于-120 度

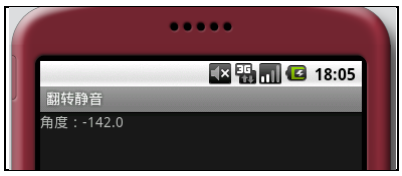


图 8.15　翻转角度小于-120 度

读者可以在如图 8.14 和图 8.15 所示的翻转状态下拨入电话，会发现翻转角度在-74.0 度时来电仍然会响铃，而翻转角度在-142.0 度时就不再响铃了。

由于 SensorSimulator 目前不支持 Android SDK 1.5 及以上版本，因此，只能使用 Android SDK 1.1 中的 SensorListener 接口来监听重力感应事件。在 Android SDK 1.5 及以上版本并不建议继续使用这个接口，代替它的是 android.hardware.SensorEventListener 接口。

8.3　时间服务

在 Android SDK 中提供了多种时间服务。这些时间服务主要处理在一定时间间隔或未来某一时间发生的任务。Android 系统中的时间服务的作用域既可以是应用程序本身，也可以是整个 Android 系统。本节将详细介绍这些时间服务的使用方法，并给出大量的实例供读者学习。

8.3.1　计时器：Chronometer

本节的例子代码所在的工程目录是 src\ch08\ch08_chronometer

Chronometer 是 TextView 的子类，也是一个 Android 组件。这个组件可以用 1 秒的时间间隔进行计时，并显示出计时结果。

Chronometer 类有 3 个重要的方法：start、stop 和 setBase，其中 start 方法表示开始计时；stop 方法表示停止计时；setBase 方法表示重新计时。start 和 stop 方法没有任何参数，setBase 方法有一个参数，表示开始计时的基准时间。如果要从当前时刻重新计时，可以将该参数值设为 SystemClock.elapsedRealtime()。

还可以对 Chronometer 组件做进一步设置。在默认情况下，Chronometer 组件只输出 MM:SS 或 H:MM:SS 的时间格式。例如，当计时到 1 分 20 秒时，Chronometer 组件会显示 01:20。如果想改变显示的信息内容，可以使用 Chronometer 类的 setFormat 方法。该方法需要一个 String 变量，并使用“%s”表示计时信息。例如，使用 setFormat("计时信息：%s")设置显示信息，Chronometer 组件会显示如下计时信息：

```
计时信息：10:20
```

Chronometer 组件还可以通过 onChronometerTick 事件方法来捕捉计时动作。该方法 1 秒调用一次。要想使用 onChronometerTick 事件方法，必须实现如下接口：

```
android.widget.Chronometer.OnChronometerTickListener
```

在本例中有 3 个按钮，分别用来开始、停止和重置计时器，并通过 onChronometerTick 事件

方法显示当前时间，代码如下：

```
package net.blogjava.mobile;

import java.text.SimpleDateFormat;
import java.util.Date;
import android.app.Activity;
import android.os.Bundle;
import android.os.SystemClock;
import android.view.View;
import android.view.View.OnClickListener;
import android.widget.Button;
import android.widget.Chronometer;
import android.widget.TextView;
import android.widget.Chronometer.OnChronometerTickListener;

public class Main extends Activity implements OnClickListener, OnChronometerTickListener
{
    private Chronometer chronometer;
    private TextView tvTime;
    @Override
    public void onClick(View view)
    {
        switch (view.getId())
        {
            case R.id.btnStart:
                //  开始计时器
                chronometer.start();
                break;
            case R.id.btnStop:
                //  停止计时器
                chronometer.stop();
                break;
            case R.id.btnReset
                //  重置计时器:
                chronometer.setBase(SystemClock.elapsedRealtime());
                break;
        }
    }
    @Override
    public void onChronometerTick(Chronometer chronometer)
    {
        SimpleDateFormat sdf = new SimpleDateFormat("HH:mm:ss");
        //  将当前时间显示在 TextView 组件中
        tvTime.setText("当前时间：" + sdf.format(new Date()));
    }
    @Override
    public void onCreate(Bundle savedInstanceState)
    {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);
        tvTime = (TextView)findViewById(R.id.tvTime);
        Button btnStart = (Button) findViewById(R.id.btnStart);
        Button btnStop = (Button) findViewById(R.id.btnStop);
        Button btnReset = (Button) findViewById(R.id.btnReset);
        chronometer = (Chronometer) findViewById(R.id.chronometer);
        btnStart.setOnClickListener(this);
        btnStop.setOnClickListener(this);
        btnReset.setOnClickListener(this);
        //  设置计时监听事件
        chronometer.setOnChronometerTickListener(this);
        //  设置计时信息的格式
        chronometer.setFormat("计时器：%s");
```

```
        }
    }
```

运行本节的例子，并单击【开始】按钮，在按钮下方会显示计时信息，在按钮的上方会显示当前时间，如图 8.16 所示。单击【重置】按钮后，按钮下方的计时信息会从“计时器：00:00”开始显示。

图 8.16　Chronometer 组件的计时效果

8.3.2　预约时间 Handler

本节的例子代码所在的工程目录是 src\ch08\ch08_handler

android.os.Handler 是 Android SDK 中处理定时操作的核心类。通过 Handler 类，可以提交和处理一个 Runnable 对象。这个对象的 run 方法可以立刻执行，也可以在指定时间后执行（也可称为预约执行）。

Handler 类主要可以使用如下 3 个方法来设置执行 Runnable 对象的时间：

```
//  立即执行 Runnable 对象
public final boolean post(Runnable r);
//  在指定的时间（uptimeMillis）执行 Runnable 对象
public final boolean postAtTime(Runnable r, long uptimeMillis);
//  在指定的时间间隔（delayMillis）执行 Runnable 对象
public final boolean postDelayed(Runnable r, long delayMillis);
```

从上面 3 个方法可以看出，第 1 个参数的类型都是 Runnable，因此，在调用这 3 个方法之前，需要有一个实现 Runnable 接口的类，Runnable 接口的代码如下：

```
public interface Runnable
{
    public void run();                  //  线程要执行的方法
}
```

在 Runnable 接口中只有一个 run 方法，该方法为线程执行方法。在本例中 Main 类实现了 Runnable 接口。可以使用如下代码指定在 5 秒后调用 run 方法：

```
Handler handler = new Handler();
handler.postDelayed(this, 5000);
```

如果想在 5 秒内停止计时，可以使用如下代码：

```
handler.removeCallbacks(this);
```

除此之外，还可以使用 postAtTime 方法指定未来的某一个精确时间来执行 Runnable 对象，代码如下：

```
Handler handler = new Handler();
handler.postAtTime(new RunToast(this)
{
}, android.os.SystemClock.uptimeMillis() + 15 * 1000);        //  在 15 秒后执行 Runnable 对象
```

其中 RunToast 是一个实现 Runnable 接口的类，代码如下：

```
class RunToast implements Runnable
{
    private Context context;
```

```
        public RunToast(Context context)
        {
            this.context = context;
        }
        @Override
        public void run()
        {
            Toast.makeText(context, "15 秒后显示 Toast 提示信息", Toast.LENGTH_LONG).show();
        }
    }
```

postAtTime 的第 2 个参数表示一个精确时间的毫秒数，如果从当前时间算起，需要使用 android.os.SystemClock.uptimeMillis()获得基准时间。

要注意的是，不管使用哪个方法来执行 Runnable 对象，都只能运行一次。如果想循环执行，必须在执行完后再次调用 post、postAtTime 或 postDelayed 方法。例如，在 Main 类的 run 方法中再次调用了 postDelayed 方法，代码如下：

```
    public void run()
    {
        tvCount.setText("Count：" + String.valueOf(++count));
        //  再次调用 postDelayed 方法，5 秒后 run 方法仍被调用，然后再一次调用 postDelayed 方法，这样就形成了
        //  循环调用
        handler.postDelayed(this, 5000);
    }
```

运行本例后，单击【开始计数】按钮，5 秒后，会在按钮上方显示计数信息。然后单击【15 秒后显示 Toast 信息框】按钮，过 15 秒后，会显示一个 Toast 信息框，如图 8.17 所示。

图 8.17　使用 Handler 预约时间

8.3.3　定时器 Timer

本节的例子代码所在的工程目录是 src\ch08\ch08_timer

java.util.Timer 与 Chronometer 在功能上有些类似，但 Timer 比 Chronometer 更强大。Timer 除了可以指定循环执行的时间间隔外，还可以设置重复执行和不重复执行。例如，下面的代码设置了在 5 秒后执行。

```
    Timer timer = new Timer();
    timer.schedule(new TimerTask()
    {
        @Override
        public void run()
        {
        }
    }, 5000);                    //  最后 1 个参数表示运行的时间间隔
```

下面的代码设置了每 2 秒执行一次。

```
Timer timer = new Timer();
timer.schedule(new TimerTask()
{
    @Override
    public void run()
    {
    }
}, 0, 2000);        //  第 2 个参数表示延迟执行的时间（这里是 0，表示立即执行），
                    //  最后 1 个参数表示重复执行的时间间隔
```

从上面的代码可以看出，Timer 类通过 schedule 方法设置执行方式和时间。schedule 方法的第 1 个参数的类型是 TimerTask，TimerTask 类实现了 Runnable 接口，因此，Timer 实际上是在线程中执行 run 方法。

虽然 Timer 和 Handler 的任务执行代码都放在 run 方法中，但 Timer 是在线程中执行 run 方法的，而 Handler 将执行的动作添加到 Android 系统的消息队列中。因此，使用 Timer 执行 run 方法时，在 run 方法中不能直接更新 GUI 组件，也就是说，下面的代码是错误的。

```
public void run()
{
    textview.setText("字符串");        // 无法成功设置 TextView 中的文本
}
```

要想在 run 方法中更新 GUI 组件，仍然需要依靠 Handler 类，代码如下：

```
private Handler handler = new Handler()
{
    public void handleMessage(Message msg)
    {
        switch (msg.what)
        {
            case 1:
                //  必须在这里更新进度条组件
                int currentProgress = progressBar.getProgress() + 2;
                if (currentProgress > progressBar.getMax())
                    currentProgress = 0;
                progressBar.setProgress(currentProgress);
                break;
        }
        super.handleMessage(msg);
    }
};
private TimerTask timerTask = new TimerTask()
{
    public void run()
    {
        //  在 run 方法中需要使用 sendMessage 方法发送一条消息
        Message message = new Message();
        message.what = 1;
        handler.sendMessage(message);          //  将任务发送到消息队列
    }
};
```

从上面的代码可以看出，在 run 方法中并没有直接更新进度条组件，而是使用 Handler 类的 sendMessage 方法发送一条消息，并在 Handler 类的 handleMessage 方法中更新进度条组件。实际上，这个 Handler 对象目前已经被加到 Android 系统的消息队列中，正等待 Android 系统的调用。使用下面的代码就可以启动 Timer 定时器，并在每 0.5 秒更新一次进度条组件。

```
Timer timer = new Timer();
timer.schedule(timerTask, 0, 500);
```

运行本节的例子后，就会看到屏幕上进度条的进度在不断变化，如图 8.18 所示。

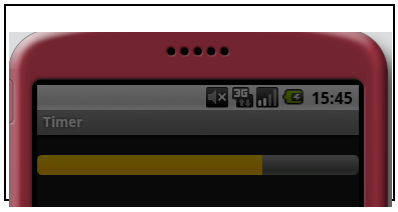


图 8.18　Timer 的定时任务

8.3.4　在线程中更新 GUI 组件

本节的例子代码所在的工程目录是 src\ch08\ch08_thread

除了前面介绍的时间服务可以执行定时任务外，也可以采用线程的方式在后台执行任务。在 Android 系统中创建和启动线程的方法与传统的 Java 程序相同，首先要创建一个 Thread 对象，然后使用 Thread 类的 start 方法开始一个线程。线程在启动后，就会执行 Runnable 接口的 run 方法。

本例中启动了两个线程，分别用来更新两个进度条组件。在 8.3.3 节曾介绍过，在线程中更新 GUI 组件需要使用 Handler 类，当然，直接利用线程作为后台服务也不例外。下面先来看看本例的完整源代码。

```
package net.blogjava.mobile;

import android.app.Activity;
import android.os.Bundle;
import android.os.Handler;
import android.widget.ProgressBar;

public class Main extends Activity
{
    private ProgressBar progressBar1;
    private ProgressBar progressBar2;
    private Handler handler = new Handler();
    private int count1 = 0;
    private int count2 = 0;
    private Runnable doUpdateProgressBar1 = new Runnable()
    {
        @Override
        public void run()
        {
            for (count1 = 0; count1 <= progressBar1.getMax(); count1++)
            {
                //  使用 post 方法立即执行 Runnable 接口的 run 方法
                handler.post(new Runnable()
                {
                    @Override
                    public void run()
                    {
                        progressBar1.setProgress(count1);
                    }
                });
            }
        }
    };
    private Runnable doUpdateProgressBar2 = new Runnable()
    {
        @Override
        public void run()
        {
            for (count2 = 0; count2 <= progressBar2.getMax(); count2++)
```

```
                {
                    //  使用 post 方法立即执行 Runnable 接口的 run 方法
                    handler.post(new Runnable()
                    {
                        @Override
                        public void run()
                        {
                            progressBar2.setProgress(count2);
                        }
                    });
                }
            }
        };
        @Override
        public void onCreate(Bundle savedInstanceState)
        {
            super.onCreate(savedInstanceState);
            setContentView(R.layout.main);
            progressBar1 = (ProgressBar) findViewById(R.id.progressbar1);
            progressBar2 = (ProgressBar) findViewById(R.id.progressbar2);
            Thread thread1 = new Thread(doUpdateProgressBar1, "thread1");
            //  启动第 1 个线程
            thread1.start();
            Thread thread2 = new Thread(doUpdateProgressBar2, "thread2");
            //  启动第 2 个线程
            thread2.start();
        }
    }
```

在编写上面代码时要注意一点，使用 Handler 类时既可以使用 sendMessage 方法发送消息来调用 handleMessage 方法处理任务（见 8.3.3 节的介绍），也可以直接使用 post、postAtTime 或 postDelayed 方法来处理任务。本例中为了方便，直接调用了 post 方法立即执行 run 方法来更新进度条组件。

运行本例后，会看到屏幕上有两个进度条的进度在不断变化，如图 8.19 所示。

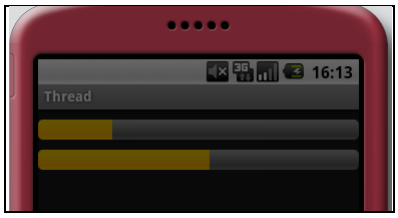


图 8.19　在线程中更新进度条组件

8.3.5　全局定时器 AlarmManager

本节的例子代码所在的工程目录是 src\ch08\ch08_alarm

前面介绍的时间服务的作用域都是应用程序，也就是说，将当前的应用程序关闭后，时间服务就会停止。但在很多时候，需要时间服务不依赖应用程序而存在。也就是说，虽然是应用程序启动的服务，但即使将应用程序关闭，服务仍然可以正常运行。

为了达到服务与应用程序独立的目的，需要获得 AlarmManager 对象。该对象需要通过如下代码获得：

```
AlarmManager alarmManager = (AlarmManager) getSystemService(Context.ALARM_SERVICE);
```

AlarmManager 类的一个非常重要的方法是 setRepeating，通过该方法，可以设置执行时间间隔和相应的动作。setRepeating 方法的定义如下：

```
public void setRepeating(int type, long triggerAtTime, long interval, PendingIntent operation);
```

setRepeating 方法有 4 个参数，这些参数的含义如下：

- type：表示警报类型，一般可以取的值是 AlarmManager.RTC 和 AlarmManager.RTC_WAKEUP。如果将 type 参数值设为 AlarmManager.RTC，表示是一个正常的定时器，如果将 type 参数值设为 AlarmManager.RTC_WAKEUP，除了有定时器的功能外，还会发出警报声（例如，响铃、震动）。
- triggerAtTime：第 1 次运行时要等待的时间，也就是执行延迟时间，单位是毫秒。
- interval：表示执行的时间间隔，单位是毫秒。
- operation：一个 PendingIntent 对象，表示到时间后要执行的操作。PendingIntent 与 Intent 类似，可以封装 Activity、BroadcastReceiver 和 Service。但与 Intent 不同的是，PendingIntent 可以脱离应用程序而存在。

从 setRepeating 方法的 4 个参数可以看出，使用 setRepeating 方法最重要的就是创建 PendingIntent 对象。例如，在下面的代码中用 PendingIntent 指定了一个 Activity。

```
Intent intent = new Intent(this, MyActivity.class);
PendingIntent pendingActivityIntent = PendingIntent.getActivity(this, 0,intent, 0);
```

在创建完 PendingIntent 对象后，就可以使用 setRepeating 方法设置定时器了，代码如下：

```
AlarmManager alarmManager = (AlarmManager) getSystemService(Context.ALARM_SERVICE);
alarmManager.setRepeating(AlarmManager.RTC, 0, 5000, pendingActivityIntent);
```

执行上面的代码，即使应用程序关闭后，每隔 5 秒，系统仍然会显示 MyActivity。如果要取消定时器，可以使用如下代码：

```
alarmManager.cancel(pendingActivityIntent);
```

运行本节的例子，界面如图 8.20 所示。单击【GetActivity】按钮，然后关闭当前应用程序，会发现系统 5 秒后会显示 MyActivity。关闭 MyActivity 后，在 5 秒后仍然会再次显示 MyActivity。

本节只介绍了如何用 PendingIntent 来指定 Activity，读者在实例 50 和实例 51 中将会看到利用 BroadcastReceiver 和 Service 执行定时任务。

实例 50：定时更换壁纸

工程目录：src\ch08\ch08_changewallpaper

使用 AlarmManager 可以实现很多有趣的功能。本例中将实现一个可以定时更换手机壁纸的程序。在编写代码之前，先来看一下如图 8.21 所示的效果。单击【定时更换壁纸】按钮后，手机的壁纸会每隔 5 秒变换一次。

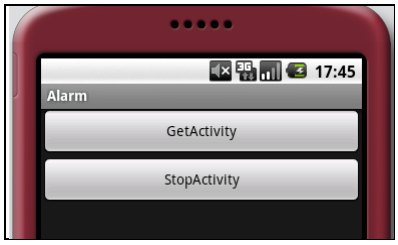


图 8.20　全局定时器（显示 Activity）

图 8.21　定时更换壁纸

本例使用 Service 来完成更换壁纸的工作，下面先编写一个 Service 类，代码如下：

```
package net.blogjava.mobile;
```

```
import java.io.InputStream;
import android.app.Service;
import android.content.Intent;
import android.os.IBinder;

public class ChangeWallpaperService extends Service
{
    private static int index = 0;
    //  保存 res\raw 目录中图像资源的 ID
    private int[] resIds = new int[]{ R.raw.wp1, R.raw.wp2, R.raw.wp3, R.raw.wp4, R.raw.wp5};
    @Override
    public void onStart(Intent intent, int startId)
    {
        if(index == 5)
            index = 0;
        //  获得 res\raw 目录中图像资源的 InputStream 对象
        InputStream inputStream = getResources().openRawResource(resIds[index++]);
        try
        {
            //  更换壁纸
            setWallpaper(inputStream);
        }
        catch (Exception e)
        {
        }
        super.onStart(intent, startId);
    }
    @Override
    public void onCreate()
    {
        super.onCreate();
    }
    @Override
    public IBinder onBind(Intent intent)
    {
        return null;
    }
}
```

在编写 ChangeWallpaperService 类时应注意如下 3 点：

- 为了通过 InputStream 获得图像资源，需要将图像文件放在 res\raw 目录中，而不是 res\drawable 目录中。
- 本例采用了循环更换壁纸的方法。也就是说，共有 5 个图像文件，系统会从第 1 个图像文件开始更换，更换完第 5 个文件后，又从第 1 个文件开始更换。
- 更换壁纸需要使用 Context.setWallpaper 方法，该方法需要一个描述图像的 InputStream 对象。该对象通过 getResources().openRawResource(...)方法获得。

在 AndroidManifest.xml 文件中配置 ChangeWallpaperService 类，代码如下：

```
<service android:name=".ChangeWallpaperService" />
```

最后来看一下本例的主程序（Main 类），代码如下：

```
package net.blogjava.mobile;

import android.app.Activity;
import android.app.AlarmManager;
import android.app.PendingIntent;
import android.content.Context;
import android.content.Intent;
import android.os.Bundle;
import android.view.View;
import android.view.View.OnClickListener;
```

```
import android.widget.Button;

public class Main extends Activity implements OnClickListener
{
    private Button btnStart;
    private Button btnStop;
    @Override
    public void onClick(View view)
    {
        AlarmManager alarmManager = (AlarmManager) getSystemService(Context.ALARM_SERVICE);
        //  指定 ChangeWallpaperService 的 PendingIntent 对象
        PendingIntent pendingIntent = PendingIntent.getService(this, 0,
                new Intent(this, ChangeWallpaperService.class), 0);
        switch (view.getId())
        {
            case R.id.btnStart:
                //  开始每 5 秒更换一次壁纸
                alarmManager.setRepeating(AlarmManager.RTC, 0, 5000, pendingIntent);
                btnStart.setEnabled(false);
                btnStop.setEnabled(true);
                break;
            case R.id.btnStop:
                //  停止更换一次壁纸
                alarmManager.cancel(pendingIntent);
                btnStart.setEnabled(true);
                btnStop.setEnabled(false);
                break;
        }
    }
    @Override
    public void onCreate(Bundle savedInstanceState)
    {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);
        btnStart = (Button) findViewById(R.id.btnStart);
        btnStop = (Button) findViewById(R.id.btnStop);
        btnStop.setEnabled(false);
        btnStart.setOnClickListener(this);
        btnStop.setOnClickListener(this);
    }
}
```

在编写上面代码时应注意如下 3 点：

在创建 PendingIntent 对象时指定了 ChangeWallpaperService.class，这说明这个 PendingIntent 对象与 ChangeWallpaperService 绑定。AlarmManager 在执行任务时会执行 ChangeWallpaperService 类中的 onStart 方法。

不要将任务代码写在 onCreate 方法中，因为 onCreate 方法只会执行一次，一旦服务被创建，该方法就不会被执行了，而 onStart 方法在每次访问服务时都会被调用。

获得指定 Service 的 PendingIntent 对象需要使用 getService 方法。在 8.3.5 节介绍过获得指定 Activity 的 PendingIntent 对象应使用 getActivity 方法。在实例 51 中将介绍使用 getBroadcast 方法获得指定 BroadcastReceiver 的 PendingIntent 对象。

实例 51：多次定时提醒

工程目录：src\ch08\ch08_multialarm

在很多软件中都支持定时提醒功能，也就是说，事先设置未来的某个时间，当到这个时间后，系统会发出声音或进行其他的工作。本例中将实现这个功能。本例不仅可以设置定时提醒功能，而且支持设置多个时间点。运行本例后，单击【添加提醒时间】按钮，会弹出设置时间点的对话

框，如图 8.22 所示。当设置完一系列的时间点后（如图 8.23 所示），如果到了某个时间点，系统就会播放一个声音文件以提醒用户。

图 8.22　设置时间点对话框

图 8.23　设置一系列的时间点

下面先介绍一下定时提醒的原理。在添加时间点后，需要将所添加的时间点保存在文件或数据库中。本例使用 SharedPreferences 来保存时间点，key 和 value 都是时间点。然后使用 AlarmManager 每隔 1 分钟扫描一次，在扫描过程中从文件获得当前时间（时:分）的 value。如果成功获得 value，则说明当前时间为时间点，需要播放声音文件，否则继续扫描。

本例使用 BroadcastReceiver 来处理定时提醒任务。BroadcastReceiver 类的代码如下：

```
package net.blogjava.mobile;

import java.util.Calendar;
import android.app.Activity;
import android.content.BroadcastReceiver;
import android.content.Context;
import android.content.Intent;
import android.content.SharedPreferences;
import android.media.MediaPlayer;

public class AlarmReceiver extends BroadcastReceiver
{
    @Override
    public void onReceive(Context context, Intent intent)
    {
        SharedPreferences sharedPreferences = context.getSharedPreferences(
                "alarm_record", Activity.MODE_PRIVATE);
        String hour = String.valueOf(Calendar.getInstance().get(Calendar.HOUR_OF_DAY));
        String minute = String.valueOf(Calendar.getInstance().get(Calendar.MINUTE));
        //  从 XML 文件中获得描述当前时间点的 value
        String time = sharedPreferences.getString(hour + ":" + minute, null);
        if (time != null)
        {
            //  播放声音
            MediaPlayer mediaPlayer = MediaPlayer.create(context, R.raw.ring);
            mediaPlayer.start();
        }
    }
}
```

配置 AlarmReceiver 类的代码如下：

```
<receiver android:name=".AlarmReceiver" android:enabled="true" />
```

在主程序中每添加一个时间点，就会在 XML 文件中保存所添加的时间点，代码如下：

```
package net.blogjava.mobile;
```

```
import android.app.Activity;
import android.app.AlarmManager;
import android.app.AlertDialog;
import android.app.PendingIntent;
import android.content.Context;
import android.content.DialogInterface;
import android.content.Intent;
import android.content.SharedPreferences;
import android.os.Bundle;
import android.view.View;
import android.view.View.OnClickListener;
import android.widget.Button;
import android.widget.TextView;
import android.widget.TimePicker;

public class Main extends Activity implements OnClickListener
{
    private TextView tvAlarmRecord;
    private SharedPreferences sharedPreferences;
    @Override
    public void onClick(View v)
    {
        View view = getLayoutInflater().inflate(R.layout.alarm, null);
        final TimePicker timePicker = (TimePicker) view.findViewById(R.id.timepicker);
        timePicker.setIs24HourView(true);
        //  显示设置时间点的对话框
        new AlertDialog.Builder(this).setTitle("设置提醒时间").setView(view)
                .setPositiveButton("确定", new DialogInterface.OnClickListener()
                {
                    @Override
                    public void onClick(DialogInterface dialog, int which)
                    {
                        String timeStr = String.valueOf(timePicker
                                .getCurrentHour()) + ":"
                                + String.valueOf(timePicker.getCurrentMinute());
                        //  将时间点添加到 TextView 组件中
                        tvAlarmRecord.setText(tvAlarmRecord.getText().toString() + "\n" + timeStr);
                        //  保存时间点
                        sharedPreferences.edit().putString(timeStr, timeStr).commit();
                    }
                }).setNegativeButton("取消", null).show();
    }
    @Override
    public void onCreate(Bundle savedInstanceState)
    {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);
        Button btnAddAlarm = (Button) findViewById(R.id.btnAddAlarm);
        tvAlarmRecord = (TextView) findViewById(R.id.tvAlarmRecord);
        btnAddAlarm.setOnClickListener(this);
        sharedPreferences = getSharedPreferences("alarm_record",
                Activity.MODE_PRIVATE);
        AlarmManager alarmManager = (AlarmManager) getSystemService(Context.ALARM_SERVICE);
        Intent intent = new Intent(this, AlarmReceiver.class);
        //  创建封装 BroadcastReceiver 的 pendingIntent 对象
        PendingIntent pendingIntent = PendingIntent.getBroadcast(this, 0,intent, 0);
        //  开始定时器，每 1 分钟执行一次
        alarmManager.setRepeating(AlarmManager.RTC, 0, 60 * 1000, pendingIntent);
    }
}
```

在使用本例添加若干个时间点后，会在 alarm_record.xml 文件中看到类似下面的内容：

```
<?xml version='1.0' encoding='utf-8' standalone='yes' ?>
<map>
<string name="18:52">18:52</string>
<string name="20:16">20:16</string>
<string name="19:11">19:11</string>
<string name="19:58">19:58</string>
<string name="22:51">22:51</string>
<string name="22:10">22:10</string>
<string name="22:11">22:11</string>
<string name="20:10">20:10</string>
</map>
```

上面每个<string>元素都是一个时间点，定时器将每隔 1 分钟查一次 alarm_record.xml 文件。

8.4　跨进程访问（AIDL 服务）

Android 系统中的进程之间不能共享内存，因此，需要提供一些机制在不同进程之间进行数据通信。第 7 章介绍的 Activity 和 Broadcast 都可以跨进程通信，除此之外，还可以使用 Content Provider（见 6.6 节的介绍）进行跨进程通信。现在我们已经了解了 4 个 Android 应用程序组件中的 3 个（Activity、Broadcast 和 Content Provider）都可以进行跨进程访问，另外一个 Android 应用程序组件 Service 同样可以。这就是本节要介绍的 AIDL 服务。

8.4.1　什么是 AIDL 服务

本章前面的部分介绍了开发人员如何定制自己的服务，但这些服务并不能被其他的应用程序访问。为了使其他的应用程序也可以访问本应用程序提供的服务，Android 系统采用了远程过程调用（Remote Procedure Call，RPC）方式来实现。与很多其他的基于 RPC 的解决方案一样，Android 使用一种接口定义语言（Interface Definition Language，IDL）来公开服务的接口。因此，可以将这种可以跨进程访问的服务称为 AIDL（Android Interface Definition Language）服务。

8.4.2　建立 AIDL 服务的步骤

建立 AIDL 服务要比建立普通的服务复杂一些，具体步骤如下：

（1）在 Eclipse Android 工程的 Java 包目录中建立一个扩展名为 aidl 的文件。该文件的语法类似于 Java 代码，但会稍有不同。详细介绍见实例 52 的内容。

（2）如果 aidl 文件的内容是正确的，ADT 会自动生成一个 Java 接口文件（*.java）。

（3）建立一个服务类（Service 的子类）。

（4）实现由 aidl 文件生成的 Java 接口。

（5）在 AndroidManifest.xml 文件中配置 AIDL 服务，尤其要注意的是，<action>标签中 android:name 的属性值就是客户端要引用该服务的 ID，也就是 Intent 类的参数值。这一点将在实例 52 和实例 53 中看到。

实例 52：建立 AIDL 服务

AIDL 服务工程目录：src\ch08\ch08_aidl

客户端程序工程目录：src\ch08\ch08_aidlclient

本例中将建立一个简单的 AIDL 服务。这个 AIDL 服务只有一个 getValue 方法，该方法返回一个 String 类型的值。在安装完服务后，会在客户端调用这个 getValue 方法，并将返回值在 TextView 组件中输出。建立这个 AIDL 服务的步骤如下：

（1）建立一个 aidl 文件。在 Java 包目录中建立一个 IMyService.aidl 文件。IMyService.aidl

文件的位置如图 8.24 所示。

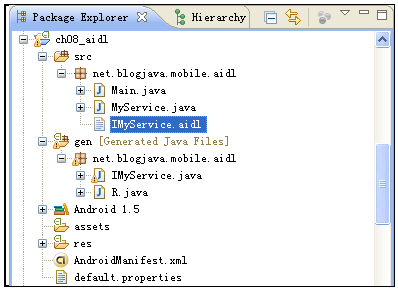


图 8.24　IMyService.aidl 文件的位置

IMyService.aidl 文件的内容如下：

```
package net.blogjava.mobile.aidl;
interface IMyService
{
    String getValue();
}
```

IMyService.aidl 文件的内容与 Java 代码非常相似，但要注意，不能加修饰符（例如，public、private）、AIDL 服务不支持的数据类型（例如，InputStream、OutputStream）等内容。

（2）如果 IMyService.aidl 文件中的内容输入正确，ADT 会自动生成一个 IMyService.java 文件。读者一般并不需要关心这个文件的具体内容，也不需要维护这个文件。关于该文件的具体内容，读者可以查看本节提供的源代码。

（3）编写一个 MyService 类。MyService 是 Service 的子类，在 MyService 类中定义了一个内嵌类（MyServiceImpl），该类是 IMyService.Stub 的子类。MyService 类的代码如下：

```
package net.blogjava.mobile.aidl;

import android.app.Service;
import android.content.Intent;
import android.os.IBinder;
import android.os.RemoteException;

public class MyService extends Service
{
    public class MyServiceImpl extends IMyService.Stub
    {
        @Override
        public String getValue() throws RemoteException
        {
            return "Android/OPhone 开发讲义";
        }
    }
    @Override
    public IBinder onBind(Intent intent)
    {
        return new MyServiceImpl();
    }
}
```

在编写上面代码时要注意如下两点：

- IMyService.Stub 是根据 IMyService.aidl 文件自动生成的，一般并不需要管这个类的内容，只需要编写一个继承于 IMyService.Stub 类的子类（MyServiceImpl 类）即可。
- onBind 方法必须返回 MyServiceImpl 类的对象实例，否则客户端无法获得服务对象。

（4）在 AndroidManifest.xml 文件中配置 MyService 类，代码如下：

```
<service android:name=".MyService" >
    <intent-filter>
```

```
            <action android:name="net.blogjava.mobile.aidl.IMyService" />
        </intent-filter>
    </service>
```

其中“net.blogjava.mobile.aidl.IMyService”是客户端用于访问 AIDL 服务的 ID。

下面来编写客户端的调用代码。首先新建一个 Eclipse Android 工程（ch08_aidlclient），并将自动生成的 IMyService.java 文件连同包目录一起复制到 ch08_aidlclient 工程的 src 目录中，如图 8.25 所示。

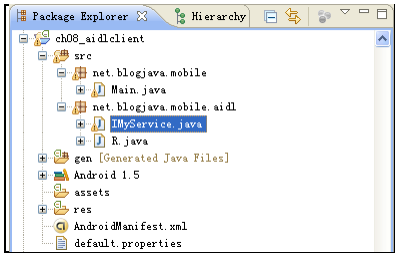


图 8.25　IMyService.java 文件在 ch08_aidlclient 工程中的位置

调用 AIDL 服务首先要绑定服务，然后才能获得服务对象，代码如下：

```
package net.blogjava.mobile;

import net.blogjava.mobile.aidl.IMyService;
import android.app.Activity;
import android.content.ComponentName;
import android.content.Context;
import android.content.Intent;
import android.content.ServiceConnection;
import android.os.Bundle;
import android.os.IBinder;
import android.view.View;
import android.view.View.OnClickListener;
import android.widget.Button;
import android.widget.TextView;

public class Main extends Activity implements OnClickListener
{
    private IMyService myService = null;
    private Button btnInvokeAIDLService;
    private Button btnBindAIDLService;
    private TextView textView;
    private ServiceConnection serviceConnection = new ServiceConnection()
    {
        @Override
        public void onServiceConnected(ComponentName name, IBinder service)
        {
            //  获得服务对象
            myService = IMyService.Stub.asInterface(service);
            btnInvokeAIDLService.setEnabled(true);
        }
        @Override
        public void onServiceDisconnected(ComponentName name)
        {
        }
    };
    @Override
    public void onClick(View view)
    {
        switch (view.getId())
        {
            case R.id.btnBindAIDLService:
```

```
                    //  绑定 AIDL 服务
                    bindService(new Intent("net.blogjava.mobile.aidl.IMyService"),
                            serviceConnection, Context.BIND_AUTO_CREATE);
                    break;
                case R.id.btnInvokeAIDLService:
                    try
                    {
                        textView.setText(myService.getValue());          //  调用服务端的 getValue 方法
                    }
                    catch (Exception e)
                    {
                    }
                    break;
            }
        }
        @Override
        public void onCreate(Bundle savedInstanceState)
        {
            super.onCreate(savedInstanceState);
            setContentView(R.layout.main);
            btnInvokeAIDLService = (Button) findViewById(R.id.btnInvokeAIDLService);
            btnBindAIDLService = (Button) findViewById(R.id.btnBindAIDLService);
            btnInvokeAIDLService.setEnabled(false);
            textView = (TextView) findViewById(R.id.textview);
            btnInvokeAIDLService.setOnClickListener(this);
            btnBindAIDLService.setOnClickListener(this);
        }
    }
```

在编写上面代码时应注意如下两点：

- 使用 bindService 方法来绑定 AIDL 服务。其中需要使用 Intent 对象指定 AIDL 服务的 ID，也就是<action>标签中 android:name 属性的值。
- 在绑定时需要一个 ServiceConnection 对象。创建 ServiceConnection 对象的过程中如果绑定成功，系统会调用 onServiceConnected 方法，通过该方法的 service 参数值可获得 AIDL 服务对象。

首先运行 AIDL 服务程序，然后运行客户端程序，单击【绑定 AIDL 服务】按钮，如果绑定成功，【调用 AIDL 服务】按钮会变为可选状态，单击这个按钮，会输出 getValue 方法的返回值，如图 8.26 所示。

图 8.26　调用 AIDL 服务的客户端程序

实例 53：传递复杂数据的 AIDL 服务

AIDL 服务工程目录：src\ch08\ch08_complextypeaidl

客户端程序工程目录：src\ch08\ch08_complextypeaidlclient

AIDL 服务只支持有限的数据类型，因此，如果用 AIDL 服务传递一些复杂的数据就需要做更一步处理。AIDL 服务支持的数据类型如下：

- Java 的简单类型（int、char、boolean 等）。不需要导入（import）。
- String 和 CharSequence。不需要导入（import）。
- List 和 Map。但要注意，List 和 Map 对象的元素类型必须是 AIDL 服务支持的数据类型。不需要导入（import）。
- AIDL 自动生成的接口。需要导入（import）。
- 实现 android.os.Parcelable 接口的类。需要导入（import）。

其中后两种数据类型需要使用 import 进行导入，将在本章的后面详细介绍。

传递不需要 import 的数据类型的值的方式相同。传递一个需要 import 的数据类型的值（例如，实现 android.os.Parcelable 接口的类）的步骤略显复杂。除了要建立一个实现 android.os.Parcelable 接口的类外，还需要为这个类单独建立一个 aidl 文件，并使用 parcelable 关键字进行定义。具体的实现步骤如下：

（1）建立一个 IMyService.aidl 文件，并输入如下代码：

```
package net.blogjava.mobile.complex.type.aidl;
import net.blogjava.mobile.complex.type.aidl.Product;
interface IMyService
{
    Map getMap(in String country, in Product product);
    Product getProduct();
}
```

在编写上面代码时要注意如下两点：

- Product 是一个实现 android.os.Parcelable 接口的类，需要使用 import 导入这个类。
- 如果方法的类型是非简单类型，例如，String、List 或自定义的类，需要使用 in、out 或 inout 修饰。其中 in 表示这个值被客户端设置；out 表示这个值被服务端设置；inout 表示这个值既被客户端设置，又被服务端设置。

（2）编写 Product 类。该类是用于传递的数据类型，代码如下：

```
package net.blogjava.mobile.complex.type.aidl;

import android.os.Parcel;
import android.os.Parcelable;

public class Product implements Parcelable
{
    private int id;
    private String name;
    private float price;
    public static final Parcelable.Creator<Product> CREATOR = new Parcelable.Creator<Product>()
    {
        public Product createFromParcel(Parcel in)
        {
            return new Product(in);
        }

        public Product[] newArray(int size)
        {
            return new Product[size];
        }
    };
    public Product()
    {
    }
    private Product(Parcel in)
    {
        readFromParcel(in);
    }
```

```
    @Override
    public int describeContents()
    {
        return 0;
    }
    public void readFromParcel(Parcel in)
    {
        id = in.readInt();
        name = in.readString();
        price = in.readFloat();
    }
    @Override
    public void writeToParcel(Parcel dest, int flags)
    {
        dest.writeInt(id);
        dest.writeString(name);
        dest.writeFloat(price);
    }
    //  此处省略了属性的 getter 和 setter 方法
    ... ...
}
```

在编写 Product 类时应注意如下 3 点：

- Product 类必须实现 android.os.Parcelable 接口。该接口用于序列化对象。在 Android 中之所以使用 Pacelable 接口序列化，而不是 java.io.Serializable 接口，是因为 Google 在开发 Android 时发现 Serializable 序列化的效率并不高，因此，特意提供了一个 Parcelable 接口来序列化对象。
- 在 Product 类中必须有一个静态常量，常量名必须是 CREATOR，而且 CREATOR 常量的数据类型必须是 Parcelable.Creator。
- 在 writeToParcel 方法中需要将要序列化的值写入 Parcel 对象。

（3）建立一个 Product.aidl 文件，并输入如下内容：

```
parcelable Product;
```

（4）编写一个 MyService 类，代码如下：

```
package net.blogjava.mobile.complex.type.aidl;

import java.util.HashMap;
import java.util.Map;
import android.app.Service;
import android.content.Intent;
import android.os.IBinder;
import android.os.RemoteException;
//  AIDL 服务类
public class MyService extends Service
{
    public class MyServiceImpl extends IMyService.Stub
    {
        @Override
        public Product getProduct() throws RemoteException
        {
            Product product = new Product();
            product.setId(1234);
            product.setName("汽车");
            product.setPrice(31000);
            return product;
        }
        @Override
        public Map getMap(String country, Product product) throws RemoteException
        {
            Map map = new HashMap<String, String>();
```

```
                map.put("country", country);
                map.put("id", product.getId());
                map.put("name", product.getName());
                map.put("price", product.getPrice());
                map.put("product", product);
                return map;
            }
        }
        @Override
        public IBinder onBind(Intent intent)
        {
            return new MyServiceImpl();
        }
    }
```

（5）在 AndroidManifest.xml 文件中配置 MyService 类，代码如下：

```
<service android:name=".MyService" >
    <intent-filter>
        <action android:name="net.blogjava.mobile.complex.type.aidl.IMyService" />
    </intent-filter>
</service>
```

在客户端调用 AIDL 服务的方法与实例 52 介绍的方法相同，首先将 IMyService.java 和 Product.java 文件复制到客户端工程（ch08_complextypeaidlclient），然后绑定 AIDL 服务，并获得 AIDL 服务对象，最后调用 AIDL 服务中的方法。完整的客户端代码如下：

```
package net.blogjava.mobile;

import net.blogjava.mobile.complex.type.aidl.IMyService;
import android.app.Activity;
import android.content.ComponentName;
import android.content.Context;
import android.content.Intent;
import android.content.ServiceConnection;
import android.os.Bundle;
import android.os.IBinder;
import android.view.View;
import android.view.View.OnClickListener;
import android.widget.Button;
import android.widget.TextView;

public class Main extends Activity implements OnClickListener
{
    private IMyService myService = null;
    private Button btnInvokeAIDLService;
    private Button btnBindAIDLService;
    private TextView textView;
    private ServiceConnection serviceConnection = new ServiceConnection()
    {
        @Override
        public void onServiceConnected(ComponentName name, IBinder service)
        {
            //  获得 AIDL 服务对象
            myService = IMyService.Stub.asInterface(service);
            btnInvokeAIDLService.setEnabled(true);
        }
        @Override
        public void onServiceDisconnected(ComponentName name)
        {
        }
    };
    @Override
    public void onClick(View view)
    {
```

```
            switch (view.getId())
            {
                case R.id.btnBindAIDLService:
                        //  绑定 AIDL 服务
                    bindService(new Intent("net.blogjava.mobile.complex.type.aidl.IMyService"),
                            serviceConnection, Context.BIND_AUTO_CREATE);
                    break;
                case R.id.btnInvokeAIDLService:
                    try
                    {
                        String s = "";
                            //  调用 AIDL 服务中的方法
                        s = "Product.id = " + myService.getProduct().getId() + "\n";
                        s += "Product.name = " + myService.getProduct().getName() + "\n";
                        s += "Product.price = " + myService.getProduct().getPrice() + "\n";
                        s += myService.getMap("China", myService.getProduct()).toString();
                        textView.setText(s);
                    }
                    catch (Exception e)
                    {
                    }
                    break;
            }
        }
        @Override
        public void onCreate(Bundle savedInstanceState)
        {
            super.onCreate(savedInstanceState);
            setContentView(R.layout.main);
            btnInvokeAIDLService = (Button) findViewById(R.id.btnInvokeAIDLService);
            btnBindAIDLService = (Button) findViewById(R.id.btnBindAIDLService);
            btnInvokeAIDLService.setEnabled(false);
            textView = (TextView) findViewById(R.id.textview);
            btnInvokeAIDLService.setOnClickListener(this);
            btnBindAIDLService.setOnClickListener(this);
        }
    }
```

首先运行服务端程序，然后运行客户端程序，单击【绑定 AIDL 服务】按钮，待成功绑定后，单击【调用 AIDL 服务】按钮，会输出如图 8.27 所示的内容。

图 8.27　调用传递复杂数据的 AIDL 服务

8.5　本章小结

本章主要介绍了 Android 系统中的服务（Service）技术。Service 是 Android 中 4 个应用程序组件之一。在 Android 系统内部提供了很多的系统服务，通过这些系统服务，可以实现更为复杂

的功能，例如，监听来电、重力感应等。Android 系统还允许开发人员自定义服务。自定义的服务可以用来在后台运行程序，也可以通过 AIDL 服务提供给其他的应用使用。除此之外，在 Android 系统中还有很多专用于时间的服务和组件，例如，Chronometer、Timer、Handler、AlarmManager 等。通过这些服务，可以完成关于时间的定时、预约等操作。