第 9 章　后台默默的劳动者，探究服务

记得在几年前，iPhone 属于少数人才拥有的稀有物品，Android 甚至还没面世，那个时候全球的手机市场是由诺基亚统治着的。当时我觉得诺基亚的 Symbian 操作系统做得特别出色，因为比起一般的手机，它可以支持后台功能。那个时候能够一边打着电话、听着音乐，一边在后台挂着 QQ 是件非常酷的事情。所以我也曾经单纯地认为，支持后台的手机就是智能手机。

而如今，Symbian 已经风光不再，Android 和 iOS 占据了大部分的智能市场份额，Windows Phone 也占据了一部分，目前已是三分天下的局面。在这三大智能手机操作系统中，iOS 是不支持后台的，当应用程序不在前台运行时就会进入到挂起状态。Android 则是沿用了 Symbian 的老习惯，加入了后台功能，这使得应用程序即使在关闭的情况下仍然可以在后台继续运行。而 Windows Phone 则是经历了一个由不支持到支持后台的过程，目前 Windows Phone 8 系统也是具备后台功能的。这里我们不会花时间去辩论到底谁的方案更好，既然 Android 提供了这个功能，而且是一个非常重要的组件，那我们自然要去学习一下它的用法了。

9.1　服务是什么

服务（Service）是 Android 中实现程序后台运行的解决方案，它非常适合用于去执行那些不需要和用户交互而且还要求长期运行的任务。服务的运行不依赖于任何用户界面，即使当程序被切换到后台，或者用户打开了另外一个应用程序，服务仍然能够保持正常运行。

不过需要注意的是，服务并不是运行在一个独立的进程当中的，而是依赖于创建服务时所在的应用程序进程。当某个应用程序进程被杀掉时，所有依赖于该进程的服务也会停止运行。

另外，也不要被服务的后台概念所迷惑，实际上服务并不会自动开启线程，所有的代码都是默认运行在主线程当中的。也就是说，我们需要在服务的内部手动创建子线程，并在这里执行具体的任务，否则就有可能出现主线程被阻塞住的情况。那么本章的第一堂课，我们就先来学习一下关于 Android 多线程编程的知识。

9.2　Android 多线程编程

熟悉 Java 的你，对多线程编程一定不会陌生吧。当我们需要执行一些耗时操作，比如

说发起一条网络请求时，考虑到网速等其他原因，服务器未必会立刻响应我们的请求，如果不将这类操作放在子线程里去运行，就会导致主线程被阻塞住，从而影响用户对软件的正常使用。那么就让我们从线程的基本用法开始学习吧。

9.2.1 线程的基本用法

Android 多线程编程其实并不比 Java 多线程编程特珠，基本都是使用相同的语法。比如说，定义一个线程只需要新建一个类继承自 Thread，然后重写父类的 run()方法，并在里面编写耗时逻辑即可，如下所示：

```
class MyThread extends Thread {

    @Override
    public void run() {
        // 处理具体的逻辑
    }

}
```

那么该如何启动这个线程呢？其实也很简单，只需要 new 出 MyThread 的实例，然后调用它的 start()方法，这样 run()方法中的代码就会在子线程当中运行了，如下所示：

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

当然，使用继承的方式耦合性有点高，更多的时候我们都会选择使用实现 Runnable 接口的方式来定义一个线程，如下所示：

```
class MyThread implements Runnable {

    @Override
    public void run() {
        // 处理具体的逻辑
    }

}
```

如果使用了这种写法，启动线程的方法也需要进行相应的改变，如下所示：

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

可以看到，Thread 的构造函数接收一个 Runnable 参数，而我们 new 出的 MyThread 正是一个实现了 Runnable 接口的对象，所以可以直接将它传入到 Thread 的构造函数里。接着调

用 Thread 的 start()方法，run()方法中的代码就会在子线程当中运行了。

当然，如果你不想专门再定义一个类去实现 Runnable 接口，也可以使用匿名类的方式，这种写法更为常见，如下所示：

```
new Thread(new Runnable() {

    @Override
    public void run() {
        // 处理具体的逻辑
    }

}).start();
```

以上几种线程的使用方式相信你都不会感到陌生，因为在 Java 中创建和启动线程也是使用同样的方式。了解了线程的基本用法后，下面我们来看一下 Android 多线程编程与 Java 多线程编程不同的地方。

9.2.2　在子线程中更新 UI

和许多其他的 GUI 库一样，Android 的 UI 也是线程不安全的。也就是说，如果想要更新应用程序里的 UI 元素，则必须在主线程中进行，否则就会出现异常。

眼见为实，让我们通过一个具体的例子来验证一下吧。新建一个 AndroidThreadTest 项目，然后修改 activity_main.xml 中的代码，如下所示：

```
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent" >

    <Button
        android:id="@+id/change_text"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="Change Text" />

    <TextView
        android:id="@+id/text"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_centerInParent="true"
        android:text="Hello world"
```

```
        android:textSize="20sp" />

</RelativeLayout>
```

布局文件中定义了两个控件，TextView 用于在屏幕的正中央显示一个 Hello world 字符串，Button 用于改变 TextView 中显示的内容，我们希望在点击 Button 后可以把 TextView 中显示的字符串改成 Nice to meet you。

接下来修改 MainActivity 中的代码，如下所示：

```
public class MainActivity extends Activity implements OnClickListener {

    private TextView text;

    private Button changeText;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        text = (TextView) findViewById(R.id.text);
        changeText = (Button) findViewById(R.id.change_text);
        changeText.setOnClickListener(this);
    }

    @Override
    public void onClick(View v) {
        switch (v.getId()) {
        case R.id.change_text:
            new Thread(new Runnable() {
                @Override
                public void run() {
                    text.setText("Nice to meet you");
                }
            }).start();
            break;
        default:
            break;
        }
    }

}
```

可以看到，我们在 Change Text 按钮的点击事件里面开启了一个子线程，然后在子线程中调用 TextView 的 setText()方法将显示的字符串改成 Nice to meet you。代码的逻辑非常简单，只不过我们是在子线程中更新 UI 的。现在运行一下程序，并点击 Change Text 按钮，你会发现程序果然崩溃了，如图 9.1 所示。

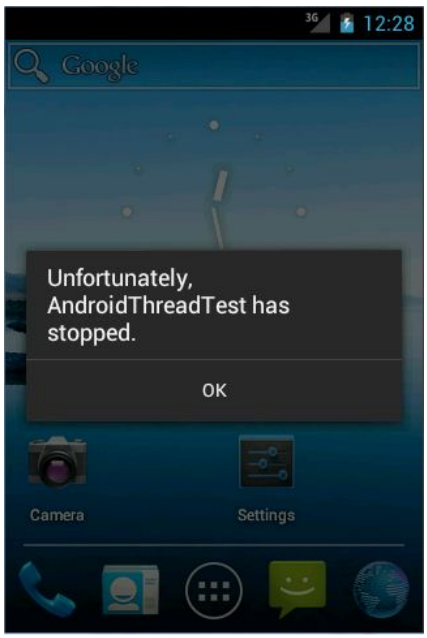

图　9.1

然后观察 LogCat 中的错误日志，可以看出是由于在子线程中更新 UI 所导致的，如图 9.2 所示。

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

图　9.2

由此证实了 Android 确实是不允许在子线程中进行 UI 操作的。但是有些时候，我们必须在子线程里去执行一些耗时任务，然后根据任务的执行结果来更新相应的 UI 控件，这该如何是好呢？

对于这种情况，Android 提供了一套异步消息处理机制，完美地解决了在子线程中进行 UI 操作的问题。本小节中我们先来学习一下异步消息处理的使用方法，下一小节中再去分析它的原理。

修改 MainActivity 中的代码，如下所示：

```
public class MainActivity extends Activity implements OnClickListener {

    public static final int UPDATE_TEXT = 1;

    private TextView text;

    private Button changeText;

    private Handler handler = new Handler() {

        public void handleMessage(Message msg) {
            switch (msg.what) {
            case UPDATE_TEXT:
                // 在这里可以进行UI操作
                text.setText("Nice to meet you");
                break;
            default:
                break;
            }
        }

    };
    ……
    @Override
    public void onClick(View v) {
        switch (v.getId()) {
        case R.id.change_text:
            new Thread(new Runnable() {
                @Override
                public void run() {
                    Message message = new Message();
                    message.what = UPDATE_TEXT;
                    handler.sendMessage(message); // 将Message对象发送出去
                }
            }).start();
            break;
        default:
            break;
```

```
            }
        }

    }
```

这里我们先是定义了一个整型常量 UPDATE_TEXT，用于表示更新 TextView 这个动作。然后新增一个 Handler 对象，并重写父类的 handleMessage 方法，在这里对具体的 Message 进行处理。如果发现 Message 的 what 字段的值等于 UPDATE_TEXT，就将 TextView 显示的内容改成 Nice to meet you。

下面再来看一下 Change Text 按钮的点击事件中的代码。可以看到，这次我们并没有在子线程里直接进行 UI 操作，而是创建了一个 Message（android.os.Message）对象，并将它的 what 字段的值指定为 UPDATE_TEXT，然后调用 Handler 的 sendMessage()方法将这条 Message 发送出去。很快，Handler 就会收到这条 Message，并在 handleMessage()方法中对它进行处理。注意此时 handleMessage()方法中的代码就是在主线程当中运行的了，所以我们可以放心地在这里进行 UI 操作。接下来对 Message 携带的 what 字段的值进行判断，如果等于 UPDATE_TEXT，就将 TextView 显示的内容改成 Nice to meet you。

现在重新运行程序，可以看到屏幕的正中央显示着 Hello world。然后点击一下 Change Text 按钮，显示的内容着就被替换成 Nice to meet you，如图 9.3 所示。

图　9.3

这样你就已经掌握了 Android 异步消息处理的基本用法，使用这种机制就可以出色地解决掉在子线程中更新 UI 的问题。不过恐怕你对它的工作原理还不是很清楚，下面我们就来分析一下 Android 异步消息处理机制到底是如何工作的。

9.2.3 解析异步消息处理机制

Android 中的异步消息处理主要由四个部分组成，Message、Handler、MessageQueue 和 Looper。其中 Message 和 Handler 在上一小节中我们已经接触过了，而 MessageQueue 和 Looper 对于你来说还是全新的概念，下面我就对这四个部分进行一下简要的介绍。

1. Message

   Message 是在线程之间传递的消息，它可以在内部携带少量的信息，用于在不同线程之间交换数据。上一小节中我们使用到了 Message 的 what 字段，除此之外还可以使用 arg1 和 arg2 字段来携带一些整型数据，使用 obj 字段携带一个 Object 对象。

2. Handler

   Handler 顾名思义也就是处理者的意思，它主要是用于发送和处理消息的。发送消息一般是使用 Handler 的 sendMessage()方法，而发出的消息经过一系列地辗转处理后，最终会传递到 Handler 的 handleMessage()方法中。

3. MessageQueue

   MessageQueue 是消息队列的意思，它主要用于存放所有通过 Handler 发送的消息。这部分消息会一直存在于消息队列中，等待被处理。每个线程中只会有一个 MessageQueue 对象。

4. Looper

   Looper 是每个线程中的 MessageQueue 的管家，调用 Looper 的 loop()方法后，就会进入到一个无限循环当中，然后每当发现 MessageQueue 中存在一条消息，就会将它取出，并传递到 Handler 的 handleMessage()方法中。每个线程中也只会有一个 Looper 对象。

了解了 Message、Handler、MessageQueue 以及 Looper 的基本概念后，我们再来对异步消息处理的整个流程梳理一遍。首先需要在主线程当中创建一个 Handler 对象，并重写 handleMessage()方法。然后当子线程中需要进行 UI 操作时，就创建一个 Message 对象，并通过 Handler 将这条消息发送出去。之后这条消息会被添加到 MessageQueue 的队列中等待被处理，而 Looper 则会一直尝试从 MessageQueue 中取出待处理消息，最后分发回 Handler 的 handleMessage()方法中。由于 Handler 是在主线程中创建的，所以此时 handleMessage()方法中的代码也会在主线程中运行，于是我们在这里就可以安心地进行 UI 操作了。整个异步消息处理机制的流程示意图如图 9.4 所示。

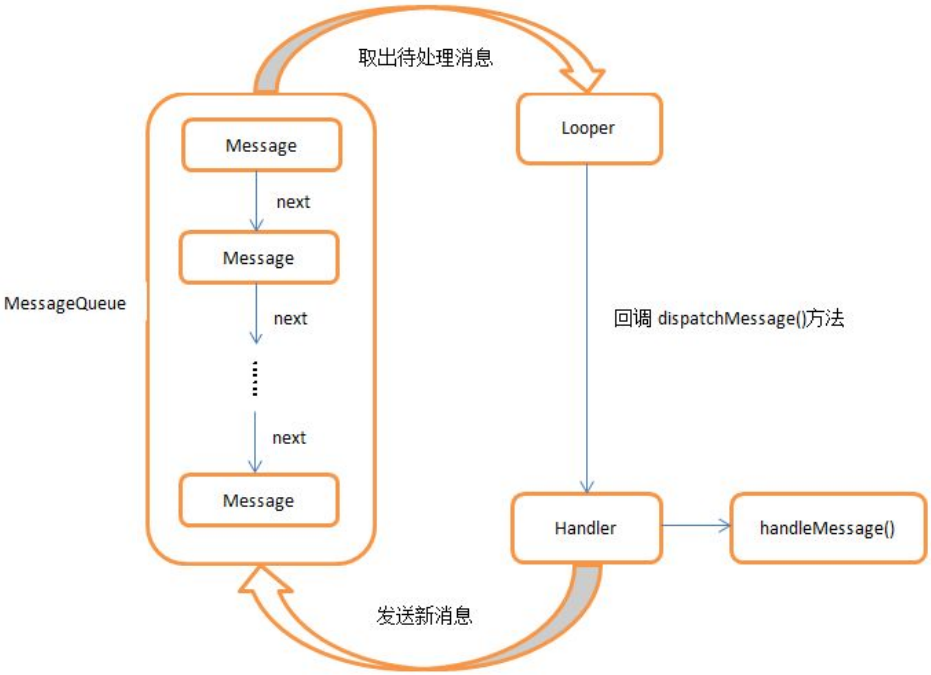

图　9.4

一条 Message 经过这样一个流程的辗转调用后，也就从子线程进入到了主线程，从不能更新 UI 变成了可以更新 UI，整个异步消息处理的核心思想也就是如此。

9.2.4　使用 AsyncTask

不过为了更加方便我们在子线程中对 UI 进行操作，Android 还提供了另外一些好用的工具，AsyncTask 就是其中之一。借助 AsyncTask，即使你对异步消息处理机制完全不了解，也可以十分简单地从子线程切换到主线程。当然，AsyncTask 背后的实现原理也是基于异步消息处理机制的，只是 Android 帮我们做了很好的封装而已。

首先来看一下 AsyncTask 的基本用法，由于 AsyncTask 是一个抽象类，所以如果我们想使用它，就必须要创建一个子类去继承它。在继承时我们可以为 AsyncTask 类指定三个泛型参数，这三个参数的用途如下。

1. Params

   在执行 AsyncTask 时需要传入的参数，可用于在后台任务中使用。

2. Progress

   后台任务执行时，如果需要在界面上显示当前的进度，则使用这里指定的泛型作为进度单位。

3. Result

当任务执行完毕后，如果需要对结果进行返回，则使用这里指定的泛型作为返回值类型。

因此，一个最简单的自定义 AsyncTask 就可以写成如下方式：

```
class DownloadTask extends AsyncTask<Void, Integer, Boolean> {
    ……
}
```

这里我们把 AsyncTask 的第一个泛型参数指定为 Void，表示在执行 AsyncTask 的时候不需要传入参数给后台任务。第二个泛型参数指定为 Integer，表示使用整型数据来作为进度显示单位。第三个泛型参数指定为 Boolean，则表示使用布尔型数据来反馈执行结果。

当然，目前我们自定义的 DownloadTask 还是一个空任务，并不能进行任何实际的操作，我们还需要去重写 AsyncTask 中的几个方法才能完成对任务的定制。经常需要去重写的方法有以下四个。

1. onPreExecute()

这个方法会在后台任务开始执行之前调用，用于进行一些界面上的初始化操作，比如显示一个进度条对话框等。

2. doInBackground(Params...)

这个方法中的所有代码都会在子线程中运行，我们应该在这里去处理所有的耗时任务。任务一旦完成就可以通过 return 语句来将任务的执行结果返回，如果 AsyncTask 的第三个泛型参数指定的是 Void，就可以不返回任务执行结果。注意，在这个方法中是不可以进行 UI 操作的，如果需要更新 UI 元素，比如说反馈当前任务的执行进度，可以调用 publishProgress(Progress...)方法来完成。

3. onProgressUpdate(Progress...)

当在后台任务中调用了 publishProgress(Progress...)方法后，这个方法就会很快被调用，方法中携带的参数就是在后台任务中传递过来的。在这个方法中可以对 UI 进行操作，利用参数中的数值就可以对界面元素进行相应地更新。

4. onPostExecute(Result)

当后台任务执行完毕并通过 return 语句进行返回时，这个方法就很快会被调用。返回的数据会作为参数传递到此方法中，可以利用返回的数据来进行一些 UI 操作，比如说提醒任务执行的结果，以及关闭掉进度条对话框等。

因此，一个比较完整的自定义 AsyncTask 就可以写成如下方式：

```
class DownloadTask extends AsyncTask<Void, Integer, Boolean> {

    @Override
```

```
    protected void onPreExecute() {
        progressDialog.show(); // 显示进度对话框
    }

    @Override
    protected Boolean doInBackground(Void... params) {
        try {
            while (true) {
                int downloadPercent = doDownload(); // 这是一个虚构的方法
                publishProgress(downloadPercent);
                if (downloadPercent >= 100) {
                    break;
                }
            }
        } catch (Exception e) {
            return false;
        }
        return true;
    }

    @Override
    protected void onProgressUpdate(Integer... values) {
        // 在这里更新下载进度
        progressDialog.setMessage("Downloaded " + values[0] + "%");
    }

    @Override
    protected void onPostExecute(Boolean result) {
        progressDialog.dismiss(); // 关闭进度对话框
        // 在这里提示下载结果
        if (result) {
            Toast.makeText(context, "Download succeeded",
Toast.LENGTH_SHORT).show();
        } else {
            Toast.makeText(context, " Download failed",
Toast.LENGTH_SHORT).show();
        }
    }
}
```

在这个 DownloadTask 中，我们在 doInBackground()方法里去执行具体的下载任务。这个方法里的代码都是在子线程中运行的，因而不会影响到主线程的运行。注意这里虚构了一个 doDownload()方法，这个方法用于计算当前的下载进度并返回，我们假设这个方法已经存在了。在得到了当前的下载进度后，下面就该考虑如何把它显示到界面上了，由于 doInBackground()方法是在子线程中运行的，在这里肯定不能进行 UI 操作，所以我们可以调用 publishProgress()方法并将当前的下载进度传进来，这样 onProgressUpdate()方法就会很快被调用，在这里就可以进行 UI 操作了。

当下载完成后，doInBackground()方法会返回一个布尔型变量，这样 onPostExecute()方法就会很快被调用，这个方法也是在主线程中运行的。然后在这里我们会根据下载的结果来弹出相应的 Toast 提示，从而完成整个 DownloadTask 任务。

简单来说，使用 AsyncTask 的诀窍就是，在 doInBackground()方法中去执行具体的耗时任务，在 onProgressUpdate()方法中进行 UI 操作，在 onPostExecute()方法中执行一些任务的收尾工作。

如果想要启动这个任务，只需编写以下代码即可：

```
new DownloadTask().execute();
```

以上就是 AsyncTask 的基本用法，怎么样，是不是感觉简单方便了许多？我们并不需要去考虑什么异步消息处理机制，也不需要专门使用一个 Handler 来发送和接收消息，只需要调用一下 publishProgress()方法就可以轻松地从子线程切换到 UI 线程了。

经验值：+18000　　升级！（由资深鸟升级至头领鸟）　　目前经验值：109905

级别：头领鸟

赢得宝物：战胜百手虫大战神。拾取百手虫大战神掉落的宝物，一把浇花的小水壶、一把小型的园艺铲、一个看起来很普通的魔方，还有一本荣获“震撼三界大奖”的计算机经典图书《设计模式演义》。百手虫大战神姓百手虫，名叫大战。神是他自封的，其实他不是神，只是半神，但神界的人们都很宽容，喜欢让别人 happy，既然人家愿意叫自己神，那为什么不让人家开心一下呢，所以大家也都这么叫他。百手虫大战神虽然不是神，但确实很威武，身长接近 50 米，像巨蟒，但比巨蟒大得多，最粗的地方直径可达 2 米，体重超过 20 吨。百手虫虽然有一百双手，但却只有一个大脑袋，所以它不能同时干一百件复杂的事，但的确可以同时干一百件简单的事情。百手虫大战神热爱花花草草，尤其喜欢绿植，神界每年的植树造林活动都少不了他，因为他一次就能种下 10 棵树。百手虫大战神的本职工作是在神界的神魔合资的软件公司——巨硬公司做程序员，由于体型过于巨大，所以百手虫大战神通常是在自己家里办公，只有需要他面试新小弟时才偶尔来公司。

9.3　服务的基本用法

了解了 Android 多线程编程的技术之后，下面就让我们进入到本章的正题，开始对服务的相关内容进行学习。作为 Android 四大组件之一，服务也少不了有很多非常重要的知识点，那我们自然要从最基本的用法开始学习了。

9.3.1　定义一个服务

首先看一下如何在项目中定义一个服务。新建一个 ServiceTest 项目，然后在这个项目中新增一个名为 MyService 的类，并让它继承自 Service，完成后的代码如下所示：

```
public class MyService extends Service {

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

}
```

目前 MyService 中可以算是空空如也，但有一个 onBind()方法特别醒目。这个方法是 Service 中唯一的一个抽象方法，所以必须要在子类里实现。我们会在后面的小节中使用到 onBind()方法，目前可以暂时将它忽略掉。

既然是定义一个服务，自然应该在服务中去处理一些事情了，那处理事情的逻辑应该写在哪里呢？这时就可以重写 Service 中的另外一些方法了，如下所示：

```
public class MyService extends Service {

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

    @Override
    public void onCreate() {
        super.onCreate();
    }

    @Override
    public int onStartCommand(Intent intent, int flags, int startId) {
```

```
        return super.onStartCommand(intent, flags, startId);
    }

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

}
```

可以看到，这里我们又重写了 onCreate()、onStartCommand()和 onDestroy()这三个方法，它们是每个服务中最常用到的三个方法了。其中 onCreate()方法会在服务创建的时候调用，onStartCommand()方法会在每次服务启动的时候调用，onDestroy()方法会在服务销毁的时候调用。

通常情况下，如果我们希望服务一旦启动就立刻去执行某个动作，就可以将逻辑写在 onStartCommand()方法里。而当服务销毁时，我们又应该在 onDestroy()方法中去回收那些不再使用的资源。

另外需要注意，每一个服务都需要在 AndroidManifest.xml 文件中进行注册才能生效，不知道你有没有发现，这是 Android 四大组件共有的特点。于是我们还应该修改 AndroidManifest.xml 文件，代码如下所示：

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.example.servicetest"
    android:versionCode="1"
    android:versionName="1.0" >
    ……
    <application
        android:allowBackup="true"
        android:icon="@drawable/ic_launcher"
        android:label="@string/app_name"
        android:theme="@style/AppTheme" >
        ……
        <service android:name=".MyService" >
        </service>
    </application>
</manifest>
```

这样的话，就已经将一个服务完全定义好了。

9.3.2　启动和停止服务

定义好了服务之后，接下来就应该考虑如何去启动以及停止这个服务。启动和停止的方法当然你也不会陌生，主要是借助 Intent 来实现的，下面就让我们在 ServiceTest 项目中尝试去启动以及停止 MyService 这个服务。

首先修改 activity_main.xml 中的代码，如下所示：

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical" >

    <Button
        android:id="@+id/start_service"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="Start Service" />

    <Button
        android:id="@+id/stop_service"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="Stop Service" />

</LinearLayout>
```

这里我们在布局文件中加入了两个按钮，分别是用于启动服务和停止服务的。

然后修改 MainActivity 中的代码，如下所示：

```
public class MainActivity extends Activity implements OnClickListener {

    private Button startService;

    private Button stopService;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        startService = (Button) findViewById(R.id.start_service);
        stopService = (Button) findViewById(R.id.stop_service);
```

```
        startService.setOnClickListener(this);
        stopService.setOnClickListener(this);
    }

    @Override
    public void onClick(View v) {
        switch (v.getId()) {
        case R.id.start_service:
            Intent startIntent = new Intent(this, MyService.class);
            startService(startIntent); // 启动服务
            break;
        case R.id.stop_service:
            Intent stopIntent = new Intent(this, MyService.class);
            stopService(stopIntent); // 停止服务
            break;
        default:
            break;
        }
    }

}
```

可以看到，这里在 onCreate()方法中分别获取到了 Start Service 按钮和 Stop Service 按钮的实例，并给它们注册了点击事件。然后在 Start Service 按钮的点击事件里，我们构建出了一个 Intent 对象，并调用 startService()方法来启动 MyService 这个服务。在 Stop Serivce 按钮的点击事件里，我们同样构建出了一个 Intent 对象，并调用 stopService()方法来停止 MyService 这个服务。startService()和 stopService()方法都是定义在 Context 类中的，所以我们在活动里可以直接调用这两个方法。注意，这里完全是由活动来决定服务何时停止的，如果没有点击 Stop Service 按钮，服务就会一直处于运行状态。那服务有没有什么办法让自己停止下来呢？当然可以，只需要在 MyService 的任何一个位置调用 stopSelf()方法就能让这个服务停止下来了。

那么接下来又有一个问题需要思考了，我们如何才能证实服务已经成功启动或者停止了呢？最简单的方法就是在 MyService 的几个方法中加入打印日志，如下所示：

```
public class MyService extends Service {

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

```
    }

    @Override
    public void onCreate() {
        super.onCreate();
        Log.d("MyService", "onCreate executed");
    }

    @Override
    public int onStartCommand(Intent intent, int flags, int startId) {
        Log.d("MyService", "onStartCommand executed");
        return super.onStartCommand(intent, flags, startId);
    }

    @Override
    public void onDestroy() {
        super.onDestroy();
        Log.d("MyService", "onDestroy executed");
    }

}
```

现在可以运行一下程序来进行测试了，程序的主界面如图 9.5 所示。

图　9.5

点击一下 Start Service 按钮，观察 LogCat 中的打印日志如图 9.6 所示。

Tag	Text
MyService	onCreate executed
MyService	onStartCommand executed

图 9.6

MyService 中的 onCreate()和 onStartCommand()方法都执行了，说明这个服务确实已经启动成功了，并且你还可以在正在运行的服务列表中找到它，如图 9.7 所示。

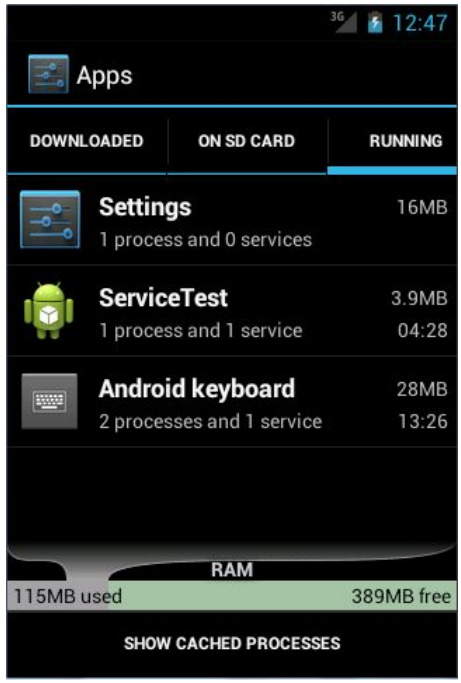

图 9.7

然后再点击一下 Stop Service 按钮，观察 LogCat 中的打印日志如图 9.8 所示。

Tag	Text
MyService	onDestroy executed

图 9.8

由此证明，MyService 确实已经成功停止下来了。

话说回来，虽然我们已经学会了启动服务以及停止服务的方法，不知道你心里现在有没有一个疑惑，那就是 onCreate()方法和 onStartCommand()到底有什么区别呢？因为刚刚点击 Start Service 按钮后两个方法都执行了。

其实 onCreate()方法是在服务第一次创建的时候调用的，而 onStartCommand()方法则在每次启动服务的时候都会调用，由于刚才我们是第一次点击 Start Service 按钮，服务此时还未创建过，所以两个方法都会执行，之后如果你再连续多点击几次 Start Service 按钮，你就会发现只有 onStartCommand()方法可以得到执行了。

9.3.3　活动和服务进行通信

上一小节中我们学习了启动和停止服务的方法，不知道你有没有发现，虽然服务是在活动里启动的，但在启动了服务之后，活动与服务基本就没有什么关系了。确实如此，我们在活动里调用了 startService()方法来启动 MyService 这个服务，然后 MyService 的 onCreate()和 onStartCommand()方法就会得到执行。之后服务会一直处于运行状态，但具体运行的是什么逻辑，活动就控制不了了。这就类似于活动通知了服务一下：“你可以启动了！”然后服务就去忙自己的事情了，但活动并不知道服务到底去做了什么事情，以及完成的如何。

那么有没有什么办法能让活动和服务的关系更紧密一些呢？例如在活动中指挥服务去干什么，服务就去干什么。当然可以，这就需要借助我们刚刚忽略的 onBind()方法了。

比如说目前我们希望在 MyService 里提供一个下载功能，然后在活动中可以决定何时开始下载，以及随时查看下载进度。实现这个功能的思路是创建一个专门的 Binder 对象来对下载功能进行管理，修改 MyService 中的代码，如下所示：

```
public class MyService extends Service {

    private DownloadBinder mBinder = new DownloadBinder();

    class DownloadBinder extends Binder {

        public void startDownload() {
            Log.d("MyService", "startDownload executed");
        }

        public int getProgress() {
            Log.d("MyService", "getProgress executed");
            return 0;
        }

    }
```

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

    ……

}
```

可以看到，这里我们新建了一个 DownloadBinder 类，并让它继承自 Binder，然后在它的内部提供了开始下载以及查看下载进度的方法。当然这只是两个模拟方法，并没有实现真正的功能，我们在这两个方法中分别打印了一行日志。

接着，在 MyService 中创建了 DownloadBinder 的实例，然后在 onBind()方法里返回了这个实例，这样 MyService 中的工作就全部完成了。

下面就要看一看，在活动中如何去调用服务里的这些方法了。首先需要在布局文件里新增两个按钮，修改 activity_main.xml 中的代码，如下所示：

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical" >
    ……
    <Button
        android:id="@+id/bind_service"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="Bind Service" />

    <Button
        android:id="@+id/unbind_service"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="Unbind Service" />
</LinearLayout>
```

这两个按钮分别是用于绑定服务和取消绑定服务的，那到底谁需要去和服务绑定呢？当然就是活动了。当一个活动和服务绑定了之后，就可以调用该服务里的 Binder 提供的方法了。修改 MainActivity 中的代码，如下所示：

```
public class MainActivity extends Activity implements OnClickListener {

    private Button startService;

    private Button stopService;

    private Button bindService;

    private Button unbindService;

    private MyService.DownloadBinder downloadBinder;

    private ServiceConnection connection = new ServiceConnection() {

        @Override
        public void onServiceDisconnected(ComponentName name) {
        }

        @Override
        public void onServiceConnected(ComponentName name, IBinder service) {
            downloadBinder = (MyService.DownloadBinder) service;
            downloadBinder.startDownload();
            downloadBinder.getProgress();
        }
    };

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        ……
        bindService = (Button) findViewById(R.id.bind_service);
        unbindService = (Button) findViewById(R.id.unbind_service);
        bindService.setOnClickListener(this);
        unbindService.setOnClickListener(this);
    }

    @Override
    public void onClick(View v) {
        switch (v.getId()) {
```

```
            ......
            case R.id.bind_service:
                Intent bindIntent = new Intent(this, MyService.class);
                bindService(bindIntent, connection, BIND_AUTO_CREATE); // 绑定服务
                break;
            case R.id.unbind_service:
                unbindService(connection); // 解绑服务
                break;
            default:
                break;
            }
        }

    }
```

可以看到，这里我们首先创建了一个 ServiceConnection 的匿名类，在里面重写了 onServiceConnected()方法和 onServiceDisconnected()方法，这两个方法分别会在活动与服务成功绑定以及解除绑定的时候调用。在 onServiceConnected()方法中，我们又通过向下转型得到了 DownloadBinder 的实例，有了这个实例，活动和服务之间的关系就变得非常紧密了。现在我们可以在活动中根据具体的场景来调用 DownloadBinder 中的任何 public 方法，即实现了指挥服务干什么，服务就去干什么的功能。这里仍然只是做了个简单的测试，在 onServiceConnected()方法中调用了 DownloadBinder 的 startDownload()和 getProgress()方法。

当然，现在活动和服务其实还没进行绑定呢，这个功能是在 Bind Service 按钮的点击事件里完成的。可以看到，这里我们仍然是构建出了一个 Intent 对象，然后调用 bindService()方法将 MainActivity 和 MyService 进行绑定。bindService()方法接收三个参数，第一个参数就是刚刚构建出的 Intent 对象，第二个参数是前面创建出的 ServiceConnection 的实例，第三个参数则是一个标志位，这里传入 BIND_AUTO_CREATE 表示在活动和服务进行绑定后自动创建服务。这会使得 MyService 中的 onCreate()方法得到执行，但 onStartCommand()方法不会执行。

然后如果我们想解除活动和服务之间的绑定该怎么办呢？调用一下 unbindService()方法就可以了，这也是 Unbind Service 按钮的点击事件里实现的功能。

现在让我们重新运行一下程序吧，界面如图 9.9 所示。

图　9.9

点击一下 Bind Service 按钮，然后观察 LogCat 中的打印日志如图 9.10 所示：

Tag	Text
MyService	onCreate executed
MyService	startDownload executed
MyService	getProgress executed

图　9.10

可以看到，首先是 MyService 的 onCreate()方法得到了执行，然后 startDownload()和 getProgress()方法都得到了执行，说明我们确实已经在活动里成功调用了服务里提供的方法了。

另外需要注意，任何一个服务在整个应用程序范围内都是通用的，即 MyService 不仅可以和 MainActivity 绑定，还可以和任何一个其他的活动进行绑定，而且在绑定完成后它们都可以获取到相同的 DownloadBinder 实例。

9.4　服务的生命周期

之前章节我们学习过了活动以及碎片的生命周期。类似地，服务也有自己的生命周期，前面我们使用到的 onCreate()、onStartCommand()、onBind()和 onDestroy()等方法都是在服务

的生命周期内可能回调的方法。

一旦在项目的任何位置调用了 Context 的 startService()方法，相应的服务就会启动起来，并回调 onStartCommand()方法。如果这个服务之前还没有创建过，onCreate()方法会先于 onStartCommand()方法执行。服务启动了之后会一直保持运行状态，直到 stopService()或 stopSelf()方法被调用。注意虽然每调用一次 startService()方法，onStartCommand()就会执行一次，但实际上每个服务都只会存在一个实例。所以不管你调用了多少次 startService()方法，只需调用一次 stopService()或 stopSelf()方法，服务就会停止下来了。

另外，还可以调用 Context 的 bindService()来获取一个服务的持久连接，这时就会回调服务中的 onBind()方法。类似地，如果这个服务之前还没有创建过，onCreate()方法会先于 onBind()方法执行。之后，调用方可以获取到 onBind()方法里返回的 IBinder 对象的实例，这样就能自由地和服务进行通信了。只要调用方和服务之间的连接没有断开，服务就会一直保持运行状态。

当调用了 startService()方法后，又去调用 stopService()方法，这时服务中的 onDestroy()方法就会执行，表示服务已经销毁了。类似地，当调用了 bindService()方法后，又去调用 unbindService()方法，onDestroy()方法也会执行，这两种情况都很好理解。但是需要注意，我们是完全有可能对一个服务既调用了 startService()方法，又调用了 bindService()方法的，这种情况下该如何才能让服务销毁掉呢？根据 Android 系统的机制，一个服务只要被启动或者被绑定了之后，就会一直处于运行状态，必须要让以上两种条件同时不满足，服务才能被销毁。所以，这种情况下要同时调用 stopService()和 unbindService()方法，onDestroy()方法才会执行。

这样你就已经把服务的生命周期完整地走了一遍。

9.5 服务的更多技巧

以上所学的都是关于服务最基本的一些用法和概念，当然也是最常用的。不过，仅仅满足于此显然是不够的，服务的更多高级使用技巧还在等着我们呢，下面就赶快去看一看吧。

9.5.1 使用前台服务

服务几乎都是在后台运行的，一直以来它都是默默地做着辛苦的工作。但是服务的系统优先级还是比较低的，当系统出现内存不足的情况时，就有可能会回收掉正在后台运行的服务。如果你希望服务可以一直保持运行状态，而不会由于系统内存不足的原因导致被回收，就可以考虑使用前台服务。前台服务和普通服务最大的区别就在于，它会一直有一个正在运行的图标在系统的状态栏显示，下拉状态栏后可以看到更加详细的信息，非常类似于通知的效果。当然有时候你也可能不仅仅是为了防止服务被回收掉才使用前台服务的，有些项目由

于特殊的需求会要求必须使用前台服务，比如说墨迹天气，它的服务在后台更新天气数据的同时，还会在系统状态栏一直显示当前的天气信息，如图 9.11 所示。

图　9.11

那么我们就来看一下如何才能创建一个前台服务吧，其实并不复杂，修改 MyService 中的代码，如下所示：

```
public class MyService extends Service {
    ……
    @Override
    public void onCreate() {
        super.onCreate();
        Notification notification = new Notification(R.drawable.ic_launcher,
"Notification comes", System. currentTimeMillis());
        Intent notificationIntent = new Intent(this, MainActivity.class);
        PendingIntent pendingIntent = PendingIntent.getActivity(this, 0,
notificationIntent, 0);
        notification.setLatestEventInfo(this, "This is title", "This is
content", pendingIntent);
        startForeground(1, notification);
        Log.d("MyService", "onCreate executed");
```

```
        }
        ......
    }
```

可以看到，这里只是修改了 onCreate()方法中的代码，相信这部分的代码你会非常眼熟。没错！这就是我们在上一章中学习的创建通知的方法。只不过这次在构建出 Notification 对象后并没有使用 NotificationManager 来将通知显示出来，而是调用了 startForeground()方法。这个方法接收两个参数，第一个参数是通知的 id，类似于 notify()方法的第一个参数，第二个参数则是构建出的 Notification 对象。调用 startForeground()方法后就会让 MyService 变成一个前台服务，并在系统状态栏显示出来。

现在重新运行一下程序，并点击 Start Service 或 Bind Service 按钮，MyService 就会以前台服务的模式启动了，并且在系统状态栏会显示一个通知图标，下拉状态栏后可以看到该通知的详细内容，如图 9.12 所示。

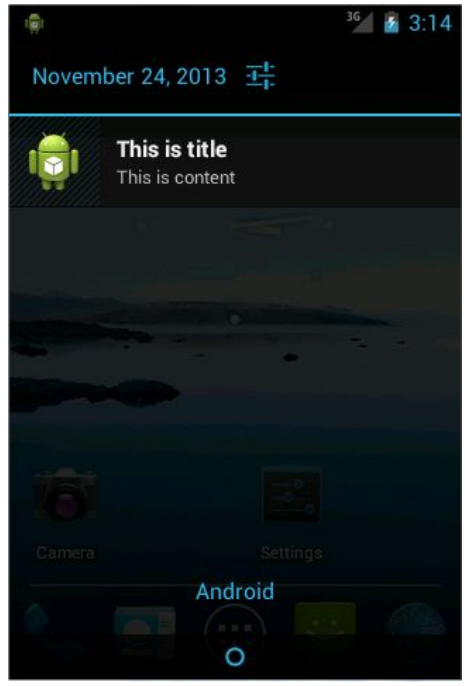

图 9.12

前台服务的用法就这么简单，只要你在上一章中将通知的用法掌握好了，学习本节的知识一定会特别轻松。

9.5.2　使用 IntentService

话说回来，在本章一开始的时候我们就已经知道，服务中的代码都是默认运行在主线程当中的，如果直接在服务里去处理一些耗时的逻辑，就很容易出现 ANR（Application Not Responding）的情况。

所以这个时候就需要用到 Android 多线程编程的技术了，我们应该在服务的每个具体的方法里开启一个子线程，然后在这里去处理那些耗时的逻辑。因此，一个比较标准的服务就可以写成如下形式：

```
public class MyService extends Service {

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

    @Override
    public int onStartCommand(Intent intent, int flags, int startId) {
        new Thread(new Runnable() {
            @Override
            public void run() {
                // 处理具体的逻辑
            }
        }).start();
        return super.onStartCommand(intent, flags, startId);
    }

}
```

但是，这种服务一旦启动之后，就会一直处于运行状态，必须调用 stopService()或者 stopSelf()方法才能让服务停止下来。所以，如果想要实现让一个服务在执行完毕后自动停止的功能，就可以这样写：

```
public class MyService extends Service {

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

```
    @Override
    public int onStartCommand(Intent intent, int flags, int startId) {
        new Thread(new Runnable() {
            @Override
            public void run() {
                // 处理具体的逻辑
                stopSelf();
            }
        }).start();
        return super.onStartCommand(intent, flags, startId);
    }

}
```

虽说这种写法并不复杂，但是总会有一些程序员忘记开启线程，或者忘记调用 stopSelf() 方法。为了可以简单地创建一个异步的、会自动停止的服务，Android 专门提供了一个 IntentService 类，这个类就很好地解决了前面所提到的两种尴尬，下面我们就来看一下它的用法。

新建一个 MyIntentService 类继承自 IntentService，代码如下所示：

```
public class MyIntentService extends IntentService {

    public MyIntentService() {
        super("MyIntentService"); // 调用父类的有参构造函数
    }

    @Override
    protected void onHandleIntent(Intent intent) {
        // 打印当前线程的id
        Log.d("MyIntentService", "Thread id is " + Thread.currentThread().
getId());
    }

    @Override
    public void onDestroy() {
        super.onDestroy();
        Log.d("MyIntentService", "onDestroy executed");
    }

}
```

这里首先是要提供一个无参的构造函数，并且必须在其内部调用父类的有参构造函数。然后要在子类中去实现 onHandleIntent()这个抽象方法，在这个方法中可以去处理一些具体的逻辑，而且不用担心 ANR 的问题，因为这个方法已经是在子线程中运行的了。这里为了证实一下，我们在 onHandleIntent()方法中打印了当前线程的 id。另外根据 IntentService 的特性，这个服务在运行结束后应该是会自动停止的，所以我们又重写了 onDestroy()方法，在这里也打印了一行日志，以证实服务是不是停止掉了。

接下来修改 activity_main.xml 中的代码，加入一个用于启动 MyIntentService 这个服务的按钮，如下所示：

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical" >
    ……
    <Button
        android:id="@+id/start_intent_service"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="Start IntentService" />
</LinearLayout>
```

然后修改 MainActivity 中的代码，如下所示：

```
public class MainActivity extends Activity implements OnClickListener {
    ……
    private Button startIntentService;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        ……
        startIntentService = (Button) findViewById(R.id.start_intent_service);
        startIntentService.setOnClickListener(this);
    }

    @Override
    public void onClick(View v) {
        switch (v.getId()) {
        ……
```

```
            case R.id.start_intent_service:
                // 打印主线程的id
                Log.d("MainActivity", "Thread id is " + Thread.currentThread().
    getId());
                Intent intentService = new Intent(this, MyIntentService.class);
                startService(intentService);
                break;
            default:
                break;
            }
        }
    }
```

可以看到，我们在 Start IntentService 按钮的点击事件里面去启动 MyIntentService 这个服务，并在这里打印了一下主线程的 id，稍后用于和 IntentService 进行比对。你会发现，其实 IntentService 的用法和普通的服务没什么两样。

最后仍然不要忘记，服务都是需要在 AndroidManifest.xml 里注册的，如下所示：

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.example.servicetest"
    android:versionCode="1"
    android:versionName="1.0" >
    ……
    <application
        android:allowBackup="true"
        android:icon="@drawable/ic_launcher"
        android:label="@string/app_name"
        android:theme="@style/AppTheme" >
        ……
        <service android:name=".MyIntentService"></service>
    </application>
</manifest>
```

现在重新运行一下程序，界面如图 9.13 所示。

图　9.13

点击 Start IntentService 按钮后，观察 LogCat 中的打印日志，如图 9.14 所示。

Tag	Text
MainActivity	Thread id is 1
MyIntentService	Thread id is 97
MyIntentService	onDestroy executed

图　9.14

可以看到，不仅 MyIntentService 和 MainActivity 所在的线程 id 不一样，而且 onDestroy() 方法也得到了执行，说明 MyIntentService 在运行完毕后确实自动停止了。集开启线程和自动停止于一身，IntentService 还是博得了不少程序员的喜爱。

好了，关于服务的知识点你已经学得够多了，下面就让我们进入到本章的最佳实践环节吧。

9.6　服务的最佳实践——后台执行的定时任务

好久已经没有来到最佳实践环节了，是不是有些想念了呢？本章中你已经掌握了关于服务非常多的使用技巧，但是当在真正的项目里需要用到服务的时候，可能还会有一些棘手的

问题让你不知所措。因此，下面我们就来学习一下在服务中经常用到的技术之一，在后台执行定时任务。

Android 中的定时任务一般有两种实现方式，一种是使用 Java API 里提供的 Timer 类，一种是使用 Android 的 Alarm 机制。这两种方式在多数情况下都能实现类似的效果，但 Timer 有一个明显的短板，它并不太适用于那些需要长期在后台运行的定时任务。我们都知道，为了能让电池更加耐用，每种手机都会有自己的休眠策略，Android 手机就会在长时间不操作的情况下自动让 CPU 进入到睡眠状态，这就有可能导致 Timer 中的定时任务无法正常运行。而 Alarm 机制则不存在这种情况，它具有唤醒 CPU 的功能，即可以保证每次需要执行定时任务的时候 CPU 都能正常工作。需要注意，这里唤醒 CPU 和唤醒屏幕完全不是同一个概念，千万不要产生混淆。

那么首先我们来看一下 Alarm 机制的用法吧，其实并不复杂，主要就是借助了 AlarmManager 类来实现的。这个类和 NotificationManager 有点类似，都是通过调用 Context 的 getSystemService()方法来获取实例的，只是这里需要传入的参数是 Context.ALARM_SERVICE。因此，获取一个 AlarmManager 的实例就可以写成：

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

接下来调用 AlarmManager 的 set()方法就可以设置一个定时任务了，比如说想要设定一个任务在 10 秒钟后执行，就可以写成：

```
long triggerAtTime = SystemClock.elapsedRealtime() + 10 * 1000;
manager.set(AlarmManager.ELAPSED_REALTIME_WAKEUP, triggerAtTime, pendingIntent);
```

上面的两行代码你不一定能看得明白，因为 set()方法中需要传入的三个参数稍微有点复杂，下面我们就来仔细地分析一下。第一个参数是一个整型参数，用于指定 AlarmManager 的工作类型，有四种值可选，分别是 ELAPSED_REALTIME、ELAPSED_REALTIME_WAKEUP、RTC 和 RTC_WAKEUP。其中 ELAPSED_REALTIME 表示让定时任务的触发时间从系统开机开始算起，但不会唤醒 CPU。ELAPSED_REALTIME_WAKEUP 同样表示让定时任务的触发时间从系统开机开始算起，但会唤醒 CPU。RTC 表示让定时任务的触发时间从 1970 年 1 月 1 日 0 点开始算起，但不会唤醒 CPU。RTC_WAKEUP 同样表示让定时任务的触发时间从 1970 年 1 月 1 日 0 点开始算起，但会唤醒 CPU。使用 SystemClock.elapsedRealtime()方法可以获取到系统开机至今所经历时间的毫秒数，使用 System.currentTimeMillis()方法可以获取到 1970 年 1 月 1 日 0 点至今所经历时间的毫秒数。

然后看一下第二个参数，这个参数就好理解多了，就是定时任务触发的时间，以毫秒为单位。如果第一个参数使用的是 ELAPSED_REALTIME 或 ELAPSED_REALTIME_WAKEUP，则这里传入开机至今的时间再加上延迟执行的时间。如果第一个参数使用的是 RTC 或 RTC_WAKEUP，则这里传入 1970 年 1 月 1 日 0 点至今的时间再加上延迟执行的时间。

第三个参数是一个 PendingIntent，对于它你应该已经不会陌生了吧。这里我们一般会调用 getBroadcast()方法来获取一个能够执行广播的 PendingIntent。这样当定时任务被触发的时候，广播接收器的 onReceive()方法就可以得到执行。

了解了 set()方法的每个参数之后，你应该能想到，设定一个任务在 10 秒钟后执行还可以写成：

```
long triggerAtTime = System.currentTimeMillis() + 10 * 1000;
manager.set(AlarmManager.RTC_WAKEUP, triggerAtTime, pendingIntent);
```

好了，现在你已经掌握 Alarm 机制的基本用法，下面我们就来创建一个可以长期在后台执行定时任务的服务。创建一个 ServiceBestPractice 项目，然后新增一个 LongRunningService 类，代码如下所示：

```
public class LongRunningService extends Service {

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

    @Override
    public int onStartCommand(Intent intent, int flags, int startId) {
        new Thread(new Runnable() {
            @Override
            public void run() {
                Log.d("LongRunningService", "executed at " + new Date().
toString());
            }
        }).start();
        AlarmManager manager = (AlarmManager) getSystemService(ALARM_SERVICE);
        int anHour = 60 * 60 * 1000;  // 这是一小时的毫秒数
        long triggerAtTime = SystemClock.elapsedRealtime() + anHour;
        Intent i = new Intent(this, AlarmReceiver.class);
        PendingIntent pi = PendingIntent.getBroadcast(this, 0, i, 0);
        manager.set(AlarmManager.ELAPSED_REALTIME_WAKEUP, triggerAtTime, pi);
        return super.onStartCommand(intent, flags, startId);
    }

}
```

我们在 onStartCommand()方法里开启了一个子线程，然后在子线程里就可以执行具体的

逻辑操作了。这里简单起见，只是打印了一下当前的时间。

创建线程之后的代码就是我们刚刚讲解的 Alarm 机制的用法了，先是获取到了 AlarmManager 的实例，然后定义任务的触发时间为一小时后，再使用 PendingIntent 指定处理定时任务的广播接收器为 AlarmReceiver，最后调用 set()方法完成设定。

显然，AlarmReceiver 目前还不存在呢，所以下一步就是要新建一个 AlarmReceiver 类，并让它继承自 BroadcastReceiver，代码如下所示：

```
public class AlarmReceiver extends BroadcastReceiver {

    @Override
    public void onReceive(Context context, Intent intent) {
        Intent i = new Intent(context, LongRunningService.class);
        context.startService(i);
    }

}
```

onReceive()方法里的代码非常简单，就是构建出了一个 Intent 对象，然后去启动 LongRunningService 这个服务。那么这里为什么要这样写呢？其实在不知不觉中，这就已经将一个长期在后台定时运行的服务完成了。因为一旦启动 LongRunningService，就会在 onStartCommand()方法里设定一个定时任务，这样一小时后 AlarmReceiver 的 onReceive()方法就将得到执行，然后我们在这里再次启动 LongRunningService，这样就形成了一个永久的循环，保证 LongRunningService 可以每隔一小时就会启动一次，一个长期在后台定时运行的服务自然也就完成了。

接下来的任务也很明确了，就是我们需要在打开程序的时候启动一次 LongRunningService，之后 LongRunningService 就可以一直运行了。修改 MainActivity 中的代码，如下所示：

```
public class MainActivity extends Activity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        Intent intent = new Intent(this, LongRunningService.class);
        startService(intent);
    }

}
```

最后别忘了，我们所用到的服务和广播接收器都要在 AndroidManifest.xml 中注册才行，代码如下所示：

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.example.servicebestpractice"
    android:versionCode="1"
    android:versionName="1.0" >
    ……
    <application
        android:allowBackup="true"
        android:icon="@drawable/ic_launcher"
        android:label="@string/app_name"
        android:theme="@style/AppTheme" >
        <activity
            android:name="com.example.servicebestpractice.MainActivity"
            android:label="@string/app_name" >
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />
                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
        <service android:name=".LongRunningService" >
        </service>
        <receiver android:name=".AlarmReceiver" >
        </receiver>
    </application>
</manifest>
```

现在就可以来运行一下程序了。虽然你不会在界面上看到任何有用的信息，但实际上 LongRunningService 已经在后台悄悄地运行起来了。为了能够验证一下运行结果，我将手机闲置了几个小时，然后观察 LogCat 中的打印日志，如图 9.15 所示。

Tag	Text
LongRunningService	executed at Sat Nov 30 16:03:17 GMT+00:00 2013
LongRunningService	executed at Sat Nov 30 17:03:17 GMT+00:00 2013
LongRunningService	executed at Sat Nov 30 18:03:17 GMT+00:00 2013
LongRunningService	executed at Sat Nov 30 19:03:17 GMT+00:00 2013
LongRunningService	executed at Sat Nov 30 20:03:17 GMT+00:00 2013
LongRunningService	executed at Sat Nov 30 21:03:17 GMT+00:00 2013
LongRunningService	executed at Sat Nov 30 22:03:17 GMT+00:00 2013
LongRunningService	executed at Sat Nov 30 23:03:17 GMT+00:00 2013
LongRunningService	executed at Sun Dec 01 00:03:17 GMT+00:00 2013
LongRunningService	executed at Sun Dec 01 01:03:17 GMT+00:00 2013
LongRunningService	executed at Sun Dec 01 02:03:17 GMT+00:00 2013
LongRunningService	executed at Sun Dec 01 03:03:17 GMT+00:00 2013

图　9.15

可以看到，LongRunningService 果然如我们所愿地运行着，每隔一小时都会打印一条日志。这样，当你真正需要去执行某个定时任务的时候，只需要将打印日志替换成具体的任务逻辑就行了。

另外需要注意的是，从 Android 4.4 版本开始，Alarm 任务的触发时间将会变得不准确，有可能会延迟一段时间后任务才能得到执行。这并不是个 bug，而是系统在耗电性方面进行的优化。系统会自动检测目前有多少 Alarm 任务存在，然后将触发时间将近的几个任务放在一起执行，这就可以大幅度地减少 CPU 被唤醒的次数，从而有效延长电池的使用时间。

当然，如果你要求 Alarm 任务的执行时间必须准备无误，Android 仍然提供了解决方案。使用 AlarmManager 的 setExact()方法来替代 set()方法，就可以保证任务准时执行了。

好了，最佳实践部分到此结束，下面我们就来回顾一下本章所学的内容吧。

9.7 小结与点评

在本章中，我们学习了很多与服务相关的重要知识点，包括 Android 多线程编程、服务的基本用法、服务的生命周期、前台服务和 IntentService 等。这些内容已经覆盖了大部分在你日常开发中可能用到的服务技术，再加上最佳实践部分学习的后台定时任务的技巧，相信以后不管遇到什么样的服务难题，你都能从容解决吧。

另外，本章同样是有里程碑式的纪念意义的，因为我们已经将 Android 中的四大组件全部学完，并且本书的内容也学习过半了。对于你来说，现在你已经脱离了 Android 初级开发者的身份，并应该具备了独立完成很多功能的能力了。

那么后面我们应该再接再厉，争取进一步地提升自身的能力，所以现在还不是放松的时候。目前我们所学的所有东西都仅仅是在本地上进行的，而实际上几乎市场上的每个应用都会涉及到网络交互的部分，所以下一章中我们就来学习一下 Android 网络编程方面的内容。

经验值：+20000　　目前经验值：129905

级别：头领鸟

赢得宝物：战胜服务生。拾取服务生掉落的宝物，高能级长老 Android 战袍一套、长老权杖一柄。不要被服务生的称呼所误导，服务生是神界具有甚高地位的一位尊者，一位长老，是神界总共 12 位长老之一。神界的 12 位长老都不叫长老，主要是为了低调，同时也是为了保护神界的低能量生灵（比如神界的小蚂蚁、小蜜蜂等），因为当长老们用长老的称谓对别人讲话时，一股强大的能量场会自动从长老身上发出，从而让周围的低能量生灵感到不适，而不使用长老称谓时则没有这个现象。