2018

Android 移动开发实训

程源

广东机电职业技术学院

2018/9/1

目录

9 实现语音服务

9.1 本章目标

第八章我们通过资源打包，为 NEC Vocab 应用添加了词汇资源。但还存在一些问题。首先是当用户查看完一个词条的"词汇详解"之后，想查看上一个词汇或下一个词条，必须先用回退键返回的词汇列表，然后再选择想查看的词条。这不是一个好的用户体验。本章，我们将为"词汇详解"添加两个前后翻页的按钮，使用可以不用返回"词汇列表"界面，就翻到上一个词条或下一个词条。当然可以有很多种方案实现这一功能，其中 ViewPager 就是一种很好的实现方式，甚至可以做到滑动屏幕来翻页。但本章我们将采用消息机制来实现，以便介绍 Android 中的重要概念——线程和服务。事实上，本章内容不用新的线程也完全可以运行。但是，如果这些词汇数据不是来自本地，而是来自互联网，就必须使用新的线程了。

9.2 创建并启动后台线程

下面我们来考虑如何启动后台线程。在动手之前，我们先来看看什么是线程。

9.2.1 Android 进程和线程

Android 系统是 Google 公司基于 Linux 内核开发的开源手机操作系统。通过调用 Linux 内核，Android 系统使用了大量操作系统服务，包括进程管理、内存管理、网络堆栈、驱动程序、安全性等相关的服务。因此，Android 系统的进程和线程就是 Linux 系统进程、线程的映射。

进程和线程

下面从操作系统层面解释一下什么是进程和线程。

进程（**Process**）是应用程序的一个运行过程，是操作系统资源管理的实体。进程是操作系

统分配和调度系统内存、CPU 时间片等资源的基本单位，为正在运行的应用程序提供运行环境。每个进程都有自己独立的内存地址空间。

线程（**Thread**），线程是进程内部执行代码的实体，它是 CPU 调度资源的最小单元。一个进程至少包括一个线程（一个进程内部可以有多个线程并发运行）。线程没有自己独立的内存资源，它只有自己的执行堆栈和局部变量，所以线程不能独立地执行，它必须依附在一个进程上。在同一个进程内多个线程之间可以共享进程的内存资源。

线程、进程、组件与 Dalvik 虚拟机

前面我们从操作系统的角度介绍了线程和进程。而我们在 Android 编程时经常碰到的概念却是**应用**（Application）、**活动**（Activity）、**任务**（Task）、**服务**（Service）等。它们之间的关系是如何的呢？

在安装 Android 应用程序的时候，默认情况下，操作系统为每个应用程序分配一个唯一的 Linux 用户的 ID。权限设置为每个应用的文件仅对用户和该应用本身可见，其他应用不能访问该应用的数据和资源，这就保证了信息安全。所以，一个 Android 应用程序运行时至少对应一个 Linux 进程。默认情况下，不同的应用在不同的进程空间里运行，每个应用都有它自己的 **Dalvik 虚拟机**，都独立于其他应用运行。

当我们单击移动设备上某个应用时，如果设备的内存中没有该应用的任何组件，那么系统会为这个应用启动一个新的 Linux 进程，在这个进程里运行一个新的 Dalvik 虚拟机实例，而应用程序则运行在这个 Dalvik 虚拟机实例里。Dalvik 虚拟机主要完成组件生命周期管理、堆栈管理、线程管理、安全和异常管理，以及垃圾回收等重要功能。虚拟机的这些功能都直接利用底层操作系统的功能来实现。默认情况下，这个进程只有一个线程——**主线程**。

主线程主要负责处理与 UI 相关的事件，如用户的按键事件、用户接触屏幕的事件等，并把相关的事件分发到对应的组件进行处理。所以主线程通常又被叫做 **UI 线程**。应用程序里的所有组件，活动 Activity、服务 Service、广播 Broadcast Receiver 等都运行在这个主线程中。

注意，应用程序里多个 Activity，以及后台运行的 Service 甚至广播接收器，在默认情况下，都是在同一个进程和同一个主线程中被实例化和被调用运行的。因为主线程负责事件的监听和绘图，所以，**必须保证主线程能够随时响应用户的需求**。也就是说，当应用程序运行时不应该有哪个组件（包括服务 Service）进行远程或者耗时操作（比如网络调用或者复杂运算），这将阻碍进程中的所有其他组件的运行。这时，我们必须新开线程去并行执行远程操作、耗时操作

等代码，而主线程里的代码应该尽量短小。

在开发 Android 应用时必须遵守**单线程模型的原则**：**Android UI 操作并不是线程安全的并且这些操作必须在主线程（UI 线程）中执行**。

我们单击的应用程序运行后，一般会在手机屏幕显示界面，界面上可能有图片、文字和按钮等控件。单击这些控件会跳转到其他界面或者跳出提示或者运行特定的程序代码。然后，我们可能按手机的 Back 按键返回到之前的界面或者退出应用，或者按 Home 键到桌面。在这些操作过程中，应用程序的各组件在主线程或各自线程中被系统调用运行。这个阶段就是组件的生命周期过程。

即使我们完全退出应用程序，这个应用程序所使用的进程、虚拟机和线程等资源还将在内存中存在，直到系统内存不足时被系统回收。Android 系统会根据进程中运行的组件类别以及组件的状态来判断各进程的重要性，并根据这个重要性来决定回收时的优先级。

进程重要性从高到低一共有 5 个级别：

- 前台进程：它是用户当前正在使用的进程，是优先级最高的进程。
- 可见进程：它是在屏幕上有显示但却不是用户当前使用的进程。
- 服务进程：运行着服务 Service 的进程，只要前台进程和可见进程有足够的内存，系统不会回收它们。
- 后台进程：运行着一个对用户不可见的 Activity（并调用过 onStop() 方法）的进程，在前 3 种进程需要内存时，被系统回收。
- 空进程：未运行任何程序组件。

采用这种懒人策略方式回收资源的优点是下次启动该应用程序时会更快速，而弊端是系统资源得不到及时回收，当需要新启动一个应用时有可能因资源不足而等待系统回收资源。

9.2.2 Android 中的消息机制

前面说过，UI 操作并不是线程安全，必须在主线程中执行。为了避免阻塞主线程，一些耗时的操作应该交给独立的线程去执行。

Android 中线程是使用标准的 **java Thread** 对象来建立的。为了解决与后台线程通信问题，Android 系统提供了一系列方便的类来管理线程——**Looper** 用来在一个线程中执行消息循环，**Handler** 用来处理消息，**HandlerThread** 创建带有消息循环的线程。同时，Android 设计了一个 **Message Queue**(消息队列)，线程间可以通过该 Message Queue 并结合 Handler 和 Looper 组件

进行信息交换。

Message（消息）

理解为线程间交流的信息。处理数据的后台线程需要更新 UI，则发送携带一些数据的 Message 给 UI 线程。

Handler（处理者）

是 Message 的主要处理者，负责 Message 的发送，Message 内容的执行处理。后台线程就是通过传进来的 Handler 对象引用来 sendMessage(Message)。而使用 Handler，需要实现该类的 handleMessage(Message) 方法，它是处理这些 Message 的操作内容，例如 Update UI。通常需要子类化 Handler 来实现 handleMessage 方法。

Message Queue（消息队列）

用来存放通过 Handler 发布的消息，按照先进先出的方式执行。每个 message queue 都有一个对应的 Handler。Handler 通过 sendMessage 或 post 两种方法向 message queue 发送消息。这两种方法发送的消息都会插在 message queue 队尾，按先进先出的方式执行。但通过这两种方法发送的消息执行的方式略有不同：sendMessage 发送的是一个 message 对象，会被 Handler 的 handleMessage()函数处理；而 post 发送的是一个 runnable 对象，可以自己执行。

Looper

Looper 是每条线程里的 Message Queue 的管家。Android 没有全局的 Message Queue，会自动替主线程建立 Message Queue，但在子线程里并没有建立 Message Queue。所以调用 Looper.getMainLooper()得到的主线程的 Looper 不为 NULL，但调用 Looper.myLooper()得到当前线程的 Looper 就有可能为 NULL。

对于子线程使用 Looper，API Doc 提供了如下使用方法：

```
package com.test;
import android.os.Handler;
import android.os.Looper;
import android.os.Message;
class LooperThread extends Thread {
    public Handler mHandler;
    public void run() {
        Looper.prepare(); //创建本线程的 Looper 并创建一个 MessageQueue
        mHandler = new Handler() {
            public void handleMessage(Message msg) {
            // process incoming messages here
            }
```

```
            };
            Looper.loop(); //开始运行 Looper,监听 Message Queue
        }
    }
```

线程的 3 种用法：

（1）继承 Thread：

```
class MyThread extends Thread{
    public void run( ){
        //执行耗时操作
    }
}
new MyThread( ).start( );
```

（2）实现 Runnable 接口：

```
class MyRunnable implements Runnable{
        public void run( ){
            //执行耗时操作
        }
}
new Thread(new MyRunnable).start( );
```

（3）使用匿名类：

```
new Thread(new Runnable( ){
    public void run( ){
        //执行耗时操作
    }
}).start( );
```

需要再次强调的是：**Android 不允许在子线程中更新 UI，只能在主线程中更新**。

不直接在子线程中进行 UI 操作，而是在子线程中通过 Handler 将 Message 传送给主线程，主线程中的 Handler 接收这个 message，然后进行 UI 操作，这叫**异步消息处理**。

9.2.3 异步消息处理

异步消息处理通常分为如下四个步骤。

（1）在主线程中创建一个 Handler 类型的类成员变量对象，并重写 handleMessage 方法，用于处理响应事件：

```
private Handler handler = new Handler( ){
    public void handleMessage(Message msg){
        switch(msg.what){
            //msg 的 what 字段是一个标志字段，整型
            case xxx:
```

```
                    //在这里可以进行 UI 操作
                    textView.setText("Change text succeed!")          //改变 textView 的字符
                    break;
                default:
                    break;
            }
        }
    }
```

（2）在子线程中需要进行 UI 操作时（如按钮点击事件中），创建一个 Message 对象，并通过 Handler 对象的 sendMessage 方法将该消息发送出去，比如：

```
public static final int UPDATE_TEXT = 1;          //修改 UI 的标志值
......
@Override
public void onClick(View v){
    switch(v.getId( )){
        case R.id.chage_text_btn:
            new Thread(new Runnable( ){
                @Override
                public void run( ){
                    Message msg = new Message( );
                    msg.what = UPDATE_TEXT ;
                    handler.sendMessage(msg);
                }
            }).start( );
        break;
            default:
            break;
        }
}
```

（3）发出的 Message 进入 MessageQueue 队列等待处理。

（4）Looper 一直尝试从 MessageQueue 中取出等待处理的消息，最后分发回 handleMessage。

异步消息处理机制如图 9-1 示意.

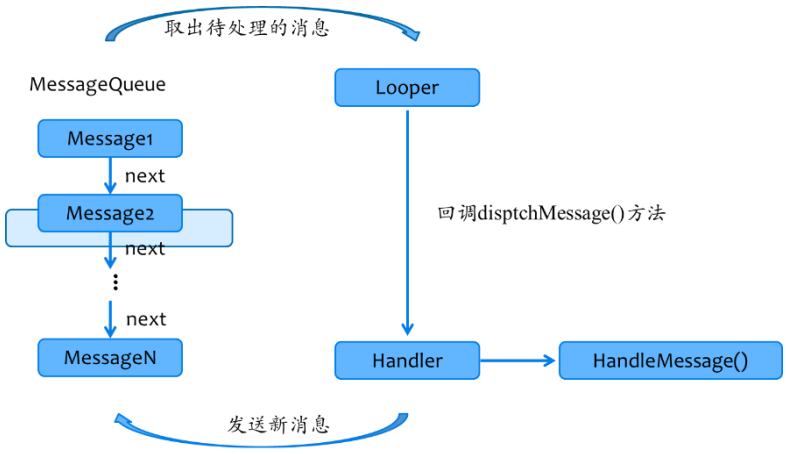

图 9-1 异步消息处理机制

9.3 使用消息机制

接下来我们利用消息机制来处理如何在直接从“词汇详解”中某词条翻转到上一词条或下一词条问题。

9.3.1 添加图标资源

为此我们首先在“词汇详解”的布局文件中添加“上一页”和 “下一页”两个按钮。如图 9-2，我们将用左右两个箭头的**图标**来表示“上一页”和 “下一页”。

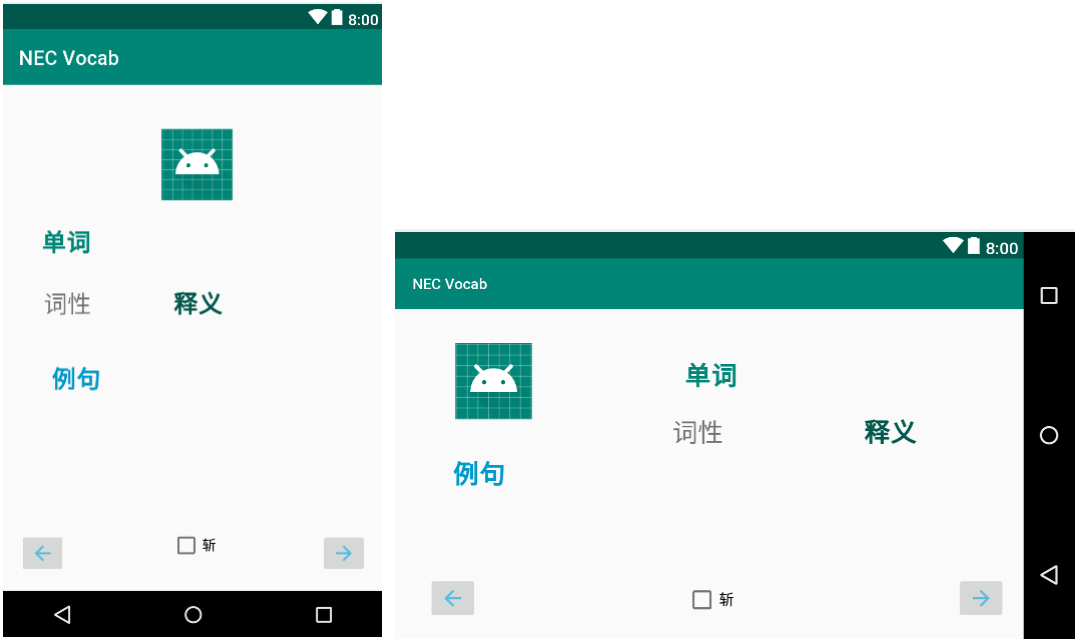

图 9-2 添加上一页和下一页图标

使用 Android Asset Studio

应用使用的图标有两种：**系统图标**和**项目资源图标**。系统图标（system icon）是 Android 操作系统内置的图标。android:icon 属性值@android:drawable/ic_menu_add 就引用了系统图标。

在应用原型设计阶段，使用系统图标不会有什么问题；而在应用发布时，无论用户运行什么设备，最好能统一应用的界面风格。因为不同设备或操作系统版本间，系统图标的显示风格往往具有很大差异。

一种解决方案是创建定制图标。这需要针对不同屏幕显示密度或各种可能的设备配置，准

备不同版本的图标。另一种解决方案是找到适合应用的系统图标，将它们直接复制到项目的 drawable 资源目录中。当然最容易的解决方案是使用 Android Studio 内置的 Android Asset Studio 工具。可以用它为工具栏定制图片。

在项目工具窗口中，右键单击 **drawable** 目录，选择 **New → Image Asset** 菜单项，弹出如图 9-3 所示的 Asset Studio 窗口。

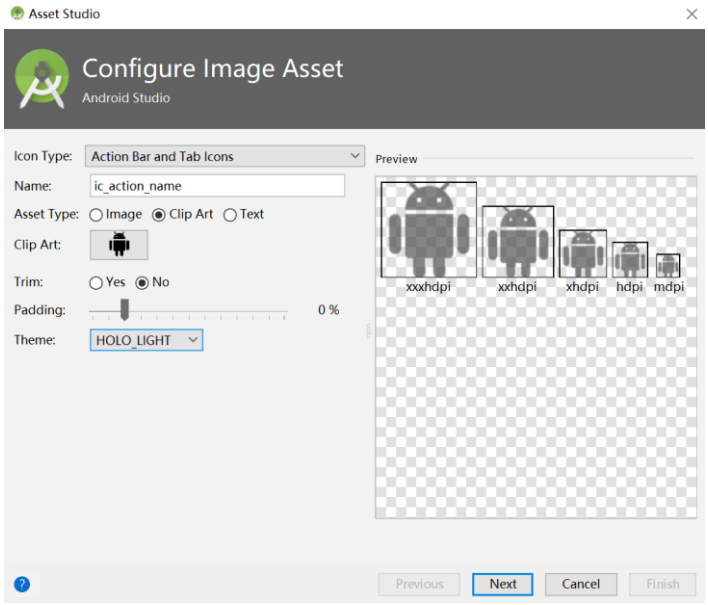

图 9-3 Asset Studio

这里，我们可以生成各类图标。在 **Icon Type**:中选择 **Action Bar and Tab Icons**，确定 **Asset Type** 选项为 **Clip Art**，然后单击 Clip Art 旁边的按钮挑选图标。

在可选图标窗口，选择看上去像"→"号的图片，如图 9-4 所示。

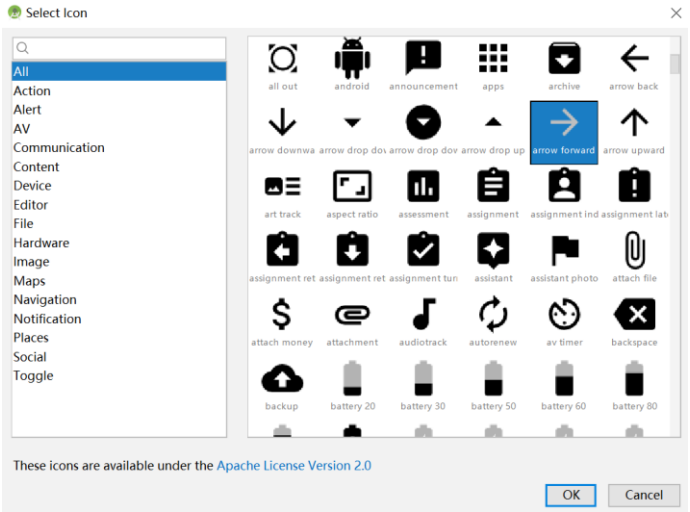

图 9-4 可选图标

最后，命名资源为 ic_button_next，可以通过 **Theme:**选择适合的样式，单击 **Next** 按钮进入图 9-5 所示的画面。

Asset Studio 会询问添加图片到哪个模块和目录。这里使用默认设置就可以了。Asset Studio 还提供了待添加的 mdpi、hdpi、xhdpi、xxhdpi 和 xxxhdpi 类型图标的预览画面。点击 Finish 按钮完成。

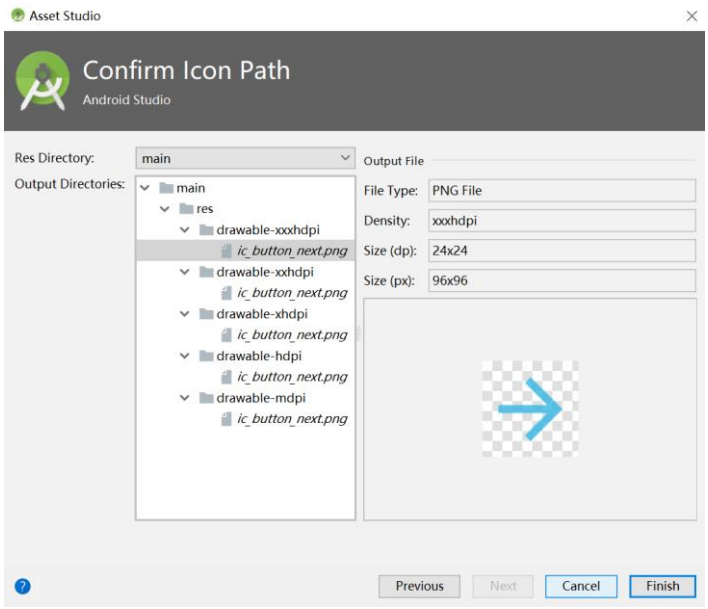

图 9-5 Asset Studio 生成的文件

以同样的方式，再添加一个向左的箭头作为“上一页”的图标。

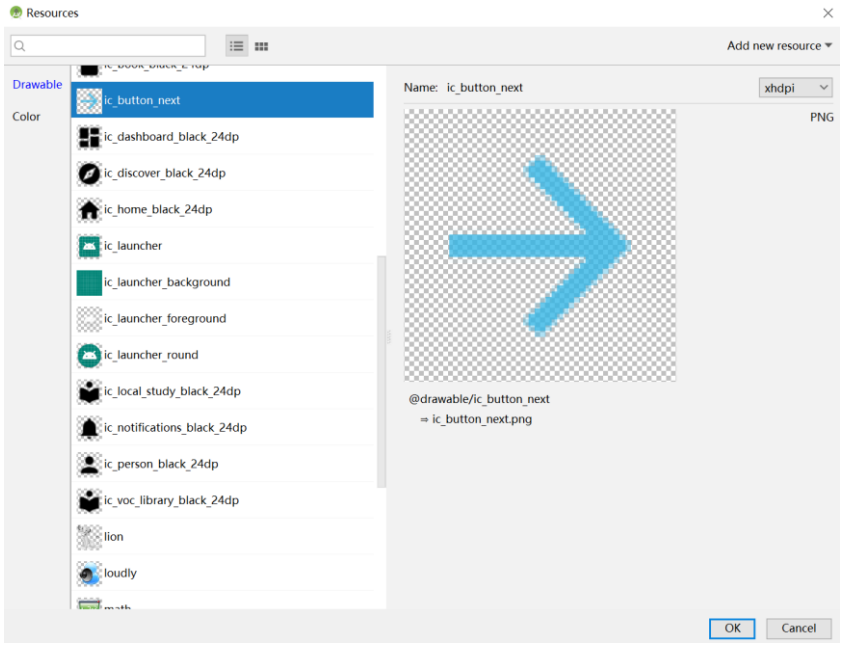

图 9-6 添加 ImageButton 的图标资源

接下来在 fragment_vocabulary.xml 文件，在 Design 标签页的组件面板（Palette）中选择 **Button → ImageButton**，并将其拖拽到布局中适合的位置，在随后出现的如图 9-6 所示的对话框中选中刚才创建的图标。

为 ImageButton 添加约束，同时为其 id 命名（如 next_button 和 last_button）。

最后，不要忘了为水平布局同样添加两个 ImageButton。

9.3.2 与主线程通信

要想显示"上一词条"或"下一词条"，我们首先要知道当前词条在 WordLib 中的位置。但是从 MainActivity 传递过来的 WordEntity 对象并没有包含这个位置信息。因此，我们需要通过该对象的 mId 获取其在 WordLib 中的位置。

如代码清单 9-1 所示，声明一个 List<WordEntity>的列表 mWordEntityList，获取**单列** WordLib 中的所有词汇，然后通过循环 mWordEntityList 得到该词条在词汇表中的位置。

代码清单 **9-1** 获取词条在 **WordLib** 中的位置（**VocabularyFragment.java**）

```
public class VocabularyFragment extends Fragment {
    private List<WordEntity> mWordEntityList;
    private int mCurrentIndex = 0;
    ......
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        if (getArguments() != null) {
            mWord = (WordEntity)getArguments().getSerializable(ARG_PARAM);
        } else {
            mWord = new WordEntity();
        }
        mWordEntityList = WordLib.get(getActivity()).getWordList();
        for (int i=0; i< mWordEntityList.size(); i++){
            if(mWordEntityList.get(i).getId().equals(mWord.getId())){
                mCurrentIndex = i;
            }
        }
    }

......
}
```

实现 OnClickListener 接口

接下来我们在 VocabularyFragment 中关联布局文件里的"上一页"和"下一页"组件，并在代

码中实现按钮的点击事件。

前几章我们实现点击事件方法是在 Activity 或 Fragment 中 **new** 出一个 OnClickListenner()，使用匿名内部类方法实现，如在 LoginActivity 中的实现方式是：

```
mRegister.setOnClickListener(new View.OnClickListener() {
    @Override
    public void onClick(View v) {
        Intent intent = new Intent(LoginActivity.this, RegisterActivity.class);
        startActivity(intent);
    }
});
```

使用这种方式，代码看上去比较直观。但当程序里面需要点击的组件较多时，尤其是采用 switch 语句判断点击按钮时，这种创建匿名内部类的方法显得有点累赘。这时可以用实现 OnClickListener 接口的方法，如代码清单 9-2 所示。

代码清单 9-2 实现 OnClickListener 接口方法（VocabularyFragment.java）

```
public class VocabularyFragment extends Fragment implements View.OnClickListener {
    ......
    private ImageButton mNextWord;
    private ImageButton mLastWord;
    ......
    @Override
    public View onCreateView(LayoutInflater inflater, ViewGroup container,
                             Bundle savedInstanceState) {
        View view = inflater.inflate(R.layout.fragment_vocabulary, container, false);
        ......
        mLastWord = (ImageButton) view.findViewById(R.id.last_word);
        mNextWord = (ImageButton) view.findViewById(R.id.next_word);
        mLastWord.setOnClickListener(this);
        mNextWord.setOnClickListener(this);
        ......
        updateWords();
        return view;
    }
    ......

    @Override
    public void onClick(View v) {
        switch (v.getId()) {
            case R.id.last_word:
                //目前什么也不做
                break;
            case R.id.next_word:
                //目前什么也不做
                break;
            default:
                break;
```

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

运行 NEC Vocab 应用，确保没有问题出现。

9.3.3 处理更新页面消息

下面我们采用 9.2.3 节介绍的异步消息处理的步骤来实现页面更新。

我们先定义两个常量用于表示更新到"上一页"和"下一页"这个两个动作：

```
public static final int LAST_WORD = 0;
public static final int NEXT_WORD = 1;
```

在主线程中创建一个 Handler 类型的类成员变量对象，并重写 handleMessage 方法，用于处理响应事件，如代码清单 9-3 所示：

代码清单 **9-3** 创建 **Handler** 类型的类成员变量（**VocabularyFragment.java**）

```
public class VocabularyFragment extends Fragment implements View.OnClickListener {
    public static final int LAST_WORD = 0;
    public static final int NEXT_WORD = 1;
    private List<WordEntity> mWordEntityList;
    private int mCurrentIndex = 0;
    ......
    @Override
    public View onCreateView(LayoutInflater inflater, ViewGroup container,
                             Bundle savedInstanceState) {
        View view = inflater.inflate(R.layout.fragment_vocabulary, container, false);
        ......
        updateWords();
        return view;
    }

    private Handler mHandler = new Handler() {
        public void handleMessage(Message msg) {
            switch (msg.what) {
                case LAST_WORD:
                    updateWords();
                    break;
                case NEXT_WORD:
                    updateWords();
                    break;
                default:
                    break;
            }
        }
    };
    ......
```

}

这里我们仍然使用 updateWords()方法更新页面。因为我们只需要更新 mWord 成员变量，updateWords()方法就可以更新页面。下面就来看看如何更新 mWord 成员变量。

我们知道，mWord 接收的是来自 MainActivity 传入的 WordEntity 对象。而且我们已经得到了该对象在 WordLib 中的位置。因此，当点击"上一页"按钮时，只需要将标记 mWord 位置的 mCurrentIndex 减 1 即可，同样当点击"下一页"按钮时，需要将 mCurrentIndex 加 1。

但所有这些工作我们都需要在一个新的线程中来完成。因此，当用户点击按钮时，我们首先新建一个线程，并在该线程中获取需要更新的词汇所在的位置。

在 onClick(...)方法中创建新线程，并覆盖其 run()方法，如代码清单 9-4 所示。

代码清单 **9-4** 创建新线程（**VocabularyFragment.java**）

```
public class VocabularyFragment extends Fragment implements View.OnClickListener {
    ......
    private Handler mHandler = new Handler() {
        public void handleMessage(Message msg) {
            switch (msg.what) {
                case LAST_WORD:
                    updateWords();
                    break;
                case NEXT_WORD:
                    updateWords();
                    break;
                default:
                    break;
            }
        }
    };

    @Override
    public void onClick(View v) {
        switch (v.getId()) {
            case R.id.last_word:
                new Thread(new Runnable() {
                    @Override
                    public void run() {
                        //创建一个 message
                        //设置 what 字段的值为 LAST_WORD,主要是为了区分不同的 message
                        //调用 Handler 的 message 对象
                        //handler 中的 handlermessage 对象是在主线程中运行的
                        if (mCurrentIndex>0){
                            mCurrentIndex--;
                            mWord = mWordEntityList.get(mCurrentIndex);
                        }
                        Message message = new Message();
                        message.what = LAST_WORD;
```

```
                        mHandler.sendMessage(message);
                    }
                }).start();
                break;
            case R.id.next_word:
                new Thread(new Runnable() {
                    @Override
                    public void run() {
                        //设置 what 字段的值为 NEXT_WORD,用于区分不同的 message
                        if(mCurrentIndex<mWordEntityList.size()-1){
                            mCurrentIndex++;
                            mWord = mWordEntityList.get(mCurrentIndex);
                        }
                        Message message = new Message();
                        message.what = NEXT_WORD;
                        mHandler.sendMessage(message);
                    }
                }).start();
                break;
            default:
                break;
        }
    }
    ......
}
```

运行 NEC Vocab 应用，进入某词条的“词汇详解”后尝试点击“上一页”和“下一页”按钮。

9.3.4 保留 fragment

还有个问题，当进入某词条的“词汇详解”后，点击“上一页”或“下一页”进入另一个词条的“词汇详解”界面。但此时，旋转设备，又回到了最初的那个词条的 “词汇详解”界面。当然，我们已经知道这是由于设备旋转后，MainActivity 进行了重新配置的结果。

虽然使用 savedInstanceState 机制，保存的 mCurrentIndex 数据，可以解决此问题（作为练习，建议读者自行完成）。但 savedInstanceState 机制只适用于可保存的对象数据，对于一些不可保存的数据，如本章后面将学习语音数据以及音频和视频，我们希望在 Activity 创建和销毁时，都需要其实例一直可用，即能在播放中断处继续。然而播放中断的那段时光是无论如何也找不回了。

这个难题该怎么解决呢？

事实上，为应对设备配置变化，fragment 有个特殊方法——**retainInstance**，可确保其实例一直存在。覆盖 VocabularyFragment.java 中的 onCreate(...)方法，并设置 fragment 的属性值，如代码清单 9-5 所示。

代码清单 9-5 调用 setRetainInstance(true)（VocabularyFragment.java）

```
public class VocabularyFragment extends Fragment implements View.OnClickListener {
    ......
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setRetainInstance(true);
        if (getArguments() != null) {
            mWord = (WordEntity)getArguments().getSerializable(ARG_PARAM);
        } else {
            mWord = new WordEntity();
        }
        mWordEntityList = WordLib.get(getActivity()).getWordList();
        for (int i=0; i< mWordEntityList.size(); i++){
            if(mWordEntityList.get(i).getId().equals(mWord.getId())){
                mCurrentIndex = i;
            }
        }
    }
.....
}
```

fragment 的 retainInstance 属性值默认为 **false**，这表明其不会被保留。因此，设备旋转时 fragment 会随托管 activity 一起销毁并重建。调用 setRetainInstance(true) 方法可保留 fragment。已保留的 fragment 不会随 activity 一起被销毁。相反，它会一直保留并在需要时原封不动地传递给新的 activity。

对于已保留的 fragment 实例，其全部实例变量（如 mWord）值也会保持不变，因此可放心继续使用。

运行 NEC Vocab 应用。点击"上一页"或"下一页"进入另一个词条的"词汇详解"界面，然后旋转设备，确认界面不受影响。

9.3.5 清除 Message

在 Android 中，Handler 类应该是静态的，否则，可能发生泄漏。为了避免泄漏，可以使用 Handler 的组件生命周期结束前清除掉 MessageQueue 中的发送给 Handler 的 Message 对象（例如在 Activity、Fragment 或 Service 的 onDestroy()或其他方法中调用 Handler 的 remove 方法）。

我们更多需要的是清除以该 Handler 为 target 的所有 Message（包括 Callback），那么调用如下方法即可：

```
handler.removeCallbacksAndMessages(null);
```

覆盖 VocabularyFragment 的 onDestroy()方法，调用 Handler 的 remove 方法，如代码清单 9-6 所示。

代码清单 9-6 调用 setRetainInstance(true)（VocabularyFragment.java）

```
.....
@Override
public void onDestroy() {
    super.onDestroy();
    mHandler.removeCallbacksAndMessages(null);
}
.....
```

9.4 建立后台服务

目前为止，本书所有的应用都离不开 activity，意味着它们都有着一个或多个看得见的用户界面。如果不给应用提供用户界面，应该怎样做呢？如果不用看、不用操作，只要任务在后台运行就行了，如朗读课文，播放音乐，又该如何做呢？好办，使用服务（service）吧。

本节和下一节，我们将利用 Service 为 NEC Vocab 应用添加一项新功能，允许在后台朗读单词和例句。

Service 是 Android 系统中的四大组件之一，它是一种长生命周期的，没有可视化界面，运行于后台的一种服务程序。根据启动方式的不同，Service 分为两种：

A started service

- 被开启的 service 通过其他组件（如 Activity、Fragment）调用 startService()被创建。
- 这种 service 可以无限地运行下去，必须调用 stopSelf()方法或者其他组件调用 stopService()方法来停止它。
- 当 service 被停止时，系统会销毁它。

A bound service

- 被绑定的 service 是当其他组件（如 Activity、Fragment）调用 bindService()来创建的。
- 组件可以通过一个 IBinder 接口和 service 进行通信。
- 组件可以通过 unbindService()方法来关闭这种连接。
- 一个 service 可以同时和多个组件绑定，当多个组件都解除绑定之后，系统会销毁 service。

NEC Vocab 应用主要使用 Started Service。

9.4.1 创建 Service

Android 中的 Service（服务）是运行在后台的服务，是不可见，没有界面的东西。Service 和其他组件一样，都是运行在主线程中，因此不能用它来做耗时的请求或者动作。你可以在 Service 中开启一个线程，在线程中做耗时动作。

创建 Service

首先来创建服务。在 Project 视图中选中 com.studio.aime.necvocab，右键 **New→Service→Service**，在弹出的对话框的 Class Name:处填写 TTSService，点击 Finish，这样就完成了服务的创建。

可以看到，向导为我们创建了一个名为 TTSService 的类，该类继承了 Service，如代码清单 9-7 所示。

代码清单 **9-7** 向导创建的 **TTSService**（**TTSService.java**）

```
public class TTSService extends Service {
    public TTSService() {
    }

    @Override
    public IBinder onBind(Intent intent) {
        // TODO: Return the communication channel to the service.
        throw new UnsupportedOperationException("Not yet implemented");
        return null;
    }
}
```

其中，onBind(...)函数是 Service 基类中的唯一抽象方法，所有子类都必须重写实现，此函数的返回值是针对 Bound Service 类型的 Service 才有用的，在 Started Service 类型中，此函数直接返回 null 即可。因此代码中我们删除了 throw new ...一行，取而代之的是 return null。

此外，向导还在 AndroidManifest.xml 中声明了 TTSService：

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

这样就实现了最基本的 Service。实际上，Service 和 activity 有点像。Service 也是一个 context

（Service 是 Context 的子类），并能够响应 intent。

9.4.2 覆盖 Service 的生命周期方法

与 Activity 类似，Service 也有自己的生命周期函数，在不同的时刻，系统会调用对应的 Service 生命周期函数，不过与 Activity 声明周期相比，Service 的声明周期更加简单，我们通过官方给出的一张图片来体会一下：

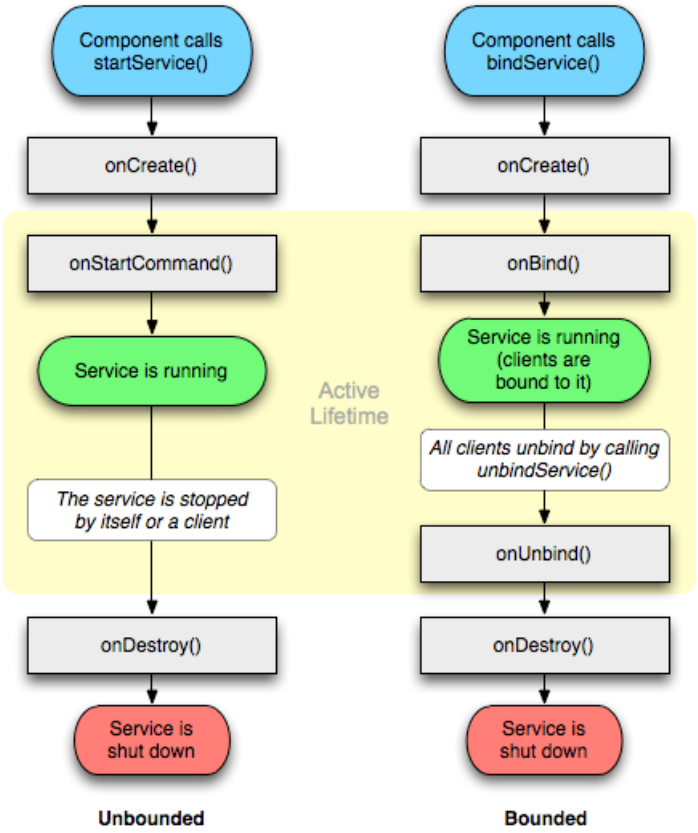

图 9-7 Service 的生命周期方法

可以看出，onBind(...)函数的返回值是针对 Bound Service 类型的 Service 才有用的，在 Started Service 类型中，此函数直接返回 null 即可。onCreate(...)、onStartCommand(...)和 onDestroy()都是 Started Service 相应生命周期阶段的回调函数。

下面我们覆盖 TTSService 的生命周期方法，如代码清单 9-8 所示。

代码清单 **9-8** 覆盖 **Service** 的生命周期方法（**TTSService.java**）

```
public class TTSService extends Service {
    public TTSService() {
    }
```

```
    @Override
    public void onCreate() {
        super.onCreate();
        Log.d("TTSService", "执行 onCreate");
    }
    @Override
    public int onStartCommand(Intent intent, int flags, int startId) {
        Log.d("TTSService", "执行 onStartCommand");
        return super.onStartCommand(intent, flags, startId);
    }
    @Override
    public void onDestroy() {
        super.onDestroy();
        Log.d("TTSService", "执行 onDestroy");
    }

    @Override
    public IBinder onBind(Intent intent) {
        // TODO: Return the communication channel to the service.
        return null;
    }
}
```

9.4.3 启动服务

接下来我们在 VocabularyFragment 中，添加启动服务的代码。

我们希望点击“词汇详解”中的英文单词和例句的时，TTSService 能够读出相应的单词和例句。然而，VocabularyFragment 中的英文单词（mEnglishTextView）和例句（mExampleGratiaView）是 TextView 类型，因此，需要将其设置为可点击，同时为其设置监听器，如代码清单 9-9 所示。

代码清单 9-9 为 TextView 设置监听器（VocabularyFragment.java）

```
public class VocabularyFragment extends Fragment implements View.OnClickListener {
    ......
    @Override
    public View onCreateView(LayoutInflater inflater, ViewGroup container,
                             Bundle savedInstanceState) {
        View view = inflater.inflate(R.layout.fragment_vocabulary, container, false);
        ......
        mEnglishTextView.setClickable(true);
        mExampleGratiaView.setClickable(true);
        mEnglishTextView.setOnClickListener(this);
        mExampleGratiaView.setOnClickListener(this);
        ......
}
```

然后，在 VocabularyFragment 覆盖的 onClick 方法中添加两条 case 语句，在该语句中添加

启动服务的代码，如代码清单 9-10 所示。

代码清单 **9-10** 添加服务启动代码（**VocabularyFragment.java**）

```
public class VocabularyFragment extends Fragment implements View.OnClickListener {
    ......
    private Intent mIntentService;
    ......
    @Override
    public View onCreateView(LayoutInflater inflater, ViewGroup container,
                             Bundle savedInstanceState) {
        ......
        mIntentService = new Intent(getActivity(), TTSService.class);
        updateWords();
        return view;
    }
    ......
    @Override
    public void onClick(View v) {
        switch (v.getId()) {
            case R.id.vocabulary_english:
                getActivity().startService(mIntentService);
                break;
            case R.id.example_gratia:
                getActivity().startService(mIntentService);
                break;
            ......
}
```

在 LogCat 中，点击位于 LogCat 右上角的筛选下拉菜单，并选择 Edit Filter Configuration，创建一个消息过滤器。在 Filter Name 字段和 Log Tag 字段输入 TTSService，（见图 9-8）。

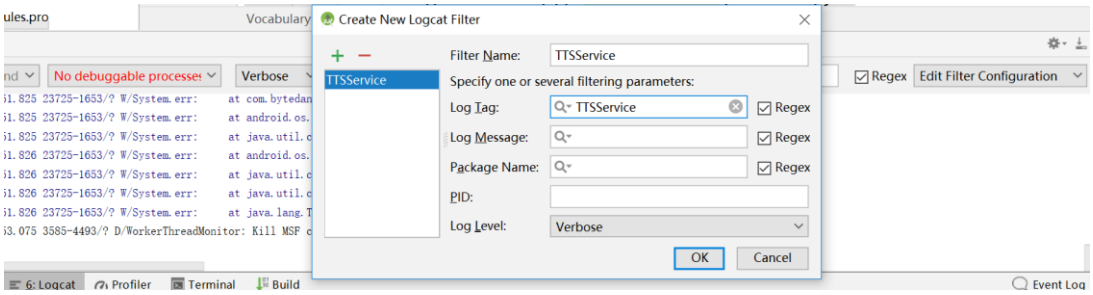

图 9-8 在 LogCat 中创建过滤器

点击 OK，系统将在新出现的标签页窗口中仅显示 TTSService 的日志信息。

运行应用，进入“词汇详解”，点击词汇，查看 LogCat 窗口，可看到类似图 9-9 所示的结果。

图 9-9 应用启动 Service 调用的两个个生命周期方法

再点击例句，可以看到仅调用了 onStartCommand(...)方法。选择另一词条，再次点击词汇、例句，可以看到，仍然只是调用了 onStartCommand(...)方法。退出应用，也未见到调用 onDestroy()方法，如图 9-10 所示。

图 9-10 点击其他词汇，甚至退出应用也未见到调用 onDestroy()方法

这是怎么回事?

事实上

- 当 Activity 或 Fragment 通过调用 startService(...)启动 Service 后，如果 Service 是第一次启动，首先会执行 onCreate()回调，然后再执行 onStartCommand(...)。
- 当 Activity 或 Fragment 再次调用 startService(...)，将只执行 onStartCommand(...)，因为此时 Service 已经创建了，无需执行 onCreate()回调。
- 通过 startService(...)启动 Service 后，此时 Service 的生命周期与 Activity 或 Fragment 本身甚至整个应用都没有任何关系。因此，我们即使退出 NEC Vocab 应用未见到调用 onDestroy()方法。该 Service 将会一直在后台运行，不管对应程序是否在运行，直到被调用 stopService(...)，或自身的 stopSelf(...)方法，才能停止此 Service。当然如果系统资源不足，android 系统也可能结束服务。
- 无论多少次的 startService(...)，只需要一次 stopService(...)或 stopSelf(...)方法即可将此 Service 终止，执行 onDestroy()函数。

onStartCommand

下面重点关注一下 onStartCommand(Intent intent, int flags, int startId)方法。

其中参数 flags 默认情况下是 0，对应的常量名为 START_STICKY_COMPATIBILITY。startId 是一个唯一的整型，用于表示此次 Client 执行 startService(...)的请求标识，在多次 startService(...)的情况下，呈现 0,1,2....递增。另外，此函数具有一个 int 型的返回值，具体的可选值及含义如下：

START_NOT_STICKY：当 Service 因为内存不足而被系统 kill 后，接下来未来的某个时间内，即使系统内存足够可用，系统也不会尝试重新创建此 Service。除非程序中 Client 明确再次调用 startService(...)启动此 Service。

START_STICKY：当 Service 因为内存不足而被系统 kill 后，接下来未来的某个时间内，当系统内存足够可用的情况下，系统将会尝试重新创建此 Service，一旦创建成功后将回调 onStartCommand(...)方法，但其中的 Intent 将是 null，pendingintent 除外。

START_REDELIVER_INTENT：与 START_STICKY 唯一不同的是，回调 onStartCommand(...)方法时，其中的 Intent 将是非空，将是最后一次调用 startService(...)中的 intent。

START_STICKY_COMPATIBILITY：START_STICKY 的兼容版本。此值一般不会使用。

9.4.4 停止服务

从图 9-10 我们看到，即使退出 NEC Vocab 应用未见到调用 onDestroy()方法。该 TTSService 将会一直在后台运行，不管对应程序是否在运行，直到调用 stopService(…)，或自身的 stopSelf(…)方法，才能停止此 Service。

那么什么时候让 Service 停止合适呢？这当然和应用的需求相关，例如音乐播放器，可以添加一个按钮，在你不想听时调用 stopService(…)，强制停止 Service。但就 NEC Vocab 应用而言，我们可能更希望在系统读完单词或例句之后一段时间自动停止 Service。这就要使用 stopSelf(…)方法。

但在这段时间之内，可能用户已经退出了 NEC Vocab 应用。在没有 Activity 运行的情况下，得想个办法停止他，比如说，设置一个 15 秒延时器。即读完后 15 秒，如果用户没有进一步的朗读要求，就自动停止 Service。

一种方式是调用 Handler 的 PostDelayed(…)方法。

使用 Handler.PostDelayed 延迟执行任务

接下来我们就使用 Handler.PostDelayed(…)来延迟执行任务。为此：

- 首先需要创建一个 Handler 对象

```
Handler handler=new Handler();
```

- 第二步创建一个 Runnable 对象

```
Runnable runnable=new Runnable(){
    @Override
    public void run() {
     // TODO Auto-generated method stub
     //要做的事情，比如调用 stopSelf()
    }
};
```

- 接着使用 PostDelayed(…)方法，15 秒后调用此 Runnable 对象

```
handler.postDelayed(runnable, 15*1000);
```

这实际上也就实现了一个 15s 的一个定时器。

- 最后如果想要关闭此定时器，可以使用：

```
handler.removeCallbacks(runnable);
```

或

```
handler.removeCallbacksAndMessages(null);
```

清除以该 Handler 的包括 Callback 的所有 Message。

下面我们就将上述方法添加的 TTSService.java 文件中，如代码清单 9-11 所示。

代码清单 **9-11** 添加定时器来启动 **stopSelf()**（**TTSService.java**）

```
public class TTSService extends Service {
    public TTSService() {
    }
    private Handler handler;
    @Override
    public void onCreate() {
        super.onCreate();
        handler = new Handler();
        Log.d("TTSService", "执行 onCreate");
    }
    @Override
    public int onStartCommand(Intent intent, int flags, int startId) {
        handler.removeCallbacksAndMessages(null);
        handler.postDelayed(new Runnable() {
            @Override
            public void run() {
                stopSelf();
```

```
            }
        }, 15*1000);
        Log.d("TTSService", "执行 onStartCommand");
        return TTSService.START_NOT_STICKY;
        return super.onStartCommand(intent, flags, startId);
    }
    ......
}
```

注意，我们在 onStartCommand(...)中首先关闭定时器，以便从新开始计时。在 PostDelayed(...)方法中使用了匿名内部类创建一个 Runnable 对象，并在其 run()方法中调用 stopSelf()。

现在运行 NEC Vocab 应用，进入“词汇详解”，点击英文单词或例句等，观察 Logcat 的运行结果。

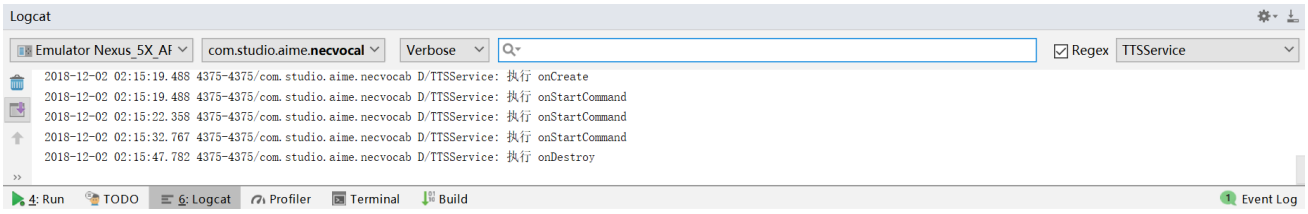

图 9-11 15 秒后停止 Service

9.5 实现语音播报服务

查看 LogCat 日志观察 Service 是很乏味的！但上一节添加的代码着实令人兴奋！为什么？利用 Service 能做什么呢？

就像一个零售店。商店内有两个地方可以工作：与客户打交道的前台，以及不与客户接触的后台。取决于零售店的规模，工作后台可大可小。

目前为止，所有应用代码都在 activity 中运行。activity 就是 Android 应用的前台。所有应用代码都专注于提供良好的用户视觉体验。

服务就是 Android 应用的后台。用户无需关心后台发生的一切。即使前台关闭，activity 消失好久了，后台服务依然可以持续不断地工作。因此可以启动一个 Service 来播放语音、音乐，或者记录你地理信息位置的改变，或者启动一个 Service 来运行并一直监听某种动作。即使离开当前应用后（打开其他应用或退回主屏幕），服务依然可以在后台运行。

下面我们就利用 Service 来实现语音。

TextToSpeech 简称 TTS，是 Android 1.6 版本以后添加的比较重要的新功能。将所指定的文本转成不同语言音频输出。它可以方便的嵌入到游戏或者应用程序中，增强用户体验。遗憾的是暂时没有我们伟大的中文。对于 NEC Vocab 而言，只要能说英语，就足够了（如果要说汉语，百度、讯飞也许是更好的选择）。

创建 TTS 对象

使用 TextToSpeech 引擎首先要实现 TextToSpeech.OnInitListener 接口，并作为参数传递给构造函数。这里需要实现其唯一的抽象方法 onInit(...)，当然目前我们在该方法中什么都不做。同事通过构造函数 TextToSpeech(...)创建 TextToSpeech 对象，并通过 OnInitListener 监听该对象是否创建成功。如代码清单 9-12 所示。

代码清单 **9-12** 实现 **TextToSpeech.OnInitListener** 接口（**TTSService.java**）

```
public class TTSService extends Service implements TextToSpeech.OnInitListener{
    public TTSService() {
    }
    private Handler handler;
    private TextToSpeech mTts;
    @Override
    public void onCreate() {
        super.onCreate();
        mTts = new TextToSpeech(getApplicationContext(), this);
        handler = new Handler();
        Log.d("TTSService", "执行 onCreate");
    }
    @Override
    public int onStartCommand(Intent intent, int flags, int startId) {
        ......
    }
    @Override
    public void onInit(int status) {
        if (status == TextToSpeech.SUCCESS) {
            //目前什么也不做
        }
    }
    ......
}
```

TextToSpeech 实体和 OnInitListener 都需要引用当前 Activity 的 Context 作为构造参数。OnInitListener 的用处是通知系统当前 TTS Engine 已经加载完成，并处于可用状态。

根据需求设置语言参数：

早在 Google I/O 大会上，官方给出了一段关于应用这项功能的鲜活体验，将翻译结果直接

通过五种不同国家语言的语音输出。加载语言的方法非常简单：

```
mTts.setLanguage(Locale.US);
```

上边代码表示当前 TTS 实体加载美式英语。其参数并没有指示某种语言的名称，而是利用国家代码来表示，这样做的好处是不但可以确定语言的选择，而且可以根据地区的不同而有所区别。例如：英语作为最广泛被应用的语种，在多个不同的地区都有一定的差别。可以通过

```
TextToSpeech.LANG_NOT_SUPPORTED
TextToSpeech.LANG_MISSING_DATA
```

判断当前系统是否支持某个地区的语言资源或数据是否丢失。

此外，还可以通过

```
mTts.setPitch(1.0f);
mTts.setSpeechRate(0.7f);
```

设置音调（值越大声音越尖（女生），值越小则变成男声,1.0 是常规）、语速（1.0 是常规，越大，越快）等。

通常，可以在 onInit(…)方法中设置语言参数，如代码清单 9-13 所示。

代码清单 **9-13** 设置语言参数，并判断是否支持该语言（**TTSService.java**）

```
public class TTSService extends Service implements TextToSpeech.OnInitListener{
    public TTSService() {
    }
    private Handler handler;
    private TextToSpeech mTts;
    private boolean isInit;
    ......
    @Override
    public void onInit(int status) {
        if (status == TextToSpeech.SUCCESS) {
            int result = mTts.setLanguage(Locale.UK);
            mTts.setPitch(1.0f);
            mTts.setSpeechRate(0.7f);
            if (result != TextToSpeech.LANG_MISSING_DATA &&
                    result != TextToSpeech.LANG_NOT_SUPPORTED) {
                isInit = true;
            }
        }
    }
    ......
}
```

如果布尔类型变量 isInit 为 true，则表明 TTS 初始化成功。这样就实现了 TextToSpeech 的初始化和参数配置。下面可开始“说话”了。

执行 Speak 的具体方法：

TextToSpeech 说什么？当然是由"词汇详解"决定。因此，需要通过 intent.getStringExtra(...)来获取说话内容，并利用 Speak(...)方法可以直接在应用程序中发挥强大的语音功能。没错，用起来就是这么简单：

代码清单 9-14 开始"说话"（TTSService.java）

```
public class TTSService extends Service implements TextToSpeech.OnInitListener{
    ......
    public static final String EXTRA_WORD = "word";
    private String word;
    ......
    @Override
    public int onStartCommand(Intent intent, int flags, int startId) {
        handler.removeCallbacksAndMessages(null);
        word = intent.getStringExtra(EXTRA_WORD);
        if (isInit) {
            speak();
        }
        ......
    }
    ......
    private void speak() {
        if (mTts != null) {
            mTts.speak(word, TextToSpeech.QUEUE_FLUSH, null,null);
        }
    }
    ......
}
```

回收 TTS：

当确定应用程序不再需要 TTS 的相关功能后，可以在 Service 的 OnDestroy()方法中调用 shutDown()释放当前 TTS 实体所占用的资源，如代码清单 9-15 所示（为了便于读者对照，这里给出 TTSService 的完整代码）。

代码清单 9-15 回收 TTS（TTSService.java）

```
public class TTSService extends Service implements TextToSpeech.OnInitListener{
    public TTSService() {
    }
    private Handler handler;
    public static final String EXTRA_WORD = "word";
    private String word;
    private TextToSpeech mTts;
    private boolean isInit;
    @Override
```

```
public void onCreate() {
    super.onCreate();
    mTts = new TextToSpeech(getApplicationContext(), this);
    handler = new Handler();
    Log.d("TTSService", "执行 onCreate");
}
@Override
public int onStartCommand(Intent intent, int flags, int startId) {
    handler.removeCallbacksAndMessages(null);
    word = intent.getStringExtra(EXTRA_WORD);
    if (isInit) {
        speak();
    }
    handler.postDelayed(new Runnable() {

        @Override
        public void run() {
            stopSelf();
        }
    }, 15*1000);
    Log.d("TTSService", "执行 onStartCommand");
    return TTSService.START_NOT_STICKY;
}

@Override
public void onInit(int status) {

    if (status == TextToSpeech.SUCCESS) {
        int result = mTts.setLanguage(Locale.UK);
        mTts.setPitch(1.0f);
        mTts.setSpeechRate(0.7f);
        if (result != TextToSpeech.LANG_MISSING_DATA &&
                result != TextToSpeech.LANG_NOT_SUPPORTED) {
            isInit = true;
        }
    }
}

private void speak() {
    if (mTts != null) {
        mTts.speak(word, TextToSpeech.QUEUE_FLUSH, null,null);
    }
}

@Override
public void onDestroy() {
    if (mTts != null) {
        mTts.stop();
        mTts.shutdown();
    }
    super.onDestroy();
    Log.d("TTSService", "执行 onDestroy");
}

@Override
```

```
    public IBinder onBind(Intent intent) {
        // TODO: Return the communication channel to the service.
        return null;
    }
}
```

启动 TTS

最后，我们从 VocabularyFragment.java 文件中启动语音播报。因为之前一切已经准备完毕，这里只需要添加两行代码即可：

代码清单 9-16 启动语音播报（VocabularyFragment.java）

```
public class VocabularyFragment extends Fragment implements View.OnClickListener {
    ......
    @Override
    public void onClick(View v) {
        switch (v.getId()) {
            case R.id.vocabulary_english:
                mIntentService.putExtra(TTSService.EXTRA_WORD, mWord.getEnglish());
                getActivity().startService(mIntentService);
                break;
            case R.id.example_gratia:
                mIntentService.putExtra(TTSService.EXTRA_WORD, mWord.getExample());
                getActivity().startService(mIntentService);
                break;
            ......
}
```

运行 NEC Vocab 应用，进入“词汇详解”，点击词汇和例句，听听“牛津”英语。