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

6 实现词汇列表

6.1 本章目标

本讲，我们将更新 NEC Vocab 应用，使其包含一个“词汇表”，并用“词汇表”替代“词汇详解”作为“词汇”Tab 的布局。当用户选择“词汇”Tab 时，出现如图 6-1 左边所示的“词汇表”，当点击某词条时出现右边的“词汇详解”。

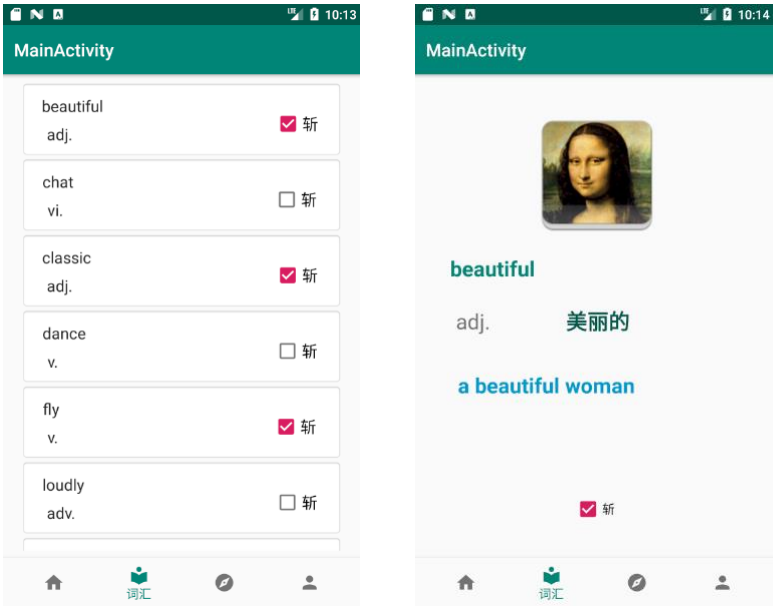

图 6-1 词汇列表

通过本章对 NEC Vocab 应用的升级，我们将学习如何利用 RecyclerView 显示列表以及利用接口回调传递数据。

6.2 词汇模块的 MVC 设计模式

NEC Vocab 应用的词汇表将显示每一个词条的英文单词、词性和是否掌握（斩）。词汇模块

可以用如图 6-2 所示的 MVC 对象图描述。

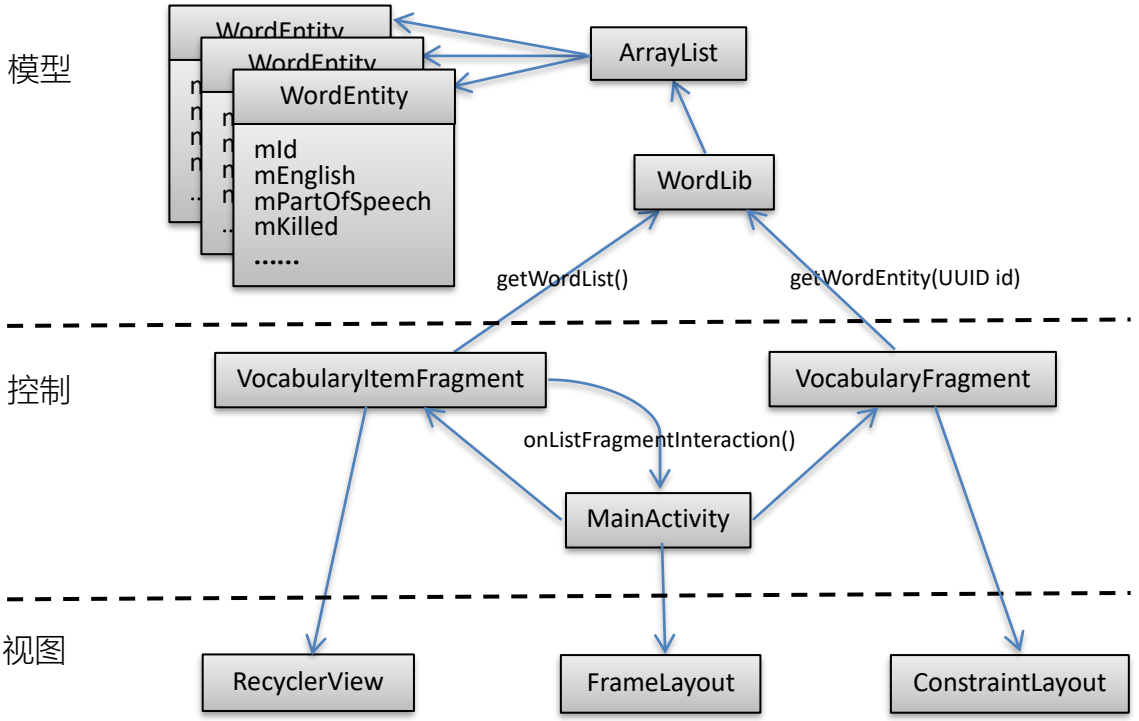

图 6-2 NEC Vocab 应用词汇模块的对象图

从图中可以看出，NEC Vocab 应用的词汇模块对象按模型、控制器和视图的类别被分为三个部分。

词汇模块的模型层由一个 WordEntity 和 **WordLib** 类构成。WordEntity 实例代表了某个单词。WordLib 对象是一个数据集中存储池，用来存储 WordEntity 对象。

控制层由 MainActivity、VocabularyItemFragment 和 VocabularyFragment 构成。其中新增的 **VocabularyItemFragment** 用于显示 WordLib，负责创建并管理用户界面，与模型对象进行交互。

视图层显示了与 MainActivity、VocabularyItemFragment 和 VocabularyFragment 关联的视图对象。MainActivity 的视图为 VocabularyItemFragment 和 VocabularyFragment 提供了一个区域（FrameLayout）。VocabularyItemFragment 视图由一个 **RecyclerView** 组成。当点击某个词条时，VocabularyItemFragment 利用接口方法 onListFragmentInteraction()将该词条传递给 MainActivity。MainActivity 再启动 VocabularyFragment 显示该词条的详解。

WordLib 和 VocabularyItemFragment 是本章我们要开发的类。同时本章还将对 MainActivity 进行升级。

接下来，我们先升级 NEC Vocab 应用的模型层。

6.3 创建词汇表

我们需要创建可容纳多个 WordEntity 的列表类 WordLib。在此之前，我们先介绍几个要用到的 Java 概念。

6.3.1 抽象类和接口

接口和抽象类是 Android 中常用的两个 Java 概念。到目前为止，我们开发的所有的 Activity 都是抽象类 AppCompatActivity 的实现类，都实现了 AppCompatActivity 的抽象方法 OnCreate()。本章开始，我们还将大量使用接口。关于抽象类和接口，简单地说，接口是抽象类的一种特例。不过具体使用中，什么情况用抽象类，什么情况用接口呢？

下面我们以门——Door 为例，来说明二者的用法。

我们知道 Door 至少有两个方法 open()和 close()。把它抽象出来，可以有两种方法：

抽象成抽象类

```
abstract class Door{
    void open() {}
    void close() {}
}
```

抽象成接口

```
interface Door{
    void open();
    void close();
}
```

看起来两者好像没啥区别。现在我们要加一个门铃功能。没问题，那就加上：

抽象类

```
abstract class Door{
        void open(){}
        void close(){}
        void bilibili(){}
}
```

接口

```
interface Door{
        void open();
        void close();
        void bilibili();
}
```

那么问题来了，所有的门都有开关门功能，是不是所有的门都有门铃呢？如果我们用抽象类描述 Door，那么所有实现 Door 的类都得具有门铃方法，显然与实际不符。

所以，我们应该把门铃理解成一种附件，一种可以加到门上面的一种点缀。对于这种附件形式的类，我们就可以用接口来表示。

所以，啥叫接口，就是在门上凿出一个门铃大小的洞，用来放门铃的，就跟电脑上的 usb 接口一样，一个洞嘛！

我们可以定义一个铃声接口：

```
interface Ring{
        void bilibili();
  }
```

定义一个 Door 类 ：

```
abstract class Door{
        void open(){}
        void close(){}
}
```

这样，一个普通的没有门铃的门可以定义为：

```
class DoorPlain extends Door{
}
```

如果有一个带铃铛的门，可以这样来写：

```
  class DoorWithRing extends Door implements Ring{
        public void bilibili() {
              // 钻个洞，放个门铃
              ……
        }
  }
```

现在我们要在门上装一个猫眼，咋装呢，很简单。买个猫眼：

```
interface CatEye{
      void see();
}
```

把猫眼装进门上的洞里：

```
class DoorWithRingAndCatEye extends Door implements Ring, CatEye{
        public void see() {
              // 钻个洞，放个门铃
              ……
        }

        public void bilibili() {
```

```
            // 钻个洞，放个猫眼
            ……
        }
    }
```

一个门上可以打无数的洞，也就是可以实现多个接口。

有人会问，门铃，猫眼，这些东西不也是类吗，为啥不做成类，做成接口呢?

对的，当然可以做成一个正常的类，有其属性什么的，这样猫眼，门铃和门的关系就变了，变成聚合关系了，不是组合，因为门没了门铃和猫眼的门还是门。

所以，对于那种功能单一（响铃，猫眼），又需要拿来作为一个附件附加到基本类上的类，我们就把它定义成接口。

总结一下。

1. 抽象类和接口的定义

抽象类

- 抽象类不能直接创建实例，是给子类继承的类；
- 抽象类的子类如果不是抽象类，子类必须实现该抽象类的抽象方法；
- 抽象类可以有一个或多个抽象方法，，也可以有具体实现的方法和成员变量；

接口

- 接口不能直接实例化；
- 接口只能定义或者声明常量和抽象方法，不能定义变量或者具体实现的方法，定义的常量默认有 public, final, static 属性，声明的方法默认具有 public 和 abstract 的属性。

2. 接口与抽象类的区别

- 接口中的所有方法都是抽象的，而抽象类可以定义带有方法体的方法。
- 一个类可以实现多个接口，但是只能继承一个抽象父类。
- 接口与实现它的类不构成类的继承体系，即接口不是类体系的一部分，因此不相关的类也可以实现相同的接口（如计算机类和手机类都可以实现耳机接口）。而抽象类是属于一个类的继承体系，并且一般位于类体系的顶层。

3. 接口的好处

- 类通过实现多个接口可以实现多重继承，这是接口最重要的作用，也是使用接口的最重要的原因。

- 能够抽象出不相关类之间的相似性，而没有强行形成类的继承关系。
- 使用接口，可以同时获得抽象类以及接口的优势，所以如果要创建的类体系的基础类不需要定义任何成员变量，并且不需要给出任何方法的完整定义，则应该将继承类定义为接口。只有在必须使用方法定义或成员变量时，才考虑采用抽象类。

4. List 接口

List 接口是 Java 中经常用到的接口，可以存放任意的数据。在 List 接口中内容是允许重复的，可以通过索引值随意访问元素。

本书将用到的 List 接口方法

- public void add(int index,E element) 在指定的位置增加元素。
- E get(int index) 返回指定位置的元素。
- public E remove(int index) 删除指定位置的元素。
- public E set(int index, E element) 替换指定位置的元素。

List 常用的实现是 ArrayList 和 LinkedList，本章我们使用 ArrayList。ArrayList 采用了常规的 Java 数组存储列表元素，实现了可变大小的数组。

5. 使用 List 存放集合的引用

可以使用接口（这里是 List）类型存放集合的引用：

```
private List<WordEntity> mWordList;
......
private WordLib(Context context) {
      mWordList = new ArrayList<>();
}
```

注意变量声明："**private List<WordEntity> mWordList;**"。当在程序中使用有序列表 List 时，一旦构建了集合，就不需要知道究竟使用了哪种实现。只有在构建集合对象时，使用具体的类才有意义。利用这种方式，一旦改变了想法，可以轻松地使用另外一种不同的实现。只需要对程序的一个地方，即调用构造方法的地方，做出修改。如果觉得 LinkedList 是一个更好的选择，就将代码修改为：

```
private List<WordEntity> mWordList;
......
private WordLib(Context context) {
      mWordList = new LinkedList<>();
     }
```

6. 菱形符号：“<>”

在 WordLib 中使用了菱形符号，“**<>**”，这是在 Java 7 中引入的。**Java SE7** 及以后的版本中，构造函数中可以省略泛型类型，省略的类型可以通过变量的类型推断得出。这里，因变量声明：“List<WordEntity> ITEMS;”和为泛型参数指定了 WordEntity，所以，编译器将认为该 ArrayList 的泛型参数是 WordEntity。

6.3.2 单例与数据集中存储

在 NEC Vocab 应用中，WordEntity 对象将存储在一个单例里。单例是特殊的 **java** 类，在创建实例时，一个类仅允许创建一个实例。

应用能够在内存里存在多久，单例就能存在多久，因此将对象列表保存在单例里可保持 WordEntity 对象数据的一直存在，不管 activity、fragment 及它们的生命周期发生什么变化。使用单例时也要小心，当 Android 从内存中删除应用程序的时候，他们将被销毁。WordLib 单例不是解决数据长期存储的方案，但它给应用提供了一个存储 WordEntity 对象数据的地方，并提供了一种在控制器类之间的传递数据的快捷方式。

在创建单列之前，我们先回顾几个 java 概念。

1. static

static 表示“全局”或者“静态”的意思，用来修饰成员变量和成员方法，也可以形成静态 static 代码块，但是 Java 语言中没有全局变量的概念。

被 static 修饰的成员变量和成员方法独立于该类的任何对象。也就是说，它不依赖类特定的实例，被类的所有实例共享。只要这个类被加载，Java 虚拟机就能根据类名在运行时数据区的方法区内定找到他们。因此，static 对象可以在它的任何对象创建之前访问，无需引用任何对象。

例如我们可以定义一个静态变量 sWordLib 和一个静态方法 get(Context context)。

```
private static WordLib sWordLib;
public static WordLib get(Context context){
    if (sWordLib == null) {
        sWordLib = new WordLib(context);
    }
    return sWordLib;
}
```

我们可以直接使用 get 方法：

```
WordLib.get(getActivity())
```

而无需创建任何对象。

2. static{}静态代码块

静态代码块，在类的构造方法之前执行，并且只会在第一次执行，之后都不会执行的方法代码块。例如我们可以事先为 NEC Vocab 准备好若干词条：

```
private static List<WordEntity> sWords = new ArrayList<>();
static{
    sWords.add(new WordEntity("beautiful","adj.","美丽的",
            "a beautiful woman",R.drawable.beautiful));
    sWords.add(new WordEntity("chat","vi.","谈话",
            "What were you chatting about?",R.drawable.chat));
    sWords.add(new WordEntity("classic","adj.","经典的",
            "a classic novel",R.drawable.classicbook));
    sWords.add(new WordEntity("dance","v.","跳舞",
            "They stayed up all night singing and dancing.",R.drawable.dance));
    sWords.add(new WordEntity("fly","v.","飞行",
            "The dog is learning to fly.",R.drawable.fly));
    sWords.add(new WordEntity("loudly","adv.","  大声地",
            "They were talking loudly.",R.drawable.loudly));
}
```

这样，在程序运行之前，我们就已经拥有了若干词汇。

3. static 和 final 一起使用

static final 用来修饰成员变量和成员方法，可以理解为“全局变量”。

对于变量，表示一旦给值就不可修改，并且通过类名可以访问。

对于方法，表示不可覆盖，并且可以通过类名直接访问。

注意：对于被 static 和 final 修饰过的实例常量，实例本身不能再改变了，但对于一些容器类型（比如，ArrayList、HashMap）的实例变量，不可以改变容器变量本身，但可以修改容器中存放的对象。

4. 创建单列

要创建单例，需要创建一个带有私有构造方法和 **get()**方法的类，其中 get()方法返回实例。如果实例已经存在，get()方法则直接返回它；如果实例还不存在，get()方法会调用构造方法来创建它。例如：

```
public class WordLib {
    private static WordLib sWordLib;
    public static WordLib get(Context context){
```

```
            if (sWordLib == null) {
                sWordLib = new WordLib(context);
            }
            return sWordLib;
        }
        private WordLib(Context context){
        }
}
```

首先注意 sWordLib 变量的前缀 **s**。这是 Android 开发的命名约定，通过该前缀很容易知道 sWordLib 是一个静态变量。

同时注意，**WordLib** 类的构造方法是私有的。这说明其他类不能直接创建一个 WordLib，必须通过 **get()**方法创建。

最后，在 get()方法里，我们传送了一个 **Context** 对象。目前我们没有使用 Context 对象，我们将在第 7 讲使用它。

6.3.3 创建 WordLib 类

有了上述概念，我们来创建 WordLib。

右键单击 com.studio.aime.necvocab 类包，选择 **New→Java Class** 菜单项，在随后出现的对话框中将类名命名为 WordLib，然后单击 **OK**。

现在，我们需要一个单例来集中存储数据，并添加获取词汇表和获取单个词条的方法，如代码清单 6-1 所示。

代码清单 6-1 创建可容纳 WordEntity 对象的 WordLib（WordLib.java）

```
public class WordLib {
    private static WordLib sWordLib;
    private List<WordEntity> mWordList;//创建存储数量类型为 WordEntity 对象的序列
    public static WordLib get(Context context){
        if (sWordLib == null) {
            sWordLib = new WordLib(context);
        }
        return sWordLib;
    }
    private WordLib(Context context){
    }
    public List<WordEntity> getWordList() {//获取 WordEntity 词汇表
        return mWordList;
    }
    public WordEntity getWordEntity(UUID id){//通过 UUID 获取单个词汇
        for (WordEntity wordEntity : mWordList){
            if (wordEntity.getId().equals(id)){
                return wordEntity;
            }
```

```
            }
            return null;
        }
    }
```

现在，我们暂时先在 WordLib 中，事先批量存入 120 个 WordEntity 对象（很糟糕的设计方式，仅仅为了教学中尽早完成示例），我们可以利用一个 static{}静态代码块来完成此项任务，如代码清单 6-2 所示。当然，这里要将一些准备好的 png 格式的图片（最好不要超过 256×256 像素）事先存放在 res/drawable 目录下，并将这些图片命名为其英文名称。

代码清单 6-2 创建 120 个 WordEntity 对象（WordLib.java）

```
public class WordLib {
    private static WordLib sWordLib;
    private List<WordEntity> mWordList;//创建存储数量类型为 WordEntity 对象的序列
    public static WordLib get(Context context){
        if (sWordLib == null) {
            sWordLib = new WordLib(context);
        }
        return sWordLib;
    }
    private WordLib(Context context) {
        mWordList = new ArrayList<>();
        mWordList.addAll(sWords);
    }
    public List<WordEntity> getWordList() {
        return mWordList;
    }
    public WordEntity getWordEntity(UUID id){
        for (WordEntity wordEntity : mWordList){
            if (wordEntity.getId().equals(id)){
                return wordEntity;
            }
        }
        return null;
    }
    private static List<WordEntity> sWords = new ArrayList<>();
    static{
        for (int i = 1; i <= 20; i++) {
            sWords.add(new WordEntity("beautiful","adj.","美丽的",
                    "a beautiful woman",R.drawable.beautiful));
            sWords.add(new WordEntity("chat","vi.","谈话",
                    "What were you chatting about?",R.drawable.chat));
            sWords.add(new WordEntity("classic","adj.","经典的",
                    "a classic novel",R.drawable.classicbook));
            sWords.add(new WordEntity("dance","v.","跳舞",
                    "They stayed up all night singing and dancing.",R.drawable.dance));
            sWords.add(new WordEntity("fly","v.","飞行",
                    "The dog is learning to fly.",R.drawable.fly));
            sWords.add(new WordEntity("loudly","adv.","  大声地",
```

```
                "They were talking loudly.",R.drawable.loudly));
            }
        }
    }
```

现在我们拥有了一个装满数据的模型层和 120 个可在屏幕上显示的 WordEntity。

6.4 引入 RecyclerView

现在我们希望 VocabularyItemFragment 将 WordLib 列表显示给用户，每一个列表都应该包含一个 WordEntity 实例。为此，我们将使用 **RecyclerView**。

从 Android 5.0 开始，谷歌推出了一个用于展示大量数据的强大而灵活的新控件 RecylerView。RecyclerView 的出现让很多开源项目，如横向滚动的 ListView、横向滚动的 GridView 和瀑布流控件等被废弃，因为 RecyclerView 能够轻易实现所有这些功能。

6.4.1 RecyclerView 基本原理

RecyclerView 是 ViewGroup 的一个子类，它显示 View 子对象的列表。简单起见，我们这里实现一个简单列表项的显示：每个列表项只显示 WordEntity 的英文单词，词性和是否掌握。

图 6-3 显示了 120 个 TextView。通过滚动屏幕，还可以显示更多，直至所有 120 条 TextView。

图 6-3 带有子 TextView 的 RecyclerView

那么这些 View 对象是否事先已经准备好了？如果是这样，随着需要显示的条目的增多和复杂度的增大，系统很快就会崩溃。可以想象，一个词汇列表可以远远超过 120 项，而 TextView

可以比我们在这里的显示的要复杂得多。

事实上，View 对象只有在屏幕上显示时才有必要存在，因此合理的做法是：只在需要显示的时候才创建视图对象。

RecyclerView 做的正是这一点。在本例中，它不是创建 120 个视图，而是创建 8 个，足以充满整个屏幕。当一个视图滚动出屏幕，RecyclerView 重用而不是销毁该视图。总之，无愧于它的名字：它一遍又一遍循环使用视图。

所以 RecyclerView 适用于那些有大量同类的 View 但是不能同时在屏幕中显示的情况，比如我们的词汇、联系人、用户、照片、图书、音乐文件列表等。想看到更多信息需要滚动视图，同时对离开屏幕的视图进行回收和重用。

图 6-4 解释了这个过程：左边是 NEC Vocab 应用的初始状态，当向上滚动视图的时候，一些子 View 被回收，如右边的框外部分中的两个不可见的 View。现在回收器可以把他们放入准备重用的 View 的列表以便在适当的时候重用。

对 View 的回收重用是非常有用。由于不需要每次都重新填充布局，这样可以节约 CPU 资源；同时由于不用保存大量的不可见的视图，也有效地节省了内存资源。

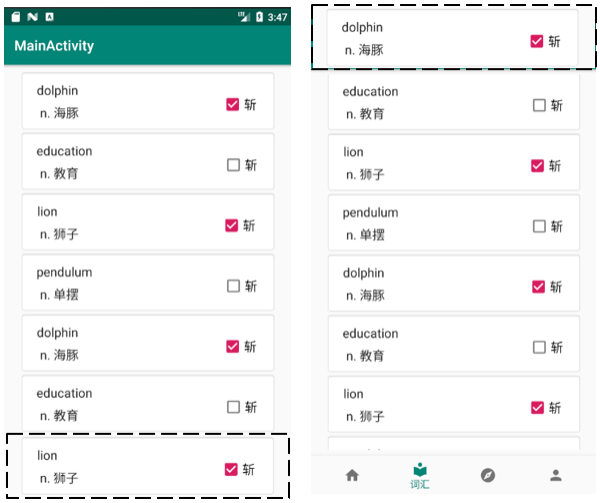

图 6-4 回收不可见的子 View

RecyclerView 本身不关心视图相关的问题

为了避免类似 ListView 的紧耦合问题，谷歌的改进就是 RecyclerView 本身不参与任何视图相关的问题。它不关心如何将子 View 放在合适的位置，也不关心如何分割这些子 View，更不关心每个子 View 各自的外观。进一步来说，**RecyclerView** 只负责回收和重用的工作，这就是它名字的由来。

所有关于布局、绘制和其他相关的问题，也就是跟数据展示相关的所有问题，都被委派给了一些"插件化"的类来处理。这使得 RecyclerView 的 API 变得非常灵活。需要一个新的布局么？接入另一个 LayoutManager 就可以了！想要不同的动画么？接入一个新的 ItemAnimator 就可以了，诸如此类。

6.4.2 RecyclerView 的基本用法

在使用 RecyclerView 时，必须指定一个适配器 Adapter 和一个布局管理器 LayoutManager。适配器继承 RecyclerView.Adapter 类。布局管理器用于确定 RecyclerView 中 Item 的展示方式以及决定何时复用已经不可见的 Item，避免重复创建以及执行高成本的 findViewById()方法。

RecyclerView 比 ListView 会多出许多操作，这也是 RecyclerView 灵活的地方，它将许多动能暴露出来，用户可以选择性的自定义属性以满足需求。

以下是 RecyclerView 中用于数据展示的一些重要的类，他们都是 RecyclerView 的内部类：

- Adapter：包装数据集合并且为每个条目创建视图；
- ViewHolder：保存用于显示每个数据条目的子 View；
- LayoutManager：将每个条目的视图放置于适当的位置；
- ItemDecoration：在每个条目的视图的周围或上面绘制一些装饰视图；
- ItemAnimator：在添加、移除或者重排序条目时添加动画效果。

我们将主要介绍 ViewHolder、Adapter 和 LayoutManager 三个类：

1. RecyclerView.ViewHolder

ViewHolder 的基本作用是缓存视图对象。

RecyclerView.ViewHolder 的子类可以通过 ViewHolder 的 public 的成员变量 itemView 来访问每个条目的根视图，所以 ViewHolder 子类中不需要再保存这个视图（见图 6-5）。

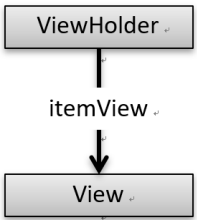

图 6-5 ViewHolder

我们以 NEC Vocab 应用为例，典型的 ViewHolder 子类如代码如下：

```
public class ViewHolder extends RecyclerView.ViewHolder {
    public final View mView;
    public final TextView mIdView;
    public final TextView mContentView;
    public WordEntity mItem;
    public ViewHolder(View itemView) {
        super(itemView);
        mView = itemView;
        mIdView = (TextView) itemView.findViewById(R.id.item_number);
        mContentView = (TextView) itemView.findViewById(R.id.content);
    }
    @Override
    public String toString() {
        return super.toString() + " '" + mContentView.getText() + "'";
    }
}
```

我们可以创建一个 ViewHolder 并访问我们自己创建的 mView 以及由超类 RecyclerView.ViewHolder 分配的 itemView 字段。该 itemView 字段就是 ViewHolder 存在的理由：它保持传递给 super(itemView)的整个视图的引用。

RecyclerView 不需要创建 View。它只创建 ViewHolder，由 ViewHolder 利用其 itemView 跟进 View，其过程如图 6-6 所示。

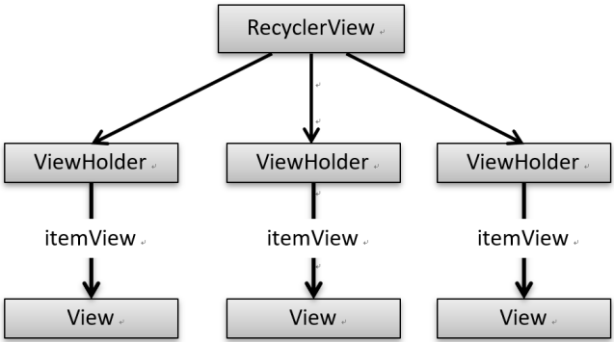

图 5-8 RecyclerView 及其 ViewHolder

2. RecyclerView.Adapter

事实上，RecyclerView 并不自己创建 ViewHolder，而是借助于 Adapter。Adapter 是一种控制器对象，从模型层获得数据，并提供给 RecyclerView 显示。Adapter 负责：

- 创建必要的 ViewHolder；
- 将 ViewHolder 与模型层的数据绑定。

Adapter 在 Android 中经常出现，比如 ListView、AutoCompleteTextView、Spinner 等，他们都继承自 AdapterView，但是 RecyclerView 并没有这样做。

对于 RecyclerView，谷歌使用新的 RecyclerView.Adapter 基类来取代旧的 Adapter 接口。

由于 RecyclerView.Adapter 是一个抽象类，要创建一个 Adapter，首先需要定义一个 RecyclerView.Adapter 的子类，该子类必须实现以下三个方法：

- public **VH** onCreateViewHolder(ViewGroup parent, int viewType)
- public void onBindViewHolder(VH holder, int position)
- public int getItemCount()

其中 **VH** 是泛型类，当继承 RecyclerView.Adapter 时需要使用具体的类来替换。

仍然以 NEC Vocab 应用为例，我们创建的 Adapter 子类将包含从 WordLib 获得的 WordEntity 列表。

当 RecyclerView 需要显示一个视图对象时，它将与其 Adapter 对话。图 5-9 显示了 RecyclerView 与其 Adapter 启动会话沟通的例子。

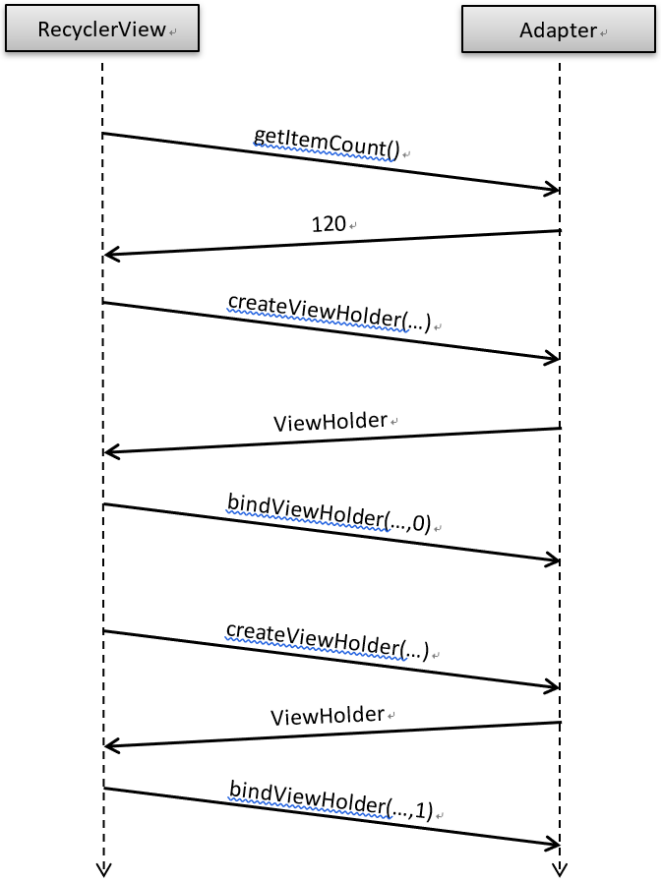

图 6-7 RecyclerView 与 Adapter 会话

首先，通过调用 adapter 的 getItemCount()方法，RecyclerView 询问列表中包含对象的数量。

接着，RecyclerView 调用 adapter 的 onCreateViewHolder(ViewGroup, int)方法创建 ViewHolder 和需要显示的视图对象 View。

最后，RecyclerView 调用 onBindViewHolder(ViewHolder, int)。RecyclerView 将传递一个 ViewHolder 和位置到该方法。Adapter 将为该位置查找模型数据并绑定到 ViewHolder 的视图 View。要进行绑定，adapter 将模型对象中的数据加载到视图中。

完成这些过程后，RecyclerView 将在屏幕上放置一个列表项。Adapter 的示例代码如下：

```
public class VocabularyItemRecyclerViewAdapter
        extends RecyclerView.Adapter<VocabularyItemRecyclerViewAdapter.ViewHolder> {
        private final List<WordEntity> mValues;
        private final OnListFragmentInteractionListener mListener;
        public VocabularyItemRecyclerViewAdapter(List<WordEntity> items
                , OnListFragmentInteractionListener listener) {
                mValues = items;
                mListener = listener;
        }
        @Override
        public ViewHolder onCreateViewHolder(ViewGroup parent, int viewType) {
                View view = LayoutInflater.from(parent.getContext())
                                .inflate(R.layout.fragment_item, parent, false);
                return new ViewHolder(view);
        }
        @Override
        public void onBindViewHolder(final ViewHolder holder, int position) {
                holder.mItem = mValues.get(position);
                holder.mIdView.setText(mValues.get(position).getEnglish());
                holder.mContentView.setText(mValues.get(position).getChinese());
                holder.mView.setOnClickListener(new View.OnClickListener() {
                        @Override
                        public void onClick(View v) {
                                if (null != mListener) {
                                        mListener.onListFragmentInteraction(holder.mItem);
                                }
                        }
                });
        }
        @Override
        public int getItemCount() {
                return mValues.size();
        }
}
```

3. RecyclerView.LayoutManager

LayoutManager 负责将所有的子 View 放置到适当位置。必须要给 RecyclerView 设置一个 LayoutManager，否则会抛出异常。

LayoutManager 的一个实现是：LinearLayoutManager，当要实现水平或垂直列表时可以使用该类。LinearLayoutManager 的用法非常简单：对它初始化就可以了，如果需要，就设置一个方向（水平/垂直）：

```
LinearLayoutManager layoutManager = new LinearLayoutManager(context);
```

```
layoutManager.setOrientation(LinearLayoutManager.VERTICAL);
```

当然，LinearLayoutManager 还提供了一些其他方法，我们对此不进行阐述，有兴趣的同学可查阅有关资料。

6.5 利用接口传递数据

我们用接口来传值的一个很典型的例子就是 RecyclerView 的点击事件。所以我们来分析一下点击事件的书写顺序：

1. 在 fragment 中定义一个内部回调接口

在接口里写一个没有方法体的方法，我们希望传递点击事件的位置所指定的那个词条，类似下面代码段。

```
public interface OnListFragmentInteractionListener {
    void onListFragmentInteraction(WordEntity item);
}
```

2. 让包含该 fragment 的 activity 实现该回调接口

这样 fragment 即可调用该回调方法将数据传给 activity。

比如我们在 MainActivity 的 VocabularyItemFragment 向 MainActivity 传值，即所选的那个词条。我们希望在 MainActivity 中启动该词条的详解，即 VocabularyFragment。这个时候，我们需要到 Activity 中进行操作。因此，我们要在 MainActivity 实现 VocabularyItemFragment 定义的接口方法 onListFragmentInteraction(WordEntity item)。

即首先在类定义中 implements 接口 OnListFragmentInteractionListener：

```
public class MainActivity extends AppCompatActivity
        implements VocabularyItemFragment.OnListFragmentInteractionListener{
```

接着实现该接口的方法，即启动 VocabularyFragment：

```
@Override
public void onListFragmentInteraction(WordEntity item) {
    getSupportFragmentManager()
            .beginTransaction()
            .replace(R.id.fragment_container,VocabularyFragment.newInstance(item))
            .addToBackStack(null)
            .commit();
}
```

3. 将 activity 实现的接口传给 fragment 呢

activity 实现了接口怎么传给 fragment 呢？当 fragment 添加到 activity 中时，会调用 fragment 的方法 onAttach()，该方法检查 activity 是否实现了 OnListFragmentInteractionListener 接口，检查方法就是对传入的 activity 的实例进行类型转换，然后赋值给我们在 fragment 中定义的接口。

```
@Override
public void onAttach(Context context) {
    super.onAttach(context);
    if (context instanceof OnListFragmentInteractionListener) {
        mListener = (OnListFragmentInteractionListener) context;
    } else {
        throw new RuntimeException(context.toString()
                + " must implement OnListFragmentInteractionListener");
    }
}
```

注意 onAttach 方法中，在赋值之前要做一个判断，检查 Activity（这里的 context 就是 activity，只有依附在该 activity 上的 fragment 可以向 activity 传值）中有没有实现了这个接口，用到了 instanceof。如果没有实现接口，我们就抛出异常。

instanceof 是 Java 的一个二元操作符（运算符），和==，>，<是同一类东西。由于它是由字母组成的，所以也是 Java 的保留关键字。它的作用是判断其左边对象是否为其右边类的实例，返回 boolean 类型的数据。

```
boolean result = object instanceof class
```

如果 object 是 class 的一个实例，则 instanceof 运算符返回 true。如果 object 不是指定类的一个实例，或者 object 是 null，则返回 false。

4. 用 onDetach 释放 activity 对象

当一个 fragment 从 activity 中剥离的时候，就会调用 onDetach 方法，这个时候要把传递进来的 activity 对象释放掉，不然会影响 activity 的销毁，产生不必要的错误。

```
@Override
public void onDetach() {
    super.onDetach();
    mListener = null;
}
```

5. 触发接口

建立完接口我们需要找到想在什么界面或什么情况下触发这个接口。比如 RecyclerView 中我们希望在它的行布局被点击时候触发，那么我们就需要找到它的行布局在哪里。通常

RecyclerView 的适配器在 onBindViewHolder 中操作行布局的视图，我们就在这里触发接口。

既然知道在哪个界面有着接口的触发条件，首先我们需要在这个界面创建一个接口对象：

```
private final OnListFragmentInteractionListener mListener;
```

在我们有了一个不为空的接口对象后，我们需要在行布局被点击的事件中（系统的）来触发我们自己写的， 如下，在 onBindViewHolder 中

```
@Override
public void onBindViewHolder(final ViewHolder holder, int position) {
        holder.mItem = mValues.get(position);
        holder.mIdView.setText(mValues.get(position).getChinese());
        holder.mContentView.setText(mValues.get(position).getEnglish());
        holder.mView.setOnClickListener(new View.OnClickListener() {
                @Override
                public void onClick(View v) {
                        if (null != mListener) {
                                mListener.onListFragmentInteraction(holder.mItem);
                        }
                }
        });
}
```

6.6 创建词汇表

有了上面的知识，接下来我们就可以创建词汇表了。右键单击 com.studio.aime.necvocab 包，选择 New → Fragment → Fragment （List）。在弹出的对话框中将 Fragment class Name 命名类名为 VocabularyItemFragment，其余保留默认，单击 OK。

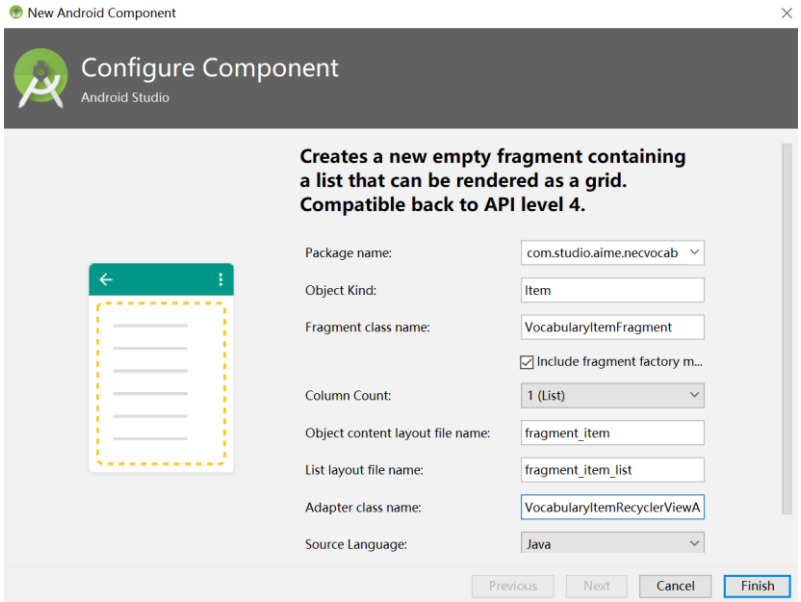

图 6-8 利用 Fragment（List）向导创建 VocabularyItemFragment

现在，我们看看 Android Studio 的 Fragment（List）向导做了什么。

向导首先在在 build.gradle 文件中引入了 recyclerview 类的依赖项。

```
implementation 'com.android.support:recyclerview-v7:28.0.0'
```

其次，正如所料，向导创建了一个名为 VocabularyItemFragment.Java 文件，它管理者一个名为 fragment_item_list.xml 的布局文件。该布局文件使用了 RecyclerView 作为根组件，用于显示列表。

同时向导还创建了一个名为 VocabularyItemRecyclerViewAdapter 的 Java 文件管理适配器。

此外，向导还创建了一个名为 fragment_item.xml 的列表项布局文件，用于显示每条列表项的信息。

当然，向导也创建了一个 dummy 目录，其中包含模型文件 DummyContent。但我们将使用 WordEntity 和 WordLib 代替模型文件。

6.6.1 设计定制列表项布局

我们先来完成列表项布局的设计。

我们需要对 fragment_item.xml 文件进行一定的修改。向导创建的列表项文件的默认根组件是 LinearLayout。为了便于使用图形布工具的拖拽方法设计列表项布局，利用与 5.3.4 节完全相同的方法将 LinearLayout 转换为 ConstraintLayout。如图 6-9 所示，为你的 constraintLayout 选择一个适合的背景，再添加一个 CheckBox 组件。这里 CheckBox 只是显示用户是否掌握该词条，我们不希望用户在"词汇表"中勾选是否掌握该词条，因此，去掉 CheckBox 的 clickable 属性的勾选。设置好各组件的 ID、大小、名称以及其他必要的参数。

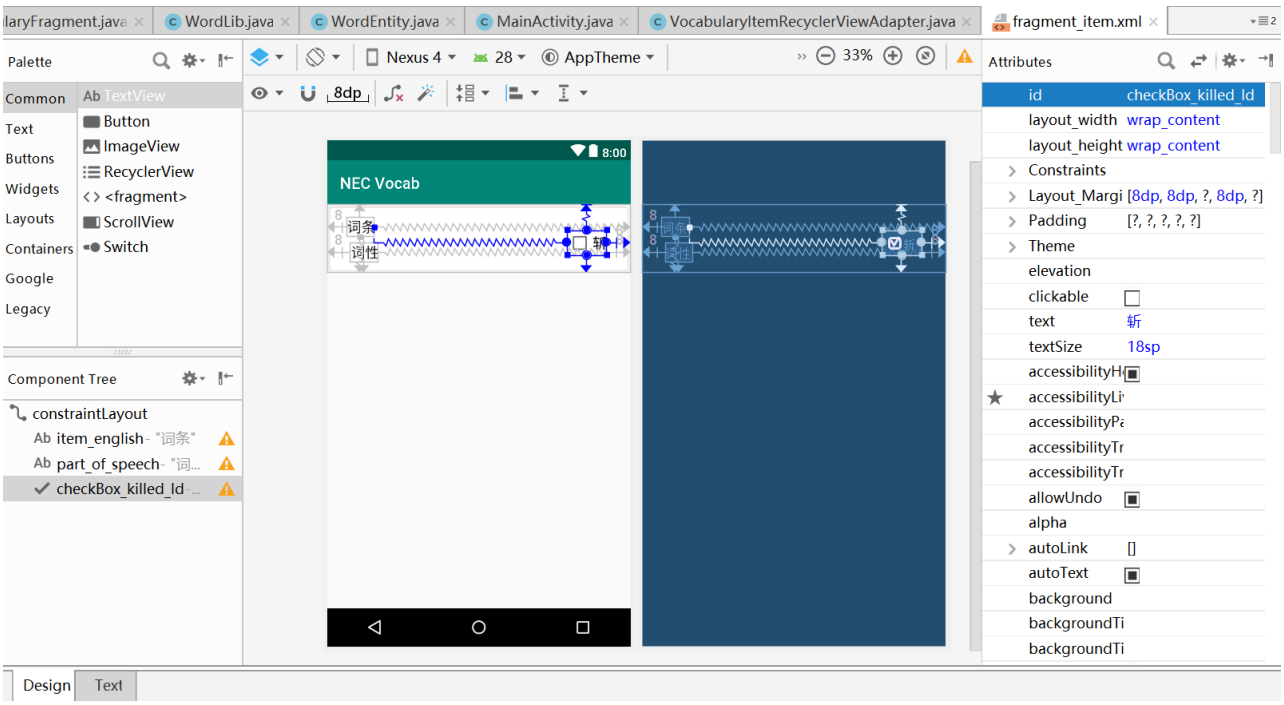

图 6-9 设置定制列表项布局

6.6.2 实现适配器

修改向导创建的 VocabularyItemRecyclerViewAdapter 文件，用 WordEntity 和 WordLib 代替了向导创建的模型，如代码清单 6-3 中粗体部分所示。

代码清单 **6-3** 修改向导创建的适配器文件（**VocabularyItemRecyclerViewAdapter.java**）

```
public class VocabularyItemRecyclerViewAdapter
        extends RecyclerView.Adapter<VocabularyItemRecyclerViewAdapter.ViewHolder> {
    private final List<WordEntity> mValues;
    private final OnListFragmentInteractionListener mListener;
    public VocabularyItemRecyclerViewAdapter(List<WordEntity> items, OnListFragmentInteractionListener
listener) {
        mValues = items;
        mListener = listener;
    }
    @Override
    public ViewHolder onCreateViewHolder(ViewGroup parent, int viewType) {
        View view = LayoutInflater.from(parent.getContext())
                .inflate(R.layout.fragment_item, parent, false);
        return new ViewHolder(view);
    }
    @Override
    public void onBindViewHolder(final ViewHolder holder, int position) {
        holder.mItem = mValues.get(position);
        holder.mIdView.setText(mValues.get(position).getEnglish());
        holder.mContentView.setText(mValues.get(position).getChinese());
        holder.mKilled.setChecked(mValues.get(position).isKilled());
```

```
            holder.mView.setOnClickListener(new View.OnClickListener() {
                @Override
                public void onClick(View v) {
                    if (null != mListener) {
                        mListener.onListFragmentInteraction(holder.mItem);
                    }
                }
            });
        }
        @Override
        public int getItemCount() {
            return mValues.size();
        }
        public class ViewHolder extends RecyclerView.ViewHolder {
            public final View mView;
            public final TextView mIdView;
            public final TextView mContentView;
            public final CheckBox mKilled;
            public WordEntity mItem;
            public ViewHolder(View view) {
                super(view);
                mView = view;
                mIdView = (TextView) view.findViewById(R.id.item_number);
                mContentView = (TextView) view.findViewById(R.id.content);
                mKilled = (CheckBox)view.findViewById(R.id.checkBox_killed_Id);
            }
            @Override
            public String toString() {
                return super.toString() + " '" + mContentView.getText() + "'";
            }
        }
    }
```

6.6.3 使用 RecyclerView

现在打开向导创建的 VocabularyItemFragment 文件。该文件的主要功能就是设置布局管理器，设置适配器。我们几乎不需要修改（仅修改了最后一行，将 DummyItem 替换为 WordEntity），便可直接使用。

代码清单 6-4 使用 RecyclerView（VocabularyItemFragment.java）

```
public class VocabularyItemFragment extends Fragment {
    private static final String ARG_COLUMN_COUNT = "column-count";
    private int mColumnCount = 1;
    private OnListFragmentInteractionListener mListener;
    public VocabularyItemFragment() {
    }
    @SuppressWarnings("unused")
    public static VocabularyItemFragment newInstance(int columnCount) {
        VocabularyItemFragment fragment = new VocabularyItemFragment();
        Bundle args = new Bundle();
        args.putInt(ARG_COLUMN_COUNT, columnCount);
```

```
        fragment.setArguments(args);
        return fragment;
    }
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        if (getArguments() != null) {
            mColumnCount = getArguments().getInt(ARG_COLUMN_COUNT);
        }
    }
    @Override
    public View onCreateView(LayoutInflater inflater, ViewGroup container,
                             Bundle savedInstanceState) {
        View view = inflater.inflate(R.layout.fragment_item_list, container, false);
        if (view instanceof RecyclerView) {
            Context context = view.getContext();
            RecyclerView recyclerView = (RecyclerView) view;
            if (mColumnCount <= 1) {
                recyclerView.setLayoutManager(new LinearLayoutManager(context));
            } else {
                recyclerView.setLayoutManager(new GridLayoutManager(context, mColumnCount));
            }
            recyclerView.setAdapter(new VocabularyItemRecyclerViewAdapter(
                WordLib.get(getActivity()).getWordList(), mListener));
        }
        return view;
    }
    @Override
    public void onAttach(Context context) {
        super.onAttach(context);
        if (context instanceof OnListFragmentInteractionListener) {
            mListener = (OnListFragmentInteractionListener) context;
        } else {
            throw new RuntimeException(context.toString()
                    + " must implement OnListFragmentInteractionListener");
        }
    }
    @Override
    public void onDetach() {
        super.onDetach();
        mListener = null;
    }
    public interface OnListFragmentInteractionListener {
        void onListFragmentInteraction(WordEntity item);
    }
}
```

这里需要做几点说明。

OnListFragmentInteractionListener

VocabularyItemFragment 代码的最后声明了一个监听器接口，这里用于和第五章中的词汇明细的 VocabularyFragment 通信。

在一个 Activity 中往往有多个 Fragment，他们之间是无法直接通信的，所以 Fragment 需要

通过它们所绑定的 Activity 作为中介来进行通信。而 OnListFragmentInteractionListener 就是用来实现不同 Fragment 之间通信的接口。该接口必须由包含该片段的活动实现，以允许该片段中的交互与该活动以及该活动中可能包含的其他片段通信。

其中：

```
@Override
    public void onAttach(Context context) {
        super.onAttach(context);
        if (context instanceof OnListFragmentInteractionListener) {
            mListener = (OnListFragmentInteractionListener) context;
        } else {
            throw new RuntimeException(context.toString()
                    + " must implement OnListFragmentInteractionListener");
        }
    }
```

这里的 context 也就是实现 OnListFragmentInteractionListener 的 Activity。在我们的项目中就是 MainActivity。为此，在 onAttach 方法首先利用 instanceof 判断该 Activity 的实例，也就是这里的 context 是否是实现了 OnListFragmentInteractionListener 的 Activity 的实例。如果是，则 OnListFragmentInteractionListener 在 OnAttach 阶段注册并复制给 mListener。否则抛出异常，并指出 Activity 必须实现 OnListFragmentInteractionListener。

6.6.4 托管 Activity

正如前面所指出的，包托管 VocabularyItemFragment 的活动必须实现该 Fragment 所定义的接口 OnListFragmentInteractionListener。我们当然希望 MainActivity 托管该 Fragment，即当用户点击“词汇”标签页时，能出现 VocabularyItemFragment 以显示词汇列表。

在 MainActivity 中实现接口 OnListFragmentInteractionListener，如代码清单 6-5 所示。

代码清单 6-5 在 MainActivity 中实现 OnListFragmentInteractionListener（MainActivity.java）

```
public class MainActivity extends AppCompatActivity
        implements VocabularyItemFragment.OnListFragmentInteractionListener{
    private static final String TAG = "MainActivity";
    private static final String KEY_INDEX = "index";
    private int mCurrentIndex = 0;
    private BottomNavigationView.OnNavigationItemSelectedListener mOnNavigationItemSelectedListener
            = new BottomNavigationView.OnNavigationItemSelectedListener() {
        @Override
        public boolean onNavigationItemSelected(@NonNull MenuItem item) {
            mCurrentIndex = item.getItemId();
            switch (item.getItemId()) {
                case R.id.navigation_home:
                    getSupportFragmentManager()
```

```
                        .beginTransaction()
                        .replace(R.id.fragment_container,new HomeFragment())
                        .commit();
                return true;
            case R.id.navigation_voclib:
                getSupportFragmentManager()
                        .beginTransaction()
                        .replace(R.id.fragment_container,VocabularyItemFragment.newInstance(1))
                        .commit();
                return true;
            case R.id.navigation_discover:
                getSupportFragmentManager()
                        .beginTransaction()
                        .replace(R.id.fragment_container,new DiscoverFragment())
                        .commit();
                return true;
            case R.id.navigation_mine:
                getSupportFragmentManager()
                        .beginTransaction()
                        .replace(R.id.fragment_container,new MyFragment())
                        .commit();
                return true;
        }
        return false;
    }
};
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);
    if (savedInstanceState == null) {
        getSupportFragmentManager()
                .beginTransaction()
                .replace(R.id.fragment_container,new HomeFragment())
                .commit();
    }
    BottomNavigationView navigation = (BottomNavigationView) findViewById(R.id.navigation);
    navigation.setOnNavigationItemSelectedListener(mOnNavigationItemSelectedListener);
}
@Override
public void onListFragmentInteraction(WordEntity item) {
    getSupportFragmentManager()
            .beginTransaction()
            .replace(R.id.fragment_container,VocabularyFragment.newInstance(item))
            .addToBackStack(null)
            .commit();
}
@Override
public void onSaveInstanceState(Bundle savedInstanceState) {
    super.onSaveInstanceState(savedInstanceState);
    savedInstanceState.putInt(KEY_INDEX, mCurrentIndex);
}
}
```

VocabularyItemFragment 将用户点击某个词条的触发事件传递 MainActivity 后，希望

MainActivity 能够启动该词条的明细。因此，MainActivity 实现 OnListFragmentInteractionListener 接口的 onListFragmentInteraction 方法就是启动 VocabularyFragment：

```
@Override
public void onListFragmentInteraction(WordEntity item) {
    getSupportFragmentManager()
            .beginTransaction()
            .replace(R.id.fragment_container,VocabularyFragment.newInstance(item))
            .addToBackStack(null)
            .commit();
}
```

现在运行 NEC Vocab 应用，进入"词汇"标签页，点击某个词条，查看运行结果。

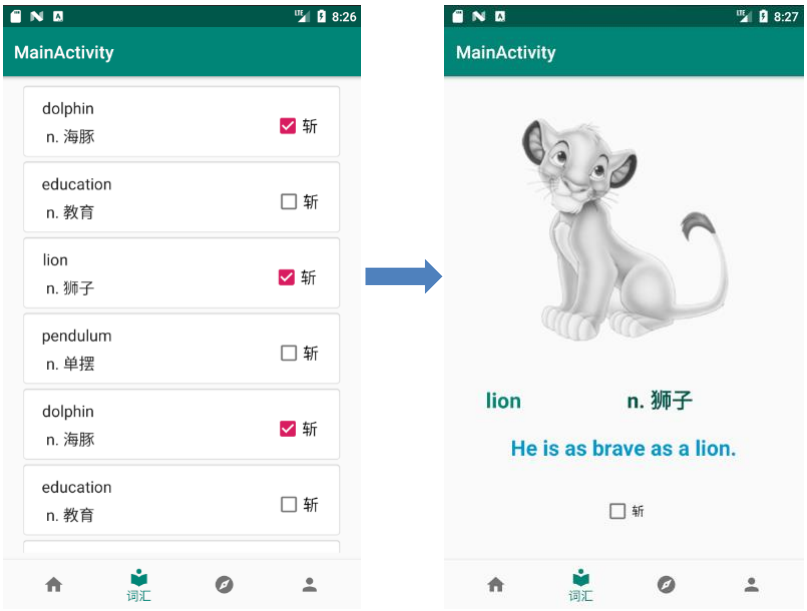

图 6-10 由词汇表查看词汇详解