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

5 实现词汇详解

5.1 本章目标

本章和接下来的两讲，我们将开发 NEC Vocab 的词汇模块。如图 5-1 所示，进入词汇模块后，主屏幕将显示需要学习的“词汇表”，用户可选中列表中的某一词条查看“词汇详解”。

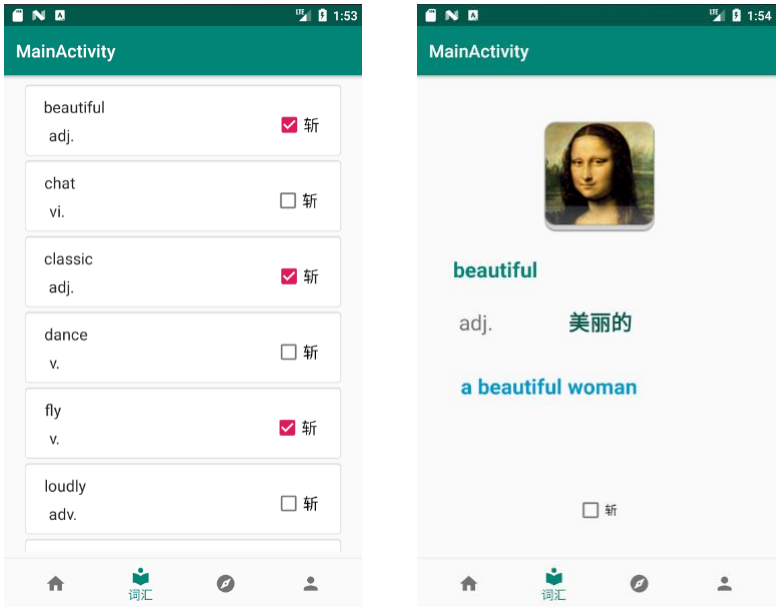

图 5-1 词汇模块是一种列表-明细应用

从图 5-1 可以看到，NEC Vocab 的词汇模块是一种列表-明细应用。其用户界面主要由“词汇表”（列表）和“词汇详解”（明细）组成。

事实上，很多种类的应用，如备忘录、图书、商品查询等，都可以归结为这种列表-明细应用。以 NEC Vocab 的词汇模块作为基本框架，可以开发出多种具有列表-明细结构的应用。

当然，NEC Vocab 的词汇模块比较复杂，我们需要 4 章的篇幅来完成它。

本章将首先开发 NEC Vocab 的词汇模块的明细部分——“词汇详解”界面。当然，“词汇详解”界面并不简单，它应当包括诸如文字、语音、图像、动画甚至视频功能。但本章我们将只开发一些最基本的功能：显示英文单词、图片、词性、中文释义和例句。

5.2 使用 UI fragment

在第四章中，我们引入了带有底部导航栏的主界面。然而，在那里我们仅仅用同一个 activity 界面，当点击不同 Tab（标签页）时，仅仅展现了不同的文字。那么，当用户在不同 Tab 选项间做切换时，如何能够展现不同的界面呢？

设想每个底部的 Tab 都通过 startActivity(Intent)启动一个 activity 实例。由于 activity 是与屏幕绑定在一起的，即 activity 将占据整个屏幕，因此，从某个底部 Tab 启动一个 activity 后，将看不到主界面的底部导航栏。如果想进入另一个 Tab 所展示的界面，需要单击后退键销毁 activity 并返回主界面，接下来再选择另一个 Tab。

理论上这是可行的，但如果需要呈现更复杂的用户界面和更多的跳转又会如何呢？如果再加上如下应用需求，我们又如何设计？

想象我们开发的 NEC Vocab 应用，其中一个界面展示了一组词汇，当点击了其中一个词条，就打开另一个界面显示该词条的详细内容。如果是在手机中设计，我们可以将词汇表放在一个活动中，将词条的详细内容放在另一个活动中，如图 5-2 左边所示。

可是如果在平板上也这么设计，那么词汇列表将会被拉长至填充满整个平板的屏幕，而英文单词一般都不会太长，这样将会导致界面上有大量的空白区域。因此，更好的设计方案是将词汇列表界面和词条详细内容界面分别放在两个碎片（fragment）中，然后在同一个活动里引入这两个碎片，这样就可以将屏幕空间充分地利用起来了，如图 5-2 右边所示。

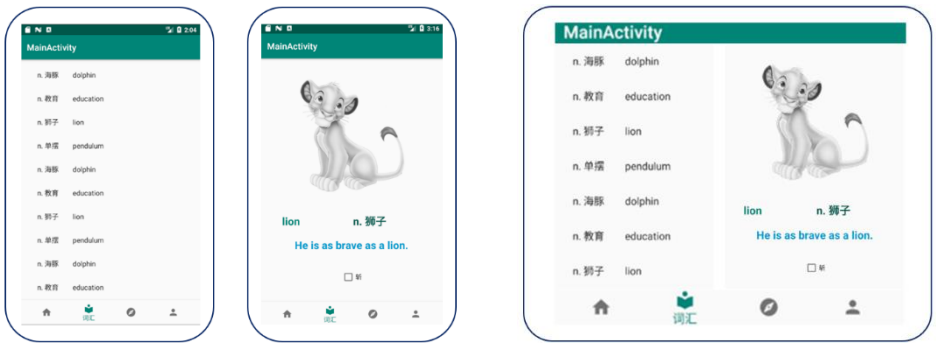

图 5-2 手机和平板设备上理想的列表和明细界面

5.2.1 fragment 的引入

activity 就像照相馆的一个墙面，当然可以装修照相馆的墙面，并以此墙面作为背景拍照。

但顾客如果想换一个背景怎么办？再装修一个墙面？理论上可以，但这绝不是一个好办法。去过照相馆的人都知道，照相馆通常会在一面墙上安装一个卷轴，卷轴上有若干背景画布。顾客需要哪种背景，就利用卷轴换上哪个背景画布。这个画布就类似我们即将引入的 fragment。

采用 fragment 而不是 activity 进行应用的 UI 管理，可以绕开 Android 系统 activity 规则的限制。受 fragment 管理的用户界面可以是一整屏或是屏幕的一部分。

管理用户界面的 fragment 又称 UI fragment。它也有自己产生布局文件的视图。fragment 视图包含了用户可以交互的可视化 UI 元素。但是，fragment 本身不具有在屏幕上显示视图的能力，只有被放置在 activity 的视图结构中，fragment 视图才能显示在屏幕上。这样，我们可联合使用 fragment 和 activity 来组装或重新组装用户界面。利用 activity 视图提供插入 fragment 视图的位置。如果有多个 fragment 要插入，activity 视图也可以提供多个位置。

从技术上来说，在整个生命周期过程中，activity 的视图可以保持不变。因此不用担心会违反 Android 系统 activity 规则。

我们可以暂时把 activity 委托 fragment 它完成一些任务理解为托管，即 **activity** 在其视图层里提供一处位置用来放置 **fragment** 的视图。

对于 NEC Vocab 应用而言，这种托管关系如图 5-3 所示。

由图 5-3 可以看出，应用的主界面将由组名为 HomeFragment、VocabularyFragment 等 UI fragment 进行管理。这些 fragment 的实例将通过一个名为 MainActivity 的 activity 来托管。

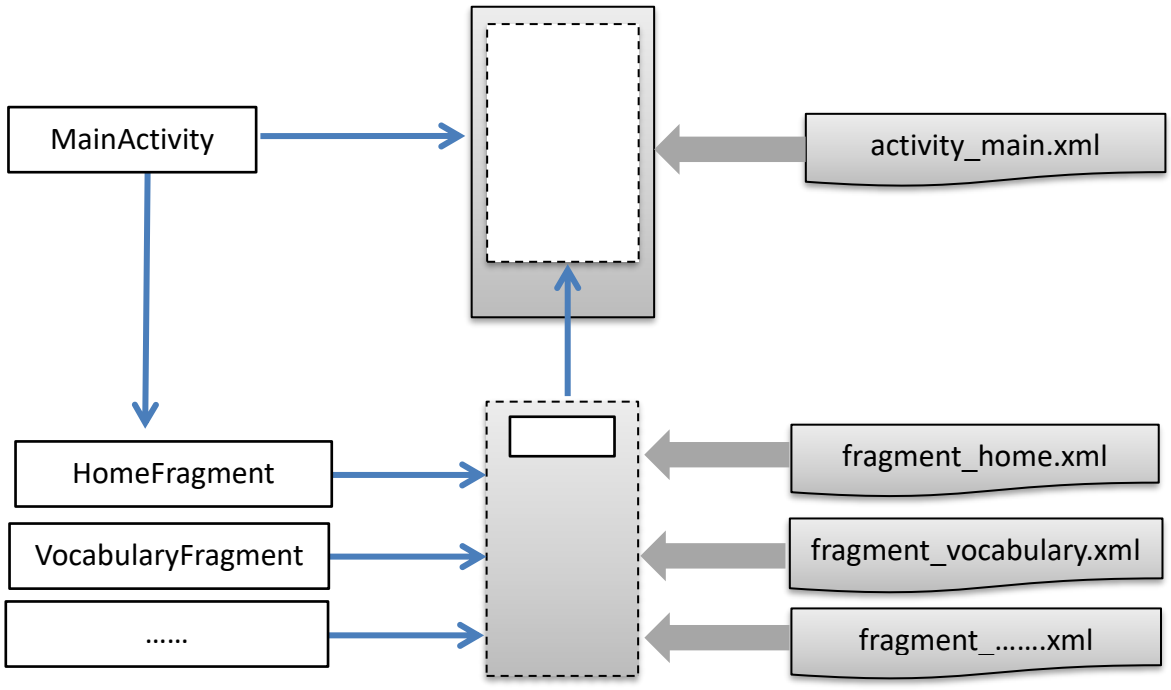

图 5-3 MainActivity 托管着 HomeFragment、VocabularyFragment 等 fragment

5.2.2 fragment 的生命周期

要托管 UI fragment，activity 必须做到：

- 在布局中为 fragment 的视图安排位置；
- 管理 fragment 实例的生命周期。

类似于 activity 的生命周期，fragment 具有停止、暂停和运行状态，也拥有可以覆盖的方法，用来在关键节点完成一些任务。可以看到，fragment 的许多方法对应着 activity 生命周期方法。

生命周期方法的对应非常重要。因为 fragment 代表 activity 在工作，它的状态应该也反映 activity 的状态。因而 fragment 需要对应的生命周期方法来处理 activity 的工作。

图 5-4 展示了 fragment 的生命周期。

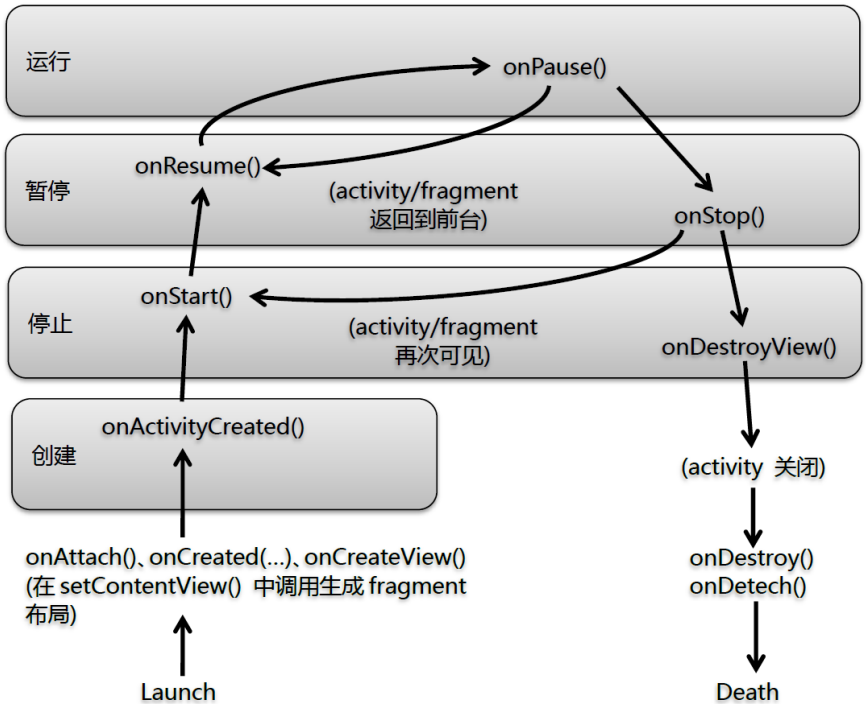

图 5-4 fragment 生命周期

fragment 生命周期与 activity 生命周期的一个关键区别在于：fragment 的生命周期方法是由托管 activity 而不是操作系统调用的。操作系统无法知晓 activity 用来管理视图的 fragment，fragment 的使用是 activity 自己内部的事情。

随着 NEC Vocab 应用开发的深入，我们会看到更多的 fragment 生命周期方法。

5.2.3 定义容器视图

虽然我们要在托管 activity 代码中添加 UI fragment，但还是需要在 activity 视图结构中为 fragment 视图安排位置。在 activity_main 布局中删除 TextView，并从组件面板的 Layout 下将 FrameLayout 拖拽到预览区域，如图 5-5 所示。将该 FrameLayout 的 ID 设为 fragment_container，同时为其添加约束，使其下方约束固定于 navigation 组件的上方。

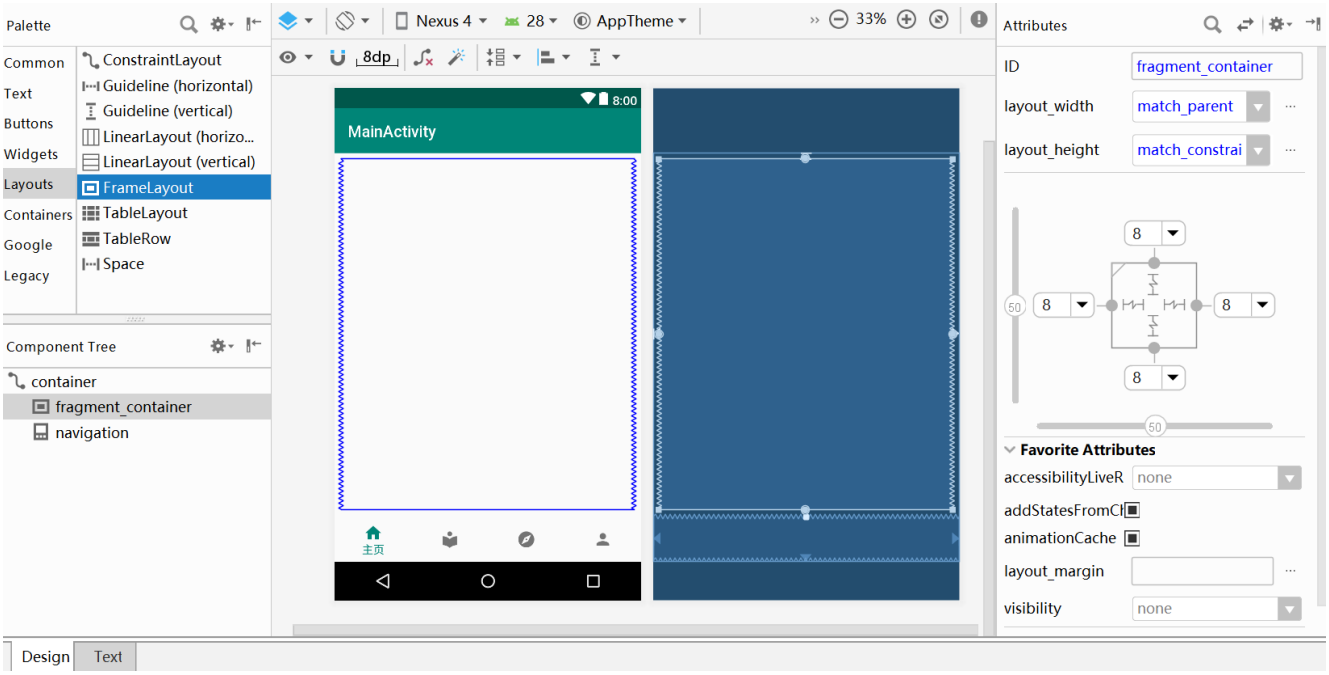

图 5-5 MainActivity 类的 fragment 托管布局

现在打开 MainActivity 文件，由于我们删除 TextView 组件，因此，MainActivity 中所有与其相关的代码都必须删除，参见代码清单 5-1。

代码清单 **5-1** 删除 **TextView** 组件相关代码（**MainActivity.java**）

```
public class MainActivity extends AppCompatActivity {
    private static final String TAG = "MainActivity";
    private static final String KEY_INDEX = "index";
    private int mCurrentIndex=0;
    private TextView mTextMessage;

    private BottomNavigationView.OnNavigationItemSelectedListener mOnNavigationItemSelectedListener
            = new BottomNavigationView.OnNavigationItemSelectedListener() {
        @Override
        public boolean onNavigationItemSelected(@NonNull MenuItem item) {
            mCurrentIndex = item.getItemId();
            switch (mCurrentIndex) {
                case R.id.navigation_home:
                    mTextMessage.setText(R.string.title_home);
```

```
                return true;
            case R.id.navigation_voclib:
                mTextMessage.setText(R.string.title_voclib);
                return true;
            case R.id.navigation_discover:
                mTextMessage.setText(R.string.title_discover);
                return true;
            case R.id.navigation_mine:
                mTextMessage.setText(R.string.title_mine);
                return true;
        }
        return false;
    }
};

@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);
    mTextMessage = (TextView) findViewById(R.id.message);
    mCurrentIndex = R.id.navigation_home;
    if (savedInstanceState != null){
        mCurrentIndex = savedInstanceState.getInt(KEY_INDEX, 0);
    }

    switch (mCurrentIndex) {
        case R.id.navigation_home:
            mTextMessage.setText(R.string.title_home);
            break;
        case R.id.navigation_voclib:
            mTextMessage.setText(R.string.title_voclib);
            break;
        case R.id.navigation_discover:
            mTextMessage.setText(R.string.title_discover);
            break;
        case R.id.navigation_mine:
            mTextMessage.setText(R.string.title_mine);
            break;
    }
    BottomNavigationView navigation = (BottomNavigationView) findViewById(R.id.navigation);
    navigation.setOnNavigationItemSelectedListener(mOnNavigationItemSelectedListener);
}

@Override
public void onSaveInstanceState(Bundle savedInstanceState) {
    super.onSaveInstanceState(savedInstanceState);
    savedInstanceState.putInt(KEY_INDEX, mCurrentIndex);
}
}
```

运行 NEC Vocab 应用检查一下实现代码。不过，由于 MainActivity 还没有托管任何 fragment，因此我们只能看到一个空的主界面。

稍后，我们会编写代码，将 fragment 的视图放置到 FrameLayout 中。现在，我们首先需要

来创建词汇 fragment 和其他三个 fragment。

5.3 利用 Fragment 创建词汇详解界面

接下来我们将首先开发 NEC Vocab 的词汇模块的明细部分——"词汇详解"界面。

5.3.1 MVC 设计模式

我们可以利用对象图来更好地理解 NEC Vocab 应用"词汇详解"。

事实上，Android 应用就是基于模型—控制器—视图（Model-View-Controller，简称 MVC）的架构模式进行设计的。采用 MVC 设计模式，应用的任何对象都可归结为模型对象、视图对象和控制对象中的一种。

模型对象存储着应用的数据和业务逻辑。模型类通常用来映射与应用相关的一些事物，如用户、商店里的商品、服务器上的图片或者一段视频，又或是 NEC Vocab 应用里的词汇。模型对象不关心用户界面，它存在的唯一目的就是存储和管理应用数据。Android 应用里模型类通常是我们创建的定制类。应用的全部模型对象组成了模型层。

视图对象知道如何在屏幕上绘制自己以及如何响应用户的输入，如用户的触摸等。一个简单的经验法则是：凡是能够在屏幕上看见的对象就是视图对象。Android 应用自带了很多可配置的视图类，如 TextView、Button 等。当然，也可以定制开发自己的视图类。应用的全部视图对象组成了视图层。NEC Vocab 应用的视图层是由各个 xml 文件中定义的各类组件构成的。

控制对象包含了应用的逻辑单元，是视图与模型对象联系的纽带。控制对象被设计用来响应由视图对象触发的各类事件，此外还用来管理模型对象与视图层间的数据流动。在 Android 的世界里，控制器通常是 Activity、Fragment 或 Service 的一个子类（后面将会介绍 Service 的概念）。NEC Vocab 词汇详解的控制层将由 MainActivity 和 VocabularyFragment 组成。

图 5-6 展示了在响应用户单击按钮等事件时，对象间的交互控制数据流。注意，模型对象与视图对象不直接交互。控制器作为它们之间的纽带，接收来自对象的消息，然后向其他对象发送操作指令。

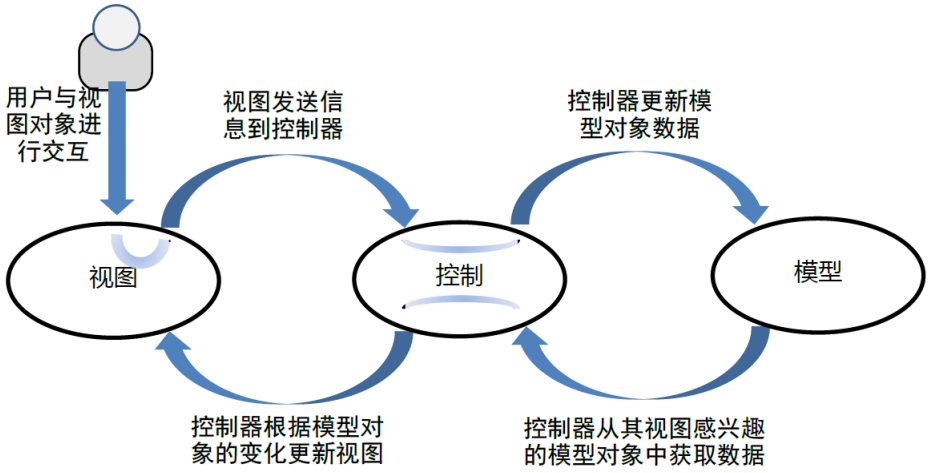

图 5-6 MVC 数据控制流与用户交互

使用 MVC 设计模式的优点

随着应用功能的持续扩展，应用往往会变得过于复杂而让人难以理解。

在面向对象编程中，我们以类的方式组织代码，这有助于我们从整体视角设计和理解应用。这样，我们就可以按类，而不是一个个变量和方法去思考设计开发问题。

同样，把类按照模型、视图和控制层进行组织，有助于我们设计和理解应用。这样，我们就可以按层，而不是一个个类来考虑设计开发了。

使用 MVC 模式还可以让类的复用更加容易。相比功能多而全的类，有特别功能限定的类更加有利于代码的复用。举例来说，模型类 WordEntity（单词）与用作显示问题的组件没有逻辑关联。这样，就很容易在应用里按需要自由使用 WordEntity 类。假设现在想显示所有词汇表，很简单，直接复用 WordEntity 对象逐条显示就可以了。

NEC Vocab 应用的词汇明细对象关系

图 5-7 展示了 NEC Vocab 应用的词汇模块对象的整体关系。

可以看到 VocabularyFragment 负责创建并管理用户界面，与模型对象进行交互。其中 WordEntity 和 VocabularyFragment 是我们要开发的类。

- WordEntity 实例代表了某个单词。在本章中，一个 WordEntity 有一张图片、一条词一个词汇释义、几个例句、一个学习情况和一个标识 ID。学习情况用布尔类型的 mKilled 表示。标识 **mId** 是识别 WordEntity 实例的唯一元素。为简单起见，本章只使用一个 WordEntity 实例，并将其存放在 VocabularyFragment 类的成员变量（mWord）中。
- MainActivity 的 FrameLayout 组件为 VocabularyFragment 要显示的视图安排了存放位

置。

- VocabularyFragment 的视图由一个 ConstraintLayout 组件及一个 ImageView 、三个 TextView 和一个 CheckBox 组件组成。VocabularyFragment 类中有一个存储 CheckBox 的成员变量（mKilledCheckBox）。mKilledCheckBox 上设有监听器，当 mKilledCheckBox 发生改变时，用来更新模型层的数据。

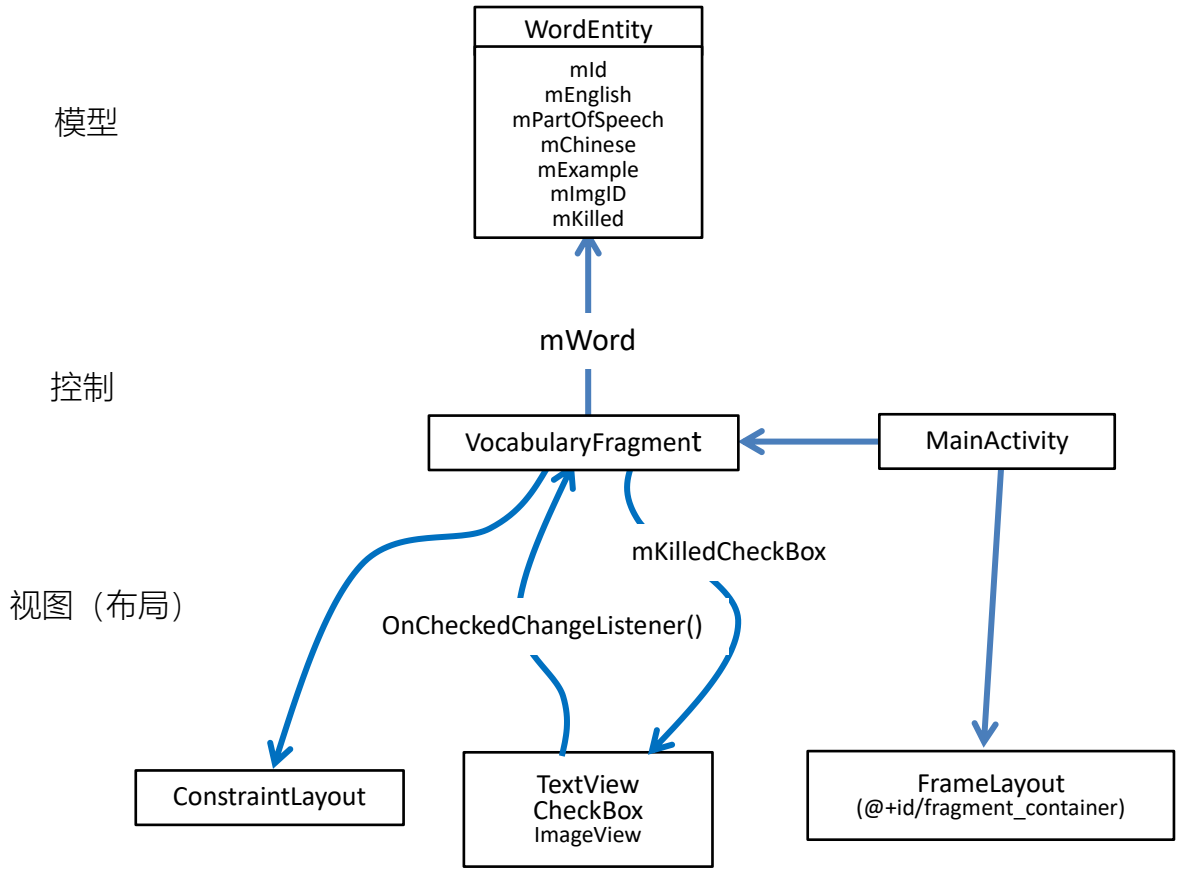

图 5-7 NEC Vocab 应用的词汇模块 MVC 关系图（本章应完成部分）

5.3.2 创建模型层

实现 NEC Vocab 应用的模型层，需要新增一个 WordEntity 类。WordEntity 的一个对象用来封装一个词汇。

创建 WordEntity 类

在 Peoject 视图选中包 com.studio.aime.necvocab，点击右键，在弹出窗口选择 New，再在随后的弹出窗口点击 Java Class，在接下来的对话框中将该类命名为 WordEntity，并点击 **OK**。

在 WordEntity.java 文件中添加 6 个成员变量和一个构造函数，如代码清单 5-2 所示。

代码清单 **5-2 WordEntity** 类（**WordEntity.java**）

```
public class WordEntity implements Serializable {
    private UUID mId;
    boolean mKilled;
    private String mEnglish;
    private String mPartOfSpeech;
    private String mChinese;
    private String mExample;
    private int mImgID;
    public WordEntity(){
        //创建唯一标识符
        mId = UUID.randomUUID();
    }
    public WordEntity(String ve,String vp, String vc, String eg, int imgId){
        //创建唯一标识符
        mId = UUID.randomUUID();
        mEnglish = ve;
        mPartOfSpeech = vp;
        mChinese = vc;
        mExample = eg;
        mImgID = imgId;
    }
```

WordEntity 类保存六部分数据：英语词条、词性、汉语释义、英语例句、释义图片和掌握情况。我们为 WordEntity 类构建了两个构造函数。

为什么 mImgID 是 int 类型的，而不是 String 类型？本讲中我们暂时将图片保存在 res/drawable 中。资源 ID 总是 int 类型，所以这里将 mImgID 设置为 int 类型。

Serializable 序列化的接口

需要指出的是，为了便于传递对象，这里实现了一个 Serializable 序列化的接口：

```
public class WordEntity implements Serializable
```

Android 的 activity 和 fragment 没有提供直接传递对象的方法，但 Intent 中的 putExtra 方法以及 fragment 的 setArguments 可以传递一个 Serializable 类型的数据。因此为便于传递对象，需要为被传递的类实现 Serializable 接口。

生成 getter 和 setter 方法

WordEntity 的成员变量需要 getter 和 setter 方法。为避免手工输入，可使用 Android Studio 代码生成工具生成 getter 和 setter 方法。

代码生成工具能够自动生成构造器、getters、setters、equals()、hashCode()、toString()等方法。在使用代码生成工具之前，需要配置 Android Studio 设置 Java 的代码风格。点击 **File** →

Settings → Editor → Code Style → Java → Code Generation 得到如图 5-8 所示的设置对话框。

如果 **Field**（字段）和 **Static field**（静态字段）文本框不包含 m 和 s，在 **Naming**（命名）表中，选择 **Field** 行，如图 5-8 所示。添加 m 作为 **Field** 的 **Name prefix**（名称前缀），然后加 s 作为 **Static fields** 的 **Name prefix**。点击 **OK**。

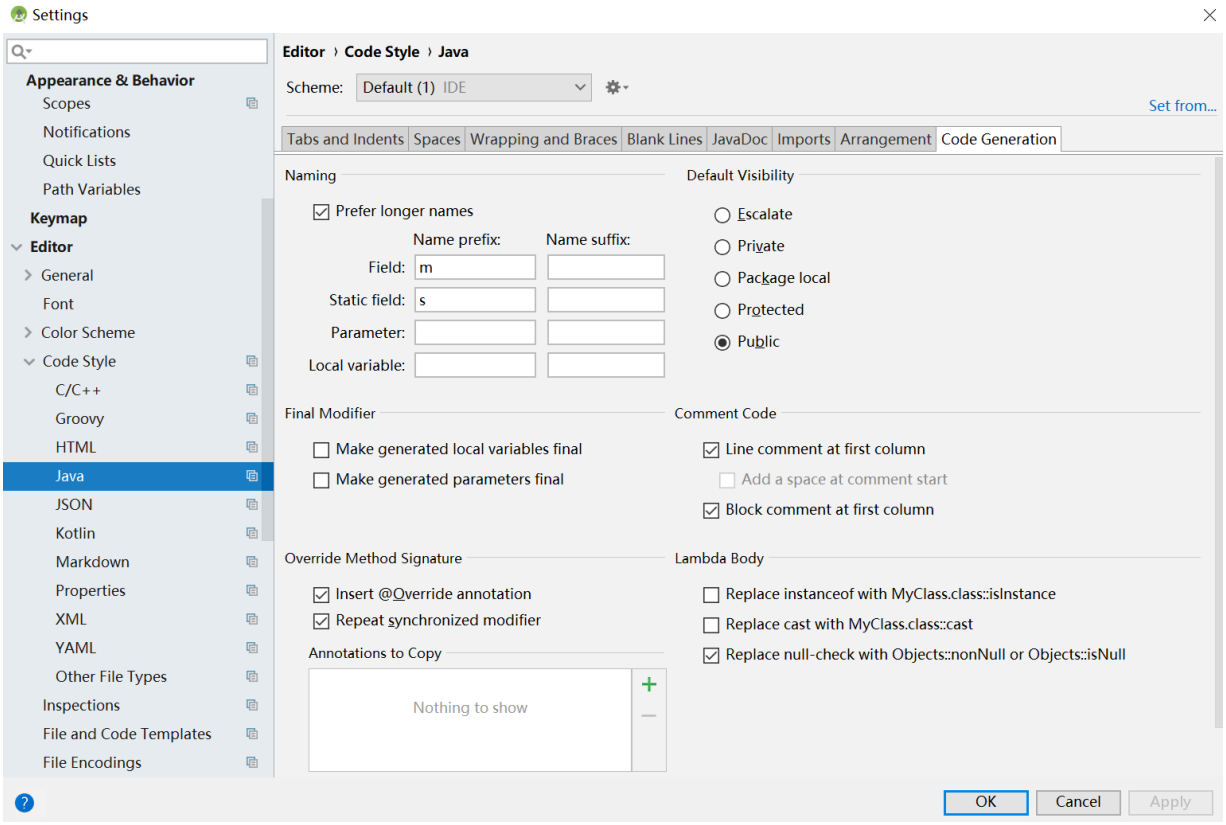

图 5-8 设置 Java 代码风格

刚才设置的前缀有何作用？如果不做此项设置，当要 Android Studio 为 mEnglish 生成 getter 方法时，它生成的将是 getMEnglish()，而不是 getEnglish()。

接下来，为只读成员变量 mId 生成一个获取方法，为其他成员变量生成获取方法和设置方法。

右键单击构造方法下面的空白处，选择 **Generate... → Getter** 菜单项，然后选择 mId 变量。再选择 **Generate... → Getter and Setter**，选中其余所有的变量，并点击 **OK**，Android Studio 随即生成了这 5 个 getter 和 setter 方法的代码，如代码清单 5-3 所示。

代码清单 **5-3 WordEntity** 类中的新增代码（**WordEntity.java**）

```
public class WordEntity implements Serializable {
    private UUID mId;
    boolean mKilled;
```

```
private String mEnglish;
private String mPartOfSpeech;
private String mChinese;
private String mExample;
private int mImgID;

public WordEntity(){
    //创建唯一标识符
    mId = UUID.randomUUID();
}

public WordEntity(String ve,String vp, String vc, String eg, int imgId){
    //创建唯一标识符
    mId = UUID.randomUUID();
    mEnglish = ve;
    mPartOfSpeech = vp;
    mChinese = vc;
    mExample = eg;
    mImgID = imgId;
}

public UUID getId() {
    return mId;
}
public boolean isKilled() {
    return mKilled;
}
public void setKilled(boolean killed) {
    mKilled = killed;
}
public String getEnglish() {
    return mEnglish;
}
public void setEnglish(String english) {
    mEnglish = english;
}
public String getPartOfSpeech() {
    return mPartOfSpeech;
}
public void setPartOfSpeech(String partOfSpeech) {
    mPartOfSpeech = partOfSpeech;
}
public String getChinese() {
    return mChinese;
}
public void setChinese(String chinese) {
    mChinese = chinese;
}
public String getExample() {
    return mExample;
}
public void setExample(String example) {
    mExample = example;
}
public int getImgID() {
    return mImgID;
```

```
    }
    public void setImgID(int imgID) {
        mImgID = imgID;
    }
}
```

这样，WordEntity 类就完成了。

5.3.3 利用向导创建 Fragment

UI fragment 的创建步骤

创建一个 UI fragment 的步骤与创建一个 activity 的步骤相同，具体步骤如下：

1. 通过向导创建 fragment，同时创建相应的布局文件；
2. 定义布局文件中的组件，组装界面；
3. 通过代码方式连接布局文件中生成的组件。

右键单击 com.studio.aime.necvocab 包，选择 **New → Fragment → Fragment**（**Blank**）。在弹出的对话框中命名类名为 HomeFragment，保留 **Create layout XML** 和 **Include fragment factory methods?**的勾选，取消 **Include interface callbacks** 的勾选，单击 **OK**。

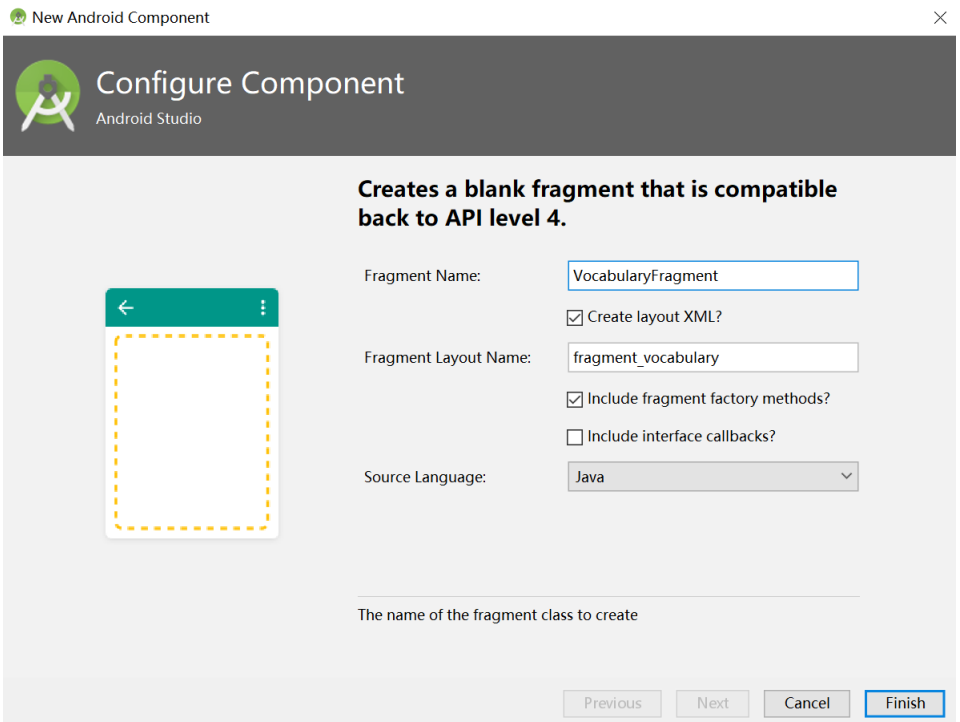

图 5-9 利用向导创建 Fragment

下面对创建过程进行一个简单说明：

创建过程很简单，包括三个 ***Optional*** *选项*

*（**1**） **Create layout XML***

*（**2**） **Include fragment factory methods***

*（**3**） **Include interface callbacks***

若默认都选，则 ***IDE*** *将创建一个* ***Fragment*** *类，及其对应的布局文件。而* ***Fragment*** *类中将包含以下内容:*

(a) *定义一个名为* ***OnFragmentInteractionListener*** *的* ***Interface***

(b) *重写的* ***onDetach()****方法*

(c) *重写的* ***onAttach()****方法*

(d) *重写的* ***onCreateView()****方法*

(e) *重写的* ***onCreate()****方法*

(f) *静态的* ***newInstance()****工厂类方法*

(g) *默认的无参构建方法* ***BlankFragment()***

(h) *自定义的* ***onButtonPressed()****方法（无意义）*

若不选择 ***Optional***（**1**），则不创建布局文件。

若不选择 ***Optional***（**2**），则不会包含内容（**e**），（**f**）。

若不选择 ***Optional***（**3**），则不会包含内容（**a**），（**b**），（**c**），（**h**）。

自动创建的 BlankFragment 布局文件，默认以 FrameLayout 作为根布局。可以改为其他布局类型,将 FrameLayout 放入父级 Activity 中,以方便与其他 Fragment 进行灵活地替换或删除。

向导为我们创建了默认的 VocabularyFragment 及布局文件 fragment_vocabulary.xml。我们首先定义 VocabularyFragment 的布局。

5.3.4 定义 VocabularyFragment 的布局

VocabularyFragment 视图将显示包含在 WordEntity 类实例中的信息。应用最终完成时，WordEntity 类及 VocabularyFragment 视图将包含很多内容。但本章，我们只需要包括:

- 一个 ImageView 组件，用于显示该词汇的图片（如果有）;
- 四个 TextView 组件，分别显示英文词汇，词性、中文释义和例句;
- 一个 CheckBox 组件，勾选该组件，表明已经掌握该词汇，今后该词汇将不再出现在学习列表中。

目前版本的 Android Studio 向导创建的 Fragment 的默认布局的根布局是 FrameLayout。我们希望能够使用图形编辑工具，因此，需要将 FrameLayout 转换为 ConstraintLayout。

在图像编辑工具的"组件树"栏，选中 FrameLayout，点击右键，在弹出窗口选择：Convert FrameLayout to ConstraintLayout。如图 5-10 所示。

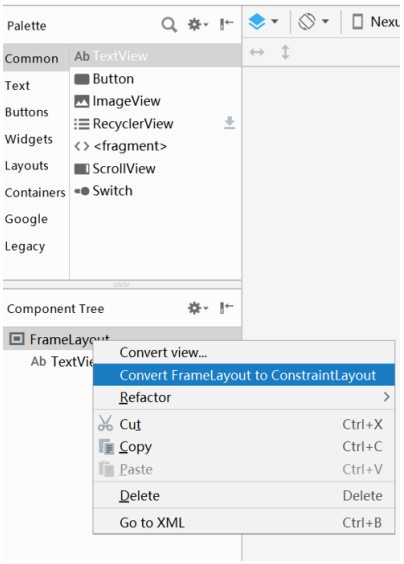

图 5-10 将 FrameLayout 转换为 ConstraintLayout

如图 5-11 所示，Android Studio 会弹出一个窗口，让你确认如何转化。这里是个简单窗口，不需要任何优化，所以接受默认值，点击 OK 按钮，确认转换。

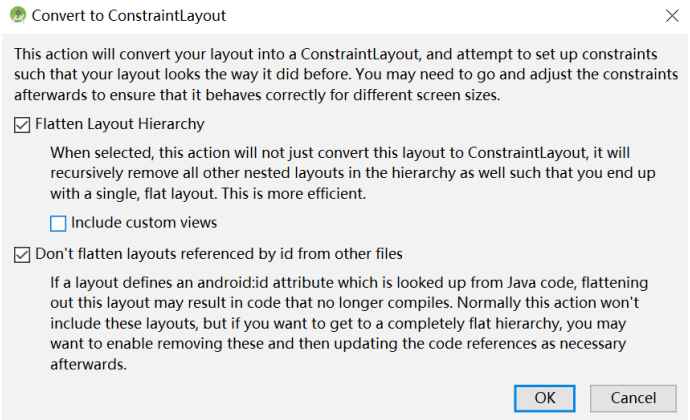

图 5-11 转换默认配置

接下来的工作和我们在第二、第三章中布局设计方法完全相同。图 5-12 显示了 VocabularyFragment 视图的布局。仿照图 5-12，在**组件面板**中选择相应的组件，拖拽到**预览**区的适当位置，**添加上约束**、**ID** 以及其他需要的属性。

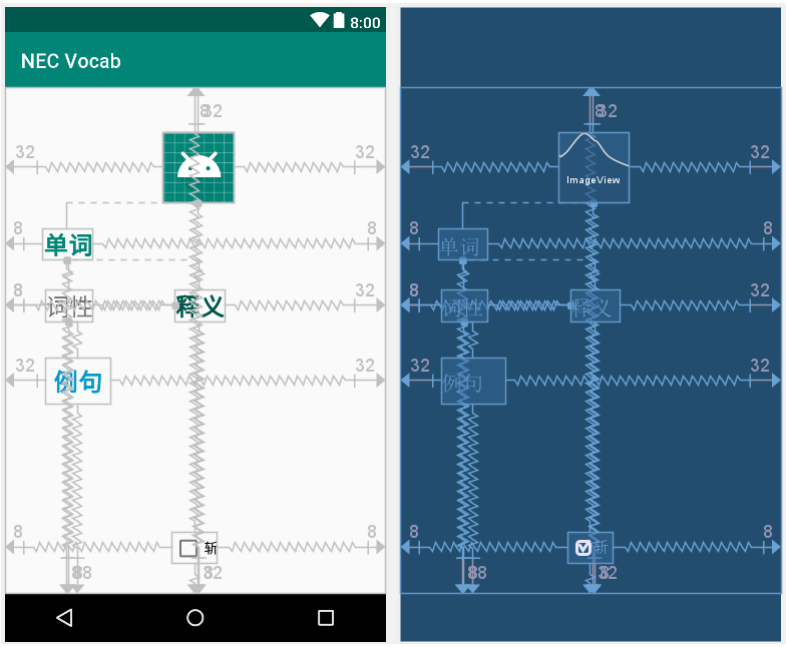

图 5-12 添加组件

由于不同读者添加的组件位置和约束位置各不相同，代码清单 5-4 主要列出各组件的 ID，以便后续引用时进行对照。

代码清单 **5-4** 定义组件（**fragment_vocabulary.xml**）

```
<?xml version="1.0" encoding="utf-8"?>
<android.support.constraint.ConstraintLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context=".VocabularyFragment">
    <TextView
        android:id="@+id/vocabulary_english"
        ……
    <ImageView
        android:id="@+id/imageview"
        ……
    <TextView
        android:id="@+id/part_of_speech"
        ……
    <TextView
        android:id="@+id/vocabulary_chinese"
        ……
    <TextView
        android:id="@+id/example_gratia"
        ……
    <CheckBox
        android:id="@+id/killed"
```

```
    ......
</android.support.constraint.ConstraintLayout>
```

5.4 实现 VocabularyFragment 类

打开向导创建的默认的 VocabularyFragment 类文件，其代码如下：

```
public class VocabularyFragment extends Fragment {
    private static final String ARG_PARAM1 = "param1";
    private static final String ARG_PARAM2 = "param2";
    private String mParam1;
    private String mParam2;
    public VocabularyFragment() {
    }

    public static VocabularyFragment newInstance(String param1, String param2) {
        VocabularyFragment fragment = new VocabularyFragment();
        Bundle args = new Bundle();
        args.putString(ARG_PARAM1, param1);
        args.putString(ARG_PARAM2, param2);
        fragment.setArguments(args);
        return fragment;
    }

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        if (getArguments() != null) {
            mParam1 = getArguments().getString(ARG_PARAM1);
            mParam2 = getArguments().getString(ARG_PARAM2);
        }
    }

    @Override
    public View onCreateView(LayoutInflater inflater, ViewGroup container,
                             Bundle savedInstanceState) {
        return inflater.inflate(R.layout.fragment_vocabulary, container, false);
    }
}
```

可以看到，利用向导创建的 VocabularyFragment 类需要完善如下工作：

1. 静态的 newInstance()工厂类方法；
2. 覆盖生命周期方法 onCreate()；
3. 覆盖生命周期方法 onCreateView()。

5.4.1 静态的 newInstance()工厂类方法

newInstance()方法是一种"静态工厂方法"，它封装和抽象了在客户端构造对象所需的步骤：

```
public static VocabularyFragment newInstance(String param1, String param2) {
    VocabularyFragment fragment = new VocabularyFragment();
    Bundle args = new Bundle();
    args.putString(ARG_PARAM1, param1);
    args.putString(ARG_PARAM2, param2);
    fragment.setArguments(args);
    return fragment;
}
```

上述代码其实就是在一个 Fragment 的 newInstance 方法中传递两个参数，并且通过 fragment.setArguments(args)保存在它自己身上，而后通过 onCreate()调用的时候将这些参数取出来。

这里提供静态工厂而不是使用默认构造函数或者自己定义一个有参的构造函数的至关重要一点是：fragmnet 经常会被销毁重新实例化，而 Android 只会调用 fragment 无参的构造函数。在系统自动实例化 fragment 的过程中，我们没有办法干预。一些需要外部传入的参数来决定的初始化就没有办法完成。使用静态工厂方法，将外部传入的参数可以通过 Fragment.setArguments(...)保存在它自己身上，这样我们可以在 Fragment.onCreate(...)调用的时候将这些参数取出来。

当然，在 NEC Vocab 项目中我们只需要传递一个来自词汇表的 WordEntity 的对象到词汇详解中（见第六章），而 activity 和 fragment 是通过 Serializable 类型数据传递对象的。因此，我们需要对向导创建的 newInstance()方法做如代码清单 5-5 所示的修改。

代码清单 5-5 向导创建的默认 VocabularyFragment（VocabularyFragment.java）

```
public class VocabularyFragment extends Fragment {
    private static final String ARG_PARAM = "param";
    private WordEntity mWord;

    public VocabularyFragment() {
    }
    public static VocabularyFragment newInstance(WordEntity param) {
        VocabularyFragment fragment = new VocabularyFragment();
        Bundle args = new Bundle();
        args.putSerializable(ARG_PARAM, param);
        fragment.setArguments(args);
        return fragment;
    }
.....
```

5.4.2 覆盖生命周期方法 onCreate()

VocabularyFragment 类是与模型和视图对象交互的控制器，用于显示特定 Word 的明细信息，并在用户修改这些信息后立即进行内容更新。

在注册登录模块中，activity 通过其生命周期方法完成了大部分逻辑控制工作。而在词汇模块中，这些工作将由 fragment 通过其生命周期方法完成。Fragment 的许多方法对应着 Activity 方法，包括我们熟知的 onCreate(Bundle)方法。

在 VocabularyFragment 类中新增一个 WordEntity 成员变量，然后实现 onCreate(Bundle)方法，如代码清单 5-6 所示。

代码清单 **5-6** 覆盖 **onCreate(Bundle)**方法（**VocabularyFragment.java**）

```
public class VocabularyFragment extends Fragment {
    ......
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        if (getArguments() != null) {
            mWord = (WordEntity)getArguments().getSerializable(ARG_PARAM);
        }
    }
......
```

以上实现代码中需要注意以下几点：

首先，Fragment 中的 onCreate(Bundle)方法是公共方法，而 Activity 中 onCreate(Bundle)方法是保护方法。因为需要被托管 fragment 的任何 activity 调用，因此，包括 onCreate(...)方法在内的 Fragment 生命周期方法必须设计为公共方法。

其次，类似于 activity，fragment 同样具有保存及获取状态的 Bundle。和使用 Activity 中的 onSaveInstanceState(Bundle) 方法一样，我们可以根据需要覆盖 Fragment 中的 onSaveInstanceState(Bundle)方法。

最后，通过 getArguments()的方法将 newInstance 方法中传递两个参数取出来。

5.4.3 覆盖生命周期方法 onCreateView()

回顾一下，在 Activity 的 onCreate(...)方法中，我们用 **setContentView(...)**获取 activity 用户界面，即通过传入布局的资源 ID，该方法生成指定布局的视图并将其放置在屏幕上。

但在 Fragment 的 onCreate(...)方法中并没有生成 fragment 视图，虽然 Fragment 的 onCreate(...)方法中配置了 fragment 实例，但创建和配置 fragment 视图是通过另一个 fragment 生命周期方法来完成的：

```
public View onCreateView(LayoutInflater inflater, ViewGroup parent, Bundle savedInstanceState)
```

通过该方法生成 fragment 视图的布局，然后将生成的 View 返回给托管 activity。LayoutInflater（布局生成器）及 ViewGroup 是用来生成布局的必要参数。Bundle 包含了供该方法在保存状态下重建视图所使用的数据。

在 VocabularyFragment.java 中，添加 onCreateView(...)方法的实现代码，从布局文件 fragment_vocabulary.xml 中产生并返回视图，如代码清单 5-7 所示。

代码清单 **5-7** 覆盖 **onCreateView(...)**方法（**VocabularyFragment.java**）

```
public class VocabularyFragment extends Fragment {
    ......
    @Override
    public View onCreateView(LayoutInflater inflater, ViewGroup container,
                             Bundle savedInstanceState) {
        View view = inflater.inflate(R.layout.fragment_vocabulary, container, false);
        return view;
    }
}
```

在 onCreateView(...)方法中，fragment 的视图是直接通过调用 LayoutInflater.inflate(...)方法并传入布局资源 ID 生成的。第二个参数是视图的父视图。通常我们需要父视图来正确配置组件。第三个参数告知布局生成器是否将生成的视图添加给父视图。这里，我们传入了 false 参数，因为我们将通过 activity 代码的方式添加视图。

注意：inflate()的作用就是将一个用 xml 定义的布局文件查找出来。注意与 findViewById()的区别，inflate 是加载一个布局文件，而 findViewById 则是从布局文件中查找一个控件。

5.4.4 在 fragment 中关联组件

onCreateView(...)方法也是生成组件并响应用户输入的地方。视图生成后，引用组件并添加对应的监听器方法。生成并使用组件的具体代码如代码清单 5-8 所示。

代码清单 **5-8** 关联组件（**VocabularyFragment.java**）

```
public class VocabularyFragment extends Fragment {
    private static final String ARG_PARAM = "param";
```

```
    private WordEntity mWord;
    private TextView mEnglishTextView;
    private TextView mPartOfSpeech;
    private TextView mChineseTextView;
    private TextView mExampleGratiaView;
    private ImageView mImageView;
    private CheckBox mKilledCheckBox;
    ......
    @Override
    public View onCreateView(LayoutInflater inflater, ViewGroup container,
                             Bundle savedInstanceState) {
        View view = inflater.inflate(R.layout.fragment_vocabulary, container, false);
        mEnglishTextView = (TextView)view.findViewById(R.id.vocabulary_english);
        mPartOfSpeech = (TextView) view.findViewById(R.id.part_of_speech);
        mChineseTextView = (TextView)view.findViewById(R.id.vocabulary_chinese);
        mExampleGratiaView = (TextView)view.findViewById(R.id.example_gratia);
        mImageView = (ImageView) view.findViewById(R.id.imageview);
        mKilledCheckBox = (CheckBox) view.findViewById(R.id.killed);
        mKilledCheckBox.setOnCheckedChangeListener(
                new CompoundButton.OnCheckedChangeListener() {
                    @Override
                    public void onCheckedChanged(CompoundButton buttonView,
                                                 boolean isKilled) {
                        mWord.setKilled(isKilled);
                    }
                });
        updateWords();
        return view;
    }

    private void updateWords(){
        mWord.setKilled(false);
        mKilledCheckBox.setChecked(mWord.isKilled());
        mEnglishTextView.setText(mWord.getEnglish());
        mPartOfSpeech.setText(mWord.getPartOfSpeech());
        mChineseTextView.setText(mWord.getChinese());
        mExampleGratiaView.setText(mWord.getExample());
        mImageView.setImageResource(mWord.getImgID());
    }
}
```

Fragment 中的 onCreateView(...)方法中的组件引用几乎等同于 Activity 的 onCreate(...)方法的处理。唯一的区别是我们调用了 fragment 视图的 View.findViewById(int)方法。Activity 中直接使用 findViewById(int)方法十分便利，能够在后台自动调用 View.findViewById(int)方法。而 Fragment 类没有对应的便利方法，因此我们必须自己完成调用。

fragment 中监听器方法的设置和 activity 中的处理完全相同。如代码清单 5-8 所示，创建实现 CheckBox 监听器接口的匿名内部类。在代码中引用它并设置监听器用于更新 WordEntity 的 mKilled 变量值。

最后，我们使用一个 WordEntity 实例，并将其存放在 VocabularyFragment 类的成员变量

（mWord）中。为此，我们需要照一张图片，这里我们使用了一个小海豚的图片 dolphin.png（你可以使用任意一张图片），将其存放在 res/drawable 目录下。添加后的 VocabularyFragment.java 的完整代码如代码清单 5-9 所示代码。

代码清单 **5-9** 添加一个 **WordEntity** 实例（**VocabularyFragment.java**）

```
public class VocabularyFragment extends Fragment {
    private static final String ARG_PARAM = "param";

    private WordEntity mWord;
    private TextView mEnglishTextView;
    private TextView mPartOfSpeech;
    private TextView mChineseTextView;
    private TextView mExampleGratiaView;
    private ImageView mImageView;
    private CheckBox mKilledCheckBox;

    public VocabularyFragment() {
    }

    public static VocabularyFragment newInstance(WordEntity param) {
        VocabularyFragment fragment = new VocabularyFragment();
        Bundle args = new Bundle();
        args.putSerializable(ARG_PARAM, param);
        fragment.setArguments(args);
        return fragment;
    }

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        if (getArguments() != null) {
            mWord = (WordEntity)getArguments().getSerializable(ARG_PARAM);
        } else {
            mWord = new WordEntity();
        }
    }

    @Override
    public View onCreateView(LayoutInflater inflater, ViewGroup container,
                             Bundle savedInstanceState) {
        View view = inflater.inflate(R.layout.fragment_vocabulary, container, false);
        mEnglishTextView = (TextView)view.findViewById(R.id.vocabulary_english);
        mPartOfSpeech = (TextView) view.findViewById(R.id.part_of_speech);
        mChineseTextView = (TextView)view.findViewById(R.id.vocabulary_chinese);
        mExampleGratiaView = (TextView)view.findViewById(R.id.example_gratia);
        mImageView = (ImageView) view.findViewById(R.id.imageview);
        mKilledCheckBox = (CheckBox) view.findViewById(R.id.killed);

        mKilledCheckBox.setOnCheckedChangeListener(
                new CompoundButton.OnCheckedChangeListener() {//自动引入 CompoundButton

                    @Override
```

```
                public void onCheckedChanged(CompoundButton buttonView,
                                             boolean isKilled) {
                    mWord.setKilled(isKilled);
                }
            });
        updateWords();
        return view;
    }

    private void updateWords(){
        mWord.setEnglish("dolphin");
        mWord.setPartOfSpeech("noun.");
        mWord.setChinese("海豚");
        mWord.setImgID(R.drawable.dolphin);
        mWord.setExample("A dolphin is a mammal which lives in the sea.");
        mWord.setKilled(false);
        mKilledCheckBox.setChecked(mWord.isKilled());
        mEnglishTextView.setText(mWord.getEnglish());
        mPartOfSpeech.setText(mWord.getPartOfSpeech());
        mChineseTextView.setText(mWord.getChinese());
        mExampleGratiaView.setText(mWord.getExample());
        mImageView.setImageResource(mWord.getImgID());
    }
}
```

至此，我们已经完成了 VocabularyFragment 类的代码。但现在还不能运行应用查看用户界面和检查代码。因为 fragment 无法将自己的视图显示在屏幕上。接下来我们需要把 VocabularyFragment 添加到 MainActivity。

但目前我们只有一个 VocabularyFragment，我们还需要诸如 HomeFragment、DiscoverFragment 和 MineFragment 等用来展现“主页”、“发现”和“我的”。

5.4.5 创建更多的 Fragment

和创建 VocabularyFragment 一样，右键单击 com.studio.aime.necvocab 包，选择 New → Fragment → Fragment（Blank）。在弹出的对话框中将 Fragment Name：栏命名为 HomeFragment，保留 Create layout XML 的勾选，取消 Include fragment factory methods? 和 Include interface callbacks 的勾选，单击 OK。

这样创建的 HomeFragment 类里面只剩下一个 onCreateView(...)方法（当然还有个空的构造函数，暂时不考虑它）。所添加的 HomeFragment 类如代码清单 5-10 所示。

代码清单 **5-10** 创建空白 **Fragment**（**HomeFragment.java**）

```
public class HomeFragment extends Fragment {
    public HomeFragment() {
```

```
    }
    @Override
    public View onCreateView(LayoutInflater inflater, ViewGroup container,
                             Bundle savedInstanceState) {
        return inflater.inflate(R.layout.fragment_home, container, false);
    }
}
```

用同样的方法再分别创建三个名为 VocabFragment、DiscoverFragment 和 MineFragment 的 Fragment。其代码形式和代码清单 5-10 完全类似。

当然，也可以对这三个 Fragment 的布局文件进行适当的设计，使其具备明细的区分度，例如将其背景设为蓝色等：

```
android:background="@android:color/holo_blue_bright"
```

组件 TextView 的 text 设为"这是主页"等等，如图 5-13 所示。

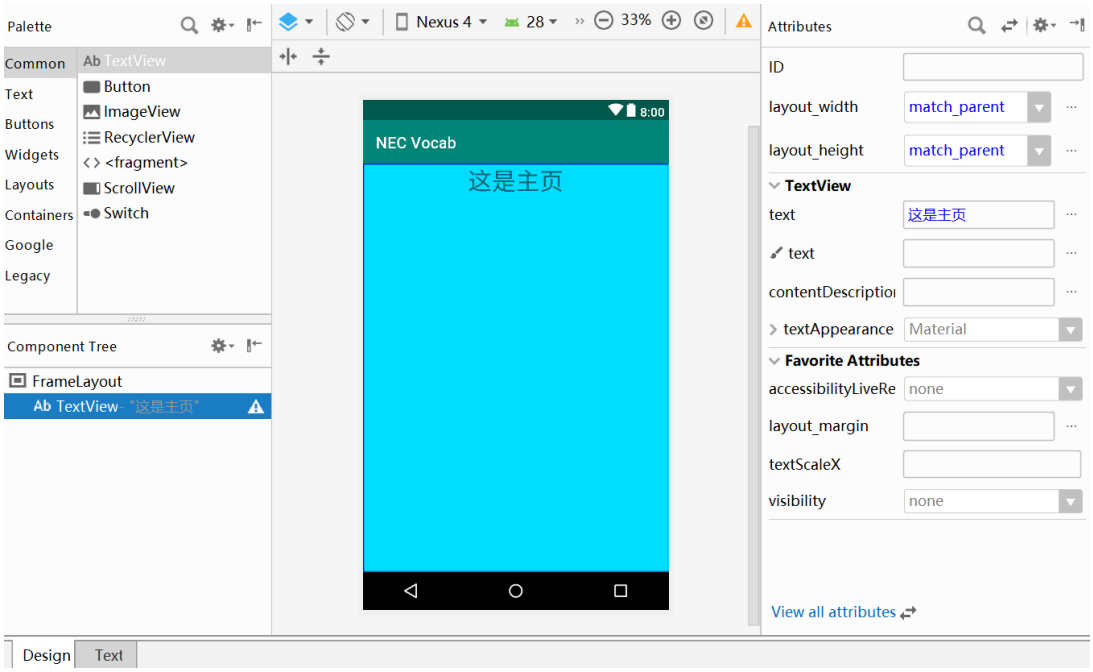

图 5-13 设置 fragment_home 布局

5.5 向 FragmentManager 添加 Fragment

要托管 Fragment，我们需要对 MainActivity 类进行修改，使其包含 FragmentManager 类。FragmentManager 类负责管理 fragment 并将它们的视图添加到 activity 的视图层级结构中。

FragmentManager 类具体管理的是：

- fragment 队列；
- fragment 事务的回退栈（这一点稍后会学到）。

FragmentManager 的关系如图 5-14 所示。

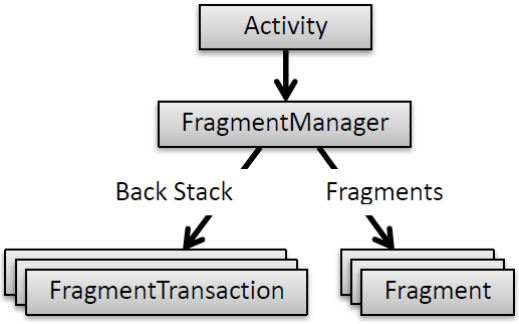

图 5-14 FragmentManager 关系图

目前只需关心 FragmentManager 管理的 fragment 队列。

要通过代码方式将 fragment 添加到 activity 中，可以直接调用 activity 的 FragmentManager。为此我们首先要获取 FragmentManager。

获取 FragmentManager 的方法是 **getSupportFragmentManager()**。

FragmentTransaction（fragment 事务）

FragmentTransaction 用来添加、移除、附加、分离或替换 fragment 队列中的 fragment。这是使用 fragment 在运行时组装和重新组装用户界面的核心方式。

FragmentManager 的 beginTransaction()方法用于创建并返回 FragmentTransaction 实例。由此可以得到一个 FragmentTransaction 队列。

因此添加 Fragment 的过程是：

- 创建一个新的 fragment 事务——beginTransaction()方法；
- 加入一个添加操作——add(...)或 replace (...)方法；
- 然后提交该事务——commit()方法。

add(...)/replace (...)方法是整个事务的核心，该方法含有两个参数：容器视图资源 ID 和新创建的 VocabularyFragment。容器视图资源 ID 是定义在 activity_vocabulary.xml 中的 FrameLayout 组件的资源 ID，主要有两个作用：

- 告知 FragmentManager，fragment 视图在 activity 视图中的位置；
- 是 FragmentManager 队列中 fragment 的唯一标识符。

如需从 FragmentManager 中获取 VocabularyFragment，可以使用容器视图资源 ID，如下列代码所示。

```
FragmentManager fm = getSupportFragmentManager();
Fragment fragment = fm.findFragmentById(R.id.fragment_container);
fragment = VocabularyFragment.newInstance(null,null);
fm.beginTransaction().replace(R.id.fragment_container,fragment).commit();
```

或用更优雅的方式：

```
getSupportFragmentManager()
        .beginTransaction()
        .replace(R.id.fragment_container,VocabularyFragment.newInstance(null,null))
        .commit();
```

将上述代码添加的 MainActivity 中，由于 BottomNavigationView 在其实现中，对于采用 replace(...)方法切换的 fragment，已经考虑了设备状态的保存，因此，这里只需要考虑初次进入时显示主页，即将 savedInstanceState != null)替换为 savedInstanceState == null，并删除 onCreate 方法里的 switch 语句及相关内容。如代码清单 5-11 所示，。

代码清单 5-11 添加 VocabularyFragment（MainActivity.java）

```
public class MainActivity extends AppCompatActivity {
    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,VocabularyFragment.newInstance(new
                                    WordEntity()))
                            .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);
            mCurrentIndex = R.id.navigation_home;
            if (savedInstanceState == null){
                mCurrentIndex = savedInstanceState.getInt(KEY_INDEX, 0);
                getSupportFragmentManager()
                        .beginTransaction()
                        .replace(R.id.fragment_container,new HomeFragment())
                        .commit();
            }
            switch (mCurrentIndex) {
                ......
            }
            BottomNavigationView navigation = (BottomNavigationView) findViewById(R.id.navigation);
            navigation.setOnNavigationItemSelectedListener(mOnNavigationItemSelectedListener);
        }
        @Override
        public void onSaveInstanceState(Bundle savedInstanceState) {
            super.onSaveInstanceState(savedInstanceState);
            savedInstanceState.putInt(KEY_INDEX, mCurrentIndex);
        }
}
```

运行 NEC Vocab 应用，应该看到如图 5-15 所示结果。

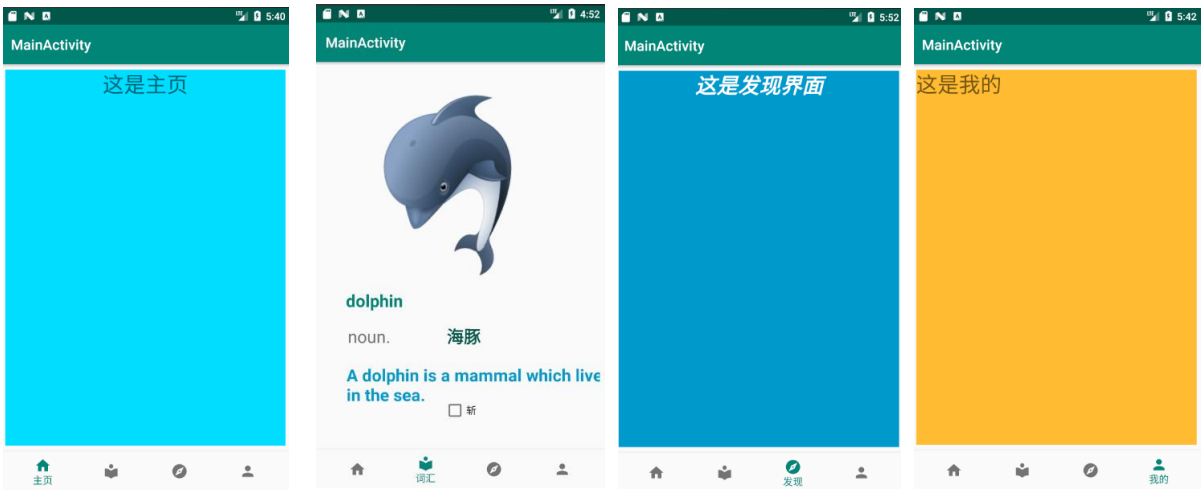

图 5-15 运行效果

5.6 创建水平布局

下面为设备配置变更新建备选资源，只要设备旋转至水平方位，Android 就会自动发现并使用它。由于在第二章我们已经为 NEC Vocab 应用创建了水平布局的目录 layout-land，现在只需要将布局文件 fragment_vocabulary.xnl 复制到该目录中即可。

接下来打开 layout-land/fragment_vocabulary.xnl，在 Design 标签页下修改该文件布局，如图 5-16 所示（当然你可以设计一个更美观的布局）。

图 5-16 词汇详解的水平布局

再次运行 NEC Vocab 应用。旋转设备至水平方位，查看新的布局界面，如图 5-17 所示。当然，这不仅是一个新的布局界面，也是一个新的 MainActivity 及其托管的 VocabularyFragment。

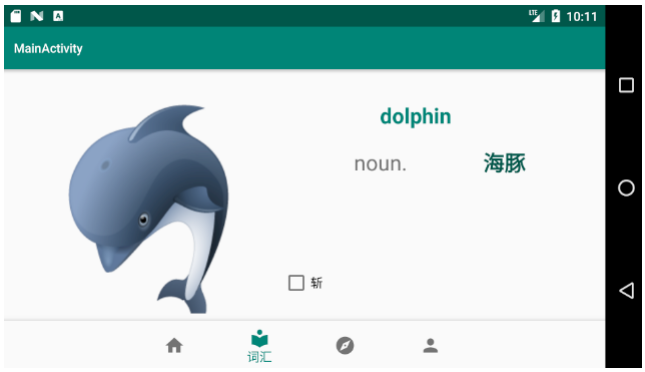

图 5-17 处于水平方位的 VocabularyFragment

设备旋转回竖直方位，可看到默认的布局界面以及另一个新的 MainActivity 及其托管的 VocabularyFragment。