第 9 章

使用ListFragment显示列表

9

当前，CriminalIntent应用的模型层仅包含一个Crime实例。本章，我们将更新CriminalIntent应用以包含一个crime列表，如图9-1所示。列表显示每一个Crime实例的标题、发生日期以及处理状态。

图9-1　crime列表

图9-2展示了本章CriminalIntent应用的整体规划设计。

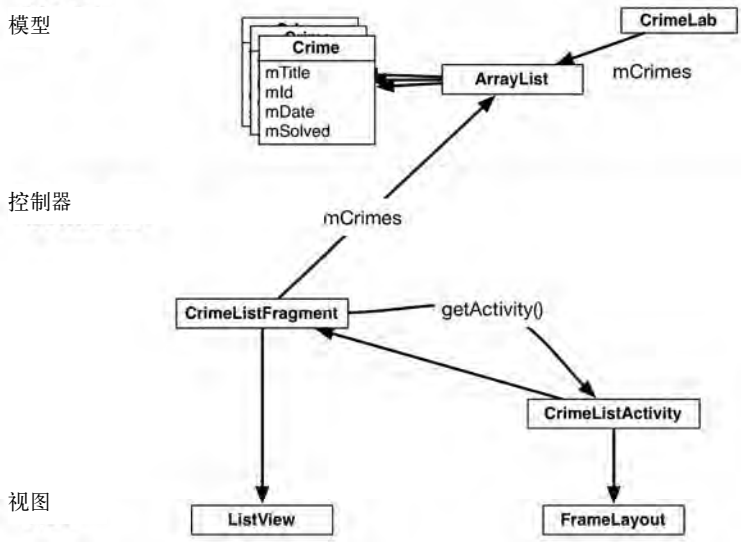

图9-2 CriminalIntent应用对象图

应用的模型层将新增一个CrimeLab对象，该对象是一个数据集中存储池，用来存储Crime对象。

显示crime列表需在应用的控制层新增一个activity和一个fragment，即CrimeListActivity和CrimeListFragment。

CrimeListFragment是ListFragment的子类，ListFragment是Fragment的子类。Fragment内置列表显示支持功能。控制层对象间、控制层对象与CrimeLab对象间彼此交互，进行模型层数据的存取。

（图9-2中怎么没看到CrimeActivity和CrimeFragment呢？因为它们是明细视图相关的类，所以，这里我们没有显示它们。第10章，我们将学习如何将CriminalIntent应用的列表视图与明细视图进行关联。）

图9-2中，也可以看到与CrimeListActivity和CrimeListFragment关联的视图对象。activity视图由包含fragment的FrameLayout组成。fragment视图由一个ListView组成。稍后，我们将学习到更多有关ListFragment与ListView间交互方式的内容。

9.1 更新 CriminalIntent 应用的模型层

首先，我们来更新应用的模型层，从原来的单个Crime对象变为可容纳多个Crime对象的ArrayList。

ArrayList<E>是一个支持存放指定数据类型对象的Java有序数组类，具有获取、新增及删除数组中元素的方法。

单例与数据集中存储

在CriminalIntent应用中，crime数组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crime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。

要创建单例，需创建一个带有私有构造方法及get()方法的类，其中get()方法返回实例。如实例已存在，get()方法则直接返回它；如实例还未存在，get()方法会调用构造方法来创建它。

右键单击com.bignerdranch.android.criminalintent类包，选择New → Class菜单项。在随后出现的对话框中，命名类为CrimeLab，然后单击Finish按钮完成。

在打开的CrimeLab.java文件中，编码实现CrimeLab类为带有私有构造方法和get(Context)方法的单例，如代码清单9-1所示。

代码清单9-1　创建单例（CrimeLab.java）

```
public class CrimeLab {
    private static CrimeLab sCrimeLab;
    private Context mAppContext;

    private CrimeLab(Context appContext) {
        mAppContext = appContext;
    }

    public static CrimeLab get(Context c) {
        if (sCrimeLab == null) {
            sCrimeLab = new CrimeLab(c.getApplicationContext());
        }
        return sCrimeLab;
    }
}
```

注意sCrimeLab变量的s前缀。这是Android开发的命名约定，通过该前缀，很容易得知sCrimeLab是一个静态变量。

CrimeLab类的构造方法需要一个Context参数。这在Android开发里很常见，使用Context参数，单例可完成启动activity、获取项目资源，查找应用的私有存储空间等任务。

注意，在get(Context)方法里，我们并没有直接将Context参数传给构造方法。该Context可能是一个Activity，也可能是另一个Context对象，如Service。在应用的整个生命周期里，我们无法保证只要CrimeLab需要用到Context，Context就一定会存在。

因此，为保证单例总是有Context可以使用，可调用getApplicationContext()方法，将不确定是否存在的Context替换成application context。application context是针对应用的全局性Context。任何时候，只要是应用层面的单例，就应该一直使用application context。

下面，我们将一些Crime对象保存到CrimeLab中去。在CrimeLab的构造方法里，创建一个空的用来保存Crime对象的ArrayList。此外，再添加两个方法，即getCrimes()方法和

getCrime(UUID)方法。前者返回数组列表，后者返回带有指定ID的Crime对象。具体代码如代码清单9-2所示。

代码清单9-2 创建可容纳Crime对象的ArrayList（CrimeLab.java）

```
public class CrimeLab {
    private ArrayList<Crime> mCrimes;

    private static CrimeLab sCrimeLab;
    private Context mAppContext;

    private CrimeLab(Context appContext) {
        mAppContext = appContext;
        mCrimes = new ArrayList<Crime>();
    }

    public static CrimeLab get(Context c) {
        ...
    }

    public ArrayList<Crime> getCrimes() {
        return mCrimes;
    }

    public Crime getCrime(UUID id) {
        for (Crime c : mCrimes) {
            if (c.getId().equals(id))
                return c;
        }
        return null;
    }
}
```

最后，新建的ArrayList将内含用户自建的Crime，用户既可以存入Crime，也可以从中调取Crime。但现在，我们暂时先往数组列表中批量存入100个Crime对象，如代码清单9-3所示。

代码清单9-3 生成100个crime（CrimeLab.java）

```
private CrimeLab(Context appContext) {
    mAppContext = appContext;
    mCrimes = new ArrayList<Crime>();
    for (int i = 0; i < 100; i++) {
        Crime c = new Crime();
        c.setTitle("Crime #" + i);
        c.setSolved(i % 2 == 0); // Every other one
        mCrimes.add(c);
    }
}
```

现在我们拥有了一个满是数据的模型层，和100个可在屏幕上显示的crime。

9.2 创建 ListFragment

创建一个名为CrimeListFragment的类。单击Browse按钮选择超类。查找并选择ListFragment– android.support.v4.app，然后单击Finish完成CrimeListFragment类的创建。

Honeycomb系统版本引入了ListFragment类，相应的，支持库也引入了该类。因此，只要记得使用支持库中的android.support.v4.app.ListFragment类，就可以避免不同系统版本的兼容性问题。

在CrimeListFragment.java中，覆盖onCreate(Bundle)方法，设置托管CrimeListFragment的activity标题，如代码清单9-4所示。

代码清单9-4　为新activity添加onCreate(Bundle)方法（CrimeListFragment.java）

```
public class CrimeListFragment extends ListFragment {

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        getActivity().setTitle(R.string.crimes_title);
    }

}
```

注意查看getActivity()方法。该Fragment便利方法不仅可以返回托管activity，且允许fragment处理更多的activity相关事务。这里，我们使用它调用Activity.setTitle(int)方法，将显示在操作栏（旧版本设备上为标题栏）上的标题文字替换为传入的字符串资源中设定的文字。

现在，无需覆盖onCreateView(...)方法或为CrimeListFragment生成布局。ListFragment类默认实现方法已生成了一个全屏ListView布局。我们先暂时使用该布局，后续章节中我们会覆盖CrimeListFragment.onCreateView(...)方法，从而添加更多的高级功能。

在strings.xml文件中，为列表activity标题添加字符串资源，如代码清单9-5所示。

代码清单9-5　为新的activity标题添加字符串资源（strings.xml）

```
    ...

    <string name="crime_solved_label">Solved?</string>
    <string name="crimes_title">Crimes</string>
</resources
```

CrimeListFragment需要获取存储在CrimeLab中的crime列表。在CrimeListFragment.onCreate(...)方法中，先获取CrimeLab单例，再获取其中的crime列表，如代码清单9-6所示。

代码清单9-6　在CrimeListFragment中获取crime（CrimeListFragment.java）

```
public class CrimeListFragment extends ListFragment {
    private ArrayList<Crime> mCrimes;

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        getActivity().setTitle(R.string.crimes_title);
        mCrimes = CrimeLab.get(getActivity()).getCrimes();
    }

}
```

9.3 使用抽象 activity 托管 fragment

下面我们来创建一个用于托管CrimeListFragment的CrimeListActivity类。首先为CrimeListActivity创建一个视图。

9.3.1 通用的fragment托管布局

对于CrimeListActivity，我们仍可使用定义在activity_crime.xml文件中的布局。该布局提供了一个用来放置fragment的FrameLayout容器视图，其中的fragment可在activity中使用代码获取，如代码清单9-7所示。

代码清单9-7 通用的布局定义文件activity_crime.xml

```
<?xml version="1.0" encoding="utf-8"?>
<FrameLayout xmlns:android="http://schemas.android.com/apk/res/android"
  android:id="@+id/fragmentContainer"
  android:layout_width="match_parent"
  android:layout_height="match_parent"
  />
```

activity_crime.xml布局文件并没有指定一个特定的fragment，因此只要有activity托管一个fragment，就可以使用该布局文件。下面，为反映出该布局的通用性，我们把该布局文件重命名为activity_fragment.xml。

首先，如果已打开了activity_crime.xml文件，请先在编辑区关闭它。接下来，在包浏览器中，右键单击res/layout/activity_crime.xml文件。（注意是单击activity_crime.xml文件，而不是fragment_crime.xml。）

在弹出的菜单里，选择Refactor → Rename...菜单项将activity_crime.xml改名为activity_ fragment.xml。重命名资源时，Eclipse会自动更新资源文件的所有引用。

如使用的是旧版本ADT，则资源重命名后，Eclipse不会自动更新引用代码。如Eclipse报告CrimeActivity.java代码有错，则需在CrimeActivity文件中手工更新引用代码，如代码清单9-8所示。

代码清单9-8 为CrimeActivity更新布局文件引用（CrimeActivity.java）

```
public class CrimeActivity extends FragmentActivity {
    /** Called when the activity is first created. */
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        ~~setContentView(R.layout.activity_crime);~~
        setContentView(R.layout.activity_fragment);
        FragmentManager fm = getSupportFragmentManager();
        Fragment fragment = fm.findFragmentById(R.id.fragmentContainer);

        if (fragment == null) {
            fragment = new CrimeFragment();
```

```
            fm.beginTransaction()
                .add(R.id.fragmentContainer, fragment)
                .commit();
        }
    }
}
```

9.3.2　抽象 activity 类

可复用CrimeActivity的代码来创建CrimeListActivity类。回顾一下前面写的CrimeActivity类代码，如代码清单9-8所示。该类代码简单且几近通用。事实上，CrimeActivity类代码唯一不通用的地方是CrimeFragment类在添加到FragmentManager之前的实例化代码部分，如代码清单9-9所示。

代码清单9-9　几近通用的CrimeActivity类（CrimeActivity.java）

```
public class CrimeActivity extends FragmentActivity {
    /** Called when the activity is first created. */
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_fragment);
        FragmentManager fm = getSupportFragmentManager();
        Fragment fragment = fm.findFragmentById(R.id.fragmentContainer);

        if (fragment == null) {
            fragment = new CrimeFragment();
            fm.beginTransaction()
                .add(R.id.fragmentContainer, fragment)
                .commit();
        }
    }
}
```

本书中几乎每一个创建的activity都需要同样的代码。为避免不必要的重复性输入，我们将这些重复代码置入一个抽象类，以供使用。

在CriminalIntent类包里创建一个名为SingleFragmentActivity的新类。选择FragmentActivity类作为它的超类，然后勾选abstract选项，让SingleFragmentActivity类成为一个抽象类，如图9-3所示。

单击Finish按钮完成创建。添加代码清单9-10所示代码到SingleFragmentActivity.java文件。可以看到，除了加亮部分代码，其余代码和原来的CrimeActivity代码完全一样。

代码清单9-10　添加一个通用超类（SingleFragmentActivity.java）

```
public abstract class SingleFragmentActivity extends FragmentActivity {
    protected abstract Fragment createFragment();

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_fragment);
        FragmentManager fm = getSupportFragmentManager();
        Fragment fragment = fm.findFragmentById(R.id.fragmentContainer);
```

```
        if (fragment == null) {
            fragment = createFragment();
            fm.beginTransaction()
                .add(R.id.fragmentContainer, fragment)
                .commit();
        }
    }
}
```

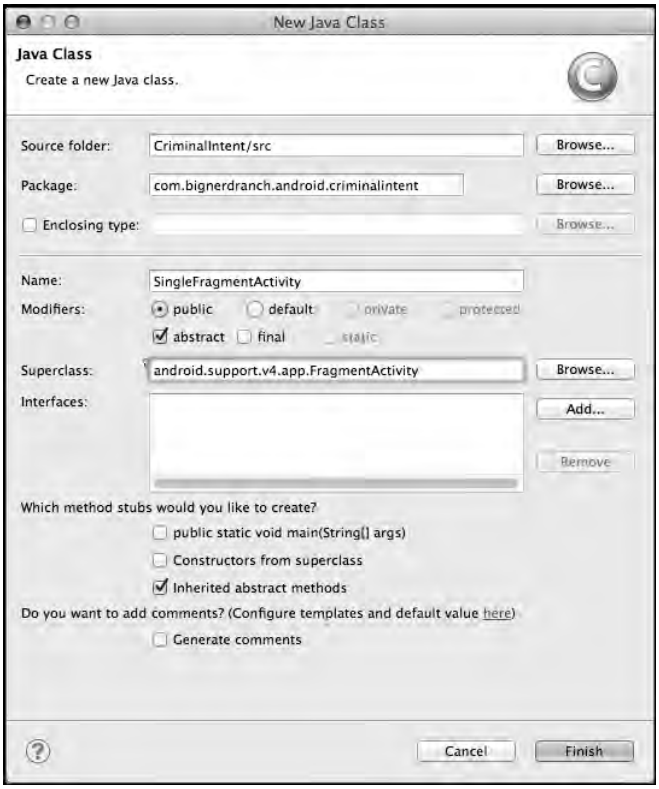

图9-3 创建SingleFragmentActivity抽象类

在以上代码里，我们设置从activity_fragment.xml布局里生成activity视图。然后在容器中寻找FragmentManager里的fragment。如fragment不存在，则创建一个新的fragment并将其添加到容器中。

代码清单9-10与原来的CrimeActivity代码唯一的区别就是，新增了一个名为createFragment()的抽象方法，我们可使用它实例化新的fragment。SingleFragmentActivity的子类会实现该方法返回一个由activity托管的fragment实例。

1. 使用抽象类

下面我们来测试一下抽象类的使用。首先创建一个名为CrimeListActivity的新类。在新建类向导窗口的设置超类栏位处，单击Browse按钮弹出超类选择对话框，输入SingleFragmentActivity，

Eclispe会自动按照输入提供超类选项，选择需要的超类后单击OK按钮确认（如图9-4所示）。最后单击Finish按钮完成创建。

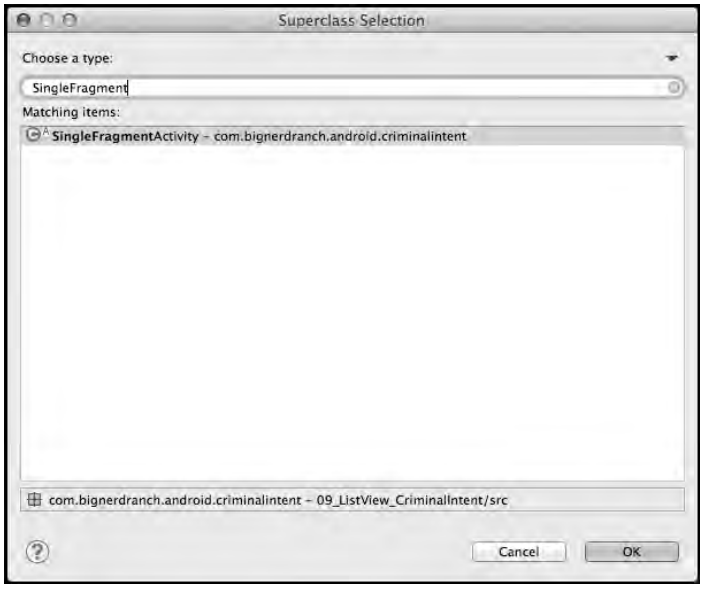

图9-4　选择SingleFragmentActivity超类

Eclipse随后打开了CrimeListActivity.java文件，可以看到，代码中有了一个待实现的createFragment()方法体。实现该方法使其能够返回一个新的CrimeListFragment实例，如代码清单9-11所示。

代码清单9-11　代码实现CrimeListActivity（CrimeListActivity.java）

```
public class CrimeListActivity extends SingleFragmentActivity {

    @Override
    protected Fragment createFragment() {
        return new CrimeListFragment();
    }

}
```

如果CrimeActivity类也可以继承通用类来实现，那就再好不过了。返回到CrimeActivity.java文件中。参照代码清单9-12，删除CrimeActivity类的现有代码，重新编写代码，使其成为SingleFragmentActivity的子类。

代码清单9-12　清理CrimeActivity类（CrimeActivity.java）

```
public class CrimeActivity extends FragmentActivity SingleFragmentActivity {
    /** Called when the activity is first created. */
    @Override
    public void onCreate(Bundle savedInstanceState) {
```

```
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_fragment);
        FragmentManager fm = getSupportFragmentManager();
        Fragment fragment = fm.findFragmentById(R.id.fragmentContainer);

        if (fragment == null) {
            fragment = new CrimeFragment();
            fm.beginTransaction()
                .add(R.id.fragmentContainer, fragment)
                .commit();
        }
    }

    @Override
    protected Fragment createFragment() {
        return new CrimeFragment();
    }
}
```

在本书的后续章节中，我们会发现使用SingleFragmentActivity抽象类可大大减少代码输入量，节约开发时间。现在，我们的activity代码看起来简练又整洁。

2. 在配置文件中声明CrimeListActivity

CrimeListActivity创建完成后，记得在配置文件中声明它。另外，为实现CriminalIntent应用启动后，用户看到的主界面是crime列表，我们还需配置CrimeListActivity为启动activity。

如代码清单9-13所示，在manifest配置文件中，首先声明CrimeListActivity，然后删除CrimeActivity的启动activity配置，改配CrimeListActivity为启动activity。

代码清单9-13 声明CrimeListActivity为启动activity（AndroidManifest.xml）

```
    ...

    <application
      android:allowBackup="true"
      android:icon="@drawable/ic_launcher"
      android:label="@string/app_name"
      android:theme="@style/AppTheme" >
      <activity android:name=".CrimeListActivity">
        <intent-filter>
          <action android:name="android.intent.action.MAIN" />
          <category android:name="android.intent.category.LAUNCHER" />
        </intent-filter>
      </activity>
      <activity android:name=".CrimeActivity"
        android:label="@string/app_name">
        <intent-filter>
          <action android:name="android.intent.action.MAIN" />
          <category android:name="android.intent.category.LAUNCHER" />
        </intent-filter>
      </activity>

    </application>

</manifest>
```

现在，CrimeListActivity被配置为了启动activity。运行CriminalIntent应用，会看到CrimeListActivity的FrameLayout托管了一个无内容的CrimeListFragment，如图9-5所示。

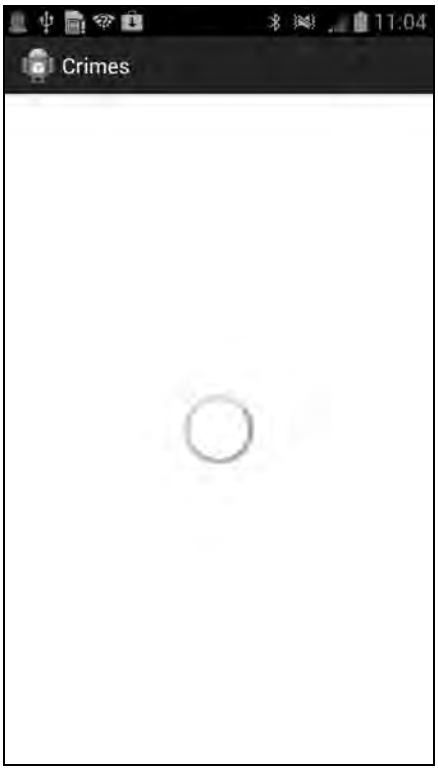

图9-5　没有内容的CrimeListActivity用户界面

当ListView没有内容可以显示时，ListFragment会通过内置的ListView显示一个圆形进度条。CrimeListFragment已经被赋予了访问Crime数组的能力，接下来，我们要将crime列表通过ListView显示在屏幕上。

9.4　ListFragment、ListView 及 ArrayAdapter

我们需要通过CrimeListFragment的ListView将列表项展示给用户，而不是什么圆形进度条。每一个列表项都应该包含一个Crime实例的数据。

ListView是ViewGroup的子类，每一个列表项都是作为ListView的一个View子对象显示的。这些View子对象既可以是复杂的View对象，也可以是简单的View对象，这取决于我们对列表显示复杂度的需要。

首先我们要实现的是一个简单形式的列表项显示，即每个列表项只显示Crime的标题，并且View对象是一个简单的TextView，如图9-6所示。

上图中，我们可以看到12个TextView，其中第12个TextView只显示了一半。要是能滚动截图屏幕的话，ListView还可显示出更多的TextView，如Crime #12、Crime #13等。

图9-6 带有子TextView的ListView

这些View对象来自哪里？ListView会提前准备好所有要显示的View对象吗？倘若这样，效率可就太低了。其实View对象只有在屏幕上显示时才有必要存在。列表的数据量非常大，为整个列表创建并储存所有视图对象不仅没有必要，而且会导致严重的系统性能及内存占用问题。

因此，比较聪明的做法是在需要显示的时候才创建视图对象。即当ListView需要显示某个列表项时，它才会去申请一个可用的视图对象。

ListView找谁去申请视图对象呢？ 答案是adapter。adapter是一个控制器对象，从模型层获取数据，并将之提供给ListView显示，起到了沟通桥梁的作用。

adapter负责：

创建必要的视图对象；

用模型层数据填充视图对象；

将准备好的视图对象返回给ListView。

adapter是实现Adapter接口的类实例。我们接下来要使用的adapter是ArrayAdapter<T>类的实例。ArrayAdapter<T>类知道如何处理数组（或ArrayList）中的T类型对象。

图9-7展示了ArrayAdapter<T>类的继承图谱。继承图谱的每一个节点都提供了该层级类或接口的专业化形式。

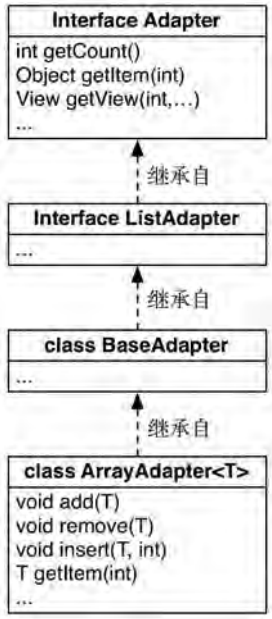

图9-7　ArrayAdapter<T>、BaseAdapter、ListAdapter、Adpter形成了自上而下的继承树

ListView需要显示视图对象时，会与其adapter展开会话沟通。图9-8展示了ListView向其数组adapter启动会话的例子。

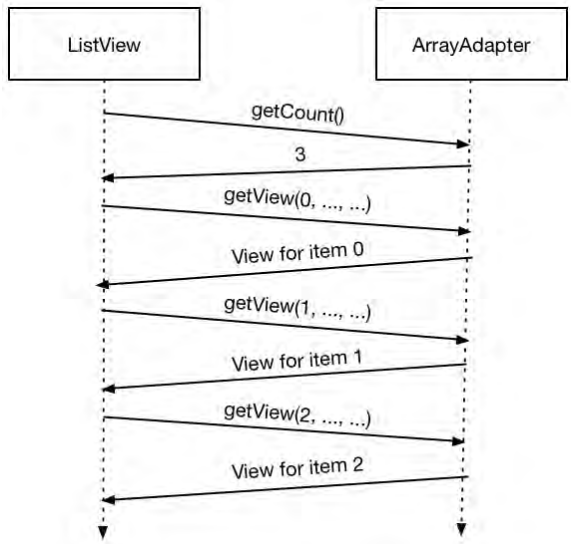

图9-8　生动有趣的ListView-Adapter会话

首先，通过调用adapter的getCount()方法，ListView询问数组列表中包含多少个对象。（为避免出现数组越界的错误，获取对象数目信息非常重要。）

紧接着，ListView就调用adapter的getView(int, View, ViewGroup)方法。该方法的第一个参数是ListView要查找的列表项在数组列表中的位置。

在getView(...)方法的内部实现里，adapter使用数组列表中指定位置的列表项创建一个视图对象，并将该对象返回给ListView。ListView继而将其设置为自己的子视图，并刷新显示在屏幕上。

稍后，通过覆盖getView(...)方法创建定制列表项的学习，我们将会了解到更多有关它的实现机制。

9.4.1 创建ArrayAdapter<T>类实例

首先，使用以下构造方法为CrimeListFragment创建一个默认的ArrayAdapter<T>类实现：

```
public ArrayAdapter(Context context, int textViewResourceId, T[] objects)
```

数组adapter构造方法的第一个参数是一个Context对象，使用第二个参数的资源ID需要该Context对象。资源ID可定位ArrayAdapter用来创建View对象的布局。第三个参数是数据集（Crime数组对象）。

在CrimeListFragment.java中，创建一个ArrayAdapter<T>实例，并设置其为CrimeListFragment中ListView的adapter，如代码清单9-14所示。

代码清单9-14 创建ArrayAdapter（CrimeListFragment.java）

```
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    getActivity().setTitle(R.string.crimes_title);
    mCrimes = CrimeLab.get(getActivity()).getCrimes();

    ArrayAdapter<Crime> adapter =
       new ArrayAdapter<Crime>(getActivity(),
                               android.R.layout.simple_list_item_1,
                               mCrimes);
    setListAdapter(adapter);
}
```

setListAdapter(ListAdapter)是一个ListFragment类的便利方法，使用它可为CrimeListFragment管理的内置ListView设置adapter。

我们在adapter的构造方法中指定的布局（android.R.layout.simple_list_item_1）是Android SDK提供的预定义布局资源。该布局拥有一个TextView根元素。布局的源码如代码清单9-15所示。

代码清单9-15 android.R.layout.simple_list_item_1资源的源码

```
<TextView xmlns:android="http://schemas.android.com/apk/res/android"
  android:id="@android:id/text1"
  style="?android:attr/listItemFirstLineStyle"
  android:paddingTop="2dip"
```

```
    android:paddingBottom="3dip"
    android:layout_width="match_parent"
    android:layout_height="wrap_content" />
```

也可在adapter的构造方法中指定其他布局，只要满足布局的根元素是TextView条件即可。

得益于ListFragment的默认行为，我们现在可以运行应用了。ListView随即会被实例化，并显示在屏幕上，然后开启与adapter间的会话。

运行CriminalIntent应用。这次屏幕上出现的是列表项，而不再是圆形进度条了。不过，视图上显示的内容对用户来说就不那么友好了，如图9-9所示。

图9-9　混合了类名和内存地址的列表项

默认的ArrayAdapter<T>.getView(...)实现方法依赖于toString()方法。它首先生成布局视图，然后找到指定位置的Crime对象并对其调用toString()方法，最后得到字符串信息并传递给TextView。

Crime类当前并没有覆盖toString()方法，因此，它默认使用了java.lang.Object类的实现方法，直接返回了混和对象类名和内存地址的字符串信息。

为让adapter针对Crime对象创建更实用的视图，可打开Crime.java文件，覆盖toString()方法返回crime标题，如代码清单9-16所示。

代码清单9-16　覆盖Crime.toString()方法（Crime.java）

```
...

public Crime() {
```

```
        mId = UUID.randomUUID();
        mDate = new Date();
    }

    @Override
    public String toString() {
        return mTitle;
    }

    ...
```

再次运行CriminalIntent应用。上下滚动列表，查看更多的Crime对象，如图9-10所示。

图9-10 显示crime标题的简单列表项

在我们上下滚动列表时，ListView调用adapter的getView(...)方法，按需获得要显示的视图。

9.4.2 响应列表项的点击事件

要响应用户对列表项的点击事件，可覆盖ListFragment类的另一便利方法：

```
public void onListItemClick(ListView l, View v, int position, long id)
```

无论用户是单击硬按键还是软按键，抑或是手指的触摸，都会触发onListItemClick(...)方法。

在CrimeListFragment.java中，覆盖onListItemClick(...)方法，使adapter返回被点击的列表项所对应的Crime对象，然后日志记录Crime对象的标题，如代码清单9-17所示。

代码清单9-17 覆盖onListItemClick(...)方法，日志记录Crime对象的标题（CrimeListFragment.java）

```
public class CrimeListFragment extends ListFragment {

    private static final String TAG = "CrimeListFragment";

    ...

    @Override
    public void onListItemClick(ListView l, View v, int position, long id) {
        Crime c = (Crime)(getListAdapter()).getItem(position);
        Log.d(TAG, c.getTitle() + " was clicked");
    }
```

getListAdapter()方法是ListFragment类的便利方法，该方法可返回设置在ListFragment列表视图上的adapter。然后，使用onListItemClick(...)方法的position参数调用adapter的getItem(int)方法，最后把结果转换成Crime对象。

再次运行CriminalIntent应用。点击某个列表项，查看日志，确认Crime对象已被正确获取。

9.5　定制列表项

截至目前，每个列表项只是显示了Crime的标题（Crime.toString()方法的返回结果）。

如不满足于此，也可以创建定制列表项。实现显示定制列表项需完成以下任务：

创建定义列表项视图的XML布局文件；

创建ArrayAdapter<T>的子类，用来创建、填充并返回定义在新布局中的视图。

9.5.1　创建列表项布局

在CriminalIntent应用中，每个列表项的视图布局应包含crime的三项内容，即标题、创建日期，以及是否已解决的状态，如图9-11所示。这要求该视图布局包含两个TextView和一个CheckBox。

图9-11　一些定制的列表项

如同创建activity或fragment视图一样，为列表项创建一个新的布局视图。在包浏览器中，右键单击res/layout目录，选择New → Other... → Android XML File。在随后出现的对话框中，资源类型选择Layout，命名布局文件为list_item_crime.xml，设置其根元素为RelativeLayout，最后单击Finish按钮完成。

在RelativeLayout里，子视图相对于根布局以及子视图相对于子视图的布置排列，需使用

一些布局参数加以布置控制。对于列表项新布局，需布置CheckBox对齐RelativeLayout布局的右手边，布置两个TextView相对于CheckBox左对齐。

图9-12展示了定制列表项布局的全部组件。CheckBox子视图须首先被定义，因为虽然它出现在布局的最右边，但TextView需使用CheckBox的资源ID作为属性值。

基于同样的理由，显示标题的TextView也必须定义在显示日期的TextView之前。总而言之，在布局文件里，一个组件必须首先被定义，这样，其他组件才能在定义时使用它的资源ID。定制列表项的布局如图9-12所示。

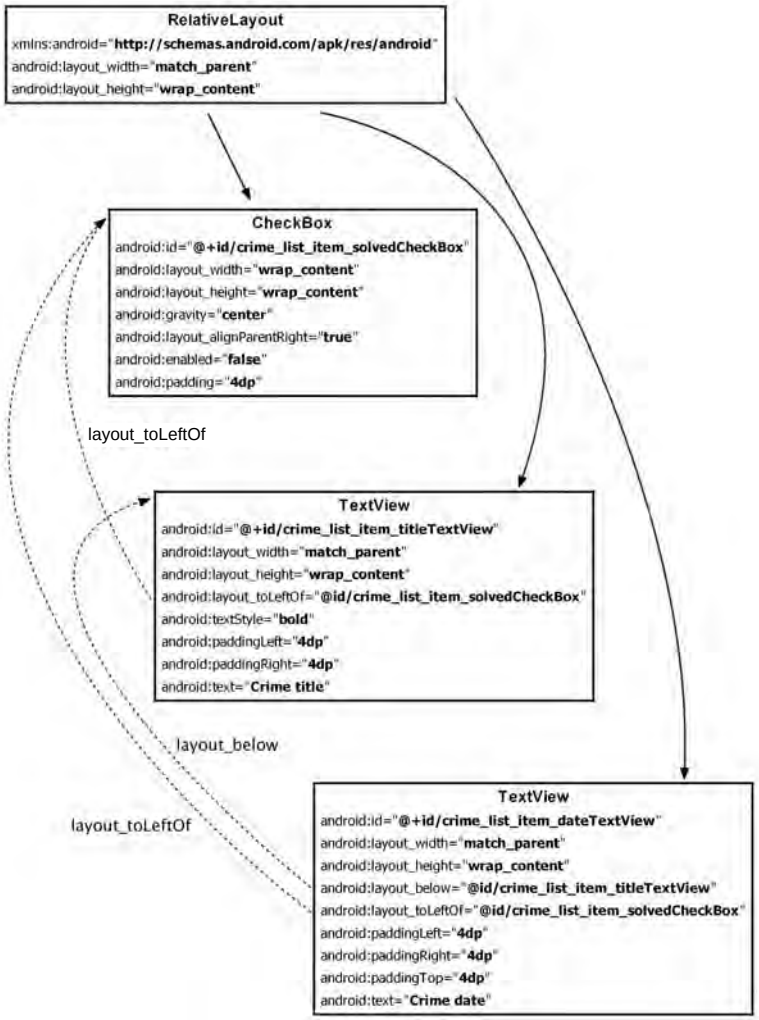

图9-12 定制列表项的布局（list_item_crime.xml）

注意，在其他组件的定义中使用某个组件的ID时，符号+不应该包括在内。符号+是在组件

首次出现在布局文件中时，用来创建资源ID的，一般出现在android:id属性值里。

另外要注意的是，我们在布局定义中使用的是固定字符串，而不是android:text属性的字符串资源。这些字符串是用作开发和测试的示例文字。adapter会提供用户想看到的信息。由于这些字符串不会显示给用户，所以也就没有必要为它们创建字符串资源了。

定制列表项布局创建就完成了。接下来，我们继续学习创建定制adapter。

9.5.2　创建adapter子类

定制布局用来显示特定Crime对象信息的列表项。列表项要显示的数据信息必须使用Crime类的getter方法才能获取，因此，我们需创建一个知道如何与Crime对象交互的adapter。

在CrimeListFragment.java中，创建一个ArrayAdapter的子类作为CrimeListFragment的内部类，如代码清单9-18所示。

代码清单9-18　添加定制的adapter内部类（CrimeListFragment.java）

```
    public void onListItemClick(ListView l, View v, int position, long id) {
        Crime c = (Crime)(getListAdapter()).getItem(position);
        Log.d(TAG, c.getTitle() + " was clicked");
    }

    private class CrimeAdapter extends ArrayAdapter<Crime> {

        public CrimeAdapter(ArrayList<Crime> crimes) {
            super(getActivity(), 0, crimes);
        }

    }

}
```

这里需调用超类的构造方法来绑定Crime对象的数组列表。由于不打算使用预定义布局，我们传入0作为布局ID参数。

创建并返回定制列表项是在以下ArrayAdapter<T>方法里实现的：

```
public View getView(int position, View convertView, ViewGroup parent)
```

convertView参数是一个已存在的列表项，adapter可重新配置并返回它，因此我们无需再创建全新的视图对象。复用视图对象可避免反复创建、销毁同一类对象的开销，应用性能因此得到了提高。ListView一次性需显示大量列表项，因此，没有理由产生大量暂不使用的视图对象来耗尽宝贵的内存资源。

在CrimeAdapter类中，覆盖getView(...)方法返回产生于定制布局的视图对象，并填充对应的Crime数据，如代码清单9-19所示。

代码清单9-19　覆盖getView(...)方法（CrimeListFragment.java）

```
private class CrimeAdapter extends ArrayAdapter<Crime> {

    public CrimeAdapter(ArrayList<Crime> crimes) {
        super(getActivity(), 0, crimes);
```

```
    }

    @Override
    public View getView(int position, View convertView, ViewGroup parent) {
        // If we weren't given a view, inflate one
        if (convertView == null) {
            convertView = getActivity().getLayoutInflater()
                .inflate(R.layout.list_item_crime, null);
        }

        // Configure the view for this Crime
        Crime c = getItem(position);

        TextView titleTextView =
            (TextView)convertView.findViewById(R.id.crime_list_item_titleTextView);
        titleTextView.setText(c.getTitle());
        TextView dateTextView =
            (TextView)convertView.findViewById(R.id.crime_list_item_dateTextView);
        dateTextView.setText(c.getDate().toString());
        CheckBox solvedCheckBox =
            (CheckBox)convertView.findViewById(R.id.crime_list_item_solvedCheckBox);
        solvedCheckBox.setChecked(c.isSolved());

        return convertView;
    }
}
```

在getView(...)实现方法里，首先检查传入的视图对象是否是复用对象。如不是，则从定制布局里产生一个新的视图对象。

无论是新对象还是复用对象，都应调用Adapter的getItem(int)方法获取列表中当前position的Crime对象。

获取Crime对象后，引用视图对象中的各个组件，并以Crime的数据信息对应配置视图对象。最后，把视图对象返回给ListView。

现在可以在CrimeListFragment中绑定定制adapter了。在CrimeListFragment.java文件头部，参照代码清单9-20，更新onCreate(...)和onListItemClick(...)实现方法以使用CrimeAdapter。

代码清单9-20 使用CrimeAdapter（CrimeListFragment.java）

```
        ~~ArrayAdapter<Crime> adapter = new ArrayAdapter<Crime>(this,~~
        ~~        android.R.layout.simple_list_item_1,~~
        ~~        mCrimes);~~
        CrimeAdapter adapter = new CrimeAdapter(mCrimes);
        setListAdapter(adapter);
    }

    public void onListItemClick(ListView l, View v, int position, long id) {
        ~~Crime c = (Crime)(getListAdapter()).getItem(position);~~
        Crime c = ((CrimeAdapter)getListAdapter()).getItem(position);
        Log.d(TAG, c.getTitle() + " was clicked");
    }
```

既然已转换为CrimeAdapter类，自然也获得了类型检查的能力。CrimeAdapter只能容纳Crime对象，因此Crime类的强制类型转换也就不需要了。

通常情况下，现在就可以准备运行应用了。但由于列表项中存在着一个CheckBox，因此还

需进行一处调整。CheckBox默认是可聚焦的。这意味着，点击列表项会被解读为切换CheckBox的状态，自然也就无法触发onListItemClick(...)方法了。

由于ListView的这种内部特点，出现在列表项布局内的任何可聚焦组件（如CheckBox或Button）都应设置为非聚焦状态，从而保证用户在点击列表项后能够获得预期效果。

当前CheckBox没有绑定应用逻辑，只是用来显示Crime信息的，因此，解决方法很简单。只需更新list_item_crime.xml布局文件，将CheckBox定义为非聚焦状态组件即可，如代码清单9-21所示。

代码清单9-21　配置CheckBox为非聚焦状态（list_item_crime.xml）

```
...

<CheckBox android:id="@+id/crime_list_item_solvedCheckBox"
  android:layout_width="wrap_content"
  android:layout_height="match_parent"
  android:gravity="center"
  android:layout_alignParentRight="true"
  android:enabled="false"
  android:focusable="false"
  android:padding="4dp" />

...
```

运行CriminalIntent应用。滚动查看定制列表项，如图9-13所示。点击某个列表项并查看日志，确认CrimeAdapter返回了正确的crime信息。如应用可运行但布局显示不正确，请返回list_item_crime.xml布局文件，检查是否存在输入或拼写等错误。

图9-13　具有定制列表项的用户界面！