第 22 章

22 Master-Detail用户界面

本章将为CriminalIntent应用打造适应平板设备的用户界面，让用户能同时查看到列表和明细界面并与它们进行交互。图22-1展示了这样的列表明细界面。通常我们也称为*主从用户界面*（master-detail interface）。

图22-1　同时显示列表和明细的用户界面

本章的测试需要一台平板设备或AVD。要创建平板AVD，首先选择Window→Android Virtual Device Manager菜单项，然后单击NEW按钮，在图22-2所示的界面，选择红框内任一AVD设备选项。最后设置AVD的系统目标版本为API级别第17级。

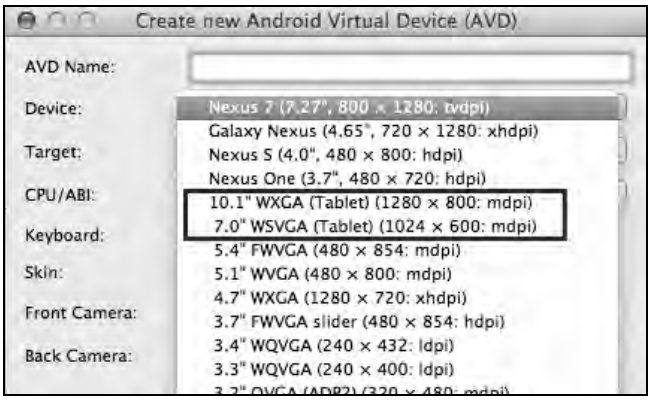

图22-2 AVD平板设备选择

22.1 增加布局灵活性

在手机设备上，CrimeListActivity生成的是单版面（single-pane）布局。而在平板设备上，为同时显示主从视图，我们需要它生成双版面（two-pane）布局。

在双版面布局中，CrimeListActivity将同时托管CrimeListFragment和CrimeFragment（如图22-3所示）。

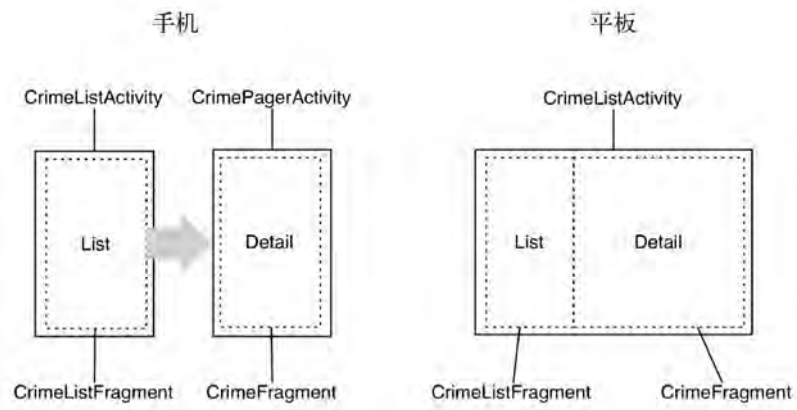

图22-3 不同类型的布局

要实现双版面布局，需执行如下操作步骤：

- 修改SingleFragmentActivity，不再硬编码实例化布局；
- 创建包含两个fragment容器的布局；
- 修改CrimeListActivity，实现在手机设备上实例化单版面布局，而在平板设备上实例化双版面布局。

22.1.1　修改SingleFragmentActivity

CrimeListActivity 是 SingleFragmentActivity 的子类。当前，SingleFragmentActivity只能实例化activity_fragment.xml布局。为使SingleFragmentActivity类更加抽象、灵活，我们让它的子类自己提供布局资源ID。

在SingleFragmentActivity.java中，添加一个protected方法，返回activity需要的布局资源ID，如代码清单22-1所示。

代码清单22-1　增加SingleFragmentActivity类的灵活性（SingleFragmentActivity.java）

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

    protected int getLayoutResId() {
        return R.layout.activity_fragment;
    }

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

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

现在，虽然SingleFragmentActivity抽象类的功能和以前一样，但是它的子类可以选择覆盖getLayoutResId()方法返回所需布局，而不用再使用固定不变的activity_fragment.xml布局。

22.1.2　创建包含两个fragment容器的布局

在包浏览器中，右键单击res/layout/目录，选择创建一个全新的XML文件。在弹出的新建XML文件界面，选择资源类型为Layout，并将文件命名为activity_twopane.xml，然后选择LinearLayout作为根元素，最后单击Finish按钮。

参照图22-4，完成双版面布局的XML内容定义。

注意，布局定义的第一个FrameLayout也有一个fragmentContainer布局资源ID，因此SingleFragmentActivity.onCreate(...)方法的相关代码能够像以前一样工作。activity创建后，createFragment()方法返回的fragment将会出现在屏幕左侧的版面中。

要测试新建布局，在CrimeListActivity类中覆盖getLayoutResId()方法，返回R.layout.activity_twopane资源ID，如代码清单22-2所示。

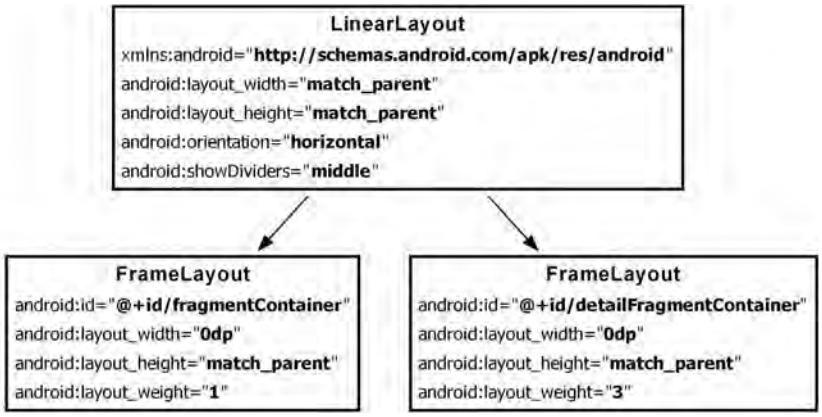

图22-4 包含两个fragment容器的布局（layout/activity_twopane.xml）

代码清单22-2 使用双版面布局（CrimeListActivity.java）

```
public class CrimeListActivity extends SingleFragmentActivity {

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

    @Override
    protected int getLayoutResId() {
        return R.layout.activity_twopane;
    }
}
```

在平板设备或AVD上运行CriminalIntent应用，确认可以看到如图22-5所示的用户界面。注意，右边的明细版面什么也没显示，点击任意列表项，也无法显示对应的陋习明细信息。本章稍后将完成crime明细fragment容器的编码及设置工作。

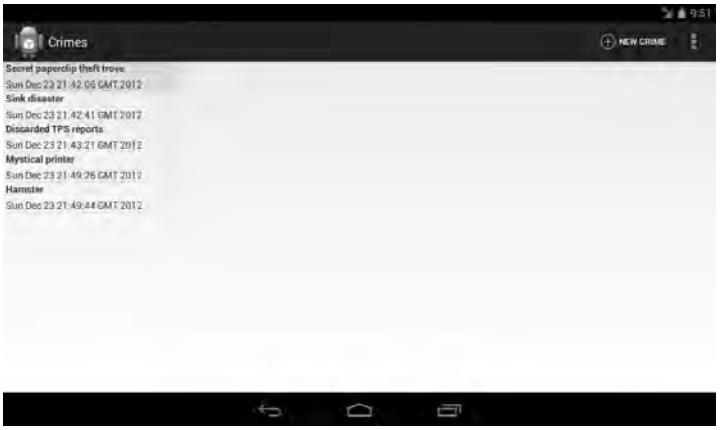

图22-5 平板设备上的双版面布局

当前，无论是在手机还是在平板设备上，CrimeListActivity都会生成双版面的用户界面。下一节将使用别名资源来解决这个问题。

22.1.3　使用别名资源

*别名资源*是一种指向其他资源的特殊资源。它存放在res/values/目录下，并按照约定定义在refs.xml文件中。

本小节将分别创建用于手机指向activity_fragment.xml布局的别名资源，以及用于平板指向activity_twopane.xml布局的别名资源。

在包浏览器中，右键单击res/values/目录，创建一个全新的XML文件。在弹出的新建XML文件界面，选择资源类型为Values，并将文件命名为refs.xml，然后选择resources作为根元素，最后单击Finish按钮。参照代码清单22-3，在新建的refs.xml中添加item节点定义。

代码清单22-3　创建默认的别名资源值（res/values/refs.xml）

```
<resources>

  <item name="activity_masterdetail" type="layout">@layout/activity_fragment</item>

</resources>
```

别名资源指向了单版面布局资源文件。别名资源自身也具有资源ID：R.layout.activity_masterdetail。注意，别名的type属性决定了资源ID属于什么内部类。即使别名资源自身存放在res/values/目录中，它的资源ID依然归属于R.layout内部类。

修改CrimeListActivity类的相应代码，以R.layout.activity_masterdetail资源ID替换R.layout.activity_fragment，如代码清单22-4所示。

代码清单22-4　再次切换布局（CrimeListActivity.java）

```
@Override
protected int getLayoutResId() {
    return R.layout.activity_twopane;
    return R.layout.activity_masterdetail;
}
```

运行CriminalIntent应用，验证别名资源是否可以正常工作。一切正常的话，CrimeListActivity应该再次生成了单版面布局。

创建平板设备专用可选资源

存放在res/values/目录下的别名资源是系统默认的别名资源，所以，CrimeListActivity生成了默认的单版面布局。

现在，创建一个可选别名资源，以实现在平板等大屏幕设备上，activity_masterdetail别名资源可以指向activity_twopane.xml双版面布局资源。

在包浏览器中，右键单击res/目录，新建一个名为values-sw600dp的目录。将res/values/refs.xml文件复制到res/values-sw600dp/新建目录下。然后参照代码清单22-5，修改别名资源指向双版面布局。

代码清单22-5 用于大屏幕设备的可选资源（res/values-sw600dp/refs.xml）

```
<resources>

  <item name="activity_masterdetail" type="layout">@layout/activity_fragment</item>
  <item name="activity_masterdetail" type="layout">@layout/activity_twopane</item>

</resources>
```


配置修饰符-sw600dp是什么意思？SW是smallest width（最小宽度）的缩写，虽然字面上是宽度的含义，但它实际指的是屏幕的最小尺寸（dimension），因而SW与设备的当前方向无关。

在确定可选资源时，-sw600dp配置修饰符表明：对任何最小尺寸为600dp或更高dp的设备，都使用该资源。对于指定平板的屏幕尺寸规格来说，这是一种非常好的做法。

需要说明的是，Android 3.2中才引入了最小宽度配置修饰符。这意味着，运行Android 3.0或Android 3.1系统的平板设备无法识别它。

为解决该问题，可以增加另一种使用-xlarge（仅适用于Android 3.2以前的版本）屏幕尺寸修饰符的可选资源。

右键单击res/目录，新建一个名为values-xlarge的目录。然后将res/values-sw600dp/refs.xml资源文件复制到新建的res/values-xlarge/目录中。现在我们有了另一个如代码清单22-6所示的资源文件。

代码清单22-6 用于Android 3.2之前版本的可选资源（res/values-xlarge/refs.xml）

```
<resources>

  <item name="activity_masterdetail" type="layout">@layout/activity_twopane</item>

</resources>
```

配置修饰符-xlarge包含的资源适用于最低尺寸为720 × 960dp的设备。该修饰符仅适用于运行Android 3.2之前版本的设备。Android 3.2及之后的系统版本会自动找到并使用-sw600dp修饰符目录下的资源。

分别在手机和平板上运行CriminalIntent应用。确认单双版面的布局达到预期效果。

22.2 Activity：fragment 的托管者

既然单双版面的布局显示已处理完成，我们来着手添加CrimeFragment给crime明细fragment容器，让CrimeListActivity可以展示一个完整的双版面用户界面。

我们的第一反应可能会认为，只需再为平板设备实现一个CrimeListFragment.onListItemClick(...)监听器方法就行了。这样，无需启动新的CrimePagerActivity，onListItemClick(...)方法会获取CrimeListActivity的FragmentManager，然后提交一个fragment事务，将CrimeFragment添加到明细fragment容器中。

这种设想的具体实现代码如下：

```
public void onListItemClick(ListView l, View v, int position, long id) {
    // Get the Crime from the adapter
    Crime crime = ((CrimeAdapter)getListAdapter()).getItem(position);
    // Stick a new CrimeFragment in the activity's layout
    Fragment fragment = CrimeFragment.newInstance(crime.getId());
```

```
        FragmentManager fm = getActivity().getSupportFragmentManager();
        fm.beginTransaction()
            .add(R.id.detailFragmentContainer, fragment)
            .commit();
    }
```

以上设想虽然行的通，但做法很老套。fragment天生是一种独立的开发构件。如果要开发一个fragment用来添加其他fragment到activity的FragmentManager，那么这个fragment就必须知道托管activity是如何工作的，这样一来，该fragment就再也无法作为独立的开发构件来使用了。

以上述代码为例，CrimeListFragment将CrimeFragment添加给了CrimeListActivity，并且它知道CrimeListActivity的布局里包含有一个detailFragmentContainer。但实际上，CrimeListFragment根本就不应关心这些，这都是它的托管activity应该处理的事情。

为保持fragment的独立性，我们可以在fragment中定义回调接口，委托托管activity来完成那些不应由fragment处理的任务。托管activity将实现回调接口，履行托管fragment的任务。

fragment回调接口

为委托工作任务给托管activity，通常的做法是由fragment定义名为Callbacks的回调接口。回调接口定义了fragment委托给托管activity处理的工作任务。任何打算托管目标fragment的activity必须实现这些定义的接口。

有了回调接口，无需知道自己的托管者是谁，fragment可以直接调用托管activity的方法。

实现CrimeListFragment.Callbacks回调接口

要实现一个Callbacks接口，首先定义一个成员变量存放实现Callbacks接口的对象。然后将托管activity强制类型转换为Callbacks对象并赋值给Callbacks类型变量。

强制类型转换activity并赋值给Callbacks类型变量是在Fragment的生命周期方法中处理的：

```
public void onAttach(Activity activity)
```

该方法是在fragment附加给activity时调用的，当然fragment是否保留并不重要。

类似地，在相应的生命周期销毁方法中，我们也应将Callbacks变量设置为null。

```
public void onDetach()
```

这里将变量清空的原因是，随后再也无法访问该activity或指望该activity继续存在了。

在CrimeListFragment.java中，添加一个Callbacks接口。另外再添加一个mCallbacks变量并覆盖onAttach(Activity)和onDetach()方法，完成变量的赋值与清空，如代码清单22-7所示。

代码清单22-7　添加回调接口（CrimeListFragment.java）

```
public class CrimeListFragment extends ListFragment {
    private ArrayList<Crime> mCrimes;
    private boolean mSubtitleVisible;
    private Callbacks mCallbacks;

    /**
```

```
     * Required interface for hosting activities.
     */
    public interface Callbacks {

        void onCrimeSelected(Crime crime);
    }

    @Override
    public void onAttach(Activity activity) {
        super.onAttach(activity);
        mCallbacks = (Callbacks)activity;
    }

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

现在，CrimeListFragment有了调用托管activity方法的途径。另外，它也不关心托管activity是谁。只要托管activity实现了CrimeListFragment.Callbacks接口，而CrimeListFragment中一切代码行为都保持不变。

注意，未经类安全性检查，CrimeListFragment就将托管activity强制转换为CrimeListFragment.Callbacks对象。这意味着，托管activity必须实现CrimeListFragment.Callbacks接口。这并非是不良的依赖关系，但记录下它非常重要。

接下来，在CrimeListActivity类中，实现CrimeListFragment.Callbacks接口，如代码清单22-8所示。暂时不用理会onCrimeSelected(Crime)空方法，稍后，我们再来处理。

代码清单22-8 实现回调接口（CrimeListActivity.java）

```
public class CrimeListActivity extends SingleFragmentActivity
    implements CrimeListFragment.Callbacks {

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

    @Override
    protected int getLayoutResId() {
        return R.layout.activity_masterdetail;
    }

    public void onCrimeSelected(Crime crime) {
    }
}
```

最终，在onListItemClick(...)方法里以及在用户创建新crime时，CrimeListFragment将调用onCrimeSelected(Crime)方法。现在，我们先来思考如何实现CrimeListActivity.onCrimeSelected(Crime)方法。

onCrimeSelected(Crime)方法被调用时，CrimeListActivity需要完成以下二选一的任务：

- 如果使用手机用户界面布局，启动新的CrimePagerActivity；

❑ 如果使用平板设备用户界面布局，将CrimeFragment放入detailFragmentContainer中。

为确定需实例化手机还是平板界面布局，可以检查布局ID。但最好最准确的检查方式是检查布局是否包含detailFragmentContainer。因为，布局文件名随时可能更改，并且我们也不关心布局是从哪个文件实例化产生。我们只需知道，布局文件是否包含可以放入CrimeFragment的detailFragmentContainer。

如果证实布局包含detailFragmentContainer，那么我们就会创建一个fragment事务，将我们需要的CrimeFragment添加到detailFragmentContainer中。如果之前就有CrimeFragment存在，首先应从detailFragmentContainer中移除它。

在CrimeListActivity.java中，实现onCrimeSelected(Crime)方法，按照布局界面的不同，响应crime的选择，如代码清单22-9所示。

代码清单22-9　有条件的CrimeFragment启动（CrimeListActivity.java）

```
public void onCrimeSelected(Crime crime) {
    if (findViewById(R.id.detailFragmentContainer) == null) {
        // Start an instance of CrimePagerActivity
        Intent i = new Intent(this, CrimePagerActivity.class);
        i.putExtra(CrimeFragment.EXTRA_CRIME_ID, crime.getId());
        startActivity(i);
    } else {
        FragmentManager fm = getSupportFragmentManager();
        FragmentTransaction ft = fm.beginTransaction();

        Fragment oldDetail = fm.findFragmentById(R.id.detailFragmentContainer);
        Fragment newDetail = CrimeFragment.newInstance(crime.getId());

        if (oldDetail != null) {
            ft.remove(oldDetail);
        }

        ft.add(R.id.detailFragmentContainer, newDetail);
        ft.commit();
    }
}
```

最后，在CrimeListFragment类中，在启动新的CrimePagerActivity的地方，调用onCrimeSelected(Crime)方法。

在CrimeListFragment.java中，修改onListItemClick(...)和onOptionsItemSelected(MenuItem)方法实现对Callbacks.onCrimeSelected(Crime)方法的调用，如代码清单22-10所示。

代码清单22-10　调用全部回调方法（CrimeListFragment.java）

```
public void onListItemClick(ListView l, View v, int position, long id) {
    // Get the Crime from the adapter
    Crime c = ((CrimeAdapter)getListAdapter()).getItem(position);
    // Start an instance of CrimePagerActivity
    Intent i = new Intent(getActivity(), CrimePagerActivity.class);
    i.putExtra(CrimeFragment.EXTRA_CRIME_ID, c.getId());
    startActivity(i);
    mCallbacks.onCrimeSelected(c);
}
```

```
...

@TargetApi(11)
@Override
public boolean onOptionsItemSelected(MenuItem item) {
    switch (item.getItemId()) {
        case R.id.menu_item_new_crime:
            Crime crime = new Crime();
            CrimeLab.get(getActivity()).addCrime(crime);
            Intent i = new Intent(getActivity(), CrimePagerActivity.class);
            i.putExtra(CrimeFragment.EXTRA_CRIME_ID, crime.getId());
            startActivity(i);
            ((CrimeAdapter)getListAdapter()).notifyDataSetChanged();
            mCallbacks.onCrimeSelected(crime);
            return true;
        ...
    }
}
```

在onOptionsItemSelected(...)方法中调用回调方法时，只要新增一项crime记录，就会立即重新加载crime列表。这很有必要，因为在平板设备上，新增crime记录后，crime列表依然可见。而在手机设备上，crime明细界面会在列表界面之前出现，列表项的刷新可以很灵活地处理。

在平板设备上运行CriminalIntent应用。新添加一项crime记录，可以看到，一个CrimeFragment视图立即被添加并显示在detailFragmentContainer容器中。然后，尝试查看其他旧记录以观察CrimeFragment视图的切换，如图22-6所示。

图22-6 已关联的主界面和明细界面

然而，如果修改crime明细内容，列表项并不会以最新数据刷新显示。当前，在CrimeList-Fragment.onResume()方法中，只有新添加一项crime记录，我们才能立即重新刷新显示列表项界面。但是，在平板设备上，CrimeListFragment和CrimeFragment将会同时可见。因此，当CrimeFragment出现时，CrimeListFragment不会暂停，自然也就永远不会从暂停状态恢复了。这就是crime列表项不能重新加载刷新的根本原因。

下面，我们将在CrimeFragment中添加另一个回调接口来修正该问题。

CrimeFragment.Callbacks回调接口的实现

CrimeFragment类中定义的接口如下：

```
public interface Callbacks {
    void onCrimeUpdated(Crime crime);
}
```

保存crime记录的修改时，CrimeFragment类都将调用托管activity的onCrimeUpdated(Crime)方法。CrimeListActivity类将会实现onCrimeUpdated(Crime)方法，从而重新加载CrimeListFragment的列表。

实现CrimeFragment的接口之前，首先在CrimeListFragment类中新增一个方法，用来重新加载刷新CrimeListFragment列表，如代码清单22-11所示。

代码清单22-11　新增updateUI()方法（CrimeListFragment.java）

```
public class CrimeListFragment extends ListFragment {

...

public void updateUI() {
    ((CrimeAdapter)getListAdapter()).notifyDataSetChanged();
}
```

然后，在CrimeFragment.java中，添加回调方法接口以及mCallbacks成员变量并实现onAttach(...)和onDetach()方法，如代码清单22-12所示。

代码清单22-12　新增CrimeFragment回调接口（CrimeFragment.java）

```
...
private ImageView mPhotoView;
private Button mSuspectButton;
private Callbacks mCallbacks;

/**
 * Required interface for hosting activities.
 */
public interface Callbacks {
    void onCrimeUpdated(Crime crime);
}

@Override
public void onAttach(Activity activity) {
    super.onAttach(activity);
    mCallbacks = (Callbacks)activity;
}

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

public static CrimeFragment newInstance(UUID crimeId) {
    ...
}
```

然后在CrimeListActivity类中实现CrimeFragment.Callbacks接口，在onCrimeUpdated(Crime)方法中重新加载crime列表项，如代码清单22-13。

代码清单22-13 刷新显示crime列表（CrimeListActivity.java）

```
public class CrimeListActivity extends SingleFragmentActivity
    implements CrimeListFragment.Callbacks, CrimeFragment.Callbacks {
    ...
    public void onCrimeUpdated(Crime crime) {
        FragmentManager fm = getSupportFragmentManager();
        CrimeListFragment listFragment = (CrimeListFragment)
                fm.findFragmentById(R.id.fragmentContainer);
        listFragment.updateUI();
    }
}
```

在CrimeFragment.java中，如果Crime对象的标题或问题处理状态发生改变，触发调用onCrimeUpdated(Crime)方法，如代码清单22-14所示。

代码清单22-14 调用onCrimeUpdated(Crime)方法（CrimeFragment.java）

```
@Override
@TargetApi(11)
public View onCreateView(LayoutInflater inflater, ViewGroup parent,
        Bundle savedInstanceState) {
    View v = inflater.inflate(R.layout.fragment_crime, parent, false);

    ...

    mTitleField = (EditText)v.findViewById(R.id.crime_title);
    mTitleField.setText(mCrime.getTitle());
    mTitleField.addTextChangedListener(new TextWatcher() {
        public void onTextChanged(CharSequence c, int start, int before, int count) {
            mCrime.setTitle(c.toString());
            mCallbacks.onCrimeUpdated(mCrime);
            getActivity().setTitle(mCrime.getTitle());
        }

        ...
    });

    mSolvedCheckBox = (CheckBox)v.findViewById(R.id.crime_solved);
    mSolvedCheckBox.setChecked(mCrime.isSolved());
    mSolvedCheckBox.setOnCheckedChangeListener(new OnCheckedChangeListener() {
        public void onCheckedChanged(CompoundButton buttonView, boolean isChecked) {
            // Set the crime's solved property
            mCrime.setSolved(isChecked);
            mCallbacks.onCrimeUpdated(mCrime);
        }
    });

    ...

    return v;
}
```

在onActivityResult(...)方法中，Crime对象的记录日期、现场照片以及嫌疑人都有可能

修改，因此，还需在该方法中调用onCrimeUpdated(Crime)方法。当前，crime现场照片以及嫌疑人并没有出现在列表项视图中，但并排的CrimeFragment视图应该显示了这些更新，如代码清单22-15所示。

代码清单22-15　再次调用onCrimeUpdated(Crime)方法（CrimeFragment.java）

```
@Override
public void onActivityResult(int requestCode, int resultCode, Intent data) {
    if (resultCode != Activity.RESULT_OK) return;
    if (requestCode == REQUEST_DATE) {
        Date date = (Date)data.getSerializableExtra(DatePickerFragment.EXTRA_DATE);
        mCrime.setDate(date);
        mCallbacks.onCrimeUpdated(mCrime);
        updateDate();
    } else if (requestCode == REQUEST_PHOTO) {
        // Create a new Photo object and attach it to the crime
        String filename = data
            .getStringExtra(CrimeCameraFragment.EXTRA_PHOTO_FILENAME);
        if (filename != null) {
            Photo p = new Photo(filename);
            mCrime.setPhoto(p);
            mCallbacks.onCrimeUpdated(mCrime);
            showPhoto();
        }
    } else if (requestCode == REQUEST_CONTACT) {
        ...

        c.moveToFirst();
        String suspect = c.getString(0);
        mCrime.setSuspect(suspect);
        mCallbacks.onCrimeUpdated(mCrime);
        mSuspectButton.setText(suspect);
        c.close();
    }
}
```

CrimeListActivity现在有了CrimeFragment.Callbacks接口的一个良好实现。然而，如果在手机设备上运行CriminalIntent应用，它将会崩溃。记住，任何托管CrimeFragment的activity都必须实现CrimeFragment.Callbacks接口。因此，我们还需要在CrimePagerActivity类中实现CrimeFragment.Callbacks接口。

对于CrimePagerActivity类，onCrimeUpdated(Crime)方法什么都不用做，因此直接实现一个空方法即可（如代码清单22-16所示）。CrimePagerActivity类托管CrimeFragment时，必需的列表加载刷新已经在OnResume()方法中完成了。

代码清单22-16　CrimeFragment.Callbacks接口的空实现（CrimePagerActivity.java）

```
public class CrimePagerActivity extends FragmentActivity
    implements CrimeFragment.Callbacks {
    ...

    public void onCrimeUpdated(Crime crime) {
    }
}
```

在平板设备上运行CriminalIntent应用。确认CrimeFragment视图中发生的任何修改，ListView视图都能够更新显示，如图22-7所示。

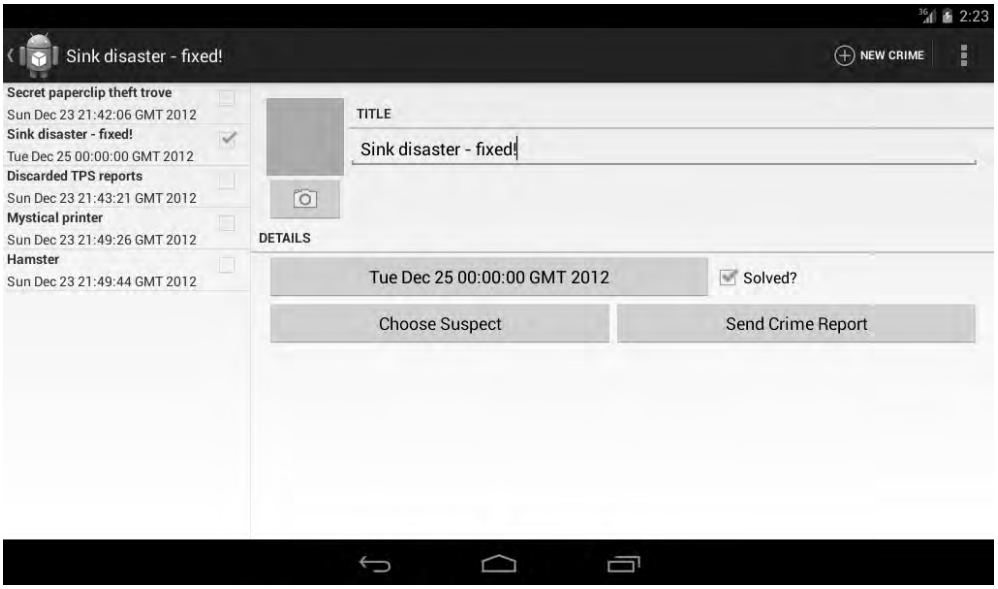

图22-7 列表刷新显示了明细界面的修改

至此，CriminalIntent应用的开发全部完成了。在这13章里，我们创建了一个使用fragment、支持应用间通信、可以拍照以及保存数据的复杂应用。吃块蛋糕庆祝一下吧，不过吃完记得清理现场，不良习惯不要有，人人都是监督者。

22.3 深入学习：设备屏幕尺寸的确定

Android 3.2之前，屏幕大小修饰符是基于设备的屏幕大小来提供可选资源的。屏幕大小修饰符将不同的设备分成了四大类别：small、normal、large及xlarge。

表22-1展示了每个类别修饰符的最低屏幕大小。

表22-1 屏幕大小修饰符

名 称	最低屏幕尺寸	名 称	最低屏幕大小
small	320x426dp	large	480x640dp
Normal	320x470dp	xlarge	720x960dp

顺应允许开发者测试设备尺寸的新修饰符的推出，屏幕大小修饰符已在Android 3.2中弃用。表22-2列出了新引入的修饰符。

表22-2　独立的屏幕尺寸修饰符

修饰符格式	描　述
wXXXdp	有效宽度：宽度大于或等于XXX dp
hXXXdp	有效高度：高度大于或等于XXX dp
swXXXdp	最小宽度：宽度或高度（两者中最小的那个）大于或等于 XXXdp

假设我们想指定某个布局仅适用于屏幕宽度至少300dp的设备。这种情况下，可以使用宽度修饰符，并将布局文件放入res/layout-w300dp目录下（w代表屏幕宽度）。类似地，我们也可以使用“hXXXdp”修饰符（h代表屏幕高度）。

依据设备的方向变换，设备的宽和高也可能会交换。为确定某个具体的屏幕尺寸，我们可以使用sw（最小宽度）。sw指定了屏幕的最小规格尺寸。基于设备的不同方向，sw可能是最小宽度，也可能是最小高度。例如，如果屏幕尺寸为1024×800，那么sw值就是800，而如果屏幕尺寸为800×1024，那么sw值仍然是800。