

第 4 章 骚扰拦截

第 4 章 骚扰拦截.....	3
4.1 模块概述.....	3
4.1.1 功能讲解.....	3
4.1.2 开发流程图.....	4
4.2 黑名单数据库.....	5
4.2.1 创建数据库.....	5
4.2.2 联系人的实体类.....	6
4.2.3 数据库操作类.....	6
4.2.4 测试数据.....	10
4.3 骚扰拦截主界面.....	12
4.3.1 主界面 UI.....	12
4.3.2 骚扰拦截界面的简单实现.....	16
4.3.3 黑名单界面 UI.....	20
4.3.4 黑名单界面逻辑代码.....	23
4.4 添加黑名单.....	27
4.4.1 添加黑名单界面.....	27
4.4.2 添加黑名单逻辑.....	29
4.5 删除黑名单.....	32
4.5.1 删除黑名单的 UI.....	32
4.5.2 删除黑名单的逻辑代码.....	34
4.6 分页加载黑名单.....	35
4.6.1 黑名单提示信息.....	35
4.6.2 ListView 滚动状态监听.....	36
4.6.3 黑名单分页加载.....	37
4.6.4 黑名单加载进度条.....	40
4.7 黑名单拦截.....	42

4.7.1 设置中心的骚扰拦截 UI.....	42
4.7.2 设置中心骚扰拦截状态的回显.....	44
4.7.3 拦截短信.....	45
4.7.4 拦截电话.....	48
4.8 短信拦截记录.....	54
4.8.1 防火墙短信数据库.....	54
4.8.2 短信拦截记录 UI.....	55
4.8.3 黑名单服务中短信拦截记录的添加.....	56
4.8.4 短信拦截记录界面的逻辑代码.....	57
4.9 本章小结.....	62

第 4 章 骚扰拦截

◆ 了解骚扰拦截模块功能

◆ 掌握 SQLite 数据库的使用

◆ 掌握如何使用服务拦截电话和短信

在日常生活中，使用手机时经常会被某些电话或短信骚扰，例如推销保险、中奖信息等，为此，我们开发了骚扰拦截模块，该模块可以将骚扰电话或垃圾短信添加到黑名单中，并对其进行拦截。本章将针对骚扰拦截模块进行详细讲解。

4.1 模块概述

4.1.1 功能讲解

骚扰拦截模块的主要功能是进行黑名单拦截，根据设置，对添加到黑名单中的号码进行电话拦截、短信拦截、电话和短信拦截。添加黑名单时可直接在编辑框中输入电话号码，设置其拦截模式，同时也可以直接删除对应的黑名单号码。

1、添加黑名单

当没有添加黑名单时，会展示一个提示信息的图片，此时点击“添加”按钮，会进入添加黑名单的对话框界面，该界面可以直接输入号码，选中电话拦截、短信拦截的 CheckBox 按钮，然后点击“确定”按钮，此时会将该号码添加到黑名单数据库中，并将黑名单信息展示主界面中，如图 4-1 所示。



图 4-1 添加黑名单信息

2、删除黑名单

在黑名单展示的界面中，每行数据后面都会有一个删除图标，点击该图标后会删除对应的黑名单数据，并及时刷新当前黑名单界面，如图 4-2 所示。



图 4-2 删除黑名单

4.1.2 开发流程图

通讯卫士模块的开发流程有些复杂，为了让大家更好的理解该模块的逻辑，接下来绘制一个流程图，具体如图 4-3 所示。

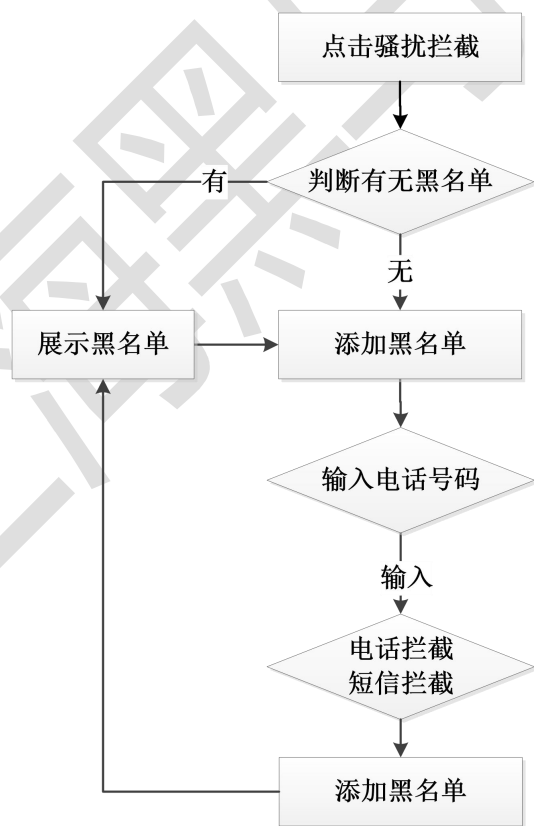


图 4-3 骚扰拦截模块流程图

从图 5-3 可以看出，当进入骚扰拦截界面时，主界面中显示所有的黑名单列表。在添加黑名单时，直接添加号码然后指定电话拦截或者短信拦截后添加黑名单，此时黑名单信息就会展示在主界面中。

4.2 黑名单数据库

黑名单中的信息需要长期保存，为此可以使用 Android 自带的数据库 SQLite 存储黑名单信息，本节将针对数据库的创建以及基本操作进行详细讲解。

4.2.1 创建数据库

要想实现黑名单拦截功能，首先需要根据需求设计一个黑名单数据库（blackNumber.db），该数据库主要用于存储黑名单中的联系人信息，创建包 com.itheima.mobilesafe_sh2.db，在该包下创建黑名单的数据库，代码如【文件 4-1】所示。

【文件 4-1】 BlackNumberOpenHelper.java

```
1. public class BlackNumberOpenHelper extends SQLiteOpenHelper {
2.     public BlackNumberOpenHelper(Context context) {
3.         /**
4.          * 第二个参数：数据库的名字
5.          * 第三个参数：游标工厂
6.          * 第四个参数：版本号
7.          */
8.         super(context, "safe.db", null, 1);
9.         // TODO Auto-generated constructor stub
10.    }
11.    /**
12.     * blacknumber :数据库的表名
13.     * number : 表示黑名单电话号码
14.     * mode   : 表示黑名单的拦截模式
15.     * _id    :主键自动增长
16.     */
17.    @Override
18.    public void onCreate(SQLiteDatabase db) {
19.        db.execSQL("create table blacknumber (_id integer primary key autoincrement,
20. number varchar(20),mode varchar(2))");
21.        db.execSQL("create table firewallsms (_id integer primary key autoincrement,
22. number varchar(20),body varchar(20))");
23.    }
24.    @Override
25.    public void onUpgrade(SQLiteDatabase db, int oldVersion, int newVersion) {
26.        // TODO Auto-generated method stub
27.    }
```

在上述代码中，创建了一个黑名单数据库 `safe.db`，并在数据库中创建 `blacknumber` 表，该表包含三个字段分别为 `id`、`number`、`mode`，其中 `id` 为自增主键，`number` 为电话号码，`mode` 为黑名单的拦截模式，因为防火墙的拦截模式分为短信和电话全部拦截、短信拦截、电话拦截这三种，所以需要区分开。

4.2.2 联系人的实体类

在实现黑名单拦截时，需要保存黑名单中联系人信息，因此需要定义一个黑名单联系人的实体类，该类中包含三个属性，电话号码、联系人姓名、拦截模式，具体代码如【文件 4-2】所示。

【文件 4-2】BlackContactInfo.xml

```
1 public class BlackContactInfo {
2     /**黑名单号码*/
3     public String phoneNumber;
4     /** 黑名单拦截模式 分为
5         1 电话、短信拦截
6         2 电话拦截
7         3 短信拦截
8     **/
9     public int mode;
10 }
```

在上述代码的 `getModeString(int mode)` 方法中，定义了 3 种拦截模式的返回值。由于黑名单拦截模式有 3 种，分别为电话拦截、短信拦截以及电话短信全部拦截，因此在 `getModeString(int mode)` 方法中以数字 1、2、3 分别进行区分，根据传递的数字不同返回不同的拦截模式。

4.2.3 数据库操作类

由于我们经常添加和删除黑名单中的电话号码，因此需要创建一个操作黑名单数据库的工具类，对黑名单中的数据进行增、删、查等操作，该工具类放到 `com.itheima.mobilesafe_sh2.dao` 下，具体代码如【文件 4-3】所示。

【文件 4-3】BlackNumberDao.java

```
1. public class BlackNumberDao {
2.     private BlackNumberSQLiteOpenHelper helper;
3.     public BlackNumberDao(Context context) {
4.         helper = new BlackNumberSQLiteOpenHelper(context);
5.     }
6.     /**
7.      * 添加黑名单到数据库
8.      * @param number 黑名单电话号码
9.      * @param mode 黑名单拦截模式
10.     */
11.     public boolean add(String number, String mode) {
12.         SQLiteDatabase db = helper.getWritableDatabase();
13.         ContentValues values = new ContentValues();
14.         values.put("number", number);
15.         values.put("mode", mode);
```

```
16.         long rowId = db.insert("blacknumber", null, values);
17.         // 判断当前添加是否成功。如果 rowid 等于-1 表示添加失败
18.         if (rowId == -1) {
19.             return false;
20.         } else {
21.             return true;
22.         }
23.     }
24.     /**
25.      * 根据电话号码进行删除
26.      * @param number
27.      * @return
28.      */
29.     public boolean delete(String number) {
30.         SQLiteDatabase db = helper.getWritableDatabase();
31.         int rowNumber = db.delete("blacknumber", "number = ?",
32.             new String[] { number });
33.         // rowNumber 表示影响的行数
34.         if (rowNumber == 0) {
35.             return false;
36.         } else {
37.             return true;
38.         }
39.     }
40.     /**
41.      * 根据电话号码修改拦截的模式
42.      * @param number 电话号码
43.      * @param newMode 新的拦截模式
44.      * @return
45.      */
46.     public boolean changeNumberMode(String number, String newMode) {
47.         SQLiteDatabase db = helper.getWritableDatabase();
48.         ContentValues values = new ContentValues();
49.         values.put("mode", newMode);
50.         int rowNumber = db.update("blacknumber", values, "number = ?",
51.             new String[] { number });
52.         if (rowNumber == 0) {
53.             return false;
54.         } else {
55.             return true;
56.         }
57.     }
58.     /**
```

```
59.      * 根据电话号码查询拦截的模式
60.      * @param number 电话号码
61.      * @return
62.      */
63.  public String findNumberMode(String number) {
64.      String mode = "0";
65.      SQLiteDatabase db = helper.getReadableDatabase();
66.      Cursor cursor = db.query("blacknumber", new String[] { "mode" },
67.          "number = ?", new String[] { number }, null, null, null);
68.      if (cursor.moveToNext()) {
69.          mode = cursor.getString(0);
70.      }
71.      return mode;
72.  }
73.  /**
74.   * 查询所有的黑名单
75.   * @return
76.   */
77.  public List<BlackNumberInfo> findAll() {
78.      SQLiteDatabase db = helper.getReadableDatabase();
79.      Cursor cursor = db
80.          .query("blacknumber", new String[] { "mode", "number" }, null,
81.              null, null, null, null);
82.      List<BlackNumberInfo> mDatas = new ArrayList<BlackNumberInfo>();
83.      while (cursor.moveToNext()) {
84.          BlackNumberInfo info = new BlackNumberInfo();
85.          info.mode = cursor.getString(0);
86.          info.number = cursor.getString(1);
87.          mDatas.add(info);
88.      }
89.      cursor.close();
90.      db.close();
91.      return mDatas;
92.  }
93.
94.  // select * from blacknumber limit 20 offset 40
95.  /**
96.   * 分页加载数据
97.   * @param pageSize 每页有多少条数据
98.   * @param startIndex 从哪条数据开始
99.   *      limit 限制多少条数据 offset 从什么时候开始加载
100.   * @return
101.   */
102.  public List<BlackNumberInfo> findPart(int pageSize, int startIndex) {
```



```
103.         SQLiteDatabase db = helper.getReadableDatabase();
104.         Cursor cursor = db.rawQuery("select mode,number from blacknumber limit ?
105.             offset ?",new String[]{String.valueOf(pageSize),String.valueOf(startIndex)
106.             });
107.         List<BlackNumberInfo> mDataas = new ArrayList<BlackNumberInfo>();
108.         while (cursor.moveToNext()) {
109.             BlackNumberInfo info = new BlackNumberInfo();
110.             info.mode = cursor.getString(0);
111.             info.number = cursor.getString(1);
112.             mDataas.add(info);
113.             SystemClock.sleep(1000);
114.         }
115.         return mDataas;
116.     }
117.
118.     /**
119.      * 返回黑名单数据库里面一共有多少条记录
120.      * @return
121.      */
122.     public int getCount() {
123.         SQLiteDatabase db = helper.getReadableDatabase();
124.         Cursor cursor = db.rawQuery("select count(*) from blacknumber", null);
125.         cursor.moveToNext();
126.         int count = cursor.getInt(0);
127.         return count;
128.     }
129. }
130.
```

- 第 11~23 行的 add()方法用于向数据库中添加数据，首先获取到数据库对象 SQLiteDatabase，然后创建 ContentValues 对象，最后将电话号码、拦截模式存入到数据库中，并返回数据是否插入成功的状态。
- 第 29~39 行的 delete()方法用于删除数据，同样先获取数据库对象 SQLiteDatabase 然后调用 delete()方法，根据电话号码删除数据，并返回删除结果。
- 第 46~57 行的 changeNumberMode()方法根据接收的电话号码修改黑名单数据库中的拦截模式。
- 第 63~72 行的 findNumberMode()方法是根据传递进来的号码获取黑名单拦截模式，判断是电话拦截、短信拦截或者电话和短信都拦截。
- 第 77~92 行的 findAll()方法查询黑名单数据库中的所有数据。
- 第 101~115 行的 findPart()方法用于分页查询数据库中的黑名单数据。
- 第 120~126 行的 getCount()方法用于获取黑名单数据库中存储数据的总记录。

4.2.4 测试数据

在程序开发中，开发者需要对每一个新模块或者方法进行测试，以保证代码可运行没有 BUG，由于数据库工具类中操作黑名单数据的方法比较多，而且这些数据需要填充到主界面中，为了避免后期出现错误导致调试困难，最好在使用这些方法之前进行测试。

Android 系统自带了测试框架 JUnit，接下来使用该框架对数据库工具类中的方法进行测试，首先在清单文件中配置相应信息，具体代码如【文件 4-4】所示。

【文件 4-4】 AndroidManifest.xml

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.itheima.mobilesafe_sh2"
    android:versionCode="1"
    android:versionName="1.0" >
    <uses-sdk
        android:minSdkVersion="16"
        android:targetSdkVersion="17" />

    <instrumentation
        android:name="android.test.InstrumentationTestRunner"
        android:targetPackage="com.itheima.mobilesafe_sh2"/>

    <application
        android:allowBackup="true"
        android:icon="@drawable/ic_launcher"
        android:label="@string/app_name"
        android:theme="@style/AppTheme" >

        <uses-library android:name="android.test.runner" >

        </uses-library>

        .....
    </application>
</manifest>
```

接下来在 com.itheima.mobilesafe_sh2.test 包下创建一个测试类 TestBlackNumberDao，它继承自于 AndroidTestCase，并对数据库中增加、删除、修改等方法进行测试。需要注意的是，在 JUnit 测试框架中，测试方法的异常必须抛出，不能 try-catch，否则测试框架捕获不到异常。测试类的代码如【文件 4-5】所示。

【文件 4-5】 TestBlackNumberDao.java

```
1. public class TestBlackNumberDao extends AndroidTestCase {
2.     private Context context;
3.     @Override
4.     protected void setUp() throws Exception {
5.         context = getContext();
6.         super.setUp();
7.     }
8.     // 测试添加
9.     public void testAdd(){
10.         BlackNumberDao dao = new BlackNumberDao(context);
11.         Random random = new Random();
```

```
12.         for (int i = 0; i < 200; i++) {
13.             long number= 133000000001 + i;
14.             dao.add(number + "", random.nextInt(3) + 1 + "");
15.         }
16.     }
17.     // 测试删除
18.     public void testDelete(){
19.         BlackNumberDao dao = new BlackNumberDao(context);
20.         dao.delete("133000000001");
21.     }
22.     // 测试修改
23.     public void testUpdate(){
24.         BlackNumberDao dao = new BlackNumberDao(context);
25.         dao.changeNumberMode("133000000000", "2");
26.     }
27. }
```

在测试数据时，首先要测试添加方法，向数据库中添加 200 条数据，然后在测试其他方法，如果测试通过则在 JUnit 窗口中显示绿条，如图 4-4 所示。

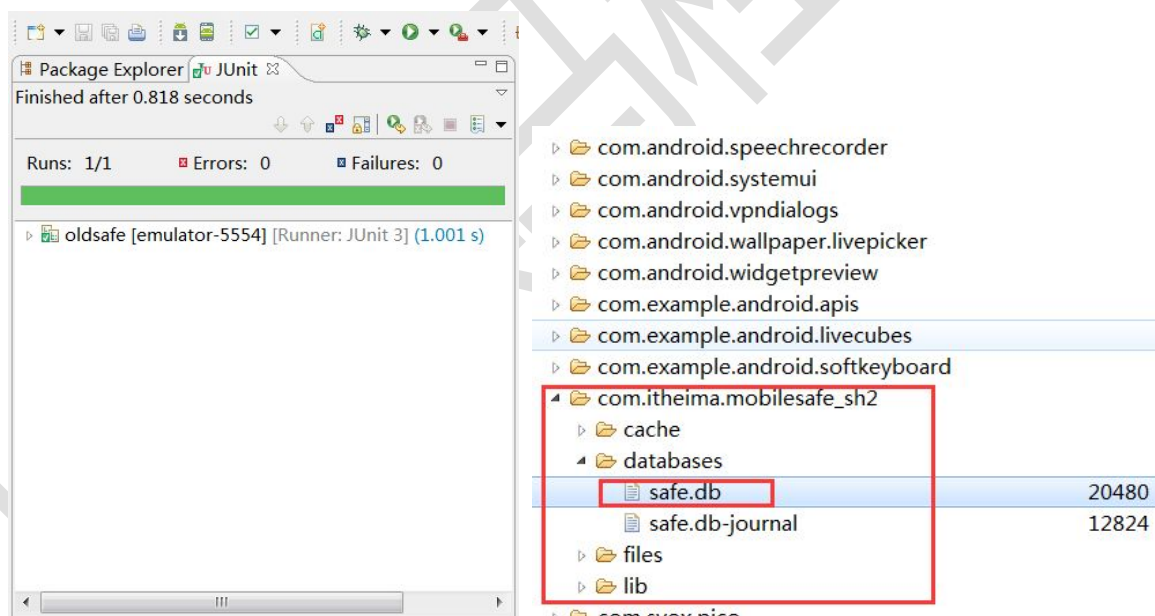
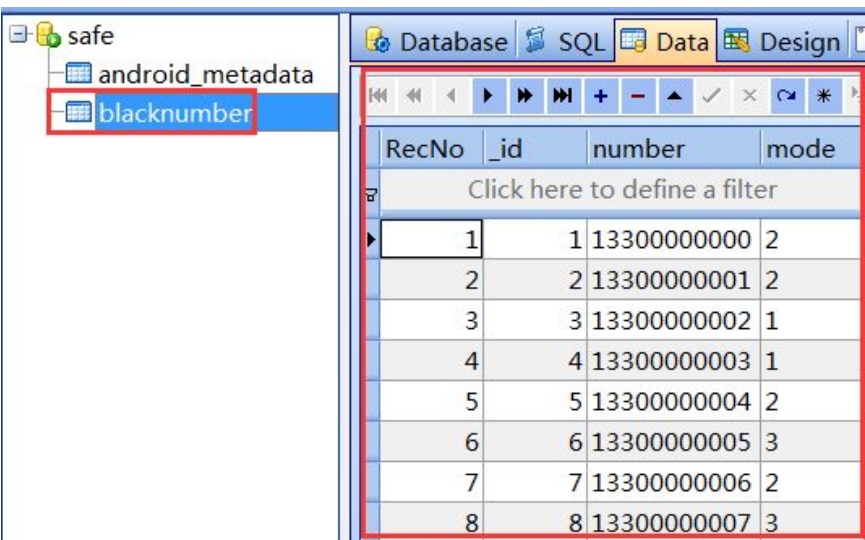


图 4-4 测试成功

值得一提的是，如果在测试这几个方法时，JUnit 窗口的测试结果显示绿条，说明数据库逻辑没问题测试通过，可以正常向数据库中添加、删除或者查看数据，如图 4-5 所示，数据库中添加数据成功。



RecNo	_id	number	mode
1	1	13300000000	2
2	2	13300000001	2
3	3	13300000002	1
4	4	13300000003	1
5	5	13300000004	2
6	6	13300000005	3
7	7	13300000006	2
8	8	13300000007	3

图 4-5 测试的黑名单数据库

4.3 骚扰拦截主界面

骚扰拦截模块的效果图如图 4-6 所示，当在设置中心界面点击开启骚扰拦截后，该功能才会启动。并且它的主界面分为三个部分，即短信拦截记录、电话拦截记录和黑名单管理三个模块，本小节首先对这三部分的界面进行详细分析。

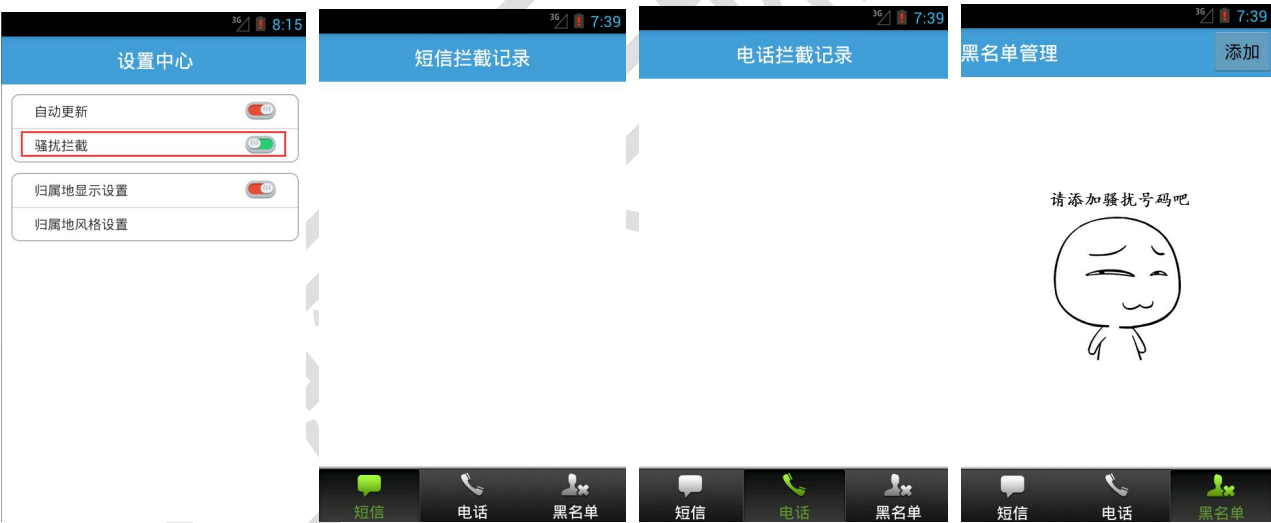


图 4-6 骚扰拦截界面效果

4.3.1 主界面 UI

为实现上面的界面效果，我们需要分析界面的组成部分，经过分析可知，界面可分为两大部分，如图 4-7 所示，上边的内容可以是帧布局，然后帧布局中使用 fragment 进行内容填充，下边的内容可以使用三个 RadioButton 进行功能的切换。

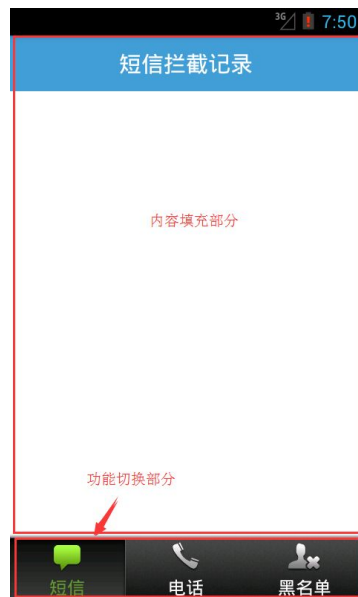


图 4-7 界面分析

根据以上分析，创建黑名单界面 FirewallActivity.java，布局文件如【文件 4-6】所示。

【文件 4-6】 res/layout/activity_firewall.xml

```
1. <?xml version="1.0" encoding="utf-8"?>
2. <LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
3.     android:layout_width="match_parent"
4.     android:layout_height="match_parent"
5.     android:orientation="vertical" >
6.     <TextView
7.         style="@style/TitleBarTextView"
8.         android:text="黑名单管理" />
9.     <!--android:layout_weight="1"表示渲染的先后顺序，如果值越小，越先绘制-->
10.    <FrameLayout
11.        android:layout_width="match_parent"
12.        android:layout_height="0dp"
13.        android:layout_weight="1" >
14.    </FrameLayout>
15.    <RadioGroup
16.        android:id="@+id/rg_firewall"
17.        android:layout_width="match_parent"
18.        android:layout_height="60dp"
19.        android:background="@drawable/traffic_tab_widget_background"
20.        android:gravity="center_vertical"
21.        android:orientation="horizontal"
22.        android:paddingTop="2dp" >
23.        <RadioButton
24.            android:id="@+id/rb_sms"
25.            style="@style/FirewallBottom"
```

```

26.         android:drawableTop="@drawable/firewall_tab_icon_sms_xml"
27.         android:text="短信"
28.         android:textColor="@color/bottom_text_color" />
29.     <RadioButton
30.         android:id="@+id/rb_phone"
31.         style="@style/FirewallBottom"
32.         android:drawableTop="@drawable/firewall_tab_icon_phone_xml"
33.         android:text="电话"
34.         android:textColor="@color/bottom_text_color" />
35.     <RadioButton
36.         android:id="@+id/rb_black"
37.         style="@style/FirewallBottom"
38.         android:drawableTop="@drawable/firewall_tab_icon_black_xml"
39.         android:text="黑名单"
40.         android:textColor="@color/bottom_text_color" />
41. </RadioGroup>
42. </LinearLayout>

```

布局文件中 `RadioButton` 的设置为了让代码更加简便，本项目在 `values` 下的 `styles.xml` 文件中定义了统一的样式 `FirewallBottom`，其内容如下所示。

```

1.  <!-- 骚扰拦截的样式 -->
2.  <style name="FirewallBottom">
3.      <item name="android:layout_width">0dp</item>
4.      <item name="android:layout_height">wrap_content</item>
5.      <item name="android:layout_weight">1</item>
6.      <item name="android:button">@null</item>
7.      <item name="android:gravity">center</item>
8.      <item name="android:padding">2dp</item>
9.      <item name="android:background">@drawable/tab_pressed_xml</item>
10. </style>

```

为了让 `RadioButton` 更加美观，`RadioButton` 的背景我们使用了选择器，即 `tab_pressed.xml.xml`，它的内容如文件【4-7】所示。

【文件 4-7】res/drawable/tab_pressed.xml.xml

```

1. <?xml version="1.0" encoding="utf-8"?>
2. <selector
3.     xmlns:android="http://schemas.android.com/apk/res/android">
4.     <item android:state_pressed="true" android:drawable="@drawable/tab_pressed" />
5.     <item android:state_focused="true" android:drawable="@drawable/tab_pressed" />
6.     <item android:state_selected="true" android:drawable="@drawable/tab_pressed" />
7.     <item android:state_checked="true" android:drawable="@drawable/tab_pressed" />
8.     <item android:drawable="@color/traffic_tou_ming_color" />
9. </selector>

```

另外，这里 `RadioGroup` 的背景是 `traffic_tab_widget_background.xml`，它位于 `drawable` 目录下，内容如文件【4-8】所示。

【文件 4-8】res/drawable/traffic_tab_widget_background.xml

```
1. <?xml version="1.0" encoding="utf-8"?>
2. <bitmap xmlns:android="http://schemas.android.com/apk/res/android"
3.     android:src="@drawable/traffic_tab_focused"
4.     android:tileMode="repeat" />
```

在上述布局文件的代码中，主要包含标题、FrameLayout 布局和 RadioGroup，RadioGroup 的每一个 Button 按钮的背景图片使用了图片选择器，在按钮按下与松开时显示不同颜色的图片。

短信按钮的图片选择器的代码如【文件 4-9】所示。

【文件 4-9】 res/drawable/firewall_tab_icon_sms.xml.xml

```
1. <?xml version="1.0" encoding="utf-8"?>
2. <selector xmlns:android="http://schemas.android.com/apk/res/android">
3.     <item android:state_pressed="true" android:drawable="@drawable/
4.         firewall_tab_icon_sms_focused" />
5.     <item android:state_focused="true" android:drawable="@drawable/
6.         firewall_tab_icon_sms_focused" />
7.     <item android:state_selected="true" android:drawable="@drawable/
8.         firewall_tab_icon_sms_focused" />
9.     <item android:state_checked="true" android:drawable="@drawable/
10.         firewall_tab_icon_sms_focused" />
11.     <item android:drawable="@drawable/firewall_tab_icon_sms" />
12. </selector>
```

电话按钮的图片选择器的代码如【文件 4-10】所示。

【文件 4-10】 res/drawable/firewall_tab_icon_phone.xml.xml

```
1. <?xml version="1.0" encoding="utf-8"?>
2. <selector xmlns:android="http://schemas.android.com/apk/res/android">
3.     <item android:state_pressed="true" android:drawable="@drawable/
4.         firewall_tab_icon_phone_focused" />
5.     <item android:state_focused="true" android:drawable="@drawable/
6.         firewall_tab_icon_phone_focused" />
7.     <item android:state_selected="true" android:drawable="@drawable/
8.         firewall_tab_icon_phone_focused" />
9.     <item android:state_checked="true" android:drawable="@drawable/
10.         firewall_tab_icon_phone_focused" />
11.     <item android:drawable="@drawable/firewall_tab_icon_phone" />
12. </selector>
```

黑名单按钮的图片选择器的代码如【文件 4-11】所示。

【文件 4-11】 res/drawable/firewall_tab_icon_black.xml.xml

```
1. <?xml version="1.0" encoding="utf-8"?>
2. <selector xmlns:android="http://schemas.android.com/apk/res/android">
3.     <item android:state_pressed="true" android:drawable="@drawable/
4.         firewall_tab_icon_black_focused" />
5.     <item android:state_focused="true" android:drawable="@drawable/
6.         firewall_tab_icon_black_focused" />
7.     <item android:state_selected="true" android:drawable="@drawable/
```

```

8.         firewall_tab_icon_black_focused" />
9.         <item android:state_checked="true" android:drawable="@drawable/
10.             firewall_tab_icon_black_focused" />
11.         <item android:drawable="@drawable/firewall_tab_icon_black" />
12.     </selector>

```

另外，三个 `RadioButton` 的标题颜色也是采用了选择器，具体如文件【4-12】所示。

【文件 4-12】 res/color/bottom_text_color.xml

```

1. <?xml version="1.0" encoding="utf-8"?>
2. <selector xmlns:android="http://schemas.android.com/apk/res/android">
3.     <item android:state_pressed="true" android:color="#669933"/>
4.     <item android:state_checked="true" android:color="#669933"/>
5.     <item android:color="#FFF"/> <!-- not selected -->
6. </selector>

```

上述代码书写好之后，运行程序，如图 4-8 所示。尤为重要的一点是，当 `RadioButton` 没有设置 `id` 的时候，运行程序，发现三个按钮可同时被选中，如图 4-8（a）所示。因此，需要给每个 `RadioButton` 设置 `id`，这样才能实现按钮的三个点击选中功能，如图 4-8（b）—（d）所示。



图 4-8 黑名单界面

4.3.2 骚扰拦截界面的简单实现

本项目中，短信、电话、黑名单三个 `RadioButton` 按钮点击切换的界面是使用 `fragment` 实现的，下面我们开始详细介绍这三个 `Fragment` 界面。

首先，创建一个包 `com.itheima.mobilesafe_sh2.fragment`，该包下存放骚扰拦截的三个 `fragment` 界面，`fragment` 内容填充的逻辑在 `FirewallActivity.java` 实现，具体的代码如文件【4-13】所示。

【文件 4-13】 com.itheima.mobilesafe_sh2.act/FirewallActivity.java

```

1. public class FirewallActivity extends FragmentActivity {
2.     // 单选按钮组
3.     private RadioGroup mRadioGroup;
4.     // 内容页面

```



```
5.     private FrameLayout mFrameLayout;
6.     // 索引
7.     private int index = 0;
8.     private FragmentManager manager;
9.     @Override
10.    protected void onCreate(Bundle savedInstanceState) {
11.        super.onCreate(savedInstanceState);
12.        setContentView(R.layout.activity_firewall);
13.        mFrameLayout = (FrameLayout) findViewById(R.id.content);
14.        mRadioGroup = (RadioGroup) findViewById(R.id.rg_firewall);
15.        // 设置状态监听
16.        mRadioGroup.setOnCheckedChangeListener(new OnCheckedChangeListener() {
17.            @Override
18.            public void onCheckedChanged(RadioGroup group, int checkedId) {
19.                switch (checkedId) {
20.                    // 短信
21.                    case R.id.rb_sms:
22.                        index = 0;
23.                        break;
24.                    // 电话
25.                    case R.id.rb_phone:
26.                        index = 1;
27.                        break;
28.                    // 黑名单
29.                    case R.id.rb_black:
30.                        index = 2;
31.                        break;
32.                }
33.                //初始化 fragment
34.                //第一个参数：填充的内容
35.                //第二个参数：表示索引
36.                Fragment fragment = (Fragment) fragments.instantiateItem(mFrameLayout,
37. index);
38.                //设置 fragment 的默认值
39.                fragments.setPrimaryItem(mFrameLayout, 0, fragment);
40.                //提交更新
41.                fragments.finishUpdate(mFrameLayout);
42.            }
43.        });
44.        //设置 RadioButton 默认选中的页面，这里默认短信页面
45.        mRadioGroup.check(R.id.rb_sms);
46.    }
47.    /**
```

```

48.      * fragment 的适配器
49.      */
50.      FragmentStatePagerAdapter fragments = new FragmentStatePagerAdapter(
51.          getSupportFragmentManager()) {
52.          private Fragment fragment;
53.          /**
54.           * 一共有多少个 fragment
55.           */
56.          @Override
57.          public int getCount() {
58.              return 3;
59.          }
60.          @Override
61.          public Fragment getItem(int position) {
62.              switch (position) {
63.                  case 0:
64.                      fragment = new FirewallSmsFragment();
65.                      break;
66.                  case 1:
67.                      fragment = new FirewallPhoneFragment();
68.                      break;
69.                  case 2:
70.                      fragment = new FirewallBlackFragment();
71.                      break;
72.              }
73.              return fragment;
74.          }
75.      };
76.  }

```

- 第 16~32 行代码是 **RadioGroup** 的状态监听，根据用户点击对应 **RadioButton** 的 **id** 而创建不同的 **fragment** 的对象，并设置 **fragments** 适配器对象实例化 **fragment** 的索引。
- 第 33~41 行中 **fragments** 通过 **fragment** 适配器对象的实例化后获得 **fragment** 对象，然后进行初始化和提交更新。
- 第 50~76 行是创建 **fragment** 的适配器，该适配器进行 **fragment** 页面的填充。

这里刚开始，对应 **fragment** 的内容我们简单只填充一个 **TextView** 来观察上述代码的设置，关于这三个 **fragment** 的代码如下面三个文件所示。

【文件 4-14】 com.itheima.mobilesafe_sh2.fragment/FirewallSmsFragment.java

```

1.  public class FirewallSmsFragment extends Fragment {
2.      @Override
3.      public View onCreateView(LayoutInflater inflater, ViewGroup container,
4.          Bundle savedInstanceState) {
5.          TextView textView = new TextView(getActivity());
6.          textView.setText("短信");
7.          return textView;

```

```
8.     }
9. }
```

【文件 4-15】 com.itheima.mobilesafe_sh2.fragment/FirewallPhoneFragment.java

```
10. public class FirewallPhoneFragment extends Fragment {
11.     @Override
12.     public View onCreateView(LayoutInflater inflater, ViewGroup container,
13.         Bundle savedInstanceState) {
14.         TextView textView = new TextView(getActivity());
15.         textView.setText("电话");
16.         return textView;
17.     }
18. }
```

【文件 4-16】 com.itheima.mobilesafe_sh2.fragment/FirewallBlackFragment.java

```
19. public class FirewallBlackFragment extends Fragment {
20.     @Override
21.     public View onCreateView(LayoutInflater inflater, ViewGroup container,
22.         Bundle savedInstanceState) {
23.         TextView textView = new TextView(getActivity());
24.         textView.setText("黑名单");
25.         return textView;
26.     }
27. }
```

此时，运行程序，观察效果，如图 4-9 所示，第一个短信界面如图 4-9（a）是默认的初始界面，其他两个界面电话和黑名单界面如图 4-9（b）和 4-9（c）所示，虽然切换后对应 TextView 的内容显示出来，但是会存在重影。



(a)

(b)

(c)

图 4-9 骚扰拦截三个界面的简单实现

为解决这个重影问题，我们需要重写当前三个 fragment 的 `setMenuVisibility()` 方法，在该方法中添加以下代码。

```
1.  /**
2.   * 去掉重影
3.   */
4.  @Override
5.  public void setMenuVisibility(boolean menuVisible) {
6.      // TODO Auto-generated method stub
7.      super.setMenuVisibility(menuVisible);
8.      if (getView() != null) {
9.          getView().setVisibility(menuVisible ? View.VISIBLE : View.INVISIBLE);
10.     }
11. }
```

上述代码中，当用户点击按钮切换时，系统会把当前的 `menuVisible` 值传到 `setMenuVisibility` 方法中，`getView()` 获取的是 `onCreateView` 返回的 `View` 对象，例如当用户第一次进入骚扰拦截界面时，首先呈现的是短信界面，当用户点击电话按钮切换时，系统会判断短信界面的 `menuVisible` 值为 `false`，此时就会隐藏短信界面，用户继续点击黑名单按钮时，系统判定电话界面的 `menuVisible` 值为 `false`，此时就会隐藏电话界面的内容。

加上上面的这段代码之后，运行程序，如图 4-10 所示，此时已经不会出现重影。



图 4-10 骚扰拦截界面的简单实现（去重影版）

4.3.3 黑名单界面 UI

首先，我们实现黑名单界面的数据展示和功能实现，该界面的效果图如图 4-11 所示。



图 4-11 黑名单界面效果图

观察上述效果图可知，我们需要自己定义一个对应的布局文件 `fragment_firewall_black.xml`，这里使用一个 `ListView` 用于显示所有黑名单数据库中的数据，如文件【4-17】所示。

【文件 4-17】 `res/layout/fragment_firewall_black.xml`

```
1. <?xml version="1.0" encoding="utf-8"?>
2. <LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
3.     android:layout_width="match_parent"
4.     android:layout_height="match_parent"
5.     android:orientation="vertical" >
6.
7.     <TextView
8.         style="@style/TitleBarTextView"
9.         android:text="黑名单管理" />
10.     <!-- android:layout_weight="1" 表示渲染的先后顺序。如果值越小。越先绘制 -->
11.
12.     <ListView
13.         android:id="@+id/list_view"
14.         android:layout_width="match_parent"
15.         android:layout_height="match_parent" >
16.     </ListView>
17. </LinearLayout>
```

由于主界面的黑名单列表是通过 `ListView` 控件展示的，因此需要定义一个黑名单的 `Item` 布局，保证每个条目的布局都是一致的，观察完整版的界面效果，`Item` 布局的图形化界面如图 4-12 所示。



图 4-12 黑名单 Item 布局

图 4-12 黑名单 Item 对应的布局文件如【文件 4-18】所示。

【文件 4-18】 item_list_blackcontact.xml

```
1. <?xml version="1.0" encoding="utf-8"?>
2. <RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
3.     android:layout_width="match_parent"
4.     android:layout_height="match_parent"
5.     android:orientation="vertical" >
6.     <TextView
7.         android:id="@+id/tv_phone"
8.         android:layout_width="wrap_content"
9.         android:layout_height="wrap_content"
10.        android:layout_marginLeft="10dp"
11.        android:layout_marginTop="10dp"
12.        android:text="电话号码"
13.        android:textColor="#000"
14.        android:textSize="18sp" />
15.     <TextView
16.         android:id="@+id/tv_mode"
17.         android:layout_width="wrap_content"
18.         android:layout_height="wrap_content"
19.         android:layout_below="@id/tv_phone"
20.         android:layout_marginLeft="10dp"
21.         android:layout_marginTop="3dp"
22.         android:text="拦截的模式"
23.         android:textColor="#88000000"
24.         android:textSize="14sp" />
25.     <ImageView
26.         android:id="@+id/iv_delete"
27.         android:layout_width="wrap_content"
28.         android:layout_height="wrap_content"
```

```
29.         android:layout_alignParentRight="true"
30.         android:layout_margin="10dp"
31.         android:background="@android:color/transparent"
32.         android:src="@drawable/ic_delete_btn" />
33. </RelativeLayout>
```

上述布局中，使用了两个 `TextView` 对象以及一个 `ImageView` 控件，其中 `ImageView` 控件用于表示删除图标，`TextView` 控件用于显示联系人号码以及拦截方式。

4.3.4 黑名单界面逻辑代码

布局文件写好之后，接下来就需要在 `FirewallBlackFragment.java` 中进行主要的逻辑代码的书写，为了将数据库中的数据全部加载到当前界面，我们需要查询数据库，并使用数据适配器将数据展示出来，具体的内容如文件【4-19】所示。

【文件 4-19】 FirewallBlackFragment.java

```
1. public class FirewallBlackFragment extends Fragment {
2.     private View view;
3.     private ListView mListView;
4.     private List<BlackNumberInfo> mDatas = new ArrayList<BlackNumberInfo>();
5.     private BlackNumberDao dao;
6.     private BlackAdapter adapter;
7.     /**
8.      * 初始化 view
9.      */
10.    @Override
11.    public View onCreateView(LayoutInflater inflater, ViewGroup container,
12.        Bundle savedInstanceState) {
13.        view = inflater.inflate(R.layout.fragment_firewall_black, null);
14.        mListView = (ListView) view.findViewById(R.id.list_view);
15.        return view;
16.    }
17.    /**
18.     * 初始化数据
19.     */
20.    @Override
21.    public void onActivityCreated(Bundle savedInstanceState) {
22.        // TODO Auto-generated method stub
23.        super.onActivityCreated(savedInstanceState);
24.        initData();
25.    }
26.    /**
27.     * 初始化所有的数据
28.     */
29.    private void initData() {
```

```
30.         dao = new BlackNumberDao(getActivity());
31.         mDatas = dao.findAll();
32.         adapter = new BlackAdapter();
33.         mListView.setAdapter(adapter);
34.     }
35.
36.     // 展示黑名单的数据适配器
37.     private class BlackAdapter extends BaseAdapter {
38.         private ViewHolder holder;
39.         @Override
40.         public int getCount() {
41.             return mDatas.size();
42.         }
43.
44.         @Override
45.         public View getView(final int position, View convertView,
46.                             ViewGroup parent) {
47.             if (convertView == null) {
48.                 convertView = View.inflate(getActivity(),
49.                     R.layout.item_firewall_black_fragment, null);
50.                 holder = new ViewHolder();
51.                 holder.tv_phone = (TextView) convertView
52.                     .findViewById(R.id.tv_phone);
53.                 holder.tv_mode = (TextView) convertView
54.                     .findViewById(R.id.tv_mode);
55.                 holder.iv_delete = (ImageView) convertView
56.                     .findViewById(R.id.iv_delete);
57.                 convertView.setTag(holder);
58.             } else {
59.                 holder = (ViewHolder) convertView.getTag();
60.             }
61.             BlackNumberInfo info = mDatas.get(position);
62.             /**
63.              * 黑名单的拦截模式 1 全部拦截(电话拦截 + 短信拦截) 2 电话拦截 3 短信拦截
64.              */
65.             String mode = info.mode;
66.             // 判断用户的拦截模式
67.             if ("1".equals(mode)) {
68.                 holder.tv_mode.setText("电话拦截 + 短信拦截");
69.             } else if ("2".equals(mode)) {
70.                 holder.tv_mode.setText("电话拦截 ");
71.             } else if ("3".equals(mode)) {
72.                 holder.tv_mode.setText("短信拦截 ");
73.             }
```



```
74.         holder.tv_phone.setText(info.number);
75.         return convertView;
76.     }
77.     @Override
78.     public Object getItem(int position) {
79.         return null;
80.     }
81.     @Override
82.     public long getItemId(int position) {
83.         // TODO Auto-generated method stub
84.         return 0;
85.     }
86. }
87. static class ViewHolder {
88.     TextView tv_phone;
89.     TextView tv_mode;
90.     ImageView iv_delete;
91. }
92. /**
93.  * 去掉重影
94.  */
95. @Override
96. public void setMenuVisibility(boolean menuVisible) {
97.     // TODO Auto-generated method stub
98.     super.setMenuVisibility(menuVisible);
99.     if (getView() != null) {
100.         getView().setVisibility(menuVisible ? View.VISIBLE : View.INVISIBLE);
101.     }
102. }
103. }
104.
```

- 第 11~16 行，加载黑名单界面的布局文件，并获取 ListView 对象
- 第 29~34 行，初始化数据，并给 ListView 设置适配器对象
- 第 36~86 行，黑名单数据的适配器，用于将黑名单数据库中的全部数据展示出来，其中第 64~72 行根据黑名单数据库中封装数据的 mode 来判断当前号码的拦截方式
- 第 87~91 行，使用 ViewHolder 优化 ListView，减少组件对象使用 findViewById 的多次创建
- 第 96~103 行，去掉 RadioButton 切换界面时界面上数据的重影效果

上面的代码书写好之后，运行程序，效果图如图 4-13 所示。



图 4-13 黑名单界面效果

考虑到黑名单界面的标题部分，在右边有个添加黑名单数据的按钮，这里需要修改之前的布局文件 `fragment_firewall_black.xml`，修改后的布局如文件【4-20】所示。

【文件 4-20】 `res/layout/fragment_firewall_black.xml`

```
1. <?xml version="1.0" encoding="utf-8"?>
2. <LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
3.     android:layout_width="match_parent"
4.     android:layout_height="match_parent"
5.     android:orientation="vertical" >
6.     <RelativeLayout
7.         android:layout_width="match_parent"
8.         android:layout_height="wrap_content" >
9.         <TextView
10.             style="@style/TitleBarTextView"
11.             android:text="黑名单管理" />
12.         <!-- android:layout_weight="1" 表示渲染的先后顺序。如果值越小。越先绘制 -->
13.
14.         <Button
15.             android:id="@+id/bt_add_black_number"
16.             android:layout_width="wrap_content"
17.             android:layout_height="wrap_content"
18.             android:layout_alignParentRight="true"
19.             android:text="添加" />
20.     </RelativeLayout>
21.     <ListView
22.         android:id="@+id/list_view"
23.         android:layout_width="match_parent"
24.         android:layout_height="match_parent" >
25.     </ListView>
26. </LinearLayout>
```

修改后运行程序，界面效果如下图 4-14 所示。



图 4-14 黑名单界面

4.4 添加黑名单

4.4.1 添加黑名单界面

从功能介绍中可以看出，添加黑名单界面主要由三部分组成，第一部分包含一个 `EditText`，用于输入要拦截的电话号码，第二部分包含两个 `CheckBox`，用于选择拦截模式，第三部分包含两个按钮，当点击“添加”按钮时，直接将输入框中的黑名单号码添加到黑名单中，添加黑名单的图形化界面如图 4-15 所示。



图 4-15 添加黑名单的对话框

为实现弹出的对话框界面，我们需要创建一个布局文件 `dialog_add_black_number.xml` 来进行填充，具体代码如文件【4-21】所示。

【文件 4-21】 res/layout/dialog_add_black_number.xml

```
1.  <?xml version="1.0" encoding="utf-8"?>
2.  <LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
3.      android:layout_width="match_parent"
4.      android:layout_height="match_parent"
5.      android:orientation="vertical" >
6.      <TextView
7.          style="@style/TitleBarTextView"
8.          android:gravity="center"
9.          android:text="添加黑名单" />
10.
11.     <!-- 输入电话号码 -->
12.     <EditText
13.         android:id="@+id/et_phone"
14.         android:layout_width="match_parent"
15.         android:layout_height="wrap_content"
16.         android:inputType="phone" />
17.     <LinearLayout
18.         android:layout_width="match_parent"
19.         android:layout_height="wrap_content"
20.         android:orientation="horizontal" >
21.         <CheckBox
22.             android:id="@+id/cb_sms"
23.             android:layout_width="0dp"
24.             android:layout_height="wrap_content"
25.             android:layout_weight="1"
26.             android:text="短信拦截" />
27.         <CheckBox
28.             android:id="@+id/cb_phone"
29.             android:layout_width="0dp"
30.             android:layout_height="wrap_content"
31.             android:layout_weight="1"
32.             android:text="电话拦截" />
33.     </LinearLayout>
34.
35.     <LinearLayout
36.         android:layout_width="match_parent"
37.         android:layout_height="wrap_content"
38.         android:orientation="horizontal" >
39.         <Button
40.             android:id="@+id/bt_ok"
41.             android:layout_width="0dp"
42.             android:layout_height="wrap_content"
43.             android:layout_weight="1"
```

```
44.         android:background="@drawable/dg_btn_confirm_selector"
45.         android:text="确定" />
46.     <Button
47.         android:id="@+id/bt_cancel"
48.         android:layout_width="0dp"
49.         android:layout_height="wrap_content"
50.         android:layout_weight="1"
51.         android:background="@drawable/dg_button_cancel_selector"
52.         android:text="取消" />
53. </LinearLayout>
54. </LinearLayout>
```

4.4.2 添加黑名单逻辑

为了让当前对话框弹出，我们需要拿到 Button 的点击事件，让 FirewallBlackFragment 实现 OnClickListener 接口，在重写的 onClick() 方法中根据 id 进行操作，在 onCreateView() 中首先拿到 Button 的 id，然后 setOnClickListener(this)，对话框弹出的具体逻辑代码如下所示。

```
1.     @Override
2.     public void onClick(View v) {
3.         switch (v.getId()) {
4.             case R.id.bt_add_black_number:
5.                 addBlackNumber();
6.                 break;
7.             default:
8.                 break;
9.         }
10.    }
11.    /**
12.     * 添加黑名单
13.     */
14.    private void addBlackNumber() {
15.        AlertDialog.Builder builder = new Builder(getActivity());
16.        View view = View.inflate(getActivity(),
17.            R.layout.dialog_add_black_number, null);
18.        builder.setView(view);
19.        builder.show();
20.    }
```

运行程序，如图 4-16 所示，发现对话框已成功弹出。



图 4-16 黑名单对话框

接下来，我们要实现向黑名单数据库中添加数据的逻辑，具体如下所示。

```
1.  /* 添加黑名单*/
2.  private void addBlackNumber() {
3.      AlertDialog.Builder builder = new Builder(getActivity());
4.      View view = View.inflate(getActivity(),
5.          R.layout.dialog_add_black_number, null);
6.      // 电话号码输入框
7.      final EditText et_phone = (EditText) view.findViewById(R.id.et_phone);
8.      // 短信拦截
9.      final CheckBox cb_sms = (CheckBox) view.findViewById(R.id.cb_sms);
10.     // 电话拦截
11.     final CheckBox cb_phone = (CheckBox) view.findViewById(R.id.cb_phone);
12.     // 确定
13.     Button bt_ok = (Button) view.findViewById(R.id.bt_ok);
14.     // 取消
15.     Button bt_cancel = (Button) view.findViewById(R.id.bt_cancel);
16.     bt_cancel.setOnClickListener(new OnClickListener() {
17.         @Override
18.         public void onClick(View v) {
19.             // TODO Auto-generated method stub
20.             dialog.dismiss();
21.         }
22.     });
23.
24.     bt_ok.setOnClickListener(new OnClickListener() {
25.         @Override
26.         public void onClick(View v) {
27.             // 获取到电话号码
28.             String str_phone = et_phone.getText().toString().trim();
```

```
29.         // 判断电话号码是否有值
30.         if (TextUtils.isEmpty(str_phone)) {
31.             Toast.makeText(getActivity(), "请输入电话号码", 0).show();
32.             return;
33.         }
34.         String mode = "0";
35.         // 判断短信和电话是否全部勾选。如果全部勾选就是全部拦截
36.         if (cb_sms.isChecked() && cb_phone.isChecked()) {
37.             mode = "1";
38.         } else if (cb_phone.isChecked()) {
39.             mode = "2";
40.         } else if (cb_sms.isChecked()) {
41.             mode = "3";
42.         } else {
43.             Toast.makeText(getActivity(), "请勾选拦截模式", 0).show();
44.             return;
45.         }
46.         // 把黑名单添加到数据库
47.         dao.add(str_phone, mode);
48.         BlackNumberInfo info = new BlackNumberInfo();
49.         info.mode = mode;
50.         info.number = str_phone;
51.         // 添加到集合
52.         mDatas.add(0, info);
53.         // 刷新界面数据
54.         adapter.notifyDataSetChanged();
55.         dialog.dismiss();
56.     }
57. });
58. builder.setView(view);
59. dialog=builder.show();
60. }
```

运行程序，向数据库中添加数据，效果图如图所示，如图 4-17 可知，可以准确向黑名单数据库中添加数据。



图 4-17 向黑名单数据库中添加数据

4.5 删除黑名单

4.5.1 删除黑名单的 UI

观察黑名单界面中每个条目的右边都有一个类似垃圾箱的图标，该图标的功能便是删除当前条目数据，下面我们就开始实现这个功能，为了让该图标只具有点击该按钮才会获取焦点的特性，这里将原来的 `ImageView` 改为 `ImageButton`，代码如下所示。

```
1. <ImageButton
2.     android:id="@+id/iv_delete"
3.     android:layout_width="wrap_content"
4.     android:layout_height="wrap_content"
5.     android:layout_alignParentRight="true"
6.     android:layout_margin="10dp"
7.     android:background="@android:color/transparent"
8.     android:src="@drawable/ic_delete_btn_selector" />
```

上面设置选择器是让图标看着更加美观，其中选择器 `ic_delete_btn_selector` 的代码如文件【4-22】所示。

【文件 4-22】 res/drawable/ic_delete_btn_selector.xml

```
1. <?xml version="1.0" encoding="utf-8"?>
2. <selector
3.     xmlns:android="http://schemas.android.com/apk/res/android">
4.     <item android:state_pressed="true" android:drawable="@drawable/
5.         ic_delete_btn_focused" />
6.     <item android:state_focused="true" android:state_window_focused="true" android:
7.         drawable="@drawable/ic_delete_btn_focused" />
8.     <item android:state_focused="true" android:state_window_focused="false" android:
9.         drawable="@drawable/ic_delete_btn_focused" />
10.    <item android:drawable="@drawable/ic_delete_btn" />
11. </selector>
```


此时，运行程序，点击第一个条目后面的图标，效果如图 4-18 所示。



图 4-18 黑名单数据图标的点击效果

但是，当点击删除图标之后再次点击该条目的其他区域时，该条目已经获取不到焦点了，这是因为 `ImageButton` 是位于该条目之上的，它抢走了该条目的焦点，为了实现该条目在点击删除图标之后仍能获取到焦点，需要在当前黑名单的条目布局中添加一条属性 `android:descendantFocusability="blocksDescendants"`，该属性的作用便是可以直接透过上面的控件，将焦点传递到下一层控件，修改后的布局文件代码如下所示。

```
1.  <?xml version="1.0" encoding="utf-8"?>
2.  <RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
3.      android:layout_width="match_parent"
4.      android:layout_height="match_parent"
5.      android:orientation="vertical"
6.      android:descendantFocusability="blocksDescendants">
7.  <!-- android:descendantFocusability="blocksDescendants" 透过上面的控件。直接把焦点传递到下层
8.  -->
9.      <TextView
10.         android:id="@+id/tv_phone"
11.         android:layout_width="wrap_content"
12.         android:layout_height="wrap_content"
13.         android:layout_marginLeft="10dp"
14.         android:layout_marginTop="10dp"
15.         android:text="电话号码"
16.         android:textColor="#000"
17.         android:textSize="18sp" />
18.      <TextView
19.         android:id="@+id/tv_mode"
20.         android:layout_width="wrap_content"
21.         android:layout_height="wrap_content"
22.         android:layout_below="@id/tv_phone"
23.         android:layout_marginLeft="10dp"
```

```

24.         android:layout_marginTop="3dp"
25.         android:text="拦截的模式"
26.         android:textColor="#88000000"
27.         android:textSize="14sp" />
28.     <ImageButton
29.         android:id="@+id/iv_delete"
30.         android:layout_width="wrap_content"
31.         android:layout_height="wrap_content"
32.         android:layout_alignParentRight="true"
33.         android:layout_margin="10dp"
34.         android:background="@android:color/transparent"
35.         android:src="@drawable/ic_delete_btn_selector" />
36. </RelativeLayout>

```

运行程序，演示效果图，如图 4-19 所示。

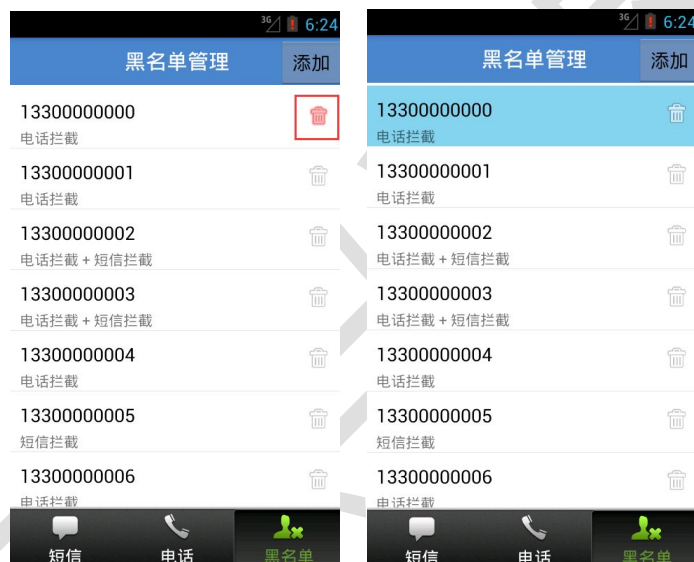


图 4-19 黑名单数据条目的点击效果

4.5.2 删除黑名单的逻辑代码

当我们点击删除图标时，需要将当前数据从黑名单数据库中删除并刷新当前界面，该部分代码的实现现在 `getView` 中进行修改，代码如下所示。

```

1. // 拿到当前条目中删除图标的对象，设置点击事件
2. holder.iv_delete.setOnClickListener(new OnClickListener() {
3.     public void onClick(View v) {
4.         //从数据库中删除数据
5.         dao.delete(info.number);
6.         //删除数据集中的数据
7.         mDataas.remove(info);
8.         //让适配器刷新当前界面
9.         adapter.notifyDataSetChanged();
10.    }
11. });

```

运行程序，依次点击删除第一条和第二条数据，效果图如图 4-20 所示。



图 4-20 删除黑名单数据的效果

4.6 分页加载黑名单

4.6.1 黑名单提示信息

为了提示用户添加黑名单数据，我们可以在用户点击进入黑名单界面时展示一个图标，当添加数据后该图标便会消失，进而展示添加的数据。修改当前的布局文件，使用帧布局将提示添加数据的图片和 ListView 包裹起来，代码如下所示。

```
1. <FrameLayout
2.     android:layout_width="match_parent"
3.     android:layout_height="match_parent" >
4.
5.     <!-- 先添加的在下面。后添加的在上面 需要注意：添加一张空图片。一定要放到最下面 -->
6.     <ImageView
7.         android:id="@+id/empty"
8.         android:layout_width="wrap_content"
9.         android:layout_height="wrap_content"
10.        android:layout_gravity="center"
11.        android:src="@drawable/empty"
12.        android:visibility="invisible" />
13.     <ListView
14.         android:id="@+id/list_view"
15.         android:layout_width="match_parent"
16.         android:layout_height="match_parent" >
17.     </ListView>
18. </FrameLayout>
```

在原文代码中，我们要获取当前图片的 View 对象，并添加 `mListView.setEmptyView(mEmpty)` 这句代码，它意味着当 `mListView` 中的数据为空时，当前图片就会显示。

然后，先清楚应用中的数据，重新进入黑名单界面，效果如图 4-21 所示。



图 4-21 第一次进入黑名单界面的效果

前面我们在展示黑名单数据时是一次性全部加载进来，但是当数据库中的数据很多时这就会造成获取数据耗时过长，用户体验非常不好，为了优化这部分内容，这里我们使用分页加载数据的方法来解决。

分页加载，顾名思义就是每一次访问数据库时都是只获取一部分数据，当用户滑动界面到当前请求数据集的最后一个数据时，便再次向数据库请求数据来展示。

4.6.2 ListView 滚动状态监听

这里，为了避免造成主线程阻塞，将请求数据库数据的代码在子线程中进行。另外，需要给当前的 `ListView` 设置滚动监听事件，在其滚动状态发生改变时根据当前所处状态进行分页判断的逻辑。

首先，观察 `ListView` 的滚动监听事件，代码如下所示。

```
1. // 设置滚动监听
2. mListView.setOnScrollListener(new OnScrollListener() {
3.     /**
4.      * 状态改变的时候才调用 第一个参数：表示 listview 第二个参数：表示滚动的状态
5.      */
6.     @Override
7.     public void onScrollStateChanged(AbsListView view, int scrollState) {
8.         switch (scrollState) {
9.             // 闲置状态(停止的时候调用)
10.            case OnScrollListener.SCROLL_STATE_IDLE:
11.                System.out.println("SCROLL_STATE_IDLE");
12.                int lastVisiblePosition = mListView.getLastVisiblePosition();
13.                System.out.println("lastVisiblePosition---"+ lastVisiblePosition);
14.                break;
```

```

15.         // SCROLL_STATE_TOUCH_SCROLL 触摸到屏幕的时候调用
16.         case OnScrollListener.SCROLL_STATE_TOUCH_SCROLL:
17.             System.out.println("SCROLL_STATE_TOUCH_SCROLL");
18.             break;
19.         // 惯性滑动
20.         case OnScrollListener.SCROLL_STATE_FLING:
21.             System.out.println("SCROLL_STATE_FLING");
22.             break;
23.     }
24. }
25. /**
26.  * 时时调用
27.  */
28. @Override
29. public void onScroll(AbsListView view, int firstVisibleItem,
30.     int visibleItemCount, int totalItemCount) {
31.     System.out.println("onScroll");
32. }
33. });

```

运行程序，观察 ListView 在滚动过程中，上面一些状态信息的输出，如图 4-22 所示。

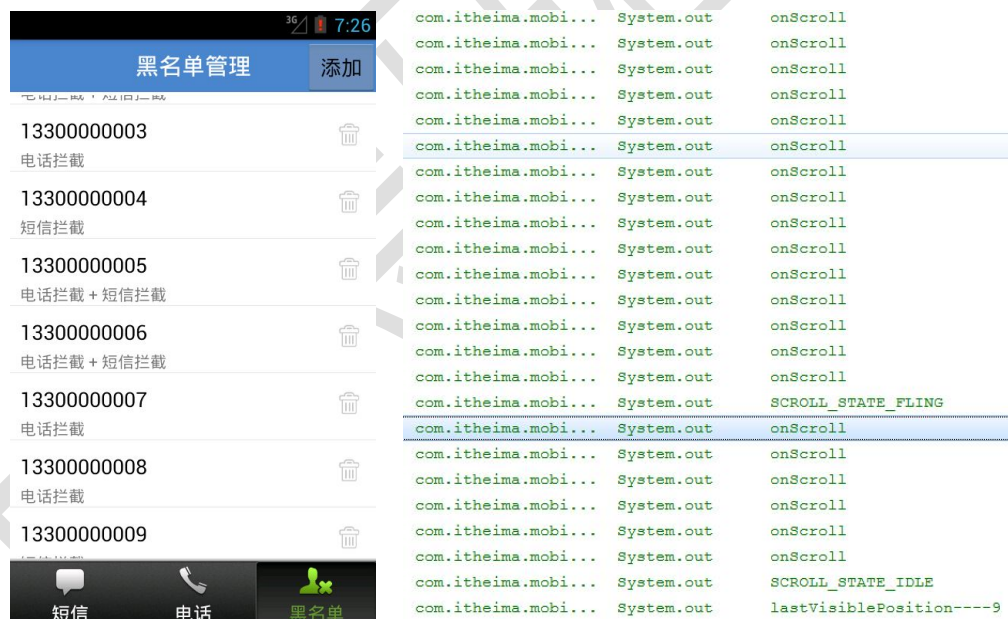


图 4-22 ListView 滚动状态的输出信息

4.6.3 黑名单分页加载

根据以上特性，可知，我们可以在 `onScrollStateChanged` 方法中，当滚动状态 `scrollState` 的值为 `OnScrollListener.SCROLL_STATE_IDLE` 时，即闲置状态时，根据当前可见的最后一个条目的位置 `lastVisiblePosition`，来判断当前数据是否是最后一条，从而决定是否要再次分页加载数据。

接下来，就是我们分页加载的主要逻辑代码的部分，首先需要在 ListView 的滚动状态监听事件中进行改写，具体代码如下所示。

```
1.  /**
2.     * 每页加载 20 条数据
3.     */
4.     private int pageSize = 20;
5.     /**
6.     * 默认从第 0 条开始加载数据
7.     */
8.     private int startIndex = 0;
9.     //下面的代码是 ListView 的滚动监听事件中的
10.    switch (scrollState) {
11.        // 闲置状态 (停止的时候调用)
12.        case OnScrollListener.SCROLL_STATE_IDLE:
13.            // System.out.println("SCROLL_STATE_IDLE");
14.            // 最后一条可见条目的位置
15.            int lastVisiblePosition = mListView
16.                .getLastVisiblePosition();
17.            System.out.println("lastVisiblePosition----"
18.                + lastVisiblePosition);
19.            // 判断当前的数据是否是最后一条
20.            if (lastVisiblePosition == mDatas.size() - 1) {
21.                startIndex += pageSize;
22.                // 判断开始的位置是否大于所有的记录
23.                if (startIndex >= mCount) {
24.                    Toast.makeText(getActivity(), "没有更多数据", 0).show();
25.                    return;
26.                }
27.                // 分页加载数据
28.                initData();
29.            }
30.            break;
```

其中，mCount 的值是数据库中所有数据的总条目数，对于 initData()的具体代码如下所示。

```
1.  /**
2.     * 初始化所有的数据
3.     */
4.     private void initData() {
5.         dao = new BlackNumberDao(getActivity());
6.         // 返回所有的记录数
7.         mCount = dao.getCount();
8.         new Thread() {
9.             public void run() {
10.                // 判断是否是第一次加载数据
11.                if (mDatas != null && mDatas.size() == 0) {
12.                    // 获取到所有的黑名单
13.                    // 初始化 20 条数据
```

```

14.         mDatas = dao.findPart(pageSize, startIndex);
15.     } else {
16.         // 追加后面 20 条数据
17.         mDatas.addAll(dao.findPart(pageSize, startIndex));
18.     }
19.     // 运行到主线程
20.     activity.runOnUiThread(new Runnable() {
21.         @Override
22.         public void run() {
23.             //进度条隐藏
24.             progressBar1.setVisibility(View.INVISIBLE);
25.             if (adapter == null) {
26.                 adapter = new BlackAdapter();
27.                 mListview.setAdapter(adapter);
28.             } else {
29.                 //刷新数据
30.                 adapter.notifyDataSetChanged();
31.             }
32.             mListview.setEmptyView(mEmpty);
33.         }
34.     });
35. };
36. }.start();
37. }

```

运行程序，观察当第一次进入黑名单界面时，滑动 ListView 到第 20 条数据时，会加载更多的 20 条数据，当滑动到第 200 条时，显示已经没有更多数据了，如图 4-23 所示。

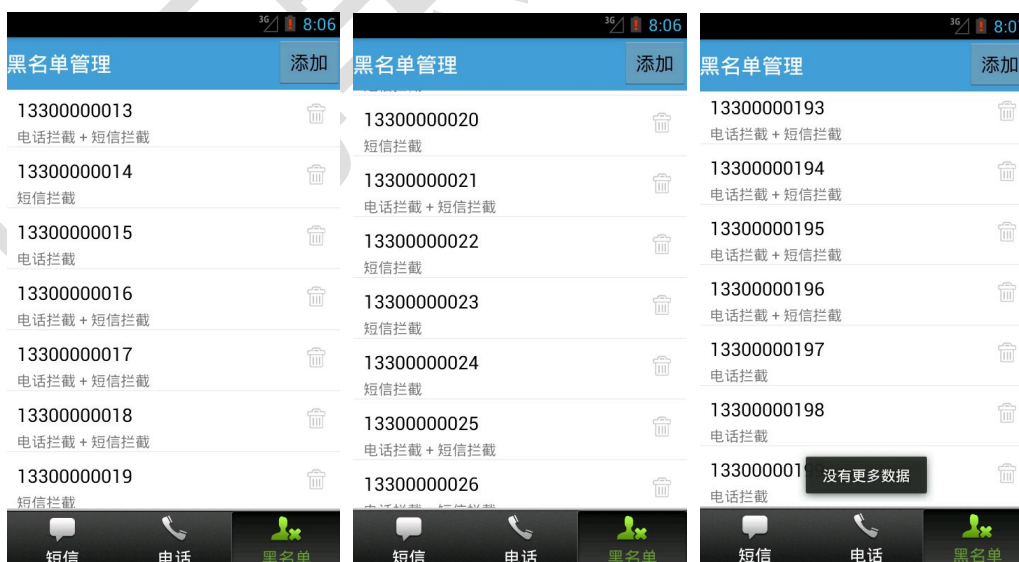


图 4-23 ListView 的分页加载数据

4.6.4 黑名单加载进度条

观察完整版的效果图可知，在黑名单加载数据时会出现一个不断转圈的进度条，如下图 4-24 所示。



图 4-24 黑名单界面进度条

为了实现这样的效果，我们需要修改黑名单的布局文件，具体代码如文件【4-23】如下所示。

【文件 4-23】 res/layout/fragment_firewall_black.xml

```
1. <?xml version="1.0" encoding="utf-8"?>
2. <LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
3.     android:layout_width="match_parent"
4.     android:layout_height="match_parent"
5.     android:orientation="vertical" >
6.     <RelativeLayout
7.         android:layout_width="match_parent"
8.         android:layout_height="wrap_content" >
9.         <TextView
10.             style="@style/TextViewTitleBar"
11.             android:text="黑名单管理" />
12.         <!-- android:layout_weight="1" 表示渲染的先后顺序。如果值越小。越先绘制 -->
13.         <Button
14.             android:id="@+id/bt_add_black_number"
15.             android:layout_width="wrap_content"
16.             android:layout_height="wrap_content"
17.             android:layout_alignParentRight="true"
18.             android:text="添加" />
19.     </RelativeLayout>
20.     <FrameLayout
21.         android:layout_width="match_parent"
22.         android:layout_height="match_parent" >
23.         <!-- 先添加的在下面。后添加的在上面 -->
24.         <!-- 需要注意：添加一张空图片。一定要放到最下面 -->
```



```
25.         <ImageView
26.             android:id="@+id/empty"
27.             android:layout_width="wrap_content"
28.             android:layout_height="wrap_content"
29.             android:layout_gravity="center"
30.             android:src="@drawable/empty"
31.             android:visibility="invisible" />
32.         <ListView
33.             android:id="@+id/list_view"
34.             android:layout_width="match_parent"
35.             android:layout_height="match_parent" >
36.         </ListView>
37.         <ProgressBar
38.             android:id="@+id/progressBar1"
39.             android:layout_width="wrap_content"
40.             android:layout_height="wrap_content"
41.             android:layout_gravity="center"
42.             android:indeterminateDrawable="@drawable/progress_medium_white"
43.             android:visibility="invisible" />
44.     </FrameLayout>
45. </LinearLayout>
```

Android 原生的进度条非常不美观，这里 `ProgressBar` 中的 `progress_medium_white` 是我们自定义的进度条样式，它是旋转的动画效果，代码如下所示。

```
1. <?xml version="1.0" encoding="utf-8"?>
2. <animated-rotate xmlns:android="http://schemas.android.com/apk/res/android"
3.     android:drawable="@drawable/loading"
4.     android:pivotX="50%"
5.     android:pivotY="50%" />
```

然后在黑名单界面的逻辑代码中也要进行代码修改，当界面刚开始加载时需要将进度条显示出来，当开辟子线程去获取数据库数据，然后进行显示时进度条需要隐藏。

首先，在 `FirewallBlackFragment` 的 `onCreateView` 添加以下代码。

```
1. progressBar1 = (ProgressBar) view.findViewById(R.id.progressBar1);
2. progressBar1.setVisibility(View.VISIBLE);
```

另外，在开辟子线程请求数据时，然后在主线程显示数据时隐藏进度条，如下所示。

```
1.         // 运行到主线程
2.         activity.runOnUiThread(new Runnable() {
3.             @Override
4.             public void run() {
5.                 //进度条隐藏
6.                 progressBar1.setVisibility(View.INVISIBLE);
7.                 if (adapter == null) {
8.                     adapter = new BlackAdapter();
9.                     mListview.setAdapter(adapter);
```

```

10.         } else {
11.             //刷新数据
12.             adapter.notifyDataSetChanged();
13.         }
14.         mListview.setEmptyView(mEmpty);
15.     }
16. });

```

4.7 黑名单拦截

这里，我们将在常用设置界面上添加一个骚扰拦截功能，点击开启后会开启一个服务，在该服务中实现黑名单拦截功能，然后将电话和短信记录删除不让其在界面中显示，本节将针对黑名单拦截功能进行详细讲解。

4.7.1 设置中心的骚扰拦截 UI

观察效果图，如下图 4-25，需要在 SettingActivity 界面的布局 activity_setting.xml 中进行修改，具体代码如文件【4-24】所示。



图 4-25 常用设置界面中的骚扰拦截

【文件 4-24】res/layout/activity_setting.xml

```

1.  <?xml version="1.0" encoding="utf-8"?>
2.  <LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
3.      xmlns:itheima="http://schemas.android.com/apk/res/com.itheima.mobilesafe_sh3"
4.      android:layout_width="match_parent"
5.      android:layout_height="match_parent"
6.      android:orientation="vertical" >
7.      <TextView
8.          style="@style/TextViewTitleBar"
9.          android:gravity="center"
10.         android:text="设置中心" />
11.      <!-- itheima:itToggle="false" 判断滑动开关是否展示 -->
12.      <com.itheima.mobilesafe_sh3.view.SettingItemView
13.          android:id="@+id/siv_auto_update"

```

```
14.         android:layout_width="match_parent"
15.         android:layout_height="wrap_content"
16.         android:layout_marginTop="10dp"
17.         android:clickable="true"
18.         android:focusable="true"
19.         itheima:itTitle="自动更新"
20.         itheima:itToggle="true"
21.         itheima:itbackground="first" >
22.     </com.itheima.mobilesafe_sh3.view.SettingItemView>
23.     <com.itheima.mobilesafe_sh3.view.SettingItemView
24.         android:id="@+id/siv_black_number"
25.         android:layout_width="match_parent"
26.         android:layout_height="wrap_content"
27.         android:clickable="true"
28.         android:focusable="true"
29.         itheima:itTitle="骚扰拦截"
30.         itheima:itToggle="true"
31.         itheima:itbackground="last" >
32.     </com.itheima.mobilesafe_sh3.view.SettingItemView>
33.     <com.itheima.mobilesafe_sh3.view.SettingItemView
34.         android:id="@+id/siv_show_address"
35.         android:layout_width="match_parent"
36.         android:layout_height="wrap_content"
37.         android:layout_marginTop="10dp"
38.         android:clickable="true"
39.         android:focusable="true"
40.         itheima:itTitle="归属地显示设置"
41.         itheima:itToggle="true"
42.         itheima:itbackground="first" >
43.     </com.itheima.mobilesafe_sh3.view.SettingItemView>
44.     <com.itheima.mobilesafe_sh3.view.SettingItemView
45.         android:id="@+id/siv_address_style"
46.         android:layout_width="match_parent"
47.         android:layout_height="wrap_content"
48.         android:clickable="true"
49.         android:focusable="true"
50.         itheima:itTitle="归属地风格设置"
51.         itheima:itToggle="false"
52.         itheima:itbackground="last" >
53.     </com.itheima.mobilesafe_sh3.view.SettingItemView>
54. </LinearLayout>
```

滑动开关的关闭状态的实现代码，如下所示。

```
1. // 获取骚扰拦截 View
2. mSivBlackNumber = (SettingItemView) findViewById(R.id.siv_black_number);
3. // 骚扰拦截 id
4. case R.id.siv_black_number:
5.     // 如果当前的开关是打开的。那么点击之后就关闭
6.     if (mSivBlackNumber.isToggle()) {
7.         // 关闭图片
8.         mSivBlackNumber.setToggle(false);
9.         System.out.println("滑动开关关闭");
10.    } else {
11.        // 如果之前是关闭的。点击之后就打开
12.        mSivBlackNumber.setToggle(true);
13.        System.out.println("滑动开关打开");
14.    }
15.    break;
```

运行程序，点击骚扰拦截，效果图如图 4-26 所示。



图 4-26 常用设置界面骚扰拦截的滑动开关

4.7.2 设置中心骚扰拦截状态的回显

当用户在常用设置界面点击开启骚扰拦截开关后，会对应的开启或者关闭黑名单拦截服务，这里我们首先需要定义一个拦截电话和短信的服务 `BlackNumberService`，另外当用户再次进入常用设置界面时，骚扰拦截按钮的回显状态需要根据手机后台是否运行 `BlackNumberService` 服务来决定。

之前，我们已经定义了一个判断服务运行的工具类 `ServiceStateUtils`，这部分的实现代码如下所示。

```
1. // 黑名单服务
2. mBlackNumberServiceIntent = new Intent(this, BlackNumberService.class);
3. //onClick 方法中骚扰拦截 id
4. case R.id.siv_black_number:
5.     // 如果当前的开关是打开的。那么点击之后就关闭
6.     if (mSivBlackNumber.isToggle()) {
7.         // 关闭图片
8.         mSivBlackNumber.setToggle(false);
9.         System.out.println("滑动开关关闭");
10.        stopService(mBlackNumberServiceIntent);
```

```
11.         } else {
12.             // 如果之前是关闭的。点击之后就打开
13.             mSivBlackNumber.setToggle(true);
14.             System.out.println("滑动开关打开");
15.             startService(mBlackNumberServiceIntent);
16.         }
17.         break;
18. //在该方法中进行服务状态的判断，用户回显常用设置界面的骚扰拦截开关状态
19. @Override
20. protected void onStart() {
21.     // TODO Auto-generated method stub
22.     super.onStart();
23.     // 判断黑名单服务是否正在运行
24.     if (ServiceStateUtils.isRunningService(getApplicationContext(),
25.         com.itheima.mobilesafe_sh3.service.BlackNumberService.class)) {
26.         mSivBlackNumber.setToggle(true);
27.     } else {
28.         mSivBlackNumber.setToggle(false);
29.     }
30. }
```

运行程序，效果图如图 4-27 所示。

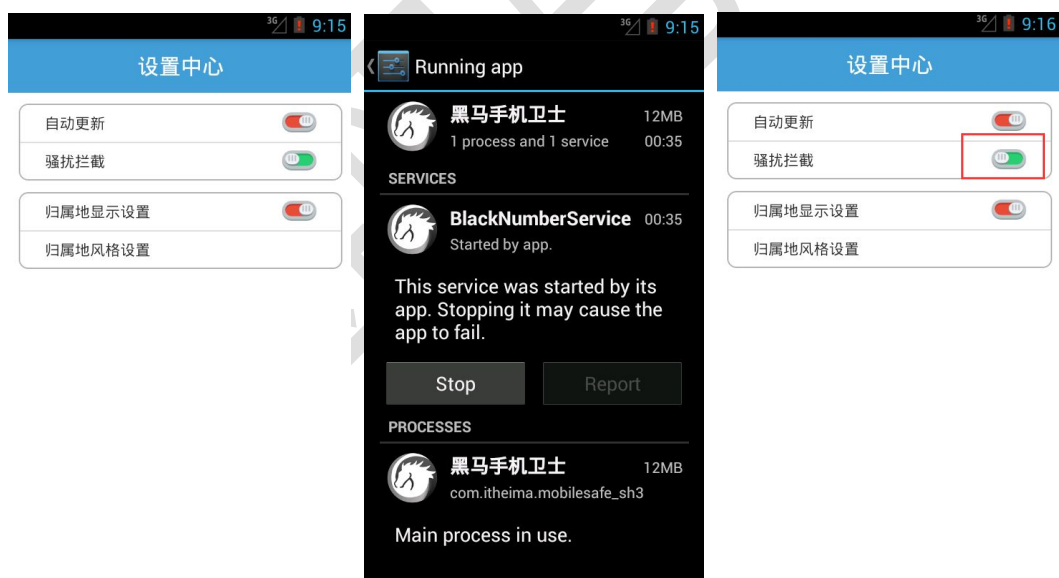


图 4-27 常用设置界面骚扰拦截的滑动开关的回显

4.7.3 拦截短信

我们可以在服务中静态注册一个拦截短信的广播，然后在广播中获取到电话号码，然后查询该号码是否在黑名单数据库中，如果在则判断是哪种拦截模式，进行拦截。接下来创建拦截短信的广播接收者，具体代码如【文件 4-25】所示。

【文件 4-25】 InnerSmsReceiver.java

```

1. private class InnerSmsReceiver extends BroadcastReceiver {
2.     @Override
3.     public void onReceive(Context context, Intent intent) {
4.         System.out.println("InnerSmsReceiver");
5.         // 获取到短信
6.         Object[] objects = (Object[]) intent.getExtras().get("pdus");
7.         for (Object obj : objects) {
8.             SmsMessage smsMessage = SmsMessage.createFromPdu((byte[]) obj);
9.             // 获取到短信内容
10.            String body = smsMessage.getMessageBody();
11.            // 获取到电话号码
12.            String phone = smsMessage.getDisplayOriginatingAddress();
13.            // 根据电话号码查询拦截的模式
14.            String mode = dao.findNumberMode(phone);
15.            /**
16.             * 黑名单的拦截模式 1 全部拦截(电话拦截 + 短信拦截) 2 电话拦截 3 短信拦截
17.             */
18.            if ("1".equals(mode) || "3".equals(mode)) {
19.                System.out.println("被哥拦截了");
20.                //往短信拦截数据库里面添加数据
21.                abortBroadcast();
22.            }
23.            /**
24.             * 根据内容拦截(智能拦截)
25.             */
26.            if (body.contains("xue sheng mei")) {
27.                System.out.println("学生妹被拦截了");
28.                abortBroadcast();
29.            }
30.        }
31.    }
11 }

```

- 第 10 行代码用于获取接收到的短信信息；
- 第 14 行代码根据电话号码在黑名单数据中查询对应的拦截模式；
- 第 18~29 行代码根据查询的拦截模式，如果是短信拦截，则直接拦截并输出语句注册静态广播的逻辑代码如下所示。

```

1. receiver = new InnerSmsReceiver();
2.     IntentFilter filter = new IntentFilter(
3.         "android.provider.Telephony.SMS_RECEIVED");
4.     // 设置优先级
5.     filter.setPriority(2147483647);
6.     // 注册一个短信监听的广播
7.     registerReceiver(receiver, filter);

```

同时需要在 `onDestroy()` 反注册广播，防止内存泄露，如下所示。

```

1.  @Override
2.      public void onDestroy() {
3.          super.onDestroy();
4.          // 反注册
5.          unregisterReceiver(receiver);
6.          receiver = null;
7.          // 当不用了。设置为 null
8.          mTelephonyManager.listen(listener, PhoneStateListener.LISTEN_NONE);
9.          listener = null;
10.     }

```

为了让广播接收者能够拦截短信，必须要在 `AndroidManifest.xml` 文件中注册，具体代码如下所示：

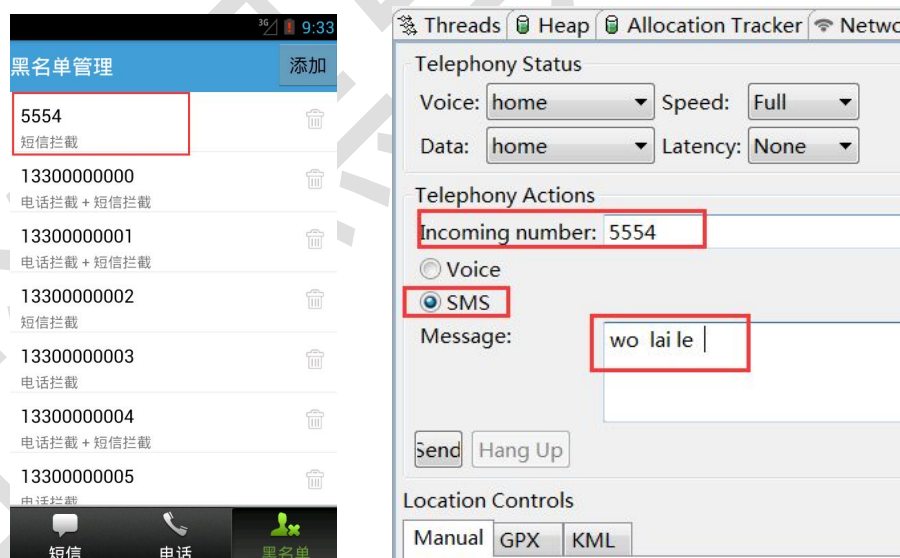
```

<!-- 拦截黑名单信息 -->
<receiver
    android:name="com.itheima.mobilesafe_sh2.receiver.InnerSmsReceiver " >
    <intent-filter android:priority="2147483647" >
        <action android:name="android.provider.Telephony.SMS_RECEIVED"/>
    </intent-filter>
</receiver>

```

上述清单文件中，定义了接收短信的广播，并将广播的优先级设置为最高，这样当有新短信到来时会优先被该广播接收者所接收。

运行程序，效果图如图 4-28 所示。



prefix with pid:, app:, tag: or text: to limit scope.

Application	Tag	Text
com.itheima.mobi...	RetryHandle...	retry error, curr re
com.itheima.mobi...	System.out	InnerSmsReceiver
com.itheima.mobi...	System.out	被哥拦截了

图 4-28 短信拦截

4.7.4 拦截电话

这里，我们要实现黑名单中电话拦截的功能，为了侦听电话状态，我们需要获得系统的电话管理器对象，具体代码如下所示。

```
1.      // 获取到电话管理者
2.      mTelephonyManager = (TelephonyManager) this.getSystemService(TELEPHONY_SERVICE);
3.      // 初始化电话状态监听
4.      listener = new MyPhoneStateListener();
5.      // 设置电话监听
6.      // 第一个参数：表示电话状态的监听
7.      // 第二个参数：表示打电话的状态
8.      mTelephonyManager.listen(listener, PhoneStateListener.LISTEN_CALL_STATE);
```

其中，MyPhoneStateListener 是我们自定义的一个侦听电话状态的监听器，首先我们来看一下这个电话侦听器是如何监听的，书写以下的代码。

```
1.  private class MyPhoneStateListener extends PhoneStateListener {
2.
3.      // public static final int CALL_STATE_IDLE = 0; 闲置状态
4.      // public static final int CALL_STATE_RINGING = 1; 响铃状态
5.      // public static final int CALL_STATE_OFFHOOK = 2; 接通状态
6.      @Override
7.      public void onCallStateChanged(int state, String incomingNumber) {
8.          // TODO Auto-generated method stub
9.          super.onCallStateChanged(state, incomingNumber);
10.         switch (state) {
11.             case TelephonyManager.CALL_STATE_IDLE:
12.                 System.out.println("CALL_STATE_IDLE");
13.                 break;
14.             case TelephonyManager.CALL_STATE_RINGING:
15.                 System.out.println("CALL_STATE_RINGING");
16.                 }
17.                 break;
18.             case TelephonyManager.CALL_STATE_OFFHOOK:
19.                 System.out.println("CALL_STATE_OFFHOOK");
20.                 break;
21.         }
22.         System.out.println("state-----" + state);
23.         System.out.println("incomingNumber-----" + incomingNumber);
24.     }
25. }
```

然后，运行程序，观察日志信息输出，如图 4-29 所示，在 DDMS 模拟 110 给当前模拟器打电话时，观察对应的输出信息可知，用于电话铃响时的 state 为 1，同时可以获取来电号码，而在其他两个状态下获取不到来电号码。因此，拦截电话的逻辑应该在电话铃响时的 CALL_STATE_RINGING 状态下进行电话的拦截。

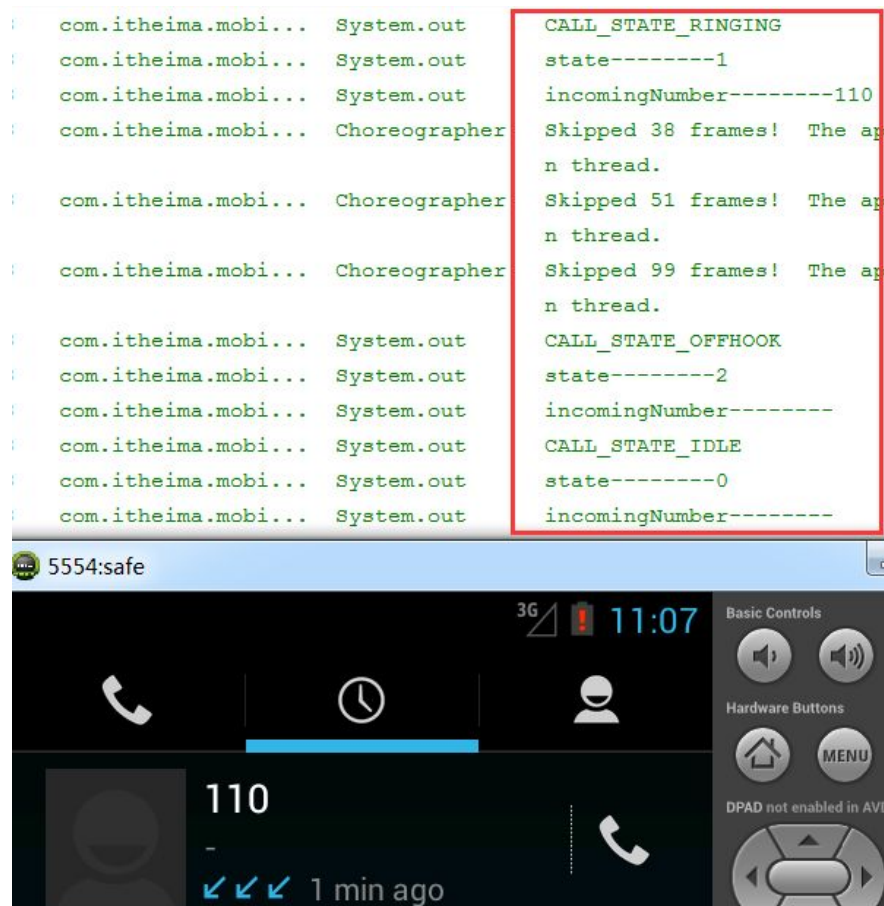


图 4-29 观察来电时侦听的状态信息

当电话铃响时需要挂断电话并且不让该记录显示在界面上，而 Google 工程师为了手机的安全性隐藏了挂断电话的服务方法，因此要实现挂断电话的操作只能通过反射获取底层服务。

这里，我们使用反射的机制获取系统底层挂断电话的方法，如下所示。

```
1.  /**
2.   * 挂断电话
3.   */
4.  public void endCall() {
5.      // 系统底层的源码
6.      // sRegistry =
7.      // ITelephonyRegistry.Stub.asInterface(ServiceManager.getService(
8.      // "telephony.registry"));
9.      // 通过放射的方式获取到到 ServiceManager。然后构建 aidl
10.
11.     try {
12.         // 获取到 ServiceManager 的类名
13.         Class<?> clazz = getClassLoader().loadClass(
14.             "android.os.ServiceManager");
15.         // 暴力反射
16.         // 第一个参数：表示方法名字
```

```

17.         // 第二个参数：表示方法的类型
18.         Method method = clazz.getDeclaredMethod("getService", String.class);
19.         // 调用当前的方法
20.         // 第一个参数：表示谁调用当前方法
21.         IBinder iBinder = (IBinder) method.invoke(null, TELEPHONY_SERVICE);
22.         ITelephony asInterface = ITelephony.Stub.asInterface(iBinder);
23.         // 挂断电话
24.         asInterface.endCall();
25.     } catch (Exception e) {
26.         e.printStackTrace();
27.     }
28. }

```

endCall()方法用于挂断黑名单的呼入电话，该段代码中首先通过反射获取到 ServiceManager 字节码，然后通过该字节码获取 getService()方法，该方法接收一个 String 类型的参数，然后通过 invoke()执行 getService()方法，由于 getService()方法是静态的，因此 invoke()的第一个参数可以为 null，第二个参数是 TELEPHONY_SERVICE。由于 getService()方法的返回值是一个 IBinder 对象（远程服务的代理类），因此需要使用 AIDL 的规则将其转化为接口类型，由于我们的操作是挂断电话，因此需要使用与电话相关的 ITelephony.aidl，然后调用接口中的 endCall()方法将电话挂断即可。

需要注意的是，与电话相关的操作一般都使用 TelephonyManager 类，但是由于挂断电话的方法在 ITelephony 接口中，而这个接口是隐藏的（@hide）在开发时看不到，因此需要使用 ITelephony.aidl。在使用 ITelephony.aidl 时，需要创建一个与它包名一致的包 com.android.internal.telephony 然后把系统的 ITelephony.aidl 文件拷贝进来，同时 ITelephony.aidl 接口关联了 NeighboringCellInfo.aidl，所以也需要一并拷贝进来。不过要注意的是，NeighboringCellInfo.aidl 所在的包名是 android.telephony，因此需要新建一个 android.telephony 包，然后把 NeighboringCellInfo.aidl 放到该包中。如图 4-30 所示。



图 4-30 创建底层的 aidl 文件

另外，想要挂断电话需要配置一个权限，如下所示。

```

1. <!-- 挂断电话权限 -->
2. <uses-permission android:name="android.permission.CALL_PHONE" />

```

运行程序，使用 DDMS 向模拟器打电话，观察结果，如图 4-31 所示，可以正常挂断电话。

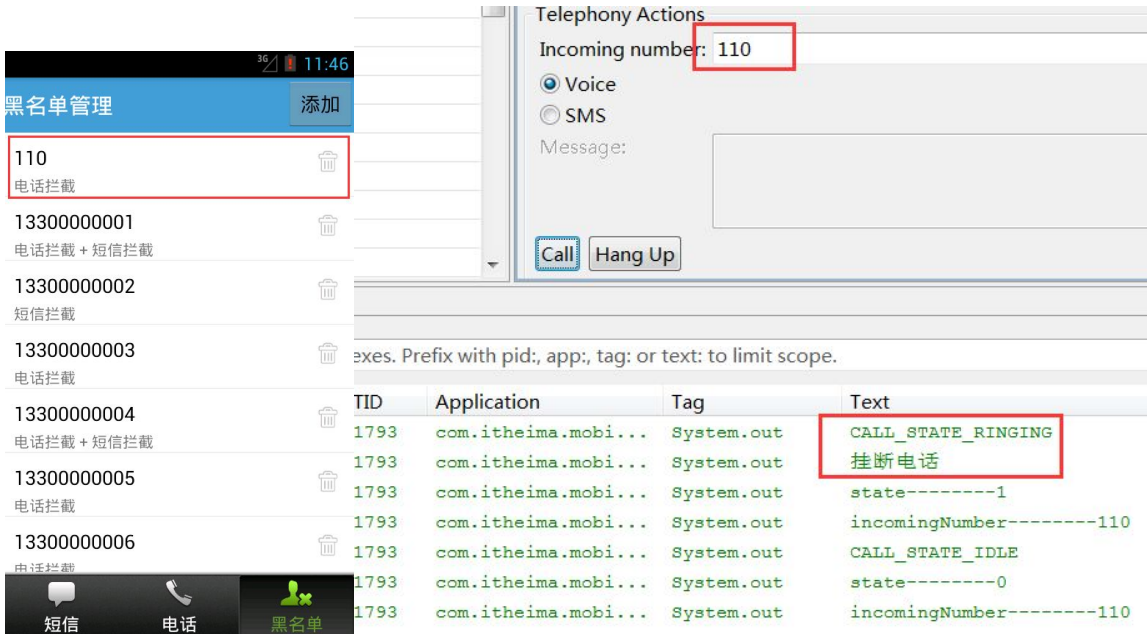


图 4-31 挂断电话功能的实现

上面我们已经实现了拦截电话的功能，但是打开手机的通话记录会发现通话记录仍然存在，如下图 4-32 所示。这样的体验会让用户觉得很奇怪，明明没有来电信息，为什么会出现通话记录呢，这是因为我们没有删除手机系统中通话记录数据库的数据，所以才会显示出来。

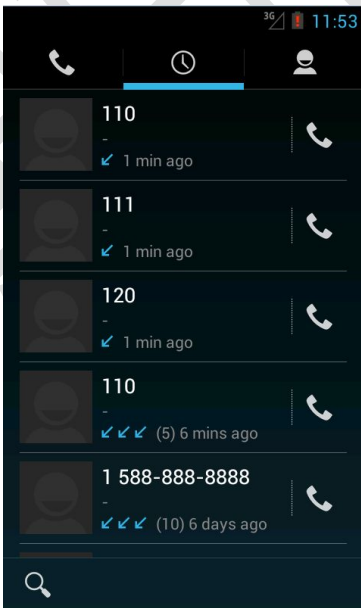


图 4-32 拦截电话后的通话记录

想要删除手机系统中的通话记录，我们需要注册一个内容解析者来操作手机的通话记录数据库，在来电响铃时的逻辑代码如下所示。

```
1. case TelephonyManager.CALL_STATE_RINGING:
2.     System.out.println("CALL_STATE_RINGING");
3.     // 根据电话号码查询拦截的模式
4.     String mode = dao.findNumberMode(incomingNumber);
```

```

5.         if ("1".equals(mode) || "2".equals(mode)) {
6.             System.out.println("挂断电话");
7.             Uri uri = Uri.parse("content://call_log/calls");
8.             // 注册内容解析者
9.             // 第一个参数：表示 uri 的地址
10.            // 第二个参数：如果是 true 表示前缀满足即可
11.            // 第三个参数：内容观察者
12.            getContentResolver().registerContentObserver(uri, true,
13.                new MyContentObserver(new Handler(), incomingNumber));
14.            // 挂断电话
15.            endCall();
16.        }
17.        break;

```

其中，MyContentObserver 是定义的内容观察者，它负责删除手机通话记录数据库中的号码，具体代码如下所示。

```

1.  /**
2.   * 内容观察者
3.   * @author mwqi
4.   */
5.  private class MyContentObserver extends ContentObserver {
6.      private String incomingNumber;
7.      public MyContentObserver(Handler handler, String incomingNumber) {
8.          super(handler);
9.          this.incomingNumber = incomingNumber;
10.     }
11.     // 当数据库发生改变的时候调用当前的方法
12.     @Override
13.     public void onChange(boolean selfChange) {
14.         // TODO Auto-generated method stub
15.         super.onChange(selfChange);
16.         deleteIncomingNumber(incomingNumber);
17.     }
18. }
19. /**
20.  * 删除当前电话号码
21.  * @param incomingNumber
22.  */
23. public void deleteIncomingNumber(String incomingNumber) {
24.     Uri uri = Uri.parse("content://call_log/calls");
25.     ContentResolver resolver = getContentResolver();
26.     resolver.delete(uri, "number = ?", new String[] { incomingNumber });
27. }

```

- 第 5~18 行的 MyContentObserver 是一个内容观察者，用于观察系统联系人的数据库，如果黑名单中的电话呼入时，在系统联系人数据库中产生记录时就调用 deleteCallLog() 方法清除历史记录。

- 第 19~27 行的 `deleteIncomingNumber()` 方法用于清除指定号码的通话历史记录，当黑名单中的电话呼入时，手机系统通话记录中会显示该条记录，因此需要把通话记录中的黑名单通话记录删除。手机上拨打电话接听电话等产生的记录都在 Uri 为 `content://call_log/calls` 的路径上，因此，可以使用 `ContentResolver` 对象查询并删除数据库中黑名单号码所产生的记录即可。

想要操作手机系统的联系人，仍然需要配置一些必须的权限，如下所示。

```
1. <!-- 读取联系人和写联系人的权限 -->
2.     <uses-permission android:name="android.permission.READ_CONTACTS" />
3.     <uses-permission android:name="android.permission.WRITE_CONTACTS" />
```

至此，骚扰拦截模块已经全部完成，接下来对该模块的功能进行完整测试。首先，在添加黑名单界面手动输入一个号码，比如号码 10086，然后使用 DDMS 模拟拨打电话，并在代码中打上 Log 以便观察结果，这时会看到在 Log 中打印出了电话号码以及短信内容，但是手机界面上没有任何显示，说明黑名单拦截功能运行正常，如图 4-33 所示。

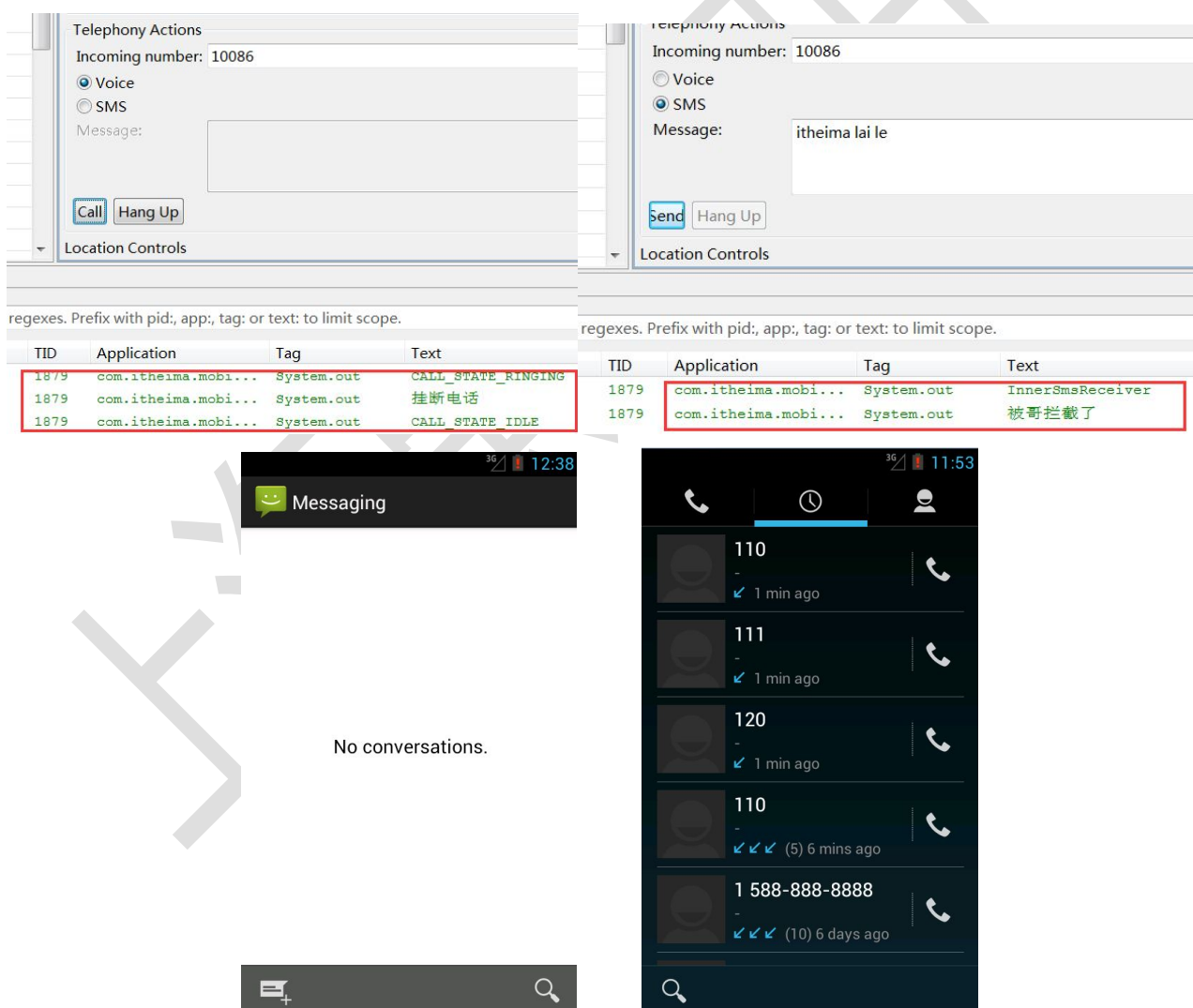


图 4-33 拦截电话和拦截短信

4.8 短信拦截记录

4.8.1 防火墙短信数据库

前面我们在 BlackNumberService 服务中实现对黑名单中号码的短信拦截，但是用户也希望能够在短信拦截界面看到这些被拦截的信息，对于这些被拦截的短信，我们可以使用数据库进行存储，在原来的黑名单数据库 safe.db 中再创建一个防火墙短信的表，用来存放这些被拦截的短信。

首先，需要在 BlackNumberOpenHelper 的 onCreate() 方法中再创建一张表 firewallsms，如下所示。

```
1. db.execSQL("create table firewallsms (_id integer primary key autoincrement,number
   varchar(20),body varchar(20))");
```

接下来，需要创建一个操作这张表的 dao 文件 FirewallSmsDao.java，它里面需要定义两个方法，一个向数据库中插入数据的方法，另一个是列出所有数据的方法，具体代码如文件【4-26】所示。

【文件 4-26】 com.itheima.mobilesafe_sh3.dao/FirewallSmsDao.java

```
2. /**
3.  * 短信拦截的数据库
4.  */
5. public class FirewallSmsDao {
6.     private BlackNumberOpenHelper helper;
7.     public FirewallSmsDao(Context context) {
8.         helper = new BlackNumberOpenHelper(context);
9.     }
10.    /**
11.     * 添加到短信数据库里面
12.     * @param number 电话号码
13.     * @param body 短信内容
14.     */
15.    public void add(String number, String body) {
16.        SQLiteDatabase db = helper.getWritableDatabase();
17.        ContentValues values = new ContentValues();
18.        values.put("number", number);
19.        values.put("body", body);
20.        db.insert("firewallsms", null, values);
21.    }
22.    /**
23.     * 查询所有的拦截短信
24.     * @return
25.     */
26.    public List<FirewallSmsInfo> findAll(){
27.        SQLiteDatabase db = helper.getReadableDatabase();
28.        Cursor cursor = db.query("firewallsms", new String[]{"number","body"}, null,
29. null, null, null, null);
30.        List<FirewallSmsInfo> mDatas = new ArrayList<FirewallSmsInfo>();
31.        while(cursor.moveToNext()){
```

```

32.         FirewallSmsInfo info = new FirewallSmsInfo();
33.         info.number = cursor.getString(0);
34.         info.body = cursor.getString(1);
35.         mDataas.add(info);
36.     }
37.     return mDataas;
38. }
39. }

```

4.8.2 短信拦截记录 UI

短信拦截界面的 UI 的界面效果如下图 4-34 所示，可以看出它跟黑名单界面没有什么大的区别，按照黑名单界面的布局形式创建 FirewallSmsFragment.java 的布局文件 fragment_firewall_sms.xml，它的具体内容如文件【4-27】所示。



图 4-34 拦截短信记录的界面分析

【文件 4-27】 res/layout/fragment_firewall_sms.xml

```

1. <?xml version="1.0" encoding="utf-8"?>
2. <LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
3.     android:layout_width="match_parent"
4.     android:layout_height="match_parent"
5.     android:orientation="vertical" >
6.     <TextView
7.         style="@style/TitleBarTextView"
8.         android:text="短信拦截记录"
9.         android:gravity="center"/>
10.    <ListView
11.        android:id="@+id/list_view"
12.        android:layout_width="match_parent"

```

```
13.         android:layout_height="match_parent" >
14.     </ListView>
15. </LinearLayout>
```

接着，关于 ListView 的每个条目创建布局文件 item_fragment_firewall_sms.xml，它的具体内容如文件【4-28】所示。

【文件 4-28】 res/layout/item_fragment_firewall_sms.xml

```
1.  <?xml version="1.0" encoding="utf-8"?>
2.  <LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
3.      android:layout_width="match_parent"
4.      android:layout_height="match_parent"
5.      android:layout_margin="10dp"
6.      android:orientation="vertical" >
7.      <TextView
8.          android:id="@+id/tv_phone"
9.          android:layout_width="match_parent"
10.         android:layout_height="wrap_content"
11.         android:text="电话号码"
12.         android:textColor="#000"
13.         android:textSize="18sp" />
14.      <TextView
15.          android:id="@+id/tv_body"
16.          android:layout_width="match_parent"
17.          android:layout_height="wrap_content"
18.          android:singleLine="true"
19.          android:text="短信内容"
20.          android:textColor="#88000000"
21.          android:textSize="16sp" />
22. </LinearLayout>
```

从上可以看出，ListView 的条目布局就只有两个 TextView，一个显示拦截短信的电话号码，另一个显示拦截的短信内容。

4.8.3 黑名单服务中短信拦截记录的添加

首先，在黑名单的服务 BlackNumberService 里面，进行短信拦截的逻辑代码中当想要操作数据库就需要先拿到操作短信拦截数据库的 dao 对象，如下所示。

```
1.     //黑名单短信拦截数据库
2.     smsDao = new FirewallSmsDao(getApplicationContext());
```

然后在，短信接收者中，检测到来电号码的拦截模式为 1 或者 3 时，我们需要将当前的来电号码和短信内容存放到之前创建的短信拦截数据库中，具体如下所示。

```
1.     // 短信接收者
2.     private class InnerSmsReceiver extends BroadcastReceiver {
3.         @Override
4.         public void onReceive(Context context, Intent intent) {
5.             System.out.println("InnerSmsReceiver");
```



```

6.          // 获取到短信
7.          Object[] objects = (Object[]) intent.getExtras().get("pdus");
8.          for (Object obj : objects) {
9.              SmsMessage smsMessage = SmsMessage.createFromPdu((byte[]) obj);
10.             // 获取到短信内容
11.             String body = smsMessage.getMessageBody();
12.             // 获取到电话号码
13.             String phone = smsMessage.getDisplayOriginatingAddress();
14.             // 根据电话号码查询拦截的模式
15.             String mode = dao.findNumberMode(phone);
16.             /**
17.              * 黑名单的拦截模式 1 全部拦截(电话拦截 + 短信拦截) 2 电话拦截 3 短信拦截
18.              */
19.             if ("1".equals(mode) || "3".equals(mode)) {
20.                 System.out.println("被哥拦截了");
21.                 //往短信拦截数据库里面添加数据
22.                 smsDao.add(phone, body);
23.                 abortBroadcast();
24.             }
25.             /**
26.              * 根据内容拦截(智能拦截)
27.              */
28.             if (body.contains("xue sheng mei")) {
29.                 System.out.println("学生妹被拦截了");
30.                 abortBroadcast();
31.             }
32.         }
33.     }
34. }

```

4.8.4 短信拦截记录界面的逻辑代码

前面我们已经创建好存放拦截短信的数据库，并且将拦截到的短信存放进去，还有用于短信界面显示的布局文件和 ListView 条目的布局文件，下面我们就要开始进行短信拦截记录界面的主要逻辑书写。

短信拦截界面 FirewallSmsFragment 的全部代码如下所示。

```

1  // 短信拦截界面
2  public class FirewallSmsFragment extends Fragment {
3      private ListView mListView;
4      private FirewallSmsDao dao;
5      private List<FirewallSmsInfo> mDatas;
6      @Override
7      public View onCreateView(LayoutInflater inflater, ViewGroup container,
8          Bundle savedInstanceState) {

```

```
9      // TextView textView = new TextView(getActivity());
10     // textView.setText("短信");
11     View view = inflater.inflate(R.layout.fragment_firewall_sms, null);
12     mListview = (ListView) view.findViewById(R.id.list_view);
13     return view;
14 }
15 @Override
16 public void onActivityCreated(Bundle savedInstanceState) {
17     // TODO Auto-generated method stub
18     super.onActivityCreated(savedInstanceState);
19     // 短信拦截的数据库
20     dao = new FirewallSmsDao(getActivity());
21 }
22 @Override
23 public void onStart() {
24     super.onStart();
25     System.out.println("onStart-----onStart");
26     initData();
27 }
28 private void initData() {
29     //获取到所有短信拦截的记录
30     mDatas = dao.findAll();
31     FirewallSmsAdapter adapter = new FirewallSmsAdapter();
32     mListview.setAdapter(adapter);
33 }
34 private class FirewallSmsAdapter extends BaseAdapter {
35     @Override
36     public int getCount() {
37         return mDatas.size();
38     }
39     @Override
40     public View getView(int position, View convertView, ViewGroup parent) {
41         View view = View.inflate(getActivity(), R.layout.item_fragment_firewall_sms,
42 null);
43         TextView tv_body = (TextView) view.findViewById(R.id.tv_body);
44         TextView tv_phone = (TextView) view.findViewById(R.id.tv_phone);
45         FirewallSmsInfo info = mDatas.get(position);
46         tv_body.setText(info.body);
47         tv_phone.setText(info.number);
48         return view;
49     }
50     @Override
51     public Object getItem(int position) {
52         return null;
```

```
53     }
54     @Override
55     public long getItemId(int position) {
56         return 0;
57     }
58 }
59 // 去掉重影
60 @Override
61 public void setMenuVisibility(boolean menuVisible) {
62     super.setMenuVisibility(menuVisible);
63     if (getView() != null) {
64         getView().setVisibility(menuVisible ? View.VISIBLE : View.INVISIBLE);
65     }
66 }
67 }
```

- 第 9~16 行是加载短信拦截界面的布局文件，返回 View 对象
- 第 31~36 行是初始化短信拦截的数据，从数据中查询所有数据，并创建数据适配器
- 第 37~64 行是用于展示短信拦截记录的数据适配器内容
- 第 69~75 行是去掉当前界面在切换时叠加造成的重影

上面的代码书写好之后，我们需要测验短信拦截记录界面是否能正确显示数据，在黑名界面添加几个黑名单号码同时设置其拦截模式，同时开启骚扰拦截的服务，如下图 4-35 所示，使用 DDMS 用黑名单号码模拟向模拟器发短信。



图 4-35 拦截短信测试

然后，我们从图 4-35 (b) 的黑名单界面直接切换到下图 4-36 (a) 时，发现该界面没有任何记录，点击返回到主界面，再次进入骚扰拦截界面时，发现短信拦截记录显示出来了，如图 4-36 (c) 所示。



图 4-36 拦截短信测试结果

分析可知，这是由于我们在 FirewallActivity.java 初始化三个 fragment 的时候是进行叠加显示的，如下所示。这样的话各自的状态就会储存起来，而切换时 fragment 并不会释放，也就造成 fragment 的生命周期 onStart()方法没有重新运行，界面的数据没有刷新。

```

1.      //初始化 fragment
2.      //第一个参数：填充的内容
3.      //第二个参数：表示索引
4.      Fragment fragment = (Fragment) fragments.instantiateItem(mFrameLayout,index);
5.      //设置 fragment 的默认值，并且调用去掉重影的 setMenuVisibility 方法
6.      fragments.setPrimaryItem(mFrameLayout, 0, fragment);
7.      //提交更新
8.      fragments.finishUpdate(mFrameLayout);
    
```

为了解决该问题，我们需要使用另外一种 fragment 的方式，如下所示。

```

1.      // 设置状态监听
2.      mRadioGroup.setOnCheckedChangeListener(new OnCheckedChangeListener() {
3.          Fragment fragment = null;
4.          @Override
5.          public void onCheckedChanged(RadioGroup group, int checkedId) {
6.              switch (checkedId) {
7.                  // 短信
8.                  case R.id.rb_sms:
9.                      index = 0;
10.                     fragment = new FirewallSmsFragment();
11.                     break;
12.                  // 电话
13.                  case R.id.rb_phone:
14.                      index = 1;
    
```

```

15.         fragment = new FirewallPhoneFragment();
16.         break;
17.         // 黑名单
18.         case R.id.rb_black:
19.             index = 2;
20.             fragment = new FirewallBlackFragment();
21.             break;
22.     }
23.
24.     //使用这种方式，fragment 的生命周期方法会重新运行
25.     getSupportFragmentManager().beginTransaction().replace(R.id.content,
26. fragment).commit();
27.     //如果通过这种方式不会释放 fragment.那么 fragment 的生命就不会重新在走
28.     //初始化 fragment
29.     //第一个参数：填充的内容
30.     //第二个参数：表示索引
31.     //Fragment fragment = (Fragment) fragments.instantiateItem(mFrameLayout,
32. index);
33.     //设置 fragment 的默认值，并且调用去掉重影的方法
34.     //fragments.setPrimaryItem(mFrameLayout, 0, fragment);
35.     //提交更新
36.     //fragments.finishUpdate(mFrameLayout);
37.     }
38.     });
    
```

修改好上面的代码之后，我们继续添加黑名单号码，模拟发送短信，直接从黑名单界面切换到短信界面，效果图如下图 4-37 所示。

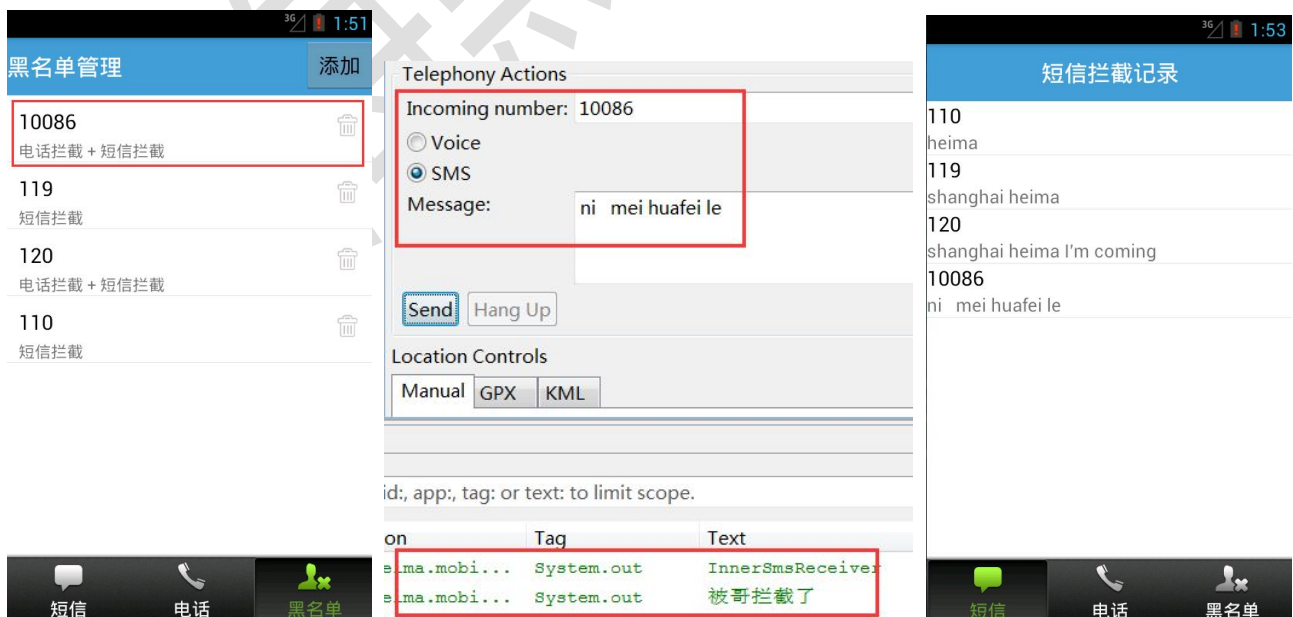


图 4-37 拦截短信效果图

根据上图可知，修改好之后的代码，在黑名单界面与短信界面的正常切换中界面的数据都正确地显示出来了，证明 `fragment` 的生命周期方法已经重走，达到我们想要的效果。

针对后面的电话拦截记录界面的代码逻辑，其实与短信拦截记录的逻辑相同，这里就不详细书写，留给读者自行编写。

4.9 本章小结

本章主要针对骚扰拦截模块进行讲解，首先针对该模块功能进行介绍，然后从创建数据库、创建 UI、编写主界面逻辑代码、编写添加黑名单界面逻辑代码到拦截电话和短信进行了详细讲解。该模块功能复杂，尤其是挂断电话操作使用了 AIDL 进程间通信，编程者要加强学习，并要动手完成所有代码，为学习后面模块的奠定基础。