第8章 数据存储与访问

学习目标

AIMS

01 掌握SharedPreferences的使用方法

02 掌握各种文件存储的区别与适用情况

03 了解SQLite数据库的特点和体系结构

04 掌握SQLite数据库的建立和操作方法

05 理解ContentProvider的用途和原理

06 掌握ContentProvider的创建与使用方法

8.1 简单存储

- **8.1.1 SharedPreferences**
  - SharedPreferences是一种轻量级的数据保存方式
  - 通过SharedPreferences开发人员可以将NVP（Name/Value Pair，名称/值对）保存在Android的文件系统中，而且SharedPreferences完全屏蔽了对文件系统的操作过程
  - 开发人员仅通过调用SharedPreferences中的函数就可以实现对NVP的保存和读取

8.1 简单存储

- **8.1.1 SharedPreferences**
  - SharedPreferences不仅能够保存数据，还能够实现不同应用程序间的数据共享
  - SharedPreferences支持三种访问模式
    - 私有（MODE_PRIVATE）：仅创建SharedPreferences的程序有权限对其进行读取或写入
    - 全局读（MODE_WORLD_READABLE）：不仅创建程序可以对其进行读取或写入，其它应用程序也具有读取操作的权限，但没有写入操作的权限
    - 全局写（MODE_WORLD_WRITEABLE）：所有程序都可以对其进行写入操作，但没有读取操作的权限

8.1 简单存储

- **8.1.1 SharedPreferences**
  - 首先， 定义SharedPreferences的访问模式，下面的代码将访问模式定义为私有模式
    - public static int MODE = MODE_PRIVATE;
  - 有的时候需要将SharedPreferences的访问模式设定为即可以全局读，也可以全局写，这就需要将两种模式写成下面的方式
  - public static int MODE = Context. MODE_PRIVATE + Context. MODE_PRIVATE;

8.1 简单存储

- **8.1.1 SharedPreferences**
  - 除了定义SharedPreferences的访问模式，还要定义SharedPreferences的名称，这个名称也是SharedPreferences在Android文件系统中保存的文件名称
  - 一般将SharedPreferences名称声明为字符串常量，这样可以在代码中多次使用
    - 1 public static final String PREFERENCE_NAME = "SaveSetting";
  - 使用SharedPreferences时需要将访问模式和SharedPreferences名称作为参数传递到getSharedPreferences()函数，则可获取到SharedPreferences实例
    - **1 SharedPreferences sharedPreferences = getSharedPreferences(PREFERENCE_NAME, MODE);**

8.1 简单存储

- **8.1.1 SharedPreferences**
  - 在获取到SharedPreferences实例后，可以通过SharedPreferences.Editor类对SharedPreferences进行修改，最后调用commit()函数保存修改内容
  - SharedPreferences广泛支持各种基本数据类型，包括整型、布尔型、浮点型和长型等

```
1  SharedPreferences.Editor editor = sharedPreferences.edit();
2  editor.putString("Name", "Tom");
3  editor.putInt("Age", 20);
4  editor.putFloat("Height", 1.81f);
5  editor.commit();
```

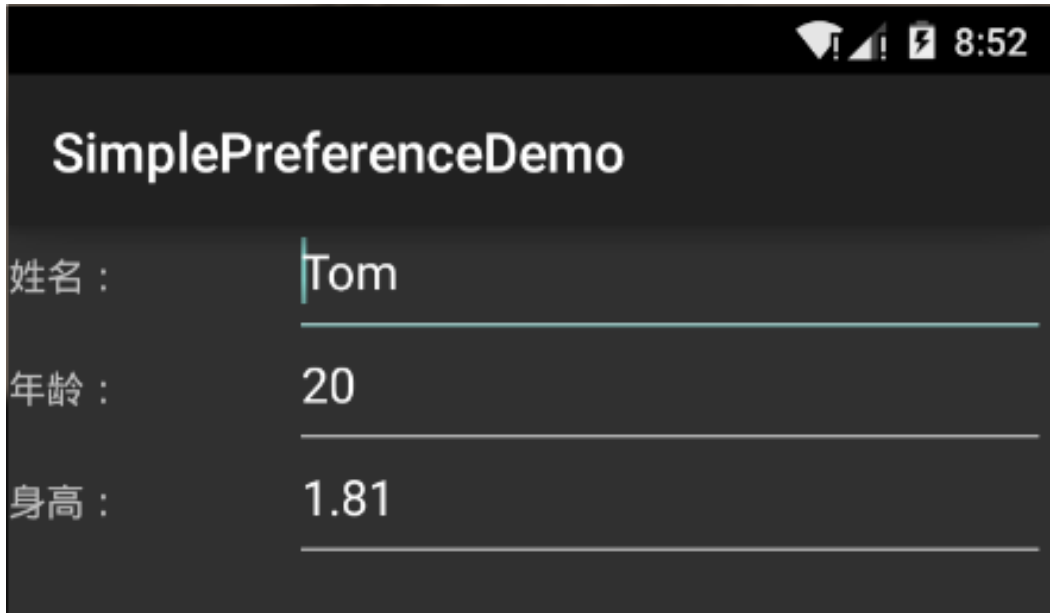
8.1 简单存储

- **8.1.1 SharedPreferences**
  - 如果需要从已经保存的SharedPreferences中读取数据，同样是调用getSharedPreferences()函数，并在函数第1个参数中指明需要访问的SharedPreferences名称，最后通过get<Type>()函数获取保存在SharedPreferences中的NVP
  - get<Type>()函数的第1个参数是NVP的名称
  - 第2个参数是在无法获取到数值的时候使用的缺省值

```
1  SharedPreferences sharedPreferences =
     getSharedPreferences(PREFERENCE_NAME, MODE);
2  String name = sharedPreferences.getString("Name","Default Name");
3  int age = sharedPreferences.getInt("Age", 20);
4  float height = sharedPreferences.getFloat("Height",1.81f);
```

8.1 简单存储

- **8.1.2 示例**
  - 下面将通过SimplePreferenceDemo示例介绍SharedPreferences的文件保存位置和保存格式
  - 下图是SimplePreferenceDemo示例的用户界面

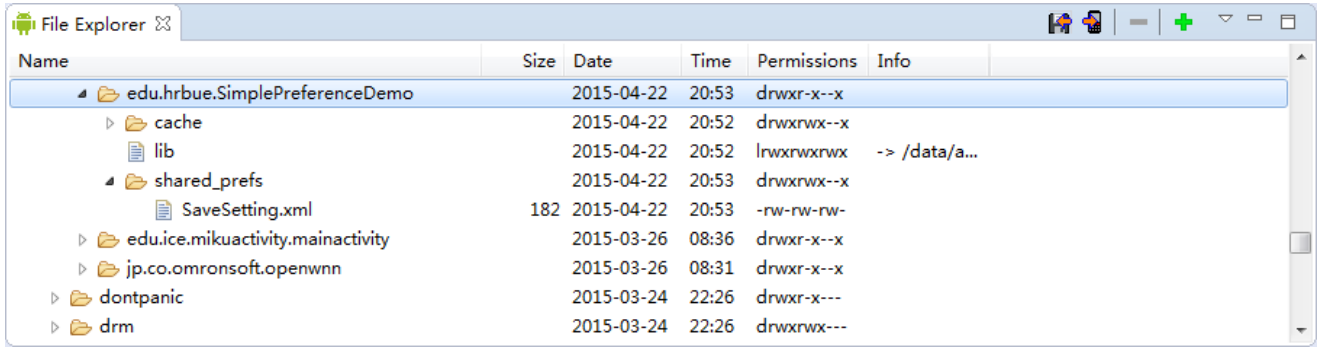


8.1 简单存储

- **8.1.2 示例**
  - 用户在界面上的输入信息，在Activity关闭时通过SharedPreferences进行保存。当应用程序重新开启时，再通过SharedPreferences将信息读取出来，并重新呈现在用户界面上
  - SimplePreferenceDemo示例运行并通过“回退键”退出，通过FileExplorer查看/data/data下的数据，Android系统为每个应用程序建立了与包同名的目录，用来保存应用程序产生的数据文件，包括普通文件、SharedPreferences文件和数据库文件等
  - SharedPreferences产生的文件就保存在/data/data/<package name>/shared_prefs目录下

8.1 简单存储

- **8.1.2 示例**
  - 在本示例中，shared_prefs目录中生成了一个名为SaveSetting.xml的文件
  - 如图8.2所示，保存在/data/data/edu.hrbeu.SimplePreferenceDemo/shared_prefs目录下
  - 这个文件就是保存SharedPreferences的文件，文件大小为170字节，在Linux下的权限为“-rw-rw-rw”



8.1 简单存储

- **8.1.2 示例**
  - 在Linux系统中，文件权限分别描述了创建者、同组用户和其它用户对文件的操作限制。x表示可执行，r表示可读，w表示可写，d表示目录，-表示普通文件
  - 因此，“-rw-rw-rw”表示SaveSetting.xml可以被创建者、同组用户和其它用户进行读取和写入操作，但不可执行
  - 产生这样的文件权限与程序人员设定的SharedPreferences的访问模式有关，“-rw-rw-rw”的权限是“全局读+全局写”的结果
  - 如果将SharedPreferences的访问模式设置为私有，则文件权限将成为“-rw-rw ---”，表示仅有创建者和同组用户具有读写文件的权限

8.1 简单存储

- **8.1.2 示例**
  - SaveSetting.xml文件是以XML格式保存的信息，内容如下：

```
1  <?xml version='1.0' encoding='utf-8' standalone='yes' ?>
2  <map>
3          <float name="Height" value="1.81" />
4          <string name="Name">Tom</string>
5          <int name="Age" value="20" />
6  </map>
```

8.1 简单存储

- **8.1.2 示例**
  - SimplePreferenceDemo示例在onStart()函数中调用loadSharedPreferences()函数，读取保存在SharedPreferences中的姓名、年龄和身高信息，并显示在用户界面上
  - 当Activity关闭时，在onStop()函数调用saveSharedPreferences()，保存界面上的信息
  - SimplePreferenceDemoActivity.java的完整代码如下：

```
1  package edu.hrbeu.SimplePreferenceDemo;
2
3  import android.app.Activity;
4  import android.content.Context;
5  import android.content.SharedPreferences;
6  import android.os.Bundle;
```

8.1 简单存储

- **8.1.2 示例**

```
7   import android.widget.EditText;
8
9   public class SimplePreferenceDemoActivity extends Activity {
10
11          private EditText nameText;
12          private EditText ageText;
13          private EditText heightText;
14          public static final String PREFERENCE_NAME = "SaveSetting";
15          public static int MODE = Context.MODE_WORLD_READABLE +
        Context.MODE_WORLD_WRITEABLE;
16
17          @Override
18          public void onCreate(Bundle savedInstanceState) {
19            super.onCreate(savedInstanceState);
20            setContentView(R.layout.main);
21            nameText = (EditText)findViewById(R.id.name);
```

8.1 简单存储

- **8.1.2 示例**

```
22          ageText = (EditText)findViewById(R.id.age);
23          heightText = (EditText)findViewById(R.id.height);
24           }
25
26           @Override
27           public void onStart(){
28          super.onStart();
29          loadSharedPreferences();
30           }
31           @Override
32           public void onStop(){
33          super.onStop();
34          saveSharedPreferences();
35           }
36
```

8.1 简单存储

- **8.1.2 示例**

```
37          private void loadSharedPreferences(){
38          SharedPreferences sharedPreferences = getSharedPreferences(PREFERENCE_NAME,
    MODE);
39          String name = sharedPreferences.getString("Name","Tom");
40          int age = sharedPreferences.getInt("Age", 20);
41          float height = sharedPreferences.getFloat("Height",1.81f);
42
43          nameText.setText(name);
44          ageText.setText(String.valueOf(age));
45          heightText.setText(String.valueOf(height));
46           }
47
48          private void saveSharedPreferences(){
49          SharedPreferences sharedPreferences = getSharedPreferences(PREFERENCE_NAME,
    MODE);
50          SharedPreferences.Editor editor = sharedPreferences.edit();
```

8.1 简单存储

- **8.1.2 示例**

```
51
52          editor.putString("Name", nameText.getText().toString());
53          editor.putInt("Age", Integer.parseInt(ageText.getText().toString()));
54          editor.putFloat("Height", Float.parseFloat(heightText.getText().toString()));
55          editor.commit();
56           }
57    }
```

8.1 文件存储

- 虽然SharedPreferences能够为开发人员简化数据存储和访问过程，但直接使用文件系统保存数据仍然是Android数据存储中不可或缺的组成部分
- Android使用Linux的文件系统，开发人员可以建立和访问程序自身建立的私有文件，也可以访问保存在资源目录中的原始文件和XML文件，还可以将文件保存在TF卡等外部存储设备中

8.2 文件存储

- **8.2.1 内部存储**
  - Android系统允许应用程序创建仅能够自身访问的私有文件，文件保存在设备的内部存储器上，在Android系统下的/data/data/<package name>/files目录中
  - Android系统不仅支持标准Java的IO类和方法，还提供了能够简化读写流式文件过程的函数
  - 这里主要介绍两个函数
    - openFileOutput()
    - openFileInput()

8.2 文件存储

• 8.2.1 内部存储

- openFileOutput()函数
  - openFileOutput()函数为写入数据做准备而打开文件
  - 如果指定的文件存在，直接打开文件准备写入数据
  - 如果指定的文件不存在，则创建一个新的文件
  - openFileOutput()函数的语法格式如下：
    - public FileOutputStream openFileOutput(String name, int mode)
    - 第1个参数是文件名称，这个参数不可以包含描述路径的斜杠
    - 第2个参数是操作模式，Android系统支持四种文件操作模式
  - 函数的返回值是FileOutputStream类型

8.2 文件存储

- **8.2.1 内部存储**
  - openFileOutput()函数
    - 两种文件操作模式

模式	说明
MODE_PRIVATE	私有模式，缺陷模式，文件仅能够被创建文件的程序访问，或具有相同UID的程序访问。
MODE_APPEND	追加模式，如果文件已经存在，则在文件的结尾处添加新数据。

8.2 文件存储

8.2.1 内部存储

- openFileOutput()函数
  - 使用openFileOutput()函数建立新文件的示例代码如下:

```
1  String FILE_NAME = "fileDemo.txt";
2  FileOutputStream fos = openFileOutput(FILE_NAME,Context.MODE_PRIVATE)
3  String text = “Some data”;
4  fos.write(text.getBytes());
5  fos.flush();
6  fos.close();
```

  - 代码首先定义文件的名称为fileDemo.txt
  - 然后使用openFileOutput()函数以私有模式建立文件，并调用write()函数将数据写入文件，调用flush()函数将缓冲中的数据写入文件，最后调用close()函数关闭FileOutputStream

8.2 文件存储

- **8.2.1 内部存储**
  - openFileOutput()函数
    - 为了提高文件系统的性能，一般调用write()函数时，如果写入的数据量较小，系统会把数据保存在数据缓冲区中，等数据量积攒到一定程度时再将数据一次性写入文件
    - 因此，在调用close()函数关闭文件前，务必要调用flush()函数，将缓冲区内所有的数据写入文件
    - 如果开发人员在调用close()函数前没有调用flush()，则可能导致部分数据丢失

8.2 文件存储

- **8.2.1 内部存储**
  - openFileInput()函数
    - openFileInput()函数为读取数据做准备而打开文件
    - openFileInput()函数的语法格式如下：
      - public FileInputStream openFileInput (String name)
      - 第1个参数也是文件名称，同样不允许包含描述路径的斜杠
      - 使用openFileInput()函数打开已有文件，并以二进制方式读取数据的示例代码如下：

```
1  String FILE_NAME = "fileDemo.txt";
2  FileInputStream fis = openFileInput(FILE_NAME);
3
4  byte[] readBytes = new byte[fis.available()];
5  while(fis.read(readBytes) != -1){
6  }
```

8.2 文件存储

- **8.2.1 内部存储**
  - openFileInput()函数
    - 上面的两部分代码在实际使用过程中会遇到错误提示，因为文件操作可能会遇到各种问题而最终导致操作失败，因此代码应该使用try/catch捕获可能产生的异常

8.2 文件存储

- **8.2.1 内部存储**
  - InternalFileDemo是用来演示在内部存储器上进行文件写入和读取的示例。
  - 用户界面如下图所示，用户将需要写入的数据添加在EditText中，通过“写入文件”按钮将数据写入到/data/data/edu.hrbeu.InternalFileDemo/files/fileDemo.txt文件中
  - 如果用户选择“追加模式”，数据将会添加到fileDemo.txt文件的结尾处
  - 通过“读取文件”按钮，程序会读取fileDemo.txt文件的内容，并显示在界面下方的白色区域中

8.2 文件存储

■ 8.2.1 内部存储

- InternalFileDemo用户界面图

8.2 文件存储

• 8.2.1 内部存储

- InternalFileDemo示例的核心代码:

```
1   OnClickListener writeButtonListener = new OnClickListener() {
2           @Override
3           public void onClick(View v) {
4           FileOutputStream fos = null;
5           try {
6               if (appendBox.isChecked()){
7                   fos = openFileOutput(FILE_NAME,Context.MODE_APPEND);
8               }else {
9                   fos = openFileOutput(FILE_NAME,Context.MODE_PRIVATE);
10              }
11          String text = entryText.getText().toString();
12          fos.write(text.getBytes());
```

8.2 文件存储

- **8.2.1 内部存储**

```
13          labelView.setText("文件写入成功，写入长度："+text.length());
14          entryText.setText("");
15          } catch (FileNotFoundException e) {
16                  e.printStackTrace();
17          }
18          catch (IOException e) {
19                  e.printStackTrace();
20          }
21          finally{
22                  if (fos != null){
23      try {
24                fos.flush();
25              fos.close();
26      } catch (IOException e) {
27              e.printStackTrace();
```

8.2 文件存储

- **8.2.1 内部存储**

```
28  }
29              }
30      }
31          }
32   };
33   OnClickListener readButtonListener = new OnClickListener() {
34          @Override
35          public void onClick(View v) {
36       displayView.setText("");
37       FileInputStream fis = null;
38       try {
39                          fis = openFileInput(FILE_NAME);
40              if (fis.available() == 0){
41  return;
42                  }
```

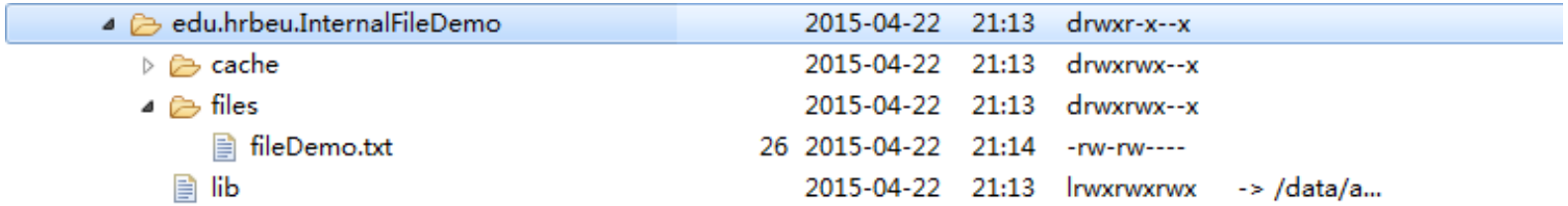
8.2 文件存储

- **8.2.1 内部存储**

```
43                  byte[] readBytes = new byte[fis.available()];
44              while(fis.read(readBytes) != -1){
45              }
46              String text = new String(readBytes);
47              displayView.setText(text);
48              labelView.setText("文件读取成功，文件长度："+text.length());
49          } catch (FileNotFoundException e) {
50              e.printStackTrace();
51          }
52          catch (IOException e) {
53              e.printStackTrace();
54          }
55           }
56      };
```

8.2 文件存储

- **8.2.1 内部存储**
  - 程序运行后，在/data/data/edu.hrbeu.InternalFileDemo/files/目录下，找到了新建立的fileDemo.txt文件，下图所示
  - 从文件权限上进行分析fileDemo.txt文件，“-rw-rw---”表明文件仅允许文件创建者和同组用户读写，其它用户无权使用
  - 文件的大小为9个字节，保存的数据为“Some data”



8.2 文件存储

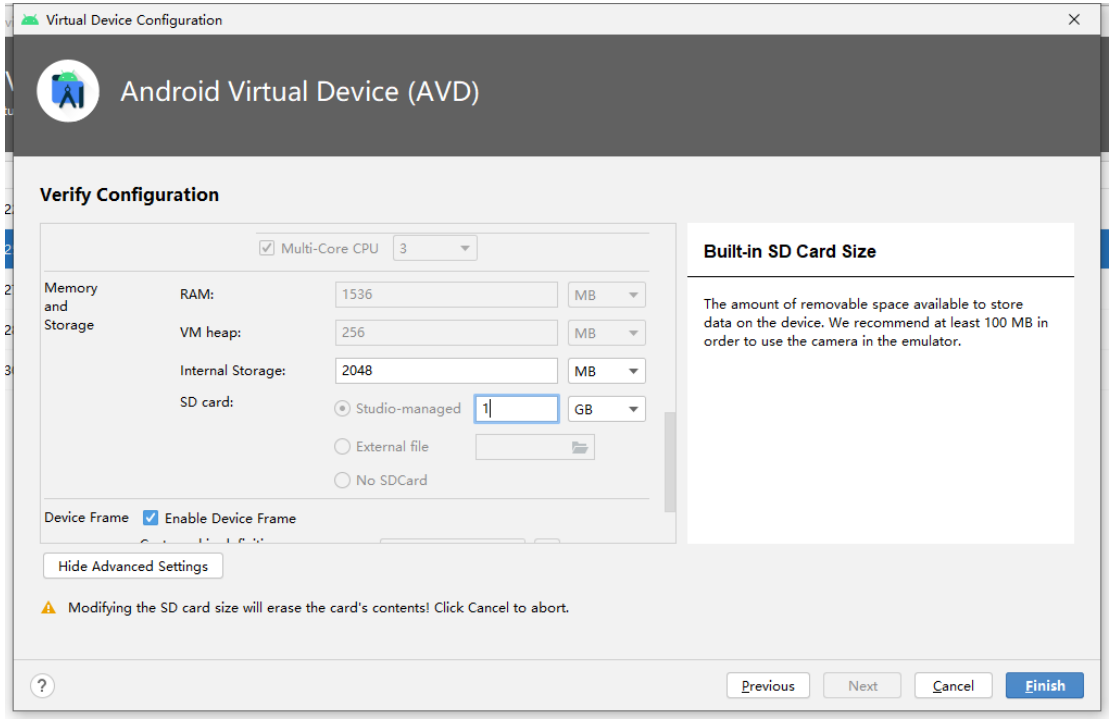
- **8.2.2 外部存储**
  - Android的外部存储设备一般指Micro SD卡，又称T-Flash，是一种广泛使用于数码设备的超小型记忆卡
  - 下图是东芝出品的32G Micro SD卡

8.2 文件存储

- **8.2.2 外部存储**
  - Micro SD卡适用于保存大尺寸的文件或者是一些无需设置访问权限的文件
  - 如果用户希望保存录制的视频文件和音频文件，因为Android设备的内部存储空间有限，所以使用Micro SD卡则是非常适合的选择
  - 但如果需要设置文件的访问权限，则不能够使用Micro SD卡，因为Micro SD卡使用FAT（File Allocation Table）文件系统，不支持访问模式和权限控制
  - Android的内部存储器使用的是Linux文件系统，则可通过文件访问权限的控制保证文件的私密性

8.2 文件存储

- **8.2.2 外部存储**
  - Android模拟器支持SD卡的模拟，在模拟器建立时可以选择SD卡的容量，如下图所示，在模拟器启动时会自动加载SD卡



8.2 文件存储

- **8.2.2 外部存储**
  - 正确加载SD卡后，SD卡中的目录和文件被映射到/mnt/sdcard目录下
  - 因为用户可以加载或卸载SD卡，所以在编程访问SD卡前首先需要检测/mnt/sdcard目录是否可用
  - 如果不可用，说明设备中的SD卡已经被卸载。如果可用，则直接通过使用标准的java.io.File类进行访问
  - SDcardFileDemo示例用来说明如何将数据保存在SD卡中
  - 首先通过“生产随机数列”按钮生产10个随机小数，然后通过“写入SD卡”按钮将生产的数据保存在SD卡的根目录下，也就是Android系统的/mnt/sdcard目录下

8.2 文件存储

- **8.2.2 外部存储**
  - SDcardFileDemo用户界面图

8.2 文件存储

- **8.2.2 外部存储**
  - SDcardFileDemo示例运行后，在每次点击“写入SD卡”按钮后，都会在SD卡中生产一个新文件，文件名各不相同，如下图所示:

名称	修改日期	类型	大小
data	2021/3/26 14:31	文件夹	
hardware-qemu.ini.lock	2021/4/11 21:02	文件夹	
snapshots	2021/3/26 14:31	文件夹	
AVD.conf	2021/4/11 21:03	CONF 文件	1 KB
cache	2021/3/26 14:32	光盘映像文件	67,584 KB
cache.img.qcow2	2021/4/11 21:00	QCOW2 文件	9,984 KB
config	2021/3/26 14:31	配置设置	2 KB
emulator-user	2021/4/11 21:00	配置设置	1 KB
emu-launch-params	2021/4/11 21:02	文本文档	1 KB
hardware-qemu	2021/4/11 21:02	配置设置	4 KB
multiinstance.lock	2021/4/11 21:02	LOCK 文件	0 KB
quickbootChoice	2021/4/11 21:00	配置设置	1 KB
read-snapshot	2021/4/11 21:02	文本文档	0 KB
sdcard	2021/3/26 14:31	光盘映像文件	524,288 KB
sdcard.img.qcow2	2021/4/11 21:00	QCOW2 文件	577 KB
userdata	2021/3/26 14:26	光盘映像文件	563,200 KB
userdata-qemu	2021/3/26 14:32	光盘映像文件	819,200 KB
userdata-qemu.img.qcow2	2021/4/11 21:00	QCOW2 文件	90,624 KB
version_num.cache	2021/3/26 14:32	CACHE 文件	1 KB

8.2 文件存储

• 8.2.2 外部存储

- SDcardFileDemo示例与InternalFileDemo示例的核心代码比较相似，不同之处在于代码中添加了/mnt/sdcard目录存在性检查（代码第7行），并使用“绝对目录+文件名”的形式表示新建立的文件（代码第8行），并在写入文件前对文件的存在性和可写入性进行检查(代码第12行)
- 为了保证在SD卡中多次写入时文件名不会重复，在文件名中使用了唯一且不重复的标识（代码第5行），这个标识通过调用System.currentTimeMillis()函数获得，表示从1970年00:00:00到当前所经过的毫秒数
- SDcardFileDemo示例的核心代码如下:

8.2 文件存储

- **8.2.2 外部存储**

```
1   private static String randomNumbersString = "";
2   OnClickListener writeButtonListener = new OnClickListener() {
3           @Override
4           public void onClick(View v) {
5           String fileName = "SdcardFile-"+System.currentTimeMillis()+".txt";
6           File dir = new File("/sdcard/");
7           if (dir.exists() && dir.canWrite()) {
8                   File newFile = new File(dir.getAbsolutePath() + "/" + fileName);
9                   FileOutputStream fos  = null;
10                  try {
11      newFile.createNewFile();
12      if (newFile.exists() && newFile.canWrite()) {
13              fos  = new FileOutputStream(newFile);
14               fos.write(randomNumbersString.getBytes());
15             TextView labelView = (TextView)findViewById(R.id.label);
16             labelView.setText(fileName + "文件写入SD卡");
```

8.2 文件存储

- **8.2.2 外部存储**

```
17      }
18              } catch (IOException e) {
19      e.printStackTrace();
20              } finally {
21      if (fos != null) {
22              try{
23                  fos.flush();
24                  fos.close();
25              }
26              catch (IOException e) { }
27          }
28              }
29          }
30           }
31  };
```

8.2 文件存储

• 8.2.2 外部存储

- 程序在模拟器中运行前，还必须在AndroidManifest.xml中注册两个用户权限，分别是加载卸载文件系统的权限和向外部存储器写入数据的权限
- AndroidManifest.xml的核心代码如下:

```
1    <uses-permission android:name="android.permission.MOUNT_UNMOUNT_FILESYSTEMS">
     </uses-permission>
2    <uses-permission android:name="android.permission.WRITE_EXTERNAL_STORAGE">
     </uses-permission>
```

8.2 文件存储

- **8.2.3 资源文件**
  - 开发人员除了可以在内部和外部存储设备上读写文件以外，还可以访问在/res/raw和/res/xml目录中的原始格式文件和XML文件，这些文件是程序开发阶段在工程中保存的文件
  - 原始格式文件可以是任何格式的文件，例如视频格式文件、音频格式文件、图像文件或数据文件等等
  - 在应用程序编译和打包时，/res/raw目录下的所有文件都会保留原有格式不变。而/res/xml目录下一般用来保存格式化数据的XML文件，则会在编译和打包时将XML文件转换为二进制格式，用以降低存储器空间占用和提高访问效率，在应用程序运行的时候会以特殊的方式进行访问

8.2 文件存储

- **8.2.3 资源文件**
  - ResourceFileDemo示例演示了如何在程序运行时访问资源文件
  - 当用户点击“读取原始文件”按钮时，程序将读取/res/raw/raw_file.txt文件，并将内容显示在界面上，如下图所示:

8.2 文件存储

- **8.2.3 资源文件**
  - 当用户点击“读取XML文件”按钮时，程序将读取/res/xml/people.xml文件，也将内容显示在界面上，如下图所示:

8.2 文件存储

• 8.2.3 资源文件

- 读取原始格式文件首先需要调用getResource()函数获得资源实例，然后通过调用资源实例的openRawResource()函数，以二进制流的形式打开指定的原始格式文件。在读取文件结束后，调用close()函数关闭文件流
- ResourceFileDemo示例中读取原始格式文件的核心代码如下:

```
1   Resources resources = this.getResources();
2   InputStream inputStream = null;
3   try {
4           inputStream = resources.openRawResource(R.raw.raw_file);
5           byte[] reader = new byte[inputStream.available()];
6           while (inputStream.read(reader) != -1) {
```

8.2 文件存储

- **8.2.3 资源文件**

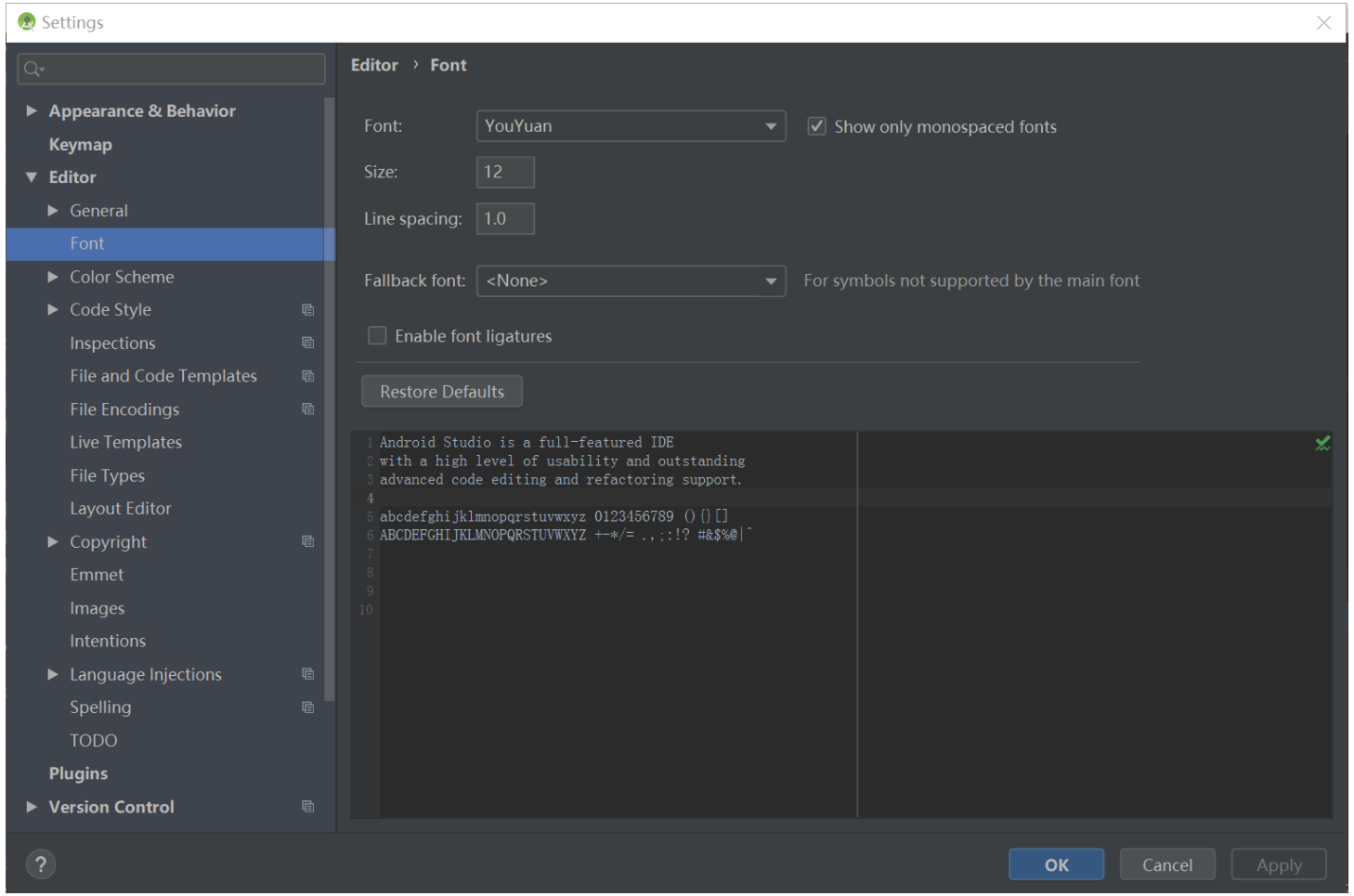
```
7           }
8           displayView.setText(new String(reader,"utf-8"));
9       } catch (IOException e) {
10          Log.e("ResourceFileDemo", e.getMessage(), e);
11      } finally {
12          if (inputStream != null) {
13          try {
14          inputStream.close();
15          }
16          catch (IOException e) { }
17          }
18      }
```

8.2 文件存储

- **8.2.3 资源文件**
  - 代码第8行的new String(reader,"utf-8")，表示以UTF-8的编码方式从字节数组中实例化一个字符串
  - 如果程序开发人员需要新建/res/raw/raw_file.txt文件，则需要选择使用UTF-8编码方式，否则程序运行时会产生乱码
  - 选择的方法是在raw_file.txt文件上点击右键，选择“Properties”打开raw_file.txt文件的属性设置框，然后在“Resource”栏下的“Text file encoding”中，选择“Other：UTF-8”，如下图所示:

8.2 文件存储

- **8.2.3 资源文件**
  - 选择raw_file.txt文件编码方式



8.2 文件存储

- **8.2.3 资源文件**
  - /res/xml目录下的XML文件与其它资源文件有所不同，程序开发人员不能够以流的方式直接读取，其主要原因在于Android系统为了提高读取效率，减少占用的存储空间，将XML文件转换为一种高效的二进制格式

8.2 文件存储

- **8.2.3 资源文件**
  - 如何在程序运行时读取/res/xml目录下的XML文件
    - 首先在/res/xml目录下创建一个名为people.xml的文件
    - XML文件定义了多个<person>元素，每个<person>元素都包含三个属性name、age和height，分别表示姓名、年龄和身高
  - /res/xml/people.xml文件代码如下:

```
1  <people>
2      <person name="李某某" age="21" height="1.81" />
3      <person name="王某某" age="25" height="1.76" />
4      <person name="张某某" age="20" height="1.69" />
5  </people>
```

8.2 文件存储

- **8.2.3 资源文件**
  - 读取XML格式文件
    - 首先通过调用资源实例的getXml()函数，获取到XML解析器XmlPullParser
    - XmlPullParser是Android平台标准的XML解析器，这项技术来自一个开源的XML解析API项目XMLPULL
  - ResourceFileDemo示例中关于读取XML文件的核心代码如下:

```
1   XmlPullParser parser = resources.getXml(R.xml.people);
2   String msg = "";
3   try {
4           while (parser.next() != XmlPullParser.END_DOCUMENT) {
5       String people = parser.getName();
6       String name = null;
```

8.2 文件存储

- **8.2.3 资源文件**
  - 读取XML格式文件

```
7           String age = null;
8           String height = null;
9           if ((people != null) && people.equals("person")) {
10                  int count = parser.getAttributeCount();
11                  for (int i = 0; i < count; i++) {
12      String attrName = parser.getAttributeName(i);
13      String attrValue = parser.getAttributeValue(i);
14              if ((attrName != null) && attrName.equals("name")) {
15      name = attrValue;
16              } else if ((attrName != null) && attrName.equals("age")) {
17      age = attrValue;
18              } else if ((attrName != null) && attrName.equals("height")) {
19      height = attrValue;
20      }
21          }
```

8.2 文件存储

- **8.2.3 资源文件**
  - 读取XML格式文件

```
22            if ((name != null) && (age != null) && (height != null)) {
23            msg += "姓名："+name+"，年龄："+age+"，身高："+height+"\n";
24             }
25           }
26       }
27    } catch (Exception e) {
28          Log.e("ResourceFileDemo", e.getMessage(), e);
29    }
30    displayView.setText(msg);
```

8.2 文件存储

• 8.2.3 资源文件

- 读取XML格式文件
  - 代码第1行通过资源实例的getXml()函数获取到XML解析器
  - 第4行的parser.next()方法可以获取到高等级的解析事件，并通过对比确定事件类型，XML事件类型参考下表

事件类型	说明
START_TAG	读取到标签开始标志
TEXT	读取文本内容
END_TAG	读取到标签结束标志
END_DOCUMENT	文档末尾

8.2 文件存储

- **8.2.3 资源文件**
  - 读取XML格式文件
    - 第5行使用getName()函数获得元素的名称
    - 第10行使用getAttributeCount()函数获取元素的属性数量
    - 第12行通过getAttributeName()函数得到属性名称
    - 最后在第14行到第19行代码中，通过分析属性名获取到正确的属性值，并在第23行将属性值整理成需要显示的信息

8.3 数据库存储

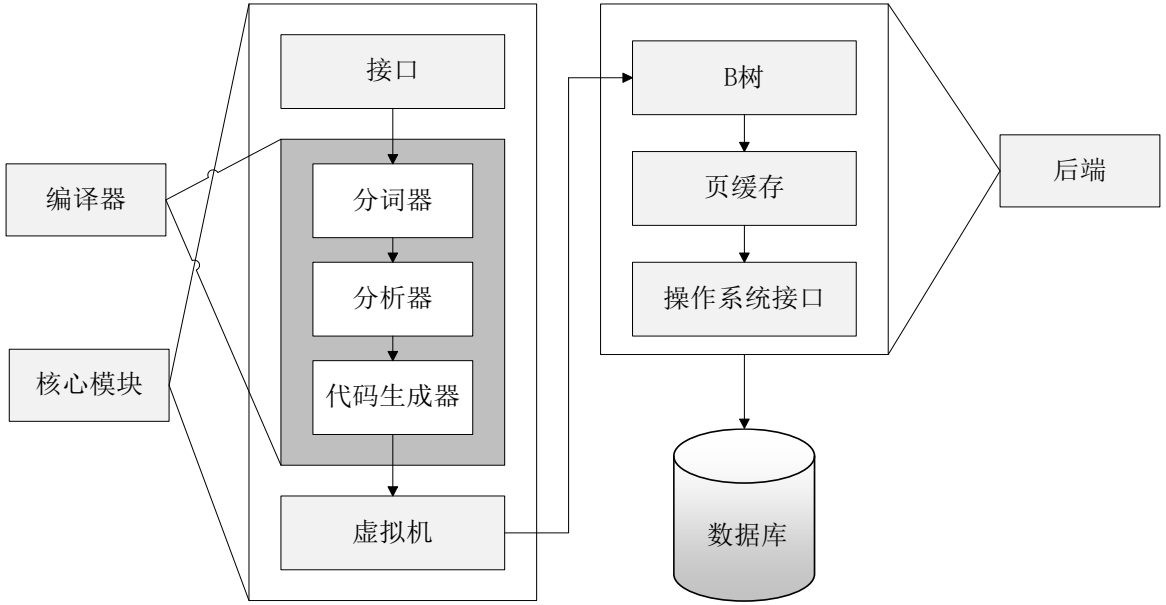
- **8.3.1 SQLite数据库**
  - SQLite是一个2000年由D.Richard Hipp发布的开源嵌入式关系数据库
  - 普通数据库的管理系统比较庞大和复杂，会占用了较多的系统资源
  - 轻量级数据库SQLite的特点
    - 比传统数据库更适合用于嵌入式系统
    - 占用资源少，运行高效可靠，可移植性强
    - 提供了零配置（zero-configuration）运行模式

8.3 数据库存储

- **8.3.1 SQLite数据库**
  - SQLite数据库的优势
    - 可以嵌入到使用它的应用程序中
      - 提高了运行效率
      - 屏蔽了数据库使用和管理的复杂性
    - 客户端和服务器在同一进程空间运行
      - 完全不需要进行网络配置和管理
      - 减少了网络调用所造成的额外开销
    - 简化了数据库的管理过程
      - 应用程序更加易于部署和使用
      - 只需要把SQLite数据库正确编译到应用程序中

8.3 数据库存储

- **8.3.1 SQLite数据库**
  - SQLite数据库采用了模块化设计，模块将复杂的查询过程分解为细小的工作进行处理
  - SQLite数据库由8个独立的模块构成，这些独立模块又构成了三个主要的子系统



8.3 数据库存储

- **8.3.1 SQLite数据库**
  - 接口
    - 由SQLite C API组成，因此无论是应用程序、脚本，还是库文件，最终都是通过接口与SQLite交互
  - 编译器
    - 在编译器中，分词器和分析器对SQL语句进行语法检查，然后把SQL语句转化为便于底层处理的分层数据结构，这种分层的数据结构称为“语法树”
    - 然后把语法树传给代码生成器进行处理，生成一种用于SQLite的汇编代码，最后由虚拟机执行

8.3 数据库存储

- **8.3.1 SQLite数据库**
  - 虚拟机
    - SQLite数据库体系结构中最核心的部分是虚拟机，也称为虚拟数据库引擎（Virtual Database Engine，VDBE）
    - 与Java虚拟机相似，虚拟数据库引擎用来解释并执行字节代码
    - 虚拟数据库引擎的字节代码由128个操作码构成，这些操作码主要用以对数据库进行操作，每一条指令都可以完成特定的数据库操作，或以特定的方式处理栈的内容

8.3 数据库存储

- **8.3.1 SQLite数据库**
  - 后端
    - 后端由B-树、页缓存和操作系统接口构成，B-树和页缓存共同对数据进行管理
    - B-树的主要功能就是索引，它维护着各个页面之间复杂的关系，便于快速找到所需数据
    - 页缓存的主要作用是通过操作系统接口在B-树和磁盘之间传递页面

8.3 数据库存储

- **8.3.1 SQLite数据库**
  - 移植性
    - 可以运行在Windows、Linux、BSD、Mac OS和一些商用Unix系统，比如Sun的Solaris或IBM的AIX
    - 嵌入式操作系统下，比如QNX、VxWorks、Palm OS、Symbin和Windows CE
  - SQLite的核心大约有3万行标准C代码，因为模块化的设计使这些代码非常易于理解

8.3 数据库存储

- **8.3.2 手动建库**
  - 在Android系统中，每个应用程序的SQLite数据库被保存在各自的/data/data/<package name>/databases目录下
  - 缺省情况下，所有数据库都是私有的，仅允许创建数据库的应用程序访问，如果需要共享数据库则可以使用ContentProvider
  - 虽然应用程序完全可以在代码中动态的建立SQLite数据库，但使用命令行手工建立和管理数据库仍然是非常重要的内容，对于调试使用数据库的应用程序非常有用

8.3 数据库存储

- **8.3.2 手动建库**
  - 手动建立数据库指的是使用sqlite3工具，通过手工输入命令行完成数据库的建立过程
  - sqlite3是SQLite数据库自带的一个基于命令行的SQL命令执行工具，并可以显示命令执行结果
  - Android SDK的tools目录有sqlite3工具，同时，该工具也被集成在Android系统中
  - 下面的内容将介绍如何连接到模拟器中的Linux系统，并在Linux系统中启动sqlite3工具，在Android程序目录中建立数据库和数据表，并使用命令在数据表中添加、修改和删除数据

8.3 数据库存储

• 8.3.2 手动建库

- 使用adb shell命令连接到模拟器的Linux系统，在Linux命令提示符下输入sqlite3可启动sqlite3工具
- 启动sqlite3后会显示SQLite的版本信息，显示内容如下:

```
1  # sqlite3
2  SQLite version 3.6.22
3  Enter ".help" for instructions
4  Enter SQL statements terminated with a ";"
5  sqlite>
```

8.3 数据库存储

- **8.3.2 手动建库**
  - 在启动sqlite3工具后，提示符从“#”变为“sqlite>”，表示用户进入SQLite数据库交互模式，此时可以输入命令建立、删除或修改数据库的内容
  - 正确退出sqlite3工具的方法是使用.exit命令

```
1  sqlite> .exit
2  #
```

8.3 数据库存储

- **8.3.2 手动建库**
  - 原则上，每个应用程序的数据库都保存在各自的/data/data/<package name>/databases目录下
  - 但如果使用手工方式建立数据库，则必须手工建立数据库目录，目前版本无需修改数据库目录的权限

```
1   # mkdir databases
2     # ls –l
3     drwxrwxrwx root     root              2011-09-19 15:43 databases
4     drwxr-xr-x system   system            2011-09-19 15:31 lib
5     #
```

8.3 数据库存储

- **8.3.2 手动建库**
  - 在SQLite数据库中，每个数据库保存在一个独立的文件中
  - 使用“sqlite3+文件名”的方式打开数据库文件，如果指定的文件不存在，sqlite3工具则自动创建新文件
  - 下面的代码将创建名为people的数据库，在文件系统中将产生一个名为people.db的数据库文件

```
6     # sqlite3 people.db
7     SQLite version 3.6.22
8     Enter “.help” for instructions
9     Enter SQL statements terminated with a “;”
10    sqlite>
```

8.3 数据库存储

• 8.3.2 手动建库

- 下面的代码在数据库中，构造了一个名为peopleinfo的表
- 使用create table命令，关系模式为peopleinfo (_id, name, age, height)
- 表包含四个属性，_id是整型主键；name表示姓名，字符型，not null表示属性值一定要填写，不可以为空值；age表示年龄，整数型；height表示身高，浮点型

```
1    sqlite> create table peopleinfo
2    ...> (_id integer primary key autoincrement,
3    ...> name text not null,
4    ...> age integer,
5    ...> height float);
6    sqlite>
```

8.3 数据库存储

- **8.3.2 手动建库**
  - 为了确认数据表是否创建成功，可以使用.tables命令，显示当前数据库中的所有表
  - 从下面的代码中可以观察到，当前的数据库中仅有一个名为peopleinfo的表

```
1    sqlite> .tables
2    poepleinfo
3    sqlite>
```

8.3 数据库存储

- **8.3.2 手动建库**
  - 也可以使用.schema命令查看建立表时使用的SQL命令
  - 如果当前数据库中包含多个表，则可以使用[.schema 表名]的形式，显示指定表的建立命令

```
sqlite>.schema
CREATE TABLE peopleinfo
(_id integer primary key autoincrement,
name text not null,
age integer,
height float);
sqlite>
```

8.3 数据库存储

• 8.3.2 手动建库

- 下一步是向peopleinfo表中添加数据，使用insert into … values命令
- 在下面的代码成功运行后，数据库的peopleinfo表将有三条数据，内容如下表所示:

```
1  sqlite> insert into peopleinfo values(null,'Tom',21,1.81);
2  sqlite> insert into peopleinfo values(null,'Jim',22,1.78);
3  sqlite> insert into peopleinfo values(null,'Lily',19,1.68);
```

- 因为_id是自动增加的主键，因此在输入null后，SQLite数据库会自动填写该项的内容

_id	name	age	height
1	Tom	21	1.81
2	Jim	22	1.78
3	Lily	19	1.68

8.3 数据库存储

- **8.3.2 手动建库**
  - 在数据添加完毕后，使用select命令，显示peopleinfo数据表中的所有数据信息，命令格式为[select 属性 from 表名]
  - 下面的代码用来显示peopleinfo表的所有数据

```
8     select * from peopleinfo;
9     1|Tom|21|1.81
10    2|Jim|22|1.78
11    3|Lily|19|1.68
12    sqlite>
```

8.3 数据库存储

• 8.3.2 手动建库

- 上面的查询结果看起来不是很直观，使用表格方式显示更应更符合习惯，因此可以使用.mode命令更改结果输出格式
- .mode命令除了支持常见的column格式外，还支持csv格式、html格式、insert格式、line格式、list格式、tabs格式和tcl格式
- 下面使用column格式显示peopleinfo数据表中的数据信息

```
1  sqlite> .mode column
2  sqlite> select * from peopleinfo;
3  1          Tom        21         1.81
4  2          Jim        22         1.78
5  3          Lily       19         1.68
6  sqlite>
```

8.3 数据库存储

- **8.3.2 手动建库**
  - 更新数据可以使用update命令，命令格式为[update 表名 set 属性=”新值” where 条件]
  - 更新数据后，同样使用select命令显示数据，确定数据是否正确更新
  - 下面的代码将Lily的身高更新为1.88

```
1   sqlite> update peopleinfo set height=1.88 where name="Lily";
2   sqlite> select * from peopleinfo;
3   select * from peopleinfo;
4   1       Tom     21      1.81
5   2       Jim     22      1.78
6   3       Lily    19      1.88
7   sqlite>
```

8.3 数据库存储

- **8.3.2 手动建库**
  - 删除数据可以使用delete命令
  - 命令格式为[delete from 表名where 条件]
  - 下面的代码将_id为3数据从表peopleinfo中删除

```
1    sqlite> delete from peopleinfo where _id=3;
2    sqlite> select * from peopleinfo;
3    select * from peopleinfo;
4    1        Tom       21        1.81
5    2        Jim       22        1.78
6    sqlite>
```

8.3 数据库存储

• 8.3.2 手动建库

- sqlite3工具还支持很多命令，可以使用.help命令查询sqlite3的命令列表，也可以参考右表

编号	命令	说明
1	.bail ON|OFF	遇到错误时停止，缺省为OFF
2	.databases	显示数据库名称和文件位置
3	.dump ?TABLE? ...	将数据库以SQL文本形式导出
4	.echo ON|OFF	开启和关闭回显
5	.exit	退出
6	.explain ON|OFF	开启或关闭适当输出模式，如果开启模式将更改为column，并自动设置宽度
7	.header(s) ON|OFF	开启或关闭标题显示
8	.help	显示帮助信息
9	.import FILE TABLE	将数据从文件导入表
10	.indices TABLE	显示表中所的列名
11	.load FILE ?ENTRY?	导入扩展库

8.3 数据库存储

• 8.3.2 手动建库

编号	命令	说明
12	.mode MODE ?TABLE?	设置输入格式
13	.nullvalue STRING	打印时使用STRING代替NULL
14	.output FILENAME	将输入保存到文件
15	.output stdout	将输入显示在屏幕上
16	.prompt MAIN CONTINUE	替换标准提示符
17	.quit	退出
18	.read FILENAME	在文件中执行SQL语句
19	.schema ?TABLE?	显示表的创建语句
20	.separator STRING	更改输入和导入的分隔符
21	.show	显示当前设置变量值
22	.tables ?PATTERN?	显示符合匹配模式的表名
23	.timeout MS	尝试打开被锁定的表MS毫秒
24	.timer ON|OFF	开启或关闭CPU计时器
25	.width NUM NUM ...	设置"column"模式的宽度

8.3 数据库存储

8.3.3 代码建库

（1）利用SQLiteDatabase类创建数据库

- openOrCreateDatabases()
- openDatabases()

（2）利用SQLiteOpenHelper抽象类辅助建立数据库

- getWritableDatabase()：返回可写的数据库对象
- getReadableDatabase()：返回可读的数据库对象

8.3 数据库存储

8.3.3 代码建库

操作数据库有两种方法：

1.执行SQL语句实现增删改查，使用SQLiteDatabase的execSQL()进行操作。

2.利用ContentValues和HashTable进行操作。前者只能存储基本类型的数据，后者可以存储对象。使用SQLiteDatabase的Insert()、Update()、Delete()、Query()函数进行操作。

8.3 数据库存储

8.3.3 代码建库

```
12    private final Context context;
13    private DBOpenHelper dbOpenHelper;
14
15    private static class DBOpenHelper extends SQLiteOpenHelper {}
16
17    public DBAdapter(Context _context) {
18        context = _context;
19    }
20
21    public void open() throws SQLiteException {
22        dbOpenHelper = new DBOpenHelper(context, DB_NAME, null,
      DB_VERSION);
23        try {
24    db = dbOpenHelper.getWritableDatabase();
```

8.3 数据库存储

8.3.3 代码建库

```
25          }catch (SQLiteException ex) {
26      db = dbOpenHelper.getReadableDatabase();
27          }
28      }
29
30      public void close() {
31          if (db != null){
32      db.close();
33      db = null;
34          }
35      }
36      }
```

8.3 数据库存储

- **8.3.3 代码建库**
  - 从代码的第2行到第9行可以看出，在DBAdapter类中首先声明了数据库的基本信息，包括数据库的文件名称、表名称和版本号，以及数据库表的属性名称
  - 从这些基本信息上不难发现，这个数据库与前一小节手动建立的数据库是完全相同的
  - 代码第11行声明了SQLiteDatabase的实例
  - SQLiteDatabase类封装了较多的方法，用以建立、删除数据库，执行SQL命令，对数据进行管理等工作

8.3 数据库存储

- **8.3.3 代码建库**
  - 代码第13行声明了一个非常重要的帮助类SQLiteOpenHelper，这个帮助类可以辅助建立、更新和打开数据库
  - 虽然在代码第21行定义了open()函数用来打开数据库，但open()函数中并没有任何对数据库进行实际操作的代码，而是调用了**SQLiteOpenHelper类的getWritableDatabase()函数和getReadableDatabase()函数**
  - 这两个函数会根据数据库是否存在、版本号和是否可写等情况，决定在返回数据库实例前，是否需要建立数据库

8.3 数据库存储

- **8.3.3 代码建库**
  - 在代码第30行的close()函数中，调用了SQLiteDatabase实例的close()方法关闭数据库
  - 这是代码中唯一一处直接调用了SQLiteDatabase实例的方法
  - SQLiteDatabase中也封装了打开数据库的函数openDatabases()和创建数据库函数openOrCreateDatabases()，因为代码中使用了帮助类SQLiteOpenHelper，从而避免直接调用SQLiteDatabase的打开和创建数据库的方法，简化了数据库打开过程中繁琐的逻辑判断过程

8.3 数据库存储

8.3.3 代码建库

DBOpenHelper继承了帮助类SQLiteOpenHelper，重载了onCreate()函数和onUpgrade()函数，代码如下:

```
1    private static class DBOpenHelper extends SQLiteOpenHelper {
2  public DBOpenHelper(Context context, String name, CursorFactory factory, int
   version){
3          super(context, name, factory, version);
4          }
5  private static final String DB_CREATE = "create table " +
6         DB_TABLE + " (" + KEY_ID + " integer primary key autoincrement, " +
7         KEY_NAME+ " text not null, " + KEY_AGE+ " integer," + KEY_HEIGHT
   + " float);";
8
9  @Override
```

8.3 数据库存储

• 8.3.3 代码建库

```
10  public void onCreate(SQLiteDatabase _db) {
11          _db.execSQL(DB_CREATE);
12          }
13
14  @Override
15  public void onUpgrade(SQLiteDatabase _db, int _oldVersion, int _newVersion) {
16          _db.execSQL("DROP TABLE IF EXISTS " + DB_TABLE);
17          onCreate(_db);
18  }
19     }
```

8.3 数据库存储

- **8.3.3 代码建库**
  - 代码的第5行到第7行是创建表的SQL命令
  - 代码第10行和第15行分别重载了onCreate()函数和onUpgrade()函数，这是继承SQLiteOpenHelper类必须重载的两个函数
  - onCreate()函数在数据库第一次建立时被调用，一般用来创建数据库中的表，并完成初始化工作
  - 在代码第11行中，通过调用SQLiteDatabase实例的execSQL()方法，执行创建表的SQL命令
  - onUpgrade()函数在数据库需要升级时被调用，一般用来删除旧的数据库表，并将数据转移到新版本的数据库表中
  - 在代码第16行和第17行中，为了简单起见，并没有做任何数据转移，而仅仅删除原有的表后建立新的数据表

8.3 数据库存储

- **8.3.3 代码建库**
  - 程序开发人员不应直接调用onCreate()和onUpgrade()函数，而应由SQLiteOpenHelper类来决定何时调用这两个函数
  - SQLiteOpenHelper类的getWritableDatabase()函数和getReadableDatabase()函数是可以直接调用的函数
  - getWritableDatabase()函数用来建立或打开可读写的数据库实例，一旦函数调用成功，数据库实例将被缓存，在需要使用数据库实例时就可以调用这个方法获取数据库实例，务必在不使用时调用close()函数关闭数据库
  - 如果保存数据库文件的磁盘空间已满，调用getWritableDatabase()函数则无法获得可读写的数据库实例，这时可以调用getReadableDatabase()函数，获得一个只读的数据库实例

8.3 数据库存储

- **8.3.3 代码建库**
  - 如果不希望使用SQLiteOpenHelper类，也可以直接使用SQL命令建立数据库，方法是：
    - 先调用openOrCreateDatabases()函数创建数据库实例
    - 然后调用execSQL()函数执行SQL命令，完成数据库和数据表的建立过程，其示例代码如下：

```
1   private static final String DB_CREATE = "create table " +
2    DB_TABLE + " (" + KEY_ID + " integer primary key autoincrement, " +
3    KEY_NAME+ " text not null, " + KEY_AGE+ " integer," + KEY_HEIGHT + "
     float);";
4   public void create() {
5    db.openOrCreateDatabases(DB_NAME, context.MODE_PRIVATE, null)
6    db.execSQL(DB_CREATE);
7   }
```

8.3 数据库存储

- **8.3.4 数据操作**
  - 数据操作指的是对数据的添加、删除、查找和更新操作
  - 虽然程序开发人员完全可以通过执行SQL命名完成数据操作，但这里仍然推荐使用Android提供的专用类和方法，这些类和方法的使用更加简洁、方便
  - 为了使DBAdapter类支持数据添加、删除、更新和查找等功能，在DBAdapter类中增加下面的函数:
    - insert(People people)用来添加一条数据
    - queryAllData()用来获取全部数据
    - queryOneData(long id)根据id获取一条数据
    - deleteAllData()用来删除全部数据
    - deleteOneData(long id)根据id删除一条数据
    - updateOneData(long id , People people)根据id更新一条数据

8.3 数据库存储

- **8.3.4 数据操作**
  - deleteAllData()用来删除全部数据
  - deleteOneData(long id)根据id删除一条数据
  - updateOneData(long id , People people)根据id更新一条数据

```
1    public class DBAdapter {
2     public long insert(People people) {}
3     public long deleteAllData() { }
4     public long deleteOneData(long id) { }
5     public People[] queryAllData() {}
6     public People[] queryOneData(long id) { }
7     public long updateOneData(long id , People people){ }
8
9     private People[] ConvertToPeople(Cursor cursor){}
10   }
```

8.3 数据库存储

- **8.3.4 数据操作**
  - ConvertToPeople(Cursor cursor)是私有函数，作用是将查询结果转换为自定义的People类实例
  - People类包含四个公共属性，分别为ID、Name、Age和Height，对应数据库中的四个属性值
  - 重载toString()函数，主要是便于界面显示的需要
  - People类的代码如下:

```
1    public class People {
2        public int ID = -1;
3        public String Name;
4        public int Age;
5        public float Height;
6
```

8.3 数据库存储

• 8.3.4 数据操作

```
7           @Override
8           public String toString(){
9       String result = "";
10      result += "ID：" + this.ID + "，";
11      result += "姓名：" + this.Name + "，";
12      result += "年龄：" + this.Age + "，  ";
13      result += "身高：" + this.Height + "，";
14      return result;
15          }
16      }
```

8.3 数据库存储

- **8.3.4 数据操作**
  - SQLiteDatabase类的公有函数insert()、delete()、update()和query()，封装了执行添加、删除、更新和查询功能的SQL命令
  - 下面分别介绍如何使用SQLiteDatabase类的公有函数，完成数据的添加、删除、更新和查询等操作

8.3 数据库存储

- **8.3.4 数据操作**
  - 添加功能
    - 首先构造一个ContentValues实例，然后调用ContentValues实例的put()方法，将每个属性的值写入到ContentValues实例中，最后使用SQLiteDatabase实例的insert()函数，将ContentValues实例中的数据写入到指定的数据表中
    - insert()函数的返回值是新数据插入的位置，即ID值。ContentValues类是一个数据承载容器，主要用来向数据库表中添加一条数据

8.3 数据库存储

- **8.3.4 数据操作**
  - 第4行代码向ContentValues对象newValues中添加一个名称/值对，put()函数的第1个参数是名称，第2个参数是值
  - 在第8行代码的insert()函数中，第1个参数是数据表的名称，第2个参数是在NULL时的替换数据，第3个参数是需要向数据库表中添加的数据

8.3 数据库存储

■ 8.3.4 数据操作

```
1    public long insert(People people) {
2     ContentValues newValues = new ContentValues();
3
4     newValues.put(KEY_NAME, people.Name);
5     newValues.put(KEY_AGE, people.Age);
6     newValues.put(KEY_HEIGHT, people.Height);
7
8     return db.insert(DB_TABLE, null, newValues);
9    }
```

8.3 数据库存储

- **8.3.4 数据操作**
  - 删除功能
    - 删除数据比较简单，只需要调用当前数据库实例的delete()函数，并指明表名称和删除条件即可

```
1      public long deleteAllData() {
2       return db.delete(DB_TABLE, null, null);
3      }
4
5      public long deleteOneData(long id) {
6       return db.delete(DB_TABLE,  KEY_ID + "=" + id, null);
7      }
```

8.3 数据库存储

- **8.3.4 数据操作**
  - 删除功能
    - delete()函数的第1个参数是数据表名称，第2个参数是删除条件
    - 在第2行代码中，删除条件为null，表示删除表中的所有数据
    - 代码第6行则指明需要删除数据的id值，因此deleteOneData()函数仅删除一条数据，此时delete()函数的返回值表示被删除的数据数量

8.3 数据库存储

- **8.3.4 数据操作**
  - 更新功能
    - 更新数据同样要使用ContentValues实例，首先构造ContentValues实例，然后调用put()函数将属性值写入到ContentValues实例中，最后使用SQLiteDatabase的update()函数，并指定数据的更新条件

```
1    public long updateOneData(long id , People people){
2     ContentValues updateValues = new ContentValues();
3     updateValues.put(KEY_NAME, people.Name);
4     updateValues.put(KEY_AGE, people.Age);
5     updateValues.put(KEY_HEIGHT, people.Height);
6
7     return db.update(DB_TABLE, updateValues,  KEY_ID + "=" + id, null);
8    }
```

8.3 数据库存储

- **8.3.4 数据操作**
  - 更新功能
    - 在代码的第7行中，update()函数的第1个参数表示数据表的名称，第2个参数是更新条件。update()函数的返回值表示数据库表中被更新的数据数量

8.3 数据库存储

- **8.3.4 数据操作**
  - 查询功能
    - 介绍查询功能前，先要介绍一下Cursor类
    - 在Android系统中，数据库查询结果的返回值并不是数据集合的完整拷贝，而是返回数据集的指针，这个指针就是Cursor类
    - Cursor类支持在查询结果的数据集合中以多种方式移动，并能够获取数据集合的属性名称和序号，具体的方法和说明可以参考下表

8.3 数据库存储

- **8.3.4 数据操作**
  - Cursor类的公有方法

函数	说明
moveToFirst	将指针移动到第一条数据上
moveToNext	将指针移动到下一条数据上
moveToPrevious	将指针移动到上一条数据上
getCount	获取集合的数据数量
getColumnIndexOrThrow	返回指定属性名称的序号，如果属性不存在则产生异常
getColumnName	返回指定序号的属性名称
getColumnNames	返回属性名称的字符串数组
getColumnIndex	根据属性名称返回序号
moveToPosition	将指针移动到指定的数据上
getPosition	返回当前指针的位置

8.3 数据库存储

- **8.3.4 数据操作**
  - 从Cursor中提取数据可以参考ConvertToPeople()函数的实现方法
  - 在提取Cursor数据中的数据前，推荐测试Cursor中的数据数量，避免在数据获取中产生异常，例如下面代码的第3行到第5行
  - 从Cursor中提取数据使用类型安全的get<Type>()函数，函数的参数是属性的序号，为了获取属性的序号，可以使用getColumnIndex()函数获取指定属性的序号

8.3 数据库存储

■ 8.3.4 数据操作

```
1    private People[] ConvertToPeople(Cursor cursor){
2      int resultCounts = cursor.getCount();
3      if (resultCounts == 0 || !cursor.moveToFirst()){
4      return null;
5      }
6      People[] peoples = new People[resultCounts];
7      for (int i = 0 ; i<resultCounts; i++){
8            peoples[i] = new People();
9            peoples[i].ID = cursor.getInt(0);
10           peoples[i].Name = cursor.getString(cursor.getColumnIndex(KEY_NAME));
11           peoples[i].Age = cursor.getInt(cursor.getColumnIndex(KEY_AGE));
12           peoples[i].Height = cursor.getFloat(cursor.getColumnIndex(KEY_HEIGHT));
13           cursor.moveToNext();
14     }
15     return peoples;
16   }
```

8.3 数据库存储

• 8.3.4 数据操作

- 要进行数据查询就需要调用SQLiteDatabase类的query()函数，query()函数的语法如下:
- **Cursor android.database.sqlite.SQLiteDatabase.query(String table, String[] columns, String selection, String[] selectionArgs, String groupBy, String having, String orderBy)**
- query()函数的参数说明

位置	类型+名称	说明
1	String table	表名称
2	String[] columns	返回的属性列名称
3	String selection	查询条件
4	String[] selectionArgs	如果在查询条件中使用通配符(?)，则需要在这里定义替换符的具体内容
5	String groupBy	分组方式
6	String having	定义组的过滤器
7	String orderBy	排序方式

8.3 数据库存储

- **8.3.4 数据操作**
  - 根据id查询数据的代码

```
1    public People[] getOneData(long id) {
2     Cursor results =  db.query(DB_TABLE, new String[] { KEY_ID, KEY_NAME,
      KEY_AGE, KEY_HEIGHT}, KEY_ID + "=" + id, null, null, null, null);
3     return ConvertToPeople(results);
4     }
```

8.3 数据库存储

- **8.3.4 数据操作**
  - 查询全部数据的代码

```
1    public People[] getAllData() {
2     Cursor results = db.query(DB_TABLE, new String[] { KEY_ID, KEY_NAME,
      KEY_AGE, KEY_HEIGHT}, null, null, null, null, null);
3     return ConvertToPeople(results);
4    }
```

8.3 数据库存储

- **8.3.4 数据操作**
  - SQLiteDemo用户界面

8.3 数据库存储

- **8.3.4 数据操作**
  - SQLiteDemo是对SQLite数据库进行操作的示例
  - 在这个示例中，用户可以在界面的上方输入数据信息，通过“添加数据”按钮将数据写入数据库
  - “全部显示”相当于查询数据库中的所有数据，并将数据显示在界面下方
  - “清除显示”仅是清除界面下面显示数据，而不对数据库进行任何操作
  - “全部删除”是数据库操作，将删除数据库中的所有数据
  - 在界面中部，以“ID+功能”命名的按钮，分别是根据ID删除数据、查询数据和更新数据，而这个ID值就取自本行的EditText控件

8.4 数据分享

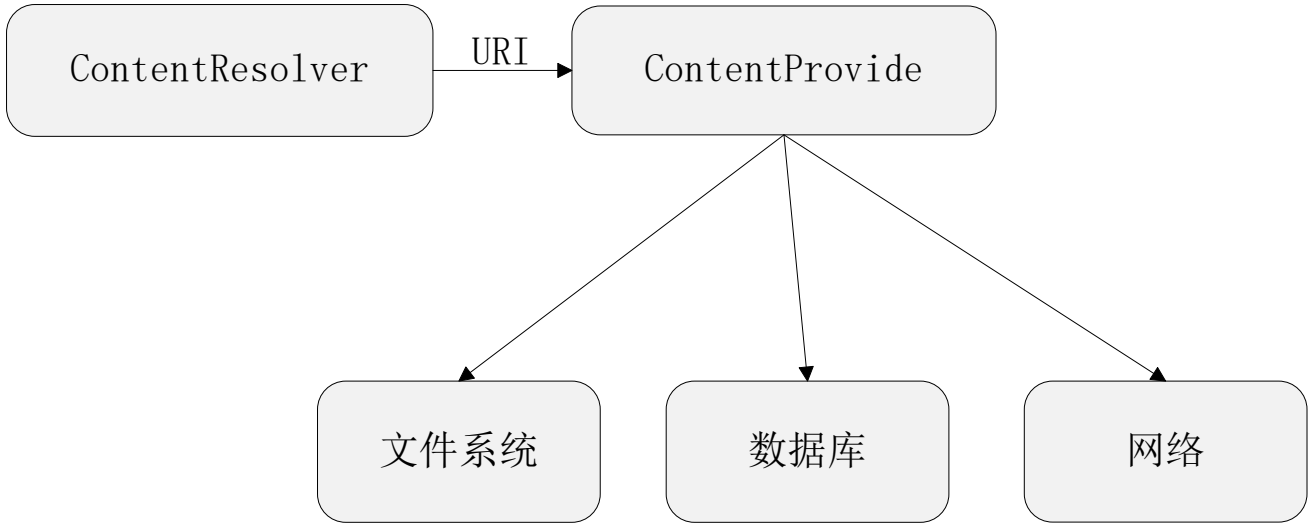
- **8.4.1 ContentProvider**
  - Android应用程序运行在不同的进程空间中，因此不同应用程序的数据是不能够直接访问的
  - 为了增强程序之间的数据共享能力，Android系统提供了像SharedPreferences这类简单的跨越程序边界的访问方法，但这些方法都存在一定的局限性
  - ContentProvider（数据提供者）是应用程序之间共享数据的一种接口机制，是一种更为高级的数据共享方法，可以指定需要共享的数据，而其它应用程序则可在不知道数据来源、路径的情况下，对共享数据进行查询、添加、删除和更新等操作

8.4 数据分享

- **8.4.1 ContentProvider**
  - 在Android系统中，许多Android系统内置的数据也是通过ContentProvider提供给用户使用，例如通讯录、音视频文件和图像文件等
  - 在创建ContentProvider前，首先要实现底层的数据源，数据源包括数据库、文件系统或网络等，然后继承ContentProvider类中实现基本数据操作的接口函数，包括添加、删除、查找和更新等功能

8.4 数据分享

- **8.4.1 ContentProvider**
  - 调用者不能直接调用ContentProvider的接口函数，而需要使用ContentResolver对象，通过URI间接调用ContentProvider，调用关系如下图所示:



8.4 数据分享

- **8.4.1 ContentProvider**
  - 在ContentResolver对象与ContentProvider进行交互时，通过URI确定要访问的ContentProvider数据集
  - 在发起一个请求的过程中，Android系统根据URI确定处理这个查询的ContentProvider，然后初始化ContentProvider所有需要的资源，这个初始化的工作是Android系统完成的，无需程序开发人员参与
  - 一般情况下只有一个ContentProvider对象，但却可以同时与多个ContentResolver进行交互

8.4 数据分享

• 8.4.1 ContentProvider

- ContentProvider完全屏蔽了数据提供组件的数据存储方法
- 在程序开发人员看来，数据提供者通过ContentProvider提供了一组标准的数据操作接口，但却无需知道数据提供者的内部数据的存储方法
- 数据提供者可以使用SQLite数据库存储数据，也可以通过文件系统或SharedPreferences存储数据，甚至是使用网络存储的方法，这些数据的存储方法和存储设备对数据使用者都是不可见的
- 同时，也正是这种屏蔽模式，很大程度上简化的ContentProvider的使用方法，使用者只要调用ContentProvider提供的接口函数，即可完成所有的数据操作，而数据存储方法则是ContentProvider设计者需要考虑的问题

8.4 数据分享

• 8.4.1 ContentProvider

- ContentProvider的数据集类似于数据库的数据表，每行是一条记录，每列具有相同的数据类型
- 如下表所示。每条记录都包含一个长型的字段_ID，用来唯一标识每条记录
- ContentProvider可以提供多个数据集，调用者使用URI对不同数据集的数据进行操作
- ContentProvider数据集

_ID	NAME	AGE	HEIGHT
1	Tom	21	1.81
2	Jim	22	1.78

8.4 数据分享

- **8.4.1 ContentProvider**
  - URI是通用资源标志符（Uniform Resource Identifier），用来定位远程或本地的可用资源
  - ContentProvider使用的URI语法结构如下:
    - **1　　content://<authority>/<data_path>/<id>**
    - content://是通用前缀，表示该URI用于ContentProvider定位资源，无需修改
    - <authority>是授权者名称，用来确定具体由哪一个ContentProvider提供资源
    - 因此，一般<authority>都由类的小写全称组成，以保证唯一性。<data_path>是数据路径，用来确定请求的是哪个数据集

8.4 数据分享

- **8.4.1 ContentProvider**
  - 如果ContentProvider仅提供一个数据集，数据路径则是可以省略的
  - 但如果ContentProvider提供多个数据集，数据路径则必须指明具体是哪一个数据集
    - 数据集的数据路径可以写成多段格式，例如people/girl和/people/boy。<id>是数据编号，用来唯一确定数据集中的一条记录，用来匹配数据集中_ID字段的值
  - 如果请求的数据并不只限于一条数据，则<id>是可以省略，例如
    - 请求整个people数据集的URI应写为
    - 1 **content://edu.hrbeu.peopleprovider/people**
    - 请求people数据集中第3条数据的URI则应写为
    - 1 **content://edu.hrbeu.peopleprovider/people/3**

8.4 数据分享

- **8.4.2 创建数据提供者**
  - 程序开发人员通过继承ContentProvider类可以创建一个新的数据提供者，过程可以分为三步
    - 继承ContentProvider，并重载六个函数
    - 声明CONTENT_URI，实现UriMatcher
    - 注册ContentProvider

8.4 数据分享

- **8.4.2 创建数据提供者**
  - 继承ContentProvider，并重载六个函数
    - 新建立的类继承ContentProvider后，共有六个函数需要重载，分别是
      - delete():删除数据集
      - insert()：添加数据集
      - qurey()：查询数据集
      - update()：更新数据集
      - onCreate()：初始化底层数据集和建立数据连接等工作
      - getType()：返回指定URI的MIME数据类型
        - 如果URI是单条数据，则返回的MIME数据类型应以vnd.android.cursor.item开头
        - 如果URI是多条数据，则返回的MIME数据类型应以vnd.android.cursor.dir/开头

8.4 数据分享

• 8.4.2 创建数据提供者

- 继承ContentProvider，并重载六个函数
  - 新建立的类继承ContentProvider后，Eclipse会提示程序开发人员需要重载部分的代码，并自动生成需要重载的代码框架
  - 下面的代码是Eclipse自动生成的代码框架

```
1   import android.content.*;
2   import android.database.Cursor;
3   import android.net.Uri;
4
5   public class PeopleProvider extends ContentProvider{
6
7       @Override
8       public int delete(Uri uri, String selection, String[] selectionArgs) {
9     // TODO Auto-generated method stub
10    return 0;
```

8.4 数据分享

• 8.4.2 创建数据提供者

- 继承ContentProvider，并重载六个函数

```
11          }
12
13          @Override
14          public String getType(Uri uri) {
15      // TODO Auto-generated method stub
16      return null;
17          }
18
19          @Override
20          public Uri insert(Uri uri, ContentValues values) {
21      // TODO Auto-generated method stub
22      return null;
23          }
24
25          @Override
26          public boolean onCreate() {
27      // TODO Auto-generated method stub
```

8.4 数据分享

• 8.4.2 创建数据提供者

- 继承ContentProvider，并重载六个函数

```
28          return false;
29              }
30
31              @Override
32              public Cursor query(Uri uri, String[] projection, String selection,
33      String[] selectionArgs, String sortOrder) {
34          // TODO Auto-generated method stub
35          return null;
36              }
37
38              @Override
39              public int update(Uri uri, ContentValues values, String selection,
40      String[] selectionArgs) {
41          // TODO Auto-generated method stub
42          return 0;
43              }
44  }
```

8.4 数据分享

- **8.4.2 创建数据提供者**
  - 声明CONTENT_URI，实现UriMatcher
    - 在新构造的ContentProvider类中，经常需要判断URI是单条数据还是多条数据，最简单的方法是构造一个UriMatcher
    - 为了便于判断和使用URI，一般将URI的授权者名称和数据路径等内容声明为静态常量，并声明CONTENT_URI

8.4 数据分享

• 8.4.2 创建数据提供者

- 声明CONTENT_URI和构造UriMatcher的代码如下:

```
1   public static final String AUTHORITY = "edu.hrbeu.peopleprovider";
2   public static final String PATH_SINGLE = "people/#";
3   public static final String PATH_MULTIPLE = "people";
4   public static final String CONTENT_URI_STRING = "content://" + AUTHORITY + "/" +
    PATH_MULTIPLE;
5   public static final Uri  CONTENT_URI = Uri.parse(CONTENT_URI_STRING);
6   private static final int MULTIPLE_PEOPLE = 1;
7   private static final int SINGLE_PEOPLE = 2;
8
9   private static final UriMatcher uriMatcher;
10  static {
11  uriMatcher = new UriMatcher(UriMatcher.NO_MATCH);
12  uriMatcher.addURI(AUTHORITY, PATH_SINGLE, MULTIPLE_PEOPLE);
13  uriMatcher.addURI(AUTHORITY, PATH_MULTIPLE, SINGLE_PEOPLE);

14  }
```

8.4 数据分享

- **8.4.2 创建数据提供者**
  - 声明CONTENT_URI，实现UriMatcher
    - 第1行声明了URI的授权者名称
    - 第2行声明了单条数据的数据路径
    - 第3行声明了多条数据的数据路径
    - 第4行声明了CONTENT_URI的字符串形式
    - 第5行则正式声明了CONTENT_URI
    - 第6行声明了多条数据的返回代码
    - 第7行声明了单条数据的返回代码
    - 第9行声明了UriMatcher
    - 第10行到第13行的静态构造函数中，声明了UriMatcher的匹配方式和返回代码

8.4 数据分享

- **8.4.2 创建数据提供者**
  - 声明CONTENT_URI，实现UriMatcher
    - 在第11行UriMatcher的构造函数中，UriMatcher.NO_MATCH是URI无匹配时的返回代码
    - 第12行的addURI()函数用来添加新的匹配项，语法如下:
    - **1　　public void addURI (String authority, String path, int code)**
      - authority表示匹配的授权者名称
      - path表示数据路径
      - #可以代表任何数字
      - code表示返回代码

8.4 数据分享

- **8.4.2 创建数据提供者**
  - 声明CONTENT_URI，实现UriMatcher
    - 使用UriMatcher时，则可以直接调用match()函数，对指定的URI进行判断，示例代码如下：

```
1   switch(uriMatcher.match(uri)){
2      case MULTIPLE_PEOPLE:
3            //多条数据的处理过程
4            break;
5      case SINGLE_PEOPLE:
6            //单条数据的处理过程
7            break;
8      default:
9            throw new IllegalArgumentException("不支持的URI:" + uri);
10  }
```

8.4 数据分享

- **8.4.2 创建数据提供者**
  - 注册ContentProvider
    - 在完成ContentProvider类的代码实现后，需要在AndroidManifest.xml文件中进行注册
    - 注册ContentProvider使用<provider>标签，示例代码如下:

```
1        <application android:icon="@drawable/icon" android:label="@string/app_name">
2            <provider android:name = ".PeopleProvider"
3                      android:authorities = "edu.hrbeu.peopleprovider"/>
4        </application>
```

    - 在上面的代码中，注册了一个授权者名称为edu.hrbeu.peopleprovider的ContentProvider，其实现类是PeopleProvider

8.4 数据分享

- **8.4.3 使用数据提供者**
  - 使用ContentProvider并不需要直接调用类中的数据操作函数，而是通过Android组件都具有的ContentResolver对象，通过URI进行数据操作
  - 程序开发人员只需要知道URI和数据集的数据格式，则可以进行数据操作，解决不同应用程序之间的数据共享问题
  - 每个Android组件都具有一个ContentResolver对象，获取ContentResolver对象的方法是调用getContentResolver()函数

```
3    ContentResolver resolver = getContentResolver();
```

8.4 数据分享

• 8.4.3 使用数据提供者

- 查询操作
  - 在获取到ContentResolver对象后，程序开发人员则可以使用query()函数查询目标数据
  - 下面的代码是查询ID为2的数据

```
1    String KEY_ID = "_id";
2    String KEY_NAME = "name";
3    String KEY_AGE = "age";
4    String KEY_HEIGHT = "height";
5
6    Uri uri = Uri.parse(CONTENT_URI_STRING + "/" + "2";
7    Cursor cursor = resolver.query(uri,
8    new String[] {KEY_ID, KEY_NAME, KEY_AGE, KEY_HEIGHT}, null, null,
     null);
```

8.4 数据分享

• 8.4.3 使用数据提供者

- 查询操作
  - 从上面的代码不难看出，在URI中定义了需要查询数据的ID后，在query()函数中则没有必要再加入其他的查询条件
  - 如果需要获取数据集中的全部数据，则可直接使用CONTENT_URI，此时ContentProvider在分析URI时将认为需要返回全部数据
- ContentResolver的query()函数与SQLite数据库的query()函数非常相似，语法结构如下:

```
1 Cursor query(Uri uri, String[] projection, String selection, String[] selectionArgs, String sortOrder)
```

  - uri定义了查询的数据集，projection定义了从应返回数据的哪些属性，selection定义了返回数据的查询条件

8.4 数据分享

• 8.4.3 使用数据提供者

- 添加操作
  - 向ContentProvider中添加数据有两种方法
    - 一种是使用insert()函数，向ContentProvider中添加一条数据
    - 另一种是使用bultInsert()函数，批量的添加数据
  - 下面的代码说明了如何使用insert()函数添加单条数据

```
1    ContentValues values = new ContentValues();
2    values.put(KEY_NAME, "Tom");
3    values.put(KEY_AGE, 21);
4    values.put(KEY_HEIGHT, 1.81f);
5
6    Uri newUri = resolver.insert(CONTENT_URI, values);
```

8.4 数据分享

• 8.4.3 使用数据提供者

- 添加操作
  - 下面的代码说明了如何使用bultInsert()函数添加多条数据

```
1    ContentValues[] arrayValues = new ContentValues[10];
2    //实例化每一个ContentValues
3    int count = resolver.bultInsert(CONTENT_URI, arrayValues);
```

8.4 数据分享

• 8.4.3 使用数据提供者

- 删除操作
  - 删除操作需要使用delete()函数
  - 如果需要删除单条数据，则可以在URI中指定需要删除数据的ID
  - 如果需要删除多条数据，则可以在selection中声明删除条件
  - 下面代码说明了如何删除ID为2的数据

```
1        Uri uri = Uri.parse(CONTENT_URI_STRING + "/" + "2");
2        int result = resolver.delete(uri, null, null);
```

  - 也可以在selection将删除条件定义为ID大于4的数据

```
1        String selection = KEY_ID + ">4";
2        int result = resolver.delete(CONTENT_URI, selection, null);
```

8.4 数据分享

• 8.4.3 使用数据提供者

- 更新操作
  - 更新操作需要使用update()函数，参数定义与delete()函数相同，同样可以在URI中指定需要更新数据的ID，也可以在selection中声明更新条件
  - 下面代码说明了如何更新ID为7的数据

```
1     ContentValues values = new ContentValues();
7     values.put(KEY_NAME, "Tom");
8     values.put(KEY_AGE, 21);
9     values.put(KEY_HEIGHT, 1.81f);
2
3     Uri uri = Uri.parse(CONTENT_URI_STRING + "/" + "7");
4     int result = resolver.update(uri, values, null, null);
```

8.4 数据分享

- **8.4.4 示例**
  - ContentProviderDemo是一个无界面的示例，仅提供一个ContentProvider组件，供其它应用程序进行数据交换
    - 底层使用SQLite数据库，支持数据的添加、删除、更新和查询等基本操作
    - ContentResolverDemo是调用ContentProvider的示例，自身不具有任何数据存储功能，仅是通过URI访问ContentProviderDemo示例提供的ContentProvider,所以在运行ContentResolverDemo之前要确保已经安装了ContentProviderDemo界面如下图所示，该界面基本与示例界面相同

8.4 数据分享

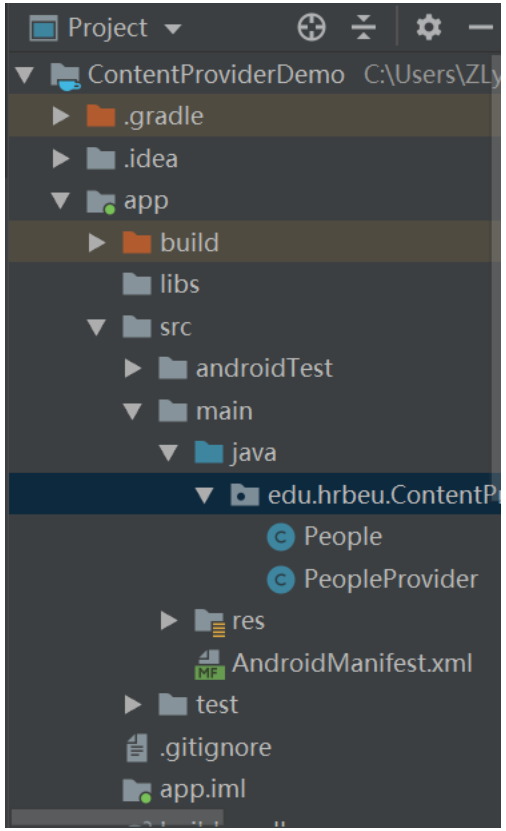
- **8.4.4 示例**

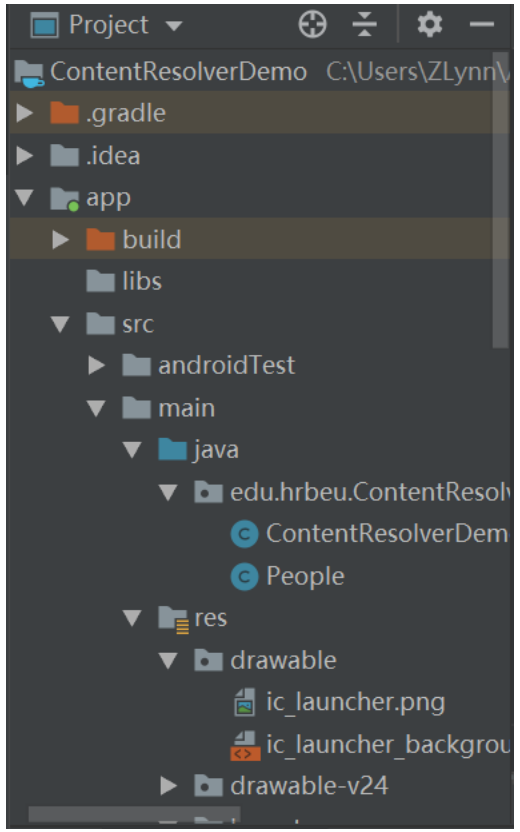
8.4 数据分享

- **8.4.4 示例**
  - 从下图的文件结构上可以发现，两个示例都包含一个相同的文件People.java，两个示例中的这个文件的内容也完全相同，定义了数据提供者和数据调用者都必须知道的信息。这些信息包括授权者名称、数据路径、MIME数据类型、CONTENT_URI和数据项名称等。

8.4 数据分享

- **8.4.4 示例**
  - 左图为ContentProviderDemo，右图为ContentResolverDemo





8.4 数据分享

- **8.4.4 示例**
  - 下面分别给出People.java文件、PeopleProvider.java文件和ContentResolverDemoActivity.java文件的完整代码，最后分别给出ContentProviderDemo示例和ContentResolverDemo示例的AndroidManifest.xml文件内容
  - People.java文件的完整代码如下：

```
1    package edu.hrbeu.ContentResolverDemo;
2    import android.net.Uri;
3
4    public class People{
5
6        public static final String MIME_DIR_PREFIX = "vnd.android.cursor.dir";
7        public static final String MIME_ITEM_PREFIX = "vnd.android.cursor.item";
8        public static final String MINE_ITEM = "vnd.hrbeu.people";
```

8.4 数据分享

- **8.4.4 示例**
  - People.java文件代码

```
9
10          public static final String MINE_TYPE_SINGLE = MIME_ITEM_PREFIX + "/" + MINE_ITEM;
11          public static final String MINE_TYPE_MULTIPLE = MIME_DIR_PREFIX + "/" + MINE_ITEM;
12
13          public static final String AUTHORITY = "edu.hrbeu.peopleprovider";
14          public static final String PATH_SINGLE = "people/#";
15          public static final String PATH_MULTIPLE = "people";
16          public static final String CONTENT_URI_STRING = "content://" + AUTHORITY + "/" +
        PATH_MULTIPLE;
17          public static final Uri  CONTENT_URI = Uri.parse(CONTENT_URI_STRING);
18
19          public static final String KEY_ID = "_id";
20          public static final String KEY_NAME = "name";
21          public static final String KEY_AGE = "age";
22          public static final String KEY_HEIGHT = "height";

23  }
```

8.4 数据分享

- **8.4.4 示例**
  - PeopleProvider.java文件的完整代码如下:

```
1   package edu.hrbeu.ContentProviderDemo;
2
3   import android.content.ContentProvider;
4   import android.content.ContentUris;
5   import android.content.ContentValues;
6   import android.content.Context;
7   import android.content.UriMatcher;
8   import android.database.Cursor;
9   import android.database.SQLException;
10   import android.database.sqlite.SQLiteDatabase;
11   import android.database.sqlite.SQLiteOpenHelper;
12   import android.database.sqlite.SQLiteQueryBuilder;
13   import android.database.sqlite.SQLiteDatabase.CursorFactory;
14   import android.net.Uri;
```

8.4 数据分享

- **8.4.4 示例**
  - PeopleProvider.java文件代码

```
15
16    public class PeopleProvider extends ContentProvider{
17
18        private static final String DB_NAME = "people.db";
19        private static final String DB_TABLE = "peopleinfo";
20        private static final int DB_VERSION = 1;
21
22        private SQLiteDatabase db;
23        private DBOpenHelper dbOpenHelper;
24
25        private static final int MULTIPLE_PEOPLE = 1;
26        private static final int SINGLE_PEOPLE = 2;
27        private static final UriMatcher uriMatcher;
```

8.4 数据分享

- **8.4.4 示例**
  - PeopleProvider.java文件代码

```
28
29      static {
30          uriMatcher = new UriMatcher(UriMatcher.NO_MATCH);
31          uriMatcher.addURI(People.AUTHORITY, People.PATH_MULTIPLE,
    MULTIPLE_PEOPLE);
32          uriMatcher.addURI(People.AUTHORITY, People.PATH_SINGLE,
    SINGLE_PEOPLE);
33      }
34
35      @Override
36      public String getType(Uri uri) {
37          switch(uriMatcher.match(uri)){
38              case MULTIPLE_PEOPLE:
39                  return People.MINE_TYPE_MULTIPLE;
```

8.4 数据分享

- **8.4.4 示例**
  - PeopleProvider.java文件代码

```
40              case SINGLE_PEOPLE:
41                  return People.MINE_TYPE_SINGLE;
42              default:
43                  throw new IllegalArgumentException("Unkown uri:"+uri);
44          }
45      }
46
47      @Override
48      public int delete(Uri uri, String selection, String[] selectionArgs) {
49          int count = 0;
50          switch(uriMatcher.match(uri)){
51              case MULTIPLE_PEOPLE:
52                  count = db.delete(DB_TABLE, selection, selectionArgs);
53                  break;
```

8.4 数据分享

- **8.4.4 示例**
  - PeopleProvider.java文件代码

```
54                 case SINGLE_PEOPLE:
55                     String segment = uri.getPathSegments().get(1);
56                     count = db.delete(DB_TABLE, People.KEY_ID + "=" + segment,
        selectionArgs);
57                     break;
58                 default:
59                     throw new IllegalArgumentException("Unsupported URI:" + uri);
60             }
61             getContext().getContentResolver().notifyChange(uri, null);
62             return count;
63         }
64
65         @Override
66         public Uri insert(Uri uri, ContentValues values) {
```

8.4 数据分享

- **8.4.4 示例**
  - PeopleProvider.java文件代码

```
67          long id = db.insert(DB_TABLE, null, values);
68          if ( id > 0 ){
69              Uri newUri = ContentUris.withAppendedId(People.CONTENT_URI, id);
70              getContext().getContentResolver().notifyChange(newUri, null);
71              return newUri;
72          }
73          throw new SQLException("Failed to insert row into " + uri);
74      }
75
76      @Override
77      public boolean onCreate() {
78          Context context = getContext();
79          dbOpenHelper = new DBOpenHelper(context, DB_NAME, null, DB_VERSION);
80          db = dbOpenHelper.getWritableDatabase();
```

8.4 数据分享

- **8.4.4 示例**
  - PeopleProvider.java文件代码

```
81
82          if (db == null)
83              return false;
84          else
85              return true;
86      }
87
88      @Override
89      public Cursor query(Uri uri, String[] projection, String selection,
90              String[] selectionArgs, String sortOrder) {
91          SQLiteQueryBuilder qb = new SQLiteQueryBuilder();
92          qb.setTables(DB_TABLE);
93          switch(uriMatcher.match(uri)){
94              case SINGLE_PEOPLE:
```

8.4 数据分享

- **8.4.4 示例**
  - PeopleProvider.java文件代码

```
95                  qb.appendWhere(People.KEY_ID + "=" + uri.getPathSegments().get(1));
96                  break;
97              default:
98                  break;
99          }
100         Cursor cursor = qb.query(db,
101                 projection,
102                 selection,
103                 selectionArgs,
104                 null,
105                 null,
106                 sortOrder);
107         cursor.setNotificationUri(getContext().getContentResolver(), uri);
108         return cursor;
```

8.4 数据分享

- **8.4.4 示例**
  - PeopleProvider.java文件代码

```
109         }
110
111         @Override
112         public int update(Uri uri, ContentValues values, String selection,
113                 String[] selectionArgs) {
114             int count;
115             switch(uriMatcher.match(uri)){
116                 case MULTIPLE_PEOPLE:
117                     count = db.update(DB_TABLE, values, selection, selectionArgs);
118                     break;
119                 case SINGLE_PEOPLE:
120                     String segment = uri.getPathSegments().get(1);
121                     count = db.update(DB_TABLE, values, People.KEY_ID+"="+segment,
    selectionArgs);
```

8.4 数据分享

- **8.4.4 示例**
  - PeopleProvider.java文件代码

```
122                     break;
123                 default:
124                     throw new IllegalArgumentException("Unknow URI:" + uri);
125             }
126             getContext().getContentResolver().notifyChange(uri, null);
127             return count;
128         }
129
130
131         private static class DBOpenHelper extends SQLiteOpenHelper {
132
133             public DBOpenHelper(Context context, String name, CursorFactory factory, int
        version) {
134                 super(context, name, factory, version);
```

8.4 数据分享

- **8.4.4 示例**
  - PeopleProvider.javar文件代码

```
135             }
136
137             private static final String DB_CREATE = "create table " +
138               DB_TABLE + " (" + People.KEY_ID + " integer primary key autoincrement, "
    +
139               People.KEY_NAME+ " text not null, " + People.KEY_AGE+ " integer," +
    People.KEY_HEIGHT + " float);";
140
141             @Override
142             public void onCreate(SQLiteDatabase _db) {
143               _db.execSQL(DB_CREATE);
144             }
145
146             @Override
```

8.4 数据分享

- **8.4.4 示例**
  - PeopleProvider.java文件代码

```
147             public void onUpgrade(SQLiteDatabase _db, int _oldVersion, int _newVersion) {
148               _db.execSQL("DROP TABLE IF EXISTS " + DB_TABLE);
149               onCreate(_db);
150             }
151           }
152     }
```

8.4 数据分享

■ 8.4.4 示例

❑ ContentResolverDemoActivity.java文件的完整代码如下:

```
1   package edu.hrbeu.ContentResolverDemo;
2
3
4   import android.app.Activity;
5   import android.content.ContentResolver;
6   import android.content.ContentValues;
7   import android.database.Cursor;
8   import android.net.Uri;
9   import android.os.Bundle;
10   import android.view.View;
11   import android.view.View.OnClickListener;
12   import android.widget.Button;
13   import android.widget.EditText;
14   import android.widget.TextView;
```

8.4 数据分享

- **8.4.4 示例**
  - ContentResolverDemoActivity.java文件代码

```
15
16  public class ContentResolverDemoActivity extends Activity {
17
18      private EditText nameText;
19      private EditText ageText;
20      private EditText heightText;
21      private EditText idEntry;
22
23      private TextView labelView;
24      private TextView displayView;
25
26      private ContentResolver resolver;
27
28      @Override
```

8.4 数据分享

• 8.4.4 示例

- ContentResolverDemoActivity.java文件代码

```
29      public void onCreate(Bundle savedInstanceState) {
30          super.onCreate(savedInstanceState);
31          setContentView(R.layout.main);
32
33          nameText = (EditText)findViewById(R.id.name);
34          ageText = (EditText)findViewById(R.id.age);
35          heightText = (EditText)findViewById(R.id.height);
36          idEntry = (EditText)findViewById(R.id.id_entry);
37
38          labelView = (TextView)findViewById(R.id.label);
39          displayView = (TextView)findViewById(R.id.display);
40
41
42
```

8.4 数据分享

- **8.4.4 示例**
  - ContentResolverDemoActivity.java文件代码

```
43          Button addButton = (Button)findViewById(R.id.add);
44          Button queryAllButton = (Button)findViewById(R.id.query_all);
45          Button clearButton = (Button)findViewById(R.id.clear);
46          Button deleteAllButton = (Button)findViewById(R.id.delete_all);
47
48          Button queryButton = (Button)findViewById(R.id.query);
49          Button deleteButton = (Button)findViewById(R.id.delete);
50          Button updateButton = (Button)findViewById(R.id.update);
51
52
53          addButton.setOnClickListener(addButtonListener);
54          queryAllButton.setOnClickListener(queryAllButtonListener);
55          clearButton.setOnClickListener(clearButtonListener);
```

8.4 数据分享

- **8.4.4 示例**
  - ContentResolverDemoActivity.java文件代码

```
56          deleteAllButton.setOnClickListener(deleteAllButtonListener);
57
58          queryButton.setOnClickListener(queryButtonListener);
59          deleteButton.setOnClickListener(deleteButtonListener);
60          updateButton.setOnClickListener(updateButtonListener);
61
62          resolver = this.getContentResolver();
63
64      }
65
66
67      OnClickListener addButtonListener = new OnClickListener() {
68          @Override
69          public void onClick(View v) {
```

8.4 数据分享

- **8.4.4 示例**
  - ContentResolverDemoActivity.java文件代码

```
70              ContentValues values = new ContentValues();
71
72              values.put(People.KEY_NAME, nameText.getText().toString());
73              values.put(People.KEY_AGE, Integer.parseInt(ageText.getText().toString()));
74              values.put(People.KEY_HEIGHT,
        Float.parseFloat(heightText.getText().toString()));
75
76              Uri newUri = resolver.insert(People.CONTENT_URI, values);
77
78              labelView.setText("添加成功，URI:" + newUri);
79
80          }
81      };
82
```

8.4 数据分享

- **8.4.4 示例**
  - ContentResolverDemoActivity.java文件代码

```
83      OnClickListener queryAllButtonListener = new OnClickListener() {
84          @Override
85          public void onClick(View v) {
86              Cursor cursor = resolver.query(People.CONTENT_URI,
87                      new String[] { People.KEY_ID, People.KEY_NAME, People.KEY_AGE,
        People.KEY_HEIGHT},
88                      null, null, null);
89              if (cursor == null){
90                  labelView.setText("数据库中没有数据");
91                  return;
92              }
93              labelView.setText("数据库：" + String.valueOf(cursor.getCount()) + "条记录");
94
95              String msg = "";
```

8.4 数据分享

- **8.4.4 示例**
  - ContentResolverDemoActivity.java文件代码

```
96            if (cursor.moveToFirst()){
97              do{
98                msg += "ID：" + cursor.getInt(cursor.getColumnIndex(People.KEY_ID)) + "，
      ";
99                msg += "姓名：" +
      cursor.getString(cursor.getColumnIndex(People.KEY_NAME))+ "，";
100               msg += "年龄：" + cursor.getInt(cursor.getColumnIndex(People.KEY_AGE))
      + "，  ";
101               msg += "身高：" +
      cursor.getFloat(cursor.getColumnIndex(People.KEY_HEIGHT)) + "\n";
102               }while(cursor.moveToNext());
103             }
104           displayView.setText(msg);
105         }
```

8.4 数据分享

- **8.4.4 示例**
  - ContentResolverDemoActivity.java文件代码

```
106       };
107
108       OnClickListener clearButtonListener = new OnClickListener() {
109
110          @Override
111          public void onClick(View v) {
112             displayView.setText("");
113          }
114       };
115
116       OnClickListener deleteAllButtonListener = new OnClickListener() {
117          @Override
118          public void onClick(View v) {
119             resolver.delete(People.CONTENT_URI, null, null);
```

8.4 数据分享

- **8.4.4 示例**
  - ContentResolverDemoActivity.java文件代码

```
120             String msg = "数据全部删除" ;
121             labelView.setText(msg);
122         }
123     };
124
125     OnClickListener queryButtonListener = new OnClickListener() {
126         @Override
127         public void onClick(View v) {
128             Uri uri = Uri.parse(People.CONTENT_URI_STRING + "/" +
    idEntry.getText().toString());
129             Cursor cursor = resolver.query(uri,
130                 new String[] { People.KEY_ID, People.KEY_NAME, People.KEY_AGE,
    People.KEY_HEIGHT},
131                     null, null, null);
```

8.4 数据分享

- **8.4.4 示例**
  - ContentResolverDemoActivity.java文件代码

```
132             if (cursor == null){
133                 labelView.setText("数据库中没有数据");
134                 return;
135             }
136
137             String msg = "";
138             if (cursor.moveToFirst()){
139                 msg += "ID：" + cursor.getInt(cursor.getColumnIndex(People.KEY_ID)) + "，
        ";
140                 msg += "姓名：" +
    cursor.getString(cursor.getColumnIndex(People.KEY_NAME))+ "，";
141                 msg += "年龄：" + cursor.getInt(cursor.getColumnIndex(People.KEY_AGE))
    + "，  ";
```

8.4 数据分享

• 8.4.4 示例

- ContentResolverDemoActivity.java文件代码

```
142                 msg += "身高：" +
    cursor.getFloat(cursor.getColumnIndex(People.KEY_HEIGHT)) + "\n";
143               }
144
145             labelView.setText("数据库：");
146             displayView.setText(msg);
147           }
148       };
149
150       OnClickListener deleteButtonListener = new OnClickListener() {
151           @Override
152           public void onClick(View v) {
153
```

8.4 数据分享

- **8.4.4 示例**
  - ContentResolverDemoActivity.java文件代码

```
154             Uri uri = Uri.parse(People.CONTENT_URI_STRING + "/" +
    idEntry.getText().toString());
155             int result = resolver.delete(uri, null, null);
156             String msg = "删除ID为"+idEntry.getText().toString()+"的数据" + (result>0?"成
    功":"失败");
157             labelView.setText(msg);
158         }
159     };
160
161     OnClickListener updateButtonListener = new OnClickListener() {
162         @Override
163         public void onClick(View v) {
164             ContentValues values = new ContentValues();
165
```

8.4 数据分享

- **8.4.4 示例**
  - ContentResolverDemoActivity.java文件代码

```
166             values.put(People.KEY_NAME, nameText.getText().toString());
167             values.put(People.KEY_AGE, Integer.parseInt(ageText.getText().toString()));
168             values.put(People.KEY_HEIGHT, Float.parseFloat(heightText.getText().toString()));
169
170             Uri uri = Uri.parse(People.CONTENT_URI_STRING + "/" +
        idEntry.getText().toString());
171             int result = resolver.update(uri, values, null, null);
172
173             String msg = "更新ID为"+idEntry.getText().toString()+"的数据" + (result>0?"成功":"
        失败");
174             labelView.setText(msg);
175         }
176     };
177 }
```

8.4 数据分享

- **8.4.4 示例**
  - ContentProviderDemo示例的AndroidManifest.xml文件内容

```
1   <?xml version="1.0" encoding="utf-8"?>
2   <manifest xmlns:android="http://schemas.android.com/apk/res/android"
3       package="edu.hrbeu.ContentProviderDemo"
4       android:versionCode="1"
5       android:versionName="1.0">
6       <application android:icon="@drawable/icon" android:label="@string/app_name">
7           <provider android:name = ".PeopleProvider"
8       android:authorities = "edu.hrbeu.peopleprovider"/>
9       </application>
10      <uses-sdk android:minSdkVersion="3" />
11  </manifest>
```

8.4 数据分享

• 8.4.4 示例

- ContentResolverDemo示例的AndroidManifest.xml文件内容

```
1   <?xml version="1.0" encoding="utf-8"?>
2   <manifest xmlns:android="http://schemas.android.com/apk/res/android"
3       package="edu.hrbeu.ContentResolverDemo"
4       android:versionCode="1"
5       android:versionName="1.0">
6     <application android:icon="@drawable/icon" android:label="@string/app_name">
7       <activity android:name=".ContentResolverDemo"
8           android:label="@string/app_name">
9         <intent-filter>
10          <action android:name="android.intent.action.MAIN" />
11          <category android:name="android.intent.category.LAUNCHER" />
12        </intent-filter>
13      </activity>
14    </application>
15    <uses-sdk android:minSdkVersion="3" />
16  </manifest>
```

习题：

- **1.应用程序一般允许用户自己定义配置信息，如界面背景颜色、字体大小和字体颜色等，尝试使用SharedPreferences保存用户的自定义配置信息，并在程序启动时自动加载这些自定义的配置信息。**
- **2.尝试把第1题的用户自己定义配置信息，以INI文件的形式保存在内部存储器上。**
- **3.简述在嵌入式系统中使用SQLite数据库的优势。**
- **4.分别使用手动建库和代码建库的方式，创建名为test.db数据库，并建立staff数据表，表内的属性值如下表所示：**

属性	数据类型	说明
_id	integer	主键
name	text	姓名
sex	text	性别
department	text	所在部门
salary	float	工资

习题：

5.利用第4题所建立的数据库和staff表，为程序提供添加、删除和更新等功能，并尝试将下表中的数据添加到staff表中。

_id	name	sex	department	salary
1	Tom	male	computer	5400
2	Einstein	male	computer	4800
3	Lily	female	1.68	5000
4	Warner	male		
5	Napoleon	male		

6.建立一个ContentProvider，用来共享第4题所建立的数据库。