
第六章 数据存储与访问

北京邮电大学 计算机学院

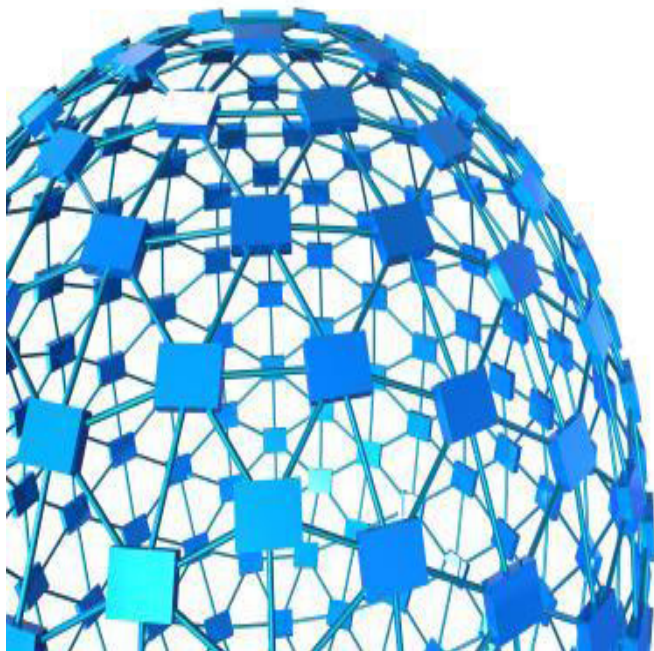
刘伟

w.liu@foxmail.com

■ 本章学习目标

- 掌握各种文件存储的区别与适用情况
- 掌握SharedPreferences的使用方法
- 了解SQLite数据库的特点和体系结构
- 掌握SQLite数据库的建立和操作方法

6.1 数据存储简介



- Android提供了以下 三种
数据存储方式：
 - 文件存储
 - SharedPreferences存储
 - SQLite数据库存储

■ 6.2 文件存储

- 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
- 可以访问保存在资源目录中的原始文件和XML文件
- 可以将文件保存在TF卡等外部存储设备中

■ 6.2 文件存储

■ 6.2.1 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()`

■ 6.2 文件存储

■ 6.2.1 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

■ 6.2 文件存储

■ 6.2.1 内部存储

□ openFileOutput()函数

■ 四种文件操作模式

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。
MODE_WORLD_READABLE	全局读模式，允许任何程序读取私有文件。
MODE_WORLD_WRITEABLE	全局写模式，允许任何程序写入私有文件。

■ 6.2 文件存储

■ 6.2.1 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

■ 6.2 文件存储

■ 6.2.1 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

6.2 文件存储

6.2.1 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    }
```

■ 6.2 文件存储

■ 6.2.1 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

■ 6.2 文件存储

■ 6.2.1 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.bupt.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

■ 6.2 文件存储

■ 6.2.1 内部存储

□ InternalFileDemo用户界面图



■ 6.2 文件存储

■ 6.2.1 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());
```

6.2 文件存储

6.2.1 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();
```

■ 6.2 文件存储

■ 6.2.1 内部存储

```
28         }
29         }
30     }
31     }
32 };
33 OnClickListener readButtonListener = new OnClickListener()
34 {
35     @Override
36     public void onClick(View v) {
37         displayView.setText("");
38         FileInputStream fis = null;
39         try {
40             fis = openFileInput(FILE_NAME);
41             if (fis.available() == 0){
42                 return;
43             }
44         }
45     }
46 }
```


6.2 文件存储

6.2.1 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 };
```

■ 6.2 文件存储

■ 6.2.1 内部存储

- 程序运行后，在
/data/data/edu.bupt.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”

└─ folder edu.bupt.InternalFileDemo		2015-04-22	21:13	drwxr-x--x	
└─ folder cache		2015-04-22	21:13	drwxrwx--x	
└─ folder files		2015-04-22	21:13	drwxrwx--x	
└─ file fileDemo.txt	26	2015-04-22	21:14	-rw-rw----	
└─ file lib		2015-04-22	21:13	lrwxrwxrwx	-> /data/a...

■ 6.2 文件存储

■ 6.2.2 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



■ 6.2 文件存储

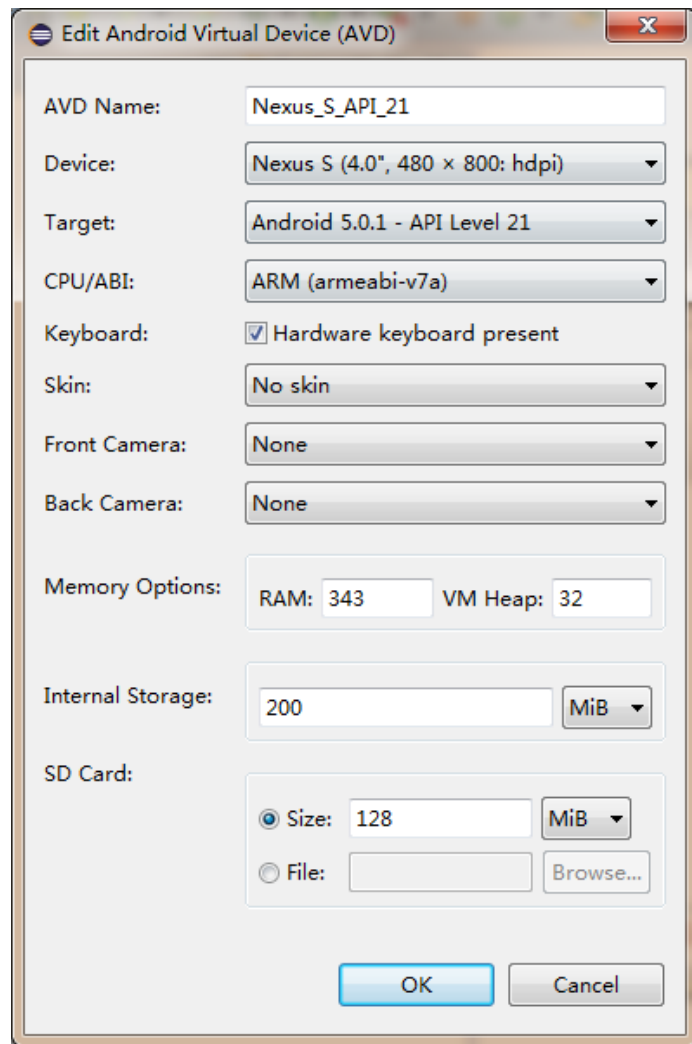
■ 6.2.2 外部存储

- 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

6.2 文件存储

6.2.2 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



■ 6.2 文件存储

■ 6.2.2 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

■ 6.2 文件存储

■ 6.2.2 外部存储

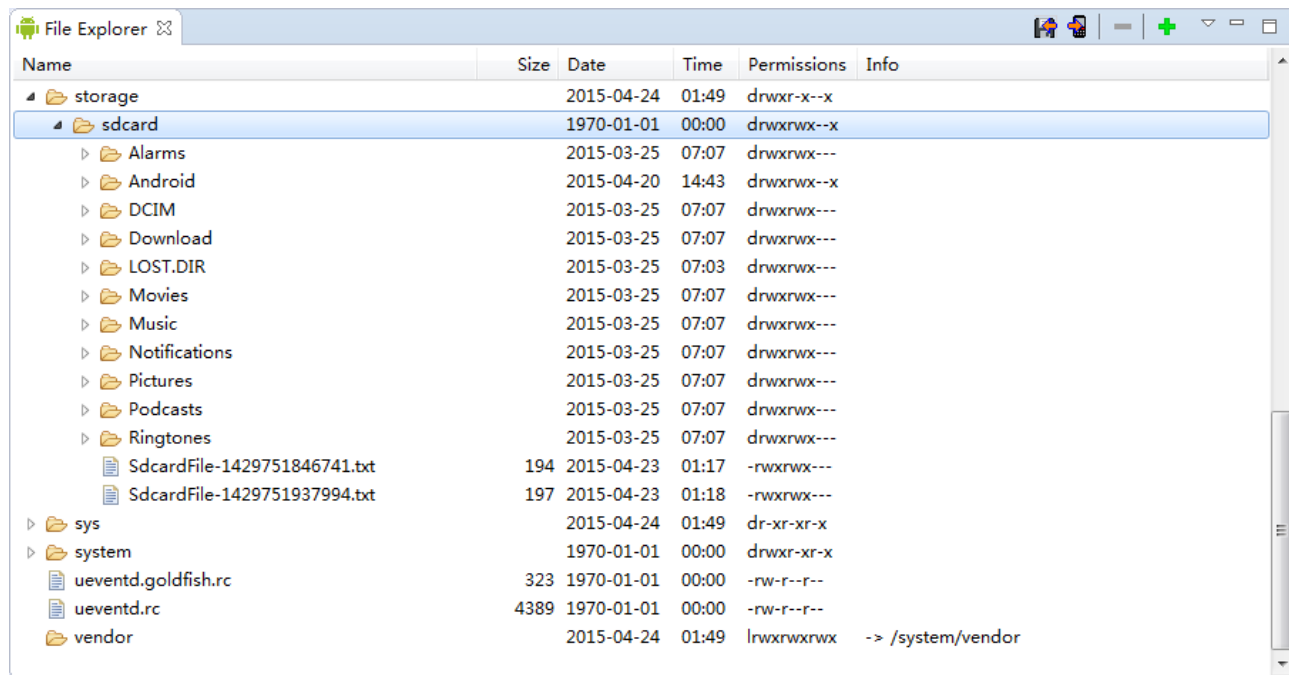
□ SDcardFileDemo用户界面图



6.2 文件存储

6.2.2 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：



■ 6.2 文件存储

■ 6.2.2 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：

6.2 文件存储

6.2.2 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-
6            "+System.currentTimeMillis()+".txt";
7            File dir = new File("/sdcard/");
8            if (dir.exists() && dir.canWrite()) {
9                File newFile = new File(dir.getAbsolutePath() + "/" +
10                fileName);
11                FileOutputStream fos = null;
12                try {
13                    newFile.createNewFile();
14                    if (newFile.exists() && newFile.canWrite()) {
15                        fos = new FileOutputStream(newFile);
16                        fos.write(randomNumbersString.getBytes());
17                        TextView labelView = (TextView)findViewById(R.id.label);
18                        labelView.setText(fileName + "文件写入SD卡");
```

■ 6.2 文件存储

■ 6.2.2 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 };
```

■ 6.2 文件存储

■ 6.2.2 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>
```

■ 6.3 简单存储

■ 6.3.1 SharedPreferences

- 轻量级的数据保存方式
- KVP（Key/Value 键/值对）保存在Android的文件系统中
- 完全屏蔽了对文件系统的操作过程
- 通过调用SharedPreferences中的函数就可以实现对KVP的保存和读取

■ 6.3 简单存储

■ 6.3.1 SharedPreferences

□ 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

6.3 简单存储

6.3.1 SharedPreferences访问方式

- 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_WORLD_READABLE +  
Context.MODE_WORLD_WRITEABLE;
```

■ 6.3 简单存储

■ 6.3.1 SharedPreferences

- 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

- 1 `public static final String PREFERENCE_NAME = "SaveSetting";`

- 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);`

6.3 简单存储

6.3.1 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();
```

6.3 简单存储

6.3.1 SharedPreferences

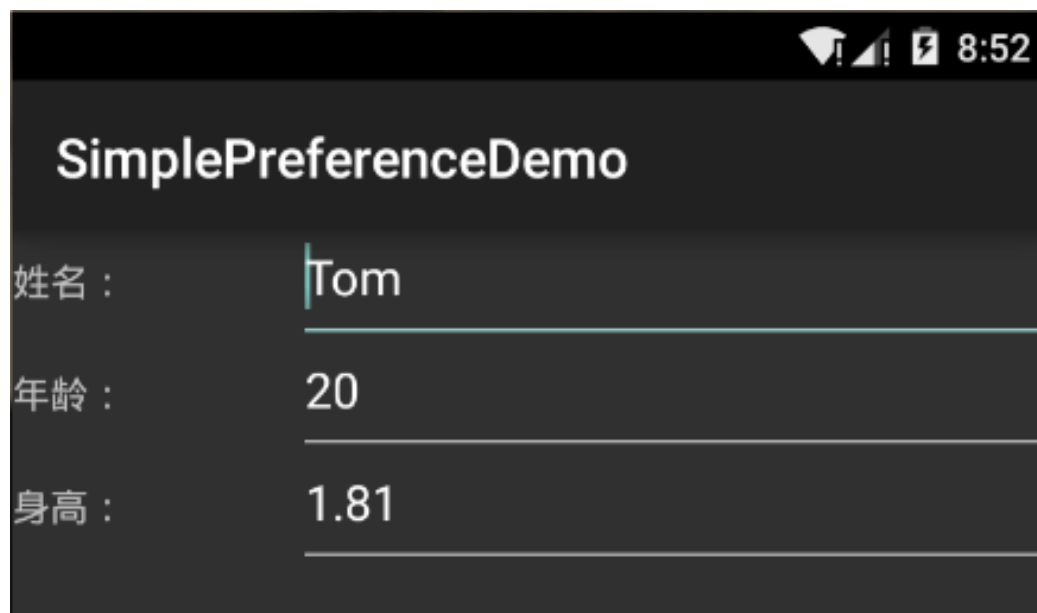
- 读取数据：调用getSharedPreferences()函数，并在函数第1个参数中指明需要访问的SharedPreferences名称，最后通过get<Type>()函数获取保存在SharedPreferences中的KVP
- get<Type>()函数的第1个参数是KVP的名称
- 第2个参数是在无法获取到数值的时候使用的缺省值

```
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);
```

■ 6.3 简单存储

■ 6.3.2 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



6.3 简单存储

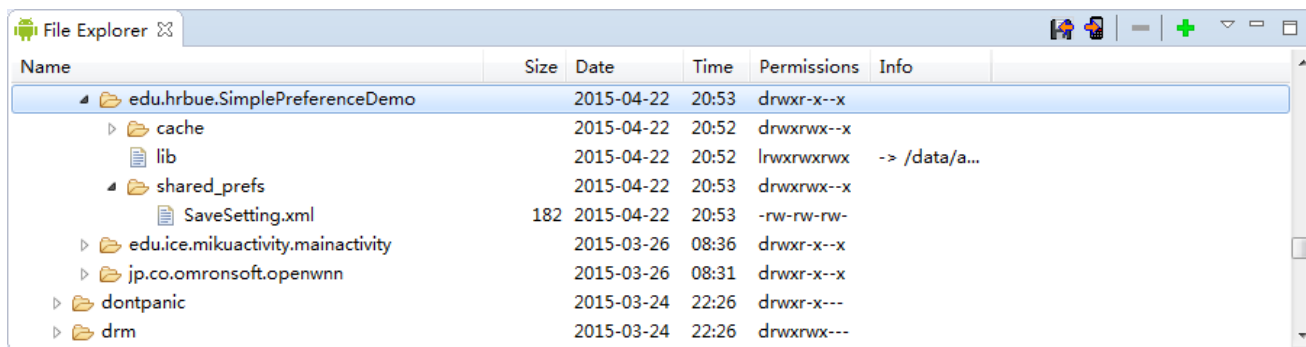
6.3.2 示例

- 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

6.3 简单存储

6.3.2 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.bupt.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”



Name	Size	Date	Time	Permissions	Info
edu.bupt.SimplePreferenceDemo		2015-04-22	20:53	drwxr-x--x	
cache		2015-04-22	20:52	drwxrwx--x	
lib		2015-04-22	20:52	lrwxrwxrwx	-> /data/a...
shared_prefs		2015-04-22	20:53	drwxrwx--x	
SaveSetting.xml	182	2015-04-22	20:53	-rw-rw-rw-	
edu.ice.mikuactivity.mainactivity		2015-03-26	08:36	drwxr-x--x	
jp.co.omronsoft.openwnn		2015-03-26	08:31	drwxr-x--x	
dontpanic		2015-03-24	22:26	drwxr-x---	
drm		2015-03-24	22:26	drwxrwx---	

6.3 简单存储

6.3.2 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

■ 6.3 简单存储

■ 6.3.2 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>
```

6.3 简单存储

6.3.2 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.bupt.SimplePreferenceDemo;  
2  
3 import android.app.Activity;  
4 import android.content.Context;  
5 import android.content.SharedPreferences;  
6 import android.os.Bundle;
```


6.3 简单存储

6.3.2 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 +
16         Context.MODE_WORLD_WRITEABLE;
17
18     @Override
19     public void onCreate(Bundle savedInstanceState) {
20         super.onCreate(savedInstanceState);
21         setContentView(R.layout.main);
22         nameText = (EditText)findViewById(R.id.name);
```

6.3 简单存储

6.3.2 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
```

6.3 简单存储

6.3.2 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();
```

6.3 简单存储

6.3.2 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 }
```

■ 6.4 数据库存储

■ 6.4.1 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

■ 6.4 数据库存储

■ 6.4.1 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复杂性
- 客户端和服务端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

■ 6.4 数据库存储

■ 6.4.1 SQLite简介

- SQLite数据库具有以下几个特征：
 - 轻量级
 - 独立
 - 操作简单
 - 便于管理和维护
 - 可移植性
 - 语言无关
 - 事务性

■ 6.4 数据库存储

■ 6.4.2 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

■ 6.4 数据库存储

■ 6.4.2 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

■ 6.4 数据库存储

■ 6.4.2 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>
```

■ 6.4 数据库存储

■ 6.4.2 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  #
```

6.4 数据库存储

6.4.2 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 #
```

6.4 数据库存储

6.4.2 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>
```

6.4 数据库存储

6.4.2 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>
```

6.4 数据库存储

6.4.2 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  peopleinfo
3  sqlite>
```

6.4 数据库存储

6.4.2 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

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


6.4 数据库存储

6.4.2 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

6.4 数据库存储

6.4.2 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>
```

6.4 数据库存储

6.4.2 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>
```

6.4 数据库存储

6.4.2 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>
```

6.4 数据库存储

6.4.2 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>
```

6.4 数据库存储

6.4.2 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，也可以参考下表

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

■ 6.4 数据库存储

■ 6.4.2 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

■ 6.4 数据库存储

■ 6.4.3 代码建库

- 在代码中动态建立数据库是比较常用的方法
- 例如在程序运行过程中，当需要进行数据库操作时，应用程序会首先尝试打开数据库，此时如果数据库并不存在，程序则会自动建立数据库，然后再打开数据库
- 在编程实现时，一般将所用对数据库的操作都封装在一个类中，因此只要调用这个类，就可以完成对数据库的添加、更新、删除和查询等操作

6.4 数据库存储

6.4.3 代码建库-SQLiteDatabase类

- 打开数据库的方法：
 - ❑ `openDatabase(String path, SQLiteDatabase.CursorFactory factory, int flags)`: 打开`path`所指定的SQLite数据库
 - ❑ `openOrCreateDatabase(String path, SQLiteDatabase.CursorFactory factory)`: 打开或创建（如果文件不存在）`path`所指定的SQLite数据库
 - ❑ `openOrCreateDatabase(File file, SQLiteDatabase.CursorFactory factory)`: 打开或创建（如果文件不存在）`file`所指定的SQLite数据库

6.4 数据库存储

6.4.3 代码建库-SQLiteDatabase类

• SQLiteDatabase常用操作方法

方法	功能描述
insert(String table,String nullColumnHack,ContentValues values)	插入一条记录
delete(String table,String whereClause,String[] whereArgs)	删除一条记录
query (boolean distinct, String table, String[] columns, String selection, String[] selectionArgs, String groupBy, String having, String orderBy, String limit)	查询记录
update(String table,ContentValues value,String whereClause, String[] whereArgs)	修改记录
execSQL(String sql)	执行一条SQL语句
rawQuery(String sql,String[] selectionArgs)	执行带占位符的SQL查询
beginTransaction()	开始事务
endTransaction()	结束事务
close()	关闭数据库

6.4 数据库存储

6.4.3 代码建库-SQLite数据库的创建和删除

● 创建或打开数据库

- 使用openDatabase()方法打开指定的数据库时，需要三个参数：
 - ✓ path用于指定数据库的路径，若指定的数据库不存在，则抛出FileNotFoundException异常
 - ✓ factory用于构造查询时的游标，若factory为null，则表示使用默认的factory构造游标
 - ✓ flags指定了数据库打开的模式，SQLite定义了4种数据库打开模式：
 - OPEN_READONLY (只读)
 - OPEN_READWRITE (可读可写)
 - CREATE_IF_NECESSARY (若数据库不存在先创建数据库)
 - NO_LOCALIZED_COLLATORS (不按照本地化语言对数据进行排序)

```
SQLiteDatabase sqLiteDatabase = SQLiteDatabase  
    .openDatabase("qst_Student.db", null, NO_LOCALIZED_COLLATORS);
```

● 使用openOrCreateDatabase()方法打开或创建指定的数据库

```
SQLiteDatabase sqLiteDatabase = SQLiteDatabase  
    .openOrCreateDatabase ("qst_Student.db", null);
```

6.4 数据库存储

6.4.3 代码建库-SQLite数据库的创建和删除

- 删除数据库

- 使用deleteDatabase()方法删除数据库

```
deleteDatabase("qst_Student.db"); //删除数据库qst_Student.db
```

- 关闭数据库

- 使用close()方法关闭数据库

```
sqliteDatabase.close(); //关闭数据库，sqliteDatabase是一个实例对象
```

6.4 数据库存储

6.4.3 代码建库-表的创建和删除

- **创建表**

- 使用execSQL()方法创建表

```
//创建表的SQL语句
```

```
String sql= "CREATE TABLE student(ID INTEGER PRIMARY KEY, age  
        INTEGER,name TEXT)";
```

```
//执行该SQL语句创建表
```

```
sqliteDatabase.execSQL(sql);
```

- **删除表**

- 使用execSQL()方法删除表

```
//创建表的SQL语句
```

```
String sql= "CREATE TABLE student(ID INTEGER PRIMARY KEY, age  
        INTEGER,name TEXT)";
```

```
//执行该SQL语句创建表
```

```
sqliteDatabase.execSQL(sql);
```

6.4 数据库存储

6.4.3 代码建库-记录的插入、修改和删除

- 插入记录

- 【语法】

```
insert(String table,String nullColumnHack,ContentValues values)
```

- 使用insert()方法插入记录

```
ContentValues contentValues = new ContentValues();  
contentValues.put("ID", 1);  
contentValues.put("age", 26);  
contentValues.put("name", "StudentA");  
//调用insert()方法将contentValues对象封装的数据插入到student表中  
sqliteDatabase.insert("student" , null, contentValues);
```

6.4 数据库存储

6.4.3 代码建库-记录的插入、修改和删除

- 使用execSQL()方法插入记录

//定义插入SQL语句

```
String sql= "INSERT INTO student (ID,age,name) values (1, 26,'StudentA')";
```

//调用execSQL()方法执行SQL语句，将数据插入到student表中

```
sqliteDatabase.execSQL(sql);
```

● 插入记录

- 【语法】

```
delete(String table,String whereClause,String[] whereArgs)
```

- 第1个参数table是需要删除数据的表名称
- 第2个参数whereClause是删除条件
- 第3个参数whereArgs是删除条件所需的参数数组

6.4 数据库存储

6.4.3 代码建库-记录的插入、修改和删除

- 使用delete()方法删除记录

```
sqliteDatabase.delete("student","name=?",new String[]{"StudentA"});
```

```
//定义更新SQL语句
```

```
String sql= "DELETE FORM student where name='StudentA'";
```

```
//调用execSQL()方法执行SQL语句更新student表中的记录
```

```
sqliteDatabase.execSQL(sql);
```


6.4 数据库存储

6.4.3 代码建库-数据查询与Cursor接口

- 使用SQLiteDatabase的query()方法可以查询记录

- 【语法】

```
public Cursor query (boolean distinct, String table, String[] columns,  
                    String selection, String[] selectionArgs, String groupBy, String having,  
                    String orderBy, String limit);
```

- distinct是一个可选的布尔值，用来说明返回的值是否只包含唯一的值
- table是表名称
- columns是由列名称构成的数组
- selection是条件where子句，可以包含 “?” 通配符，在子句中用作占位符
- selectionArgs是参数数组，替换where子句中的 “?” 占位符
- groupBy表示分组列

6.4 数据库存储

6.4.3 代码建库

- 下面内容是DBAdapter类的部分代码，封装了数据库的建立、打开和关闭等操作

```
1 public class DBAdapter {  
2     private static final String DB_NAME = "people.db";  
3     private static final String DB_TABLE = "peopleinfo";  
4     private static final int DB_VERSION = 1;  
5  
6     public static final String KEY_ID = "_id";  
7     public static final String KEY_NAME = "name";  
8     public static final String KEY_AGE = "age";  
9     public static final String KEY_HEIGHT = "height";  
10  
11     private SQLiteDatabase db;
```

6.4 数据库存储

6.4.3 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,
23      DB_VERSION);
24      try {
25          db = dbOpenHelper.getWritableDatabase();
26      } catch (SQLException e) {
27          // TODO: Handle exception
28      }
29  }
```

■ 6.4 数据库存储

■ 6.4.3 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 }
```

■ 6.4 数据库存储

■ 6.4.3 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

■ 6.4 数据库存储

■ 6.4.3 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

■ 6.4 数据库存储

■ 6.4.3 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

6.4 数据库存储

6.4.3 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
```


6.4 数据库存储

6.4.3 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 }
```

■ 6.4 数据库存储

■ 6.4.3 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

■ 6.4 数据库存储

■ 6.4.3 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

6.4 数据库存储

6.4.3 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 }
```

■ 6.4 数据库存储

■ 6.4.4 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

6.4 数据库存储

6.4.4 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 }
```

■ 6.4 数据库存储

■ 6.4.4 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 }
```

6.4 数据库存储

6.4.4 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 }
```


■ 6.4 数据库存储

■ 6.4.4 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

■ 6.4 数据库存储

■ 6.4.4 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

■ 6.4 数据库存储

■ 6.4.4 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

■ 6.4 数据库存储

■ 6.4.4 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  }
```

■ 6.4 数据库存储

■ 6.4.4 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 }
```

■ 6.4 数据库存储

■ 6.4.4 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

■ 6.4 数据库存储

■ 6.4.4 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 }
```

■ 6.4 数据库存储

■ 6.4.4 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

■ 6.4 数据库存储

■ 6.4.4 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

6.4 数据库存储

6.4.4 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

■ 6.4 数据库存储

■ 6.4.4 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

6.4 数据库存储

6.4.4 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 }
```

■ 6.4 数据库存储

■ 6.4.4 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

■ 6.4 数据库存储

■ 6.4.4 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,  
3         KEY_AGE, KEY_HEIGHT}, KEY_ID + "=" + id, null, null, null, null);  
4     return ConvertToPeople(results);  
5 }
```

■ 6.4 数据库存储

■ 6.4.4 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 }
```