



数据存储





学习目标

- 1、掌握sharedPreferences的使用方法；
- 2、掌握各种文件存储的区别与适用情况；
- 3、SQLite数据库的使用；
- 4、ContentProvider的使用。





数据存取

Android有4种数据存取的方式：

- SharedPreferences

轻量级**NVP** (Name/Value Pair, 名称/值对) 方式存储, 以**XML**的文件方式保存；

- 文件

采用**java.io.***库提供的I/O接口读写文件；

- SQLite数据库

轻量级嵌入式内置**数据库**；

- ContentProvider

封装各种数据源（文件、数据库、网络），**共享**给多个应用。





9.1 SharedPreferences存储

应用场景：

- 保存播放位置：用手机播放器播放音乐，我们希望重启播放器时，播放器能从上次停止的那首曲目开始播放，如何实现？
- 自动登录：记住登录用户名（密码）
- 其他应用？





9.1 SharedPreferences存储

什么是SharedPreferences？

- 一种轻量级的数据保存方式。
- 类似于我们常用的ini文件，用来保存应用程序的一些属性设置、较简单的参数设置。
- 保存现场：保存用户所作的修改或者自定义参数设定，当再次启动程序后回复上次退出时的状态。





9.1 SharedPreferences存储

什么是SharedPreferences ?

- 将**NVP** (Name/Value Pair, 名称/值对) 保存在Android的文件系统中 (XML文件), 完全屏蔽的对文件系统的操作过程。
 - NVP举例: (姓名,张三), (性别,男), (年龄,30), ...
- 开发人员仅是通过调用SharedPreferences的API对NVP进行保存和读取.





9.1 SharedPreferences存储

什么是SharedPreferences ?

- 除数据保存，还提供数据共享功能。
 - 主要支持3种数据访问模式（读写权限）
 - 私有（MODE_PRIVATE）：仅创建程序可读、写
 - 全局读（MODE_WORLD_READABLE）：创建程序可读写，其他程序可读不可写
 - 全局写（MODE_WORLD_WRITEABLE）：创建程序和其他程序都可写，但不可读！
- Ps：后两种模式可组合，用+号或|号。





9.1 SharedPreferences存储

使用方法：

— 第1步：定义访问模式

- 下面的代码将访问模式定义为私有模式

```
public static int MODE = MODE_PRIVATE
```

- 访问模式可组合：既可以全局读，也可以全局写，将两种模式组合（+号或|号）成下面的方式：

```
public static int MODE =  
    Context.MODE_WORLD_READABLE +  
    Context.MODE_WORLD_WRITEABLE
```





9.1 SharedPreferences存储

使用方法:

- 第2步：定义SharedPreferences的名称
 - 该名称与Android文件系统中保存的XML文件同名。
 - (保存在:/data/data/<package name>/shared_prefs/)
 - 相同名称的NVP内容，都会保存在同一个文件中。

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





9.1 SharedPreferences存储

使用方法:

- 第3步：创建SharedPreferences对象
 - 将**访问模式**和**名称**作为参数，传递到getSharedPreferences()函数，并获得SharedPreferences对象

```
SharedPreferences sharedPreferences =  
    getSharedPreferences ( PREFERENCE_NAME, MODE );
```





9.1 SharedPreferences存储

使用方法:

- 第4步：修改与保存
 - 通过**SharedPreferences.Editor**类进行修改
 - 调用commit()函数保存修改内容
- 支持数据类型：**整型**、**布尔型**、**浮点型**和**长整型**等

```
SharedPreferences.Editor editor = sharedPreferences.edit();  
editor.putString("Name", "Tom");  
editor.putInt("Age", 20);  
editor.putFloat("Height", (float)163.00 );  
editor.commit();
```





9.1 SharedPreferences存储

使用方法：

- 第5步：读取数据(可紧接第3步)
 - (先调用getSharedPreferences()函数获得对象)
 - 通过get<Type>()函数获取NVP
 - 第1个参数是NVP的名称(Name)
 - 第2个参数是在无法获取到数值的时候使用的缺省值

```
SharedPreferences sharedPreferences =  
    getSharedPreferences(PREFERENCE_NAME, MODE);  
String name = sharedPreferences.getString("Name", "Default Name");  
int age = sharedPreferences.getInt("Age", 20);
```





9.1 SharedPreferences存储

示例：

- 我们已经知道了用Preferences来存取数据，那么这些数据究竟被保存在手机的什么地方了呢？
- 每安装一个应用程序时， SharedPreferences文件就保存在
/data/data/<package name>/shared_prefs/
目录下，其中的就是我们的数据文件。





9.1 SharedPreferences存储

示例：

- 如何读取程序SharedPreferences的数据？
- 可通过DDMS的FileExplorer查看/data/data下的数据，Android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，这些数据包括文件、SharedPreferences文件和数据库等





9.1 SharedPreferences存储

示例：

- 名为edu.hrbeu.SimplePreferenceDemo的程序，其shared_prefs目录生成了一个SaveSetting.xml文件

edu.hrbeu.SimplePreferenceDemo	2009-07-10	02:18	drwxr-xr-x
lib	2009-07-10	02:18	drwxr-xr-x
shared_prefs	2009-07-10	03:01	drwxrwx--x
SaveSetting.xml	170	2009-07-15	08:45 -rw-rw-rw-
edu.hrbeu.SimpleRandomServiceDemo	2009-06-30	12:17	drwxr-xr-x
edu.hrbeu.SpinnerDemo	2009-06-21	07:01	drwxr-xr-x

该文件名取决于之前第2步的代码：

public static final String PREFERENCE_NAME = "SaveSetting";

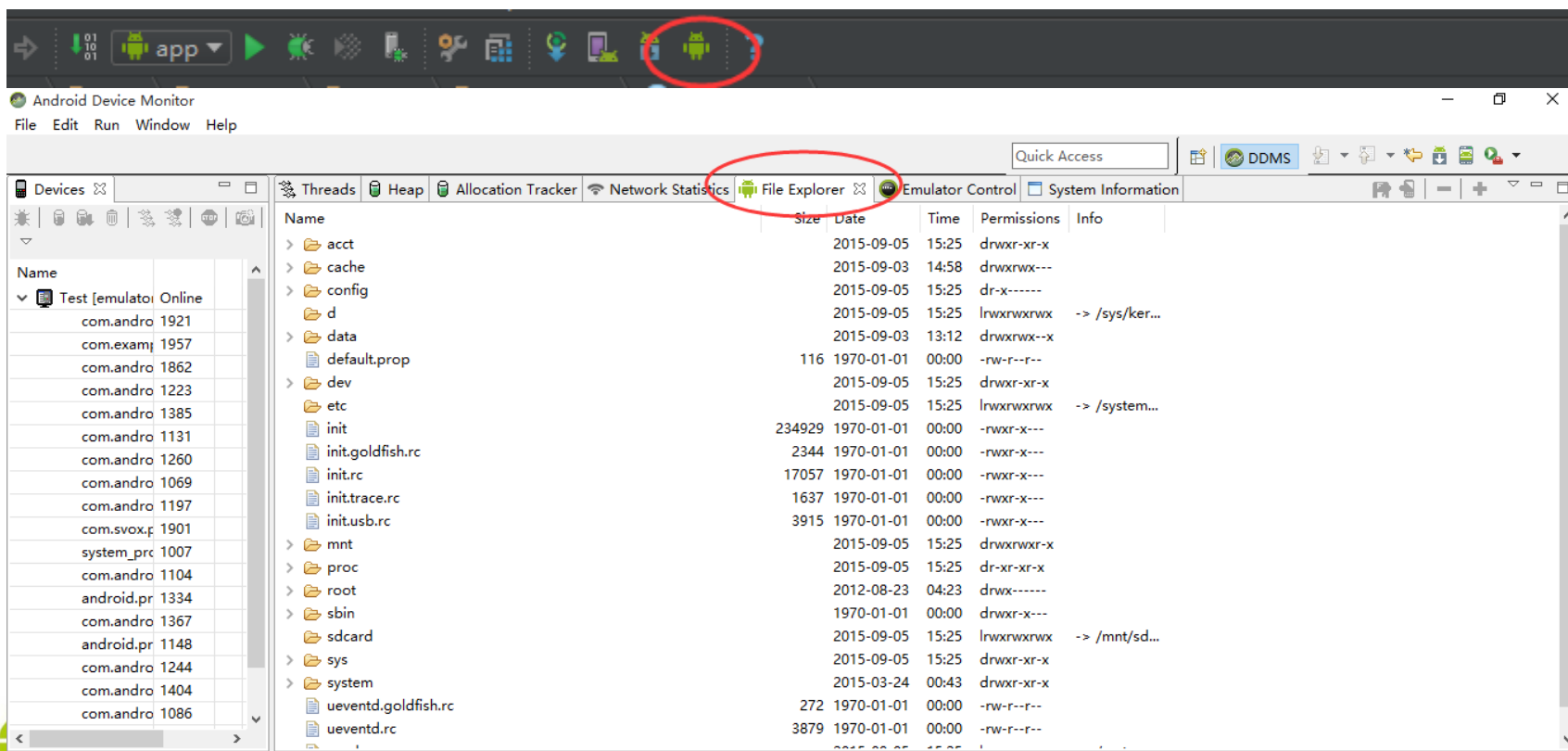
- 这个文件就是保存SharedPreferences的文件，文件大小为170字节，在Linux下的权限为“-rw-rw-rw”





9.1 SharedPreferences存储

示例：DDMS > File Explorer





9.1 SharedPreferences存储

示例：文件权限说明

- 在Linux系统中，文件权限描述符每3个字符分别描述了**创建者（第1-3字符）、同组用户（第4-6字符）和其他用户（第7-9字符）**对文件的操作限制（权限）；
- 每字符意义：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(无操作权限)。
- 例如：“-rw-rw-rw”表示SaveSetting.xml可以被创建者、同组用户和其他用户进行读取和写入操作，但不可执行。





9.1 SharedPreferences存储

示例：文件权限说明

- 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





9.1 SharedPreferences存储

示例：数据文件格式

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

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





9.1 SharedPreferences存储

示例：共享的实现

- 如何读取其他应用程序保存的SharedPreferences数据
- 示例将读取程序名为SimplePreferenceDemo保存的信息，在程序启动时显示在用户界面上

SharePreferenceDemo

姓名：Tom
年龄：20
身高：1.81





9.1 SharedPreferences存储

示例：SharedPreferencesDemo示例的核心代码

```
public static final String PREFERENCE_PACKAGE =  
    "edu.hrbeu.SimplePreferenceDemo";
```

(新) 定义要访问的应用的名称

```
public static int MODE = Context.MODE_WORLD_READABLE +  
    Context.MODE_WORLD_WRITEABLE;  
public static final String PREFERENCE_NAME = "SaveSetting";
```

第1-2步: 定义模式、名称

```
public void onCreate(Bundle savedInstanceState) {  
    Context c = null;  
    try {  
        // 获取SimplePreferenceDemo示例的Context  
        c = this.createPackageContext(PREFERENCE_PACKAGE,  
            Context.CONTEXT_IGNORE_SECURITY);  
    }  
    catch (NameNotFoundException e) {e.printStackTrace();}  
    // 将正确的SharedPreferences名称传递给函数  
    SharedPreferences sharedPreferences =  
        c.getSharedPreferences(PREFERENCE_NAME, MODE);  
    // 读取NVP  
    String name = sharedPreferences.getString("Name", "Tom");  
    int age = sharedPreferences.getInt("Age", 20);  
    float height = sharedPreferences.getFloat("Height",);  
}
```

(新) 创建要访问的应用的上下文

第3步: 得到上下文的SP

第5步: 读NVP





9.1 SharedPreferences存储

示例：访问共享设置的额外步骤

- 第1行定义要访问的应用的包名
- 第8行代码调用了createPackageContext()获取到了要访问的应用的上下文Context
 - 第1个参数是要访问的应用的包名称
 - 第2个参数Context.CONTEXT_IGNORE_SECURITY表示忽略所有可能产生的安全问题。
 - 这段代码可能引发异常，因此必须放在try/catch中。
- 在代码第11行，通过Context得到要访问应用的SharedPreferences对象。
 - （有别于访问自己配置过程的第3步）
 - 同样在getSharedPreferences()函数中，需要将正确的SharedPreferences名称、访问模式传递给函数。





9.1 SharedPreferences存储

示例：访问其他应用程序的**共享设置**必须满足三个条件

- 共享者需要将SharedPreferences的访问模式设置为全局读或全局写
- 访问者需要知道共享者的包名称和SharedPreferences的名称，以通过Context获得SharedPreferences对象
- 访问者需要确切知道每个数据的名称和数据类型，用以正确读取数据





9.2 文件存储

- Android使用的是基于Linux的文件系统
- 程序开发人员可以建立和访问程序自身的私有文件
- 也可以访问保存在资源（res）目录中的原始文件和XML文件
- 还可以在SD卡等外部存储设备中保存与读取文件





9.2 文件存储

- 文件主要用于存储大容量的数据
- 采用java.io.*库所提供的I/O接口读写文件。
- 只有本地文件可以被访问
 - 优点：可以存储大容量的数据
 - 缺点：文件更新或是格式改变可能会导致大量的编程工作





9.2 文件存储

• 9.2.1 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Linux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
- 主要介绍两个函数
 - openFileOutput()
 - openFileInput()





9.2 文件存储

- 9.2.1 内部存储：

- openFileOutput()函数

- 用于写入数据，如果指定的文件不存在，则创建一个新的文件
 - 语法格式

```
public FileOutputStream openFileOutput(String name, int mode)
```

- 第1个参数是文件名称，这个参数不能包含描述路径的斜杠
 - 第2个参数是操作模式
 - 函数的返回值是FileOutputStream类型





9.2 文件存储

- 9.2.1 内部存储：
 - openFileOutput()函数
 - Android系统支持四种文件操作模式

模式	说明
MODE_PRIVATE	私有模式，缺陷模式，文件仅能够被文件创建程序访问，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。





9.2 文件存储

- 9.2.1 内部存储：

- openFileOutput()函数

使用openFileOutput()函数建立新文件的示例代码如下

```
String FILE_NAME = "fileDemo.txt";  
FileOutputStream fos =  
    openFileOutput(FILE_NAME, Context.MODE_PRIVATE);  
String text = "Some data";  
fos.write(text.getBytes());  
fos.flush();  
fos.close();
```

- 第1行代码定义了建立文件的名称fileDemo.txt
 - 第2行代码使用openFileOutput()函数以私有模式建立文件
 - 第4行代码调用write()函数将数据写入文件
 - 第5行代码调用flush()函数将所有剩余的数据写入文件
 - 第6行代码调用close()函数关闭FileOutputStream





9.2 文件存储

- 9.2.1 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，等数据量累积到一定程度时再一次性的写入文件中
 - 由上可知，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





9.2 文件存储

- 9.2.1 内部存储：

- openFileInput()函数

- openFileInput()函数用于打开一个与应用程序联系的私有文件输入流
 - 当文件不存在时抛出FileNotFoundException 异常
 - openFileInput()函数的语法格式如下

```
public FileInputStream openFileInput (String name)
```

- 参数是文件名称，同样不允许包含描述路径的斜杠





9.2 文件存储

- 9.2.1 内部存储：
 - openFileInput()函数
 - openFileInput ()函数打开已有文件的示例代码如下

```
String FILE_NAME = "fileDemo.txt";
FileInputStream fis = openFileInput(FILE_NAME);

byte[] readBytes = new byte[fis.available()];
while(fis.read(readBytes) != -1){
    //对读入的数据进行处理
}
```

- 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





9.2 文件存储

- 9.2.1 内部存储：示例

请看实验七 数据存取（一）

- InternalFileDemo示例用来演示在内部存储器上进行文件写入和读取
- InternalFileDemo示例用户界面如图





9.2 文件存储

- 9.2.1 内部存储：示例核心代码InternalFileDemo (write)

```
OnClickListener writeButtonListener = new OnClickListener() {  
    @Override  
    public void onClick(View v) {  
        FileOutputStream fos = null;  
        try {  
            if (appendBox.isChecked()){//判断是否是以追加的模式写入文件  
                fos = openFileOutput(FILE_NAME, Context.MODE_APPEND);  
            }  
            else {  
                fos = openFileOutput(FILE_NAME, Context.MODE_PRIVATE);  
            }  
            String text = entryText.getText().toString();  
            fos.write(text.getBytes()); //写入  
            labelView.setText("文件写入成功, 写入长度: "+text.length());  
            entryText.setText("");  
        }  
        catch (FileNotFoundException e) {  
            e.printStackTrace();  
        }  
        catch (IOException e) {  
            e.printStackTrace();  
        }  
    }  
}
```





9.2 文件存储

- 9.2.1 内部存储：示例核心代码InternalFileDemo (write)

```
finally{  
    if (fos != null){  
        try {  
            fos.flush(); //将缓冲区内所有的数据写入文件  
            fos.close(); //关闭文件  
        }  
        catch (IOException e) {  
            e.printStackTrace();  
        }  
    }  
}  
};
```





9.2 文件存储

- 9.2.1 内部存储：示例核心代码InternalFileDemo (read)

```
OnClickListener readButtonListener = new OnClickListener() {  
    @Override  
    public void onClick(View v) {  
        displayView.setText("");  
        FileInputStream fis = null;  
        try {  
            fis = openFileInput(FILE_NAME);  
            if (fis.available() == 0) return; // 如果文件为空  
            byte[] readBytes = new byte[fis.available()];  
            while(fis.read(readBytes) != -1){  
                // 读文件!!!Fixme  
            }  
            String text = new String(readBytes);  
            displayView.setText(text);  
            labelView.setText("文件读取成功，文件长度: "+text.length());  
        }  
    }  
}
```





9.2 文件存储

- 9.2.1 内部存储：示例核心代码InternalFileDemo (read)

```
        catch (FileNotFoundException e) {  
            e.printStackTrace();  
        }  
        catch (IOException e) {  
            e.printStackTrace();  
        }  
    }  
};
```





9.2 文件存储

- 9.2.1 内部存储：
 - fileDemo.txt文件

	edu.hrbeu.InternalFileDemo		2009-07-16	02:28	drwxr-xr-x
	files		2009-07-16	02:31	drwxrwx--x
	fileDemo.txt	9	2009-07-16	03:24	-rw-rw----
	lib		2009-07-16	02:28	drwxr-xr-x

- fileDemo.txt从文件权限上进行分析，“-rw-rw- - -”表明文件仅允许文件创建者和同组用户读写，其他用户无权使用
- 文件的大小为9个字节，保存的数据为 “Some data”





9.2 文件存储

- 9.2.2 外部存储：
 - Android的外部存储设备指的是SD卡（Secure Digital Memory Card），是一种广泛使用于数码设备上的记忆卡



- 不是所有的Android手机都有SD卡，但Android系统提供了对SD卡的便捷的访问方法





9.2 文件存储

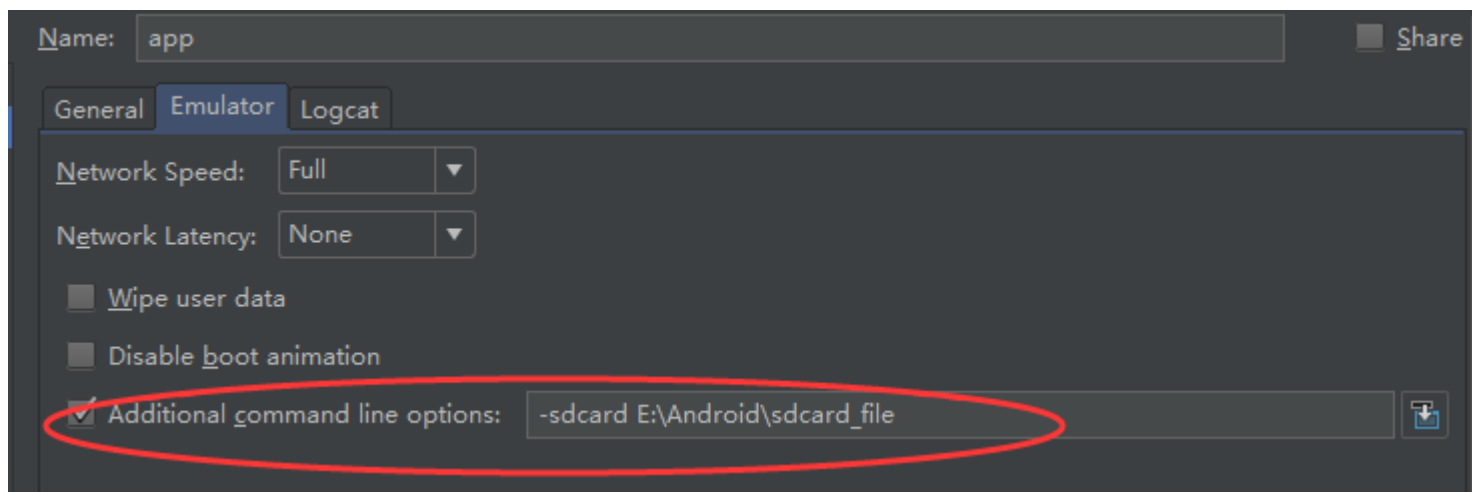
- 9.2.2 外部存储
 - SD卡适用于保存**大尺寸**的文件或者是一些无需设置访问权限的文件，可以保存录制的大容量的视频文件和音频文件等
 - SD卡使用的是FAT（File Allocation Table）的文件系统，**不支持访问模式和权限控制**，但可以通过Linux文件系统的文件访问权限的控制保证文件的私密性
 - Android模拟器支持SD卡，但模拟器中没有缺省的SD卡，开发人员须在模拟器中手工添加SD卡的映像文件





9.2 文件存储

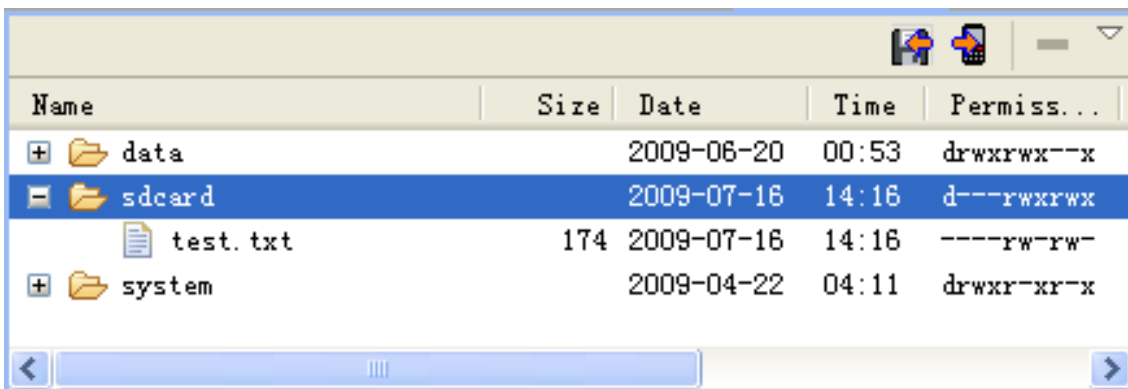
- 9.2.2 外部存储
 - 如果希望Android模拟器启动时能够自动加载指定的SD卡，还需要在模拟器的“运行设置”（Run Configurations）中添加SD卡加载命令
 - SD卡加载命令中只要指明映像文件位置即可
 - SD卡加载命令





9.2 文件存储

- 9.2.2 外部存储
 - 测试SD卡映像是否正确加载
 - 在模拟器启动后，使用FileExplorer向SD卡中随意上传一个文件，如果文件上传成功，则表明SD卡映像已经成功加载
 - 向SD卡中成功上传了一个测试文件test.txt，文件显示在/sdcard目录下



Name	Size	Date	Time	Permiss...
+ data		2009-06-20	00:53	drwxrwx--x
- sdcard		2009-07-16	14:16	d---rwxrwx
test.txt	174	2009-07-16	14:16	----rw-rw-
+ system		2009-04-22	04:11	drwxr-xr-x





9.2 文件存储

- 9.2.2 外部存储
 - 编程访问SD卡
 - 首先需要检测系统的/sdcard目录是否可用
 - 如果不可用，则说明设备中的SD卡已经被移除，在Android模拟器则表明SD卡映像没有被正确加载
 - 如果可用，则直接通过使用标准的Java.io.File类进行访问
 - 将数据保存在SD卡
 - SDcardFileDemo示例说明了如何将数据保存在SD卡





9.2 文件存储

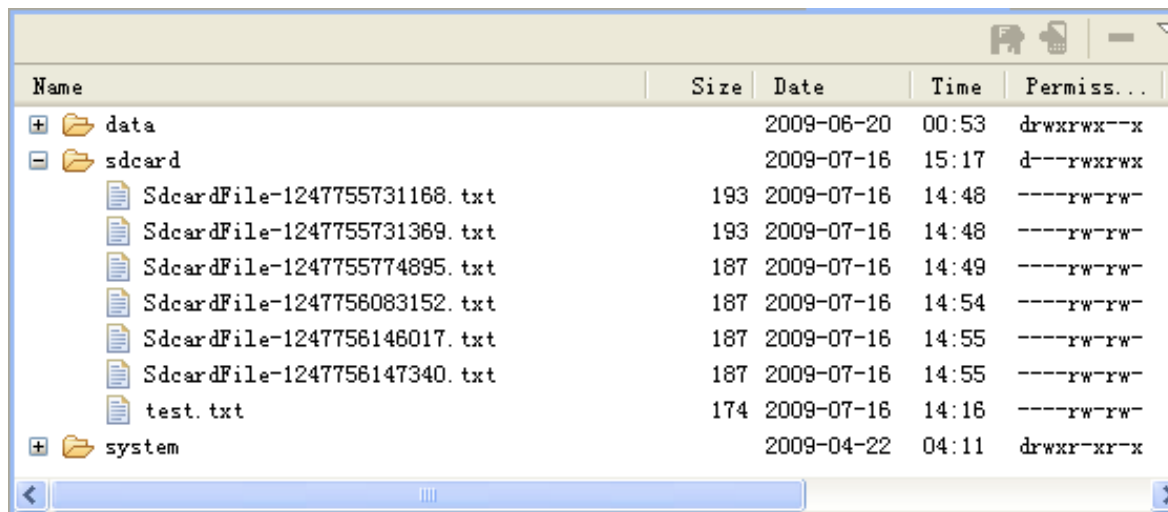
- 9.2.2 外部存储
 - 下图是SDcardFileDemo示例的用户界面
 - 通过“生产随机数列”按钮生产10个随机小数
 - 通过“写入SD卡”按钮将生产的数据保存在SD卡的目录下





9.2 文件存储

- 9.2.2 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
 - SD卡中生产的文件



Name	Size	Date	Time	Permiss...
+ data		2009-06-20	00:53	drwxrwx--x
- sdcard		2009-07-16	15:17	d--rwxrwx
SdcardFile-1247755731168.txt	193	2009-07-16	14:48	----rw-rw-
SdcardFile-1247755731369.txt	193	2009-07-16	14:48	----rw-rw-
SdcardFile-1247755774895.txt	187	2009-07-16	14:49	----rw-rw-
SdcardFile-1247756083152.txt	187	2009-07-16	14:54	----rw-rw-
SdcardFile-1247756146017.txt	187	2009-07-16	14:55	----rw-rw-
SdcardFile-1247756147340.txt	187	2009-07-16	14:55	----rw-rw-
test.txt	174	2009-07-16	14:16	----rw-rw-
+ system		2009-04-22	04:11	drwxr-xr-x





9.2 文件存储

- 9.2.2 外部存储
 - SDcardFileDemo示例与InternalFileDemo示例的核心代码比较相似
 - SDcardFileDemo示例与InternalFileDemo示例的不同之处，在下文中的注释中会加以说明。





9.2 文件存储

- 9.2.2 外部存储：SDcardFileDemo示例核心代码

```
private static String randomNumbersString = "";
OnClickListener writeButtonListener = new OnClickListener() {
    @Override
    public void onClick(View v) {
        //通过当前时间来为文件名命名，避免重复命名
        String fileName = "SdcardFile-"+System.currentTimeMillis()+".txt";
        File dir = new File("/sdcard/");
        //sdcard目录存在性检查
        if (dir.exists() && dir.canWrite()) {
            //使用“绝对目录+文件名”的形式表示新建立的文件
            File newFile = new File(dir.getAbsolutePath() + "/" + fileName);
            FileOutputStream fos = null;
            try {
                newFile.createNewFile();
                //文件存在性和可写入性进行检查
                if (newFile.exists() && newFile.canWrite()) {
                    fos = new FileOutputStream(newFile);
                    fos.write(randomNumbersString.getBytes());
                    TextView labelView = (TextView)findViewById(R.id.label);
                    labelView.setText(fileName + "文件写入SD卡");
                }
            }
        }
    }
}
```





9.2 文件存储

- 9.2.2 外部存储：SDcardFileDemo示例核心代码

```
catch (IOException e) {  
    e.printStackTrace();  
}  
finally {  
    if (fos != null) {  
        try{  
            fos.flush();  
            fos.close();  
        }  
        catch (IOException e) { }  
    }  
}  
}  
};
```





9.2 文件存储

• 9.2.3 资源文件

- 程序开发人员可以将程序开发阶段已经准备好的原始格式文件和XML文件分别存放在`/res/raw`和`/res/xml`目录下, 供应用程序在运行时进行访问
- 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、图像文件和数据文件等等, 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





9.2 文件存储

- 9.2.3 资源文件
 - /res/xml目录下的XML文件，一般用来保存格式化的数据，在应用程序编译和打包时会将XML文件转换为高效的二进制格式，应用程序运行时会以特殊的方式进行访问





9.2 文件存储

- 9.2.3 资源文件：如何读取原始格式文件？
 - 首先调用getResources()函数**获得资源对象**
 - 然后通过调用资源对象的openRawResource()函数，以二进制流的形式**打开文件**
 - 在读取文件结束后，调用close()函数**关闭文件流**

```
InputStream input = this.getResources().openRawResource(R.raw.filename);  
  
byte[] reader = new byte[inputStream.available()];  
while (inputStream.read(reader) != -1) {}  
txt_text.setText(new String(reader, "utf-8")); //获得Context资源  
  
input.close(); //关闭输入流
```





9.2 文件存储

- 9.2.3 资源文件
 - /res/xml目录下的XML文件会转换成高效的二进制格式
 - 在程序运行时读取/res/xml目录下的XML文件
 - 在/res/xml目录下创建一个名为toys.xml的文件
 - XML文件定义了多个<toy>元素，每个<toy>元素都包含2个属性name和price，表示姓名和价格

```
<toys>
<toy name = "Winnie" price = "100" />
<toy name = "Teddy" price = "125" />
<toy name = "BearBear" price = "120" />
</toys>
```





9.2 文件存储

• 9.2.3 资源文件

— 读取XML格式文件

- 首先通过调用资源对象的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)

```
XmlPullParser parser = resources.getXml(R.xml.toys);  
//通过资源对象的getXml()函数获取到XML解析器  
while (parser.next() != XmlPullParser.END_DOCUMENT) {  
    String toys = parser.getName(); //getName()函数获得元素的名称  
    //查看元素名是否匹配  
    if ((toys != null) && toys.equals("toy")) {  
        ..... /*逐个元素地读取属性名和属性值（本例中的toys.xml共有3个  
            元素），并通过分析属性名获取到正确的属性值*/  
    }  
}
```





9.2 文件存储

- 9.2.3 资源文件
 - 如何获取元素个数

```
int count = parser.getAttributeCount();
```

- 如何获得属性名和属性值

```
String attrName = parser.getAttributeName(i); // 获得属性名  
String attrValue = parser.getAttributeValue(i); // 获得属性值
```

- 如何分析已获得的属性名和属性值

```
// 通过分析属性名获取到正确的属性值  
if ((attrName != null) && attrName.equals("name")) {  
    name = attrValue;  
}  
else if ((attrName != null) && attrName.equals("price")) {  
    price = attrValue;  
}
```





9.2 文件存储

- 9.2.3 资源文件
 - XmlPullParser的XML事件类型

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

- 读取文件过程中遇到END_DOCUMENT时停止分析





9.3 数据库存储

- 9.3.1 SQLite数据库
 - SQLite是一个开源的嵌入式关系数据库
 - 在2000年由D. Richard Hipp发布





9.3 数据库存储

- 9.3.1 SQLite数据库
 - SQLite数据库特点
 - 更加适用于嵌入式系统，嵌入到使用它的应用程序中
 - 占用非常少，运行高效可靠，可移植性好
 - 提供了零配置（zero-configuration）运行模式
 - SQLite数据库不仅提高了运行效率，而且屏蔽了数据库使用和管理的复杂性，程序仅需要进行最基本的数据操作，其他操作可以交给进程内部的数据库引擎完成





9.3 数据库存储

- 9.3.1 SQLite数据库
- SQLite 是一个轻量级的软件库
 - 原子量性
 - 坚固性
 - 独立性
 - 耐久性
 - 体积大小只用几千字节
 - 一些SQL 的指令只是部分支持（例如：不支持ALTER TABLE）





9.3 数据库存储

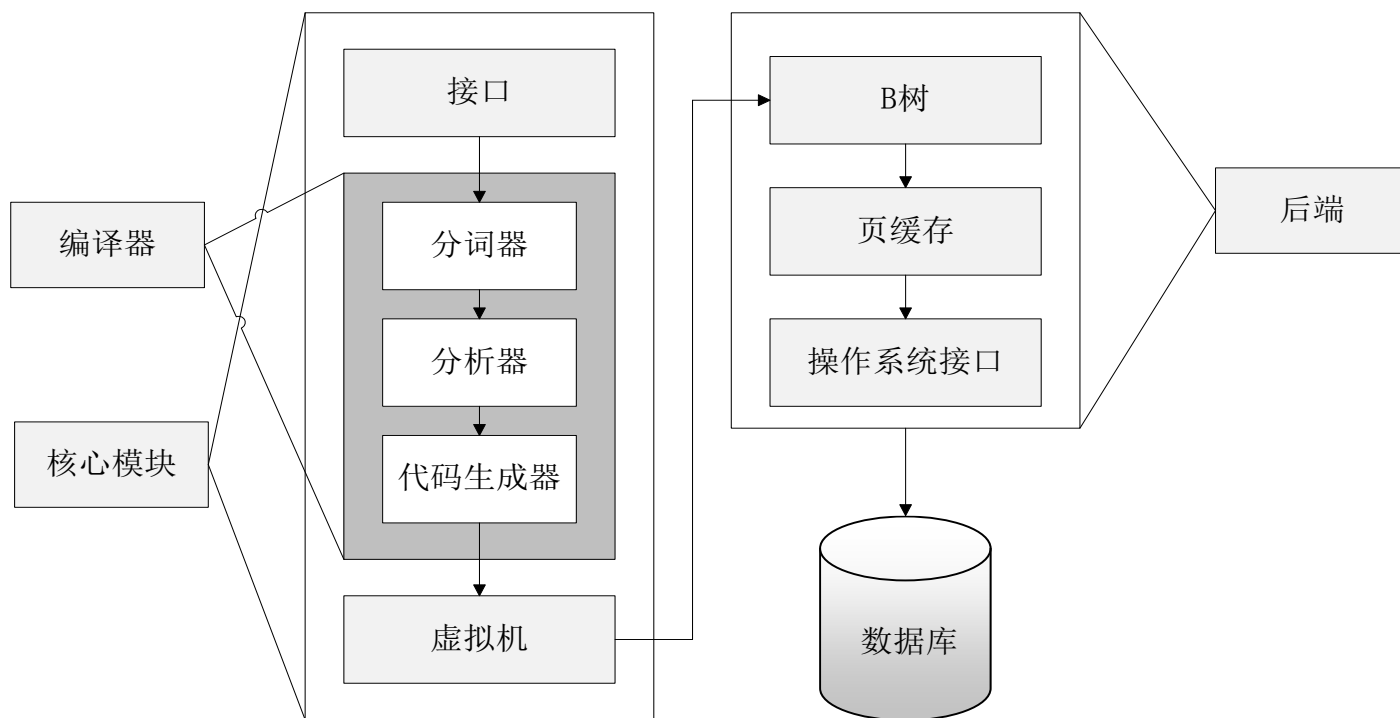
- 9.3.1 SQLite数据库
 - 这些数据库和其中的数据是应用程序所私有的
 - 不应该用其来存储文件
 - 如果要将其同其他应用程序共享，则必须把应用程序变为一个ContentProvider（后面将讲到）





9.3 数据库存储

- 9.3.1 SQLite数据库
 - SQLite数据库采用了模块化设计，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，模块将复杂的查询过程分解为细小的工作进行处理





9.3 数据库存储

- 9.3.1 SQLite数据库

- SQLite数据库具有很强的移植性，可以运行在Windows, Linux, BSD, Mac OS X和一些商用Unix系统，比如Sun的Solaris, IBM的AIX
- SQLite数据库也可以工作在许多嵌入式操作系统下，例如QNX, VxWorks, Palm OS, Symbian和Windows CE
- SQLite的核心大约有3万行标准C代码，模块化的设计使这些代码更加易于理解





9.3 数据库存储

- 9.3.1 SQLite数据库
 - Android SDK包含了若干有用的SQLite数据库管理类
 - 大多都存在于android.database.sqlite包中
 - 包中含有许多功能包类：管理数据库的创建和版本信息、数据库管理以及查询生成类等
 - 利用这些包能帮助你生成正确的SQL表达式和查询





9.3 数据库存储

9.3.2 使用SQLite数据库

• 第一步：创建实体类

- 数据库管理系统中的各种用于数据管理方便而设定的各种数据管理对象，如：数据库表、视图、存储过程等都是数据库实体。广义上讲，这些对象中所存储的数据也是数据库实体。因为它们也是确切存在着的实体。
- 实现一个实体类需要有如下元素：
 - 属性、构造函数、属性Get函数、属性Set函数

```
public class Member {  
    private Integer id;  
    private String name;  
    private String info;  
  
    public Member(){  
  
    }  
  
    public Member(String name, String info) {  
        this.name = name;  
        this.info = info;  
    }  
  
    public Member(Integer id, String name, String info) {  
        this.id = id;  
        this.name = name;  
        this.info = info;  
    }  
}
```



```
    public Integer getId() {  
        return id;  
    }  
    public void setId(Integer id) {  
        this.id = id;  
    }  
    public String getName() {  
        return name;  
    }  
    public void setName(String name) {  
        this.name = name;  
    }  
    public String getInfo() {  
        return info;  
    }  
    public void setInfo(String info) {  
        this.info = info;  
    }  
}
```



9.3 数据库存储

- 9.3.2 使用SQLite数据库
 - 第二步：使用应用程序上下文创建SQLite数据库
 - 首先创建的类需要继承SQLiteOpenHelper类

```
public class MemberDAO extends SQLiteOpenHelper
```

- 然后定义数据库的名字、版本以及里面的TABLE的名字，定义SQL的创建表格命令：

```
private static final String DB_NAME = "member.db";  
private static final int DB_VRESION = 1;  
private static final String TABLE_NAME = "member";  
private static final String SQL_CREATE_TABLE = "create table " + TABLE_NAME  
    + " (_id integer primary key autoincrement,"  
    + " name text not null, info text);";
```

- 在onCreate函数中调用execSQL函数执行创建表格指令即可成功创建表格

```
/**  
 * 第一次调用 getWritableDatabase() 或 getReadableDatabase() 时调用  
 */  
@Override  
public void onCreate(SQLiteDatabase db) {  
    // 创建数据库表  
    db.execSQL(SQL_CREATE_TABLE);  
}
```





9.3 数据库存储

- 9.3.2 使用SQLite数据库
 - 完整代码示例如下：

```
public class MemberDAO extends SQLiteOpenHelper {
    private static final String DB_NAME = "member.db";
    private static final int DB_VRESION = 1;
    private static final String TABLE_NAME = "member";
    private static final String SQL_CREATE_TABLE = "create table " + TABLE_NAME
        + " (_id integer primary key autoincrement,"
        + " name text not null, info text);";

    public MemberDAO(Context c) {
        /*
         * 创建数据库访问对象 它实际上没有创建数据库，马上返回。 只有调用 getWritableDatabase() 或
         * getReadableDatabase() 时才会创建数据库 数据库文件位于 /data/data/<包名>/databases
         */
        super(c, DB_NAME, null, DB_VRESION);
    }

    /* 第一次调用 getWritableDatabase() 或 getReadableDatabase() 时调用 */
    @Override
    public void onCreate(SQLiteDatabase db) {
        // 创建数据库表
        db.execSQL(SQL_CREATE_TABLE);
    }

    /* DB_VRESION 变化时调用此函数 */
    @Override
    public void onUpgrade(SQLiteDatabase db, int oldVersion, int newVersion) {
        // 更新数据库版本（这里不使用）
        // db.execSQL("DROP TABLE IF EXISTS " + TABLE_NAME);
        // onCreate(db);
    }
}
```





9.3 数据库存储

- 9.3.2 使用SQLite数据库

- **第三步：实现数据操作**

- 数据操作是指对数据的添加、删除、查找和更新操作；
 - 可以通过执行SQL命令来完成数据操作，通过调用SQLiteDatabase类的公共函数insert()、delete()、update()和query()这四个函数，封装了执行的添加、删除、更新和查询功能的SQL命令；**但是我们比较推荐使用Android提供的专用类和方法，更加简洁，在介绍完这些公共函数后会有介绍；**
 - 下面通过前面的Member实例，分别介绍如何使用SQLiteDatabase类的公共函数，完成数据的添加、删除、更新和查询等操作





9.3 数据库存储

- 添加功能

- 首先构造一个ContentValues对象，然后调用ContentValues对象的put()方法，将每个属性的值写入到ContentValues对象中，最后使用SQLiteDatabase对象的insert()函数，将ContentValues对象中的数据写入指定的数据库表中
- 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

```
/* 插入操作 */
public long insert(Member entity) {
    SQLiteDatabase db = getWritableDatabase();
    ContentValues values = new ContentValues();

    // 名称/值对，第一个参数是名称，第二个参数是值；
    values.put("name", entity.getName());
    values.put("info", entity.getInfo());

    // 必须保证 values 至少一个字段不为null，否则出错
    long rid = db.insert(TABLE_NAME, null, values);
    db.close();
    return rid;
}
```





9.3 数据库存储

- 删除功能

- 删除数据比较简单，只需要调用当前数据库对象的delete()函数，并指明表名称和删除条件即可

```
/* 删除操作 */  
public int deleteById(Integer id) {  
    SQLiteDatabase db = getWritableDatabase();  
    String whereClause = "_id = ?";  
    String[] whereArgs = { id.toString() };  
    int row = db.delete(TABLE_NAME, whereClause, whereArgs);  
    db.close();  
    return row;  
}
```

- delete()函数第1个参数是数据库表名，后面的参数是删除条件
- 参数id指明了需要删除数据的id值，因此deleteById()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的数量；
- 如果后面两个参数均为null，那么表示删除数据库中的全部数据。





9.3 数据库存储

- 更新功能

- 更新数据同样要使用ContentValues对象
- 首先构造ContentValues对象
- 然后调用put()函数将属性的值写入到ContentValues对象
- 最后使用SQLiteDatabase对象的update()函数，并指定数据的更新条件，函数返回的值是更新的数据数量

/ 更新操作 */*

```
public int update(Member entity) {  
    SQLiteDatabase db = getWritableDatabase();  
    String whereClause = "_id = ?";  
    String[] whereArgs = { entity.getId().toString() };  
    ContentValues values = new ContentValues();  
    values.put("name", entity.getName());  
    values.put("info", entity.getInfo());  
    int rows = db.update(TABLE_NAME, values, whereClause, whereArgs);  
    db.close();  
    return rows;  
}
```





9.3 数据库存储

- 查询功能

- 首先介绍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类支持在查询的数据集合中多种方式移动，并能够获取数据集合的属性名称和序号，Cursor类方法说明如下：

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





9.3 数据库存储

- 查询操作：
 - SQLiteDatabase类的query()函数参数说明

```
Cursor android.database.sqlite.SQLiteDatabase.query(String table,  
String[] columns, String selection, String[] selectionArgs,  
String groupBy, String having, String orderBy)
```

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的问号，则需要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





9.3 数据库存储

- 查询操作：
 - 示例代码如下：

```
/* 查询操作 */
public Member getById(Integer id){
    Member m = null;
    SQLiteDatabase db = getReadableDatabase();
    String selection = "_id = ?";
    String[] selectionArgs = { id.toString() };
    Cursor c = db.query(TABLE_NAME, null, selection, selectionArgs, null,null, null);
    if (c.moveToNext()){
        m = new Member(c.getInt(0),c.getString(1),c.getString(2));
    }
    c.close();
    db.close();
    return m;
}
```

- 在从Cursor中提取数据之前，推荐先测试一下Cursor中的数据数量，避免在数据获取的过程中产生异常情况。



```
if (resultCounts == 0 || !cursor.moveToFirst()){
    return null;
}
```




9.3 数据库存储

- 9.3.3 DBAdapter
 - DBAdapter是Android提供的专用数据库处理方法；
 - 为了使DBAdapter类支持对数据的添加、删除、更新和查找等功能，在DBAdapter类中增加下面的这些函数
 - 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





9.3 数据库存储

- 9.3.3 DBAdapter

```
public class DBAdapter {  
    public long insert(People people) {}  
    public long deleteAllData() { }  
    public long deleteOneData(long id) { }  
    public People[] queryAllData() {}  
    public People[] queryOneData(long id) { }  
    public long updateOneData(long id , People people){ }  
    private People[] ConvertToPeople(Cursor cursor){}  
}
```

- ConvertToPeople(Cursor cursor)是私有函数，作用是将查询结果转换为用来存储数据自定义的People类对象
- People类的包含四个公共属性，分别为ID、Name、Age和Height，对应数据库中的四个属性值





9.3 数据库存储

- 9.3.3 DBAdapter
 - People类的代码如下

```
public class People {  
    public int ID = -1;  
    public String Name;  
    public int Age;  
    public float Height;  
  
    @Override  
    public String toString(){  
        String result = "";  
        result += "ID: " + this.ID + ", ";  
        result += "姓名: " + this.Name + ", ";  
        result += "年龄: " + this.Age + ", ";  
        result += "身高: " + this.Height + ", ";  
        return result;  
    }  
}
```





9.4 数据分享

- 9.4.1 ContentProvider
 - ContentProvider（数据提供者）是在应用程序间共享数据的一种接口机制
 - 提供了更为高级的数据共享方法，应用程序可以指定需要共享的数据，而其他应用程序则可以在不知数据来源、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
 - 许多Android系统的内置数据通过ContentProvider提供给用户使用，例如通讯录、音视频文件和图像文件等





9.4 数据分享

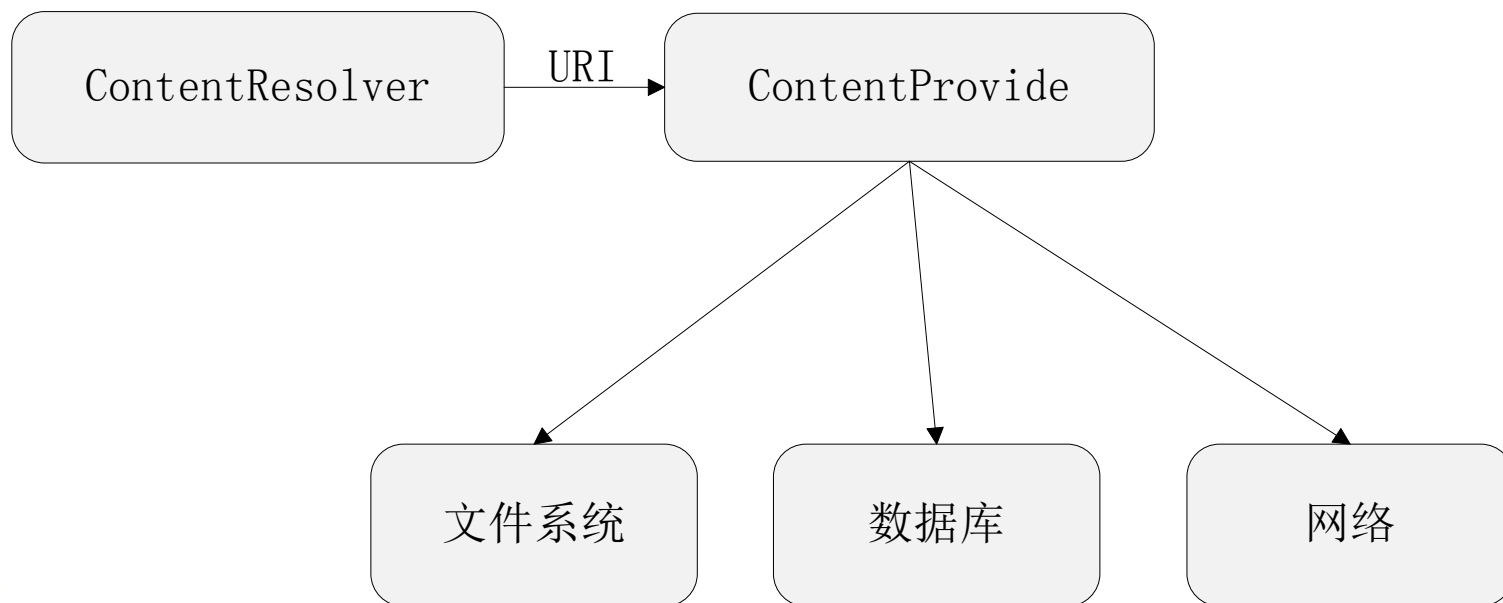
- 9.4.1 ContentProvider：创建ContentProvider基本过程
 - 首先使用数据库、文件系统或网络实现底层存储功能
 - 然后在继承ContentProvider的类中实现基本数据操作的接口函数，包括添加、删除、查找和更新等功能





9.4 数据分享

- 9.4.1 ContentProvider
 - 调用者**不能够直接调用**ContentProvider的接口函数
 - 需要使用ContentResolver对象，通过URI间接调用ContentProvider





9.4 数据分享

- 9.4.1 ContentProvider
 - ContentProvider完全屏蔽了数据提供组件的数据存储方法
 - 在使用者看来，数据提供者通过ContentProvider提供了一组标准的数据操作接口，却无法得知数据提供者的数据存储方式





9.4 数据分享

- 9.4.1 ContentProvider
 - 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，这些内容对数据使用者都是不可见
 - 同时也正是因为屏蔽数据的存储方法，很大程度上简化的ContentProvider的使用难度，使用者只要调用ContentProvider提供的接口函数，就可完成所有的数据操作





9.4 数据分享

- 9.4.1 ContentProvider
 - ContentProvider的数据模式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
 - 每条记录都包含一个长型的字段_ID，唯一标识该记录
 - ContentProvider可以提供多个数据集，调用者使用URI对不同的数据集的数据进行操作
 - ContentProvider数据模型

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





9.4 数据分享

- 9.4.1 ContentProvider
 - URI是通用资源标志符 (Uniform Resource Identifier) , 用来定位任何远程或本地的可用资源
 - ContentProvider使用的URI语法结构

```
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**唯一确定**由哪一个ContentProvider提供资源, 一般由**类的小写全称**组成





9.4 数据分享

- 9.4.1 ContentProvider
 - `<data_path>`是数据路径，用来确定请求的是哪个数据集
 - 如果ContentProvider仅提供一个数据集，数据路径可以省略
 - 如果ContentProvider提供多个数据集，数据路径则必须指明
 - 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





9.4 数据分享

- 9.4.1 ContentProvider
 - 如果请求的数据并不只限于一条数据，则<id>可以省略
 - 例 1：请求整个people数据集的URI应写为

```
content://edu.hrbeu.peopleprovider/people
```

- 例 2：请求people数据集中第3条数据的URI则应写为

```
content://edu.hrbeu.peopleprovider/people/3
```





9.4 数据分享

- 9.4.2 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
 - 第1步：继承ContentProvider，并重载六个函数
 - 第2步：声明CONTENT_URI，实现UriMatcher
 - 第3步：注册ContentProvider





9.4 数据分享

- 9.4.2 创建数据提供者

- 第1步：继承ContentProvider，并重载六个函数

- delete():删除数据集
- insert()：添加数据集
- query()：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





9.4 数据分享

- 9.4.2 创建数据提供者
 - 第1步：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代码，并自动生成需要重载的代码框架

```
import android.content.*;
import android.database.Cursor;
import android.net.Uri;
public class PeopleProvider extends ContentProvider{
    @Override
    public int delete(Uri uri, String selection, String[]
selectionArgs) {
        // TODO Auto-generated method stub
        return 0;
    }
}
```





9.4 数据分享

- 9.4.2 创建数据提供者

```
@Override
public String getType(Uri uri) {
    // TODO Auto-generated method stub
    return null;
}
@Override
public Uri insert(Uri uri, ContentValues values) {
    // TODO Auto-generated method stub
    return null;
}
@Override
public boolean onCreate() {
    // TODO Auto-generated method stub
    return false;
}
```





9.4 数据分享

- 9.4.2 创建数据提供者

```
@Override
    public Cursor query(Uri uri, String[] projection, String selection,
        String[] selectionArgs, String sortOrder) {
        // TODO Auto-generated method stub
        return null;
    }
    @Override
    public int update(Uri uri, ContentValues values, String selection,
        String[] selectionArgs) {
        // TODO Auto-generated method stub
        return 0;
    }
}
```





9.4 数据分享

- 9.4.2 创建数据提供者
 - 第2步：声明CONTENT_URI，实现UriMatcher
 - 在新构造的ContentProvider类中，通过构造一个UriMatcher，判断URI是单条数据还是多条数据
 - 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





9.4 数据分享

- 9.4.2 创建数据提供者
 - 第2步：声明CONTENT_URI和构造UriMatcher的代码

```
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. private static final UriMatcher uriMatcher;
9. static {
10.     uriMatcher = new UriMatcher(UriMatcher.NO_MATCH);
11.     uriMatcher.addURI(AUTHORITY, PATH_SINGLE, SINGLE_PEOPLE);
12.     uriMatcher.addURI(AUTHORITY, PATH_MULTIPLE, MULTIPLE_PEOPLE);
13. }
```





9.4 数据分享

- 9.4.2 创建数据提供者
 - 第1行代码声明了URI的授权者名称
 - 第2行代码声明了单条数据的数据路径
 - 第3行代码声明了多条数据的数据路径
 - 第4行代码声明了CONTENT_URI的字符串形式
 - 第5行代码则正式声明了CONTENT_URI
 - 第6行代码声明了多条数据的返回代码
 - 第7行代码声明了单条数据的返回代码
 - 第8行代码声明了UriMatcher
 - 第9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





9.4 数据分享

- 9.4.2 创建数据提供者
 - 其中第11行UriMatcher的构造函数中，UriMatcher.NO_MATCH表示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

```
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





9.4 数据分享

- 9.4.2 创建数据提供者
 - 第2步：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

```
switch(uriMatcher.match(uri)){  
    case MULTIPLE_PEOPLE:  
        // 多条数据的处理过程  
        break;  
    case SINGLE_PEOPLE:  
        // 单条数据的处理过程  
        break;  
    default:  
        throw new IllegalArgumentException("不支持的URI:"+uri);  
}
```





9.4 数据分享

- 9.4.2 创建数据提供者
 - 第3步：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，代码如下

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





9.4 数据分享

- 9.4.3 使用数据提供者
 - 使用ContentProvider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

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





9.4 数据分享

- 9.4.3 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

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

- 在URI中定义了需要查询数据的ID，在query()函数并没有额外声明查询条件





9.4 数据分享

- 9.4.3 使用数据提供者

- 查询操作

- 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

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

- uri定义了查询的数据集
 - projection定义了从数据集返回哪些数据项
 - selection定义了返回数据的查询条件





9.4 数据分享

- 9.4.3 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bulkInsert()函数，批量的添加数据





9.4 数据分享

- 9.4.3 使用数据提供者

- 添加操作

- 例1：如何使用insert()函数添加单条数据

```
ContentValues values = new ContentValues();  
values.put(KEY_NAME, "Tom");  
values.put(KEY_AGE, 21);  
values.put(KEY_HEIGHT, );  
Uri newUri = resolver.insert(CONTENT_URI, values);
```

- 例2：如何使用bulkInsert()函数添加多条数据

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





9.4 数据分享

- 9.4.3 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





9.4 数据分享

- 9.4.3 使用数据提供者

- 删除操作

- 例1：如何删除ID为2的数据

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

- 例2：在selection将删除条件定义为ID大于4的数据

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





9.4 数据分享

- 9.4.3 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
 - 例：如何更新ID为7的数据

```
ContentValues values = new ContentValues();  
values.put(KEY_NAME, "Tom");  
values.put(KEY_AGE, 21);  
values.put(KEY_HEIGHT, );  
Uri uri = Uri.parse(CONTENT_URI_STRING + "/" + "7");  
int result = resolver.update(uri, values, null, null);
```



A large crowd of 3D human figures, mostly light blue, with one prominent red figure in the foreground on the left. The figures are stylized and have a slight shadow on the ground.

Questions?

