第 4 章

数据存储

本章介绍 Android 五种主要存储方式的用法，包括共享参数 SharedPreferences、数据库 SQLite、SD 卡文件、App 的全局内存，另外介绍重要组件之一的应用 Application 的基本概念与常见用法，以及四大组件之一的内容提供器 ContentProvider 的基本概念与常见用法。最后，结合本章所学的知识演示实战项目“购物车”的设计与实现。

4.1 共享参数 SharedPreferences

本节介绍 Android 的键值对存储方式——共享参数 SharedPreferences 的使用方法，包括如何保存数据与读取数据，通过共享参数结合“登录 App”项目实现记住密码功能。

4.1.1 共享参数的基本用法

SharedPreferences 是 Android 的一个轻量级存储工具，采用的存储结构是 Key-Value 的键值对方式，类似于 Java 的 Properties 类，二者都是把 Key-Value 的键值对保存在配置文件中。不同的是 Properties 的文件内容是 Key=Value 这样的形式，而 SharedPreferences 的存储介质是符合 XML 规范的配置文件。保存 SharedPreferences 键值对信息的文件路径是/data/data/应用包名/shared_prefs/文件名.xml。下面是一个共享参数的 XML 文件示例：

```
<?xml version='1.0' encoding='utf-8' standalone='yes' ?>
<map>
    <string name="name">Mr Lee</string>
    <int name="age" value="30" />
    <boolean name="married" value="true" />
    <float name="weight" value="100.0" />
```

```
</map>
```

基于 XML 格式的特点，SharedPreferences 主要适用于如下场合：

（1）简单且孤立的数据。若是复杂且相互间有关的数据，则要保存在数据库中。
（2）文本形式的数据。若是二进制数据，则要保存在文件中。
（3）需要持久化存储的数据。在 App 退出后再次启动时，之前保存的数据仍然有效。

实际开发中，共享参数经常存储的数据有 App 的个性化配置信息、用户使用 App 的行为信息、临时需要保存的片段信息等。

SharedPreferences 对数据的存储和读取操作类似于 Map，也有 put 函数用于存储数据、get 函数用于读取数据。在使用共享参数之前，要先调用 getSharedPreferences 函数声明文件名与操作模式，示例代码如下：

```
// 从 share.xml 中获取共享参数对象
SharedPreferences shared = getSharedPreferences("share", MODE_PRIVATE);
```

getSharedPreferences 方法的第一个参数是文件名，上面的 share 表示当前使用的共享参数文件名是 share.xml；第二个参数是操作模式，一般都填 MODE_PRIVATE，表示私有模式。

共享参数存储数据要借助于 Editor 类，示例代码如下：

```
SharedPreferences.Editor editor = shared.edit();  // 获得编辑器的对象
editor.putString("name", "Mr Lee");  // 添加一个名叫 name 的字符串参数
editor.putInt("age", 30);  // 添加一个名叫 age 的整型参数
editor.putBoolean("married", true);  // 添加一个名叫 married 的布尔型参数
editor.putFloat("weight", 100f);  // 添加一个名叫 weight 的浮点数参数
editor.commit();  // 提交编辑器中的修改
```

共享参数读取数据相对简单，直接使用对象即可完成数据读取方法的调用，注意 get 方法的第二个参数表示默认值，示例代码如下：

```
String name = shared.getString("name", "");  // 从共享参数中获得名叫 name 的字符串
int age = shared.getInt("age", 0);  // 从共享参数中获得名叫 age 的整型数
boolean married = shared.getBoolean("married", false);  // 从共享参数中获得名叫 married 的布尔数
float weight = shared.getFloat("weight", 0);  // 从共享参数中获得名叫 weight 的浮点数
```

下面通过页面录入信息演示 SharedPreferences 的存取过程，如图 4-1 所示。在页面上利用 EditText 录入用户注册信息，并保存到共享参数文件中。在另一个页面，App 从共享参数文件中读取用户注册信息，并将注册信息依次显示在页面中，如图 4-2 所示。

图 4-1　写入共享参数

图 4-2　从共享参数读取

4.1.2　实现记住密码功能

上一章的实战项目“登录 App”页面下方有一个“记住密码”复选框，当时只是为了演示控件的运用，并未真正记住密码。因为用户退出后重新进入登录页面，App 没有回忆起上次用户的登录密码。现在利用共享参数对该项目进行改造，使之实现记住密码的功能。

改造的内容主要有 3 处：

（1）声明一个 SharedPreferences 对象，并在 onCreate 函数中调用 getSharedPreferences 方法对该对象进行初始化操作。

（2）登录成功时，如果用户勾选了“记住密码”，就使用共享参数保存手机号码与密码。也就是在 loginSuccess 函数中增加如下代码：

```
// 如果勾选了“记住密码”，则把手机号码和密码都保存到共享参数中
if (bRemember) {
    SharedPreferences.Editor editor = mShared.edit();  // 获得编辑器的对象
    editor.putString("phone", et_phone.getText().toString());  // 添加名叫 phone 的手机号码
    editor.putString("password", et_password.getText().toString());  // 添加名叫 password 的密码
    editor.commit();  // 提交编辑器中的修改
}
```

（3）在打开登录页面时，App 从共享参数中读取手机号码与密码，并展示在界面上。也就是在 onCreate 函数中增加如下代码：

```
// 从 share.xml 中获取共享参数对象
mShared = getSharedPreferences("share_login", MODE_PRIVATE);
// 获取共享参数中保存的手机号码
String phone = mShared.getString("phone", "");
// 获取共享参数中保存的密码
String password = mShared.getString("password", "");
et_phone.setText(phone);  // 给手机号码编辑框填写上次保存的手机号
et_password.setText(password);  // 给密码编辑框填写上次保存的密码
```

修改完毕后，如果不出意料，只要用户上次登录成功时勾选“记住密码”，下次进入登录页面时 App 就会自动填写上次登录的手机号码与密码。具体的效果如图 4-3 和图 4-4 所示。其中，图 4-3 所示为用户首次登录成功，此时勾选了“记住密码”；图 4-4 所示为用户再次进入登录页面，因为上次登录成功时已经记住密码，所以这次页面会自动展示保存的登录信息。

图 4-3　将登录信息保存到共享参数

图 4-4　从共享参数读取登录信息

4.2　数据库 SQLite

本节介绍 Android 的数据库存储方式—— SQLite 的使用方法，包括如何建表和删表、变更表结构以及对表数据进行增加、删除、修改、查询等操作，然后通过 SQLite 结合“登录 App”项目改进记住密码功能。

4.2.1　SQLite 的基本用法

SQLite 是一个小巧的嵌入式数据库，使用方便、开发简单，手机上最早由 iOS 运用，后来 Android 也采用了 SQLite。SQLite 的多数 SQL 语法与 Oracle 一样，下面只列出不同的地方：

（1）建表时为避免重复操作，应加上 IF NOT EXISTS 关键词，例如 CREATE TABLE IF NOT EXISTS table_name。

（2）删表时为避免重复操作，应加上 IF EXISTS 关键词，例如 DROP TABLE IF EXISTS table_name。

（3）添加新列时使用 ALTER TABLE table_name ADD COLUMN ...，注意比 Oracle 多了一个 COLUMN 关键字。

（4）在 SQLite 中，ALTER 语句每次只能添加一列，如果要添加多列，就只能分多次添加。

（5）SQLite 支持整型 INTEGER、字符串 VARCHAR、浮点数 FLOAT，但不支持布尔类型。布尔类型数要使用整型保存，如果直接保存布尔数据，在入库时 SQLite 就会自动将其转为 0 或 1，0 表示 false，1 表示 true。

（6）SQLite 建表时需要一个唯一标识字段，字段名为_id。每建一张新表都要例行公事加上该字段定义，具体属性定义为_id INTEGER PRIMARY KEY AUTOINCREMENT NOT NULL。

（7）条件语句等号后面的字符串值要用单引号括起来，如果没用使用单引号括起来，在运行时就会报错。

SQLiteDatabase 是 SQLite 的数据库管理类，开发者可以在活动页面代码或任何能取到 Context 的地方获取数据库实例，参考代码如下：

```
// 创建名叫 test.db 的数据库。数据库如果不存在就创建它，如果存在就打开它
SQLiteDatabase db = openOrCreateDatabase(getFilesDir() + "/test.db", Context.MODE_PRIVATE, null);
// 删除名叫 test.db 数据库
// deleteDatabase(getFilesDir() + "/test.db");
```

SQLiteDatabase 提供了若干操作数据表的 API，常用的方法有 3 类，列举如下：

1. 管理类，用于数据库层面的操作。

- openDatabase：打开指定路径的数据库。
- isOpen：判断数据库是否已打开。
- close：关闭数据库。
- getVersion：获取数据库的版本号。
- setVersion：设置数据库的版本号。

2. 事务类，用于事务层面的操作。

- beginTransaction：开始事务。
- setTransactionSuccessful：设置事务的成功标志。
- endTransaction: 结束事务。执行本方法时，系统会判断是否已执行 setTransactionSuccessful，如果之前已设置就提交，如果没有设置就回滚。

3. 数据处理类，用于数据表层面的操作。

- execSQL：执行拼接好的 SQL 控制语句。一般用于建表、删表、变更表结构。
- delete：删除符合条件的记录。
- update：更新符合条件的记录。
- insert：插入一条记录。
- query：执行查询操作，返回结果集的游标。
- rawQuery：执行拼接好的 SQL 查询语句，返回结果集的游标。

4.2.2 数据库帮助器 SQLiteOpenHelper

SQLiteDatabase 存在局限性，例如必须小心、不能重复地打开数据库，处理数据库的升级很不方便。Android 提供了一个辅助工具—— SQLiteOpenHelper，用于指导开发者进行 SQLite 的合理使用。

SQLiteOpenHelper 的具体使用步骤如下：

步骤 01 新建一个继承自 SQLiteOpenHelper 的数据库操作类，提示重写 onCreate 和 onUpgrade 两个方法。其中，onCreate 方法只在第一次打开数据库时执行，在此可进行表结构创建的操作；

onUpgrade 方法在数据库版本升高时执行，因此可以在 onUpgrade 函数内部根据新旧版本号进行表结构变更处理。

步骤 02 封装保证数据库安全的必要方法，包括获取单例对象、打开数据库连接、关闭数据库连接。

- 获取单例对象：确保 App 运行时数据库只被打开一次，避免重复打开引起错误。
- 打开数据库连接：SQLite 有锁机制，即读锁和写锁的处理；故而数据库连接也分两种，读连接可调用 SQLiteOpenHelper 的 getReadableDatabase 方法获得，写连接可调用 getWritableDatabase 获得。
- 关闭数据库连接：数据库操作完毕后，应当调用 SQLiteDatabase 对象的 close 方法关闭连接。

步骤 03 提供对表记录进行增加、删除、修改、查询的操作方法。

可被 SQLite 直接使用的数据结构是 ContentValues 类，类似于映射 Map，提供 put 和 get 方法用来存取键值对。区别之处在于 ContentValues 的键只能是字符串，查看 ContentValues 的源码会发现其内部保存键值对的数据结构就是 HashMap“private HashMap<String, Object> mValues;”。ContentValues 主要用于记录增加和更新操作，即 SQLiteDatabase 的 insert 和 update 方法。

对于查询操作来说，使用的是另一个游标类 Cursor。调用 SQLiteDatabase 的 query 和 rawQuery 方法时，返回的都是 Cursor 对象，因此获取查询结果要根据游标的指示一条一条遍历结果集合。Cursor 的常用方法可分为 3 类，说明如下：

1. 游标控制类方法，用于指定游标的状态。

- close：关闭游标。
- isClosed：判断游标是否关闭。
- isFirst：判断游标是否在开头。
- isLast：判断游标是否在末尾。

2. 游标移动类方法，把游标移动到指定位置。

- moveToFirst：移动游标到开头。
- moveToLast：移动游标到末尾。
- moveToNext：移动游标到下一条记录。
- moveToPrevious：移动游标到上一条记录。
- move：往后移动游标若干条记录。
- moveToPosition：移动游标到指定位置的记录。

3. 获取记录类方法，可获取记录的数量、类型以及取值。

- getCount：获取结果记录的数量。
- getInt：获取指定字段的整型值。
- getFloat：获取指定字段的浮点数值。

- getString：获取指定字段的字符串值。
- getType：获取指定字段的字段类型。

鉴于数据库操作的特殊性，不方便单独演示某个功能，接下来从创建数据库开始介绍，完整演示一下数据库的读写操作。如图 4-5 和图 4-6 所示，在页面上分别录入两个用户的注册信息并保存到 SQLite。从 SQLite 读取用户注册信息并展示在页面上，如图 4-7 所示。

图 4-5　第一条注册信息保存到数据库

图 4-6　第二条注册信息保存到数据库

图 4-7　从 SQLite 中读取两条注册记录

下面是用户注册信息数据库的 SQLiteOpenHelper 操作类的完整代码：

```
public class UserDBHelper extends SQLiteOpenHelper {
    private static final String DB_NAME = "user.db";   // 数据库的名称
    private static final int DB_VERSION = 1;   // 数据库的版本号
    private static UserDBHelper mHelper = null;   // 数据库帮助器的实例
    private SQLiteDatabase mDB = null;   // 数据库的实例
    public static final String TABLE_NAME = "user_info";   // 表的名称

    private UserDBHelper(Context context) {
        super(context, DB_NAME, null, DB_VERSION);
    }
```

```
private UserDBHelper(Context context, int version) {
    super(context, DB_NAME, null, version);
}

// 利用单例模式获取数据库帮助器的唯一实例
public static UserDBHelper getInstance(Context context, int version) {
    if (version > 0 && mHelper == null) {
        mHelper = new UserDBHelper(context, version);
    } else if (mHelper == null) {
        mHelper = new UserDBHelper(context);
    }
    return mHelper;
}

// 打开数据库的读连接
public SQLiteDatabase openReadLink() {
    if (mDB == null || !mDB.isOpen()) {
        mDB = mHelper.getReadableDatabase();
    }
    return mDB;
}

// 打开数据库的写连接
public SQLiteDatabase openWriteLink() {
    if (mDB == null || !mDB.isOpen()) {
        mDB = mHelper.getWritableDatabase();
    }
    return mDB;
}

// 关闭数据库连接
public void closeLink() {
    if (mDB != null && mDB.isOpen()) {
        mDB.close();
        mDB = null;
    }
}

// 创建数据库，执行建表语句
public void onCreate(SQLiteDatabase db) {
    String drop_sql = "DROP TABLE IF EXISTS " + TABLE_NAME + ";";
    db.execSQL(drop_sql);
    String create_sql = "CREATE TABLE IF NOT EXISTS " + TABLE_NAME + " ("
```

```
                + "_id INTEGER PRIMARY KEY    AUTOINCREMENT NOT NULL,"
                + "name VARCHAR NOT NULL," + "age INTEGER NOT NULL,"
                + "height LONG NOT NULL," + "weight FLOAT NOT NULL,"
                + "married INTEGER NOT NULL," + "update_time VARCHAR NOT NULL"
                + ",phone VARCHAR" + ",password VARCHAR" + ");";
        db.execSQL(create_sql);
    }

    // 修改数据库，执行表结构变更语句
    public void onUpgrade(SQLiteDatabase db, int oldVersion, int newVersion) {}

    // 根据指定条件删除表记录
    public int delete(String condition) {
        // 执行删除记录动作，该语句返回删除记录的数目
        return mDB.delete(TABLE_NAME, condition, null);
    }

    // 往该表添加多条记录
    public long insert(ArrayList<UserInfo> infoArray) {
        long result = -1;
        for (int i = 0; i < infoArray.size(); i++) {
            UserInfo info = infoArray.get(i);
            ArrayList<UserInfo> tempArray = new ArrayList<UserInfo>();
            // 如果存在同样的手机号码，则更新记录
            // 注意条件语句的等号后面要用单引号括起来
            if (info.phone != null && info.phone.length() > 0) {
                String condition = String.format("phone='%s'", info.phone);
                tempArray = query(condition);
                if (tempArray.size() > 0) {
                    update(info, condition);
                    result = tempArray.get(0).rowid;
                    continue;
                }
            }
            // 不存在唯一性重复的记录，则插入新记录
            ContentValues cv = new ContentValues();
            cv.put("name", info.name);
            cv.put("age", info.age);
            cv.put("height", info.height);
            cv.put("weight", info.weight);
            cv.put("married", info.married);
            cv.put("update_time", info.update_time);
            cv.put("phone", info.phone);
```

```
            cv.put("password", info.password);
            // 执行插入记录动作，该语句返回插入记录的行号
            result = mDB.insert(TABLE_NAME, "", cv);
            // 添加成功后返回行号，失败后返回-1
            if (result == -1) {
                return result;
            }
        }
        return result;
    }

    // 根据条件更新指定的表记录
    public int update(UserInfo info, String condition) {
        ContentValues cv = new ContentValues();
        cv.put("name", info.name);
        cv.put("age", info.age);
        cv.put("height", info.height);
        cv.put("weight", info.weight);
        cv.put("married", info.married);
        cv.put("update_time", info.update_time);
        cv.put("phone", info.phone);
        cv.put("password", info.password);
        // 执行更新记录动作，该语句返回记录更新的数目
        return mDB.update(TABLE_NAME, cv, condition, null);
    }

    // 根据指定条件查询记录，并返回结果数据队列
    public ArrayList<UserInfo> query(String condition) {
        String sql = String.format("select rowid,_id,name,age,height,weight,married,update_time," +
                "phone,password from %s where %s;", TABLE_NAME, condition);
        ArrayList<UserInfo> infoArray = new ArrayList<UserInfo>();
        // 执行记录查询动作，该语句返回结果集的游标
        Cursor cursor = mDB.rawQuery(sql, null);
        // 循环取出游标指向的每条记录
        while (cursor.moveToNext()) {
            UserInfo info = new UserInfo();
            info.rowid = cursor.getLong(0);    // 取出长整型数
            info.xuhao = cursor.getInt(1);    // 取出整型数
            info.name = cursor.getString(2);    // 取出字符串
            info.age = cursor.getInt(3);
            info.height = cursor.getLong(4);
            info.weight = cursor.getFloat(5);    // 取出浮点数
            //SQLite 没有布尔型，用 0 表示 false，用 1 表示 true
```

```
                info.married = (cursor.getInt(6) == 0) ? false : true;
                info.update_time = cursor.getString(7);
                info.phone = cursor.getString(8);
                info.password = cursor.getString(9);
                infoArray.add(info);
            }
            cursor.close();   // 查询完毕，关闭游标
            return infoArray;
        }

        // 根据手机号码查询指定记录
        public UserInfo queryByPhone(String phone) {
            UserInfo info = null;
            ArrayList<UserInfo> infoArray = query(String.format("phone='%s'", phone));
            if (infoArray.size() > 0) {
                info = infoArray.get(0);
            }
            return info;
        }
    }
```

4.2.3 优化记住密码功能

在“4.1.2 实现记住密码功能”中，我们利用共享参数实现了记住密码的功能，不过这个方法有局限，只能记住一个用户的登录信息，并且手机号码跟密码不存在从属关系，如果换个手机号码登录，前一个用户的登录信息就被覆盖了。真正意义上的记住密码功能是先输入手机号码，然后根据手机号匹配保存的密码，一个密码对应一个手机号码，从而实现具体手机号码的密码记忆功能。

现在运用 SQLite 技术分条存储不同用户的登录信息，并提供根据手机号码查找登录信息的方法，这样可以同时记住多个手机号码的密码。具体的改造主要有以下 3 点：

（1）声明一个 UserDBHelper 对象，然后在活动页面的 onResume 方法中打开数据库连接，在 onPasue 方法中关闭数据库连接，示例代码如下：

```
    private UserDBHelper mHelper;   // 声明一个用户数据库帮助器对象

    @Override
    protected void onResume() {
        super.onResume();
        // 获得用户数据库帮助器的一个实例
        mHelper = UserDBHelper.getInstance(this, 2);
        // 恢复页面，则打开数据库连接
        mHelper.openWriteLink();
    }
```

```
    @Override
    protected void onPause() {
        super.onPause();
        // 暂停页面，则关闭数据库连接
        mHelper.closeLink();
    }
```

（2）登录成功时，如果用户勾选了“记住密码”，就使用数据库保存手机号码与密码在内的登录信息。也就是在 loginSuccess 函数中增加如下代码：

```
        // 如果勾选了“记住密码”，则把手机号码和密码保存为数据库的用户表记录
        if (bRemember) {
            // 创建一个用户信息实体类
            UserInfo info = new UserInfo();
            info.phone = et_phone.getText().toString();
            info.password = et_password.getText().toString();
            info.update_time = DateUtil.getNowDateTime("yyyy-MM-dd HH:mm:ss");
            // 往用户数据库添加登录成功的用户信息（包含手机号码、密码、登录时间）
            mHelper.insert(info);
        }
```

（3）再次打开登录页面，用户输入手机号完毕后点击密码输入框时，App 到数据库中根据手机号查找登录记录，并将记录结果中的密码填入密码框。

看到这里，读者也许已经想到给密码框注册点击事件，然后在 onClick 方法中补充数据库读取操作。可是 EditText 比较特殊，点击后只是让其获得焦点，再次点击才会触发点击事件。也就是说，要连续点击两次 EditText 才会处理点击事件。Android 有时就是这么调皮捣蛋，你让它往东，它偏偏往西。难不成叫用户将就一下点击两次？用户肯定觉得这个 App 古怪、难用，还是卸载好了……这里提供一个解决办法，先给密码框注册一个焦点变更监听器，比如下面这行代码：

```
        // 给密码编辑框注册一个焦点变化监听器，一旦焦点发生变化，就触发监听器的
onFocusChange 方法
        et_password.setOnFocusChangeListener(this);
```

这个焦点变更监听器要实现接口 OnFocusChangeListener，对应的事件处理方法是 onFocusChange，将数据库查询操作放在该方法中，详细代码示例如下：

```
    // 焦点变更事件的处理方法，hasFocus 表示当前控件是否获得焦点。
    // 为什么光标进入密码框事件不选 onClick？因为要点两下才会触发 onClick 动作（第一下是切换
焦点动作）
    @Override
    public void onFocusChange(View v, boolean hasFocus) {
        String phone = et_phone.getText().toString();
        // 判断是否是密码编辑框发生焦点变化
```

```
            if (v.getId() == R.id.et_password) {
                // 用户已输入手机号码，且密码框获得焦点
                if (phone.length() > 0 && hasFocus) {
                    // 根据手机号码到数据库中查询用户记录
                    UserInfo info = mHelper.queryByPhone(phone);
                    if (info != null) {
                        // 找到用户记录，则自动在密码框中填写该用户的密码
                        et_password.setText(info.password);
                    }
                }
            }
        }
```

这样，就不再需要点击两次才处理点击事件了。

代码写完后，再来看登录页面的效果图，用户上次登录成功时已勾选“记住密码”，现在再次进入登录页面，用户输入手机号后光标还停留在手机框，如图 4-8 所示。接着点击密码框，光标随之跳到密码框，这时密码框自动填入了该手机号对应的密码串，如图 4-9 所示。如此便真正实现了记住密码功能。

图 4-8　光标在手机号码框

图 4-9　光标在密码输入框

4.3　SD 卡文件操作

本节介绍 Android 的文件存储方式—— SD 卡的用法，包括如何获取 SD 卡目录信息、公有存储空间与私有存储空间的区别、在 SD 卡上读写文本文件、在 SD 卡读写图片文件等功能。

4.3.1　SD 卡的基本操作

手机的存储空间一般分为两块，一块用于内部存储，另一块用于外部存储（SD 卡）。早期的 SD 卡是可插拔式的存储芯片，不过自己买的 SD 卡质量参差不齐，经常会影响 App 的正

常运行，所以后来越来越多的手机把 SD 卡固化到手机内部，虽然拔不出来，但是 Android 仍然称之为外部存储。

获取手机上的 SD 卡信息通过 Environment 类实现，该类是 App 获取各种目录信息的工具，主要方法有以下 7 种。

- getRootDirectory：获得系统根目录的路径。
- getDataDirectory：获得系统数据目录的路径。
- getDownloadCacheDirectory：获得下载缓存目录的路径。
- getExternalStorageDirectory：获得外部存储（SD 卡）的路径。
- getExternalStorageState：获得 SD 卡的状态。

SD 卡状态的具体取值说明见表 4-1。

表 4-1　SD 卡的存储状态取值说明

Environment 类的存储状态常量名	常量值	常量说明
MEDIA_UNKNOWN	unknown	未知
MEDIA_REMOVED	removed	已经移除
MEDIA_UNMOUNTED	unmounted	未挂载
MEDIA_CHECKING	checking	正在检查
MEDIA_NOFS	nofs	不支持的文件系统
MEDIA_MOUNTED	mounted	已经挂载，且是可读写状态
MEDIA_MOUNTED_READ_ONLY	mounted_ro	已经挂载，且是只读状态
MEDIA_SHARED	shared	当前未挂载，但通过 USB 共享
MEDIA_BAD_REMOVAL	bad_removal	未挂载就被移除
MEDIA_UNMOUNTABLE	unmountable	无法挂载
MEDIA_EJECTING	ejecting	正在弹出

- getStorageState：获得指定目录的状态。
- getExternalStoragePublicDirectory：获得 SD 卡指定类型目录的路径。

目录类型的具体取值说明见表 4-2。

表 4-2　SD 卡的目录类型取值说明

Environment 类的目录类型	常量值	常量说明
DIRECTORY_DCIM	DCIM	相片存放目录（包括相机拍摄的图片和视频）
DIRECTORY_DOCUMENTS	Documents	文档存放目录
DIRECTORY_DOWNLOADS	Download	下载文件存放目录
DIRECTORY_MOVIES	Movies	视频存放目录
DIRECTORY_MUSIC	Music	音乐存放目录
DIRECTORY_PICTURES	Pictures	图片存放目录

为正常操作 SD 卡，需要在 AndroidManifest.xml 中声明 SD 卡的权限，具体代码如下：

```
<!-- SD 卡读写权限 -->
<uses-permission android:name="android.permission.WRITE_EXTERNAL_STORAGE" />
<uses-permission android:name="android.permission.READ_EXTERNAL_STORAG" />
<uses-permission android:name="android.permission.MOUNT_UNMOUNT_FILESYSTEMS" />
```

下面演示一下 Environment 类各方法的使用效果，如图 4-10 所示。页面上展示了 Environment 类获取到的系统及 SD 卡的相关目录信息。

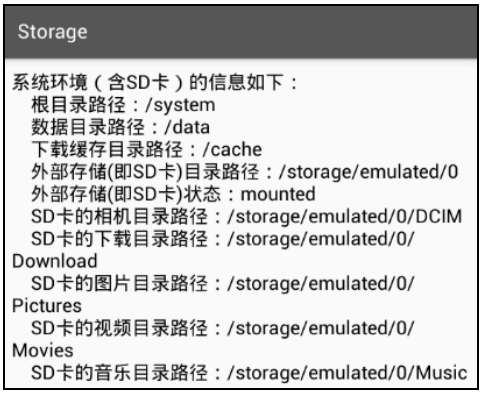

图 4-10　某设备上的 SD 卡目录信息

4.3.2　公有存储空间与私有存储空间

本来在 AndroidManifest.xml 里面配置了存储空间的权限，代码就能正常读写 SD 卡的文件。可是 Android 从 7.0 开始加强了 SD 卡的权限管理，即使 App 声明了完整的 SD 卡操作权限，系统仍然默认禁止该 App 访问外部存储。打开 7.0 系统的设置界面，进入到具体应用的管理页面，会发现应用的存储功能被关闭了（指外部存储），如图 4-11 所示。

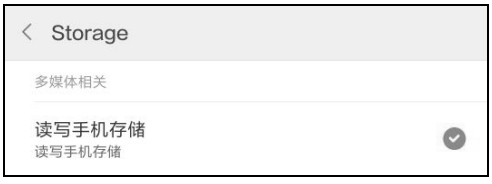

图 4-11　系统设置页面里的 SD 卡读写权限开关

不过系统默认关闭存储其实只是关闭外部存储的公共空间，外部存储的私有空间依然可以正常读写。这是缘于 Android 把外部存储分成了两块区域，一块是所有应用均可访问的公共空间，另一块是只有应用自己才可访问的专享空间。之前讲过，内部存储保存着每个应用的安装目录，但是安装目录的空间是很紧张的，所以 Android 在 SD 卡的“Android/data”目录下给每个应用又单独建了一个文件目录，用于给应用保存自己需要处理的临时文件。这个给每个应用单独建立的文件目录，只有当前应用才能够读写文件，其他应用是不允许进行读写的，故而“Android/data”目录算是外部存储上的私有空间。这个私有空间本身已经做了访问权限控制，因此它不受系统禁止访问的影响，应用操作自己的文件目录就不成问题了。当然，因为私有的文件目录只有属主应用才能访问，所以一旦属主应用被用户卸载，那么对应的文件目录也会一

起被清理掉。

既然外部存储分成了公共空间和私有空间两部分，这两部分空间的路径获取也就有所不同。获取公共空间的存储路径，调用的是 Environment.getExternalStoragePublicDirectory 方法；获取应用私有空间的存储路径，调用的是 getExternalFilesDir 方法。下面是分别获取两个空间路径的代码例子：

```
// 获取系统的公共存储路径
String publicPath = Environment.getExternalStoragePublicDirectory(
        Environment.DIRECTORY_DOWNLOADS).toString();
// 获取当前 App 的私有存储路径
String privatePath = getExternalFilesDir(Environment.DIRECTORY_DOWNLOADS).toString();
TextView tv_file_path = findViewById(R.id.tv_file_path);
String desc = "系统的公共存储路径位于" + publicPath +
        "\n\n 当前 App 的私有存储路径位于" + privatePath +
        "\n\nAndroid7.0 之后默认禁止访问公共存储目录";
tv_file_path.setText(desc);
```

该例子运行之后获得的路径信息如图 4-12 所示，可见应用的私有空间路径位于“外部存储根目录/Android/data/应用包名/files/Download”这个目录之下。

图 4-12　公共存储与私有存储的各自目录路径

4.3.3　文本文件读写

文本文件的读写一般借助于 FileOutputStream 和 FileInputStream。其中，FileOutputStream 用于写文件，FileInputStream 用于读文件。文件输出输入流是 Java 语言的基础工具，这里不再赘述，直接给出具体的实现代码：

```
// 把字符串保存到指定路径的文本文件
public static void saveText(String path, String txt) {
    try {
        FileOutputStream fos = new FileOutputStream(path);  // 根据路径构建文件输出流对象
        fos.write(txt.getBytes());  // 把字符串写入文件输出流
        fos.close();  // 关闭文件输出流
    } catch (Exception e) {
        e.printStackTrace();
    }
```

```
    }

    // 从指定路径的文本文件中读取内容字符串
    public static String openText(String path) {
        String readStr = "";
        try {
            FileInputStream fis = new FileInputStream(path);  // 根据路径构建文件输入流对象
            byte[] b = new byte[fis.available()];
            fis.read(b);  // 从文件输入流读取字节数组
            readStr = new String(b);  // 把字节数组转换为字符串
            fis.close();  // 关闭文件输入流
        } catch (Exception e) {
            e.printStackTrace();
        }
        return readStr;  // 返回文本文件中的文本字符串
    }
```

文本文件的读写效果如图 4-13 所示，此时 App 把注册信息保存到 SD 卡的文本文件中。接着进入文件列表读取页面，选中某个文件，页面就展示该文件的文本内容，如图 4-14 所示。

图 4-13　将注册信息保存到文本文件

图 4-14　从文本文件读取注册信息

4.3.4　图片文件读写

Android 的图片处理类是 Bitmap，App 读写 Bitmap 可以使用 FileOutputStream 和 FileInputStream。不过在实际开发中，读写图片文件一般用性能更好的 BufferedOutputStream 和 BufferedInputStream。

保存图片文件时用到 Bitmap 的 compress 方法，可指定图片类型和压缩质量；打开图片文件时使用 BitmapFactory 的 decodeStream 方法。读写图片的具体代码如下：

```
// 把位图数据保存到指定路径的图片文件
public static void saveImage(String path, Bitmap bitmap) {
    try {
        // 根据指定文件路径构建缓存输出流对象
        BufferedOutputStream bos = new BufferedOutputStream(new FileOutputStream(path));
        // 把位图数据压缩到缓存输出流中
        bitmap.compress(Bitmap.CompressFormat.JPEG, 80, bos);
        // 完成缓存输出流的写入动作
        bos.flush();
        // 关闭缓存输出流
        bos.close();
    } catch (Exception e) {
        e.printStackTrace();
    }
}

// 从指定路径的图片文件中读取位图数据
public static Bitmap openImage(String path) {
    Bitmap bitmap = null;
    try {
        // 根据指定文件路径构建缓存输入流对象
        BufferedInputStream bis = new BufferedInputStream(new FileInputStream(path));
        // 从缓存输入流中解码位图数据
        bitmap = BitmapFactory.decodeStream(bis);
        bis.close();   // 关闭缓存输入流
    } catch (Exception e) {
        e.printStackTrace();
    }
    // 返回图片文件中的位图数据
    return bitmap;
}
```

接下来是演示时间，如图 4-15 所示，用户在注册页面录入注册信息，App 调用 getDrawingCache 方法把整个注册界面截图并保存到 SD 卡；然后在另一个页面的图片列表选择 SD 卡上的指定图片文件，页面就会展示上次保存的注册界面图片，如图 4-16 所示。

图 4-15　保存注册信息图片

图 4-16　读取注册信息图片

刚才从 SD 卡读取图片文件用到了 BitmapFactory 的 decodeStream 方法，其实 BitmapFactory 还提供了其他方法，用起来更简单、方便，说明如下：

- decodeFile：该方法直接传文件路径的字符串，即可将指定路径的图片读取到 Bitmap 对象。
- decodeResource：该方法可从资源文件中读取图片信息。第一个参数一般传 getResources()，第二个参数传 drawable 图片的资源 id，如 R.drawable.phone。

4.4　应用 Application 基础

本节介绍 Android 重要组件 Application 的基本概念和常见用法。首先说明 Application 的生命周期，接着利用 Application 的持久特性实现 App 内部全局内存中的数据保存和获取。

4.4.1　Application 的生命周期

Application 是 Android 的一大组件，在 App 运行过程中有且仅有一个 Application 对象贯穿整个生命周期。打开 AndroidManifest.xml 时会发现 activity 节点的上级正是 application 节点，只是默认的 application 节点没有指定 name 属性，不像 activity 节点默认指定 name 属性值为.MainActivity，让人知晓这个 activity 的入口代码是 MainActivity.java。现在试试给 application 节点加上 name 属性，看看其庐山真面目。

（1）打开 AndroidManifest.xml，给 application 节点加上 name 属性，表示 application 的入口代码是 MainApplication.java。

```
android:name=".MainApplication"
```

（2）创建 MainApplication 类，该类继承自 Application，可以重写的方法主要有以下 4 个。

- onCreate：在 App 启动时调用。
- onTerminate：在 App 退出时调用（按字面意思）。
- onLowMemory：在低内存时调用。
- onConfigurationChanged：在配置改变时调用，例如从竖屏变为横屏。

（3）运行 App，同时开启日志的打印。但是只在一开始看到 MainApplication 的 onCreate 操作（先于 Activity 的 onCreate），却始终无法看到它的 onTerminate 操作，无论是自行退出还是强行杀死 App 的进程，日志都不会打印 onTerminate。

无论你怎么折腾，这个 onTerminate 都不会出来。Android 明明提供了这个函数，同时提供了关于该函数的解释，说明文字如下：This method is for use in emulated process environments. It will never be called on a production Android device, where processes are removed by simply killing them; no user code (including this callback) is executed when doing so。这段话的意思是该方法是供模拟环境用的，在真机上永远不会被调用，无论是直接杀进程还是代码退出。

现在很明确了，onTerminate 方法就是个摆设，中看不中用。如果读者想在 App 退出前做资源回收操作，那么千万不要放在 onTerminate 方法中。

4.4.2　利用 Application 操作全局变量

C/C++有全局变量，因为全局变量保存在内存中，所以操作全局变量就是操作内存，内存的读写速度远比读写数据库或读写文件快得多。全局的意思是其他代码都可以引用该变量，因此全局变量是共享数据和消息传递的好帮手。不过，Java 没有全局变量的概念。与之比较接近的是类里面的静态成员变量，该变量可被外部直接引用，并且在不同地方引用的值是一样的（前提是在引用期间不能修改该变量的值），所以可以借助静态成员变量实现类似全局变量的功能。

前面花费很大功夫介绍 Application 的生命周期，目的是说明其生命周期覆盖 App 运行的全过程。不像短暂的 Activity 生命周期，只要进入别的页面，原页面就被停止或销毁。因此，通过利用 Application 的持久存在性可以在 Application 对象中保存全局变量。

适合在 Application 中保存的全局变量主要有下面 3 类数据：

（1）会频繁读取的信息，如用户名、手机号等。

（2）从网络上获取的临时数据，为节约流量、减少用户等待时间，想暂时放在内存中供下次使用，如 logo、商品图片等。

（3）容易因频繁分配内存而导致内存泄漏的对象，如 Handler 对象等。

要想通过 Application 实现全局内存的读写，得完成以下 3 项工作：

（1）写一个继承自 Application 的类 MainApplication。该类要采用单例模式，内部声明自身类的一个静态成员对象，在创建 App 时把自身赋值给这个静态对象，然后提供该静态对象的获取方法 getInstance。

（2）在 Activity 中调用 MainApplication 的 getInstance 方法，获得 MainApplication 的一个静态对象，通过该对象访问 MainApplication 的公共变量和公共方法。

（3）不要忘了在 AndroidManifest.xml 中注册新定义的 Application 类名，即在 application

节点中增加 android:name 属性，值为.MainApplication。

下面继续演示全局内存的读写效果，如图 4-17 所示。App 把注册信息保存到 MainApplication 的全局变量中，然后在另一个页面从 MainApplication 的全局变量中读取保存好的注册信息，如图 4-18 所示。

图 4-17　注册信息保存到全局内存

图 4-18　从全局内存读取注册信息

下面是自定义 MainApplicaton 类的代码框架：

```
public class MainApplication extends Application {

    // 声明一个当前应用的静态实例
    private static MainApplication mApp;
    // 声明一个公共的信息映射，可当作全局变量使用
    public HashMap<String, String> mInfoMap = new HashMap<String, String>();

    // 利用单例模式获取当前应用的唯一实例
    public static MainApplication getInstance() {
        return mApp;
    }

    @Override
    public void onCreate() {
        super.onCreate();
        // 在打开应用时对静态的应用实例赋值
        mApp = this;
    }
}
```

完成以上编码后，Activity 页面代码即可直接通过 MainApplication.getInstance().mInfoMap 对全局变量进行增、删、改、查操作。

4.5　内容提供与处理

本节介绍 Android 四大组件之一的 ContentProvider 的基本概念和常见用法。首先说明如何使用内容提供器封装数据的外部访问接口；接着阐述如何通过内容解析器在外部查询和修改数据，以及使用内容操作器完成批量数据操作；然后说明内容观察器的应用场合，并演示如何借助内容观察器实现流量校准的功能。

4.5.1　内容提供器 ContentProvider

Android 号称提供了 4 大组件，分别是页面 Activity、广播 Broadcast、服务 Service 和内容提供器 ContentProvider。其中内容提供器是跟数据存取有关的组件，完整的内容组件由内容提供器 ContentProvider、内容解析器 ContentResolver、内容观察器 ContentObserver 这三部分组成。

ContentProvider 为 App 存取内部数据提供统一的外部接口，让不同的应用之间得以共享数据。像我们熟知的 SQLite 操作的是应用自身的内部数据库；文件的上传和下载操作的是后端服务器的文件；而 ContentProvider 操作的是本设备其他应用的内部数据，是一种中间层次的数据存储形式。

在实际编码中，ContentProvider 只是一个服务端的数据存取接口，开发者需要在其基础上实现一个具体类，并重写以下相关数据库管理方法。

- onCreate：创建数据库并获得数据库连接。
- query：查询数据。
- insert：插入数据。
- update：更新数据。
- delete：删除数据。
- getType：获取数据类型。

这些方法看起来是不是很像 SQLite？没错，ContentProvider 作为中间接口，本身并不直接保存数据，而是通过 SQLiteOpenHelper 与 SQLiteDatabase 间接操作底层的 SQLite。所以要想使用 ContentProvider，首先得实现 SQLite 的数据表帮助类，然后由 ContentProvider 封装对外的接口。

下面是使用 ContentProvider 提供用户信息对外接口的代码：

```
public class UserInfoProvider extends ContentProvider {
    private UserDBHelper userDB;  // 声明一个用户数据库的帮助器对象
    public static final int USER_INFO = 1;  // Uri 匹配时的代号
    public static final UriMatcher uriMatcher = new UriMatcher(UriMatcher.NO_MATCH);
    static {  // 往 Uri 匹配器中添加指定的数据路径
        uriMatcher.addURI(UserInfoContent.AUTHORITIES, "/user", USER_INFO);
```

```
    }

    // 根据指定条件删除数据
    public int delete(Uri uri, String selection, String[] selectionArgs) {
        int count = 0;
        if (uriMatcher.match(uri) == USER_INFO) {
            // 获取 SQLite 数据库的写连接
            SQLiteDatabase db = userDB.getWritableDatabase();
            // 执行 SQLite 的删除操作，返回删除记录的数目
            count = db.delete(UserInfoContent.TABLE_NAME, selection, selectionArgs);
            db.close();  // 关闭 SQLite 数据库连接
        }
        return count;
    }

    // 插入数据
    public Uri insert(Uri uri, ContentValues values) {
        Uri newUri = uri;
        if (uriMatcher.match(uri) == USER_INFO) {
            // 获取 SQLite 数据库的写连接
            SQLiteDatabase db = userDB.getWritableDatabase();
            // 向指定的表插入数据，返回记录的行号
            long rowId = db.insert(UserInfoContent.TABLE_NAME, null, values);
            if (rowId > 0) {  // 判断插入是否执行成功
                // 如果添加成功，利用新记录的行号生成新的地址
                newUri = ContentUris.withAppendedId(UserInfoContent.CONTENT_URI, rowId);
                // 通知监听器，数据已经改变
                getContext().getContentResolver().notifyChange(newUri, null);
            }
            db.close();  // 关闭 SQLite 数据库连接
        }
        return uri;
    }

    // 创建 ContentProvider 时调用，可在此获取具体的数据库帮助器实例
    public boolean onCreate() {
        userDB = UserDBHelper.getInstance(getContext(), 1);
        return false;
    }

    // 根据指定条件查询数据库
    public Cursor query(Uri uri, String[] projection, String selection,
                        String[] selectionArgs, String sortOrder) {
```

```
        Cursor cursor = null;
        if (uriMatcher.match(uri) == USER_INFO) {
            // 获取 SQLite 数据库的读连接
            SQLiteDatabase db = userDB.getReadableDatabase();
            // 执行 SQLite 的查询操作
            cursor = db.query(UserInfoContent.TABLE_NAME,
                    projection, selection, selectionArgs, null, null, sortOrder);
            // 设置内容解析器的监听
            cursor.setNotificationUri(getContext().getContentResolver(), uri);
        }
        return cursor;
    }

    // 获取 Uri 数据的访问类型，暂未实现
    public String getType(Uri uri) {}

    // 更新数据，暂未实现
    public int update(Uri uri, ContentValues values, String selection, String[] selectionArgs) {}
}
```

既然内容提供器是四大组件之一，就得在 AndroidManifest.xml 中注册它的定义，并开放外部访问权限，注册代码如下：

```
        <provider
            android:name=".provider.UserInfoProvider"
            android:authorities="com.example.storage.provider.UserInfoProvider"
            android:enabled="true"
            android:exported="true" />
```

注册完毕后就完成了服务端 App 的封装工作，接下来可由其他 App 进行数据存取。

4.5.2　内容解析器 ContentResolver

前面提到了利用 ContentProvider 实现服务端 App 的数据封装，如果客户端 App 想访问对方的内部数据，就要通过内容解析器 ContentResolver 访问。内容解析器是客户端 App 操作服务端数据的工具，相对应的内容提供器是服务端的数据接口。要获取 ContentResolver 对象，在 Activity 代码中调用 getContentResolver 方法即可。

ContentResolver 提供的方法与 ContentProvider 是一一对应的，比如 query、insert、update、delete、getType 等方法，连方法的参数类型都一模一样。其中，最常用的是 query 函数，调用该函数返回一个游标 Cursor 对象，这个游标与 SQLite 的游标是一样的，想必读者早已用得炉火纯青。

下面是 query 方法的具体参数说明（依顺序排列）。

- uri：Uri 类型，可以理解为本次操作的数据表路径。

- projection：字符串数组类型，指定将要查询的字段名称列表。
- selection：字符串类型，指定查询条件。
- selectionArgs：字符串数组类型，指定查询条件中的参数取值列表。
- sortOrder：字符串类型，指定排序条件。

针对前面 UserInfoProvider 提供的数据接口，下面使用 ContentResolver 在客户端添加用户信息，代码如下：

```
// 添加一条用户记录
private void addUser(ContentResolver resolver, UserInfo user) {
    ContentValues name = new ContentValues();
    name.put("name", user.name);
    name.put("age", user.age);
    name.put("height", user.height);
    name.put("weight", user.weight);
    name.put("married", false);
    name.put("update_time", DateUtil.getNowDateTime(""));
    // 通过内容解析器往指定 Uri 中添加用户信息
    resolver.insert(UserInfoContent.CONTENT_URI, name);
}
```

下面是使用 ContentResolver 在客户端查询所有用户信息的代码：

```
// 读取所有的用户记录
private String readAllUser(ContentResolver resolver) {
    ArrayList<UserInfo> userArray = new ArrayList<UserInfo>();
    // 通过内容解析器从指定 Uri 中获取用户记录的游标
    Cursor cursor = resolver.query(UserInfoContent.CONTENT_URI, null, null, null, null);
    // 循环取出游标指向的每条用户记录
    while (cursor.moveToNext()) {
        UserInfo user = new UserInfo();
        user.name = cursor.getString(cursor.getColumnIndex(UserInfoContent.USER_NAME));
        user.age = cursor.getInt(cursor.getColumnIndex(UserInfoContent.USER_AGE));
        user.height = cursor.getInt(cursor.getColumnIndex(UserInfoContent.USER_HEIGHT));
        user.weight = cursor.getFloat(cursor.getColumnIndex(UserInfoContent.USER_WEIGHT));
        userArray.add(user);   // 添加到用户信息队列
    }
    cursor.close();   // 关闭数据库游标
    String result = "";
    for (UserInfo user : userArray) {
        // 遍历用户信息队列，逐个拼接到结果字符串
        result = String.format("%s%s   年龄%d    身高%d    体重%f\n", result,
                user.name, user.age, user.height, user.weight);
    }
    return result;
```

```
    }
```

添加用户信息的效果如图 4-19 所示，一开始服务端的用户表不存在用户记录，客户端使用 ContentResolver 添加一条记录后，服务端的用户记录数返回 1。用户信息的查询明细如图 4-20 所示，点击页面上的用户记录数量文字，弹出一个对话框，提示当前找到的所有用户的明细数据，包括姓名、年龄、身高、体重等信息。

图 4-19　利用内容提供器添加用户信息

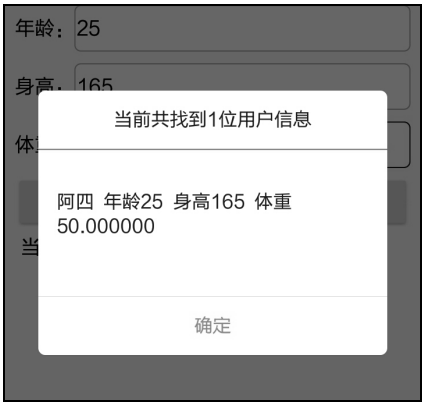

图 4-20　利用内容解析器查询获得用户信息

在实际开发中，普通 App 很少会开放数据接口给其他应用访问，作为服务端接口的 ContentProvider 基本用不到。内容组件能够派上用场的情况往往是 App 想要访问系统应用的通信数据，比如查看联系人、短信、通话记录，以及对这些通信信息进行增、删、改、查。

下面是使用 ContentResolver 添加联系人信息的代码片段，此时访问的数据来源变成了系统自带的 raw_contacts：

```
// 往手机中添加一个联系人信息（包括姓名、电话号码、电子邮箱）
public static void addContacts(ContentResolver resolver, Contact contact) {
    // 构建一个指向系统联系人提供器的 Uri 对象
    Uri raw_uri = Uri.parse("content://com.android.contacts/raw_contacts");
    // 创建新的配对
    ContentValues values = new ContentValues();
    // 往 raw_contacts 中添加联系人记录，并获取添加后的联系人编号
    long contactId = ContentUris.parseId(resolver.insert(raw_uri, values));
    // 构建一个指向系统联系人数据的 Uri 对象
    Uri uri = Uri.parse("content://com.android.contacts/data");
    // 创建新的配对
    ContentValues name = new ContentValues();
    // 往配对中添加联系人编号
    name.put("raw_contact_id", contactId);
    // 往配对中添加数据类型为“姓名”
    name.put("mimetype", "vnd.android.cursor.item/name");
    // 往配对中添加联系人的姓名
    name.put("data2", contact.name);
```

```
        // 往 data 中添加联系人的姓名
        resolver.insert(uri, name);
        // 创建新的配对
        ContentValues phone = new ContentValues();
        // 往配对中添加联系人编号
        phone.put("raw_contact_id", contactId);
        // 往配对中添加数据类型为“电话号码”
        phone.put("mimetype", "vnd.android.cursor.item/phone_v2");
        phone.put("data2", "2");
        // 往配对中添加联系人的电话号码
        phone.put("data1", contact.phone);
        // 往 data 中添加联系人的电话号码
        resolver.insert(uri, phone);
        // 创建新的配对
        ContentValues email = new ContentValues();
        // 往配对中添加联系人编号
        email.put("raw_contact_id", contactId);
        // 往配对中添加数据类型为“电子邮箱”
        email.put("mimetype", "vnd.android.cursor.item/email_v2");
        email.put("data2", "2");
        // 往配对中添加联系人的电子邮箱
        email.put("data1", contact.email);
        // 往 data 中添加联系人的电子邮箱
        resolver.insert(uri, email);
    }
```

注意上述代码用了 4 条 insert 语句，但业务上只添加了一个联系人信息。这样处理有一个问题，就是 4 个 insert 操作不在同一个事务中，要是中间某步 insert 操作失败，那么之前插入成功的记录就无法自动回滚，从而产生垃圾数据。

为了避免这种情况的发生，Android 提供了内容操作器 ContentProviderOperation 进行批量数据的处理，即在一个请求中封装多条记录的修改动作，然后一次性提交给服务端，从而实现在一个事务中完成多条数据的更新操作。即使某条记录处理失败，ContentProviderOperation 也能根据事务一致性原则自动回滚本事务已经执行的修改操作。

下面是使用 ContentProviderOperation 批量添加联系人信息的代码片段：

```
    // 往手机中一次性添加一个联系人信息（包括主记录、姓名、电话号码、电子邮箱）
    public static void addFullContacts(ContentResolver resolver, Contact contact) {
        // 构建一个指向系统联系人提供器的 Uri 对象
        Uri raw_uri = Uri.parse("content://com.android.contacts/raw_contacts");
        // 构建一个指向系统联系人数据的 Uri 对象
        Uri uri = Uri.parse("content://com.android.contacts/data");
        // 创建一个插入联系人主记录的内容操作器
        ContentProviderOperation op_main = ContentProviderOperation
```

```
                .newInsert(raw_uri).withValue("account_name", null).build();
        // 创建一个插入联系人姓名记录的内容操作器
        ContentProviderOperation op_name = ContentProviderOperation
                .newInsert(uri).withValueBackReference("raw_contact_id", 0)
                .withValue("mimetype", "vnd.android.cursor.item/name")
                .withValue("data2", contact.name).build();
        // 创建一个插入联系人电话号码记录的内容操作器
        ContentProviderOperation op_phone = ContentProviderOperation
                .newInsert(uri).withValueBackReference("raw_contact_id", 0)
                .withValue("mimetype", "vnd.android.cursor.item/phone_v2")
                .withValue("data2", "2").withValue("data1", contact.phone).build();
        // 创建一个插入联系人电子邮箱记录的内容操作器
        ContentProviderOperation op_email = ContentProviderOperation
                .newInsert(uri).withValueBackReference("raw_contact_id", 0)
                .withValue("mimetype", "vnd.android.cursor.item/email_v2")
                .withValue("data2", "2").withValue("data1", contact.email).build();
        // 声明一个内容操作器的队列，并将上面四个操作器添加到该队列中
        ArrayList<ContentProviderOperation> operations = new ArrayList<ContentProviderOperation>();
        operations.add(op_main);
        operations.add(op_name);
        operations.add(op_phone);
        operations.add(op_email);
        try {
            // 批量提交四个内容操作器所做的修改
            resolver.applyBatch("com.android.contacts", operations);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
```

添加联系人信息的效果如图 4-21 和图 4-22 所示。其中，图 4-21 所示为添加之前的截图，此时联系人个数为 157 位；图 4-22 所示为添加成功之后的截图，此时联系人个数为 158 位。

图 4-21　联系人添加之前的界面

图 4-22　联系人添加之后的界面

4.5.3 内容观察器 ContentObserver

ContentResolver 获取数据采用的是主动查询方式，有查询就有数据，没查询就没数据。有时我们不但要获取以往的数据，还要实时获取新增的数据，最常见的业务场景是短信验证码。电商 App 经常在用户注册或付款时发送验证码短信，为了给用户省事，App 通常会监控手机刚收到的验证码数字，并自动填入验证码输入框。这时就用到了内容观察器 ContentObserver，给目标内容注册一个观察器，目标内容的数据一旦发生变化，观察器规定好的动作马上触发，从而执行开发者预先定义的代码。

内容观察器的用法与内容提供器类似，也要从 ContentObserver 派生一个观察器类，然后通过 ContentResolver 对象调用相应的方法注册或注销观察器。下面是 ContentResolver 与观察器有关的方法说明。

- registerContentObserver：注册内容观察器。
- unregisterContentObserver：注销内容观察器。
- notifyChange：通知内容观察器发生了数据变化。

为了让读者更好理解，下面举一个实际应用的例子。手机号码的每月流量限额一般由用户手动配置，但流量限额其实是由移动运营商指定的。以中国移动为例，只要发送流量校准短信给运营商客服号码（如发送 18 到 10086），运营商就会给用户发送本月的流量数据，包括月流量额度、已使用流量、未使用流量等信息。手机 App 只需监控 10086 发送的短信内容，即可自动获取手机号码的月流量额度，无须用户手工配置。

下面是利用 ContentObserver 实现流量校准的代码片段：

```
private Handler mHandler = new Handler();   // 声明一个处理器对象
private SmsGetObserver mObserver;   // 声明一个短信获取的观察器对象
private static Uri mSmsUri;   // 声明一个系统短信提供器的 Uri 对象
private static String[] mSmsColumn;   // 声明一个短信记录的字段数组

// 初始化短信观察器
private void initSmsObserver() {
    mSmsUri = Uri.parse("content://sms");
    mSmsColumn = new String[]{"address", "body", "date"};
    // 创建一个短信观察器对象
    mObserver = new SmsGetObserver(this, mHandler);
    // 给指定 Uri 注册内容观察器，一旦 Uri 内部发生数据变化，就触发观察器的 onChange 方法
    getContentResolver().registerContentObserver(mSmsUri, true, mObserver);
}

// 在页面销毁时触发
protected void onDestroy() {
    // 注销内容观察器
    getContentResolver().unregisterContentObserver(mObserver);
    super.onDestroy();
```

```
        }

        // 定义一个短信获取的观察器
        private static class SmsGetObserver extends ContentObserver {
            private Context mContext;   // 声明一个上下文对象
            public SmsGetObserver(Context context, Handler handler) {
                super(handler);
                mContext = context;
            }

            // 观察到短信的内容提供器发生变化时触发
            public void onChange(boolean selfChange) {
                String sender = "", content = "";
                // 构建一个查询短信的条件语句，这里使用移动号码测试，故而查找 10086 发来的短信
                String selection = String.format("address='10086' and date>%d", System.currentTimeMillis() -
1000 * 60 * 60);
                // 通过内容解析器获取符合条件的结果集游标
                Cursor cursor = mContext.getContentResolver().query(
                        mSmsUri, mSmsColumn, selection, null, " date desc");
                // 循环取出游标所指向的所有短信记录
                while (cursor.moveToNext()) {
                    sender = cursor.getString(0);
                    content = cursor.getString(1);
                    break;
                }
                cursor.close();   // 关闭数据库游标
                mCheckResult = String.format("发送号码：%s\n 短信内容：%s", sender, content);
                // 依次解析流量校准短信里面的各项流量数值，并拼接流量校准的结果字符串
                String flow = String.format("流量校准结果如下：\n\t 总流量为：%s\n\t 已使用：%s" +
                        "\n\t 剩余：%s", findFlow(content, "总流量为", "MB"),
                        findFlow(content, "已使用", "MB"), findFlow(content, "剩余", "MB"));
                if (tv_check_flow != null) {   // 离开该页面时就不再显示流量信息
                    // 把流量校准结果显示到文本视图 tv_check_flow 上面
                    tv_check_flow.setText(flow);
                }
                super.onChange(selfChange);
            }
        }

        // 解析流量校准短信里面的流量数值
        private static String findFlow(String sms, String begin, String end) {
            int begin_pos = sms.indexOf(begin);
            if (begin_pos < 0) {
```

```
                return "未获取";
            }
            String sub_sms = sms.substring(begin_pos);
            int end_pos = sub_sms.indexOf(end);
            if (end_pos < 0) {
                return "未获取";
            }
            return sub_sms.substring(begin.length(), end_pos + end.length());
        }
```

流量校准的效果如图 4-23 和图 4-24 所示。其中，图 4-23 所示为用户实际收到的短信内容，图 4-24 所示为 App 监视短信并解析完成的流量数据页面。

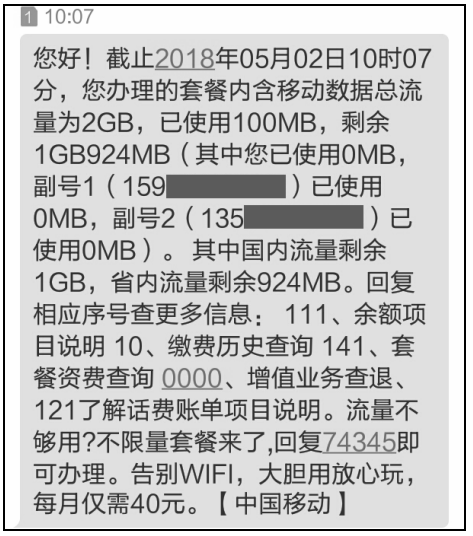

图 4-23　用户收到的短信内容

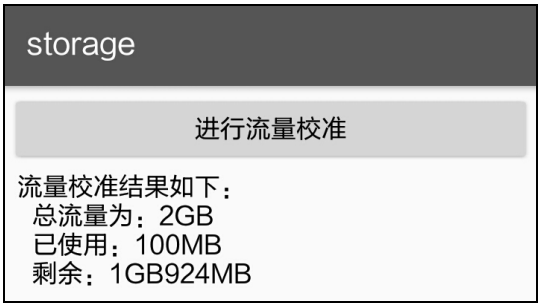

图 4-24　内容观察器监视并解析得到的流量信息

总结一下在 Content 组件经常使用的系统 URI，详细的 URI 取值说明见表 4-3。

表4-3　常用的系统URI取值说明

内容名称	URI 常量名	实际路径
联系人	ContactsContract.Contacts.CONTENT_URI	content://com.android.contacts/contacts
联系人电话	ContactsContract.CommonDataKinds.Phone.CONTENT_URI	content://com.android.contacts/data/phones
联系人邮箱	ContactsContract.CommonDataKinds.Email.CONTENT_URI	content://com.android.contacts/data/emails
SIM 卡联系人		content://icc/adn
短信	Telephony.Sms.CONTENT_URI	content://sms
彩信	Telephony.Mms.CONTENT_URI	content://mms
通话记录	CallLog.Calls.CONTENT_URI	content://call_log/calls

（续表）

内容名称	URI 常量名	实际路径
收件箱	Telephony.Sms.Inbox.CONTENT_URI（短信相关的 URI）	content://sms/inbox
已发送	Telephony.Sms.Sent.CONTENT_URI（短信相关的 URI）	content://sms/sent
草稿箱	Telephony.Sms.Draft.CONTENT_URI（短信相关的 URI）	content://sms/draft
发件箱	Telephony.Sms.Outbox.CONTENT_URI（短信相关的 URI）	content://sms/outbox
发送失败	无	content://sms/failed
待发送列表	无。比如开启飞行模式后，该短信就在待发送列表里	content://sms/queued

4.6　实战项目：购物车

购物车的应用面很广，凡是电商 App 都可以看到购物车的身影。本章以购物车为实战项目，除了购物车使用广泛的特点，还因为购物车用到多种存储方式。现在我们开启购物车的体验之旅吧！

4.6.1　设计思路

先来看常见的购物车的外观。第一次进入购物车频道，购物车里面是空的，如图 4-25 所示。接着去商品频道选购手机，随便挑几款加入购物车，然后返回购物车，即可看到购物车里的商品列表，有商品图片、名称、数量、单价、总价等信息，如图 4-26 所示。

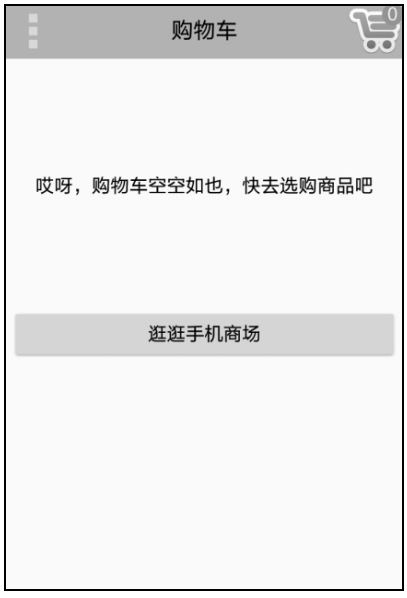

图 4-25　空空如也的购物车

图 4-26　购物车的商品列表

购物车的存在感很强，并不仅仅在购物车页面才能看到。往往在商场频道，甚至某个商

品详情页面，都会看到某个角落冒出一个购物车图标。一旦有新商品加入购物车，购物车图标上的商品数量就立马加一。当然，用户也可以点击购物车图标直接跳转到购物车页面。商品频道除了商品列表外，页面右上角还有一个购物车图标，这个图标有时在页面右上角，有时又在页面右下角，如图 4-27 所示。商品详情页面通常也有购物车图标，如果用户在详情页面把商品加入购物车，那么图标上的数字也会加一，如图 4-28 所示。

现在看看购物车到底采取了哪些存储方式。

- 数据库 SQLite：最直观的是数据库，购物车里的商品列表一定放在 SQLite 中，增删改查都少不了 SQLite。

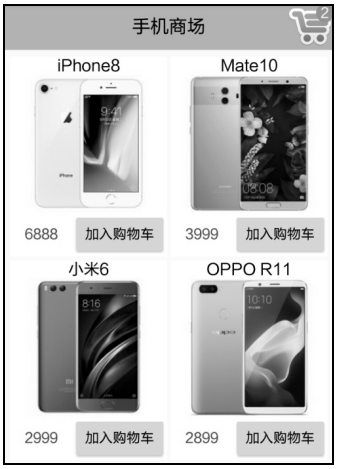

图 4-27　手机商场的商品列表

图 4-28　商品详情页面

- 共享参数 SharedPreferences：注意不同页面的右上角购物车图标都有数字，表示购物车中的商品数量，商品数量建议保存在共享参数中。因为每个页面都要显示商品数量，如果每次都到数据库中执行 count 操作，就会很消耗资源。因为商品数量需要持久地存储，所以不适合放在全局内存中，不然下次启动 App 时，内存中的变量又从 0 开始。
- SD 卡文件：通常情况下，商品图片来自于电商平台的服务器，这年头流量是很宝贵的，可是图片恰恰很耗流量（尤其是大图）。从用户的钱包着想，App 得把下载的图片保存在 SD 卡中。这样一来，下次用户访问商品详情页面时，App 便能直接从 SD 卡获取商品图片，不但不花流量而且加快浏览速度，一举两得。
- 全局内存：访问 SD 卡的图片文件固然是个好主意，然而商品频道、购物车频道等可能在一个页面展示多张商品小图，如果每张小图都要访问 SD 卡，频繁的 SD 卡读写操作也很耗资源。更好的办法是把商品小图加载进全局内存，这样直接从内存中获取图片，高效又快速。之所以不把商品大图放入全局内存，是因为大图很耗空间，一不小心就会占用几十兆内存。

不找不知道，一找吓一跳，原来购物车用到了这么多种存储方式。

4.6.2　小知识：菜单 Menu

之前的章节在进行某项控制操作时一般由按钮控件触发。如果页面上需要支持多个控制

操作，比如去商场购物、清空购物车、查看商品详情、删除指定商品等，就得在页面上添加多个按钮。如此一来，App 页面显得杂乱无章，满屏按钮既碍眼又不便操作。这时，就可以使用菜单控件。

菜单无论在哪里都是常用控件，Android 的菜单主要分两种，一种是选项菜单 OptionMenu，通过按菜单键或点击事件触发，对应 Windows 上的开始菜单；另一种是上下文菜单 ContextMenu，通过长按事件触发，对应 Windows 上的右键菜单。无论是哪种菜单，都有对应的菜单布局文件，就像每个活动页面都有一个布局文件一样。不同的是页面的布局文件放在 res/layout 目录下，菜单的布局文件放在 res/menu 目录下。

下面来看 Android 的选项菜单和上下文菜单。

1. 选项菜单 OptionMenu

弹出选项菜单的途径有 3 种：

（1）按菜单键。
（2）在代码中手动打开选项菜单，即调用 openOptionsMenu 方法。
（3）按工具栏右侧的溢出菜单按钮，这个在第 7 章介绍工具栏时进行介绍。

实现选项菜单的功能需要重写以下两种方法。

- onCreateOptionsMenu：在页面打开时调用。需要指定菜单列表的 XML 文件。
- onOptionsItemSelected：在列表的菜单项被选中时调用。需要对不同的菜单项做分支处理。

下面是菜单布局文件的代码，很简单，就是 menu 与 item 的组合排列：

```
<menu xmlns:android="http://schemas.android.com/apk/res/android" >
    <item
        android:id="@+id/menu_change_time"
        android:orderInCategory="1"
        android:title="改变时间"/>
    <item
        android:id="@+id/menu_change_color"
        android:orderInCategory="8"
        android:title="改变颜色"/>
    <item
        android:id="@+id/menu_change_bg"
        android:orderInCategory="9"
        android:title="改变背景"/>
</menu>
```

接下来是使用选项菜单的代码片段：

```
    // 在选项菜单的菜单界面创建时调用
    public boolean onCreateOptionsMenu(Menu menu) {
        // 从 menu_option.xml 中构建菜单界面布局
```

```
        getMenuInflater().inflate(R.menu.menu_option, menu);
        return true;
    }

    // 在选项菜单的菜单项选中时调用
    public boolean onOptionsItemSelected(MenuItem item) {
        int id = item.getItemId();  // 获取菜单项的编号
        if (id == R.id.menu_change_time) {  // 点击了菜单项“改变时间”
            setRandomTime();
        } else if (id == R.id.menu_change_color) {  // 点击了菜单项“改变颜色”
            tv_option.setTextColor(getRandomColor());
        } else if (id == R.id.menu_change_bg) {  // 点击了菜单项“改变背景”
            tv_option.setBackgroundColor(getRandomColor());
        }
        return true;
    }
```

按菜单键和调用 openOptionsMenu 方法弹出的选项菜单都是在页面下方，如图 4-29 所示。

图 4-29　选项菜单的菜单列表

2. 上下文菜单 ContextMenu

弹出上下文菜单的途径有两种：

（1）默认在某个控件被长按时弹出。通常在 onStart 函数中加入 registerForContextMenu 方法为指定控件注册上下文菜单，在 onStop 函数中加入 unregisterForContextMenu 方法为指定控件注销上下文菜单。

（2）在除长按事件之外的其他事件中打开上下文菜单。先执行 registerForContextMenu 方法注册菜单，然后执行 openContextMenu 方法打开菜单，最后执行 unregisterForContextMenu 方法注销菜单。

实现上下文菜单的功能需要重写以下两种方法。

- onCreateContextMenu：在此指定菜单列表的 XML 文件，作为上下文菜单列表项的来源。
- onContextItemSelected：在此对不同的菜单项做分支处理。

上下文菜单的布局文件格式同选项菜单，下面是使用上下文菜单的代码片段：

```
    // 在页面恢复时调用
```

```
protected void onResume() {
    // 给文本视图 tv_context 注册上下文菜单。
    // 注册之后，只要长按该控件，App 就会自动打开上下文菜单
    registerForContextMenu(tv_context);
    super.onResume();
}

// 在页面暂停时调用
protected void onPause() {
    // 给文本视图 tv_context 注销上下文菜单
    unregisterForContextMenu(tv_context);
    super.onPause();
}

// 在上下文菜单的菜单界面创建时调用
public void onCreateContextMenu(ContextMenu menu, View v, ContextMenuInfo menuInfo) {
    // 从 menu_option.xml 中构建菜单界面布局
    getMenuInflater().inflate(R.menu.menu_option, menu);
}

// 在上下文菜单的菜单项选中时调用
public boolean onContextItemSelected(MenuItem item) {
    int id = item.getItemId();  // 获取菜单项的编号
    if (id == R.id.menu_change_time) {  // 点击了菜单项“改变时间”
        setRandomTime();
    } else if (id == R.id.menu_change_color) {  // 点击了菜单项“改变颜色”
        tv_context.setTextColor(getRandomColor());
    } else if (id == R.id.menu_change_bg) {  // 点击了菜单项“改变背景”
        tv_context.setBackgroundColor(getRandomColor());
    }
    return true;
}
```

上下文菜单的菜单列表固定显示在页面中部，菜单外的其他页面区域颜色会变深，具体效果如图 4-30 所示。

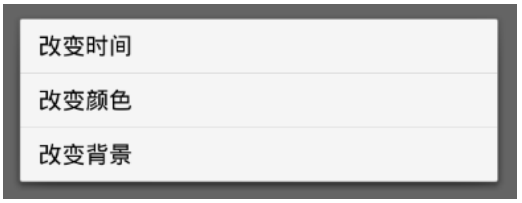

图 4-30　上下文菜单的菜单列表

4.6.3 代码示例

这一章的编码开始有些复杂了，不但有各种控件和布局的操作，还有 4 种存储方式的使用，再加上 Activity 与 Application 两大组件的运用，已然是一个正规 App 的雏形。

编码过程分为 4 步（增加的一步是对 AndroidManifest.xml 认真配置）：

步骤 01 想好代码文件与布局文件的名称，比如购物车页面的代码文件取名 ShoppingCartActivity.java，对应的布局文件名是 activity_shopping_cart.xml；商场频道页面的代码文件取名 ShoppingChannelActivity.java，对应的布局文件名是 activity_shopping_channel.xml；商品详情页面的代码文件取名 ShoppingDetailActivity，对应的布局文件名是 activity_shopping_detail.xml；另有一个全局应用的代码文件 MainApplication.java。

步骤 02 在 AndroidManifest.xml 中补充相应配置，主要有以下 3 点：

（1）注册 3 个页面的 acitivity 节点，注册代码如下：

```
<activity android:name=".ShoppingCartActivity" android:theme="@style/AppBaseTheme" />
<activity android:name=".ShoppingChannelActivity" />
<activity android:name=".ShoppingDetailActivity" />
```

（2）给 application 补充 name 属性，值为 MainApplication，举例如下：

```
android:name=".MainApplication"
```

（3）声明 SD 卡的操作权限，主要补充下面 3 行权限配置：

```
<!-- SD 卡读写权限 -->
<uses-permission android:name="android.permission.WRITE_EXTERNAL_STORAGE" />
<uses-permission android:name="android.permission.READ_EXTERNAL_STORAG" />
<uses-permission android:name="android.permission.MOUNT_UNMOUNT_FILESYSTEMS" />
```

步骤 03 res 目录下的 XML 文件编写也多了起来，主要工作包括：

（1）在 res/layout 目录下创建布局文件 activity_shopping_cart.xml、activity_shopping_channel.xml、activity_shopping_detail.xml，分别根据页面效果图编写 3 个页面的布局定义文件。

（2）在 res/menu 目录下创建菜单布局文件 menu_cart.xml 和 menu_goods.xml，分别用于购物车的选项菜单和商品项的上下文菜单。

（3）在 values/styles.xml 中补充下面的样式定义，给不带导航栏的购物车页面使用：

```
<style name="AppBaseTheme" parent="Theme.AppCompat.Light" />
```

步骤 04 在项目的包名目录下创建类 MainApplication、ShoppingCartActivity、ShoppingChannelActivity 和 ShoppingDetailActivity，并填入具体的控件操作与业务逻辑代码。

购物车项目的完整代码参见本书附录源码 storage 模块的 ShoppingCartActivity.java、ShoppingChannelActivity.java 和 ShoppingDetailActivity.java。下面列出两块与存储技术有关的代码片段，首先是商场页面把商品加入购物车的逻辑处理，主要涉及到共享参数和 SQLite 数

据库的运用，关键代码如下所示：

```
private int mCount;   // 购物车中的商品数量
private GoodsDBHelper mGoodsHelper;   // 声明一个商品数据库的帮助器对象
private CartDBHelper mCartHelper;   // 声明一个购物车数据库的帮助器对象

// 把指定编号的商品添加到购物车
private void addToCart(long goods_id) {
    mCount++;
    tv_count.setText("" + mCount);
    // 把购物车中的商品数量写入共享参数
    SharedUtil.getIntance(this).writeShared("count", "" + mCount);
    // 根据商品编号查询购物车数据库中的商品记录
    CartInfo info = mCartHelper.queryByGoodsId(goods_id);
    if (info != null) {   // 购物车已存在该商品记录
        info.count++;   // 该商品的数量加一
        info.update_time = DateUtil.getNowDateTime("");
        // 更新购物车数据库中的商品记录信息
        mCartHelper.update(info);
    } else {   // 购物车不存在该商品记录
        info = new CartInfo();
        info.goods_id = goods_id;
        info.count = 1;
        info.update_time = DateUtil.getNowDateTime("");
        // 往购物车数据库中添加一条新的商品记录
        mCartHelper.insert(info);
    }
}

// 在页面恢复时调用
protected void onResume() {
    super.onResume();
    // 获取共享参数保存的购物车中的商品数量
    mCount = Integer.parseInt(SharedUtil.getIntance(this).readShared("count", "0"));
    tv_count.setText("" + mCount);
    // 获取商品数据库的帮助器对象
    mGoodsHelper = GoodsDBHelper.getInstance(this, 1);
    // 打开商品数据库的读连接
    mGoodsHelper.openReadLink();
    // 获取购物车数据库的帮助器对象
    mCartHelper = CartDBHelper.getInstance(this, 1);
    // 打开购物车数据库的写连接
    mCartHelper.openWriteLink();
```

```
        // 展示商品列表
        showGoods();
    }

    // 在页面暂停时调用
    protected void onPause() {
        super.onPause();
        // 关闭商品数据库的数据库连接
        mGoodsHelper.closeLink();
        // 关闭购物车数据库的数据库连接
        mCartHelper.closeLink();
    }
```

然后是购物车页面模拟从网络上下载商品图片，并构建简单的图片缓存机制的逻辑处理，主要涉及到 SD 卡文件操作与 Application 全局变量的运用，关键代码如下所示：

```
    private String mFirst = "true";  // 是否首次打开

    // 模拟网络数据，初始化数据库中的商品信息
    private void downloadGoods() {
        // 获取共享参数保存的是否首次打开参数
        mFirst = SharedUtil.getIntance(this).readShared("first", "true");
        // 获取当前 App 的私有存储路径
        String path = MainApplication.getInstance().getExternalFilesDir(
                Environment.DIRECTORY_DOWNLOADS).toString() + "/";
        if (mFirst.equals("true")) {  // 如果是首次打开
            ArrayList<GoodsInfo> goodsList = GoodsInfo.getDefaultList();
            for (int i = 0; i < goodsList.size(); i++) {
                GoodsInfo info = goodsList.get(i);
                // 往商品数据库插入一条该商品的记录
                long rowid = mGoodsHelper.insert(info);
                info.rowid = rowid;
                // 往全局内存写入商品小图
                Bitmap thumb = BitmapFactory.decodeResource(getResources(), info.thumb);
                MainApplication.getInstance().mIconMap.put(rowid, thumb);
                String thumb_path = path + rowid + "_s.jpg";
                FileUtil.saveImage(thumb_path, thumb);
                info.thumb_path = thumb_path;
                // 往 SD 卡保存商品大图
                Bitmap pic = BitmapFactory.decodeResource(getResources(), info.pic);
                String pic_path = path + rowid + ".jpg";
                FileUtil.saveImage(pic_path, pic);
                pic.recycle();
                info.pic_path = pic_path;
```

```
                    // 更新商品数据库中该商品记录的图片路径
                    mGoodsHelper.update(info);
                }
            } else {  // 不是首次打开
                // 查询商品数据库中所有商品记录
                ArrayList<GoodsInfo> goodsArray = mGoodsHelper.query("1=1");
                for (int i = 0; i < goodsArray.size(); i++) {
                    GoodsInfo info = goodsArray.get(i);
                    // 从指定路径读取图片文件的位图数据
                    Bitmap thumb = BitmapFactory.decodeFile(info.thumb_path);
                    // 把该位图对象保存到应用实例的全局变量中
                    MainApplication.getInstance().mIconMap.put(info.rowid, thumb);
                }
            }
            // 把是否首次打开写入共享参数
            SharedUtil.getIntance(this).writeShared("first", "false");
        }
```

4.7　小　　结

本章主要介绍了 Android 常用的几种数据存储方式，包括共享参数 SharedPreferences 的键值对存取、数据库 SQLite 的关系型数据存取、SD 卡的文件写入与读取操作（含文本文件读写和图片文件读写）、App 全局内存的读写以及为实现全局内存而学习的 Application 组件的生命周期及其用法、ContentProvider 内容组件的用法（内容提供器、内容解析器、内容操作器、内容观察器）。最后设计了一个实战项目“购物车”，通过该项目的编码进一步复习巩固本章几种存储方式的使用，另外介绍了选项菜单和上下文菜单的基本用法。

通过本章的学习，读者应该能够掌握以下 4 种开发技能：

（1）学会共享参数 SharedPreferences、数据库 SQLite、SD 卡文件、全局内存、内容提供器共 5 种存储方式的用法。

（2）学会应用组件 Application 的用法。

（3）学会内容组件 ContentProvider 的用法，如封装数据的对外接口，对开放内容接口的系统数据进行查询、修改和监视操作。

（4）学会选项菜单和上下文菜单的基本用法。