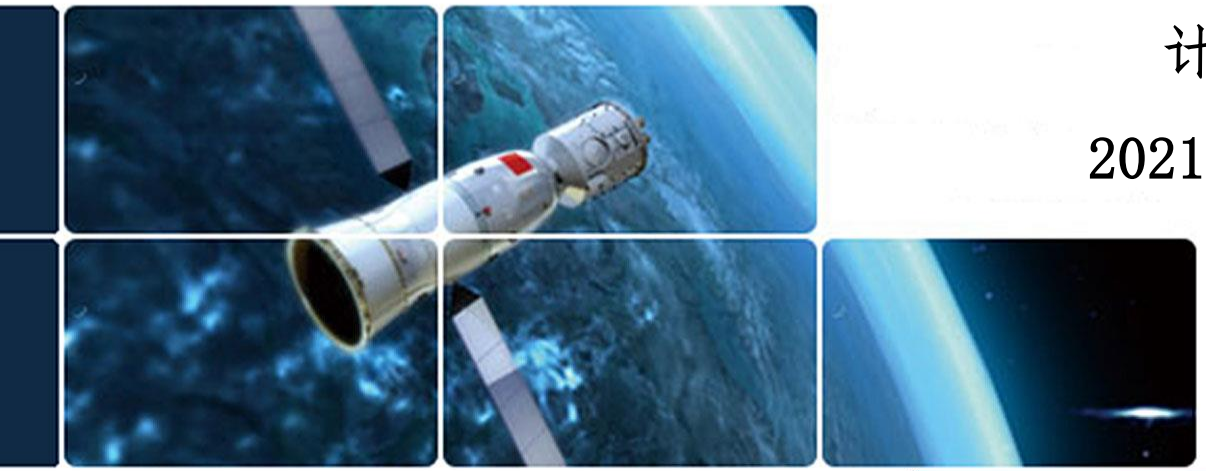
第7讲 移动互联网应用

存储设计和实现



计算学部

2021年12月2日

移动互联网应用存储设计和实现

登录数据

业务数据

连接数据

Q1：数据存储在哪里?

A1：终端内存、终端*Flash*、数据库服务器

Q2：数据如何组织?

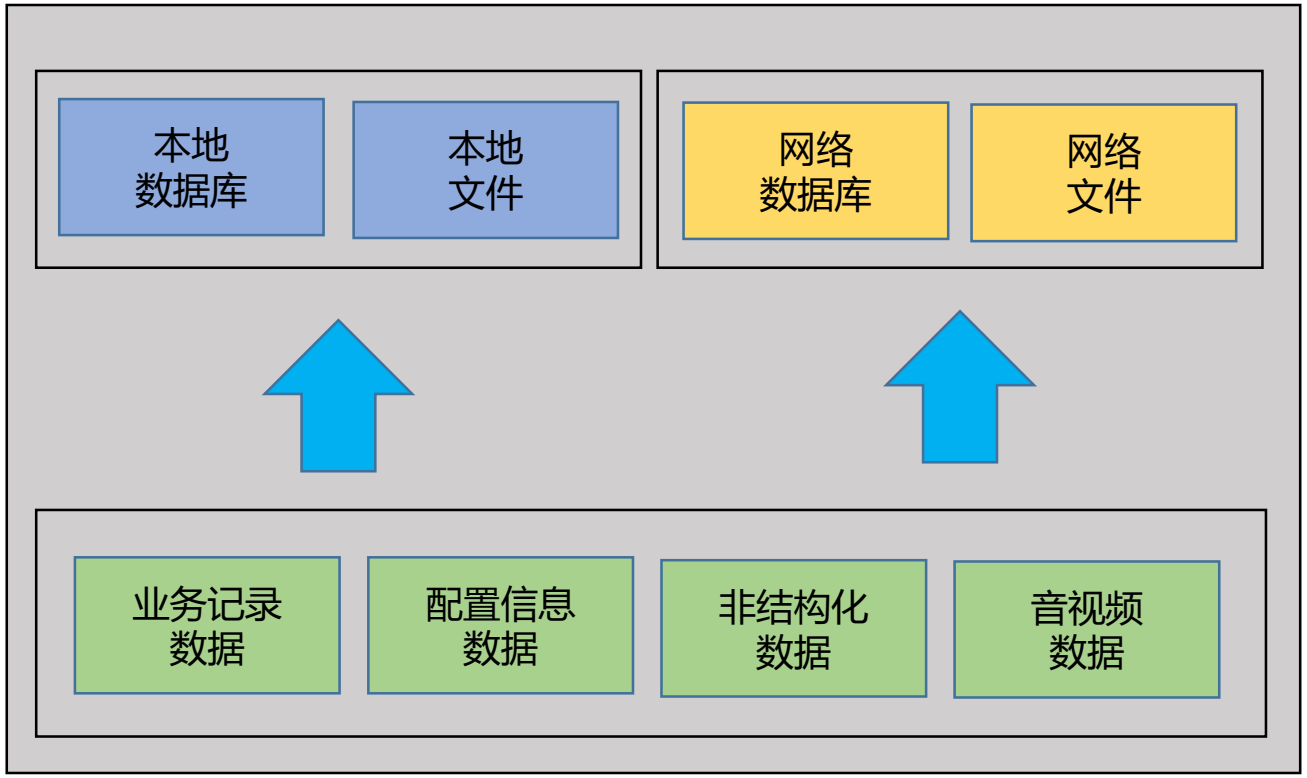
A2：数组、对象，键值对、实体表、关系表

Q3：数据如何关联到UI?

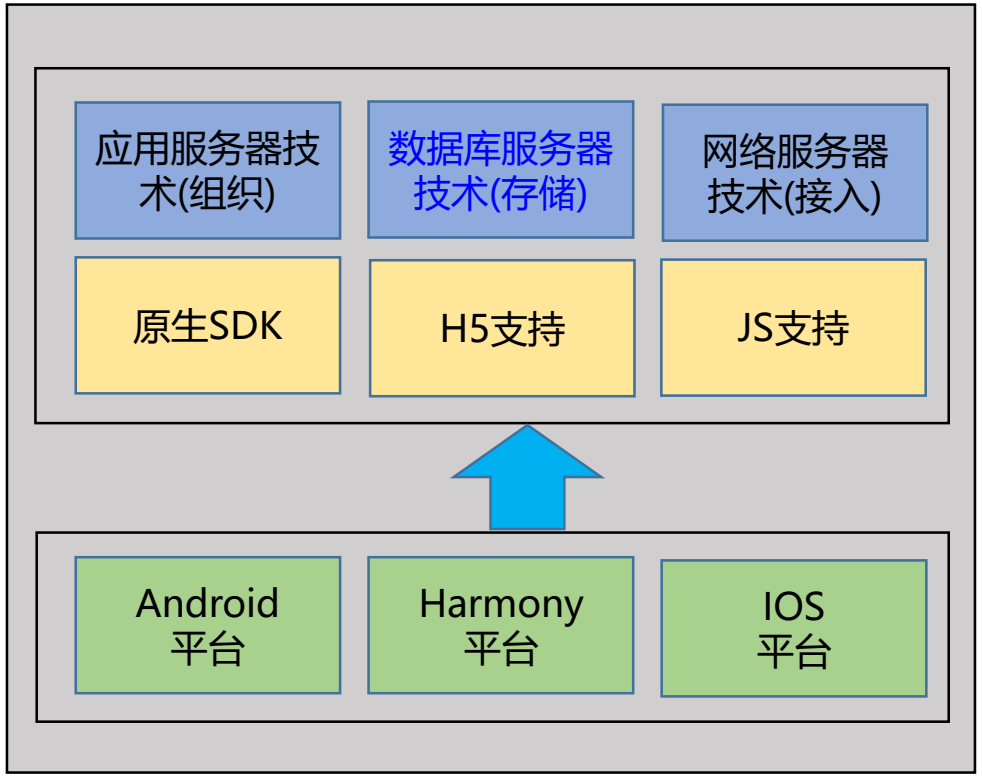
A3：简单变量、结果集等

移动互联网应用存储设计和实现

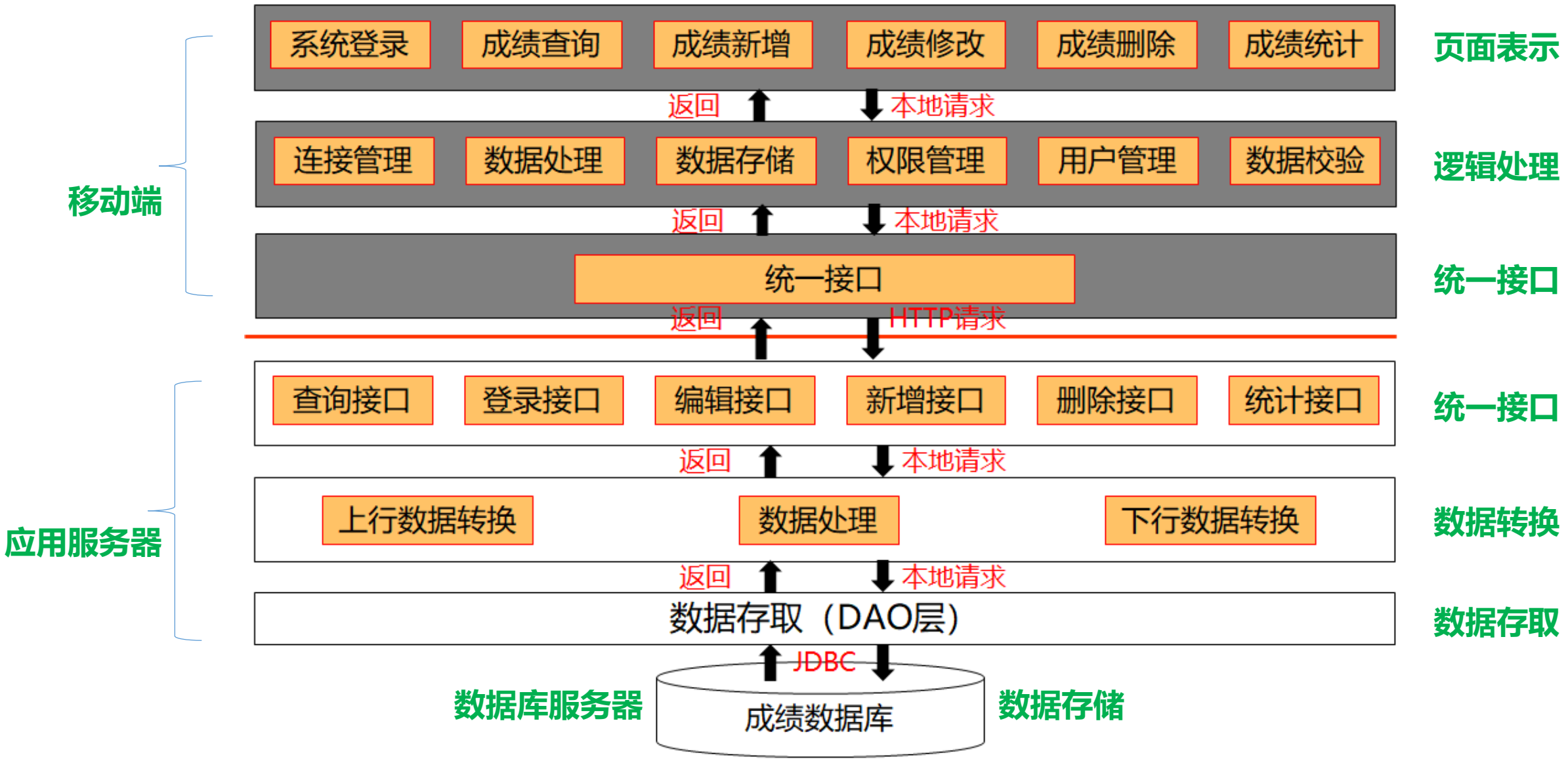
设计视图



实现视图



移动互联网应用项目中的设计实现



第一部分：*Android* APP存储实现

主要内容

- **简单存储**
- 文件存储
- 数据库存储
- 跨应用数据源

Shared Preferences

- *SharedPreferences*允许存储和获取由基本数据类型（*boolean*、*float*、*int*、 *long*和*string* ）组成的键值对(*Key-Value*对)
- 只要程序未被卸载，数据会一直存在
- 通过*getSharedPreferences*()或*getPreferences()* 可获得*SharedPreferences*入口

- ***SharedPreferences***支持多种数据访问模式
  - 私有模式***MODE_PRIVATE***：仅创建程序有权限进行读写
  - 全局读模式 ***MODE_WORLD_READABLE***：创建程序可进行读写，其他应用程序可读，但无写入权限
  - 全局写模式***MODE_WORLD_WRITEABLE***：创建程序和其他程序都可进行写入操作，但其它程序无读取权限
  - 全局读写模式***MODE_WORLD_READABLE*** + ***MODE_WORLD_WRITEABLE***：创建程序和其他程序都可进行读写操作

Shared Preferences使用

第一步：获得***SharedPreferences***对象，或指定存储*key-value*的文件名称

第二步：获得***SharedPreferences.Editor***对象

第三步：保存*key-value*对。接口一般支持不同数据类型，例如***putString***和***putBoolean***等

第四步：提交保存*key-value*对，相当于数据库提交（***commit***）操作

Shared Preferences存取复杂类型的数据

- 使用*SharedPreferences*保存复杂类型（对象、图像等）数据时需要①对数据进行编码，将复杂数据类型转换成Base64格式的编码，②然后将转换后的数据以字符串的形式保存到XML文件中

可以保存设置的Activity

- *SharedPreferences*可保存系统配置信息

  PreferenceActivity/ *PreferenceFragment*可更简洁的保存配置信息，封装了*Preferences*

- *PreferenceActivity* /*PreferenceFragment*提供了一些常用的控件，满足大多数配置界面要求，常用控件有
  - *CheckBoxPreference*：相当于*CheckBox*
  - *EditTextPreference*：弹出带*EditText*的对话框
  - *ListPreference*：弹出带*ListView*的对话框

可以保存设置的Activity

```
<PreferenceScreen xmlns:android="http://schemas.android.com/apk/res/android">
    <PreferenceCategory android:title="我的位置源">
        <CheckBoxPreference android:key="wireless_network"
            android:title="使用无线网络" android:summary="使用无线网络查看应用程序（例如Google地图）中的位置" />
        <CheckBoxPreference android:key="gps_satellite_setting"
            android:title="启用GPS卫星设置" android:summary="定位时，精确到街道级别（取消选择可节约电量）" />
    </PreferenceCategory>
    <PreferenceCategory android:title="个人信息设置">
        <CheckBoxPreference android:key="yesno_save_individual_info"
            android:title="是否保存个人信息" />
        <EditTextPreference android:key="individual_name"
            android:title="姓名" android:summary="请输入真实姓名" />
        <PreferenceScreen android:key="other_individual_msg"
            android:title="其他个人信息" android:summary="是否工作、手机">
            <CheckBoxPreference android:key="is_an_employee"
                android:title="是否工作" />
            <EditTextPreference android:key="mobile"
                android:title="手机" android:summary="请输入真实的手机号" />
        </PreferenceScreen>
    </PreferenceCategory>
</PreferenceScreen>
```

可以保存设置的Activity

- 每个设置界面对应一个<PreferencesScreen>，如果使用嵌套<PreferencesScreen>标签，说明该设置页有子设置页，单击可进入该设置页
- <PreferenceCategory>标签表示一个设置分类，android:title属性表示分类/标题，会显示在设置界面上, android:summary表示摘要信息
- 控件标签都有android:key属性,该属性的值保存在XML文件中的key-value对中的key
- 控件标签均可放在<PreferenceCategory>标签中，也可直接放在<PreferencesScreen>中，表示不属于任何设置分类

可以保存设置的Activity：PreferenceActivity

主要内容

- 简单存储
- 文件存储
- 数据库存储
- 跨应用数据源

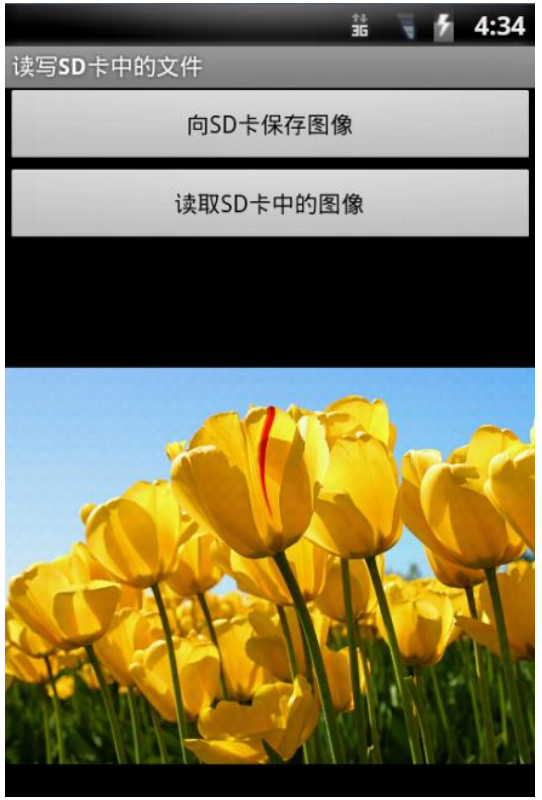
- 文件存储的核心是输入输出流

- 创建文件并保存数据的步骤：
  - 第一步：调用***openFileOutput()***方法，确定文件名和操作模式
  - 第二步：使用***write()***方法向文件写入数据
  - 第三步：调用***close()***关闭输出流

- 从文件读取数据的步骤：
  - 第一步：调用***openFileInput()***方法，确定读取的文件名
  - 第二步：调用***read()***方法读取字节
  - 第三步：调用***close()***关闭输入流

读写SD卡中的文件（例）

- 支持从终端的SD卡读写数据



通过SAX引擎读取XML文件

- ***SharedPreference*** 操作的文件为***XML***文件，但操作细节被隐藏了
- ***Android SDK*** 提供了操作***XML***文件的类库 ***（SAX：Simple API for XML）***
- ***SAX*** 技术处理***XML***文件采用事件驱动方式，边读取边解析
  - 不同于***DOM：Document Object Model***方式

SAX引擎读取XML时的主要分析事件

- 开始分析*XML*文件。表示*SAX*引擎开始处理*XML*文件，使用*startDocument()*方法，可在该方法加入初始化工作
- 开始处理XML标签。 *SAX*引擎每次扫描到新的*XML*标签起始标记时触发该事件，使用*startElement()*方法，在该方法中可以获得当前标签的名称和属性
- 处理结束XML标签。使用*endElement()*方法，在该方法中可以获得当前处理完的标签的所有信息
- 处理完XML文件。使用*endDocument()*方法，该方法不是必须的，资源释放工作可在该方法中完成
- 读取字符事件。使用*characters()*方法，其主要的作用是处理*SAX*引擎读取的*XML*文件内容，保存标签内容

将XML文件转换为Java对象

```
<products>
  <product>
    <id>10 </id>
    <name> 电脑 </name>
     <price>2067.25 </price>
  </product>
  <product>
    <id> 20 </id>
     <name> 微波炉 </name>
     <price> 520 </price>
  </product>
  <product>
    <id> 30 </id>
     <name>洗衣机 </name>
    <price> 2400 </price>
  </product>
</products>
```

文件压缩（Jar/Zip）

- ***Android SDK*** 提供压缩成***jar***和***zip***包的功能，基本步骤为：

第一步：创建***(Jar|Zip)OutputStream***对象，用于生成***jar***或***zip***文件

第二步：创建***(Jar|Zip)Entry***对象，代表一个被压缩的文件，并指定被压缩文件在压缩包中的文件名和路径

第三步：调用***(Jar|Zip)OutputStream.putNextEntry***设置当前打开的***(Jar|Zip)Entry***对象

第四步：向***(Jar|Zip)OutputStream***对象写入数据

第五步：调用***(Jar|Zip)OutputStream.closeEntry***关闭当前打开的***Entry***对象

第六步：如果还有待压缩的文件，回到第二步继续执行

文件解压缩（Jar/Zip）

- 文件解压缩的基本步骤为：

第一步：创建***(Jar|Zip)IutputStream***对象

第二步：使用***(Jar|Zip)IutputStream.getNextEntry***方法枚举压缩包中的文件，如果返回*null*，表示所有的压缩文件均被处理完

第三步：通过***(Jar|Zip)Entry.getName***方法获得文件压缩后的路径和文件名，使用***FileOutputStream***对象指定已解压缩的文件

第四步：向***FileOutputStream***对象输出已解压的数据流

第五步：调用***(Jar|Zip)Entry.closeEntry***关闭当前打开的***Entry***对象

第六步：如果压缩包中还有待解压的文件，回到第二步继续执行

主要内容

- 简单存储
- 文件存储
- 数据库存储
- 跨应用数据源

- *SQLite*是一款开源嵌入式轻量级数据库软件，适合于移动平台使用
- 全部移动平台均使用*SQLite*创建、存取本地数据库

SQLite支持SQL的主要类型

支持的SQL	说明	支持的SQL	说明
ATTACH DATABASE	将已存在库添加到当前库连接	*DETACH DATABASE*	拆分已存在库
COMMENT	注释		
CREATE TABLE	创建表	*DROP TABLE*	删除表
CREATE VIEW	创建视图	*DROP VIEW*	删除视图
CREATE TRIGGER	创建触发器	*DROP TRIGGER*	删除触发器
CREATE INDEX	创建索引	*DROP INDEX*	删除索引
BEGIN TRANSACTION	开启事务	*DROP TRANSACTION*	提交事务
END TRANSACTION	结束事务	*ROLLBACK TRANSACTION*	回滚事务
SELECT	查询	*INSERT*	插入
UPDATE	更新	*DELETE*	删除

SQLiteOpenHelper

- *android.database.sqlite.SQLiteDatabase*是*android SDK*数据库核心类，可打开数据库，可对数据进行操作，为了数据库升级需要和使用更加便捷，通常使用其子类*SQLiteOpenHelper*来完成创建、打开数据库及各种操作
- *SQLiteOpenHelper*可自动检测数据库文件是否存在，如文件存在，则打开数据库，如文件不存在，则先创建数据库，然后打开数据库
- *onCreate()*方法用于新创建的数据库中建立表、视图等数据库组件，*onCreate()*方法在数据库第一次被创建时调用
- 如果数据库文件存在，会调用*onUpgrade()*方法升级数据库，并更新版本号

SQLite数据库操作

- 向库表添加数据

```
insert("tableName", null, ContentValues values)
例：db.insert(“student”,null,contentValues);
```

- 从库表删除数据

```
delete( "tableName",  String arg2, String arg3 )
例：db.delete(“student”, “result>?” , new String[]{“80”});
```

- 修改库表数据

```
update("tableName", ContentValues values, String arg3, String arg4)
例：db.update(“student”, contentValue1, “name=?” , new String[]{“张三”});
```

- 查询数据

```
query("tableName", null, null, null, null, null, null)
例：db.query(“student”, null, null, null, null, null, null);
```

具体实现时，通常将全部数据库操作封装在公共类中！

数据绑定与SimpleCursorAdapter类

- 如需将表中的数据显示在*ListView*、*Gallery*等控件中，可使用*SimpleCursorAdapter*
- *SimpleCursorAdapter*与*SimpleAdapter*的使用方法非常接近，只是将数据源由*List*对象换成了*Cursor*对象

还支持其它数据库存取

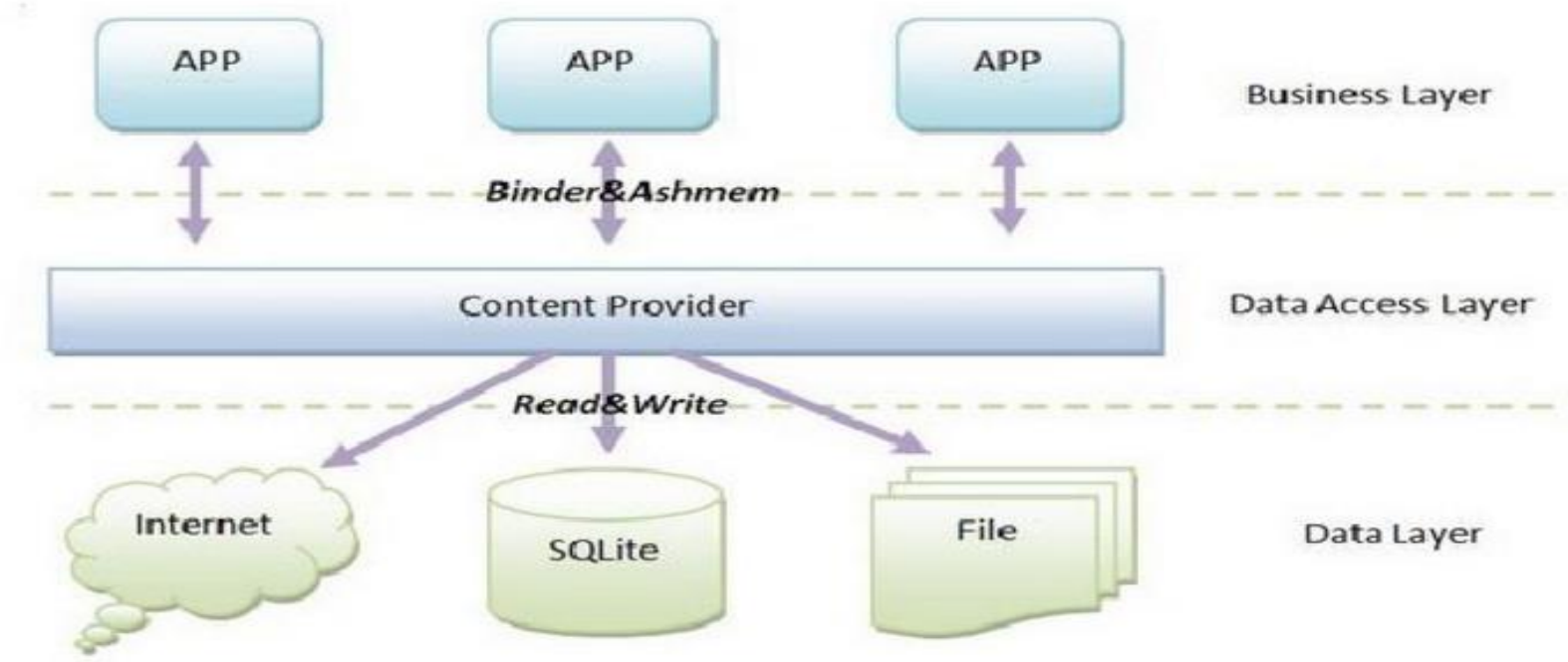
- 支持操作SD卡上的数据库
- 可以将数据库与应用程序一起发布
- 可以创建并操作内存数据库

主要内容

- 简单存储
- 文件存储
- 数据库存储
- 跨应用数据源

什么是Content Provider

- *Content Provider* 提供了应用程序之间共享数据的方法
- 应用程序通过*Content Provider*访问数据而不需要关心数据具体的存储及访问过程，既提高了数据的访问效率，也保护了数据



什么是Content Provider

- *Content Provider*提供了对外部数据进行增、删、改、查的功能，相当于跨应用的数据操作，也可以看做是操作数据库的代理

- *Content Provider*相当于数据库接口，通过它可以将程序内部使用的数据向其它程序公开，其它的程序可以通过*Context.getContentResolver*方法获得*ContentResolver*对象，并使用*insert*、*delete*、*update*、*query*等方法对程序内部的数据进行增删改查操作

Content Provider的作用

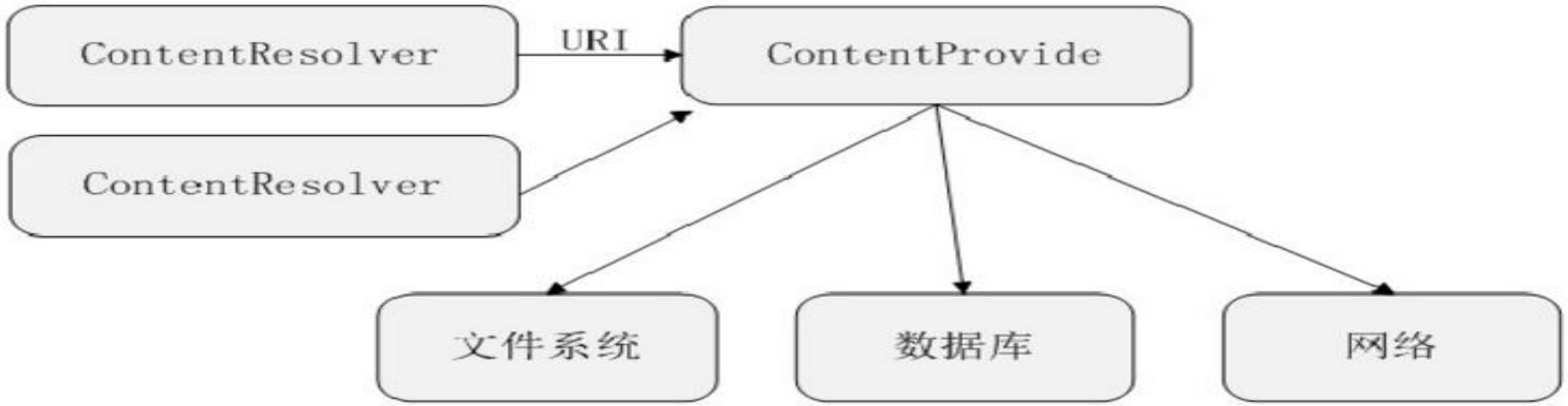
- 程序内部的数据文件通常保存在私有目录中，其它的程序使用常规的方法无法访问这些数据库，通过***Content Provider***可以使外部程序访问私有数据库，类似于***java***类中的***private***变量和***setter***、***getter***方法
- 由于程序内部使用的数据库结构可能很复杂，例如，数据通过若干个表和视图连接而获得，对于不了解数据结构的开发人员，即使可以访问私有数据库，也无从下手，通过***Content Provider***可以向开发人员提供一个更为人性化的查询结果集
- 处于安全考虑，有些数据需要设置权限才能使用***Content Provider***正常操作，这种访问控制可以通过***Content Provider***机制来完成，作为应用程序和操作系统之间的一层，可以有效的控制应用程序的访问权限

系统附带的Content Provider

- ***Browser***：存储浏览器的信息
- ***CallLog***：存储通话记录信息
- ***Contacts***：存储联系人信息
- ***MediaStore***：存储媒体文件信息
- ***Setting***：存储设备的设置和首选项信息
- ***SMS***：存储短消息
- ***……***

ContentResolver对象

- 应用程序使用*ContentResolver*对象（含*URI*）才能访问*ContentProvider*提供数据集
- *ContentResolver*对象提供*query*、*insert*、*update*、*delete*等方法



一个ContentProvider可以提供多个数据集
可以为多个ContentResolver服务

Content Provider的数据集

- *ContentProvider*数据集类似于数据库的数据表，每行是1条记录，每列具有相同的数据类型

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

URI的基本格式

- *URI*用来定位远程或者本地的可用资源

- content://<authority>/<data_path>/<id>

  - content:// 固定前缀
  - <authority> 授权者名称(保证唯一性) 用来确定具体由哪一个ContentProvider提供资源
  - <data_path> 数据路径 用来确定请求的是哪个数据集
  - <id> 数据编号，用来匹配数据集中_ID字段的值（用来唯一确定数据集中的一条记录）如果请求的数据并不只限于一条数据，则<id>可以省略

- content://com.android.contacts/contacts/ 原生写法
- ContactsContract.Contacts.CONTENT_URI 常量写法

全部联系人信息的URI

由于URI通常比较长，而且容易写错，所以，在Android系统中定义了一些辅助类和常量来代替这些长字符串。

获得系统数据

- 通过*Content Provider*查询系统数据需要调用*ContentResolver*对象的*query*方法，返回的*cursor*对象与查询*SQLite*数据库返回的*Cursor*对象完全一样

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

  - *Uri*表示*Content Provider Uri*，如*uri.parse("content://sms/inbox")*;
  - *Projection*表示查询返回内容，相当于SQL中***select***和***from***之间部分
  - *Selection*相当于SQL中***where***子句，表示查询条件
  - *selection Args*，如果***selection***参数的值包含了参数，也就是**(?)**，*selectionArgs*则表示这些参数值，如果***selection***参数的值不包含任何参数，则***selectionArgs***为*null*
  - *sortOrder*相当于SQL中***orderby***子句

可获得联系人数据

- ***CONTENT_URI***是一个常量的组合，包括
  - ***AUTHORITY*** = *“com.android.contacts”*
  - ***AUTHORITY_URI*** = *Uri.parse(“content://”* + ***AUTHORITY);***
  - ***CONTENT_URI*** = *Uri.withAppendedPath(**AUTHORITY_URI**,”contacts”);*

- 如果不使用这些常量，可以使用
  - ***uri*** = *Uri.parse(“**content://com.android.contacts/contacts**”);*
- 查询联系人信息需要在*AndroidManifest.xml*文件中打开权限
  - ***<uses-permission android:name="android.permission.READ_CONTACTS" />***

自定义Content Provider

- 自己开发的程序也可通过*Content Provider*向外共享数据，由于*Content Provider*是一个抽象类，因此所有的抽象方法都必须实现，编写完*Content Provider*类后，必须在*AndroidManifest.xml*文件中注册
- 实现*Content Provider*的步骤
  - 继承*ContentProvider*类
  - 实现所有的抽象方法
  - 定义*Content Provider*的*URI*，*URI*分为*authority*和*path*两个部分，前者相当于域名，后者表示具体地址
  - 使用*UriMather*对象映射*Uri*和返回代码，相当于定义各种快捷查询*URI*
  - 根据实际需要实现相应的方法
  - 注册*Content Provider*

注册自定义Content Provider

```
<permission
    android:name="mobile.android.ch11.permission.regioncontentprovider.READ_REGION"
    android:protectionLevel="normal" android:label="region"
    android:description="read_region" />

<application android:icon="@drawable/icon" android:label="查询城市、省信息（带权限）">

    <provider android:name="RegionContentProvider"
        android:authorities="mobile.android.ch11.permission.regioncontentprovider"
        android:readPermission="mobile.android.ch11.permission.regioncontentprovider.READ_REGION" />

    <activity android:name=".Main" android:label="查询城市、省信息（带权限）">
        <intent-filter>
            <action android:name="android.intent.action.MAIN" />
            <category android:name="android.intent.category.LAUNCHER" />
        </intent-filter>
    </activity>
```

定义权限

设置权限

Content Provider的权限设置和使用

第一步：定义权限，需要使用*permission*标签定义

```
<permission
    android:name="mobile.android.ch11.permission.regioncontentprovider.READ_REGION"
    android:protectionLevel="normal" android:label="region"
    android:description="read_region" />
```

- 第二步：设置权限，*<provider>*标签的*android:readPermission*属性设置*query*方法，如需写权限，可设置*android:writePermission*，可同时设置读写权限

```
<provider android:name="RegionContentProvider"
    android:authorities="mobile.android.ch11.permission.regioncontentprovider"
    android:readPermission="mobile.android.ch11.permission.regioncontentprovider.READ_REGION" />
```

- 第三步：使用权限，数据应用侧须通过*<uses-permission>*设置

```
<uses-permission android:name="mobile.android.ch11.permission.regioncontentprovider.READ_REGION" />
```

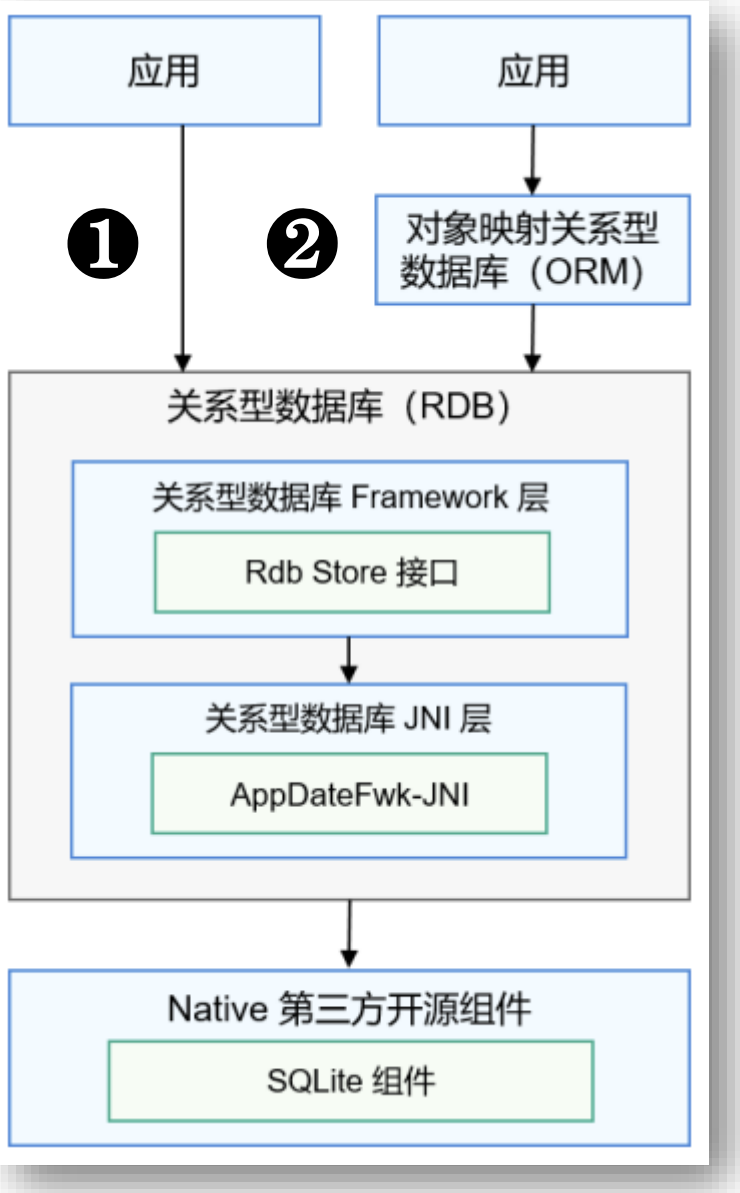
第二部分：*HarmonyOS* 平台存储实现

Harmony OS 关系数据库及运行机制

- *HarmonyOS* 关系型数据库基于 *SQLite* 组件提供了一套完整的对本地数据库进行管理的机制，对外提供增、删、改、查接口，也可以直接运行用户输入的 *SQL* 语句来满足复杂场景需要

- *HarmonyOS* 关系型数据库对外提供通用操作接口，底层使用 *SQLite* 作为持久化存储引擎，支持 *SQLite* 具有的所有数据库特性，包括但不限于事务、索引、视图、触发器、外键、 参数化查询和预编译 *SQL* 语句

- *HarmonyOS* 关系型数据库在 *SQLite* 基础上实现的本地数据操作机制，提供给用户无需编写原生 *SQL* 语句就能进行数据增删改查的方法，同时也支持原生 *SQL* 操作



Harmony OS 关系数据库接口

类名	接口名	描述
StoreConfig.Builder	*public builder()*	配置数据库，包括设置数据库名、存储模式、日志模式、同步模式、是否为只读及对数据库加密
RdbOpenCallback	*public abstract void onCreate (RdbStore store)*	库创建时回调，可以在该方法中初始化表结构，并添加初始化数据
RdbOpenCallback	*public abstract void onUpgrade(RdbStore store, int currentVersion, int targetVersion)*	数据库升级时被回调
DatabaseHelper	*public RdbStore getRdbStore(StoreConfig config, int version, RdbOpenCallback openCallback, ResultSetHook resultSetHook)*	根据配置创建或打开数据库
DatabaseHelper	*public boolean deleteRdbStore(String name)*	删除指定的数据库

Harmony OS 关系数据库接口

类名	接口名	描述
StoreConfig.Builder	*Builder setEncryptKey(byte[] encryptKey)*	设置数据库加密密钥，创建或打开数据库时传入含加密密钥的配置类，即可创建或打开加密数据库
RdbStore	*long insert (String table, ValuesBucket initialValues)*	插入数据，*initialValues* 是以 *ValuesBucket* 存储的待插入数据
RdbStore	*int update(ValuesBucket values, AbsRdbPredicates predicates)*	更新数据。*values*存储要更新的数据。*predicates*指定更新操作的表名和条件

Harmony OS 关系数据库接口

类名	接口名	描述
RdbStore	*int **delete**(AbsRdbPredicates predicates）*	删除数据，*predicates*指定删除操作的表名和条件
RdbStore	*ResultSet **query** (AbsRdbPredicates predicates, String[] columns)*	查询数据。predicates设置查询条件
RdbStore	*ResultSet **querySql** (String sql, String[] sqlArgs)*	执行原生SQL语句。*sqlArgs*表示*sql*语句中占位符参数，若未使用占位符，可设置为null

Harmony OS 关系数据库查询结果集的使用

类名	接口名	描述
ResultSet	*boolean goTo(int offset)*	从结果集当前位置移动指定偏移量
ResultSet	*boolean goToRow(int position)*	将结果集移动到指定位置
ResultSet	*boolean goToNextRow()*	将结果集向后移动一行
ResultSet	*boolean goToPreviousRow()*	将结果集向前移动一行
ResultSet	*boolean isStarted()*	判断结果集是否被移动过
ResultSet	*boolean isEnded()*	判断当前位置是否在最后一行之后
ResultSet	*boolean isAtFirstRow()*	判断结果集当前位置是否在第一行
ResultSet	*boolean isAtLastRow()*	判断结果集当前位置是否在最后一行
ResultSet	*int getRowCount()*	获取当前结果集中的记录条数
ResultSet	*int getColumnCount()*	获取结果集中的列数
ResultSet	*String getString(int columnIndex)*	获取当前行指定索列的值，以 *String* 类型返回
ResultSet	*byte[] getBlob(int columnIndex)*	获取当前行指定列的值，以字节数组形式返回
ResultSet	*double getDouble(int columnIndex)*	获取当前行指定列的值，以 *double* 型返回

1.创建数据库

a. 配置数据库相关信息，包括数据库的名称、存储模式、是否为只读模式等

b. 初始化数据库表结构和相关数据

c. 创建数据库

```
public void onCreate(RdbStore store) {
    store.executeSql("CREATE TABLE IF NOT EXISTS test (id INTEGER PRIMARY KEY AUTOINCREMENT,
                name TEXT NOT NULL, age INTEGER, salary REAL, blobType BLOB)");
}
```

2.插入数据

a.构造要插入的数据，以 *ValuesBucket* 形式存储

b.调用关系型数据库提供的插入接口

```
1. ValuesBucket values = new ValuesBucket();
2. values.putInteger("id", 1);
3. values.putString("n("name", "zhangsan");
4. values.putInteger("age", 18);
5. values.putDouble("salary", 100.5);
6. values.putByteArray("blobType", new byte[] {1, 2, 3});
7. long id = store.insert("test", values);
```

3. 查询数据

a.构造用于查询的谓词对象，设置查询条件

b.指定查询返回的数据列

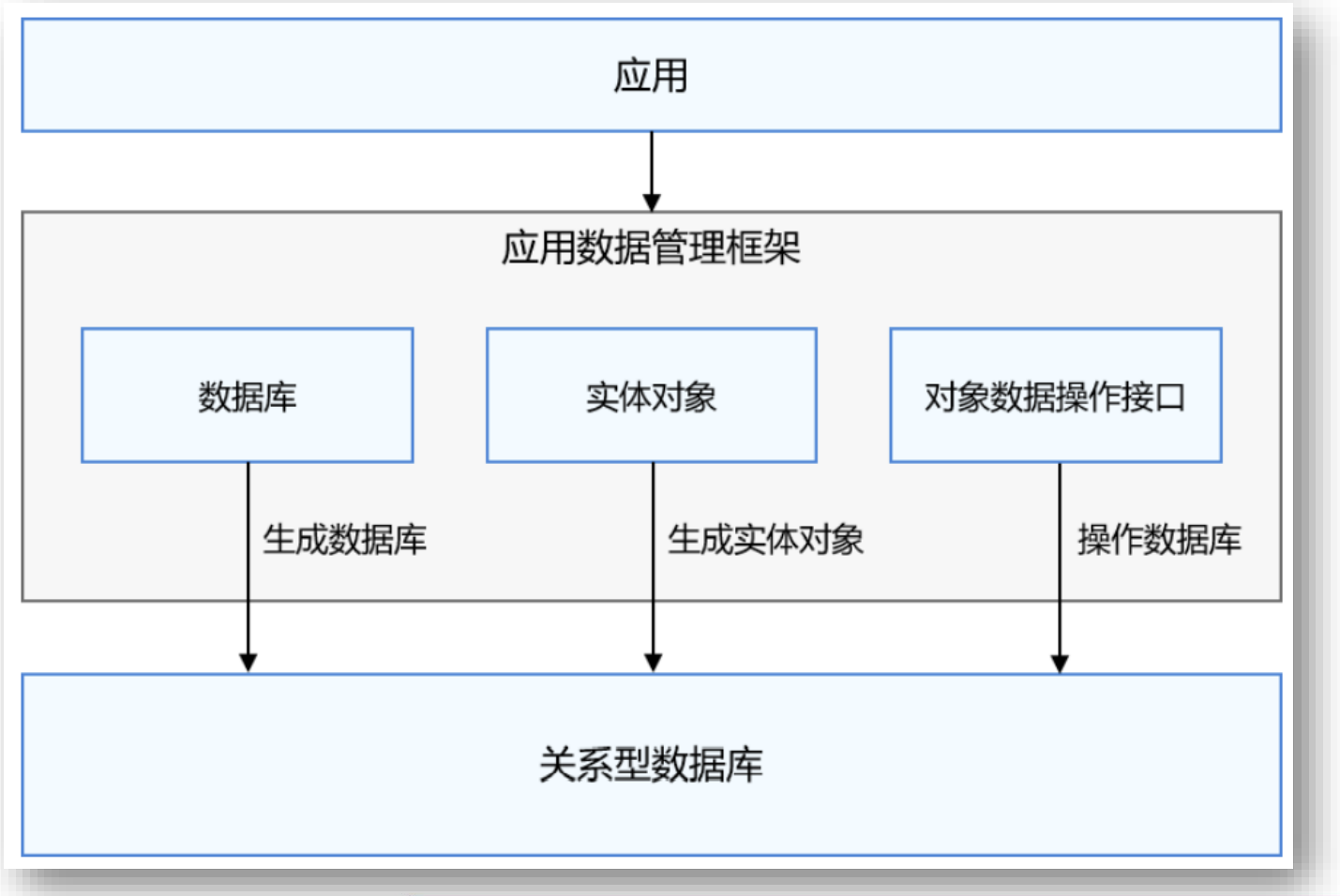
c.调用查询接口查询数据。

d.调用结果集接口，遍历返回结果。

```
1. String[] columns = new String[] {"id", "name", "age", "salary"};
2. RdbPredicates rdbPredicates = new RdbPredicates("test").equalTo("age", 25).orderByAsc("salary");
3. ResultSet resultSet = store.query(rdbPredicates, columns);
4. resultSet.goToNextRow();
```

Harmony OS 对象映射关系型数据库

- ***HarmonyOS***的***ORM***数据库是一款基于 ***SQLite*** 的数据库框架，屏蔽了底层 ***SQLite*** 数据库的 ***SQL*** 操作，针对实体和关系提供了增删改查等一系列的面向对象接口。应用开发者不必编写***SQL*** 语句，以操作对象的形式操作数据库



Harmony OS 轻量级偏好数据库

- ***HarmonyOS***轻量级偏好数据库主要提供轻量级***Key-Value***操作，支持本地应用存储少量数据，数据存储在本地文件中，同时也加载在内存中的，访问速度更快，效率更高。轻量级偏好数据库属于非关系型数据库，不宜存储大量数据，用于操作“键值对”数据

1. Key-Value 数据库：

（1）一种以键值对存储数据的数据库，类似 *Java* 中的 *map*，*Key*是关键字，*Value*是值

（2）*Key*键为*String*类型，*Value*可为任意数据类型

（3）存储数据量是轻量级的，建议存储数据不超过1万条，否则会产生较大内存开销

2. 非关系型数据库：

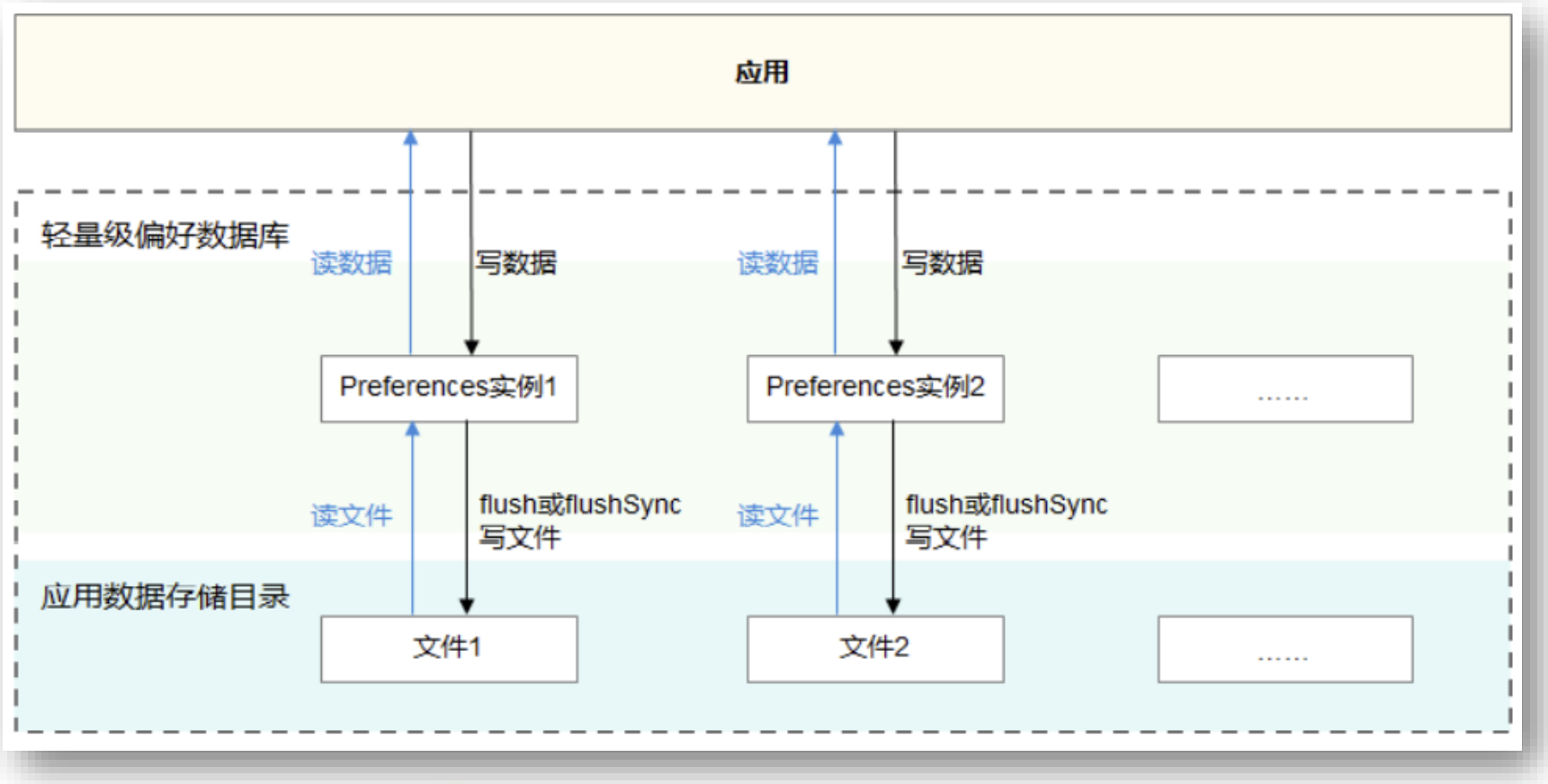
区别于关系数据库，不保证遵循 *ACID*（原子性*Atomic*、一致性*Consistency*、独立性*Isolation* 及 持久性*Durability*）特性，不采用关系模型来组织数据，数据之间无关系，扩展性好

3. 偏好数据：

用户经常访问和使用的数据

Harmony OS 轻量级偏好数据库运行机制

- 借助*DatabaseHelper API*，可将指定文件内容加载到*Preferences*实例，系统会通过静态容器将该实例存储在内存中，直到应用主动从内存中移除该实例或者删除该文件
- 获取*Preferences*实例后，可借助*Preferences API*读取数据或者将数据写入*Preferences*实例



Harmony OS 轻量级偏好数据库开发接口

- 轻量级偏好数据库是轻量级存储，主要用于保存应用的一些常用配置，并不适合存储大量数据和频繁改变数据的场景。用户数据保存在文件中，可以持久化的存储在设备上。需要注意的是用户访问的实例包含文件所有数据，并一直加载在设备的内存中，并通过轻量级偏好数据库的***API***完成数据操作

- 轻量级偏好数据库向本地应用提供了操作偏好型数据库的***API***，支持本地应用读写少量数据及观察数据变化。数据存储形式为键值对，键的类型为字符串型，值的存储数据类型包括整型、字符串型、布尔型、浮点型、长整型、字符串型***Set*** 集合

Harmony OS 轻量级偏好数据库开发接口

类名	接口名	描述
DatabaseHelper	*Preferences getPreferences(String name)*	获取文件对应的*Preferences*单实例
Preferences	*int getInt(String key, int defValue)*	获取键对应的 int 类型值
Preferences	*float getFloat(String key, float defValue)*	获取键对应的 float 类型值
Preferences	*Preferences putInt(String key, int value)*	设置键对应的 int 类型值
Preferences	*Preferences putString(String key, String value)*	设置键对应的 String 类型值
Preferences	*void flush()*	将Preferences实例异步写入文件
Preferences	*boolean flushSync()*	将Preferences实例同步写入文件
Preferences	*void registerObserver(PreferencesObserver preferencesObserver)*	注册观察者
Preferences	*void unRegisterObserver(PreferencesObserver preferencesObserver)*	注销观察者
Preferences.PreferencesObserver	*void onChange(Preferences preferences, String key)*	任意数据变化都回调该方法
DatabaseHelper	*boolean deletePreferences(String name)*	删除文件和对应的Preferences实例
DatabaseHelper	*void removePreferencesFromCache(String name)*	删除Preferences 单实例
DatabaseHelper	*boolean movePreferences(Context sourceContext, String sourceName, String targetName)*	移动数据库文件

第一步：获取*Preferences*实例，读取指定文件，将数据加载到*Preferences*实例，用于数据操作

第二步：对指定文件读取/写入数据，支持同步和异步方式

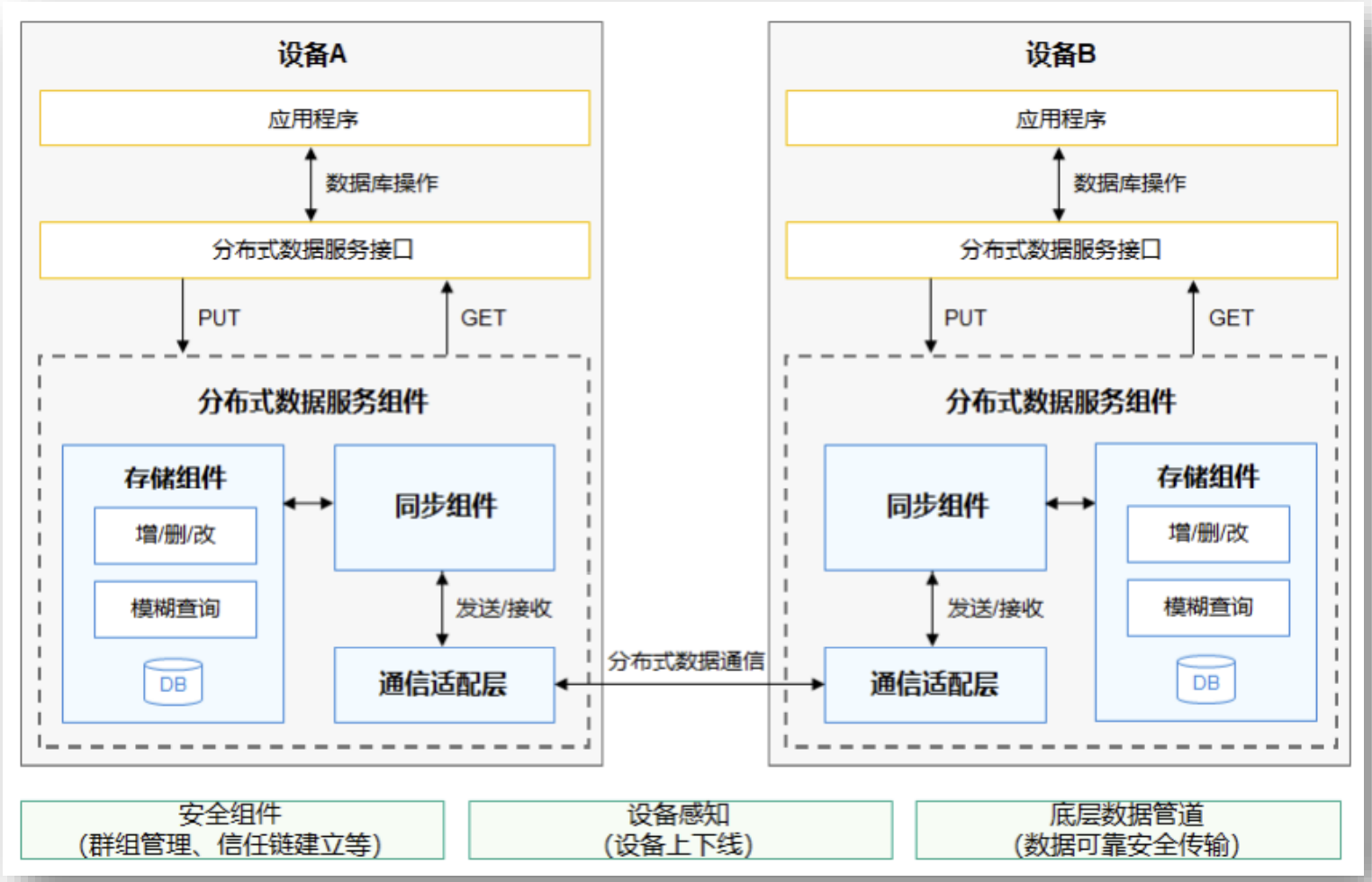
第三步：注册观察者，向*Preferences*实例注册观察者，实例注册的所有观察者的*onChange()*方法都会被回调，不再需要观察者时可注销

第四步：移除*Preferences*实例，移除 *Preferences*实例时，不允许再使用该实例进行数据操作，否则会出现数据一致性问题

第五步：删除指定文件，从内存中移除指定文件对应的*Preferences*单实例，并删除指定文件及其备份文件、损坏文件

Harmony OS 分布式数据库

- 分布式数据服务（***DDS***）为应用程序提供不同设备间数据库数据分布式存储和存取能力
- 通过(**帐号、应用和数据库**)三元组，分布式数据服务对属于不同应用的数据进行隔离，保证不同应用之间的数据不能通过分布式数据服务互相访问
- 在互信认证的设备之间，分布式数据服务支持应用数据同步，为用户提供在多种终端设备上一致的数据访问体验



Harmony OS 分布式数据库服务

（1）服务接口：分布式数据服务提供专用数据库创建、数据访问、数据订阅等接口，支持KV数据模型，支持常用的数据类型，同时确保接口的兼容性、易用性和可发布性

（2）服务组件：服务组件负责服务内元数据管理、权限管理、加密管理、备份和恢复管理以及多用户管理等，同时负责初始化底层存储组件、同步组件和通信适配层

（3）存储组件：存储组件负责数据访问、数据压缩、事务、快照、数据库加密，以及数据合并和冲突解决等特性

（4）同步组件：连结存储组件与通信组件，目标是保持在线设备间数据一致性，包括将本地产生的未同步数据同步给其他设备；接收其他设备数据并合并到本地设备中

（5）通信适配层：负责调用底层公共通信层的接口完成通信管道的创建、连接，接收设备上下线消息，维护已连接和断开设备列表的元数据，同时将设备上下线信息发送给上层同步组件

Harmony OS 分布式文件服务

- 分布式文件服务能为用户设备中的应用程序提供多设备之间的文件共享能力，支持相同帐号下同一应用文件的跨设备访问，应用程序可以不感知文件所在的存储设备，能够在多个设备之间无缝获取文件

- 分布式文件服务采用无中心节点的设计，每个设备都存储一份全量的文件元数据和本设备上产生的分布式文件，元数据在多台设备间互相同步。当应用需要访问分布式文件时，分布式文件服务首先查询本设备上的文件元数据，获取文件所在的存储设备，然后对存储设备上的分布式文件服务发起文件访问请求，将文件内容读取到本地

Harmony OS 分布式文件的约束与限制

- 使用分布式文件服务完整功能，需申请 ***ohos.permission.DISTRIBUTED_DATASYNC*** 权限
- 多个设备需要打开蓝牙，连接同一WLAN局域网，登录相同华为帐号才能实现文件的分布式共享
- 存在多设备并发写的场景下，为了保证文件独享，需要对文件进行加锁保护
- 应用访问分布式文件时，如果文件所在设备离线，文件不能访问
- 非持锁情况下，并发写冲突时，后一次会覆盖前一次
- 网络情况差时，访问存储在远端的分布式文件时，可能会长时间不返回或返回失败，应用需要考虑这种场景的处理
- 当两台设备有同名文件时，同步元数据时会产生冲突，分布式文件服务根据时间戳将文件按创建的先后顺序重命名，为避免此场景，建议应用在文件名上做设备区分，例如:***deviceID***+时间戳

Harmony OS 数据存储管理

- 数据存储管理可管理存储设备（包含本地存储、SD 卡、U 盘等），包括获取存储设备列表，获取存储设备视图等

- 用统一的视图结构可以表示各种存储设备，该视图结构的内部属性会因为设备的不同而不同。 每个存储设备可以抽象成两部分，一部分是存储设备自身信息区域，一部分是用来存放数据的区域

Harmony OS 数据存储管理-开发场景和接口

- 可使用数据存储管理接口获取存储设备信息，也可根据文件名获取对应存储设备信息

功能分类	类名	接口名	描述
查询设备视图	*ohos.data.usage.DataUsage*	*getVolumes()*	获取当前用户可用的设备列表视图
		getVolume(File file)	获取存储该文件的存储设备视图
		getVolume(Context context, Uri uri)	获取该URI对应文件所在的设备视图
		getDiskMountedStatus()	获取默认存储设备的挂载状态
		getDiskMountedStatus(File path)	获取存储该文件设备的挂载状态
		isDiskPluggable()	默认存储设备是否为可插拔设备
		……	……
查询设备视图属性	*ohos.data.usage.Volume*	*isEmulated()*	该设备是否是虚拟存储设备
		isPluggable()	该设备是否支持插拔
		getDescription()	获取设备描述信息
		getState()	获取设备挂载状态
		getVolUuid()	获取设备唯一标识符

The End