中国民航大学

软件工程(I)

小米便签开源代码精读报告

组长学号姓名：220340147 童文科

成员学号姓名：220340175 衡文昊

220340173 张珈衔

220340180 王喆宇

220340174 何声炬

二〇二四年四月

1. 精读小米便签的功能及涉及的类

精读功能名称：**新建便签、删除便签、便签的移动**
精读功能描述：**可以在当前目录创建一个便签并打开进入文本编辑**
可以当前目录删除一个或多个便签
允许用户将便签移动到不同的文件夹中，便于管理

表 1 1.　精读功能所涉及的类

包	子包	类	主要作用
data		Contact	联系人管理类，用于获取手机中联系人的信息。
		Notes	用于存放小米便签中需要的变量与常量及其数据类型，是小米便签最底层的数据类。
		NotesDatabaseHelper	此类是一个数据库帮助类，用于管理小米便签的数据库，包括：创建数据库和数据库版本更新。其继承自 Android 所提供的一个工具类 SOLiteOpenHelper 类。
		NotesProvider	内容提供类，主要为其他应用程序或组件提供对“Notes”类的数据的访问。它允许其他应用程序查询、插入、更新和删除数据。
gtask	data	MetaData	用于同步任务的元数据。
		Node	同步认为的节点管理，用于设置、保存同步任务的信息。
		SqlData	数据库中的数据类，功能包括数据读取、数据获取、数据提交
		SqlNode	数据库中的便签数据，功能包括便签内容读取、从数据库中获取便签、设置便签内容、提交便签到数据库
		Task	同步任务，将创建、更新和同步动作包装为 JSON 对象，用 JSON 对节点内容进行设置、获取同步信息进行设置，获取同步信息进行本地与远程进行同步
		TaskList	管理 task 的同步信息，将其编为列表形式。

2. 精读功能的模型

2.1 用例描述

（1）用例 1：

用例名称:新建便签

功能:可以在当前目录创建一个便签并打开进入文本编辑

执行者:用户

触发条件:点击小米便签软件的主界面下方的“写便签”快捷键

前置条件:进入小米便签软件的主界面

基本交互动作过程:

1.用户点击“写便签”快捷键可以在当前目录创建一个便签并打开进入文本编辑

扩展交互动作过程:在点击“写便签”快捷键进入便签中时，可以长按左滑退出编辑

后置条件:新的便签被创建

业务规则:无

非功能需求:无

（2）用例 2：

用例名称:删除便签

功能:可以在当前目录删除一个或者多个便签

执行者:用户

触发条件:长按便签并选择删除选项

前置条件:进入小米便签软件的主界面

基本交互动作过程:

1.用户长按便签并选择删除选项可以删除已创建的便签

扩展交互动作过程:在进行删除时，可以长按右滑取消删除

后置条件:便签被删除

业务规则:无

非功能需求:无

（3）用例 3：

用例名称: 移动便签

功能：允许用户将便签移动到不同的文件夹中，便于管理。

执行者: 用户

触发条件: 用户想要重新整理便签位置。

前置条件: 用户已登录并创建了至少一个便签和文件夹。

基本交互动作过程:

1. 用户选择想要移动的便签。
2. 用户选择“移动”操作。
3. 用户选择一个目标文件夹。
4. 系统将便签移动到目标文件夹。

后置条件：便签成功移动，界面更新显示新位置。

业务规则：

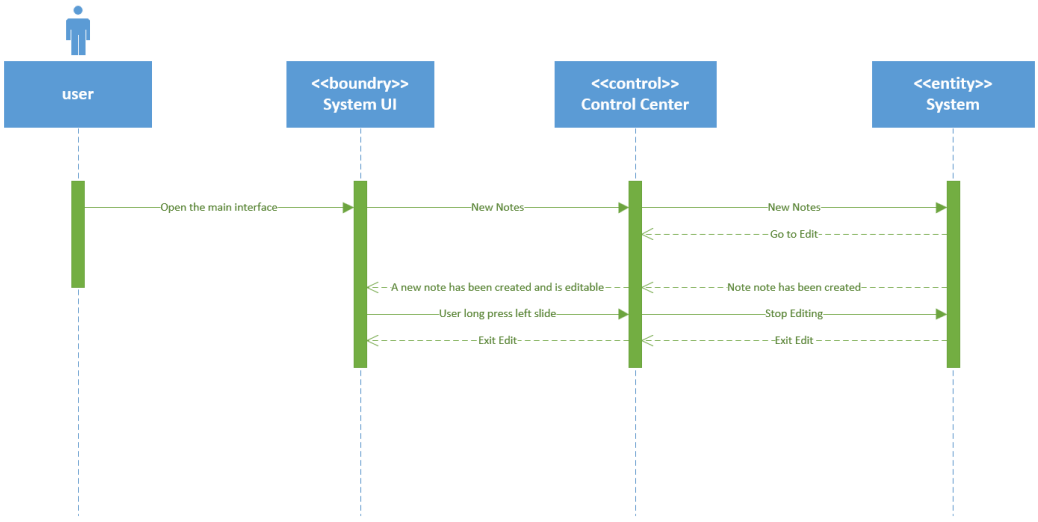
1. 一个便签只能存在于一个文件夹中。
2. 用户只能移动自己创建的便签。

非功能需求：

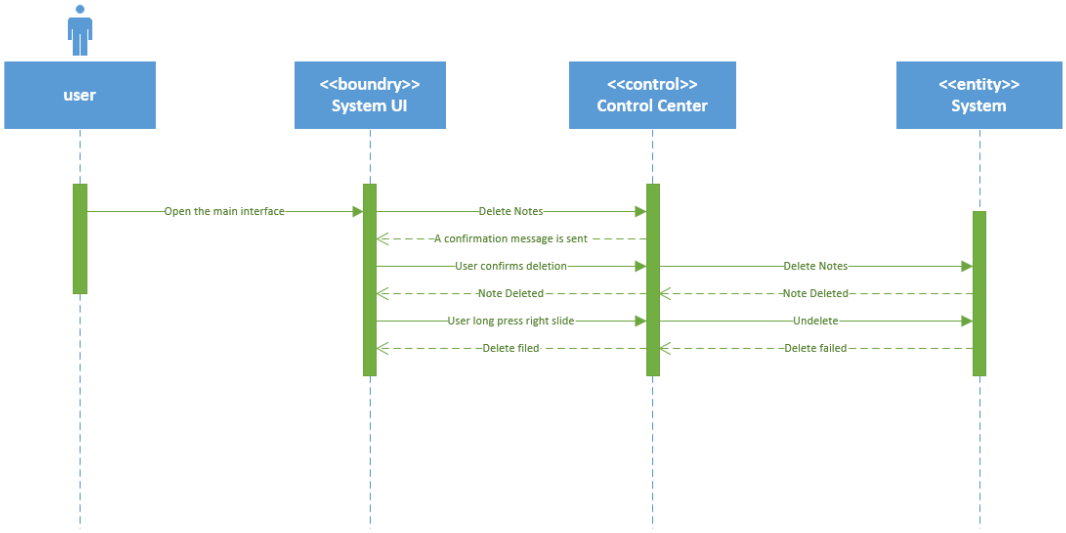
1. 操作响应迅速。
2. 数据完整性和一致性得到保证。

2.2 设计顺序图

（1）

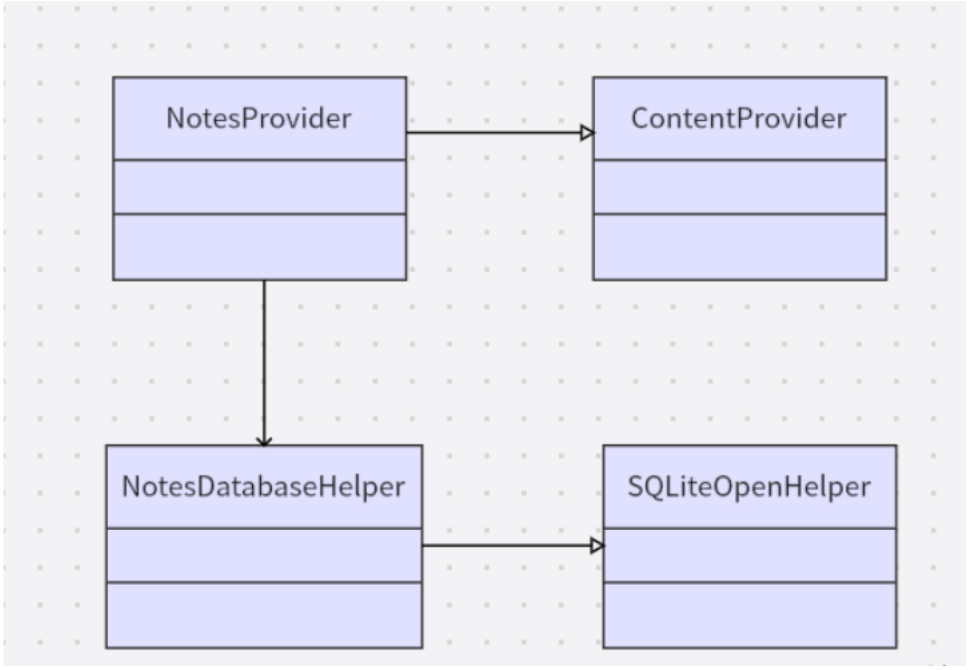


（2）

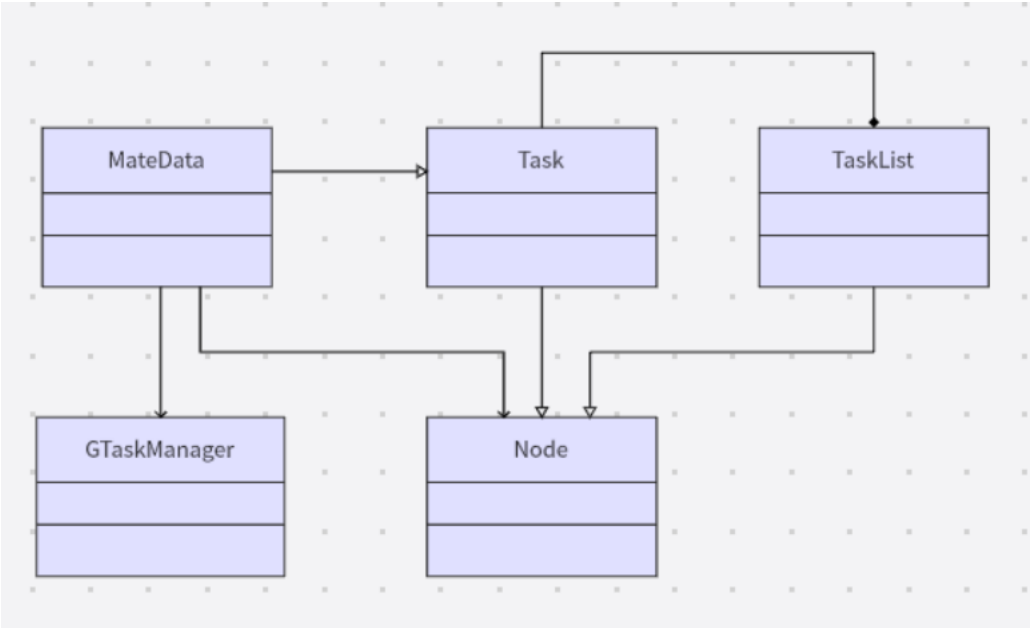


2.3 设计类图

（1）Data 类：



（2）Gtask 类：



3. 精读功能的代码标注

（1） 类名：Contact

```
package net.micode.notes.data;

import android.content.Context;
import android.database.Cursor;
import android.provider.ContactsContract.CommonDataKinds.Phone;
import android.provider.ContactsContract.Data;
import android.telephony.PhoneNumberUtils;
import android.util.Log;

import java.util.HashMap;

public class Contact {
//      用于处理联系人信息
//      实现了从联系人数据库中获取指定电话号码对应的联系人姓名的功能

    //sContactCache：用于缓存电话号码和对应的联系人姓名
    //TAG：用于日志输出的标识
    private static HashMap<String, String> sContactCache;
    private static final String TAG = "Contact";

    //SQL 查询条件（ WHERE 后面的语句），用于从联系人数据库中筛选出与给定电话
```

号码匹配的联系人。

```
private static final String CALLER_ID_SELECTION = "PHONE_NUMBERS_EQUAL(" + Phone.NUMBER
+ ",?) AND " + Data.MIMETYPE + "='" + Phone.CONTENT_ITEM_TYPE + "'"
+ " AND " + Data.RAW_CONTACT_ID + " IN "
        + "(SELECT raw_contact_id "
        + " FROM phone_lookup"
        + " WHERE min_match = '+')";
```

//功能简介：用于从 Android 设备的联系人数据库中获取与给定电话号码对应的联系人姓名。

//参数:Context 对象:用于访问系统服务和应用资源 phoneNumber:需要查询的联系人电话号码

```
public static String getContact(Context context, String phoneNumber) {
    // 没映射表就建表，有就查缓存中有没有这个联系人
    if(sContactCache == null) {
        sContactCache = new HashMap<String, String>();
    }

    if(sContactCache.containsKey(phoneNumber)) {
        return sContactCache.get(phoneNumber);
    }

    //缓存没有，就查询数据库
    //构造一个 SQL 查询条件：CALLER_ID_SELECTION 中的"+"被替换为电话号码的最小匹配值
    //然后执行查询语句
    String selection = CALLER_ID_SELECTION.replace("+",
            PhoneNumberUtils.toCallerIDMinMatch(phoneNumber));
    Cursor cursor = context.getContentResolver().query(
            Data.CONTENT_URI,
            new String [] { Phone.DISPLAY_NAME },
            selection,
            new String[] { phoneNumber },
            null);

    //判断查询结果：
    //查询结果不为空，且能够移动到第一条记录：
    //  那么就尝试从 Cursor 中获取联系人姓名，并将其存入缓存 sContactCache。然后返回联系人姓名。
    //  异常情况：如果在获取字符串时发生数组越界异常，则记录一个错误日志并返回 null。
    //  最后都要确保关闭 Cursor 对象，以避免内存泄漏。
    //如果查询结果为空或者没有记录可以移动到（即没有找到匹配的联系人）：
```

```
            //   则记录一条调试日志并返回 null
            if (cursor != null && cursor.moveToFirst()) {
                try {
                    String name = cursor.getString(0);
                    sContactCache.put(phoneNumber, name);
                    return name;
                } catch (IndexOutOfBoundsException e) {
                    Log.e(TAG, " Cursor get string error " + e.toString());
                    return null;
                } finally {
                    cursor.close();
                }
            } else {
                Log.d(TAG, "No contact matched with number:" + phoneNumber);
                return null;
            }
        }
}
```

（2） 类名：Notes

```
package net.micode.notes.data;

import android.net.Uri;
public class Notes {
//      用于表示笔记应用中的各种类型、标识符以及 Intent 的额外数据
    public static final String AUTHORITY = "micode_notes";
    public static final String TAG = "Notes";

    //对 NoteColumns.TYPE 的值进行设置时使用：
    //即不同种类：笔记、文件夹和系统文件夹
    public static final int TYPE_NOTE       = 0;
    public static final int TYPE_FOLDER     = 1;
    public static final int TYPE_SYSTEM     = 2;

    /**
     * Following IDs are system folders' identifiers
     * {@link Notes#ID_ROOT_FOLDER } is default folder
     * {@link Notes#ID_TEMPARAY_FOLDER } is for notes belonging no folder
     * {@link Notes#ID_CALL_RECORD_FOLDER} is to store call records
     */
    //以下 id 是系统文件夹的标识符（即系统文件夹的分类）
    //ID_ROOT_FOLDER：默认文件夹
    //ID_TEMPARAY_FOLDER：不属于文件夹的笔记
    //ID_CALL_RECORD_FOLDER：用于存储通话记录，以便返回
```

```
    //ID_TRASH_FOLER：垃圾回收站
    public static final int ID_ROOT_FOLDER = 0;
    public static final int ID_TEMPARAY_FOLDER = -1;
    public static final int ID_CALL_RECORD_FOLDER = -2;
    public static final int ID_TRASH_FOLER = -3;


    // 额外的数据键，个人理解为就是定义一些布局的 ID
    // 这部分就是用于设置 UI 界面的一些布局或小组件的 id，给它定义成常量了。
    // （这样的封装性可能比较好？因为如果有部分要修改，则直接来这边修改即可，不用在 activity 部分一个一个修改。）
    public static final String INTENT_EXTRA_ALERT_DATE = "net.micode.notes.alert_date";
    public static final String INTENT_EXTRA_BACKGROUND_ID = "net.micode.notes.background_color_id";
    public static final String INTENT_EXTRA_WIDGET_ID = "net.micode.notes.widget_id";
    public static final String INTENT_EXTRA_WIDGET_TYPE = "net.micode.notes.widget_type";
    public static final String INTENT_EXTRA_FOLDER_ID = "net.micode.notes.folder_id";
    public static final String INTENT_EXTRA_CALL_DATE = "net.micode.notes.call_date";

    public static final int TYPE_WIDGET_INVALIDE      = -1;
    public static final int TYPE_WIDGET_2X            = 0;
    public static final int TYPE_WIDGET_4X            = 1;

    // 数据常量：里面定义了两种类型：文本便签和通话记录
    public static class DataConstants {
        public static final String NOTE = TextNote.CONTENT_ITEM_TYPE;
        public static final String CALL_NOTE = CallNote.CONTENT_ITEM_TYPE;
    }

    //下面这些有类似指针的效果？ 其实就是定义一堆访问笔记和文件的 uri
    //GPT：Android 开发中常见的用于定义内容提供者（Content Provider）URI
    //内容提供者是一种 Android 组件，它允许应用程序共享和存储数据。这里定义了一个 URI 来查询数据
    /**
     * Uri to query all notes and folders
     */
    public static final Uri CONTENT_NOTE_URI = Uri.parse("content://" + AUTHORITY + "/note");

    /**
     * Uri to query data
     */
```

```
    public static final Uri CONTENT_DATA_URI = Uri.parse("content://" + AUTHORITY +
"/data");

    public interface NoteColumns {
        // 雨：这个接口定义了一系列静态的、最终的字符串常量，这些常量代表数据库表中的列名。
        // 作用：用于后面创建数据库的表头
        // 总的属性有：ID、父级 ID、创建日期、修改日期、提醒日期、文件（标签）名（摘要？）、小部件 ID、小部件类型、背景颜色 ID、附件、文件中的标签数量、
        //          文件（标签）类型、最后一个同步 ID、本地修改标签、移动前的 ID、谷歌任务 ID、代码版本信息。
        // GPT 提示：在 Android 开发中，当使用 SQLite 数据库时，通常会为表中的每一列定义一个常量，以便在代码中引用。
        // 这样做的好处是，如果以后需要更改列名，只需要在一个地方修改，而不需要在整个代码中搜索和替换。

        /**
         * The unique ID for a row
         * <P> Type: INTEGER (long) </P>
         */
        public static final String ID = "_id";

        /**
         * The parent's id for note or folder
         * <P> Type: INTEGER (long) </P>
         */
        public static final String PARENT_ID = "parent_id";

        /**
         * Created data for note or folder
         * <P> Type: INTEGER (long) </P>
         */
        public static final String CREATED_DATE = "created_date";

        /**
         * Latest modified date
         * <P> Type: INTEGER (long) </P>
         */
        public static final String MODIFIED_DATE = "modified_date";


        /**
         * Alert date
```

```
     * <P> Type: INTEGER (long) </P>
     */
    public static final String ALERTED_DATE = "alert_date";

    /**
     * Folder's name or text content of note
     * <P> Type: TEXT </P>
     */
    // 摘要？
    public static final String SNIPPET = "snippet";

    /**
     * Note's widget id
     * <P> Type: INTEGER (long) </P>
     */
    public static final String WIDGET_ID = "widget_id";

    /**
     * Note's widget type
     * <P> Type: INTEGER (long) </P>
     */
    public static final String WIDGET_TYPE = "widget_type";

    /**
     * Note's background color's id
     * <P> Type: INTEGER (long) </P>
     */
    public static final String BG_COLOR_ID = "bg_color_id";

    /**
     * For text note, it doesn't has attachment, for multi-media
     * note, it has at least one attachment
     * <P> Type: INTEGER </P>
     */
    public static final String HAS_ATTACHMENT = "has_attachment";

    /**
     * Folder's count of notes
     * <P> Type: INTEGER (long) </P>
     */
    public static final String NOTES_COUNT = "notes_count";

    /**
     * The file type: folder or note
```

```
         * <P> Type: INTEGER </P>
         */
        public static final String TYPE = "type";

        /**
         * The last sync id
         * <P> Type: INTEGER (long) </P>
         */
        //雨：在数据同步过程中，这个 ID 可能用来跟踪和识别每次同步操作的唯一性，确保数据的一致性。
        public static final String SYNC_ID = "sync_id";

        /**
         * Sign to indicate local modified or not
         * <P> Type: INTEGER </P>
         */
        public static final String LOCAL_MODIFIED = "local_modified";

        /**
         * Original parent id before moving into temporary folder
         * <P> Type : INTEGER </P>
         */
        public static final String ORIGIN_PARENT_ID = "origin_parent_id";

        /**
         * The gtask id
         * <P> Type : TEXT </P>
         */
        public static final String GTASK_ID = "gtask_id";

        /**
         * The version code
         * <P> Type : INTEGER (long) </P>
         */
        public static final String VERSION = "version";
    }

    public interface DataColumns {
        // DataColumns 的接口，这个接口包含了一系列静态常量，这些常量代表了数据库表中用于存储数据的列名。
        // 每个常量都有相应的注释，说明该列的作用和数据类型。

        /**
         * The unique ID for a row
```

```
     * <P> Type: INTEGER (long) </P>
     */
    public static final String ID = "_id";

    /**
     * The MIME type of the item represented by this row.
     * <P> Type: Text </P>
     */
    //MIME 类型是一种标准，用于标识文档、文件或字节流的性质和格式。在数据库中，这个字段可以用来识别不同类型的数据，例如文本、图片、音频或视频等。
    public static final String MIME_TYPE = "mime_type";

    /**
     * The reference id to note that this data belongs to
     * <P> Type: INTEGER (long) </P>
     */
    //归属的 Note 的 ID
    public static final String NOTE_ID = "note_id";

    /**
     * Created data for note or folder
     * <P> Type: INTEGER (long) </P>
     */
    //创建日期
    public static final String CREATED_DATE = "created_date";

    /**
     * Latest modified date
     * <P> Type: INTEGER (long) </P>
     */
    //最近修改日期
    public static final String MODIFIED_DATE = "modified_date";

    /**
     * Data's content
     * <P> Type: TEXT </P>
     */
    //数据内容
    public static final String CONTENT = "content";


    // 以下 5 个是通用数据列，它们的具体意义取决于 MIME 类型（由 MIME_TYPE 字段指定）。
```

```
        // 不同的 MIME 类型可能需要存储不同类型的数据，这五个字段提供了灵活性，允许根据 MIME 类型来存储相应的数据。
        // 读后面的代码感觉这部分是在表示内容的不同状态？
        /**
         * Generic data column, the meaning is {@link #MIMETYPE} specific, used for
         * integer data type
         * <P> Type: INTEGER </P>
         */
        public static final String DATA1 = "data1";

        /**
         * Generic data column, the meaning is {@link #MIMETYPE} specific, used for
         * integer data type
         * <P> Type: INTEGER </P>
         */
        public static final String DATA2 = "data2";

        /**
         * Generic data column, the meaning is {@link #MIMETYPE} specific, used for
         * TEXT data type
         * <P> Type: TEXT </P>
         */
        public static final String DATA3 = "data3";

        /**
         * Generic data column, the meaning is {@link #MIMETYPE} specific, used for
         * TEXT data type
         * <P> Type: TEXT </P>
         */
        public static final String DATA4 = "data4";

        /**
         * Generic data column, the meaning is {@link #MIMETYPE} specific, used for
         * TEXT data type
         * <P> Type: TEXT </P>
         */
        public static final String DATA5 = "data5";
    }

    //以下是文本便签的定义
    public static final class TextNote implements DataColumns {
        /**
         * Mode to indicate the text in check list mode or not
         * <P> Type: Integer 1:check list mode 0: normal mode </P>
```

```
         */
        public static final String MODE = DATA1; //模式？这个被存在 DATA1 列中

        public static final int MODE_CHECK_LIST = 1; //所处检查列表模式？

        public static final String CONTENT_TYPE = "vnd.android.cursor.dir/text_note"; //定义了 MIME 类型，用于标识文本标签的目录

        public static final String CONTENT_ITEM_TYPE = "vnd.android.cursor.item/text_note";// 定义了 MIME 类型，用于标识文本标签的单个项

        public static final Uri CONTENT_URI = Uri.parse("content://" + AUTHORITY + "/text_note");//文本标签内容提供者（Content Provider）的 URI，用于访问文本标签数据
    }

    // 通话记录的定义？
    public static final class CallNote implements DataColumns {
        /**
         * Call date for this record
         * <P> Type: INTEGER (long) </P>
         */
        public static final String CALL_DATE = DATA1; //一个字符串常量，表示通话记录的日期

        /**
         * Phone number for this record
         * <P> Type: TEXT </P>
         */
        public static final String PHONE_NUMBER = DATA3; //意味着在数据库表中，这个电话号码信息将被存储在 DATA3 列中

        public static final String CONTENT_TYPE = "vnd.android.cursor.dir/call_note";//同样定义了 MIME 类型，是用于标识通话记录的目录。

        public static final String CONTENT_ITEM_TYPE = "vnd.android.cursor.item/call_note";// 同样定义了 MIME 类型，是用于标识通话记录的单个项。

        public static final Uri CONTENT_URI = Uri.parse("content://" + AUTHORITY + "/call_note");//定义了通话记录内容提供者的 URI，用于访问通话记录数据。
    }
}
```

（3） 类名：NotesDatabaseHelper

```
package net.micode.notes.data;

import android.content.ContentValues;
import android.content.Context;
import android.database.sqlite.SQLiteDatabase;
import android.database.sqlite.SQLiteOpenHelper;
import android.util.Log;

import net.micode.notes.data.Notes.DataColumns;
import net.micode.notes.data.Notes.DataConstants;
import net.micode.notes.data.Notes.NoteColumns;


public class NotesDatabaseHelper extends SQLiteOpenHelper {
//      数据库帮助类，用于管理名为 note.db 的 SQLite 数据库。
//      它继承自 SQLiteOpenHelper 类，这是 Android 提供的一个方便的工具类，用于管理数据库的创建和版本更新.
        // 数据库的基本信息；数据库名称和版本信息（在创建实例对象时会用到）
        private static final String DB_NAME = "note.db";

        private static final int DB_VERSION = 4;

        //内部接口：个人理解为两个表名，一个 note，一个 data
        public interface TABLE {
            public static final String NOTE = "note";

            public static final String DATA = "data";
        }

        //一个标签，方便日志输出时识别出信息来自哪里
        private static final String TAG = "NotesDatabaseHelper";

        //静态所有变量，提供一个全局访问点来获取数据库辅助类的唯一实例，使得在应用的任何地方都可以方便地使用它
        private static NotesDatabaseHelper mInstance;

        /* 以下都是一些 SQL 语句，辅助我们来对数据库进行操作 */
        //创建 note 表的语句，这里的 NoteColumns 就是我们刚刚在 Notes 中定义的一个接口，里面定义了一系列静态的数据库表中的列名
        private static final String CREATE_NOTE_TABLE_SQL =
            "CREATE TABLE " + TABLE.NOTE + "(" +
                NoteColumns.ID + " INTEGER PRIMARY KEY," +
                NoteColumns.PARENT_ID + " INTEGER NOT NULL DEFAULT 0," +
```

```
                NoteColumns.ALERTED_DATE + " INTEGER NOT NULL DEFAULT 0," +
                NoteColumns.BG_COLOR_ID + " INTEGER NOT NULL DEFAULT 0," +
                NoteColumns.CREATED_DATE + " INTEGER NOT NULL DEFAULT
(strftime('%s','now') * 1000)," +
                NoteColumns.HAS_ATTACHMENT + " INTEGER NOT NULL DEFAULT 0," +
                NoteColumns.MODIFIED_DATE + " INTEGER NOT NULL DEFAULT
(strftime('%s','now') * 1000)," +
                NoteColumns.NOTES_COUNT + " INTEGER NOT NULL DEFAULT 0," +
                NoteColumns.SNIPPET + " TEXT NOT NULL DEFAULT ''," +
                NoteColumns.TYPE + " INTEGER NOT NULL DEFAULT 0," +
                NoteColumns.WIDGET_ID + " INTEGER NOT NULL DEFAULT 0," +
                NoteColumns.WIDGET_TYPE + " INTEGER NOT NULL DEFAULT -1," +
                NoteColumns.SYNC_ID + " INTEGER NOT NULL DEFAULT 0," +
                NoteColumns.LOCAL_MODIFIED + " INTEGER NOT NULL DEFAULT 0," +
                NoteColumns.ORIGIN_PARENT_ID + " INTEGER NOT NULL DEFAULT 0," +
                NoteColumns.GTASK_ID + " TEXT NOT NULL DEFAULT ''," +
                NoteColumns.VERSION + " INTEGER NOT NULL DEFAULT 0" +
            ")";
```

//同上，创建 data 表的语句，这里的 DataColumns 就是我们刚刚在 Notes 中定义的一个接口，里面定义了一系列静态的数据库表中的列名

```
        private static final String CREATE_DATA_TABLE_SQL =
            "CREATE TABLE " + TABLE.DATA + "(" +
                DataColumns.ID + " INTEGER PRIMARY KEY," +
                DataColumns.MIME_TYPE + " TEXT NOT NULL," +
                DataColumns.NOTE_ID + " INTEGER NOT NULL DEFAULT 0," +
                NoteColumns.CREATED_DATE + " INTEGER NOT NULL DEFAULT
(strftime('%s','now') * 1000)," +
                NoteColumns.MODIFIED_DATE + " INTEGER NOT NULL DEFAULT
(strftime('%s','now') * 1000)," +
                DataColumns.CONTENT + " TEXT NOT NULL DEFAULT ''," +
                DataColumns.DATA1 + " INTEGER," +
                DataColumns.DATA2 + " INTEGER," +
                DataColumns.DATA3 + " TEXT NOT NULL DEFAULT ''," +
                DataColumns.DATA4 + " TEXT NOT NULL DEFAULT ''," +
                DataColumns.DATA5 + " TEXT NOT NULL DEFAULT ''" +
            ")";

        // 功能简介：
        // 创建一个以 note 的 ID 为索引
        // 解读：
        // 用于在 TABLE.DATA 表上创建一个名为 note_id_index 的索引。
        // 这个索引是基于 DataColumns.NOTE_ID 列的。IF NOT EXISTS 确保了如果索引已
经存在，那么就不会尝试重新创建它，避免了可能的错误。
```

```
        // 索引通常用于提高查询性能，特别是在对某个字段进行频繁查询时。
        private static final String CREATE_DATA_NOTE_ID_INDEX_SQL =
            "CREATE INDEX IF NOT EXISTS note_id_index ON " +
            TABLE.DATA + "(" + DataColumns.NOTE_ID + ");";

        /* 以下是一些对便签增删改定义的触发器 */
        /* 总结
        * 这些触发器都是用来维护 NOTE 表和与之相关联的 DATA 表之间数据一致性的。
        * 当在 NOTE 表中发生删除或更新操作时，这些触发器会自动执行相应的数据清理或更新操作，确保数据库中的数据保持正确和一致。
        * 特别是在处理文件夹和回收站等逻辑时，这些触发器起到了非常重要的作用，可以自动管理数据的移动和删除。*/
        /**
         * Increase folder's note count when move note to the folder
         */
        // 功能简介：
        // 添加触发器：增加文件夹的便签个数记录（因为我们会移动便签进入文件夹，这时候文件夹的计数要进行更新）
        // 解读：
        // 定义了一个 SQL 触发器 increase_folder_count_on_update。
        // 触发器是一种特殊的存储过程，它会在指定表上的指定事件（如 INSERT、UPDATE、DELETE）发生时自动执行。
        // 这个触发器会在 TABLE.NOTE 表的 NoteColumns.PARENT_ID 字段更新后执行。
        // 触发器的逻辑是：当某个笔记的 PARENT_ID（即父文件夹 ID）被更新时，它会找到对应的文件夹（通过新的 PARENT_ID），并将该文件夹的 NOTES_COUNT（即笔记数）增加 1。
        private static final String NOTE_INCREASE_FOLDER_COUNT_ON_UPDATE_TRIGGER =
            "CREATE TRIGGER increase_folder_count_on_update "+
            " AFTER UPDATE OF " + NoteColumns.PARENT_ID + " ON " + TABLE.NOTE +
            " BEGIN " +
            "    UPDATE " + TABLE.NOTE +
            "     SET " + NoteColumns.NOTES_COUNT + "=" + NoteColumns.NOTES_COUNT +
" + 1" +
            "    WHERE " + NoteColumns.ID + "=new." + NoteColumns.PARENT_ID + ";" +
            " END";

        /**
         * Decrease folder's note count when move note from folder
         */
        // 功能简介：（触发器和上面的 “增加文件夹的便签个数记录” 同理，就不细节解读了）
        // 添加触发器：减少文件夹的便签个数记录（因为我们会移动便签移出文件夹，这时候文件夹的计数要进行更新）
        private static final String NOTE_DECREASE_FOLDER_COUNT_ON_UPDATE_TRIGGER =
```

```
            "CREATE TRIGGER decrease_folder_count_on_update " +
            " AFTER UPDATE OF " + NoteColumns.PARENT_ID + " ON " + TABLE.NOTE +
            " BEGIN " +
            "    UPDATE " + TABLE.NOTE +
            "      SET " + NoteColumns.NOTES_COUNT + "=" + NoteColumns.NOTES_COUNT +
"-1" +
            "    WHERE " + NoteColumns.ID + "=old." + NoteColumns.PARENT_ID +
            "    AND " + NoteColumns.NOTES_COUNT + ">0" + ";" +
            " END";

        /**
         * Increase folder's note count when insert new note to the folder
         */
        // 功能简介：（触发器原理和上面的 “增加文件夹的便签个数记录” 同理，就不细节解读了）
        // 添加触发器：当我们在文件夹插入便签时，增加文件夹的便签个数记录
        private static final String NOTE_INCREASE_FOLDER_COUNT_ON_INSERT_TRIGGER =
            "CREATE TRIGGER increase_folder_count_on_insert " +
            " AFTER INSERT ON " + TABLE.NOTE +
            " BEGIN " +
            "    UPDATE " + TABLE.NOTE +
            "      SET " + NoteColumns.NOTES_COUNT + "=" + NoteColumns.NOTES_COUNT +
" + 1" +
            "    WHERE " + NoteColumns.ID + "=new." + NoteColumns.PARENT_ID + ";" +
            " END";

        /**
         * Decrease folder's note count when delete note from the folder
         */
        // 功能简介：（触发器原理和上面的 “增加文件夹的便签个数记录” 同理，就不细节解读了）
        // 添加触发器：当我们在文件夹删除便签时，减少文件夹的便签个数记录
        private static final String NOTE_DECREASE_FOLDER_COUNT_ON_DELETE_TRIGGER =
            "CREATE TRIGGER decrease_folder_count_on_delete " +
            " AFTER DELETE ON " + TABLE.NOTE +
            " BEGIN " +
            "    UPDATE " + TABLE.NOTE +
            "      SET " + NoteColumns.NOTES_COUNT + "=" + NoteColumns.NOTES_COUNT +
"-1" +
            "    WHERE " + NoteColumns.ID + "=old." + NoteColumns.PARENT_ID +
            "    AND " + NoteColumns.NOTES_COUNT + ">0;" +
            " END";

        /**
```

```
     * Update note's content when insert data with type {@link DataConstants#NOTE}
     */
    // 功能简介：
    // 添加触发器：当向 DATA 表中插入类型为 NOTE（便签）的数据时，更新 note 表对应的笔记内容。
    // 解读：
    // 在 DATA 表上进行 INSERT 操作后，如果新插入的数据的 MIME_TYPE 为 NOTE，则触发此操作。
    // 它会更新 NOTE 表，将与新插入数据相关联的标签的 SNIPPET（摘要）字段设置为新插入数据的 CONTENT 字段的值
    private static final String DATA_UPDATE_NOTE_CONTENT_ON_INSERT_TRIGGER =
            "CREATE TRIGGER update_note_content_on_insert " +
            " AFTER INSERT ON " + TABLE.DATA +
            " WHEN new." + DataColumns.MIME_TYPE + "='" + DataConstants.NOTE + "'" +
            " BEGIN" +
            "    UPDATE " + TABLE.NOTE +
            "      SET " + NoteColumns.SNIPPET + "=new." + DataColumns.CONTENT +
            "    WHERE " + NoteColumns.ID + "=new." + DataColumns.NOTE_ID + ";" +
            " END";

    /**
     * Update note's content when data with {@link DataConstants#NOTE} type has changed
     */
    // 功能简介：
    // 添加触发器：当 DATA 表中，类型为 NOTE（便签）的数据更改时，更新 note 表对应的笔记内容。
    // 解读：
    // 在 DATA 表上进行 UPDATE 操作后，如果更新前的数据的 MIME_TYPE 为 NOTE，则触发此操作。
    // 它会更新 NOTE 表，将与更新后的数据相关联的笔记的 SNIPPET 字段设置为新数据的 CONTENT 字段的值
    private static final String DATA_UPDATE_NOTE_CONTENT_ON_UPDATE_TRIGGER =
            "CREATE TRIGGER update_note_content_on_update " +
            " AFTER UPDATE ON " + TABLE.DATA +
            " WHEN old." + DataColumns.MIME_TYPE + "='" + DataConstants.NOTE + "'" +
            " BEGIN" +
            "    UPDATE " + TABLE.NOTE +
            "      SET " + NoteColumns.SNIPPET + "=new." + DataColumns.CONTENT +
            "    WHERE " + NoteColumns.ID + "=new." + DataColumns.NOTE_ID + ";" +
            " END";

    /**
     * Update note's content when data with {@link DataConstants#NOTE} type has
```

```
deleted
     */
    // 功能简介：
    // 添加触发器：当 DATA 表中，类型为 NOTE（便签）的数据删除时，更新 note 表对应的笔记内容（置空）。
    // 解读：
    // 在 DATA 表上进行 DELETE 操作后，如果删除的数据的 MIME_TYPE 为 NOTE，则触发此操作。
    // 它会更新 NOTE 表，将与删除的数据相关联的笔记的 SNIPPET 字段设置为空字符串。
    private static final String DATA_UPDATE_NOTE_CONTENT_ON_DELETE_TRIGGER =
        "CREATE TRIGGER update_note_content_on_delete " +
        " AFTER delete ON " + TABLE.DATA +
        " WHEN old." + DataColumns.MIME_TYPE + "='" + DataConstants.NOTE + "'" +
        " BEGIN" +
        "   UPDATE " + TABLE.NOTE +
        "     SET " + NoteColumns.SNIPPET + "=''" +
        "   WHERE " + NoteColumns.ID + "=old." + DataColumns.NOTE_ID + ";" +
        " END";

    /**
     * Delete datas belong to note which has been deleted
     */
    // 功能简介：
    // 添加触发器：当从 NOTE 表中删除笔记时，删除与该笔记相关联的数据（就是删除 data 表中为该 note 的数据）
    // 解读：
    // 在 NOTE 表上进行 DELETE 操作后，此触发器被激活。
    // 它会从 DATA 表中删除所有与已删除的笔记（由 old.ID 表示）相关联的数据行（通过比较 DATA 表中的 NOTE_ID 字段与已删除笔记的 ID 来实现）
    private static final String NOTE_DELETE_DATA_ON_DELETE_TRIGGER =
        "CREATE TRIGGER delete_data_on_delete " +
        " AFTER DELETE ON " + TABLE.NOTE +
        " BEGIN" +
        "   DELETE FROM " + TABLE.DATA +
        "     WHERE " + DataColumns.NOTE_ID + "=old." + NoteColumns.ID + ";" +
        " END";

    /**
     * Delete notes belong to folder which has been deleted
     */
    // 功能简介：
    // 添加触发器：当从 NOTE 表中删除一个文件夹时，删除该文件夹下的所有笔记。
```

```
        // 解读：
        // 在 NOTE 表上进行 DELETE 操作后，如果删除的是一个文件夹（由 old.ID 表示）
        // 触发器会删除所有以该文件夹为父级（PARENT_ID）的笔记（通过比较 NOTE 表中的 PARENT_ID 字段与已删除文件夹的 ID 来实现）
        private static final String FOLDER_DELETE_NOTES_ON_DELETE_TRIGGER =
            "CREATE TRIGGER folder_delete_notes_on_delete " +
            " AFTER DELETE ON " + TABLE.NOTE +
            " BEGIN" +
            "    DELETE FROM " + TABLE.NOTE +
            "      WHERE " + NoteColumns.PARENT_ID + "=old." + NoteColumns.ID + ";" +
            " END";

        /**
         * Move notes belong to folder which has been moved to trash folder
         */
        // 功能简介：
        // 添加触发器：当某个文件夹被移动到回收站时，移动该文件夹下的所有笔记到回收站
        // 解读：
        // 在 NOTE 表上进行 UPDATE 操作后，如果某个文件夹的新 PARENT_ID 字段值等于回收站的 ID（Notes.ID_TRASH_FOLER）
        // 触发器会更新所有以该文件夹为父级（PARENT_ID）的笔记，将它们也移动到回收站。
        private static final String FOLDER_MOVE_NOTES_ON_TRASH_TRIGGER =
            "CREATE TRIGGER folder_move_notes_on_trash " +
            " AFTER UPDATE ON " + TABLE.NOTE +
            " WHEN new." + NoteColumns.PARENT_ID + "=" + Notes.ID_TRASH_FOLER +
            " BEGIN" +
            "    UPDATE " + TABLE.NOTE +
            "      SET " + NoteColumns.PARENT_ID + "=" + Notes.ID_TRASH_FOLER +
            "    WHERE " + NoteColumns.PARENT_ID + "=old." + NoteColumns.ID + ";" +
            " END";

        // 构造器
        public NotesDatabaseHelper(Context context) {
            super(context, DB_NAME, null, DB_VERSION);
        }

        // 创建 note（标签）表
        public void createNoteTable(SQLiteDatabase db) {
            db.execSQL(CREATE_NOTE_TABLE_SQL);
            reCreateNoteTableTriggers(db);
            createSystemFolder(db);
            Log.d(TAG, "note table has been created");
```

```
    }

    // 重新创建或更新与笔记表相关的触发器。
    // 首先，使用 DROP TRIGGER IF EXISTS 语句删除已存在的触发器。确保在重新创建触发器之前，不存在同名的触发器。
    // 然后，使用 db.execSQL()方法执行预定义的 SQL 语句，这些语句用于创建新的触发器。
    private void reCreateNoteTableTriggers(SQLiteDatabase db) {
        db.execSQL("DROP TRIGGER IF EXISTS increase_folder_count_on_update");
        db.execSQL("DROP TRIGGER IF EXISTS decrease_folder_count_on_update");
        db.execSQL("DROP TRIGGER IF EXISTS decrease_folder_count_on_delete");
        db.execSQL("DROP TRIGGER IF EXISTS delete_data_on_delete");
        db.execSQL("DROP TRIGGER IF EXISTS increase_folder_count_on_insert");
        db.execSQL("DROP TRIGGER IF EXISTS folder_delete_notes_on_delete");
        db.execSQL("DROP TRIGGER IF EXISTS folder_move_notes_on_trash");

        db.execSQL(NOTE_INCREASE_FOLDER_COUNT_ON_UPDATE_TRIGGER);
        db.execSQL(NOTE_DECREASE_FOLDER_COUNT_ON_UPDATE_TRIGGER);
        db.execSQL(NOTE_DECREASE_FOLDER_COUNT_ON_DELETE_TRIGGER);
        db.execSQL(NOTE_DELETE_DATA_ON_DELETE_TRIGGER);
        db.execSQL(NOTE_INCREASE_FOLDER_COUNT_ON_INSERT_TRIGGER);
        db.execSQL(FOLDER_DELETE_NOTES_ON_DELETE_TRIGGER);
        db.execSQL(FOLDER_MOVE_NOTES_ON_TRASH_TRIGGER);
    }

    /* 以下部分是操作 SQLite 数据库部分 */
    // 功能简介：
    // 创建通话记录文件夹、默认文件夹、临时文件夹和回收站，并插入相关数据
    // 具体解读：
    // ContentValues 是一个用于存储键值对的类，常用于 SQLite 数据库的插入操作
    // values.put 方法可以向 ContentValues 对象中添加数据。
    // NoteColumns.ID 是存储文件夹 ID 的列名，Notes.ID_CALL_RECORD_FOLDER 是通话记录文件夹的 ID。
    // NoteColumns.TYPE 是存储文件夹类型的列名，Notes.TYPE_SYSTEM 表示这是一个系统文件夹。
    // 使用 db.insert 方法将 values 中的数据插入到 TABLE.NOTE（即标签表）中。
    // 每次插入新数据前，都使用 values.clear()方法清除 ContentValues 对象中的旧数据，确保不会重复插入旧数据。
    // 然后分别创建默认文件夹、临时文件夹和回收站，并以同样的方法插入数据。
    private void createSystemFolder(SQLiteDatabase db) {
        ContentValues values = new ContentValues();

        /**
         * call record foler for call notes
```

```
         */
        values.put(NoteColumns.ID, Notes.ID_CALL_RECORD_FOLDER);
        values.put(NoteColumns.TYPE, Notes.TYPE_SYSTEM);
        db.insert(TABLE.NOTE, null, values);

        /**
         * root folder which is default folder
         */
        // 创建默认文件夹：重复上述步骤，但这次是为根文件夹插入数据。
        values.clear();
        values.put(NoteColumns.ID, Notes.ID_ROOT_FOLDER);
        values.put(NoteColumns.TYPE, Notes.TYPE_SYSTEM);
        db.insert(TABLE.NOTE, null, values);

        /**
         * temporary folder which is used for moving note
         */
        // 创建“临时”文件夹：同样地，为临时文件夹插入数据。
        values.clear();
        values.put(NoteColumns.ID, Notes.ID_TEMPARAY_FOLDER);
        values.put(NoteColumns.TYPE, Notes.TYPE_SYSTEM);
        db.insert(TABLE.NOTE, null, values);

        /**
         * create trash folder
         */
        // 创建“回收站”文件夹：最后，为回收站文件夹插入数据。
        values.clear();
        values.put(NoteColumns.ID, Notes.ID_TRASH_FOLER);
        values.put(NoteColumns.TYPE, Notes.TYPE_SYSTEM);
        db.insert(TABLE.NOTE, null, values);
    }

    //功能简介：
    //创建 data（数据）表
    //解读：
    //这个方法用于创建数据表，以及与之相关的触发器。
    // 创建数据表：使用 db.execSQL 方法执行预定义的 SQL 语句 CREATE_DATA_TABLE_SQL，用于创建数据表。
    //重新创建数据表触发器：调用 reCreateDataTableTriggers 方法，用于删除并重新创建与数据表相关的触发器。
    //创建索引：使用 db.execSQL 方法执行 CREATE_DATA_NOTE_ID_INDEX_SQL 语句，为数据表创建索引。
    //记录日志：使用 Log.d 方法记录一条调试级别的日志，表示数据表已经创建。
```

```
public void createDataTable(SQLiteDatabase db) {
    db.execSQL(CREATE_DATA_TABLE_SQL);
    reCreateDataTableTriggers(db);
    db.execSQL(CREATE_DATA_NOTE_ID_INDEX_SQL);
    Log.d(TAG, "data table has been created");
}

//和上面的 note 表的 reCreate...同理
//重新创建或更新与笔记表相关的触发器。
//首先，使用 DROP TRIGGER IF EXISTS 语句删除已存在的触发器。确保在重新创建触发器之前，不存在同名的触发器。
//然后，使用 db.execSQL()方法执行预定义的 SQL 语句，这些语句用于创建新的触发器。
private void reCreateDataTableTriggers(SQLiteDatabase db) {
    db.execSQL("DROP TRIGGER IF EXISTS update_note_content_on_insert");
    db.execSQL("DROP TRIGGER IF EXISTS update_note_content_on_update");
    db.execSQL("DROP TRIGGER IF EXISTS update_note_content_on_delete");

    db.execSQL(DATA_UPDATE_NOTE_CONTENT_ON_INSERT_TRIGGER);
    db.execSQL(DATA_UPDATE_NOTE_CONTENT_ON_UPDATE_TRIGGER);
    db.execSQL(DATA_UPDATE_NOTE_CONTENT_ON_DELETE_TRIGGER);
}

//解读：
//synchronized 关键字确保在多线程环境下，只有一个线程能够进入这个方法，防止了同时创建多个实例的情况
//getInstance(Context context)方法使用了单例模式来确保整个应用程序中只有一个 NotesDatabaseHelper 实例。
//它首先检查 mInstance（类的静态成员变量，没有在代码片段中显示）是否为 null。
//如果是 null，则创建一个新的 NotesDatabaseHelper 实例，并将其赋值给 mInstance。最后返回 mInstance。
static synchronized NotesDatabaseHelper getInstance(Context context) {
    if (mInstance == null) {
        mInstance = new NotesDatabaseHelper(context);
    }
    return mInstance;
}

//功能简介：
//当数据库首次创建时，onCreate 方法会被调用。
// 这里重写 onCreate 方法，它调用了上述 createNoteTable(db) 和 createDataTable(db)两个方法
//这样首次创建数据库时就多出了两张表。
```

```
@Override
public void onCreate(SQLiteDatabase db) {
    createNoteTable(db);
    createDataTable(db);
}

//功能简介：
//当数据库需要升级时（即数据库的版本号改变），onUpgrade 方法会被调用。
//该方法会根据当前的 oldVersion 和新的 newVersion 来执行相应的升级操作
@Override
public void onUpgrade(SQLiteDatabase db, int oldVersion, int newVersion) {
    boolean reCreateTriggers = false;
    boolean skipV2 = false;

    if (oldVersion == 1) {
        upgradeToV2(db);
        skipV2 = true; // this upgrade including the upgrade from v2 to v3
        oldVersion++;
    }

    if (oldVersion == 2 && !skipV2) {
        upgradeToV3(db);
        reCreateTriggers = true;
        oldVersion++;
    }

    if (oldVersion == 3) {
        upgradeToV4(db);
        oldVersion++;
    }

    if (reCreateTriggers) {
        reCreateNoteTableTriggers(db);
        reCreateDataTableTriggers(db);
    }

    if (oldVersion != newVersion) { //数据库升级失败，抛出一个异常，表示数据库升级失败
        throw new IllegalStateException("Upgrade notes database to version " + newVersion
                + "fails");
    }
}
```

//功能简介：

// 将数据库从版本 1 升级到版本 2。

//解读：

// 首先，它删除了已经存在的 NOTE 和 DATA 表（如果存在的话）。DROP TABLE IF EXISTS 语句确保了即使这些表不存在，也不会抛出错误。

// 然后，它调用了 createNoteTable(db)和 createDataTable(db)方法来重新创建这两个表。这意味着在升级到版本 2 时，这两个表的内容会被完全清除，并重新创建新的空表。

```
private void upgradeToV2(SQLiteDatabase db) {
    db.execSQL("DROP TABLE IF EXISTS " + TABLE.NOTE);
    db.execSQL("DROP TABLE IF EXISTS " + TABLE.DATA);
    createNoteTable(db);
    createDataTable(db);
}
```

//功能简介：

// 将数据库从版本 2（或可能是跳过版本 2 的某个状态）升级到版本 3。

//解读：

// 首先，删除了三个不再使用的触发器（如果存在的话）。触发器是数据库中的一种对象，可以在插入、更新或删除记录时自动执行某些操作。

// 然后，使用 ALTER TABLE 语句修改表结构，向 NOTE 表中添加了一个名为 GTASK_ID 的新列，并设置默认值为空字符串。

// 最后，向 NOTE 表中插入了一条新的系统文件夹记录，表示一个名为“trash folder”的系统文件夹。这可能是用于存储已删除笔记的回收站功能。

```
private void upgradeToV3(SQLiteDatabase db) {
    // drop unused triggers
    db.execSQL("DROP TRIGGER IF EXISTS update_note_modified_date_on_insert");
    db.execSQL("DROP TRIGGER IF EXISTS update_note_modified_date_on_delete");
    db.execSQL("DROP TRIGGER IF EXISTS update_note_modified_date_on_update");
    // add a column for gtask id
    db.execSQL("ALTER TABLE " + TABLE.NOTE + " ADD COLUMN " + NoteColumns.GTASK_ID
            + " TEXT NOT NULL DEFAULT ''");
    // add a trash system folder
    ContentValues values = new ContentValues();
    values.put(NoteColumns.ID, Notes.ID_TRASH_FOLER);
    values.put(NoteColumns.TYPE, Notes.TYPE_SYSTEM);
    db.insert(TABLE.NOTE, null, values);
}
```

//功能简介：

// 这个方法负责将数据库从版本 3 升级到版本 4。

//解读：

// 它向 NOTE 表中添加了一个名为 VERSION 的新列，并设置了默认值为 0。这个新列用于记录标签版本信息。

```
        private void upgradeToV4(SQLiteDatabase db) {
            db.execSQL("ALTER TABLE " + TABLE.NOTE + " ADD COLUMN " + NoteColumns.VERSION
                    + " INTEGER NOT NULL DEFAULT 0");
        }
    }
```

（4） 类名：NotesProvider

```
    package net.micode.notes.data;


    import android.app.SearchManager;
    import android.content.ContentProvider;
    import android.content.ContentUris;
    import android.content.ContentValues;
    import android.content.Intent;
    import android.content.UriMatcher;
    import android.database.Cursor;
    import android.database.sqlite.SQLiteDatabase;
    import android.net.Uri;
    import android.text.TextUtils;
    import android.util.Log;

    import net.micode.notes.R;
    import net.micode.notes.data.Notes.DataColumns;
    import net.micode.notes.data.Notes.NoteColumns;
    import net.micode.notes.data.NotesDatabaseHelper.TABLE;


    public class NotesProvider extends ContentProvider {
    //      Android 应用程序中的一部分：内容提供者（ContentProvider）。
    //      内容提供者是 Android 四大组件之一，它允许应用程序之间共享数据。

        //概述：
        //NotesProvider 的主要功能是作为一个内容提供者，为其他应用程序或组件提供对“Notes”数据的访问。
        //它允许其他应用程序查询、插入、更新或删除标签数据。
        //通过 URI 匹配，NotesProvider 能够区分对哪种数据类型的请求（例如，单独的标签、标签的数据、文件夹操作等），并执行相应的操作。

        //用于匹配不同 URI 的 UriMatcher 对象，通常用于解析传入的 URI，并确定应该执行哪种操作。
        private static final UriMatcher mMatcher;
```

```
        //NotesDatabaseHelper 实类，用来操作 SQLite 数据库，负责创建、更新和查询数据库。
        private NotesDatabaseHelper mHelper;

        //标签，输出日志时用来表示是该类发出的消息
        private static final String TAG = "NotesProvider";

        //6 个 URI 的匹配码，用于区分不同的 URI 类型
        private static final int URI_NOTE                   = 1;
        private static final int URI_NOTE_ITEM              = 2;
        private static final int URI_DATA                   = 3;
        private static final int URI_DATA_ITEM              = 4;

        private static final int URI_SEARCH                 = 5;
        private static final int URI_SEARCH_SUGGEST    = 6;

        //进一步定义了 URI 匹配规则和搜索查询的投影
        //功能概述：
        //初始化了一个 UriMatcher 对象 mMatcher，并添加了一系列的 URI 匹配规则。
        //解读：
        static {
            //创建了一个 UriMatcher 实例，并设置默认匹配码为 NO_MATCH，表示如果没有任何 URI 匹配，则返回这个码。
            mMatcher = new UriMatcher(UriMatcher.NO_MATCH);
            //添加规则，当 URI 的 authority 为 Notes.AUTHORITY，路径为 note 时，返回匹配码 URI_NOTE。
            mMatcher.addURI(Notes.AUTHORITY, "note", URI_NOTE);
            //添加规则，当 URI 的 authority 为 Notes.AUTHORITY，路径为 note/后跟一个数字（#代表数字）时，返回匹配码 URI_NOTE_ITEM。
            mMatcher.addURI(Notes.AUTHORITY, "note/#", URI_NOTE_ITEM);
            //和上面两句同理，但用于匹配数据相关的 URI
            mMatcher.addURI(Notes.AUTHORITY, "data", URI_DATA);
            mMatcher.addURI(Notes.AUTHORITY, "data/#", URI_DATA_ITEM);
            //用于匹配搜索相关的 URI
            mMatcher.addURI(Notes.AUTHORITY, "search", URI_SEARCH);
            //这两行用于匹配搜索建议相关的 URI
            mMatcher.addURI(Notes.AUTHORITY,
SearchManager.SUGGEST_URI_PATH_QUERY, URI_SEARCH_SUGGEST);
            mMatcher.addURI(Notes.AUTHORITY,
SearchManager.SUGGEST_URI_PATH_QUERY + "/*", URI_SEARCH_SUGGEST);
        }

        /**
```

```
     * x'0A' represents the '\n' character in sqlite. For title and content in the search result,
     * we will trim '\n' and white space in order to show more information.
     */
    //功能概述：
    //一个 SQL 查询的投影部分，用于定义查询返回的结果集中应该包含哪些列。
    //解读：（每行对应）
    //返回笔记的 ID。
    //笔记的 ID 也被重命名为 SUGGEST_COLUMN_INTENT_EXTRA_DATA，这通常用于 Android 的搜索建议中，作为传递给相关 Intent 的额外数据。
    //对 SNIPPET 列的处理：首先使用 REPLACE 函数将 x'0A'（即换行符 \n）替换为空字符串，然后使用 TRIM 函数删除前后的空白字符，处理后的结果分别重命名为 SUGGEST_COLUMN_TEXT_1
    //对 SNIPPET 列的处理：首先使用 REPLACE 函数将 x'0A'（即换行符 \n）替换为空字符串，然后使用 TRIM 函数删除前后的空白字符，处理后的结果分别重命名为 SUGGEST_COLUMN_TEXT_2
    //返回一个用于搜索建议图标的资源 ID，并命名为 SUGGEST_COLUMN_ICON_1。
    //返回一个固定的 Intent 动作 ACTION_VIEW，并命名为 SUGGEST_COLUMN_INTENT_ACTION。
    //返回一个内容类型，并命名为 SUGGEST_COLUMN_INTENT_DATA。
    private static final String NOTES_SEARCH_PROJECTION = NoteColumns.ID + "," //返回笔记的 ID
            + NoteColumns.ID + " AS " + SearchManager.SUGGEST_COLUMN_INTENT_EXTRA_DATA + ","
            + "TRIM(REPLACE(" + NoteColumns.SNIPPET + ", x'0A','')) AS " + SearchManager.SUGGEST_COLUMN_TEXT_1 + ","
            + "TRIM(REPLACE(" + NoteColumns.SNIPPET + ", x'0A','')) AS " + SearchManager.SUGGEST_COLUMN_TEXT_2 + ","
            + R.drawable.search_result + " AS " + SearchManager.SUGGEST_COLUMN_ICON_1 + ","
            + "'" + Intent.ACTION_VIEW + "' AS " + SearchManager.SUGGEST_COLUMN_INTENT_ACTION + ","
            + "'" + Notes.TextNote.CONTENT_TYPE + "' AS " + SearchManager.SUGGEST_COLUMN_INTENT_DATA;

    //功能概述：
    //完整的 SQL 查询语句，用于从 TABLE.NOTE 表中检索信息
    //解读：
    // 使用上面定义的投影来选择数据。
    // 并指定从哪个表中选择数据。
    //WHERE 子句包含三个条件：
    // ①搜索 SNIPPET 列中包含特定模式的行（? 是一个占位符，实际查询时会用具体的值替换）。
    // ②父 ID 不为回收站的 ID：排除那些父 ID 为回收站的行。
    // ③只选择类型为 note（标签）的行。
```

```
        private static String NOTES_SNIPPET_SEARCH_QUERY = "SELECT " +
NOTES_SEARCH_PROJECTION
                + " FROM " + TABLE.NOTE
                + " WHERE " + NoteColumns.SNIPPET + " LIKE ?"
                + " AND " + NoteColumns.PARENT_ID + "<>" + Notes.ID_TRASH_FOLER
                + " AND " + NoteColumns.TYPE + "=" + Notes.TYPE_NOTE;

        //重写 onCreate 方法：
        //getContext() 方法被调用以获取当前组件的上下文（Context），以便
NotesDatabaseHelper 能够访问应用程序的资源和其他功能
        //mHelper 用于存储从 NotesDatabaseHelper.getInstance 方法返回的实例。这样，
该实例就可以在整个组件的其他方法中被访问和使用。
        @Override
        public boolean onCreate() {
            mHelper = NotesDatabaseHelper.getInstance(getContext());
            return true;
        }

        //功能：查询数据
        @Override
        public Cursor query(Uri uri, String[] projection, String selection, String[] selectionArgs,
                String sortOrder) {
            //初始化变量：
            //Cursor 对象 c，用来存储查询结果
            //使用 NotesDatabaseHelper 的实例 mHelper 来获取一个可读的数据库实
例
            //定义一个字符串 id，用来存储从 URI 中解析出的 ID
            Cursor c = null;
            SQLiteDatabase db = mHelper.getReadableDatabase();
            String id = null;

            //根据匹配不同的 URI 来进行不同的查询
            switch (mMatcher.match(uri)) {
                //  URI_NOTE：查询整个 NOTE 表。
                //  URI_NOTE_ITEM：查询 NOTE 表中的特定项。ID 从 URI 的路径段
中获取，并添加到查询条件中。
                //  URI_DATA：查询整个 DATA 表。
                //  URI_DATA_ITEM：查询 DATA 表中的特定项。ID 的获取和处理方式
与 URI_NOTE_ITEM 相同。
                case URI_NOTE:
                    c = db.query(TABLE.NOTE, projection, selection, selectionArgs, null, null,
                            sortOrder);
                    break;
                case URI_NOTE_ITEM:
```

```
                id = uri.getPathSegments().get(1);
                c = db.query(TABLE.NOTE, projection, NoteColumns.ID + "=" + id
                        + parseSelection(selection), selectionArgs, null, null,
sortOrder);
                break;
            case URI_DATA:
                c = db.query(TABLE.DATA, projection, selection, selectionArgs, null, null,
                        sortOrder);
                break;
            case URI_DATA_ITEM:
                id = uri.getPathSegments().get(1);
                c = db.query(TABLE.DATA, projection, DataColumns.ID + "=" + id
                        + parseSelection(selection), selectionArgs, null, null,
sortOrder);
                break;

            //URI_SEARCH 和 URI_SEARCH_SUGGEST：处理搜索查询。
            //  代码首先检查是否提供了不应与搜索查询一起使用的参数（如
sortOrder, selection, selectionArgs, 或 projection）。
            //  如果提供了这些参数，则抛出一个 IllegalArgumentException。
            //  根据 URI 类型，从 URI 的路径段或查询参数中获取搜索字符串
searchString。
            //  如果 searchString 为空或无效，则返回 null，表示没有搜索结果。
            case URI_SEARCH:
            case URI_SEARCH_SUGGEST:
                if (sortOrder != null || projection != null) {
                    throw new IllegalArgumentException(
                            "do not specify sortOrder, selection, selectionArgs, or
projection" + "with this query");
                }

                String searchString = null;
                if (mMatcher.match(uri) == URI_SEARCH_SUGGEST) {
                    if (uri.getPathSegments().size() > 1) {
                        searchString = uri.getPathSegments().get(1);
                    }
                } else {
                    searchString = uri.getQueryParameter("pattern");
                }

                if (TextUtils.isEmpty(searchString)) {
                    return null;
                }
```

```
                    //字符串格式化：格式化后的字符串就会是 "%s%"，即包含 s 是任何文本
                    //然后执行原始 SQL 查询
                    try {
                        searchString = String.format("%%%s%%", searchString);
                        c = db.rawQuery(NOTES_SNIPPET_SEARCH_QUERY,
                                new String[] { searchString });
                    } catch (IllegalStateException ex) {
                        Log.e(TAG, "got exception: " + ex.toString());
                    }
                    break;

                    //未知 URI 处理：
                default:
                    throw new IllegalArgumentException("Unknown URI " + uri);
            }
            //如果查询结果不为空（即 Cursor 对象 c 不是 null），则为其设置一个通知 URI。
            //这意味着当与这个 URI 关联的数据发生变化时，任何注册了监听这个 URI 的 ContentObserver 都会被通知。
            if (c != null) {
                c.setNotificationUri(getContext().getContentResolver(), uri);
            }
            return c;
        }

        //功能：插入数据
        //参数：Uri 用来标识要插入数据的表，ContentValues 对象包含要插入的键值对
        @Override
        public Uri insert(Uri uri, ContentValues values) {
            //获取数据库
            //三个长整型变量，分别用来存储数据项 ID、便签 ID 和插入行的 ID
            SQLiteDatabase db = mHelper.getWritableDatabase();
            long dataId = 0, noteId = 0, insertedId = 0;

            //对于 URI_NOTE，将 values 插入到 TABLE.NOTE 表中，并返回插入行的 ID。
            //对于 URI_DATA，首先检查 values 是否包含 DataColumns.NOTE_ID，如果包含，则获取其值。如果不包含，记录一条日志信息。然后，将 values 插入到 TABLE.DATA 表中，并返回插入行的 ID。
            //如果 uri 不是已知的 URI 类型，则抛出一个 IllegalArgumentException。
            switch (mMatcher.match(uri)) {
                case URI_NOTE:
                    insertedId = noteId = db.insert(TABLE.NOTE, null, values);
                    break;
```

```
                case URI_DATA:
                    if (values.containsKey(DataColumns.NOTE_ID)) {
                        noteId = values.getAsLong(DataColumns.NOTE_ID);
                    } else {
                        Log.d(TAG, "Wrong data format without note id:" +
values.toString());
                    }
                    insertedId = dataId = db.insert(TABLE.DATA, null, values);
                    break;
                default:
                    throw new IllegalArgumentException("Unknown URI " + uri);
            }

            //功能：通知变化
            //如果 noteId 或 dataId 大于 0（即成功插入了数据），则使用
ContentResolver 的 notifyChange 方法通知监听这些 URI 的观察者，告知数据已经改变。
            //ContentUris.withAppendedId 方法用于在基本 URI 后面追加一个 ID，形成
完整的 URI。
            // Notify the note uri
            if (noteId > 0) {
                getContext().getContentResolver().notifyChange(
                        ContentUris.withAppendedId(Notes.CONTENT_NOTE_URI,
noteId), null);
            }

            // Notify the data uri
            if (dataId > 0) {
                getContext().getContentResolver().notifyChange(
                        ContentUris.withAppendedId(Notes.CONTENT_DATA_URI, dataId),
null);
            }

            //返回包含新插入数据项 ID 的 Uri。允许调用者知道新插入的数据项的位
置
            return ContentUris.withAppendedId(uri, insertedId);
        }

        //功能：删除数据项
        //参数：uri：标识要删除数据的表或数据项。 selection：一个可选的 WHERE 子
句，用于指定删除条件。 selectionArgs：一个可选的字符串数组，用于替换 selection 中
的占位符
        @Override
        public int delete(Uri uri, String selection, String[] selectionArgs) {
            //count：记录被删除的行数。
```

```
        //id：用于存储从 URI 中解析出的数据项 ID。
        //db：可写的数据库对象，用于执行删除操作。
        //deleteData：一个布尔值，用于标记是否删除了 DATA 表中的数据。
        int count = 0;
        String id = null;
        SQLiteDatabase db = mHelper.getWritableDatabase();
        boolean deleteData = false;

        switch (mMatcher.match(uri)) {
            //URI_NOTE:          修改 selection 语句：确保只删除 ID 大于 0 的笔记。然后执行删除操作并返回被删除的行数。
            //URI_NOTE_ITEM:  从 URI 中解析出 ID。检查 ID 是否小于等于 0，如果是，则不执行删除操作；否则执行删除操作并返回被删除的行数
            //URI_DATA：          执行删除操作并返回被删除的行数。设置 deleteData 为 true，表示删除了 DATA 表中的数据。
            //URI_DATA_ITEM：  先从 URI 中解析出 ID，然后执行删除操作并返回被删除的行数，并设置 deleteData 为 true，表示删除了 DATA 表中的数据。
            case URI_NOTE:
                selection = "(" + selection + ") AND " + NoteColumns.ID + ">0 ";
                count = db.delete(TABLE.NOTE, selection, selectionArgs);
                break;
            case URI_NOTE_ITEM:
                id = uri.getPathSegments().get(1);
                /**
                 * ID that smaller than 0 is system folder which is not allowed to
                 * trash
                 */
                long noteId = Long.valueOf(id);
                if (noteId <= 0) {
                    break;
                }
                count = db.delete(TABLE.NOTE,
                        NoteColumns.ID + "=" + id + parseSelection(selection), selectionArgs);
                break;
            case URI_DATA:
                count = db.delete(TABLE.DATA, selection, selectionArgs);
                deleteData = true;
                break;
            case URI_DATA_ITEM:
                id = uri.getPathSegments().get(1);
                count = db.delete(TABLE.DATA,
                        DataColumns.ID + "=" + id + parseSelection(selection), selectionArgs);
```

```
                    deleteData = true;
                    break;
                default:
                    throw new IllegalArgumentException("Unknown URI " + uri);
            }

            //如果 count 大于 0，说明有数据被删除。
            //如果 deleteData 为 true，则通知监听 Notes.CONTENT_NOTE_URI 的观察者，数据已改变。
            //通知监听传入 uri 的观察者数据已改变。
            if (count > 0) {
                if (deleteData) {

getContext().getContentResolver().notifyChange(Notes.CONTENT_NOTE_URI, null);
                }
                getContext().getContentResolver().notifyChange(uri, null);
            }

            return count;
        }

        //功能：更新数据库的数据
        //参数：uri：标识要更新数据的表或数据项。  values：一个包含新值的键值对集合。
        //      selection：一个可选的 WHERE 子句，用于指定更新条件。 selectionArgs：一个可选的字符串数组，用于替换 selection 中的占位符。
        @Override
        public int update(Uri uri, ContentValues values, String selection, String[] selectionArgs) {
            //count：记录被更新的行数。
            //id：用于存储从 URI 中解析出的数据项 ID。
            //db：可写的 SQLite 数据库对象，用于执行更新操作。
            //updateData：用于标记是否更新了 data 表中的数据。
            int count = 0;
            String id = null;
            SQLiteDatabase db = mHelper.getWritableDatabase();
            boolean updateData = false;

            switch (mMatcher.match(uri)) {
                //URI_NOTE：调用 increaseNoteVersion 方法（用于增加便签版本），然后在 note 表执行更新操作并返回被更新的行数。
                //URI_NOTE_ITEM：从 URI 中解析出 ID，并调用 increaseNoteVersion 方法，传入解析出的 ID，最后在 note 表执行更新操作并返回被更新的行数。
                //URI_DATA：在 data 表执行更新操作并返回被更新的行数。设置
```

updateData 为 true，表示更新了 DATA 表中的数据。

//URI_DATA_ITEM：从 URI 中解析出 ID。执行更新操作并返回被更新的行数。置 updateData 为 true，表示更新了 DATA 表中的数据。

```
                case URI_NOTE:
                    increaseNoteVersion(-1, selection, selectionArgs);
                    count = db.update(TABLE.NOTE, values, selection, selectionArgs);
                    break;
                case URI_NOTE_ITEM:
                    id = uri.getPathSegments().get(1);
                    increaseNoteVersion(Long.valueOf(id), selection, selectionArgs);
                    count = db.update(TABLE.NOTE, values, NoteColumns.ID + "=" + id
                            + parseSelection(selection), selectionArgs);
                    break;
                case URI_DATA:
                    count = db.update(TABLE.DATA, values, selection, selectionArgs);
                    updateData = true;
                    break;
                case URI_DATA_ITEM:
                    id = uri.getPathSegments().get(1);
                    count = db.update(TABLE.DATA, values, DataColumns.ID + "=" + id
                            + parseSelection(selection), selectionArgs);
                    updateData = true;
                    break;
                default:
                    throw new IllegalArgumentException("Unknown URI " + uri);
            }

            //如果 count 大于 0，说明有数据被更新。
            //如果 updateData 为 true，则通知监听 Notes.CONTENT_NOTE_URI 的观察者数据已改变。
            //通知监听传入 uri 的观察者数据已改变。
            if (count > 0) {
                if (updateData) {

getContext().getContentResolver().notifyChange(Notes.CONTENT_NOTE_URI, null);
                }
                getContext().getContentResolver().notifyChange(uri, null);
            }
            return count;
        }

        //解析传入的条件语句：一个 SQL WHERE 子句的一部分
        private String parseSelection(String selection) {
            return (!TextUtils.isEmpty(selection) ? " AND (" + selection + ')' : "");
```

```
        }

        //更新 note 表的 version 列，将其值增加 1。
        private void increaseNoteVersion(long id, String selection, String[] selectionArgs) {
            StringBuilder sql = new StringBuilder(120);
            sql.append("UPDATE ");
            sql.append(TABLE.NOTE);
            sql.append(" SET ");
            sql.append(NoteColumns.VERSION);
            sql.append("=" + NoteColumns.VERSION + "+1 ");

            if (id > 0 || !TextUtils.isEmpty(selection)) {
                sql.append(" WHERE ");
            }
            if (id > 0) {
                sql.append(NoteColumns.ID + "=" + String.valueOf(id));
            }
            if (!TextUtils.isEmpty(selection)) {
                String selectString = id > 0 ? parseSelection(selection) : selection;
                for (String args : selectionArgs) {
                    selectString = selectString.replaceFirst("\\?", args);
                }
                sql.append(selectString);
            }

            mHelper.getWritableDatabase().execSQL(sql.toString());
        }

        @Override
        public String getType(Uri uri) {
            // TODO Auto-generated method stub
            return null;
        }

    }
```

（5） 类名：MetaData

```
    package net.micode.notes.gtask.data;

    public class MetaData extends Task {
      /*
       * 功能描述：得到类的简写名称存入字符串 TAG 中
       * 实现过程：调用 getSimpleName ()函数
       */
        private final static String TAG = MetaData.class.getSimpleName();
```

```
private String mRelatedGid = null;
/*
 * 功能描述：设置数据，即生成元数据库
 * 实现过程：调用 JSONObject 库函数 put ()，Task 类中的 setNotes ()和 setName
()函数
 * 参数注解：
 */
public void setMeta(String gid, JSONObject metaInfo)
{
  //对函数块进行注释
    try {
        metaInfo.put(GTaskStringUtils.META_HEAD_GTASK_ID, gid);
        /*
         * 将这对键值放入 metaInfo 这个 jsonobject 对象中
         */
    } catch (JSONException e) {
        Log.e(TAG, "failed to put related gid");
        /*
         * 输出错误信息
         */
    }
    setNotes(metaInfo.toString());
    setName(GTaskStringUtils.META_NOTE_NAME);
}
/*
 * 功能描述：获取相关联的 Gid
 */
public String getRelatedGid() {
    return mRelatedGid;
}
/*
 * 功能描述：判断当前数据是否为空，若为空则返回真即值得保存
 * Made By CuiCan
 */
@Override
public boolean isWorthSaving() {
    return getNotes() != null;
}
/*
 * 功能描述：使用远程 json 数据对象设置元数据内容
 * 实现过程：调用父类 Task 中的 setContentByRemoteJSON ()函数，并
 * 参数注解：
 */
@Override
```

```
    public void setContentByRemoteJSON(JSONObject js) {
        super.setContentByRemoteJSON(js);
        if (getNotes() != null) {
            try {
                JSONObject metaInfo = new JSONObject(getNotes().trim());
                mRelatedGid                                                    =
metaInfo.getString(GTaskStringUtils.META_HEAD_GTASK_ID);
            } catch (JSONException e) {
                Log.w(TAG, "failed to get related gid");
                /*
                 * 输出警告信息
                 */
                mRelatedGid = null;
            }
        }
    }
    /*
     * 功能描述：使用本地 json 数据对象设置元数据内容，一般不会用到，若用到，
则抛出异常
     * Made By CuiCan
     */
    @Override
    public void setContentByLocalJSON(JSONObject js) {
        // this function should not be called
        throw new IllegalAccessError("MetaData:setContentByLocalJSON should not be
called");
        /*
         * 传递非法参数异常
         */
    }
    /*
     * 功能描述：从元数据内容中获取本地 json 对象，一般不会用到，若用到，则
抛出异常
     * Made By CuiCan
     */
    @Override
    public JSONObject getLocalJSONFromContent() {
        throw new IllegalAccessError("MetaData:getLocalJSONFromContent should not
be called");
        /*
         * 传递非法参数异常
         * Made By Cui Can
         */
    }
```

```
        /*
         * 功能描述：获取同步动作状态，一般不会用到，若用到，则抛出异常
         * Made By CuiCan
         */
        @Override
        public int getSyncAction(Cursor c) {
            throw new IllegalAccessError("MetaData:getSyncAction should not be called");
            /*
             * 传递非法参数异常
             * Made By Cui Can
             */
        }

    }
```

（6） 类名：Node

```
    package net.micode.notes.gtask.data;

    import android.database.Cursor;

    import org.json.JSONObject;

    /**
     * 应该是同步操作的基础数据类型，定义了相关指示同步操作的常量
     * 关键字：abstract
     */
    public abstract class Node {
      //定义了各种用于表征同步状态的常量
        public static final int SYNC_ACTION_NONE = 0;// 本地和云端都无可更新内容（即本地和云端内容一致）

        public static final int SYNC_ACTION_ADD_REMOTE = 1;// 需要在远程云端增加内容

        public static final int SYNC_ACTION_ADD_LOCAL = 2;// 需要在本地增加内容

        public static final int SYNC_ACTION_DEL_REMOTE = 3;// 需要在远程云端删除内容

        public static final int SYNC_ACTION_DEL_LOCAL = 4;// 需要在本地删除内容

        public static final int SYNC_ACTION_UPDATE_REMOTE = 5;// 需要将本地内容更新到远程云端

        public static final int SYNC_ACTION_UPDATE_LOCAL = 6;// 需要将远程云端内容更新到本地
```

```
public static final int SYNC_ACTION_UPDATE_CONFLICT = 7;// 同步出现冲突

public static final int SYNC_ACTION_ERROR = 8;// 同步出现错误

private String mGid;

private String mName;

private long mLastModified;//记录最后一次修改时间

private boolean mDeleted;//表征是否被删除

public Node() {
    mGid = null;
    mName = "";
    mLastModified = 0;
    mDeleted = false;
}

public abstract JSONObject getCreateAction(int actionId);

public abstract JSONObject getUpdateAction(int actionId);

public abstract void setContentByRemoteJSON(JSONObject js);

public abstract void setContentByLocalJSON(JSONObject js);

public abstract JSONObject getLocalJSONFromContent();

public abstract int getSyncAction(Cursor c);

public void setGid(String gid) {
    this.mGid = gid;
}

public void setName(String name) {
    this.mName = name;
}

public void setLastModified(long lastModified) {
    this.mLastModified = lastModified;
}
```

```
    public void setDeleted(boolean deleted) {
        this.mDeleted = deleted;
    }

    public String getGid() {
        return this.mGid;
    }

    public String getName() {
        return this.mName;
    }

    public long getLastModified() {
        return this.mLastModified;
    }

    public boolean getDeleted() {
        return this.mDeleted;
    }

}
```

（7）类名：SqlData

```
/*
 * Description：用于支持小米便签最底层的数据库相关操作，和 sqlnote 的关系上是子集关系，即 data 是 note 的子集（节点）。
 * SqlData 其实就是也就是所谓数据中的数据
 */

package net.micode.notes.gtask.data;
/*
 *  功能描述：
 *  实现过程：
 *  参数注解：
 * Made By CuiCan
 */

public class SqlData {
   /*
    *  功能描述：得到类的简写名称存入字符串 TAG 中
    *  实现过程：调用 getSimpleName ()函数
    * Made By CuiCan
    */
     private static final String TAG = SqlData.class.getSimpleName();
```

private static final int INVALID_ID = -99999;

=为 mDataId 置初始值-99999

```
        /**
         * 来自 Notes 类中定义的 DataColumn 中的一些常量
         */

        // 集合了 interface DataColumns 中所有 SF 常量
        public static final String[] PROJECTION_DATA = new String[] {
                DataColumns.ID,  DataColumns.MIME_TYPE,  DataColumns.CONTENT,
DataColumns.DATA1,
                DataColumns.DATA3
        };

        /**
         * 以下五个变量作为 sql 表中 5 列的编号
         */
        public static final int DATA_ID_COLUMN = 0;

        public static final int DATA_MIME_TYPE_COLUMN = 1;

        public static final int DATA_CONTENT_COLUMN = 2;

        public static final int DATA_CONTENT_DATA_1_COLUMN = 3;

        public static final int DATA_CONTENT_DATA_3_COLUMN = 4;

        private ContentResolver mContentResolver;
        //判断是否直接用 Content 生成，是为 true，否则为 false
        private boolean mIsCreate;

        private long mDataId;

        private String mDataMimeType;

        private String mDataContent;

        private long mDataContentData1;

        private String mDataContentData3;

        private ContentValues mDiffDataValues;
```

```
    /*
     * 功能描述：构造函数，用于初始化数据
       * 参数注解：mContentResolver 用于获取 ContentProvider 提供的数据
       * 参数注解： mIsCreate 表征当前数据是用哪种方式创建（两种构造函数的参数不同）
       * 参数注解：
     * Made By CuiCan
     */
      public SqlData(Context context) {
            mContentResolver = context.getContentResolver();
            mIsCreate = true;
            mDataId = INVALID_ID;//mDataId 置初始值-99999
            mDataMimeType = DataConstants.NOTE;
            mDataContent = "";
            mDataContentData1 = 0;
            mDataContentData3 = "";
            mDiffDataValues = new ContentValues();
      }


    /*
     * 功能描述：构造函数，初始化数据
       * 参数注解：mContentResolver 用于获取 ContentProvider 提供的数据
       * 参数注解： mIsCreate 表征当前数据是用哪种方式创建（两种构造函数的参数不同）
       * 参数注解：
     * Made By CuiCan
     */
      public SqlData(Context context, Cursor c) {
            mContentResolver = context.getContentResolver();
            mIsCreate = false;
            loadFromCursor(c);
            mDiffDataValues = new ContentValues();
      }

      /*
       * 功能描述：从光标处加载数据
       * 从当前的光标处将五列的数据加载到该类的对象
       * Made By CuiCan
       */
      private void loadFromCursor(Cursor c) {
            mDataId = c.getLong(DATA_ID_COLUMN);
            mDataMimeType = c.getString(DATA_MIME_TYPE_COLUMN);
            mDataContent = c.getString(DATA_CONTENT_COLUMN);
```

```
            mDataContentData1 = c.getLong(DATA_CONTENT_DATA_1_COLUMN);
            mDataContentData3 = c.getString(DATA_CONTENT_DATA_3_COLUMN);
        }


    /*
     * 功能描述：设置用于共享的数据，并提供异常抛出与处理机制
     * 参数注解：
     * Made By CuiCan
     */
      public void setContent(JSONObject js) throws JSONException {
            //如果传入的 JSONObject 对象中有 DataColumns.ID 这一项，则设置，否则设为 INVALID_ID
            long dataId = js.has(DataColumns.ID) ? js.getLong(DataColumns.ID) : INVALID_ID;
            if (mIsCreate || mDataId != dataId) {
                mDiffDataValues.put(DataColumns.ID, dataId);
            }
            mDataId = dataId;

            String dataMimeType = js.has(DataColumns.MIME_TYPE) ? js.getString(DataColumns.MIME_TYPE)
                    : DataConstants.NOTE;
            if (mIsCreate || !mDataMimeType.equals(dataMimeType)) {
                mDiffDataValues.put(DataColumns.MIME_TYPE, dataMimeType);
            }
            mDataMimeType = dataMimeType;

            String dataContent = js.has(DataColumns.CONTENT) ? js.getString(DataColumns.CONTENT) : "";
            if (mIsCreate || !mDataContent.equals(dataContent)) {
                mDiffDataValues.put(DataColumns.CONTENT, dataContent);
            }
            mDataContent = dataContent;

            long dataContentData1 = js.has(DataColumns.DATA1) ? js.getLong(DataColumns.DATA1) : 0;
            if (mIsCreate || mDataContentData1 != dataContentData1) {
                mDiffDataValues.put(DataColumns.DATA1, dataContentData1);
            }
            mDataContentData1 = dataContentData1;

            String dataContentData3 = js.has(DataColumns.DATA3) ? js.getString(DataColumns.DATA3) : "";
            if (mIsCreate || !mDataContentData3.equals(dataContentData3)) {
```

```
                mDiffDataValues.put(DataColumns.DATA3, dataContentData3);
            }
            mDataContentData3 = dataContentData3;
        }


    /*
     * 功能描述：获取共享的数据内容，并提供异常抛出与处理机制
     * 参数注解：
     * Made By CuiCan
     */
      public JSONObject getContent() throws JSONException {
            if (mIsCreate) {
                Log.e(TAG, "it seems that we haven't created this in database yet");
                return null;
            }
            //创建 JSONObject 对象。并将相关数据放入其中，并返回。
            JSONObject js = new JSONObject();
            js.put(DataColumns.ID, mDataId);
            js.put(DataColumns.MIME_TYPE, mDataMimeType);
            js.put(DataColumns.CONTENT, mDataContent);
            js.put(DataColumns.DATA1, mDataContentData1);
            js.put(DataColumns.DATA3, mDataContentData3);
            return js;
      }

    /*
     * 功能描述：commit 函数用于把当前造作所做的修改保存到数据库
     * 参数注解：
     * Made By CuiCan
     */
      public void commit(long noteId, boolean validateVersion, long version) {

            if (mIsCreate) {
                if (mDataId == INVALID_ID &&
mDiffDataValues.containsKey(DataColumns.ID)) {
                    mDiffDataValues.remove(DataColumns.ID);
                }

                mDiffDataValues.put(DataColumns.NOTE_ID, noteId);
                Uri uri = mContentResolver.insert(Notes.CONTENT_DATA_URI,
mDiffDataValues);
                try {
                    mDataId = Long.valueOf(uri.getPathSegments().get(1));
```

```
                    } catch (NumberFormatException e) {
                        Log.e(TAG, "Get note id error :" + e.toString());
                        throw new ActionFailureException("create note failed");
                    }
                } else {
                    if (mDiffDataValues.size() > 0) {
                        int result = 0;
                        if (!validateVersion) {
                            result = mContentResolver.update(ContentUris.withAppendedId(
                                    Notes.CONTENT_DATA_URI, mDataId), mDiffDataValues,
null, null);
                        } else {
                            result = mContentResolver.update(ContentUris.withAppendedId(
                                    Notes.CONTENT_DATA_URI, mDataId), mDiffDataValues,
                                    " ? in (SELECT " + NoteColumns.ID + " FROM " +
TABLE.NOTE
                                            + " WHERE " + NoteColumns.VERSION + "=?)",
new String[] {
                                            String.valueOf(noteId), String.valueOf(version)
                                    });
                        }
                        if (result == 0) {
                            Log.w(TAG, "there is no update. maybe user updates note when
syncing");
                        }
                    }
                }

                mDiffDataValues.clear();
                mIsCreate = false;
            }

            /*
             * 功能描述：获取当前 id
             * 实现过程：
             * 参数注解：
             * Made By CuiCan
             */
            public long getId() {
                return mDataId;
            }
        }
```

（8）类名：SqlNote

```
        /*
```

```
    * Description：用于支持小米便签最底层的数据库相关操作，和 sqlnote 的关系上是子集关系，即 data 是 note 的子集（节点）。
    * SqlData 其实就是也就是所谓数据中的数据
    */

   package net.micode.notes.gtask.data;
   /*
    * 功能描述：
    * 实现过程：
    * 参数注解：
    * Made By CuiCan
    */

   public class SqlData {
     /*
      * 功能描述：得到类的简写名称存入字符串 TAG 中
      * 实现过程：调用 getSimpleName ()函数
      * Made By CuiCan
      */
       private static final String TAG = SqlData.class.getSimpleName();

       private static final int INVALID_ID = -99999;//为 mDataId 置初始值-99999


       /**
        * 来自 Notes 类中定义的 DataColumn 中的一些常量
        */

       // 集合了 interface DataColumns 中所有 SF 常量
       public static final String[] PROJECTION_DATA = new String[] {
               DataColumns.ID,    DataColumns.MIME_TYPE,    DataColumns.CONTENT,
DataColumns.DATA1,
               DataColumns.DATA3
       };

       /**
        * 以下五个变量作为 sql 表中 5 列的编号
        */
       public static final int DATA_ID_COLUMN = 0;

       public static final int DATA_MIME_TYPE_COLUMN = 1;

       public static final int DATA_CONTENT_COLUMN = 2;
```

```
    public static final int DATA_CONTENT_DATA_1_COLUMN = 3;

    public static final int DATA_CONTENT_DATA_3_COLUMN = 4;

    private ContentResolver mContentResolver;
    //判断是否直接用 Content 生成，是为 true，否则为 false
    private boolean mIsCreate;

    private long mDataId;

    private String mDataMimeType;

    private String mDataContent;

    private long mDataContentData1;

    private String mDataContentData3;

    private ContentValues mDiffDataValues;

  /*
   * 功能描述：构造函数，用于初始化数据
     * 参数注解：mContentResolver 用于获取 ContentProvider 提供的数据
     * 参数注解： mIsCreate 表征当前数据是用哪种方式创建（两种构造函数的参数不同）
     * 参数注解：
   * Made By CuiCan
   */
    public SqlData(Context context) {
        mContentResolver = context.getContentResolver();
        mIsCreate = true;
        mDataId = INVALID_ID;//mDataId 置初始值-99999
        mDataMimeType = DataConstants.NOTE;
        mDataContent = "";
        mDataContentData1 = 0;
        mDataContentData3 = "";
        mDiffDataValues = new ContentValues();
    }


  /*
   * 功能描述：构造函数，初始化数据
     * 参数注解：mContentResolver 用于获取 ContentProvider 提供的数据
     * 参数注解： mIsCreate 表征当前数据是用哪种方式创建（两种构造函数的参
```

数不同）

```
     * 参数注解：
    * Made By CuiCan
    */
     public SqlData(Context context, Cursor c) {
          mContentResolver = context.getContentResolver();
          mIsCreate = false;
          loadFromCursor(c);
          mDiffDataValues = new ContentValues();
     }

     /*
      * 功能描述：从光标处加载数据
      * 从当前的光标处将五列的数据加载到该类的对象
      * Made By CuiCan
      */
     private void loadFromCursor(Cursor c) {
          mDataId = c.getLong(DATA_ID_COLUMN);
          mDataMimeType = c.getString(DATA_MIME_TYPE_COLUMN);
          mDataContent = c.getString(DATA_CONTENT_COLUMN);
          mDataContentData1 = c.getLong(DATA_CONTENT_DATA_1_COLUMN);
          mDataContentData3 = c.getString(DATA_CONTENT_DATA_3_COLUMN);
     }


   /*
    * 功能描述：设置用于共享的数据，并提供异常抛出与处理机制
    * 参数注解：
    * Made By CuiCan
    */
     public void setContent(JSONObject js) throws JSONException {
          //如果传入的 JSONObject 对象中有 DataColumns.ID 这一项，则设置，否则设为 INVALID_ID
          long dataId = js.has(DataColumns.ID) ? js.getLong(DataColumns.ID) : INVALID_ID;
          if (mIsCreate || mDataId != dataId) {
               mDiffDataValues.put(DataColumns.ID, dataId);
          }
          mDataId = dataId;

          String dataMimeType = js.has(DataColumns.MIME_TYPE) ? js.getString(DataColumns.MIME_TYPE)
                    : DataConstants.NOTE;
          if (mIsCreate || !mDataMimeType.equals(dataMimeType)) {
               mDiffDataValues.put(DataColumns.MIME_TYPE, dataMimeType);
```

```
            }
            mDataMimeType = dataMimeType;

            String dataContent = js.has(DataColumns.CONTENT) ?
js.getString(DataColumns.CONTENT) : "";
            if (mIsCreate || !mDataContent.equals(dataContent)) {
                mDiffDataValues.put(DataColumns.CONTENT, dataContent);
            }
            mDataContent = dataContent;

            long dataContentData1 = js.has(DataColumns.DATA1) ?
js.getLong(DataColumns.DATA1) : 0;
            if (mIsCreate || mDataContentData1 != dataContentData1) {
                mDiffDataValues.put(DataColumns.DATA1, dataContentData1);
            }
            mDataContentData1 = dataContentData1;

            String dataContentData3 = js.has(DataColumns.DATA3) ?
js.getString(DataColumns.DATA3) : "";
            if (mIsCreate || !mDataContentData3.equals(dataContentData3)) {
                mDiffDataValues.put(DataColumns.DATA3, dataContentData3);
            }
            mDataContentData3 = dataContentData3;
        }


    /*
     * 功能描述：获取共享的数据内容，并提供异常抛出与处理机制
     * 参数注解：
     * Made By CuiCan
     */
      public JSONObject getContent() throws JSONException {
            if (mIsCreate) {
                Log.e(TAG, "it seems that we haven't created this in database yet");
                return null;
            }
            //创建 JSONObject 对象。并将相关数据放入其中，并返回。
            JSONObject js = new JSONObject();
            js.put(DataColumns.ID, mDataId);
            js.put(DataColumns.MIME_TYPE, mDataMimeType);
            js.put(DataColumns.CONTENT, mDataContent);
            js.put(DataColumns.DATA1, mDataContentData1);
            js.put(DataColumns.DATA3, mDataContentData3);
            return js;
```

```
        }

    /*
     * 功能描述：commit 函数用于把当前造作所做的修改保存到数据库
     * 参数注解：
     * Made By CuiCan
     */
      public void commit(long noteId, boolean validateVersion, long version) {

            if (mIsCreate) {
                if (mDataId == INVALID_ID &&
mDiffDataValues.containsKey(DataColumns.ID)) {
                    mDiffDataValues.remove(DataColumns.ID);
                }

                mDiffDataValues.put(DataColumns.NOTE_ID, noteId);
                Uri uri = mContentResolver.insert(Notes.CONTENT_DATA_URI,
mDiffDataValues);
                try {
                    mDataId = Long.valueOf(uri.getPathSegments().get(1));
                } catch (NumberFormatException e) {
                    Log.e(TAG, "Get note id error :" + e.toString());
                    throw new ActionFailureException("create note failed");
                }
            } else {
                if (mDiffDataValues.size() > 0) {
                    int result = 0;
                    if (!validateVersion) {
                        result = mContentResolver.update(ContentUris.withAppendedId(
                                Notes.CONTENT_DATA_URI, mDataId), mDiffDataValues,
null, null);
                    } else {
                        result = mContentResolver.update(ContentUris.withAppendedId(
                                Notes.CONTENT_DATA_URI, mDataId), mDiffDataValues,
                                " ? in (SELECT " + NoteColumns.ID + " FROM " +
TABLE.NOTE
                                        + " WHERE " + NoteColumns.VERSION + "=?)",
new String[] {
                                        String.valueOf(noteId), String.valueOf(version)
                                });
                    }
                    if (result == 0) {
                        Log.w(TAG, "there is no update. maybe user updates note when
syncing");
```

```
                    }
                }
            }

            mDiffDataValues.clear();
            mIsCreate = false;
        }

        /*
         * 功能描述：获取当前 id
         * 实现过程：
         * 参数注解：
         * Made By CuiCan
         */
        public long getId() {
            return mDataId;
        }
}
```

（9）类名：Task

```
package net.micode.notes.gtask.data;

public class Task extends Node {
    private static final String TAG = Task.class.getSimpleName();

    private boolean mCompleted;//是否完成

    private String mNotes;

    private JSONObject mMetaInfo;//将在实例中存储数据的类型

    private Task mPriorSibling;//对应的优先兄弟 Task 的指针（待完善）

    private TaskList mParent;//所在的任务列表的指针

    public Task() {
        super();
        mCompleted = false;
        mNotes = null;
        mPriorSibling = null;//TaskList 中当前 Task 前面的 Task 的指针
        mParent = null;//当前 Task 所在的 TaskList
        mMetaInfo = null;
    }

    public JSONObject getCreateAction(int actionId) {
```

```
        JSONObject js = new JSONObject();

        try {
            // action_type
            js.put(GTaskStringUtils.GTASK_JSON_ACTION_TYPE,
                    GTaskStringUtils.GTASK_JSON_ACTION_TYPE_CREATE);

            // action_id
            js.put(GTaskStringUtils.GTASK_JSON_ACTION_ID, actionId);

            // index
            js.put(GTaskStringUtils.GTASK_JSON_INDEX,
mParent.getChildTaskIndex(this));

            // entity_delta
            JSONObject entity = new JSONObject();
            entity.put(GTaskStringUtils.GTASK_JSON_NAME, getName());
            entity.put(GTaskStringUtils.GTASK_JSON_CREATOR_ID, "null");
            entity.put(GTaskStringUtils.GTASK_JSON_ENTITY_TYPE,
                    GTaskStringUtils.GTASK_JSON_TYPE_TASK);
            if (getNotes() != null) {
                entity.put(GTaskStringUtils.GTASK_JSON_NOTES, getNotes());
            }
            js.put(GTaskStringUtils.GTASK_JSON_ENTITY_DELTA, entity);

            // parent_id
            if (mParent!= null) {
            js.put(GTaskStringUtils.GTASK_JSON_PARENT_ID, mParent.getGid());
            }

            // dest_parent_type
            js.put(GTaskStringUtils.GTASK_JSON_DEST_PARENT_TYPE,
                    GTaskStringUtils.GTASK_JSON_TYPE_GROUP);

            // list_id
            if (mParent!= null) {
            js.put(GTaskStringUtils.GTASK_JSON_LIST_ID, mParent.getGid());
            }

            // prior_sibling_id
            if (mPriorSibling != null) {
                js.put(GTaskStringUtils.GTASK_JSON_PRIOR_SIBLING_ID,
mPriorSibling.getGid());
            }
```

```
            } catch (JSONException e) {
                Log.e(TAG, e.toString());
                e.printStackTrace();
                throw new ActionFailureException("fail to generate task-create
jsonobject");
            }

            return js;
        }

        public JSONObject getUpdateAction(int actionId) {
            JSONObject js = new JSONObject();

            try {
                // action_type
                js.put(GTaskStringUtils.GTASK_JSON_ACTION_TYPE,
                        GTaskStringUtils.GTASK_JSON_ACTION_TYPE_UPDATE);

                // action_id
                js.put(GTaskStringUtils.GTASK_JSON_ACTION_ID, actionId);

                // id
                js.put(GTaskStringUtils.GTASK_JSON_ID, getGid());

                // entity_delta
                JSONObject entity = new JSONObject();
                entity.put(GTaskStringUtils.GTASK_JSON_NAME, getName());
                if (getNotes() != null) {
                    entity.put(GTaskStringUtils.GTASK_JSON_NOTES, getNotes());
                }
                entity.put(GTaskStringUtils.GTASK_JSON_DELETED, getDeleted());
                js.put(GTaskStringUtils.GTASK_JSON_ENTITY_DELTA, entity);

            } catch (JSONException e) {
                Log.e(TAG, e.toString());
                e.printStackTrace();
                throw new ActionFailureException("fail to generate task-update
jsonobject");
            }

            return js;
        }
```

```
    public void setContentByRemoteJSON(JSONObject js) {
        if (js != null) {
            try {
                // id
                if (js.has(GTaskStringUtils.GTASK_JSON_ID)) {
                    setGid(js.getString(GTaskStringUtils.GTASK_JSON_ID));
                }

                // last_modified
                if (js.has(GTaskStringUtils.GTASK_JSON_LAST_MODIFIED)) {

setLastModified(js.getLong(GTaskStringUtils.GTASK_JSON_LAST_MODIFIED));
                }

                // name
                if (js.has(GTaskStringUtils.GTASK_JSON_NAME)) {
                    setName(js.getString(GTaskStringUtils.GTASK_JSON_NAME));
                }

                // notes
                if (js.has(GTaskStringUtils.GTASK_JSON_NOTES)) {
                    setNotes(js.getString(GTaskStringUtils.GTASK_JSON_NOTES));
                }

                // deleted
                if (js.has(GTaskStringUtils.GTASK_JSON_DELETED)) {

setDeleted(js.getBoolean(GTaskStringUtils.GTASK_JSON_DELETED));
                }

                // completed
                if (js.has(GTaskStringUtils.GTASK_JSON_COMPLETED)) {

setCompleted(js.getBoolean(GTaskStringUtils.GTASK_JSON_COMPLETED));
                }
            } catch (JSONException e) {
                Log.e(TAG, e.toString());
                e.printStackTrace();
                throw new ActionFailureException("fail to get task content from
jsonobject");
            }
        }
    }
```

```
    public void setContentByLocalJSON(JSONObject js)
     {
            if (js == null || !js.has(GTaskStringUtils.META_HEAD_NOTE)
                    || !js.has(GTaskStringUtils.META_HEAD_DATA)) {
                Log.w(TAG, "setContentByLocalJSON: nothing is avaiable");
            }

            try {
                JSONObject note = js.getJSONObject(GTaskStringUtils.META_HEAD_NOTE);
                JSONArray dataArray = js.getJSONArray(GTaskStringUtils.META_HEAD_DATA);

                if (note.getInt(NoteColumns.TYPE) != Notes.TYPE_NOTE) {
                    Log.e(TAG, "invalid type");
                    return;
                }

                for (int i = 0; i < dataArray.length(); i++) {
                    JSONObject data = dataArray.getJSONObject(i);
                    if (TextUtils.equals(data.getString(DataColumns.MIME_TYPE),
DataConstants.NOTE)) {
                        setName(data.getString(DataColumns.CONTENT));
                        break;
                    }
                }

            } catch (JSONException e) {
                Log.e(TAG, e.toString());
                e.printStackTrace();
            }
        }

        public JSONObject getLocalJSONFromContent() {
            String name = getName();
            try {
                if (mMetaInfo == null) {
                    // new task created from web
                    if (name == null) {
                        Log.w(TAG, "the note seems to be an empty one");
                        return null;
                    }

                    JSONObject js = new JSONObject();
```

```
                    JSONObject note = new JSONObject();
                    JSONArray dataArray = new JSONArray();
                    JSONObject data = new JSONObject();
                    data.put(DataColumns.CONTENT, name);
                    dataArray.put(data);
                    js.put(GTaskStringUtils.META_HEAD_DATA, dataArray);
                    note.put(NoteColumns.TYPE, Notes.TYPE_NOTE);
                    js.put(GTaskStringUtils.META_HEAD_NOTE, note);
                    return js;
                } else {
                    // synced task
                    JSONObject note = mMetaInfo.getJSONObject(GTaskStringUtils.META_HEAD_NOTE);
                    JSONArray dataArray = mMetaInfo.getJSONArray(GTaskStringUtils.META_HEAD_DATA);

                    for (int i = 0; i < dataArray.length(); i++) {
                        JSONObject data = dataArray.getJSONObject(i);
                        if (TextUtils.equals(data.getString(DataColumns.MIME_TYPE),
DataConstants.NOTE)) {
                            data.put(DataColumns.CONTENT, getName());
                            break;
                        }
                    }

                    note.put(NoteColumns.TYPE, Notes.TYPE_NOTE);
                    return mMetaInfo;
                }
            } catch (JSONException e) {
                Log.e(TAG, e.toString());
                e.printStackTrace();
                return null;
            }
        }

        public void setMetaInfo(MetaData metaData) {
            if (metaData != null && metaData.getNotes() != null) {
                try {
                    mMetaInfo = new JSONObject(metaData.getNotes());
                } catch (JSONException e) {
                    Log.w(TAG, e.toString());
                    mMetaInfo = null;
                }
            }
```

```
    }

    public int getSyncAction(Cursor c) {
        try {
            JSONObject noteInfo = null;
            if (mMetaInfo != null &&
mMetaInfo.has(GTaskStringUtils.META_HEAD_NOTE)) {
                noteInfo =
mMetaInfo.getJSONObject(GTaskStringUtils.META_HEAD_NOTE);
            }

            if (noteInfo == null) {
                Log.w(TAG, "it seems that note meta has been deleted");
                return SYNC_ACTION_UPDATE_REMOTE;
            }

            if (!noteInfo.has(NoteColumns.ID)) {
                Log.w(TAG, "remote note id seems to be deleted");
                return SYNC_ACTION_UPDATE_LOCAL;
            }

            // validate the note id now
            if (c.getLong(SqlNote.ID_COLUMN) != noteInfo.getLong(NoteColumns.ID))
{
                Log.w(TAG, "note id doesn't match");
                return SYNC_ACTION_UPDATE_LOCAL;
            }

            if (c.getInt(SqlNote.LOCAL_MODIFIED_COLUMN) == 0) {
                // there is no local update
                if (c.getLong(SqlNote.SYNC_ID_COLUMN) == getLastModified()) {
                    // no update both side
                    return SYNC_ACTION_NONE;
                } else {
                    // apply remote to local
                    return SYNC_ACTION_UPDATE_LOCAL;
                }
            } else {
                // validate gtask id
                if (!c.getString(SqlNote.GTASK_ID_COLUMN).equals(getGid())) {
                    Log.e(TAG, "gtask id doesn't match");
                    return SYNC_ACTION_ERROR;
                }
                if (c.getLong(SqlNote.SYNC_ID_COLUMN) == getLastModified()) {
```

```
                        // local modification only
                        return SYNC_ACTION_UPDATE_REMOTE;
                    } else {
                        return SYNC_ACTION_UPDATE_CONFLICT;
                    }
                }
            } catch (Exception e) {
                Log.e(TAG, e.toString());
                e.printStackTrace();
            }

            return SYNC_ACTION_ERROR;
        }

        public boolean isWorthSaving() {
            return mMetaInfo != null || (getName() != null && getName().trim().length() >
0)
                    || (getNotes() != null && getNotes().trim().length() > 0);
        }

        public void setCompleted(boolean completed) {
            this.mCompleted = completed;
        }

        public void setNotes(String notes) {
            this.mNotes = notes;
        }

        public void setPriorSibling(Task priorSibling) {
            this.mPriorSibling = priorSibling;
        }

        public void setParent(TaskList parent) {
            this.mParent = parent;
        }

        public boolean getCompleted() {
            return this.mCompleted;
        }

        public String getNotes() {
            return this.mNotes;
        }
```

```
    public Task getPriorSibling() {
        return this.mPriorSibling;
    }

    public TaskList getParent() {
        return this.mParent;
    }

}
```

（10）类名：TaskList

```
package net.micode.notes.gtask.data;

public class TaskList extends Node {
    private static final String TAG = TaskList.class.getSimpleName();//tag 标记

    private int mIndex;//当前 TaskList 的指针

    private ArrayList<Task> mChildren;//类中主要的保存数据的单元，用来实现一个以 Task 为元素的 ArrayList

    public TaskList() {
        super();
        mChildren = new ArrayList<Task>();
        mIndex = 1;
    }

    /* (non-Javadoc)
     * @see net.micode.notes.gtask.data.Node#getCreateAction(int)
     * 生成并返回一个包含了一定数据的 JSONObject 实体
     */
    public JSONObject getCreateAction(int actionId) {
        JSONObject js = new JSONObject();

        try {
            // action_type
            js.put(GTaskStringUtils.GTASK_JSON_ACTION_TYPE,
                    GTaskStringUtils.GTASK_JSON_ACTION_TYPE_CREATE);

            // action_id
            js.put(GTaskStringUtils.GTASK_JSON_ACTION_ID, actionId);

            // index
            js.put(GTaskStringUtils.GTASK_JSON_INDEX, mIndex);
```

```
                // entity_delta
                JSONObject entity = new JSONObject();//entity 实体
                entity.put(GTaskStringUtils.GTASK_JSON_NAME, getName());
                entity.put(GTaskStringUtils.GTASK_JSON_CREATOR_ID, "null");
                entity.put(GTaskStringUtils.GTASK_JSON_ENTITY_TYPE,
                        GTaskStringUtils.GTASK_JSON_TYPE_GROUP);
                js.put(GTaskStringUtils.GTASK_JSON_ENTITY_DELTA, entity);

            } catch (JSONException e) {
                Log.e(TAG, e.toString());
                e.printStackTrace();
                throw new ActionFailureException("fail to generate tasklist-create
jsonobject");
            }

            return js;
        }

        /* (non-Javadoc)
         * @see net.micode.notes.gtask.data.Node#getUpdateAction(int)
         * 生成并返回一个包含了一定数据的 JSONObject 实体
         */
        public JSONObject getUpdateAction(int actionId) {
            JSONObject js = new JSONObject();

            try {
                // action_type
                js.put(GTaskStringUtils.GTASK_JSON_ACTION_TYPE,
                        GTaskStringUtils.GTASK_JSON_ACTION_TYPE_UPDATE);

                // action_id
                js.put(GTaskStringUtils.GTASK_JSON_ACTION_ID, actionId);

                // id
                js.put(GTaskStringUtils.GTASK_JSON_ID, getGid());

                // entity_delta
                JSONObject entity = new JSONObject();
                entity.put(GTaskStringUtils.GTASK_JSON_NAME, getName());
                entity.put(GTaskStringUtils.GTASK_JSON_DELETED, getDeleted());
                js.put(GTaskStringUtils.GTASK_JSON_ENTITY_DELTA, entity);

            } catch (JSONException e) {
                Log.e(TAG, e.toString());
```

```
                e.printStackTrace();
                throw new ActionFailureException("fail to generate tasklist-update
jsonobject");
            }

            return js;
        }

        public void setContentByRemoteJSON(JSONObject js) {
            if (js != null) {
                try {
                    // id
                    if (js.has(GTaskStringUtils.GTASK_JSON_ID)) {
                        setGid(js.getString(GTaskStringUtils.GTASK_JSON_ID));
                    }

                    // last_modified
                    if (js.has(GTaskStringUtils.GTASK_JSON_LAST_MODIFIED)) {

setLastModified(js.getLong(GTaskStringUtils.GTASK_JSON_LAST_MODIFIED));
                    }

                    // name
                    if (js.has(GTaskStringUtils.GTASK_JSON_NAME)) {
                        setName(js.getString(GTaskStringUtils.GTASK_JSON_NAME));
                    }

                } catch (JSONException e) {
                    Log.e(TAG, e.toString());
                    e.printStackTrace();
                    throw new ActionFailureException("fail to get tasklist content from
jsonobject");
                }
            }
        }

        public void setContentByLocalJSON(JSONObject js) {
            if (js == null || !js.has(GTaskStringUtils.META_HEAD_NOTE)) {
                Log.w(TAG, "setContentByLocalJSON: nothing is avaiable");
            }

            try {
                JSONObject                    folder                    =
js.getJSONObject(GTaskStringUtils.META_HEAD_NOTE);
```

```
                    if (folder.getInt(NoteColumns.TYPE) == Notes.TYPE_FOLDER) {
                        String name = folder.getString(NoteColumns.SNIPPET);
                        setName(GTaskStringUtils.MIUI_FOLDER_PREFFIX + name);
                    } else if (folder.getInt(NoteColumns.TYPE) == Notes.TYPE_SYSTEM) {
                        if (folder.getLong(NoteColumns.ID) == Notes.ID_ROOT_FOLDER)
                            setName(GTaskStringUtils.MIUI_FOLDER_PREFFIX +
GTaskStringUtils.FOLDER_DEFAULT);
                        else if (folder.getLong(NoteColumns.ID) ==
Notes.ID_CALL_RECORD_FOLDER)
                            setName(GTaskStringUtils.MIUI_FOLDER_PREFFIX
                                    + GTaskStringUtils.FOLDER_CALL_NOTE);
                        else
                            Log.e(TAG, "invalid system folder");
                    } else {
                        Log.e(TAG, "error type");
                    }
                } catch (JSONException e) {
                    Log.e(TAG, e.toString());
                    e.printStackTrace();
                }
            }

            public JSONObject getLocalJSONFromContent() {
                try {
                    JSONObject js = new JSONObject();
                    JSONObject folder = new JSONObject();

                    String folderName = getName();
                    if (getName().startsWith(GTaskStringUtils.MIUI_FOLDER_PREFFIX))
                        folderName =
folderName.substring(GTaskStringUtils.MIUI_FOLDER_PREFFIX.length(),
                                folderName.length());
                    folder.put(NoteColumns.SNIPPET, folderName);
                    if (folderName.equals(GTaskStringUtils.FOLDER_DEFAULT)
                            || folderName.equals(GTaskStringUtils.FOLDER_CALL_NOTE))
                        folder.put(NoteColumns.TYPE, Notes.TYPE_SYSTEM);
                    else
                        folder.put(NoteColumns.TYPE, Notes.TYPE_FOLDER);

                    js.put(GTaskStringUtils.META_HEAD_NOTE, folder);

                    return js;
                } catch (JSONException e) {
```

```
            Log.e(TAG, e.toString());
            e.printStackTrace();
            return null;
        }
    }

    public int getSyncAction(Cursor c) {
        try {
            if (c.getInt(SqlNote.LOCAL_MODIFIED_COLUMN) == 0) {
                // there is no local update
                if (c.getLong(SqlNote.SYNC_ID_COLUMN) == getLastModified()) {
                    // no update both side
                    return SYNC_ACTION_NONE;
                } else {
                    // apply remote to local
                    return SYNC_ACTION_UPDATE_LOCAL;
                }
            } else {
                // validate gtask id
                if (!c.getString(SqlNote.GTASK_ID_COLUMN).equals(getGid())) {
                    Log.e(TAG, "gtask id doesn't match");
                    return SYNC_ACTION_ERROR;
                }
                if (c.getLong(SqlNote.SYNC_ID_COLUMN) == getLastModified()) {
                    // local modification only
                    return SYNC_ACTION_UPDATE_REMOTE;
                } else {
                    // for folder conflicts, just apply local modification
                    return SYNC_ACTION_UPDATE_REMOTE;
                }
            }
        } catch (Exception e) {
            Log.e(TAG, e.toString());
            e.printStackTrace();
        }

        return SYNC_ACTION_ERROR;
    }

    /**
     * @return
     * 功能：获得 TaskList 的大小，即 mChildren 的大小
     */
    public int getChildTaskCount() {
```

```
        return mChildren.size();
    }

    /**
     * @param task
     * @return  返回值为是否成功添加任务。
     *  功能：在当前任务表末尾添加新的任务。
     */
    public boolean addChildTask(Task task) {
        boolean ret = false;
        if (task != null && !mChildren.contains(task)) {
            ret = mChildren.add(task);
            if (ret) {
                // need to set prior sibling and parent
                task.setPriorSibling(mChildren.isEmpty() ? null : mChildren
                        .get(mChildren.size() - 1));
                task.setParent(this);
                //注意：每一次 ArrayList 的变化都要紧跟相关 Task 中 PriorSibling
的更改
                //，接下来几个函数都有相关操作
            }
        }
        return ret;
    }

    /**
     * @param task
     * @param index
     * @return
     *  功能：在当前任务表的指定位置添加新的任务。
     */
    public boolean addChildTask(Task task, int index) {
        if (index < 0 || index > mChildren.size()) {
            Log.e(TAG, "add child task: invalid index");
            return false;
        }

        int pos = mChildren.indexOf(task);
        if (task != null && pos == -1) {
            mChildren.add(index, task);

            // update the task list
            Task preTask = null;
            Task afterTask = null;
```

```
            if (index != 0)
                preTask = mChildren.get(index - 1);
            if (index != mChildren.size() - 1)
                afterTask = mChildren.get(index + 1);

            task.setPriorSibling(preTask);
            if (afterTask != null)
                afterTask.setPriorSibling(task);
        }

        return true;
    }

    /**
     * @param task
     * @return  返回删除是否成功
     *  功能：删除 TaskList 中的一个 Task
     */
    public boolean removeChildTask(Task task) {
        boolean ret = false;
        int index = mChildren.indexOf(task);
        if (index != -1) {
            ret = mChildren.remove(task);

            if (ret) {
                // reset prior sibling and parent
                task.setPriorSibling(null);
                task.setParent(null);

                // update the task list
                if (index != mChildren.size()) {
                    mChildren.get(index).setPriorSibling(
                            index == 0 ? null : mChildren.get(index - 1));
                }
            }
        }
        return ret;
    }

    /**
     * @param task
     * @param index
     * @return
     *  功能：将当前 TaskList 中含有的某个 Task 移到 index 位置
```

```
 */
public boolean moveChildTask(Task task, int index) {

    if (index < 0 || index >= mChildren.size()) {
        Log.e(TAG, "move child task: invalid index");
        return false;
    }

    int pos = mChildren.indexOf(task);
    if (pos == -1) {
        Log.e(TAG, "move child task: the task should in the list");
        return false;
    }

    if (pos == index)
        return true;
    return (removeChildTask(task) && addChildTask(task, index));
    //利用已实现好的功能完成当下功能；
}

/**
 * @param gid
 * @return 返回寻找结果
 * 功能：按 gid 寻找 Task
 */
public Task findChildTaskByGid(String gid) {
    for (int i = 0; i < mChildren.size(); i++) {
        Task t = mChildren.get(i);
        if (t.getGid().equals(gid)) {
            return t;
        }
    }
    return null;
}

/**
 * @param task
 * @return
 * 功能：返回指定 Task 的 index
 */
public int getChildTaskIndex(Task task) {
    return mChildren.indexOf(task);
}
```

```
    /**
     * @param index
     * @return
     * 功能：返回指定 index 的 Task
     */
    public Task getChildTaskByIndex(int index) {
        if (index < 0 || index >= mChildren.size()) {
            Log.e(TAG, "getTaskByIndex: invalid index");
            return null;
        }
        return mChildren.get(index);
    }

    /**
     * @param gid
     * @return
     * 功能：返回指定 gid 的 Task
     */
    public Task getChilTaskByGid(String gid) {
        for (Task task : mChildren) {//一种常见的 ArrayList 的遍历方法（四种，见精读笔记）
            if (task.getGid().equals(gid))
                return task;
        }
        return null;
    }

    public ArrayList<Task> getChildTaskList() {
        return this.mChildren;
    }

    public void setIndex(int index) {
        this.mIndex = index;
    }

    public int getIndex() {
        return this.mIndex;
    }
}
```

（7）