[SCA] Driver App 数据库框架迁移方案

一、背景及目的

1.1 背景：

随着 SSC 各 Android 项目的快速发展，对数据库的应用越来越多，早期各项目使用的数据库 ORM 框架各异，且大多都已停止维护，目前组内已基于 Room 封装了一套统一的数据库 ORM 框架，部分项目已经进行了迁移，目前 SPX Driver App 还是在使用 LiteOrm 框架，由于 Driver App 对数据库的使用场景较多较复杂，迁移到统一的数据库框架会遇到以下问题：

- 目前统一的数据库框架拷贝 Room 源码并修改包名，无法快速 follow Room 的升级以及 bugfix
- Room 与 LiteOrm 框架的数据库访问 API 差异较大
- 从 LiteOrm 框架的自适应升级切换到 Room 的自动迁移，会带来很大的版本管理负担

为什么会带来很大的版本管理负担？其他项目有没有这个问题？

引起该问题的主要原因是：Driver App 的大部分需求在开发阶段都不能确定上线时间，只能确定 UAT 版本。

在传统的项目迭代中，每个大版本的具体需求都能确定，即使两个大版本并行开发，也能通过版本发布顺序确定隐性的依赖关系，数据库版本的增长非常自然，一般不会有冲突。而目前 Driver App 的大部分需求在不能确定上线时间的情况下，各需求分支相对独立，且各需求上 test、uat、release 的节奏各异，各分支打出的 apk 包存在混装情况，目前 LiteOrm 框架的自适应升级可以兼容这些复杂场景。如果切换到 Room 的自动迁移，各公共分支（如 test、uat、release）都需要管理一套独立的数据库版本，且同时需要独立各分支数据库（不同公共分支取不同的 DB name）或者独立各分支 apk 包（不同公共分支取不同的 applicationId），才能保证混装的不同分支不同数据库版本的 apk 能够正常运行。

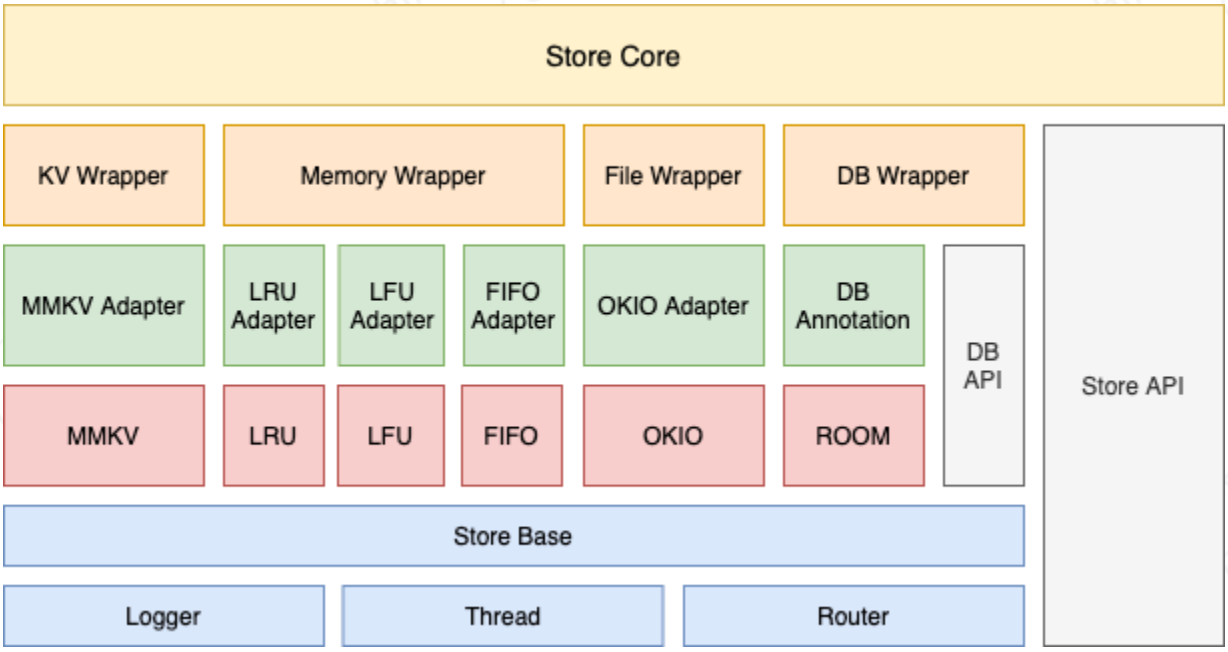
其他项目如果也是需求不能确定上线版本，且数据库改动频繁，也会遇到这个问题。

1.2 目的：

- 修改数据库框架实现，使其可以快速 follow Room 的升级以及 bugfix
- 在 Room 的基础上封装与 LiteOrm 框架相似的数据库访问 API，项目切换时只需修改少量代码
- 在 Room 的自动迁移的基础上，新增自适应升级能力，项目中的绝大部分数据库改动可以走自适应升级，少部分特殊修改走自动迁移
- 将 Driver App 的数据库框架由 LiteOrm 框架迁移至统一的基于 Room 的数据库框架

二、逻辑架构设计

整体架构和 存储框架设计文档 中的一致：



改动的地方有：

- DB Annotation 对应 corelib store-db-annotation，为了做依赖隔离进行了包名替换，这里直接移除，项目修改回 room-common
- DB API 对应 corelib store-db-api，在原来的基础上新增与 LiteOrm 框架相似的数据库访问 API，在实现模块 store-db 中增加自适应升级能力
- 移除 corelib store-db-room 目录下的 room 模块，新增 store-db-room-origin:room-compiler 对 room-compiler 进行功能扩展

三、核心逻辑详细设计

3.1 新增与 LiteOrm 框架相似的数据库访问 API

先看下 Room 与 LiteOrm 框架在注解和 API 上的差异：

常用注解差异：

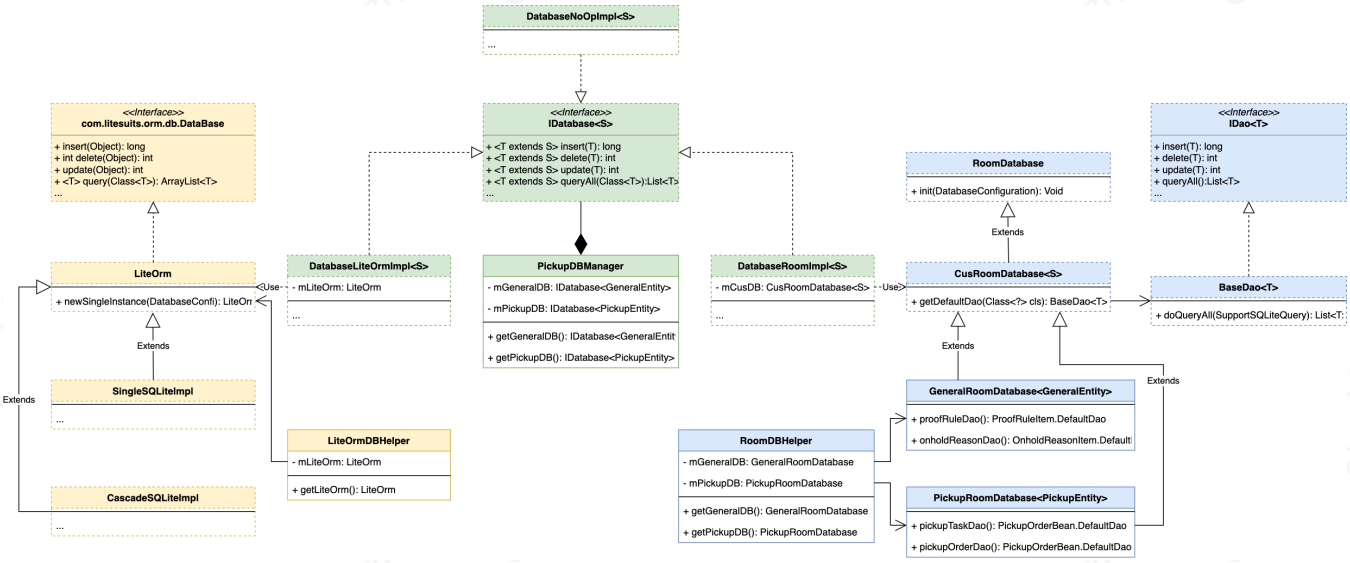
注解用途	LiteOrm 注解	LiteOrm 注解属性	Room 注解	Room 注解属性	差异
实体类/表	Table	表名: String value()	Entity	表名: String tableName()	无差异
实体类属性/列	PrimaryKey	是否自增: AssignType value()	PrimaryKey	是否自增: boolean autoGenerate()	无差异
实体类属性/列	Column	列名: String value()	ColumnInfo	列名: String name()	无差异
实体类属性/列	NotNull	是否非空	NonNull	是否非空	Room 基础类型默认非空，其他类型默认可空
实体类属性/列	Unique	唯一性约束	Entity - Index	唯一索引: boolean unique()	Room 目前只能通过增加唯一索引来对普通列增加唯一性约束
实体类属性/列	Ignore	忽视字段	Ignore	忽视字段	无差异
实体类属性/列	Default	默认值: String value()	ColumnInfo	默认值: String defaultValue()	无差异
冲突策略	Conflict	冲突策略: Strategy value()	Insert /Update	冲突策略: OnConflictStrategy value()	LiteOrm 的冲突策略注解在实体类字段上，Room 的冲突策略注解在 dao 的 Insert/Update 方法上

从上表可以看出，Room 与 LiteOrm 框架在常用注解上的差异不大，需要注意的是：

1. Room 对字段的唯一性约束需要通过增加索引，这将对自适应升级提出更多要求；
2. Room 注解的保留策略是 class 级别，即运行时无法获取注解信息，如果自适应升级需要拿到实体类对应的表结构信息，有以下几种方式：
    a. fork Room 的注解模块（room-common），修改相关注解的保留策略为 Runtime，以相同的 group_id:artifact_id + 特定版本发布至内部 maven，项目中锁定 room-common 为特定版本即可；
    b. fork Room 的 APT 模块，修改生成 RoomDatabase 实现类代码的部分，将所有有注解的表结构生成至特定区域可供外部查找，发布至内部 maven，项目中使用特定版本的 Room APT；
    c. 自行实现一个简单的 APT，将所有有注解的表结构生成至特定区域可供外部查找，发布至内部 maven，项目中新增该 APT；

综合来看，b 方式最简单易行，同时由于 Room 的注解处理模块变更极少，后续的更新维护成本也较低。

API 及类层次差异（兼统一 API 类图）：



从上图可以看出，LiteOrm 框架通过构造一个 LiteOrm 实例即可对任意类型的实体进行存取操作；而 Room 的数据库类需要依赖其中所有表的实体类，同时增加了 Dao 层，这将带来两个问题：

1. 目前 Driver App 所有模块共用一个数据库如何解决?
2. 如何设计泛型可以让统一 API 即拥有 LiteOrm 的灵活性，又不失编译时约束？

答案也能从上图中得出：

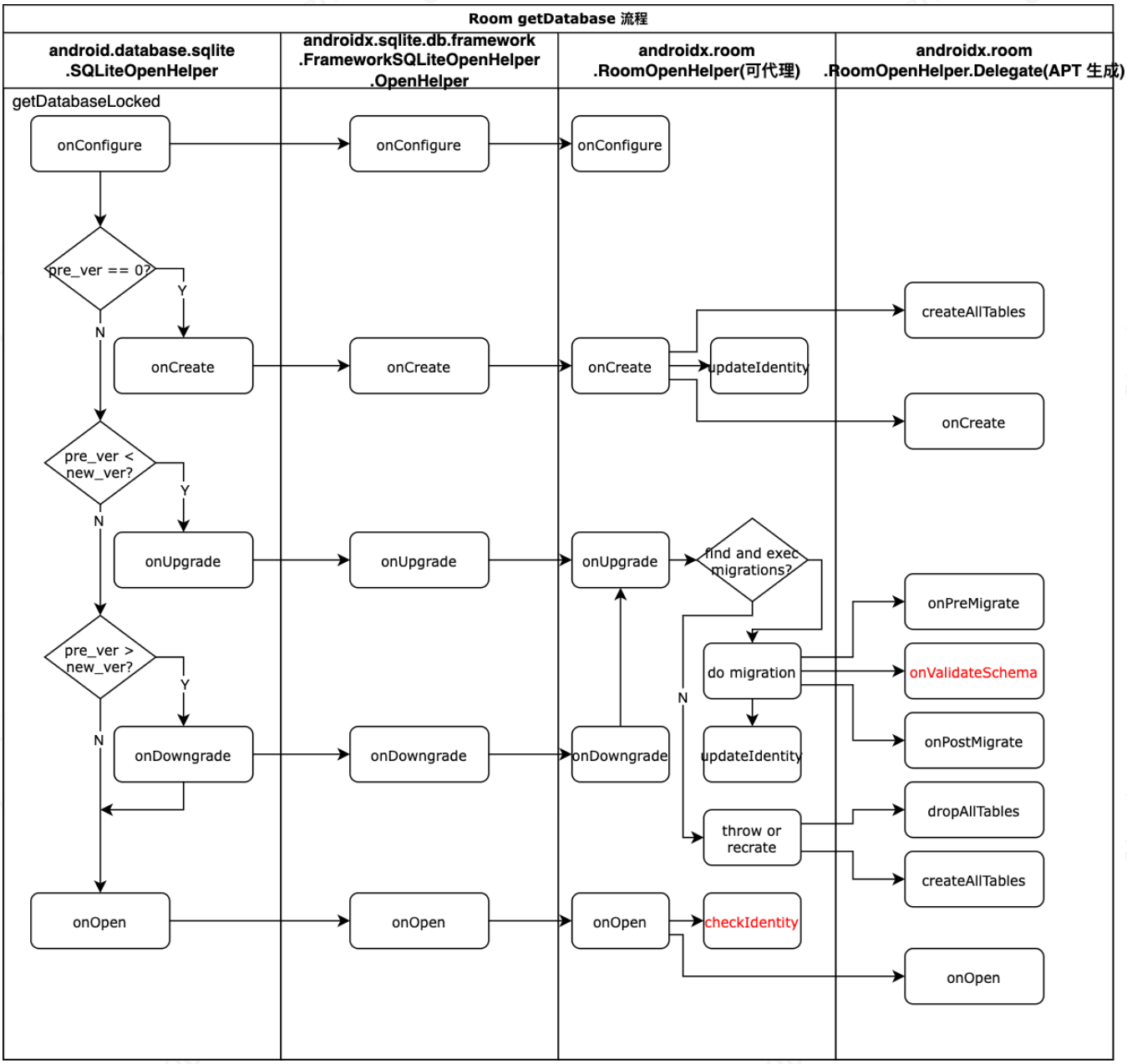
1. 每个模块都维护一套 OrmDBManager/DBHelper，就和现在网络访问的 ServiceManager 相似，如 pickup 模块维护 PickupOrmDBManager；（也可以将所有实体类下沉至 base 模块，这样可以保持一个数据库，但实体类的界限将变模糊）
2. 对于统一的数据库访问接口 IDatabase，增加泛型 S，限制数据库可操作实体类的类型上限，如 pickup 模块定义空接口 PickupEntity，需要存储在 pickup 数据库中的实体类都实现该接口。因 LiteOrm 框架可操作的类型比限制广，LiteOrm 框架忽视该泛型即可。在 Room 的设计中，虽然一个 Dao 类并不需要对应一个实体类，但 Room 的 APT 支持对 Dao 父类的部分泛型抽象方法生成实现代码，如此可设计一套类似 LiteOrm DataBase 接口的 Dao 层公共接口 IDao<T>，每个实体类都构造一个空的默认抽象 Dao 类继承 BaseDao<T> 提供泛型，然后在每个抽象数据库类中（CusRoomDatabase 子类），实现实体类与默认 Dao 的映射即可。

综上，根据目前项目中使用到的 LiteOrm API，提取统一的数据库访问 API 接口类 IOrmDatabase，并提供基于 LiteOrm 的实现 OrmDatabasLiteOrmImpl，将项目中的使用都切换到接口调用；再提供基于 Room 的实现 OrmDatabasRoomImpl，通过切换 IOrmDatabase 接口的实现以及切换相关注解，即可快速切换数据库框架。

3.2 新增自适应升级能力，兼容自动迁移

目前 LiteOrm 的自适应升级主要包含自动建表以及自动加列能力，可以覆盖 Driver App 的绝大部分数据库改动需求，基于此，对 Room 新增的自适应升级也需包含这两个能力。其他的数据库改动，如新增外键、索引等，也可以通过自适应升级来处理，只是实现上会更为复杂。

Room 的 getDatabase 流程如下：



其中 onValidateSchema 和 checkIdentity 方法都会对数据库模式进行校验，当数据库中的实体类发生变化但并未修改数据库版本号或者未提供迁移实现时，这两个方法都可能抛出异常。

经过调研，可以对 androidx.room.RoomOpenHelper 进行代理，在其 onOpen 之前移除 room_master_table，即可跳过 identity 相关校验；在 onUpgrade 或者 onOpen 前插入自适应升级处理，使数据库中各表的 schema 与代码中的 schema 一致，则后续的 onValidateSchema 也可以正常通过。即保留了自适应升级的灵活，又通过了 Room 的严格校验。

自适应升级的稳定性不言而喻，通过以下方式来保证稳定性：

- 自测 demo 覆盖足够复杂的场景，远超当前项目所需
  - 删表、列，修改列（列名不变，字段变化）
  - 增删外键
  - 增删索引
  - 增删嵌套字段
  - 继承获得的列、外键、索引
- 增加自适应升级失败的降级策略
  - 缓存数据库，删表重建
  - 重要数据库，切回 LiteOrm 框架

四、接口设计

IDatabase 接口设计如下：

```
/**
 * Database common Api.
 */
public interface IDatabase<S> {

    /**
     * Insert an entity into the database.
     *
     * @return row ID of newly inserted entity if successful. -1 otherwise.
     */
    <T extends S> long insert(T entity);

    /**
     * Insert all entities into the database.
     *
     * @return total success count if successful. -1 otherwise.
     */
    <T extends S> int insert(Collection<T> entities);

    /**
     * Insert an entity into the database.
     *
     * @return row ID of newly inserted entity if successful. -1 otherwise.
     */
    <T extends S> long insertOrReplace(T entity);

    /**
     * Insert all entities into the database.
     *
     * @return total success count if successful. -1 otherwise.
     */
    <T extends S> int insertOrReplace(Collection<T> entities);


    <T extends S> int deleteById(Class<T> entityClass, String id);

    <T extends S> int delete(T entity);

    <T extends S> int delete(Collection<T> entities);

    <T extends S> int deleteAll(Class<T> entityClass);


    <T extends S> int update(T entity);

    <T extends S> int update(Collection<T> entities);


    <T extends S> T queryById(Class<T> entityClass, String id);

    <T extends S> T queryById(Class<T> entityClass, long id);

    <T extends S> List<T> queryAll(Class<T> entityClass);

    <T extends S> List<T> queryRaw(Class<T> entityClass, String where, String... selectionArgs);

    /**
     * query count of table rows and return
     *
     * @return the count of query result
     */
    <T extends S> long queryCount(Class<T> entityClass);

}
```

五、存储设计

无

六、外部依赖与限制

room 2.4.2

七、框架迁移实施计划

1. sdk 完成的同时，输出自测 demo，覆盖足够复杂的场景 - 0311
2. 先对部分不重要的缓存数据库进行数据迁移及框架切换，通过监控观察触发降级策略的场景，针对修复 - 0314 - 0325
3. 新业务场景使用新框架，持续监控及优化
4. 对所有数据进行数据迁移及框架切换，持续监控及优化

八、风险

1. 项目分支复杂，某些未知场景下自适应升级可能会失败，如果不进行处理则会进一步触发 Room 的 crash 或者导致 DB 不可用
2. 系统异常时（如存储空间不足），自适应升级或自动迁移可能会失败

以上风险在不增加自适应升级的情况下也会发生，可以通过增加降级策略，保证数据不丢失。