存储框架使用文档

1. 背景

整合所有存储到一个库中，方便切换存储方式。存储方式有：内存数据存储、kv存储（SharedPreference）、文件存储和数据库DB存储。

2. 功能

功能点	功能描述
内存存储	支持缓存回收机制，可选LRU、LFU、FIFO。
KV存储	支持数据迁移，默认使用MMKV框架
文件存储	I/O方式可选，目前默认使用Okio
DB存储	数据库框架可选，目前默认使用ROOM

3. 使用介绍

demo工程：corelib工程中的:store-framework:store-sample

3.1 依赖引入

```
//
implementation 'com.shopee.sc:store-core:{lastReleaseVersion}'
//
runtimeOnly 'com.shopee.sc:store-file:{lastReleaseVersion}'
runtimeOnly 'com.shopee.sc:store-kv:{lastReleaseVersion}'
runtimeOnly 'com.shopee.sc:store-memory:{lastReleaseVersion}'
runtimeOnly 'com.shopee.sc:store-db:{lastReleaseVersion}'
//
implementation 'com.shopee.sc:store-db-room-runtime:{lastReleaseVersion}'
annotationProcessor 'com.shopee.sc:store-db-room-compiler:{lastReleaseVersion}'
```

`store-coreruntimeOnly`

3.2 使用

3.2.1 初始化

在Application初始化时调用

```
StoreFrameworkCore.init(context);
```

3.2.2 基本使用

```
StoreFrameworkCore.of(StoreType)
```

根据不同的存储方式获取StoreApi对应的实现，再调用StoreApi的接口实现存储操作。

3.2.3 通用的传入、返回参数

统一的入参为StoreParam，相关参数变量如下

```
public class StoreParam {
    /**
     *
     * {@link com.shopee.sc.store.api.constant.StoreStrategy}
     */
    private String mStrategy;
    /**
     * IDID
     * KVKVID
     * IDIDLRU
     */
    private String mId;

    /**
     * String FileID
     * KVStringKey
     * Key
     */
    private Object mKey;
    /**
     * putwritegetread
     * Stringbyte[]
     */
    private Object mValue;
    /**
     *
     */
    private Lifecycle mLifecycle;

        ...
}
```

mStrategy对应于各个存储方式的策略，如内存方式的LRU、LFU等，传空则使用默认的策略。其余的参数根据不同的存储方式有不同的使用，mLifecycle只在异步操作或者内存操作可选择传入。

StoreParam还提供了一些静态方法，根据使用场景返回相应的StoreParam对象，包括forKv，forFile，forFile等方法。

统一的返回参数StoreResult，数据是一个泛型，还包含一个返回码和返回信息，返回码见ResultCode。

3.2.4 内存存储使用

内存存储可以看成提供线程安全的容器，不同的容器依靠id来区分，同一个id对应的容器只支持一种缓存策略——LRU、LFU、FIFO。容器按照Key、Value来存储数据，两者支持各种类型（除基础类型）。可以传入LifeCycle，在onDestroy时清除缓存；若不传入，则默认外部自行管理缓存生命周期。

使用内存缓存分两步，先在适当时机（只需要调一次）通过id初始化容器，同时指定好Key、Value的类型、缓存策略、缓存容量等；然后就可以根据id在各个场景，get或put数据了。

```
//
//
StoreParam param = new StoreParam.Builder().id("id").key("key").value(1).build();
StoreParam param = StoreParam.forMemory("id", "key", 2, null, StoreStrategy.MemoryStrategy.LFU);
//
StoreFrameworkCore.of(StoreType.MEMORY).buildMemCache(param, 100);

//
StoreParam param = new StoreParam.Builder()
                                            .id("id")
                                            .key("key1")
                                            .value(66)
                                            .build();
StoreFrameworkCore.of(StoreType.MEMORY).put(param);

//
StoreParam param = new StoreParam.Builder()
                                            .id("id")
                                            .key("key1")
                                            .value(0) // default value
                                            .build();
StoreResult r = StoreFrameworkCore.of(StoreType.MEMORY).get(param);

//
StoreFrameworkCore.of(StoreType.MEMORY).clear("id");

// id
StoreFrameworkCore.of(StoreType.MEMORY).destroy("id");
```

3.2.5 KV存储使用

KV存储的使用可以参考SharedPreferences，其中StoreParam的id参数对应于SP的name，对应MMKV的id。SP数据迁移调用importFromSharePreferences方法，可以多次重复调用，因为若旧数据为空则不会迁移，在第一次成功迁移数据后会将旧数据文件清空。

```
// sp
SharedPreferences sp = StoreFrameworkCore.of(StoreType.KV).getSharedPreferences("sp_name");

//
StoreParam param = new StoreParam.Builder().key("key1").value("abc").build(); // id
StoreParam param = StoreParam.forKv("sp_name", "key", "value", StoreStrategy.KvStrategy.MMKV);
StoreFrameworkCore.of(StoreType.KV).put(param);

//
StoreParam param = new StoreParam.Builder()
                                            .id("id")
                                            .key("key1")
                                            .value(0) // default value
                                            .build();
StoreResult r = StoreFrameworkCore.of(StoreType.KV).get(param);

// SPid
StoreFrameworkCore.of(StoreType.KV).importFromSharePreferences(spName, spName);

// id
StoreFrameworkCore.of(StoreType.KV).clear(id);
```

3.2.6 文件存储使用

文件存储时，可以指定绝对路径定位具体文件并操作，如果不关心文件路径，则可以通过StoreParam的id来指定不同文件。StoreParam的key，可以传入String类型的绝对路径，也可以传入File类型的文件对象。StoreParam的value作为文件的内容，可以传入String类型或byte数组类型。对于异步操作，需传入LifeCycle，并通过StoreLoadCallback回调操作结果。同步操作需在子线程中调用。

```
//
StoreParam param = new StoreParam.Builder()
                                                .id("dd")
                                                .value("content")
                                                .setLifecycle(getLifecycle())
                                                .build();
StoreFrameworkCore.of(StoreType.FILE).asyncWriteFile(param, new StoreLoadCallback<byte[]>() {
        @Override
        public void onLoadFinish(StoreResult<byte[]> result) {
        }
});

//
StoreParam param = new StoreParam.Builder()
                                                .id("dd")
                                                .value("null") // default content
                                                .setLifecycle(getLifecycle())
                                                .build();
StoreFrameworkCore.of(StoreType.FILE).asyncReadFile(param, new StoreLoadCallback<byte[]>() {
        @Override
        public void onLoadFinish(StoreResult<byte[]> result) {
        }
});
```

3.2.7 DB存储使用

DB的使用步骤：

1. 定义实体类
2. 定义DAO
3. 定义数据库
4. 创建数据库

1、定义实体类

```
@Entity
public class UserInfo {
    @PrimaryKey
    private long uid;
    @ColumnInfo(name = "first_name")
    private String firstName;
    @ColumnInfo(name = "last_name")
    private String lastName;

    //  set/get
}
```

2DAO

```
/**
 * BaseDao
 */
@Dao
public abstract class UserInfoDao extends BaseDao<UserInfo> {

}
```

3

```
/**
 *  StoreDatabase
 */
@Database(entities = {
        UserInfo.class}, //
        version = 1, //
        exportSchema = false)
public interface AppDatabase extends StoreDatabase {
    UserInfoDao userDao();
}
```

4

```
/**
 *
 */
public class DbHelper {

    /**
     *
     */
    private final static String DB_NAME_STORE = "StoreDatabase";

    private volatile static AppDatabase sAppDatabase;

    public static AppDatabase getAppDatabase() {
        if (sAppDatabase == null) {
            synchronized (AppDatabase.class) {
                if (sAppDatabase == null) {
                    sAppDatabase = createAppDatabase();
                }
            }
        }
        return sAppDatabase;
    }

    private static AppDatabase createAppDatabase() {
        DatabaseConfig databaseConfig = new DatabaseConfig.Builder()
                .setDatabaseClass(AppDatabase.class)
                .build();
        StoreResult<AppDatabase> storeResult =
                StoreFrameworkCore.of(StoreType.DB).getDatabase(DB_NAME_STORE, databaseConfig);
        if (storeResult.isStrictSuccess()) {
            return storeResult.data;
        } else {
            Log.e("DbHelper", storeResult.message);
            return null;
        }
    }
}
```

5、数据库升级

创建升级版本的 SupportMigration 编写升级SQL语句即可。

```
List<SupportMigration> supportMigrationList = new ArrayList<>();
supportMigrationList.add(new SupportMigration(1, 2) {
    @Override
    public void migrate(@NonNull SupportSQLiteDatabase database) {
        database.execSQL("ALTER TABLE 'user' ADD COLUMN 'age' INTEGER NOT NULL DEFAULT 0");
    }
});
DatabaseConfig databaseConfig = new DatabaseConfig.Builder()
        .setDatabaseClass(AppDatabase.class)
        .setSupportMigrationList(supportMigrationList)
        .build();
```

4. 混淆配置

```
#
-keep class * implements com.shopee.sc.store.base.router.processor.IProvider
```