存储框架设计文档

版本	修订日期	修订者	修订内容
v1.0	2021.05.18	tao.yangyt@shopee.com jianwei.luo@shopee.com zheng.zeng@shopee.com	存储框架初次设计

1、需求概况

1.1 需求背景

App数据按照存储来分，大概会分为4类，第一是保存在内存中的数据，第二是保存在SharedPreferences中数据，第三是保存在文件中的数据，第四是保存在数据库中。

目前，应用中对于这几种存储的使用，都是直接引入对应框架（直接使用SharedPreferences或GreenDao等）。这种使用方式存在以下弊端：

- 对代码的侵入性较强，如果需要替换框架可能需要大量的工作量。
- 四种类型的存储，都是独立使用，不能统一管理与配置。
- 新项目需要接入上述框架，会增加很多重复工作。

因此我们需要一个统一的数据容器，使上层应用忽略掉这些存储类型的差异，并且对各种使用的框架进行统一管理和解耦。

1.2 需求目的

我们希望通过这个通用存储框架达到以下目的：

- **精简代码。**只使用一个通用的数据容器，就可以让程序通过简单的调用实现内存、文件、KV和数据库的存储，使应用层的代码更简洁。
- **灵活地扩展和优化。**上层应用也无需关注内部实现，即便需要替换使用框架，也可以做到业务无感知。
- **降低数据存储的接入成本。**让开发者从持久化方案选型和处理中解脱出来，可以把更多的开发重心放在业务逻辑中，减少开发者的重复性工作。

1.3 需求功能

调研文档：存储框架调研文档

1）分别实现4种存储方式的封装，方便替换底层实现框架。

2）外部根据需要使用同步或异步的方式，调用任意种类的存储方式，包括对数据的增删查改。

3）KV方式支持数据迁移。

2、技术方案设计

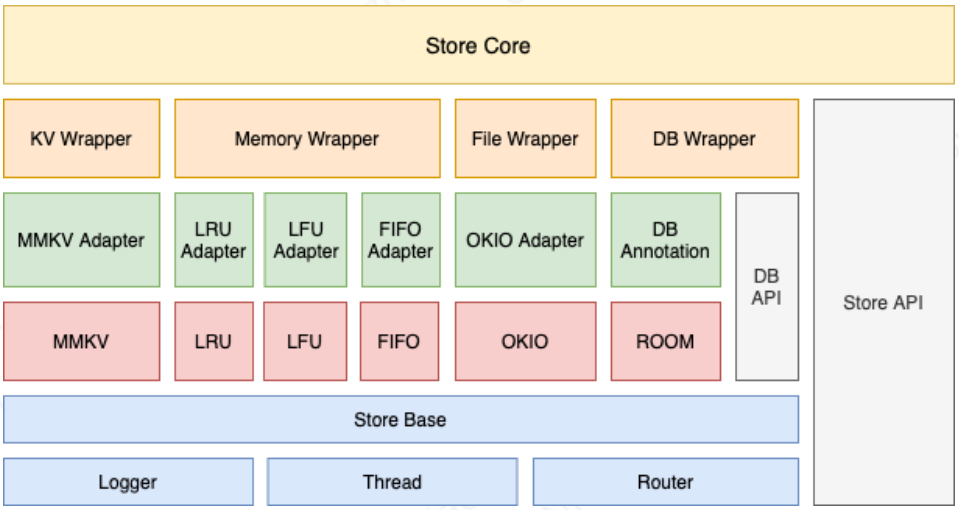
补充选型原因

补充接口设计，包装层的具体设计

线程管理

2.1 整体方案设计

使用组件化实现各个模块，4种数据存储方式对应4个组件，还有一个总的基础组件，负责调用各个组件及管理基础功能，作为外部的调用组件，根据需求依赖底层组件。



每个底层组件分为包装层和实现层，在包装层自定义一套自己的实现框架，与实际的底层实现框架独立开，方便无感知地替换底层实现框架。

基础组件管理线程异步回调，以及各组件的初始化，并提供接口给外部调用。

2.2 详细方案设计

2.2.1 基础组件

包含对外提供调用方法的核心组件，以及所有组件都可以依赖的公共组件两个组件。

核心组件对各组件的Wrapper 层提供的方法，进行统一的封装。输入数据和返回结果都可以可做一层统一包装，输入参数可以考虑支持各种数据类型；返回结果考虑各种错误类型的返回结果，成功则带回正常返回结果，失败则返回错误类型，如IO异常、空间不足等。

初始化模块，根据外部依赖，只初始化已依赖的组件。判断是否依赖某个组件，可以利用反射Class.forName，看能否取得组件的关键类。

公共组件包含线程管理以及管理返回错误码。

线程管理，用线程池管理线程。对于异步回调，传入Callback的同时还可以传入Lifecycle，管理两者Map关系，在页面销毁时销毁页面Lifecycle对应的Callback，防止不必要的回调。

2.2.2 内存存储组件

针对内存部分，会提供LRU、FIFO、LFU三种淘汰策略，默认使用LRU策略，但是声明内存时可以自定义使用其他两种策略。

- LRU策略内存实现：官方提供的LruCahce来实现内存框架，LRUCache天然支持LRU算法，接入成本较低。LRUCache使用的主体数据结构是LinkedHashMap，在初始化时将LinkedHashMap的accessOrder设为true，即按照节点访问顺序排序，所以可以达到LRU的效果。
- FIFO策略内存实现：参考LruCahce的实现，当把LinkedHashMap初始化参数中accessOrder设为false，即是按照节点插入顺序进行排序，可以达到FIFO的效果。
- LFU策略内存实现：使用PriorityQueue和HashMap实现，每次调用节点会自动增加节点的frequency，并且自定义优先队列中节点的compareTo方法为按照frequency排序。

在包装层对外提供build方法，根据id创建cache，创建后的cache实体会存入map中，后续可以直接通过id对对应的cache进行get、put、remove、clear、snapshot、size等操作。

对内存的销毁提供两种方式：1、可以在Activity销毁时手动调用destroy方法，将用到的cache从统一管理的map中删除；2、创建cache时传入lifecycle，这样不需手动调用destroy，可以实现自动监听Activity销毁触发destroy的功能。

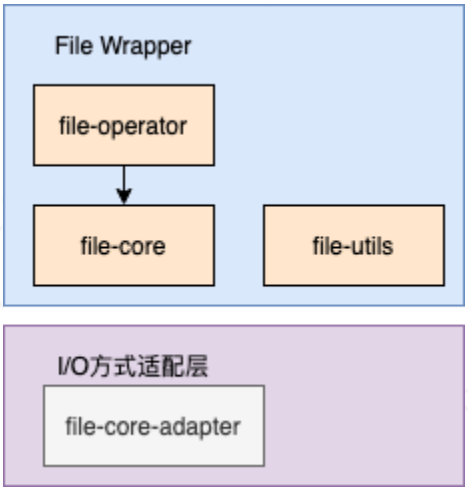
后续可以拓展支持多级缓存。

2.2.3 文件存储组件

2.2.3.1 架构

支持的功能为，保存数据流到特定文件，输入路径和数据流，或者不关心路径时可输入Key和数据流，自动生成Key对应的默认路径。

组件架构：



把模块整体分成两层：

1. **File Wrapper 层**：对外提供文件相关操作的方法，屏蔽I/O方式的具体实现。
2. **I/O方式适配层**：根据不同的I/O实现方式进行不同进行适配。

file-core提供适配目标的接口定义，接口包括对文件的读写操作。file-operator是对外提供的类，实现了file-core的接口，实际上是操作适配目标来实现文件的读写。file-utils是文件相关的工具。

file-core-adapter是适配器模式的适配器，不同的I/O方式对应不同的适配器。若要新增I/O实现方式，只需要新增对应的适配器，外部通过参数控制使用不同的适配器就能相应地使用不同的I/O实现方式。

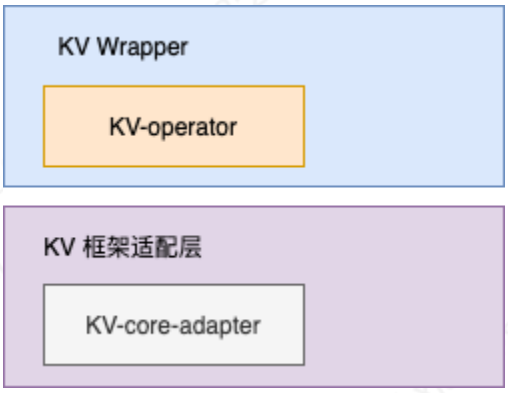
例如Okio会使用Sink和Source来管理读写流，所以需要在Okio的adapter层将基础的OutputStream、InputStream、byte[]数据类型做一下转换。

另外，在读写文件时首先要检查读写权限，若无权限需要申请权限。写文件时，要判断存储空间是否足够。

架构方式，用适配器模式适配不同的IO方式，如okio、mmap等，用工厂模式提供不同的适配器供外部调用。

2.2.4 KV存储组件

KV部分会通过封装MMKV实现。由于MMKV底层基于MMAP内存映射，操作内存就相当于操作文件，所以不需要做另外开启线程的工作。但是原先App中的KV存储在SharePreferences中，所以需要关注数据迁移的实现。



把模块整体分成两层：

1. **KV Wrapper 层**：对外提供KV相关操作的方法，包括init、put、get、clear等基础方法和importFromSP数据迁移方法。
2. **KV框架适配层**：根据使用的KV框架进行Wrapper层方法的封装与转化。

KV Wrapper层对外提供以下接口：

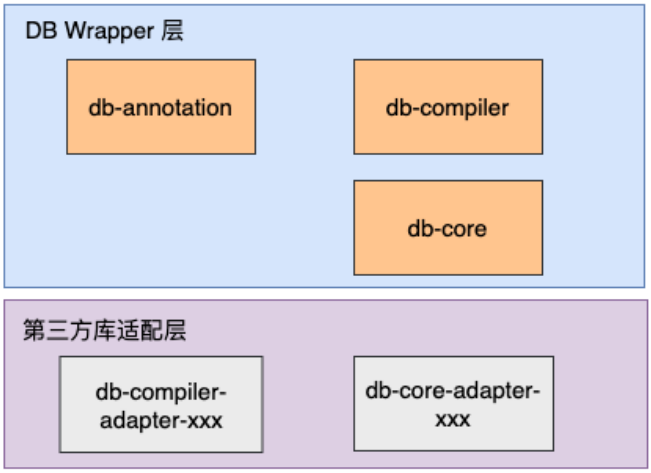
- init：初始化
- put (id，key，value)：往特定ID的MMKV中写入数据
- get (id，key，defualtValue)：从特定ID的MMKV读取数据
- clear (id)：清楚特定ID的MMKV中的所有数据
- importFromSP(spID, mmkvID)：定向迁移SP的数据到特定ID的MMKV

后续拓展性：

MMKV已经具备从SharePreference直接迁移数据的方法，但是以后如果采用其他KV框架替换MMKV时，需要自己实现从MMKV到新框架的数据迁移方法。

2.2.5 数据库存储组件

组件架构：



把模块整体分成两层：

1. **DB Wrapper 层**：对外提供注解和接口方法，屏蔽第三方库的具体实现。
2. **第三数据库框架适配层**：根据地方库API的不同进行适配

2.2.5.1 db-annotation

现在主流的 Android 数据库框架都是通过提供注解来定义数据库、表和DAO等，主要是为了简化模板代码，所以 Wrapper 层需要提供对应的注解进行映射。该模块包含下面常用的注解:

- @Database：创建数据库
- @Entity：数据库中的表
    - @PrimaryKey：主键
    - @ColumnInfo：字段信息
    - @Ignore：忽略的变量

2.2.5.2. db-compiler

db-compiler 是注解处理模块，是比较核心的模块，需要对地方库的注解实现进行抽象，定义通用的接口，然后通过 db-compiler-adapter-xxx 模块实现具体第三库注解处理器，主要工作量在这里。

2.2.5.3. db-core

db-core 主要是对第三方数据库框架接口进行抽象，定义通用接口和通用实现，对第三库的接口进行隔离。db-core 主要是定义接口，db-core-adapter-xxx 功能是实现 db-core 抽象的接口，去适配不同第三库。

2.3 可扩展项

除了满足基础的使用外，如果上层业务有相应的需要，还可以对以下功能进行扩展。

1）内存存储支持多级缓存：封装LRUCache时可以通过重写entryRemoved()，将被回收的节点数据放入另一个二级缓存存入文件。访问数据时，原本的LRUCache没有，可以再访问二级缓存。主要用于图片缓存的场景。

2）文件存储支持多种I/O方式：大部分场景倾向于用标准I/O，mmap可以作为后续版本的功能扩展，用于对同一块区域频繁读写的情况，如日志数据上报。

3）文件缓存系统：提供给内存的多级缓存或外部使用，管理缓存时间，在特定场景清理超时的缓存文件，可通过新增一个缓存索引文件记录缓存时间来实现。

4）数据库支持高阶用法：支持更多高阶用法，优化使用体验。

5）加密解密：考虑对数据加解密的扩展。

3、风险评估

1）数据库注解实现？

数据库注解编译器需要调用第三库未对外开放的API。

其他

以上未包含的其他补充项说明