存储框架调研文档

1. 背景

目前组内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。为了方便管理及访问，使上层应用忽略掉这些差异，统一由一个数据容器处理，从而方便程序的管理及上层应用层的访问，使应用层的代码更简洁。此外期望做到能较低成本的替换数据存储，比如用MMKV取代SharedPreferences。因此对以下几个方向的框架进行调研和选择。

2. 内存

内存缓存主要需考虑过期淘汰的策略。常用的缓存过期策略有：

2.1 FIFO（First In First out）：先进先出，淘汰最先近来的数据，完全符合队列。

我们可以简单地用队列实现，如LinkedList。

2.2 LRU（Least recently used）：最近最少使用，淘汰最近不使用的数据。

官方给我们提供了LruCache，它支持了LRU缓存策略，并且是线程安全的，同时还支持对多级缓存的扩展。

它的内部使用了LinkedHashMap来存储数据，该数据结构支持基于访问顺序来排列链表数据，便于根据LRU算法删除最近使用最少的数据。

它在构造方法中可以设置最大容量，当数据超过容量时，会调用trimToSize方法，开启Lru策略清除链表头的数据直到容量达标。并且提供了sizeOf和safeSizeOf方法让我们重写，从而自行计算数据的大小。

它还提供了一个可重写的方法entryRemoved()，该方法是一个空方法，它主要作用就是在结点数据value需要被删除或回收的时候，给开发者的回调。开发者就可以在这个方法里面实现一些自己的逻辑：
（1）可以进行资源的回收；
（2）可以实现二级内存缓存，可以进一步提高性能，思路如下： 重写LruCache的entryRemoved()函数，把删除掉的item，再次存入另外一个LinkedHashMap<String, SoftWeakReference<Bitmap>>中，这个数据结构当做二级缓存，每次获得图片的时候，先判断LruCache中是否缓存，没有的话，再判断这个二级缓存中是否有。
entryRemoved()在LruCache中有四个地方：put()、get()、trimToSize()、remove()中进行了调用。

结论是可以通过对LruCache进行改造，来实现内存缓存。

2.3 LFU（Least frequently used）： 最近使用次数最少， 淘汰使用总次数最少的数据。

需要记录数据的使用次数，每次清理缓存时移除次数最小的即可。

3. 文件

音频、图片等数据，需要存储到文件中，这将涉及到文件读写的I/O性能。

	标准I/O	mmap	直接I/O
优点	1）在一定程度上分离了内核空间和用户空间，保护系统本身的运行安全； 2）可以减少读盘的次数，从而提高性能。 3）学习和使用成本较低。	1）减少系统调用。只需要一次 mmap() 系统调用，后续所有的调用像操作内存一样。 2）mmap 只需要从磁盘拷贝一次，由于做过内存映射，不需要再拷贝回用户空间。 3）可靠性高。mmap 把数据写入页缓存后，跟缓存 I/O 的延迟写机制一样。	性能极高。很大程度降低了 CPU 的使用率以及内存的占用。
缺点	数据在传输过程中需要在应用程序地址空间和缓存之间进行多次数据拷贝操作，这些数据拷贝操作所带来的CPU以及内存开销非常大。	1）虚拟内存增大。mmap 会导致虚拟内存增大，mmap 大文件容易出现 OOM。 2）磁盘延迟。mmap 通过缺页中断向磁盘发起真正的磁盘 I/O，不能通过 mmap 消除磁盘 I/O 的延迟。 3）需要一定的学习和使用成本。	1）读写操作都是同步执行，导致应用程序等待。 2）学习和使用成本很高，容易踩坑。

由于我们对性能要求并没有特别高，所以我们先排除难以使用的直接I/O。

mmap适用于对同一块区域频繁读写的情况，如日志数据上报。其实现可以调用Java nio包下的MappedByteBuffer及FileChannel等。在存储框架中可以保留该实现方式供外部合适的场景下调用。

标准I/O可以使用到其余文件存储的场景。

对于标准I/O，square的Okio对其做了很好的优化，利用空间换时间使I/O速度更快，提供超时机制使I/O更稳定，另外使用也更方便。另外有一个比较优秀的缓存库ASimpleCache可以借鉴，结合Okio可以实现文件存储标准I/O方式。

结论是，由于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，故文件存储可以在Okio的基础上做一些改造。

4. KV

项目中广泛使用的SharedPreferences是谷歌提供的轻量级存储方案，使用起来比较方便，可以直接进行数据存储，不必另起线程。不过也带来很多问题，尤其是由SP引起的ANR问题，非常常见。因此寻找可以替代SP且替换成本不高的对应框架。

4.1 MMKV框架

MMKV是基于 mmap 内存映射的 key-value 组件，底层序列化/反序列化使用 protobuf 实现。

性能：使用了mmap，MMKV的读写速度很快。使用 protobuf 进行数据序列化，写入操作是增量修改，直接将 kv 对象加到内存末尾，操作这块内存的 Linux 内核会自动将这部分数据写入到文件。

稳定性：由于使用了mmap，App 只管往里写数据，最后由操作系统负责将内存回写到文件，并且不用担心 crash 导致的数据丢失。且从2015开始在微信内部使用，稳定性较高。

使用：使用方式与SharedPreferences类似，学习成本和接入成本较低。MMKV还实现了SP的接口，可以直接使用。

数据迁移：提供了专门的api `importFromSharedPreferences()`，可以根据SP的文件名进行统一迁移。

4.2 Preferences DataStore

DataStore是Jetpack 的成员，主要用来替换 SharedPreferences。基于Flow实现，提供了Proto DataStore（存储类的对象）和 Preferences DataStore（存储键值对）。这里调研的是其中的Preferences DataStore。

稳定性：基于 Flow 实现的，不会阻塞主线程，并且保证类型安全。

使用：需要通过flow链式调用。

数据迁移：在创建Preference DataStore时，可以通过传入 `SharedPreferencesMigration`，自动完成 SharedPreferences 迁移到 DataStore，保证数据一致性，不会造成数据损坏。

4.3对比

	SharePreferences	MMKV	Preference DataStore
阻塞主线程	是	否	否
支持跨进程	否	是	否
性能	较弱	mmap和protobuf加成，性能更好	较弱
类型安全	不保证	不保证	保证
使用方法	-	与SP非常类似，且实现了SP的接口	需要使用Flow链式调用
数据迁移	-	提供api	创建时定义SharedPreferencesMigration
版本稳定性	-	在微信中使用多年，稳定性强	目前还是beta版本

整体来说，与 MMKV 相比，Preference DataStore 的优点是：

- 类型安全

但是 MMKV 的优点：

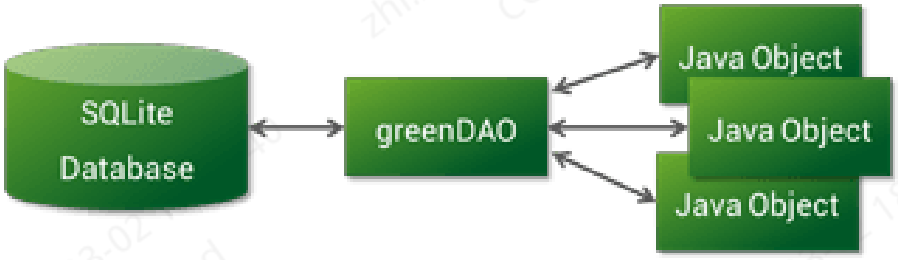
- 效率超级高
- 内存占用较小
- 支持多进程
- 发生crash也可以将数据写入到文件
- 稳定性得到验证

所以总结下来就是，建议选择MMKV作为KV框架。

5. 数据库

5.1 GreenDAO

GreenDAO 是基于 SQLite 的 ORM 框架，将 Java 对象映射到 SQLite 数据库中，不在需要编写复杂的 SQL语句， 在性能方面，GreenDAO 针对 Android 进行了高度优化，最小的内存开销、依赖体积小。



优点：

1. 高性能；
2. API 非常易用，提升了开发效率；
3. 轻量级（ < 150K ）
4. greenDAO 支持 SQLCipher 来保证用户数据的安全；

缺点：

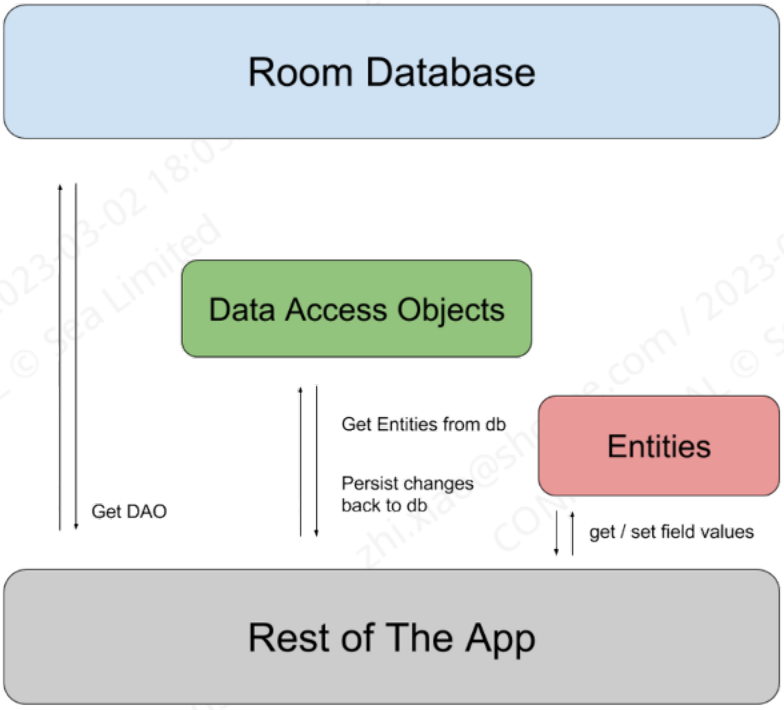
1. 已经不再积极维护, 官方目前积极维护旗下另外开源数据库ObjectBox；
2. 不支持监听数据表、Kotlin 协程等特性；
3. 需要使用第三方库支持数据库升级。

5.2 Room

Room 是 Android Jetpack 组件库一员，属于 ORM（对象关系映射）库，主要是对 SQLite 做了一层抽象，采用注解的方式，从而简化开发者对数据库操作，提升数据库性能。

Room 包含 3 个主要组件：

1. Database：包含数据库持有者，并作为应用已保留的持久关系型数据的底层连接的主要接入点。
2. Entity：表示数据库中的表。
3. DAO：包含用于访问数据库的方法。



应用使用 Room 数据库来获取与该数据库关联的数据访问对象 (DAO)。然后，应用使用每个 DAO 从数据库中获取实体，然后再将对这些实体的所有更改保存回数据库中。 最后，应用使用实体来获取和设置与数据库中的表列相对应的值。

优点：

1. Android 官方维护, 可靠性较高。
2. 支持和 LiveData 结合使用，实现数据的动态刷新和绑定组件生命周期功能。
3. 支持 Kotlin 协程 / RxJava。
4. 支持数据库的升级和降级。
5. 默认会让主线程的数据库查询操作崩溃，可以通过allowMainThreadQueries绕过这个限制。

缺点：

1. 对数据库的增删改查要写 SQL 语句，虽然编译时会检查 SQL 语法，但仍然需要关心表名、字段名等。

5.3 Realm

Realm 是一个 MVCC（多版本并发控制）数据库，它并不是基于SQLite所构建的，底层用 C++ 编写，目标是取代 SQLite。它拥有自己的数据库存储引擎，可以高效且快速地完成数据库的构建操作，它比传统的SQLite 操作更快。目前支持Android, iOS, React Native, Xamarin 等平台。

优点：

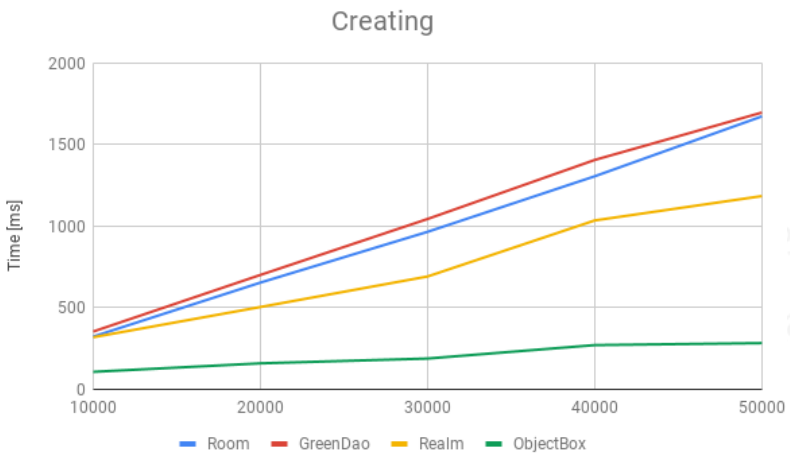
1. 由于它是完全重新开始开发的数据库实现，所以它比任何的ORM速度都快很多。
2. 支持加密，格式化查询，易于移植，支持JSON，流式api，数据变更通知等高级特性
3. 提供了一个轻量级的数据库查看工具，开发者可以查看数据库当中的内容，执行简单的插入和删除数据的操作。

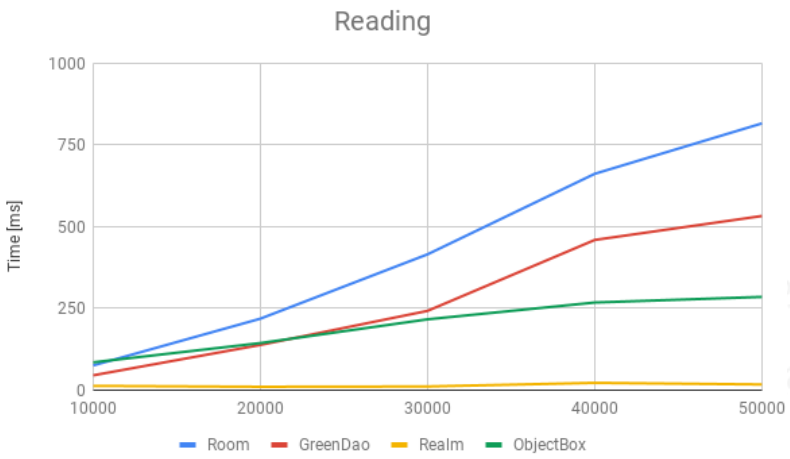
缺点：

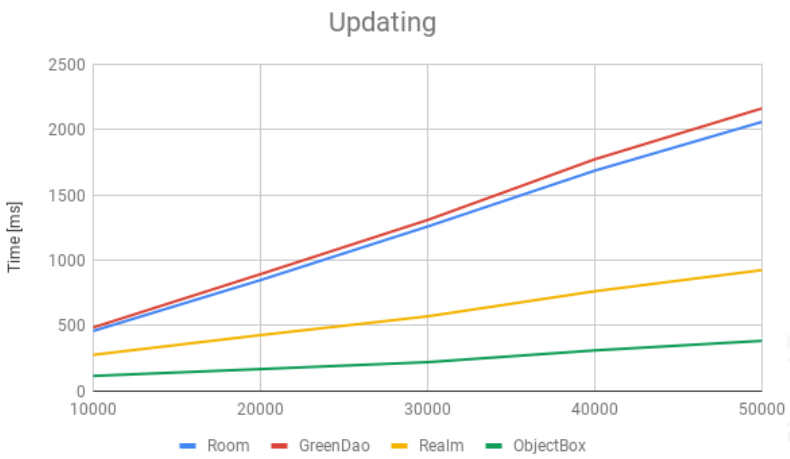
1. 自定义数据库引擎, 故会引入JNI库, 会致使apk体积增大。

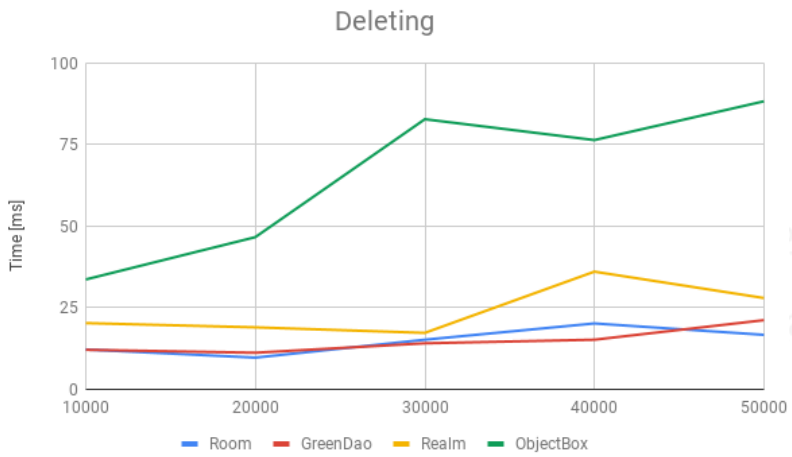
5.4 性能对比

GreenDao、Room、Realm 和 ObjectBox 增删改查性能对比（仅供参考，数据来自网络）


Creating
Time [ms]
2000
1500
1000
500
0
10000
20000
30000
40000
50000
Room
GreenDao
Realm
ObjectBox


Reading
Time [ms]
1000
750
500
250
0
10000
20000
30000
40000
50000
Room
GreenDao
Realm
ObjectBox


Updating
Time [ms]
2500
2000
1500
1000
500
0
10000
20000
30000
40000
50000
Room
GreenDao
Realm
ObjectBox



5.5 数据库框架的选用

虽然 Realm 和 ObjectBox 性能比基于 SQLite 实现的 ORM 框架性能更好，但是由于 Realm 和 ObjectBox 会显著增大 APK 体积，不推荐使用。

Room 和 GreenDao 增删改查性能差距不大，Room 由 Android 官方维护，可以方便的和 Jetpack 其它组件结合使用，推荐使用 Room。

5.6 数据库适配层方案

主流的数据库框架都是使用注解处理器来实现，可以极大的方便开发者使用，所以我们的适配层需要定义一套自定义注解，通过适配器实现不同第三方库数据库的切换。

6. 总结

本文对于各种存储场景，都讨论出了在底层的实现方案。对于内存存储，可使用LruCache来实现；对于文件存储，其I/O方式可选择Okio，维护文件路径数据即可；KV存储，考虑使用MMKV，兼顾SharedPreference的数据迁移；数据库存储推荐使用Room，考虑到第三方数据库的切换方便，使用自定义注解统一封装，也要兼顾对已有数据的迁移。

对于内存存储的扩展，可以考虑使用文件存储来做二级缓存。对于文件存储，可以考虑加入mmap的支持。