# SharedPreferences【以下简称SP】

## 是什么

Android提供的，基于XML文件存储的，轻量级的键值对数据的持久化方式。

保存少量的数据，且这些数据的格式非常简单：字符串型、基本类型的值。比如应用程序的各种配置信息（如是否打开音效、是否使用震动效果、小游戏的玩家积分等），解锁口令密码等。

## 特点

|  |  |
| --- | --- |
| 数据格式 | XML格式保存 |
| 初始化 | 子线程使用Java IO读取整个文件，进行XML解析，存入内存Map集合，对于SP的其他操作都需要等待初始化完成 |
| 保存 | commit同步提交，阻塞主线程；  apply异步提交，无法获取结果；  没有事务性API，无法保证数据一致性；  哪怕异步提交都可能会导致ANR的发生。 |
| 更新 | 在Map中的数据，全部序列化为XML，覆盖文件保存。即全量更新 |

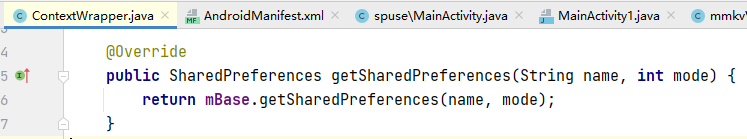
## 主要问题分析

### 1.3.1 初始化

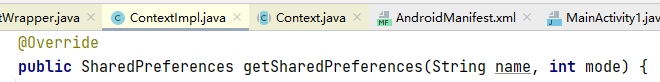
|  |
| --- |
| SharedPreferences sp = getSharedPreferences(**"test"**,  ***MODE\_PRIVATE***); |

#### 1.3.1.1 问题1

getSharedPreferences（）源码逻辑如下：



这里的mBase的运行类型是ContextImpl，找到



|  |
| --- |
| **public** SharedPreferences getSharedPreferences(String name, **int** mode) {  *// At least one application in the world actually passes in a null  // name. This happened to work because when we generated the file name  // we would stringify it to "null.xml". Nice.* **if** (**mPackageInfo**.getApplicationInfo().**targetSdkVersion** <  Build.VERSION\_CODES.***KITKAT***) {  **if** (name == **null**) {  name = **"null"**;  }  }  //分析1  File file;  **synchronized** (ContextImpl.**class**) {  **if** (**mSharedPrefsPaths** == **null**) {  **mSharedPrefsPaths** = **new** ArrayMap<>();  }  file = **mSharedPrefsPaths**.get(name);  **if** (file == **null**) {  file = getSharedPreferencesPath(name);  **mSharedPrefsPaths**.put(name, file);  }  }  //分析2  **return** getSharedPreferences(file, mode); } |

1. 分析1

用ArrayMap做了缓存，对应的key=sp文件前缀，value=sp文件对应File对象。

1. 分析2

进入

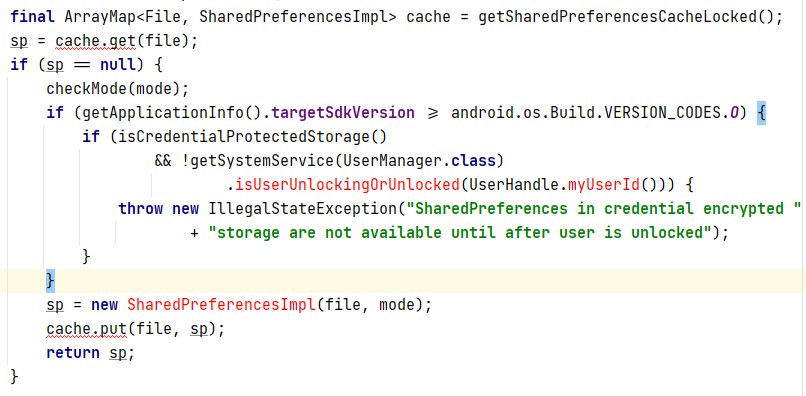
|  |
| --- |
| @Override **public** SharedPreferences getSharedPreferences(File file, **int** mode) {  SharedPreferencesImpl sp;  **synchronized** (ContextImpl.**class**) {  //分析3，4  **final** ArrayMap<File, SharedPreferencesImpl> cache = getSharedPreferencesCacheLocked();  sp = cache.get(file);  **if** (sp == **null**) {  checkMode(mode);  **if** (getApplicationInfo().**targetSdkVersion** >= android.os.Build.VERSION\_CODES.***O***) {  **if** (isCredentialProtectedStorage()  && !getSystemService(UserManager.**class**)  .isUserUnlockingOrUnlocked(UserHandle.myUserId())) {  **throw new** IllegalStateException(**"SharedPreferences in credential encrypted "** + **"storage are not available until after user is unlocked"**);  }  }  sp = **new** SharedPreferencesImpl(file, mode);  cache.put(file, sp);  **return** sp;  }  }  **if** ((mode & Context.***MODE\_MULTI\_PROCESS***) != 0 ||  getApplicationInfo().**targetSdkVersion** < android.os.Build.VERSION\_CODES.***HONEYCOMB***) {  *// If somebody else (some other process) changed the prefs  // file behind our back, we reload it. This has been the  // historical (if undocumented) behavior.* sp.startReloadIfChangedUnexpectedly();  }  **return** sp; } |

1. 分析3



再次用ArrayMap做了缓存，对应key=sp文件对应的File对象，value=和这个File对象关联的SP对象

1. 分析4：问题



假设sp中保存了100条k-v数据，一旦要使用sp，这些数据将被加载到内存中来。

之后如果不需要sp了，那sp能否得到释放？不能，因为有cache的存在，sp对象会一直在cache中。

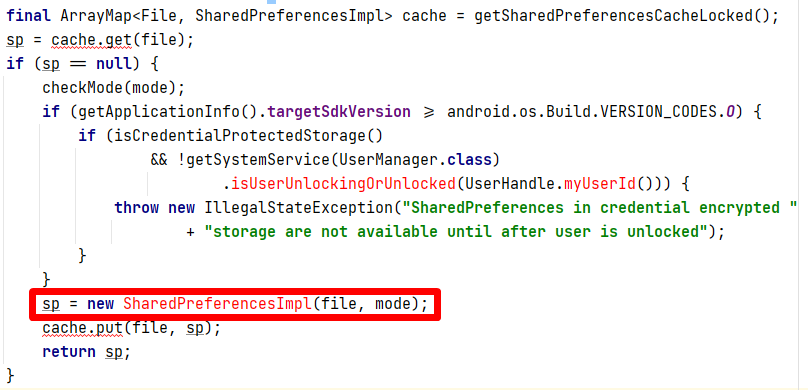
例如：在项目中对sp的使用，存在场景是在应用启动时从sp中读配置信息【如项目中cta相关SP】，然后后续运行过程中就不再需要使用sp了，那么sp对象理应被释放。由于cache的存在，sp中保存的所有数据就会出现内存泄漏。

所以缓存有好有坏。

#### 1.3.1.2 问题2

##### （1） 逻辑分析

看实例化SP，即



进入

|  |
| --- |
| @UnsupportedAppUsage SharedPreferencesImpl(File file, **int** mode) {  **mFile** = file;  **mBackupFile** = *makeBackupFile*(file);  **mMode** = mode;  **mLoaded** = **false**;  **mMap** = **null**;  **mThrowable** = **null**;  startLoadFromDisk(); } |

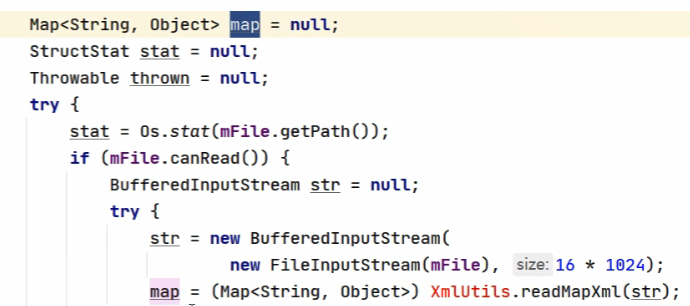
startLoadFromDisk()用于从文件恢复数据到内存，具体如下

|  |
| --- |
| @UnsupportedAppUsage(maxTargetSdk = Build.VERSION\_CODES.***R***, trackingBug = 170729553) **private void** startLoadFromDisk() {  **synchronized** (**mLock**) {  **mLoaded** = **false**;  }  **new** Thread(**"SharedPreferencesImpl-load"**) {  **public void** run() {  loadFromDisk();  }  }.start(); } |

这里的**mLoaded**意思是“数据是否加载完毕”。

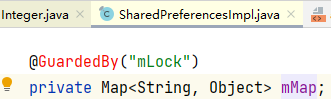
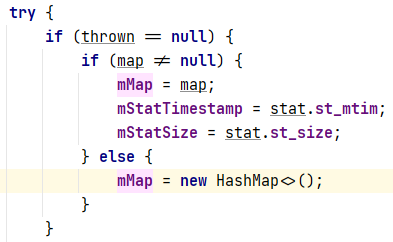
在子线程中调用**loadFromDisk（）**，该方法用于“从磁盘文件中完成数据加载”。

看重点代码，



使用**FileInputStream**完成文件读取，然后使用**XmlUtils**进行xml解析，变成内存中的Map集合。

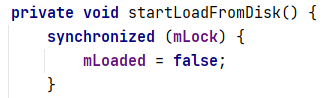
最后把临时变量**map**赋值给全局的**mMap**，



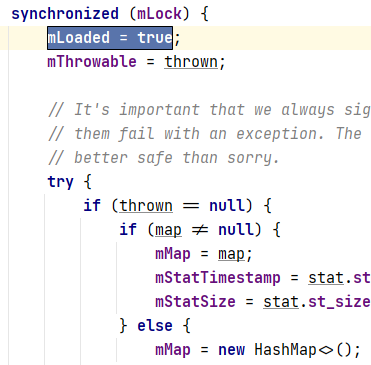
即**mMap**中保存了所有的k-v数据。到此sp的初始化完成。之后要从SP中读数据，只需mMap.get()即可。

##### （2） 问题分析

在数据加载之前，mLoaded变量在同步代码块中被置为false，



在完成数据加载时，会在同步代码块中将mLoaded置为true，



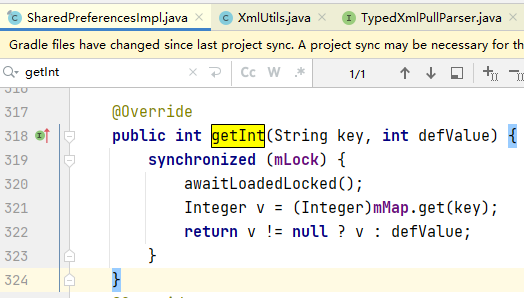
这里就需要注意mLoaded。

因为sp的用法为

|  |
| --- |
| SharedPreferences sp = getSharedPreferences(**"test"**, ***MODE\_PRIVATE***); sp.getInt(); *//读* sp.edit().putBoolean().apply(); *//写* |

###### 读数据

getInt（）源码



功能即从文件解析出来的数据的mMap集合中get（）。

但在完成mMap.get之前有个问题：需要等子线程执行完毕，mMap才有数据，否则mMap还是null。若在调mMap.get时子线程没有跑完，则是否会出现空指针？

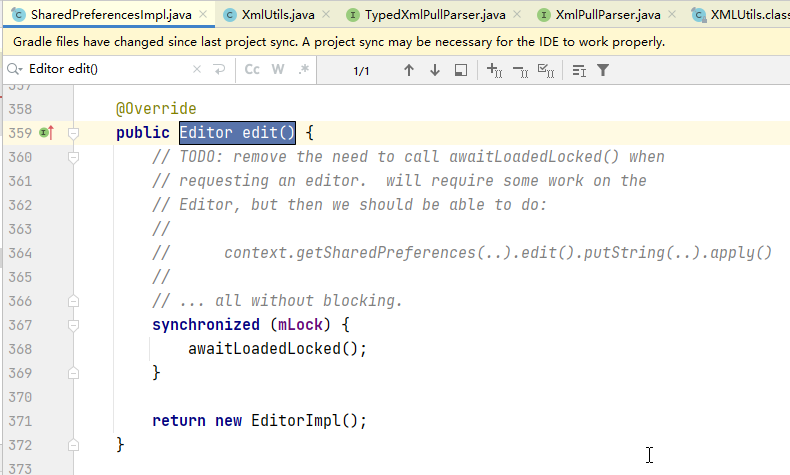
不会，因为有，



会对mLoaded进行判断，如果mLoaded为false，就会阻塞在

。即要从sp中获取数据，一定要先等待子线程执行完毕。

###### 写数据



也会调用

即必须等SP初始化完毕，才会执行下面代码。

###### 总结

当调用getSharedPreferences（），会进行耗时的io操作来完成初始化。虽然会把耗时的io操作放到子线程，不会影响主线程。但是由于我们要使用sp的读写方法时，都需要等它初始化完毕，否则一旦调用读写api就会阻塞在这，需要等待sp初始化完成。

所以sp使用时要注意：不能用sp保存太过大量，太过复杂的数据。否则可能由于sp初始化耗时很长，导致程序一个是占用太大内存，另一个是调用例如getInt（）时需要等很长的时间。

### 1.3.2 保存数据

#### 1.3.2.1 两种提交概述

（1） 返回值

* commit提交：返回值是布尔类型。如果是true的话，就表示提交成功，false就提交失败
* apply提交：返回类型是void。没法得到提交结果，也就没法知道数据到底有没有保存成功

（2） 提交方式

* commit提交：在调用线程完成提交。如果在主线程调用就在主线程提交。是同步提交。
* apply提交：使用HandlerThread创建一个Looper+子线程Handler 完成数据提交。即apply是异步提交。

#### 1.3.2.2 两种提交的问题

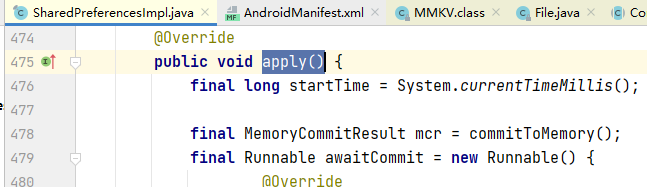
##### （1） commit

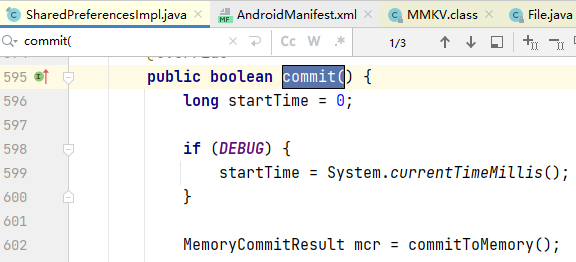
如果在主线程调用commit，且一次性保存的数据很庞大，设备的io性能较低，则需要花费较久的时间。可能导致程序出现卡顿，ANR 【概率极低的情况】。

##### （2） apply

按理来说apply异步提交，并不会阻塞主线程。

下面对apply源码分析，





无论是apply还是commit，在把数据持久化到文件之前，都会先把数据提交到内存mMap中【见上commitToMemory（）】，然后才会把内存中最新的数据写入文件。

###### 问题1

这里就有个问题：如果同步到内存成功了，但是写出到文件失败，就会导致数据不同步。即内存中的数据和文件中实际保存的数据不一致。

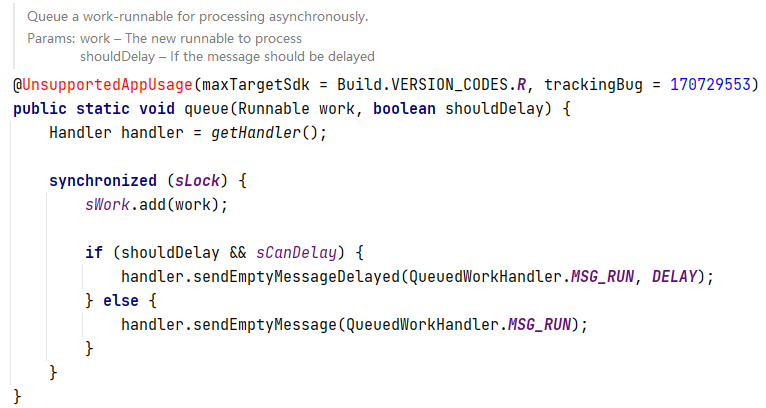
###### 问题2

接着看apply，在同步到内存后，就要提交到文件。

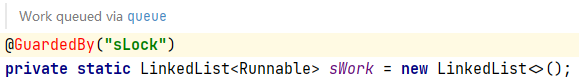
有2个方法调用，

|  |
| --- |
| QueuedWork.addFinisher()  QueuedWork.queue() |

1. queue方法

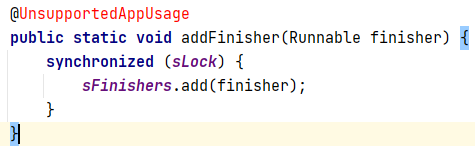


将一个任务work放到sWork集合中，

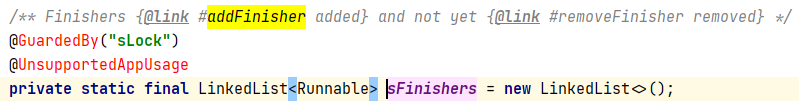


sWork是一个持久化到文件的任务队列。

1. addFinisher方法

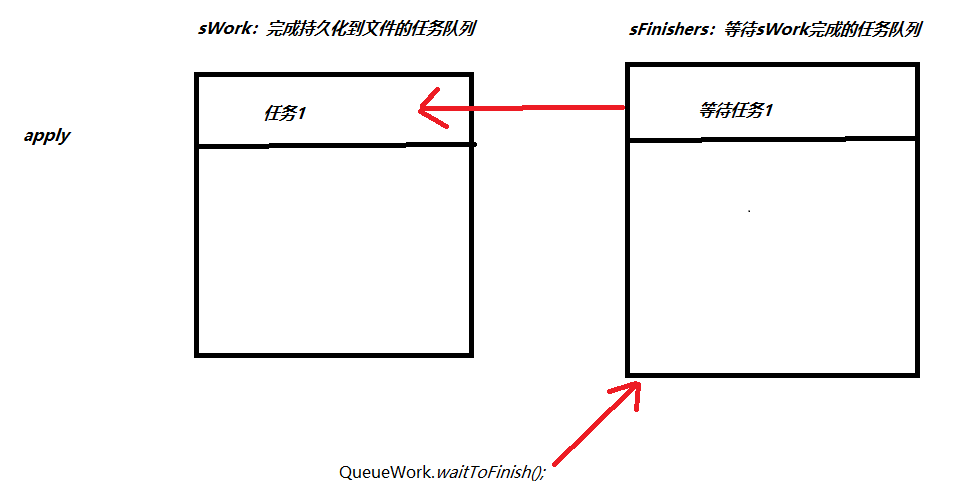


把一个任务finisher放到sFinisher集合中



sFinisher是等待sWork中work完成的任务队列。

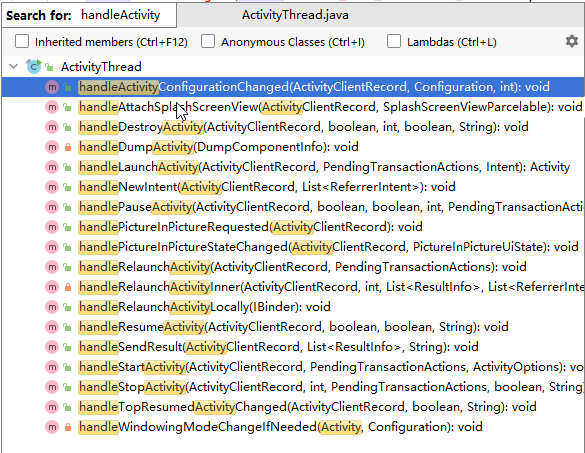
1. ANR问题



一旦调用apply来进行数据的异步提交，会往sWork中加一个任务，同时往sFinishers中加一个对应的等待任务。

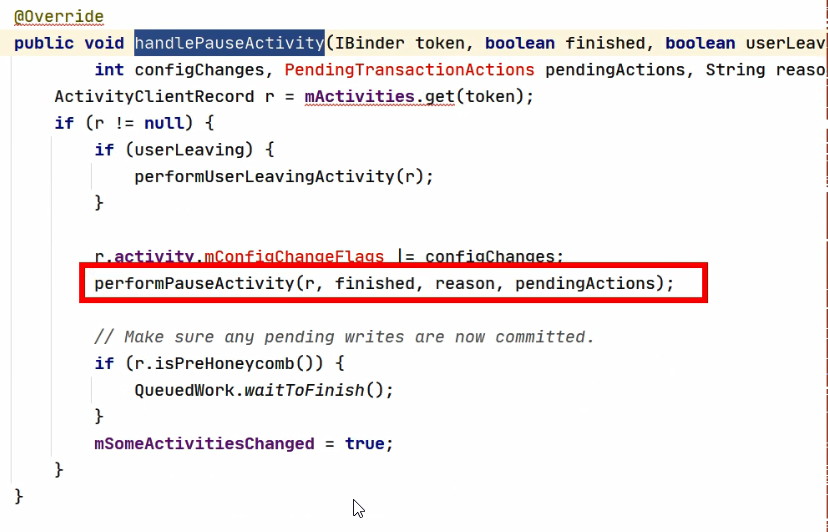
sWork中任务就是通过Handler+HandlerThread在子线程跑起来的，这不会导致程序出现ANR。

出现ANR的原因是ActivityThread。Activity的生命周期如onCreate，onResume等，是通过ActivityThread中的handleXXActivity被执行。



现在有个场景：从ActivityA跳到ActivityB，ActivityA会onPause-》onStop，ActivityB会onCreate-》onStart-》onResume

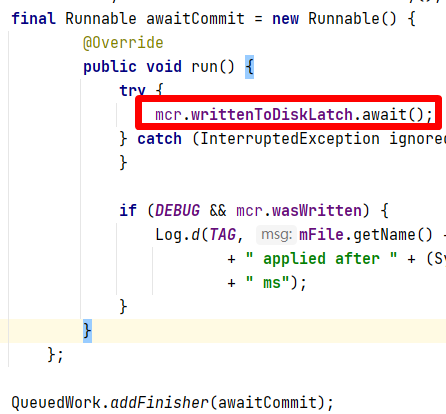
一旦要跳转，就要通过handleXXActivity调生命周期函数。如handlePauseActivity:



上面标红的就是去执行Activity的onPause，

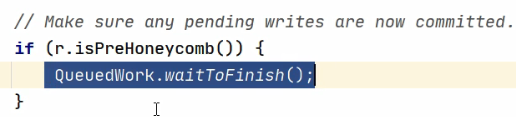
在这个之后会调用

这个方法就是去启动sFinishers队列，因为sFinishers中任务都是些等待任务，



用于等待sWork中的持久化任务执行完毕。

这里就可能出现ANR，因为虽然持久化任务在子线程提交【sWork】，但sFinishers会因在主线程等待sWork中任务执行完毕而阻塞在



导致无法调用ActivityB的onCreate（）。若sWork中任务耗时太久，超过阈值，程序就会ANR。

### 1.3.3 更新数据

在Map中的数据，全部序列化为XML，覆盖文件保存。即全量更新

# MMKV

## 是什么

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，性能高，稳定性强。

从 2015 年中至今在微信上使用，其性能和稳定性经过了时间的验证。一开始用于iOS，目前也已移植到 Android / macOS / Win32 / POSIX 平台，一并开源。

## 使用

### 引入

* 方式1：通过Maven（推荐）

在app module的build.gradle中添加如下：

|  |
| --- |
| dependencies **{** implementation **'com.tencent:mmkv:1.2.11'** *// replace "1.2.11" with any available version* **}** |

* 方式2：通过编译源码

https://github.com/Tencent/MMKV

### 基本用法

#### 1.2.2.1 初始化

##### （1） 用法

|  |
| --- |
| *//默认根目录（/data/data/包名/files/mmkv/）* String rootDir1 = MMKV.*initialize*(**this**); *//手动设根目录* String rootDir2 = MMKV.*initialize*(**this**, **"/sdcard/mmkv\_2"**); |

##### （2） 补充

【1】 单进程

《1》 MMKV.initialize()用一次即可；

《2》 若多次调MMKV.initialize()且每次的dir都不同，会生成不同目录，但仍会存在第一次MMKV.initialize()时的文件中。

【2】 多进程

MMKV.initialize()必须在每个进程中写一次。

#### 1.2.2.2 获取实例

|  |
| --- |
| MMKV.*initialize*(**this**); //必须先执行这句 *//获取默认的全局实例，非单例，持有同个文件句柄* MMKV kv = MMKV.*defaultMMKV*(); *//根据业务区别存储，创建不同id的实例* MMKV kv1 = MMKV.*mmkvWithID*(**"MyID1"**); kv1.encode(**"bool"**, **true**); System.***out***.println(**"ID=KV1, bool: "** + kv1.decodeBool(**"bool"**)); MMKV kv2 = MMKV.*mmkvWithID*(**"MyID2"**); kv2.encode(**"bool"**, **false**); System.***out***.println(**"ID=KV2, bool: "** + kv2.decodeBool(**"bool"**)); *//多进程同步支持* MMKV kv3 = MMKV.*mmkvWithID*(**"MyID"**, MMKV.***MULTI\_PROCESS\_MODE***); |

#### 1.2.2.3 CRUD

|  |
| --- |
| MMKV.*initialize*(**this**); //必须先执行这句 MMKV kv = MMKV.*defaultMMKV*(); //必须先执行这句 *//法1：使用MMKV自有方法 //-添加/更新 数据* kv.encode(key, value); *//-删除数据* kv.removeValueForKey(); *//-获取数据* type value = kv.decodeXX(key);  *//法2：使用SP简化方式 //-添加/更新 数据* kv.putXX(key, value); *//-删除数据* kv.remove(key); *//-获取数据* type value = kv.getXX(key);  *//法3：使用SP原有方式* SharedPreferences.Editor editor = kv.edit(); editor.putXX(key, value); *//无需调用commit/apply //editor.commit();* |

#### 1.2.2.4 从SP迁移

|  |
| --- |
| MMKV.*initialize*(**this**); MMKV kv = MMKV.*defaultMMKV*(); SharedPreferences oldData = getSharedPreferences(**"myData"**, ***MODE\_PRIVATE***); *//迁移旧数据* kv.importFromSharedPreferences(oldData); *//清空旧数据* oldData.edit().clear().commit(); |

#### 1.2.2.5 多进程

##### （1）mmkv支持的多进程有以下场景：

《1》 应用内多进程：共享文件可在内部存储也可在外部存储；

《2》 跨应用多进程：共享文件只能在外部存储，不能在内部存储。

注：尽量不要在外部存储上共享数据。

参考资料：

<https://github.com/Tencent/MMKV/issues/838>

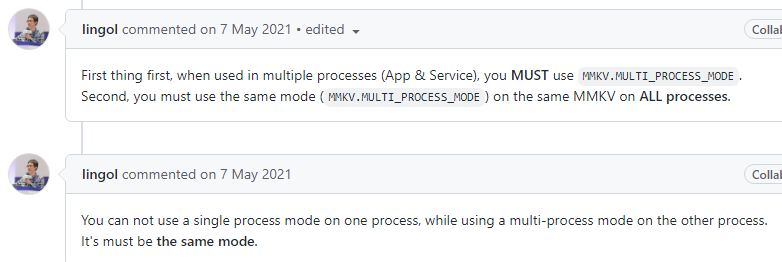
<https://github.com/Tencent/MMKV/issues/568>

<https://github.com/Tencent/MMKV/issues/838>

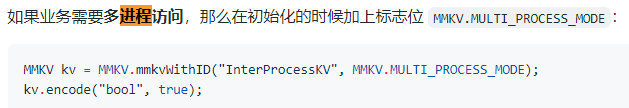
<https://github.com/Tencent/MMKV/issues/802>

##### mmkv多进程mode的设置要求

<https://github.com/Tencent/MMKV/issues/681>



<https://github.com/Tencent/MMKV/wiki/android_tutorial_cn>



### 进阶用法

#### 2.2.3.1 日志

<https://github.com/Tencent/MMKV/wiki/android_advance_cn>

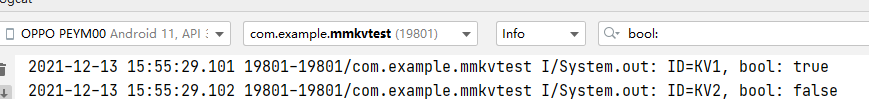
<https://javadoc.io/doc/com.tencent/mmkv/latest/com/tencent/mmkv/MMKVHandler.html>

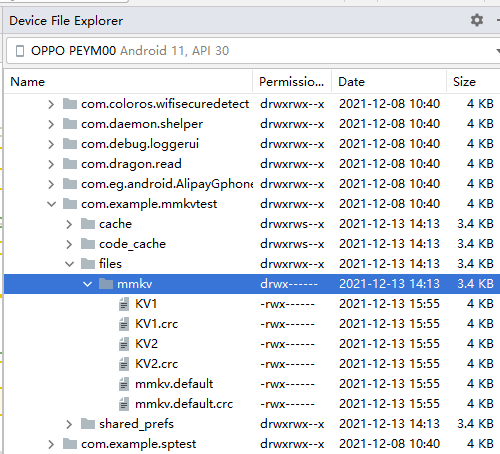
见TestMMKV项目的logtest包

### 示例

1. 多实例+存取+存储位置

|  |
| --- |
| *//根目录* MMKV.*initialize*(**this**); *//不同实例* MMKV kv = MMKV.*defaultMMKV*(); MMKV kv1 = MMKV.*mmkvWithID*(**"KV1"**); kv1.encode(**"bool"**, **true**); System.***out***.println(**"ID=KV1, bool: "** + kv1.decodeBool(**"bool"**)); MMKV kv2 = MMKV.*mmkvWithID*(**"KV2"**); kv2.encode(**"bool"**, **false**); System.***out***.println(**"ID=KV2, bool: "** + kv2.decodeBool(**"bool"**)); |





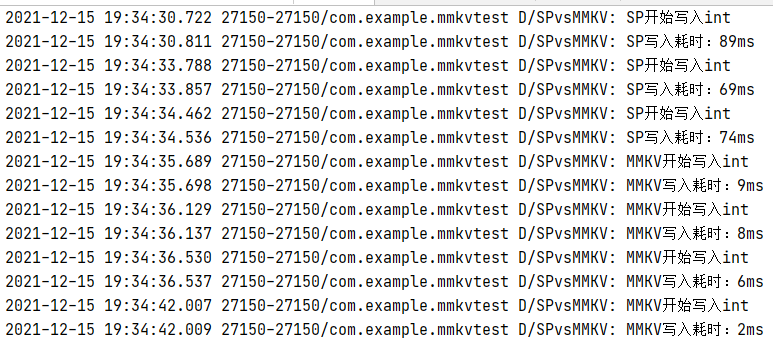
Crc文件：主要用于判别文件是否存在错误。当crc校验出错误时，若仍按正常文件读取，会丢弃部分数据；若按错误文件读取，丢弃所有数据。

## 和SP的定量比对

|  |
| --- |
| **public class** MainActivity **extends** AppCompatActivity {   **private** SharedPreferences **mSharedPreferences**;  **private** MMKV **mMMKV**;   **private static final** String ***TAG*** = **"SPvsMMKV"**;   @Override  **protected void** onCreate(Bundle savedInstanceState) {  **super**.onCreate(savedInstanceState);  setContentView(R.layout.***activity\_main1***);   **mSharedPreferences** = getSharedPreferences(**"testSP"**, ***MODE\_PRIVATE***); String rootDir = MMKV.*initialize*(**this**);  Log.*d*(***TAG***, **"mmkv root:"** + rootDir);  **mMMKV** = MMKV.*defaultMMKV*();  }   **public void** testSPWrite(View view) {  Log.*d*(***TAG***, **"SP开始写入int"**);  Random random = **new** Random();  SharedPreferences.Editor edit = **mSharedPreferences**.edit();  **long** start = System.*currentTimeMillis*();  **for** (**int** i = 0; i < 1000; i++) { *// edit.putInt(String.valueOf(i), random.nextInt()).commit();* edit.putInt(String.*valueOf*(i), random.nextInt()).apply();  }  **long** time = System.*currentTimeMillis*() - start;  Log.*d*(***TAG***, **"SP写入耗时："** + time + **"ms"**);  }   **public void** testMMKVWrite(View view) {  Log.*d*(***TAG***, **"MMKV开始写入int"**);  Random random = **new** Random();  **long** start = System.*currentTimeMillis*();  **for** (**int** i = 0; i < 1000; i++) {**mMMKV**.encode(String.*valueOf*(i), random.nextInt());  }  **long** time = System.*currentTimeMillis*() - start;  Log.*d*(***TAG***, **"MMKV写入耗时："** + time + **"ms"**);}   **public void** testSPRead(View view) {  Log.*d*(***TAG***, **"SharedPreferences开始读取int"**);  **long** start = System.*currentTimeMillis*();  **for** (**int** i = 0; i < 1000; i++) {  **mSharedPreferences**.getInt(String.*valueOf*(i), -1);  }  **long** time = System.*currentTimeMillis*() - start;  Log.*d*(***TAG***, **"SP读取耗时："** + time + **"ms"**);  }   **public void** testMMKVRead(View view) {  Log.*d*(***TAG***, **"MMKV开始读取int"**);  **long** start = System.*currentTimeMillis*();  **for** (**int** i = 0; i < 1000; i++) {  **mMMKV**.getInt(String.*valueOf*(i), -1);  }  **long** time = System.*currentTimeMillis*() - start;  Log.*d*(***TAG***, **"MMKV读取耗时："** + time + **"ms"**);  } } |

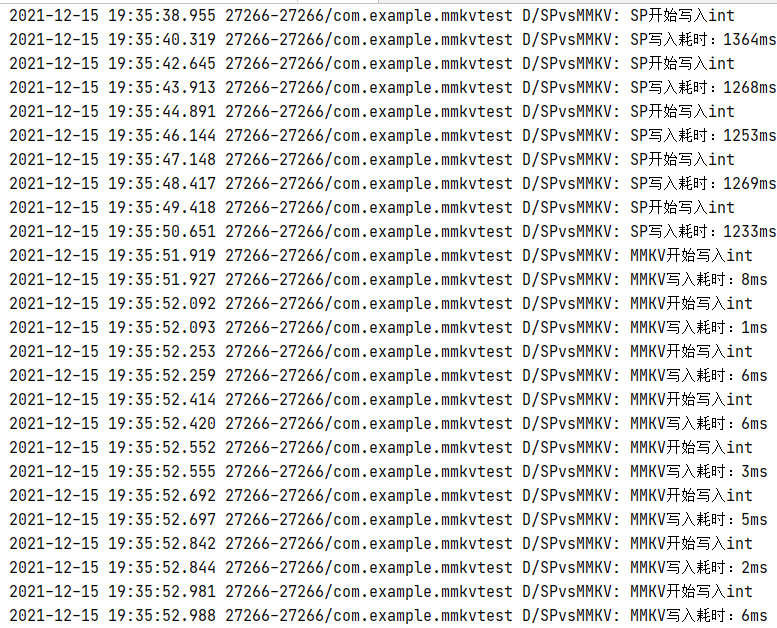
### 2.3.1 写数据

#### 【1】 SP用apply，MMKV用encode



快10倍左右。

#### 【2】 SP用commit，MMKV用encode



快200倍左右。

### 2.3.2 读数据



差不多，因为都在内存中读取map，不会涉及文件操作。

## 和SP的定性比对

这里主要针对1.2中SP的特点进行比对，基于4个方向，介绍了mmkv的优化思路以及如何解决sp的各种设计问题。

### 2.4.1 高效的文件操作

#### 2.4.1.1 和SP比对点

##### （1） SP的问题

SP需要子线程使用Java IO进行文件操作。

##### （2） MMKV的改进

mmkv针对此优化文件操作，使用mmap内存映射技术。

#### 2.4.1.2 文件操作的常用方式

##### （1） 传统I/O

###### 【1】 是什么

如java的InputStream，OutputStream

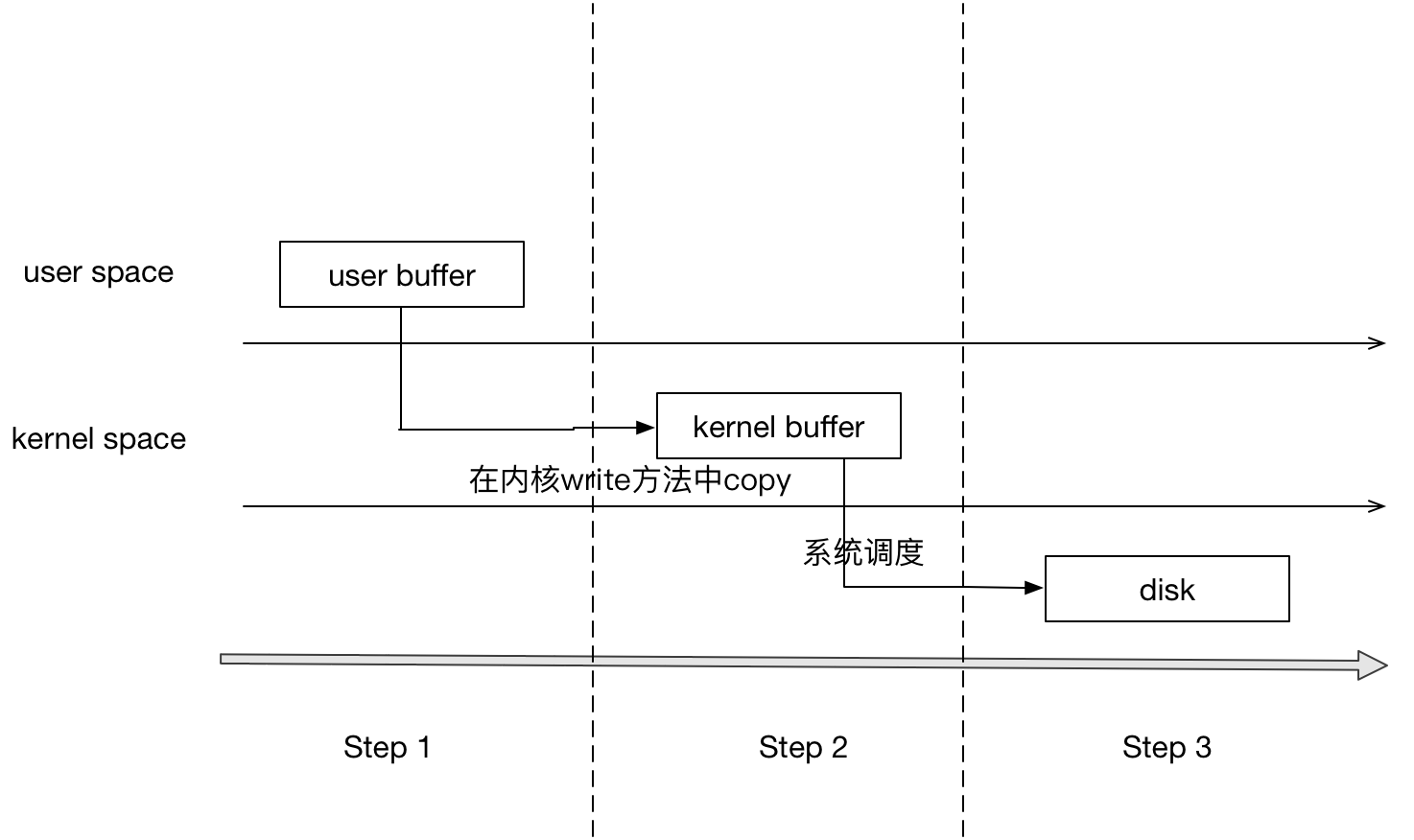
###### 【2】 用户空间和内核空间

虚拟内存被操作系统划分成两块：用户空间和内核空间。

1. 用户空间是用户程序代码运行的地方。如在程序中new一个对象，对象使用的内存属于用户空间。
2. 内核空间是内核代码运行的地方。

为了安全，它们是隔离的，即使用户的程序崩溃了，内核也不受影响。

###### 【3】 写文件流程



1. 第1步

调用write方法向内核发起系统调用，上下文从用户态切换为内核态；

1. 第2步

CPU将用户缓冲区中的数据拷贝到内核空间的缓冲区。【CPU拷贝】

1. 第3步

CPU利用DMA控制器将数据从内核缓冲区拷贝到磁盘缓冲区进行数据传输。【DMA拷贝】

1. 第4步

上下文从内核态切换回用户态，write系统调用执行返回。

总结：一次写操作的详细过程很复杂，包括2次上下文切换，2次数据拷贝【CPU+DMA】。

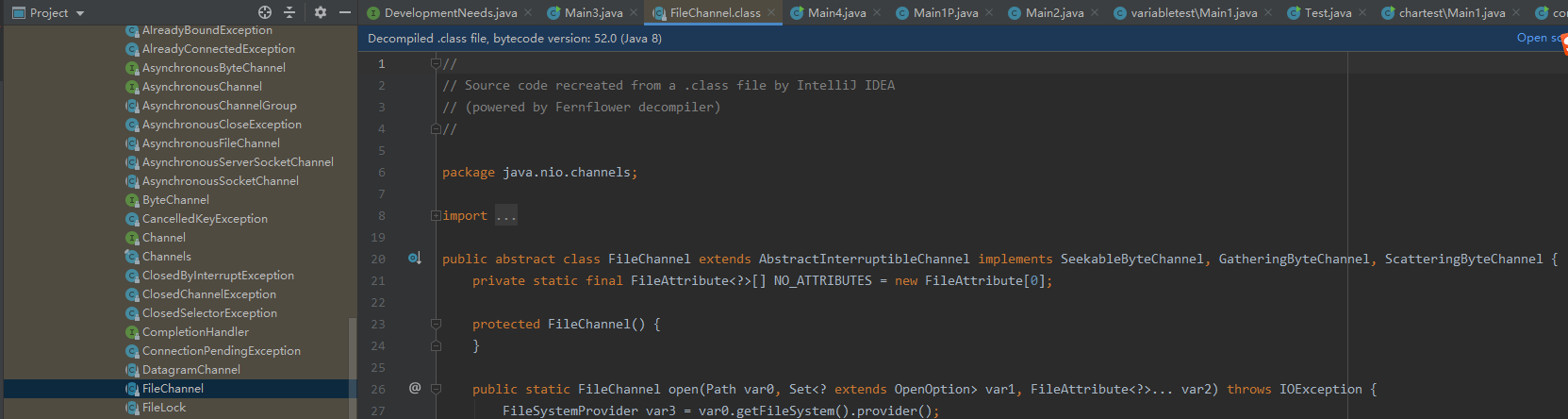
##### （2） 零拷贝技术

###### 【1】 是什么

没有CPU参与拷贝的技术。

###### 【2】 JDK中实现-FileChannel

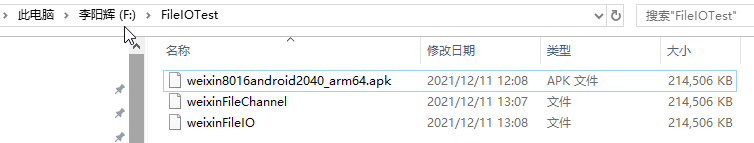
《1》 是什么



A channel for reading, writing, mapping, and manipulating a file.

《2》 测试

|  |
| --- |
| package unittest;  import javax.swing.plaf.FileChooserUI; import java.io.FileInputStream; import java.io.FileNotFoundException; import java.io.FileOutputStream; import java.io.IOException; import java.nio.channels.FileChannel;  public class FileUnitTest {   public void testFileIO() throws IOException {  long start = System.*currentTimeMillis*();  FileInputStream fis = new  FileInputStream("F:\\FileIOTest\\weixin8016android2040\_arm64.apk");  FileOutputStream fos = new  FileOutputStream("F:\\FileIOTest\\weixinFileIO");  int len;  byte[] buffer = new byte[4096];   while ((len = fis.read(buffer)) != -1) {  fos.write(buffer, 0, len);  }   fis.close();  fos.close();  long time = System.*currentTimeMillis*() - start;  System.*out*.println("testFileIO耗时：" + time + "ms");  }   public void testFileChannel() throws IOException {  long start = System.*currentTimeMillis*();  FileChannel inChannel = new FileInputStream("F:\\FileIOTest\\weixin8016android2040\_arm64.apk").getChannel();  FileChannel outChannel = new FileOutputStream("F:\\FileIOTest\\weixinFileChannel").getChannel();  outChannel.transferFrom(inChannel, 0, inChannel.size());  inChannel.close();  outChannel.close();  long time = System.*currentTimeMillis*() - start;  System.*out*.println("testFileChannel耗时：" + time + "ms");  } }  class Client {   public static void main(String[] args) {  try {  new FileUnitTest().testFileIO(); // new FileUnitTest().testFileChannel();  } catch (IOException e) {  e.printStackTrace();  }  } } |







《3》 原理

根据jvm运行的系统，依次选择sendfile，mmap或IO。其中sendfile和mmap都是零拷贝技术。

###### 【3】 MMAP-使用零拷贝技术

《1》 是什么

Linux通过将一个虚拟内存区域与一个磁盘上的对象关联起来，以初始化这个虚拟内存区域的内容，这个过程称为内存映射(memory mapping)。

mmap（）是Linux的系统调用函数，是实现内存映射的接口。

|  |
| --- |
| **void \*mmap(void \****addr***, size\_t** *length***, int** *prot***, int** *flags***,** **int** *fd***, off\_t** *offset***);** |

参数：

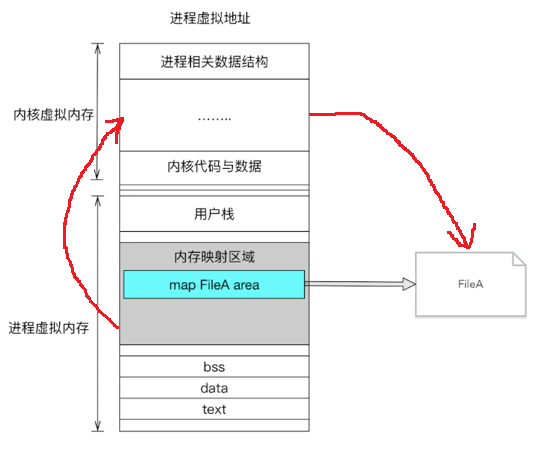
* addr：映射区的开始地址
* length：映射区的长度。//长度单位是 以字节为单位，不足一内存页按一内存页处理
* fd：有效的文件描述符
* offset：被映射对象内容的起点

返回值：

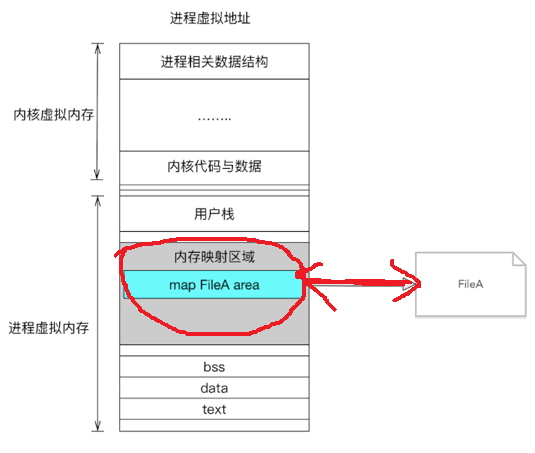
On success, **mmap**() returns a pointer to the mapped area. On error, the value **MAP\_FAILED** (that is, *(void \*) -1*) is returned, and [*errno*](https://man7.org/linux/man-pages/man3/errno.3.html) is set to indicate the error.

《2》 和传统IO写数据 对比

传统IO



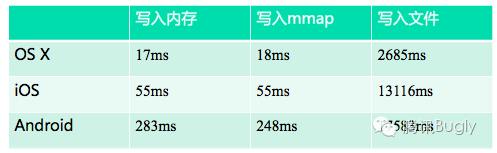
MMAP



《3》 速度测试

<https://mp.weixin.qq.com/s/kDPTt9Rtd-PERXXW-UyUlQ>

腾讯团队：为了验证 mmap 是否真的有直接写内存的效率，我们写了一个简单的测试用例：把512 Byte的数据分别写入150 kb大小的内存和 mmap，以及磁盘文件100w次并统计耗时



#### 2.4.1.3 mmkv优化方案

通过 mmap 内存映射文件，提供一段可供随时写入的内存块，App 只管往里面写数据，由操作系统在如内存不足、进程退出等时候负责将内存回写到文件，不必担心 crash 导致数据丢失。



之后能通过m\_ptr进行内存读写，就相当于对文件进行读写。

例如：m\_ptr[0] = 1，类似于操作java的byte数组。

### 2.4.2 比XML更精简的数据格式

#### 2.4.2.1 和SP比对点

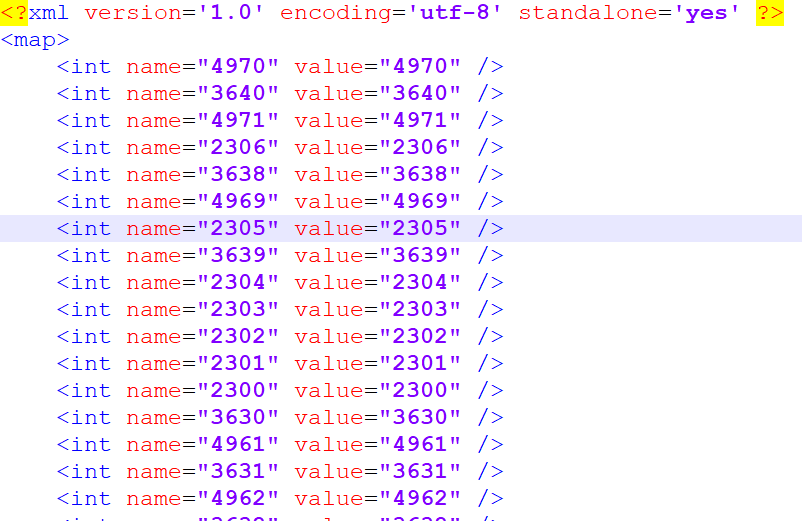
##### （1） SP的问题

SP的数据格式是xml，xml格式解析、操作起来很麻烦，且冗余信息很多，占用空间较大。所以思路是完全抛弃xml，设计一个比xml更精简的数据格式。

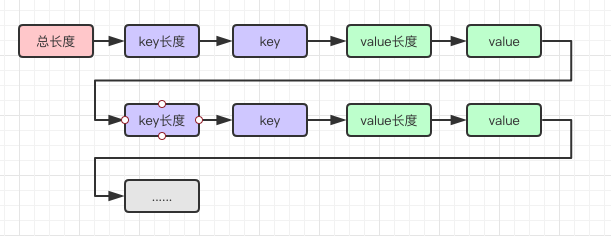
##### （2） MMKV的改进

MMKV使用protobuf协议进行数据序列化，pb在性能上总体优于XML。

SP的xml格式：基于文本/字符串



MMKV数据格式：类似链表，基于二进制

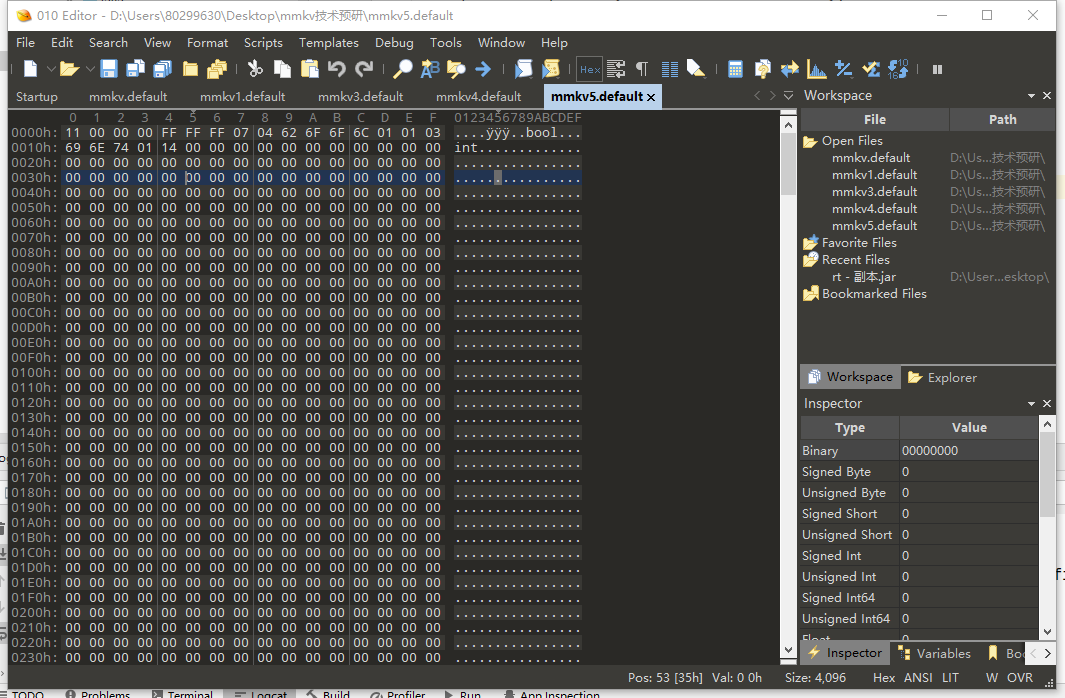


#### 2.4.2.2 MMKV存储格式解析

##### （1） 代码

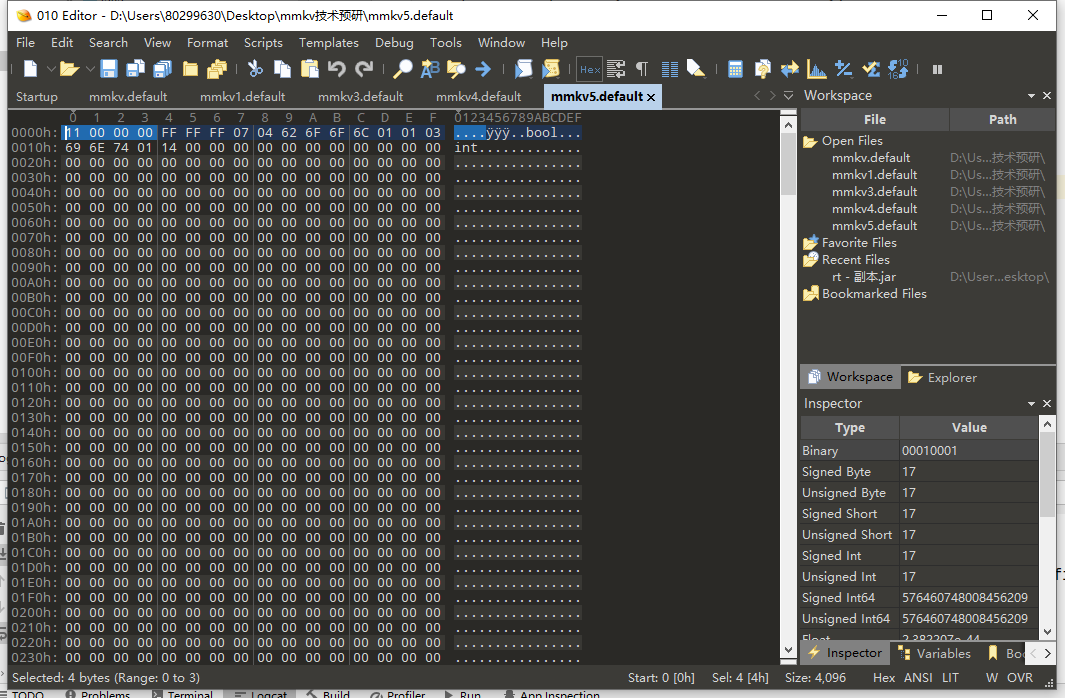
|  |
| --- |
| String rootDir = MMKV.*initialize*(**this**); System.***out***.println(**"mmkv root: "** + rootDir); MMKV kv = MMKV.*defaultMMKV*();  kv.encode(**"bool"**, **true**); System.***out***.println(**"bool: "** + kv.decodeBool(**"bool"**));  kv.encode(**"int"**, 20); System.***out***.println(**"int: "** + kv.decodeInt(**"int"**)); |

##### （2） 数据格式解析

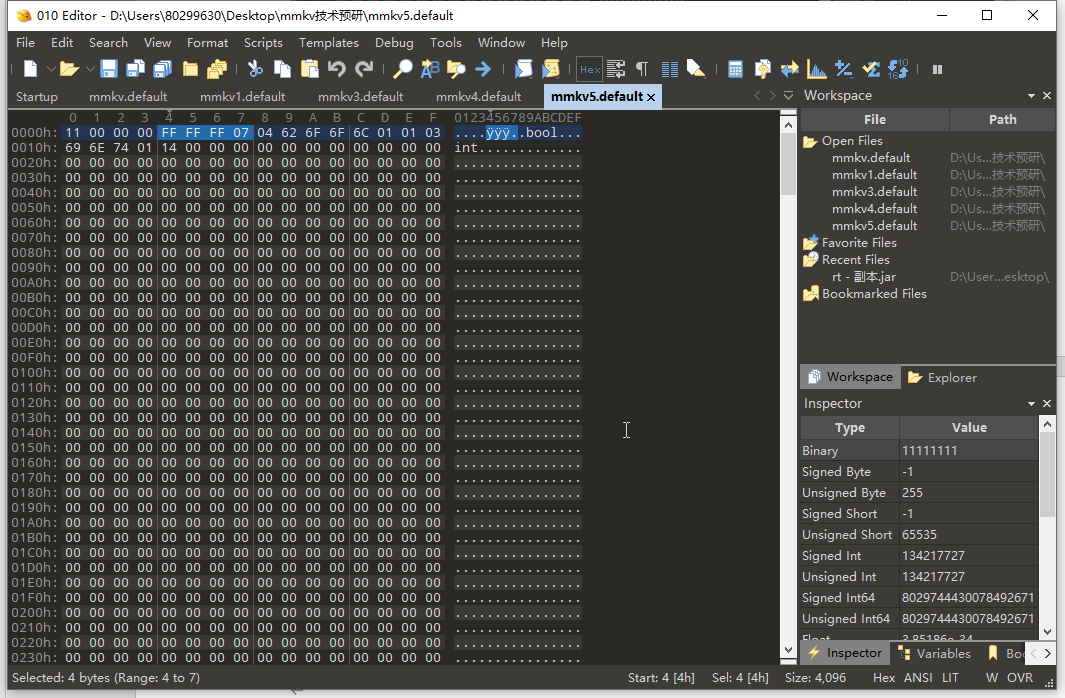


其中：

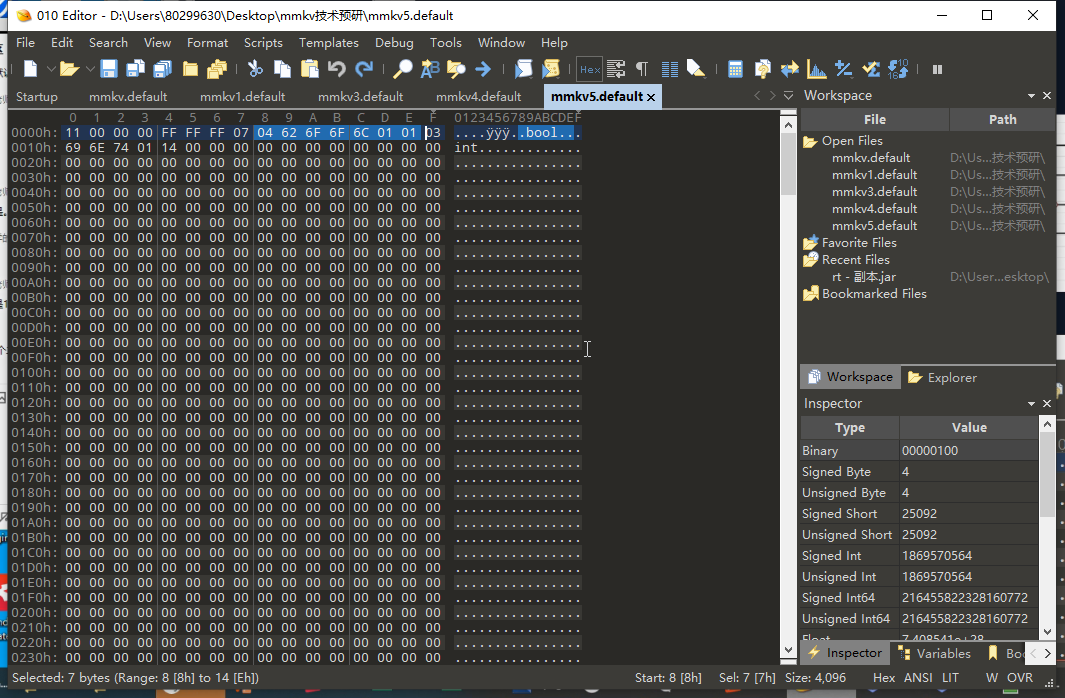
1. 1-4个字节：表有效数据的长度【单位字节】，固定4字节



1. 5-8字节：MMKV固定标识



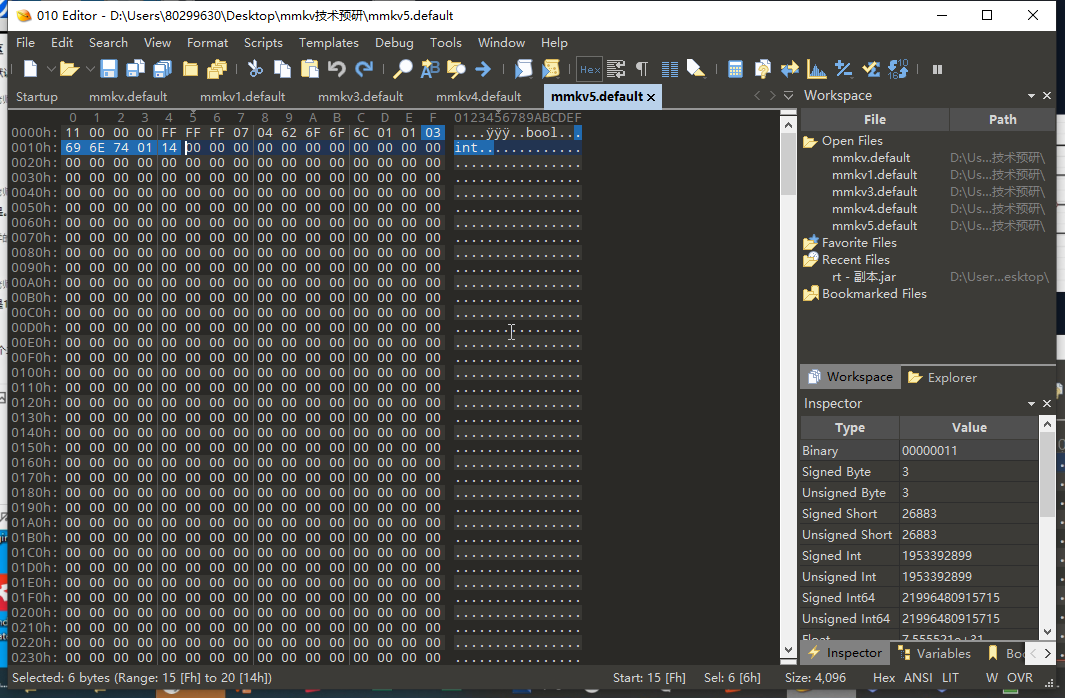
1. 第1个key-value



04表key为4字节-》“bool”占4字节

01表value为1字节-》true占1字节

1. 第2个key-value

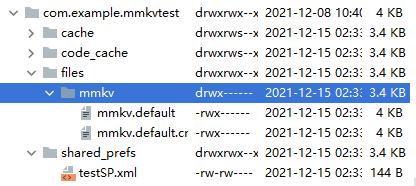


03表key为3字节-》“int”是3字节

01表value为1字节-》20是1字节

##### （3） MMKV的限制

|  |
| --- |
| **private void** sp() {  SharedPreferences.Editor editor = getSharedPreferences(**"testSP"**, ***MODE\_PRIVATE***).edit();  editor.putBoolean(**"bool"**, **true**).commit();  editor.putInt(**"int"**, 1).commit();}   **private void** mmkv() {  String rootDir = MMKV.*initialize*(**this**);  System.***out***.println(**"mmkv root: "** + rootDir);  MMKV kv = MMKV.*defaultMMKV*();   kv.encode(**"bool"**, **true**);  System.***out***.println(**"bool: "** + kv.decodeBool(**"bool"**));   kv.encode(**"int"**, 1);  System.***out***.println(**"int: "** + kv.decodeInt(**"int"**));} |



因为mmap存在限制

|  |
| --- |
| **void \*mmap(void \****addr***, size\_t** *length***, int** *prot***, int** *flags***,** **int** *fd***, off\_t** *offset***);** |

第二个参数length，表要映射到内存中的字节数量。但length必须是：x页【x>=1】。在Linux的内存管理中，将内存按页为单位来管理。页的默认大小为4096B。

所以MMKV映射最小也要4KB，即文件最小占4KB。

MMKV起始按4KB设置，之后若更新数据所需空间>4KB需要扩容，见2.4.3.2。

### 2.4.3 更优的数据更新方式

#### 2.4.3.1 和SP比对点

##### （1） SP的问题

SP因为使用xml，所以是 **全量更新**。

因此，xml中数据量的大小，的确会对 **写操作** 的效率有一定的影响。

假设xml中原本有100条数据，就算只改1条数据，也要操作100条数据，把100条数据变成xml，然后覆盖文件。

##### （2） MMKV的改进

因此MMKV采用增量更新+全量更新的方式。即优先增量写入，不行再全量写入。

#### 2.4.3.2 MMKV写入详解

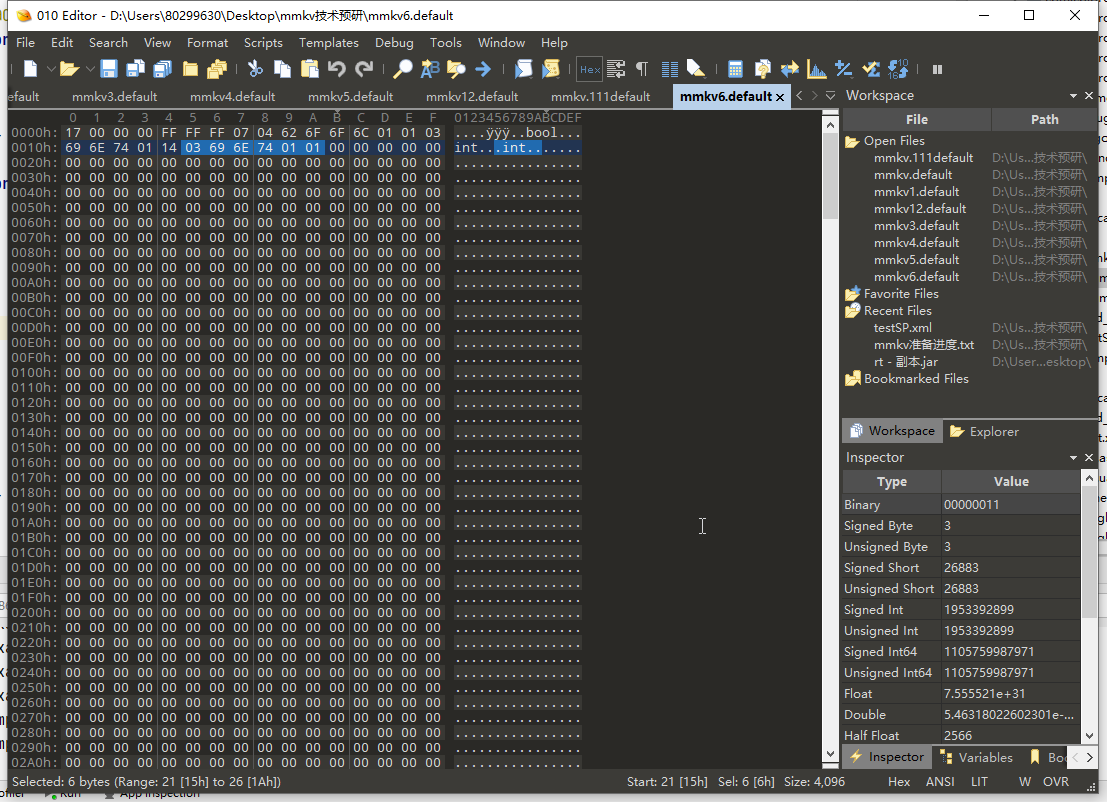
##### （1） MMKV的增量写入

###### 【1】 代码

在2.4.2.2的代码上修改

|  |
| --- |
| String rootDir = MMKV.*initialize*(**this**); System.***out***.println(**"mmkv root: "** + rootDir); MMKV kv = MMKV.*defaultMMKV*();  //原本 kv.encode(**"bool"**, **true**); System.***out***.println(**"bool: "** + kv.decodeBool(**"bool"**));  kv.encode(**"int"**, 20); System.***out***.println(**"int: "** + kv.decodeInt(**"int"**));  //新增 kv.encode(**"int"**, 1); System.***out***.println(**"int: "** + kv.decodeInt(**"int"**)); |

如下：



03表key为3字节-》“int”是3字节

01表value为1字节-》1是1字节

###### 原理

MMKV同SP一样，在实例化时，也要去解析整个文件，把文件中数据保存到内存散列表。当往后面追加来完成数据的更新不会有任何问题。因为解析文件时是从前往后解析，当解析到key为“int”，value为20的数据，直接往Map中put。之后解析到key为“int”，value为2的数据，也是直接put。所以Map中保存的一定是最新修改的数据。

假如mmkv已经保存了200条数据，现在要更新不用管前面的200条数据，只要管这1条数据即可。这种增量更新的方式，效率肯定比覆盖整个文件来的快。

##### （2） MMKV的全量写入

###### 【1】 前提

如果不断增量追加内容，文件越来越大，怎么处理？

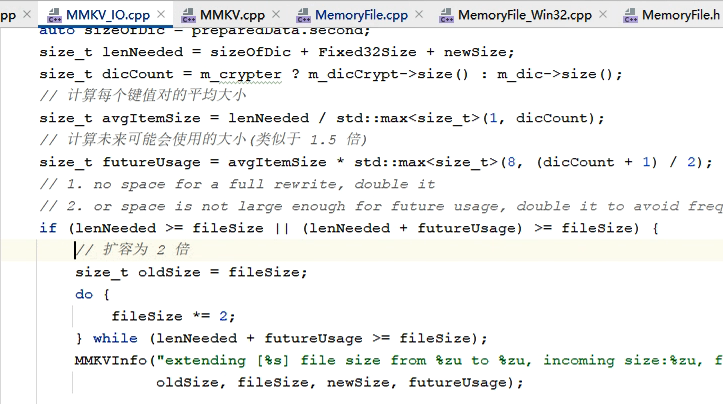
###### 【2】 解决思路

当文件大小不够，将数据去掉重复key后，

* 如果文件大小满足写入的数据大小，则可以直接更新全量写入；
* 去重后文件大小依旧不够（如：无重复KEY），则先扩容再写入。

###### 【3】 扩容代码

《1》 算法



《2》 重新映射

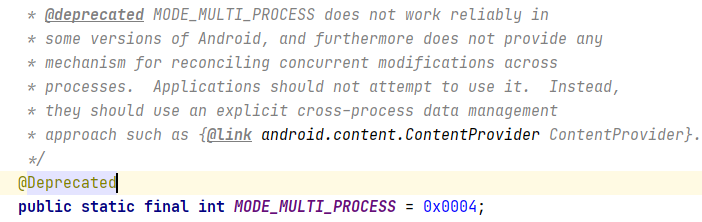
|  |
| --- |
| *//解除映射* munmap(m\_ptr, oldSize); *//重新设定文件大小* ftruncate(m\_fd, m\_size\*2); *//重新映射* m\_ptr = (int8\_t \*) mmap(0, m\_size\*2, **PROT\_READ** | **PROT\_WRITE**, **MAP\_SHARED**, m\_fd,0); |

### 2.4.4 多进程问题

#### 2.4.4.1 和SP比对点

##### （1） SP的问题

MODE\_MULTI\_PROCESS，跨进程模式，如果项目有多个进程使用同一个sp文件，需要使用该模式，但是从Android6.0开始废弃。原因如下：

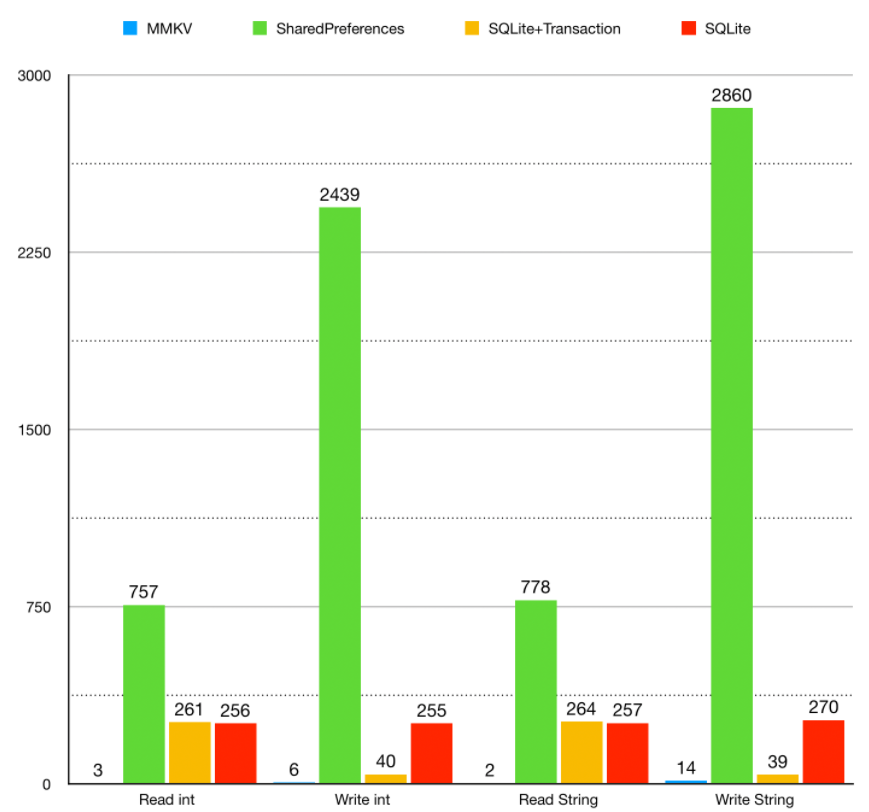


Android不保证该模式总是能正确的工作，建议使用ContentProvider替代。推荐使用ContentProivder来处理多进程间的文件共享。

##### （2） MMKV的改进

用了flock文件锁来保证多进程操作文件的同步，等原理。

<https://github.com/Tencent/MMKV/wiki/android_ipc>



测试环境：Pixel 2 XL 64G, Android 8.1.0，单位：ms。每组测试分别循环 1000 次；MultiProcessSharedPreferences 使用 apply() 同步数据；SQLite 打开 WAL 选项。