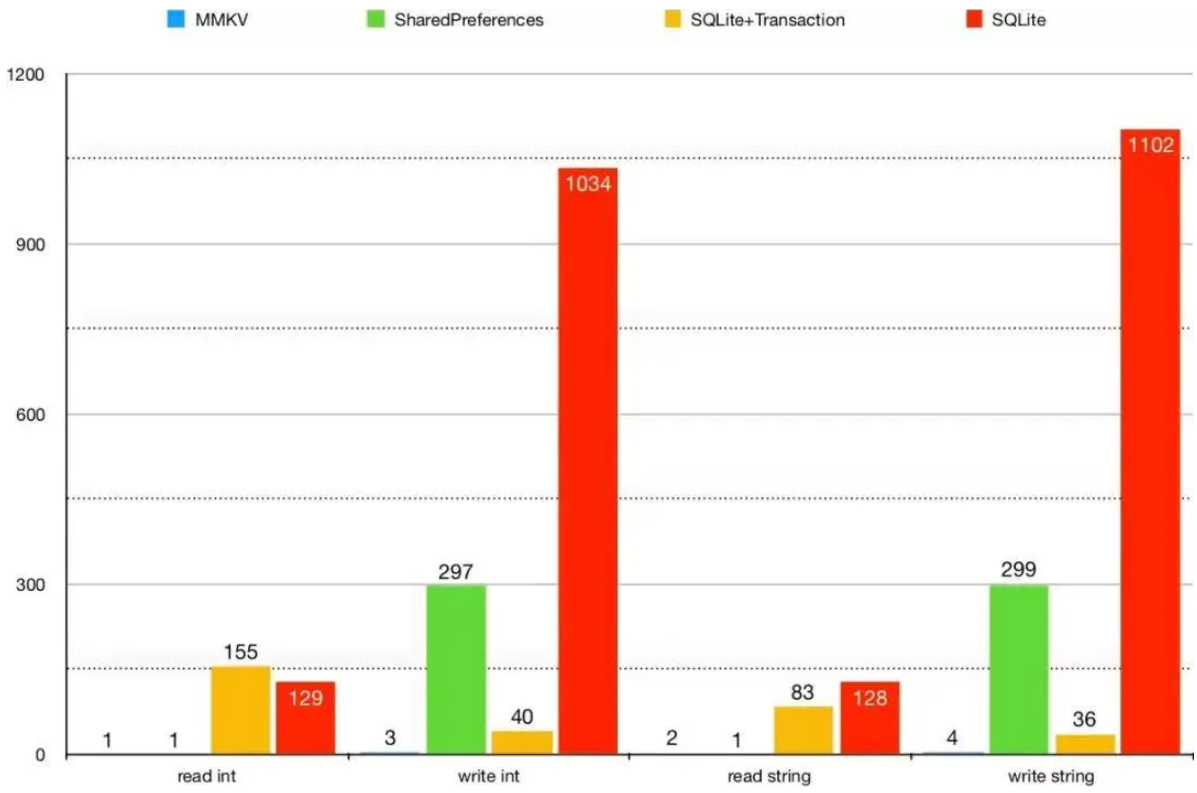
性能对比

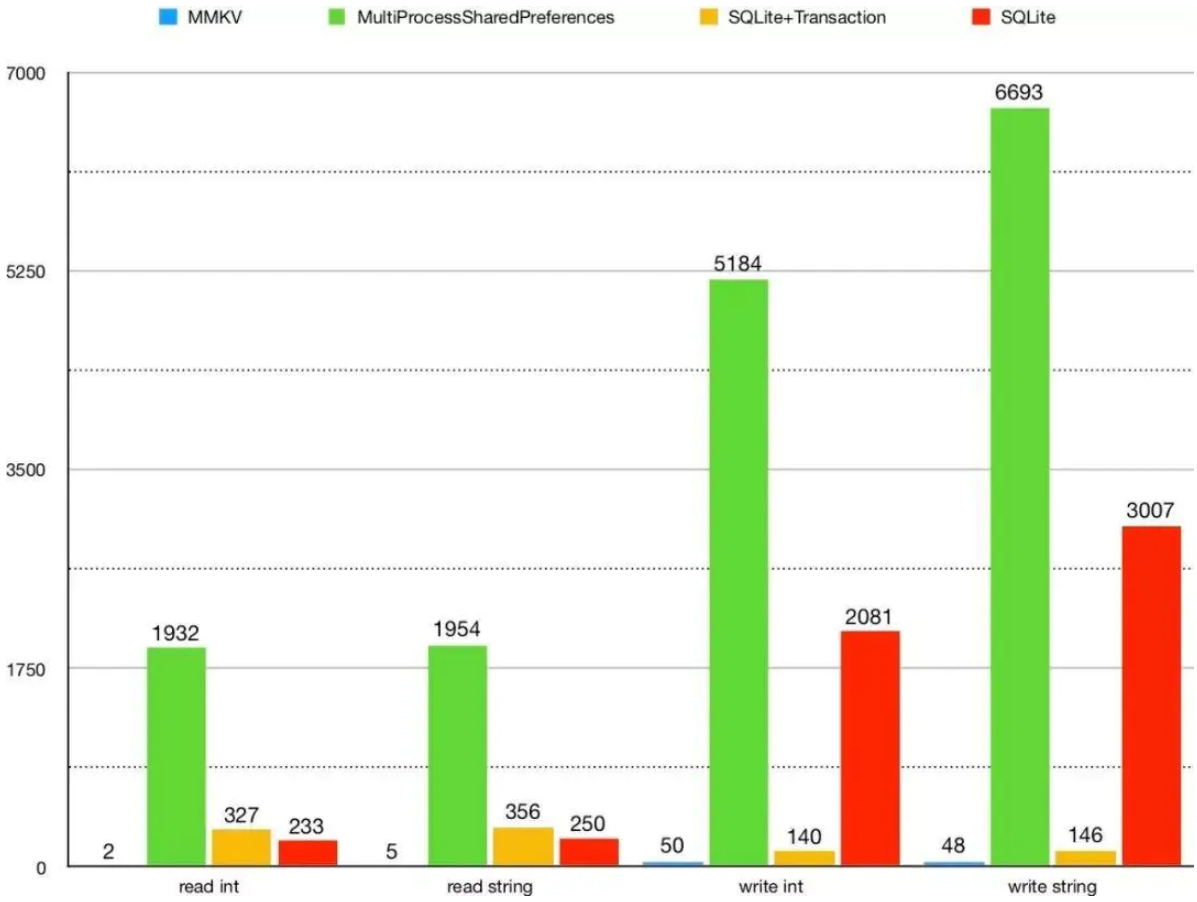
我们将 MMKV 和 SharedPreferences、SQLite 进行对比, 重复读写操作 1k 次。相关测试代码在 Android/MMKV/mmkvdemo/。结果如下图表。

单进程性能



可见，MMKV 在写入性能上远远超越 SharedPreferences & SQLite，在读取性能上也有相近或超越的表现。

多进程性能



可见，MMKV 无论是在写入性能还是在读取性能，都远远超越 MultiProcessSharedPreferences & SQLite & SQLite，
MMKV 在 Android 多进程 key-value 存储组件上是不二之选。

补充适用建议

如果使用请务必做code19版本的适配，这个在github官网有说明

依赖下面这个库，然后对19区分处理
implementation ‘com.getkeepsafe.relinker:relinker:1.3.1’

```
if (android.os.Build.VERSION.SDK_INT == 19) {
    MMKV.initialize(relativePath, new MMKV.LibLoader() {
        @Override
        public void loadLibrary(String libName) {
            ReLinker.loadLibrary(context, libName);
        }
    });
} else {
    MMKV.initialize(context);
}
```

限制

可看到，一个键会存入多分实例，最后存入的就是最新的。
MMKV 在大部分情况下都性能强劲，key/value 的数量和长度都没有限制。
然而 MMKV 在内存里缓存了所有的 key-value，在总大小比较大的情况下（例如 100M+），App 可能会爆内存，触发重整回写时，写入速度也会变慢。
支持大文件的 MMKV 正在开发中，有望在下一个大版本发布。

问题

数据变化监听 怎么获取?

```
// content change notification of other process
// trigger by getXXX() or setXXX() or checkContentChangedByOuterProcess()
```

//CallStaticVoidMethod 错误写成 CallStaticIntMethod，方法匹配crash

registerOnSharedPreferenceChangeListener not support
//官方推荐使用event方式通知更新
Data-change-listener is not supported by design.
We suggest using something like event-bus to notify any interesting clients.
Doing this inside a storage framework smells really bad.

defaultMMKV 是单进程SINGLE_PROCESS_MODE
使用MULTI_PROCESS_MODE创建多进程

带来的APK尺寸增加问题

libc++_shared.so 252.5k

libmmkv.so 43.5k

implementation 'com.tencent:mmkv:1.0.23'

//   implementation 'com.tencent:mmkv-static:1.0.23' (无libc++_shared.so)

只打包需要的平台对应.so

```
    ndk {
            abiFilters "armeabi-v7a", 'x86'
        }
```

.so加载问题

implementation 'com.getkeepsafe.relinker:relinker:1.3.1'

log太多

初始化可以设置log打印层级 initialize(rootDir, MMKVLogLevel.LevelInfo);
设置log转发，控制log输出格式、文件 MMKVHandler wantLogRedirecting=true

多进程

锁 lock unlock tryLock

注意如果一个进程lock住，另一个进程mmkvWithID获取MMKV时就阻塞住，直到持有进程释放。

```
        // get the lock immediately
        MMKV mmkv2 = MMKV.mmkvWithID(LOCK_PHASE_2, MMKV.MULTI_PROCESS_MODE);
        mmkv2.lock();
        Log.d("locked in child", LOCK_PHASE_2);

        Runnable waiter = new Runnable() {
            @Override
            public void run() {
                //阻塞住 直到其他进程释放
                MMKV mmkv1 = MMKV.mmkvWithID(LOCK_PHASE_1,
MMKV.MULTI_PROCESS_MODE);
                mmkv1.lock();
                Log.d("locked in child", LOCK_PHASE_1);
            }
        };
```

注意：如果其他进程有进行修改，不会立即触发onContentChangedByOuterProcess，
checkLoadData如果变化，会clearMemoryState，重新loadFromFile。//数据量大时不要太频繁
读取decodeXXX会阻塞住，先回调onContentChangedByOuterProcess，再返回值，保证值是最新的。

mmkvWithAshmemID 匿名共享内存

可以进行进程间通信，可设置pageSize
// a memory only MMKV, cleared on program exit
// size cannot change afterward (because ashmem won't allow it)

测试

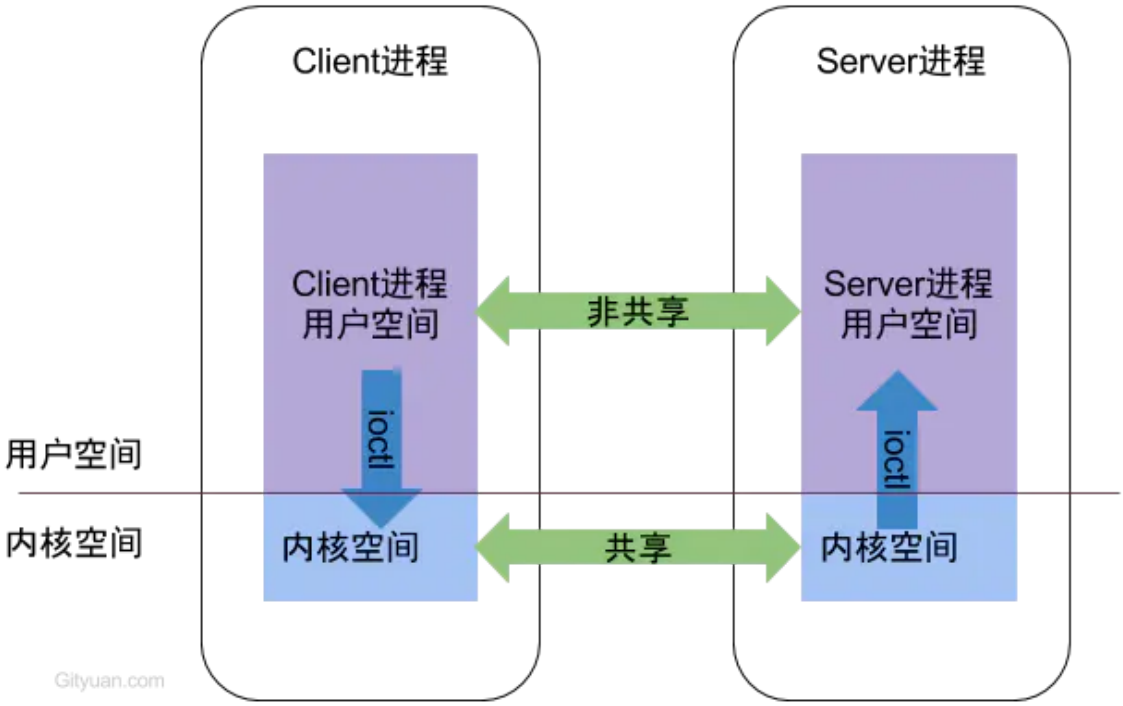
write速度 mmkv > cryptKV >> sp
read速度 sp > cryptKV > mmkv

Binder MMAP（一次拷贝）

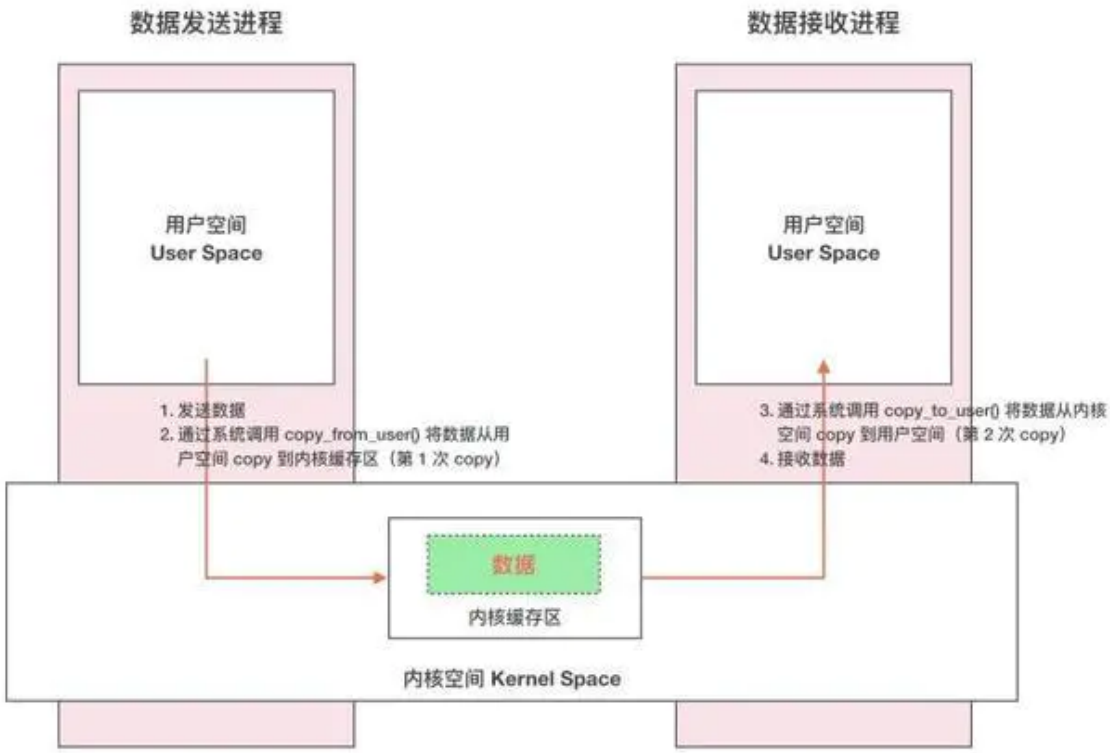
Linux的内存分用户空间跟内核空间，同时页表有也分两类，用户空间页表跟内核空间页表，每个进程有一个用户空间页表，但是系统只有一个内核空间页表。
而Binder mmap的关键是：更新用户空间对应的页表的同时也同步映射内核页表，让两个页表都指向同一块地址，
这样一来，数据只需要从A进程的用户空间，直接拷贝到B所对应的内核空间，而B多对应的内核空间在B进程的用户空间也有相应的映射，这样就无需从内核拷贝到用户空间了。

copy_from_user() //将数据从用户空间拷贝到内核空间
copy_to_user() //将数据从内核空间拷贝到用户空间

Liunx进程隔离

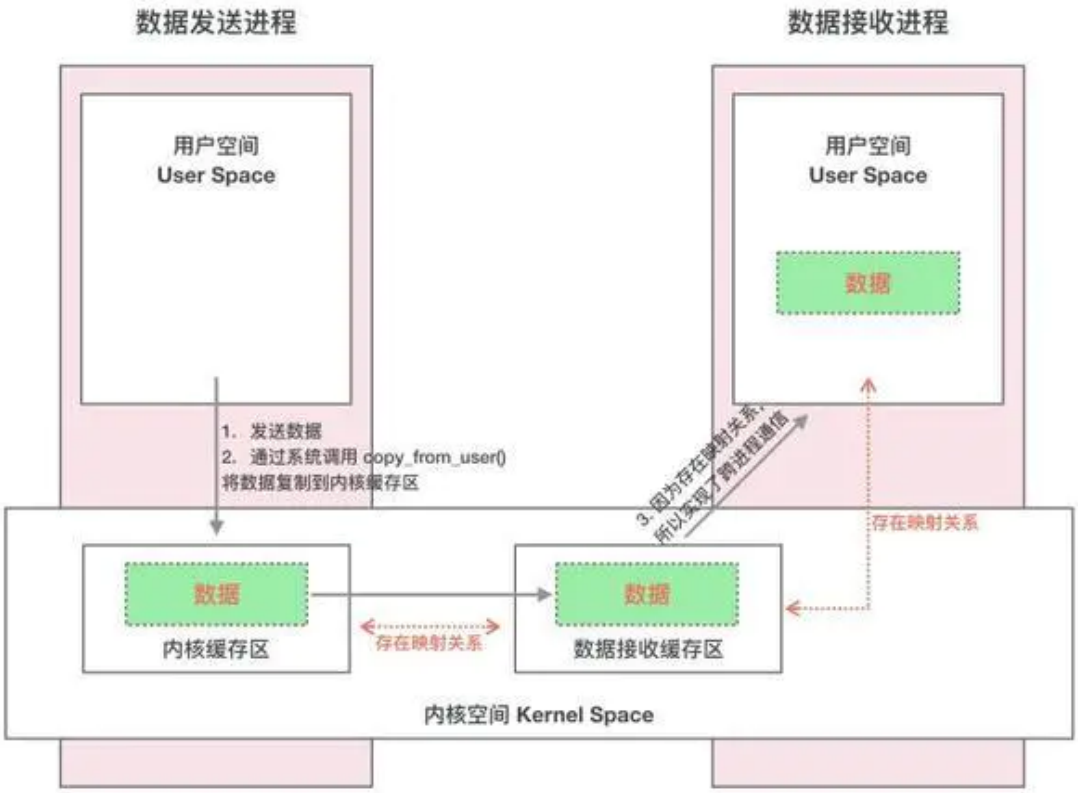


传统IPC



image

Binder通信



image

普通文件mmap原理

普通文件的访问方式有两种：

第一种是通过read/write系统调访问，先在用户空间分配一段buffer，然后，进入内核，将内容从磁盘读取到内核缓冲，最后，拷贝到用户进程空间，至少牵扯到两次数据拷贝；
同时，多个进程同时访问一个文件，每个进程都有一个副本，存在资源浪费的问题。

另一种是通过mmap来访问文件，mmap()将文件直接映射到用户空间，文件在mmap的时候，内存并未真正分配，
只有在第一次读取/写入的时候才会触发，这个时候，会引发缺页中断，在处理缺页中断的时候，完成内存也分配，同时也完成文件数据的拷贝。
并且，修改用户空间对应的页表，完成到物理内存到用户空间的映射，这种方式只存在一次数据拷贝，效率更高。
同时多进程间通过mmap共享文件数据的时候，仅需要一块物理内存就够了。

Android中使用mmap，可以通过RandomAccessFile与MappedByteBuffer来配合。
通过randomAccessFile.getChannel().map获取到MappedByteBuffer。然后调用ByteBuffer的put方法添加数据。

```
        RandomAccessFile randomAccessFile = new RandomAccessFile("path","rw");
        MappedByteBuffer mappedByteBuffer=
randomAccessFile.getChannel().map(FileChannel.MapMode.READ_WRITE,0,
randomAccessFile.length());

        mappedByteBuffer.putChar('c');
        mappedByteBuffer.getChar();
```

共享内存中mmap的使用

共享内存是在普通文件mmap的基础上实现的，其实就是基于tmpfs文件系统的普通mmap。