从RocketMQ的持久化谈起

探秘字节流

在《重学RocketMQ之深化理解与实践思考(一) 架构与消息》中我们讲到了刷盘，也就将数据从内存刷新到磁盘上，也就是持久化，提到持久化这里想到了Redis的持久化，MySQL的持久化。我想到了字节流和字符流、FileInputStream、FileOutputStream、BufferedOutputStream、BufferedInputStream、零拷贝。

字节流是用于处理字节级别读写操作的基础类，其中最常用的是FileInputStream和FileOutputStream。在Java中，所有字节流类都分别继承自InputStream和OutputStream这两个基类:

```
public void write(byte b[])
```

然后write方法底层是一个native调用:

```
private native void writeBytes(byte b[], int off, int len, boolean append)
```

Windows下面的实现

这里的native实现，我们看windows下面的调用，对应的实现是FileOutputStream_md.c，文件路径为:jdk/src/windows/native/java/io/FileOutputStream_md.c

在OpenJDK 8(jdk8-b115,后文不做说明, 统一在这个版本下讨论问题)的FileOutputStream_md.c我们可以看到对应的调用:

```
JNIEXPORT void JNICALL
Java_java_io_FileOutputStream_writeBytes(JNIEnv *env,
    jobject this, jbyteArray bytes, jint off, jint len, jboolean append) {
    writeBytes(env, this, bytes, off, len, append, fos_fd);
}
```

在io_util.c(文件路径为jdk/src/share/native/java/io/io_util.c)可以看到对应的writeByes调用

```
void writeBytes(JNIEnv *env, jobject this, jbyteArray bytes,
```

```
            jint off, jint len, jboolean append, jfieldID fid)

{
     if (len == 0) {
        return;
    } else if (len > BUF_SIZE) {
        buf = malloc(len);
        if (buf == NULL) {
            JNU_ThrowOutOfMemoryError(env, NULL);
            return;
        }
    } else {
        buf = stackBuf;
    }
    // 省略无关代码调用
    if (!(*env)->ExceptionOccurred(env)) {

            if (append == JNI_TRUE) {
                n = IO_Append(fd, buf+off, len);
            } else {
                n = IO_Write(fd, buf+off, len);
            }
            off += n;
            len -= n;
        }
    }
    if (buf != stackBuf) {
        free(buf);
    }
}
```

这里的IO_write是一个宏，在io_util_md.h(文件目标路径为:jdk/src/windows/native/java/io/io_util_md.c)里面可以看到对应的宏定义:

```
#define IO_Write handleWrite
```

在io_util_md.h(jdk/src/windows/native/java/io/io_util_md.h)我们可以看到handleWrite的定义:

```
JNIEXPORT jint handleWrite(jlong fd, const void *buf, jint len) {
    return writeInternal(fd, buf, len, JNI_FALSE);
}
static jint writeInternal(jlong fd, const void *buf, jint len, jboolean append)
{
    BOOL result = 0;
    DWORD written = 0;
    HANDLE h = (HANDLE)fd;
    if (h != INVALID_HANDLE_VALUE) {
        OVERLAPPED ov;
        LPOVERLAPPED lpOv;
        if (append == JNI_TRUE) {
            ov.Offset = (DWORD)0xFFFFFFFF;
            ov.OffsetHigh = (DWORD)0xFFFFFFFF;
            ov.hEvent = NULL;
            lpOv = &ov;
        } else {
            lpOv = NULL;
```

```
        }
        //
        result = WriteFile(h,               /* File handle to write */
                           buf,             /* pointers to the buffers */
                           len,             /* number of bytes to write */
                           &written,        /* receives number of bytes written
*/
                           lpOv);           /* overlapped struct */
    }
    if ((h == INVALID_HANDLE_VALUE) || (result == 0)) {
        return -1;
    }
    return (jint)written;
}
```

在参考文档[33] 里面可以看到，由于缺少了LPOVERLAPPED 参数，这是一个同步调用，但是也不代表会立即刷新到磁盘中, 这是因为内存到磁盘太慢了，默认情况下，Windows 缓存从磁盘读取的和写入到磁盘的文件数据。 这意味着读取操作从从系统内存中的某个区域（称为系统文件缓存的区域) 中读取文件数据，而不是从物理磁盘读取文件数据。 相应地，写入操作将文件数据写入系统文件缓存，而不是写入磁盘。这类缓存称为回写缓存。 缓存按文件对象进行管理(见参考文档[31]和[32])。

FileOutputStream的构造函数，对应到FileOutputStream_md.c(jdk/src/windows/native/java/io/FileOutputStream_md.c)的

```
NIEXPORT void JNICALL
Java_java_io_FileOutputStream_open(JNIEnv *env, jobject this,
                                   jstring path, jboolean append) {
    fileOpen(env, this, path, fos_fd,
             O_WRONLY | O_CREAT | (append ? O_APPEND : O_TRUNC));
}
```

fileOpen函数的实现在io_util_md.c(jdk/src/windows/native/java/io/io_util_md.c)中:

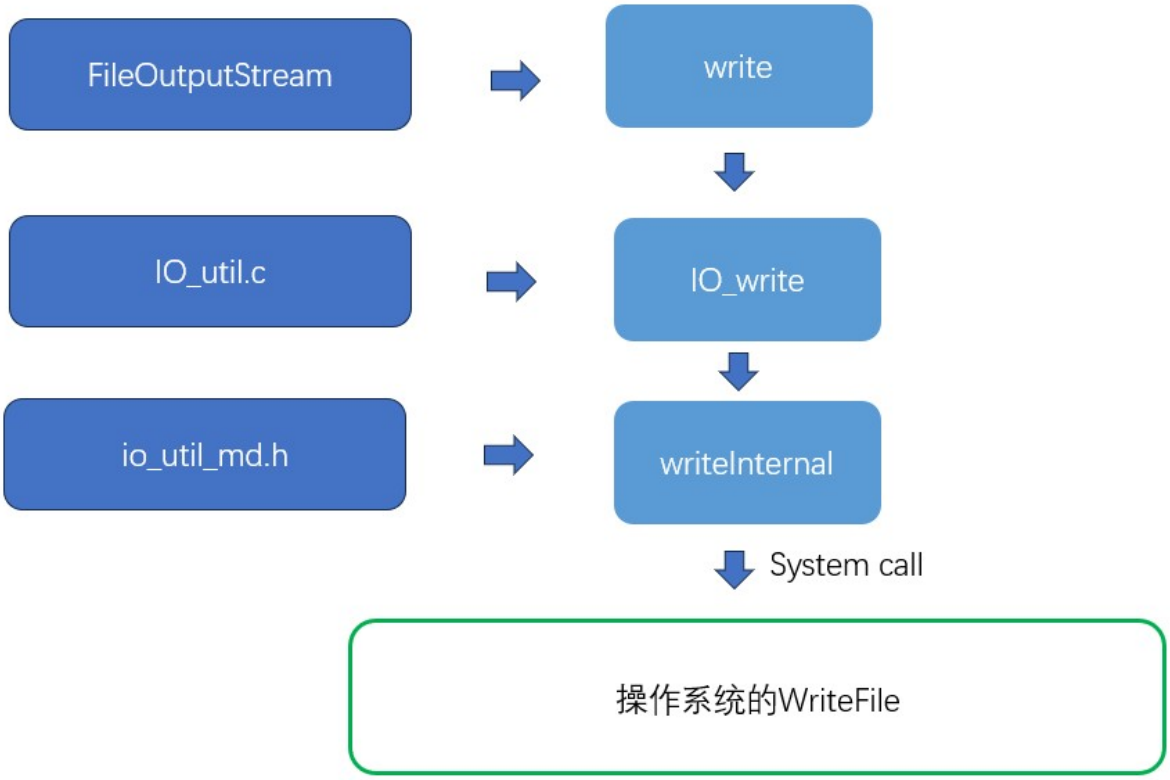
```
void fileOpen(JNIEnv *env, jobject this, jstring path, jfieldID fid, int flags)
{
    jlong h = winFileHandleOpen(env, path, flags);
    if (h >= 0) {
        SET_FD(this, h, fid);
    }
}
```

```
jlong winFileHandleOpen(JNIEnv *env, jstring path, int flags)
{
    const DWORD access =
        (flags & O_WRONLY) ?  GENERIC_WRITE :
        (flags & O_RDWR)   ? (GENERIC_READ | GENERIC_WRITE) :
        GENERIC_READ;
    const DWORD sharing =
        FILE_SHARE_READ | FILE_SHARE_WRITE;
    const DWORD disposition =
        /* Note: O_TRUNC overrides O_CREAT */
        (flags & O_TRUNC) ? CREATE_ALWAYS :
        (flags & O_CREAT) ? OPEN_ALWAYS   :
        OPEN_EXISTING;
    const DWORD  maybeWriteThrough =
```

```
        (flags & (O_SYNC | O_DSYNC)) ?
        FILE_FLAG_WRITE_THROUGH :
        FILE_ATTRIBUTE_NORMAL;
    const DWORD maybeDeleteOnClose =
        (flags & O_TEMPORARY) ?
        FILE_FLAG_DELETE_ON_CLOSE :
        FILE_ATTRIBUTE_NORMAL;
    const DWORD flagsAndAttributes = maybeWriteThrough | maybeDeleteOnClose;
    HANDLE h = NULL;

    if (onNT) {
        WCHAR *pathbuf = pathToNTPath(env, path, JNI_TRUE);
        if (pathbuf == NULL) {
            /* Exception already pending */
            return -1;
        }
        h = CreateFileW(
            pathbuf,            /* Wide char path name */
            access,             /* Read and/or write permission */
            sharing,            /* File sharing flags */
            NULL,               /* Security attributes */
            disposition,        /* creation disposition */
            flagsAndAttributes, /* flags and attributes */
            NULL);
        free(pathbuf);
    } else {
        WITH_PLATFORM_STRING(env, path, _ps) {
            h = CreateFile(_ps, access, sharing, NULL, disposition,
                           flagsAndAttributes, NULL);
        } END_PLATFORM_STRING(env, _ps);
    }
    if (h == INVALID_HANDLE_VALUE) {
        int error = GetLastError();
        if (error == ERROR_TOO_MANY_OPEN_FILES) {
            JNU_ThrowByName(env, JNU_JAVAIOPKG "IOException",
                            "Too many open files");
            return -1;
        }
        throwFileNotFoundException(env, path);
        return -1;
    }
    return (jlong) h;
}
```

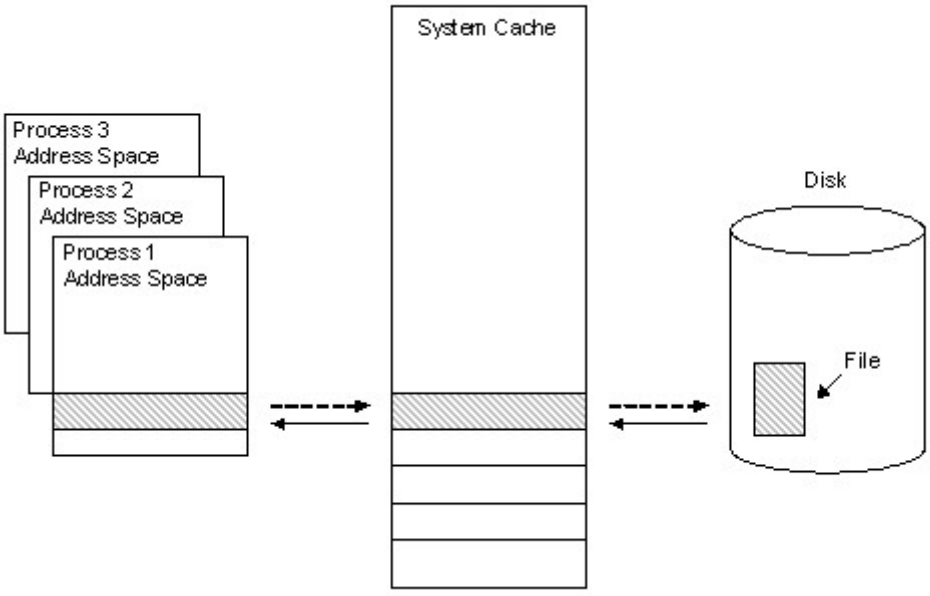
windows下面的调用链路图如下所示:



所谓系统调用是一种特殊的函数(一般由操作系统提供)，它允许你跨越保护域。当一个程序在用户态(用户模式下)执行的时候，它不被允许执行在内核态下面运行的代码所允许的操作。例如在用户态下面的程序无法在没有内核帮助的情况下读取文件。当用户程序向操作系统请求服务的时候，系统通过系统调用保护自己不受恶意或有缺陷程序的影响。系统调用执行一条特殊的硬件指令，通常成为"陷阱"，将控制权转移到内核，然后内核决定是否满足该请求。

缓存管理器

在对应的函数中我们可以看到应该是开启了缓存(应该表示推测，那段代码适配了windows的多个版本，见在参考文档[9])，也就是说写入到磁盘的数据首先到操作系统缓冲区，然后再由缓冲管理器刷新到磁盘中:



读取操作从系统内存中的某个区域（称为系统文件缓存）读取文件数据，而不是从物理磁盘读取。

参考文档[31]这么说道:

> 某些应用程序（如病毒检查软件）要求其写入操作立即刷新到磁盘;Windows 通过写通缓存提供此功能。 进程通过将 FILE_FLAG_WRITE_THROUGH 标志传递到对 CreateFile 的调用中，为特定 I/O 操作启用写通缓存。 启用写通缓存后，数据仍会写入缓存，但缓存管理器会立即将数据写入磁盘，而不会因使用延迟编写器而产生延迟。 进程还可以通过调用 FlushFileBuffers 函数强制刷新已打开的文件。

上面Windows的缓存管理器让我想起来Linux的page cache(见参考文档[17]):

> The Linux kernel implements a disk cache called the `page cache`. The goal of this cache is to minimize disk I/O by storing data in physical memory that would otherwise require disk access.
>
> Linux内核实现了磁盘缓存称之为页面缓存，设计目标是通过将数据存储在物理内存中来减少磁盘I/O。

Linux下面的实现

通过上面的分析，我们了解了Windows系统下IO的实现机制。不同的操作系统在IO实现上往往有其特殊的考虑，让我们看看Linux系统是如何处理这个问题的

在Linux下面对应的实现在jdk/src/solaris/native/java/io/FileOutputStream_md.c中:

```
JNIEXPORT void JNICALL
Java_java_io_FileOutputStream_writeBytes(JNIEnv *env,
    jobject this, jbyteArray bytes, jint off, jint len, jboolean append) {
    writeBytes(env, this, bytes, off, len, append, fos_fd);
}
```

对应的实现是(jdk/src/share/native/java/io/io_util.c)

```
#define BUF_SIZE 8192
void writeBytes(JNIEnv *env, jobject this, jbyteArray bytes,
           jint off, jint len, jboolean append, jfieldID fid)
{
    // 省略无关代码
    if (!(*env)->ExceptionOccurred(env)) {
        off = 0;
        while (len > 0) {
            fd = GET_FD(this, fid);
            if (fd == -1) {
                JNU_ThrowIOException(env, "Stream Closed");
                break;
            }
            if (append == JNI_TRUE) {
                n = IO_Append(fd, buf+off, len);
            } else {
                n = IO_Write(fd, buf+off, len);
            }
            if (n == JVM_IO_ERR) {
                JNU_ThrowIOExceptionWithLastError(env, "Write error");
                break;
            } else if (n == JVM_IO_INTR) {
                JNU_ThrowByName(env, "java/io/InterruptedIOException", NULL);
                break;
            }
            off += n;
            len -= n;
```

```
        }
    }
    if (buf != stackBuf) {
        free(buf);
    }
}
```

在jdk/src/solaris/native/java/io/io_util_md.h看到这个IO_Write也是一个宏定义：

```
#define IO_Write JVM_Write
```

这个JVM_Write在hotspot的JVM.cpp中:

```
JVM_LEAF(jint, JVM_Write(jint fd, char *buf, jint nbytes))
    JVMWrapper2("JVM_Write (0x%x)", fd);
    //%note jvm_r6
    return (jint)os::write(fd, buf, nbytes);
JVM_END
```

对应Linux 的实现在(hotspot/src/os/linux/vm/os_linux.inline.hpp)中:

```
#include <unistd.h>
inline size_t os::write(int fd, const void *buf, unsigned int nBytes) {
  size_t res;
  RESTARTABLE((size_t) ::write(fd, buf, (size_t) nBytes), res);
  return res;
}
```

这里其实调用的是Linux的write函数, Linux中write的调用说明为:

> write() writes up to count bytes from the buffer starting at buf to the file referred to by the file descriptor fd.
>
> - `write()` 函数尝试将最多 `count` 个字节从 `buf` 指向的缓冲区写入到文件描述符 `fd` 所引用的文件中。

> A successful return from write() does not make any guarantee that data has been committed to disk. On some filesystems, including NFS, it does not even guarantee that space has successfully been reserved for the data. In this case, some errors might be delayed until a future write(), fsync(2), or even close(2).
>
> 成功调用并不承诺数据到达磁盘，在一些文件系统上，包括NFS，甚至不保证已为数据成功预留了空间。某些错误可能会延迟到未来的 `write()`、`fsync(2)` 或甚至 `close(2)` 调用时才显现。

> The only way to be sure is to call fsync(2) after you are done writing all your data.
>
> 确保数据已写入磁盘的唯一方法是在完成所有数据写入后调用 `fsync(2)`

这里有两个问题，第一个问题在Java中如何将数据确保刷到磁盘上，第二个问题既然没刷到磁盘，这个数据在调用返回之后在哪个位置?

如何将数据确保刷新到磁盘上

- 方式一

```
try(FileOutputStream fileOutputStream = new FileOutputStream("D:\\学习资料
\\SDK\\1.txt");){
    fileOutputStream.write("hello world".getBytes());
    fileOutputStream.getFD().sync();
}catch (Exception e) {
    e.printStackTrace();
}
```

最终调用的是FileDescriptor的sync方法，这同样也是一个系统调用：

```
public native void sync() throws SyncFailedException;
```

对应的实现位于jdk/src/solaris/native/java/io/FileDescriptor_md.c下面:

```
JNIEXPORT void JNICALL
Java_java_io_FileDescriptor_sync(JNIEnv *env, jobject this) {
    int fd = (*env)->GetIntField(env, this, IO_fd_fdID);
    if (JVM_Sync(fd) == -1) {
        JNU_ThrowByName(env, "java/io/SyncFailedException", "sync failed");
    }
}
```

JVM_syn的实现在hotspot/src/share/vm/prims/jvm.cpp:

```
JVM_LEAF(jint, JVM_Sync(jint fd))
  JVMWrapper2("JVM_Sync (0x%x)", fd);
  //%note jvm_r6
  return os::fsync(fd);
JVM_END
```

Linux上的实现

然后分配到对应操作系统的调用上，在Linux位于:hotspot/src/os/linux/vm/os_linux.inline.hpp

```
inline int os::fsync(int fd) {
  return ::fsync(fd);
}
```

fsync是一个Linux系统调用将对文件的更改写入内容刷新到磁盘上，由此这让我想起了Redis的AOF持久化机制，那么也要通过Linux的系统调用来将内容刷新到磁盘上，

Windows的实现

在windows上的实现为:

```
// This code is a copy of JDK's sysSync
// from src/windows/hpi/src/sys_api_md.c
// except for the legacy workaround for a bug in Win 98
```

```
int os::fsync(int fd) {
  HANDLE handle = (HANDLE)::_get_osfhandle(fd);

  if ( (!::FlushFileBuffers(handle)) &&
         (GetLastError() != ERROR_ACCESS_DENIED) ) {
    /* from winerror.h */
    return -1;
  }
  return 0;
}
```

FlushFileBuffers 是windows的函数作用为将缓冲区的内容刷新到磁盘上。

通过Channel刷

```
try (FileChannel fileChannel = new FileOutputStream(new File("")).getChannel()){
    ByteBuffer buffer = ByteBuffer.wrap("hello".getBytes());
    fileChannel.write(buffer);
    fileChannel.force(true);
} catch (FileNotFoundException e) {
    throw new RuntimeException(e);
} catch (IOException e) {
    throw new RuntimeException(e);
}
```

数据刷新到了哪里

其实在上面的讨论中这个问题已经呼之欲出了，在Linux中是page cache(页面缓存), 我们姑且可以理解为Linux的缓存管理器，包含两个方面一个是页面，一个是页面管理策略。 我们知道在Linux内核中将物理内存分成一页一页进行管理(每页4K大小)，根据存储的数据来源不同，我们可以将其分为两类：

- 文件页:文件页中的数据来自磁盘中的文件，当我们进行文件读取的时候，内核会根据局部性原理将读取的磁盘数据缓存在page cache中，page cache里面存放的就是文件页。当进程再次读取文件页中的数据时，内核直接会从page cache中获取并拷贝给进程。省去了读取磁盘的开销。

  所谓局部性原理也就是说: 内存中的数据一旦被访问，那么它很有可能在短期内被再次访问。这是一种假设，一种先验预测。

- 匿名页: 所谓匿名页就是它背后并没有一个磁盘中的文件作为数据来源，匿名页中的数据都是通过进程运行过程中产生的，比如我们应用程序中动态分配的堆内存。

现在为止我们知道数据刷新到了哪里，也就是说在没调用同步操作之前我们的写操作还停留在内存里面，当页面缓存中的数据比磁盘中的数据更新时，我们称页面缓存中的数据为“脏数据”，注意我们以页为单位管理数据，所以这些记录了更新数据的文件页被称为脏页，在下面三种情况发生时会触发脏页回写:

- 当可用内存降低到指定阈值以下的时候，数据变干净之后可以从缓存中驱逐，缓存可以收缩，释放更多内存。
- 当脏数据超过特定时间阈值时，足够“老”的数据会被写会磁盘，确保脏数据不会无限期保持脏状态。
- 用户进程调用sync()和fsync()系统调用时， 内核按需执行写会操作，这是主动触发的写会机制。

当内存紧张需要对不经常使用的那些匿名页进行回收时，需要将匿名页的数据线保存在磁盘空间，然后对匿名页进行回收。并把释放出来的这部分内存分配给更需要的进程使用，当进程再次访问这块内存时，会重新把之前匿名页中的数据从磁盘空间中读取到内存就可以了，而这块磁盘空间可以是单独的一片磁盘分区(Swap 分区)或者是一个特殊的文件，这也就是swap机制。

那当内存紧张的时候，内核到底是该回收文件页呢？ 还是该回收匿名页？ Linux提供了一个swappiness的内核选项，我们可以通过cat /proc/sys/vm/swappiness命令查看，swappiness选项的取值范围为0到100，默认为60。

swappiness用于表示Swap机制的积极程度，数值越大，Swap的积极程度越高，内核越倾向于回收匿名页。数值越小，Swap的积极程度越低，内核就越倾向于回收文件页。注意： swappiness 只是表示 Swap 积极的程度，当内存非常紧张的时候，即使将 swappiness 设置为 0 ， 也还是会发生 Swap 的。

Page Cache的局限性

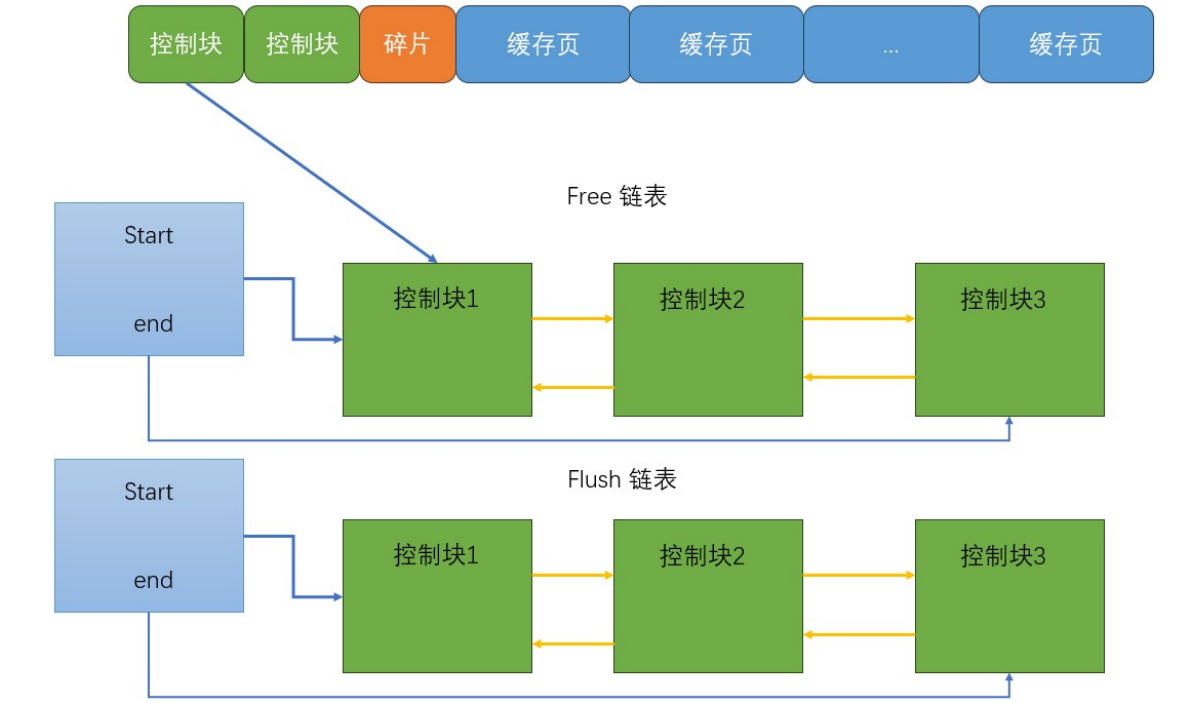
既然有了page cache 为什么关系型数据库还要自己做缓存？考虑操作系统面对的程序不是为一个程序来设计，所以做的是通用的缓存逻辑，对于应用自身要做特化逻辑。 以关系型数据库为例, 为了提升查询速度都引入了缓冲池，在MySQL里面我们知道这被称为buffer pool，我们知道MySQL中的buffer pool中的默认缓存页大小和磁盘上默认的页大小是一样的，都是16KB。那为什么是16KB呢？ MySQL官方是这么回答的：

> Consider using a page size that matches the internal sector size of the disk. Early-generation SSD devices often have a 4KB sector size. Some newer devices have a 16KB sector size. The default `InnoDB` page size is 16KB. Keeping the page size close to the storage device block size minimizes the amount of unchanged data that is rewritten to disk.
>
> 考虑使用与磁盘内部扇区大小相匹配的页面大小。早期的SSD设备通常有4KB的扇区大小。一些较新的设备具有16KB的扇区大小。InnoDB的默认页面大小是16KB。保持页面大小接近存储设备的块大小可以最小化重写到磁盘的未更改数据量。

理想情况下页的大小应该和存储设备的存储块大小是相互匹配的，比如页面大小是16KB，记录大小为100字节，那么我只想改记录中的一部分，修改的流程即为读16KB修改其中的内容，然后回写磁盘，一次物理写入。如果是页面大小16KB，块4KB，则需要四次回写，虽然我们可能只改了一块，但是数据库的页信息要随之变动。除此之外，还有时间局部性和空间局部性假设:

> 程序局部性原理表现为：时间局部性和空间局部性。时间局部性是指如果程序中的某条指令一旦执行，则不久之后该指令可能再次被执行；如果某块数据被访问，则不久之后该数据可能再次被访问。空间局部性是指一旦程序访问了某个存储单元，则不久之后，其附近的存储单元也将被访问。

因此如果遇到了页面大小和存储设备块大小不一致的情况，我们需要改写页面大小来最小化重写到磁盘的未更改数据量。我们接着回忆buffer pool管理策略和具体组成就会发现page cache 不适应需求的地方。为了更好的管理缓存页，MySQL给每个缓存页创建了一些所谓的控制信息，这些控制信息包含该页所属的表空间编号、页号等等。如下图所示:



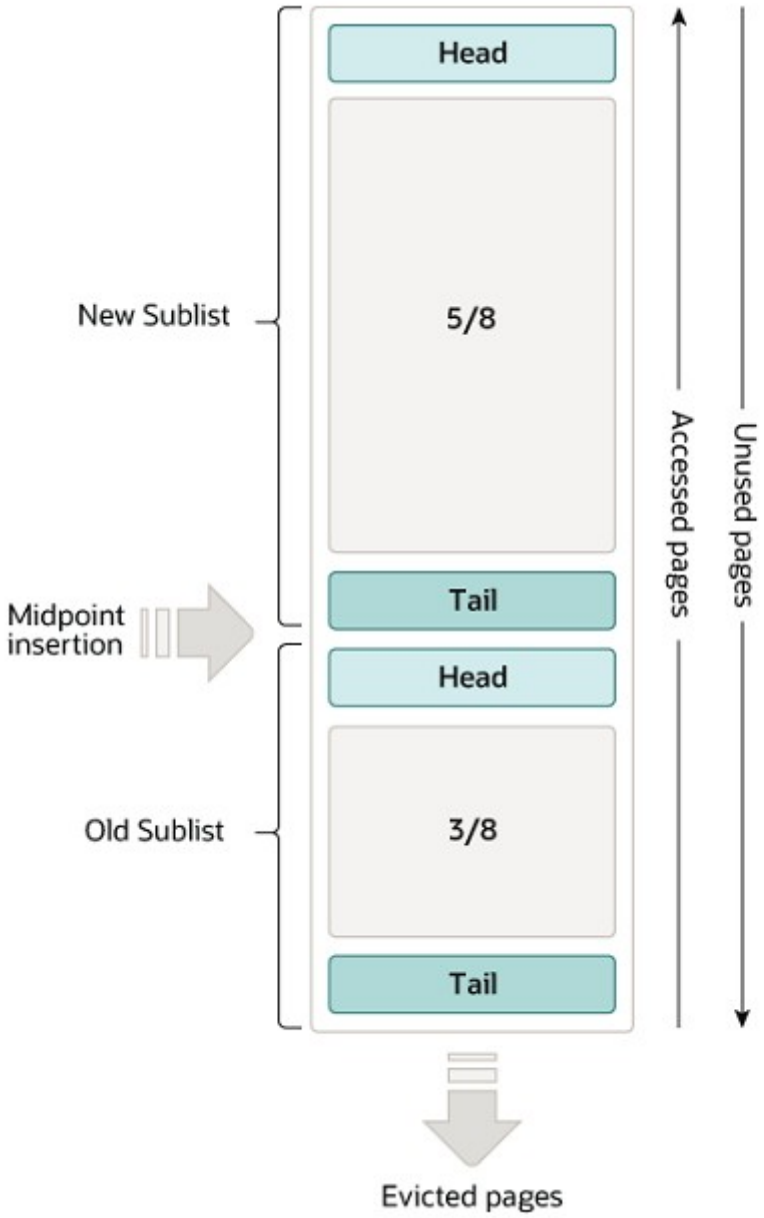
Free链表存储的是空闲的内存块，每当需要从磁盘中加载一个页到Buffer Pool中的时候，就从free链表中取一个空闲的缓存页，并且把该缓存页中对应的控制块信息填上(就是该页所在的表空间、页号质量的信息)，然后把缓存页对应的free链表节点从链表中移除，表示该缓存页已经被使用了。

Flush链表存储的是脏页，所谓脏页也就是缓存页比磁盘上的数据页要新的缓存页，那么如何维持一致呢，最简单的方法就是发生一次修改立即同步到磁盘上对应的页上，但频繁写入磁盘会严重影响程序性能。如果不采用即时同步策略，我们就需要一种机制来追踪Buffer Pool中的页面状态，区分哪些是脏页（已修改），哪些是未修改的页面。 这也就是引入Flush链表的原因，凡是修改过的缓存页对应的控制块都会作为一个节点加入到一个链表中，这个链表对应的缓存页都是需要被刷新到磁盘上的，所以也叫flush链表。

现在考虑怎么有效的利用缓存页，非常简单的一个策略是LRU, 也就是按照按照最近最少使用策略，为没有缓存页使用的时候淘汰最少使用的缓存页，于是我们就可以这么访问链表:

- 如果该页不在 buffer pool 中，在把该页从磁盘加载到 buffer pool 中的缓存页时，就把该缓存页对应的`控制块`作为节点塞到链表的头部。
- 如果该页已经缓存在buffer pool 中，则直接把该页对应的控制块移动到LRU链表的头部

那现在考虑我不小心采用了全表扫描语句，那这意味着innodb辛苦加热的数据页都要被换出？ 这会严重影响其他查询使用buffer pool的效率。InnoDB选择将这个LRU链表裁成两段，一部分存储使用频率非常高的缓存页，这一部分也叫热数据，一部分存储频率不高的缓存页，这一部分也叫冷数据。



我们简单的介绍一下上面LRU List链表的管理策略(BUF_LRU_OLD_MIN_LEN)，在逻辑上被分为两部分，前面部分存储最热的数据页，这部分链表被称为young list，后面部分则存储冷数据页，这部分被称为old list，一旦Free List 中没有页面了，就会从冷页面中驱逐，两部分长度的参数由buf_LRU_old_adjust_len控制。每次加入或者驱逐一个数据页之后，都要调整young list 和 old list的长度(buf_LRU_old_adjust_len), 同时引入BUF_LRU_OLD_TOLERANCE来防止链表调整过于频繁，新读取进来的页面默认被放在old list头，在经过innodb_old_blocks_time后，如果再次被访问了，就挪到young list头上。一个数据页被读入Buffer Pool后，在小于innodb_old_blocks_time的时间内被访问了很多次，之后就不再被访问了，这样的数据页也很快被驱逐。这个设计认为这种数据页是不健康的，应该被驱逐。此外，如果一个数据页已经处于young list，当它再次被访问的时候，不会无条件的移动到young list头上，只有当其处于young list长度的1/4(大约值)之后，才会被移动到young list头部，这样做的目的是减少对LRU List的修改，否则每访问一个数据页就要修改链表一次，效率会很低，因为LRU List的根本目的是保证经常被访问的数据页不会被驱逐出去，因此只需要保证这些热点数据页在头部一个可控的范围内即可。

事实上，在内存回收这一块，很多时候都是 empirical（经验主义）的，根据实际效果的好坏差异来选用合适的机制。

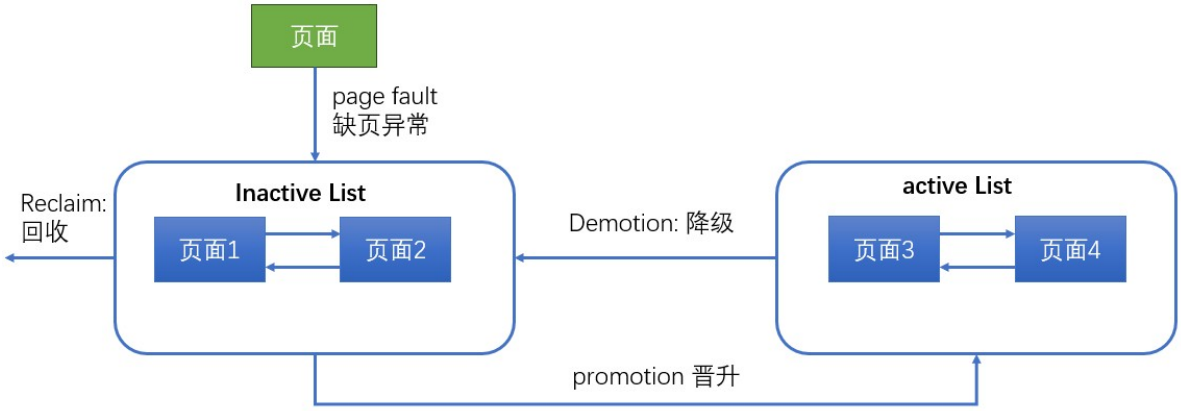
对于全表扫描这种场景，在对某个处在`old`区域的缓存页进行第一次访问时就在它对应的控制块中记录下来这个访问时间，如果后续的访问时间与第一次访问的时间在某个时间间隔内，那么该页面就不会被从old区域移动到young区域的头部，否则将它移动到young区域的头部。上述的这个间隔时间是由系统变量`innodb_old_blocks_time`控制的。 那这里我就想了，我在一段时间里面密集访问，怎么就不是

热点数据了呢，我想这应该也基于一种预设，MySQL的定位是OLTP(联机事务处理)，对于OLTP系统来说，真正的热点数据是访问是分散的，是跨越innodb_old_blocks_time这个时间窗口的，是有一定的业务间隔的，innodb_old_blocks_time的默认值是1s，在小于这个时间间隔频繁访问的页面就像你临时频繁的翻阅某本课外书一样，真正的热点书籍，应当是你的教科书，这些教科书会相隔一定的间隔被翻开。

> OLTP 或联机事务处理是一种数据处理类型，包括执行多个并发的事务，例如网上银行、购物、订单输入或发送文本消息。这些事务传统上被称为经济或财务事务，会被记录并加以保护，帮助企业随时访问这些信息，以用于会计或报告目的。过去，OLTP 仅限于交换金钱、产品、信息、服务请求等某些东西的实际交互。但多年来，这种情景下事务的定义已经扩大，尤其是自互联网问世以来，它涵盖了可以从世界上任何地方，通过任何网络连接的传感器触发的任何类型数字交互或与企业的互动。它还包括任何类型的交互或操作，例如在网页上下载 PDF、查看特定视频或社交渠道上的自动维护触发器或备注；这些触发器或评论可能对于企业记录以更好地为客户提供服务至关重要。

MySQL还实现了更为精细的预读控制机制（限于篇幅，这里不做详述）。相比之下，Linux的page cache机制则相对简单:

- active list: 活跃链表
- inactive list: 不活跃链表



当内核需要分配内存或加载磁盘文件时，会首先触发缺页中断（page fault）。新加载的页面会被放入Inactive链表。如果这些页面被多次访问，则会被提升到Active链表中。 当Active 链表过大的时候，当Active 链表增长的过大，活跃页面会被降级到非活跃链表上。

我们这里简单的介绍一下页面从InActive List 到 Active List的流程，在内存页面里面有两个标志位: **PG_referenced**、**PG_active** 。PG_active决定页面在哪个链表，也就是说 active list 中的 pages 的 PG_active 都为 1，而 inactive list 中的 pages 的 PG_active 都为 0。PG_referenced 标志位则是表明 page 最近是否被使用过。

如果inactive List上PG_referenced为1的page在回收之前被再次访问到，也就是说它在Inactive list中被访问了2次，那么就会通过active_page()被调整到active list的头部，同时将其PG_active位设置为1，PG_referenced设置为0。可以理解为两个PG_referenced才换来一个PG_Active. 这里就是内核相对粗糙的地方，只根据次数，没有根据时间频率，不太能满足MySQL的需要。

注意这个page fault，这是不可控的，这比一般的上下文切换高的多，当内存紧张的时候，应用依赖的页面就会被换出(后面我们会介绍Linux的新特性，可以锁住一些页面)，性能会下降的厉害，这让我想到了概率论中期望这个概念:

> 概率论里面一个非常重要的概念就是随机变量的期望，如果X是一个离散型的随机变量，如果X是一个离散型的随机变量，并且具有分布列p(x)，那么X 的期望或者期望者记为E[X]， 定义如下:

$$E[X] = \sum_{x:p(x)>0} xp(x)$$

也就是说X的期望值就是X所有可能取值的一个加权平均，每个值的权重就是X取该值的概率。而MySQL选择的Buffer Pool 则颠簸的风险更小一些，相对更为可控。

探秘缓冲流

在上面的调用中假如我们频繁写入，就会经历多次上下文切换，每次上下文切换都需要成本，参考文档中[16] 中我们可以看到在X86上切换的成本为100ns, 除了这个成本以外，如果每次写入的字节比较小，这就像一辆车没有满载，对系统调用的利用率就不够高。那我们能不能像快递站一样到一个快递就开始派送，而是快递员装满某个小区的快递之后才开始配送。这也就是缓冲流的设计思路。

在oracle写的教程中《**The Java™ Tutorials** 》(见参考文档8)中对缓冲流是这么说道:

> Most of the examples we've seen so far use *unbuffered* I/O. This means each read or write request is handled directly by the underlying OS. This can make a program much less efficient, since each such request often triggers disk access, network activity, or some other operation that is relatively expensive.
>
> 到目前为止，我们看到的大多数例子都使用了非缓冲I/O。这意味着每个读取或写入请求都直接由底层操作系统处理。这可能使程序效率大大降低，因为每个这样的请求通常会触发磁盘访问、网络活动或其他相对昂贵的操作。
>
> To reduce this kind of overhead, the Java platform implements *buffered* I/O streams. Buffered input streams read data from a memory area known as a *buffer*; the native input API is called only when the buffer is empty. Similarly, buffered output streams write data to a buffer, and the native output API is called only when the buffer is full.
>
> 为了减少这种开销，Java平台实现了缓冲I/O流。缓冲输入流从称为缓冲区的内存区域读取数据；只有当缓冲区为空时，才会调用原生输入API。类似地，缓冲输出流将数据写入缓冲区，只有当缓冲区满时，才会调用原生输出API。

在我刚学IO流的时候，老师在讲这个缓冲流的时候，说道你就将这个理解为用了一辆小推车去运输数据，所以快了很多，这种解释并不接近本质，如果沿用这种思路去解释原理，那么再遇到零拷贝的时候，那解释是不是要上航天飞机，现在我们直接从源码的级别去看缓冲流的设计思路:

```
protected byte buf[];

public BufferedOutputStream(OutputStream out) {
    this(out, 8192);
}
```

```
public BufferedOutputStream(OutputStream out, int size) {
    super(out);
    if (size <= 0) {
        throw new IllegalArgumentException("Buffer size <= 0");
    }
    buf = new byte[size];
}
```

```
public synchronized void write(byte b[], int off, int len) throws IOException {
    if (len >= buf.length) {
        /* If the request length exceeds the size of the output buffer,
           flush the output buffer and then write the data directly.
           In this way buffered streams will cascade harmlessly. */
        flushBuffer();
        out.write(b, off, len);
        return;
```

```
    }
    if (len > buf.length - count) {
        flushBuffer();
    }
    System.arraycopy(b, off, buf, count, len);
    count += len;
}
```

从源码中我们可以看到， 使用缓冲流再刷数据的时候，BufferedOutputStream内置了一个字节数组，写入的时候，首先判断传入的字节数组要写入的大小超没超过内置字节数组的大小，如果超过了，先将内置的字节数组通过系统调用刷到系统缓存里面，然后再将传入的字节数组写入。如果内置的字节数组剩余的容量小于要写入的字节大小，则刷缓存，然后将传入的数组复制到内置的缓冲区中。这种设计思路有点像是，单个插入转批量插入。在《译: 通过零拷贝实现高效数据传输》我们提到，一些Web应用提供大量的静态资源，这相当于从磁盘读取大量数据，然后通过Socket将这些数据回写，在Java中也就就是SocketOutputStream的write方法,

```
// 将数据从缓冲区刷新到磁盘上
public synchronized void flush() throws IOException
public void write(byte b[], int off, int len) throws IOException {
    socketWrite(b, off, len);
}
```

```
private void socketWrite(byte b[], int off, int len) throws IOException {


    if (len <= 0 || off < 0 || len > b.length - off) {
        if (len == 0) {
            return;
        }
        throw new ArrayIndexOutOfBoundsException("len == " + len
                + " off == " + off + " buffer length == " + b.length);
    }

    FileDescriptor fd = impl.acquireFD();
    try {
        socketWrite0(fd, b, off, len);
    } catch (SocketException se) {
        if (se instanceof sun.net.ConnectionResetException) {
            impl.setConnectionResetPending();
            se = new SocketException("Connection reset");
        }
        if (impl.isClosedOrPending()) {
            throw new SocketException("Socket closed");
        } else {
            throw se;
        }
    } finally {
        impl.releaseFD();
    }
}
```

从磁盘中读取数据我们通常用输入流，也就是FileInputStream的read方法：

```
public int read(byte b[]) throws IOException {
    return readBytes(b, 0, b.length);
}
private native int readBytes(byte b[], int off, int len) throws IOException;
```

在这种情况下数据先从磁盘到应用进程中，然后通过SocketInputStream写入到目的地，应用进程充当了无效的媒介:

在发起读取和写入的时候还要经过内核态到用户态之间的切换，在文件比较大的时候，这种切换加上数据移动造成效率低下，那么能否将磁盘的数据直接传输到目的地，数据不再经过应用进程，而只是由应用进程在磁盘和目的地之间建立联系，指明要传输的文件和传输的目的地。

写在最后

本篇文章的心路历程是从一般的IO接口到特殊的IO接口，引出操作系统的缓存策略，再到数据库的设计策略，我们抽取共性会发现他们都是缓存，但是应对的场景不同，操作系统的更为通用，Linux会更加激进的使用内存，也就是说Linux会将一部分磁盘上的内容加载到内存中，来提升访问速度。但是内存是有限的，于是Linux引入了active List 和 inactive List来提升缓存的利用率，刚开始页面加载会进入Inactive List，访问两次之后会进入Inactive List，我想这是出于一种假设，经验之谈。 而MySQL内存管理策略引入了频率，如果在指定时间窗口内访问到，会被MySQL认为属于临时起意访问，而不是MySQL理解的热点数据，就像是高中的教科书，某天你会打开课外书籍看了又看，但真正的热点书籍是教科书，他们都会在固定的间隔被你访问到，应当被放到书桌上显眼的位置。

刚开始是在medium看到一篇文章，里面言及零拷贝并不是在所有场景都领先， 于是后面跟别人讨论，后面就有了这篇文章。写作的时候就将其联系起来了，就像一个一个神经元一样，联系起来才是有价值的(这句话来自万维网创始人的纪录片，我很喜欢这句话)。

我写文章看待事物不喜欢孤立的看待事物，在我看来设计原则是共通的，将他们联系起来是有价值的，所以讨论缓存的时候专门再把MySQL的缓存管理策略单独拎出来讨论了一下，这会让我们对缓存有进一步的认识，原本这篇文章是将零拷贝页混入其中的，但是落笔的时候就发现了失控的现象，有些读者批评我行文过长，不够流畅。 这次我也听取意见，裁剪的文章的长度。在上面的描述中好像传统IO一无是处，做了一些无用功，我们将在后面的文章里面看到一些在某些时刻更加高效的文件复制技术，这里我们做了暗示，某些场合更加高效，不是在所有场合都更加高效意味着不稳定，有极好的成绩，也有极差的成绩。以前我自己学习文章，看到的论述都是拿极好的成绩，这一度让我认为传统IO可以从标准IO中移除，当我自己去探索的时候，我才发现并非如此，这也是我不喜欢孤立看待一些技术的原因，有些文章给我的感觉颇有种从《不要断章取义》中摘录了五个字要断章取义一样。只讲最好不讲最差，是不是属于报喜不报忧呢?

这里感觉还是要恢复更新的节奏，本来按照打算是打算再打磨打磨的，后面想一想，还是早点发出来吧，我们下期再见。

参考资料

[1] What they don't tell you about demand paging in school https://offlinemark.com/demand-paging/

[2] malloc vs mmap in C https://stackoverflow.com/questions/1739296/malloc-vs-mmap-in-c

[3] What are void pointers for in C++? https://stackoverflow.com/questions/2860626/what-are-void-pointers-for-in-c
[4] Is there any difference between mmap vs mmap64? https://stackoverflow.com/questions/59453555/is-there-any-difference-between-mmap-vs-mmap64

[5] CreateFileMappingW https://learn.microsoft.com/zh-cn/windows/win32/api/memoryapi/nf-memoryapi-createfilemappingw?redirectedfrom=MSDN

[6] Why is malloc() considered a library call and not a system call? https://stackoverflow.com/questions/71413587/why-is-malloc-considered-a-library-call-and-not-a-system-call

[7] Is there any difference between mmap vs mmap64? https://stackoverflow.com/questions/59453555/is-there-any-difference-between-mmap-vs-mmap64

[8] Why is malloc() considered a library call and not a system call? https://stackoverflow.com/questions/71413587/why-is-malloc-considered-a-library-call-and-not-a-system-call

[9] How to use mmap to allocate a memory in heap? https://stackoverflow.com/questions/4779188/how-to-use-mmap-to-allocate-a-memory-in-heap

[10] mmap 内存映射，是越过了操作系统，直接通过内存访问文件吗？ https://www.zhihu.com/question/522132580/answer/3241695059

[11] 一步一图带你深入理解 Linux 虚拟内存管理 https://mp.weixin.qq.com/s?__biz=Mzg2MzU3Mjc3Ng==&mid=2247486732&idx=1&sn=435d5e834e9751036c96384f6965b328&chksm=ce77cb4bf900425d33d2adfa632a4684cf7a63beece166c1ffedc4fdacb807c9413e8c73f298&scene=178&cur_album_id=2559805446807928833#rd

[12] 大量类加载器创建导致诡异FullGC https://heapdump.cn/article/1924890

[13] JDK-8268893 https://bugs.openjdk.org/browse/JDK-8268893

[14] Temurin™ Supported Platforms https://adoptium.net/zh-CN/supported-platforms/

[15] glibc https://sourceware.org/glibc/wiki/MallocInternals

[16] Why is malloc() considered a library call and not a system call? https://stackoverflow.com/questions/71413587/why-is-malloc-considered-a-library-call-and-not-a-system-call

[17] Chapter 16: The Page Cache and Page Writeback https://github.com/firmianay/Life-long-Learner/blob/master/linux-kernel-development/chapter-16.md

[18] Why does not Redis use linux zero-copy syscall api ? https://github.com/redis/redis/issues/12682

[19] 一步一图带你深入理解 Linux 物理内存管理 https://www.cnblogs.com/binlovetech/p/16914715.html

[20] LWN 495543: 一种更好的平衡 active/inactive 链表长度的算法（Refault Distance 算法） https://tinylab.org/lwn-495543/

[21] Linux active/inactive lists https://jlhu.io/2023-08/linux-active-inactive-lists/

[22] Linux中的内存回收 [一] https://zhuanlan.zhihu.com/p/70964195

[23] Making the Buffer Pool Scan Resistant https://dev.mysql.com/doc/refman/8.0/en/innodb-performance-midpoint_insertion.html

[24] linux内存源码分析 - 内存回收(lru链表) https://www.cnblogs.com/tolimit/p/5447448.html

[25] LWN 495543: 一种更好的平衡 active/inactive 链表长度的算法（Refault Distance 算法） https://tinylab.org/lwn-495543/

[26] Linux中的内存回收 [一] https://zhuanlan.zhihu.com/p/70964195

[27] RocketMQ 存储机制浅析 https://developer.volcengine.com/articles/7359487602480644122

[28] MySQL · 引擎特性 · InnoDB Buffer Pool http://mysql.taobao.org/monthly/2017/05/01/

[29] 什么是 OLTP? https://www.oracle.com/cn/database/what-is-oltp/

[30] 一文聊透 Linux 缺页异常的处理 —— 图解 Page Faults https://www.cnblogs.com/binlovetech/p/17918733.html

[31] 性能优化缓存和内存管理器 https://learn.microsoft.com/zh-cn/windows-server/administration/performance-tuning/subsystem/cache-memory-management/

[32] 文件缓存 https://learn.microsoft.com/zh-cn/windows/win32/fileio/file-caching

[33] WriteFile 函数 https://learn.microsoft.com/zh-cn/windows/win32/api/fileapi/nf-fileapi-writefile