1.提供者

fflush是libc.a中提供的方法，

fsync是系统提供的系统调用。

2.原形

fflush接受一个参数FILE *.

fflush(FILE *);

fsync接受的时一个Int型的文件描述符。

fsync(int fd);

3.功能

fflush:是把C库中的缓冲调用write函数写到磁盘[其实是写到内核的缓冲区]。

fsync：是把内核缓冲刷到磁盘上。

c库缓冲-----fflush---------〉内核缓冲--------fsync-----〉磁盘

linux 同步IO: sync msync、fsync、fdatasync与fflush

最近阅读leveldb源码，作为一个保证可靠性的kv数据库其数据与磁盘的交互可谓是极其关键，其中涉及到了不少内存和磁盘同步的操作和策略。为了加深理解，从网上整理了linux池畔同步IO相关的函数，这里做一个罗列和对比。大部分为copy，仅为记录，请各位看官勿喷。

传统的UNIX实现在内核中设有缓冲区高速缓存或页面高速缓存，大多数磁盘I/O都通过缓冲进行。当将数据写入文件时，内核通常先将该数据复制到其中一个缓冲区中，如果该缓冲区尚未写满，则并不将其排入输出队列，而是等待其写满或者当内核需要重用该缓冲区以便存放其他磁盘块数据时，再将该缓冲排入输出队列，然后待其到达队首时，才进行实际的I/O操作。这种输出方式被称为延迟写（delayed write）（Bach [1986]第3章详细讨论了缓冲区高速缓存）。

延迟写减少了磁盘读写次数，但是却降低了文件内容的更新速度，使得欲写到文件中的数据在一段时间内并没有写到磁盘上。当系统发生故障时，这种延迟可能造成文件更新内容的丢失。为了保证磁盘上实际文件系统与缓冲区高速缓存中内容的一致性，UNIX系统提供了sync、fsync和fdatasync三个函数。

sync函数只是将所有修改过的块缓冲区排入写队列，然后就返回，它并不等待实际写磁盘操作结束。

通常称为update的系统守护进程会周期性地（一般每隔30秒）调用sync函数。这就保证了定期冲洗内核的块缓冲区。命令sync(1)也调用sync函数。

fsync函数只对由文件描述符filedes指定的单一文件起作用，并且等待写磁盘操作结束，然后返回。fsync可用于数据库这样的应用程序，这种应用程序需要确保将修改过的块立即写到磁盘上。

fdatasync函数类似于fsync，但它只影响文件的数据部分。而除数据外，fsync还会同步更新文件的属性。

对于提供事务支持的数据库，在事务提交时，都要确保事务日志（包含该事务所有的修改操作以及一个提交记录）完全写到硬盘上，才认定事务提交成功并返回给应用层。

一个简单的问题：在*nix操作系统上，怎样保证对文件的更新内容成功持久化到硬盘？

1. write不够，需要fsync

一般情况下，对硬盘（或者其他持久存储设备）文件的write操作，更新的只是内存中的页缓存（page cache），而脏页面不会立即更新到硬盘中，而是由操作系统统一调度，如由专门的flusher内核线程在满足一定条件时（如一定时间间隔、内存中的脏页达到一定比例）内将脏页面同步到硬盘上（放入设备的IO请求队列）。

因为write调用不会等到硬盘IO完成之后才返回，因此如果OS在write调用之后、硬盘同步之前崩溃，则数据可能丢失。虽然这样的时间窗口很小，但是对于需要保证事务的持久化（durability）和一致性（consistency）的数据库程序来说，write()所提供的“松散的异步语义”是不够的，通常需要OS提供的同步IO（synchronized-IO）原语来保证：

```
int fsync(int fd);
```

fsync的功能是确保文件fd所有已修改的内容已经正确同步到硬盘上，该调用会阻塞等待直到设备报告IO完成。

PS：如果采用内存映射文件的方式进行文件IO（使用mmap，将文件的page cache直接映射到进程的地址空间，通过写内存的方式修改文件），也有类似的系统调用来确保修改的内容完全同步到硬盘之上：

```
int msync(void *addr, size_t length, int flags)
```

msync需要指定同步的地址区间，如此细粒度的控制似乎比fsync更加高效（因为应用程序通常知道自己的脏页位置），但实际上（Linux）kernel中有着十分高效的数据结构，能够很快地找出文件的脏页，使得fsync只会同步文件的修改内容。

2. fsync的性能问题，与fdatasync

除了同步文件的修改内容（脏页），fsync还会同步文件的描述信息（metadata，包括size、访问时间st_atime & st_mtime等等），因为文件的数据和metadata通常存在硬盘的不同地方，因此fsync至少需要两次IO写操作，fsync的man page这样说：

> "Unfortunately fsync() will always initialize two write operations : one for the newly written data and another one in order to update the modification time stored in the inode. If the modification time is not a part of the

> transaction concept fdatasync() can be used to avoid unnecessary inode disk write operations."

多余的一次IO操作，有多么昂贵呢？根据Wikipedia的数据，当前硬盘驱动的平均寻道时间（Average seek time）大约是3~15ms，7200RPM硬盘的平均旋转延迟（Average rotational latency）大约为4ms，因此一次IO操作的耗时大约为10ms左右。这个数字意味着什么？下文还会提到。

Posix同样定义了fdatasync，放宽了同步的语义以提高性能：

```
int fdatasync(int fd);
```

fdatasync的功能与fsync类似，但是仅仅在必要的情况下才会同步metadata，因此可以减少一次IO写操作。那么，什么是"必要的情况"呢？根据man page中的解释：

> "fdatasync does not flush modified metadata unless that metadata is needed in order to allow a subsequent data retrieval to be corretly handled."

举例来说，文件的尺寸（st_size）如果变化，是需要立即同步的，否则OS一旦崩溃，即使文件的数据部分已同步，由于metadata没有同步，依然读不到修改的内容。而最后访问时间(atime)/修改时间(mtime)是不需要每次都同步的，只要应用程序对这两个时间戳没有苛刻的要求，基本无伤大雅。

PS：open时的参数O_SYNC/O_DSYNC有着和fsync/fdatasync类似的语义：使每次write都会阻塞等待硬盘IO完成。（实际上，Linux对O_SYNC/O_DSYNC做了相同处理，没有满足Posix的要求，而是都实现了fdatasync的语义）相对于fsync/fdatasync，这样的设置不够灵活，应该很少使用。

3. 使用fdatasync优化日志同步

文章开头时已提到，为了满足事务要求，数据库的日志文件是常常需要同步IO的。由于需要同步等待硬盘IO完成，所以事务的提交操作常常十分耗时，成为性能的瓶颈。

> 在Berkeley DB下，如果开启了AUTO_COMMIT（所有独立的写操作自动具有事务语义）并使用默认的同步级别（日志完全同步到硬盘才返回），写一条记录的耗时大约为5~10ms级别，基本和一次IO操作（10ms）的耗时相同。

我们已经知道，在同步上fsync是低效的。但是如果需要使用fdatasync减少对metadata的更新，则需要确保文件的尺寸在write前后没有发生变化。日志文件天生是追加型（append-only）的，总是在不断增大，似乎很难利用好fdatasync。

且看Berkeley DB是怎样处理日志文件的：

1. 每个log文件固定为10MB大小，从1开始编号，名称格式为"log.%010d"
2. 每次log文件创建时，先写文件的最后1个page，将log文件扩展为10MB大小

3.向log文件中追加记录时，由于文件的尺寸不发生变化，使用fdatasync可以大大优化写log的效率

4.如果一个log文件写满了，则新建一个log文件，也只有一次同步metadata的开销

4. fflush

标准IO函数（如fread，fwrite等）会在内存中建立缓冲，该函数刷新内存缓冲，将内容写入内核缓冲，而要想将其真正写入磁盘，还需要调用fsync。（即先调用fflush然后再调用fsync，否则不会起作用）。fflush以指定的文件流描述符为参数（对应以fopen等函数打开的文件流），仅仅是把上层缓冲区中的数据刷新到内核缓冲区就返回，因此相对于fsync而言不是很安全，还需要再调用一下fsync来把数据真正写入硬盘。为了实现以上功能，需要把文件流描述符（fp）转换为文件描述符（fd），以方便fsync的调用，使用以下函数：int fileno(FILE *fp); <- in stdio.