IO 分为网络和存储 IO 两种类型。

网络 IO 对应的是网络数据传输过程，网络是分布式系统的基石，通过网络把离散的物理节点连接起来，形成一个有机的系统。

存储 IO 对应的就是数据存储到物理介质的过程，通常我们物理介质对应的是磁盘，磁盘上一般会分个区，然后在上面格式化个文件系统出来，所以我们普通程序员最常看见的是文件 IO 的形式。

文件的读写最核心的要素是什么？

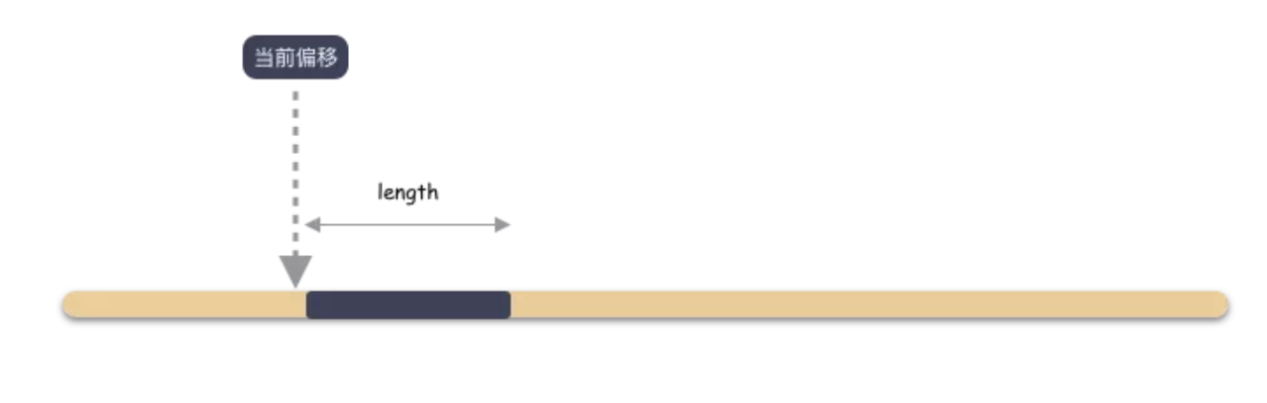
通俗来讲：读文件，就是把磁盘上的文件的**特定位置**的数据读到**内存的 buffer** 。写文件，就是把**内存 buffer** 的数据写到磁盘的文件的**特定位置**。

这里注意到两个关键词：

1. 特定位置；
2. 内存 buffer；

**特定位置怎么理解？怎么指定所谓的**特定位置**？**

很简单，用 [ offset, length ] 这两个参数就能标识一段位置。



也就是 IO 偏移和长度，Offset 和 Length。

**内存 buffer 怎么理解？**

归根结底，文件的数据和谁直接打交道？**内存**，写的时候是从内存写到磁盘文件的，读的时候是从磁盘文件读到内存的。

本质上，下面的 IO 函数都离不开 Offset，Length，buffer 这三个要素。

系统调用IO

直接使用 syscall 库，也就是系统调用。从名字也能看出来，系统调用是和操作系统强相关的，因为是操作系统提供给你的调用接口，所以系统调用会因为操作系统不同而导致不同的特性，不同的接口。

所以，如果你直接使用 syscall 库来使用系统调用，那么需要你自己来承受系统带来的兼容性问题。

**划重点：存储 IO 要对齐。**

资深存储人员为啥总叫你注意 IO 对齐的？机械磁盘 IO 为什么要 512 对齐呢，SSD 盘为啥要 4K 对齐？不对齐又会如何？

**重要的知识点**：

1. 机械盘的 IO 要扇区对齐（绝大部分的扇区是 512 字节大小），磁盘的读写最小单元就是扇区；
2. SSD 盘的 IO 要 4K 对齐，SSD 盘的读写单元是 page，一个 page 为 4K 大小；

**如果不对齐，会有问题问题？**

1. 性能极差；
2. SSD 寿命缩短；

**本质原因是内部 IO 流量和次数的放大。**

这个秘密就在于前面提到的，**机械盘和 SSD 盘的 IO 都有最小单元的概念**。机械盘是以扇区为最小的空间单位，SSD 盘则是 4k 的 page 作为 IO 的最小单元。所以当在机械盘上读 1 个字节的数据，本质上是至少读一个扇区，写 1 个字节则更复杂，先要把这个扇区读出来，然后在内存里 update，最后又把这个扇区写回到磁盘。

**你可能又有疑问了**：为啥机械盘和 SSD 盘都要按照一个单元来管理呢，为啥不 1 字节 1 字节管理呢？如果这样的话，业务读写不就方便了嘛。

**答案是:开销。** 考虑以下两个方面：

* 第一个是存储的开销：数据存储到磁盘上还有 SSD 盘上是会有校验的（比如 ECC 校验），如果每个字节都对应一个校验，存 1T 的数据有 1T 的校验？那这个存储开销无比的大。那如果是现在 512 字节对应一个 1 字节的校验，则开销完全可控；
* 第二个是性能的开销：物理硬件是跟真实世界的设计匹配的，对于机械磁盘的写是利用磁化介质来存储数据，如果磁盘上是1 个字节，甚至 1 个 bit 来独立存储，那磁盘磁头的次数将无比庞大，性能也将无比的下降（每一次都是写数据+校验）。SSD 的写则更复杂，**因为 SSD 由于自身的存储特点，是无法覆盖写的，每次写都是写新位置，旧的位置则是作为垃圾等待后台 GC ，粒度太小则会导致擦写的次数无比的多，性能和寿命都将不可接受；**

所以说，在综合因素的权衡下，硬件对于空间管理会划分固定单元，并对 IO 也提出单元对齐的要求。至于磁盘固定单元为 512 字节，SSD 的 IO 单元为 4K 则是经过综合的科学测试和验证的一个数值而已。

**对齐其实很简单，只有两个维度：**

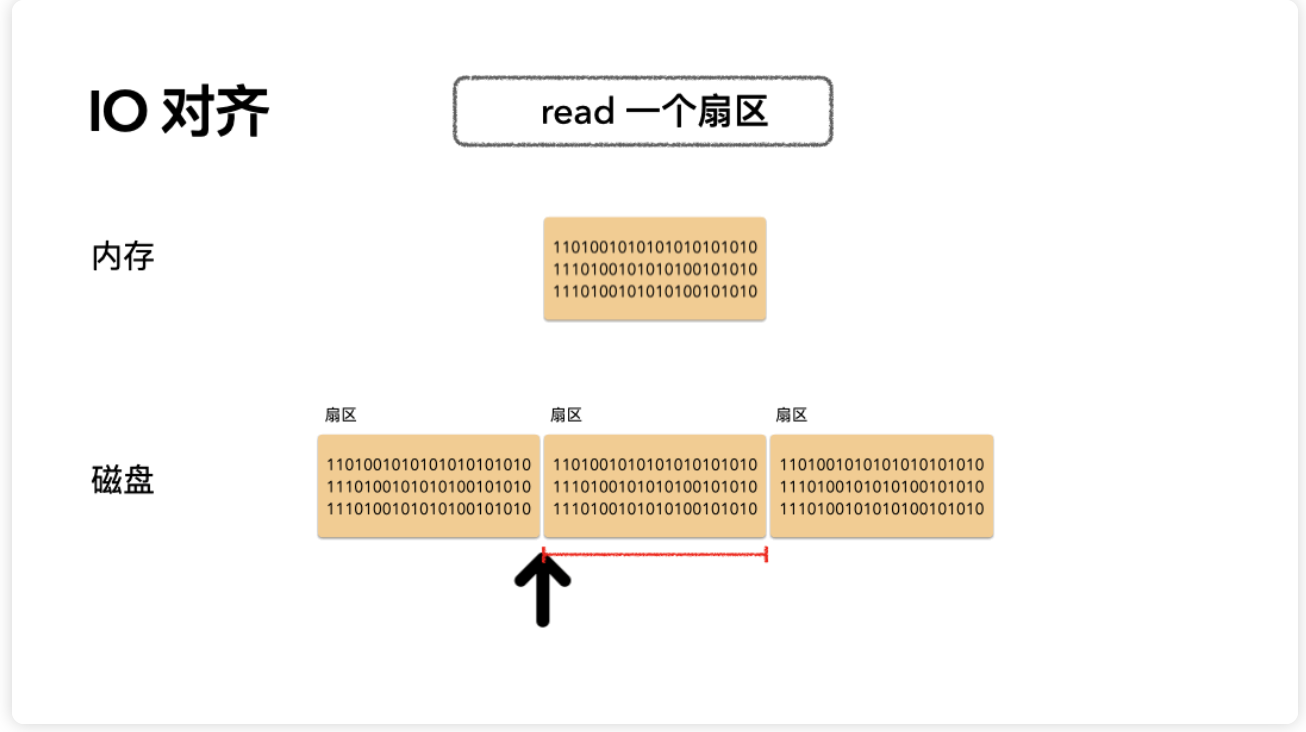
* **偏移对齐**
* **长度对齐**

是不是很简单呀。偏移和长度就是我们常说的 offset，length，对齐通常就是既要保证偏移对齐也要保证长度对齐。

**场景：读 512 个字节数据是怎么做的？**

步骤如下：

1. 磁头摆到指定偏移；
2. 读取一个扇区的数据到内存；

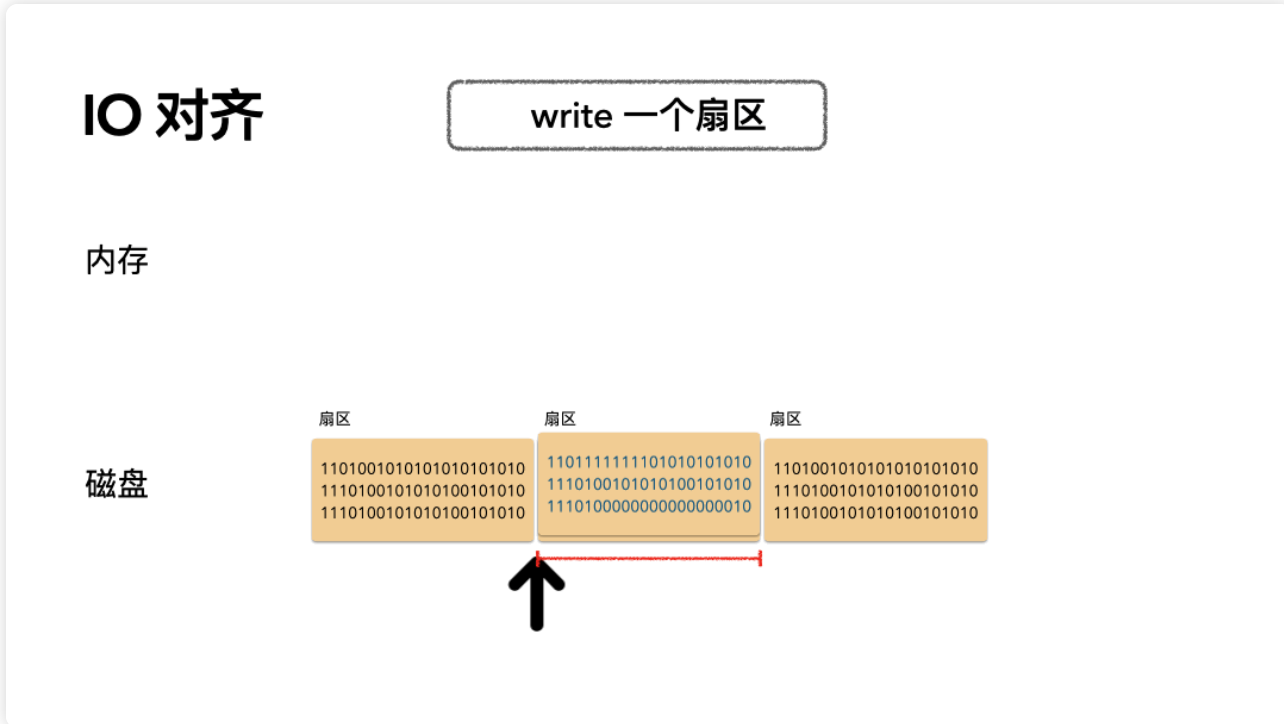


**开销：只需要一次磁盘读 IO。**

**场景：写 512 个字节数据是怎么做的？**

步骤如下：

1. 磁头摆到指定偏移；
2. 覆盖写一个扇区的数据到磁盘；

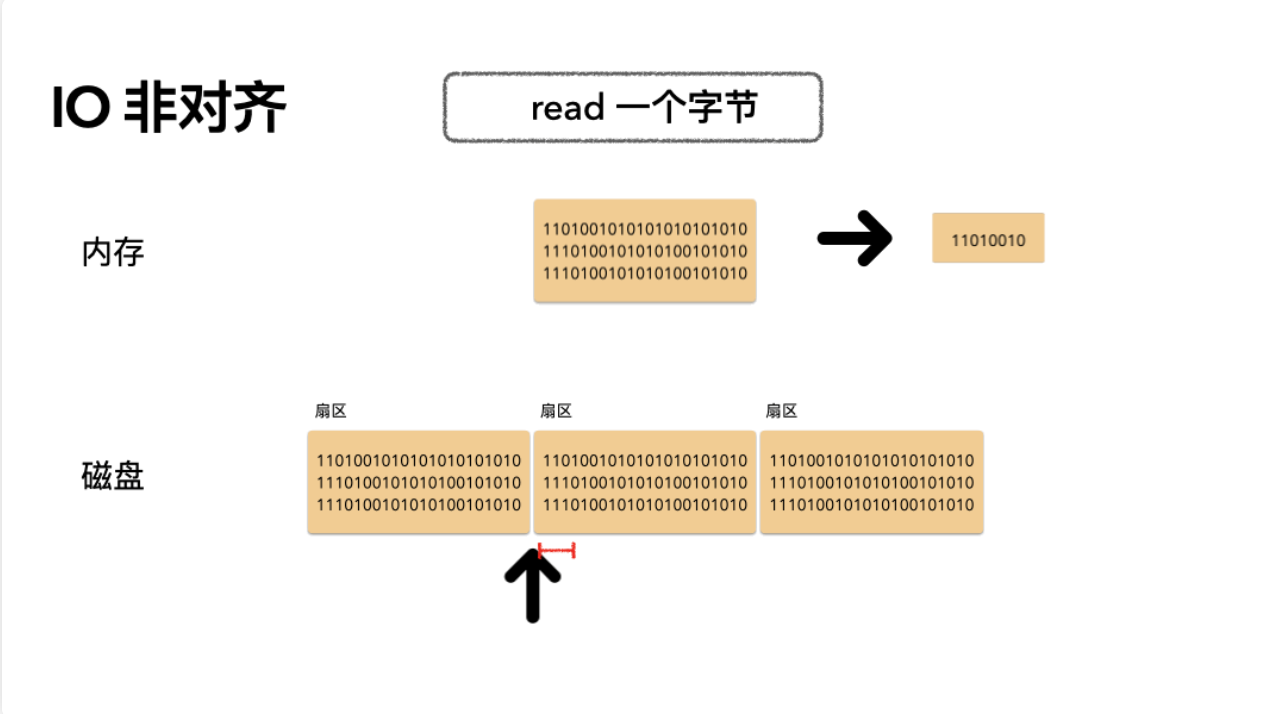


**开销：只需要一次磁盘写 IO。**

**场景：读 1 个字节数据是怎么做的？**

步骤如下：

1. 磁头摆到这 1 个字节数据所在扇区位置，对齐到扇区开始的偏移；
2. 读取 1 个完整扇区（ 512 字节）的数据到内存；
3. 从这 512 字节的内存中，copy 出用户要用的 1 个字节，给到用户；

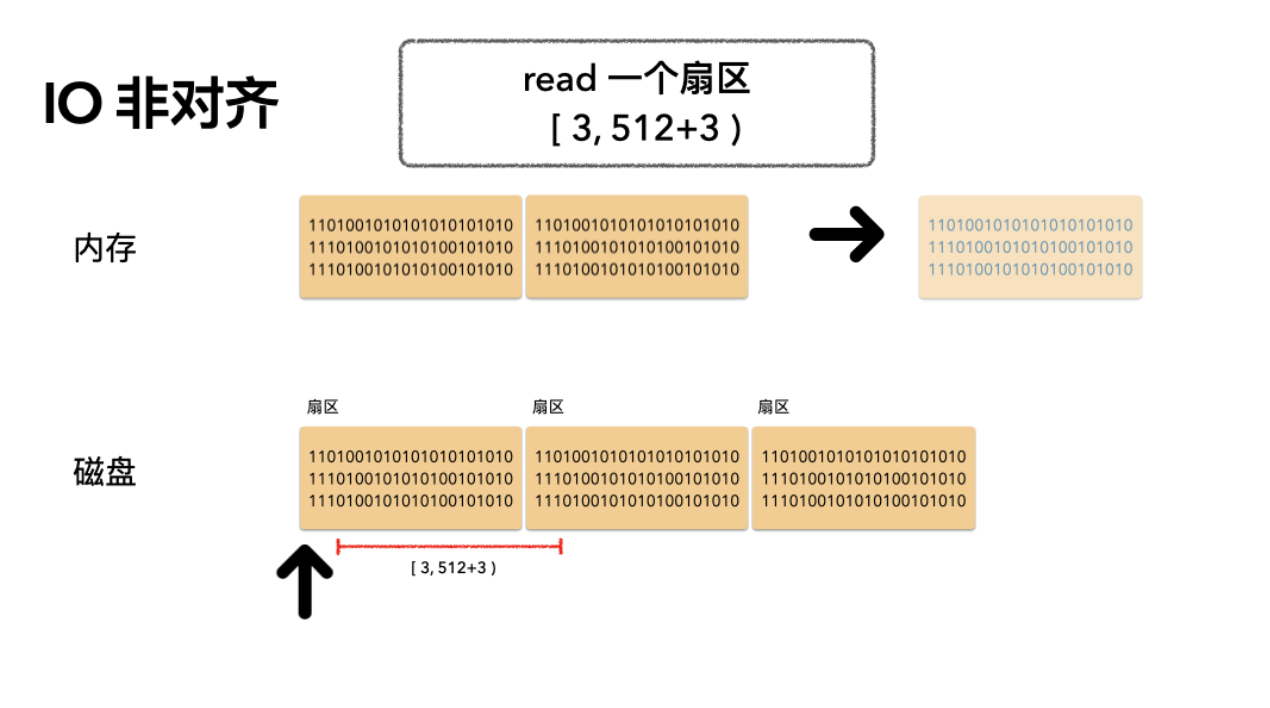


**开销：读放大。如果是读 512 字节的数据，但是偏移不对齐，那么可能导致 2 次 IO。**

**场景：读 512 个字节，但是偏移不对齐？**

步骤如下：

1. 磁头摆到这 512 个字节数据所在 2 个扇区位置，对齐到扇区开始的偏移；
2. 读取 2 个完整扇区（1024 Bytes）的数据到内存；
3. 从这 1024 字节的内存中，copy 出用户要用的 512 个字节，给到用户；

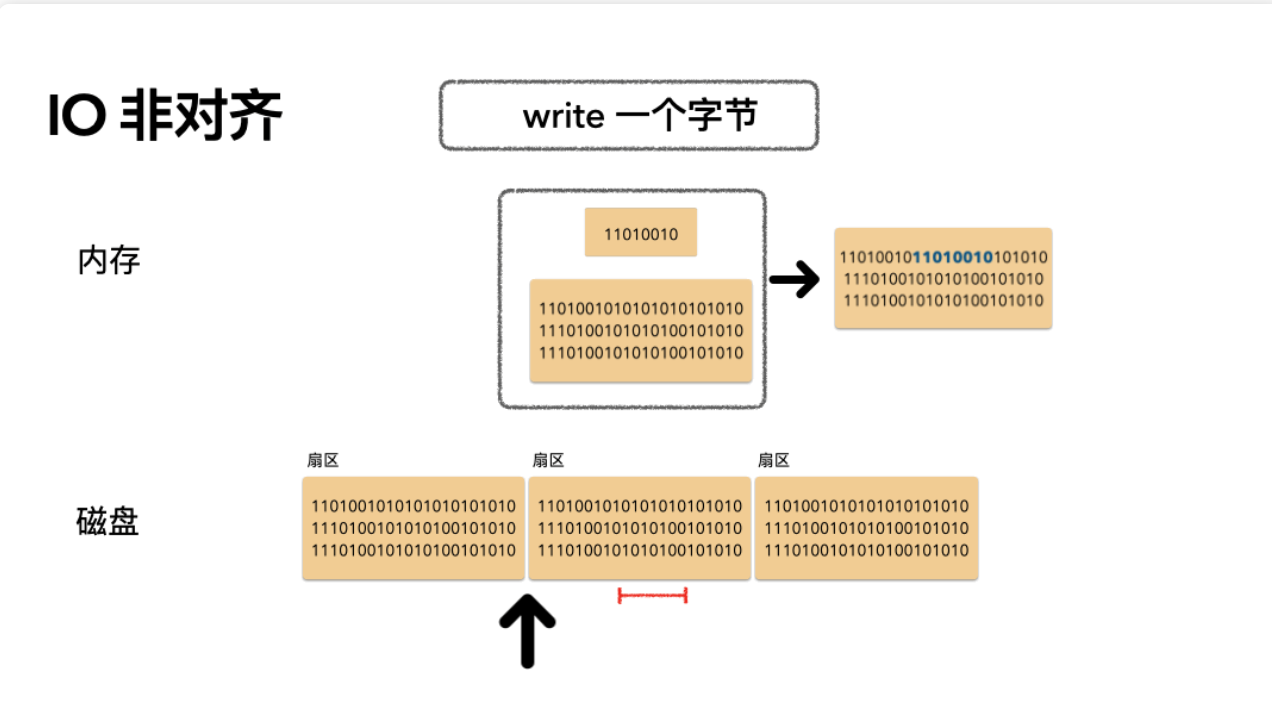


开销：读放大。虽然读的是 1 个扇区的数据，但是偏移却没对齐，所以必须要读 2 个扇区，也放大了一倍的流量。

**场景：写 1 个字节数据是怎么做的？**

写是比较复杂的，用户现在手握 1 个字节的数据，想要写到磁盘，但是磁盘的 IO 单元是扇区，所以本质上是读改写的方式。步骤如下：

1. 先要把这要改动的 1 字节所在的扇区完整的读到内存；
   1. 磁头摆动到扇区偏移；
   2. 读一个扇区，读到内存；
2. 然后在内存中，把这 1 个字节对应的位置的数据修改；
3. 然后重新写回扇区；
   1. 磁盘寻道道扇区偏移；
   2. 覆盖写着一个扇区；



**开销：一次磁盘读 IO，内存合并，最后再一次磁盘写 IO。明明是写，却必须要先读，IO 即放大了流量，又放大了次数**

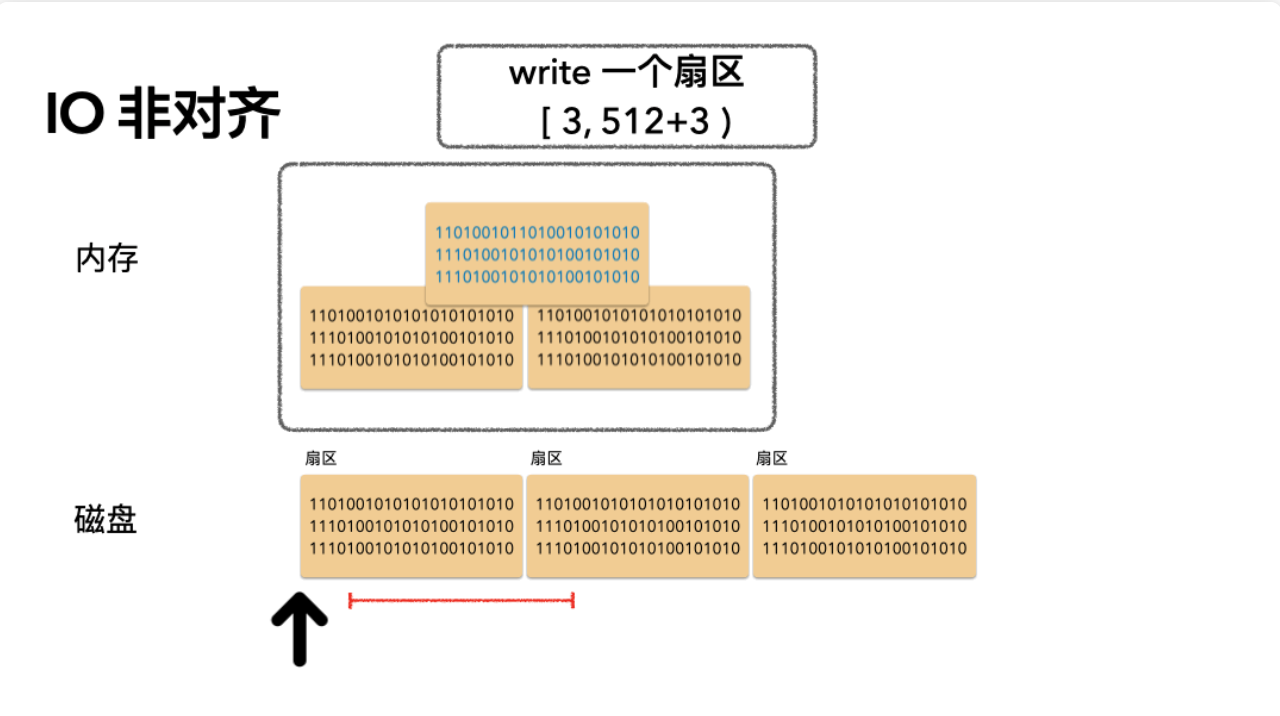
所以你看出来了吗，非对齐的 IO 多了很多步骤。存在 IO 流量和次数的放大。这样会极度拖累性能。

**本来只需要一次 IO 就能完成的操作，放大了一次，那么性能就至少下降一倍。就这么简单。**

**场景：写 1 个扇区的数据，但是偏移不对齐会怎么样？**

**操作步骤**：

1. 读 2 个扇区的数据（1024 Bytes）上来；
2. 合并内存中的数据；
3. 把这 2 个新的扇区数据，写到磁盘；



**开销：所以，我们看到由于不对齐，读的时候导致多读了一个扇区，写的时候导致多写了 1 个扇区。性能自然是极差的。**

机械盘来说，随机 iops 就是几百的样子，带宽也就几百兆。一次寻道都是 10ms 级别的。所以磁盘读写数据的代价远比你想象中的大。

这个有多慢？cpu 执行指令都是纳秒和微秒级别的，一次 IO 性能和 cpu 或者内存的操作相差十万百千里。所以，但凡你能省 1 次 IO 都是非常大的性能提升。

而对于 SSD 来说，如果你 IO 不对齐，很有可能峰值能跑 5 万 iops 的盘只能跑 2 万甚至更低。**大家一定要有个意识：IO 能节省就节省，多一次 IO 性能可能慢 1 倍。**

这个时候你可能反问了，我读写文件从没考虑过对齐的问题呀？

是的，你是没考虑过，那是因为有一个苦逼帮你把这个活干了，谁呢？

**文件系统。这个世界并不简单，只是有人替你负重前行。**

绝大部分的程序员都是基于文件系统之上操作磁盘。文件系统则会使用 buffer cache 自动帮用户对齐 IO ，然后再下发磁盘。一旦你想要深入优化 IO 的性能，IO 对齐一定是第一道要考量的坎。

还有一点要特别注意，一个特别重要的环节：**文件系统格式化化的时候。**

**格式化的时候，一定要注意对齐。** 格式化文件系统的时候，一定要注意对齐的偏移，不要故意搞成非对齐的偏移，不然一旦文件系统格式化的时候都不对齐，后面谁也救不了，除非你重新格式化文件系统。

举个例子，本来文件系统从偏移磁盘 0 这个位置格式化，你偏不，你偏要从 3 字节偏移的位置开始格式化。那后面的所有的貌似对齐的 IO 偏移都将是不对齐的，性能也自然是下降的。

1. IO 对齐是做存储必须要考量的一个因素；
2. IO 对齐的两个核心是：偏移和长度；
3. 非 IO 对齐的请求会导致内部 IO 流量和次数的放到，从而性能下降（SSD 则会因为放到而导致擦写次数过多，更会影响寿命）；
4. 程序员一般很少要主动对齐，因为文件系统帮你 hold 住了一层。但是如果程序员自己不注意，还是会踩坑。比如明明是 SSD 盘，你偏要每次发送 512 字节的 IO 请求，那性能肯定惨不忍睹；
5. 对齐的 IO 简简单单，不对齐的 IO 千奇百怪；

**思考一个问题：写数据做到什么程度才叫安全了？**

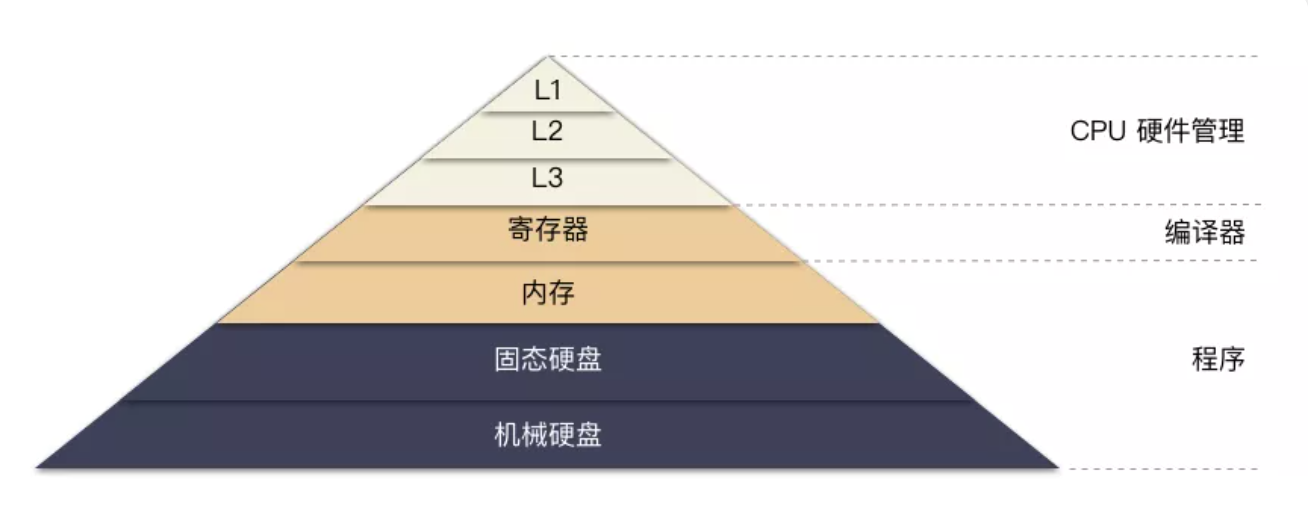
就是：用户发过来一个写 IO 请求，只要你给他回复了 “写成功了”，那么无论机器发生掉电，还是重启等等之类的，数据都还能读出来。

所以，在我们不考虑数据静默错误的前提下，数据安全的最本质要求是什么？

**划重点：那就是数据一定要在非易失性的存储介质里，你才能给用户回复“写成功”。**请一定要记住这句话，做存储开发的人员，80% 的时间都在思考这句话。

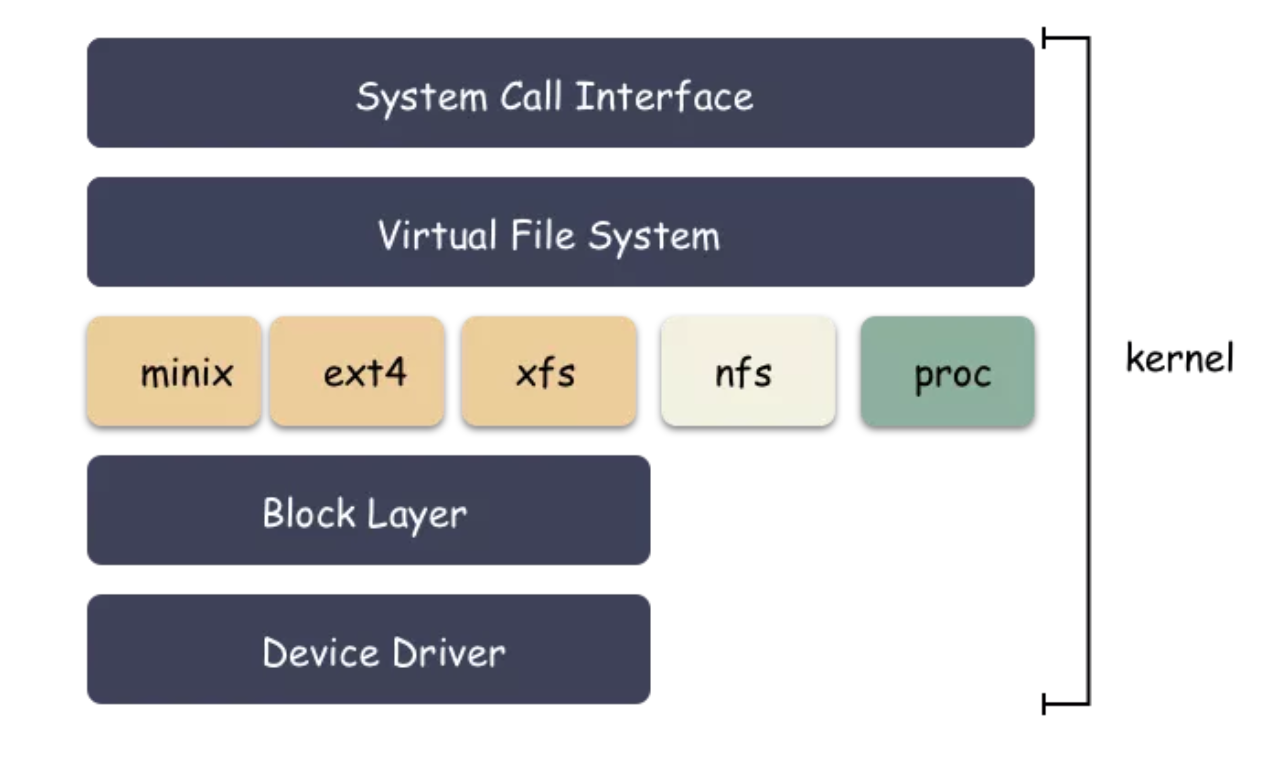
那么常见的易失性介质和非易失性介质有哪些呢？

**易失性介质：寄存器，内存 等；非易失性介质：磁盘，固态硬盘 等；**



**从上到下，速度递减，容量递增，价格递减。**

我们前面提到一个文件的读写方式，标准库的方式和系统调用的方式。无论是哪一种，本质上都是基于文件的一种形式，下面承接了一层文件系统，主要层次：系统调用 -> vfs -> 文件系统 -> 块设备 -> 硬件驱动

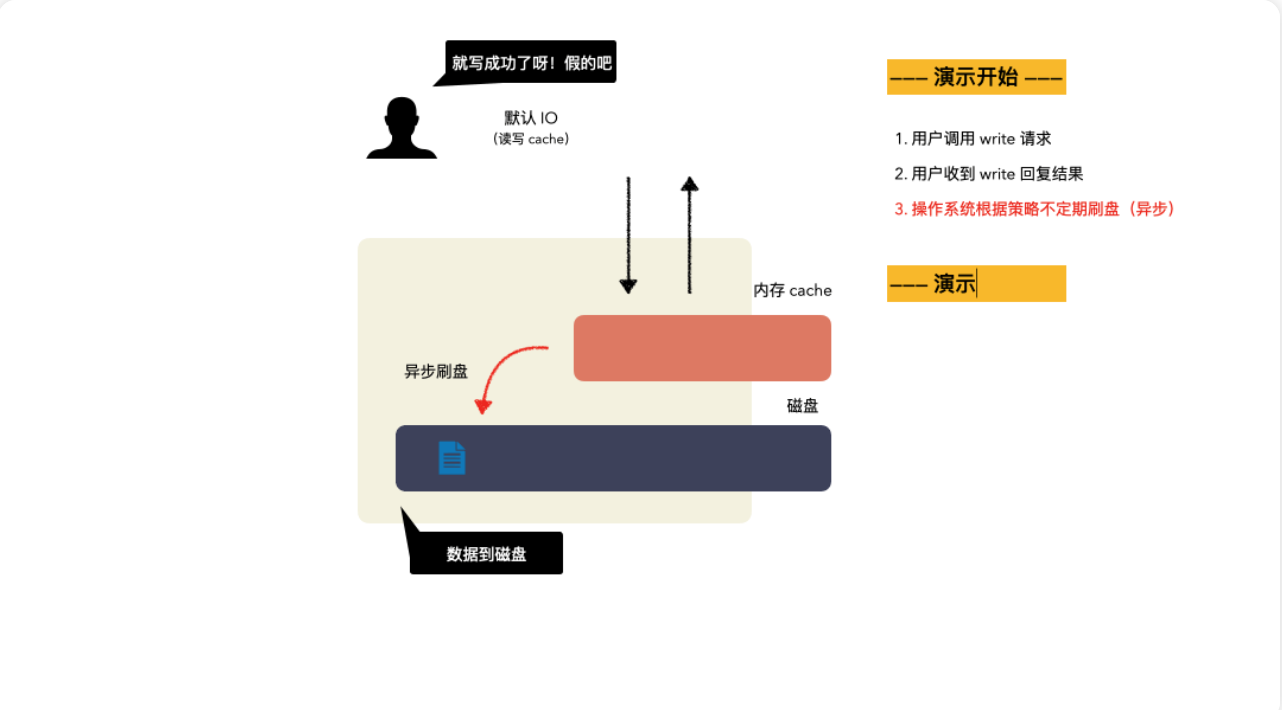


我们 open 了这个文件，然后 write 数据进去。好，现在思考一个问题，当 write 返回成功之后，数据到磁盘了吗？

**答案是：不确定。**

因为有文件系统的 cache ，默认是 write back 的模式，**数据写到内存就返回成功**了，然后内核根据实际情况（比如定期或者脏数据达到某个阈值），异步刷盘。

这样的好处是保证了写的性能，貌似写的性能非常好（可不好嘛，数据写内存的速度），坏处是存在数据风险。因为用户收到成功的时候，数据可能还在内存，这个时候整机掉电，由于内存是易失性介质，数据就丢了。丢数据 是存储最不能接受的事情，相当于丢失了存储的生命线。



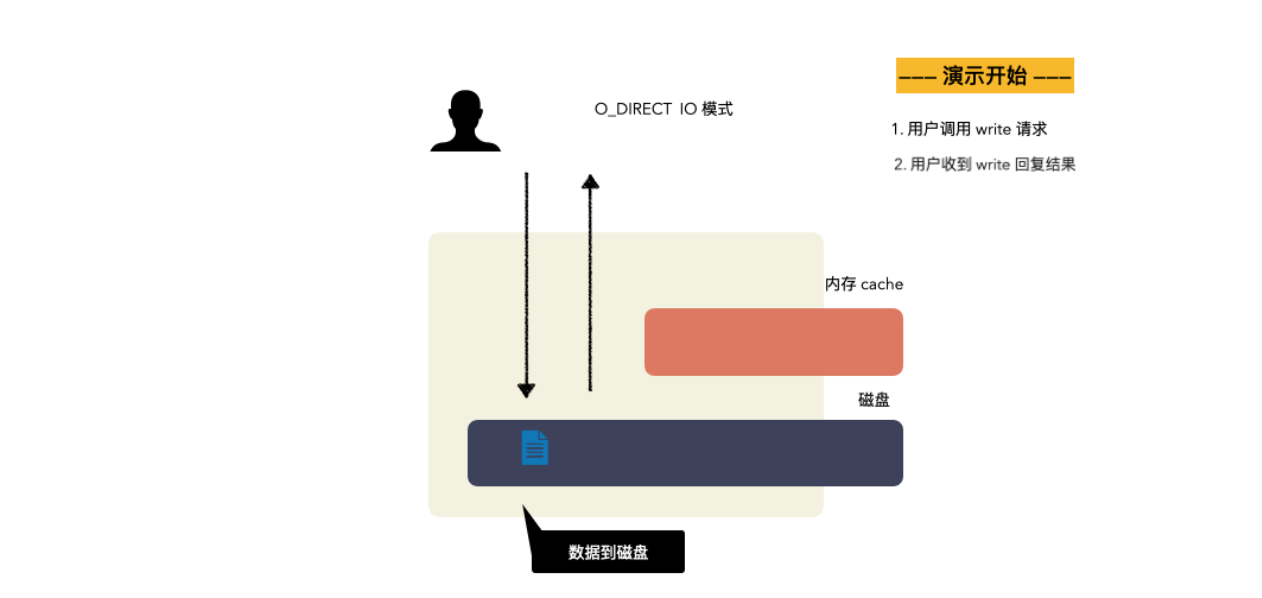
**划重点：还是那句话，一定要确保数据落盘之后，才向用户返回成功。**

**那么怎么才能保证这一点？有以下 3 种方法。**

1. open 文件的时候，用 O\_DIRECT 模式打开，这样 write/read 的时候，文件系统的 IO 会绕过 cache，直接跟磁盘 IO；
2. open 文件的时候，使用 O\_SYNC 模式，确保每一笔 IO 都是同步落盘的。或者 write 之后，主动调用一把 fsync ，强制数据落盘；
3. 读写文件的另一种方式是通过 mmap 函数把文件映射到进程的地址空间，读写进程内存的地址的数据其实是转发到磁盘上去读写，write 之后主动调用一把 msync 强制刷盘；

**O\_DIRECT 模式**

DIRECT IO 模式能够保证每次 IO 都直接访问磁盘数据，而不是数据写到内存就向用户返回成功的结果，这样才能确保数据安全。**因为内存是易失性**的，掉电就丢了，数据只有写到持久化的介质才能安心。



读的时候也是直接读磁盘，而不会缓存到内存中，从而也能节省整机内存的使用。

缺点也同样明显，由于每次 IO 都要落盘，那么性能肯定看起来差(但你要明白，其实这才是真实的磁盘性能)。

**划重点：使用了**O\_DIRECT**模式之后，必须要用户自己保证对齐规则，否则 IO 会报错**，有 3 个需要对齐的规则：

1. 磁盘 IO 的大小必须扇区大小（512字节）对齐
2. 磁盘 IO 偏移按照扇区大小对齐；
3. 内存 buffer 的地址也必须是扇区对齐；

#include <stdio.h>  
#include <stdlib.h>  
#include <assert.h>  
#include <fcntl.h>  
#include <errno.h>  
#include <string.h>  
#include <stdint.h>  
  
extern int errno;  
#define align\_ptr(p, a) \  
    (u\_char \*)(((uintptr\_t)(p) + ((uintptr\_t)a - 1)) & ~((uintptr\_t)a - 1))  
int main(int argc, char \*\*argv)  
{  
    char timestamp[8192] = {0,};  
    char \*timestamp\_buf = NULL;  
    int timestamp\_len = 0;  
    ssize\_t n = 0;  
    int fd = -1;  
  
    fd = open("./test\_directio.txt", O\_CREAT | O\_RDWR | O\_DIRECT, 0644);  
    assert(fd >= 0);  
  
    // 对齐内存地址  
    timestamp\_buf = (char \*)(align\_ptr(timestamp, 512));  
    timestamp\_len = 512;  
  
    n = pwrite(fd, timestamp\_buf, timestamp\_len, 0);  
    printf("ret (%ld) errno (%s)\n", n, strerror(errno));  
  
    return 0;  
}

思考问题：有些童鞋可能会好奇问了？IO 大小和偏移按照 512 对齐我会，但是怎么才能保证 malloc 的地址是 512 对齐的呢？

是啊，我们无法用 malloc 来控制生成的地址。这对这个需求，我们有两个解决办法：

**方法一：分配大一点的内存，然后在这个大块内存里找到对齐的地址，只需要确保 IO 大小不会超过最后的边界即可；**

我上面的 demo 例子就是如此，分配了 8192 的内存块，然后从里面找到 512 对齐的地址。从这个地址开始往后 512 个字节是绝对到不了这个大内存块的边界的。对齐的目的安全达成。



**这种方式实现简单且通用，但是比较浪费内存。**

**方法二：使用 posix 标准封装的接口**posix\_memalign**来分配内存，这个接口分配的内存能保证对齐；**

如下，分配 1 KiB 的内存 buffer，内存地址按照 512 字节对齐。

ret = posix\_memalign (&buf, 512, 1024);  
if (ret) {  
    return -1;  
}

**思考一个问题：**O\_DIRECT**模式 的 IO 一般是哪些应用场景？**

* 最常见的是数据库系统，数据库有自己的缓存体系和 IO 优化，无需内核消耗内存再去完成相同的事情，并且可能好心办坏事；
* 不格式化文件系统，而是直接管理块设备的场景；

**标准 IO + sync**

sync 功能：强制刷新内核缓冲区到输出磁盘。

在 Linux 的缓存 I/O 机制中，用户和磁盘之间有一层易失性的介质——内核空间的 buffer cache；

* 读的时候会 cache 一份到内存中以便提高后续的读性能；
* 写的时候用户数据写到内存 cache 就向用户返回成功，然后异步刷盘，从而提高用户的写性能。

**读操作描述如下**：

1. 操作系统先看内核的 buffer cache 有缓存不？有，那么就直接从缓存中返回；
2. 否则从磁盘中读取，然后缓存在操作系统的缓存中；

**写操作描述如下**：

1. 将数据从用户空间复制到内核的内存 cache 中，这时就向用户返回成功，对用户来说写操作就已经完成；
2. 至于内存的数据什么时候才真正写到磁盘由操作系统策略决定（如果此时机器掉电，那么就会丢失用户数据）；
3. 所以，如果你要保证落盘，必须显式调用了 sync 命令，显式把数据刷到磁盘（只有刷到磁盘，机器掉电才不会导致丢数据）；

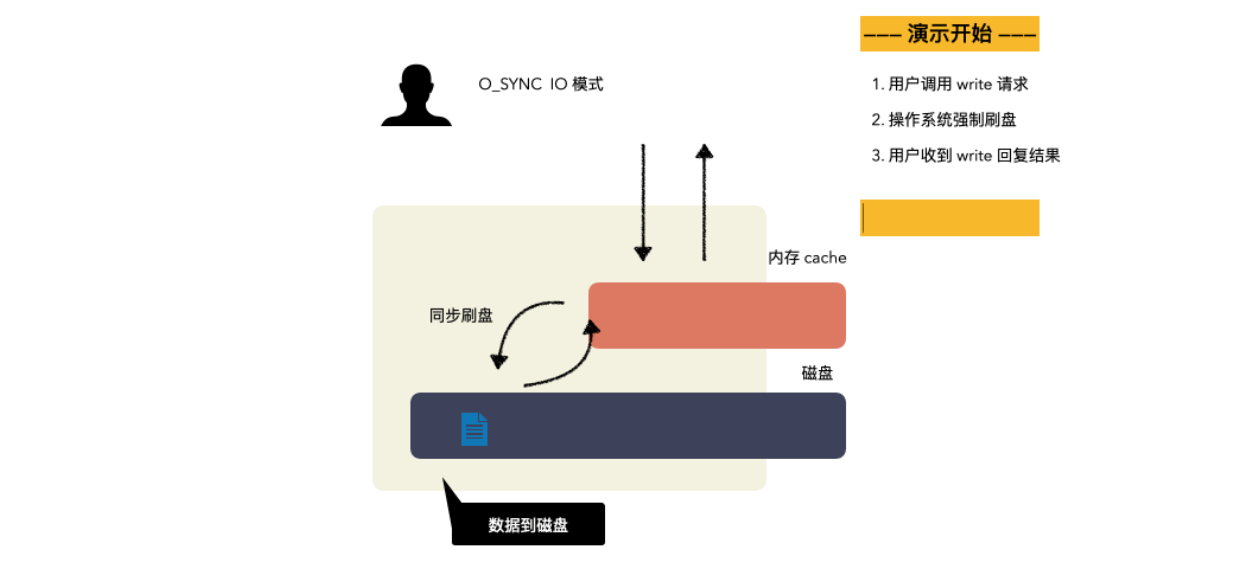
**划重点：**sync**机制能保证当前时间点之前的数据全部刷到磁盘。**。而关于 sync 的方式大概有两种：

1. open 的使用使用 O\_SYNC 标识；
2. 显式调用 fsync 之类的系统调用；

**方法一：**open**使用**O\_SYNC**标识**；

#include <stdio.h>  
#include <stdlib.h>  
#include <assert.h>  
#include <fcntl.h>  
#include <errno.h>  
#include <string.h>  
#include <stdint.h>  
  
extern int errno;  
  
int main(int argc, char \*\*argv)  
{  
    char buffer[512] = {0,};  
    ssize\_t n = 0;  
    int fd = -1;  
  
    fd = open("./test\_sync.txt", O\_CREAT | O\_RDWR | O\_SYNC, 0644);  
    assert(fd >= 0);  
  
    n = pwrite(fd, buffer, 512, 0);  
    printf("ret (%ld) errno (%s)\n", n, strerror(errno));  
  
    return 0;  
}

这种方式能保证每一笔 IO 都是同步 IO，一定是刷到磁盘才返回，但是这种使用姿势一般少见，因为这个性能会很差，并且不利于批量优化。



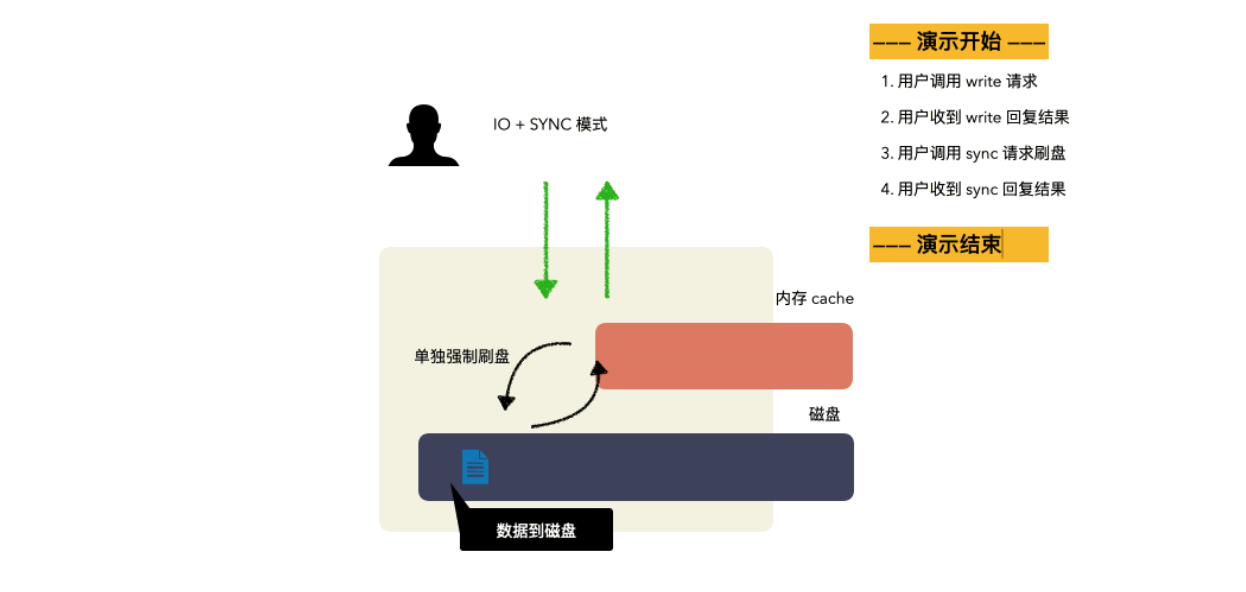
**方法二：单独调用函数**fsync

这个则是在 write 之后 fsync 一把数据到磁盘，这种方式实际生产环境用的多些，因为方便业务优化。**这种方式对程序员提出了更高的要求，要求必须自己掌握好 fsync 的时机，达到既保证安全又保证性能的目的，这里通常是个权衡点。**

比如，你可以 write 10 次之后，最后才调用一把 fsync，这样既能保证刷盘，又达成了批量 IO 的优化目的。

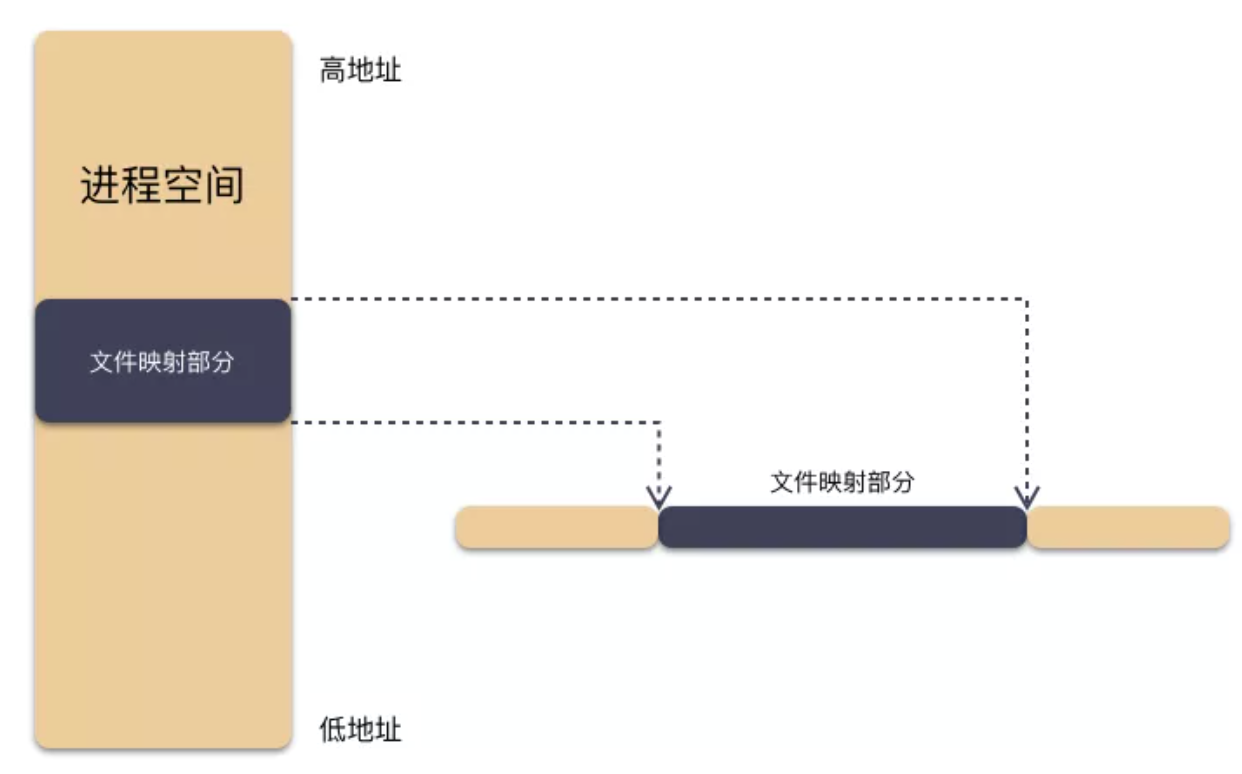
关于这种使用姿势，有几个类似函数，其中有些差异，各自体会下：

// 文件数据和元数据部分都刷盘  
int fsync(int fildes);  
// 文件数据部分都刷盘  
int fdatasync(int fildes);  
// 整个内存 cache 都刷磁盘  
void sync(void);



**mmap + msync**

这是一个非常有趣的 IO 模式，通过 mmap 函数将硬盘上文件与进程地址空间大小相同的区域映射起来，之后当要访问这段内存中一段数据时，内核会转换为访问该文件的对应位置的数据。**从使用姿势上，就跟操作内存一样，但从结果上来看，本质上是文件 IO**。



void \*  
mmap(void \*addr, size\_t len, int prot, int flags, int fd, off\_t offset)  
  
int  
munmap(void \*addr, size\_t len);

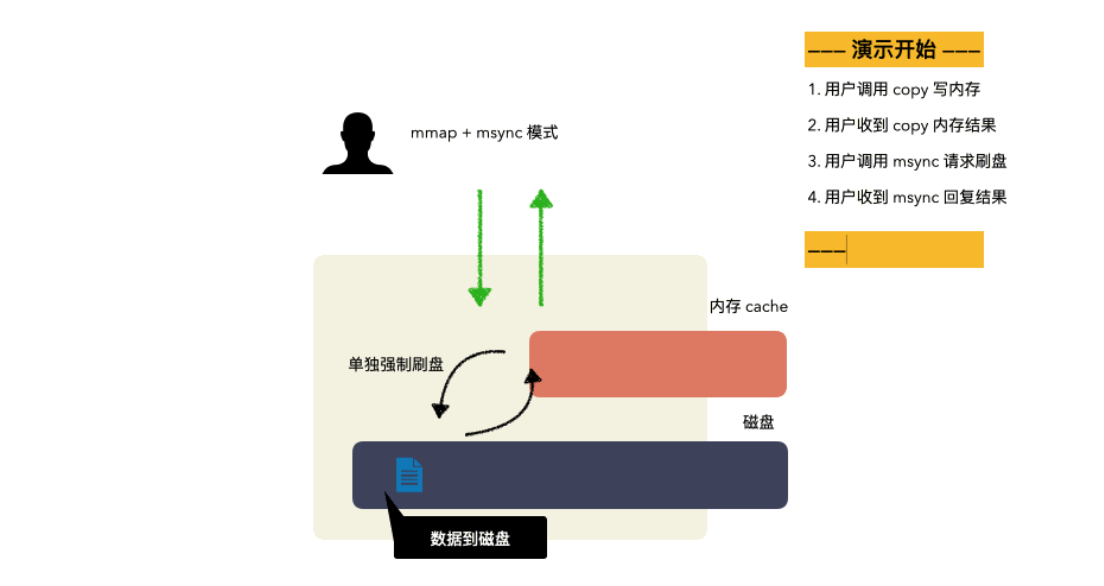
mmap 这种方式可以减少数据在用户空间和内核空间之间的拷贝操作，当数据大的时候，采用内存映射方式去访问文件会获得比较好的效率（因为可以减少内存拷贝量，并且聚合 IO，数据批量下盘，有效的减少 IO 次数）。

当然，你 write 数据也还是异步落盘的，并没有实时落盘，如果要保证落盘，那么必须要调用 msync ，调用成功，才算持久化落盘。

mmap 的优点：

* 减少系统调用的次数。只需要 mmap 一次的系统调用，后续的操作都是内存拷贝操作姿势，而不是 write/read 的系统调用；
* 减少数据拷贝次数；

#include <stdio.h>  
#include <stdlib.h>  
#include <sys/mman.h>  
#include <sys/types.h>  
#include <unistd.h>  
#include <sys/stat.h>  
#include <assert.h>  
#include <fcntl.h>  
#include <string.h>  
  
int main()  
{  
    int ret = -1;  
    int fd = -1;  
  
    fd = open("test\_mmap.txt", O\_CREAT | O\_RDWR, 0644);  
    assert(fd >= 0);  
  
    ret = ftruncate(fd, 512);  
    assert(ret >= 0);  
  
    char \*const address = (char \*)mmap(NULL, 512, PROT\_READ | PROT\_WRITE, MAP\_SHARED, fd, 0);  
    assert(address != MAP\_FAILED);  
  
    // 神奇在这里（看起来是内存拷贝，其实是文件 IO）  
    strcpy(address, "hallo, world");  
    ret = close(fd);  
    assert(ret >= 0);  
  
    // 落盘确保  
    ret = msync(address, 512, MS\_SYNC);  
    assert(ret >= 0);  
  
    ret = munmap(address, 512);  
    assert(ret >= 0);  
  
    return 0;  
}



以上方式保证了文件系统那一层的落盘，但是磁盘硬件其实本身也有缓存，这个属于硬件缓存，这层缓存也是易失的。所以最后一点是，为了保证数据的落盘，硬盘缓存也要关掉。

# 查看写缓存状态；  
hdparm -W  /dev/sda   
# 关闭 HDD Cache,保证数据强一致性；避免断电时数据未落盘；  
hdparm -W  0 /dev/sda  
# 打开 HDD Cache（断电时可能导致丢数据）  
hdparm -W  1 /dev/sda

按照内核保证数据落盘，硬件保证关闭缓存，综合以上的 IO 姿势，当你写一笔 IO 落盘之后，才能说数据写到磁盘了，才能保证数据是掉电非易失的。

1. 数据一定要写在非易失性的存储介质里，你才能给用户回复“写成功”。**其他的取巧的方式都是耍流氓、走钢丝；**
2. 本文总结 3 种最根本的 IO 安全的方式，分别是 O\_DIRECT 写，标准 IO + Sync 方式，mmap 写 + msync 方式。**要么每次都是同步写盘，要么就是每次写完，再调用 sync 主动刷，才能保证数据安全**；
3. O\_DIRECT 对使用者提出了苛刻的要求，必须要满足 IO 的 offset，length 扇区对齐，内存 buffer 地址也要扇区对齐；
4. 注意硬盘也有缓存，这个也是易失性的，必须要考虑在内，可以通过 hdparm 命令开关；