SSD 就是大U盘？聊聊闪存类存储的转换层

目录

上篇 「柱面-磁头-扇区寻址的一些旧事」 整理了一下我对磁盘类存储设备（包括软盘、硬盘，不包括光盘、磁带）的一些理解， 算是为以后讨论文件系统作铺垫；这篇整理一下我对闪存类存储设备的理解。

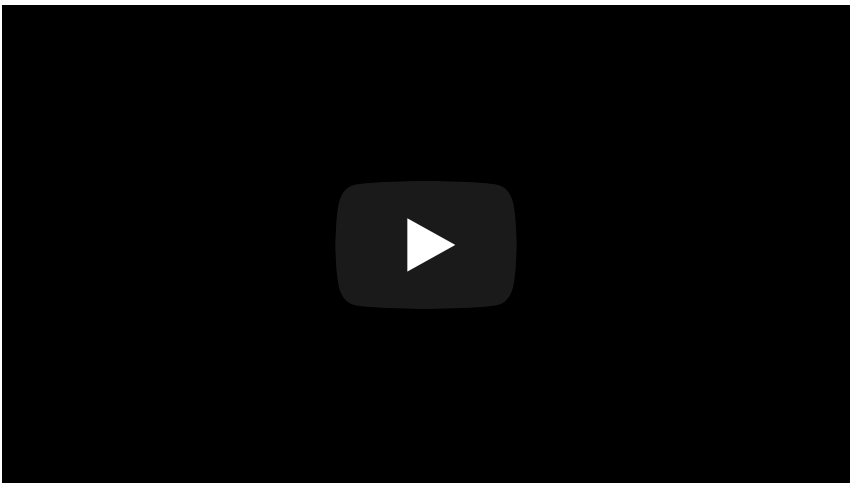
这里想要讨论的闪存类存储是指 SSD 、SD卡、U盘、手机内置闪存等基于 NAND 又有闪存转换层的存储设备（下文简称闪存盘），但不包括裸 NAND 设备、3D Xpoint （Intel Optane）等相近物理结构但是没有类似的闪存转换层的存储设备。 闪存类存储设备这几年发展迅猛，SD卡和U盘早就替代软盘成为数据交换的主流，SSD 大有替代硬盘的趋势。 因为发展迅速，所以其底层技术变革很快，不同于磁盘类存储技术有很多公开资料可以获取， 闪存类存储的技术细节通常是厂商们的秘密，互联网上能找到很多外围资料， 但是关于其如何运作的细节却很少提到。所以我想先整理一篇笔记，记下我搜集到的资料，加上我自己的理解。 本文大部分信息

来源是 Optimizing Linux with cheap flash drives 和 A Summary on SSD & FTL，加上我的理解，文中一些配图也来自这两篇文章。

1 NAND Flash 原理

比 NAND Flash 更早的 EEPROM 等存储技术 曾经用过 NOR Flash cell，用于存储主板配置信息等少量数据已经存在 PC 中很久了。后来 NAND Flash 的微型化使得 NAND Flash 可以用于存储大量数据，急剧降低了存储成本，所以以 NAND Flash 为基础的存储技术能得以替代硬盘等存储设备。

Tutorial: Why NAND Flash Breaks Down



这里不想涉及太多 NAND Flash 硬件细节，有个演讲 Tutorial: Why NAND Flash Breaks Down 和 YouTube 视频 介绍了其原理，感兴趣的可以参考一下。只罗列一下视频中提到的一些 NAND Flash 的特点：

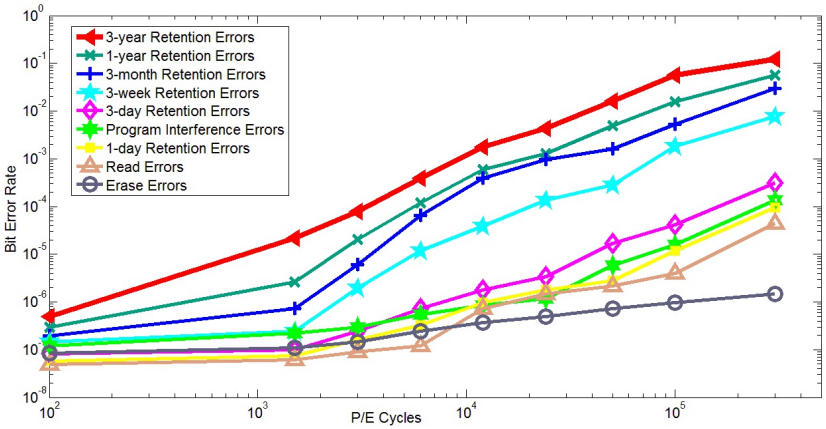
- NAND Flash 使用 floating gate 中束缚电子来保存二进制数据，对这些 Cell 有读取（Read）、 写入（Programming）、擦除（Erase）的操作。擦写次数叫 P/E cycle。
- 电子的量导致的电势差可以区别 1 和 0 ，这是 Single Level Cell (SLC) 的存储方式。 或者可以用不同的电势差区分更多状态保存更多二进制位，从而有 Multi-Level Cell (MLC)， TLC， QLC 等技术。可以对 MLC 的 Flash Cell 使用类似 SLC 的写入模式，物理区别只是参考电压， 只是 SLC 模式写入下容量减半。
- 高密度设计下，一组 NAND Flash Cell 可以同时并发读写。所以有了读写页 2KiB/4KiB 这样的容量。 页面越大，存储密度越高，为了降低成本厂商都希望提高读写页的大小。
- 为了避免添加额外导线，NAND Flash Cell 是使用基板上加负电压的方式擦除 floating gate 中的二进制位的，所以擦除操作没法通过地址线选择特定 Cell 或者读写页，于是整块擦除有块大小。
- 写入操作对 SLC 单个 Cell 而言，就是把 1 置 0 ，而擦除操作则是把整块置 1 。SLC 可以通过地址线单独选择要写入的 Cell ，MLC 则把不同页的二进制放入一个 Cell ，放入时有顺序要求， 先写处于高位的页，再写低位的。所以 MLC 中不同页面地

址的页面是交错在同一组 Cell 中的。

- SLC 其实并没有特别要求擦除块中的写入顺序，只是要求仅写一次（从 1 到 0）。 MLC 则有先写高位页再写低位页的要求。厂商规格中的要求更严格，擦除块中必须满足按页面编号顺序写入。
- 写入和擦除操作是通过量子隧道效应把电子困在 floating gate 中的，所以是个概率事件。通过多次脉冲 可以缩小发生非预期概率事件的可能性，但是没法完全避免，所以需要 ECC 校验纠错。
- 根据 ECC 强度通常有三种 ECC 算法，强度越强需要越多算力：
    - 汉民码 可根据 n bit 探测 $2^n - n - 1$ 中的 2 bit 错误，修正 1 bit 错误。
    - BCH码 可根据 $n * m$ bit 纠错 2^n bit 中的 m bit 错误。
    - LDPC 原理上类似扩展的汉民码，能做到使用更少校验位纠错更多错误。
- 因为 ECC 的存在，所以读写必须至少以 ECC 整块为单位，比如 256 字节或者整个页面。
- 也因为 ECC 的存在， $ECC(\mathrm{0xFF}) \neq \mathrm{0xFF}$ ，空页（擦除后全1的页面）必须特殊处理。所以需要区分写了数据全 1 的页和空页。
- ECC校验多次失败的页面可以被标记为坏页，出厂时就可能有一些坏页，这些由转换层隐藏起来。
- 断电后，也有小概率下束缚的电子逃逸出 floating gate ，时间越长越可能发生可以探测到的位反转。 所以基于 NAND Flash 的存储设备应该避免作为存档设备离线保存。

- 电子逃逸的概率也和温度有关，温度越高越容易逃逸，所以高温使用下会有更高的校验错误率。
- 读取时，因为用相对较高的电压屏蔽没有读取的地址线，有一定概率影响到没被读取的页面中存储的数据。 控制器可能考虑周期性地刷新这些写入后多次读取的页面，这可能和后文的静态擦写均衡一起做。
- 正在写入或者擦除中突然断电的话下，写入中的一整页数据可能并不稳定，比如短期内能正常读取但是难以持续很长时间。

MLC 擦写次数与错误率



上篇讲硬盘的笔记中提到过，硬盘物理存储也有越来越强的校验机制，不过相比之下 NAND Flash 出现临时性校验失败的可能性要高很多，需要控制器对校验出错误的情况有更强的容忍能力。 厂商们制作存储设备的时候，有一个需要达到的错误率目标（比如平均 10^{14} bit

出现一次位反转），针对这个目标和实际物理错误率，相应地设计纠错强度。校验太强会浪费存储密度和算力， 从而提升成本，这里会根据市场细分找折衷点。

2 封装结构

从外部来看，一个闪存盘可能有这样的结构：

从上往下，我们买到的一个闪存盘可能一层层分级：

1. 整个闪存盘有个控制器，其中含有一部分 RAM 。然后是一组 NAND Flash 封装芯片（chip）。
2. 每个封装芯片可能还分多个 Device ，每个 Device

分多个 Die ，这中间有很多术语我无法跟上，大概和本文想讨论的事情关系不大。

3. 每个 Die 分多个平面（Plane），平面之间可以并行控制，每个平面相互独立。从而比如在一个平面内 做某个块的擦除操作的时候，别的平面可以继续读写而不受影响。
4. 每个平面分成多个段（Segment)，段是擦除操作的基本单位，一次擦除一整个段。
5. 每个段分成多个页面（Page），页面是读写操作的基本单位，一次可以读写一整页。
6. 页面内存有多个单元格（Cell），单元格是存储二进制位的基本单元，对应 SLC/MLC/TLC/QLC 这些， 每个单元格可以存储一个或多个二进制位。

以上这些名字可能不同厂商不同文档的称法都各有不同，比如可能有的文档把擦除块叫 page 或者叫 eraseblock 。随着容量不断增大，厂商们又新造出很多抽象层次，比如 chip device die 这些， 不过这些可能和本文关系不大。如果看别的文档注意区别术语所指概念，本文中我想统一成以上术语。 重要的是有并行访问单元的平面（Plane）、擦除单元的段（Segment）、读写单元的页（Page）这些概念。 抽象地列举概念可能没有实感，顺便说一下这些概念的数量级：

1. 每个 SSD 可以有数个封装芯片。
2. 每个芯片有多个 Die 。
3. 每个 Die 有多个平面。
4. 每个平面有几千个段。比如 2048 个。
5. 每个段有数百个页到几千页，比如 128~4096 页，

可能外加一些段内元数据。

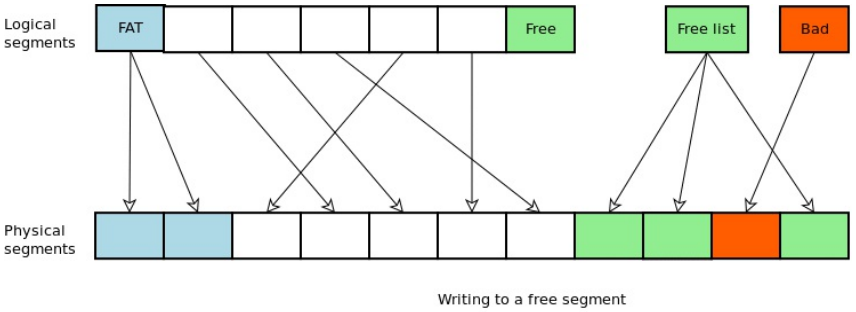
6. 每个页面是 2KiB~8KiB 这样的容量，外加几百字节的元数据比如 ECC 校验码。

和硬盘相比，一个闪存页面大概对应一个到数个物理扇区大小，现代硬盘也逐渐普及 4KiB 物理扇区， 文件系统也基本普及 4KiB 或者更大的逻辑块（block）或者簇（cluster）大小，可以对应到一个闪存页面。 每次读写都可以通过地址映射直接对应到某个闪存页面，这方面没有硬盘那样的寻址开销。 闪存盘的一个页面通常配有比硬盘扇区更强的 ECC 校验码，因为 NAND 单元格丧失数据的可能性比磁介质高了很多。

闪存有写入方式的限制，每次写入只能写在「空」的页面上，不能覆盖写入已有数据的页面。 要重复利用已经写过的页面，需要对页面所在段整个做擦除操作，每个段是大概 128KiB 到 8MiB 这样的数量级。每个擦除段需要统计校验失败率或者跟踪擦除次数，以进行擦写均衡（Wear Leveling）。

3 擦写均衡（Wear Leveling）和映射层（Flash Translation Layer）

Animation: wear leveling on SSD drives



擦除段的容量大小是个折衷，更小的擦除段比如 128KiB 更适合随机读写， 因为每随机修改一部分数据时需要垃圾回收的粒度更小；而使用更大的擦除段可以减少元数据和地址映射的开销。 从擦除段的大小这里，已经开始有高端闪存和低端闪存的差异，比如商用 SSD 可能比 U 盘和 SD 卡使用更小的擦除段大小。

闪存盘中维护一个逻辑段地址到物理段地址的映射层，叫闪存映射层（Flash Translation Layer ）。每次写一个段的时候都新分配一个空段， 写完后在映射表中记录其物理地址。映射表用来在读取时做地址转换，所以映射表需要保存在闪存盘控制器的 RAM 中，同时也需要记录在闪存内。具体记录方式要看闪存盘控制器的实现，可能是类似日志的方式记录的。

「段地址映射表」的大小可以由段大小和存储设备容量推算出来。比如对一个 64GiB 的 SD 卡，如果使用 4MiB 的段大小，那么需要至少 16K 个表项。假设映射表中只记录 2B 的物理段地址， 那么需要 32KiB 的 RAM 存

储段地址映射表。对一个 512GiB 的 SSD ，如果使用 128KiB 的段大小， 那么至少需要 4M 个表项。记录 4B 的物理段地址的话，需要 16MiB 的 RAM 存储地址映射，或者需要动态加载的方案只缓存一部分到 RAM 里。控制器中的 RAM 比 NAND 要昂贵很多，这里可以看出成本差异。

除了地址映射表，每个物理段还要根据擦除次数或者校验错误率之类的统计数据，做擦写均衡。有两种擦写均衡：

- 动态擦写均衡（Dynamic Wear Leveling）：每次写入新段时选择擦除次数少的物理段。
- 静态擦写均衡（Static Wear Leveling）：空闲时，偶尔将那些许久没有变化的逻辑段搬运到 多次擦除的物理段上。

低端闪存比如 SD 卡和 U 盘可能只有动态擦写均衡，更高端的 SSD 可能会做静态擦写均衡。 静态擦写均衡想要解决的问题是：盘中写入的数据可以根据写入频率分为冷热， 总有一些冷数据写入盘上就不怎么变化了，它们占用着的物理段有比较低的擦除计数。 只做动态擦写均衡的话，只有热数据的物理段被频繁擦写，加速磨损， 通过静态擦写均衡能将冷数据所在物理段释放出来，让整体擦写更平均。 但是静态擦写均衡搬运数据本身也会磨损有限的擦写次数，这需要优秀的算法来折衷。

除了擦写均衡用的统计数据外， FTL 也要做坏块管理。闪存盘出厂时就有一定故障率，可能有一部分坏块。 随着消耗擦写周期、闲置时间、环境温度等因素影响，也会遇到一些无法再保证写入正确率的坏块。NAND Flash 上因为量子隧道效应，偶尔会有临时的校验不一致，遇到这种情况，除了根据 ECC 校验恢复数据，FTL 也负责尝试对同一个物理段多次擦除和读写，考察它的可用性。排除了临时故障后， 如果校验不一致的情况仍然持续，那么需要标注它为坏块，避免今后再写入它。

出厂时，闪存盘配有的物理段数量就高于标称的容量，除了出厂时的坏块之外，剩余的可用物理段可以用于 擦写均衡，这种行为称作 Over Provisioning 。除了盘内预留的这些空间，用户也可以主动通过分区的方式或者文件系统 TRIM 的方式预留出更多可用空间， 允许 FTL 更灵活地均衡擦写。

4 段内写入顺序与垃圾回收策略

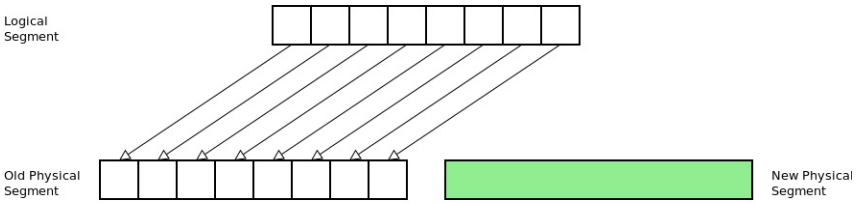
段是闪存盘的擦写单元，考虑到段是 128KiB ~ 8MiB 这样的数量级，现实中要求每次连续写入一整段的话，这样的块设备接口不像硬盘的接口，不方便普通文件系

统使用。所以在段的抽象之下有了更小粒度的页面抽象， 页面对应到文件系统用的逻辑块大小，是 2KiB~8KiB 这样的数量级，每次以页面为单位读写。

写入页面时有段内连续写入的限制，于是需要段内映射和垃圾回收算法，提供对外的随机写入接口。 写入操作时， FTL 控制器内部先「打开（open）」一个段，等写入完成，再执行垃圾回收「关闭(close)」一个段。写入过程中处于打开状态的段需要一些额外资源（RAM 等）跟踪段内的写入状况，所以闪存盘同时能「打开」的段数量有限。并且根据不同的垃圾回收算法，需要的额外资源也不尽相同，在 Optimizing Linux with cheap flash drives 一文中介绍几种可能的垃圾回收算法：

4.1 线性写入优化

Animations: linear-access optimized



假设写入请求大部分都是连续写入，很少有地址跳转，那么可以使用线性优化算法。

- Open：当第一次打开一个段，写入其中一页时，分配一个新段。如果要写入的页不在段的开头位置，那么搬运写入页面地址之前的所有页面到新段中。
- Write: 在 RAM 中跟踪记录当前写入位置，然后按顺序写下新的页面。
- Close: 最后搬运同段中随后地址上的页面，并关闭整段，调整段映射表。

如果在段内写入了几页之后，又跳转到之前的位置，那需要在跳转时关闭当前段写入（并完整搬运剩下的页面），然后重新打开这一段，搬运调转地址之前的页面，从跳转的页面位置开始写入。

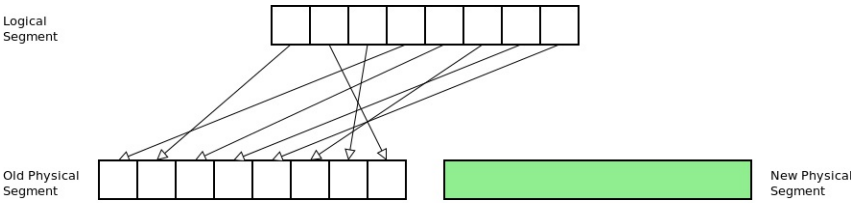
线性优化算法的好处在于：没有复杂的页面地址映射，段内的逻辑页面地址就是物理页面地址。读一页的时候根据页面偏移和当前写入位置就能判断读新物理段还是老物理段。遇到突然断电之类的情况，即使丢失最近写入的新物理段，老物理段的数据仍然还在，所以没必要保存 RAM 中的地址映射到闪存元数据中。

线性优化算法的坏处是:每遇到一次乱序的写入，都要整段执行一次搬运，造成 写入放大（Write Amplification）。

一些文档中，将这种地址映射垃圾回收方式叫做「段映射（Segment Mapping）」，因为从 FTL 全局来看只维护了擦写段的地址映射关系。

4.2 段内地址映射

Animations: block remapping



对需要随机乱序写入的数据，可以使用段内地址映射。方式是额外在段外的别的闪存区域维护一张段内地址映射表， 像段地址一样，通过查表间接访问页面地址。

- Open: 分配一块新的段，同时分配一个新的段内映射表。
- Write: 每写入一页，在段内映射表记录页面的在新段中的物理地址。
- Close: 复制老段中没有被覆盖写入的页到新段，并记录在段内映射表中，然后释放老段和老的段内映射表。

也就是说同时维护两块不同大小的闪存空间，一块是记录段数据的，一块是记录段内地址映射表的， 两块闪存空间有不同的写入粒度。可以在每个物理段内额外留出一些空间记录段内地址映射表，也可以在 FTL 全局

维护一定数量的段内地址映射表。 每次读取段内的数据时，根据映射表的内容，做地址翻译。新段中页面的排列顺序将是写入的顺序， 而不是地址顺序。

根据实现细节，段内地址映射可以允许覆盖写入老段中的页面，但是可能不允许覆盖写入新段（正在写入的段） 中已经写入的页面，遇到一次连续的写请求中有重复写入某一页面的时候，就需要关闭这一段的写入，然后重新打开。

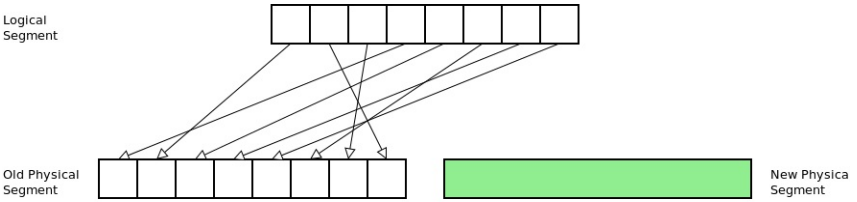
段内地址映射的优点是：支持随机写入，并且只要段处于打开状态，随机写入不会造成写入放大（Write Amplification）。

缺点是：首先地址映射这层抽象有性能损失。其次遇到突然断电之类的情况， 下次上电后需要扫描所有正打开的段并完成段的关闭操作。

和「段映射」术语一样，在一些文档中，将这种段内地址映射的方式叫做「页面映射（Page Mapping）」，因为从 FTL 全局来看跳过了擦写段这一层，直接映射了页面的地址映射。

4.3 日志式写入

Animations: data logging



除了大量随机写入和大量连续写入这两种极端情况，大部分文件系统的写入方式可能会是对某个地址空间 进行一段时间的随机写入，然后就长时间不再修改，这时适合日志式的写入方式。

日志式的写入方式中写入一段采用三个物理段：老物理段，用于日志记录的新物理段，和垃圾回收后的段。

- Open: 分配一块新的段。可能额外分配一个用于记录日志的段，或者将日志信息记录在数据段内。
- Write：每写入一页，同时记录页面地址到日志。
- Close：再分配一个新段执行垃圾回收。按日志中记录的地址顺序将数据段中（新写入）的页面或者老段中 没有被覆盖的页面复制到垃圾回收结束的新段中。

日志式写入在写入过程中像段内地址映射的方式一样，通过日志记录维护页面地址映射关系， 在写入结束执行垃圾回收之后，则像线性写入的方式一样不再需要维护页面映射。 可以说日志式写入某种程度上综合了前面两种写入方式的优点。

日志式写入的优点：允许随机顺序写入，并且在执行垃圾回收之后，不再有间接访问的地址转换开销。

日志式写入的缺点：触发垃圾回收的话，可能比段地址映射有更大的写入放大（Write Amplification）。

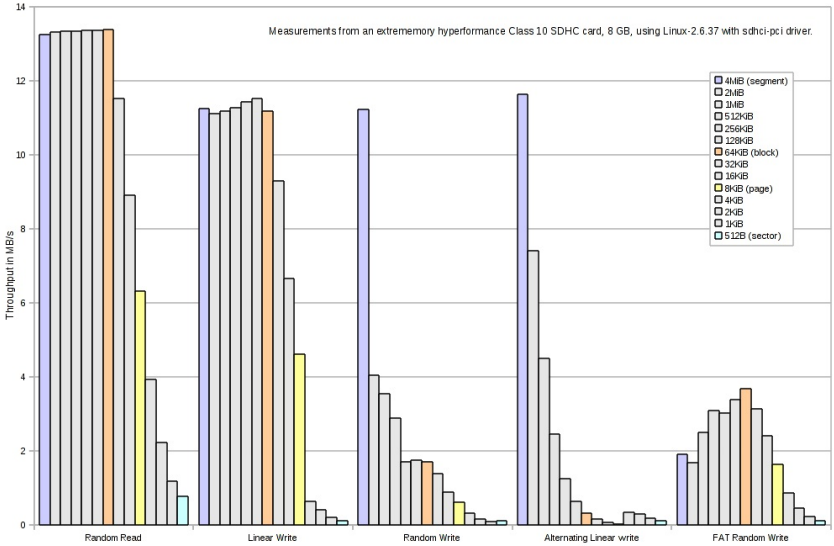
在一些文档中，将这种日志式写入方式称作「混合映射（Hybrid Mapping）」，因为在段开启写入期间行为像页面映射， 在段关闭写入后行为像段映射。

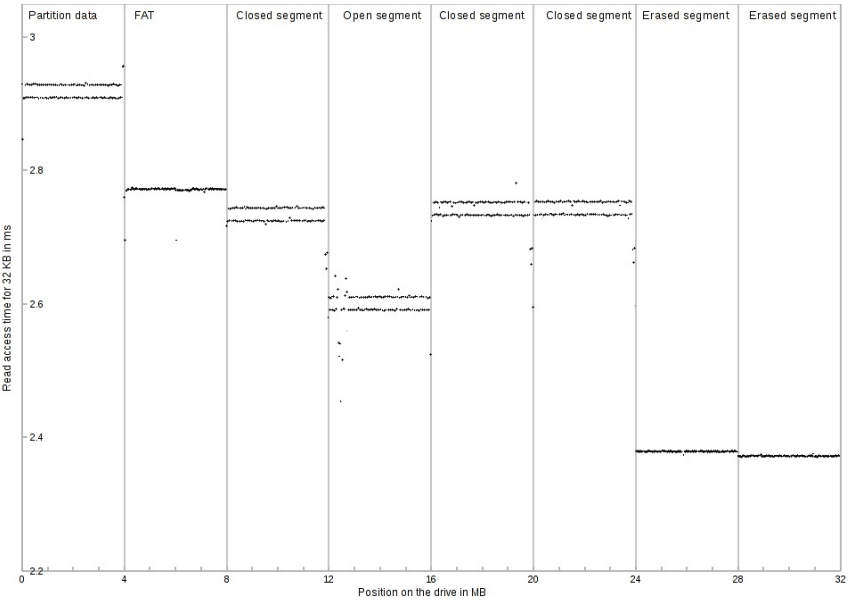
5 针对特定写入模式的优化

上述三种地址映射和垃圾回收方式，各有不同的优缺点，根据数据块的写入模式可能需要挑选相应的策略。 并且「全局段地址映射表」、「段内页面地址映射表」、「写入页面地址日志」之类的元数据因为频繁修改， FTL 也可能需要用不同的策略来记录这些元数据。这里面向不同使用场景的闪存设备可能有不同的 FTL 策略，并且 FTL 可能根据逻辑地址来选择哪种策略。

5.1 混合垃圾回收策略

Performance measurements on a class 10 SDHC card





用来记录照片、视频等的 SD 卡、microSD、U盘等设备可能根据数据的逻辑地址，为特定文件系统布局优化， 这里特定文件系统主要是指 FAT32 和 exFAT 这两个 FAT 系文件系统。 FAT 系文件系统的特点在于， 地址前端有一块空间被用来放置 文件分配表(File Allocation Table) ，可以根据文件系统簇大小和设备存储容量推算出 FAT 表占用大小，这块表内空间需要频繁随机读写。对 FTL 自身的元数据，和 FAT 表的逻辑地址空间，需要使用「段内地址映射」来保证高效的随机读写， 而对随后的数据空间可使用「线性写入优化」的策略。

右侧上图有张性能曲线，测量了一个 class 10 SDHC 卡上，不同读写块大小时，顺序读取、顺序写入、随机写入、 对 FAT 区域的写入之类的性能差异。下图是测量的读取延迟。可以看出 FAT 区域的随机写入和其余逻辑地址上有明显不同的性能表现。

为容纳普通操作系统设计的 eMMC 和 SSD 难以预测文件系统的读写模式，可能需要使用更复杂的地址映射和垃圾回收策略。 比如一开始假定写入会是顺序写入，采用「线性优化」方式；当发生乱序写入时，转变成类似「日志式写入」 的方式记录写入地址并做地址映射；关闭段时，再根据积累的统计数据判断，可能将记录的日志与乱序的数据 合并（merge）成顺序的数据块，也可能保持页面映射转变成类似「段内地址映射」的策略。

5.2 利用 NAND Flash 物理特性的优化

再考虑 NAND Flash 的物理特性，因为 MLC 要不断调整参考电压做写入， MLC 的写入比 SLC 慢一些，但是可以对 MLC Flash 使用 SLC 式的写入， FTL 控制器也可能利用这一点，让所有新的写入处于 SLC 模式，直到关闭整段做垃圾回收时把积攒的 SLC 日志段回收成 MLC 段用于长期保存。 一些网页将这种写入现象称作「SLC 缓存」甚至称之为作弊，需要理解这里并不是用单独的 SLC Flash 芯片做 writeback 缓存，更不是用大 RAM 做缓存，处于 SLC 模式的写入段也是持久存储的。

5.3 同时打开段数

上述地址映射和垃圾回收策略都有分别的打开（open）、写入（write）、关闭（close）时的操作，闪存盘通常允许同时打开多个段，所以这三种操作不是顺序进行的，某一时刻可能同时有多个段处在打开的状态， 能接受写入。不过一个平面（Plane）通常只能进行一种操作（读、写、擦除），所以打开写入段时， FTL 会尽量让写入分部在不同的平面上。还可能有更高层次的抽象比如 Device、 Chip 、 Die 等等，可能对应闪存盘内部的 RAID 层级。

闪存盘能同时打开的段不光受平面之类的存储结构限制，还受控制器可用内存（RAM）限制之类的。 为 FAT 和顺序写入优化的 FTL ，可能除了 FAT 区域之外，只允许少量（2~8）个并发写入段， 超过了段数之后就会对已经打开的段触发关闭操作（close），执行垃圾回收调整地址映射，进而接受新的写入。 更高端的 SSD 的 FTL 如果采用日志式记录地址的话，同时打开的段数可能不再局限于可用内存限制， 连续的随机写入下按需动态加载段内地址映射到内存中，在空闲时或者剩余空间压力下才触发垃圾回收。

5.4 预格式化

FTL 可能为某种文件系统的写入模式做优化，同时如果文件系统能得知 FTL 的一些具体参数（比如擦除段大小、 读写页大小、随机写入优化区域），那么可能更好地安排数据结构，和 FTL 相互配合。 F2FS 和 exFAT 这些文件系统都在最开头的文件系统描述中包含了一些区域，记录这些闪存介质的物理参数。 闪存盘出厂时，可能预先根据优化的文件系统做好格式化，并写入这些特定参数。

5.5 TRIM 和 discard

另一种文件系统和 FTL 相互配合的机制是 TRIM 指令。TRIM 由文件系统发出，告诉底层闪存盘（或者别的类型的 thin provisioning 块设备）哪些空间已经不再使用， FTL 接受 TRIM 指令之后可以避免一些数据搬运时的写入放大。关于 TRIM 指令在 Linux 内核中的实现，有篇 The best way to throw blocks away 介绍可以参考。

考虑到 FTL 的上述地址映射原理， TRIM 一块连续空间对 FTL 而言并不总是有帮助的。 如果被 TRIM 的地址位于正在以「段内地址映射」或「日志式映射」方式打开的写入段中，那么 TRIM 掉一些页面可能减少垃圾回收时搬运的页面数量。但是如果 TRIM 的地址发生在已经垃圾回收结束的段中， 此时如果 FTL 选择立刻对被 TRIM 的段执行垃圾回收，可能造成更多写入放大， 如果选择不回收只记录地址信息，记录这些地址信息也需要耗费一定的 Flash 写入。 所以 FTL 的具体实现中，可能只接受 TRIM 请求中，整段擦除段的 TRIM ，而忽略细小的写入页的 TRIM 。

可见 FTL 对 TRIM 的实现是个黑盒操作，并且 TRIM 操作的耗时也非常难以预测，可能立刻返回， 也可能需要等待垃圾回收执行结束。

对操作系统和文件系统实现而言，有两种方式利用 TRIM ：

1. 通过 discard 挂载选项，每当释放一些数据块时就执行 TRIM 告知底层块设备。
2. 通过 fstrim 等外部工具，收集连续的空块并定期发送 TRIM 给底层设备。

直觉来看可能 discard 能让底层设备更早得知 TRIM 区域的信息并更好利用，但是从实现角度来说， discard 不光影响文件系统写入性能，还可能发送大量被设备忽略掉的小块 TRIM 区域。可能 fstrim 方式对连续大块的区间执行 TRIM 指令更有效。

6 TL;DR 低端 vs 高端

标题中的疑问「SSD就是大U盘？」相信看到这里已经有一些解答了。 即使 SSD 和U盘中可以采用类似的 NAND Flash 存储芯片，由于他们很可能采用不同的 FTL 策略，导致在读写性能和可靠性方面都有不同的表现。（何况他们可能采用不同品质的 Flash ）。

如果不想细看全文，这里整理一张表，列出「高端」闪存盘和「低端」闪存盘可能采取的不同策略。 实际上大家买到的盘可能处于这些极端策略中的一些中间点，市场细分下并不是这么高低端分明。 比如有些标明着「为视频优化」之类宣传标语的「外置SSD」，对消费者来说可能会觉得为视频优化的话一定性能好， 但是理解了 FTL 的差异后就可以看出这种「优化」只针对线性写入，不一定适合放系统文件根目录的文件系统。

参数	低端	高端
段大小	8MiB	128KiB
段地址映射	静态段映射	日志式映射

随机写入范围	FTL元数据与FAT表区域	全盘
同时打开段数	4~8	全盘
物理段统计信息	无（随机挑选空闲段）	擦除次数、校验错误率等
擦写均衡	动态均衡（仅写入时分配新段考虑）	静态均衡（空闲时考虑搬运）
写入单元模式	TLC	长期存储 MLC，模拟 SLC 日志

介绍完闪存类存储，下篇来讲讲文件系统的具体磁盘布局，考察一下常见文件系统如何使用 HDD/SSD 这些不同读写特性的设备。