FTL Write

一、Backgroud

2754 的空间管理将分为 user lu\middle lu\sys lu\boot lu\rpmb lu，其中 middle lu\sys lu 主要 FW 内部自己的管理空间，而 host 会直接访问的是 user lu\boot lu\rpmb lu，这些空间的写都将经过 data cache 放入其中的 cache node，同时 data cache 依靠自己的机制和策略，将 cache node 以 FEReqNode 的形式放入 queue。cache 当前的设计是将有两个 queue：分高优先级和低优先级，高优先级存放 read 和 HPL，低优先级存放 write 和其他。而 FTL 最终将从这两个 queue 上摘下 FTLReqNode，进行不同的读写等操作处理。

二、FTLReqNode Queue Process

- 由于目前计划设计是两个 queue 将读写分开，期望读比写更高优先级的被调度执行，FTL 将无法保证读写的互斥情况（读比写先执行，读到老数据？），所以需要 data cache 去做读写互斥。
- Queue 里面放入了 HPL 和其它操作的 node，这些 node 与 IO node 的处理优先级？
- 针对 write node 的管理：

目前由于一个 write queue 里面还放入了其它 node，导致 FTL 还需遍历 queue 把所有 write node 摘出来管理，设计是将 queue 中所有 write node 全部取出用链表管理。或者 cache 进行优化设计，让 IO 读写的 queue 更纯粹？

- write node 的组装：主要是针对 user lu，boot lu 和 rpmb lu 是 SLC spb，最小写单元就是一个 page，而 user lu 的 SLC 和 TLC 的 PU 考虑到 die 大小、坏块、raid 等情况，其大小并不是固定的，所以需要将 write node 组装成符合 PU 的大小。比如一个 write node 32K size，而 TLC 的 PU 是 192K，那一笔 FTL 请求就需要取 6 个 write node。

	CH0/CE0/PL0	CH0/CE0/PL1	CH0/CE0/PL2	CH0/CE0/PL3	CH1/CE0/PL0	CH1/CE0/PL1	CH1/CE0/PL2	CH1/CE0/PL3	CH0/CE1/PL0	CH0/CE1/PL1	CH0/CE1/PL2	CH0/CE1/PL3	CH1/CE1/PL0	CH1/CE1/PL1	CH1/CE1/PL2	CH1/CE1/PL3
Page 0	D0	D1	D2	D3	D12	D13	D14	D15	D24	D25	D26	D27	D36	D37	D38	Parity 0
Page 1	D4	D5	D6	D7	D16	D17	D18	D19	D28	D29	D30	D31	D39	D40	D41	Parity 1
Page 2	D8	D9	D10	D11	D20	D21	D22	D23	D32	D33	D34	D35	D42	D43	D44	Parity 2
RAID																

- io write node 取的规则，分两种情况：

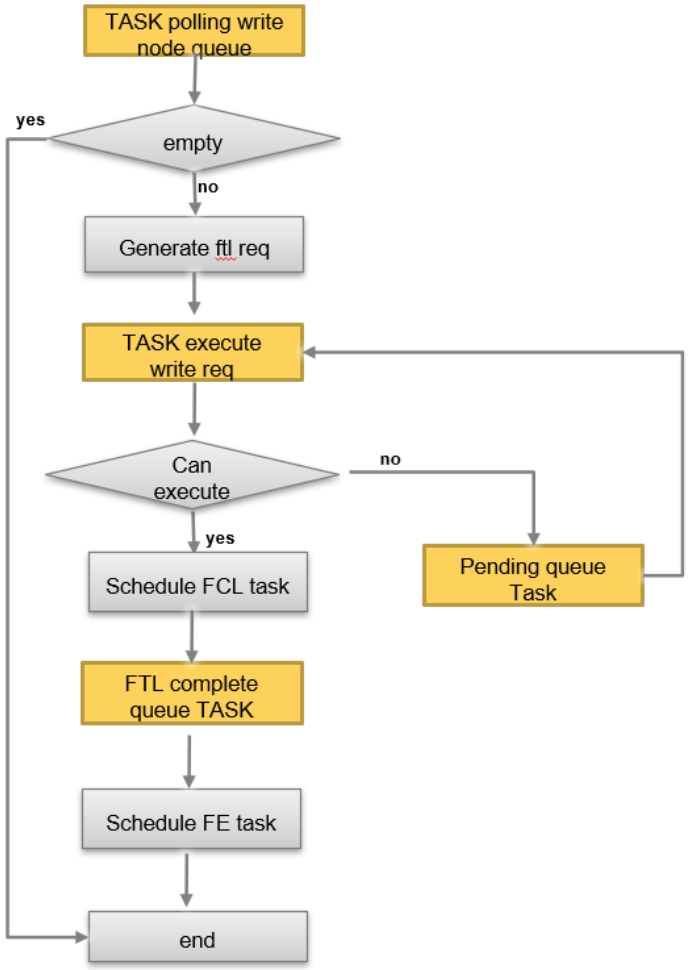
- normal write：只要 queue 里面有 write node，将一直按照 PU 大小去取，满足一个 PU 就生成 ftl req 去执行，直到取到剩余 write node 无法满足 PU 大小时停止，或者拿不到 ftl req 资源时停止。
- flush all：主要是 Idle flush、PMU flush、Shutdown Flush、Host flush。所有 write node 将全部取出写下去，先按 PU 大小取，满足一个 ftl req 就执行，而剩余不满足 PU 的 write node 也会取出，同时 FTL 会 PAD dummy data 成一个 PU 然后写下去。

三、Write Task schedule

为适配当前 FW 架构设计，整个写 flow 也将以 task 形式进行拆分。

首先，ftl req 也将拥有两个队列来管理，分别是 pending queue 和 complete queue。Pending queue 将管理由于各种原因无法执行的 ftl req，complete queue 将用来处理完成 FCL 操作后回来的 ftl req。注意此处 FTL 没有 submit queue，因为这个其实就是 cache 创建的 ftl write node queue，从 write node queue 取出生成 req 后会直接执行，无法执行的话会进入 pending queue，不再需要单独的 submit queue 来管理。

所以总共 3 个 queue：write node queue/pending queue/complete queue，每个 queue 将拥有一个 task 来处理，再加上本身业务的执行需要一个 task，一共 4 个 task。写流程这 4 个 task 的简单调度过程：



四、Write flow detail step

- Get PU info

User lu 的 PU 可能不是固定的，写之前需要获取该信息，其它 LU 的最小写单元就是一个 page，不需要去获取。

- Generate ftl req and submit to execute。

将多个 write node 的 buffer 及 laa list，放入 buffer sgl 及 lrange，然后生成写 req 之后调度 ftl req execute task。

- Judge ftl req 。

主要是判断当前 LU 被 block 条件，如：LU 正在 flush、lack page resource、p2l resource、flow control 等等，若 LU 被 block，当前 req 将无法执行，会挂入 pending

queue，被 pending 之前 req 已经拿到的资源需要释放。放入 pending queue 的 req，会被 pending TASK 重新调度执行。

- Check flow control

目前计划做一个简单的 fc，主要用来 balance between io write and gc，大致策略是通过 GC threshold and perf 去控制 io。

- Reserve resource
- Reserve page：若 page 不满足 PU 写，req 挂入 pending queue，同时 trigger allocation task 去分配 SPB。
- Reserve p2l entry：若 p2l entry 不够，req 挂 pending queue，同时 trigger p2l merge TASK，以释放 p2l resource.
- Reserve raid entry

因为资源不够而需要 pending 的 req，已经拿到的资源需要释放出来。

- Assign page

真正的分配物理 page，生成 PAA

- Setup meta \Setup ncl cmd \ setup RAID

填 meta info，填 ncl cmd 需要的信息，若支持 RAID，需要完成 RAID 相关的 setup

10、Submit to fcl to execute

11、FCL 执行将完成操作的 req push 到 FTL complete queue

Fcl cmd 中需要记住自己的 caller 是哪个 ftl req，done 后将 ftl req push 到 queue。

12、FTL task 处理 complete queue，执行 write done 操作，write done 分两种：

Early done:

- 会先直接 call back 前端 io done，执行 free write node、cache node 等操作。
- FCL 真正 complete 回来后，执行与 normal done 同样的操作，但不再 call back FE。

Normal done:

A、check FCL 执行结果，若有 err，将执行 user write err handling

B、update mapping，可能会 trigger flush p2l 或者 trigger merge

C、release lock，若该 lock block 其它 req，将 trigger 该 req 执行

D、release req

关于是否做 early done 的 FTL 与 FCL 的交互：需要 early done 的 req，将在 ncl cmd 置一个 try early done 的 flag，FCL 完成 early done 也需要置一个 flag 告诉 FTL。未置 flag 都当做 normal done 处理。

五、FUA write

当 req 取到的包含设置 FUA 的 write node 时，该 req 将设置 cache off write flag，并在 fcl cmd 上设置 no cache program 的 flag。

六、write err handling

write err 发生后，不同的情况将有不同的处理方式。

- 该 write 是 normal done：set grown defect\force close spb\GC SPB at later。如果需要 retry，将分配新的 SPB 把数据重新写入。
- 该 write 是 early done：set grown defect\force close spb\GC SPB at later。在 enable

RAID feature 的情况下，如果需要 retry，将通过先通过 RAID 恢复数据，然后写入新的 SPB。若未开 RAID，数据将丢失，无法 retry。

FTL Write Booster Design

一、功能说明

Write Booster Buffer 是 FTL 提供的一片高性能存储空间（SLC），在用户指定的特定数据写入时，使用该空间进行存储，以提供更好的存储速度和用户体验。

二、空间分配

目前，FTL 已有的空间分配策略为：优先为用户提供 SLC 空间(slc_cache & dynamic slc)，当 SLC 空间耗尽时，再使用 TLC 空间进行存储。

即，当前的空间分配策略不区分用户数据的类型，采用先来先分配的原则。

在开启 Write Booster 功能后，FTL 需要识别用户数据的类型，对于写入 booster buffer 中的数据，在其配置容量可用的情况下，需要使用 SLC 进行存储。

FTL 提供的 Write Booster 容量相关的接口:

1. 设置 write booster buffer 容量：

由用户配置，FE 通知 FTL， FTL 根据配置项中的最大 write booster buffer size 以及当前 pool 中实际可用的 SLC 空间，判决是否可以配置成功；

若剩余 SLC 空间满足需求，将该 SLC 空间配额预留给 writeBooster，优先使用 SLC_POOL 中的空间，然后使用 TLC POOL 中 dynamic SLC 的空间。剩余的空间为非 write booster 空间

（该空间中也可以包含 SLC）。

2. 查询 write booster buffer 当前容量：

FTL 支持查询当前 write booster buffer 的总体容量，默认情况下该容量等于配置容量。当 NAND 空间不足时，非 write booster 空间可以抢占 write booster 空间，

即， write booster buffer 的当前容量会相应减少，同时会减少同等容量的 write booster available buffer。

3. 查询 write booster buffer 可用容量：

FTL 支持查询 write booster buffer 可用容量，可用容量会随着写入 write booster buffer 的数据增多而相应的减少，同时也会受 2 中提到的因素减少。

当可用容量为 0 时，后续的 IO 将不再写入 write booster buffer。

4. 查询 write booster buffer life：

FTL 支持查询 write booster buffer 生命周期（百分比展示）。用 buffer 中已使用的 SLC 和 TLC 的各自生命周期加权后等到实时的生命周期。

三、 SPB 分配

目前，USER LU 只有一个 ASPB 用于提供 HOST IO 的写入。使能 Write Booster 功能后，存在 HOST IO 同时写入 Write Booster Buffer 空间和非 Write Booster Buffer 空间。

因此，USER LU 需要新增一个 ASPB 存储 Write Booster Buffer 的写入。则，相应需要新增的功能有：

1. 新增 Write Booster PageAllocate 管理：

用于管理 Write Booster ASPB 上的 page 分配；

2. 新增 Write Booster Spb apply 管理:

用于管理 Write Booster Spb 从 POOL 中的分配逻辑， 支持从 SLC POOL 和 TLC POOL 中进行 SPB 的分配

3. 新增 Write Booster Rlut 管理：

用于记录 Write Booster Aspb 的 P2L 信息。

四、IO 路径

1. HOST 写：

a. FE WriteNode 上新增标记，用于识别 Write Booster 写和非 Write Booster 写：

规则： 同一个 WriteNode 内只能有一种类型的 SubNode。

前后两个 WriteNode 如果为不同类型，前面一个 WriteNode 需要进行 force_Flush（设置为 FUA）

两种类型的 WriteNode 之前需要进行冲突检测

b. UserLu 根据 WriteNode 的类型，分别写入 WriteBooster 的 ASPB 和 normal ASPB，同时将 mapping 写入各自的 RLUT：

那么会存在一个问题：同一个 LAA 可能会同时存在于两个 RLUT entry 当中，就会涉及到新旧数据的问题。如果处理

该问题的方案如下：

1）、 WriteBooster 的 Rlut entry 直接写入 mapping;

2）、Normal 的 Rlut entry 在 insert mapping 时，需要 搜索 WriteBooster 的处于 busy 状态的 Rlut entry， 将所有相同的 LAA 都设置为无效值，

即需要保证 WriteBooster Rlut entry 中是最新的 mapping。

2. CONVERT

当 RLUT entry 写满，或者有 trim 命令时，会触发 RLUT entry 的 convert。 新增 WriteBooster Rlut entry 后，在触发 convert 流程时，需要同时

触发 WriteBooster Rlut entry 和对应的 Normal Rlut entry，且需要按照先执行 Normal Rlut entry 的 convert， 再执行 WriteBooster Rlut entry 的

convert 的顺序。

而当前 Rlut entry 的 partical-convert 逻辑只适用于 trim 触发的 convert， 并且 partical 与 full entry 的 convert 并没有前后关系。且当前的 full entry

之前的 convert 并没有要求顺序执行，且存在 full、convert 等多个 queue。 为了适应 WriteBooster Rlut entry 的需求，需要重构 RLUT entry 进行

convert 的逻辑：

1）去掉 Rlut full queue， partical queue， 保留 convert queue，添加 convert entry node 结构：

```
typedef struct _RlutConvertNode
{
    RlutEntry_t *Entry;
    U32  EntryStart;
    U32  EntryEnd;
    QTAILQ_ENTRY(_RlutConvertNode) link;
}RlutConvertNode;
```

Node 中包含待 convert 的 entry， 此次 convert 的范围。

同时， Rlut entry 中需要添加记录 convert 信息的结构：

```
typedef struct _RlutConvertInfo_t
{
    U8 ConvertIdx;
    U16 EntryStart[RLUT_ENTRY_MAX_CONVERT_NUM];
    U16 EntryEnd[RLUT_ENTRY_MAX_CONVERT_NUM];
    U32 ConvertSn[RLUT_ENTRY_MAX_CONVERT_NUM];
}RlutConvertInfo_t;
```

每个 Rlut entry 可以被触发多少 convert，需要记录多笔 convert 信息，使用 convertIdx 来记录；

EntryStart 和 EntryEnd 是每次 convert 的范围；

convertSn 是一个全局唯一的 SN，用于标记此次 convert 的序列，用于 recover 时，恢复 convert queue。

2） 当 WriteBooster Rlut entry 写满时，需要将 WriteBooster Rlut entry flush 到 NAND 上，

生成 WriteBooster Rlut entry 的 convert 节点后加入 convert queue，若 Normal Rlut entry 中存在未 convert 的 mapping， 需要

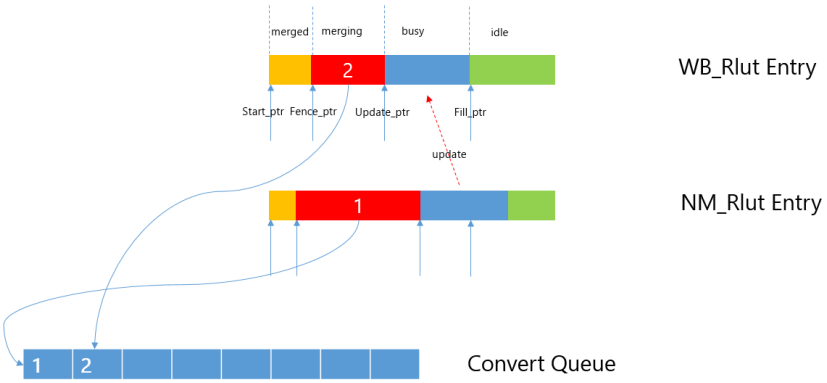
按照相同的操作先处理处理 Normal Rlut entry；

3） 当 Normal Rlut entry 写满时，需要将 Normal Rlut entry flush 到 NAND 上，生成 Normal Rlut entry 的 convert 节点后

加入 convert queue，若 WriteBooster Rlut entry 中存在未 convert 的 mapping， 需要按照相同的操作再处理 WriteBooster Rlut entry；

4）当收到 trim 命令时，需要将两种 Rlut entry 都加入到 convert queue 中。为避免大量 trim 命令触发添加过多的 convert node，

trim 命令触发 convert 的机制修改为当 convert queue 为空时，同时 trim record 不为 0，再执行该操作。



3. Host 读

Host 读执行 lookup mapping 操作时，查找 RLUT 的顺序修改为：

1） 查询 WriteBooster Rlut entry 的 busy 区域；

2） 查询 Normal Rlut entry 的 busy 区域；

3） 倒序查询 convert queue 中的 Rlut entry 的 merging 区域。

Garbage Collection

Garbage Collection(GC) 概述

Nand Flash 空间使用到一定程度之后进行脏数据空间回收，同时要将有效数据搬移到新的位置，将原来脏数据所占用的空间重新释放出来。

两种类型 GC：

BGC： Backend GC, FTL idle 时 trigger

FGC： 前台 GC

选源

1. Wear leaving

   主要是选择 EPC counter 小的 SPB

1. free block 不足

   主要是选择 valid counter 小的 SPB ，如果 valid counter 相差不大（SPB DU 个数千分之一）但是 SN 差距较大就选择 SN 比较小的 SPB.

1. 不稳定 block

   主要选择有 read/write err 或者 BER risk block

主要流程

1. Find Valid PAA

GC 目的是将有效数据搬移到新的位置，所以在搬移前须明确哪些数据是有效。

1.
   1. User lu

      通过读取 candidate SPB P2L 建立

1.
    1.
        1. Read P2L
        2. Read L2P
            1. 通过 P2L 查找相关 L2P
            2. 每当搜集到的 L2P 到达一定个数后才会去 load L2P table
        3. L2P check
            1. L2P PAA 若属于当前正在处理的 P2L 则此 PAA 列为有效
    2. Mid lu

通过读取 candidate SPB L2P 建立，由于 mid lu size 已经不大且没有 P2L， 所以直接 scan mid lu 的所有 L2P 来 check 有效数据

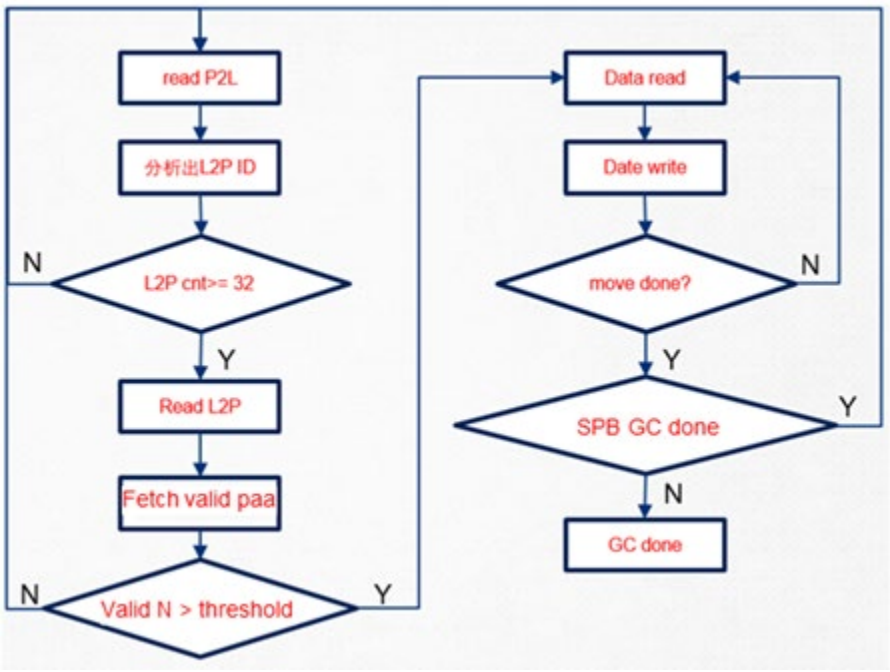
-
    - Sys lu

读取 lut table（in TCM）建立，sys lu 的 maaping size 很小（only 2K for 2T capacity），所以直接 sys lu lut table 可以直接决定有效数据

2. Data move

根据有效数据 向 BE 发 read->write 请求。每次读写 DU 个数需兼顾 resource 及 candidate SPB 的 WFU。

Fetch valid PAA data move



GC 过程中，user/mid/sys lu 的 gc 在 Date move 部分的实现 flow 相同，只有在 fetch valid paa 的过程 有差异。

GC write 更新 P2L 时为优化 host write performance，需要记录 GC destination SPB 所有 DU 的 source SPB，

当 merge GC SPB 的 P2L 时如果发现所记录的 source SPB 已经不是 L2P 中的 PDA 对应 SPB 即认为该 DU 在 GC 后又有 host 写入所以放弃该 du 的 merge，

优势是：host write merge 时不需要再去检查 GC spb 的 P2L。

Schedule

User/mid/sys 三个 lu 的 gc 如果全部都触发，则按照 sys > mid > user 的优先级顺序调度然后发给送给 folding，

不过鉴于 resource 原因同一时间只允许一个 GC 运行，mid/sys GC 可以抢占 user lu GC。

异常处理

1. GC read err（读 PDA err）
    1. Mask 坏块

2. 通过 P2L 或者 L2P 找到 err PDA 对应 LDA
3. Check 这个 LDA 是否已经/正在被上层写/trim（此检查可以删掉）
    1. 如果是， skip
    2. 如果否，insert err PDA of LDA（P2L 如何实现）
4. Data 处理
    1. User LU

Data 还是会继续写入 nand，但是要保证修改 mapping

1.
    1.
        1. Internal LU
            1. Buffer set 为 err pda，写入 nand
2. 读 P2L err

放弃 P2L， 改为读 LU 所有的 L2P 来 fetch valid paa

3. Abort/cancel/stop

三个 LU 的 GC req 可单独设置是否 abort/cancle/stop

GC 在运行时会随时关注是否有被 Abort/cancel/stop

SPOR

一、LUN 的重建：

重建规则：

- 对于所有 LUN，其 ASPB 完成扫描重建后，在补写一部分 Pad data 后继续作为 ASPB 使用。
- 在重建阶段，重建回来的数据使用内存缓存，在重建完成时，再 merge & flush 到 NAND 上去，以防止反复掉电的场景下导致 SPB 快速耗尽。

1. SysTemLu：

触发条件： 上电后，读到的 SystemLuDesc 描述 Lu 的 flag 为 dirty。

重建过程：

1）恢复 ASPB， readySpb 等内存信息：

通过 LuDesc 中记录的 ASPB 的 UpPtr 信息，恢复控制结构上的 ASPB 基本信息：

```
16:
17: typedef struct _AspbDesc_t
18: {
19:   SpbId_t spbID;  ///< SPB ID
20:   U32 u32UpPtr;   ///< determined fence before convert
21:   U32 u32WrPtr;   ///< determined write pointer before convert
22: } AspbDesc_t;
23:
24:
```

```
typedef struct _ActiveSpb_t
{
    SpbId_t SpbId;   ///< SPB id of active SPB
    U8 u8State;      ///< ASPB_ST_xxx
    U8 u8Flags;      ///< ASPB_F_xxx
    U32 u32WritePtr;        ///< current write pointer in page
    U32 u32MaxWritePtr; ///< max page count
    U32 u32CompleteCnt; ///< how many pages are programmed

    U32 u32OccupyPgCnt;
    U32 u32MaxOccupyCnt;

    FtlDefect_t ftlDf;  ///< defect bitmap of active SPB
    U8 u8Mode;          ///< 0: slc 1:xlc
    U8 u8Rsvd0;
    U16 u16CntWrDu;
} ActiveSpb_t;
```

2）将 readySpb 转为 ASPB：

因为 readSpb 可能已知作为 ASPB 进行数据的存储，只是还未更新 LuDesc 描述，故需要在重建流程中将 readSpb 转换为 ASPB，

并执行 ASPB 的重建。

3）SLUT 的恢复：

对于 SystemLu 而已，其 mapping(SLUT)为全内存缓存。在对 SLUT 恢复时，对 ASPB 从 AspbDesc_t 中记录的 UpPtr 开始作为扫描

的起始，以连续出现一段 erase page 时作为扫描结束。通过扫描获取 spare 中的 LAA 信息，从而完成对 SLUT 的恢复。

4） ASPB 补 pad：

由于当前的 ASPB 使用策略为继续往后使用，故需要对 ASPB 补充一定量的 pad 数据，来稳定 ASPB 的已有数据，然后该 ASPB 继续提供

空间。补 pad 数据时，需要同步修改 activeSpb_t 上的一些列相关的控制和状态信息。若发现最新的位置已经有补充 pad 数据，则无需

再补。

SystemLu 重建完成后，所有的最新信息都存储在内存中，此时不会对 NAND 进行任何的操作（除了 PAD 数据），若在此期间出现再次

掉电，不会有任何影响，待上电后再重新执行重建流程即可。

2. middleLu：

触发条件： 上电后，读到的 MiddleLuDesc 描述 Lu 的 flag 为 dirty。

重建过程：

1）恢复 ASPB， readySpb 等内存信息：

同 1.1）

2）将 readySpb 转为 ASPB：

同 1.2）

3）mapping 恢复：

采用扫描 ASPB 的方式来恢复 mapping。 对于如果存储最新 mapping 存在一个问题: middleLu 本身的 mapping 并没有全内存缓存，而是通过 LUT 从 NAND 上换入换出

的进行更新。若重建时，也采用 LUT 的方式来更新最新的 mapping 就会在重建过程中对 SystemLu 产生新的写入。那么如果在 middleLu 的重建过程

中又再次发生了掉电，会导致下次重建时，SystemLu 上的 Aspb 数据发生变化，导致其 Aspb 的耗损加快（新的写入数据+重新补 pad 的数据）。同时，

有可能出现下次重建时，systemLu 将 middleLu 的最新 mapping 恢复出来，而 middleLu 对应的数据可能读失败的情况（即 mapping 恢复了，但是对应

的数据读不到）。

为了更好的支持闪掉的场景，考虑对 middleLu 的重建 mapping 不进行下盘，那么如果采用 LUT 的方式进行存储，可能需要大量的 memory（需要视重建数据的离散程度来定），

很可能所有的 memory 都不够用。那么考虑采用 P2L 的方式来进行 mapping 的记录：

重建流程中临时申请一个(或多个)P2L 的 record, 用于记录扫描 ASPB 产生的 new mapping。扫描完成后，该 P2L record 不下盘，依然缓存在内存中。但是需要支持 read，即读

middleLu 时需要查询该 P2L record。故，对当前 read IO 流程有一定的影响，需要 middle read 流程支持查询 P2L record。 该 record 结构体如下：

```
typedef struct _P2lRecord_t
{
    Paa_t StartPaa;
    U32    length;
    Laa_t *LaaArray;
}P2lRecord_t;
```

同时为了减少重建时扫描的数据量，middle Lu 在正常运行时，需要加大下刷 LUT 的频率。可以配置为分配了 N 个 paa 后就将 LUT flush 一次，同时更新 AspbDesc 中的 UpPtr。重建

时的 P2L record 上记录的个数与之前下刷的粒度相匹配。

4） ASPB 补 pad：

同 1.4）

middleLu 重建完成后，只会在内存中生成最新的 P2L record, 此时不会对 NAND 进行任何的操作（除了 PAD 数据）。故在此期间发生掉电，上电后重新执行重建即可。

3. UserLu:

触发条件： 上电后，读到的 UserLuDesc 描述 Lu 的 flag 为 dirty。

重建过程：

1）恢复 ASPB， readySpb 等内存信息：

同 1.1）

2）将 readySpb 转为 ASPB：

同 1.2）

3) RLUT 重建：（mapping 恢复）

为了简化重建逻辑，将当前 RLUT 的 flush 和 merge 逻辑调整为在 flush 完成后再触发 merge。那么重建的时候， 在 ASPB UpPtr 之前的数据不用再扫描。需要将 UpPtr 所在的

RLUT 加载起来，如果能读到，则根据 UpPtr 的位置信息对当前 RLUT entry 进行配置；若读不到，则说明当前 UpPtr 的位置刚好是一个新的 RLUT entry 的位置，直接 get 一个新的 RLUT

entry 并初始化即可。

UpPtr 之后 mapping 需要通过扫描 ASPB data 重新插入 RLUT entry， 从而完成最新 mapping 的恢复。

另，UserLu 的 spb 扫描区别于其他的 Lu， 需要安装写入的粒度来进行扫描： 在一个写入粒度内，若有 DU 读失败，则这边写入的所有 page 都将视为失败。

4） ASPB 补 pad：

同 1.4）

UserLu 重建完成后，只会更新内存中的 RLUT 信息，此时不会对 NAND 进行任何的操作（除了 PAD 数据）。故在此期间发生掉电，上电后重新执行重建即可。

4. 重建信息下盘：

当执行完 UserLu 的重建后，此时所有 LU 的重建回的数据都在内存中，在此之前的阶段发生掉电，都可以上电后重新进行，不会对 ASPB 有多余的空间损耗。在此之后，需要

将重建回的部分信息下刷，包括：SLUT(systemLut mapping), SystemLu Desc, MiddleLu Desc, MiddleLu P2L record。

1） MiddleLu P2L record 触发一次 merge：

将 record 的内容 merge 到 MiddleLu 的 Lut 中， 然后将 Lut 下刷。

下刷会对 SysLu 的 ASPB 产生写入，若此时发生掉电，再次上电后，可能会重建出此次写入的部分数据。故，在写入时， meta 数据中可做特殊标记 atomic_head 和 atomic_tail。

当 SystemLu 扫描 ASPB 的数据时，发现当前的数据若只有 atomic_head 而没有 atomic_tail, 则将扫描回来的 mapping 抛弃。(atomic_tail 一直不会来设置)

在 merge 期间无法处理新来的写 IO，若此时有写 IO 下发，只能将其 pending。可以处理读 IO， 需要 middle read 流程添加对 P2L record 的查询处理。

2）将 SLUT&systemLu Desc&middleLu Desc flush 到 sys log 当中：

此时将 systemLu 和 middleLu 都设置为 clean 状态，通过 sys log 原子的 flush 到 nand 上。由于已经将 systemLu 的 flag 设置为了 clean，且 UpPtr 也会设置到最新的写入位置，故无需

再设置 atomic_tail 标记到 SysTemLu 的 ASPB 上。

3）UserLu 处理：

UserLu 在其 RLUT 完成重建后，触发一次 RLUT 的 flush 和 merge 操作，并设置 FIRST_CONVERT 标记。该标记用于识别是重建后发起的第一次 convert 操作，在此次 convert 过程中，ASPB

上 VC cnt 无条件++，防止上次掉电前， LUT 改了但是 VC 没有改的情况出现。故在发生了掉电后，会出现 spb 的 VC 比正常值大的情况。

存在的问题：

在重建信息下盘的时候若发生掉电，会导致 SystemLu ASPB 发生数据写入，虽然该部分写入的数据可以被 ignore，但是仍然会导致 SPB 的空间损耗。当 SystemLu 所在的 pool 的 free spb 水位

很低时，有分不出 spb 的风险，需要先启动 GC 流程来释放 spb,。此时等待 middle P2L record merge 完成的时间就会变长。有两个优化的方式：

a. 为 systemLu 固定预留一个 spb 用于重建，IO 以及 GC 都无法分配到该 spb。重建过程中， systemLu 通过正常的方式获取不到 spb 时，才来获取该预留的 spb。

b. 取消 middleLu P2L record 的 merge 过程，让 P2L record 支持写 IO。相当于 middleLu 也跟 userLu 一样在内存中维护 RLUT 表，那么相应的 middleLu 也需要有对应的 convert 流程。

a 方案实现方式简单，但是只能缓解不能根治，；b 方案可以彻底解决闪掉耗损 Spb 的问题，可以真的做到在全部重建周期内没有新的写入，且去掉 merge 过程可以不用 pending IO。但是对

现有的 middle 读写冲击太大，所以考虑还是使用 a 方案。

5. Spb vaildCnt 信息

与各 LU 重建强相关的就是 spb 的 VC 信息，在重建过程中涉及到对 spb 的 VC 的调整。目前 spb 的 descinfo 是一起下盘的，就会出现 userLu 触发 Spb descInfo 下盘时，将其他的 LU 的 spb descInfo 也

一并下盘的问题。因为存在多个 LU share 同一个 pool 的情况，很难做到按照 LU 进行 spb descInfo 下盘。考虑为 systemLu 和 middleLu 的 spb 提供一个缓存 spb VC 变更的 array， 在各 Lu 触发 Spb 的

的 DescInfo 下刷的时候将缓存中的值更新到 spb descInfo 中进行下刷，同时清除对应 spb 的 VC 缓存。

则在重建时，对于各 Lu 的 Spb VC 处理：

1）systemLu:

SLUT 与其 SpbDescInfo 同时下盘，不会产生不匹配的场景。

2）middleLu:

在 LUT 更新的过程中， SpbDescInfo 不会时时更新，存在 LUT 更新而 SpbDescInfo 未更新的情况，在掉电后可能出现 Spb VC 偏大。

3）userLu：

RLUT 通过 convert 流程进行更新，期间也会不停的更新 LUT，完成时再修改 SpbDescInfo，同样存在掉电后 spb VC 偏大的问题。

在出现 spb VC 偏大后，可以通过 GC 来得到实际的 VC 给予修正（修正方式需要参考最新的 GC 方案）。

二、 SystemLog 重建

SystemLog 空间由 sub_pool 中的 SLC spb 提供，本身基于 block 进行管理，其 spb id 信息存储在 FRB header section 中。重建过程如下：

1. 从 FRB 中获取到当前正在使用的 Spbs:

   判断哪个 spb 是最新的 spb, 通过读取 meta 信息中的 flush id 判断。

2. 获取正在使用的 block:

   查找到没有写过的 block，上一个 block 就是正在使用的 block。

3. 获取最新的 page：

   通过二分法查找出最后写入的 page。

4. 检查 last_page 是否正确：

   1）该 page 是否为该笔写入的最后一笔，即最后一笔写入是否完整；

   2）该 page 所在的这笔写入是否都能读到。

   若 check 不过，则将 last_page 往前回退（只回退一次），继续进行 check。

5. 根据 mete 信息获取到 header 的位置：

   若 header 地址是有效的 paa, 则读取 paa 内容恢复到 header，同时将 header paa 作为后续扫描结束位置；

   若 header 地址是无效的，则将当前 block 的 page 0 对应的 paa 作为扫描结束位置。

6. 恢复最新的 mapping 和数据:

以 4）中的 last_page 作为起始位置，以 5）中 header 地址作为结束位置进行倒序扫描和恢复数据。

读 page 出错后，会对备份位置进行重读，若重读也出错，则重建失败。

若所有 PageId 都恢复，则提前结束扫描。

三、Frb 重建

1. 根据 SRB 重建的结果，获取 Frb 的 blockId 以及 lastPage。

2. 重建 mapping，从 lastPage 开始往回扫，遇到存放 header 的 page 为止，

从 header 中获取出下盘的 mapping 与之前扫描回的 mapping merge 恢复出最新的 mapping.

3. 根据 mapping 恢复出各个域的数据。

四、Srb 重建

1. 扫描 boot_blk_pool 找到 mgr block:

按照 mgr block 的分配规则，其位于 boot_blk_pool 刚开始的几个 block 中。通过读 meta 识别到是否为 mgr block。

2. 找到 header page 恢复出 block 分配信息：

从 page0 开始读 mgr block, 找到第一个 header page，恢复出分配信息。

3. 恢复出各域的 write block， mirror block 和 spare block

page 的 spare 信息中记录的当前的 page 是写的 write block 还是 mirror block;

spare block 为 erase block

4. 找到各域的 last write page:

同个二分查找的方式，查询各域的 write block 的 last page。

5. 恢复出 Srb mgr block 上的各 section 的 mapping 和 data

从 last page 往前扫描 mgr block 上的 page， 恢复出其个 section 的 mapping 和 data。

TRIM

一、 FE 与 FTL 交互：

FE 对于 trim 命令的处理：

1. 对于小范围的 trim 命令处理，FE 直接将 trim 命令转为写命令即可；

2. 对于大范围的 trim 命令处理：

1）对原始 trim 进行处理，

一个 trim 命令内 range 若有互相重叠的情况，需要对原始 trim 命令进行合并操作；

range 中若包含非 4K 对其的头或者尾，需要将头尾进行拆分后，将对齐部分的 range 下发给 FTL，非对齐部分作为写 IO 处理。（目前应该不涉及非对齐的 场景）

2）trim 命令下发前，若 CACHE 中存在写 node 且还未下发给 FTL 的，需要将与 trim 范围有冲突的 强制 flush 到 FTL（简单点可以直接 flush 掉所有 dirty 的 node）

3）cache 需要记录所有的未完成的 trim 信息，当 read 命令进来后，若 read 与 trim 有冲突，则需要将 read 命令 pending 住。

3. trim 命令交互：

trim 命令由 FE 生成 trimNode，其结构与 writeNode 一致：

```
typedef struct _FeReqTrim_t
{
   U08 u08Lun;//2
   U08 u08Fua;//3
   U08 u08Rev;//4
   IDXList_t SubReqList;//12
```

```
    U08 *pu08DataBuffer;//16
} FeReqTrim_t;
```

通过 SubNode，将具体的 trim 的 range 范围传递给 FTL。

```
typedef struct _FeSubReqNode_t
{
    U32 u32Lba;
    U32 u32SectCnt;//each sector is 4K
} FeSubReqNode_t;
```

trim 命令放入 FE2BE 的 write_queue 中，trim 命令完成后，放入 write_cmpl_queue 中，执行优先级与 write 命令一致。

二、 FTL 处理 TRIM 流程

FTL 处理 trim 命令的思路是将 trim 范围内的 mapping 进行更改，故 trim 命令需要触发 FTL 的 mapping merge 流程。具体处理过程如下：

1. 获取 trim 命令：

从 write_queue 中获取 trim 命令，即，在获取当前 trim 命令的时候，一定能保证 trim 命令之前的 write 命令都已经下发，若有之前的

write 命令 pending， 则按照当前的处理 write_queue 的逻辑，不会获取到后面的 tirm 命令的，保证了 trim 和 write 命令的先后关系。

需要保证在处理 trim 命令之前，已经处理完了所有 FE2FTL 的 read_queue 中的命令，故 FTL 在进行调度时，发现当前命令为 trim 时，需要

先处理掉所有的 read 命令。

若出现当前 trim 命令时出现 pending， 也不能处理后续的 write 命令。

2. 处理 trim 命令：

1). 生成一条 trim 命令的 record，用于记录当前 trim 命令时刻， host 写入的数据的情况，record 结构：

```
typedef struct _TrimCkpt_t
{
    SpbId_t Spb;
```

```
    U32 Wptr;
    U32 SpbSn;
    U32 LuSn;
} TrimCkpt_t;
```

对于每一个命令， FTL 需要记录当前的 HOST 写入的 ASPB 的 SpbId，SN 以及 write_ptr，记录当前 LU 的 SN。

该记录用于 trim 与读写数据冲突时，比较新老数据的依据。

当 merge mapping 或者 read 的时候，已知 LAA 命中在一个 trim 命令的范围内，则比较当前 mapping 中的 PAA 与 trim 的新老，

具体的比较方式为： 当前 PAA 的 SpbSn 与 TrimCkpt_t 中的 SpbSn 相比，谁更小则谁为老数据；若 SpbSn 相同，则比较 PAA 对应

的 PageOff 与 TrimCkpt_t 中的 Wptr, 谁更小谁为老数据。

2). 将 trim 命令入 trim_proc_queue：

FTL 将所有收到的 trim 命令入 trim_proc_queue, 入队完成后，即可返回，让 FTL 从 FE2FTL 的 queue 中继续处理其他的 CMD。

同时，FTL 启动处理 trim queue 的任务。

入队时，根据当前 trim 命令的大小，将小范围的命令放入入 high_queue ，大范围的命令放入 low_queue。

3). 处理 trim_proc_queue:

优先从 high_queue 中获取 trim 命令进行处理，若 high_queue 为空，则从 low_queue 中获取。

支持当正在处理一个 low_queue 中的 trim 命令时，有高优先级的 trim 命令入队。则可以打断正在执行的低优先级的 trim 命令，先执行

高优先级的命令；高优先级命令处理完成后，若没有新的高优先级命令，则返回之前打断的低优先级命令任务继续执行。

4). trim 超时处理：

从 trim 命令入队开始，开始记录 trim 命令处理时间， 当 trim 命令在设定的阈值时间内没有完成，则需要提前返回 trim 命令至 FE。同时

该 trim 信息需要持久化到 NAND 上进行保存，防止在返回 FE 后，发生了掉电，导致 trim 信息丢失。（保存到 NAND 上的命令存在一个上限，

即只支持设定阈值的 trim 命令支持超时返回。若已经达到给阈值， 则 trim 命令需要等待处理完成后，才返回 FE）

trim 的持久化信息如下：

```
typedef struct _TrimInfo_t
{
    TrimCkpt_t ckpt;
    U16 LaaCnt;
    U16 RangeCnt;
    u64 lba[MAX_RANGE_PER_TRIM];
    u32 count[MAX_RANGE_PER_TRIM];
} TrimInfo_t;
```

trim 的持久化信息存储在 sysLu 中，位于 SLUT 之后。

、5). trim merge mapping:

FTL 处理 trim 命令的主流程。 需要将 trim 命令的 range 中包含的 mapping 修改为 UNMAP_PAA，用以标志该映射项被 trim 掉了。FTL 中

存放最新的 mapping 的地方为 RLUT，则需要启动 merge 流程将 RLUT 和 trim 一起进行处理，否则若单独处理 trim 的 merge， 当 trim 命令

完成后，无法获知 RLUT 中的记录与 trim 命令的先后顺序，造成数据不一致。

基于现有的 RLUT 的 merge 流程（即 convert 流程），增加一种触发 merge 的条件，即 trim 命令触发 convert。

需要修改当前的 convert 流程实现，细节如下：

a. convert 运行后，check 下是否有需要处理的 trim range 信息。若有，则优先根据 trim range 中包含的 L2PP 进行加载， 然后再根据 RLUT

当中的 LAA 进行 L2PP 加载。

b. 对加载起来的 L2PP 进行 merge 操作时，先 merge RLUT 中的 mapping，再 merge trim range。 merge trim range 时，需要根据 trim 的 ckpt_info

来 16 进行判断 mapping 的新旧。

c. 在 convert 流程完成时，需要将当前处理的 trim 的信息记录到 UserLuDesc 中，记录的信息结构体如下：

```
typedef struct _TrimFence_t
{
    U32 Sn;
    u32 RangeIdx;
    LbaRange_t Range;
} TrimFence_t;
```

Sn 为当前 merge 的 trim 的 LuSn，用于上电重建过程中判断是否是已经完成的 trim 命令。

RangeIdx 为当前 trim 命令的哪个 range；

Range 为当前正常处理的 tirm range 范围。

需要修改当前的 RLUT 流程实现，细节如下：

a. 需要支持 RLUT 在没有 FULL 的情况下进行 merge 操作，依然采用先将 RLUT flush 后再进行 merge，只是在 merge

完成后，当前 RLUT 不进行回收，继续作为 BUSY 的 RLUT 进行使用。ASPB 上需要记录当前的 RLUT 更新的位置（updatePtr）。

b. RLUT 在 merge 的过程中支持插入新的 mapping。同时， convert 需要记录一个 boundary，在 merge RLUT 的时候，超过

boundary 的 mapping 不进行 merge 处理。

c. RLUT 支持进行多次 merge 操作，以当前 ASPB 的 updatePtr 作为起点，以当前的 RLUT 的 fillIndex 作为终点进行 merge

操作。

d. 若发现 updataPtr 与当前 RLUT 的 fillIndex 相等，则当 trim 触发 convert 时，无需进行 RLUT 的 flush 和 merge 操作。

6） trim info 重建

当 FTL 重新上电时，需要对 TRIM 的持久化信息进行重建，已完成之前掉电后为成功执行的 trim 处理，重建过程如下：

a. 读取存储在 sysLu 中的 tirm info；

b. check 每笔 trim info 是否需要继续处理：

因为该笔 trim 命令可能已经处理完成，但是持久化信息还在。故需要将 trim info 中记录的 luSn 与 UserLuDesc 中的记录的 trim

的 SN 进行比较，若小于后者，则不需要恢复；若相等，说明是真正执行的这一笔 trim 信息，则比较 range 范围来确定恢复的 range；

若大于后者，说明是还未执行的 trim 命令，需要恢复。

c. 将恢复出的 trim info 重新生成 trim 命令，进入 trim 的 queue 中进行后续的处理。

7）trim 与读写命令交互

a. trim 与 write：

不需要特殊处理，trim 命令和 write 命令位于同一个 FE2FTL queue 中，按照顺序处理命令即可；

b. trim 与 read：

当 trim 命令没有达到 FTL 的时候，需要 FE pending read 命令；

当 trim 命令达到 FTL 后，FTL 通过 trim queue，记录了 trim 的信息，read 命令到达 FTL 后，需要比较

mapping 和 trim 的新旧，从而返回正确的 read 数据。

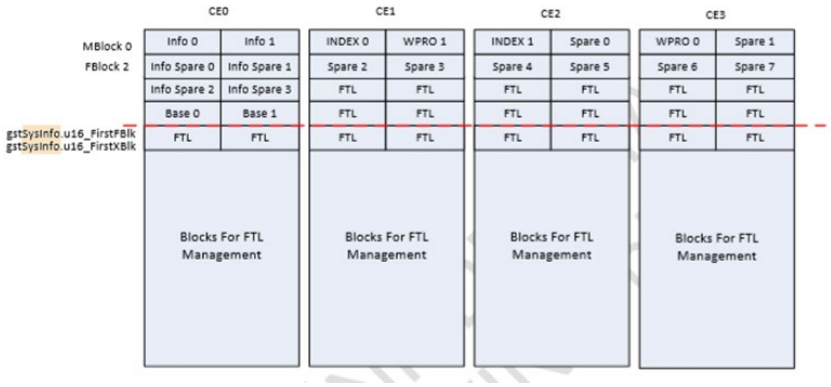
，

空间管理

1. 物理空间管理：

使用 pool 管理不同物理空间，NAND 提供的空间按照 FTL 使用上的区别，分为：

1） ROOT_BLK_POOL:管理一些 ROOT meta 以及 specail 的 user data(boot LUN data， rpmb LUN data、health data 以及 FTL boot info 等数据)，该 POOL 以 BLOCK 作为粒度来分配空间。



（图中 index/wpro 取消，由 FTL 统一管理）

该 POOL 空间来自于前 N 个 NAND BLOCK,待 system info block 分配完成后，剩余的 BLOCK 就被 FTL 管理到 ROOT_BLK_POOL 中。具体空间划分为：

a) FTL_MGR_BLOCK:

存放 FTL 管理 boot block 的信息，控制 boot_blk_pool 中所有 block 的分配，是 FTL 的根节点。

b) FTL_META_BLOCK:

用于存放 FTL 的 ROOT meta 信息：

c) DEV_INFO_BLOCK:

用于存放 device 信息相关的 block，包含 health info 等设备相关的数据信息

d) EVLOG_BLOCK:

存放 evLog 的 block

e) WK_LUN_BLOCK:

存放 BOOT LUN 和 RPMB LUN data 的 block

f)　　OTHER_BLOCK:

冗余 BLOCK,用于上述 BLOCK 产生坏块时进行替换。

注：

除 evLog block 外，其他的各分区有双备份或者多备份的策略，并且各自具备一定数量的 spare block。

BLOCK 的分配方式：

ROOT_BLK_POOL 中维护一张 block 的分配表 root_blk_bitmap，将 rom 的 info block 标记为 1， 然后剩余的为 0 即可分配 bit。其中 FTL_MGR_BLOCK 从前完后申请，其余的 BLOCK 从后往前申请，申请出 BLOCK 后将相应的 bit 置位，root_blk_bitmap 需要持久化保存。

2）SLC_SUB_SPB_POOL

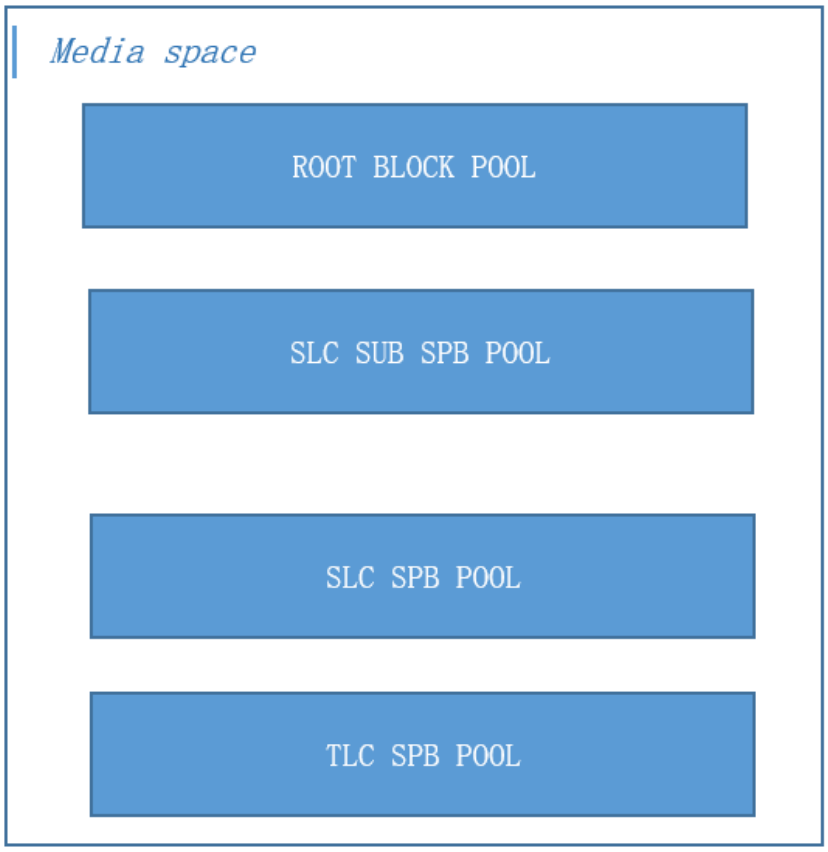
该 POOL 提供以 SPB 作为分配粒度的 SLC 空间，支持 SPB 的 BLOCK 数配置（可以为最大并发度的 N 分之一）。

3）SLC_SPB_POOL

该 POOL 提供以 SPB 作为分配粒度的 SLC 空间，SPB 长度为最大并发度，不可以配置。

4） TLC_SPB_POOL

该 POOL 提供以 SPB 作为分配粒度的 TLC 空间，SPB 长度为最大并发度，不可以配置。支持 dynamic SLC, 将满足条件的 TLC 临时作为 SLC 使用，支持 dynamic SLC 占比的配置。



2. 空间地址描述：

1） WeruId & SpbId:

WeruId: 用于标识物理 superBlock， 从 block0 开始，将整个 media space 顺序编码。

SpbId: FTL 使用的逻辑 superBlock 编号，spb 与 weru 存在一对一（普通 pool）和多对一(sub_slc_pool)的关系。

2) Lpda:

Lpda 用于描述 SPB 中的逻辑地址，在一个 SPB 中顺序编码。Lpda 的构成由

SpbId + offset 组成：

Lpda = spbId<< block_shift | offset;

同时，FTL 内部管理的所有 mapping 都是 lda 到 Lpda 的映射。

3) Wpda:

Wpda 即 wearLine pda，用于描述在物理 SPB 上的地址，是 FTL 与 BE 交互的地址信息。

Wpda = WeruId<< block_shift | offset;

即 Wpda 这里看不到 subSpb 的存在。

4）地址转换：

FTL 与 BE 交互时，需要完成 Lpda 到 Wpda 的地址转换。

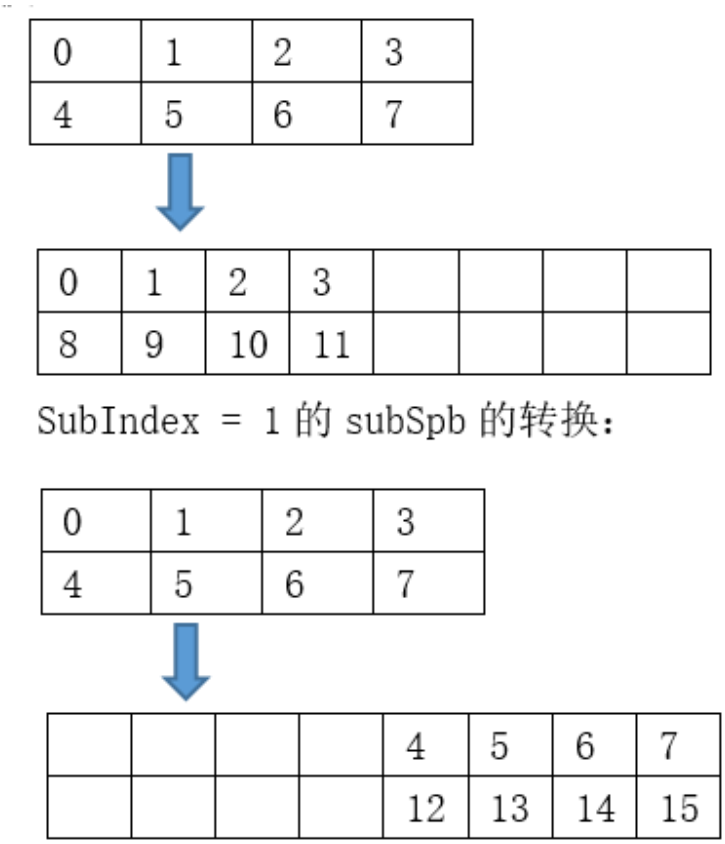
a. 对于非 SUB_SLC_POOL 的 slc spb:

这里的 spb 与 weru 的大小一致，只需要转换 BLOCKID 即可。

b. 对于 SUB_SLC_POOL 中的 slc spb:

需要根据其 subIndex 来看是位于物理 spb 上的哪一段，再结合 subSpb 上的 offset 来进行转换。

举例说明：假如物理 spb 分为 2 个 subSpb，那么 subIndex = 0 的 subSpb 上的转换：



c. 对于 TLC_SPB_POOL 中的 spb:

需要根据不同的 nand 的 program order 来决定转换逻辑：（已一个 4plane, 2 die 的 tlc 为例）

Die0:

0	1	2	3
4	5	6	7
8	9	10	11

0	1	2	3				
8	9	10	11				
16	17	18	19				

Die1:

12	13	14	15
16	17	18	19
20	21	22	23

				4	5	6	7
				12	13	14	15
				20	21	22	23

3. 逻辑 LUN 空间管理

在物理空间之上，基于不同的逻辑需求，划分出几个逻辑 LUN：

1）LUN 类型：

a. USER_LUN

提供用户空间的 LUN，用于存储用户数据,最小粒度为 4k。

b. MIDDLE_LUN

存放 USER_LUN meta 信息的 LUN,主要包括 USER_LUN 的 12p, p21，最小粒度为 4k。

c. SYS_LUN

存放 MIDDLE_LUN 的 meta 的 LUN，主要是 MIDDLE_LUN 的 l2p，最小粒度为 4k。

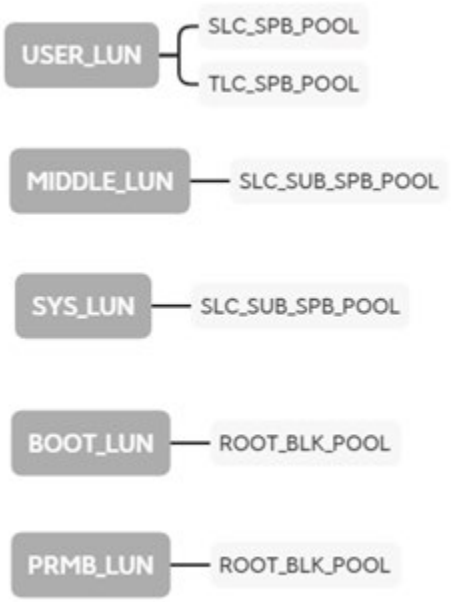
d. WK_LUN

存放 BOOT_LUN 和 PRMB LUN 的 data。

（middle_lun 不是必须的，当空间比较小的时候，可以不需要 middle_lun）

2） POOL 空间分配：

不同的 LUN 可以根据自身需求从不同的物理 pool 中获取空间，当前的分配策略为：



a) force_slc 模式：

该模式下，user data 首先需要写入 SLC 中，若 SLC 空间不足，需要 gc SLC 到 TLC 后继续使用 SLC。

在该模式下，又可以配置是否支持 dynamic SLC。配置了 dynamic SLC 后，TLC 中的部分或者全部的 SPB 在满足一定条件下可以转为 SLC 使用，在初始写入时，转换比为 100%；后续转换比为 dynamic SLC 的配置数据。

b) normal_slc 模式：

该模式下，优先使用 SLC_CACHE 即 SLC_SPB_POOL 中的 SPB, 当 SLC_CACHE 中的 SPB 耗尽后，使用 TLC direct write。同样，可以配置 dynamic SLC。同 a 中的方式一致。(初次写都要为 SLC)

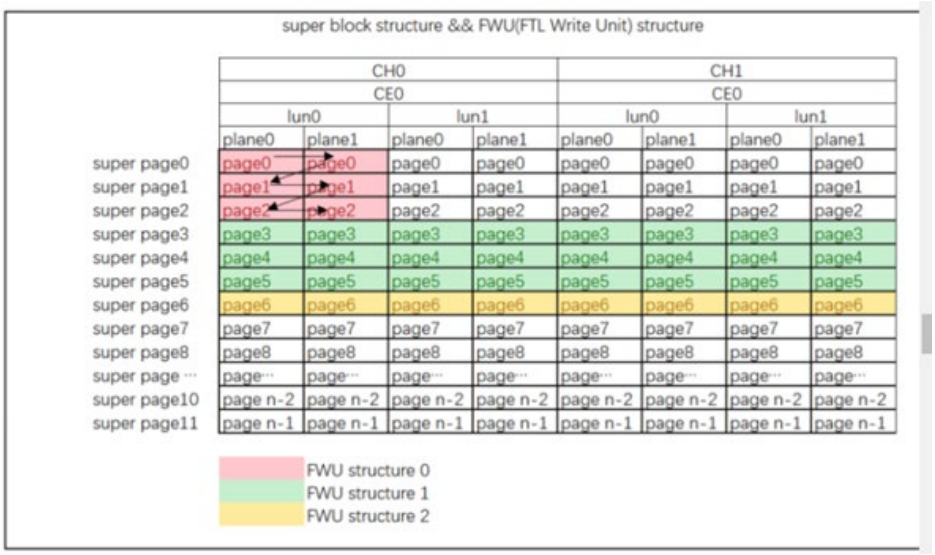
MIDDLE_LUN 以及 SYS_LUN 使用 SLC_SUB_SPB_POOL 中的 SPB,根据具体的 NAND 配置 SUB SPB 的大小。

BOOT_LUN&RPMB_LUN 使用 ROOT_BLK_POOL 中的 BLOCK 空间，该 POOL 中的空间采用预留的机制，不需要动态申请和释放。

3） SPB 内的空间分配

a. USER_LUN

使用 FWU（FTL write unit）来封装不同的 program 要求。一个 FWU 会记录对应 program 空间的起始位置，结束位置与大小，也就是一堆 PDA 的集合，对外呈现一段连续的逻辑空间。其它业务在向一个 FWU 写数据时，完全可以把它当成一段连续的数据 buffer，而不需要考虑最终数据的 Layout。真正决定怎么去放一个 FWU 数据由专门的逻辑根据不同的 NAND 以及当前 SPB 的坏块情况等因素来确定其具体大小。



b. MIDDLE_LUN & SYS_LUN

这两个 LUN 上按照 page 对齐的规则写入（实际下刷 meta 时也会尽力按照 muti-plane 方式，但受下刷数据量影响，不做强制 mutil-plane）。

c. BOOT_LUN & RPMB_LUN

这两个 LUN 基于 BLOCK 管理，按照 page 写入。

4. SPB 规划:

对 spb 按照 EC 值进行排序，偏大的优先考虑设置为 SLC，小的为 TLC。对都使用 SLC 的 LUN， 按照 EC 由小到大，分别分配给 SLC_SUB_SPB_POOL 和 SLC_SPB_POOL；ROOT_BLOCK_POOL 固定分配前几个 SPB。

1） 预留出 ROOT_BLOCK_POOL 中的 BLOCK：根据需求，预留出 3-4 个 SPB 的 BLOCK 数，分配到 ROOT_BLK_POOL 中。

2） 根据设定的 user_data 数据量大小，计算出 mapping 的大小（l2p 和 p2l），得到 middle_lun 需要的空间和 sys_lun 需要的空间，然后根据各自的 OP 需求，得到所需的 SPB 个数分配到 SLC_SUB_SPB_POOL 中。

3）根据设置的配置参数[A1] 和 SLC 模式，得到 user_lun 中 slc_cache 的大小，得到 SLC_SPB_POOL 的 SPB 个数；最后剩余的 SPB 分配到 TLC_SPB_POOL 中。此时需要 check TLC_SPB_POOL 中的有效空间（除去坏块）是否满足 user_data 的容 量以及 OP， 若不满足则会导致 pool 的初始化失败，尝试调整各 LUN 的 op 后重新进行初始化，若仍然不满足，则开卡失败。

5. 坏块管理：

FTL 对于坏块的处理，支持初始替换和动态替换，同时在无法替换后，支持带坏块继续使用。

1） 坏块替换：

a. 替换范围： 除 ROOT_BLK_POOL 以外的其他 POOL 的 SPB。

b. 替换限制： 达到替换设置的最大替换次数后，后续坏块不再替换。

c. 替换时机： 在初始开卡时执行初始替换，后续新增坏块时执行动态替换。

d. 替换逻辑： 只执行同 plane 替换

初始替换：按照 SPB 坏块个数从小到大进行排序，从头尾开始进行替换，即选用尾部上的 SPB 上的 BLOCK 去替换头上的 SPB， 替换次数达到最大替换次数，或者替换到无法再执行时退出。

动态替换：在未达到最大替换次数时执行动态替换。不预留专门提供替换而存在的 SPB， 当产生新增坏块时，从 TLC_SPB_POOL 中获取一个 free 的包含坏块且能够提供替换块的 SPB 作为坏块提供方，执行替换后，坏块提供方的 SPB 在坏块总数未达到不能使用的阈值时，可以继续供 TLC_SPB_POOL 使用。

e. 设置最大可用坏块个数阈值 ：当某个 SPB 的坏块个数达到这一阈值时，整个 SPB 将不再使用。

f.替换信息存储：预留最大替换次数个 node 内存，将该信息固化到 BOOT_BLK_POOL 中 ftl root block 当中进行存储和恢复。（用 spb arry 方式）

g.坏块信息表： FTL 中只保存一份坏块信息表，即替换后的坏块信息表，不记录原始坏块信息表。

2） 使用坏块 SPB：

当替换不能再执行时，SPB 中就会包含有坏块。FTL 支持包含坏块的 SPB 继续使用。在写 IO path， GC ，P2L 表以及重建等与 SPB 空间布局相关的操作都需要考虑坏块的存在。

3）大量坏块的处理：

当产生大量无法替换的坏块后，需要 check FTL 是否需要进入 read_only 状态：SYS_LUN & MIDDLE_LUN & USER_LUN 的空间是否已经足够。

6. GC 策略：

每个 POOL 各自有自己的 GC 策略：

1） ROOT_BLK_POOL: 由于基于 BLOCK 管理，没有 GC 逻辑，而是使用 spare BLOCK 的方式进行 BLOCK 的乒乓使用。

2） SLC_SUB_SPB_POOL:

按照 free spb 设定 5 个水位:block/gc_start/gc_end/bgc_start/bgc_end。其中 block 为完全 pending 上层 IO; gc_end >= gc_start + 1; bgc_end >= bgc_start + 1; bgc_start > gc_end。

3） SLC_SPB_POOL：

同 2）中设定 5 个水位。同时根据不同是 slc cache 模式设定不同的值。

a. force_slc：

5 个阈值为：0/2/3/MAX/MAX (MAX 是指全部的 BLOCK)

b. force_slc + dynamic_slc:

该模式下 SLC_SPB_POOL 将 GC 的控制权交由 TLC_SPB_POOL 执行。

c. normal_slc: (direct tlc mode)

5 个阈值为：NA/NA/NA/MAX/MAX (NA 是指无效水位，即该阈值不发生作用)

d. normal_slc + dynamic_slc:

该模式下 SLC_SPB_POOL 将 GC 的控制权交由 TLC_SPB_POOL 执行。

4） TLC_SPB_POOL:

同 2）中设定 5 个水位。

a. Force_slc + dynamic:

该模式下，根据 slc_cache + dynamic slc 中 free 的量来启动 slc GC:

5 个阈值： 0/2/3/MAX/MAX

同时，还需要根据 free 的 tlc spb 个数设定 TLC 的 gc 水位：

5 个阈值： 3/5/7/8/10 (暂定)

b. Normal_slc+ dynamic:

slc gc：NA/NA/NA/MAX/MAX (包含 slc_cache 和 dynamic slc)

tlc gc: 3/5/7/8/10 (暂定)

slc gc 优先级高于 tlc。

c. 非 dynamic:

只有 tlc gc: 3/5/7/8/10　　(暂定)

元数据管理

1. 主要元数据类型

1） MAPPING

a. L2P:

逻辑地址到物理地址的映射，存在于每个 FTL LUN 中:

映射粒度为 4k（BOOT_BLK_POOL 中的 LUN 除外）

使用 L2PP_ID 描述一段 mapping, 长度为 1024 个 mapping，大小为 4k，4M 的 user_data 产生 middle_lun 的 4k 的数据写入。

b. P2L:

物理地址到逻辑地址的映射，目前只存在 USER_LUN 中。其作用为：

当有 UMT 存在时，P2L 只是用于加速重建和 GC；当没有 UMT 存在时，P2L 不仅要用于加速重建和 GC，还要作为 mapping 的 changeLog。

c. UMT:

L2p 的 changeLog，保存最新的 l2p 映射，只存在 USER_LUN 中。UMT 作为可选项，主要用于优化 IO 上啊对最新 mapping 的查询和写入，有额外的内存开销。

2） SPB_DESC_INFO

SPB 的描述信息，包含 SPB 的归属 pool, valid_cnt，Ec_cnt, flags 等状态信息。

3） BLOCK_DESC_INFO

ROOT_BLK_POOL 中描述 BLOCK 的信息，包含 ec_cnt 等信息。

4） LUN_DESC_INFO

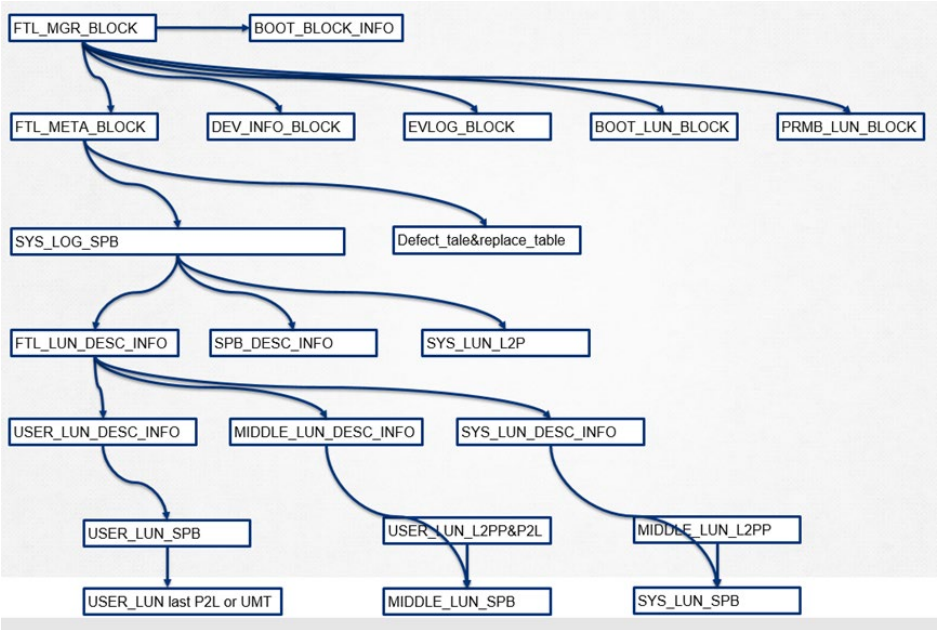
描述每个 LUN 的信息，包含 active SPB 的描述信息, sn, flags 等。

5） defect_bitmap & replace_table & 新增坏块

坏块表信息以及替换表信息；

新增坏块为发生了坏块的 SPB 的 BLOCK 信息。

2. 元数据布局



1） FTL_MGR_BLOCK:

该 BLOCK 为 FTL 所有 meta 的根，位于 ROOT_BLK_POOL 中。该 BLOCK 中的结构体定义：

```
Typedef struct _MGR_BLK_INFO

{

   MGR_BLK_HEADER   header;

   BLK_INFO                ftlMgrBlock;

   BLK_INFO               ftlMetaBlock;

    BLK_INFO                DevInfoBlock;

   BLK_INFO                EvLogBlock;

   BLK_INFO                WkLunBlock;

   U32                          BlockBitmap[];

BOOT_BLK_DESC_INFO   descInfo;
```

```
} MGR_BLK_INFO;

Typedef struct _MGR_BLK_HEADER

{

   U32                        signature;

   U32                          crc;

   U32                          flushId;

   U32                          dataLength;

} MGR_BLK_HEADER;

Typedef struct _BLK_INFO

{

   PDA_T                 writeBlock;

   PDA_T                 mirrorBlock[];

   PDA_T                 spareBlock[];

   U8                       mirrorCnt;

   U8                       spareCnt;

} BLK_INFO;
```

descInfo 中主要记录每个 BLOCK 的一些基本信息，比如 EC 次数等。

各 block 的镜像策略根据每个 BLOCK 自身的需求来确定镜像个数已经 spare 的大小。

当某个 BLOCK 的在读写时发生错误时，可以从 block_bitmap 中重新分配一个新的 block 来进行替换。

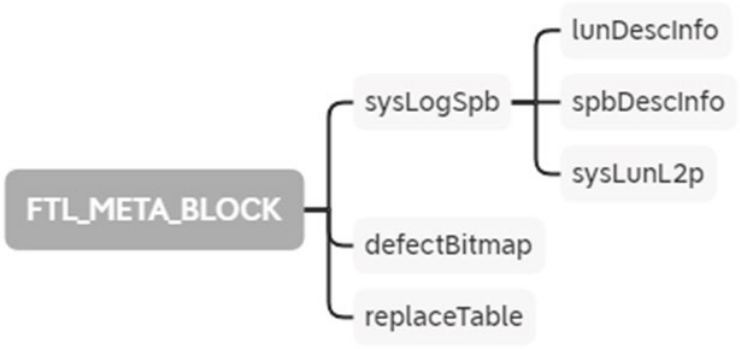
一个 writeBlockk 对应一个或多个 spareBlock。

针对 FTL_MGR_BLOCK, 采用 5 镜像 2 spare 策略，故占用 BLOCK 数：1+5+2 = 8。

2） FTL_META_BLOCK:

存放 FTL 模块根信息的 BLOCK。

该 BLOCK 中存放的信息为：



其中， sysLogSpb 中存放的是一些变化比较频繁的 FTL meta； defectBitmap 为 SPB 的坏块表；replaceTable 为 SPB 的坏块替换表。

针对 FTL_MGR_BLOCK, 采用 3 镜像 2 spare 策略，故占用 BLOCK 数：1+3+2 = 6。

3） DEV_INFO_BLOCK

主要用于存储 device 相关详细的 block:比如 smart 信息等。 采用 3 镜像 2 spare 策略，故占用 BLOCK 数：1+3+2 = 6。

4） WK_LUN_BLOCK

用于存储 well konow LUN 的 data 和描述符信息，以及 data 对应的 mapping 信息也保存到该 BLOCK 上。采用 3 镜像 2 spare 策略，故占用 BLOCK 数：1+3+2 = 6。（按容量）

5） EVLOG_BLOCK

存放 log 的 block，后续考虑可以放到其他的 BLOCK 上去。不采用镜像策略，分配 4 个 block 用于记录 evlog。

6） SYS_LOG

使用 SPB 用于存储 FTL 中的一些频繁变化的数据量相对比较大（与 ROOT_BLK_POOL 比较）的元数据，主要包含：

1. Lun_desc_info: sys_lun， middle_lun 和 user_lun 的描述信息，主要描述信息如下：

```
Typedef struct _LUN_DESC_INFO

{

   U32                  sn;

   U32                  flags;

   ASPB_DESC_T     aspbDesc[];

} LUN_DESC_INFO;

Typedef struct _ASPB_DESC_INFO

{

SpbId_t   spbId;

U32           writePtr;

U32           updatePtr;
} ASPB_DESC_INFO;
```

Sn 记录当前 LUN 的写 sn 号， 所有 LUN 的 sn 统一编码。

Flags 记录当前 LUN 是否是 clean 状态。

AspbDesc 描述当前 LUN 正在使用的 SPB 的信息，同时 ASPB 也是 LUN 重建的对象。其中 writePtr 描述写 page 分配的位置，updatePtr 描述 mapping 更新完成的位置。

2. Spb_desc_info:

描述 SPB 的状态信息：

```
Typedef struct _SPB_DESC_INFO
```

```
{
U32       flags:10;
   U32       validCnt:22;
   U32       readCnt:28;
   U32       subSpb:1;
   U32       spbType:1
   U32       poolId:2
   U32       ecCount;
   ……      ……
} SPB_DESC_INFO;
```

3. SYS_LUN 的 l2p：

即 sys_lun 的 mapping 表，该表常驻内存，需要全量存储到 nand。

7） USER_LUN mapping & MIDDLE_LUN mapping

USER_LUN 的 mapping 包含 P2L 和 L2P， 都作为 data 存储在 MIDDLE_LUN 中；而 MIDDLE_LUN 的 mapping 包含 L2P，作为 data 存储在 SYS_LUN 中。

4. USER_LUN P2L:

该表在没有 enable UMT 时，发挥两个作用：一是用于重建 ASPB 和 GC 时，查找数据是否有效；二是作为最新的 mapping changeLog，在 IO path 上提供最新的 mapping 查询。在 enable UMT 时，只有作用一。

P2L 的组织形式：

LDA0	LDA1	LDA2	LDA3	LDA4	LDA5	LDA6	….	….	LDAn

即在一段连续的 buffer 内记录 LDA 的信息， 而 pda 则是 buffer 数组的 index。Buffer 的长度为 16K，一个 ASPB 至少需要使用一个 16K（所有的

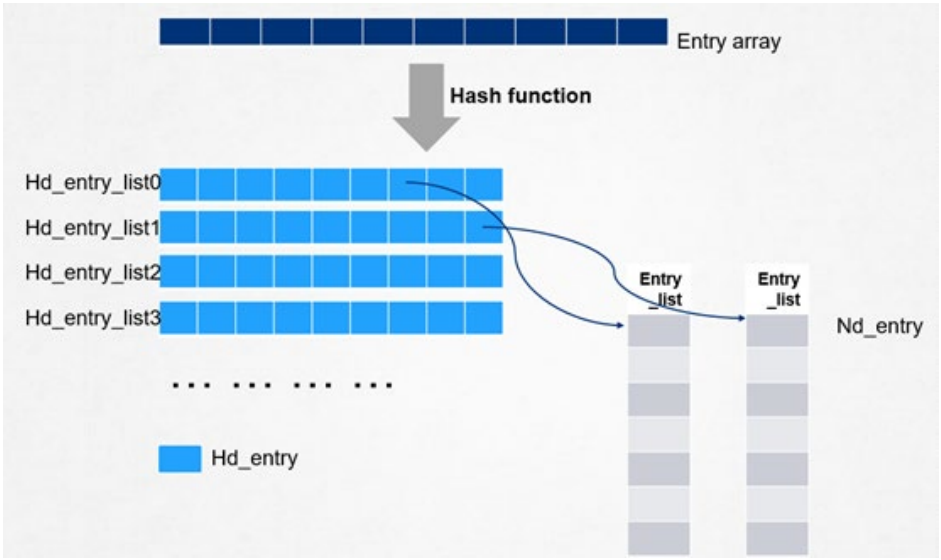
ASPB 额外需要一个或两个 16K 作为冗余）

b. UMT

UMT 为 L2P 的一部分，为当前内存中最新的 L2P。采用 hash 链表结构来进行管理：

Node 的结构为

链结构:



UMT 作为可选项，视当前系统的 memory 多少来开启或者关闭。若用 UMT，则 UMT 代替 P2L 充当 L2P 的 change log，提供 IO path 的增删查改。

c. L2P:

L2P 在内存中也需要进行管理（L2PP），组织为一段连续的 buffer:

PDA0	PDA1	PDA2	PDA3	PDA4	PDA5	PDA6	…	…	PDAn

,而在 LDA 上采用 l2pp_id + offset 的方式，其中 l2pp_id 为 L2P 在整个 LUN 的逻辑空间上的编码。例如：0~4M 为 l2pp 0， 4M~8M 为 l2pp 1。

3. 元数据操作

1） P2L:

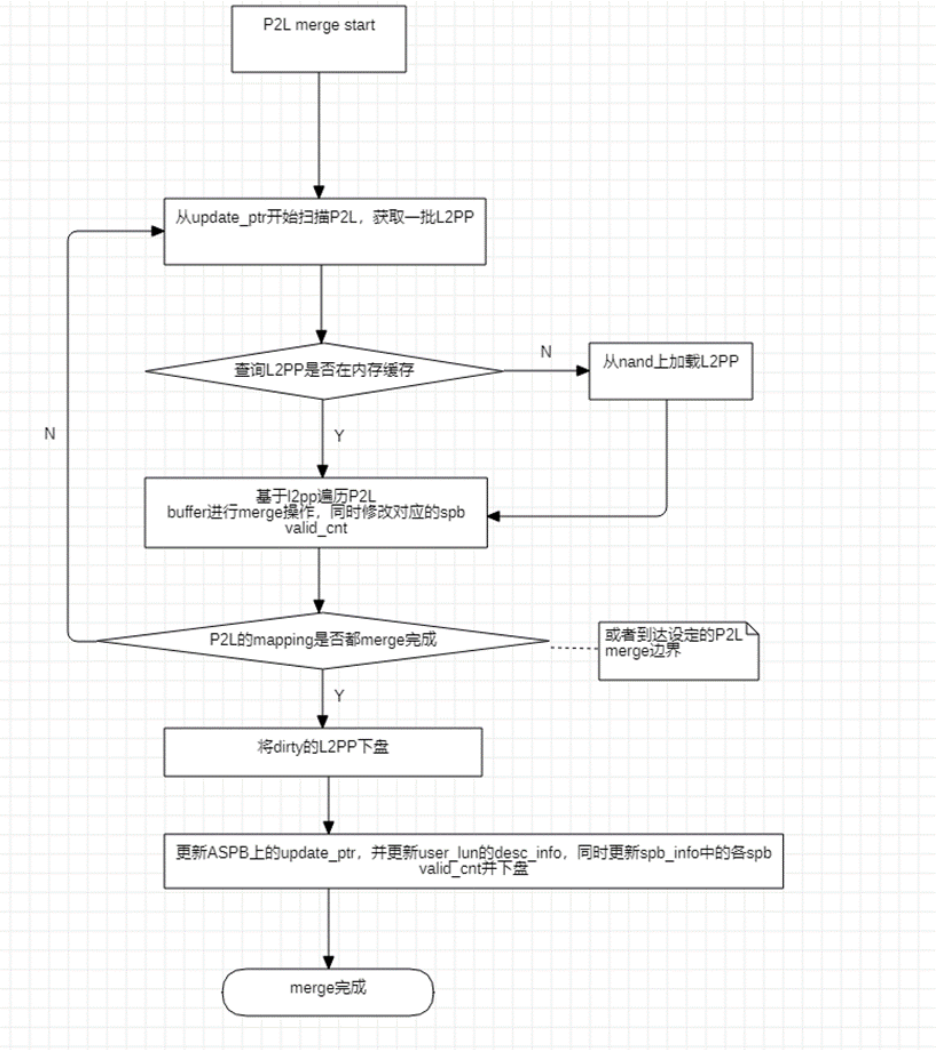
a. 写入：user write 以及 GC write 会将 mapping 写入到 P2L 中。host write 在写入时，需要 check user 的 P2L[A1] 中是否已经存在相同的 LDA。若有，需要将该 LDA 设置为无效 LDA；GC write 在 写入时，需要在 P2L 表上记录 GC 的源 spb_id 信息，用于后续的 merge 操作。

b. 下盘： 当 P2L buffer 写满后，会将 P2L flush 到 nand 保存。

c. Merge: 将 P2L 中的 mapping 更新到 L2P 中的行为

[A1]若搜索引擎支持倒序搜索，则可以采用最佳写入的方式，不需要 modify 之前的 LDA

具体操作流程为:



Merge 的时机：

当 P2L 表写满的时候，只触发当前的 P2L 表 merge；

有 trim update mapping 的时候，需要触发 host P2L 表 merge;

Flush 等流程触发的 force merge。

d. 释放：

当一个 P2L 在即完成了刷盘，又完成了 merge 后，才能将内存进行释放。

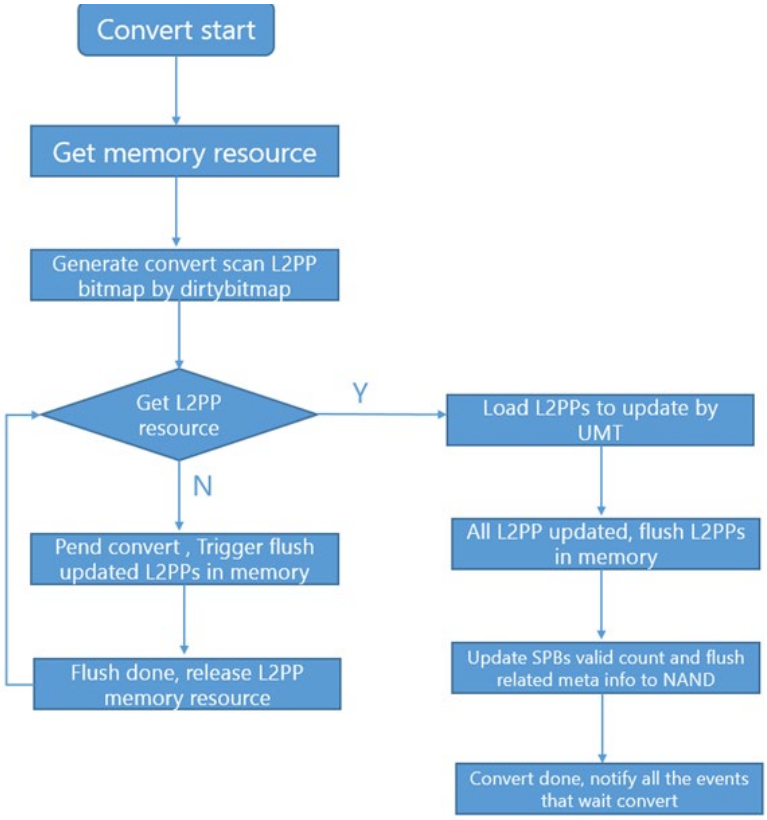
2） UMT

若存在 UMT 时，UMT 作为最新 mapping 的 change log：

a. 写入：

Host write 和 GC write 完成时，往 UMT 中写入最新 mapping。其中，GC write 写入前需要先 check UMT 中是否已经有相同的 LDA 存在，若有则该位置不写入。

b. Merge: 将 UMT 中的 mapping 更新到 L2P 中的操作。



UMT 不区分 mapping 是哪个 ASPB 的，都会一起 update。

Merge 的时机：

当 UMT 资源不足时；

有 trim update mapping 时;

Flush 等流程触发的 force merge。

c. 释放： 在 merge 进行的过程中，已经 merge 完成的那部分 entry 就可以进行释放。

3） LUN_DESC_INFO

a. 当 LUN 分配新的 ASPB 时，需要将新分配的 SPB 信息更新到 LUN_DESC_INFO 中并下盘。

b. 当 ASPB close 时，需要将 ASPB 的信息从 LUN_DESC_INFO 中移除并下盘。

c. 触发 mapping merge 时，需要更新 LUN_DESC_INFO 中的 ASPB 信息，并下盘。

d. 重建 LUN 时，需要将重建后的信息更新到 LUN_DESC_INFO，并下盘。

4) SPB_DESC_INFO

该描述性中包含多个属性字段，不同的流程会针对性的修改不同的属性字段，但不是所有的修改都需要马上下盘，需要视具体流程和修改的属性而定。

5) LUN 状态

Clean: 指与 LUN 相关的所有 meta 都完成下盘并处于一致状态。

比如 USER_LUN, 当其 P2L 全部完成 merge， 并下盘；L2PP 完成下盘；LUN_DESC_INFO 以及 SPB_DESC_INFO 也完成了更新并下盘后，该 LUN 就处于 clean 状态。此时若下电，再上电后就不需要重建任何数据。

Dirty: 指该 LUN 上还有一些 meta 存在脏数据，没有完成下盘。当上电后发现 LUN 是 dirty 的，就需要执行重建操作。

Clean->dirty: LUN 收到写 io；

Dirty->clean: LUN 执行 flush 流程，meta 完成刷盘。

6）SYS_LOG

SPB 空间来源于 SLC_SUB_SPB_POOL 中，动态申请。当 LUN_DESC_INFO|SPB_DESC_INFO|SYS_LUN L2P table 更新数据时，写入 SYS_LOG 的 SPB 当中。

SYS_LOG 根据写入的类型，管理一个 mapping_table。

当 SPB 快满时，触发分配一个新的 SPB；当 SPB 满时，触发切换 SPB 的动作，需要将旧的 SPB 上最新的数据 copy 到新的 spb 当中，然后释放旧的 SPB。同时，将新的 SPB 信息存入 FTL_META_BLOCK 当中。

SYS_LOG 的写都会触发一次镜像写以保证数据的可靠性。

7）Defect_bitmap & replace_table

a. 在开卡阶段，会生成初始扫描的 defect_bitmap 和初始替换的 replace_table， 将其存放在 FTL_META_BLOCK 当中；

b. 在正常运行阶段，读写 IO 失败导致产生新增坏块时，记录新增坏块信息到 FTL_META_BLOCK 上， 等待后台任务来进行 diagnose 和标记坏块。

8）ROOT_BLK_POOL

该 POOL 当中的 block 都预留有 spare block，当前 block 写满后，触发切换 block 的操作，将最新的数据 copy 到新的 block 当中去，然后修改为当前 BLOCK，之前的 BLOCK 修改为 spare block。

4. 元数据重建

在 FW 重建上电时，FTL 需要进行元数据重建操作：

1） 重建 FTL_MGR_BLOCK：

通过扫描指定位置的 ROOT_BLK_POOL 中的 block，找到最新的 FTL_MGR_BLOCK 信息所在的 BLOCK，恢复出其他 META 的 block root 信息。

2） 重建 WK_LUN_BLOCK:

为了尽快相应 host，该 BLOCK 的重建优先级仅次于 1）。通过扫描 WK_LUN_BLOCK 的 data,恢复出 WK_LUN 的 mapping_table 以支持 host 对齐的读写操作。

3） 重建 FTL_META_BLOCK:

通过扫描 FTL_META_BLOCK 所在的几个 block，找到最新的 page 恢复出 ftl_meta，包括 defect_bitmap, replace_table 和新增坏块信息。

4） 重建 SYS_LOG_SPB

通过 FTL_META_BLOCK 的信息可以获取到 sys_log_spb 的 spb_id；

如果有多个 spb 都存在有效数据，先找到最新数据的 spb;

扫描 spb 找到最新写入的 table 的位置；

基于 table 扫描 table 后续新写入的 page,恢复出最新的 table。

5） 重建 SYS_LUN

从 sys_log_spb 中获取到当前 SYS_LUN 的 LUN_DESC_INFO， 若 flag 标记为 clean 状态，则不需要重建；否则需要进行重建：

a. 扫描 SYS_LUN aspb，从 aspb 的 update_ptr 往后扫描，恢复出最新的 mapping，同时在 erase_page 后补充些一定数据量的 pad 数据，更新 aspb 的 wptr 信息。

b. 根据 mapping， 修正 SYS_LUN 中的 spb 的 valid_cnt 信息。

c. 将新的 l2p table 下盘，更新 update_ptr 后，将 LUN_DESC_INFO 下盘。

6） 重建 MIDDLE_LUN

从 sys_log_spb 中获取到当前 MIDDLE_LUN 的 LUN_DESC_INFO， 若 flag 标记为 clean 状态，则不需要重建；否则需要进行重建：

a. 扫描 MIDDLE_LUN aspb，从 aspb 的 update_ptr 往后扫描，恢复出最新的 mapping，同时在 erase_page 后补充些一定数据量的 pad 数据，更新 aspb 的 wptr 信息。

b. 将恢复出来的 mapping(l2p)信息下盘，更新 aspb update_ptr。

c. 将 LUN_DESC_INFO 下盘。

7）重建 USER_LUN

从 sys_log_spb 中获取到当前 MIDDLE_LUN 的 LUN_DESC_INFO， 若 flag 标记为 clean 状态，则不需要重建；否则需要进行重建：

a. 分别扫描 HOST 和 GC 的 aspb，从 update_ptr 往后扫描，获取对应位置的 P2L 表，通过扫描出的 mapping，build P2L 表。若得到的 P2L 表 ptr 小于 update_ptr，则需要从 P2L 表的 ptr 往后扫描来恢复 P2L 表。（存在 P2L 表 merge 但是未下盘的 场景）

b. 同时 aspb 在扫描到 erase_page 后，需要继续补充一定数据量的 pad 数据，更新 aspb 的 wptr 信息。

c. 触发 P2L merge，并进行特殊标记，在该标记下，对 spb_valid count 会有特殊处理。

d. Merge 完成后，更新 aspb update_ptr，将 LUN_DESC_INFO 下盘。