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。