NVM与NAND Flash混合文件系统设计文档

# 项目背景及目标

## 背景介绍

随着IT及互联网技术的飞速发展，研究开发满足大容量、低成本、高可靠性、快速存取、低能耗等条件的新型存储技术，成为未来能构建大容量高性能存储系统的关键。

基于闪存的固态盘（Flash-based Solid State Disk， 简称flash SSD），是一种基于闪存芯片的新型半导体存储设备。由于固态盘内部没有机械部件，所以相比较磁盘来说性能有极大提升，固态盘可提供低延迟、低能耗、较少的噪音以及高可靠性。近年来固态盘已经被运用于计算机系统中，且预计在未来几年，固态盘将全面替代磁盘，成为二级存储的主要形式。相比较磁盘，使用基于NAND类型的闪存在随机读方面和写方面，整个系统的IOPS分别提高了85倍和49倍。

近年来，随着STT-RAM、PCM等非易失存储器（Non-Volalite Memory, NVM）技术的飞快发展，STT-RAM、PCM等存储器在读写延迟、寿命等方面越来越接近DRAM，同时在能耗、集成度、可靠性等方面的技术指标也在不断提升中。目前Intel公司已经发行了PCIe接口的Optane盘，Intel Optane技术结合了目前英特尔在存储研究上最为先进的硬件介质和软件方案，其中硬件介质3D XPoint是整个Optane技术的核心。按官方数据，3D XPoint的读写速度和寿命均为NAND闪存的1000倍，延迟是NAND闪存的千分之一，内存（DRAM）的10倍，而存储密度则是内存的10倍。虽然NVM的速度与DRAM相比还有一定差距，但它高性能的存取速度和稳定的读写特性，使得它能够成为下一代主流存储器件。目前NVM最大的问题是成本和容量。由于成本太高，对于普通的企业还不能将NVM代替磁盘和SSD作为主要的存储介质。

## 动机和意义

综合来看，NVM与flash SSD各有长处和不足。NVM有接近DRAM的读写性能，并具有可直接就地更新的特点；但同时，NVM还存在单个容量较小、价格较高等问题，不能满足大容量的需求。SSD优点在于容量大、无机械寻址、访问速度较磁盘更快等方面；但同时，固态盘不支持就地更新，随机写的性能很差，对于经常更新的数据，容易造成写放大。

在文件系统中，数据被分为普通数据和元数据。研究表明，50%以上对数据的访问集中于对元数据的访问，而元数据只是用来管理内容数据属性的数据，在介质中所占数据量很小，只需要很小的空间就能存储所有元数据；而普通数据也可按访存频率分为热数据、温数据、冷数据，冷数据即为那些不经常被访问的数据，而热数据表示那些常常被访问到的数据，温数据介于两者之间。因此，如何根据数据访问的不同特点并利用性价不一的设备来设计一个存储系统是目前研究的热点和难点。

由于NVM和SSD各有优缺点，为单个存储层设计的文件系统不能共同支持所有这些需求。因此，提出一种基于NVM和flash SSD的混合文件系统，利用一种存储器件的优势来弥补另一种器件的弱点。本方案针对F2FS文件系统进行改进，构建一个大容量、低成本、高可靠性、快速存取的文件系统。

## 设计目标

1、基于NVM和SSD两类存储设备的特性，实现高效的文件系统的组织和管理。通过分析文件系统中元数据和数据各自的特性，将元数据存储在NVM上，将数据存储在SSD上。

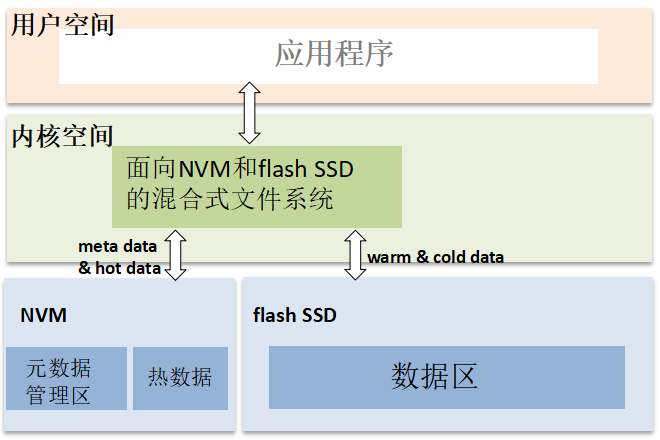
2、在文件系统语义上区分出数据的冷热程度，并将它们定向到不同的存储介质中。具体而言，将访存较不频繁的温数据和冷数据存放在SSD上，而访存较频繁的热数据则存放在NVM上。

3、实现一个迁移数据的模块，当NVM空间不足时，将NVM中相对较“冷”的数据块刷回到SSD中。

4、针对SSD和NVM两种介质，设计不同的垃圾回收算法。

# 总体设计

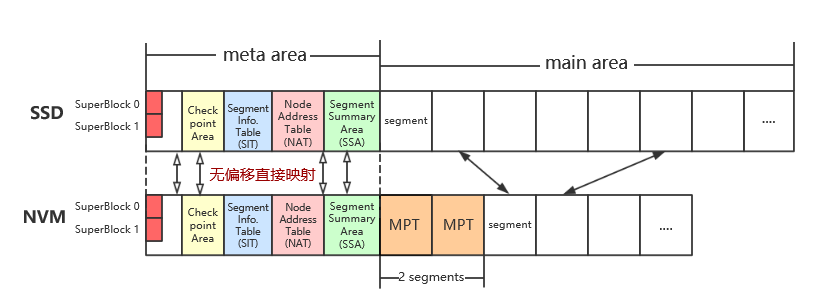
## 系统架构



**图1 基于NVM和SSD的混合文件系统架构图**

如图1所示，是该混合文件系统的架构图。混合文件系统将同时管理NVM和flash SSD，并根据其各自的特性，将元数据放在NVM上，剩余部分将存放普通热数据。而flash SSD将存储普通温数据和冷数据。

## 混合文件系统的布局分析



**图2 布局策略图**

如图2所示，是混合文件系统的布局策略图。其中SSD上的布局较F2FS相比并未更改，NVM则是本方案新加入的用于混合文件系统的加速盘。可以看到NVM上存放了混合文件系统的元数据区，新增了MPT区域，以及剩余部分用于存放该文件系统普通数据中的热数据。

由于SSD中的布局和F2FS相同，因此本文不再赘述，接下来详细分析NVM上的布局。

## NVM布局分析

**Meta区域：**可以看到，混合文件系统有一个“meta”区域，包含了各种不同的元数据（例如SB、CP、SIT、NAT、SSA），这一部分原本放在flash SSD上，并且在F2FS中，对“meta”区域并没有用log-structured流水线方式管理，而是采用了延迟刷回或双备份机制来就地更新的管理该区域。虽然元数据在文件系统中的数据量比较小，但是访问与更新都是很频繁的，由于Flash SSD物理特性的原因，对Flash的频繁写或者更新操作会造成很大的开销。如果把元数据存储到NVM中，可以大大减少元数据的更新开销，提高存储系统的性能。在本方案中加入了NVM加速盘后，“meta”区域的管理变得更加简单高效，在NVM设备初始化时将“meta”区域从SSD读取并转移到NVM上，此后对该区域的任何更改都直接定位到NVM上，由于NVM支持就地更新，因此可以使用少量的NVM来提高整体的相应时间；其次，这一设计也能降低对flash SSD的磨损，因为F2FS中对“meta”区域的更新采用就地更新的方式，而由于flash介质的写操作瓶颈，即它需要在更新前擦除以及其具有擦除次数的限制，因此把经常需要更新的“meta”区域定位到NVM上，可以减少对Flash SSD的磨损次数。

**MPT区域：**MPT是本方案在Main区域首部新增的区域，保存了SSD中segment和NVM中segment之间的映射信息。segment映射表可以看成一个数组，数组长度是SSD和NVM Main区域segment的总数，数组下标对应着SSD（NVM） segment号，数组中的值对应着NVM（SSD） segment号。

**NVM Main区域：**在F2FS中，使用6个活跃的segment管理来 cold/warm/hot 的node/data数据。然而热数据区域读写频繁，冷数据区域和温数据区域读写相对较少。因此，本方案将读写频繁的热数据区域放置在NVM设备，将冷数据和温数据放置在SSD设备，以发挥NVM高效的读写性能。同时，由于经常更新的热数据直接被定向到NVM，对Flash SSD的写入压力与磨损情况有很大的改善，能够增加整个混合外存的使用寿命。NVM Main区域与SSD中的管理方式类似，以segment为单位，当需要使用时，分配一个segment，映射了SSD中的热数据。

# 详细设计

## 新增主要结构

### NVM超级块结构：nvm\_super\_block

结构体nvm\_super\_block用来表示NVM超级块，该内存结构与NVM中首部的硬盘超级块结构对应，需要回写和保证一致性。包含以下成员变量：

|  |  |  |
| --- | --- | --- |
| 类型 | 名称 | 说明 |
| \_\_u8 | uuid[16] | 128-bit uuid，与所要加速的SSD设备UUID相同 |
| \_\_le32 | mpt\_blkaddr | 映射表区域起始块地址 |
| \_\_le32 | main\_blkaddr | NVM中MAIN区域起始块地址 |
| \_\_le32 | segment\_main | NVM中MAIN区域segment总数，该值也是segment位图（segment\_map）的大小（以位为单位） |
| \_\_le32 | segment\_free | NVM中MAIN区域空闲segment总数 |
| unsigned int | mpt\_ver\_map\_size | MPT版本位图的大小：即MPT所占page数目 |
| unsigned int | mpt\_size | MPT表大小：即映射项的数目 |
| unsigned char | map[1] | MPT版本位图：记录每个有效MPT块在双份segment中的哪一个 |

### NVM设备相关内存结构：nvm\_sb\_info

结构体nvm\_sb\_info保存了与NVM设备相关的信息，是一个内存结构，没有对应的磁盘结构（部分信息来自NVM超级块，回写时随NVM超级块回写）。包含以下成员变量：

|  |  |  |
| --- | --- | --- |
| 类型 | 名称 | 说明 |
| struct nvm\_super\_block \* | nsb | 关联NVM超级块 |
| struct  block\_device \* | nbdev | 指向NVM设备信息的指针。初始化时根据ndev\_path建立block\_device结构并关联到nbdev |
| unsigned int \* | mpt | MPT区域在内存中的Cache,挂载时设置，checkpoint时同步到硬盘 |
| unsigned long \* | mpt\_dirty\_map | MPT表脏页位图：每一个位代表一个MPT page是否为脏，总大小也就是MPT page的数目。映射项被修改后将该项所在页标记为脏，用于回写 |
| unsigned int | mpt\_dirty\_map\_size | mpt\_dirty\_map的大小：（以位为单位） |
| spinlock\_t | mpt\_dirty\_map\_lock | mpt\_dirty\_map lock |
| unsigned long \* | mpt\_ver\_map | MPT版本位图：用于双备份机制，记录每个有效MPT块在双份segment中的哪一个 |
| spinlock\_t | mpt\_ver\_map\_lock | mpt\_ver\_map lock |
| unsigned long \* | segment\_map | MAIN区域segment有效性位图 |
| spinlock\_t | segment\_map\_lock | segment\_map lock |
| \_\_u8 | nvm\_flag | ①NVM\_NSB\_DIRTY:最低位记录nsb是否变脏并回写，ckpt\_flag（f2fs\_checkpoint结构中的一个成员，下同）判断这个字段来决定是否更换nsb版本号（见3.1.5）  ②NVM\_FIRST\_MOUNR:初始化时需要从SSD的META区域读取数据，不应重定向到NVM设备，次低位作此标志 |
| #define NVM\_NSB\_DIRTY 0x01 | | 记录nsb是否变脏并回写 |
| #define NVM\_FIRST\_MOUNR 0x02 | | 是否第一次挂载，读META转移到NVM设备 |

### sbi关联NVM设备信息

在sbi（f2fs\_sb\_info结构实例，下同）中新增成员指针nsbi，关联NVM设备

|  |  |  |
| --- | --- | --- |
| 类型 | 名称 | 说明 |
| struct nvm\_sb\_info \* | nsbi | 关联NVM设备信息 |

### fsb保存NVM设备路径

在fsb（f2fs\_super\_block结构实例，下同）中利用保留区域，新增ndev\_path成员，在SSD关联NVM设备之后，保存NVM设备路径。

新增联合体：

union {

/\*\_\_u8 reserved[314]; /\* valid reserved region \*/\*/

\_\_u8 reserved[314]; /\* valid reserved region \*/

\_\_u8 ndev\_path[MAX\_PATH\_LEN];//设备路径

};

|  |  |  |
| --- | --- | --- |
| 类型 | 名称 | 说明 |
| \_\_u8 | ndev\_path[MAX\_PATH\_LEN] | NVM设备路径：需要在第一次关联NVM设备时设置；取消关联NVM设备时清除。注意fsb在第一次设置之后要立即同步到硬盘 |

### 记录nsb版本号保证一致性

利用f2fs\_checkpoint结构中ckpt\_flags字段，记录nsb版本号，代表当前有效的checkpoint版本对应的有效nsb是两个NVM超级块中的哪一个，以此保证一致性。

|  |  |  |
| --- | --- | --- |
| 类型 | 名称 | 说明 |
| #define CP\_NSB\_VER\_FLAG 0x00001000 | | 标志nsb版本：0或1 |

### 其他说明

在原有f2fs代码基础上需要额外实现的相关逻辑代码，封装成函数，定义到nvm.c文件，声明到nvm.h文件

## 初始化模块

### 概述

初始化主要面临两种情况：

1. **第一次联合挂载SSD和NVM设备。**需要初始化NVM设备，包括建立设备布局，统一SSD和NVM设备标识码等。然后按照②操作。
2. **非第一次挂载。**加载NVM超级块到内存，关联到sbi.

### 具体实现

在挂载f2fs文件系统的过程中，在适当位置添加初始化NVM设备信息的代码，主要包含以下流程：

1. **解析挂载选项。**（参考parse\_options）。**解析挂载选项之前，首先判断fsb中是否已经设置ndev\_path：**

YES：表明之前已经关联NVM设备，则无需额外解析挂载选项，进入③；

NO：解析是否含有ndev\_path挂载选项：

NO：非法退出（因为这意味着没有给出加速设备）；

YES：进入②.

1. **设置NVM设备路径。**把fsb->ndev\_path设置为挂载参数中ndev\_path的值，记录标志，第一次挂载：设置nsbi->nvm\_flag标志位NVM\_FIRST\_MOUNR.
2. **读取NVM设备信息，关联到**sbi**。**创建nvm\_sb\_info内存结构nsbi，根据ndev\_path获取NVM设备信息block\_device (参考blkdev\_get\_by\_path函数)，将设备信息关联到nsbi->nbdev，并将nsbi关联到sbi->nsbi.
3. **创建NVM超级块内存结构。**先判断是否是第一次挂载：

YES：创建nvm\_super\_block内存结构nsb，根据设备信息（sbi->nsbi->nbdev）和sbi信息初始化nsb和nsbi中所有字段，并将其关联到sbi->nsbi->nsb，进入⑦.

NO：判断nsb的uuid与fsb的uuid是否一致：

NO：非法退出；

YES：则到第⑤步

1. **读取NVM超级块。**由设备信息，读取NVM超级块（参考read\_raw\_super\_block函数），并关联到sbi->nsbi->nsb；根据超级块设置nsbi中相关字段（mpt\_ver\_map、segment\_map）。
2. **读取MPT映射表信息。**

A：读取MPT到sbi->nsbi->mpt，即在nsbi中缓存了一份 最新MPT信息供频繁查找使用；

B：计算mpt\_size，并为mpt\_dirty\_map分配内存，两者均关联到sbi->nsbi中。进入⑧.

1. **初始化NVM设备布局。**

A：读取SSD的META区域数据，读取完成后取消设置nsbi->nvm\_flag标志位：NVM\_FIRST\_MOUNR，标记所有META page为脏，并重新写入NVM的META区域；

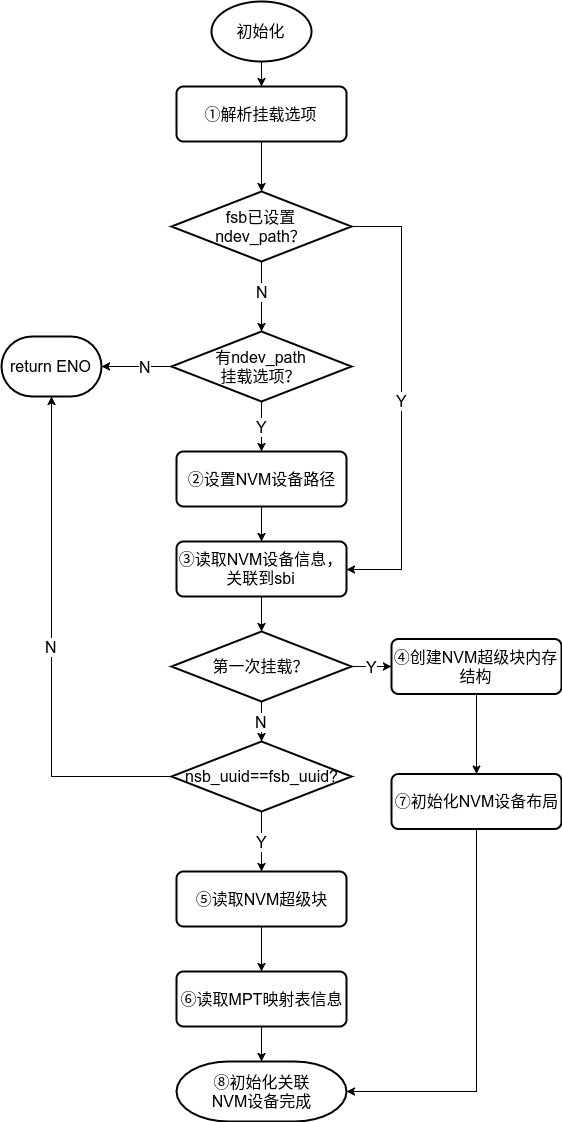
B：将NVM超级块数据写入mapping，设置对应page为脏（初始化写回0号超级块就可以了）；（A、B两个步骤数据均视为META，直接回写META即可）

C：写回两个SSD的fsb信息（参考f2fs\_commit\_super）到SSD超级块区域，这里主要为了持久化ndev\_path信息。

等待以上三者写回完成才算关联设备成功。注意含有ndev\_path的fsb要最后写回（防止系统崩溃带来一致性问题），因为这个数据是我们下一次挂载设备时候判断是否关联设备成功的标志。

1. **初始化关联NVM设备完成**

### 流程图



## 读写模块

### 概述

为SSD使用加速盘，现有的f2fs文件系统的读写逻辑主要有两个部分受到影响：

1. **新建NVM segment。**按照原有机制，当一个segment的块耗尽时，只需从SSD找到一个空闲segment，并分配给文件系统使用。对于本方案需要额外从NVM分配segment，并建立两者之间segment的映射关系。
2. **BIO重定向。**提交BIO时，需要根据映射关系，视情况将读写请求转向到NVM设备。

### 具体实现

#### 新建NVM segment

当前segment没有空闲块时，会建立新的segment(见f2fs\_allocate\_data\_block函数)，在原有机制中，新建segment主要通过调用allocate\_segment\_by\_default函数，从free\_segmap中寻找空位，置为1，并得到segno，设置curseg信息并返回。

在使用NVM加速的混合存储实现中，在上述操作之后，需要额外添加以下操作：

1. **判断当前segment是否是HOT类型**

NO：直接结束额外操作；

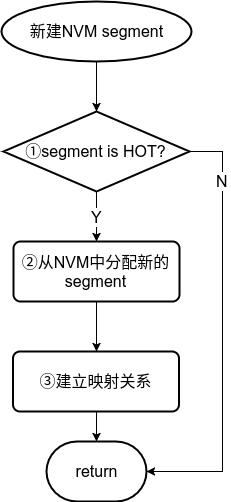
YES：进入下一步。

1. **从NVM中分配新的segment.**从sbi->nsbi->segment\_map位图查找空位，将空位 置为1，并得到NVM的segno；然后将空闲segment数目减1：sbi->nsbi->nsb->segment\_free

在此之后，判断空闲segment数目是否低于一个阈值（待定），如果低于阈值，调用后台NVM GC，释放部分NVM segment.

1. **建立映射关系。**根据得到的SSD segno和NVM segno，建立映射关系到sbi->nsbi->mpt,并设置mpt脏页信息：sbi->nsbi->mpt\_dirty\_map，供读写块设备时候使用。
2. **新建NVM segment操作完成。**

#### 流程图



#### BIO重定向

根据SSD和NVM的segment映射关系，读写时需要对BIO进行重定向。BIO的重定向对上层透明，这意味着，在提交BIO（submit\_bio）之前，所有针对MAIN区域和META区域的块操作所看到的都是SSD中的块号。

在submit\_bio的前置函数\_\_submit\_bio中，实现BIO的重定向逻辑，主要操作如下：

1. **SSD块地址获取和判断。**根据BIO中sector号（bio->bi\_iter.bi\_sector）得到对应SSD中块地址：ssd\_blk\_addr（参考SECTOR\_TO\_BLOCK函数）

IF：ssd\_blk\_addr小于NVM的MAIN区域起始地址，则读写的是META数据，进入第②步；

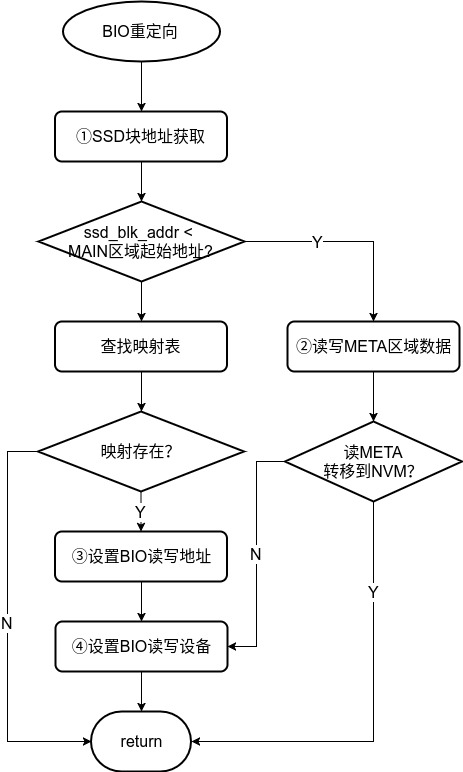
ELSE：根据ssd\_blk\_addr计算得到这个块在SSD中segment号：ssd\_segno，以及在这个segment中的偏移offset，根据ssd\_segno查询映射表：sbi->nsbi->mpt：

IF：不存在映射，结束重定向返回；

ELSE：存在映射，得到对应的NVM中segment号：nvm\_segno，进入第③步。

1. **读写META区域数据。**判断sbi->nsbi->nvm\_flag标志位：NVM\_FIRST\_MOUNR，如果是第一次挂载时转移META数据，无需重定向。否则，重定向到NVM设备，由于NVM的META区域采用无偏移的直接映射，所以BIO中sector不需要进行更改，只需更改读写设备即可，进入第④步。
2. **设置BIO读写地址。**根据SSD中块地址在所在segment中偏移offset，计算得到该块在NVM设备中的块号nvm\_blk\_addr，由此计算bio->bi\_iter.bi\_sector（参考SECTOR\_FROM\_BLOCK函数）。
3. **设置BIO读写设备。**根据sbi->nsbi->nbdev设置bio->bi\_disk和bio->bi\_partno（参考bio\_set\_dev函数）。
4. **完成BIO重定向**

#### 流程图



## Checkpoint模块

### 概述

Checkpoint检查点操作保证一致性，并实现部分前滚，尽可能减少数据损失。按照本方案，新的Checkpoint机制，需要额外保证NVM超级块和MPT区域数据的一致性，其中MPT区域视为META区域一部分。

### 具体实现

原有的Checkpoint机制主要有以下几个步骤（黑体部分）：

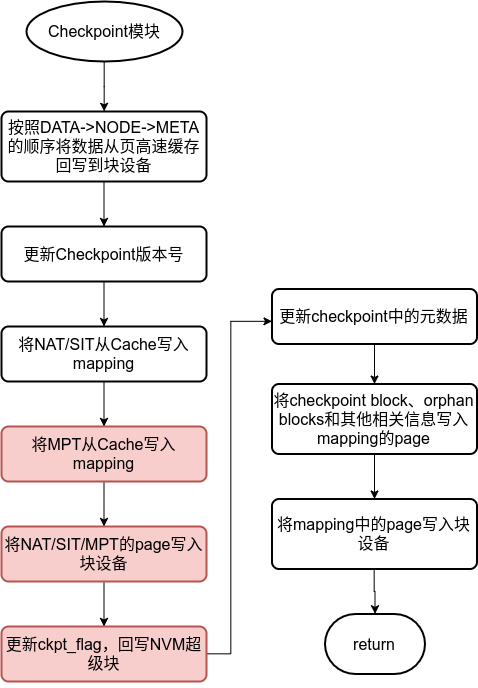
1. 按照DATA->NODE->META的顺序将数据从页高速缓存回写到块设备（META包括了SIT/NAT/Checkpoint）
2. 完成数据的写入后更新Checkpoint版本号：checkpoint\_ver
3. 再将NAT/SIT从Cache写入mapping（代称页高速缓存，下同）的page
4. 根据mpt\_dirty\_map是否为脏，将mpt、nsb从Cache写入mapping的page，并将该page设为dirty；设置nvm\_flag标志位：NVM\_NSB\_DIRTY.

具体：

1. 循环查找mpt\_dirty\_map中每一个位，找到脏的位（代表一个MPT区域脏页），下标即脏页在MPT区域的块偏移量，只要找到一个，说明有脏页，设置nvm\_flag标志位：NVM\_NSB\_DIRTY.C步骤中需要回写nsb.
2. 根据偏移获取META区域mapping中下一个要使用的version block（参考get\_next\_nat\_page函数），将缓存的mpt\_map中的对应page长度内存拷贝到该version block在mapping中对应的page里；将该page设为dirty（参考set\_page\_dirty函数），等待回写。
3. 如果有脏MPT page，则nsb一定也为脏，切换版本号(这里并不是真正切换CP中的版本号，只是切换版本用于写入)，并拷贝cache中的nsb到切换版本后mapping中对应的NVM超级块page中，将该page设为dirty。NVM超级块和MPT区域统一视为META数据进行回写。
4. 完成上述工作后调用do\_checkpoint()进行实际的Checkpoint，即 将最新的元数据写入块设备。do\_checkpoint()包括以下工作：
5. 将NAT/SIT/MPT/NSB的page写入块设备（这一步将所有META区域脏页刷回块设备，MPT、NSB视为META）
6. 更新checkpoint中的元数据
7. 将checkpoint block写入page
8. 将orphan blocks写入page
9. 根据NVM\_NSB\_DIRTY，视情况切换nsb版本，更新ckpt\_flag中的CP\_NSB\_VER\_FLAG。
10. 将其他相关信息写入page
11. 将上述mapping中的page写入块设备

为了与原有机制融合，保证文件系统一致性，我们在原有Checkpoint机制中融入了nsb和MPT区域的回写（红色部分为新加）。

### 流程图



## Recovery模块

## SSD GC模块

### 概述

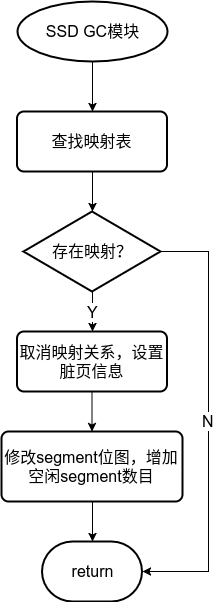
也就是原有的GC机制。受害segment选择维持原来的方式，不同的是，每次选中受害segment，回收SSD中segment同时，需要检查是否该segment号在NVM中有映射，如果有映射，需要同步回收NVM中的segment，并取消映射关系。

### 具体实现

代码逻辑集中在do\_garbage\_collect函数中。对每个SSD segment回收之后，需要额外做以下操作：

1. **查找映射表。**根据回收的SSD segno查询sbi->nsbi->mpt映射表，如果不存在映射则忽略额外操作；存在映射就进行后续操作。
2. **取消映射关系，设置脏页信息。**将SSD segno和NVM segno对应的映射置为0，以取消映射；同时计算两个segment映射项在MPT区域的块偏移offset，根据偏移将sbi->nsbi->mpt\_dirty\_map（MPT区域脏块位图）对应的位 置为1，表示该MPT块被修改，需要回写。
3. **修改segment位图，增加空闲segment数目。**根据NVM segno，修改sbi->nsbi->segment\_map位图，将对应的位 置为0（表示该位空闲可用），同时将sbi->nsbi->nsb->segment\_free空闲segment数目加1.

### 流程图



## NVM GC模块

### 概述

原有的GC显然无法满足本方案的需求，一个明显的例子：SSD空间足够，NVM空间已经不足，那么我们需要将NVM中相对的“冷”数据刷回SSD，并取消映射，以此为之后映射关系建立保留足够的NVM空闲块。

NVM的GC与原有GC触发的条件和所要进行的工作存在很大差别，两者的触发也有不同的标准，不能混为一谈。

### 具体实现

#### NVM GC的触发

参照原有GC的触发机制，本方案同样在f2fs\_balance\_fs函数中添加触发条件。

对于f2fs\_balance\_fs函数，我们主要关心触发垃圾回收部分代码。按照原有的机制，如果没有足够的可用segment（垃圾回收以section为单位，但是通常1个section=1个segment）（参考has\_not\_enough\_free\_secs函数），则调用f2fs\_gc函数进行垃圾回收。原有的判断只针对SSD可用segment，在此，我们需要额外增加对NVM中可用segment的判断，如果可用segment不足，触发NVM GC进行相应空间回收操作。

判断的逻辑很简单，我们只需要获取sbi->nsbi->nsb中空闲segment数目（segment\_free），与总数目（segment\_main）作对比，根据空闲比例，来确定是否触发NVM的垃圾回收。

#### NVM垃圾回收

垃圾回收的基本理论很简单：选中segment，回收segment。

关于如何选中segment，整体思路是，尽可能选中IO少，数据“冷”的segment，因为这些segment使用频率低，对于奢侈的NVM空间是一种浪费。

选中segment之后，我们对其进行所谓的“回收”，我们的“回收”主要有以下流程：

1. **读取segment。**将该segment所有page读取到内存中，此时page不应被标记为脏，因为映射未取消，可能造成无用回写。
2. **取消映射关系。**全部读取完毕之后，取消SSD和NVM对应segment之间映射关系（sbi->nsbi->mpt）。
3. **提交回写。**将所有page标记为脏（参考set\_page\_dirty函数），提交给系统或Checkpoint机制回写。这里无需强制回写并等待完成。因为如果此时系统崩溃，映射修改并未反映到块设备，不会产生一致性问题。
4. **垃圾回收完成。**

**流程图**

