NAND FLASH

# 1 flash的基本组成单元 SLC/MLC/TLC

## 1.1特点

1 以page为单位读写

2 以block为单位擦除

3 集成密度高、单位比特成本低和可靠性高

4 对于Nand的读写，只能由1变成0，不能由0变成1。因此要先对Nand进行erase(0变成1)，然后再写。

## 1.2 分类

SLC是对电子数目做的一阶量化，所以一个晶体管可以代表两个状态：0和1。

如果我们对晶体管中的电子数目做二阶量化，一个晶体管就可以代表四个状态：少于25%代表00，25% ~ 50%代表01，50% ~ 75%代表10、大于75%代表11。这就是MCL(Multi Level Cell)的做法。

当然我们还可以对晶体管中的电子数目做三阶量化，一个晶体管就可以代表八个状态：000、001、010、011、100、101、110、111。这是TLC(Three Level Cell)的做法。

也就是说，量化等级越高，一个晶体管可以表示的状态越多，存储密度就越大，同等数量的存储单元组成的存储介质，存储容量也越大。



如上图所示，可以把晶体管比作水桶，晶体管内的电子比作水：

由于SLC只有两个状态，只需要保证电子数目多于一半或少于一半即可，所以注入/排放电子的过程比较简单，执行起来很快；对于MLC，有四个状态，对注入/排放电子的精度要求就比较高，执行起来就没那么快了；对于TLC，有八个状态，对注入/排放电子的精度要求更高，执行的就更慢了。另外，写入的数据也会影响效率，比如对于MLC，写入代表满状态的00和代表空状态的11，效率要高于01和10（清空一杯水和倒满一杯水肯定比倒1/4,、1/2杯水简单，当然这个影响在业务层看来可能比较小）。

另一方面，如果水桶发生损坏，比如在上半部分产生了一个缺口（擦除操作会导致介质磨损，最终导致电子泄露），对于SLC可能没影响，可以正常使用，对于MLC和TLC可能由于无法区分多个状态，就无法使用了。NAND Flash的寿命在很大程度上受所用存储单元类型影响，单个晶体管中存放的状态越多，容错性越差，寿命越短。

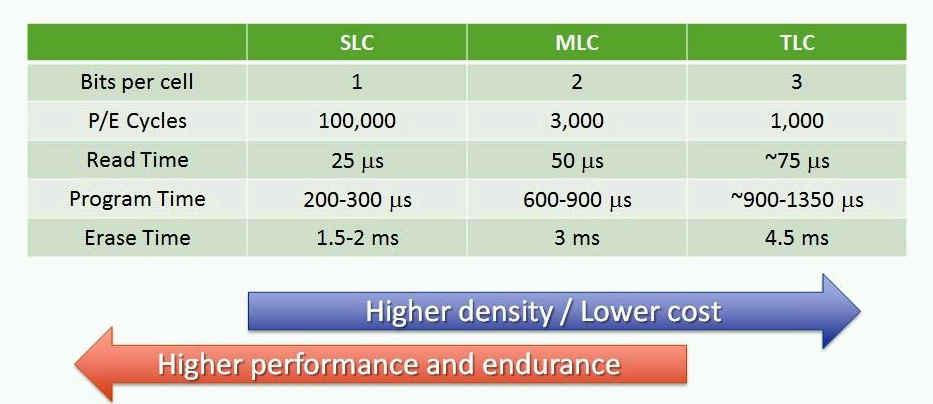
不同组成单元对Flash性能和寿命的影响

从上面的原理可以看出，SLC、MLC、TLC的性能和寿命是递减的，存储密度是递增的。

为什么寿命是递减的？

因为SLC只有两种状态，如果将晶体管比作水桶，就算上半部分由于擦除等操作，出现一个缺口，但还是可以用的，因为其电压范围很宽，但是如果是MLC/TLC，出现一个缺口，将会影响flash的数据状态，导致数据错误，这是非常严重的。所以其寿命相比SLC，要差很多。

下面是一组具体的数据：



从上图可以看出：

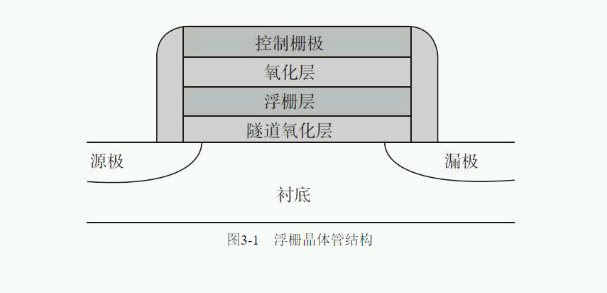
随着每个单元代表的比特数增加，读、写、擦除耗时也在增加

无论是SLC还是MCL、TLC，擦除耗时（ms数量级）都远高于读写耗时(us数量级)

SLC的擦写次数远大于MLC、TLC，也就是说寿命长。

SLC每个晶体管只能代表一个比特，从存储密度看，是最低的，TLC存储密度最高，MLC次之。

## 1.3 闪存的基本存储单元(cell)数据存储过程



# 2 flash的结构和特点

Flash中存在下面几个基本概念：package、die、plane、block、page(page对应于普通硬盘HDD中的sector，即常说的扇区)。 下面是一个示意图，我们由大到小拆解下：

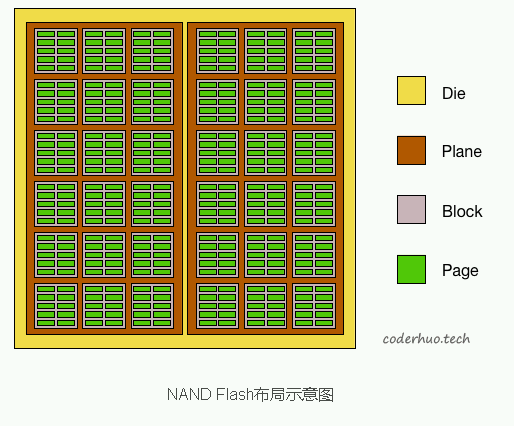
package是存储芯片，即拆解固态硬盘或者SD卡后看到的NAND Flash颗粒。

每个package包含一个或多个die。die是可以独立执行命令并上报状态的最小单元。

每个die包含一个或多个plane(通常是1个或2个)。不同的plane间可以并发操作，不过有一些限制。

每个plane包含多block\*，block是最小擦除单元。

每个block包含多个page， page是最小的读写单元。



**Program/Erase Cycles**

Flash还有一个重要特性：Flash不支持更新操作，严格说应该是不支持原址更新。 如果我们已经往某个page中写入了数据，想修改这个page中的内容，只能通过下面的方法：(读改写)

先把本page所属block的数据全部读出来，比如先读到内存DRAM

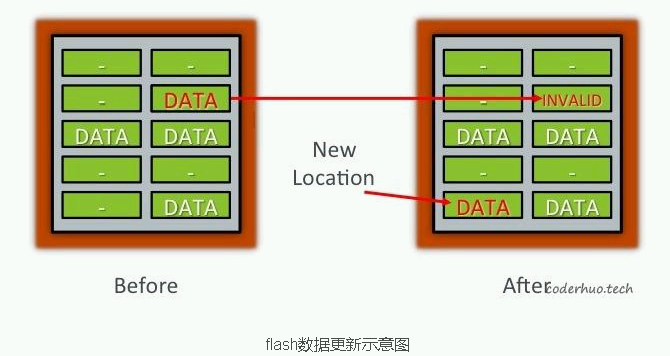
然后修改对应page的内容

接下来擦除整个block

最后把修改后的block数据回写到Flash

Flash芯片上block的擦写次数是有限的，最大擦写次数称为PE Cycles(Program erase cycles, 往Flash写入的过程又称为编程过程，即program)。如果采用上面的方法进行原址更新，Flash很容易就会被用坏的。一个折中的方法是：将新数据写到一个新的page中，并将原来的page标记为无效，如下图所示：

说明：新的page和老的page可以位于同一个block，也可以位于不同的block，甚至位于不同的die。



这样做会带来另外的问题：

1 数据所在page变了，后续如何访问新的数据（谁来维护这个映射关系）

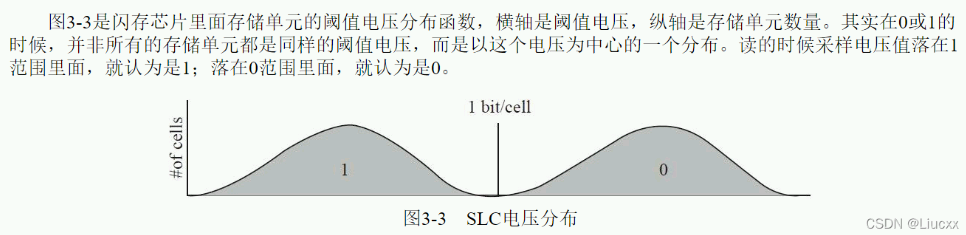
2 无效的page什么时候回收（上面的做法只是延迟了擦除block的时间，但是空闲page迟早有用完的时候）

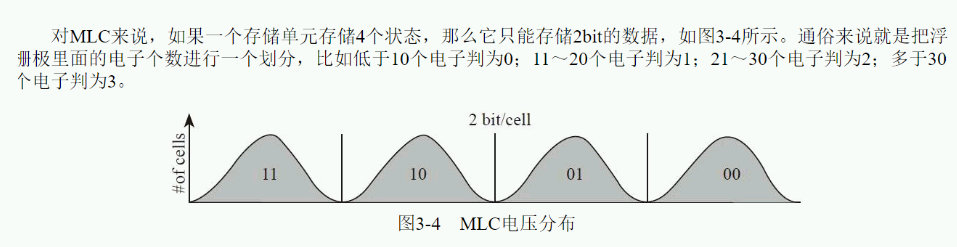
3 如何选择新的page，保证整个flash的擦写均衡（避免有的block擦除次数多，有的block擦除次数少）

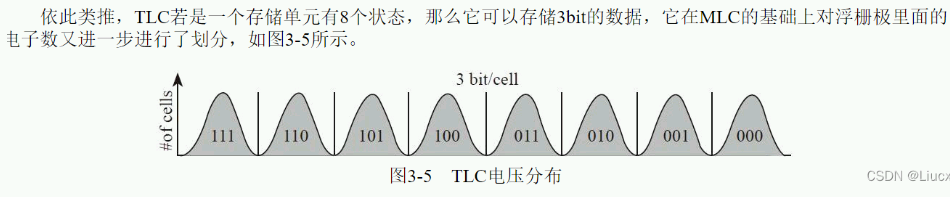
接下来登场的FTL会解决上面的问题。需要注意的是，擦除的耗时远大于读写耗时，相关逻辑处理不好的话会影响性能。所以目前有很多FTL的优化算法。

关于Flash的原址更新补充如下说明：假设一个空白page是全1，比如1111 1111，对它的写操作只能把其中的某些位由1变为0（第一次写可以把1111 1111改成1111 0000，第二次可以继续把1111 0000改成0011 0000，从这个角度看，page可以执行多次写操作），而无法再把0变为1（如果某个写操作涉及把0变为1，那就无能为力了，只有整个block擦除后再写入了）。

## 2.1 电压分布

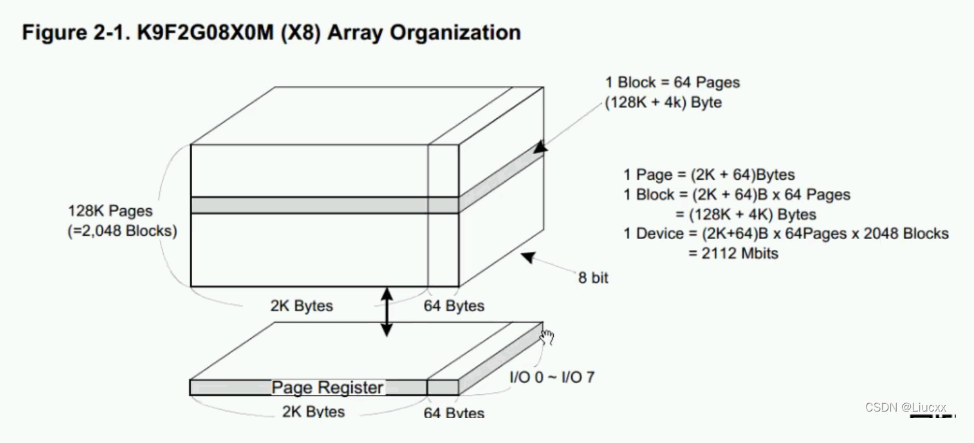






## 2.2 nand flash架构

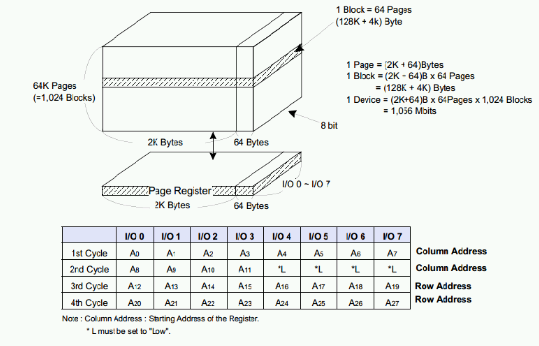
1block有64page，这个flash有128kpage。



硬件管脚图：



由上面来看，nandflash的外围引脚很少，没有地址线，只有数据线，相对于由数据线和地址线的norflash，大大的减少了引脚数目，使得芯片的体积更小，硬件设计也更简单。现在的数据线是复用的，也就是说既可以传数据，也可以传地址，也可以传命令，那么是怎么做到的呢？硬件为了区分当前到底是需要传啥？设计了CLE/ALE命令，所以你必须先发一个CLE/ALE命令，告诉控制器，我现在是命令还是地址，这样就能识别并做相应的处理。我们了解了硬件的连接，那么对于flash是怎么存储的呢？下面来看看这款芯片的的存储布局。



由图可以看出一片Nand flash为一个设备(device)，其数据存储分层为：

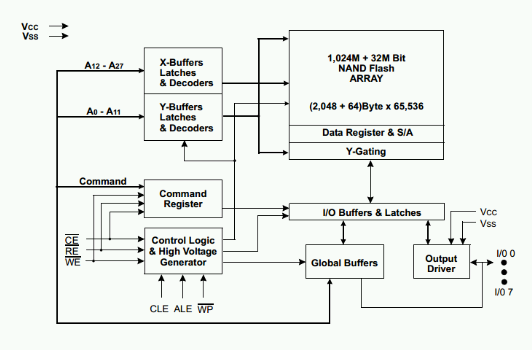
1. 1个设备（device）=1024个块(Blocks)，块也是Nand flash擦除操作的最小单位。

2. 1个块（block) = 64页(Pages)，页是Nand flash写入的最小单位，对于每一个页，由数据块区域和空闲区域。数据区，也容易理解，就是存储一些数据，而对于空闲区，一般也叫做OOB(out of Band)，这个区域，是基于Nand flash的硬件特性设计的，Nand flash在数据读写的时候很容易错误，所以为了保证数据的正确性，必须要有对应的检测和纠错的机制，此机制就被叫做ECC，所以设计了该多余的区域，用来放置数据的校验值。

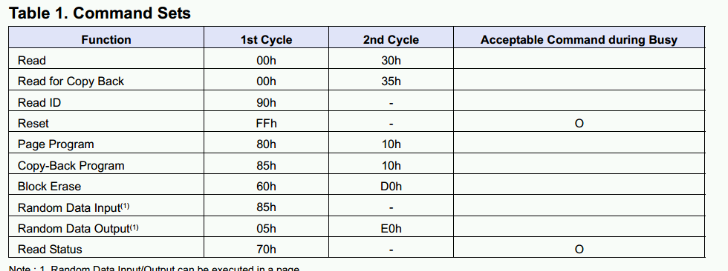
3. 1个页(page) = 数据块大小(2K)+OOB块大小(64Bytes)

**那么通过上面我们就可以计算出怎么访问一个物理地址：块大小\*块号+页大小\*页号+页内地址。**

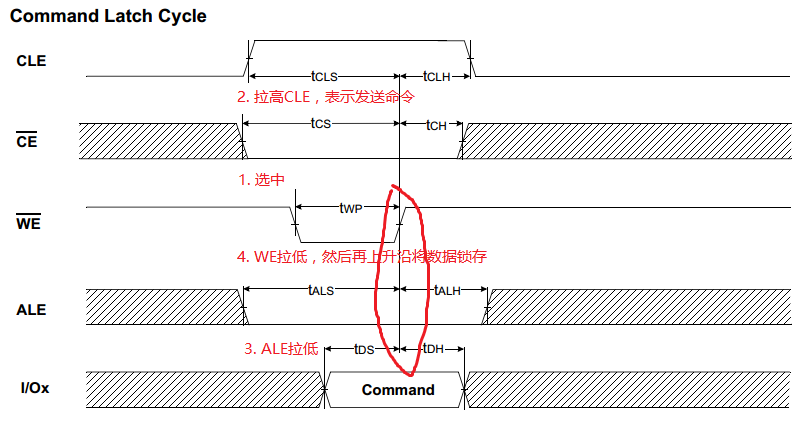
从硬件的图来看，对于K9F1G08X0C这款芯片，其容量为132MB，那么就应该需要28条地址线，而现在只有8条地址线，对于Nand flash就导入了地址周期的概念，对于该款flash，所以需要4个周期：2个列地址(Column)周期和2个行地址(ROW)周期。从下面的功能框图来看，对于列地址A0--A11，就是页内地址，地址范围就是0--4094，与页内地址(2K+64)吻合，其实对于页内地址，其实只需要A0-A10，而对于多出来的A11，是用来表示页内的oob区域。那么对于A12-A27就是用来表示属于哪一个块和块里面的哪一个页号。

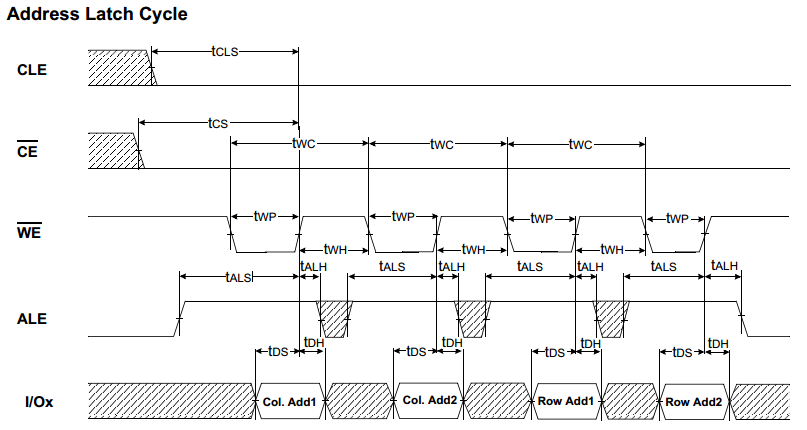


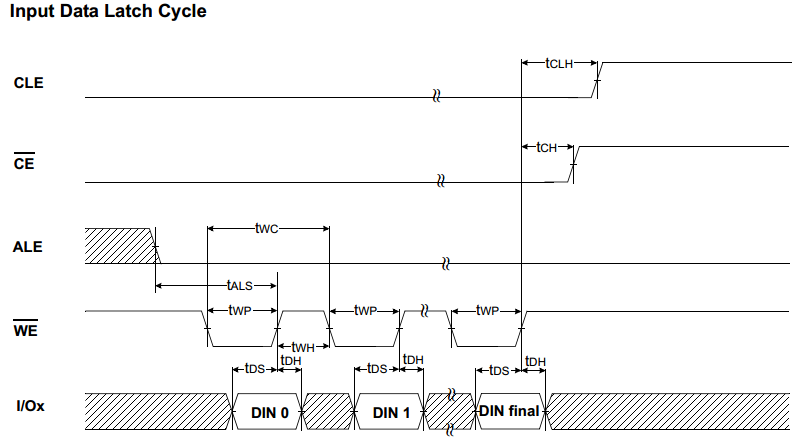
通过前面对于nand flash的基本原理的讲解，下面来看看怎么操作nand flash，下面是K9F1G08X0C支持的命令格式。



下图是发送一个命令/地址/数据的完整过程







通过时序图看出，上面有很多时间参数，在设置Nand flash控制器的时候，需要确认这些时间参数是否在合适的范围内，对于控制命令都是通过上面的几个基础命令组合而来。

NAND闪存是由成千上万个cell组成的Array（阵列）。Cell之间是串行连接的，32或64个一组，组成一个String（串）。每个cell的控制极连接Word Line（字线），string末端的cell漏极连接Bit Line（位线），首端的cell源极连接Source Line（源线）。串与旁边的串共用位线。Page（页）由同一根字线上的cell组成，是一种逻辑概念。位线分为两组，even和odd，分别组成交叉的2组页。一根字线上的页数和每cell存储的比特数有关，SLC为1bit/cell，MLC为2bit/cell，TLC为3bit/cell，QLC为4bit/cell。对于SLC闪存，一根字线上有2页。而MLC闪存每cell能存储2bit数据，可以在逻辑上分成2个页。因此对于MLC闪存，一根字线上有4页。一组字线连接的所有串组成一个Block（块）。

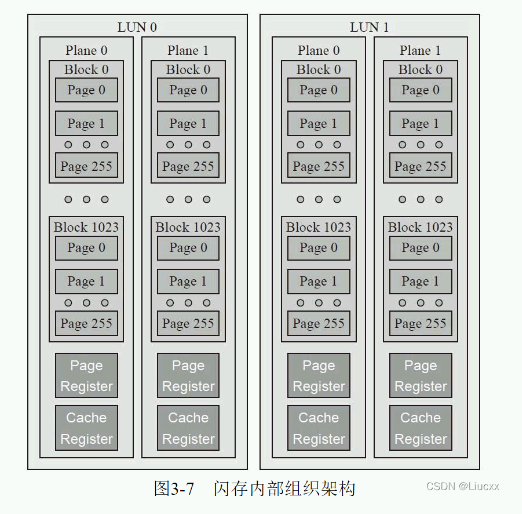
在NAND闪存中，一个块中的所有cell共用一个衬底，因此NAND闪存的最小擦除单位是块。



## 2.3 DIE/LUN

是接收和执行闪存命令的基本单元

不同的LUN可以接收和执行不同的命令，同一个LUN一次只能执行一次命令。(不能同时读和写)



1 主控写数据: 先写到cache register，再由cache register写到闪存阵列。 以page为单位，读数据相反。



### 2.3.1 cache register、page register作用？

1 优化闪存访问速度。

发送数据时(cache -> 主控)，可以同时读下一个page到page register(闪存-> page register).

接收数据时(page -> 闪存)，可以将下一个要存的数据放到cache中(主控-> cache register).

## 2.4闪存的接口

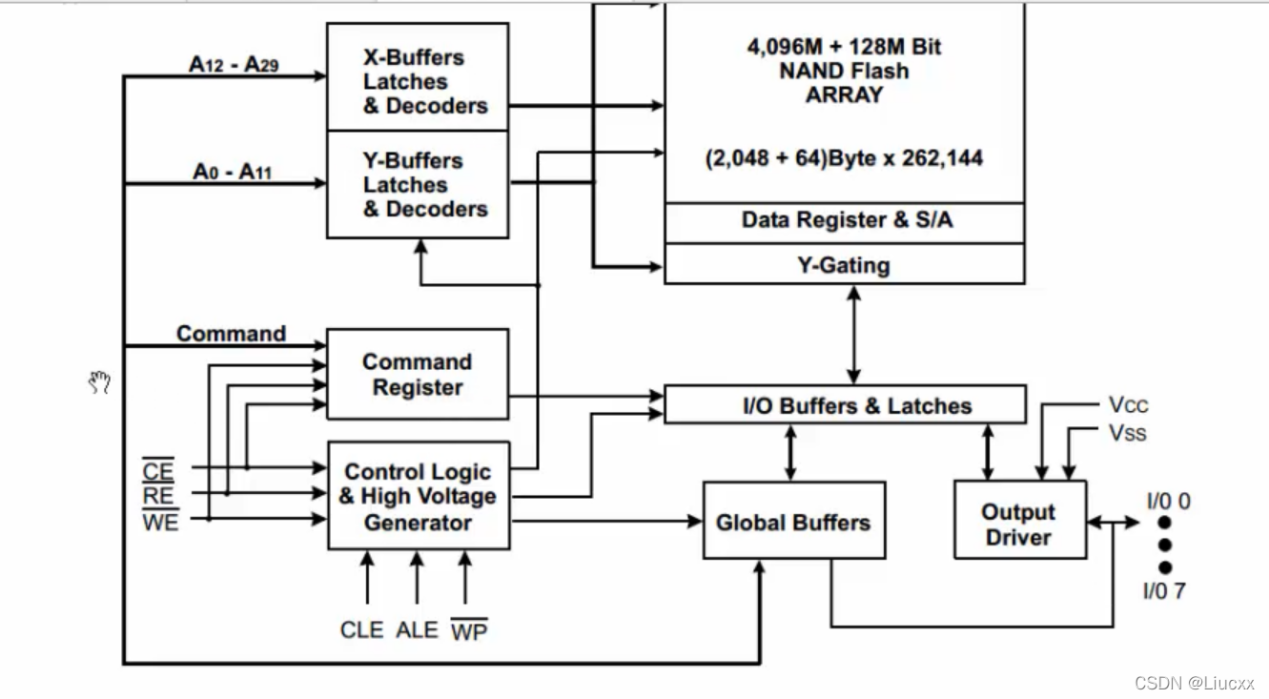
### 2.4.1 同步和异步闪存接口对比

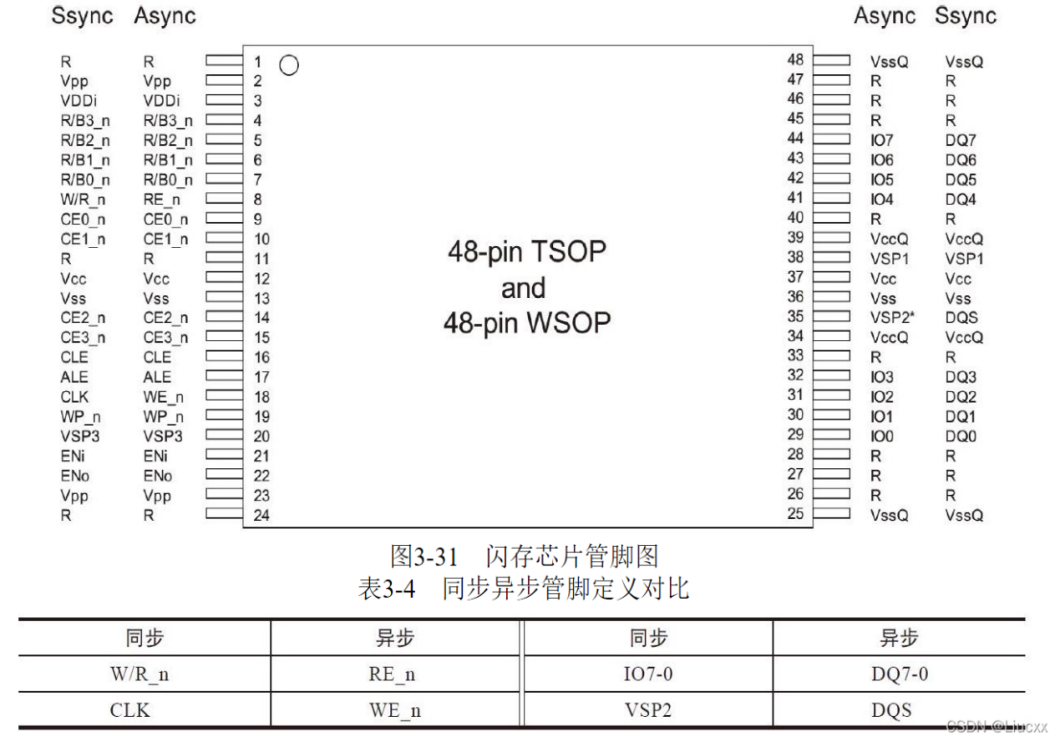
同步和异步啥意思？

闪存接口有同步异步之分。一般来说，异步传输速率慢，同步传输速率快。

（1）异步接口没有时钟，每个数据读由一次RE\_n信号触发，每个数据写由一次WE\_n信号触发。

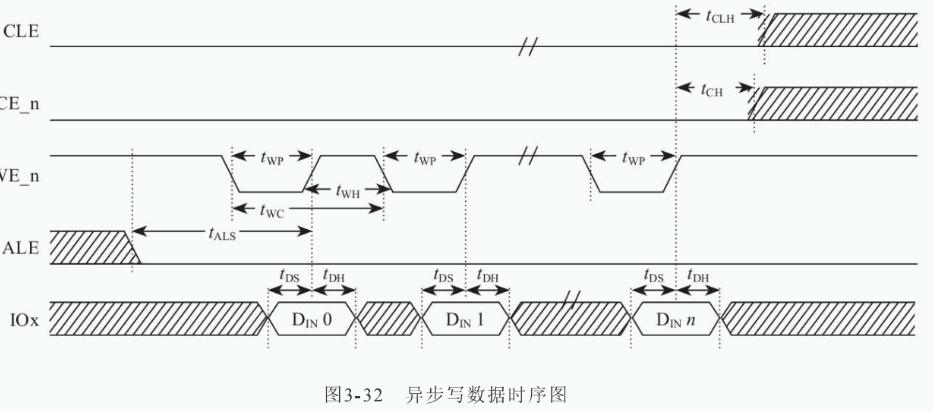
（2）同步接口有一个时钟信号，数据读写和时钟同步。

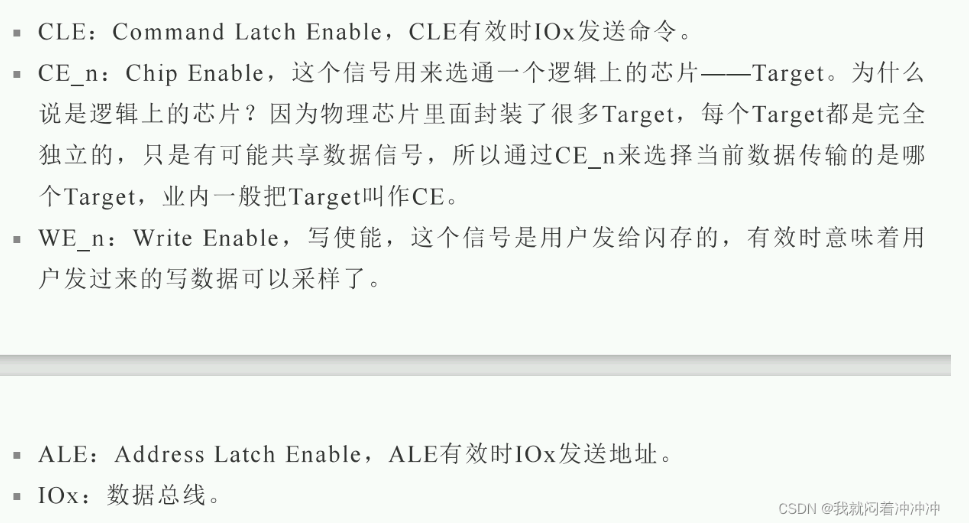




#### 2.4.1.1 异步时序

异步写时序图：

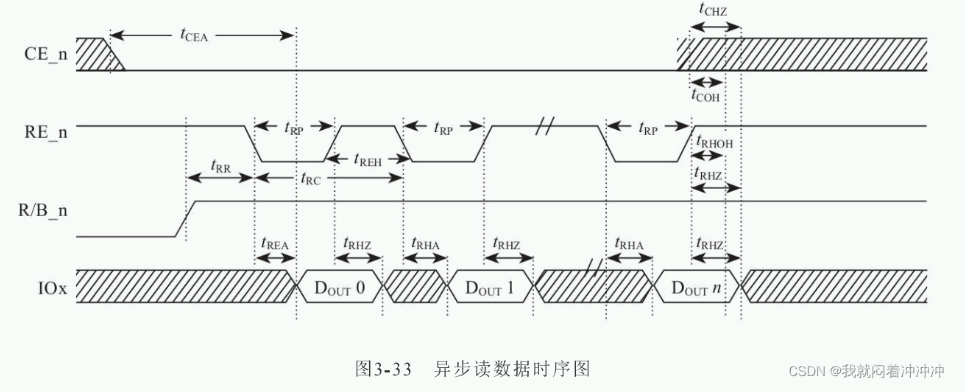


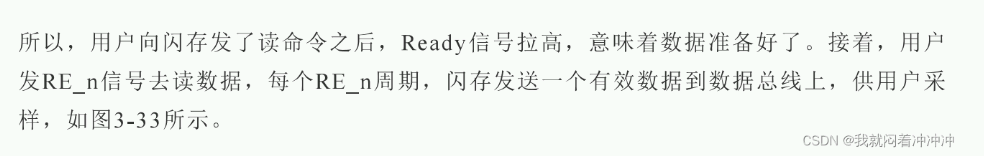


从上图看出，数据写入的时候，数据总线不能传输地址和命令，所以ALE和CLE无效。每一个WE\_n周期对应一次有效的数据传输。

异步读数据：





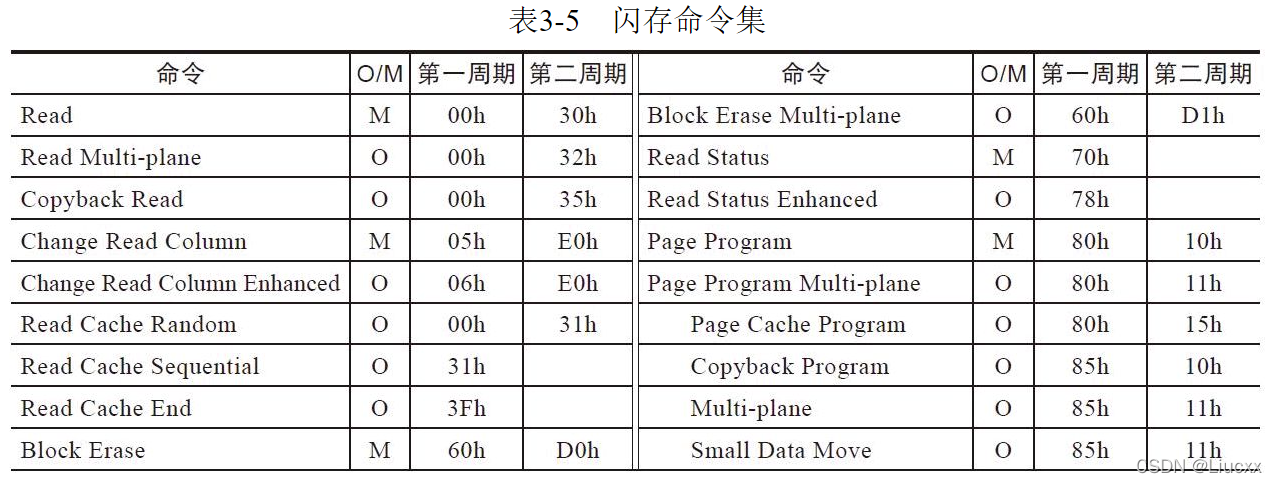


同步模式下DQS信号的上升沿与下级沿都能控制信号的传输，使传输速度翻倍。

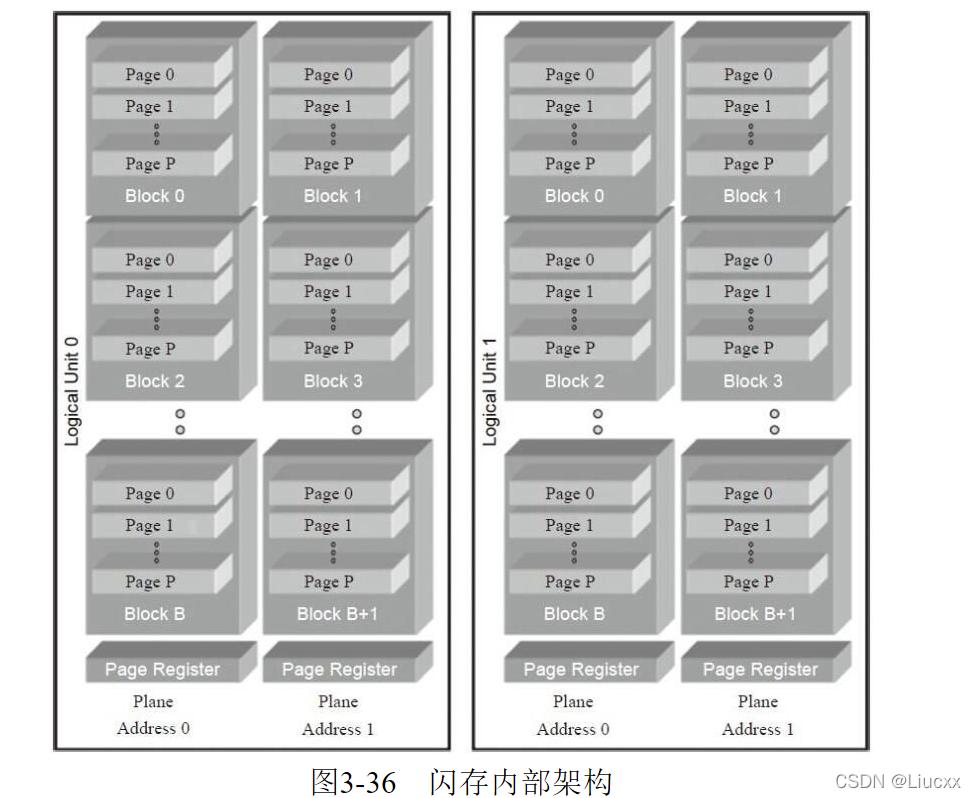
#### 2.4.1.3闪存命令集

1 SSD主控通过闪存命令集来与NandFlash交互。

2 ONFI 2.3协议部分命令集如下



#### 2.4.1.4 闪存寻址



1 闪存使用了行地址、列地址进行访问和管理

列地址: Page内部的偏移地址

行地址: 在ONFI协议中，高位到低位依次表示 : LUN、Block(Plane在Block地址的最低几位)、Page地址。具体位宽由芯片决定。



#### 2.4.1.5读、写、擦时序

读时序如图所示，用户在发送00h-30h之间发送了所有地址，包括2个列地址和2个行地址，发完命令后，SR[6](status register寄存器的Bit6)变为busy，经过一段时间后，SR状态变为ready，就可以读数据了。



写时序如图所示，用户发送80h命令之后，发送写地址，列地址一般为0，因为一般要把一个page写满，所以需要从0page开始写，如果不从头写满一个page，会导致出错，发完地址tADL时间之后，开始传输数据到缓存的闪存。数据传完，发送10h，闪存芯片开始向介质写入数据，SR[6]状态转为busy，在写操作完成后SR[6]状态转为ready。

## 2.5 ONFI与Toggle协议

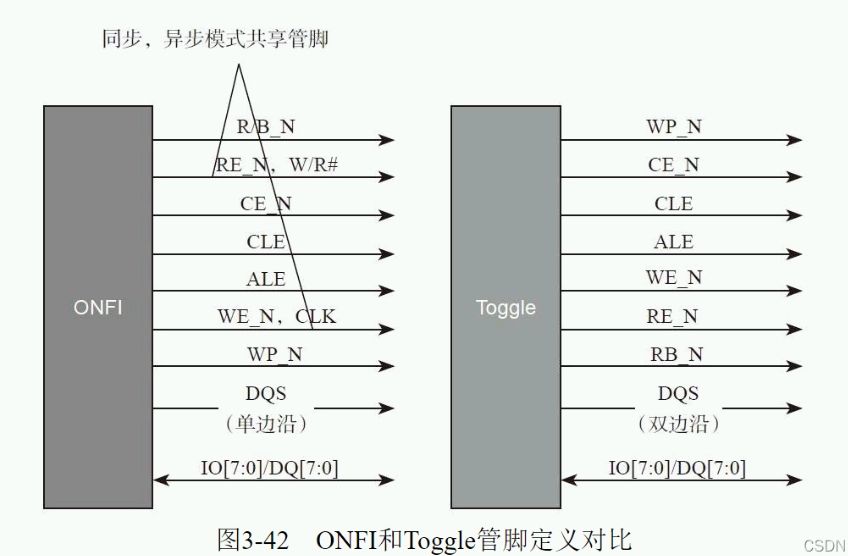
### 2.5.1. Toggle协议

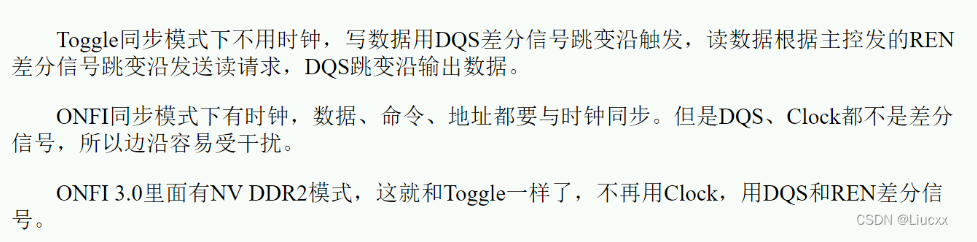
由三星和东芝一起创建。研发出 一款新的闪存产品 Toogle Nand。

### 2.5.2. ONFI协议

ONFI(open NAND Flash Interface): 许多个公司一起制定的协议。

### 2.5.3两种协议对比





## 2.6闪存特性

### 2.6.1闪存存在的问题

闪存坏块。 需要有坏块管理机制

读干扰。 可能会影响其它page的某些bit反转

写干扰。 可能导致bit翻转

存储单元间的耦合

电荷泄露

### 2.6.2. NandFlash寿命

问题: 频繁读写某些块,会导致磨损、坏块等问题

解决方法:

Wear Leveling 磨损平衡算法

降低写放大

纠错算法优化(提高容错率)

### 2.6.3MCL使用特性

1 对于MCL或TLC:

写一个闪存块的闪存页，应该顺序写page0, 1. 2...

读没有这样的限制

1 MLC特有的问题

1.1 MLC最大擦写次数会变小。 （使用wear leveing保证寿命）

1.2 一个闪存页写失败，可能导致另外页损坏

1.3 不能随机写

1.4 写lower page快，写upper page慢

2 MLC应对存在问题的一些方法

2.1可以只写lower page

2.2Lower和upper打包写

2.3定期填充upper page

2.4 写Lower page时，备份数据，直到upper写完。

2.5MLC当SLC使用

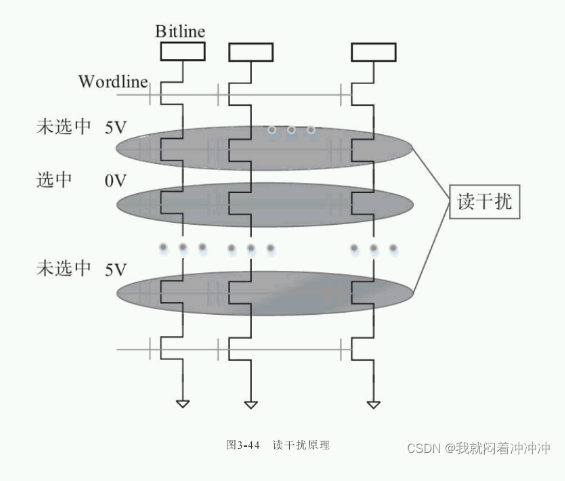
2.6企业级固态硬盘，配备大电容

### 2.6.4读干扰问题

从闪存读取原理来看：当读取一个闪存页Page的时候，闪存块中未被选中的Page的控制栅极都会加一个正电压以保证MOS管是导通的。但，频繁在MOS管的控制栅极加正电压，可能会导致电子被吸进浮栅极，造成轻微写，从而导致比特翻转。但这不是永久性损伤，对Block执行擦除操作后可正常使用。

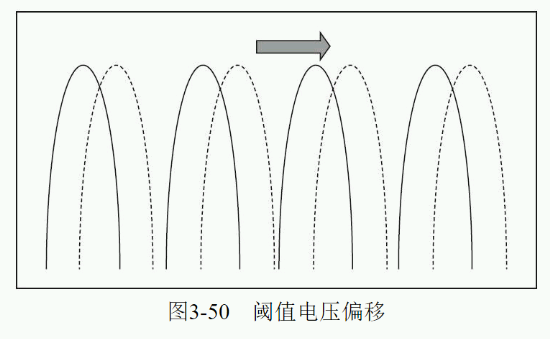
读干扰会导致浮栅极进入电子。有电子进入，会导致晶体管的阈值电压右移！

注意：读干扰影响的是同一个Block上的其他Page，而非读取的Page本身。



读干扰为什么会导致性能下降？

读干扰会导致浮栅进入电子，由于有额外的电子进入，会导致晶体管阈值电压右移(data retention问题会使阈值电压左移)，如下图



由于晶体管的阈值电压发生了变化(变大了)，闪存内部逻辑如果还是按照之前的参考电压加在控制栅极上，然后去判断数据，肯定会发生误判，也就是读到错误的数据。

如何应对干扰？

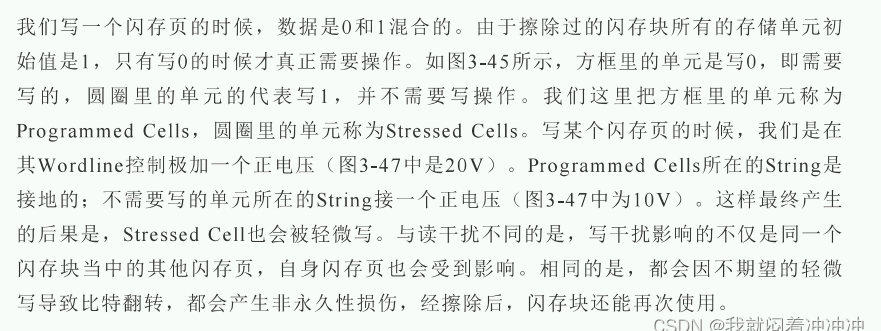
一般的做法是记录每个闪存块读的次数，赶在这个数值到达阈值(厂家提供的)之前，把闪存上的数据重新刷写一遍(读、擦除、写回)，或者把数据搬移到别的地方。

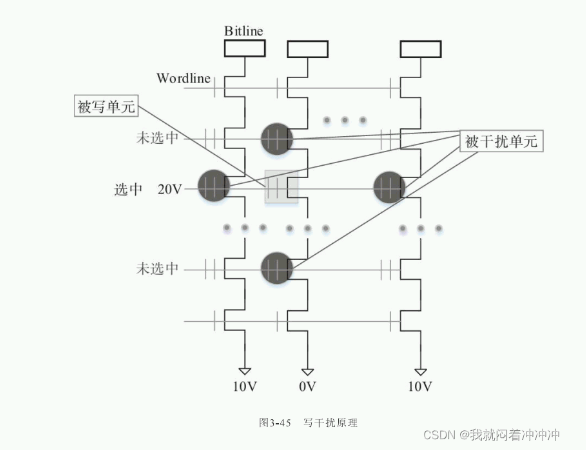
读干扰为什么会导致性能下降？

这是因为需要在到达阈值之前搬移数据，这需要占用带宽，消耗主控性能。

### 2.6.5 写干扰

除了读干扰会造成比特翻转，写干扰也会。





### 2.6.5 闪存数据保存期

1 数据保存期(Data Retention), 从写入操作到电子慢慢泄漏，直到数据出错，这个期限叫作数据保存期。

到了期限的标致是: 闪存读出来的数据无法用 ECC纠错成功。

闪存一般可能存在的错误: 电气问题 、 读写擦失败、ECC纠错失败。

SLC时代，期限有好几年

TLC时代，期限只有几个月

#### 2.6.5.1如何解决数据保存期短问题

1 Read Scrub技术（数据巡检、扫描重写技术）

与ZFS类似，固态硬盘空闲时，扫描全盘，对某个闪存页翻转bit数进行计算，超过一定数量，重写数据。

## 2.7闪存数据完整性

### 2.7.1问题及解决方法

1 问题：容易发生bit位翻转

2 解决方法：

ECC纠错

RAID数据恢复

重读(read retry)

扫描重写技术(read scrub)

数据随机化

### 2.7.2 读错误来源

1 擦写次数多

氧化层老化，电荷容易位置异常，导致读错误

2 Date Retention

随时间变化，电子流失，阈值电压值向左移动，读数据发生误判

3 读干扰

读数据时，需要施加电压，影响到其它page

读的次数多后，阈值电压右移

4 存储单元之间的干扰

存储电子的浮栅极是导体，两导体间构成电容。一个存储单元电荷变化引起另其它单元发生电荷变化 。

周围单元是不同状态时，中心单元的阈值电压不同。

5 写错误

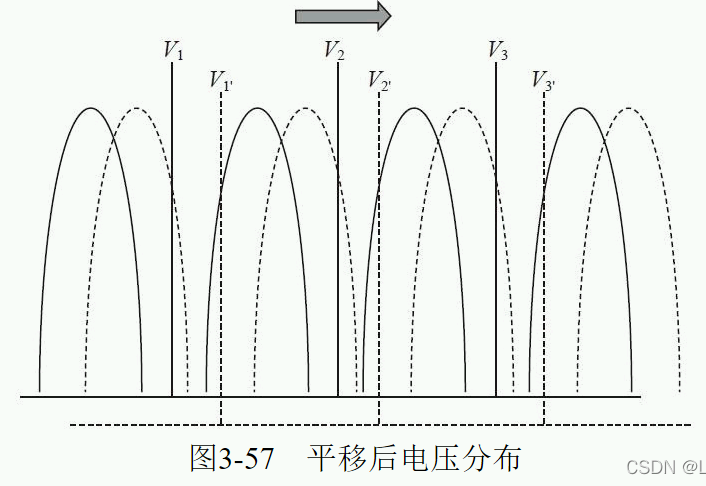
一般发生在MLC、TLC2-pass中，是因为upper page是基于lower page的数据的

TLC1-pass中不存在这个问题，因为upper和lower同时写入。

### 2.7.3 重读

方法1： 不断改变参考电压，找到可读出数据的电压点，直到正确读出数据

方法2: （更复杂的重读） Advanced Retry. 先读附近的单元确定状态，再用不同的参考电压读两次要读的单元，根据附近数据选择结果。



上面那个图不是很懂

### 2.7.4 ECC纠错码(error checking and correcting)

ecc一般属于SSD控制器的一个模块，有些闪存内部也集成了ECC模块

原理:

先通过奇偶校验的方法检查错误，也就是存储一个"奇偶检验"，其值为8 个比特组里所有1的和，结果为奇或偶(1或0)。

下次再访问数据时，进行奇偶校验，与之前存储的校验值对比。若不相同，就采用一段存储时使用的特殊代码进行校正。

BCH 比较常用

LDPC 逐渐成为主流

Nand 中额外的预留空间，部分用来写ECC校验数据。 因此，纠错强度受限于闪存页的预留空间。

#### 2.7.4.1 静态ECC和动态ECC纠正

1 静态ECC纠错方案

ECC纠错单元（用户数据）和ECC校验数据大小在整个固态硬盘生命周期都是固定的。纠错能力一直不变

2 动态ECC纠正方案

刚开始使用少的纠错码，能存放更多用户数据(ECC纠正单元)。随着SSD的使用，将纠错码比例增多，用户数据区变小。从而达到动态调整纠错能力。

优点:

开始时使用少的纠错码，op（Over Provisioning，预留空间）更多，减小写放大。

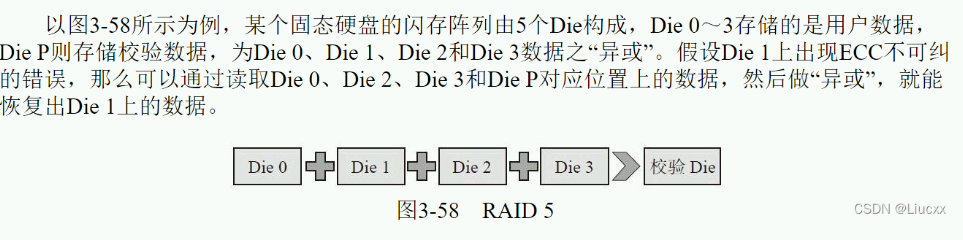
当ECC较少时，带宽利用率越高。

### 2.7.5 raid

RAID(Redundant Arrays of Independent Disks)是另一种纠错算法。

Nand中数据bit翻转个数太多，超过ECC纠错能力，ECC就无法纠正了。因此越来越多的SSD都使用RAID（一般是RAID 5）。

借鉴磁盘阵列技术来确保数据的完整性：



采用RAID 5技术的SSD，只能恢复单个ECC不可纠的数据。

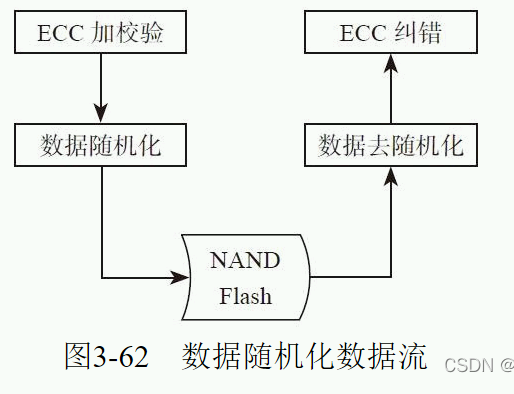
### 2.7.5 数据随机化

若未进行随机化写。会导致Nand内部电量不均匀，导致抗干扰性下降。

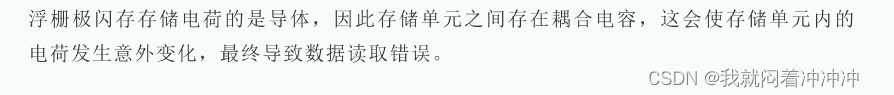
让0和1的分布充分隔离

降低相邻单元之间的耦合电压产生的影响

SSD内部一般都有 数据随机化模块， 对用户写入数据加入扰码，使0 和1比较均匀，减小数据发生翻转概率。一般Nand厂商推荐使用AES算法实现数据随机化。



### 2.7.6 存储单元间的耦合Cell-to-Cell interference



### 2.7.7 电荷泄露

存储在闪存存储单元的电荷，如果长期不使用，会发生电荷泄露。这同样会造成非永久性损伤，擦除后Block还能使用。

## 2.8 ROW/column 地址

Row address：行地址 Column Address：列地址

行地址表示 LUN, Block, Page的地址。列地址表示的是page内部偏移的地址。一般来说，列是由2个Bytes地址组成，而行是由3个Bytes地址组成。行地址的位宽由flash的容量决定（block可能一个字节不够表达）

什么时候发送代表lun/block/page？



\*有一些功能函数只需要发送行地址的LUN和Block即可，eg：擦除（Erase）。

**LSB/CSB/MSB**

LSB：最低有效位。MSB：最高有效位。CSB：中央有效位。

单元上CSB在TLC(3bits)更能体现出来，数据的写入遵循从低位到高位（LSB->CSB->MSB）。



**Upper page和lower page**

Upper/lower page同时绕不开LSB和MSB概念，其实这只是一个映射关系。通过cell与其中的关系就能更易懂了，cell是物理存储单元。eg：MLC的cell由两个bit组成，因此存在LSB位和MSB位，LSB对应着lower page，MSB对应着upper page。

由cell的写原理知道，先写低位再写高位，因此Upper page的编写时间长，lower page的写时间短。

Lower page编写：

1． 写1，vth不变，保持擦除状态；

2． 写0，vth升高，达到“10”或“00”所需要的电压值；

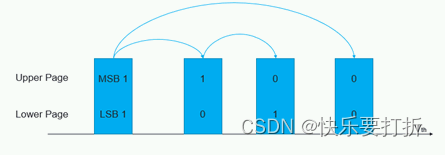
Upper page编写：

1． 写1，Vth不变，保持擦除状态；

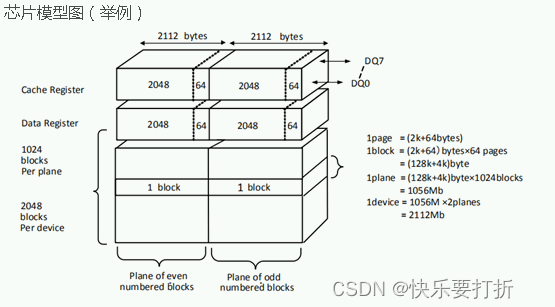
2． 写0，那么：

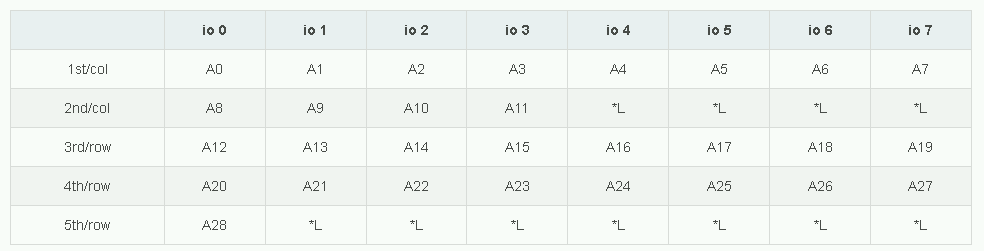
若lower page是‘1’，Vth将会升高到“01” 电压值；

若lower page是‘0’，Vth将会升高到“00” 电压值；



**内存寻址**





\*L must be set to “low”

列地址A0-A11：

页内地址，地址范围时0—4096，与页内地址（2k+64）对应，其实只需要A0-A10，而A11是用于表示oob区域（spare区）也就是图中多出64bytes。

行地址A12-A28:

表示LUN, BLOCK和PAGE地址，具体位宽根据flash容量大小设置，从数据手册得知。

\*有一些功能函数只需要发送行地址而不需要发送列地址，eg：擦除（Erase）。

**OOB/Spare Area**

每个页都有一块区域，叫空闲区域。数据在读写的时候容易出错，为了保证数据正确性，要有对应的纠正机制（ECC），所以该区域用于存放数据的校验值。

OOB的读写是随着页的操作一起完成。

用途：

标记所处的block是否为坏块。

存储ECC数据。

存储一些和文件系统相关的数据。

**Close/Open Block**

对于一个Block里面，含有许多的page，假如Block0里面有255个page，那么在写入数据的时候，如果写了255个page的话，那么就叫做Close Block，如果从page0开始写，但是没写到page255的话，那么就叫做Open Block。

## 2.9 nand flash的读取寻址过程

在NAND Flash中有8个I/O引脚（IO0—IO7）、5个全能信号（nWE ALE CLE nCE nRE）、一个引脚，1个写保护引脚。操作NAND Flash时，先传输命令，然后传输地址，最后读写数据。对于64MB的NAND Flash，需要一个26位的地址。只能8个I/O引脚充当地址、数据、命令的复用端口，所以每次传地址只能传8位。这样就需要4个地址序列。因此读写一次nand flash需要传送4次（A[7:0] A[16:9] A[24:17] A[25]）。64M的NAND Flash的地址范围为0x00000000—0x03FFFFFF。128M的NAND Flash的地址范围为0x00000000---0x07FFFFFF。1KB = 0x000-0x3FF.128字节=0x00H--7FH。

一页有528个字节，而在前512B中存放着用户的数据。在后面的16字节中（OOB）中存放着执行命令后的状态信息。主要是ECC校验的标识。列地址A0-A7可以寻址的范围是256个字节，要寻址528字节的话，将一页分为了A.（1half array）B(2 half array) C(spare array)。A区0—255字节，B区 256-511 字节C区512—527字节。访问某页时必须选定特定的区。这可以使地址指针指向特定的区实现。

在NAND Flash 中存在三类地址，分别为Block Address 、Column Address 、Page Address.。（实际就是块地址和页地址）

Column Address 用来选择是在上半页寻址还是在下半页寻址A[0]—A[7].也就相当于页内的偏移地址。在进行擦除时不需要列地址，因为擦除是以块为单位擦除。32个Page需要5bit来表示。也就是A[13:9];也就是页在块内的相对地址。A8这一位用来设置512字节的上半页，还是下半页，1表示是在上半页，而2表示是在下半页。Block的地址有A[25:14]组成.

一个容量为64M(512Mbit)的NAND Flash,分为131072页，528列。（实际中由于存在spare area,故都大于这个值），有4096块，需要12bit来表示即A[25:14].如果是128M（1Gbit）的话，blodk Address为A[26:14].由于地址只能在IO0—IO7上传送。编程时通常通过移位来实现地址的传送。传送过程如下：

第1个地址序列：传递column address，也就是NAND Flash[7:0],这一周期不需要移位即可传递到I/O[7:0]上，而half page pointer 即A8是由操作指令决定，00h，在A区，01h在B区，指令决定在哪个half page上进行读写，而真正A8的值是不需要程序员关心的；

第2个地址序列：就是将NAND\_ADDR 右移9位，而不是8位，将NAND\_ADDR[16:9]传递到I/O[7:0]上；

第3个地址序列：将NAND\_ADDR[24:17] 传递到I/O[7:0]上；

第4个地址序列：将NAND\_ADDR[25]传送到I/O上。

整个地址的传送过程需要4步才能完成。如果NAND Flash 的大小是32MB的以下的话，那么block address 最高位只到bit24,因此寻址只需要3步，就可以完成。

在进行擦除操作时由于是以块进行擦除，所以只需要3个地址序列，也就是只传递块的地址，即A[14:25]。

NAND Flash地址的计算：

Column Address 翻译过来是列地址，也就是在一页里的偏移地址。其实是指定Page上的某个Byte，指定这个Byte，其实也就是指定此页的读写起始地址。

Page Address:页地址。页的地址总是以512Bytes对齐的，所以它的低9位问题0，确定读写操作在NAND Flash中的哪个页进行。

当我们得到一个Nand Flash地址addr时，我们可以这样分解出Column Address和Page Address。

Columnaddr = addr % 512 // column address

Pageaddr = addr>>9 // page address

实际上A0~A7是页内地址，比如从第2个开始读起。不过一般都从0开始读起，呵呵。

也就是一个Nand Flash地址的A0-A7是它的column address ,A9—A25是它的Page Address，地址A8被忽略。

现在假设我要从Nand Flash中的第5000字节处开始读取1024个字节到内存的0x30000000处，我们这样调用read函数

NF\_Read(5000, 0x30000000,1024);

我们来分析5000这个src\_addr.

根据：

column\_addr=src\_addr%512;

page\_address=(src\_addr>>9);

我们可得出column\_addr=5000%512=392

page\_address=(5000>>9)=9

于是我们可以知道5000这个地址是在第9页的第392个字节处，于是我们的NF\_read函数将这样发送命令和参数

column\_addr=5000%512;

page\_address=(5000>>9);

NF\_CMD=0x01; //要从2nd half开始读取 所以要发送命令0x01

NF\_ADDR= column\_addr &0xff; //1st Cycle A[7:0]

NF\_ADDR=page\_address& 0xff

NF\_ADDR=(page\_address>>8)&0xff; //3rd.Cycle A[24:17]

NF\_ADDR=(page\_address>>16)&0xff; //4th.Cycle A[25]

向NandFlash的命令寄存器和地址寄存器发送完以上命令和参数之后,我们就可以从rNFDATA寄存器(NandFlash数据寄存器)读取数据了.

我用下面的代码进行数据的读取.

for(i=column\_addr;i<512;i++)

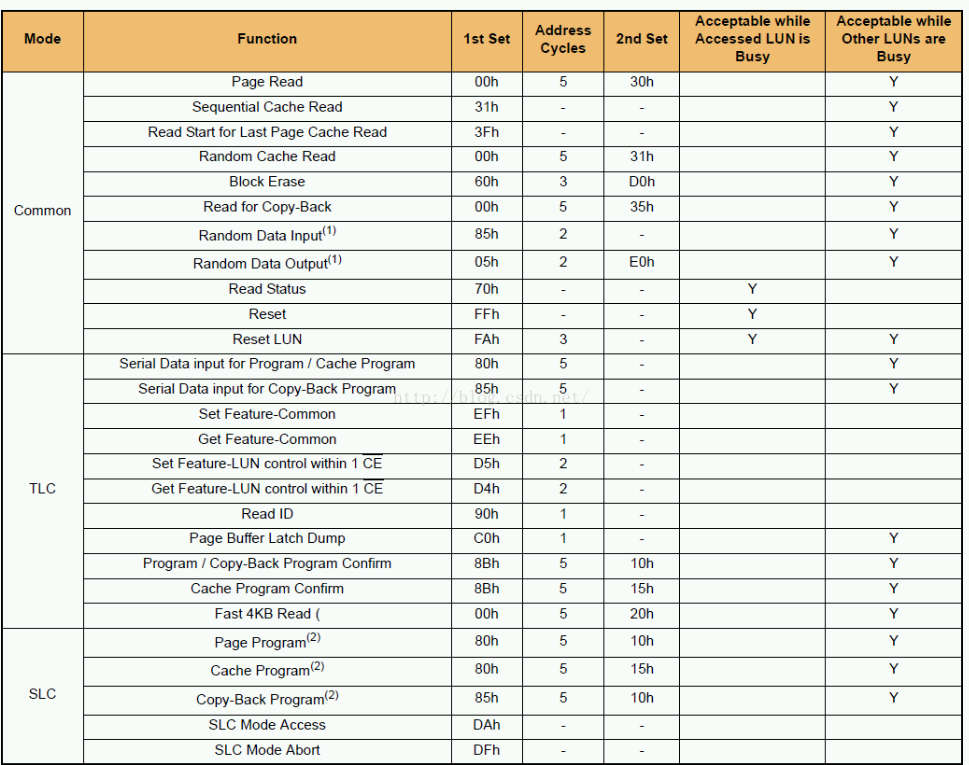
\*buf++=NF\_RDDATA();

每当读取完一个Page之后,数据指针会落在下一个Page的0号Column(0号Byte).

## 2.10 三星OBP Nand Flash的TLC编程规则

OBP Nand Flash即为三星与其他一些flash厂商共用的TLC协议，其TLC部分的编程规则与常规ONFI协议和东芝的ED3协议有着明显的区别，在这里根据K9ADGD8U0D的Datasheet，说一下OBP的TLC编程规则。

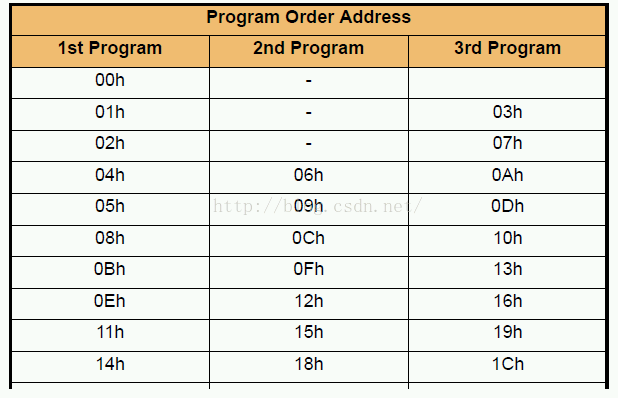
下表为该Nand Flash的Datasheet当中的指令表

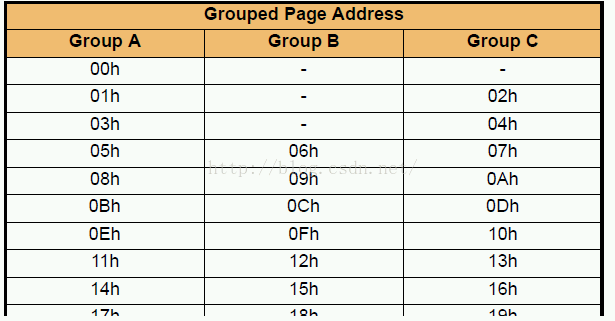


SLC操作与普通SLC和MLC操作没太大区别，就只用在开头的命令前面加一个DA表示SLC模式即可。而在所有需要TLC操作的读写命令之前，都要加一个0xDF表示进入TLC模式。

在OBP TLC模式中，与普通模式编程最大的区别首先就是编程顺序，TLC Page分为分组地址(Grouped Page Address)和编程地址(Program Order Address)。

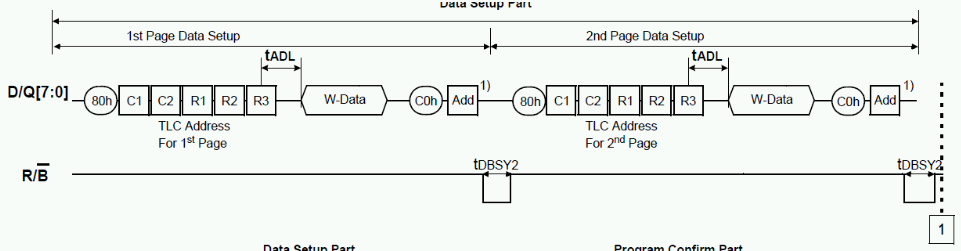
在OBP TLC模式中，与普通模式编程最大的区别首先就是编程顺序，TLC Page分为分组地址(Grouped Page Address)和编程地址(Program Order Address)。

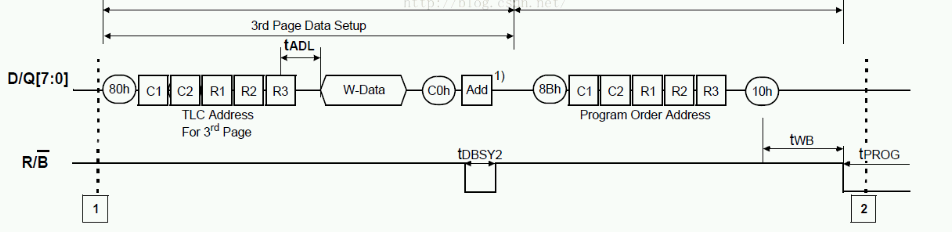




分组地址指的就是3字节行地址中的页地址，而编程地址则指示了在编程操作中需要遵循的编程顺序。下面就根据Datasheet上所列出的栗子来说明一下

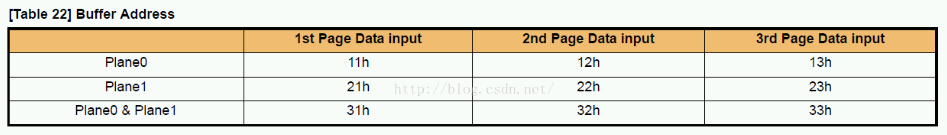
常规编程的时候，一般的命令都是先发一个0x80，后面接5个字节的行列地址，然后发送数据，最后以0x10结尾，等待RB拉高后，一个Page的数据就被编程进去了。但是在OBP规则中，通常一个Page需要被编程三次(不是所有Page都需要)之后，才能够确保数据的稳定性，保证正确的把数据读出。但是把一个页编程三次并不是指简单的重复对一个页发送三次80 10命令就可以了，这时候就需要按照上图中的编程地址顺序来逐一编程。





上图是Datasheet中所列举的栗子，但是有些地方没有说明，在这里详细的说明一下。

一开始，进入第一次页编程(1st Page Data Setup)， 此时的行地址一定是属于Group Page Address图中Group A中的某一个页的行地址，结尾的编程命令不是0x10而是0xC0，Add所代表的地址如下图所示



现在我们假设是在Plane0当中进行编程，那么此时Add所代表的命令自然就是0x11啦。接下来就进入了第二次页编程(2nd Page Data Setup)，命令与第一次类似，这里面的行地址代表Group Page Address图中Group B中的跟第一次编程中的页同一个Word Line的页的行地址，然后发送0xC0 和0x12,。第三次编程与前两次一样，行地址属于图中Group C中的第一次编程中的页同一个Word Line的页的行地址。最后是Program Confirm Part，0x8B后面所代表的地址不是刚刚任何一个页的行地址，而是指Program Order Address中的Order Address。

当初我看完这个的时候是一脸懵逼的，所以还是根据一个比较具体的实例来说明比较容易理解。

就拿一个Block中的第一个Page开始。假设是第0个Block，那么第一个页的行地址自然是0x000000，先看Group Page Address，是属于Group A。好，那么开始编程，按照GroupA的规则，发命令0x80，发两个字节列地址0x00,0x00 再发3个字节行地址0x00,0x00,0x00，发0xC0,和1st Data input的命令 0x11，然后就准备发Confirm Part命令了。

等等，上面不是说要编程了三个页之后才要发Confirm Part吗？在OBP中，不是每一个Word Line当中都有三个页，因此刚刚上面所说的是一般情况下的编程规则，在OBP当中，前几个World Line和最后几个Word Line都稍微有一点特殊，需要特殊处理一下。

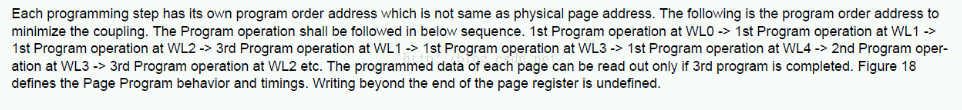
继续回到刚刚的Confirm Part命令，发送命令0x8B 然后是Order Address，此时根据Group Page Address当中Page 0对应的位置查看Program Order Address对应的值，一看，也是0，好吧，那么Order Address就是0x00,0x00,0x00,0x00,0x00 五个0，最后发送0x10，第一个Word Line就编程结束了，此时Page 0的数据可以正确读出。

继续第二个Word Line编程，根据Order Address当中的0x01的位置，对应到Group Page Address，可以看出这个页是Page 1。发送命令0x80，列地址0x00,0x00,行地址0x01,0x00,0x00，发送命令0xC0，然后是地址0x11。然后跳过了第二次编程，直接开始第三次编程，因为参照Group Page Address，第二个Word Line是没有Group B的页，就如同第一个Word Line中只有Group A的页一样。但是根据Group Page Address，此时编程的页还是Page 2，因此行地址是0x02,0x00,0x00，0xC0后面跟的是0x13。接下来又到了Confirm Part了，此时的Order Address就是0x01。那么Confirm Part的地址就是0x00,0x00,0x01,0x00,0x00，最后0x10编程结束。但是注意，由于这个Word拥有两个Page，但是此时该Word line只编程了一次，因此数据还不能正确读出。

好，到了第三个Word Line了。为啥是第三个Word Line，不应该还是第二个Word Line吗。No No No，在OBP TLC编程当中，编程的顺序要按照Order Address，所以，此时看Order Address的表，0x02所表示的Page在第三个Word Line，所以此时是编程第三个Word Line中Group A的Page，根据Group Page Address可以看到就是Page 3！开始发送命令0x80,，然后是5字节地址0x00,0x00,0x03,0x00,0x00，发完数据然后0xC0，0x11，下面是Group C的Page，0x80,0x00,0x00,0x04,0x00,0x00,0xC0,0x13，最后的Confirm Part，发送命令0x8B,Order地址0x00,0x00,0x02,0x00,0x00，最后发送0x10。

下面再次根据Order Address，看向0x03所代表的Page。呦呵，又跑回第二个Word Line去了，那么此时就重复刚刚的第二个Word Line的编程，命令0x80,地址0x00,0x00,0x01,0x00,0x00，命令0xC0跟地址0x11,。后面0x80, 地址0x00,0x00,0x02,0x00,0x00，命令0xC0跟地址0x13。但是这个是，Confirm Part中的地址不一样了，发送完0x8B后，Order地址为0x00,0x00,0x03,0x00,0x00，最后发0x10。这个真的是最后的命令，此时编程完代表这个Word line已经编程结束，Page 1和Page 2的数据已经可以正常读出了。

这几个编程做完可以发现，OBP中TLC的编程顺序是一个斜线的顺序，后面的编程按照刚刚的规律操作即可全部完成，在Datasheet中只有一段话稍微说明了一下



## 2.11大页 小页

NAND FLASH有两类：大页的NAND和小页的NAND。每种NAND的一页中都有数据段(datafield)和附加段(Spare Field)，datafield用于存放数据用的，Spare Field读写操作的时候存放校验码用的，大页的NAND中数据段2048B、附加段64B；小页的NAND中数据段512B、附加段16B。

小页的NAND是：One page = 512B(datafield) + 16B(Spare Field), One block = 32page

大页的NAND是：One page = 2048B(datafield) + 64B(Spare Field), One block = 64page or 128page

注：一块NAND FLASH中的所有页可以不按顺序烧写。

对于烧写大页的NAND有这样的要求：在一页中，数据段在2次擦除之间的编程操作不能超过四次，附加段在2次擦除之间的编程操作不能超过四次。

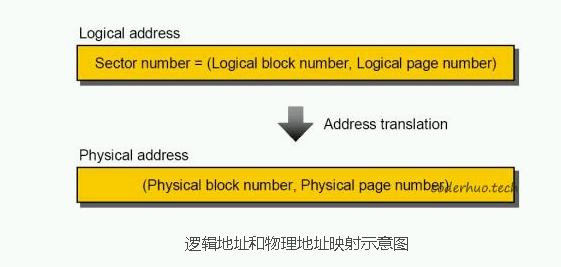
对于烧写小页的NAND有相似的要求：在一页中，数据段在2次擦除之间的编程操作不能超过两次，附加段在2次擦除之间的编程操作不能超过三次。

# 3 FTL

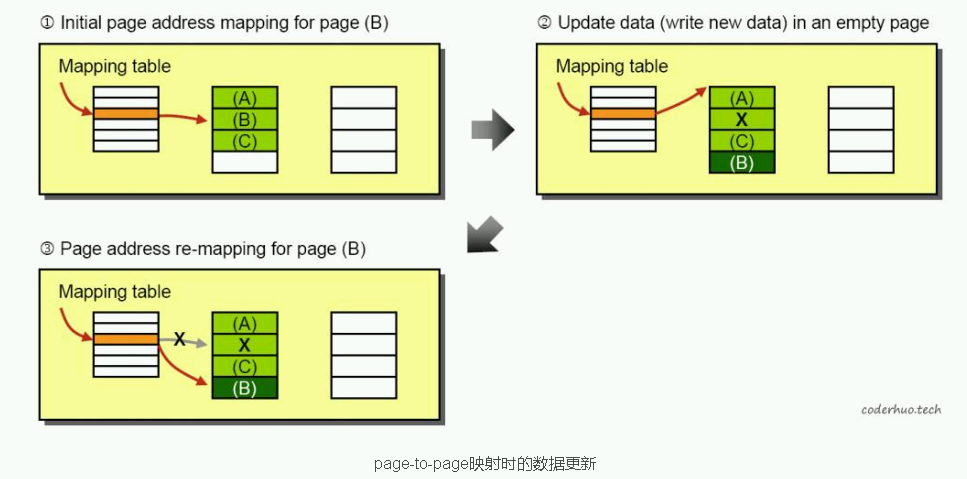
逻辑地址映射：

在Nnad flash出现以前，逻辑地址映射就已经出现了。作用是，对上层的文件系统屏蔽物理地址的细节，让寻址，简单灵活。为什么简单，灵活？

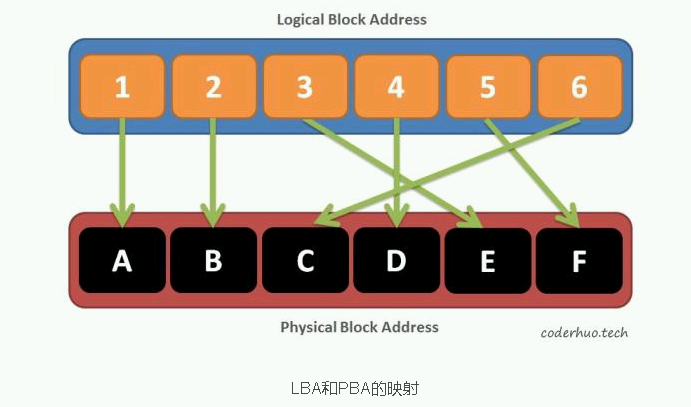
逻辑地址和物理地址的映射图：



上图所示属于page-to-page的映射，这种映射的缺点是FTL中维护大量的映射关系，好处是管理方便（某个page更新时，不用关心新数据是否和原数据位于同一个block，如下图所示）。



实际一般使用block-to-block的映射（这种情况下，逻辑page和物理page的映射是固定的，比如逻辑的page1对应物理的page1，逻辑的page2对应物理的page2），如下图所示：



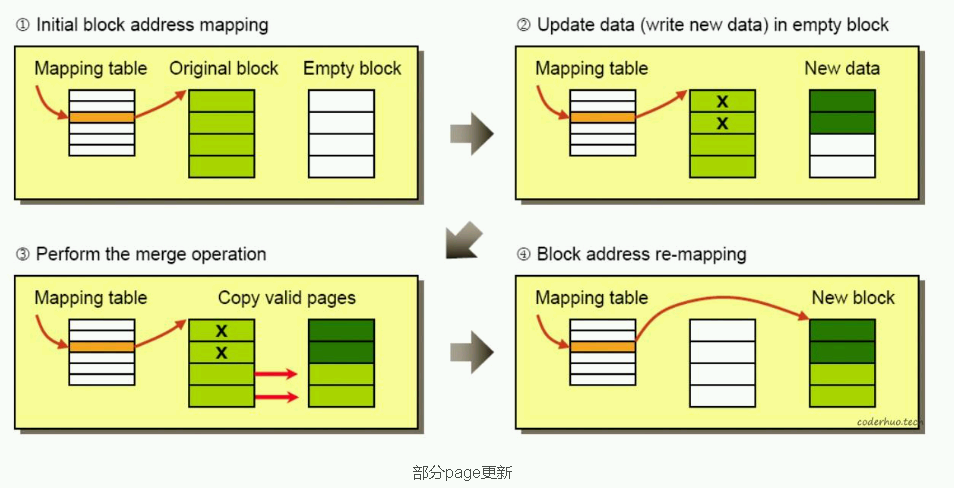
block-to-block的映射，好处是维护的映射关系较少，节省了存储空间，缺点是在数据更新的时候比较麻烦，如下图所示：

步骤1展示的是初始状态，FTL的映射指向Original block

步骤2想要更新前两个page的数据：先将新数据写入新的的block（注意：page在block中的相对位置保持不变），原来block中对应的page被标记为无效

步骤3把原来block中下面的两个page搬移到新的block中（注意：page在block中的相对位置保持不变）

步骤4更新FTL映射关系，指向新的block，然后擦除原来的block（具体什么时候擦除，由Flash内部的垃圾回收机制决定）。



block-to-block方式的映射，在数据更新的时候产生了额外的数据拷贝，需要付出的代价较高。为了解决这个问题，FAST、BAST等算法应运而生。

可以看到，无论是page-to-page的映射还是block-to-block的映射，虽然Flash内部的映射关系发生了变化，**但是该变化对文件系统是透明的，因为FTL的映射表对外并未发生变化。**

## 3.1 磨损均衡

它是FTL管理算法的一部分。

因为flash的每个blcok的擦写次数都基本一样，磨损均衡作用是将每个block的擦写次数基本保持一致，这就需要将擦写次数均摊到每个block。

如果所有block上的数据都经常更新，磨损均衡算法执行起来问题不大。如果有些block上存在冷数据（写入之后就很少更改的数据），我们必须根据一定的策略强制搬移这些数据并擦写对应的block，否则这些block就永远不会被擦除。当然这种操作会增加系统负载，同时也加大了整个系统的磨损（产生了不必要的擦写）。

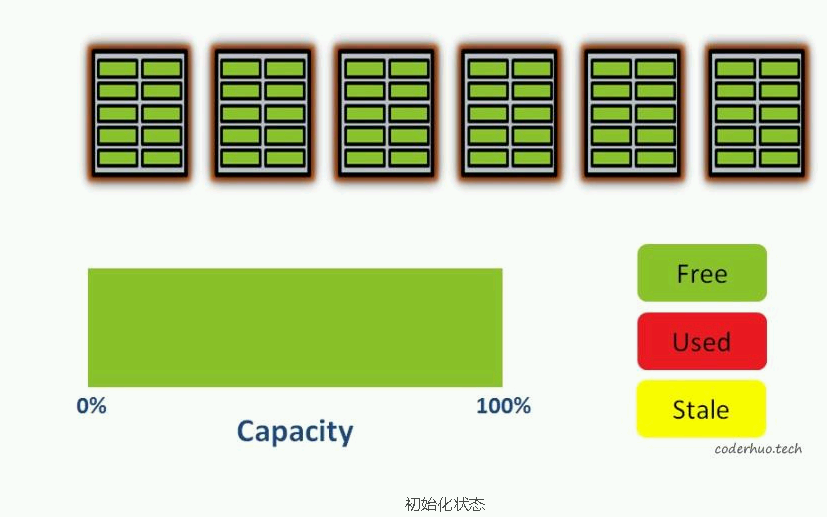
实际上，磨损均衡算法越激进，系统的磨损越严重；但是如果磨损均衡算法太消极，会导致两极分化，部分block被擦除次数较多，部分block被擦除次数较少。

## 3.2 垃圾回收

接下来我们需要处理page的回收问题（Garbage Collection）。Flash的擦除单元是block，这决定了垃圾回收的最小单元也是block。block回收过程中，需要确保待擦除block上都是无有效数据；如果有有效数据的话，需要搬移到其他的block（和磨损均衡一样，这也会增加额外的负担，实际应用中需要找到一个平衡点）。

**为什么需要垃圾回收？**

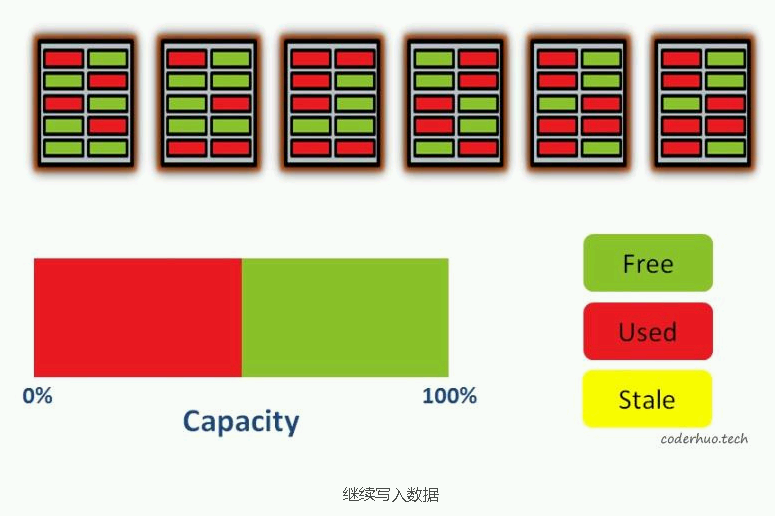
我们假设一个简单的存储介质只包含5个block，每个block包含10个page。在初始化状态，所有的page都是空的，存储介质的可用空间是100%。



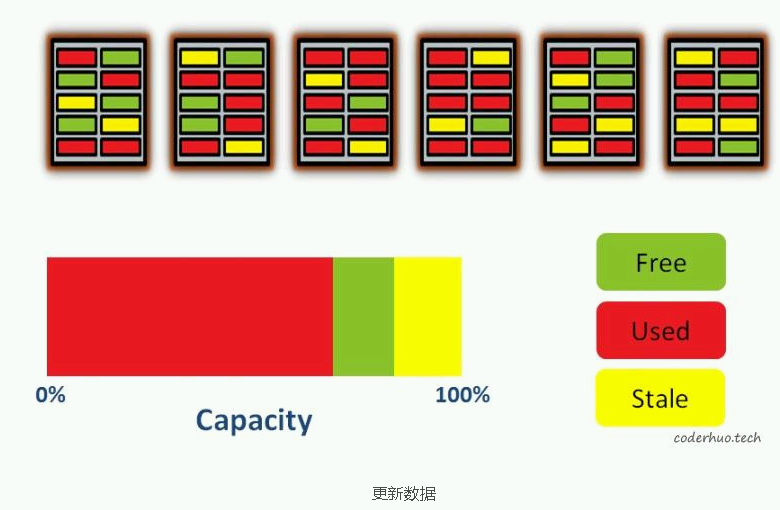
接下来写入一些数据（注意：写入的最小单元是page）。从下图可以看出，有些page已经被占用了，并且由于磨损均衡算法的作用，他们被分散在不同的block上：



我们再继续写入一些数据，现在50%的空间被占用了，并且数据分散在各个block上（尽管在物理层面数据是分散在各个block的，**FTL对外展现的可能是连续的**）：



如果这时更新数据，FTL会选择一个空的page写入新数据，然后把老的page标记为stale(旧的)状态（黄色标记块），如下图所示:



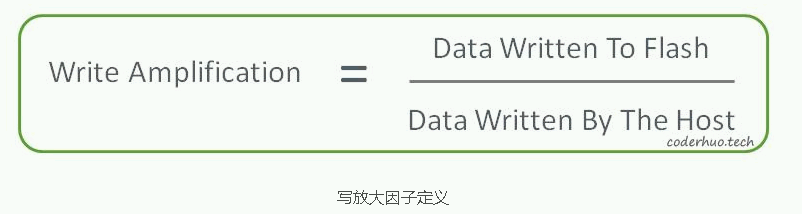
这时最左边的block包含**2个stale**状态的page和**4个used**状态的page，为了回收stale状态的page，必须先把4个used状态的page拷贝到其他的block，然后再把最左边的block整个擦除掉。如果此时不执行该操作，继续写入新数据（或者更新现有数据），会耗尽所有free状态的page，尽管此时还存在stale状态的page，但是已经无法回收了（有效数据没法腾挪了），这时候整个存储介质会进入只读状态。

所以，Flash的FTL层需要执行垃圾回收策略，释放stale状态的page。

# 4 写放大因子

从上面的介绍我们了解到，磨损均衡和垃圾回收在一定程度上都会触发后台数据搬运。这些操作是在Flash内部进行的，外部通过任何方法都监控不到，外部唯一能感受到的就是性能受到影响，比如某次写很耗时。

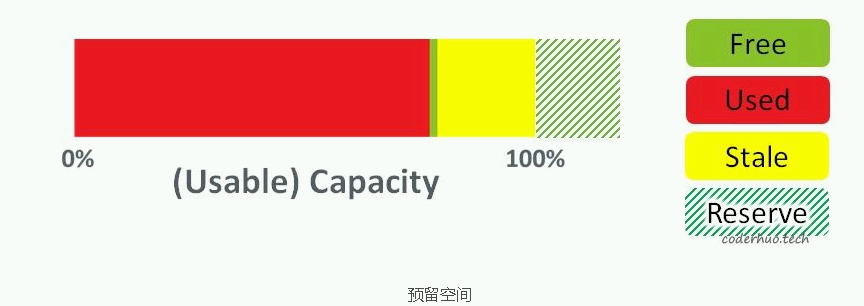
这种现象叫做写放大（Write Amplification），可以通过下面的公式衡量。该值越大说明效率越低，会对存储介质的性能和寿命造成不良影响：



公式的分子是实际写入到Flash的数据量，分母是有效数据量。比如一次写入5KB数据，但是由于磨损均衡或者垃圾回收导致后台产生了数据搬运，实际写入数据量是10KB，那么，写放大因子就是2。

# 5 预留空间

一般情况下，存储介质的实际存储空间都大于标称空间(一般多7%左右，具体依赖生产商)，多出来的存储空间被称为预留空间（Over-Provisioning），这部分空间用户是无法使用的。它可以被用来进行数据腾挪，保证垃圾回收、擦写均衡的正常进行，如果有坏块产生，还可以作为替补block顶上去（在一定程度上，让用户感知不到坏块的存在）。

6 关于解code

Nand flash在出厂时，厂商会往flash中写入参数，这些参数会影响flash的好坏

可以通过主控去调nand flash的参数，使其质量变好。