NAND ABC

Tao.Zhou

这份文档对NAND相关的设置，常见故障及debug方法都有说明，原理描述不多，偏重于实际的应用问题。碰到NAND相关问题时，请先查询本文档。居家旅行，前线作战，必备良药。带颜色的文字是碰到问题时可以先重点排查的地方。

1. NAND基本问题
   1. 总体的软硬件层次

Application(including related library)

VFS

File System

UBIFS(log, replay, TNC, commit, GC, lpt…)

UBI(leb2peb, volume, vtbl, WL, error handling…)

MTD(interface of nand accessing, partition, BBT…)

Driver(bouncing buffer, FCIE register operation…)

FCIE(HW controller, DMA, ECC…)

Hardware

NAND Flash Chip

NAND存储的层次结构如上图所示。其中VFS和UBIFS属于FS层；UBI严格来说不属于FS的范畴，它是一个专门的NAND管理软件，将一个不可靠的，会产生坏块的NAND转换为逻辑上可靠的，无坏块的理想NAND。但通常UBI和UBIFS被混为一个东西。

FCIE即我们chip的NAND Controller。早期的T3，U4用的是FCIE2，T3之后的chip用的是FCIE3。FCIE2和FCIE3相应的driver code结构也是完全不一样的，所以我们的nand driver也有独立的两套，这一点对用户而言是完全透明的。FCIE3对应的nand driver被称为UNFD（所以driver中可以看到FCIE3，UNFD相关的宏），即uniform driver，因为公司的其它产品也是使用的这套driver。

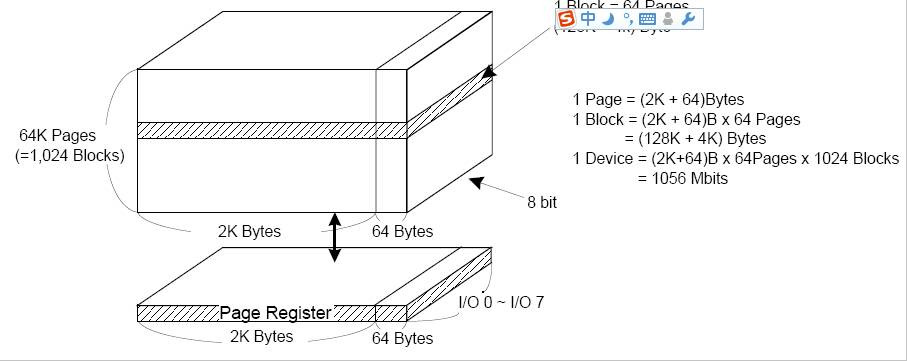
但mboot中，目前FCIE2和FCIE3还是共用一套老的driver。

* 1. NAND的物理结构

一颗NAND是由一个个的block组成的，block是erase的最小单位；一个block是由一个个的page组成的，page是read和write的最小单位。

一个page又分为main和spare区域，对于最常见的大page size NAND，main区域为2048个byte，spare区域为64个byte，所以一个page物理上有2112个byte。

一般在NAND spec的第一页会集中列出各项Features，从这里可以很快的找到page size，block size等基本参数。下面是一个典型的NAND block结构图，NAND spec上都能找到这种示意图。



main区域一般存放实际的数据，而spare区域则存放ECC（目前我们的平台上，spare区域只存放了ECC，UBI是不使用spare区域的）。由于我们一般只关注main区域的数据，很多情况下一个page的size是以2048 byte来计算的，而一个page物理上又有2112 byte，所以有时候会遇到2048和2112的折算问题。

mboot中的bininfo命令可以显示出nand的结构信息和分区的基本信息，如下图所示：

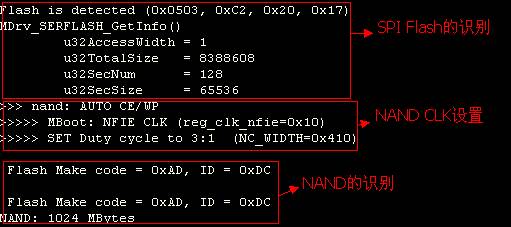


bininfo命令中对于同样一个参数都给出了三个计算结果，分别是以main区域为一个page size计算的byte数，以main+spare区域为一个page size计算的byte数，以block为单位计算的block数。三种计算结果指向的是同一个东西，只是计算单位不同罢了，就如同60秒和1分钟是一回事，虽然数值看上去不同。

理解了注释过的数据含义，其余没有加注释的数据也很容易理解了。当然，NAND的结构到底是怎样的最终还是以spec为准。

* 1. 开机过程中对NAND的识别

MBoot中对nand的识别：



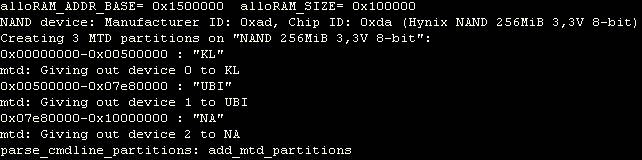
正常的mboot识别过程如图所示，图中的0xAD，0xDC就是读到的NAND的ID。

如果看到“unknown flash”这样的打印，而没有看到NAND ID，那说明NAND的识别都出了问题。这有两种可能性：

1）我们还不支持这种型号的NAND，一般简单修改driver就可以解决；

2）由于硬件问题，导致读出的ID不对。这种情况多半是NAND没有焊好，可以重新焊一颗NAND试试。

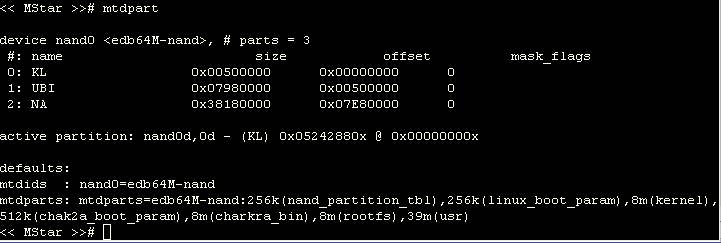
Kernel中对NAND的识别：



上图是kernel中对NAND的识别及根据mboot传入的bootargs设置mtd分区的过程。只有这些步骤通过后，才会进行UBI内部各个volume的mount。

如果mboot和kernel的运行还没有走到识别NAND的步骤就死掉了，则死机的原因多半和NAND没有太大关系。

1. mboot中nand相关的常用命令
   1. mtdpart

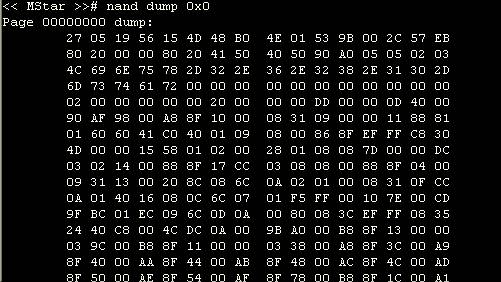


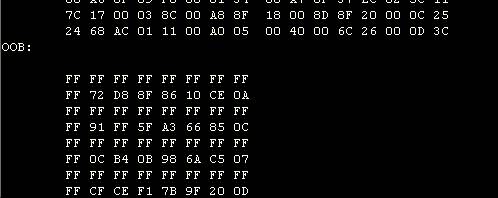
这些内容在printenv中也可以查到，只不过用mtdparts看地更直接。

可以看到，这一命令的结果告诉我们系统有三个分区，最后一个NA代表剩下没有分配的空间（比如100MB的NAND上如果分了两个区，第一个分区有20MB，第二个分区有70MB，则剩下的10MB自然就是NA），可以不用理会。因此这里实质上是KL和UBI两个分区。

offset列指明了分区的起始地址，size列则是显示分区的长度。最后的default是默认分区的设置。这里的size和offset都是以page为2048 byte来计算的，也就是说没有考虑64 byte的spare区域，如果以一个page 2112 byte来计算（(size / 2048) \* 2112），则计算出的值会大一点。

* 1. nand dump





这个命令会分两部分显示一个page的数据，前一部分是main区域的数据，后一部分是spare区域的数据（即所谓的OOB，out-of-bound，spare在MTD中被称为OOB）。

这个命令的优点是方便快速，可以很快查看到任何一个page的内容；缺点是只能一个page一个page的显示，无法一次看到整颗NAND上的数据。命令的格式为：nand dump start\_addr

start\_addr即你要查看的page起始地址，也是以page为2048 byte计算出来的。比如要查看第9个page的内容（从0开始数的第9个），则start\_addr = 2048 \* 9 = 0x4800，所以可以用nand dump 0x4800；

如果要查看第23个block（也是从0开始数）上的第11个page，则start\_addr = 23 \* 128 \* 1024 + 11 \* 2048 = 0x2e5800；

如果一个NAND有2048个block，要查看最后一个block上第一个page的内容，则start\_addr = 2047 \* 128 \* 1024 = 0xffe0000；

要查看UBI分区第一个page的内容，根据前面mtdpart的结果，start\_addr = 0x500000。

* 1. nand erase, nand scrub, nand erase clean

nand erase：擦除前会判断一个block是否为bad block，对bad block不做任何操作；

nand scrub：不会判断是否为bad block，所有block一律擦除，坏块标记为也有可能被擦掉，bbt也会被擦掉并重新扫描建立bbt。

如果发现坏块特别多，比如接近一半的block都是坏块，这种情况一般是某些原因导致的误判，要使板子恢复正常，则只能通过nand scrub将误判的bad block强行擦除。

由于连坏块也被强制擦除，因此一般只用于debug，通常情况下用nand erase就够了；

nand erase clean：在nand erase的基础上，每擦完一个block还会在spare区域写入一个CLEAN MARKER的标志，也就是这个命令中clean的含义，clean在这里并不是擦除干净的意思。clean marker是JFFS2中会用到的，UBI并不使用，这个marker会引起一些副作用，目前禁用这一擦除命令。

* 1. nand read, nand write以及nand read.e, nand write.e

带.e是表示操作时会考虑bad block的因素，不带.e则不考虑。

比如0 ~ 0x400000之间有2个bad block，那么：

nand read 0x81000000 0x0 0x400000会读取0 ~ 0x400000这个范围的NAND数据，包括bad block的内容；而：

nand read.e 0x81000000 0x0 0x400000会读取0 ~ 0x440000这个范围的NAND数据，不包括bad block的内容。

因此，通常情况下请使用带.e的nand读写命令。

* 1. nand erase addr size

如果直接使用nand erase，就执行整颗NAND的擦除，也就是我们在系统升级中常见的一种情况；如果加入addr和size参数，就可以擦除指定的block。比如：

nand erase 0x20000 0x80000，则会擦除第1 ~ 4（从0开始数）这4个block。

由于NAND擦除的最小单位是block，因此addr和size必须是block size对齐的。

* 1. nandbin

这个命令最初是用于制作NAND烧写文件的，随着mstnize工具的使用，目前这一命令主要用于debug。

由于nand dump命令只能一个page一个page的查看nand内容，如果想得到整颗NAND的数据并直接在PC上做深入分析，就只能用这个命令了。

Dump整颗NAND数据的方法：

1. 找一个**FAT32**格式（只支持fat32）的U盘（U盘使用前最好format一遍，可以加快dump速度）;
2. 手动创建一个nanddump.bin的空文件并插到板子上;
3. 在mboot中输入nandbin命令等待即可。完成后，整颗nand的数据就保存到nanddump.bin中了。

比较老的MBoot写U盘的速度很慢，需要等待很久。后来的mboot做过改进，256MB NAND dump数据大概十几分钟就可以了。

* 1. md

即显示memory的内容（memory display）。通常的用法是：

md.b addr size

这里的.b表示以字节为单位计算数据长度。这个命令和nand read.e配合使用也可以达到nand dump命令的效果，而且可以一次显示多余一个page的数据，但无法查看spare区域的数据。

1. MBoot的环境变量（ENV）

OBama上ENV是存放于NAND中的，而Supernova上ENV转移到了SPI Flash，不管存放在哪里都是有备份机制的。

ENV只在mboot启动的时候从flash上读取一次，setenv只是修改了DDR内的ENV设置，reset之后DDR中的设置就丢掉了，只有saveenv才会将DDR中的ENV写入flash中。ENV只是给MBoot使用的一个参数存放区域，kernel并不直接访问ENV。

* 1. set\_partition脚本
     1. *dynpart*

**dynpart edb64M-nand:0x400000(KL),0xF700000(UBI),-(NA)**

mtdpart命令的结果就是由这一句决定的，这一句影响的只是ENV中mtdparts分区参数的内容，如果ENV存放在spi flash中，则它对NAND上的数据不会有任何改变。

OBama平台上典型的dynpart：

**dynpart edb64M-nand:0x20000(NPT),0x100000(BL0),0x100000(BL1),0x7000000(UBI),-(NA)**

NPT，BL0和BL1是几个被废弃的partition，最初是有打算做特定用途的，但没有用起来，其实是可以去掉的。留着不仅浪费空间，而且使NAND烧写变复杂（offset不为0了）。

Supernova上的NPT和KL\_BP也是无用的废弃partition，可以去掉。

* + 1. *ubi part UBI*

nand

nand

这个命令用于mboot中的UBI分区初始化，要在mboot中对UBI/UBIFS 做任何操作之前（ubi create，ubi read，ubi write，ubifsmount等），ubi part UBI都应该确保被执行过。

这一命令执行后，会将UBI分区的范围及其在MTD上的分区编号（MTD上的分区从0开始编号，比如KL为0，UBI为1等）打印出来。开机的打印中能看到这个打印则说明是mboot自己在调用“ubi part UBI”。如果打印中还有“UBI already initialized”，说明ubi part UBI不是第一次被调用。

UBI init的时候有一个scan的动作，它会扫描UBI partition中的所有block，并在memory中记录下相关的信息。

系统要识别出NAND上的UBI volume，首先会去寻找volume table，它记录了所有的volume信息。volume table有两份完全一样的，互为备份。如果两份table都没有找到，则会报出找不到vtbl的错误，程序也不会往下进行。volume table找不到的一种可能性是ENV中的mtdpart参数不正确。

* + 1. *ubi part UBI常见错误范例*

下面的图是分别修改ENV mtdparts中UBI的分区size和分区名的结果。如果分区size或分区名不对，即使NAND上没有任何数据被破坏，系统肯定也是无法开机的。所以出现这类问题时可以首先printenv看一下mtdpart的内容是否正确。



ubi part UBI失败，虽然找到了UBI分区，但由于扫描的范围变小了，导致一些数据没有读取到而报错。



由于mtdparts中已经没有名字为UBI的分区了，因此ubi part UBI失败。

Hit any key to stop autoboot: 0

0x00220000-0x07240000 : "mtd=3"

UBI error: ubi\_read\_volume\_table: the layout volume was not found

UBI error: ubi\_init: cannot attach mtd0

UBI error: ubi\_init: UBI error: cannot initialize UBI, error -22

UBI init error -22

Error, no UBI device/partition selected!

## Booting image at 81000000 ...

Bad Magic Number

上面也是一个典型的错误打印，volume table没有找到导致ubi part UBI失败，从而导致从UBI中读取kernel失败，最终导致bootm时报bad magic number错误。

0x00000000-0x0fa00000 : "mtd=0"

UBI error: ubi\_read\_volume\_table: the layout volume was not found

UBI error: ubi\_init: cannot attach mtd0

UBI error: ubi\_init: UBI error: cannot initialize UBI, error -22

UBI init error -22

UBIFS error (pid 0): ubifs\_get\_sb: cannot open "ubi:FACTORY", error -19

Error reading superblock on volume 'ubi:FACTORY'!

ubifs\_mount FAIL !!

这也是一个类似的打印，由于UBI init的失败，自然导致后面ubifsmount FACORY的失败，从而导致开机的一些功能不正常。

ENV不正确，NAND数据损坏，NAND几乎是一颗空片子（没有写入数据的oxff状态），都有可能导致这类开机异常。碰到这类问题，首先应该检查的是printenv看到的结果是否正常，这是最容易确认的一点。

* 1. bootcmd
     1. *bootcmd的作用*

这个ENV参数有时候是放在[[kernel中设置（因为它涉及到kernel的引导），有时候又是放在set\_config中设置，这本是件无所谓的事情。

目前常见的bootcmd写法：

**bootcmd：nand read.e 81000000 KL 400000; bootm 81000000**

bootcmd的作用就是告诉mboot运行完成后，如何将控制权转交给linux kernel。如果系统不开机停在mboot命令行，可以先printenv查看一下是否有bootcmd，如果没有的话，即使NAND上没有任何的数据损坏，肯定也是开不了机的。

上面的bootcmd其实是由两条mboot命令组成的，第一条是通过nand read.e读取kernel分区到DDR的0x81000000开始的memory space，这里不要使用nand read，否则会将bad block的内容读进来，有可能导致后面的kernel解压失败；第二条则是通过bootm执行0x81000000处的程序，从这之后mboot完成使命，kernel开始运行。

如果KL是放在UBI分区内的，则bootcmd的写法类似下面这样：

bootcmd：ubi part UBI;ubi read 81000000 KL 400000; bootm 81000000



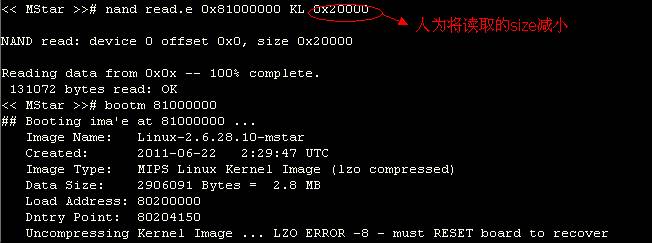
上面是一个正常开机的bootm所打印出来的内容，kernel的头部有一些基本信息，bootm命令会去读取这些信息并打印出来（也就是Image Name，Created，Image Type等）。对于uImage.lzo还有一个解压的过程，uImage则是非压缩版的kernel，启动速度会一些。

* + 1. *bootcmd相关的常见不开机情况*

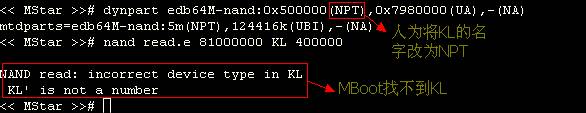
常见的不开机情况是解压失败，或是kernel的magic number不对。这种情况表明用nand read.e或是ubi read读出的kernel不对。

遇到这种情况有两个怀疑方向：一是NAND上存放的kernel真的被破坏了；二是kernel本身没有损坏，但nand read.e或ubi read读错了地方，或是读取的数据长度短了。比如使用了nand read；ENV中的分区设置不对；读取命令中的长度（比如0x400000）比kernel的实际大小要小等等，都会导致这类问题。

下面通过人为的破坏某些参数，重现了一些常见的不开机场景。



上图是人为的改小uImage.lzo的读取size，由于一个不完整的读取而导致kernel在解压的时候报错，系统有时候会因此反复reset。



再比如分区参数不对，mtdparts中没有KL分区，或是起了另外一个名字，导致nand read.e 81000000 KL 400000报错，找不到KL分区，也会导致开机失败。



uImage.lzo在编译完成后会在头部自带一个magic number，bootm如果找不到这个magic number也会报错而导致不开机。上图是人为制造了这样一个出错场景。

EEPROM @0x0 write: addr 874bfefc off 1fec count 1 ... cmd: md.b 0x874bfefc 0x1

874bfefc: 00 .

done

keypad\_pressed\_1 is:0

keypad\_pressed\_2 is:0

keypad\_pressed is [0]

<< MStar >>#

上面是一个典型的没有bootcmd而导致的不开机打印，mboot走完自己的流程后没有发现bootcmd，于是停在了命令行。一般来说ENV中没有bootcmd是ENV本身损坏而使用了默认ENV导致的（不仅仅是bootcmd的损坏），而如果默认ENV也是无法开机的设置就会出现这样的情况。

* + 1. *单独dump kernel的方法*

由于kernel是只读的数据，所以要确认其是否损坏比较容易：

1. 先通过nand read.e或是ubi read将kernel读取出来；

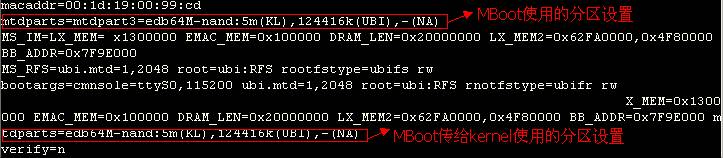
nand read.e 81000000 KL size（根据kernel的分区size设定）

或是：

ubi part UBI

ubi read 81000000 KL size（根据kernel的分区size设定）

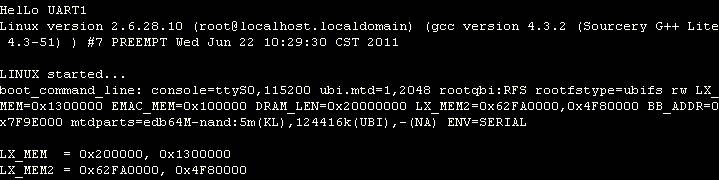
1. 找一个**FAT32**的U盘插到板子上，然后usb start，即初始化USB口（如果USB 0口不能用，则要用usb start 1）；
2. fatwrite usb 0 0x81000000 kl.bin size（根据kernel的分区size设定），即将读到DDR中的kernel再写入U盘的kl.bin文件中；
3. 将dump到kl.bin中的kernel数据与原始的uImage.lzo或是uImage进行字节比对，看数据的损坏有什么特征。
   * 1. *几个不同的分区设置及bootargs*



从上图printenv的内容可以看到两个mtdparts参数，一个是供mboot自己使用的（由dynpart生成的），而另一个是bootargs中的mtdpart，mboot自己并不使用，bootargs是mboot将要传递给kernel的参数。

setenv bootargs console=ttyS0,115200 $(MS\_RFS) $(MS\_MEM) **$(mtdparts)**

在set\_config脚本中可以看到类似上面这样的bootargs设置语句，可以看到其中的mtdparts是直接使用了mboot中的设置，因此它的源头也是dynpart。



而进入kernel之后，也能看到kernel拿到了bootargs后打印出的boot\_command\_line。mboot自己用的mtdparts，bootargs中的mtdparts以及kernel boot\_command\_line中的mtdparts必须完全一致才行，否则有可能导致不开机的故障。

一般第一次创建ENV的时候，这三者是完全一致的，但不排除某些平台会单独二次修改其中的某个mtdparts，导致不一致。这种不一致导致的现象一般是NAND上的数据并没有损坏，但同样的操作在mboot和kernel中，一个正常，另一个则失败（因为它们看到的UBI分区的范围不一样）。

mtdparts=mtdparts=edb64M-nand:256512k(UBI),-(NA)  
MS\_RFS=ubi.mtd=0,2048 root=ubi:RFS rootfstype=ubifs rw  
bootcmd=ubi part UBI;ubi read 81000000 KL 400000; bootm 81000000;  
verify=n  
filesize=A4  
MIU0\_GROUP\_PRIORITY=1:2:0:3  
MIU1\_GROUP\_PRIORITY=1:2:0:3  
MIU0\_GROUP\_SELMIU=0D80:00F8:0200:00F1  
MIU1\_GROUP\_SELMIU=4200:0E00:0030:4000  
MS\_MEM=LX\_MEM=0x4c00000 EMAC\_MEM=0x100000 DRAM\_LEN=0x10000000 LX\_MEM2=0x4A400000,0x177F000   
bootargs=ip=off console=ttyS0,115200 ubi.mtd=0,2048 root=ubi:RFS rootfstype=ubifs rw LX\_MEM=0x4c00000 EMAC\_MEM=0x100000 DRAM\_LEN=0x10000000 LX\_MEM2=0x4A400000,0x177F000 MS\_GOP0\_MIU= MS\_GOP1\_MIU= MS\_GOP2\_MIU= MS\_GOP3\_MIU= mtdparts=edb64M-nand:257408k(UBI),-(NA)

比如上面的这份ENV中，mboot自己用的mtdparts中UBI的size为256512KB，而传给kernel的bootargs中mtdparts为257408KB，这一定是有问题的。

* 1. ROOTFS

**setenv MS\_RFS ubi.mtd=3,2048 root=ubi:RFS rootfstype=ubifs rw**

MS\_RFS（有的脚本中写做ROOT）是描述rootfs的属性的，这个参数如果不正确，kernel中mount rootfs时就会失败。

ubi.mtd=3：是说ubi在mtd中的分区编号为3，比如：

dynpart edb64M-nand:0x400000(KL),0xF700000(UBI),-(NA)

它对应的就是ubi.mtd=1，KL的MTD分区编号是0。而：

dynpart edb64M-nand:0x20000(NPT),0x100000(BL0),0x100000(BL1),0x7000000(UBI),-(NA)

它对应的就是ubi.mtd=3，而NPT，BL0，BL1的MTD分区编号分别是0,1,2。

现在还有一种类似“ubi.mtd=UBI”的写法，这样写的好处是不用数UBI的分区编号，如果dynpart发生变动时，也不用修改MS\_RFS，只要rootfs始终在一个名叫UBI的分区中即可。

中间的2048表明NAND的page size是2048 byte，通常是不用修改的。但如果使用的NAND page size不为2048 byte，则要注意了。

root=ubi:RFS：即根文件系统位于ubi分区中的RFS volume。

rootfstype=ubifs rw：根文件系统的类别是ubifs，读写属性是rw，即可读可写。

如果想将rootfs设为只读，则这里要改为ro；如果想将rootfs之外的分区设为只读，则可以将rootfs/etc/init.d/rcS脚本中的相应mount命令加入-o ro选项。

* 1. filesize

filesize也是ENV中的一个参数，在tftp或是usb fatload完一个file到DDR之后，mboot会将对应的文件大小saveenv到filesize中。最新的mboot在用usb.bin文件做升级时，还会在每次解压完一个分区文件后将解压后的文件大小存入filesize。

因此，filesize并不是一个固定的值，而是随着tftp，fatload，decompress等不断变化。它使我们可以不用关心一个file具体的size，而且文件的大小也是经常变动的。

脚本中使用$(filesize)的好处（即类似ubi write 0x80400000 CONFIG $(filesize)这样的写法）：

1. 脚本可以固定下来，不用随着分区size的调整而跟着改动；
2. 通常来说，filesize是小于分区size的，就如同一个10MB的分区中通常只存放七八兆的数据。如果使用类似“ubi write 0x80400000 CONFIG 0xA00000”这样的写法，意味着尾部多写了一些垃圾数据，这是没必要的，有潜在的风险；
3. 某些大容量NAND上一些分区的size特别大（比如300MB），但烧写文件的size很小（比如30MB），如果不用$(filesize)，升级时会浪费大量的时间写无用的数据。

但是，类似“bootcmd：nand read.e 81000000 KL **400000**; bootm 81000000”不要写成“bootcmd：nand read.e 81000000 **KL $(filesize)**; bootm 81000000”，因为开机引导时，filesize中保存的不一定是uImage.lzo文件的大小。

* 1. 默认ENV的设置

ENV如果不正确往往导致不开机，code中已经做了一些保护措施，但还是很难确保ENV 100%不被破坏。从可靠性的角度看，建议客户的项目都设置可以开机的默认ENV。默认ENV是在mboot中写死的，一旦flash中的ENV损坏，mboot会启用默认ENV。

修改方法是，根据正确的ENV（printenv可以显示）修改u-boot-1.1.6/include/configs/your\_chip.h中的CONFIG\_EXTRA\_ENV\_SETTINGS宏定义，下面是一个参考的例子：

#define CONFIG\_EXTRA\_ENV\_SETTINGS \

"mtdparts=mtdparts=edb64M-nand:128k(NPT),1m(BL0),1m(BL1),112m(UBI),-(NA)\0" \

"MS\_MEM=LX\_MEM=0x2200000 EMAC\_MEM=0x100000 DRAM\_LEN=0x10000000 LX\_MEM2=0x4E404000,0x1B00000 \0" \

"MS\_GOP0\_MIU=0 \0" \

"MS\_RFS=ubi.mtd=3,2048 root=ubi:RFS rootfstype=ubifs rw \0" \

"bootargs=console=ttyS0,115200 ubi.mtd=3,2048 root=ubi:RFS rootfstype=ubifs rw LX\_MEM=0x2200000 EMAC\_MEM=0x100000 DRAM\_LEN=0x10000000 LX\_MEM2=0x4E404000,0x1B00000 MS\_GOP0\_MIU=0 mtdparts=edb64M-nand:128k(NPT),1m(BL0),1m(BL1),112m(UBI),-(NA)\0" \

"verify=n \0" \

"MIU0\_GROUP\_PRIORITY=1:2:0:3 \0" \

"MIU1\_GROUP\_PRIORITY=1:2:0:3 \0" \

"MIU0\_GROUP\_SELMIU=0D80:0088:0200:00F1 \0" \

"MIU1\_GROUP\_SELMIU=4200:0E00:0030:4000 \0" \

"filesize=a9 \0" \

"fileaddr=87549380 \0" \

""

其实没有一个通用的默认值，不同客户，不同型号和同一型号不同时期的产品，这一默认值都可能要修改才能保证开机（因为ENV不是固定的）。所以公版上是不可能提供一个万能的默认值的，它只能是在客户端设置，并且应该有人维护，确认其始终可以确保开机。

要确认默认ENV是否生效，可以通过flash (spi or nand)的write和erase命令分别将ENV写坏和擦除，再看系统是否还能开机。

ENV是有CRC校验值的，开头的4个byte即是crc的计算结果。所谓ENV损坏，mboot中的判断原则就是：根据读出的ENV重新计算的crc和ENV开头保留的crc不一致。所以，如果是ENV本身不正确导致系统开不了机，但crc并没有错误，那么默认ENV也是不会被使用的。

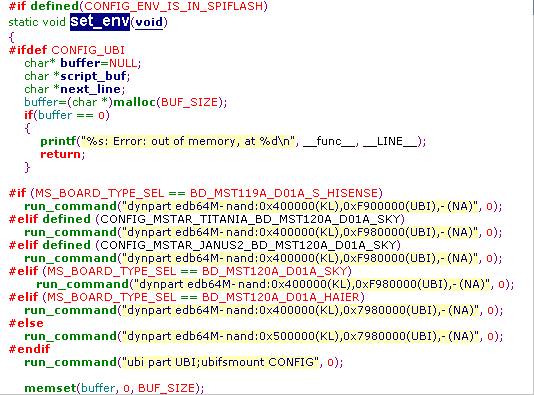
* 1. 量产烧录时ENV的处理

在OBama时代，ENV是单独作为一个烧写bin文件存在的。而到了Supernova时代，ENV转移到SPI Flash，ENV烧录的处理办法也有两种。

一是在升级好的机器上将SPI flash的内容通过ISP Tool完整的读取出来，然后发给工厂作为烧写spi flash的文件。这种方式下，spi flash不能只烧写mboot.bin，因为它不含ENV，所以这会给客户端的软件发布增加一点工作量。

第二种方式就是第一次开机自动生成ENV。要达到自动生成的目的，mboot main.c的set\_env函数中要添加你自己的partition配置，如下图所示。

这种方式的优点是工厂无需考虑ENV的烧录问题，spi flash直接烧写mboot.bin就可以了；但缺点是需要有人维护mboot的set\_env函数，分区设置发生变动时，这个函数也要做相应修改。



mboot能自动设置ENV的原因是：当mboot在spi flash中找不到ENV时（即读到了0xff）就会根据set\_env函数中的分区设置初始化UBI分区并ubifsmount CONFIG volume，然后读出其中的set\_env脚本并根据脚本自动设置ENV。而set\_env脚本是make image时自动生成的。这也是需要在set\_env函数中事先手动设置dynpart的原因。

**验证第二种方法是否有效的办法**：将ISP tool中erase的方式选择为all，重新升级一遍mboot，ENV就可以被擦除了，如果系统还能正常开机，并且开机打印中有设置ENV的动作，说明自动设置是有用的。

* 1. ENV的手动恢复

从前面的一些举例中可以看到，很多不开机只是ENV损坏了，NAND上的数据可能并没有损坏。如果恢复正确的ENV设置，系统还是有可能正常开机的。手动恢复ENV可以帮助我们判断是只有ENV损坏，还是ENV以及NAND上的数据都有损坏。

ENV一般是在正常升级的流程中自动设置的。而设置ENV的实质是一系列setenv，最后saveenv。所以手动恢复ENV就是手动执行升级脚本中和ENV相关的那些内容：

1. 执行set\_partition中的dynpart，比如dynpart edb64M-nand: 0x7000000(UBI),-(NA)；
2. mstar set\_config （ENV相关的绝大部分设置都在set\_config脚本中）；
3. mstar other related file，比如miu\_settings，里面也有ENV的相关设置。

<< MStar >># printenv

bootdelay=0

sw\_version=0.0.0

baudrate=115200

preboot=echo;echo Type "help" for more commands.

MS\_BOARD=BD\_MST087F\_D01A\_TCL\_I48

mtdids=nand0=edb64M-nand

mtdparts=mtdparts=edb64M-nand:256k(nand\_partition\_tbl),256k(linux\_boot\_param),8m

(kernel),512k(chakra\_boot\_param),8m(charkra\_bin),8m(rootfs),39m(usr)

stdin=serial

stdout=serial

stderr=serial

partition=nand0,0

Environment size: 360/131068 bytes

<< MStar >>#

上面的打印是一个典型的ENV被破坏后使用默认ENV，而默认ENV没有特别设置的场景。

1. NAND坏块及坏块表（BBT，bad block table）
   1. bad block

对于SLC，bad block的上限是2%，比如1G bit的NAND有1024个block，bad block的上限为20，而2G bit的NAND上限为40，4G bit NAND上限为80等等。

小于等于上限值的bad block数量都是属于合格的。并不是说出厂的NAND一定要有坏块才是正常情况。有些客户对NAND有深深的坏块情节，认为是NAND就一定有坏块，没有坏块肯定是有什么问题；数据损坏了，一定是因为坏块导致的（客户报问题时也常常有这类含混的描述）。这些属于误解，坏块个数和分布上的随机性的确使系统的一些设置复杂化，但真正因为坏块而导致的数据损坏其实很少见。

从经验来看，一个坏块都没有的情况是最常见的。NAND厂家除了保证第一个block出厂时不是坏块以及坏块上限外，对是否有坏块，有多少坏块以及坏块的分布规律不做任何保证。

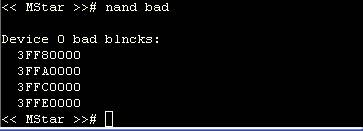
坏块分两类，出厂就无法使用的属于原始坏块，厂家也会将这类坏块标记出来。开机之后，BBT会将这些坏块记录下来，不再去操作；还有一类是使用中产生的坏块，即一个块由好变为了坏，UBI会仔细处理这种情况，在block损坏之前就会转移数据，然后在BBT中标记之，因此通常情况下，坏块并不会导致数据的损坏。

* 1. BBT

BBT是从尾部最后一个block开始倒着存放的，它实际只占用两个block（互为备份），但bbt的范围不只两个block，这是为可能的bad block做的预留。如果没有坏块，那么最后两个block就是存放bbt的。MBoot和Kernel都会访问BBT。

第一次开机的时候，NAND上可能找不到BBT（工厂烧录是不会烧写BBT的，因为每颗NAND的坏块分布都可能不一样），这时系统会扫描整颗NAND并建立BBT存入NAND中，以后开机就不会再整颗NAND扫描了。

* 1. nand bad命令



这个命令可以显示具体有多少bad block及其位置，显示的位置是这个block中第一个byte的地址。这里要注意的是，尾部的若干block是分给BBT使用的，为了确保这些block不被误操作，系统故意将其标记为bad block。因此nand bad显示的尾部地址连续的那些block要排除在bad block范围之外。比如上面的图示就表明，这颗NAND其实没有一个坏块。

1. 系统分区的设置原则
   1. KL
      1. *KL分区的位置*

老的OBama平台上KL属于UBI中的一个volume，是从属于UBI的；而在Supernova平台上，KL独立了出来，和UBI分区是平级的关系。这样做的目的是开机时mboot通过driver直接读取kernel分区，而不必透过UBI去读，开机速度上会快一点。但是也有缺点：

1. KL不在UBI分区中，失去了UBI对NAND的各种保护措施。但kernel是只读的，而且SLC的NAND可靠性比较高，所以一般来说这一点不会引起严重问题。如果是使用MLC，KL则必须放入UBI中；
2. KL单独拿出来导致KL分区要单独考虑坏块预留，浪费空间，预留不足的话量产时也有可能引起问题；
3. 额外增加了一个分区，导致NAND烧录时要用分区烧录的方式分别烧写两个分区，烧录器的设置变得复杂。烧录器设置越复杂，客户越难以掌握，也越容易出错。
   * 1. *KL分区的size设置*

由于要考虑bad block预留，因此KL分区的size肯定要大于uImage.lzo的实际大小。如果为了加快开机速度而使用非压缩的uImage，则KL分区的size还要大一些。

以2G bit NAND为例，bad block上限为40，如果使用压缩的uImage.lzo，其size通常在3MB左右，而我们的KL分区通常为4 ~ 5MB，因此预留是1 ~ 2MB，也就是8 ~ 16个block，可以看到并没有达到40个预留的绝对安全标准。

这其实是一种权衡。通常来说，坏块上限为40个的NAND其坏块数超过20个的比例非常罕见，超过10个的也不多。就算有超过10个坏块的，这10几个bad block集中在某个特定区域的概率也很小。所以虽然预留不充足，但通常不会引起问题。

* 1. NAND尾部空间的size设置

NAND尾部存放有BBT和ENV。如果是新的supernova平台，则ENV不再存放在NAND上，而转移到SPI Flash上了。但为防止以后又有将ENV搬回NAND的可能，NAND上的ENV区域还是应该预留出来。

最早的做法是给BBT区域固定分配4个block，后来改为可调的，客户可以自己修改一个宏来设定，但这样做还是有不便之处。

目前给bbt分配的block数为NAND总block数的1.2%，对于2G bit的NAND，bbt会占用2048 \* 1.2% = 24个block。ENV实际占用4个block，建议为其预留8个block。

* 1. UBI分区的size设置

当KL分区的size以及尾部空间的BBT和ENV的size确定下来之后，用NAND总容量减去这些确定区域的size，剩下的空间就是可以分配给UBI使用的上限。

假定2G bit NAND，KL有5MB，BBT有24个block，ENV有8个block，则bbt + env = 32个block，也就是4MB。那么，UBI size limit = 256MB – 5MB – 4MB = 247MB。

如果还有NPT等分区，则也要减去相应的size。UBI size是决不能超过这个上限的，从更安全的角度考虑，小一点没关系。

* 1. UBI内部各个volume的size设置

UBI中的各个分区严格来说叫做volume（卷）而不是partition（比如RFS,APP等，volume比partition低一个级别），不管UBI中有多少volume，整个UBI在MTD中都只是一个partition。

Volume的信息是由UBI管理的，和ENV无关，在NAND上有特定区域保存其信息，这是和partition不同的地方。

* + 1. *volume size的下限*

由于UBIFS的需求，一个volume（如RFS，APP等）至少需要22个block，对于2KB page size的NAND，就是将近3MB，我们通常规定这个size不小于4MB。这是对volume size下限的约束。

这个约束通常不会被违反，但某些大容量的NAND，block size有可能达到2MB，这时22个block就有44MB了，不注意的话，就有可能违背约束了。如果不满足这一点，mkfs.ubifs会报出block数不足的error。

* + 1. *UBI的预留量计算*

对于volume size上限没有规定，但是各个volume的size和应该比前面算出的UBI size要小，小的差额也就是所谓的预留量。预留量的计算公式是：

**total block num \* 2% + ubi block num \* 1% + 4 + ubi block num \* 2 / (page num of a block)**

可以看到预留量是由四个部分组成的：

1. total block num \* 2%：即block总数的2%，为出厂坏块做的预留。2G bit NAND为2048 \* 2% = 40；
2. ubi block num \* 1%：即block总数的1%，为NAND使用中可能新产生的坏块做的预留；如果ubi分区占用了2000个block，则2000 \* 1% = 20；
3. “4”：UBI用作一些特定用途，固定值；
4. ubi block num \* 2 / (page num of a block)：由于UBI将每个block的前两个page用于存放自己的内部管理数据，因此这两个page也是算入预留量的。

如果ubi分区占用了2000个block，一个block有64个page，则这一部分空间有2000 \* 2 / 64 = 63个block，以一个block为128KB来计，则约为8MB。

以上4个部分中，第1和2部分同NAND的容量成比例关系，第3部分和NAND容量无关，是固定的，第4部分和UBI分区的容量以及NAND结构都有关系。

需要注意的是，这里直接算出来的都是block数，而用户需要的是字节数，因此需要知道一个block的大小来转换一下。目前最常用NAND的block size是128KB，但这个是依NAND型号有可能变化的，以所用NAND spec为准，不能教条。

常见的1G bit，2G bit，4G bit NAND的预留量算出来为9MB，17MB，34MB，可以不用自己再去计算。对于最新的MBoot，预留量如果不足，ubi create后面的几个volume时会报出空间不足的error作为提醒。

ubi create时如果返回error -28， 就表示空间不足；如果返回error -17，则表示重复创建volume，即同一个名字的volume不能ubi create两次。如果某个volume因为空间不足而创建失败，则只能考虑在不同volume之间挪出空间或是换用更大容量的NAND。

* 1. mkfs.ubifs

我们通常不会直接用到这个命令，而是在输入make IMAGE\_ALL之后由脚本自动调用这个命令，这个命令用于将各个目录打包为.ubifs的升级文件。但我们在修改分区设置后会涉及对这个命令相关参数的修改，如果修改不当，也有可能导致make IMAGE\_ALL的过程出错。

这一命令的典型用法如下：

mkfs.ubifs -r ROOTFS -o ROOTFS.ubifs -m $(NAND\_PAGE\_SIZE) -e $(UBI\_LEB\_SIZE) -c 66

NAND\_PAGE\_SIZE：是指一个page的大小，通常为0x800，也就是2048 byte；

UBI\_LEB\_SIZE：即一个LEB（logical erase block）的size，由于UBI占用了每个block中的前两个page，因此LEB比PEB（physical erase block）要小两个page。对于目前最常见的NAND，UBI\_LEB\_SIZE = 2048 \* （64 - 2） = 126976；

-c参数的计算：用set\_partition脚本里ubi create中的size参数除以126976即可，碰到不是整除的情况，比如算出了65.31，则-c取66。

mkfs.ubifs默认使用LZO算法压缩数据，所以打包之后的.ubifs文件都是经过压缩的。

1. ECC Error

ECC error比较少见，它是从NAND driver层直接报出来的。如果有ecc error，说明存放在NAND上的数据有损坏，这种损坏一般是很严重的情况，无法修复。ECC是纯硬件计算出来的，软件完全不参与，所以ECC error一般和软件的关系不大，而板级硬件方面的怀疑比较大。

有两种常见的原因导致ecc error，一是插CI卡，二是NAND的timing不满足spec的要求。这两类原因导致的ecc error都已有修正，但如果再出现问题，还是可以往这方面怀疑。

一般来说，如果是NAND中的数据有损坏，系统重新升级之后能正常开机并不能说明什么问题。重新升级可以作为一种维修的方法，但对问题的调试没有太多帮助。而且系统升级后，原来被损坏的数据也就被彻底清除了，问题的现场不复存在。

反过来说，如果升级之后依旧开不了机，则多半是硬件有问题。

1. UBIFS中的数据损坏
   1. 文件丢失

文件丢失分两种情况，一种是个别文件的丢失；第二种是整个分区中都找不到一个文件。

对于第二种情况又有两种可能：一是FS真的有损坏，导致一个分区中所有的文件都读不出来；二是分区mount失败。

前一种情况目前还没有碰到过，mount失败倒是比较常见。所以，当发现开机后某个分区中找不到一个文件时，请先检查开机的打印，看mount是否成功，如果mount失败一定会有相关的error信息打印出来。打印信息中往往还会给出mount失败的直接原因，这是进一步分析的入口。

还有一种文件相关的error是由AP层的操作导致的。某些AP delete file和write/read file的操作都会存在，从code来看也许是先做的read/write，后做的delete。但如果因为task抢断的原因，导致实际运行时出现delete在前的情况，也会出现问题。而这种问题和FS是没有关系的。

* 1. 小文件的丢失及sync

出于性能的考虑，AP层写入文件中的数据，文件系统都不会立即将其写入Flash中。任何文件系统都不会这么做，因此对于任何文件系统而言，突然断电都会有数据丢失的风险。特别对于KB级别的小文件，往往整个文件都会被缓存起来，断电之后也容易导致整个文件的丢失，这也是最常见的文件丢失情况。

使用sync命令或是write使用SYNC模式都可以强迫FS完成write后将数据立即写入flash中。sync可以减少丢数据的概率，而无法100%避免数据丢失（如果断电的时机恰到好处）。但sync也不能使用地太滥，否则FS的效率会很低，得不偿失。一般只对小size且重要的文件考虑加sync。

另一种类似的情况是，AP先创建了一个文件，然后写入了一些数据，然后掉电。再次开机后可能发现文件还在，但文件里没有任何数据。

* 1. UBIFS的df

linux的df命令可以查看各分区空间的使用情况，但对于UBIFS，用df看到的结果是不准确的。由于UBIFS写入flash上的数据都是压缩过的，压缩率随不同的数据而变化。此外，UBIFS中的index，over space等内部数据使用的空间也无法通过固定的公式来准确计算，因此在数据写入到flash上之前，UBIFS也无法准确预估剩余的flash空间可以存放多少用户数据。

UBIFS只能在df中给出一个最保守的结果（比如假定将要写入的数据完全无法压缩），确保df中看到的剩余空间一定可用，但实际的可写入数据往往比UBIFS给出的结果要多一些，这是正常的。

此外，sync之后，由于缓存中的数据被写到flash上了，再通过df看到的结果会更准确一些。

* 1. UBIFS中的数据损坏

kernel中最常见的出错情况是ubifs报出CRC error的错误和bad node type，但没有ECC错误。表现形式可能是某个partition mount失败（最严重的情况是rootfs mount失败直接导致不开机），某个文件读取失败（这个一般影响某个功能不能正常使用，但不影响开机）。

出现这种情况说明数据的出错发生在这部分数据写入NAND之前（ECC是FCIE按已经出错的数据计算出来的，所以ECC不报错）。

如果损坏的是rootfs分区，则由于没有根目录会直接导致不开机；如果损坏的是其它分区或某个文件，则一般只是影响某些功能，有可能不会导致死机。也许出错的根源是一样的，但损坏的数据部位不同而导致现象不同。

根据之前的经验，CRC error的具体数据损坏形式是五花八门，只能具体问题具体分析。

ENV和BBT内部的地址都是绝对的，所以比较容易分析。UBI内部，文件系统是按逻辑地址来访问的，并不一定是物理的从前向后写NAND。另外，每次开机后，文件系统内部也会有各种写的动作，所以，每开一次机，NAND上的内容都会不同。如果问题机器已经开过机，则其image和原始image不一致不能说明任何问题（正常的机器也一定是不一致的）。

* 1. 常见UBIFS中数据损坏打印的初步分析

UBIFS error (pid 1): ubifs\_check\_node: bad node length 8

UBIFS error (pid 1): ubifs\_check\_node: bad node at LEB 52:36864

UBIFS error (pid 1): ubifs\_scanned\_corruption: corruption at LEB 52:36864

UBIFS error (pid 1): ubifs\_recover\_leb: LEB 52 scanning failed

VFS: Cannot open root device "ubi:RFS" or unknown-block(0,0)

Please append a correct "root=" boot option; here are the available partitions:

1f00 128 mtdblock0 (driver?)

1f01 253952 mtdblock1 (driver?)

1f02 8064 mtdblock2 (driver?)

Kernel panic - not syncing: VFS: Unable to mount root fs on unknown-block(0,0)

上面是一个典型的出错打印，最后一句kernel panic只是告诉我们rootfs mount失败，此外就没有任何有用的信息了。顺着往前找，我们可以看到几个UBIFS error。

**UBIFS error (pid 1): ubifs\_check\_node: bad node length 8**

这一句表明UBIFS读取某个node时发现这个node的length为8，很显然，UBIFS期望的length不是8.

**UBIFS error (pid 1): ubifs\_check\_node: bad node at LEB 52:36864**

这一句是最有用的信息，它直接告诉我们出错数据的位置，也就是LEB 52:36864，即损坏数据位于RFS分区的第52个逻辑块的第36864 byte开始的地方，这是进一步分析问题的入口。

至于怎么找到损坏的具体数据，一般有两种方法：

一是修改mboot，将u u-boot-1.1.6\include\ubi\_uboot.h中“CONFIG\_MTD\_UBI\_DEBUG”、“CONFIG\_MTD\_UBI\_DEBUG\_MSG”、“CONFIG\_MTD\_UBI\_DEBUG\_MSG\_BLD”三个宏前面的“#undef”改为“#define”，开机的时候，ubi part UBI就会给出大量的打印，主要是LEB和PEB的映射关系；

二是用前面提到的办法直接将整个nand内容dump出来，这是最彻底的方法，但需要时间，客户端传递几百MB的数据往往也比较吃力。不过后期的分析则会方便很多。

* 1. NAND的内容几乎为空的情况

某些不开机的板子分析下来，会发现NAND上除了极个别的地方有少量数据，整颗NAND几乎为一个空片子，即大部分区域的数据读出来都是0xFF（将这种NAND中的数据整片dump出来，压缩后会发现几百MB的dump文件变为几十KB，因为没有什么有效数据）。一般除了系统升级的时候，我们是不会做整片擦除的动作的。对于这种情况，目前已知的有两种可能性：

1）存有升级文件的U盘被无意中插到板子上，导致无意中的升级。而升级没有完成的情况下，又被拔U盘或是关机，导致问题板子保持在不完整升级的状态；

2）用烧录器烧写NAND的时候，相应的UBI分区就没有写入数据，是空的（甚至整颗NAND都没有写入数据，完全是一颗空片）。

工人机械性的工作久了也会疲劳，即使烧写器可以提示烧写失败，烧写错误的NAND也有可能被混到正常NAND中。