在 Linux 运行期间升级 Linux 系统(Uboot+kernel+Rootfs)

版本：**1.0**

作者：crifan

邮箱：green-waste (at) 163.com

版本历史

版本号	时间	内容
1.0	2011-05-03	介绍了如何实现在线升级 linux 系统，即 uboot，kernel，rootfs，以及相关的前提知识和准备工作

目录

图表

1. 正文之前

1.1. 此文目的

目前嵌入式 Linux 系统的升级，即升级 uboot，kernel，rootfs 等，的传统的方式，都是用烧写工具去烧写，相对来说，显得很繁琐和效率比较低，而利用 mtd 工具的方式去升级系统，相对比较方便。

此文主要就是介绍，在嵌入式 Linux 系统下，已经实现了 nand 和（或）nor flash 驱动后，如何利用 mtd 工具，进行实时（runtime）/在线（online）的情况下，升级 Linux 系统。

1.2. 一点说明

1. 本文所写内容，主要是之前的一些相关的工作总结，如果内容有误，请及时告知:
green-waste（at）163.com
其他技术问题的探讨，任何的问题，意见，建议等，都欢迎邮件交流。

2. 另外，如果需要的 **mtd-utils-1.3.1** 的源码的话，也可以发邮件索取。

3. 如果你当前查看的是 pdf 版本的，那么应该可以找到 pdf 中所包含的**两个附件**：
compiled_mtd-utils_arm.7z：已经编译好了的 arm 平台的，包含了 u32 和 u64 版本的，本文所用到的那 4 个 mtd 的工具，即 flash_erase，flash_eraseall，nanddump，nandwrite。
mtd-utils-1.3.1_support_u32u64.7z：我之前所用的 mtd util 的源码。你如果是其他平台的，那么用此源码，可以自己编译出对应的 mtd 的一系列的工具。关于如何编译，请参考 Readme 文件。

2. 嵌入式系统中，如何在 Linux 运行的时候去升级 Linux 系统

2.1. 前提

简单点说，在利用 mtd 工具升级系统之前，需要你的嵌入式 linux 本身具备一定条件。下面依次介绍这些前提条件。

2.1.1. Linux 中已经实现 Nor Flash 驱动

常见的嵌入式系统，都是从 nor flash 启动，然后对应的 uboot 是放在 nor flash 里面的。一般 nor flash，容量相对较小，只有 512KB 等，有的大的一点的是 1MB，2MB 之类的。一般的情况是，uboot 大约有 200 多 KB，而 linux 的 kernel 镜像文件，比如我遇到过的，大约在 1M 左右。所以，对于这些稍微大一些的 Nor Flash，往往除了放了 uboot 的代码之外，还可以放 linux 的 kernel。
如果是小的 Nor Flash，那么往往是把 kernel 放在 Nand Flash 的某个分区。

而此处用 mtd 工具升级 linux 的前提之一 ,是你 linux 系统中 ,已经实现了对应的 nand flash 的驱动。而对于 nor flash 驱动的话 ,如果还没有实现对应驱动 ,那么就先去实现对应的 nor flash 驱动。
下面这里只是对于如何实现普通的 nor flash 驱动 ,就我接触到的相关内容 ,给出一些提示。

对于常见的 spi 接口的 nor flash 来说，如果你的 nor flash 型号是常见的型号，那么很可能你不用另外单独再自己完全从头写一个完整的 nor flash 驱动了。

关于不同的接口的 Nor Flash 之间的区别，不了解的可以参考这里：
CFI Flash， JEDEC Flash ，Parellel Flash, SPI Flash， Nand Flash，Nor Flash 的区别和联系
http://hi.baidu.com/serial_story/blog/item/3f6ba1511c8b552d43a75b47.html
和
CFI (Common Flash Interface) 详解
http://hi.baidu.com/serial_story/blog/item/8d082aceb0cf8f30b700c86c.html
因为，往往你的 linux 中已经实现了 spi 驱动的，所以此时，你只需要做下面两件事情，一个是在板子相关部分，添加对应 nor flash 对应的初始化代码，二是利用 linux 默认自带的，对于常见 nor flash 都已经默认支持的 nor flash 驱动：m25p80.c
下面分别详细解释。

2.1.1.1. 在开发板相关部分添加对应 nor flash 初始化相关代码

此处，只是简单介绍一下，我之前所遇到的一个 nor flash 驱动，是如何做的。

关于添加 nor flash 初始化的代码，其实很简单，就是在开发板的最核心的那个文件：
（此处以 arm 系统为例）：
linux-2.6.28.4\arch\arm\mach-XXX\core.c
中，添加类似于这样的代码：

```
static const struct spi_board_info const XXX_spi_devices[] = {
    {                /* SSP NOR Flash chip */
     .modalias = "ssp_nor",
     .chip_select = XXX_SPI_NOR_CS,
     .max_speed_hz = 20 * 1000 * 1000,
     .bus_num = 1,
     },
。。。
};
```

然后在自己开发板设备初始化的部分，添加对应 spi nor 设备的注册函数：

```
    spi_register_board_info(XXX_spi_devices,    ARRAY_SIZE(XXX_spi_devices));
```

以实现对应的 spi 接口的 nor flash 设备的注册和添加。
具体内部逻辑是如何实现的，就要自己去看代码了。
此处只是给个框架，告诉你大概是怎么去实现的，具体的实现，肯定要你自己去看代码搞懂。

2.1.1.2. Linux 通用 nor flash 驱动 m25p80.c 简介

在 spi 接口的 nor flash 设备注册部分搞定后，再来看 Linux 中的，默认已经帮我们实现好了的一个通用的 nor flash 的驱动。
具体的文件是：
linux-2.6.28.4\drivers\mtd\devices\ m25p80.c
其中，对于支持的设备，可以去看源码中的设备列表部分的代码：

```
/* NOTE: double check command sets and memory organization when you add
 * more flash chips.  This current list focusses on newer chips, which
 * have been converging on command sets which including JEDEC ID.
 */
static struct flash_info __devinitdata m25p_data [] = {

    /* Atmel -- some are (confusingly) marketed as "DataFlash" */
    { "at25fs010",  0x1f6601, 0, 32 * 1024, 4, SECT_4K, },
    { "at25fs040",  0x1f6604, 0, 64 * 1024, 8, SECT_4K, },
```

```
{ "at25df041a", 0x1f4401, 0, 64 * 1024, 8, SECT_4K, },
{ "at25df641",   0x1f4800, 0, 64 * 1024, 128, SECT_4K, },

{ "at26f004",     0x1f0400, 0, 64 * 1024, 8, SECT_4K, },
{ "at26df081a", 0x1f4501, 0, 64 * 1024, 16, SECT_4K, },
{ "at26df161a", 0x1f4601, 0, 64 * 1024, 32, SECT_4K, },
{ "at26df321",   0x1f4701, 0, 64 * 1024, 64, SECT_4K, },

/* Spansion -- single (large) sector size only, at least
 * for the chips listed here (without boot sectors).
 */
{ "s25sl004a", 0x010212, 0, 64 * 1024, 8, },
{ "s25sl008a", 0x010213, 0, 64 * 1024, 16, },
{ "s25sl016a", 0x010214, 0, 64 * 1024, 32, },
{ "s25sl032a", 0x010215, 0, 64 * 1024, 64, },
{ "s25sl064a", 0x010216, 0, 64 * 1024, 128, },
    { "s25sl12800", 0x012018, 0x0300, 256 * 1024, 64, },
{ "s25sl12801", 0x012018, 0x0301, 64 * 1024, 256, },

/* SST -- large erase sizes are "overlays", "sectors" are 4K */
{ "sst25vf040b", 0xbf258d, 0, 64 * 1024, 8, SECT_4K, },
{ "sst25vf080b", 0xbf258e, 0, 64 * 1024, 16, SECT_4K, },
{ "sst25vf016b", 0xbf2541, 0, 64 * 1024, 32, SECT_4K, },
{ "sst25vf032b", 0xbf254a, 0, 64 * 1024, 64, SECT_4K, },

/* ST Microelectronics -- newer production may have feature updates */
{ "m25p05",   0x202010,   0, 32 * 1024, 2, },
{ "m25p10",   0x202011,   0, 32 * 1024, 4, },
{ "m25p20",   0x202012,   0, 64 * 1024, 4, },
{ "m25p40",   0x202013,   0, 64 * 1024, 8, },
{ "m25p80",             0,   0, 64 * 1024, 16, },
{ "m25p16",   0x202015,   0, 64 * 1024, 32, },
{ "m25p32",   0x202016,   0, 64 * 1024, 64, },
{ "m25p64",   0x202017,   0, 64 * 1024, 128, },
{ "m25p128", 0x202018, 0, 256 * 1024, 64, },

{ "m45pe80", 0x204014,   0, 64 * 1024, 16, },
{ "m45pe16", 0x204015,   0, 64 * 1024, 32, },

{ "m25pe80", 0x208014,   0, 64 * 1024, 16, },
```

```
    { "m25pe16", 0x208015,   0, 64 * 1024, 32, SECT_4K, },

    /* Winbond -- w25x "blocks" are 64K, "sectors" are 4KiB */
    { "w25x10", 0xef3011, 0, 64 * 1024, 2, SECT_4K, },
    { "w25x20", 0xef3012, 0, 64 * 1024, 4, SECT_4K, },
    { "w25x40", 0xef3013, 0, 64 * 1024, 8, SECT_4K, },
    { "w25x80", 0xef3014, 0, 64 * 1024, 16, SECT_4K, },
    { "w25x16", 0xef3015, 0, 64 * 1024, 32, SECT_4K, },
    { "w25x32", 0xef3016, 0, 64 * 1024, 64, SECT_4K, },
    { "w25x64", 0xef3017, 0, 64 * 1024, 128, SECT_4K, },
};
```

如果要添加此驱动，以实现支持我们的通用的 nor flash，则在 make menuconfig 的时候，添加对应设备的支持即可。

对应选项的 kconfig 的配置内容在：

linux-2.6.28.4\drivers\mtd\devices\kconfig

中：

```
config MTD_M25P80
    tristate "Support most SPI Flash chips (AT26DF, M25P, W25X, ...)"
    depends on SPI_MASTER && EXPERIMENTAL
    help
      This enables access to most modern SPI flash chips, used for
      program and data storage.  Series supported include Atmel AT26DF,
      Spansion S25SL, SST 25VF, ST M25P, and Winbond W25X.  Other chips
      are supported as well.  See the driver source for the current list,
      or to add other chips.

      Note that the original DataFlash chips (AT45 series, not AT26DF),
      need an entirely different driver.

      Set up your spi devices with the right board-specific platform data,
      if you want to specify device partitioning or to use a device which
      doesn't support the JEDEC ID instruction.
```

如上所述，如果这些步骤都做完了，最后新编译生成的 linux 内核，运行后，就应该可以可以通过：

cat /proc/mtd

查看到对应的 mtd 设备了。如果没有，那么说明你的驱动还是没有添加正常。

2.1.2. Linux 中已实现了 U 盘挂载，以方便拷贝要升级的文件

简单来说就是，你的 linux 系统中已经有了 USB 驱动，并且已经实现了 USB 的 gadget 或者 USB File storage，即实现了 U 盘的挂载。

有了 U 盘挂载，每次升级系统文件，包括 uboot，kernel 的 uImage，rootfs 等文件的话，就很方便了。

具体如何实现，不是本文所能说得清楚的，所以不再多赘述。

对于新的 Linux 内核，在已经实现了 USB device 驱动的前提下，如何实现 U 盘的功能，可以参考这个：

在 Linux USB Gadget 下使用 U 盘

http://hi.baidu.com/serial_story/blog/item/e020421e33b477fe1ad576f8.html

2.1.3. Linux 中 Nor Flash 和 Nand Flash 已能正常工作

要用 mtd 工具升级系统之前，肯定是对应的 nand flash 以及 nor flash 都是已经正常工作了。即，除了系统正常运行外，通过：

cat /proc/mtd

可以看到对应的 nor 和 nand 的 flash 所对应的分区信息了。

2.1.4. 已经准备好了 mtd 工具

此处所说的准备好了 mtd 的工具，即编译好了某个版本的 mtd-utils,比如 mtd-utils-1.3.1，然后得到对应的可执行的一系列的工具，其中这几个是用得到的：

图表 1 MTD 工具简介

MTD 工具名称	功能简介
flash_erase	擦除 flash（nand 或 nor）的某个部分
flash_eraseall	擦除整个 mtd 的分区（某个 nor 或 nand 分区）
nanddump	用于查看当前某个 mtd 分区的数据(nand 的话,也支持显示 oob 数据）
nandwrite	用于将某个文件/数据，写入到某个 mtd 分区（的某个位置）

其中，对于如何得到 mtd-util 的这些工具，有两种办法。

一种是你本身用的 buildroot 编译的整个 rootfs，这时候，可以在配置里面选择上 mtd-util 的工具，这样生成的出来的 rootfs，就有了对应的 mtd-util 的一系列工具。

另一种是，自己去 mtd 官网下载对应的 mtd-util 的源码，然后自己编译生成对应的 mtd-util 的工具。

两种方法，都很简单，只是提醒一下，编译的话，肯定是用交叉编译器，而不是 X86 的 PC 上的编辑器去编译，呵呵。

2.1.4.1. mtd-util 简介

mtd-util，即 mtd 的 utilities，是 mtd 相关的很多工具的总称，包括常用的 mtdinfo,flash_erase, flash_eraseall, nanddump, nandwrite 等，每一个工具，基本上都对应着一个同文件名的 C 文件。

mtd-util，由 mtd 官方维护更新，开发这一套工具，目的是为了 Linux 的 MTD 层提供一系列工具，方便管理维护 mtd 分区。
mtd 工具对应的源码，叫做 mtd-utils，随着时间更新，发布了很多版本。
我之前用到的版本是 mtd-utils-1.3.1，截止 2011-05-01，最新版本到了 v1.4.1。
mtd-util 源码的下载地址，请去 **MTD 源码的官网**：
http://git.infradead.org/mtd-utils.git

另外多说一句，**MTD 的官网**，资料很丰富，感兴趣的自己去看：
http://www.linux-mtd.infradead.org/index.html

【linux 的 mtd 要和 mtd-util 中的一致】
不过，对于之前的版本的 Linux 的 kernel 来说，使用 mtd-util 的话，一定要配套，主要是后来新的 linux 的版本，开始支持 mtd 的大小，即 nand 的大小，大于 4GB，对应的 linux 内核中的 mtd 层的有些变量，就必须从 u32 升级成 u64，才可以支持。
对应的 mtd 的 util 中一些变量，也是要和你当前 linux 版本的 mtd 匹配。
简单说就是，无论你用哪个版本的 Linux 内核，如果要去用 mtd-util 的话，那么两者的版本要一直，即查看 linux 内核中的 mtd 的一些头文件，主要是 include\mtd\mtd-abi.h 和你的 mtd-util 中的 include\mtd\mtd-abi.h,两个要一致。
否则，就会出现我之前遇到的问题，当然 linux 内核是 u64 版本的，支持 nand flash 大于 4GB 的，而用的 mtd-util 中的变量的定义，却还是 u32，所以肯定会出错的。
为了同一套 mtd-util 工具即支持 u32 又支持 u64，我定义了一个宏来切换，下面贴出来，供需要的人参考：
【加了宏以支持 u32 和 u64 的 mtd-abi.h 文件】
mtd-util 中的 include\mtd\mtd-abi.h：

```
/*
 * Portions of MTD ABI definition which are shared by kernel and user space
 */

#ifndef __MTD_ABI_H__
#define __MTD_ABI_H__

#include <linux/types.h>

/* from u32 to u64 to support >4GB */
#define U64_VERSION             1

struct erase_info_user {
#if U64_VERSION
```

```
	__u64 start;
	__u64 length;
#else
	__u32 start;
	__u32 length;
#endif
};

struct mtd_oob_buf {
#if U64_VERSION
	__u64 start;
#else
	__u32 start;
#endif
	__u32 length;
	unsigned char __user *ptr;
};

#define MTD_ABSENT		0
#define MTD_RAM			1
#define MTD_ROM			2
#define MTD_NORFLASH		3
#define MTD_NANDFLASH		4
#define MTD_DATAFLASH		6
#define MTD_UBIVOLUME		7

#define MTD_WRITEABLE		0x400	/* Device is writeable */
#define MTD_BIT_WRITEABLE	0x800	/* Single bits can be flipped */
#define MTD_NO_ERASE		0x1000	/* No erase necessary */
#define MTD_STUPID_LOCK		0x2000	/* Always locked after reset */

// Some common devices / combinations of capabilities
#define MTD_CAP_ROM		0
#define MTD_CAP_RAM		(MTD_WRITEABLE | MTD_BIT_WRITEABLE |
MTD_NO_ERASE)
#define MTD_CAP_NORFLASH	(MTD_WRITEABLE | MTD_BIT_WRITEABLE)
#define MTD_CAP_NANDFLASH	(MTD_WRITEABLE)

/* ECC byte placement */
#define MTD_NANDECC_OFF		0	// Switch off ECC (Not recommended)
```

```
#define MTD_NANDECC_PLACE     1     // Use the given placement in the structure
(YAFFS1 legacy mode)
#define MTD_NANDECC_AUTOPLACE   2     // Use the default placement scheme
#define MTD_NANDECC_PLACEONLY   3     // Use the given placement in the structure
(Do not store ecc result on read)
#define MTD_NANDECC_AUTOPL_USR        4     // Use the given autoplacement
scheme rather than using the default

#define MTD_MAX_OOBFREE_ENTRIES 8

/* This constant declares the max. oobsize / page, which
 * is supported now. If you add a chip with bigger oobsize/page
 * adjust this accordingly.
 */
#define MTD_NAND_MAX_PAGESIZE    8192
/*
 * for special chip, page/oob is 4K/218,
 * so here alloc more than 256+256 for 8192 pagesize for future special chip like that
 */
#define MTD_NAND_MAX_OOBSIZE          (256 + 256)

/* OTP mode selection */
#define MTD_OTP_OFF           0
#define MTD_OTP_FACTORY   1
#define MTD_OTP_USER          2

struct mtd_info_user {
      __u8 type;
      __u32 flags;
#if U64_VERSION
      __u64 size;     // Total size of the MTD
#else
      __u32 size;     // Total size of the MTD
#endif
      __u32 erasesize;
      __u32 writesize;
      __u32 oobsize;     // Amount of OOB data per block (e.g. 16)
      /* The below two fields are obsolete and broken, do not use them
       * (TODO: remove at some point) */
      __u32 ecctype;
```

```
    __u32 eccsize;
};

struct region_info_user {
#if U64_VERSION
    __u64 offset;       /* At which this region starts,
                         * from the beginning of the MTD */
#else
    __u32 offset;       /* At which this region starts,
                         * from the beginning of the MTD */
#endif
    __u32 erasesize;        /* For this region */
    __u32 numblocks;            /* Number of blocks in this region */
    __u32 regionindex;
};

struct otp_info {
    __u32 start;
    __u32 length;
    __u32 locked;
};

#define MEMGETINFO              _IOR('M', 1, struct mtd_info_user)
#define MEMERASE            _IOW('M', 2, struct erase_info_user)
#define MEMWRITEOOB             _IOWR('M', 3, struct mtd_oob_buf)
#define MEMREADOOB              _IOWR('M', 4, struct mtd_oob_buf)
#define MEMLOCK                 _IOW('M', 5, struct erase_info_user)
#define MEMUNLOCK               _IOW('M', 6, struct erase_info_user)
#define MEMGETREGIONCOUNT   _IOR('M', 7, int)
#define MEMGETREGIONINFO_IOWR('M', 8, struct region_info_user)
#define MEMSETOOBSEL            _IOW('M', 9, struct nand_oobinfo)
#define MEMGETOOBSEL            _IOR('M', 10, struct nand_oobinfo)
#define MEMGETBADBLOCK          _IOW('M', 11, __kernel_loff_t)
#define MEMSETBADBLOCK          _IOW('M', 12, __kernel_loff_t)
#define OTPSELECT           _IOR('M', 13, int)
#define OTPGETREGIONCOUNT       _IOW('M', 14, int)
#define OTPGETREGIONINFO    _IOW('M', 15, struct otp_info)
#define OTPLOCK                 _IOR('M', 16, struct otp_info)
#define ECCGETLAYOUT            _IOR('M', 17, struct nand_ecclayout)
#define ECCGETSTATS             _IOR('M', 18, struct mtd_ecc_stats)
```

```
#define MTDFILEMODE		_IO('M', 19)
/*
 * set/clear prepare oob support
 * usage:
 * 1. set prep_oob_support
 * 2. call write_oob will only prepare, not actually write
 * 3. clear prep_oob_support
 * 4. write_page will use the previously prepared oob buffer, then clear it automatically
 */
#define SETPREPAREOOB	_IOWR('M', 20, int)
#define CLEARPREPAREOOB	_IOWR('M', 21, int)

/*
 * Obsolete legacy interface. Keep it in order not to break userspace
 * interfaces
 */
struct nand_oobinfo {
	__u32 useecc;
	__u32 eccbytes;
	__u32 oobfree[MTD_MAX_OOBFREE_ENTRIES][2];
	__u32 eccpos[MTD_NAND_MAX_OOBSIZE];
};

struct nand_oobfree {
	__u32 offset;
	__u32 length;
};

/*
 * ECC layout control structure. Exported to userspace for
 * diagnosis and to allow creation of raw images
 */
struct nand_ecclayout {
	__u32 eccbytes;
	__u32 eccpos[MTD_NAND_MAX_OOBSIZE];
	__u32 oobavail;
	struct nand_oobfree oobfree[MTD_MAX_OOBFREE_ENTRIES];
};

/**
```

```
 * struct mtd_ecc_stats - error correction stats
 *
 * @corrected:    number of corrected bits
 * @failed:    number of uncorrectable errors
 * @badblocks:    number of bad blocks in this partition
 * @bbtblocks:    number of blocks reserved for bad block tables
 */
struct mtd_ecc_stats {
        __u32 corrected;
        __u32 failed;
        __u32 badblocks;
        __u32 bbtblocks;
};

/*
 * Read/write file modes for access to MTD
 */
enum mtd_file_modes {
        MTD_MODE_NORMAL = MTD_OTP_OFF,
        MTD_MODE_OTP_FACTORY = MTD_OTP_FACTORY,
        MTD_MODE_OTP_USER = MTD_OTP_USER,
        MTD_MODE_RAW,
};

#endif /* __MTD_ABI_H__ */
```

2.1.4.2. mtd 中的/dev/mtdN 与/dev/mtdblockN 的区别

简单说就是：

/dev/mtdN： 某个字符设备，对应的 mtd 的 util，就是对其操作，实现对对应的 mtd 分区进行管理的。

/dev/mtdblockN： 某个块设备，可以直接像操作其他块设备一样来操作此块设备，比如直接 cat 数据进去等等常见的操作。

更加详细的解释，请去看这个帖子：

Linux 系统中/dev/mtd 与/dev/mtdblock 的区别，即 MTD 字符设备和块设备的区别

http://hi.baidu.com/serial_story/blog/item/263bcdd3e321ebd5a9ec9a9f.html

2.2. 准备工作

2.2.1.准备好要升级的文件

将你新编译和制作出来的，要升级的文件准备好，此处为：

图表 2 要升级的 Linux 系统的文件

文件	文件名	说明
uboot 文件	u-boot.bin	只是一个普通的二进制文件
linux 的 kernel 文件	uImage	也是一个普通的二进制文件
rootfs 文件	rootfs.4k.arm.yaffs2	是用 mkyafffs2 工具制作而成，内部数据格式是 page 数据+oob 数据+page 数据+oob 数据+。。。，用于烧写到 Nand Flash 中

2.2.2.拷贝文件并挂载分区

此处，我的系统的 U 盘，是挂载在/dev/mtdblock4 中。
所以，先要通过挂载/dev/mtdblock4，即 Data 分区,作为 U 盘到 PC 上，
拷贝要升级的文件和 util 文件夹及其下面的工具：
nandwrite,flash_erase,flash_eraseall,nanddump
到 U 盘上，然后弹出 U 盘，之后将/dev/mtdblock4 挂载到/mnt/dos 下
此时，/mnt/dos 下就该有
u-boot.bin
uImage
rootfs.4k.arm.yaffs2
util/

2.3. 利用 mtd 工具升级 Linux 系统

利用 mtd 工具升级系统，其实说白了，就是：

1. 用 flasherase 擦除数据

先用 flasherase 擦除对应 mtd 分区中的内容

2. 用 nandwrite 写入数据

然后将对应的数据（uboot 或 uImage 或 rootfs）用 nandwrite 写入到对应的 mtd 中对应的位置即可。

前面介绍过了，对于常见的是把 uboot（和 kernel）放到 nor flash 中，而把 kernel 和 rootfs 放在 nand flash 中的。
而我此处的举的例子，是另外一种，即全部内容都放在 nand flash 上的。
但是，不论是是 nor flash，还是 nand flash，都在 Linux 的 MTD 框架下，管理起来，都是一样的。都是可以用对应的 mtd 的工具去操作的。所以，如果你本身是要升级对应的

uboot（和 kernel）到 nor flash，对于整个过程，也是一样的，自己照葫芦画瓢即可。

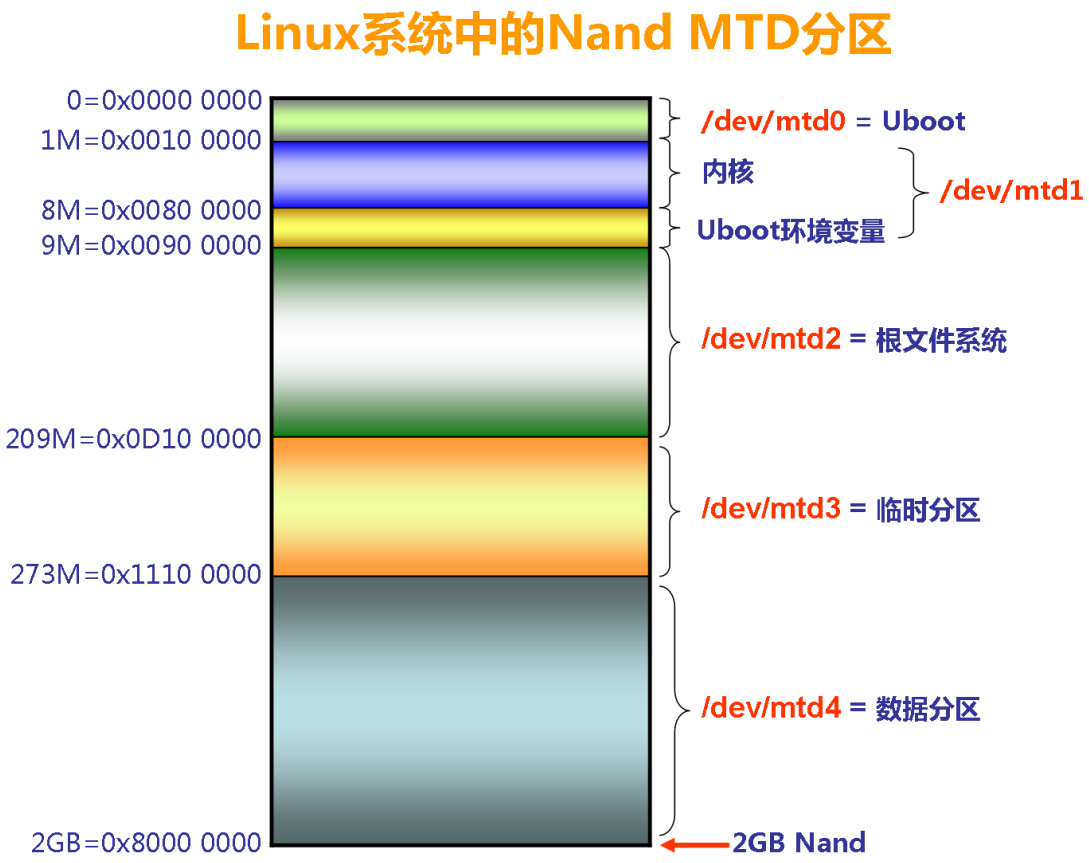
关于我此处举例所用的 MTD 的分区是如何的，此处先给出相关部分的代码：

```
#define UBOOT_SIZE        (SZ_1M)
#define KERNEL_SIZE       (SZ_8M)
#define ROOTFS_SIZE (SZ_1M*200)
#define TEMP_SIZE     (SZ_1M*64)

#define BEFORE_DATA_PARTION_SIZE                        \
          (ROOTFS_SIZE + KERNEL_SIZE + UBOOT_SIZE + TEMP_SIZE)
。。。
static struct mtd_partition XXX_default_nand_part[] = {
     [0] = {
          .name    = "U-Boot",
          .offset  = 0,
          .size    = UBOOT_SIZE,
     },
     [1] = {
          .name    = "Kernel",
          .offset  = UBOOT_SIZE,
          .size    = KERNEL_SIZE
     },
     [2] = {
          .name    = "Root filesystem",
          .offset  = UBOOT_SIZE + KERNEL_SIZE,
          .size    = ROOTFS_SIZE,
     },
     [3] = {
          .name    = "Temp",
          .offset  = UBOOT_SIZE + KERNEL_SIZE + ROOTFS_SIZE,
          .size    = TEMP_SIZE,
     },
     [4] = {
          .name    = "Data",
          .offset  = BEFORE_DATA_PARTION_SIZE,
          .size    = 0, /* set in XXX_init_nand_partion() */
     },
};
```

对应的用图表来说明，如下：

图表 3 Linux 系统中的 Nand MTD 分区



下面就来介绍，如何一步步升级 uboot，kernel 和 rootfs。

2.3.1.升级 Uboot

1. 擦除 uboot 所在分区的所有数据

./util/flash_eraseall /dev/mtd0

2.擦除旧的 uboot 的环境变量：

./util/flash_erase /dev/mtd1 0x700000 2

注：

A．0x800000~0x900000 即/dev/mtd1 中的 0x700000~0x800000，用于存放 uboot 中的环境变量。

重新升级 uboot 的同时，先把旧的环境变量擦除掉。

3.写入 uboot 数据

./util/nandwrite -p -s 0x80000 /dev/mtd0 u-boot_addHeader.bin

注：

A. -p 参数表示，如果要写入的数据不是页大小的整数倍，会自己加填充数据即，如需要，自动 padding。

B. 0x80000 是当前 4K 的 pagesize 的 nand flash 的一个块的大小。

2.3.2.升级 Kernel

1.擦除旧的 kernel 数据

./util/flash_erase /dev/mtd1 0 10

注：

A. 其中的参数 0，表示从/dev/mtd1 起始位置开始擦除。

B. 参数 10 是表示要擦除的 block 数目。

/dev/mtd1 的物理起始地址是 0x100000，而 0x100000~0x600000 之间，是用于保存 uImage 的数据，所以：

要擦除的 block 的数目

=要擦除的大小/块大小

=0x500000/块大小

=5M/512KB

=10

其中，当前用的是这个 4K pagesize 的 nand 的块大小是 512KB。

2.写入 kernel 数据

./util/nandwrite -p /dev/mtd1 uImage

2.3.3.升级 rootfs

1.擦除 rootfs 所在分区数据

./util/flash_eraseall /dev/mtd2

2.写入新的 rootfs

./util/nandwrite -o /dev/mtd2 rootfs.4k.arm.yaffs2

注:

A. 因为此处的 rootfs 镜像文件是 yaffs2 文件系统，包含了 oob 数据。所以此处加上参数 -o，意思是写入页数据同时也写入 oob 数据，而且，加了-o 参数同时就不能再像之前的 uboot 和 uImage 一样，加-p 参数了，因为包含了 oob 数据的 rootfs，本身就是页大小的整数倍，不需要 padding。

B. 不论实际使用的是 4K+128 还是对于 4K+218（内部处理为 4K+192）的 nand，此处都是使用 4K+128 的 rootfs 镜像。

2.3.4.总结

整个 runtime 的升级 linux 的过程，其实很简单。

如果说有难度的话 ,那么算是 ,在升级数据之前 ,你自己本身要清楚你原先的数据 ,即 uboot，kernel，rootfs，都是放在哪个分区的哪个位置的，然后分别擦除数据，写入新数据即可。

另外有个要注意的是，升级 rootfs 的话，尽量把其他非内核必须的进程都关闭掉，防止在升级过程中，还有进程或和程序去读取 nand flash 上的 rootfs。

此外，在烧写某个文件之后，如果希望查看当前写入的数据，是否是我们所期望的，那么可以用 nanddump 工具，将对应部分的数据“打印”出来，比如：
查看 uboot 的第一 page 的数据：
./nanddump -l 0x1000 -s 0x80000 -p /dev/mtd0

其他 mtd-util 的工具的用法，请自己参考 mtd-util 中源码的具体实现，通过看源码，可以了解其具体是如何实现，以及参数的完整的含义。