# Linux TTY framework(4)_TTY driver

作者：wowo 发布于：2016-10-25 22:40 分类：TTY子系统

## 1. 前言

本文将从驱动工程师的角度去看TTY framework：它怎么抽象、管理各个TTY设备？它提供了哪些编程接口以方便TTY driver的开发？怎么利用这些接口编写一个TTY driver？等等。

注1：话说介绍各个framework的时候，我一直比较喜欢用provider、consumer等概念，因为非常生动、易懂。不过在TTY framework的官方俗语中，压根没有provider、consumer等概念，为了不混淆试听，就算了吧。

注2：TTY framework在Linux kernel中算得上一个比较繁琐、庞杂的framework了，再加上现在很少有人会直接去写一个TTY driver，因此本文只是介绍一些概念性的东西，以加深对TTY及其driver的理解，为后续学习serial framework打基础。一些细节的东西，大家可参考callme_friend同学写的"TTY驱动分析[2]"，特别是其中的一些图示，很清晰！

注3：本文所使用的kernel版本为“X Project”初始的“Linux 4.6-rc5”版本。

## 2. 关键数据结构

注4：阅读本章内容时可对照callme_friend画的的TTY个数据结构的关系图[3]以加深理解。

### 2.1 TTY device

Linux TTY framework的核心功能，就是管理TTY设备，以方便应用程序使用。于是，问题来了，Linux kernel是怎么抽象TTY设备的呢？答案很尴尬，kernel并不认为TTY device是一个设备，这很好理解：

> 比如，我们熟悉的串口终端，串口控制器（serial controller）是一个实实在在的硬件设备，一个控制器可以支持多个串口（serial port），软件在串口上收发数据，就相当于在驱动“串口终端”。此处的TTY device，就是从串口控制器中抽象出来的一个数据通道；
>
> 再比如，我们常用的网络终端，只有以太网控制器（或者WLAN控制器）是实实在在的设备，sshd等服务进程，会基于网络socket，虚拟出来一个数据通道，软件在这个通道上收发数据，就相当于在驱动“网络终端”。

因此，从kernel的角度看，TTY device就是指那些“虚拟的数据通道”。

另外，由于TTY driver在linux kernel中出现的远比设备模型早，所以在TTY framework中，没有特殊的数据结构用于表示TTY设备。当然，为了方便，kernel从设备模型和字符设备两个角度对它进行了抽象：

1）设备模型的角度

为每个“数据通道”注册了一个stuct device，以便可以在sysfs中体现出来，例如：

```
/sys/class/tty/tty

/sys/class/tty/console

/sys/class/tty/ttyS0
```

站内搜索

请输入关键词 搜索

功能

留言板
评论列表

最新评论

luke
因公司网络限制，文章里的图片不能显示，看的好辛苦啊！！
linuxer
@hit201j：已经修改，多谢！
bigpillow
Hi Linuxer, 我们这边有写一套完整的GPIO &...
hit201j
错别字：“还以一个就是bootloader传递过来的bl...
GrayMonkey
膜拜大佬，早找到你的文章就好了,Android开发一枚，一直...
fy
@wowo：是的。我只是就这里的“禁止抢占”非必要，问问wo...

文章分类

Linux内核分析(9)
统一设备模型(13)
电源管理子系统(42)
中断子系统(14)
进程管理(13)
内核同步机制(17)
GPIO子系统(5)
时间子系统(14)
通信类协议(7)
内存管理(20)
图形子系统(1)
文件系统(3)
TTY子系统(5)
u-boot分析(3)
Linux应用技巧(13)
软件开发(6)
基础技术(13)
蓝牙(16)
ARMv8A Arch(13)
显示(3)
基础学科(9)
技术漫谈(12)
项目专区(0)
X Project(28)

随机文章

2）字符设备的角度

为每个“数据通道”注册一个struct cdev，以便在用户空间可以访问，例如：

```
/dev/tty

/dev/console

/dev/ttyS0
```

## 2.2 TTY driver

从当前设备模型的角度看，TTY framework有点奇怪，它淡化了device的概念（参考2.1的介绍），却着重突出driver。由struct tty_driver所代表的TTY driver，几乎大包大揽了TTY device有关的所有内容，如下：

```
struct tty_driver {
	int	magic;		/* magic number for this structure */
	struct kref kref;	/* Reference management */
	struct cdev **cdevs;
	struct module	*owner;
	const char	*driver_name;
	const char	*name;
	int	name_base;	/* offset of printed name */
	int	major;		/* major device number */
	int	minor_start;	/* start of minor device number */
	unsigned int	num;	/* number of devices allocated */
	short	type;		/* type of tty driver */
	short	subtype;	/* subtype of tty driver */
	struct ktermios init_termios; /* Initial termios */
	unsigned long	flags;		/* tty driver flags */
	struct proc_dir_entry *proc_entry; /* /proc fs entry */
	struct tty_driver *other; /* only used for the PTY driver */

	/*
	 * Pointer to the tty data structures
	 */
	struct tty_struct **ttys;
	struct tty_port **ports;
	struct ktermios **termios;
	void *driver_state;

	/*
	 * Driver methods
	 */

	const struct tty_operations *ops;
	struct list_head tty_drivers;
}
```

原则上来说，在编写TTY driver的时候，我们只需要定义一个struct tty_driver变量，并根据实际情况正确填充其中的字段后，注册到TTY core中，即可完成驱动的设计。当然，我们不需要关心struct tty_driver中的所有字段，下面我们捡一些重点的字段一一说明。

1）需要TTY driver关心的字段

driver_name，该TTY driver的名称，在软件内部使用；

name，该TTY driver所驱动的TTY devices的名称，会体现到sysfs以及/dev/等文件系统下；

major、minor_start，该TTY driver所驱动的TTY devices的在字符设备中的主次设备号。因为一个tty driver可以支持多个tty device，因此次设备号只指定了一个start number；

num，该driver所驱动的tty device的个数，可以在tty driver注册的时候指定，也可以让TTY core自行维护，具体由TTY_DRIVER_DYNAMIC_DEV flag决定（可参考“”中的介绍）；



**文章存档**

> type、subtype，TTY driver的类型，具体可参考“include/linux/tty_driver.h”中的定义；
>
> init_termios，初始的termios，可参考2.5小节的介绍；
>
> flags，可参考2.6小节的介绍；
>
> ops，tty driver的操作函数集，可参考2.7小节的介绍；
>
> driver_state，可存放tty driver的私有数据。

2）内部使用的字段

> ttys，一个struct tty_struct类型的指针数组，可参考2.3小节的介绍；
>
> ports，一个struct tty_port类型的指针数组，可参考2.4小节的介绍；
>
> termios，一个struct ktermios类型的指针数组，可参考2.5小节的介绍。

**2.3 TTY struct(struct tty_struct)**

TTY struct是TTY设备在TTY core中的内部表示。

从TTY driver的角度看，它和文件句柄的功能类似，用于指代某个TTY设备。

从TTY core的角度看，它是一个比较复杂的数据结构，保存了TTY设备生命周期中的很多中间变量，如：

> dev，该设备的struct device指针；
>
> driver，该设备的struct tty_driver指针；
>
> ops，该设备的tty操作函数集指针；
>
> index，该设备的编号（如tty0、tty1中的0、1）；
>
> 一些用于同步操作的mutex锁、spinlock锁、读写信号量等；
>
> 一些等待队列；
>
> write buffer有关的信息；
>
> port，该设备对应的struct tty_port（可参考2.4小节的介绍）
>
> 等等。

由于编写TTY driver的时候不需要特别关心struct tty_struct的内部细节，这里不再详细介绍。

**2.4 TTY port(struct tty_port)**

在TTY framework中TTY port是一个比较难理解的概念，因为它和TTY struct类似，也是TTY device的一种抽象。那么，既然有了TTY struct，为什么还需要TTY port呢？先看一下kernel代码注释的解释：

```
/* include/linux/tty.h */

/*
* Port level information. Each device keeps its own port level information
* so provide a common structure for those ports wanting to use common support
* routines.
*
* The tty port has a different lifetime to the tty so must be kept apart.
* In addition be careful as tty -> port mappings are valid for the life
* of the tty object but in many cases port -> tty mappings are valid only
* until a hangup so don't use the wrong path.
*/
```

我的理解是：

> TTY struct是TTY设备的“动态抽象”，保存了TTY设备访问过程中的一些临时信息，这些信息是有生命周期的：从打开TTY设备开始，到关闭TTY设备结束；
>
> TTY port是TTY设备固有属性的“静态抽象”，保存了该设备的一些固定不变的属性值，例如是否是一个控制台设备（console）、打开关闭时是否需要一些delay操作、等等；
>
> 另外（这一点很重要），TTY core负责的是逻辑上的抽象，并不关心这些固有属性。因此从层次上看，这些属性完全可以由具体的TTY driver自行维护；
>
> 不过，由于不同TTY设备的属性有很多共性，如果每个TTY driver都维护一个私有的数据结构，将带来代码的冗余。所以TTY framework就将这些共同的属性抽象出来，保存在struct tty_port数据结构中，同时提供一些通用的操作接口，供具体的TTY driver使用；
>
> 因此，总结来说：TTY struct是TTY core的一个数据结构，由TTY core提供并使用，必要的时候可以借给具体的TTY driver使用；TTY port是TTY driver的一个数据结构，由TTY core提供，由具体的TTY driver使用，TTY core完全不关心。

**2.5 termios(struct ktermios)**

说实话，在Unix/Linux的世界中，终端（terminal）编程是一个非常繁琐的事情，为了改善这种状态，特意制订了符合POSIX规范的应用程序编程接口，称作POSIX terminal interface[3]。POSIX terminal interface操作的对象，就是名称为termios的数据结构（在用户空间为struct termios，内核空间为struct ktermios）。以kernel中的struct ktermios为例，其定义如下：

```
/* include/uapi/asm-generic/termbits.h */

struct ktermios {
	tcflag_t c_iflag;		/* input mode flags */
	tcflag_t c_oflag;		/* output mode flags */
	tcflag_t c_cflag;		/* control mode flags */
	tcflag_t c_lflag;		/* local mode flags */
	cc_t c_line;			/* line discipline */
	cc_t c_cc[NCCS];		/* control characters */
	speed_t c_ispeed;		/* input speed */
	speed_t c_ospeed;		/* output speed */
};
```

说实话，要理解上面的数据结构，真的不是一件容易的事情，这里只能简单介绍一些我们常用的内容，更为具体的，如有需要，后面会用单独的文章去分析：

> c_cflag，可以控制TTY设备的一些特性，例如data bits、parity type、stop bit、flow control等（例如串口设备中经常提到的8N1）；
>
> c_ispeed、c_ospeed，可以分别控制TTY设备输入和输出的速度（例如串口设备中的波特率）；
>
> 其它，暂不介绍。

**2.6 tty driver flags**

TTY driver在注册struct tty_driver变量的时候，可以提供一些flags，以告知TTY core一些额外的信息，例如（具体可参考include/linux/tty_driver.h中的定义和注释，写的很清楚）：

> TTY_DRIVER_DYNAMIC_DEV：如果设置了该flag，则表示TTY driver会在需要的时候，自行调用tty_register_device接口注册TTY设备（相应地回体现在字符设备以及sysfs中）；如果没有设置，TTY core会在tty_register_driver时根据driver->num信息，自行创建对应的TTY设备。

**2.7 TTY操作函数集**

TTY core将和硬件有关的操作，抽象、封装出来，形成名称为struct tty_operations的数据结构，具体的TTY driver不需要关心具体的业务逻辑，只需要根据实际的硬件情况，实现这些操作接口即可。

听着还挺不错啊（framework的中心思想一贯如此，大家牢记即可），不过不能高兴的太早，看过这个数据结构之后，估计心会凉半截：

```
struct tty_operations {
	struct tty_struct * (*lookup)(struct tty_driver *driver,
			struct inode *inode, int idx);
	int  (*install)(struct tty_driver *driver, struct tty_struct *tty);
	void (*remove)(struct tty_driver *driver, struct tty_struct *tty);
	int  (*open)(struct tty_struct * tty, struct file * filp);
	void (*close)(struct tty_struct * tty, struct file * filp);
	void (*shutdown)(struct tty_struct *tty);
	void (*cleanup)(struct tty_struct *tty);
	int  (*write)(struct tty_struct * tty,
		      const unsigned char *buf, int count);
	int  (*put_char)(struct tty_struct *tty, unsigned char ch);
	void (*flush_chars)(struct tty_struct *tty);
	int  (*write_room)(struct tty_struct *tty);
	int  (*chars_in_buffer)(struct tty_struct *tty);
	int  (*ioctl)(struct tty_struct *tty,
		    unsigned int cmd, unsigned long arg);
	long (*compat_ioctl)(struct tty_struct *tty,
				unsigned int cmd, unsigned long arg);
	void (*set_termios)(struct tty_struct *tty, struct ktermios * old);
	void (*throttle)(struct tty_struct * tty);
	void (*unthrottle)(struct tty_struct * tty);
	void (*stop)(struct tty_struct *tty);
	void (*start)(struct tty_struct *tty);
	void (*hangup)(struct tty_struct *tty);
	int (*break_ctl)(struct tty_struct *tty, int state);
	void (*flush_buffer)(struct tty_struct *tty);
	void (*set_ldisc)(struct tty_struct *tty);
	void (*wait_until_sent)(struct tty_struct *tty, int timeout);
	void (*send_xchar)(struct tty_struct *tty, char ch);
	int (*tiocmget)(struct tty_struct *tty);
	int (*tiocmset)(struct tty_struct *tty,
			unsigned int set, unsigned int clear);

	int (*resize)(struct tty_struct *tty, struct winsize *ws);
	int (*set_termiox)(struct tty_struct *tty, struct termiox *tnew);
	int (*get_icount)(struct tty_struct *tty,
				struct serial_icounter_struct *icount);
#ifdef CONFIG_CONSOLE_POLL
	int (*poll_init)(struct tty_driver *driver, int line, char *options);
	int (*poll_get_char)(struct tty_driver *driver, int line);
	void (*poll_put_char)(struct tty_driver *driver, int line, char ch);
#endif
	const struct file_operations *proc_fops;
};
```

这也太复杂了吧？确实如此，好在终端设备正在慢慢的退出历史舞台，看这篇文章的同学们，可能这一辈子都不会直接去写一个TTY driver。所以，这里我也不打算详细介绍，仅仅出于理解TTY framework的目的，做如下说明：

> 这些操作函数的操作对象，基本上都是struct tty_struct类型的指针，这也印证了我们在2.3中所说的----TTY struct是TTY设备的操作句柄；
>
> 当然，具体的TTY driver，可以从struct tty_struct指针中获取足够多的有关该TTY设备的信息，例如TTY port等；
>
> TTY core会通过".write"接口，将输出信息送给终端设备并显示。因此具体的TTY driver需要实现该接口，并通过硬件操作将数据送出；
>
> 你一定会好奇：既然有".write"接口，为什么没有相应的read接口？TTY设备上的输入信息，怎么经由TTY core送给Application呢？具体可以参考"TTY驱动分析[2]"，本文不再详细介绍了。

## 3. 提供的用于编写TTY driver的API

在提供了一系列的数据结构的同时，TTY framework向下封装了一些API，以方便TTY driver的开发，具体如下。

#### 3.1 TTY driver有关的API

用于struct tty_driver数据结构的分配、初始化、注册等：

```
/* include/linux/tty_driver.h */

extern struct tty_driver *__tty_alloc_driver(unsigned int lines,
                struct module *owner, unsigned long flags);
extern void put_tty_driver(struct tty_driver *driver);
extern void tty_set_operations(struct tty_driver *driver,
                        const struct tty_operations *op);
extern struct tty_driver *tty_find_polling_driver(char *name, int *line);

extern void tty_driver_kref_put(struct tty_driver *driver);

/* Use TTY_DRIVER_* flags below */
#define tty_alloc_driver(lines, flags) \
                __tty_alloc_driver(lines, THIS_MODULE, flags)
```

```
/* include/linux/tty.h */

extern int tty_register_driver(struct tty_driver *driver);
extern int tty_unregister_driver(struct tty_driver *driver);
```

tty_alloc_driver，分配一个struct tty_driver指针，并初始化那些不需要driver关心的字段：

lines，指明该driver最多能支持多少个设备，TTY core会根据该参数，分配相应个数的ttys、ports、termios数组；

flags，请参考2.6小节的说明。

tty_set_operations，设置TTY操作函数集。

tty_register_driver，将TTY driver注册给TTY core。

#### 3.2 TTY device有关的API

如果TTY driver设置了TTY_DRIVER_DYNAMIC_DEV flag，就需要自行注册TTY device，相应的API包括：

```
/* include/linux/tty.h */

extern struct device *tty_register_device(struct tty_driver *driver,
                                unsigned index, struct device *dev);
extern struct device *tty_register_device_attr(struct tty_driver *driver,
                                unsigned index, struct device *device,
                                void *drvdata,
                                const struct attribute_group **attr_grp);
extern void tty_unregister_device(struct tty_driver *driver, unsigned index);
```

tty_register_device，分配并注册一个TTY device，最后将新分配的设备指针返回给调用者：

driver，对应的TTY driver；

> index，该TTY设备的编号，它会决定该设备在字符设备中的设备号，以及相应的设备名称，例如/dev/ttyS0中的‘0’；
>
> dev，可选的父设备指针。

tty_register_device_attr，和tty_register_device类似，只不过可以额外指定设备的attribute。

#### 3.3 数据传输有关的API

当TTY core有数据需要发送给TTY设备时，会调用TTY driver提供的.write或者.put_char回调函数，TTY driver在这些回调函数中操作硬件即可。

当TTY driver从TTY设备收到数据并需要转交给TTY core的时候，需要调用TTY buffer有关的接口，将数据保存在缓冲区中，并等待Application读取，相关的API有：

```
/* include/linux/tty_flip.h */

static inline int tty_insert_flip_char(struct tty_port *port,
                                       unsigned char ch, char flag)
{
        ....
}

static inline int tty_insert_flip_string(struct tty_port *port,
                const unsigned char *chars, size_t size)
{
        return tty_insert_flip_string_fixed_flag(port, chars, TTY_NORMAL, size);
}
```

注5：本文没有涉及TTY buffer、TTY flip有关的知识，大家知道有这么回事就行了。

## 4. TTY driver的编写步骤

说实话，TTY driver的编写步骤，就和“把大象放进冰箱”一样“简单”：

> 步骤1：实现TTY设备有关的操作函数集，并保存在一个struct tty_operations变量中。
>
> 步骤2：调用tty_alloc_driver分配一个TTY driver，并根据实际情况，设置driver中的字段（包括步骤1中的struct tty_operations变量）。
>
> 步骤3：调用tty_register_driver将driver注册到kernel。
>
> 步骤4：如果需要动态注册TTY设备，在合适的时机，调用tty_register_device或者tty_register_device_attr，向kernel注册TTY设备。
>
> 步骤5：接收到数据时，调用tty_insert_flip_string或者tty_insert_flip_char将数据交给TTY core；TTY core需要发送数据时，会调用driver提供的回调函数，在那里面访问硬件送出数据即可。

好吧，相信任何人看到上面的步骤之后，都没办法去写一个完整的TTY driver（我也是）。好在这是一个幸福的时代，我们很少需要直接去写这样的一个driver。那么本文的目的又是什么呢？权当是一个科普吧，如果你真的打算去写一个TTY driver，不至于无从下手。

## 5. 参考文档

[1] Linux TTY framework(2)_软件架构

[2] TTY驱动分析（链接失效，TODO）

[3] TTY各数据结构关系图，http://www.wowotech.net/content/uploadfile/201505/eb451430746679.gif（（链接失效，TODO））

[4] POSIX terminal interface，https://en.wikipedia.org/wiki/POSIX_terminal_interface#CITEREFZlotnick1991

[5] Linux设备模型(4)_sysfs

*原创文章，转发请注明出处。蜗窝科技，www.wowotech.net。*

**标签: Linux Kernel 内核 driver tty tty_struct tty_port**


« 进程管理和终端驱动：基本概念 | DRAM 原理 5 ：DRAM Devices Organization»

**评论：**

**magicse7en**
2017-06-19 13:50

参考文档 [2][3] 的链接都已经失效

回复

**wowo**
2017-06-19 17:11

@magicse7en：多谢提醒，我处理一下。

回复

**hit20j**
2017-09-22 23:30

@wowo：请问 TTY驱动分析 链接在哪里？

回复

**wowo**
2017-09-23 10:24

@hit20j：没有链接了，被原作者删除了:-)

回复

**numbqq**
2017-06-15 16:07

Hi wowo，

看了你的系列文章，受益匪浅，非常感谢！我最近调试终端碰到点问题想请教一下。忘不吝赐教。我现在使用HDMI输出，配置framebuffer console使信息输出到显示器，但是输出显示碰到一个问题就是显示器字符刷新慢，我通过键盘输入字符，字符不会马上在屏幕上显示完整，而是慢慢才会显示完整，感觉刷新很慢。通过
echo kkkkkkk > /dev/tty1，kkkkkkk也是慢慢才显示完整，请问你知道这会是哪方面的问题吗？非常感谢！

回复

**wowo**
2017-06-15 18:27

@numbqq：fb console每画一个字符，都是把这个字符，和屏幕上已有的内容一起，转成一副完整的图片，然后写入到buffer中。
因此，基于当前的分辨率、刷新率、fb的位数，你可以算算显示一个字符的时间（估计快不到哪里去）。

回复

**numbqq**
2017-06-15 18:56

@wowo：谢谢这么快回复！
正常来说从键盘输入字符到显示器显示出这个字符，时间应该是一致的，至少肉眼看不出时间差，但是现在我碰到的问题是键盘输入一个字符后，屏幕不会马上显示出这个完整的字符，而是会显示字符其中一部分像素点，在慢慢显示出完整的字符，肉眼明显能感受到显示字符不对。以前没有调过这方面的，不知道问题到底出在哪，能不能给一些建议呢？谢谢！

回复

**wowo**
2017-06-19 08:43

@numbqq：出现这种情况，是因为fbcon在一个像素点一个像素点地画这个字符的时候，你的现实硬件还在不停的从ram中刷新到显示屏上，二者没有同步，你可以从这个方面思考一下。

回复

**randy**
2017-04-10 11:50

问一个问题：如何屏蔽掉tty对tty ldisc线性规程的处理？我现在板子上只有一个uart设备，但是这个设备默认使用line ldisc进行字符串预处理了，比如换行什么的，我现在不希望它做预处理，而是直接把数据从uart->line ldisc->tty->fs透传上来到用户态应用程序。我想到一个方案是自己写一个ldisc的驱动，然后在应用层告诉这个tty文件节点使用我写的ldisc驱动进行透传数据，但是这个比较麻烦，在应用层有可以更简单的方式使用uart接收到的数据么？

回复

**wowo**
2017-04-10 16:05

@randy：如果仅仅是去掉线路规程的预处理，在用户空间设置tty设备的termios，将input设置为non-canonical模式，并disable output的OPOST flag，应该可以实现。
具体你可以查查termios的flag，例如：
http://man7.org/linux/man-pages/man3/tcsetattr.3.html

回复

**randy**
2017-04-10 17:11

@wowo：谢谢回复。再请教一个问题：我看你前面的文章有说tty不是一个device，也就是说一个uart设备可以虚拟出多个tty设备吗？如果是这样，假如我单板上一个uart口已经用作了控制台，那再抽象出一个ttyA的话，对于读操作而言，uart怎么知道数据是扔给控制台还是ttyA呢？还是说谁先读就扔给谁？

回复

**wowo**
2017-04-11 13:25

@randy：这样的用法有点奇怪。通常情况下，一个串口，对应一个tty设备，不会有两个。

回复

**adrian**
2017-06-11 19:26

@wowo：我的看法是：
1, 你可以试试cfmakeraw
2, 如果真有这重需求的话（serial multiplexing），需要自己实现，可以考虑自己在driver里实现一个复用协议，但是需要serial两端都遵守，没有通用意义

回复

**发表评论：**

昵称

邮件地址 (选填)

个人主页 (选填)

发表评论