第十八章

TTY 驱动程序

tty 设备的名字来源于非常古老的电传打字机缩写,最初只与与 Unix 机器的物理或虚拟终端连接相关联。随着时间的推移,这个名称也开始意味着任何串行端口类型的设备,因为终端连接也可以通过这样的连接创建。物理 tty 设备的一些示例是串行端口、USB 到串行端口转换器,以及需要特殊处理才能正常工作的某些类型的调制解调器（例如传统的 Win Modem 风格的设备）。 tty 虚拟设备支持用于从键盘、通过网络连接或通过 xterm 会话登录计算机的虚拟控制台。

Linux tty 驱动程序核心位于标准字符驱动程序级别之下,并提供了一系列功能,专注于为终端样式设备提供接口。核心负责控制跨 tty 设备的数据流和数据格式。这允许 tty 驱动程序专注于处理进出硬件的数据,而不用担心如何以一致的方式控制与用户空间的交互。为了控制数据流,有许多不同的线路规程可以虚拟地“插入”到任何 tty 设备中。这是由不同的 tty 线路规程驱动程序完成的。

如图 18-1 所示,tty 核心从用户那里获取要发送到 tty 设备的数据。然后它将它传递给 tty 线路规则驱动程序,然后再将它传递给 tty 驱动程序。 tty 驱动程序将数据转换为可以发送到硬件的格式。从 tty 硬件接收的数据通过 tty 驱动程序回流,进入 tty 线路规程驱动程序,然后进入 tty 核心,用户可以在其中检索数据。有时tty驱动直接与tty核心通信,tty核心直接向tty驱动发送数据,但通常tty线路纪律有机会修改两者之间发送的数据。

tty 驱动程序永远不会看到 tty 线路规则。司机不能直接与线路纪律沟通,甚至没有意识到它的存在。驱动程序的工作是以硬件可以理解的方式格式化发送给它的数据,并从硬件接收数据。 tty line 规程的工作是格式化数据

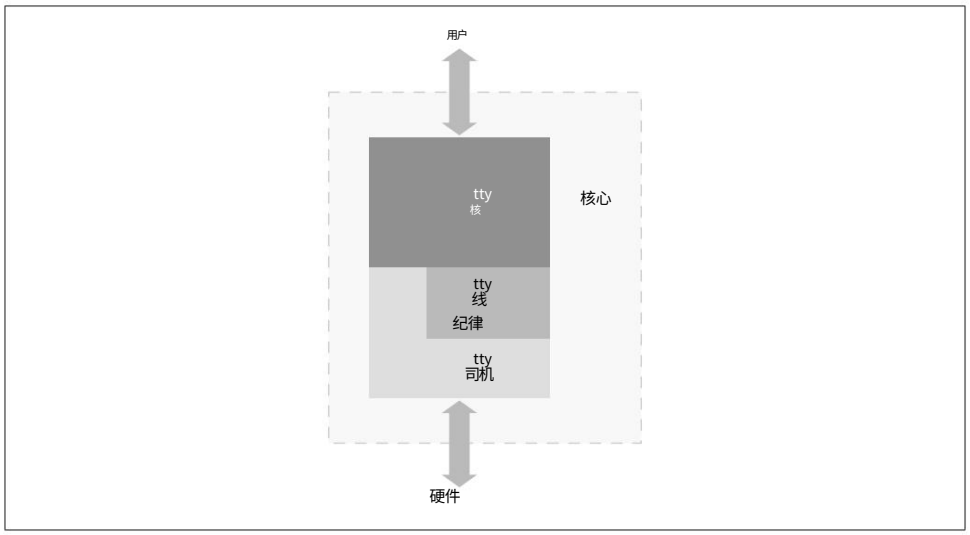

图 18-1。 tty 核心概述

以特定方式从用户或硬件接收到的信息。这种格式通常
采用协议转换的形式,如 PPP 或蓝牙。

存在三种不同类型的 tty 驱动程序:控制台、串行端口和 pty。 con sole 和 pty 驱动程序已经
编写好了,可能是唯一需要的
这些类型的 tty 驱动程序。这会留下任何使用 tty 核心进行交互的新驱动程序
与用户和系统作为串行端口驱动程序。

确定内核中当前加载了哪些类型的 tty 驱动程序以及哪些 tty
当前存在设备,请查看/proc/tty/drivers文件。该文件由一个
当前存在的不同 tty 驱动程序的列表,显示驱动程序的名称,
默认节点名称,驱动程序的主要编号,次要使用的范围
驱动程序,以及 tty 驱动程序的类型。以下是此文件的示例:

```
/dev/tty /                  /dev/tty 0 系统：/dev/tty
dev/console /               5 /dev/console 1 系统:控制台
dev/ptmx /dev/              5 /dev/ptmx 5 2 系统
vc/0 usbserial 串           /dev/vc/0 4 0 系统:vtmaster
行 pty_slave                /dev/ttyUSB 188 0-254 串行
pty_master                  /dev/ttyS 4 64-67 串行
pty_slave                   /dev/pts 136 0-255 pty:slave
pty_master 未知             /dev/ptm 128 0-255 pty:master
                            /dev/ttyp 3 0-255 pty:slave
                            /dev/pty 2 0-255 pty:master
                            /dev/tty 1-63 控制台 4
```

/proc/tty/driver/目录包含一些 tty 驱动程序的单独文件,如果
他们实现了该功能。默认串行驱动程序在此目录中创建一个文件,其中显示了许多有关硬件的串
行端口特定信息。关于如何在此目录中创建文件的信息将在后面介绍。

当前在内核中注册和存在的所有 tty 设备在/sys/class/tty下都有自己的子目录。在该子目录中，有一个“dev”文件，其中包含分配给该 tty 设备的主要和次要编号。如果驱动程序告诉内核物理设备和与 tty 设备关联的驱动程序的位置，它会创建返回它们的符号链接。这棵树的一个例子是：

```
/sys/class/tty/ |-- 控
制台 | `-- 开发 |--
ptmx | `-- 开发 |--
tty | `-- 开发 |-- tty0
| `--开发

    ...
|-- ttyS1 |
`-- 开发 |--
ttyS2 | `-- 开发
|-- ttyS3 | `--开
发

    ...
|-- ttyUSB0 |
|-- 开发 |-- 设
> ../../devices/pci0000:00/0000:00:09.0/usb3/3-1/3-1:1.0/ttyUSB0 | `-- 驱动程序-> ../../../bus/usb-serial/drivers/
keyspan_4 |-- ttyUSB1 | |-- 开发 | |-- 设备-> ../../../devices/pci0000:00/0000:00:09.0/usb3/3-1/3-1:1.0/ttyUSB1 | `--
驱动程序-> ../../../bus/usb-serial/drivers/keyspan_4 |-- ttyUSB2 | |-- 开发 | |-- 设备-> ../../../devices/
pci0000:00/0000:00:09.0/usb3/3-1/3-1:1.0/ttyUSB2 | `-- 驱动程序-> ../../../bus/usb-serial/drivers/keyspan_4

`--  ttyUSB3
     |-- 开发
     |-- 设备 -> ../../../devices/pci0000:00/0000:00:09.0/usb3/3-1/3-1:1.0/ttyUSB3 `-- 驱动程序 - > ../../../bus/usb-
     serial/drivers/keyspan_4
```

小型 TTY 驱动程序

为了解释 tty 核心是如何工作的，我们创建了一个可以加载、写入和读取以及卸载的小型 tty 驱动程序。任何 tty 驱动程序的主要数据结构是struct tty_driver。它用于向 tty 内核注册和注销 tty 驱动程序，并在内核头文件<linux/tty_driver.h> 中进行了描述。

要创建struct tty_driver,必须以该驱动程序支持的 tty 设备数量为参数调用函数 alloc_tty_driver 。这可以通过以下简短代码完成：

```
/* 分配 tty 驱动程序 */
tiny_tty_driver = alloc_tty_driver(TINY_TTY_MINORS);如果（!
tiny_tty_driver)返回-ENOMEM；
```

成功调用alloc_tty_driver函数后,应根据 tty 驱动程序的需要使用适当的信息初始化struct tty_driver 。此结构包含许多不同的字段,但并非所有字段都必须初始化才能有一个工作的 tty 驱动程序。这是一个示例,它显示了如何初始化结构并设置足够的字段来创建工作的 tty 驱动程序。它使用tty_set_operations函数来帮助复制驱动程序中定义的函数操作集：

```
静态结构 tty_operations serial_ops = { .open =
      tiny_open，.close = tiny_close，.write =
      tiny_write，.write_room =
      tiny_write_room，.set_termios =
      tiny_set_termios，

};

...

      /* 初始化 tty 驱动程序 */ tiny_tty_driver-
      >owner = THIS_MODULE; tiny_tty_driver-
      >driver_name =    tiny_tty   ; tiny_tty_driver-
      >name =    ttty   ; tiny_tty_driver->devfs_name
      =    tts/ttty%d   ; tiny_tty_driver->major =
      TINY_TTY_MAJOR,tiny_tty_driver->type =
      TTY_DRIVER_TYPE_SERIAL,tiny_tty_driver->subtype =
      SERIAL_TYPE_NORMAL,tiny_tty_driver->flags =
      TTY_DRIVER_REAL_RAW | TTY_DRIVER_NO_DEVFS,tiny_tty_driver->init_termios
      = tty_std_termios；  tiny_tty_driver->init_termios.c_cflag = B9600 | CS8 |阅读 |
      HUPCL |本地；  tty_set_operations(tiny_tty_driver, &serial_ops);
```

上面列出的变量和函数,以及如何使用这个结构,将在本章的其余部分进行解释。

要向 tty 内核注册此驱动程序,必须将struct tty_driver传递给tty_register_driver函数：

```
/* 注册 tty 驱动程序 */ retval =
tty_register_driver(tiny_tty_driver); if (retval)
{ printk(KERN_ERR    注册微型 tty 驱动程序失败   );
      put_tty_driver(tiny_tty_driver);返回 retval；


}
```

当tty_register_driver被调用时,内核为这个 tty 驱动程序可以拥有的所有次要设备创建所有不同的 sysfs tty 文件。如果您使用devfs （本书未介绍)并且除非指定了TTY_DRIVER_NO_DEVFS标志,否则也会创建devfs文件。如果您只想为系统上实际存在的设备调用tty_register_device ,则可以指定该标志,因此用户始终拥有内核中存在的设备的最新视图,这正是devfs用户所期望的。

注册自己后,驱动程序通过tty_register_device函数注册它控制的设备。这个函数有三个参数：

- 指向设备所属的struct tty_driver的指针。 · 设备的次要编号。
- 指向此 tty 设备绑定到的结构设备的指针。如果 tty 设备是
  未绑定到任何结构设备,此参数可以设置为NULL。

我们的驱动程序一次注册所有的 tty 设备,因为它们是虚拟的并且不绑定到任何物理设备：

```
对于 (i = 0; i < TINY_TTY_MINORS; ++i)
    tty_register_device(tiny_tty_driver, i, NULL);
```

要使用 tty 内核取消注册驱动程序,所有通过调用 tty_register_device 注册的 tty 设备都需要通过调用 tty_unregister_device 进行清理。
然后必须通过调用 tty_unregister_driver 来注销struct tty_driver：

```
对于 (i = 0; i < TINY_TTY_MINORS; ++i)
    tty_unregister_device(tiny_tty_driver, i);
tty_unregister_driver(tiny_tty_driver);
```

结构术语

struct tty_driver中的init_termios变量是一个struct termios。如果端口在用户初始化之前使用，则此变量用于提供一组合理的线路设置。驱动程序使用从tty_std_termios变量复制的一组标准值初始化变量。 tty_std_termios在 tty 核心中定义为：

```
结构 termios tty_std_termios = { .c_iflag
    = ICRNL | IXON，.c_oflag = OPOST
    | ONLCR，.c_cflag = B38400 | CS8 |
    创建 | HUPCL，.c_lflag = ISIG |爱卡侬 |回声 |回声
    |回声 |回声 |回声 | IEXTEN，.c_cc = INIT_C_CC


};
```

struct termios结构用于保存 tty 设备上特定端口的所有当前线路设置。这些线路设置控制当前波特率、数据

大小、数据流设置和许多其他值。这种结构的不同领域
是：

tcflag_t c_iflag;
输入模式标志tcflag_t

c_oflag;
输出模式标志tcflag_t

c_cflag;
控制模式标志

tcflag_t c_lflag;
本地模式标志

cc_t c_line;
线路规程类型cc_t

c_cc[NCCS];
控制字符数组

所有模式标志都定义为一个大位域。这些模式的不同值以及它们的用途，可以在任何 Linux 发行版中可用的 termios 手册页中看到。内核提供了一组有用的宏来获取不同的位。这些宏定义在头文件include/linux/tty.h 中。

tiny_tty_driver变量中定义的所有字段都是有工作的 tty 驱动程序所必需的。为了防止在 tty 端口打开时卸载 tty 驱动程序，必须使用owner字段。在以前的内核版本中，由 tty 驱动程序本身来处理模块引用计数逻辑。但是内核程序员认为解决所有不同的可能竞争条件是很困难的，因此 tty 内核现在为 tty 驱动程序处理所有这些控制。

driver_name和name字段看起来非常相似，但用于不同的目的。
driver_name变量应该设置为简短的、描述性的，并且在内核中的所有 tty 驱动程序中是唯一的。这是因为它显示在/proc/tty/ 驱动程序文件中，用于向用户描述驱动程序以及当前加载的 tty 驱动程序的 sysfs tty 类目录。名称字段用于定义分配给/dev树中此 tty 驱动程序的各个 tty 节点的名称。此字符串用于通过在字符串末尾附加正在使用的 tty 设备的编号来创建 tty 设备。

它还用于在 sysfs /sys/class/tty/目录中创建设备名称。如果在内核中启用了 devfs，则此名称应包括 tty 驱动程序想要放入的任何子目录。例如，如果启用了devfs ，则内核中的串行驱动程序将 name 字段设置为tts/ ，如果未启用，则设置为ttyS 。该字符串也显示在/proc/tty/drivers文件中。

正如我们提到的， /proc/tty/drivers文件显示了所有当前注册的 tty
司机。在内核中注册了tiny_tty驱动程序并且没有devfs,这个文件看起来
类似于以下内容：

```
$ cat /proc/tty/驱动程序
tiny_tty /dev/ttty 0-3 串行 240
usbserial /dev/ttyUSB 188 0-254 串行
串行 /dev/ttyS 4 64-107 串行
pty_slave /dev/pts 136 0-255 pty:slave
pty_master /dev/ptm 128 0-255 pty:master
pty_slave /dev/ttyp 3 0-255 pty:slave
pty_master /dev/pty 2 0-255 pty:master
未知 /dev/vc/ 1-63 控制台 4
/dev/vc/0 /dev/vc/0 0 系统:vtmaster 4
/dev/ptmx /dev/ptmx 2 系统 5
/dev/console /dev/console 1 系统:控制台 5
/dev/tty /dev/tty 0 系统：/dev/tty 5
```

此外,sysfs 目录/sys/class/tty在
tiny_tty 驱动程序注册到 tty 核心：

```
$树/sys/class/tty/ttty*
/sys/class/tty/ttty0
`--开发

/sys/class/tty/ttty1
`--开发

/sys/class/tty/ttty2
`--开发

/sys/class/tty/ttty3
`--开发


$猫/sys/class/tty/ttty0/dev
240:0
```

主要变量描述此驱动程序的主要编号是什么。类型和
子类型变量声明此驱动程序是什么类型的 tty 驱动程序。对于我们的示例,我们
是“正常”类型的串行驱动程序。 tty 驱动程序的唯一其他子类型是
成为“标注”类型。标注设备传统上用于控制设备的线路设置。数据将通过一个设备节点发送和接收,并且
任何线路设置更改都将发送到不同的设备节点,即呼出设备。这需要为每个 tty 设备使用两个次要编号。
值得庆幸的是,几乎所有驱动程序都同时处理数据和线路设置
设备节点,标注类型很少用于新驱动。

tty 驱动程序和 tty 核心都使用flags变量来指示驱动程序的当前状态以及它是哪种 tty 驱动程序。几个位掩码宏是
定义在测试或操作标志时必须使用的。中的三个位
flags变量可以由驱动程序设置：

TTY_DRIVER_RESET_TERMIOS

该标志表明,只要最后一个进程关闭了设备,tty 核心就会重置 termios 设置。这对于控制台和 pty 驱动程序很有用。为了

例如,假设用户让终端处于奇怪的状态。设置此标志后,当用户注销或控制会话的进程被“终止”时,终端将重置为正常值。

TTY_DRIVER_REAL_RAW
此标志表明 tty 驱动程序保证发送奇偶校验通知或中断字符上线规则。这允许线路规程以更快的方式处理接收到的字符,因为它不必检查从 tty 驱动程序接收到的每个字符。由于速度优势,通常为所有 tty 驱动程序设置此值。

TTY_DRIVER_NO_DEVFS
此标志表明当调用tty_register_driver时,tty 核心不会为 tty 设备创建任何 devfs 条目。这对于任何动态创建和销毁次要设备的驱动程序都很有用。设置此项的驱动程序示例包括 USB 转串口驱动程序、USB 调制解调器驱动程序、USB 蓝牙 tty 驱动程序和许多标准串行端口驱动程序。

当 tty 驱动程序稍后想要向 tty 内核注册特定的 tty 设备时,它必须调用tty_register_device,并带有指向 tty 驱动程序的指针,以及已创建的设备的次要编号。如果不这样做,tty 核心仍会将所有调用传递给 tty 驱动程序,但可能不存在某些与 tty 相关的内部功能。这包括新 tty 设备的/sbin/hotplug通知和 tty 设备的 sysfs 表示。当注册的 tty 设备从机器中移除时,tty 驱动程序必须调用tty_unregister_device。

该变量中剩下的一位由 tty 内核控制,称为TTY_DRIVER_INSTALLED。此标志在驱动程序注册后由 tty 核心设置,并且不应由 tty 驱动程序设置。

tty_driver 函数指针

最后， tiny_tty驱动程序声明了四个函数指针。

open 和 close当用户

在分配 tty 驱动程序的设备节点上调用open时,tty 核心调用open函数。 tty 核心使用指向分配给此设备的tty_struct结构的指针和文件指针来调用它。 open field 必须由 tty 驱动程序设置才能正常工作;否则,在调用 open 时将-ENODEV返回给用户。

当调用这个open函数时,tty 驱动程序要么在传递给它的tty_struct变量中保存一些数据,要么将数据保存在一个静态数组中,该数组可以根据端口的次要编号进行引用。这是必要的,因此

当后面的 close、write 和其他函数被调用时,tty 驱动程序知道正在引用哪个设备。

tiny_tty驱动程序在 tty 结构中保存了一个指针,如以下代码所示：

```
静态 int tiny_open(struct tty_struct *tty, struct file *file) {

    结构 tiny_serial *tiny;结构
    timer_list *timer;整数索引；


    /* 初始化指针以防万一失败 */ tty->driver_data = NULL;


    /* 获取与此 tty 指针关联的串行对象 */ index = tty->index; tiny = tiny_table[索
    引]； if (tiny = = NULL) { /* 第一次访问这个设备,让我们创建它 */ tiny =
    kmalloc(sizeof(*tiny), GFP_KERNEL);如果 (!tiny) 返回 -ENOMEM;




        init_MUTEX(&tiny->sem);
        微小->open_count = 0;微
        小->计时器=空；

        tiny_table[索引] = 小；
    }

    向下(&tiny->sem);

    /* 将我们的结构保存在 tty 结构中 */ tty->driver_data = tiny;微
    小->tty = tty；
```

在这段代码中， tiny_serial结构保存在tty结构中。这允许tiny_write、 tiny_write_room和tiny_close函数检索tiny_serial结构并正确操作它。

tiny_serial结构定义为：

```
结构 tiny_serial { 结构
    tty_struct *tty; /* 指向此设备的 tty 的指针 */ int open_count; /* 这个端口被打开的次数 */
    struct semaphore sem;结构 timer_list *timer;
                                        /* 锁定这个结构 */


};
```

正如我们所见， open_count变量在第一次打开端口时在 open 调用中被初始化为0 。这是一个典型的引用计数器,因为 tty 驱动程序的open和close函数可以为同一个设备调用多次

为了允许多个进程读取和写入数据。为了正确处理所有事情,必须记录端口打开或关闭的次数;驱动程序在使用端口时递增和递减计数。首次打开端口时,可以完成任何所需的硬件初始化和内存分配。当端口最后一次关闭时,可以完成任何需要的硬件关闭和内存清理。

tiny_open函数的其余部分显示了如何跟踪设备打开的次数:

```
++tiny->open_count;
if (tiny->open_count = = 1) { /* 这
    是第一次打开这个端口 */ /* 在这里做任何需要的硬件初始化 */
```

如果发生了阻止打开成功的情况,打开函数必须返回一个负错误号,或者返回一个0表示成功。

当用户在文件句柄上调用close时,tty 核心调用close函数指针,该文件句柄是先前通过调用open创建的。这表明此时应关闭设备。但是,由于可以多次调用open函数,因此也可以多次调用close函数。所以这个函数应该跟踪它被调用了多少次,以确定此时硬件是否真的应该关闭。 tiny_tty驱动程序使用以下代码执行此操作: static void do_close(struct tiny_serial *tiny) {

```
    向下(&tiny->sem);

    if (!tiny->open_count) { /* 端
        口从未打开 */ goto exit;

    }

    --tiny->open_count; if
    (tiny->open_count <= 0) { /* 端口
        正在被最后一个用户关闭。 */ /* 在这里做任何硬件特定的东西
        */

        /* 关闭我们的定时器 */
        del_timer(tiny->timer);

} 出口:
    向上(&tiny->sem);
}

静态无效 tiny_close（结构 tty_struct *tty,结构文件 *file){

    结构 tiny_serial *tiny = tty->driver_data;
```

```
    如果（小）
        do_close（小）；
}
```

tiny_close函数只是调用do_close函数来完成关闭设备的真正工作。这样做是为了不必在此处复制关闭逻辑以及在卸载驱动程序并打开端口时。 close函数没有返回值,因为它不应该失败。

数据流

当有数据要发送到硬件时,用户调用write函数调用。首先 tty 核心接收调用,然后将数据传递给 tty 驱动程序的写入函数。 tty 核心还告诉 tty 驱动程序数据的大小

发送。

有时,由于 tty 硬件的速度和缓冲容量,在调用write函数的那一刻,并非所有写入程序请求的字符都可以发送。写入函数应返回能够发送到硬件（或排队等待稍后发送)的字符数,以便用户程序可以检查是否所有数据都已真正写入。在用户空间中进行这种检查比让内核驱动程序坐下来睡觉直到所有请求的数据都能够发送出去要容易得多。如果在write调用期间发生任何错误,则应返回负错误值而不是写入的字符数。

可以从中断上下文和用户上下文调用write函数。了解这一点很重要,因为 tty 驱动程序不应调用任何在中断上下文中可能休眠的函数。其中包括任何可能调用schedule的函数,例如常用函数 copy_from_user、 kmalloc和printk。如果你真的想睡觉,请确保首先通过调用in_interrupt 检查驱动程序是否处于中断上下文中。

这个示例微型 tty 驱动程序不连接到任何实际硬件,因此它的写入功能只是在内核调试日志中记录应该写入的数据。它使用以下代码执行此操作：

```
static int tiny_write(struct tty_struct *tty, const
                      unsigned char *buffer, int count)
{
    结构 tiny_serial *tiny = tty->driver_data;诠释我; int
    retval = -EINVAL;


    如果 (!tiny)
        返回 -ENODEV;

    向下(&tiny->sem);
```

```
        if (!tiny->open_count) /*
            端口没有打开 */ goto exit;


        /* 通过将数据写入内核调试日志来伪造从硬件端口发送数据。
         */

        printk(KERN_DEBUG    %s -    , __FUNCTION__);
        对于 (i = 0; i < 计数; ++i)
            printk(    %02x    , 缓冲区[i]);
        printk(    \n    );

    退出：
        向上（&tiny-
        >sem）;返回 retval;
    }
```

当tty 子系统本身需要向 tty 设备发送一些数据时,可以调用write函数。如果 tty 驱动程序没有在 tty_struct 中实现put_char函数,就会发生这种情况。在这种情况下,tty 核心使用数据大小为 1 的write函数回调。这通常发生在 tty 核心想要将换行符转换为换行符加换行符时。这里可能出现的最大问题是 tty 驱动程序的write函数不能为这种调用返回0 。这意味着驱动程序必须将该字节数据写入设备,因为调用者（tty 核心)不会缓冲数据并稍后重试。由于write函数无法确定是否在 put_char位置被调用,因此即使只发送了一个字节的数据,请尝试实现write函数,使其始终在返回之前写入至少一个字节。许多当前的 USB 转串口 tty 驱动程序不遵循此规则,因此,某些终端类型在连接到它们时无法正常工作。

当tty 内核想知道 tty 驱动程序在写缓冲区中有多少可用空间时,调用write_room函数。这个数字随着时间的推移而改变,因为字符从写缓冲区中清空,并且当调用write函数时,将字符添加到缓冲区中。

```
    静态 int tiny_write_room(struct tty_struct *tty) {

        结构 tiny_serial *tiny = tty->driver_data; int 房间 =
        -EINVAL;

        如果 (!tiny)
            返回 -ENODEV;

        向下(&tiny->sem);

        if (!tiny->open_count) { /* 端
            口没有打开 */ goto exit;

        }
```

```
        /* 计算设备还剩多少空间 */
        房间 = 255;

退出：
        向上（&tiny-
        >sem）;回房；
}
```

其他缓冲函数tty_driver结构中的

chars_in_buffer函数对于具有工作的 tty 驱动程序不是必需的,但建议使用。当 tty 内核想知道 tty 驱动程序的写缓冲区中还有多少字符要发送出去时,就会调用这个函数。如果驱动程序可以在将字符发送到硬件之前存储字符,则它应该实现此功能,以便 tty 内核能够确定驱动程序中的所有数据是否已耗尽。

tty_driver结构中的三个回调函数可用于刷新驱动程序保留的任何剩余数据。这些不是必须实现的,但如果 tty 驱动程序可以在将数据发送到硬件之前缓冲数据,则建议使用。前两个函数回调称为flush_chars和wait_until_sent。

当 tty 核心使用put_char函数回调将许多字符发送到 tty 驱动程序时,将调用这些函数。当tty 核心希望 tty 驱动程序开始将这些字符发送到硬件时调用flush_chars函数回调（如果尚未启动）。在所有数据发送到硬件之前,允许此函数返回。 wait_until_sent函数回调的工作方式大致相同;但它必须等到所有字符都发送完毕后再返回到 tty 核心,或者直到传入的超时值过期,以先发生者为准。允许 tty 驱动程序在此函数内休眠以完成它。如果传递给wait_until_sent函数回调的超时值设置为0,则该函数应等待操作完成。

剩下的数据刷新函数回调是flush_buffer。当 tty 驱动程序要将仍在其写入缓冲区中的所有数据刷新出内存时,它由 tty 核心调用。缓冲区中剩余的任何数据都会丢失并且不会发送到设备。

没有读取功能?

仅使用这些函数,就可以注册tiny_tty驱动程序、打开设备节点、将数据写入设备、关闭设备节点以及取消注册驱动程序并从内核中卸载。但是 tty 核心和tty_driver结构不提供读取功能;换句话说;不存在将数据从驱动程序获取到 tty 核心的函数回调。

与传统的读取功能不同,tty 驱动程序负责在接收到从硬件接收到的任何数据时将其发送到 tty 核心。 tty 核心缓冲

数据,直到用户要求。因为 tty 内核提供了缓冲逻辑,所以没有必要每个 tty 驱动程序都实现自己的缓冲逻辑。当用户希望驱动程序停止并开始发送数据时,tty 核心会通知 tty 驱动程序,但如果内部 tty 缓冲区已满,则不会发生此类通知。

tty 核心将 tty 驱动程序接收到的数据缓存在一个名为struct tty_flip_buffer 的结构中。翻转缓冲区是包含两个主要数据数组的结构。从 tty 设备接收的数据存储在第一个数组中。当该数组已满时,任何等待数据的用户都会收到数据可供读取的通知。当用户从这个数组中读取数据时,任何新的传入数据都存储在第二个数组中。当该数组完成后,数据再次刷新给用户,驱动程序开始填充第一个数组。本质上,接收到的数据从一个缓冲区“翻转”到另一个缓冲区,希望不会溢出它们。为了防止数据丢失,tty 驱动程序可以监视传入数组的大小,如果它已满,则告诉 tty 驱动程序在此时刷新缓冲区,而不是等待下一个可用的机会。

struct tty_flip_buffer结构的细节对tty 驱动程序并不重要,除了变量计数。此变量包含缓冲区中当前用于接收数据的字节数。如果此值等于值TTY_FLIPBUF_SIZE,则需要通过调用 tty_flip_buffer_push 将翻转缓冲区刷新给用户。这显示在以下代码中:

```
for (i = 0; i < data_size; ++i) { if (tty-
    >flip.count >= TTY_FLIPBUF_SIZE)
        tty_flip_buffer_push(tty);
    tty_insert_flip_char（tty,数据[i],TTY_NORMAL）；

} tty_flip_buffer_push(tty);
```

从 tty 驱动程序接收到的要发送给用户的字符通过调用tty_insert_flip_char添加到翻转缓冲区。这个函数的第一个参数是应该保存数据的struct tty_struct ,第二个参数是要保存的字符,第三个参数是应该为这个字符设置的任何标志。如果这是接收到的普通字符,则标志值应设置为 TTY_NORMAL 。如果这是表示接收数据错误的特殊类型字符,则应将其设置为TTY_BREAK、TTY_FRAME、 TTY_PARITY或TTY_OVERRUN,具体取决于

错误。

为了将数据“推送”给用户,调用tty_flip_buffer_push 。如果翻转缓冲区即将溢出,也应调用此函数,如本例所示。因此,每当数据被添加到翻转缓冲区,或者翻转缓冲区已满时,tty 驱动程序必须调用tty_flip_buffer_push。如果 tty 驱动程序可以以非常高的速率接受数据,则应设置tty->low_latency标志,这会导致对tty_flip_buffer_push的调用在调用时立即执行。否则,该

tty_flip_buffer_push调用安排自己在不久的将来某个时间点将数据推出缓冲区。

TTY 线路设置

当用户想要更改 tty 设备的线路设置或检索当前线路设置时,他会进行许多不同的 termios 用户空间库函数调用之一或直接在 tty 设备节点上进行ioctl调用。 tty 核心将这两个接口转换为许多不同的 tty 驱动程序函数回调和ioctl调用。

set_termios大多

数 termios 用户空间函数由库转换为对驱动程序节点的ioctl调用。大量不同的 tty ioctl调用随后由 tty 核心转换为对 tty 驱动程序的单个set_termios函数调用。 set_termios回调需要确定它被要求更改的线路设置,然后在 tty 设备中进行这些更改。 tty 驱动程序必须能够解码 termios 结构中的所有不同设置并对任何需要的更改做出反应。这是一项复杂的任务,因为所有线路设置都以多种方式打包到 termios 结构中。

set_termios函数应该做的第一件事是确定是否确实需要更改任何内容。这可以通过以下代码完成:

```
无符号整数 cflag;

cflag = tty->termios->c_cflag;

/* 检查他们是否真的希望我们改变某些东西 */ if (old_termios) { if ((cflag
= = old_termios->c_cflag) &&

            (RELEVANT_IFLAG(tty->termios->c_iflag) = =
              RELEVANT_IFLAG(old_termios->c_iflag)))
            { printk(KERN_DEBUG 返回什么可改变的...\n );

      }
}
```

RELEVANT_IFLAG宏定义为:

```
#define RELEVANT_IFLAG(iflag) ((iflag) & (IGNBRK|BRKINT|IGNPAR|PARMRK|INPCK))
```

并用于屏蔽cflags变量的重要位。然后将其与旧值进行比较,看看它们是否不同。如果不是,则无需更改任何内容,因此我们返回。请注意,在访问 old_termios变量之前,首先检查它是否指向有效结构。这是必需的,因为有时此变量设置为NULL。尝试从NULL指针访问字段会导致内核恐慌。

要查看请求的字节大小,可以使用CSIZE位掩码从cflag变量中分离出正确的位。如果无法确定大小，则习惯上默认为 8 个数据位。这可以按如下方式实现：

```
/* 获取字节大小 */ switch
(cflag & CSIZE) { case CS5:

        printk(KERN_DEBUG    - 数据位 = 5\n    );休息;
        案例 CS6：

        printk(KERN_DEBUG    - 数据位 = 6\n    );休息;
        案例 CS7：

        printk(KERN_DEBUG    - 数据位 = 7\n    );休息;
        默认值:案例 CS8：


        printk(KERN_DEBUG    - 数据位 = 8\n    );休息;

}
```

要确定请求的奇偶校验值,可以对照 cflag 变量检查 PARENB 位掩码,以判断是否要设置任何奇偶校验。如果是这样， PARODD位掩码可用于确定奇偶校验应该是奇数还是偶数。一个实现是：

```
/* 确定奇偶校验 */ if (cflag &
PARENB) if (cflag & PARODD)
    printk(KERN_DEBUG    -
        parity = odd\n    );否则 printk(KERN_DEBUG
    - 奇偶校验 = 偶数\n    );

否则
    printk(KERN_DEBUG    - parity = none\n    );
```

请求的停止位也可以使用CSTOPB位掩码从cflag变量中确定。一个实现是：

```
/* 找出请求的停止位 */ if (cflag & CSTOPB)
printk(KERN_DEBUG    - stop bits = 2\n    );否
    则 printk(KERN_DEBUG    - 停止位 = 1\n    );
```

流量控制有两种基本类型:硬件和软件。要确定用户是否要求硬件流控制,可以对照 cflag 变量检查 CRTSCTS位掩码。这方面的一个例子是：

```
/* 找出硬件流控制设置 */ if (cflag & CRTSCTS)
printk(KERN_DEBUG
                                - RTS/CTS 已启用\n    ); 否则

    printk(KERN_DEBUG    - RTS/CTS 被禁用\n    );
```

确定软件流控制的不同模式和不同的停止和开始字符有点复杂：

最后，需要确定波特率。 tty 核心提供了一个函数tty_get_baud_rate来帮助执行此操作。该函数返回一个整数，指示特定 tty 设备的请求波特率：

```
/* 得到想要的波特率 */
printk(KERN_DEBUG "baud rate = %d\n", tty_get_baud_rate(tty));
```

现在 tty 驱动程序已经确定了所有不同的线路设置，它可以根据这些值正确设置硬件。

tiocmget 和 tiocmset在 2.4 和

更早的内核中，曾经有许多 tty ioctl调用来获取和设置不同的控制线设置。这些由常数 TIOCMGET、 TIOCMBIS、 TIOCMBIC和TIOCMSET 表示。 TIOCMGET用于获取内核的行设置值，从 2.6 内核开始，这个ioctl调用已经变成了一个 tty 驱动回调函数，称为tiocmget。其他三个ioctl已被简化，现在由一个名为tiocmset 的 tty 驱动程序回调函数表示。

tty 驱动程序中的tiocmget函数由 tty 内核调用，当内核想知道特定 tty 设备的控制线的当前物理值时。这通常用于检索串行端口的 DTR 和 RTSlines 的值。如果 tty 驱动程序不能直接读取串口的 MSR 或 MCR 寄存器，因为硬件不允许这样做，应该在本地保留一份副本。一些

USB 转串口驱动程序必须实现这种“影子”变量。如果保留这些值的本地副本，以下是如何实现此功能：

```
静态 int tiny_tiocmget(struct tty_struct *tty, struct file *file) {

    结构 tiny_serial *tiny = tty->driver_data;

    无符号整数结果 = 0;无符号 int
    msr = tiny->msr;无符号整数 mcr =
    tiny->mcr;

    结果 = ((mcr & MCR_DTR) ? TIOCM_DTR : 0) | /* DTR 已设置 */ ((mcr &
                MCR_RTS) ? TIOCM_RTS : 0) | /* RTS 已设置 */ ((mcr & MCR_LOOP) ?
                TIOCM_LOOP : 0) | /* LOOP 已设置 */ ((msr & MSR_CTS) ? TIOCM_CTS :
                0) | /* CTS 已设置 */ ((msr & MSR_CD) ? TIOCM_CAR : 0) | /* 载波检测
                已设置*/ ((msr & MSR_RI) ? TIOCM_RI : 0) | /* 设置环指示器 */ ((msr & MSR_DSR) ?
                TIOCM_DSR : 0); /* DSR 已设置 */


    返回结果；
}
```

当内核想要设置特定 tty 设备的控制线的值时，tty 驱动程序中的tiocmset函数由 tty 内核调用。 tty 核心通过将它们传递到两个变量中来告诉 tty 驱动程序要设置什么值以及要清除什么： set和clear。这些变量包含应更改的行设置的位掩码。

ioctl调用从不要求驱动程序同时设置和清除特定位，因此哪个操作先发生并不重要。下面是一个 tty 驱动程序如何实现此功能的示例：

```
static int tiny_tiocmset(struct tty_struct *tty, struct file *file, unsigned int set,
                         unsigned int clear)
{
    结构 tiny_serial *tiny = tty->driver_data;无符号整数
    mcr = tiny->mcr；

    if (set & TIOCM_RTS)
        mcr |= MCR_RTS; if
    (set & TIOCM_DTR) mcr
        |= MCR_RTS;

    if (clear & TIOCM_RTS)
        mcr &= ~MCR_RTS;
    if (clear & TIOCM_DTR)
        mcr &= ~MCR_RTS;

    /* 在设备中设置新的 MCR 值 */ tiny->mcr = mcr;返回
    0；

}
```

ioctl

当在设备节点上调用ioctl(2)时,tty 核心会调用struct tty_driver中的ioctl函数回调。如果 tty 驱动程序不知道如何处理传递给它的ioctl值,它应该返回-ENOIOCTLCMD以尝试让 tty 核心实现调用的通用版本。

2.6 内核定义了大约 70 种不同的 tty ioctl ,可以发送到 tty 驱动程序。
大多数 tty 驱动程序并不处理所有这些,而只是更常见的驱动程序的一小部分。以下是更流行的 tty ioctl 列表,它们的含义以及如何实现它们:

TIOCSERGETLSR

获取此 tty 设备的线路状态寄存器 (LSR) 的值。

TIOCG系列

获取串行线路信息。在此调用中,调用者可能会同时从 tty 设备获得大量串行线路信息。一些程序(如setserial和dip)调用此函数以确保正确设置波特率并获取有关 tty 驱动程序控制的设备类型的一般信息。

调用者传入一个指向serial_struct 类型的大型结构的指针， tty 驱动程序应该用正确的值填充该结构。这是一个如何实现的示例:

```
static int tiny_ioctl(struct tty_struct *tty, struct file *file, unsigned int cmd,
                      unsigned long arg)
{
    结构 tiny_serial *tiny = tty->driver_data;如果（cmd
    = = TIOCGSERIAL){
        结构serial_struct tmp;如果（!
        arg)返回-EFAULT;
            memset(&tmp, 0,
        sizeof(tmp)); = 微小->serial.type;
        tiny->serial.line; = 微小->serial.port;
        tmp.port tmp.irq = tmp.flags =
        ASYNC_AUTO_IRQ; ASYNC_SKIP_TEST |
        tiny->serial.xmit_fifo_size; tmp.baud_base =
        tmp.close_delay = 5*HZ; tmp.closing_wait =
        >serial.custom_divisor; tmp.hub6 = tiny->serial.hub6;
        tmp.io_type = tiny->serial.io_type; if (copy_to_user((void
        回 -EFAULT;返回0;

    } 返回 -ENOIOCTLCMD;
}
```

TIOCSSERIAL

设置串行线路信息。这与TIOCGSERIAL正好相反,允许用户一次性设置 tty 设备的串行线路状态。一个指向struct serial_struct的指针被传递给这个调用,其中充满了 tty 设备现在应该设置的数据。如果 tty 驱动程序没有实现这个调用,大多数程序仍然可以正常工作。

TIOCMIWAIT

等待 MSR 更改。用户在它想在内核中休眠直到 tty 设备的 MSR 寄存器发生某些事情的异常情况下请求这个ioctl 。 arg参数包含用户正在等待的事件类型。这通常用于等待状态行发生变化,发出更多数据已准备好发送到设备的信号。

实现这个ioctl时要小心,不要使用interruptible_sleep_on调用,因为它是不安全的（涉及到很多讨厌的竞争条件）。

相反,应该使用wait_queue来避免这些问题。下面是如何实现这个 ioctl 的示例： static int tiny_ioctl(struct tty_struct *tty, struct file *file, unsigned int cmd, unsigned long arg)

```
{
    结构 tiny_serial *tiny = tty->driver_data;如果（cmd =
    = TIOCMIWAIT){ DECLARE_WAITQUEUE（等待,当前）；
        结构 async_icount cnow;结构 async_icount
        cprev; cprev = tiny->icount； while (1)
        { add_wait_queue(&tiny->wait, &wait);
        set_current_state(TASK_INTERRUPTIBLE);日程
        （）; remove_wait_queue(&tiny->wait, &wait); /
            * 查看是否有信号唤醒我们 */ if
            (signal_pending(current)) return
            -ERESTARTSYS; cnow = tiny->icount; if
            (cnow.rng = = cprev.rng && cnow.dsr = =
            cprev.dsr && cnow.dcd = = cprev.dcd &&
            cnow.cts = = cprev.cts) 返回-EIO； /* 没有变化
                => 错误 */ if (((arg & TIOCM_RNG) &&
            (cnow.rng != cprev.rng)) || ((arg & TIOCM_DSR)
            && (cnow.dsr != cprev.dsr)) | | ((arg & TIOCM_CD) && (cnow.dcd !=
                cprev.dcd)) || ((arg & TIOCM_CTS) && (cnow.cts != cprev.cts)) )
                { return 0;




            } cprev = 知道；
        }

    } 返回 -ENOIOCTLCMD;
}
```

在识别 MSR 寄存器更改的 tty 驱动程序代码中的某处,必须调用以下行以使该代码正常工作:

```
唤醒中断（&tp->等待）；
```

票数

获取中断计数。当用户想知道发生了多少串行线路中断时调用它。如果驱动程序有一个中断处理程序,它应该定义一个计数器的内部结构来跟踪这些统计信息,并在每次内核运行该函数时递增适当的计数器。

这个ioctl调用向内核传递了一个指向结构serial_icounter_struct的指针,该结构应该由 tty 驱动程序填充。此调用通常与先前的TIOCMIWAIT ioctl调用一起进行。如果 tty 驱动程序在驱动程序运行时跟踪所有这些中断,则实现此调用的代码可以非常简单:

```
static int tiny_ioctl(struct tty_struct *tty, struct file *file, unsigned int
                      cmd, unsigned long arg)
{
    结构 tiny_serial *tiny = tty->driver_data; if (cmd =
    = TIOCGICOUNT) { struct async_icount cnow =
        tiny->icount; struct serial_icounter_struct
        icount；  icount.cts = cnow.cts; icount.dsr =
        cnow.dsr; icount.rng = cnow.rng; icount.dcd
        = cnow.dcd; icount.rx = cnow.rx; icount.tx =
        cnow.tx; icount.frame = cnow.frame;
        icount.overrun = cnow.overrun; icount.parity
        = cnow.parity; icount.brk = cnow.brk;
        icount.buf_overrun = cnow.buf_overrun; if
        (copy_to_user((void __user *)arg, &icount,
                      sizeof(icount)))  return
        -EFAULT;返回0；

    } 返回 -ENOIOCTLCMD;
}
```

TTY 设备的 proc 和 sysfs 处理

tty 核心为任何 tty 驱动程序提供了一种非常简单的方法来维护/proc/tty/driver目录中的文件。如果驱动程序定义了read_proc或write_proc函数,则创建此文件。然后,对该文件的任何读取或写入调用都将发送到驱动程序。这些函数的格式就像标准的/proc文件处理函数一样。

例如,这里是read_proc tty回调的简单实现,它仅打印出当前注册的端口数:

```
静态 int tiny_read_proc(char *page, char **start, off_t off, int count, int *eof, void
                                        *data)
{
    结构 tiny_serial *tiny; off_t 开
    始 = 0;整数长度 = 0;诠释我;



    长度 += sprintf(page,    tinyserinfo:1.0 driver:%s\n    , DRIVER_VERSION);对于 (i = 0; i
    < TINY_TTY_MINORS && 长度 < PAGE_SIZE; ++i) { tiny = tiny_table[i]; if (tiny = = NULL)
        继续;



        长度 += sprintf(页面+长度,    %d\n    , i); if ((length
        + begin) > (off + count)) 完成;如果（（长度+开始)<
            关闭){开始+=长度;长度 = 0;



        }

    } *eof = 1;
完成:如果（关闭
    > =（长度+开始））返回0;  *start =
        page + (off-begin);返回（计数
    <开始+长度关闭）?计数:开始+长度
    关闭;
}
```

当注册 tty 驱动程序或创建单个 tty 设备时,tty 核心处理所有 sysfs 目录和设备创建,这取决于struct tty_driver中的TTY_DRIVER_NO_DEVFS标志。单个目录始终包含dev文件,它允许用户空间工具确定分配给设备的主要和次要编号。如果在对tty_register_device的调用中传递了指向有效结构设备的指针,则它还包含设备和驱动程序符号链接。

除了这三个文件之外,单个 tty 驱动程序不可能在此位置创建新的 sysfs 文件。这可能会在未来的内核版本中改变。

tty_driver 结构详解

tty_driver结构用于向 tty 内核注册一个 tty 驱动程序。以下是结构中所有不同字段的列表以及 tty 核心如何使用它们:

结构模块*所有者;此驱动程
    序的模块所有者。

诠释魔法;这
种结构的“神奇”价值。应始终设置为TTY_DRIVER_MAGIC。
在alloc_tty_driver函数中初始化。 const char
*driver_name;
驱动程序的名称,在/proc/tty和 sysfs 中使用。常量字符*
名称;驱动程序的节点名称。

int name_base;
为设备创建名称时使用的起始编号。这在内核创建分配给 tty 驱动程序的特定 tty 设备的字符串表示时使用。

短期专业;驱动
程序的主要编号。短小的开始;驱动
程序的起始次要编号。这通常设置为与
name_base 相同的值。通常,此值设置为0。

短号;分配给
驱动程序的次要编号的数量。如果驱动程序使用整个主编号范围,则该值应设置为 255。此变量在alloc_tty_driver函数中初始化。

短型;短亚型;
描述向 tty 核心
注册的 tty 驱动程序类型。 subtype的值取决于类型。类型字段可以是:
TTY_DRIVER_TYPE_SYSTEM由 tty 子系统内部使用,以记住它正在处理内部 tty 驱动程序。子类型应设置为SYSTEM_TYPE_TTY、 SYSTEM_TYPE_CONSOLE、 SYSTEM_TYPE_SYSCONS
或SYSTEM_TYPE_SYSPTMX。任何“普通”的 tty 驱动程序都不应使用此类型。

TTY_DRIVER_TYPE_CONSOLE
仅由控制台驱动程序使用。
TTY_DRIVER_TYPE_SERIAL
由任何串行类型驱动程序使用。 subtype应设置为SERIAL_TYPE_NORMAL或
SERIAL_TYPE_CALLOUT,具体取决于您的驱动程序的类型。这是类型字段最常见的设置之一。

TTY_DRIVER_TYPE_PTY
由伪终端接口 (pty) 使用。子类型需要设置为PTY_TYPE_MASTER或PTY_TYPE_SLAVE。

结构 termios init_termios;
设备创建时的初始 struct termios 值。

整数标志;驱
动程序标志,如本章前面所述。

结构 proc_dir_entry *proc_entry;
此驱动程序的/proc条目结构。如果驱动程序实现了write_proc或read_proc函数,它由 tty 核心创建。该字段不应由 tty 驱动程序本身设置。

结构 tty_driver *其他;
指向 tty 从驱动程序的指针。这仅由 pty 驱动程序使用,不应由任何其他 tty 驱动程序使用。无效*驱动程序状态; tty 驱动程序的内部状态。只能由 pty 驱动程序使用。

结构 tty_driver *next;结构
tty_driver *prev;链接变量。
tty 核心使用这些变量将所有不同的 tty 驱动程序链接在一起,并且不应被任何 tty 驱动程序触及。

tty_operations 结构详解

tty_operations结构包含所有可以由 tty 驱动程序设置并由 tty 核心调用的函数回调。目前,这个结构中包含的所有函数指针也在tty_driver结构中,但很快就会被这个结构的一个实例所取代。

int (*open)(struct tty_struct打开 * tty, 结构文件 * filp);
函数。 void (*close)(struct
tty_struct关闭函数。 * tty, 结构文件 * filp);

int (*write)(struct tty_struct写函 * tty, const unsigned char *buf, int count);
数。

void (*put_char)(struct tty_struct *tty, unsigned char ch);单字符写
入功能。当将单个字符写入设备时,此函数由 tty 核心调用。如果 tty 驱动程序未定义此函数,则当 tty 核心想要发送单个字符时,将调用write函数。 void (*flush_chars)(struct tty_struct *tty); void (*wait_until_sent)(struct tty_struct *tty, int timeout);将数据刷新到硬件的函数。

int (*write_room)(struct tty_struct *tty);指示有多
少缓冲区可用的函数。

int (*chars_in_buffer)(struct tty_struct *tty);指示缓冲区
中有多少已满数据的函数。

int (*ioctl)(struct tty_struct *tty, struct file * file, unsigned int cmd, unsigned long arg); ioctl函数。当在设备节点上调用ioctl(2)时,tty 核心会调用此函数。

void (*set_termios)(struct tty_struct *tty, struct termios * old);
set_termios函数。当设备的 termios 设置更改时,tty 核心调用此函数。 void (*throttle)(struct tty_struct * tty); void (*unthrottle)(struct tty_struct * tty); void (*stop)(struct tty_struct *tty); void (*start)(struct tty_struct *tty);

数据节流功能。这些函数用于帮助控制 tty 内核输入缓冲区的溢出。当tty 核心的输入缓冲区快满时调用节流函数。 tty 驱动程序应该尝试向设备发出信号,不再向其发送字符。unthrottle函数在 tty 核心的输入缓冲区被清空时调用,它现在可以接受更多数据。然后 tty 驱动程序应该向设备发出可以接收数据的信号。

stop和start函数很像油门和unthrottle函数,但它们表示 tty 驱动程序应该停止向设备发送数据,然后再继续发送数据。 void (*hangup)(struct tty_struct *tty);

挂机功能。当 tty 驱动程序应该挂断 tty 设备时调用此函数。此时应进行执行此操作所需的任何特殊硬件操作。

void (*break_ctl)(struct tty_struct *tty, int state);
换行控制功能。当 tty 驱动程序要打开或关闭 RS-232 端口的线路 BREAK 状态时调用该函数。如果 state 设置为–1,则应打开 BREAK 状态。如果状态设置为0,则应关闭 BREAK 状态。如果此功能由 tty 驱动程序实现,则 tty 核心将处理TCSBRK、 TCSBRKP、 TIOCSBRK和TIOCCBRK ioctl。否则,这些ioctl将被发送到驱动程序的ioctl函数。

void (*flush_buffer)(struct tty_struct *tty);刷新缓冲区并丢失任何剩余数据。 void (*set_ldisc)(struct tty_struct *tty);设置线路纪律功能。当 tty 核心更改了 tty 驱动程序的线路规则时,将调用此函数。此函数一般不使用,也不应由驱动程序定义。 void (*send_xchar)(struct tty_struct *tty, char ch);

发送X 型 char函数。该函数用于向 tty 设备发送高优先级 XON 或 XOFF 字符。要发送的字符在ch变量中指定。

int (*read_proc)(char *page, char **start, off_t off, int count, int *eof, void *data);
int (*write_proc)(struct file *file, const char *buffer, unsigned
long count, void *data); /proc 读取和写入函数。

int (*tiocmget)(struct tty_struct *tty, struct file *file);获取特定 tty
设备的当前线路设置。如果从 tty 设备成功检索,则该值应返回给调用者。 int (*tiocmset)
(struct tty_struct *tty, struct file *file, unsigned int set,

无符号整数清除）；
设置特定 tty 设备的当前线路设置。 set和clear包含应该设置或清除的不同线路设置。

tty_struct 结构详解

tty 核心使用tty_struct变量来保持特定 tty 端口的当前状态。几乎所有的字段都只能由 tty 核心使用,除了少数例外。 tty 驱动程序可以使用的字段如下所述：

无符号长标志； tty 设
备的当前状态。这是一个位域变量,可通过以下宏访问： TTY_THROTTLED当驱动程序调用了节流函数时设置。不应由 tty 驱动程序设置,只能由 tty 核心设置。

TTY_IO_ERROR
当驱动程序不希望从驱动程序读取或写入任何数据时由驱动程序设置。如果用户程序试图这样做,它会收到来自内核的 -EIO 错误。这通常在设备关闭时设置。

TTY_OTHER_CLOSED
仅由 pty 驱动程序用于通知端口何时关闭。

TTY_EXCLUSIVE
由 tty 核心设置,表示端口处于独占模式,一次只能由一个用户访问。

TTY_DEBUG在内核的任何地方都没有使用。

TTY_DO_WRITE_WAKEUP如果设置了此项,则允许调用线路规程的 write_wakeup 函数。这通常在tty 驱动程序调用wake_up_interruptible函数的同时被调用。

TTY_PUSH
仅由默认 tty 线路规程在内部使用。

TTY_CLOSING
由 tty 核心用来跟踪端口是否在那个时刻正在关闭。

TTY_DONT_FLIP
由默认 tty 线路规程使用,用于通知 tty 核心在设置时不应更改翻转缓冲区。

TTY_HW_COOK_OUT
如果由 tty 驱动程序设置,它会通知线路规程它将“烹饪”发送给它的输出。如果未设置,则线路规程以块的形式复制驱动程序的输出;否则,它必须评估单独发送的每个字节以进行行更改。此标志通常不应由 tty 驱动程序设置。

TTY_HW_COOK_IN几乎与在驱动程序标志变量中设置TTY_DRIVER_REAL_RAW标志相同。此标志通常不应由 tty 驱动程序设置。

TTY_PTY_LOCK
由 pty 驱动程序用于锁定和解锁端口。

TTY_NO_WRITE_SPLIT
如果设置,tty 核心不会将 tty 驱动程序的写入拆分为正常大小的块。此值不应用于通过向端口发送大量数据来防止对 tty 端口的拒绝服务攻击。 struct tty_flip_buffer 翻转； tty 设备的翻转缓冲区。结构 tty_ldisc ldisc; tty 设备的线路规程。

wait_queue_head_t 写等待； tty 写入函数的wait_queue 。 tty 驱动程序应该唤醒它,以便在它可以接收更多数据时发出信号。结构 termios *termios;指向 tty 设备的当前 termios 设置的指针。

无符号字符停止:1;
指示 tty 设备是否已停止。 tty 驱动程序可以设置这个值。

无符号字符 hw_stopped:1;
指示 tty 设备的硬件是否停止。 tty 驱动程序可以设置这个值。

无符号字符低延迟:1;
指示 tty 设备是否是低延迟设备,能够以非常高的速度接收数据。 tty 驱动程序可以设置这个值。

无符号字符关闭:1;指示 tty 设
备是否正在关闭端口。 tty 驱动程序可以设置这个值。

结构 tty_driver 驱动程序;控
制此 tty 设备的当前tty_driver结构。无效*驱动程序数据； tty_driver
可用于存储 tty 驱动程序本地数据的指针。 tty 核心不修改此变量。

快速参考

本节为本章介绍的概念提供参考。它还解释了 tty 驱动程序需要包含的每个头文件的作用。但是，tty_driver和tty_device结构中的字段列表在此不再赘述。

#include <linux/tty_driver.h>包含
struct tty_driver定义的头文件，并声明了该结构中使用的一些不同标志。

#include <linux/tty.h>包
含struct tty_struct 定义和许多不同宏的头文件，可以轻松访问struct termios字段的各个值。它还包含 tty 驱动程序核心的函数声明。

#include <linux/tty_flip.h>
包含一些 tty 翻转缓冲区内联函数的头文件，可以更轻松地操作翻转缓冲区结构。

#include <asm/termios.h>
包含构建内核的特定硬件平台的struct termio定义的头文件。 struct tty_driver
*alloc_tty_driver(int lines);创建struct tty_driver的函数，稍后可以将其传递给
tty_register_driver和tty_unregister_driver函数。

无效 put_tty_driver(struct tty_driver *driver);清理尚
未成功向 tty 核心完全注册的struct tty_driver结构的函数。 void
tty_set_operations(struct tty_driver *driver, struct tty_operations *op);初始化
struct tty_driver的函数回调的函数。这是在调用tty_register_driver之前必须调用的。

int tty_register_driver(struct tty_driver *driver); int
tty_unregister_driver(struct tty_driver *driver);
从 tty 核心注册和注销 tty 驱动程序的函数。

void tty_register_device(struct tty_driver *driver, unsigned minor, struct device
*device);
void tty_unregister_device(struct tty_driver *driver, unsigned minor);向 tty 核
心注册和注销单个 tty 设备的函数。
void tty_insert_flip_char(struct tty_struct *tty, unsigned char ch, char flag);

将字符插入 tty 设备的翻转缓冲区以供 a 读取的函数
用户。

TTY_NORMAL
TTY_BREAK TTY_FRAME TTY_PARITY TTY_OVERRUN tty_insert_flip_char函数中
使用的标志参数的不同值。 int tty_get_baud_rate(struct tty_struct *tty);
获取当前为特定 tty 设备设置的波特率的函数。无效 tty_flip_buffer_push(struct
tty_struct *tty);将当前翻转缓冲区中的数据推送给用户的函数。 tty_std_termios
使用一组通用默认行设置初始化 termios 结构的变量。