# Linux 设备模型之终端设备（tty）驱动架构分析

**********



这里感谢：晓刚的分析和指点，及网上的所有串口资源

**********

## 一 ： 前言

终端设备

在 Linux 系统中，终端是一种字符型设备，它有多种类型，通常使用 tty 来简称各种类型的终端设备。tty 是 Teletype 的缩写，Teletype 是最早出现的一种终端设备，很像电传打字机，是由 Teletype 公司生产的。Linux 中包含如下几类终端设备：

1. 串行端口终端（/dev/ttySn）

   串行端口终端（Serial Port Terminal）是使用计算机串行端口连接的终端设备。

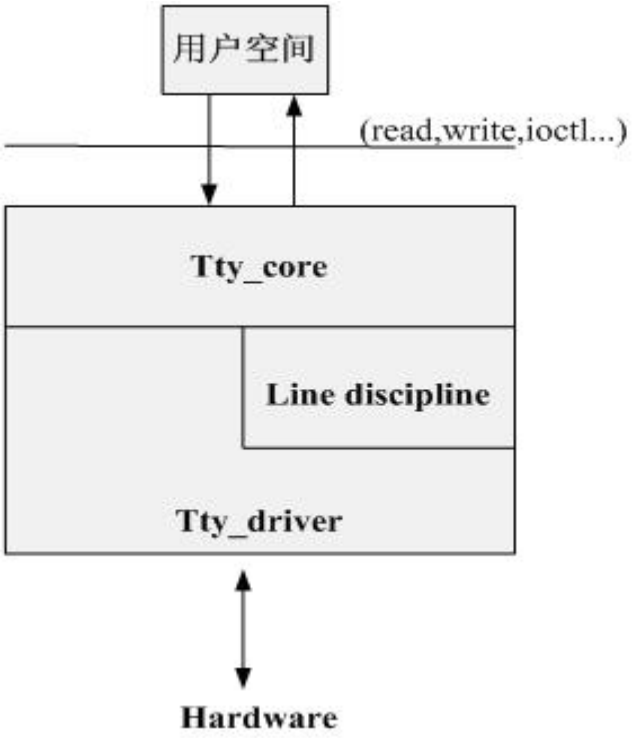
2. 伪终端（/dev/pty/）

   伪终端（Pseudo Terminal）是成对的逻辑终端设备

3. 控制台终端（/dev/ttyn, /dev/console）

## 二：终端设备驱动结构

1. **Tty 架构如下所示：**




**Tty_core:** Tty 核心层

**Line discipline :** 是线路规程的意思(链路层)。正如它的名字一样，它表示的是这条终端”线程”的输入与输出规范设置.主要用来进行输入/输出数据的预处理

**Tty_driver:** Tty_driver就是终端对应的驱动了。它将字符转换成终端可以理解的字串.将其传给终端设备。

**tty 设备发送数据的流程为：**

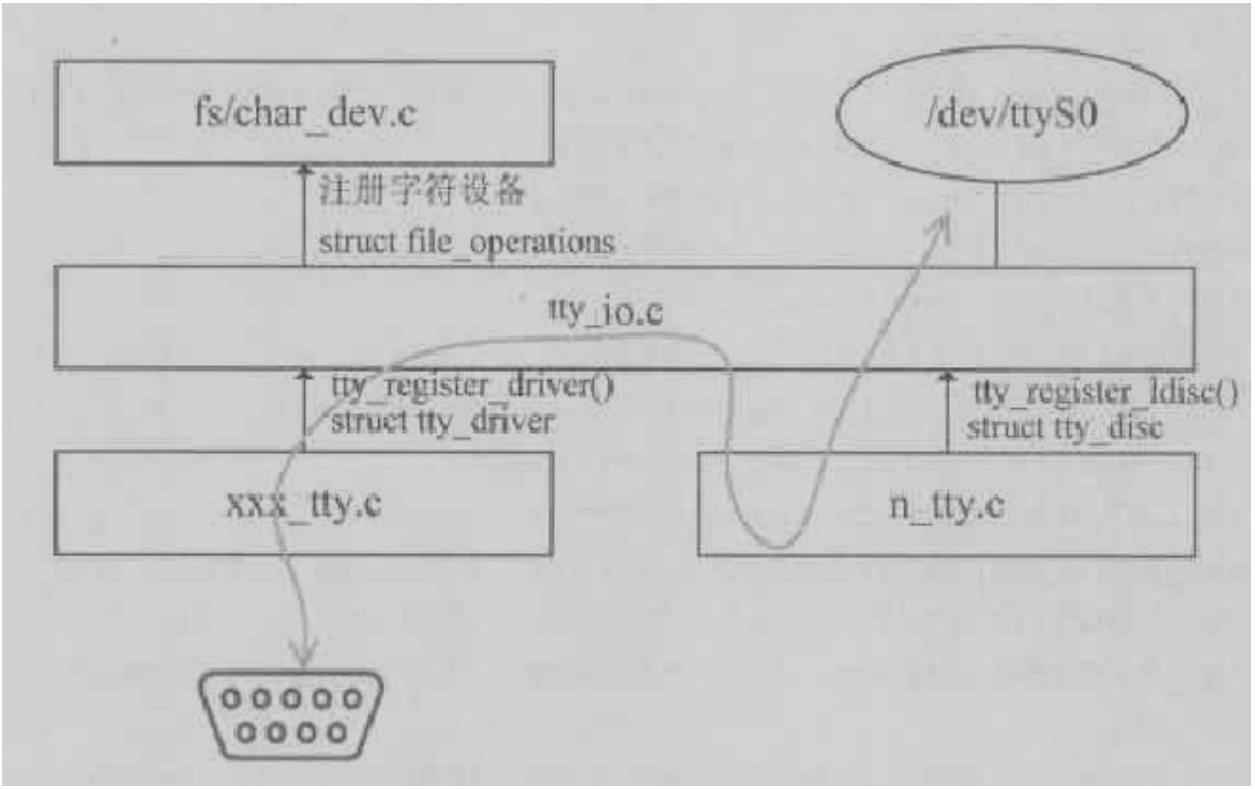
tty_core 从一个用户获得将要发送给一个 tty 设备的数据，tty_core 将数据传递给 Line discipline 处理，接着数据被传递到 tty_driver，tty 驱动将数据转换为可以发送给硬件的格式

**tty 设备接受数据的流程为：**

从 tty 硬件接收到得数据向上交给 tty_driver，进入 Line discipline 处理，再进入 tty_core，在这里它被一个用户获取

尽管大多数时候 tty_core 和 tty_driver 之间的数据传输会经历 Line discipline 的转换，但是 tty_core 和 tty_driver 之间也可以直接传输数据

**2.tty 主要源文件关系及数据流向：**




图二：tty 主要源文件关系及数据流向

上图显示了与 tty 相关的主要源文件及数据流向。

**tty_io.c：**tty_io.c 定义了 tty 设备通用的的 file_operation 结构体，并实现了接口函数 tty_register_driver()用于注册 tty 设备，它会利用 fs/char_dev.c 提供的接口函数注册字符设备。tty_io.c 也提供了 tty_register_ldisc()接口函数，用于注册线路规程。

**xxx_tty.c:**与具体设备对应的 tty 驱动(xxx_tty.c)将实现 tty_driver 结构体中的成员函数。

**Ntty.c：**ntty.c 文件则实现了 tty_disc 结构体中的成员

特定 tty 设备的主体工作是填充 tty_driver 结构体中的成员，实现其中的成员函数，tty_driver 结构体的源码（linux/driver/char/tty_driver.h）如下：

```
struct tty_driver {
   int magic;      /* magic number for this structure */
   struct cdev cdev;
   struct module   *owner;
   const char  *driver_name;
   const char  *devfs_name;
   const char  *name;
   int name_base;  /* offset of printed name */
   int major;      /* major device number */
   int minor_start;    /* start of minor device number */
   int minor_num;  /* number of *possible* devices */
   int num;        /* number of devices allocated */
   short   type;       /* type of tty driver */
   short   subtype;    /* subtype of tty driver */
   struct termios init_termios; /* Initial termios */
   int flags;      /* tty driver flags */
   int refcount;   /* for loadable tty drivers */
   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 termios **termios;
   struct termios **termios_locked;
   void *driver_state; /* only used for the PTY driver */

   /*
    * Interface routines from the upper tty layer to the tty
    * driver.  Will be replaced with struct tty_operations.
    */
   int  (*open)(struct tty_struct * tty, struct file * filp);
   void (*close)(struct tty_struct * tty, struct file * filp);
   int  (*write)(struct tty_struct * tty,
                 const unsigned char *buf, int count);
   void (*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, struct file * file,
            unsigned int cmd, unsigned long arg);
   void (*set_termios)(struct tty_struct *tty, struct termios * 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);
        void (*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 (*read_proc)(char *page, char **start, off_t off,
                  int count, int *eof, void *data);
        int (*write_proc)(struct file *file, const char __user *buffer,
                  unsigned long count, void *data);
        int (*tiocmget)(struct tty_struct *tty, struct file *file);
        int (*tiocmset)(struct tty_struct *tty, struct file *file,
                unsigned int set, unsigned int clear);

        struct list_head tty_drivers;
    };
```

## 三： tty 驱动接口分析

### 1.分配 tty 驱动

tty driver 的所有操作都包含在 tty_driver 中。内核即供了一个名叫 alloc_tty_driver() 来分配这个 tty_driver。当然我们也可以在自己的驱动中将它定义成一个静态的结构。对 tty_driver 进行一些必要的初始化之后，调用 tty_register_driver()将其注册.

```
struct tty_driver *alloc_tty_driver(int lines)
{
```

alloc_tty_driver()接口代码（Linux/driver/char/tty_io.c）如下所示：

```
struct tty_driver *alloc_tty_driver(int lines)
{
    struct tty_driver *driver;

    driver = kmalloc(sizeof(struct tty_driver), GFP_KERNEL);
    if (driver) {
        memset(driver, 0, sizeof(struct tty_driver));
        driver->magic = TTY_DRIVER_MAGIC;
        driver->num = lines;
        /* later we'll move allocation of tables here */
    }
    return driver;
}
```

这个函数只有一个参数。这个参数的含义为 line 的个数。也即次设备号的个数。注意每个设备文件都会对应一个 line.
在这个接口里为 tty_driver 分配内存，然后将 driver->magic 与 driver->num 初始化之后就返回了.其中 driver->magic 为 tty_driver 这个结构体的“幻数”，设为 TTY_DRIVER_MAGIC，在这里被初始化。

**2.注册 tty 驱动**

```
 */
int tty_register_driver(struct tty_driver *driver)
{
```

在这里， tty_register_driver() 用来注册一个 tty_driver, 源码如下（Linux/driver/char/tty_io.c）：

```
int tty_register_driver(struct tty_driver *driver)
{
    int error;
        int i;
    dev_t dev;
    void **p = NULL;

    if (driver->flags & TTY_DRIVER_INSTALLED)
        return 0;

    //TTY_DRIVER_DEVPTS_MEM:使用 devpts 进行动态内存映射
    if (!(driver->flags & TTY_DRIVER_DEVPTS_MEM)) {
        p = kmalloc(driver->num * 3 * sizeof(void *), GFP_KERNEL);
        if (!p)
            return -ENOMEM;
        memset(p, 0, driver->num * 3 * sizeof(void *));
    }

    // /如果没有指定 driver->major,注册字符设备号
    if (!driver->major) {
        error = alloc_chrdev_region(&dev, driver->minor_start, driver->num,
                        (char*)driver->name);
        if (!error) {
            driver->major = MAJOR(dev);
            driver->minor_start = MINOR(dev);
        }
    } else {
        dev = MKDEV(driver->major, driver->minor_start);
        error = register_chrdev_region(dev, driver->num,
                        (char*)driver->name);
    }
```

```
    if (error < 0) {
        kfree(p);
        return error;
    }

    if (p) {
        driver->ttys = (struct tty_struct **)p;
        driver->termios = (struct termios **)(p + driver->num);
        driver->termios_locked = (struct termios **)(p + driver->num * 2);
    } else {
        driver->ttys = NULL;
        driver->termios = NULL;
        driver->termios_locked = NULL;
    }
     //注册字符设备
    cdev_init(&driver->cdev, &tty_fops);
    driver->cdev.owner = driver->owner;
    error = cdev_add(&driver->cdev, dev, driver->num);
    if (error) {
        cdev_del(&driver->cdev);
        unregister_chrdev_region(dev, driver->num);
        driver->ttys = NULL;
        driver->termios = driver->termios_locked = NULL;
        kfree(p);
        return error;
    }
      //指定默认的 put_char
    if (!driver->put_char)
        driver->put_char = tty_default_put_char;

    list_add(&driver->tty_drivers, &tty_drivers);

   //如果没有指定 TTY_DRIVER_DYNAMIC_DEV.即动态设备管理
    if ( !(driver->flags & TTY_DRIVER_NO_DEVFS) ) {
        for(i = 0; i < driver->num; i++)
            tty_register_device(driver, i, NULL);
    }
    proc_tty_register_driver(driver);
    return 0;
}
```

注册 tty 驱动成功时返回 0;参数为由 alloc_tty_driver()分配的 tty_driver 结构体指针。这个函数操作比较简单。就是为 tty_driver 创建字符设备。然后将字符设备的操作集指定为 tty_fops.并且将 tty_driver 挂载到 tty_drivers 链表中。以设备号为关键字找到对应的 driver.

## 四：设备文件的打开操作

### 1. 打开 tty 设备的操作

在下面得程序里，我们会遇到 tty_struct 结构体，tty_struct 结构体被 tty 核心用来保存当前 tty 端口的状态，它的大多数成员只被 tty 核心使用。

从注册的过程可以看到，所有的操作都会对应到 tty_fops 中。Open 操作对应的操作接口是 tty_open()。

```
static int tty_open(struct inode * inode, struct file * filp)
```

代码如下（linux/drivers/char/tty_io.c）：

```
static int tty_open(struct inode * inode, struct file * filp)
{
	struct tty_struct *tty;
	int noctty, retval;
	struct tty_driver *driver;
	int index;
	dev_t device = inode->i_rdev;
	unsigned short saved_flags = filp->f_flags;

	nonseekable_open(inode, filp);

retry_open:

/*O_NOCTTY  如果路径名指向终端设备，不要把这个设备用作控制端
noctty:需不需要更改当前进程的控制终端*/

	noctty = filp->f_flags & O_NOCTTY;
	index  = -1;
	retval = 0;

/*O_NOCTTY  如果路径名指向终端设备，不要把这个设备用作控制终端*/

	down(&tty_sem);
//设备号(5,0)  即/dev/tty.表示当前进程的控制终端
	if (device == MKDEV(TTYAUX_MAJOR,0)) {
		if (!current->signal->tty) {
			//如果当前进程的控制终端不存在,退出
			up(&tty_sem);
			return -ENXIO;
		}
		//取得当前进程的 tty_driver
		driver = current->signal->tty->driver;
		index = current->signal->tty->index;
```

```
            filp->f_flags |= O_NONBLOCK; /* Don't let /dev/tty block */
            /* noctty = 1; */
            goto got_driver;
        }
#ifdef CONFIG_VT
//打开的设备节点是否是当前的控制终断/dev/ttys0
//(4,0)
        if (device == MKDEV(TTY_MAJOR,0)) {
            extern struct tty_driver *console_driver;
            driver = console_driver;
                //fg_console: 表示当前的控制台
            index = fg_console;
            noctty = 0;
            goto got_driver;
        }
#endif
//设备号(5,1).即/dev/console.表示外接的控制台. 通过 regesit_console()
        if (device == MKDEV(TTYAUX_MAJOR,1)) {
            driver = console_device(&index);
            if (driver) {
                /* Don't let /dev/console block */
                filp->f_flags |= O_NONBLOCK;
                noctty = 0;
                goto got_driver;
            }
            up(&tty_sem);
            return -ENODEV;
        }
       //index 就是文件的设备号
        //以文件的设备号为关键字,到 tty_drivers 中搜索所注册的 driver
        driver = get_tty_driver(device, &index);
        if (!driver) {
            up(&tty_sem);
            return -ENODEV;
        }
got_driver:
        retval = init_dev(driver, index, &tty);
        up(&tty_sem);
        if (retval)
            return retval;

        filp->private_data = tty;
        file_move(filp, &tty->tty_files);
/*
```

```
*这儿我们来看一下 file_move 函数，
*void file_move(struct file *file, struct list_head *list)//(linux/fs.h)
*{
*     if (!list)
*          return;
*     file_list_lock();
*     list_move(&file->f_u.fu_list, list);
*     file_list_unlock();
*}
*其中调用了 list_move() ；
*static inline void list_move(struct list_head *list, struct list_head *head)
*{
*          __list_del(list->prev, list->next);
*          list_add(list, head);
*}
*_list_del()
*在了解数据结构的基础上，_list_del()很好理解，通过
*     next->prev = prev;
*     prev->next = next;
*来删除链表中的一个节点，其中 prev 和 next 是节点的 2 个指针，分别指向链表中的上一个
*节点和下一个节点。
*还调用了_list_add();
*下面继续看一下_list_add(struct list_head *new,struct list_head *prev, Struct list_head *next))
*是如何实现的
*     next->prev = new;
*     new->next = next;
*     new->prev = prev;
*     prev->next = new;
*通过上面的代码来将节点 new 加入到节点 prev 和 next 中间
*那么 file_move 的作用就是将 file 结构体中的一些成员拷贝到 tty->tty_file 中去
*/

/*
下面来看一下 check_tty_count();
主要是查看一下 tty_file 链表中成员数
*/

     check_tty_count(tty, "tty_open");
     if (tty->driver->type == TTY_DRIVER_TYPE_PTY &&
          tty->driver->subtype == PTY_TYPE_MASTER)
          noctty = 0;
#ifdef TTY_DEBUG_HANGUP
     printk(KERN_DEBUG "opening %s...", tty->name);
#endif
```

```
        if (!retval) {
                if (tty->driver->open)
/*
*tty 核心调用 tty 驱动的 open 回调函数
*/
                        retval = tty->driver->open(tty, filp);
                else
                        retval = -ENODEV;
        }
        filp->f_flags = saved_flags;
//test_bit(n,addr)用于检测地址中数据的某一位是否为 1

        if (!retval && test_bit(TTY_EXCLUSIVE, &tty->flags) && !capable(CAP_SYS_ADMIN))
                retval = -EBUSY;
//Capable()检测进程的能力用户违反 LIDS 定义的规则的时候报警

         if (retval) {
#ifdef TTY_DEBUG_HANGUP
                printk(KERN_DEBUG "error %d in opening %s...", retval,
                        tty->name);
#endif
                release_dev(filp);
                if (retval != -ERESTARTSYS)
                        return retval;
                if (signal_pending(current))
                        return retval;

//signal_pending(current)检查当前进程是否有信号处理，返回不为0表示有信号需要处理,这儿
如果有信号需要处理，函数返回。

                schedule();

//选定下一个进程并切换到它去执行是通过 schedule()函数实现的。


                /*
                 * Need to reset f_op in case a hangup happened.
                 */
                if (filp->f_op == &hung_up_tty_fops)
                        filp->f_op = &tty_fops;
                goto retry_open;
        }
        if (!noctty &&
            current->signal->leader &&
            !current->signal->tty &&
```

```
            tty->session == 0) {
                task_lock(current);
            current->signal->tty = tty;
            task_unlock(current);
            current->signal->tty_old_pgrp = 0;
            tty->session = current->signal->session;
            tty->pgrp = process_group(current);
        }
        return 0;
}
```

上面的过程中我们漏掉了 init_dev 函数，

```
static int init_dev(struct tty_driver *driver, int idx,
        struct tty_struct **ret_tty)
```

下面我们继续解析一下：

```
static int init_dev(struct tty_driver *driver, int idx, struct tty_struct **ret_tty)
{
        struct tty_struct *tty, *o_tty;
        struct termios *tp, **tp_loc, *o_tp, **o_tp_loc;
        struct termios *ltp, **ltp_loc, *o_ltp, **o_ltp_loc;
        int retval=0;

        /* check whether we're reopening an existing tty */
        if (driver->flags & TTY_DRIVER_DEVPTS_MEM) {
                tty = devpts_get_tty(idx);
                if (tty && driver->subtype == PTY_TYPE_MASTER)
                        tty = tty->link;
                } else {
                tty = driver->ttys[idx];
        }
        if (tty) goto fast_track;

        /*
          * First time open is complex, especially for PTY devices.
          * This code guarantees that either everything succeeds and the
          * TTY is ready for operation, or else the table slots are vacated
          * and the allocated memory released.    (Except that the termios
          * and locked termios may be retained.)
          */
/*
```

try_module_get() -- 如果模块已经插入内核，则递增该模块引用计数；如果该模块还没有插入内核，则返回 0 表示出错。

Driver->owner 是指这个驱动模块的拥有者

```
*/
        if (!try_module_get(driver->owner)) {
                retval = -ENODEV;
                goto end_init;
        }//如果模块没有加入内核，那么退出

        o_tty = NULL;
        tp = o_tp = NULL;
        ltp = o_ltp = NULL;

        tty = alloc_tty_struct();//分配结构体空间
        if(!tty)
                goto fail_no_mem;
        initialize_tty_struct(tty);//初始化结构体中的各成员变量
        tty->driver = driver;
        tty->index = idx;
        tty_line_name(driver, idx, tty->name);//打印一些提示信息
/*
我们知道伪终端(pty)也属于终端设备的一种
 * TTY_DRIVER_DEVPTS_MEM -- don't use the standard arrays, instead
 *    use dynamic memory keyed through the devpts filesystem.    This
 *    is only applicable to the pty driver.
所以下面是对于伪终端的一些初始化设置？
看下面的设置应该是关于控制台终端的初始化
*/
        if (driver->flags & TTY_DRIVER_DEVPTS_MEM) {
                tp_loc = &tty->termios;
                ltp_loc = &tty->termios_locked;
        } else {
                tp_loc = &driver->termios[idx];//根据索引号到数组中查找对应的 termios 结构体
                ltp_loc = &driver->termios_locked[idx];
        }
//如果没有查找到对应的 termios 结构体，那么我们就手动分配一个
        if (!*tp_loc) {
                tp = (struct termios *) kmalloc(sizeof(struct termios),
                                                GFP_KERNEL);
                if (!tp)
                        goto free_mem_out;
                *tp = driver->init_termios;//用统一的 init_termios 结构体来初始化它
        }

        if (!*ltp_loc) {//同上
                ltp = (struct termios *) kmalloc(sizeof(struct termios),
                                                 GFP_KERNEL);
```

```
            if (!ltp)
                goto free_mem_out;
            memset(ltp, 0, sizeof(struct termios));
        }

        if (driver->type == TTY_DRIVER_TYPE_PTY) {
            o_tty = alloc_tty_struct();
            if (!o_tty)
                goto free_mem_out;
            initialize_tty_struct(o_tty);
/*
如果是 PTY，那么 o_tty->driver = driver->other;
查看 tty_driver 中对于 other 的定义
*/
            o_tty->driver = driver->other;
            o_tty->index = idx;
            tty_line_name(driver->other, idx, o_tty->name);//打印一些提示信息

            if (driver->flags & TTY_DRIVER_DEVPTS_MEM) {
                o_tp_loc = &o_tty->termios;
                o_ltp_loc = &o_tty->termios_locked;
            } else {
                o_tp_loc = &driver->other->termios[idx];
                o_ltp_loc = &driver->other->termios_locked[idx];
            }

            if (!*o_tp_loc) {//如果 o_tp_loc = &o_tty->termios 不存在，那么我们分配一个 termios
结构体
                o_tp = (struct termios *)
                    kmalloc(sizeof(struct termios), GFP_KERNEL);
                if (!o_tp)
                    goto free_mem_out;
                *o_tp = driver->other->init_termios;
            }

            if (!*o_ltp_loc) {
                o_ltp = (struct termios *)
                    kmalloc(sizeof(struct termios), GFP_KERNEL);
                if (!o_ltp)
                    goto free_mem_out;
                memset(o_ltp, 0, sizeof(struct termios));
            }

            /*
```

```
         * Everything allocated ... set up the o_tty structure.
         */
        if (!(driver->other->flags & TTY_DRIVER_DEVPTS_MEM)) {
            driver->other->ttys[idx] = o_tty;//将我们初始化的的伪终端的 tty_struct 结构体添加到数组中去
        }
        if (!*o_tp_loc)//如果 o_tp_loc 还是不存在，那么我用 o_tp 来初始化 o_tp_loc，不过为什么会不存在呢？
上面已经很系统的处理的一次~~~
            *o_tp_loc = o_tp;
        if (!*o_ltp_loc)
            *o_ltp_loc = o_ltp;
        o_tty->termios = *o_tp_loc;
        o_tty->termios_locked = *o_ltp_loc;
        driver->other->refcount++;
        if (driver->subtype == PTY_TYPE_MASTER)
            o_tty->count++;

        /* Establish the links in both directions */
        tty->link      = o_tty;
        o_tty->link = tty;
    }

    /*
     * All structures have been allocated, so now we install them.
     * Failures after this point use release_mem to clean up, so
     * there's no need to null out the local pointers.
     */
    if (!(driver->flags & TTY_DRIVER_DEVPTS_MEM)) {
        driver->ttys[idx] = tty;
    }

    if (!*tp_loc)
        *tp_loc = tp;
    if (!*ltp_loc)
        *ltp_loc = ltp;
    tty->termios = *tp_loc;
    tty->termios_locked = *ltp_loc;
    driver->refcount++;
    tty->count++;

    /*
     * Structures all installed ... call the ldisc open routines.
     * If we fail here just call release_mem to clean up.    No need
```

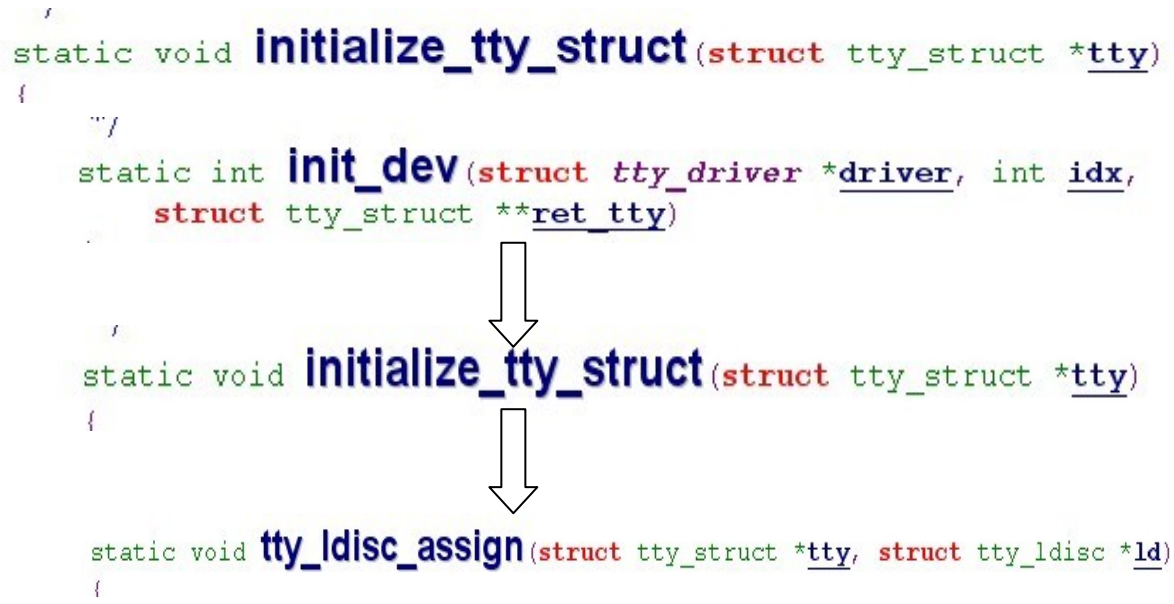
```
         * to decrement the use counts, as release_mem doesn't care.
         */

        if (tty->ldisc.open) {
//如果函数指针 tty_ldisc.open 存在，调用之
//线路规程的定义处于/linux/dirver/char/n_tty.c 中
                retval = (tty->ldisc.open)(tty);
                if (retval)
                        goto release_mem_out;
        }
        if (o_tty && o_tty->ldisc.open) {
                retval = (o_tty->ldisc.open)(o_tty);
                if (retval) {
                        if (tty->ldisc.close)
                                (tty->ldisc.close)(tty);
                        goto release_mem_out;
                }
                tty_ldisc_enable(o_tty);
/*
static void tty_ldisc_enable(struct tty_struct *tty)
{
        set_bit(TTY_LDISC, &tty->flags);
如果线路规程已经附加上去，那么唤醒队列
        wake_up(&tty_ldisc_wait);
}
*/
        }
        tty_ldisc_enable(tty);
        goto success;

        /*
         * This fast open can be used if the tty is already open.
         * No memory is allocated, and the only failures are from
         * attempting to open a closing tty or attempting multiple
         * opens on a pty master.
         */
fast_track:
        if (test_bit(TTY_CLOSING, &tty->flags)) {
                retval = -EIO;
                goto end_init;
        }
        if (driver->type == TTY_DRIVER_TYPE_PTY &&
                driver->subtype == PTY_TYPE_MASTER) {
                /*
```

```
                 * special case for PTY masters: only one open permitted,
                 * and the slave side open count is incremented as well.
                 */
                if (tty->count) {
                        retval = -EIO;
                        goto end_init;
                }
                tty->link->count++;
        }
        tty->count++;
        tty->driver = driver; /* N.B. why do this every time?? */

        /* FIXME */
        if(!test_bit(TTY_LDISC, &tty->flags))
                printk(KERN_ERR "init_dev but no ldisc\n");
success:
        *ret_tty = tty;

        /* All paths come through here to release the semaphore */
end_init:
        return retval;

        /* Release locally allocated memory ... nothing placed in slots */
free_mem_out:
        kfree(o_tp);
        if (o_tty)
                free_tty_struct(o_tty);
        kfree(ltp);
        kfree(tp);
        free_tty_struct(tty);

fail_no_mem:
        module_put(driver->owner);
        retval = -ENOMEM;
        goto end_init;

        /* call the tty release_mem routine to clean out this slot */
release_mem_out:
        printk(KERN_INFO "init_dev: ldisc open failed, "
                        "clearing slot %d\n", idx);
        release_mem(tty, idx);
        goto end_init;
}
```

上面关于的 **init_dev()**的大体函数调用关系如下：

Init_dev() -> initialize_tty_struct() -> tty_ldisc_assign(tty, tty_ldisc_get(N_TTY));

linux/drivers/char/tty_io.c

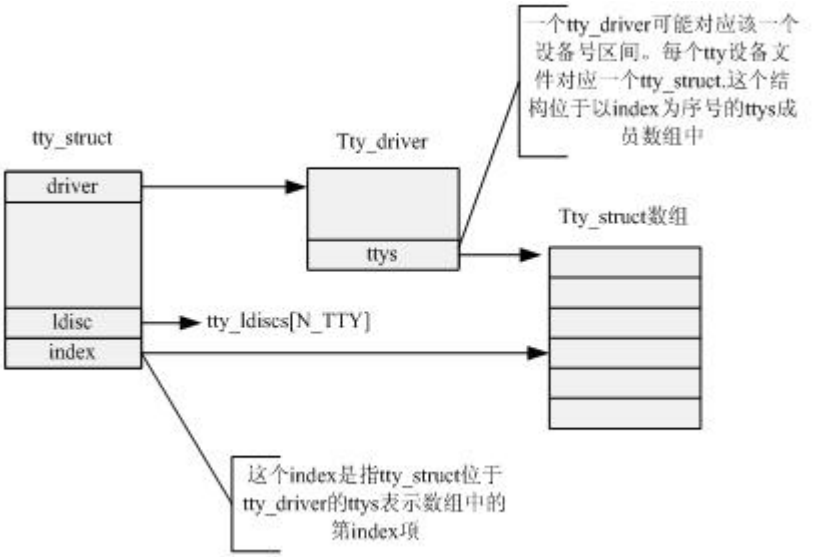



看一下tty_ldisc_assign(tty, tty_ldisc_get(N_TTY))的操作:

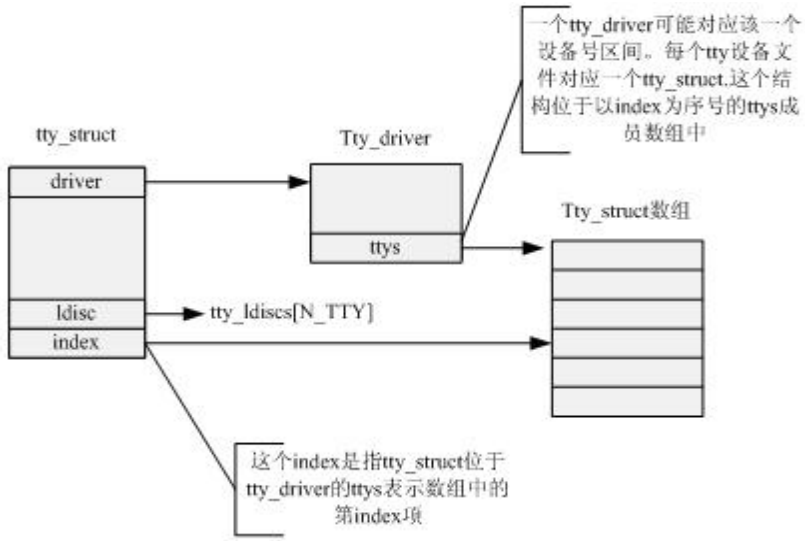
Tty_ldisc_get():

```
struct tty_ldisc *tty_ldisc_get(int disc)
{
        unsigned long flags;
        struct tty_ldisc *ld;

        if (disc < N_TTY || disc >= NR_LDISCS)
                return NULL;

        spin_lock_irqsave(&tty_ldisc_lock, flags);

        ld = &tty_ldiscs[disc];
        /* Check the entry is defined */
        if(ld->flags & LDISC_FLAG_DEFINED)
        {
                /* If the module is being unloaded we can't use it */
```

```
        if (!try_module_get(ld->owner))
                ld = NULL;
        else /* lock it */
                ld->refcount++;
    }
    else
        ld = NULL;
```

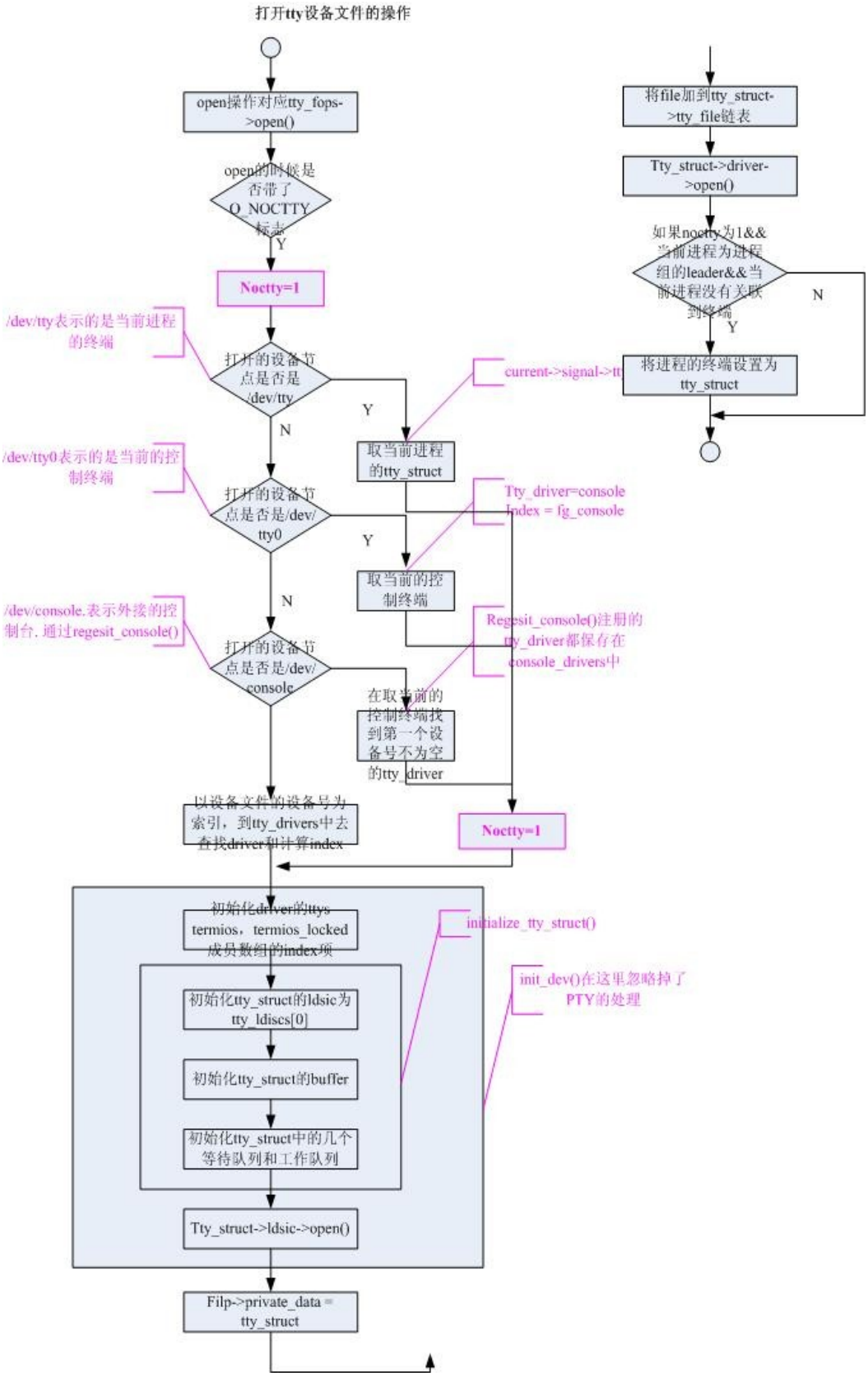



用 tty_register_ldisc()将在这段代码中涉及到了 tty_driver,tty_struct, struct tty_ldisc.这三者之间的关系用下图表示如下：




在这里,为tty_struct的ldisc是默认指定为tty_ldiscs[N_TTY].该ldisc对应的是控制终端的线路规范。可以在用空间用带TIOCSETD的ioctl调用进行更改.

**将上述 open 用流程图的方式表示如下：**


打开tty设备文件的操作
open操作对应tty_fops->open()
open的时候是否带了O_NOCTTY标志
Y
Noctty=1
/dev/tty表示的是当前进程的终端
打开的设备节点是否是/dev/tty
Y
N
current->signal->tt
取当前进程的tty_struct
/dev/tty0表示的是当前的控制终端
打开的设备节点是否是/dev/tty0
Y
N
Tty_driver=console
Index = fg_console
取当前的控制终端
/dev/console.表示外接的控制台. 通过regesit_console()
打开的设备节点是否是/dev/console
Regesit_console()注册的tty_driver都保存在console_drivers中
在取当前的控制终端找到第一个设备号不为空的tty_driver
以设备文件的设备号为索引，到tty_drivers中去查找driver和计算index
Noctty=1
初始化driver的ttys termios，termios_locked成员数组的index项
initialize_tty_struct()
init_dev()在这里忽略掉了PTY的处理
初始化tty_struct的ldsic为tty_ldiscs[0]
初始化tty_struct的buffer
初始化tty_struct中的几个等待队列和工作队列
Tty_struct->ldsic->open()
Filp->private_data = tty_struct
将file加到tty_struct->tty_file链表
Tty_struct->driver->open()
如果noctty为1&&当前进程为进程组的leader&&当前进程没有关联到终端
N
Y
将进程的终端设置为tty_struct

**下面分析一下：线路规划层的 open()函数（linux/driver/char/n_tty.c）**

```
static int n_tty_open(struct tty_struct *tty)
{
    if (!tty)//如果 tty_struct 结构体不存在
        return -EINVAL;
    /* This one is ugly. Currently a malloc failure here can panic */
    if (!tty->read_buf) {//如果 tty->read_buf 不存在，read_buf 是一个环形缓冲区
        tty->read_buf = alloc_buf();//分配一块内存
        if (!tty->read_buf)
            return -ENOMEM;
    }
    memset(tty->read_buf, 0, N_TTY_BUF_SIZE);//初始化全0
    reset_buffer_flags(tty);//初始化 tty 中的一些字段
check_unthrottle():是用来判断是否需要打开”阀门”,允许输入数据流入
    tty->column = 0;
    n_tty_set_termios(tty, NULL);
    tty->minimum_to_wake = 1;
    tty->closing = 0;
    return 0;
}
```

**1.tty_set_ldisc(),设置线路规程()**

```
static int tty_set_ldisc(struct tty_struct *tty, int ldisc)
{
```

```
static int tty_set_ldisc(struct tty_struct *tty, int ldisc)
{
    int retval = 0;
    struct tty_ldisc o_ldisc;
    char buf[64];
```

```
        int work;
        unsigned long flags;
        struct tty_ldisc *ld;
        struct tty_struct *o_tty;
        if ((ldisc < N_TTY) || (ldisc >= NR_LDISCS))
                return -EINVAL;
restart:
        ld = tty_ldisc_get(ldisc);
/*
从 tty_ldiscs[ ]找到对应项，这个数组中的成员是调用 tty_register_ldisc()将其设置进去的
*/
        if (ld == NULL) {//这儿 ld==NULL 是因为模块没有加载
                request_module("tty-ldisc-%d", ldisc);
/*
Request_module()是用在内核态加载内核的
int try_module_get(struct module *module); 用于增加模块使用计数；若返回为0，表示调用失败，希望使用的模块没有被加载或正在被卸载中。
void module_put(struct module *module); 减少模块使用计数。
从设备使用的角度出发，当需要打开、开始使用某个设备时，使用 try_module_get(dev->owner)去增加管理此设备的 owner 模块的使用计数；当关闭、不再使用此设备时，使用 module_put(dev->owner)减少对管理此设备的 owner 模块的使用 计数。这样，当设备在使用时，管理此设备的模块就不能被卸载；只有设备不再使用时模块才能被卸载。
void module_put(struct module *module); 减少模块使用计数。
*/
                ld = tty_ldisc_get(ldisc);
        }
        if (ld == NULL)//如果模块还没有加载，函数出错退出
                return -EINVAL;

        /*
         *    No more input please, we are switching. The new ldisc
         *    will update this value in the ldisc open function
```

```
         */
        tty->receive_room = 0;
        /*
         *    Problem: What do we do if this blocks ?
         */
        tty_wait_until_sent(tty, 0);//等待终端输出设备的数据发送完
        if (tty->ldisc.num == ldisc) {
/*
这儿需要注意 tty_ldisc_put()函数和 tty_ldisc_get（）函数；
根据个人理解，tty_ldisc_put()是用来减少模块使用计数器
因为我们下面需要重新设置线路规程，下面需要解除线路规程与模块的绑定
*/
                tty_ldisc_put(ldisc);
                return 0;
        }
//下面2行是用来处理伪终端的情况的
        o_ldisc = tty->ldisc;
        o_tty = tty->link;
        /*
         *    Make sure we don't change while someone holds a
         *    reference to the line discipline. The TTY_LDISC bit
         *    prevents anyone taking a reference once it is clear.
         *    We need the lock to avoid racing reference takers.
         */
        spin_lock_irqsave(&tty_ldisc_lock, flags);
        if (tty->ldisc.refcount || (o_tty && o_tty->ldisc.refcount)) {
                if(tty->ldisc.refcount) {
                        /* Free the new ldisc we grabbed. Must drop the lock
                            first. */
```

```
                spin_unlock_irqrestore(&tty_ldisc_lock, flags);//打开自旋锁
                tty_ldisc_put(ldisc);//继续减少使用计数器
                /*
                 * There are several reasons we may be busy, including
                 * random momentary I/O traffic. We must therefore
                 * retry. We could distinguish between blocking ops
                 * and retries if we made tty_ldisc_wait() smarter. That
                 * is up for discussion.
                 */
                if (wait_event_interruptible(tty_ldisc_wait, tty->ldisc.refcount == 0) < 0)
//程序在这儿进入睡觉状态，tty->ldisc.refcount==0条件可打破睡眠状态
                        return -ERESTARTSYS;
                goto restart;
            }
            if(o_tty && o_tty->ldisc.refcount) {
                spin_unlock_irqrestore(&tty_ldisc_lock, flags);
                tty_ldisc_put(ldisc);
                if (wait_event_interruptible(tty_ldisc_wait, o_tty->ldisc.refcount == 0) < 0)
                    return -ERESTARTSYS;
                goto restart;
            }
        }
/*
上面一段代码的作用就是减小模块使用计数器，直到 refcount==0
*/
        /* if the TTY_LDISC bit is set, then we are racing against another ldisc change */
/*
如果线路规程已经绑定，那么我们解除之，
*/
```

```
        if (!test_bit(TTY_LDISC, &tty->flags)) {//如果线路规程还没绑定
            spin_unlock_irqrestore(&tty_ldisc_lock, flags);
            tty_ldisc_put(ldisc);//以 ldisc 为下标从数组中获得一个初始化好的 tty_ldisc 结构体，然后将使用计数器 refcount 减1，如果使用计数器为0，则 tty_ldsic 没有绑定
/*
Tty_ldisc_ref_wait 中用到了 wait_event(tty_ldisc_wait, tty_ldisc_try(tty));
Tty_ldisc_try 用于检测一个 tty_ldisc 是否被绑定
Tty_ldisc_ref_wait 用于从 tty_ldisc_wait 队列中返回一个已经绑定的 tty_ldisc 结构体
*/
            ld = tty_ldisc_ref_wait(tty);//wait for the tty ldisc
/*
上面一步我们已经找出了那个与模块绑定的 tty_ldisc 结构体，下面一步我们需要解除绑定
*/
            tty_ldisc_deref(ld);//free a tty ldisc reference
            goto restart;
        }

        clear_bit(TTY_LDISC, &tty->flags);.//清除 flags 标志中的 TTY_LDISC 位
        clear_bit(TTY_DONT_FLIP, &tty->flags);
        if (o_tty) {//如果 o_tty 存在，也就是伪终端的情况
            clear_bit(TTY_LDISC, &o_tty->flags);
            clear_bit(TTY_DONT_FLIP, &o_tty->flags);
        }
        spin_unlock_irqrestore(&tty_ldisc_lock, flags);
        /*
         *    From this point on we know nobody has an ldisc
         *    usage reference, nor can they obtain one until
         *    we say so later on.
         */
        work = cancel_delayed_work(&tty->buf.work)；//用于取消一个工作队列
```

如果当一个取消操作的调用返回时,任务正在执行中，那么这个任务将继续执行下去，但不会再加入到队列中。

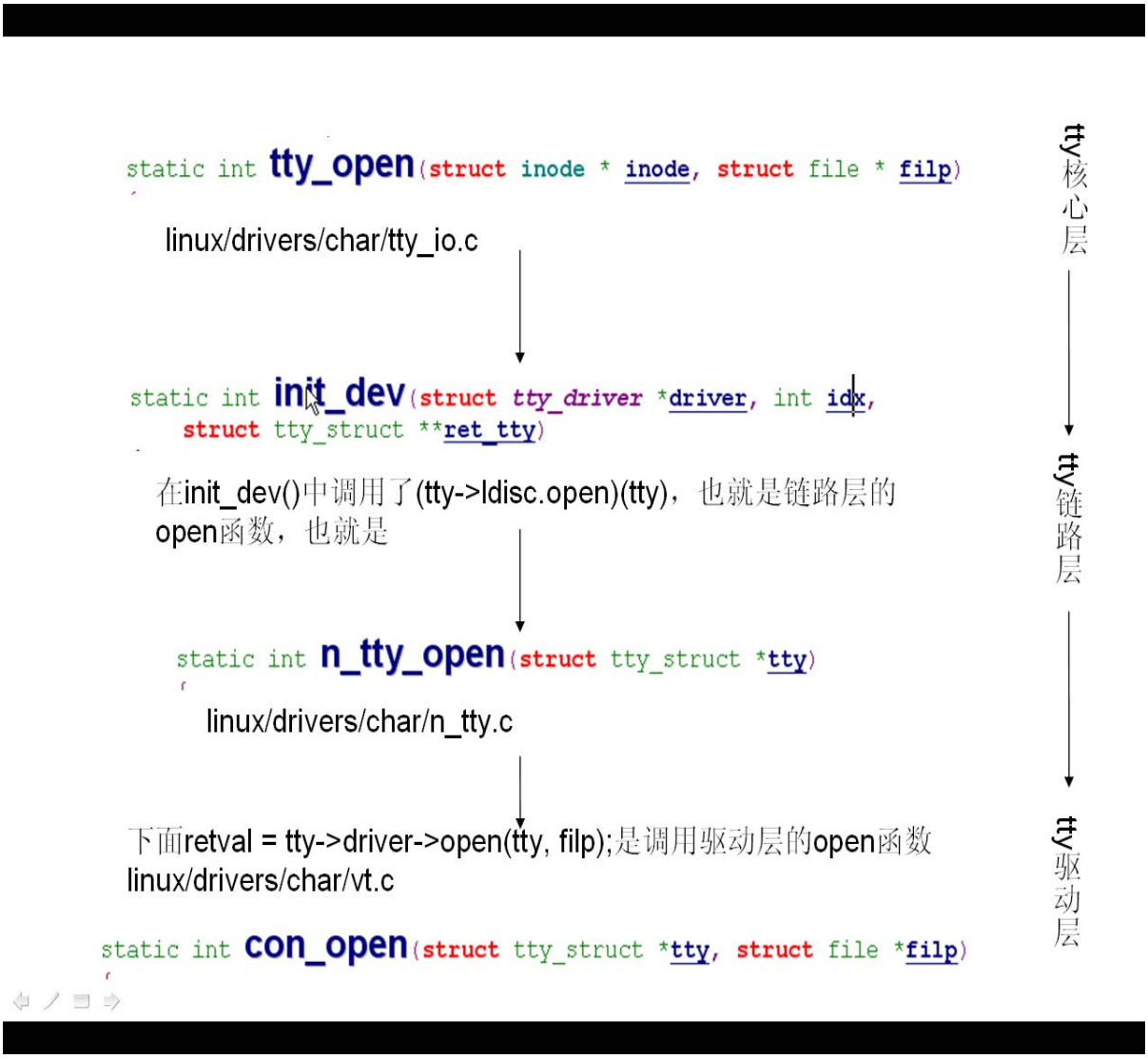
```
	/*
	 * Wait for ->hangup_work and ->buf.work handlers to terminate
	 */

	flush_scheduled_work();//清空工作队列中的所有任务使用：
	/* Shutdown the current discipline. */
	if (tty->ldisc.close)//关闭当前使用的线路规程，ldisc_close 和 ldisc_open 函数下面需要进一步分析
		(tty->ldisc.close)(tty);

	/* Now set up the new line discipline. */
//设置新的线路规程
	tty_ldisc_assign(tty, ld);
	tty_set_termios_ldisc(tty, ldisc);
	if (tty->ldisc.open)//打开线路规程
		retval = (tty->ldisc.open)(tty);
	if (retval < 0) {//如果打开失败的话
		tty_ldisc_put(ldisc);
		/* There is an outstanding reference here so this is safe */
		tty_ldisc_assign(tty, tty_ldisc_get(o_ldisc.num));
		tty_set_termios_ldisc(tty, tty->ldisc.num);
		if (tty->ldisc.open && (tty->ldisc.open(tty) < 0)) {
			tty_ldisc_put(o_ldisc.num);
			/* This driver is always present */
			tty_ldisc_assign(tty, tty_ldisc_get(N_TTY));
			tty_set_termios_ldisc(tty, N_TTY);
			if (tty->ldisc.open) {
```

```
                int r = tty->ldisc.open(tty);//再次尝试打开


                if (r < 0)
                    panic("Couldn't open N_TTY ldisc for "
                        "%s --- error %d.",
                        tty_name(tty, buf), r);
            }
        }
    }
    /* At this point we hold a reference to the new ldisc and a
       a reference to the old ldisc. If we ended up flipping back
       to the existing ldisc we hav two references to it */

    if (tty->ldisc.num != o_ldisc.num && tty->driver->set_ldisc)
        tty->driver->set_ldisc(tty);//调用驱动层的 set_ldisc 函数
    tty_ldisc_put(o_ldisc.num);
    /*
     *    Allow ldisc referencing to occur as soon as the driver
     *    ldisc callback completes.
     */
    tty_ldisc_enable(tty);//函数从函数指针返回就使能新的线路规程
    if (o_tty)
        tty_ldisc_enable(o_tty);


    /* Restart it in case no characters kick it off. Safe if
       already running */
    if (work)
        schedule_delayed_work(&tty->buf.work, 1);//向工作队列中添加一个任务并延迟执
行，上面在修改线路规程之前丢弃了一个工作队列，现在需要重新执行被丢弃的队列
    return retval;
```

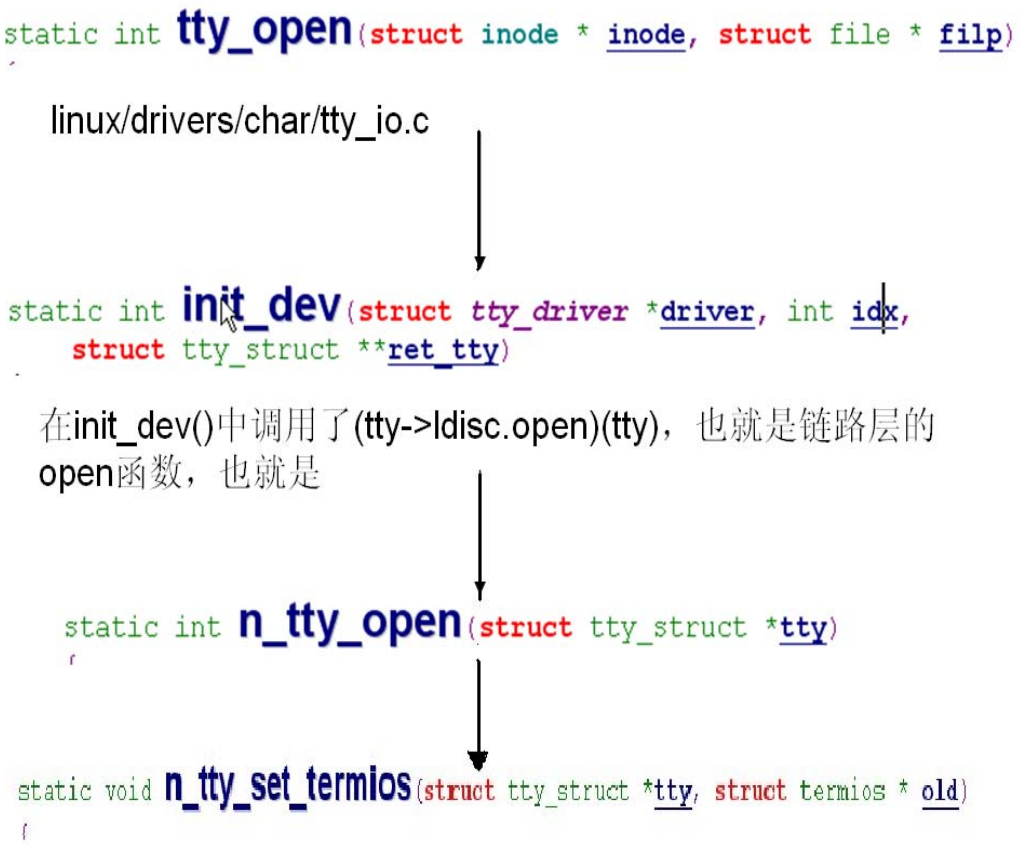
}

对于上面的理解还是有一些问题，需要继续读 ldisc_open 函数，也就是 n_tty_open 函数

Tty_open 操作的函数流程图如下：




对串口进行线路设置的调用关系如下：




这里对线路设置的控制室在链路层里，通过

n_tty_set_termios(struct tty_struct *tty, struct termios * old) 来进行设置的。

## 五：设备文件的 write/read 操作

终端设备数据的发送和接受过程中的数据流以及函数调用的关系。用户在数据发送给终端设备时，通过“write()→tty 核心→线路规划”的层层调用，最后调用 tty_driver 结构体中的 write()函数完成发送。

终端设备数据发送和接受过程中的数据流和函数调用关系：

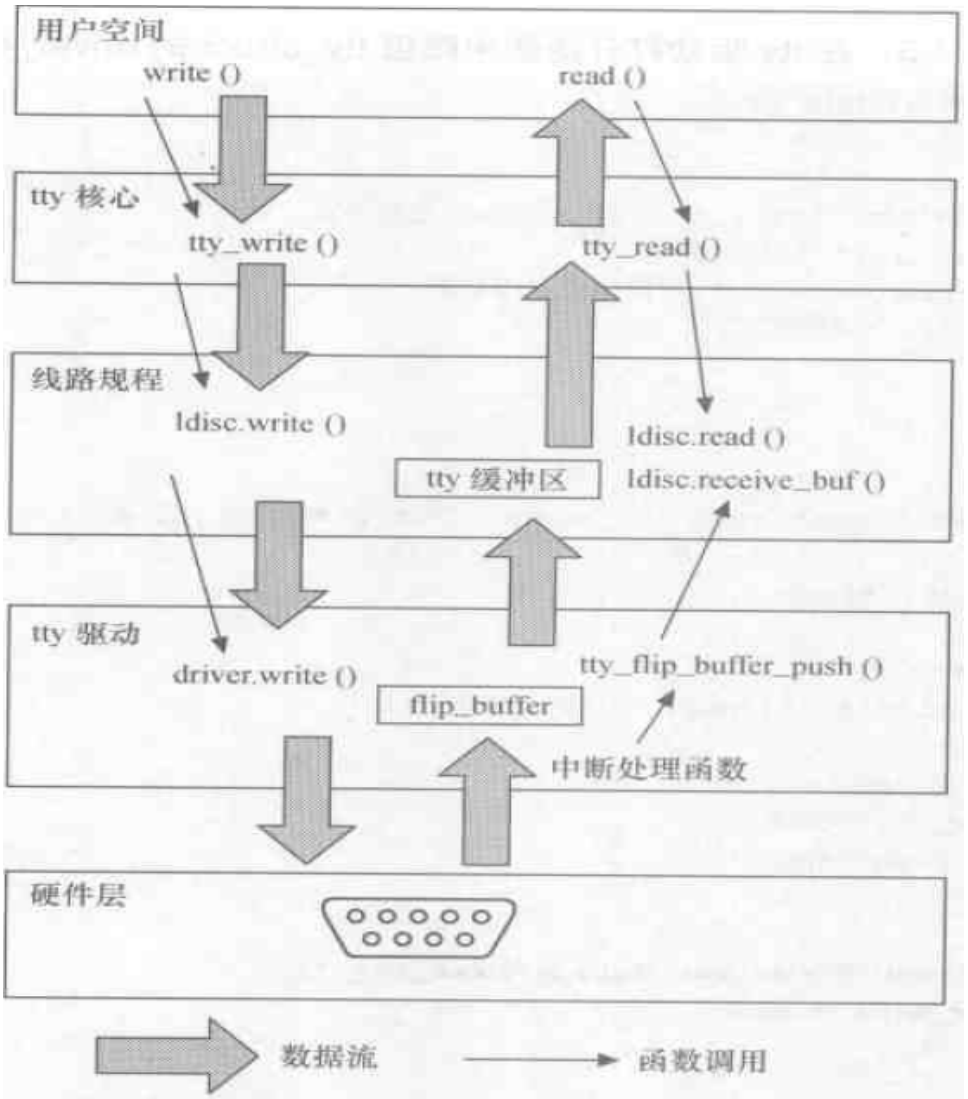



图 14.3 终端设备数据发送和接收过程中的数据流和函数调用关系

设备文件的 write 操作对应 tty_fops->write 即 tty_write(). 代码如下(linux/driver/char/tty_io.c):

```
static ssize_t tty_write(struct file * file, const char __user * buf, size_t count,
                         loff_t *ppos)
{
        struct tty_struct * tty;
        struct inode *inode = file->f_dentry->d_inode;
        ssize_t ret;
        struct tty_ldisc *ld;
```

```
    tty = (struct tty_struct *)file->private_data;
    if (tty_paranoia_check(tty, inode, "tty_write"))
        return -EIO;
    if (!tty || !tty->driver->write || (test_bit(TTY_IO_ERROR, &tty->flags)))
        return -EIO;

    ld = tty_ldisc_ref_wait(tty);
    if (!ld->write)
        ret = -EIO;
    else
        ret = do_tty_write(ld->write, tty, file, buf, count);
    tty_ldisc_deref(ld);
    return ret;
}
```

在open的过程中，将tty_struct存放在file的私有区。在write中,从file的私有区中就可以取到要操作的tty_struct. 如果tty_driver中没有write.如果tty有错误都会有效性判断失败返回。如果一切正常，递增ldsic的引用计数。将用do_tty_wirte()再行写操作。写完之后，再递减ldsic的引用计数.

Do_tty_write代码分段分析如下：

```
static inline ssize_t do_tty_write(
    ssize_t (*write)(struct tty_struct *, struct file *, const unsigned char *, size_t),
    struct tty_struct *tty,
    struct file *file,
    const char __user *buf,
    size_t count)
{
    ssize_t ret = 0, written = 0;
    unsigned int chunk;

    if (down_interruptible(&tty->atomic_write)) {
        return -ERESTARTSYS;
    }

    /*
     * We chunk up writes into a temporary buffer. This
     * simplifies low-level drivers immensely, since they
     * don't have locking issues and user mode accesses.
     *
```

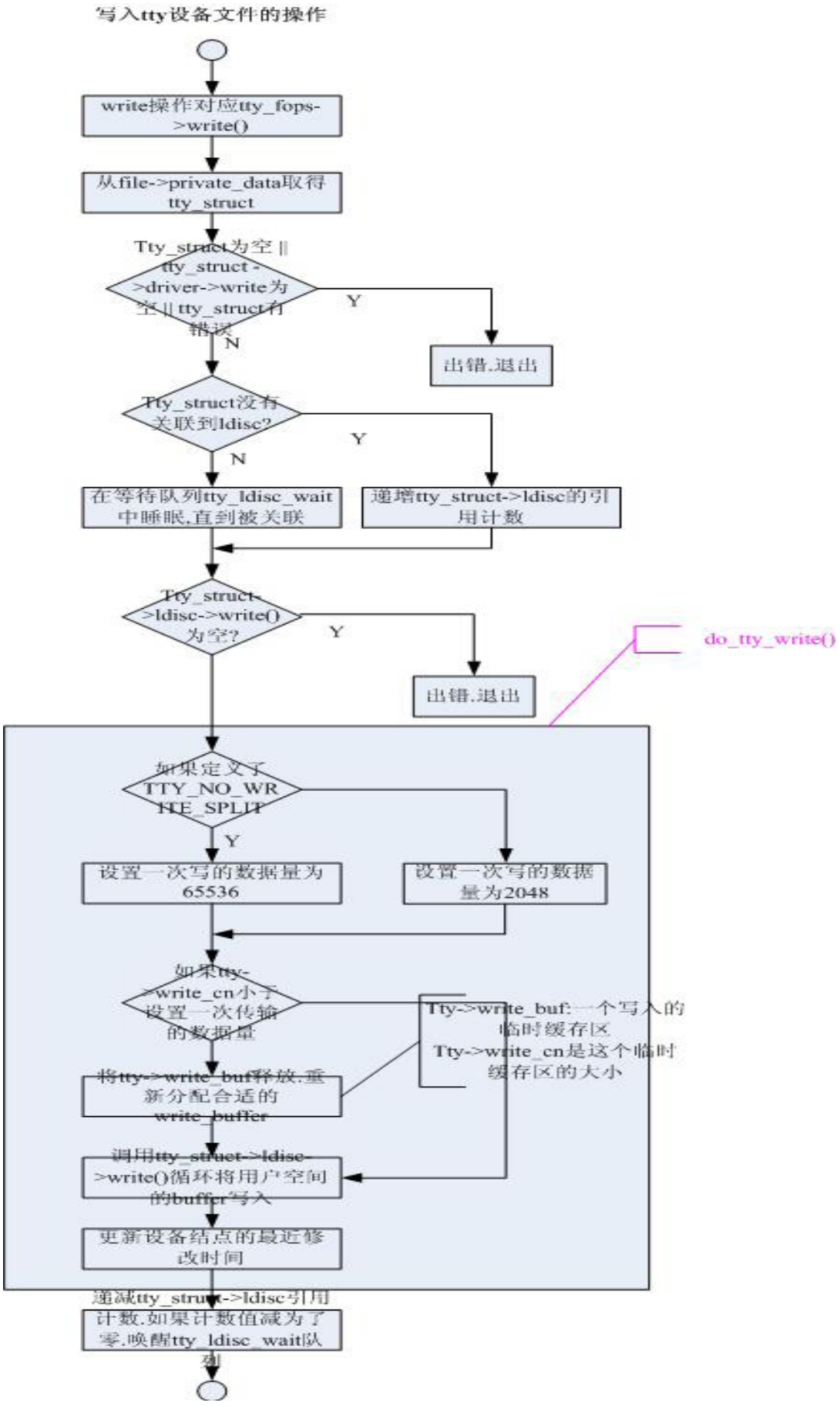
```
         * But if TTY_NO_WRITE_SPLIT is set, we should use a
         * big chunk-size..
         *
         * The default chunk-size is 2kB, because the NTTY
         * layer has problems with bigger chunks. It will
         * claim to be able to handle more characters than
         * it actually does.
         */
        chunk = 2048;
        if (test_bit(TTY_NO_WRITE_SPLIT, &tty->flags))
                chunk = 65536;
        if (count < chunk)
                chunk = count;

        /* write_buf/write_cnt is protected by the atomic_write semaphore */
        if (tty->write_cnt < chunk) {
                unsigned char *buf;

                if (chunk < 1024)
                        chunk = 1024;

                buf = kmalloc(chunk, GFP_KERNEL);
                if (!buf) {
                        up(&tty->atomic_write);
                        return -ENOMEM;
                }
                kfree(tty->write_buf);
                tty->write_cnt = chunk;
                tty->write_buf = buf;
        }
/*默认一次写数据的大小为2K.如果设置了TTY_NO_WRITE_SPLIT.则将一次写的数据量扩大为65536.
*Tty->write_buf是写操作的临时缓存区。即将用户空的数据暂时存放到这里
*Tty->write_cnt是临时缓存区的大小。
*在这里，必须要根据一次写的数据量对这个临时缓存区做调整
*/
        /* Do the write .. */
        for (;;) {
                size_t size = count;
                if (size > chunk)
                        size = chunk;
                ret = -EFAULT;
                if (copy_from_user(tty->write_buf, buf, size))
                        break;
```

```
            lock_kernel();
            ret = write(tty, file, tty->write_buf, size);
            unlock_kernel();
            if (ret <= 0)
                break;
            written += ret;
            buf += ret;
            count -= ret;
            if (!count)
                break;
            ret = -ERESTARTSYS;
            if (signal_pending(current))
                break;
            cond_resched();
        }
        if (written) {
            struct inode *inode = file->f_dentry->d_inode;
            inode->i_mtime = current_fs_time(inode->i_sb);
            ret = written;
        }
        up(&tty->atomic_write);
        return ret;
}
```

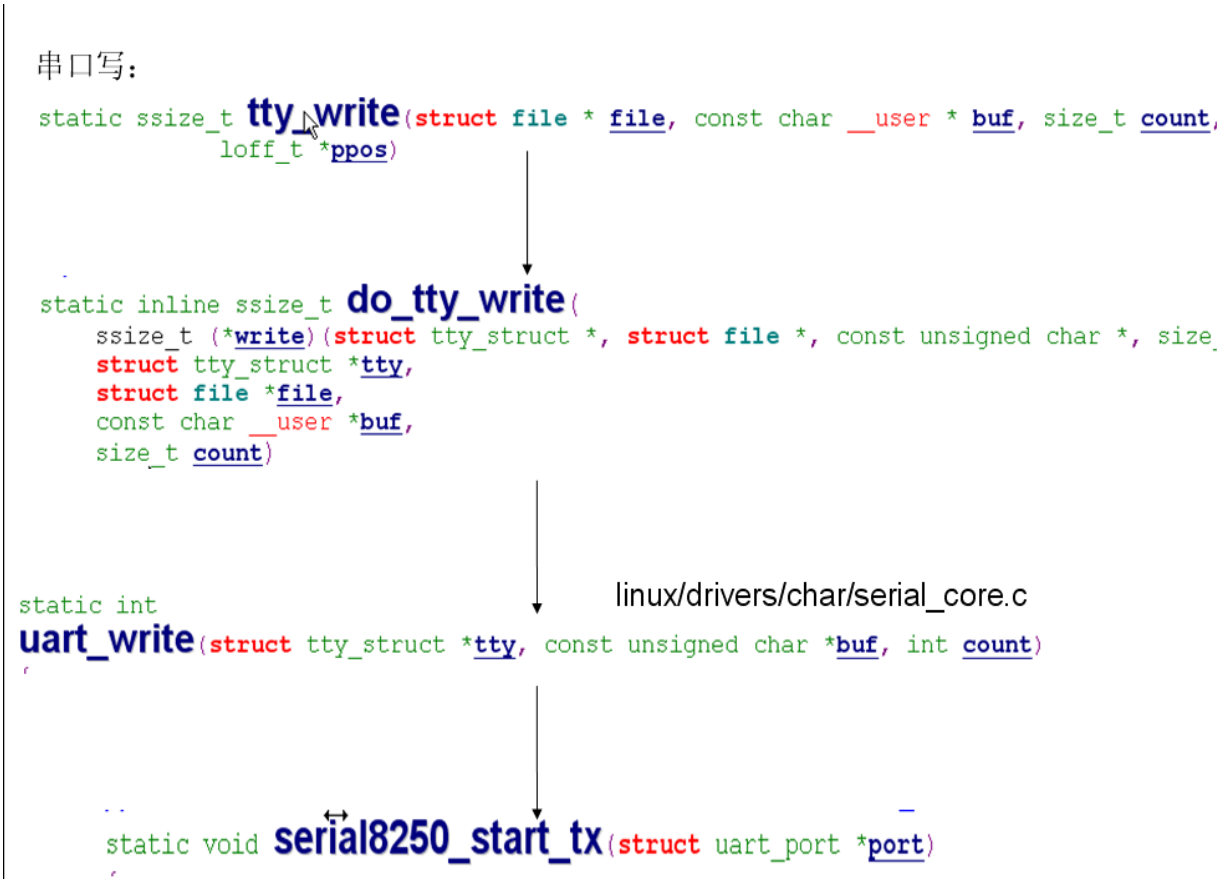
后面的操作就比较简单了。先将用户空间的数据copy到临时缓存区，然后再调用ldisc->write()完成这次写操作.最后再更新设备结点的时间.

Write操作的流程图如下示：

在这里，我们只看到将数据写放到了ldisc->write().没有看到与tty_driver相关的部份。实际上在ldisc中对写入的数据做预处理过后,还是会调用tty_driver->write()将其写入硬件.


写入tty设备文件的操作
write操作对应tty_fops->write()
从file->private_data取得tty_struct
Tty_struct为空 || tty_struct->driver->write为空 || tty_struct有错误
Y
N
出错.退出
Tty_struct没有关联到ldisc?
Y
N
在等待队列tty_ldisc_wait中睡眠,直到被关联
递增tty_struct->ldisc的引用计数
Tty_struct->ldisc->write()为空?
Y
出错.退出
do_tty_write()
如果定义了TTY_NO_WRITE_SPLIT
Y
设置一次写的数据量为65536
设置一次写的数据量为2048
如果tty->write_cn小于设置一次传输的数据量
Tty->write_buf:一个写入的临时缓存区
Tty->write_cn是这个临时缓存区的大小
将tty->write_buf释放,重新分配合适的write_buffer
调用tty_struct->ldisc->write()循环将用户空间的buffer写入
更新设备结点的最近修改时间
递减tty_struct->ldisc引用计数,如果计数值减为了零,唤醒tty_ldisc_wait队列

串口写的函数调用关系如下：




## 2. 设备文件的read操作

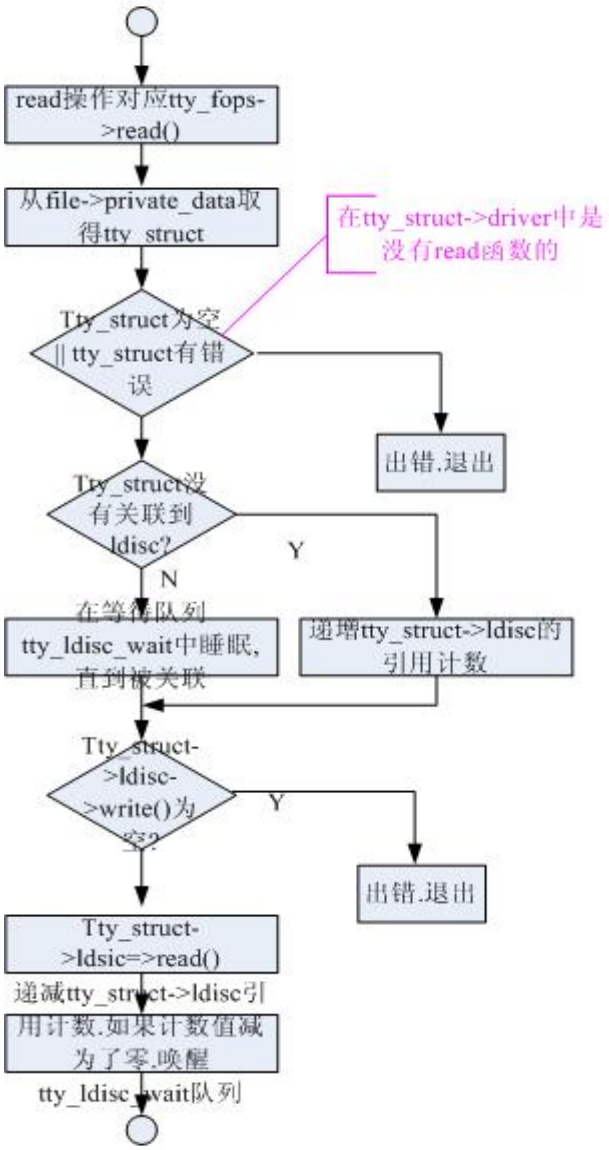
```
static ssize_t tty_read(struct file * file, char __user * buf, size_t count,
            loff_t *ppos)
{
    int i;
    struct tty_struct * tty;
    struct inode *inode;
    struct tty_ldisc *ld;

    tty = (struct tty_struct *)file->private_data;
    inode = file->f_dentry->d_inode;
    if (tty_paranoia_check(tty, inode, "tty_read"))
        return -EIO;
    if (!tty || (test_bit(TTY_IO_ERROR, &tty->flags)))
        return -EIO;

    /* We want to wait for the line discipline to sort out in this
```

```
        situation */
    ld = tty_ldisc_ref_wait(tty);
    lock_kernel();
    if (ld->read)
        i = (ld->read)(tty,file,buf,count);
    else
        i = -EIO;
    tty_ldisc_deref(ld);
    unlock_kernel();
    if (i > 0)
        inode->i_atime = current_fs_time(inode->i_sb);
    return i;
}
```

这个read操作就更简单。直接调用ldsic->read()完成工作

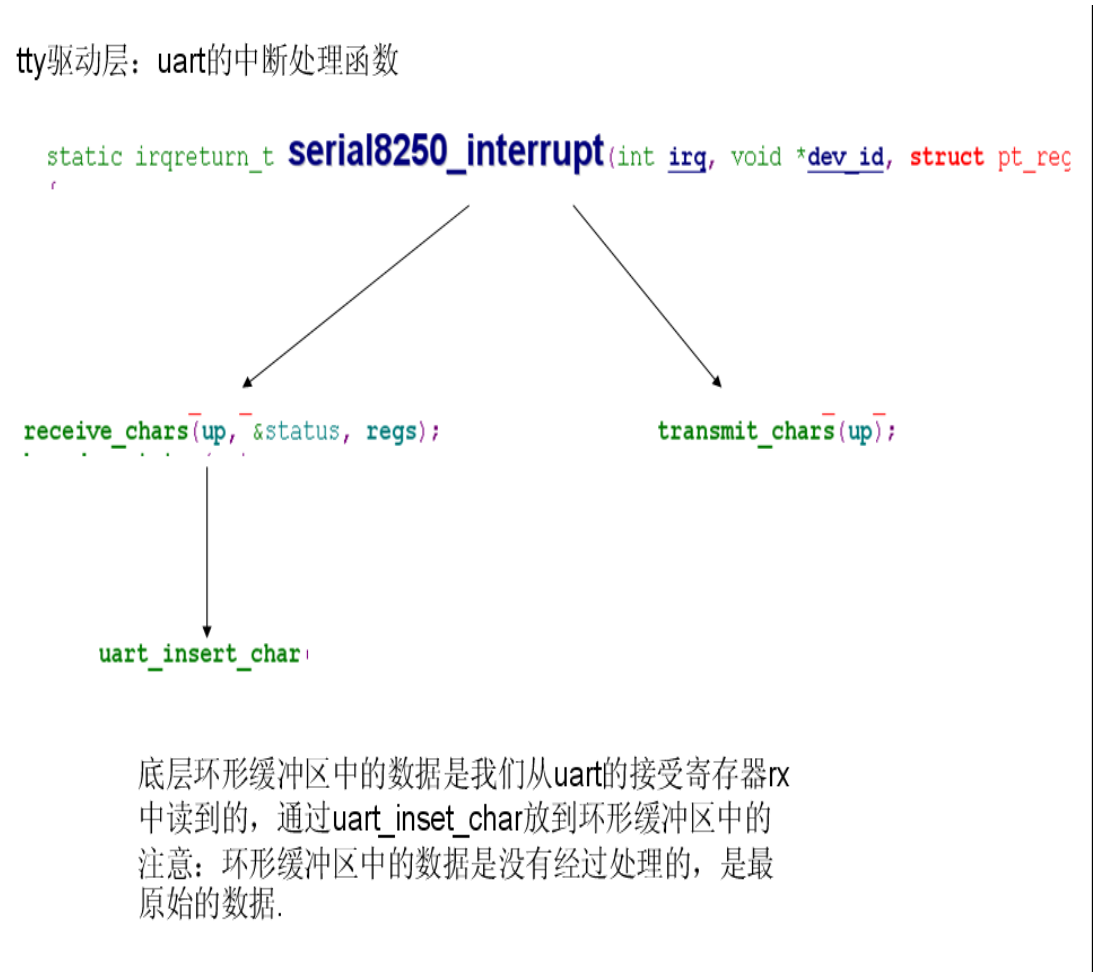
流程图如下：

读取tty设备文件的操作



**串口读的函数调用关系如下：**


这个read_chan读到是我们通过flush_to_ldisc从底层环形缓冲区刷到read_buf中的数据，但是底层环形缓冲区中的数据从哪儿来的呢？？



总结：在 tty 设备文件的操作中。Open 操作会进行一系统初始化。然后调用 ldsic->open tty_driver->open。在 write 和 read 调用中只 tty_core 只会用到 ldisc->wirte/ldisc->read.除了上面分析的几个操作之外，还有一个 ioctl 操作，以及它封装的几个 termios。这些 ioctl 类的操作会直接和 tty_driver 相关联.