# 综论

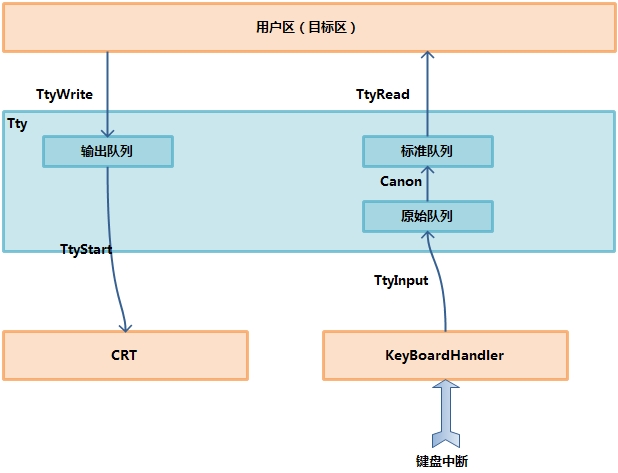
字符设备，是处理字符 输入/输出的设备。在UnixV6++中主要涉及的字符设备如下：

1. 键盘，输入字符
2. CRT，输出字符
3. Tty终端，直观上是一个键盘和一个CRT的输入/输出组合，用于管理单一用户的输入/输出

由于设备速度与CPU处理速度的差异，我们需要使用缓存技术提高字符的处理效率。主要用到的缓存结构是缓存队列，由Tty提供了三个缓存队列：

1. 原始队列，存放从Keyboard获得的原始输入字符
2. 标准队列，对原始队列中的元素进行处理（如遇到删除字符进行删除操作，遇到删行进行删行操作）后的队列
3. 输出队列，向CRT类输出的字符队列

下图展示了 Tty与CRT及键盘的字符流向，及各缓存队列间的关系和主要函数名，在这里只需有个直观的印象，具体内容在相应章节中会介绍



# CRT类

CRT类负责向屏幕的字符输出，提供基本的输出接口，将输出队列的内容输出到屏幕上

## 核心变量

包括：

1. m\_VideoMemory 显存地址，前面知识我们已知这个常量的来历
2. m\_CursorX，m\_CursorY 记录光标位置，即下一个输出字符的位置
3. m\_Position 在输出队列中，标记下一个输出字符
4. m\_BeginChar 在输出队列中，标记已确认的字符的下一字符。回车操作，被认为是确认字符的操作，回车符前的所有字符将被确认，不能再被”退格键”删除，而未确认字符可以被”退格键”删除。直观上说，m\_BeginChar指向当前行首（由于回车键的换行），规定了”退格键”的作用范围

## 核心函数

核心函数的介绍将按照调用关系从底层向高层的顺序：

### MoveCursor

负责移动光标到指定位置

光标的移动主要是对寄存器的读写操作，先介绍所用到的寄存器：

VGA系统中，CRT Controller Registers寄存器组，共25个寄存器，我们用到的是

1. Cursor Location High Register，索引0Eh
2. Cursor Location Low Register，索引0Fh

这个寄存器组包括两个端口

1. 地址端口0x3D4
2. 数据端口0x3D5

通过端口读写Cursor Location的方法代码如下：

**IOPort::OutByte(0x3D4, 0xE);**

**IOPort::OutByte(0x3D5, cursorPosition >> 8);**

**IOPort::OutByte(0x3D4,0xF);**

**IOPort::OutByte(0x3D5, cursorPosition & 0xFF);**

如代码，先向地址端口输出寄存器索引，再向数据端口输出寄存器数据来写指定寄存器。这里将光标位置低8位和高8位分别输出到对应寄存器，完成光标位置的设定

MoveCursor的代码：

void CRT::MoveCursor(unsigned int col, unsigned int row)

{

if ( (col < 0 || col >= CRT::COLUMNS) || (row < 0 || row >= CRT::ROWS) )

{

return;

}

/\* 计算光标偏移量 \*/

unsigned short cursorPosition = row \* CRT::COLUMNS + col;

/\* 选择 r14和r15寄存器，分别为光标位置的高8位和低8位 \*/

IOPort::OutByte(CRT::VIDEO\_ADDR\_PORT, 14);

IOPort::OutByte(CRT::VIDEO\_DATA\_PORT, cursorPosition >> 8);

IOPort::OutByte(CRT::VIDEO\_ADDR\_PORT, 15);

IOPort::OutByte(CRT::VIDEO\_DATA\_PORT, cursorPosition & 0xFF);

}

### ClearScreen

清屏，代码中0x0A00的作用已在前面章节说明过

void CRT::ClearScreen()

{

unsigned int i;

for ( i = 0; i < COLUMNS \* ROWS; i++ )

{

m\_VideoMemory[i] = (unsigned short)' ' | 0x0A00;

}

}

### NextLine

换行时，进行参数调整

void CRT::NextLine()

{

m\_CursorX = 0;

m\_CursorY += 1;

/\* 超出最大行数 \*/

if ( m\_CursorY >= CRT::ROWS )

{

m\_CursorY = 0;

ClearScreen();

}

MoveCursor(m\_CursorX, m\_CursorY);

}

### WriteChar

向屏幕光标当前位置输出字符，超出列限时换行

void CRT::WriteChar(char ch)

{

m\_VideoMemory[m\_CursorY \* CRT::COLUMNS + m\_CursorX] = (unsigned char) ch | 0x0A00;

m\_CursorX++;

if ( m\_CursorX >= CRT::COLUMNS )

{

NextLine();

}

MoveCursor(m\_CursorX, m\_CursorY);

}

### BackSpace

负责进行退格的参数修正，考虑一下的情况

1. 光标不在行首，退格将光标后退一个位置
2. 光标在行首时，退格将光标退回上一行末尾
3. 退行时，已经到顶行，重置参数

void CRT::BackSpace()

{

m\_CursorX--;

/\* 移动光标，如果要回到上一行的话 \*/

if ( m\_CursorX < 0 )

{

m\_CursorX = CRT::COLUMNS - 1;

m\_CursorY--;

if ( m\_CursorY < 0 )

{

m\_CursorY = 0;

}

}

MoveCursor(m\_CursorX, m\_CursorY);

/\* 在光标所在位置填上空格 \*/

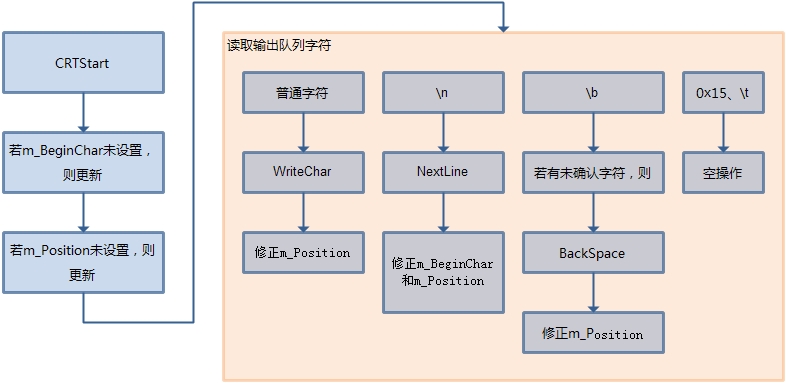
m\_VideoMemory[m\_CursorY \* COLUMNS + m\_CursorX] = ' ' | 0x0A00;

}

### CRTStart

这是CRT类最核心的函数，负责输出输出队列中的内容，直到队列空

流程图如下:



代码如下:

void CRT::CRTStart(TTy\* pTTy)

{

char ch;

if ( 0 == CRT::m\_BeginChar)

{

m\_BeginChar = pTTy->t\_outq.CurrentChar();

}

if ( 0 == m\_Position )

{

m\_Position = m\_BeginChar;

}

while ( (ch = pTTy->t\_outq.GetChar()) != TTy::GET\_ERROR )

{

switch (ch)

{

case '\n':

NextLine();

CRT::m\_BeginChar = pTTy->t\_outq.CurrentChar();

m\_Position = CRT::m\_BeginChar;

break;

case 0x15:

//del\_line();

break;

case '\b':

if ( m\_Position != CRT::m\_BeginChar )

{

BackSpace();

m\_Position--;

}

break;

case '\t':

//m\_Position++;

break;

default: /\* 在屏幕上回显普通字符 \*/

WriteChar(ch);

m\_Position++;

break;

}

}

}

这里需要解释的是参数m\_BeginChar和 m\_Position 的修正，内容如下：

1. 若m\_BeginChar无值，则置为输出队列的CurrentChar（实际为输出队列下一个输出字符），意义是之前所有的字符都被”确认”了
2. 若m\_Position无值，置初值为m\_BeginChar。只在初始时有效，与上句实际意义就是m\_Position = m\_BeginChar = 输出队列的CurrentChar。
3. 处理”\n”时，m\_Position = m\_BeginChar = 输出队列的CurrentChar，意义是确认之前的所有字符
4. 处理”\b”时，m\_Position != CRT::m\_BeginChar的意义就是存在未确认字符

# KeyBoard类

## 键盘基础知识

我们在编写键盘驱动程序时，需要了解以下两个问题：

1. 按键过程中，键盘的硬件接口提供给我们的是怎样的数据
2. 这些数据跟按键所代表的意义（字母等）有什么对应关系

首先，键盘的硬件接口提供给我们的数据是扫描码（ScanCode），分为MakeCode和BreakCode，对应于一个键的按下和释放。举个例子，当我们按下A键时，我们将收到0x1E（MakeCode），当我们释放A键时，我们将收到0x9E（BreakCode）。这里的规律是BreakCode=MakeCode | 0x80

当然，并不是所有键都是类似于A键的情况，有以下两种特殊的键由于特殊的原因具有一个特殊的扫描码序（而非单一的码）：

1. 诸如Right Shift、Insert这些功能键，具有 0xE0 + Code 的扫描码序列
2. Pause键，扫描码序为 0xE1 0x1D 0x45 和 0xE1 0x9D 0xC5

在处理键盘扫描码的过程中我们将其分为 单码、双码、三码 三种情况讨论。

每个键按下/释放时，会产生键盘中断，此时我们可以通过以下两个寄存器读取键盘的状态和数据：

1. 0x60 数据寄存器，可以从这个寄存器读取键盘键的扫描码
2. 0x64 状态寄存器，可以读取键盘的状态，具体用到时我们在代码中解释相应位的含义

## 核心变量

### Keymap 和 Shift\_Keymap

分别是 无shift和shift 两种状态下，扫描码到字母的映射表，具体值如下：

char Keyboard::Keymap[] = {

0,0x1b,'1','2','3','4','5','6', /\* 0x00-0x07 0, <esc>,1,2,3,4,5,6, \*/

'7','8','9','0','-','=',0x8,0x9, /\* 0x08-0x0f 7,8,9,0,-,=,<backspace><tab>\*/

'q','w','e','r','t','y','u','i', /\* 0x10-0x17 qwertyui\*/

'o','p','[',']','\n',0,'a','s', /\* 0x18-0x1f op[] <enter><ctrl>as \*/

'd','f','g','h','j','k','l',';', /\* 0x20-0x27 dfghjkl; \*/

'\'','`',0,'\\','z','x','c','v', /\* 0x28-0x2f '`<lshift>\zdcv \*/

'b','n','m',',','.','/',0,'\*', /\* 0x30-0x37 bnm,./<rshitf><printscr> \*/

0,' ',0,0,0,0,0,0, /\*0x38-0x3f <alt><space><caps><f1><f2><f3><f4><f5> \*/

0,0,0,0,0,0,0,'7', /\* 0x40-0x47 <f0><><><><><numlock><scrlock>7\*/

'8','9','-','4','5','6','+','1', /\* 0x48-x04f 89-456+1 \*/

'2','3','0','.',0,0,0,0 /\* 0x50-0x57 230.<><><><> \*/

};

char Keyboard::Shift\_Keymap[] = {

0,0x1b,'!','@','#','$','%','^', /\* 0x00-0x07 0, <esc>,1,2,3,4,5,6, \*/

'&','\*','(',')','\_','+',0x8,0x9, /\* 0x08-0x0f 7,8,9,0,-,=,<backspace><tab>\*/

'q','w','e','r','t','y','u','i', /\* 0x10-0x17 qwertyui\*/

'o','p','{','}','\n',0,'a','s', /\* 0x18-0x1f op[] <enter><ctrl>as \*/

'd','f','g','h','j','k','l',':', /\* 0x20-0x27 dfghjkl; \*/

'\"','~',0,'|','z','x','c','v', /\* 0x28-0x2f '`<lshift>\zdcv \*/

'b','n','m','<','>','?',0,'\*', /\* 0x30-0x37 bnm,./<rshitf><printscr> \*/

0,' ',0,0,0,0,0,0, /\*0x38-0x3f <alt><space><caps><f1><f2><f3><f4><f5> \*/

0,0,0,0,0,0,0,0, /\* 0x40-0x47 <f0><><><><><numlock><scrlock>7\*/

0,0,'-',0,0,0,'+',0, /\* 0x48-x04f 89-456+1 \*/

0,0,0,0x7f/\*del\*/,0,0,0,0 /\* 0x50-0x57 230.<><><><> \*/

};

### Mode

模式变量，表示 Caps Lock、Num Lock 等状态，以及 Shift、Ctrl等是否按下，状态常量如下：

/\* 以下为控制键按下的状态，mode中各比特位的定义 \*/

static const int M\_LCTRL = 0x01;

static const int M\_RCTRL = 0x02;

static const int M\_LALT = 0x04;

static const int M\_RALT = 0x08;

static const int M\_LSHIFT = 0x10;

static const int M\_RSHIFT = 0x20;

static const int M\_NUMLOCK = 0x40;

static const int M\_CAPSLOCK = 0x80;

static const int M\_SCRLOCK = 0x100;

static const int M\_DOWN\_NUMLOCK = 0x200;

static const int M\_DOWN\_CAPSLOCK = 0x400;

static const int M\_DOWN\_SCRLOCK = 0x800;

## 核心函数

核心函数的介绍顺序将按照调用高层到调用底层的顺序，从键盘中断函数开始

### KeyboardHandler

键盘上每个键的按下/释放，都会引发键盘中断，我们从键盘中断处理函数开始介绍

void Keyboard::KeyboardHandler( struct pt\_regs\* reg, struct pt\_context\* context )

{

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

//for test

//Diagnose::Write("key pressed ! ");

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

unsigned char status = IOPort::InByte(Keyboard::STATUS\_PORT);

int limit = 10;

static int pre\_state = 0;

while ( (status & Keyboard::DATA\_BUFFER\_BUSY) && limit-- )

{

/\* 如果键盘缓存满就要读入剩下的扫描码 \*/

unsigned char scancode = IOPort::InByte(Keyboard::DATA\_PORT);

/\* 以下的判断过程有点像有限状态机，不过貌似有的地方没有化简 \*/

if ( 0 == pre\_state )

{

if ( 0xE0 == scancode || 0xE1 == scancode )

{

/\* 将状态改为0xe0表示还有字符没有读入 \*/

/\* 产生0xe1只有一种状态，那就是pause键被按下，按键按下序列为0xe1,0x1d,0x45，按键断开序列为0xe1,0x9d,0xc5 \*/

pre\_state = scancode;

}

else /\* 非扩展键 \*/

{

pre\_state = 0;

Keyboard::HandleScanCode(scancode, 0);

}

}

else if ( 0xE0 == pre\_state )

{

/\* 扩展键的第二个扫描码 \*/

pre\_state = 0;

Keyboard::HandleScanCode(scancode, 0xe0);

}

else if ( 0xE1 == pre\_state && ( 0x1d == scancode || 0x9d == scancode ) )

{

pre\_state = 0x100; /\* 中间状态，表示pause已经有两个键吻合 \*/

}

else if ( pre\_state == 0x100 && 0x45 == scancode ) /\* 只需要知道pause键什么时候按下 \*/

{

pre\_state = 0;

Keyboard::HandleScanCode(scancode, 0xe1);

}

else

{

pre\_state = 0;

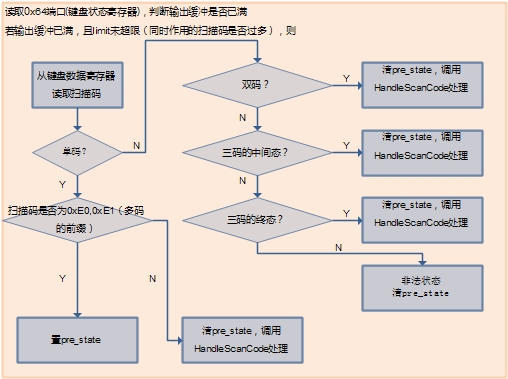
}

status = IOPort::InByte(Keyboard::STATUS\_PORT);

}

}

程序流程图如下：



### HandleScanCode

负责处理提取出的扫描码，参数的形式为 expand + scanCode 的组合，expand为双码和三码的前缀状态。代码如下：

void Keyboard::HandleScanCode(unsigned char scanCode, int expand)

{

int isOK = 0;

char ch = 0;

if ( 0xE1 == expand )

{

ch = Keyboard::ScanCodeTranslate(scanCode, expand);

}

switch ( scanCode )

{

case SCAN\_ALT:

if ( 0xE0 == expand ) /\* 使用扩展键，表示是右边的 alt按下 \*/

Mode |= M\_RALT;

else

Mode |= M\_LALT;

break;

case SCAN\_CTRL:

if ( 0xE0 == expand ) /\* 使用扩展键，表示是右边的 ctrl按下 \*/

Mode |= M\_RCTRL;

else

Mode |= M\_LCTRL;

break;

case SCAN\_LSHIFT:

Mode |= M\_LSHIFT;

break;

case SCAN\_RSHIFT:

Mode |= M\_RSHIFT;

break;

/\* 处理按键被松开的情况，清空Mode中按键对应的标志位 \*/

case SCAN\_ALT + 0x80:

if ( 0xE0 == expand )

Mode &= ~M\_RALT;

else

Mode &= ~M\_LALT;

break;

case SCAN\_CTRL + 0x80:

if ( 0xE0 == expand )

Mode &= ~M\_RCTRL;

else

Mode &= ~M\_LCTRL;

break;

case SCAN\_LSHIFT + 0x80:

Mode &= ~M\_LSHIFT;

break;

case SCAN\_RSHIFT + 0x80:

Mode &= ~M\_RSHIFT;

break;

/\* 每次按下就反转改变状态位 \*/

/\*

\* M\_DOWN\_NUMLOCK宏表示NUMLOCK被按下，在没有按下的

\* 状态下按下NumLock键需要反转状态，并置numlock键没有松开

\*/

case SCAN\_NUMLOCK:

isOK = Mode & M\_DOWN\_NUMLOCK;

if ( !isOK )

{

Mode ^= M\_NUMLOCK;

Mode |= M\_DOWN\_NUMLOCK;

}

break;

case SCAN\_CAPSLOCK:

isOK = Mode & M\_DOWN\_CAPSLOCK;

if ( !isOK )

{

Mode ^= M\_CAPSLOCK;

Mode |= M\_DOWN\_CAPSLOCK;

}

break;

case SCAN\_SCRLOCK:

isOK = Mode & M\_DOWN\_SCRLOCK;

if ( !isOK )

{

Mode ^= M\_SCRLOCK;

Mode |= M\_DOWN\_SCRLOCK;

}

break;

/\* 释放NumLock，CapsLock，ScrollLock键 \*/

case SCAN\_NUMLOCK + 0x80:

Mode &= ~M\_DOWN\_NUMLOCK;

break;

case SCAN\_CAPSLOCK + 0x80:

Mode &= ~M\_DOWN\_CAPSLOCK;

break;

case SCAN\_SCRLOCK + 0x80:

Mode &= ~M\_DOWN\_SCRLOCK;

break;

default:

ch = Keyboard::ScanCodeTranslate(scanCode, expand);

break;

}

if ( 0 != ch )

{

TTy\* pTTy = Kernel::Instance().GetDeviceManager().GetCharDevice(DeviceManager::TTYDEV).m\_TTy;

if ( NULL != pTTy )

{

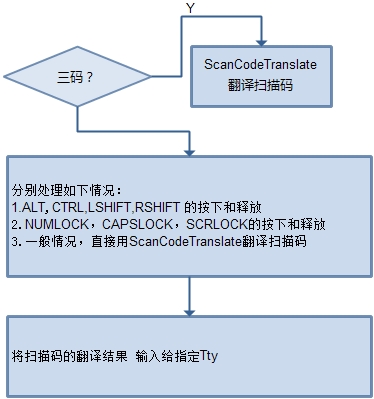
pTTy->TTyInput(ch);

}

}

}

流程图如下：



显然，对于功能键的处理，主要是修改Mode。对于一般键的翻译，交给了ScanCodeTranslate函数

### ScanCodeTranslate

负责对一般按键、Pause键、小键盘键 的扫描码进行翻译。

由以上我们知道，其他功能键已由HandleScanCode处理。

char Keyboard::ScanCodeTranslate(unsigned char scanCode, int expand)

{

char ch = 0;

bool bReverse = false;

if ( 0xE1 == expand ) /\* Pause Key \*/

{

ch = 0x05; /\* Pause ASCII \*/

}

else if ( scanCode < 0x45 ) /\* 非小键盘和控制键 \*/

{

/\* 根据扫描码映射表找到对应按键的ASCII码 \*/

if ( (Mode & M\_LSHIFT) || (Mode & M\_RSHIFT) )

{

ch = Shift\_Keymap[scanCode];

}

else

{

ch = Keymap[scanCode];

}

if ( ch >= 'a' && ch <= 'z' )

{

/\* 是小写字符而且已经Capslock了，那么转换成大写字符 \*/

bReverse = ( (Mode & M\_CAPSLOCK) ? 1 : 0 ) ^ ( (Mode & M\_LSHIFT) || (Mode & M\_RSHIFT) );

if ( (Mode & M\_LCTRL) || (Mode & M\_RCTRL) ) /\* 按下ctrl进行转意 \*/

{

ch -= 'a';

ch++; /\* 转义从0x1 开始\*/

}

else if ( bReverse )

{

ch += 'A' - 'a';

}

}

}

else if ( scanCode < 0x58 )

{

bReverse = ( (Mode & M\_NUMLOCK) ? 1 : 0 ) ^ ( (Mode & M\_LSHIFT) || (Mode & M\_RSHIFT) );

if ( 0xE0 == expand )

{

ch = Shift\_Keymap[scanCode];

}

else if ( bReverse )

{

ch = Keymap[scanCode];

}

else

{

ch = Shift\_Keymap[scanCode];

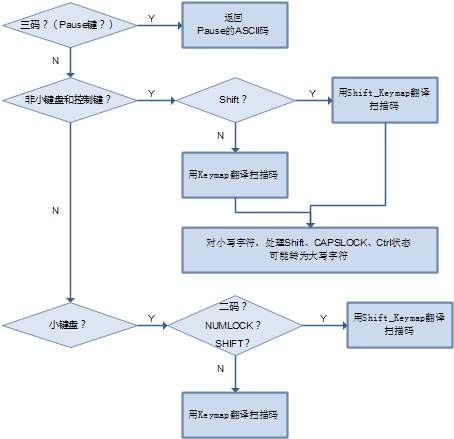
}

}

return ch;

}

流程图如下：



# Tty类

## 综述

Tty类主要负责提供终端的基本功能，包括输入功能和输出功能。

其中最主要的结构是 Tty类的三个队列：

1. 原始队列，存放从Keyboard获得的原始输入字符
2. 标准队列，对原始队列中的元素进行处理（如遇到删除字符进行删除操作，遇到删行进行删行操作）后的队列
3. 输出队列，向CRT类输出的字符队列

由函数控制的元素在各个队列中的转移过程如下：

1. 键盘按键引发 键盘中断，由Keyboard类的KeyboardHandler处理，其中调用HandleScanCode解析扫描码，并将解析的结果字符提交给Tty的TtyInput函数处理。
2. TtyInput负责将字符送入原始队列。回显时，TtyInput调用TtyOutput回显字符
3. TtyOutput负责将字符送入输出队列，在TtyStart被调用时（Tty设备开始工作时），输出队列的内容交由Crt类输出在屏幕上
4. TTRead函数的功能是 输入到Tty标准队列的字符输出到目标区（由用户指定）。其中调用到Canon函数将原始队列的字符进行规范化，并交给标准队列
5. TTWrite函数的功能是 将目标区的字符 输出到 Tty的输出队列

由以上，可以完成基本的终端I/O功能

## 核心变量

### 三个队列

TTy\_Queue t\_rawq; /\* 原始输入字符缓存队列 \*/

TTy\_Queue t\_canq; /\* 标准输入字符缓存队列 \*/

TTy\_Queue t\_outq; /\* 输出字符缓存队列 \*/

### 状态字t\_flags

状态常量包括：

/\* modes (t\_flags设置) \*/

static const int HUPCL = 0x1;

static const int XTABS = 0x2;

static const int LCASE = 0x4;

static const int ECHO = 0x8;

static const int CRMOD = 0x10;

static const int RAW = 0x20;

其中有一些时UnixV6遗留下来的，已经用不到了，对于用到的状态常量进行解释：

1. LCASE 小写终端
2. ECHO 回显终端，对输入的字符回显
3. CRMOD 使用CR风格的换行 “\r\n”
4. RAW 原始终端，对所有的输入保持原样，不操作

## 核心函数

### PassC和CPass

PassC的功能 是将字符送入目标区

CPass的功能 是从目标区读入字符

目标区属性 由 User.u\_IOParam确定

int TTy::PassC(char ch)

{

User& u = Kernel::Instance().GetUser();

/\* 将字符送入用户目标区 \*/

if ( u.u\_IOParam.m\_Count > 0 )

{

\*(u.u\_IOParam.m\_Base++) = ch;

//u.u\_IOParam.m\_Offset++;

u.u\_IOParam.m\_Count--;

return 0;

}

return -1;

}

char TTy::CPass()

{

char ch;

User& u = Kernel::Instance().GetUser();

ch = \*(u.u\_IOParam.m\_Base++);

if ( u.u\_IOParam.m\_Count > 0 )

{

u.u\_IOParam.m\_Count--;

//u.u\_IOParam.m\_Offset++;

return ch;

}

else

{

return -1;

}

}

### TtyInput

向终端输入一个字符。会将字符送入原始队列中。代码如下：

void TTy::TTyInput(char ch)

{

if ( (ch &= 0xFF) == '\r' && (this->t\_flags & TTy::CRMOD) )

{

ch = '\n';

}

/\* 如果是小写终端 \*/

if ( (this->t\_flags & TTy::LCASE) && ch >= 'A' && ch <= 'Z' )

{

ch += 'a' - 'A';

}

/\* 将输入字符放入原始字符缓存队列 \*/

this->t\_rawq.PutChar(ch);

if ( this->t\_flags & TTy::RAW || ch == '\n' || ch == TTy::CEOT )

{

Kernel::Instance().GetProcessManager().WakeUpAll((unsigned long)&this->t\_rawq);

this->t\_rawq.PutChar(0x7);

this->t\_delct++;

}

if ( this->t\_flags & TTy::ECHO )

{

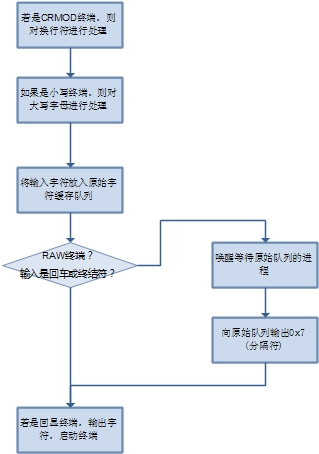
this->TTyOutput(ch);

this->TTStart();

}

}

流程图如下：



### TtyOutput

向终端输出一个字符。字符将送入输出队列。代码如下：

void TTy::TTyOutput(char ch)

{

/\* 如果输入字符为文件结束符，并且终端工作在原始方式下，则返回 \*/

if ( (ch &= 0xFF) == TTy::CEOT && (this->t\_flags & TTy::RAW) == 0 )

{

return;

}

if ( '\n' == ch && (this->t\_flags & TTy::CRMOD) )

{

this->TTyOutput('\r');

}

/\* 将字符放入输出字符缓存队列中 \*/

if (ch)

{

this->t\_outq.PutChar(ch);

}

}

流程图如下：



### TTStart

启动终端。在这里就是启动CRT进行输出操作。

void TTy::TTStart()

{

CRT::CRTStart(this);

}

### Canon

对原始队列中的字符进行处理，处理BackSpace和 删行，处理到下一个分隔符，并将结果送入标准队列。

int TTy::Canon()

{

char\* pChar;

char ch;

User& u = Kernel::Instance().GetUser();

X86Assembly::CLI();

while ( this->t\_delct == 0 )

{

if ( (this->t\_state & TTy::CARR\_ON) == 0 )

{

/\* 设备没有打开，返回 \*/

return 0;

}

u.u\_procp->Sleep((unsigned long)&this->t\_rawq, ProcessManager::TTIPRI);

}

X86Assembly::STI();

loop:

pChar = &Canonb[2];

while ( (ch = this->t\_rawq.GetChar()) >= 0 )

{

if ( 0x7 == ch ) /\* 0x07是每个输入串的分隔符 \*/

{

this->t\_delct--;

break;

}

if ( (this->t\_flags & TTy::RAW) == 0 ) /\* 如果不是raw模式，需要在遇到转义字符时作相应的变换 \*/

{

if ( pChar[-1] != '\\' )

{

if ( ch == this->t\_erase ) /\* 如果是backspace键\*/

{

if ( pChar > &Canonb[2] ) /\* 直接删除前面的字符 \*/

{

pChar--;

}

continue;

}

if ( ch == this->t\_kill ) /\* 是删除一行 \*/

{

/\* 从行首重新开始，这里使用loop虽然破坏了程序结构，但的确是简化了许多操作，v6原来就是这样写的 \*/

goto loop;

}

if ( ch == TTy::CEOT ) /\* CEOT == 0x4 \*/

{

/\* 遇到文件结束符并不作任何操作，这个是在将输入输出转向时将文件当作终端读写时为了保持一致 \*/

continue;

}

}

}

else

{

/\* 暂时不处理raw模式下的情况 \*/

}

\*pChar++ = ch; /\* 普通字符直接放入工作缓存 \*/

if ( pChar >= Canonb + TTy::CANBSIZ )

{

break;

}

}

char\* pEnd = pChar;

pChar = &Canonb[2];

/\* 将cook过的字符送到行规则缓存，供ttread读取 \*/

while ( pChar < pEnd )

{

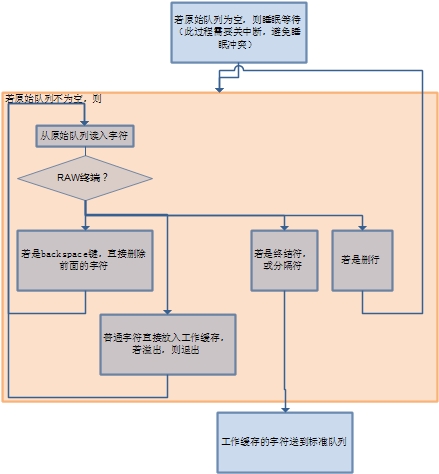
this->t\_canq.PutChar(\*pChar++);

}

return 1;

}

流程图如下：



### TTRead

TTRead很简单，就是将标准队列中的元素送到指定目标区

void TTy::TTRead()

{

/\* 设备没有开始工作，返回 \*/

if ( (this->t\_state & TTy::CARR\_ON) == 0 )

{

return;

}

if ( this->t\_canq.CharNum() || this->Canon() )

{

while ( this->t\_canq.CharNum() && (this->PassC(this->t\_canq.GetChar()) >= 0) );

}

}

### TTWrite

TTWrite也很简单，将目标区的字符送入输出队列，在输出队列满时，启动TTy进行屏幕输出

void TTy::TTWrite()

{

/\*

\* 因为现在的输出设备是内存，其相应速度相当快，所以

\* 不需要在结束回显后发送中断，这样的代价非常大。这和

\* 原来unix v6的有些不同。本来可以在输出字符的过程中关

\* 中断来防止用户输入但不能被正常删除的bug,但因为这样

\* 可能会导致时钟中断相应的延迟，这个问题更加严重，因

\* 次没有这样做。

\*/

char ch;

/\* 设备没有开始工作，返回 \*/

if ( (this->t\_state & TTy::CARR\_ON) == 0 )

{

return;

}

while ( (ch = CPass()) > 0 )

{

/\*输出队列中超过规定字符数，需要赶快显示 \*/

if ( this->t\_outq.CharNum() > TTy::TTHIWAT)

{

this->TTStart();

/\* 重新设置BeginChar指向输出字符缓存队列中，未确认部分的起始处。

\* 目的在于不允许Backspace键删除写在标准输出上的内容，譬如命令提示符之类。

\*/

CRT::m\_BeginChar = this->t\_outq.CurrentChar();

}

this->TTyOutput(ch);

}

this->TTStart();

CRT::m\_BeginChar = this->t\_outq.CurrentChar();

/\* 重设BeginChar为了防止错误删除打印的字符，这里需要清空显示缓存，但会造成前面输入回显

\* 的字符在被删除时并不能被删掉，但是在实际中已经被删除了。

\*/

}

### FlushTTy

清空Tty涉及的所有队列，在此不列出代码