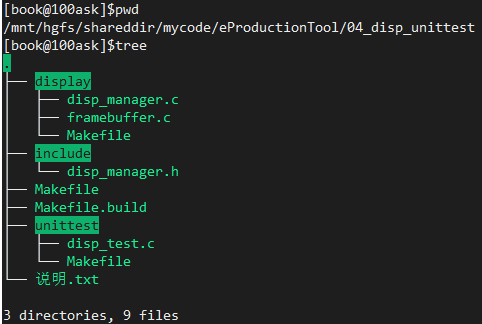
# 软件的总体框架



# 第一部分 编程逻辑以及显示单元测试

## 文件的组织



Makefile的使用参考说明.txt文件.

## 显示管理器与具体显示设备的交互

由上到下的层级结构:

应用层: disp\_test.c

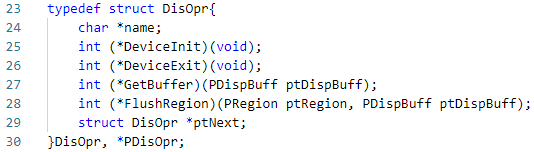
硬件抽象层: disp\_manager.c, 对底层硬件的抽象

硬件层: framebuffer.c, 以后还会添加web\_displayer.c等



显示管理器需要底层的硬件层提供以下接口的具体实现:

display\disp\_manager.c



硬件层对应函数的命名规范为xx<函数名>, 如FbDeviceInit, FbDeviceExit, FbGetBuffer, FbFlushRegion.

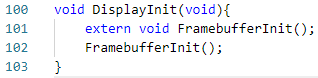
disp\_manager对framebuffer的感知通过以下函数调用链来完成:

(1) DisplayInit(), 在disp\_manager.c

(2) FramebufferInit(), 在framebuffer.c

(3) RegisterDisplay(), 在disp\_manager.c(又回去了)

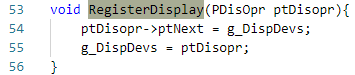
disp\_manager.c



framebuffer.c



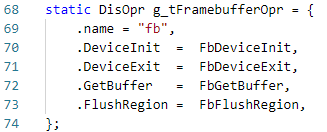
disp\_manager.c



g\_DispDevs是一个显示设备链表.

framebuffer.c向上层提供static DisOpr g\_tFramebufferOpr结构体, 这个结构体由disp\_manager.c定义, 由framebuffer.c提供实现, 即上层提供类, 下层提供实例化的对象.

display\framebuffer.c



硬件层的DeviceInit()函数主要做了两件事:

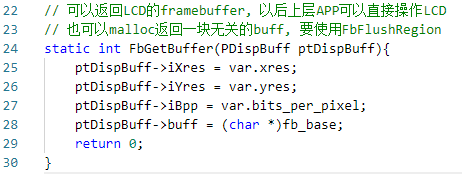
(1) 打开显示设备.



(2) 将显示设备的显存映射到内存空间, mmap().

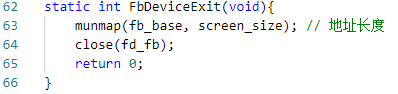


fb\_base是硬件层的私有数据, 上层通过GetBuffer()函数可得到,



最基本的显示函数PutPixel()就是在fb\_base这块空间修改数据.

对应的FbDeviceExit()就要调用munmap()和close()函数.



## 应用层的使用步骤

(1) DisplayInit(), 构建显示设备的链表.

(2) SelectDefaultDisplay("fb")选择默认的显示设备, 这里选择为framebuffer.

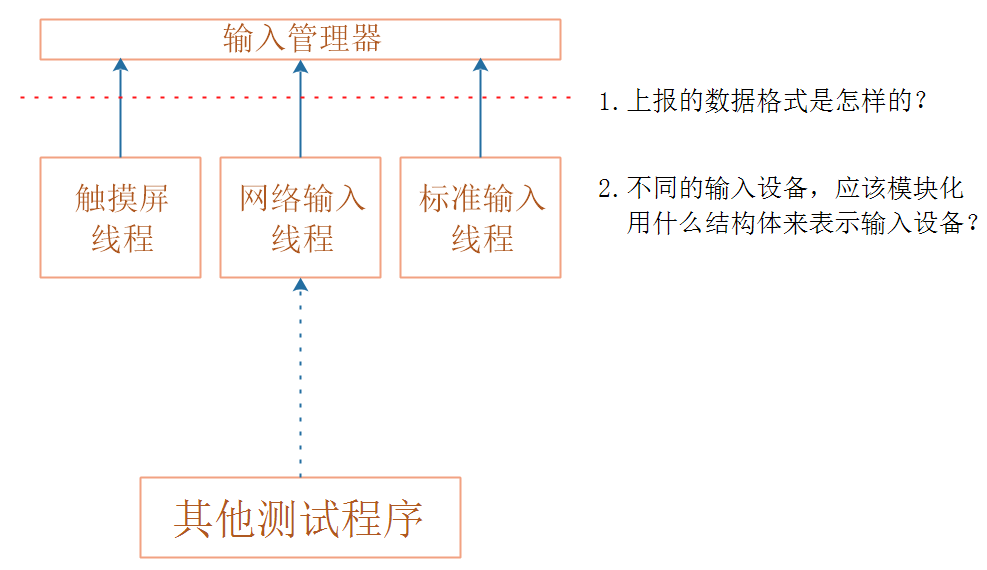
(3) InitDefaultDisplay(), 初始化默认的显示设备, 其实就是调用了DeviceInit()和GetBuffer()函数.

(4) lcd\_put\_ascii(100, 100, 'W'), 真正的画图函数.

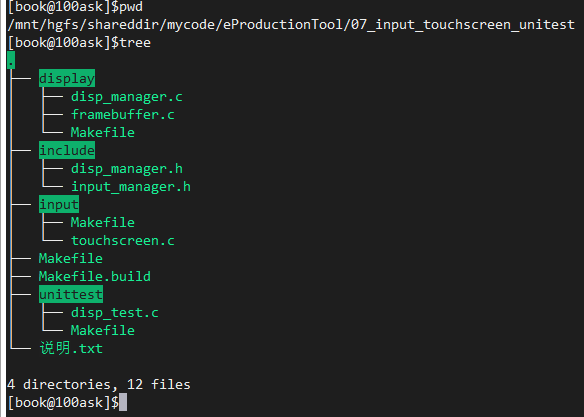
(5) FlushDisplayRegion(), 将显存刷到硬件上, 不过framebuffer不需要这一步, 只是为了兼容.

# 第二部分 输入部分---触摸屏功能

## 输入系统的层次结构



## 文件的组织



新增加了

input\_manager.h

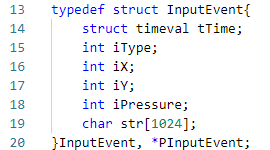
touchscreen.c

## input\_manager.h的数据结构

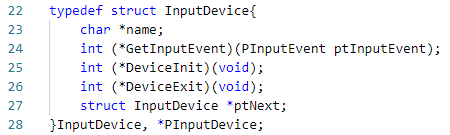
用宏定义了输入的类型, 目前有触摸屏输入和网络输入.



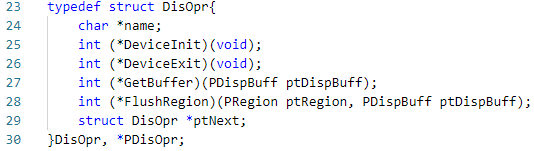
对输入事件进行了抽象, 输入时间, 类型, 坐标值和压力值, 对于网络输入事件应该有字符串来表示数据.



对输入设备的抽象

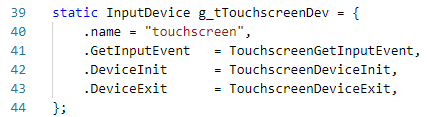


与disp\_manager.h中的DisOpr对比.

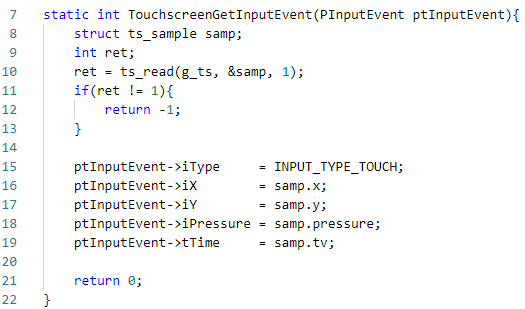


基本上对底层实际器件的抽象要包括名字, 初始化函数, 退出函数以及链表指针.

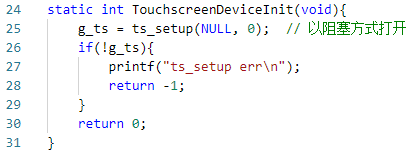
## touchscreen.c的具体实现



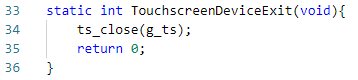
TouchscreenGetInputEvent()函数主要调用了ts\_read()获取触摸屏的输入事件, 并填充InputEvent类型的结构体.



TouchscreenDeviceInit()就是调用ts\_setup()函数.

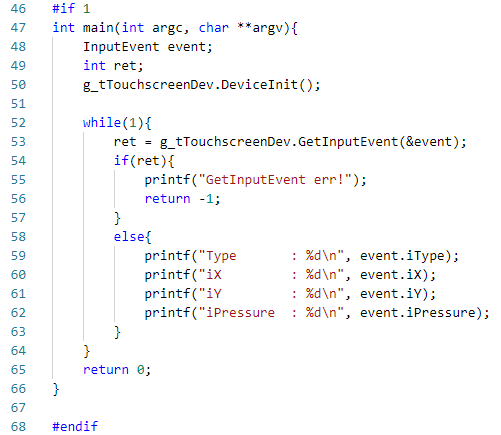


TouchscreenDeviceExit()就是调用ts\_close()函数.



## 触摸屏输入测试单元的main()函数

在touchscreen.c中编写



因为用到了tslib库, 要在顶层Makefile设置链接选项.

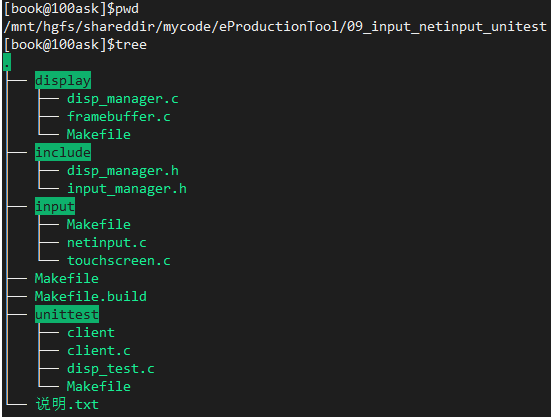


TSLIB的安装及使用可参考

D:\document\01\_all\_series\_quickstart\笔记.docx

# 第三部分 输入部分---网络输入

## 文件的组织



新增加了netinput.c文件和client.c文件.

IMXBULL作为一个UDP服务器.

## netinput.c的具体实现

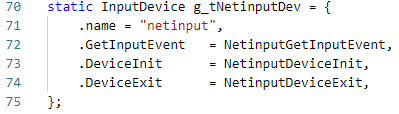
改文件主要实现了

NetinputDeviceInit(),

NetinputDeviceExit(),

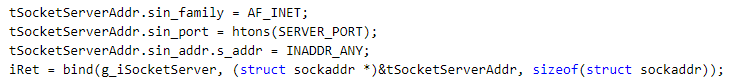
NetinputGetInputEvent(),

提供了g\_tNetinputDev结构体:



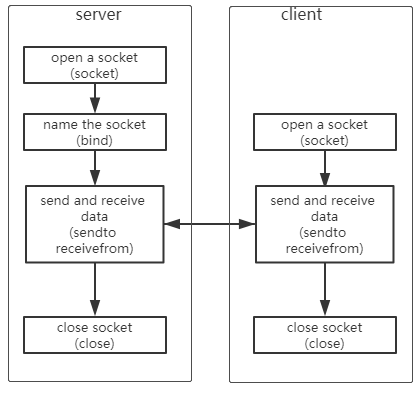
NetinputDeviceInit()主要使用了socket()和bind(),



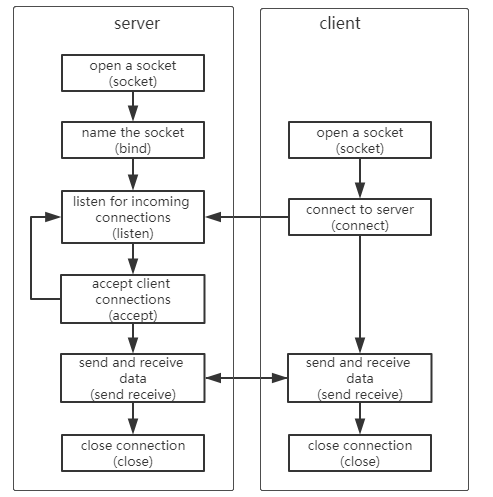


SOCK\_DGRAM 表示使用UDP协议, SERVER\_PORT被宏定义为8888

下图为UDP应用程序的工作流程

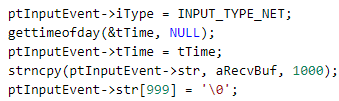


作为对比, 下图为TCP应用程序的工作流程



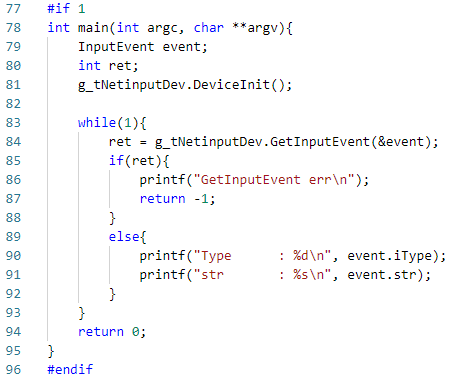
NetinputGetInputEvent()主要调用了recvfrom()函数, 并填充InputEvent类型的结构体.



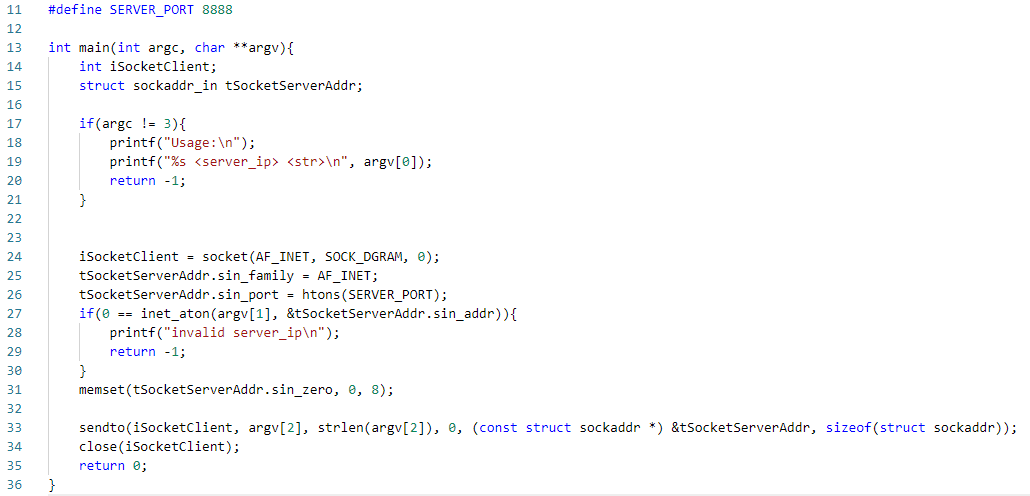


## 单元测试

服务器端代码



客户端代码, 运行在ubuntu

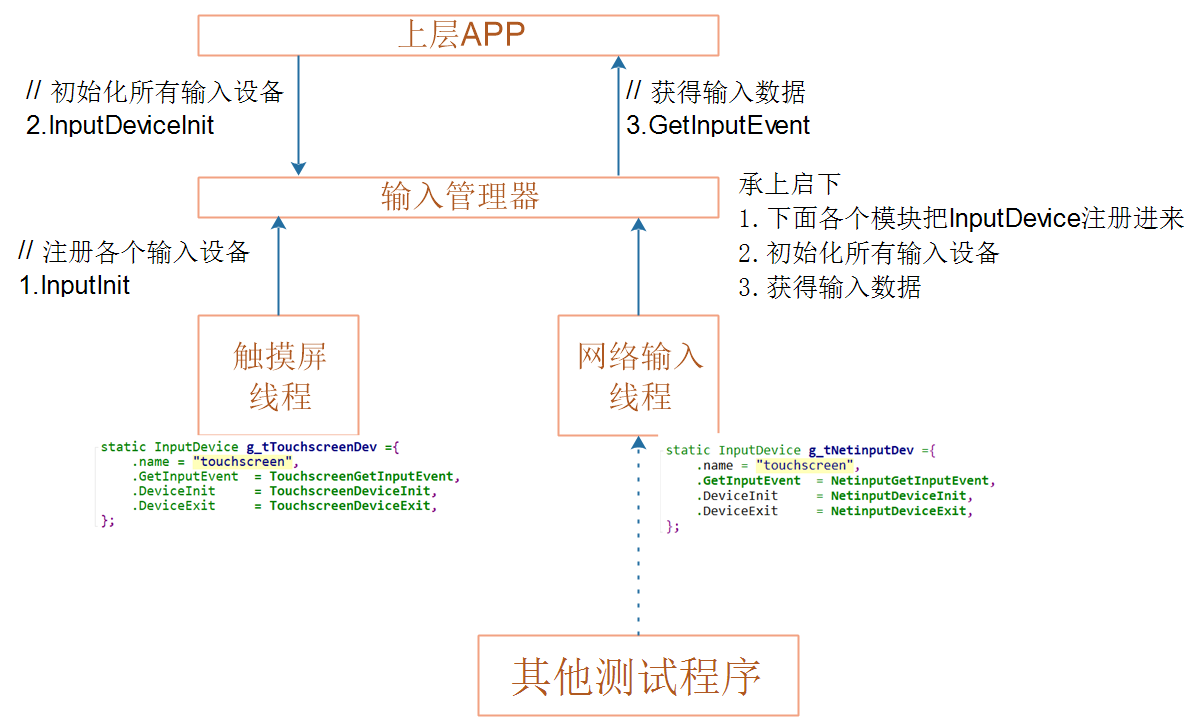


在Ubuntu中按以下命令使用, 可以在imx6ull打印出hello

$ ./client 113.54.150.139 hello

# 第四部分 输入部分---输入管理器框架

## 输入管理器所处位置



输入管理器的文件名为input\_manager.c

输入管理器主要负责三部分工作

(1) 将下面各个模块的InputDevice注册到链表中.

(2) 初始化所有输入设备, 为每个设备创建线程.

(3) 获得输入数据.

线程创建函数

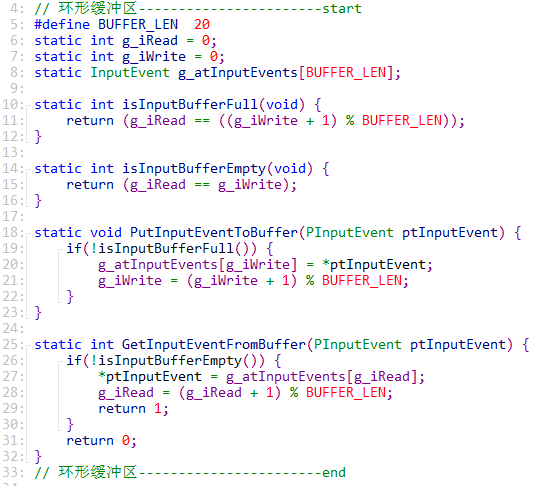
|  |
| --- |
| #include <pthread.h>  int pthread\_create(pthread\_t \*thread, const pthread\_attr\_t \*attr,  void \*(\*start\_routine) (void \*), void \*arg);  在编译时需要与-pthread链接.  成功返回0, 错误返回错误码. |

各个设备公用一个线程函数:

void \*input\_recv\_thread\_func(void \*data);

调用了每个设备的GetInputEvent().

## 环形缓冲区的实现



## 线程的同步与互斥

ubuntu默认没有pthread\_mutex\_xxx的一些函数, 需要先进行安装

|  |
| --- |
| $ sudo apt-get install glibc-doc  $ sudo apt-get install manpages-posix-dev |

|  |
| --- |
| #include <pthread.h>  int pthread\_mutex\_lock(pthread\_mutex\_t \*mutex);  给互斥量mutex上锁.  int pthread\_mutex\_unlock(pthread\_mutex\_t \*mutex);  给互斥量mutex解锁.  int pthread\_cond\_wait(pthread\_cond\_t \*cond, pthread\_mutex\_t \*mutex);  给互斥量mutex解锁, 并等待条件变量cond发出信号, 线程会进入休眠.  在返回到调用线程前, mutex又会被上锁.  int pthread\_cond\_signal(pthread\_cond\_t \*cond);  使用条件变量cond发出信号, 等待这个变量的线程会被唤醒. |

## 获取输入数据的具体实现

先定义一个互斥量和条件变量.



实现获取输入数据的函数

int GetInputEvent(PInputEvent ptInputEvent)



先对环形缓冲区上锁

如果成功读到数据则返回

否则对环形缓冲区解锁, 并设置信号量g\_tCondVar.

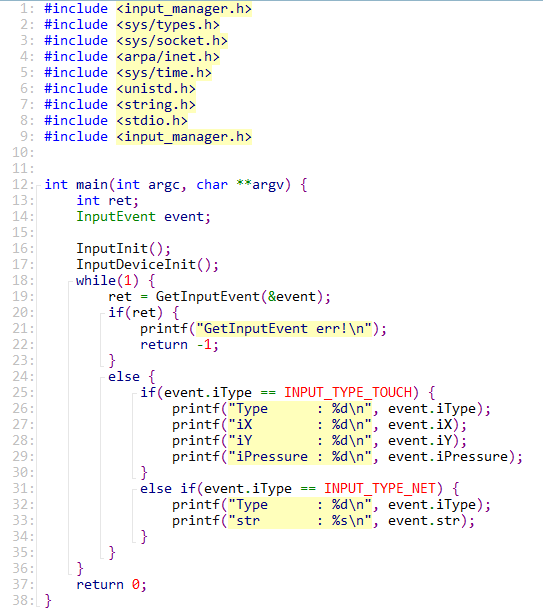
如果有输入设备获取到了设备, 则会通过g\_tCondVar唤醒该线程.

唤醒后再次读取数据, 然后对环形缓冲区解锁.

输入设备的线程处理函数, 对照着GetInputEvent()看即可.



## 单元测试代码



摸一下触摸屏或者

./client 113.54.150.139 hello

即可完成测试

# 第五部分 字符显示

## 库文件与头文件问题

编译时去哪里找头文件?

可以指定编译选项: -I dir

链接时去哪里找库文件?

可以指定链接选项: -L dir

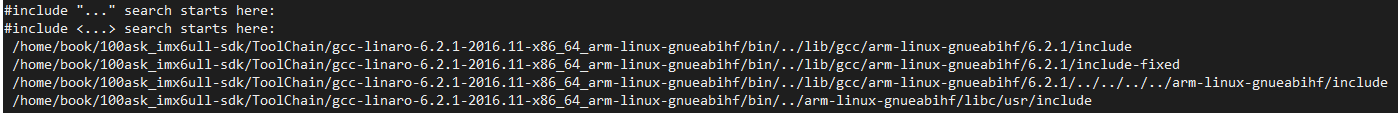
运行时去哪里找库文件?

板子上的/lib, /usr/lib目录.

也可以在运行程序前用环境变量LD\_LIBRARY\_PATH指定.

查看系统目录的方法(以IMX6ULL为例):

|  |
| --- |
| echo 'main(){}'| arm-linux-gnueabihf-gcc -E -v – |



可以看到有四个目录可以查找以<>包含的头文件, 这四个目录的前缀都是:

/home/book/100ask\_imx6ull-sdk/ToolChain/gcc-linaro-6.2.1-2016.11-x86\_64\_arm-linux-gnueabihf/bin/../

然后分别是

(1) lib/gcc/arm-linux-gnueabihf/6.2.1/include

(2) lib/gcc/arm-linux-gnueabihf/6.2.1/include-fixed

(3) lib/gcc/arm-linux-gnueabihf/6.2.1/../../../../arm-linux-gnueabihf/include

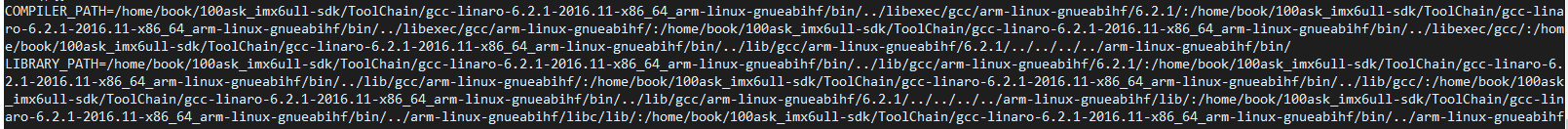
(4) arm-linux-gnueabihf/libc/usr/include

常用的stdio.h, stdlib.h就在(4)这个目录下

后面就用到目录(4), 把完整的路径写下来, 方便粘贴

|  |
| --- |
| /home/book/100ask\_imx6ull-sdk/ToolChain/gcc-linaro-6.2.1-2016.11-x86\_64\_arm-linux-gnueabihf/bin/../arm-linux-gnueabihf/libc/usr/include |

还有编译路径和链接路径



LIBRARY\_PATH=

(1)/home/book/100ask\_imx6ull-sdk/ToolChain/gcc-linaro-6.2.1-2016.11-x86\_64\_arm-linux-gnueabihf/bin/../lib/gcc/arm-linux-gnueabihf/6.2.1/:

(2)/home/book/100ask\_imx6ull-sdk/ToolChain/gcc-linaro-6.2.1-2016.11-x86\_64\_arm-linux-gnueabihf/bin/../lib/gcc/arm-linux-gnueabihf/:

(3)/home/book/100ask\_imx6ull-sdk/ToolChain/gcc-linaro-6.2.1-2016.11-x86\_64\_arm-linux-gnueabihf/bin/../lib/gcc/:

(4)/home/book/100ask\_imx6ull-sdk/ToolChain/gcc-linaro-6.2.1-2016.11-x86\_64\_arm-linux-gnueabihf/bin/../lib/gcc/arm-linux-gnueabihf/6.2.1/../../../../arm-linux-gnueabihf/lib/:

(5)/home/book/100ask\_imx6ull-sdk/ToolChain/gcc-linaro-6.2.1-2016.11-x86\_64\_arm-linux-gnueabihf/bin/../arm-linux-gnueabihf/libc/lib/:

(6)/home/book/100ask\_imx6ull-sdk/ToolChain/gcc-linaro-6.2.1-2016.11-x86\_64\_arm-linux-gnueabihf/bin/../arm-linux-gnueabihf/libc/usr/lib/

后面使用库目录(6)

## 交叉编译freetype

万能的编译命令

|  |
| --- |
| ./configure --host=arm-linux-gnueabihf --prefix=$PWD/tmp |

arm-linux-gnueabihf是编译器的前缀.

$PWD/tmp是编译好后头文件和库文件存放的地方.

tslib库不是以这样的方式指定生成的库存放的地方, 而是这样:

make install DESTDIR=$PWD/tmp

先解压缩库文件:

tar xJf freetype-2.10.2.tar.xz

tar xJf libpng-1.6.37.tar.xz

freetype依赖于libpng库, 要先安装libpng库

(1) cd libpng-1.6.37/

(2) ./configure --host=arm-linux-gnueabihf --prefix=$PWD/tmp

(3) make

(4) make install

(5) cd tmp/

(6) ls



(7) cd include/

将这个目录下的所有头文件拷贝到前面的头文件目录(4)中

(8) cp \* -rf <目录(4)>

(9) cd ../lib/



将这个目录下的所有库文件拷贝到前面的库文件目录(6)中, -d选项是为了保持连接文件的链接属性.

(10) cp -drf \* <目录(6)>

回去freetype-2.10.2/目录下

(1) ./configure --host=arm-linux-gnueabihf --prefix=$PWD/tmp

(2) make

(3) make install

(4) cd tmp/



(5) cd include/



(6) cp \* -rf <头文件目录(4)>

(7) cd ../lib/



(8) cp \* -drf <库文件目录(6)>

如果.c 是以 ANSI(GB2312)格式保存, 那么需要使用以下命令来编译

gcc -finput-charset=GB2312 -fexec-charset=UTF-8 -o test xxx.c

将freetype目录下的东西全部移动到上一层

cd /home/book/100ask\_imx6ull-sdk/ToolChain/gcc-linaro-6.2.1-2016.11-x86\_64\_arm-linux-gnueabihf/bin/../arm-linux-gnueabihf/libc/usr/include/freetype

mv \* ../

编译时还要加上-lfreetype