**Operating system developer's manual**

**操作系统开发者手册**

Book OS

系统版本：Ver0.2

修订日期：2019/2/26

**版权声明**

BookOS是一个基于x86平台的32位操作系统，其版权属于BookOS开发者所有。BookOS使用C和汇编开发，我们采用开源的方式发布，其源码遵循BSD-2协议。BSD-2协议声明如下：

BSD 2-Clause License

Copyright (c) 2017-2018, Mediocre Operating System Project Developers.

All rights reserved.

Redistribution and use in source and binary forms, with or without

modification, are permitted provided that the following conditions are met:

\* Redistributions of source code must retain the above copyright notice, this

list of conditions and the following disclaimer.

\* Redistributions in binary form must reproduce the above copyright notice,

this list of conditions and the following disclaimer in the documentation

and/or other materials provided with the distribution.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS"

AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE

IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE

DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR CONTRIBUTORS BE LIABLE

FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL

DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR

SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER

CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY,

OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE

OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

目录

[第一章 开发环境要求 5](#_Toc2118946)

[1.1 环境需求 5](#_Toc2118947)

[1.2 Windows下的环境搭建 5](#_Toc2118948)

[1.3 Linux下的环境搭建 5](#_Toc2118949)

[第二章 系统的一生 6](#_Toc2118950)

[2.1 从零到写入磁盘 6](#_Toc2118951)

[2.2 从引导到进入内核 6](#_Toc2118952)

[2.3 从初始化到完整运行 7](#_Toc2118953)

[第三章 系统的要点 9](#_Toc2118954)

[3.1 loader部分要点 9](#_Toc2118955)

[3.2 init部分要点 9](#_Toc2118956)

[3.3 console部分要点 9](#_Toc2118957)

[3.4 page部分要点 9](#_Toc2118958)

[3.5 memory部分要点 10](#_Toc2118959)

[3.6 device部分要点 10](#_Toc2118960)

[3.7 fs 部分要点 10](#_Toc2118961)

[3.8 gui 部分要点 10](#_Toc2118962)

[3.9 syscall 部分要点 10](#_Toc2118963)

[3.10 lib部分要点 11](#_Toc2118964)

[3.11 源码结构部分要点 11](#_Toc2118965)

[3.12 Makefile部分要点 12](#_Toc2118966)

[3.13 内核态与用户态要点 19](#_Toc2118967)

[第四章 一些问题的解决 20](#_Toc2118968)

[4.1 如何添加一个中断类device？ 20](#_Toc2118969)

[4.2 如何编写自己的图形界面？ 20](#_Toc2118970)

[4.3 如何编写自己的文件系统？ 20](#_Toc2118971)

[4.4 如何添加一个系统调用？ 20](#_Toc2118972)

[4.5 如何添加一个线程/进程？ 21](#_Toc2118973)

[第五章 应用程序开发 23](#_Toc2118974)

[5.1 原理介绍 23](#_Toc2118975)

[5.2 程序的编写、编译与写入磁盘 23](#_Toc2118976)

[5.3 程序的载入文件系统 26](#_Toc2118977)

[5.4 使用C++语言开发BookOs程序................................................................26](#_Toc2118977)

[后记 29](#_Toc2118978)

# 开发环境要求

## 1.1 环境需求

你所需要的工具是gcc，nasm，ld，dd，rm，make，bochs虚拟机或者qemu虚拟机。你可以在windows和linux下搭建你的环境。你需要做的就是下载好工具，配置好环境变量就可以了。

## 1.2 Windows下的环境搭建

Windows下的环境配置请参考视频（自己制作）：<https://www.bilibili.com/video/av44483615>

## 1.3 Linux下的环境搭建

Linux下的环境配置请参考如下（Boss制作）：

How to run BookOS in linux(ubuntu linuxmint...):

1.You need setup some essential packages:

sudo apt-get install gcc make build-essential

sudo apt-get install qemu qemu-user-static

sudo apt-get install nasm vgabios bochs bochs-x bximage

2.Changing your current dictionary to src and make:

cd $workdir/src

make

3.Have fun.

到此，你就已经搭建好了开发环境，可以开始你的屌丝人生了。^-^

# 系统的一生

## 2.1 从零到写入磁盘

最开始，我们把代码写成源程序，通过nasm和gcc分别编译.c和.asm文件，生成.o目标文件，通过ld连接成我们的所需要的可执行文件。由于boot和loader分别只有一个文件，所以直接把他们生成可执行文件。内核的话就需要编译链接成可执行文件。于是，我们就有了boot/boot.bin（二进制）、boot/loader.bin（二进制）和kernel/kernel.elf（ELF格式文件，其实也可以做成二进制，但ELF优点多，我们就选用ELF文件）。

于是，我们的操作系统所需要的文件就制作完成，下一步就是把他们写入磁盘，使他们可以运行。由于软盘是可以直接读取某个扇区的数据，所以我们直接把这些文件写入到一个绝对扇区位置，然后在运行的时候，从软盘对应的地方读取数据就可以了。写入工具是dd，我把boot.bin写入到0扇区（引导必须位于0扇区），loader.bin写到2扇区（偶数貌似好看点），loader我固定使用4KB（8扇区），所以，我把内核放到了10扇区，对于内核的扇区数，可以根据内核的大小进行调整，目前写入400个扇区（即使内核没有200KB也不要紧，就相当于后面写入的是数据为空的扇区）。这些可以再src/Makefile中查看。

此时，我们的操作系统就成功地住在软盘上了。

## 2.2 从引导到进入内核

大家都知道，电脑开机后进行一系列操作之后就会检测是否有启动扇区，有的话就把启动扇区加载到0x7c00这个内存地址。然后跳到那个地方执行我们的系统。在BookOS中，boot的作用就是加载loader，因为boot只能存放512字节，所以我们把大任交给loader，它先从软盘读取loader进入内存，然后跳入loader执行。这下，loader就开始忙碌地工作了（真勤奋啊）。

我们的loader先从磁盘上读取内核进入内存，这之后，就没有软盘操作了，然后关闭软盘马达，不让他运转（节约用电？Maybe）。接下来就检测我们有多少内存，并把这些信息放入到ARDS结构体中，我们在内核中初始化内存管理的时候需要用到这些信息。下一步，初始化图形模式，根据我们定义的图形模式（在boot/const.inc），切换到指定的图形模式。

“好累呀！不过还是得继续做事情。”loader接下来做的便是进入保护模式，先关闭中断（以免发生中断后出现错误），然后再加gdt描述符表，再打开A20总线（让我们可以访问1MB以上的内存），然后设置CR0的最低一位为1就变成保护模式了，下一步是清空处理器流水线，用一个跳转就行。

此后，我们便进入保护模式了，现在访问内存不是 段：偏移了，而是 段选择子：偏移，所以先初始化数据段，栈段，其他段的选择子和栈顶。由于数据段和代码的基地址设置成0，所以偏移的地址就是对应的程序内的地址。在前面说过，我们的内核是elf格式的，不能直接执行，所以需要解析一下，于是就把内核文件的内容解析到另外一个地址，这样的话，我们就可以直接跳转到对应的地址执行内核了。“咚”，进入内核！

## 2.3 从初始化到完整运行

自从loader因为无事可做，抑郁而死之后，kernel登基了。我们内核的入口函数是\_start，所以内核先会在\_start里面执行。在loader里面在此初始化段选择子以及栈顶（loader栈顶和kernel栈顶不一样），然后调用main函数进入init/mian.c中的main函数初始化整个系统。

首先是初始化系统的分页机制，从此我们访问内存就需要访问页目录表内定义了的地址才行。接下来是初始化GDT/IDT，我们重新定义了GDT，并且初始化了IDT,于是就能够正确处理系统产生的异常和外部产生的中断（IRQ中断和系统调用）。接下来是初始化控制台，这个控制台主要用于文本模式下输出信息，这是没有开启图形模式前用于输出信息的初始化，现在有图形了，可以把这部分去掉，但是不能忘了本，有新欢就把旧爱抛弃了，万一哪天图形出问题了，咱们还可以回到文本模式做测试对吧（应该不会出问题，比竟我们是最棒的开发者！）

接下来，初始化内存管理模块，内存管理模块是根据分页来进一步进行管理的。分页管理中提供了可以分配/释放整个页（1 PAGE=4096 bytes= 4 kb）的函数。因此内存管理是基于分页管理来的，我们可以分配更小的空间，也可以分配更大的空间。接下来初始化图形模式的基本信息，获取屏幕的宽高，显存地址，像素宽度等。并且显示系统logo。再然后初始化cpu，获取cpu的相关信息。而后，初始化系统调用表，把系统调用表填写好，产生系统调用后会执行表格内对应的函数。

要迎来新的黎明了，多线程/多进程即将到来。首先是初始化多线程环境（进程也相当于一个线程，只不过拥有一套自己的资源），并把当前执行流当做mian thread主线程，同时初始化一个idle线程（当没有其它线程和进程运行的时候来运行它）

此时我们的系统还不会发生线程/进程切换，因为我们的任务切换时基于时钟中断的，每个0.01秒检测一次是否需要切换任务。那么接下来就是初始化时钟中断，初始化完之后还不能进行任务切换，必须开启中断才会发生任务切换，所以我们io\_sti打开中断，此时，就可以发生任务切换了。

我们的main thread 主线程还在继续工作，它现在初始化硬盘（因为硬盘初始化会发出检测硬盘信息的命令，会产生中断，所以必须在此之前打开中断，硬盘中断程序才能相应）。硬盘可能有多个，我们会将有的IDE硬盘都初始化（最多4个硬盘）。然后选择hda第一个硬盘，并且在它上面初始化一个Fatxe文件系统。如果我们有要往文件系统中写入文件的需要，也就是说WRITE\_DISK为1时，我们写入一个名字为WRITE\_NAME，大小为FILE\_SECTORS的文件进入文件系统。这个机制是我用来把可执行文件，文本或者图形文件写入我们的文件系统所写的一套方法，你也可以自己实现一种，毕竟，条条大路通罗马。然后ls一下，看我们的文件系统里面有什么文件。

接下来就是初始化键盘，鼠标，并且开启几个线程，keyboard线程用于处理键盘操作产生的数据，mouse线程用于处理鼠标操作产生的数据，clock线程用于每次发生时钟中断后，把当前的时间增加，这样我们的时间就是变化的（不然就只有一个启动时的时间，你都不知道你启动了多久了）

由于之前初始化图形后就显示了一个logo，所以屏幕上只有现实的logo，没有其它内容。在这里，我们初始化图形界面，就可以替代原有的logo，现实自己的内容了。这样做的原因是，启动后就显示logo，中间初始化其它内容需要一定时间，等它们都初始化完后，在初始化这里的图形界面，这样，就有一种类似于Windows启动的感觉了。

目前都是运行在内核中的，所以我们初始化一个进程，进程运行在用户态，但是呢，init进程是运行在内核中的，它拥有进程的所有东西，只是说它的执行入口在内核里面，和一般的进程没有区别，其他的进程我们将采用从磁盘加载的方式。而后，主线程就进入一个循环，先判断是否有被强制结束的进程，有的话就去把该进程从系统中释放掉，然后在休眠100毫秒。

就这样主线程的所作所为已经被我们看的清清楚楚，我们已经看清他这个人了。（斜眼笑）

那么init做了什么呢？在src/kernel/thread.c中我们可以找到init，它其实也没做啥，就是从磁盘加载了一个boshell程序，然后循环等待子进程结束。那么刚好，boshell就是我们进入图形界面后看到的那个shell程序。Shell程序就等待用户输入命令，并作出相应的操作。

从此，我们的系统就进入就绪状态，等待和用户交互了。

# 系统的要点

## 3.1 loader部分要点

***要点1：***

加载内核的时候，如果内核大小变大了，就需要把在LoadeKernel后面把内核的扇区数和写入磁盘的扇区数满足，只要比内核实际的大小大都可以的。与此同时，他也会加载一个应用程序的数据进入一个内存地址，我们将在后面写入文件系统的时候使用。

***要点2：***

在切换图形模式的时候，在const.inc中修改要切换的模式。如果不想开启图形模式就把切换到指定的模式的那个int注释掉，不产生中断请求，就不会执行，后面就会进入文本模式，就不能初始化图形（当然这个功能可能不太会用到）

## 3.2 init部分要点

***要点1：***

在src/init/main.c中，会执行很多初始化操作，他们之间其实是有关联的，也就是不能随便修改他们的顺序。例如在初始化内存管理中，需要调用到分页管理里面的函数，因此，就不能把内存管理放到分页机制前初始化，不然会出错。

***要点2：***

你可以修改WRITE\_ID来往文件系统写入不同的文件，这样就比较方便从外部导入文件到文件系统了。

## 3.3 console部分要点

***要点1：***

在初始化完console之后，你就可以调用printk来打印你想要输出的信息，printk也许还不太完善，你可以自己去完善，让它可以支持更多格式。

## 3.4 page部分要点

***要点1：***

在初始化完page之后，你就可以调用kernel\_alloc\_page和kernel\_free\_page来分配和释放N个页。

***要点2：***

在分页机制中，我把低端2G内存给内核使用，用户使用高端2G，这通过页目录表映射之后就可以实现。

## 3.5 memory部分要点

***要点1：***

在初始化完memory之后，你就可以调用kmalloc和kfree来分配和释放内存，便于后面的开发，他们是在内核下面的内存分配和释放的函数。

***要点2：***

如果你在开发系统的时候需要分配大量内存，如果害怕内存不够，可以在你的虚拟机配置的时候配置更多内存。

## 3.6 device部分要点

***要点1：***

在device中，有中断设备（时钟，鼠标，键盘，硬盘，以及未实现的网络），也有非中断设备（视频，ramdisk），在中断设备中，他们都很类似，如果要实现自己的某个设备，可以仿造他们的方法来实现（例如网络）。

***要点2：***

如果是中断设备，要记得到descriptor中注册一个中断描述符，以及在interrupt中注册一个入口函数。

## 3.7 fs 部分要点

***要点1：***

在文件系统中，删除小文件（几kb）没有问题，如果文件太大就可能会出现问题，所以大家在使用的时候稍微注意一下，后面更新文件系统后可能会好点。

***要点2：***

文件系统是建立在硬盘的读写函数之上的，你可以根据这两个基层函数建立一套自己的文件系统，也可以去移植ext2或者fat32文件系统，但是要完善一点，不然的话，残缺了就不好了。

## 3.8 gui 部分要点

***要点1：***

图形界面是基于绘制像素点进入显存来实现的，因此在video中已经有了绘制和读取像素的函数，我们后面只需要在这基础上进行开发就行了。也可以在视频信息的结构体获取屏幕宽高。

## 3.9 syscall 部分要点

***要点1：***

系统调用是一个数组，里面存放的是函数的地址，一个新的系统调用就需要有一个新的NR，并且在你系统调用函数的前面加上sys\_来表示它是一个系统调用函数。

## 3.10 lib部分要点

***要点1：***

Lib下面的函数分为2种，一种是独立可以被调用的，一种是调用系统调用的接口。系统调用接口使用汇编编写的，因为要调用int软中断，来实现调用系统内部的函数。你可以把某一类的函数写到一个文件里面，例如文件操作的都写到file.asm中，也可以把他们独立出来fopen.asm, fread.asm,fwrite.asm…都可以的。

## 3.11 源码结构部分要点

***要点1：***

打开BookOS，你会发现有这些目录和文件，bin，bochs，doc，img，src，LICENSE，README.md。

Bin目录下面存放的是应用程序的源代码以及其Makefile，在需要的是后可以重新编写代码。

Bochs目录下存放的是bochs虚拟机的配置文件，如果你想用bochs调试，你可以使用到这个文件。

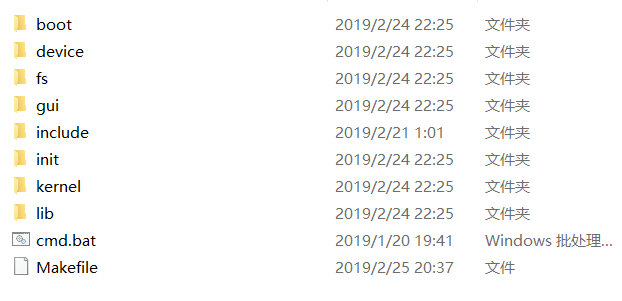
Doc目录下存放的是一些文档，一些log，bug，note之类的东西。可以把内容写在这里面，以辅助开发。

Img下面存放的是磁盘镜像，a.img表示软盘，c.img表示第一个硬盘。还可以添加d.img,e.img.f.img等等。

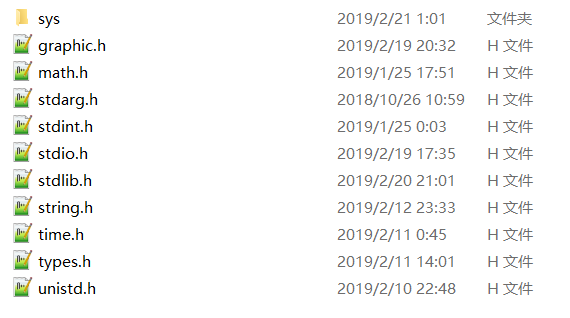
LICENSE是源码开放协议，BookOS遵循BSD-2协议开放。

README.md是BookOS的readme文件。

Src目录下是存放的系统源代码。



在boot中存放的是引导代码，device中存放的是设备/驱动文件，fs存放的是文件系统的代码，gui存放的是图形界面的代码，init存放的是初始化的代码，kernel存放的是内核代码，lib存放的是库文件代码，makefile是整个系统编译运行的主管，cmd是用于在windows下面打开控制台后快速进入当前目录。Include存放的是头文件代码，在include中有个sys目录，下面存放的是系统相关的代码，例如内存管理，进程管理，文件系统，图形界面的头文件等。

而直接在include下面的其他文件是一些类似于标准库的代码，这些既可以在内核中使用，也可以在用户程序中使用，用户程序只需要调用这些头文件。

***要点2：***

如果你想添加一个文件，记得在对应的目录中添加，例如要添加一个kernel内部的文件，就添加到src/kernel目录文件下，其头文件应当放在src/include/sys下面。如果是添加一个系统调用的接口，把文件放在src/lib下面，头文件放到src/include下的某个头文件和或者自己添加的头文件里面。并且记得添加一个文件后，得到对应目录下面的makefile中修改添加该文件，并且还要到src/makefile这个主管里面添加该文件。

## 3.12 Makefile部分要点

***要点1：***

搞清楚Makefile后就知道整个系统是如何编译，链接，写入磁盘，虚拟机运行的了。先打开src/makefile，可以发现这里面就是设置了一些工具的变量，以方便在后面的引用。由于在环境搭建的时候设置了环境变量，这个地方可以直接使用这些工具的名字。

|  |
| --- |
| #tool dir  BOCHS\_DIR = ../bochs/  NASM = nasm  CC = gcc  LD = ld  DD = dd  QEMU = qemu-system-i386  BOCHS = bochs  BXRC = $(BOCHS\_DIR)bochsrc.bxrc |

后面也是定义一些变量，这些事img镜像的位置，在写入磁盘的时候需要用到。如果要更改写入的镜像，可以在这儿修改。

|  |
| --- |
| #img dir  IMG\_DIR = ../img/  FLAPPY\_IMG = $(IMG\_DIR)a.img  HDA\_IMG = $(IMG\_DIR)c.img |

再往后就是连接所需要的参数，如需要更改连接的方式，可以在这儿修改。

|  |
| --- |
| #flags  LDFLAGS = -m elf\_i386 -e \_start -Ttext 0x100000 |

然后定义了loader和kernel写入磁盘的扇区偏移以及扇区数量。

|  |
| --- |
| #system disk  LOADER\_OFF = 2  LOADER\_CNTS = 8  KERNEL\_OFF = 10  KERNEL\_CNTS = 400 |

为了能让主makefile调用其它目录下面的makefile，我们需要把其它目录写进来。

|  |
| --- |
| #src dir  BOOT\_DIR = ./boot/  KERNEL\_DIR = ./kernel/  INIT\_DIR = ./init/  LIB\_DIR = ./lib/  DEVICE\_DIR = ./device/  GUI\_DIR = ./gui/  FS\_DIR = ./fs/ |

然后就是要输出那些文件，输出到哪个地方。

|  |
| --- |
| BOOT\_BIN = $(BOOT\_DIR)boot.bin  LOADER\_BIN = $(BOOT\_DIR)loader.bin  KERNEL\_ELF = $(KERNEL\_DIR)kernel.elf |

因为链接操作是在这个主管makefile里进行的，所以我们需要知道其它目录产生的目标.o文件。因此，我们用一个OBJS变量保存，如下：

|  |
| --- |
| #objs  OBJS = $(KERNEL\_DIR)\_start.o\  $(KERNEL\_DIR)x86.o\  $(KERNEL\_DIR)descriptor.o\  $(KERNEL\_DIR)8259a.o\  $(KERNEL\_DIR)interrupt.o\  $(KERNEL\_DIR)console.o\  $(KERNEL\_DIR)bitmap.o\  $(KERNEL\_DIR)page.o\  $(KERNEL\_DIR)kernel.o\  $(KERNEL\_DIR)memory.o\  $(KERNEL\_DIR)ards.o\  $(KERNEL\_DIR)cpu.o\  $(KERNEL\_DIR)ioqueue.o\  $(KERNEL\_DIR)debug.o\  $(KERNEL\_DIR)cmos.o\  $(KERNEL\_DIR)syscall.o\  $(KERNEL\_DIR)thread.o\  $(KERNEL\_DIR)sync.o\  $(KERNEL\_DIR)tss.o\  $(KERNEL\_DIR)process.o\  $(KERNEL\_DIR)system.o\  $(FS\_DIR)dir.o\  $(FS\_DIR)fatxe.o\  $(FS\_DIR)fat.o\  $(FS\_DIR)file.o\  $(INIT\_DIR)main.o\  $(GUI\_DIR)main.o\  $(GUI\_DIR)graphic.o\  $(GUI\_DIR)font.o\  $(GUI\_DIR)ft\_simsun.o\  $(GUI\_DIR)ft\_standard.o\  $(GUI\_DIR)image.o\  $(DEVICE\_DIR)vga.o\  $(DEVICE\_DIR)clock.o\  $(DEVICE\_DIR)keyboard.o\  $(DEVICE\_DIR)harddisk.o\  $(DEVICE\_DIR)video.o\  $(DEVICE\_DIR)ramdisk.o\  $(DEVICE\_DIR)mouse.o\  $(LIB\_DIR)string.o\  $(LIB\_DIR)vsprintf.o\  $(LIB\_DIR)math.o\  $(LIB\_DIR)printf.o\  $(LIB\_DIR)malloc.o\  $(LIB\_DIR)free.o\  $(LIB\_DIR)exit.o\  $(LIB\_DIR)getchar.o\  $(LIB\_DIR)putchar.o\  $(LIB\_DIR)write.o\  $(LIB\_DIR)execv.o\  $(LIB\_DIR)wait.o\  $(LIB\_DIR)fopen.o\  $(LIB\_DIR)fclose.o\  $(LIB\_DIR)fread.o\  $(LIB\_DIR)fwrite.o\  $(LIB\_DIR)fstat.o\  $(LIB\_DIR)lseek.o\  $(LIB\_DIR)unlink.o\  $(LIB\_DIR)opendir.o\  $(LIB\_DIR)closedir.o\  $(LIB\_DIR)readdir.o\  $(LIB\_DIR)rewinddir.o\  $(LIB\_DIR)mkdir.o\  $(LIB\_DIR)rmdir.o\  $(LIB\_DIR)rename.o\  $(LIB\_DIR)copy.o\  $(LIB\_DIR)move.o\  $(LIB\_DIR)getcwd.o\  $(LIB\_DIR)chdir.o\  $(LIB\_DIR)clear.o\  $(LIB\_DIR)ps.o\  $(LIB\_DIR)reboot.o\  $(LIB\_DIR)random.o\  $(LIB\_DIR)access.o\  $(LIB\_DIR)gettime.o\  $(LIB\_DIR)graphic.o\  $(LIB\_DIR)mouse.o\  $(LIB\_DIR)igl.o\  $(LIB\_DIR)video.o\  $(LIB\_DIR)mm.o\ |

准备工作就开始了。接下来就是开始真正地做事情了。由于makefile的第一个命令开始是文件中的第一个标签，那么就找到了all: compile link disk qemu，默认的话就是执行这里面的操作，可以知道，它的第一步就是compile，于是找到compile如下：

|  |
| --- |
| #compile file  compile:  cd $(BOOT\_DIR) && $(MAKE)  cd $(KERNEL\_DIR) && $(MAKE)  cd $(INIT\_DIR) && $(MAKE)  cd $(LIB\_DIR) && $(MAKE)  cd $(DEVICE\_DIR) && $(MAKE)  cd $(GUI\_DIR) && $(MAKE)  cd $(FS\_DIR) && $(MAKE) |

它所做的就是进入某个目录，然后运行make，引用该makefile执行操作，执行完后会返回这儿，然后执行下一个cd，知道把所有子目录都make完。因此compile就完成了，产生了许多.o文件以及boot.bin和loader.bin，.o文件是用来链接成kernel.elf的。

下一步就是链接，看看它的代码：

|  |
| --- |
| link: $(KERNEL\_ELF)  $(KERNEL\_ELF): $(OBJS)  $(LD) $(LDFLAGS) -o $(KERNEL\_ELF) $(OBJS) |

它要产生KERNEL\_ELF,于是就会执行$(LD) $(LDFLAGS) -o $(KERNEL\_ELF) $(OBJS)

这句代码，由此，链接操作完成，生成了kernel.elf文件。

接下来就是把文件写入磁盘了。在这里，默认把boot写入第0个扇区，loader写入第二个扇区，kernel写入第10个扇区。

|  |
| --- |
| #write file into disk  disk:  $(DD) if=$(BOOT\_BIN) of=$(FLAPPY\_IMG) bs=512 count=1 conv=notrunc  $(DD) if=$(LOADER\_BIN) of=$(FLAPPY\_IMG) bs=512 seek=$(LOADER\_OFF) count=$(LOADER\_CNTS) conv=notrunc  $(DD) if=$(KERNEL\_ELF) of=$(FLAPPY\_IMG) bs=512 seek=$(KERNEL\_OFF) count=$(KERNEL\_CNTS) conv=notrunc |

这样，我们的系统就住在软盘上了。等运行的时候就可以启动咱们的系统了，好激动。

默认是使用qemu虚拟机启动，你也可以选择bochs，甚至可以把镜像放到virtual box或者vmware虚拟机中运行。

|  |
| --- |
| #run kernel in vm  bochs:  $(BOCHS) -q -f $(BXRC)  qemu:  $(QEMU) -m 64 -fda $(FLAPPY\_IMG) -hda $(HDA\_IMG) -boot a |

由此，我们的系统就在虚拟机中飞速运行啦，qemu比bochs快，你可以自己选择。

我们还有一个clean标签，可以用它来清除编译过程中产生的临时文件，让我们的源码变得干净。

|  |
| --- |
| #clean temporary files  clean:  cd $(BOOT\_DIR) && $(MAKE) clean  cd $(KERNEL\_DIR) && $(MAKE) clean  cd $(INIT\_DIR) && $(MAKE) clean  cd $(LIB\_DIR) && $(MAKE) clean  cd $(DEVICE\_DIR) && $(MAKE) clean  cd $(GUI\_DIR) && $(MAKE) clean  cd $(FS\_DIR) && $(MAKE) clean |

至此，主makefile就完了，你可以根据自己的需求添加其他的标签，满足自己的需要。

***要点2：***

除此之外，再看一个子makefile，就能明白整个流程了。首先也是进行准备操作，定义变量保存工具和头文件路径。

|  |
| --- |
| #The tools name  NASM = nasm  CC = gcc  #The file path  INCLUDE\_PATH = ../include/ |

然后定义gcc和nasm的编译参数：

|  |
| --- |
| #flags  ASM\_FLAGS = -I $(INCLUDE\_PATH) -f elf  C\_FLAGS = -I $(INCLUDE\_PATH) -c -fno-builtin -Wall -Wunused |

就像这样，会在后面编译的时候引用。

用OBJS把所有的对象文件列出来，我们根据列出来的文件来选择那些事要编译的。

|  |
| --- |
| OBJS = \_start.o\  x86.o\  descriptor.o\  8259a.o\  interrupt.o\  console.o\  bitmap.o\  page.o\  kernel.o\  memory.o\  ards.o\  cpu.o\  ioqueue.o\  debug.o\  cmos.o\  syscall.o\  thread.o\  sync.o\  tss.o\  process.o\  system.o |

然后就是第一个标签是需要操作的标签，因此如下

|  |
| --- |
| #First read here  .PHONY: all  all: compile  #Compile files  compile: $(OBJS) |

可以看出，默认就是编译文件。于是便会引用到自动编译汇编和c语言的命令。

|  |
| --- |
| %.o: %.asm  $(NASM) $(ASM\_FLAGS) -o $@ $<    %.o: %.c  $(CC) $(C\_FLAGS) -o $@ $< |

就是像这样，%号就是任意一个文件名。后面就是后缀名，汇编文件是NASM编译，gcc文件是CC编译。这样就会逐渐编译每一个文件。

与此同时你也可以在里面输入clean来清楚产生的临时文件。

|  |
| --- |
| #Clean temporary files  clean:  -rm \*.o  -rm kernel.elf |

由此，makefile就解析完了，你就知道整个系统是怎么编译，链接，运行的了。

## 3.13 内核态与用户态要点

***要点1：***

我们要注意某个线程/进程对一个函数的调用，如果你是一个内核线程，那么你可以调用内核态的函数，也可以调用用户态的函数（不会出错，但尽量不要调用，毕竟要通过中断才可以调用到内核里面的函数），用户态的函数只能调用库函数，不能直接调用内核态的函数，不然会出错。

***要点2：***

用户进入内核调用某个函数，是通过系统调用中断实现的，他要通过发生中断来切换运行的特权级，从3特权级到0特权级，然后调用0特权级的函数，就不会发生错误了，调用完后就会切换特权级，从0特权级切换到3特权级。返回到用户态继续执行3特权级的函数。

# 一些问题的解决

## 4.1 如何添加一个中断类device？

我们拿Keyboard来举例。

第一步，创建一个keyboard.c。

第二步，写一个init\_keyboard函数，用于初始化keyboard

第三步，写一个keyboard\_handler函数，用于初始化处理键盘中断

第四步，在init\_descriptor中初始化一个keyboard对应的中断描述符

set\_gate\_descriptor(idt + 0x20+KEYBOARD\_IRQ, (int )&IRQ\_keyboard, 0x08, DA\_386IGate, 0);

这里涉及到一个IRQ\_keyboard，他是键盘中断发生后的入口，是汇编函数编写的。

第五步，在interrupt.asm中编写IRQ\_keyboard函数，当发生中断后，会首先执行这个函数，这个函数会调用keyboard\_handler来进行c语言部分的处理。

到这里，就可以接受keyboard中断了，如果要进行其它处理，可以自己添加函数处理。

## 4.2 如何编写自己的图形界面？

由于底层的像素写入和读取函数已经实现了，就可以自己实现一个图形界面了，也可以使用原生的图形界面，在那个基础之上进行图形编程。

/\*把颜色值写入显存\*/

void vram\_write\_pixel (int32 x, int32 y,uint32 color);

/\*从显存中读取一个像素保存到color里面\*/

void vram\_read\_pixel (int32 x, int32 y,uint32 \*color);

## 4.3 如何编写自己的文件系统？

文件系统是基于扇区的读/写函数来实现的，所以只要有了这两个函数，你就可以实现任何类型的文件系统了。

/\*读取几个扇区进入buf\*/

void hd\_read\_sectors (uint32 lba, void \*buf, uint32 counts);

/\*从buf中写入几个扇区到磁盘\*/

void hd\_write\_sectors (uint32 lba, void \*buf, uint32 counts);

## 4.4 如何添加一个系统调用？

如果你想要让用户程序调用内核函数，你需要添加一个系统调用，让用户程序陷入内核态执行内核里面的某个函数。

我们拿\_NR\_FOPEN来举例：

第一步，编写你的系统里面的函数

int32\_t sys\_open(const char \*pathname,uint8\_t flags);

第二步，需要在syscall.h中添加一个系统调用函数的编号

#define \_NR\_FOPEN 10

如果你要添加的编号超过了MAX\_SYS\_CALLS，你需要更新MAX\_SYS\_CALLS的大小，因为它是一个数组，你要把函数的地址存放在这个数组里面才行。

第三步，你需要在src/kernel/syscall.c里面把函数地址放到sys\_call\_talbe里面对应的位置上去。

sys\_call\_table[\_NR\_FOPEN] = sys\_open;

这样，当系统调用发生的时候，他就可以找到sys\_open这个函数了。

第四步，编写用户的接口。把它保存到src/lib目录下面。我给它起名叫fopen.asm，他是一个汇编文件。

|  |
| --- |
| [bits 32]  [section .text]  INT\_VECTOR\_SYS\_CALL equ 0x80  \_NR\_FOPEN EQU 10  global fopen  fopen:  mov eax, \_NR\_FOPEN  mov ebx, [esp + 4]  mov ecx, [esp + 8]  int INT\_VECTOR\_SYS\_CALL  ret |

在这里先声明成32位程序，.text代码段，然后有个系统调用号，这个我们和linux一样，都是0x80，我们让\_NR\_FOPEN为10，和我们之前定义的一样。下面就是写了fopen的标签，并且把它变成全局标签，也就是导出去，让其它函数可以调用到。eax的值是系统调用函数对应的NR号，然后ebx,ecx来进行参数的传递，再调用系统调用，完了后就返回，这样，只要调用fopen，就可以到内核里去调用某个函数。（这里的参数传递，我们最多只支持4个参数分别是ebx,ecx,edx,esi）

第五步，写一个头文件，fopen(const char \*pathname,uint8\_t flags);，然后放到include/stdio.h里面，只要某个程序导入了这个头文件就可以调用这个函数了。

第六步，把fopen.o写入src/lib/Makefile的OBJS，与此同时，为了让整个系统也知道它，还要把fopen.o写入src/Makefile，这样内核就能知道它了

第七步，当你需要写某个程序的时候，你也需要把fopen.o写到你程序的Makefile里面，例如/bin/test/Makefile里面的OBJS后面。

就这样，你就可以添加一个系统调用了。

## 4.5 如何添加一个线程/进程？

我们的线程是运行在内核中的，所以只能在内核里面创建线程，那这些线程有什么用呢？它可以来处理中断产生的数据，也可以是一段运行在内核里面的代码。那么如何添加一个内核线程呢？其实很简单的。

struct thread \*thread\_start(char \*name, int prio, thread\_func function, void \*arg);

Name参数是线程的名字，prio是线程的特权级，function是线程要执行的函数，arg是传入的参数。返回值是一个thread结构体。这样，你就可以生成一个线程，它可以执行系统中的某段代码。

进程的话，有运行在内核中的进程和执行磁盘程序的进程。

int process\_execute(void \*filename, char \*name);

Filename是进程的入口地址，name是进程的名字，返回进程的pid。这个进程是有自己的资源（斜眼笑）的，有页目录表，一套完整的映射。不过，他是直接运行某个函数，并把那个函数当做一个进程。

int sys\_execv(char \*path, char \*argv[]);是只从磁盘加载一个程序运行。Path是路径，argv是传入的参数，返回进程pid。这个函数可以在内核调用，他还有个用户接口int execv(char \*path, char \*argv[]); 这个函数可以在应用程序中调用，以执行另外一个程序。只要在文件系统中有某个可执行文件，你就可以把它拿来执行了。

# 应用程序开发

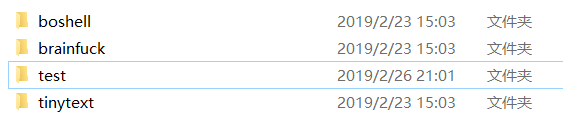
## 5.1 原理介绍

对于应用程序的开发，我们先来搞懂它的原理。

首先，我们得编写我们的程序代码，如果调用需要调用库函数，那么导入对应的头文件就可以了。然后，需要对代码进行编译，编译成.o目标文件。再和库函数的.o文件进行链接，生成可执行二进制文件。接下来，把文件写入磁盘，我们这里是软盘。在loader中，我们会把磁盘上的文件读入到一个内存里面。在文件系统初始化完之后，我们把那个内存里面的数据通过sys\_write写入到文件系统中，后面就可以直接从文件系统中加载执行了。

## 5.2 程序的编写、编译与写入磁盘

第一步，先从/bin目录下面复制一个文件目录，在这里，我们复制一下brainfuck目录，并把它命名为test，如下：



第二步，打开test，把main.c清理一下，保留基本框架。

|  |
| --- |
| #include <stdio.h>  int main(int argc,char \*argv[])  {    return 0;  } |

然后把example.txt删除掉，我们不用它。Version.txt可以保留，他是用来记录该程序的版本的。

第三部，再打开makefile，和其它makefile很相似，也是先定义工具和路径变量以及其他信息。

|  |
| --- |
| #The file path  INCLUDE\_DIR = ../../src/include/  LIB\_DIR = ../../src/lib/ |

这里的头文件目录需要引用到src/include。

我们生成的是一个二进制文件所以有BIN = bin。然后就是文件写入哪个扇区和写入多少个扇区数，写入的扇区数可以根据程序的大小调整，如果程序是5.26kb，那么你可以把扇区的大小调整为12个扇区，也就是说和文件大小最为接近但是大于文件大小。再例如文件大小事22.3kb，你可以写入45个扇区。

|  |
| --- |
| BIN\_OFF = 500  BIN\_CNTS = 20 |

后面也是一些目录的定义和编译的参数传递，和其它makefile差不多。

|  |
| --- |
| #flags  ASM\_FLAGS = -I $(INCLUDE\_DIR) -f elf  C\_FLAGS = -I $(INCLUDE\_DIR) -c -fno-builtin -Wall -Wunused  LDFLAGS = -e \_start -Ttext 0x80000000 --oformat binary |

值得注意的是，我们最终把它链接成一个二进制文件。而--oformat binary

就是做这个请求的。

下面就是OBJS文件，这个是在程序源码中的.o对象文件。除此之外，还有一个LD\_OBJS,这个变量存放的是需要链接的链接库文件，如果你需要调用某个函数，你就要链接对应的链接库才行。

|  |
| --- |
| OBJS = \_start.o\  main.o  LD\_OBJS = $(LIB\_DIR)printf.o\  $(LIB\_DIR)malloc.o\  $(LIB\_DIR)free.o\  $(LIB\_DIR)exit.o\  $(LIB\_DIR)getchar.o\  $(LIB\_DIR)putchar.o\  $(LIB\_DIR)write.o\  $(LIB\_DIR)vsprintf.o\  $(LIB\_DIR)string.o\  $(LIB\_DIR)fopen.o\  $(LIB\_DIR)fclose.o\  $(LIB\_DIR)fread.o\  $(LIB\_DIR)fwrite.o\  $(LIB\_DIR)fstat.o\  $(LIB\_DIR)lseek.o\  $(LIB\_DIR)unlink.o\ |

后面的内容和之前的makefile一样，在这儿就不在阐述，就是编译，链接，写入磁盘的具体命令操作。你不需要改变它们，你需要做的就是根据你的源文件数量和调用的库函数数量来添加不同的OBJS文件和LD\_OBJS文件。

我们也把Makefile的OBJS文件也整理一下。如下：

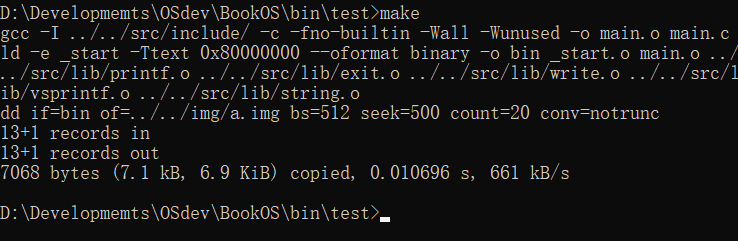
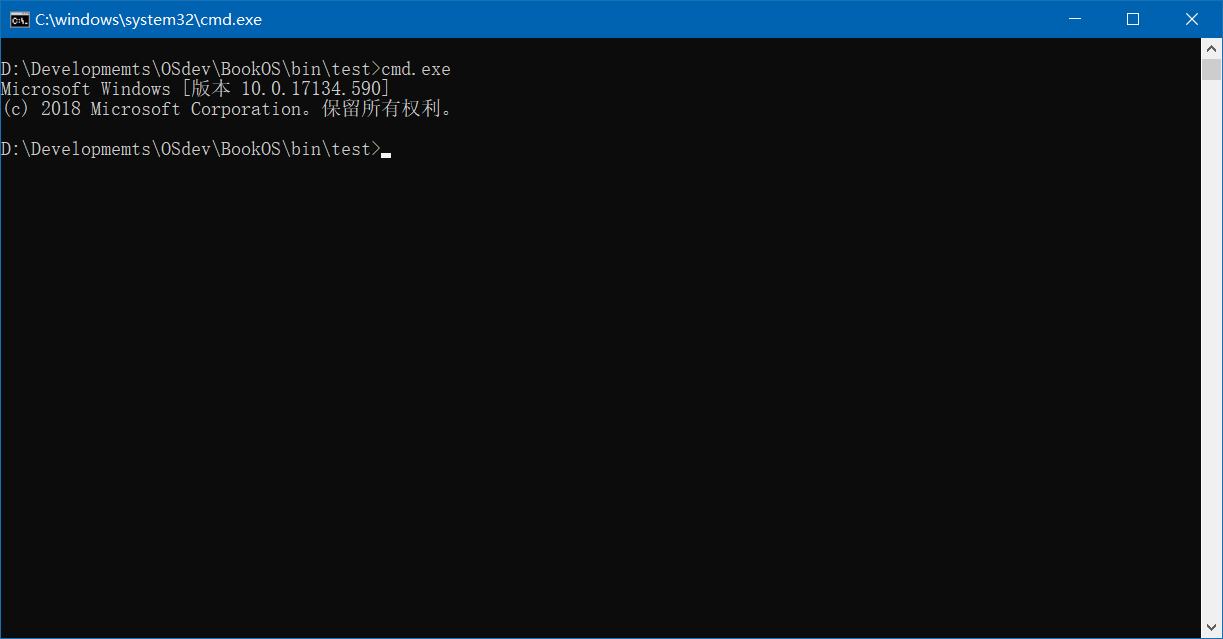
|  |
| --- |
| OBJS = \_start.o\  main.o  LD\_OBJS = $(LIB\_DIR)printf.o\  $(LIB\_DIR)exit.o\  $(LIB\_DIR)write.o\  $(LIB\_DIR)vsprintf.o\  $(LIB\_DIR)string.o\ |

我们的test程序只需要调用printf，所以只需要以上链接库文件。Write是用来往控制台写字符串，printf会调用它。printf是用来解析传入的参数，vsprintf是printf中调用的一个中间函数，vsprintf要调用string中的某些函数。那exit呢？它是程序调用exit库文件。如果代码中没有，我们也会默认调用它，所以这里要保留它。如果你想添加其它功能，那么就针对不同的功能添加不同的库文件就行了。

第四步，就是编写我们自己程序代码了。我们用printf输出hello，world！123

|  |
| --- |
| #include <stdio.h>  int main(int argc,char \*argv[])  {  printf("hello,world!\n%d", 123);  return 0;  } |

代码写好后，可以进行编译了。打开cmd.bat，输入make就可以，如果想删除生成的文件，就make clean。如下：



如果没有错误提示，就说明编译成功了，并且生成二进制文件，写入了磁盘。

## 5.3 程序的载入文件系统

现在，我们的程序已经写入磁盘了，接下来，我们就是把它载入内存。这就需要loader来做了。

第一步，在src/boot/loader.asm中找到LoadeApp这个标签，它后面做的就是加载程序到一个内存地址。

|  |
| --- |
| LoadeApp:  ;loade app  mov ax, APP\_SEG  mov si, APP\_OFF  mov cx, BLOCK\_SIZE  call LoadeBlock    add ax, 0x1000  mov cx, BLOCK\_SIZE  call LoadeBlock |

在boot/const.inc中找到APP\_SEG和APP\_OFF的定义

|  |
| --- |
| APP\_OFF EQU 500  APP\_SEG EQU 0x6000  BLOCK\_SIZE EQU 128 |

应用程序的偏移和我们之前makefile中的写入扇区偏移一样。我们把app加载到段为0x6000的地址里面（这里用ax保存参数，会在Loadeblock中给es），其物理地址是0x60000，BLOCK\_SIZE是128（因为扇区寻址原因，这儿最大只能是128），也就是每次加载128个扇区，然后call LoadeBlock，就把从磁盘上500偏移出开始的128个扇区读取到0x60000这个地址了。然后ax加上0x1000，也就是说段地址变成了0x7000，那为什么是加0x1000呢？我们来计算一下：128扇区\*512字节=65536字节=0x10000字节。也就是说读取完128扇区后，应用程序就读取了0x10000字节，下一次读取就把段移动一下，然后偏移清零，就可以继续在后面读取数据了。

在第二次读取之后，就读取了256个扇区到内存中，也就是0x20000字节=128KB，也就是说，我们要写入的文件的的大小最大为128kb。

第二步，打开src/init/main.c

你会看到一些宏定义如下：

|  |
| --- |
| #define WRITE\_DISK 1  #define WRITE\_ID 1  #if WRITE\_ID == 1  #define WRITE\_NAME "/test"  #define FILE\_SECTORS 20  #elif WRITE\_ID == 2  #define WRITE\_NAME "/boshell"  #define FILE\_SECTORS 40  #elif WRITE\_ID == 3  #define WRITE\_NAME "/tinytext"  #define FILE\_SECTORS 40  #elif WRITE\_ID == 4  #define WRITE\_NAME "/brainfuck"  #define FILE\_SECTORS 20  #endif  #define APP\_PHY\_ADDR 0x60000 |

WRITE\_DISK 为1的时候就是需要从内存读取数据进入文件系统，为0则不。

WRITE\_ID 是选择要写入哪个类型的文件。下面会根据不同的ID定义不同的WRITE\_NAME和FILE\_SECTORS。在这里我们把它设置为1，就对应了/test这个文件

APP\_PHY\_ADDR是0x60000，这和loader中把磁盘上的数据读入内存的地址一样。我们将在这个地址中读取数据来写入文件系统。

在main函数中你会看到这个：

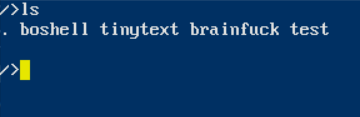
|  |
| --- |
| if(WRITE\_DISK){  write\_bin();  } |

它拿来写入文件操作，write\_bin就是执行操作的核心。

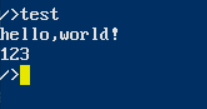
|  |
| --- |
| void write\_bin()  {  printk("write\_bin start...\n");    char \*app\_addr = (char \*)APP\_PHY\_ADDR;  printk("app:%x\n", \*app\_addr);  int fd;  int written;    fd = sys\_open(WRITE\_NAME, O\_CREAT|O\_RDWR);  written = sys\_write(fd, app\_addr, FILE\_SECTORS\*SECTOR\_SIZE);  printk("write bin %s size:%d success!\n",WRITE\_NAME, written);  sys\_close(fd);  } |

它先输出一段文字，再定义一个指针变量指向APP\_PHY\_ADDR然后打开WRITE\_NAME这个文件,如果不存在就创建，如果存在就读写。然后sys\_write把app\_addr指向的内存的数据写入FILE\_SECTORS\*SECTOR\_SIZE字节到文件。就这样，我们就把这个我们编写的程序写入到了文件系统。可以直接在文件系统中加载执行了。

然后启动系统，输入ls查看文件系统上的文件，就会看到有一个test文件



我们知道他是一个可执行文件，所以直接输入该文件就可以执行了。



你看，就输出了hello,world!123，是不是很棒？（必须的，杠杠的）

就这样，一个程序的编写，编译，链接，写入磁盘，载入内存，载入文件系统，运行就这样完成了，是不是很期待根据系统提供的库函数来开发自己的软件呢？那就赶快动手试试吧。

至此，你的屌丝人生就已经到达了巅峰！

# 5.4 如何开发C++程序

使用C++语言在某些情况下可以极大的提高开发效率，所以Bookos同样支持C++开发。不过现在的标准库支持并不是很完善。具体支持的标准库以及支持的程度可以参照/src/include中的头文件。

我个人建议使用C++的开发者参考/bin/cpptest内的程序进行开发。

因为在开发C++程序中有不少关于Makefile的坑。如果自己去试，很浪费时间，也学习不到什么。

以及对于C++标准的支持，依赖于开发者环境的gcc版本。

C++程序可以完全调用C语言程序的库。因为我所提供的C++库都是对C库的封装。

# 后记

由于本人能力有限，文档中可能还存在些许错误，希望大家能够指出，我将会及时修正。可能大家还会在开发中遇到问题，就可以联系我，我会更新解决方法到文档之中。也感谢大家对本系统的关注与支持。

编撰者：胡自成,哆啦B梦

**联系方式**

胡自成

电子邮箱：[2323168280@qq.com](mailto:2323168280@qq.com)

QQ:2323168280 微信：hu2323168280

Bookos QQ群：913813452

**文档版本发布历史记录**

**2019/2/26** 基于BookOSv0.2 发布v0.1文档