Ausar的指导书 lab1

> 之前以为自己负责的是内存那方面的，也就是lab2，于是写了一个lab2的指导书
>
> 结果发现自己是lab1的，有点囧了……

说明

本指导书重点关于于对实验作业内容的介绍，但是大指导书仍有许多有用的背景知识，希望大家两本指导书都看一看，取长补短，相互应证。

若有疏漏，请尽快与本人或者课程组反馈

> 3.9日更新：补充了printf的代码导读

Exercise 1.1

实验背景

这一个lab的任务主要是让我们的操作系统能够成功启动。先来看看我们的实验文件都有哪些：

```
┌─[ausar@b6a8b7d28e74] - [~/OS-lab] - [Tue Feb 18, 10:39]
└─[$] <git:(lab1)> ls
boot  drivers  gxemul  include  include.mk  init  lib  Makefile  readelf  tools
```

似乎文件目录有点多，各个不同的文件夹名称大致说明了他们各自的功用。但是似乎挨个文件进行浏览还是有点难度。

不过我们看见有一个文件非常熟悉，叫做`Makefile`。

我相信大家在lab0中，已经对Makefile有了初步的了解，这个Makefile文件即为构建我们整个操作系统所用的顶层Makefile。那么，我们就可以通过浏览这个文件来对整个操作系统的布局产生初步的了解：

(为了方便理解，加入了部分注释)

如果对于Makefile还不甚了解的同学，可以参考这个网站的教程：http://c.biancheng.net/view/7097.html

```
# Main makefile
#
# Copyright (C) 2007 Beihang University
# Written by Zhu Like ( zlike@cse.buaa.edu.cn )
#

drivers_dir   := drivers
boot_dir      := boot
init_dir      := init
lib_dir       := lib
tools_dir     := tools
test_dir          :=
#上面定义了各种文件夹名称
vmlinux_elf   := gxemul/vmlinux
#这个是我们最终需要生成的elf文件
link_script   := $(tools_dir)/scse0_3.lds#连接用的脚本
```

```
modules       := boot drivers init lib $(test_dir)
objects       := $(boot_dir)/start.o              \
                 $(init_dir)/main.o           \
                 $(init_dir)/init.o           \
                 $(drivers_dir)/gxconsole/console.o \
                 $(lib_dir)/*.o
#定义了需要生成的各种文件

ifneq ($(test_dir),)
objects :=$(objects) $(test_dir)/*.o
endif


.PHONY: all $(modules) clean

all: $(modules) vmlinux #我们的“最终目标”

vmlinux: $(modules)
    $(LD) -o $(vmlinux_elf) -N -T $(link_script) $(objects)
##注意，这里调用了一个叫做$(LD)的程序
$(modules):
    $(MAKE) --directory=$@
#进入各个子文件夹进行make
clean:
    for d in $(modules);    \
        do                  \
            $(MAKE) --directory=$$d clean; \
        done; \
    rm -rf *.o *~ $(vmlinux_elf)

include include.mk
```

我们再来打开`/lib/Makefile`这个文件，查看内容

```
INCLUDES := -I./ -I../ -I../include/ #这里定义了编译时候的include列表
%.o: %.c
    $(CC) $(CFLAGS) $(INCLUDES) -c $<
#注意，这里调用了一个叫做$(CC)的程序
%.o: %.S
    $(CC) $(CFLAGS) $(INCLUDES) -c $<
#上面两条均为生成.o文件的规则
.PHONY: clean

all: print.o printf.o

clean:
    rm -rf *~ *.o


include ../include.mk
```

通过浏览Makefile，我们大致可以了解到，编译我们的操作系统的时候，会进入各个子目录，分别进行编译。编译出了结果之后，再连接成我们最终的`gxemul/vmlinux`这个elf文件

但是有一点需要注意，我们的编译器是以变量的形式调用的：`$(CC)`

但是我们似乎从来没有定义过这个变量。那么这个变量定义在哪呢？仔细观察可以发现，所有的Makefile都调用了一个叫做`include.mk`的文件。让我们来看看这个文件:

```
# Common includes in Makefile
#
# Copyright (C) 2007 Beihang University
# Written By Zhu Like ( zlike@cse.buaa.edu.cn )


CROSS_COMPILE :=  bin/mips_4KC-
CC            := $(CROSS_COMPILE)gcc
CFLAGS        := -O -G 0 -mno-abicalls -fno-builtin -Wa,-xgot -Wall -fPIC
LD            := $(CROSS_COMPILE)ld
```

原来我们的`CC`和`LD`是在这里被定义的，并且用了一个比较讨巧的方法。因为我们是交叉编译，所以我们用的编译器都会有一个前缀，也就是`CROSS_COMPILE`这个变量中定义的字符串，在上面的例子中，我们的`CC`最终会被翻译为`bin/mips_4KC-gcc`，这也就是我们最终调用的编译器。

实验内容

Exercise 1.1 请修改 include.mk 文件，使交叉编译器的路径正确。之后执行 make指令，如果配置一切正确，则会在 gxemul 目录下生成 vmlinux 的内核文件。

实验提示

可以去`/opt/eldk/usr/bin/`这个目录，或者`/OSLAB/compiler/usr/bin/`目录看一眼

我们需要使用的交叉编译器是`mips_4KC-`系列

> 此前出现过实验环境与评测机环境不一致的问题，如果评测时找不到opt目录下的文件，可以尝试改用/OSLAB下的文件

Exercise 1.2

实验背景

完成了之前的任务，在实验文件夹的根目录运行`make`,应该就会在`gxemul`目录下生成`vmlinux`这个`elf`文件了

等等，elf文件？那是什么？

> ELF （文件格式）
>
> 在计算机科学中，是一种用于二进制文件、可执行文件、目标代码、共享库和核心转储格式文件。
>
> 是UNIX系统实验室（USL）作为应用程序二进制接口（Application Binary Interface，ABI）而开发和发布的，也是Linux的主要可执行文件格式。

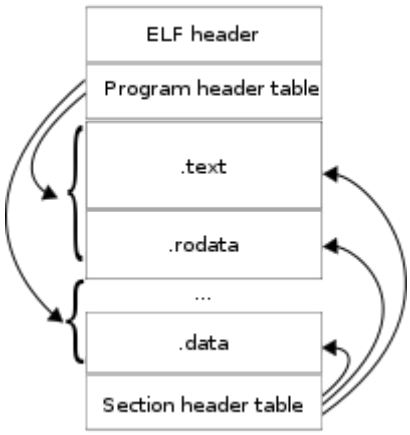
(以上摘自百度百科)

简单来说，在Linux下，elf是可执行文件的格式。在前几个lab中，我们会用elf来启动我们的系统。

而后面的lab中，我们还需要从elf中把新程序的代码给**加载**到正常运行的操作系统中。所以我们了解elf的文件内部构造是很有必要的。

由于本指导书一个比较"精简版"的说明，所以在这里我只会对ELF文件进行简要的介绍。想了解更详细的知识，同学们可以参考旧的指导书或者去网上查询相关资料。

下面是一个ELF文件的大致结构图



可以发现ELF文件被划分为了很多块区域，分别存储不同的内容。

但是问题来了，ELF文件大小不尽相同，我该如何获得ELF文件文件的各种**信息**呢?

请同学们打开`/readelf/kerelf.h`这个文件，这个文件揭示了一些ELF内部格式的细节。

(为了方便观看，我对文件中的英文进行了翻译)

```
/*文件的前面是各种变量类型定义，在此省略*/
/* The ELF file header.  This appears at the start of every ELF file.  */
/* ELF 文件的文件头。所有的ELF文件均以此为起始 */
#define EI_NIDENT (16)

typedef struct {
        unsigned char   e_ident[EI_NIDENT];     /* Magic number and other info
*/
                                                // 存放魔数以及其他信息
        Elf32_Half      e_type;                 /* Object file type */
                                                // 文件类型
        Elf32_Half      e_machine;              /* Architecture */
                                                // 机器架构
        Elf32_Word      e_version;              /* Object file version */
                                                // 文件版本
        Elf32_Addr      e_entry;                /* Entry point virtual address
*/
                                                // 入口点的虚拟地址
        Elf32_Off       e_phoff;                /* Program header table file
offset */
                                                // 程序头表所在处与此文件头的偏移
        Elf32_Off       e_shoff;                /* Section header table file
offset */
                                                // 段头表所在处与此文件头的偏移
        Elf32_Word      e_flags;                /* Processor-specific flags */
                                                // 针对处理器的标记
        Elf32_Half      e_ehsize;               /* ELF header size in bytes */
                                                // ELF文件头的大小（单位为字节）
        Elf32_Half      e_phentsize;            /* Program header table entry
size */
                                                // 程序头表入口大小
        Elf32_Half      e_phnum;                /* Program header table entry
count */
```

```
                                                // 程序头表入口数
        Elf32_Half      e_shentsize;            /* Section header table entry
size */
                                                // 段头表入口大小
        Elf32_Half      e_shnum;                /* Section header table entry
count */
                                                // 段头表入口数
        Elf32_Half      e_shstrndx;             /* Section header string table
index */
                                                // 段头字符串编号
} Elf32_Ehdr;
/*……*/
```

通过阅读代码可以发现，原来ELF的文件头，就是一个存了关于ELF文件信息的结构体。

首先，结构体中存储了ELF的**魔数**，以验证这是一个有效的ELF

当我们验证了这是个ELF文件之后，便可以通过ELF头中提供的信息，进一步地解析ELF文件了

在ELF头中，提供了段头表的入口偏移，这个是什么意思呢?

假设`*binary`为ELF的文件头，`offset`为入口偏移，那么`binary+offset`即为一个指向第一个段头的指针。

实验内容

Exercise 1.2 阅读./readelf 文件夹中 kerelf.h、readelf.c 以及 main.c 三个文件中的代码，并完成 readelf.c 中缺少的代码，readelf 函数需要输出 elf 文件的所有 sectionheader 的序号和地址信息，对每个 section header，输出格式为:`"%d:0x%x\n"`，两个标识符分别代表序号和地址。

实验提示

请也阅读`Elf32_Shdr`这个结构体的定义

关于如何取得后续的段头，可以采用代码中所提示的，先读取段头大小，随后每次累加。

也可以利用C语言中指针的性质，算出第一个段头的指针之后，当作数组进行访问。

第一部分作业可以提交

完成了之前的任务之后，现在应该可以commit之后，进行push了。完成这一阶段，能获得40分。

Exercise 1.3

实验背景

现在我们知道了，我们编译出来的操作系统内核最终会被存放在一个ELF文件中。并且在ELF文件中指定了其被加载到的内存段。

还记得我们上计组的时候，有很重要的一个步骤是在MARS中设置内存布局吗？这样我们才能把代码加载到正确的地方，以方便我们的CPU开始运行。

那么，如果想让我们的操作系统能正常启动，我们也必须要正确地设置我们的内存布局。

之前我们写readelf程序的时候，需要我们去读取section header，即段头的信息。

而这个段头，指明了连接过程中的必要信息。

在这里，我们只用重点关心这三个段：

- **.text** 保存可执行文件的操作指令。
- **.data** 保存已初始化的全局变量和静态变量。
- **.bss** 保存未初始化的全局变量和静态变量。

相信`.text`段无需我多言，各位学习计组时，写mips汇编语言的时候没少和他打交道。也就是需要运行的代码所在的段。

`.data`段和`.bss`段这两者就有点似曾相识了。他们都是存变量的，但是区别是什么呢？

假设有如下的程序：

```
char str[] = "I'm Ausar";
int empty_int[666];
int main()
{
    empty_int[0] = 0;
    //dummy program
}
```

那么，对于str这个数组而言，需要用"I''m Ausar"这个字符串来被初始化，而empty_int这个数组，没有被初始化，只是被自动填0。

因而，这两者分别归属于`.data`与`.bss`段

那么，我们怎么控制段该被加载到哪呢？

回想一下，我们最开始看到的Makefile文件中，下面的这一部分：

```
#……
link_script   := $(tools_dir)/scse0_3.lds#连接用的脚本
#……
vmlinux: $(modules)
    $(LD) -o $(vmlinux_elf) -N -T $(link_script) $(objects)
#……
```

似乎，在连接这一步，有一个奇妙的东西叫做连接器脚本（Linker Script）

这个脚本中记录了各个 section 应该如何映射到 segment，以及各个 segment 应该被加载到的位置。

接下来，我们通过 Linker Script 来尝试调整各段的位置。这里，我们选用 GNU LD 官方帮助文档上的例子 （https://www.sourceware.org/binutils/docs/ld/Simple-Example.html#Simple-Example）

```
SECTIONS
{
  . = 0x10000;
  .text : { *(.text) }
  . = 0x8000000;
  .data : { *(.data) }
  .bss : { *(.bss) }
}
```

在其中，`.`这个特殊符号，叫做定位计数器。我们可以把他类比为MIPS CPU中的PC。他会根据输出段的大小自动增长。而`*`这个通配符，会匹配所有相应的段。

在这个例子中，首先`.`将会被设到0x10000的位置，然后所有的`.text`段内容会被依次存放在0x10000开始的地方。

然后我们从0x80000000开始存放`.data`段的数据，存放结束之后，在其之后再接着存放`.bss`段的数据。

当然，这只是一个示例。我们怎么知道究竟要把我们的代码载入到什么地方呢？

请打开`include/mmu.h`这个文件，发现里面有这一张内存"地图"：

```
/*
 o     4G ----------->  +----------------------------+------------0x100000000
 o                      |       ...                  |  kseg3
 o                      +----------------------------+------------0xe000 0000
 o                      |       ...                  |  kseg2
 o                      +----------------------------+------------0xc000 0000
 o                      |   Interrupts & Exception   |  kseg1
 o                      +----------------------------+------------0xa000 0000
 o                      |      Invalid memory        |   /|\
 o                      +----------------------------+----|-------Physics Memory
Max
 o                      |       ...                  |  kseg0
 o  VPT,KSTACKTOP-----> +----------------------------+----|-------0x8040 0000---
----end
 o                      |       Kernel Stack         |    | KSTKSIZE
/|\
 o                      +----------------------------+----|------
|
 o                      |       Kernel Text          |    |
PDMAP
 o      KERNBASE -----> +----------------------------+----|-------0x8001 0000
|
 o                      |   Interrupts & Exception   |   \|/
\|/
 o      ULIM     -----> +----------------------------+------------0x8000 0000---
----
 o                      |         User VPT           |     PDMAP
/|\
 o      UVPT     -----> +----------------------------+------------0x7fc0 0000
|
 o                      |         PAGES              |     PDMAP
|
 o      UPAGES   -----> +----------------------------+------------0x7f80 0000
|
 o                      |         ENVS               |     PDMAP
|
 o  UTOP,UENVS   -----> +----------------------------+------------0x7f40 0000
|
 o  UXSTACKTOP -/       |     user exception stack   |     BY2PG
|
 o                      +----------------------------+------------0x7f3f f000
|
 o                      |       Invalid memory       |     BY2PG
|
 o      USTACKTOP ----> +----------------------------+------------0x7f3f e000
|
 o                      |     normal user stack      |     BY2PG
|
 o                      +----------------------------+------------0x7f3f d000
|
```

```
 a                     |                             |
|
 a                     ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
|
 a                     .                             .
|
 a                     .                             .
kuseg
 a                     .                             .
|
 a                     |~~~~~~~~~~~~~~~~~~~~~~~~~~~~~|
|
 a                     |                             |
|
 o       UTEXT    -----> +-----------------------------+
|
 o                     |                             |     2 * PDMAP
\|/
 a     0 ------------>  +-----------------------------+ ------------------------
----
 o
*/
```

其中，U代表的是User，即用户，K代表的是Kernel，即内核。

通过查看这个地图，同学们应该能找到应该控制段加载到哪了。

只需要让.text加载到正确位置，.data和.bss紧随其后即可

实验内容

Exercise 1.3 填写 tools/scse0_3.lds 中空缺的部分，将内核调整到正确的位置上。

实验提示

看内存布局图。

在原实验手册中，提及了大量的MIPS内存划分的知识。这一部分知识在lab2中会相当有用。虽然不需要了解也可以完成本lab，但是希望大家还是多看看。

Exercise 1.4

实验背景

如果仔细观察了上一个实验中的`tools/scse0_3.lds`这个文件，会发现在其中有`ENTRY(_start)`这一行命令

旁边的注释写着，这是为了把入口定为`_start`这个函数。那么，这个函数是啥呢

我们可以在实验的根目录运行一下下列命令进行文件内容的查找（这个命令在以后会很常用的，特别是想查找函数相互调用关系的时候）

```
grep -r _start *
```

-r的意思是进行递归查找，*的意思是查找所有文件。

```
  mips-lab git:(standard_lab1_2019)  grep -r _start *
boot/start.S:LEAF(_start) /*LEAF is defined in asm.h and LEAF functions don't call other functions*/

boot/start.S:END(_start)            /*the function defined in asm.h*/
gxemul/elfinfo:    30: 82000000    64 FUNC    GLOBAL DEFAULT    1 _start
匹配到二进制文件 gxemul/test
include/env.h:  extern u_char binary_##x##_start[];\
include/env.h:  env_create(binary_##x##_start, \
lib/printf.c:    va_start(ap, fmt);
lib/printf.c:   va_start(ap, fmt);
匹配到二进制文件 readelf/testELF
tools/scse0_3.lds:ENTRY(_start)
tools/scse0_3.lds:Set the ENTRY point of the program to _start.
```

然后发现，`boot/start.S`中似乎定义了我们所需要的函数

以下截取其中一段：

```
    /* Disable interrupts */
    mtc0    zero, CP0_STATUS

        /* Disable watch exception. */
        mtc0    zero, CP0_WATCHLO
        mtc0    zero, CP0_WATCHHI

    /* disable kernel mode cache */
    mfc0    t0, CP0_CONFIG
    and t0, ~0x7
    ori t0, 0x2
    mtc0    t0, CP0_CONFIG

/*To do:
  set up stack
  you can reference the memory layout in the include/mmu.h
*/

loop:
    j   loop
    nop
END(_start)             /*the function defined in asm.h*/
```

发现这个函数的前半段都是在初始化CPU。

后半段来了一个TODO，告诉我们需要初始化栈。

实际上我们初始化了栈之后，还需要做一些工作，否则我们会进入下方的死循环中。

还记得我们每次写C语言时必写的main函数吗？对的，我们还必须跳转到`main`函数。

实验内容

Exercise 1.4 完成 boot/start.S 中空缺的部分。设置栈指针，跳转到 main 函数。使用`gxemul -E testmips -C R3000 -M 64 elf-file` 运行 (其中 elf-file 请替换为你编译好的内核文件)。

实验提示

在旧的指导书中，`gxemul`所在路径为`OSLAB/gxemul`，在本人书写此指导书时，新版的实验环境中，找不到该目录。反之可以直接运行`gxemul`此文件。同时从旧指导书上复制该代码时，用于传递参数的半角`-`复制下来之后变为了全角的`—`，同时出现丢失空格的问题，造成许多同学运行失败。这些地方请大家注意。

在mips中，栈指针为sp寄存器。只需要把这个寄存器的值设置正确即可。可以参考`include/mmu.h`

main函数虽然为c语言所书写，但是在被编译成汇编之后，其入口点会被翻译为一个标签，类似于：

```
main:
xxxxxx
```

想想看汇编程序中，如何调用函数?

Exercise 1.5

实验背景

咱们的内核似乎快弄完了，是不是想摩拳擦掌打印一个"Hello world"了呢?

慢着，咱们这个系统不能使用标准库中的printf函数。所以我们得自己手搓。

还好，我们已经帮你实现了大部分的代码，你只用完成核心部分的函数即可。

与printf有关的文件有这几个：

- `drivers/gxconsole/console.c`
  - 这个文件负责往gxemul的控制台输出字符，其原理为读写某一个特殊的内存地址
- `lib/printf.c`
  - 此文件中，实现了`printf`,但其所做的，实际上是把输出字符的函数，接受的输出参数给传递给了`lp_Print`这个函数
- `lib/print.c`
  - 此文件中，实现了`lp_Print`函数，这个函数是Printf函数的真正内核。

> 请你阅读这三个文件，弄明白当你执行一条Printf的时候，操作系统到底都完成了什么。

为了方便大家理解，我对其中的几个关键部分进行讲解：

其中，lib/printf.c中定义了我们的printf函数。但是，仔细观察就可以发现，这个printf函数实际上基本什么事情都没有做。他只是把接受到的参数，以及myoutput()函数指针给传入ld_Print这个函数中。

```
void printf(char *fmt, ...)
{
    va_list ap;
    va_start(ap, fmt);
    lp_Print(myoutput, 0, fmt, ap);
    va_end(ap);
}
```

同学们可能对`va_list`,`va_start`,`va_end`这三个语句比较陌生。实际上，这是用于获取不定个数参数的宏。感兴趣的同学们可以参考这篇博客：https://www.cnblogs.com/qiwu1314/p/9844039.html

然后我们来看看`myoutput`这个函数，这个函数实际上是用来输出一个字符串的：

```
static void myoutput(void *arg, char *s, int l)
{
    int i;

    // special termination call
    if ((l==1) && (s[0] == '\0')) return;

    for (i=0; i< l; i++) {
    printcharc(s[i]);
    if (s[i] == '\n') printcharc('\n');
    }
}
```

可以发现，他实际上的核心是调用了一个叫做`printcharc`的函数。这个函数在哪呢?

我们可以在`./drivers/gxconsole/gxconsole.c`下面找到:

```
void printcharc(char ch)
{
    *((volatile unsigned char *) PUTCHAR_ADDRESS) = ch;
}
```

原来，想让控制台输出一个字符，实际上是对某一个内存地址写了一串数据。

看起来输出字符的函数一切正常，那为什么我们还不能使用printf呢?

还记得之前说了，printf函数实际上只是把相关信息传入到了`ld_Print`函数里面吗？而这个函数现在有一部分代码缺失了，需要你来帮忙补全。

为了方便大家理解这个比较复杂的函数，我来给大家简单介绍一下。

首先，这个函数先用宏定义来简化了`myoutput`这个函数指针的使用

```
#define     OUTPUT(arg, s, l)  \
  { if (((l) < 0) || ((l) > LP_MAX_BUF)) { \
       (*output)(arg, (char*)theFatalMsg, sizeof(theFatalMsg)-1); for(;;); \
    } else { \
      (*output)(arg, s, l); \
    } \
  }
```

然后定义了一些需要使用到的变量。有几个变量我们需要重点了解其含义:

```
    int longFlag;//标记是否为long型
    int negFlag;//标记是否为负数
    int width;//标记输出宽度
    int prec;//标记小数精度
    int ladjust;//标记是否左对齐
    char padc;//填充多余位置所用的字符
```

整个的函数主体是一个无限循环。只有当读到字符串末尾时才会退出

```
    for(;;)
    {
        //第一部分，找到%，并读取控制字符

        //第二部分，根据控制字符进行输出
    }
```

那么，现在就请你通过大指导书中提供的printf格式说明，以及自己阅读代码之后的理解，把printf函数给补全吧！

实验内容

Exercise 1.5 阅读相关代码和下面对于函数规格的说明，补全`lib/print.c`中`lp_Print()`函数中缺失的部分来实现字符输出。

实验提示

这个函数非常重要，希望大家即使评测得到满分，也要多加测试。否则可能出现一些诡异的情况下函数出错，造成后续lab评测结果显示错误而导致无法过关。

对两个初次看比较费解的变量作用予以提示：

- ladjust:标记是否为左对齐
- padc:填充位宽所用字符

具体printf的格式字符串规定，可以参考旧指导书，上面有比较详细的解答。或者可以自行上网搜索。

实验正确结果

如果你成功正确地完成了上面的所有实验，可以用如下命令进行运行：

`gxemul -E testmips -C R3000 -M 64 elf-file`

（elf-file需要修改为你编译出来的内核文件所在路径）

（（同时，为了方便起见，可以把这个命令写入makefile中，使得编译完成之后直接运行））

正确运行结果如下：

```
GXemul 0.4.6 Copyright (C) 2003-2007 Anders Gavare
Read the source code and/or documentation for other Copyright messages.

Simple setup...
    net: simulating 10.0.0.0/8 (max outgoing: TCP=100, UDP=100)
        simulated gateway: 10.0.0.254 (60:50:40:30:20:10)
            using nameserver 202.112.128.51
    machine "default":
        memory: 64 MB
        cpu0: R3000 (I+D = 4+4 KB)
        machine: MIPS test machine
        loading gxemul/vmlinux
        starting cpu0 at 0x80010000
 --------------------------------------------------------------------------------

 main.c: main is start ...

 init.c: mips_init() is called
```

```
panic at init.c:24: ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
```

如何退出gxemul

1. 按`Ctrl+C`,以中断模拟
2. 输入`quit`以退出模拟器

别把模拟器挂后台就走人了，很占内存的