实验七：实现五态进程模型及多线程应用

19335174施天予

一、实验题目

实现五态进程模型及多线程应用

二、实验目的

1、理解5状态的进程模型

2、掌握操作系统内核线程模型设计与实现方法

3、掌握实现5状态的进程模型方法

4、实现C库封装多线程服务的相关系统调用。

三、实验要求

1、学习内核级线程模型理论，设计总体实现方案

2、理解类unix的内核线程做法，明确全局数据、代码、局部变量的映像内容哪些共享。

2、扩展实验6的的内核程序，增加阻塞进程状态和阻塞过程、唤醒过程两个进程控制过程。

3、修改内核，提供创建线程、撤销线程和等待线程结束，实现你的线程方案。

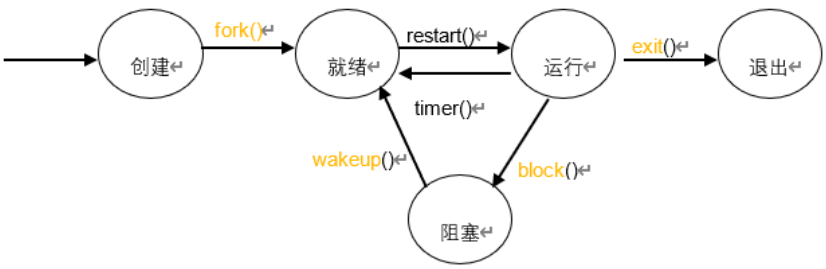
4、增加创建线程、撤销线程和等待线程结束等系统调用。修改扩展C库，封装创建线程、撤销线程和等待线程结束等系统调用操作。

5、设计一个多线程应用的用户程序，展示你的多线程模型的应用效果。

6、编写实验报告，描述实验工作的过程和必要的细节，如截屏或录屏，以证实实验工作的真实性

四、实验内容

1、修改内核代码，增加阻塞队列，实现5状态进程模型



2，如果采用线程控制块，就要分离进程控制块的线程相关项目，组成线程控制块，重构进程表数据结构。

3、修改内核代码，增加阻塞block()、唤醒wakeup()、睡眠sleep()、创建线程do_fork()、撤销线程do_exit()和等待线程结束do_wait()等过程，实现你的线程控制。

```
int do_fork() {
    PCB *p;
```

```
    p = PcbList[0];
    while (p <= PcbList[NrPCB] && p->PCBstate) p = p + 1;
    if (p > PcbList[NrPCB]) PcbList[CurrentPcbNo].ax=-1;
    memcopy(PcbList[CurrentPcbNo],p,SizeOfPCBtype);
    p->pcbID = p - pcblist[0];
    p->pcbFID = CurrentPcbNo;
    p->ss = stacksect[p->pcbID];
    memcopy(StackSect[CurrentPcbNo], StackSect[p->pcbID], SizeOfStack);
    p->ax = 0;
    PcbList[CurrentPcbNo]->ax = p->pcbID;
}
void do_wait() {
    PcbList[CurrentPcbNo]->state = BLOCKED;
    Schedule();
}
void do_exit(char ch) {
    PcbList[CurrentPcbNo]->state = END;
    PcbList[CurrentPcbNo]->used = FREE;
    PcbList[PcbList[CurrentPcbNo]->pcbFID]->state = READY;
    PcbList[PcbList[CurrentPcbNo]->pcbPID]->ax = ch;
    Schedule();
}
Void block(BlockQ *p){
    PcpList[CurrentPCBno].states = BLOCKed;
    PcpList[CurrentPCBno].next= p->next
    p->next = PcpList[CurrentPCBno]
    Schedule();
}
Void wakeup(BlockQ *p ){
    PcpList[CurrentPCBno]. states = READY;
    p->next->PcpList[CurrentPCBno].next
}
```

4、修改扩展C库，封装创建线程**fork()**、撤销线程**exit(0)**、睡眠**sleep()**和等待线程结束**wait()**等系统调用操作。

5、设计一个多线程应用的用户程序，展示你的多线程模型的应用效果。示范：进程创建2个线程，分别统计全局数组中的字母和数字的出现次数。你也可以另选其他多进程应用。

```
char str[80] = "129djwqhdsajd128dw9i39ie93i8494urjoiew98kdkd";
int LetterNr = 0;
Void main() {
    int pid;
    int ch;
    (AB)……..;
    pid = fork();
    if (pid == -1) {
        printf(“error in fork!”);
        exit(-1);
    }
    if (pid)  {
        ch = wait();
        printf(“LetterNr=%d”, LetterNr);
        exit(0);
    }
    else {
        LetterNr =CountLetter(str);
```

```
        exit(0);
    }
}
```

五、实验方案

实验环境

1、实验运行环境：Windows10

2、虚拟机软件：VirtualBox和DOSBox

3、NASM编译器

4、TCC+TASM+TLINK混合编译

实验工具

1、编程语言：NASM，TASM，C

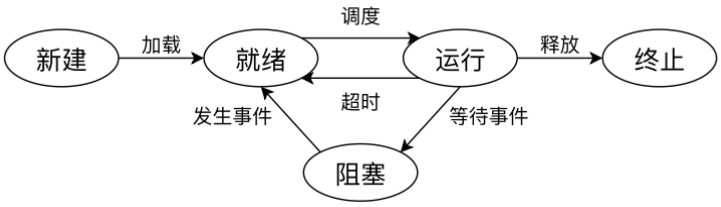
2、文本编辑器：Notepad++

3、软盘编辑软件：WinHex

实验思想

五状态进程模型

前一个原型中，操作系统可以用固定数量的进程解决多道程序技术。但是，这些进程模型还比较简单，进程之间互不往来，没有直接的合作。在实际的应用开发中，如果可以建立一组合作进程，并发运行，那么软件工程意义上更有优势。在这个项目中，我们完善进程模型：进程能够按需要产生子进程，一组进程分工合作，并发运行，各自完成一定的工作。合作进程在并发时，需要协调一些事件的时序，实现简单的同步，确保进程并发的情况正确完成使命。



进程控制的基本操作

在这个项目中，我们完善进程模型：

1、扩展PCB结构，增加必要的数据项

2、进程创建do_fork()原语，在c语言中用fork()调用

3、进程终止do_exit()原语，在c语言中用exit(int exit_value)调用

4、进程等待子进程结束do_wait()原语，在c语言中用wait(&exit_value)调用

5、进程唤醒wakeup原语（内核过程）

6、进程唤醒blocked原语（内核过程）

do_fork()的功能描述

原理中讲到，进程创建主要工作是完成一个进程映像的构造。参考unix早期的fork()做法，我们实现的进程创建功能中，父子进程共享代码段和全局数据段。子进程的执行点(CS:IP)从父进程中继承过来，复制而得。创建成功后，父子进程以后执行轨迹取决于各自的条件和机遇，即程序代码、内核的调度过程和同步要求。系统调用的返值如何获得：C语言用ax传递，所以放在进程PCB的ax寄存器中。fork()调用功能如下：

1、寻找一个自由的PCB块，如果没有，创建失败，调用返回 - 1。

2、以调用fork()的当前进程为父进程，复制父进程的PCB内容到自由PCB中。产生一个唯一的ID作为子进程的ID，存入至PCB的相应项中。

3、为子进程分配新栈区，从父进程的栈区中复制整个栈的内容到子进程的栈区中。

4、调整子进程的栈段和栈指针，子进程的父亲指针指向父进程。Pcb_list[s].fPCB= pcb_list[CurrentPCBno]。

5、在父进程的调用返回ax中送子进程的ID，子进程调用返回ax送0。

do_exit()的功能描述

父进程如果想等待子进程结束后再处理子进程的后事，需要一个系统调用实现同步。我们模仿UNIX的做法，设置wait()实现这一功能。相应地，内核的进程应该增加一种阻塞状态，。当进程调用wait()系统调用时，内核将当前进程阻塞，并调用进程调度过程挑选另一个就绪进程接权。

do_wait()的功能描述

父进程如果想等待子进程结束后再处理子进程的后事，需要一个系统调用wait()，进程被阻塞。而子进程终止时，调用exit()，向父进程报告这一事件，可以传递一个字节的信息给父进程，并解除父进程的阻塞，并调用进程调度过程挑选另一个就绪进程接权。

模块结构

磁头号	扇区号	扇区数（大小）	内容
0	1	1（512 B）	引导程序
0	2	1（512 B）	用户程序1
0	3	1（512 B）	用户程序2
0	4	1（512 B）	用户程序3
0	5	1（512 B）	用户程序4
0	6~7	2（897 B）	混合编译用户程序：输入输出
0	8~9	2（959 B）	多线程用户程序：统计字符个数
1	1~9	9（4119 B）	操作系统内核

命令功能

命令	功能
help	显示支持的命令及其功能
cls	清屏
run-	并发运行任意数量的用户程序，如“run-1234"
int22h	显示“INT22H”
test	运行混合编译用户程序（输入输出）
count	运行多线程用户程序（统计字符个数）
ls	显示文件列表
reboot	重启
poweroff	退出操作系统并关机

六、实验过程

五状态数据结构的设计

根据实验6的进程控制块做了相应的改进。增加了五状态进程模型NEW、RUNNING、READY、BLOCKED、END。并且将程序的栈段分离出来，放在内核的StackList中，便于实现子进程等功能。

```
typedef struct {
    int ax;
    int bx;
    int cx;
    int dx;
    int di;
    int bp;
    int es;
    int ds;
    int si;
    int ss;
    int sp;
    int ip;
    int cs;
    int flags;
}cpuRegisters;

int NEW = 0;
int RUNNING = 1;
int READY = 2;
int BLOCKED = 3;
int END = 4;

typedef struct stru{
    cpuRegisters cpu;
    int pid;
    int fid;/*父进程id*/
    char name[10];
    int state;/*五种状态*/
    struct stru *next;/*阻塞队列*/
```

```
}PCB;

PCB pcbList[10];
PCB *curPCB = &pcbList[9];/*指向当前运行的PCB块*/
PCB *kerPCB = &pcbList[9];
char StackList[9][256];/*程序的栈段*/
int StackSize = 0x100;
int NumOfProcess = 0;
int MaxNumOfProcess = 10;
```

线程控制过程

do_fork()

do_fork()的主要操作如下：

1. 先在PCB表查找空闲的表项，若没有找到，则派生失败。
2. 若找到空闲的PCB，则将父进程的PCB表复制给子进程，并将父进程的堆栈内容复制给子进程的堆栈，然后为子进程分配一个 ID；子进程的fid为当前进程的pid。
3. 父子进程中对do_fork函数的返回值分别设置，根据fid和pid确定父进程和子进程。
4. ax寄存器是系统调用后的函数返回值，0代表子进程，1代表父进程，-1代表派生失败。

```
void do_fork() {
    int i = (curPCB->pid + 1) % MaxNumOfProcess;
    while (i != curPCB->pid) {
        if (pcbList[i].state == NEW) {
            memcpy(&pcbList[curPCB->pid], &pcbList[i], sizeof(PCB));/*save父进程
*/
            memcpy(StackList[curPCB->pid], StackList[i], StackSize);/*子进程copy父
进程*/
            pcbList[i].state = READY;
            pcbList[i].fid = curPCB->pid;
            pcbList[i].pid = i;
            pcbList[i].cpu.sp += ((i - curPCB->pid) * StackSize);
            pcbList[i].cpu.ax = 0;/*子进程*/
            curPCB->cpu.ax = 1;/*父进程*/
            NumOfProcess++;
            break;
        }
        i = (i + 1) % MaxNumOfProcess;
    }
    if (i == curPCB->pid)
        curPCB->cpu.ax = -1;/*创建失败*/
}
```

do_exit()

子进程终止时，调用 exit，向父进程报告这一事件，从而解除父进程的阻塞。然后调用进程调度过程挑选另一个就绪进程接权。

do_exit()将子进程的状态设置为退出态END，并通过子进程 PCB 中保存的fid找到父进程的 PCB，并将父进程由阻塞态恢复为就绪态READY，然后减少总进程数，等待父进程运行。

```
void do_exit() {
    pcbList[curPCB->fid].state = READY;
    curPCB->state = END;
    NumOfProcess--;
    setIF();
    while (1);
}
```

do_wait()

当进程调用 wait 时，内核将该进程阻塞，然后通过调度挑选另一个就绪进程接权执行。

当前进程状态改为阻塞态BLOCKED，然后调度进程，使子进程运行。

```
void do_wait() {
    curPCB->state = BLOCKED;
    schedule();
}
```

block()

将当前进程设为阻塞态BLOCKED。在阻塞队列中，当前进程的下一个进程设为p进程的下一个进程，把p进程放到当前进程之前，这样就可以达到阻塞p进程的效果。

```
void block(int p) {
    curPCB->state = BLOCKED;
    curPCB->next = pcbList[p].next;
    pcbList[p].next = curPCB;
    schedule();
}
```

wakeup()

将当前进程设为就绪态READY，唤醒p进程。

```
void wakeup(int p) {
    curPCB->state = READY;
    pcbList[p].next = curPCB->next;
}
```

sleep()

当前进程状态改为阻塞态BLOCKED，然后调度进程。（因为没有具体用到，暂时设计为与do_wait相同）

```
void sleep() {
    curPCB->state = BLOCKED;
    schedule();
}
```

封装线程系统调用

将fork()作为int 21h的5号功能，exit()作为int 21h的6号功能，wait()作为int 21h的7号功能。值得注意的是，在中断处理程序中，fork0和wait0都要用save和restart保护和恢复现场，而exit0最后的结果是返回所以要用jmp near ptr _do_exit指令。

```
public _fork
_fork proc
    mov ah,5h
    int 21h
    ret
_fork endp

public _exit
_exit proc
    mov ah,6h
    int 21h
    ret
_exit endp

public _wait
_wait proc
    mov ah,7h
    int 21h
    ret
_wait endp

; int21h 完成系统内核的调用
int21h:

......

next4:
    cmp ah,5h
    jnz next5
    jmp fork0
next5:
    cmp ah,6h
    jnz next6
    jmp exit0
next6:
    cmp ah,7h
    jnz quit
    jmp wait0
quit:
    iret

......

fork0:
    call _save
    call near ptr _do_fork
    jmp _restart
exit0:
    mov ax,cs
    mov ss,ax
    mov ds,ax
```

```
    mov es,ax
    mov si,word ptr [_kerPCB]
    mov sp,[si+20]
    jmp near ptr _do_exit
wait0:
    call _save
    call near ptr _do_wait
    jmp _restart
```

多线程用户程序

这个程序仿写老师的代码，在子进程统计字符串中的字母的个数，父进程输出结果。 这个用户程序我沿用了之前输入输出用户程序的clibs.c库和libs.asm库的代码，并在libs.asm中添加了fork，exit和wait的系统调用。具体代码如下：

```
public _fork
_fork proc
    mov ah,5h
    int 21h
    ret
_fork endp

public _exit
_exit proc
    mov ah,6h
    int 21h
    ret
_exit endp

public _wait
_wait proc
    mov ah,7h
    int 21h
    ret
_wait endp
```

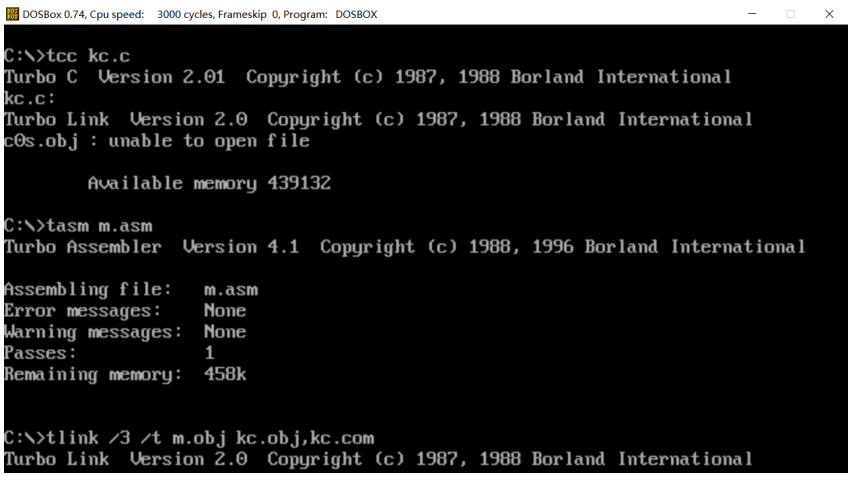
多线程用户程序我在老师的基础上做了一个创新。父进程计算字符串中字母的个数，子进程计算字符串中数字的个数，最后这两个结果都由父进程输出。具体代码如下：

```
#include "clibs.c"

extern int fork();
extern void wait();
extern void exit();

char str[80];
int LetterNr = 0;
int DigitNr = 0;

void countLetter() {
    int i;
    for (i = 0; str[i] != 0; i++)
        if ((str[i] <= 'z' && str[i] >= 'a') || (str[i] <= 'Z' && str[i] >=
'A'))
            LetterNr++;
}
```

```
void countDigit() {
    int i;
    for (i = 0; str[i] != 0; i++)
        if (str[i] <= '9' && str[i] >= '0')
            DigitNr++;
}

int main() {
    int pid;
    printf("Input a string: ");
    scanf("%s", str);
    pid = fork();
    if (pid == -1) {
        printf("Fork failed!\r\n");
        exit();
    }
    if (pid) { /*父进程*/
        countLetter();
        wait();/*等待子进程*/
        printf("LetterNr = %d\r\n", LetterNr);
        printf("DigitNr = %d\r\n", DigitNr);
    }
    else { /*子进程*/
        countDigit();
        exit();
    }
}
```

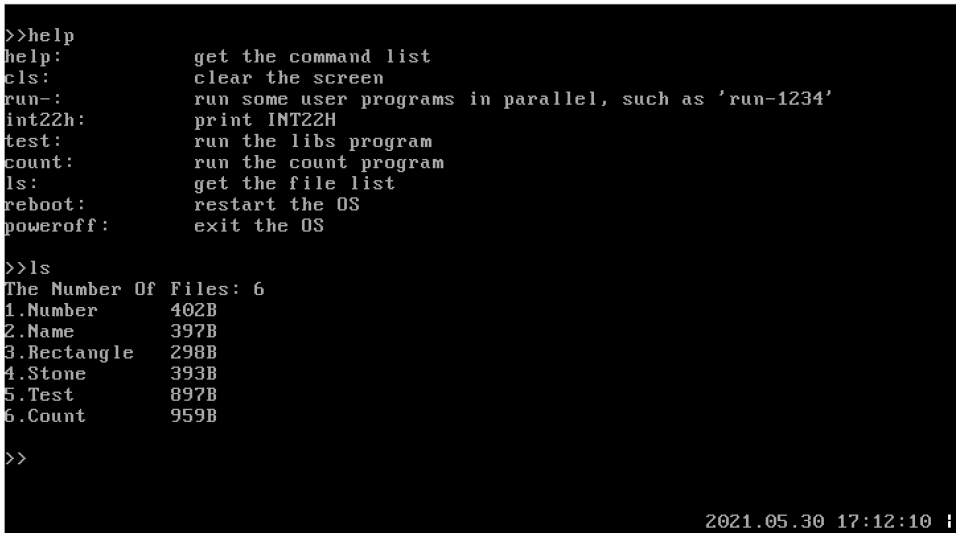
七、实验结果

内核混合编译

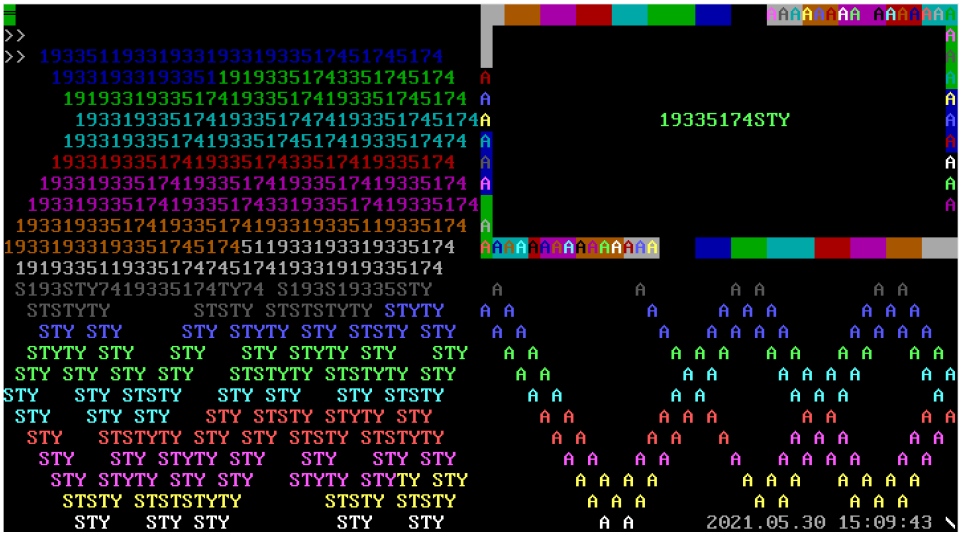


操作系统界面

帮助菜单和文件列表



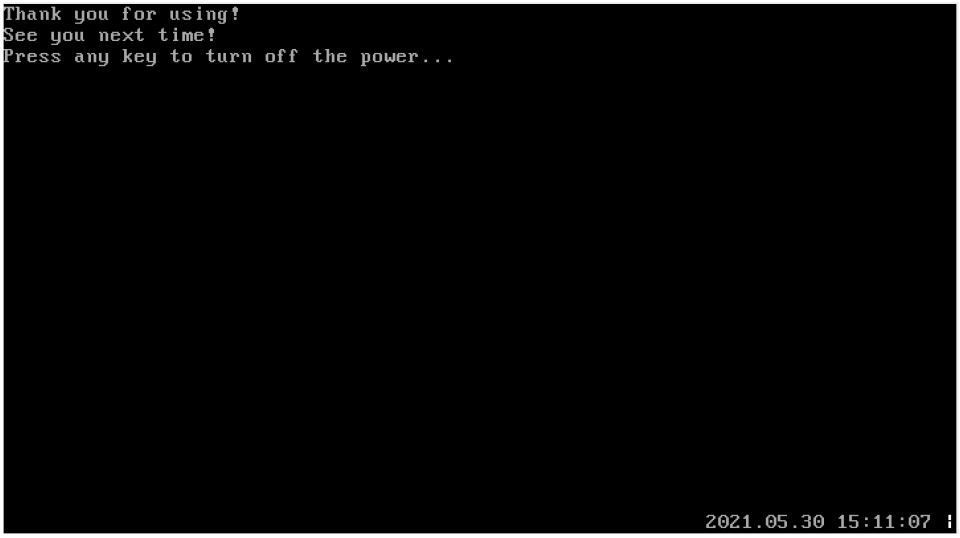
4个用户程序并行



多线程用户程序

关闭电源（int 21h的3号功能）

按下任意按键后关闭电源！



八、实验总结

本以为上次实验构建好了多进程的框架，后面的实验只要改善并增添一下C代码就可以了，没想到这次实验还是让我心力交瘁了好久，bug很多解决起来十分麻烦。

五状态进程不同于二态进程，各种进程状态间的转换复杂了许多，不过这一部分也并不是很难。一系列线程函数让我头疼了好久，特别是fork函数。fork 产生一个子进程，该子进程和父进程共享代码段，而各自有独立的堆栈段。fork 将在父进程和子进程产生不同的返回值，要想实现不同的返回值，需要将父进程和子进程的 PCB 中的 ax 修改为不同的值，非常巧妙。由于父子进程的堆栈相互独立，因此 fork 后必须把父进程的堆栈拷贝给子进程。

对于栈的处理必须格外谨慎小心。进程切换涉及到栈切换，而进程切换这一过程本身也需要用到栈，因此必须适当地安排栈的位置。一开始我的子进程根本不运行，输出字母个数时永远是0。后来虽然子进程是运行了，但我的多线程用户程序又乱码了，而且时钟中断也消失了，虽然我知道肯定是栈的问题，但要找出错误还是十分繁琐的。我发现我一开始数组开小了，这样栈的位置就不正确；然后我发现我的ss寄存器的赋值不正确，从而导致sp寄存器也不正确。就这样，我对一个个小细节仔细检查、改进，终于完成了。

这次实验还有一个要注意的点，因为我的内核占了9个扇区，我把它放到了软盘的18号扇区。而从18号扇区开始，调度时磁头号要设为1，然后扇区号也设为1，具体如下图所示：

磁道号	磁头号	扇区号		相对扇区号
0	0	扇区号	1	0
			2	1
			*	*
			18	17
	1	扇区号	1	18
			2	19
			*	*
			18	35

总而言之，这次实验虽然过程十分痛苦，但我也学到了很多知识。不知道后面还有多少个实验，希望我能再接再厉，勇往直前！

九、参考文献

《汇编语言 （第3版）》

《X86汇编语言：从实模式到保护模式》

https://wenku.baidu.com/view/9ac0446c48d7c1c708a145aa.html

https://blog.csdn.net/qq_35212671/article/details/52761585

https://blog.csdn.net/qq_41936805/article/details/103059300