# 操作系统实验日志

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 学号 | 201808010718 | 姓名 | 肖鹏 | 专业年级班级 | 智能1802 |
| 实验日期 | 2020.12.24 | 实验项目 | 第14天：高分辨率及键盘输入 | | |

## 零、快速引导

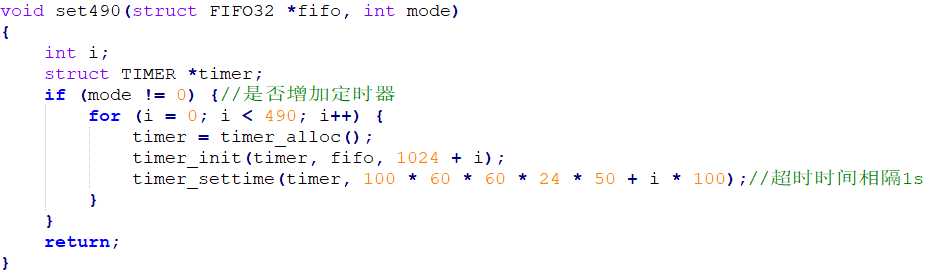
* [*内容1：定时器测试内容1：定时器测试*](#内容1：定时器测试)
* [*内容2：高分辨率*](#内容2：提高分辨率)
* [*内容3：键盘输入*](#内容3：键盘输入)
* [*内容4：追忆内容*](#内容4：追忆内容)
* [*创新点：（大设计）登陆界面*](#创新点：（大设计）登陆界面)

## 一、实验主要内容

### 1、性能测试

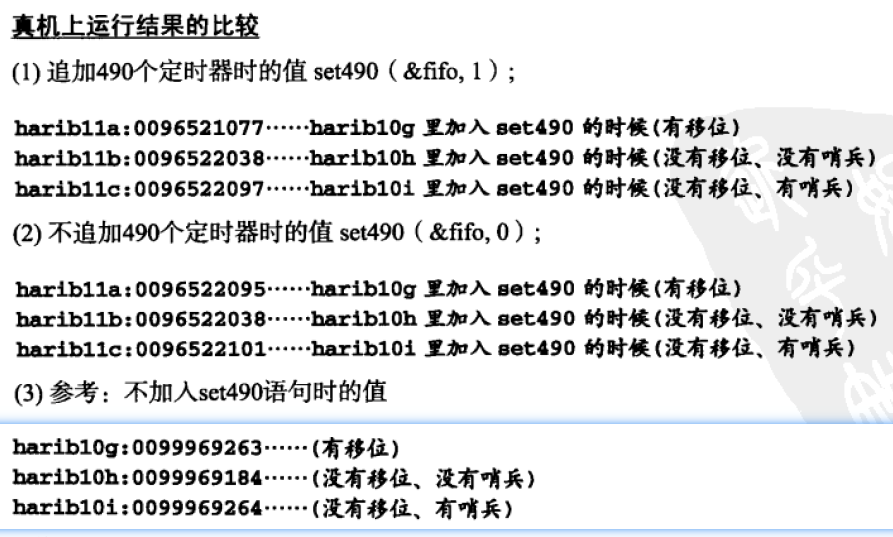
#### 内容：增加定时器测试

由于之前只有3个计时器，运行起来“移位”操作比较少，所以我们优化之后的对比不是 很明显。所以现在增加了490个定时器，且超时时间都设定在50天后，这样能够“拉满” 优化前的移位操作，增大与优化后的对比。



*（性能测试：bootpack.c中的set490( )）*

测试结果：



*（性能测试：比较结果）*

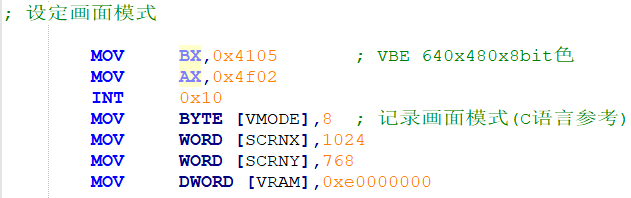
测试结果分析： ①通过（1），追加定时器的情况下，取消移位、设置哨兵都会令程序更快。 ②通过（2），定时器较少时，取消移位反而会令程序变慢，可能是因为if语句变多了， 加上哨兵后和原来移位速度差不多。 ③在有哨兵、没有移位的情况下，无论定时器的数量多少，对性能影响不大。 ④（3）的情况和②描述的现象差不多，但是（3）更快。原因在这是因为C编译器的 问题，因为跳转目标地址不同，CPU的JMP指令执行的时钟周期数也不一样。在程序 中，循环执行count++的for循环虽然最终会被编译为JMP指令，但是前面还有个set490 指令，会使 得指令的地址都相应的错开几个字节，导致了JMP指令的地址也会变化， 因此执行时间也就出现了延迟，结果变差了3%。这个问题的解决方法就是用nask来 编写HariMain，但因为这个太麻烦了，就暂时不做了。

### 2、提高分辨率

#### 内容：

##### a）在模拟器上运行

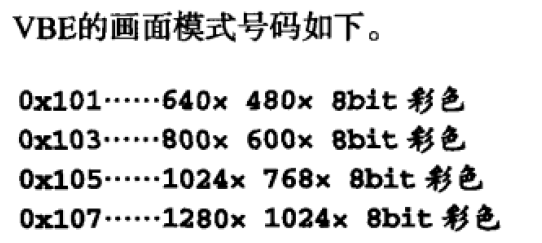
要利用高分辨率，就要考虑到显卡的因素，首先只考虑QEMU模拟器的显卡，等到这个可 以用了，再去考虑其他的显卡。在画面切换中，需要使用BIOS，因此改写asmhead.nas的 画面模式设定。



*（asmhead.nas：新画面设定）*

当给AX赋值0x4f02，给BX赋值画面模式号码时，就可以切换到高分辨率画面模式。

使用320x200的分辨率时，采用的是AH=0，AL=画面模式号码的方式，当切换到新画面时， 就可以使用AX=0x4f02。以前的方式是不使用VBE的画面模式，切换到使用VBE的画面模 式时，就要使用AX=0x4f02，BX=画面模式号码。



*（VBE其他画面模式）*

实际指定时，需要将上述的画面模式号码值加上0x4000，再复制到BX中去。

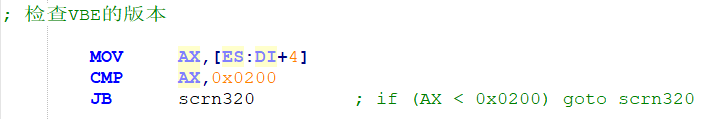
##### b）在真机上运行

之前的高分辨率只能在模拟器中运行，而要在真机上运行的话，首先需要检测真机上的 显卡能否使用VBE，如果不能的话，只能退回到原处，继续使用320x200的分辨率。



*（asmhead.nas：检查机器上是否存在VBE）*

将ES赋值为0x9000，将DI赋值为0，将AX赋值为0x4f00，再执行INT 0x10，将AX与 0x004f进行比较；如果有VBE的话，AX将会变成0x004f，如果没有变化，则表示没有VBE， 因此便无法切换画面模式。 这段程序运行的原理是，显卡能利用的VBE信息是要写入到内存中以ES:DI开始的512字 节中，赋值是为了指定写入地址。 接下来判断VBE的版本是否是2.0以上，如果不是的话，就不能使用高分辨率。



*（asmhead.nas：VBE版本检查）*

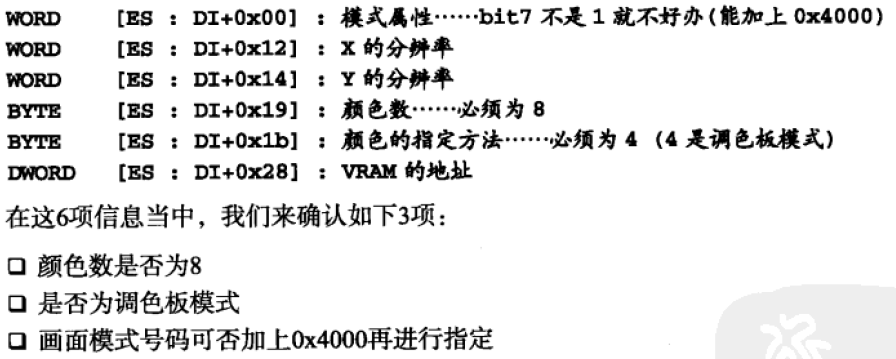
如果AX比0x0200小的话，那么只能继续使用320x200的画面模式。

然后，还需要检查画面模式0x105是否能够使用：



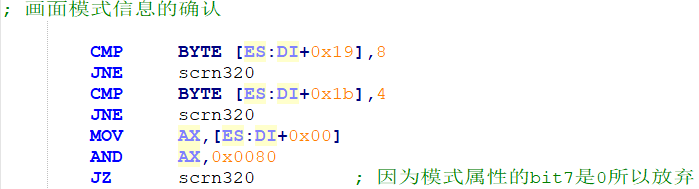
*（asmhead.nas：画面模式检查）*

如果AX是0x004f以外的值，就意味着所指定的画面模式不能使用。



*（画面模式重要信息）*

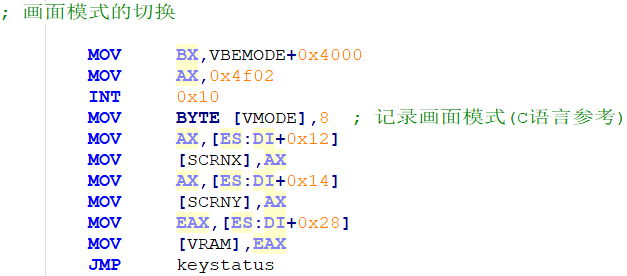
最后我们进行画面信息的确认：



*（asmhead.nas：画面模式信息的确认）*

在完成以上检查步骤之后，便可以使用所指定的VBE画面模式。

下面进行画面模式切换：



*（asmhead.nas：画面模式切换）*

这样便可进入大画面模式。

### 3、键盘输入

#### 内容：实现键盘输入

**a）找到按键码与字符的映射关系表：**

按键弹起的按键码=按键按下时的按键码+0x80

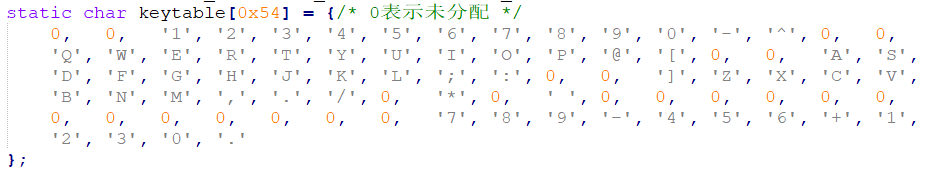


*（按键码表）*

**b）实现输出：**

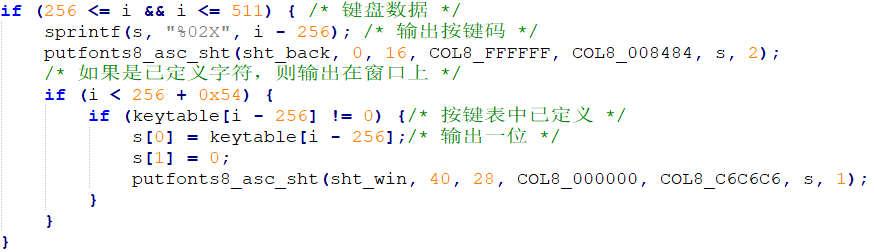
①修改main函数，删去不再使用的计数count变量；

②构建映射关系keytable数组：



*（bootpack.c：字符表）*

③根据按键码输出字符：

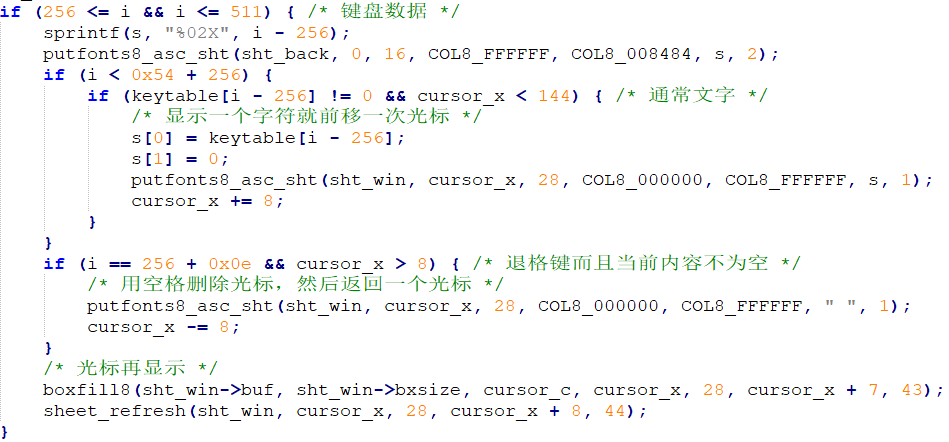


*（bootpack.c：输出按下字符）*

### 4、追忆内容

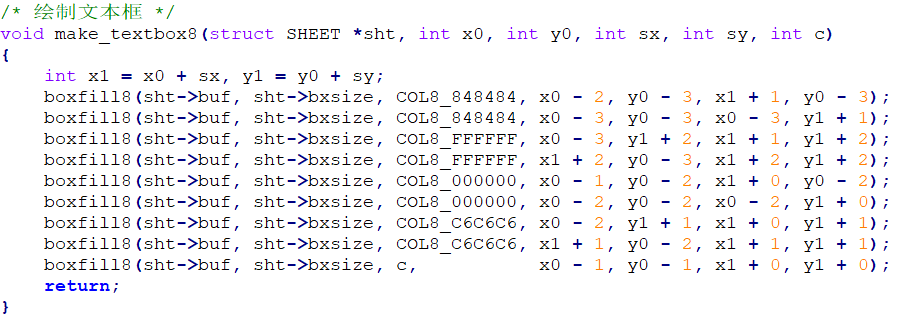
#### 内容：实现文本输入框

**①绘制文本框、实现输入：**



*（bootpack.c：实现文本框输入）*

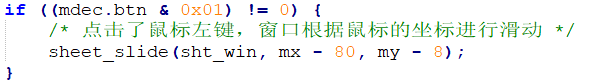
其中cursor\_x是用来记住光标显示位置的变量，当输入一个字符后，这个变量就递增8， 因为一个字符的长度就是8。cursor\_c变量则表示光标的颜色，每0.5秒变化一次。 Make\_textbox8函数是用来描绘文字输入背景的函数：



*（bootpack.c：绘制文本框）*

**②实现窗口移动：**

这部分实现窗口的移动，鼠标点击屏幕的一个位置，窗口就出现在哪里，同时为了让窗口 可以移动到屏幕以外的部分，所以窗口移动到的位置不应该是鼠标的坐标，而是mx-80， my-8。



*（bootpack.c：实现窗口移动）*

但是这个实现有个小瑕疵，就是只要鼠标点击了窗口的一个地方，即使还没有开始拖动， 此时窗口也会立即移动到鼠标的位置。

## 二、遇到的问题及解决方法

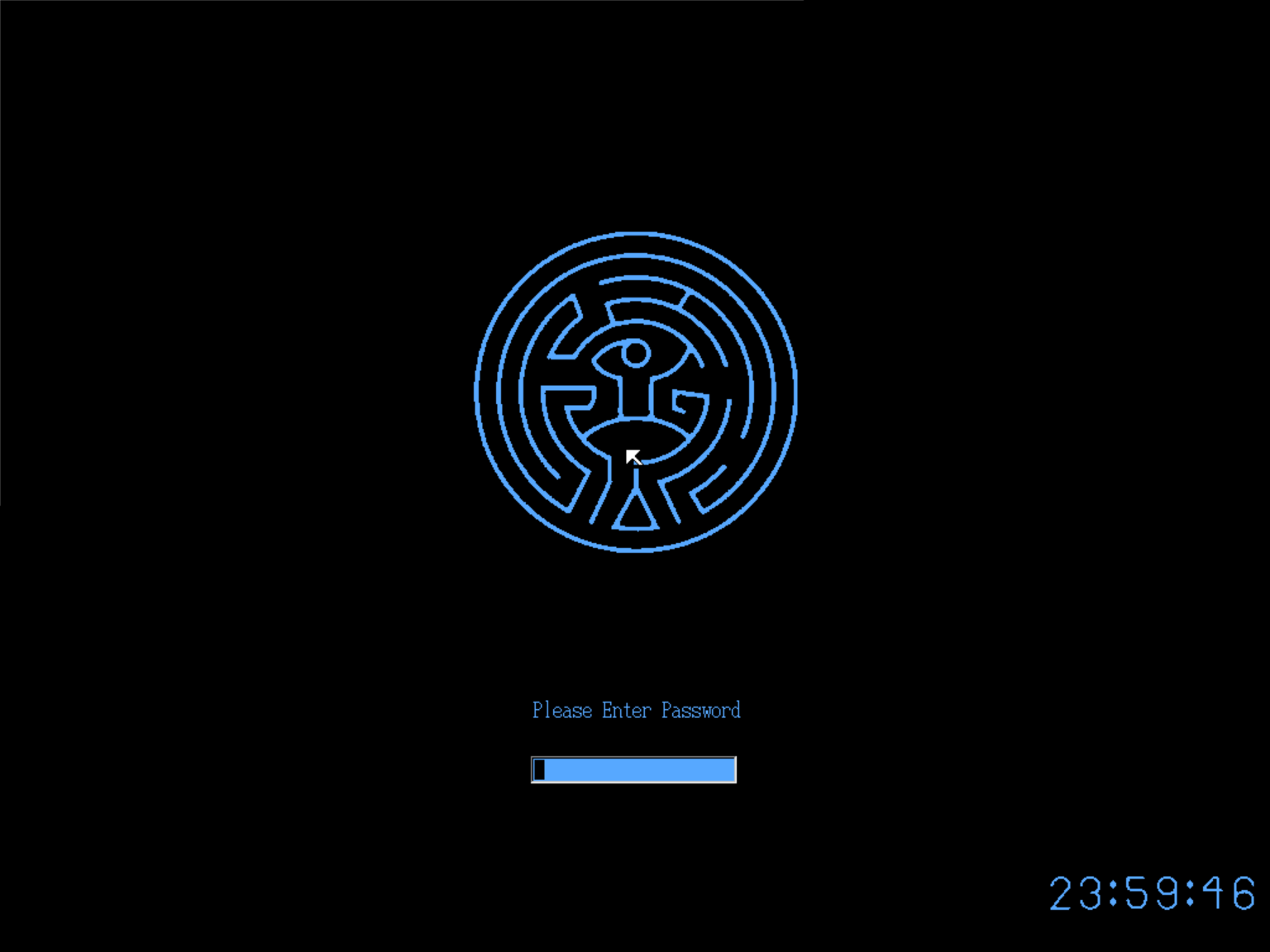
无

## 三、程序设计创新点

### 1、实现登录界面以及相关动画

**创新点：**根据今天所学知识：键盘输入，结合之前制作窗口的方法，实现一个电脑登陆界面（也就是最终大设计的）。主要功能有：播放开机启动画面、登陆界面、输入提示、错误/成功提示。

**效果截图：**



*（登陆界面截图）*



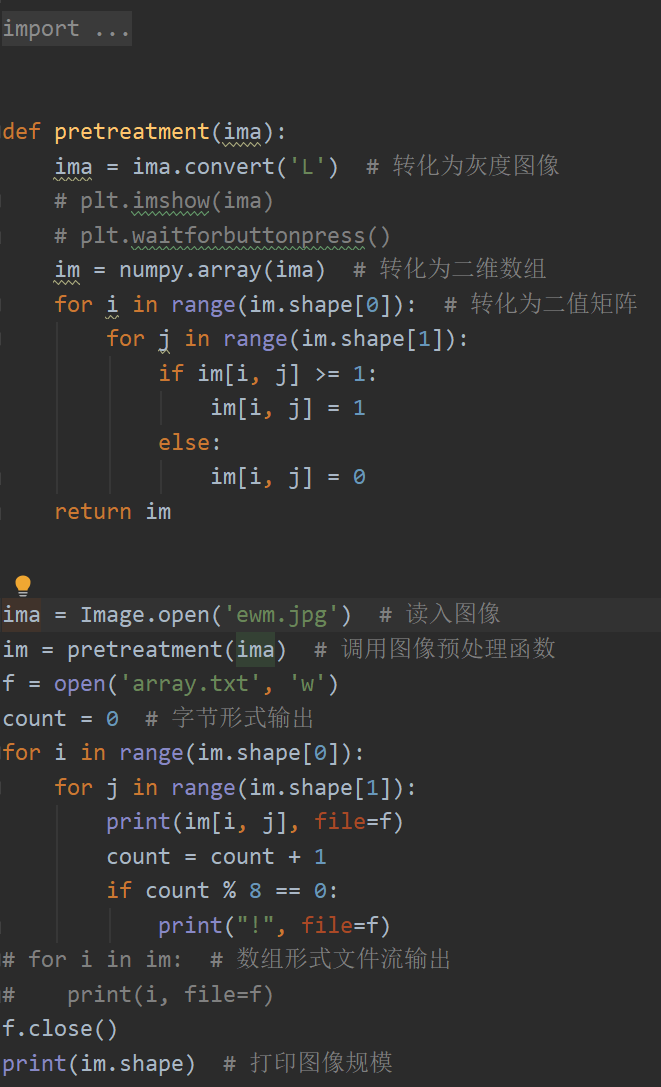




*（检测输入状态）*

**实现重点：**

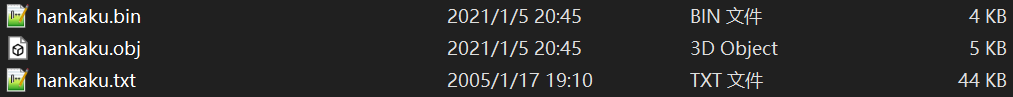
①首先要介绍一下如何绘制出中间这个迷宫图案：大家肯定觉得很炫酷很哇塞（至少 我最开始做出来的时候是这样觉得的哈哈哈哈哈）。我们之前已经做过了图标的设计，就是在一个二维数组里面，“手动绘制图案”——一个一个像素点抠出来，对于小一点的图案也许行得通，稍微大一点就会看得眼睛生疼，像这样一个接近300\*300规模的图案，肯定不能用就方法了。恰巧这个学期选修了一门图像处理的课程，我们可以通过处理现有图像，通过灰度值提取来得到一个数组。使用到的Python代码如下：



*（灰度图像处理）*

然后我们根据得到的数组，就能够用之前的方法，给不同的值分配不同颜色。

需要注意的是，如果数组太大，那就（我在实现过程中是这样的）不能直接把得到的数组声明成static chat [][]，需要使用更为“原始”的方法，还记得Day 5时候介绍的导入所有字符的这三个文件吗？



所以我们需要把得到的数组稍做转换，再写一个适用的函数即可，具体方法见大设计报告（emmm虽然现在还没写，但我应该会写到这个点）。

## 四、实验心得体会

为期末大设计做铺垫，冲冲冲！