重庆大学编译原理课程实验报告

年级、专业、班级	2020 级计算机科学与技术 05 班	姓名	杨奎
实验题目	编译器设计与实现		
实验时间	2023/4/16- 2023/6/20	实验地点	DS3304
实验成绩		实验性质	□验证性 □设计性 ■综合性

教师评价：

□算法/实验过程正确；□源程序/实验内容提交 □程序结构/实验步骤合理；

□实验结果正确； □语法、语义正确； □报告规范；

其他：

评价教师签名:

一、实验目的

以系统能力提升为目标，通过实验逐步构建一个将类 C 语言翻译至汇编的编译器，最终生成的汇编代码通过 GCC 的汇编器转化为二进制可执行文件，并在物理机或模拟器上运行。实验内容还包含编译优化部分，帮助深入理解计算机体系结构、掌握性能调优技巧，并培养系统级思维和优化能力。

二、实验项目内容

本次实验将实现一个由 SysY (精简版 C 语言，来自 https://compiler.educg.net/) 翻译至 RISC-V 汇编的编译器，生成的汇编通过 GCC 的汇编器翻译至二进制，最终运行在模拟器 qemu-riscv 上

实验至少包含四个部分: 词法和语法分析、语义分析和中间代码生成、以及目标代码生成，每个部分都依赖前一个部分的结果，逐步构建一个完整编译器

实验一：词法分析和语法分析，将读取源文件中代码并进行分析，输出一颗语法树

实验二：接受一颗语法树，进行语义分析、中间代码生成，输出中间表示 IR (Intermediate Representation)

实验三：根据 IR 翻译成为汇编

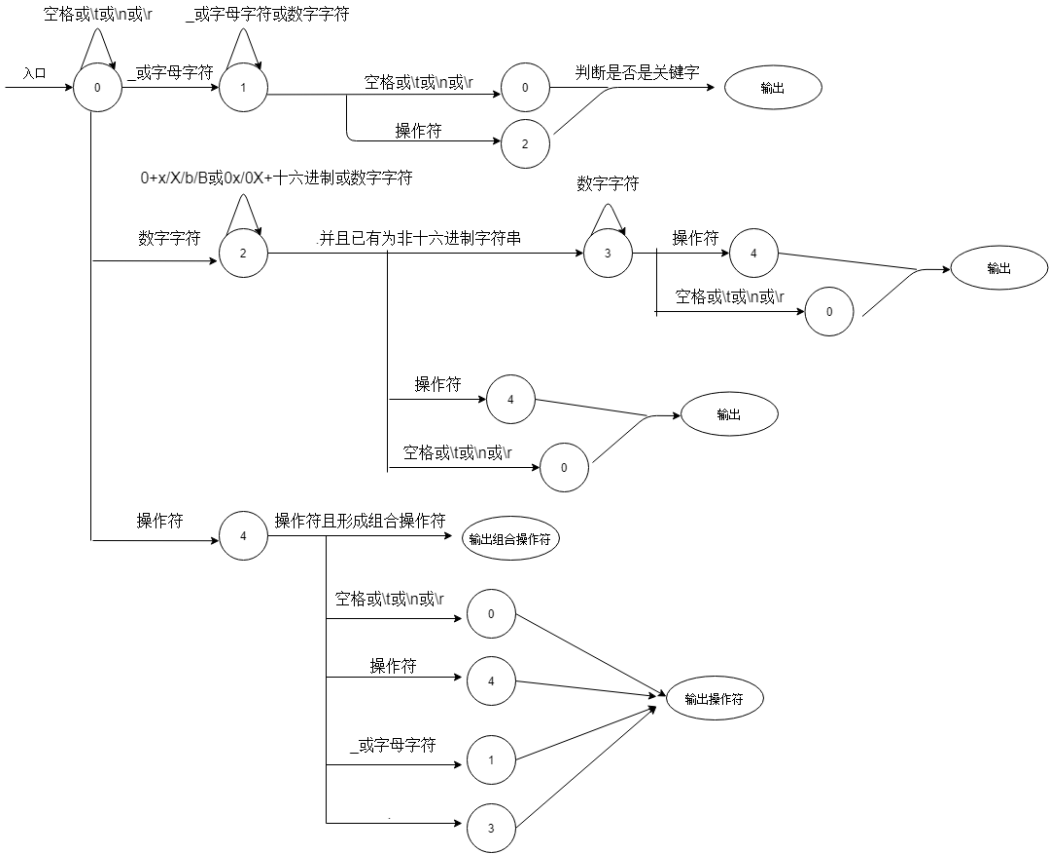
实验四(可选)：IR 和汇编层面的优化

三、实验内容实现

1. 实验实现的内容

（1）实验一实现了编译器的前端部分：词法分析+语法分析，将输入的源文件转换得到一颗语法分析树 CST（本次实验语法分析生成的应该不是抽象语法树 AST，抽象语法树会在此基础上剥离掉一些不重要的细节，进行压缩和简化）。

词法分析部分，定义了 token 类型、五个状态（Empty 空，Ident 标识符，IntLiteral 整数立即数，FloatLiteral 浮点数立即数，op 操作符），确定有限状态自动机 DFA 结构体以及扫描器 Scanner 结构体。Scanner 实现了 run()函数用于启动词法分析，还在其中实现了单行注释//和多行注释/* */的去除。DFA 的 next 函数实现了五大状态间的转移函数，同时自定义了函数：①isoperator—判断字符是否为操作符，②get_op_type—获取操作符的 token 类型，③iskeyword—判断字符串是否为关键字，④get_keyword_type—获取关键字的 token 类型，⑤ishex—判断是否是十六进制数，⑥isdec—判断是否是八进制数，⑦is_comb_op—判断下一字符是否能和当前字符组成组合运算符。上述自定义的函数使得 DFA 中 next 函数实现起来容易许多。DFA 中 next()函数的状态转移过程如下图，其中 0-4 分别对应 Empty-op 状态（写报告的时候为直观理解才画的，可能会画错）：



语法分析部分定义了不同树节点的基类 AstNode 节点，包含节点类型 type、父节点 parent、子节点向量 children 三个属性，在此基础上继承了不同的非终结符节点，使用 type 属性区分，终结符节点类型均为 TERMINAL，但增加了 token 属性，可以通过 token 区分其类别。本次实验的文法没有左递归，无需进行消除左递归的操作，用递归下降法实现 LL（1）语法分析方法。文法中的每个产生式都可以展开成树的结构，以产生式左侧非终结符为父节点，从左至右的节点作为其子节点，需要给文法的每个非终结符节点创建 parse 函数用于递归分析，当遇到非终结符时就调用其 parse 函数。如果遇到下一个分析的节点有多种情况，例如 Decl -> ConstDecl | VarDecl，要下一个要处理的 token 和可供选择非终结符的的 first 集来选择下一步处理哪一个产生式。

本次实验文法的 First 集，有了此表，实现起来非常快：

CompUnit : {'const', 'void', 'int', 'float'}
Decl : {'const', 'int', 'float'}
ConstDecl : {'const'}
BType : {'int', 'float'}

ConstDef : {'Ident'}
ConstInitVal : {'(' , 'Ident' , 'IntConst' , 'floatConst', '+' , '-' , '!' , '{' }
VarDecl : { 'int' , 'float' }
VarDef : {'Ident'}
InitVal : {'(' , 'Ident' , 'IntConst' , 'floatConst' , '+' , '-' , '!' , '{' }
FuncDef : { 'void' , 'int' , 'float'}
FuncTYpe : { 'void' , 'int' , 'float'}
FuncFParam : { 'int' , 'float' }
FfucFParams : { 'int' , 'float' }
Block : { '{' }
Blockltem : { 'const' , 'int' , 'float' , '{' , 'if'' , 'while' , 'break' , 'continue' , 'return' , '(' , 'Ident' , 'IntConst' , 'floatConst' , '+' , '-' , '!' , ';'}
Stmt : {'{' , 'if'' , 'while' , 'break' , 'continue' , 'return' , '(' , 'Ident' , 'IntConst' , 'floatConst' , '+' , '-' , '!' , ';'}
Exp : {'(' , 'Ident' , 'IntConst' , 'floatConst' , '+' , '-' , '!' }
Cond : {'(' , 'Ident' , 'IntConst' , 'floatConst' , '+' , '-' , '!' }
LVal : {'Ident'}
Number : {'IntConst' , 'floatConst'}
PrimaryExp : {'(' , 'Ident' , 'IntConst' , 'floatConst'}
UnaryEXP : {'(' , 'Ident' , 'IntConst' , 'floatConst' , '+' , '-' , '!'}
UnaryOp : {'+' , '-' , '!'}
FuncRParams : {'(' , 'Ident' , 'IntConst' , 'floatConst' , '+' , '-' , '!'}
MulEXP : {'(' , 'Ident' , 'IntConst' , 'floatConst' , '+' , '-' , '!'}
AddEXP : {'(' , 'Ident' , 'IntConst' , 'floatConst' , '+' , '-' , '!'}
RelExp : {'(' , 'Ident' , 'IntConst' , 'floatConst' , '+' , '-' , '!'}
EqExp : {'(' , 'Ident' , 'IntConst' , 'floatConst' , '+' , '-' , '!'}
LAndExp : {'(' , 'Ident' , 'IntConst' , 'floatConst' , '+' , '-' , '!'}
LOrExp : {'(' , 'Ident' , 'IntConst' , 'floatConst' , '+' , '-' , '!'}
ConstExp : {'(' , 'Ident' , 'IntConst' , 'floatConst' , '+' , '-' , '!'}

（2）实验二实现了语义分析、类型检查和语法制导翻译，翻译成形如（opcode，des，operand1，operand2）的中间表示 IR，实验一得到语法分析树后，要进行语义分析。整个实验三完成的事情很多，将语义分析、类型检查和语法制导翻译三件事在一个实验中进行，导致处理起来较为麻烦，既要考虑属性的传递，又要进行类型检查与翻译。

使用栈式结构来存储作用域，每个作用域都有创建与维护符号表并且提供了函数用于向符号表中查询与插入，其中作用域有 cnt 和 name 属性，cnt 用于标识唯一作用域，将变量 a 重命名为 a_0 可以解决变量重名的问题，name 属性用于作用域的类别，重命名时可以转换为:a_0_i，表示第一个作用域（if）的 a 变量，只是为了 debug 时直观清晰，并无其他含义。在本次实验中，我舍弃了 name 属性，同时将 cnt 作为标识整段代码中作用域的唯一标识而不是函数中的唯一标识，以 a_0 的方式重命名变量，但是并不重命名函数形参。

与表达式相关的节点要考虑常数的合并优化，例如 a=3*4 直接优化为 a=12，其中 const 类型的变量也参与常数优化，例如 a=b*4，b 是 const 常量 3，优化为 a=12，代码中与此相关的属性为 computable 属性，通过该属性的自底向上传播，可以减少 IR 的生成。自底向上传播时，例如 AddExp -> MulExp，只有一个节点，AddExp.computable=MulExp.computable，如果是 AddExp -> MulExp * MulExp，当两个 MulExp 节点都可简化时，AddExp=true，否则 AddExp=false。同时，当该节点 computable 属性为 true 时，该节点的 v 属性置为值，否则 v 属性置为变量名。向上传递时，如果父节点 computable=true，就可以将计算结果置于 v 中。

本次实验同时进行了大量的类型检查与类型转换，我将类型转换进行了很好的函数封装，命名为 type_transform，需要的地方调用即可，使得代码量大幅减少。基本逻辑如下，涵盖了所有可能的情况，将两变量类型转换为一致。

```
if (a.type == Type::Int){
    if (b.type == Type::Float){     // Int-Float
        auto tmp_op = Operand("t" + std::to_string(tmp_cnt++), Type::Float);    // a转Float
        buffer.push_back(new Instruction(a, {}, tmp_op, Operator::cvt_i2f));
        a = tmp_op;
    }else if (b.type == Type::FloatLiteral){    // Int-FloatLiteral
        auto tmp_op1 = Operand("t" + std::to_string(tmp_cnt++), Type::Float);    // a转Float
        buffer.push_back(new Instruction(a, {}, tmp_op1, Operator::cvt_i2f));

        auto tmp_op2 = Operand("t" + std::to_string(tmp_cnt++), Type::Float);    // b转Float
        buffer.push_back(new Instruction(b, {}, tmp_op2, Operator::fdef));

        a = tmp_op1;
        b = tmp_op2;
```

经过观察我发现本次实验的测试用例有关浮点数的类型检查与转换只在最后一个点集中出现，且从表达式到返回值等处要求很多，故而没有完全实现浮点数相关内容。

根据 IR 的定义，分析语法树的同时我们将语义动作翻译成 IR。

（3）实验三将实验二生成的中间表示 IR 转换为 RISC-V 汇编，需要大量查看 RSIC-V 中文手册，了解 ABI，进行内存分配，逐层分析 ir::program，给 IR 涉及到的变量运算分配寄存器，以及实现函数保存现场和还原现场的功能。我在本次实验中灵活运用了 RISC-V 中的伪指令，例如 mul、div、li、seqz、snez 等，这些伪指令容易理解并且使用起来方便，能较好的实现 IR 到 RISC-V 的转换。本次实验三的完成由于时间紧迫采取了简单的实现方法，舍弃复杂的寄存器分配策略，将 t0 和 t1 作为 rs1 和 rs2，将 t2 作为 rd，虽然运行性能会变慢，但是结果是正确的。因为利用的寄存器较少，所以有些地方需要频繁的加载和存放数据，因此封装了 lw()和 sw()函数用于在内存和寄存器之间存取。其中函数 gen()分析 program，gen_global()分析全局变量 global_val，gen_func()分析函数 function，gen_instr()分析 IR。其中 gen_str()负责将单条 IR 翻译成对应 RISC-V 指令，可能是一对一，也可能一条翻译成多条。

2. 在 IR 中你如何处理全局变量的，这样的设计在后端有什么好处？后端中如何处理全局变量

本次实验将全局变量的定义 IR，全部放在了自定义增加的 global 函数中，记录了全局变量的名字、类型和值，语义分析开始时就向 function 向量压入 global()函数，因此 global 函数是第一个。

```
Function* global_func = new Function("global", Type::null);

symbol_table.functions.insert({"global", global_func});  // 符号表插入全局函数
```

ir::program 中除了设置 ir::function 向量记录函数体，还设置了 GlobalVal 向量记录所有全局变量名和类型，其中 GlobalVal 是一个 ir::operand，对于数组而言还有一个长度的属性。本次实验的测试用例没有全局变量与全局函数的交错定义，所以对全局变量的记录是在第一次进入全局函数定义时进行的，扫描 global()函数中全局变量定义的 IR，保存全局变量。在我的实现方式中，GlobalVal 向量的作用就是快速用变量名判断该变量是否为全局变量，用于实验三的 find_global_operand()函数中。

```
if (buffer.functions.size() == 1){     // 如果函数只有global,此时来新函数了,需要扫描global函数的IR向
    auto global_ir = buffer.functions[0].InstVec;
    for (int i=0; i<(int)global_ir.size(); i++){   // 扫描global函数的定义IR
        buffer.globalVal.push_back(ir::GlobalVal(global_ir[i]->des));  // 填入全局变量
    }
}
```

这样的设计在实验三的 gen_global()中分析 global 函数，可以对数据段.data 进行静态区空间的分配，较为便捷。

下图是分析一条全局变量定义 IR，并在静态区进行空间分配，.globl 标识变量名，.type 标识变量类型，.size 标识大小为 4 字节，.align 标识对齐，用.word 赋值：

```
// 只处理Int
if (inst->op == Operator::def)
{                                ir::Instruction *inst
    fout << "\t.globl\t" << inst->des.name << "\n";
    fout << "\t.type\t" << inst->des.name << ", @object\n";
    fout << "\t.size\t" << inst->des.name << ", 4\n";
    fout << "\t.align\t" << "4\n";
    set_label(inst->des.name);

    fout << "\t.word\t" << inst->op1.name << "\n";
}
```

5. 如何处理数组作为参数的情况，为什么可以这么做？

在语法制导翻译中，对于整数数组变量，其类型为 IntPtr。在 IR 翻译成 RISC-V 过程中，数组指针可以作为函数实参，例如 func(arr)，我的处理方式是先判断该数组指针来自何处，有三种情况：全局数组、当前函数内定义的数组、当前函数的数组参数。全局数组：使用 LA 指令读取地址存入 a0 寄存器；当前函数内定义的数组：将栈指针与数组栈中偏移量相加，存入 a0 寄存器；当前函数的数组参数：读取地址后存入 a0 寄存器。总而言之，存入 a0 寄存器的都是数组第一个元素的地址。

对于函数形参，进入函数时，同其他变量一样将 a0 的值存入栈中。调用时也有全局数组、当前函数内定义的数组、当前函数的数组参数这三种情况，获取 arr[i]地址时根据数组的第一个元素地址+偏移量*4 即可得到其地址。

上述对于数组作为参数的实现方式，较好地符合了数组指针的含义，但是存在问题：函数内定义的数组和函数的数组参数都会存在栈中，当使用 arr[i]时如何区分 arr 对应栈中的值是第一个元素的地址还是第一个元素针对栈指针的偏移？我的做法是遍历当前函数的参数列表 ParameterList，判断是否在其中，确定是否是数组参数。当然，完全可以在定义数组的时候在最前面 4 个字节存放一下数组第一个元素的地址，这样在用的时候就不用区分了。

6. 如何支持短路运算？

支持短路运算相关内容在实验二的语法制导翻译过程中，在 analysisLAndExp()和 analysisLOrExp()函数中要实现短路运算，即 a || b 逻辑表达式，如果 a 成立了，条件成立，不用判断 b，对于 a && b，如果 a 不成立，条件不成立，不用判断 b。

对于 LOrExp，主要实现代码如下：

```
buffer.push_back(new Instruction({op1, {}, t1, Operator::mov}));    // 将op1转换为变量
buffer.push_back(new Instruction({t1, {}, {std::to_string(tmp.size()+3), Type::IntLiteral}, Operator::_goto}));
buffer.insert(buffer.end(), tmp.begin(), tmp.end());    // 在尾部加入orexp节点的IR向量
buffer.push_back(new Instruction({op2, {}, des, Operator::mov}));   // else -> des = op2
buffer.push_back(new Instruction({{}, {}, {"2", Type::IntLiteral}, Operator::_goto}));
buffer.push_back(new Instruction({"1", Type::IntLiteral}, {}, des, Operator::mov));
```

a 成立就会跳过执行 b 生成的 IR，条件赋值为 1，a 不成立会执行 b 生成的 IR，跳过条件赋值为 1 的 IR，条件赋值为 b 的结果。

对于 LAndExp，主要实现代码如下：

```
buffer.push_back(new Instruction({op1, {}, t1, Operator::mov}));    // 将op1转换为变量
buffer.push_back(new Instruction({t1, {}, {"2", Type::IntLiteral}, Operator::_goto}));  // op1成立,判断op2
buffer.push_back(new Instruction({{}, {}, {std::to_string(tmp.size()+3), Type::IntLiteral}, Operator::_goto}));
buffer.insert(buffer.end(), tmp.begin(), tmp.end());    // 在尾部加入landexp节点的IR向量
buffer.push_back(new Instruction({op2, {}, des, Operator::mov}));   // op2成立,des = op2
buffer.push_back(new Instruction({{}, {}, {"2", Type::IntLiteral}, Operator::_goto}));
buffer.push_back(new Instruction({"0", Type::IntLiteral}, {}, des, Operator::mov));
```

a 不成立就会跳过执行 b 生成的 IR，条件赋值为 0，a 成立会执行 b 生成的 IR，跳过条件赋值为 0 的 IR，条件赋值为 b 的结果。

四、实验测试

1. 测试程序是如何运行的：

CmakeLists.txt 设置了项目的基本配置，使用 cmake 构建和编译项目得到可执行文件 compiler，运行 compiler 编译测试用例得到输出结果，并将结果与参考结果比对打分。

2. 进行测试执行的命令：

实验一在 windows 本地环境中进行，实验二和实验三在 docker linux 环境中进行。在 docker 容器中进行测试：

1. 创建并运行容器：

   `docker run -it -v D:\Compiler\lab2:/coursegrader frankd35/demo:v3`,

2. 进入 test 目录：

   `cd /coursegrader/test`

3. 编译代码：

   `python3 build.py`

   调用 build.py 文件进行编译，该文件自动创建所需目录，并且判断运行平台是 windows 还是 linux，执行对应的编译语句。

4. 编译单个测试用例：

（1）`cd /bin`　　进入 bin 目录

（2）`compiler <源文件路径> [s0/s1/s2/S] -o <输出文件路径 >`

5. 编译所有测试用例并打分

处于 test 目录下，执行：

（1）`python3 run.py s0/s1/s2/S`

调用 run.py 文件编译所有测试用例，将结果输出在 output 目录中

（2）`python3 socre.py s0/s1/s2/S`

调用 score.py 文件对结果进行比对打分

3. 汇编是如何变成 RISC 程序并被执行的：

根据 RISC-V 中文手册中的 ABI，将实验二生成的中间标识 IR 逐条转换为 RISC-V 汇编，并且加入了数据定义、函数调用时保留现场和函数返回时还原现场等内容，生成 RISC-V 汇编后使用 riscv32-unknown-linux-gnu-gcc 指定编译器编译汇编程序生成二进制流，qemu 可以模拟 CPU 和内存，运行 qemu-riscv32.sh 脚本在 qemu 上运行 risc-v 程序，输出结果。另外对照结果并打分使用的是 diff。

五、实验总结

1. 遇到的问题与解决方法：

Debug 中遇到的问题多了去了，解决过程都是根据 ref 中的预期结果，一步一步看 output 中输出的结果存在什么问题，再对源代码进行修改。写报告时有些问题也记不清了，列举其中一些问题：

（1）前期遇到较为麻烦的问题就是环境了，由于之前不是很懂如何使用 docker，故在自己的机器上不断调试和配置，勉强能解决毛病，但是实验二的时候就发现环境报错难以解决。因此实验二和实验三转为使用 docker 容器，发现使用方法十分简单，再也没有环境问题的忧虑了。更离谱的是，不在 docker 里面运行还会出现有些测试点时而对、时而不对的情况，docker 就不会出现这种情况。

（2）实验二语义分析中本来为了编码的简易，对于 LAndExp 和 LOrExp 打算直接使用 and 和 or IR 进行翻译，有点偷懒的意味，没想到竟有测试用例特意考察了这个短路机制，所以不得不重写这一部分，实现了短路机制。短路机制确实是比较实用，也不是很复杂，可以减少生成的 IR 和执行的操作。

（3）实验三将每一条 IR 翻译成 RISC-V 的汇编时，原以为直接能找到对应的，没想到还是挺麻烦的，比如一条简单的 leq(<=)，要使用 RISC-V 的话，我的方式是将其转换为 a>b 取反，还得将 a>b 转换为 b<a，翻译成了 5 条以上 RISC-V 汇编，还是挺麻烦的，其他很多 IR 的翻译都要考虑许多，容易出错。好在是我们寝室几个人及时交流心得，一些难点我们一同讨论得到解决思路和做法，大大提高了效率。

```
case Operator::leq: {   // <=, 例如a<=b, 转换为a>b取反, a>b转换为b<a

    lw(inst.op2, rvREG::X5, str);   // t0 load op2
    lw(inst.op1, rvREG::X6, str);   // t1 load op1
    str += ("\t" + toString(rvOPCODE::SLT) + "\t" + toString(rvREG::X5) + "," + toString(rvREG::X5) + "," + toStr
    str += ("\t" + toString(rvOPCODE::SEQZ) + "\t" + toString(rvREG::X7) + "," + toString(rvREG::X5) + "\n");
    sw(inst.des, str);  // des save t2

} break;
```

其他很多编译报错、结果与预期不符等问题，都是一步一步排查过来的，基本上都是从通过 0 个点开始一路排查到 50 多个点，这个过程还是比较有成就感，做到后面理解就比较深刻了，真正学到了东西。

通过本次实验加深了 cmake 工具编译，git 管理代码，docker 虚拟环境的使用，意识到合适的代码工具确实十分重要，可以减少工作量，同时，课上学到总觉得比较晦涩难懂的知识也在本次实验自己亲自动手后深刻地理解到内涵了，果然实践出真知，这门课尤其是。

2. 建议：

（1）做实验的话需要课上讲了对应的基础知识，由于本次实验二涉及的工作很多，导致课上还没讲到实验二相关知识，实验课却已经开始进行实验二的讲解了，此时大部

分人都在写实验一，没有心思听实验二的讲解，讲实验三的时候大家都在忙实验二，时间都错开了，只能回头看视频。

（2）实验一的框架包括宏定义很好用，简化了代码，减少了 debug 的时间，但是实验二和实验三的框架就非常不好用，需要进行大改动。指导书上的描述并不详细，缺乏讨论的话很多地方很难想出解决方法。

（3）这个实验框架初入手时感觉非常难，做到一半的时候才觉得思路慢慢清晰起来，越做越顺利。缺乏前期入门的指导，很容易就想放弃写，需要加强前期指导。