摘 要

本文主要讨论了针对SysY2020语言的编译器实现及优化。首先设计源代码层的预优化，实现了循环展开和表达式优化等预优化。然后将代码经过词法分析，语法分析转化为AST，并自主设计了中间代码表示，从而遍历AST根据语义信息生成中间代码。经过中间代码优化器，完成了公共子表达式消除等优化手段。最后将中间代码通过所实现的映射规则转换为Arm v7-M汇编代码，在Rasp-bery 4B上运行，并进行结果校验。编译器使用C++实现，完成了所有功能测试的编译，并通过了正确性验证。

**关键词：编译器，中间代码生成，编译优化，Arm v7**

# 目 录

[目 录 II](#_Toc49994780)

[第一章 针对复杂工程问题的方案设计与实现 1](#_Toc49994781)

[1.1 针对复杂工程问题的方案设计 1](#_Toc49994782)

[1.1.1 汇编语言分析及信息提取 1](#_Toc49994783)

[1.1.2 目标文件生成 4](#_Toc49994784)

[1.1.3 链接及可执行文件生成 5](#_Toc49994785)

[1.2 针对复杂工程问题的推理分析 6](#_Toc49994786)

[1.2.1 汇编语言信息收集分析 6](#_Toc49994787)

[1.2.2 目标文件生成分析 7](#_Toc49994788)

[1.2.3 链接过程分析 7](#_Toc49994789)

[1.3 针对复杂工程问题的方案实现 8](#_Toc49994791)

[1.3.1 汇编分析器的实现 8](#_Toc49994792)

[1.3.2 目标文件生成的实现 21](#_Toc49994793)

[1.3.3 链接器的实现 23](#_Toc49994794)

[第二章 系统测试 30](#_Toc49994796)

[2.1 测试环境的搭建 30](#_Toc49994797)

[2.2 测试用例的构造与测试结果 31](#_Toc49994798)

[第三章 知识技能学习情况 41](#_Toc49994799)

[3.1 ELF文件格式 41](#_Toc49994800)

[3.2 静态链接机制 41](#_Toc49994801)

[3.3 等等等等 42](#_Toc49994802)

[第四章 分工协作与交流情况 43](#_Toc49994803)

[4.1 分工协作 43](#_Toc49994804)

[4.2 团队交流情况 43](#_Toc49994805)

[参考文献 44](#_Toc49994806)

[致谢 45](#_Toc49994807)

# 第一章 针对复杂工程问题的方案设计与实现

## 针对复杂工程问题的方案设计

### 1.1.1 语言描述及编译器架构

### 1.1.2 源代码预优化

### 1.1.3 中间代码设计及其优化

## 针对复杂工程问题的推理分析

### 1.2.1 汇编语言信息收集分析

首先，对于系统前期汇编语言信息收集分析这一部分模块的总体架构的设计我们采用的是基本基于传统汇编器的流程，代码采用C++面向对象的方法进行编写。从本质上讲，汇编器也是编译器，也都需要进行词法分析、语法分析、语义处理、符号表管理和代码生成（机器代码）等阶段。但我们在这一模块的设计上与传统的汇编器不同，在借鉴了汇编器采用两边扫描的方式设计的基础上，为了方便后续的操作和效率的提高，我们大胆尝试在在传统的流程上略做改动。这样做原因有以下几点：

1、传统的汇编器设计流程已经较为成熟，且如果想要修改具有较高的风险性和困难性，因此我们考虑系统主干依然是基于传统的汇编器流程。

2、在保留传统流程主干的同时，由于我们在进行需求分析时完成了一些假定，因而在这种预设条件下稍做修改有利于优化的执行和效率的提高。

对于汇编信息的读入，我们设计了一个名为ARM\_analyze的C++类，这是因为通过C++面向对象进行编写可以使得代码逻辑更为清晰。同时根据我们的了解，在现实的应用场景中，大多数实现此类软件的软件开发者都会采用C++的类去组织代码。

为了完成这个类，我们需要定义很多辅助的数据结构。经过详细地设计，我们认为需要一下积累辅助结构：

①符号：这里的符号包括函数、标号、全局变量。如果是文件中定义的函数 或 标号 会记录函数定义的位置和标号的位置 如果是全局变量则记录在data段中的偏移。

②重定位符号：记录与重定位相关的符号。需要重定位的可以是函数或者全局变量。

③ARM指令结构：将ARM的信息保存在结构体里便于我们使用。

④数据：记录数据段中数据元素的信息，便于接下来目标文件生成。

相对应的，我们设计出以下函数类型来完成模块的整体功能：

①分发arm代码的函数。这也是本模块的入口函数。

②处理 .global 语句的函数。这是新建一个符号。

③处理 .type 语句的函数。这是对符号类型的补充声明。

④处理 Lable 语句的函数。在这里需要处理对函数的声明Merge、对全局变量声明以及标号的声明。

⑤处理 .space/.word 语句的函数。 这里假定我们在之前生成的代码里.text段里没有.word和.space 这种数据声明语句只存在data段中。

⑥处理一般的指令的函数。这个函数要处理除上述特殊语句外其余的所有函数。因而这个函数的工作量较大。

⑦在所有指令生成后回去处理指令中的跳转标号。

为了记录必要的信息，在C++类中还需要设计两个特殊的记录变量。一个用于记录当前指令或标号（或函数）在.text中的相对位置，一个用于记录全局变量在data段中的偏移。

### 1.2.2 源代码预优化分析

### 1.2.3 中间代码设计分析

### 1.2.4 数据流分析

## 针对复杂工程问题的方案实现

### 1.3.1 汇编分析器的实现

对于该项目的开发，我们采用了C++语言进行开发，一是为了使用C++容器对数据进行管理，二是为了方便实现面向对象的方式。

在汇编分析器的实现阶段，我们对编译器生成的汇编代码进行读取并分析，得到了代码变量、符号等信息，为下一步目标文件的生成打下了坚实的基础。

我们设计的部分辅助函数的代码如下所示：

|  |
| --- |
| typedef struct{  int type; //区分是函数还是全局变量, -1表示NOTYPE(未定义) 0表示函数, 1表示全局变量, 2表示标号  bool defined; //判断该符号是否在此文件中定义过  std::string name; //符号名称  int value; //如果是文件中定义的函数 或 标号 会记录函数定义的位置和标号的位置 如果是全局变量在data段中的偏移  int bind; //判断符号的作用域是global还是local 0表示global 1表示local  }symbols;  typedef struct{  int type; //需要重定位的是函数还是全局变量  std::string name; //符号名称  int value; //需要重定位的地方在.text中的偏移  //！！！当为 type = 0 时，value 为该跳转语句的偏移，需要再往后移动几个bit才是重定位地址！！！  }reloc\_symbol;  typedef struct{  std::string op\_name; //操作符名称 "mov" "ldr" "str"等  std::string Operands1; //操作数  std::string Operands2;  std::string Operands3;  std::vector<std::string> reglist; //如果操作符是push或pop 需要一个寄存器列表  }arm\_assem;  typedef struct{  std::string op\_name; //数据声明语句 .word或.space  int value; //声明数据的值  }data\_element; |

由上述代码可知，我们设计的数据结构能够较好地达到我们实际需求以及设计的要求。

我们将C++类的定义写在了HPP文件中。头文件部分代码如下：

|  |
| --- |
| class ARM\_analyze  {  public:  static vector<symbols \*> symbol\_list;  static vector<reloc\_symbol \*> reloc\_symbol\_list;  static vector<arm\_assem \*> arm\_assem\_list;  static vector<data\_element \*> data\_element\_list;  static vector<bss\_element \*> bss\_element\_list;  private:  int offset\_text = 0; //用于记录当前指令或标号（或函数）在.text中的相对位置，每处理一个正常指令（非伪指令、标号、函数），offset\_text+=4  int offset\_data = 0; //还设立一个全局变量offset\_data记录全局变量在data段中的偏移,每处理一个.word或.sapce offset\_data增加对应的数目  bool is\_in\_text = false; //判断汇编代码是否进入.text区  public:  void clear();  //逐行读取arm汇编代码arm\_assemble将每一行汇编交给arm\_handler向不同模块分发  void arm\_analyze(string arm\_assemble);  private:  void arm\_handler(string arm); //分发arm代码  void \_globl\_handler(string arm); //处理 .global 语句  void \_type\_handler(string arm); //处理 .type 语句  void \_label\_handler(string arm); //处理 Lable 语句  void \_data\_handler(string arm); //处理 .space/.word 语句  void \_instruction\_handler(string arm); //处理一般的指令  void \_lable\_fixer(); //在所有指令生成后回去处理指令中的跳转标号  string get\_next\_reg(string arm, int &index);  public:  // 构造函数  ARM\_analyze(string arm\_assemble) {  arm\_analyze(arm\_assemble);  }  ARM\_analyze() {};  ~ARM\_analyze() {  this->clear();  }  };  #endif |

对类中的函数做一下解释：

①void ARM\_analyze::arm\_analyze(string arm\_assemble)：这个函数在我们设计的汇编信息读取模块中处于入口的低位。在这个部分中，输入的形式为字符串形式的完整的汇编代码。利用C++中分割String类型字符串的方法将预备处理的大段汇编代码拆分为每一行单独的汇编代码，在输入中每一行的代码用\n做分割字符串的标识。最终分割得到的每一行汇编代码传递到arm\_handler函数中做下一步的处理。

②void ARM\_analyze::arm\_handler(string arm)：这个函数紧跟①的函数做处理。此函数将会根据不同指令特点将指令分发给不同模块处理。

③void ARM\_analyze::\_globl\_handler(string arm)：创建一个symbols并做相应初始化、并加入symbol\_list。

④void ARM\_analyze::\_type\_handler(string arm)： \* 根据name在symbol\_list中查找对应符号（通常在链表的结尾，可以从后往前查找），将其中的type改为%type对应的值, 没找到的话创建一个Symbols并初始化。

⑤void ARM\_analyze::\_label\_handler(string arm)：根据名称在symbol\_list中查找（通常在链表的结尾，可以从后往前查找），如果没找到则创建一个symbols并做相应初始化后加入symbol\_list，如果找到的话将对应项中是否定义改为已定义。在这里根据标识符is\_in\_text区分是全局变量还是函数。

⑥void ARM\_analyze::\_data\_handler(string data\_inst)：根据数据大小移动offset\_data,然后根据相应语句和数据生成data\_element并加入data\_element\_list。

⑦void ARM\_analyze::\_instruction\_handler(string arm)：根据字符串的拆分，分析相应的操作符和操作数，创建一个arm\_assem并填写相应的值后加入arm\_assem\_list。

C++类主体部分代码如下所示，下面这一部分是输入字符串形式的完整的汇编代码，利用String类自带的剪裁字符串的方法逐行拆分arm代码。其中注意需要加一句代码，其作用是去除每一句结尾的换行符。

|  |
| --- |
| vector<symbols \*> ARM\_analyze::symbol\_list;  vector<reloc\_symbol \*> ARM\_analyze::reloc\_symbol\_list;  vector<arm\_assem \*> ARM\_analyze::arm\_assem\_list;  vector<data\_element \*> ARM\_analyze::data\_element\_list;  vector<bss\_element \*> ARM\_analyze::bss\_element\_list;  void ARM\_analyze::arm\_analyze(string arm\_assemble) {  string arm\_code = arm\_assemble;  int size = arm\_code.length();  int head = 0;  int index = 0;  while (head != size) {  if (head == 0) {  index = arm\_code.find("\n");  }  else {  index = arm\_code.find("\n", head);  }  string str = arm\_code.substr(head, index - head + 1);  index++;  head = index;  str = str.substr(0, str.length() - 2);  arm\_handler(str);  }  \_lable\_fixer();  } |

下面这一段代码的意图是承接上面的代码结果，输入为单独一行的汇编代码，同时根据不同指令特点将指令分发给不同模块处理。依旧是利用C++ String类的功能函数匹配去转移到对应处理机制。涉及到较多的枚举。

|  |
| --- |
| void ARM\_analyze::arm\_handler(string arm) {  if (arm.find(":") != arm.npos) {  \_label\_handler(arm);  return;  }  if (arm.find(".text") != arm.npos) {  is\_in\_text = true;  return;  }  if (arm.find(".globl") != arm.npos || arm.find(".global") != arm.npos) {  \_globl\_handler(arm);  return;  }  if (arm.find(".type") != arm.npos) {  \_type\_handler(arm);  return;  }  if (arm.find(".word") != arm.npos || arm.find(".space") != arm.npos) {  \_data\_handler(arm);  return;  }  if (arm.find("pop ") != arm.npos ||  arm.find("push ") != arm.npos ||  arm.find("pop") != arm.npos ||  arm.find("mov ") != arm.npos ||  arm.find("str ") != arm.npos ||  arm.find("ldr ") != arm.npos ||  arm.find("cmp ") != arm.npos ||  arm.find("mul ") != arm.npos ||  arm.find("add ") != arm.npos ||  arm.find("sub ") != arm.npos ||  arm.find("bl ") != arm.npos ||  arm.find("b ") != arm.npos ||  arm.find("beq ") != arm.npos ||  arm.find("bne ") != arm.npos ||  arm.find("ble ") != arm.npos ||  arm.find("bge ") != arm.npos ||  arm.find("blt ") != arm.npos ||  arm.find("bgt ") != arm.npos) {  \_instruction\_handler(arm);  return;  }  return;  } |

下面这一段是处理一般的指令。输入为单独一行的汇编代码。对于正常的汇编指令，它根据字符串的拆分，分析相应的操作符和操作数，创建一个arm\_assem并填写相应的值后加入arm\_assem\_list。特殊注意的点为，对于跳转语句 如 bl memset(各种跳转)，需要根据跳转的目标标号名称在symbol\_list中查找，如果没有找到或找到的对应项中defined=false(该符号没有在此文件中定义) 则根据该名称创建一个reloc\_symbol,填写相关信息后加入reloc\_symbol\_list，然后留空跳转的相对位移(bl #0)。如果找到了函数或者标号，根据当前指令的位置减去跳转符号的位置(在symbol\_list中可以找到)可以得到相对位移。然后跳转语句转化成“ bl #相对位移”的形式。用该语句创建一个arm\_assem并填写相应的值后加入arm\_assem\_list。

为了简便，这里只展示此函数的部分代码。

|  |
| --- |
| void ARM\_analyze::\_instruction\_handler(string arm)  {  reloc\_symbol \*rel = new reloc\_symbol;  rel->name = var;  rel->type = 1;  rel->value = offset\_text;  reloc\_symbol\_list.push\_back(rel);  arm\_asm->Operands2 = "pc";  arm\_asm->Operands3 = "#-4";  arm\_assem \*arm\_asm2 = new arm\_assem;  arm\_asm2->op\_name = "nop";  arm\_assem\_list.push\_back(arm\_asm);  arm\_assem\_list.push\_back(arm\_asm2);  flag = false;  offset\_text += 4;  }  }  else if (opera == "mul" || opera == "add" || opera == "sub")  {  // find op1  int ptr = index2;  arm\_asm->Operands1 = get\_next\_reg(arm, ptr);  ptr++;  int len;  // find op2  arm\_asm->Operands2 = get\_next\_reg(arm, ptr);  ptr++;  // find op3  while (arm[ptr] != ',')  ptr++;  ptr++;  while (arm[ptr] == ' ')  ptr++;  if (arm[ptr] == 'r' || arm[ptr] == 'R' ||  arm[ptr] == 'p' || arm[ptr] == 'f' ||  arm[ptr] == 'l' || arm[ptr] == 's') // e.g. add r1, r2, r3  {  arm\_asm->Operands3 = get\_next\_reg(arm, ptr);  ptr++;  }  else if (arm[ptr] == '#') //e.g. add r1, r2, #100  {  arm\_asm->Operands3 = "#";  ptr++;  if (arm[ptr] == '-')  {  arm\_asm->Operands3.append("-");  ptr++;  }  for (int i = 0;; i++)  if (arm[i + ptr] < '0' || arm[i + ptr] > '9')  {  len = i;  break;  }  arm\_asm->Operands3.append(arm.substr(ptr, len));  }  }  else if (opera == "bl" || opera == "b" || opera == "beq" ||  opera == "bne" || opera == "ble" || opera == "bge" ||  opera == "bgt" || opera == "blt")  {  while (arm[index2] == ' ')  index2++;  string str = arm.substr(index2, arm.length() - index2 + 1);  if (str.find("(PLT)") != str.npos)  arm\_asm->Operands1 = str.substr(0, str.length() - 5);  else  arm\_asm->Operands1 = str;  arm\_asm->Operands2 = to\_string(offset\_text);  }  if (flag)  arm\_assem\_list.push\_back(arm\_asm);  } |

下面这一段功能函数用于在arm 里获取从 index 开始的最近一个 reg，结束后 index 在 arm 里 reg 的最后一个字符上。

|  |
| --- |
| string ARM\_analyze::get\_next\_reg(string arm, int &index)  {  while (arm[index] != 'r' && arm[index] != 'R' &&  arm[index] != 'p' && arm[index] != 'f' &&  arm[index] != 'l' && arm[index] != 's' &&  index < arm.length())  ++index;  if (index == arm.length())  return "ERROR";  if (arm[index] == 'p')  {  if (arm[++index] == 'c')  return "pc";  else  return "ERROR";  }  else if (arm[index] == 'f')  {  if (arm[++index] == 'p')  return "fp";  else  return "ERROR";  }  else if (arm[index] == 'l')  {  if (arm[++index] == 'r')  return "lr";  else  return "ERROR";  }  else if (arm[index] == 's')  {  if (arm[++index] == 'p')  return "sp";  else  return "ERROR";  }  else  {  int len = (arm[index + 2] >= '0' && arm[index + 2] <= '9') ? 3 : 2;  string ret = arm.substr(index, len);  index += (len - 1);  return ret;  }  } |

下面这段代码的意图是用于程序运行时的内存释放。设计这个函数考虑到了在实际场景中运行的量可能会很大，不释放内存会存在内存不足以满足业务需求的问题。

|  |
| --- |
| void ARM\_analyze::clear() {  #define FREE\_LIST(list) \  list.clear(); \  list.resize(0)  FREE\_LIST(this->arm\_assem\_list);  FREE\_LIST(this->bss\_element\_list);  FREE\_LIST(this->data\_element\_list);  FREE\_LIST(this->reloc\_symbol\_list);  FREE\_LIST(this->symbol\_list);  #undef FREE\_LIST  } |

下面这段代码的输入依旧是单独一行的汇编代码。这里假定我们在之前生成的代码里.text段里没有.word和.space 。这种数据声明语句只存在data段中。根据数据大小移动offset\_data,然后根据相应语句和数据生成data\_element并加入data\_element\_list 。

|  |
| --- |
| void ARM\_analyze::\_data\_handler(string data\_inst) {  data\_element \*d = new data\_element();  string op\_name;  int value;  // 按空格分成 .word 和 400 两部分  int split\_ndx = 0;  split\_ndx = data\_inst.find(' ');  if (split\_ndx == string::npos) {  fprintf(stderr, "[data\_handler]: invalid instruction!\n");  exit(EXIT\_FAILURE);  }  op\_name = data\_inst.substr(0, split\_ndx);  //added by yrc  op\_name = op\_name.substr(op\_name.find('.') + 1);  value = atoi(data\_inst.substr(split\_ndx).c\_str());  d->op\_name = op\_name;  d->value = value;  // 加入data组中  data\_element\_list.push\_back(d);  //added by yrc  if (op\_name == "word")  offset\_data += 4;  if (op\_name == "space")  offset\_data += value;  } |

下面这段代码根据name在symbol\_list中查找对应符号（通常在链表的结尾，可以从后往前查找），将其中的type改为%type对应的值, 没找到的话创建一个Symbols并初始化。

|  |
| --- |
| void ARM\_analyze::\_type\_handler(string arm)  {  static std::map<string, int> typeHandleMap = {  {"function", FUNCTION},  {"object", GLOBAL\_VAR},  };  //此处做了修改  string name = arm.substr(PREFIX\_LEN, arm.find(',') - PREFIX\_LEN);  name.erase(0, name.find\_first\_not\_of(" "));  // for (auto ch : arm.substr(PREFIX\_LEN))  // if ((ch <= 'z' && ch >= 'a') || (ch <= 'Z' && ch >= 'A') || (ch <= '9' && ch >= '0') || ch == '\_')  // name += ch;  // else  // break;  string type = arm.substr(arm.find\_first\_of('%') + 1);  bool isFind = false;  for (auto symbol : ARM\_analyze::symbol\_list) {  if (symbol->name == name) {  symbol->type = typeHandleMap[type];  isFind = true;  return;  }  }  if (!isFind) {  symbols \*newSym = new symbols();  newSym->type = typeHandleMap[type];  newSym->name = name;  ARM\_analyze::symbol\_list.emplace\_back(newSym);  }  } |

下面这段代码要对三种指令进行操作。分别是Merge:、.L0:和a:。对函数的声明Merge：进行处理时根据名称在symbol\_list中查找（通常在链表的结尾，可以从后往前查找），如果没找到则创建一个symbols并做相应初始化后加入symbol\_list，如果找到的话将对应项中是否定义改为已定义。如果是标号声明比如.L0，直接创建symbols初始化后加入symbol\_list。对全局变量声明a:进行处理时，根据名称在symbol\_list中查找（通常在链表的结尾，可以从后往前查找），如果没找到则创建一个symbols并做相应初始化后加入symbol\_list，如果找到的话将对应项中是否定义改为已定义。

|  |
| --- |
| void ARM\_analyze::\_label\_handler(string arm) {  if (arm.find(".") != string::npos) {  int front = 0;  int end = arm.find(":");  string n = arm.substr(front, end);  symbols \*a = new symbols();  a->defined = true;  a->value = offset\_text;  a->type = 3;  a->name = n;  symbol\_list.push\_back(a);  return;  }  if (is\_in\_text == true) {  //判断进入了text，说明是函数  int front = 0;  int end = arm.find(":");  string n = arm.substr(front, end); //找到函数名  int i = 0;  for(; i < symbol\_list.size(); i++) {  symbols \*b = symbol\_list[i];  if (n == b->name)  break;  } //寻找等于此符号名的symbol  if (i < symbol\_list.size()) //找到了  symbol\_list[i]->defined = true; //直接改  if (i == symbol\_list.size()) {  //没找到，新建插入  int front = 0;  int end = arm.find(":");  string n = arm.substr(front, end);  symbols \*a = new symbols();  a->defined = true;  a->value = offset\_text;  a->type = 0;  a->name = n;  symbol\_list.push\_back(a);  }  } else if (is\_in\_text == false) {  //全局变量也要先找symbol\_list里有没有  int front = 0;  int end = arm.find(":");  string n = arm.substr(front, end);  //added by yrc  int i = 0;  for (; i < symbol\_list.size(); i++) {  symbols \*b = symbol\_list[i];  if (n == b->name)  break;  } //寻找等于此符号名的symbol  if (i < symbol\_list.size()) //找到了  symbol\_list[i]->defined = true; //直接改  if (i == symbol\_list.size()) {  //没找到，新建插入  symbols \*a = new symbols();  a->defined = true;  a->value = offset\_data;  a->type = 1;  a->name = n;  symbol\_list.push\_back(a);  }  }  } |

下面这一部分代码主要完成创建一个symbols、做相应初始化并加入symbol\_list的流程。

|  |
| --- |
| void ARM\_analyze::\_globl\_handler(string arm) {  symbols \*a = new symbols();  int front = 0, end = 0;  front = arm.find(" ") + 1;  end = arm.length();  string n = arm.substr(front, end);  if (is\_in\_text) {  //函数  a->type = 0;  a->defined = false; //?  a->name = n;  a->value = offset\_text;  a->bind = 0;  } else {  a->type = 1;  a->defined = false; //?  a->name = n;  a->value = offset\_data;  a->bind = 0;  }  symbol\_list.push\_back(a);  } |

### 1.3.2 源代码预优化的实现

### 1.3.3 中间代码的设计与流图的构建

### 1.3.4 中间代码优化的实现

# 第二章 系统测试

## 2.1 测试环境的搭建

## 2.2 测试用例的构造与测试结果

# 第三章 知识技能学习情况

## 3.1 编译器前端相关知识

## 3.2 静态链接的学习

在我们的实际开发中，不可能将所有代码放在一个源文件中，所以会出现多个源文件，而且多个源文件之间不是独立的，而会存在多种依赖关系，如一个源文件可能要调用另一个源文件中定义的函数，但是每个源文件都是独立编译的，即每个\*.c文件会形成一个\*.o文件，为了满足前面说的依赖关系，则需要将这些源文件产生的目标文件进行链接，从而形成一个可以执行的程序。这个链接的过程就是静态链接。

静态链接生成可执行文件，是将所有的目标文件或静态库都链接成为一个可执行文件，换句话说，这个生成的可执行文件包含了所有的目标文件或静态库，目标文件或静态库被复制、嵌入到静态链接生成的可执行文件中。

静态链接对象一般为多个可重定位目标模块以及静态库（标准库、自定义库）。静态库将所有相关的目标模块（.o）打包为一个单独的库文件（.a），称为静态库文件 ，也称存档文件（archive）使用静态库，可增强链接器功能，使其能通过查找一个或多个库文件中定义的符号来解析符号。在构建可执行文件时，只需指定库文件名，链接器会自动到库中寻找那些应用程序用到的目标模块，并且只把用到的模块从库中拷贝出来。

静态链接一般采用两步链接（Two-pass Linking）的方法。

第一步，空间与地址分配。扫描所有的输入目标文件，获得它们的各个段的长度、属性和位置，并且将输入目标文件中的符号表中所有的符号定义和符号引用收集起来，统一放到一个全局符号表。连接器获取所有输入目标文件的段长度后，将它们合并。每个.o文件都有自己的段属性，比如.text、.data等等这些，链接的第一步就是将这些段属性合并在一起。

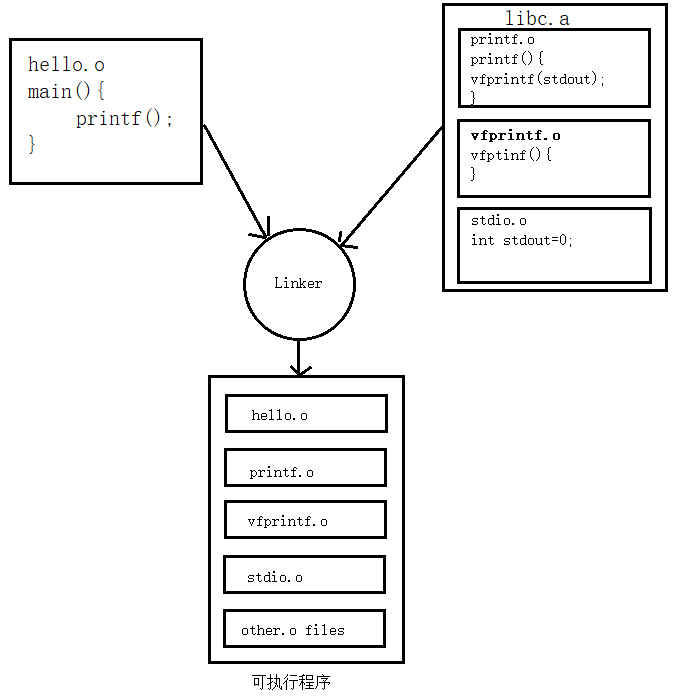
第二步，符号解析与重定位。使用上一步收集到的所有信息，读取输入目标文件中段的数据、重定位信息，并且进行符号解析与重定位、调整代码中的地址等。符号解析时，由于编译器将定义的符号存放在一个符号表（ symbol table）中，因而链接器可以将每个符号的引用都与一个确定的符号定义建立关联。重定位时将多个代码段与数据段分别合并为一个单独的代码段和数据段。计算每个定义的符号在虚拟地址空间中的绝对地址。将可执行文件中符号引用处的地址修改为重定位后的地址信息。

对上述步骤进行解读。

“链接器为目标文件分配地址和空间”这句话中的“地址和空间”其实有两个含义：第一个是在输出的可执行文件中的空间：第二个是在装载后的虚拟地址中的虚拟地址空间。对于有实际数据的段，比如“。text”和“。data”来说，它们在文件中和虚拟地址中都要分配空间，因为它们在这两者中都存在；而对于“。bss”这样的段来说，分配空间的意义只局限于虚拟地址空间，因为它在文件中并没有内容。事实上，我们在这里谈到的空间分配只关注于虚拟地址空间的分配，因为这个关系到链接器后面的关于地址计算的步骤，而可执行文件本身的空间分配与链接过程关系并不是很大。

那么链接器是怎么知道哪些指令是要被调整的呢？这些指令的哪些部分要被调整？怎么调整？事实上在ELF文件中，有一个叫重定位表（Relocation Table)的结构专门用来保存这些与重定位相关的信息，我们在学习ELF文件结构时已经学习过了重定位表，它在ELF文件中往往是个或多个段。

对于可重定位的ELF 文件来说，它必须包含有重定位表，用来描述如何修改相应的段甲的客对干每个要被重定位的 ELF段都有一个对应的重定位表，而一个重定位表往往表可以叫重定位段，我们在这里统一称作重定位表。

以下面这个图来简单说明一下从静态链接到可执行文件的过程，根据在源文件中包含的头文件和程序中使用到的库函数，如stdio.h中定义的printf()函数，在libc.a中找到目标文件printf.o(这里暂且不考虑printf()函数的依赖关系)，然后将这个目标文件和我们hello.o这个文件进行链接形成我们的可执行文件。

我们知道，链接器在链接静态链接库的时候是以目标文件为单位的。比如我们引用了静态库中的printf()函数，那么链接器就会把库中包含printf()函数的那个目标文件链接进来，如果很多函数都放在一个目标文件中，很可能很多没用的函数都被一起链接进了输出结果中。由于运行库有成百上千个函数，数量非常庞大，每个函数独立地放在一个目标文件中可以尽量减少空间的浪费，那些没有被用到的目标文件就不要链接到最终的输出文件中。

静态链接的缺点很明显，一是浪费空间，因为每个可执行程序中对所有需要的目标文件都要有一份副本，所以如果多个程序对同一个目标文件都有依赖，如多个程序中都调用了printf()函数，则这多个程序中都含有printf.o，所以同一个目标文件都在内存存在多个副本；另一方面就是更新比较困难，因为每当库函数的代码修改了，这个时候就需要重新进行编译链接形成可执行程序。但是静态链接的优点就是，在可执行程序中已经具备了所有执行程序所需要的任何东西，在执行的时候运行速度快。

## 3.3 优化相关知识

# 第四章 分工协作与交流情况

## 4.1 分工协作

## 4.2 团队交流情况

# 参考文献

1. GNU.Bison Table of Cotent[OL].

https://www.gnu.org/software/bison/manual/bison.html, December 2019

1. GNU. Flex Table of Content[OL]. http://www.gnu.org/software/flex/
2. 周尔强,周帆,韩蒙,陈文宇. 编译技术[M].北京:机械工业出版社, 2019,121-122
3. ARM v7-M Architecture Reference Manual[OL]. https://www.arm.com/
4. Alfred V.Aho, Monica S.Lam, Ravi Sethi, Jeffrey D.Ullnab. 编译原理第2版[M]. 北京:机械工业出版社, 2009.1

# 致谢

本报告所展示的工作是在我们的指导教师周尔强老师、廖勇老师悉心指导下完成的，感谢两位老师每周组间交流时为我们提供技术和方向上的指导；同时感谢全国大学生计算机系统能力大赛-编译系统设计赛（华为毕昇杯）组委会给我们提供了一个展示编译技术和编译优化能力的竞赛平台。