

教育部高等学校大学计算机课程教学指导委员会

中国大学生计算机设计大赛



软件开发类作品文档简要要求

作品编号： 2021025036

作品名称： WebC-基于自主开发编译器的高性能易用 Web 后端语言

作 者： 原毅哲 柯常仁 金韬

版本编号： 2.2.1

填写日期： 2021 年 5 月 31 日

填写说明：

- 1、本文档适用于**所有**涉及软件开发的作品，包括：软件应用与开发、大数据、人工智能、物联网应用；
- 2、正文一律用五号宋体，一级标题为二号黑体，其他级别标题如有需要，可根据需要设置；
- 3、本文档为简要文档，不宜长篇大论简明扼要为上；
- 4、提交文档时，以 PDF 格式提交本文档；
- 5、本文档内容是正式参赛内容组成部分，务必真实填写。如不属实，将导致奖项等级降低甚至终止本作品参加比赛。

目 录

第一章 需求分析	3
第二章 概要设计	4
第三章 详细设计	7
第四章 测试报告	34
第五章 安装及使用	42
第六章 项目总结	43

第一章 需求分析

随着操作系统的发展，计算机的功能日益增强。如何将计算机功能封装起来，满足具体某研究或工业用途，一直是计算机学界热门的话题。最方便有效的方法是设计一门语言。比如，MATLAB 就是一种在 Java 的基础上进行修改后产生的更适合各种科学计算的专用语言。

众所周知：在搭建一个简单的 Web 后端时，常用的语言有 Java，Python 等。C/C++的运行速度快，但是由于其接近操作系统底层，编写起来费时费力，往往不是敏捷开发的主流选择。Java 配合使用 Spring 框架能够在短时间内完成一个小型的 Web 服务，但是代码需要解释执行，且需要 Java 虚拟机支持，对于设备的性能要求较高。Python 编写后端最为简单，但是其性能不佳，在高并发的场景下不适用，且作为脚本语言，源代码直接暴露，具有较高的安全风险。NodeJS 是基于 ChromeV8 的运行环境，同样需要较好的硬件设备。此外，三者作为通用型高级语言，各类语法较多，具有较高的学习成本。对比如表 1.1 所示。

表 1.1 预期语言与常用语言对于 Web 后端开发的比较

开发框架	使用语言	学习成本	上手难度	性能要求	后端性能	执行方式
Boost	C++	高	高	低	优	编译
Spring	Java/Kotlin	中	中	高	优	解释
Flask	Python	低	低	中	中	解释
NodeJS	Javascript	低	低	中	优	解释
自定义	自定义	低	低	低	优	编译

该项目致力于打造一门语言，学习成本低廉，上手简单，运行时对硬件平台性能要求低，具备文件小，高并发的特性，且该语言注重源代码的可读性及可写性，更偏向于生成 Web 方面的可执行代码，帮助小白开发者能够在极短时间内用极少的代码来搭建一个简单、安全、高性能的 Web 后端程序。在模块化的 WebC 编译器架构的支持下，开发者能够快速地拓展新功能（目前已扩展 Kweb、Kjson、Ksql 等库用于支持 Web 服务的开发），无需任何包，无需管理依赖问题，生成的产物即为编译好的可执行文件，文件小、安全，且能够使用 Python 不能使用的多核 CPU 优势。

与此同时，基于 LLVM 强大的工具链，使得其拥有原生跨平台的特性，一份代码无需修改即可编译为各平台架构支持的目标代码，如 X86、ARM、WASM、MIPS 以及龙芯等平

台的二进制文件，实现跨平台兼容。

第二章 概要设计

WebC 编译器的总体设计思路为：在宏观架构上，采用基于 LLVM 标准的编译器三段式架构开发，如图所示。首先针对 WebC 语言开发 LLVM 支持的前端，同时在 LLVM IR 优化器中，加入 WebC 优化器以及 WebC IR 插桩器。其前端负责对用户输入的源程序进行解析并将其转化为 LLVM IR 代码，随后通过 WebC 优化器进行优化，通过 WebC IR 插桩器对其进行自定义修改（如函数时间分析等）。图 2.1 展示了 WebC 编译器的架构。

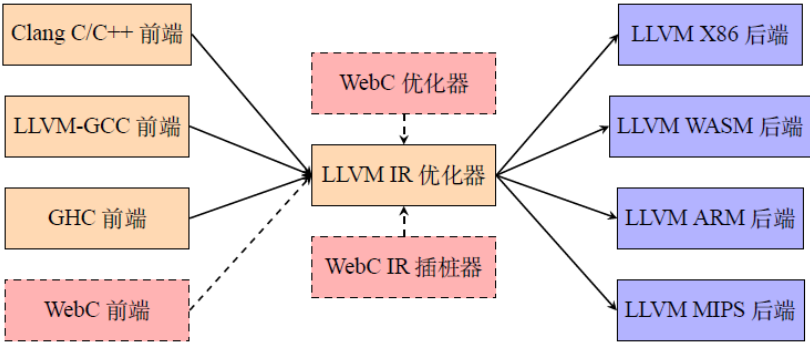


图 2.1 WebC 编译器架构

从编程的角度来看，编译器可按层级进行划分，即基础层、模块层、用户接口层以及集成开发环境层，如图 2.2 所示。

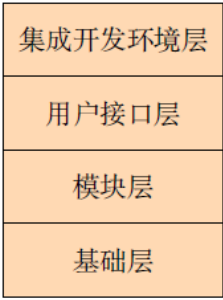


图 2.2 WebC 编译器层级关系

2.1 基础层

基础层为上层提供了原子级别的能力，方便上层实现逻辑，从而不用关心底层逻辑。同时，基础层为上层提供了基础功能如调试开关，日志记录等功能，统一上层的逻辑。总体上看，基础层由调试系统、读写系统、日志系统、LLVM IR 代码读写系统组成，如图 2.3 所示。

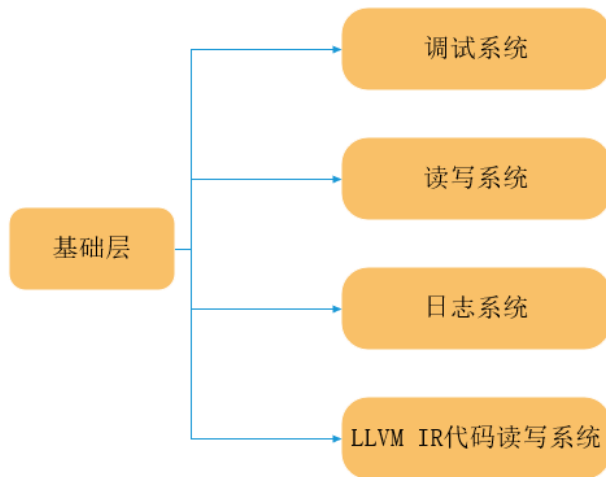


图 2.3 基础层组成

2.2 模块层

模块层在编译器架构上占有重要的低位。其将基础层提供的功能进行组合，为编译过程中的每一个过程提供支持。

模块层主要由编译器模块和库模块组成。编译器模块用于支持编译器翻译，含有词法、语法、语义、代码优化、目标文件生成模块，如图 2.4 所示。库模块用于库的扩展，为语言带来自定义功能，如图 2.5 所示。同时，库模块提供统一接口，使得其具备拓展性，开发者可以编写相应库参与编译，使得编译器支持新的特性。

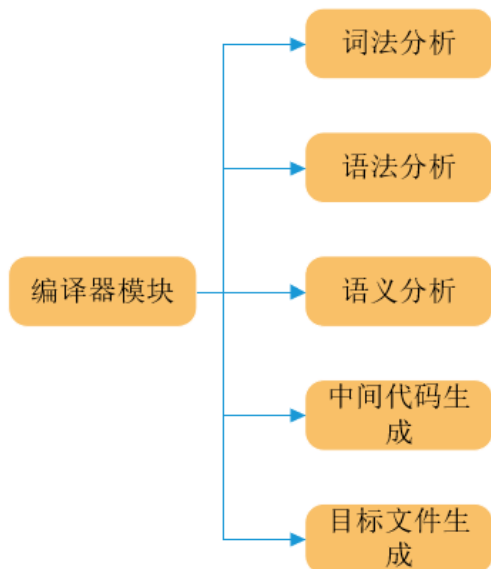


图 2.4 编译器模块组成

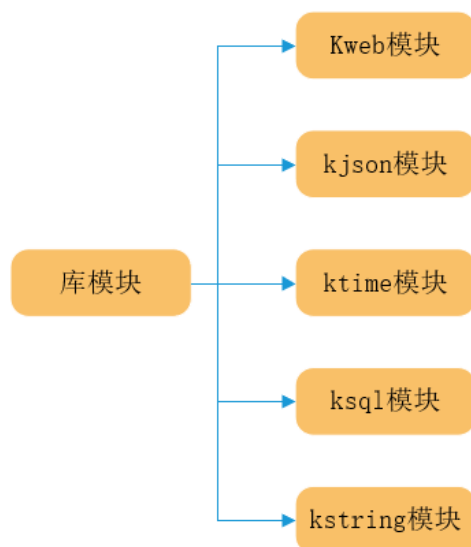


图 2.5 库模块组成

2.3 用户接口层

编译器用户接口层对编译器模块层进行封装。编译器用户接口层分为针对开发者、IDE（集成开发环境）设计。

面向用户。编译器直接为使用者提供服务。为方便用户，编译器以命令行界面(Command-Line Interface, CLI) 的形式供用户使用。用户传入设定好的相应选项，来控制编译器的编译逻辑。可用的编译选项如表 2.1 所示。

表 2.1 用户接口层编译选项

编译选项	参数个数	说明
-i/---input	≥1	编译输入文件路径/URL 路径
--o/--output	1	编译输出目标文件
-s/--as	0	生成可读汇编代码
-t/--time_analysis	0	运行时加入函数级时间分析

面向集成开发环境。集成开发环境拥有的功能如下：

- (1) 获取代码提示表，提供代码提示内容，如库函数名，语言关键字等
- (2) 单词法分析。单独运行词法分析，检查用户是否有非法字符
- (3) 语法分析。通过构建抽象语法树，检测用户输入是否合法，如字符串赋值给数值变量。
- (4) 编译。用户使用集成开发环境的编译功能，直接调用编译器对当前代码进行编译。
- (5) 日志输出。分为编译日志、运行日志。在编译时使用编译日志接口输出信息。在目标代码运行时，运行输出定向到运行日志接口。

2.4 IDE 总体设计

IDE 层是编译器的最高层，属于应用层，其对编译器功能进行了集成封装。通过集成编译器、代码编辑器、窗体界面、代码分析等功能，形成一个完整，易用的用户环境。

- (1) 编译器。使用编译器用户接口层提供的编译功能，对下层编译模块进行操作。
- (2) 窗体界面。IDE 参考界面仿照 Arduino IDE 进行设计，并加入实时分析内容。原型如图 2.6 所示。分为菜单栏、代码编辑器、日志三大组件。菜单栏负责文件读取，编译操作。代码编辑器给用户编写代码的空间。日志分为编译日志、运行日志与代码分析。在用户编辑过程中，需要在代码分析中实时显示用户代码问题；在运行时，程序输出需要显示在运行日志上；在编译时，编译器的输出需要显示在编译日志上。
- (3) 代码分析。使用编译器用户接口层提供的单词法分析、语法分析接口进行不同级别的代码分析，分析结果通过日志告知给用户。

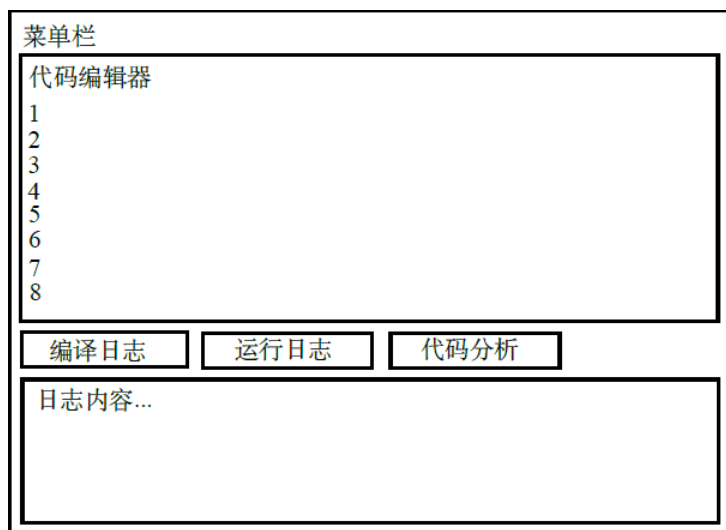


图 2.6 窗体原型图

集成开发环境下的业务流程图如图 2.7 所示。用户只需要编写代码，保存后进行编译，即可直接在窗口内得到执行结果。用户不需要关心链接库，只需要关注自身业务代码即可，极大简化了开发流程，提高了效率。

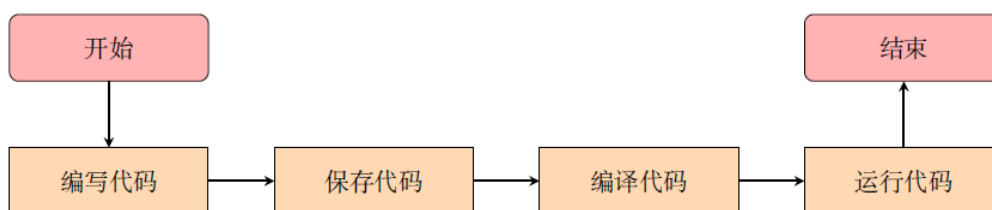


图 2.7 集成开发环境业务流程图

第三章 详细设计

3.1 基础层详细设计

基础层为编译器提供了基础服务，用于抽象底层实现。接口，用于规定不同类的共同方法，适用于基础业务的抽象。同时，上层业务只持有相应实现类的接口类，符合不同类之间信息最小共享原则，满足数据隔离的编程需求。

3.1.1 调试系统

调试系统是一个大型系统都应具备的信息，通过开关调试可以达到多种使用目的。打开调试，可以观察到编译器整体运行流程，方便进行问题修复。关闭调试，使得编译器只记录少量信息，从而加快编译器运行。故在开发时，应以调试模式为主。在用户使用时，应以非

调试模式为主。

CMake 是用于组织 WebC 编译器开发的一个编译管理工具。其具备的依赖查找，链接库查找，自定义编译选项等功能为开发调试模式提供了可能。在配置项目时，可在 CmakeLists.txt 以添加定义的形式，定义一个调试标志，命名为 DEBUG_FLAG，如代码 3.1 所示：

代码 3.1 添加调试标识

```
add_compile_definitions ( sysplus_compiler PUBLIC DEBUG_FLAG=1)
```

此处将调试标识定义为（Public）类型，表示此定义全编译器代码可见。代码中若遇到某些时机，需要打印关键的信息，可使用宏定义完成调试信息的打印。如代码 3.2 所示：

代码 3.2 代码调试域标识

```
# ifdef DEBUG_FLAG  
// 进行详细调试信息打印  
# endif
```

使用宏定义的好处是，在不定义调试标识时，调试代码域中的代码直接不参加编译，相比于“定义一个调试布尔值，每次查询是否处于调试模式”，效率更高。

编译器被定义为通过统一的调试标志（Debug Flag，DF）来感知用户当前的模式。若当前为调试（Debug）模式，词法分析时，将打印出每一个识别的标识符及其类型，具体内容参考下文词法分析详细设计部分。

3.1.2 读写系统

为编译器封装读取系统，提供整体、流式（stream）读取。使得上层只需要提供文件路径，即可通过返回的类的读写方法对类进行代码读取，不用考虑字节、流读写等操作。两种读取方法中，整体读取即为一次将文件所有内容读入内存进行分析，但对于一个大文件整体读取将不适用。若由于用户的疏忽，误将一个超大文件作为输入读入，这将导致内存超限，在配有交换分区（Swap）的系统上还将频繁触发额外的磁盘读写操作，导致操作系统不稳定或编译器被杀死。而流式读取很好的解决了读取大文件的问题，在编译器进行编译的时候设置缓冲，动态读取数据，很好的解决了大文件编译问题。编译器具备两种功能，在小型文件时采用整体读取，大型文件或网络文件使用流式读取。

IO 系统为流式读取提供了公共方法。为了保证最大的兼容性，系统定义了如代码 3.3 的接口，通过派生子类的形式实现源代码 IO 读取的各种方式，使其具有拓展性，同时屏蔽了底层差异。

```

class IFileReader {
public :
virtual int getChar () = 0; // 向后读取一个字符
virtual int seek () = 0; // 向后观察一个字符
virtual unsigned int getLineNo() = 0; // 获取当前行数
virtual unsigned int getCCol() = 0; // 获取当前列数
};

```

编译器为了满足不同场景下的使用需求，派生了三个大类，如图 3.1 所示。

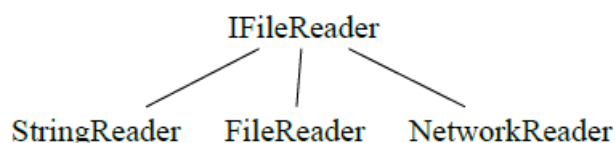


图 3.1 IO 类关系

(1) **StringReader**。用于直接从内存中读取代码数据。

(2) **FileReader**。用于从存储器中读取代码数据。

(3) **NetworkReader**。用于从 HTTP 服务器中读取代码数据。该类支持提供一个统一资源定位器 (Uniform Resource Locator, URL)，通过对其进行 GET 请求，获取流式相应进行编译。其中，网络流与前两者不同的是，网络数据流无法进行二次读取，无法进行向后观察。故上层在调用 seek 函数时，实际上是对流进行读取，同时将读取的数据放入缓冲区内。在下次 getChar 时首先读取缓冲区内数据，直到为空。

3.1.3 日志系统

日志作为向用户进行提示的方式，具备非常重要的作用。基础层为上层提供了统一的日志接口，方便统一日志格式。另外，在分析文件生成过程中若发现错误，都可及时通过日志向用户进行告知。同时，日志还可以发送给开发者，为编译器的完善提供有力的数据支持。编译器将日志按照重要等级划分，分为普通日志和错误日志。如图 3.2 所示，对于普通日志，利用标准输出流进行输出。对于错误日志，利用错误输出流输出。在大部分终端中，普通日志与错误日志输出颜色不同，日志系统会将错误日志标红，更符合用户的习惯。

若用户处于集成开发环境中，编译器需要将日志同时打印至窗口中日志显示位置。由此，所有的日志信息都通过一个常量字符指针 Str 传入，显示到窗口中。

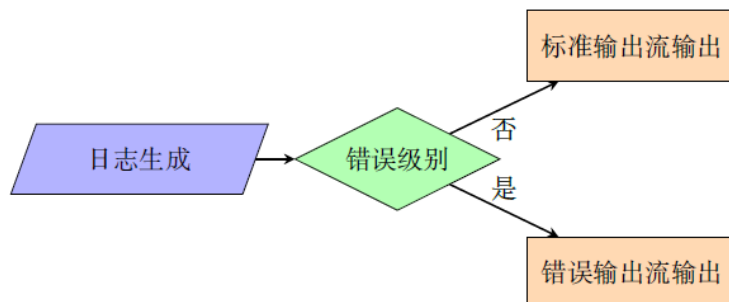


图 3.2 流输出选择

3.1.4 LLVM IR 代码读写系统

此系统负责生成 LLVM IR 代码。基础层为上层提供 LLVM IR 代码的原子级别操作，如插入、追加 LLVM IR 代码，获取当前函数域信息等，如下所示。

(1) 获取 LLVM IR 常量表。在解析用户代码时，当遇到用户输入的值为常量时，编译器能够通过 LLVM IR 代码读写系统查询相应的常量数据。LLVM IR 常量表是一张含有所有 LLVM IR 常量类型的表。开发者通过此表可构造任意的常量数据类型。如：整型（Integer），浮点（Float-Point）类型，数组（Array）类型，结构体（Struct）类型，空（Void）类型，标签（Label）类型，指针（Pointer）类型。

(2) 构造 LLVM IR 指令。在生成 LLVM IR 指令时，系统将 LLVM IR 指令的书写转化为对一个个函数的调用。方便上层对整体 LLVM IR 代码进行操作。综合考虑到 WebC 语言的语法，其应具有：分配内存（Alloca）、存储至内存（Store）、指针类型转换（Bitcast）、获取指针指向内容（GetElementPtr），函数调用（Call）、运算符比较（Cmp）、分配基础块（Create Basic Block），字位拓展或修剪（ZExtOrTrunc）。分配内存、存储到内存遵循系统 Load-Store 设计理念，通过 Load-Store 对变量进行存储；使用指针类型指针转换，用于丢弃/补充数据长度数据，适配含有无长度参数数组的函数调用，如输出函数 `int printf(const char * __restrict __fmt, ...)`，`__fmt` 接收的是无长度的字符数组，在进行调用时需要在 LLVM IR 代码中去除长度信息，转化为字符数组指针；获取指针指向内容用于取出数据内容；分配基础块用于更好构建程序执行流程图，使得编译过程清晰，易于优化。字位拓展或修建主要用于不同数据类型的转换，如 32 位整型与 64 位整型的互相转换。

(3) 获取函数域信息。系统提供获取当前的函数域功能，从而使得编译过程可感知是否处于函数内部，从而可感知用户声明的是局部变量或全局变量。另外，系统能够提供当前函数域内已写入的 LLVM IR 指令，在需要是可以修改其他 LLVM IR 指令达到特殊目的，如

代码优化阶段对未使用变量的删除。

3.2 模块层详细设计

模块层在编译器架构上占有重要的地位。其将基础层提供的功能进行组合，为编译过程中的每一个过程提供支持。

模块层由模块组成，模块又分为编译器模块与库模块。编译器模块用于支持编译器翻译，含有词法、语法、语义、代码优化、目标文件生成模块。库模块用于库拓展，为语言带来自定义功能，使用统一接口，使其具备拓展性，开发者可以编写相应的库参与编译，使编译器支持新的特性。

3.2.1 编译器模块

该模块主要分为词法分析、语法分析、语义分析、代码优化、目标文件生成 5 大模块，接下来分别详细介绍。

(1) 词法分析

词法分析通过基础层提供的读写系统读入源代码数据。相关业务如图 3.3 所示。在读入字符后，判断此时是否读取完数据，若读取完直接退出词法分析，若未读取完则查看字符是否有效。若字符为空白字符(如空格，Tab 字符等)，则将字符缓冲区内的数据进行标识符识别，否则将该字符加入字符缓冲区，和之前的字符一起参加标识符识别。

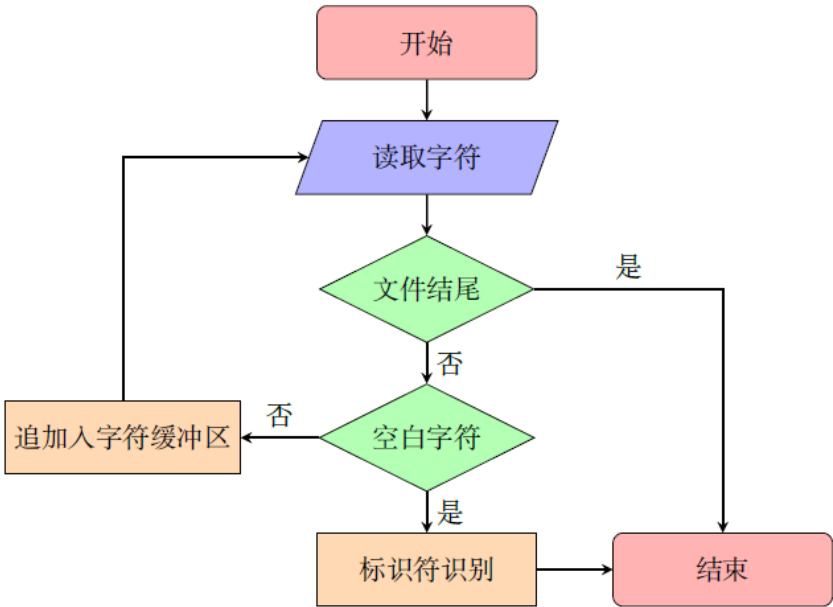


图 3.3 词法分析业务流程图

在标识符识别阶段，编译器通过分析字符缓冲区内的数据，将其进行分类为不同的标识

符类型。标识符类型在语言中含有：EOF 文件尾、字符标识符、小数、加减、乘除模、赋值、等于、不等于、小于、大于、取反、小于等于、大于等于、逗号、分号、小中大括号、常量标识、字符串、空值（null）、函数指针、循环保留字（lp、wh、out、cont）、条件保留字（if、el）、函数保留字（ret）、空返回值（void）、与或符、正负号。这些类型也称为词法终结符。

为了方便表示，编译器将标识符类型进行命名，如表 3.1 所示。词法分析器通过识别用户的输入，将其转换为表中标识符的类型。语法分析器只需根据这些约定好的标识符类型序列进行规约等操作。

在识别过程中，词法可能会出现二义性的类型，这使得词法分析器应该拥有向前分析 (seek) 的功能。如在区分“-”是负号还是减号时，向后查看一有效字符是否为数字，从而来判断是负号还是减号。

词法分析器的类定义如代码 3.4 所示，其通过传入一个 IO 系统接口类进行构造，提供一个共有方法 getNextToken 获取标识。和同名私有方法不同的是，公有方法对私有获取标识方法返回的内容进行了拦截，并将当前获取的标识以及字符输入日志和调试系统。

表 3.1 标识符类型说明

标识符类型	说明	标识符类型	说明	标识符类型	说明
T_IDENTIFIER	字符串标识符	T_INTEGER	整型数字	T_DOUBLE	浮点数字
T_ADD	加号	T_SUB	减号	T_MUL	乘号
T_DIV	除号	T_MOD	取余	T_ASSIGN	赋值
T_EQU	等于号	T_N_EQU	不等于号	T_LESS	小于号
T_GREATER	大于号	T_REVERSE	感叹号	T_LESS_EQU	小于等于号
T_GREATER_EQU	大于等于号	T_COMMA	逗号	T_SEMICOLON	分号
T_L_SPAR	(T_R_SPAR)	T_L_MPAR	[
T_R_MPAR]	T_L_LPAR	{	T_R_LPAR	}

T_CONST	常量关键字	T_STR	字符串	T_NULL	null
T_FUNC_PTR	函数指针	T_FOR	for	T_WHILE	while
T_OUT	跳出	T_CONTINUE	继续	T_IF	如果
T_ELSE	else	R_RETURN	返回	T_VOID	void

代码 3.4 词法分析类定义

```

class Lexer {
public :
...
explicit Lexer( IFileReader * fileReader ) : reader ( fileReader ) {}
...
int getNextToken(); // 获取下一个标识
private :
IFileReader * reader ; // 源代码 IO 系统接口类
int last_char = ' ' ; // 上一次获取的字符
int _getNextToken(); // 获取下一个标识
...
};

```

标识类型返回值类型为整型，编译器使用 `enum` 枚举类型对其进行编号，如代码 3.5 所示。

当词法分析器的 `getNextToken` 被调用后，会调用基础层的 IO 系统按字符读取代码，忽略空格、双斜杠开头的注释语句。

代码 3.5 标识符编号

```

enum yytokentype
{END = 0,T_IDENTIFIER = 258,T_INTEGER = 259,T_DOUBLE = 260,T_ADD = 261,T_SUB = 262,
T_MUL = 263,T_DIV = 264,T_MOD = 265,T_ASSIGN = 266,T_EQU = 267,T_N_EQU = 268,
T_LESS = 269,T_GREATER = 270,T_REVERSE = 271,T_LESS_EQU = 272,T_GREATER_EQU = 273,
T_COMMA = 274,T_SEMICOLON = 275,T_L_SPAR = 276,T_R_SPAR = 277,T_L_MPAR = 278,
T_R_MPAR = 279,T_L_LPAR = 280,T_R_LPAR = 281,T_CONST = 282,T_STR = 283,
T_NULL = 284,T_FUNC_PTR = 285,T_FOR = 286,T_WHILE = 287,T_OUT = 288,T_CONTINUE = 289,
T_IF = 290,T_ELSE = 291,T_RETURN = 292,T_VOID = 293,T_INT = 294,T_OR = 296,
T_AND = 297,T_MINUS = 298,T_POS = 299
};

```

(2) 语法分析

语法分析器使用 **Bison** 进行构造，需要在此基础上对语法进行约束。

① 定义标识优先级。在语言中需要对二元操作符进行优先级排序。**Bison** 中使用`%left`定义向左结合性，如代码 3.6 所示，代码中变量的具体含义可以参考词法分析。

代码 3.6 标识符优先级声明

```
%left T_ASSIGN
%left T_OR
%left T_AND
%left T_EQU T_N_EQU T_LESS T_GREATER T_LESS_EQU T_GREATER_EQU
%left T_ADD T_SUB
%left T_MUL T_DIV T_MOD
```

其中，优先级按照声明顺序升序排列。即乘、除、求模三个操作的优先级最高，加减操作次之，接下来是二元比较符、与、或、赋值。

② 定义无结合性标识。部分非终结符是不允许连续结合的，如“`x op y op z`”中的 `op`。若 `op` 定义为无结合性，语法分析则判断上述为错误。**WebC** 语法中的 `else`，`void` 等关键字就属于无结合性标识。使用`%nonassoc` 定义如代码 3.7:

代码 3.7 无结合性定义

```
%nonassoc T_L_MPAR
%nonassoc T_R_SPAR
%nonassoc T_MINUS T_POS T_REVERSE
%nonassoc T_ELSE
%nonassoc T_VOID T_INT
```

③ 定义规约类型。根据语法规则，定义如图 3.4 的类与关系。由图可得，**NodeAST** 为抽象语法树结点的父结点，定义了所有结点的公用方法，其详细如代码 3.8 所示。**NodeAST** 所有公有方法均标记为虚函数，其中析构函数标记为虚函数用于子结点重载内存释放逻辑。**NodeAST** 的 `codegen` 方法用于语义分析，执行后可得到当前结点的 **LLVM IR** 代码，将在语义分析详细设计中详细叙述。**NodeAST** 的 `toString` 方法用于描述当前结点的数据，用于传入日志或调试系统，方便打印日志。

代码 3.8 抽象语法树结点接口父类

```
class NodeAST {
public :
virtual ~NodeAST() = default; // 析构函数
virtual llvm::Value *codegen() = 0; // 生成 LLVM Value
virtual string toString () = 0; // 结点名称输出};
```

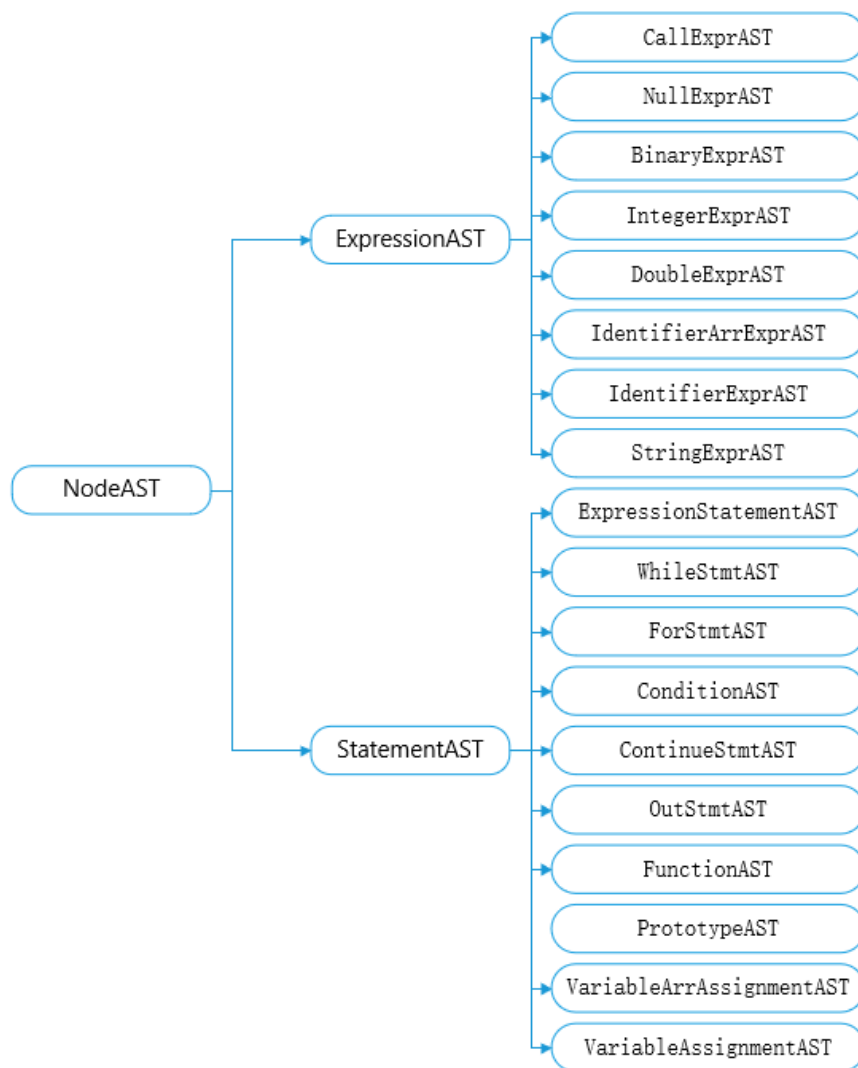


图 3.4 抽象语法树结点类关系

StatementAST、ExpressionAST 均由 NodeAST 派生而来，是抽象语法树结点的两大类型。StatementAST 为声明结点，如变量、函数、代码声明等。ExpressionAST 为表达式结点，如一个常量数字、字符串、数组等。同时，一个表达式也可单独单做一个声明，如语句“a;”，为一个无意义但符合语法的声明。

④ 定义分析起点。Bison 使用 %start 标识语法解析的起点。如代码 3.9 所示，此处表示从非终结符“program”开始进行规约。

代码 3.9 分析起点

```
%start program
```

⑤ 定义非终结符与类关系。编译器为每一个非终结符定义一个相应的数据类，使用 %type 进行声明。具体如下：

```
%type <block> program block stmts
%type <ident> ident
%type <identarr> ident_arr
%type <expr> number expr str
%type <cond> if_condition
%type <forexpr> for_stmt
%type <whilestmt> while_stmt
%type <node> for_args
%type <stmt> stmt func_decl
%type <vdecl> var_decl
%type <varvec> func_args
%type <identvec> func_ptr_args
%type <exprvec> call_args
%type <arrayvalvec> array_init_val array_init_list
%type <aivec> array_index
```

每行条目的格式为“%type <C++ 类型变量名> 非终结符名称...”，用于表示非终结符对应的 C++ 类型变量名。C++ 类型变量名说明如下：

- 1) block。对应 BlockAST，其是 {...} 域中代码的集合。
- 2) ident。对应 IdentifierExprAST，用于存放一个标识符。
- 3) identarr。对应 IdentifierArrExprAST，用于存放一个带数组索引的标识符。
- 4) expr。对应 ExprAST，用于存放表达式。其为父类，适合存放可能有多种子类的数据。
- 5) cond。对应 ConditionAST，用于存放条件跳转结点。
- 6) forexpr。对应 ForStmtAST，用于存放类 for 循环结点。
- 7) whilestmt。对应 WhileStmtAST，用于存放类 while 循环结点。
- 8) node。对应 NodeAST。其为所有结点的基类，用于存放可为声明和表达式的结点数据。
- 9) stmt。对应 StatementAST，用于存放声明。其为父类，适合存放可能有多种子类的数据。
- 10) vdecl。对应 VariableDeclarationAST，用于存放一个变量声明。
- 11) varvec。对应 VariableDeclarationAST 数组，用于存放一系列变量声明。
- 12) exprvec。对应 ExpressionAST 数组，用于存放表达式数组。
- 13) arrayvalvec。对应 NodeAST 数组，专用于存放所有数组索引。

14) aivec (array index vector)。对应 ExpressionAST 数组，专用于存放常量数组索引。

接下来按照 WebC 语法定义逐条进行语义分析的实现。

1) program。程序由一系列声明组成。当开始规约 program 时，证明用户输入合法。参考代码 3.10 可知，program 和 stmts 同属于 block，可直接进行将 stmts 赋值给 program，完成语义分析。

2) stmts。声明可以有多个。此处采用 vector 可变长数组进行装填。装填完毕后赋值给 stmts。

3) stmt。声明可以为变量声明。stmt 对应的类型为 StatementAST，为父类结点。其可接受子类传参。子类类型见图 3.4。

4) ident。当读取到一个以字母为首的字符串，且后方无数组索引“[]”，将其赋值为 IdentifierExprAST。

5) ident_arr。当读取到一个以字母为首的字符串，后跟随索引“[]”，将其赋值为 IdentifierArrExprAST。

6) var_decl。当读取到变量类型与变量名时，将其赋值给 VariableDeclarationAST，若后方在声明时还给出了初始值，则将初始值传入构造函数。此处要求 VariableDeclarationAST 拥有初始值逻辑，即在未传入时使用默认初始值，默认初始值如表 3.2 所示。

表 3.2 默认初始值列表

WebC 数据类型	初始值
int	0 (32 位)
long	0 (64 位)
short	0 (16 位)
bool	0 (1 位)
str	null

7) func_ptr_args。此项标识函数指针的形参表，由标识符组成。由于形参个数可变，故使用 vector 可变长数组存放形参。

8) expr。此项规约所有的二元表达式、字符串、null、数字、数组等值。

9) array_index。为所有数组索引的总非终结符。如：a[5][6] 中“[5][6]”规约为 array_index，a[5]中的“[5]”规约为 array_index，a[][5] 中的“[][5]”规约为 array_index。[]内可以为空值，因为函数形参可以如此声明。

10) call_args。此项可表示零个或多个用逗号连接起来的函数调用参数。编译器递归的进行规约，若发现后方仍然含有逗号和一个表达式，则将后方的表达式推入可变长数组中，直

到规约完毕。对于零个函数调用参数，此处最终生成长度为零的表达式数组。

11) **number**。此项用于规约数字。数字分为整数与浮点数。词法分析阶段已将整数与浮点数进行区分，分别赋值给 **DoubleExprAST** 或 **IntegerExprAST**。

12) **block**。此项用于规约 {} 划定的代码域。代码域由零至多条 **stmt** 组成。**stmt** 使用可变长数组装入，传入 **block** 构造函数。

13) **func_decl**。此项用于规约函数定义。解析分为两步，分别为解析函数信息与解析函数体。函数信息包括：返回值、函数名、函数形参列表。函数体即为一个 **block**，按 **block** 解析得到 **stmt** 数组即可。

14) **func_args**。类似于 **call_args**，此项表示零个或多个用逗号连接起来的函数形参。编译器递归的进行规约，若发现后方仍然含有逗号和一个表达式，则将后方的表达式推入可变长数组中，直到规约完毕。对于零个函数调用参数，此处最终生成长度为零的表达式数组。

15) **if_condition**。用于匹配 **if** 条件语句。其中，**else** 遵循可选匹配、最近匹配。若无法匹配到 **else**，则表示 **else** 分支为空，并传入 **nullptr** 告知 **if** 语句结点。**if** 含有 4 个基础块，分别为判断块，判断为真块，判断为假块，判断结束块。判断块由 **if** 内表达式构造，判断为真块由 **if** 后的 **block** 进行构造，判断为假块由 **else** 后的 **block** 提供，若 **else** 后的 **block** 不存在，则直接跳转到判断结束块。

16) **for_stmt**。用于声明 **for** 循环声明。其含有：初始块，判断条件，步进块，循环块与循环结束块。每个块传入 **ForExprAST** 的构造函数。

17) **for_args**。用于匹配 **for** 括号内的表达式。表达式接受 3 种类型的传参，包括变量声明，变量赋值以及一个表达式。**for_args** 为 **NodeAST**，为表达式和声明结点的父类，故可以将匹配后将参数直接赋值给 **for_args**。

18) **while_stmt**。用于匹配 **while** 循环声明。其接收 **expr** 与 **block** 作为构造参数。其中 **expr** 为判断表达式，**block** 为循环块。

19) **array_init_val**。其代表一个常量数组内的元素。{{1,2,3,4},{1,123,23,3}}是一个常量数组元素，{}也是一个常量数组元素，1也是一个常量数组元素。综上，其元素可为：表达式、{}包裹的常量数组以及无内部元素的大括号 {}。

20) **array_init_list**。其代表一个常量数组。其由 **array_init_val** 组成，使用逗号隔开。编译器递归的将每个元素插入数组，最终作为整体传入构造函数。

21) **str**。其从词法接收一个字符串，作为函数构造参数传入 **StringExprAST**。

(3) 语义分析

语义分析器获取到的语法分析中得到的抽象语法树，进行 LLVM IR 代码的生成。其中关键的，WebC 语言编译器在语义中要用大量数据类型查询，WebC 编译器将大量常用类型的获取封装到函数中，如代码 3.10 所示：

函数输入为类型字符串，输出为 LLVM 类型（Type）。其采用轮询的形式，获取相应的类型，并调用 Type 下的 getXXTy 方法获取相应符合的类型。方法需要传入 LLVM 模块上下文，由基础层的 LLVM IR 代码读写系统提供。

接下来按照图 3.4 中的每个结点进行语义实现。

(1) CallExprAST。其处理逻辑如图 3.5 所示。首先处理结点描述，包括调用函数名，调用参数列表。接下来按照函数定义、函数形参、外部定义的顺序进行调用猜测。若用户在当前代码中定义过当前函数，则直接使用用户定义的函数。若未定义过，则查看当前函数的形参表中是否有符合名称的函数指针，若有则使用函数指针，否则查看是否在外部库中可以处理该调用。若可以在外部库处理，则将函数信息填充至结点进行 LLVM IR 调用生成，否则报错，告知基础层的编译上下文系统停止编译。外部库分为编译器内部模块，外部模块。如 WebC 语言的 Web 库等；外部模块由用户加入，属于用户自定义库。

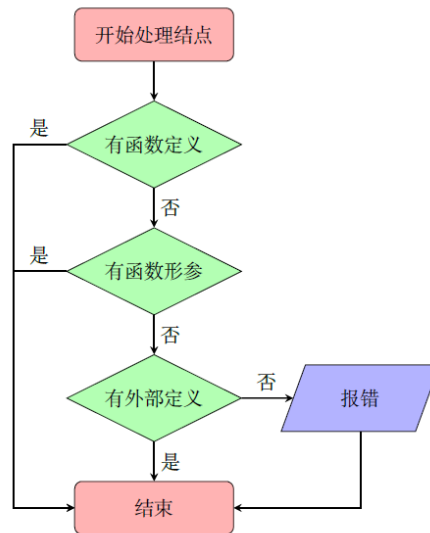


图 3.5 函数调用解析逻辑

代码 3.10 常用类型定义

```
Type *getTypeFromStr(const std::string &type) {  
    if (type == "int") {  
        return Type::getInt32Ty(*TheContext);  
    } else if (type == "long") {
```

```

return Type::getInt64Ty(*TheContext);
} else if (type == "short") {
return Type::getInt8Ty(*TheContext);
} else if (type == "bool") {
return Type::getInt1Ty(*TheContext); // 本质上就是 1 字节的 int
} else if (type == "void") {
return Type::getVoidTy(*TheContext);
} else if (type == "double") {
return Type::getDoubleTy(*TheContext);
} else if (type == "float") {
return Type::getFloatTy(*TheContext);
} else if (type == "char") {
return Type::getInt8Ty(*TheContext);
} else if (type == "str") {
return Type::getInt8Ty(*TheContext)->getPointerTo(); // int8 的指针
}
return NIL;
}

```

(2) `NullExprAST`。表示为 `null` 值。由于在 `WebC` 中只有在字符串声明中允许使用空值，故使用 `ConstantExpr::getNullValue(getTypeFromStr("char")>getPointerTo())` 获取空值，空值类型为 `char` 类型指针。

(3) `BinaryExprAST`。其为二元表达式。二元操作分为两大类，分别为逻辑操作、算术操作。先判断是否是逻辑操作，如与或操作。若是逻辑操作则按条件分支的形式进行解析，否则按算术操作解析。对于逻辑操作，其类似于 `IfStmtAST`，可提供参考。对于算术操作，首先解析二者表达式的 `LLVM Value` 值。若解析失败，返回空值，中止语义生成。否则进行二者类型判断。如二者并不是相同类型，则尝试扩大或损失精度使二者能进行二元操作，最后进行赋值。若在赋值时也遇到类型不匹配问题，也尝试扩大或损失精度。

(4) `IntegerExprAST`。用于存放常量数字的结点。通过向基础层的 `LLVM IR` 代码读写系统获取常量表的形式拿到整型对应的 `LLVM Value`。由于 32 位数字较为常用，故默认为 32 位。若赋值给位数较多的变量，则进行位扩展。相关代码如下所示。

代码 3.11 常量表获取 32 位长度的值

```

ConstantInt::get (Type::getInt32Ty(*TheContext), Val, true );

```

`getInt32Ty` 需要传入 `LLVM` 上下文变量、变量值以及是否有符号。此处默认为有符号 32 位整数。

(5) `DoubleExprAST`。用于存放浮点数字的结点。`LLVM` 中的 `APFloat` 可以保持任意精

度的数值。故使用其保存 WebC 语言的结点。相关构造代码如代码 3.12 所示。

代码 3.12 浮点数常量获取

```
llvm :: ConstantFP:: get(*TheContext, llvm :: APFloat(Val)) ;
```

(6) IdentifierExprAST。其中标识符解析逻辑如下图所示。一个标识符在 WebC 语言中，可能为本地变量、局部变量与函数指针。在优先级中，先查看是否是本地变量。其中，由于域（block）支持嵌套，故本地变量由各个域叠加组成。此处需要在栈结构中读取每层的本地变量进行查找。若未查找到，则查看是否在全局变量声明中，若有则将全局变量通过 Load 指令从内存中读取到 LLVM 虚拟寄存器中。若全局变量中没有该名称的变量，则判断其是否是函数名称，若是则尝试将其为按函数指针处理。若中无标识符名称命名的函数，则通知基础层的编译上下文系统错，停止编译。

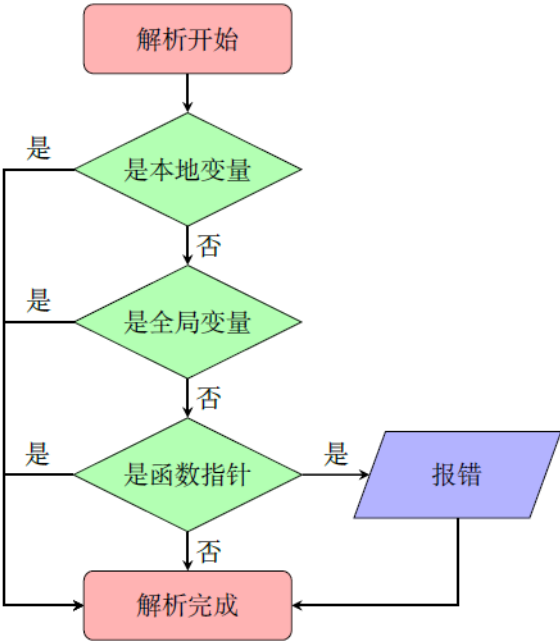


图 3.6 标识符解析流程

(7) IdentifierArrExprAST。类似与 IdentifierExprAST，但不判断是否是函数指针。若在本地图变量与全局变量中都未找到，则直接报错。

(8) StringExprAST。该结点通过查询基础层的 LLVM IR 代码读写系统，从常量表中获取相应的字符串的表达式。注意，此处需要判断是否在全局还是局部环境中。若在全局环境中，则使用 GlobalVariable 进行初始化，如代码 3.13 所示。

代码 3.13 全局变量定义

```
gv = new GlobalVariable(*TheModule, arr_type, true, GlobalValue::PrivateLinkage,
ConstantDataArray::getString(*TheContext, *str), getUniqueId());
```

字符串事实上就是连续的字符数组、其中，`arr_type` 表示字符数组类型，通过 `ArrayType::get(getTypeFromStr("char"), sz + 1)`；进行初始化。其中 `sz` 表示字符串长度大小，声明时在长度处加上 1，即预留 `\0` 字符串后缀。`true` 标识该字符数组为常量。`PrivateLinkage` 标识该字符串在链接时不对其他模块可见。`ConstantDataArray::getString` 用于从常量表中拿到对应字符串的常量数据。`getUniqueId` 函数用于获取字符串的唯一标识。创建完成后，使用 `bitcast` 指令将字符串指针类型转为与 C 中等效的 `char*` 类型。

若在局部环境中，由于字符串声明是只读、不可更改的，其同样可以放在全局变量中。由于在局部环境中，LLVM IR 代码读写系统可以拿到函数以及模块信息，故可以使用代码 3.14 的方法进行声明。

代码 3.14 局部变量声明

```
Builder->CreateGlobalString(*str, StringExprAST::getUniqueId());
```

通过该方法进行声明的全局变量是一个指向字符串数组内存的指针。而在使用时需要拿到第一个元素所在地址。故需要使用 `GEP` 指令（`GetElementPointer`），首先拿到字符串数组所在内存（LLVM 将取内存定义为取第 0 号元素），再在该基础上拿到第 0 号元素所在空间。语法如代码 3.15 所示。

代码 3.15 局部数组元素

```
Builder->CreateInBoundsGEP(gv, {ConstantInt::get(getTypeFromStr("int"), 0), ConstantInt::get(getTypeFromStr("int"), 0)});
```

(9) `ExpressionStatementAST`。此针对一个表达式作为声明的情况。通常是无意义的。如“`a;`”、“`1+2;`”。为了保证兼容性和未来的拓展性，WebC 语言允许将单表达式直接作为声明的情况。LLVM IR 代码直接由表达式提供。

(10) `WhileStmtAST`。其含有条件判断结点、循环结束结点、循环结点。每个结点使用一个基础块代替。其解析顺序如图 3.7 所示。图中，每个块都是一个基础块。每一条边都为一条单分支跳转指令。当循环结点中出现 `break`、`continue`（WebC 语言中定义为 `out`、`cont`，此处为符合大众习惯讲解）结点时，会出现循环结点到循环退出的边。

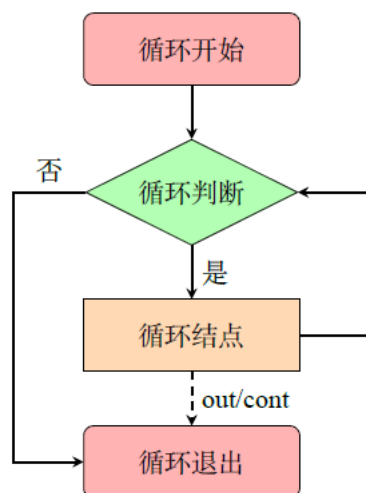


图 3.7 循环 while 结点流程

(11) ForStmtAST。其含有条件判断结点、循环结束结点、循环结点。具体流程如图 3.8 所示。

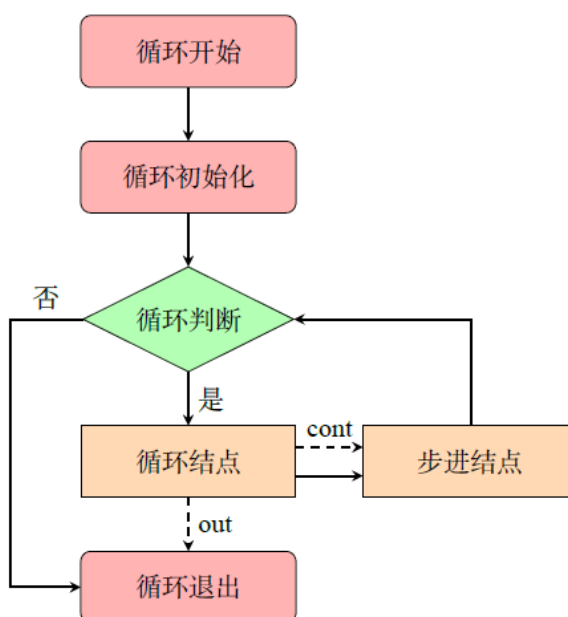


图 3.8 循环 for 结点流程

for 相比 while 增加了步进结点与循环初始化结点。同时在使用 cont 重新开始循环时，跳入步进结点。相比 while 来说，跳入的结点不同。故在解析 cont 时应该被告知当前环境。

(12) ConditionAST。用于 if 分支的解析。相关流程如图 3.9 所示。先分析条件判断语句，之后分配 2 个基础块，并将条件判断条件转入其中。2 个基础块一个存放真结点指令，另一个存放假结点指令。在解析后共同进入条件分支结束结点。注意，若出现条件判断嵌套的情

况，需要记住当前解析条件分支的结束基础块位置，以防嵌套出现问题。

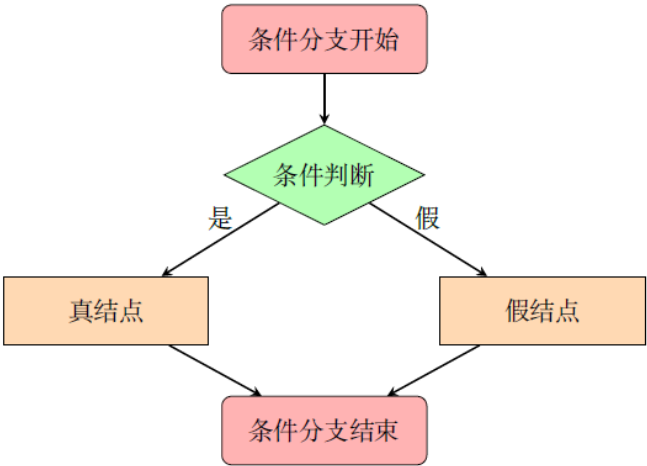


图 3.9 判断 if 结点流程

(13) ContinueStmtAST。用于重新开始一次循环，但对于不同环境需要跳转到不同的基础块。循环含有 for 与 while 两种类型。如图 3.7 和图 3.8，对于 for 环境，cont 需要跳转到步进结点；对于 while 环境，cont 需要跳转到循环判断结点。相关逻辑如代码 3.16 所示。对于 for 环境，传入步进结点，即 bbStep；对于 while 环境，传入条件判断结点，即 bbCond。

代码 3.16 不同循环情景下的跳转问题

```
case FOR:
return Builder->CreateBr(blk->context.forCodeGenBlockContext->bbStep);
case WHILE:
return Builder->CreateBr(blk->context.whileCodeGenBlockContext->bbCond);
```

其中，blk → context 指向 CodeGenBlockContext，其定义如代码 3.17 所示。

代码 3.17 代码块上下文定义

```
union CodeGenBlockContext {
  ForCodeGenBlockContext *forCodeGenBlockContext;
  IfCodeGenBlockContext *ifCodeGenBlockContext;
  WhileCodeGenBlockContext *whileCodeGenBlockContext;};
```

CodeGenBlockContext 是一个 union 类型，被 for、if、while 三者环境配置信息共用。故在不同环境下，CodeGenBlockContext 代表这不同的上下文信息，从而可以拿到当前环境所有的基础块，并执行操作。

(14) OutStmtAST。用于跳出循环体。对于 for、while 环境，都跳出至循环结束结点。相关逻辑如代码 3.18 所示。对于 for 环境，跳出到 bbEndFor 结点；对于 while 环境，跳出到 bbEndWhile 结点。

代码 3.18 跳出循环体代码

```
case FOR:
return Builder->CreateBr(blk->context.forCodeGenBlockContext->bbEndFor);
case WHILE:
return Builder->CreateBr(blk->context.whileCodeGenBlockContext->bbEndWhile);
```

(15) PrototypeAST。用于描述函数类型，是 FunctionAST 结点的子结点。其构造 FuncFunctionType 的逻辑如代码 3.19 所示。

代码 3.19 函数类型信息构建

```
llvm :: FunctionType *FT = llvm :: FunctionType:: get (getTypeFromStr(returnType) , Args,
false ) ;;
llvm :: Function *F = llvm :: Function :: Create(FT, llvm :: Function :: ExternalLinkage ,
getName(),
TheModule.get());
```

结点获取到返回类型 returnType，形参列表 Args 后即可构建函数类型，并通过该类型定义 LLVM IR 的函数。函数默认可对外链接。

(16) FunctionAST。用于定义整个函数。在 PrototypeAST 解析完后，开始解析函数体。整体流程如图 3.10 所示。首先针对函数的形参表、函数返回值分配内存。之后，向基础层的编译上下文系统申请当前函数的上下文，本地变量存储空间。在做好准备工作后，开始解析并校验函数体。若用户没有显式的书写 return 语句，则在解析时补充 return 语句，并返回类型的默认值。若函数解析失败，则清除当前函数的上下文，并告知终止编译。

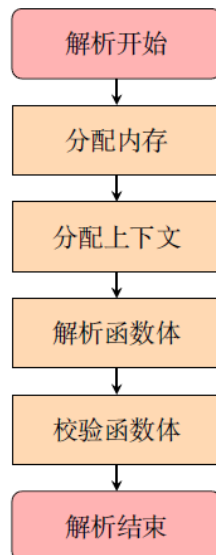


图 3.10 函数结点解析流程

(17) VariableAssignmentAST。用于变量的赋值。先依次从本地变量、局部变量中查找相

关变量。若未找到则提示变量不存在，终止编译。否则将要赋值的值通过 `Store` 命令写入相关变量地址。

(18) `VariableArrAssignmentAST`。用于数组变量的赋值。解析逻辑类似与 `VariableAssignmentAST`，但对于数组赋值，可以使用 LLVM 的 `memcpy` 函数对内存块进行复制，相较逐元素赋值大大提高了效率。

(4) 代码优化

代码优化器分为内置 LLVM 优化器与 WebC 语言实现的优化器。

内置优化器在 WebC 语言中使用如代码 3.20 所示，其详细实现见 LLVM 文档。

内置遍优化含有：

(1) Tail Call 优化。针对以函数调用返回值为返回值的指令，如“`return func()`”，可以将函数调用转化为跳转，复用函数栈。

(2) 关联性优化。可以在编译期提前计算相关值。

(3) CFG 优化。通过建立执行流图，寻找不可达结点，合并基础块等。

代码 3.20 使用 LLVM 内置优化器

```
pass.add( createTailCallEliminationPass() );
pass.add( createReassociatePass() );
pass.add( createCFGSimplificationPass() );
```

另外，WebC 语言为每一个 WebC 函数提供了函数计时功能，可以计算函数执行时间，如图 3.11 所示。优化时可以获取到所有已创建的基础块列表，通过找到首部基础块，定位 LLVM IR 读写位置到该基础块的第一条指令，调用 WebC 语言的 `ktime` 库得到当前时间。再找到尾部结点，定位 LLVM IR 读写位置到该基础块的最后一条指令（通常是 `return`），在最后一条指令之前计算差值，并调用系统 `libc` 库提供的 `printf` 函数进行输出。WebC 提供 `ExternFunctionHandler` 对外部函数调用进行描述。如创建 `printf` 时，需要调用代码 3.22 所示的获取到的 `printf` 对象示例。

代码 3.21 加载 `printf` 函数

```
ExternFunctionHandler :: getOrAddPrintfFunc(*TheContext, *TheModule)
```

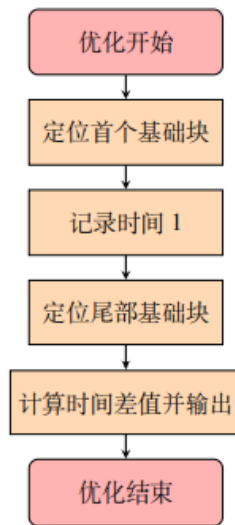


图 3.11 时间分析插桩流程

由于获取后会对当前模块加入 `printf` 函数定义，所以所有的外部函数均采用懒加载的形式加载。若当前函数环境中存在 `printf` 定义则直接复用，否则进行创建。以 `printf` 为例，创建该函数的逻辑如代码 3.22 所示。

代码 3.22 添加 `printf` 函数实现

```

Function *ExternFunctionHandler :: getOrAddPrintfFunc(LLVMContext &context, Module
&module) {
    auto funcs = module.functions();
    auto it = funcs.begin();
    for (; it != funcs.end(); it++) {
        if ((*it).getName() == "printf") {
            return &(*it);
        }
    }
    FunctionType *ty = FunctionType::get(Type::getInt32Ty(context), {Type::getInt8PtrTy(
context
)}, true);
    auto func = Function::Create(ty, llvm::GlobalValue::ExternalLinkage, "printf", module);
    return func;
}
  
```

`printf` 在系统 `libc` 库中的定义为 `int printf(const char *__restrict __format, ...)`。故代码中，在创建 `FunctionType` 时，同样按照相应返回值与形参列表进行创建。代码中，`getInt32` 函数获取 `int` 返回值，`getInt8PtrTy` 获取 `char*` 类型参数，标记为 `true` 表示后方有自定义参数。

(5) 目标文件生成

目标文件即用户最后想得到的文件。根据用户不同的需要，可以生成目标架构下的汇编代码、二进制可执行文件、可链接的目标文件。其生成顺序如图 3.12 所示：

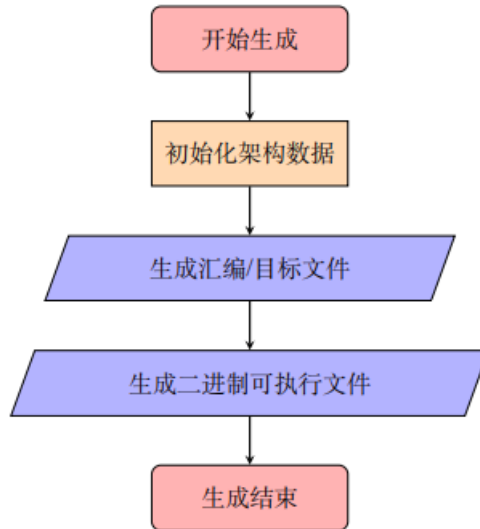


图 3.12 目标代码生成流程图

① 初始化架构数据。编译器默认对宿主处理器架构的编译进行支持。CMake 可在编译配置时期进行架构检测，并根据不同的架构添加不同的宏定义。相关 CMake 代码如代码 3.23 所示。CMAKE_HOST_SYSTEM_PROCESSOR 为 CMake 环境变量，存储当前系统的架构。故可以通过条件语句，并使用 `add_compile_definition` 命令添加宏定义。

代码 3.23 CMake 系统架构配置

```
if(CMAKE_HOST_SYSTEM_PROCESSOR MATCHES "aarch64")
add_compile_definitions ( sysyplus_compiler PUBLIC SYSY_AARCH64=0)
elseif(CMAKE_HOST_SYSTEM_PROCESSOR MATCHES "arm")
add_compile_definitions ( sysyplus_compiler PUBLIC SYSY_ARM=0)
elseif(CMAKE_HOST_SYSTEM_PROCESSOR MATCHES "mips64el")
add_compile_definitions ( sysyplus_compiler PUBLIC SYSY_MIPS64EL=0)
else ()
add_compile_definitions ( sysyplus_compiler PUBLIC SYSY_X86=0)
endif ()
```

在初始化中，可以根据宏定义动态初始化 LLVM 相应后端。如代码 3.24 所示。

代码 3.24 初始化 LLVM 目标后端部分代码

```
# ifdef SYSY_ARM
LLVMInitializeARMTARGETInfo();
LLVMInitializeARMTarget();
LLVMInitializeARMTargetMC();
```

```

LLVMInitializeARMAsmParser();
LLVMInitializeARMAsmPrinter();
#endif
#ifdef SYSY_AARCH64
LLVMInitializeAArch64TargetInfo();
...
#else

```

初始化完成后，获取系统三元组描述，传入 LLVM，获得支持系统架构的 LLVM 后端实例 TheTargetMachine。

② 生成汇编/目标文件。根据想要的文件格式，传入对应的 file_type 变量即可。如代码 3.25 所示。对于其余的参数，pass 即代码优化中定义的优化器，dest 即目标输出文件流，nullptr 表示不进行代码检测。若生成出现错误，可进行 LogError 函数进行错误输出。

代码 3.25 生成汇编、目标文件关键代码

```

if (TheTargetMachine->addPassesToEmitFile(pass, dest, nullptr, file_type) ) {
    LogError("TheTargetMachine can't emit a file of this type");
    return 1;}

```

③ 生成二进制可执行文件。生成可执行文件需要使用目标系统的开发套件。Windows 为 Visual C++，Linux 为平台兼容的 GCC、Clang，macOS 为 XCode 编译套件。其中，Clang 是 XCode 默认内置的编译器，且兼容 Linux、Windows，相比 GCC 具有编译速度快，占用内存小等特点。WebC 语言编译器使用 Clang 前端进行可执行二进制文件的生成。首先对用户系统环境变量进行扫描，若未发现 Clang 位置，则证明用户未安装 Clang，警告用户需要安装缺失组件。重点逻辑如代码 3.26 所示。

代码 3.26 Clang 编译调用

```

IntrusiveRefCntPtr <clang :: DiagnosticIDs> DiagID(new DiagnosticIDs());
DiagnosticsEngine diag_engine(DiagID, new DiagnosticOptions());
driver :: Driver driver ( args [0], sys :: getDefaultTargetTriple(), diag_engine);
auto webc_compilation = driver.BuildCompilation( args );
int compile_res = driver.ExecuteCompilation(*webc_compilation, failingCommands);
if (compile_res < 0) {
    driver.generateCompilationDiagnostics (*webc_compilation, *failingCommand);
    return RERR;
}
LogInfo("编译完成");

```

代码中，Driver 为 Clang 编译的主类。在编译时，Clang 要求在初始化 Driver 时提供可

供诊断信息输出的引擎（DiagnosticsEngine）。在构造完 Driver 后，调用 ExecuteCompilation 即可。若返回值为 0，证明编译完成，否则进行诊断信息输出。

3.2.2 库模块详细设计

（1）kweb 模块

kweb 是 WebC 语言的重要网络库，其用于给 WebC 语言提供网络客户端、服务器功能。其含有的接口如代码 3.27 所示。接口函数统一以 _web_ 为前缀，为 web 库的预留标识。

① init。为了保证最小的内存占用。WebC 库按需进行初始化。该函数不需要用户显式调用，由内部按需调用。

② getSocket。由用户调用，向 WebC 库申请套接字，并向用户返回套接字 ID。

③ connectSocket。由用户调用，通过套接字连接对应的服务器端口。

④ closeSocket。由用户调用，通过提供的套接字 ID，关闭对应的套接字。成功返回 ROK，若无该 ID 对应的套接字，则返回 SOCKET_NOT_EXISTS。

⑤ isSocketConnected。由用户调用，若相关套接字正处于连接状态则返回 ROK，否则返回 NOT_CONNECTED。

代码 3.27 模块 kweb 接口描述

```
int _web_init();
int _web_getSocket();
int _web_connectSocket(int socketId, const char *baseUrlOrIp, const char *port);
int _web_closeSocket(int socketId);
int _web_isSocketConnected(int socketId);
const char *_web_callGetRequest(int socketId, char *host, char *path);
const char *_web_callPostRequest(int socketId, char *host, char *path, char *body);
int _web_getServerId(const char *addr, int port, int core);
int _web_addUrlHandler(int sId, const char *method, const char *path, const char *content_type,
const char *handler());
int _web_startServe(int sId);
```

⑥ callGetRequest。用于发送 GET 请求，可向其中提供套接字 ID，主机名以及 URL。返回数据使用 char*数组。

⑦ callPostRequest。用于发送 POST 请求，可向其中提供套接字 ID，主机名以及 URL、Body。返回数据使用 char*数组。

⑧ getServerId。用于获取一个可用的服务器 ID。其中可配置服务器的监听地址、端口。特别的，为了方便用户更方便的使用多线程技术，可以通过 core 传入需要开启的线程数。

线程数越大，性能越好。

⑨ `addUrlHandler`。服务器配置接口，用于增加 URL 拦截器，当用户使用 `method` 方法访问 `path` 路径，并告知要返回 `content_type` 类型的数据时，执行 `handler` 函数指针指向的函数获取常量 `char*`数据并返回给用户。

⑩ `startServe`。用于启动对应 ID 的服务器。通常用于配置完成后，启动服务器。调用后主线程出于阻塞状态。

(2) `kjson` 模块 `kjson` 为重要的 `json` 数据序列化、反序列化库。其接口定义如代码 3.28 所示：

代码 3.28 模块 `kjson` 接口定义

```
struct JsonData {
std :: shared_ptr <boost :: property_tree :: ptree > pt ;
};
typedef JsonData *SYSY_JSON_DATA;
typedef std :: string SYSY_STR;
extern "C" {
SYSY_JSON_DATA strToJson(SYSY_STR str);
SYSY_STR jsonToStr(SYSY_JSON_DATA json);}
```

为了保证拓展性，`kjson` 模块使用自定义结构 `JsonData` 封装 Boost 库的 `property_tree`，保证后期增加、删除结构体内数据类型时可以不改动接口。`property_tree` 是 Boost 内提供的 `xml`、`json` 的解析器，具有良好的兼容性。接口定义 `typedef` 封装数据格式。`strToJson` 和 `jsonToStr` 两个函数共同提供了 `json` 类型与字符串数据类型的相互转换。

将字符串转化为 `json` 类型时，使用 `property_tree` 库提供的解析方法进行解析。若 `json` 出现问题（如非 `json` 类型等情况）则返回 `nullptr`，并在日志系统报错。当 `json` 类型需要转化为字符串时，则使用 `property_tree` 提供的 `write_json` 函数进行转化。

(3) `ktime` 模块

`ktime` 模块用于向 WebC 语言提供时间支持。但各系统上对于时间的获取不尽相同，故需要针对不同的系统采用不同的函数实现。其函数接口如代码 3.29 所示：

代码 3.29 模块 `ktime` 接口描述

```
/// 获取 ms 数，时间戳
long __getms();
```

对于 Windows 系统，其在 `Windows.h` 中提供了 `SYSTEMTIME` 数据结构、`GetLocal-Time` 函数获取时间，故可将二者结合起来，计算出当前的时间戳。对于 Unix 系的系统，其在 `sys/time.h` 中定义了 `gettimeofday` 函数，`timeval` 数据结构，结合起来也可

以实现获取时间戳功能。

CMake 在编译时提供了以操作系统名为名称的宏，故可以使用`#ifdef` 进行条件编译，针对不同系统使用不同实现。

(4) ksql 模块

ksql 模块用于给 WebC 语言提供 MySQL 支持。其拥有的接口如代码 3.30 所示：

代码 3.30 模块 ksql 接口描述

```
/// 连接到 mysql
int _ksql_connect_db(const char *host , const char *user , const char *passwd, const char
*database , const char* port );
/// 释放资源
int _ksql_free_memory();
/// 查询数据
const char * _ksql_query_db(const char *sqlSentence );
```

① `connect_db`。用于连接一个 MySQL 实例。其为连接提供了基础的配置参数，如主机名，用户名，密码，数据库名以及端口。

② `free_memory`。用于释放连接的 MySQL 实例。

③ `query_db`。用于执行 MySQL 命令。并把执行结果以常量字符串的形式返回。

`libmysqlcppconn` 库是 MySQL 组织开发的，是 C/C++连接 MySQL 的支持库。`ksql` 库封装了对 `libmysqlcppconn` 的调用，简化调用。

(5) kstring 模块

`kstring` 为 WebC 数据类型提供了转字符串功能。其接口描述如代码 3.31 所示。

代码 3.31 模块 kstring 接口描述

```
typedef int sysytype_t ;
#define SYSYTYPE_INT 0
#define SYSYTYPE_LONG 1
#define SYSYTYPE_DOUBLE 2
const char * toString (void *addr, sysytype_t type);
```

`toString` 函数将 `addr` 上的指定数据类型的数据转化为常量字符数组。其中 `sysytype_t` 可为整形与浮点型数字。

`toString` 函数封装了 `std::to_string` 函数。该标准函数是 C++ 11 版本加入的，用于各种 C++基础数据类型转化为字符串。

3.3 用户接口层详细设计

用户接口层抽象了模块层的实现，给上层开发提供了统一接口，简化调用，透明化编程。

编译器用户接口层对编译器模块层进行封装。编译器用户接口层分为针对开发者、集成开发环境（Integrated Development Environment，IDE）设计。

3.3.1 代码分析接口

代码分析接口主要被集成开发环境与命令行环境使用。在集成开发环境中，用于实时分析用户代码，在命令行环境中用于代码生成。其接口定义如代码 3.32 所示：

代码 3.32 代码分析接口

```
int analysis ( std :: string * buf ) ;  
int startAnalyze ( ArgsParser* parser ) ;
```

analysis 函数接收一个 string 指针，通过基础层提供的 StringReader 读入词法、语法分析模块进行分析。分析结果分为两部分。总体分析结果会直接通过函数的返回值返回，若出现错误则返回 RERR，否则返回 ROK。分析日志则通过 logOnUi 函数向界面打印分析日志。

startAnalyze 函数用于接收一系列命令行参数，通过用户传入的文件路径，编译选项进行分析，得出抽象语法树结构

3.3.2 编译接口

编译分为命令行编译和集成开发环境编译，编译接口的定义如代码 3.33 所示：

代码 3.33 编译接口定义

```
int genCode(const set <ArgsParser :: Options>& opts, const char *outputPath ) ;  
int build ( std :: string *buf, const char* outputPath , const std :: set <ArgsParser :: Options>&  
opts);
```

genCode 函数用于执行完 startAnalyze 分析函数后，立即执行，将抽象语法树转化为目标文件，输出至 outputPath 处。

build 函数用于集成环境将当前读取的代码缓冲区传入，并执行编译。

ArgsParser::Options 可用于设置目标文件类型，如输出汇编代码、编译目标文件、可执行二进制文件。

3.4 IDE GUI 层详细设计

集成开发环境使用 Glade 进行界面设计，使用 GTK+3 进行渲染。

3.4.1 IDE 菜单

在 Glade 中，使用 GtkMenuBar 声明一个菜单，并添加 GtkMenuItem，向内填入 GtkMenu，实现菜单界面。如图 3.13 所示。由图中可见，GtkMenuBar 中含有 3 个 GtkMenuItem，分别为“文件”、“构建”、“帮助”。文件选项卡下含有新文件、打开文件、保存文件、另存为、退

出等常用按钮。构建选项卡下含有编译为编译目标文件、汇编文件、仅编译以及执行按钮。帮助下含有关于按钮。



图 3.13 WebC 集成开发环境菜单设置

3.4.2 IDE 代码编辑器

代码编辑器使用 GtkSourceView 进行构建。但为了适配 WebC 语言，还需要对其进行定制。部分重要配置如代码 3.34 所示。

代码 3.34 代码编辑器配置

```
// 代码提示
m_completion_words = Gsv::CompletionWords::create("代码提示",RefPtr<Gdk::Pixbuf>());
m_completion_words->register_provider(m_tip_buffer);
m_gsv->get_completion()->add_provider(m_completion_words);
// 高亮设置
m_gsv->set_highlight_current_line ( true );
m_lm = Gsv::LanguageManager::get_default();
auto lan = m_lm->get_language("c");
m_gsv->get_source_buffer()->set_language(lan);
m_gsv->get_source_buffer()->set_highlight_syntax ( true );
```

通过一个 TextBuffer 向 CompletionWords 提供代码提示关键词，并将代码编辑器设置为 C 语言，并设置为语法高亮。

3.4.3 IDE 多线程模块管理

为了保证界面的流畅性，集成开发环境需要将 UI 与逻辑代码进行线程隔离。集成开发环境为其提供了大小为 3 的线程池。用于守护线程，编译线程与一个运行监测线程。

```
class CompilerWindow : public Gtk::ApplicationWindow {
...
boost :: asio :: thread_pool threads {3}; // 1 个守护线程，1 个编译线程，一个运行监测线程
...
}
```

(1) 守护线程。用于实时监测代码编译器内的代码变化。若代码发生变化，且 1 秒内没有任何操作，则调用用户接口层的代码分析接口进行代码分析。若含有分析日志，则在 UI 线程空闲时（接收到 `SIGNAL_IDLE` 信号）将日志输出。

(2) 编译线程。用于用户程序编译。用户执行编译操作时，使得集成开发环境进入编译状态。

(3) 运行监测线程。用于执行编译产物，并将编译产物的执行输出重定向至集成开发环境日志接口。

第四章 测试报告

4.1 编译器单元测试

为了确保功能的可用性，在开发过程中应当按单元编写相应的测试样例。避免系统运行时出现功能或性能问题。在发版前，运行所有的测试样例，保证程序的正确性，避免出现发版后程序错误的重大事故。本单元测试含有多个测试样例，分别对编译器的各个单元进行测试。此处采用 5 个具有代表性的程序进行叙述。

- (1) gcd。求最大公约数算法代码。
- (2) helloserver。最小的 HTTPS 服务器，通过 GET 请求访问 `hello` 可返回字符串内容。
- (3) qsort。快速排序的 WebC 语言实现代码。
- (4) sqltest。用于简单测试 SQL 查询功能。
- (5) toStr。用于测试 WebC 数据类型转字符串功能。

4.1.1 词法分析单元测试

词法分析是编译的第一步，其生成的标识符是整个编译过程的重要信息。故需要保证其稳定性。本单元测试涵盖不同函数个数、代码行数、标识符数量、是否带有词法错误等。测试结果如表 4.1 所示。

表 4.1 词法分析样例表

样例名	代码总行数	标识符数量	标识符实际识别数量	运行时报错
gcd	15	69	69	否
helloserver	16	92	92	否
qsort	33	278	278	否
sqltest	5	40	40	否
toStr	8	58	58	否

测试过程遇到错误不终止分析，错误标识符也算作一个标识符。测试发现结果均符合预期，满足词法预期的正确性与稳定性。

4.1.2 语法分析单元测试

语法分析单元测试结果如表 4.2 所示。每个测试用例针对性的对不同特征进行测试，保证每个指标均可通过测试。

表 4.2 语法分析测试结果

样例名	函数个数	条件分支	循环分支	函数指针	函数参数	运行时报错
gcd	2	2	0	0	2	否
helloserver	2	0	0	1	0	否
qsort	2	1	3	0	3	否
sqltest	1	1	0	0	0	否
toStr	1	0	0	0	0	否

最终测试结果均符合预期，在语法分析单元测试中通过。

4.1.3 语义分析单元测试

语义分析针对抽象语法树的特征进行测试，相关测试结果如表 4.3 所示。

表 4.3 语义分析结果

样例名	抽象语法树层数	抽象语法树结点数	运行时报错
gcd	9	42	否
helloserver	5	52	否
qsort	10	151	否
sqltest	5	20	否
toStr	6	32	否

其中，层数的计算从根结点算起，根节点为 1。经测试，结果均符合预期，语义分析通过。

4.1.4 代码优化单元测试

代码优化测试主要检测：无用代码是否删除、常量计算是否优化、可合并基础块是否已经合并、递归函数是否优化。经过测试，相应结果如表 4.4 所示。经过代码优化器，其均成功优化，且运行过程无报错。

表 4.4 代码优化结果

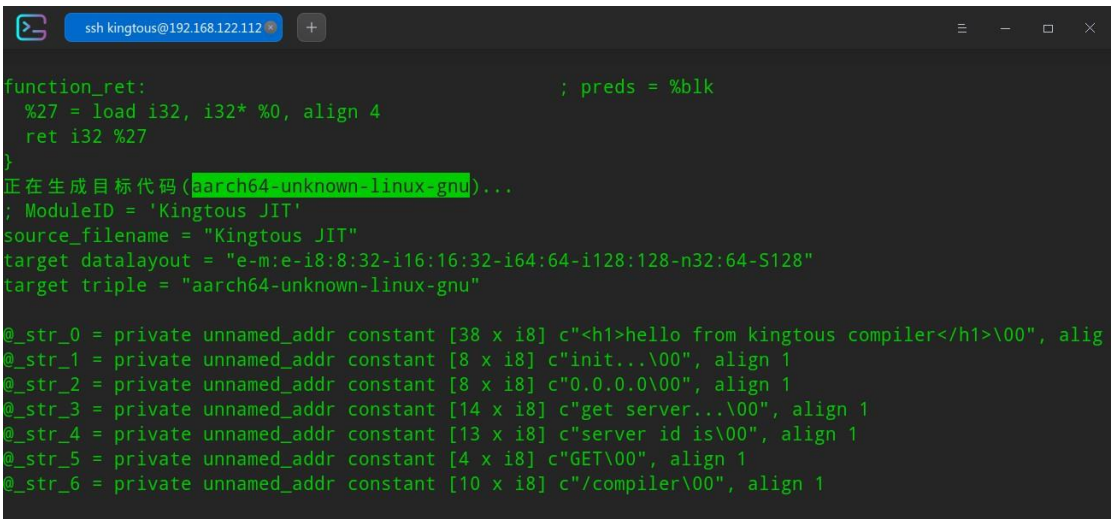
样例名	成功优化	运行时报错
gcd	是	否
helloserver	是	否
qsort	是	否
sqltest	是	否
toStr	是	否

4.1.5 目标代码生成单元测试

目标代码生成单元测试主要是针对不同平台进行针对性测试，检测其是否能在平台上编译成功。QEMU 是一个可以模拟计算机架构的虚拟机，广泛用于平台虚拟。本单元测试选取了 3 大具有代表性的架构体系（arm、mips、x86），使用 Debian 系列操作系统进行测试。另外，除下 Linux 环境，本单元测试另外对 macOS、Windows 系统进行编译测试。由于各个平台运行结果一致，本节不一一列举运行结果。

首先进行架构测试。

（1）arm64。arm64 又称 aarch64 架构，二者等效。其为 arm 系列的 64 位架构。与当前的移动设备（手机、部分 arm 系笔记本）架构相同。如图 4.1 所示，其可在 arm64 上成功编译。



```
function_ret:                                ; preds = %blk
    %27 = load i32, i32* %0, align 4
    ret i32 %27
}
正在生成目标代码(aarch64-unknown-linux-gnu)...
; ModuleID = 'Kingtous JIT'
source_filename = "Kingtous JIT"
target datalayout = "e-m:e-i8:8:32-i16:16:32-i64:64-i128:128-n32:64-S128"
target triple = "aarch64-unknown-linux-gnu"

@_str_0 = private unnamed_addr constant [38 x i8] c"<h1>hello from kingtous compiler</h1>\00", align 1
@_str_1 = private unnamed_addr constant [8 x i8] c"init...\00", align 1
@_str_2 = private unnamed_addr constant [8 x i8] c"0.0.0.0\00", align 1
@_str_3 = private unnamed_addr constant [14 x i8] c"get server...\00", align 1
@_str_4 = private unnamed_addr constant [13 x i8] c"server id is\00", align 1
@_str_5 = private unnamed_addr constant [4 x i8] c"GET\00", align 1
@_str_6 = private unnamed_addr constant [10 x i8] c"/compiler\00", align 1
```

图 4.1 aarch64 编译成功示例

编译后运行目标文件，使用谷歌浏览器访问设定好的/compiler 路径，结果如图 4.2 所示。网页成功访问，源代码被正确执行。

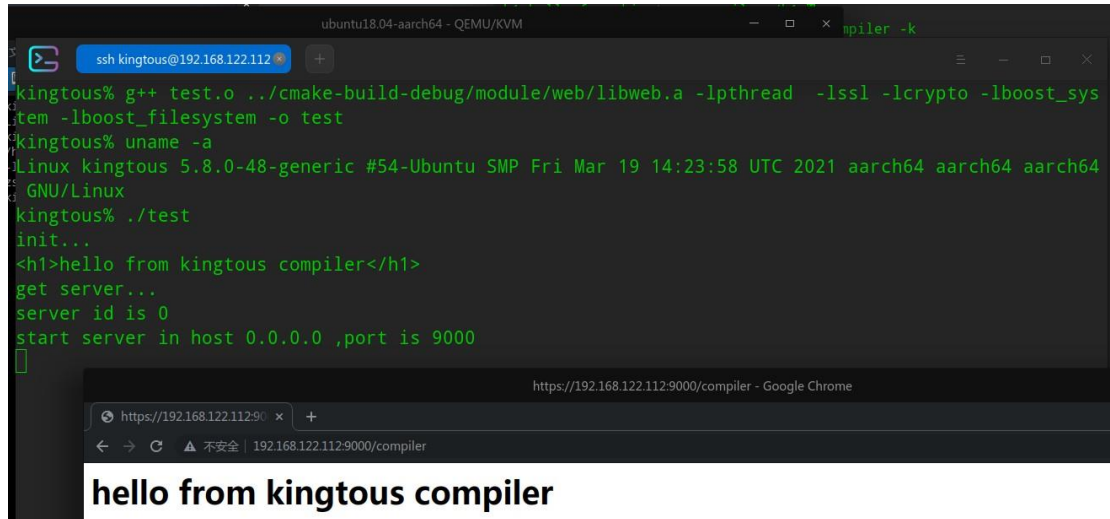


图 4.2 arm64 运行简单的 WebServer

(2) mipsel。mipsel 为 mips 架构的 32 位版本，采用小端字节序。其中，龙芯处理器使用的是 mips64el 架构，可兼容 mipsel 架构的可执行二进制文件。在该架构下编译运行 WebC 源代码并运行，为了验证服务器是否成功运行，在终端使用 curl 网络测试工具进行测试，结果成功返回，如图 4.3 所示。

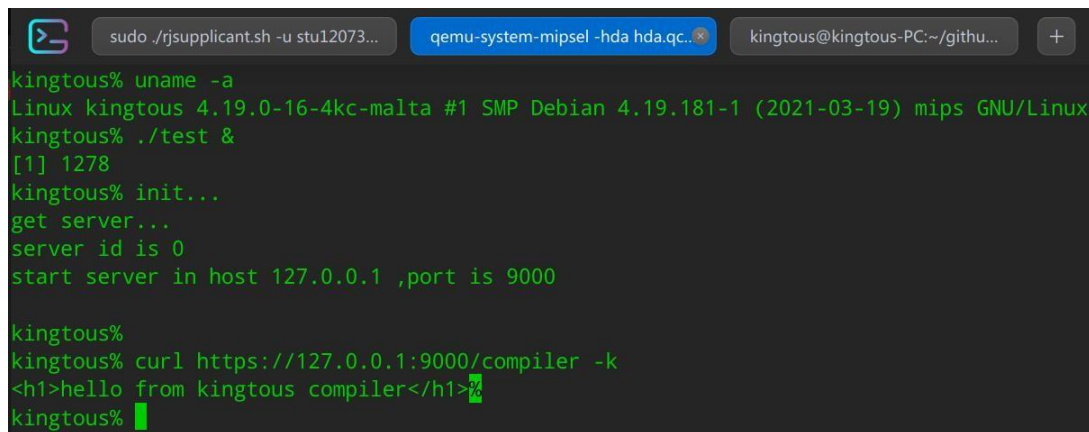


图 4.3 mipsel 架构下成功运行 HTTPS 服务器

(3) x86。x86 为当今桌面端最为常用的处理器架构。测试环境使用武汉深之度有限公司出品的国产 Linux 系统 Deepin 20.2。其编译过程如图 4.4 所示。程序被正确编译。

```
→ webc_test git:(test/webc_stat) X uname -a
Linux kingtous-PC 5.10.5-amd64-desktop+ #1 SMP Mon Jan 11 14:55:28 CST 2021 x86_64 GNU/Linux
→ webc_test git:(test/webc_stat) X webc_compiler -i helloserver.webc -o helloserver
正在分析 helloserver.webc...
正在生成目标代码(x86_64-pc-linux-gnu)...
Info: 已生成目标文件
Info: helloserver.o
Info: 开始生成可执行文件
Info: /usr/bin/clang++
Info: 编译完成
Info: helloserver
→ webc_test git:(test/webc_stat) X ./helloserver
init...
get server...
server id is 0
start server in host 0.0.0.0 ,port is 9000
```

图 4.4 x86 的 Deepin 下进行编译

接下来进行额外的操作系统测试。

(1) macOS。在 macOS 上使用 XCode 工具链进行编译构建并运行，结果如图 4.5 所示。经过 XCode 的 Clang++工具，ld 工具编译链接后，代码被成功编译为可执行文件，并成功执行。

```
Info: 开始生成可执行文件
Info: /usr/local/opt/llvm/bin/clang++
+- 0: input, "../test/test.o", object
|- 1: input, "ssl", object
|- 2: input, "crypto", object
|- 3: input, "ksql", object
|- 4: input, "kweb", object
|- 5: input, "ktime", object
|- 6: input, "kjson", object
|- 7: input, "kstring", object
+- 8: linker, {0, 1, 2, 3, 4, 5, 6, 7}, image
9: bind-arch, "x86_64", {8}, image
ld: warning: directory not found for option '-L/Applications/Xcode.app/Content
(base) kingtous@jintaodeAir X:/Library/Developer/CommandLineTools/SDKs/MacOSX10.15.sdk/usr/bin/clang++ -o helloserver.o -L/Applications/Xcode.app/Content
<h1>hello from kingtous compiler</h1>
get server...
server id is 0
start server in host 127.0.0.1 ,port is 9000
```

图 4.5 macOS 环境下编译运行结果

(2) Windows。mingw (Minimalist GNU for Windows) 是一个在 Windows 下的 GNU 工具包,使得 Linux 下的代码经过少量修改后,可移植到 Windows 下。本测试环境为 Windows10 20H2 版本,使用 msys2 mingw 环境。代码经过编译运行后,结果如图 4.6 所示。代码成功编译并运行。并在任务管理器中查询得,在 Windows 下使用 WebC 语言编写的简单 HTTPS 服务器,在进行网络请求后,内存占用稳定在 1.4MB,相比等效情况下,Flask (Python) 占用 19.1MB, Spring Boot (Java/Kotlin) 占用 321.2MB, 具有非常高的资源利用率,符合嵌

入式设备内存限制。

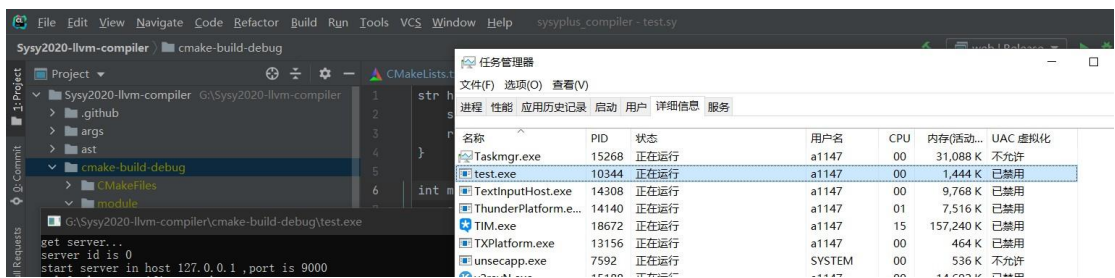


图 4.6 Windows 环境下编译运行结果

4.2 编译器性能测试

为了保证稳定性，分别对编译时和集成开发环境的实时分析稳定性进行测试。测试结果以毫秒为单位显示。

4.2.1 文件编译稳定性测试

对上述每个样例进行重复性编译测试。编译目标文件为二进制可执行文件，静态编译。环境为 x86_64 位的 Linux 环境。从代码优化开始计时，到目标文件输出到文件系统时停止计时。测试结果如图 4.7 所示。

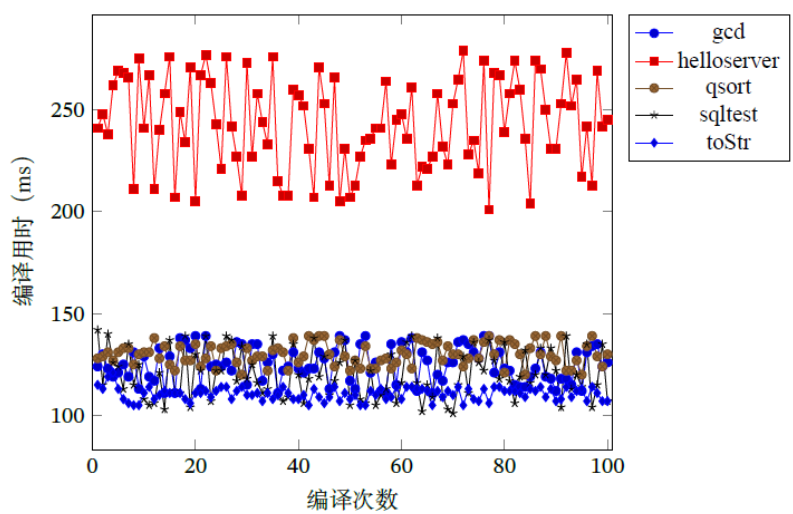


图 4.7 文件稳定性测试结果

从图 4.7 中可得，文件编译整体处于稳定状态，在含有 Web 功能的源代码编译中会额外多出 100ms 的编译时间，在可接受的范围内。由于 WebC 编译器是静态编译，故其会将如 boost 等运行环境打包于一身，优点是独立于用户环境，减小对用户环境的要求。以 helloserver 为例，最终的二进制可执行文件大小为 2.0MB，其包括 WebC 代码逻辑以及 MySQL 连接库以及高性能 HTTPS 服务器库，生成文件小，便携，运行速度快。

4.2.2 实时分析稳定性测试

对上述每个样例进行重复性实时分析测试。实时分析包括词法、语法分析以及分析日志的输出。测试结果如图 4.8 所示。

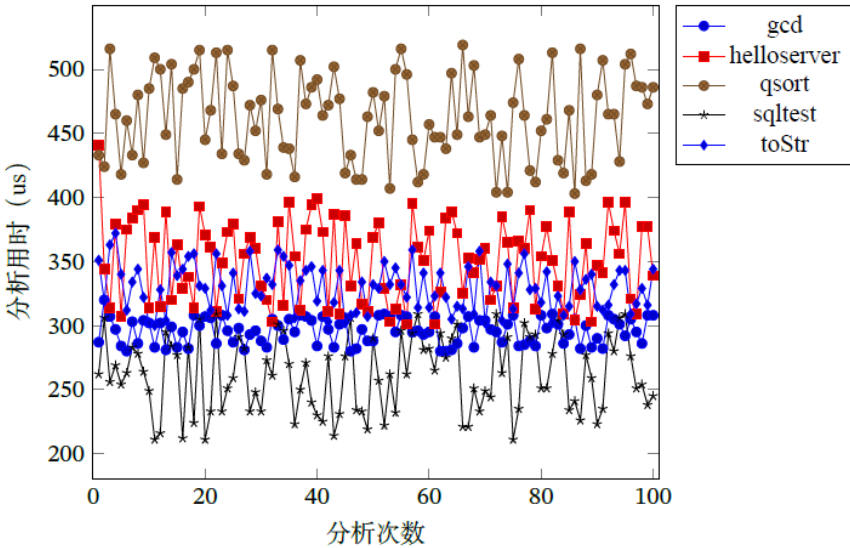


图 4.8 实时分析测试结果

由图 4.8 中可得出，所有测试样例中的分析用时均<1 毫秒，做到了无感知的实时分析。特别的，在抽象语法树结点比其他样例大 2 倍的 qsort 样例中，分析用时仅多于约 0.1ms，几乎可以忽略不计。

4.3 编译产物性能测试

apachebench 是 apache 组织开发的 HTTPS 性能测试工具，常用于用来测试接口性能。本测试使用此工具模拟高并发对接口进行测试，接口访问时对数据库取出学生数据，并通过 json 的形式返回给工具。工具计算整个过程的耗时，以毫秒为单位。

本测试模拟 20000 次测试，并发量为 1000。相关测试结果如表 6.5、表 6.6 所示。从表中中位数可看出，近乎一半的请求的耗时在 200 毫秒左右，80%的请求耗时在 1s 左右，在高并发情况下可以保持正常服务。另外，只有 2%左右的请求耗时在 2-3.5 秒左右，但均处于可接受状态。在 20000 次中，无失败请求，均请求成功。在整个请求过程中，服务器占用的峰值内存为 2.5MB，整体稳定在 1.1MB，体现了在保证性能的情况下，对资源的高利用。

表 4.5 连接耗时测试

	最小	平均	中位数	最大
连接	2	437	137	3361

处理	10	52	49	165
等待	1	29	26	125
总共	40	489	192	3460

表 6.6 接口数据返回总耗时百分比

百分比	总耗时（ms）	90%	1275
50%	192	95%	1609
66%	243	98%	2061
75%	290	99%	3152
80%	1176	100%	3460

第五章 安装及使用

此安装说明基于 Ubuntu20.04LTS，其他系统或者可能存在差异。

5.1 安装依赖

(1) 安装 llvm11
<code>sudo apt install -y llvm-11-dev libclang-11-dev</code>
(2) 安装 libedit
<code>sudo apt install -y libedit-dev</code>
(3) 安装 libboost
<code>sudo apt install -y libboost-dev libboost-system-dev libboost-filesystem-dev libboost-chrono-dev libboost-thread-dev libboost-regex-dev</code>
(4) 安装 libgtkmm-3
<code>sudo apt install -y libgtk-3-dev libgtkmm-3.0-dev libgtksourceviewmm-3.0-dev</code>
(5) 安装 mysql-connector-c++
<code>sudo apt install -y libmysqlcppconn-dev</code>
(6) 安装 openssl 库
<code>sudo apt install -y openssl libssl-dev</code>
(7) 安装 pkgconfig
<code>sudo apt install -y pkg-config</code>
(8) 安装 ninja
<code>sudo apt install -y ninja-build</code>

5.2 编译项目

在命令行模式下，新建 `build` 文件于工作目录下，使用 `cd` 命令进入 `build` 文件，然后使用 `cmake ../`，在 `build` 文件目录下生成 `makefile`，然后使用 `make` 命令即可编译成对应的目标文件。如果要编译对应的库文件，使用 `cd` 命令，进入 `module` 模块，使用 `make` 命令，即可完成编译完成对应的库文件。

5.3 使用项目

项目编译完成生成可执行文件 `WebC_compiler` 以及多种库文件，使用时通过命令行来完成编译操作，用法与 `GCC` 类似：`-i` 指定源代码文件，`-o` 指定输出文件，`-l` 链接相应的库文件。

第六章 项目总结

在该项目的开发过程中，遇到的困难有很多，但是通过多种途径终究是克服了，总结来说主要有以下几个方面：

- 1.内存管理是 C++ 中最令人切齿痛恨的问题，也是 C++ 最有争议的问题，内存管理在 C++ 中无处不在，内存泄漏几乎在每个 C++ 程序中都会发生。为了解决内存泄漏问题，起初考虑为单个的类重载 `new[]` 和 `delete[]`，由于其并不能达到预期目标，最终采用智能指针的方式来初始化变量，效果良好。

- 2.国内有关 `LLVM`、`Bison` 以及 `Boost` 库的资料较少，可以找到的教程绝大部分都是较老的版本，接口相比于最新版本发生了巨大的变化，其中绝大部分接口只能参考官网最新版的 `API`，`API` 的不同直径导致了适配的费时费力。但是通过对于官方文档的阅读，不仅提高了阅读 `API` 文档的能力，规范了代码的编写。

- 3.在实现部分功能时，由于并没有较好的思路，所以选择将 C++ 的源代码进行编译，生成中间代码，然后对应于 `LLVM` 提供的接口函数进行逐行比较，以此来理解功能的实现原理。

- 4.由于项目比较庞大，程序出现 `bug` 是在所难免的，而有些 `bug` 一旦触发，就会导致程序的崩溃，因此我们在项目中加入异常处理语句以及错误日志打印，通过编译日志定位到问题所在。

- 5.为了使 `WebC` 能够生成多平台架构支持的目标代码，需要对于目标机器的寄存器集、指令集等内容充分理解，因此在开发基本功能之余，我们也在了解学习不同平台的区别，代码编写后的测试在 `qemu` 虚拟机中进行。

正因为有各种各样的困难存在，所以有关编译器方面的人才培养以及自主知识产权才显得尤为重要。国家现在倡导自主知识产权，鼓励创新，加大了计算机系统能力的培养，我们正是在这种背景下，响应国家开发拥有自主知识产权的计算机生态系统，设计并开发了该项目。

关于未来的展望，我们认为 **WebC** 应该继续在 **Web** 方面进行深度开发，同时，也应在此基础上扩展出更多 **WebC** 独有的特性，丰富 **WebC** 的生态。