**Nina的软件开发技术手册**

**作者：谭镇洋**

**# 前言**

由于本项目代码量非常庞大，没有时间做详细的技术手册，故在此仅对整个项目的思路、框架和部分API作说明。

值得一提的是，Nina所有文档中灰色字体代表代码相关内容，请稍注意。

**# 思路**

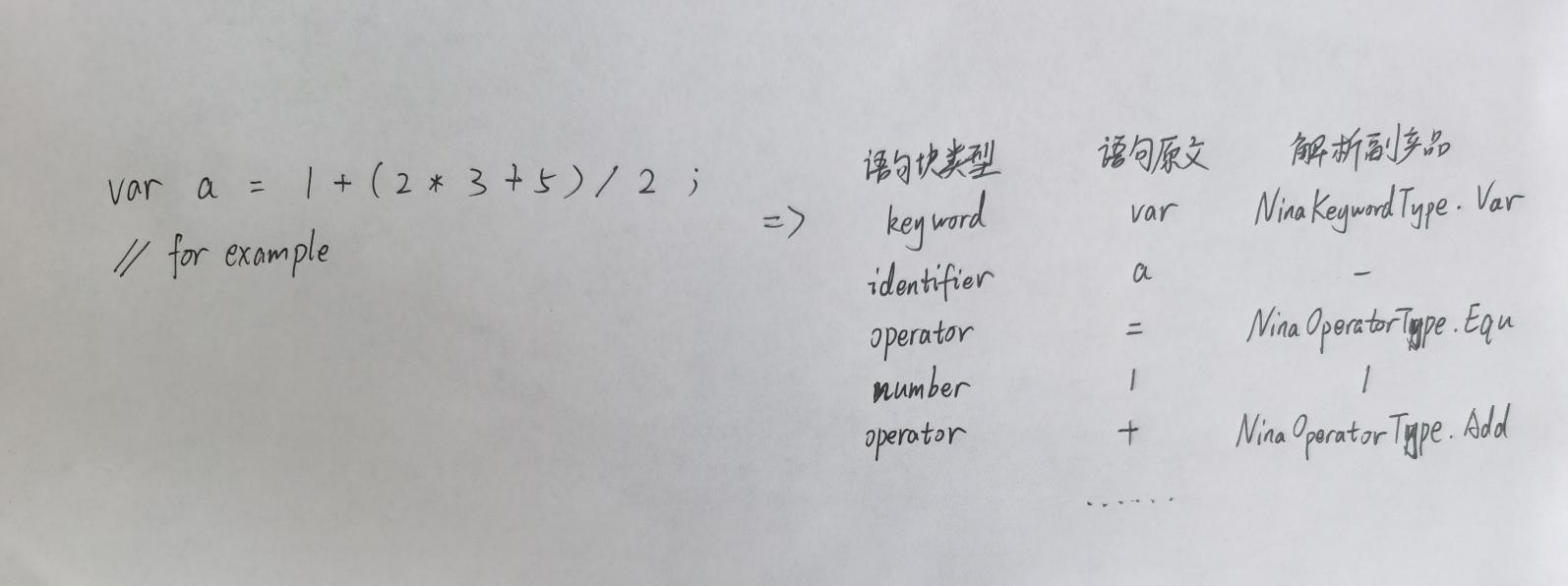
Nina是编译型脚本语言。和其他编译型语言的设计思路一样，Nina编译器的大体运行流程如下：

**- 词法分析**

遍历代码字符串，以状态机（Finite State Machine）将代码元素分块，即将空白和注释过滤，关键字（keyword）、字符串（string）、标识符（identifier）、运算符（operator）和符号（symbol）等进行初步解析；

最终生成NinaCodeBlock代码块对象的数组。

此步大致目的如下图：

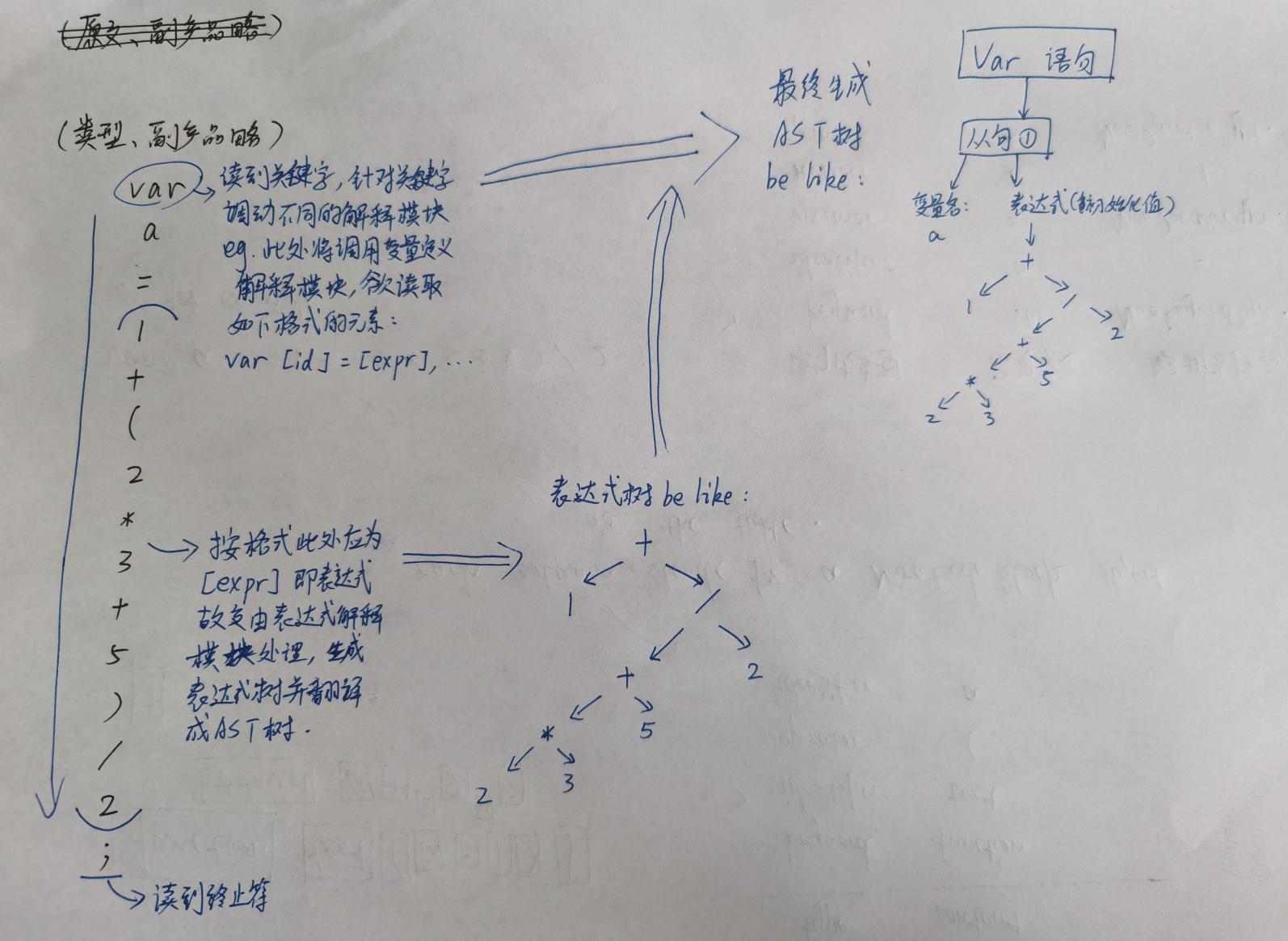


- **语法分析**

遍历NinaCodeBlock对象数组，以解释器模式（Interpreter Pattern）递归解析代码逻辑，即将语句和表达式翻译成AST树（Abstract Syntax Tree，抽象语法树，在Nina中其实质为二叉树）；

语句直接生成ANinaASTStatement（AST树语句节点基类），表达式先生成NinaExprTree表达式树，再翻译成ANinaASTExpression（AST树表达式节点基类），所有元素最终都转换成ANinaAST（AST树基类）。

此步大致目的如下图：



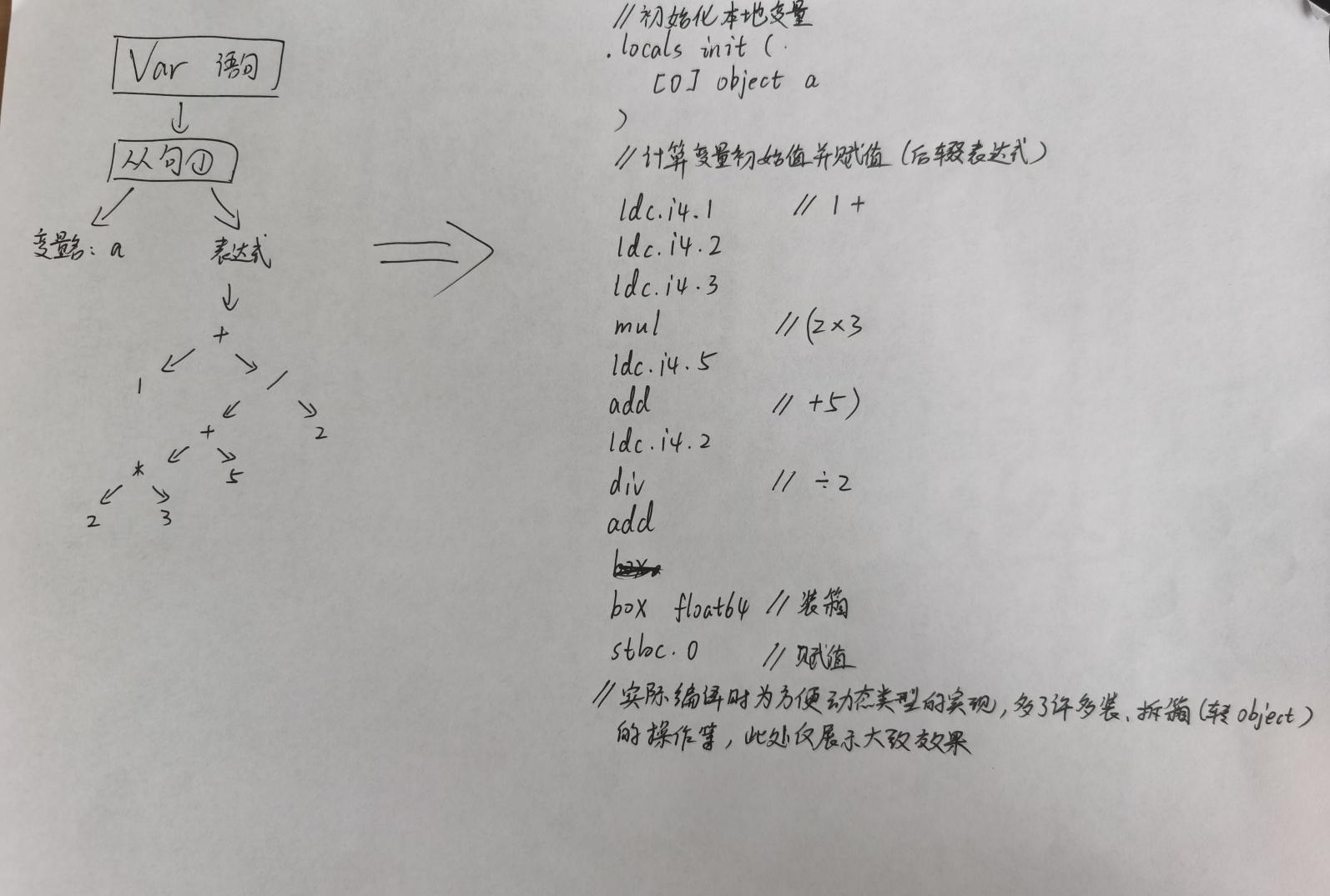
**- 编译为汇编代码并执行**

以先序顺序遍历AST树，将AST树编译为IL汇编代码（即MSIL，.NET框架的通用中间语言，为高级的虚拟机汇编语言）。

借助C#提供的System.Reflection、System.Reflection.Emit 反射API构建IL模块和IL代码，借助.NET框架的通用语言运行时（CLR，Common Language Runtime）执行。

另外，由于同在.NET框架下，上述编译成的代码能与Nina核心相互调用，得以用C#构建Nina API等。

此步大致目的如下图：



**# 部分API说明**

**- 按文件划分**

|  |  |  |
| --- | --- | --- |
| **文件名** | **概述** | **核心功能** |
| NinaAPI.cs | Nina语言的核心库 | 提供Nina语言中的内置函数、用户API，供Nina内部功能实现和用户调用 |
| NinaAST.cs | 存放AST树的各类 | - |
| NinaCodeBlock.cs | 存放代码块类 | - |
| NinaCodeResolver.cs | 存放代码块生成类 | 进行词法分析，生成代码块对象数组 |
| NinaCompiler.cs | 存放代码块解析器相关类 | 进行语法分析，生成AST树 |
| NinaCore.cs | Nina主程序入口文件 | 定义核心入口函数 |
| NinaDebugger.cs | 存放Nina开发者的调试工具类 | 开发Nina时辅助调试 |
| NinaEnums.cs | 存放Nina项目中各类枚举 | - |
| NinaError.cs | 存放Nina错误处理工具类 | 提供错误处理相关函数 |
| NinaExprTree.cs | 存放表达式树相关类 | 对表达式代码元素进行语法分析，先生成中间对象——表达式树，最终翻译为AST树 |
| NinaHacks.cs | 存放Nina中各扩展类 | - |
| NinaILCompiler.cs | 存放Nina的IL编译、执行器 | 将AST树编译到IL汇编代码并执行 |
| NinaUtils.cs | 存放Nina中各工具类 | - |
| Program.cs | C#程序默认入口文件 | 提供命令行界面（Command-Line Interface，CLI） |

**- 按函数划分（按执行流程；仅展示核心函数）**

|  |  |  |  |
| --- | --- | --- | --- |
| **层次** | **所在文件** | **函数名** | **描述** |
| 0 | NinaCore.cs | NinaCore.execute(...)  （+1重载） | 执行Nina代码的入口函数 |
| 1 | NinaCodeResolver.cs | NinaCodeResolver.blocking(...) | **词法分析。**生成NinaCodeBlock代码块对象数组 |
| 1 | NinaCompiler.cs | NinaCompiler.compile(...)  （+1重载） | **语法分析。**生成ANinaAST AST语法树。此函数为语法分析入口函数 |
| 2 | NinaCompiler.cs | NinaCompiler.resolve\_expr(...) | **语法分析。**针对表达式。先生成NinaExprTree表达式树 |
| 3 | NinaExprTree.cs | NinaExprTree.compile(...) | **语法分析。**将表达式树翻译成ANinaAST AST语法树 |
| 1 | NinaILCompiler.cs | NinaILCompiler.execute(...) | **编译到IL汇编代码并执行。**此函数为编译AST树的入口函数，编译完后会直接执行生成的IL代码 |
| 2 | NinaILCompiler.cs | NinaILCompiler.init\_apis(...) | **编译到IL汇编代码。**初始化Nina内置用户API的动态链接（Nina中内置函数默认为静态链接，即编译时即可明确地知道调用的内置函数，这样便于优化生成的IL代码的性能。故当用户不直接调用内置API函数、并将其作为普通对象使用、存储，之后再调用时，即动态链接时，需要在IL代码中构建辅助代码加以支持） |
| 2 | NinaILCompiler.cs | NinaILCompiler.compile\_main(...) | **编译到IL汇编代码。**此函数为遍历AST树的入口函数 |
| 3 | NinaILCompiler.cs | NinaILCompiler.compile\_block(...) | **编译到IL汇编代码。**编译代码块（block，相当于多个语句（statement）组成的语句数组） |
| 4 | NinaILCompiler.cs | NinaILCompiler.compile\_stm(...) | **编译到IL汇编代码。**编译语句 |
| 5 | NinaILCompiler.cs | NinaILCompiler.compile\_expr(...) | **编译到IL汇编代码。**编译表达式 |
| 6 | NinaILCompiler.cs | NinaILCompiler.compile\_identifier(...) | **编译到IL汇编代码。**编译标识符，进行变量或成员的取值或赋值 |
| 6 | NinaILCompiler.cs | NinaILCompiler.compile\_innerFunc(...) | **编译到IL汇编代码。**解析内置函数（静态链接）。注意此函数返回的是MethodInfo对象，便于直接推送到IL代码流中 |
| >=3 | NinaILCompiler.cs | NinaILCompiler.potable(...) | **记录调试信息（行数映射表）。**生成的行数映射表用于出错时查找错误位置（后由内置函数NinaAPI.error\_ex\_report(...)使用并算出源文件的具体错误位置） |

**# 特别说明**

**- 表达式树的生成**

为了便于遍历，Nina中的表达式树（NinaExprTree）采用二叉树形式，且遵循“上低下高”原则：运算符优先级低的节点在上，反之在下（以根节点在最上边来看）。

但是，这里有个非常隐蔽的注意事项：考虑到单目运算符的向右结合性，若两个优先级相等的是单目运算符，则应较新读到的节点在下；若不是，则新读到的节点在上在下均可。

另外，单目运算符的处理也比较吃力。若按普通思路来解决非常费力，Nina采用了一个骚气的解法：读到单目运算符时，在其左边“放置”一个空块作其左手值，即可将单目运算符转换为双目运算符来处理——只要正确区分单、双目节点、处理好空块，像这样就可以轻松解决单目运算符问题。

**- 闭包的实现**

闭包几乎是脚本语言的必备——闭包的含义类似于：函数A定义里的函数B可以访问函数A甚至更上级的函数内的变量。但我们的汇编语言——IL中并没有直接的闭包实现，我们需要用复杂一些的方法来实现闭包。参考其他语言的闭包实现，Nina采用了如下的闭包实现方法：

1. Nina中每个函数都是闭包
2. 编译时每出现一个函数（后称“该函数”），都会在IL代码流中创建一个抽象、静态的临时类（后称“临时类”），该类内含有该函数作为成员函数（后称“临时函数”）
3. 临时函数中用户定义的本地变量（局部变量）都存储在一个字典（Dictionary）类型的、临时函数的IL本地变量中（后称“变量集”）
4. 实际执行时，每次生成该函数时，都将父函数（闭包所在的函数）当前的变量集的引用赋值到临时类的一个字段中
5. 临时类创建时，都将父函数存储的变量集字段（即父函数的父函数等等的变量集）引用复制（或更贴切地说：继承）到该临时类
6. 临时函数查找标识符时，会先查找本地变量集，再一级一级往上查找（先查找父函数的变量集，再查找父函数的父函数的变量集……）

其实总体上参照了JavaScript的“链式作用域”做法，其基本思路就是在各函数之间以链式存储、共享变量集。不过Nina目前的闭包会有内存泄露问题，因时间问题暂未解决。

**# 其他**

|  |  |
| --- | --- |
| **项目** | **明细** |
| 开发环境/工具 | Windows 11，.NET 6.0 Framework，.NET 6.0 SDK，Visual Studio Code |
| 运行环境 | Windows 10+，无需安装其他框架、环境（Nina发行版自带.NET 6.0环境） |
| 安装包制作工具 | Inno Setup |
| Github | <https://github.com/jwhgzs/nina> |
| 个人网站 | <https://jwhgzs.com> |