Preface

> This series was written in Chinese originally. This Engine version is mainly translated by Google Translate. So please forgive me for the terrible writing.

This series of articles introduces the implementation of a Lua interpreter from scratch in the Rust language.

The Rust language has a distinctive personality and is also widely popular, however the learning curve is steep. After I finished reading "Rust Programming Language" and wrote some practice codes, I deeply felt that I had to go through a larger project practice to understand and master.

Implementing a Lua Interpreter is very suitable as this exercise project. Because of its moderate scale, it is enough to cover most of the basic features of Rust without being difficult to reach; clear goal, no need to spend energy discussing requirements; in addition, Lua language It is also an excellently designed and widely used language. Implementing a Lua interpreter can not only practice Rust language skills, but also gain an in-depth understanding of Lua language.

This series of articles documents the learning and exploration process during this project. Similar to other from scratch Build your own X projects, this project also has a clear big goal, an unknown exploration process and a continuous sense of accomplishment, but with some differences:

- Most of the authors of other projects have been immersed in related fields for many years, but my job is not in the direction of programming language or compilation principles. I don't have complete theoretical knowledge for implementing an interpreter, and I just cross the river by feeling the stones. But think of the good in everything, which also provides a real beginner's perspective.

- Most of the other projects are for the purpose of learning or teaching, simplifying the complexity and realizing a prototype with only the most basic functions. But my goal is to implement a production-level Lua interpreter, pursuing stability, completeness, and performance.

In addition, since the original intention of the project is to learn the Rust language, there will also be some learning notes and usage experience of the Rust language in the article.

Content

The content is organized as follows. Chapter 1 implements a minimal interpreter that can only parse `print "hello, world!"` statements. Although simple, it includes the complete process of the interpreter and builds the basic framework. Subsequent chapters will gradually add Lua features to this minimal interpreter.

Chapter 2 introduces the most basic concepts of types and variables in programming languages. Chapter 3 introduces several features of the Rust language with the goal of perfecting the string type. Chapter 4 implements the table structure in Lua and introduces the key ExpDesc concept in syntax analysis. Chapter 5 is about tedious arithmetic calculations.

In Chapter 6 Control Structures, things start to get interesting, jumping back and forth between bytecodes based on judgment conditions. Chapter 7 introduces logical and relational operations, combined with the control structures of the previous chapter through specific optimizations.

Chapter 8 introduces functions. The basic concept and implementation of functions are relatively simple, but variable parameters and multiple return values require careful management of the stack. The closure introduced in Chapter 9 is a powerful feature in the Lua language, and the key here is Upvalue and its escape.

Every feature is designed on demand, but not completed in one step like a prophet. Take the conditional jump instruction as example, at the beginning, in order to support the `if` statement, we add `Test(u8, u16)` bytecode, which means if the value of the first associated parameter is *false*, then jump *forward* to the distance represented by the second associated parameter; then in order to support the `while` statement and need to jump *backward*, we change the second associated parameter from `u16` to `i16` type, and use a negative number to represent backward jump; then in order to support logical operations, which may jump if *true* or *false* both, we add `TestAndJump` and `TestOrJump` two bytecodes to replace `Test`. As a result, according to our own learning and development path, we produced a set of bytecodes slightly different from the official version of Lua.

Each chapter starts with Lua's functional features, discussing how to design and then introducing specific implementations. It's not only important to explain "how to do it," but also to explain "why to do it". However, to achieve complete Lua features, some articles may be boring, especially in the first few chapters. Readers can browse the relatively interesting Definition of String Type and Escape of Upvalue, to judge whether this series of articles is to your taste.

Each chapter has a complete runnable code, and the code for each chapter is based on the final code of the previous chapter, ensuring that the entire project is continuous. In the beginning chapters, after introducing the design principles, the code will be explained line by line; later on, only the key parts of the code will be explained; and in the last two chapters will basically not talk about the code.

At present, these chapters only complete the core part of the Lua interpreter, and are still far from a complete interpreter. The To be continued section lists a partial list of unfinished features.

The basic syntax of Lua and Rust will not be explained in this article. We expect readers to have a basic understanding of both languages. The more familiar with the Lua language, the better. There are no high requirements for the Rust language, as long as you have read "Rust Programming Language" and understand the basic grammar. After all, the original intention of this project is to learn Rust. In addition, when it comes to implementing a language interpreter, it will remind people of the difficult compilation principle. However, in reality, since Lua is a very simple language and there is Lua's official implementation of the interpreter code as a reference, this project requires little theoretical knowledge and is mainly focused on practical engineering.

Due to my limited technical ability in compiling principles, Lua language, Rust language, etc., there must be mistakes in projects and articles. In addition, this English version articles are automatically translated from the Chinese version mostly, so there may be many not appropriate or not fluent sentences. You are welcome to come to the project's github homepage to submit issue feedback.

hello, world!

Following the tradition of introducing programming languages, we start with "hello, world!". However, instead of writing a program to directly output this sentence, we need to implement a minimal Lua interpreter to interpret and execute the following Lua code:

```
print "hello, world!"
```

Although this code is simple, our minimal version of the interpreter will still contain the complete process of a general-purpose interpreter, including steps such as lexical analysis, syntax analysis, bytecode generation, and virtual machine execution. In the future, as long as features are added on the basis of this process, a complete Lua interpreter can be gradually realized.

However, this Lua code is not as simple as it seems. It contains many concepts such as global variables (print), string constants ("hello, world!"), standard library (print) and function calls. These concepts depend on Lua's internal concepts such as values and stacks. Being able to interpret and execute this code gives you an intuitive understanding of how the interpreter works.

In order to complete this interpreter, this chapter first introduces the necessary knowledge of compilation principles. This should be the only theoretical part in the entire series of articles, and it may also be the section with the most errors. It then introduces the two core concepts of bytecode and value. Then gradually implement lexical analysis, syntax analysis and virtual machine. Finally, an interpreter (only) capable of executing the above Lua code is completed.

Compilation principle

The principle of compilation is a very profound and mature subject. It is not necessary or capable to give a complete or accurate introduction here. It is just a simple concept introduction according to the subsequent implementation process. If you are familiar with the concept of an interpreter, you can skip this section.

Compiled and Interpreted language

Regardless of the programming language, before the source code is handed over to the computer for execution, a translation process is necessary to translate the source code into a computer-executable language. According to the timing of this translation, programming languages can be roughly divided into two types:

- Compiled type, that is, the compiler first compiles the source code into a computer language and generates an executable file. This file is subsequently executed directly by the computer. For example, under Linux, use the compiler gcc to compile the C language source code into an executable file.
- Interpreted type requires an interpreter, which loads and parses the source program in real time, and then maps the parsed results to pre-compiled subroutine and executes them. This interpreter is generally implemented by the above compiled language.

```
interprete
+-------------+  compile  +-----------------+      +-------------+  & execute
+-----------------+
| source file | --------> | executable file |      | source file |
-----------> | Lua interpreter |
|  bar.c      |           |  bar.exe        |      |  bar.lua    |
|  lua.exe        |
+-------------+           +-----------------+      +-------------+
+-----------------+
                               ^
^
                               | execute
| execute
                               | machine code
| machine code
                           +----------------+
+----------------+
                           |    computer    |
|    computer    |
                           +----------------+
+----------------+

                 Compiled
Interpreted
```

The figure above roughly shows the two types of translation and execution processes. Lua is an interpreted language, and our goal is to implement a Lua interpreter.

Parse and Execute

The general compilation principle process is as follows:

```
            Lexical Analysis        Syntax Analysis       Semantic Analysis
Character Stream --------> Token Stream --------> Syntax Tree -------->
Intermediate Code ...
```

- The character stream corresponds to the source code, that is, the source code is treated as a character stream.
- Lexical analysis, which splits the character stream into tokens supported by the language. For example, the above Lua code is split into two Tokens: "identification `print` " and "string `"hello, world!"` ". Lua ignores whitespace characters.
- Syntax analysis, which parses the Token stream into a syntax tree according to grammer rules. For example, the two tokens just now are recognized as a function call statement, in which "identity `print` " is the function name, and "string `"hello, world!"` " is the parameter.

- Semantic analysis, which generates the corresponding intermediate code from the statement of this function call, these codes indicate where to find the function body, where to load the parameters and so on.

After intermediate code is generated, compiled and interpreted languages diverge. The compiled language moves on, eventually generating machine code that can be executed directly, and packaged as an executable file. For the interpreted language, this is the end, the generated intermediate code (generally called bytecode) is the result of compilation; and the execution of the bytecode is the task of the virtual machine.

The virtual machine converts the bytecode into a corresponding series of precompiled subroutine, and then executes them. For example, to execute the bytecode generated above, the virtual machine first finds the corresponding function, namely `print`, which is a function in the Lua standard library; then loads the parameters, namely "hello, world"; finally calls the `print` function which outputs "hello, world!".

The above just describes a general process. Specific to each language or each interpreter process may be different. For example, some interpreters may not generate bytecode, but let the virtual machine directly execute the syntax tree. The official implementation of Lua omits the syntax tree, and the bytecode is directly generated by syntax analysis. Each of these options has advantages and disadvantages, but they are beyond the scope of our topic and will not be discussed here. Our interpreter is a full reference to the official Lua implementation in the main process, so the final process is as follows:

```
            Lexical Analysis         Syntax Analysis
Character Stream --------> Token Stream --------> Bytecode
                                                     ^
                                                     |
                                               virtual machine
```

From this we can clarify the main functional components of our interpreter: lexical analysis, syntax analysis and virtual machine. The combination of lexical analysis and syntax analysis can be called the "parsing" process, and the virtual machine is the "execution" process, then the bytecode is the link connecting the two processes. The two processes of parsing and execution are relatively independent. Next, we use the bytecode as a breakthrough to start implementing our interpreter.

Bytecode

As a beginner, it is natural to feel lost and unsure of how to start implementing an interpreter. No way to start.

Fortunately, the previous section introduces the bytecode at the end, and divides the entire interpreter process into two stages: parsing and execution. Then we can start with the bytecode:

- Determine the bytecode first,
- then let the parsing process (lexical analysis and parsing) try to generate this set of bytecodes,
- Then let the execution process (virtual machine) try to execute this set of bytecodes.

```
         generate             execute
   parse -------> bytecode <------- virtual machine
```

But what does bytecode look like? How to define? What type? We can refer to the official implementation of Lua.

Output of `luac`

For the convenience of description, the object code is listed again here:

```lua
print "hello, world!"
```

The official implementation of Lua comes with a very useful tool, `luac`, namely Lua Compiler, which translates the source code into bytecode and outputs it. It is our right-hand man in this project. Take a look at its output to the "hello, world!" program:

```
$ luac -l hello_world.lua

main <hello_world.lua:0,0> (5 instructions at 0x600000d78080)
0+ params, 2 slots, 1 upvalue, 0 locals, 2 constants, 0 functions
    1   [1]     VARARGPREP      0
    2   [1]     GETTABUP        0 0 0   ; _ENV "print"
    3   [1]     LOADK           1 1     ; "hello, world!"
    4   [1]     CALL            0 2 1   ; 1 in 0 out
    5   [1]     RETURN          0 1 1   ; 0 out
```

The first 2 lines of the output are incomprehensible, so ignore them now. The following should be the bytecode, and there are comments, which is great. But still do not understand. Check out Lua's official manual, but I can't find any explanation about bytecode. It turns out that the Lua language standard only defines the characteristics of the language, while the bytecode belongs to the "concrete implementation" part, just like the variable naming in the interpreter code, which does not belong to the definition scope of the Lua standard. In fact, the Luajit project, that is fully compatible with Lua 5.1, uses a completely different bytecode. We can even implement an interpreter without bytecode. Since the manual does not explain it, we can only check the the comment in source code. Here we only introduce the 5 bytecodes that appear above:

1. VARARGPREP, temporarily unused, ignored.
2. GETTABUP, this is a bit complicated, it can be temporarily understood as: loading global variables onto the stack. The three parameters are the stack index (0) as the target address, (ignore the second one,) and the index (0) of the global variable name in the constant table. The global variable name listed in the comments later is "print".
3. LOADK, load constants onto the stack. The two parameters are the stack index (1) as the destination address, and the constant index (1) as the loading source. The value of the constant listed in the comment below is "hello, world!".
4. CALL, function call. The three parameters are the stack index (0) of the function, the number of parameters, and the number of return values. The following comment indicates that there is 1 parameter and 0 return value.
5. RETURN, temporarily unused, ignored.

Take a look at it together:

- First load the global variable named `print` into the stack (0);
- Then load the string constant `"hello, world!"` into the stack (1);
- Then execute the function at the stack (0) position, and take the stack (1) position as a parameter.

The stack diagram during execution is as follows:

```
  +-----------------+
0 | print           | <- function
  +-----------------+
1 | "hello, world!" |
  +-----------------+
  |                 |
```

We currently only need to implement the above three bytecodes 2, 3, and 4.

Bytecode Definition

Now define the bytecode format.

First refer to the format definition of Lua's official implementation. Source code has comments on the bytecode format:

```
  We assume that instructions are unsigned 32-bit integers.
  All instructions have an opcode in the first 7 bits.
  Instructions can have the following formats:

        3 3 2 2 2 2 2 2 2 2 2 2 1 1 1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0
        1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0
iABC          C(8)     |      B(8)     |k|     A(8)      |   Op(7)     |
iABx                Bx(17)               |     A(8)      |   Op(7)     |
iAsBx              sBx (signed)(17)      |     A(8)      |   Op(7)     |
iAx                           Ax(25)                     |   Op(7)     |
isJ                           sJ(25)                     |   Op(7)     |

  A signed argument is represented in excess K: the represented value is
  the written unsigned value minus K, where K is half the maximum for the
  corresponding unsigned argument.
```

The bytecode is represented by a 32bit unsigned integer. The first 7 bits represent the command, and the following 25 bits represent the parameters. There are 5 formats of bytecode, and the parameters of each format are different. If you like this sense of precise bit control, you may immediately think of various bit operations, and you may already be excited. But don't worry, let's look at Luajit's bytecode format first:

```
A single bytecode instruction is 32 bit wide and has an 8 bit opcode field
and
several operand fields of 8 or 16 bit. Instructions come in one of two
formats:

+---+---+---+---+
| B | C | A | OP|
|   D   | A | OP|
+---+---+---+---+
```

It is also a 32bit unsigned integer, but the division of fields is only accurate to bytes, and there are only 2 formats, which is much simpler than the official Lua implementation. In C language, by defining matching struct and union, bytecode can be constructed and parsed more conveniently, thus avoiding bit operations.

Since the Lua language does not specify the bytecode format, we can also design our own bytecode format. For different types of commands like this, where each command has unique associated parameters, it is very suitable to use Rust's enum: use tags as commands, and use associated values as parameters. Let's define the bytecodes like this:

```rust
#[derive(Debug)]
pub enum ByteCode {
    GetGlobal(u8, u8),
    LoadConst(u8, u8),
    Call(u8, u8),
}
```

Luajit's bytecode definition can avoid bit operations, and using Rust's enum can go a step further, where you don't even need to care about the memory layout of each bytecode. You can use the enum creation syntax to construct bytecode, such as `ByteCode::GetGlobal(1,2)`; use pattern matching `match` to parse bytecode. The parsing and virtual-matchine modules in Section 1.4 construct and parse bytecodes respectively.

But also pay attention to ensure that the enum does not exceed 32bit, so we still need to understand the layout of the enum. The size of the enum tag in Rust is in bytes and is allocated on demand. So as long as there are less than 2^8=256 kinds of bytecodes, the tag only needs 1 byte. Only 7 bits are used to indicate the command type in Lua's official bytecode, so 256 is enough. Then there is still 3 bytes of space to store parameters. In the two bytecode types of Luajit, the parameters only occupy 3 bytes, which is enough. This article introduces the method of static checking, but due to the need for third-party libraries or macros, we don't use it here for the time being.

> Rust's enum is really nice!

Two Tables

As you can see from analysis above, we also need two tables except the bytecodes.

First, we need a *constant table* to store all the constants during the parsing process. The generated bytecodes refer to the corresponding constant through the index parameter. And during the execution process, the virtual machine reads the constants from this table through the bytecodes' parameter. In this example, there are two constants, one is the name of the global variable `print`, and the other is the string constant "hello, world!". This is the meaning of `2 constants` in the second line of the above `luac` output.

Then we need a *global variable table* to save global variables according to variable names. During execution the virtual matchine first queries the global variable name in the constant table through the parameters in the bytecodes, and then queries the global variable table according to the name. The global variable table is only used (add, read, modify) during execution, and has nothing to do with the parsing process.

The specific definition of these two tables needs to rely on the concept of Lua's "value", which will be introduced in the next section.

Value and Type

The previous section defined the bytecode, and mentioned at the end that we need two tables, the constant table and the global variable table, respectively to maintain the relationship between constants/variables and "values", so their definitions depend on the "Value" 's definition in Lua. This section introduces and defines Lua's value.

For the convenience of description, all the words "variable" in this section later include variables and constants.

Lua is a dynamically typed language, and the "type" is bound to a value, not to a variable. For example, in the first line of the following code, the variable `n` contains the information: "the name is n"; while value `10` contains the information: "the type is an integer" and "the value is 10". So in line 2, it's OK to assign `n` to a different type value.

```lua
local n = 10
n = "hello" -- OK
```

For comparison, here's the statically typed language Rust. In the first line, the information of `n` is: "the name is n" and "the type is i32"; the information of `10` is: "the value is 10". It can be seen that the "type" information has changed from the attribute of the variable to the attribute of the value. So you can't assign `n` to a string value later.

```rust
let mut n: i32 = 10;
n = "hello"; // !!! Wrong
```

The following two diagrams represent the relationship between variables, values, and types in dynamically typed and statically typed languages, respectively:

```
 variable         values                     variable               values
+----------+     +----------------+         +----------------+
+------------+
| name: n |--\-->| type: Integer |          | name: n        |----->| value: 10
|
+----------+  |    | value: 10     |          | type: Integer |  |
+------------+
             |    +----------------+         +----------------+  X
             |                                                    |
             |    +-----------------+                             |
+-----------------+
             \-->| type: String   |                             \-->| value:
"hello" |
                 | value: "hello" |
+-----------------+
                 +-----------------+

          dynamic type                              static type
    "type" is bound to values                 "type" is bound to variables
```

Value

In summary, the value of Lua contains type information. This is also very suitable for defining with enum:

```
use std::fmt;
use crate::vm::ExeState;

#[derive(Clone)]
pub enum Value {
    Nil,
    String(String),
    Function(fn (&mut ExeState) -> i32),
}

impl fmt::Debug for Value {
    fn fmt(&self, f: &mut fmt::Formatter) -> Result<(), fmt::Error> {
        match self {
            Value::Nil => write!(f, "nil"),
            Value::String(s) => write!(f, "{s}"),
            Value::Function(_) => write!(f, "function"),
        }
    }
}
```

Currently 3 types are defined:

- `Nil` , Lua's null value.
- `String` for the `hello, world!` string. For the associated value type, the simplest

  `String` is temporarily used, and it will be optimized later.
- `Function` for `print`. The associated function type definition refers to the C API function definition `typedef int (*lua_CFunction) (lua_State *L);` in Lua, and will be improved later. Among them, `ExeState` corresponds to `lua_State`, which will be introduced in the next section.

Other types such as integers, floating-point numbers and tables will be added in the future.

Above the Value definition, the `Clone` trait is implemented via `#[derive(Clone)]`. This is because Value will definitely involve assignment operations, and the String type includes Rust's string `String`, which does not support direct copying, namely the `Copy` trait is not implemented, or it owns the data on the heap. So we can only declare the whole Value as `Clone`. All assignments involving Value need to be done through `clone()`. It seems that the performance is worse than direct assignment. We will discuss this issue later when we define more types.

We also manually implemented the `Debug` trait to define the print format, after all, the function of the current object code is to print "hello, world!". Since the function pointer parameter associated with `Function` does not support the `Debug` trait, it cannot be automatically implemented by `#[derive(Debug)]`.

Two Tables

After defining the Value, we can define the two tables mentioned at the end of the previous section.

Constant table stores constants. Bytecodes refer to constants by index directly, so constant tables can be represented by Rust's variable-length array `Vec<Value>`.

The global variable table, which stores global variables according to their names, can *temporarily* be represented by Rust's `HashMap<String, Value>`. We will change this later.

> Compared with the ancient C language, components such as `Vec` and `HashMap` in the Rust standard library have brought great convenience and consistent experience.

Let's Do It

The previous chapters introduced the basics of compilation principles, and defined the two most important concepts, ByteCode and Value. Next, we can start coding to implement our interpreter!

The code corresponding to this series of articles is all managed by Cargo that comes with Rust. Projects currently using the binary type will be changed to the library type in the future.

The minimalist interpreter to be implemented at present is very simple, with very little code. I wrote all the code in one file at the beginning. However, it is foreseeable that the code volume of this project will increase with the increase of features. So in order to avoid subsequent changes to the file, we directly create multiple files now:

- Program entry: `main.rs` ;
- Three components: lexical analysis `lex.rs` , syntax analysis `parse.rs` , and virtual machine `vm.rs` ;
- Two concepts: byte code `byte_code.rs` , and value `value.rs` .

The latter two concepts and their codes have been introduced before. The other 4 files are described below. Let's start with the program entry.

Program Entry

For the sake of simplicity, our interpreter has only one way of working, which is to accept a parameter as a Lua source code file, and then parse and execute it. Here is the code:

```rust
use std::env;
use std::fs::File;

mod value;
mod bytecode;
mod lex;
mod parse;
mod vm;

fn main() {
    let args: Vec<String> = env::args().collect();
    if args.len() != 2 {
        println!("Usage: {} script", args[0]);
        return;
    }
    let file = File::open(&args[1]).unwrap();

    let proto = parse::load(file);
    vm::ExeState::new().execute(&proto);
}
```

The first 2 lines reference two standard libraries. `env` is used to obtain command line arguments. `fs::File` is used to open Lua source files.

The middle lines refer to other file modules through `use` keyword.

Then look at the `main()` function. The first few lines read the parameters and open the source file. For the sake of simplicity, we use `unwrap()` to terminate the program if fail to open file. We will improve the error handing later.

The last 2 lines are the core function:

- First, the syntax analysis module `parse` (who also calls lexical analysis `lex` internally) parses the file and returns the parsing result `proto`;
- Then create a virtual machine and execute `proto`.

This process is different from Lua's officially APIs (complete example) :

```c
lua_State *L = lua_open(); // Create lua_State
luaL_loadfile(L, filename); // Parse and put the parsing result on the top of
the stack
lua_pcall(L, 0, 0, 0); // top of execution stack
```

This is because the official implementation of Lua is a "library", and the API only exposes the `lua_State` data structure, which contains both parsing and executing parts. So you must first create `lua_State`, and then call parsing and execution based on it. The parsing result is also passed through the stack of `Lua_state`. However, we currently do not have a similar unified state data structure, so we can only call the parsing and execution functions separately.

Let's look at the analysis and execution process respectively.

Lexical Analysis

Although the `main()` function calls the syntax analysis `parse` module firstly, but the syntax analysis calls the lexical analysis `lex` module internally. So let's see the lexical analysis first.

The output of lexical analysis is Token stream. For the "hello, world!" program, you only need to use the two Tokens "identity `print` " and "string `"hello, world!"` ". For simplicity, we only support these two kinds of tokens for the time being. In addition, we also define an `Eos` to indicate the end of the file:

```rust
#[derive(Debug)]
pub enum Token {
    Name(String),
    String(String),
    Eos,
}
```

Instead of returning a whole Token list after parsing the input file at one time, we provide a function similar to an iterator so that the syntax analysis module can be called on demand. To do this first define a lexical analyzer:

```rust
#[derive(Debug)]
pub struct Lex {
    input: File,
}
```

For now only one member is included, the input file.

It provides 2 APIs: `new()` creates a parser based on the input file; `next()` returns the next Token.

```rust
impl Lex {
     pub fn new(input: File) -> Self ;
     pub fn next(&mut self) -> Token;
}
```

The specific parsing process is pure and boring string handling, and the code is skipped.

According to the Rust convention, the return value of the `next()` function here should be defined as `Option<Token>` , where `Some<Token>` means that a new token has been read, and `None` means the end of the file. But since `Token` itself is an `enum` , it seems more convenient to directly add an `Eos` in it. And if it is changed to the `Option<Token>` type, then an additional layer of judgment will be required in the syntax analysis call, as shown in the following code. So I chose to add the `Eos` type.

```
loop {
    if let Some(token) = lex.next() { // extra check
        match token {
            ... // parse
        }
    } else {
        break
    }
}
```

Syntax Analysis

The parsing result `proto` in the `main()` function concats the parsing and execution phases. But in view of Rust's powerful type mechanism, `proto` does not show a specific type in the above code. Now let's define it. It has been introduced in the bytecode section that the analysis result needs to contain two parts: bytecode sequence and constant table. Then you can define the format of the parsing result as follows:

```
#[derive(Debug)]
pub struct ParseProto {
    pub constants: Vec::<Value>,
    pub byte_codes: Vec::<ByteCode>,
}
```

The constant table `constants` is a `Vec` containing the `Value` type, and the bytecode sequence `byte_codes` is a `Vec` containing the `ByteCode` type. They are both `Vec` structures with the same functionality but different containment types. In the ancient C language, to include the two types `Value` and `ByteCode`, either write a set of codes for each type, or use complex features such as macros or function pointers. Generics in the Rust language can abstract the same set of logic for different types. More features of generics will be used in subsequent code.

After defining `ParseProto`, let's look at the syntax analysis process. We currently only support the statement of `print "hello, world!"`, which is the format of `Name String`. The Name is first read from the lexer, followed by the string constant. If it is not in this format, an error will be reported. The specific code is as follows:

```
pub fn load(input: File) -> ParseProto {
    let mut constants = Vec::new();
    let mut byte_codes = Vec::new();
    let mut lex = Lex::new(input);

    loop {
        match lex.next() {
            Token::Name(name) => { // `Name LiteralString` as function call
                constants.push(Value::String(name));
                byte_codes.push(ByteCode::GetGlobal(0, (constants.len()-1) as
u8));

                if let Token::String(s) = lex.next() {
                    constants.push(Value::String(s));
                    byte_codes.push(ByteCode::LoadConst(1,
(constants.len()-1) as u8));
                    byte_codes.push(ByteCode::Call(0, 1));
                } else {
                    panic!("expected string");
                }
            }
            Token::Eos => break,
            t => panic!("unexpected token: {t:?}"),
        }
    }

    dbg!(&constants);
    dbg!(&byte_codes);
    ParseProto { constants, byte_codes }
}
```

The input is the source file `File`, and the output is the `ParseProto` just defined.

The main body of the function is a loop, and the Token is cyclically read through the `next()` function provided by the lexical analyzer `lex` created at the beginning of the function. We currently only support one type of statement, `Name LiteralString`, and the semantics are function calls. So the analysis logic is also very simple:

- When `Name` is encountered, it is considered to be the beginning of a statement:
    - Use `Name` as a global variable and store it in the constant table;
    - Generate `GetGlobal` bytecode, load the global variable on the stack according to the name. The first parameter is the index of the target stack. Since we currently only support the function call statement, the stack is only used for function calls, so the function must be at position 0; the second parameter is the index of the global variable name in the global variable;
    - Read the next Token, and it is expected to be a string constant, otherwise panic;
    - Add string constants to the constant table;
    - Generate `LoadConst` bytecode to load constants onto the stack. The first parameter is the target stack index, which is behind the function and is 1; the

    second parameter is the index of the constant in the constant table;
  - Once the function and parameters are ready, `Call` bytecode can be generated to call the function. At present, the two parameters are the function position and the number of parameters, which are fixed at 0 and 1 respectively.
- When `Eos` is encountered, exit the loop.
- When encountering other Tokens (currently only of `Token::String` type), panic.

After the function, the constant table and bytecode sequence are output through `dbg!` for debugging. It can be compared with the output of `luac` .

Finally returns `ParseProto` .

Virtual Machine Execution

After parsing and generating `ParseProto` , it is the turn of the virtual machine to execute. According to the previous analysis, the virtual machine currently requires two components: the stack and the global variable table. So define the virtual machine state as follows:

```
pub struct ExeState {
    globals: HashMap<String, Value>,
    stack: Vec::<Value>,
}
```

When creating a virtual machine, you need to add the `print` function in the global variable table in advance:

```
impl ExeState {
    pub fn new() -> Self {
        let mut globals = HashMap::new();
        globals.insert(String::from("print"), Value::Function(lib_print));

        ExeState {
            globals,
            stack: Vec::new(),
        }
    }
```

The `print` function is defined as follows:

```
// "print" function in Lua's std-lib.
// It supports only 1 argument and assumes the argument is at index:1 on
stack.
fn lib_print(state: &mut ExeState) -> i32 {
    println!("{:?}", state.stack[1]);
    0
}
```

Currently the `print` function only supports one parameter, and it is assumed that this parameter is at position 1 of the stack. The function prints this parameter. Because this function does not need to return data to the caller, it returns 0.

After the initialization is completed, the following is the core virtual machine execution function, that is, the big bytecode dispatching loop: read the bytecode sequence in turn and execute the corresponding predefined subrotines. The specific code is as follows:

```
    pub fn execute(&mut self, proto: &ParseProto) {
        for code in proto.byte_codes.iter() {
            match *code {
                ByteCode::GetGlobal(dst, name) => {
                    let name = &proto.constants[name as usize];
                    if let Value::String(key) = name {
                        let v =
self.globals.get(key).unwrap_or(&Value::Nil).clone();
                        self.set_stack(dst, v);
                    } else {
                        panic!("invalid global key: {name:?}");
                    }
                }
                ByteCode::LoadConst(dst, c) => {
                    let v = proto.constants[c as usize].clone();
                    self.set_stack(dst, v);
                }
                ByteCode::Call(func, _) => {
                    let func = &self.stack[func as usize];
                    if let Value::Function(f) = func {
                        f(self);
                    } else {
                        panic!("invalid function: {func:?}");
                    }
                }
            }
        }
    }
```

Currently only 3 bytecodes are supported. All subrotines are clear, needless to explain.

Test

So far, we have implemented a Lua interpreter with a complete process! Look at the running effect:

```
$ cargo r -q --test_lua/hello.lua
[src/parse.rs:39] &constants = [
     print,
     hello, world!,
]
[src/parse.rs:40] &byte_codes = [
     GetGlobal(
         0,
         0,
     ),
     LoadConst(
         1,
         1,
     ),
     Call(
         0,
     ),
]
hello world!
```

The output is divided into 3 parts. Part 1 is the constant table, containing 2 string constants. The second part is the bytecode sequence, which can be compared with the output of `luac` in the Bytecode section. The last line is the result we expected: "hello, world!".

There is an additional function. The parsing part does not support only one line statement, but a loop. So we can support multiple `print` statements, such as:

```
print "hello, world!"
print "hello, again..."
```

There is a small problem which is `print` appears twice in the constant table. It can be optimized here that every time adding a value to the constant table, check whether it already exists first. We will finish this in the next chapter.

Summary

The purpose of this chapter is to implement a minimal Lua interpreter but with complete process, in order to get familiar with the interpreter architecture. To this end, we first introduced the basics of compiling principles, then introduced the two core concepts of Lua's bytecode and value, and finally accomplished it!

We have been emphasizing the "complete process" because we only need to add features onto this framework in the following chapters. Let's move on!

Variables and Assignment

In the last chapter, we completed a simple Lua interpreter but with a complete process. In the future, we will continue to add new features based on this interpreter.

This chapter begins by adding some simple types, including boolean, integer, and floating-point number. Then it introduces local variables.

More Types

This section adds simple types, including boolean, integer, and float. Other types such as Table and UserData will be implemented in subsequent chapters.

We first improve the lexical analysis to support the tokens corresponding to these types, and then generate the corresponding bytecodes through the syntax analysis, and add support for these bytecodes in the virtual machine. Finally we modify the function call to support printing these types.

Improve Lexical Analysis

The lexical analysis in the previous chapter only supports 2 tokens. So now no matter what features are added, the lexical analysis must be changed first to add the corresponding Tokens. In order to avoid adding tokens piecemeal in each chapter in the future, it is now added here in one go.

The Lua official website lists the complete lexical conventions. It includes:

- Name, which has been implemented before, is used for variables, etc.
- Constants, including string, integer, and floating-point constants.
- Keywords:

```
and break do else else if end
false for function goto if in
local nil not or repeat return
then true until while
```

- Symbols:

```
+ - * / % ^ #
& ~ | << >> //
== ~= <= >= < > =
( ) { } [ ] ::
; : , . .. ...
```

The corresponding Token is defined as:

```
#[derive(Debug, PartialEq)]
pub enum Token {
    // keywords
    And,    Break,  Do,     Else,   Elseif, End,
    False,  For,    Function, Goto, If,     In,
    Local,  Nil,    Not,    Or,     Repeat, Return,
    Then,   True,   Until,  While,

 // +       -       *       /       %       ^       #
    Add,    Sub,    Mul,    Div,    Mod,    Pow,    Len,
 // &       ~       |       <<      >>      //
    BitAnd, BitXor, BitOr,  ShiftL, ShiftR, Idiv,
 // ==       ~=     <=      >=      <       >        =
    Equal,  NotEq,  LesEq,  GreEq,  Less,   Greater, Assign,
 // (       )       {       }       [       ]       ::
    ParL,   ParR,   CurlyL, CurlyR, SqurL,  SqurR,  DoubColon,
 // ;               :       ,       .       ..      ...
    SemiColon,      Colon,  Comma,  Dot,    Concat, Dots,

    // constant values
    Integer(i64),
    Float(f64),
    String(String),

    // name of variables or table keys
    Name(String),

    // end
    Eos,
}
```

The specific implementation is nothing more than tedious string parsing, which is skipped here. For the sake of simplicity, this implementation only supports most simple types, but does not support complex types such as long strings, long comments, string escapes, hexadecimal numbers, and floating point numbers only support scientific notation Law. These do not affect the main features to be added later.

Type of Values

After lexical analysis supports more types, we add these types to Value:

```rust
#[derive(Clone)]
pub enum Value {
    Nil,
    Boolean(bool),
    Integer(i64),
    Float(f64),
    String(String),
    Function(fn (&mut ExeState) -> i32),
}

impl fmt::Debug for Value {
    fn fmt(&self, f: &mut fmt::Formatter) -> Result<(), fmt::Error> {
        match self {
            Value::Nil => write!(f, "nil"),
            Value::Boolean(b) => write!(f, "{b}"),
            Value::Integer(i) => write!(f, "{i}"),
            Value::Float(n) => write!(f, "{n:?}"),
            Value::String(s) => write!(f, "{s}"),
            Value::Function(_) => write!(f, "function"),
        }
    }
}
```

One of the special places is that the debug mode is used for the output of floating-point numbers: `{:?}`. Because Rust's common output format `{}` for floating-point numbers is integer + decimal format, and a more reasonable way should be to choose a more suitable one between "integer decimal" and "scientific notation", corresponding to `%g` in C language's `printf`. For example, it is unreasonable to output `"0.000000"` for the number `1e-10`. This seems to be a historical issue of Rust. For compatibility and other reasons, only the debug mode `{:?}` can be used to correspond to `%g`. I don't get into it here.

In addition, in order to facilitate the distinction between "integer" and "floating point number without decimal part", in Lua's official implementation, `.0` will be added after the latter. For example, `2` will be output as `2.0` for the floating point number. The code is as follows. This is so sweet. And this is also the default behavior of Rust's `{:?}` mode, so we don't need special handling for this.

```c
     if (buff[strspn(buff, "-0123456789")] == '\0') { /* looks like an int?
*/
       buff[len++] = lua_getlocaledecpoint();
       buff[len++] = '0'; /* adds '.0' to result */
     }
```

Before Lua 5.3, Lua has only one numeric type, the default is floating point. I understand this because Lua was originally intended for configuration files, for users rather than programmers. For ordinary users, the concepts of integer and floating point numbers are not distinguished, and there is no difference between configuring `10 seconds` and `10.0 seconds`; in addition, for some calculations, such as `7/2`, the result is obviously `3.5` and Not `3`. However, with the expansion of Lua's use, for example, as a glue language between many large programs, the demand for integers has become increasingly strong, so integers and floating-point numbers are distinguished at the language level.

Syntax Analysis

Now we add support for these types in the parser. Since currently only the function call statement is supported, that is, the format of `function parameter`; and the "function" only supports global variables, so this time only the "parameter" part needs to support these new types. For function calls in Lua voice, if the parameter is a string constant or a table structure, then the parentheses `()` can be omitted, as in the "hello, world!" example in the previous chapter. But for other cases, such as several new types added this time, brackets `()` must be required. So the modification of the parameter part is as follows:

```
    Token::Name(name) => {
        // function, global variable only
        let ic = add_const(&mut constants, Value::String(name));
        byte_codes.push(ByteCode::GetGlobal(0, ic as u8));

        // argument, (var) or "string"
        match lex. next() {
            Token::ParL => { // '('
                let code = match lex. next() {
                    Token::Nil => ByteCode::LoadNil(1),
                    Token::True => ByteCode::LoadBool(1, true),
                    Token::False => ByteCode::LoadBool(1, false),
                    Token::Integer(i) =>
                        if let Ok(ii) = i16::try_from(i) {
                            ByteCode::LoadInt(1, ii)
                        } else {
                            load_const(&mut constants, 1, Value::Integer(i))
                        }
                    Token::Float(f) => load_const(&mut constants, 1,
    Value::Float(f)),
                    Token::String(s) => load_const(&mut constants, 1,
    Value::String(s)),
                    _ => panic!("invalid argument"),
                };
                byte_codes. push(code);

                if lex.next() != Token::ParR { // ')'
                    panic!("expected `)`");
                }
            }
            Token::String(s) => {
                let code = load_const(&mut constants, 1, Value::String(s));
                byte_codes. push(code);
            }
            _ => panic!("expected string"),
        }
    }
```

This code first parses the function. Like the code in the previous chapter, it still only supports global variables. Then parse the parameters. In addition to the support for string constants, a more general way of parentheses `()` is added. Which handles various type constants:

- Floating-point constants, similar to string constants, call the `load_const()` function, put it in the constant table at compile time, and then load it through `LoadConst` bytecode during execution.

- Nil and Boolean types, there is no need to put Nil, true and false in the constant table. It is more convenient to encode directly into bytecode, and it is faster at execution time (because there is one less memory read). So `LoadNil` and `LoadBool` bytecodes are added.

- Integer constants combine the above two approaches. Because a bytecode has 4 bytes, the opcode occupies 1 byte, the destination address occupies 1 byte, and there are 2 bytes left, which can store the integer of `i16` . Therefore, for numbers in the range of `i16` (this is also a high probability event), it can be directly encoded into the bytecode, and the `LoadInt` bytecode is added for this purpose; if it exceeds the range of `i16` , it is stored in the constant table . This is also the official implementation of Lua for reference. From this we can see Lua's pursuit of performance, it adds a bytecode and process codes in order to reduce a memory access only. We will see many such cases in the future.

Since only the function call statment is currently supported, the function is fixed at the `0` position of the stack during execution, and the parameter is fixed at the `1` position. The target addresses of the above bytecodes are also fixedly filled with `1` .

The main code has been introduced. The definition of the function `load_const()` used to generate `LoadConst` bytecode is listed below:

```
fn add_const(constants: &mut Vec<Value>, c: Value) -> usize {
    constants. push(c)
}

fn load_const(constants: &mut Vec<Value>, dst: usize, c: Value) -> ByteCode {
    ByteCode::LoadConst(dst as u8, add_const(constants, c) as u8)
}
```

Test

So far, the parsing process has completed the support for new types. The rest of the virtual machine execution part just supports the newly added bytecode `LoadInt` , `LoadBool` and `LoadNil` . Skip it here.

Then you can test the following code:

```
print(nil)
print(false)
print(123)
print(123456)
print(123456.0)
```

The output is as follows:

```
[src/parse.rs:64] &constants = [
     print,
     print,
     print,
     print,
     123456,
     print,
     123456.0,
]
byte_codes:
   GetGlobal(0, 0)
   LoadNil(1)
   Call(0, 1)
   GetGlobal(0, 0)
   LoadBool(1, false)
   Call(0, 1)
   GetGlobal(0, 0)
   LoadInt(1, 123)
   Call(0, 1)
   GetGlobal(0, 0)
   LoadConst(1, 1)
   Call(0, 1)
   GetGlobal(0, 0)
   LoadConst(1, 2)
   Call(0, 1)
nil
false
123
123456
123456.0
```

There is a small problem left over from the last chapter, that is, `print` appears many times in the constant table. This needs to be modified to check whether it already exists every time a constant is added.

Add Constants

Modify the `add_const()` function above as follows:

```rust
fn add_const(constants: &mut Vec<Value>, c: Value) -> usize {
     constants.iter().position(|v| v == &c)
         .unwrap_or_else(|| {
             constants. push(c);
             constants.len() - 1
         })
}
```

`constants.iter().position()` positions the index. Its parameter is a closure, which needs to compare two `Value`, for which `Value` needs to be implemented `PartialEq` trait:

```
impl PartialEq for Value {
    fn eq(&self, other: &Self) -> bool {
        // TODO compare Integer vs Float
        match (self, other) {
            (Value::Nil, Value::Nil) => true,
            (Value::Boolean(b1), Value::Boolean(b2)) => *b1 == *b2,
            (Value::Integer(i1), Value::Integer(i2)) => *i1 == *i2,
            (Value::Float(f1), Value::Float(f2)) => *f1 == *f2,
            (Value::String(s1), Value::String(s2)) => *s1 == *s2,
            (Value::Function(f1), Value::Function(f2)) => std::ptr::eq(f1,
f2),
            (_, _) => false,
        }
    }
}
```

Here we think that two integers and floating point numbers that are numerically equal are different, such as `Integer(123456)` and `Float(123456.0)`, because these are indeed two values, and the two cannot be combined when dealing with constant tables value, otherwise in the test code in the previous section, the last line will also load the integer `123456`.

But during Lua execution, these two values are equal, that is, the result of `123 == 123.0` is `true`. We will deal with this issue in a later chapter.

Going back to the `position()` function, its return value is `Option<usize>`, `Some(i)` means found, and returns the index directly; while `None` means not found, you need to add a constant first, and then return the index. According to the programming habit of C language, it is the following if-else judgment, but here we try to use a more functional way. Personally, I feel that this method is not clearer, but since you are learning Rust, try to use the Rust method first.

```
    if let Some(i) = constants.iter().position(|v| v == &c) {
        i
    } else {
        constants. push(c);
        constants.len() - 1
    }
```

After completing the transformation of the `add_const()` function, duplicate values can be avoided in the constant table. The relevant output is intercepted as:

```
[src/parse.rs:64] &constants = [
    print,
    123456,
    123456.0,
]
```

Although the above will check for duplicates when adding constants, the check is done by traversing the array. The time complexity of adding all constants is O(N^2). If a Lua code segment contains a lot of constants, such as 1 million, the parsing will be too slow. For this we need a hash table to provide fast lookups. TODO.

Local Variables

This section describes the definition and access of local variables, while the assignment is covered in the next section.

For the sake of simplicity, we only support the simplified format for defining local variable statements: `local name = expression`, that is to say, it does not support multiple variables or no initialization. We will support the full format in later chapter. The target code is as follows:

```
local a = "hello, local!" -- define new local var 'a'
print(a) -- use 'a'
```

How are local variables managed, stored, and accessed? First refer to the results of `luac`:

```
main <local.lua:0,0> (6 instructions at 0x6000006e8080)
0+ params, 3 slots, 1 upvalue, 1 local, 2 constants, 0 functions
    1   [1]     VARARGPREP      0
    2   [1]     LOADK           0 0     ; "hello, world!"
    3   [2]     GETTABUP        1 0 1   ; _ENV "print"
    4   [2]     MOVE            2 0
    5   [2]     CALL            1 2 1   ; 1 in 0 out
    6   [2]     RETURN          1 1 1   ; 0 out
```

Compared with the program that directly prints "hello, world!" in the previous chapter, there are several differences:

- `1 local` in the second line of the output, indicating that there is 1 local variable. But this is just an illustration and has nothing to do with the following bytecode.
- LOADK, loads the constant at index 0 of the stack. Corresponding to line [1] of the source code, that is, defining local variables. It can be seen that variables are stored on the stack and assigned during execution.
- The target address of GETTABUP is 1 (it was 0 in the previous chapter), that is, `print` is loaded into location 1, because location 0 is used to store local variables.
- MOVE, the new bytecode, is used to copy the value in the stack. The two parameters are the destination index and the source index. Here is to copy the value of index 0 to index 2. It is to use the local variable a as the parameter of print.

After the first 4 bytecodes are executed, the layout on the stack is as follows:

```
  +-----------------+   MOVE
0 | local a         |----\
  +-----------------+    |
1 | print           |    |
  +-----------------+    |
2 | "hello, world!" |<---/
  +-----------------+
  |                 |
```

It can be seen that local variables are stored on the stack during execution. In the previous chapter, the stack was only used for *function calls*, and now it *stores local variables* too. Relatively speaking, local variables are more persistent and only become invalid after the end of the current block. The function call is invalid after the function returns.

Define Local Variables

Now we add support of handling local variables. First define the local variable table `locals`. In the value and type section, it shows that Lua variables only contain variable name information, but no type information, so this table only saves variable names, defined as `Vec<String>`. In addition, this table is only used during syntax analysis, but not needed during virtual machine execution, so it does not need to be added to `ParseProto`.

Currently, 2 statements are supported (2 formats of function calls):

```
Name String
Name ( exp )
```

Among them, `exp` is an expression, which currently supports a variety of constants, such as strings and numbers.

Now we add a new statement, the simplified form of defining a local variable:

```
localName = exp
```

This also includes `exp`. So extract this part as a function `load_exp()`. Then the syntax analysis code corresponding to the definition of local variables is as follows:

```
        Token::Local => { // local name = exp
            let var = if let Token::Name(var) = lex.next() {
                var // can not add to locals now
            } else {
                panic!("expected variable");
            };

            if lex.next() != Token::Assign {
                panic!("expected `=`");
            }

            load_exp(&mut byte_codes, &mut constants, lex.next(), locals.len());

            // add to locals after load_exp()
            locals. push(var);
        }
```

The code is relatively simple and needs no explaination. The `load_exp()` function refers to the following section.

What needs special attention is that when the variable name `var` is first parsed, it cannot be directly added to the local variable table `locals`, but can only be added *after* the expression is parsed. It can be considered that when `var` is parsed, there is no complete definition of local variables; it needs to wait until the end of the entire statement to complete the definition and add it to the local variable table. The following subsections explain the specific reasons.

Access Local Variables

Now access the local variable, that is, the code `print(a)`. That is to increase the processing of local variables in `exp`.

> In fact, in the `Name ( exp )` format of the function call statement in the previous section, you can add global variables in `exp`. In this way, Lua code such as `print(print)` can be supported. It's just that at that time, I only cared about adding other types of constants, and forgot to support global variables. This also reflects the current state, that is, the addition of functional features is all based on feeling, while the completeness or even correctness cannot be guaranteed at all. We will address this issue in subsequent chapters.

So modify the code of `load_exp()` (the processing part of the original various constant types is omitted here):

```rust
fn load_exp(byte_codes: &mut Vec<ByteCode>, constants: &mut Vec<Value>,
        locals: &Vec<String>, token: Token, dst: usize) {

    let code = match token {
        ... // other type consts, such as Token::Float()...
        Token::Name(var) => load_var(constants, locals, dst, var),
        _ => panic!("invalid argument"),
    };
    byte_codes. push(code);
}

fn load_var(constants: &mut Vec<Value>, locals: &Vec<String>, dst: usize,
name: String) -> ByteCode {
    if let Some(i) = locals.iter().rposition(|v| v == &name) {
        // local variable
        ByteCode::Move(dst as u8, i as u8)
    } else {
        // global variable
        let ic = add_const(constants, Value::String(name));
        ByteCode::GetGlobal(dst as u8, ic as u8)
    }
}
```

The processing of variables in the `load_exp()` function is also placed in a separate `load_var()` function, because the "function" part of the previous function call statement can also call this `load_var()` function, so that local variables can also be supported as a function.

The processing logic for variables is to search the Name in the local variable table `locals`,

- if exist, it is a local variable, then generate the `Move` bytecode, which is a new bytecode;
- otherwise, it is a global variable. The handling process was introduced in the previous chapter, so it is skipped here.

> It is foreseeable that after supporting Upvalue, it will also be handled in this function.

When the `load_var()` function looks up variables in the variable table, it searches from the back to the front, that is, the `.rposition()` function is used. This is because we did not check for duplicate names when registering local variables. If there is a duplicate name, it will be registered as usual, that is, it will be pushed at the end of the local variable table. In this case, the reverse search will find the variable registered later, and the variable registered first will never be located. It is equivalent to the variable registered later covering the previous variable. For example, the following code is legal and outputs `456`:

```lua
local a = 123
local a = 456
print(a) -- 456
```

I find this approach very ingenious. If you check if a local variable exists every time adding a local variable, it will definitely consume performance. And this kind of repeated definition of local variables is rare (maybe I am ignorant), and it is not worth checking duplication (whether it is error reporting or reuse) for this small probability situation. The current approach (reverse lookup) not only guarantees performance, but also can correctly support this situation of repeated definitions.

There are similar shadow variables in Rust. However, I guess Rust should not be able to ignore it so simply, because when a variable in Rust is invisible (such as being shadowed), it needs to be dropped, so it is still necessary to specially judge this shadow situation and handle it specially.

Another problem is that as mentioned at the end of the previous paragraph Define Local Variables, when the variable name `var` is parsed, it cannot be directly added to the local variable table `locals`, but must only be added after parsing the expression. At that time, because there was no "access" to the local variable, the specific reason was not explained. Now it can be explained. For example for the following code:

```lua
local print = print
```

This kind of statement is relatively common in Lua code, that is, assign a commonly used "global variable" to a "local variable" with the same name, so that it will be the local variable accessed when this name is referenced later. Local variables are much faster than global variables (local variables are accessed through the stack index, while global variables need to look up the global variable table in real time, which is the difference between the two bytecodes of `Move` and `GetGlobal`), which will improve performance.

Going back to the question just now, if the variable name `print` is just added to the local variable table when it is parsed, then when the expression `print` behind `=` is parsed, the local variable table will find the newly added `print`, then it is equivalent to assigning the local variable `print` to the local variable `print`, and the cycle is meaningless (if you do this, `print` will be assigned the value of nil).

To sum up, variables must be added to the local variable table after parsing the expression behind `=`.

Where the Function is Called

Previously, our interpreter only supported function call statements, so the stack is only a place for function calls. When a function call is executed, the function and parameters are fixed at 0 and 1 respectively. Now that local variables are supported, the stack is not just a place for function calls, and the positions of functions and parameters are not fixed, but need to become the first free position on the stack, that is, the next position of local variables. to this end:

- During syntax analysis, we can get the number of local variables through `locals.len()` , that is, the first free position on the stack.

- When the virtual machine is executing, we need to add a field `func_index` in `ExeState` , set this field before the function call to indicate this position, and use it in the function. The corresponding codes are as follows:

  ```
  ByteCode::Call(func, _) => {
      self.func_index = func as usize; // set func_index
      let func = &self. stack[self. func_index];
      if let Value::Function(f) = func {
          f(self);
      } else {
          panic!("invalid function: {func:?}");
      }
  }
  ```

```
fn lib_print(state: &mut ExeState) -> i32 {
    println!("{:?}", state.stack[state.func_index + 1]); // use func_index
    0
}
```

Test

So far, we have realized the definition and access of local variables, and also re-organized the code in the process, making the previous function call statement more powerful. Both the function and the parameter support global variables and local variables. So the 2-line object code at the beginning of this article is too simple. You can try the following code:

```
local a = "hello, local!" -- define a local by string
local b = a -- define a local by another local
print(b) -- print local variable
print(print) -- print global variable
local print = print --define a local by global variable with same name
print "I'm local-print!" -- call local function
```

Results of the:

```
[src/parse.rs:71] &constants = [
    hello, local!,
    print,
    I'm local-print!,
]
byte_codes:
   LoadConst(0, 0)
   Move(1, 0)
   GetGlobal(2, 1)
   Move(3, 1)
   Call(2, 1)
   GetGlobal(2, 1)
   GetGlobal(3, 1)
   Call(2, 1)
   GetGlobal(2, 1)
   Move(3, 2)
   LoadConst(4, 2)
   Call(3, 1)
hello, local!
function
I'm local-print!
```

In line with expectations! The bytecodes are a bit long, you can compare it with the output of `luac`. We used to be able to analyze and imitate the bytecode sequence compiled by `luac`, but now we can compile and output bytecode independently. Great progress!

OO in Syntax Analysis Code

The feature has been completed. However, with the increase of features, the code in the syntax analysis part becomes more chaotic. For example, the definition of the above `load_exp()` function has a bunch of parameters. In order to organize the code, the syntax analysis is also transformed into an object-oriented model, and methods are defined around `ParseProto`. These methods can get all the information through `self`, so there is no need to pass many parameters. For specific changes, see commit f89d2fd.

Bringing together several independent members also presents a small problem, a problem specific to the Rust language. For example, the original code for reading a string constant is as follows, first call `load_const()` to generate and return the bytecode, and then call `byte_codes.push()` to save the bytecode. These two function calls can be written together:

```
byte_codes.push(load_const(&mut constants, iarg, Value::String(s)));
```

After changing to object-oriented mode, the code is as follows:

```
self.byte_codes.push(self.load_const(iarg, Value::String(s)));
```

But this cannot be compiled, and the error is as follows:

```
error[E0499]: cannot borrow `*self` as mutable more than once at a time
  --> src/parse.rs:70:38
   |
70 |                  self.byte_codes.push(self.load_const(iarg,
Value::String(s)));
   |
---------------------^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^-
   |                 |                |    |
   |                 |                |    second mutable borrow occurs here
   |                 |               first borrow later used by call
   |                 first mutable borrow occurs here
   |
help: try adding a local storing this argument...
  --> src/parse.rs:70:38
   |
70 |                  self.byte_codes.push(self.load_const(iarg,
Value::String(s)));
   |
^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
help: ...and then using that local as the argument to this call
  --> src/parse.rs:70:17
   |
70 |                  self.byte_codes.push(self.load_const(iarg,
Value::String(s)));
   |
^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^

For more information about this error, try `rustc --explain E0499`.
```

Although the Rust compiler is very strict, the error message is still very clear, and even gives correct modification method.

`self` is referenced 2 times by mut. Although `self.byte_codes` is not used in `self.load_const()`, and there is no conflict in fact, the compiler does not know these details. The compiler only knows that `self` is referenced twice. This is the consequence of bringing together multiple members. The solution is to introduce a local variable as suggested by Rust, and then split this line of code into two lines:

```rust
let code = self.load_const(iarg, Value::String(s));
self.byte_codes.push(code);
```

The situation here is simple and easy to fixed, because the returned bytecode `code` is not related to `self.constants`, so it has no connection with `self`, so `self.byte_codes` can be used normally below. If the content returned by a method is still associated with this data structure, the solution becomes not so simple. This situation will be encountered later when the virtual machine is executed.

Variable Assignment

In the program that prints "hello, world!" at the beginning of Chapter 1, we support global variables, namely the `print` function. However, it only supports *access*, but not *assignment* or *creation*. Now the only global variable `print` is manually added to the global variable table when creating a virtual machine. In the previous section, we implemented the definition and access of local variables, but assignment is also not supported. This section will implement *assignment* of global variables and local variables.

The assignment of simple variables is relatively simple, but the complete assignment statement in Lua is very complicated, such as `t[f()] = 123`. Here we first realize the variable assignment, and then briefly introduce the difference between the complete assignment statement.

Combination of Assignments

The variable assignment statements to be supported in this section are expressed as follows:

```
Name = exp
```

The left side of the equal sign `=` (lvalue) currently has two categories, local variables and global variables; the right side is the expression `exp` in the previous chapter, which can be roughly divided into three categories: constants, local variables, and global variables. So this is a 2*3 combination:

- `local = const`, load the constant to the specified location on the stack, corresponding to the bytecode `LoadNil`, `LoadBool`, `LoadInt` and `LoadConst`, etc.
- `local = local`, copy the value on the stack, corresponding to the bytecode `Move`.
- `local = global`, assign the value on the stack to the global variable, corresponding to the bytecode `GetGlobal`.
- `global = const`, to assign a *constant* to a global variable, you need to add the constant to the constant table first, and then complete the assignment through the bytecode `SetGlobalConst`.
- `global = local`, assign *local variable* to global variable, corresponding to bytecode `SetGlobal`.
- `global = global`, assign *global variable* to global variable, corresponding to

bytecode `SetGlobalGlobal`.

Among these 6 cases, the first 3 are assigned to local variables. The `load_exp()` function in the previous section has been implemented and will not be introduced here. The latter three are assigned to global variables, and three new bytecodes are added accordingly. The parameter format of these 3 bytecodes is similar, and they all have 2 parameters, which are:

1. The index of the name of the target global variable in the constant table, similar to the second parameter of the previous `GetGlobal` bytecode. So in these three cases, you need to add the name of the global variable to the constant table first.
2. Source index, the three bytecodes are: the index in the constant table, the address on the stack, and the index of the name of the global variable in the constant table.

The fourth case above, that is `global = const`, handles all constant types with only one bytecode, not like previous local variables which set different bytecodes for some types (such as `LoadNil`, `LoadBool`, etc.). This is because *local variables* are located directly through the index on the stack, and the virtual machine executes its assignment very quickly. If the source data can be inlined into the bytecode and reduce the access to the constant table once, it can be significantly proportional performance improvement. However, accessing *global variables* requires a table lookup, and the execution of the virtual machine is slow. At this time, the performance improvement brought by the inline source data is relatively small, so it is unnecessary. After all, more bytecodes bring more complexity in the parsing and execution stages.

Lexical Analysis

Originally, function calls and local variable definitions were supported, but now variable assignment statements are added. as follows:

```
Name String
Name ( exp )
localName = exp
Name = exp # add new
```

There is a problem here. The newly added *variable assignment* statement also starts with `Name`, which is the same as *function call*. Therefore, based on the indistinguishability of the first token at the beginning, it is necessary to "peek" forward at another token: if it is an equal sign `=`, it is a variable assignment statement, otherwise it is a function call statement. The "peek" here is in quotation marks to emphasize that it is a real *peek* but not *take* the token, because the subsequent statement analysis still needs to use this token. To this end, the lexical analysis also adds a `peek()` method:

```
    pub fn next(&mut self) -> Token {
        if self.ahead == Token::Eos {
            self.do_next()
        } else {
            mem::replace(&mut self.ahead, Token::Eos)
        }
    }

    pub fn peek(&mut self) -> &Token {
        if self.ahead == Token::Eos {
            self.ahead = self.do_next();
        }
        &self.ahead
    }
```

The `ahead` is a newly added field in the `Lex` structure, which is used to save the Token that is parsed from the character stream but cannot be returned. According to the convention of Rust language, this `ahead` should be of type `Option<Token>`, `Some(Token)` means that there is a Token read ahead, and `None` means there is no Token. But for the same reason as `next()` return value type, the `Token` type is directly used here, and `Token::Eos` is used to represent no Read Token in advance.

The original external `next()` function is changed to `do_next()` internal function, which is called by the newly added `peek()` and new `next()` functions.

The newly added `peek()` function returns `&Token` instead of `Token`, because the owner of the Token is still Lex, and it has not been handed over to the caller. Just "lending" it to the caller to "look". If the caller not only wants to "see" but also "change", then `&mut Token` is needed, but we only need to look, and do not need to change. Now that there is `&` borrowing, it involves lifetime in Rust. Since this function has only one input lifetime parameter, that is `&mut self`, according to elision rules, which is given to all output lifetime parameters, the annotation of the lifetime can be omitted below. This default lifetime means to the compiler that the legal cycle of the returned reference `&Token` is less than or equal to the input parameter, namely `&mut self`, that is, `Lex` itself.

> I personally think that the owner of variables, borrowing (reference), and variable borrowing are the core concepts of the Rust language. The concept itself is very simple, but it takes a period of in-depth struggle with the compiler to understand it deeply. The concept of lifetime is based on the above-mentioned core concepts, but it is also slightly more complicated and needs to be understood in practice.

The new `next()` is a simple wrapper for the original `do_next()` function, which handles the Token that may be stored in `ahead` and peeked before: if it exists, it will directly return this Token without calling `do_next()`. But this "direct return" in Rust is not very straightforward. Since `Token` type is not `Copy` (because its `String(String)` type is not

`Copy`), so cannot return directly. The simple solution is to use `Clone` , but the meaning of Clone is to tell us that there is a price to pay, for example, for string type, we need to copy the string content; and we don't need 2 copies of strings, because the Token is returned. After that, we don't need this Token anymore. So the result we need now is: return the Token in `ahead` , and *simultaneously* clean up `ahead` (here naturally set to represent "no" `Token::Eos`). This scene is very similar to the gif of "Raiders of the Lost Ark" that is widely circulated on the Internet (search for "Raiders of the Lost Ark gif" on the internet), and the sandbag in the hand "replaces" the treasure on the mechanism. "Replace" here is a keyword, and this requirement can be fulfilled with the `std::mem::replace()` function in the standard library. This requirement is so common (at least very common in C language projects) that it feels surprised to use such a long-name function to achieve it. But it is precisely because of these restrictions that the security promised by Rust is guaranteed. But if `ahead` is of `Option<Token>` type, then you can use the `take()` method of `Option` , which looks simpler and has exactly the same function.

Syntax Analysis

With the increase of functions, there will be more and more internal codes in the big cycle of syntax analysis, so we first put each statement into an independent function, namely `function_call()` and `local()` , and then add variable assignment statement `assignment()` . The `peek()` function added in the lexical analysis just now is used here:

```rust
fn chunk(&mut self) {
    loop {
        match self. lex. next() {
            Token::Name(name) => {
                if self.lex.peek() == &Token::Assign {
                    self. assignment(name);
                } else {
                    self. function_call(name);
                }
            }
            Token::Local => self. local(),
            Token::Eos => break,
            t => panic!("unexpected token: {t:?}"),
        }
    }
}
```

Then look at the `assignment()` function:

```
    fn assignment(&mut self, var: String) {
        self. lex. next(); // `=`

        if let Some(i) = self. get_local(&var) {
            // local variable
            self. load_exp(i);
        } else {
            // global variable
            let dst = self.add_const(var) as u8;

            let code = match self. lex. next() {
                // from const values
                Token::Nil => ByteCode::SetGlobalConst(dst,
self.add_const(Value::Nil) as u8),
                Token::True => ByteCode::SetGlobalConst(dst,
self.add_const(Value::Boolean(true)) as u8),
                Token::False => ByteCode::SetGlobalConst(dst,
self.add_const(Value::Boolean(false)) as u8),
                Token::Integer(i) => ByteCode::SetGlobalConst(dst,
self.add_const(Value::Integer(i)) as u8),
                Token::Float(f) => ByteCode::SetGlobalConst(dst,
self.add_const(Value::Float(f)) as u8),
                Token::String(s) => ByteCode::SetGlobalConst(dst,
self.add_const(Value::String(s)) as u8),

                // from variable
                Token::Name(var) =>
                    if let Some(i) = self. get_local(&var) {
                        // local variable
                        ByteCode::SetGlobal(dst, i as u8)
                    } else {
                        // global variable
                        ByteCode::SetGlobalGlobal(dst, self.
add_const(Value::String(var)) as u8)
                    }

                _ => panic!("invalid argument"),
            };
            self.byte_codes.push(code);
        }
    }
```

For the case where the lvalue is a local variable, call `load_exp()` to handle it. For the case of global variables, according to the type of the expression on the right, generate `SetGlobalConst`, `SetGlobal` and `SetGlobalGlobal` bytecodes respectively.

Test

Use the following code to test the above six variable assignments:

```lua
local a = 456
a = 123
print(a)
a = a
print(a)
a = g
print(a)
g = 123
print(g)
g = a
print(g)
g = g2
print(g)
```

Execution is as expected. The specific execution results will no longer be posted.

Complete Assignment Statement

The function of the above variable assignment is very simple, but the complete assignment statement of Lua is very complicated. Mainly manifested in the following two places:

First of all, the left side of the equal sign `=` now only supports local variables and global variables, but the assignment of table fields is also supported in the complete assignment statement, such as `t.k = 123`, or the more complex `t[f()+g ()] = 123`. The above `assignment()` function is difficult to add table support. For this reason, it is necessary to add an intermediate expression layer, that is, the `ExpDesc` structure introduced by the subsequent chapter.

Second, the expression following the equal sign `=` is now divided into 3 categories, for 3 bytecodes. If we want to introduce other types of expressions later, such as upvalue, table index (such as `t.k`), or operation results (such as `a+b`), do we have to add a bytecode to each type? The answer is, no. But this will involve some problems that have not been encountered yet, so it is not easy to explain. If not, what needs to be done? This also involves the `ExpDesc` mentioned above.

We will implement Lua's complete assignment statement in the future, and the current assignment code will be completely discarded at that time.

String

Before moving on to improve our interpreter, this chapter pauses to discuss the string type in Lua in detail. In a high-level language like Lua, strings are easy to use; but in a low-level language like Rust, strings are not so simple. Here is a quote from "Rust Programming Language":

> New Rustaceans commonly get stuck on strings for a combination of three reasons: Rust's propensity for exposing possible errors, strings being a more complicated data structure than many programmers give them credit for, and UTF-8. These factors combine in a way that can seem difficult when you're coming from other programming languages.

Implementing and optimizing strings in the Lua interpreter is a great opportunity to explore Rust strings.

Based on the definition of string type, this chapter will also make an important decision: use `Rc` to implement garbage collection.

String Definition

This section does not add new features for now, but stops to discuss and optimize the string type.

Ownership section in the book "Rust Programming Language" introduces the heap using strings as an example of stack and heap, and the relationship with ownership; String section says the strings are complex. We will now use strings to explore Rust's allocation of heaps and stacks, and initially experience the complexity of strings.

Heap and Stack

"Rust Programming Language" uses strings as an example to introduce the concept of heap and stack, and the relationship between stack and ownership. Here is a brief recap. Rust's String consists of two parts:

1. Metadata, generally located on the stack, includes 3 fields: a pointer to the memory block, the length of the string, and the capacity of the memory block. The following are represented by `buffer` , `len` and `cap` respectively.
2. The private memory block used to store the string content is applied on the heap. Owned by the string, so is freed when the string ends. Because of this piece of memory on the heap, String is not `Copy` , which in turn leads to `Value` not being `Copy` . In order to copy `Value` , it can only be defined as `Clone` .

For example, for a string whose content is "hello, world!", the memory layout is as follows. On the left is the metadata on the stack, where `buffer` points to the memory block on the heap, `len` is the length of the string, which is 13, and `cap` is the capacity of the memory block, which is likely to be aligned to 16. On the right is the block of memory on the heap that stores the contents of the string.

```
stack            heap
+--------+
| buffer +-------->+-----------------+
|--------|         |hello, world!    |
| len=13 |         +-----------------+
|--------|
| cap=16 |
+--------+
```

What needs to be explained here is that the metadata "generally" located on the stack above is for simple types. But for complex types, such as `Vec<String>` , the String metadata part is also stored on the heap as the content of the array (similar to the memory block part of String). Below is an array Vec with 2 string members. The metadata of the array itself is on the stack, but the metadata of the string is on the heap.

```
stack              heap
+--------+
| buffer +-------->+-------------+-------------+----
|--------|         | buf|len|cap | buf|len|cap | ...
| len=2  |         +--+----------+--+----------+----
|--------|            |              V
| cap=4  |            V              +-----------------+
+--------+         +---------+       |hello, world!    |
                   |print    |       +-----------------+
                   +---------+
```

In this case, although the metadata array part is on the heap, it still has the characteristics of the stack, including last-in-first-out, fast access through indexes, fixed known size, and no need for management (allocation and free). In fact, the stack of the virtual machine of our Lua interpreter is a similar `Vec<Value>` type. Similarly, although its data is on the heap, it has the characteristics of a stack. The term "stack" has two meanings here: the stack at the Rust level, and the stack at the Lua virtual machine. The latter is on the heap at the Rust level. The "stack" mentioned below in this article is the latter meaning, that is, the stack of the Lua virtual machine. But it doesn't matter if you understand it as a Rust stack.

Use String

Currently the string type of Value uses `String` in the Rust standard library directly:

```rust
#[derive(Clone)]
struct Value {
    String(String),
```

The biggest problem with this definition is that if you want to copy the Value of a string, you must deeply copy the string content, that is, Clone. The following diagram represents the memory layout for copying a string:

```
stack              heap
|        |
+--------+
|t|      |
|-+------|
| buffer +-------->+----------------+
|--------|         |hello, world!   |
| len=13 |         +----------------+
|--------|
| cap=16 |
+--------+
:        :
:        :
+--------+
|t|      |
|-+------|
| buffer +-------->+----------------+
|--------|         |hello, world!   |
| len=13 |         +----------------+
|--------|
| cap=16 |
+--------+
|        |
```

The left side of the figure is the stack of the Lua virtual machine, and each line represents a word. Since we are developing based on a 64-bit system, a word is 8 bytes.

The `t` in line 1 represents the tag of `enum Value`. Since our Value type is less than 256 types, 1 byte can be represented, so t occupies 1 byte. The next 3 lines `buffer`, `len` and `cap` form a Rust standard library String. Each field occupies one word. `buffer` is 8-byte aligned, so there are 7 bytes empty between `t` and this part is unusable. These 4 lines (the rectangle surrounded by four `+` in the figure) constitute a value of string type in total.

> There is no default layout for enums in Rust (although it can be specified). We only list one layout possibility here. This does not affect the discussion in this section.

To deeply copy the string Value, you need to copy the metadata on the stack and the memory block on the heap, which is a great waste of performance and memory. The most straightforward way to solve this problem in Rust is to use `Rc`.

Use Rc<String>

In order to quickly copy the string String, it is necessary to allow multiple owners of the string at the same time. Rust's Rc provides this feature. Encapsulate `Rc` outside String, and only need to update the Rc count when copying. It is defined as follows:

```rust
#[derive(Clone)]
struct Value {
    String(Rc<String>),
```

The memory layout is as follows:

```
     stack              heap
     |         |
     +---------+
     |t|       |
     |-+-------|
     |   Rc    +----+-->+---------+---------+---------+---------+---------+
     +---------+     |   |count=2 | weak=0 | buffer | len=13 | cap=16 |
     :         :     |   +---------+---------+-+-------+---------+---------+
     :         :     |                        |
     +---------+     |                        V
     |t|       |     |                        +------------------+
     |-+-------|     |                        |hello, world!     |
     |   Rc    +----/                         +------------------+
     +---------+
     |         |
```

The `count` and `weak` on the right side of the figure are the packages of `Rc`. Since there are currently 2 Values pointing to this string, `count` is 2.

Using `Rc` directly causes the interpreter to use reference counting to implement garbage collection. The subsection below is devoted to this high-impact decision.

Although this solution solves the problem of copying, it also brings a new problem, that is, accessing the content of the string requires 2 pointer jumps. This wastes memory and affects execution performance. Some optimization schemes are introduced below.

Use Rc<str>

Strings in Lua have a feature that they are read-only! If you want to process the string, such as truncation, connection, replacement, etc., a new string will be generated. Rust's String is designed for mutable strings, so it is a bit wasteful to represent read-only strings. For example, the `cap` field in metadata can be removed, and there is no need to reserve memory for possible modifications. For example, in the above example, the length of "hello, world!" is only 13, but a memory block of 16 is allocated for. In Rust, it is more suitable to represent a read-only string is `&str`, which is the slice of `String`. But `&str` is a reference, and does not have ownership of the string, but needs to be attached to a string. However, it has a reference that is not a string (the reference of a string is `&String`). Intuitively, it should be a reference to `str`. What is `str`? It's as if it never appeared alone.

For example. For code like this:

```
let s = String::from("hello, world!"); // String
let r = s[7..12]; // &str
```

Where `r` is `&str` type, and the memory layout is as follows:

```
    stack             heap
s:  +--------+
    | buffer +-------->+-----------------+
    |--------|        |hello, world!   |
    | len=13 |        +--------^--------+
    |--------|                 |
    | cap=16 |                 |
    +--------+                 |
                               |
r:  +--------+                 |
    | buffer +-----------------/
    |--------|
    | len=5  |
    +--------+
```

Then dereferencing `&str`, what you get is the "world" memory. However, a general reference is an address, but length information is also added here, indicating that `str` includes length information in addition to memory. The length information is not on the original data like String, but follows the reference together. In fact, `str` does not exist independently, it must follow a reference (such as `&str`) or a pointer (such as `Box(str)`). This is dynamic size type.

And `Rc` is also a pointer, so `Rc<str>` can be defined. It is defined as follows:

```
#[derive(Clone)]
struct Value {
     String(Rc<str>),
```

The memory layout is as follows:

```
stack            heap
|        |
+--------+
|t|      |
|-+------|
|   Rc   +----+-->+--------+--------+-------------+
|--------|    |   |count=2 | weak=0 |hello, world!|
| len=13 |    |   +--------+--------+-------------+
+--------+    |
:        :    |
:        :    |
+--------+    |
|t|      |    |
|-+------|    |
|   Rc   +----/
+--------+
| len=13 |
+--------+
|        |
```

Among them, "hello, world!" is the original data, which is encapsulated by Rc. The length information `len=13` is stored on the stack along with `Rc`.

This scheme looks very good! Compared with the `Rc<String>` scheme above, this scheme removes the useless `cap` field, does not need to reserve memory, and also saves a layer of pointer jumps. But this solution also has 2 problems:

First, the content needs to be copied when creating the string value. The previous solution only needs to copy the metadata part of the string, which is only 3 words long. And this scheme should copy the string content to the newly created Rc package. Imagine creating a 1M long string, this copying affects performance a lot.

Secondly, it occupies 2 words of space on the stack. Although the problem was more serious in the earliest scheme of directly using String to occupy 3 characters, it can be understood that our current standard has been improved. At present, other types in Value only occupy a maximum of 1 word (plus tag, a total of 2 words). What can be spoiled is that the types of tables and UserData to be added in the future also only occupy 1 word, so it is a waste to change the size of Value from 2 to 3 just because of the string type. Not only does it take up more memory, but it's also unfriendly to the CPU cache.

The key to these problems is that `len` follows `Rc`, not the data. It would be perfect if we could put `len` on the heap, say between `weak` and "hello, world!" in the picture. This is trivial for C, but Rust doesn't support it. The reason is that `str` is a dynamically sized type. So if you choose a fixed size type, can it be realized? Such as arrays.

Use Rc<(u8, [u8; 47])>

Arrays in Rust have intrinsic size information. For example, the sizes of `[u8; 10]` and `[u8; 20]` are 10 and 20 respectively. This size is known at compile time, and there is no need to store it following the pointer. Two arrays with different lengths are different types, such as `[u8; 10]` and `[u8; 20]` are different types. Therefore, the array is a fixed-size type, which can solve the problem in the previous section, that is, only one word is needed on the stack.

Since it is a fixed length, it can only store strings smaller than this length, so this solution is incomplete and can only be a supplementary solution for performance optimization. However, most of the strings encountered in Lua are very short, at least in my experience, so this optimization is still very meaningful. To do this, we need to define 2 string types, one is a fixed-length array, which is used to optimize short strings, and the other is the previous `Rc<String>` scheme, which is used to store long strings. The first byte of the fixed-length array is used to represent the actual length of the string, so the array can be split into two parts. Let's first assume that an array with a total length of 48 is used (1 byte represents the length, and 47 bytes store the string content), then the definition is as follows:

```
struct Value {
    FixStr(Rc<(u8, [u8; 47])>), // len<=47
    String(Rc<String>), // len>47
```

The memory layout for short strings is as follows:

```
stack              heap
|         |
+---------+
|t|       |
|-+-------|
|   Rc    +----+-->+--------+--------+----------------------------+
+---------+    |   |count=2 | weak=0 |len|hello, world!           |
:         :    |   +--------+--------+----------------------------+
:         :    |
+---------+    |
|t|       |    |
|-+-------|    |
|   Rc    +----/
+---------+
|         |
```

The first byte `len` at the beginning of the array part on the right in the figure indicates the actual length of the following string. The next 47 bytes can be used to store string content.

This solution is the same as `Rc<str>` mentioned above, it needs to copy the string content, so it is not suitable for long strings. This is not a big problem. Originally, this solution was designed to optimize short strings. Then even if it is a short string, the selection of the array length is also critical. If it is very long, the space waste is serious for short strings; if it is very short, the coverage ratio is not high. However, this solution can continue to be optimized, using arrays with multi-level lengths, such as 16, 32, 48, 64, etc. However, this also creates some complications.

In addition, the selection of the array length also depends on the memory management library used by Rust. For example, if we choose the length to be 48, plus the two counting fields encapsulated by Rc of 16 bytes, then the length of the memory block on the right heap in the above figure is 64 bytes, which is a very "regular" length. For example, the memory management library jemalloc manages small memory blocks into lengths of 16, 32, 48, 64, and 128, so the above-mentioned memory application with a total length of 64 is not wasted. If we choose the array length to be 40, the total length of the memory block is 56, and it will still be matched to the category of 64, and 64-56=8 bytes will be wasted. Of course, it is very bad behavior to rely on the specific implementation of other libraries to make decisions, but fortunately, the impact is not great.

Here we choose an array length of 48, that is, only strings with lengths from 0 to 47 can be represented.

Then compare it with the `Rc<String>` scheme to see how the optimization works. First of all, the biggest advantage of this solution is that only one memory allocation is required, and only one pointer jump is required during execution.

Second, compare the allocated memory size. In the `Rc<String>` scheme, you need to apply for 2 blocks of memory: one is the Rc count and string metadata, fixed 2+3=5 words, 40 bytes, according to the memory strategy of jemalloc, it will occupy 48 bytes of memory ; The second is the string content. The memory size is related to the length of the string, and also depends on the memory management strategy of Rust String and the implementation of the underlying library. For example, a string with a length of 1 may occupy 16 bytes of memory; A character string with a length of 47 may occupy 48 bytes or 64 bytes of memory. The two blocks of memory together occupy 64 to 112 bytes, which is greater than or equal to this fixed-length array solution.

Let's look at the next solution along the line of "optimizing short strings".

Use Inline Arrays

Compared with `Rc<String>` , the previous solution reduces one layer of pointer jumps. The following solution goes a step further, directly removing the storage on the heap, and storing the string completely on the stack.

We want the size of the `Value` type to be 2 words, or 16 bytes. One of them is used for tag, and one is used for string length, so there are 14 bytes remaining, which can be used to store strings with a length less than 14. This scheme is also a supplementary scheme, and it must also be used in conjunction with a long string definition. details as follows:

```
// sizeof(Value) - 1(tag) - 1(len)
const INLSTR_MAX: usize = 14;

struct Value {
     InlineStr(u8, [u8; INLSTR_MAX]), // len<=14
     String(Rc<String>), // len>14
```

The short string `InlineStr` is associated with two parameters: the string length of the `u8` type, and the `u8` array with a length of 14, which also makes full use of the 7 bytes behind `t` that have been wasted before. hollow. The long string `String` still uses the `Rc<String>` scheme.

The memory layout for short strings is as follows:

```
     stack
    |        |
    +vv------+
    |tlhello,|
    |--------|
    | world! |
    +--------+
    :        :
    :        :
    +vv------+
    |tlhello,|
    |--------|
    | world! |
    +--------+
    |        |
```

The `t` and `l` pointed to by the arrow `v` on the grid represent the tag and length of 1 byte, respectively. The actual string content spans 2 words. If you draw the stack horizontally, it looks clearer:

```
stack:
    --+-+-+--------------+......+-+-+--------------+--
      |t|l|hello, world! |      |t|l|hello, world! |
    --+------------------+......+------------------+--
```

This solution has the best performance, but the worst ability, and can only handle strings with a length no greater than 14. There are three usage scenarios of string type in Lua: global variable name, table index, and string value. Most of the first two are not larger than 14 bytes, so it should be able to cover most cases.

It can be further optimized by adding another type to store strings with a length of 15.

Since the length is known, one byte originally used to store the length can also be used to store the content of the string. However, the optimization brought by this solution is not obvious and less than the complexity brought, so it is not used. It is defined as follows.

```rust
struct Value {
    InlineStr(u8, [u8; INLSTR_MAX]), // len<=14
    Len15Str([u8; 15]), //len=15
    String(Rc<String>), // len>15
```

Summary and Choice

In this section, we used and analyzed `String`, `Rc<String>`, `Rc<str>`, `Rc<(u8, [u8; 47])>` and inline `(u8, [u8; 14])` and several other schemes. Each has advantages and disadvantages. A reasonable approach is to treat long and short strings differently, use short strings to optimize, and use long strings to cover the bottom line. 3 options are available:

- In order to guarantee the length of the `Value` type as 2 words, only `Rc<String>` can be used for long strings.
- For short strings, the final inlining solution does not use heap memory at all, and the optimization effect is the best.
- The penultimate fixed-length array scheme is a compromise between the above two schemes, which is slightly tasteless. However, there is only one disadvantage, which is to introduce greater complexity, and strings need to deal with 3 types. In the next section, generics are used to shield these three types, which solves this shortcoming.

The final solution is as follows:

```rust
const SHORT_STR_MAX: usize = 14; // sizeof(Value) - 1(tag) - 1(len)
const MID_STR_MAX: usize = 48 - 1;

struct Value {
    ShortStr(u8, [u8; SHORT_STR_MAX]),
    MidStr(Rc<(u8, [u8; MID_STR_MAX])>),
    LongStr(Rc<Vec<u8>>),
```

The original `InlineStr` and `FixStr` both represent specific implementation solutions, and the characteristics of external performance are long and short, so they are renamed `ShortStr`, `MidStr` and `LongStr`, which are more intuitive.

In this way, most cases (short strings) can be processed quickly, and for a small number of cases (long strings), although slow, they can also be processed correctly, and do not affect the overall situation (for example, `Rc<str>` takes up 2 word, directly makes `Value` larger, even if it affects the overall situation), and ultimately improves the overall processing efficiency. This is a very common and effective optimization idea. Our scheme achieves optimization by distinguishing between two sets of definitions, which is a typical example. It would be even more beautiful if this goal can be achieved with only one set of definitions and one set of algorithms without distinguishing definitions. We will encounter such an example later in the syntax analysis of assignment statement.

After distinguishing between long and short strings, it also brings two new problems:

1. When generating the string type `Value`, we need to choose `ShortStr`, `MidStr` or `LongStr` according to the length of the string. This choice should be implemented automatically, not by the caller, otherwise it will be troublesome and may make mistakes. For example, the `self.add_const(Value::String(var))` statement that appears many times in the syntax analysis code needs to be improved.

2. Strings are composed of "characters", but `ShortStr` and `MidStr` are both composed of `u8`, what is the difference? How does `u8` express Unicode correctly? How to deal with illegal characters?

The next few sections discuss these two issues.

Type Conversion

The previous section introduced three string types in the `Value` type. When creating a string type, different types need to be generated according to the length. This judgment should not be handed over to the caller, but should be done automatically. For example, the existing statement:

```
self. add_const(Value::String(var));
```

should be changed to:

```
self.add_const(str_to_value(var));
```

The `str_to_value()` function converts the string `var` into the string type corresponding to `Value` .

`From` trait

This function of converting (or generating) from one type to another is very common, so the `From` and `Into` traits are defined in the Rust standard library for this. These two operations are opposite to each other, and generally only need to implement `From` . The following implements the conversion of the string `String` type to `Value` type:

```
impl From<String> for Value {
    fn from(s: String) -> Self {
        let len = s.len();
        if len <= SHORT_STR_MAX {
            // A string with a length of [0-14]
            let mut buf = [0; SHORT_STR_MAX];
            buf[..len].copy_from_slice(s.as_bytes());
            Value::ShortStr(len as u8, buf)

        } else if len <= MID_STR_MAX {
            // A string with a length of [15-47]
            let mut buf = [0; MID_STR_MAX];
            buf[..len].copy_from_slice(s.as_bytes());
            Value::MidStr(Rc::new((len as u8, buf)))

        } else {
            // Strings with length greater than 47
            Value::LongStr(Rc::new(s))
        }
    }
}
```

Then, the statement at the beginning of this section can be changed to use the `into()` function:

```
    self.add_const(var.into());
```

Generics

So far, the requirements at the beginning of this section have been completed. But since strings can do this, so can other types. And other types of transformations are more intuitive. Listed below are only two conversions from numeric types to the `Value` type:

```
impl From<f64> for Value {
    fn from(n: f64) -> Self {
        Value::Float(n)
    }
}

impl From<i64> for Value {
    fn from(n: i64) -> Self {
        Value::Integer(n)
    }
}
```

Then, adding a numerical type `Value` to the constant table can also pass the `into()` function:

```
    let n = 1234_i64;
    self.add_const(Value::Integer(n)); // old way
    self.add_const(n.into()); // new way
```

This may seem like a bit of an overkill. But if you implement `From` for all types that can be converted to `Value`, then you can put `.into()` inside `add_const()`:

```
    fn add_const(&mut self, c: impl Into<Value>) -> usize {
        let c = c.into();
```

Only the first 2 lines of code of this function are listed here. The following is the original logic of adding constants, which is omitted here.

Look at the second line of code first, put `.into()` inside the `add_const()` function, then there is no need for `.into()` when calling externally. For example, the previous statement of adding strings and integers can be abbreviated as:

```
    self. add_const(var);
    self. add_const(n);
```

Many places in the existing code can be modified in this way, and it will become much clearer, so it is worthwhile to implement the `From` trait for these types.

However, here comes the problem: in the above 2 lines of code, the types of parameters accepted by the two `add_const()` function calls are inconsistent! In the function definition, how to write this parameter type? The answer lies in the definition of the `add_const()` function above: `c: impl Into<Value>`. Its full writing is as follows:

```
    fn add_const<T: Into<Value>>(&mut self, c: T) -> usize {
```

This definition means: the parameter type is `T`, and its constraint is `Into<Value>`, that is, this `T` needs to be able to be converted into `Value`, and no arbitrary type or data structure can be added to the constant table inside.

This is generic in the Rust language! Many books and articles have introduced them very clearly, so we do not introduce generics completely here. In fact, we have used generics very early, such as the definition of the global variable table: `HashMap<String, Value>`. In most cases, some library *defines* types and functions with generics, and we just *use*. And `add_const()` here is *defining* a function with generics. The next section will introduce another generic usage example.

Reverse Conversion

The above is to convert the basic type to `Value` type. But in some cases, the reverse conversion is required, that is, converting the `Value` type to the corresponding base type. For example, the global variable table of the virtual machine is indexed by the string type, and the name of the global variable is stored in the `Value` type constant table, so it is necessary to convert the `Value` type to a string type to be used as an index use. Among them, the read operation and write operation of the global variable table are different, and the corresponding HashMap APIs are as follows:

```
pub fn get<Q: ?Sized>(&self, k: &Q) -> Option<&V> // omit the constraint of K, Q
pub fn insert(&mut self, k: K, v: V) -> Option<V>
```

The difference between reading and writing is that the parameter `k` of the read `get()` function is a reference, while the parameter `k` of the write `insert()` function is the index itself. The reason is also simple, just use the index when reading, but add the index to the dictionary when writing, and consume `k`. So we need to realize the conversion of the `Value` type to the string type itself and its reference, namely `String` and `&String`. But for the latter, we use the more generic `&str` instead. (TODO: should use `AsRef` here)

```
impl<'a> From<&'a Value> for &'a str {
    fn from(v: &'a Value) -> Self {
        match v {
            Value::ShortStr(len, buf) => std::str::from_utf8(&buf[..*len as
usize]).unwrap(),
            Value::MidStr(s) => std::str::from_utf8(&s.1[..s.0 as
usize]).unwrap(),
            Value::LongStr(s) => s,
            _ => panic!("invalid string Value"),
        }
    }
}

impl From<&Value> for String {
    fn from(v: &Value) -> Self {
        match v {
            Value::ShortStr(len, buf) => String::from_utf8_lossy(&buf[..*len
as usize]).to_string(),
            Value::MidStr(s) => String::from_utf8_lossy(&s.1[..s.0 as
usize]).to_string(),
            Value::LongStr(s) => s.as_ref().clone(),
            _ => panic!("invalid string Value"),
        }
    }
}
```

The function names of the two conversion calls here are different, `std::str::from_utf8()` and `String::from_utf8_lossy()`. The former does not take `_lossy` and the latter does. The reason lies in UTF-8, etc., which will be explained in detail when UTF8 is introduced later.

In addition, this reverse conversion may fail, such as converting a string `Value` type to an integer type. But this involves error handling, and we will make modifications after sorting out the error handling in a unified manner. Here still use `panic!()` to handle possible failures.

> After supporting Environment, the global variable table will be re-implemented with Lua table type and Upvalue, then the index will be directly of `Value` type, and the conversion here is no need any more.

In the code executed by the virtual machine, when reading and writing the global variable table, the conversion of the `Value` type to a string is completed through `into()` twice:

```
                ByteCode::GetGlobal(dst, name) => {
                    let name: &str = (&proto.constants[name as
usize]).into();
                    let v =
self.globals.get(name).unwrap_or(&Value::Nil).clone();
                    self.set_stack(dst.into(), v);
                }
                ByteCode::SetGlobal(name, src) => {
                    let name = &proto.constants[name as usize];
                    let value = self.stack[src as usize].clone();
                    self.globals.insert(name.into(), value);
                }
```

Input Type

In the previous section we defined a function with generics. In fact, we "use" more generic types than "define". This chapter discusses another "use" example, which is the input type of the entire interpreter, that is, the lexical analysis module reads the source code.

Currently only reading source code from files is supported, and Rust's file type `std::fs::File` does not even include standard input. The lexical analysis data structure Lex is defined as follows:

```
pub struct Lex {
    input: File,
    // omit other members
```

The method `read_char()` for reading characters is defined as follows:

```
impl Lex {
    fn read_char(&mut self) -> char {
        let mut buf: [u8; 1] = [0];
        self.input.read(&mut buf).unwrap();
        buf[0] as char
    }
```

Here we only focus on the `self.input.read()` call.

Use `Read`

The official implementation of Lua supports two types of input files (including standard input) and strings as source code. According to the idea of Rust generics, the input we want to support may not be limited to *some specific types*, but *a type that supports certain features (ie traits)*. In other words, as long as it is a character stream, you can read characters one by one. This feature is so common that the `std::io::Read` trait is provided in the Rust standard library. So modify the definition of Lex as follows:

```
pub struct Lex<R> {
    input: R,
```

There are two changes here:

- Changed the original `Lex` to `Lex<R>`, indicating that Lex is based on the generic type `R`,
- Change the original field input type `File` to `R`.

Correspondingly, the implementation part should also be changed:

```
impl<R: Read> Lex<R> {
```

Added `<R: Read>`, indicating that the constraint of `<R>` is `Read`, that is, the type R must support the `Read` trait. This is because the `input.read()` function is used in the `read_char()` method.

The `read_char()` method itself does not need to be modified, and the `input.read()` function can still be used normally, but its meaning has changed slightly:

- When the input used the `File` type before, the `read()` function called was a method of the `File` type that implemented the `Read` trait;
- The `read()` function is now called on all types that implement the `Read` trait.

The statement here is rather convoluted, so you can ignore it if you don't understand it.

In addition, generic definitions must be added to other places where Lex is used. For example, the definition of ParseProto is modified as follows:

```
pub struct ParseProto<R> {
    lex: Lex<R>,
```

The parameter of its `load()` method is also changed from `File` to `R`:

```
    pub fn load(input: R) -> Self {
```

`load()` supports `R` just to create `Lex<R>`, and `ParseProto` does not use `R` directly. But `<R>` still needs to be added to the definition of `ParseProto`, which is a bit long-winded. What's more verbose is that if there are other types that need to include `ParseProto`, then `<R>` should also be added. This is called generic type propagate. This problem can be circumvented by defining `dyn`, which will also bring some additional performance overhead. However, here `ParseProto` is an internal type and will not be exposed to other types, so `<R>` in `Lex` is equivalent to only spreading one layer, which is acceptable, and `dyn` will not be adopted.

Now that `Read` is supported, types other than files can be used. Next look at using stdin like and string types.

Use Standard Input

The standard input `std::io::Stdin` type implements the `Read` trait, so it can be used directly. Modify the `main()` function to use standard input:

```
fn main() {
    let input = std::io::stdin(); // standard input
    let proto = parse::ParseProto::load(input);
    vm::ExeState::new().execute(&proto);
}
```

Test source code from standard input:

```
echo 'print "i am from stdin!"' | cargo r
```

Use String

The string type does not directly support the `Read` trait, because the string type itself does not have the function of recording the read position. `Read` can be realized by encapsulating `std::io::Cursor` type, which is used to encapsulates `AsRef<[u8]>` to support recording position. Its definition is clear:

```
pub struct Cursor<T> {
    inner: T,
    pos: u64,
}
```

This type naturally implements the `Read` trait. Modify the `main()` function to use strings as source code input:

```
fn main() {
    let input = std::io::Cursor::new("print \"i am from string!\""); //
string+Cursor
    let proto = parse::ParseProto::load(input);
    vm::ExeState::new().execute(&proto);
}
```

Use BufReader

Reading and writing files directly is a performance-intensive operation. The above implementation only reads one byte at a time, which is very inefficient for file types. This frequent and small amount of file reading operation requires a layer of cache outside. The `std::io::BufReader` type in the Rust standard library provides this functionality. This type naturally also implements the `Read` trait, and also implements the `BufRead` trait using the cache, providing more methods.

I originally defined Lex's input field as `BufReader<R>` type, instead of `R` type above. But later it was found to be wrong, because when `BufReader` reads data, it first reads from the source to the internal cache, and then returns. Although it is very practical for file types, while the internal cache is unnecessary for string types, and there is one more unnecessary memory copy. And also found that the standard input `std::io::Stdin` also has its own cache already, so no need to add another layer. Therefore, `BufReader` is not used inside Lex, but let the caller add it according to the needs (for example, for `File` type).

Let's modify the `main()` function to encapsulate `BufReader` outside the original `File` type:

```
fn main() {
    // omit parameter handling
    let file = File::open(&args[1]).unwrap();

    let input = BufReader::new(file); // encapsulate BufReader
    let proto = parse::ParseProto::load(input);
    vm::ExeState::new().execute(&proto);
}
```

Give Up `Seek`

At the beginning of this section, we only require that the input type supports character-by-character reading. In fact, it is not true, we also require that the read position can be modified, that is, the `Seek` trait. This is what the original `putback_char()` method requires, using the `input.seek()` method:

```
    fn putback_char(&mut self) {
        self.input.seek(SeekFrom::Current(-1)).unwrap();
    }
```

The application scenario of this function is that in lexical analysis, sometimes it is necessary to judge the type of the current character according to the next character. For example, after reading the character `-`, if the next character is still `-`, it is a comment; otherwise it is Subtraction, at this time the next character will be put back into the input source as the next Token. Previously introduced that the same is true for reading Token in syntax analysis, and the current statement type must be judged according to the next Token. At that time, the `peek()` function was added to Lex, which could "peek" at the next Token without consuming it. The `peek()` here and the `putback_char()` above are two ways to deal with this situation. The pseudo codes are as follows:

```
// Method 1: peek()
if input.peek() == xxx then
     input.next() // Consume the peek just now
     handle(xxx)
end

// Method 2: put_back()
if input.next() == xxx then
     handle(xxx)
else
     input.put_back() // plug it back and read it next time
end
```

When using the `File` type before, because the `seek()` function is supported, it is easy to support the `put_back` function later, so the second method is adopted. But now the input has been changed to `Read` type, if `input.seek()` is still used, then the input is also required to have `std::io::Seek` trait constraints. Among the three types we have tested above, the cached file `BufReader<File>` and the string `Cursor<String>` both support `Seek`, but the standard input `std::io::Stdin` does not support it, and there may be other input types that support `Read` but not `Seek` (such as `std::net::TcpStream`). If we add `Seek` constraints here, the road will be narrowed.

Since `Seek` cannot be used, there is no need to use the second method. You can also consider the first method, which is at least consistent with Token's `peek()` function.

The more straightforward approach is to add an `ahead_char: char` field in Lex to save the character peeked to, similar to the `peek()` function and the corresponding `ahead: Token` field. It's simpler to do this, but there's a more general way of doing it in the Rust standard library, using `Peekable`. Before introducing Peekable, let's look at the `Bytes` type it depends on.

Use Bytes

The implementation of the `read_char()` function listed at the beginning of this section is a bit complicated relative to its purpose (reading a character). I later discovered a more abstract method, the `bytes()` method of the `Read` triat, which returns an iterator `Bytes`, and each call to `next()` returns a byte. Modify the Lex definition as follows:

```
pub struct Lex<R> {
     input: Bytes::<R>,
```

Modify the constructor and `read_char()` function accordingly.

```
impl<R: Read> Lex<R> {
     pub fn new(input: R) -> Self {Lex {
             input: input.bytes(), // generate iterator Bytes
             ahead: Token::Eos,
         }
     }
     fn read_char(&mut self) -> char {
         match self.input.next() { // just call next(), simpler
             Some(Ok(ch)) => ch as char,
             Some(_) => panic!("lex read error"),
             None => '\0',
         }
     }
```

The code for `read_char()` does not seem to be reduced here. But its main body is just `input.next()` call, and the rest is the processing of the return value. After the error handling is added later, these judgment processing will be more useful.

Use `Peekable`

The `peekable()` method in the `Bytes` document, which returns the `Peekable` type, is exactly what we need. It based on the iterator, and we can "peek" a piece of data forward. Its definition is clear:

```
pub struct Peekable<I: Iterator> {
     iter: I,
     /// Remember a peeked value, even if it was None.
     peeked: Option<Option<I::Item>>,
}
```

To this end, modify the definition of Lex as follows:

```
pub struct Lex<R> {
     input: Peekable::<Bytes::<R>>,
```

Modify the constructor accordingly, and add the `peek_char()` function:

```
impl<R: Read> Lex<R> {
    pub fn new(input: R) -> Self {
        Lex {
            input: input.bytes().peekable(), // generate iterator Bytes
            ahead: Token::Eos,
        }
    }
    fn peek_char(&mut self) -> char {
        match self. input. peek() {
            Some(Ok(ch)) => *ch as char,
            Some(_) => panic!("lex peek error"),
            None => '\0',
        }
    }
```

Here `input.peek()` is basically the same as `input.next()` above, the difference is that the return type is a reference. This is the same as the reason why the `Lex::peek()` function returns `&Token`, because the owner of the returned value is still input, and it does not move out, but just "peek". But here we are of `char` type, which is Copy, so directly dereference `*ch`, and finally return char type.

Summary

So far, we have completed the optimization of the input type. From the beginning, only the `File` type is supported, and finally the `Read` trait is supported. There is not much content to sort out, but in the process of realization and exploration at the beginning, it took a lot of effort to bump into things. In this process, I also thoroughly figured out some basic types in the standard library, such as `Read`, `BufRead`, `BufReader`, also discovered and learned the `Cursor` and `Peekable` types, and also learned more about the official website documents way of organization. Learning the Rust language by doing is the ultimate goal of this project.

Unicode and UTF-8

The previous sections of this chapter refine the string-related content, clarifying some issues, but also introducing some confusion. For example, the definitions of the three string types in `Value`, some are of `[u8]` type, some are of `String` type:

```
pub enum Value {
    ShortStr(u8, [u8; SHORT_STR_MAX]), // [u8] type
    MidStr(Rc<(u8, [u8; MID_STR_MAX])>), // [u8] type
    LongStr(Rc<String>), // String type
```

Another example is the mixed use of "byte" and "character" in the previous section. The same is true for the lexical analysis code, which reads bytes of type `u8` from the input character stream, but converts them to characters of type `char` via `as`.

```
    fn read_char(&mut self) -> char {
        match self. input. next() {
            Some(Ok(ch)) => ch as char, // u8 -> char
```

The reason these confusions haven't caused problems so far is because our test programs only involve ASCII characters. Problems arise if other characters are involved. For example, for the following Lua code:

```
print "你好"
```

The execution result is wrong:

```
$ cargo r -q --  test_lua/nihao.lua
constants: [print, ä½ å¥½]
byte_codes:
  GetGlobal(0, 0)
  LoadConst(1, 1)
  Call(0, 1)
ä½ å¥½
```

The output is not the expected `你好`, but `ä½ å¥½`. Let's explain the reason for this "garbled code" and fix this problem.

Unicode and UTF-8 Concepts

These two are very general concepts, and only the most basic introduction is given here.

Unicode uniformly encodes most of the characters in the world. Among them, in order to be compatible with the ASCII code, the encoding of the ASCII character set is consistent. For example, the ASCII and Unicode encodings of the English letter `p` are both 0x70, and `U+0070` is written in Unicode. The Unicode encoding of Chinese `你` is `U+4F60`.

Unicode just numbers the character, while how the computer stores it is another matter. The easiest way is to store directly according to the Unicode encoding. Since Unicode currently supports more than 140,000 characters (still increasing), it needs at least 3 bytes to represent, so the English letter `p` is `00 00 70`, and the Chinese `你` is `00 4F 60`. The problem with this method is that 3 bytes are required for the ASCII part, which (for English) is wasteful. So there are other encoding methods, UTF-8 is one of them. UTF-8 is a variable-length encoding. For example, each ASCII character only occupies 1 byte. Here the encoding of the English letter `p` is still 0x70, and it is written as `\x70` according to UTF-8; while each Chinese character occupies 3 bytes, for example, the UTF-8 encoding of Chinese `你` is `\xE4\xBD\xA0`. The more detailed encoding rules of UTF-8 are omitted here. Here are a few examples:

```
char | Unicode | UTF-8
-----+---------+---------------
p    |  U+0070 | \x70
r    |  U+0072 | \x72
你   |  U+4F60 | \xE4\xBD\xA0
好   |  U+597D | \xE5\xA5\xBD
```

Garbled Code

After introducing the coding concepts, let's analyze the reasons for the garbled characters in the Lua test code at the beginning of this section. Use hexdump to view source code files:

```
$ hexdump -C test_lua/nihao.lua
00000000  70 72 69 6e 74 20 22 e4  bd a0 e5 a5 bd 22 0a     |print "......".|
#         p  r  i  n  t     "  |--你---| |--好---| "
```

The last line is the comment I added, indicating each Unicode text. You can see that the encoding of `p` and `你` is consistent with the UTF-8 encoding introduced above. Indicates that this file is UTF-8 encoded. How the file is encoded depends on the text editor and operating system used.

Our current lexical analysis reads "bytes" one by one, so for the Chinese `你`, it is considered by the lexical analysis to be 3 independent bytes, namely `e4`, `bd` and `a0`. Then use `as` to convert to `char`. Rust's `char` is Unicode encoded, so we get 3 Unicode characters. By querying Unicode, we can get these 3 characters are `ä`, `½` and ` ` (the last one is a blank character), which is the first half of the "garbled characters" we encountered at the beginning. The following `好` corresponds to the second half of the garbled characters. The 6 characters represented by these 6 bytes are sequentially pushed into `Token::String` (Rust `String` type), and finally printed out by `println!` . Rust's `String` type is UTF-8 encoded, but this does not affect the output.

Summarize the process of garbled characters:

- The source file is UTF-8 encoded;
- Read byte by byte, at this time UTF-8 encoding has been fragmented;
- Each byte is interpreted as Unicode, resulting in garbled characters;
- Storage and printing.

You can also verify it again through Rust coding:

```
    let s = String::from("print hello"); // Rust's String is UTF-8 encoded, so it can simulate Lua source files
    println!("string: {}", &s); // normal output
    println!("bytes in UTF-8: {:x?}", s.as_bytes()); // View UTF-8 encoding

    print!("Unicode: ");
    for ch in s.chars() { // Read "characters" one by one, check the Unicode encoding
        print!("{:x} ", ch as u32);
    }
    println!("");

    let mut x = String::new();
    for b in s.as_bytes().iter() { // read "bytes" one by one
        x.push(*b as char); // as char, bytes are interpreted as Unicode, resulting in garbled characters
    }
    println!("wrong: {}", x);
```

Click on the upper right corner to run and see the result.

The core of the garbled problem lies in the conversion of "byte" to "char". So there are 2 workarounds:

1. When reading the source code, modify it to read "char" one by one. This solution has bigger problems:

    - The input type of Lex we introduced in the previous section is `Read` trait, which only supports reading by "byte". If you want to read according to the "character char", you need to convert it to the `String` type first, and you need the `BufRead` trait, which has stricter requirements for input, such as `Cursor<T>` encapsulated outside the string. not support.
    - If the source code input is UTF-8 encoding, and finally Rust's storage is also UTF-8 encoding, if it is read according to the Unicode encoding "character char", then it needs two meaningless steps from UTF-8 to Unicode and then to

UTF-8 conversion.
    - There is another most important reason, which will be discussed soon, Lua strings can contain arbitrary data, not necessarily legal UTF-8 content, and may not be correctly converted to "character char" .

2. When reading the source code, still read byte by byte; when saving, it is no longer converted to "character char", but directly saved according to "byte". This makes it impossible to continue to use Rust's `String` type to save, the specific solution is shown below.

It is obvious (it just seems obvious now. I was confused at the beginning, and tried for a long time) should choose the second option.

String Definition

Now let's see the difference between the contents of strings in Lua and Rust languages.

Lua introduction to strings: We can specify any byte in a short literal string. In other words, Lua strings can represent arbitrary data. Rather than calling it a string, it is better to say that it is a series of continuous data, and does not care about the content of the data.

And the introduction of the Rust string `String` type: A UTF-8–encoded, growable string. Easy to understand. Two features: UTF-8 encoding, and growable. Lua's strings are immutable, Rust's are growable, but this distinction is beyond the scope of this discussion. Now the focus is on the former feature, which is UTF-8 encoding, which means that Rust strings cannot store arbitrary data. This can be better observed through the definition of Rust's string:

```
pub struct String {
    vec: Vec<u8>,
}
```

You can see that `String` is the encapsulation of `Vec<u8>` type. It is through this encapsulation that the data in `vec` is guaranteed to be legal UTF-8 encoding, and no arbitrary data will be mixed in. If arbitrary data is allowed, just define the alias `type String = Vec<u8>;` directly.

To sum up, Rust's string `String` is only a subset of Lua string; the Rust type corresponding to the Lua string type is not `String`, but `Vec<u8>` that can store arbitrary data.

Modify the Code

Now that we have figured out the cause of the garbled characters and analyzed the difference between Rust and Lua strings, we can start modifying the interpreter code. The places that need to be modified are as follows:

- The type associated with `Token::String` in lexical analysis is changed from `String` to `Vec<u8>` to support arbitrary data, not limited to legal UTF-8 encoded data.
- Correspondingly, the type associated with `Value::LongStr` is also changed from `String` to `Vec<u8>` . This is consistent with the other two string types ShortStr and MidStr.
- In lexical analysis, the original reading functions `peek_char()` and `read_char()` are changed to `peek_byte()` and `next_byte()` respectively, and the return type is changed from "char" to "byte". It turns out that although the name is `char` , it actually reads "bytes" one by one, so there is no need to modify the function content this time.
- In the code, the original matching character constant such as `'a'` should be changed to a byte constant such as `b'a'` .
- If the original `read_char()` reads to the end, it will return `\0` , because `\0` is considered to be a special character at that time. Now Lua's string can contain any value, including `\0` , so `\0` cannot be used to indicate the end of reading. At this point, Rust's `Option` is needed, and the return value type is defined as `Option<u8>` .

  But this makes it inconvenient to call this function, requiring pattern matching (`if let Some(b) =`) every time to get out the bytes. Fortunately, there are not many places where this function is called. But another function `peek_byte()` is called in many places. It stands to reason that the return value of this function should also be changed to `Option<u8>` , but in fact the bytes returned by this function are used to "look at it", as long as it does not match several possible paths, it can be regarded as No effect. So when this function reads to the end, it can still return `\0` , because `\0` will not match any possible path. If you really read to the end, then just leave it to the next `next_byte()` to process.

  > It is the inconvenience brought by `Option` (it must be matched to get the value) that withdraws its value. In my C language programming experience, the handling of this special case of function return is generallyIt is represented by a special value, such as `NULL` for the pointer type, and `0` or `-1` for the int type. This brings two problems: one is that the caller may not handle this special value, which will directly lead to bugs; the other is that these special values may later become ordinary values (for example, our `\0` this time is a

typical example), then all places that call this function must be modified. Rust's `Option` perfectly solves these two problems.

- In lexical analysis, strings support escape. This part is all boring character processing, and the introduction is omitted here.
- Add `impl From<Vec<u8>> for Value` to convert the string constant in `Token::String(Vec<u8>)` to `Value` type. This also involves a lot of details of Vec and strings, which is very cumbersome and has little to do with the main line. The following two sections will be devoted to it.

Conversion from &str, String, &[u8], Vec to Value

The conversion of `String` and `&str` to `Value` has been implemented before. Now add `Vec<u8>` and `&[u8]` to `Value` conversion. The relationship between these 4 types is as follows:

```
          slice
&[u8] <---------> Vec<u8>
                    ^
                    |encapsulate
          slice     |
&str <---------> String
```

- `String` is an encapsulation of `Vec<u8>`. The encapsulated `Vec<u8>` can be returned by `into_bytes()`.
- `&str` is a slice of `String` (can be considered a reference?).
- `&[u8]` is a slice of `Vec<u8>`.

So `String` and `&str` can depend on `Vec<u8>` and `&[u8]` respectively. And it seems that `Vec<u8>` and `&[u8]` can also depend on each other, that is, only one of them can be directly converted to `Value`. However, this will lose performance. analyse as below:

- Source type: `Vec<u8>` is owned, while `&[u8]` is not.
- Destination type: `Value::ShortStr/MidStr` only needs to copy the string content (into Value and Rc respectively), without taking ownership of the source data. And `Value::LongStr` needs to take ownership of `Vec`.

2 source types, 2 destination types, 4 conversion combinations are available:

```
          | Value::ShortStr/MidStr  | Value::LongStr
----------+-------------------------+----------------------
  &[u8]   | 1. Copy string content  | 2. Create a Vec and allocate memory
  Vec<u8> | 3. Copy string content  | 4. Transfer ownership
```

If we directly implement `Vec<u8>`, and for `&[8]`, first create `Vec<u8>` through `.to_vec()` and then indirectly convert it to `Value`. So for the first case above, only the content of the string needs to be copied, and the Vec created by `.to_vec()` is wasted.

If we directly implement `&[8]`, and for `Vec<u8>`, it is first converted to `&[u8]` by reference and then indirectly converted to `Value`. Then for the fourth case above, it is necessary to convert the reference to `&u[8]` first, and then create a Vec through `.to_vec()` to obtain ownership. One more unnecessary creation.

So for the sake of efficiency, it is better to directly implement the conversion of `Vec<u8>` and `&[u8]` to `Value`. However, maybe the compiler will optimize these, and the above considerations are nothing to worry about. However, this can help us understand the two types `Vec<u8>` and `&[u8]` more deeply, and the concept of ownership in Rust. The final conversion code is as follows:

```
// convert &[u8], Vec<u8>, &str and String into Value
impl From<&[u8]> for Value {
    fn from(v: &[u8]) -> Self {

vec_to_short_mid_str(v).unwrap_or(Value::LongStr(Rc::new(v.to_vec())))
    }
}
impl From<&str> for Value {
    fn from(s: &str) -> Self {
        s.as_bytes().into() // &[u8]
    }
}

impl From<Vec<u8>> for Value {
    fn from(v: Vec<u8>) -> Self {
        vec_to_short_mid_str(&v).unwrap_or(Value::LongStr(Rc::new(v)))
    }
}
impl From<String> for Value {
    fn from(s: String) -> Self {
        s.into_bytes().into() // Vec<u8>
    }
}

fn vec_to_short_mid_str(v: &[u8]) -> Option<Value> {
    let len = v.len();
    if len <= SHORT_STR_MAX {
        let mut buf = [0; SHORT_STR_MAX];
        buf[..len].copy_from_slice(&v);
        Some(Value::ShortStr(len as u8, buf))

    } else if len <= MID_STR_MAX {
        let mut buf = [0; MID_STR_MAX];
        buf[..len].copy_from_slice(&v);
        Some(Value::MidStr(Rc::new((len as u8, buf))))

    } else {
        None
    }
}
```

Reverse Conversion

The conversion from `Value` to `String` and `&str` has been implemented before. Now to add the conversion to `Vec<u8>` . First list the code:

```
impl<'a> From<&'a Value> for &'a [u8] {
    fn from(v: &'a Value) -> Self {
        match v {
            Value::ShortStr(len, buf) => &buf[..*len as usize],
            Value::MidStr(s) => &s.1[..s.0 as usize],
            Value::LongStr(s) => s,
            _ => panic!("invalid string Value"),
        }
    }
}

impl<'a> From<&'a Value> for &'a str {
    fn from(v: &'a Value) -> Self {
        std::str::from_utf8(v.into()).unwrap()
    }
}

impl From<&Value> for String {
    fn from(v: &Value) -> Self {
        String::from_utf8_lossy(v.into()).to_string()
    }
}
```

- Since the three strings of `Value` are all consecutive `u8` sequences, it is easy to convert to `&[u8]` .
- The conversion to `&str` needs to be processed by `std::str::from_utf8()` to handle the `&[u8]` type just obtained. This function does not involve new memory allocation, but only verifies the validity of the UTF-8 encoding. If it is illegal, it will fail, and here we panic directly through `unwrap()` .
- Conversion to `String` , through `String::from_utf8_lossy()` to process the `&[u8]` type just obtained. This function also verifies the legality of UTF-8 encoding, but if the verification fails, a special character `u+FFFD` will be used to replace the illegal data. But the original data cannot be modified directly, so a new string will be created. If the verification is successful, there is no need to create new data, just return the index of the original data. The return type `Cow` of this function is also worth learning.

The different processing methods of the above two functions are because `&str` has no ownership, so new data cannot be created, but an error can only be reported. It can be seen that ownership is very critical in the Rust language.

The conversion from `Value` to `String` , the current requirement is only used when the global variable table needs to be set. You can see that this conversion always calls `.to_string()` to create a new string. This makes the optimization of strings in our chapter (mainly Section 1) meaningless. Later, after introducing the Lua table structure, the index type of the global variable table will be changed from `String` to `Value` , and then the operation of the global variable table will not need this conversion. However, this conversion is still used in other places.

Test

So far, the function of Lua string is more complete. The test code at the beginning of this section can also be output normally. More methods can be handled by escape, and verified with the following test code:

```
print "tab:\thi" -- tab
print "\xE4\xBD\xA0\xE5\xA5\xBD" -- 你好
print "\xE4\xBD" -- invalid UTF-8
print "\72\101\108\108\111" -- Hello
print "null: \0." -- '\0'
```

Summarize

This chapter has learned the Rust string type, which involves ownership, memory allocation, Unicode and UTF-8 encoding, etc., and deeply understands what is said in "Rust Programming Language": Rust strings are complex because the string itself is complex of. Through these learnings, Lua's string type is optimized, and generics and `From` traits are also involved. Although it did not add new features to our Lua interpreter, it also gained a lot.

Garbage Collection and Rc

In the above section String Definition, we used `Rc` to define the string type in Lua, which involves an important topic: garbage collection(GC). Garbage collection is a very general and in-depth topic. Here we only introduce the parts related to our interpreter implementation.

GC vs RC

The Lua language manages memory automatically, meaning it releases unused memory automatically through garbage collection. There are two main ways to implement garbage collection: mark-and-sweep and reference counting (RC). Sometimes RC is not considered GC, so the narrow GC refers to the former, that is, the mark-clear scheme. The GC mentioned below in this section is its narrow meaning.

In comparison, RC has two disadvantages:

- It is impossible to judge circular references, which can lead to memory leaks. This is fatal. In fact, `Rc` in Rust also has this problem. Rust's strategy for this is: it's up to the programmer to avoid circular references.
- Performance is worse than GC. This is not absolute, but it seems to be the mainstream view. The main reason is that each clone or drop operation needs to update the reference counter, which in turn affects the CPU cache.

Based on the above reasons, the mainstream languages will not adopt the RC scheme, but chose the GC scheme, including the official implementation version of Lua. However, we still chose to use `Rc` in the definition of strings in this chapter, that is, to adopt the RC scheme, because of two shortcomings of GC:

- Implement complex. Although it may be relatively simple to implement a simple GC solution, it is very difficult to pursue performance. Many languages (such as Python, Go, Lua) also continuously improve their GC during versions. It's hard to get there in one step.
- More complex in Rust. Originally, the biggest feature of the Rust language is automatic memory management. The manual memory management function of the GC scheme is contrary to this feature of Rust, will make it more complicated. There are many discussions and projects on the Internet about implementing GC with Rust (such as 1, 2, [3](https://manishearth.github.io/blog/2021/04/05/a-tour-of-safe-tracing-gc- designs-in-rust/), [4] (https://coredumped.dev/2022/04/11/implementing-a-safe-garbage-collector-in-rust/), etc.), obviously beyond Rust beginners range of capabilities.

In contrast, if you adopt the RC scheme, you only need to use `Rc` in Rust, and no additional memory management is required. That is, the garbage collection part can be avoided entirely.

Countermeasures for the two shortcomings of the above-mentioned RC scheme: one is circular references, which can only be handed over to Lua programmers to avoid circular references, but in common cases, such as a table setting itself as a metatable, we can handle it specially to avoid memory leaks. The second is performance, and we can only give up the pursuit of this aspect.

It is a difficult decision to adopt the RC scheme to realize garbage collection. Because the goal of this project from the beginning is to fully comply with the Lua manual and be fully compatible with the official implementation version. After adopting the RC scheme, the scenario of circular reference cannot be handled, which destroys this goal. Due to my limited ability, I have to do this for the time being. However, GC solutions may also be tried in the future. Alternative GC schemes have very little impact on the rest of our interpreter. Interested readers can try it out by themselves first.

Rc in Rust

Ok, now let's leave the topic of garbage collection and discuss `Rc` in Rust simply.

In many articles introducing Rust, it is mentioned that `Rc` should be avoided as much as possible, because the unique ownership mechanism of Rust language not only provides automatic memory management at compile time, but also optimizes program design. Other languages that support pointers (such as C, C++) can use pointers to point at will, and each object may be pointed to by many other objects, and the entire program can easily form a chaotic "Object Sea". However, Rust's ownership mechanism forces Rust programmers to have only one owner for each object when designing a program, and the entire program forms a clear "Object Tree". In most scenarios, the latter (Object Tree) is obviously a better design. However, `Rc` breaks this specification, and the whole program becomes a chaotic "Object Sea" again. So try to avoid using `Rc`.

I agree with this point of view very much. In the process of implementing this Lua interpreter project, in order to follow Rust's ownership mechanism, I had to adjust the previous C language design ideas, and the adjusted results were often indeed clearer.

From a certain point of view, the Lua interpreter project can be divided into two parts:

- The interpreter itself, mainly lexical analysis and syntax analysis;
- The Lua code to be interpreted and executed, including the value, stack, and preset execution flow corresponding to the bytecode, which is the virtual machine part.

For the former part, we fully follow the design requirements of Object Tree and strive for a clear program structure. For the latter, since we cannot limit the Lua code written by Lua programmers (for example, Lua code can easily realize the data structure of the graph, which obviously does not conform to Object Tree), so we will not go into this part pursue Object Tree. Even if GC is used to achieve garbage collection, it will inevitably involve a lot of unsafe code, which is more contrary to the design intent of Rust than `Rc`.

Table

This chapter implements the only data structure in Lua: the table. The definition, construction, and reading and writing of tables are completed in sequence.

In order to realize the write operation, that is, the assignment of table members, the key data structure `ExpDesc` is introduced in the syntax analysis stage.

Table Definition

Lua's table is externally represented as a unified hash table, and its index can be a number, a string, or all other Value types except `nil` and `Nan`. However, for performance considerations, there is a special treatment for numeric types, that is, an array is used to store items indexed by consecutive numbers. So in the implementation, the table is actually composed of two parts: an array and a hash table. For this we define the table:

```
pub struct Table {
    pub array: Vec<Value>,
    pub map: HashMap<Value, Value>,
}
```

In order to support the characteristics of the meta table, other fields will be added in the future, which will be ignored here.

The table (and thread, UserData, etc. introduced later) type in the Lua language does not represent the object data itself, but just a reference to the object data, all operations on table types are references to operations. For example, the assignment of a table only copies the reference of the table, rather than "deep copying" the data of the entire table. So the table type defined in `Value` cannot be `Table`, but must be a reference or a pointer. When the string type was defined in the previous chapter, `Rc` was introduced and references and pointers were discussed. For the same reason, the pointer `Rc` is also used to encapsulate `Table` this time. In addition, `RefCell` needs to be introduced here to provide internal mutability. In summary, the table type is defined as follows:

```
pub enum Value {
     Table(Rc<RefCell<Table>>),
```

The definition of the hash table part in `Table` is `HashMap<Value, Value>`, that is, the type of index and value are `Value` both. The index type of `HashMap` is required to implement the two traits `Eq` and `Hash`. This is also easy to understand. The working principle of the hash table is to quickly locate by calculating the hash value (`Hash`) of the index when inserting and searching, and to handle hash conflicts by comparing the index (`Eq`). Next, implement these two traits.

`Eq` trait

We have implemented the `PartialEq` trait for `Value` before, which compares whether two Values are equal, or we can use the `==` operator on the Value type. The requirement of `Eq` is higher, which requires reflexivity on the basis of `PartialEq`, that is, it is required to satisfy `x==x` for any value `x` of this type. In most cases, it is reflexive, but there are also counterexamples. For example, in floating-point numbers, `Nan != Nan`, so although the floating-point type implements `PartialEq`, it does not implement `Eq`. Although our `Value` type includes floating-point numbers, since the Lua language prohibits the use of `Nan` as an index (specifically, we will judge whether the index is Nan when the virtual machine performs a table insertion operation), it can be considered that `Value` type satisfies reflexivity. For types that satisfy reflexivity, we just tell Rust that it does, no special implementation is required:

```
impl Eq for Value {}
```

`Hash` trait

Most of the basic types in Rust have already implemented the `Hash` trait, and we only need to call `.hash()` for each type according to the semantics.

The code to implement the `Hash` trait is as follows:

```
impl Hash for Value {
    fn hash<H: Hasher>(&self, state: &mut H) {
        match self {
            Value::Nil => (),
            Value::Boolean(b) => b.hash(state),
            Value::Integer(i) => i.hash(state),
            Value::Float(f) => // TODO try to convert to integer
                unsafe {
                    mem::transmute::<f64, i64>(*f).hash(state)
                }
            Value::ShortStr(len, buf) => buf[..*len as usize].hash(state),
            Value::MidStr(s) => s.1[..s.0 as usize].hash(state),
            Value::LongStr(s) => s.hash(state),
            Value::Table(t) => Rc::as_ptr(t).hash(state),
            Value::Function(f) => (*f as *const usize).hash(state),
        }
    }
}
```

Many types, such as `bool`, `Rc` pointers, etc., have already implemented the hash method, but the floating-point type `f64` does not. The reason is also because of `Nan`. Here is a detailed [discussion](https: //internals.rust-lang.org/t/f32-f64-should-implement-hash/5436/2). It has been stated in the `Eq` trait section that Lua prohibits the use of Nan as an index, we can ignore Nan and the default floating-point type can be hashed. One way is to treat the floating-point number as a piece of memory for hashing. Here we choose to convert to the simpler integer `i64` for hashing.

This conversion uses the `mem::transmute()` function of the standard library, and this function is `unsafe`. We can clearly know that this conversion is safe (really?), so we can use this `unsafe` with confidence.

> When I first learned the Rust language, I saw that the description of some libraries clearly stated that "unsafe code is not included", and I felt that this is a very proud feature. So when I started this project, I also hoped not to have any unsafe code. But now it seems that unsafe is not a scourge. It may be similar to `goto` in C language. As long as it is used reasonably, it can bring great convenience.

For the string type, the hash needs to be calculated for the string content. For the table type, only the hash of the pointer needs to be calculated, and the contents of the table are ignored. This is because string comparisons are content comparisons, and table comparisons are comparisons of table references.

`Debug` and `Display` traits

Because Rust's matches are exhaustive, so the compiler will remind us to add the Table type in the `Debug` trait:

```
            Value::Table(t) => {
                let t = t.borrow();
                write!(f, "table:{}:{}", t.array.len(), t.map.len())
            }
```

There are 2 lines in the code block. Line 1 uses `borrow()`, which is a dynamic reference to `RefCell` type, to ensure that there are no other variable references. This dynamic reference introduces additional runtime overhead relative to most compile-time checks in Rust.

In the official implementation of Lua, the output format of the table type is the address of the table, which can be used for simple debugging. We have increased the length of the array and hash table parts of the table here, which is more convenient for debugging. In addition, we implement the `Display` trait for Value, which is used for the official output of `print`:

```
impl fmt::Display for Value {
    fn fmt(&self, f: &mut fmt::Formatter) -> Result<(), fmt::Error> {
        match self {
            Value::Table(t) => write!(f, "table: {:?}", Rc::as_ptr(t)),
```

Table Construction

This section describes the construction of tables. The construction supports 3 types: list type, record type, and general type. See the following sample codes respectively:

```
local key = "kkk"
print { 100, 200, 300; -- list style
        x="hello", y="world"; -- record style
        [key]="vvv"; -- general style
}
```

Let's first look at how the official implementation of Lua handles the construction of tables. The output of luac is as follows:

```
$ luac -l test_lua/table.lua

main <test_lua/table.lua:0,0> (14 instructions at 0x600001820080)
0+ params, 6 slots, 1 upvalue, 1 local, 7 constants, 0 functions
    1   [1]     VARARGPREP      0
    2   [1]     LOADK           0 0     ; "kkk"
    3   [2]     GETTABUP        1 0 1   ; _ENV "print"
    4   [2]     NEWTABLE        2 3 3   ; 3
    5   [2]     EXTRAARG        0
    6   [2]     LOADI           3 100
    7   [2]     LOADI           4 200
    8   [2]     LOADI           5 300
    9   [3]     SETFIELD        2 2 3k  ; "x" "hello"
    10  [3]     SETFIELD        2 4 5k  ; "y" "world"
    11  [4]     SETTABLE        2 0 6k  ; "vvv"
    12  [5]     SETLIST         2 3 0
    13  [2]     CALL            1 2 1   ; 1 in 0 out
    14  [5]     RETURN          1 1 1   ; 0 out
```

The bytecodes related to the construction of the table are lines 4 to 12:

- Line 4, NEWTABLE, is used to create a table. There are 3 parameters in total, which are the position of the new table on the stack, the length of the array part, and the part length of the hash table.
- Line 5, I don't understand it, ignore it for now.
- Lines 6, 7, and 8, three LOADIs, respectively load the values 100, 200, and 300 of the array part to the stack for later use.
- Lines 9 and 10, bytecode SETFIELD, insert `x` and `y` into the hash table part respectively.
- Line 11, bytecode SETTABLE, inserts the key into the hash table.
- Line 12, SETLIST, loads the data loaded on the stack in lines 6-8 above, and inserts it into the array at one time.

The stack situation corresponding to the execution of each bytecode is as follows:

```
          |       |        /<--- 9.SETFILED
          +-------+        |<---10.SETFILED
4.NEWTABLE |  { }  |<----+--+<---11.SETTABLE
          +-------+      |
   6.LOADI |  100  |---->|
          +-------+      |12.SETLIST
   7.LOADI |  200  |---->|
          +-------+      |
   8.LOADI |  300  |---->/
          +-------+
          |       |
```

First of all, it can be seen that the table is constructed in real time by inserting members one by one during the execution of the virtual machine. This is a bit beyond my expectation (although I didn't think about the process before). I have previously written code similar to the following:

```lua
local function day_of_week(day)
     local days = {
         "Sunday"=0, "Monday"=1, "Tuesday"=2,
         "Wednesday"=3, "Thursday"=4, "Friday"=5,
         "Saturday"=6,
     }
     return days[day]
end
```

It is natural to put `days` inside the `day_of_week()` function, because this variable is only used inside this function. However, according to the realization of the above table structure, every time this function is called, the table will be constructed in real time, that is, the 7 dates will be inserted into the table. This cost is a bit high (8 string hashes and 1 string are required In comparison, at least 9 bytecodes are required, and there is more than one memory allocation brought about by creating a table). It feels not even as fast as comparing week by week names (an average of 4 string comparisons are required, and 2 bytecodes are compared for a total of 8). A better way is to put the `days` variable outside the function (that is UpValue introduced later), and there is no need to construct a table every time you enter the function, but in this way it is not a good programming practice to put variables inside a function outside. Another approach (not supported by Lua's official implementation) is to construct a table composed of all constants in the parsing stage, and then just quote it later, but this will bring some complexity. No energy to finish by now.

Back to the construction of the table, the processing methods for the array part and the hash table part are different:

- The array part is to first load the values onto the stack in sequence, and finally insert them into the array at one time;
- The hash table part is directly inserted into the hash table each time.

One is batch and one is sequential. The reasons for the different methods are speculated as follows:

- If the array part is also inserted one by one, then inserting certain types of expressions requires 2 bytecodes. For example, for global variables, you need to use `GetGlobal` bytecode to load it on the stack first, and then use a bytecode similar to `AppendTable` to insert into the array, then inserting N values requires at most 2N bytecodes . If you insert in batches, only N+1 bytecodes are needed for N values. So bulk insert is better for the array part.

- As for the hash table part, each piece of data has two values of key and value. If the batch method is also used, 2 bytecodes are required to load both values onto the stack. And if it is inserted one by one, only one bytecode is needed in many cases. For example, the last three items in the above sample code only correspond to one bytecode. In this way, the batch method requires more bytecodes, so inserting one by one is more suitable for the hash table part.

In this section, according to the official Lua implementation method, the following 4 bytecodes are correspondingly added:

```rust
pub enum ByteCode {
    NewTable(u8, u8, u8),
    SetTable(u8, u8, u8), // key is on the stack
    SetField(u8, u8, u8), // key is a string constant
    SetList(u8, u8),
```

However, the two bytecodes in the middle do not support the case where the value is a constant, only the index on the stack is supported. We'll add optimizations to constants in a later section.

Syntax Analysis

After introducing the principle of table construction, let's look at the specific implementation. Look at the syntax analysis section first. The code is very long, but it is just according to the above introduction, the logic is very simple. The code is posted here for reference only, readers who are not interested can skip here.

```
fn table_constructor(&mut self, dst: usize) {
    let table = dst as u8;
    let inew = self.byte_codes.len();
    self.byte_codes.push(ByteCode::NewTable(table, 0, 0)); // Create a new
table

    let mut narray = 0;
    let mut nmap = 0;
    let mut sp = dst + 1;
    loop {
        match self. lex. peek() {
            Token::CurlyR => { // `}`
                self. lex. next();
                break;
            }
            Token::SqurL => { // `[` exp `]` `=` exp, general formula
                nmap += 1;
                self. lex. next();

                self.load_exp(sp); // key
                self.lex.expect(Token::SqurR); // `]`
                self.lex.expect(Token::Assign); // `=`
                self. load_exp(sp + 1); // value

                self.byte_codes.push(ByteCode::SetTable(table, sp as u8, sp
as u8 + 1));
            },
            Token::Name(_) => { // Name `=` exp | Name
                nmap += 1;
                let key = if let Token::Name(key) = self. lex. next() {
                    self. add_const(key)
                };
                if self.lex.peek() == &Token::Assign { // Name `=` exp,
recorded
                    self. lex. next();
                    self. load_exp(sp); // value
                    self.byte_codes.push(ByteCode::SetField(table, key as
u8, sp as u8));
                } else {
                    narray += 1;
                    self.load_exp_with_ahead(sp, Token::Name(key)); // exp,
list

                    sp += 1;
                    if sp - (dst + 1) > 50 { // too many, reset it
                        self.byte_codes.push(ByteCode::SetList(table, (sp -
(dst + 1)) as u8));
                        sp = dst + 1;
                    }
                }
            },
            _ => { // exp, list
                narray += 1;
                self. load_exp(sp);

                sp += 1;
                if sp - (dst + 1) > 50 { // too many, reset it
                    self.byte_codes.push(ByteCode::SetList(table, (sp - (dst
+ 1)) as u8));
                    sp = dst + 1;
                }
            },
        }

        match self. lex. next() {Token::SemiColon | Token::Comma => (),
            Token::CurlyR => break,
            t => panic!("invalid table {t:?}"),
        }
    }

    if sp > dst + 1 {
        self.byte_codes.push(ByteCode::SetList(table, (sp - (dst + 1)) as
u8));
    }

    // reset narray and nmap
    self.byte_codes[inew] = ByteCode::NewTable(table, narray, nmap);
}
```

The `NewTable` bytecode is generated at the beginning of the function, but since the number of members of the array and the hash table is not yet known, the latter two parameters are temporarily filled with 0. And write down the position of this bytecode, and modify the parameters at the end of the function.

The intermediate loop is to traverse all members of the table. There are 3 syntax types in total:

- General type, `[ exp ] = exp`, key and value are both expressions, respectively loaded to the sp and sp+1 positions of the stack through the `load_exp()` function, and then generate `SetTable` bytecode;

- Record type, `Name = exp`, key is Name, which is a string constant, added to the constant table, value is an expression, and finally generates `SetField` bytecode. There is a place here that is related to Rust's ownership mechanism, that is, the `key` obtained by matching the pattern branch `Token::Name(key)` of `match self.lex.peek()` cannot be directly passed through `add_const(* key)` added to the constant table. This is because `peek()` returns not `Token` itself, but a reference to `Token`, which is returned by `self.lex.peek()`, so the associated `self.lex` and `self` are also in the referenced state; calling `self.add_const()` is also a mut reference to `self`, which violates the reference rules. The correct way is to abandon the return value of `peek()`, but call `self.lex.next()` to return Token and re-match. At this time, Rust's inspection is too strict, because the Token reference returned by `self.lex.peek()` does not affect `self.add_const()`. It should be that Rust has no ability to determine that there is no influence between the two.

- List type, `exp`, is loaded to the `sp` position of the stack, and `sp` is updated, waiting for the last `SetList` to perform insertion. But you can't load data on the stack infinitely, because this will cause the stack to reallocate memory all the time, so if the current data on the stack exceeds 50, generate a `SetList` bytecode to clean up the stack.

What needs to be explained here is that when the `Name` is parsed, it may be either a record type or a list type. We need to peek the next Token to distinguish between the two: if the next Token is `=`, it is a record type , otherwise it is tabular. The problem here is that `Name` is already peeked, and the lexical analysis only supports peek one Token because of [using `Peekable`](./ch03-03.read_input.md#use peekable), so it can only Modify the expression parsing function `load_exp()` to support a Token read in advance, and add `load_exp_with_ahead()` function for this purpose. In the entire Lua grammar, there is only one place that needs to look forward to two Tokens.

> This kind of behavior that needs to look forward to two Tokens to determine the expression, I wonder if it is called LL(2)?

Virtual Machine Execution

The following is the virtual machine execution code of the newly added 4 bytecodes, which is also very simple and can be skipped:

```
        ByteCode::NewTable(dst, narray, nmap) => {
            let table = Table::new(narray as usize, nmap as usize);
            self.set_stack(dst, Value::Table(Rc::new(RefCell::new(table))));
        }
        ByteCode::SetTable(table, key, value) => {
            let key = self.stack[key as usize].clone();
            let value = self.stack[value as usize].clone();
            if let Value::Table(table) = &self. stack[table as usize] {
                table.borrow_mut().map.insert(key, value);
            } else {
                panic!("not table");
            }
        }
        ByteCode::SetField(table, key, value) => {
            let key = proto.constants[key as usize].clone();
            let value = self.stack[value as usize].clone();
            if let Value::Table(table) = &self. stack[table as usize] {
                table.borrow_mut().map.insert(key, value);
            } else {
                panic!("not table");
            }
        }
        ByteCode::SetList(table, n) => {
            let ivalue = table as usize + 1;
            if let Value::Table(table) = self.stack[table as usize].clone() {
                let values = self. stack. drain(ivalue .. ivalue + n as usize);
                table.borrow_mut().array.extend(values);
            } else {
                panic!("not table");
            }
        }
```

The first bytecode `NewTable` is very simple and will not be introduced. The latter two bytecodes `SetTable` and `SetField` are similar, and both need to get the mut reference of the table through `borrow_mut()`. The final bytecode `SetList` encounters Rust's ownership problem again, and needs to explicitly call the `clone()` function on the list on the stack to create a pointer to an independent list. If `clone()` is not called, then the `table` variable obtained by the first line `if let` statement matching is a reference to the member on the stack, that is, a reference to the stack, and this reference needs to continue until the third line, so it cannot be released in advance; the second line calling `stack.drain()` needs to obtain the variable reference of the stack, which conflicts with the reference obtained by the `table` variable in the first line. Therefore, `clone()` needs to generate a pointer to an independent table, so that the `table` variable matched in the first line is only a reference to the table, and is separated from the reference to the stack, thereby avoiding conflicts.

The mandatory `clone()` here increases performance consumption, but also avoids potential bugs. For example, the stack location where the table is located may be included in the subsequent `stack.drain()`, so the address becomes invalid, and then the operation of inserting data into the table in the third subsequent line will be abnormal. Of course, in the scenario of `SetList`, the syntax analysis will ensure that the stack location cleaned by `stack.drain()` does not include the table, but the Rust compiler does not know, and there is no guarantee that it will not be included in the future. So `clone()` here completely eliminates this hidden danger, and it is worthwhile.

So far, we have completed the construction of the table, and the following sections will introduce the reading and writing of the table.

ExpDesc Concept

Before introducing the reading and writing of tables, this section first introduces `ExpDesc`, the core data structure of syntax analysis.

Problems with Table Construction

There is a performance issue with the construction of the table implemented in the previous section. For example, for the general type `[ exp ] = exp`, the expression corresponding to the key and value will be loaded to the top of the stack through the `load_exp()` function in turn as a temporary variable; then `SetTable` bytecode will be generated, including The indices of the two temporary variables on top of the stack. code show as below:

```
        Token::SqurL => { // `[` exp `]` `=` exp
            nmap += 1;
            self. lex. next();

            self.load_exp(sp); // load key to the top of the stack
            self.lex.expect(Token::SqurR); // `]`
            self.lex.expect(Token::Assign); // `=`
            self.load_exp(sp + 1); // load value to the top of the stack

            self.byte_codes.push(ByteCode::SetTable(table, sp as u8, sp as u8 +
1));
        },
```

This is wasteful because certain expression types, such as local variables, temporary variables, and constants, can be referenced directly without being loaded on the stack. For example, the following Lua code:

```lua
local k = 'kkk'
local v = 'vvv'
local t = { [k] = v }
```

According to the current implementation, the runtime stack and bytecode sequence are as follows:

```
  +---------+
0 |  "kkk"  |---\[1] Move 3 0
  +---------+   |
1 |  "vvv"  |---|-\[2] Move 4 1
  +---------+   | |
2 |   { }   |<--|-|---------\
  +---------+   | |         |
3 |  "kkk"  |<--/ |   --\   |
  +---------+     |      >--/[3] SetTable 2 3 4
4 |  "vvv"  |<----/   --/
  +---------+
  |         |
```

In fact, neither of these two temporary variables is needed, but only one bytecode is needed: `SetTable 2 0 1`, where the three parameters are the indexes of the table, key, and value on the stack. This is also the way Lua is officially implemented, that is, to directly refer to the index as much as possible, and avoid unnecessary temporary variables. The runtime stack and bytecode sequences are as follows:

```
  +---------+
0 |  "kkk"  |---\
  +---------+    >--\[1] SetTable 2 0 1
1 |  "vvv"  |---/   |
  +---------+       |
2 |   { }   |<------/
  +---------+
  |         |
```

These two methods (whether to introduce temporary variables) correspond to two types of virtual machines: stack-based and register-based.

Stack-based and Register-based

First list the bytecode of the current implementation in the above example:

```
Move 3 0    # Load k to position 3. Now 3 is the top of the stack
Move 4 1    # Load v to position 4. Now 4 is the top of the stack
SetTable 2 3 4
```

In the current implementation, we can be sure that the key and value are to be loaded to the top of the stack, so the first parameter (that is, the target address) in the two `Move` bytecodes can be omitted; in addition, we can also be sure when setting the table, the key and value must be at the top of the stack, so the last two parameters of `SetTable` bytecode can also be omitted. So the bytecode sequence can be simplified as follows:

```
Push 0     # load k to the top of the stack
Push 1     # load v to the top of the stack
SetTable 2 # Use the two values at the top of the stack as key and value,
           # and set them to the table at position 2
```

This method of operating parameters through the top of the stack is called a *stack-based* virtual machine. The virtual machines of many scripting languages such as Java and Python are based on stacks. The method of directly indexing parameters in the bytecode (such as `SetTable 2 0 1`) is called *register-based* virtual machine. The "register" here is not a register in the computer CPU, but a virtual concept. For example, in our Lua interpreter, it is a register implemented by a stack and a constant table. Lua was the first (official virtual machine) register-based mainstream language.

The above is the write statement through the table as an example. Let's introduce another example that is easier to understand, the addition statement (although we have not implemented addition yet, it is indeed easier to understand). For the following Lua code:

```lua
local
local a = 1
local b = 2
r = a + b
```

The bytecode generated by a stack-based virtual machine might look like this:

```
Push 1    # load a to the top of the stack
Push 2    # load b to the top of the stack
Add       # Pop and add the 2 numbers at the top of the stack, and push the
result to the top of the stack
Pop 0     # Pop the result at the top of the stack and assign it to r
```

The bytecode generated by a register-based virtual machine might look like this:

```
Add 0 1 2
```

It can be seen intuitively that the number of bytecode sequences based on the register virtual machine is small, but each bytecode is longer. It is generally believed that the performance of register-based bytecode is slightly better, but the implementation is more complicated. A more detailed description and comparison of these two types is beyond the scope of this article, and I have no ability to introduce them. The reason why I chose register-based in this project is simply because that's what the official Lua implementation does. I didn't know these two ways until I even wrote part of the project. Next, just continue to follow the register-based method instead of entangled with the stack-based method.

One thing to note is that the register-based method is just *trying* to avoid using the temporary variable on the top of the stack. It is also needed when necessary. How to choose a register or a temporary variable will be described in detail later.

Intermediary ExpDesc

Since we want to follow the register-based method, why do we need to load both key and value to the top of the stack and use the stack-based method in the construction of the table in the previous section? It's because we can't implement a register-based approach yet. Now `load_exp()` function directly generates bytecode and loads it to the specified position of the stack after encountering Token. code show as below:

```
    fn load_exp(&mut self, dst: usize) {
        let code = match self. lex. next() {
            Token::Nil => ByteCode::LoadNil(dst as u8),
            Token::True => ByteCode::LoadBool(dst as u8, true),
            Token::False => ByteCode::LoadBool(dst as u8, false),
            Token::Float(f) => self. load_const(dst, f),
            Token::String(s) => self. load_const(dst, s),
            // omit other tokens
        };
        self.byte_codes.push(code);
    }
```

Therefore, when parsing the general-purpose writing statement of the above table, when the expression of key and value is encountered, it is immediately loaded to the top of the stack, and it becomes a stack-based method.

And if we want to realize the register-based method to generate bytecodes such as `SetTable 2 0 1`, when encountering key or value expressions, we cannot generate bytecodes immediately, but need to save them temporarily and wait for the opportunity When it is mature, deal with it according to the situation. Or use the Lua code at the beginning of this section as an example:

```
local k = 'kkk'
local v = 'vvv'
local t = { [k] = v }
```

The table construction statement in line 3 is parsed as follows:

- `[`, determined as a general formula;
- `k`, as a Name, first determine that it is a local variable, the index is 0, and then save it as a key;
- `]` and `=`, as expected;
- `v`, as a Name, is also determined to be a local variable, the index is 1, and then saved as a value;

- At this point, an initialization statement is completed, and the indexes of the key and value saved before are 0 and 1 respectively, so the bytecode `SetTable 2 0 1` can be generated.

The key here is "save the key/value". We are going to add this staging mechanism now. A solution is to directly save the read Token. For example, in this example, the key and value are saved as `Token::Name("k")` and `Token::Name("v")` respectively. But doing so has several problems:

- A small problem is that Name may be a local variable or a global variable. We will see later that the handling of these two variables is different, and `Token::Name` cannot distinguish between these two types.
- The slightly bigger problem is that some expressions are more complex and contain more than one Token, such as `t.k`, `a+1`, `foo()`, etc., which cannot be represented by a Token. To support tables in this chapter, we must support expression statements such as `t.k` or even `t.k.x.y`.
- The bigger problem is that table reads `t.k` can at least be implemented in a stack-based way, but table writes cannot. For example, `t.k = 1` is the left part of the assignment statement. When parsing, it must be saved first, then parse the rvalue expression, and finally execute the assignment. To support the write statement of the table, it is necessary to add this temporary storage mechanism first. This is why this section must be inserted before supporting the read and write functions of the table.

So, we need a new type to hold intermediate results. To this end we introduce `ExpDesc` (the name comes from the official Lua implementation code):

```rust
#[derive(Debug, PartialEq)]
enum ExpDesc {
    Nil,
    Boolean(bool),
    Integer(i64),
    Float(f64),
    String(Vec<u8>),
    Local(usize), // on stack, including local and temporary variables
    Global(usize), // global variable
}
```

Now it seems that its type is the type currently supported by the expression, but `Token::Name` is split into `Local` and `Global`, so introducing this type is a bit of a fuss. But in the next section to support the reading and writing of tables, as well as subsequent statements such as calculation expressions and conditional jumps, ExpDesc will show its talents!

The original parsing process is to directly generate bytecode from Token:

```
Token::Integer -> ByteCode::LoadInt
Token::String -> ByteCode::LoadConst
Token::Name -> ByteCode::Move | ByteCode::GetGlobal
...
```

Now that the ExpDesc layer is added in the middle, the parsing process becomes:

```
Token::Integer -> ExpDesc::Integer -> ByteCode::LoadInt
Token::String -> ExpDesc::String -> ByteCode::LoadConst
Token::Name -> ExpDesc::Local -> ByteCode::Move
Token::Name -> ExpDesc::Global -> ByteCode::GetGlobal
...
```

Syntax Analysis and ExpDesc

ExpDesc is very important, and I will introduce it from another angle here.

Section 1.1 The general compilation process is introduced in the basic compilation principle:

```
               Lexical Analysis      Syntax Analysis      Semantic Analysis
Character Stream --------> Token Stream --------> Syntax Tree -------->
Intermediate Code ...
```

We still use the above addition code as an example:

```lua
local
local a = 1
local b = 2
r = a + b
```

According to the above general compilation process, for the addition statement in the last line, the syntax analysis will get the syntax tree:

```
      |
      V
      +
     / \
    a   b
```

Then during semantic analysis, first see `+`, and know that this is an addition statement, so you can directly generate bytecode: `Add ? 1 2`. Among them, `?` is the target address of the addition, which is handled by the assignment statement and is ignored here; `1` and `2` are the stack indexes of the two addends respectively.

But our current approach, which is also the official implementation of Lua, omits the "semantic analysis" step, directly generates intermediate code from syntax analysis, and generates code while analyzing. Then when syntactic analysis, you can't have a global perspective like the above-mentioned semantic analysis. For example, for the addition statement `a+b`, when `a` is read, it is not known that it is an addition statement, so it can only be saved first. When `+` is read, it is determined that it is an addition statement, and then the second addend is read, and then bytecode is generated. We introduce an intermediate layer `ExpDesc` for this purpose. So ExpDesc is equivalent to the role of the "syntax tree" in the general process. It's just that the syntax tree is global, and ExpDesc is local, and it is the smallest granularity.

```
             Lexical Analysis           Syntax Analysis
Character stream --------> Token stream ----(ExpDesc)---> intermediate code
...
```

It can be seen intuitively that this method of Lua omits the semantic analysis step, and the speed should be slightly faster, but because there is no global perspective, the implementation is relatively complicated. A more detailed description and comparison of these two approaches is beyond the scope of this article. We choose to follow Lua's official implementation method and choose the method of syntactic analysis to directly generate bytecode.

Summary

This section introduces the concept of ExpDesc and describes its role. In the next section, the existing code will be modified based on ExpDesc.

ExpDesc Rewrite

The previous section introduced the concept of ExpDesc and introduced its function and role. This section is based on ExpDesc to modify the existing code. This transformation does not support new features, but only lays the foundation for the reading and writing functions of the next table and more features to follow.

First of all, the most important thing is the function `load_exp()` for parsing expressions. This function originally generated bytecode directly from Token. Now it needs to be split into two steps: Token to ExpDesc, and generating bytecode from ExpDesc. Then, on this basis, transform the table constructor and variable assignment statement.

exp()

Transform the `load_exp()` function step 1, Token to ExpDesc, create a new `exp()` function, the code is as follows:

```
    fn exp(&mut self) -> ExpDesc {
        match self. lex. next() {
            Token::Nil => ExpDesc::Nil,
            Token::True => ExpDesc::Boolean(true),
            Token::False => ExpDesc::Boolean(false),
            Token::Integer(i) => ExpDesc::Integer(i),
            Token::Float(f) => ExpDesc::Float(f),
            Token::String(s) => ExpDesc::String(s),
            Token::Name(var) => self. simple_name(var),
            Token::CurlyL => self. table_constructor(),
            t => panic!("invalid exp: {:?}", t),
        }
    }

    fn simple_name(&mut self, name: String) -> ExpDesc {
        // search reversely, so new variable covers old one with same name
        if let Some(ilocal) = self.locals.iter().rposition(|v| v == &name) {
            ExpDesc::Local(ilocal)
        } else {
            ExpDesc::Global(self. add_const(name))
        }
    }
```

It is relatively simple, similar to the main structure of the previous `load_exp()` function, or even simpler, that is, several Token types supported by the expression statement are converted into the corresponding ExpDesc. Among them, Name and table construction need further processing. Name is to be distinguished from a local variable or a global variable by the `simple_name()` function. The processing of the table construction branch becomes a lot more reasonable, [before] (./ch04-02.table_constructor.md#other scenes) need to add an ugly `return` in this branch, now because this function does not generate bytes code, so this branch can also end naturally. However, although bytecode is no longer needed, ExpDesc is required, so the table constructor `table_constructor()` needs to return an ExpDesc. Because the newly created table is finally put on the stack, it returns `ExpDesc::Local(i)`. Note that the `ExpDesc::Local` type does not just represent "local variables", but "variables on the stack". The name "Local" is used to be consistent with the official Lua code.

In addition to not generating bytecode, this function has another change compared with `load_exp()`, that is, there is no `dst` parameter. In most cases, it is fine, but there is a problem with the constructor of the table. Because the table construction process is to create a table on the stack first, the bytecode generated by the subsequent initialization statement needs to bring the index of the table on the stack as a parameter. For example `SetTable 3 4 5`, the first parameter is the index of the table on the stack. So the original `table_constructor()` function needs a `dst` parameter. Now there is no such parameter, what should I do? We can assume that all table constructions create new tables at the top of the stack. So it is necessary to maintain the current top position of the stack.

Stack Top `sp`

To maintain the current stack top position, first add `sp` indicating the current stack top in `ParseProto`. In the past, the current position of the top of the stack was calculated in real time wherever it was needed, but now it is changed to a global variable, and many places are suddenly coupled. Later, as the characteristics increase, this coupling will become larger and larger, and it will become more and more out of control. But it is too cumbersome to pass the top position of the stack through parameters. In comparison, it is more convenient to maintain a global stack top delegate, but be careful.

The stack has three functions: function calls, local variables, and temporary variables. The first two have specific statements (function call statements and local variable definition statements) for specific processing. The last one, temporary variables, are used in many places, such as the table construction statement mentioned above, so they need to be carefully managed when they are used, and they cannot affect each other. In addition, because local variables also occupy the stack space, before parsing a statement each time, the value of sp on the top of the stack is initialized to the number of current local variables, which is where temporary variables are allowed to be used.

Let's look at the use of the `sp` in the table constructor `table_constructor()`:

```
    fn table_constructor(&mut self) -> ExpDesc {
        let table = self.sp; // Create a new table at the top of the stack
        self.sp += 1; // update sp, if subsequent statements need temporary
variables, use the stack position behind the table

        // omit intermediate construction code

        self.sp = table + 1; // Before returning, set the sp on the top of
the stack, keep only the newly created table, and clean up other temporary
variables that may be used during construction
        ExpDesc::Local(table) // return the type of the table (temporary
variable on the stack) and the position on the stack
    }
```

Use the `sp` at the beginning of the function to replace the `dst` parameter passed in the previous version as the location of the new table. Before the function ends, reset the top position of the stack. In the following subsections, we will continue to introduce the use of the `sp` of the stack when this function actually builds the table.

discharge()

The second step of transforming the `load_exp()` function is from ExpDesc to bytecode. In fact, it is more accurate to say that ExpDesc is loaded onto the stack. We use the function name discharge in the official Lua code to represent "loading".

```
    // discharge @desc into @dst, and set self.sp=dst+1
    fn discharge(&mut self, dst: usize, desc: ExpDesc) {
        let code = match desc {
            ExpDesc::Nil => ByteCode::LoadNil(dst as u8),
            ExpDesc::Boolean(b) => ByteCode::LoadBool(dst as u8, b),
            ExpDesc::Integer(i) =>
                if let Ok(i) = i16::try_from(i) {
                    ByteCode::LoadInt(dst as u8, i)
                } else {
                    self. load_const(dst, i)
                }
            ExpDesc::Float(f) => self. load_const(dst, f),
            ExpDesc::String(s) => self. load_const(dst, s),
            ExpDesc::Local(src) =>
                if dst != src {
                    ByteCode::Move(dst as u8, src as u8)
                } else {
                    return;
                }
            ExpDesc::Global(iname) => ByteCode::GetGlobal(dst as u8, iname
as u8),
        };
        self.byte_codes.push(code);
        self.sp = dst + 1;
    }
```

This function is also very simple. Generate the corresponding bytecode according to ExpDesc, and discharge the expression statement represented by ExpDesc onto the stack. Note that the last line of this function updates the top position of the stack to the next position of dst. In most cases, it is as expected. If it is not as expected, the caller needs to update the top position of the stack after the function returns.

In addition to this most basic function, there are several helper functions. The `discharge()` function is to force the expression discharge to the dst position of the stack. But sometimes you just want to discharge the expression on the stack. If the expression is already on the stack, such as `ExpDesc::Local` type, then you don’t need to discharge it. A new function `discharge_if_need()` is introduced for this purpose. In most cases, it doesn't even care where it is loaded, so create a new function `discharge_top()`, using the top position of the stack. The two function codes are as follows:

```
    // discharge @desc into the top of stack, if need
    fn discharge_top(&mut self, desc: ExpDesc) -> usize {
        self.discharge_if_need(self.sp, desc)
    }

    // discharge @desc into @dst, if need
    fn discharge_if_need(&mut self, dst: usize, desc: ExpDesc) -> usize {
        if let ExpDesc::Local(i) = desc {
            i // no need
        } else {
            self.discharge(dst, desc);
            dst
        }
    }
```

In addition, the `discharge_const()` function is added, and several constant types are added to the constant table, and other types are discharged as needed. This function will be used in the construction and assignment statements of the following tables:

```
    // for constant types, add @desc to constants;
    // otherwise, discharge @desc into the top of stack
    fn discharge_const(&mut self, desc: ExpDesc) -> ConstStack {
        match desc {
            // add const
            ExpDesc::Nil => ConstStack::Const(self.add_const(())),
            ExpDesc::Boolean(b) => ConstStack::Const(self.add_const(b)),
            ExpDesc::Integer(i) => ConstStack::Const(self.add_const(i)),
            ExpDesc::Float(f) => ConstStack::Const(self.add_const(f)),
            ExpDesc::String(s) => ConstStack::Const(self.add_const(s)),

            // discharge to stack
            _ => ConstStack::Stack(self.discharge_top(desc)),
        }
    }
```

After completing the `exp()` and `discharge()` functions, the previous `load_exp()` function can be combined with these two new functions:

```
     fn load_exp(&mut self) {
         let sp0 = self.sp;
         let desc = self. exp();
         self. discharge(sp0, desc);
     }
```

At the end of this chapter, the parsing of expressions in the parsing process will directly call a series of functions of `exp()` and discharge, instead of calling the `load_exp()` function.

table_constructor()

After splitting the `load_exp()` function into `exp()` and `discharge()`, the constructor of the table can be transformed. Or take general-purpose initialization as an example, in previous version, the key and value are directly loaded onto the stack, no matter what type. We can now call `exp()` to read the key and value, and then do different processing according to the type. The specific processing method can refer to the official implementation of Lua. There are three bytecodes `SETTABLE`, `SETFIELD` and `SETI`, corresponding to the three types of key variables on the stack, string constants, and small integer constants. In addition, these 3 bytecodes have 1 bit to mark whether the value is a variable or a constant on the stack. There are 3 key types and 2 value types, a total of 3*2=6 situation. Although we can also distinguish between variables and constants on the stack by reserving a bit in value, this will result in only 7bit address space. So we still distinguish variables and constants on the stack by adding bytecode types. It ends up as follows:

```
 value\key | variable      | string constant | small integer constant
-----------+---------------+-----------------+---------------
 variable  | SetTable      | SetField        | SetInt
-----------+---------------+-----------------+---------------
 constant  | SetTableConst | SetFieldConst   | SetIntConst
```

Another rule is that `nil` and `Nan` of floating-point numbers are not allowed to be used as keys. The parsing code for the key is as follows:

```
        let entry = match self. lex. peek() {
            Token::SqurL => { // `[` exp `]` `=` exp
                self. lex. next();

                let key = self.exp(); // read key
                self.lex.expect(Token::SqurR); // `]`
                self.lex.expect(Token::Assign); // `=`

                TableEntry::Map(match key {
                    ExpDesc::Local(i) => // variables on the stack
                        (ByteCode::SetTable, ByteCode::SetTableConst, i),
                    ExpDesc::String(s) => // string constant
                        (ByteCode::SetField, ByteCode::SetFieldConst, self.
add_const(s)),
                    ExpDesc::Integer(i) if u8::try_from(i).is_ok() => // small
integer
                        (ByteCode::SetInt, ByteCode::SetIntConst, i as usize),
                    ExpDesc::Nil =>
                        panic!("nil can not be table key"),
                    ExpDesc::Float(f) if f.is_nan() =>
                        panic!("NaN can not be table key"),
                    _ => // For other types, discharge them onto the stack
uniformly and turn them into variables on the stack
                        (ByteCode::SetTable, ByteCode::SetTableConst,
self.discharge_top(key)),
                })
            }
```

The above code handles three types of keys: local variables, string constants, and small integers. In addition, nil and floating-point numbers Nan are prohibited. For other types, they are uniformly discharged to the top of the stack and converted to variables on the stack.

Then parse the value to distinguish between variables and constants on the stack. code show as below:

```
        match entry {
            TableEntry::Map((op, opk, key)) => {
                let value = self.exp(); // read value
                let code = match self. discharge_const(value) {
                    // value is a constant, use opk, such as
`ByteCode::SetTableConst`
                    ConstStack::Const(i) => opk(table as u8, key as u8, i as
u8),

                    // value is not a constant, then discharge to the stack, and
use op, such as `ByteCode::SetTable`
                    ConstStack::Stack(i) => op(table as u8, key as u8, i as u8),
                };
                self.byte_codes.push(code);

                nmap += 1;
                self.sp = sp0;
            }
```

The logic of the above two pieces of code itself is very clear, but the parameter types associated with `TableEntry::Map` are somewhat special. The first piece of code deals with the type of the key, and determines the type of 2 bytecodes, or the tag of `ByteCode`. This tag is to be used as an associated parameter of `TableEntry::Map`. Then what type is that? It must not be `ByteCode`, because the enum type includes not only the tag, but also the associated value. If it is a `ByteCode` type, then it is not `ByteCode::SetTable` but a complete `ByteCode::SetTable(table,key,0)`, that is, first generate a complete bytecode, and then read Modify the bytecode when the value is reached. That would be too complicated.

《Rust Programming Language》 introduces these enums with `()` as initialization syntax, looks like a function call, they are indeed implemented as functions returning an instance constructed from parameters. That is to say `ByteCode::SetTable` can be regarded as a function, and its parameter type is `fn(u8,u8,u8)->ByteCode`. When I read this book for the first time, I was confused by the countless new concepts in it, so I had no impression of reading this sentence at all, and even if I saw it, I couldn't understand it or remember it. When I wrote this project for more than half, I read this book completely again. This time, I understood most of the concepts in it very smoothly, and I can also notice the introduction of function pointers. And it just happened to work, what a find!

Table Read/Write and BNF

After introducing ExpDesc and modifying the existing syntax analysis in the previous two sections, this section implements table reading and writing.

The index of the table in Lua supports two methods, examples are as follows: `t["k"]` and `t.k`, where the latter is a special form of the former. All table read and write operations need to use the table index. Need to add the type of table index in ExpDesc.

The read and write operations of the table itself are not complicated, but it will make other statements suddenly become complicated:

- The read operation of the table may have multiple consecutive levels, such as `t.x.y`, so when parsing the expression, the end cannot be judged immediately, but the next Token needs to be peeked to judge.
- The write operation of the table, that is, the assignment statement. The current assignment statement only supports the assignment of "variables", that is, the lvalue only supports one Token::Name. To add support for table indexes, the handling of lvalues needs to be reimplemented. It is not enough to parse only one Token, but to parse an lvalue. So how is it considered a complete lvalue? For example, not all expressions can be used as lvalues, such as function calls or table constructions.
- Previously, the assignment statement and function call statement were distinguished based on the second Token. If it is an equal sign `=`, it is an assignment statement. Now to support the write operation of the table, such as `t.k = 123`, then the second Token is a dot `.` instead of the equal sign `=`, but it is still an assignment statement. The previous judgment method is invalid. So is there any new way to distinguish between assignment statements and function call statements?

The first read operation problem is easy to solve. The next two questions related to write operations are very difficult. We cannot answer them accurately now, but can only guess the answers. This leads to a bigger problem, that is, the previous syntax analysis is based on guesswork! For example, the format of the definition statement of local variables, etc., are guessed based on the experience of using the Lua language, and cannot guarantee its accuracy and completeness. But it was relatively simple before, so you can make a guess. In addition, in order not to interrupt the rhythm of the entire project, I did not delve into this issue. Now to introduce the reading and writing of the table, the statement becomes complicated, and it is impossible to continue to mix it up by guessing. It is necessary to introduce a formal grammatical description.

BNF

The last chapter of the Lua manual is called: The Complete Syntax of Lua, the content is mainly a set of BNF descriptions. We don't need to know the meaning of the term "BNF", we just need to know that this is a formal grammar description method, where the Lua grammar can be described completely and accurately. The grammatical rules of BNF itself are also very simple, and most of them are clear at a glance. Here are only two:

- `{A}` represents 0 or more A
- `[A]` represents optional 1 A

The code segment of Lua is called `chunk`, so the definition of `chunk` is used as the entry, and several descriptions are listed:

```
chunk ::= block

block ::= {stat} [retstat]

stat ::=  ‘;’ |
     varlist ‘=’ explist |
     functioncall |
     label |
     break |
     goto Name |
     do block end |
     while exp do block end |
     repeat block until exp |
     if exp then block {elseif exp then block} [else block] end |
     for Name ‘=’ exp ‘,’ exp [‘,’ exp] do block end |
     for namelist in explist do block end |
     function funcname funcbody |
     local function Name funcbody |
     local attnamelist [‘=’ explist]
```

It can be obtained from these rules: a `chunk` contains a `block`. A `block` contains zero or more `stat`s and an optional `retstat`. A `stat` has many types of statements. Among them, we have implemented the two statements `functioncall` and `local`, and then implemented the remaining types one by one to complete the entire grammar of Lua (although it is still far from the complete Lua language).

> I don't quite understand what is the difference between `chunk` and `block` here? Why list two separately?

That is to say, we will implement the interpreter according to this set of specifications in the future, and we no longer need to rely on guesswork! Pick a few and compare them with our previous ones, such as local variable definition statements, and you can find that it should support multiple variables and multiple initializations expression, even without an initialization expression. This shows that our previous statement analysis is very imperfect. Later in this section, we will improve the sentences we already support based on BNF. Now find out the rules related to the table index:

```
var ::= Name | prefixexp '[' exp ']' | prefixexp '.' Name

exp ::= nil | false | true | Numeral | LiteralString | '...' | functiondef |
prefixexp | tableconstructor | exp binop exp | unop exp

prefixexp ::= var | functioncall | '(' exp ')'

functioncall ::= prefixexp args | prefixexp ':' Name args
```

At first glance it looks a bit complicated. Take `var` as an example for analysis. Here `var` deduces three cases, the first `Name` is a simple variable, and the latter two are table indexes, which are grammar sugar for general methods and string indexes. It involves `prefixexp` and `exp` . Among them, `exp` is very similar to the `exp()` function we currently implement, but we still lack some situations, which also need to be added later. In addition, `Name` is directly in the `exp()` function, and now it has to be moved to `var` .

Eliminate Left Recursion

There is a big problem here, the above 3 rules are recursively referenced. for example:

- `var` refers to `prefixexp` which refers to `var` ;
- `exp` refers to `prefixexp` which refers to `exp` .

But these two examples are fundamentally different.

For the first example, after bringing in `var` and expanding it, it is

```
prefixexp ::= Name | prefixexp '[' exp ']' | prefixexp '.' Name | prefixexp
args | prefixexp ':' Name args | '(' exp ')'
```

The problem is that the 2nd and 3rd items of the derivation rule start with `prefixexp` both. Then during syntax analysis, for example, if you read a Name, you can match item 1, or items 2 and 3, so it is impossible to judge which rule should be selected. This was a headache. I spent two days on this problem, and tried various solutions but couldn't solve it. Later, I searched the Internet and found the concept of "eliminating left recursion", and I vaguely recalled that this was a compulsory topic in the course of compiling principles. And there is a standard method for elimination: For rules that contain left recursion, they can be expressed as follows:

```
A := Aα | β
```

Then it can be rewritten as follows:

```
A := βA'
A’ := αA’ | ε
```

where `ε` is not matched. This eliminates left recursion. Take the above `prefixexp` as an example, first apply the above standard form, you can get:

```
α = '[' exp ']' | '.' Name | args | ':' Name args
β = Name | '(' exp ')'
```

Then bring in the above rewritten formula to get:

```
prefixexp := ( Name | '(' exp ')' ) A'
A' := ( '[' exp ']' | '.' Name | args | ':' Name args ) A' | ε
```

This way we get rules without left recursion.

And the second example at the beginning of this section, about `exp`, although there are recursive references, but it is not "left" recursion, so there is no such problem.

Read Table and `prefixexp`

The advantage of using BNF rules is that you don't need to think about Lua's grammar, just follow the rules to implement.

After obtaining the above BNF rules, the analysis of prefixexp can be completed:

```
    fn prefixexp(&mut self, ahead: Token) -> ExpDesc {
        let sp0 = self.sp;

        // beta
        let mut desc = match ahead {
            Token::Name(name) => self.simple_name(name),
            Token::ParL => { // `(` exp `)`
                let desc = self.exp();
                self.lex.expect(Token::ParR);
                desc
            }
            t => panic!("invalid prefixexp {t:?}"),
        };

        // A' = alpha A'
        loop {
            match self.lex.peek() {
                Token::SqurL => { // `[` exp `]`
                    self.lex.next();
                    let itable = self.discharge_if_need(sp0, desc);
                    desc = match self.exp() {
                        ExpDesc::String(s) => ExpDesc::IndexField(itable,
self.add_const(s)),
                        ExpDesc::Integer(i) if u8::try_from(i).is_ok() =>
ExpDesc::IndexInt(itable, u8::try_from(i).unwrap()),
                        key => ExpDesc::Index(itable,
self.discharge_top(key)),
                    };

                    self.lex.expect(Token::SqurR);
                }
                Token::Dot => { // .Name
                    self.lex.next();
                    let name = self.read_name();
                    let itable = self.discharge_if_need(sp0, desc);
                    desc = ExpDesc::IndexField(itable, self.add_const(name));
                }
                Token::Colon => todo!("args"), // :Name args
                Token::ParL | Token::CurlyL | Token::String(_) => { // args
                    self.discharge(sp0, desc);
                    desc = self.args();
                }
                _ => { // Epsilon
                    return desc;
                }
            }
        }
    }
```

The first paragraph of code corresponds to `β` mentioned above, namely `Name | '(' exp ')'`.

The loop in the second paragraph corresponds to the above `A' := αA' | ε`, if it matches the `α` part, it is `'[' exp ']' | '.' Name | args | ':' Name args`, then the loop continues after parsing; if there is no match, it corresponds to `ε`, and the loop exits. Here this loop supports many continuous operations, such as `t.f()`, which is a table index followed by a function call. Or more sequential operations like `t.t.t.k` and `f()()()`. If you follow the native method in the previous chapters and make a function as soon as you think of it, it will be difficult to support this kind of continuous operation, it is difficult to realize and it is difficult to think of it. But according to BNF, it can be realized correctly and completely.

Corresponding to the three types of bytecodes in the construction of the table, that is, the key is a variable on the stack, a string constant and a small integer. There are also three types of ExpDesc here, namely `Index`, `IndexField` and `IndexInt`. When discharging, add 3 corresponding bytecodes, `GetTable`, `GetField` and `GetInt`. This naturally solves the first problem at the beginning of this section, that is, the reading operation of the table is realized, and it is implemented correctly and completely!

Another feature of encoding according to the BNF rule is that you can only understand the processing logic inside each matching branch, but not the overall relationship between each branch. This is like solving a physics application problem. First, analyze the physical principles and list the equations, each of which has a corresponding physical meaning; but when solving the equations, the specific solution steps have been completely separated from the physical correspondence, which is a math tools.

The `prefixexp()` function is listed above, and the implementation of the `exp()` function is similar, which is omitted here.

Write Table and Assignment Statement

After implementing `prefixexp` and `exp` according to BNF, the problem about table write operation at the beginning of this section can be solved. The problem can be solved by reimplementing the assignment statement according to BNF. What we want to achieve this time is "complete assignment statement", and finally there is no need to emphasize "variable assignment statement".

Although the assignment statement looks similar to the local variable definition statement, it is actually completely different and much more complicated. The assignment statement in BNF is defined as follows:

```
varlist '=' explist
varlist ::= var {‘,’ var}
var ::= Name | prefixexp '[' exp ']' | prefixexp '.' Name
```

The left side of the assignment operator `=` is the `var` list. `var` expands to 3 kinds. The first `Name` is a variable, currently supports local variables and global variables, and will support upvalue after the introduction of closures. The latter two are table indexes. It can be seen from this that only these types of assignment are supported, while other types such as function calls do not support assignment. Look at the right side of `=`, which is a list of expressions, which can be parsed directly using the completed `exp()` function.

After reading the BNF grammatical rules of the assignment statement, there are three semantic rules.

First, compare the number of variables on the left of `=` and the number of expressions on the right side:

- If equal, assign values one by one;
- If the number of variables is less than the number of expressions, the variable list and the corresponding expression list are assigned one by one, and the extra expressions are ignored;
- If the number of variables is greater than the number of expressions, the expression list and the corresponding variable list are assigned one by one, and the extra variables are assigned to `nil`.

Second, if the last expression on the right side of `=` has multiple values (such as function calls and variable parameters), it will be expanded as much as possible. However, we don't support these two types yet, so ignore this case for now.

Finally, all expressions to the right of `=` are evaluated *before* assignment. Instead of evaluating and assigning values simultaneously. For example, in the following Lua statement, the two expressions `b` and `a` on the right should be evaluated first to obtain `2` and `1`, and then assigned to `a` and `b` on the left respectively. This exchanges the two variables. But if we assign a value while evaluating, we first evaluate `b` on the right, get `2`, and assign it to `a`. Then evaluate `a` on the right to get the `2` that was just assigned, and then assign it to `b`. The end result is that both variables will be `2`.

```lua
local a, b = 1, 2
a, b = b, a --swap 2 variables!!!
```

The diagram below depicts the execution process of *Error*:

```
          +--------+
/--(1)--|    a    |<------\
|         +--------+        |
\------>|    b    |--(2)--/
          +--------+
          |         |
```

Since all values must be evaluated first, the obtained value must be stored in one place first, which is naturally the top of the stack as a temporary variable. The diagram below describes the *correct* execution process:

```
             +--------+
    /---(1)--|   a    |<-------\
    |        +--------+        |
    |  /-(2)-|   b    |<----\  |
    |  |     +--------+     |  |
    \------->|  tmp1  |-(3)-/  |
       |     +--------+        |
       \---->|  tmp2  |--(4)---/
             +--------+
             |        |
```

In the figure, (1) and (2) are to evaluate the expression and put it in the temporary position on the top of the stack; (3) and (4) are to assign the value of the temporary position on the top of the stack to the variable.

The functionality of this approach is correct, but the performance is relatively poor. Because each assignment requires 2 operations, first evaluating and then assigning, it requires 2 bytecodes. But in most cases, only one operation is required. For example, assigning a local variable to another local variable requires only one `Move` bytecode. In particular, the most common assignment statement in a program is the assignment of a single variable. The order of a single variable does not matter, and there is no need to evaluate a temporary variable first. Therefore, the above method of first evaluating to the top of the stack and then assigning a value is *for the correctness of a few cases, while sacrificing the performance of most cases*. This situation is relatively common in programming. The general solution is to add a quick path to *most cases*. For example, the following logic can be used in our current situation:

```
if single variable then
     var = exp // direct assignment, quick path
else // multiple variables
     tmp_vars = exp_list // evaluate all to temporary variables first
     var_list = tmp_vars // assign values uniformly
```

There is a more elegant solution to this specific problem, though. The key here is that in the case of multiple assignments, the assignment of the last variable does not depend on the assignment of other variables, and it can be assigned directly without first evaluating to a temporary variable. So the new solution is: special treatment (direct assignment) is made to the last variable, and other variables are still evaluated first and then assigned. In this way, for the assignment statement of a single variable (the single variable is naturally the last variable), it degenerates into a direct assignment. In this way, the correctness of multiple variables is guaranteed, and the performance of most cases (single variable) is also guaranteed. Pretty!

The following figure describes this scheme: for the previous variable `a`, first evaluate to the temporary variable on the top of the stack, and assign the last variable `b` directly, and then assign the temporary variable on the top of the stack to the corresponding variable in turn.

```
          +--------+
/---(1)--|   a    |<------\
|         +--------+        |
|         |   b    |--(2)--/
|         +--------+ <-------\
\------->|  tmp1  |--(3)----/
          +--------+
          |        |
```

Since we execute the last expression first, thenthe previous expressions are also assigned in reverse order. In this way, all expressions are assigned in reverse order.

So far, the syntax and semantic rules of assignment statements have been introduced. The next step is to rewrite the `assignment()` function. The logic of the function body is as follows:

1. Call `prefixexp()` to read the lvalue list and save it as ExpDesc;
2. Call `exp()` to read the rvalue expression list, the last expression retains ExpDesc, and the remaining expressions are discharged to the top of the stack;
3. Align the number of lvalues and rvalues;
4. Assignment, first assign the last expression to the last lvalue, and then assign the temporary variable on the top of the stack to the corresponding lvalue in turn.

The specific code is omitted here. Only step 4, assignment, will be described in detail below.

Execute the Assignment

Assignment statements consist of lvalues and rvalues:

- Each lvalue is read by the `prefixexp()` function, returning an ExpDesc type. However, it can be seen from BNF that the assignment statement only supports variables and table indexes. The variables include local variables and global variables, corresponding to the two ExpDesc types `Local` and `Global` respectively, and the table indexes include `Index` , `IndexField` and `IndexInt` . So there are up to a total of five types.

- Each rvalue is read by the `exp()` function, which also returns an ExpDesc type, and supports arbitrary ExpDesc types.

To sum up, there are 5 types on the left and N types on the right (N is the number of all types of ExpDesc), and there are a total of 5*N combinations. A bit much, need to sort out.

First of all, for the case where the lvalue is a local variable, the assignment is equivalent to discharging the expression to the stack location of the local variable. Just call the `discharge()` function. This function already handles all N types of ExpDesc.

The remaining four lvalue types are a bit more complicated, but these four cases are similar. The following uses the global variable `Global` type as an example to introduce.

Several combinations of assignments were introduced in the previous Assignment section. For the case where the lvalue is a global variable, the rvalue supports three types of expressions: constant, local variable, and global variable. At that time, for the sake of simplicity, the three expressions `SetGlobalConst` , `SetGlobal` , and `SetGlobalGlobal` were directly generated. Now it can be foreseen that there will be more types of expressions in the future, such as the reading of tables added in this section (such as `t.k`), and subsequent additions such as UpValue and operations (such as `a+b`). If a new bytecode is added for each new type, it will become very complicated.

Moreover, expressions such as table indexing and operations require 2 parameters to represent, and the assignment bytecode of this series of global variables cannot be filled with 2 parameters to represent the source expression of the assignment (one bytecode supports up to 3 `u8` Type parameter, this series of bytecodes needs 1 parameter to represent the destination address, and it seems that 2 parameters can be used to represent the source expression. But through the output of luac, you can see Lua's official global variable assignment bytecode `SETTABUP` has 3 parameters. In addition to the 2 parameters representing the source and destination addresses, there is an additional parameter. Although it is not clear what the function of the extra parameter is, let’s assume that we will use it later That parameter, so our series of bytecodes leaves only one parameter position for the source expression). So how to deal with such complex expressions? The answer is to first evaluate these complex expressions to the top of the stack as temporary variables, which are of `Local` type, and then use `SetGlobal` to complete the assignment.

Here are two extremes:

- The previous practice was to define a bytecode for each source expression type;
- The solution just discussed is to discharge all types on the stack first, and then only use one `SetGlobal` bytecode.

Between these two extremes, we refer to the choice of Lua's official implementation, which is to define a bytecode for the constant type (ExpDesc's `String` , `Float` , etc.), while other types are first discharged to the stack and converted to `Local` type. Although the constant type is actually not a specific type (including multiple types such as `String` , `Float`), but the processing method is the same, through the `add_const()` function to add to the constant table, and use the constant table Index to represent, so when dealing with assignment statements, it can be seen as a type. Thus, our rvalue expressions are simplified to two types: constants and `Local` variables on the stack! In the official implementation of Lua, the `SETTABUP` bytecode of global variable assignment uses 1 bit to indicate whether the source expression is a constant or a variable on the stack. The generation of our bytecode is inconvenient to precisely manipulate bits, so a new bytecode `SetGlobalConst` is added to represent constants.

Why does the official Lua implementation treat constants specially, but not optimize other types (such as global variables, UpValue, table indexes, etc.)? There are two reasons for my personal guess:

- If a global variable or UpValue or table index is accessed so frequently that it is necessary to optimize, then you can simply create a local variable to optimize, such as `local print = print`. For constants, it is inappropriate to assign values to local variables in many cases. For example, changing an assignment statement `g = 100` to `local h = 100; g = a` seems awkward and unnecessary.

- Accessing global variables is based on the variable name table lookup, which is a relatively time-consuming operation, and the cost of adding a bytecode is not obvious in comparison. Access to other types is similar. The access constant is directly referenced through the index, and the cost of adding a bytecode is relatively high.

So far, the assignment of global variables has been introduced, and the assignment of table indexes (that is, the write operation of the table) is similar. For the three types `Index`, `IndexField` and `IndexInt`, we define `SetTable`, `SetField`, `SetInt`, `SetTableConst`, `SetFieldConst`, `SetIntConst` 6 bytecodes.

Finally, the code for assignment is as follows:

```
    // process assignment: var = value
    fn assign_var(&mut self, var: ExpDesc, value: ExpDesc) {
        if let ExpDesc::Local(i) = var {
            // self.sp will be set to i+1 in self.discharge(), which is
            // NOT expected, but it's ok because self.sp will not be used
            // before next statement.
            self.discharge(i, value);
        } else {
            match self.discharge_const(value) {
                ConstStack::Const(i) => self.assign_from_const(var, i),
                ConstStack::Stack(i) => self.assign_from_stack(var, i),
            }
        }
    }

    fn assign_from_stack(&mut self, var: ExpDesc, value: usize) {
        let code = match var {
            ExpDesc::Local(i) => ByteCode::Move(i as u8, value as u8),
            ExpDesc::Global(name) => ByteCode::SetGlobal(name as u8, value as
u8),
            ExpDesc::Index(t, key) => ByteCode::SetTable(t as u8, key as u8,
value as u8),
            ExpDesc::IndexField(t, key) => ByteCode::SetField(t as u8, key as
u8, value as u8),
            ExpDesc::IndexInt(t, key) => ByteCode::SetInt(t as u8, key, value
as u8),
            _ => panic!("assign from stack"),
        };
        self.byte_codes.push(code);
    }

    fn assign_from_const(&mut self, var: ExpDesc, value: usize) {
        let code = match var {
            ExpDesc::Global(name) => ByteCode::SetGlobalConst(name as u8,
value as u8),
            ExpDesc::Index(t, key) => ByteCode::SetTableConst(t as u8, key as
u8, value as u8),
            ExpDesc::IndexField(t, key) => ByteCode::SetFieldConst(t as u8,
key as u8, value as u8),
            ExpDesc::IndexInt(t, key) => ByteCode::SetIntConst(t as u8, key,
value as u8),
            _ => panic!("assign from const"),
        };
        self.byte_codes.push(code);
    }
```

So far, according to the BNF re-assignment statement, it naturally supports the read and write operations of the table.

Assignment and Function Call statement

Now look back at the last of the three questions raised at the beginning of this section, that is, how to distinguish between assignment statements and function call statements during syntax analysis.

Let’s start with the BNF representation of the assignment statement:

```
varlist '=' explist
varlist ::= var {‘,’ var}
var ::= Name | prefixexp '[' exp ']' | prefixexp '.' Name
```

The beginning of the statement is `varlist`, after expansion is the variable `var`, and then it is `Name` and `prefixexp`. `Name` corresponds to `Token::Name`, but `prefixexp` still needs to be expanded. Here is its definition:

```
prefixexp ::= var | functioncall | '(' exp ')'
functioncall ::= prefixexp args | prefixexp ':' Name args
```

Among them, the first `var` returns to the beginning of the assignment statement just now, and the circular reference is ignored. The last one starts with `(`, which is also very simple. After the `functioncall` in the middle is expanded, it also starts with `prefixexp`, which is also a circular reference, but this time it cannot be ignored, because `functioncall` itself is also a complete statement, that is, if a A statement starts with `prefixexp`, which may be an assignment statement or a function call statement. How to distinguish between these two statements? As explained in the previous section, the left value of an assignment statement can only be a variable or a table index. types, and function calls cannot be used as lvalues. This is the key to the distinction!

In summary, the final parsing logic is: if it starts with `Name` or `(`, parse it according to `prefixexp`, and judge the parsing result:

- If it is a function call, it is considered a complete `functioncall` statement;
- Otherwise, it is considered as an assignment statement, and the result of this parsing is only the first `var` of the assignment statement.

To do this, add a function call type `Call` in ExpDesc and let the function call statement `args()` return. In the `load()` function, this part of the code is as follows:

```
            match self.lex.next() {
                Token::SemiColon => (),
                t@Token::Name(_) | t@Token::ParL => {
                    // functioncall and var-assignment both begin with
                    // `prefixexp` which begins with `Name` or `(`.
                    let desc = self.prefixexp(t);
                    if desc == ExpDesc::Call {
                        // prefixexp() matches the whole functioncall
                        // statement, so nothing more to do
                    } else {
                        // prefixexp() matches only the first variable, so we
                        // continue the statement
                        self.assignment(desc);
                    }
                }
```

Summary

In this section, the parsing of the assignment statement is re-analyzed through BNF, and finally the read and write operations of the table are realized. In addition, the statement of local variable definition also needs to be rewritten according to BNF, which is relatively simple, and the introduction is omitted here.

So far, this chapter has completed the basic operations of table definition, construction, reading and writing; and introduce the very important ExpDesc concept and BNF rules.

Arithmetic Operations

Supported operations are described in the Lua Manual. This chapter mainly discusses and implements arithmetic operations and bit operations, and also implements string concatenation operations and length operations by the way. These operations are handled in the same way. As for relational operations and logical operations, special processing needs to be done after conditional statements are introduced.

Simple unary operations are introduced first, then binary operations. Finally we discuss the floating-point conversions.

Unary Operation

The syntax of unary operations in Lua:

```
exp ::= nil | false | true | Numeral | LiteralString | '...' | functiondef |
prefixexp | tableconstructor | exp binop exp | unop exp
```

The unary operation is in the last term: `exp ::= unop exp`. That is, in the expression `exp`, unary operators can be preceded.

Lua supports 4 unary operators:

- `-`, take the negative. This token is also a binary operator: subtraction.
- `not`, logical negation.
- `~`, bitwise inversion. This Token is also a binary operator: bitwise xor.
- `#`, take the length, used for strings and tables, etc.

In the syntax analysis code, just add these 4 unary operators:

```
    fn exp(&mut self) -> ExpDesc {
        match self. lex. next() {
            Token::Sub => self. unop_neg(),
            Token::Not => self. unop_not(),
            Token::BitNot => self. unop_bitnot(),
            Token::Len => self. unop_len(),
            // omit other exp branches
```

The following takes negative `-` as an example, and the others are similar.

Negative

It can be seen from the above BNF that the operand of the negation operation is also the expression `exp`, and the expression is represented by ExpDesc, so several types of ExpDesc are considered:

- Integers and floating-point numbers are directly negated, for example, `ExpDesc::Integer(10)` is directly converted to `ExpDesc::Integer(-10)`. That is to say, for `-10` in the source code, two tokens `Sub` and `Integer(10)` will be generated during the lexical analysis stage, and then converted into `-10` by the syntax analysis. There is no need to directly support negative numbers in lexical analysis, because there can also be the following situation `- -10`, that is, multiple consecutive negative operations. For this case, grammatical analysis is more suitable than lexical analysis.

- Other constant types, such as strings, do not support negation, so a panic is reported.
- Other types are evaluated when the virtual machine is running. Generate a new bytecode `Neg(u8, u8)`, and the two parameters are the destination and source operand addresses on the stack. Only 1 bytecode is added here. In contrast, the Read Global Variables and Table Read operations introduced in the previous chapters both set 3 for optimization Bytecode, three types of parameters are processed separately: variables on the stack, constants, and small integers. But for the negative operation here, the last two types (constants and small integers) have been processed in the above two cases, so we only need to add the bytecode `Neg(u8, u8)` to handle the first type type (variables on the stack). However, the binary operation in the next section cannot fully handle the constant type, so it is necessary to add 3 bytecodes for each operator like the table reading operation.

According to the previous chapter Introduction to ExpDesc, for the last case, two steps are required to generate the bytecode: first, the `exp()` function returns the ExpDesc type, and then `discharge()` function generates bytecode based on ExpDesc. Currently, the existing type of ExpDesc cannot express a unary operation statement, and a new type UnaryOp is required. How is this new type defined?

From an execution point of view, unary operations are very similar to assignments between local variables. The latter is to copy a value on the stack to another location; the former is also, but an operation conversion is added during the copying process. Therefore, the ExpDesc type returned by the unary operation statement can refer to the local variable. For local variables, the expression `exp()` function returns the `ExpDesc::Local(usize)` type, and the associated usize type parameter is the position of the local variable on the stack. For the unary operation, the `ExpDesc::UnaryOp(fn(u8,u8)->ByteCode, usize)` type is added. Compared with the `ExpDesc::Local` type, an associated parameter is added, which is done during the copying process, operation. The parameter type of this operation is `fn(u8,u8)->ByteCode`. This method of passing the enum tag through the function type is described in Use ExpDesc to rewrite the table structure, and will not be repeated here. Also take the negative operation as an example to generate `ExpDesc::UnaryOp(ByteCode::Neg, i)`, where `i` is the stack address of the operand.

The specific parsing code is as follows:

```
    fn unop_neg(&mut self) -> ExpDesc {
        match self.exp_unop() {
            ExpDesc::Integer(i) => ExpDesc::Integer(-i),
            ExpDesc::Float(f) => ExpDesc::Float(-f),
            ExpDesc::Nil | ExpDesc::Boolean(_) | ExpDesc::String(_) => panic!
("invalid - operator"),
            desc => ExpDesc::UnaryOp(ByteCode::Neg, self.discharge_top(desc))
        }
    }
```

After generating the `ExpDesc::UnaryOp` type, generating bytecode from this type is simple:

```
    fn discharge(&mut self, dst: usize, desc: ExpDesc) {
        let code = match desc {
            ExpDesc::UnaryOp(op, i) => op(dst as u8, i as u8),
```

So far, we have completed the unary operation of negation, and the other three unary operations are similar and omitted here.

In addition, since the unary operation statement is defined as: `exp ::= unop exp`, the operand is also an expression statement, here is a recursive reference, so it naturally supports multiple consecutive unary operations, such as `not - ~123` statement .

The above is the syntax analysis part; and the virtual machine execution part needs to add the processing of these 4 new bytecodes. It is also very simple and omitted here.

The next section introduces binary operations, which are much more complicated.

Binary operations

Compared with the unary operation in the previous section, although the binary operation only has one more operand, it introduces many problems, mainly including BNF left recursion, priority, operand type, and evaluation order, etc.

BNF Left Recursive

The complete syntax of the binary operation statement in Lua is as follows:

```
exp ::= nil | false | true | Numeral | LiteralString | '...' | functiondef |
prefixexp | tableconstructor | exp binop exp | unop exp
```

For simplicity, the other parts are simplified to `OTHERS`, then we get:

```
exp ::= exp binop exp | OTHERS
```

It is a left recursion rule, we need to eliminate left recursion according to the method introduced before, and get:

```
exp ::= OTHERS A'
A' := binop exp A' | Epsilon
```

The previous `exp()` function only implemented the `OTHERS` part of the first line above, and now we need to add the `A'` part of the second line, which is also a recursive reference, which is implemented using a loop. Modify the `exp()` function structure as follows:

```
    fn exp(&mut self) -> ExpDesc {
        // OTHERS
        let mut desc = match self. lex. next() {
            // The original various OTHERS type processing is omitted here
        };

        // A' := binop exp A' | Epsilon
        while is_binop(self. lex. peek()) {
            let binop = self.lex.next(); // operator
            let right_desc = self.exp(); // second operand
            desc = self. process_binop(binop, desc, right_desc);
        }
        desc
    }
```

Among them, the second operand right_desc is also recursively called `exp()` function to read, which leads to a problem: priority.

Priority

In the unary operation statement in the previous section, the `exp()` function is also called recursively to read the operand, but because there is only one operand, so no need for priority. Or we can say that all unary operators have the same priority. And unary operators are right associative. For example, the following two examples of consecutive unary operations are executed in order from right to left, regardless of the specific operator:

- `~ -10`, take negative first, then invert bit by bit,
- `- ~10`, first bitwise invert, then negative.

But for the binary operation statement, it is necessary to consider the priority. For example, the following two statements:

- `a + b - c`, perform the previous addition first, and then perform the subsequent subtraction,
- `a + b * c`, perform the subsequent multiplication first, and then perform the previous addition.

Corresponding to the `exp()` function code above, the `OTHERS` part at the beginning reads the first operand `a`; then reads the operator `+` in the `while` loop; and then calls the `exp()` function recursively to read the right operand, so it needs to be calculated at this time. Also take the above two sentences as an example:

- `a + b - c`, end after reading `b` and use it as the right operand; then perform addition `a + b`; and then loop through the following `- c` part again;
- `a + b * c`, after reading `b`, continue down, read and execute the entire `b * c` and use the execution result as the right operand; then perform addition; and end the loop.

```
      -                 +
    /   \             /   \
   +      c          a      *
 /   \                    /   \
a     b                  b     c
```

So in syntax analysis, how to judge which of the above situations is the case? After reading `b`, should we stop parsing and calculate addition first, or continue parsing? It depends on the *priorities* of the next operator and the current operator:

- When the priority of the next operator is *not greater than* the current operator, it is

the first case, stop parsing and complete the current operation first;
- When the priority of the next operator is *greater than* the current operator, it is the second case and needs to continue parsing.

For this, refer to the list of all operator precedence in the Lua language:

```
or
and
<     >     <=    >=    ~=    ==
|
~
&
<<    >>
..
+     -
*     /     //    %
unary operators (not   #     -     ~)
^
```

From top to bottom, the priority becomes higher. The connectors `..` and exponentiation `^` are right associative, and other operators are left associative. In the judging rules listed above, parsing is stopped (instead of continuing parsing) for cases of equal priority, so the default is left associative. Therefore, special treatment is required for two right-associated operators, that is, different priorities are defined for them to the left and to the right, and the one to the left is higher, which will become a right-association.

In summary, define the priority function:

```rust
fn binop_pri(binop: &Token) -> (i32, i32) {
    match binop {
        Token::Pow => (14, 13), // right associative
        Token::Mul | Token::Mod | Token::Div | Token::Idiv => (11, 11),
        Token::Add | Token::Sub => (10, 10),
        Token::Concat => (9, 8), // right associative
        Token::ShiftL | Token::ShiftR => (7, 7),
        Token::BitAnd => (6, 6),
        Token::BitNot => (5, 5),
        Token::BitOr => (4, 4),
        Token::Equal | Token::NotEq | Token::Less | Token::Greater |
Token::LesEq | Token::GreEq => (3, 3),
        Token::And => (2, 2),
        Token::Or => (1, 1),
        _ => (-1, -1)
    }
}
```

For Tokens that are not binary operators, `-1` is returned, which is the lowest priority, and parsing can be stopped no matter what the current operator is. According to Rust's customary practice, this function should return `Option<(i32, i32)>` type, and then return `None` for tokens that are not binary operators. But it is simpler to return `-1` at the calling place, and there is no need to process Option one more time.

This function appears to be a property of the `Token` type, so it seems to be a suitable method defined as `Token`. But `Token` type is defined in `lex.rs`; while priority is a concept of syntax, it should be implemented in `parse.rs`. The Rust language does not allow methods to be added to a type's non-defining file. So the above function is defined as an ordinary function in the `parse.rs` file (rather than the method of `ParseProto` like other functions).

Now, according to the priority, modify the `exp()` function again:

```
    fn exp(&mut self) -> ExpDesc {
        self.exp_limit(0)
    }
    fn exp_limit(&mut self, limit: i32) -> ExpDesc {
        // OTHERS
        let mut desc = match self. lex. next() {
            // The original various OTHERS type processing is omitted here
        };

        // A' := binop exp A' | Epsilon
        loop {
            let (left_pri, right_pri) = binop_pri(self. lex. peek());
            if left_pri <= limit {
                return desc; // stop parsing
            }

            // continue parsing
            let binop = self. lex. next();
            let right_desc = self.exp_limit(right_pri);
            desc = self. process_binop(binop, desc, right_desc);
        }
    }
```

First, add a `limit` parameter to `exp()`, as the priority of the current operator, and limit the subsequent parsing range. However, this parameter belongs to the internal concept of the statement, and the caller of this function does not need to know this parameter; therefore, the actual processing function `exp_limit()` is added, and `exp()` is turned into an outer encapsulation function, using `limit=0` to call the former. The reason why the initial call uses `limit=0` is that `0` is less than any binary operator priority defined in the `binop_pri()` function, so the first operator will continue to be parsed (rather than return to exit the loop); but `0` is greater than the priority `-1` of the non-operator, so if it is followed by the non-operator, it will also exit normally.

The above parsing code combines loops and recursive calls, which is very difficult for those who are not familiar with the algorithm (like me), and it is difficult to write the complete code directly. However, according to the BNF specification after eliminating left recursion, the loop and recursion can be completed, and then the function can be easily completed according to the priority and conditional exit.

In addition, it should be noted that unary operators are also listed in the operator precedence table above, so when parsing unary operation statements in the previous section, the `exp()` function cannot be used when reading the operand expression (initial Priority 0), instead specify an initial priority of 12:

```
fn exp_unop(&mut self) -> ExpDesc {
    self.exp_limit(12) // 12 is all unary operators' priority
}
```

The priority of the exponentiation operation `^` is actually higher than that of the unary operator, so the execution order of the statement `-a^10` is: first exponentiation, and then negation.

Evaluation Order

There is a very subtle bug in the parsing code above, which concerns the order in which the operands are evaluated.

The processing of each operand requires 2 steps: first call the `exp()` function to read the operand and return ExpDesc, and then call the `discharge()` function to discharge the operand to the stack for bytecode operation. The binary operation has 2 operands, so a total of 4 steps are required. Now discuss the sequence of these 4 steps.

According to the processing logic of the binary operation in the `exp()` function of the current version:

- read the first operand first, `desc`;
- After judging that it is a binary operation, call `exp_limit()` recursively, and read the second operand, `right_desc`;
- Then discharge the ExpDesc of the above two operands to the stack in the `process_binop()` function.

Simplified is:

- parse the first operand;
- parse the second operand;
- discharge the first operand;
- discharge the second operand.

During the parsing and discharge stages, bytecode may be generated. So in this order, the bytecodes related to the two operands may be interspersed. Like the following example:

```lua
local a = -g1 + -g2
```

Ignoring the previous local variable definition, and ignoring the operation of undefined global variables will throw an exception. Here, the focus is only on the subsequent addition statement. Generates the following bytecode sequence with the current version of the interpreter:

```
constants: ['g1', 'g2']
byte_codes:
   GetGlobal(0, 0) # parse the first operand
   GetGlobal(1, 1) # parse the second operand
   Neg(2, 0)       # discharge the first operand
   Neg(3, 1)       # discharge the second operand
   Add(0, 2, 3)
```

It can be seen that the bytecodes related to the two operands are interspersed here. In this example, interleaving is fine. But in some cases, parsing the second operand will affect the evaluation of the first operand, and interleaving will cause problems at this time. Like the following example:

```lua
local t = { k = 1 }
local function f(t) t.k = 100; return 2 end -- modify the value of t.k
local r = t.k + f(t)*3
```

For the last sentence, we expected `1 + 2*3`, but if we follow the current order of evaluation:

1. First parse the left operand `t.k` to generate `ExpDesc::IndexField`, but not discharge;
2. Then parse the right operand `f(t)*2`, and execute f(t) during the parsing process, thus modifying the value of `t.k` to `100`;
3. Then discharge the left operationNumber, generate `GetField` bytecode, but at this time `t.k` has been modified by the previous step! Here comes the error. What is actually executed is `100 + 2*3`.

In summary, we need to ensure that the bytecodes of the two operands cannot be interspersed! Then modify the `exp_limit()` function as follows:

```
    fn exp_limit(&mut self, limit: i32) -> ExpDesc {
        // The original various OTHERS type processing is omitted here

        loop {
            // Omit the processing of judging the priority

            // discharge the first operand! ! !
            if !matches!(desc, ExpDesc::Integer(_) | ExpDesc::Float(_) |
ExpDesc::String(_)) {
                desc = ExpDesc::Local(self. discharge_top(desc));
            }

            // continue parsing
            let binop = self. lex. next();
            let right_desc = self.exp_limit(right_pri); // parse the second
operand
            desc = self. process_binop(binop, desc, right_desc);
        }
    }
```

Discharge the first operand onto the stack before parsing the second operand. However, this is not necessary for constant types, because:

- the constant will not be affected by the second operand as in the above example;
- Constants are also to be directly folded in subsequent attempts.

So far, the transformation of `exp_limit()` function for binary operation syntax analysis has been completed. As for the specific processing `process_binop()` function of the binary operation, it is introduced below.

Bytecode

The unary operation introduced in the previous section has only one operand, which can be divided into two cases: constants and variables. Constants are evaluated directly, and variables generate bytecodes. So each unary operation has only one bytecode. Binary operations are more complicated because they involve 2 operands.

First of all, although binary operators are mostly numerical calculations, because Lua's metatable is similar to operator overloading, other types of constants (such as strings, bools, etc.) may be legal operands. When parsing unary operations, these types of constants will directly report an error, but for binary operations, it needs to be executed at the execution stage to determine whether it is legal.

Secondly, if both operands are constants of numeric type (integer and floating point), then the result can be directly calculated during syntax analysis, which is called constant folding.

Otherwise, bytecode is generated and executed by the virtual machine. Similar to Read Global Variables and Read Table operations that have been supported before, each binary operator is also set to 3 types of right operands: variables on the stack, constants, and small integers.

The left operand is uniformly discharged to the stack, because it is rare for the left operand to be a constant. If we also add corresponding bytecodes for constants and small integer types, such as `10-a`, then there are too many bytecode types.

Finally, for addition and multiplication that satisfy the commutative law, if the left operation is a constant, then it can be exchanged. For example, `10+a` can be converted to `a+10` first. Since the right operand `10` is a small integer, it can be use `AddInt` bytecode then.

ExpDesc

Similar to the new ExpDesc type introduced by the unary operation introduced in the previous section, the binary operation also needs a new type because it has one more operand:

```
enum ExpDesc {
     UnaryOp(fn(u8,u8)->ByteCode, usize), // (opcode, operand)
     BinaryOp(fn(u8,u8,u8)->ByteCode, usize, usize), // (opcode, left-
operand, right-operand)
```

Syntax analysis

So far, the basic requirements of the binary operation statement have been introduced. Let's look at the code implementation, that is, the `process_binop()` function called in the `exp()` function:

```
    fn process_binop(&mut self, binop: Token, left: ExpDesc, right: ExpDesc)
-> ExpDesc {
        if let Some(r) = fold_const(&binop, &left, &right) { // constant
fold
            return r;
        }

        match binop {
            Token::Add => self.do_binop(left, right, ByteCode::Add,
ByteCode::AddInt, ByteCode::AddConst),
            Token::Sub => self.do_binop(left, right, ByteCode::Sub,
ByteCode::SubInt, ByteCode::SubConst),
            Token::Mul => self.do_binop(left, right, ByteCode::Mul,
ByteCode::MulInt, ByteCode::MulConst),
            // omit more types
        }
    }
```

Try constant folding first. This part of the function is introduced in the next section because it involves the processing of integer and floating point types. Because the two operands are not necessarily constants, they may not be able to be folded. If the fold is not successful, then the operator and the two operands will be used later, so the `fold_const()` function here can only pass in references.

If it is not a constant and cannot be folded, then call the `do_binop()` function to return ExpDesc. Here, the enum tag is used as a function, which has been introduced before, and will not be introduced here.

Let's look at the `do_binop()` function:

```rust
    fn do_binop(&mut self, mut left: ExpDesc, mut right: ExpDesc, opr:
fn(u8,u8,u8)->ByteCode,
            opi: fn(u8,u8,u8)->ByteCode, opk: fn(u8,u8,u8)->ByteCode) ->
ExpDesc {

        if opr == ByteCode::Add || opr == ByteCode::Mul { // commutative
            if matches!(left, ExpDesc::Integer(_) | ExpDesc::Float(_)) {
                // swap the left-const-operand to right, in order to use
opi/opk
                (left, right) = (right, left);
            }
        }

        let left = self.discharge_top(left);

        let (op, right) = match right {
            ExpDesc::Integer(i) =>
                if let Ok(i) = u8::try_from(i) {
                    (opi, i as usize)
                } else {
                    (opk, self.add_const(i))
                }
            ExpDesc::Float(f) => (opk, self.add_const(f)),
            _ => (opr, self.discharge_top(right)),
        };

        ExpDesc::BinaryOp(op, left, right)
    }
```

First, judge if it is addition or multiplication, and the left operand is a numeric constant, then exchange the two operands, so that the bytecode of `xxCoust` or `xxInt` can be generated later.

Then, discharge the left operand onto the stack;

Then, judge whether the type of the right operand is a numeric constant, or discharge it to the stack.

Finally, `ExpDesc::BinaryOp` is generated.

So far, the grammatical analysis of the binary operation statement is basically completed.

Integer and Float

So far, we have introduced the general analysis process of binary operations, but there is still a detail, that is, the different processing rules for integer and floating point types. Since there is a lot of content in this aspect, and it is relatively independent from the above-mentioned main analysis process, it will be introduced separately in the next section.

Integer and Float

In versions before Lua 5.3, only one type of number is supported, which is floating-point by default. You can use integers by modifying the source code of the Lua interpreter. I understand that this is because Lua was originally used as a configuration language, and most of its users are not programmers, and it does not distinguish between integers and floating-point numbers. For example, `5` and `5.0` are two identical numbers. Later, as the use of Lua expanded, and the need to support integers became stronger (such as bit operations), finally in Lua version 5.3, integers and floating-point numbers were distinguished. This also brings some complexity. The main binary operators are divided into the following three types of processing rules kind:

- Supports integer and floating point numbers, including `+`, `-`, `*`, `//` and `%`. If both operands are integers, the result is also an integer; otherwise (both operands have at least one floating-point number) the result is a floating-point number.
- Only floats are supported, including `/` and `^`. Regardless of the type of the operands, the result is a floating point number. For example `5/2`, although both operands are integers, they will be converted to floating point numbers, and then the result is `2.5`.
- Only integers are supported, including 5 bit operations. The operands must be integers, and the result is also an integer.

The processing of the above three types will be reflected in the constant folding `fold_const()` function of syntax analysis and when the virtual machine executes. The code is cumbersome and omitted here.

Type Conversion

Lua also defines the above rules of type conversion (mainly the rules in the case of incomplete conversion):

- Integer to Float: If the full conversion is not possible, the closest floating point number is used. i.e. the conversion will not fail, only precision will be lost.
- Float to integer: If the conversion cannot be completed, an exception will be thrown.

In the Rust language, the rules for converting integers to floating-points are the same, but converting floating-points to integers is different. This is considered a bug and will be fixed. Before the fix, we can only do this integrity check ourselves, that is, throw an exception if the conversion fails. For this we implement the `ftoi()` function:

```
pub fn ftoi(f: f64) -> Option<i64> {
    let i = f as i64;
    if i as f64 != f {
        none
    } else {
        Some(i)
    }
}
```

You can directly use `as` when converting an integer to a floating-point type, and you need to use this function when converting a floating-point type to an integer.

This conversion will be involved in the syntax analysis and virtual machine execution stages, so create a new `utils.rs` file to put these general functions.

Compare

In the Lua language, in most cases, the distinction between integers and floating-point numbers is avoided as much as possible. The most direct example is that the result of the statement `5 == 5.0` is true, so `Value::Integer(5)` and `Value::Float(5.0)` are equal in the Lua language. Another point is that if these two values are used as the key of the table, they are also considered to be the same key. To this end, we have to modify the two trait implementations of Value before.

The first is the `PartialEq` trait that compares for equality:

```
impl PartialEq for Value {
    fn eq(&self, other: &Self) -> bool {
        match (self, other) {
            (Value::Integer(i), Value::Float(f)) |
            (Value::Float(f), Value::Integer(i)) => *i as f64 == *f && *i ==
*f as i64,
```

Then there is the `Hash` trait:

```
impl Hash for Value {
    fn hash<H: Hasher>(&self, state: &mut H) {
        match self {
            Value::Float(f) =>
                if let Some(i) = ftoi(*f) {
                    i.hash(state)
                } else {
                    unsafe {
                        mem::transmute::<f64, i64>(*f).hash(state)
                    }
                }
```

However, there is still one place where the type needs to be distinguished, that is, when adding a constant to the constant table during syntax analysis, when querying whether the constant already exists. To do this, implement a type-sensitive comparison method:

```
impl Value {
    pub fn same(&self, other: &Self) -> bool {
        // eliminate Integer and Float with same number value
        mem::discriminant(self) == mem::discriminant(other) && self == other
    }
}
```

Test

At this point, the syntax analysis of the binary operation statement is finally completed. The virtual machine execution part is very simple and is skipped here. You can test the Lua code as follows:

```
g = 10
local a,b,c = 1.1, 2.0, 100

print(100+g) -- commutative, AddInt
print(a-1)
print(100/c) -- result is float
print(100>>b) -- 2.0 will be convert to int 2
print(100>>a) -- panic
```

Control Structure

This chapter introduces the control structure. The most obvious change is that since now, the virtual machine no longer only executes sequentially, but jumps. And because the parsing of the syntax block is called recursively during syntax analysis, the local variable scope needs to be dealt with, which makes the meaning and boundary of the block clearer.

Several control structures in Lua language are very common, similar to other languages, nothing special. Next, the first section introduces the if branch of the simplest `if` statement, and introduces conditional jumps and block processing. Then introduce other control structures in turn, most of which are implemented through conditional jumps (Test bytecode) and unconditional jumps (Jump bytecode). Except that the numeric-for statement uses 2 special bytecodes for performance considerations due to its complex semantics. The generic-for statement needs to use functions, so it will be introduced after introducing functions in subsequent chapters.

In addition, this chapter also discusses and attempts to introduce the continue statement that does not exist in Lua, and guarantees backward compatibility.

In addition, although this chapter fully implements each control structure functionally, the implementation here will be optimized after the introduction of relational and logical operations in the next chapter.

`if` statement

The biggest difference between the conditional judgment statement and the previously implemented statement is that the bytecode is no longer executed sequentially, and jumps may occur. To this end, we add a new bytecode `Test`, associated with 2 parameters:

- The first parameter, `u8` type, determines the location of the condition on the stack;
- The second parameter, `u16` type, the number of bytecodes to jump forward.

The semantics of this bytecode is: if the statement represented by the first parameter is false, then jump forward to the bytecode of the number specified by the second parameter. The control structure diagram is as follows:

```
+-------------------+
| if condition then |---\ skip the block if $condition is false
+-------------------+   |
                        |
    block               |
                        |
+-----+                 |
| end |                 |
+-----+                 |
<-----------------------/
```

The definition of Test bytecode is as follows:

```rust
pub enum ByteCode {
     // condition structures
     Test(u8, u16),
```

The second parameter is the number of bytecodes to jump to, that is, the relative position. If absolute positions are used, the code to parse and execute is slightly simpler, but less expressive. The range of 16bit is 65536. If absolute position is used, the code beyond 65536 in a function cannot use jump bytecode. And if you use the relative position, then it supports jumping within the range of 65536 of the bytecode itself, and you can support very long functions. So we use relative positions. This also introduces a problem that has been ignored, which is the range of parameters in the bytecode. For example, the stack index parameters are all of the `u8` type, so if there are more than 256 local variables in a function, it will overflow and cause bugs. In the follow-up, the range of parameters needs to be specially dealt with.

According to the above control structure diagram, the syntax analysis code for completing the if statement is as follows:

```
    fn if_stat(&mut self) {
        let icond = self.exp_discharge_top(); // read condition statement
        self.lex.expect(Token::Then); // `then` keyword

        // generate `Test` placeholder, and the 2 parameters will be added
later
        self.byte_codes.push(ByteCode::Test(0, 0));
        let itest = self.byte_codes.len() - 1;

        // parse the block! And it is expected to return the `end` keyword,
        // does not support `elseif` and `else` branches temporarily.
        assert_eq!(self. block(), Token::End);

        // Fix Test bytecode parameter.
        // `iend` is the current position of the bytecode sequence,
        // `itest` is the position of the Test bytecode, and the difference
        // between the two is the number of bytecodes that need to be
jumped.
        let iend = self.byte_codes.len() - 1;
        self.byte_codes[itest] = ByteCode::Test(icond as u8, (iend - itest)
as u16);
    }
```

The code flow has been explained line by line in the comments. What needs to be explained in detail here is the `block()` function called recursively.

End of Block

The original `block()` function is actually the entry point of the entire syntax analysis, which is executed only once (without recursive calls), and reads to the end of the source code `Token::Eos` as the end:

```
    fn block(&mut self) {
        loop {
            match self. lex. next() {
                // Other statement parsing is omitted here
                Token::Eos => break, // Eos exits
            }
        }
    }
```

The expected end of the code block in the `if` statement to be supported is the keyword `end`; other keywords such as `elseif` and `else` will be included in the future. The end of the code block is not just `Token::Eos`, we need to modify the `block()` function, and consider the Token that is not the beginning of a legal statement (such as `Eos`, keyword `end`, etc.) as a block End, and it is up to the caller to determine whether it is the expected end. There are 2 ways to modify the specific code:

- Use `lex.peek()` instead of `lex.next()` in the above code. If the Token you see is not the beginning of a legal statement, exit the loop. At this time, the Token has not been read by consumption. The external caller then calls `lex.next()` to read the Token for judgment. If this is done, then all the current statement processing codes must add a `lex.next()` at the very beginning to skip the seen Token, which is more verbose. For example, in the `if_stat()` function in the previous paragraph, it is necessary to use `lex.next()` to skip the keyword `if` .

- Still use `lex.next()` , for the Token that is not read at the beginning of a legal statement, it will be returned to the caller as the function return value. We adopt this method, the code is as follows:

```
    fn block(&mut self) -> Token {
        loop {
            match self. lex. next() {
                // Other statement parsing is omitted here
                t => break t, // return t
            }
        }
    }
```

So in the `if_stat()` function above, it is necessary to judge the return value of `block()` as `Token::End` :

```
        // parse syntax block! And it is expected to return the end keyword,
 temporarily does not support elseif and else branches
        assert_eq!(self. block(), Token::End);
```

The original syntax analysis entry function `chunk()` also needs to increase the judgment of the return value of `block()` :

```
    fn chunk(&mut self) {
        assert_eq!(self. block(), Token::Eos);
    }
```

Variable Scope in Block

Another area of the `block()` function that needs to be changed is the scope of local variables. That is, local variables defined inside the block are not visible outside.

This feature is very core! But the implementation is very simple. Just record the number of current local variables at the entry of `block()` , and then clear the newly added local variables before exiting. code show as below:

```
    fn block(&mut self) -> Token {
        let nvar = self.locals.len(); // record the original number of local
variables
        loop {
            match self. lex. next() {
                // Other statement parsing is omitted here
                t => {
                    self.locals.truncate(nvar); // invalidate local
variables defined inside the block
                    break t;
                }
            }
        }
    }
```

After the Upvalue is introduced later, other processing is required.

do Statement

The above two subsections deal with the problem of blocks. The simplest statement to create a block is the `do` statement. Because it is too simple, we introduce it here by the way. The syntax analysis code is as follows:

```
    // BNF:
    // do block end
    fn do_stat(&mut self) {
        assert_eq!(self. block(), Token::End);
    }
```

Virtual Machine Execution

The previous virtual machine execution was to execute bytecodes sequentially, and use Rust's for statement to loop through:

```
    pub fn execute<R: Read>(&mut self, proto: &ParseProto<R>) {
        for code in proto.byte_codes.iter() {
            match *code {
                // All bytecode pre-defined logic is omitted here
            }
        }
    }
```

Now to support the jump of the `Test` bytecode, it is necessary to be able to modify the position of the next traversal during the loop traversal of the bytecode sequence. Rust's `for` statement does not supported modifies the traversal position during the loop, so we need to manually control the loop:

```
    pub fn execute<R: Read>(&mut self, proto: &ParseProto<R>) {
        let mut pc = 0; // bytecode index
        while pc < proto.byte_codes.len() {
            match proto.byte_codes[pc] {
                // The pre-defined logic of other bytecodes is omitted here

                // condition structures
                ByteCode::Test(icond, jmp) => {
                    let cond = &self. stack[icond as usize];
                    if matches!(cond, Value::Nil | Value::Boolean(false)) {
                        pc += jmp as usize; // jump if false
                    }
                }
            }

            pc += 1; // next bytecode
        }
    }
```

Loop execution is controlled by the bytecode location `pc`. After all bytecodes are executed, `pc` will be incremented by 1, pointing to the next bytecode; for the jump bytecode `Test`, `pc` will be modified additionally. Since `Test` bytecode will also execute `pc` auto-increment at the end, so its jump position is actually the target address minus 1. In fact, you can add a `continue;` statement here to skip the last auto-increment of `pc`. I don't know which of these two approaches is better.

As can be seen from the judgment of the above code, there are only two false values in the Lua language: `nil` and `false`. Other values, such as 0, empty table, etc., are all true values.

Test

So far we have implemented the simplest `if` statement.

Since we do not yet support relational operations, the judgment condition after `if` can only use other statements. The test code is as follows:

```
if a then
    print "skip this"
end
if print then
    local a = "I am true"
    print(a)
end

print (a) -- should be nil
```

The conditional statement `a` in the first judgment statement is an undefined global variable, the value is `nil`, which is false, so the internal statement is not executed.

The conditional statement `print` in the second judgment statement is a defined global variable and is true, so the internal statement will execute. The local variable `a` is defined inside the block, which is executed normally inside, but after the end of the block, `a` is invalid, and then it is used as an undefined global variable, and the print is `nil`.

`elseif` and `else` branches

The previous section supported the `if` statement. This section continues with the `elseif` and `else` branches.

The complete BNF specification is as follows:

```
    if exp then block {else if exp then block} [else block] end
```

In addition to the if judgment, there can also be multiple optional elseif judgment branches in a row, followed by an optional else branch at the end. The control structure diagram is as follows:

```
     +-------------------+
     | if condition then |-------\ jump to the next `elseif` branch if
$condition is false
     +-------------------+       |
                                 |
         block                   |
/<----                           |
|    +-----------------------+<--/
|    | elseif condition then |-----\ jump to the next `elseif` branch if
$condition is false
|    +-----------------------+     |
|                                  |
|        block                     |
+<----                             |
|    +-----------------------+<----/
|    | elseif condition then |-------\ jump to the `else` branch if
$condition is false
|    +-----------------------+       |
|                                    |
|        block                       |
+<----                               |
|    +------+                        |
|    | else |                        |
|    +------+<-----------------------/
|
|        block
|
|    +-----+
|    | end |
|    +-----+
\---> All block jump here.
      The last block gets here without jump.
```

The above diagram depicts the situation where there are 2 `elseif` branches and 1 `else` branch. Except for the judgment jump of `if` in the upper right corner, the rest are jumps to be added. There are 2 types of jumps:

- The conditional jump on the right side of the figure is executed by the `Test` bytecode added in the previous section;
- The unconditional jump on the left side of the figure needs to add `Jump` bytecode, which is defined as follows:

```rust
pub enum ByteCode {
    // condition structures
    Test(u8, u16),
    Jump(u16),
```

The syntax analysis process is as follows:

- For the `if` judgment branch, compared with the previous section, the position of the conditional jump remains unchanged, and it is still the end position of the block; however, an unconditional jump instruction needs to be added at the end of the block to jump to the end of the entire if statement;

- For the `elseif` branch, it is handled in the same way as the `if` branch.

- For the `else` branch, no processing is required.

The format of the final generated bytecode sequence should be as follows, where `...` represents the bytecode sequence of the inner code block:

```
     Test --\  `if` branch
     ...    |
/<-- Jump   |
|      /<---/
|    Test ----\  `elseif` branch
|    ...      |
+<-- Jump     |
|      /<-----/
|    Test ------\  `elseif` branch
|    ...        |
+<-- Jump       |
|      /<-------/
|    ...   `else` branch
|
\--> end of all
```

The syntax analysis code is as follows:

```
    fn if_stat(&mut self) {
        let mut jmp_ends = Vec::new();

        // `if` branch
        let mut end_token = self. do_if_block(&mut jmp_ends);

        // optional multiple `elseif` branches
        while end_token == Token::Elseif { // If the previous block ends
with the keyword `elseif`
            end_token = self.do_if_block(&mut jmp_ends);
        }

        // optional `else` branch
        if end_token == Token::Else { // If the previous block ends with the
keyword `else`
            end_token = self. block();
        }

        assert_eq!(end_token, Token::End); // Syntax: `end` at the end

        // Repair the unconditional jump bytecode at the end of the
        // block in all `if` and `elseif` branches, and jump to the
        // current position
        let iend = self.byte_codes.len() - 1;
        for i in jmp_ends.into_iter() {
            self.byte_codes[i] = ByteCode::Jump((iend - i) as i16);
        }
    }
```

The processing function `do_if_block()` for if and elseif is as follows:

```
    fn do_if_block(&mut self, jmp_ends: &mut Vec<usize>) -> Token {
        let icond = self.exp_discharge_top(); // read judgment statement
        self.lex.expect(Token::Then); // Syntax: `then` keyword

        self.byte_codes.push(ByteCode::Test(0, 0)); // generate Test
bytecode placeholder, leave the parameter blank
        let itest = self.byte_codes.len() - 1;

        let end_token = self. block();

        // If there is an `elseif` or `else` branch, then the current
        // block needs to add an unconditional jump bytecode, to jump
        // to the end of the entire `if` statement. Since the position
        // of the end is not known yet, the parameter is left blank and the
        // The bytecode index is recorded into `jmp_ends`.
        // No need to jump if there are no other branches.
        if matches!(end_token, Token::Elseif | Token::Else) {
            self.byte_codes.push(ByteCode::Jump(0));
            jmp_ends.push(self.byte_codes.len() - 1);
        }

        // Fix the previous Test bytecode.
        // `iend` is the current position of the bytecode sequence,
        // `itest` is the position of the Test bytecode, and the difference
        // between the two is the number of bytecodes that need to be
jumped.
        let iend = self.byte_codes.len() - 1;
        self.byte_codes[itest] = ByteCode::Test(icond as u8, (iend - itest)
as i16);

        return end_token;
    }
```

Virtual Machine Execution

The implementation of the newly added unconditional jump bytecode `Jump` is very simple. Compared with the previous conditional jump bytecode `Test`, only the conditional judgment is removed:

```
// conditional jump
ByteCode::Test(icond, jmp) => {
    let cond = &self. stack[icond as usize];
    if matches!(cond, Value::Nil | Value::Boolean(false)) {
        pc += jmp as usize; // jump if false
    }
}

// unconditional jump
ByteCode::Jump(jmp) => {
    pc += jmp as usize;
}
```

`while` and `break` Statements

This section introduces the `while` and `break` statement.

`while` Statement

Compared with the simple form of the `if` statement (excluding `elseif` and `else` branches), the `while` statement just adds an unconditional jump bytecode at the end of the internal block, jumping back to the beginning of the statement. As shown in the jump on the left in the figure below:

```
/--->+----------------------+
|    | while condition then |---\ skip the block if $condition is false
|    +----------------------+   |
|                               |
|        block                  |
\<----                          |
     +------+                   |
     | end  |                   |
     +------+                   |
     <--------------------------/
```

The format of the final generated bytecode sequence is as follows, where `...` represents the bytecode sequence of the inner code block:

```
/-->  Test --\  `if` branch
|     ...    |
\---  Jump   |
        <----/ The end of the entire `while` statement
```

The syntax analysis process and code also add an unconditional jump bytecode on the basis of the `if` statement. We skip the code here. One thing that needs to be changed is that the unconditional jump here is a backward jump. But the second parameter of the previous `Jump` bytecode is `u16` type, which can only jump forward. Now we need to change to `i16` type, and use a negative number to represent a backward jump:

```
pub enum ByteCode {
     Jump(i16),
```

Correspondingly, the execution part of the virtual machine needs to be modified as follows:

```
        // unconditional jump
        ByteCode::Jump(jmp) => {
            pc = (pc as isize + jmp as isize) as usize;
        }
```

Compared with C language, Rust's type management is stricter, so it looks more verbose.

`break` Statement

The `while` statement itself is very simple, but it introduces another statement: `break`. The `break` statement itself is also very simple, just unconditionally jump to the end of the block, but the problem is that not all blocks support `break`, for example, the block inside the if introduced earlier does not support `break`, only the block of the loop statement supports `break`. To be precise, what the `break` wants to jump out of is the *loop* block of the *nearest* layer. For example, the following example:

```
while 123 do -- outer loop block, support `break`
    while true do -- middle-level loop block, support `break`
        a = a + 1
        if a < 10 then -- inner block, does not support `break`
            `break` -- `break` out of the `while true do` loop
        end
    end
end
```

There are 3 layers of blocks in the code, the outer and middle while blocks support `break`, and the inner if block does not support `break`. At this time, `break` is to jump out of the middle block.

If the `break` statement is not within a loop block, it is a syntax error.

In order to realize the above functions, a parameter can be added to the `block()` function to indicate the latest loop block when calling recursively. Since the block has not ended when the jump bytecode is generated, and the jump destination address is not yet known, so the jump bytecode can only be generated first, and the parameters are left blank; and then the byte is repaired at the end of the block code parameter. So the parameter of the `block()` function is the index list of the `break` jump bytecode of the latest loop block. When calling the `block()` function,

- If it is a loop block, create a new index list as a call parameter, and after the call ends, use the current address (that is, the end position of the block) to repair the bytecode in the list;
- If it is not a cyclic block, use the current list (that is, the current most recent cyclic block) as the call parameter.

But the recursive call of `block()` function is not direct recursion, but indirect recursion. If you want to pass parameters in this way, then all parsing functions must add this parameter, which is too complicated. So put this index list into the global `ParseProto`. Locality is sacrificed for coding convenience.

Let's look at the specific coding implementation. First add the `break_blocks` field in `ParseProto`, the type is a list of "jump bytecode index list":

```
pub struct ParseProto<R: Read> {
     break_blocks: Vec::<Vec::<usize>>,
```

When parsing the while statement, add a list before calling the `block()` function; after calling, fix the jump bytecode in the list:

```
      fn while_stat(&mut self) {

          // Omit the conditional judgment statement processing part

          // Before calling block(), append a list
          self.break_blocks.push(Vec::new());

          // call block()
          assert_eq!(self.block(), Token::End);

          // After calling block(), pop up the list just added, and fix the
jump bytecode in it
          for i in self.break_blocks.pop().unwrap().into_iter() {
              self.byte_codes[i] = ByteCode::Jump((iend - i) as i16);
          }
      }
```

After the block is prepared, the `break` statement can be implemented:

```
      fn `break`_stat(&mut self) {
          // Get the bytecode list of the nearest loop block
          if let Some(breaks) = self.break_blocks. last_mut() {
              // Generate a jump bytecode placeholder, the parameter is left
blank
              self.byte_codes.push(ByteCode::Jump(0));
              // Append to the bytecode list
              `break`s.push(self.byte_codes.len() - 1);
          } else {
              // Syntax error if there is no loop block
              panic!("break outside loop");
          }
      }
```

`continue` Statement?

After implementing the `break` statement, the `continue` statement naturally comes to mind. Moreover, the implementation of `continue` is similar to `break`, the difference is that one jumps to the end of the loop, and the other jumps to the beginning of the loop. Adding this function is a convenient thing. But Lua does not support the `continue` statement! A small part of this has to do with the `repeat..until` statement. We discuss the `continue` statement in more detail after introducing the `repeat..until` statement in the next section.

`repeat..until` and `continue` Statements

This section introduces the `repeat..until` statement, and discusses and attempts to introduce the `continue` statement that Lua language does not support.

`repeat..until` Statement

The `repeat..until` statement is similar to the `while` statement, except that the judgment condition is placed behind to ensure that the internal code block is executed at least once.

```
     +--------+
     | repeat |
     +--------+
/--->
|        block
|
|    +-----------------+
\----| until condition |
     +-----------------+
```

The format of the final generated bytecode sequence is as follows, where `...` represents the bytecode sequence of the inner code block:

```
    ... <--\
    Test ---/ `until` judgment condition
```

Compared with the bytecode sequence of the `while` statement, it seems that the Test is put at the end and the original Jump bytecode is replaced. But the situation is not that simple! Putting the judgment conditional statement behind the block will introduce a big problem. The local variables defined in the block may be used in the judgment conditional statement. For example, the following example:

```lua
-- keep retrying until the request succeeds
repeat
    local ok = request_xxx()
until ok
```

The variable `ok` after the last line `until` is obviously intended to refer to the local variable defined in the second line. However, the previous code block analysis function `block()` has deleted the internally defined local variables at the end of the function. That is to say, according to the previous syntax analysis logic, when `until` is parsed, the internally defined `ok` local variable has become invalid and cannot be used. This is clearly unacceptable.

In order to support the ability to read internal local variables during `until`, the original `block()` function needs to be modified (the code is always messed up by these strange requirements), and the control of local variables is independent. For this reason, a `block_scope()` function is added, which only does syntax analysis; while the scope of internal local variables is completed by the outer `block()` function. In this way, the place where the `block()` function was originally called (such as if, while statement, etc.) does not need to be modified, and this special `repeat..until` statement calls the `block_scope()` function for finer control. code show as below:

```
    fn block(&mut self) -> Token {
        let nvar = self. locals. len();
        let end_token = self. block_scope();
        self.locals.truncate(nvar); // expire internal local variables
        return end_token;
    }
    fn block_scope(&mut self) -> Token {
        ... // The original block parsing process
    }
```

Then, the analysis code of the `repeat..until` statement is as follows:

```
    fn repeat_stat(&mut self) {
        let istart = self.byte_codes.len();

        self. push_break_block();

        let nvar = self.locals.len(); // Internal local variable scope
control!

        assert_eq!(self. block_scope(), Token::Until);

        let icond = self.exp_discharge_top();

        // expire internal local variables AFTER condition exp.
        self.locals.truncate(nvar); // Internal local variable scope
control!

        let iend = self.byte_codes.len();
        self.byte_codes.push(ByteCode::Test(icond as u8, -((iend - istart +
1) as i16)));

        self. pop_break_block();
    }
```

In the above code, the 2 lines commented complete the scope control of the internal local variables in the original `block()` function. After calling `exp_discharge_top()` and parsing the conditional judgment statement, the internally defined local variables are deleted.

`continue` statement

It took a lot of space to explain the scope of variables in the `repeat..until` statement, which has a lot to do with the `continue` statement that does not exist in Lua.

When the `break` statement was supported in the previous section, it was mentioned that the Lua language does not support the `continue` statement. There is a lot of debate on this issue, and there is a high demand for adding a `continue` statement in Lua. As early as 2012, there was a related proposal, which listed in detail the advantages and disadvantages of adding the `continue` statement and related discussions. Twenty years have passed, and even though the stubborn Lua added the `goto` statement in version 5.2, it still did not add the `continue` statement.

The "Unofficial FAQ" explains this:

- The `continue` statement is just one of many control statements, similar ones include `goto`, `break` with label, etc. The `continue` statement is nothing special, there is no need to add this statement;
- Conflicts with existing `repeat..until` statements.

In addition, an email from Roberto, the author of Lua, is more representative of the official attitude. The reason for this is the first point above, that is, the `continue` statement is just one of many control statements. An interesting thing is that there are two examples in this email, and the other example just happens to be `repeat..until` besides `continue`. The above unofficial FAQ also mentioned that these two statements conflict.

The reason for the conflict between these two statements is that if there is a `continue` statement in the `repeat..until` internal code block, then it will jump to the until conditional judgment position. If there are local variables defined in the block are used in `until` statement, while the `continue` statement may skip the definition and jump to the `until`, then this local variable is meaningless in `until`. This is where the conflict lies. For example the following code:

```lua
repeat
    `continue` -- jump to until, skip the definition of `ok`
    local ok = request_xxx()
until ok -- how to deal with `ok` here?
```

In contrast, the equivalent of the `repeat..until` statement in the C language is the `do..while` statement, which supports `continue`. This is because in the `do..while` statement of the C language, the conditional judgment after the while is outside the scope of the internal code block. For example, the following code will compile error:

```
do {
    bool ok = request_xx();
} while (ok); // error: 'ok' undeclared
```

Such a specification (the conditional judgment is outside the scope of the inner code block) is not convenient in some usage scenarios (such as the above example), but there are also very simple solutions (such as move `ok` definition outside the loop), and the syntax analysis is simpler, for example, there is no need to separate the `block_scope()` function. Then why does Lua stipulate that the conditional judgment statement should be placed within the inner scope? The speculation is as follows, if Lua also follows the practice of C language (the conditional judgment is outside the scope of the internal code block), and then the user writes the following Lua code, `ok` after the until will be parsed as a Global variables, without reporting errors like C language! This is not the user's intention, thus causing a serious bug.

```
repeat
    local ok = request_xxx()
until ok
```

To sum up, the `repeat..until` statement needs to put the conditional judgment statement after `until` in the scope of the internal code block in order to avoid bugs with a high probability; then when the `continue` statement jumps to the conditional statement, it may skip the definition of local variables, and then there is a conflict.

Try Adding `continue` Statement

Lua's official reason for not supporting the `continue` statement is mainly that they think the frequency of use of the `continue` statement is very low and it is not worth supporting. But in my personal programming experience, whether in Lua or other languages, the frequency of use of the `continue` statement is still very high. Although it may not be as good as `break`, it is far more than `goto` and `break` with labels, and even more than `repeat..until` statement. Besides, the way to implement the `continue` function in Lua (`repeat..until true` + `break`, or `goto`) is more verbose than using `continue` directly. So can we add a `continue` statement to our interpreter?

First of all, we have to resolve the conflict with `repeat..until` mentioned above. There are several solutions:

- Make a rule that the `continue` statement is not supported in `repeat..until`, just like the `if` statement does not support `continue`. But this is very easy to cause misunderstanding. For example, a piece of code has two layers of loops, the outer layer is a `while` loop, and the inner layer is a `repeat` loop; the user wrote a `continue` statement in the inner loop, intending to make the inner `repeat` loop take effect, but because `repeat` does not actually support `continue`, Then it will take effect in the outer while loop, and `continue` the outer `while` loop. This is a serious potential bug.

- Make a rule that the `continue` statement is prohibited in `repeat..until`. If there is `continue`, an error will be reported. This can avoid the potential bugs of the above scheme, but this prohibition is too strict.

- Make a rule that if an internal local variable is defined in `repeat..until`, the `continue` statement is prohibited. This plan is a little more relaxed than the last one, but it can be more relaxed.

- Make a rule that after the `continue` statement appears in `repeat..until`, the definition of internal local variables is prohibited; in other words, `continue` prohibits jumping to local variable definitions. This is similar to the restriction on subsequent `goto` statements. However, it can be more relaxed.

- On the basis of the previous solution, only the local variables defined after the `continue` statement are used in the conditional judgment statement after the `until`, which is prohibited. It's just that the judgment of whether to use local variables in the statement is very complicated. If function closures and Upvalue are supported later, it is basically impossible to judge. So this plan is not feasible.

In the end, I chose to use the second-to-last solution. For specific coding implementation, there used to be `break_blocks` in `ParseProto` to record break statements, and now a similar `continue_blocks` is added, but the member type is `(icode, nvar)`. Among them, the first variable icode is the same as the members of `break_blocks`, and records the position of the Jump bytecode corresponding to the `continue` statement for subsequent correction; the second variable `nvar` represents the number of local variables in the `continue` statement, which is used for Subsequent checks to see if the new local variable has been jumped.

Second, adding a `continue` statement cannot affect existing code. In order to support the `continue` statement, it is necessary to use `continue` as a keyword (similar to the `break` keyword), so many existing Lua codes use `continue` as a label, or even a variable name or function name (essentially a variable name) will fail to parse. To this end, a tricky solution is not to use `continue` as a keyword, but to judge when parsing a statement that if it starts with `continue` and is followed by a block-ending Token (such as `end`, etc.), it is considered to be `continue` statement. Thus in most other places, `continue` will still be interpreted as a normal Name.

In the corresponding `block_scope()` function, the part starting with Token::Name, the newly added code is as follows:

```
        loop {
            match self. lex. next() {
                // Omit parsing of other types of statements
                t@Token::Name(_) | t@Token::ParL => {
                    // this is not standard!
                    if self.try_continue_stat(&t) { // !! New !!
                        continue;
                    }

                    // The following omits the parsing of standard
                    // function calls and variable assignment statements
                }
```

The `try_continue_stat()` function is defined as follows:

```
    fn try_continue_stat(&mut self, name: &Token) -> bool {
        if let Token::Name(name) = name {
            if name.as_str() != "continue" { // The beginning of the
judgment statement is `continue`
                return false;
            }
            if !matches!(self.lex.peek(), Token::End | Token::Elseif |
Token::Else) {
                return false; // Judgment followed by one of these 3 Tokens
            }

            // Then, it's the `continue` statement. The following processing
            // is similar to the break statement processing
            if let Some(continues) = self.continue_blocks.last_mut() {
                self.byte_codes.push(ByteCode::Jump(0));
                continues.push((self.byte_codes.len() - 1,
self.locals.len()));
            } else {
                panic!("continue outside loop");
            }
            true
        } else {
            false
        }
    }
```

Before parsing to the code block of the loop body, it must be prepared first, which is the `push_loop_block()` function. After the block ends, use `pop_loop_block()` to handle `break` s and `continue` s. The jump corresponding to `break` s is to jump to the end of the block, that is, the current position; the jump position corresponding to `continue` s is determined according to different loops (for example, the while loop jumps to the beginning of the loop, and the repeat loop jumps to the end of the loop) , so parameters are required to specify; in addition, when processing continus, it is necessary to check whether there are new definitions of local variables, that is, compare the number of current local variables with the number of local variables in the `continue` statement.

```
    // before entering loop block
    fn push_loop_block(&mut self) {
        self. break_blocks. push(Vec::new());
        self. `continue`_blocks. push(Vec::new());
    }

    // after leaving loop block, fix `break` and `continue` Jumps
    fn pop_loop_block(&mut self, icon`continue`: usize) {
        // breaks
        let iend = self.byte_codes.len() - 1;
        for i in self.break_blocks.pop().unwrap().into_iter() {
            self.byte_codes[i] = ByteCode::Jump((iend - i) as i16);
        }

        // continues
        let end_nvar = self. locals. len();
        for (i, i_nvar) in self.`continue`_blocks.pop().unwrap().into_iter()
 {
            if i_nvar < end_nvar {
               // i_nvar is the number of local variables in the
               // `continue` statement, end_nvar is the number of
               // current local variables
                panic!("`continue` jump into local scope");
            }
            self.byte_codes[i] = ByteCode::Jump((i`continue` as isize - i as
 isize) as i16 - 1);
        }
    }
```

So far, we have implemented the `continue` statement while ensuring backward compatibility! You can use the following code to test:

```lua
-- validate compatibility
continue = print -- continue as global variable name, and assign it a value
continue(continue) -- call continue as function

-- continue in while loop
local c = true
while c do
    print "hello, while"
    if true then
      c = false
      continue
    end
    print "should not print this!"
end

-- continue in repeat loop
repeat
    print "hello, repeat"
    local ok = true
    if true then
      continue -- continue after local
    end
    print "should not print this!"
until ok

-- continue skip local in repeat loop
-- PANIC!
repeat
    print "hello, repeat again"
    if true then
      continue -- skip `ok`!!! error in parsing
    end
    local ok = true
until ok
```

`repeat..until` Existence

As can be seen above, the existence of the `repeat..until` statement introduces two problems because the scope of the local variables defined in the block needs to be extended in the `until` part:

- In programming implementation, it is necessary to create a `block_scope()` function;
- Conflict with `continue` statement.

I personally think that introducing the above two problems in order to support a statement that is rarely used like `repeat..until` is not worth the candle. If I were to design the Lua language, this statement would not be supported.

In the 8.4 Exercise section of the official "Lua Programming (4th Edition)" book, the following questions are raised:

> Exercise 8.3: Many people think that because `repeat-until` is rarely used, it should not appear at the end in a simple programming language like Lua language. What do you think?

I really want to know the author's answer to this question, but unfortunately, none of the exercises in this book give an answer.

numerical-for Statement

Lua's `for` statement supports two types:

- Numeric: `for Name '=' exp ',' exp [',' exp] do block end`
- Generics: `for namelist in explist do block end`

Generic-for requires function support, and it will be implemented after introducing functions in the next chapter. This section implements numeric-for. It can be seen from the BNF definition that the first two tokens of the two types are the same, and the third token of the numeric type is `=`. By this distinction two types can be distinguished:

```
    fn for_stat(&mut self) {
        let name = self. read_name();
        if self.lex.peek() == &Token::Assign {
            self.for_numerical(name); // numerical
        } else {
            todo!("generic for"); // generic
        }
    }
```

Control Structure

The semantics of the numerical-for statement is obvious. The three expressions after the equal sign `=` are the initial value `init`, the `limit`, and the `step`. `step` can be positive or negative, but not 0. The control structure diagram is as follows (assuming step>0 in the diagram):

```
       +--------------+
  /--->| i <= limit ? |--No--\ jump to the end if exceed limit
  |    +--------------+      |
  |                          |
  |        block             |
  |                          |
  |    +-----------+         |
  \----| i += step |         |
       +-----------+         |
           <-----------------/
```

The execution logic in the boxes can be implemented with 1 bytecode respectively, so 2 bytecodes must be executed in each loop: first `i+=step`, and then judge `i<=limit`. For performance, the judgment function of the first bytecode can also be added to the bottom bytecode, so that only one bytecode is executed each loop. The control structure diagram is as follows:

```
       +--------------+
       | i <= limit ? |--No--\ jump to the end if exceed limit
       +--------------+      |
/------>                     |
|       block                |
|                            |
|       +--------------+     |
|       | i += step    |     |
\--Yes--| i <= limit ? |     |
        +--------------+     |
            <----------------/
```

Add 2 new bytecodes:

```rust
pub enum ByteCode {
    // for-loop
    ForPrepare(u8, u16),
    ForLoop(u8, u16),
```

These two bytecodes correspond to the bytecodes of the two boxes in the above figure, and the two associated parameters are the stack start position and jump position respectively. Later, we will see that the first bytecode needs to do other preparations besides judging the jump, so it is called prepare.

Variable Storage

The first parameter associated with the above two bytecodes is the starting position of the stack. To be precise, it is the location where the above three values (init, limit, step) are stored. These 3 values naturally need to be stored on the stack, because one of the functions of the stack is to store temporary variables, and because there is no other place available. The 3 values are stored sequentially, so only one parameter is needed to locate 3 values.

In addition, the for statement also has a control variable, which can reuse the position on the stack of init. During parsing, create an internal temporary variable whose name is Name in BNF, pointing to the position of the first variable on the stack. In order to keep the positions of the other 2 temporary variables from being occupied, 2 more anonymous local variables need to be created. Therefore, the stack at execution time is as follows:

```
      |        |
sp    +--------+
      | init/i |  control variable Name
sp+1  +--------+
      | limit  |  anonymous variable ""
sp+2  +--------+
      | step   |  anonymous variable ""
      +--------+
      |        |
```

The numerical-for statement is special only in the above three temporary variables, and the rest is similar to the control structure introduced before, which is nothing more than jumping according to the conditional judgment statement. The syntax analysis code is as follows:

```
    fn for_numerical(&mut self, name: String) {
        self.lex.next(); // skip `=`

        // Read 3 expressions: init, limit, step (default is 1), and place
        // them on the stack in turn
        match self.explist() {
            2 => self.discharge(self.sp, ExpDesc::Integer(1)),
            3 => (),
            _ => panic!("invalid numerical-for exp"),
        }

        // Create 3 local variables to occupy the position on the stack.
        // Subsequent if the internal block needs local or temporary
variables,
        // The position after these 3 variables on the stack will be used.
        self.locals.push(name); // control variable, can be referenced in
internal block
        self.locals.push(String::from("")); // anonymous variable, purely
for placeholder
        self.locals.push(String::from("")); // Same as above

        self.lex.expect(Token::Do);

        // Generate ForPrepare bytecode
        self.byte_codes.push(ByteCode::ForPrepare(0, 0));
        let iprepare = self.byte_codes.len() - 1;
        let iname = self.sp - 3;

        self. push_loop_block();

        // inner block
        assert_eq!(self. block(), Token::End);

        // delete 3 temporary variables
        self. locals. pop();
        self. locals. pop();
        self. locals. pop();

        // Generate ForLoop bytecode and fix the previous ForPrepare
        let d = self.byte_codes.len() - iprepare;
        self.byte_codes.push(ByteCode::ForLoop(iname as u8, d as u16));
        self.byte_codes[iprepare] = ByteCode::ForPrepare(iname as u8, d as
u16);

        self.pop_loop_block(self.byte_codes.len() - 1);
    }
```

Integer and Float-point Types

The previously supported control statements(such as `if`, `while`) mainly introduce the syntax analysis part; while the virtual machine execution part only performs simple operations on the stack according to the bytecode. However, the syntax analysis part of the numerical-for loop is relatively simple (mainly because it is similar to the previous control structures), while the virtual machine execution part is very complicated. In fact, it is not difficult, it is just cumbersome. The reason is that Lua supports 2 numeric types, integers and floats. There are a total of 3 expressions (or called variables) in the numeric-for statement, `init`, `limit`, and `step`, each of which may be one of two types, and there are 8 possibilities in total. Although in some cases the type of some variables (such as constants) can be determined in the syntax analysis stage, it is of little significance to deal with this special case alone, and finally it is necessary to deal with all three variables of unknown type situation, which needs to be handled during the execution phase of the virtual machine.

It is too complicated to deal with 8 types one by one; and they cannot be completely classified into one type, because the representation ranges of integers and floating-point numbers are different. In this regard, the Lua language regulations is divided into two categories:

- If `init` and `step` are integers, then treat them as integers;
- Otherwise, handle them as floating point numbers.

As for why the second `limit` variable is not considered in the first category, it is not clear. I think there are some possible reasons, but I'm not sure about them, so I won't discuss them here. It can be realized according to the regulations of Lua. But it does introduce some complications.

Somewhere the 8 possibilities need to be grouped into the 2 types above. It can't be done in the syntax analysis phase, and it is too costly to perform each time the loop is executed, so it is classified once at the beginning of the loop. This is what the `ForPrepare` bytecode does:

- If `init` and `step` are integers, then convert `limit` to an integer;
- Otherwise, convert all 3 variables to floats.

In this way, each time the loop is executed, that is, the ForLoop bytecode, only two cases need to be handled.

It is easy to convert integers to floating-point numbers in the second category, but to convert the floating-point limit to integers in the first category, you must pay attention to the following two points:

- If `step` is positive, `limit` is rounded down; if `step` is negative, `limit` is rounded up.
- If the `limit` exceeds the representation range of the integer, then it is converted to the maximum or minimum value of the integer. There is an extreme situation here, such as `step` is negative, `init` is the maximum value of an integer, and `limit` exceeds the maximum value of an integer, then `init` is smaller than `limit`, and

because Lua clearly stipulates that the control variable of the numerical-for loop will not overflow and reverse, So the expectation is that the loop will not be executed. But according to the above conversion, `limit` is converted to the maximum value because it exceeds the maximum value of the integer, which is equal to `init`, and 1 cycle will be executed. Therefore, for special treatment, you can set `init` and `limit` to 0 and 1 respectively, so that the loop will not be executed.

The specific code for `limit` variable conversion is as follows:

```
fn for_int_limit(limit: f64, is_step_positive: bool, i: &mut i64) -> i64 {
     if is_step_positive {
         if limit < i64::MIN as f64 {
             *i = 0; // Modify init together to ensure that the loop will not
be executed
             -1
         } else {
             limit.floor() as i64 // round down
         }
     } else {
         if limit > i64::MAX as f64 {
             *i = 0;
             1
         } else {
             limit.ceil() as i64 // round up
         }
     }
}
```

Virtual Machine Execution

After introducing the above integer and floating-point number types and conversion details, the next step is to implement the virtual machine execution part of the two bytecodes.

The ForPrepare bytecode does two things: first, it is divided into integer and floating point type loops according to the variable type; Then compare `init` and `limit` to determine whether to execute the first cycle. code show as below:

```
                ByteCode::ForPrepare(dst, jmp) => {
                    // clear into 2 cases: integer and float
                    // stack: i, limit, step
                    if let (&Value::Integer(mut i), &Value::Integer(step)) =
                            (&self.stack[dst as usize], &self.stack[dst as usize + 2]) {
                        // integer case
                        if step == 0 {
                            panic!("0 step in numerical for");
                        }
                        let limit = match self.stack[dst as usize + 1] {
                            Value::Integer(limit) => limit,
                            Value::Float(limit) => {
                                let limit = for_int_limit(limit, step>0, &mut i);

                                self.set_stack(dst+1, Value::Integer(limit));
                                limit
                            }
                            // TODO convert string
                            _ => panic!("invalid limit type"),
                        };
                        if !for_check(i, limit, step>0) {
                            pc += jmp as usize;
                        }
                    } else {
                        // float case
                        let i = self.make_float(dst);
                        let limit = self.make_float(dst+1);
                        let step = self.make_float(dst+2);
                        if step == 0.0 {
                            panic!("0 step in numerical for");
                        }
                        if !for_check(i, limit, step>0.0) {
                            pc += jmp as usize;
                        }
                    }
                }
```

The ForLoop bytecode also does two things: first, add `step` to the control variable; then compare the control variable and `limit` to determine whether to execute the next loop. The code is omitted here.

So far, we have completed the numeric-for statement.

`goto` Statement

This section describes the `goto` statement.

The `goto` statement and label can be used together for more convenient code control. But the `goto` statement also has the following restrictions:

- You cannot jump to the label defined by the inner block, but you can jump to the outer block;
- You cannot jump outside the function (note that the above rule has restricted jumping into the function). Since we do not support functions yet, ignore this for now;
- You cannot jump into the scope of local variables, that is, you cannot skip local statements. Note here that the scope ends at the last non-void statement, and the label is considered a void statement. My personal understanding is the statement that does not generate bytecode. For example the following code:

```lua
while xx do
    if yy then goto continue end
    local var = 123
    -- some code
    ::continue::
end
```

The `continue` label is behind the local variable `var`, but because it is a void statement, it does not belong to the scope of var, so the above `goto` is a legal jump.

The implementation of the `goto` statement naturally uses `Jump` bytecode. The main task of syntax analysis is to match `goto` and label, and generate `Jump` bytecode at the place of `goto` statement to jump to the corresponding label. Since the `goto` statement can jump forward, the definition of the corresponding label may not be encountered when the `goto` statement is encountered; it can also jump backward, so when the label statement is encountered, it needs to be saved for subsequent `goto` matching. Therefore, two new lists need to be added to `ParseProto` to save the goto and label information encountered during syntax analysis:

```rust
struct GotoLabel {
    name: String, // The label name to jump to/defined
    icode: usize, // current bytecode index
    nvar: usize, // the current number of local variables, used to determine
whether to jump into the scope of local variables
}

pub struct ParseProto<R: Read> {
    gotos: Vec<GotoLabel>,
    labels: Vec<GotoLabel>,
```

Both lists have the same member type, `GotoLabel`. Among them, `nvar` is the current number of local variables. Make sure that the nvar corresponding to the paired `goto` statement cannot be smaller than the nvar corresponding to the label statement, otherwise it means that there is a new local variable definition between the `goto` and label statements, that is, `goto` jump into the scope of the local variable.

There are two implementations of matching `goto` statement and lable:

- One-time match at the end of the block:
    - When encountering a `goto` statement, create a new `GotoLabel` to join the list, and generate a placeholder Jump bytecode;
    - When encountering a label statement, create a new `GotoLabel` to add to the list.

  Finally, at the end of the block, match once and fix the placeholder bytecode.

- Live match:
    - When encountering a `goto` statement, try to match from the existing label list, if the match is successful, directly generate a complete Jump bytecode; otherwise create a new `GotoLabel`, and generate a placeholder Jump bytecode;
    - When encountering a label statement, try to match it from the existing `goto` list, and if it matches, repair the corresponding placeholder bytecode; since there may be other `goto` statements adjusted to this point, it is still necessary to create a new `GotoLabel`.

  At the end of the block, all matches have completed already.

It can be seen that although real-time matching is a little more complicated, it is more cohesive, and there is no need to execute a final function at the end. But this solution has a big problem: it is difficult to judge non-void statements. For example, in the example at the beginning of this section, when the `continue` label is parsed, it cannot be judged whether there are other non-void statements in the future. If there is, it is an illegal jump. It can only be judged after parsing to the end of the block. In the first one-time matching scheme, the matching is done at the end of the block. At this time, it is convenient to judge the non-void statement. Therefore, we choose one-time matching here. It should be noted that when Upvalue is introduced later, it will be found that the one-time matching scheme is flawed.

After introducing the above details, the overall process of syntax analysis is as follows:

- After entering the block, first record the number of `goto` and label before (outer layer);
- Parse block, record `goto` and label statement information;

- Before the end of the block, match the `goto` statement that appears in this block with all (including the outer layer) label statements: if there is a `goto` statement that is not matched, it will still be returned to the `goto` list, because it may be a jump to the block The label defined in the outer layer after exiting; finally delete all the labels defined in the block, because after exiting the block, there should be no other goto statements to jump in.
- Before the end of the entire Lua chunk, judge whether the `goto` list is empty. If it is not empty, it means that some `goto` statements have no destination, and an error is reported.

The corresponding code is as follows:

Record the number of `goto` and label existing in the outer layer at the beginning of parsing the block; and match and clean up the goto and label defined inside before the end of the block:

```
    fn block_scope(&mut self) -> Token {
        let igoto = self.gotos.len(); // Record the number of outer goto
before
        let ilabel = self.labels.len(); // record the number of outer labels
        loop {
            // omit other statement analysis
            t => { // end of block
                // Match goto and label before exiting the block
                self.close_goto_labels(igoto, ilabel);
                break t;
            }
        }
    }
```

The specific matching code is as follows:

```
    // The parameters igoto and ilable are the starting positions of goto
    //  and label defined in the current block
    fn close_goto_labels(&mut self, igoto: usize, ilabel: usize) {
        // Try to match "goto defined in the block" and "all labels".
        let mut no_dsts = Vec::new();
        for goto in self. gotos. drain(igoto..) {
            if let Some(label) = self.labels.iter().rev().find(|l|l.name ==
goto.name) { // matches
                if label.icode != self.byte_codes.len() && label.nvar >
goto.nvar {
                    // Check whether to jump into the scope of local
variables.
                    // 1. The bytecode corresponding to the label is not the
last one,
                    //    indicating that there are non-void statements in
the follow-up
                    // 2. The number of local variables corresponding to the
label is
                    //    greater than that of goto, indicating that there
are newly
                    //    defined local variables
                    panic!("goto jump into scope {}", goto.name);
                }
                let d = (label.icode as isize - goto.icode as isize) as i16;
                self.byte_codes[goto.icode] = ByteCode::Jump(d - 1); // fix
bytecode
            } else {
                // If there is no match, put it back
                no_dsts.push(goto);
            }
        }
        self. gotos. append(&mut no_dsts);

        // Delete the label defined inside the function
        self. labels. truncate(ilabel);
    }
```

Finally, before the chunk is parsed, check that all gotos are matched:

```
    fn chunk(&mut self) {
        assert_eq!(self. block(), Token::Eos);
        if let Some(goto) = self. gotos. first() {
            panic!("goto {} no destination", &goto.name);
        }
    }
```

This completes the `goto` statement.

Logical and Relational Operations

This chapter introduces logical operations and relational operations. Both types of operations have two application scenarios: conditional judgment and evaluation. For example the following code:

```lua
-- logic operation
if a and b then -- conditional judgment
     print(t.k or 0) -- evaluate
end

-- Relational operations
if a > b then -- conditional judgment
     print(c > d) -- evaluate
end

-- Combination of logical operations and relational operations
if a > b and c < d then -- conditional judgment
     print (x > 0 and x or -x) -- evaluate
end
```

The analysis methods in these two scenarios are slightly different. Generally speaking, conditional judgments occur more often than evaluations, so when introducing these two types of operations in this chapter, we first introduce the parsing in conditional judgment scenarios and optimize them; then complete the evaluation scenario.

The scene of conditional judgment is derived from the control structure in the previous chapter, which is why these two types of operations are not introduced immediately after arithmetic operations in Chapter 5, but must be introduced after the control structure.

Logical Operations in Conditional Judgment

Logical operations include 3: `and`, `or`, and `not`. The last `not` is a unary operation, which has been introduced in the previous section Unary Operation. This chapter only introduces the first two `and` and `or`.

Then why not introduce `and` and `or` in the previous binary operation section? Because of "short circuit"! In mainstream programming languages (such as C, Rust), logical operations are short-circuited. For example, for the AND operation, if the first operand is false, then there is no need (and cannot) to execute or check the second operand. For example, the statement `is_valid() and count()`, if the return value of `is_valid()` is false, then the subsequent `count()` cannot be executed. Therefore, the execution process of logical operations is: 1. First judge the left operand, 2. If it is false, exit, 3. Otherwise judge the right operand. While the execution process of the binary arithmetic operation is: 1. First find the left operand, 2. Then find the right operand, 3. Finally calculate. It can be seen that the flow of logical operations is different from that of arithmetic operations, so the previous methods cannot be applied.

Before introducing the logic operation in detail, let's look at two usage scenarios of logic operations:

1. As a judgment condition, such as the judgment condition statement in if, while and other statements in the previous chapter, such as `if t and t.k then ... end`;
2. Evaluation, such as `print(v>0 and v or -v)`.

In fact, the first scenario can be regarded as a special case of the second scenario. For example, the above if statement example is equivalent to the following code:

```
local tmp = t and t.k
if tmp then
    ...
end
```

It is to first evaluate the operation statement `t and t.k`, then put the value into a temporary variable, and finally judge whether the value is true or false to decide whether to jump. However, here we don't actually care whether the specific evaluation result is `t` or `t.k`, but only care about true or false, so we can save the temporary variable! As you can see below, the omission of temporary variables can save a bytecode, which is a great optimization. Since most applications of logical operations are in the first scenario, it is worthwhile to separate this scenario from the second general scenario for special optimization, by omitting temporary variables and directly judging whether to jump based on the evaluation result.

As the title of this section indicates, this section only introduces the first scenario; while the next section will introduce the second scenario.

Jump Rules

The short-circuit characteristics of logic operations are introduced above. After each operand is judged, a jump may occur and the next operand is skipped. The bytecode corresponding to the logical operation is to jump according to each operand. Different operation combinations will lead to various jump combinations. Now it is necessary to summarize jump rules from various jump combinations, so as to be used as subsequent parsing rules. This is probably the most convoluted part of the whole interpreter.

The following uses the simplest `if` statement as the application scenario, and first looks at the most basic and and or operations. The following two figures are the jump schematic diagrams of `if A and B then ... end` and `if X or Y then ... end` respectively:

```
 A and B                      X or Y

+-------+                    +-------+
|   A   +-False-\    /--True-+   X   |
+---+---+       |    |       +---+---+
    |True       |    |           |False
    V           |    |           V
+-------+       |    |       +-------+
|   B   +-False>+    |       |   Y   +-False-\
+---+---+       |    |       +---+---+       |
    |True       |    \---------->|True       |
    V           |                V           |
  block         |              block         |
    |           |                |           |
    +<----------/                +<--------/
    V                            V
```

The left figure is the AND operation. The processing after the judgment of the two operands A and B is the same: if True, continue to execute; if False, jump to the end of the code block.

The figure on the right is the OR operation. The processing flow of the two operands is different. The processing of the first operand X is: False continues execution, and True jumps to the following code block to start. While the processing of the second operand Y is the same as the processing of A and B before.

However, just looking at these two examples is not able to sum up the general law. Also need to look at some complex:

```
A and B and C               X or Y or Z                 (A and B) or Y
A and (X or Y)

+--------+                   +--------+                    +--------+
+--------+
|   A   +-False-\    /--True-+   X   |                    |   A   |-False-\
|   A   +-False-\
+---+----+      |    |       +---+----+                   +---+----+      |
+---+----+      |
    |True       |    |           |False                       |True       |
|True       |
    V           |    |           V                            V           |
V           |
+--------+      |    |       +--------+                   +--------+      |
+--------+      |
|   B   +-False>+    +<-True-+   Y   |             /--True-+   B   |       |
/--True-+   X   |       |
+---+----+      |    |       +---+----+             |       +---+----+      |
|       +---+----+      |
    |True       |    |           |False             |      False|<---------/
|           |False      |
    V           |    |           V                  |           V
|           V           |
+--------+      |    |       +--------+             |       +--------+
|       +--------+      |
|   C   +-False>+    |       |   Z   +-False-\     |       |   Y   +-False-\
|       |   Y   +-False>+
+---+----+      |    |       +---+----+       |     |       +---+----+      |
|       +---+----+      |
    |True       |    \---------->|True        |     \---------->|True       |
\---------->|True       |
    V           |                V            |                 V           |
V           |
  block         |              block          |               block         |
block         |
    |           |                |            |                 |           |
|           |
    +<---------/                 +<----------/                  +<---------/
+<---------/
    V                            V                              V
V
```

According to these 4 diagrams, the following rules can be summarized (the specific steps of induction are omitted here. In practice, more examples may be needed to summarize, but too many examples are too bloated):

- The jump condition depends on the logical operator (that is, `and` or `or`) behind the statement (such as A, B, X, Y, etc. in the above example):
    - If it is followed by `and` operation, False jumps and True continues execution. For example, A and B in the first picture are followed by and operations, so they are all False jumps.

  - If it is followed by an `or` operation, True jumps and False continues. For example, X and Z in the second picture are followed by or operations, so they are all True jumps.
  - If there is no logical operator behind, that is, the entire judgment statement ends, False jumps and True continues to execute. This rule is the same as for `and` above. This is true for the last judgment statement in the above four figures.

- Rules for jump target positions:
  - If the same jump condition continues, jump to the same position. For example, there are 3 consecutive False jumps in the first picture, and 2 consecutive True jumps in the second picture; and the two False jumps in the third picture are not continuous, so the jump positions are different. Then during syntax analysis, if the two operands have the same jump condition, the jump list is merged.
  - If different jump conditions are encountered, terminate the previous jump list and jump to the end of the current judgment statement. For example, the False of Z in the second figure terminates the previous two True jump lists and jumps to the end of the Z statement; another example is the False jump list before the termination of B's True in the third figure, and jumps to After the B statement.
  - However, the fourth picture does not seem to comply with the above two rules. The two False jumps are not continuous but connected, or the True jump of X does not end the False jump list of A. This is because A does not operate with `X`, but with `(X or Y)`; you need to ask `(X or Y)` first, and the True jump of X is brand new at this time, and you don't know the previous The False jump list of A; and then when asking `A and (X or Y)`, the two jump lists of True and False coexist; the False at the end of the final statement merges the False jump list of A before, and Termination of X's True jump list.
  - The end of the judgment statement corresponds to the False jump, so the True jump list will be terminated and the False jump list will continue. After the end of the block, terminate the False jumpGo to the end of the block list. This is the case in the 4 figures above.

So far, the preparation knowledge has been introduced. Let's start coding.

Bytecode

Several conditional judgment statements in the control structure in the previous chapter, including `if`, `while`, and `repeat..until`, etc., all deal with the judgment conditions and jump on False, so there is only one bytecode for testing and jumping, namely `Test`. But now we need 2 kinds of jumps, jump on False and jump on True. For this reason, we remove the previous `Test` and add 2 bytecodes:

```
pub enum ByteCode {
    TestAndJump(u8, i16), // If Test is True, then Jump.
    TestOrJump(u8, i16), // Jump if Test is False. Same function as `Test`
in the previous chapter.
```

The "And" and "Or" in the naming have nothing to do with the logical operations introduced in this section, but are derived from the method names of the Option and Error types in the Rust language, meaning "and then" and "otherwise then" respectively. However, in the two examples at the beginning of this section, `t and t.k` can be described as: if t exists "then then" take t.k, `t.k or 100` can be described as: if t.k exists then take its value "otherwise then" Take 100. It can also be said to be related.

It's just that the first jump rule introduced above, if it is followed by `and` operation, False jumps, corresponding to `TestOrJump`. The `and` and `or` here do not correspond, but it doesn't matter much.

In the official Lua implementation, there is still only one bytecode `TEST`, which is associated with two parameters: the stack address of the judgment condition (same as ours), and the jump condition (True jump or False jump). For the specific jump position, you need to add a `JUMP` bytecode for an unconditional jump. It seems that 2 bytecodes are not very efficient. This is done for another application scenario, which will be introduced in the next section.

ExpDesc

When parsing logical operators to generate jump bytecodes, the destination of the jump is not yet known. Only one bytecode placeholder can be generated first, and the parameter of the jump position is left blank. The parameters are filled in after the destination location is determined later. This approach is the same as when we introduced control structures in the previous chapter. The difference is that there was only one jump bytecode in the previous chapter, but this time there may be multiple bytecode zippers, such as the first picture above, 3 bytecode jumps Go to the same location. This zipper may be a True jump or a False jump, or these two chains may exist at the same time, such as when Y is resolved in the fourth figure above. So a new ExpDesc type is needed to save the jump list. To this end, a new `Test` type is defined as follows:

```
enum ExpDesc {
    Test(usize, Vec<usize>, Vec<usize>), // (condition, true-list, false-
list)
```

Associate 3 parameters. The first one is to determine the position of the condition on the stack. No matter what type (constant, variable, table index, etc.) it will be discharged to the stack first, and then the true or false will be judged. The next two parameters are the two jump lists of True and False, and the contents are the positions of the bytecodes that need to be completed.

In the official implementation of Lua, the jump list is implemented by jumping to the blank parameters in the bytecode. For example, if there are three consecutive False jumps in the first figure above, the bytecodes generated by judging A, B, and C are `JUMP 0`, `JUMP $A`, `JUMP $B`, and then save them in ExpDesc `$C`. In this way, `$B` can be found through `$C`, `$A` can be found through `$B`, and the parameter `0` indicates the end of the linked list. Finally, while traversing, it is uniformly fixed as `JUMP $end`. This design is very efficient, without additional storage, and the zipper can be realized by using the Jump parameter that is temporarily left blank. At the same time, it is also slightly obscure and error-prone. This kind of full use of resources and micro-manipulation of memory according to bits is a very typical practice of C language projects. The Rust language standard library provides a list Vec, although it will generate memory allocation on the heap, which slightly affects performance, but the logic is much clearer and clear at a glance. As long as it is not a performance bottleneck, obscure and dangerous practices should be avoided as much as possible, especially when using the safety-oriented Rust language.

Syntax Analysis

Now it is finally ready to parse. Start with the binary operation part of the `exp()` function. Before introducing the evaluation order of binary numerical operations, the first operand must be processed first. It is also introduced at the beginning of this section that for the processing order of logical operations, due to the short-circuit characteristics, the first operation and possible jumps must be processed first, and then the second operand can be parsed. So, before continuing to parse the second operand, the jump is handled:

```
    fn preprocess_binop_left(&mut self, left: ExpDesc, binop: &Token) ->
ExpDesc {
        match binop {
            Token::And => ExpDesc::Test(0, Vec::new(), self.
test_or_jump(left)),
            Token::Or => ExpDesc::Test(0, self. test_and_jump(left),
Vec::new()),

            _ => // Omit the part of other types of discharge
        }
    }
```

In this function, the processing part of logical operation is added. Take `and` as an example, generate `ExpDesc::Test` type, temporarily save the processed 2 jump lists, and the associated first parameter is useless, fill in 0 here. Call the `test_or_jump()` function to process the jump list. According to the rules introduced above, the and operator corresponds to the False jump, which will terminate the previous True jump list, so the `test_or_jump()` function will terminate the previous True jump list and return only the False jump list. Then create a new list `Vec::new()` here as the True jump list.

Look at the specific implementation of `test_or_jump()`:

```
    fn test_or_jump(&mut self, condition: ExpDesc) -> Vec<usize> {
        let (icondition, true_list, mut false_list) = match condition {
            // It is a constant of True, no need to test or jump, skip it
directly.
            // Example: while true do ... end
            ExpDesc::Boolean(true) | ExpDesc::Integer(_) | ExpDesc::Float(_)
| ExpDesc::String(_) => {
                return Vec::new();
            }

            // The first operand is already of type Test, indicating that
this
            // is not the first logical operator.
            // Just return the existing two jump lists directly.
            ExpDesc::Test(icondition, true_list, false_list) =>
                (icondition, Some(true_list), false_list),

            // The first operand is another type, indicating that this is
the
            // first logical operator.
            // Only need to discharge the first operand to the stack.
            // There was no True jump list before, so return None.
            // There was no False jump list before, so create a new list to
save
            // this jump instruction.
            _ => (self. discharge_any(condition), None, Vec::new()),
        };

        // generate TestOrJump, but leave the second parameter blank
        self.byte_codes.push(ByteCode::TestOrJump(icondition as u8, 0));

        // Put the newly generated bytecode, if it is in the False jump
list,
        // for subsequent repair
        false_list.push(self.byte_codes.len() - 1);

        // Finalize the previous True jump list and jump here, if any
        if let Some(true_list) = true_list {
            self.fix_test_list(true_list);
        }

        // return False jump list
        false_list
    }
```

For the `or` operator and the corresponding `test_and_jump()` function, it is similar, just flip the True and False jump lists. It will not be introduced here.

After processing the first operand and the jump, it is very simple to process the second operand, just connect the jump list:

```
    fn process_binop(&mut self, binop: Token, left: ExpDesc, right: ExpDesc)
-> ExpDesc {
        match binop {
            // omit other binary operator processing
            Token::And | Token::Or => {
                // The first operand has been converted to ExpDesc::Test in
preprocess_binop_left() above
                if let ExpDesc::Test(_, mut left_true_list, mut
left_false_list) = left {
                    let icondition = match right {
                        // If the second operand is also of Test type, such
as the example
                        // of `A and (X or Y)` in the fourth figure above in
this section,
                        // Then connect the two jump lists separately.
                        ExpDesc::Test(icondition, mut right_true_list, mut
right_false_list) => {
                            left_true_list.append(&mut right_true_list);
                            left_false_list.append(&mut right_false_list);
                            icondition
                        }
                        // If the second operand is another type, there is
no need to deal with the jump list
                        _ => self.discharge_any(right),
                    };

                    // After returning to the connection, I want to create a
new jump list
                    ExpDesc::Test(icondition, left_true_list,
left_false_list)
                } else {
                    panic!("impossible");
                }
            }
```

After dealing with the binary operation part, the next step is the application scenario. This section only introduces the application scenarios used as judgment conditions, and the evaluation will be introduced in the next section. Several control structure statements (if, while, repeat..until, etc.) directly process the jump bytecode, and the code logic is similar. In the jump rules introduced at the beginning of this section, the judgment statement of the entire logical operation ends, which is a False jump, so calling the test_or_jump() function just introduced can replace and simplify the code that directly processes bytecodes in the previous chapter logic. Here we still use the if statement as an example:

```
    fn do_if_block(&mut self, jmp_ends: &mut Vec<usize>) -> Token {
        let condition = self. exp();

        // In the previous chapter, here is to generate Test bytecode.
        // Now, replace and simplify to the test_or_jump() function.
        // Terminate the True jump list and return a new False jump list.
        let false_list = self. test_or_jump(condition);

        self.lex.expect(Token::Then);

        let end_token = self. block();

        if matches!(end_token, Token::Elseif | Token::Else) {
            self.byte_codes.push(ByteCode::Jump(0));
            jmp_ends.push(self.byte_codes.len() - 1);
        }

        // In the last chapter, here is to fix a Test bytecode just
generated.
        // Now, a False jump list needs to be modified.
        self.fix_test_list(false_list);

        end_token
    }
```

This completes the syntax analysis part.

Virtual Machine Execution

The execution part of the virtual machine first needs to process the newly added 2 bytecodes, which are very simple and will be ignored here. What needs to be said is the details of a stack operation. The function when assigning a value to the stack before is as follows:

```
    fn set_stack(&mut self, dst: u8, v: Value) {
        let dst = dst as usize;
        match dst.cmp(&self.stack.len()) {
            Ordering::Equal => self. stack. push(v),
            Ordering::Less => self.stack[dst] = v,
            Ordering::Greater => panic!("fail in set_stack"),
        }
    }
```

First determine whether the target address dst is within the range of the stack:

- If it is, assign it directly;
- If it is not and it is just the next position, use `push()` to push it onto the stack;
- If not, and past the next position, it was impossible to appear before, so call

`panic!()` .

However, the short-circuit characteristics of logic operations may lead to the above-mentioned third situation. For example the following statement:

```
if (g1 or g2) and g3 then
end
```

According to our analysis method, the following temporary variables will be generated, occupying the position on the stack:

```
|       |
+-------+
|  g1   |
+-------+
|  g2   |
+-------+
|  g3   |
+-------+
|       |
```

But during execution, if `g1` is true, the processing of `g2` will be skipped, and `g3` will be processed directly. At this time, the position of g2 in the above figure is not set, then g3 will exceed the top of the stack position, as shown in the figure below:

```
|       |
+-------+
|  g1   |
+-------+
|       |
:       :
:       : <-- set g3, beyond the top of the stack
```

Therefore, it is necessary to modify the above `set_stack()` function to support setting elements beyond the top of the stack. This can be achieved by calling `set_vec()` .

Test

So far, the application scenario of logical operation in conditional judgment has been completed. This can be tested with the examples in the figures at the beginning of this section. omitted here.

Logical Operations in Evaluation

The previous section introduced the logical operations in conditional judgment. This section introduces another scenario, that is, the evaluation.

In the previous section, the syntax analysis process of logical operations in *conditional judgment* scenarios can be divided into two parts:

- Process the logical operation itself, specifically, after encountering the `and` or `or` operator in the `exp()` function, generate the corresponding bytecode and process the True and False jump lists;
- After the entire logic operation statement is parsed, put the parsing result into the conditional judgment scene of the `if` statement, first terminate the True jump list, and then terminate the False jump list after the end of the block.

In the *evaluation* scenario to be introduced in this section, it is also divided into two parts:

- Dealing with the logical operation itself, this part is exactly the same as the previous section;
- After the entire logical operation statement is parsed, the statement is *evaluated*, which is the part to be introduced in this section.

As shown in the figure below, the previous section completed parts (a) and (b), and this section implements part (c) on the basis of (a).

```
                                                  +------------------------+
+---------------------+                      /--->| (b) Condition judgment |
| (a) Process         |   ExpDesc::Test     |     +------------------------+
| logical operations  |-------------------->+
+---------------------+                     |     +------------------+
                                             \--->| (c) Evaluation   |
                                                  +------------------+
```

Result Type

Logical operations in Lua are different from those in C and Rust. The results of logical operations in C and Rust languages are Boolean types, which only distinguish between true and false. For example, the following C language code:

```c
int i=10, j=11;
printf("%d\n", i && j); // output: 1
```

Will output `1`, because the `&&` operator will first convert the two operands to Boolean type (both are true in this example), and then execute the `&&` operation, the result is true, which is `1` in C language. The Rust language is stricter, both operands of `&&` must be of Boolean type, so the result is also of Boolean type.

But logical operations in Lua evaluate to the last *evaluated* operand. For example, the following are very common usages:

- `print(t and t.k)`, first judge whether `t` exists, and then find the index of `t`. If `t` does not exist, then there is no need to judge `t.k`, so the result is `t` which is `nil`; otherwise, it is `t.k`.
- `print(t.k or 100)`, index the table and provide a default value. First judge whether there is `k` in `t`, if there is, then there is no need to judge `100`, so the result is `t.k`; otherwise it is `100`.
- `print(v>0 and v or -v)`, find the absolute value. The result is `v` if positive, and `-v` otherwise. Simulates the `?:` ternary operator in C.

Evaluation Rules

In order to understand the sentence "the evaluation result of a logical operation is the last evaluated operand" more clearly, some examples are shown below. Here we still use the flowchart at the beginning of the previous section as an example. Let's look at the most basic operations first:

```
  A and B                      X or Y

 +-------+                    +-------+
 |   A   +-False-\    /--True-+   X   |
 +---+---+       |    |       +---+---+
     |True       |    |           |False
     V           |    |           V
 +-------+       |    |       +-------+
 |   B   |       |    |       |   Y   |
 +---+---+       |    |       +---+---+
     |<----------/    \---------->|
     V                            V
```

In the figure on the left, if A is False, the evaluation result is A; otherwise, when B is evaluated, since B is the last operand, there is no need to make a judgment, and B is the evaluation result.

In the figure on the right, if X is True, the evaluation result is X; otherwise, when Y is evaluated, since Y is the last operand, there is no need to make a judgment, and Y is the evaluation result.

Let's look at a few more complex examples:

```
A and B and C               X or Y or Z                  (A and B) or Y
A and (X or Y)

+-------+                    +-------+                     +-------+
+-------+
|   A   +-False-\    /--True-+   X   |                     |   A   |-False-\
|   A   +-False-\
+---+---+       |    |       +---+---+                     +---+---+       |
+---+---+       |
    |True       |    |           |False                        |True       |
|True       |
    V           |    |           V                             V           |
V           |
+-------+       |    |       +-------+                     +-------+       |
+-------+       |
|   B   +-False>+    +<-True-+   Y   |             /--True-+   B   |       |
/--True-+   X   |       |
+---+---+       |    |       +---+---+             |       +---+---+       |
|       +---+---+       |
    |True       |    |           |False            |      False|<---------/
|           |False       |
    V           |    |           V                 |           V
|           V           |
+-------+       |    |       +-------+             |       +-------+
|       +-------+       |
|   C   |       |    |       |   Z   |             |       |   Y   |
|       |   Y   |       |
+---+---+       |    |       +---+---+             |       +---+---+
|       +---+---+       |
    |<---------/     \---------->|                 \---------->|
\---------->|<---------/
    V                            V                             V
V
```

The process of summarizing based on these 4 figures is omitted here, and the evaluation rules are directly given:

1. The last operand does not need to be judged, as long as the previous judgment does not skip the last operand, then the last operand is the final evaluation result. For example, in the first figure above, if both A and B are True, then C will be executed, and C is the evaluation result of the entire statement. C itself does not need to make judgments.

2. In the syntax analysis stage, after the parsing of the entire logical operation statement is completed, the operands on the unterminated jump list may be used as the final evaluation result. This statement is rather convoluted, and the following example illustrates it. For example, in the first figure above, the True jump lists of A and B end in B and C respectively, but the False jump lists are not terminated, then

both A and B may be the final evaluation results, for example, if A is False Then A is the final evaluation result. As another counter-example, for example, the two jump lists of A's True and False in the third figure above are terminated in B and Y respectively, that is to say, when the entire statement is parsed, the jump lists of A are terminated. , then A cannot be the evaluation result, and in either case A will not reach the end of the statement. Except for the third figure, all judgment conditions in other figures may be used as the final evaluation result.

After summarizing the evaluation rules, let's start coding.

ExpDesc

A new ExpDesc type representing logical operations was introduced in the previous section and is defined as follows:

```
enum ExpDesc {
     Test(usize, Vec<usize>, Vec<usize>), // (condition, true-list, false-
list)
```

The latter two parameters respectively represent two jump linked lists, which will not be introduced here, and focus on the first parameter: the position of the judgment conditional statement on the stack. As mentioned in the previous section, all statements (such as variables, constants, table indexes, etc.) must be discharged to the stack first to determine whether they are true or false, so here we can use the stack index of `usize` type to represent the statement. This is no problem in the previous section, but in the evaluation scenario in this section, as mentioned above, the last operand does not need to be judged, so it may not need to be discharged to the stack. Like the following example:

```
local x = t and t.k
```

According to the current practice, first discharge the second operand t.k to a temporary variable on the stack; if t is true, assign the temporary variable to x through `Move` bytecode. Obviously this temporary variable is unnecessary, and t.k can be directly assigned to x. To do this, we need to delay the evaluation of the conditional statement, or delay the discharge. Then you need to transform the `ExpDesc::Test` type.

Lua's official approach is to assign two jump lists to all types of ExpDesc:

```
typedef struct expdesc {
   expkind k; // type tag
   union {
     // Data associated with various expkinds, omitted here
   } u;
   int t; /* patch list of 'exit when true' */
   int f; /* patch list of 'exit when false' */
} expdesc;
```

`t` and `f` in the above code are the jump lists of True and False respectively. But it is a bit inconvenient to define it in the Rust language. Because Rust's enum includes tags and associated data, corresponding to `k` and `u` above, one enum can define ExpDesc; but if you add two jump lists, you need to encapsulate a layer of struct outside Defined. And the struct variable is defined in the Rust languageWhen all members must be explicitly initialized, then in all places where ExpDesc is defined in the code, `t` and `f` must be initialized to Vec::new(). It's not worth it to affect other types for this one type.

Our approach is to define ExpDesc::Test recursively. Change the first parameter type of `ExpDesc::Test` from `usize` to `ExpDesc`. Of course, it cannot be defined directly, but it needs to encapsulate a layer of Box pointer:

```
enum ExpDesc {
     Test(Box<ExpDesc>, Vec<usize>, Vec<usize>), // (condition, true-list,
false-list)
```

This definition has no effect on other types of ExpDesc in the existing code. For the `Test` type in the existing code, it is only necessary to remove the discharge processing.

Bytecode

The functions of the two new bytecodes `TestAndJump` and `TestOrJump` in the previous section are both: "test" + "jump". And the function we need now is: "test" + "assignment" + "jump". To this end, we add 2 more bytecodes:

```
pub enum ByteCode {
     Jump(i16),
     TestAndJump(u8, i16),
     TestOrJump(u8, i16),
     TestAndSetJump(u8, u8, u8), // add
     TestOrSetJump(u8, u8, u8), // add
```

The function of `TestAndSetJump` is: if the value of the first parameter is tested to be true, it is assigned to the stack position of the second parameter and jumps to the bytecode position of the third parameter. Similar to `TestOrSetJump`.

Here comes a problem. In the previous jump bytecodes (the first 3 in the above code), the jump parameters are all 2 bytes, `i16` type, and the range of jumps can be large. And the newly added 2 bytecodes are associated with 3 parameters, so there is only one byte left for the jump parameter.

This is why, as mentioned in the previous section, in the official implementation of Lua, 2 bytecodes are used to represent conditional jump instructions. For example, as opposed to `TestAndJump(t, jmp)` , it is `TEST(t, 0); JUMP(jmp)` ; and in the evaluation scenario introduced in this section, it is necessary to add a target address parameter dst, which is `TESTSET (dst, t, 0); JUMP(jmp)` . This ensures that the jump parameter has 2 bytes of space. Moreover, although there are 2 bytecodes, during the execution of the virtual machine, when the `TEST` or `TESTSET` bytecode is executed, if a jump is required, the parameter of the next bytecode JUMP can be directly removed And execute the jump without having to do another instruction dispatch for the JUMP. It is equivalent to 1 bytecode, and JUMP is only used as an extended parameter, so it does not affect the performance during execution.

But we still use 1 byte code here, and use 1 byte to represent the jump parameter. In the conditional judgment scenario in the previous section, the judgment of the last operand is to jump to the end of the entire block, and the jump distance may be very long, requiring 2 bytes of space. In the evaluation scenario in this section, only jumps are made within the logic operation statement. You can refer to the above 6 figures, and the jump distance will not be very long; and since it only jumps forward, there is no need to represent negative numbers. So 1 byte `u8` type means that 256 distances are enough to cover. When conditions permit, 1 bytecode is always better than 2.

Syntax Analysis

After introducing the above modification points, now start the syntax analysis. The so-called evaluation is discharge. So we only need to complete the `ExpDesc::Test` type in the `discharge()` function. In the previous section, this is not complete. The specific discharge method is: first discharge the recursively defined conditional statement, and then repair the judgment bytecodes in the two jump lists.

```
    fn discharge(&mut self, dst: usize, desc: ExpDesc) {
        let code = match desc {
            // omit other types
            ExpDesc::Test(condition, true_list, false_list) => {
                // fix TestSet list after discharging

                // first discharge the recursively defined conditional
 statement
                self.discharge(dst, *condition);

                // Fix the judgment bytecode in the True jump list
                self.fix_test_set_list(true_list, dst);
                // Fix the judgment bytecode in the False jump list
                self.fix_test_set_list(false_list, dst);
                return;
            }
```

Fixing the jump list `fix_test_set_list()` function needs to do 2 things:

- fill jump parameters that were left blank before;
- Replace the previously generated `TestAndJump` and `TestOrJump` bytecodes with `TestAndSetJump` and `TestOrSetJump` respectively.

The specific code is as follows:

```
    fn fix_test_set_list(&mut self, list: Vec<usize>, dst: usize) {
        let here = self.byte_codes.len();
        let dst = dst as u8;
        for i in list.into_iter() {
            let jmp = here - i - 1; // should not be negative
            let code = match self. byte_codes[i] {
                ByteCode::TestOrJump(icondition, 0) =>
                    if icondition == dst {
                        // If the conditional statement is just at the
target position,
                        // there is no need to change it to TestAndSetJump
                        ByteCode::TestOrJump(icondition, jmp as i16)
                    } else {
                        // Modify to TestAndSetJump bytecode
                        ByteCode::TestOrSetJump(dst as u8, icondition, jmp
as u8)
                    }
                ByteCode::TestAndJump(icondition, 0) =>
                    if icondition == dst {
                        ByteCode::TestAndJump(icondition, jmp as i16)
                    } else {
                        ByteCode::TestAndSetJump(dst as u8, icondition, jmp
as u8)
                    }
                _ => panic!("invalid Test"),
            };
            self.byte_codes[i] = code;
        }
    }
```

Test

So far, the application scenario of logical operations in evaluation has been completed. This can be tested with the examples in the figures at the beginning of this section. omitted here.

Relational Operations in Conditional Judgment

The previous two sections describe logical operations, and the next two describe relational operations.

Relational operations, that is, compares, have 6 operators: equal, not equal, greater than, less than, greater than or equal to, less than or equal to. When introducing logical operations in the previous two sections, it was said that logical operations cannot use the analysis process of binary numerical operations in Chapter 5 because of the short-circuit feature. The relational operations did not use the parsing process in Chapter 5, for a different reason: for performance.

If performance is not considered, relational operations can use the parsing process in Chapter 5. For example, for the equal operation, the following bytecode can be generated: `EQ $r $a $b`, that is, compare `a` and `b`, and assign the Boolean result to `r`. If performance is to be considered, it depends on the application scenarios of relational operations. This part is almost the same as the logical operations introduced in the previous two sections, and there are also two application scenarios:

1. As a judgment condition, such as the judgment condition statement in the if, while and other statements in the previous chapter, such as `if a == b then ...`;
2. Evaluation, such as `print(a == b)`.

Like logical operations, the first scenario can be regarded as a simplified version of the second scenario. It does not require specific evaluation, but only needs to judge whether it is true or false. For example, the example of the if statement above can also be interpreted according to the second scenario. It is considered that `a == b` is first evaluated to a temporary variable, and then it is judged whether the temporary variable is true to decide whether to jump. Temporary variables can be omitted here! Since most applications of relational computing are in the first scenario, it is worthwhile to separate this scenario from the second general scenario for special optimization, by omitting temporary variables and directly judging whether to jump based on the evaluation result.

As the title of this section indicates, this section only introduces the first scenario; the next section will introduce the second scenario.

Bytecode

Still using the `if` statement and the equal operation as an example, in the `if a == b then ... end` scenario, the first bytecode sequence that comes to mind is as follows:

```
EQ $tmp $a $b    # Compare whether a and b are equal, and the result is
stored in a temporary variable
TEST $tmp $jmp   # Determine whether to jump according to the temporary
variable
```

Now save the temporary variable $tmp and merge the two bytecodes, as follows:

```
EQ $a $b $jmp   # Compare whether a and b are equal to decide whether to jump
```

But the problem is that this requires 3 parameters, leaving only 1 byte of space for the last jump parameter, indicating that the range is too small. For this reason, it can be split into 2 bytecodes:

```
EQ $a $b      # Determine whether a and b are equal, if they are equal, skip
the next statement, ie pc++
JUMP $jmp     # unconditional jump
```

In this way, 2 bytes can be used to represent the jump parameter. However, since 2 bytecodes are still needed, what is the difference from the original "EQ+TEST" scheme? Why make it so complicated?

- When the virtual machine is executing, if it is judged that `a` and `b` are equal and the following JUMP bytecode is skipped, then only 1 bytecode is executed; while the original "EQ+TEST" scheme always executes 2 bytes code. I don't know the probability that the if statement is true, but the probability of the while statement is true is still very high, so this is equivalent to saving the execution of 1 bytecode with a high probability;

- Even if the judgment is false and the following JUMP bytecode needs to be executed, then the next bytecode can be read directly when the EQ bytecode is executed, without having to go through another instruction distribution. The JUMP bytecode here is equivalent to an extended parameter of the EQ bytecode, rather than an independently executed bytecode. This is what Lua's official implementation does. This is also because the type of bytecode can be ignored in C language, and the parameters in the bytecode can be directly read through bit operations. But in the Rust language, if unsafe is not used, the enum tag cannot be ignored and the parameters can be read directly, so this optimization cannot be implemented in our interpreter.

- We can directly decide whether to jump or not according to the judgment result. In the original "EQ+TEST" scheme, it is necessary to write the judgment result into a temporary variable on the stack first, then read the temporary variable when the TEST bytecode is executed, and then judge true or false again, thus adding a temporary variable Reading and writing, but also a true or false judgment.

The advantage is such an advantage. Yes, but not much. Especially compared with the implementation complexity it brings, it is even less. The original "EQ+TEST" scheme only needs to add a few operators to the Binary Numerical Operation introduced earlier; but the new scheme needs to be described earlier logical operation coordination. However, we still choose to follow the official implementation of Lua, and trade the complexity of the implementation for some execution efficiency optimization.

In addition, regarding the types of the two operands in the bytecode, according to the previous description of Bytecode Parameter Type, it is similar to the bytecode of the binary value operation, each relational operator also corresponds to 3 bytecodes, for example, for equality operators: `Equal` , `EqualInt` and `EqualConst` , a total of 3 bytecodes. A total of 6 relational operators are 18 bytecodes.

Combined with Logical Operations

Combining relational and logical operations is very common. Take the `a>b and b<c` statement as an example. According to the introduction in the previous two sections, this is a logical operation statement. The two operands are `a>b` and `b<c` respectively. The operand discharges to a temporary variable on the stack in order to judge true or false. In order to avoid the use of temporary variables here, it is necessary to make relational operations and logical operations cooperate with each other.

For relational operation statements, the ExpDesc type needs to be added: `Compare` . Let's see what parameters need to be associated with this type if it is to be combined with logical operations, that is, for logical operation statements that use relational operations as operands.

First of all, if it is not converted to the `ExpDesc::Test` type, then the `Compare` type needs to maintain two jump lists of True and False;

Secondly, for the two jumps of True and False, the previous logical operations are distinguished by 2 bytecodes, `TestAndJump` and `TestOrJump` . The same can be done for relational operations, such as `EqualTrue` and `EqualFalse` bytecodes for equal operations. However, the relational operators have a total of 18 bytecodes. If each bytecode needs to distinguish between True and False jumps, then 36 bytecodes are required. That's too many! Fortunately, there is another method. The `EQ` bytecode introduced above only has 2 parameters, and a Boolean parameter can be added to indicate whether to jump True or False.

Finally, for the two jumps of True and False, it needs to be determined according to the logical operator behind it. For example, in the above example of `a>b and b<c` , it cannot be determined when it is parsed to `a>b` , but it can only be determined when it is parsed to `and` . Therefore, the complete bytecode cannot be generated when parsing the relational operation statement, so the relevant information can only be stored in the

`Compare` type first, and then the bytecode is generated after the jump type is determined.

In summary, the new types of relational operations are defined as follows:

```
enum ExpDesc {
    Compare(fn(u8,u8,bool)->ByteCode, usize, usize, Vec<usize>, Vec<usize>),
```

The first 3 parameters are bytecode type and the first 2 parameters are used to generate bytecode after determining the jump type; the latter 2 parameters are True and False jump lists. The whole type is equivalent to the combination of `BinaryOp` and `Test` types.

Here is the same problem as the logical operation introduced earlier. When the bytecode is generated, the destination address of the jump cannot be determined, and the complete bytecode cannot be generated immediately. It needs to be processed after determining the destination address. . However, this is different from the previous logical operation solution. The previous logical operation method is: first generate a bytecode placeholder, and only leave the parameters of the jump destination address blank; after determining the destination address, fix the corresponding parameters in the bytecode (`fix_test_list()` function). The method of relational operation here is to store all the information in `ExpDesc::Compare` (causing the definition of this type to be very long), and then directly generate the complete bytecode after the destination address is determined later.

In fact, for the processing of relational operations, theoretically, logical operations can also be used to generate bytecodes and then repair them. However, there are 18 bytecodes corresponding to relational operations, which is too many. If you still follow `fix_test_list()` the method of function matching first and then generating bytecode, the code is too complicated. If it is in the C language, the parameters in the bytecode can be directly corrected by bit operations, regardless of the bytecode type; while directly modifying the associated parameters in the enum in Rust requires unsafe.

Another difference is that when parsing logical operations, bytecodes must be generated immediately to take place. The `Compare` type operand of the relational operation will determine the jump type in the `test_or_jump()` function immediately after, and then the bytecode can be generated, so there is no need to occupy a place, and there is no need to generate a word first. section code then fixed it again.

Syntax Analysis

The syntax analysis of relational operations is divided into two parts:

- The parsing operation itself generates the corresponding `ExpDesc::Compare` according to the operator. This part is similar to Binary Numerical Operation, which is skipped here.

- The combination of relational operations and logical operations, that is, the combination of `ExpDesc::Compare` and `ExpDesc::Test` . In the previous analysis of logical operations, the processing of `ExpDesc::Compare` has been added.

For example, when the left operand is logically operated, bytecode is generated and two jump lists are processed:

```
    fn test_or_jump(&mut self, condition: ExpDesc) -> Vec<usize> {
        let (code, true_list, mut false_list) = match condition {
            ExpDesc::Boolean(true) | ExpDesc::Integer(_) | ExpDesc::Float(_)
| ExpDesc::String(_) => {
                return Vec::new();
            }
            // Add a Compare type.
            // Generate 2 bytecodes.
            // The two jump lists are handled in the same way as
`ExpDesc::Test` below.
            ExpDesc::Compare(op, left, right, true_list, false_list) => {
                // If it is determined to be a True jump, that is, the
associated
                // third parameter, the complete bytecode can be generated.
                self.byte_codes.push(op(left as u8, right as u8, true));

                // Generate Jump bytecode, but the jump destination address
is not
                // yet known, and subsequent repairs are required. to this
end,
                // Add processing of Jump bytecode in fix_test_list().
                (ByteCode::Jump(0), Some(true_list), false_list)
            }
            ExpDesc::Test(condition, true_list, false_list) => {
                let icondition = self.discharge_any(*condition);
                (ByteCode::TestOrJump(icondition as u8, 0), Some(true_list),
false_list)
            }
            _ => {
                let icondition = self.discharge_any(condition);
                (ByteCode::TestOrJump(icondition as u8, 0), None,
Vec::new())
            }
        };
```

now dealing with the right operand:

```
    fn process_binop(&mut self, binop: Token, left: ExpDesc, right: ExpDesc)
-> ExpDesc {
        match binop {
            Token::And | Token::Or => {
                if let ExpDesc::Test(_, mut left_true_list, mut
left_false_list) = left {
                    match right {
                        // Add a Compare type.
                        // The processing method is similar to the
`ExpDesc::Test` type below.
                        ExpDesc::Compare(op, l, r, mut right_true_list, mut
right_false_list) => {
                            left_true_list.append(&mut right_true_list);
                            left_false_list.append(&mut right_false_list);
                            ExpDesc::Compare(op, l, r, left_true_list,
left_false_list)
                        }
                        ExpDesc::Test(condition, mut right_true_list, mut
right_false_list) => {
                            left_true_list.append(&mut right_true_list);
                            left_false_list.append(&mut right_false_list);
                            ExpDesc::Test(condition, left_true_list,
left_false_list)
                        }
                        _ => ExpDesc::Test(Box::new(right), left_true_list,
left_false_list),
                    }
                } else {
                    panic!("impossible");
                }
            }
```

Virtual Machine Execution

There are 6 relational operators in total. Since we have previously implemented the `Eq` trait for `Value`, the equal and not equal operations can use `==` and `!=` to directly compare the Value operands. But for the other 4 operators, you need to implement a new trait for `Value`, which is `PartialOrd`. The reason why it is not `Ord` is because different types of Value cannot be compared in size. There is no need to use `PartialEq` because different types of Value can be compared for equality, and the return result is False. For example, the following two statements:

```
print (123 == 'hello') -- prints false
print (123 > 'hello') -- throw exception
```

Lua's comparison operators only support numeric and string types. So the `PartialOrd` implementation of `Value` is as follows:

```
impl PartialOrd for Value {
    fn partial_cmp(&self, other: &Self) -> Option<std::cmp::Ordering> {
        match (self, other) {
            // numbers
            (Value::Integer(i1), Value::Integer(i2)) => Some(i1.cmp(i2)),
            (Value::Integer(i), Value::Float(f)) => (*i as
f64).partial_cmp(f),
            (Value::Float(f), Value::Integer(i)) => f.partial_cmp(&(*i as
f64)),
            (Value::Float(f1), Value::Float(f2)) => f1.partial_cmp(f2),

            // strings
            (Value::ShortStr(len1, s1), Value::ShortStr(len2, s2)) =>
Some(s1[..*len1 as usize].cmp(&s2[..*len2 as usize])),
            (Value::MidStr(s1), Value::MidStr(s2)) => Some(s1.1[..s1.0 as
usize].cmp(&s2.1[..s2.0 as usize])),
            (Value::LongStr(s1), Value::LongStr(s2)) => Some(s1.cmp(s2)),

            // strings of different types
            (Value::ShortStr(len1, s1), Value::MidStr(s2)) => Some(s1[..*len1
as usize].cmp(&s2.1[..s2.0 as usize])),
            (Value::ShortStr(len1, s1), Value::LongStr(s2)) =>
Some(s1[..*len1 as usize].cmp(s2)),
            (Value::MidStr(s1), Value::ShortStr(len2, s2)) =>
Some(s1.1[..s1.0 as usize].cmp(&s2[..*len2 as usize])),
            (Value::MidStr(s1), Value::LongStr(s2)) => Some(s1.1[..s1.0 as
usize].cmp(s2)),
            (Value::LongStr(s1), Value::ShortStr(len2, s2)) =>
Some(s1.as_ref().as_slice().cmp(&s2[..*len2 as usize])),
            (Value::LongStr(s1), Value::MidStr(s2)) =>
Some(s1.as_ref().as_slice().cmp(&s2.1[..s2.0 as usize])),

            (_, _) => None,
        }
    }
}
```

For floating-point numbers, the `partial_cmp()` method needs to be called because the `Nan` of floating-point numbers cannot be compared.

Types that implement the `PartialOrd` trait can directly use several relatively large symbols such as `>`, `<`, `>=`, and `<=`. But `PartialOrd` actually has 3 return results for comparison: true, false, and not comparable. Corresponding to the Lua language, they are true, false, and throw an exception. However, the above-mentioned 4 comparison symbols can only give 2 results, and return false if they cannot be compared. So in order to be able to judge the situation that cannot be compared, we cannot use these 4 symbols directly, but use the original `partial_cmp()` function. The following is the execution code of `LesEq` and `Less` two bytecodes:

```
            ByteCode::LesEq(a, b, r) => {
                let cmp = &self.stack[a as usize].partial_cmp(&self.stack[b as
 usize]).unwrap();
                if !matches!(cmp, Ordering::Greater) == r {
                    pc += 1;
                }
            }
            ByteCode::Less(a, b, r) => {
                let cmp = &self.stack[a as usize].partial_cmp(&self.stack[b as
 usize]).unwrap();
                if matches!(cmp, Ordering::Less) == r {
                    pc += 1;
                }
            }
```

Here `unwarp()` is used to throw an exception. In the follow-up, when standardizing error handling, improvements need to be made here.

Relational Operations in Evaluation

The previous section introduced the relational operations in *conditional judgment*. This section introduces another scenario, that is, the processing during *evaluation*.

Similar to logical operations, to process the relationship judgment in the evaluation, we only need to discharge the `ExpDesc::Compare` parsed in the previous section to the stack. As shown in the figure below, the previous section completed parts (a) and (b), and this section implements part (c) on the basis of (a).

```
                                                   +------------------------+
+-----------------------+                     /--->| (b) Condition judgment |
| (a) Process           |   ExpDesc::Test    |     +------------------------+
| relational operations |------------------->+
+-----------------------+                    |     +-----------------+
                                              \--->| (c) Evaluation  |
                                                   +-----------------+
```

The evaluation of the logical operation is to replace the `TestAndJump` and `TestOrJump` bytecodes in the two jump lists with `TestAndSetJump` and `TestOrSetJump` respectively. For relational operations, although we can also do it like this, it would be too verbose to add a Set version to all 18 bytecodes. Here we refer to the official implementation of Lua. For the following Lua code:

```
print(123 == 456)
```

Compile the available bytecode sequence:

```
luac  -l tt.lua

main <tt.lua:0,0> (9 instructions at 0x6000037fc080)
0+ params, 2 slots, 1 upvalue, 0 locals, 1 constant, 0 functions
    1   [1]     VARARGPREP      0
    2   [1]     GETTABUP        0 0 0   ; _ENV "print"
    3   [1]     LOADI           1 456
    4   [1]     EQI             1 123 1
    5   [1]     JMP             1       ; to 7
    6   [1]     LFALSESKIP      1
    7   [1]     LOADTRUE        1
    8   [1]     CALL            0 2 1   ; 1 in 0 out
    9   [1]     RETURN          0 1 1   ; 0 out
```

Among them, the 4th and 5th bytecodes are comparison operations. The key lies in the following two bytecodes:

- The sixth bytecode `LFALSESKIP` is specially used for the evaluation of relational operations. The function is to set False to the target address and skip the next

statement;
- The seventh bytecode `LOADTRUE`, the function is to load True to the target address.

These two bytecodes, together with the 4th and 5th bytecodes above, can realize the function of finding Boolean values:

- If the fourth bytecode comparison result is true, execute the JMP of the fifth, skip the next statement, execute the seventh statement, and set True;
- If the comparison result of the fourth bytecode is false, then skip the fifth article, and execute the LFALSESKIP of the sixth article, set False and skip the next article.

This is very clever, but also very long-winded. If you follow the previous method of Binary Arithmetic Operation, the above function only needs one bytecode: `EQ $dst $a $b`. The reason why it is so complicated now is to optimize for relational operations in *conditional judgment* scenarios, thus hurting performance in *evaluation* scenarios, after all, the latter appears too little.

Syntax Analysis

The evaluation process is to discharge `ExpDesc::Compare` onto the stack,

```
        fn discharge(&mut self, dst: usize, desc: ExpDesc) {
            let code = match desc {
                // omit other types of processing

                // Evaluation of the logical operations introduced earlier
                ExpDesc::Test(condition, true_list, false_list) => {
                    self.discharge(dst, *condition);
                    self.fix_test_set_list(true_list, dst);
                    self.fix_test_set_list(false_list, dst);
                    return;
                }

                // evaluation of relational operations
                ExpDesc::Compare(op, left, right, true_list, false_list) => {
                    // Generate 2 bytecodes for relational operations
                    self.byte_codes.push(op(left as u8, right as u8, false));
                    self.byte_codes.push(ByteCode::Jump(1));

                    // Terminate False jump list, go to `SetFalseSkip` bytecode,
      evaluate False
                    self.fix_test_list(false_list);
                    self.byte_codes.push(ByteCode::SetFalseSkip(dst as u8));

                    // Terminate True jump list, go to `LoadBool(true)`
      bytecode, evaluate True
                    self.fix_test_list(true_list);
                    ByteCode::LoadBool(dst as u8, true)
                }
            };
            self.byte_codes.push(code);
```

In comparison, the evaluation of the logical operation `ExpDesc::Test` is simple.

Function

This chapter introduces functions. There are two types of functions in Lua:

- Lua function, defined in Lua;
- External functions are generally implemented in the interpreter language. For example, in the official implementation of Lua, they are C functions; while in our case, they are Rust functions. For example, the `print` function at the beginning of this project was implemented in Rust in the interpreter.

The definition (syntax analysis) and call (virtual machine execution) of the former are both in the Lua language, and the process is complete, so the former will be discussed and implemented first. Then introduce the latter and related API.

Define and Call

Our interpreter only supported sequential execution at first, and later added control structures to support conditional jumps, and blocks also make the scope of variables. Functions, on the other hand, exist more independently in terms of resolution, execution, or scope. To do this, the current framework for parsing and virtual machine execution needs to be modified.

Transform ParseProto

The definition of a function can be nested, that is, the function can be defined again inside another function. If the entire code is regarded as the main function, then our current syntax analysis is equivalent to only supporting this one function. In order to support nested definitions of functions, the parsing needs to be modified. First transform the data structure.

Currently, the context structure of the parsing process is `ParseProto`, and this is also the structure returned to the virtual machine for execution. It is defined as follows:

```
pub struct ParseProto<R: Read> {
    pub constants: Vec<Value>,
    pub byte_codes: Vec<ByteCode>,

    sp: usize,
    locals: Vec<String>,
    break_blocks: Vec<Vec<usize>>,
    continue_blocks: Vec<Vec<(usize, usize)>>,
    gotos: Vec<GotoLabel>,
    labels: Vec<GotoLabel>,
    lex: Lex<R>,
}
```

The specific meaning of each field has been introduced in detail before, and will be ignored here. Here only the fields are distinguished according to the independence of the function:

- the final `lex` field is parsed throughout the code;
- All remaining fields are data inside the function.

In order to support nested definitions of functions, the global part (`lex` field) and the function part (other fields) need to be disassembled. The newly defined data structure `PerFuncProto_` parsed by the function (because we will not adopt this solution in the end, `_` is added to the name of the structure), including other fields left after removing `lex` from the original `ParseProto`:

```
struct PerFuncProto_ {
    pub constants: Vec<Value>,
    pub byte_codes: Vec<ByteCode>,
    sp: usize,
    ... // omit more fields
}
```

In order to support the nesting of functions, it is necessary to support multiple function analysis bodies at the same time. The most intuitive idea is to define a list of function bodies:

```
struct ParseProto<R: Read> {
    funcs: Vec<PerFuncProto_>, // The list of function analysis body
PerFuncProto_ just defined
    lex: Lex<R>, // global data
}
```

Each time a new layer of functions is nested, a new member is pushed into the `funcs` field; it pops up after the function is parsed. The last member of `funcs` represents the current function. This definition is very intuitive, but there is a problem. It is very troublesome to access all the fields of the current function. For example, to access the `constants` field, you need `self.funcs.last().unwrap().constants` to read or `self.funcs.last_mut().unwrap().constants` writes. It's too inconvenient, and the execution efficiency should also be affected.

If it is C language, then this problem is easy to solve: add a pointer member of type `PerFuncProto_` in `ParseProto`, such as `current`, which points to the last member of `funcs`. This pointer is updated every time the function body is pushed or popped. Then we can directly use this pointer to access the current function, such as `self.current.constants`. This approach is very convenient but Rust thinks it is not "safe", because the validity of this pointer cannot be guaranteed at the Rust syntax level. Although there are only two places to update this pointer, which is relatively safe, but since you use Rust, you must follow the rules of Rust.

For Rust, a feasible solution is to add an index (rather than a pointer), such as `icurrent`, pointing to the last member of `funcs`. This index is also updated every time the function body is pushed or popped. When accessing the current function information, we can use `self.funcs[icurrent].constants`. While the Rust language allows this, it's really just a variant of the pointer scheme above, and can still cause bugs due to incorrect updates of the index. For example, if the index exceeds the length of `funcs`, it will panic, and if it is smaller than expected, there will be code logic bugs that are more difficult to debug. In addition, during execution, Rust's list index will be compared with the length of the list, which will also slightly affect performance.

There is also a less intuitive solution that doesn't have the problems above: use recursion. When parsing nested functions, the most natural way is to recursively call the code of the parsing function, then each call will have an independent stack (Rust's call stack), so we can create a function parsing body every time you call it and use it Parse the current Lua function, and return the parsing body for the outer function to process after the call ends. In this solution, only the information of the current function can be accessed during the parsing process, and the information of the outer function cannot be accessed. Naturally, the problem of inconvenient access to the information of the current function just mentioned does not exist. For example, accessing constants still uses `self.constants`, even without modifying existing code. The only thing to solve is the global data `Lex`, which can be passed on as a parameter of the analysis function.

In this solution, there is no need to define a new data structure, just change the `lex` field in the original `ParseProto` from `Lex` type to `&mut Lex`. The syntax analysis function definition for parsing Lua functions is originally the method of `ParseProto`, which is defined as:

```
impl<'a, R: Read> ParseProto<'a, R> {
     fn chunk(&mut self) {
          ...
     }
```

Now change to a normal function, defined as:

```
fn chunk(lex: &mut Lex<impl Read>) -> ParseProto {
     ...
}
```

The parameter `lex` is global data, and each recursive call is directly passed to the next layer. The return value is the parsed information of the current Lua function created inside `chunk()`.

In addition, the `chunk()` function internally calls the `block()` function to parse the code, and the latter returns the end Token of the block. Previously, the `chunk()` function was only used to process the entire code block, so the end Token could only be `Token::Eos`; but now it may also be used to parse other internal functions, and the expected end Token is `Token ::End`. Therefore, the `chunk()` function needs to add a new parameter, indicating the expected end Token. So the definition is changed to:

```
fn chunk(lex: &mut Lex<impl Read>, end_token: Token) -> ParseProto {
     ...
}
```

Add FuncProto

We just modified `ParseProto` and the type of `lex`. Now let's do a small optimization by the way. The first two `pub` modified fields in `ParseProto` are also returned to the virtual machine for execution; most of the latter fields are only used for syntax analysis, which are internal data and do not need to be returned to the virtual machine. These two parts can be disassembled so that only the part needed by the virtual machine is returned. To do this, add the `FuncProto` data structure:

```
// Return information to the virtual machine to execute
pub struct FuncProto {
    pub constants: Vec<Value>,
    pub byte_codes: Vec<ByteCode>,
}

#[derive(Debug)]
struct ParseProto<'a, R: Read> {
    // Return information to the virtual machine to execute
    fp: FuncProto,

    // syntax analysis internal data
    sp: usize,
    locals: Vec<String>,
    break_blocks: Vec<Vec<usize>>,
    continue_blocks: Vec<Vec<(usize, usize)>>,
    gotos: Vec<GotoLabel>,
    labels: Vec<GotoLabel>,
    lex: Lex<R>,

    // global data
    lex: &'a mut Lex<R>,
}
```

So the return value of the `chunk()` function is changed from `ParseProto` to `FuncProto`. Its full definition is as follows:

```
fn chunk(lex: &mut Lex<impl Read>, end_token: Token) -> FuncProto {
    // Generate a new ParseProto to parse the current new Lua function
    let mut proto = ParseProto::new(lex);

    // call block() parsing function
    assert_eq!(proto.block(), end_token);
    if let Some(goto) = proto. gotos. first() {
        panic!("goto {} no destination", &goto.name);
    }

    // only returns the FuncProto part
    proto.fp
}
```

In this way, when syntactically analyzing Lua built-in functions, just recursively call `chunk(self.lex, Token::End)`. The specific syntax analysis is introduced below.

Syntax Analysis

The general process of parsing Lua functions is introduced above, now let's look at the specific syntax analysis. By now, we should be familiar with syntax analysis already, and it can be executed according to BNF. Lua's function definition has 3 places:

1. Global functions;
2. Local functions:
3. An anonymous function is a case of the expression `exp` statement.

The BNF rules are as follows:

```
stat :=
     `function` funcname funcbody | # 1. Global function
     `local` `function` Name funcbody | # 2. Local function
     # omit other cases

exp := functiondef | omit other cases
functiondef := `function` funcbody # 3. Anonymous function

funcbody ::= '(' [parlist] ')' block end # Function definition
```

It can be seen from the above rules that the difference between these three definitions is only at the beginning, and at the end they all belong to `funcbody`. Here only the simplest second case, the local function, is introduced.

```
    fn local_function(&mut self) {
        self.lex.next(); // skip keyword `function`
        let name = self.read_name(); // function name, or local variable
name
        println!("== function: {name}");

        // currently does not support parameters, skip `()`
        self.lex.expect(Token::ParL);
        self.lex.expect(Token::ParR);

        // Call the chunk() parsing function
        let proto = chunk(self.lex, Token::End);

        // Put the parsed result FuncProto into the constant table
        let i = self.add_const(Value::LuaFunction(Rc::new(proto)));
        // load function through LoadConst bytecode
        self.fp.byte_codes.push(ByteCode::LoadConst(self.sp as u8, i as
u16));

        // create local variable
        self. locals. push(name);
    }
```

The parsing process is simple. It should be noted that the processing method of the function prototype FuncProto returned by the `chunk()` function is to put it in the constant table as a constant. It can be compared that a string is a constant composed of a series of character sequences; and the function prototype FuncProto is a constant composed of a series of constant tables and bytecode sequences. It also exists in the constant table, and it is also loaded with `LoadConst` bytecode.

To this end, it is necessary to add a new Value type `LuaFunction` to represent the Rust function, and change the type that originally represented the Lua function from `Function` to `RustFunction`:

```
pub enum Value {
    LongStr(Rc<Vec<u8>>),
    LuaFunction(Rc<FuncProto>),
    RustFunction(fn (&mut ExeState) -> i32),
```

The data type associated with `LuaFunction` is `Rc<FuncProto>`, and it can also be seen from here that it is similar to a string constant.

The syntax analysis of "defining a function" is completed above, and the syntax analysis of "calling a function" is related to functions. But when "calling a function", the Lua function and the Rust function are treated equally, and the Lua programmer does not even know what the function is implemented when calling the function; since the Rust function `print()` has been called before Syntactic analysis, so there is no need to perform syntax analysis specifically for Lua function calls.

Virtual Machine Execution

Like syntax analysis, our previous virtual machine execution part only supports one layer of Lua functions. In order to support function calls, the easiest way is to recursively call the virtual machine to execute, that is, the `execute()` function. code show as below:

```
    ByteCode::Call(func, _) => {
        self. func_index = func as usize;
        match &self. stack[self. func_index] {
            Value::RustFunction(f) => { // previously supported Rust
functions
                f(self);
            }
            Value::LuaFunction(f) => { // new Lua function
                let f = f. clone();
                self.execute(&f); // recursively call the virtual machine!
            }
            f => panic!("invalid function: {f:?}"),
        }
    }
```

However, special handling of the stack is required. During parsing, each time a new function is parsed, the stack pointer (the `sp` field in the `ParseProto` structure) starts from 0. Because during syntax analysis, the absolute starting address of the stack when the virtual machine is executed is not known. Then, when the virtual machine is executing, when accessing the stack, the stack index in the bytecode used needs to add the offset of the stack start address of the current function. For example, for the following Lua code:

```lua
local a, b = 1, 2
local function foo()
    local x, y = 1, 2
end
foo()
```

When parsing the `foo()` function definition, the stack addresses of the local variables x and y are 0 and 1, respectively. When the last line of code is executed and the `foo()` function is called, the function `foo` is placed at the absolute index 2 of the stack. At this time, the absolute indexes of the local variables x and y are 3 and 4. Then when the virtual machine executes, it needs to convert the relative addresses 0 and 1 into 3 and 4.

```
  absolute     relative
  address      address
        +-----+ <---base of main function
      0 |  a  | 0
        +-----+
      1 |  b  | 1
        +-----+
      2 | foo | 2
        +-----+ <---base of foo()
      3 |  x  | 0
        +-----+
      4 |  y  | 1
        +-----+
        |     |
```

When executing the Rust function `print()` before, in order to allow the `print()` function to read the parameters, the `func_index` member is set in `ExeState` to point to the address of the function on the stack. Now call the Lua function, still the same. However, `func_index` is renamed to `base` here, and points to the next address of the function.

```
    ByteCode::Call(func, _) => {
        self.base += func as usize + 1; // Set the absolute address of the
function on the stack
        match &self.stack[self.base-1] {
            Value::RustFunction(f) => {
                f(self);
            }
            Value::LuaFunction(f) => {
                let f = f. clone();
                self. execute(&f);
            }
            f => panic!("invalid function: {f:?}"),
        }
        self.base -= func as usize + 1; // restore
    }
```

All previous write operations to the stack were called `set_stack()` method, now need to add self.base offset:

```
    fn set_stack(&mut self, dst: u8, v: Value) {
        set_vec(&mut self.stack, self.base + dst as usize, v); // plus
self.base
    }
```

All previous read operations on the stack were directly `self.stack[i]`, and now a new function `get_stack()` is also extracted, and the self.base offset is added when accessing the stack:

```
    fn get_stack(&self, dst: u8) -> &Value {
        &self.stack[self.base + dst as usize] // plus self.base
    }
```

So far, we have completed the most basic definition and calling of Lua functions. Thanks to the power of recursion, the code changes are not big. But it's just the beginning of the full feature. The next section adds support for parameters and return values.

Arguments

The previous section introduced the definition and calling process of Lua functions. This section describes the arguments of the function.

The term "argument" has two concepts:

- "parameter", refers to the variable in the function prototype, including information such as parameter name and parameter type;
- "argument", refers to the actual value when the function is called.

When introducing syntax analysis and virtual machine execution later in this section, "parameter" and "argument" must be clearly distinguished sometimes.

A very important point is: in the Lua language, the parameters of the function are local variables! During syntax analysis, the parameters will also be placed in the initial position of the local variable table, so that if there is a reference to the parameter in the subsequent code, it will also be located in the local variable table. In the virtual machine execution phase, the arguments are loaded onto the stack immediately following the function entry, followed by local variables, which is consistent with the order in the local variable table in the syntax analysis phase. For example, for the following functions:

```lua
local function foo(a, b)
     local x, y = 1, 2
end
```

When the `foo()` function is executed, the stack layout is as follows (numbers 0-3 on the right side of the stack are relative indices):

```
|      |
+------+
| foo  |
+======+ <---base
|  a   | 0  \
+------+     + arguments
|  b   | 1  /
+------+
|  x   | 2  \
+------+     + local variables
|  y   | 3  /
+------+
|      |
```

The only difference between arguments and local variables is that the value of the parameter is passed in by the caller when calling, while the local variable is assigned inside the function.

Syntax Analysis of Parameter

The syntax analysis of the parameter is also the syntax analysis of the function definition. When the function definition was introduced in the previous section, the parameter part was omitted in the process of syntax analysis, and it is added now. The BNF of the function definition is funcbody, which is defined as follows:

```
funcbody ::= `(` [parlist] `)` block end
parlist ::= namelist [`,` `...`] | `...`
namelist ::= Name {`,` Name}
```

As you can see, the parameter list consists of two optional parts:

- Multiple optional Names are fixed parameters. In the previous section, when parsing the new function and creating the `FuncProto` structure, the `locals` field of the local variable table was initialized to an empty list. Now initialize to a parameter list instead. In this way, the parameters are at the front of the local variable table, and the subsequent newly created local variables follow, which is consistent with the stack layout diagram at the beginning of this section. In addition, since the number of arguments of the calling function in Lua language is allowed to be different from the number of parameters. If it is more, it will be discarded, and if it is less, it will be filled with nil. Therefore, the number of parameters should also be added to the result of `FuncProto` for comparison during virtual machine execution.

- The last optional `...` indicates that this function supports variable arguments. If it is supported, then in the subsequent syntax analysis, `...` can be used in the body of the function to refer to variable parameters, and in the virtual machine execution stage, special processing should also be done for variable parameters. Therefore, a flag needs to be added in `FuncProto` to indicate whether this function supports variable parameters.

In summary, there are three modification points in total. Add two fields to `FuncProto`:

```rust
pub struct FuncProto {
    // Whether to support variable parameters.
    // Used in both parsing and virtual machine execution.
    pub has_varargs: bool,

    // The number of fixed parameters.
    // Used in virtual machine execution.
    pub nparam: usize,

    pub constants: Vec<Value>,
    pub byte_codes: Vec<ByteCode>,
}
```

In addition, when initializing the `ParseProto` structure, use the parameter list to initialize the local variable `locals` field. code show as below:

```
impl<'a, R: Read> ParseProto<'a, R> {
    // Add has_varargs and params two parameters
    fn new(lex: &'a mut Lex<R>, has_varargs: bool, params: Vec<String>) ->
Self {
        ParseProto {
            fp: FuncProto {
                has_varargs: has_varargs, // Whether to support variable
parameters
                nparam: params.len(), // number of parameters
                constants: Vec::new(),
                byte_codes: Vec::new(),
            },
            sp: 0,
            locals: params, // Initialize the locals field with the
parameter list
            break_blocks: Vec::new(),
            continue_blocks: Vec::new(),
            gotos: Vec::new(),
            labels: Vec::new(),
            lex: lex,
        }
    }
```

At this point, the syntax analysis of the parameters is completed. It involves variable parameters, virtual machine execution and other parts, which will be described in detail below.

Syntax Analysis of Arguments

Syntactic analysis of arguments, that is, syntactic analysis of function calls. This has been implemented in the previous chapter when prefixexp was implemented: the parameter list is read through the `explist()` function, and loaded to the position behind the function entry on the stack in turn. Consistent with the stack layout diagram at the beginning of this section, it is equivalent to assigning values to parameters. The actual number of arguments is parsed here and written into the arguments of the bytecode `Call` for comparison with the formal during the execution phase of the virtual machine.

But the implementation at the time was incomplete and did not support variable parameters. More details later in this section.

Virtual Machine Execution

In the above syntax analysis of the arguments, the arguments have been loaded onto the stack, which is equivalent to assigning values to the parameters, so when the virtual machine executes the function call, it does not need to process the parameters. However, in the Lua language, the number of arguments may not be equal to the number of parameters when a function is called. If there are more arguments than parameters, there is no need to deal with it, and the extra part is considered to be a temporary variable that occupies the stack position but is useless; but if the argument is less than the parameter, then the insufficient part needs to be set to nil, otherwise the subsequent words The reference to this parameter by the section code will cause Lua's stack access exception. In addition, the execution of `Call` bytecode does not require other processing of parameters.

As mentioned above in the grammatical analysis, the number of parameters and arguments are respectively in the `nparam` field in the `FuncProto` structure and the associated parameters of `Call` bytecode. So the virtual machine execution code of the function call is as follows:

```
     ByteCode::Call(func, narg) => { // `narg` is the actual number of
arguments passed in
         self.base += func as usize + 1;
         match &self.stack[self.base - 1] {
             Value::LuaFunction(f) => {
                 let narg = narg as usize;
                 let f = f. clone();
                 if narg < f.nparam { // `f.nparam` is the number of
parameters in the function definition
                     self.fill_stack(narg, f.nparam - narg); // fill nil
                 }
                 self. execute(&f);
             }
```

So far, the fixed parameter part is completed, which is relatively simple; the variable parameter part is introduced below, and it becomes complicated.

Variable Parameters

In Lua, the `...` expression is used for variable parameters and variable arguments both. It's variable parameters in parameter list in function definition; and it's variable arguments otherwise.

Variable parameters have been mentioned in Syntax Analysis of parameters above, and their functions are relatively simple, which just indicates that this function does support variable parameters. The rest of this section mainly introduces the processing of variable arguments when executing a function call.

At the beginning of this section, the parameters of the function are introduced as local variables, and the layout of the stack is drawn. However, this statement is only suitable for fixed arguments, but not for variable parameters. Add variable parameters to the previous `foo()` function as an example, the code is as follows:

```lua
local function foo(a, b, ...)
    local x, y = 1, 2
    print(x, y, ...)
end
foo(1, 2, 3, 4, 5)
```

What should the stack layout look like after adding variable parameters? In other words, where does the variable argument exist? When the last line `foo()` in the above code is called, `1` and `2` correspond to the parameters `a` and `b` respectively, and `3` , `4` and `5` are variable Arguments. Before the call starts, the stack layout is as follows:

```
|     |
+-----+
| foo |
+=====+ <-- base
|  1  |  \
+-----+   + Fixed arguments, corresponding to `a` and `b`
|  2  |  /
+-----+
|  3  |  \
+-----+   |
|  4  |   + Variable arguments, corresponding to `...`
+-----+   |
|  5  |  /
+-----+
|     |
```

After entering the `foo()` function, where should the next three arguments exist? The most direct idea is to keep the above layout unchanged, that is, the variable arguments are stored behind the fixed arguments. However, this is not acceptable! Because this will occupy the space of local variables, that is, `x` and `y` in the example will be moved back, and the distance moved back is the number of variable arguments. However, the number of variable arguments cannot be determined during the syntax analysis stage, so the position of the local variable on the stack cannot be determined, and the local variable cannot be accessed.

The official implementation of Lua is to ignore the variable parameters in the syntax analysis stage, so that the local variables are still behind the fixed parameters. But when the virtual machine is executing, after entering the function, the variable parameters are moved to the front of the function entry, and the number of variable arguments is recorded. In this way, when accessing variable parameters, the stack position can be located according to the function entry position and the number of variable arguments, that is, `stack[self.base - 1 - number of arguments.. self.base - 1]` . The following is a stack layout diagram:

```
|     |
+-----+
|  3  | -4 \
+-----+    |                                  num_varargs: usize  // record
the #variable arguments
|  4  | -3  + move the variable arguments           +-----+
+-----+     | to the front of the function entry    |  3  |
|  5  | -2 /                                        +-----+
+-----+
| foo | <-- function entry
+=====+ <-- base
| a=1 | 0  \
+-----+     + fixed arguments, corresponding to `a` and `b`
| b=2 | 1  /
+-----+
|  x  | 2  \
+-----+     + local variables
|  y  | 3  /  following to fixed arguments
```

Since this solution needs to record additional information (the number of variable arguments) when the virtual machine is executed, and also move the parameters on the stack, it is easier to record the variable arguments directly:

```
|     |
+-----+
| foo | <-- function entry                varargs: Vec<Value>  // record
variable arguments
+=====+                                      +-----+-----+-----+
| a=1 | 0  \                                 |  3  |  4  |  5  |
+-----+     + fixed arguments                +-----+-----+-----+
| b=2 | 1  /
+-----+
|  x  | 2  \
+-----+     + local variables
|  y  | 3  /
```

Compared with the official implementation of Lua, this method does not use the stack, but uses `Vec`, which will have additional memory allocation on the heap. But more intuitive and clear.

After determining the storage method of the variable arguments, we can perform syntax analysis and virtual machine execution.

ExpDesc::VarArgs and Application Scenarios

The above is about passing variable parameters when the function is called, and then introduces how to access variable parameters in the function body.

Access to a variable argument is an independent expression, the syntax is `...`, parsed in the `exp_limit()` function, and a new expression type `ExpDesc::VarArgs` is added, this type is not associated parameter.

It is very simple to read this expression, first check whether the current function supports variable parameters (whether there is `...` in the function prototype), and then return `ExpDesc::VarArgs`. The specific code is as follows:

```
    fn exp_limit(&mut self, limit: i32) -> ExpDesc {
        let mut desc = match self. lex. next() {
            Token::Dots => {
                if !self.fp.has_varargs { // Check if the current function
 supports variable parameters?
                    panic!("no varargs");
                }
                ExpDesc::VarArgs // New expression type
            }
```

But what to do with `ExpDesc::VarArgs` read? This requires first sorting out the three scenarios of using variable arguments:

1. When `...` is used as the last argument of a function call, the last value of a return statement, or the last list member of a table construction, it represents all the arguments passed in. For example, the following example:

   ```
   print("hello: ", ...) -- last argument
   local t = {1, 2, ...} -- last list member
   return a+b, ... -- the last return value
   ```

2. When `...` is used as the last expression after the equal sign `=` of a local variable definition statement or an assignment statement, the number will be expanded or reduced as required. For example, the following example:

   ```
   local x, y = ... -- Take the first 2 arguments and assign them to x and
   y respectively
   t.k, t.j = a, ... -- Take the first argument and assign it to t.j
   ```

3. Other places only represent the first actual argument passed in. For example, the following example:

```
local x, y = ..., b -- not the last expression, only take the first
argument and assign it to x
t.k, t.j = ..., b -- not the last expression, only take the first
argument and assign it to t.k
if ... then -- conditional judgment
    t[...] = ... + f -- table index, and operands of binary operations
end
```

Among them, the first scenario is the most basic, but it is also the most complicated to implement; the latter two scenarios are special cases and relatively simple to implement. The three scenarios are analyzed in turn below.

Scenario 1: All Variable Arguments

The first scenario is introduced first, that is, loading all variable arguments. The 3 statements in this scenario are as follows:

1. The last argument of the function call, is to use the variable arguments of the *current* function as the variable arguments of the *calling* function. Here it involves variable arguments in 2 functions, which is a bit confusing and inconvenient to describe;
2. The last value of the return statement, but the return value is not supported yet, which will be introduced in the next section;
3. The last list member of the table construction.

The implementation ideas of these three statements are similar. When parsing the expression list, only the previous expression is discharged, and the last expression is not discharged; and then after the complete statement is parsed, it is checked separately whether the last statement is `ExpDesc::VarArgs`:

- If not, discharge normally. In this case, the quantity of all expressions can be determined during parsing, and the number of values can be encoded into the corresponding bytecode.
- If yes (`ExpDesc::VarArgs`), use the newly added bytecode `VarArgs` to load all variable parameters, and the number of arguments is not known during syntax analysis, but can only be known when the virtual machine is executed, so the total number of expressions can't be encoded into the corresponding bytecode, so it needs to be handled with a special value or a new bytecode.

Among the three statements, the third statement table structure is relatively the simplest, so we introduce it first.

The syntax analysis process of the previous table construction is: in the process of reading all members in a loop, if an array member is parsed, it will be immediately discharged to the stack; after the loop reading is completed, all array members are loaded on the stack in turn, and then generated `SetList` bytecode adds it to the list. The second associated parameter of this `SetList` bytecode is the number of array members. For simplicity, the processing of batch loading when there are more than 50 members is ignored here.

Now modify the process: in order to process the last expression alone, when parsing to an array member, we need delay the discharge. The specific method is relatively simple but not easy to describe, you can refer to the following code. The code is excerpted from the `table_constructor()` function, and only the content related to this section is kept.

```
        // Add this variable to save the last read array member
        let mut last_array_entry = None;

        // Loop to read all members
        loop {
            let entry = // omit the code to read members
            match entry {
                TableEntry::Map((op, opk, key)) => // omit the code of the
 member part of the dictionary
                TableEntry::Array(desc) => {
                    // Use the replace() function to replace the last read
 member with the new member desc
                    // and discharge. And the new member, the current "last
 member", is
                    // Store in last_array_entry.
                    if let Some(last) = last_array_entry. replace(desc) {
                        self.discharge(sp0, last);
                    }
                }
            }
        }

        // process the last expression, if any
        if let Some(last) = last_array_entry {
            let num = if self. discharge_expand(last) {
                // variable arguments. It is impossible to know the specific
 number
                // of arguments in the syntax analysis stage, so `0` is used to
                // represent all arguments on the stack.
                0
            } else {
                // not variable arguments, so we can calculate the total number
 of members
                (self.sp - (table + 1)) as u8
            };
            self.fp.byte_codes.push(ByteCode::SetList(table as u8, num));
        }
```

The above code sorting process is relatively simple, so I won't introduce them line by line here. There are a few details to cover when dealing with the last expression:

- Added `discharge_expand()` method for special handling of `ExpDesc::VarArgs` type expressions. It is foreseeable that this function will be used by the other two statements (return statement and function call statement) later. Its code is as follows:

```rust
fn discharge_expand(&mut self, desc: ExpDesc) -> bool {
    match desc {
        ExpDesc::VarArgs => {
            self.fp.byte_codes.push(ByteCode::VarArgs(self.sp as u8));
            true
        }
        _ => {
            self.discharge(self.sp, desc);
            false
        }
    }
}
```

- If the last expression is a variable parameter, then the second associated parameter of `SetList` bytecode is set to `0`. Previously (when variable arguements expressions were not supported), this parameter of `SetList` bytecode could not be 0, because if there is no array member, then it is sufficient not to generate `SetList` bytecode, and there is no need to generate an association `SetList` with parameter 0. So here we can use `0` as a special value. In contrast, the other two statements in this scenario (return statement and function call statement) originally support 0 expressions, that is, there is no return value and no parameters, so `0` cannot be used as a special value. Then think of other ways.

  Of course, the special value `0` may not be used here, but a new bytecode, such as `SetListAll`, is specially used to deal with this situation. These two approaches are similar, we still choose to use the special value `0`.

- When the virtual machine is executing, if the second associated parameter of `SetList` is `0`, all the values behind the table on the stack will be fetched. That is, from the position of the table to the top of the stack, they are all expressions used for initialization. The specific code is as follows, adding the judgment of `0`:

```
    ByteCode::SetList(table, n) => {
        let ivalue = self.base + table as usize + 1;
        if let Value::Table(table) = self.get_stack(table). clone() {
            let end = if n == 0 { // 0, variable arguments, means all
expressions up to the top of stack
                self.stack.len()
            } else {
                ivalue + n as usize
            };
            let values = self.stack.drain(ivalue .. end);
            table.borrow_mut().array.extend(values);
        } else {
            panic!("not table");
        }
    }
```

- Since the actual number of expressions can be obtained according to the top of the stack when the virtual machine is executed in the case of variable arguments, then can we do this in the case of fixed expressions before? In this way, is the second parameter associated with `SetList` useless? The answer is no, because there may be temporary variables on the stack! For example the following code:

```
t = { g1+g2 }
```

The two operands of the expression `g1+g2` are global variables. Before evaluating the entire expression, they must be loaded on the stack separately, and two temporary variables need to be occupied. The stack layout is as follows:

```
|        |
+--------+
|   t    |
+--------+
| g1+g2  | load the global variable g1 into here temporaryly, and covered by
g1+g2 later
+--------+
|   g2   | load the global variable g2 into here temporaryly
+--------+
|        |
```

At this time, the top of the stack is g2. If the method of going from the back of the list to the top of the stack is also followed, then g2 will also be considered a member of the list. Therefore, for the previous case (fixed number of expressions), it is still necessary to determine the number of expressions in the syntax analysis stage.

- Then, why can the top of the stackto determine the number of expressions in the case of variable arguments? This requires the virtual machine to clean up temporary variables when executing bytecodes that load variable parameters. this point is very important. The specific code is as follows:

```
        ByteCode::VarArgs(dst) => {
            self.stack.truncate(self.base + dst as usize); // Clean up temporary
 variables! ! !
            self.stack.extend_from_slice(&varargs); // load variable parameters
        }
```

So far, the processing of the variable arguments as the last expression of the table construction is completed. There are not many related codes, but it is not easy to sort out the ideas and some details.

Scenario 1: All Variable Arguments (continued)

The table construction statement in the first scenario was introduced above, and now we introduce the case where variable arguments are used as the last parameter of a function call. Just listening to this description is confusing. These two statements handle variable arguments in the same way, and only the differences are introduced here.

It has been explained in the introduction of Syntax Analysis of arguments above that all arguments are loaded to the top of the stack sequentially through the `explist()` function, and the number of arguments is written to `Call` bytecode. But the implementation at the time did not support variable arguments. Now in order to support variable arguments, the last expression needs to be treated specially. To do this, we modify the `explist()` function to keep and return the last expression, but just discharge the previous expressions onto the stack in turn. The specific code is relatively simple, skip it here. To review, in the assignment statement, when reading the expression list on the right side of the equal sign `=`, we also need to keep the last expression not discharged. After modifying the `exp_list()` function this time, it can also be used in the assignment statement.

After modifying the `explist()` function, combined with the above introduction to the table construction statement, the variable arguments in the function call can be realized. code show as below:

```
    fn args(&mut self) -> ExpDesc {
        let ifunc = self.sp - 1;
        let narg = match self. lex. next() {
            Token::ParL => { // parameter list wrapped in brackets ()
                if self.lex.peek() != &Token::ParR {
                    // Read the argument list. Keep and return the last
expression
                    // `last_exp`, and discharge the previous expressions
onto the
                    // stack in turn and return their number `nexp`.
                    let (nexp, last_exp) = self.explist();
                    self.lex.expect(Token::ParR);

                    if self. discharge_expand(last_exp) {
                        // Variable arguments !!!
                        // Generate the newly added `VarArgs` bytecode
                        // and read all variable arguments.
                        none
                    } else {
                        // Fixed arguments. `last_exp` is also discharged
onto the stack as the last argument.
                        Some(nexp + 1)
                    }
                } else { // no parameters
                    self. lex. next();
                    some(0)
                }
            }
            Token::CurlyL => { // table construction without parentheses
                self. table_constructor();
                some(1)
            }
            Token::String(s) => { // string constant without parentheses
                self.discharge(ifunc+1, ExpDesc::String(s));
                some(1)
            }
            t => panic!("invalid args {t:?}"),
        };

        // For `n` fixed arguments, convert to `n+1`;
        // Converts to `0` for variable arguments.
        let narg_plus = if let Some(n) = narg { n + 1 } else { 0 };

        ExpDesc::Call(ifunc, narg_plus)
    }
```

The difference from the table construction statement introduced before is that the bytecode corresponding to the table construction statement is `SetList`, and in the case of fixed members, the associated parameter used to represent the quantity will not be `0`; so we can use `0` as a special value to represent a variable number of members. However, for the function call statement, it supports the case of no argument, that is to say, the parameter of the argument value associated with the bytecode `Call` may already be `0`, so it is not possible to simply put `0` as a special value. Then, there are 2 options:

- Choose another special value, such as `u8::MAX`, that is, 255 as a special value;
- Still use `0` as a special value, but in the case of a fixed argument, add 1 to the parameter. For example, if there are 5 arguments, then write 6 in the `Call` bytecode; if N bytecodes, write N+1; in this way, you can ensure that in the case of a fixed parameter, this parameter must be greater than 0.

I feel that the first solution is slightly better, it's clearer and less error-prone. But the official implementation of Lua uses the second solution. We also use the second option. Corresponding to the two variables in the above code:

- `narg: Option<usize>` indicates the actual number of arguments, `None` indicates variable arguments, `Some(n)` indicates that there are `n` fixed arguments;
- `narg_plus: usize` is the corrected value to be written into `Call` bytecode.

The same thing as the table construction statement introduced before is that since the special value `0` is used to represent the variable parameter, then when the virtual machine executes, there must be a way to know the actual number of arguments. The number of arguments can only be calculated by the distance between the pointer on the top of the stack and the function entry, so it is necessary to ensure that the top of the stack is all arguments and there are no temporary variables. For this requirement, there are two cases:

- The argument is also a variable arguments `...`, that is, the last argument is `VarArgs`, for example, the call statement is `foo(1, 2, ...)`, then since the virtual machine execution of `VarArgs` introduced before will ensure clean up temporary variables, so there is no need to clean up again in this case;
- The argument is fixed arguments. For example, if the calling statement is `foo(g1+g2)`, then it is necessary to clean up the possible temporary variables.

Correspondingly, the function call in the virtual machine execution phase, that is, the execution of `Call` bytecode, needs to be modified as follows:

- Modify the associated parameter narg_plus;
- Clean up possible temporary variables on the stack when needed.

code show as below:

```
        ByteCode::Call(func, narg_plus) => { // `narg_plus` is the corrected
number of real parameters
            self.base += func as usize + 1;
            match &self.stack[self.base - 1] {
                Value::LuaFunction(f) => {
                    let narg = if narg_plus == 0 {
                        // Variable arguments. As mentioned above, the execution
                        // of VarArgs bytecode will clean up possible temporary
                        // variable, so the top of the stack can be used to
determine
                        // the actual number of arguments.
                        self.stack.len() - self.base
                    } else {
                        // Fixed arguments. Need to subtract 1 for correction.
                        narg_plus as usize - 1
                    };

                    if narg < f.nparam { // fill nil, original logic
                        self.fill_stack(narg, f.nparam - narg);
                    } else if f.has_varargs && narg_plus != 0 {
                        // If the called function supports variable arguments,
and the
                        // call is a fixed argument, then we need to clean up
possible
                        // temporary variables on the stack
                        self.stack.truncate(self.base + narg);
                    }

                    self. execute(&f);
                }
```

So far, we have completed the part of the first scenario of variable arguments. This part is the most basic and also the most complex. Two other scenarios are described below.

Scenario 2: The first N Variable Arguments

Now introduce the second scenario of variable arguments, which requires a fixed number of variable arguments. The number of paraargumentsmeters to be used in this scenario is fixed and can be compiled into bytecode, which is much simpler than the previous scenario.

This scenario includes 2 statements: a local variable definition statement and an assignment statement. When the variable arguments are used as the last expression after the equal sign `=`, the number will be expanded or reduced as required. For example, the following sample code:

```
    local x, y = ... -- Take the first 2 arguments and assign them to x and
y respectively
    t.k, t.j = a, ... -- Take the first argument and assign it to t.j
```

The processing of these two statements is basically the same. Only the first local variable definition statement is introduced here.

The processing flow of the previous statement is to first load the expressions on the right side of `=` onto the stack in order to complete the assignment of local variables. If the number of expressions on the right side of `=` is less than the number of local variables on the left, then generate `LoadNil` bytecode to assign values to the extra local variables; if it is not less than, no processing is required.

Now special treatment is required for the last expression: if the number of expressions is less than the number of local variables, and the last expression is a variable arguments `...`, then the arguments is read as needed; if it is not variable arguments, it still falls back to the original method, which is filled with `LoadNil`. The `explist()` function that was modified just now comes in handy again, the specific code is as follows:

```
    let want = vars.len();

    // Read the list of expressions.
    // Keep and return the last expression `last_exp`, and discharge the
previous
    // the expressions onto the stack in turn and return their number
`nexp`.
    let (nexp, last_exp) = self.explist();
    match (nexp + 1).cmp(&want) {
        Ordering::Equal => {
            // If the expression is consistent with the number of local
variables,
            // the last expression is also dischargeed on the stack.
            self.discharge(self.sp, last_exp);
        }
        Ordering::Less => {
            // If the expressions are less than the number of local
variables,
            // we need to try to treat the last expression specially! ! !
            self.discharge_expand_want(last_exp, want - nexp);
        }
        Ordering::Greater => {
            // If the expression is more than the number of local variables,
            // adjust the top pointer of the stack; the last expression
            // is no need to deal with it.
            self.sp -= nexp - want;
        }
    }
```

In the above code, the added logic is `discharge_expand_want()` function, which is used to load `want - nexp` expressions onto the stack. code show as below:

```
    fn discharge_expand_want(&mut self, desc: ExpDesc, want: usize) {
        debug_assert!(want > 1);
        let code = match desc {
            ExpDesc::VarArgs => {
                // variadic expression
                ByteCode::VarArgs(self.sp as u8, want as u8)
            }
            _ => {
                // For other types of expressions, still use the previous
 method, that is, use LoadNil to fill
                self.discharge(self.sp, desc);
                ByteCode::LoadNil(self.sp as u8, want as u8 - 1)
            }
        };
        self.fp.byte_codes.push(code);
    }
```

This function is similar to the `discharge_expand()` function in the first scenario above, but there are two differences:

- Previously, *all* variable arguments in the actual execution were required, but this function has a certain number of requirements, so there is an additional parameter `want`;
- The previous function needs to return whether it is a variable arguments, so that the caller can make a distinction; but this function has no return value because the requirement is clear and the caller does not need to make a distinction.

Compared with the first scenario above, another important change is that `VarArgs` bytecode adds an associated parameter to indicate how many arguments need to be loaded onto the stack. Because in this scenario, this parameter is definitely not less than 2, and in the next scenario, this parameter is fixed at 1, and 0 is not used, so 0 can be used as a special value to represent the value in the first scenario above: *all* arguments at execution time.

The virtual machine execution code of this bytecode is also changed as follows:

```
        ByteCode::VarArgs(dst, want) => {
            self.stack.truncate(self.base + dst as usize);

            let len = varargs.len(); // actual number of arguments
            let want = want as usize; // need the number of arguments
            if want == 0 { // All arguments are required, and the process
 remains unchanged
                self.stack.extend_from_slice(&varargs);
            } else if want > len {
                // Need more than actual, fill `nil` with fill_stack()
                self.stack.extend_from_slice(&varargs);
                self.fill_stack(dst as usize + len, want - len);
            } else {
                // needs as much or less than actual
                self.stack.extend_from_slice(&varargs[..want]);
            }
        }
```

So far, the second scenario of variable parameters has been completed.

Scenario 3: Only Take the First Variable Argument

The two scenarios introduced above are in a specific statement context, and the variable arguments are loaded onto the stack through the `discharge_expand_want()` or `discharge_expand()` function respectively. And the 3rd scenario is everywhere except the above specific statement context. So from this perspective, the third scene can be regarded as a general scene, so a general loading method must be used. Before the variable arguments expression is introduced in this section, all other expressions are loaded onto the stack by calling the `discharge()` function, which can be regarded as a general loading method. So in this scenario, the variable arguments expression should also be loaded through the `discharge()` function.

In fact, this scenario has already been encountered above. For example, in the second scenario above, if the number of expressions on the right side of `=` is equal to the number of local variables, the last expression is processed by the `discharge()` function:

```
        let (nexp, last_exp) = self.explist();
        match (nexp + 1).cmp(&want) {
            Ordering::Equal => {
                // If the expression is consistent with the number of
                // local variables, the last expression is also normal
                // discharged on the stack.
                self.discharge(self.sp, last_exp);
            }
```

Here the last expression of `discharge()` may also be a variable arguments expression `...`, then it is the current scene.

For another example, the `explist()` function is called in the above two scenarios to process the expression list. Except for the last expression, all previous expressions are loaded onto the stack by this function by calling `discharge()`. If there is a variable arguments expression `...` in the previous expression, such as `foo(a, ..., b)`, then it is also the current scene.

In addition, the above also lists examples of variable expressions in other statements, all of which belong to the current scene.

Since this scene belongs to a general scene, there is no need to make any changes in the syntax analysis stage, but only need to complete the processing of the variable expression `ExpDesc::VarArgs` in the `discharge()` function. This process is also very simple, just use the `VarArgs` bytecode introduced above, and only load the first argument to the stack:

```
    fn discharge(&mut self, dst: usize, desc: ExpDesc) {
        let code = match desc {
            ExpDesc::VarArgs => ByteCode::VarArgs(dst as u8, 1), // 1 means
 only load the first argument
```

This completes the third scenario.

At this point, all the scenarios of variable arguments are finally introduced.

Summary

This section begins by introducing the mechanism of parameters and arguments respectively. For parameters, syntax analysis puts the parameters in the local variable table and uses them as local variables. For arguments, the caller loads the parameters onto the stack, which is equivalent to assigning values to the parameters.

Most of the following pages introduce the processing of variable arguments, including three scenarios: all arguments, fixed number of arguments, and the first argument in general scenarios.

Return Value

This section describes the return values of Lua functions. First introduce the case of fixed number return values, and then introduce the case of variable number.

Similar to the parameter characteristics in the previous section involving parameters and arguments, there are two places involved in realizing the return value of a function:

- The called function generates a return value before exiting. This is done with the `return` statement in Lua. Correspondingly, the `Return` bytecode needs to be added.
- The caller reads and processes the return value. This part of the functionality is implemented in `Call` bytecode. Previous `Call` bytecode just called the function without processing the return value.

Just as the arguments are passed by the stack, the return values are also passed by the stack.

We first introduce the `return` statement and `Return` bytecode that the function generates the return value.

`Return` Bytecode

Between the called function and the caller, the return values are passed using the stack. The called function generate return values and loads them on the stack, and then notifies the caller of the return values' positions on the stack, and the caller reads the return values from the stack.

Functions in Lua language support multiple return values. If the positions of these return values on the stack are discontinuous, it is difficult to inform the caller of the specific return value. Therefore, all return values are required to be arranged continuously on the stack, so that the caller can be informed by the starting index on the stack and the number of return values. To do this, all return values need to be loaded onto the top of the stack in turn. Like the following example:

```lua
local function foo()
    local x, y = 1, 2
    return x, "Yes", g1+g2
end
```

The stack layout before the function returns is as follows:

```
|        |
+--------+
|  foo   | The caller loads `foo` onto the stack
+========+ <--base
|   x    | 0 \
+--------+    + local variables
|   y    | 1 /
+--------+
|   x    | 2 \
+--------+    |
| "yes"  | 3   + return value
+--------+    |
| g1+g2  | 4 /
+--------+
|  g2    | 5<-- temporary variables
+--------+
|        |
```

The numbers 0~5 on the right side of the stack are relative addresses. Among them, 2~4 is the position of the return values on the stack, then the information to be returned by this function is `(2, 3)`, where `2` is the starting position of the return value on the stack, and `3` is the return values number. It can be seen that the newly added bytecode `Return` needs to be associated with 2 parameters.

In addition to the above-mentioned general cases, there are two special cases, that is, the cases where the number of return values is 0 and 1.

First of all, for the case where the number of return values is 0, that is, the return statement with no return value, although the `Return` bytecode can also be used to return `(0, 0)`, but for clarity, we add bytecode `Return0` without associate parameter for this case.

Secondly, for the case where the number of return values is 1, it can be optimized during syntax analysis. In the case of the above multiple return values, it is mandatory to load all the return values onto the stack sequentially for the sake of continuity and to be able to notify the caller of the position of the return value. And if there is only one return value, continuity is not required, so for local variables that are already on the stack, there is no need to load them on the stack again. Of course, other types of return values (such as global variables, constants, table indexes, etc.) still need to be loaded. Like the following example:

```lua
local function foo()
    local x, y = 1, 2
    return x
end
```

The stack layout before the function returns is as follows:

```
|        |
+--------+
|  foo   | The caller loads foo onto the stack
+========+ <--base
|   x    | 0 \    <-----return values
+--------+    + local variables
|   y    | 1 /
+--------+
|        |
```

There is only one return value `x`, and it is a local variable, which is already on the stack, and it is enough to return `(0, 1)`, without loading it to the top of the stack again.

In summary, the newly added two bytecodes are defined as follows:

```
pub enum ByteCode {
     Return0,
     Return(u8, u8),
```

The parsing process of the `return` statement is as follows:

- for no return value, generate `Return0` bytecode;
- For a single return value, *on-demand* loaded onto the stack and generate `Return(?, 1)` bytecode;
- For multiple return values, *force* to be loaded onto the stack sequentially and generate `Return(?, ?)` bytecode.

Syntax Analysis of `return` statement

The parsing process of the `return` statement is summarized above, and now the syntax analysis begins. The BNF definition of the `return` statement is as follows:

```
retstat ::= return [explist][';']
```

In addition to optional multiple return value expressions, there can be 1 optional `;`. In addition, there is another rule, that is, the end token of a block must be followed by the return statement, such as `end`, `else`, etc. This statement is relatively simple, but there are more details. The code is first listed below:

```
        fn ret_stat(&mut self) {
            let code = match self. lex. peek() {
                // return;
                Token::SemiColon => {
                    self. lex. next();
                    ByteCode::Return0 // no return value
                }

                // return
                t if is_block_end(t) => {
                    ByteCode::Return0 // no return value
                }

                _ => { // has return values
                    let mut iret = self.sp;

                    // Read the list of expressions. Only the last one is kept
and ExpDesc
                    // is returned, while the previous ones are loaded onto the
stack.
                    // Return value: `nexp` is the number of previously loaded
expressions,
                    // and `last_exp` is the last expression.
                    let (nexp, last_exp) = self.explist();

                    // check optional ';'
                    if self.lex.peek() == &Token::SemiColon {
                        self. lex. next();
                    }
                    // check block end
                    if !is_block_end(self.lex.peek()) {
                        panic!("'end' expected");
                    }

                    if nexp == 0 {
                        // single return value, loaded *on demand*
                        iret = self.discharge_any(last_exp);
                    } else {
                        // Multiple return values, other return values have been
loaded to
                        // the top of the stack in turn, now we need to put the
last
                        // Expressions are also *forced* to be loaded on top of
the stack,
                        // after other return values
                        self.discharge(self.sp, last_exp);
                    }

                    ByteCode::Return(iret as u8, nexp as u8 + 1)
                }
            };
            self.fp.byte_codes.push(code);
        }
```

Because the processing of single and multiple return values is different, when reading the return value list, keep the last expression and not directly load it on the stack. At this point, the modified `explist()` function in the previous section comes in handy again. If there is only the last expression, that is, `nexp == 0`, then it is a single expression, and it is loaded on the stack as needed; otherwise, it is the case of multiple return values, and other return values have been loaded to the stack in turn At the top, it is necessary to force the last expression to be loaded on the top of the stack, behind other return values.

To review, in the above code, the `discharge_any()` method in the case of a single return value is *on-demand* loading, that is, it does not process expressions already on the stack (local variables or temporary variables, etc.); and The `discharge()` method in the case of multiple return values is *forced* to load.

Return Bytecode Execution

After completing the syntax analysis, the next step is to introduce the execution of the `Return` bytecode by the virtual machine. Two things need to be done:

- To exit from the execution of the current function `execute()`, use the `return` statement of Rust;
- The most intuitive way to tell the caller the position of the return value is to return the two parameters associated with `Return` bytecode: the starting position and number of return values on the stack. However, the starting position here needs to be converted from a relative position to an absolute position. code show as below:

  ```
  ByteCode::Return(iret, nret) => {
      return (self. base + iret as usize, nret as usize);
  }
  ```

This is a bit long-winded, and there are 2 problems:

- The prototype of the Rust function type in Lua (such as the `print` function) is `fn (&mut ExeState) -> i32`, and there is only 1 return value `i32`, which represents the number of Rust function return values. If the Lua function type returns 2 values, the return information of these two types of functions is inconsistent, which is inconvenient to handle later.
- Later in this section, a variable number of return values of Lua functions will be supported, and the specific number of return values needs to be calculated according to the execution situation.

So it is also changed here to only return the number of Lua function return values, but not returning the starting position. For this reason, possible temporary variables on the stack need to be cleaned up to ensure that the return value is at the top of the stack. In this way, the caller can determine the position of the return value only according to the number of return values. Also using the above example:

```
 |        |
 +--------+
 |  foo   | The caller loads foo onto the stack
 +========+ <--base
 |   x    | 0 \
 +--------+    + local variables
 |   y    | 1 /
 +--------+
 |   x    | 2 \
 +--------+    |
 | "yes"  | 3   + return values
 +--------+    |
 | g1+g2  | 4 /
 +--------+
 |        | <--clean up the temporary variable `g2`
```

In this example, after clearing the temporary variable `g2` at the top of the stack, only return `3` to the calling function, and the calling function can read the 3 values at the top of the stack as the return value.

So why do we need to associate 2 parameters in the `Return` bytecode? In addition to the number of return values, but also the starting position of the return value? This is because it is difficult to determine whether there are temporary variables on the top of the stack during execution during the syntax analysis phase (such as `g2` in the above example), and even if it can be determined, there is nothing to do with these temporary variables (unless a bytecode is added to clean up the temporary variables). Therefore, the return value cannot be expressed only by the number. In the virtual machine execution stage, since possible temporary variables can be cleaned up, there is no need to return to the starting address without the interference of temporary variables.

In summary, the execution code of `Return` bytecode is as follows:

```
        ByteCode::Return(iret, nret) => {
            // convert relative address to absolute address
            let iret = self.base + iret as usize;

            // clean up temporary variables to ensure that `nret`
            // at the top of the stack is the return value
            self.stack.truncate(iret + nret as usize);

            return nret as usize;
        }
        ByteCode::Return0 => {
            return 0;
        }
```

Correspondingly, the entry function `execute()` executed by the virtual machine also needs to modify the prototype, change it to return a `usize` value:

```
    pub fn execute(&mut self, proto: &FuncProto) -> usize {
```

Bytecode Traversal and Function Exit

Now that the `execute()` function is mentioned, let's talk about the traversal and exit of the bytecode sequence.

At the beginning, this project only supported sequential execution, using Rust Vec's iterator:

```
    for code in proto.byte_codes.iter() {
        match *code {
```

Later, after the jump statement is supported, it is necessary to traverse manually, and judge whether to exit by whether the `pc` exceeds the bytecode sequence:

```
    let mut pc = 0;
    while pc < proto.byte_codes.len() {
        match proto.byte_codes[pc] {
```

Lua's `return` statement is now supported, and the execution of the corresponding `Return` bytecode will exit the `execute()` function. If all Lua functions eventually contain the `Return` bytecode, there is no need to check whether the pc has exceeded the bytecode sequence to determine whether to exit. In this way, the original `while` loop in the `execute()` function can be changed to a `loop` loop, reducing a conditional judgment:

```
    let mut pc = 0;
    loop {
        match proto.byte_codes[pc] {
            ByteCode::Return0 => { // Return or Return0 bytecode, exit
 function
                return 0;
            }
```

To do this, we append the `Return0` bytecode at the end of all Lua functions:

```
fn chunk(lex: &mut Lex<impl Read>, end_token: Token) -> FuncProto {
    let mut proto = ParseProto::new(lex);
    assert_eq!(proto.block(), end_token);
    if let Some(goto) = proto. gotos. first() {
        panic!("goto {} no destination", &goto.name);
    }

    // All Lua functions end with `Return0` bytecode
    proto.fp.byte_codes.push(ByteCode::Return0);

    proto.fp
}
```

So far, the function of exiting the function and generating a return value is completed. Next, introduce the second part: the caller reads the return value.

Read Return Values: Position

After the called function returns through the `return` statement, the virtual machine execution sequence returns back to the `Call` bytecode of the outer calling function, where the return values are read and processed. How to handle the return values? It depends on the different application scenarios where the function call is made. Because the Lua function supports multiple return values, and the specific number of return values cannot be determined during the syntax analysis stage, similar to the variable parameters expression `...` in the previous section, the processing of the function return values is simlar with the variable parameter and also includes 3 scenarios:

1. When used as the last argument of a function call, the last value of a `return` statement, or the last array member of a table construction, read all return values. For example, the following example:

   ```
   print("hello: ", foo(1, 2)) -- last argument
   local t = {1, 2, foo()} -- last list member
   return a+b, foo() -- the last return value
   ```

2. When used as the last expression after the equal sign `=` of a local variable definition statement or an assignment statement, the number of return values will be expanded or reduced as required. For example, the following example:

   ```
   local x, y = foo() -- take the first 2 actual parameters and assign them
   to x and y respectively
   t.k, t.j = a, foo() -- take the first actual parameter and assign it to
   t.j
   ```

3. Other places only represent the first actual parameter passed in. For example, the following example:

```lua
local x, y = foo(), b -- not the last expression, just take the first argument and assign it to x
t.k, t.j = foo(), b -- not the last expression, just take the first argument and assign it to t.k
if foo() then -- conditional judgment
   t[foo()] = foo() + f -- table index, and binary operands
end
```

In addition, there is another scenario:

4. For a single function call statement, the return values are ignored at this time. For example, the following example:

```lua
print("no results")
foo(1, 2, 3)
```

The fourth scenario does not need to deal with the return values, so ignore it for now. In the previous three scenarios, it is necessary to move the return values from the top of the stack to the position of the function entry. For example, for the `print("hello", sqr(3, 4))` statement, the stack layout before calling the `sqr()` function is shown in the left figure below:

```
|       |        |       |                |       |
+--------+       +--------+               +--------+
| print |        | print |                | print |
+--------+       +--------+               +--------+
|"hello"|        |"hello"|                |"hello"|
+--------+       +--------+               +--------+
|  sqr  |        |  sqr  |              / |   9   | <--original sqr entry position
+--------+       +--------+ <--base   /-+  +--------+
|   3   |        |   3   |           |  \ |  16   |
+--------+       +--------+          |    +--------+
|   4   |        |   4   |           |    |       |
+--------+       +--------+          |
|       |        |   9   | \         |
                 +--------+ +return--/
                 |  16   | / values
                 +--------+
                 |       |
```

In the left picture, the `print` function is at the top of the stack, followed by the parameter `"hello"` string constant and the `sqr()` function, and then the two parameters of the `sqr()` function: `3` and `4`. The important point here is that in the syntax analysis stage, the arguments of the function are generated by `explist()` bytecodes, which are loaded onto the stack in turn, so the `sqr()` function must be located in the `print()` function's argument location. Then, the return value of the `sqr()` function should be moved to the position of the `sqr()` function as the argument of the `print()` function, as shown in the rightmost figure in the above figure.

Therefore, the above three stack layout diagrams are summarized as follows:

- The picture on the left is the state before the `sqr()` function call;
- The picture in the middle is after the `sqr()` function is called, that is, the state after the `Return` bytecode introduced in the previous part of this section is executed;
- The picture on the right is the expected state after calling the `sqr()` function, that is, the return value of the `sqr()` function is used as the return value of the `print()` function.

Therefore, what we need to do is to change the stack layout from the middle picture to the right picture, so in the processing flow of `Call` bytecode, move the return value from the top of the stack to the position of the function entry, which is the last line in the following code :

```
        ByteCode::Call(func, narg_plus) => {
            self.base += func as usize + 1;
            match &self.stack[self.base - 1] {
                Value::LuaFunction(f) => {
                    // The processing of parameters is omitted here.

                    // Call the function, `nret` is the number of return values
 at the top of the stack
                    let nret = self. execute(&f);

                    // Delete the stack values from the function entry to the
                    // starting position of the return values, so the return
                    // values are moved to the function entry position.
                    self.stack.drain(self.base+func as usize ..
 self.stack.len()-nret);
                }
```

Here, the return values are not directly moved to the function entry position, but the stack data from the function entry to the start position of the return value is cleared through the `Vec::drain()` method to realize the return value in place. This is also done to clean up the stack space occupied by the called function at the same time, so as to release resources in time.

Read Return Value: Number

The above describes where to put the return values, now let's deal with the number of return values. This is also the same as the variable parameter expression in the previous section. According to the above four scenarios, it is also divided into four types:

1. All return values;
2. Fixed the first N return values;
3. The first return value;
4. No return value is required.

Similar to `VarArgs` bytecode, `Call` bytecode also needs to add a parameter to indicate how many return values are needed:

```
pub enum ByteCode {
    Call(u8, u8, u8) // Add the third associated parameter, indicating how
many return values are required
```

But there is a difference here, that is the number of parameters associated with `VarArgs`, and the value `0` means all variable arguments. The fourth scenario is added here for the function call, which does not need a return value, that is, a return value of `0` is required, so the new associated parameters of the `Call` bytecode cannot be represented by `0` as a special value for all return values. This is like the scene in the previous section Number of parameters, that is, there are already `0` parameters, so it cannot be simply used `0` is a special value. There are two solutions to this problem:

- Refer to the processing method of the number of parameters in the previous section, use `0` to represent all return values, and change the case of fixed N return values to N+1 and encode them into the `Call` bytecode. This is also the scheme adopted by Lua's official implementation;

- Take the "no need to return value" in the fourth scenario as "ignore the return value", that is, there is no need to process the return value, or it doesn't matter how to process the return value. Then in this scenario, we can fill in any number for this associated parameter. Here we choose to fill in `0`.

We choose the latter option. That is to say, the value `0` has two meanings:

- All return values are required;
- No return value is required.

Although the meanings of these two scenarios are different, the processing method is the same when the virtual machine is executed, and the return value is not processed. In other words, all return values (if any) will be placed at the function entry.

If the value of this parameter is not `0`, it corresponds to the second and third scenarios above, that is, the situation where the first N and the first return values need to be fixedIn this case, you need to deal with:

- If the actual return value is less than the expected demand, then `nil` needs to be added;
- Otherwise, no processing is required. The extra return value is considered as a temporary variable on the stack and has no effect.

Next, add this nil filling process in the process of executing `Call` bytecode in the virtual machine:

```
            ByteCode::Call(func, narg_plus, want_nret) => {
                self.base += func as usize + 1;
                match &self.stack[self.base - 1] {
                    Value::LuaFunction(f) => {
                        let nret = self. execute(&f);
                        self.stack.drain(self.base+func as usize ..
 self.stack.len()-nret);

                        // Fill nil as needed
                        // If want_nret==0, there is no need to process it, and it
 will not enter the if{} branch.
                        let want_nret = want_nret as usize;
                        if nret < want_nret {
                            self.fill_stack(nret, want_nret - nret);
                        }
                    }
```

At this point, the virtual machine execution part of `Call` bytecode is completed.

Syntax Analysis of Scenarios

In previous chapters, we always introduce syntax analysis to generate bytecode first, and then introduce the virtual machine to execute the bytecode. But this time is different. The above only introduces the virtual machine execution of `Call` bytecode in different scenarios; it does not introduce syntax analysis, that is, how to generate `Call` bytecode in each scenario. Make it up now.

The first and second scenarios above are exactly the same as the corresponding scenario of variable parameter expressions, so there is no need to do these statements here to modify, we only need to add `ExpDesc::Call` expressions in `discharge_expand()` and `discharge_expand_want()`. The code of `discharge_expand()` is listed below, and ``discharge_expand_want()` is similar, so it is omitted here.

```
    fn discharge_expand(&mut self, desc: ExpDesc) -> bool {
        let code = match desc {
            ExpDesc::Call(ifunc, narg_plus) => { // Add function call
expression
                ByteCode::Call(ifunc as u8, narg_plus as u8, 0)
            }
            ExpDesc::VarArgs => {
                ByteCode::VarArgs(self.sp as u8, 0)
            }
            _ => {
                self.discharge(self.sp, desc);
                return false
            }
        };
        self.fp.byte_codes.push(code);
        true
    }
```

In Lua, when the number of values cannot be determined during the syntax analysis stage, there are only variable arguments and function calls. So these two functions are now complete. If there are other similar statements, we can also add statements to this function without modifying the specific application scenario.

Next, look at the third scenario, which only takes the first return value. The same as the variable arguments statement in the previous section, the loading of the `ExpDesc::Call` expression is also completed in the `discharge()` function. Unlike the variable arguments statement, the first associated parameter of the `VarArgs` bytecode generated by the variable arguments is the target address, and the three parameters associated with the `Call` bytecode here have no target address of. It is introduced above that when the virtual machine is executed, the return value is placed at the entry address of the function, but the `discharge()` function is to load the value of the expression to the specified address. Therefore, the loading of the `ExpDesc::Call` expression may require 2 bytecodes: first generate the `Call` bytecode to call the function and put the return value at the function entry position, and then generate the `Move` bytecode to put the first A return value is assigned to the target address. code show as below:

```
    fn discharge(&mut self, dst: usize, desc: ExpDesc) {
        let code = match desc {
            ExpDesc::Call(ifunc, narg_plus) => {
                // Generate Call, keep only 1 return value, and put it in
ifunc position
                self.fp.byte_codes.push(ByteCode::Call(ifunc as u8,
narg_plus as u8, 1));

                // Generate Move, copy return value from ifunc to dst
position
                self.fp.byte_codes.push(ByteCode::Move(dst as u8, ifunc as
u8));
            }
```

For example, the following sample code:

```lua
local x, y
x = foo()
```

Its stack layout is as follows:

```
|        |          |        |          |        |
+--------+          +--------+          +--------+
|   x    |          |   x    |    /---->|   x    |
+--------+          +--------+    |     +--------+
|   y    |          |   y    |    |     |   y    |
+--------+          +--------+    |     +--------+
|  foo   |    /---->|  100   |----/     |        |
+--------+    |     +--------+    Move bytecode assigns the return value to the
target address
:        :    |     |        |
+--------+    |
|  100   |----/ `Call` bytecode moves the returns value 100 to `foo` position
+--------+
|        |
```

- The picture on the left is the stack layout before the `foo()` function returns, assuming `100` at the top of the stack is the return value of the function;
- The picture in the middle shows that after the `Call` bytecode is executed, the return value is moved to the function entry position, which is the function completed above in this section;
- The figure on the right is the `Move` bytecode assigning the return value to the target address, that is, the local variable `x` .

It can be seen that 2 bytecodes are generated in this scenario, and the return value is also moved 2 times. There is room for optimization here. The reason why 2 bytecodes are needed is because the `Call` bytecode has no parameters associated with the target address, so it cannot be directly assigned. The reason why there is no associated target address parameter is because the `Call` bytecode has already stuffed 3 parameters, and there is no space to stuff it into the target address.

Once the problem is identified, the optimization solution becomes obvious. Since only one return value is always required in this scenario, the third associated parameter (the number of required return values) in `Call` bytecode is meaningless. So you can add a bytecode dedicated to this scenario, delete the third parameter in the `Call` bytecode, and make room for the parameter of the target address. For this, we add `CallSet` bytecode:

```
pub enum ByteCode {
     Call(u8, u8, u8), // Associated parameters: function entry, number of
arguments, number of expected return values
     CallSet(u8, u8, u8), // Associated parameters: target address, function
entry, number of arguments
```

In this way, in the `discharge()` function, the function call statement only needs one bytecode:

```
    fn discharge(&mut self, dst: usize, desc: ExpDesc) {
        let code = match desc {
            ExpDesc::Call(ifunc, narg) => {
                ByteCode::CallSet(dst as u8, ifunc as u8, narg as u8)
            }
```

The virtual machine execution of `CallSet` bytecode is as follows:

```
    ByteCode::CallSet(dst, func, narg) => {
        // Call functions
        let nret = self. call_function(func, narg);

        if nret == 0 { // no return value, set nil
            self. set_stack(dst, Value::Nil);
        } else {
            // use swap() to avoid clone()
            let iret = self.stack.len() - nret as usize;
            self.stack.swap(self.base+dst as usize, iret);
        }

        // Clean up the stack space occupied by the function call
        self.stack.truncate(self.base + func as usize + 1);
    }
```

The `call_function()` method in the above is a function that extracts the execution flow of `Call` bytecode. After calling the function, if there is no return value, set the target address to `nil`, otherwise assign the first return value to the target address. The last line cleans up the stack space occupied by function calls, and there are 2 cases in cleaning:

- If the target address is a local variable, then the cleanup location is from the function entry;
- If the target address is a temporary variable, set the target address of the function return value as the function entry position in `discharge_any()`, so the cleaning position starts from one position behind the function entry.

In summary, always start cleaning from a position behind the function entry position, which can satisfy the above two conditions. Only in the case of local variables, one more function entry will be reserved.

Variable Number of Return Values

The syntax analysis and virtual machine execution of the return value are introduced above, but one place is still missing. Among the three application scenarios of variable parameters listed in the previous section, the first scenario includes three statements: table construction, function argument, and function return value. At that time, only the first two statements were introduced. Now that the return value statement is supported, the last statement is added.

This section above introduces the syntax analysis of the return statement, but at that time, all expressions of the return value were loaded onto the stack in sequence, that is, only a fixed number of return values was supported. When the last expression of the function return value statement is a variable parameter or a function call statement, then all variable parameters or all return values of the function when the virtual machine is executed will be used as the return value of this function, that is to say, the number of return values cannot be determined during the parsing phase, that is, a variable number of return values.

Variable number of return values, syntax analysis can refer to the previous section table construction or function arguments, that is, use the modified `explist()` function , special treatment is given to the last expression. The specific code is omitted here.

What needs to be explained is how to represent "variable number" in bytecode. In this section, two new return value-related bytecodes are added, `Return0` and `Return` . Among them, `Return0` is used when there is no return value, so the parameter of the number of return values associated in `Return` bytecode will not be `0` , then `0` can be used as a special value to indicate variable number.

Summary of Variable Number Statements and Scenario

Here is a summary of statements and scenarios related to variable quantities. Statements that directly result in variable numbers include:

- Variable argument statement `...` , there are 3 application scenarios;
- Function call statement, in addition to the 3 application scenarios of variable parameters, there is also a scenario of ignoring the return value.

Among the several application scenarios of these two statements, the first scenario is to take all the actual parameters or return values when the virtual machine is executed. This scenario includes 3 statements:

- Table construction, corresponding to `SetList` bytecode;
- Function arguments, corresponding to `Call/CallSet` bytecode;

- The return value of the function corresponds to the `Return` bytecode of the called function and the `Call/CallSet` bytecode of the calling function.

In the above bytecodes, in order to represent the state of "actually all expressions when the virtual machine is executed", `0` is used as a special value, among which:

- The second parameter of `Call/CallSet` bytecode represents the number of actual parameters. Because the function call originally supports no parameters, in order to use `0` as a special value, we have to correcte the number with adding by 1 for fixed number case, that is, N fixed parameters are encoded into N+1 in the bytecode;

- The third parameter of `Call/CallSet` bytecode represents the number of expected return values. The function call also supports the situation that the return value is not required, but we understand "no need" as "ignore", then it is no problem to read all the return values, so `0` can be used as a special value;

- The second parameter of `SetList` and `Return` bytecodes both represent the number. However, when these two bytecodes are used for fixed numbers, no expressions are supported, so `0` can be directly used as a special value.

In addition, it needs to be emphasized again that when `0` is used to represent a special value in the above bytecode, the number of specific expressions is calculated from the top of the stack, which must ensure that there is no temporary variable on the top of the stack, so the virtual machine must explicitly clean up temporary variables when executing variable parameter and function call statements.

Summary

This section begins by introducing fixed number return values. The called function puts the return value on the top of the stack through the `Return/Return0` bytecode, and then the calling function reads the return value in the `Call/CallSet` bytecode.

The variable number of return values was introduced later, which is similar to the variable parameters in the previous section.

Rust Function and APIs

The previous three sections of this chapter introduce the functions defined in Lua, and this section introduces the functions defined in Rust. For the sake of simplicity, these two types of functions are called "Lua functions" and "Rust functions" respectively.

In fact, we have already been exposed to Rust functions. The `print()` that was supported in the `hello, world!` version of the first chapter is the Rust function. The interpreter at that time realized the definition and calling process of Rust functions. which is defined as follows:

```
pub enum Value {
    RustFunction(fn (&mut ExeState) -> i32),
```

Here is an example of the implementation code of `print()` function:

```
fn lib_print(state: &mut ExeState) -> i32 {
    println!("{}", state.stack[state.base + 1]);
    0
}
```

The calling method of the Rust function is also similar to the Lua function, and the Rust function is also called in the `Call` bytecode:

```
    ByteCode::Call(func, _) => {
        let func = &self. stack[func as usize];
        if let Value::Function(f) = func {
            f(self);
```

The codes listed above are the functions of the implemented Rust functions, but they are only the most basic definitions and calls, and still lack parameters and return values. This section adds these two features to Rust functions.

One thing that needs to be explained is that in Lua code, the function call statement does not distinguish between Lua functions and Rust functions. In other words, the two types are not distinguished during the parsing phase. It is only in the virtual machine execution stage that the two types need to be treated differently. Therefore, what is described below in this section is all about the virtual machine stage.

Argument

The arguments of Rust functions are also passed through the stack.

You can see that the implementation of the current `print()` function only supports one parameter, which is by directly reading the data on the stack: `state.stack[state.base + 1])` , where `self.base` is the function entry address , `+1` is the address immediately following, that is, the first parameter.

Now to support multiple parameters, it is necessary to inform the Rust function of the specific number of parameters. There are two options:

- Modify the Rust function prototype definition, add a parameter to express the number of parameters. This solution is simple to implement, but it is inconsistent with Lua's official C function prototype;
- Adopt the variable parameter mechanism in the previous Lua function, that is, determine the number of parameters by the position of the top of the stack.

We take the latter approach. This requires cleaning up possible temporary variables on the top of the stack before calling the function Rust:

```
    ByteCode::Call(func, narg_plus) => {
        let func = &self. stack[func as usize];
        if let Value::Function(f) = func {
            // narg_plus!=0, fixed parameters, need to clean up possible
            //               temporary variables on the top of the stack;
            // narg_plus==0, variable parameters, no need to clean up.
            if narg_plus != 0 {
                self.stack.truncate(self.base + narg_plus as usize - 1);
            }

            f(self);
```

After cleaning up the possible temporary variables at the top of the stack, in the Rust function, the specific number of parameters can be judged through the top of the stack: `state.stack.len() - state.base` ; we can also directly read any argument, such as the Nth parameter: `state.stack[state.base + N])` . So modify the `print()` function as follows:

```
fn lib_print(state: &mut ExeState) -> i32 {
    let narg = state.stack.len() - state.base; // number of arguments
    for i in 0 .. narg {
        if i != 0 {
            print!("\t");
        }
        print!("{}", state.stack[state.base + i]); // print the i-th
argument
    }
    println!("");
    0
}
```

Return Value

The return value of the Rust function is also passed through the stack. The Rust function puts the return value on the top of the stack before exiting, and returns the number, which is the function of the `i32` type return value of the Lua function prototype. This is the same mechanism as the Lua function introduced in the previous section. We only need to process the return value of the Rust function according to the return value of the Lua function introduced in the previous section when the `Call` bytecode is executed:

```
        ByteCode::Call(func, narg_plus) => {
            let func = &self. stack[func as usize];
            if let Value::Function(f) = func {
                if narg_plus != 0 {
                     self.stack.truncate(self.base + narg_plus as usize - 1);
                }

                // Return the number of return values of the Rust function,
                // which is consistent with the Lua function
                f(self) as usize
```

Convert the return value of the Rust function `f()` from `i32` to `usize` type and return, indicating the number of return values. Here the type conversion from `i32` to `usize` feels bad, because the C function in the official Lua implementation returns a negative number to indicate failure. We have directly panicked on all errors so far. Subsequent chapters will deal with errors uniformly. When `Option<usize>` is used instead of `i32`, this garish conversion will be removed.

The previous `print()` function had no return value and returned `0`, so it did not reflect the feature of return value. Let's take another Lua standard library function `type()` with a return value as an example. The function of this function is to return the type of the first parameter, and the type of the return value is a string, such as "nil", "string", "number" and so on.

```
fn lib_type(state: &mut ExeState) -> i32 {
     let ty = state.stack[state.base + 1].ty(); // the type of the first
parameter
     state.stack.push(ty); // Push the result onto the stack
     1 // Only 1 return value
}
```

Among them, the `ty()` function is a new method for the `Value` type, which returns a description of the type, and the specific code is omitted here.

Rust API

So far, the characteristics of the parameters and return values of Rust functions have been realized. However, the access and processing of arguments and return values above are too direct, and the ability of Rust functions is too strong, not only can access the parameters of the current function, but also can access the entire stack space, and even the entire `state` state. This is irrational and dangerous. It is necessary to restrict the access of Rust functions to `state`, including the entire stack, which requires the limited ability of Rust functions to access `state` through the API. We have come to a new world: the Rust API, of course called the C API in the official Lua implementation.

The Rust API is an API provided by the Lua interpreter for Rust functions (the Lua library implemented by Rust). Its roles are as follows:

```
       +------------------+
       |     Lua code     |
       +---+----------+---+
           |          |
           |  +-------V----------+
           |  | Standard Library |
           |  |    (Rust)        |
           |  +-------+----------+
           |          |Rust API
           |          |
 +---------V----------V--------+
 | Lua Virtual Machine (Rust) |
 +-----------------------------+
```

There are 3 functional requirements in the Rust function in the above section, all of which should be fulfilled by the API:

- Read the actual number of arguments;
- read specified argument;
- create return value

These three requirements are described in turn below. The first is the function of reading the actual number of arguments, which corresponds to the `lua_gettop()` API in the official implementation of Lua. For this we provide `get_top()` API:

```rust
impl<'a> ExeState {
    // Return to the top of the stack, that is, the number of parameters
    pub fn get_top(&self) -> usize {
        self.stack.len() - self.base
    }
```

Although the `get_top()` function is also a method of the `ExeState` structure, it is provided as an API for external calls. The methods before `ExeState` (such as `execute()`, `get_stack()`, etc.) are all internal methods for virtual machine execution calls. In order to distinguish these two types of methods, we add an `impl` block to the `ExeState` structure to implement the API alone to increase readability. It's just that Rust does not allow the method of implementing the structure in different files, so it cannot be split into another file.

Then, the function of reading the specified parameters does not correspond to a function in the official Lua implementation, but a series of functions, such as `lua_toboolean()`, `lua_tolstring()`, etc., for different types. With the generic capabilities of the Rust language, we can provide only one API:

```
    pub fn get<T>(&'a self, i: isize) -> T where T: From<&'a Value> {
        let narg = self. get_top();
        if i > 0 { // positive index, counting from self.base
            let i = i as usize;
            if i > narg {
                panic!("invalid index: {i} {narg}");
            }
            (&self. stack[self. base + i - 1]). into()
        } else if i < 0 { // Negative index, counting from the top of the
stack
            let i = -i as usize;
            if i > narg {
                panic!("invalid index: -{i} {narg}");
            }
            (&self.stack[self.stack.len() - i]).into()
        } else {
            panic!("invalid 0 index");
        }
    }
```

You can see that this API also supports negative indexes, which means counting down from the top of the stack, which is the behavior of Lua's official API, and it is also a very common method of use. This also reflects the advantages of the API over direct access to the stack.

However, there is also a behavior that is inconsistent with the official API: when the index exceeds the stack range, the official will return `nil`, but here we panic directly. We will discuss this in detail later when we introduce error handling.

Based on the above two APIs, you can redo the `print()` function:

```
fn lib_print(state: &mut ExeState) -> i32 {
    for i in 1 ..= state. get_top() {
        if i != 1 {
            print!("\t");
        }
        print!("{}", state.get::<&Value>(i).to_string());
    }
    println!("");
    0
}
```

Finally, let's look at the last function, creating the return value. Like the above API for reading arguments, there are also a series of functions in the official Lua implementation, such as `lua_pushboolean()`, `lua_pushlstring()`, etc. And here you can also add only one API with the help of generics:

```
    pub fn push(&mut self, v: impl Into<Value>) {
        self.stack.push(v.into());
    }
```

Based on this API, `self.stack.push()` in the last line of `type()` function above can be changed to `self.push()`.

Although the implementation of the `print()` and `type()` functions has not changed significantly after replacing the API, the API provides a encapsulation for `ExeState`, which will gradually reflect the convenience in the process of gradually adding library functions safety.

Tail Call

The Lua language supports tail call elimination. This section describes and supports tail calls.

First we introduce the concept of tail calls. A tail call is formed when the last action of a function is to call another function without doing any other work. For example, the following sample code:

```lua
function foo(a, b)
     return bar(a + b)
end
```

The last action of the `foo()` function (and in this case the only action) is to call the `bar()` function. Let's take a look at the execution process of the `foo()` function without introducing a tail call, as shown in the following figure:

```
|        |          |        |          |        |          |        |
+--------+          +--------+          +--------+          +--------+
| foo()  |          | foo()  |          | foo()  |       /  |  ret1  |
+--------+<<        +--------+          +--------+<<   /-+  +--------+
|   a    |          |   a    |          |   a    |     | \  |  ret2  |
+--------+          +--------+          +--------+     |    +--------+
|   b    |          |   b    |          |   b    |     |    |        |
+--------+          +--------+          +--------+     |
| bar()  |          | bar()  |       /  |  ret1  | \   |
+--------+          +--------+<<   /-+  +--------+  >-/return values
|  a+b   |          |  a+b   |     | \  |  ret2  | /
+--------+          +--------+     |    +--------+
|        |          :        :     |    |        |
                    +--------+     |
                    |  ret1  | \   |
                    +--------+  >-/return values
                    |  ret2  | /
                    +--------+
                    |        |
```

- The first picture on the far left is the stack layout before calling the `bar()` function inside the `foo()` function. That is, before calling `Call(bar)` bytecode.
- The second figure is the stack layout immediately after the `bar()` function call has completed. That is, after the `Return` bytecode of the `bar()` function is executed, but before returning to the `Call(bar)` bytecode of the `foo()` function. Suppose this function has two return values `ret1` and `ret2`, which are currently on the top of the stack.
- The third figure is the stack layout after the `bar()` function returns. That is, the `Call(bar)` bytecode of `foo()` is executed. That is, move the two return values to

the entry function position of `bar()` .

- The fourth figure is the stack layout after the `foo()` function returns. That is, after the `Call(foo)` bytecode of the outer caller is executed. That is, move the two return values to the entry function position of `foo()` .

The next three graphs are executed consecutively. Observe the optimization space in it:

- An obvious optimization idea is that the copying of the last two return values can be completed in one step. But this is difficult to optimize, and it doesn't optimize much performance;
- Another not-so-obvious point is that the stack space of the `foo()` function is no longer used after the `bar()` function in the first leftmost figure is ready to be called. Therefore, we can clean up the stack space occupied by the `foo()` function before calling the `bar()` function. According to this idea, the following redraws the calling process:

```
|        |         |        |         |        |         |        |
+--------+         +--------+         +--------+         +--------+
| foo()  |       / | bar()  |         | bar()  |       / |  ret1  |
+--------<<    /-+ +--------<<        +--------<<    /-+ +--------+
|   a    |     | \ |  a+b   |         |  a+b   |     | \ |  ret2  |
+--------+     |   +--------+         +--------+     |   +--------+
|   b    |     |   |        |         :        :     |   |        |
+--------+     |                      +--------+     |
| bar()  | \   |                      |  ret1  | \   |
+--------+   >-/                      +--------+   >-/
|  a+b   | /                          |  ret2  | /
+--------+                            +--------+
|        |                            |        |
```

- The first picture on the left remains unchanged, and it is still the state before the `bar()` function call;
- In the second picture, before calling `bar()` , the stack space of the `foo()` function is cleared;
- The third picture, corresponding to the second picture above, is after calling `bar()` function.
- The fourth picture corresponds to the last picture above. Since the stack space of the `foo()` function has been cleaned up just now, the third figure above is skipped.

Compared with the above ordinary process, although the operation steps of this new process have been changed, they have not been reduced, so the performance is not optimized. However, there are optimizations in the use of stack space! The stack space of `foo()` has been freed before the `bar()` function is executed. 2 layers of function calls, but only takes up 1 layer of space. The advantage brought by this is not obvious in this example, but it is obvious in recursive calls, because there are usually many layers of recursive calls. If the last item of the recursive call satisfies the above tail call, then after applying the new process, it can support unlimited recursive calls without causing stack overflow! The stack overflow here refers to the stack of the Lua virtual machine drawn in the above figure, not the stack overflow of the Rust program.

Compared with the normal process above, this new process has a small difference. The `<<` on the stack in each figure above represents the current `self.base` position. It can be seen that in the above ordinary process, `self.base` has changed; but in the new process, the whole process has not changed.

After introducing the concept of tail call, the specific implementation is introduced below.

Syntax Analysis

Before starting the syntax analysis, clarify the rules of the next tail call again: when the last action of a function is to call another function without doing other work, it forms a tail call. Here are some counterexamples from the book "Lua Programming":

```lua
function f1(x)
     g(x) -- discard the return value of g(x) before f1() returns
end
function f2(x)
     return g(x) + 1 -- also execute +1
end
function f3(x)
     return x or g(x) -- also limit the return value of g(x) to 1
end
function f4(x)
     return (g(x)) -- also limit the return value of g(x) to 1
end
```

In the Lua language, only calls of the form `return func(args)` are tail calls. Of course, `func` and `args` here can be very complicated, such as `return t.k(a+b.f())` is also a tail call.

After the rules are clarified, it is relatively simple to judge tail calls during syntax analysis. When parsing the return statement, add a judgment on the tail call:

```
        let iret = self.sp;
        let (nexp, last_exp) = self.explist();

        if let (0, &ExpDesc::Local(i)) = (nexp, &last_exp) {
            // There is only 1 return value and it is a local variable
            ByteCode::Return(i as u8, 1)

        } else if let (0, &ExpDesc::Call(func, narg_plus)) = (nexp, &last_exp) {
            // New tail call: only one return value, and it is a function call
            ByteCode::TailCall(func as u8, narg_plus as u8)

        } else if self. discharge_expand(last_exp) {
            // The last return value is a variable type, such as variable
 arguements or function calls,
            // then the number of return values cannot be known during the
 syntax analysis phase
            ByteCode::Return(iret as u8, 0)

        } else {
            // The last return value is fixed
            ByteCode::Return(iret as u8, nexp as u8 + 1)
        }
```

There are 4 cases in the above code. The second case is a newly added tail call, and the other three cases are already supported in the previous sections of this chapter, so they will not be introduced here.

The newly added bytecode `TailCall` is similar to the function call bytecode `Call`, but since the return value of the tail call must be a function call, the number of return values must be unknown, so the third associated parameter is omitted. So far, there are three bytecodes related to function calls:

```
pub enum ByteCode {
    Call(u8, u8, u8),
    CallSet(u8, u8, u8),
    TailCall(u8, u8), // add tail call
```

Virtual Machine Execution

Next, look at the virtual machine execution part of the tail call. From the tail call process introduced at the beginning of this section, it can be concluded that compared with ordinary function calls, there are three differences in the execution of tail calls:

- Before calling the inner function, the stack space of the outer function should be cleared in advance, which is also the meaning of tail call;
- After the inner function returns, since the outer function has been cleaned up, there is no need to return to the outer function, but directly return to the outer calling

function.
- There is no need to adjust `self.base` throughout.

Thus, the execution flow of `TailCall` bytecode can be realized as follows:

```
ByteCode::TailCall(func, narg_plus) => {
    self.stack.drain(self.base-1 .. self.base+func as usize);
    return self. do_call_function(narg_plus);
}
```

Very simple, just two lines of code:

Line 1, through `self.stack.drain()` to clean up the stack space of the outer function.

Line 2 returns directly from the current `execute()` through the `return` statement, that is to say, after the inner function is executed, it does not need to return to the current function, but directly returns to the outer caller. In addition, according to the rules of tail calls listed above, this line of Rust code itself is also a tail call. So as long as the Rust language also supports tail call elimination, then our Lua interpreter will not increase its own stack during execution.

In addition, the newly added `do_call_function()` method in line 2 executes the function call, which is extracted from the `call_function()` method called by the `Call` and `CallSet` bytecodes in the previous section, except that the update to `self.base` is removed. And the `call_function()` method is modified to wrap this new method:

```
fn call_function(&mut self, func: u8, narg_plus: u8) -> usize {
    self.base += func as usize + 1; // get into new world
    let nret = self. do_call_function(narg_plus);
    self.base -= func as usize + 1; // come back
    nret
}
```

Test

So far, we have completed the tail call. Verify with the following Lua code:

```
function f(n)
    if n > 10000 then return n end
    return f(n+1)
end
print(f(0))
```

But I get a stack overflow error when executing:

```
$ cargo r --test_lua/tailcall.lua

thread 'main' has overflowed its stack
fatal runtime error: stack overflow
[1] 85084 abort cargo r -- test_lua/tailcall.lua
```

At first I thought that the debug version of Rust did not perform tail call optimization, but after adding `--release`, it can only support a greater recursion depth, which delays the stack overflow, but eventually the stack overflow will still occur. This goes back to what I just said: "So as long as the Rust language also supports tail call elimination, then...", the assumption in front of this sentence may not be true, that is, the Rust language may not support tail call elimination. Here is an article that introduces the discussion of tail calls in the Rust language. The conclusion is probably due to the implementation too complicated (may involve resource drop), and the benefits are limited (programmers can manually change recursion to loop if necessary), so in the end Rust language does not support tail call elimination. In this way, in order to make the tail call elimination of Lua completed in this section meaningful, we can only change the recursive call to the `execute()` function into a loop. This change itself is not difficult, but there are still two places to modify the function call process in the future, one is the calling method of the entry function of the entire program, and the other is to support the state preservation of the function in the coroutine. So we will make this change after completing the final function call process.

Closure

The previous chapter introduced functions, and all functions in the Lua language are actually closures. This chapter introduces closures.

The so-called closure is the function prototype associated with some variables. In Lua, these associated variables are called Upvalue. If you understand closures in Rust, then according to "Rust Programming Language" it is "capturing environment" means the same thing as "associated variable". So Upvalue is fundamental to understanding and implementing closures.

The section 1 of this chapter introduces the most basic concept of Upvalue; the following sections 2 and 3 introduce the important feature of Upvalue, escape, which is what makes the closure really powerful; Section 4 introduces the Rust closure corresponding to the Rust function. The following sections 5 and 6 are the two application scenarios of closure and Upvalue respectively.

Upvalue

Before introducing closures, this section first introduces an important part of closures: upvalue.

This section mainly introduces the concept of upvalue, and introduces the changes needed to support upvalue in the syntax analysis and virtual machine execution stages. It is very complicated to realize the complete features of upvalue, so in order to focus on the change of the overall structure and process, this section only supports the most basic upvalue features, and leave it to the next section to introduce the difficult part: escape.

The sample code below shows the most basic scenario of upvalue:

```lua
local a = 1
local function foo()
    print(a) -- What type of variable is `a`? Local variable, or global
variable?
end
```

The entire code can be seen as a top-level function, which defines two local variables: `a` and the function `foo`. The reference `a` in `print(a)` inside the `foo()` function refers to a local variable defined outside the function, so what kind of variable is the `a` inside the function? First, it is not defined inside the `foo()` function, so it is not a local variable; second, it is a local variable defined in the outer function, so it is not a global variable. Local variables that refer to outer functions like this are called *upvalue* in Lua. There is also the concept of closure in the Rust language, and local variables in the outer function can also be referenced, which is called "capture environment", which should be the same concept with upvalue.

upvalues are very common in Lua. In addition to the above-mentioned obvious cases, there is also the fact that calling local functions at the same level is also an upvalue, such as the following code:

```lua
local function foo()
    print "hello, world"
end
local function bar()
    foo() -- upvalue
end
```

The `foo()` function called in the `bar()` function is upvalue. In addition, recursive calls to local functions are also upvalue.

After introducing the concept of upvalue, the syntax analysis process of upvalue is as follows.

Variable Resolution Process

The previous interpreter only supports two variable types: local variables and global variables. The variable parsing process is as follows:

1. Match in the local variable list of the current function, if found, it is a local variable;
2. Otherwise, it is a global variable.

Now to add support for the upvalue type, the parsing process of the variable needs to be added one step, which is changed to:

1. Match in the local variable list of the current function, if found, it is a local variable;
2. Match in the local variables list of upper layer functions, if found, it will be upvalue; (NEW STEP)
3. Otherwise it is a global variable.

The newly added step 2 looks simple, but the specific implementation is very complicated, and the description here is not accurate, which will be described in detail in the next section. This section focuses on the overall process, that is, how to deal with it after parsing the upvalue.

Similar to local variables and global variables, a new type of ExpDesc is also added for upvalue:

```
enum ExpDesc {
    Local(usize), // local variables or temporary variables on the stack
    upvalue(usize), // upvalue
    Global(usize), // global variable
```

To review, the associated parameter of the local variable `ExpDesc::Local` represents the index on the stack, and the associated parameter of the global variable `ExpDesc::Global` represents the index of the variable name in the constant table. What parameters do upvalue need to be associated with? Take the following sample code that contains multiple upvalues as an example:

```
local a, b, c = 100, 200, 300
local function foo()
    print (c, b)
end
```

In the above code, there are two upvalues in the `foo()` function, `c` and `b`, which correspond to the index 2 and 1 of the local variable in the upper function respectively (the index starts counting from 0), so naturally, they can be represented by `ExpDesc::upvalue(2)` and `ExpDesc::upvalue(1)`. In this way, when the virtual machine is executing, it can also conveniently index to the local variables on the stack of the upper layer function. Simple and natural. But when the escape of upvalue is introduced in the next section, this solution cannot meet the requirements. But for the sake of simplicity, this section will be used for the time being.

Parsing Context

The new step 2 in the above variable resolution process requires access to local variables of the outer functions. In the last chapter Analysis Function, recursion is used to support multi-layer function definition. This is not only simple to implement, but also provides a certain degree of encapsulation, that is, only the information of the current function can be accessed. This is originally an advantage, but now in order to support upvalue, we need to access the local variables of the outer function, so this encapsulation becomes a disadvantage that needs to be overcome. Programs are becoming more and more complex and confusing in such ever-increasing demands.

When recursively parsing multi-layer functions before, there is one member throughout, that is, `lex: Lex<R>` in `ParseProto`, which needs to be accessed when parsing all functions. Now in order to be able to access the local variables of the outer function, a similar member is needed throughout to store the local variables of each function. To do this, we create a new data structure containing the original `lex` and the new list of local variables:

```rust
struct ParseContext<R: Read> {
    all_locals: Vec<Vec<String>>, // Local variables of each layer function
    lex: Lex<R>,
}
```

The `all_locals` member represents the local variable list of each layer function. Each time a function of a new layer is parsed, a new member is pushed into it; after parsing is complete, it is popped. So the last member in the list is the list of local variables for the current function.

Then in `ParseProto`, replace the original `lex` with `ctx`, and delete the original locals:

```rust
struct ParseProto<'a, R: Read> {
    // delete: locals: Vec<String>,
    ctx: &'a mut ParseContext<R>, // add ctx to replace the original lex
    ...
```

And all places where the locals field is used in the syntax analysis code must also be modified to the last member of ctx.all_locals, which is the local variable list of the current function. The specific code is omitted here.

So far, there are three data structures related to syntax analysis:

- `FuncProto` , which defines the function prototype, is the output of the syntax analysis stage and the input of the virtual machine execution stage, so all fields are `pub` ;
- `ParseProto` , used internally in the parsing phase, and only in the current function;
- `ParseContext` , the global state used internally by the parsing phase and accessible at all function levels.

After the transformation of `ParseProto` , with the ability to access the outer function, the upvalue can be parsed. But here is just saying that it has the ability to analyze, and the specific analysis process will be introduced in the next section.

Bytecode

After parsing the upvalue, for its processing, you can refer to the previous discussion of global variables, the conclusion is as follows:

- read, first loaded on the stack, converted to a temporary variable;
- Assignment, only supports assignment from local/temporary variables and constants. For other types of expressions, it is first loaded into a temporary variable on the stack and then assigned.

To this end, compared with global variables, add 3 upvalue-related bytecodes:

```
pub enum ByteCode {
    // global variable
    GetGlobal(u8, u8),
    SetGlobal(u8, u8),
    SetGlobalConst(u8, u8),

    //upvalue
    Getupvalue(u8, u8), // Load upvalue onto the stack
    Setupvalue(u8, u8), // Assign value from the stack
    SetupvalueConst(u8, u8), // assign value from constant
```

The generation of these three new bytecodes can also be completed by referring to global variables. The specific code is omitted here.

Virtual Machine Execution

The analysis process of upvalue is introduced above, and the corresponding bytecode has completed the syntax analysis stage. The rest is the virtual machine execution phase.

According to the above processing scheme for upvalue, that is, the associated parameter of `ExpDesc::upvalue` represents the local variable index of the upper-level function. When the virtual machine is executed, it will also encounter the same problem as the syntax analysis: the current function needs to access the upper-level functions' local variables. Therefore, in order to complete the virtual machine execution phase, big changes must be made to the current code structure.

However, the above-mentioned upvalue processing scheme is only a temporary scheme in this section. In the next section, in order to support the escape of upvalue, there will be a completely different scheme and a completely different virtual machine execution process. Therefore, in order to avoid useless work, the execution of the virtual machine under this scheme will not be implemented for the time being. Interested friends can try to change it.

Upvalue Escape and Closure

The previous section introduced the concept of upvalue, and took the most basic usage of upvalue as an example to introduce the modification of syntax analysis to support upvalue. This section introduces the complete features of upvalue, mainly the escape of upvalue.

The following refers to the sample code in the book "Lua Programming":

```lua
local function newCounter()
     local i = 0
     return function ()
         i = i + 1 -- upvalue
         print(i)
     end
end

local c1 = newCounter()
c1() -- output: 1
c1() -- output: 2
```

`newCounter()` in the above code is a typical factory function, which creates and returns an anonymous function. What needs to be explained here is that the returned anonymous function refers to the local variable `i` in `newCounter()` , which is upvalue. The second half of the code calls the `newCounter()` function and assigns the returned anonymous function to `c1` and calls it. At this time, the `newCounter()` function has ended, and it seems that the local variable `i` defined in it has also exceeded the scope. At this time, calling `c1` to refer to the local variable `i` will cause problems (if you are C Language programmers should understand this). However, in Lua, the closure mechanism ensures that calling `c1` here is no problem. That is, the escape of upvalue.

The book "Lua Programming" is for Lua programmers, and it is enough to introduce the concept of upvalue escape. But our purpose is to *implement* an interpreter (not just *use* an interpreter), so we must not only know that this is no problem, but also know how to do it, that is, how to realize the escape of upvalue.

Unfeasible Static Storage Solution

The easiest way is to refer to the static variable inside the function in C language. For the local variable referenced by upvalue (such as `i` in the `newCounter()` function here), it is not placed on the stack, but placed in a static area. But this solution is not feasible, because the static variable in C language is globally unique, and the upvalue in Lua will generate a new copy every time it is called. For example, following the above code, continue with the following code:

```
local c2 = newCounter()
c2() -- output: `1`. A new count starts.
c1() -- output: `3`. Continue with the output of c1 above.
```

Calling `newCounter()` again will generate a new counter, in which the local variable `i` will be re-initialized to 0 and start counting again. At this point, there are two counters: `c1` and `c2`, each of which has an independent local variable `i`. So when `c2()` is called, it will start counting again from 1; if interspersed with `c1()` before calling, it will continue the previous counting. How interesting!

```
   --+---+--               +---+   +---+
     | i |                 | i |   | i |
   --+-^-+--               +-^-+   +-^-+
       |                     |       |
   /---+---\                 |       |
   |       |                 |       |
+-----+  +-----+          +-----+  +-----+
| c1  |  | c2  |          | c1  |  | c2  |
+-----+  +-----+          +-----+  +-----+
```

The left figure above shows that `i` is placed in the globally unique static storage, then all counter functions point to the unique i. This does not meet our needs. What we need is a separate `i` for each counter function as shown on the right.

Storage Scheme on the Heap

Since it cannot be placed on the stack, nor can it be placed in the global static area, it can only be placed on the heap. The next question is, when to put it on the heap? There are several possible scenarios:

1. When entering the function, put all local variables referenced by upvalue on the heap;
2. When a local variable is referenced by upvalue, it is moved from the stack to the heap;
3. When the function exits, move all local variables referenced by upvalue to the heap;

The first solution does not work, because a local variable may have been used as a local variable before it is referenced by upvalue, and related bytecodes have been generated. The second solution should be feasible, but after the local variable is referenced by upvalue, it may be used as a local variable in the current function. It is not necessary to move it to the heap in advance. After all, access to the stack is faster and more convenient. So we choose option 3.

This operation of moving local variables from the stack to the heap, we follow the code implemented by Lua official, is also called "close".

Next, in order to demonstrate that an upvalue is accessed before and after escaping, we modify the sample code based on the counter sample above. The inner function is called once inside the `newCounter()` function before returned. To do this, we assign this anonymous function to a local variable `retf`:

```lua
local function newCounter()
     local i = 0
     local function retf()
         i = i + 1 -- upvalue
         print(i)
     end
     retf() -- called inside newCounter()
     return retf -- return retf
end
```

This example is introduced in two parts. First, `retf()` is called inside the factory function, and then the upvalue escape caused by the factory function returning `retf`.

First of all, when `retf()` is called inside the factory function, `i` to be operated by `retf` is still on the stack. The schematic diagram is as follows.

```
     |           |
     +-----------+
base |newCounter|
     +-----------+
   0 |     i     |<- - - - - - - - - - \
     +-----------+                     |
   1 |    retf   +--+->+-FuncProto-----+--+
     +-----------+  |  |byte_codes:    |  |
   2 |    retf   +--/  | GetUpvalue(0, 0) |
     +-----------+     | ...              |
     |           |     +------------------+
```

In the figure, the stack is on the left, where `newCounter` is the entry of the function call and the base position of the current function. The `i` and the first `retf` are local variables, and the second `retf` is the entry on the stack of the function call. The two `retf`s point to same function prototype. In the bytecode sequence in the `retf` function prototype, the first bytecode `Getupvalue` is to load upvalue `i` onto the stack to perform addition. This bytecode has two associated parameters. The first is the target address loaded onto the stack, which is ignored here; the second is the source address of upvalue, refer to the syntax analysis of upvalue in the previous section, the meaning of this parameter is: the stack of local variables of the upper function index. In this example, it is the index of `i` in the newCounter() function, which is `0`. So far, it is still the content of the previous section, and escape has not been involved.

Now consider the escape of upvalue. After the `newCounter()` function exits, the three spaces on the left stack will be destroyed, and `i` will no longer exist. In order for the retf function to continue to access `i`, before the `newCounter()` function exits, it is necessary to close the local variable `i` and move `i` from the stack to the heap. The schematic diagram is as follows:

```
     |          |
     +----------+
base |newCounter|         +===+
     +----------+  close  | i |<- - - \
   0 |    i     +-------->+===+       ?
     +----------+                     ?
   1 |   retf   +---->+-FuncProto-----?--+
     +----------+     |byte_codes:    ?  |
     |          |     | GetUpvalue(0, 0) |
                      | ...              |
                      +------------------+
```

Although this ensures that `i` can continue to be accessed, there is a very obvious problem: the second parameter associated with the bytecode `Getupvalue` cannot locate `i` on the heap (the continuous `?` in the figure Wire). This is also mentioned in the previous section, it is not feasible to directly use the index of the local variable on the stack to represent the upvalue scheme. Improvements need to be made on the basis of this scheme.

Improvement: Upvalue Intermediary

In order to still be able to be accessed by upvalue after closing the local variable, we need an upvalue intermediary. At the beginning, the index on the stack is used to represent the upvalue, and when the local variable of the outer function is closed, it is moved to this intermediary.

The following two figures show the situation after adding the upvalue intermediary.

```
     |          |             - - - - - - \
     +----------+            |            |
base |newCounter|            |      *-----+-+---
     +----------+            |      |Open(0)|
   0 |    i     |<- - - - - -       *-^-----+---
     +----------+                     |
   1 |   retf   +--+->+-FuncProto-----+--+
     +----------+  |  |byte_codes:    |  |
   2 |   retf   +--/  | GetUpvalue(0, 0) |
     +----------+     | ...              |
     |          |     +------------------+
```

The figure above is a schematic diagram of calling the `retf()` function inside the `newCounter()` function. Compared with the previous version, the upvalue intermediary list (the list with `*` as the corner in the figure) is added, and there is only one member: `Open(0)` , which means that this local variable has not been closed and is on the stack The relative index is 0. In the function prototype of `retf` , although the second parameter associated with the bytecode `Getupvalue` has not changed, its meaning has changed, and it has become the index of the intermediary list. It just happens to be 0 in this example.

```
     |          |           /----------------\
     +----------+           |                |
base |newCounter|           |        *-------V-+---
     +----------+  close    |        |Closed(i)|
   0 |    i     +-----------/        *-^-------+---
     +----------+                      |
   1 |   retf   +---->+-FuncProto------+--+
     +----------+     |byte_codes:     |  |
     |          |     | GetUpvalue(0, 0) |
                      | ...              |
                      +-------------------+
```

The figure above is a schematic diagram after the local variable `i` is closed before the `newCounter()` function returns. The members of the upvalue intermediary list added in the figure above become `Closed(i)` , that is, the local variable `i` is moved to this intermediary list. In this way, `Getupvalue` can still locate the 0th upvalue intermediary and access the closed `i` .

Improvement: Shared Upvalue

The above scheme can support the current simple escape scenario, but it does not support the scenario where multiple closures share the same local variable. For example, the following sample code:

```lua
local function foo()
    local i, ip, ic = 0, 0, 0
    local function producer()
        i = i + 1
        ip = ip + 1
    end
    local function consumer()
        i = i - 1
        ic = ic + 1
    end
    return produce, consume
end
```

The two internal functions returned by the above `foo()` function both refer to the local variable `i`, and it is obvious that the two functions share `i` and operate on the same `i` instead of being independent `i`. Then when the `foo()` function finishes closing `i`, two functions are needed to share the closed `i`. Since these two functions have different upvalue lists, namely `i, ip` and `i, ic`, the two functions do not need to share the same upvalue list. Then it can only be shared separately for each upvalue.

The following figure shows the scheme of sharing each upvalue separately:

```
     |          |
     +----------+     +=======+
base |    foo   |     |Open(0)|<===============+------------\
     +----------+     +=======+                |            |
   0 |    i     |<- -/  +=======+              |            |
     +----------+       |Open(1)|<-------------|---\        |
   1 |    ip    |<- - - +=======+              |   |        |
     +----------+         +=======+            |   |        |
   2 |    ic    |<- - - - |Open(2)|<-----------|---|--------|---\
     +----------+         +=======+          *-+-+-+-+--    |   |
   3 | producer +---->+-FuncProto--------+   | i |ip |      |   |
     +----------+     |byte_codes:       |   *-^-+-^-+--    |   |
   4 | consumer +--\  | GetUpvalue(0, 0)-+----/    |        |   |
     +----------+  |  | ...              |         |        |   |
     |          |  |  | GetUpvalue(0, 1)-+---------/        |   |
                   |  | ...              |                  |   |
                   |  +------------------+                  |   |
                   |                                      *-+-+-+-+--
                   \-------------->+-FuncProto--------+    | i |ic |
                                  |byte_codes:       |    *-^-+-^-+--
                                  | GetUpvalue(0, 0)-+-----/    |
                                  | ...              |          |
                                  | GetUpvalue(0, 1)-+----------/
                                  | ...              |
                                  +------------------+
```

The picture above is slightly more complicated, but most of it is the same as the previous scheme. The leftmost is still the stack. Then see that the content pointed to by the `producer()` function is still the function prototype and the corresponding upvalue list. Since this function uses two upvalues, two bytecodes are listed. Then there is a difference: in the upvalue list, it is not directly the upvalue, but the address of the upvalue. The real upvalue is allocated on the heap alone, which is `Open(0)`, `Open(1)` and `Open(2)` in the figure. These 3 upvalues can access local variables on the stack through indexes. The last `consumer()` function is similar, the difference is that different upvalues are referenced.

When the `foo()` function ends and all local variables referenced by upvalue are closed, `Open(0)`, `Open(1)` and `Open(2)` in the above figure are replaced by `Closed(i)`, `Closed(ip)` and `Closed(ic)`. At this time, the `i` in the upvalue lists corresponding to `producer()` and `consumer()` functions point to the same `Closed(i)`. In this way, after the outer `foo()` function exits, these two functions can still access the same `i`. Only 3 upvalues are replaced, the changes are relatively small, and the closed picture is omitted here.

Definition of Closure

Before continuing to introduce more upvalue usage scenarios, we first introduce the concept of closure based on the above scheme.

According to the above scheme, the returned `retf` is not only a function prototype, but also includes the corresponding upvalue list. And the function prototype plus upvalue is *closure*! Add Lua closure type in `Value`:

```rust
pub enum upvalue { // upvalue intermediary in the above figure
     Open(usize),
     Closed(Value),
}
pub struct LuaClosure {
     proto: Rc<FuncProto>,
     upvalues: Vec<Rc<RefCell<upvalue>>>,
}
pub enum Value {
     LuaFunction(Rc<FuncProto>), // Lua function
     LuaClosure(Rc<LuaClosure>), // Lua closure
```

In this way, although the different closures returned by multiple calls to the `newCounter()` function share the same function prototype, each has an independent upvalue. This is also the reason why the two counters c1 and c2 at the beginning of this section can count independently.

The following figure shows a schematic diagram of two counters:

```
             +-LuaClosure--+
|      |     |       proto-+-----------------------+-->+-FuncProto--------+
+------+     |    upvalues-+--->+---+--            |   |byte_codes:       |
|  c1  +---->+-------------+    | i |              |   | GetUpvalue(0, 0) |
+------+                        +-+-+--            |   | ...              |
|  c2  +-\                        |                |   +------------------+
+------+ |                        V=========+      |
|      | |                        |Closed(i)|      |
         |                        +=========+      |
         \-->+-LuaClosure--+                       |
             |       proto-+-----------------------/
             |    upvalues-+--->+---+--
             +-------------+    | i |
                                +-+-+--
                                  |
                                  V=========+
                                  |Closed(i)|
                                  +=========+
```

Similarly, we also modify the schematic diagram in the shared upvalue example above. For clarity, delete the specific content of `FuncProto`; then merge the function prototype and upvalue list into `LuaClosure`. As shown below.

```
     |          |
     +----------+     +=======+
base |    foo   |     |Open(0)|<========+----------\
     +----------+     +=======+         |          |
   0 |    i     |<- -/  +=======+       |          |
     +----------+       |Open(1)|<------|---\      |
   1 |    ip    |<- - - +=======+       |   |      |
     +----------+         +=======+     |   |      |
   2 |    ic    |<- - - - |Open(2)|<----|---|------|---\
     +----------+         +=======+     |   |      |   |
   3 | producer +---->+-LuaClosure--+   |   |      |   |
     +----------+     | proto       |   |   |      |   |
   4 | consumer +--\  | upvalues   -+>*-+-+-+--    |   |
     +----------+  |  +-------------+ | i |ip |    |   |
     |          |  |                  *---+---+--  |   |
                   |                               |   |
                   \------------>+-LuaClosure--+   |   |
                                 | proto       |   |   |
                                 | upvalues   -+>*-+-+-+--
                                 +-------------+ | i |ic |
                                                 *---+---+--
```

It can be seen from the figure that compared with the Lua function `LuaFunction` defined in the previous chapter, although the closure `LuaClosure` can have an independent upvalue list, it has one more memory allocation and pointer jump. Here we are faced with a choice: to completely replace the function with the closure, or to coexist? The official implementation of Lua is the former, which is also the source of the phrase "all functions in Lua are closures". The advantage of substitution is that there is one less type, and the code is a little simpler; the advantage of coexistence is that the function type allocates less memory and one pointer jump after all. In addition to these two advantages and disadvantages, there is a larger difference that affects behavior. For example, the following sample code:

```lua
local function foo()
    return function () print "hello, world!" end
end
local f1 = foo()
local f2 = foo()
print(f1 == f2) -- true or false?
```

Here the anonymous function returned by calling `foo()` does not include the upvalue. So the question is, are the two return values of the two calls to `foo()` equal?

- If the `LuaFunction` type is reserved, then the return value is `LuaFunction` type, and `f1` and `f2` only involve the function prototype and are equal. Validation can be performed with the code from the previous chapter.
- If the `LuaFunction` type is not reserved, the returned function is of the `LuaClosure` type. Although it does not contain upvalue, it is also two different closures, `f1` and `f2` are not equal.

So which of the above behaviors meets the requirements of Lua language? The answer is: both can. The description of function comparison in the Lua manual is as follows:

> Functions created at different times but with no detectable differences may be classified as equal or not (depending on internal caching details).

That is, it doesn't matter, and there is no guarantee on it. Then we can choose whatever we want. In this project, we initially chose closures instead of functions, and later added function types back. I don't feel much difference.

Syntax Snalysis of Closure

When there was no closure before and it was still LuaFunction, the processing of function definition was very intuitive:

- Parse the function definition and generate the function prototype FuncProto;
- Wrap FuncProto with `Value::LuaFunction` and put it in the constant table;
- Generate bytecodes such as `LoadConst` to read the constant table.

Function definitions are treated in a similar way to other types of constants. Recall that the relevant code is as follows:

```
    fn funcbody(&mut self, with_self: bool) -> ExpDesc {
        // omit preparation

        // The proto returned by the chunk() function is the FuncProto type
        let proto = chunk(self. lex, has_varargs, params, Token::End);
        ExpDesc::Function(Value::LuaFunction(Rc::new(proto)))
    }
    fn discharge(&mut self, dst: usize, desc: ExpDesc) {
        let code = match desc {
            // omit other types

            // Add the function reason to the constant table and generate
LoadConst bytecode
            ExpDesc::Function(f) => ByteCode::LoadConst(dst as u8, self.
add_const(f) as u16),
```

Now in order to support closures, the following improvements need to be made:

- The relevant Value type definition has been changed to `LuaClosure(Rc<LuaClosure>)` , so the parsed function prototype FuncProto cannot be directly put into the constant table. Although it can be placed indirectly, it is not intuitive. It is better to add a new table in the function prototype FuncProto to save the prototype list of the inner function.

- When the virtual machine executes the function definition, an upvalue is generated in addition to the function prototype. Then the bytecode that directly reads the constant table like `LoadConst` does not meet the demand. A special bytecode needs to be added to aggregate the function prototype and the generated upvalue into a closure.

- In addition, when generating upvalue, we need to know which local variables of the upper function are used by this function. Therefore, the function prototype also needs to add a list of upvalue references to upper-level local indexes.

In summary, the newly added bytecode for creating a closure is as follows:

```
pub enum ByteCode {
    Closure(u8, u16),
```

The two parameters associated with this bytecode are similar to the `LoadConst` bytecode, which are the target address on the stack and the index of the internal function prototype list `inner_funcs` .

In addition, two new members need to be added to the function prototype as follows:

```
pub struct FuncProto {
    pub upindexes: Vec<usize>,
    pub inner_funcs: Vec<Rc<FuncProto>>,
```

where `inner_funcs` is a list of prototypes of the inner functions defined inside the function. `upindexes` is the index of the local variable that the current function refers to the upper function, and this member needs to be modified later. It should be noted that `inner_funcs` is used when the current function acts as an outer function, and `upindexes` is used when the current function acts as a inner function.

After we introduce the complete features of upvalue later, we will introduce the analysis of the upvalue index `upindexes`.

Now, after introducing the definition and syntax analysis of closures, let's look at other scenarios of upvalue.

Improvement: References to Upvalue

The upvalues introduced before are all references to the *local variables* of the upper-level functions. Now let’s look at the references to the *upvalues* of the upper-level functions. Make a modification to the counting closure example at the beginning of this section, and put the incremental code `i = i + 1` into a layer of functions:

```lua
local function newCounter()
     local i = 0
     return function ()
         print(i) -- upvalue
         local function increase()
             i = i + 1 -- where does `i` refer?
         end
         increase()
     end
end

local c1 = newCounter()
c1()
```

In this example, the `i` in the first line of the print statement of the anonymous function returned by the `newCounter()` function is the ordinary upvalue introduced before, pointing to the local variable of the upper-level function. And what is `i` in the internal function `increase()` function? Also upvalue. Who is this upvalue a reference to?

Can it be regarded as a *cross-layer* reference to the local variable `i` in the outermost `newCounter()` function? No, because it cannot be realized when the virtual machine is executed. When the anonymous function returns, the internal `increase()` function has not been created; only when the anonymous function is called outside, the internal `increase()` function will be created and executed; at this time the outermost `newCounter()` has ended, and the local variable `i` no longer exists, so it cannot be referenced.

Since it cannot be a *cross-layer* reference to the *local variable* `i` in the outermost `newCounter()` function, it can only be a reference to the *upvalue* `i` in the anonymous function of the outer layer.

In order to support references to upvalue, first, modify the definition of the upvalue list in `FuncProto` just now, from only supporting local variables to also supporting upvalue:

```rust
pub enum UpIndex {
    Local(usize), // index of local variables in upper functions
    Upvalue(usize), // index of upvalues in upper functions
}

pub struct FuncProto {
    pub upindexes: Vec<UpIndex>, // change from usize to UpIndex
    pub inner_funcs: Vec<Rc<FuncProto>>,
```

Then, look at the schematic diagram of calling the internal `increase()` function when executing the returned anonymous function counter `c1` in the above example:

```
|          |
+----------+
|    c1    +-------------------->+-LuaClosure--+
+----------+                     | proto       |
| increase |                     | upvalues    +--->+---+--
+----------+                     +-------------+    | i |
| increase +-->+-LuaClosure--+                      +-+-+--
+----------+   | proto       |                        |
|          |   | upvalues    +--->+---+--             |
               +-------------+    | i |               |
                                  +-+-+--             V=========+
                                    \---------------->|Closed(i)|
                                                      +=========+
```

On the left is the stack. Among them, `c1` is the function call entry, and the corresponding closureThe upvalue `i` contained in the package is referenced in the print statement.

The first `increase` below the stack is a local variable in `c1`. The second `increase` is the function call entry, and the upvalue `i` contained in the corresponding closure is referenced in the statement that performs the increment operation. In the function prototype, this upvalue should correspond to the 0th upvalue of the upper-level function, namely `UpIndex::upvalue(0)`, so when the virtual machine executes and generates this closure, this upvalue points to the 0th of `c1` upvalue, which is `Closed(i)` in the figure. In this way, the increment operation of `i` in this function will also be reflected in the print statement of `c1` function.

Improvement: References Across Multiple Layer Functions

Let's look at another scenario: cross-layer references. Slightly modifying the above use case to put the `print` statement after the increment operation, we get the following sample code:

```lua
local function newCounter()
    local i = 0
    return function ()
        local function increase()
            i = i + 1 -- upvalue of upper-upper local
        end
        increase()
        print(i) -- upvalue
    end
end
```

The difference between this example and the above example is that when the `increase()` function is parsed, the upvalue `i` has not been generated in the anonymous function to be returned, so `i` in the `increase()` function points to who? Summarize the previous upvalue types: either it refers to the local variable of the upper-level function, or the upvalue of the upper-level function, and analyzes that it cannot be referenced across multiple layers of functions. Therefore, there is only one solution: create an upvalue in the middle layer function. This upvalue is not used in the current function (by now), it is only used to reference the inner function.

The current function does not use the created upvalue "by now". But in the subsequent analysis process, it may still be used. For example, after the above example, the following `print` statement uses this upvalue.

In this example, the prototypes and schematic diagrams of the two functions are the same as the above example. omitted here.

At this point, all the upvalue features are finally introduced, and the final solution is given. During this period, syntax analysis and virtual machine execution are also involved. Next, according to the final plan, we will briefly organize syntax analysis and virtual machine execution.

Syntax Analysis of Upvalue Index

When introducing Syntax Analysis of Closures, it is pointed out that in the function prototype `FuncProto`, a new member `upindexes` needs to be added to represent the upvalue index of the current function.

In the previous section, the variable parsing process is listed:

1. Match in the local variable list of the current function, if found, it is a local variable;
2. Match in the local variable list of upper-level functions, if found, it will be upvalue;
3. Otherwise it is a global variable.

According to the introduction of the complete features of upvalue earlier in this section, the above-mentioned step 2 is extended to the more detailed analysis steps of the upvalue index. The final process of variable analysis is as follows:

1. Match in the local variable list of the current function, if found, it is a local variable;
2. Match in the upvalue list of the current function, if found, the upvalue already exists; (reuse upvalue)
3. Match in the local variable list of outer functions, if found, add an upvalue; (ordinary upvalue)
4. Match in the upvalue list of the outer function, if found, add an upvalue; (reference to the upvalue in the upper function)
5. Match in the local variable list of the outer functions, if found, create an upvalue in all intermediate layer functions, and add an upvalue; (references across multi-layer functions)
6. Match in the upvalue list of the outer functions, if found, create an upvalue in all intermediate layer functions, and add an upvalue; (a reference to an upvalue across multi-layer functions)
7. Repeat steps 5 and 6 above, if the outermost function is still not matched, it is a global variable.

There is obviously a lot of duplication in this process. The most obvious is that steps 3 and 4 are special cases of steps 5 and 6, that is, there is no intermediate layer function, so steps 3 and 4 can be removed. In addition, when the code is implemented, steps 1 and 2 can also be omitted as special cases. Since there is too much content in this section, the specific code will not be posted here.

In the syntax analysis in the previous section, in order to support upvalue, it is necessary to access the local variable list of the upper-level function, so the new context `ParseContext` data structure is added, which contains the local variable list of functions at all levels. This section introduces that upvalue can also refer to the upvalue of upper-level functions, so it is also necessary to add the upvalue list of functions at all levels in `ParseContext`.

```
struct ParseContext<R: Read> {
    all_locals: Vec<Vec<String>>,
    all_upvalues: Vec<Vec<(String, UpIndex)>>, // new
    lex: Lex<R>,
}

pub struct FuncProto {
    pub upindexes: Vec<UpIndex>,
```

In the above code, `ParseContext` is the parsing context, which is the internal data structure of parsing. The member type of its upvalue list `all_upvalues` is `(String, UpIndex)`, where String is the name of the upvalue variable, which is used for the matching in step 4 and step 6; `UpIndex` is the index of upvalue.

`FuncProto` is the output of the syntax analysis stage, which is used by the virtual machine execution stage. At this time, the upvalue variable name is not needed, and only the UpIndex index is needed.

Virtual Machine Execution

In the front part of this section, when introducing the upvalue design scheme, it was basically introduced according to the execution phase of the virtual machine, so we will go through it again here.

First, the closure is created, that is, the function is defined. To this end, a new bytecode `ByteCode::Closure` is introduced, whose responsibility is to generate upvalue, package it together with the function prototype as a closure, and load it on the stack.

What needs to be explained here is that in the syntax analysis phase, in order to access the local variables of the upper-level function, the `ParseContext` context needs to be introduced; however, in the virtual machine execution phase, although upvalue also needs to access the stack space of the upper-level function, it does not need for a similar context. This is because when the closure is created, the upvalue list is generated by the outer function and passed into the closure, and the inner function can indirectly access the stack space of the outer function through the upvalue list.

Besides, in addition to passing the closure into the generated upvalue list, the outer function itself also needs to maintain the list for two purposes:

- As mentioned in the Shared upvalue section above, if a function contains multiple closures, the upvalue of these closures must share local variables. Therefore, when creating an upvalue, first check whether the upvalue associated with this local variable has been created. If so, share; otherwise, create a new one.

  There is a small problem here, the check of whether this has been created is carried out during the virtual machine execution phase. There are generally not many upvalue lists, so it is not necessary to use a hash table. If Vec is used, the time complexity of this matching check is O(n). When there are many upvalues, this may affect performance. Can this matching check be placed in the syntax analysis stage? This issue will be addressed in detail in the next section.

- When the outer function exits, upvalue needs to be closed.

  It should be noted that, theoretically speaking, only escaped upvalues need to be closed; there is no need to close unescaped upvalues. However, it is very difficult to determine whether an upvalue escapes or not at the syntax stage. Because except for the obvious escape case where the internal function is used as the return value in the above example, there are also situations such as assigning the internal function to an external table. It is also very troublesome to judge whether to escape in the virtual machine stage. So for the sake of simplicity, we refer to the official implementation of Lua here, and close all upvalues at the end of the function, regardless of whether they escape.

The timing of closing upvalue is where all functions exit, including `Return`, `Return0` and `TailCall` bytecodes. The specific closing code is omitted here.

Summary

This section introduces the escape of upvalue and adds closure types. But it mainly introduces how to design and manage upvalue, but does not talk about specific operations, including how to create, read, write, and close upvalue. However, after the design plan is explained clearly, these specific operations are relatively simple. This section is already very long, so the introduction and code of this part will be omitted.

Rust DST

Now introduce a feature of the Rust language, DST.

The definition of the closure data structure `LuaClosure` earlier in this section is as follows:

```rust
pub struct LuaClosure {
    proto: Rc<FuncProto>,
    upvalues: Vec<Rc<RefCell<upvalue>>>,
}
```

The function prototype `proto` field is ignored here, and only the upvalue list `upvalues` field is concerned. In order to store any upvalue, the upvalues here are defined as a list Vec. This requires an additional allocation of memory. The memory layout of the entire closure is as follows:

```
+-LuaClosure--+
| proto       |
| upvalues:   |    upvalue list
|   ptr     --+--->+------+------+-
|   capacity  |    |      |      |
|   length    |    +------+------+-
+-------------+
```

In the figure above, the closure `LuaClosure` is on the left, and the extra memory on the right pointed to by `ptr` is the actual storage space of the upvalue list Vec. There are three disadvantages of allocating an additional memory in this way:

- Waste of memory, each segment of memory requires additional management space and waste due to alignment;
- When applying for memory, one more allocation needs to be performed, which affects performance;
- When accessing upvalue, one more pointer jump is required, which also affects performance.

For the requirement of this variable-length array, the classic approach in C language is: define a zero-length array in the data structure, and then specify the actual length as needed when actually allocating memory. The sample code is as follows:

```c
// define the data structure
struct lua_closure {
     struct func_proto *proto;
     int n_upavlue; // actual number
     struct upvalue upvalues[0]; // zero-length array
}

// request memory
struct lua_closure *c = malloc(sizeof(struct lua_closure) // basic space
         + sizeof(struct upvalue) * n_upvalue); // extra space

// initialization
c->n_upvalue = n_upvalue;
for (int i = 0; i < n_upvalue; i++) {
     c->upvalues[i] = ...
}
```

The corresponding memory layout is as follows:

```
+-------------+
| proto       |
| n_upvalue   |
:             : \
:             :  + Upvalue列表
:             : /
+-------------+
```

This approach can avoid the above three disadvantages. Can this be done in Rust? For example, the following definition:

```rust
pub struct LuaClosure {
    proto: Rc<FuncProto>,
    upvalues: [Rc<RefCell<upvalue>>], // slice
}
```

In this definition, the type of upvalues has changed from list `Vec` to slice `[]`. The good news is that Rust supports the DST type (that is, the slice here) as the last field of the data structure, which means that the above definition is legal. The bad news is that such data structures cannot be initialized. A data structure that cannot be initialized, is of course useless. To quote The Rustonomicon: custom DSTs are a largely half-baked feature for now.

We can think about why it cannot be initialized? For example, `Rc` has `Rc::new_uninit_slice()` API to create slices, so can a similar API be added to create this data structure containing slices? In addition, you can also refer to dyn_struct.

However, even if it can be initialized, and the definition of the above data structure can be used, but there will be another problem: since the upvalues field is DST, then the entire `LuaClosure` will also become DST, so the pointer will become a fat pointer, including the actual length of the slice, `Rc<LuaClosure>` becomes 2 words, which in turn causes `enum Value` to change from 2 words to 3 words. This does not meet our requirements, just like `Rc<str>` cannot be used to define the string type before.

Since slice cannot be used, is there any other solution? Fixed-length arrays can be used. For example, modify the definition as follows:

```rust
enum Varupvalues {
    One(Rc<RefCell<upvalue>>), // 1 upvalue
    Two([Rc<RefCell<upvalue>>; 2]), // 2 upvalues
    Three([Rc<RefCell<upvalue>>; 3]), // 3 upvalues
    Four([Rc<RefCell<upvalue>>; 4]), // 4 upvalues
    More(Vec<Rc<RefCell<upvalue>>>), // more upvalue
}

pub struct LuaClosure {
    proto: Rc<FuncProto>,
    upvalues: Varupvalues,
}
```

In this way, for closures with no more than 4 upvalues, additional memory allocation can be avoided. This should satisfy most cases. In the case of more upvalues, the waste of allocating another piece of memory is relatively not that great. Another advantage of this solution is that it does not involve unsafe. Of course, the problem with this solution is that it will bring coding complexity. Since the creation of `LuaClosure` is only generated once when the closure is created, it is not a high-frequency operation, so there is no need to make it so complicated. Therefore, in the end, we still use the original `Vec` solution.

Escape from Block and `goto`

The previous section covered upvalue escapes from functions. But in fact, the scope of local variables is the block, so whenever the block ends, upvalue escape may occur. And a function can also be regarded as a kind of block, so the escape from a function introduced in the previous section can be regarded as a special case of escape from a block.

In addition, there is another escape scenario, that is, the `goto` statement jumps backwards and skips the definition of local variables, and the local variables will also become invalid at this time.

In the previous section, there was too much content, so in order not to add extra details, these two escape scenes are introduced separately in this section.

Escape from block

First look at a sample code that escapes from the block:

```lua
do
    local i = 0
    c1 = function()
        i = i + 1 -- upvalue
        print(i)
    end
end -- the end of the block, the local variable `i` becomes invalid
```

In this example, the anonymous function defined in `do .. end` block refers to the local variable `i` defined in it as upvalue. When the block ends, the local variable `i` will be invalid, but because it is still referenced by the anonymous function, it needs to escape.

Although a function can be regarded as a special case of a block, a special case is a special case after all, and the more general escape from a block is still very different. When the function ends in the previous section, close all upvalues in relevant bytecodes such as `Return/Return0/TailCall`, because each function will have one of these bytecodes at the end. However, there is no similar fixed bytecode at the end of the block, so a new bytecode `Close` is added for this purpose. This bytecode closes the local variable referenced by upvalue in the current block.

The easiest way is to generate a `Close` bytecode at the end of each block, but since the escape from the block is very rare, it's not worth to add a bytecode to all blocks. Therefore, it is necessary to judge whether there is an escape in this block during the syntax analysis stage. If not, there is no need to generate `Close` bytecode.

The next step is how to judge whether there are local variables escaping in a block. There are several possible implementations. For example, refer to the multi-level function nesting method in the previous section, and also maintain a block nesting relationship. However, there is a lighter approach, which is to add a flag bit to each local variable, and set this flag bit if it is referenced by upvalue. Then at the end of the block, judge whether the local variables defined in this block have been marked, and then you can know whether you need to generate `Close` bytecode.

The specific definition and execution flow of `Close` bytecode is omitted here.

Escape from `goto`

I really can't think of an example of a reasonable escape from `goto`. But it is still possible to construct an unreasonable example:

```
::again::
     if c1 then -- false when the first execution reaches here
         c1() -- after assigning a value to c1 below, c1 is a closure that
includes an upvalue
     end

     local i = 0
     c1 = function()
         i = i + 1 -- upvalue
         print(i)
     end
     go to again
```

In the above code, `if` is judged to be false at the first execution, the call to `c1` is skipped; after assigning a value to c1 below, c1 is a closure that includes an upvalue; then goto jumps back to the beginning, and at this time we can call c1; but at this time the local variable `i` is also invalid, so it needs to be closed.

In the above code, from the definition of `again` label at the beginning to the last `goto` statement can also be regarded as a block, then the method of escaping from the block just introduced can be used to process the goto statement. But the goto statement has a special place. We introduced goto statement before, there are two ways to match label and goto statement:

- Match while parsing. That is, when the label is parsed, the goto statement that has already appeared is matched; when the goto statement is parsed, the label that has already appeared is matched;
- After the block ends (that is, when the label defined in it becomes invalid), match the existing label and goto statement at one time.

The implementation difficulty of these two methods is similar, but due to another feature of the goto statement, that is, the goto statement that jumps forward needs to ignore the void statement. In order to process the void statement more conveniently, the second solution above was adopted. However, now to support escapes, when a goto statement is parsed (precisely before the generated `Jump` bytecode), *may* generate a `Close` bytecode. Whether it will be generated or not depends on whether the definition of escaped local variables is skipped when goto jumps backwards. That is, only by matching the label and goto statement can we know whether the `Close` bytecode is required. If we still follow the second scheme to do the matching after the end of the block, even if you we that `Close` needs to be generated at the end of the block, it can no longer be inserted into the bytecode sequence. Therefore, it can only be changed to the first solution of matching while parsing, and judge whether it is necessary to generate `Close` bytecode in time during matching.

Rust Closure

The previous sections introduced closures defined in Lua. In addition, the official implementation of the Lua language also supports C language closures. Our interpreter is implemented by Rust, so it will naturally be changed to Rust closures. This section introduces Rust closures.

C Closures in the Official Implementation of Lua

Let's first look at the C closures in the official implementation of Lua. The C language itself does not support closures, so it must rely on the cooperation of Lua to realize closures. Specifically, the upvalue is stored on the Lua stack, and then bound to the C function prototype to form a C closure. Lua provides a way for C functions to access upvalue on the stack through API.

Here is the C closure version of the counter example code:

```
// counter function prototype
static int counter(Lua_State *L) {
     int i = lua_tointeger(L, lua_upvalueindex(1)); // read upvalue count
     lua_pushinteger(L, ++i); // add 1 and push it to the top of the stack
     lua_copy(L, -1, lua_upvalueindex(1)); // Update the upvalue count with
the new value at the top of the stack
     return 1; // return the count at the top of the stack
}

// factory function, create closure
int new_counter(Lua_State *L) {
     lua_pushinteger(L, 0); // push onto the stack

     // Create a C closure, the function prototype is counter, and also
     // includes 1 upvalue, which is 0 pushed in the previous line.
     lua_pushcclosure(L, &counter, 1);

     // The created C closure is pressed on the top of the stack, and the
     // following return 1 means return to the C closure on the top of the
stack
     return 1;
}
```

Let's look at the second function `new_counter()` first, which is also a factory function for creating closures. First call `lua_pushinteger()` to push the upvalue count to the top of the stack; then call `lua_pushcclosure()` to create a closure. To review, a closure consists of a function prototype and some upvalues, which are specified by the last two parameters of the `lua_pushcclosure()` function. The first parameter specifies the function prototype `counter`, and the second parameter `1` means that the 1 value at the top of the stack is an upvalue, that is, the 0 just pushed. The following figure is a schematic diagram of the stack before and after calling this function to create a C closure:

```
|     |                            |         |
+------+                           +---------+
|  i  +--\  +-C_closure------+<----+ closure |
+------+  |  | proto: counter |     +---------+
|     |  |  | upvalues:      |     |         |
         \--+--> i           |
            +----------------+
```

The far left of the above figure is to push the count i=0 to the top of the stack. In the middle is the created C closure, including the function prototype and upvalue. On the far right is the stack layout after the closure is created, and the closure is pushed onto the stack.

Look at the first function `counter()` in the above code, which is the function prototype of the closure created. This function is relatively simple, the most critical of which is the `lua_upvalueindex()` API, which generates an index representing the upvalue, which can be used to read and write the upvalue encapsulated in the closure.

Through the call flow of the code in the above example to the relevant API, we can basically guess the specific implementation of the C closure. Our Rust closures can also refer to this approach. However, Rust natively supports closures! So we can use this feature to implement Rust closures in Lua more simply.

Rust Closure Definition

To implement the "Rust closure" type in Lua with the closure of the Rust language, it is to create a new Value type including the closure of the Rust language.

"Rust Programming Language" has introduced Rust's closures in detail, so I won't say more here. We just need to know that Rust closures are a trait. Specifically, the Rust closure type in Lua is `FnMut (&mut ExeState) -> i32`. Then you can try to define the Rust closure type of Value in Lua as follows:

```rust
pub enum Value {
    RustFunction(fn (&mut ExeState) -> i32), // normal function
    RustClosure(FnMut (&mut ExeState) -> i32), // Closure
```

However, this definition is illegal, and the compiler will report the following error:

```
error 782| trait objects must include the `dyn` keyword
```

This involves the *static dispatch* and *dynamic dispatch* of traits in Rust. "Rust Programming Language" also has a detailed introduction for this, so I won’t say more here.

Then, we add `dyn` according to the compiler's prompt:

```
pub enum Value {
    RustClosure(dyn FnMut (&mut ExeState) -> i32),
```

The compiler still reports an error, but with a different one:

```
error 277| the size for values of type `(dyn for<'a> FnMut(&'a mut ExeState)
-> i32 + 'static)` cannot be known at compilation time
```

That is to say, the trait object is a DST. This was introduced in Introduction to String Definition before, but what I encountered at that time was slice, and now it is trait, which are also the two most important DSTs in Rust. "Rust Programming Language" also has a detailed introduction for this. The solution is to encapsulate a layer of pointers outside. Since Value supports Clone, `Box` cannot be used, only `Rc` can be used. And because it is `FnMut` instead of `Fn`, it will change the captured environment when it is called, so another layer of `RefCell` is needed to provide internal variability. So the following definition is obtained:

```
pub enum Value {
    RustClosure(Rc<RefCell<dyn FnMut (&mut ExeState) -> i32>>),
```

Finally compiled this time! However, think about why `Rc<str>` was not used when introducing various definitions of strings? Because for the DST type, the actual length needs to be stored in the external pointer or reference, then the pointer will become a "fat pointer", which needs to occupy 2 words. This will further cause the size of the entire Value to become larger. In order to avoid this situation, we can only add another layer of `Box`, let the Box contain a specific length and become a fat pointer, so that `Rc` can restore 1 word. It is defined as follows:

```
pub enum Value {
    RustClosure(Rc<RefCell<Box<dyn FnMut (&mut ExeState) -> i32>>>),
```

After defining the type of Rust closure, I also encountered the same problem as Lua closure: should I keep the type of Rust function? It doesn't make much difference whether to keep it or not. We chose to keep it here.

Virtual Machine Execution

The virtual machine implementation of Rust closures is very simple. Because closures and functions are called in the same way in the Rust language, the invocation of Rust closures is the same as the invocation of previous Rust functions:

```
    fn do_call_function(&mut self, narg_plus: u8) -> usize {
        match self.stack[self.base - 1].clone() {
            Value::RustFunction(f) => { // Rust normal function
                // omit the preparation of parameters
                f(self) as usize
            }
            Value::RustClosure(c) => { // Rust closure
                // Omit the same parameter preparation process
                c.borrow_mut()(self) as usize
            }
```

Test

This completes the Rust closure type. After borrowing the closure of the Rust language itself, this implementation is very simple. There is no need to use the Lua stack to cooperate like the official Lua implementation, and there is no need to introduce some special APIs.

The following code shows how the counter example at the beginning of this section can be implemented using Rust closures:

```
fn test_new_counter(state: &mut ExeState) -> i32 {
    let mut i = 0_i32;
    let c = move |_: &mut ExeState| {
        i += 1;
        println!("counter: {i}");
        0
    };
    state.push(Value::RustClosure(Rc::new(RefCell::new(Box::new(c)))));
    1
}
```

Compared with the C closure at the beginning of this section, this version is clearer except for the last statement to create a closure, which is very verbose. The last statement will also be optimized when the interpreter API is sorted out later.

Limitations of Rust Closures

As you can see from the sample code above, the captured environment (or upvalue) `i` needs to be moved into the closure. This also leads to the fact that upvalues cannot be shared among multiple closures. Lua's official C closure does not support sharing, so there is no problem.

Another point that needs to be explained is that Lua's official C closure uses Lua's stack to store upvalue, which leads to the type of upvalue being Lua's Value type. And we use the closure of Rust language, then upvalue can be "more" types, not limited to the Value type. However, the two should be functionally equivalent:

- The "more" types supported by Rust closures can be implemented in Lua with LightUserData, that is, pointers; although this is very unsafe for Rust.
- The internal types supported in Lua, such as Table, can also be obtained through the API `get()` in our interpreter (and In Lua's official implementation, the table type is internal and not external).

generic-for Statement

Closures were introduced in the previous sections of this chapter. The most typical application scenario of a closure is an iterator, and the most common place for an iterator is a `for` statement. So much so that "Lua Programming" and "Rust Programming Language" both put Closures, iterators, and the `for` statement introduced together. The `for` statement in the Lua language has two formats, numeric and generic. Previously has introduced the numerical-for statement, this section introduces the generic-for statement using iterators.

After introducing closures, the concept of an iterator itself is straightforward. The counter closure that has been used as an example in the previous sections of this chapter can be regarded as an iterator, which generates an incremented number each time. Let's look at a slightly more complex iterator to traverse the array part of a table. This is also the function of the `ipairs()` function that comes with the Lua language:

```lua
function ipairs(t)
     local i = 0
     return function ()
         i = i + 1
         local v = t[i]
         if v then
             return i, v
         end
     end
end
```

In the above code, `ipairs()` is a factory function that creates and returns a closure as an iterator. This iterator has 2 upvalues, one is the fixed table `t`, and the other is the traversed position `i`. We can call these two upvalues as "iteration environments". During the traversal process, the iterator returns the index and value of the array; when the traversal ends, it does not return a value, and it can also be considered as returning `nil`.

This iterator can be called directly, but is more commonly used in a generic-for statement:

```lua
-- call the iterator directly
local iter = ipairs(t)
while true do
     local i, v = iter()
     if not i then break end
     block -- do something
end

-- used in a generic-for statement
for i, v in ipairs(t) do
     block -- do something
end
```

The use of iterators is certainly very convenient, but the previous sections also introduced that creating a closure requires additional overhead compared to creating a normal function, that is, both Lua closures and Rust closures require 2 extra times Memory allocation and 2 pointer jumps. Therefore, the generic for statement in the Lua language is specially optimized for this purpose, that is, the generic for statement itself replaces the closure to save the "iteration environment" (that is, 2 upvalues). Since there is no need for upvalue, the iterator does not need to use closures, but only ordinary functions.

Specifically, the grammar of the generic for statement is as follows:

```
stat ::= for namelist in explist do block end
namelist ::= Name {‘,’ Name}
```

Its execution process is as follows:

- At the beginning of the loop, `explist` is evaluated to get 3 values: the iteration function, the immutable state, and the control variable. In most cases, `explist` is a function call statement, so the evaluation follows the evaluation rules of the function return value, that is, if there are less than 3, it will be filled with nil, and if it exceeds 3, the excess will be discarded. Of course, instead of using a function call, you can directly list 3 values.

- Then, before each execution of the loop, use the latter two values (immutable state and control variables) as parameters to call the iteration function, and judge the first return value: if it is nil, terminate the loop; otherwise, assign the return value to `namelist`, and additionally assign the first return value to the control variable as a parameter for subsequent calls to the iteration function.

It can be seen that the three values returned by `explist` are put together to form a closure function: the iteration function is the function prototype, and the latter two are upvalues. It's just that the generic-for statement helps maintain these two upvalues. Using this feature of the generic-for statement, reimplement the above iterator for traversing the array as follows:

```lua
local function iter(t, i) -- both t and i are changed from upvalue to
parameter
     i = i + 1
     local v = t[i]
     if v then
         return i, v
     end
end

function ipairs(t)
     return iter, t, 0
end
```

Compared with the closure version above, here both `t` and `i` have changed from upvalue to parameters, and `iter` has become an ordinary function.

From this point of view, the generic-for statement can be completed without the need for a closure (such as after the function was introduced in the previous chapter). But after all, this is an optimization based on closures. Only by mastering closures can we understand why we do this. That's why we implement the generic-for statement after introducing closures.

In addition, the function call statement `ipairs(t)` here only returns 3 variables, and these 3 variables can also be listed directly in the generic for statement:

```lua
for i, v in ipairs(t) do ... end
for i, v in iter, t, 0 do ... end -- directly list 3 variables
```

The following direct list method omits a function call, but it is inconvenient. So the first one is more common.

After introducing the characteristics of the generic-for statement in Lua, let's start to implement it.

Accomplish

According to the above introduction, the generic-for statement saves and maintains the iteration environment by itself. Where is that saved? Naturally, it is still on the stack. Just like the numerical-for statement will automatically create 3 variables (1 count variable and 2 anonymous variables) on the stack, the generic-for statement also needs to automatically create 3 anonymous variables, corresponding to the above iteration environment: iteration function, immutable state, control variables. These 3 variables are obtained after evaluating `explist`, as shown in the stack diagram shown in the left figure below:

```
|           |        |           |     |           |
+-----------+        +-----------+     +-----------+
| iter func |entry   | iter func |     | iter func |
+-----------+        +-----------+     +-----------+
| state     |\       | state     |     | state     |
+-----------+ 2args  +-----------+     +-----------+
| ctrl var  |/       | ctrl var  |     | ctrl var  |<--first return value
+-----------+        +-----------+   ->+-----------+
|           |        :           :  /  | name-     | i
                     +-----------+ /   | list      | v
                     | return-   |-    |           |
                     | values    |
                     |           |
```

Next, the loop is executed, including three steps: calling the iterative function, judging whether to continue the loop, and controlling variable assignment.

First, the iteration function `iter func` is called with the immutable state `state` and the control variable `ctrl var` as two parameters. Look at the stack diagram in the left figure, it has just been arranged into a function call formation, so the function can be called directly;

Secondly, after calling the function (the middle picture above), judge whether the first return value is nil, if so, exit the loop; if there is no return value, also exit the loop; otherwise, continue to execute the loop. Before executing the loop body, the return value needs to be processed (right picture above):

- Assign the first return value to the control variable (ctrl-var in the above figure) as the parameter for the next call to the iterative function;
- Assign the return value to the variable list, which is `namelist` in the above BNF. For example, in the above example of traversing the array, it is `i, v`. If the number of return values is less than the variable list, it will be filled with nil. This padding operation is consistent with ordinary function calls. The inconsistency is that the return value of an ordinary function call will be moved to the function entry, which is the position of `iter func` in the above figure; but here it is shifted down by 3 positions.

One thing that needs to be explained here is that the control variable `ctrl-var` is the first name of `namelist`. So in fact, there is no need to reserve a place for `ctrl-var` on the stack; after calling the iterative function each time, just move all the return values directly to the place of `ctrl-var` in the figure, so that the first return value happens to be where `ctrl-var` is. The figure below is a comparison of the two schemes. The left picture is the original plan, which specially reserved the position for `ctrl-var`; the right picture is the new plan, which only needs 2 anonymous variables to save the iteration environment, and `ctrl-var` overlaps with the first name:

```
  |            |       |            |
  +------------+       +------------+
  | iter func  |       | iter func  |
  +------------+       +------------+
  | state      |       | state      |
  +------------+       +------------+
  | ctrl var   |      i| name-      | <--ctrl var
  +------------+      v| list       |
 i| name-      |       |            |
 v| list       |
  |            |
```

The solution on the right is simpler, with one less variable assignment. And under normal circumstances, the functions of the two programs are the same. But in one case, the function will be different, that is, when the control variable is modified in the loop body. For example, the following sample code:

```
for i, v in ipairs(t) do
    i = i + 1 -- modify the control variable `i`
    print(i)
end
```

According to the scheme in the left figure above, the `i` modified here is a variable exposed to programmers, and the control variable `ctrl var` is a hidden anonymous variable, and these two variables are independent. So modifications to `i` do not affect the control variable `ctrl var`. So this loop can still traverse the entire array.

According to the scheme on the right, `i` and `ctrl var` are the same value, and modifying `i` means modifying `ctrl var`, which affects the next iteration function call, and eventually leads to the failure to traverse the entire array normally.

Which behavior is more reasonable? Lua Manual explains: You should not change the value of the control variable during the loop. In other words, there is no clear definition of this behavior, so any solution is fine. However, the official implementation of Lua is based on the behavior in the left picture. In order to maintain consistency, we also choose the left picture here.

Bytecode

The above describes the operation of the generic-for statement in the loop. These operations require a new bytecode `ForCallLoop` to complete.

Before defining this bytecode, let's see where this bytecode should be placed? Is it the beginning of the loop, or the end of the loop? If you follow the Lua code, it should be placed at the beginning of the loop, and then generate a `Jump` bytecode at the end of the loop to jump back and continue the loop, like this:

```
ForCallLoop # If calling the iteration function returns nil, jump to the end
... block ...
Jump (back-to-ForCallLoop)
```

But in this case, 2 bytecodes are executed for each loop, `ForCallLoop` at the beginning and `Jump` at the end. To reduce the bytecode once, we can put `ForCallLoop` at the end of the loop, so that only 2 bytecodes need to be executed in the first loop, and only 1 bytecode needs to be executed in each subsequent loop:

```
Jump (forward-to-ForCallLoop)
... block ...
ForCallLoop # If the return of calling the iteration function is not nil,
then jump to the above block
```

After determining the bytecode location, let's look at the definition. This bytecode needs to be associated with 3 parameters:

- the stack index of the iteration function `iter func`;
- The number of variables used for the assignment of the return value. If the number of return values is less than the number of variables, you need to fill in nil;
- Jump distance.

Both the first two parameters can be represented by 1 byte, so there is only 1 byte left for the final jump distance, which can only represent a distance of 255, which is not enough. For this reason, we can only add a `Jump` bytecode to complete this function. However, in most cases, the loop body is not large, and the distance does not exceed 255. It is a bit wasteful to add another bytecode for a small number of large loop bodies. The best thing to do in this situation is to:

- For the small loop body, the jump distance can be programmed into `ForCallLoop` bytecode, only this 1 bytecode is used;
- For the large loop body, set the jump distance of the third parameter in the `ForCallLoop` bytecode to 0, and add a `Jump` bytecode to cooperate.

In this way, when the virtual machine is executed:

- In the case where the loop needs to jump backwards: for a small loop body, jump directly according to the third parameter; for a large loop body, the third parameter is 0, actually jump to the next Jump bytecode, and then Execute the jump of the loop body again.
- In the case where the loop needs to continue to execute forward: for small loop bodies, no special processing is required; for large loop bodies, the next Jump bytecode needs to be skipped.

In summary, the bytecode `ForCallLoop` is defined as follows:

```rust
pub enum ByteCode {
    ForCallLoop(u8, u8, u8),
```

The specific syntax analysis and virtual machine execution code are omitted here.

At this point, the generic-for statement is completed. We also finished all the syntax of Lua! (Applause please!)

Environment _ENV

Go back to the first "hello, world!" example in Chapter 1. In the output of `luac -l` displayed at that time, the bytecode for reading the global variable `print` is as follows:

```
 2 [1] GETTABUP 0 0 0 ; _ENV "print"
```

Looking at the complex name of the bytecode and the strange `_ENV` comment behind it, it is not simple. At that time, this bytecode was not introduced, but the more intuitive bytecode `GetGlobal` was redefined to read global variables. In this section, let’s supplement the introduction of `_ENV`.

Current Global Variables

Our current handling of global variables is straightforward:

- In the syntax analysis stage, variables that are not local variables and upvalues are considered as global variables, and corresponding bytecodes are generated, including `GetGlobal`, `SetGlobal` and `SetGlobalConst`;
- In the execution phase of the virtual machine, define `global: HashMap<String, Value>` in the execution state `ExeState` data structure to represent the global variable table. Subsequent reads and writes to global variables operate on this table.

This approach is intuitive and has no downsides. However, there is another way to bring more powerful features, which is the environment `_ENV` introduced in Lua version 5.2. "Lua Programming" has a very detailed description of `_ENV`, including why `_ENV` should be used instead of global variables and application scenarios. We will not go into details here, but directly introduce its design and implementation.

How _ENV Works

The principle of `_ENV`:

- In the parsing phase, convert all global variables into indexes of table "_ENV", such as `g1 = g2` to `_ENV.g1 = _ENV.g2`;
- So what is `_ENV` itself? Since all Lua code segments can be considered as a function, `_ENV` can be considered as a local variable outside the code segment, which is

upvalue. For example, for the above code segment `g1 = g2`, a more complete conversion result is as follows:

```
local _ENV = XXX -- predefined global variable table
return function (...)
    _ENV.g1 = _ENV.g2
end
```

All "global variables" have become indexes of `_ENV`, and `_ENV` itself is also an upvalue, so there are no global variables! In addition, the key point is that `_ENV` itself has nothing special except that it is preset in advance, it is just an ordinary variable. This means that it can be manipulated like ordinary variables, which brings a lot of flexibility, such as a sandbox can be easily implemented. The specific usage scenarios will not be expanded here. If you are interested, you can refer to "Lua Programming".

Implementation of _ENV

According to the above introduction, use `_ENV` to transform global variables.

First, in the syntax analysis stage, the global variable is transformed into an index to `_ENV`. The relevant code is as follows:

```
fn simple_name(&mut self, name: String) -> ExpDesc {
    // Omit the matching of local variables and upvalue, and return directly
if they match.

    // If there is no match,
    // - Previously considered to be a global variable, return
ExpDesc::Global(name)
    // - Now transformed into _ENV.name, the code is as follows:
    let env = self.simple_name("_ENV".into()); // call recursively, look for
_ENV
    let ienv = self. discharge_any(env);
    ExpDesc::IndexField(ienv, self. add_const(name))
}
```

In the above code, first try to match the variable `name` from local variables and upvalue. This part was introduced in detail in upvalue and is omitted here. Here we only look at the case where the matching fails. In this case, `name` was previously considered to be a global variable, and `ExpDesc::Global(name)` was returned. Now to transform it into `_ENV.name`, it is necessary to locate `_ENV` first. Since `_ENV` is also an ordinary variable, the `simple_name()` function is called recursively with `_ENV` as an argument. In order to ensure that this call does not recurse infinitely, it is necessary to pre-set `_ENV` in the preparation phase of syntax analysis. So in this recursive call, `_ENV` will definitely be matched as a local variable or upvalue, and will not be called recursively again.

So how to pre-set `_ENV` ? In the above introduction, `_ENV` is the upvalue as the whole code block. But for the sake of convenience, we can use `_ENV` as a parameter in the `load()` function to achieve the same effect:

```
pub fn load(input: impl Read) -> FuncProto {
    let mut ctx = ParseContext { /* omitted */ };

    // _ENV as the first and only parameter
    chunk(&mut ctx, false, vec!["_ENV".into()], Token::Eos)
}
```

In this way, when parsing the outermost code of the code block, when the `simple_name()` function is called, a local variable of `_ENV` will be matched; and an upvalue of `_ENV` will be matched for the code inside the function.

This is just a promise that there must be a `_ENV` variable. And the fulfillment of this promise needs to be performed in the virtual machine execution stage. When creating an execution state `ExeState` , immediately after the function entry, `_ENV` must be pushed onto the stack as the first parameter. In fact, the previous initialization of the `global` member in `ExeState` is transferred to the stack. code show as below:

```
impl ExeState {
    pub fn new() -> Self {
        // global variable table
        let mut env = Table::new(0, 0);
        env.map.insert("print".into(), Value::RustFunction(lib_print));
        env.map.insert("type".into(), Value::RustFunction(lib_type));
        env.map.insert("ipairs".into(), Value::RustFunction(ipairs));
        env.map.insert("new_counter".into(),
Value::RustFunction(test_new_counter));

        ExeState {
            // Push 2 values on the stack: the virtual function entry, and
the global variable table _ENV
            stack: vec![Value::Nil,
Value::Table(Rc::new(RefCell::new(env)))],
            base: 1, // for entry function
        }
    }
```

In this way, the transformation of `_ENV` is basically completed. This transformation is very simple, but the function it brings is very powerful, so `_ENV` is a very beautiful design.

In addition, since there is no concept of global variables, the previous codes related to global variables, such as `ExpDesc::Global` and the generation and execution of 3 bytecodes related to global variables, can be deleted. Note that no new ExpDesc or bytecode is introduced to implement `_ENV` . But just not yet.

Optimization

Although the above transformation is fully functional, there is a performance problem. Since `_ENV` is upvalue in most cases, for global variables, two bytecodes will be generated in the above `simple_name()` function:

```
Getupvalue ($tmp_table, _ENV) # first load _ENV onto the stack
GetField ($dst, $tmp_table, $key) # before indexing
```

In the original scheme that does not use `_ENV`, only one bytecode `GetGlobal` is needed. This new solution obviously reduces performance. To make up for the performance loss here, it is only necessary to provide bytecodes that can directly index the upvalue table. To do this, add 3 bytecodes:

```
pub enum ByteCode {
     // Deleted 3 old bytecodes that directly manipulate the global variable
table
     // GetGlobal(u8, u8),
     // SetGlobal(u8, u8),
     // SetGlobalConst(u8, u8),

     // Add 3 corresponding bytecodes for operating the upvalue table
     GetUpField(u8, u8, u8),
     SetUpField(u8, u8, u8),
     SetUpFieldConst(u8, u8, u8),
```

Correspondingly, the expression of the upvalue table index should also be increased:

```
enum ExpDesc {
     // deleted global variable
     // Global(usize),

     // Added index to upvalue table
     IndexUpField(usize, usize),
```

The index to the upvalue table here only supports string constants, which is also the scenario for global variables. Although this `IndexUpField` is added for global variable optimization, it can also be applied to ordinary upvalue table indexes. Therefore, in the function of parsing table indexes, `IndexUpField` optimization can also be added. The specific code is omitted here.

After defining `IndexUpField`, the original variable parsing function can be modified:

```rust
fn simple_name(&mut self, name: String) -> ExpDesc {
     // Omit the matching of local variables and upvalue, and return directly
if they match.

     // If there is no match,
     // - Previously considered to be a global variable, return
ExpDesc::Global(name)
     // - Now transformed into _ENV.name, the code is as follows:
     let iname = self. add_const(name);
     match self. simple_name("_ENV".into()) {
         ExpDesc::Local(i) => ExpDesc::IndexField(i, iname),
         ExpDesc::upvalue(i) => ExpDesc::IndexUpField(i, iname), // new
IndexUpField
         _ => panic!("no here"), // because "_ENV" must exist!
     }
}
```

As before, after a variable fails to match both the local variable and the upvalue, it still uses `_ENV` as a parameter to recursively call the `simple_name()` function. But here we know that the result returned by `_ENV` must be a local variable or an upvalue. In these two cases, `ExpDesc::IndexField` and `ExpDesc::IndexUpField` are generated respectively. Then generate the 3 new bytecodes above when reading and writing `ExpDesc::IndexUpField`.

In this way, it is equivalent to replacing `ExpDesc::Global` with `ExpDesc::IndexUpField`. The processing of `ExpDesc::Global` was deleted before, and now it is added back from `ExpDesc::IndexUpField`.

To Be Continued

We have implemented the core features of the Lua interpreter. Still, we're far from our original goal - a complete, performant, production-grade interpreter. I will continue to improve this interpreter, but due to busy work and insufficient spare time, I will suspend this series of articles. Writing articles is more tiring than writing code. Combined with my experience of reading Yu Yuan's "Handwriting Operating System by Yourself" when I was in school, I only followed the first half of the book and practiced it. After mastering the basic development methods and getting started with writing an operating system, the rest is to write it by myself. I think the parts of this series of articles that have been completed so far should also provide an introductory knowledge of implementing a Lua interpreter, and interested readers can implement the remaining parts independently.

Here is a partial list of some unfinished features:

- Metatable is a very important feature of Lua language, providing flexible and powerful features. However, its implementation principle is very simple. It only needs to make an extra layer of judgment when the virtual machine executes the relevant bytecode, and it does not even need to modify the part of syntax analysis. Here is an implementation detail: the garbage collection of our interpreter uses RC, which may cause circular references and lead to memory leaks. A table setting itself as its own metatable is a common circular reference. To avoid circular references in this common scenario, special handling is required for this case.

- UserData, is one of the basic types of Lua. However, we have not yet encountered the need to use UserData. We can implement this type later when we encounter this requirement when implementing the standard library. In the official implementation of Lua, creating a new UserData is to allocate for memory in Lua, and then hand it over to the C function to initialize. However, uninitialized memory is not allowed in Rust, so we have to think about how to create a UserData value.

- LightUserData, also one of the basic types of Lua. It's just a raw pointer, and doesn't need to do anything special about it.

- Error handling. Our current way of handling all errors is to panic, which is not feasible. At least we need to distinguish between expected errors and program bugs. The former may also need to subdivide lexical analysis, syntax analysis, virtual machine execution, Rust API and other types. Error handling is also a feature of the Rust language. It's also a great opportunity to experience Rust's error handling.

- Performance optimization. High performance is one of our initial goals, and some optimizations have been made during the implementation, such as the design of the string type, but the final result is not yet known, and we still need to test to know. There are some benchmark examples codes of Lua performance tests on the Internet, we can follow the Lua official Implement a comparative test. This also verifies correctness by the way.

- Optimized table construction. For the table construction with all constant elements, there is no need to load it on the stack, and even the table can be created directly in the syntax analysis stage.
- Rust API. The more usage scenarios of the Lua language are glue languages, so the external API is very important. Our interpreter is mainly used for programs written in Rust language, so it should provide a set of APIs that conform to the Rust calling method. This is inconsistent with the C API provided by the official Lua implementation. We have already implemented some basic APIs, such as reading values on the stack, etc., using generics, which simplifies the API and calling methods, which is inconsistent with the C API. Here has a comparative survey of the calling methods of the scripting language implemented by Rust.
- Library. The current interpreter is a stand-alone program, but the most common usage scenario for Lua is a library that is called by other programs. So we need to transform our project into a library.
- Support parameter passing and return value of the entire code segment.
- The standard library, which is a feature other than the core of the interpreter, involves more aspects. In addition to the packages listed below, there are some basic functions in the standard library, such as `type()` and `ipairs()`, which we have already implemented, and most of the rest are not difficult. The only trouble is `pairs()` function. The efficient implementation of the `pairs()` function in the official Lua implementation depends on the implementation of the table. And we use Rust's `HashMap` to implement the dictionary part of the table, there may be no simple way to implement it.
- The math library, most functions have corresponding implementations in the Rust standard library, the only thing that needs to be manually implemented is the function to generate random numbers. Since this function is not provided in the C language standard, Lua's official implementation makes this function itself. Although we can also use the `random` crate, it is better to refer to the official Lua implementation and implement this random number generation function by ourselves. In addition, generating random numbers requires maintaining a global state. In the official implementation of Lua, this state is a UserData type and is added to Lua's Register. And we can use the characteristics of Rust closures to put this state in closures, which is more convenient and efficient.
- The string library, the trouble is regular matching. For convenience, Lua language defines and implements a set of regular matching rules. So we can only follow its definition and reimplement it in Rust. It should be very complicated here, but after completion, we will have a deeper understanding of regular matching.

- The io library, the trouble is the representation of the file. The `FILE` type is provided in the C language standard, which can represent all file types, including standard input and output, ordinary files, etc., and can also represent multiple modes such as read-only, write-only, and read-write. But in the Rust language these seem to be independent. If we want to provide an API consistent with the io library, we need to do encapsulation.
- The coroutine library, requires a thorough understanding of Lua's coroutines, and will also make great adjustments to the existing function call process.
- The debug library, I have not used this library, I don't know much, but I feel that if to implement this library I will either need a lot of unsafe code, or make a lot of changes to the existing process. So in the end one may choose not to implement this library.

In addition to the above list of unfinished functions, there are some small improvements to the current code, such as refining the comments, applying the `let..else` syntax supported in the new version of Rust, and some small code optimizations, etc. For this reason, we add to_be_continued, which can also be seen as the final version of the code corresponding to this series of articles.

References

- Lua 5.4 Reference Manual, which is also the requirements document for this project.
- "Lua Programming (4th Edition)", the official Lua tutorial. Although it is based on the Lua 5.3 version, it has little impact due to the not many changes of the 5.4 version.
- Why is there no continue statement?, an explanation of why there is no continue statement in Lua.
- 《Rust Programming Language》, the official Rust tutorial.
- Official Rust Documentation, mainly refer to the standard library part.
- Designing a GC in Rust, introduces the design idea of implementing GC in Rust.
- gc-crate, an implementation based on the above design ideas.
- A Tour of Safe Tracing GC Designs in Rust , introducing a GC design implemented in Rust. I just remember one thing: implementing GC in Rust is hard.
- Implementing a safe garbage collector in Rust, another project that uses Rust to implement GC.
- When Zig is safer and faster than Rust, starting from Roc language using Zig instead of Rust to implement the GC part, to illustrate the use It is difficult to implement certain functions in unsafe Rust.
- Luster, a Lua interpreter implemented in Rust, also uses GC instead of RC, but the project is not completed.
- The Story of Tail Call Optimizations in Rust, a discussion of tail call support in the Rust language.
- Lua bindings: lua, hlua or rlua?, there are three existing Lua crates on Reddit: lua, hlua and A simple comparison of rlua.
- A Survey of Rust Embeddable Scripting Languages, for several that can be used in Rust A comparison of the usage of different scripting languages (including Lua).
- Implement TryFrom for float to integer types.
- Floating Point Arcade, an introduction to converting integer random numbers to floating point numbers.
- The Rust Performance Book.
- 《Lua设计与实现》, it feels like a source code reading note of Lua's official

implementation. It directly talks about the details of the code implementation, and it is very difficult to read when you first get started.

- 《自己动手实现Lua》, which is very similar to this series of articles, also implements a Lua interpreter from scratch. But this book is based on the bytecode definition in the official implementation of Lua as the starting point. First implement the virtual machine to execute the bytecode, and then implement the compiler to generate the bytecode. And our series of articles is based on the Lua language manual, designing and implementing the compilation process, virtual machine, bytecode definition, etc.