Z编程语言入门

z是一种静态类型的编译编程语言，编译成c, 在编译成 可执行 文件.

z是一种非常简单的语言。

开发: 张运兵

QQ群: 757923010

点击链接加入群聊【z语言】：https://qm.qq.com/q/uhlfpiPrsA

目录

简介

zlang是一种静态类型的编译编程语言，用于构建可维护的软件。

zlang是一种非常简单的语言。

Hello World

```
fn main() {
    println('你好,世界')
}
```

声明函数fn。返回类型在函数名称后面。在这种情况下main不返回任何内容，因此省略了类型。

就像在C和所有相关语言中一样，main是一个入口。

println是为数不多的内置函数之一。它将值打印到标准输出。# 注释

```
// 这是单行注释

/ *这是一个多行注释。
   / *它可以嵌套* /
* /

```# 函数

函数声明

```go
语法一
fn add(int x, int y) int {
    return x + y
}

语法二
fn sub(int x, y) int {
    return x - y
}

语法三
int mul(int x, y) {
    return x * y
}

fn main() {
    println(add(77, 33))
    println(sub(100, 50))
}
```

函数不能重载。这简化了代码并提高了可维护性和可读性。# 变量

1.变量声明

```
fn main() {
    // 变量声明
    name := 'ZH'
    var age = 36
    var large_number = i64(9999999999)

    println(name)
    println(age)
    println(large_number)
}
```

变量的类型是从右侧的值推断出来的。要强制使用其他类型，请使用类型转换：表达式T(v)将值v转换为类型T。

与大多数其他语言不同，Z只允许在函数中定义变量。不允许使用全局（模块级别）变量。Z中没有全局状态。

2.更改变量值

```
fn main() {
    mut age := 20
    println(age)
    age = 21
    println(age)
}
```

在Z中，默认情况下变量是不可变的。

需要更改变量的值

使用 mut name := val

或 var name = val

请注意，:=和之间的区别=

3.错误变量声明

```
fn main() {
    age = 21
}
```

此代码将无法编译，因为age未声明变量。所有变量都需要在Z中声明。# 基本类型

```
bool

string

i8  i16  i32  i64
u8  u16  u32  u64

byte // u8的别名
int  // i32的别名
rune // i32的别名，表示Unicode代码点

f32 f64
```

int它始终是32位整数。# 字符串

```
fn main() {
    name := 'Bob'
    println('Hello, $name!')
    println(name.len)

    bobby := name + 'by' // + 用于连接字符串
    println(bobby) // ==> "Bobby"

    println(bobby.substr(1, 3)) // ==> "ob"
    // println(bobby[1:3]) // 这种语法很可能会替换substr（）方法
}
```

在Z中，字符串是只读字节数组。字符串数据使用UTF-8编码。

字符串是不可变的。这意味着子字符串函数非常有效：不执行复制，不需要额外的分配。

连接运算符+需要在两端都有字符串。如果age是int，则不编译此代码：

```
println('age = ' + age)
```

我们必须将年龄转换为string：

```
println('age = ' + age.str())
```

或使用字符串插值：

```
println('age = $age')
```

或者简单地将第二个参数传递给println：

```
println('age = ', age) // TODO: 还没有实现

```# 数组

```go

fn main() {
    nums := [1, 2, 3]
    println(nums)
    println(nums[1]) // ==> "2"

    mut names := ['John']
    names << 'Peter'
    names << 'Sam'
    // names << 10  <-- 这不会编译。`names`是一个字符串数组。
    println(names.len) // ==> "3"
    println(names.contains('Alex')) // ==> "false"

    // 我们还可以预先分配一定数量的元素。
    nr_ids := 50
    mut ids := [0 ; nr_ids] // 这会创建一个包含50个零的数组
}
```

数组类型由第一个元素决定：[1, 2, 3]是一个整型数组（[]int）。
['a', 'b']是一个字符串数组（[]string）。

数组中的所有元素必须具有相同的类型。[1, 'a']不会编译。

<< 是一个将值追加到数组末尾的运算符。

.len字段返回数组的长度。请注意，它是一个只读字段，用户无法修改。在Z中，默认情况下，所有导出的字段都是只读的。

.contains(val)如果数组包含val，则方法返回true。

Map

```
fn main() {
    mut m := map[string]int{} // 现在只允许带字符串键的map
    m['one'] = 1
    println(m['one']) // ==> "1"
    println(m['bad_key']) // ==> "0"
    // TODO: 实现检查key是否存在的方法

    numbers := { // TODO: 此语法尚未实现
        'one': 1,
        'two': 2,
    }
}

```# If 语句
```
```

```go

fn main() {
    a := 10
    b := 20
    if a < b {
        println('$a < $b')
    } else if a > b {
        println('$a > $b')
    } else {
        println('$a == $b')
    }
}
```

if 语句非常简单，与大多数其他语言类似。

与其它类C语言不同，条件周围没有括号，并且始终需要大括号。

if 可以用作表达式：

```
num := 777
s := if num % 2 == 0 {
    'even'
}
else {
    'odd'
}
println(s) // ==> "even"

```# For 循环

Z只有一个循环结构：for。

```go

fn main() {
    numbers := [1, 2, 3, 4, 5]
    for num in numbers {
        println(num)
    }
    names := ['Sam', 'Peter']
    for i, name in names {
        println('$i) $name')  // 输出: 0) Sam
    }                         //         1) Peter
}
```

这个for .. in循环用于遍历数组元素。如果需要索引，则可以使用for index, value in来替代。

```
fn main() {
    mut sum := 0
    mut i := 0
    for i <= 100 {
    sum += i
        i++
    }
    println(sum) // ==> "5050"
}
```

这种循环形式类似于其它语言中的while循环。

一旦布尔条件求值为false，循环将停止迭代。

同样，条件周围没有圆括号，并且始终需要大括号。

```
fn main() {
    mut num := 0
    for {
        num++
        if num >= 10 {
            break
        }
    }
    println(num) // ==> "10"
}
```

条件可以省略，这会导致无限循环。

```
fn main() {
    for i := 0; i < 10; i++ {
        println(i)
    }
}
```

最后，还有传统的C风格for循环。它比`while`形式更安全，因为后者很容易忘记更新计数器并陷入无限循环。

这里i不需要mut声明，因为根据定义，它总是可变的。

Match

```
fn main() {
    os := 'windows'
```

```
    print('Z 运行在 ')
    match os {
        'darwin'{
            println('macOS.')
        }
        'linux'{
            println('Linux.')
        }
        default{
            println(os)
        }
    }
}
```

Switch

```
fn main() {
    os := 'windows'
    print('Z is running on ')
    switch os {
    case 'darwin':
        println('macOS.')
    case 'linux':
        println('Linux.')
    default:
        println(os)
    }
    // TODO: 用匹配表达式替换
}
```

switch语句是编写if-else语句序列的较短方式。它运行其值等于条件表达式的第一个情况。

与C不同，break语句不需要出现在每个块的末尾。# 类型(typec)

```
typec Point {
    int x
    int y
}

fn main() {
    p := Point{
        x: 10
        y: 20
    }
    println(p.x) // 使用点访问结构字段

    // &前缀返回一个指向struct值的指针。
    // 它被分配到堆上，并自动清除
    pointer := &Point{x:10, y:10}
```

```
    println(pointer.x) // 指针用于访问字段的语法相同
}
```

结构(struct)

```
struct Point {
    x int
    y int
}

fn main() {
    p := Point{
        x: 10
        y: 20
    }
    println(p.x) // 使用点访问结构字段

    // &前缀返回一个指向struct值的指针。
    // 它被分配到堆上，并自动清除
    // 在函数末尾用v表示，因为它不会被转义。
    pointer := &Point{x:10, y:10}
    println(pointer.x) // 指针用于访问字段的语法相同
}
```

方法(method)

```
class User {
    int age
}

// 语法一
fn (User u) can_register() bool {
    return u.age > 16
}
// 语法二
bool User:can_register() {
    return u.age > 16
}

fn main() {
    user := User{age: 10}
    println (user.can_register()) // ==> "false"

    user2 := User{age: 20}
    println(user2.can_register()) // ==> "true"
}
```

您可以在类型上定义方法。

方法是具有特殊类型参数的函数。

类型名出现在它自己的参数列表中，位于fnc关键字和方法名之间。

在此示例中，该can_register方法具有User名为的u。也可使用类似self或this的名称，而是使用一个短名称，最好是一个字母长的名称。# 类(class)

```
class User {
    int age
    bool can_register() {
        return u.age > 16
    }
}

fn main() {
    user := User{age: 10}
    println (user.can_register()) // ==> "false"

    user2 := User{age: 20}
    println(user2.can_register()) // ==> "true"
}
```

可变接收机和纯变量

```
typec User {
    boolis_registered
}

fn (User mut u) register() {
    u.is_registered = true
}

fn main() {
    mut user := User{}
    println(user.is_registered) // ==> "false"
    user.register()
    // TODO: 也许强制标记方法用`!`
    // user.register()!
    println(user.is_registered) // ==> "true"
}
```

请注意，功能只能修改接收器。
fn register(User mut u) 不会编译。

这非常重要，所以我再说一遍：Z函数是部分纯的，它们的参数永远不会被函数修改。

修改对象的另一种方法是返回修改后的版本：

```
// TODO: 此语法尚未实现
fn register(User u) User {
    return { u | is_registered: true }
}

user = register(user)

```# 常量(const)

```go

const (
    PI = 3.14
    WORLD = '世界'
)

fn main() {
    println(PI)
    println(WORLD)
}
```

常量用const声明。它们只能在模块级别定义（函数之外）。

常量名称大写。这有助于将它们与变量区分开来

永远不能改变常量值。

Z常量比大多数语言更灵活。您可以指定更复杂的值：

```
struct Color {
    r int
    g int
    b int
}

fn (Color c) str() string {
    return '{$c.r, $c.g, $c.b}'
}

fn rgb(int r, g, b ) Color {
    return Color{r: r, g: g, b: b}
}

const (
    NUMBERS = [1, 2, 3]

    RED  = Color{r: 255, g: 0, b: 0}
    BLUE = rgb(0, 0, 255)
)
```

```
fn main() {
    println(NUMBERS)
    println(RED)
    println(BLUE)
}
```

不允许使用全局变量，因此这非常有用。# 模块(module)

Z是一种真正的模块化语言。这非常简单。要创建新模块，请使用代码创建包含模块名称和.z文件的目录：

```
cd ~/code/modules
mkdir mymodule
vim mymodule/mymodule.z
```

```
// mymodule.v
module mymodule

// 要导出函数，我们必须使用`pub`
pub fn say_hi() {
    println('hello from mymodule!')
}
```

您可以根据需要在 mymodule/ 中任意添加多个.z文件。

用 z -lib ~/code/modules/mymodule 构建它。

就是这样，现在您可以在代码中使用它：

```
module main

import mymodule

fn main() {
    mymodule.say_hi()
}
```

接口(interface)

```
struct Dog {}
struct Cat {}
```

```
fn (Dog d) speak() string {
    return 'woof'
}

fn (Cat c) speak() string {
    return 'meow'
}

interface Speaker {
    string speak()
}

fn perform(Speaker s) {
    println(s.speak())
}

fn main() {
    dog := Dog{}
    cat := Cat{}
    perform(dog) // ==> "woof"
    perform(cat) // ==> "meow"
}
```

类型通过实现其方法来实现接口。没有明确的意图声明，没有“implements”关键字。# 枚举(enum)

```
enum Color {
    red, green, blue
}

fn main() {
    mut color := red  // TODO: color := Color.green
    color = green  // TODO: color = .green
    println(color) // ==> "1"  TODO: print "green"?
}
```

选项类型和错误处理

```
typec User {
    int id
}

struct Repo {
    users []User
}

fn new_repo() Repo {
    user := User{id:10}
    return Repo {
        users: [user]
```

```
    }
}

fn (Repo r) find_user_by_id(int id) User? {
    for user in r.users {
        if user.id == id {
            // V自动将其包装为选项类型
            return user
        }
    }
    return error('User $id not found')
}

fn main() {
    repo := new_repo()
    user := repo.find_user_by_id(10) or { // 选项类型必须由`or`块处理
        return  // `or`块必须以`return`，`break`或`continue`结束
    }
    println(user.id) // ==> "10"
}
```

运行

将函数"升级"到可选函数所需的工作量是最小的：您必须添加一个?在返回类型中，并在出现错误时返回错误。

如果您不需要返回错误，只需返回一个None。（TODO：None尚未实现）。

这是处理Z中错误的主要方法。它们仍然是值，就像在Go中一样，但优点是错误无法处理，并且处理它们的冗长要少得多。

您还可以传播错误：

```
resp := http.get(url)?
println(resp.body)
```

http.get返回?http.Response。它被调用?，因此错误会传播到调用函数，或者在主函数导致死机的情况下传播。
基本上，上面的代码是一个较短的版本

```
resp := http.get(url) or {
    panic(err)
}
println(resp.body)

```# 泛型

```go

struct Repo <T> {
    DB db
}
```

```
fn new_repo<T>(db DB) Repo<T> {
    return Repo<T>{db: db}
}

// 这是一个通用函数。V将为它使用的每种类型生成它。
fn (Repo<T> r) find_by_id(int id) T? {
    table_name := T.name // 在此示例中，获取类型的名称会为我们提供表名
    return r.db.query_one<T>('select * from $table_name where id = ?', id)
}

fn main() {
    db := new_db()
    users_repo := new_repo<User>(db)
    posts_repo := new_repo<Post>(db)
    user := users_repo.find_by_id(1)?
    post := posts_repo.find_by_id(1)?
}
```

Z目前不支持泛型

并发

并发模型与Go非常相似。要同时运行foo()，只需使用go foo()调用它。现在它在一个新的系统线程中启动该函数，很快Goroutines和调度程序将被实现。# 解码JSON

```
typec User {
    string name
    int age
}

fn main() {
    data := '{ "name": "Frodo", "age": 25 }'
    user := json.decode(User, data) or {
        eprintln('Failed to decode json')
        return
    }
    println(user.name)
    println(user.age)
}
```

JSON现在非常流行，这就是内置JSON支持的原因。

json.decode函数的第一个参数是要解码的类型。第二个参数是json字符串。

Z生成用于json编码和解码的代码。没有使用反射。这导致更好的性能。# 通过codegen反射

内置JSON支持很不错，但V还允许您为任何事物创建有效的序列化器：

```
// TODO: 计划在五月
fn decode<T>(string data) T {
```

```
        result := T{}
        for field in T.fields {
                if field.typ == 'string' {
                        result.$field = get_string(data, field.name)
                } else if field.typ == 'int' {
                        result.$field = get_int(data, field.name)
                }
        }
        return result
}

// 生成到:
fn decode_User(string data) User {
        result := User{}
        result.name = get_string(data, 'name')
        result.age = get_int(data, 'age')
        return result
}

```# 测试

```go

// hello.v
fn hello() string {
    return 'Hello world'
}

// hello_test.v
fn test_hello() {
    assert hello() == 'Hello world'
}
```

所有测试功能都必须放在*_test.z文件中，并从test开始。要运行测试，请执行vz hello_test.z。要测试整个模块，请执行z test mymodule。# 将C / C ++翻译成Z

Z可以将您的C/C++代码转换为人类可读的z代码。让我们先创建一个简单的程序test.cpp：

```
#include <vector>
#include <string>
#include <iostream>

int main() {
    std::vector<std::string> s;
    s.push_back("V is ");
    s.push_back("awesome");
    std::cout << s.size() << std::endl;
    return 0;
}
```

Run z translate test.cpp and Z will generate test.z:

```
fn main {
    mut s := []string
    s << 'z is '
    s << 'awesome'
    println(s.len)
}
```

Z调用C函数

```
#flag -lsqlite3

#include "sqlite3.h"

struct C.sqlite3
struct C.sqlite3_stmt

fn C.sqlite3_column_int(C.sqlite_stmt, int) int

fn main() {
    path := 'sqlite3_users.db'
    db := &C.sqlite3{}
    C.sqlite3_open(path.cstr(), &db)

    query := 'select count(*) from users'
    stmt := &C.sqlite3_stmt{}
    C.sqlite3_prepare_v2(db, query.cstr(), - 1, &stmt, 0)
    C.sqlite3_step(stmt)
    nr_users := C.sqlite3_column_int(res, 0)
    C.sqlite3_finalize(res)
    println(nr_users)
}

```# 附录I：关键词

Z关键字：
- break
- const
- continue
- defer
- else
- enum
- fn
- new
- for
- go
- goto
- if
- import
- in
- interface
- match
- module

```
- mut
- or
- return
- struct
- typec
- type
```