golang基础

基本形式

```
package main

import "fmt" //导入fmt包，该包提供了与输入输出相关的函数

func main() {//main函数，是Go程序的入口函数
    fmt.Println("Hello world")
}
```

变量声明

`var a = xxx`: 直接定义，var是变量声明，go会自动推断类型

`const b = xxx`: const代替var是常量声明，类似c

go的变量类型一般都是后置的

`var c string = "lll"`: 同时也可以说明是什么变量，类似py的注解

还有一种更简单的声明方式 : `:=`: `:=`是一个短变量声明操作符，用于声明和初始化变量。它可以在不显式指定变量类型的情况下，通过初始化值的类型推导来自动推断变量类型。

对于多个变量的定义，go还有类似py的定义方式: `a, b := 1, "gogo"`

if else语句

~~和C几乎一样~~，但是有些不同点需要注意

0. 格式go有严格要求 `if`和`else`关键字后面必须紧跟空格。(没有也没关系，编译器最后会自动格式化) 条件表达式后面必须有一个空格。 开始的大括号 `{` 必须与 `if` 或 `else` 关键字位于同一行，并且位于条件表达式的末尾。 `else`要紧跟在上一个`}`的后面(同一行)

   ```
   if condition {
       // 执行满足条件时的代码块
   } else {
       // 执行条件不满足时的代码块
   }
   ```

1. 不需要使用括号：在C语言中，条件表达式需要使用括号将其括起来，而在Go语言中，不需要使用括号。写代码时可以加上，但是编译器会自动去掉 例如，在C语言中，if语句可能是这样的：`if (condition) { ... }`，而在Go语言中，是这样的：`if condition { ... }`。

2. 必须使用花括号：在Go语言中，条件语句的代码块必须使用花括号括起来，即使只有一行代码也不能省略。强制保持代码的结构清晰和一致性。

3. 可以在if语句中定义变量：与先前讨论的类似，Go语言允许在if语句中定义并初始化变量。这在一定程度上可以提高代码的简洁性和可读性。 在Go语言中，如果在if语句中定义了变量，则该变量的作用域仅限于if语句的代码块内部。

```
if num := 10; num <= 10 {
    fmt.Println(num) // 可以访问并输出变量num
}

fmt.Println(num) // 编译错误，无法在这里访问变量num
```

4. 布尔类型的直接判断：在Go语言中，条件表达式的结果必须是布尔类型（true或false），而不像C语言那样，可以由非零值和零值来表示真假。

循环

相比于C语言，go有着更加简洁的循环格式，对比如下:

for循环

```
// C语言
for (初始化; 条件; 更新) {
    // 循环体代码
}
```

```
// go
for 初始化; 条件; 更新 {
    // 循环体代码
}
```

while循环

```
while (条件) {
    // 代码
}
```

```
for 条件 {
    // 代码
}
```

通过使用break关键字，go也可以实现类似do-while循环

```
for {
    // 循环体代码
    if !条件 {
        break
    }
}
```

switch语句

go的switch和C语言类似，但是比C更牛逼一些，没有类型要求，不需要加break，就像是简洁版的if-else语句。

数组

1. 声明和初始化数组：
    - 声明数组的语法是 `var 变量名 [长度]元素类型`，例如：

      ```
      var arr [5]int // 声明一个长度为5的int类型数组
      ```

    - 使用初始化表达式进行数组初始化，例如：

      ```
      arr := [5]int{1, 2, 3, 4, 5} // 声明并初始化一个长度为5的int类型数组
      ```

    - 可以使用`[...]`来忽略数组的长度，让编译器根据初始化表达式中的元素个数来确定长度，例如：

      ```
      arr := [...]int{1, 2, 3, 4, 5} // 声明并初始化一个长度为5的int类型数组
      ```

2. 访问数组元素：
    - 使用索引操作符`[]`来访问数组元素，索引从0开始计数，例如：

      ```
      arr := [5]int{1, 2, 3, 4, 5}
      fmt.Println(arr[0]) // 输出第一个元素，即1
      ```

    - 可以通过索引来修改数组元素的值，例如：

      ```
      arr[0] = 10 // 将第一个元素的值修改为10
      fmt.Println(arr[0]) // 输出修改后的值，即10
      ```

3. 数组长度和容量：

   - 数组的长度是在声明时指定的固定值。可以使用`len()`函数获取数组的长度，例如：

```
arr := [5]int{1, 2, 3, 4, 5}
fmt.Println(len(arr)) // 输出数组的长度，即5
```

   - Go语言中的数组是定长的，长度不可更改。因此，不能直接增加或删除数组中的元素。

4. 数组的遍历：

   - 使用`for`循环和索引遍历数组，例如：

```
arr := [5]int{1, 2, 3, 4, 5}
for index, value := range arr {
    fmt.Println(index, value)
}
```

   - 使用`for`循环和常规的`for`索引遍历数组，例如：

```
arr := [5]int{1, 2, 3, 4, 5}
for i := 0; i < len(arr); i++ {
    fmt.Println(i, arr[i])
}
```

切片

切片（slice）是Go语言中一种可变长度的序列类型，可以使用切片来对数组或其他切片进行截取、分割和连接等操作。

1. 创建切片：

   - 使用切片操作符`:`来创建切片，例如：

```
arr := [5]int{1, 2, 3, 4, 5}
slice := arr[1:3] // 创建一个从数组索引1到2的切片，包含元素[2, 3]
```

   - 使用`make()`函数来创建指定长度和容量的切片，例如：

```
slice := make([]int, 5) // 创建一个长度和容量都为5的切片，初始值为默认零值
```

2. 切片的长度和容量：

- 使用`len()`函数获取切片的长度，即切片中元素的个数。
- 使用`cap()`函数获取切片的容量，即底层数组中可用于存储元素的空间大小。

3. 修改切片元素：
   - 可以通过索引来修改切片中的元素值，例如：

```
slice := []int{1, 2, 3, 4, 5}
slice[1] = 10 // 将索引为1的元素值修改为10
```

4. 追加元素到切片：
   - 使用`append()`函数将一个或多个元素追加到切片的末尾，例如：

```
slice := []int{1, 2, 3}
slice = append(slice, 4, 5) // 追加两个元素到切片末尾
```

5. 切片的复制：
   - 使用`copy()`函数将一个切片的内容复制到另一个切片，例如：

```
slice := []int{1, 2, 3}
newSlice := make([]int, len(slice))
copy(newSlice, slice) // 将slice的内容复制到newSlice
```

map

1. 创建和初始化`map`：
   - 使用`make()`函数来创建一个空的`map`，例如：

```
m := make(map[keyType]valueType)
```

   - 使用初始化表达式来创建并初始化`map`，例如：

```
m := map[string]int{
    "a": 1,
    "b": 2,
    "c": 3,
}
```

2. 插入和修改`map`元素：

- 使用键值对语法向map中插入元素，例如：

```
m := make(map[string]int)
m["a"] = 1 // 插入键为"a"，值为1的元素
```

- 通过键来修改map中的元素值，例如：

```
m := map[string]int{
    "a": 1,
    "b": 2,
}
m["a"] = 10 // 修改键为"a"的元素值为10
```

3. 获取和删除map元素：

- 使用键来获取map中的元素值，同时还可检查是否存在该键，例如：

```
m := map[string]int{
    "a": 1,
    "b": 2,
}
value, ok := m["a"] // 获取键为"a"的元素值，并检查是否存在该键
```

- 使用delete()函数按键删除map中的元素，例如：

```
m := map[string]int{
    "a": 1,
    "b": 2,
}
delete(m, "a") // 删除键为"a"的元素
```

4. 遍历map：

- 使用for range语句遍历map的所有键值对，例如：

```
m := map[string]int{
    "a": 1,
    "b": 2,
    "c": 3,
}
for key, value := range m {
    fmt.Println(key, value)
}
```

- 只遍历map的键或值，可以使用下划线 _ 替代不需要的变量，例如：

```
for key, _ := range m {
    fmt.Println(key)
}
```

函数

1. 函数定义：
    - 使用func关键字定义函数。
2. 参数传递：
    - 函数可以接收零个或多个参数。
    - 参数可以是任意数据类型。
    - 参数可以传值或传引用。
    - 示例：

```
func add(a, b int) int {
    return a + b
}
```

3. 返回值：
    - 函数可以有一个或多个返回值。
    - 返回值可以是任意数据类型。
    - 示例：

```
func divide(dividend, divisor float64) (float64, error) {
    if divisor == 0 {
        return 0, errors.New("division by zero")
    }
    return dividend / divisor, nil // nil类似于null
}
```

4. 多返回值：
    - 函数可以返回多个值。
    - 在函数签名中指定返回值的类型。
    - 调用函数时可以接收多个返回值。
    - 示例：

```
func swap(a, b int) (int, int) {
    return b, a
```

```
}
```

5. 可变参数：

- 函数可以接收可变数量的参数。
- 使用...表示可变参数。
- 在函数体内对可变参数进行处理。
- 示例：

```
func sum(nums ...int) int {
    total := 0
    for _, num := range nums {
        total += num
    }
    return total
}
```

6. 匿名函数：

- 在函数内部定义没有函数名的匿名函数。
- 匿名函数可以直接调用，也可以作为值传递给其他函数。
- 示例：

```
func main() {
    add := func(a, b int) int {
        return a + b
    }
    result := add(1, 2)
    fmt.Println(result)
}
```

7. 函数作为参数：

- 函数可以作为参数传递给其他函数。
- 可以在函数中调用传入的函数参数。
- 示例：

```
func process(numbers []int, callback func(int) int) {
    for _, num := range numbers {
        result := callback(num)
        fmt.Println(result)
    }
}

func double(num int) int {
    return num * 2
```

```
}

func main() {
    nums := []int{1, 2, 3, 4, 5}
    process(nums, double)
}
```

8. 递归函数：

    - 函数可以调用自身。
    - 递归函数需要定义终止条件，避免无限递归。
    - 示例：

```
func factorial(n int) int {
    if n <= 1 {
        return 1
    }
    return n * factorial(n-1)
}
```

指针

用法和C基本一样

```
// C
int* ptr; // C语言中的指针声明
int value = 10;
ptr = &value; // 获取变量的地址并赋值给指针
int dereferenced_value = *ptr; // 使用`*`来操作指针，获取指针指向的值
```

```
// go
var ptr *int // Go语言中的指针声明
value := 10
ptr = &value // 获取变量的地址并赋值给指针
dereferenced_value := *ptr
```

结构体

以下是go和c语言的结构体对比

```
// go
// 声明结构体类型
type Person struct {
    name string
```

```
    age  int
}

func main() {
    // 创建结构体变量，并对字段赋值
    p := Person{
        name: "Alice",
        age:  25,
    }

    // 访问结构体的字段
    fmt.Println("Name:", p.name)
    fmt.Println("Age:", p.age)
}
```

```
// C
#include <stdio.h>
#include <string.h>

// 定义结构体类型
typedef struct {
    char name[20];
    int age;
} Person;

int main() {
    // 创建结构体变量，并对字段赋值
    Person p;
    strncpy(p.name, "Alice", sizeof(p.name));
    p.age = 25;

    // 访问结构体的字段
    printf("Name: %s\n", p.name);
    printf("Age: %d\n", p.age);

    return 0;
}
```

结构体方法和函数

结构体方法是与特定的结构体类型关联的，通过结构体实例调用；而函数则是独立存在的，直接用函数名进行调用。

```
package main

import "fmt"

// 定义一个结构体类型
type Rectangle struct {
```

```
    width  float64
    height float64
}

// 结构体方法：计算矩形的面积
func (r Rectangle) Area() float64 {
    return r.width * r.height
}

// 函数：计算两个数的乘积
func Multiply(a, b int) int {
    return a * b
}

func main() {
    // 创建一个矩形实例
    rect := Rectangle{width: 10, height: 5}

    // 调用结构体方法
    fmt.Println("Rectangle Area:", rect.Area())

    // 调用函数
    product := Multiply(3, 4)
    fmt.Println("Product:", product)
}
```

学习过程迷惑了一下，因为对函数和方法的理解还很浅

错误处理

函数可以在需要时返回一个error类型的值，以指示函数执行过程中是否发生了错误。调用者可以根据返回的错误值来决定如何处理。

```
func Divide(a, b float64) (float64, error) {
    if b == 0 {
        return 0, fmt.Errorf("division by zero")
    }
    return a / b, nil
}

func main() {
    result, err := Divide(10, 0)
    if err != nil {
        fmt.Println("Error:", err)
    } else {
        fmt.Println("Result:", result)
    }
}
```

字符串操作

需要`import "strings"`

下面是一些常用操作: 以下是Go语言中常用的字符串操作及其示例:

1. 字符串长度: `len(str)`: 返回字符串的字节数。

```
str := "Hello, World!"
length := len(str)
fmt.Println(length) // Output: 13
```

2. 字符串拼接: 使用`+`运算符或`strings.Join()`函数进行字符串拼接。

```
str1 := "Hello"
str2 := "World!"
result := str1 + ", " + str2
fmt.Println(result) // Output: Hello, World!

strSlice := []string{"Hello", "World!"}
result := strings.Join(strSlice, ", ")
fmt.Println(result) // Output: Hello, World!
```

3. 字符串切片: 使用索引或切片操作提取字符串中的子串。

```
str := "Hello, World!"
// 索引获取单个字符
char := str[0]
fmt.Println(string(char)) // Output: H

// 切片获取子串
substr := str[7:12]
fmt.Println(substr) // Output: World
```

4. 字符串查找和替换: 使用`strings.Contains()`、`strings.Index()`和`strings.Replace()`进行查找和替换。

```
str := "Hello, World!"
contains := strings.Contains(str, "World")
fmt.Println(contains) // Output: true

index := strings.Index(str, "World")
fmt.Println(index) // Output: 7

newStr := strings.Replace(str, "World", "Gopher", -1)
fmt.Println(newStr) // Output: Hello, Gopher!
```

5. 字符串大小写转换： 使用`strings.ToLower()`和`strings.ToUpper()`进行大小写转换。

```
str := "Hello, World!"
lower := strings.ToLower(str)
fmt.Println(lower) // Output: hello, world!

upper := strings.ToUpper(str)
fmt.Println(upper) // Output: HELLO, WORLD!
```

6. 字符串分隔和拼接： 使用`strings.Split()`将字符串分割为多个子串，使用`strings.Join()`将多个子串连接为一个字符串。

```
str := "a,b,c,d"
strSlice := strings.Split(str, ",")
fmt.Println(strSlice) // Output: [a b c d]

joined := strings.Join(strSlice, "-")
fmt.Println(joined) // Output: a-b-c-d
```

字符串格式化

```
import "fmt"

num := 42
fmt.Printf("%v\n", num)
// Output: 42

str := "Hello"
fmt.Printf("%v\n", str)
// Output: Hello
```

类似有 `%+v`：打印结构体时，除了包含结构体字段的值，还会显示字段的名称。

```
import "fmt"

type Person struct {
    Name string
    Age  int
}

person := Person{Name: "Alice", Age: 30}
fmt.Printf("%+v\n", person)
// Output: {Name:Alice Age:30}
```

`%#v`：显示了值的类型和详细的属性结构。

```
import "fmt"

type Person struct {
    Name string
    Age  int
}

person := Person{Name: "Alice", Age: 30}
fmt.Printf("%#v\n", person)
// Output: main.Person{Name:"Alice", Age:30}
```

go处理json

JSON（JavaScript Object Notation）是一种轻量级的数据交换格式，它使用易于阅读和编写的文本格式，常用于在不同系统之间传递和存储数据。JSON数据结构以键值对的形式组织数据，支持基本数据类型（如字符串、数字、布尔值）以及复杂数据类型（如对象和数组）。

在计算机领域中，JSON被广泛用于Web应用程序的数据交换、API通信以及配置文件等场景。它具有易于理解、轻量级、语言无关、跨平台等优点，可以被大多数编程语言轻松地解析和生成。

在Go语言中，encoding/json包提供了处理JSON数据的功能。它允许开发人员将Go数据结构编码为JSON格式的字符串，或者将JSON格式的字符串解析为Go数据结构，以方便在Go程序中操作和处理JSON数据。通过JSON的编码和解码操作，可以实现与其他语言或系统之间的数据交换和协作。

需要`import "encoding/json"` 在Go语言中，标准库`encoding/json`提供了一套用于JSON编码和解码的功能。你可以使用这个包来处理JSON数据。

下面是一些常见的JSON处理操作：

1. 结构体与JSON的转换：
    - 将结构体编码为JSON：使用`json.Marshal`函数将Go语言的结构体序列化为JSON格式的字节流或字符串。
    - 将JSON解码为结构体：使用`json.Unmarshal`函数将JSON数据解析为Go语言的结构体对象。

```
import "encoding/json"

type Person struct {
    Name string `json:"name"`
    Age  int    `json:"age"`
}

// 编码为JSON
person := Person{Name: "John Doe", Age: 25}
data, err := json.Marshal(person)
if err != nil {
    // 处理错误
} else {
```

```
    jsonStr := string(data)
    // 处理序列化后的JSON字符串
}

// 解码JSON为结构体
jsonStr := `{"name":"John Doe","age":25}`
var decodedPerson Person
err := json.Unmarshal([]byte(jsonStr), &decodedPerson)
if err != nil {
    // 处理错误
} else {
    // 处理解码后的结构体对象
}
```

2. 字符串与JSON的转换：
    - 将字符串解码为JSON：使用`json.Unmarshal`函数将JSON格式的字符串解析为`interface{}`类型的数据结构。
    - 将JSON编码为字符串：使用`json.Marshal`函数将`interface{}`类型的数据结构编码为JSON格式的字符串。

```
import "encoding/json"

// 解码JSON字符串
jsonStr := `{"name":"John Doe","age":25}`
var jsonData interface{}
err := json.Unmarshal([]byte(jsonStr), &jsonData)
if err != nil {
    // 处理错误
} else {
    // 处理解码后的数据结构
}

// 编码为JSON字符串
data, err := json.Marshal(jsonData)
if err != nil {
    // 处理错误
} else {
    jsonStr := string(data)
    // 处理编码后的JSON字符串
}
```

这里，`interface{}`类型是一种动态类型，在JSON解析时可以用来存储各种类型的数据。

time包

以下是`time`包中一些重要的函数及其功能说明：

- `time.Now()`：获取当前的本地时间。

- `time.Date(year int, month Month, day int, hour int, minute int, second int, nsec int, loc *Location)`：根据提供的年、月、日、时、分、秒等参数，创建一个指定时区的时间。
- `time.Time.Format(layout string)`：将时间根据提供的格式字符串进行格式化，并返回格式化后的字符串。
- `time.Parse(layout, value string)`：根据给定的格式字符串解析时间字符串，返回相应的`Time`类型。
- `time.Time.Equal(other Time) bool`：判断两个时间是否相等。
- `time.Time.Before(other Time) bool`：判断一个时间是否在另一个时间之前。
- `time.Time.After(other Time) bool`：判断一个时间是否在另一个时间之后。
- `time.Time.Add(d Duration) Time`：将时间加上指定的时间间隔。
- `time.Duration`：表示持续时间的类型，可以用来表示一段时间，比如`time.Second`表示1秒，`time.Minute`表示1分钟等。
- `time.Sleep(d Duration)`：暂停当前的Goroutine，让程序休眠一段时间。
- `time.NewTimer(d Duration) *Timer`：创建一个定时器，它会在指定的时间间隔之后触发。
- `timer.C`：定时器的通道，通过向该通道发送数据，等待定时器的触发。

数字解析

导入`strconv`包 这个包提供了一系列函数来处理字符串与整数、浮点数等基本数据类型之间的转换

1. 整数转字符串：

```
i := 42
str := strconv.Itoa(i)// 输出: "42"
```

2. 字符串转整数：

```
str := "42"
i, _ := strconv.Atoi(str)// 输出: 42
```

3. 字符串解析为浮点数：

```
str := "3.14"
f, _ := strconv.ParseFloat(str, 64) // 输出: 3.14
```

4. 整数转指定进制字符串：

```
i := 42
str := strconv.FormatInt(int64(i), 16)// 输出: "2a"
```

5. 字符串解析为布尔值：

```
str1 := "true"
b1, _ := strconv.ParseBool(str1)// 输出: true

str2 := "false"
b2, _ := strconv.ParseBool(str2)// 输出: false
```

...

进程信息

基本类型和复合类型

1. 基本类型：

- 布尔类型：`bool`，表示真假两个值（`true`和`false`）。
- 整数类型：`int`、`int8`、`int16`、`int32`、`int64`、`uint`、`uint8`、`uint16`、`uint32`、`uint64`，分别表示有符号整数和无符号整数。
- 浮点数类型：`float32`、`float64`，分别表示单精度浮点数和双精度浮点数。
- 复数类型：`complex64`、`complex128`，分别表示32位和64位的复数类型。
- 字符串类型：`string`，表示字符串值的序列。
- 字符类型：`byte`、`rune`，分别表示字节和Unicode字符。

1. 复合类型：

- 数组类型：使用`[n]T`表示具有固定长度`n`的数组，其中`T`为数组元素类型。
- 切片类型：`[]T`，表示可变长度的序列，是对数组的一种封装。
- 映射类型：`map[K]V`，表示键值对的无序集合，其中`K`为键类型，`V`为值类型。
- 结构体类型：`struct`，用于组织和存储不同类型的字段的聚合数据类型。
- 接口类型：`interface{}`，用于表示一组方法签名的抽象类型，可以通过实现接口的方法来达到多态性。
- 函数类型：用于表示函数的类型，可以将函数作为参数或返回值传递。

除了这些基本类型和复合类型，Go还提供了指针类型、通道类型、函数类型、类型别名等。此外，Go还支持自定义类型，通过`type`关键字可以为现有类型创建新的命名类型。