

Go 基本语法

大明

目标

基础目标：切片的辅助方法、map 辅助方法，以及用内置 map 封装一个 Set 出来。

中级目标：设计 List、普通队列、HashMap 和 LinkedHashMap

高级目标：基于树形结构衍生出来的类型、基于跳表衍生出来的类型、bean copier 机制。

超级目标：并发工具。

熟练掌握 GO 语言。|

项目经历

GO 泛型工具库

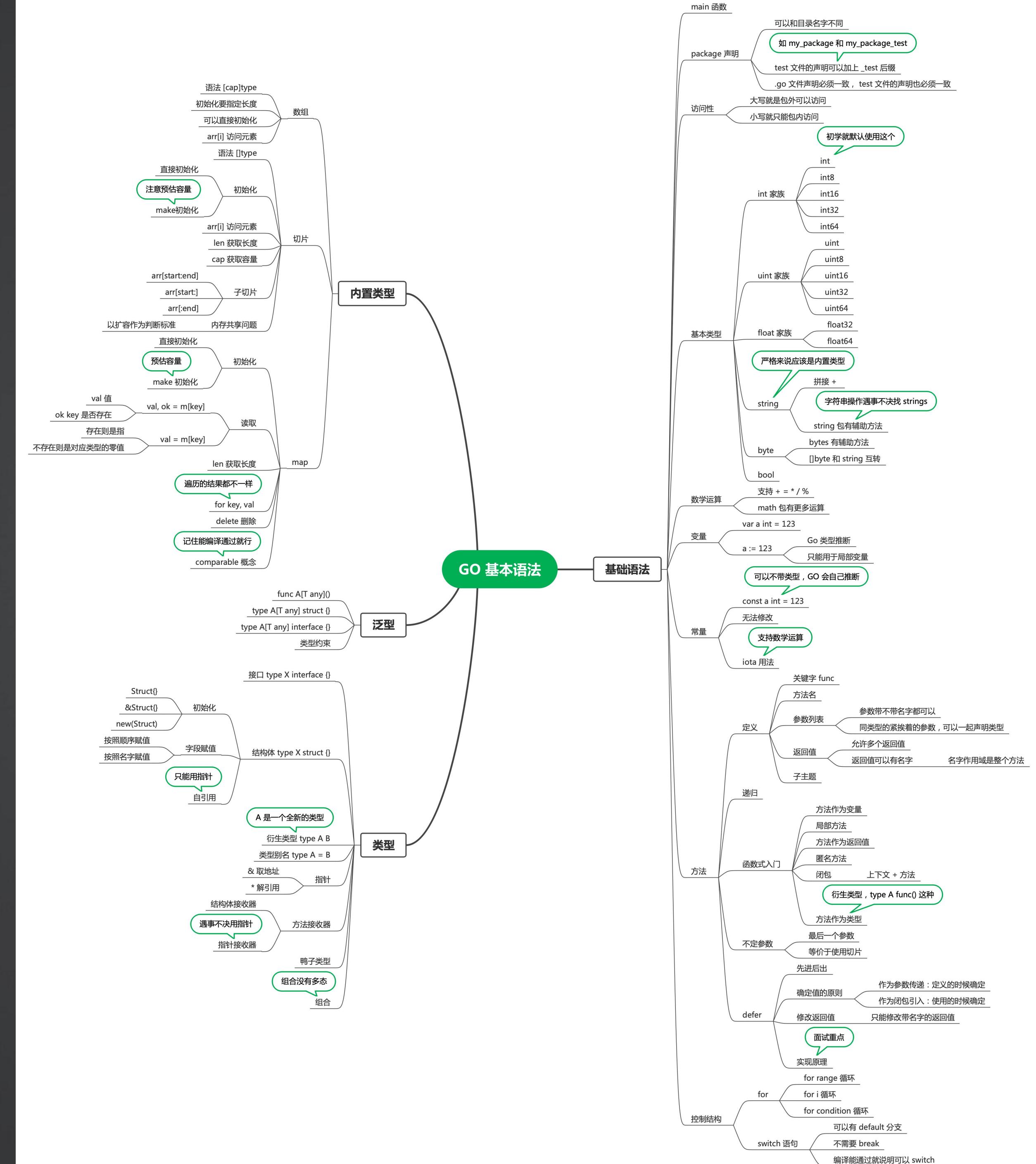
在 GO 出来之后，我在公司内部研发了一个 GO 泛型工具库，用于重构在业务代码中的冗余、模板代码。该工具库提供了高性能的辅助方法和数据结构，包括：

- 切片的辅助方法：添加、删除、查找，求并集、交集，map reduce API
- map 辅助方法
- 扩展 map 实现：接受任意类型的 HashMap, TreeMap, LinkedMap
- List 实现：LinkedList、ArrayList 和 SkipList
- Set：包括 HashSet 和 TreeSet, SortedSet
- 队列：普通队列、优先级队列
- bean 操作辅助类：高性能高扩展的 bean copier 机制
- 并发扩展工具：包括并发队列、并发阻塞队列、并发阻塞优先级队列
- 协程池

目录

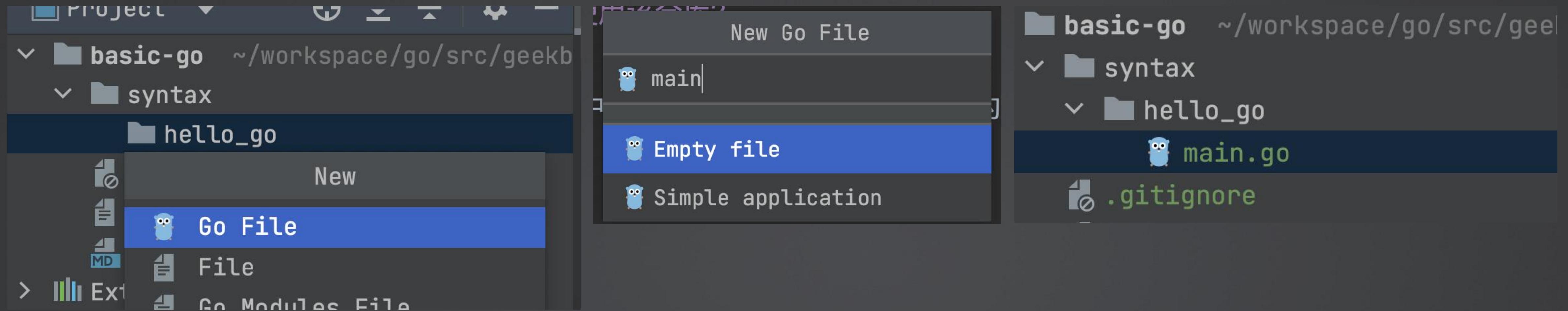
- Hello Go!
- 基础语法
- 内置类型
- 接口与类型定义
- 泛型
- 升职加薪
- 面试要点

知识体系



Hello Go!

Hello Go! —— 第一个 Go 程序

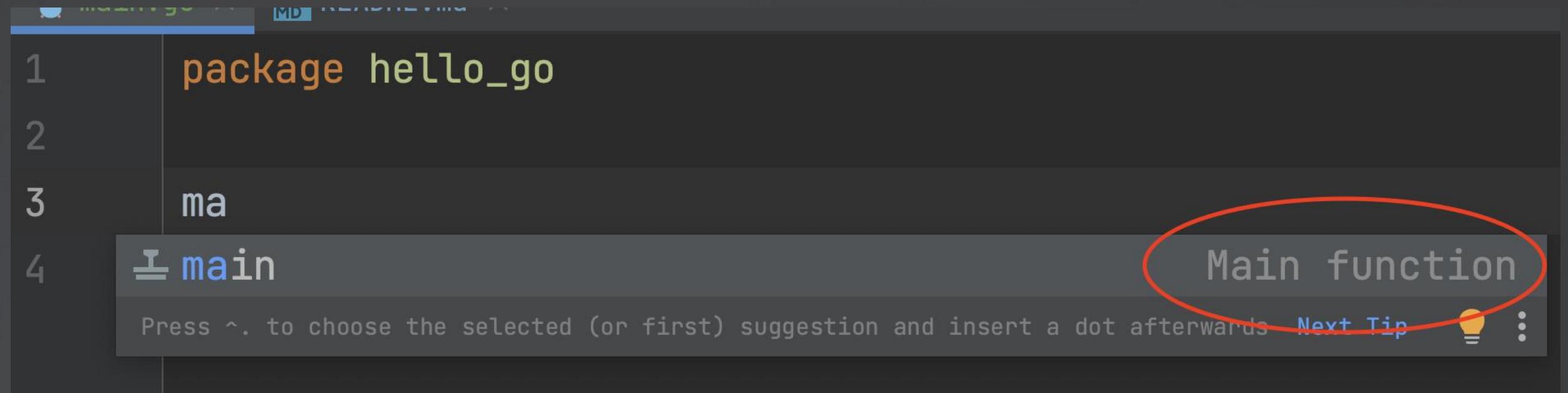


创建一个叫做 **main** 的 go 文件。

Hello Go! 的 main 函数

输入 ma 之后 IDE 有提示，按下 enter 键 IDE 会自动帮你补充 main 函数的定义。

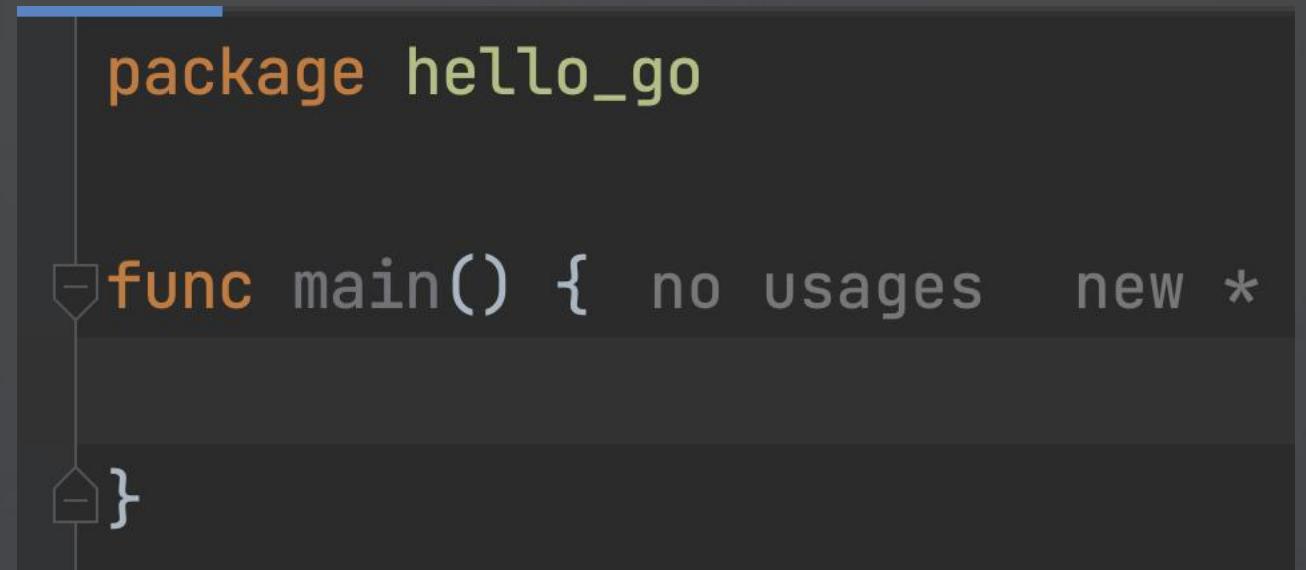
main 函数是 Go 程序的启动入口。



A screenshot of an IDE showing a code editor with the following Go code:

```
1 package hello_go
2
3 ma
4 func main()
```

The word "main" is highlighted in blue. A tooltip "Main function" is displayed next to it, with a red oval highlighting the text. Below the code editor, a status bar message says: "Press ^ to choose the selected (or first) suggestion and insert a dot afterwards".



A screenshot of an IDE showing the completed Go code:

```
package hello_go

func main() { }
```

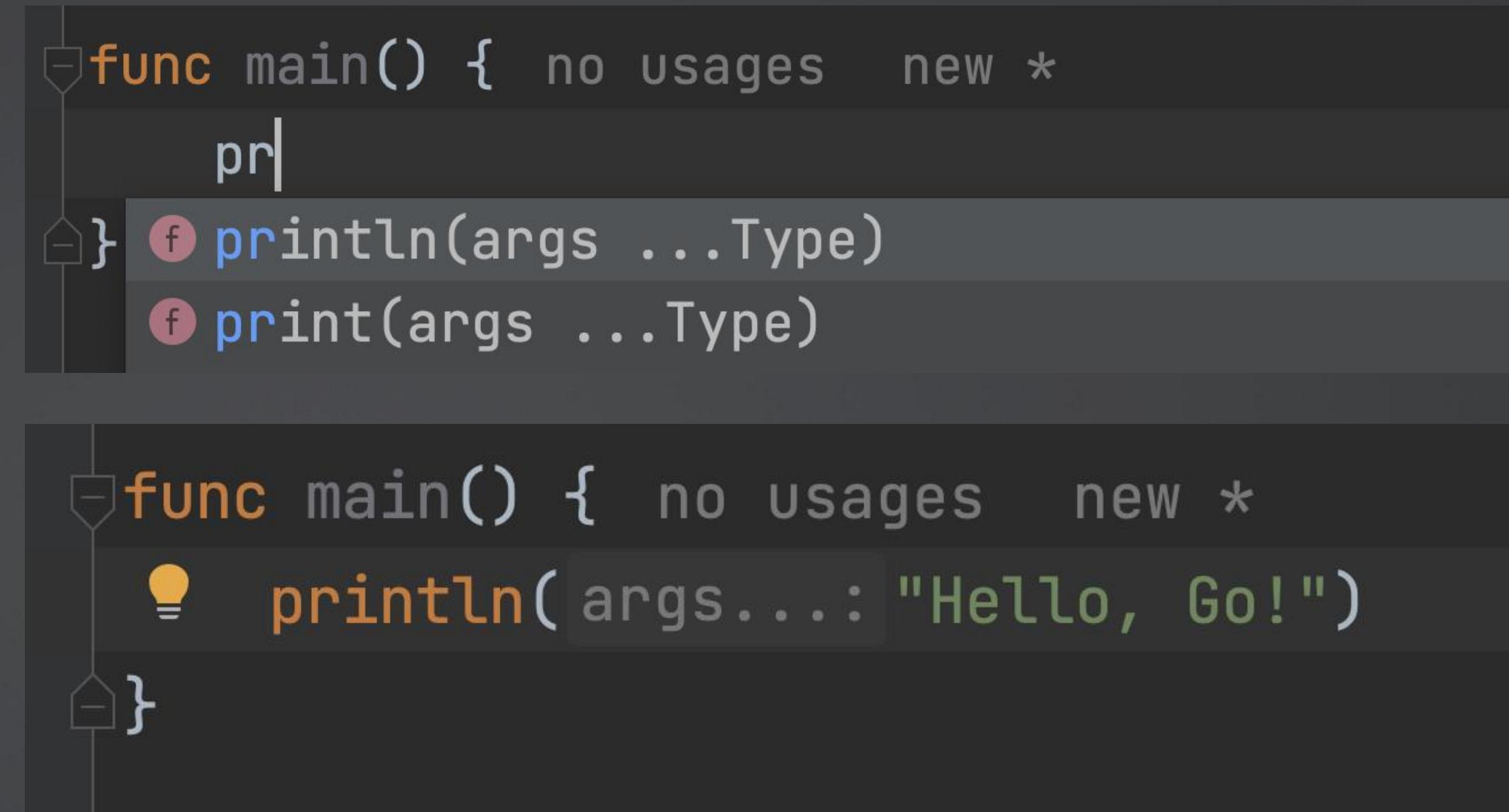
Hello Go! 的 main 函数

输入 pr 就会有提示，其中：

- **println**: 输出并且换行。也就是它会在打印完你的内容之后，自动换行。
- **print**: 和 println 类似，只是最后不会换行。

在 IDE 的提示里面，前面的 f 是指这是一个方法。

这里我们让它打印一句话 “Hello,Go!”



The screenshot shows a code editor with the following Go code:

```
func main() { no usages new *
    pr|}
    } f println(args ...Type)
    f print(args ...Type)
```

The cursor is at the end of the word "pr". A code completion dropdown is open, showing two suggestions: "f println(args ...Type)" and "f print(args ...Type)".

func main() { no usages new *
 ⚡ println(args...: "Hello, Go!")
 }

The word "println" is highlighted in orange, and the string "Hello, Go!" is highlighted in green. A yellow lightbulb icon is positioned next to the "println" call.

运行 main 函数

尝试运行一下。

首先进去 main 函数所在的目录下，而后运行 `go run main.go`。

这个命令的意思是运行这个 main.go 里面的 main 函数。

紧接着你遇到了一个错误：“ package command-line-arguments is not a main package”。

```
→ basic-go git:(main) ✘ cd syntax
→ syntax git:(main) ✘ cd hello_go
→ hello_go git:(main) ✘ pwd
/Users/mindeng/workspace/go/src/geekbang/basic-go/syntax/hello_go
→ hello_go git:(main) ✘ go run main.go
package command-line-arguments is not a main package
→ hello_go git:(main) ✘
```

这个错误的意思是：你传入的 main.go 的 package 不是声明为 main。

运行 main 函数

在将包名修改为 main 函数之后，就发现 IDE 在 main 函数边上加了一个绿色小图标，它的意思是可以直接运行了。

点击这个小图表，出来两个选项，以及对应的快捷键：

- Run：这个是运行
- Debug：以 DEBUG 模式运行。你在研发阶段，基本上都用这个模式。

在 DEBUG 模式下，你就可以在 IDE 里面打断点。

```
package hello_go      这边的改成 main

func main() { no usages new *
    println(args...: "Hello, Go!") }
```

```
1 package main
2
3 3 ▶ func main() { new *
4     println(args...: "Hello, Go!") }
5 }
```

```
3 ▶ Run 'go build gitee.com/g...' ⌘⇧R
4 Debug 'go build gitee.com/g...' ⌘⇧D o!
5 Modify Run Configuration...
```

运行 main 函数

```
API server listening at: 127.0.0.1:58818
debugserver-@(#)PROGRAM:LLDB  PROJECT:lldb-1403.0.17.67
for arm64.

Got a connection, launched process /Users/mindeng/Library/C
_gitee_com_geekbang_basic_go_syntax_hello_go (pid = 41397).
Hello, Go!
Exiting.

Debugger finished with the exit code 0
```

IDE DEBUG 运行输出结果。

```
→ hello_go git:(main) ✘ go run main.go
Hello, Go!
→ hello_go git:(main) ✘
```

控制台运行 go run 命令。

基础语法

Go 基础语法

- main 函数概览
- 基本类型和 string
- 变量与常量声明
- 方法声明与调用
- 控制结构

main 函数概览

从这个 main 函数中，你可能看出来很多 Go 程序的特点。

- 每一个文件都需要有一个 package 声明。
- 方法由方法关键字 func + 方法名 + 参数列表 + 返回值 + 方法体组成。
- Go 程序不要求在一句代码后面加分号，换行就被认为一句代码结束了。
- main 函数无参数、无返回值。
- main 方法必须要在 main 包里面，`go run main.go` 就可以执行。



A screenshot of a code editor showing a Go program. The code is:

```
package main    包名
func main() {    方法关键字    方法名
    new *
    println(args...: "Hello, Go!")
}
```

The code editor highlights the word 'args' in green, indicating it is a variable. A yellow lightbulb icon is shown next to the closing brace of the main function, likely indicating a suggestion or error.

main 函数特性

如果 main 函数里面引用了同一个包的其它方法、类型，那么要加上对应的文件，可以：

- go run 命令加上其它文件，例如 `go run main.go hello.go`
- go run .
- go build . 然后执行 ./hello_go

The screenshot shows a Go code editor interface. At the top, there is a code editor window with the following Go code:

```
2
3 ► func main() { new *
4     println(args...: "Hello, Go!")
5     Hello()
6 }
7
```

The word "Hello" is highlighted in green. Below the code editor is a terminal window showing the execution of the program:

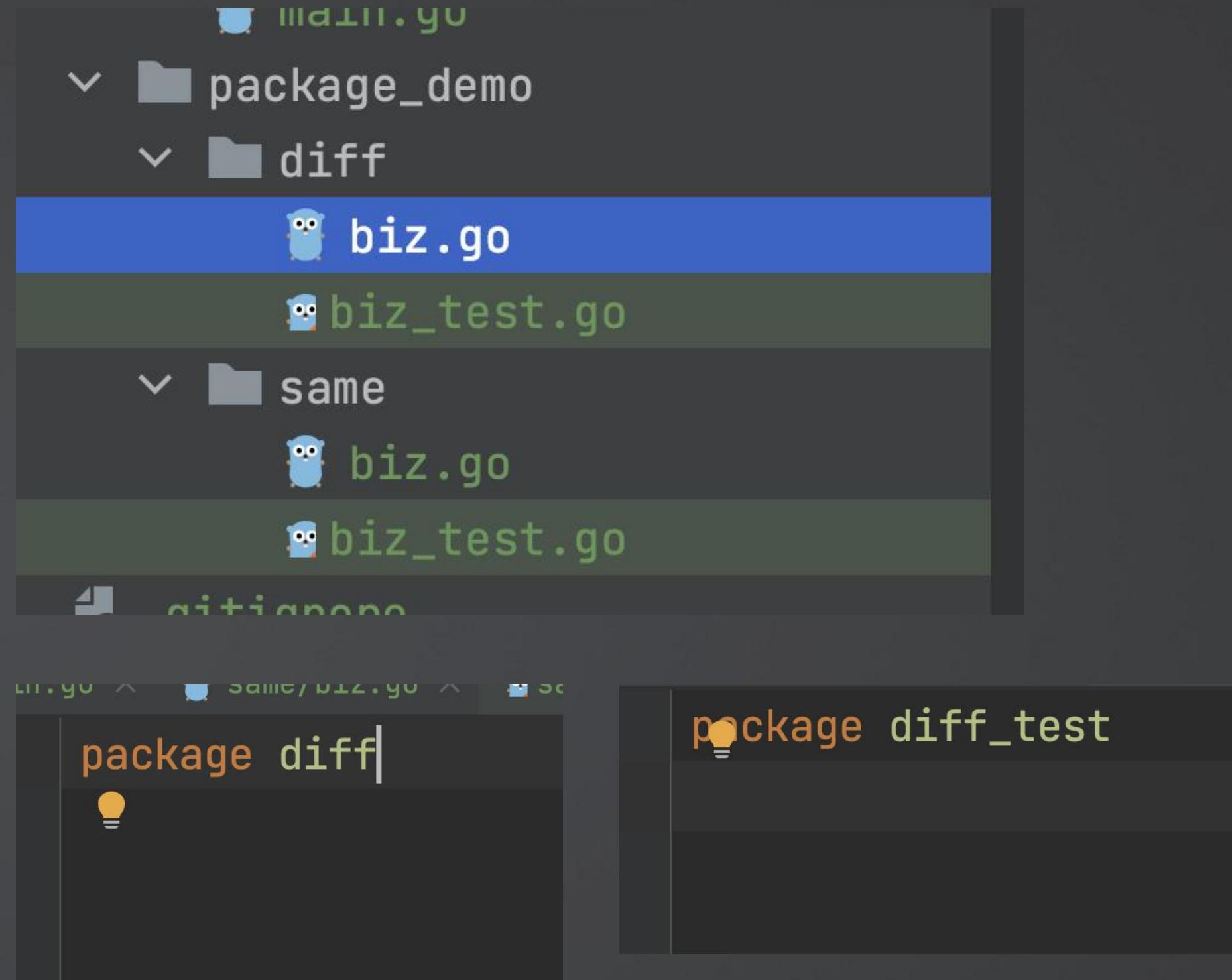
```
→ hello_go git:(main) ✘ go run main.go
# command-line-arguments
./main.go:5:2: undefined: Hello
→ hello_go git:(main) ✘ go run main.go hello.go
Hello, Go!
调用了 hello, Go!
→ hello_go git:(main) ✘
```

The terminal output shows the program running and printing "Hello, Go!" followed by a message indicating it called "hello, Go!".

package 声明

每一个文件都必须有一个 package 声明：

- 同一个目录下，go 文件 package 声明必须一致。
- 同一个目录下，go 文件和 test.go 文件的 package 声明可以不一致。
 - 例如 go 文件的 package 叫做 `daming`，那么同一个目录下的 test 文件可以的 package 可以叫做 `daming_test`。
- package 名字可以和目录名字不一样。



Go 基础语法

- main 函数概览
- 基本类型和 string
- 变量与常量声明
- 方法声明与调用
- 控制结构

基本类型：int、uint、float

Go 里面的基本类型：

- int 家族：int8、int16、int32、int64、int。在内存中分别用 1、2、4、8 个字节来表达，而 int 用几个字节则取决于CPU。目前大多数 int 都是 8 字节。
- uint 家族：uint8、uint16、uint32、uint64、uint。无符号整数，类似于 int 家族，优先使用 uint。
- float 家族：float32、float64。浮点数，优先使用 float64。



Tip: 不必死记硬背，依赖于 Goland 的提示。

基本类型：数字类型的加减乘除

在 Go 里面，四则运算分别对应于：

- +：加法，还有特殊的自增写法 ++
- -：减法，还有特殊的自减写法 --
- *：乘法
- /：除法

Go 里面，只有同类型才可以执行加减乘除。

某些语言有自动类型转换，Go 是没有的。

```
func main() { new *
    // a 和 b 都是 int
    var a int = 456
    var b int = 123
    println(a + b)
    println(a - b)
    println(a * b)
    println(a / b)
```

```
var c float64 = 12.3
// 编译不通过
println(a + c)
```

💡 var d int32 = 12
// 一样编译不通过
println(a + d)

基本类型：math 包

在 Go 里面，更加复杂的运算都在 math 包里面，
包括三角函数、聚合函数等。

注意！不要死记硬背，需要的时候再到数学包里找。
你只需要记住，**复杂数学操作找 math 包。**

```
func Log(x float64) float64
func Log10(x float64) float64
func Log1p(x float64) float64
func Log2(x float64) float64
func Logb(x float64) float64
func Max(x, y float64) float64
func Min(x, y float64) float64
func Mod(x, y float64) float64
func Modf(f float64) (int float64, frac float64)
func NaN() float64
func Nextafter(x, y float64) (r float64)
func Nextafter32(x, y float32) (r float32)
func Pow(x, y float64) float64
func Pow10(n int) float64
func Remainder(x, y float64) float64
func Round(x float64) float64
func RoundToEven(x float64) float64
func Signbit(x float64) bool
func Sin(x float64) float64
func Sincos(x float64) (sin, cos float64)
func Sinh(x float64) float64
func Sqrt(x float64) float64
func Tan(x float64) float64
func Tanh(x float64) float64
func Trunc(x float64) float64
func Y0(x float64) float64
func Y1(x float64) float64
func Yn(n int, x float64) float64
```

基本类型：数字类型的极值

数字类型的极值都在 `math` 包里面，作为常量。

- `int` 和 `uint` 族都有最大值和最小值。
- `float32` 和 `float64` 只有最大值和最小正数，没有最小值。

注意，低版本的 Go SDK 不一定有全部的极值的常量。

```
// Extremum 极值
func Extremum() { 1 usage  new *
    println(args...: "float64 最大值", math.MaxFloat64)
    // 没有 float64 最小值
    //println("float64 最小值", math.MinFloat64)
    println(args...: "float64 最小的正数", math.SmallestNonzeroFloat64)

    println(args...: "float32 最大值", math.MaxFloat32)
    // 没有 float32 最小值
    //println("float32 最小值", math.MinFloat32)
    println(args...: "float32 最小正数", math.SmallestNonzeroFloat32)

    // int 族和 uint 族都有最大值最小值
}
```

string 类型

有两种写法：

- 使用双引号“”起来。如果在字符串里面的“”就需要用\来进行转义。
- 使用反引号`起来。这种写法可以换行，但是内部不能有反引号（转义也不行）。

The screenshot shows a Go code editor with the following code:

```
func String() { no usages new *
    // He said:"hello, go!"
    println(args...: "He said:\\"hello, go!\\")
    println(args...: `我可以换行
    这是新的行
    但是这里不能有反引号`)
}
```

A tooltip is displayed over the second `println` call, containing the text: "这是新的行" (This is a new line) and "但是这里不能有反引号" (But you can't have a backtick here). A yellow lightbulb icon is next to the first line of the tooltip.

Tip：不建议自己手写转义，而是自己先写好，然后复制到 Goland，IDE 会自动完成转义。

string 基本操作

string 的拼接直接使用 + 号就可以。需要注意的是，某些语言支持 string 和别的类型拼接，但是 Go 不可以。

```
println(args...: "hello, " + "Go!")  
// 字符串只能和字符串拼接  
//println("hello, " + 123)
```

string 的长度很特殊：

- **字节长度**：和编码无关，用 len(str) 获取
- **字符数量**：和编码有关，用编码库来计算，
默认情况下使用 utf8 库

```
println(len(v: "你好")) // 输出 6  
println(utf8.RuneCountInString(s: "你好")) // 输出2  
println(utf8.RuneCountInString(s: "你好ab")) // 输出4
```

在国内研发，大部分情况下都是中文语境，所以要千万小心中文处理。

字符串操作：strings 操作

- strings 主要方法（你所需要的全部都可以找到，依赖于 IDE 提示最好）：
 - 查找和替换
 - 大小写转换
 - 子字符串相关
 -

The screenshot shows a code editor with the following code snippet:

```
import "strings"

func StringReplaceAll(s string, old string, new string) string {
    // Help text: This is a new line, but here it can't be used.
    print("This is a new line, but here it can't be used.\n")
    print("```\n")
    return strings.ReplaceAll(s, old, new)
}

func main() {
    str := "Hello, world!"
    str = StringReplaceAll(str, "world", "Go")
    fmt.Println(str)
}
```

A code completion dropdown is open over the word `strings.`. It lists various methods available on the `strings` package:

- f Title(s string)
- f ReplaceAll(s string, old string, new string)
- f Map(mapping func(rune) rune, s string)
- f HasSuffix(s string, suffix string)
- f Replace(s string, old string, new string, n int)
- T Builder
- f Clone(s string)
- f Compare(a string, b string)
- f Contains(s string, substr string)
- f ContainsAny(s string, chars string)
- f ContainsRunes(s string, runes []rune)

Below the list, a note says: "Press ^ to choose the selected (or first) suggestion and insert a dot afterward".

依赖于 IDE 提示，不要死记硬背！

基本类型：byte

byte 就是字节，本质上就是 uint8，也会用它来表达 ASCII 字符。

对应的操作在 bytes 包上。

```
// byte is an alias for uint8 and is equivalent to uint8 in all ways. It is
// used, by convention, to distinguish byte values from 8-bit unsigned
// integer values.
type byte = uint8

// rune is an alias for int32 and is equivalent to int32 in all ways. It is
// used, by convention, to distinguish character values from integer values.
type rune = int32
```

The screenshot shows a code editor with the following code snippet:

```
// byte is an alias for uint8 and is equivalent to uint8 in all ways. It is
// used, by convention, to distinguish byte values from 8-bit unsigned
// integer values.
type byte = uint8

// rune is an alias for int32 and is equivalent to int32 in all ways. It is
// used, by convention, to distinguish character values from integer values.
type rune = int32
```

Below the code, a code completion dropdown is open, showing various methods available on the `bytes` package:

- f bytes.Compare(a []byte, b []byte) bytes
- f bytes.Contains(b []byte, subslice []byte) bytes
- T bytes.Buffer bytes
- f bytes.Count(s []byte, sep []byte) bytes
- f bytes.ContainsAny(b []byte, chars string) bytes
- f bytes.ContainsRune(b []byte, r rune) bytes
- f bytes.Equal(a []byte, b []byte) bytes
- / f bytes.EqualFold(s []byte, t []byte) bytes
- V bytes.ErrTooLarge bytes
- f bytes.Fields(s []byte) bytes
- / f bytes.FieldsFunc(s []byte, f func(rune) bool) bytes
- P f bytes.HasPrefix(s []byte, prefix []byte) bytes

The documentation at the bottom of the dropdown states: "Press ^ to choose the selected (or first) suggestion and insert a bytes."

[]byte 和 string

一般来说很少单独使用 byte，都是使用 []byte，即 byte 的切片。

[]byte 和 string 之间可以互相转换。

你可以看到，类型转化在 Go 里面的语法就是 Dst(Src)。Dst 就是你的目标类型。

不要担心不能转，因为不能转的编译器都会直接报错。

```
func Byte() { 1 usage  new *
    var a byte = 'a'
    // 输出的是的 a 的ASC II 表达 97
    println(a)

    var str string = "this is string"
    var bs []byte = []byte(str)  类型转换
    println(bs)
}
```

bool 类型

bool 类型就两个取值： true 和 false。

bool 运算：

- 且 (AND) : `a && b`
- 或 (Or) : `a || b`
- 取反: `!a`

组合取反是一个很容易犯错的地方：

- `!(a && b)` 等价于什么？
- `!(a || b)` 等价于什么？

```
func Bool() { no usages new *
    var a bool = true
    var b bool = false
    var c bool = a || b // true
    println(c)
    var d bool = a && b // false
    println(d)
    var e bool = !a
    println(e)
}
```

Go 基础语法

- main 函数概览
- 基本类型和 string
- 变量与常量声明
- 方法声明与调用
- 控制结构

变量声明 var

- var, 语法: var name type = value
 - 局部变量
 - 包变量
 - 块声明
- 驼峰命名
- 首字符是否大写控制了访问性: 大写包外可访问
- Golang 支持类型推断:
 - 如果是整数, 默认是 int 类型
 - 如果是浮点数, 默认是 float64 类型

如果局部变量声明了但是没有用, 那么会编译错误。

```

func main() {
    // int 是灰色的, 是因为 golang 自己可以做类型推断, 它觉得你可以省略
    var a int = 13
    println(a)

    // 这里我们省略了类型
    var b = 14
    println(b)

    // 这里 uint 不可省略, 因为生路之后, 因为不加 uint 类型, 15会被解释为 int 类型
    var c uint = 15
    println(c)

    // 这一句无法通过编译, 因为 golang 是强类型语言, 并且不会帮你做任何的转换
    // println(a == c)

}

// Global 首字母大写, 全局可以访问
var Global = "全局变量"

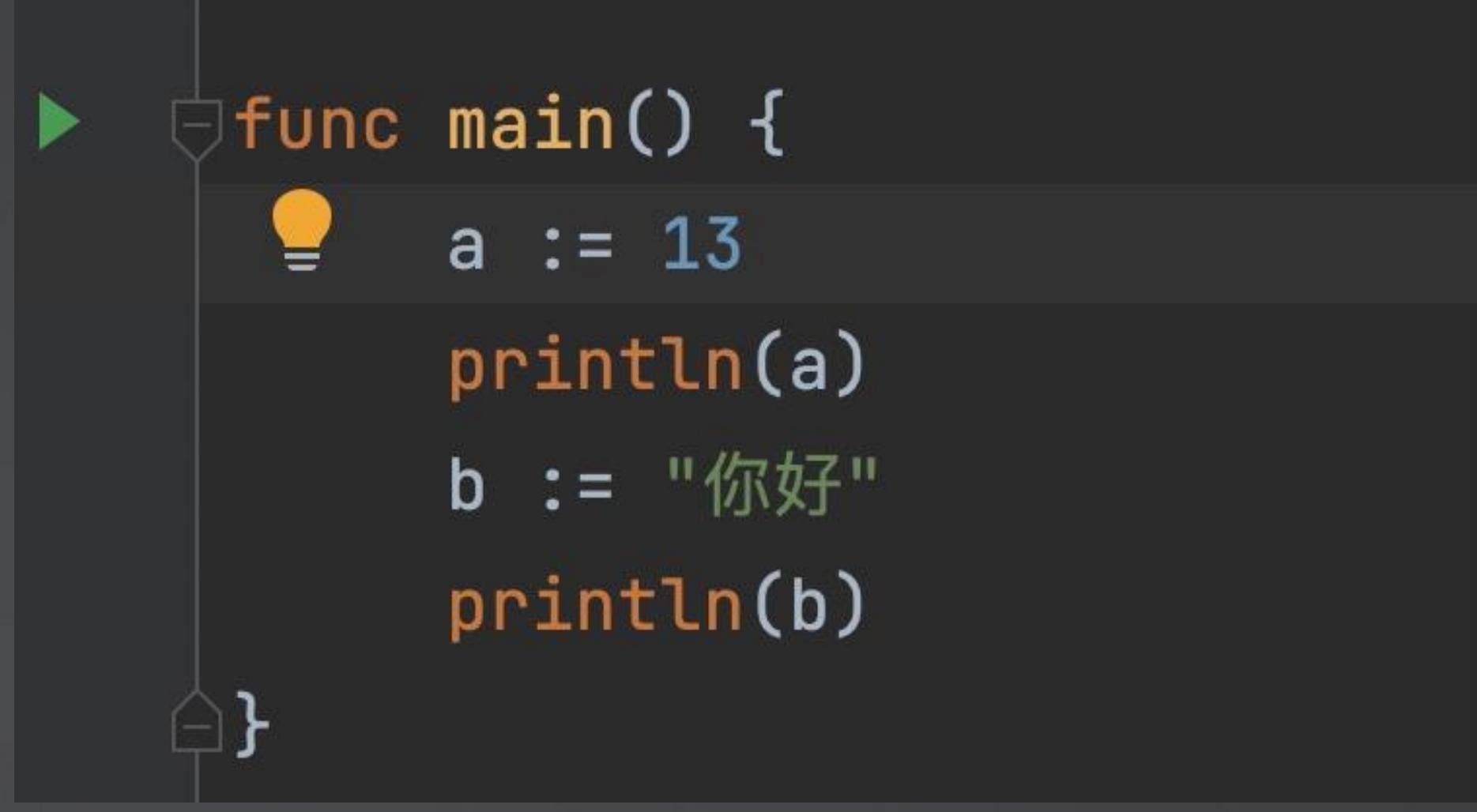
// 首字母小写, 只能在这个包里面使用
// 其子包也不能用
var local = "包变量"

var (
    First string = "abc"
    second int32 = 16
)

```

变量声明 :=

- 只能用于局部变量，即方法内部
- Golang 使用类型推断来推断类型。数字会被理解为 int 或者 float64。（所以要其它类型的数字，就得用 var 来声明）



```
▶ func main() {  
    a := 13  
    println(a)  
    b := "你好"  
    println(b)  
}
```

A screenshot of a code editor showing a Go program. The code defines a main function that declares two variables: 'a' and 'b'. Variable 'a' is assigned the value 13 and its type is inferred as int. Variable 'b' is assigned the string value "你好" and its type is inferred as string. Both assignments use the := operator. The code is syntax-highlighted, with 'func' in orange, 'main' in purple, and variable names in blue. Brackets and braces are shown in grey. A yellow lightbulb icon is placed next to the assignment of 'a'.

基础语法：变量声明易错点

- 变量声明了没有使用
- 类型不匹配
- 同作用域下，同名变量只能声明一次

```
var aa = "hello"
// var aa = "bbb" 这个包已经有一个 aa 了，所以再次声明会导致编译

func main() {
    aa := 13 // 虽然包外面已经有一个 aa 了，但是这里从包变成了局部变量
    println(aa)

    var bb = 15
    //var bb = 16 // 重复声明，也会导致编译不通过
    println(bb)

    bb = 17 // OK, 没有重复声明，只是赋值了新的值
    // bb := 18 // 不行，因为 := 就是声明并且赋值的简写，相当于重复声明了 bb
}
```

常量声明 const

- 首字符是否大写控制了访问性：大写包外可访问
- 驼峰命名
- 支持类型推断
- 无法修改值

```
const internal = "包内可访问" no usages new *
const External = "包外可访问" no usages new *

func Const() { no usages new *
    const a = "你好"
    println(a)
}
```

常量声明 iota 用法

- iota 可以方便地控制常量初始化。
- 可以使用 iota，也可以使用 iota 的数学运算，包括位移操作。
- 只有主动赋值或者另起一个 iota 才会从头开始计算值。

```
const (
    Status0 = 0 = iota 每次加1
    Status1 = 1 no usages new *
    Status2 = 2 no usages new *
    Status3 = 3 no usages new *
    // 不管隔多少行，都是接着4
    Status4 = 4 no usages new *

    // 插入一个主动赋值的就中断了 iota
    Status6 = 6 no usages new *
    Status7 = 6 no usages new *
)

const ( 每次左移1位
    One = 0 = iota << 1 no usages new *
    Two = 2 no usages new *
    Four| = 4 no usages new *
)
```

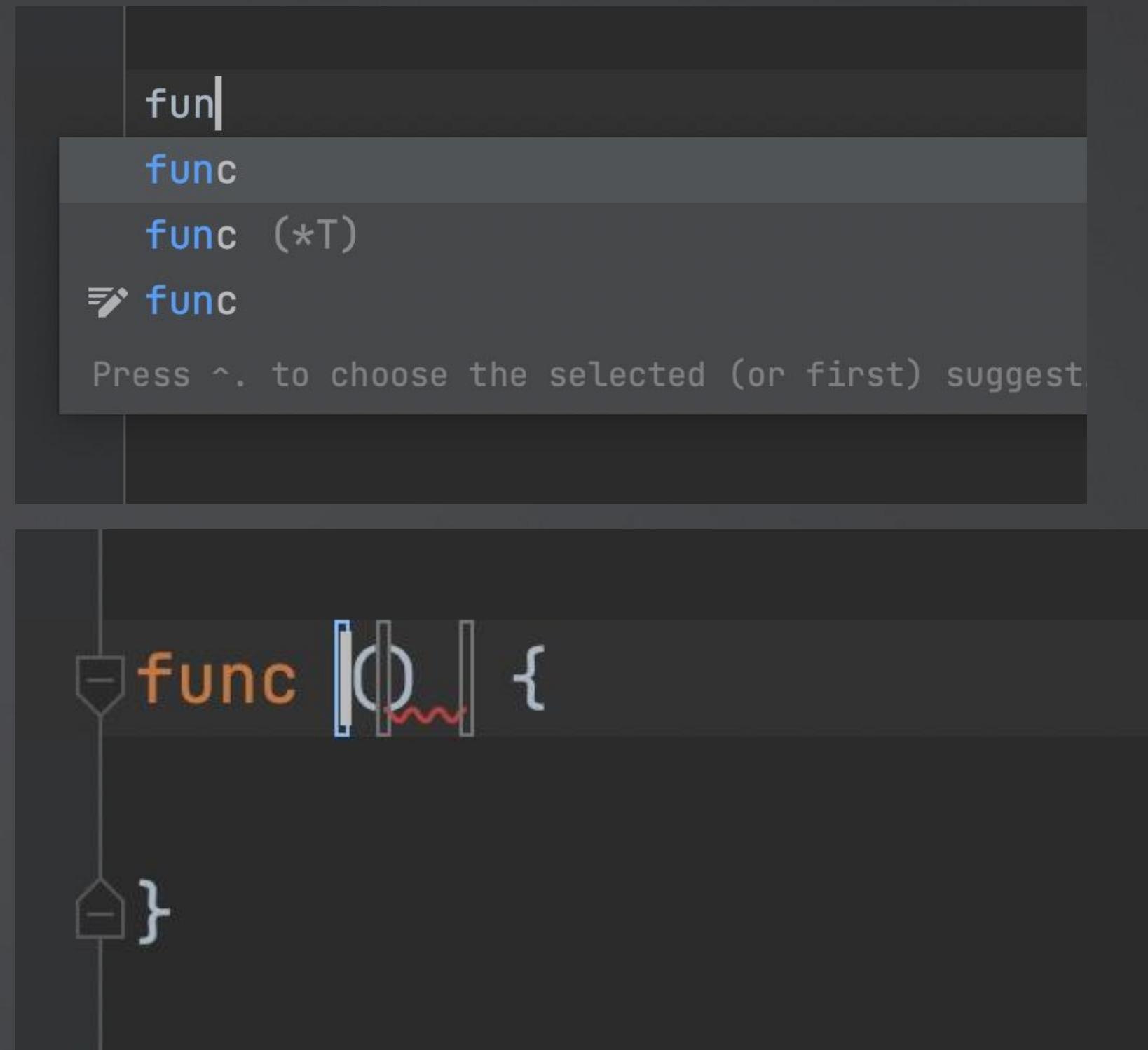
Go 基础语法

- main 函数概览
- 基本类型和 string
- 变量与常量声明
- 方法声明与调用
- 控制结构

方法声明

- 四个部分：
 - 关键字 func
 - 方法名字：首字母是否大写决定了作用域
 - 参数列表：[<name type>]
 - 返回列表：[<name type>]

名字 + 参数列表 + 返回值，也叫做方法签名
(signature)。



Tip: 使用 GoLand 来创建文件夹，它会自动加上 package。

方法声明：返回值

Go 的方法和别的语言比起来，有一个很大的不同：

- 可以有多个返回值，并且返回值可以有名字。

带名字的返回值，就相当于你声明了一个作用域在整个方法的局部变量。

用不用带名字的返回值纯粹是个人风格，你看自己喜好。

```
// Func0 单一返回值
func Func0(name string) string { 1 usage  new *
    return "hello, world"
}

// Func1 多个返回值
func Func1(a, b, c int, str1 string) (string, error) {
    return "", nil
}

// Func2 带名字的返回值
func Func2(a int, b int) (str string, err error) { 1 usage
    str = "hello"
    // 带名字的返回值，可以直接 return
    return
}

// Func3 带名字的返回值
func Func3(a int, b int) (str string, err error) { 1 usage
    res := "hello"
    // 虽然带名字，但是我们并没有用
    return res, err: nil
}
```

方法调用：接收返回值

方法调用只需要用足够的参数去接收返回值，并且传入了调用参数就可以。

在多个返回值的时候，如果你想忽略一些返回值，可以使用 `_`

记住一条核心原则：同一个作用域内，如果左边出现了新的变量，那么就需要使用 `:=` 来接收返回值。

```
func Invoke() { no usages new *
    str, err := Func2(a: 1, b: 2)
    println(str, err)
    // 忽略返回值
    _, _ = Func2(a: 1, b: 3)

    // 部分忽略返回值
    // str 是已经声明好了
    str, _ = Func2(a: 3, b: 4)
    // str1 是新变量，需要使用 :=
    str1, _ := Func2(a: 3, b: 4)
    println(str1)

    // str2 是新变量，需要使用 :=
    str2, err := Func2(a: 3, b: 4)
    println(str2)
```

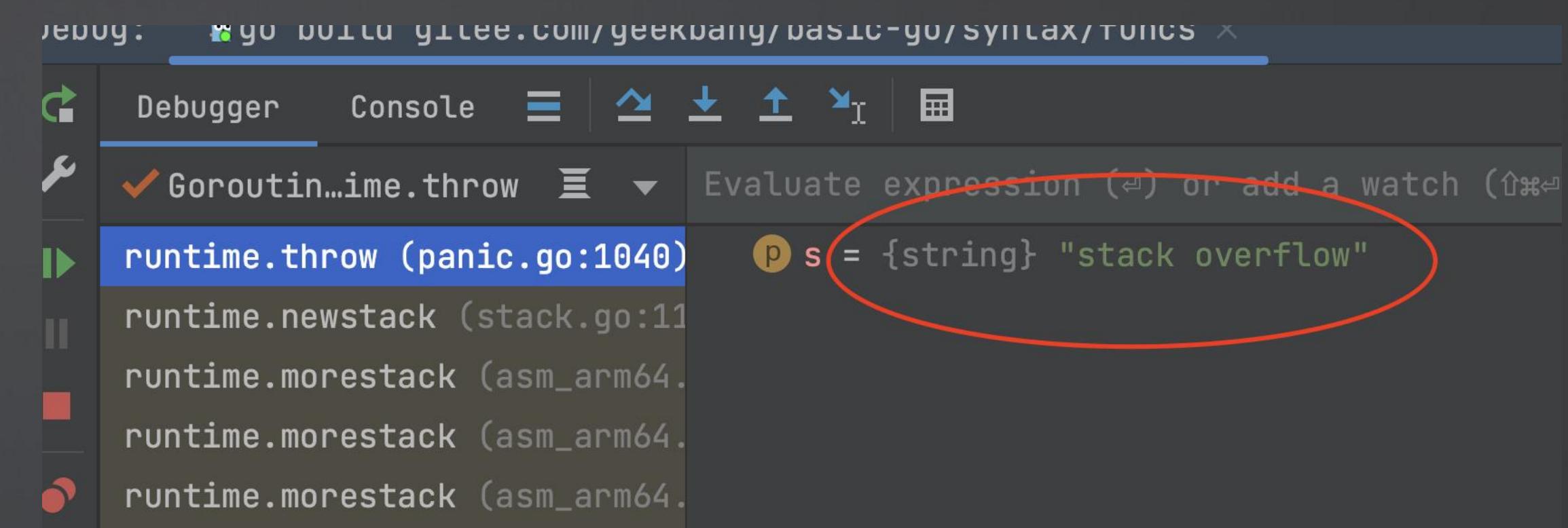
方法调用：递归

一个方法内部可以调用它自己，这也就是所谓的
递归。

递归要小心，如果递归嵌套太深的话，可能会出现栈溢出错误。

- 治标的话，栈溢出可以考虑通过增加栈大小来解决。
- 治本的话，还是要找到对应的递归代码，进行修改。

```
5 // Recursive 递归
6 // 这个方法运行的时候会出现错误
7 func Recursive() { 3 usages new *
8     Recursive()
9 }
```



函数式编程入门：方法做为变量

方法本身就可以赋值给某个变量，而这个变量就可以直接发起调用。

右图中的 myFunc3 本质上是一个变量，只不过这个变量等于 Func3。

```
- func Func4() { no usages new *
  myFunc3 := Func3
  _, _ = myFunc3( a: 1, b: 2)
}
```

函数式编程入门：局部方法

可以在方法内部声明一个局部方法，它的作用域就在本方法内。

```
func Func5() { no usages new *
    fn := func(name string) string {
        return "hello, " + name
    }
    fn(name: "大明")
}
```

函数式编程入门：方法作为返回值

方法也可以作为返回值。

注意，不是方法的返回值，是方法本身就可以作为返回值。

后面你们会经常看到我的这种用法，它本身也是闭包的一种应用。

右图中的 sayHello 就是返回来的那个方法，所以可以直接调用。

```
// Func6 的返回值是一个方法,  
func Func6() func(name string) string { 2 usages new *  
    return func(name string) string {  
        return "hello," + name  
    }  
}  
  
func Func6Invoke() { no usages new *  
    // sayHello 的签名就是 func(name string) string  
    sayHello := Func6()  
    sayHello(name: "大明")  
}
```

函数式编程入门：匿名方法发起调用

在前面你已经见过声明一个匿名方法，然后赋值给一个变量。

那么你也可以声明一个匿名方法，而后可以立刻发起调用。

右图中hello就是匿名函数直接发起调用之后返回的数据。

这种用法在 defer 中用得多。

```
// Func7 演示匿名方法
func Func7() { 1 usage  new *
    hello := func() string {
        return "hello, world"
    }()
    println(hello)
}
```

函数式编程入门：闭包

闭包（closure），方法 + 它绑定的运行上下文。

右图返回的是闭包，它由两部分构成：

- 方法
- 运行时上下文：它只引用了 name 这一个变量

闭包也是一个面试热点，所以你至少需要记住闭包的定义。

闭包如果使用不当可能会引起内存泄露的问题。即一个对象被闭包引用的话，它是不会被垃圾回收的。记住这个结论，你后面面试用得上。

```
func Closure(name string) func() string {  
    // 返回的这个函数，就是一个闭包。  
    // 它引用到了 Closure 这个方法的入参  
    return func() string {  
        return "hello, " + name  
    }  
}
```

不定参数

Go 支持不定参数。不定参数是指最后一个参数，可以传入任意多个值。

注意必须是最后一个参数才可以声明为不定参数。

不定参数在方法内部可以被当成切片来使用。

Option 模式大量应用了不定参数。

```
// YourName 不定参数例子
// 一个人可能有很多个别名，也可能一个都没有
func YourName(name string, alias ...string) { 4 usages  new *
    if len(alias) > 0 {
        println(alias[0])
    }
}

func YourNameInvoke() { no usages  new *
    YourName( name: "Deng Ming")
    YourName( name: "Deng Ming", alias...: "Da Ming")
    YourName( name: "Deng Ming", alias...: "Da Ming", "Zhong Ming")
}
```

defer

Go 里面有一个机制，允许你从方法返回的前一刻，执行一段逻辑。

这个也就是 `defer`，也叫做延迟调用。

`defer` 类似于栈，也就是后进先出。也就是说，先定义的后执行；后定义的先执行。

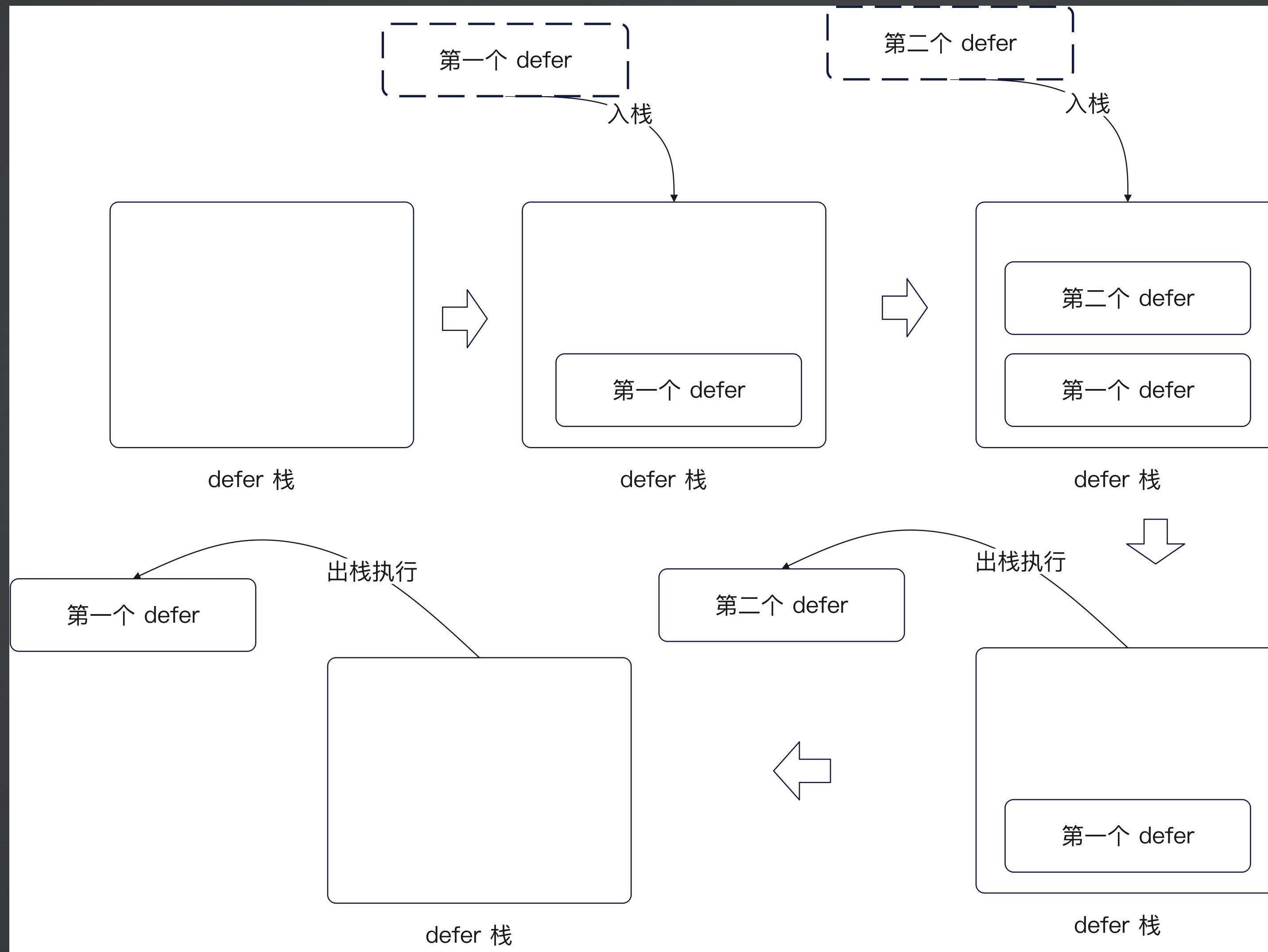
一个方法不能超过 8 个 `defer`。

右图我们利用匿名函数定义了两个 `defer`。

```
func Defer() { no usages new *  
    defer func() {  
        println(args...: "第一个 defer")  
    }()  
  
    defer func() {  
        println(args...: "第二个 defer")  
    }()  
}  
  
_gitee_com_geekbang_basic_go_syntax_funcs (pid =  
    第二个 defer  
    第一个 defer  
    Exiting.
```

注意定义顺序和执行顺序。

defer 执行顺序



defer 与闭包

我们通常都用闭包来写 defer，所以就会涉及到一个问题，那就是 defer 执行的时候怎么确定里面的值？



```
func DeferClosure() { 1 usage  new *
    i := 0
    defer func() {
        println(i)
    }()
    i = 1
}
```

输出多少？

defer 与闭包

确定值的原则：

- 作为参数传入的：定义 defer 的时候就确定了。
- 作为闭包引入的：执行 defer 对应的方法的时候才确定。

```
func DeferClosure() { 1 usage  new *
    i := 0
    defer func() {
        println(i)    闭包写法
    }()
    i = 1
}

func DeferClosureV1() { no usages  new *
    i := 0
    defer func(val int) {  参数传递
        println(val)
    }(i)
    i = 1
}
```

defer 修改返回值

```
func DeferReturn() int { 1 usage  ne  
    a := 0  
    defer func() {  
        a = 1  
    }()  
    return a  
}
```

```
func DeferReturnV1() (a int) { 1 usag  
    a = 0  
    defer func() {  
        a = 1  
    }()  
    return a  
}
```

```
func DeferReturnV2() *MyStruct { 1 u  
    a := &MyStruct{  
        name: "Jerry",  
    }  
    defer func() {  
        a.name = "Tom"  
    }()  
    return a  
}  
  
type MyStruct struct { 2 usages ne  
    name string  
}
```

如果是带名字的返回值，那么可以修改这个返回值，否则不能修改。

注意：最右边的并没有修改 a，而是修改 a 指向的结构体。

自测题

先猜一下，然后课后自己去运行一下。

```
func DeferClosureLoopV1() { 1 usage
    for i := 0; i < 10; i++ {
        defer func() {
            println(i)
        }()
    }
}
```

```
func DeferClosureLoopV2() { 1 usage
    for i := 0; i < 10; i++ {
        defer func(val int) {
            println(val)
        }(i)
    }
}
```

```
func DeferClosureLoopV3() { no usages  new *
    for i := 0; i < 10; i++ {
        j := i
        defer func() {
            println(j)
        }()
    }
}
```

方法调用总结

- Go 支持多返回值，这是一个很大的不同点。
- Go 方法的作用域和变量作用域一样，通过大小写控制。
- Go 的返回值是可以有名字的，可以通过给予名字让调用方清楚知道你返回的是什么。
- Go 中方法是一等公民，所以函数式编程非常常见。在初学的时候，不需要掌握函数式编程，确保自己能够看得懂就可以。
- 闭包是指一个方法与跟着这个方法绑定的运行时刻上下文。初学的时候不要求掌握闭包用法，确保能看懂就行。面试的时候要能回答出来。
- defer 是先进后出，或者说后进先出。

Go 基础语法

- main 函数概览
- 基本类型和 string
- 变量与常量声明
- 方法声明与调用
- 控制结构

控制结构

- if - else
- for 循环
- switch
- select: 这个部分我们在课程后面再讲，它和并发相关。

控制结构：if - else

Go 的控制结构和别的语言差不多，都是if -else if -else 这种用法。

```
func IfOnly(age int) string { no usages
    if age >= 18 {
        return "成年了"
    }
    return "他还是一个孩子"
}

func IfElse(age int) string { no usages
    if age >= 18 {
        return "成年了"
    } else {
        return "他还是一个孩子"
    }
}

func IfElseIf(age int) string { no usages
    if age >= 18 {
        return "成年了"
    } else if age >= 12 {
        return "骚年"
    } else {
        return "他还是一个孩子"
    }
}

func IfElseIfV1(age int) string { no usages
    if age >= 18 {
        return "成年了"
    } else if age >= 12 {
        return "骚年"
    }
    return "他还是一个孩子"
}
```

控制结构：if - else

Go 的 if else 支持一种新的写法，可以在 if -else 块里面定义一个新的局部变量。

在右边 distance 的只作用于 if -else 块，离开了这个范围就无法使用了。

```
func IfNewVariable(start int, end int) string {  
    if distance := start - end; distance > 100 {  
        println(distance)  
        return "距离太远了"  
    } else {  
        println(distance)  
        return "距离比较近"  
    }  
    // 编译错误  
    //println(distance)  
}
```

控制结构：For 循环

for 有多种写法，第一种写法类似于别的语言的 for 循环。

这里循环输出了 0-9.

第二种写法展示了在 for 循环中，你可以不写第三段。

```
func Loop1() { 1 usage  new *
    for i := 0; i < 10; i++ {
        println(i)
    }
}

// 这样也可以
for i := 0; i < 10; {
    println(i)
    i++
}
```

控制结构：For 循环

第二种写法类似于别的语言中的 while 循环，基本语法是：

```
for condition {  
    // 循环内部操作  
}
```

```
func Loop2() {  
    i := 0  
    for i < 10 {  
        println(i)  
        i++  
    }  
}
```

控制结构：For 循环

for 后面可以不跟任何条件，那么就相当于 for true。

要谨慎使用，因为如果你没办法退出的话，**有可能引起 CPU 100%**

```
// Loop3 是无限循环
func Loop3() { 1 usage  new *
    for {
        println(args...: "hello")
    }
}
```

控制结构：For 循环

for 还有一种 for range 循环，它可以用于遍历数组、切片和 map。

- 遍历数组或者切片：
 - 使用两个迭代参数：第一个是下标，第二个是值
 - 使用单个迭代参数：下标
- 遍历 map：
 - 使用两个迭代参数：第一个是 key，第二个是 value
 - 使用单个迭代参数：key

它也可以用于 channel 遍历，我们后面学 channel 再说。

```
println(args...: "遍历数组")
arr := [3]string{"11", "12", "13"}
for i, val := range arr {
    println(i, val)
}
for i := range arr {
    println(i, arr[i])
}
```

```
println(args...: "遍历 map")
m := map[string]string{
    "key1": "value1",
    "key2": "value2",
}
for k, v := range m {
    println(k, v)
}
for k := range m {
    println(k, m[k])
}
```

控制结构：For 循环

千万不要对迭代参数取地址！！！

在内存里面，迭代参数都是放在一个同一个地方的，你循环开始就确定了，所以你一旦取地址，那么你拿到的就是这个地址。

所以，右边的 map 中的键值对的值，最终都是同一个，也就是最后一个。

```
func LoopBug() { 1 usage  new *
    users := []User{
        {
            Name: "Tom",
        },
        {
            Name: "Jerry",
        }
    }
    m := make(map[string]*User, 2)
    for _, u := range users {
        m[u.Name] = &u
    }

    for k, v := range m {
        fmt.Printf(format: "name: %s, user: %v", k, v)
    }
}
```

控制结构：For 循环

Go 也支持 break 和 continue 关键字。

- break: 中断循环
- continue: 立刻执行下一次循环

```
func LoopBreak() { no usages new *
    i := 0
    for {
        if i >= 10 {
            break
        }
        println(i)
        i++
    }
}
```

```
func LoopContinue() { no usages new *
    for i := 0; i < 10; i++ {
        if i%2 == 0 {
            continue
        }
        println(i)
    }
}
```

控制结构：Switch 语句

switch 语句和别的语言类似。基本语法形态是：

```
switch val {  
    case xxx:  
    case yyy:  
    default:  
}
```

default 分支可以有也可以没有。switch 的值必须是可比较的。实践中，不能用于 switch 的值，编译器会报错。

注意，switch 语句不需要 break。

```
func Switch(status int) string { no usage  
    switch status {  
        case 0:  
            return "初始化"  
        case 1:  
            return "运行中"  
        default: default 分支可以有，也可以没有  
            return "未知状态"  
    }  
}
```

控制结构：Switch 语句

switch 语句也可以没有 val:

```
switch {  
    case condition0:  
    case condition1:  
}
```

这种情况下，case 后面跟 bool 表达式。

从实践上来说，这种写法比较少见。正常使用这种写法，你应该尽量做到 case 的每一个条件是互斥的。



A screenshot of a code editor showing a Go language file. The code contains a function named `SwitchNoVal` that takes an integer parameter `age`. Inside the function, there is a `switch` statement. The first `case` block checks if `age >= 18`, and its body contains a `println` call with the argument `args...: "成年人"`. The second `case` block checks if `age < 6`, and its body contains a `println` call with the argument `args...: "小孩子"`. Both `case` blocks have a colon at the end, indicating they follow a boolean expression. The code editor has syntax highlighting and shows code completion suggestions.

```
func SwitchNoVal(age int) { no usages  
    switch {  
        case age >= 18:  
            println(args...: "成年人")  
        case age < 6:  
            println(args...: "小孩子")  
    }  
}
```

Go 内置类型

内置类型

- 数组
- 切片
- map
- channel: 这个后面讲并发的时候再讲

数组

数组和别的语言的数组差不多，语法是：

[cap]type

1. 初始化要指定长度（或者叫做容量）
2. 可以直接初始化
3. arr[i]的形式访问元素
4. len 和 cap 操作用于获取数组长度
5. 使用 for range 来循环

```
func Array() { 1 usage  new *
    // 直接初始化一个三个元素的数组，大括号里面只能少，不能多。
    a1 := [3]int{9, 8, 7}
    fmt.Printf(format: "a1: %v, len: %d, cap: %d", a1, len(a1), cap(a1))

    // 少了的部分就是默认零值，等价于 9, 8, 0
    a2 := [3]int{9, 8}
    fmt.Printf(format: "a2: %v, len: %d, cap: %d", a2, len(a2), cap(a2))

    // 虽然没有显式初始化，但是实际上内存已经分配好，等价于 0, 0, 0
    var a3 [3]int
    fmt.Printf(format: "a3: %v, len: %d, cap: %d", a3, len(a3), cap(a3))

    // 数组不支持 append 操作
    // a3 = append(a3, 1)

    // 按下标索引，如果编译器能判断出来下标越界，那么会编译错误，
    // 如果不能，那么运行时候会报错，出现 panic
    fmt.Printf(format: "a[1]:%d", a1[1])}
```

切片

切片，语法是：`[]type`

1. 直接初始化
2. `make` 初始化：`make([]type, length, capacity)`
3. `arr[i]` 的形式访问元素
4. `append` 追加元素
5. `len` 获取元素数量
6. `cap` 获取切片容量
7. 推荐写法：`s1 := make([]type, 0, capacity)`
8. 使用 `for range` 来遍历

```
func Slice() { no usages new *  
    s1 := []int{1, 2, 3, 4} //直接初始化了 4 个元素的切片  
    fmt.Printf("s1: %v, len: %d, cap: %d \n", s1, len(s1), cap(s1))  
  
    s2 := make([]int, 3, 4) //直接初始化了三个元素, 容量为 4 的切片  
    fmt.Printf("s2: %v, len: %d, cap: %d \n", s2, len(s2), cap(s2))  
  
    s2 = append(s2, elems...: 7) // 追加一个元素, 没有扩容  
    fmt.Printf("s2: %v, len: %d, cap: %d \n", s2, len(s2), cap(s2))  
  
    s2 = append(s2, elems...: 8) // 再追加一个元素, 扩容了  
    fmt.Printf("s2: %v, len: %d, cap: %d \n", s2, len(s2), cap(s2))  
  
    s3 := make([]int, 4) // make 只传入一个参数, 表示创建一个 4 个元素的切片  
    fmt.Printf("s3: %v, len: %d, cap: %d \n", s3, len(s3), cap(s3))  
  
    // 按照下标索引  
    fmt.Printf("s3[2]: %d", s3[2])  
    // 超出下标返回, panic  
    fmt.Printf("s3[2]: %d", s3[99])  
}
```

最佳实践：在初始化切片的时候要预估容量。

数组和切片的区别

	数组	切片
直接初始化	支持	支持
make	不支持	支持
访问元素	arr[i]	arr[i]
len	长度	已有元素个数
cap	长度	容量
append	不支持	支持
是否可以扩容	不可以	可以

遇事不决用切片！

如何理解切片？

最直观的对比：**ArrayList**，即基于数组的 List 的实现，切片的底层也是数组。

跟 ArrayList 的区别：

1. 切片操作是有限的，不支持随机增删（即没有add, delete方法，需要自己写代码）。
2. 只有append操作。
3. 切片支持子切片操作，和原本切片是共享底层数组。

子切片

数组和切片都可以通过[start:end] 的形式来获取子切片：

1. arr[start:end], 获得[start, end)之间的元素。
2. arr[:end], 获得[0, end)之间的元素。
3. arr[start:], 获得[start, len(arr))之间的元素。

都是左闭右开！

```
func SubSlice() { no usages new *
    s1 := []int{2, 4, 6, 8, 10}
    s2 := s1[1:3]
    fmt.Printf(format: "s2: %v, len: %d, cap: %d \n", s2, len(s2), cap(s2))

    s3 := s1[2:] // make 只传入一个参数, 表示创建一个 4 个元素的切片
    fmt.Printf(format: "s3: %v, len: %d, cap: %d \n", s3, len(s3), cap(s3))

    s4 := s1[:3] // make 只传入一个参数, 表示创建一个 4 个元素的切片
    fmt.Printf(format: "s4: %v, len: %d, cap: %d \n", s4, len(s4), cap(s4))
}
```

内存共享问题

核心：共享数组。

子切片和切片究竟会不会互相影响，就抓住一点：**它们是不是还共享数组？**

什么意思？

- 就是如果它们结构没有变化，那肯定是共享的；但是结构变化了，就可能不是共享了。

什么情况下结构会发生变化？**扩容了。**

所以，切片与子切片，切片作为参数传递到别的方法、结构体里面，任何情况下你要判断是否内存共享，那么就一个点：**有没有扩容。**

```
func ShareSlice() {  
  
    s1 := []int{1, 2, 3, 4}  
    s2 := s1[2:]  
    fmt.Printf(format: "s1: %v, len %d, cap: %d \n", s1, len(s1), cap(s1))  
    fmt.Printf(format: "s2: %v, len %d, cap: %d \n", s2, len(s2), cap(s2))  
  
    s2[0] = 99  
    fmt.Printf(format: "s1: %v, len %d, cap: %d \n", s1, len(s1), cap(s1))  
    fmt.Printf(format: "s2: %v, len %d, cap: %d \n", s2, len(s2), cap(s2))  
  
    s2 = append(s2, elems...: 199)  
    fmt.Printf(format: "s1: %v, len %d, cap: %d \n", s1, len(s1), cap(s1))  
    fmt.Printf(format: "s2: %v, len %d, cap: %d \n", s2, len(s2), cap(s2))  
  
    s2[1] = 1999  
    fmt.Printf(format: "s1: %v, len %d, cap: %d \n", s1, len(s1), cap(s1))  
    fmt.Printf(format: "s2: %v, len %d, cap: %d \n", s2, len(s2), cap(s2))  
}
```

map

map 是 Go 里面的内置类型之一，使用起来也很简单。

- 初始化：
 - make 方法：记得预估容量。
 - 直接初始化元素。
- 赋值：使用中括号。

```
m1 := map[string]string{  
    "key1": "value1",  
}
```

```
m2 := make(map[string]string, 4)  
m2["key2"] = "value2"
```

map

- 读取元素：有两个返回值
 - 第一个是值，第二个是这个元素是否存在。
 - 如果只用一个返回值，那么就是对应的元素；元素不存在，那么就是对应类型的零值。

```
m1 := map[string]string{
    "key1": "value1",
}

val, ok := m1["key1"]
if ok {
    println(args...: "第一步:", val)
}

val = m1["key2"]
println(args...: "第二步:", val)
```

map

- 读取长度: `len`
- 遍历: `for`, 第一个 is `key`, 第二个是 `value`
- 删除: 使用 `delete` 方法

注意: `map` 的遍历是随机的, 也就是说你遍历两遍,
输出的结果都不一样。

```
m2 := make(map[string]string, 4)
m2["key2"] = "value2"

println(len(m2))
for k, v := range m1 {
    println(k, v)
}

for k := range m1 {
    println(k)
}

delete(m1, key: "key1")
```

comparable 概念

- 在 switch 里面，值必须是可比较的。
- 在 map 里面，key 也必须是可比较的。

所谓可比较的（comparable）在 Go 里面就是指：Go 在编译的时候、运行的时候能够判断出来元素是不是相等。

初学的时候，你就知道：

- 基本类型和 string 都是可比较的。
- 如果元素是可比较的，那么该数组也是可比较的。

参考这里的详细说明：https://go.dev/ref/spec#Comparison_operators

面试也基本不可能面到这个内容，你有一个概念就可以。

代码演示

1. Go 是强类型语言，能不能设计一个方法，可以计算任意数字类型切片的和的方法？

- `func SumInt64([]int64)` 只能用于 `int64`
- `func SumInt32([]int32)` 只能用于 `int32`
- ...

2. 获得 map 的所有 key、所有 value。

Go 接口与类型定义

接口定义

基本语法： type 名字 interface {}

- 里面只能有方法，方法也不需要 func 关键字。

啥是接口（interface）： 接口是一组行为的抽象

- 尽量用接口，以实现面向接口编程。

我个人原则上认为，即便是业务开发，也应该面向接口编程。

```
5 I↑ I↓ type List interface { 1 usage 3 implementati
6   I↓      Add(index int, val any) 3 implementati
7   I↓      Append(val any) 3 implementations
8   I↓      Delete(index int) 3 implementations
9 } }
```

当你怀疑要不要定义接口的时候，加上去。

结构体定义

- 基本语法:
- `type Name struct {`
- `fieldName FieldType`
- `// ...`
- `}`

结构体和结构体的字段都遵循大小写控制访问性的原则。

通过`.`这个符号来访问字段或者方法。

```
// LinkedList 是一个链表
①↑ type LinkedList struct { 5 usages new *
  head *node
  //Head *node
}

// Add 添加一个元素
②↑ func (l *LinkedList) Add(index int, val any) {
  //TODO implement me
  ● panic(v: "implement me")
}
```

当你怀疑要不要定义接口的时候，加上去。

结构体初始化

- Go 没有构造函数！！
- 初始化语法：Struct{}
- 获取指针：
 - &Struct{}
 - new(Struct)

new 可以理解为 Go 会为你的变量分配内存，并且把内存都置为0。

```
// u1 是指向一个 User 对象的指针  
u1 := &User{}  
println(u1)  
  
// u2 中的字段都是零值  
u2 := User{}  
println(u2)  
// 修改 u2 的字段  
u2.Name = "Jerry"  
  
// u3 中的字段也都是零值  
var u3 User  
println(u3)
```

结构体字段初始化

有两种形态：

- 按照字段名赋值。
- 按照字段顺序赋值：不推荐使用，非常不利于后期维护。

```
// 初始化的同时，还赋值了 Name  
var u4 User = User{Name: "Tom"}  
println(u4)
```

```
// 没有指定字段名，按照字段顺序赋值  
// 必须全部赋值  
var u5 User = User{ Name: "Tom", Age: 18}  
println(u5)
```

结构体与指针

Go 的指针和别的语言的概念一样，本质上都是一个内存地址。

- 和 C、C++ 一样，***表示指针，&取地址。**
- 如果声明了一个指针，但是没有赋值，那么它是 nil。

```
func UseList() { no usages new *
    l1 := LinkedList{}
    l1Ptr := &l1
    var l2 LinkedList = *l1Ptr
    println(l2)

    // 这个是 nil
    var l3Ptr *LinkedList
    println(l3Ptr)
}
```

方法接收器

一个方法可以定义在这个结构体上，也可以定义在
这个结构体的指针上。

前者叫做结构体接收器，后者叫做指针接收器。

```
func (u User) ChangeName(name string) { 2 usages new *  
    fmt.Printf("u address %p\n", &u)  
    u.Name = name  
}  
  
func (u *User) ChangeAge(age int) { 2 usages new *  
    u.Age = age  
}
```

方法接收器

结构体接收器内部修改字段，不会生效。

因为方法调用本身是值传递的。

目前没有统一的说法什么时候用什么接收器，你在初学的时候：遇事不决用指针。

也就是：

- 如果是基本类型，结构体，那么就相当于复制了一份。
- 如果是指针，那么就复制了一份指针（也就是地址），但是指向的结构体还是同一个。
- 内置类型复制了一份，但是它们的内部数据结构，还是同一个。（初学你就这么理解）

```
func ChangeUser() { no usages new *
    u1 := User{Name: "Tom", Age: 18}
    fmt.Printf("#{u1} \n")
    fmt.Printf(format: "u1 address %p \n", &u1)

    u1.ChangeName(name: "Jerry")
    u1.ChangeAge(age: 35)
    fmt.Printf("#{u1} \n")

    u2 := &User{Name: "小明", Age: 18}
    fmt.Printf("#{u2} \n")
    fmt.Printf("u2 address #{u2} \n")
```

```
u2.ChangeName(name: "Jerry")
u2.ChangeAge(age: 35)
fmt.Printf("#{u2} \n")
```

指针可以调用结构体方法，反过来也可以。

结构体自引用

如果在结构体内部还要引用自身，那么只能使用指针。

准确来说，在整个引用链上，如果构成循环，那就只能用指针。

如果引用链路很长，你比较难发现这种错误。

```
type node struct { 2 usages new *  
    next *node  
    // 自引用不用指针会编译错误  
    //next node  
}
```

衍生类型

- 基本语法: `type TypeA TypeB`

我一般在想使用第三方库，又没有办法修改源码，又想扩展这个库的结构体的方法的情况下，就会用这个。

记住核心：衍生类型是一个全新的类型。

衍生类型可以互相转换，使用 `()` 进行转换。

注意，`TypeB` 实现了某个接口，不等于 `TypeA` 也实现了某个接口。

```
func UseFish() { no usages new *
    f1 := Fish{}
    f1.Swim()
    f2 := FakeFish{}
    // f2 将不能调用 Fish 上的方法,
    // 因为 f2 是一个全新的类型
    f2.FakeSwim()

    // 类型转换
    f3 := Fish(f2)
    f3.Swim()
```

类型别名

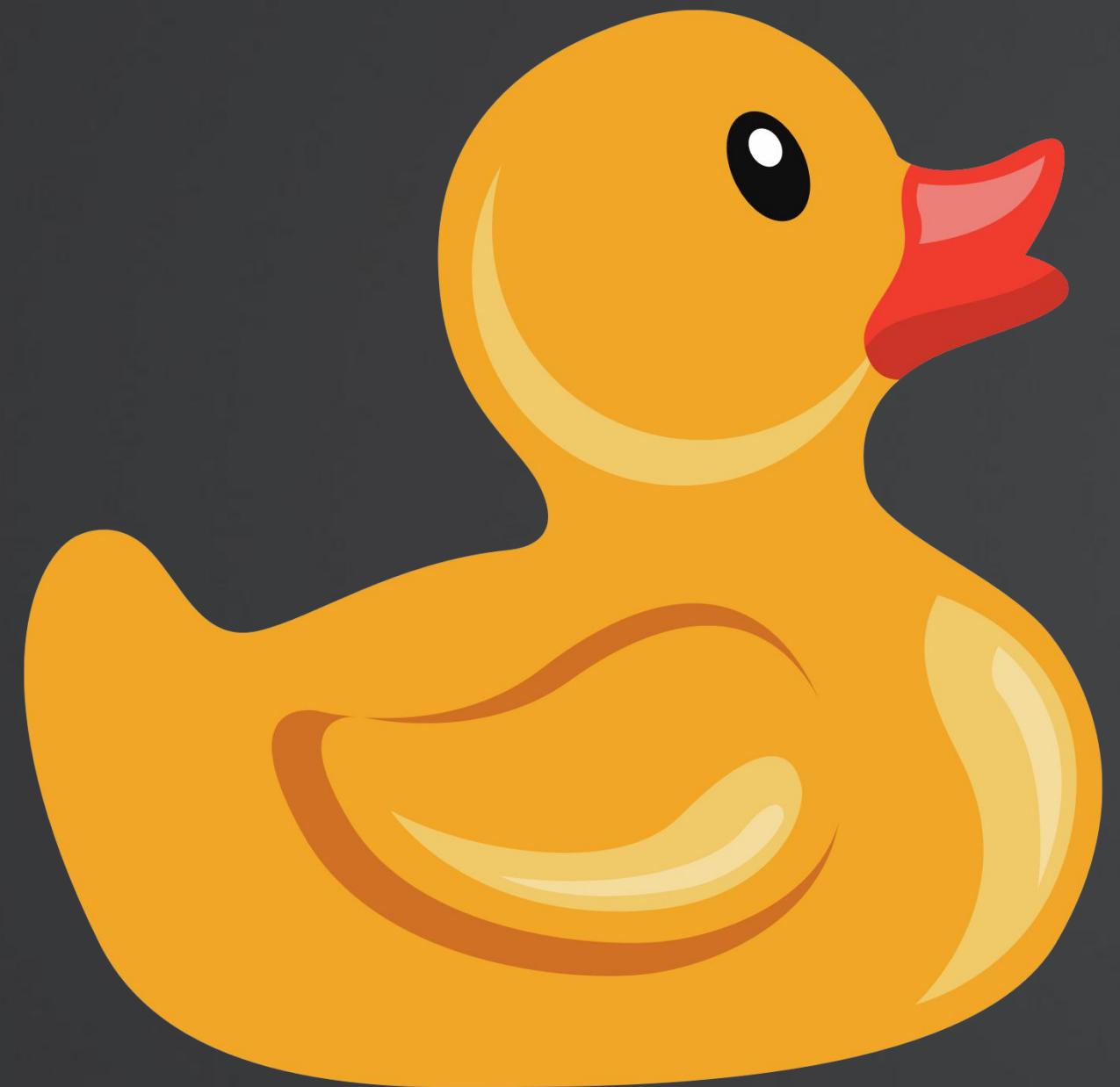
- 基本语法：`type TypeA = TypeB`
 - 别名，除了换了一个名字，没有任何区别
 - 它和衍生类型的区别，就是用了 =

类型别名一般用在导出类型、兼容性修改里面，也不常见。

```
// Yu 鱼
// 鱼是 Fish 的别名
type Yu = Fish 2 usages ne
```

```
y := Yu{}  
y.Swim()
```

结构体实现接口



当看到某个东西走起来像鸭子、游泳起来像鸭子、叫起来也像鸭子，那么这个东西就可以被称为鸭子。

当一个结构体具备接口的所有的方法的时候，它就实现了这个接口。

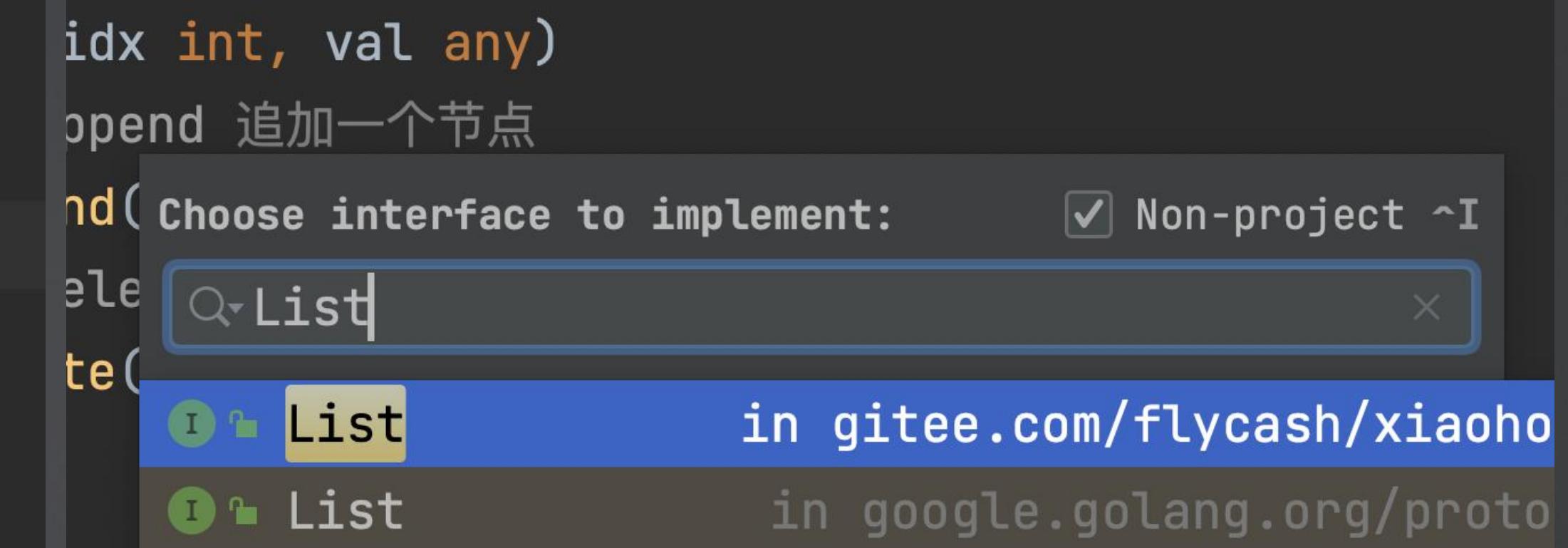
结构体实现接口

```
// LinkedList 公开类型
type LinkedList struct { 1 usage  new *
    head *node
}

// node 私有
type node struct {
    next *n
}
```

Generate

- Constructor
- Getter and Setter
- Getter
- Setter
- Struct Fields from JSON
- Implement Methods... ^I**
- Empty test file
- Copyright



结构体实现接口

```
// LinkedList 公开类型
↑ type LinkedList struct { 4 usages  new *
    head *node
}

↑ func (l *LinkedList) Add(idx int, val any) { new *
    //TODO implement me
    panic(v: "implement me")
}

↑ func (l *LinkedList) Append(val any) { new *
    //TODO implement me
    panic(v: "implement me")
}

↑ func (l *LinkedList) Delete(idx int) { new *
    //TODO implement me
    panic(v: "implement me")
}
```

组合

组合是 Go 里面独有的语法概念。基本语法形态是：

```
type A struct {  
    B  
}
```

组合可以是以下几种情况：

- 接口组合接口
- 结构体组合结构体
- 结构体组合结构体指针
- 结构体组合接口

可以组合多个。

```
type Outer struct { no usages }  
type Inner  
type Outer1 struct { no usages }  
    *Inner  
type Inner struct { 2 usages }  
type Outer2 struct { no usages }  
    // 组合了接口  
    io.Closer  
}
```

组合的特性

- 当 A 组合了 B 之后：
 - 可以直接在 A 上调用 B 的方法。
 - B 实现的所有接口，都认为 A 已经实现了。
- A 组合 B 之后，在初始化 A 的时候将 B 看做普通字段来初始化。
- 组合不是继承，没有多态。

右边的例子输出什么？

- hello, Inner
- hello, Outer

```
① func (o Outer) Name() string { no usage
    return "Outer"
}

func (i Inner) SayHello() { 1 usage   new *
    println("hello," + i.Name())
}

② func (i Inner) Name() string { 1 usage
    return "Inner"
}

func UseOuter() { no usages   new *
    var o Outer
    o.SayHello()
}
```

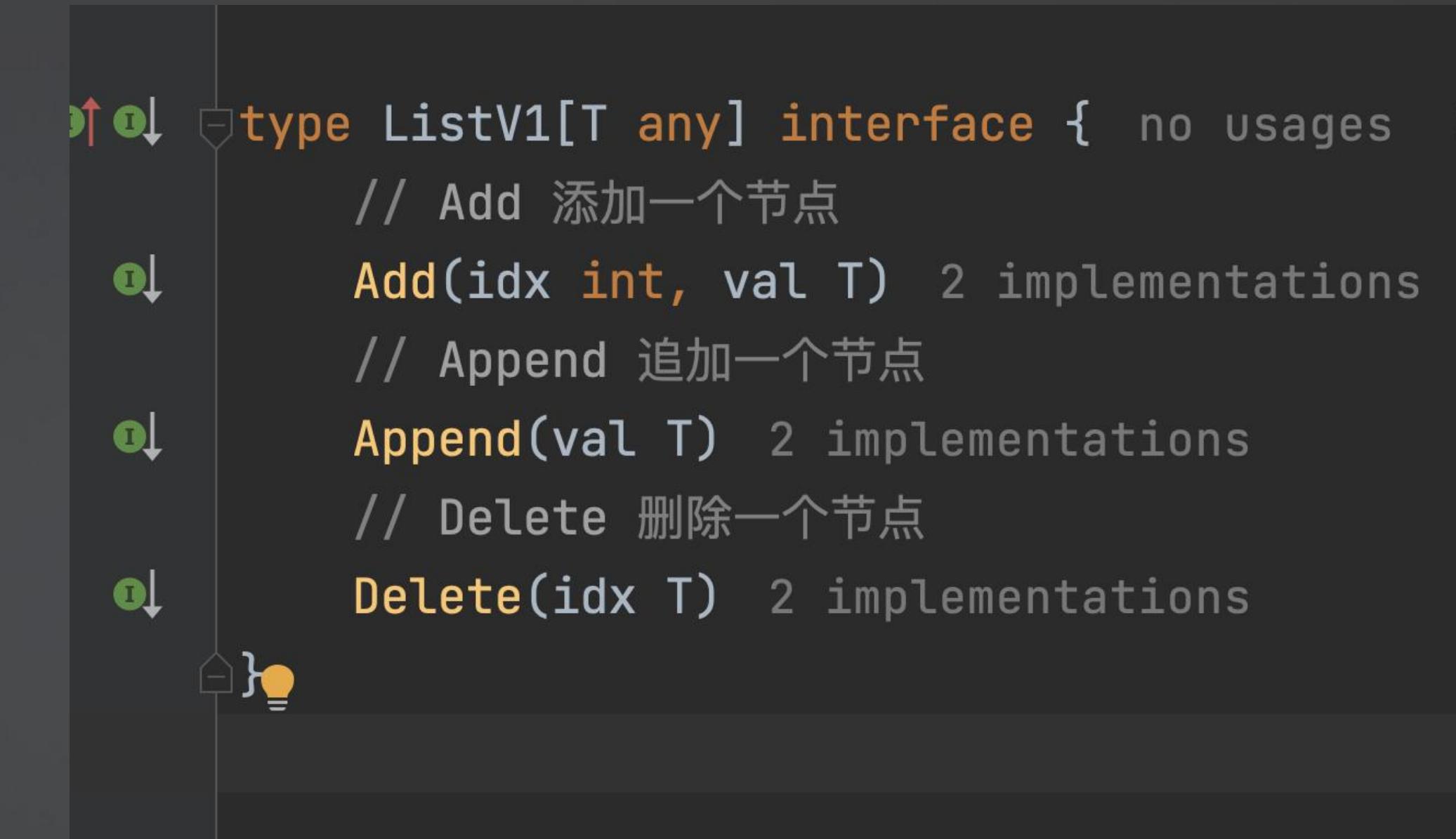
Java 转的同学尤其要小心。

泛型

泛型语法

Go 的泛型语法很简单。

右边是一个泛型接口，其中类型参数 T 可以是任意类型。



```
type ListV1[T any] interface { no usages :>
    // Add 添加一个节点
    Add(idx int, val T) 2 implementations
    // Append 追加一个节点
    Append(val T) 2 implementations
    // Delete 删除一个节点
    Delete(idx T) 2 implementations
}
```

泛型语法

结构体也可以使用泛型。

```
↑ ↴ type LinkedListV1[T any] struct { 3 usages  new *
    head *nodeV1[T]
}

↑ ↴ func (l *LinkedListV1[T]) Add(idx int, val T) { new *
    //TODO implement me
    panic(v: "implement me")
}

↑ ↴ func (l *LinkedListV1[T]) Append(val T) { new *
    //TODO implement me
    panic(v: "implement me")
}

↑ ↴ func (l *LinkedListV1[T]) Delete(idx T) { new *
    //TODO implement me
    panic(v: "implement me")
}
```

泛型语法

方法也可以使用泛型。

Sum 是一个泛型方法，其中 Number 是一个泛型约束。

当然，普通的接口也可以用作泛型约束，可以直接操作接口的方法。

```
func Sum[T Number](data ...T) T { no usages new *
    var res T
    for _, v := range data {
        res = res + v
    }
    return res
}

type Number interface { 1 usage new *
    int | uint
}
```

代码演示

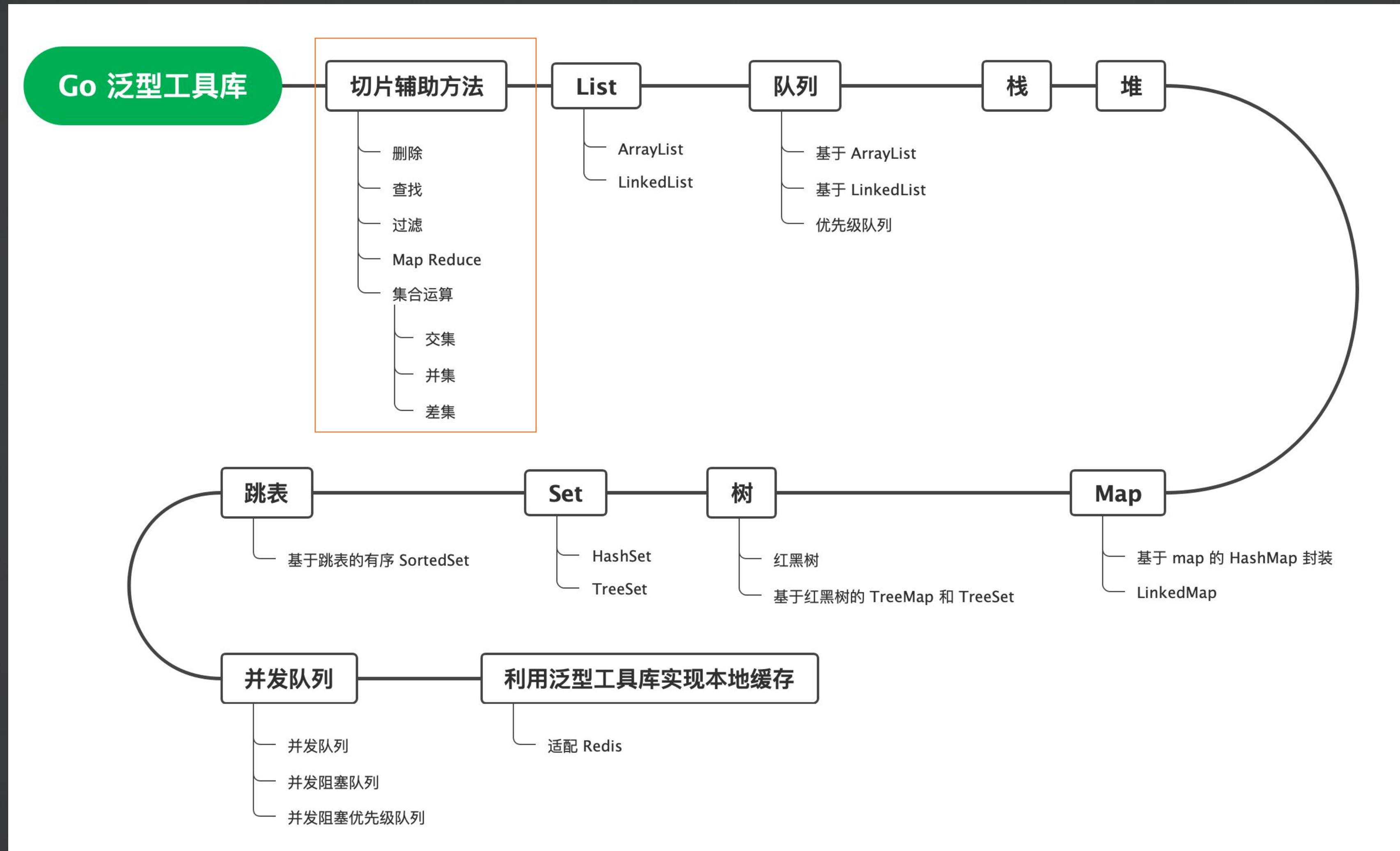
用泛型实现以下方法：

- 求和
- 求最大值、最小值
- 过滤和查找
- 在切片特定索引位置插入元素。

标准实现参考：<https://github.com/ecodeclub/ekit/tree/dev/slice>

升职加薪指南

基于泛型的 Go 工具库



基于泛型的 Go 工具库

你最容易做到的，就是利用 Go 的泛型来实现一个综合的泛型工具库。它一方面可以让你在公司内部用泛型来改造已有的 Go 代码，另外一方面可以开源出去。**不会写不会设计，经过各种思考查找之后都还不会，就直接来问我，顺利毕业之后找我看简历的时候，记得把这个仓库写完，我会教你刷亮点。**

其中并发工具和复杂的数据结构，你拿出去面试的时候可以证明你的并发能力和数据结构与算法的能力。

- 内置类型的辅助方法：切片、map 等的辅助方法
- 并发工具：这个难度比较高
- 实现复杂的数据结构：
 - 链表：可以进一步封装成队列
 - 红黑树：可以用于封装成 TreeMap、TreeSet
 - 跳表
 - 多叉树：包括 B 树、B+ 树

熟练掌握 GO 语言。|

项目经历

GO 泛型工具库

在 GO 出来之后，我在公司内部研发了一个 GO 泛型工具库，用于重构在业务代码中的冗余、模板代码。该工具库提供了高性能的辅助方法和数据结构，包括：

- 切片的辅助方法：添加、删除、查找，求并集、交集，map reduce API
- map 辅助方法
- 扩展 map 实现：接受任意类型的 HashMap, TreeMap, LinkedMap
- List 实现：LinkedList、ArrayList 和 SkipList
- Set：包括 HashSet 和 TreeSet, SortedSet
- 队列：普通队列、优先级队列
- bean 操作辅助类：高性能高扩展的 bean copier 机制
- 并发扩展工具：包括并发队列、并发阻塞队列、并发阻塞优先级队列
- 协程池

面试要点

面试要点

- 什么是闭包？闭包有什么缺陷？
- 什么情况下会出现栈溢出？
- 什么是不定参数？调用方法的时候，不定参数可以传入 0 个值吗？方法内部怎么使用不定参数？
- 什么是 defer？你能解释一下 defer 的运作机制吗？
- 一个方法内部 defer 能不能超过 8 个？
- defer 内部能不能修改返回值？怎么改？
- 数组和切片有什么区别？
- 切片怎么扩容的？

面试：defer 实现机制

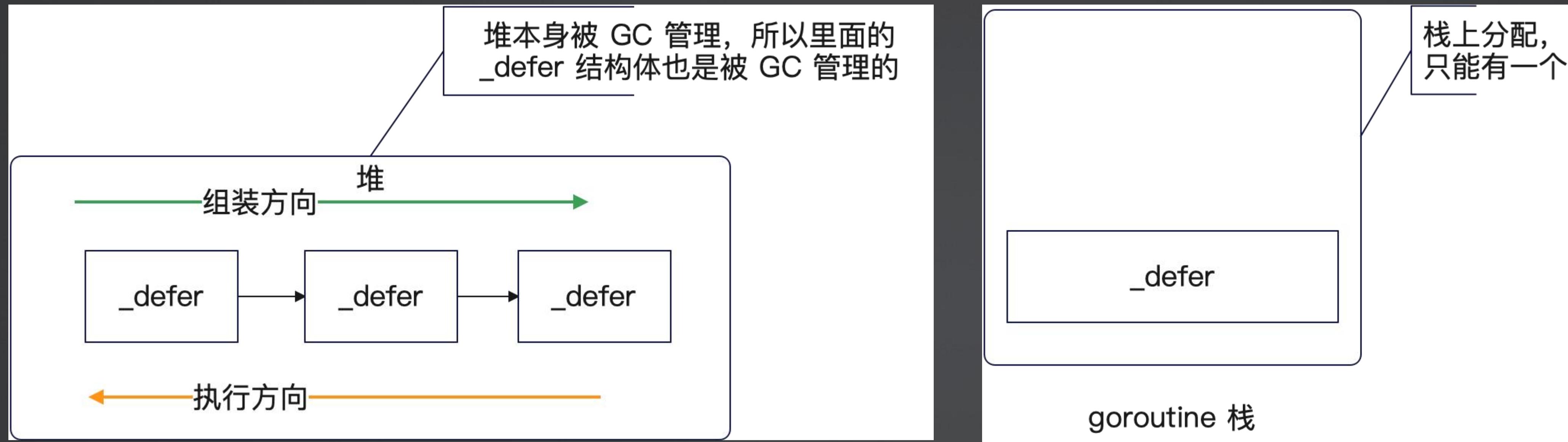
defer 实现机制

在面试问到了 defer 的时候，你要主动回答这些内容。它能让你拉开和面试者的差距。

defer 的内部实现分成三种机制：

- **堆上分配**：是指整个 defer 直接分配到堆上，缺点就是要被 GC 管理。
- **栈上分配**：整个 defer 分配到了 goroutine 栈上，不需要被 GC 管理。比堆上分配性能提升了 30%。
- **开放编码（Open Code）**：启用内联的优化，你直观理解就是相当于把你的 defer 内容放到了你的函数最后。启用条件：
 - 函数的 defer 数量少于或者等于 8 个；
 - 函数的 defer 关键字不能在循环中执行；
 - 函数的 return 语句与 defer 语句的乘积小于或者等于 15 个。

defer 堆上分配与栈上分配



defer 开放编码

开放编码就类似于编译器帮你把 defer 的内容挪到了你方法的最后。

The diagram illustrates the concept of "open coding" (开放编码) in Go. It shows two versions of the same code. The top version is the original source code:

```
func DeferOpenCode() { no usages new *
    defer func() {
        println(args...: "hello, world")
    }()
}
// 假如说这个是你的业务代码
YourBusiness()
```

The bottom version shows the code after compilation, where the `defer` statement has been moved to the end of the function body:

```
// 开放编码就类似于编译器帮你把 defer 的内容放过来了这里|
//func() {
//    println("hello, world")
//}
}
```

A red circle highlights the text "编译器帮你挪下来" (The compiler helps you move it down), and a red arrow points from this text to the moved `println` statement in the bottom code.

defer 开放编码

如何理解启用条件？

- 函数的 defer 数量少于或者等于 8 个：用了一个 byte 来记录哪些 defer 要执行。
- 函数的 defer 关键字不能在循环中执行：编译的时候不知道有多少个 defer。
- 函数的 return 语句与 defer 语句的乘积小于或者等于 15 个：一个硬性规定。

```
func DeferConditional(input bool) { no usages new *  
    if input {  
        defer func() {  
            println(args...: "hello, world")  
        }()  
    }  
}
```

这个 defer 可能执行，也可能不执行。

你要想面出来亮点，就要一路回答到这里。同时注意从 defer 的栈上分配和堆上分配引导过去内存逃逸相关的内容上。

面试思路

你的回答要分成几个部分：

- defer 的作用
- defer 的底层实现原理。这算是你在初级工程师面试中刷亮点的地方：
 - 堆上分配与栈上分配：这个地方你要尝试引导到内存逃逸、GO 性能优化这种话题上。
 - 开放编码。

切片：扩容

切片扩容

切片扩容的原理非常简单，就是重新分配一段连续内存，而后把原本的数据拷贝过去。

```
func SliceExtend() { 1 usage  new *
    s1 := []int{1, 2, 3, 4} //直接初始化了 4 个元素的切片
    fmt.Printf(format: "s1: %v, len: %d, cap: %d \n", s1, len(s1), cap(s1))

    s1 = append(s1, elems...: 5)
    fmt.Printf(format: "s1: %v, len: %d, cap: %d \n", s1, len(s1), cap(s1))
}
```

底层是长度为 4 的数组

1	2	3	4
---	---	---	---



底层是长度为 8 的数组

1	2	3	4	5			
---	---	---	---	---	--	--	--

切片扩容

在面试中，刷亮点的地方在于你要进一步解释切片扩容的系数。

总结为：

- 当容量小于 256 的时候，两倍扩容；
- 否则，按照 1.25 扩容。

低版本是 1024 作为分界点。

面试你要强调几个点：

- 为什么一开始是两倍扩容，后面是 1.25 倍扩容？
- 为什么低版本是 1024 作为分界点，而高版本是 256？

还有一个问题是，面试官可能会问为什么切片没有缩容？如果你已经实现了泛型工具库，那么你可以说你提供了切片的删除辅助方法，而后引入了缩容机制。

作业

实现切片的删除操作

实现删除切片特定下标元素的方法。

- 要求一：能够实现删除操作就可以。
- 要求二：考虑使用比较高性能的实现。
- 要求三：改造为泛型方法。
- 要求四：支持缩容，并且设计缩容机制。

THANKS