

极客时间Go初级工程师第一课

Golang 基础语法和 Web 框架起步

大明



# 关于本课程

- 掌握 Go 的基本语法
  - 类型、方法、控制结构
  - 接口与结构体
  - 基本库与并发编程入门
- 掌握 Go Web 框架设计要点
  - 如何设计路由树
- 掌握 Go 常用的设计模式

# 如何学习

- 上课听讲——有条件的跟着一起写代码
- 课后练习——编程不过是熟能生巧

多练习！！！！！！

# 目录

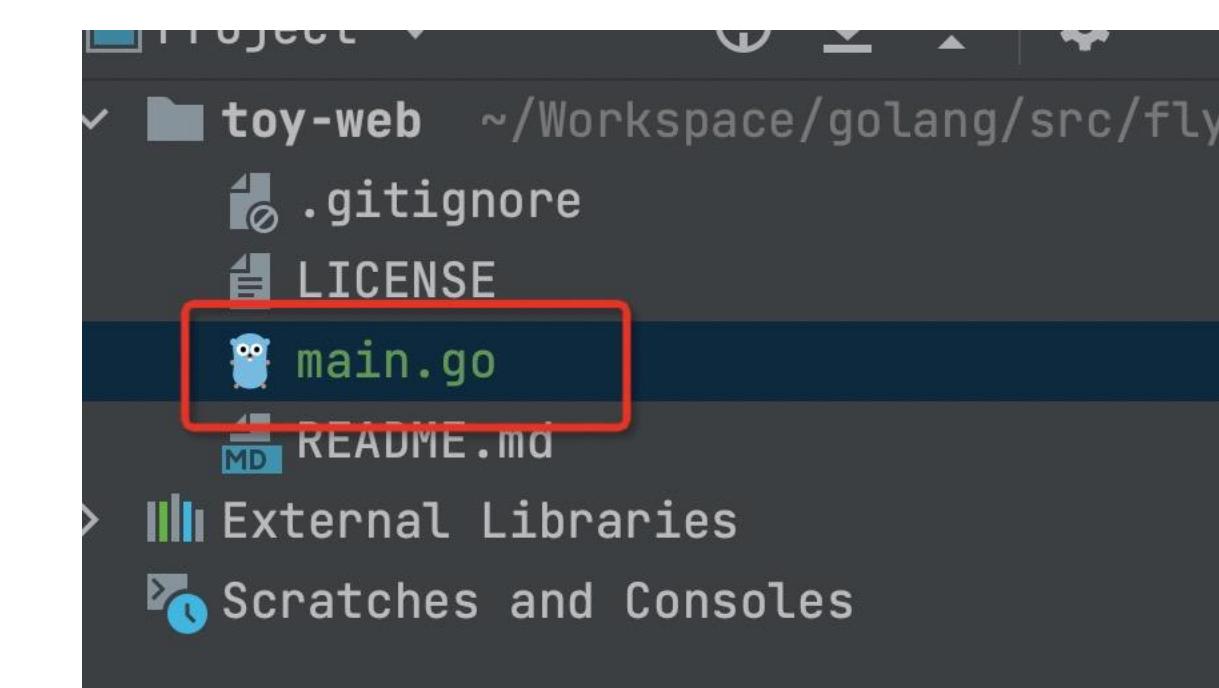
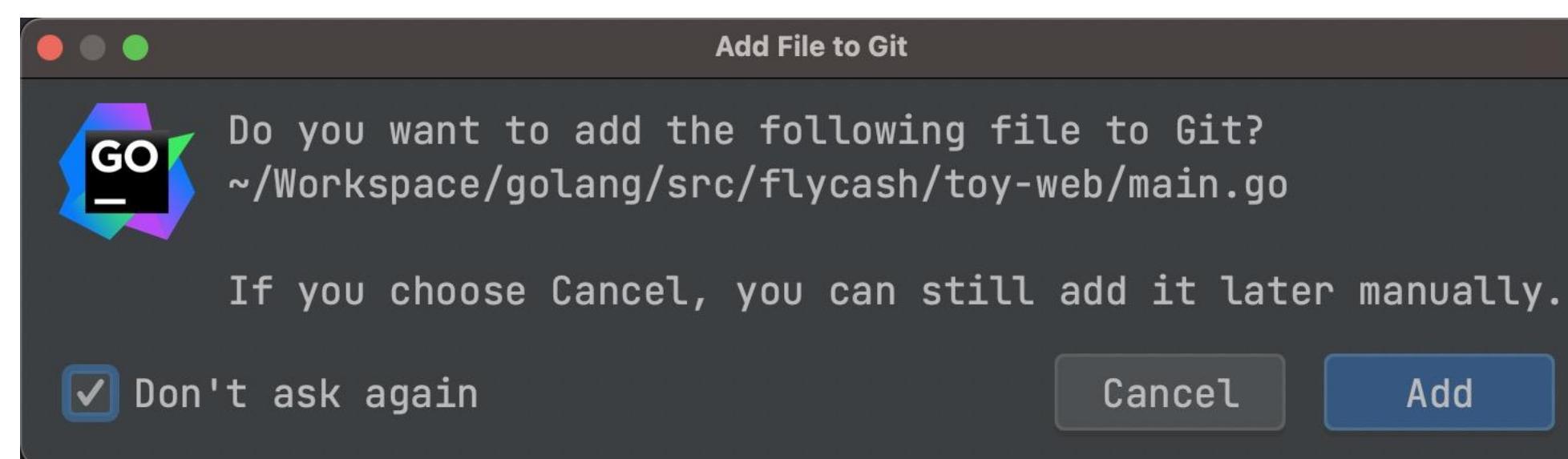
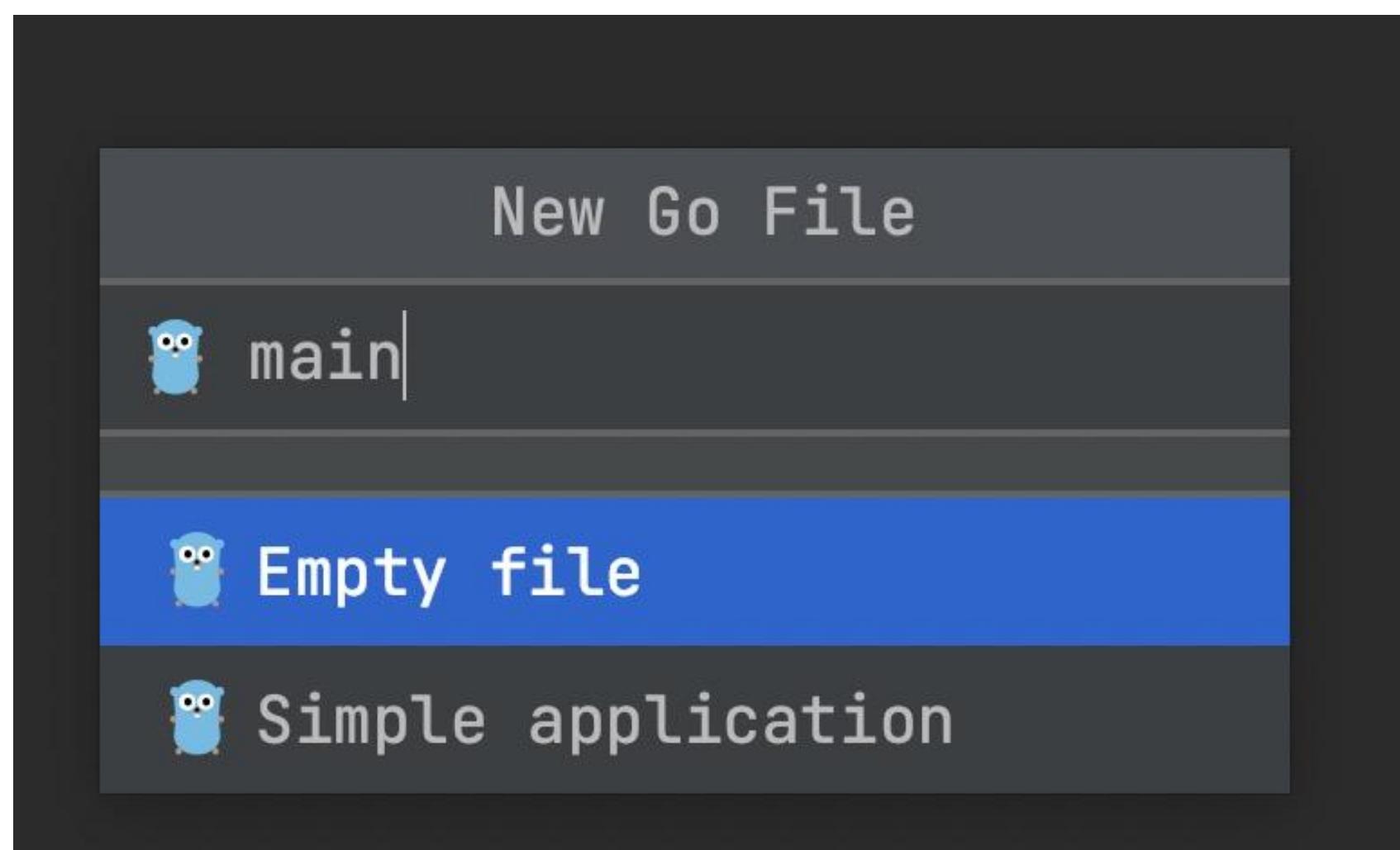
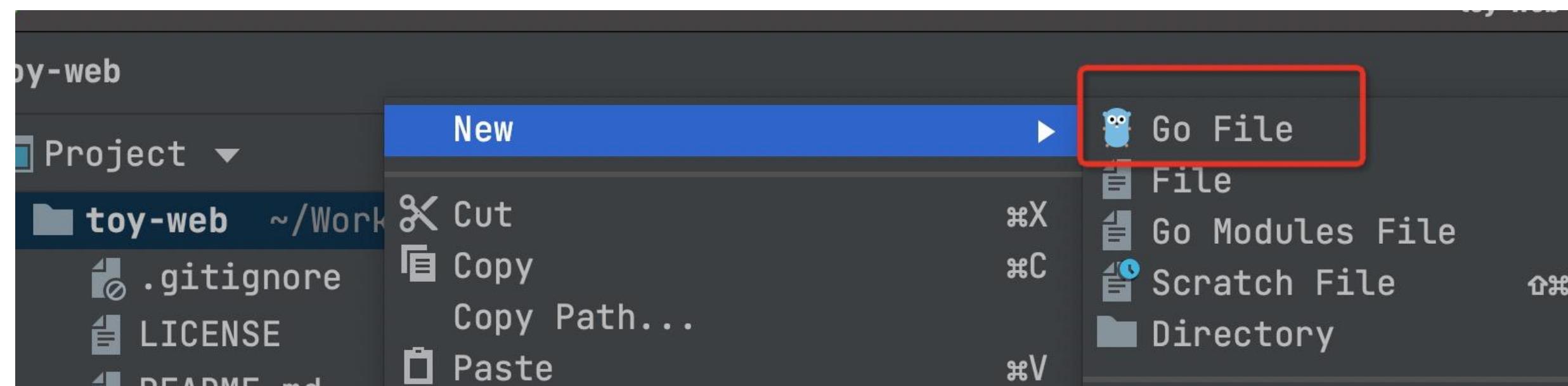
1. 环境安装
2. 创建项目
3. Hello Go!
4. 基础语法
5. 最简单的 Web 服务器

# 目录

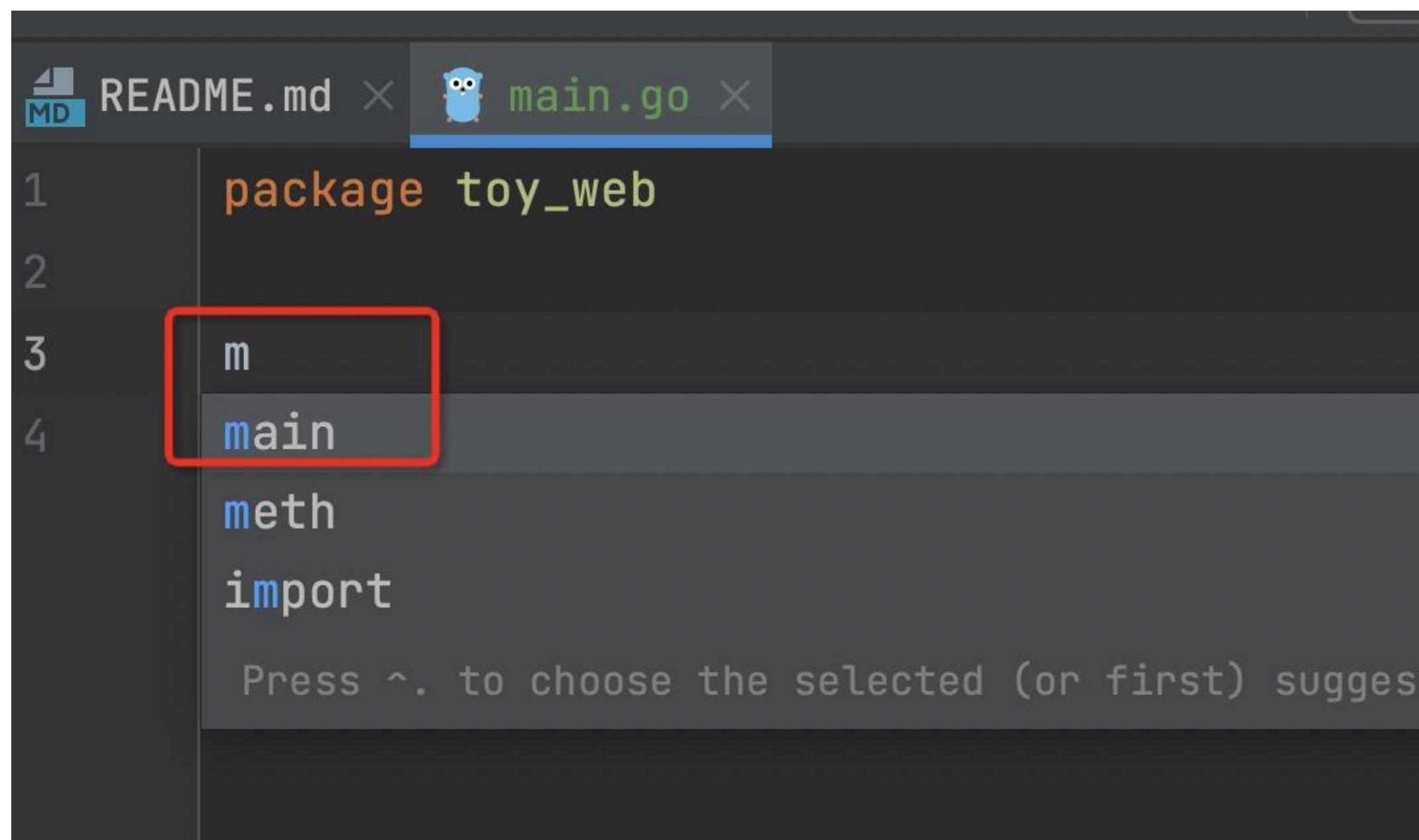
1. 环境安装
2. 创建项目
3. Hello Go!
4. 基础语法
5. 最简单的 Web 服务器

# Hello Go! — 第一个 Go 程序

- 创建 main.go 文件



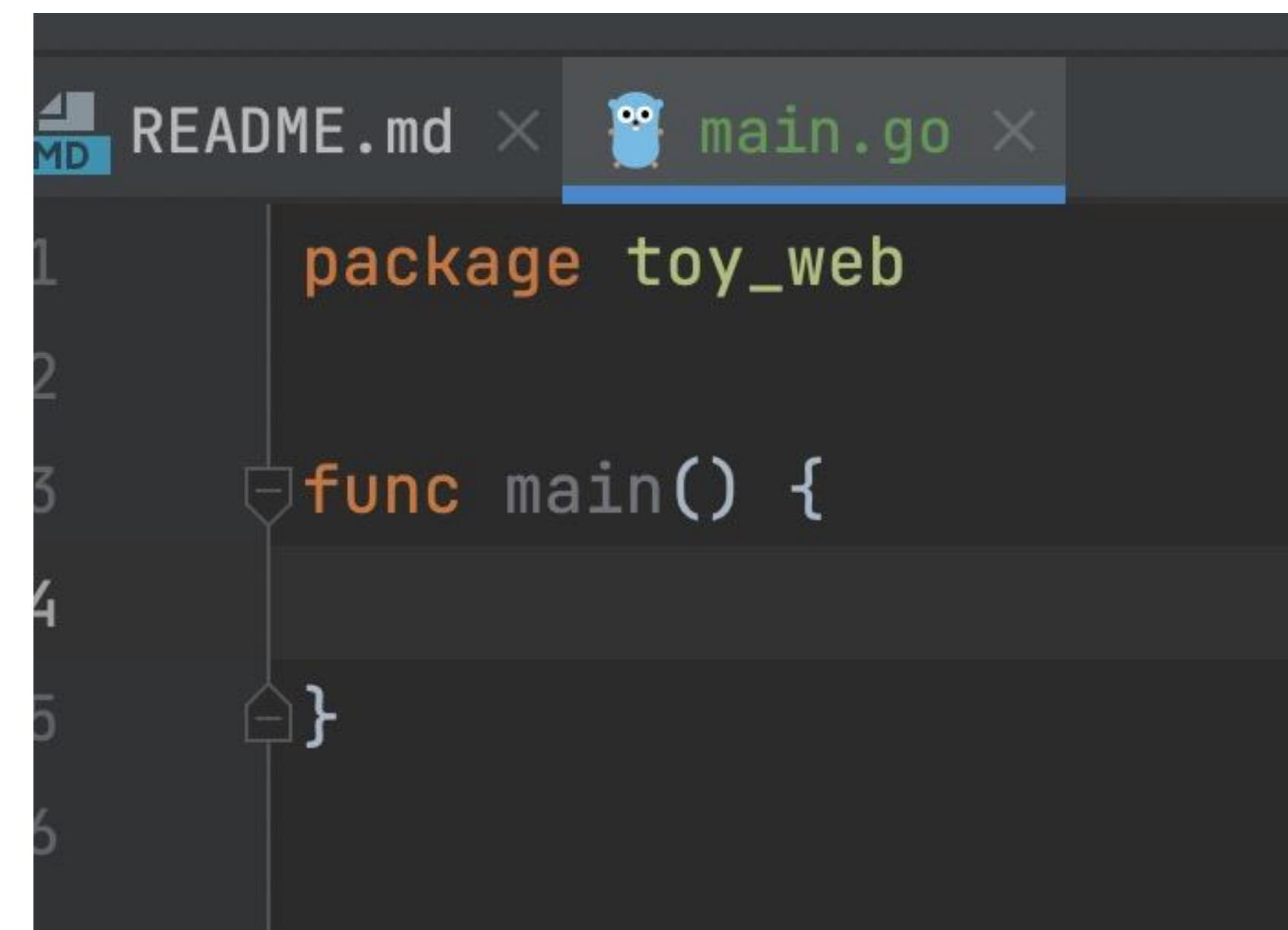
# Hello Go! — 写下第一行代码



A screenshot of a Go code editor showing a code completion dropdown menu. The menu contains suggestions: 'm', 'main', 'meth', and 'import'. The 'main' suggestion is highlighted with a red border. The code editor interface shows a file named 'main.go' with the following content:

```
1 package toy_web
2
3 m
4 main
5 meth
6 import
7
8 Press ^ to choose the selected (or first) suggestion
```

输入 m, 按下 enter  
依赖于goland的强大提示



A screenshot of the same Go code editor after the user has entered 'm' and pressed 'enter'. The code now includes a 'func main()' function definition. The code editor interface shows a file named 'main.go' with the following content:

```
1 package toy_web
2
3 func main() {
4 }
```

# Hello Go! — 输出 Hello, Go

```
func main() {  
    p  
    f panic(v interface{})  
    f print(args ...Type) print(args ...Type) println(args ...Type)  
    printf
```

输入 pr, 按下 enter

```
README.md × main.go ×  
package toy_web  
func main() {  
    print(args...: "Hello, Go!")  
}
```

# Hello Go! — 第一次运行 Go

```
doc      show documentation for package or symbol
env      print Go environment information
fix      update packages to use new APIs
fmt      gofmt (reformat) package sources
generate generate Go files by processing source code
get      add dependencies to current module and download them
install  compile and install packages and dependencies
list     list packages or modules
mod      module maintenance
run      compile and run Go program
test     test packages
tool     run specified go tool
version  print Go version
vet      report likely mistakes in packages
```

```
→ toy-web git:(main) ✘ go run main.go
go run: cannot run non-main package
→ toy-web git:(main) ✘
```

字面意思：无法在非main包下运行

# Hello Go! — 第一次运行 Go

```
1 package toy_web 改成 main 试试
2
3 func main() {
4     print(args...: "Hello, Go!")
5 }
6
```

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

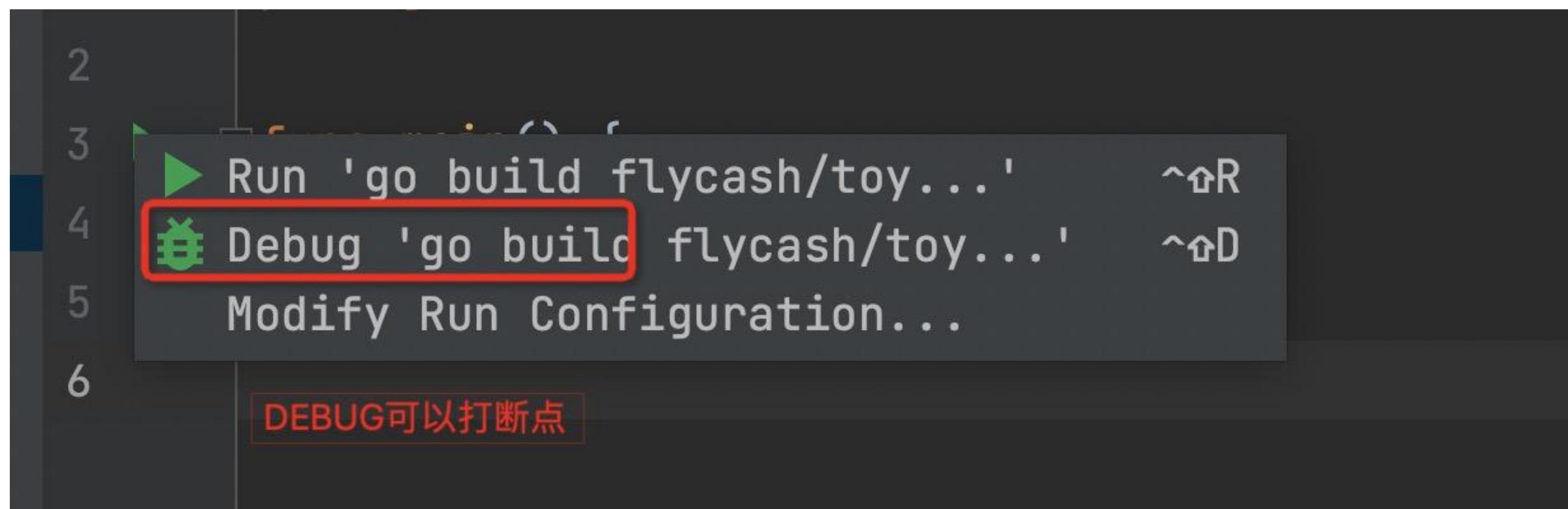
# Hello Go! — 第一次运行 Go

```
1 package toy_web 改成 main 试试
2
3 func main() {
4     print(args...: "Hello, Go!")
5 }
6
```

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

```
→ toy-web git:(main) ✘ go run main.go
Hello, Go!%
→ toy-web git:(main) ✘
```

# Hello Go! — IDE 运行程序



```
API server listening at: [::]:57608
debugserver-@(#)PROGRAM:LLDB  PROJECT:lldb
for x86_64.

Got a connection, launched process /private/tmp/ld-2018-03-13-144444-11444/ld-2018-03-13-144444-11444

Hello, Go! Exiting.

Debugger finished with the exit code 0
```

# 目录

1. 环境安装
2. 创建项目
3. Hello Go!
4. 基础语法
5. 最简单的 Web 服务器

# 目录

1. main 函数概览
2. package 声明
3. string 和 基础类型
4. 变量声明
5. 方法声明与调用

# 基础语法 — main 函数概览

```
1 package main 包声明
2 方法关键字
3 ► func main() {
4     print( args...: "Hello, Go!" ) 输出语句
5 }
6 和别的语言一样使用小括号和大括号来组织代码
```

# 基础语法 — main 函数要点

- 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
- 如果文件不叫 `main.go`， 则需要 `go build` 之后再 `go run`



```
package main
func main() {
    print(args...: "Hello, Go!")
}
```

# 目录

1. main 函数概览
2. package 声明
3. string 和 基础类型
4. 变量声明
5. 方法声明与调用

# 基础语法 — 包声明

- 语法形式: package xxxx
- 字母和下划线的组合
- 可以和文件夹不同名字
- 同一个文件夹下的声明一致



The screenshot shows the Goland IDE interface with a dark theme. At the top, there are tabs for 'README.md' and 'main.go'. The 'main.go' tab is active, indicated by a blue underline. Below the tabs, the code editor displays the following Go code:

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

The code editor uses color coding: 'package' and 'func' are orange, while 'main' and 'print' are green. Line numbers 1 through 6 are visible on the left side of the editor.

Tip: 使用Goland来创建文件夹，它会自动加上package

# 基础语法 — 包声明

- 引入包语法形式: `import [alias] xxx`
- 如果一个包引入了但是没有使用, 会报错
- 匿名引入: 前面多一个下划线



```
import (
    "fmt"
    ! "net/http"
    _ "strings"
)
```

The screenshot shows a code editor window with a dark theme. An import statement is displayed: `import ("fmt" ! "net/http" _ "strings")`. The package `"net/http"` is underlined with a red wavy line and has a red exclamation mark icon next to it, indicating that the package is imported but not used in the current file.

Tip: 除了匿名引用, Goland 会帮你自动引入你代码里面用得包, 在你不用之后也会删除

# 目录

1. main 函数概览
2. package 声明
3. string 和 基础类型
4. 变量声明
5. 方法声明与调用

# 基础语法 — string声明

- string:
  - 双引号引起来，则内部双引号需要使用\转义
  - `引号引起来，则内部`需要\转义



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

```
func main() {
    // 一般推荐用于短的，不用换行的，不含双引号的
    print(args...: "He said:\\" Hello Go \"")
    // 长的，复杂的。比如说放个 json 串
    print(args...: `He said: "hello, Go"
我还可以换个行
`)
}
```

The code demonstrates two ways to declare strings: using double quotes (") and using backticks (`). The backticks version allows for multi-line strings and JSON-like escaping.

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

# 基础语法 — string 长度

- 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

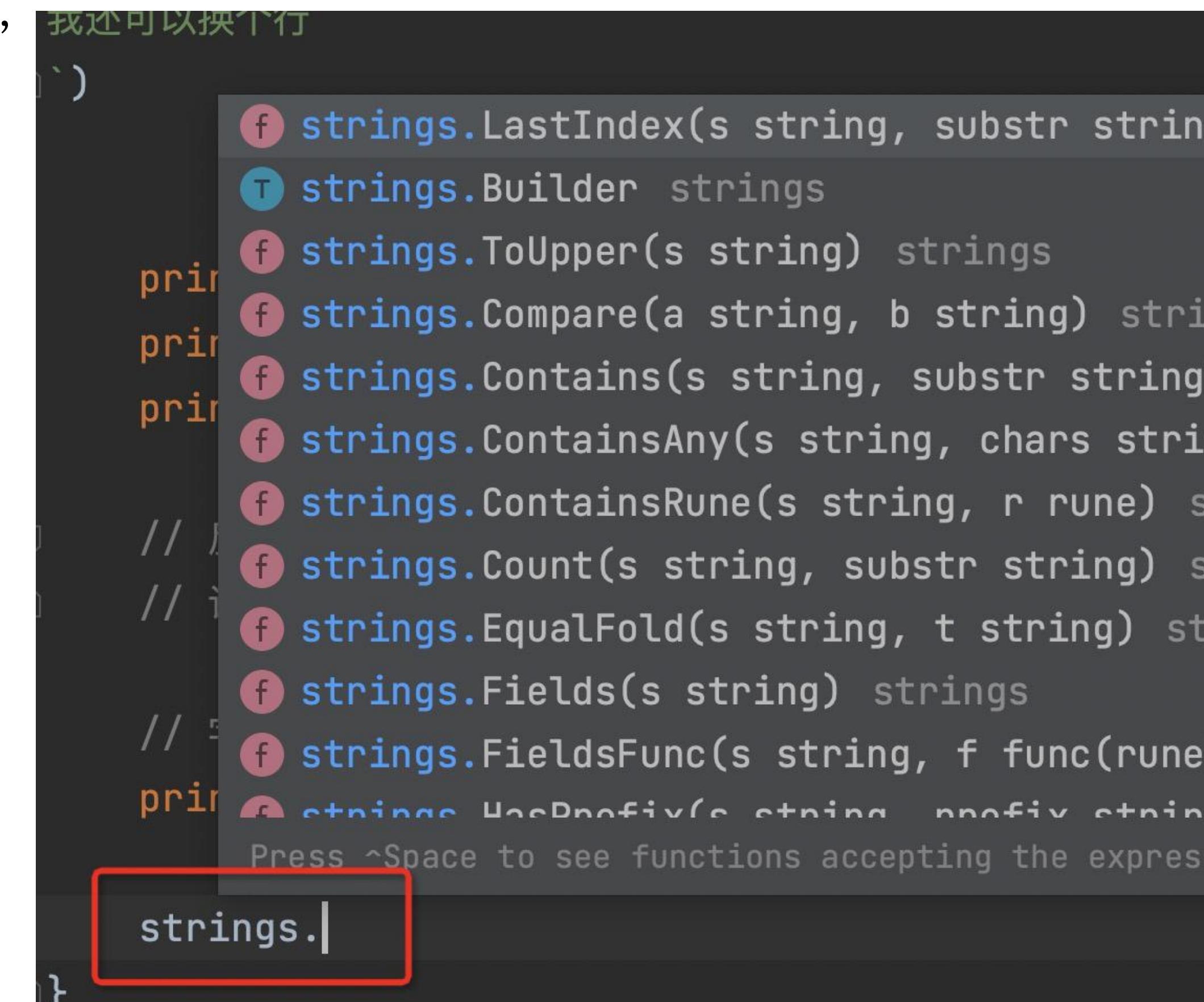
```
println(len(v: "你好")) // 输出6
println(utf8.RuneCountInString(s: "你好")) // 输出 2
println(utf8.RuneCountInString(s: "你好ab")) // 输出 4

// 反正遇到计算字符个数，比如说用户名多长，博客多长这种字符个数
// 记得用 utf8.RuneCountInString
```

Tip: 如果你觉得字符串里边会出现非 ASCII 的字符，就记得用 utf8 库来计算“长度”

# 基础语法 — strings 包

- string 的拼接直接使用 + 号就可以。注意的是，某些语言支持 string 和别的类型拼接，但是 golang 不可以
- strings 主要方法（你所需要的全部都可以找到）：
  - 查找和替换
  - 大小写转换
  - 子字符串相关
  - 相等



Tip: 同样不需要死记硬背，依赖于 Goland 的提示

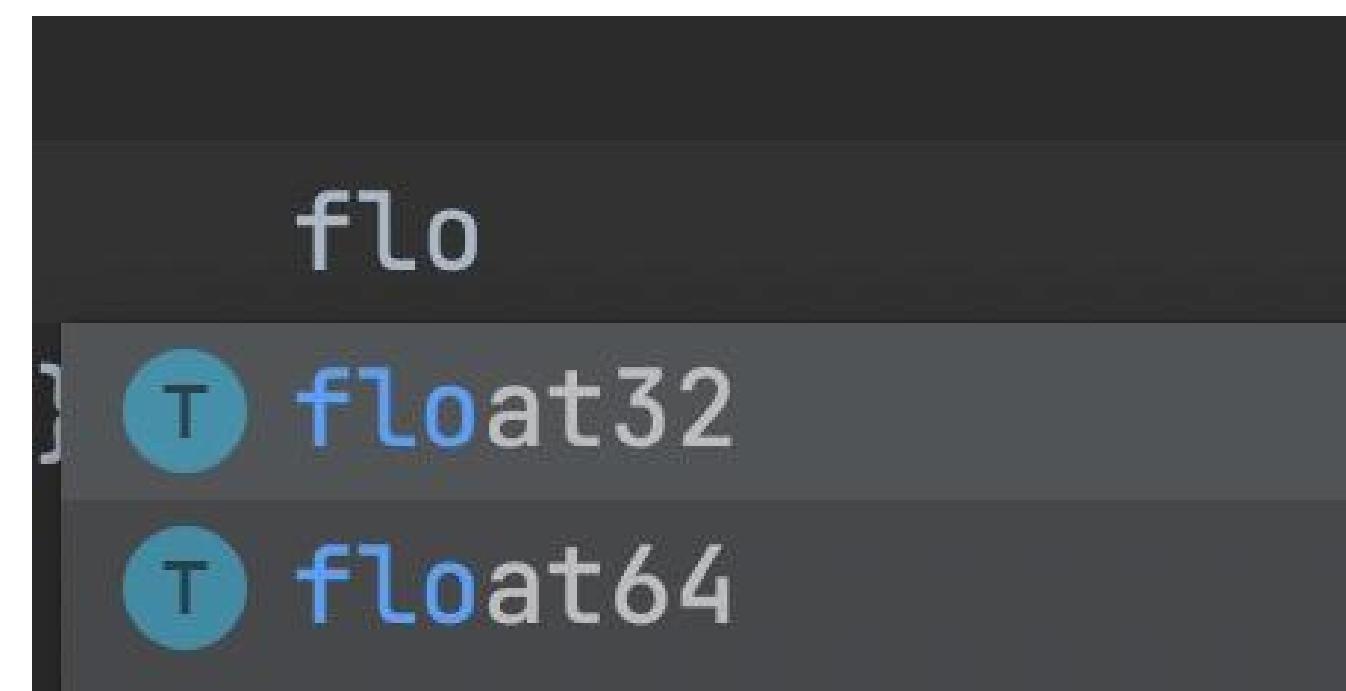
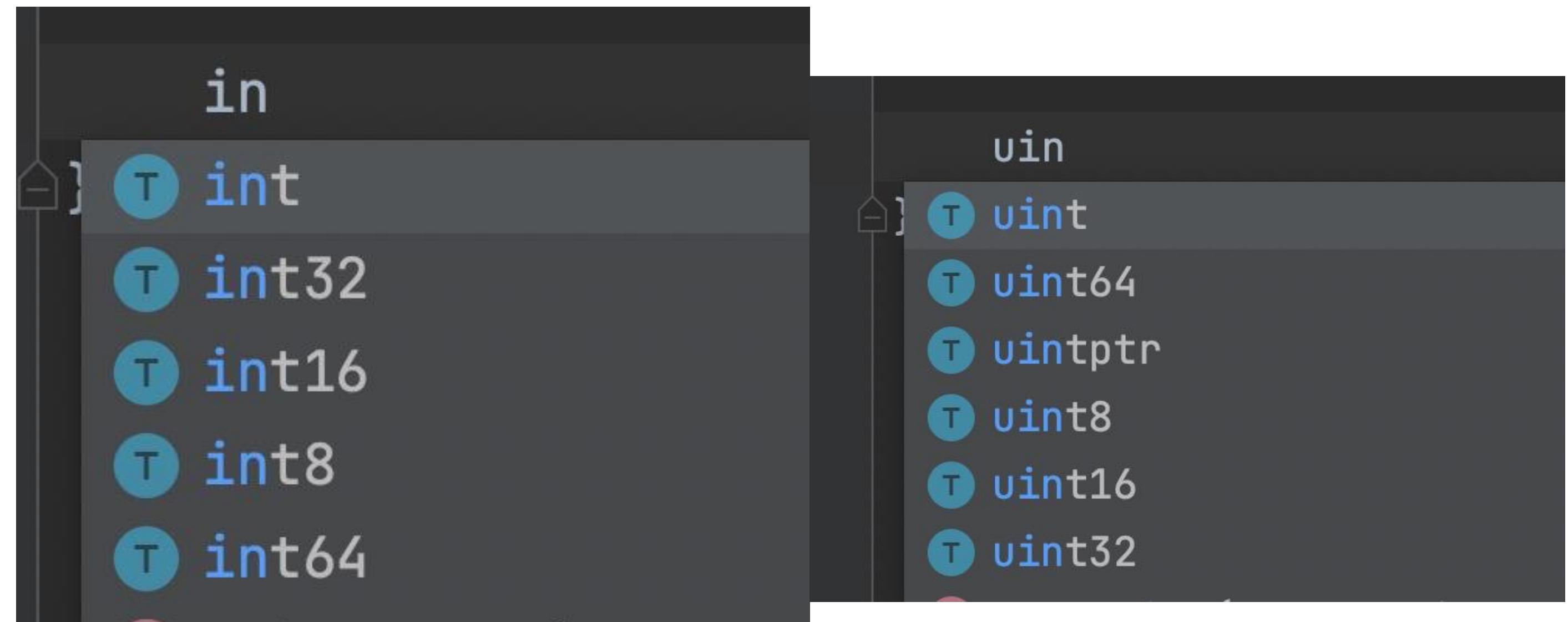
# 基础语法 — rune 类型

- rune, 直观理解, 就是字符
- rune 不是 byte!
- rune 本质是 int32, 一个 rune 四个字节
- rune 在很多语言里面是没有的, 与之对应的是, golang 没有 char 类型。rune 不是数字, 也不是 char, 也不是 byte!
- 实际中不太常用

```
// 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
```

# 基础语法 ——bool, int, uint, float 家族

- bool: true, false
- int8, int16, int32, int64, int
- uint8, uint16, uint32, uint64, uint
- float32, float64



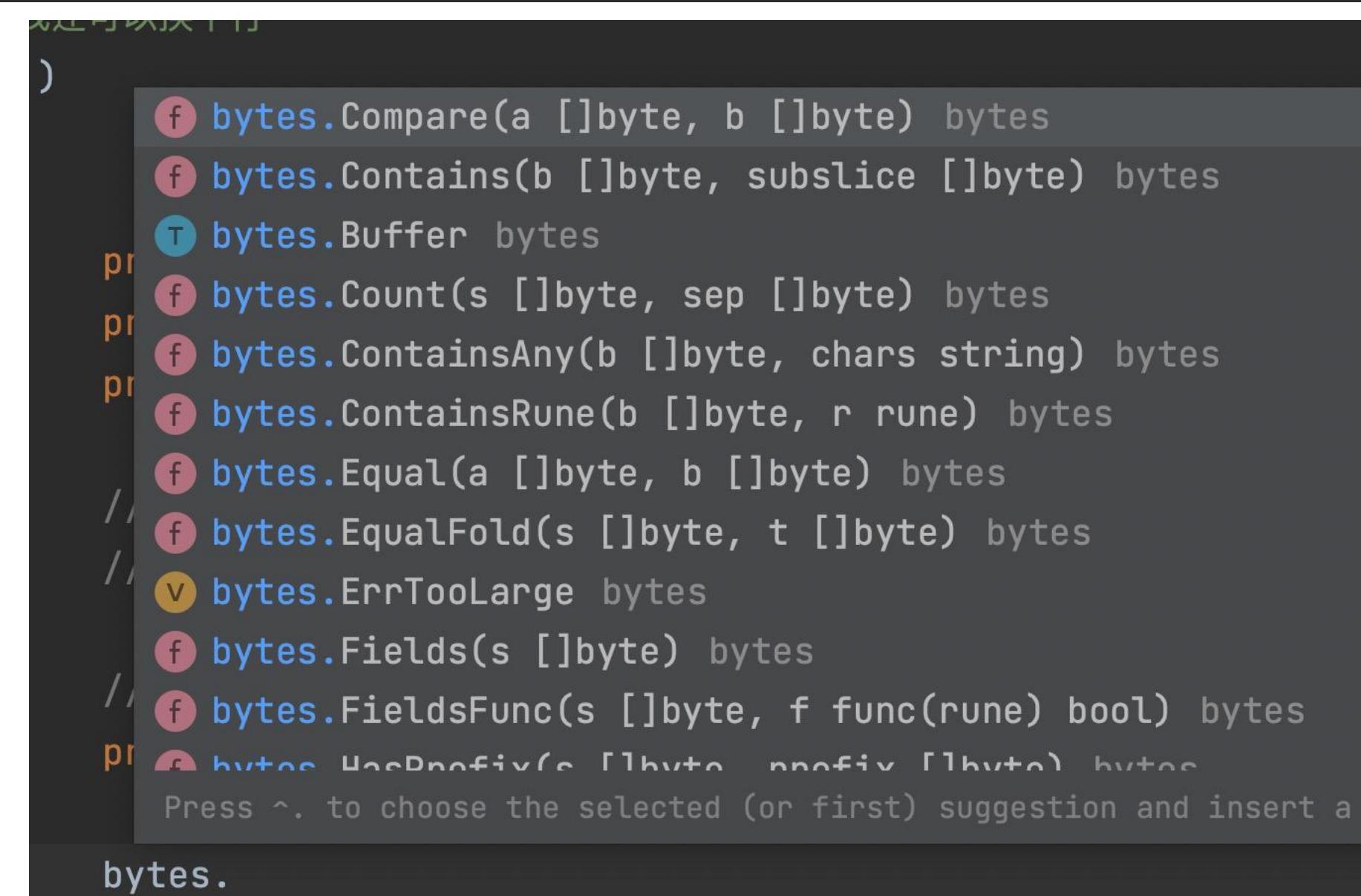
Tip: 不必死记硬背，依赖于 GoLand 的提示

# 基础语法 — byte 类型

- byte, 字节, 本质是 uint8
- 对应的操作包在 bytes 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
```



# 基础语法 — 类型总结

- golang 的数字类型明确标注了长度、有无符号
- golang 不会帮你做类型转换，类型不同无法通过编译。也因此，string 只能和 string 拼接
- golang 有一个很特殊的 rune 类型，接近一般语言的 char 或者 character 的概念，非面试情况下，可以理解为 “rune = 字符”
- string 遇事不决找 strings 包

# 目录

1. main 函数概览
2. package 声明
3. string 和 基础类型
4. 变量声明
5. 方法声明与调用

# 基础语法 — 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

```
// 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
)
```

```
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)

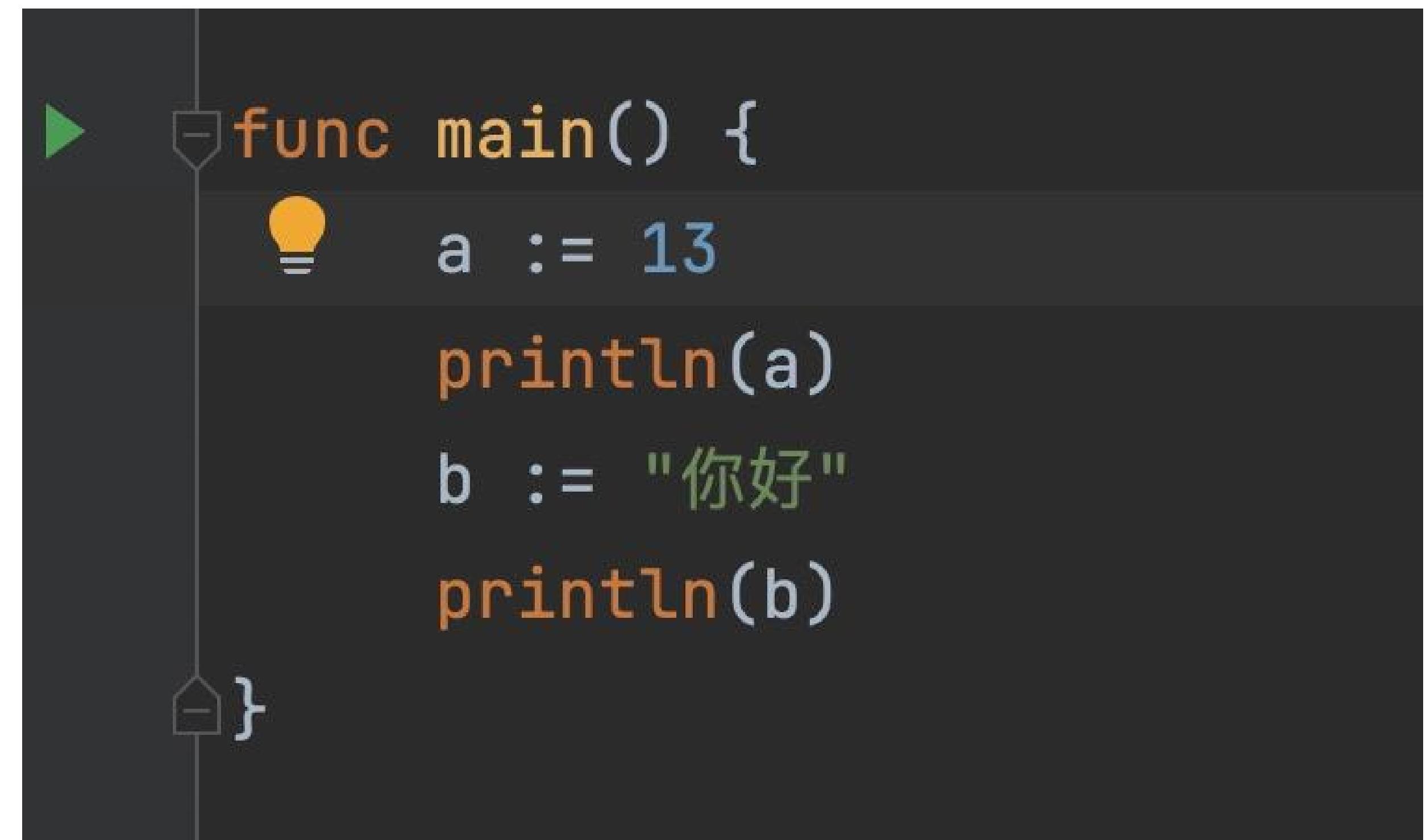
    // 这里我们省略了类型
    var b = 14
    println(b)

    // 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)
}
```

# 基础语法 — 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 consists of a single function named 'main'. Inside the function, there are two variable declarations: 'a' is assigned the value '13' and 'b' is assigned the string value '\"你好\"'. Both assignments use the shorthand assignment operator ':=' which allows for type inference. The code is syntax-highlighted, with 'func' and 'println' in orange, variable names in blue, and strings in green. A yellow lightbulb icon is placed next to the first assignment line, likely indicating a code completion or suggestion. The code editor has a dark theme.

# 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，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)

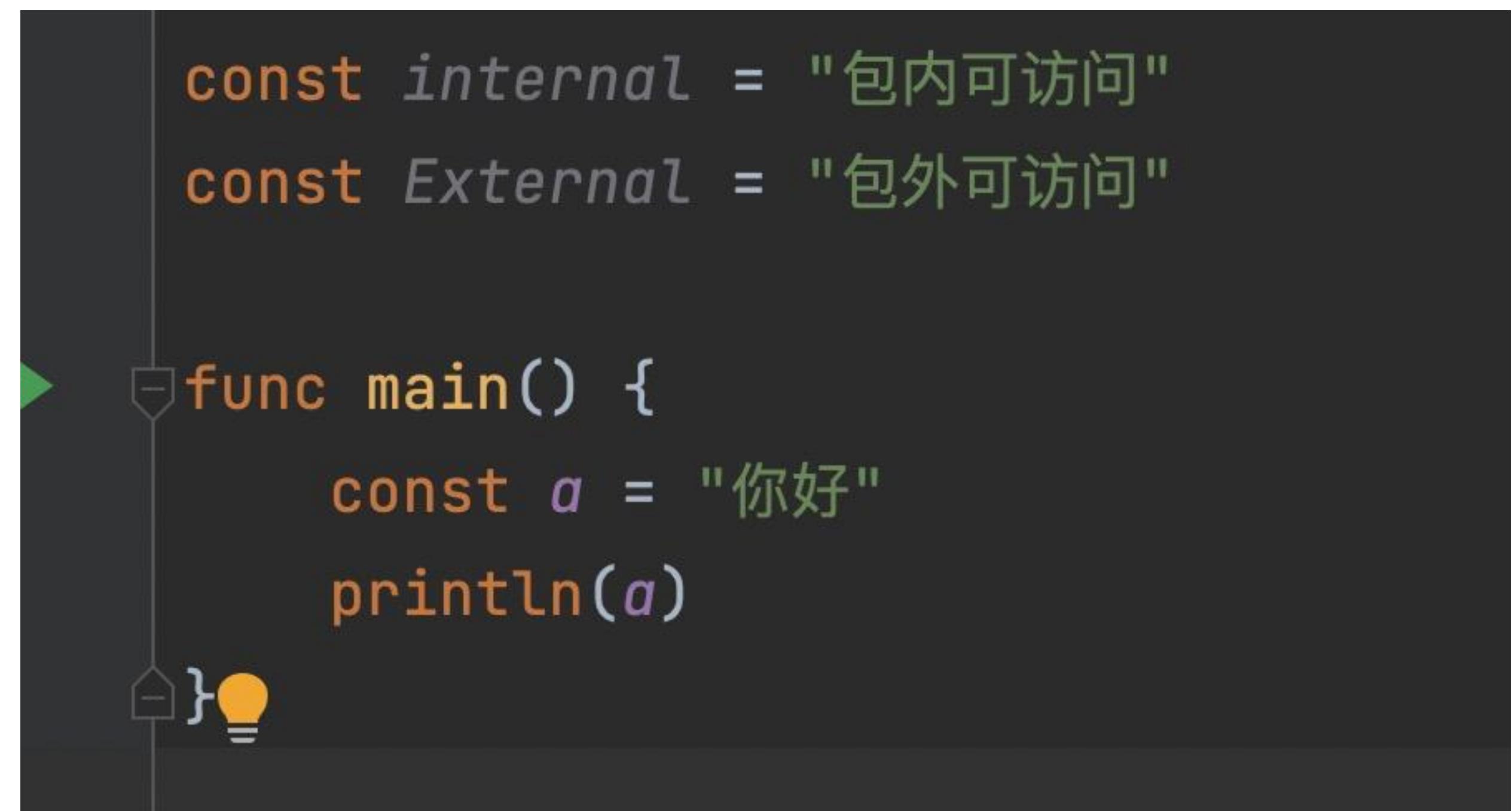
      var bb = 15
      //var bb = 16 // 重复声明，也会导致编译不通过
      println(bb)

      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
    }
```

Tip: 不必死记硬背，编译不了了看 Goland IDE 的提示

# 基础语法 — 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"
const External = "包外可访问"

func main() {
    const a = "你好"
    println(a)
}
```

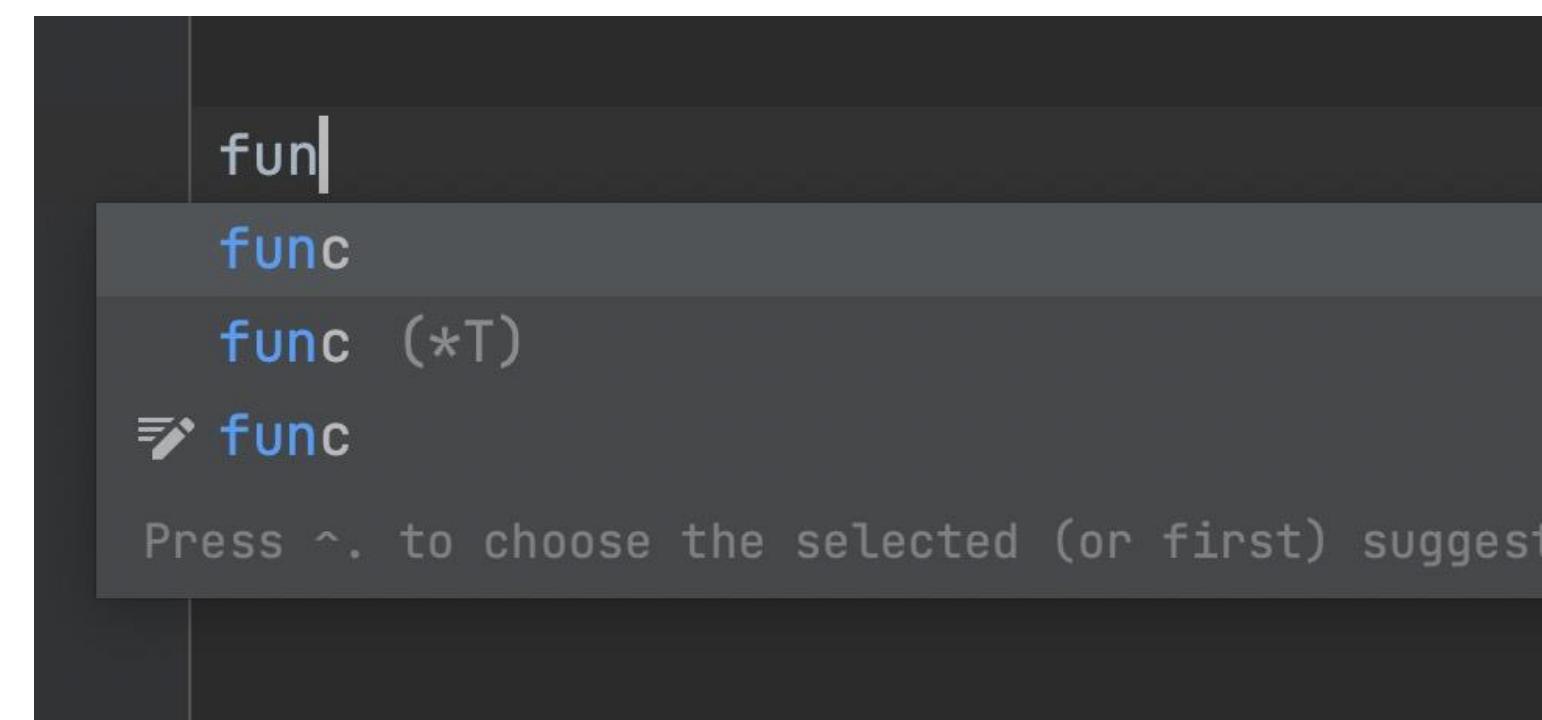
A screenshot of a code editor showing a Go program. The code defines two constants: `internal` (with a capital 'I') and `External` (with a capital 'E'). It then defines a `main` function that prints the value of `a` using `println`. The code editor highlights the constants and the function name in orange, while the variable `a` is highlighted in purple. A yellow lightbulb icon is positioned next to the closing brace of the `main` function, likely indicating a suggestion or a warning.

# 目录

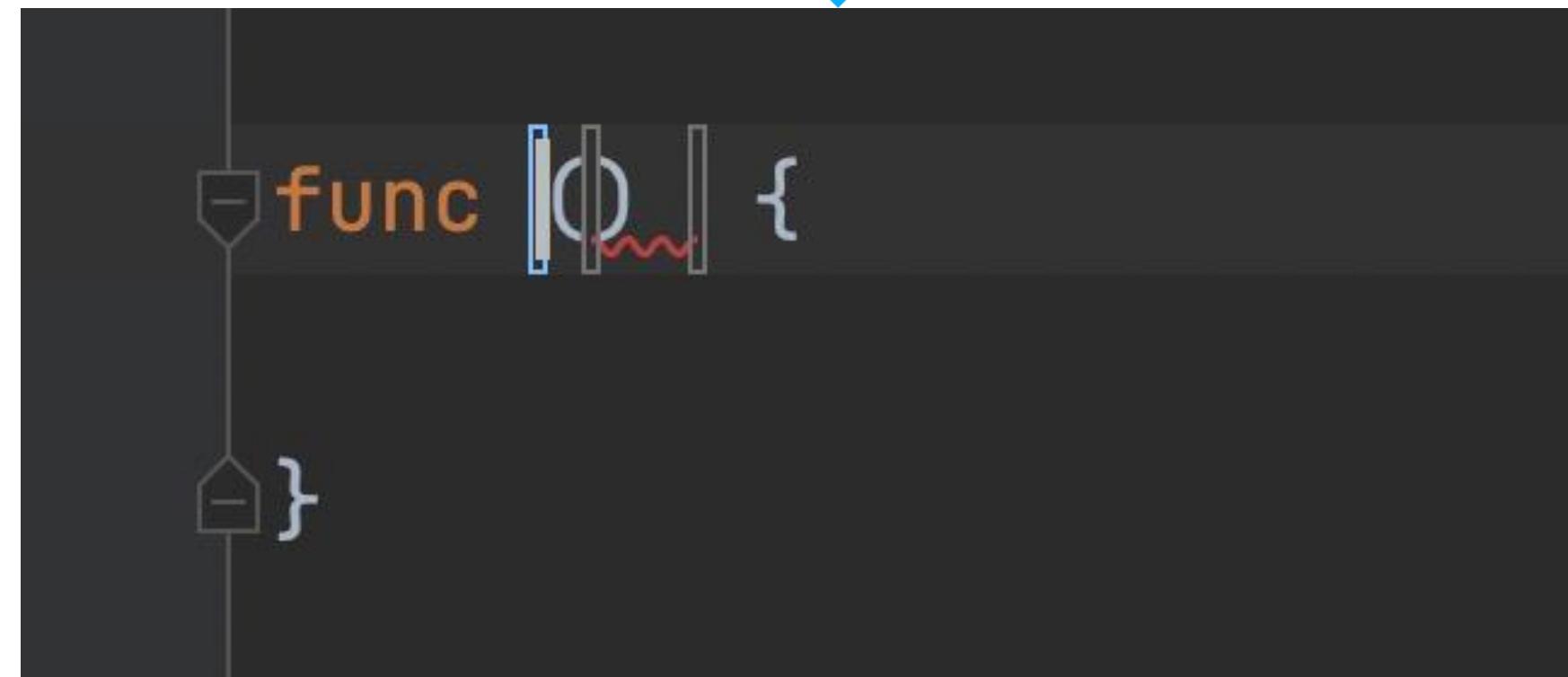
1. main 函数概览
2. package 声明
3. string 和 基础类型
4. 变量声明
5. 方法声明与调用

# Golang 语法 — 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>]



输入 fu, 按下 enter



Tip: 使用Goland来创建文件夹，它会自动加上package

# Golang 语法 — 方法声明（推荐写法）

- 参数列表含有参数名
- 返回值不具有返回值名

```
// Fun0 只有一个返回值，不需要括号括起来
func Fun0(name string) string {
    return "Hello, " + name
}

// Fun1 多个参数，多个返回值。参数有名字，但是返回值没有
func Fun1(a string, b int) (int, string) {
    return 0, "你好"
}
```

Tip: 我个人偏好的写法，比较接近其它语言的习惯

# Golang 语法 — 方法声明 (看看就好)

```
// Fun2 的返回值具有名字，可以在内部直接复制，然后返回
// 也可以忽略age, name, 直接返回别的。
func Fun2(a string, b string) (age int, name string) {
    age = 19
    name = "Tom"
    return
    //return 19, "Tom" // 这样返回也可以
}

// Fun3 多个参数具有相同类型放在一起，可以只写一次类型
func Fun3(a, b, c string, abc, bcd int, p string) (d, e int, g string) {
    d = 15
    e = 16
    g = "你好"
    return
    //return 0, 0, "你好" // 这样也可以
}
```

Tip: 具体怎么写，看公司规范和个人偏好，实际上没啥最佳实践

# Golang 语法 — 方法调用

- 使用 \_ 忽略返回值

The screenshot shows a dark-themed Go code editor interface. On the left, there's a vertical navigation bar with icons for file operations. The main area contains the following Go code:

```
▶ func main() {
    a := Fun0(name: "Tom")
    println(a)

    b, c := Fun1(a: "a", b: 17)
    println(b)
    println(c)

    _, d := Fun2(a: "a", b: "b")
    println(d)
}
```

The code demonstrates method calls and variable declarations. The variable 'a' is assigned the result of the 'Fun0' function call, which takes a parameter 'name' with the value 'Tom'. The variable 'b' and 'c' are assigned the results of the 'Fun1' function call, which takes parameters 'a' ('a') and 'b' (17). The variables 'd' and '\_' are assigned the results of the 'Fun2' function call, which takes parameters 'a' ('a') and 'b' ('b'). The 'println' statements output the values of these variables. A yellow lightbulb icon is positioned next to the line containing the assignment of 'd'.

# Golang 语法 — 方法声明与调用总结

- golang 支持多返回值，这是一个很大的不同点
- golang 方法的作用域和变量作用域一样，通过大小写控制
- golang 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

Q & A

# 目录

1. 环境安装
2. 创建项目
3. Hello Go!
4. 基础语法
5. 最简单的 Web 服务器

# 最简单的 web 服务器——官网例子

## Introducing the `net/http` package (an interlude)

Here's a full working example of a simple web server:

```
// +build ignore

package main

import (
    "fmt"
    "log"
    "net/http"
)

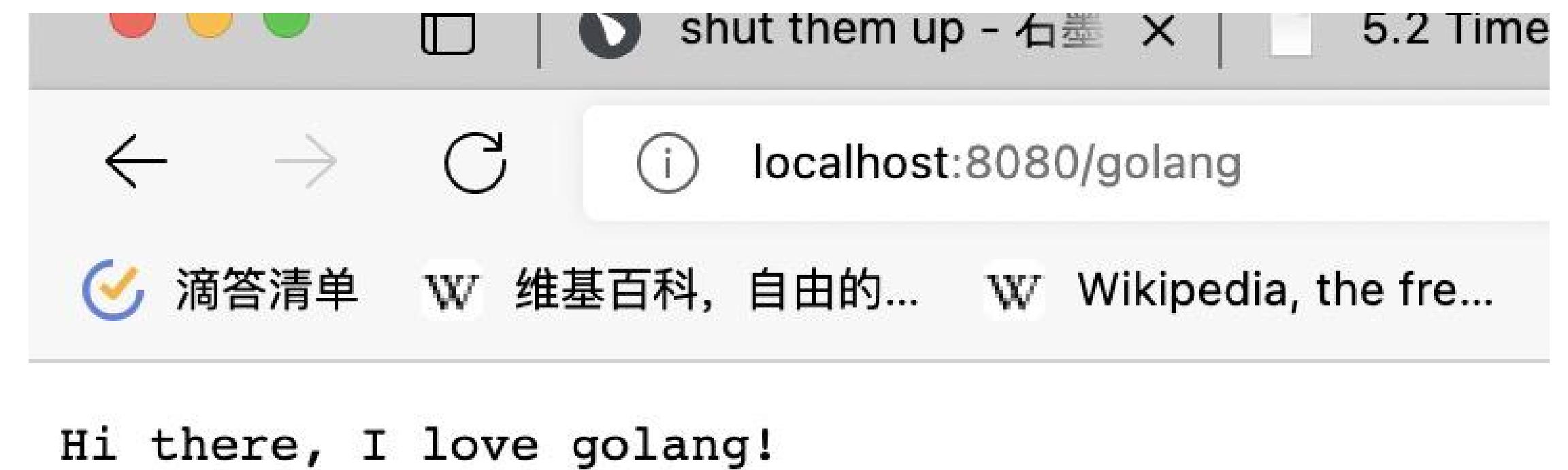
func handler(w http.ResponseWriter, r *http.Request) {
    fmt.Fprintf(w, "Hi there, I love %s!", r.URL.Path[1:])
}

func main() {
    http.HandleFunc("/", handler)
    log.Fatal(http.ListenAndServe(":8080", nil))
}
```

Tip: 要熟练掌握找官网，找靠谱例子的能力

# 最简单的 web 服务器

- 直接 Goland 启动 main 函数
- 浏览器输入 `http://localhost:8080/golang`



# 最简单的 web 服务器 —— 增加几个路由

```
func home(w http.ResponseWriter, r *http.Request) {
    fmt.Fprintf(w, "这是主页")
}

func user(w http.ResponseWriter, r *http.Request) {
    fmt.Fprintf(w, "这是用户")
}

func createUser(w http.ResponseWriter, r *http.Request) {
    fmt.Fprintf(w, "这是创建用户")
}

func order(w http.ResponseWriter, r *http.Request) {
    fmt.Fprintf(w, "这是订单")
}
```

```
func main() {
    http.HandleFunc("/", home)
    http.HandleFunc("/user", user)
    http.HandleFunc("/user/create", createUser)
    http.HandleFunc("/order", order)
    log.Fatal(http.ListenAndServe(":8080", nil))
}
```

“哎呀，我想知道怎么操作 http 请求，怎么返回复杂响应~~”

# fmt 格式化输出

```
▶  func main() {
    name := "Tom"
    age := 17
    // 这个 API 是返回字符串的，所以大多数时候我们都是用这个
    str := fmt.Sprintf(format: "hello, I am %s, I am %d years old \n", name, age)
    println(str)

    // 这个是直接输出，一般简单程序 DEBUG 会用它输出到一些信息到控制台
    fmt.Printf(format: "hello, I am %s, I am %d years old \n", name, age)
}
```

# fmt 格式化输出

fmt 包有完整的说明

- 掌握常用的: %s, %d, %v, %+v, %#v
- 不仅仅是`fmt`的调用, 所有格式化字符串的 API 都可以用
- 因为golang字符串拼接只能在 string 之间, 所以这个包非常常用

Tip: 如果不知道使用哪个占位符, 就一个个试过去

```
General:  
    %v  the value in a default format  
        when printing structs, the plus flag (%+v)  
    %#v  a Go-syntax representation of the value  
    %T  a Go-syntax representation of the type of  
    %%  a literal percent sign; consumes no value  
  
Boolean:  
    %t  the word true or false  
  
Integer:  
    %b  base 2  
    %c  the character represented by the correspondi...  
    %d  base 10  
    %o  base 8  
    %0  base 8 with 0o prefix  
    %q  a single-quoted character literal safely embedd...  
    %x  base 16, with lower-case letters for a-f  
    %X  base 16, with upper-case letters for A-F  
    %U  Unicode format: U+1234; same as "U+%04X"  
  
Floating-point and complex constituents:  
    %b  decimalless scientific notation with exponent  
        in the manner of strconv.FormatFloat with
```

# fmt 格式化输出

```
func replaceHolder() {
    u := &user{
        Name: "Tom",
        Age: 17,
    }
    fmt.Printf("v => %v \n", u)
    fmt.Printf("+v => %+v \n", u)
    fmt.Printf("#v => %#v \n", u)
    fmt.Printf("T => %T \n", u)
}

type user struct {
    Name string
    Age int
}
```

# 目录

1. 数组和切片
2. for
3. if - else
4. switch

# 基础语法 —— 数组和切片

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. 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

```
func main() {  
    // 直接初始化一个三个元素的数组。大括号里面多一个或者少一个都编译不通过  
    a1 := [3]int{9, 8, 7}  
    fmt.Printf("a1: %v, len: %d, cap: %d", a1, len(a1), cap(a1))  
  
    // 初始化一个三个元素的数组，所有元素都是0  
    var a2 [3]int  
    fmt.Printf("a2: %v, len: %d, cap: %d", a2, len(a2), cap(a2))  
  
    // a1 = append(a1, 12) 数组不支持 append 操作  
  
    // 按下标索引  
    fmt.Printf("a1[1]: %d", a1[1])  
    // 超出下标范围，直接崩溃，编译不通过  
    //fmt.Printf("a1[99]: %d", a1[99])  
}
```

Tip: 数组的len 和 cap 结果是一样的，就是数组的长度

# 基础语法 —— 数组和切片

切片语法: `[]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)`

```
▶ func main() {
    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) // 创建了一个包含三个元素, 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) // 后边添加一个元素, 没有超出容量限制, 不会发生扩容
    fmt.Printf("s2: %v, len %d, cap: %d \n", s2, len(s2), cap(s2))

    s2 = append(s2, elems...: 8) // 后边添加了一个元素, 触发扩容
    fmt.Printf("s2: %v, len %d, cap: %d \n", s2, len(s2), cap(s2))

    s3 := make([]int, 4) // 只传入一个参数, 表示创建一个含有四个元素的
    fmt.Printf("s3: %v, len %d, cap: %d \n", s3, len(s3), cap(s3))

    // 按下标索引
    fmt.Printf("s3[2]: %d", s3[2])
    // 超出下标范围, 直接崩溃
    // runtime error: index out of range [99] with length 4
    fmt.Printf("s3[99]: %d", s3[99])
```

Tip: 初学的时候不必关心什么时候扩容, 什么时候不扩容

# 基础语法 —— 数组和切片

	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

Tip: 遇事不决用切片，基本不会出错

# 基础语法 —— 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

```
func subSlice() {  
    s1 := []int{2, 4, 6, 8, 10}  
    s2 := s1[1:3]  
    fmt.Printf("s2: %v, len %d, cap: %d\n", s2, len(s2), cap(s2))  
  
    s3 := s1[2:]  
    fmt.Printf("s3: %v, len %d, cap: %d\n", s3, len(s3), cap(s3))  
  
    s4 := s1[:3]  
    fmt.Printf("s4: %v, len %d, cap: %d\n", s4, len(s4), cap(s4))  
}
```

Tip: 左闭右开原则

# 基础语法 —— 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

**Tip:** 遇事不决用切片，不容易错

# 基础语法 —— 共享底层 (optional)

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

有余力的同学可以运行一下 **ShareSlice()**

# 基础语法 —— 共享底层 (optional)

```
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))  
}
```

# 目录

1. 数组和切片
2. for
3. if - else
4. switch

# 基础语法 —— for

for 和别的语言差不多，有三种形式：

1. for {}, 类似 while 的无限循环
2. for i, 一般的按照下标循环
3. for range 最为特殊的 range 遍历
4. break 和 continue 和别的语言一样

```
func ForLoop() {  
    arr := []int {9, 8, 7, 6}  
    index := 0  
    for {  
        if index == 3{  
            // break 跳出循环  
            break  
        }  
        fmt.Printf( format: "%d => %d", index, arr[index])  
        index ++  
    }  
    fmt.Println( a...: "for loop end ")  
}
```

# 基础语法 —— for

```
func ForI() {  
    arr := []int {9, 8, 7, 6}  
    for ~  
    for  
        fori  
        forr  
    f  ForLoop()  
    f  ForR()
```

```
func ForI() {  
    arr := []int {9, 8, 7, 6}  
    for i := 0; i <~; i++ {  
    }  
    }  
}
```

```
func ForI() {  
    arr := []int {9, 8, 7, 6}  
    for i := 0; i < len(arr); i++ {  
        fmt.Printf( format: "%d => %d", i, arr[i])  
    }  
    fmt.Println( a...: "for loop end ")  
}
```

# 基础语法 —— for

```
func ForR() {  
    arr := []int {9, 8, 7, 6}  
    for |  
        for i|  
        for r|  
    } ForLoop()
```

```
func ForR() {  
    arr := []int {9, 8, 7, 6}  
    for i, i2 := range collection {  
    }  
}
```

```
func ForR() {  
    arr := []int {9, 8, 7, 6}  
    // 如果只是需要 value, 可以用 _ 代替 index  
    // 如果只需要 index 也可以去掉 写成 for index := range arr  
    for index, value := range arr {  
        fmt.Printf(format: "%d => %d", index, value)  
    }  
    fmt.Println(a...: "for r loop end")  
}
```

# 目录

1. 数组和切片
2. for
3. if - else
4. switch

# 基础语法 —— if - else

if-else 和别的语言也差不多

```
func Young(age int) {  
    if age < 18{  
        fmt.Println( a...: "I am a child!")  
    } else {  
        // else 分支也可以没有  
        fmt.Println( a...: "I not a child")  
    }  
}
```

# 基础语法 —— if - else

带局部变量声明的 if- else:

1. distance 只能在 if 块， 或者后边所有的 else 块里面使用
2. 脱离了 if - else 块，则不能再使用

```
func IfUsingNewVariable(start int, end int) {  
    if distance := end - start; distance > 100 {  
        fmt.Printf( format: "距离太远, 不来了: %d\n", distance)  
    } else {  
        // else 分支也可以没有  
        fmt.Printf( format: "距离并不远, 来一趟: %d\n", distance)  
    }  
  
    // 这里不能访问 distance  
    //fmt.Printf("距离是: %d\n", distance)  
}
```

# 目录

1. 数组和切片
2. for
3. if - else
4. switch

# 基础语法 —— switch

switch 和别的语言差不多

switch 后面可以是基础类型和字符串，或者满足特定条件的结构体

最大的差别：

终于不用加 break 了！

```
func ChooseFruit(fruit string) {  
    switch fruit {  
        case "苹果":  
            fmt.Println("这是一个苹果")  
        case "草莓", "蓝莓":  
            fmt.Println("这是霉霉")  
        default:  
            fmt.Printf("不知道是啥: %s \n", fruit)  
    }  
}
```

Tip: 大多数时候，switch 后面只会用基础类型或者字符串

# 要点总结

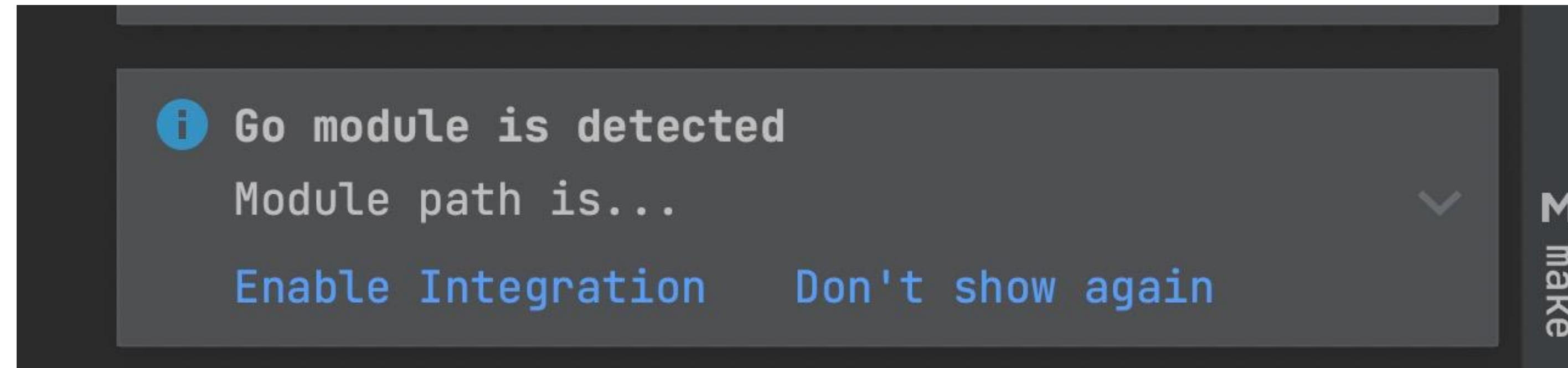
1. string 类型 —— 和别的语言没啥区别
2. 基础类型 —— 不必死记硬背， Goland IDE 会提示你
3. 切片 —— make, [i], len, cap, append
4. 数组 —— 和别的语言没啥区别
5. for, if, switch —— 和别的语言区别不大， IDE 会提示你

# 课后练习

- 计算斐波那契数列
- 实现切片的 Add 和 Delete 方法
- 去 leetcode 上试试（先看答案，再尝试用 go 写出来）：
  - <https://leetcode-cn.com/problems/two-sum/>
  - <https://leetcode-cn.com/problems/search-insert-position/>
- 我们课上用了很多 fmt 来格式化字符串，那么如何输出：
- 3.1 保留两位小数的数字
- 3.2 将 []byte 输出为 16 进制
- 预习 type 的用法

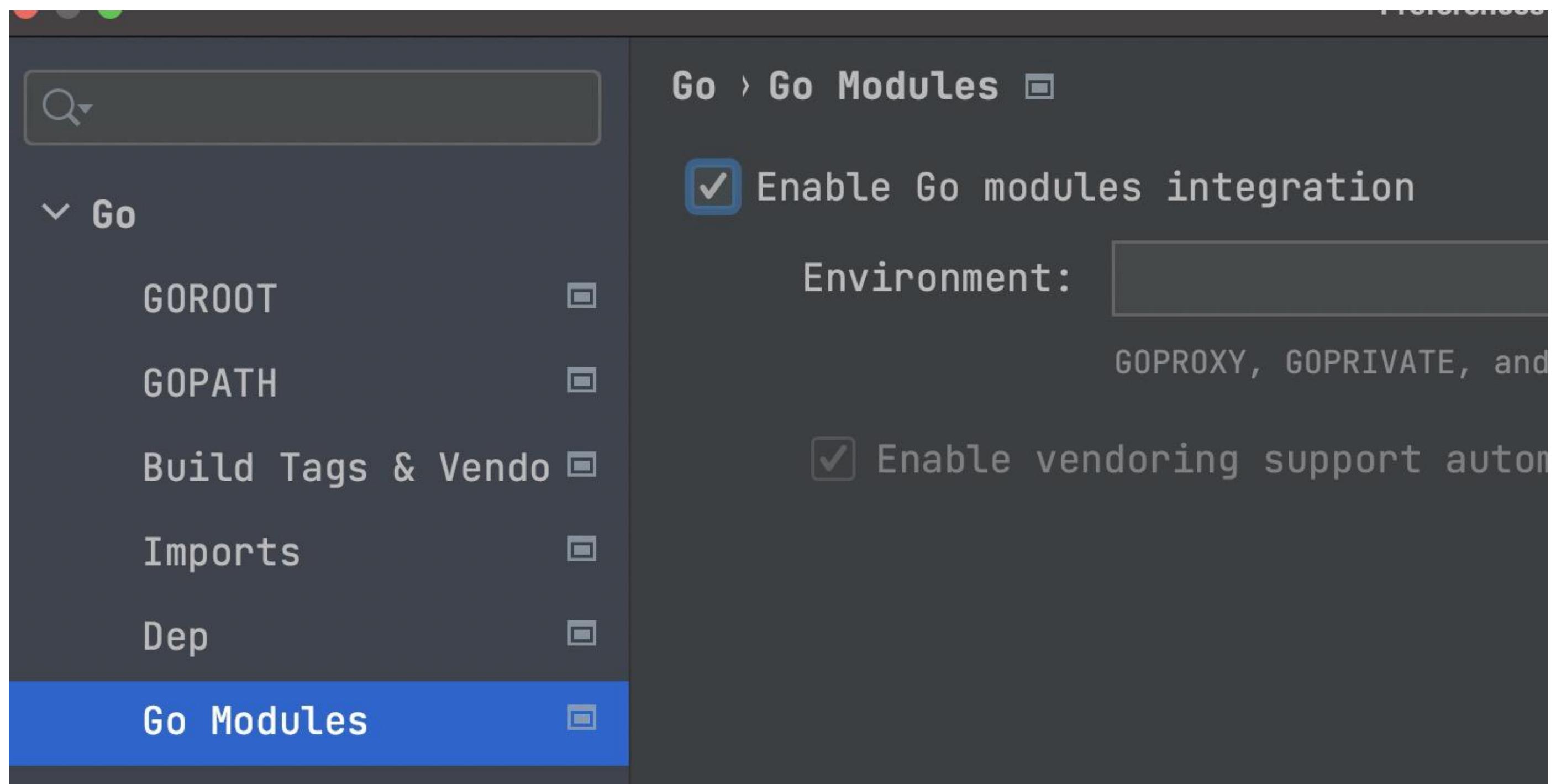
# 最简单的 web 服务器 —— go mod 管理依赖

- 配置 Goland



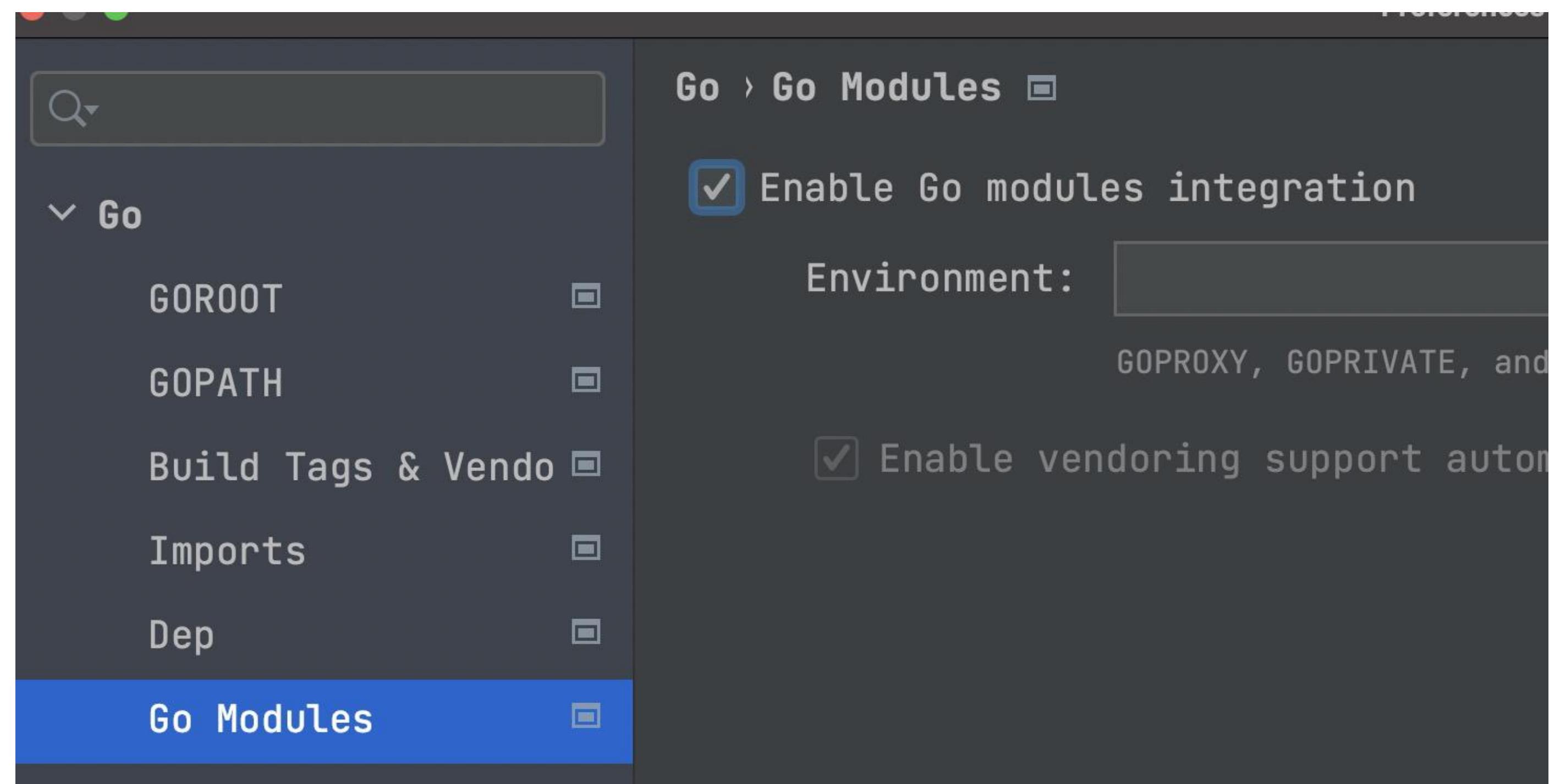
# 最简单的 web 服务器 —— go mod 管理依赖

- 直接开发设置配置也可以



# 最简单的 web 服务器 —— go mod 管理依赖

- 直接开发设置配置也可以



# THANKS

