Go by Example 学习笔记

1. Hello World (你好世界)

https://gobyexample.com/hello-world

1.1 核心概念

"Hello World" 是学习任何编程语言的第一步，它展示了一个最简单的可运行程序的基本结构。在 Go 语言中，一个 Hello World 程序包含了 Go 程序的几个核心要素：包声明、导入语句、主函数以及基本的输出操作。

这个简单的程序就像是打开一扇门，让我们初步了解 Go 语言的语法风格和程序组织方式。

1.2 程序结构解析

1.2.1 包声明 (Package Declaration)

```
package main
```

说明：

- 每个 Go 程序都必须以包声明开始
- `package main` 表示这是一个可执行程序的入口包

- 只有 `main` 包才能被编译成可执行文件，其他包名的文件会被编译成库文件

类比：包声明就像是给程序贴上一个身份标签，告诉编译器"我是一个独立运行的程序"还是"我是一个工具库"。

1.2.2 导入语句 (Import Statement)

```
import "fmt"
```

说明：

- `import` 关键字用于导入其他包
- `fmt` 是 Go 标准库中的格式化输入输出包（format 的缩写）
- 导入的包必须被使用，否则编译器会报错

小贴士：Go 编译器对未使用的导入非常严格，这有助于保持代码整洁，避免不必要的依赖。

1.2.3 主函数 (Main Function)

```
func main() {
    fmt.Println("hello world")
}
```

说明：

- `func` 是声明函数的关键字
- `main` 函数是程序的入口点，程序从这里开始执行
- `fmt.Println()` 用于在控制台打印一行文本，并自动添加换行符
- Go 使用大括号 `{}` 来定义代码块

重要概念：`main` 包中的 `main` 函数是 Go 可执行程序的必需组成部分，缺少任何一个都无法编译成可执行文件。

1.3 运行 Go 程序的两种方式

1.3.1 直接运行 (go run)

使用 `go run` 命令可以直接编译并运行 Go 源代码，无需生成中间文件。

```
go run hello-world.go
```

特点：适合开发和测试阶段，快速验证代码逻辑。

1.3.2 编译后运行 (go build)

使用 `go build` 命令将源代码编译成可执行文件，然后运行该文件。

```
go build hello-world.go  # 生成可执行文件
./hello-world            # 运行可执行文件
```

特点：生成的可执行文件可以独立分发和运行，无需 Go 环境。

1.4 完整代码示例

```
package main

import "fmt"

func main() {
    fmt.Println("hello world")
}
```

1.5 运行结果

```
$ go run hello-world.go
hello world

$ go build hello-world.go
$ ls
hello-world     hello-world.go

$ ./hello-world
hello world
```

1.6 关键要点

1. **三要素结构**：Go 可执行程序必须包含 `package main` 声明、必要的导入语句和 `main` 函数。
2. **严格的导入检查**：导入的包必须被使用，否则编译失败，这促使开发者保持代码整洁。
3. **两种运行方式**：`go run` 适合快速测试，`go build` 适合生成可分发的可执行文件。
4. **自动换行**：`fmt.Println()` 会自动在输出末尾添加换行符，如果不需要换行可以使用 `fmt.Print()`。
5. **入口唯一性**：一个 Go 程序只能有一个 `main` 包和一个 `main` 函数作为入口点。
6. **编译型语言**：Go 是编译型语言，代码需要先编译成机器码才能执行，这带来了更好的性能表现。

---

2. Values (值)

https://gobyexample.com/values

2.1 核心概念

Go 支持多种基本数据类型的值，包括字符串、整数、浮点数和布尔值等。这些值类型是构建 Go 程序的基础，就像建筑的砖块一样，通过不同的组合和运算可以构建出复杂的程序逻辑。

Go 是一门静态类型语言，每个值都有明确的类型，编译器会在编译时进行类型检查，这有助于在程序运行前发现潜在的类型错误。

2.2 基本值类型与运算

2.2.1 字符串 (String)

字符串是字符序列，在 Go 中使用双引号表示。字符串可以通过 `+` 运算符进行拼接。

```
fmt.Println("go" + "lang")  // 输出: golang
```

注：Go 中的字符串是不可变的，每次拼接都会创建新的字符串。

2.2.2 整数 (Integer)

整数支持常见的算术运算，如加法、减法、乘法和除法。

```
fmt.Println("1+1 =", 1+1)      // 输出: 1+1 = 2
fmt.Println("7-3 =", 7-3)      // 输出: 7-3 = 4
fmt.Println("2*3 =", 2*3)      // 输出: 2*3 = 6
fmt.Println("10/3 =", 10/3)    // 输出: 10/3 = 3 (整数除法，结果向下取整)
```

小贴士：整数之间的除法会舍弃小数部分，如果需要保留小数，至少有一个操作数应该是浮点数。

2.2.3 浮点数 (Float)

浮点数用于表示带小数的数值，支持精确的小数运算。

```
fmt.Println("7.0/3.0 =", 7.0/3.0)  // 输出: 7.0/3.0 = 2.3333333333333335
```

2.2.4 布尔值 (Boolean)

布尔值只有两个可能的值：`true` 和 `false`。布尔值支持逻辑运算，包括与 (`&&`)、或 (`||`) 和非 (`!`)。

```
fmt.Println(true && false)   // 输出: false (逻辑与: 两者都为真时才为真)
fmt.Println(true || false)   // 输出: true  (逻辑或: 至少一个为真时就为真)
fmt.Println(!true)           // 输出: false (逻辑非: 取反)
```

类比：布尔逻辑就像电路开关，`&&` 相当于串联（都通才通），`||` 相当于并联（一个通就通），`!` 相当于反向开关。

2.3 类型系统的重要性

Go 的类型系统确保了不同类型的值不能随意混用。例如，不能直接将字符串和整数相加，这种严格性虽然在编写时需要更多注意，但能在编译阶段就发现许多潜在错误，避免运行时的意外行为。

2.4 完整代码示例

```
package main

import "fmt"

func main() {
    // 字符串拼接
```

```
    fmt.Println("go" + "lang")

    // 整数运算
    fmt.Println("1+1 =", 1+1)
    fmt.Println("7-3 =", 7-3)

    // 浮点数运算
    fmt.Println("7.0/3.0 =", 7.0/3.0)

    // 布尔运算
    fmt.Println(true && false)  // 逻辑与
    fmt.Println(true || false)  // 逻辑或
    fmt.Println(!true)          // 逻辑非
}
```

2.5 运行结果

```
$ go run values.go
golang
1+1 = 2
7-3 = 4
7.0/3.0 = 2.3333333333333335
false
true
false
```

2.6 关键要点

1. **多种基本类型**：Go 支持字符串、整数、浮点数、布尔值等基本值类型，每种类型都有其特定的运算规则。
2. **字符串拼接**：使用 `+` 运算符可以连接字符串，但要注意字符串是不可变的。
3. **整数除法**：整数之间的除法会舍弃小数部分，结果仍为整数。
4. **布尔逻辑**：`&&`（与）、`||`（或）、`!`（非）是三种基本的布尔运算符，用于构建条件判断逻辑。
5. **静态类型**：Go 的类型系统在编译时进行检查，确保类型安全，减少运行时错误。
6. **类型明确**：每个值都有明确的类型，不同类型之间不能隐式转换，需要显式转换才能混合使用。

---

3. Variables (变量)

https://gobyexample.com/variables

3.1 核心概念

在Go中，变量是显式声明的，编译器会使用这些声明来检查函数调用的类型正确性等。

生活化类比：变量就像一个带标签的储物柜。`var a = "initial"` 就好比你拿了一个名为 `a` 的柜子，在里面存放了 "initial" 这个物品。你可以随时查看柜子里的东西，也可以把里面的东西换成别的。

3.2 变量声明的几种方式

3.2.1 var 关键字声明并初始化

```
var a = "initial"  // Go会自动推断类型为string
```

3.2.2 同时声明多个变量

```
var b, c int = 1, 2  // 显式指定类型为int
```

3.2.3 类型推断

```
var d = true  // Go推断类型为bool
```

3.2.4 零值初始化

```
var e int  // 未初始化，自动赋予零值0
```

重要概念：Go中每种类型都有零值：

- `int` 的零值是 `0`
- `string` 的零值是 `""`
- `bool` 的零值是 `false`
- 指针、切片、映射、通道、函数和接口的零值是 `nil`

3.2.5 短变量声明（:= 语法）

```
f := "apple"  // 等价于 var f string = "apple"
```

注意： `:=` 语法只能在函数内部使用，是声明和初始化变量的简写形式。

3.3 完整代码示例

```
package main

import "fmt"

func main() {
  var a = "initial"  // 类型推断为string
  fmt.Println(a)

  var b, c int = 1, 2  // 显式声明为int类型
  fmt.Println(b, c)

  var d = true  // 类型推断为bool
  fmt.Println(d)
```

```
  var e int  // 零值初始化为0
  fmt.Println(e)

  f := "apple"  // 短变量声明，等价于var f string = "apple"
  fmt.Println(f)
}
```

3.4 运行结果

```
mac@macdeMacBook-Pro Go by Example % go run variables.go
initial     # 字符串变量a的值
1 2         # 整数变量b和c的值
true        # 布尔变量d的值
0           # 未初始化的int变量e的零值
apple       # 短变量声明的f的值
mac@macdeMacBook-Pro Go by Example % go build variables.go
mac@macdeMacBook-Pro Go by Example % ./variables
initial
1 2
true
0
apple
```

3.5 关键要点

1. **显式声明**：Go要求变量必须显式声明，有助于编译时类型检查
2. **类型推断**：Go可以根据初始值自动推断变量类型
3. **零值安全**：所有变量都有合理的零值，避免了未初始化变量的问题
4. **作用域限制**：`:=` 只能在函数内使用，函数外必须使用 `var`
5. **多变量声明**：可以一次声明多个相同类型的变量

---

4. Constants (常量)

https://gobyexample.com/constants

4.1 核心概念

Go支持字符、字符串、布尔值和数值类型的常量。常量是在编译期确定的值，运行时不可修改。

生活化类比：常量就像圆周率 π (约等于 3.14159)，它是一个在数学中固定不变的值。在程序中，无论你什么时候引用这个常量，它的值都是一样的，并且你不能试图去改变它，就像你不能重新定义 π 的值一样。

4.2 常量声明的特点

4.2.1 const关键字声明

```
const s string = "constant"  // 显式指定类型的常量
```

4.2.2 常量可以出现在任何var语句可以出现的地方

```
const n = 500000000  // 数值常量，没有指定类型
```

4.2.3 常量表达式具有任意精度运算

```
const d = 3e20 / n  // 编译时计算，支持高精度运算
```

重要概念：常量表达式在编译时进行计算，支持任意精度的算术运算。

4.2.4 数值常量的类型推断

```
fmt.Println(int64(d))     // 显式类型转换
fmt.Println(math.Sin(n))  // 根据函数要求隐式转换为float64
```

关键特性：数值常量在使用前是无类型的，只有在需要特定类型的上下文中才会被赋予类型。

4.3 完整代码示例

```
package main

import (
  "fmt"
  "math"
)

const s string = "constant"  // 字符串常量

func main() {
  fmt.Println(s)

  const n = 500000000  // 数值常量（无类型）

  // 常量表达式支持任意精度运算
  const d = 3e20 / n
  fmt.Println(d)

  // 显式类型转换
  fmt.Println(int64(d))

  // 根据函数参数要求进行隐式类型转换
  fmt.Println(math.Sin(n))
}
```

4.4 运行结果

```
$ go run constants.go
constant            # 字符串常量s的值
6e+11              # 常量表达式d的科学计数法表示
600000000000       # d转换为int64后的值
-0.28470407323754404  # math.Sin(n)的结果，n被隐式转换为float64
```

4.5 关键要点

1. **编译时确定**：常量值在编译时就已确定，运行时不可修改
2. **支持多种类型**：字符、字符串、布尔值和数值类型都可以声明为常量
3. **任意精度运算**：常量表达式支持任意精度的算术运算
4. **无类型特性**：数值常量在使用前是无类型的，根据使用上下文确定类型
5. **灵活的类型转换**：可以显式转换，也可以根据函数参数要求隐式转换
6. **作用域灵活**：`const` 可以在包级别或函数级别声明

---

5. For (循环)

https://gobyexample.com/for

5.1 核心概念

`for`是Go语言唯一的循环结构。Go没有`while`、`do-while`或其他循环语句，所有循环都使用`for`关键字来实现。

生活化类比：`for`循环就像是你在操场上跑步。教练可能会给你下三种指令：

1. **跑10圈** (经典`for`循环: `for i := 0; i < 10; i++`)。
2. **跑到你累了为止** (条件`for`循环: `for !tired`)。
3. **一直跑，直到我叫停** (无限`for`循环: `for { ... if stop { break } }`)。
   无论哪种，核心都是“重复跑圈”这个动作。

5.2 for循环的五种基本形式

5.2.1 类似while循环的条件循环

```
i := 1
for i <= 3 {
    fmt.Println(i)
    i = i + 1
}
```

特点：只有一个条件表达式，类似其他语言的`while`循环。

5.2.2 经典的三部分for循环

```go
for j := 0; j < 3; j++ {
    fmt.Println(j)
}
```

结构：`for 初始化; 条件; 后置语句 { 循环体 }`，这是最常见的循环形式。

5.2.3 range循环遍历整数

```go
for i := range 3 {
    fmt.Println("range", i)
}
```

说明：Go 1.22+新特性，`range 3`等价于遍历0到2的整数序列。

5.2.4 无限循环

```go
for {
    fmt.Println("loop")
    break  // 使用break跳出循环
}
```

特点：没有条件的`for`循环会无限执行，直到遇到`break`或`return`。

5.2.5 使用continue控制循环

```go
for n := range 6 {
    if n%2 == 0 {
        continue  // 跳过当前迭代，继续下一次
    }
    fmt.Println(n)
}
```

控制流：`continue`跳过当前迭代的剩余部分，直接进入下一次循环。

5.3 完整代码示例

```go
package main

import "fmt"

func main() {
    // 1. 类似while的条件循环
    i := 1
    for i <= 3 {
        fmt.Println(i)
        i = i + 1
    }

    // 2. 经典的三部分for循环
```

```
    for j := 0; j < 3; j++ {
        fmt.Println(j)
    }

    // 3. range循环遍历整数（Go 1.22+）
    for i := range 3 {
        fmt.Println("range", i)
    }

    // 4. 无限循环
    for {
        fmt.Println("loop")
        break
    }

    // 5. 使用continue控制循环
    for n := range 6 {
        if n%2 == 0 {
            continue
        }
        fmt.Println(n)
    }
}
```

5.4 运行结果

```
$ go run for.go
1           # 条件循环输出
2
3
0           # 经典for循环输出
1
2
range 0     # range循环输出
range 1
range 2
loop        # 无限循环输出（break后退出）
1           # continue控制的循环，只输出奇数
3
5
```

5.5 关键要点

1. **唯一循环结构**：Go只有`for`一种循环，但形式多样，能满足所有循环需求
2. **灵活的语法**：支持类似while、经典三部分、range等多种写法
3. **控制语句**：`break`用于跳出循环，`continue`用于跳过当前迭代
4. **无限循环安全**：可以使用`for {}`创建无限循环，通过`break`或`return`退出
5. **range新特性**：Go 1.22+支持`range 整数`的语法，简化了计数循环

6. **变量作用域**：循环初始化的变量作用域仅限于循环内部

---

6. If/Else (条件判断)

https://gobyexample.com/if-else

6.1 核心概念

Go中的条件分支使用`if`和`else`语句，语法直观简洁。Go的条件语句有一些独特的特性和设计理念。

生活化类比：`if-else`就像是走到了一个岔路口。你抬头看路牌（`if`后面的条件），如果路牌指示"去公园向左"（条件为真），你就向左走（执行`if`代码块）。否则（`else`），你就走另一条路（执行`else`代码块）。它帮你根据情况做出二选一的决定。

6.2 if-else语句的四种基本形式

6.2.1 基本的if-else语句

```
if 7%2 == 0 {
    fmt.Println("7 is even")
} else {
    fmt.Println("7 is odd")
}
```

特点：条件不需要小括号`()`，但大括号`{}`是必须的。

6.2.2 单独的if语句（没有else）

```
if 8%4 == 0 {
    fmt.Println("8 is divisible by 4")
}
```

用法：当只需要在条件为真时执行操作，可以省略`else`部分。

6.2.3 使用逻辑运算符的条件

```
if 8%2 == 0 || 7%2 == 0 {
    fmt.Println("either 8 or 7 are even")
}
```

逻辑运算符：

- `&&`：逻辑与（AND）
- `||`：逻辑或（OR）
- `!`：逻辑非（NOT）

6.2.4 带初始化语句的if（重要特性）

```
if num := 9; num < 0 {
    fmt.Println(num, "is negative")
} else if num < 10 {
    fmt.Println(num, "has 1 digit")
} else {
    fmt.Println(num, "has multiple digits")
}
```

关键特性：

- 可以在条件前执行一个语句（通常用于声明变量）
- 在此语句中声明的变量在当前和所有后续分支中都可用
- 变量作用域限制在整个if-else语句块内

6.3 完整代码示例

```
package main

import "fmt"

func main() {
    // 1. 基本的if-else语句
    if 7%2 == 0 {
        fmt.Println("7 is even")
    } else {
        fmt.Println("7 is odd")
    }

    // 2. 单独的if语句
    if 8%4 == 0 {
        fmt.Println("8 is divisible by 4")
    }

    // 3. 使用逻辑运算符的条件
    if 8%2 == 0 || 7%2 == 0 {
        fmt.Println("either 8 or 7 are even")
    }

    // 4. 带初始化语句的if（重要特性）
    if num := 9; num < 0 {
        fmt.Println(num, "is negative")
    } else if num < 10 {
        fmt.Println(num, "has 1 digit")
    } else {
        fmt.Println(num, "has multiple digits")
    }
}
```

6.4 运行结果

```
$ go run if-else.go
7 is odd                    # 7%2 != 0，执行else分支
8 is divisible by 4         # 8%4 == 0，条件为真
either 8 or 7 are even      # 8%2 == 0为真，||运算结果为真
9 has 1 digit               # num=9，第二个条件num < 10为真
```

6.5 Go条件语句的设计特点

6.5.1 语法规则

- **不需要小括号**：条件表达式不需要用`()`包围
- **必须有大括号**：代码块必须用`{}`包围，即使只有一行代码

6.5.2 没有三元运算符

```
// Go中没有类似 a ? b : c 的三元运算符
// 必须使用完整的if语句
var result string
if condition {
    result = "true case"
} else {
    result = "false case"
}
```

6.5.3 变量作用域管理

```
if x := getValue(); x > 0 {
    // x 在这里可用
    fmt.Println("positive:", x)
} else {
    // x 在这里也可用
    fmt.Println("non-positive:", x)
}
// x 在这里不可用，作用域已结束
```

6.6 关键要点

1. **简洁语法**：条件不需要小括号，但大括号是必须的
2. **灵活性**：支持单独的if、if-else、if-else if-else等多种形式
3. **初始化语句**：可以在条件前声明变量，作用域限制在if块内
4. **逻辑运算符**：支持`&&`、`||`、`!`等逻辑运算符组合条件
5. **没有三元运算符**：需要使用完整的if语句处理条件赋值
6. **作用域安全**：初始化语句中的变量作用域被严格限制，避免变量泄漏

---

7. Switch (分支语句)

https://gobyexample.com/switch

7.1 核心概念

Switch语句用于在多个分支中表达条件判断。Go的switch语句非常灵活，支持多种不同的用法，比传统的switch更加强大。

生活化类比：`switch`就像一个酒店前台。你告诉前台你要去哪个房间（`switch`的变量），前台会根据你的房号给你对应的钥匙（执行匹配的`case`）。如果你说的房号不存在（所有`case`都不匹配），前台会告诉你“没有这个房间”（执行`default`分支）。它处理的是“多选一”的问题，比`if-else`更清晰。

7.2 Switch语句的四种主要形式

7.2.1 基本的switch语句

```
i := 2
fmt.Print("Write ", i, " as ")
switch i {
case 1:
    fmt.Println("one")
case 2:
    fmt.Println("two")
case 3:
    fmt.Println("three")
}
```

特点：

- 不需要`break`语句，默认不会穿透到下一个case
- 每个case匹配成功后会自动退出switch

7.2.2 多值匹配和default分支

```
switch time.Now().Weekday() {
case time.Saturday, time.Sunday:
    fmt.Println("It's the weekend")
default:
    fmt.Println("It's a weekday")
}
```

特性：

- 可以在同一个case中使用逗号分隔多个值
- `default`分支处理所有不匹配的情况（可选）

7.2.3 无表达式switch（类似if-else链）

```
t := time.Now()
switch {
case t.Hour() < 12:
    fmt.Println("It's before noon")
default:
    fmt.Println("It's after noon")
}
```

重要特性：

- 没有switch表达式，是表辽if/else逻辑的替代方式
- case表达式可以是非常量，可以是函数调用的结果

7.2.4 类型匹配的switch（Type Switch）

这是Go语言一个非常强大的特性，用于判断一个接口（`interface{}`）变量内部存储的实际类型。

```
whatAmI := func(i interface{}) {
    switch t := i.(type) {
    case bool:
        fmt.Println("I'm a bool")
    case int:
        fmt.Println("I'm an int")
    default:
        fmt.Printf("Don't know type %T\n", t)
    }
}
whatAmI(true)
whatAmI(1)
whatAmI("hey")
```

关键特性解析：

- **`func(i interface{})`**：函数接收一个`interface{}`类型的参数。空接口可以存储任何类型的值，这使得函数具有通用性。
- **`switch t := i.(type)`**：这是类型开关的核心语法。`i.(type)`只能用在`switch`语句中，它会提取接口`i`的动态类型。
- **`t`的类型和值**：在`switch`的初始化语句中，变量`t`被声明。在每个`case`分支中，`t`都会被赋予接口`i`中存储的值，并且其类型就是该`case`所匹配的类型。例如，在`case bool:`分支中，`t`就是一个布尔型变量。
- **`case bool:`** / **`case int:`**：这些分支检查`i`的实际类型是否为`bool`或`int`。
- **`default:`**：如果没有任何`case`匹配成功，`default`分支会被执行。在这个分支中，`t`的类型和值与原始接口变量`i`相同。`%T`格式化占位符用于打印变量的类型。

7.3 完整代码示例

```
package main

import (
```

```
    "fmt"
    "time"
)

func main() {
    // 1. 基本的switch语句
    i := 2
    fmt.Print("Write ", i, " as ")
    switch i {
    case 1:
        fmt.Println("one")
    case 2:
        fmt.Println("two")
    case 3:
        fmt.Println("three")
    }

    // 2. 多值匹配和default分支
    switch time.Now().Weekday() {
    case time.Saturday, time.Sunday:
        fmt.Println("It's the weekend")
    default:
        fmt.Println("It's a weekday")
    }

    // 3. 无表达式switch（类似if-else链）
    t := time.Now()
    switch {
    case t.Hour() < 12:
        fmt.Println("It's before noon")
    default:
        fmt.Println("It's after noon")
    }

    // 4. 类型匹配的switch（Type Switch）
    whatAmI := func(i interface{}) {
        switch t := i.(type) {
        case bool:
            fmt.Println("I'm a bool")
        case int:
            fmt.Println("I'm an int")
        default:
            fmt.Printf("Don't know type %T\n", t)
        }
    }
    whatAmI(true)
    whatAmI(1)
    whatAmI("hey")
}
```

7.4 运行结果

```
$ go run switch.go
Write 2 as two          # 基本switch，i=2匹配case 2
It's a weekday          # 当前不是周末，执行default
It's after noon         # 当前时间大于12点，执行default
I'm a bool             # true的类型是bool
I'm an int             # 1的类型是int
Don't know type string # "hey"的类型是string，执行default
```

7.5 Go Switch的设计特点

7.5.1 与传统switch的区别

```
// Go的switch默认不穿透，不需要break
switch value {
case 1:
    fmt.Println("one")
    // 自动break，不会继续执行case 2
case 2:
    fmt.Println("two")
}

// 如果需要穿透行为，使用fallthrough
switch value {
case 1:
    fmt.Println("one")
    fallthrough  // 继续执行下一个case
case 2:
    fmt.Println("two")
}
```

7.5.2 case表达式的灵活性

```
// case可以是任意表达式，不仅仅是常量
switch {
case time.Now().Hour() < 12:
    fmt.Println("morning")
case isWeekend():
    fmt.Println("weekend")
case temperature > 30:
    fmt.Println("hot")
}
```

7.5.3 类型安全的类型switch

```go
func processValue(v interface{}) {
    switch val := v.(type) {
    case string:
        // val 的类型是 string
        fmt.Printf("String length: %d\n", len(val))
    case int:
        // val 的类型是 int
        fmt.Printf("Integer value: %d\n", val)
    case []byte:
        // val 的类型是 []byte
        fmt.Printf("Byte slice length: %d\n", len(val))
    }
}
```

7.6 关键要点

1. **无需break**：Go的switch默认不穿透，每个case执行后自动退出
2. **多值匹配**：可以在同一个case中匹配多个值，用逗号分隔
3. **灵活表达式**：无表达式switch可以替代复杂的if-else链
4. **类型判断**：Type switch支持运行时类型检查和类型断言
5. **非常量case**：case表达式可以是函数调用或复杂表达式
6. **fallthrough关键字**：需要穿透行为时可以显式使用`fallthrough`

---

8. Arrays (数组)

https://gobyexample.com/arrays

8.1 核心概念

在Go中，数组是一个具有特定长度的、编号的元素序列。数组的长度是其类型的一部分，例如`[5]int`和`[10]int`是两种不同的类型。

生活化类比：数组就像一个鸡蛋盒。一个标准的鸡蛋盒有固定数量的格子（比如12个），这个数量不能改变（固定长度）。每个格子要么放一个鸡蛋，要么是空的（零值）。你可以通过格子的位置（索引，从0开始数）精确地找到、放入或拿出任何一个鸡蛋。

8.2 数组的使用方式

8.2.1 数组的声明和零值

```go
// 声明一个包含5个整数的数组a
// 默认情况下，数组是零值的，对于整数来说就是0
var a [5]int
fmt.Println("emp:", a)
```

特点：声明后，数组`a`的所有元素都被初始化为其类型的零值。

8.2.2 设置和获取元素

```
// 使用 array[index] = value 语法设置值
a[4] = 100
fmt.Println("set:", a)
// 使用 array[index] 获取值
fmt.Println("get:", a[4])
```

说明：数组索引从 `0` 开始。

8.2.3 获取数组长度

```
// 使用内置的len()函数获取数组长度
fmt.Println("len:", len(a))
```

8.2.4 声明并初始化数组

```
// 使用数组字面量在声明时初始化
b := [5]int{1, 2, 3, 4, 5}
fmt.Println("dcl:", b)
```

8.2.5 多维数组

```
// 数组也可以是多维的
var twoD [2][3]int
for i := 0; i < 2; i++ {
    for j := 0; j < 3; j++ {
        twoD[i][j] = i + j
    }
}
fmt.Println("2d: ", twoD)
```

说明：多维数组的声明和初始化与一维数组类似。

8.3 完整代码示例

```
package main

import "fmt"

func main() {

    var a [5]int
    fmt.Println("emp:", a)

    a[4] = 100
    fmt.Println("set:", a)
    fmt.Println("get:", a[4])
```

```
    fmt.Println("len:", len(a))

    b := [5]int{1, 2, 3, 4, 5}
    fmt.Println("dcl:", b)

    b = [...]int{1, 2, 3, 4, 5}
    fmt.Println("dcl:", b)

    b = [...]int{100, 3: 400, 500}
    fmt.Println("idx:", b)

    var twoD [2][3]int
    for i := range 2 {
        for j := range 3 {
            twoD[i][j] = i + j
        }
    }
    fmt.Println("2d: ", twoD)

    twoD = [2][3]int{
        {1, 2, 3},
        {1, 2, 3},
    }
    fmt.Println("2d: ", twoD)
}
```

8.4 运行结果

```
$ go run arrays.go
emp: [0 0 0 0 0]
set: [0 0 0 0 100]
get: 100
len: 5
dcl: [1 2 3 4 5]
dcl: [1 2 3 4 5]
idx: [100 0 0 400 500]
2d:  [[0 1 2] [1 2 3]]
2d:  [[1 2 3] [1 2 3]]
```

8.5 关键要点

1. **固定长度**：数组的长度是其类型的一部分，一旦声明，长度不能改变。
2. **值类型**：数组是值类型。将一个数组赋值给另一个数组会复制所有元素。
3. **零值初始化**：未显式初始化的数组元素会被自动设置为其类型的零值。
4. **索引访问**：通过 `array[index]` 的形式访问元素，索引从0开始。
5. **`len()` 函数**：使用 `len()` 可以获取数组的长度。
6. **多维数组**：Go支持多维数组，可用于构建矩阵等数据结构。

7. **与切片的区别**：数组在Go中不常用，更常用的是更灵活的 **切片 (Slices)**。

---

9. Slices (切片)

https://gobyexample.com/slices

9.1 核心概念

切片是 Go 中处理序列数据的核心工具，它比数组更强大、更灵活。与数组不同，切片的类型只由它所包含的元素决定，而与元素数量无关。切片是一个指向底层数组的引用，包含了指针、长度和容量信息。

生活化类比：如果说数组是固定长度的火车，那么切片就是你在火车上选定的一段连续的车厢。你可以选择从第3节到第5节车厢（这是一个切片），你的朋友可以选择从第4节到第6节（这是另一个切片）。你们看到的都是同一列火车的不同部分。更重要的是，你可以让你的选择范围“变长”（`append`），如果原来的火车（底层数组）还有空余车厢（容量），你就在原来的火车上扩展；如果没有，铁路系统会给你换一列更长的火车，并把原来的乘客和行李都搬过去。

9.2 切片的基本用法

9.2.1 创建切片

```
// 使用内置的 make 函数创建一个非零长度的空切片
// 这里我们创建一个长度为3的字符串切片（初始值为零值 ""）
s := make([]string, 3)
fmt.Println("emp:", s)
```

9.2.2 设置和获取元素

```
// 和数组一样，可以使用索引设置和获取元素
s[0] = "a"
s[1] = "b"
s[2] = "c"
fmt.Println("set:", s)
fmt.Println("get:", s[2])
```

9.2.3 长度

```
// len 返回切片的长度
fmt.Println("len:", len(s))
```

9.2.4 追加元素 (Append)

```
// append 是一个内置函数，用于向切片追加元素
// 它会返回一个包含新元素的、可能指向新底层数组的切片
s = append(s, "d")
s = append(s, "e", "f")
fmt.Println("apd:", s)
```

9.2.5 复制切片 (Copy)

```
// copy 函数可以将一个切片的内容复制到另一个切片
// 这里我们创建一个与s长度相同的新切片c，并将s的内容复制过去
c := make([]string, len(s))
copy(c, s)
fmt.Println("cpy:", c)
```

9.2.6 切片操作 (Slice Operator)

```
// 切片支持 "slice" 操作符，形式为 `slice[low:high]`
// 这会从索引 low 到 high-1 选取元素
l := s[2:5] // 从索引2到4 (5-1)
fmt.Println("sl1:", l)

// 这个切片从索引0到4 (5-1)
l = s[:5]
fmt.Println("sl2:", l)

// 这个切片从索引2到结尾
l = s[2:]
fmt.Println("sl3:", l)
```

9.2.7 声明并初始化

这是创建切片最直接的方式，当你预先知道所有元素时使用。

```
// 使用切片字面量 (slice literal) 在一行内声明并初始化
t := []string{"g", "h", "i"}
fmt.Println("dcl:", t)
```

9.2.8 比较切片 (Comparing Slices)

在Go中，不能直接使用 `==` 来比较两个切片的内容是否相等。`==` 只能用于检查一个切片是否为 `nil`。要比较切片的内容，需要使用 `slices.Equal` 函数（需要导入 `"slices"` 包）。

```
t2 := []string{"g", "h", "i"}
// slices.Equal 检查两个切片是否有相同的长度和对应位置上相等的元素
if slices.Equal(t, t2) {
    fmt.Println("t==t2")
}
```

9.2.9 多维切片 (Multi-dimensional Slices)

切片可以组合成多维数据结构。与多维数组不同，内部切片的长度可以变化，这使得它们非常灵活。

详细逻辑分解

让我们来详细拆解下面这段代码的执行过程：

```go
// 创建一个外层长度为3的二维切片
twoD := make([][]int, 3)

// 遍历外层切片，并为每个元素创建一个不同长度的内层切片
for i := range 3 {
    innerLen := i + 1 // 内层长度分别为 1, 2, 3
    twoD[i] = make([]int, innerLen)
    // 填充内层切片
    for j := range innerLen {
        twoD[i][j] = i + j
    }
}
// 最终得到一个 "锯齿状" 或 "三角形" 的二维切片
fmt.Println("2d:", twoD) // 输出: [[0] [1 2] [2 3 4]]
```

1. 初始状态

代码执行 `twoD := make([][]int, 3)`。

- `[][]int` 定义了这是一个“元素类型为 `[]int` (整数切片)”的切片。
- `make(..., 3)` 指定了这个 **外层切片** 的长度为 3。
- 由于没有提供初始值，Go 会用 `[]int` 类型的 **零值** (`nil`) 来填充它。

此时，`twoD` 的内存状态是：
`twoD -> [nil, nil, nil]`

2. 循环第 1 轮: `i = 0`

- `innerLen := i + 1` 结果为 `1`。
- `twoD[0] = make([]int, 1)`: 创建一个长度为1的整数切片 `[0]`，并赋值给 `twoD` 的第一个位置。
- `twoD` 状态变为: `[[0], nil, nil]`。
- **内层循环** (`j=0`): `twoD[0][0] = 0 + 0`，值被设为 `0`。
- **本轮结束时 `twoD` 状态**: `[[0], nil, nil]`。

3. 循环第 2 轮: `i = 1`

- `innerLen := i + 1` 结果为 `2`。
- `twoD[1] = make([]int, 2)`: 创建一个长度为2的整数切片 `[0, 0]`，并赋值给 `twoD` 的第二个位置。
- `twoD` 状态变为: `[[0], [0, 0], nil]`。
- **内层循环**:
    - `j=0`: `twoD[1][0] = 1 + 0`，值被设为 `1`。
    - `j=1`: `twoD[1][1] = 1 + 1`，值被设为 `2`。
- **本轮结束时 `twoD` 状态**: `[[0], [1, 2], nil]`。

4. 循环第 3 轮: `i = 2`

- `innerLen := i + 1` 结果为 `3`。

- `twoD[2] = make([]int, 3)`: 创建一个长度为3的整数切片 `[0, 0, 0]`，并赋值给 `twoD` 的第三个位置。
- `twoD` 状态变为: `[[0], [1, 2], [0, 0, 0]]`。
- **内层循环**:
  - `j=0`: `twoD[2][0] = 2 + 0`，值被设为 `2`。
  - `j=1`: `twoD[2][1] = 2 + 1`，值被设为 `3`。
  - `j=2`: `twoD[2][2] = 2 + 2`，值被设为 `4`。
- **本轮结束时 `twoD` 状态**: `[[0], [1, 2], [2, 3, 4]]`。

5. 最终结果
循环结束后，`twoD` 的最终值就是 `[[0], [1, 2], [2, 3, 4]]`。这个过程形象地展示了如何构建一个内部长度不一的“锯齿”切片。

9.3 完整代码示例

```
package main

import (
  "fmt"
  "slices"
)

func main() {
  var s []string
  fmt.Println("uninit:", s, s == nil, len(s) == 0)

  s = make([]string, 3)
  fmt.Println("emp:", s, "len:", len(s), "cap:", cap(s))

  s[0] = "a"
  s[1] = "b"
  s[2] = "c"
  fmt.Println("set:", s)
  fmt.Println("get:", s[2])

  fmt.Println("len:", len(s))

  s = append(s, "d")
  s = append(s, "e", "f")
  fmt.Println("apd:", s)

  c := make([]string, len(s))
  copy(c, s)
  fmt.Println("cpy:", c)

  l := s[2:5]
  fmt.Println("sl1:", l)

  l = s[:5]
```

```
  fmt.Println("sl2:", l)

  l = s[2:]
  fmt.Println("sl3:", l)

  t := []string{"g", "h", "i"}
  fmt.Println("dcl:", t)

  t2 := []string{"g", "h", "i"}
  if slices.Equal(t, t2) {
    fmt.Println("t==t2")
  }

  twoD := make([][]int, 3)
  for i := range 3 {
    innerLen := i + 1
    twoD[i] = make([]int, innerLen)
    for j := range innerLen {
      twoD[i][j] = i + j
    }
  }
  fmt.Println("2d:", twoD)

}
```

9.4 运行结果

```
$ go run slices.go
uninit: [] true true
emp: [  ] len: 3 cap: 3
set: [a b c]
get: c
len: 3
apd: [a b c d e f]
cpy: [a b c d e f]
sl1: [c d e]
sl2: [a b c d e]
sl3: [c d e f]
dcl: [g h i]
t==t2
2d: [[0] [1 2] [2 3 4]]
```

9.5 关键要点

1. **动态长度**：切片是动态的，长度不固定，可以通过 `append` 函数增长。
2. **引用类型**：切片是对底层数组的引用。修改切片的元素会影响到底层数组和其他指向同一数组的切片。
3. **`make` 函数**：通常使用 `make([]T, length, capacity)` 来创建切片，可以指定长度和容量。

4. **`append` 和 `copy`**：内置函数 `append` 用于新增元素，`copy` 用于复制切片内容。`append` 可能会导致底层数组的重新分配。
5. **切片操作**：使用 `s[low:high]` 语法可以轻松创建子切片，这是一个非常强大且高效的操作。
6. **零值是 `nil`**：一个未初始化的切片的零值是 `nil`，其长度和容量都为0。
7. **多维切片**：可以创建切片的切片来实现多维数据结构，且内层切片的长度可以不同。

---

10. Maps (映射)

https://gobyexample.com/maps

10.1 核心概念

Map (映射) 是 Go 语言内置的关联数据类型，在其他语言中通常被称为哈希 (hash) 或字典 (dictionary)。Map 用于存储键值对 (key-value pairs) 的集合。与数组和切片不同，Map 的键是无序的。

生活化类比：Map 就像一本真正的字典。你想查一个词（键 `key`），然后找到它的释义（值 `value`）。例如，`map["apple"] = "苹果"`，就是你在字典里找到了 "apple" 这个词，它的意思是“苹果”。你不需要从第一页开始翻，而是可以直接通过词条快速定位，这使得查找效率非常高。

10.2 Map的基本用法

10.2.1 创建Map

使用内置的 `make` 函数创建一个空的 Map。

```
m := make(map[string]int) // 创建一个键为string，值为int的Map
```

10.2.2 设置和获取元素

使用 `map[key] = value` 语法设置键值对，使用 `map[key]` 获取值。

```
m["k1"] = 7
m["k2"] = 13
fmt.Println("map:", m) // 输出: map: map[k1:7 k2:13]

v1 := m["k1"]
fmt.Println("v1:", v1) // 输出: v1: 7
```

如果获取一个不存在的键，会返回该值类型的零值。

```
v3 := m["k3"]
fmt.Println("v3:", v3) // int的零值为0, 输出: v3: 0
```

10.2.3 长度 (len)

使用 `len()` 函数获取 Map 中键值对的数量。

```
fmt.Println("len:", len(m)) // 输出: len: 2
```

10.2.4 删除元素 (delete)

使用内置的 `delete` 函数从 Map 中删除一个键值对。

```
delete(m, "k2")
fmt.Println("map:", m) // 输出: map: map[k1:7]
```

10.2.5 检查键是否存在

从 Map 中获取值时，可以接收一个可选的第二个返回值，该值是一个布尔类型，表示键是否存在。这对于区分“值为零值”和“键不存在”的场景非常有用。

```
_, prs := m["k2"]
fmt.Println("prs:", prs) // k2已被删除, 输出: prs: false
```

10.2.6 声明并初始化

可以在声明时使用 Map 字面量进行初始化。

```
n := map[string]int{"foo": 1, "bar": 2}
fmt.Println("map:", n) // 输出: map: map[bar:2 foo:1]
```

10.2.7 比较Map

从 Go 1.21 开始，可以使用 `maps.Equal` 函数来比较两个 Map 的内容是否相等。

```
n2 := map[string]int{"foo": 1, "bar": 2}
if maps.Equal(n, n2) {
    fmt.Println("n == n2")
}
```

10.3 完整代码示例

```
package main

import (
  "fmt"
  "maps"
)

func main() {
  m := make(map[string]int)

  m["k1"] = 7
  m["k2"] = 13
  fmt.Println("map:", m)
```

```
  v1 := m["k1"]
  fmt.Println("v1:", v1)

  v3 := m["k3"]
  fmt.Println("v3:", v3)

  fmt.Println("len:", len(m))

  delete(m, "k2")
  fmt.Println("map:", m)

  _, prs := m["k2"]
  fmt.Println("prs:", prs)

  n := map[string]int{"foo": 1, "bar": 2}
  fmt.Println("map:", n)

  n2 := map[string]int{"foo": 1, "bar": 2}
  if maps.Equal(n, n2) {
    fmt.Println("n == n2")
  }
}
```

10.4 运行结果

```
$ go run maps.go
map: map[k1:7 k2:13]
v1: 7
v3: 0
len: 2
map: map[k1:7]
prs: false
map: map[bar:2 foo:1]
n == n2
```

10.5 关键要点

1. **无序集合**：Map 是无序的，遍历 Map 时元素的顺序是不确定的。
2. **引用类型**：Map 是引用类型。当将一个 Map 赋值给另一个变量时，它们都指向同一个底层数据结构。
3. **零值是 `nil`**：未初始化的 Map 的零值是 `nil`。对 `nil` map 进行读操作是安全的，但写操作会引发 panic。
4. **并发不安全**：Go 的内置 Map 不是并发安全的。在多个 goroutine 中同时读写一个 Map 需要使用互斥锁等同步机制。
5. **键的类型**：Map 的键必须是可比较的类型（支持 `==` 和 `!=` 操作），如字符串、数字、布尔值、指针、结构体等。切片、函数和包含切片的结构体不能作为键。

11. Functions (函数)

https://gobyexample.com/functions

11.1 核心概念

函数是 Go 语言的核心。函数用于封装代码逻辑，提供代码的复用性和模块化。Go 语言要求显式返回值，即不会自动返回最后一个表达式的值。

生活化类比：函数就像一台榨汁机。你把水果（参数）放进去，它会按照预设的程序（函数体内的代码）进行处理，然后从出口给你一杯果汁（返回值）。这个过程被封装起来了，你不需要关心榨汁机内部的刀片是怎么转的，只需要知道放什么水果进去，就能得到什么果汁出来。这让你可以重复使用这台机器来榨不同的水果。

11.2 函数的基本用法

11.2.1 基本函数声明和调用

```
// 这是一个接收两个 int 参数并返回它们的和的函数
func plus(a int, b int) int {
    return a + b
}
```

特点：

- 使用 `func` 关键字声明函数
- 参数列表在函数名后的圆括号中
- 返回类型在参数列表之后
- Go 要求显式使用 `return` 语句返回值

11.2.2 相同类型参数的简化写法

```
// 当有多个连续的相同类型参数时，可以省略前面参数的类型声明
func plusPlus(a, b, c int) int {
    return a + b + c
}
```

特点：可以省略除最后一个参数之外的类型声明，只在最后一个参数处声明类型。

11.2.3 函数调用

```
// 使用 函数名(参数) 的方式调用函数
res := plus(1, 2)
res = plusPlus(1, 2, 3)
```

特点：函数调用语法简洁明了，与大多数编程语言一致。

11.3 完整代码示例

```
package main
```

```go
import "fmt"

// 这是一个接收两个 int 参数并返回它们的和的函数
func plus(a int, b int) int {
    // Go 要求显式返回，不会自动返回最后一个表达式的值
    return a + b
}

// 当有多个连续的相同类型参数时，可以省略前面参数的类型声明
// 只需要在最后一个参数处声明类型
func plusPlus(a, b, c int) int {
    return a + b + c
}

func main() {
    // 使用 函数名(参数) 的方式调用函数
    res := plus(1, 2)
    fmt.Println("1+2 =", res)

    res = plusPlus(1, 2, 3)
    fmt.Println("1+2+3 =", res)
}
```

11.4 运行结果

```
$ go run functions.go
1+2 = 3
1+2+3 = 6
```

11.5 关键要点

1. **显式返回**：Go 要求使用 `return` 语句明确返回值，不会自动返回最后一个表达式
2. **类型声明**：函数参数必须声明类型，但相同类型的连续参数可以简化写法
3. **函数是一等公民**：函数可以作为值传递、赋值给变量、作为参数传递给其他函数
4. **命名规范**：函数名遵循驼峰命名法，公开函数以大写字母开头，私有函数以小写字母开头
5. **多样化特性**：Go 函数支持多返回值、可变参数、闭包等高级特性
6. **模块化设计**：函数是 Go 语言代码组织和复用的基本单元

---

12. Multiple Return Values (多返回值)

https://gobyexample.com/multiple-return-values

12.1 核心概念

Go 内置支持多返回值功能。这个特性在惯用的 Go 代码中经常使用，例如从函数中同时返回结果和错误值。多返回值使得 Go 函数能够一次性返回多个相关的数据，提高了代码的表达能力和实用性。

生活化类比：多返回值就像去餐厅点一份套餐。你只下了一次单（调用一次函数），但服务员会同时给你端来汉堡和可乐（返回多个值）。在Go中，一个常见的“套餐”是“结果”和“一张小票（错误信息）”。你拿到结果的同时，也拿到一张小票，如果小票是空的（`err`为`nil`），说明交易成功；如果小票上写着“库存不足”（`err`有内容），你就知道结果可能有问题。

12.2 多返回值的基本用法

12.2.1 声明多返回值函数

```
// 函数签名中的 (int, int) 表示该函数返回两个 int 值
func vals() (int, int) {
    return 3, 7
}
```

特点：

- 在函数签名中用圆括号包围多个返回类型
- 使用逗号分隔不同的返回类型
- `return` 语句中用逗号分隔多个返回值

12.2.2 接收多个返回值

```
// 使用多重赋值从函数调用中接收多个返回值
a, b := vals()
fmt.Println(a)  // 输出: 3
fmt.Println(b)  // 输出: 7
```

特点：使用多重赋值语法 `:=` 同时接收多个返回值。

12.2.3 使用空白标识符忽略返回值

```
// 如果只需要部分返回值，可以使用空白标识符 _ 忽略不需要的值
_, c := vals()
fmt.Println(c)  // 输出: 7，忽略了第一个返回值
```

重要特性：

- `_` (下划线) 是 Go 中的空白标识符
- 用于忽略不需要的返回值
- 可以用在任意位置忽略对应的返回值

12.3 完整代码示例

```
package main

import "fmt"
```

```go
// 函数签名中的 (int, int) 表示该函数返回两个 int 值
func vals() (int, int) {
    return 3, 7
}

func main() {
    // 使用多重赋值从函数调用中接收多个返回值
    a, b := vals()
    fmt.Println(a)
    fmt.Println(b)

    // 如果只需要部分返回值，可以使用空白标识符 _ 忽略不需要的值
    _, c := vals()
    fmt.Println(c)
}
```

12.4 运行结果

```
$ go run multiple-return-values.go
3
7
7
```

12.5 关键要点

1. **内置支持**：Go 语言原生支持多返回值，无需特殊语法或库
2. **惯用模式**：多返回值在 Go 中是常见的惯用法，特别是返回值和错误的组合
3. **类型安全**：所有返回值都必须在函数签名中明确声明类型
4. **灵活接收**：可以接收全部返回值，也可以使用空白标识符忽略部分返回值
5. **多重赋值**：支持一次性将多个返回值赋给多个变量
6. **错误处理模式**：通常与错误处理结合使用，如 `result, err := someFunction()`
7. **性能优势**：避免了使用结构体或其他复合类型来返回多个值的开销

---

13. Variadic Functions (可变参数函数)

https://gobyexample.com/variadic-functions

13.1 核心概念

可变参数函数（Variadic functions）是一种可以接受任意数量的尾随参数的函数。例如，`fmt.Println` 就是一个常见的可变参数函数。

生活化类比：可变参数函数就像一个可以装任意数量物品的购物篮。你去超市，可以放一个苹果，也可以放十个苹果，还可以再加一瓶牛奶（传入不同数量的参数）。这个购物篮（可变参数）会把所有东西都装进去，方便你统一处理（在函数内部，这些参数会变成一个切片）。

13.2 可变参数函数的基本用法

13.2.1 定义可变参数函数

通过在参数类型前加上 `...` 来定义一个可变参数函数。在函数内部，这个参数会变成一个相应类型的切片。

```
// 这个函数接收任意数量的 int 作为参数
func sum(nums ...int) {
    fmt.Print(nums, " ") // nums 的类型是 []int
    total := 0
    for _, num := range nums {
        total += num
    }
    fmt.Println(total)
}
```

说明：在 `sum` 函数内部，`nums` 的类型等价于 `[]int`，因此可以对其使用 `len()`、`range` 等操作。

13.2.2 使用独立参数调用

可以像普通函数一样，传入任意数量的独立参数来调用可变参数函数。

```
sum(1, 2)
sum(1, 2, 3)
```

13.2.3 使用切片调用

如果参数已经存在于一个切片中，可以通过在切片名后加上 `...` 的方式，将切片中的所有元素作为独立参数传递给函数。

```
nums := []int{1, 2, 3, 4}
sum(nums...) // 将切片 nums 的所有元素展开并传入
```

13.3 完整代码示例

```
package main

import "fmt"

// 这是一个接收任意数量 int 参数的函数
func sum(nums ...int) {
    fmt.Print(nums, " ")
    total := 0
    // 在函数内部，nums 的类型等价于 []int
    for _, num := range nums {
        total += num
    }
    fmt.Println(total)
}
```

```go
func main() {
    // 使用独立参数调用
    sum(1, 2)
    sum(1, 2, 3)

    // 如果参数已存在于切片中，使用 func(slice...) 的方式调用
    nums := []int{1, 2, 3, 4}
    sum(nums...)
}
```

13.4 运行结果

```
$ go run variadic-functions.go
[1 2] 3
[1 2 3] 6
[1 2 3 4] 10
```

13.5 关键要点

1. `...` **语法**：在函数参数类型前使用 `...` 来定义可变参数。
2. **内部是切片**：可变参数在函数内部被当作一个切片来处理。
3. **两种调用方式**：可以直接传入多个独立参数，也可以通过 `slice...` 的语法将一个切片展开后传入。
4. **位置限制**：可变参数必须是函数的最后一个参数。
5. **灵活性**：可变参数函数为处理不定数量的输入提供了极大的便利。

14. Closures (闭包)

https://gobyexample.com/closures

14.1 核心概念

Go 支持匿名函数，匿名函数可以形成闭包。匿名函数在你想要定义一个不需要命名的内联函数时非常有用。闭包是一个函数值，它引用了其函数体之外的变量。

生活化类比：闭包就像一个拥有“专属记忆”的机器人。你启动一个计数机器人（调用`intSeq()`），它内部有一个自己的计数器`i`。这个机器人（返回的函数）被派给你（赋值给`nextInt`），它就只认你。你每次戳它一下（调用`nextInt()`），它就在自己的记忆里把数字加一。别人再启动一个新的计数机器人（调用`intSeq()`赋值给`newInts`），那个新机器人会有自己独立的记忆，从头开始计数，跟你的机器人互不影响。

14.2 闭包的基本用法

14.2.1 返回匿名函数的函数

```go
// intSeq 函数返回另一个函数，这个返回的函数在 intSeq 的函数体内匿名定义
// 返回的函数对变量 i 形成闭包
func intSeq() func() int {
    i := 0
    return func() int {
        i++
        return i
    }
}
```

特点：

- 外层函数 `intSeq` 返回一个匿名函数
- 匿名函数"捕获"了外层函数的变量 `i`
- 每次调用返回的函数时，都会修改并访问同一个 `i` 变量

14.2.2 调用闭包函数

```go
// 调用 intSeq，将结果（一个函数）赋值给 nextInt
// 这个函数值捕获了它自己的 i 值，每次调用 nextInt 时都会更新这个值
nextInt := intSeq()
```

14.2.3 观察闭包的效果

```go
// 通过多次调用 nextInt 来观察闭包的效果
fmt.Println(nextInt()) // 输出：1
fmt.Println(nextInt()) // 输出：2
fmt.Println(nextInt()) // 输出：3
```

关键特性：每次调用 `nextInt()`，内部的 `i` 变量都会递增，状态被保持在闭包中。

14.2.4 独立的闭包状态

```go
// 为了确认状态对于特定函数是唯一的，创建并测试一个新的闭包
newInts := intSeq()
fmt.Println(newInts()) // 输出：1（重新开始计数）
```

重要概念：每个闭包都维护自己独立的状态，不同的闭包实例之间不会相互影响。

14.3 完整代码示例

```go
package main

import "fmt"

// intSeq 函数返回另一个函数，这个返回的函数在 intSeq 的函数体内匿名定义
// 返回的函数对变量 i 形成闭包
func intSeq() func() int {
```

```
    i := 0
    return func() int {
        i++
        return i
    }
}

func main() {
    // 调用 intSeq，将结果（一个函数）赋值给 nextInt
    // 这个函数值捕获了它自己的 i 值，每次调用 nextInt 时都会更新这个值
    nextInt := intSeq()

    // 通过多次调用 nextInt 来观察闭包的效果
    fmt.Println(nextInt())
    fmt.Println(nextInt())
    fmt.Println(nextInt())

    // 为了确认状态对于特定函数是唯一的，创建并测试一个新的闭包
    newInts := intSeq()
    fmt.Println(newInts())
}
```

14.4 运行结果

```
$ go run closures.go
1
2
3
1
```

14.5 关键要点

1. **匿名函数**：Go 支持在函数内部定义匿名函数，无需为其命名
2. **变量捕获**：闭包可以访问和修改其外层函数的变量，形成"捕获"关系
3. **状态保持**：闭包能够在多次调用之间保持状态，变量的值会被持久化
4. **独立实例**：每个闭包实例都有自己独立的变量副本，互不干扰
5. **函数作为返回值**：Go 支持函数作为返回值，这是实现闭包的基础
6. **内存管理**：Go 的垃圾回收器会自动管理闭包捕获的变量的生命周期
7. **实用场景**：闭包常用于回调函数、事件处理、状态机等场景

15. Recursion (递归)

https://gobyexample.com/recursion

15.1 核心概念

Go 支持递归函数。递归是一种函数调用自身来解决问题的编程技术。递归函数通常包含一个基础情况（终止条件）和一个递归情况（函数调用自身）。

生活化类比：递归就像是俄罗斯套娃。你想找到最里面的那个小娃娃（基础情况），你的操作方法（递归函数）是“打开当前的娃娃”。当你打开一个，会发现里面还有一个更小的娃娃，于是你对这个小娃娃重复同样的操作，直到你打开一个发现里面是实心的，不能再开了，你就找到了。

15.2 递归的基本用法

15.2.1 经典递归函数 - 阶乘

```
// fact 函数调用自身直到达到基础情况 fact(0)
func fact(n int) int {
    if n == 0 {
        return 1  // 基础情况：0! = 1
    }
    return n * fact(n-1)  // 递归情况：n! = n * (n-1)!
}
```

特点：

- **基础情况**：`n == 0` 时返回 1，这是递归的终止条件
- **递归情况**：函数调用自身 `fact(n-1)`，每次递归参数都在减小
- **数学原理**：n! = n × (n-1) × (n-2) × ... × 1

15.2.2 调用递归函数

```
fmt.Println(fact(7))  // 计算 7! = 5040
```

15.2.3 匿名递归函数 - 斐波那契数列

匿名函数也可以是递归的，但这需要在定义函数之前用 `var` 显式声明一个变量来存储函数。

```
// 首先声明一个函数类型的变量
var fib func(n int) int

// 然后将匿名递归函数赋值给这个变量
fib = func(n int) int {
    if n < 2 {
        return n  // 基础情况：fib(0)=0, fib(1)=1
    }
    // 由于 fib 之前在 main 中声明过，Go 知道这里调用的是哪个函数
    return fib(n-1) + fib(n-2)  // 递归情况：fib(n) = fib(n-1) + fib(n-2)
}
```

重要说明：

- 必须先用 `var` 声明函数变量，然后再赋值
- 这样做是因为在匿名函数内部需要引用函数自身

- 如果不先声明，编译器无法识别递归调用中的函数名

15.2.4 调用匿名递归函数

```
fmt.Println(fib(7))  // 计算斐波那契数列第7项 = 13
```

15.3 完整代码示例

```
package main

import "fmt"

// fact 函数调用自身直到达到基础情况 fact(0)
func fact(n int) int {
    if n == 0 {
        return 1
    }
    return n * fact(n-1)
}

func main() {
    fmt.Println(fact(7))

    // 匿名函数也可以是递归的，但这需要在定义函数之前
    // 用 var 显式声明一个变量来存储函数
    var fib func(n int) int

    fib = func(n int) int {
        if n < 2 {
            return n
        }
        // 由于 fib 之前在 main 中声明过，Go 知道这里调用的是哪个函数
        return fib(n-1) + fib(n-2)
    }

    fmt.Println(fib(7))
}
```

15.4 运行结果

```
$ go run recursion.go
5040
13
```

15.5 关键要点

1. **基础情况**：每个递归函数都必须有一个或多个基础情况来终止递归
2. **递归情况**：函数必须在某个点调用自身，通常使用修改过的参数

3. **参数变化**：每次递归调用的参数都应该朝着基础情况的方向变化
4. **匿名递归**：匿名函数的递归需要先用 `var` 声明函数变量
5. **性能考虑**：递归可能导致栈溢出或性能问题，特别是对于深度递归
6. **经典应用**：阶乘、斐波那契数列、树遍历、分治算法等
7. **调试技巧**：递归函数的调试需要仔细跟踪调用栈和参数变化

---

16. Range over Built-in Types (遍历内置类型)

https://gobyexample.com/range

16.1 核心概念

`range` 用于遍历各种内置数据结构中的元素。我们可以使用 `range` 来遍历之前学过的数据结构，如切片、数组、映射和字符串。`range` 提供了一种统一且简洁的方式来迭代不同类型的集合。

生活化类比：`range` 就像是进行一次“巡视”。如果你有一个队伍（切片或数组），`range` 会带着你从第一个人开始，逐个点名（提供索引和值）。如果你有一份清单（映射），`range` 会帮你把清单上的每一项（键和值）都读出来。如果你有一句话（字符串），`range` 则会带你逐个看这句话里的每个字（rune）。它是一个通用的、帮你“挨个看一遍”的工具。

16.2 Range的基本用法

16.2.1 遍历切片和数组

```
nums := []int{2, 3, 4}
sum := 0
// range 返回索引和值，这里用 _ 忽略索引
for _, num := range nums {
    sum += num
}
fmt.Println("sum:", sum)
```

特点：

- `range` 在数组和切片上提供索引和值
- 使用空白标识符 `_` 可以忽略不需要的索引
- 数组的遍历方式与切片完全相同

16.2.2 同时获取索引和值

```
// 有时我们确实需要索引
for i, num := range nums {
    if num == 3 {
        fmt.Println("index:", i)
    }
}
```

说明：`range` 返回两个值：第一个是索引，第二个是该位置的元素值。

16.2.3 遍历映射 (Map)

```
kvs := map[string]string{"a": "apple", "b": "banana"}
// range 在 map 上迭代键值对
for k, v := range kvs {
    fmt.Printf("%s -> %s\n", k, v)
}
```

特点：

- `range` 在映射上返回键值对
- 第一个返回值是键 (key)，第二个是值 (value)
- 映射的遍历顺序是不确定的

16.2.4 只遍历映射的键

```
// range 也可以只遍历 map 的键
for k := range kvs {
    fmt.Println("key:", k)
}
```

用法：当只需要键而不需要值时，可以省略第二个返回值。

16.2.5 遍历字符串

```
// range 在字符串上遍历 Unicode rune
// 第一个值是 rune 的起始字节索引，第二个是 rune 本身
for i, c := range "go" {
    fmt.Println(i, c)
}
```

重要概念：

- 字符串的 `range` 遍历的是 Unicode rune (rune)，不是字节
- 第一个返回值是该 rune 在字符串中的字节索引
- 第二个返回值是 rune 的 Unicode rune值
- 对于 ASCII 字符，一个字符占一个字节；对于非 ASCII 字符，可能占多个字节

16.3 完整代码示例

```
package main

import "fmt"

func main() {
    // 使用 range 来计算切片中数字的和，数组也是这样工作的
```

```
    nums := []int{2, 3, 4}
    sum := 0
    for _, num := range nums {
        sum += num
    }
    fmt.Println("sum:", sum)

    // range 在数组和切片上提供索引和值
    // 上面我们不需要索引，所以用空白标识符 _ 忽略了它
    // 有时我们确实需要索引
    for i, num := range nums {
        if num == 3 {
            fmt.Println("index:", i)
        }
    }

    // range 在 map 上迭代键值对
    kvs := map[string]string{"a": "apple", "b": "banana"}
    for k, v := range kvs {
        fmt.Printf("%s -> %s\n", k, v)
    }

    // range 也可以只遍历 map 的键
    for k := range kvs {
        fmt.Println("key:", k)
    }

    // range 在字符串上遍历 Unicode rune
    // 第一个值是 rune 的起始字节索引，第二个是 rune 本身
    for i, c := range "go" {
        fmt.Println(i, c)
    }
}
```

16.4 运行结果

```
$ go run range-over-built-in-types.go
sum: 9
index: 1
a -> apple
b -> banana
key: a
key: b
0 103
1 111
```

16.5 关键要点

1. **统一语法**：`range` 为不同数据类型提供了统一的遍历语法
2. **双返回值**：`range` 通常返回两个值，具体含义取决于数据类型

3. **灵活使用**：可以使用空白标识符 `_` 忽略不需要的返回值
4. **切片/数组**：返回索引和元素值
5. **映射**：返回键和值，遍历顺序不确定
6. **字符串**：返回字节索引和 Unicode rune，支持多字节字符
7. **性能优势**：`range` 是遍历集合的推荐方式，性能优于手动索引循环

---

17. Pointers (指针)

https://gobyexample.com/pointers

17.1 核心概念

Go 支持指针，允许在程序中传递值和记录的引用。指针存储的是变量的内存地址，而不是变量的值本身。通过指针，我们可以直接访问和修改内存中的数据，这在需要高效传递大型数据结构或需要在函数中修改原始数据时非常有用。

生活化类比：变量 `i` 就像一座房子，里面住着它的值（比如数字 `1`）。当你把房子本身（值传递 `zeroval(i)`）给别人看时，别人只是看到了一个房子的复制品，他们对复制品做什么，都影响不到你真正的房子。
而指针 `&i` 则是你家房子的地址。当你把地址（指针传递 `zeroptr(&i)`）告诉别人时，他们就能根据地址找到你真正的房子，然后进到房子里去修改里面的东西（比如把数字 `1` 改成 `0`）。

17.2 指针的基本用法

17.2.1 值传递函数

```
// zeroval 有一个 int 参数，所以参数将按值传递给它
// zeroval 将得到 ival 的一个副本，与调用函数中的变量是不同的
func zeroval(ival int) {
    ival = 0  // 只修改副本，不影响原始变量
}
```

特点：

- 参数按值传递，函数接收的是原始值的副本
- 在函数内修改参数不会影响原始变量
- 这是 Go 中的默认传递方式

17.2.2 指针传递函数

```
// zeroptr 有一个 *int 参数，意味着它接收一个 int 指针
// 函数体中的 *iptr 代码将指针从其内存地址解引用到该地址的当前值
// 给解引用的指针赋值会改变引用地址处的值
func zeroptr(iptr *int) {
    *iptr = 0  // 通过指针修改原始变量的值
}
```

关键概念：

- `*int` 表示指向 int 类型的指针
- `*iptr` 是解引用操作，获取指针指向的值
- 通过解引用可以直接修改原始变量

17.2.3 获取变量的地址

```
i := 1
fmt.Println("initial:", i)

// &i 语法给出 i 的内存地址，即指向 i 的指针
zeroptr(&i)
```

地址操作符：

- `&` 操作符用于获取变量的内存地址
- `&i` 返回变量 i 的指针

17.2.4 对比值传递和指针传递

```
// 值传递：不会改变原始变量
zeroval(i)
fmt.Println("zeroval:", i)  // i 仍然是原来的值

// 指针传递：会改变原始变量
zeroptr(&i)
fmt.Println("zeroptr:", i)  // i 被修改为 0
```

17.2.5 打印指针

```
// 指针也可以被打印出来
fmt.Println("pointer:", &i)  // 输出内存地址
```

17.3 完整代码示例

```
package main

import "fmt"

// zeroval 有一个 int 参数，所以参数将按值传递给它
// zeroval 将得到 ival 的一个副本，与调用函数中的变量是不同的
func zeroval(ival int) {
    ival = 0
}

// zeroptr 有一个 *int 参数，意味着它接收一个 int 指针
// 函数体中的 *iptr 代码将指针从其内存地址解引用到该地址的当前值
```

```go
// 给解引用的指针赋值会改变引用地址处的值
func zeroptr(iptr *int) {
    *iptr = 0
}

func main() {
    i := 1
    fmt.Println("initial:", i)

    zeroval(i)
    fmt.Println("zeroval:", i)

    // &i 语法给出 i 的内存地址，即指向 i 的指针
    zeroptr(&i)
    fmt.Println("zeroptr:", i)

    // 指针也可以被打印出来
    fmt.Println("pointer:", &i)
}
```

17.4 运行结果

```
$ go run pointers.go
initial: 1
zeroval: 1
zeroptr: 0
pointer: 0x42131100
```

17.5 关键要点

1. **值传递 vs 指针传递**：`zeroval` 不会改变 `main` 中的 `i`，但 `zeroptr` 会，因为它有对该变量内存地址的引用
2. **指针声明**：使用 `*T` 声明指向类型 T 的指针
3. **地址操作符**：使用 `&` 获取变量的内存地址
4. **解引用操作符**：使用 `*` 访问指针指向的值
5. **内存效率**：指针传递避免了大型数据结构的复制
6. **直接修改**：通过指针可以在函数中直接修改原始数据
7. **安全性**：Go 的指针是安全的，不支持指针算术运算，避免了内存安全问题

---

18. Strings and Runes (字符串和符文)

https://gobyexample.com/strings-and-runes

18.1 核心概念

Go 字符串是一个只读的字节切片。语言和标准库对字符串进行特殊处理 - 作为以 UTF-8 编码的文本容器。在其他语言中，字符串由"字符"组成。在 Go 中，字符的概念被称为 `rune` - 它是一个表示 Unicode rune的整数。

生活化类比：把 Go 的字符串想象成一串由不同大小的珠子串成的项链。`len()` 函数告诉你的是制作这条项链总共用了多长的线（字节数）。而 `rune` 则是项链上每一颗独立的珠子（字符）。对于像 'a', 'b' 这样的小珠子，它占的线长度是1。但对于像 '你', '好' 这样的大珠子，它可能需要占3个长度的线才能串起来。`for range` 循环则能智能地识别每一颗珠子，而不是只看线的长度。

18.2 字符串和rune的基本用法

18.2.1 UTF-8 编码的字符串

```
// s 是一个字符串，被赋予一个字面值，表示泰语中的"hello"
// Go 字符串字面值是 UTF-8 编码的文本
const s = "สวัสดี"
```

特点：Go 字符串字面值默认使用 UTF-8 编码，可以包含任何 Unicode 字符。

18.2.2 字符串长度（字节数）

```
// 由于字符串等价于 []byte，这将产生存储在其中的原始字节的长度
fmt.Println("Len:", len(s))
```

重要概念：`len()` 函数返回的是字符串的字节数，不是字符数。对于 UTF-8 编码的非 ASCII 字符，一个字符可能占用多个字节。

18.2.3 字节级别的访问

```
// 对字符串进行索引会产生每个索引处的原始字节值
// 这个循环生成构成 s 中rune的所有字节的十六进制值
for i := 0; i < len(s); i++ {
    fmt.Printf("%x ", s[i])
}
fmt.Println()
```

说明：直接索引字符串访问的是字节，不是字符。对于多字节的 UTF-8 字符，需要多个字节来表示一个字符。

18.2.4 计算rune数量

```
// 要计算字符串中有多少个 rune，我们可以使用 utf8 包
// 注意 RuneCountInString 的运行时间取决于字符串的大小，
// 因为它必须顺序解码每个 UTF-8 rune
fmt.Println("Rune count:", utf8.RuneCountInString(s))
```

关键特性：

- `utf8.RuneCountInString()` 返回字符串中 Unicode 字符的实际数量
- 泰语等语言的某些字符由可以跨越多个字节的 UTF-8 rune表示

18.2.5 使用 range 遍历rune

```
// range 循环对字符串进行特殊处理，解码每个 rune 及其在字符串中的偏移量
for idx, runeValue := range s {
    fmt.Printf("%#U starts at %d\n", runeValue, idx)
}
```

特点：

- `range` 在字符串上会自动解码 UTF-8 字符
- 返回字节索引和 rune 值
- `%#U` 格式化显示 Unicode rune和字符

18.2.6 手动解码rune

```
// 我们可以通过显式使用 utf8.DecodeRuneInString 函数来实现相同的迭代
fmt.Println("\nUsing DecodeRuneInString")
for i, w := 0, 0; i < len(s); i += w {
    runeValue, width := utf8.DecodeRuneInString(s[i:])
    fmt.Printf("%#U starts at %d\n", runeValue, i)
    w = width
    examineRune(runeValue)
}
```

详细解析：

- `utf8.DecodeRuneInString()` 手动解码字符串中的 rune
- 返回 rune 值和该 rune 占用的字节数
- 通过字节宽度来正确移动索引位置

18.2.7 rune字面值和比较

```
func examineRune(r rune) {
    // 用单引号括起来的值是 rune 字面值
    // 我们可以直接将 rune 值与 rune 字面值进行比较
    if r == 't' {
        fmt.Println("found tee")
    } else if r == 'ส' {
        fmt.Println("found so sua")
    }
}
```

特点：

- 单引号 `'t'` 表示 rune 字面值
- 可以直接比较 rune 值
- 支持任何 Unicode 字符的字面值

18.3 完整代码示例

```
package main

import (
    "fmt"
    "unicode/utf8"
)

func main() {
    // s 是一个字符串，被赋予一个字面值，表示泰语中的"hello"
    // Go 字符串字面值是 UTF-8 编码的文本
    const s = "สวัสดี"

    // 由于字符串等价于 []byte，这将产生存储在其中的原始字节的长度
    fmt.Println("Len:", len(s))

    // 对字符串进行索引会产生每个索引处的原始字节值
    // 这个循环生成构成 s 中rune的所有字节的十六进制值
    for i := 0; i < len(s); i++ {
        fmt.Printf("%x ", s[i])
    }
    fmt.Println()

    // 要计算字符串中有多少个 rune，我们可以使用 utf8 包
    fmt.Println("Rune count:", utf8.RuneCountInString(s))

    // range 循环对字符串进行特殊处理，解码每个 rune 及其在字符串中的偏移量
    for idx, runeValue := range s {
        fmt.Printf("%#U starts at %d\n", runeValue, idx)
    }

    // 我们可以通过显式使用 utf8.DecodeRuneInString 函数来实现相同的迭代
    fmt.Println("\nUsing DecodeRuneInString")
    for i, w := 0, 0; i < len(s); i += w {
        runeValue, width := utf8.DecodeRuneInString(s[i:])
        fmt.Printf("%#U starts at %d\n", runeValue, i)
        w = width
        examineRune(runeValue)
    }
func examineRune(r rune) {
    // 用单引号括起来的值是 rune 字面值
    // 我们可以直接将 rune 值与 rune 字面值进行比较
    if r == 't' {
        fmt.Println("found tee")
    } else if r == 'ส' {
        fmt.Println("found so sua")
    }
}
```

18.4 运行结果

```
$ go run strings-and-runes.go
Len: 18
e0 b8 aa e0 b8 a7 e0 b8 b1 e0 b8 aa e0 b8 94 e0 b8 b5
Rune count: 6
U+0E2A 'ส' starts at 0
U+0E27 'ว' starts at 3
U+0E31 'ั' starts at 6
U+0E2A 'ส' starts at 9
U+0E14 'ด' starts at 12
U+0E35 'ี' starts at 15

Using DecodeRuneInString
U+0E2A 'ส' starts at 0
found so sua
U+0E27 'ว' starts at 3
U+0E31 'ั' starts at 6
U+0E2A 'ส' starts at 9
found so sua
U+0E14 'ด' starts at 12
U+0E35 'ี' starts at 15
```

18.5 关键要点

1. **UTF-8 编码**：Go 字符串默认使用 UTF-8 编码，支持所有 Unicode 字符
2. **字节 vs 字符**：`len()` 返回字节数，`utf8.RuneCountInString()` 返回字符数
3. **多字节字符**：非 ASCII 字符可能占用多个字节（如泰语字符占用 3 个字节）
4. **rune 类型**：rune 是 int32 的别名，表示一个 Unicode rune
5. **range 特殊处理**：`range` 在字符串上会自动解码 UTF-8 字符
6. **手动解码**：可以使用 `utf8.DecodeRuneInString()` 手动处理字符串
7. **国际化支持**：Go 的字符串设计天然支持国际化和多语言文本处理

---

19. Structs (结构体)

https://gobyexample.com/structs

19.1 核心概念

Go 的结构体是字段的类型化集合。它们对于将数据组合在一起形成记录非常有用。结构体是 Go 中创建自定义数据类型的主要方式，类似于其他语言中的类或对象，但更加简洁。

生活化类比：结构体就像一张个人信息登记表（模板）。这张表上规定了需要填写“姓名”（`name` 字段）和“年龄”（`age` 字段）。每个人（结构体实例）都可以拿这张表来填写自己的信息，形成一份份独立的、包含固定字段的档案。

19.2 结构体的基本用法

19.2.1 定义结构体类型

```
// 这个 person 结构体类型有 name 和 age 字段
type person struct {
    name string
    age  int
}
```

特点：

- 使用 `type` 关键字定义新的结构体类型
- 结构体包含多个命名字段，每个字段都有特定的类型
- 字段名遵循 Go 的可见性规则（大写开头为公开，小写开头为私有）

19.2.2 构造函数模式

```
// newPerson 构造一个具有给定名称的新 person 结构体
func newPerson(name string) *person {
    p := person{name: name}
    p.age = 42
    return &p
}
```

重要概念：

- Go 是垃圾回收语言，可以安全地返回指向局部变量的指针
- 只有在没有活动引用时，垃圾回收器才会清理它
- 惯用做法是将新结构体的创建封装在构造函数中

19.2.3 创建结构体实例的多种方式

19.2.3.1 按位置初始化

```
// 这种语法创建一个新的结构体
fmt.Println(person{"Bob", 20})
```

19.2.3.2 命名字段初始化

```
// 初始化结构体时可以命名字段
fmt.Println(person{name: "Alice", age: 30})
```

19.2.3.3 部分初始化（零值）

```
// 省略的字段将被零值化
fmt.Println(person{name: "Fred"})  // age 将为 0
```

19.2.3.4 获取结构体指针

```
// & 前缀产生指向结构体的指针
fmt.Println(&person{name: "Ann", age: 40})
```

19.2.4 访问和修改结构体字段

```
// 使用点号访问结构体字段
s := person{name: "Sean", age: 50}
fmt.Println(s.name)

// 也可以对结构体指针使用点号 - 指针会被自动解引用
sp := &s
fmt.Println(sp.age)

// 结构体是可变的
sp.age = 51
fmt.Println(sp.age)
```

关键特性：

- 使用点号 `.` 访问结构体字段
- 对于结构体指针，Go 会自动解引用，无需使用 `*`
- 结构体字段可以直接修改（如果结构体本身是可变的）

19.2.5 匿名结构体

```
// 如果结构体类型只用于单个值，我们不必给它命名
// 该值可以具有匿名结构体类型
// 这种技术通常用于表驱动测试
dog := struct {
    name   string
    isGood bool
}{
    "Rex",
    true,
}
fmt.Println(dog)
```

用途：

- 适用于一次性使用的数据结构
- 常用于测试中的表驱动测试
- 减少代码中不必要的类型定义

19.3 完整代码示例

```
package main
```

```go
import "fmt"

// 这个 person 结构体类型有 name 和 age 字段
type person struct {
    name string
    age  int
}

// newPerson 构造一个具有给定名称的 *person 结构体
func newPerson(name string) *person {
    // Go 是垃圾回收语言，可以安全地返回指向局部变量的指针
    // 只有在没有活动引用时，垃圾回收器才会清理它
    p := person{name: name}
    p.age = 42
    return &p
}

func main() {
    // 这种语法创建一个新的结构体
    fmt.Println(person{"Bob", 20})

    // 初始化结构体时可以命名字段
    fmt.Println(person{name: "Alice", age: 30})

    // 省略的字段将被零值化
    fmt.Println(person{name: "Fred"})

    // & 前缀产生指向结构体的指针
    fmt.Println(&person{name: "Ann", age: 40})

    // 惯用做法是将新结构体的创建封装在构造函数中
    fmt.Println(newPerson("Jon"))

    // 使用点号访问结构体字段
    s := person{name: "Sean", age: 50}
    fmt.Println(s.name)

    // 也可以对结构体指针使用点号 - 指针会被自动解引用
    sp := &s
    fmt.Println(sp.age)

    // 结构体是可变的
    sp.age = 51
    fmt.Println(sp.age)

    // 如果结构体类型只用于单个值，我们不必给它命名
    // 该值可以具有匿名结构体类型
    // 这种技术通常用于表驱动测试
    dog := struct {
        name   string
        isGood bool
```

```
    }{
        "Rex",
        true,
    }
    fmt.Println(dog)
}
```

19.4 运行结果

```
$ go run structs.go
{Bob 20}
{Alice 30}
{Fred 0}
&{Ann 40}
&{Jon 42}
Sean
50
51
{Rex true}
```

19.5 关键要点

1. **数据组织**：结构体是 Go 中组织相关数据的主要方式
2. **类型安全**：结构体提供编译时类型检查，确保字段访问的安全性
3. **零值安全**：未初始化的字段会自动设置为其类型的零值
4. **指针友好**：Go 自动处理结构体指针的解引用，语法简洁
5. **构造函数模式**：虽然 Go 没有构造函数，但可以用函数模拟这种模式
6. **内存安全**：垃圾回收器自动管理结构体的内存生命周期
7. **灵活初始化**：支持位置初始化、命名字段初始化和部分初始化
8. **匿名结构体**：适用于临时数据结构，减少类型定义的复杂性

---

20. Methods (方法)

https://gobyexample.com/methods

20.1 核心概念

Go 支持在结构体类型上定义方法。方法是一种特殊的函数，它有一个接收者参数，将函数与特定类型关联起来。这使得 Go 能够实现面向对象编程的某些特性，同时保持语言的简洁性。

生活化类比：如果说结构体 `rect`（矩形）是一个“名词”，定义了“是什么”（有宽度和高度），那么方法就是与这个名词搭配的“动词”，定义了它“能做什么”。`area()` 方法就是为“矩形”这类事物量身定做的专属技能，告诉它如何计算自己的面积。任何一个具体的矩形实例都可以调用这个技能。

20.2 方法的基本用法

20.2.1 定义结构体

```
type rect struct {
    width, height int
}
```

说明：首先定义一个矩形结构体，包含宽度和高度字段。

20.2.2 指针接收者方法

```
// 这个 area 方法有一个 *rect 类型的接收者
func (r *rect) area() int {
    return r.width * r.height
}
```

特点：

- 方法定义在 `func` 关键字后面加上接收者参数 `(r *rect)`
- `*rect` 表示指针接收者，方法可以修改接收者的值
- 接收者参数通常使用类型名的首字母作为变量名

20.2.3 值接收者方法

```
// 方法可以为指针或值接收者类型定义
// 这里是一个值接收者的例子
func (r rect) perim() int {
    return 2*r.width + 2*r.height
}
```

特点：

- `rect`（不带 `*`）表示值接收者
- 值接收者会复制结构体，不能修改原始值
- 适用于不需要修改接收者状态的方法

20.2.4 调用方法

```
r := rect{width: 10, height: 5}

// 调用为结构体定义的两个方法
fmt.Println("area: ", r.area())
fmt.Println("perim:", r.perim())
```

语法：使用点号 `.` 调用方法，语法与访问字段相同。

20.2.5 自动转换

```
// Go 自动处理方法调用时值和指针之间的转换
// 你可能希望使用指针接收者类型来避免在方法调用时复制，
// 或者允许方法修改接收结构体
rp := &r
fmt.Println("area: ", rp.area())
fmt.Println("perim:", rp.perim())
```

重要特性：

- Go 会自动在值和指针之间进行转换
- 值类型可以调用指针接收者方法（Go 会自动取地址）
- 指针类型可以调用值接收者方法（Go 会自动解引用）

20.3 完整代码示例

```
package main

import "fmt"

type rect struct {
    width, height int
}

// 这个 area 方法有一个 *rect 类型的接收者
func (r *rect) area() int {
    return r.width * r.height
}

// 方法可以为指针或值接收者类型定义
// 这里是一个值接收者的例子
func (r rect) perim() int {
    return 2*r.width + 2*r.height
}

func main() {
    r := rect{width: 10, height: 5}

    // 调用为结构体定义的两个方法
    fmt.Println("area: ", r.area())
    fmt.Println("perim:", r.perim())

    // Go 自动处理方法调用时值和指针之间的转换
    // 你可能希望使用指针接收者类型来避免在方法调用时复制，
    // 或者允许方法修改接收结构体
    rp := &r
    fmt.Println("area: ", rp.area())
    fmt.Println("perim:", rp.perim())
}
```

20.4 运行结果

```
$ go run methods.go
area:  50
perim: 30
area:  50
perim: 30
```

20.5 指针接收者 vs 值接收者

20.5.1 何时使用指针接收者

- 方法需要修改接收者的值
- 接收者是大型结构体，避免复制开销
- 保持一致性（如果某些方法使用指针接收者，其他方法也应该使用）

20.5.2 何时使用值接收者

- 接收者是小型结构体或基本类型
- 方法不需要修改接收者
- 接收者是不可变的值类型

20.5.3 自动转换规则

```
// 假设有以下方法定义：
// func (r *rect) area() int    // 指针接收者
// func (r rect) perim() int    // 值接收者

var r rect = rect{width: 10, height: 5}
var rp *rect = &r

// 以下调用都是有效的：
r.area()    // Go 自动转换为 (&r).area()
r.perim()   // 直接调用
rp.area()   // 直接调用
rp.perim()  // Go 自动转换为 (*rp).perim()
```

20.6 关键要点

1. **接收者语法**：方法通过在函数名前添加接收者参数来定义
2. **两种接收者类型**：指针接收者 `*T` 和值接收者 `T`
3. **自动转换**：Go 自动处理值和指针之间的转换，使方法调用更便利
4. **性能考虑**：指针接收者避免复制，值接收者保证不可变性
5. **修改能力**：只有指针接收者可以修改接收者的状态
6. **一致性原则**：同一类型的方法应该使用一致的接收者类型

7. **面向对象特性**：方法为 Go 提供了面向对象编程的基础
8. **接口准备**：方法是实现接口的基础，为后续学习接口做准备

---

21. Interfaces (接口)

https://gobyexample.com/interfaces

21.1 核心概念

接口是方法签名的命名集合。接口定义了一组方法，但不实现它们。任何类型只要实现了接口中的所有方法，就自动实现了该接口。这是 Go 语言实现多态性和抽象的主要机制。

生活化类比：接口就像一个“电源插座”的标准。`geometry` 接口定义了一个标准：“任何能被称为‘几何图形’的东西，都必须能计算面积（`area()`）和周长（`perim()`）”。它不关心你是圆（`circle`）还是方（`rect`），只要你的“插头”符合这个标准（实现了这两个方法），你就能插到 `measure` 函数这个“电源”上正常工作。这就是所谓的“鸭子类型”——如果它走起来像鸭子，叫起来也像鸭子，那它就是一只鸭子。

21.2 接口的基本用法

21.2.1 定义接口

```
// 这是一个几何图形的基本接口
type geometry interface {
    area() float64
    perim() float64
}
```

特点：

- 使用 `interface` 关键字定义接口
- 接口只包含方法签名，不包含实现
- 方法签名包括方法名、参数类型和返回类型

21.2.2 定义实现接口的类型

```
// 在我们的例子中，我们将在 rect 和 circle 类型上实现这个接口
type rect struct {
    width, height float64
}
type circle struct {
    radius float64
}
```

21.2.3 实现接口方法 - rect 类型

```go
// 要在 Go 中实现接口，我们只需要实现接口中的所有方法
// 这里我们在 rect 上实现 geometry 接口
func (r rect) area() float64 {
    return r.width * r.height
}
func (r rect) perim() float64 {
    return 2*r.width + 2*r.height
}
```

重要概念：

- Go 中的接口实现是隐式的
- 不需要显式声明"实现了某个接口"
- 只要实现了接口的所有方法，就自动实现了接口

21.2.4 实现接口方法 - circle 类型

```go
// circle 的实现
func (c circle) area() float64 {
    return math.Pi * c.radius * c.radius
}
func (c circle) perim() float64 {
    return 2 * math.Pi * c.radius
}
```

21.2.5 使用接口类型

```go
// 如果变量具有接口类型，那么我们可以调用命名接口中的方法
// 这里是一个通用的 measure 函数，利用这一点来处理任何 geometry
func measure(g geometry) {
    fmt.Println(g)
    fmt.Println(g.area())
    fmt.Println(g.perim())
}
```

关键特性：

- 函数可以接受接口类型作为参数
- 可以调用接口中定义的任何方法
- 实现了多态性：同一个函数可以处理不同的具体类型

21.2.6 类型断言

```
// 有时了解接口值的运行时类型很有用
// 一个选择是使用类型断言，如这里所示
// 另一个是类型 switch
func detectCircle(g geometry) {
    if c, ok := g.(circle); ok {
        fmt.Println("circle with radius", c.radius)
    }
}
```

类型断言语法：

- `g.(circle)` 尝试将接口 `g` 断言为 `circle` 类型
- 返回两个值：转换后的值和布尔值（表示断言是否成功）
- 如果断言失败，`ok` 为 `false`

21.2.7 使用接口

```
r := rect{width: 3, height: 4}
c := circle{radius: 5}

// circle 和 rect 结构体类型都实现了 geometry 接口
// 所以我们可以使用这些结构体的实例作为 measure 的参数
measure(r)
measure(c)

detectCircle(r)  // 不会输出任何内容，因为 r 不是 circle
detectCircle(c)  // 输出圆的半径信息
```

21.3 完整代码示例

```
package main

import (
    "fmt"
    "math"
)

// 这是一个几何图形的基本接口
type geometry interface {
    area() float64
    perim() float64
}

// 在我们的例子中，我们将在 rect 和 circle 类型上实现这个接口
type rect struct {
    width, height float64
}
type circle struct {
    radius float64
```

```
}

// 要在 Go 中实现接口，我们只需要实现接口中的所有方法
// 这里我们在 rect 上实现 geometry 接口
func (r rect) area() float64 {
    return r.width * r.height
}
func (r rect) perim() float64 {
    return 2*r.width + 2*r.height
}

// circle 的实现
func (c circle) area() float64 {
    return math.Pi * c.radius * c.radius
}
func (c circle) perim() float64 {
    return 2 * math.Pi * c.radius
}

// 如果变量具有接口类型，那么我们可以调用命名接口中的方法
// 这里是一个通用的 measure 函数，利用这一点来处理任何 geometry
func measure(g geometry) {
    fmt.Println(g)
    fmt.Println(g.area())
    fmt.Println(g.perim())
}

// 有时了解接口值的运行时类型很有用
// 一个选择是使用类型断言，如这里所示
// 另一个是类型 switch
func detectCircle(g geometry) {
    if c, ok := g.(circle); ok {
        fmt.Println("circle with radius", c.radius)
    }
}

func main() {
    r := rect{width: 3, height: 4}
    c := circle{radius: 5}

    // circle 和 rect 结构体类型都实现了 geometry 接口
    // 所以我们可以使用这些结构体的实例作为 measure 的参数
    measure(r)
    measure(c)

    detectCircle(r)
    detectCircle(c)
}
```

21.4 运行结果

```
$ go run interfaces.go
{3 4}
12
14
{5}
78.53981633974483
31.41592653589793
circle with radius 5
```

21.5 接口的高级特性

21.5.1 空接口

```
// 空接口可以存储任何类型的值
var i interface{}
i = 42
i = "hello"
i = []int{1, 2, 3}
```

21.5.2 接口组合

```
type Reader interface {
    Read([]byte) (int, error)
}

type Writer interface {
    Write([]byte) (int, error)
}

// 组合接口
type ReadWriter interface {
    Reader
    Writer
}
```

21.5.3 类型 Switch

```
func describe(i interface{}) {
    switch v := i.(type) {
    case int:
        fmt.Printf("整数: %d\n", v)
    case string:
        fmt.Printf("字符串: %s\n", v)
    case bool:
        fmt.Printf("布尔值: %t\n", v)
    default:
        fmt.Printf("未知类型: %T\n", v)
    }
}
```

21.6 关键要点

1. **隐式实现**：Go 中的接口实现是隐式的，无需显式声明
2. **多态性**：接口实现了多态，同一接口可以有多种实现
3. **类型断言**：可以使用类型断言检查接口的具体类型
4. **空接口**：`interface{}` 可以存储任何类型的值
5. **接口组合**：可以通过嵌入其他接口来组合接口
6. **鸭子类型**：如果它走起来像鸭子，叫起来像鸭子，那它就是鸭子
7. **设计原则**：接口应该小而专注，通常只包含少数几个方法
8. **解耦合**：接口帮助实现代码的解耦合和可测试性

---

22. Enums (枚举)

https://gobyexample.com/enums

22.1 核心概念

Go 没有内置的枚举类型，但可以使用语言习惯用法来实现枚举。主要通过自定义类型、常量和 `iota` 关键字来创建类型安全的枚举值。

生活化类比：枚举就像是交通信号灯的三种状态：红、黄、绿。这三种状态是预先定义好的、有限的集合。你不能发明一个“蓝色”的交通信号。在 Go 中，通过 `const` 和 `iota`，我们为 `ServerState` 这个“交通灯”定义了 `StateIdle`、`StateConnected` 等几个明确的状态，从而确保程序只会在这些合法的状态之间切换，避免了使用任意数字或字符串带来的混乱。

22.2 基本枚举实现

22.2.1 定义枚举类型

```
// 枚举类型通常基于 int 实现
type ServerState int

// 使用 iota 自动生成递增的常量值
const (
    StateIdle ServerState = iota // 0
    StateConnected               // 1
    StateError                   // 2
    StateRetrying                // 3
)
```

特点：

- 创建自定义类型来表示枚举
- `iota` 自动为每个常量分配递增的整数值
- 第一个值从 0 开始，后续值自动递增

22.2.2 枚举的字符串表示

```
// 创建映射表提供枚举值的字符串表示
var stateName = map[ServerState]string{
    StateIdle:      "idle",
    StateConnected: "connected",
    StateError:     "error",
    StateRetrying:  "retrying",
}

// 实现 String() 方法，使枚举值可以直接打印
func (ss ServerState) String() string {
    return stateName[ss]
}
```

重要特性：

- 实现 `fmt.Stringer` 接口，提供友好的字符串输出
- 使用 map 建立枚举值与字符串的对应关系
- 支持 `fmt.Println()` 直接打印枚举的字符串表示

22.2.3 枚举状态转换

```
// transition 模拟服务器状态的变化
func transition(s ServerState) ServerState {
    switch s {
    case StateIdle:
        return StateConnected
    case StateConnected, StateRetrying:
        // 假设这些状态可以转为 idle
        return StateIdle
    case StateError:
        return StateError
    default:
        panic(fmt.Errorf("unknown state: %s", s))
    }
}
```

用途：展示枚举在状态机或业务逻辑中的实际应用。

22.3 完整代码示例

```
package main

import "fmt"

// 枚举类型通常基于 int 实现
type ServerState int
```

```go
const (
    StateIdle ServerState = iota
    StateConnected
    StateError
    StateRetrying
)

// 通过实现 fmt.Stringer 接口，我们可以打印出枚举值的字符串名称
var stateName = map[ServerState]string{
    StateIdle:      "idle",
    StateConnected: "connected",
    StateError:     "error",
    StateRetrying:  "retrying",
}

func (ss ServerState) String() string {
    return stateName[ss]
}

// 如果我们有一个返回 int 的值，我们可以将其转换为 ServerState 类型
// 然后使用转换工作上的方法

func main() {
    ns := transition(StateIdle)
    fmt.Println(ns)

    ns2 := transition(ns)
    fmt.Println(ns2)
}

// transition 模拟服务器状态的变化
func transition(s ServerState) ServerState {
    switch s {
    case StateIdle:
        return StateConnected
    case StateConnected, StateRetrying:
        // 假设这些状态可以转为 idle
        return StateIdle
    case StateError:
        return StateError
    default:
        panic(fmt.Errorf("unknown state: %s", s))
    }
}
```

22.4 运行结果

```
$ go run enums.go
connected
idle
```

22.5 高级枚举模式

22.5.1 带值的枚举

```
// 为枚举指定特定的值
const (
    Sunday    Weekday = iota // 0
    Monday                   // 1
    Tuesday                  // 2
    Wednesday                // 3
    Thursday                 // 4
    Friday                   // 5
    Saturday                 // 6
)

// 或者指定特殊值
const (
    StatusOK       Status = 200
    StatusNotFound Status = 404
    StatusError    Status = 500
)
```

22.5.2 使用 stringer 工具

```
//go:generate stringer -type=Direction
type Direction int

const (
    North Direction = iota
    East
    South
    West
)
```

说明：`stringer` 工具可以自动生成 `String()` 方法。

22.5.3 基于字符串的枚举

```
type Color string

const (
    Red   Color = "red"
    Green Color = "green"
    Blue  Color = "blue"
)
```

22.6 关键要点

1. **无内置支持**：Go 没有原生枚举类型，需要使用语言习惯用法实现

2. **iota 关键字**：提供自动递增的常量值，简化枚举定义
3. **类型安全**：自定义类型提供编译时类型检查
4. **字符串表示**：实现 `String()` 方法提供友好的输出格式
5. **灵活性**：可以基于不同的底层类型（int、string 等）创建枚举
6. **工具支持**：`stringer` 工具可以自动生成字符串方法
7. **状态机应用**：枚举常用于状态机、错误码、配置选项等场景
8. **最佳实践**：枚举值应该有明确的语义，避免直接使用数字比较

---

23. Struct Embedding (结构体嵌入)

https://gobyexample.com/struct-embedding

23.1 核心概念

Go 支持结构体嵌入（struct embedding），这是一种用于组合类型的技术。它允许一个结构体类型包含另一个结构体类型的匿名字段，从而使得外部结构体可以直接访问内部结构体的字段和方法，实现类似“继承”的效果，但其本质是组合。

生活化类比：结构体嵌入就像是给一个基础款的手机（`base` 结构体）套上一个带附加功能的手机壳（`container` 结构体）。这个手机壳自己可能有一个挂绳孔（`str` 字段），但它把手机包了进去，所以你拿着这个套了壳的手机，不仅能用挂绳孔，还能直接用手机的摄像头和屏幕（`base` 的字段和方法被“提升”了）。你并没有改造手机本身，只是通过组合让它拥有了更多的功能。

23.2 结构体嵌入的基本用法

23.2.1 定义基础和容器结构体

首先，我们定义一个基础结构体 `base`，它包含一个字段和一个方法。

```
type base struct {
    num int
}

func (b base) describe() string {
    return fmt.Sprintf("base with num=%v", b.num)
}
```

然后，我们定义一个 `container` 结构体，它嵌入了 `base` 结构体。嵌入时，我们只提供类型名，不提供字段名。

```
type container struct {
    base
    str string
}
```

23.2.2 初始化和访问嵌入字段

创建 `container` 实例时，我们可以像普通字段一样初始化嵌入的 `base` 结构体。

```
co := container{
    base: base{
        num: 1,
    },
    str: "some name",
}
```

由于字段提升（field promotion），我们可以直接通过 `container` 实例访问 `base` 的字段 `num`。

```
fmt.Printf("co={num: %v, str: %v}\n", co.num, co.str)
```

当然，也可以通过完整的路径访问嵌入的字段。

```
fmt.Println("also num:", co.base.num)
```

23.2.3 方法提升和接口实现

`base` 的方法 `describe` 也被提升到了 `container`，因此我们可以直接在 `container` 实例上调用它。

```
fmt.Println("describe:", co.describe())
```

由于 `container` 拥有 `describe` 方法，它也隐式地实现了包含该方法的接口。

```
type describer interface {
    describe() string
}

var d describer = co
fmt.Println("describer:", d.describe())
```

23.3 完整代码示例

```
package main

import "fmt"

// 定义一个基础结构体
type base struct {
    num int
}

// 定义 base 的方法
func (b base) describe() string {
    return fmt.Sprintf("base with num=%v", b.num)
}
```

```go
// 定义一个嵌入了 base 的 container 结构体
type container struct {
    base
    str string
}

func main() {
    // 初始化 container
    co := container{
        base: base{
            num: 1,
        },
        str: "some name",
    }

    // 我们可以直接访问嵌入结构体的字段
    fmt.Printf("co={num: %v, str: %v}\n", co.num, co.str)

    // 也可以通过嵌入的类型名访问
    fmt.Println("also num:", co.base.num)

    // 嵌入结构体的方法也被提升到了容器结构体
    fmt.Println("describe:", co.describe())

    // 容器结构体实现了 describer 接口，因为嵌入的 base 实现了它
    type describer interface {
        describe() string
    }

    var d describer = co
    fmt.Println("describer:", d.describe())
}
```

23.4 运行结果

```
$ go run struct-embedding.go
co={num: 1, str: some name}
also num: 1
describe: base with num=1
describer: base with num=1
```

23.5 关键要点

1. **组合优于继承**：结构体嵌入是 Go 实现代码复用的主要方式，它遵循“组合优于继承”的设计哲学。
2. **字段和方法提升**：嵌入结构体的字段和方法会被“提升”到外部结构体，可以直接访问，简化了代码。
3. **隐式接口实现**：如果嵌入的结构体实现了某个接口，那么外部结构体也自动实现了该接口。
4. **无名称冲突**：当外部结构体和嵌入结构体有同名字段或方法时，外部结构体的成员优先。可以通过显式路径（如 `co.base.num`）访问被遮蔽的嵌入成员。

5. **初始化语法**：在初始化时，嵌入的结构体类型名作为字段名进行初始化。

---

24. Generics (泛型)

https://gobyexample.com/generics

24.1 核心概念

Go 1.18 版本引入了泛型，它允许我们编写不指定具体类型的函数或数据结构，从而大大提高了代码的复用性和灵活性，同时保持了 Go 语言的类型安全特性。

生活化类比：泛型就像一个“万能模板”。想象一下，你想设计一个能装任何尺寸水果的果篮（泛型函数或数据结构），而不是为苹果、香蕉、西瓜分别设计三种不同的果篮。你只需要设计一个“果篮模板”，并告诉使用者“这里可以放任何水果”（类型参数），这样同一个果篮设计就能适用于所有水果，既省事又灵活。

24.2 泛型函数与类型约束

24.2.1 定义一个简单的泛型函数

让我们从一个例子开始：一个函数，它可以在一个切片里查找某个元素的位置。这个函数应该能处理任何类型的切片，只要这个类型的元素是“可比较的”（比如可以用 `==` 来判断是否相等）。

```
// SlicesIndex 是一个泛型函数
// [S ~[]E, E comparable] 定义了类型参数和约束
// S 是一个切片类型，其元素类型是 E
// E 必须是 comparable (可比较的)
func SlicesIndex[S ~[]E, E comparable](s S, v E) int {
    for i, vs := range s {
        if v == vs {
            return i
        }
    }
    return -1
}
```

- `[S ~[]E, E comparable]` 这部分就是类型参数声明。
- `S ~[]E`：`S` 是一个类型参数，它必须是一个切片类型，我们用 `E` 来代表切片里元素的类型。`~` 符号表示 `S` 的底层类型是 `[]E`。
- `E comparable`：`E` 是另一个类型参数，代表切片元素的类型。`comparable` 是一个内置的约束，它要求 `E` 必须是支持 `==` 和 `!=` 比较的类型（如 `int`, `string`, `struct` 等）。

24.2.2 使用泛型函数

现在我们可以用这个函数来处理不同类型的切片了。

```go
// 用 int 切片测试
xi := []int{10, 20, 15, -10}
// 在 xi 中查找 15，应该返回其索引 2
fmt.Println(SlicesIndex(xi, 15))

// 用 string 切片测试
xs := []string{"foo", "bar", "baz"}
// 在 xs 中查找 "hello"，切片中不存在，应该返回 -1
fmt.Println(SlicesIndex(xs, "hello"))
```

类型推断：注意，在调用 `SlicesIndex` 时，我们没有写 `SlicesIndex[[]int, int]` 这样的东西。Go 编译器足够聪明，能根据我们传入的参数（`xi` 是 `[]int`，`15` 是 `int`）自动推断出 `S` 是 `[]int`，`E` 是 `int`。这就是类型推断。

24.3 泛型类型

除了泛型函数，我们还可以定义泛型类型。比如，我们可以定义一个可以存放任何类型数据的链表。

24.3.1 定义一个泛型链表

```go
// List 是一个泛型类型，它可以存放任何类型的元素
// [T any] 表示类型参数 T 可以是任何类型 (any 是 interface{} 的别名)
type List[T any] struct {
    head, tail *element[T]
}

// element 是链表中的节点，也是一个泛型类型
type element[T any] struct {
    next *element[T]
    val  T
}
```

- **`List[T any]`**：`List` 是一个泛型结构体，`T` 是类型参数，`any` 是一个内置约束，表示 `T` 可以是任何类型。
- **`element[T any]`**：链表的节点 `element` 也需要是泛型的，因为它存储的值 `val` 的类型是 `T`。

24.3.2 在泛型类型上定义方法

我们可以在泛型类型上定义方法。这些方法可以使用类型参数。

```go
// Push 方法在链表尾部添加一个元素
func (lst *List[T]) Push(v T) {
    if lst.tail == nil {
        lst.head = &element[T]{val: v}
        lst.tail = lst.head
    } else {
        lst.tail.next = &element[T]{val: v}
        lst.tail = lst.tail.next
    }
}
```

```
// GetAll 方法返回链表中所有元素的切片
func (lst *List[T]) GetAll() []T {
    var elems []T
    for e := lst.head; e != nil; e = e.next {
        elems = append(elems, e.val)
    }
    return elems
}
```

- 方法的接收者是 `*List[T]`，方法内部可以使用类型参数 `T`。

24.4 完整代码示例

```
package main

import "fmt"

// SlicesIndex 是一个泛型函数，用于在切片中查找元素
func SlicesIndex[S ~[]E, E comparable](s S, v E) int {
    for i, vs := range s {
        if v == vs {
            return i
        }
    }
    return -1
}

// List 是一个泛型单向链表
type List[T any] struct {
    head, tail *element[T]
}

// element 是链表节点
type element[T any] struct {
    next *element[T]
    val  T
}

// Push 方法向链表尾部添加元素
func (lst *List[T]) Push(v T) {
    if lst.tail == nil {
        lst.head = &element[T]{val: v}
        lst.tail = lst.head
    } else {
        lst.tail.next = &element[T]{val: v}
        lst.tail = lst.tail.next
    }
}

// GetAll 方法返回链表中所有元素的切片
func (lst *List[T]) GetAll() []T {
```

```
    var elems []T
    for e := lst.head; e != nil; e = e.next {
        elems = append(elems, e.val)
    }
    return elems
}

func main() {
    // --- 测试泛型函数 SlicesIndex ---
    xi := []int{10, 20, 15, -10}
    // 在整数切片中查找 15，它在索引 2 的位置
    fmt.Println(SlicesIndex(xi, 15))

    xs := []string{"foo", "bar", "baz"}
    // 在字符串切片中查找 "hello"，它不存在，所以返回 -1
    fmt.Println(SlicesIndex(xs, "hello"))

    // --- 测试泛型类型 List ---
    // 实例化一个存放 int 的链表
    var lst List[int]
    lst.Push(10)
    lst.Push(20)
    lst.Push(30)
    // 获取并打印链表中的所有元素
    fmt.Println("list:", lst.GetAll())
}
```

24.5 运行结果

```
$ go run generics.go
2       # 15 在整数切片中的索引是 2
-1      # "hello" 在字符串切片中不存在，返回 -1
list: [10 20 30] # 打印泛型链表中的所有元素
```

24.6 关键要点

1. **代码复用**：泛型让你能编写一次代码，然后用于多种不同的数据类型，避免了重复。
2. **类型安全**：与使用 `interface{}` 的“伪泛型”不同，Go 的泛型是类型安全的。编译器会在编译时检查类型，防止类型错误。
3. **类型参数**：使用方括号 `[]` 来声明类型参数，如 `[T any]`。
4. **类型约束**：约束定义了类型参数必须满足的条件（比如必须是可比较的）。`comparable` 和 `any` 是两个重要的内置约束。你也可以自定义约束。
5. **类型推断**：在调用泛型函数时，编译器通常能自动推断出类型参数，让代码更简洁。
6. **泛型类型**：不仅函数可以是泛型的，`struct` 和 `interface` 等类型也可以是泛型的。

---

25. Range over Iterators (遍历迭代器)

https://gobyexample.com/range-over-iterators

25.1 核心概念

从 Go 1.22 开始，`range` 表达式可以作用于实现了迭代器协议的任何类型。这个特性极大地增强了 `range` 的灵活性，使其不再局限于内置的集合类型。现在，我们可以为自定义的集合（如链表、树等）定义迭代器，并使用 `for...range` 循环来优雅地遍历它们。

生活化类比：`range` 遍历迭代器就像是订阅一份杂志。你不需要一次性拿到所有的期刊。相反，出版社（迭代器）每次只寄给你最新的一期（通过 `yield` “生产”数据），你收到后就阅读（`for range` “消费”数据）。如果你想中途停止订阅（`break` 循环），出版社就会收到通知，停止寄送后续的杂志，非常智能。

25.2 迭代器协议

25.2.1 `iter.Seq` 类型

迭代器协议的核心是 `iter.Seq[V]` 和 `iter.Seq2[K, V]` 这两个泛型类型，它们定义在新的 `iter` 包中。

- `iter.Seq[V]`：用于表示一个产生单个值的迭代器。
- `iter.Seq2[K, V]`：用于表示一个产生键值对的迭代器。

这两个类型实际上都是函数类型。一个迭代器就是一个函数，它接受一个 `yield` 函数作为参数。`yield` 函数负责“生产”数据，`range` 循环则“消费”这些数据。

25.2.2 编写一个自定义迭代器

让我们为一个自定义的链表类型 `List[T]` 创建一个迭代器。

```
// All 方法返回一个 iter.Seq[T] 类型的迭代器
func (lst *List[T]) All() iter.Seq[T] {
    // 返回一个函数，这是迭代器的主体
    return func(yield func(T) bool) {
        // 遍历链表
        for e := lst.head; e != nil; e = e.next {
            // 调用 yield 函数来“产生”值
            // 如果 yield 返回 false，则停止迭代
            if !yield(e.val) {
                return
            }
        }
    }
}
```

- `All` 方法返回一个函数，该函数的签名符合 `iter.Seq[T]` 的要求。
- 在这个返回的函数内部，我们遍历链表的每个元素。
- `yield(e.val)` 这行代码是关键：它将链表中的值 `e.val` 发送给 `range` 循环。
- `yield` 函数的返回值是一个布尔值，用于控制迭代是否继续。如果 `range` 循环因为 `break` 或 `return` 而提前退出，`yield` 就会返回 `false`，从而使迭代器函数也能够优雅地终止。

25.3 使用 `range` 遍历迭代器

25.3.1 遍历自定义链表

有了 `All` 方法，我们现在可以直接在 `List` 上使用 `range` 循环。

```
var lst List[int]
lst.Push(1)
lst.Push(2)
lst.Push(3)

// 直接在 lst.All() 返回的迭代器上进行 range 循环
for v := range lst.All() {
    fmt.Println(v)
}
```

25.3.2 遍历一个无限序列

迭代器也可以用来表示一个无限的序列，比如斐波那契数列。

```
// Fibonacci 函数返回一个无限的斐波那契数列迭代器
func Fibonacci() iter.Seq[int] {
    return func(yield func(int) bool) {
        a, b := 0, 1
        // 只要 yield 返回 true，就一直产生下一个数
        for yield(a) {
            a, b = b, a+b
        }
    }
}

// 在 range 循环中使用 break 来控制迭代次数
for v := range Fibonacci() {
    if v > 50 {
        break // 当数值大于50时停止
    }
    fmt.Println(v)
}
```

25.4 迭代器辅助函数

`iter` 包和 `slices` 包提供了一些方便的辅助函数来处理迭代器。

25.4.1 `iter.Pull` 和 `iter.Take`

- `iter.Pull`：将一个“推”模式的迭代器（由 `range` 使用）转换成一个“拉”模式的迭代器。你可以手动调用它来获取下一个值。
- `iter.Take`：创建一个新的迭代器，它只从原始迭代器中获取前 N 个元素。

```
// “拉”模式的例子
pull, stop := iter.Pull(Fibonacci())
defer stop() // 确保资源被释放
for i := 0; i < 10; i++ {
    v, ok := pull()
    if !ok {
        break
    }
    fmt.Println(v)
}
```

25.4.2 `slices.Collect`

`slices.Collect` 函数可以从一个迭代器中收集所有的值，并将它们存入一个切片。

```
// 从 Fibonacci 迭代器中获取前10个元素，并收集到一个切片中
fib10 := slices.Collect(iter.Take(Fibonacci(), 10))
fmt.Println(fib10) // 输出: [0 1 1 2 3 5 8 13 21 34]
```

25.5 完整代码示例

```
package main

import (
  "fmt"
  "iter"
  "slices"
)

// A List is a linked list of values of type T.
type List[T any] struct {
  head, tail *element[T]
}

// An element is a node in a linked list.
type element[T any] struct {
  next *element[T]
  val  T
}

// Push adds the given value to the end of the list.
func (lst *List[T]) Push(v T) {
  if lst.tail == nil {
    lst.head = &element[T]{val: v}
    lst.tail = lst.head
  } else {
    lst.tail.next = &element[T]{val: v}
    lst.tail = lst.tail.next
  }
}
```

```
// All returns an iterator over the list values.
func (lst *List[T]) All() iter.Seq[T] {
  return func(yield func(T) bool) {
    for e := lst.head; e != nil; e = e.next {
      if !yield(e.val) {
        return
      }
    }
  }
}

// Fibonacci returns an iterator that yields Fibonacci numbers.
func Fibonacci() iter.Seq[int] {
  return func(yield func(int) bool) {
    a, b := 0, 1
    for yield(a) {
      a, b = b, a+b
    }
  }
}

func main() {
  // Ranging over a list.
  var lst List[int]
  lst.Push(1)
  lst.Push(2)
  lst.Push(3)
  for v := range lst.All() {
    fmt.Println(v)
  }

  // Ranging over the first 10 Fibonacci numbers.
  for v := range Fibonacci() {
    if v > 50 {
      break
    }
    fmt.Println(v)
  }

  // We can also use a pull-style iterator by passing a
  // function that always returns true to the iterator,
  // and then calling the resulting function.
  pull, stop := iter.Pull(Fibonacci())
  defer stop()
  for i := 0; i < 10; i++ {
    v, ok := pull()
    if !ok {
      break
    }
    fmt.Println(v)
  }
```

```
  // The slices package has various functions for working
  // with iterators, like Collect.
  fib10 := slices.Collect(iter.Take(Fibonacci(), 10))
  fmt.Println(fib10)
}
```

25.6 运行结果

```
$ go run range-over-iterators.go
1
2
3
0
1
1
2
3
5
8
13
21
34
0
1
1
2
3
5
8
13
21
34
[0 1 1 2 3 5 8 13 21 34]
```

25.7 关键要点

1. **统一的遍历方式**：`range` 现在可以用于任何实现了迭代器协议的自定义类型，提供了一种统一的遍历语法。
2. **`iter` 包**：新的 `iter` 包定义了迭代器协议的核心类型 `iter.Seq[V]` 和 `iter.Seq2[K, V]`。
3. **基于函数的设计**：Go 的迭代器是基于函数的，一个迭代器就是一个接受 `yield` 回调的函数。
4. **优雅的终止**：`yield` 函数的布尔返回值使得 `range` 循环可以与迭代器协作，在循环提前退出时，迭代器也能被正确地终止。
5. **无限序列**：迭代器可以轻松地表示无限序列，并通过 `range` 中的 `break` 来控制遍历的范围。
6. **辅助工具**：`iter` 和 `slices` 包提供了如 `Pull`、`Take`、`Collect` 等实用工具，方便对迭代器进行转换和操作。

26. Errors (错误处理)

https://gobyexample.com/errors

26.1 核心概念

在 Go 中，错误处理遵循“返回值 + `error`”的模式：函数把潜在错误作为最后一个返回值暴露给调用者，从而让错误处理逻辑与业务分支保持紧密耦合。`error` 本质是一个只有 `Error() string` 方法的接口，`nil` 表示成功，非 `nil` 则代表需要处理的问题。

生活化类比：Go 的错误处理就像一个严谨的工匠在完成每一步操作后都立即检查工作质量。每切一块木头（调用一个函数），工匠都会马上用尺子量一下（`if err != nil`）。如果尺寸不对，他会立刻停下来处理这个问题，而不是继续用错误的零件建造，导致最后整个作品（程序）都无法使用。

> 注：初学者常忘记判断 `err`。一个简单的习惯是：只要函数返回 `error`，就立即写出 `if err != nil { ... }`，哪怕稍后再补充体内逻辑，避免漏判。

26.2 创建基础错误

26.2.1 `errors.New` 的直观用法

`errors.New` 返回一个带固定消息的错误值，适合表达“遇到这个条件时就失败”。

```
func f(arg int) (int, error) {
    if arg == 42 {
        return -1, errors.New("can't work with 42")
    }
    return arg + 3, nil
}
```

这里的 `f` 函数在遇到特殊输入 42 时提前返回，调用者在收到 `err != nil` 时就知道执行失败。换成现实例子，我们也会在“用户输入为空”或“库存不足”时立即返回错误给前端。

26.2.2 `fmt.Errorf` + 占位符

当需要在错误消息里携带上下文时，用 `fmt.Errorf` 更灵活。

```
var ErrNoProfile = errors.New("profile not found")

func loadProfile(id int) error {
    return fmt.Errorf("load profile %d: %w", id, ErrNoProfile)
}
```

> 小例子：加载用户资料失败时，直接写 "load failed" 很难排查；把用户 ID 拼进错误文本可快速定位问题，同时利用 `%w` 继续保留底层原始错误，方便后续解包。

26.3 哨兵错误与错误包装

26.3.1 哨兵错误（sentinel error）

哨兵错误通常是预声明的全局变量，用来描述某种固定的失败状态。

```
var ErrOutOfTea = fmt.Errorf("no more tea available")
var ErrPower = fmt.Errorf("can't boil water")
```

这两个变量相当于“状态码”，后续任何地方只要返回它们，调用者就知道是缺茶还是停电。

26.3.2 错误包装与 `%w`

`fmt.Errorf("...%w", err)` 可以把下层错误包裹起来，形成错误链。示例中的 `makeTea` 在遇到停电时返回 `fmt.Errorf("making tea: %w", ErrPower)`，既保留高层语境，又能透出底层原因。

> 注：如果只是想拼接字符串，请用 `%v`；只有当你希望后续 `errors.Is` / `errors.As` 能识别原始错误时才使用 `%w`。

26.4 错误处理模式

26.4.1 基础判错流程

使用 `if err != nil` 是 Go 里最常见的模式，使得错误处理显式而可预期。

```
if r, err := f(i); err != nil {
    fmt.Println("f failed:", err)
} else {
    fmt.Println("f worked:", r)
}
```

这里在循环里调用 `f`，每次都立即判断错误。习惯这样写可以避免“错误悄悄冒泡不处理”的隐患。

26.4.2 `errors.Is` 判别哨兵错误

`errors.Is` 会沿着错误链查找目标哨兵错误，非常适合与 `%w` 搭配：

```
if err := makeTea(i); err != nil {
    if errors.Is(err, ErrOutOfTea) {
        fmt.Println("We should buy new tea!")
    } else if errors.Is(err, ErrPower) {
        fmt.Println("Now it is dark.")
    } else {
        fmt.Printf("unknown error: %s", err)
    }
    continue
}
fmt.Println("Tea is ready!")
```

> 小贴士：与其比较字符串，不如比较哨兵错误；字符串一改动或者本地化就全线崩溃，而哨兵错误始终是同一个变量。

26.5 完整代码示例

```
package main
```

```
import (
    "errors"
    "fmt"
)

func f(arg int) (int, error) {
    if arg == 42 {
        return -1, errors.New("can't work with 42")
    }
    return arg + 3, nil
}

var ErrOutOfTea = fmt.Errorf("no more tea available")
var ErrPower = fmt.Errorf("can't boil water")

func makeTea(arg int) error {
    if arg == 2 {
        return ErrOutOfTea
    } else if arg == 4 {
        return fmt.Errorf("making tea: %w", ErrPower)
    }
    return nil
}

func main() {
    for _, i := range []int{7, 42} {
        if r, err := f(i); err != nil {
            fmt.Println("f failed:", err)
        } else {
            fmt.Println("f worked:", r)
        }
    }

    for i := range 5 {
        if err := makeTea(i); err != nil {
            if errors.Is(err, ErrOutOfTea) {
                fmt.Println("We should buy new tea!")
            } else if errors.Is(err, ErrPower) {
                fmt.Println("Now it is dark.")
            } else {
                fmt.Printf("unknown error: %s
", err)
            }
            continue
        }
        fmt.Println("Tea is ready!")
    }
}
```

26.6 运行结果

```
$ go run errors.go
f worked: 10
f failed: can't work with 42
Tea is ready!
Tea is ready!
We should buy new tea!
Tea is ready!
Now it is dark.
```

26.7 关键要点

1. **错误是普通值**：遵循“最后一个返回值是 `error`”，让调用者一眼看出需要处理错误。
2. **哨兵错误 + `%w`**：用全局变量代表特定错误，并通过 `%w` 保留链路，后续可用 `errors.Is` 精准判定。
3. **即时检查**：在产生错误的附近立即处理或返回，避免错误状态在调用栈中迷失。
4. **错误消息要有上下文**：适当加入参数信息、调用阶段，排查时更高效。
5. **别忘了 happy path**：错误分支 `continue` 后，记得保留成功路径的输出，确保逻辑完整。

---

27. Custom Errors (自定义错误)

https://gobyexample.com/custom-errors

27.1 核心概念

自定义错误允许我们把额外的上下文信息绑定在错误值上，只要类型实现了 `Error() string` 方法，就能被当作 `error` 使用。这种方式非常适合表达“参数错在哪”“缺什么字段”等细节，相较于纯字符串错误更易观测和调试。

生活化类比：如果说普通错误 `errors.New("failed")` 像是一张内容只有“不合格”三个字的质检报告，那么自定义错误就像是一份详细的质检单。它不仅告诉你“不合格”，还会清楚地标明“哪个零件（`field`）不合格”、“违反了哪条生产标准（`rule`）”、“具体数值是多少（`value`）”。这份详细的报告让返工和修复变得有据可依，而不是一头雾水。

> 注：社区约定自定义错误类型命名以 `...Error` 结尾；看到这样的类型，读代码的人就能立刻意识到它实现了 `error` 接口。

27.2 定义自定义错误类型

27.2.1 结构体承载上下文

```
type argError struct {
    arg     int
    message string
}
```

这个结构体额外保存了触发错误的参数值 `arg`，以及配套的解释文案 `message`，能够给调用方更多排查信息。

27.2.2 实现 `Error` 方法

```go
func (e *argError) Error() string {
    return fmt.Sprintf("%d - %s", e.arg, e.message)
}
```

使用指针接收者可以避免在复制结构体时丢失状态，也让 `errors.As` 这种需要获取指针的函数更容易匹配。

27.3 在函数中返回自定义错误

```go
func f(arg int) (int, error) {
    if arg == 42 {
        return -1, &argError{arg: arg, message: "can't work with it"}
    }
    return arg + 3, nil
}
```

当参数等于 42 时，直接构造 `argError` 返回；否则返回计算结果。自定义错误值和普通错误一样遵循“成功返回 `nil`”的惯例。

> 小例子：如果你在注册流程里校验密码强度，可在错误里塞进 `rule: "length"` 或 `rule: "upper-case"` 字段，前端就能定制化提示“请至少输入 8 个字符”之类的消息。

27.4 使用 `errors.As` 读取字段

```go
if _, err := f(42); err != nil {
    var ae *argError
    if errors.As(err, &ae) {
        fmt.Println("参数:", ae.arg)
        fmt.Println("原因:", ae.message)
    }
}
```

`errors.As` 会沿着错误链查找第一个满足目标类型的错误，并把它解包到 `ae` 中，让我们方便地访问额外字段。

> 小贴士：相比直接使用类型断言（`err.(*argError)`），`errors.As` 能处理被 `%w` 包装过的错误，健壮性更好。

27.5 业务场景快速示例

```go
// 需要 import "strings"
type validationError struct {
    field string
    rule  string
}

func (e *validationError) Error() string {
    return fmt.Sprintf("field %q violates %s", e.field, e.rule)
}

func checkEmail(email string) error {
```

```
    if !strings.Contains(email, "@") {
        return &validationError{field: "email", rule: "must contain @"}
    }
    return nil
}
```

这个示例展示了如何把字段名与失败规则封装在错误里，调用方能据此决定是提示用户、落日志还是重试。

27.6 完整代码示例

```
package main

import (
    "errors"
    "fmt"
)

type argError struct {
    arg     int
    message string
}

func (e *argError) Error() string {
    return fmt.Sprintf("%d - %s", e.arg, e.message)
}

func f(arg int) (int, error) {
    if arg == 42 {
        return -1, &argError{arg: arg, message: "can't work with it"}
    }
    return arg + 3, nil
}

func main() {
    _, err := f(42)
    var ae *argError
    if errors.As(err, &ae) {
        fmt.Println(ae.arg)
        fmt.Println(ae.message)
    } else {
        fmt.Println("err doesn't match argError")
    }
}
```

27.7 运行结果

```
$ go run custom-errors.go
42
can't work with it
```

27.8 关键要点

1. **实现 `Error()` 即可满足接口**：只要类型提供 `Error() string` 方法，就能作为 `error` 被返回和比较。
2. **结构体存上下文**：把触发错误的参数、字段或阶段保存下来，后续排查更高效。
3. **指针接收者更通用**：让 `errors.As`、`errors.Is` 等函数工作顺畅，还能避免复制成本。
4. **与包装链配合**：即便外层用 `%w` 包裹，自定义错误依然能被匹配，形成完备的错误链。
5. **示例驱动落地**：随手写个小类型，就能把业务中的“为什么失败”说清楚，比通用字符串更友好。

---

28. Goroutines (协程)

https://gobyexample.com/goroutines

28.1 核心概念

`goroutine` 是 Go 运行时调度的轻量级线程。你可以用非常低的成本同时运行成百上千个 `goroutine`，他们共享同一地址空间，并通过通道（channel）或其他同步原语交换数据。

生活化类比：`goroutine` 就像是雇佣了一位超级高效且独立的助理。你交给他一项任务（`go myFunc()`），他就会立刻开始独立工作，而你（主程序）无需等待，可以继续处理自己的事情。你可以轻易地雇佣成千上万这样的助理（启动大量 goroutine），他们都在同一个办公室里（共享内存地址空间），可以高效地协同工作。

> 注：`goroutine` 由 Go 调度器在内核线程上运行，不等于 OS 线程；创建它的开销接近函数调用，非常适合构建并发服务。

28.2 启动一个 goroutine

28.2.1 同步调用

先看一个普通的同步函数调用：

```
func f(from string) {
    for i := range 3 {
        fmt.Println(from, ":", i)
    }
}

f("direct")
```

调用会阻塞当前 goroutine（也就是 `main`）直到循环结束。

28.2.2 使用 `go` 关键字

在函数前加上 `go` 关键字，就会把调用放到一个新的 goroutine 中并立即返回：

```
go f("goroutine")
```

此时 `main` goroutine 会继续执行后续语句，而 `f("goroutine")` 在后台并发运行。

28.3 匿名函数也能并发执行

你可以把匿名函数直接包装成 goroutine，并给它传参：

```
go func(msg string) {
    fmt.Println(msg)
}("going")
```

> 小例子：在 Web 服务里，收到请求后可以用匿名 goroutine 做异步日志或者缓存刷新，避免阻塞主流程。

28.4 等待 goroutine 结束

示例里用 `time.Sleep` 人为等待一秒，确保后台 goroutine 有机会输出：

```
time.Sleep(time.Second)
fmt.Println("done")
```

这是为了教学演示，真实项目里应使用 `sync.WaitGroup` 或 channel 来同步完成事件。

28.5 完整代码示例

```
package main

import (
    "fmt"
    "time"
)

func f(from string) {
    for i := range 3 {
        fmt.Println(from, ":", i)
    }
}

func main() {
    f("direct")

    go f("goroutine")

    go func(msg string) {
        fmt.Println(msg)
    }("going")

    time.Sleep(time.Second)
    fmt.Println("done")
}
```

28.6 运行结果

```
$ go run goroutines.go
direct : 0
direct : 1
direct : 2
goroutine : 0
going
goroutine : 1
goroutine : 2
done
```

输出顺序中，`direct` 的内容先打印出来，因为它是同步调用。两个异步 `goroutine` 的输出则可能交错，体现了并发调度。

28.7 关键要点

1. **`go` 关键字**：在函数调用前加 `go` 即可启动并发执行的 goroutine。
2. **轻量线程**：goroutine 由 Go 调度器管理，成本远低于 OS 线程。
3. **匿名函数休想逃**：将闭包用 `go` 启动是处理异步任务的常见技巧。
4. **同步手段**：示例用 `Sleep`，正式代码应使用 `WaitGroup` 或 channel 确认 goroutine 完成。
5. **输出顺序不确定**：并发执行意味着结果可能交叉出现，写日志或测试时要做好准备。

---

29. Channels (通道)

https://gobyexample.com/channels

29.1 核心概念

通道（channel）是 goroutine 之间通信的管道。写入端使用 `ch <- val` 发送数据，读取端使用 `<-ch` 接收数据。无缓冲通道的发送和接收都会阻塞，直到对端准备就绪，这个特性天然提供了同步机制。

生活化类比：通道就像是两个工人（goroutine）之间的一条神奇的传送带。一个工人把零件放到传送带上（`ch <- item`），传送带就会停下来，直到另一个工人从另一端把零件取走（`<-ch`）。这种机制确保了零件一次只传递一个，并且完美地同步了两个工人的工作节奏，让他们既能协作又不会互相干扰。

> 注：因为读写会阻塞，所以只要主 goroutine 在 `<-ch` 上等待，程序就不会提前退出，避免了“为什么没打印”这类问题。

29.2 创建与发送

29.2.1 `make` 创建无缓冲通道

```
messages := make(chan string) // 无缓冲通道，发送时会等待接收者
```

29.2.2 在 goroutine 中发送

```
go func() {
    messages <- "ping" // 向通道发送数据，直到主 goroutine 接收才会返回
}()
```

> 小例子：后台任务完成后把状态写入通道，主流程阻塞等待即可，无需手动 `Sleep`。

29.3 接收并保持同步

```
msg := <-messages // 阻塞等待数据，确保发送 goroutine 已经运行到这一步
fmt.Println(msg)
```

因为接收操作会阻塞，`main` 不会提前结束。只有当后台 goroutine 把数据放入通道，接收才会继续往下执行。

29.4 完整代码示例（含注释）

```
package main

import "fmt"

func main() {
    messages := make(chan string) // 建立一个 string 类型的无缓冲通道

    go func() {
        messages <- "ping" // 发送方：写入后阻塞，直到主 goroutine 取走
    }()

    msg := <-messages // 接收方：阻塞等待，从而保证 goroutine 的输出不会丢
    fmt.Println(msg)  // 输出：ping
}
```

29.5 运行结果

```
$ go run channels.go
ping
```

29.6 关键要点

1. **双向通信**：使用 `make(chan T)` 创建通道，让 goroutine 之间安全传值。
2. **阻塞特性**：无缓冲通道在发送或接收时都会等待对端，天然实现同步。
3. **避免 `Sleep`**：通过 `<-ch` 等待即可保证 goroutine 运行，不必像上节那样依赖 `time.Sleep`。
4. **匿名 goroutine 发送**：把发送逻辑放进匿名函数，写法简洁常见。
5. **主 goroutine 执行顺序**：只有在收到数据后才会继续执行，从而防止程序提前退出。

---

30. Channel Buffering (通道缓冲)

https://gobyexample.com/channel-buffering

30.1 核心概念

带缓冲的通道允许发送方在没有立即接收者的情况下临时存放一定数量的消息。与无缓冲通道相比，它们在生产者-消费者速度不一致时能起到“缓冲区”的作用。

生活化类比：带缓冲的通道就像是在传送带中间加了一个小货架。生产零件的工人（发送方）可以先把几个零件放在货架上，即使另一端的工人（接收方）暂时没空来取。这让生产工人可以不用停下来等待，从而平滑了两人工作速度上的微小差异。当然，如果货架满了，传送带还是会停下来。

> 注：缓冲并不意味着无限排队，一旦缓冲区满了，发送操作依然会阻塞，直到有接收者消费掉消息。

30.2 创建带缓冲通道

```
messages := make(chan string, 2) // 缓冲容量为 2，最多存 2 条消息
```

第二个参数就是缓冲大小，内部会维护一个循环队列来存放暂未被消费的元素。

30.3 写入与读取

```
messages <- "buffered" // 第一条写入后，缓冲区剩余 1 个空位
messages <- "channel"  // 第二条写入后，缓冲区已满

fmt.Println(<-messages) // 取出 "buffered"，此时缓冲区腾出 1 个位置
fmt.Println(<-messages) // 再取出 "channel"
```

> 小例子：爬虫抓取网页时，可先把 URL 放进缓冲通道，工人 goroutine 再按节奏消费，避免因网络抖动导致的停顿。

30.4 完整代码示例（含注释）

```
package main

import "fmt"

func main() {
    messages := make(chan string, 2) // 创建一个容量为 2 的缓冲通道

    messages <- "buffered" // 发送时不需要立刻有接收者
    messages <- "channel"  // 再发送一条，缓冲区现在已满

    fmt.Println(<-messages) // 接收第一条消息
    fmt.Println(<-messages) // 接收第二条消息
}
```

30.5 运行结果

```
$ go run channel-buffering.go
buffered
channel
```

30.6 关键要点

1. **缓冲容量**：`make(chan T, n)` 的 `n` 决定通道最多可以排队多少未消费的值。
2. **发送不再立即阻塞**：只要缓冲没满，发送操作会立刻返回。
3. **缓冲满仍会阻塞**：当容量用尽时，再次发送会等待接收者消费。
4. **读取仍旧阻塞**：没有数据可读时，接收操作会等待，语义不变。
5. **平滑生产消费**：适合处理生产者和消费者处理速度不一致的场景。

---

31. Channel Synchronization (通道同步)

https://gobyexample.com/channel-synchronization

31.1 核心概念

通过通道可以让不同 goroutine 协调执行顺序：发送方发送信号，接收方阻塞等待，从而保证某段逻辑完成后再继续下一步。相比盲目 `Sleep`，这种方式既精确又安全。

生活化类比：使用通道进行同步就像是接力赛中的交接棒。第一个选手（worker goroutine）完成自己的赛程后，将接力棒（在 `done` 通道中发送一个值）交给下一位选手（main goroutine）。下一位选手必须在交接区等待（阻塞在 `<-done`），直到接到棒才能开始自己的赛程。这确保了前一个任务的完成是下一个任务开始的精确信号。

> 注：如果移除文末的 `<-done`，主 goroutine 会直接退出，`worker` 输出也会随之丢失——这正是上一节"没打印"问题的根源。

31.2 使用通知通道

```
done := make(chan bool, 1) // 缓冲为 1，防止 worker 在程序退出前阻塞
```

使用缓冲通道的好处是：即使接收方稍晚一步 `<-done`，发送方也能先写入，不会因为主 goroutine 忙别的而阻塞住。

31.3 worker 中发送完成信号

```
func worker(done chan bool) {
    fmt.Print("working...")
    time.Sleep(time.Second) // 模拟耗时任务
    fmt.Println("done")

    done <- true // 告诉主 goroutine：任务完成了
}
```

> 小例子：在文件上传场景中，处理完压缩再发送 `done <- true`，主 goroutine 就能准确地进行后续操作（比如回写数据库状态）。

31.4 主 goroutine 等待信号

```
func main() {
    done := make(chan bool, 1)
    go worker(done)

    <-done // 阻塞，直到接收到 worker 发来的 true
}
```

这行 `<-done` 保证 `worker` 有机会打印 `working...done`，主 goroutine 会在收到信号前一直等待，不会提前退出。

31.5 完整代码示例（含注释）

```
package main

import (
    "fmt"
    "time"
)

func worker(done chan bool) {
    fmt.Print("working...")
    time.Sleep(time.Second) // 等待 1 秒模拟工作
    fmt.Println("done")

    done <- true // 通知主 goroutine 工作完成
}

func main() {
    done := make(chan bool, 1) // 缓冲通道，避免发送端阻塞

    go worker(done) // 启动 worker goroutine

    <-done // 等待 worker 完成，防止主程序过早结束
}
```

31.6 运行结果

```
$ go run channel-synchronization.go
working...done
```

输出中的两段文本挨在一起，是因为 `fmt.Print` 和 `fmt.Println` 紧挨着执行，没有额外换行时间差。

31.7 关键要点

1. **通道是同步工具**：通过发送/接收控制执行顺序，避免无控制的并发。
2. **缓冲防抖**：给通知通道配备容量，防止发送方在通知瞬间被阻塞。
3. **不要省略 `<-done`**：缺少阻塞等待，程序会在 goroutine 运行前结束。
4. **比 `Sleep` 精确**：等待真正的完成信号，而非依赖时间猜测。
5. **扩展到多任务**：多个 goroutine 可各自发信号，或者升级到 `sync.WaitGroup` 做批量等待。

---

32. Channel Directions (通道方向)

https://gobyexample.com/channel-directions

32.1 核心概念

在函数参数中标注通道的方向（仅发送或仅接收）能提升类型安全性，让编译器帮我们防止误用。例如，只发送通道无法执行接收操作，反之亦然。

生活化类比：为通道指定方向就像是给气动管道的两端贴上“只能放入”（`chan<-`）和“只能取出”（`<-chan`）的标签。这样一来，使用管道的人就不会在出口处硬塞东西，或是在入口处傻等东西出来。这让整个系统（程序）的意图更清晰，也更不容易出错。

> 注：方向限制只在函数签名里生效，通道本身依旧是双向的，只是类型系统帮我们“施加约束”。

32.2 定义只发送的通道形参

```
func ping(pings chan<- string, msg string) {
    pings <- msg // 只能发送，不能 `fmt.Println(<-pings)`
}
```

尝试从 `pings` 中接收会在编译时报错，避免了误把消息消耗掉的错误操作。

32.3 定义只接收 + 只发送的组合

```
func pong(pings <-chan string, pongs chan<- string) {
    msg := <-pings  // 只能读取
    pongs <- msg    // 只能写入
}
```

这种签名表明：`pong` 负责把来自 `pings` 的消息转发到 `pongs`，不会在函数内部错写方向。

> 小例子：在生产者-消费者模式中，消费者函数只需要读数据，不该向输入通道写入；利用方向限定能在编译时识别这些违规行为。

32.4 完整代码示例（含注释）

```
package main

import "fmt"
```

```go
func ping(pings chan<- string, msg string) {
    pings <- msg // 只负责发送消息
}

func pong(pings <-chan string, pongs chan<- string) {
    msg := <-pings    // 从输入通道取出消息
    pongs <- msg      // 再把消息发到输出通道
}

func main() {
    pings := make(chan string, 1)
    pongs := make(chan string, 1)

    ping(pings, "passed message")
    pong(pings, pongs)

    fmt.Println(<-pongs) // 从输出通道读取最终结果
}
```

32.5 运行结果

```
$ go run channel-directions.go
passed message
```

32.6 关键要点

1. **方向限定提升安全性**：`chan<-` 只发送，`<-chan` 只接收，编译器会阻止错误方向的操作。
2. **函数签名约束语义**：明确地表达该函数“只生产”或“只消费”数据，阅读更直观。
3. **通道本体仍双向**：方向限制只在形参生效，函数外部仍可用双向通道。
4. **组合使用**：常见模式是一个输入参数用 `<-chan`，另一个输出参数用 `chan<-`，形成转发管道。
5. **利于团队协作**：调用者可以放心地把通道交给函数，而不用担心函数里做了不该做的读写。

---

33. Select (选择器)

https://gobyexample.com/select

33.1 核心概念

`select` 语句允许我们同时等待多个通道操作，一旦某个 `case` 可执行就立刻调度到该分支。结合 goroutine 可以实现非阻塞的多路复用和超时控制，是 Go 并发模型的重要基石。

生活化类比：`select` 就像一个同时监控多条电话线的总机接线员。他不会只盯着一条线等电话，而是看着整个电话板。哪条线先亮灯（哪个通道先准备好），他就立刻接起哪一条。如果好几条线同时亮灯，他就随便挑一条接，确保不会冷落任何一个来电。

> 注：如果多个分支同时就绪，`select` 会随机选取一个执行，避免任何分支长期饥饿。

33.2 构建两个并发数据源

```
c1 := make(chan string)
c2 := make(chan string)

go func() {
    time.Sleep(1 * time.Second)
    c1 <- "one"
}()

go func() {
    time.Sleep(2 * time.Second)
    c2 <- "two"
}()
```

两个 goroutine 分别在 1 秒和 2 秒后写入数据，模拟并发的 RPC 或 IO 响应。

33.3 使用 `select` 同步多路输入

```
for range 2 {
    select {
    case msg1 := <-c1:
        fmt.Println("received", msg1)
    case msg2 := <-c2:
        fmt.Println("received", msg2)
    }
}
```

循环执行两次，每次等待任意一个通道返回数据。先到先处理，不需要显式指定优先级。

> 小例子：爬虫抓取多个网站时，用 `select` 可以优雅地处理先返回的结果，避免单个耗时请求阻塞全局。

33.4 完整代码示例（含注释）

```
package main

import (
    "fmt"
    "time"
)

func main() {
    c1 := make(chan string)
    c2 := make(chan string)

    go func() {
        time.Sleep(1 * time.Second) // 模拟耗时请求
        c1 <- "one"
    }()
```

```
    go func() {
        time.Sleep(2 * time.Second)
        c2 <- "two"
    }()

    for range 2 { // 共等待两个结果
        select {
        case msg1 := <-c1:
            fmt.Println("received", msg1)
        case msg2 := <-c2:
            fmt.Println("received", msg2)
        }
    }
}
```

33.5 运行结果

```
$ time go run select.go
received one
received two
real    0m2.2s
```

输出中可以看到总耗时约 2 秒，说明两个休眠是并发执行的，而不是串行等待 3 秒。

33.6 关键要点

1. **多路等待**：`select` 能同时监听多个通道，谁先就绪就处理谁。
2. **无优先权**：多个分支同时可用时随机挑选，写法上要做好可交换的准备。
3. **结合循环**：常见模式是外层 `for` 搭配 `select`，持续消费事件流。
4. **天然超时支持**：与 `time.After` 通道组合可实现请求超时控制（下一节将看到）。
5. **减少阻塞**：避免单一慢操作拖累整体进度，提升并发程序的吞吐。

---

34. Timeouts (超时处理)

https://gobyexample.com/timeouts

34.1 核心概念

当我们依赖外部资源（网络请求、RPC 调用等）时，需要防止长时间等待。Go 结合 `select` 和 `time.After` 可以优雅地实现超时控制，一旦超时立即走备用逻辑。

生活化类比：使用 `select` 实现超时就像是点外卖时设定一个“我最多等30分钟”的心理预期。你要么在30分钟内等到了外卖（从业务通道 `c1` 收到结果），要么30分钟一到，闹钟响了（从 `time.After` 通道收到信号），你就不再等了，直接取消订单或者吃泡面。哪个事件先发生，就按哪个剧本走。

> 注：`time.After(d)` 会返回一个通道，`select` 在等待超过 `d` 后会从该通道收到一个事件。

34.2 设置超时的基本模式

```
c1 := make(chan string, 1) // 带 1 个缓冲，避免超时后发送方阻塞

go func() {
    time.Sleep(2 * time.Second) // 模拟慢响应
    c1 <- "result 1"            // 如果主流程超时没取，也能成功发送
}()

select {
case res := <-c1: // 按时返回结果
    fmt.Println(res)
case <-time.After(1 * time.Second): // 超过 1 秒未返回就超时
    fmt.Println("timeout 1")
}
```

由于 `time.After(1s)` 更早触发，`select` 会执行超时分支，避免主 goroutine 被无限挂起。

> 小贴士：`c1` 设置缓冲的原因是，如果超时分支先执行但我们不再读取 `c1`，后台 goroutine 的发送操作也不会被卡住，避免 goroutine 泄漏。

34.3 调整超时时间以等待成功结果

```
c2 := make(chan string, 1) // 同样使用带缓冲通道

go func() {
    time.Sleep(2 * time.Second)
    c2 <- "result 2" // 2 秒后返回成功
}()

select {
case res := <-c2: // 在 3 秒超时前收到结果
    fmt.Println(res)
case <-time.After(3 * time.Second): // 若超过 3 秒则视为失败
    fmt.Println("timeout 2")
}
```

这里超时时间为 3 秒，足以等到 `c2` 返回，因此可以成功打印 `result 2`。

34.4 完整代码示例（含注释）

```
package main

import (
    "fmt"
    "time"
)

func main() {
    c1 := make(chan string, 1) // 为避免超时导致的阻塞，使用带缓冲通道
```

```
    go func() {
        time.Sleep(2 * time.Second) // 模拟一个慢接口
        c1 <- "result 1"            // 尝试写入结果
    }()

    select {
    case res := <-c1: // 如在 1 秒内拿到结果，直接打印
        fmt.Println(res)
    case <-time.After(1 * time.Second): // 否则触发超时逻辑
        fmt.Println("timeout 1")
    }

    c2 := make(chan string, 1) // 第二个请求，同样带缓冲
    go func() {
        time.Sleep(2 * time.Second)
        c2 <- "result 2"
    }()

    select {
    case res := <-c2: // 3 秒内返回成功
        fmt.Println(res)
    case <-time.After(3 * time.Second): // 超时则输出超时信息
        fmt.Println("timeout 2")
    }
}
```

34.5 运行结果

```
$ go run timeouts.go
timeout 1
result 2
```

34.6 关键要点

1. **`time.After` 辅助超时**：配合 `select`，可以在等待通道结果时设定最长期限。
2. **缓冲通道防泄漏**：超时后不再读取时，缓冲位能保证发送操作完成。
3. **灵活调整期限**：根据业务需求设置不同超时时间，避免统一超时造成的误判。
4. **优雅的并发控制**：比无限等待更安全，也比全局 `Sleep` 更精准。
5. **可扩展模式**：在多路 `select` 中，同时监听多个请求和多个超时线路，实现复杂容错。

---

35. Non-Blocking Channel Operations (非阻塞通道操作)

https://gobyexample.com/non-blocking-channel-operations

35.1 核心概念

默认的通道读写都会阻塞等待对端。如果不希望阻塞，可利用 `select` 的 `default` 分支实现“立即返回”的非阻塞操作：当所有 `case` 都无法立刻执行时，会落到 `default`，从而给我们机会处理“暂时没有数据”的情况。

生活化类比：非阻塞操作就像是路过信箱时瞥一眼。如果信箱里有信（通道可读/可写），就顺手取走或放入。如果信箱是空的（通道不可读/写），你不会站在原地等，而是立刻走开去做别的事情（执行 `default` 分支）。这是一种“有就处理，没有就拉倒”的轻快交互模式。

> 注：非阻塞操作只检查当前是否可读/可写，不会等待，因此适合轮询或状态探测，但要注意避免忙等。

35.2 非阻塞读取

```
messages := make(chan string) // 无缓冲通道，当前没有发送者

select {
case msg := <-messages: // 只在有数据时才会执行
    fmt.Println("received message", msg)
default: // 否则立即走 default，不会卡住程序
    fmt.Println("no message received")
}
```

35.3 非阻塞写入

```
msg := "hi"

select {
case messages <- msg: // 通道有空间（或有人在读）时才能成功发送
    fmt.Println("sent message", msg)
default:
    fmt.Println("no message sent") // 当前没人接收，直接返回
}
```

> 小例子：日志系统里可以尝试把日志写入通道，如果通道拥堵就及时丢弃或降级，避免主流程被拖慢。

35.4 多路非阻塞 `select`

```
signals := make(chan bool)

select {
case msg := <-messages: // 同时探测消息通道
    fmt.Println("received message", msg)
case sig := <-signals:  // 探测信号通道
    fmt.Println("received signal", sig)
default:
    fmt.Println("no activity")
}
```

通过在一个 `select` 里列出多个通道，可以快速判断当前是否有任何事件需要处理。

35.5 完整代码示例（含注释）

```go
package main

import "fmt"

func main() {
    messages := make(chan string) // 无缓冲通道，此刻没有数据
    signals := make(chan bool)

    // 非阻塞读取：没有人发送，因此落入 default
    select {
    case msg := <-messages:
        fmt.Println("received message", msg)
    default:
        fmt.Println("no message received")
    }

    // 非阻塞写入：同样因为无人接收而进入 default
    msg := "hi"
    select {
    case messages <- msg:
        fmt.Println("sent message", msg)
    default:
        fmt.Println("no message sent")
    }

    // 多路非阻塞：同时检查 messages 与 signals
    select {
    case msg := <-messages:
        fmt.Println("received message", msg)
    case sig := <-signals:
        fmt.Println("received signal", sig)
    default:
        fmt.Println("no activity")
    }
}
```

35.6 运行结果

```
$ go run non-blocking-channel-operations.go
no message received
no message sent
no activity
```

35.7 关键要点

1. **`default` 分支决定非阻塞**：若所有通道都暂不可用，立即执行 `default`。
2. **避免阻塞主流程**：适合在高优先级任务里探测通道状态，防止被慢通道拖累。
3. **注意忙等风险**：需要配合 `time.Sleep`、计时器或其他机制避免空转。

4. **多路探测**：一个 `select` 可以同时检查多个通道，有事件就处理，没有就快速返回。
5. **与缓冲通道结合**：若通道有缓冲或有人接收，非阻塞发送也可能立即成功。

---

36. Closing Channels (关闭通道)

https://gobyexample.com/closing-channels

36.1 核心概念

关闭通道表示后续不会再有发送操作。接收方能通过第二个返回值判别通道是否已关闭，常用于通知消费者“工作已完成”。

生活化类比：关闭通道就像是工厂下班时，生产线领班（发送方）按下停止按钮并大喊一声：“今天的活儿全在这了，没新的了！”传送带上的工人（接收方）听到后，会把传送带上剩下的所有零件处理完，然后就知道可以下班了，而不会傻傻地等下一个永远不会来的零件。

> 注：只有发送方应该关闭通道，接收方没必要关闭；往已关闭的通道再次发送会导致 panic。

36.2 生产者关闭通道通知消费者

```
jobs := make(chan int, 5)
done := make(chan bool)
```

生产者往 `jobs` 投递任务，消费者从中读取。当不再有任务时，生产者通过 `close(jobs)` 告知消费者收工。

36.3 消费者识别关闭状态

```
go func() {
    for {
        j, more := <-jobs
        if more {
            fmt.Println("received job", j)
        } else {
            fmt.Println("received all jobs")
            done <- true // 通知主 goroutine
            return
        }
    }
}()
```

双返回值中的 `more` 为 `false` 表示通道已关闭且没有更多数据，此时消费者可以安全退出。

36.4 发送完毕后关闭通道

```
for j := 1; j <= 3; j++ {
    jobs <- j
    fmt.Println("sent job", j)
}
close(jobs) // 关闭通道，告诉消费者没有新任务
fmt.Println("sent all jobs")

<-done // 等待消费者确认全部完成
```

> 小例子：在 worker 池中，派发者在所有任务入队后调用 `close(jobs)`，避免消费者因等待更多任务而阻塞。

36.5 读取关闭通道的零值

```
_, ok := <-jobs
fmt.Println("received more jobs:", ok) // ok 为 false，说明通道已关闭
```

从关闭的通道读取会立刻返回类型的零值（这里是 `0`），同时 `ok` 为 `false`，适合用来验证通道状态。

36.6 完整代码示例（含注释）

```
package main

import "fmt"

func main() {
    jobs := make(chan int, 5)
    done := make(chan bool)

    // worker goroutine: 持续接收任务，直到通道被关闭
    go func() {
        for {
            j, more := <-jobs
            if more {
                fmt.Println("received job", j)
            } else {
                fmt.Println("received all jobs")
                done <- true // 告诉主 goroutine 工作结束
                return
            }
        }
    }()

    for j := 1; j <= 3; j++ {
        jobs <- j
        fmt.Println("sent job", j)
    }

    close(jobs)           // 关闭通道，避免 goroutine 阻塞等待
    fmt.Println("sent all jobs")
```

```
    <-done                // 等待 worker 完成所有任务

    _, ok := <-jobs       // 再次读取得到零值与 false
    fmt.Println("received more jobs:", ok)
}
```

36.7 运行结果

```
$ go run closing-channels.go
sent job 1
received job 1
sent job 2
received job 2
sent job 3
received job 3
sent all jobs
received all jobs
received more jobs: false
```

36.8 关键要点

1. **关闭表示完成**：只在发送者侧关闭通道，避免资源浪费。
2. **双返回值判定**：`v, ok := <-ch` 中的 `ok` 用来判断是否还有数据。
3. **零值读取**：通道关闭后读取会返回零值，注意区分业务值和“无值”的情况。
4. **不要重复关闭**：多次关闭或关闭非发送者持有的通道会 panic。
5. **搭配 `done` 通道**：关闭任务通道后，可使用额外的 `done` 通道同步所有 worker 的结束状态。

---

37. Range over Channels (遍历通道)

https://gobyexample.com/range-over-channels

37.1 核心概念

`for range` 不仅可以迭代切片、字典，也可以遍历通道中的数据。它会不断接收，直到通道被关闭并且队列里的数据消费完毕，这让代码更简洁，不需要显式构建 `for { ... break }` 循环。

生活化类比：`for range` 遍历通道就像是从一个扭蛋机里不断地取出扭蛋。你只需要一直转动把手（`for elem := range queue`），每次都会出来一个扭蛋（元素）。你不需要关心里面到底还有多少个，只管取就是了。直到扭蛋机空了，并且管理员贴上了“已售罄”的封条（通道被关闭），你的 `for range` 循环就自动结束了。

> 注：`range` 依赖通道被关闭后才能退出；如果通道永远不关闭，循环会无限阻塞。

37.2 先写入再关闭

```
queue := make(chan string, 2)
queue <- "one"
queue <- "two"
close(queue) // 关闭后仍可以读取缓冲中剩余的元素
```

即使通道已经关闭，只要缓冲中还有数据，接收方仍能读取完这些值。

37.3 使用 `range` 接收所有元素

```
for elem := range queue {
    fmt.Println(elem)
}
```

循环会依次输出 `one`、`two`，直到通道耗尽并关闭，此时自动结束。

> 小例子：在生产者-消费者模型中，生产者关闭通道即可让消费者使用 `for range` 自动收尾，而无需额外的结束标记。

37.4 完整代码示例（含注释）

```
package main

import "fmt"

func main() {
    queue := make(chan string, 2)
    queue <- "one"
    queue <- "two"
    close(queue) // 告诉消费者没有更多元素

    for elem := range queue { // 自动遍历直到通道关闭
        fmt.Println(elem)
    }
}
```

37.5 运行结果

```
$ go run range-over-channels.go
one
two
```

37.6 关键要点

1. **自动退出**：`range` 会在通道关闭且没有元素时结束循环。
2. **需要关闭通道**：若通道不关闭，`range` 会一直阻塞等待新数据。
3. **可先关闭后消费**：关闭通道并不会丢失缓冲中的数据。
4. **更简洁的消费模式**：替代手动 `for {}` 和 `ok` 判断结构。

5. **配合关闭语义**：适用于生产者在结束时调用 `close`，让消费者自然退出。

---

38. Timers (定时器)

https://gobyexample.com/timers

38.1 核心概念

`time.NewTimer` 用于在未来某个时刻触发一次事件。定时器提供 `C` 通道，等到指定时长后会往里面发送信号，结合 `select` 或直接 `<-timer.C` 即可等待或取消。

生活化类比：`Timer` 就像一个厨房定时器。你为烤蛋糕设定了10分钟，然后就可以去做别的事情。10分钟一到，定时器就会"叮"地响一声（向 `timer.C` 通道发送一个值），提醒你去查看烤箱。如果你提前完成了其他事，也可以在它响之前手动关掉它（`timer.Stop()`）。

> 注：定时器和 `time.Sleep` 的区别在于可以提前 `Stop`、重置或将 `timer.C` 与其他通道一起监听。

38.2 基本触发流程

```
timer1 := time.NewTimer(2 * time.Second)
<-timer1.C // 阻塞 2 秒后继续执行
fmt.Println("Timer 1 fired")
```

`<-timer1.C` 会阻塞直到定时器到期，这里相当于等待 2 秒后打印信息。

38.3 取消定时器

```
timer2 := time.NewTimer(1 * time.Second)

go func() {
    <-timer2.C
    fmt.Println("Timer 2 fired") // 若被 Stop，几乎不会执行到
}()

if stop2 := timer2.Stop(); stop2 {
    fmt.Println("Timer 2 stopped")
}
```

通过 `Stop()` 可以阻止定时器触发。返回值为 `true` 表示定时器确实被取消；若返回 `false`，说明要么已经触发，要么已经被取消过。

> 小贴士：如果 `Stop()` 返回 `false`，常见做法是非阻塞地读取一次 `timer.C`，以清理剩余的触发信号，防止后续误读（示例中等待 2 秒以确保 goroutine 不会误打印）。

38.4 等待确保取消成功

```
time.Sleep(2 * time.Second) // 给 goroutine 时间验证是否真的停止
```

若未 `Stop`，`timer2` 会触发并输出；这里因为成功调用 `Stop`，所以只会看到“Timer 2 stopped”。

38.5 完整代码示例（含注释）

```
package main

import (
    "fmt"
    "time"
)

func main() {
    timer1 := time.NewTimer(2 * time.Second)
    <-timer1.C // 等待 2 秒
    fmt.Println("Timer 1 fired")

    timer2 := time.NewTimer(1 * time.Second)
    go func() {
        <-timer2.C             // 如果未被 Stop，就会触发
        fmt.Println("Timer 2 fired")
    }()

    if stop2 := timer2.Stop(); stop2 { // 尝试取消
        fmt.Println("Timer 2 stopped")
    }

    time.Sleep(2 * time.Second) // 确定 goroutine 不会输出
}
```

38.6 运行结果

```
$ go run timers.go
Timer 1 fired
Timer 2 stopped
```

38.7 关键要点

1. **一次性事件**：Timer 只触发一次；若需周期执行请使用 ticker。
2. **通道通知**：`timer.C` 提供触发信号，可与其他通道组合等待。
3. **可取消**：`Stop()` 让你在超时前取消事件，返回值表明取消是否成功。
4. **注意清理**：若 Stop 失败，需要 drain `timer.C` 避免后续读取到旧信号。
5. **结合 goroutine**：常见模式是由后台 goroutine 等待定时器，而主流程决定是否提前取消。

---

39. Tickers (打点器)

https://gobyexample.com/tickers

39.1 核心概念

`ticker` 用于按照固定间隔重复触发事件。与一次性的 `timer` 不同，`ticker` 每隔设定的时长就在其 `C` 通道上发送当前时间，直到被 `Stop()` 停止。

生活化类比：`Ticker` 就像一个节拍器。一旦启动，它就会以固定的节奏“滴答”作响（定期向 `ticker.C` 通道发送信号），永不停止。这对于需要按时重复执行的任务（比如每秒发送一次心跳检测）非常有用。当练习结束时，你可以关掉节拍器（`ticker.Stop()`）。

> 注：`ticker.C` 是一个只读通道，按节奏推送 `time.Time` 值，适合做心跳、定时任务等场景。

39.2 创建与监听 ticker

```
ticker := time.NewTicker(500 * time.Millisecond)
done := make(chan bool)

go func() {
    for {
        select {
        case <-done: // 外部通知停止
            return
        case t := <-ticker.C: // 每隔 500ms 触发一次
            fmt.Println("Tick at", t)
        }
    }
}()
```

通过 `select` 同时监听 `ticker.C` 与 `done`，既能打印 tick，也能随时退出循环。

39.3 停止 ticker

```
time.Sleep(1600 * time.Millisecond) // 等待 3 次 tick 左右
ticker.Stop()
done <- true
fmt.Println("Ticker stopped")
```

调用 `Stop()` 后不再有新事件发送到 `ticker.C`，务必通知后台 goroutine 退出，避免泄漏。

> 小例子：服务健康检查可以用 ticker 定时探测；当应用关闭时停止 ticker 并退出协程，确保清理资源。

39.4 完整代码示例（含注释）

```
package main

import (
    "fmt"
    "time"
)

func main() {
```

```
    ticker := time.NewTicker(500 * time.Millisecond)
    done := make(chan bool)

    go func() {
        for {
            select {
            case <-done:
                return // 收到停止信号
            case t := <-ticker.C:
                fmt.Println("Tick at", t) // 定期打印当前时间
            }
        }
    }()

    time.Sleep(1600 * time.Millisecond) // 约 3 次 tick
    ticker.Stop()
    done <- true
    fmt.Println("Ticker stopped")
}
```

39.5 运行结果

```
$ go run tickers.go
Tick at 2024-05-01 10:00:00.500 +0800 CST
Tick at 2024-05-01 10:00:01.000 +0800 CST
Tick at 2024-05-01 10:00:01.500 +0800 CST
Ticker stopped
```

39.6 关键要点

1. **周期触发**：ticker 每隔固定时长发送一次当前时间。
2. **停止后不再触发**：调用 `Stop()` 后应尽快退出相关 goroutine。
3. **协程通信**：常用 `done` 或 `context` 控制 ticker 的生命周期。
4. **与 timer 区别**：timer 只触发一次，而 ticker 周期性触发。
5. **避免资源泄漏**：如果忘记停止，ticker 会持续运行并持有定时器资源。

---

40. Worker Pools (工作池)

https://gobyexample.com/worker-pools

40.1 核心概念

Worker Pool 通过一组长期存活的 goroutine 来并行处理任务。典型模式是生产者把任务放入通道，多个 worker 同时从通道取任务执行，再把结果写入另一个通道或做后续处理。

生活化类比：工作池就像一个运营着固定数量出租车的调度中心。乘客（`jobs`）在调度中心排队等候，调度中心不会为每个乘客都派一辆新车，而是让固定数量的出租车（`worker` goroutines）循环接单。一辆车送完乘客（完成一个任务）后，就立刻回来接下一位。这种模式既能高效地服务所有乘客，又能避免街上同时出现过多出租车造成拥堵（控制并发度）。

> 注：Worker Pool 的关键在于“任务队列 + 多个消费者 + 控制退出”。这样能充分利用多核、限制并发度，同时避免频繁创建 goroutine 的开销。

40.2 定义 worker 函数

```
func worker(id int, jobs <-chan int, results chan<- int) {
    for j := range jobs {
        fmt.Println("worker", id, "started job", j)
        time.Sleep(time.Second)    // 模拟耗时操作
        fmt.Println("worker", id, "finished job", j)
        results <- j * 2           // 将结果写回
    }
}
```

- `jobs <-chan int`：只读通道，worker 从中获取任务。
- `results chan<- int`：只写通道，将处理结果发送给主 goroutine。
- 使用 `range jobs` 在 `jobs` 被关闭后自动退出。

40.3 启动固定数量的 worker

```
const numJobs = 5
jobs := make(chan int, numJobs)
results := make(chan int, numJobs)

for w := 1; w <= 3; w++ {
    go worker(w, jobs, results)
}
```

三个 worker 并行工作。如果想限制并发度，就控制创建的 worker 数量。

40.4 投递任务并关闭通道

```
for j := 1; j <= numJobs; j++ {
    jobs <- j
}
close(jobs) // 关闭后，worker 遇到 range 结束
```

通道关闭后，所有 worker 完成手头任务即会退出循环，避免 goroutine 漏泄。

40.5 收集结果并等待 worker 完成

```
for a := 1; a <= numJobs; a++ {
    <-results // 逐个读取，确保所有任务已完成
}
```

读取结果不仅获取输出，还能作为“等待所有 worker 完成”的手段；另一种做法是搭配 `sync.WaitGroup`。

> 小例子：在文件处理系统中，一边读取待处理文件名（投递任务），一边由固定数量的 worker 执行转换，主线程从 `results` 读取转换结果或错误信息。

40.6 完整代码示例（含注释）

```
package main

import (
    "fmt"
    "time"
)

func worker(id int, jobs <-chan int, results chan<- int) {
    for j := range jobs {
        fmt.Println("worker", id, "started  job", j)
        time.Sleep(time.Second)
        fmt.Println("worker", id, "finished job", j)
        results <- j * 2
    }
}

func main() {
    const numJobs = 5
    jobs := make(chan int, numJobs)
    results := make(chan int, numJobs)

    for w := 1; w <= 3; w++ {
        go worker(w, jobs, results)
    }

    for j := 1; j <= numJobs; j++ {
        jobs <- j
    }
    close(jobs)

    for a := 1; a <= numJobs; a++ {
        <-results
    }
}
```

40.7 运行结果

```
$ time go run worker-pools.go
worker 1 started  job 1
worker 2 started  job 2
worker 3 started  job 3
worker 1 finished job 1
worker 1 started  job 4
worker 2 finished job 2
worker 2 started  job 5
worker 3 finished job 3
worker 1 finished job 4
worker 2 finished job 5
real    0m2.3s
```

40.8 关键要点

1. **通道管理任务**：`jobs` 是共享任务队列，通过关闭信号结束消费。
2. **限制并发度**：worker 数量决定同时处理任务的上限。
3. **worker 自动退出**：使用 `range` + 关闭通道即可优雅收尾。
4. **收集结果**：单独的 `results` 通道或 `WaitGroup` 都可以确认任务完成。
5. **防止泄漏**：及时关闭 `jobs`、读取 `results`，避免 goroutine 因等待或通道阻塞而堆积。

---

41. WaitGroups (等待组)

https://gobyexample.com/waitgroups

41.1 核心概念

`sync.WaitGroup` 提供一种简单方式等待一组 goroutine 完成。当你知道要启动多少个任务，但不想手动管理计数时，调用 `Add`/`Done`（或在 Go 1.22+ 使用 `WaitGroup.Go`）即可确保主 goroutine 在所有子任务结束后再继续。

生活化类比：`WaitGroup` 就像是聚会的主人。主人知道邀请了5位客人（`wg.Add(5)`）。每到一位客人，就在名单上划掉一个名字（`wg.Done()`）。主人则会在门口一直等到名单上所有客人都到齐为止（`wg.Wait()`），然后才开始派对。

> 注：`WaitGroup` 必须通过指针传递；无论使用 `Add/Done` 还是 `Go` 方法，底层都依赖同一个计数器。如果复制 `WaitGroup`，会导致 panic。

41.2 定义任务函数

```
func worker(id int) {
    fmt.Printf("Worker %d starting\n", id)
    time.Sleep(time.Second) // 模拟耗时操作
    fmt.Printf("Worker %d done\n", id)
}
```

41.3 使用 `WaitGroup.Go` 启动任务

```
var wg sync.WaitGroup

for i := 1; i <= 5; i++ {
    i := i // 避免闭包捕获同一个循环变量
    wg.Go(func() {
        worker(i)
    })
}

wg.Wait() // 阻塞直到所有 goroutine 返回
```

- `WaitGroup.Go` 会在内部调用 `Add(1)`，随后执行传入的函数，完成后自动调用 `Done()`。
- 若你使用较低版本 Go，可以手动 `wg.Add(1)` 和 `defer wg.Done()` 来实现同样效果。

> 小例子：在微服务中异步调用多个下游服务，把每次调用封装在 `wg.Go` 里，主流程 `wg.Wait()` 等待所有结果返回再聚合。

41.4 完整代码示例（含注释）

```
package main

import (
    "fmt"
    "sync"
    "time"
)

func worker(id int) {
    fmt.Printf("Worker %d starting\n", id)
    time.Sleep(time.Second)
    fmt.Printf("Worker %d done\n", id)
}

func main() {
    var wg sync.WaitGroup

    for i := 1; i <= 5; i++ {
        i := i // 每个 goroutine 捕获自己的编号
        wg.Go(func() {
            worker(i)
        })
    }

    wg.Wait() // 等待所有 worker 完成
}
```

41.5 运行结果

```
$ go run waitgroups.go
Worker 5 starting
Worker 3 starting
Worker 4 starting
Worker 1 starting
Worker 2 starting
Worker 4 done
Worker 1 done
Worker 2 done
Worker 5 done
Worker 3 done
```

实际启动/完成顺序取决于调度器，每次运行可能不同。

41.6 关键要点

1. **计数型同步**：`WaitGroup` 记录尚未完成的 goroutine 数量。
2. **指针传递**：务必避免复制 `WaitGroup`，否则会出现 “copy of sync.WaitGroup value” panic。
3. **版本差异**：Go 1.22 起可使用 `WaitGroup.Go`，旧版本需手动 `Add` / `Done`。
4. **闭包变量**：循环内的 goroutine 要创建局部变量（`i := i`）避免捕获同一个值。
5. **无错返回**：`WaitGroup` 不负责错误汇总，如需收集错误可搭配通道或 `errgroup`。

---

42. Rate Limiting (速率限制)

https://gobyexample.com/rate-limiting

42.1 核心概念

限流（Rate Limiting）用于控制请求处理的速率，避免系统被突发流量压垮。Go 可以通过 `time.Tick` / `time.Ticker` 与通道协作，实现排队发放“令牌”或允许短暂“突发”的漏斗效果。

生活化类比：限流器就像是游乐园里热门项目的入口闸机。闸机每隔一段时间才放一个人进去（固定速率），保证了项目不会因为一次涌入太多人而混乱。而支持“突发”的限流器则像是在闸机前设置了一个等候区，可以先容纳一小批人，让他们快速通过，但等候区满了之后，依然要恢复到“一个一个进”的慢速模式。

> 注：可以把限流器想象成发放令牌的值班员——没有令牌就不能处理请求；令牌按固定节奏产出，也可以预先备一些应对小规模尖峰。

42.2 基础限流——严格的固定频率

```
requests := make(chan int, 5)
for i := 1; i <= 5; i++ {
    requests <- i
}
close(requests)

limiter := time.Tick(200 * time.Millisecond) // 每 200ms 产出一个令牌

for req := range requests {
    <-limiter // 没令牌就阻塞在这里，相当于排队等节奏
    fmt.Println("request", req, time.Now())
}
```

- `time.Tick` 返回的通道会按照设定间隔持续产出当前时间，常用于简易的固定节奏限流。
- 每次循环先读取 `limiter`（拿令牌），再处理请求，确保两次输出间隔约 200ms。

42.3 突发限流——允许瞬时拥簇但总体受控

```
burstyLimiter := make(chan time.Time, 3) // 令牌桶容量 3

// 预填充 3 个令牌，让前三个请求立刻通过
for i := 0; i < 3; i++ {
    burstyLimiter <- time.Now()
}

// 后续每 200ms 再往桶里添加一个令牌
go func() {
    for t := range time.Tick(200 * time.Millisecond) {
        burstyLimiter <- t
    }
}()

burstyRequests := make(chan int, 5)
for i := 1; i <= 5; i++ {
    burstyRequests <- i
}
close(burstyRequests)

for req := range burstyRequests {
    <-burstyLimiter // 令牌耗尽后会阻塞，恢复为固定速率
    fmt.Println("request", req, time.Now())
}
```

- 缓冲通道 `burstyLimiter` 就像一个“令牌桶”，容量 3 表示允许 3 个请求瞬时通过。
- 再次获取令牌时，如果桶空了，就会阻塞直到后台 goroutine 按周期补充令牌，整体速率仍受限。

> 通俗理解：`burstyLimiter` 就像提前存了 3 张通行证的小盒子，前 3 个请求一拿就走；之后只能等管理员每 200ms 补一张。

42.4 完整代码示例（含注释）

```go
package main

import (
    "fmt"
    "time"
)

func main() {
    // ---- 严格限流：一个请求必须等一个令牌 ----
    requests := make(chan int, 5)
    for i := 1; i <= 5; i++ {
        requests <- i
    }
    close(requests)

    limiter := time.Tick(200 * time.Millisecond) // 固定速率产出令牌

    for req := range requests {
        <-limiter // 没拿到令牌就不能继续，保证间隔
        fmt.Println("request", req, time.Now())
    }

    // ---- 突发限流：允许短暂的高峰 ----
    burstyLimiter := make(chan time.Time, 3)

    for i := 0; i < 3; i++ {
        burstyLimiter <- time.Now() // 先塞满 3 个令牌应对突发
    }

    go func() {
        for t := range time.Tick(200 * time.Millisecond) {
            burstyLimiter <- t // 每 200ms 补充一个令牌
        }
    }()

    burstyRequests := make(chan int, 5)
    for i := 1; i <= 5; i++ {
        burstyRequests <- i
    }
    close(burstyRequests)

    for req := range burstyRequests {
        <-burstyLimiter // 令牌用完会阻塞，速率恢复到 200ms
        fmt.Println("request", req, time.Now())
    }
}
```

42.5 运行结果

```
$ go run rate-limiting.go
request 1 2024-05-01 10:00:00.200 +0800 CST
request 2 2024-05-01 10:00:00.400 +0800 CST
request 3 2024-05-01 10:00:00.600 +0800 CST
request 4 2024-05-01 10:00:00.800 +0800 CST
request 5 2024-05-01 10:00:01.000 +0800 CST
request 1 2024-05-01 10:00:01.000 +0800 CST
request 2 2024-05-01 10:00:01.000 +0800 CST
request 3 2024-05-01 10:00:01.000 +0800 CST
request 4 2024-05-01 10:00:01.200 +0800 CST
request 5 2024-05-01 10:00:01.400 +0800 CST
```

上半段是严格限流，输出间隔接近 200ms；下半段前三个请求瞬间完成，后两个恢复 200ms 间隔。

42.6 关键要点

1. **固定速率**：`time.Tick` + `<-limiter` 简洁实现“一个请求一个令牌”。
2. **突发处理**：带缓冲的令牌通道允许预存令牌，实现“借一点未来的额度”。
3. **后台补充令牌**：用 goroutine + `time.Tick` 持续往桶里放令牌，保持长期速率。
4. **不可关闭 `Tick` 返回值**：`time.Tick` 内部用全局 ticker，若需停止应该用 `time.NewTicker` 并 `Stop()`。
5. **灵活扩展**：可以根据业务把数字换成真实请求对象、统计延迟、记录丢弃等，核心逻辑依旧是"先拿令牌，再执行"。

---

43. Atomic Counters (原子计数器)

https://gobyexample.com/atomic-counters

43.1 核心概念

Go 语言中管理状态的主要机制是通过通道在 goroutine 之间通信。但当你需要一个简单的、多个 goroutine 共享的计数器时，使用"原子操作"（atomic operations）会更高效。`sync/atomic` 包提供了低级别的原子内存操作，适合实现无锁的并发计数器。

生活化类比：原子操作就像是体育馆入口处保安使用的电子计数器。即使同时有多个门开放，大量观众（并发的 goroutine）涌入，每次按动按钮（`ops.Add(1)`）都是一个不可分割的、瞬间完成的操作。这避免了两个保安同时去拨动一个老式机械计数器而导致计数混乱或损坏的问题，确保最终入场人数绝对准确。

> 注：原子操作保证在多核 CPU 上对同一内存地址的读写不会产生数据竞争，避免了传统互斥锁的开销。

43.2 为什么需要原子操作

43.2.1 数据竞争问题

如果多个 goroutine 同时对一个普通变量进行读写，会产生数据竞争：

```
var counter int

// 多个 goroutine 同时执行
counter++  // 不是原子操作！实际分为三步：读取、加一、写回
```

这种情况下最终的计数值往往会小于预期，因为多个 goroutine 可能读到相同的旧值。

43.2.2 原子操作的优势

使用原子操作可以确保计数器的增减操作在硬件层面是不可分割的：

```
var ops atomic.Uint64
ops.Add(1)  // 原子地增加，无论多少 goroutine 并发执行都安全
```

43.3 使用原子计数器

43.3.1 声明原子类型

Go 1.19+ 提供了类型安全的原子类型，无需指针即可使用：

```
var ops atomic.Uint64  // 无符号 64 位整数的原子类型
```

对于旧版本 Go，需要使用 `uint64` 配合 `atomic.AddUint64(&ops, 1)` 的方式。

43.3.2 并发增加计数器

```
var wg sync.WaitGroup

for range 50 {  // 创建 50 个 goroutine
    wg.Go(func() {
        for range 1000 {
            ops.Add(1)  // 每个 goroutine 增加 1000 次
        }
    })
}

wg.Wait()  // 等待所有 goroutine 完成
```

这里启动了 50 个 goroutine，每个执行 1000 次递增，总共应该是 50,000 次操作。

43.3.3 安全读取最终值

```
fmt.Println("ops:", ops.Load())  // 原子地读取当前值
```

`Load()` 方法确保读取操作也是原子的，不会读到"半个数据"。

> 小贴士：在运行程序时加上 `-race` 标志（`go run -race`）可以检测数据竞争。如果使用普通变量而非原子操作，race detector 会报错。

43.4 完整代码示例

```
package main

import (
  "fmt"
  "sync"
  "sync/atomic"
)

func main() {
  // 声明一个原子计数器
  var ops atomic.Uint64

  // 使用 WaitGroup 等待所有 goroutine 完成
  var wg sync.WaitGroup

  // 启动 50 个 goroutine，每个执行 1000 次递增
  for range 50 {
    wg.Go(func() {
      for range 1000 {
        // 原子地增加计数器
        ops.Add(1)
      }
    })
  }

  // 等待所有 goroutine 完成
  wg.Wait()

  // 原子地读取最终计数值
  fmt.Println("ops:", ops.Load())
}
```

43.5 运行结果

```
$ go run atomic-counters.go
ops: 50000
```

输出恒定为 50,000，因为原子操作保证了所有增量都被正确记录。

43.6 其他原子操作

43.6.1 原子类型家族

`sync/atomic` 提供了多种原子类型：

```
var i32 atomic.Int32      // 有符号 32 位整数
var i64 atomic.Int64      // 有符号 64 位整数
var u32 atomic.Uint32     // 无符号 32 位整数
var u64 atomic.Uint64     // 无符号 64 位整数
var b atomic.Bool         // 布尔值
var p atomic.Pointer[T]   // 指针类型
```

43.6.2 常用方法

```
// 读取
val := ops.Load()

// 存储（直接赋值）
ops.Store(100)

// 增加
ops.Add(5)

// 比较并交换（CAS）
swapped := ops.CompareAndSwap(oldVal, newVal)

// 交换（返回旧值）
oldVal := ops.Swap(newVal)
```

43.7 原子操作 vs 互斥锁

43.7.1 何时使用原子操作

- 简单的计数器或标志位
- 性能要求高的热点代码
- 无需保护多个变量的场景

43.7.2 何时使用互斥锁

- 需要保护多个相关变量
- 操作逻辑复杂，不止是读写
- 需要条件变量等高级同步机制

```
// 原子操作适用场景
var counter atomic.Uint64
counter.Add(1)

// 互斥锁适用场景
var mu sync.Mutex
var balance int
var transactions []string

mu.Lock()
balance += 100
transactions = append(transactions, "deposit")
mu.Unlock()
```

43.8 关键要点

1. **无锁并发**：原子操作在硬件层面保证线程安全，比互斥锁开销更小。
2. **类型安全**：Go 1.19+ 的 `atomic.Uint64` 等类型避免了手动传递指针的麻烦。
3. **适用场景**：适合简单的计数器、标志位等单一变量的并发访问。
4. **race detector**：使用 `go run -race` 可以检测程序中的数据竞争问题。
5. **读写都要原子**：不仅写入要用 `Add/Store`，读取也要用 `Load`，确保完整的内存可见性。
6. **高性能需求**：在高并发场景下，原子操作的性能明显优于锁。

---

44. Mutexes (互斥锁)

https://gobyexample.com/mutexes

44.1 核心概念

在前面的例子中，我们学习了如何使用原子操作来管理简单的、数字类型的共享状态。当共享的状态变得更加复杂时（例如，一个结构体或一个映射），我们就需要一种更强大的同步机制，这就是**互斥锁 (Mutex)**。

互斥锁用于在多个 Go 协程 (goroutine) 之间安全地访问共享数据，确保在任何时刻只有一个协程可以进入代码的“临界区”，从而防止数据竞争 (data race)。

生活化类比：想象一个只有一个座位的图书馆阅览室。互斥锁就像是这个阅览室的门锁。一个学生进去后，会把门锁上（`Lock`），用完后出来时再把门打开（`Unlock`）。这样就保证了任何时候都只有一个人在里面，避免了混乱。

44.2 互斥锁的基本用法

44.2.1 定义一个需要同步的结构体

为了保护一个共享的数据结构，我们通常会将 `sync.Mutex` 直接嵌入到该结构体中。

```go
// Container 包含一个我们希望并发访问的 map
// 我们在结构体中加入一个 Mutex 来同步对 map 的访问
type Container struct {
    mu       sync.Mutex
    counters map[string]int
}
```

注：互斥锁不能被复制。如果一个包含互斥锁的结构体需要被传递，应该使用指针（`*Container`）来传递，以确保所有协程共享的是同一个锁实例。

44.2.2 创建加锁的修改方法

在任何需要访问共享数据的方法中，我们必须先获取锁，操作完成后再释放锁。

```go
func (c *Container) inc(name string) {
    // 在访问 counters 之前，先锁定互斥锁
    c.mu.Lock()
    // 使用 defer 语句确保在函数退出时，互斥锁一定会被解锁
    // 这是 Go 中最常用、最安全的模式
    defer c.mu.Unlock()

    // 现在可以安全地访问共享数据了
    c.counters[name]++
}
```

关键模式：`c.mu.Lock()` 和 `defer c.mu.Unlock()` 的组合是 Go 并发编程中的一个核心模式。`defer` 保证了即使函数发生 `panic`，锁也会被正确释放，从而避免死锁。

44.2.3 并发访问

在主程序中，我们可以启动多个协程来并发地调用加锁后的方法。

```go
// 创建一个 WaitGroup 来等待所有协程完成
var wg sync.WaitGroup

// 启动多个协程并发地增加计数器
wg.Add(3)
go doIncrement("a", 10000)
go doIncrement("b", 10000)
go doIncrement("a", 10000)

// 等待所有协程执行完毕
wg.Wait()
```

由于 `inc` 方法被互斥锁保护，即使有多个协程同时调用它，对 `counters` map 的写操作也是串行发生的，保证了数据的最终一致性。

44.3 完整代码示例

```go
package main

import (
    "fmt"
    "sync"
)

// Container 持有一个计数器的 map；因为我们想从多个 Go 协程并发地更新它，
// 所以我们添加一个 Mutex 来同步访问。
// 注意，互斥锁不能被复制，所以如果这个结构体需要被传递，
// 应该通过指针来完成。
type Container struct {
    mu       sync.Mutex
    counters map[string]int
}

// inc 方法在访问 counters 之前锁定互斥锁，
// 并在函数结束时使用 defer 语句来解锁。
func (c *Container) inc(name string) {
    c.mu.Lock()
    defer c.mu.Unlock()
    c.counters[name]++
}

func main() {
    // 注意，互斥锁的零值可以直接使用，所以这里不需要初始化。
    c := Container{
        counters: map[string]int{"a": 0, "b": 0},
    }

    var wg sync.WaitGroup

    // doIncrement 函数在一个循环中增加一个指定名称的计数器。
    doIncrement := func(name string, n int) {
        for i := 0; i < n; i++ {
            c.inc(name)
        }
        wg.Done()
    }

    // 我们启动多个 Go 协程来并发地增加计数器。
    wg.Add(3)
    go doIncrement("a", 10000)
    go doIncrement("b", 10000)
    go doIncrement("a", 10000)

    // 等待所有 Go 协程完成。
    wg.Wait()
    fmt.Println(c.counters)
}
```

44.4 运行结果

```
$ go run mutexes.go
map[a:20000 b:10000]
```

44.5 关键要点

1. **保护复杂状态**：互斥锁是保护复杂共享状态（如 map、struct）的标准方法。
2. **`sync.Mutex`**：Go 提供了 `sync.Mutex` 类型和它的两个核心方法：`Lock()` 和 `Unlock()`。
3. **`defer` 解锁**：始终使用 `defer mu.Unlock()` 来确保锁在函数退出时被释放，这是防止死锁的最佳实践。
4. **零值可用**：互斥锁的零值是未锁定状态，可以直接使用，无需显式初始化。
5. **指针传递**：包含互斥锁的结构体应通过指针传递，以避免复制锁。
6. **性能成本**：与原子操作相比，互斥锁的开销更大，因为它涉及到操作系统层面的线程调度。应只在必要时使用。

---

45. Stateful Goroutines (有状态的协程)

https://gobyexample.com/stateful-goroutines

45.1 核心概念

这是另一种在 Go 中管理共享状态的方法，也是一种非常符合 Go 设计哲学的模式。与使用互斥锁（Mutex）在多个 Goroutine 间"共享内存"不同，这种方法的核心是**"通过通信来共享内存"**。

具体做法是：将共享的状态（例如一个 map）完全封装在一个独立的 Goroutine 中。任何其他需要访问这份状态的 Goroutine，都不能直接读写，而是必须通过一个 channel 发送"请求消息"给这个"状态管理员" Goroutine。这个管理员 Goroutine 依次处理收到的请求，从而保证了所有访问都是串行和安全的，自然地避免了数据竞争。

生活化类比：想象一下，互斥锁模式像是大家共用一个会议室，但需要抢同一把钥匙。而"有状态的 Goroutine"模式则更像是在银行办理业务：

- **共享状态**：银行金库里的钱（`state` map）。
- **状态管理员 Goroutine**：银行里唯一的业务柜员。只有他能接触到金库。
- **其他 Goroutines**：来办理业务的客户。
- **Channels**：客户用来提交业务单（`reads` 和 `writes` 通道）和接收回执（`resp` 通道）的窗口。

客户（其他 Goroutine）不能自己去金库拿钱，而是要填一张"取款单"或"存款单"（`readOp` 或 `writeOp`），通过窗口（channel）递给柜员。柜员（状态管理员）一个一个地处理这些单子，处理完后把钱或回执通过另一个窗口（`resp` channel）递还给客户。这样，金库的操作永远是安全、有序的。

45.2 代码分解解释

45.2.1 定义"业务单" (Read/Write Operations)

```
type readOp struct {
    key  int
    resp chan int
}
type writeOp struct {
    key  int
    val  int
    resp chan bool
}
```

- 我们定义了两种“业务单”结构体：`readOp` 用于读请求，`writeOp` 用于写请求。
- **关键设计**：每张“业务单”里都包含一个 `resp` channel。这是一个**回传通道**，专门用于柜员（状态管理员）将处理结果（读取到的值或写入成功的确认）返回给对应的客户（请求方 Goroutine）。

45.2.2 “柜员” Goroutine (The State Owner)

```
go func() {
    var state = make(map[int]int) // 金库 (state) 只在这里可见
    for {
        select {
        case read := <-reads:
            read.resp <- state[read.key] // 处理读请求，并通过回传通道返回值
        case write := <-writes:
            state[write.key] = write.val // 处理写请求
            write.resp <- true           // 通过回传通道发送确认
        }
    }
}()
```

- 这个 Goroutine 是唯一一个能直接访问 `state` map 的地方。
- 它在一个无限循环 `for {}` 中，使用 `select` 等待来自 `reads` 或 `writes` 通道的“业务单”。
- 收到单子后，它执行相应的操作，然后把结果塞回单子里的 `resp` 通道。整个过程是串行的，一次只处理一个请求。

45.2.3 “客户” Goroutines (The Requesters)

```
// 读请求的客户
go func() {
    for {
        read := readOp{
            key:  rand.Intn(5),
            resp: make(chan int), // 为这次请求创建一个专属的回传通道
        }
        reads <- read // 提交“读”业务单
        <-read.resp   // 等待柜员通过回传通道给回复
        // ...
    }
}()
```

- 每个“客户”（读或写的 Goroutine）在发起请求时，都会创建一个**新的、专属于本次请求的**回传通道 `resp`。
- 它把填好的“业务单”（包含这个回传通道）发送给“柜员”。
- 然后它就在自己的回传通道 `<-read.resp` 上**阻塞等待**，直到“柜员”处理完并把结果发回来。

45.3 完整代码示例

```
package main

import (
    "fmt"
    "math/rand"
    "sync/atomic"
    "time"
)

// readOp 和 writeOp 结构体封装了读和写操作，
// 但更重要的是，它们都包含一个用于响应的通道 resp。
type readOp struct {
    key  int
    resp chan int
}
type writeOp struct {
    key  int
    val  int
    resp chan bool
}

func main() {
    // ops 将记录我们执行的总操作数。
    var readOps uint64
    var writeOps uint64

    // reads 和 writes 通道分别用于发起读和写请求。
    reads := make(chan readOp)
    writes := make(chan writeOp)

    // 这是拥有状态的 goroutine，它独占 state map。
    // 这个 goroutine 反复从 reads 和 writes 通道中选择请求并执行。
    go func() {
        var state = make(map[int]int)
        for {
            select {
            case read := <-reads:
                read.resp <- state[read.key]
            case write := <-writes:
                state[write.key] = write.val
                write.resp <- true
            }
        }
    }()
```

```
    // 启动 100 个 goroutine 来模拟并发读取。
    // 每个读取请求都构造一个 readOp，发送到 reads 通道，
    // 并通过其 resp 通道接收结果。
    for r := 0; r < 100; r++ {
        go func() {
            for {
                read := readOp{
                    key:  rand.Intn(5),
                    resp: make(chan int),
                }
                reads <- read
                <-read.resp
                atomic.AddUint64(&readOps, 1)
                time.Sleep(time.Millisecond)
            }
        }()
    }

    // 同样，我们启动 10 个 goroutine 来模拟并发写入。
    for w := 0; w < 10; w++ {
        go func() {
            for {
                write := writeOp{
                    key:  rand.Intn(5),
                    val:  rand.Intn(100),
                    resp: make(chan bool),
                }
                writes <- write
                <-write.resp
                atomic.AddUint64(&writeOps, 1)
                time.Sleep(time.Millisecond)
            }
        }()
    }

    // 让 goroutine 运行一秒钟。
    time.Sleep(time.Second)

    // 最后，获取并报告总操作数。
    readOpsFinal := atomic.LoadUint64(&readOps)
    fmt.Println("readOps:", readOpsFinal)
    writeOpsFinal := atomic.LoadUint64(&writeOps)
    fmt.Println("writeOps:", writeOpsFinal)
}
```

45.4 运行结果

```
$ go run stateful-goroutines.go
readOps: 82013
writeOps: 8169
```

(注意：由于并发调度的随机性，你每次运行的结果都会略有不同)

45.5 关键要点

1. **单一所有权**：这是此模式的核心。共享状态由且仅由一个 Goroutine拥有和管理，从根本上消除了数据竞争。
2. **通信代替锁定**：完全不使用 `Mutex`。所有同步都是通过 channel 的阻塞特性自然实现的。
3. **请求与回传**：每个请求都是一个包含了“回传地址”（`resp` channel）的消息，实现了请求-响应模式。
4. **串行化访问**：“状态管理员”的 `select` 循环将并发的请求转换为了串行的、一次一个的安全操作。
5. **何时使用**：当你的并发逻辑比简单的计数器更复杂，特别是当状态的更新逻辑本身还涉及到其他 channel 或复杂的协调时，这种模式比管理多个互斥锁要更清晰、更不容易出错。

---

46. Sorting (排序)

https://gobyexample.com/sorting

46.1 核心概念

Go 的 `slices` 包（在 Go 1.21 之前是 `sort` 包）提供了对切片进行排序的功能。对于常见的内置类型（如 `int`, `float64`, `string`），Go 提供了开箱即用的排序函数。这些函数会**就地 (in-place)** 对切片进行排序，也就是说，它们会直接修改原始切片，而不是返回一个排好序的新切片。

生活化类比：`slices.Sort` 就像一个能自动整理书架的机器人。你有一排乱七八糟的书（一个无序的切片），启动机器人后，它会直接在**原地**把这些书按照首字母顺序或高度排列好（就地排序），而不是给你复制一套新的、排好序的书。

46.2 基本排序用法

46.2.1 排序字符串切片

```
strs := []string{"c", "a", "b"}
slices.Sort(strs)
fmt.Println("Strings:", strs)
```

- `slices.Sort()` 函数会按照字典序对字符串切片进行升序排序。
- 排序是就地的，所以 `strs` 本身的内容被改变了。

46.2.2 排序整数切片

```
ints := []int{7, 2, 4}
slices.Sort(ints)
fmt.Println("Ints:   ", ints)
```

- 对于整数切片，`slices.Sort()` 会按数值大小进行升序排序。

46.2.3 检查是否已排序

```
s := slices.IsSorted(ints)
fmt.Println("Sorted: ", s)
```

- `slices.IsSorted()` 是一个方便的工具函数，用于检查一个切片是否已经处于有序状态，它会返回一个布尔值。

46.3 完整代码示例

```
package main

import (
    "fmt"
    "slices"
)

func main() {

    // 创建一个字符串切片
    strs := []string{"c", "a", "b"}
    // 使用 slices.Sort 对其进行原地排序
    slices.Sort(strs)
    fmt.Println("Strings:", strs)

    // 创建一个整数切片
    ints := []int{7, 2, 4}
    // 同样使用 slices.Sort 进行排序
    slices.Sort(ints)
    fmt.Println("Ints:   ", ints)

    // 我们也可以检查一个切片是否已经排好序
    s := slices.IsSorted(ints)
    fmt.Println("Sorted: ", s)
}
```

46.4 运行结果

```
$ go run sorting.go
Strings: [a b c]
Ints:    [2 4 7]
Sorted:  true
```

46.5 关键要点

1. **就地排序**：Go 的标准排序函数会直接修改你传入的切片，而不是创建一个新的。
2. **`slices` 包**：在现代 Go 版本 (1.21+) 中，推荐使用 `slices` 包提供的函数，它比旧的 `sort` 包更通用和易用。
3. **内置类型支持**：对于 `string`、`int`、`float64` 等常见类型，都有直接的排序函数。

4. **排序检查**：`slices.IsSorted()` 可以方便地验证切片是否有序。
5. **自定义排序**：对于自定义的结构体切片，Go 也支持通过提供自定义的比较函数（使用 `slices.SortFunc`）或实现 `sort.Interface` 接口来进行排序（这在更高级的例子中会看到）。

---

47. Sorting by Functions (自定义函数排序)

https://gobyexample.com/sorting-by-functions

47.1 核心概念

有时，我们希望根据数据结构中字段的自然顺序以外的方式进行排序。例如，我们可能想按字符串的长度而不是按字母顺序对字符串进行排序。在 Go 中，可以通过向 `slices.SortFunc` 函数提供一个自定义的**比较函数 (comparison function)** 来轻松实现这一点。

这个自定义函数告诉 `SortFunc` 如何判断两个元素哪个应该排在前面，从而实现任意逻辑的排序。

生活化类比：如果说 `slices.Sort` 是一个只会按字母顺序整理书架的机器人，那么 `slices.SortFunc` 就是一个可编程的、更聪明的机器人。你可以给它一张“规则卡片”（自定义的比较函数），上面写着：“这次不按书名，按书的厚度来排”或者“按出版年份来排”。机器人会严格按照你的这张卡片来决定任意两本书的先后顺序，从而实现任何你想要的排序方式。

47.2 自定义排序的用法

47.2.1 定义一个比较函数

假设我们想按长度对一个字符串切片进行排序。我们需要一个函数，它能比较两个字符串 `a` 和 `b` 的长度。

```
// lenCmp 是一个比较函数，它比较两个字符串的长度
// cmp.Compare 会返回 -1 (a<b), 0 (a==b), 或 +1 (a>b)
lenCmp := func(a, b string) int {
    return cmp.Compare(len(a), len(b))
}
```

- 这个函数 `lenCmp` 接受两个待比较的元素（这里是字符串 `a` 和 `b`）。
- 它返回一个 `int`，这个整数的含义是：
    - 如果为负数，表示 `a` 应该排在 `b` 前面。
    - 如果为零，表示 `a` 和 `b` 的顺序无所谓。
    - 如果为正数，表示 `a` 应该排在 `b` 后面。
- `cmp.Compare` 是 Go 1.21+ 引入的辅助函数，它能方便地比较两个可排序的值并返回 `-1, 0, 1`。

47.2.2 使用 `slices.SortFunc`

定义好比较函数后，我们就可以把它和要排序的切片一起传给 `slices.SortFunc`。

```
fruits := []string{"peach", "banana", "kiwi"}
slices.SortFunc(fruits, lenCmp)
fmt.Println(fruits) // 输出: [kiwi peach banana]
```

- `slices.SortFunc` 会使用我们提供的 `lenCmp` 函数来两两比较 `fruits` 切片中的元素，并最终将切片就地排序。

47.2.3 对结构体切片排序

这种模式对于自定义结构体切片尤其有用。

```
type Person struct {
    name string
    age  int
}
people := []Person{ ... }

// 直接在 SortFunc 中使用一个匿名函数按年龄排序
slices.SortFunc(people, func(a, b Person) int {
    return cmp.Compare(a.age, b.age)
})
```

- 这里我们甚至不需要预先定义一个具名的比较函数，直接在调用时传入一个匿名函数即可，代码非常紧凑。

47.3 完整代码示例

```
package main

import (
    "cmp"
    "fmt"
    "slices"
)

func main() {
    fruits := []string{"peach", "banana", "kiwi"}

    // 定义一个比较函数，用于按字符串长度排序
    lenCmp := func(a, b string) int {
        return cmp.Compare(len(a), len(b))
    }

    // 使用 slices.SortFunc 和我们自定义的比较函数进行排序
    slices.SortFunc(fruits, lenCmp)
    fmt.Println(fruits)

    // --- 另一个例子：使用结构体切片 ---
    type Person struct {
        name string
        age  int
```

```
    }

    people := []Person{
        {"Alice", 30},
        {"Bob", 25},
        {"Charlie", 35},
    }

    // 按年龄排序
    slices.SortFunc(people, func(a, b Person) int {
        return cmp.Compare(a.age, b.age)
    })

    fmt.Println(people)
}
```

47.4 运行结果

```
$ go run sorting-by-functions.go
[kiwi peach banana]
[{Bob 25} {Alice 30} {Charlie 35}]
```

47.5 关键要点

1. **`slices.SortFunc` 是关键**：当你需要自定义排序逻辑时，这个函数是首选。
2. **提供比较逻辑**：你需要提供一个函数，告诉 Go 如何比较切片中的任意两个元素。
3. **`cmp.Compare` 辅助函数**：对于可比较的类型（数字、字符串等），使用 `cmp.Compare` 可以轻松地编写比较函数。
4. **灵活性**：通过改变比较函数，你可以用任何可以想象到的方式对同一个数据集进行排序（例如，按年龄、按姓名、按分数等）。
5. **匿名函数**：对于一次性的排序逻辑，使用匿名函数会让代码更简洁。

---

48. Panic (恐慌)

https://gobyexample.com/panic

48.1 核心概念

`panic` 是 Go 语言中一个内置函数，用于产生一个意外的、无法正常处理的错误。当程序遇到一个无法恢复的错误时，可以调用 `panic` 来中断程序的正常执行流程。`panic` 通常用于表示程序中出现了意料之外的严重问题，例如，逻辑上不应该发生的错误。

与错误处理（`error`）不同，`panic` 是一种更激进的错误处理机制。在 Go 中，惯例是使用 `error` 返回值来处理可预见的错误，而 `panic` 则用于处理那些不应该发生的、灾难性的错误。

48.2 panic 的基本用法

48.2.1 调用 panic

可以直接调用 `panic` 函数，并传入一个任意类型的参数，这个参数将作为 `panic` 的信息被打印出来。

```
panic("a problem")
```

48.2.2 panic 与 defer

当 `panic` 被调用时，它会立即停止当前函数的执行，并开始逐层向上执行当前 goroutine 中的 `defer` 语句。这个过程会一直持续到 goroutine 的顶层，然后程序会崩溃并打印出 `panic` 信息和堆栈跟踪。

```
func main() {
    defer fmt.Println("deferred call in main")
    f()
    fmt.Println("main finished") // 这行不会被执行
}

func f() {
    defer fmt.Println("deferred call in f")
    panic("a problem")
    fmt.Println("f finished") // 这行不会被执行
}
```

48.2.3 panic 与 error 的应用场景对比

为了更好地区分 `panic` 和 `error`，可以想象一个厨房的场景：

- **返回 `error`**：就像发现盐用完了。这是一个**可预见的、可处理的**问题。厨师（调用者）可以决定是去买盐、换一道菜，还是告诉客人今天某道菜做不了。程序不会因此崩溃，而是进入一个备用流程。

  ```
  salt, err := getSalt()
  if err != nil {
      // 盐没了，换个菜谱
      useAlternativeRecipe()
      return // 正常处理，程序继续
  }
  ```

- **调用 `panic`**：就像厨房突然着火了。这是一个**灾难性的、程序不应该继续运行**的状况。此时最重要的事情是立即停止所有烹饪活动（中断正常流程），拉响火警（打印 panic 信息和堆栈），然后所有人赶紧撤离（程序崩溃）。

```
// 假设这是一个关键的初始化步骤
db, err := gorm.Open(mysql.Open(dsn), &gorm.Config{})
if err != nil {
    // 数据库连不上，整个程序的核心功能都无法工作
    // 此时强行运行下去只会导致更多不可预知的错误
    panic("failed to connect database")
}
```

另一个常见的 `panic` 场景是在程序初始化阶段，如果必要的配置或资源加载失败，程序就无法正常工作。此时，与其返回一个 `error` 让顶层调用者去处理一个几乎无法处理的问题，不如直接 `panic`，让问题尽早暴露。

48.3 完整代码示例

```
package main

import "os"

func main() {
    // 我们将使用 panic 来检查一个意料之外的错误。
    // panic 的一个常见用法是，如果一个函数返回了一个我们不知道如何处理（或不想处理）的错误值，就中止程序。
    panic("a problem")

    // 在一个文件中创建了一个文件，但该文件是不存在的。
    // 然后我们使用 panic 来检查这个错误。
    _, err := os.Create("/tmp/file")
    if err != nil {
        panic(err)
    }
}
```

48.4 运行结果

```
$ go run panic.go
panic: a problem

goroutine 1 [running]:
main.main()
    /Users/mac/Documents/person/knowledges/go_learns/go_by_example/panic.go:8 +0x39
exit status 2
```

48.5 关键要点

1. **中断执行**：`panic` 会立即中断当前函数的正常执行流程。
2. **执行 defer**：在 `panic` 发生后，会按后进先出的顺序执行当前 goroutine 中的 `defer` 语句。
3. **程序崩溃**：`defer` 执行完毕后，程序会崩溃，并打印出 `panic` 的信息和堆栈跟踪。
4. **与 error 的区别**：`error` 用于处理可预见的、可恢复的错误；`panic` 用于处理不可恢复的、灾难性的错误。

5. **恢复（Recover）**：Go 提供了 `recover` 函数，可以在 `defer` 语句中捕获 `panic`，阻止程序崩溃并恢复正常执行。这通常用于库代码中，以避免因为内部 `panic` 而导致整个应用程序崩溃。

---

49. Defer (延迟执行)

https://gobyexample.com/defer

49.1 核心概念

`defer` 语句用于安排一个函数调用（延迟调用），使其在执行 `defer` 的函数即将返回之前执行。这个特性常用于执行清理操作，例如关闭文件、解锁互斥锁或关闭网络连接。`defer` 确保了即使函数因为 `panic` 而异常终止，这些清理操作也能够被执行。

生活化类比：
可以把 `defer` 想象成出门前穿衣服和回家后脱衣服的过程，这能很好地解释其“后进先出”（LIFO）的特性。

- **出门（函数执行）**：你先穿上衬衫，然后穿上毛衣 (`defer` 脱毛衣)，最后穿上外套 (`defer` 脱外套)。
- **回家（函数返回）**：你回家的第一件事是脱掉外套（最后被 `defer` 的），然后脱掉毛衣（先被 `defer` 的），最后才脱掉衬衫。
  `defer` 就像是你每穿上一件需要稍后脱下的衣服时，都对自己说：“我保证待会儿回来时会脱掉它”。这些“承诺”会按照相反的顺序被执行。

49.2 Defer 的基本用法与 LIFO 顺序

49.2.1 基本用法

`defer` 关键字后面跟着一个函数调用。这个调用不会立即执行，而是会被推迟。

```
package main

import (
    "fmt"
    "os"
)

func createFile(p string) *os.File {
    fmt.Println("creating")
    f, err := os.Create(p)
    if err != nil {
        panic(err)
    }
    return f
}

func writeFile(f *os.File) {
    fmt.Println("writing")
    fmt.Fprintln(f, "data")
}

// closeFile 函数被延迟执行
```

```go
func closeFile(f *os.File) {
    fmt.Println("closing")
    err := f.Close()
    if err != nil {
        fmt.Fprintf(os.Stderr, "error: %v", err)
        os.Exit(1)
    }
}

func main() {
    f := createFile("/tmp/defer.txt")
    // defer 语句安排 closeFile 在 main 函数结束时执行
    defer closeFile(f)
    writeFile(f)
}
```

说明：`defer closeFile(f)` 安排了 `closeFile` 函数的调用。这个调用将在 `main` 函数的末尾，即 `writeFile` 执行完毕后执行。

49.2.2 LIFO (后进先出) 顺序

当有多个 `defer` 语句时，它们会被添加到一个栈中。当函数返回时，这些延迟调用会以“后进先出”（LIFO）的顺序执行。

```go
func lifoExample() {
    fmt.Println("start")
    defer fmt.Println("1")
    defer fmt.Println("2")
    defer fmt.Println("3")
    fmt.Println("end")
}
// 输出:
// start
// end
// 3
// 2
// 1
```

关键特性：最后被 `defer` 的调用最先执行。

49.3 完整代码示例

```go
package main

import (
    "fmt"
    "os"
)

// 创建一个文件并返回
func createFile(p string) *os.File {
```

```
    fmt.Println("creating")
    f, err := os.Create(p)
    if err != nil {
        panic(err)
    }
    return f
}

// 向文件中写入数据
func writeFile(f *os.File) {
    fmt.Println("writing")
    fmt.Fprintln(f, "data")
}

// 关闭文件，这个函数将被延迟执行
func closeFile(f *os.File) {
    fmt.Println("closing")
    // 在关闭文件时检查并处理错误
    err := f.Close()
    if err != nil {
        fmt.Fprintf(os.Stderr, "error: %v
", err)
        os.Exit(1)
    }
}

func main() {
    // 创建文件
    f := createFile("/tmp/defer.txt")
    // 使用 defer 安排文件关闭操作
    // 这将在 main 函数的末尾执行
    defer closeFile(f)
    // 向文件写入数据
    writeFile(f)
}
```

49.4 运行结果

执行程序后，你会看到函数按照 `creating` -> `writing` -> `closing` 的顺序执行。

```
$ go run defer.go
creating
writing
closing
```

你可以通过检查文件内容来确认写入是否成功。

```
$ cat /tmp/defer.txt
data
```

49.5 关键要点

1. **确保执行**：`defer` 确保函数调用在包含它的函数返回前执行，无论是正常返回还是发生 `panic`。
2. **LIFO 顺序**：多个 `defer` 调用按后进先出（LIFO）的顺序执行。
3. **清理任务**：`defer` 是执行清理任务（如关闭资源）的理想选择，因为它将打开和关闭资源的代码放在一起，提高了代码的可读性。
4. **参数立即求值**：`defer` 语句的函数参数在 `defer` 声明时就会被求值，而不是在函数实际执行时。
5. **与 `panic` 结合**：`defer` 在 `panic/recover` 机制中扮演关键角色，允许程序从 `panic` 中恢复。

---

50. Recover (恢复)

https://gobyexample.com/recover

50.1 核心概念

Go 提供了 `recover` 内置函数，用于从 `panic` 中恢复。`recover` 可以阻止 `panic` 中止整个程序，并允许程序继续执行。这就像是为可能出现意外的代码装上一个“安全网”。

一个典型的应用场景是 Web 服务器。如果某个客户端连接引发了严重错误导致 `panic`，服务器不应该因此崩溃，而是应该关闭这个出问题的连接，并继续为其他客户端提供服务。

生活化类比：
`panic` 就像是高空走钢丝的演员突然失足。如果在表演前（`defer`）预先在下方设置了安全网（`recover`），那么演员掉下来时会被接住，表演可以继续（或者至少能安全退场），而不会导致整个马戏团关门。如果没有安全网，一次失足就会导致灾难性的后果。

50.2 `panic` 与 `recover` 的交互

`recover` 只有在 `defer` 修饰的函数中调用时才有效。在正常的执行流程中调用 `recover` 会返回 `nil`，并且没有任何其他效果。如果当前 goroutine 正在 `panic`，调用 `recover` 会捕获 `panic` 的值，并恢复正常的执行。

50.2.1 `defer` 中的 `recover`

```
defer func() {
    // recover() 用于捕获 panic
    if r := recover(); r != nil {
        // r 是 panic 传入的值
        fmt.Println("Recovered. Error:\n", r)
    }
}()
```

关键逻辑：

1. `defer` 语句确保其后的函数在当前函数返回前（包括发生 `panic` 时）执行。
2. `recover()` 检查当前 goroutine 是否处于 `panic` 状态。

3. 如果是，`recover()` 会捕获 `panic` 的值，并阻止 `panic` 继续向上传播。程序流程会跳转到 `defer` 函数的末尾，然后正常返回。
4. 如果不是，`recover()` 返回 `nil`。

50.2.2 `panic` 后的执行流程

当 `panic` 发生时，函数的执行会立即停止，但 `defer` 的函数会按后进先出的顺序执行。如果在 `defer` 中成功 `recover`，执行将在 `defer` 结束后从调用 `defer` 的函数返回点继续，但 `panic` 点之后的代码不会被执行。

```
mayPanic() // 发生 panic
// 这一行代码永远不会被执行
fmt.Println("After mayPanic()")
```

50.3 完整代码示例

```
package main

import "fmt"

// 一个可能引发 panic 的函数
func mayPanic() {
    panic("a problem")
}

func main() {
    // 必须在 defer 函数中调用 recover
    defer func() {
        if r := recover(); r != nil {
            // r 是 panic 传入的值
            fmt.Println("Recovered. Error:\n", r)
        }
    }()

    mayPanic()

    // 这行代码不会被执行，因为 mayPanic 发生 panic 后，
    // 执行会跳转到 defer 的函数，并在 defer 执行完毕后直接返回。
    fmt.Println("After mayPanic()")
}
```

50.4 运行结果

```
$ go run recover.go
Recovered. Error:
 a problem
```

注意：`After mayPanic()` 没有被打印出来，因为 `recover` 虽然阻止了程序崩溃，但执行流程在 `defer` 之后就结束了，不会回到 `main` 函数中 `panic` 发生点之后的位置。

50.5 关键要点

1. **`recover` 必须在 `defer` 中使用**：只有在 `defer` 函数中调用 `recover` 才能捕获 `panic`。
2. **阻止程序崩溃**：`recover` 用于从 `panic` 中恢复，避免程序因意外错误而完全终止。
3. **捕获 `panic` 值**：`recover` 返回传递给 `panic` 的值，如果当前没有 `panic`，则返回 `nil`。
4. **中断正常流程**：`panic` 会立即停止当前函数的执行，并开始执行 `defer` 栈。
5. **恢复点**：成功 `recover` 后，执行从 `defer` 所在的函数返回，而不会回到 `panic` 发生点。
6. **适用场景**：主要用于保护关键程序（如服务器）不被意外的 `panic` 击垮，或者用于清理资源后向上层报告一个普通错误。

---

51. String Functions (字符串函数)

https://gobyexample.com/string-functions

51.1 核心概念

Go 的 `strings` 标准库包提供了大量用于处理字符串的实用函数。这些函数涵盖了搜索、替换、比较、修剪、拆分和连接等多种操作。由于 Go 的字符串是 UTF-8 编码的，`strings` 包中的函数能够正确处理多字节字符。

生活化类比：`strings` 包就像一个功能齐全的文本编辑工具箱。里面有各种工具：`Contains` 像放大镜，帮你检查一个词是否存在；`Split` 像一把剪刀，能把一句话按空格剪成多个单词；`Join` 像胶水，能把多个单词粘成一句话；`Replace` 则像涂改液和笔，能把错字替换成正确的字。

51.2 常用字符串函数概览

以下是 `strings` 包中一些最常用函数的介绍。

51.2.1 搜索与检查

- **`Contains(s, substr)`**: 检查字符串 `s` 是否包含子串 `substr`。
- **`Count(s, substr)`**: 计算字符串 `s` 中子串 `substr` 出现的次数。
- **`HasPrefix(s, prefix)`**: 检查字符串 `s` 是否以 `prefix` 开头。
- **`HasSuffix(s, suffix)`**: 检查字符串 `s` 是否以 `suffix` 结尾。
- **`Index(s, substr)`**: 返回子串 `substr` 在字符串 `s` 中首次出现的位置索引，如果不存在则返回 -1。

51.2.2 修改与转换

- **`Join(a []string, sep string)`**: 将一个字符串切片 `a` 的所有元素用分隔符 `sep` 连接成一个单一的字符串。
- **`Repeat(s string, count int)`**: 将字符串 `s` 重复 `count` 次并返回新字符串。
- **`Replace(s, old, new, n)`**: 在字符串 `s` 中，将旧子串 `old` 替换为新子串 `new`。`n` 控制替换次数，如果 `n < 0` 则替换所有出现的 `old`。
- **`ToLower(s)`**: 将字符串 `s` 中的所有字符转换为小写。
- **`ToUpper(s)`**: 将字符串 `s` 中的所有字符转换为大写。

51.2.3 拆分

- `Split(s, sep)`: 将字符串 `s` 按照分隔符 `sep` 拆分成一个字符串切片。

51.3 完整代码示例

这个示例集中演示了 `strings` 包中多个函数的用法。

```
package main

import (
    "fmt"
    "strings"
)

// 为 fmt.Println 创建一个别名，方便使用
var p = fmt.Println

func main() {
    // 检查、计数、前后缀判断
    p("Contains:  ", strings.Contains("test", "es"))
    p("Count:     ", strings.Count("test", "t"))
    p("HasPrefix: ", strings.HasPrefix("test", "te"))
    p("HasSuffix: ", strings.HasSuffix("test", "st"))

    // 查找索引
    p("Index:     ", strings.Index("test", "e"))

    // 连接与重复
    p("Join:      ", strings.Join([]string{"a", "b"}, "-"))
    p("Repeat:    ", strings.Repeat("a", 5))

    // 替换
    p("Replace:   ", strings.Replace("foo", "o", "0", -1)) // 全部替换
    p("Replace:   ", strings.Replace("foo", "o", "0", 1))  // 只替换1次

    // 拆分与大小写转换
    p("Split:     ", strings.Split("a-b-c-d-e", "-"))
    p("ToLower:   ", strings.ToLower("TEST"))
    p("ToUpper:   ", strings.ToUpper("test"))
}
```

51.4 运行结果

```
$ go run string-functions.go
Contains:  true
Count:     2
HasPrefix: true
HasSuffix: true
Index:     1
Join:      a-b
Repeat:    aaaaa
Replace:   f00
Replace:   f0o
Split:     [a b c d e]
ToLower:   test
ToUpper:   TEST
```

51.5 关键要点

1. **功能丰富**: `strings` 包提供了全面的字符串操作函数，是处理文本的基础工具。
2. **UTF-8 安全**: 这些函数能正确处理 Unicode 字符，而不仅仅是单字节的 ASCII 字符。
3. **非原地修改**: 字符串在 Go 中是不可变的。所有 `strings` 包中看似修改字符串的函数（如 `Replace`, `ToLower`）实际上都会返回一个新的字符串，而不会改变原始字符串。
4. **易用性**: 函数命名直观，易于理解和使用，例如 `HasPrefix` 和 `Contains`。
5. **性能**: `strings` 包中的函数经过优化，通常比手动实现相同逻辑更高效。
6. **与 `strconv` 的区别**: `strings` 包主要用于字符串的操作和处理，而基本数据类型（如 `int`, `float`）与字符串之间的转换则由 `strconv` 包负责。

---

52. String Formatting (字符串格式化)

https://gobyexample.com/string-formatting

52.1 核心概念

Go 的 `fmt` 包提供了丰富的函数，用于根据格式说明符（verbs）来格式化字符串，类似于 C 语言的 `printf`。这使得我们可以将不同类型的值（如整数、浮点数、字符串等）优雅地嵌入到文本中。

- **`fmt.Printf`**: 将格式化后的字符串直接打印到标准输出。
- **`fmt.Sprintf`**: 返回一个格式化后的字符串，而不打印它。这是最常用的，便于将结果赋值给变量或用于其他目的。
- **`fmt.Fprintf`**: 将格式化后的字符串写入任何实现了 `io.Writer` 接口的对象，如文件或网络连接。

生活化类比: `fmt.Printf` 就像一个公告员，直接对着大厅（控制台）喊话。`fmt.Sprintf` 则像一个秘书，把要说的话写在一张纸条上递给你，你可以自己决定怎么用这张纸条。`fmt.Fprintf` 则更像一个邮差，可以把信（格式化内容）投递到指定的邮箱（文件、`os.Stderr` 等）。

52.2 常用格式化谓词 (Verbs)

格式化谓词是 `%` 后面跟一个字符，用于指定如何格式化对应的参数。

52.2.1 通用谓词

- `%v`: 按值的默认格式输出。
- `%+v`: 当值为结构体时，会额外输出字段名。
- `%#v`: 输出值的 Go 语法表示，例如字符串会带上双引号。
- `%T`: 输出值的类型。

52.2.2 特定类型谓词

- **布尔值**:
  - `%t`: 输出 `true` 或 `false`。
- **整数**:
  - `%d`: 十进制表示。
  - `%b`: 二进制表示。
  - `%c`: 对应的 Unicode 字符。
  - `%x`: 十六进制表示（小写字母）。
- **浮点数**:
  - `%f` 或 `%F`: 标准浮点表示。
  - `%e` 或 `%E`: 科学计数法。
- **字符串**:
  - `%s`: 直接输出字符串内容。
  - `%q`: 为字符串加上双引号，并转义特殊字符。
- **指针**:
  - `%p`: 十六进制表示，前缀为 `0x`。

52.2.3 宽度与精度控制

可以在 `%` 和谓词之间加入数字来控制宽度和精度。

- `%-10s`: 宽度为 10，左对齐。
- `%10.2f`: 总宽度为 10，保留 2 位小数。

52.3 完整代码示例

```
package main

import (
    "fmt"
    "os"
)

type point struct {
    x, y int
```

```
}

func main() {
    p := point{1, 2}

    // %v 打印结构体的值
    fmt.Printf("p = %v\n", p)

    // %+v 打印结构体字段名
    fmt.Printf("p = %+v\n", p)

    // %#v 打印 Go 语法表示
    fmt.Printf("p = %#v\n", p)

    // %T 打印类型
    fmt.Printf("Type of p: %T\n", p)

    // %t 打印布尔值
    fmt.Printf("Is p valid? %t\n", true)

    // %d 打印整数
    fmt.Printf("p.x = %d\n", p.x)

    // %b 打印二进制
    fmt.Printf("7 in binary: %b\n", 7)

    // %s 打印字符串
    fmt.Printf("Hello, %s\n", "Go")

    // %q 打印带引号的字符串
    fmt.Printf("Quoted string: %q\n", "Hello, Go")

    // %f 打印浮点数
    fmt.Printf("Pi is approx %f\n", 3.14159)

    // 控制宽度和精度
    fmt.Printf("Pi with precision: %.3f\n", 3.14159)
    fmt.Printf("Right-aligned: |%6s|\n", "foo")
    fmt.Printf("Left-aligned:  |%-6s|\n", "foo")

    // Sprintf 返回字符串
    s := fmt.Sprintf("a %s", "string")
    fmt.Println(s)

    // Fprintf 写入 io.Writer
    fmt.Fprintf(os.Stderr, "An error occurred: %s\n", "file not found")
}
```

52.4 运行结果

```
$ go run string-formatting.go
```

```
p = {1 2}
p = {x:1 y:2}
p = main.point{x:1, y:2}
Type of p: main.point
Is p valid? true
p.x = 1
7 in binary: 111
Hello, Go
Quoted string: "Hello, Go"
Pi is approx 3.141590
Pi with precision: 3.142
Right-aligned: |    foo|
Left-aligned:  |foo    |
a string
An error occurred: file not found
```

52.5 关键要点

1. **`fmt` 包是核心**：`Printf`，`Sprintf`，`Fprintf` 是实现格式化输出的三大主力。
2. **`%v` 是万金油**：当你不确定用哪个谓词时，`%v` 通常能给出合理的默认输出。
3. **`%+v` 和 `%#v` 用于调试**：它们能提供更详细的结构体和值的表示，便于调试。
4. **类型特定谓词**：使用 `%d`，`%s`，`%f` 等可以更精确地控制输出格式。
5. **宽度和精度控制**：通过在谓词中加入数字，可以实现对齐和精度控制，使输出更美观。
6. **`Sprintf` 用于生成字符串**：当你需要将格式化结果存入变量而不是直接打印时，`Sprintf` 是不二之选。

---

53. Text Templates (文本模板)

https://gobyexample.com/text-templates

53.1 核心概念

Go 的 `text/template` 包为创建动态内容和自定义输出提供了内置支持。它允许将静态文本与包裹在 `{{...}}` 中的“动作”混合，以动态插入内容。`html/template` 包提供了相同的 API，但为生成 HTML 增加了额外的安全功能。

生活化类比：把模板想象成一个带有预留空白的信件模板。你可以准备好信件的固定内容（“你好，：”），然后根据不同的收件人，在空白处（`{{.Name}}`）填上具体的名字。模板引擎就是那个帮你自动填写并生成最终信件的工具。

53.2 模板基本用法

53.2.1 创建与解析模板

模板可以从字符串创建和解析。

- `template.New("name").Parse(templateString)`：创建一个新模板并解析模板字符串。

- `template.Must`: 用于包装解析过程，如果发生错误，它会引发 panic。这在全局作用域初始化模板时很方便。

53.2.2 执行模板

使用 `template.Execute(writer, data)` 来执行模板，将解析后的模板与数据结合，并将结果写入一个 `io.Writer`（如 `os.Stdout`）。

53.2.3 插入数据 (Actions)

- `{{.}}`: 这是最基本的动作，它会插入传递给模板的当前数据。
- `{{.FieldName}}`: 当数据是结构体时，用于访问其导出的字段。
- `{{.KeyName}}`: 当数据是映射时，用于访问键对应的值。

53.2.4 条件判断

使用 `{{if .Value}}...{{else}}...{{end}}` 来实现条件逻辑。如果 `.Value` 是其类型的零值（如 `false`, `0`, `""`, `nil`），则条件为假。

53.2.5 循环 (Range)

使用 `{{range .Items}}...{{end}}` 来遍历切片、数组、映射或通道。在 `range` 块内部，`{{.}}` 指的是当前的迭代项。

53.2.6 空白符控制

在 `{{` 或 `}}` 旁边添加 `-` (例如 `{{- .Value -}}`) 可以去除动作任意一侧的空白符，这对于生成整洁的输出很有用。

53.3 完整代码示例

```
package main

import (
  "os"
  "text/template"
)

func main() {
  // 创建一个简单的模板
  t1 := template.New("t1")
  t1, err := t1.Parse("Value is {{.}}\n")
  if err != nil {
    panic(err)
  }

  // 使用 template.Must 简化错误处理，适用于全局变量
  t1 = template.Must(t1.Parse("Value is {{.}}\n"))

  // 执行模板，传入简单数据
  t1.Execute(os.Stdout, "some string")
```

```
  t1.Execute(os.Stdout, 5)
  t1.Execute(os.Stdout, []string{
    "Go",
    "Rust",
    "C++",
    "C#",
  })

  // 创建一个辅助函数
  Create := func(name, t string) *template.Template {
    return template.Must(template.New(name).Parse(t))
  }

  // 演示 if/else
  t2 := Create("t2",
    "{{if . -}} yes {{else -}} no {{end}}\n")
  t2.Execute(os.Stdout, "not empty")
  t2.Execute(os.Stdout, "")

  // 演示空白符控制
  t3 := Create("t3",
    "{{- if . -}} yes {{else -}} no {{end -}}\n")
  t3.Execute(os.Stdout, "not empty")
  t3.Execute(os.Stdout, "")

  // 演示 range
  t4 := Create("t4",
    "Range: {{range .}}{{.}} {{end}}\n")
  t4.Execute(os.Stdout,
    []string{
      "Go",
      "Rust",
      "C++",
      "C#",
    })
}
```

53.4 运行结果

```
$ go run text-templates.go
Value is some string
Value is 5
Value is [Go Rust C++ C#]
 yes
 no
yes
no
Range: Go Rust C++ C#
```

53.5 关键要点

1. **动态内容生成**：`text/template` 是 Go 中生成动态文本（如配置文件、代码）的强大工具。
2. **数据驱动**：模板的输出由传入的数据驱动，实现了逻辑与表现的分离。
3. **动作 `{{...}}`**：所有动态部分都由 `{{...}}` 包裹的动作来控制，包括值插入、条件和循环。
4. **`html/template` 用于 Web**：在生成 HTML 时，应优先使用 `html/template`，因为它能自动处理上下文并转义数据，防止 XSS 攻击。
5. **空白符控制**：使用 `{{- ... -}}` 可以精细控制输出格式，避免不必要的换行和空格。
6. **`Must` 简化初始化**：`template.Must` 在定义全局模板时非常有用，可以确保模板在编译时就是有效的。

---

54. Regular Expressions (正则表达式)

https://gobyexample.com/regular-expressions

54.1 核心概念

Go 语言通过内置的 `regexp` 包为正则表达式提供了强大的支持。正则表达式是一种用于在文本中进行复杂模式匹配的强大工具。

生活化类比：把正则表达式想象成一种高级的“查找和替换”指令。基础的文本搜索只能找到固定的词语，而正则表达式允许你定义一个模式，比如“查找所有以‘img’开头，以‘.jpg’结尾的单词”或“查找所有看起来像电话号码的数字串”，这使得文本处理变得异常灵活和强大。

54.2 正则表达式基本用法

54.2.1 匹配字符串 (MatchString)

这是最简单的用法，用于判断一个模式是否存在于字符串中。

```
match, _ := regexp.MatchString("p([a-z]+)ch", "peach")
fmt.Println(match) // true
```

说明: `MatchString` 直接使用，无需先创建 `Regexp` 对象，适合一次性的匹配检查。

54.2.2 编译正则表达式 (Compile)

对于需要多次使用的正则表达式，更好的做法是先用 `regexp.Compile` 将其编译成一个 `Regexp` 对象。这可以显著提升性能，因为编译只需进行一次。

```
r, _ := regexp.Compile("p([a-z]+)ch")
```

小贴士: 如果正则表达式是常量，应在程序初始化时（例如在 `init` 函数或全局变量中）使用 `regexp.MustCompile` 进行编译。`MustCompile` 在编译失败时会引发 panic，确保了在程序启动时就能发现无效的正则表达式。

54.2.3 `Regexp` 对象的常用方法

编译后的 `Regexp` 对象 `r` 提供了许多方法：

- `r.MatchString(s)`：功能同 `regexp.MatchString`，但使用已编译的 `Regexp` 对象。
- `r.FindString(s)`：查找模式的第一个匹配项。
- `r.FindStringIndex(s)`：查找第一个匹配项的开始和结束索引。
- `r.FindStringSubmatch(s)`：查找第一个匹配项及其所有子匹配（捕获组）。
- `r.FindStringSubmatchIndex(s)`：查找第一个匹配项及其子匹配的索引。
- `r.FindAllString(s, n)`：查找所有匹配项（最多 `n` 个，如果 `n` 为 -1 则查找全部）。
- `r.ReplaceAllString(s, repl)`：替换所有匹配项为指定字符串。
- `r.ReplaceAllStringFunc(s, fn)`：使用自定义函数对每个匹配项进行替换。

54.3 完整代码示例

```
package main

import (
    "bytes"
    "fmt"
    "regexp"
)

func main() {
    // 测试一个字符串是否符合一个模式
    match, _ := regexp.MatchString("p([a-z]+)ch", "peach")
    fmt.Println(match)

    // 上面我们是直接使用字符串，但对于其他正则任务，
    // 你需要先将正则编译为一个优化的 Regexp 结构体
    r, _ := regexp.Compile("p([a-z]+)ch")

    // 这个结构体有很多方法
    // 这是一个类似于我们前面见到的匹配测试
    fmt.Println(r.MatchString("peach"))

    // 这是查找匹配字符串
    fmt.Println("FindString:", r.FindString("peach punch"))

    // 这个也是查找第一次匹配的字符串，但是返回的是开始和结束位置的索引，
    // 而不是匹配的内容
    fmt.Println("FindStringIndex:", r.FindStringIndex("peach punch"))

    // Submatch 返回完全匹配和局部匹配的字符串
    // 例如，这里会返回 p([a-z]+)ch 和 ([a-z]+) 的信息
    fmt.Println("FindStringSubmatch:", r.FindStringSubmatch("peach punch"))

    // 类似的，这个会返回完全匹配和局部匹配的索引
    fmt.Println("FindStringSubmatchIndex:", r.FindStringSubmatchIndex("peach punch"))

    // 带 All 的这些函数返回所有匹配项，而不仅仅是首次匹配项
    // 例如，下面的例子会返回所有匹配到 `p([a-z]+)ch` 的字符串
```

```
    fmt.Println("FindAllString:", r.FindAllString("peach punch pinch", -1))

    // All 同样可以对应到上面的所有函数
    fmt.Println("FindAllStringSubmatchIndex:",
        r.FindAllStringSubmatchIndex("peach punch pinch", -1))

    // 这个函数提供一个正整数来限制匹配次数
    fmt.Println("FindAllString(n=2):", r.FindAllString("peach punch pinch", 2))

    // 上面的例子中，我们使用了字符串作为参数，
    // 并使用了如 MatchString 之类的方法
    // 我们也可以提供 []byte 类型的参数，并将 String 从函数命中去掉
    fmt.Println("Match:", r.Match([]byte("peach")))

    // 创建正则结构体时，可以使用 MustCompile
    // MustCompile 在编译失败时会 panic，而不是返回一个错误
    // 这使得它在创建全局变量时更安全
    r = regexp.MustCompile("p([a-z]+)ch")
    fmt.Println("Regexp:", r)

    // regexp 包也可以用来替换部分字符串
    fmt.Println("ReplaceAllString:", r.ReplaceAllString("a peach", "<fruit>"))

    // Func 变体允许你用给定的函数来转换匹配到的文本
    in := []byte("a peach")
    out := r.ReplaceAllFunc(in, bytes.ToUpper)
    fmt.Println("ReplaceAllFunc:", string(out))
}
```

54.4 运行结果

```
$ go run regular-expressions.go
true
true
FindString: peach
FindStringIndex: [0 5]
FindStringSubmatch: [peach ea]
FindStringSubmatchIndex: [0 5 1 3]
FindAllString: [peach punch pinch]
FindAllStringSubmatchIndex: [[0 5 1 3] [6 11 7 9] [12 17 13 15]]
FindAllString(n=2): [peach punch]
Match: true
Regexp: p([a-z]+)ch
ReplaceAllString: a <fruit>
ReplaceAllFunc: a PEACH
```

54.5 关键要点

1. **性能考量**：对于需要多次使用的正则表达式，务必使用 `regexp.Compile` 或 `regexp.MustCompile` 进行预编译，以获得最佳性能。

2. **`MustCompile` 的应用**：在定义全局或包级别的正则常量时，`MustCompile` 是首选，它能确保在程序启动时就捕获无效的模式。
3. **子匹配（捕获组）**：通过在模式中使用括号 `()`，可以捕获字符串的特定部分。`FindStringSubmatch` 等方法是提取这些部分的关键。
4. **查找全部 vs. 查找首个**：`Find...` 系列函数只查找第一个匹配项，而 `FindAll...` 系列函数可以查找所有匹配项。
5. **字符串与字节切片**：`regexp` 包为字符串 (`string`) 和字节切片 (`[]byte`) 提供了几乎对称的 API，可以根据数据源灵活选用。
6. **强大的替换功能**：除了简单的字符串替换，`ReplaceAllFunc` 允许传入一个函数来动态生成替换内容，极大地增强了替换操作的灵活性。

55. JSON (JSON 处理)

https://gobyexample.com/json

55.1 核心概念

JSON (JavaScript Object Notation) 是一种轻量级的数据交换格式，易于人阅读和编写，也易于机器解析和生成。在 Go 中，`encoding/json` 包提供了对 JSON 的内置支持，可以轻松地将 Go 的数据结构编码为 JSON 文本，以及将 JSON 文本解码为 Go 的数据结构。

生活化类比：将 Go 的数据结构（如 struct）与 JSON 之间的转换，想象成在说不同语言的人之间进行翻译。Go 的 struct 是程序内部的“母语”，结构清晰、类型安全。当你需要将这些信息发送给外部系统（如一个 Web 前端）时，你需要一个“通用语”，JSON 就是这个通用语。`json.Marshal` 就像一个翻译官，把你的 Go “母语”翻译成 JSON “通用语”；而 `json.Unmarshal` 则是另一个方向的翻译官，把接收到的 JSON “通用语”翻译回你的程序能够理解的 Go “母语”。

55.2 JSON 的基本用法

55.2.1 编码 (Marshal) - 从 Go 结构到 JSON

Go 的 `json.Marshal` 函数用于将 Go 的数据结构（如 struct、map、slice 等）转换为 JSON 字节切片 (`[]byte`)。

关键点:

- 只有**可导出的字段**（首字母大写）才会被编码。
- 默认情况下，JSON 的键名与 Go 结构体的字段名相同。
- 可以使用 struct tag `json:"..."` 来自定义 JSON 键名。

```go
// 示例：将一个包含基本类型和自定义类型的结构体编码为 JSON
type Response1 struct {
    Page   int
    Fruits []string
}
res1D := &Response1{
    Page:   1,
    Fruits: []string{"apple", "peach", "pear"},
}
res1B, _ := json.Marshal(res1D)
fmt.Println(string(res1B)) // 输出: {"Page":1,"Fruits":["apple","peach","pear"]}
```

55.2.2 解码 (Unmarshal) - 从 JSON 到 Go 结构

`json.Unmarshal` 函数用于将 JSON 字节切片解析到 Go 的数据结构中。

关键点:

- 需要传递一个指向目标数据结构的指针，以便函数可以修改它。
- `json.Unmarshal` 会根据 JSON 的键名和 struct tag 来匹配并填充相应的字段。

```go
// 示例：将 JSON 字符串解码到一个自定义的 struct 中
type Response2 struct {
    Page   int      `json:"page"`
    Fruits []string `json:"fruits"`
}
str := `{"page": 1, "fruits": ["apple", "peach", "pear"]}`
res2 := Response2{}
json.Unmarshal([]byte(str), &res2)
fmt.Println(res2) // 输出: {1 [apple peach pear]}
```

55.2.3 解码到通用接口

当 JSON 结构未知或不固定时，可以将其解码到一个 `map[string]interface{}` 中。值会根据 JSON 类型自动转换为 Go 的 `string`, `float64`, `bool`, `[]interface{}`, `map[string]interface{}` 等类型。

```go
// 示例：将 JSON 解码到 map 中
byt := []byte(`{"num":6.13,"strs":["a","b"]}`)
var dat map[string]interface{}
json.Unmarshal(byt, &dat)
fmt.Println(dat) // 输出: map[num:6.13 strs:[a b]]
```

55.2.4 流式编解码

对于大型 JSON 数据或网络连接，使用 `json.Encoder` 和 `json.Decoder` 进行流式处理是更高效的方式，可以避免将整个 JSON 数据读入内存。

```
// 示例：将 JSON 数据流式编码到 os.Stdout
enc := json.NewEncoder(os.Stdout)
d := map[string]int{"apple": 5, "lettuce": 7}
enc.Encode(d) // 直接将编码后的 JSON 写入标准输出
```

55.3 完整代码示例

```
package main

import (
    "encoding/json"
    "fmt"
    "os"
)

// Response1 演示了基本类型的编码
type Response1 struct {
    Page   int
    Fruits []string
}

// Response2 演示了使用 struct tag 自定义 JSON 键名
type Response2 struct {
    Page   int      `json:"page"`
    Fruits []string `json:"fruits"`
}

func main() {
    // --- 编码 (Marshal) ---

    // 基本数据类型到 JSON
    bolB, _ := json.Marshal(true)
    fmt.Println(string(bolB))

    intB, _ := json.Marshal(1)
    fmt.Println(string(intB))

    fltB, _ := json.Marshal(2.34)
    fmt.Println(string(fltB))

    strB, _ := json.Marshal("gopher")
    fmt.Println(string(strB))

    // 切片和 map 到 JSON
    slcD := []string{"apple", "peach", "pear"}
    slcB, _ := json.Marshal(slcD)
    fmt.Println(string(slcB))

    mapD := map[string]int{"apple": 5, "lettuce": 7}
    mapB, _ := json.Marshal(mapD)
    fmt.Println(string(mapB))
```

```
    // 自定义结构体到 JSON
    res1D := &Response1{
        Page:   1,
        Fruits: []string{"apple", "peach", "pear"}}
    res1B, _ := json.Marshal(res1D)
    fmt.Println(string(res1B))

    // 使用带 struct tag 的结构体
    res2D := &Response2{
        Page:   1,
        Fruits: []string{"apple", "peach", "pear"}}
    res2B, _ := json.Marshal(res2D)
    fmt.Println(string(res2B))

    // --- 解码 (Unmarshal) ---

    // 将 JSON 解码到通用接口 (map)
    byt := []byte(`{"num":6.13,"strs":["a","b"]}`)
    var dat map[string]interface{}
    if err := json.Unmarshal(byt, &dat); err != nil {
        panic(err)
    }
    fmt.Println(dat)

    // 为了访问 map 中的值，需要进行类型断言
    num := dat["num"].(float64)
    fmt.Println(num)

    // 访问嵌套数据
    strs := dat["strs"].([]interface{})
    str1 := strs[0].(string)
    fmt.Println(str1)

    // 将 JSON 解码到自定义结构体
    str := `{"page": 1, "fruits": ["apple", "peach", "pear"]}`
    res := Response2{}
    json.Unmarshal([]byte(str), &res)
    fmt.Println(res)
    fmt.Println(res.Fruits[0])

    // --- 流式编码 ---
    // 可以将 JSON 编码直接流式传输到 os.Writer (如 os.Stdout) 或 HTTP 响应体
    enc := json.NewEncoder(os.Stdout)
    d := map[string]int{"apple": 5, "lettuce": 7}
    enc.Encode(d)
}
```

55.4 运行结果

```
$ go run json.go
```

```
true
1
2.34
"gopher"
["apple","peach","pear"]
{"apple":5,"lettuce":7}
{"Page":1,"Fruits":["apple","peach","pear"]}
{"page":1,"fruits":["apple","peach","pear"]}
map[num:6.13 strs:[a b]]
6.13
a
{1 [apple peach pear]}
apple
{"apple":5,"lettuce":7}
```

55.5 关键要点

1. **`json.Marshal` 与 `json.Unmarshal`**：这是 Go 与 JSON 之间进行转换的两个核心函数，分别用于编码和解码。
2. **导出字段**：只有首字母大写的 struct 字段才会被 JSON 包处理，这是 Go 的可见性规则在序列化中的体现。
3. **Struct Tags 的妙用**：使用 `json:"..."` struct tag 可以自定义 JSON 字段名，解耦 Go 结构体命名和外部 JSON 格式。
4. **解码到 `interface{}`**：当 JSON 结构不确定时，解码到 `map[string]interface{}` 提供了极大的灵活性，但需要通过类型断言来访问数据。
5. **流式处理**：对于网络数据或大文件，使用 `json.Encoder` 和 `json.Decoder` 进行流式处理是更高效、内存占用更低的选择。
6. **错误处理**：在实际应用中，必须检查 `json.Marshal` 和 `json.Unmarshal` 返回的错误，以确保数据处理的健壮性。

56. XML (XML 处理)

https://gobyexample.com/xml

56.1 核心概念

Go 通过 `encoding/xml` 包为解析和生成 XML 提供了内置支持。这个包提供了一套强大而灵活的工具，用于将 Go 的数据结构与 XML 文档进行相互转换。

生活化类比：如果说 JSON 是现代应用间交流的“通用语”，那么 XML 就像是更为传统和正式的“官方文书”。它格式严谨，层级分明，常用于企业级应用、配置文件和旧有的 Web 服务（如 SOAP）中。处理 XML 就像是填写或阅读一份结构复杂的官方表格，每个字段都有明确的标签和层级，`encoding/xml` 包就是帮助你自动完成这份“表格”填写与读取的工具。

56.2 XML 的基本用法

56.2.1 编码 (Marshal) - 从 Go 结构到 XML

`xml.Marshal` 和 `xml.MarshalIndent` 用于将 Go 结构体编码为 XML。`MarshalIndent` 会生成带缩进的、更易读的输出。

关键点:

- **`xml.Name` 字段**: `XMLName` 字段用于指定结构体对应的 XML 元素的名称。
- **Struct Tags**: 使用 `xml:"..."` struct tag 来控制 XML 的结构。
  - `xml:"id,attr"`: 将字段编码为 XML 元素的属性 (attribute)。
  - `xml:"parent>child"`: 创建嵌套的 XML 元素。

```
// Plant 结构体定义了 XML 的基本结构
type Plant struct {
    XMLName xml.Name `xml:"plant"`
    Id      int      `xml:"id,attr"`
    Name    string   `xml:"name"`
    Origin  []string `xml:"origin"`
}

// 编码示例
coffee := &Plant{Id: 27, Name: "Coffee"}
coffee.Origin = []string{"Ethiopia", "Brazil"}
out, _ := xml.MarshalIndent(coffee, " ", "   ")
fmt.Println(string(out))
```

56.2.2 解码 (Unmarshal) - 从 XML 到 Go 结构

`xml.Unmarshal` 用于将 XML 数据解析到 Go 结构体中。它会根据 `XMLName` 和 struct tags 自动匹配和填充字段。

```
// 解码示例
var p Plant
if err := xml.Unmarshal(out, &p); err != nil {
    panic(err)
}
fmt.Println(p)
```

56.2.3 处理嵌套和父级元素

通过在 struct tag 中使用 `>`，可以轻松地处理嵌套的 XML 结构，这使得 Go 的数据结构可以与复杂的 XML 格式进行映射。

```
// Nesting 结构体演示了如何处理嵌套元素
type Nesting struct {
    XMLName xml.Name `xml:"nesting"`
    Plants  []*Plant `xml:"parent>child>plant"`
}

// 编码嵌套结构
nesting := &Nesting{}
nesting.Plants = []*Plant{coffee, &Plant{Id: 81, Name: "Tomato"}}
out, _ = xml.MarshalIndent(nesting, " ", "   ")
fmt.Println(string(out))
```

56.3 完整代码示例

```
package main

import (
    "encoding/xml"
    "fmt"
)

// Plant 结构体将被映射到 XML
// 字段标签决定了 XML 元素的名称和属性
type Plant struct {
    XMLName xml.Name `xml:"plant"`
    Id      int      `xml:"id,attr"`
    Name    string   `xml:"name"`
    Origin  []string `xml:"origin"`
}

func (p Plant) String() string {
    return fmt.Sprintf("Plant id=%v, name=%v, origin=%v",
        p.Id, p.Name, p.Origin)
}

func main() {
    // --- 编码 (Marshal) ---
    coffee := &Plant{Id: 27, Name: "Coffee"}
    coffee.Origin = []string{"Ethiopia", "Brazil"}

    // 使用 MarshalIndent 生成带缩进的、人类可读的 XML
    out, _ := xml.MarshalIndent(coffee, " ", "   ")
    fmt.Println(string(out))

    // 添加通用的 XML 头部
    fmt.Println(xml.Header + string(out))

    // --- 解码 (Unmarshal) ---
    var p Plant
    if err := xml.Unmarshal(out, &p); err != nil {
        panic(err)
```

```
    }
    fmt.Println(p)

    // --- 处理嵌套元素 ---
    tomato := &Plant{Id: 81, Name: "Tomato"}
    tomato.Origin = []string{"Mexico", "California"}

    type Nesting struct {
        XMLName xml.Name `xml:"nesting"`
        Plants  []*Plant `xml:"parent>child>plant"`
    }

    nesting := &Nesting{}
    nesting.Plants = []*Plant{coffee, tomato}

    out, _ = xml.MarshalIndent(nesting, " ", "  ")
    fmt.Println(string(out))
}
```

56.4 运行结果

```
$ go run xml.go
 <plant id="27">
   <name>Coffee</name>
   <origin>Ethiopia</origin>
   <origin>Brazil</origin>
 </plant>
<?xml version="1.0" encoding="UTF-8"?>
 <plant id="27">
   <name>Coffee</name>
   <origin>Ethiopia</origin>
   <origin>Brazil</origin>
 </plant>
Plant id=27, name=Coffee, origin=[Ethiopia Brazil]
 <nesting>
   <parent>
     <child>
       <plant id="27">
         <name>Coffee</name>
         <origin>Ethiopia</origin>
         <origin>Brazil</origin>
       </plant>
       <plant id="81">
         <name>Tomato</name>
         <origin>Mexico</origin>
         <origin>California</origin>
       </plant>
     </child>
   </parent>
 </nesting>
```

56.5 关键要点

1. **`encoding/xml` 包**: Go 处理 XML 的标准库，功能强大且全面。
2. **`MarshalIndent`**: 生成格式化、易于阅读的 XML 输出，非常适合调试和配置文件。
3. **Struct Tags 是核心**: `xml:"..."` 标签是连接 Go 结构体和 XML 文档结构的关键，提供了极高的灵活性。
   - `id,attr`: 定义属性。
   - `parent>child`: 定义嵌套关系。
4. **`XMLName` 字段**: 用于指定根元素的名称，是良好实践的一部分。
5. **解码的健壮性**: `Unmarshal` 会尽力解析 XML，并能处理字段不完全匹配的情况，但在生产代码中必须检查其返回的错误。
6. **与 JSON 的异同**: 与 `encoding/json` 类似，`encoding/xml` 也通过 struct tags 来控制序列化，但 XML 的层级和属性结构使其标签语法更为丰富。

57. Time (时间)

https://gobyexample.com/time

57.1 核心概念

Go 语言通过其标准库中的 `time` 包，为测量和展示时间提供了强大的支持。`time` 包涵盖了获取当前时间、时间点创建、时间运算、格式化等多种常用功能。

生活化类比：你可以把 Go 的 `time` 包想象成一个功能强大的“瑞士军刀”手表。它不仅能告诉你现在几点了（`time.Now()`），还能让你设定一个未来的闹钟（`time.Date()`），计算两个时刻之间过去了多久（`Sub()`），或者把时间向前或向后拨动（`Add()`）。更重要的是，它还能根据你的需要，把时间显示成各种不同的格式（`Format()`），就像一块既有数字显示又有指针表盘，还能显示多国时区的手表一样。

57.2 时间的基本用法

57.2.1 获取当前时间

使用 `time.Now()` 可以获取当前的本地时间。

```
p := fmt.Println

now := time.Now()
p(now)
```

57.2.2 创建指定时间

使用 `time.Date()` 可以创建一个具有指定年、月、日、时、分、秒、纳秒和时区的时间点。

```
then := time.Date(
    2009, 11, 17, 20, 34, 58, 651387237, time.UTC)
p(then)
```

57.2.3 提取时间分量

`time.Time` 对象提供了多种方法来提取时间的各个部分。

```
p(then.Year())
p(then.Month())
p(then.Day())
p(then.Hour())
p(then.Minute())
p(then.Second())
p(then.Nanosecond())
p(then.Location())

// 获取星期几
p(then.Weekday())
```

57.2.4 时间比较

可以使用 `Before`, `After`, `Equal` 方法来比较两个时间点。

```
p(then.Before(now))
p(then.After(now))
p(then.Equal(now))
```

57.2.5 时间运算

`Sub` 方法返回一个 `time.Duration` 对象，表示两个时间点之间的时间差。`Add` 方法用于在一个时间点上增加一个时间段。

```
// Sub 返回两个时间点的时间差 (Duration)
diff := now.Sub(then)
p(diff)

// Duration 对象可以被转换成不同的单位
p(diff.Hours())
p(diff.Minutes())
p(diff.Seconds())
p(diff.Nanoseconds())

// Add 方法可以在一个时间点上增加一个 Duration
p(then.Add(diff))
// Add 也可以用负数来表示减去一个 Duration
p(then.Add(-diff))
```

57.3 完整代码示例

```
package main

import (
```

```
    "fmt"
    "time"
)

func main() {
    p := fmt.Println

    // 获取并打印当前时间
    now := time.Now()
    p(now)

    // 通过提供年月日等信息创建一个 time.Time 对象
    // 时间总是与一个 Location（时区）相关联
    then := time.Date(
        2009, 11, 17, 20, 34, 58, 651387237, time.UTC)
    p(then)

    // 你可以提取时间的各个组成部分
    p(then.Year())
    p(then.Month())
    p(then.Day())
    p(then.Hour())
    p(then.Minute())
    p(then.Second())
    p(then.Nanosecond())
    p(then.Location())

    // 输出是星期几
    p(then.Weekday())

    // 比较两个时间，测试一个是否在另一个之前、之后或相等
    p(then.Before(now))
    p(then.After(now))
    p(then.Equal(now))

    // Sub 方法返回一个 Duration，表示两个时间之间的时间差
    diff := now.Sub(then)
    p(diff)

    // 我们可以用各种单位来表示 Duration 的长度
    p(diff.Hours())
    p(diff.Minutes())
    p(diff.Seconds())
    p(diff.Nanoseconds())

    // 可以使用 Add 将一个 Duration 添加到一个时间点上，
    // 得到一个新的时间点
    p(then.Add(diff))
    // Add 也可以接收负数，表示从一个时间点减去一个 Duration
    p(then.Add(-diff))
}
```

57.4 运行结果

```
$ go run time.go
2023-10-27 10:30:00.123456789 +0800 CST m=+0.000000001
2009-11-17 20:34:58.651387237 +0000 UTC
2009
November
17
20
34
58
651387237
UTC
Tuesday
true
false
false
122525h55m1.472069552s
122525.91707557488
7351555.024534492
441093301.4720695
441093301472069552
2023-10-27 10:30:00.123456789 +0800 CST m=+0.000000001
1996-01-01 01:09:57.179314474 +0000 UTC
```

(注：运行结果中的当前时间会根据实际运行时刻而变化)

57.5 关键要点

1. **`time.Time` 核心类型**：Go 中所有的时间点都由 `time.Time` 结构体表示，它包含了纳秒级别的精度和一个时区信息。
2. **`time.Now()` 获取当前**：这是获取当前本地时间最直接的方式。
3. **`time.Date()` 创建指定时间**：用于精确创建一个过去或未来的时间点，时区（`Location`）是其重要组成部分。
4. **时间是可比较的**：`Before()`, `After()`, `Equal()` 提供了一套安全且易读的时间点比较方法。
5. **`time.Duration` 表示时间段**：时间点之间的运算结果是一个 `Duration` 类型，它可以被方便地转换为小时、分钟、秒等单位。
6. **时间点与时间段的运算**：使用 `Add()` 方法可以方便地进行时间点的推移计算。

---

58. Epoch (纪元时间)

https://gobyexample.com/epoch

58.1 核心概念

Epoch（纪元时间）是指自世界标准时间（UTC）1970年1月1日 00:00:00 起经过的秒数。在计算机科学中，这是一种广泛使用的时间表示方法，常用于时间戳和跨系统的时间同步。Go 的 `time` 包提供了方便的函数来获取和转换 Epoch 时间。

58.2 Epoch 时间的使用

58.2.1 获取不同精度的 Epoch 时间

Go 允许你以秒、毫秒或纳秒的精度获取当前时间的 Epoch 值。

- **秒 (Seconds)**

  ```
  now := time.Now()
  secs := now.Unix()
  ```

  `now.Unix()` 返回自 Epoch 以来的秒数。

- **纳秒 (Nanoseconds)**

  ```
  nanos := now.UnixNano()
  ```

  `now.UnixNano()` 返回自 Epoch 以来的纳秒数。

- **毫秒 (Milliseconds)**

  ```
  millis := nanos / 1000000
  ```

  Go 的 `time` 包中没有直接的 `UnixMillis()` 函数，但可以通过纳秒转换得到。

58.2.2 将 Epoch 时间转换回 `time.Time`

你可以将一个 Epoch 时间（秒或纳秒）转换回一个 `time.Time` 对象。

- **从秒和纳秒创建**

  ```
  // time.Unix 接收秒和纳秒作为参数
  fmt.Println(time.Unix(secs, 0))
  ```

- **从纳秒创建**

  ```
  // 第二个参数可以用于指定纳秒偏移
  fmt.Println(time.Unix(0, nanos))
  ```

58.3 完整代码示例

```
package main

import (
    "fmt"
```

```
    "time"
)

func main() {
    // 使用 time.Now() 获取当前时间
    now := time.Now()

    // 使用 Unix() 或 UnixNano() 获取自 Unix 纪元以来的秒数或纳秒数
    secs := now.Unix()
    nanos := now.UnixNano()
    fmt.Println(now)

    // 注意，Go 中没有直接获取毫秒数的 UnixMillis，
    // 但你可以手动从纳秒转换
    millis := nanos / 1000000
    fmt.Println(secs)
    fmt.Println(millis)
    fmt.Println(nanos)

    // 你也可以将自纪元以来的秒数或纳秒数转换回 time.Time
    fmt.Println(time.Unix(secs, 0))
    fmt.Println(time.Unix(0, nanos))
}
```

58.4 运行结果

```
$ go run epoch.go
2023-10-27 11:00:00.123456789 +0800 CST m=+0.000000001
1698375600
1698375600123
1698375600123456789
2023-10-27 11:00:00 +0800 CST
2023-10-27 11:00:00.123456789 +0800 CST
```

(注：运行结果中的时间戳会根据实际运行时刻而变化)

58.5 关键要点

1. **Epoch 的定义**：Epoch 时间是自 1970-01-01 UTC 以来的持续时间，是跨系统通信的常用时间标准。
2. **多种精度**：Go 提供了 `Unix()` (秒) 和 `UnixNano()` (纳秒) 来获取不同精度的 Epoch 时间戳。
3. **手动转换毫秒**：虽然没有直接获取毫秒的函数，但可以通过纳秒 `UnixNano() / 1000000` 轻松得到。
4. **双向转换**：`time.Unix()` 函数可以将秒或纳秒的 Epoch 值转换回 `time.Time` 对象，实现了时间的序列化与反序列化。
5. **生活化类比**：Epoch 时间就像一个从固定起点开始计数的“宇宙秒表”，任何一个时间点都可以用这个秒表上的读数来唯一表示，不受时区和日历规则的干扰。

59. Time Formatting / Parsing (时间格式化与解析)

https://gobyexample.com/time-formatting-parsing

59.1 核心概念

Go 使用一种独特的、基于示例的方法来格式化和解析时间，而不是使用常见的格式化字符串（如 `YYYY-MM-DD`）。你需要提供一个布局字符串，该字符串是通过格式化一个特定的参考时间点来创建的：`Mon Jan 2 15:04:05 MST 2006` (即 Unix 时间 `1136239445`)。

生活化类比：Go 的时间格式化就像是“照样子画画”。你不用学习一套复杂的指令（像 `yyyy-mm-dd`），而是直接给 Go 一个“样板”：`2006-01-02`。Go 看到这个样板，就明白了“哦，你是想要‘年-月-日’的格式”，然后它就会把任何时间都按照这个样子画出来。这个神奇的“样板时间”就是 `Mon Jan 2 15:04:05 MST 2006`，你只需要用它来组合出你想要的任何格式。

59.2 时间格式化与解析

59.2.1 使用标准布局进行格式化

`time` 包提供了一系列预定义的布局常量，如 `time.RFC3339`，方便直接使用。

```
t := time.Now()
// 使用 RFC3339 格式化当前时间
fmt.Println(t.Format(time.RFC3339))
```

59.2.2 使用标准布局进行解析

`time.Parse` 函数接收一个布局字符串和一个时间字符串，并尝试将其解析为一个 `time.Time` 对象。

```
t1, _ := time.Parse(
    time.RFC3339,
    "2012-11-01T22:08:41+00:00")
fmt.Println(t1)
```

59.2.3 自定义布局

你可以通过格式化参考时间 `Mon Jan 2 15:04:05 MST 2006` 来创建任何你需要的布局。

```
// 仅格式化为 "小时:分钟AM/PM"
fmt.Println(t.Format("3:04PM"))

// 一个更复杂的自定义格式
fmt.Println(t.Format("Mon Jan _2 15:04:05 2006"))

// 解析一个自定义格式的字符串
form := "3 04 PM"
t2, _ := time.Parse(form, "8 41 PM")
fmt.Println(t2)
```

记忆技巧：参考时间可以记为 `1（月）2（日）3（时）4（分）5（秒）6（年）7（时区）`。

59.2.4 纯数字格式化

对于纯数字的时间表示，可以直接从 `time.Time` 对象中提取年月日等部分，并使用 `fmt.Printf` 进行格式化。

```
fmt.Printf("%d-%02d-%02dT%02d:%02d:%02d-07:00\n",
    t.Year(), t.Month(), t.Day(),
    t.Hour(), t.Minute(), t.Second())
```

59.2.5 解析错误处理

如果输入的时间字符串与布局不匹配，`time.Parse` 会返回一个错误。

```
// 尝试用一个完整的日期布局来解析一个不匹配的字符串
_, e := time.Parse(time.RFC822, "8:41PM")
fmt.Println(e) // 会报告解析错误
```

59.3 完整代码示例

```
package main

import (
    "fmt"
    "time"
)

func main() {
    p := fmt.Println

    t := time.Now()
    p(t.Format(time.RFC3339))

    t1, _ := time.Parse(
        time.RFC3339,
        "2012-11-01T22:08:41+00:00")
    p(t1)

    p(t.Format("3:04PM"))
    p(t.Format("Mon Jan _2 15:04:05 2006"))
    p(t.Format("2006-01-02T15:04:05.999999-07:00"))
    form := "3 04 PM"
    t2, _ := time.Parse(form, "8 41 PM")
    p(t2)

    fmt.Printf("%d-%02d-%02dT%02d:%02d:%02d-07:00\n",
        t.Year(), t.Month(), t.Day(),
        t.Hour(), t.Minute(), t.Second())

    ansic := "Mon Jan _2 15:04:05 2006"
    _, e := time.Parse(ansic, "8:41PM")
    p(e)
}
```

59.4 运行结果

```
$ go run time-formatting-parsing.go
2023-10-27T11:15:00+08:00
2012-11-01 22:08:41 +0000 UTC
11:15AM
Fri Oct 27 11:15:00 2023
2023-10-27T11:15:00.123456+08:00
0000-01-01 20:41:00 +0000 UTC
2023-10-27T11:15:00-07:00
parsing time "8:41PM" as "Mon Jan _2 15:04:05 2006": cannot parse "8:41PM" as "Mon"
```

(注：运行结果中的当前时间会根据实际运行时刻而变化)

59.5 关键要点

1. **基于示例的布局**：Go 的时间格式化和解析不使用抽象的占位符（如 `yyyy-mm-dd`），而是通过格式化一个固定的参考时间来定义布局。
2. **参考时间是关键**：所有自定义布局都必须基于 `Mon Jan 2 15:04:05 MST 2006` 这个时间点来创建。
3. **`Format` 和 `Parse`**：`Format` 用于将 `time.Time` 转换为字符串，`Parse` 用于将字符串转换为 `time.Time`，两者使用相同的布局逻辑。
4. **预定义常量**：`time` 包提供了如 `time.RFC3339`, `time.ANSIC` 等多种标准布局常量，应优先使用。
5. **错误处理**：当解析的字符串与布局不匹配时，`Parse` 会返回一个详细的错误，方便调试。

---

60. Random Numbers (随机数)

https://gobyexample.com/random-numbers

60.1 核心概念

Go 的 `math/rand` 包提供了伪随机数生成的功能。从 Go 1.22 开始，推荐使用 `math/rand/v2` 版本，它提供了更现代、更一致的 API，并且默认情况下无需手动播种 (seeding)。

生活化类比：随机数生成器就像一副被洗过的扑克牌。默认情况下，每次你运行程序，Go 都会帮你用一种新的、不可预测的方式洗牌（自动播种），所以你每次抽牌（生成随机数）的结果都不同。但如果你想在测试时重现某次“牌局”，你可以自己指定一种固定的洗牌方法（提供一个种子），这样每次用同一种方法洗出来的牌，抽牌的顺序就完全一样了。

60.2 随机数生成

60.2.1 生成随机整数

`rand.IntN(n)` 返回一个在 `[0, n)` 区间内的随机整数。

```
// 生成一个 0 到 99 之间的随机整数
fmt.Println(rand.IntN(100))
```

60.2.2 生成随机浮点数

`rand.Float64()` 返回一个在 `[0.0, 1.0)` 区间内的随机 `float64` 值。

```
// 生成一个 0.0 到 1.0 之间的随机浮点数
fmt.Println(rand.Float64())
```

你可以通过这个函数来生成其他范围内的浮点数。

```
// 生成一个 5.0 到 10.0 之间的随机浮点数
fmt.Println((rand.Float64() * 5) + 5)
```

60.2.3 使用自定义种子源

默认情况下，`math/rand/v2` 包会自动处理种子，确保每次程序运行时都能得到不同的随机数序列。但如果你需要一个可复现的随机数序列（例如在测试中），你可以创建一个自定义的随机数源（`Source`）和生成器（`Rand`）。

```
// 1. 创建一个种子源，例如 PCG (Permuted Congruential Generator)
//    使用固定的种子值 (例如 42)
s1 := rand.NewSource(42)

// 2. 使用该种子源创建一个新的随机数生成器
r1 := rand.New(s1)

// 3. 使用这个生成器来产生随机数
fmt.Println(r1.IntN(100))
```

60.2.4 种子的重要性

如果你使用相同的种子创建多个随机数源，那么它们将产生完全相同的随机数序列。

```
// 创建另一个使用相同种子 (42) 的源和生成器
s2 := rand.NewSource(42)
r2 := rand.New(s2)

// r2 生成的序列将与 r1 完全相同
fmt.Println(r2.IntN(100))
```

60.3 完整代码示例

```
package main

import (
    "fmt"
    "math/rand/v2"
)

func main() {
    // 默认情况下，v2版本的rand无需手动播种
```

```
    fmt.Print(rand.IntN(100), ",")
    fmt.Print(rand.IntN(100))
    fmt.Println()

    fmt.Println(rand.Float64())

    // 生成 [5.0, 10.0) 范围的浮点数
    fmt.Print((rand.Float64()*5)+5, ",")
    fmt.Print((rand.Float64()*5)+5)
    fmt.Println()

    // 如果需要可复现的随机序列，可以提供一个种子
    s1 := rand.NewSource(42)
    r1 := rand.New(s1)

    fmt.Print(r1.IntN(100), ",")
    fmt.Print(r1.IntN(100))
    fmt.Println()

    // 使用相同种子的源将产生相同的随机数序列
    s2 := rand.NewSource(42)
    r2 := rand.New(s2)
    fmt.Print(r2.IntN(100), ",")
    fmt.Print(r2.IntN(100))
    fmt.Println()
}
```

60.4 运行结果

```
$ go run random-numbers.go
81,87
0.6645600532184904
7.442981739261645,8.74243950033655
5,87
5,87
```

(注：前三行的随机数在每次运行时都会变化，而后两行由于使用了固定的种子，其结果是固定的)

60.5 关键要点

1. **使用 `math/rand/v2`**：从 Go 1.22 开始，这是推荐的随机数包，它默认自动播种，使用更方便。
2. **`IntN` 和 `Float64`**：是生成整数和浮点数最常用的函数。
3. **确定性与种子**：随机数是伪随机的。默认情况下，种子是变化的，产生不重复的序列。如果需要固定的、可复现的序列，必须手动创建并提供一个确定的种子源。
4. **并发安全**：`math/rand/v2` 的全局函数（如 `rand.IntN`）是并发安全的。如果你创建了自己的 `rand.Rand` 实例，它不是并发安全的，需要在多个 goroutine 中使用时进行同步。

61. Number Parsing (数字解析)

https://gobyexample.com/number-parsing

61.1 核心概念

在处理字符串时，一个常见的需求是将其解析为数字。Go 的 `strconv` 包提供了强大的功能来完成这项任务。

生活化类比：`strconv` 包就像一个“翻译官”，专门负责在“文本世界”和“数字世界”之间进行翻译。当你看到纸上写着 "123"（一个字符串），你的大脑会自动把它理解成数字一百二十三。`strconv.Atoi` 等函数就是做了同样的事情：它读取文本形式的数字，并将其转换成程序可以进行数学运算的真正数值。

61.2 数字解析函数

61.2.1 解析浮点数

`strconv.ParseFloat` 用于将字符串解析为浮点数。第二个参数 `64` 指定了解析出的数字的位数，这里是 `float64`。

```
f, _ := strconv.ParseFloat("1.234", 64)
fmt.Println(f)
```

61.2.2 解析整数

`strconv.ParseInt` 用于解析整数。它有三个参数：

- `s`: 要解析的字符串。
- `base`: 数字的进制（2到36）。如果 `base` 为 0，则会根据字符串的前缀自动推断进制（例如 `0x` 表示16进制）。
- `bitSize`: 结果的整数类型大小（例如 0, 8, 16, 32, 64 分别对应 `int`, `int8`, `int16`, `int32`, `int64`）。

```
// 解析一个标准的十进制整数
i, _ := strconv.ParseInt("123", 0, 64)
fmt.Println(i)

// 自动推断并解析一个十六进制数
d, _ := strconv.ParseInt("0x1c8", 0, 64)
fmt.Println(d)
```

61.2.3 解析无符号整数

`strconv.ParseUint` 与 `ParseInt` 类似，但用于解析无符号整数。

```
u, _ := strconv.ParseUint("789", 0, 64)
fmt.Println(u)
```

61.2.4 `Atoi` - 便捷的十进制整数解析

`strconv.Atoi` 是一个方便的函数，专门用于将十进制字符串解析为 `int` 类型。它是 `ParseInt(s, 10, 0)` 的一个快捷方式。

```
k, _ := strconv.Atoi("135")
fmt.Println(k)
```

61.2.5 错误处理

如果字符串无法被解析为有效的数字，解析函数会返回一个错误。

```
_, e := strconv.Atoi("wat")
fmt.Println(e)
```

61.3 完整代码示例

```
package main

import (
    "fmt"
    "strconv"
)

func main() {
    // 使用 ParseFloat 解析浮点数，64 表示解析为 float64
    f, _ := strconv.ParseFloat("1.234", 64)
    fmt.Println(f)

    // 使用 ParseInt 解析整数。0 表示从字符串中推断进制
    i, _ := strconv.ParseInt("123", 0, 64)
    fmt.Println(i)

    // ParseInt 可以识别十六进制格式
    d, _ := strconv.ParseInt("0x1c8", 0, 64)
    fmt.Println(d)

    // 使用 ParseUint 解析无符号整数
    u, _ := strconv.ParseUint("789", 0, 64)
    fmt.Println(u)

    // Atoi 是一个方便的函数，用于将十进制字符串解析为 int
    k, _ := strconv.Atoi("135")
    fmt.Println(k)

    // 解析函数在输入错误时会返回一个错误
    _, e := strconv.Atoi("wat")
    fmt.Println(e)
}
```

61.4 运行结果

```
$ go run number-parsing.go
1.234
123
456
789
135
strconv.Atoi: parsing "wat": invalid syntax
```

61.5 关键要点

1. **`strconv` 包是核心**：所有字符串到数字的转换都由 `strconv` 包提供。
2. **`Parse` 系列函数**：`ParseFloat`, `ParseInt`, `ParseUint` 是最基础和最灵活的解析函数，允许你指定进制和位数。
3. **`Atoi` 的便利性**：`Atoi` 是 `ParseInt` 的一个常用快捷方式，专门用于解析十进制 `int`。
4. **错误处理是必须的**：在实际应用中，必须检查解析函数返回的错误，以确保字符串是有效的数字。
5. **进制推断**：将 `ParseInt` 的 `base` 参数设为 0，可以让它自动识别如 `0x` (十六进制) 或 `0` (八进制) 的前缀。

62. URL Parsing (URL 解析)

https://gobyexample.com/url-parsing

62.1 核心概念

URL (Uniform Resource Locator) 是网络上资源的地址。在 Go 中，`net/url` 包提供了强大的工具来解析和构建 URL。解析 URL 意味着将其分解为各个组成部分，如协议、主机、路径和查询参数。

生活化类比：URL 解析就像是分析一个完整的邮寄地址。一个地址“中国北京市朝阳区幸福路1号张三收”包含了国家、城市、街道、门牌号和收件人等多个部分。`url.Parse` 函数就是这样一个地址分析专家，它能把一长串的网址 `postgres://user:pass@host.com:5432/path?k=v#f`，精准地拆解成协议 (`postgres`)、用户名 (`user`)、主机 (`host.com`)、路径 (`/path`) 和查询参数 (`k=v`) 等结构化的部分，方便你单独使用其中任何一部分。

62.2 URL 解析

62.2.1 `url.Parse` 函数

`url.Parse` 是解析 URL 字符串的核心函数。它返回一个 `*url.URL` 结构体和一个错误。

```
s := "postgres://user:pass@host.com:5432/path?k=v#f"

u, err := url.Parse(s)
if err != nil {
    panic(err)
}
```

62.2.2 解析协议 (Scheme)

`u.Scheme` 字段包含了 URL 的协议部分。

```
fmt.Println(u.Scheme) // 输出: postgres
```

62.2.3 解析用户信息 (User)

`u.User` 字段包含了认证信息。你可以从中提取用户名和密码。

```
fmt.Println(u.User) // 输出: user:pass
fmt.Println(u.User.Username()) // 输出: user
p, _ := u.User.Password() // Password() 返回 (string, bool)
fmt.Println(p) // 输出: pass
```

62.2.4 解析主机和端口 (Host, Port)

`u.Host` 字段包含了主机名和端口。`net.SplitHostPort` 函数可以方便地将它们分开。

```
fmt.Println(u.Host) // 输出: host.com:5432
host, port, _ := net.SplitHostPort(u.Host)
fmt.Println(host) // 输出: host.com
fmt.Println(port) // 输出: 5432
```

62.2.5 解析路径和片段 (Path, Fragment)

`u.Path` 包含路径部分，`u.Fragment` 包含 URL 的片段标识符（`#` 后面的部分）。

```
fmt.Println(u.Path)     // 输出: /path
fmt.Println(u.Fragment) // 输出: f
```

62.2.6 解析查询参数 (Query Parameters)

`u.RawQuery` 存储了 URL 中的原始查询字符串。`url.ParseQuery` 函数可以将其解析为一个 `map[string][]string`。

```
fmt.Println(u.RawQuery) // 输出: k=v

m, _ := url.ParseQuery(u.RawQuery)
fmt.Println(m)          // 输出: map[k:[v]]
fmt.Println(m["k"][0]) // 输出: v
```

注意：查询参数的值是一个字符串切片 `[]string`，以支持像 `?k=v1&k=v2` 这样的重复键。

62.3 完整代码示例

```
package main

import (
    "fmt"
    "net"
    "net/url"
```

```
)

func main() {
    s := "postgres://user:pass@host.com:5432/path?k=v#f"

    u, err := url.Parse(s)
    if err != nil {
        panic(err)
    }

    // 协议
    fmt.Println(u.Scheme)

    // 用户认证信息
    fmt.Println(u.User)
    fmt.Println(u.User.Username())
    p, _ := u.User.Password()
    fmt.Println(p)

    // 主机和端口
    fmt.Println(u.Host)
    host, port, _ := net.SplitHostPort(u.Host)
    fmt.Println(host)
    fmt.Println(port)

    // 路径和片段
    fmt.Println(u.Path)
    fmt.Println(u.Fragment)

    // 查询参数
    fmt.Println(u.RawQuery)
    m, _ := url.ParseQuery(u.RawQuery)
    fmt.Println(m)
    fmt.Println(m["k"][0])
}
```

62.4 运行结果

```
$ go run url-parsing.go
postgres
user:pass
user
pass
host.com:5432
host.com
5432
/path
f
k=v
map[k:[v]]
v
```

62.5 关键要点

1. **`net/url` 包**：是 Go 中处理 URL 的标准库。
2. **`url.Parse` 函数**：是将原始 URL 字符串分解为结构化 `url.URL` 对象的核心。
3. **结构化访问**：解析后，可以通过 `url.URL` 结构体的字段（如 `Scheme`, `Host`, `Path`）轻松访问 URL 的各个部分。
4. **查询参数是切片**：`url.ParseQuery` 将查询参数解析为 `map[string][]string`，以正确处理同名参数的情况。
5. **错误处理**：对于格式不正确的 URL，`url.Parse` 会返回错误，需要进行检查。

63. SHA256 Hashes (SHA256 哈希)

https://gobyexample.com/sha256-hashes

63.1 核心概念

SHA256 (Secure Hash Algorithm 256-bit) 是一种密码学哈希函数，用于为数据（文本或二进制）生成一个唯一的、固定长度的摘要（哈希值）。在 Go 中，`crypto/sha256` 包提供了生成 SHA256 哈希的功能。

生活化类比：SHA256 哈希就像是为任何一份文件（无论是一句话还是一整本书）生成一个独一无二的“数字指纹”。这个指纹有固定的长度，且任何对原文的微小改动（哪怕只是一个标点符号）都会导致生成一个完全不同的指纹。你无法从指纹反推出原文，但可以用它来快速验证两份文件是否一模一样，常用于校验文件下载是否完整、密码存储等场景。

63.2 SHA256 哈希的计算步骤

计算哈希值的过程通常分为三步：

1. 创建一个新的哈希对象。
2. 向哈希对象中写入要计算的数据。
3. 计算最终的哈希值。

63.2.1 创建哈希对象

使用 `sha256.New()` 创建一个新的 `hash.Hash` 对象。

```
h := sha256.New()
```

63.2.2 写入数据

`hash.Hash` 对象实现了 `io.Writer` 接口，因此我们可以使用 `Write` 方法向其写入字节数据。通常需要将字符串转换为字节切片 `[]byte`。

```
s := "sha256 this string"
h.Write([]byte(s))
```

63.2.3 计算哈希值

`Sum` 方法用于计算并返回最终的哈希值。它接收一个可选的字节切片参数，用于将计算出的哈希值追加到该切片之后。通常我们传入 `nil`，表示只获取计算出的哈希值。

```
bs := h.Sum(nil)
```

返回的结果 `bs` 是一个字节切片。

63.2.4 格式化输出

为了方便阅读，通常将哈希值以十六进制字符串的形式显示。

```
fmt.Printf("%x\n", bs)
```

63.3 完整代码示例

```
package main

import (
    "crypto/sha256"
    "fmt"
)

func main() {
    s := "sha256 this string"

    // 1. 创建一个新的 sha256 哈希对象
    h := sha256.New()

    // 2. 写入要哈希的数据（作为字节）
    h.Write([]byte(s))

    // 3. 计算最终的哈希值
    // Sum 的参数用于将哈希值追加到现有的字节切片中，通常传入 nil
    bs := h.Sum(nil)

    fmt.Println(s)
    // 4. 将哈希值格式化为十六进制字符串
    fmt.Printf("%x\n", bs)
}
```

63.4 运行结果

```
$ go run sha256-hashes.go
sha256 this string
1af17a7568ff252c072599dd803f4a95685314b11375f7633465c68b5922932a
```

63.5 关键要点

1. **`crypto` 标准库**：Go 的所有加密和哈希功能都在 `crypto/*` 系列包中。
2. **`hash.Hash` 接口**：不同的哈希算法（如 SHA1, SHA512, MD5）都实现了 `hash.Hash` 接口，提供了统一的 `Write` 和 `Sum` 方法。
3. **数据流式处理**：由于 `hash.Hash` 实现了 `io.Writer`，你可以方便地将文件或其他数据流直接写入哈希对象，而无需将所有内容一次性读入内存。
4. **不可逆性**：哈希函数是单向的，意味着你无法从哈希值反推出原始数据。
5. **唯一性**：对于不同的输入，SHA256 产生不同哈希值的概率极高，因此常用于数据完整性校验。

64. Base64 Encoding (Base64 编码)

https://gobyexample.com/base64-encoding

64.1 核心概念

Base64 是一种将二进制数据编码为 ASCII 字符串的方法，以便在只支持文本的环境中传输。它通常用于在 HTTP、XML、JSON 等协议中嵌入二进制数据。Go 的 `encoding/base64` 包提供了标准的 Base64 编码和解码功能。

生活化类比：Base64 编码就像是把一些不方便直接邮寄的物品（比如液体、粉末等二进制数据）装进一个标准的、写满字母和数字的“快递盒”（ASCII 字符串）里。这样，任何只认识标准快递盒的邮政系统（只支持文本的协议）都能顺利传输它。收件人收到后，再打开盒子，把里面的东西还原出来（解码）。

64.2 Base64 编码与解码

64.2.1 标准 Base64 编码

`base64.StdEncoding` 提供了标准的 Base64 编码器。`EncodeToString` 方法接收一个字节切片并返回一个 Base64 编码的字符串。

```
data := "hello world"
sEnc := base64.StdEncoding.EncodeToString([]byte(data))
fmt.Println(sEnc) // 输出: aGVsbG8gd29ybGQ=
```

64.2.2 标准 Base64 解码

`DecodeString` 方法接收一个 Base64 编码的字符串，并返回解码后的字节切片和一个错误。

```
sDec, _ := base64.StdEncoding.DecodeString(sEnc)
fmt.Println(string(sDec)) // 输出: hello world
```

64.2.3 URL 兼容的 Base64 编码

标准的 Base64 编码包含 `+` 和 `/` 字符，这在 URL 中有特殊含义。因此，存在一种 URL 兼容的 Base64 编码方案，它使用 `-` 和 `_` 来代替。`base64.URLEncoding` 提供了这种编码器。

```
data := "hello+world/"
uEnc := base64.URLEncoding.EncodeToString([]byte(data))
fmt.Println(uEnc) // 输出: aGVsbG8rd29ybGQv
```

64.2.4 URL 兼容的 Base64 解码

解码过程与标准解码类似，使用 `URLEncoding` 的 `DecodeString` 方法。

```
uDec, _ := base64.URLEncoding.DecodeString(uEnc)
fmt.Println(string(uDec)) // 输出: hello+world/
```

64.3 完整代码示例

```
package main

import (
    "encoding/base64"
    "fmt"
)

func main() {
    data := "hello world"

    // 标准 Base64 编码
    sEnc := base64.StdEncoding.EncodeToString([]byte(data))
    fmt.Println(sEnc)

    // 标准 Base64 解码
    sDec, _ := base64.StdEncoding.DecodeString(sEnc)
    fmt.Println(string(sDec))

    // URL 兼容的 Base64 编码
    // 使用不同的字符集，使其在 URL 中安全
    uEnc := base64.URLEncoding.EncodeToString([]byte(data))
    fmt.Println(uEnc)

    // URL 兼容的 Base64 解码
    uDec, _ := base64.URLEncoding.DecodeString(uEnc)
    fmt.Println(string(uDec))
}
```

64.4 运行结果

```
$ go run base64-encoding.go
aGVsbG8gd29ybGQ=
hello world
aGVsbG8gd29ybGQ=
hello world
```

64.5 关键要点

1. **`encoding/base64` 包**：是 Go 中处理 Base64 编码的标准库。

2. **输入为字节切片**：编码和解码操作都围绕字节切片 `[]byte` 进行。
3. **两种编码方案**：`StdEncoding` (标准编码) 和 `URLEncoding` (URL安全编码) 是最常用的两种编码器。
4. **`EncodeToString` 和 `DecodeString`**：是进行字符串与 Base64 之间转换的核心方法。
5. **错误处理**：`DecodeString` 会在输入格式无效时返回错误，在实际应用中需要检查。

65. Reading Files (读取文件)

https://gobyexample.com/reading-files

65.1 核心概念

文件读取是 Go 中一项基本且常见的 I/O 操作。Go 提供了多种读取文件的方式，从简单地将整个文件读入内存，到更精细地控制读取过程，如分块读取和带缓冲的读取。

生活化类比：读取文件就像是阅读一本书。`os.ReadFile` 相当于你把整本书复印下来一次性看完，简单直接但书太厚（文件太大）时会很占地方（内存）。而 `os.Open` 则是你把书借到手，然后可以一页一页地读（`f.Read`），或者直接翻到某一页开始读（`f.Seek`），这种方式更灵活，也更节省精力（内存）。

65.2 文件读取方法

65.2.1 一次性读取整个文件

`os.ReadFile` 是最简单的文件读取方式，它将文件的全部内容一次性读入一个字节切片中。这对于读取小文件非常方便。

```
dat, err := os.ReadFile("/tmp/dat")
check(err)
fmt.Print(string(dat))
```

65.2.2 精细控制的读取

对于更复杂的读取需求，你可以先用 `os.Open` 打开文件，得到一个 `*os.File` 对象，然后进行更细粒度的操作。

```
f, err := os.Open("/tmp/dat")
check(err)
defer f.Close() // 使用 defer 确保文件在函数结束时被关闭
```

65.2.3 读取指定字节数

`Read` 方法可以从文件中读取指定数量的字节到一个切片中。它返回读取的字节数和可能发生的错误。

```
b1 := make([]byte, 5)
n1, err := f.Read(b1)
check(err)
fmt.Printf("%d bytes: %s\n", n1, string(b1[:n1]))
```

65.2.4 文件指针定位 (Seek)

`Seek` 方法可以在文件内移动读取指针。它接收一个偏移量和一个相对位置（`0`=文件开头, `1`=当前位置, `2`=文件末尾）。

```
o2, err := f.Seek(6, 0) // 移动到第6个字节处
check(err)
b2 := make([]byte, 2)
n2, err := f.Read(b2)
check(err)
fmt.Printf("%d bytes @ %d: %s\n", n2, o2, string(b2[:n2]))
```

65.2.5 `io` 包的辅助函数

`io` 包提供了一些有用的辅助函数，例如 `ReadAtLeast`，它可以保证至少读取指定数量的字节。

```
o3, err := f.Seek(6, 0)
check(err)
b3 := make([]byte, 2)
n3, err := io.ReadAtLeast(f, b3, 2)
check(err)
fmt.Printf("%d bytes @ %d: %s\n", n3, o3, string(b3))
```

65.2.6 带缓冲的读取

`bufio` 包提供了带缓冲的读取器，这对于需要进行多次小规模读取的场景可以提高效率。`Peek` 方法可以在不移动读取指针的情况下预读指定数量的字节。

```
_, err = f.Seek(0, 0) // 重置文件指针到开头
check(err)

r4 := bufio.NewReader(f)
b4, err := r4.Peek(5)
check(err)
fmt.Printf("5 bytes: %s\n", string(b4))
```

65.3 完整代码示例

```
package main

import (
    "bufio"
    "fmt"
    "io"
    "os"
)

// 定义一个辅助函数来检查错误
func check(e error) {
    if e != nil {
        panic(e)
```

```
    }
}

func main() {
    // 准备工作：创建一个临时文件用于演示
    // (在实际运行前，需要确保 /tmp/dat 文件存在且内容为 "hello go")

    // 1. 一次性读取整个文件
    dat, err := os.ReadFile("/tmp/dat")
    check(err)
    fmt.Print(string(dat), "\n")

    // 2. 打开文件以进行更细粒度的操作
    f, err := os.Open("/tmp/dat")
    check(err)
    defer f.Close() // 确保文件最后被关闭

    // 3. 读取指定字节
    b1 := make([]byte, 5)
    n1, err := f.Read(b1)
    check(err)
    fmt.Printf("%d bytes: %s\n", n1, string(b1[:n1]))

    // 4. 使用 Seek 移动文件指针并读取
    o2, err := f.Seek(6, 0)
    check(err)
    b2 := make([]byte, 2)
    n2, err := f.Read(b2)
    check(err)
    fmt.Printf("%d bytes @ %d: %s\n", n2, o2, string(b2[:n2]))

    // 5. 使用 io.ReadAtLeast 保证读取足够字节
    o3, err := f.Seek(6, 0)
    check(err)
    b3 := make([]byte, 2)
    n3, err := io.ReadAtLeast(f, b3, 2)
    check(err)
    fmt.Printf("%d bytes @ %d: %s\n", n3, o3, string(b3))

    // 6. 使用带缓冲的读取器
    _, err = f.Seek(0, 0)
    check(err)
    r4 := bufio.NewReader(f)
    b4, err := r4.Peek(5)
    check(err)
    fmt.Printf("5 bytes: %s\n", string(b4))
}
```

65.4 运行结果

```
# 首先创建一个演示文件
$ echo "hello go" > /tmp/dat

# 运行程序
$ go run reading-files.go
hello go
5 bytes: hello
2 bytes @ 6: go
2 bytes @ 6: go
5 bytes: hello
```

65.5 关键要点

1. **多种读取方式**：Go 提供了从简单到复杂的多种文件读取方法，以适应不同场景的需求。
2. **错误处理**：文件操作（如 `Read`, `Open`, `Seek`）几乎都会返回错误，必须进行检查。
3. **`defer` 关闭文件**：使用 `defer f.Close()` 是一个良好实践，能确保文件在函数退出时被可靠地关闭，防止资源泄漏。
4. **`io` 和 `bufio` 包**：`io` 包提供了 I/O 原语，而 `bufio` 包通过缓冲机制提高了 I/O 效率。
5. **文件指针**：`*os.File` 对象内部维护一个文件指针，`Read` 和 `Seek` 操作都会影响它的位置。

66. Writing Files (写入文件)

https://gobyexample.com/writing-files

66.1 核心概念

文件写入是 Go 中与文件读取相对应的基本 I/O 操作。Go 提供了多种将数据写入文件的方式，从简单地将字节切片一次性写入，到使用带缓冲的写入器进行高效写入。

生活化类比：写入文件就像是写日记。`os.WriteFile` 相当于你把一整天的经历一次性写在一张新纸上。而 `os.Create` 则是你先翻开新的一页，然后用 `f.WriteString` 一段一段地写。使用带缓冲的 `bufio.NewWriter` 就像是你先把草稿打在另一张纸上，然后一次性誊写到日记本里，这样可以减少反复翻开日记本的次数，更高效。最后 `w.Flush()` 就是确保草稿上的内容都抄完了。

66.2 文件写入方法

66.2.1 一次性写入整个文件

`os.WriteFile` 是最简单的文件写入方式。它接收文件名、要写入的字节切片和文件权限作为参数，然后将数据一次性写入文件。如果文件不存在，它会创建文件；如果文件已存在，它会清空文件内容再写入。

```
d1 := []byte("hello\ngo\n")
err := os.WriteFile("/tmp/dat1", d1, 0644)
check(err)
```

66.2.2 精细控制的写入

对于需要分步或更精细控制的写入操作，可以先用 `os.Create` 创建一个文件，得到一个 `*os.File` 对象。

```
f, err := os.Create("/tmp/dat2")
check(err)
defer f.Close() // 确保文件在函数结束时关闭
```

66.2.3 写入字节切片和字符串

`*os.File` 对象提供了 `Write` 和 `WriteString` 方法来写入数据。

```
// 写入字节切片
d2 := []byte{115, 111, 109, 101, 10} // "some\n"
n2, err := f.Write(d2)
check(err)

// 写入字符串
n3, err := f.WriteString("writes\n")
check(err)
```

66.2.4 同步到磁盘

调用 `Write` 或 `WriteString` 后，操作系统可能会将数据保留在内存缓冲区中。调用 `Sync` 方法可以确保将缓冲区中的数据刷新到稳定的存储设备（如磁盘）上。

```
f.Sync()
```

66.2.5 带缓冲的写入

`bufio.NewWriter` 提供了一个带缓冲的写入器，将多次小的写入操作合并为一次大的写入，可以提高性能。

```
w := bufio.NewWriter(f)
n4, err := w.WriteString("buffered\n")
check(err)
```

重要：使用带缓冲的写入器时，必须在最后调用 `Flush` 方法，以确保所有缓冲的数据都被写入到底层的文件中。

```
w.Flush()
```

66.3 完整代码示例

```
package main

import (
    "bufio"
    "fmt"
    "os"
)

func check(e error) {
    if e != nil {
```

```
        panic(e)
    }
}

func main() {
    // 1. 一次性写入字节切片到文件
    d1 := []byte("hello\ngo\n")
    err := os.WriteFile("/tmp/dat1", d1, 0644)
    check(err)

    // 2. 创建文件以进行更细粒度的写入
    f, err := os.Create("/tmp/dat2")
    check(err)
    defer f.Close()

    // 3. 写入字节切片
    d2 := []byte{115, 111, 109, 101, 10} // "some\n"
    n2, err := f.Write(d2)
    check(err)
    fmt.Printf("wrote %d bytes\n", n2)

    // 4. 写入字符串
    n3, err := f.WriteString("writes\n")
    check(err)
    fmt.Printf("wrote %d bytes\n", n3)

    // 5. 调用 Sync 将数据刷新到磁盘
    f.Sync()

    // 6. 使用带缓冲的写入器
    w := bufio.NewWriter(f)
    n4, err := w.WriteString("buffered\n")
    check(err)
    fmt.Printf("wrote %d bytes\n", n4)

    // 7. 刷新缓冲区，确保所有数据都已写入
    w.Flush()
}
```

66.4 运行结果

```
# 运行程序
$ go run writing-files.go
wrote 5 bytes
wrote 7 bytes
wrote 9 bytes

# 检查第一个文件的内容
$ cat /tmp/dat1
hello
go
```

```
# 检查第二个文件的内容
$ cat /tmp/dat2
some
writes
buffered
```

66.5 关键要点

1. **多种写入方式**：Go 提供了从简单的一次性写入到带缓冲的精细写入等多种方式。
2. **`os.WriteFile`**：是写入小文件的最简单方法。
3. **`os.Create`**：用于创建文件并获取 `*os.File` 对象以进行更复杂的操作。
4. **`Sync` 和 `Flush`**：`Sync` 用于将文件系统的缓冲区刷新到磁盘，而 `Flush` 用于将 `bufio` 的缓冲区刷新到底层写入器。使用缓冲写入时，`Flush` 是必不可少的。
5. **`defer f.Close()`**：与文件读取一样，这是确保文件被关闭、资源被释放的关键实践。

67. Line Filters (行过滤器)

https://gobyexample.com/line-filters

67.1 核心概念

行过滤器（Line Filter）是一种常见的程序类型，它读取标准输入（stdin）的内容，对其进行处理，然后将结果打印到标准输出（stdout）。在 Go 中，可以利用 `bufio` 包的 `Scanner` 来轻松实现行过滤器，这在处理文本流或与其他命令行工具组合使用时非常有用。

生活化类比：行过滤器就像一个加工站。原料（文本行）从传送带的一端（标准输入）进来，经过加工站的处理（比如把所有字母都变成大写），然后成品从传送带的另一端（标准输出）出去。这种模式的强大之处在于，你可以把多个不同的加工站串联起来（通过管道 `|`），形成一条流水线，对数据进行多重处理。

67.2 实现行过滤器

67.2.1 从标准输入读取

`bufio.NewScanner(os.Stdin)` 创建一个扫描器，它会从程序的标准输入中读取数据。

```
scanner := bufio.NewScanner(os.Stdin)
```

67.2.2 逐行扫描

`scanner.Scan()` 方法会在读取到新的一行时返回 `true`，在没有更多行可读时返回 `false`。这使得我们可以用一个 `for` 循环来逐行处理输入。

```
for scanner.Scan() {
    // ... process the line
}
```

67.2.3 处理每一行

在循环内部，`scanner.Text()` 返回当前行的内容（作为字符串）。我们可以对其进行任意处理，例如转换为大写。

```
ucl := strings.ToUpper(scanner.Text())
fmt.Println(ucl) // 将处理结果打印到标准输出
```

67.2.4 错误处理

在扫描结束后，应该调用 `scanner.Err()` 来检查在扫描过程中是否发生了错误（例如，读取错误）。

```
if err := scanner.Err(); err != nil {
    fmt.Fprintln(os.Stderr, "error:", err)
    os.Exit(1)
}
```

67.3 完整代码示例

```
package main

import (
    "bufio"
    "fmt"
    "os"
    "strings"
)

func main() {
    // 创建一个从标准输入读取的扫描器
    scanner := bufio.NewScanner(os.Stdin)

    // 逐行扫描输入
    for scanner.Scan() {
        // 将当前行转换为大写
        ucl := strings.ToUpper(scanner.Text())
        // 将结果打印到标准输出
        fmt.Println(ucl)
    }

    // 检查扫描过程中是否出错
    if err := scanner.Err(); err != nil {
        fmt.Fprintln(os.Stderr, "error:", err)
        os.Exit(1)
    }
}
```

67.4 运行结果

行过滤器通常与管道（`|`）结合使用，将一个命令的输出作为另一个命令的输入。

```
# 准备一个包含多行文本的文件
$ echo 'hello' > /tmp/lines
$ echo 'filter' >> /tmp/lines

# 使用管道将文件内容发送到我们的行过滤器程序
$ cat /tmp/lines | go run line-filters.go
HELLO
FILTER
```

67.5 关键要点

1. **标准 I/O**：行过滤器是围绕标准输入（stdin）、标准输出（stdout）和标准错误（stderr）构建的，这是 Unix/Linux 命令行哲学的核心。
2. **`bufio.Scanner`**：是处理行文本流的强大工具，它能高效地处理输入并按行分割。
3. **管道（Piping）**：行过滤器最大的威力在于它可以作为管道的一部分，与其他命令（如 `cat`, `grep`, `sed`）组合，形成强大的数据处理工作流。
4. **错误检查**：即使在简单的行过滤器中，检查 `scanner.Err()` 也是一个好习惯，可以确保程序在遇到 I/O 问题时能正确报告错误。

68. File Paths (文件路径)

https://gobyexample.com/file-paths

68.1 核心概念

在不同操作系统上（如 Windows 和 Linux），文件路径的表示方式不同（例如，分隔符分别是 `\` 和 `/`）。为了编写可移植的代码，Go 的 `path/filepath` 包提供了一系列函数来处理文件路径，使其与操作系统无关。

生活化类比：`filepath` 包就像一个专业的地址格式化工具。你知道一个地址由“国家”、“城市”、“街道”几部分组成，但不同国家的书写习惯不同。你只需要把这些部分告诉 `filepath.Join`，它就能根据你当前所在的“国家”（操作系统），自动用正确的连接符（`/` 或 `\`）帮你生成一个标准格式的地址，确保你的信能寄到任何地方。

68.2 文件路径操作

68.2.1 构造路径

`filepath.Join` 是一个安全且可移植的函数，用于将多个路径部分连接成一个单一的路径。它会自动处理正确的分隔符，并清理多余的分隔符和 `..` 等。

```
p := filepath.Join("dir1", "dir2", "filename")
fmt.Println("p:", p) // 在 Linux/macOS 上输出: dir1/dir2/filename

// Join 会自动处理多余的分隔符
fmt.Println(filepath.Join("dir1//", "filename")) // 输出: dir1/filename

// Join 还会计算和简化路径
fmt.Println(filepath.Join("dir1/../dir1", "filename")) // 输出: dir1/filename
```

68.2.2 分解路径

`filepath.Dir` 和 `filepath.Base` 分别用于获取路径中的目录部分和文件名部分。

```
fmt.Println("Dir(p):", filepath.Dir(p))   // 输出: dir1/dir2
fmt.Println("Base(p):", filepath.Base(p))  // 输出: filename
```

68.2.3 判断绝对路径

`filepath.IsAbs` 用于检查一个路径是否是绝对路径。

```
fmt.Println(filepath.IsAbs("dir/file")) // 输出: false
fmt.Println(filepath.IsAbs("/dir/file")) // 输出: true
```

68.2.4 获取文件扩展名

`filepath.Ext` 用于提取路径中文件名的扩展名。

```
filename := "config.json"
ext := filepath.Ext(filename)
fmt.Println(ext) // 输出: .json
```

要获取不带扩展名的文件名，可以结合使用 `strings.TrimSuffix`。

```
fmt.Println(strings.TrimSuffix(filename, ext)) // 输出: config
```

68.2.5 计算相对路径

`filepath.Rel` 用于计算一个目标路径相对于一个基础路径的相对路径。

```
rel, err := filepath.Rel("a/b", "a/b/t/file")
check(err)
fmt.Println(rel) // 输出: t/file

rel, err = filepath.Rel("a/b", "a/c/t/file")
check(err)
fmt.Println(rel) // 输出: ../c/t/file
```

68.3 完整代码示例

```go
package main

import (
    "fmt"
    "path/filepath"
    "strings"
)

func main() {
    // 使用 Join 构造路径
    p := filepath.Join("dir1", "dir2", "filename")
    fmt.Println("p:", p)

    // Join 会自动处理多余的分隔符和 ..
    fmt.Println(filepath.Join("dir1//", "filename"))
    fmt.Println(filepath.Join("dir1/../dir1", "filename"))

    // Dir 和 Base 用于分解路径
    fmt.Println("Dir(p):", filepath.Dir(p))
    fmt.Println("Base(p):", filepath.Base(p))

    // IsAbs 判断是否为绝对路径
    fmt.Println(filepath.IsAbs("dir/file"))
    fmt.Println(filepath.IsAbs("/dir/file"))

    // Ext 获取文件扩展名
    filename := "config.json"
    ext := filepath.Ext(filename)
    fmt.Println(ext)

    // 获取不带扩展名的文件名
    fmt.Println(strings.TrimSuffix(filename, ext))

    // Rel 计算相对路径
    rel, err := filepath.Rel("a/b", "a/b/t/file")
    if err != nil {
        panic(err)
    }
    fmt.Println(rel)

    rel, err = filepath.Rel("a/b", "a/c/t/file")
    if err != nil {
        panic(err)
    }
    fmt.Println(rel)
}
```

68.4 运行结果

```
$ go run file-paths.go
p: dir1/dir2/filename
dir1/filename
dir1/filename
Dir(p): dir1/dir2
Base(p): filename
false
true
.json
config
t/file
../c/t/file
```

68.5 关键要点

1. **可移植性**：`path/filepath` 包是编写跨平台文件操作代码的关键，它能自动处理不同操作系统的路径分隔符和规则。
2. **`Join` 优于手动拼接**：应始终使用 `filepath.Join` 来构造路径，而不是手动用 `+` 和 `/` 或 `\` 来拼接字符串，这样更安全、更可移植。
3. **路径分解**：`Dir` 和 `Base` 是解析路径、获取目录和文件名的标准方法。
4. **扩展名处理**：`Ext` 和 `strings.TrimSuffix` 组合是获取文件名和扩展名的常用模式。
5. **相对路径计算**：`Rel` 在需要计算两个路径之间的相对关系时非常有用，例如在生成配置文件或构建系统中。

69. Directories (目录操作)

https://gobyexample.com/directories

69.1 核心概念

在 Go 中，`os` 包提供了一系列与操作系统交互的函数，包括创建、读取和删除目录。这些功能是编写能够管理文件系统结构的程序的关键。

生活化类比：目录操作就像是整理你的文件柜。`os.Mkdir` 是在文件柜里新建一个抽屉；`os.ReadDir` 是打开一个抽屉看看里面有哪些文件夹；`os.RemoveAll` 则是把整个抽屉连同里面的所有东西都扔掉。而 `filepath.WalkDir` 就像一个勤劳的档案管理员，他会从你指定的抽屉开始，一层一层地深入，把所有抽屉里的所有文件夹和文件都清点一遍。

69.2 目录操作

69.2.1 创建目录

`os.Mkdir` 用于创建一个新目录。它接收目录名和文件权限作为参数。`0755` 是一个常见的目录权限设置。

```
err := os.Mkdir("subdir", 0755)
check(err)
```

`os.MkdirAll` 则可以递归地创建多层目录（类似于 `mkdir -p`）。

```
err = os.MkdirAll("subdir/parent/child", 0755)
check(err)
```

69.2.2 清理目录

使用 `defer` 和 `os.RemoveAll` 是一个常见的模式，用于在程序结束时清理创建的临时目录。`os.RemoveAll` 会删除整个目录树（类似于 `rm -rf`）。

```
defer os.RemoveAll("subdir")
```

69.2.3 读取目录内容

`os.ReadDir` 用于读取一个目录的内容。它返回一个 `[]os.DirEntry` 切片，其中包含了目录中的文件和子目录信息。

```
c, err := os.ReadDir("subdir/parent")
check(err)

fmt.Println("Listing subdir/parent")
for _, entry := range c {
    fmt.Println(" ", entry.Name(), entry.IsDir())
}
```

69.2.4 切换当前目录

`os.Chdir` 用于改变当前的G作目录（类似于 `cd` 命令）。

```
err = os.Chdir("subdir/parent/child")
check(err)
```

69.2.5 递归遍历目录

`filepath.WalkDir` 是一个非常强大的函数，用于递归地遍历一个目录树。它接收一个起始路径和一个访问函数（回调函数）作为参数。每当 `WalkDir` 访问到一个文件或目录时，它都会调用这个访问函数。

```
func visit(path string, d fs.DirEntry, err error) error {
    if err != nil {
        return err
    }
    fmt.Println(" ", path, d.IsDir())
    return nil
}

fmt.Println("Visiting subdir")
err = filepath.WalkDir("subdir", visit)
check(err)
```

69.3 完整代码示例

```go
package main

import (
    "fmt"
    "io/fs"
    "os"
    "path/filepath"
)

func check(e error) {
    if e != nil {
        panic(e)
    }
}

func main() {
    // 创建一个子目录
    err := os.Mkdir("subdir", 0755)
    check(err)

    // 使用 defer 在程序结束时清理目录
    defer os.RemoveAll("subdir")

    // 辅助函数：创建一个空文件
    createEmptyFile := func(name string) {
        d := []byte("")
        check(os.WriteFile(name, d, 0644))
    }

    createEmptyFile("subdir/file1")

    // 递归创建多层目录
    err = os.MkdirAll("subdir/parent/child", 0755)
    check(err)

    createEmptyFile("subdir/parent/file2")
    createEmptyFile("subdir/parent/file3")
    createEmptyFile("subdir/parent/child/file4")

    // 读取目录内容
    c, err := os.ReadDir("subdir/parent")
    check(err)

    fmt.Println("Listing subdir/parent")
    for _, entry := range c {
        fmt.Println(" ", entry.Name(), entry.IsDir())
    }

    // 切换当前工作目录
    err = os.Chdir("subdir/parent/child")
    check(err)
```

```
    // 读取当前目录（现在是 child）的内容
    c, err = os.ReadDir(".")
    check(err)

    fmt.Println("Listing .")
    for _, entry := range c {
        fmt.Println(" ", entry.Name(), entry.IsDir())
    }

    // 切换回初始目录
    err = os.Chdir("../../..")
    check(err)

    // 递归遍历目录
    fmt.Println("Visiting subdir")
    err = filepath.WalkDir("subdir", visit)
    check(err)
}

// WalkDir 的访问函数
func visit(path string, d fs.DirEntry, err error) error {
    if err != nil {
        return err
    }
    fmt.Println(" ", path, d.IsDir())
    return nil
}
```

69.4 运行结果

```
$ go run directories.go
Listing subdir/parent
  child true
  file2 false
  file3 false
Listing .
  file4 false
Visiting subdir
  subdir true
  subdir/file1 false
  subdir/parent true
  subdir/parent/child true
  subdir/parent/child/file4 false
  subdir/parent/file2 false
  subdir/parent/file3 false
```

69.5 关键要点

1. **`os` 包**：提供了与操作系统交互的基本功能，包括目录的创建、读取和删除。
2. **`Mkdir` vs `MkdirAll`**：`Mkdir` 用于创建单个目录，而 `MkdirAll` 可以像 `mkdir -p` 一样创建完整的路径。

3. **`defer` 与 `RemoveAll`**：这是一个管理临时目录的常用且可靠的模式，确保在程序退出时进行清理。
4. **`ReadDir`**：用于非递归地列出单个目录的内容。
5. **`filepath.WalkDir`**：是递归遍历整个目录树的强大工具，通过回调函数对每个文件和目录进行操作，非常灵活。

70. Temporary Files and Directories (临时文件和目录)

https://gobyexample.com/temporary-files-and-directories

70.1 核心概念

在程序执行过程中，有时需要创建一些临时文件或目录来存储中间数据，这些数据在程序结束后就不再需要了。Go 的 `os` 包提供了创建临时文件和目录的功能，可以确保它们的名字是唯一的，并能方便地进行清理。

生活化类比：临时文件和目录就像是你在做手工时用的草稿纸和临时工具盒。`os.CreateTemp` 会给你一张干净的、保证不会和别人混淆的草稿纸。`defer os.Remove` 则是一个好习惯，相当于你对自己说："等我做完手工，一定记得把这张废纸扔进垃圾桶"，从而保持工作台的整洁。

70.2 临时文件和目录的操作

70.2.1 创建临时文件

`os.CreateTemp` 用于在系统默认的临时目录中（如 `/tmp`）创建一个新的临时文件。它返回一个 `*os.File` 对象和一个错误。

- 第一个参数是临时文件所在的目录。如果为空字符串 `""`，则使用系统默认临时目录。
- 第二个参数是文件名的前缀。

```
f, err := os.CreateTemp("", "sample")
check(err)
fmt.Println("Temp file name:", f.Name())
```

70.2.2 清理临时文件

创建临时文件后，最佳实践是使用 `defer` 和 `os.Remove` 来确保在程序退出时删除该文件。

```
defer os.Remove(f.Name())
```

70.2.3 写入临时文件

返回的 `*os.File` 对象可以像普通文件一样进行读写操作。

```
_, err = f.Write([]byte{1, 2, 3, 4})
check(err)
```

70.2.4 创建临时目录

`os.MkdirTemp` 的工作方式与 `CreateTemp` 类似，但它创建的是一个目录。它返回创建的目录名和一个错误。

```
dname, err := os.MkdirTemp("", "sampledir")
check(err)
fmt.Println("Temp dir name:", dname)
```

70.2.5 清理临时目录

对于临时目录，应使用 `os.RemoveAll` 来递归删除目录及其所有内容。

```
defer os.RemoveAll(dname)
```

70.3 完整代码示例

```
package main

import (
    "fmt"
    "os"
    "path/filepath"
)

func check(e error) {
    if e != nil {
        panic(e)
    }
}

func main() {
    // 1. 创建一个临时文件
    f, err := os.CreateTemp("", "sample")
    check(err)

    // 打印临时文件的路径
    fmt.Println("Temp file name:", f.Name())

    // 2. 使用 defer 确保在程序结束时清理临时文件
    defer os.Remove(f.Name())

    // 3. 向临时文件写入数据
    _, err = f.Write([]byte{1, 2, 3, 4})
    check(err)

    // 4. 创建一个临时目录
    dname, err := os.MkdirTemp("", "sampledir")
    check(err)
    fmt.Println("Temp dir name:", dname)

    // 5. 使用 defer 确保清理临时目录及其内容
    defer os.RemoveAll(dname)

    // 6. 在临时目录中创建一个文件
```

```
    fname := filepath.Join(dname, "file1")
    err = os.WriteFile(fname, []byte{1, 2}, 0666)
    check(err)
}
```

70.4 运行结果

```
$ go run temporary-files-and-directories.go
Temp file name: /tmp/sample2280923705
Temp dir name: /tmp/sampledir3334999422
```

(注：每次运行时，操作系统都会选择不同的随机名称)

70.5 关键要点

1. **`os.CreateTemp` 和 `os.MkdirTemp`**：是创建唯一的临时文件和目录的标准方法。
2. **自动命名**：这些函数会自动在指定的前缀后附加一个随机数，以保证名称的唯一性，避免冲突。
3. **清理是关键**：操作系统最终可能会清理临时目录，但最佳实践是使用 `defer` 和 `os.Remove` 或 `os.RemoveAll` 在程序退出时立即进行清理。
4. **系统默认位置**：将目录参数设置为空字符串 `""` 会在系统的标准临时位置（如 `/tmp` 或 `C:\TEMP`）创建文件或目录。

71. Embed Directive (嵌入指令)

https://gobyexample.com/embed-directive

71.1 核心概念

Go 1.16 引入了一个新的编译器指令 `//go:embed`，它允许在编译时将文件内容直接嵌入到 Go 二进制文件中。这对于分发包含静态资源（如配置文件、模板、Web 静态文件）的独立应用程序非常有用，因为它避免了在运行时依赖外部文件。

生活化类比：`//go:embed` 就像是你在打包行李（编译程序）时，直接把旅游指南（静态文件）塞进了你的背包（二进制文件）里。这样，无论你走到哪里，你只需要带上这个背包就行了，而不需要再额外拿着一本书。你的程序变得“自给自足”，不再依赖于外部的文件是否存在。

71.2 使用 `//go:embed`

要使用 `embed` 功能，首先需要导入 `embed` 包。

```
import "embed"
```

71.2.1 嵌入单个文件到字符串或字节切片

你可以将单个文件的内容嵌入到一个 `string` 或 `[]byte` 变量中。

```go
//go:embed folder/single_file.txt
var fileString string

//go:embed folder/single_file.txt
var fileByte []byte
```

- `//go:embed` 指令必须紧跟在 `var` 声明之前。
- 路径是相对于 Go 源文件的相对路径。

71.2.2 嵌入多个文件或整个目录

要嵌入多个文件或整个目录，你需要使用 `embed.FS` 类型。这会创建一个虚拟的文件系统。

```go
// 使用多条指令嵌入单个文件和通配符匹配的文件
//go:embed folder/single_file.txt
//go:embed folder/*.hash
var folder embed.FS
```

`embed.FS` 实现了 `fs.FS` 接口，这意味着你可以像操作普通文件系统一样使用它，例如读取文件、列出目录等。

71.2.3 从 `embed.FS` 读取文件

`embed.FS` 提供了 `ReadFile` 方法来读取嵌入的文件内容。

```go
content1, _ := folder.ReadFile("folder/file1.hash")
fmt.Print(string(content1))
```

71.3 完整代码示例

文件结构:

```
.
├── embed-directive.go
└── folder
    ├── file1.hash
    ├── file2.hash
    └── single_file.txt
```

代码:

```go
package main

import (
    "embed"
    "fmt"
)

// 将文件内容嵌入到 string
```

```go
//go:embed folder/single_file.txt
var fileString string

// 将文件内容嵌入到 []byte
//go:embed folder/single_file.txt
var fileByte []byte

// 将多个文件嵌入到一个虚拟文件系统
//go:embed folder/single_file.txt
//go:embed folder/*.hash
var folder embed.FS

func main() {
    // 直接访问嵌入的 string 和 []byte
    fmt.Print(fileString)
    fmt.Print(string(fileByte))

    // 从虚拟文件系统中读取文件
    content1, _ := folder.ReadFile("folder/file1.hash")
    fmt.Print(string(content1))

    content2, _ := folder.ReadFile("folder/file2.hash")
    fmt.Print(string(content2))
}
```

71.4 运行结果

```
# 准备文件和目录
$ mkdir -p folder
$ echo "hello go" > folder/single_file.txt
$ echo "123" > folder/file1.hash
$ echo "456" > folder/file2.hash

# 运行程序
$ go run embed-directive.go
hello go
hello go
123
456
```

71.5 关键要点

1. **编译器指令**：`//go:embed` 是一个在编译时执行的指令，而不是运行时代码。
2. **静态资源打包**：`embed` 的主要用途是将静态资源（如 HTML, CSS, JS, SQL 文件）直接打包到二进制文件中，简化部署。
3. **三种嵌入方式**：可以将文件嵌入到 `string`、`[]byte` 或 `embed.FS` 中。
4. **`embed.FS`**：用于嵌入多个文件或目录，提供了一个只读的虚拟文件系统接口。
5. **路径规则**：嵌入的路径是相对于源文件的，不能包含 `.` 或 `..`，也不能是绝对路径。

72. Testing and Benchmarking (测试与基准测试)

https://gobyexample.com/testing-and-benchmarking

72.1 核心概念

Go 语言内置了强大的测试和基准测试工具，通过 `go test` 命令和 `testing` 包来支持。编写测试是保证代码质量和稳定性的关键环节，而基准测试则有助于理解和优化代码性能。

生活化类比：单元测试就像是给你的代码进行一次“体检”。你写的每一个函数都是一个“器官”，测试函数就是对应的检查项目，确保这个器官功能正常。而基准测试则更像是“体能测试”，它不关心功能对错，而是反复测量你的代码跑得到底有多快，帮你找到性能瓶颈。

72.2 单元测试 (Unit Testing)

单元测试用于验证代码中最小的功能单元（通常是函数）是否按预期工作。

72.2.1 测试文件和函数命名

- **文件名**: 测试代码必须放在与源文件同目录下的 `_test.go` 文件中。例如，`intutils.go` 的测试文件是 `intutils_test.go`。
- **函数签名**: 测试函数必须以 `Test` 开头，并接收一个 `*testing.T` 类型的参数。例如：`func TestIntMinBasic(t *testing.T)`。

72.2.2 报告失败

`*testing.T` 类型提供了多种报告失败的方法：

- `t.Error` 或 `t.Errorf`：报告失败，但会继续执行当前测试函数中的剩余代码。
- `t.Fatal` 或 `t.Fatalf`：报告失败，并立即停止当前测试函数的执行。

72.2.3 表驱动测试 (Table-Driven Tests)

这是一种常见的 Go 测试模式，通过将测试用例定义在一个结构体切片中，可以方便地管理和扩展测试用例。

```
var tests = []struct {
    a, b int
    want int
}{
    {0, 1, 0},
    {1, 0, 0},
}

for _, tt := range tests {
    // ...
}
```

使用 `t.Run` 可以在一个测试函数中创建多个子测试，使得测试输出更清晰，也更容易定位失败的用例。

72.3 基准测试 (Benchmarking)

基准测试用于衡量特定代码段的性能。

- **函数签名**: 基准测试函数必须以 `Benchmark` 开头，并接收一个 `*testing.B` 类型的参数。例如：`func BenchmarkIntMin(b *testing.B)`。
- **`b.N`**: 基准测试函数中的代码会运行 `b.N` 次。`go test` 会自动调整 `b.N` 的值，直到测量结果足够稳定。

```
func BenchmarkIntMin(b *testing.B) {
    for i := 0; i < b.N; i++ {
        IntMin(1, 2)
    }
}
```

72.4 示例代码

源文件: `intutils.go`

```
package main

// IntMin 返回两个整数中较小的一个。
func IntMin(a, b int) int {
    if a < b {
        return a
    }
    return b
}
```

测试文件: `intutils_test.go`

```
package main

import (
    "fmt"
    "testing"
)

// 一个基本的测试函数
func TestIntMinBasic(t *testing.T) {
    ans := IntMin(2, -2)
    if ans != -2 {
        t.Errorf("IntMin(2, -2) = %d; want -2", ans)
    }
}

// 表驱动测试
func TestIntMinTableDriven(t *testing.T) {
    var tests = []struct {
        a, b int
        want int
    }{
        {0, 1, 0},
```

```
        {1, 0, 0},
        {2, -2, -2},
        {0, -1, -1},
        {-1, 0, -1},
    }

    for _, tt := range tests {
        testname := fmt.Sprintf("%d,%d", tt.a, tt.b)
        t.Run(testname, func(t *testing.T) {
            ans := IntMin(tt.a, tt.b)
            if ans != tt.want {
                t.Errorf("got %d, want %d", ans, tt.want)
            }
        })
    }
}

// 基准测试函数
func BenchmarkIntMin(b *testing.B) {
    for i := 0; i < b.N; i++ {
        IntMin(1, 2)
    }
}
```

72.5 运行测试和基准测试

运行单元测试:
使用 `go test -v` 命令运行测试并显示详细结果。

```
$ go test -v intutils.go intutils_test.go
=== RUN   TestIntMinBasic
--- PASS: TestIntMinBasic (0.00s)
=== RUN   TestIntMinTableDriven
=== RUN   TestIntMinTableDriven/0,1
=== RUN   TestIntMinTableDriven/1,0
=== RUN   TestIntMinTableDriven/2,-2
=== RUN   TestIntMinTableDriven/0,-1
=== RUN   TestIntMinTableDriven/-1,0
--- PASS: TestIntMinTableDriven (0.00s)
    --- PASS: TestIntMinTableDriven/0,1 (0.00s)
    --- PASS: TestIntMinTableDriven/1,0 (0.00s)
    --- PASS: TestIntMinTableDriven/2,-2 (0.00s)
    --- PASS: TestIntMinTableDriven/0,-1 (0.00s)
    --- PASS: TestIntMinTableDriven/-1,0 (0.00s)
PASS
ok      command-line-arguments  0.008s
```

运行基准测试:
使用 `go test -bench=.` 命令运行基准测试。

```
$ go test -bench=. intutils.go intutils_test.go
goos: darwin
goarch: amd64
cpu: Intel(R) Core(TM) i7-9750H CPU @ 2.60GHz
BenchmarkIntMin-12          1000000000               0.2752 ns/op
PASS
ok      command-line-arguments  0.751s
```

72.6 关键要点

1. **约定优于配置**：Go 的测试工具依赖于文件和函数的命名约定（`_test.go`，`TestXxx`，`BenchmarkXxx`）。
2. **`testing` 包**：提供了测试所需的核心工具，如 `*testing.T` 和 `*testing.B`。
3. **表驱动测试**：是 Go 社区推荐的一种编写可维护、可扩展测试的模式。
4. **`go test` 命令**：是执行测试和基准测试的入口，提供了丰富的标志（如 `-v`，`-bench`）来控制执行和输出。
5. **性能测量**：基准测试提供了一种标准化的方法来评估代码性能，并能发现性能退化问题。

73. Command-Line Arguments (命令行参数)

https://gobyexample.com/command-line-arguments

73.1 核心概念

命令行参数是在执行程序时，在程序名之后传递给程序的一系列字符串。这是一种向程序提供基本输入的常见方式。在 Go 中，可以通过 `os.Args` 变量来访问这些参数。

生活化类比：命令行参数就像是你点外卖时给的备注。你运行一个程序（点餐），然后在后面跟上一串“备注”（参数），比如“多加辣”、“不要香菜”。程序（餐厅）接收到这些备注后，就可以根据你的要求来定制它的行为（菜的口味）。

73.2 访问命令行参数

`os.Args` 是一个字符串切片（`[]string`），它包含了程序启动时的所有命令行参数。

73.2.1 `os.Args` 的结构

- `os.Args[0]`：切片的第一个元素始终是程序的路径或名称。
- `os.Args[1:]`：切片的其余部分包含了用户传递给程序的实际参数。

73.2.2 示例

```
package main

import (
    "fmt"
    "os"
)

func main() {
```

```
    // os.Args 包含了程序路径和所有参数
    argsWithProg := os.Args
    fmt.Println(argsWithProg)

    // os.Args[1:] 只包含传递给程序的参数
    argsWithoutProg := os.Args[1:]
    fmt.Println(argsWithoutProg)

    // 可以通过索引访问单个参数
    // 注意：如果索引超出范围，程序会 panic
    if len(os.Args) > 3 {
        arg := os.Args[3]
        fmt.Println(arg)
    }
}
```

73.3 运行示例

要测试命令行参数，你需要先使用 `go build` 构建程序，然后直接运行生成的可执行文件并附带参数。

```
# 1. 构建程序
$ go build command-line-arguments.go

# 2. 运行可执行文件并传递参数 "a", "b", "c","d"
$ ./command-line-arguments a b c d
[./command-line-arguments a b c d]
[a b c d]
c
```

73.4 关键要点

1. **`os.Args`**：是访问所有命令行参数的入口，它是一个 `[]string` 类型的切片。
2. **程序名在首位**：`os.Args[0]` 总是程序的名称，实际的用户参数从 `os.Args[1]` 开始。
3. **参数是字符串**：所有命令行参数都被读取为字符串。如果需要数字或其他类型，你需要手动进行解析（例如，使用 `strconv` 包）。
4. **需要构建后运行**：与 `go run` 不同，要测试传递参数，通常需要先 `go build` 生成可执行文件，然后从终端运行它。
5. **更高级的解析**：对于更复杂的命令行接口（例如，带标志 `-port=8080` 或 `--verbose`），推荐使用 `flag` 包，它提供了更强大和结构化的解析功能。

74. Command-Line Flags (命令行标志)

https://gobyexample.com/command-line-flags

74.1 核心概念

命令行标志（Command-Line Flags）是向命令行程序传递键值对选项的常用方式。例如，在 `go run -v` 中，`-v` 就是一个标志。Go 的 `flag` 包提供了对命令行标志的强大支持，可以方便地定义和解析它们。

生活化类比：命令行标志就像是操作一台机器时的各种调节旋钮。机器本身（主命令）功能是固定的，但你可以通过调节 `-speed=100` 或者打开 `-verbose` 开关来微调某一次运行的具体行为，而无需改变机器的内部设计。

74.2 使用 `flag` 包

74.2.1 声明标志

`flag` 包提供了多种函数来声明不同类型的标志。这些函数通常接收标志名称、默认值和帮助信息作为参数，并返回一个指向该值的指针。

- **`flag.String`**: 声明一个字符串标志。
- **`flag.Int`**: 声明一个整数标志。
- **`flag.Bool`**: 声明一个布尔标志。

```
// 声明一个名为 "word" 的字符串标志，默认值为 "foo"
wordPtr := flag.String("word", "foo", "a string")

// 声明整数和布尔标志
numbPtr := flag.Int("numb", 42, "an int")
boolPtr := flag.Bool("fork", false, "a bool")
```

你也可以将标志绑定到一个已有的变量上，这时需要使用 `flag.StringVar`、`flag.IntVar` 等函数。

```
var svar string
// 将名为 "svar" 的标志绑定到 svar 变量上
flag.StringVar(&svar, "svar", "bar", "a string var")
```

74.2.2 解析标志

在声明完所有标志后，必须调用 `flag.Parse()` 来从命令行参数中解析它们。

```
flag.Parse()
```

74.2.3 访问标志的值

解析后，你可以通过解引用指针（对于 `flag.String` 等函数返回的指针）或直接访问变量（对于 `flag.StringVar` 等绑定的变量）来获取标志的值。

```
fmt.Println("word:", *wordPtr)
fmt.Println("svar:", svar)
```

74.2.4 访问非标志参数

`flag.Args()` 返回一个包含所有非标志参数（也称为尾随参数）的字符串切片。

```
fmt.Println("tail:", flag.Args())
```

74.3 完整代码示例

```
package main

import (
    "flag"
    "fmt"
)

func main() {
    // 声明标志，返回指针
    wordPtr := flag.String("word", "foo", "a string")
    numbPtr := flag.Int("numb", 42, "an int")
    boolPtr := flag.Bool("fork", false, "a bool")

    // 将标志绑定到现有变量
    var svar string
    flag.StringVar(&svar, "svar", "bar", "a string var")

    // 解析命令行参数
    flag.Parse()

    // 访问标志的值
    fmt.Println("word:", *wordPtr)
    fmt.Println("numb:", *numbPtr)
    fmt.Println("fork:", *boolPtr)
    fmt.Println("svar:", svar)

    // 访问非标志参数
    fmt.Println("tail:", flag.Args())
}
```

74.4 运行示例

与命令行参数一样，测试标志也需要先构建程序。

1. 提供所有标志的值

```
$ go build command-line-flags.go
$ ./command-line-flags -word=opt -numb=7 -fork -svar=flag
word: opt
numb: 7
fork: true
svar: flag
tail: []
```

2. 省略部分标志（使用默认值）

```
$ ./command-line-flags -word=opt
word: opt
numb: 42
fork: false
svar: bar
tail: []
```

3. 包含尾随参数

```
$ ./command-line-flags -word=opt a1 a2 a3
word: opt
numb: 42
fork: false
svar: bar
tail: [a1 a2 a3]
```

4. 标志必须在参数之前

`flag` 包要求所有标志都必须出现在非标志参数之前。如果标志出现在后面，它将被视为普通的非标志参数。

```
$ ./command-line-flags -word=opt a1 a2 a3 -numb=7
word: opt
numb: 42
fork: false
svar: bar
tail: [a1 a2 a3 -numb=7]
```

5. 查看帮助信息

`flag` 包会自动生成帮助信息，可以通过 `-h` 或 `--help` 查看。

```
$ ./command-line-flags -h
Usage of ./command-line-flags:
  -fork
        a bool
  -numb int
        an int (default 42)
  -svar string
        a string var (default "bar")
  -word string
        a string (default "foo")
```

74.5 关键要点

1. **`flag` 包**：是 Go 处理命令行标志的标准库，比手动解析 `os.Args` 更强大、更方便。
2. **声明、解析、访问**：使用 `flag` 包遵循“声明所有标志 -> 调用 `flag.Parse()` -> 访问标志值”的基本流程。
3. **指针与变量绑定**：你可以选择让 `flag` 函数返回一个指向值的指针，或者将标志直接绑定到一个已有的变量上。
4. **自动生成帮助信息**：`flag` 包可以根据你提供的帮助字符串自动生成 `-h` 帮助文本，非常实用。

5. **标志与参数的顺序**：必须先提供所有标志，然后才能提供非标志的尾随参数。

75. Command-Line Subcommands (命令行子命令)

https://gobyexample.com/command-line-subcommands

75.1 核心概念

许多命令行工具，如 `go` 或 `git`，都支持子命令（subcommands）。例如，`go build` 和 `go get` 就是 `go` 工具的两个不同子命令。子命令让一个程序可以支持多种不同的功能，每种功能可以有自己专属的一套命令行标志（flags）。Go 的 `flag` 包通过创建不同的标志集（`FlagSet`）来支持这种模式。

生活化类比：子命令就像是一个多功能工具箱（主命令 `go`），里面有不同的工具（子命令 `build`, `run`）。每个工具都有自己独特的功能和可以调节的旋钮（每个子命令专属的标志）。当你需要编译程序时，你拿出 `build` 这个工具；当你需要运行它时，你换用 `run` 这个工具。

75.2 实现子命令

75.2.1 为每个子命令创建标志集

实现子命令的关键是为每个子命令创建一个独立的 `flag.FlagSet`。这允许每个子命令拥有自己的一套标志，而不会相互干扰。

```
// 为 'foo' 子命令创建一个新的标志集。
// 第一个参数是子命令的名称，第二个参数定义了当解析出错时的行为。
// flag.ExitOnError 表示如果发生错误，程序将打印帮助信息并退出。
fooCmd := flag.NewFlagSet("foo", flag.ExitOnError)

// 在 'foo' 标志集上定义该子命令专属的标志。
fooEnable := fooCmd.Bool("enable", false, "enable")
fooName := fooCmd.String("name", "", "name")

// 同样，为 'bar' 子命令创建另一个独立的标志集和专属标志。
barCmd := flag.NewFlagSet("bar", flag.ExitOnError)
barLevel := barCmd.Int("level", 0, "level")
```

75.2.2 判断要执行哪个子命令

程序需要检查命令行参数来确定用户想要执行哪个子命令。子命令通常是程序名后的第一个参数，即 `os.Args[1]`。

```
// 检查命令行参数的数量，如果少于2个（程序名 + 子命令），则说明用户没有提供子命令。
if len(os.Args) < 2 {
    fmt.Println("expected 'foo' or 'bar' subcommands")
    os.Exit(1)
}

// 使用 switch 语句来判断第一个参数是什么，从而决定执行哪个子命令的逻辑。
switch os.Args[1] {
// ... case "foo": ... case "bar": ...
}
```

75.2.3 解析特定子命令的标志

在确定了子命令后，需要调用该子命令对应标志集的 `Parse` 方法。**重要的是，传递给 `Parse` 的参数应该是子命令之后的所有参数**，即 `os.Args[2:]`。

```
case "foo":
    // 调用 'foo' 标志集的 Parse 方法，解析剩余的参数。
    fooCmd.Parse(os.Args[2:])
    // ... 然后访问 'foo' 子命令的标志和参数。
```

75.3 完整代码示例

```
package main

import (
    "flag"
    "fmt"
    "os"
)

func main() {

    // --- 步骤 1: 为 'foo' 子命令定义一个标志集 ---
    // NewFlagSet 创建一个新的标志集，用于隔离不同子命令的标志。
    // "foo" 是子命令的名称。
    // flag.ExitOnError 指定了当解析标志出错时，程序应打印错误并退出。
    fooCmd := flag.NewFlagSet("foo", flag.ExitOnError)
    // 为 'foo' 子命令定义一个布尔型标志 "-enable"。
    fooEnable := fooCmd.Bool("enable", false, "enable a feature")
    // 为 'foo' 子命令定义一个字符串型标志 "-name"。
    fooName := fooCmd.String("name", "", "a name")

    // --- 步骤 2: 为 'bar' 子命令定义另一个标志集 ---
    // 这创建了另一套完全独立的标志。
    barCmd := flag.NewFlagSet("bar", flag.ExitOnError)
    // 为 'bar' 子命令定义一个整型标志 "-level"。
    barLevel := barCmd.Int("level", 0, "a level")

    // --- 步骤 3: 检查用户是否提供了子命令 ---
```

```
    // os.Args[0] 是程序本身的名称，所以至少需要2个参数才算提供了子命令。
    if len(os.Args) < 2 {
        fmt.Println("expected 'foo' or 'bar' subcommands")
        os.Exit(1) // 如果没有子命令，则退出程序。
    }

    // --- 步骤 4: 根据第一个参数决定执行哪个子命令的逻辑 ---
    // os.Args[1] 应该是子命令的名称。
    switch os.Args[1] {

    // 如果子命令是 "foo"
    case "foo":
        // --- 步骤 5a: 解析 'foo' 子命令的标志 ---
        // 重要: 只把子命令后面的参数 (os.Args[2:]) 传给 Parse 方法。
        fooCmd.Parse(os.Args[2:])

        // 打印确认信息和解析出的标志值。
        fmt.Println("subcommand 'foo'")
        fmt.Println("  enable:", *fooEnable) // 标志返回的是指针，需要用 * 解引用来获取值。
        fmt.Println("  name:", *fooName)
        fmt.Println("  tail:", fooCmd.Args()) // Args() 方法获取不属于标志的尾随参数。

    // 如果子命令是 "bar"
    case "bar":
        // --- 步骤 5b: 解析 'bar' 子命令的标志 ---
        barCmd.Parse(os.Args[2:])

        fmt.Println("subcommand 'bar'")
        fmt.Println("  level:", *barLevel)
        fmt.Println("  tail:", barCmd.Args())

    // 如果不是任何已知的子命令
    default:
        fmt.Println("expected 'foo' or 'bar' subcommands")
        os.Exit(1)
    }
}
```

75.4 运行示例

1. 执行 `foo` 子命令

```
# 构建程序
$ go build command-line-subcommands.go

# 运行 foo 子命令，并提供其专属的标志和参数
$ ./command-line-subcommands foo -enable -name=joe a1 a2
subcommand 'foo'
  enable: true
  name: joe
  tail: [a1 a2]
```

2. 执行 `bar` 子命令

```
# 运行 bar 子命令
$ ./command-line-subcommands bar -level=8 a1
subcommand 'bar'
  level: 8
  tail: [a1]
```

3. 子命令会忽略不属于它的标志
`bar` 子命令没有定义 `-enable` 标志，所以它被当作一个普通的尾随参数。

```
$ ./command-line-subcommands bar -enable a1
subcommand 'bar'
  level: 0
  tail: [-enable a1]
```

75.5 关键要点

1. **`flag.NewFlagSet` 是核心**：通过为每个子命令创建独立的 `FlagSet`，可以隔离它们的标志，避免冲突。
2. **两步解析**：首先，程序需要手动检查 `os.Args[1]` 来确定子命令；然后，调用对应 `FlagSet` 的 `Parse` 方法来解析 `os.Args[2:]`。
3. **独立的标志空间**：每个 `FlagSet` 有自己独立的标志定义。一个子命令的标志对于另一个子命令是未知的，会被当作普通参数处理。
4. **错误处理**：`flag.ExitOnError` 是一种方便的错误处理策略，它可以在解析失败时自动打印用法信息并退出程序。
5. **尾随参数**：每个 `FlagSet` 在 `Parse` 之后，都可以通过其 `Args()` 方法获取不属于其已定义标志的那些尾随参数。

76. Environment Variables (环境变量)

https://gobyexample.com/environment-variables

76.1 核心概念

环境变量（Environment Variables）是在操作系统级别设置的键值对，它们可以影响正在运行的进程的行为。例如，`PATH` 环境变量决定了系统在哪些目录中查找可执行文件。在 Go 中，可以使用 `os` 包来读取和设置环境变量。

生活化类比：环境变量就像是贴在房间墙上的一些“通用规则”或“背景信息”，比如“室温：26度”或“模式：安静”。房间里的任何人（任何程序）都可以抬头看到这些信息，并根据这些信息来调整自己的行为。例如，程序看到 `MODE=production` 这条规则，就知道现在是正式环境，需要表现得更稳重（比如关闭调试信息）。

76.2 环境变量操作

76.2.1 设置环境变量

`os.Setenv` 用于在当前 Go 进程中设置一个环境变量。这个设置只对当前进程及其子进程有效，不会影响到父进程（如你的 shell）。

```
os.Setenv("FOO", "1")
```

76.2.2 读取环境变量

`os.Getenv` 用于读取一个环境变量的值。如果该变量未被设置，它会返回一个空字符串。

```
fmt.Println("FOO:", os.Getenv("FOO")) // 输出: 1
fmt.Println("BAR:", os.Getenv("BAR")) // 输出: (空字符串)
```

76.2.3 列出所有环境变量

`os.Environ` 返回一个包含了所有环境变量的字符串切片，每个字符串的格式都是 `KEY=VALUE`。

```
fmt.Println()
for _, e := range os.Environ() {
    // 使用 strings.SplitN 可以安全地将字符串按第一个 "=" 分割成两部分
    pair := strings.SplitN(e, "=", 2)
    fmt.Println(pair[0]) // 打印键名
}
```

76.3 完整代码示例

```
package main

import (
    "fmt"
    "os"
    "strings"
)

func main() {

    // 使用 os.Setenv 来设置一个键值对
    os.Setenv("FOO", "1")
    // 使用 os.Getenv 来获取一个键的值
    fmt.Println("FOO:", os.Getenv("FOO"))
    // 如果键不存在，Getenv 会返回一个空字符串
    fmt.Println("BAR:", os.Getenv("BAR"))

    fmt.Println()
    // os.Environ 返回一个包含所有环境变量的字符串切片
    // 格式为 "KEY=VALUE"
    for _, e := range os.Environ() {
        // 使用 strings.SplitN 将字符串按第一个 "=" 分割成两部分
        pair := strings.SplitN(e, "=", 2)
        // 打印出环境变量的键
```

```
        fmt.Println(pair[0])
    }
}
```

76.4 运行示例

你可以在运行 Go 程序时，在命令前加上环境变量的设置，这些变量只对本次运行有效。

```
# 直接运行程序
$ go run environment-variables.go
FOO: 1
BAR:

...
USER
HOME
...

# 在运行时设置一个环境变量
$ BAR=2 go run environment-variables.go
FOO: 1
BAR: 2

...
USER
HOME
BAR
...
```

(注：`os.Environ` 的输出会很长，且因系统环境而异，这里只显示部分示意)

76.5 关键要点

1. **`os` 包**：是 Go 中与环境变量交互的标准库。
2. **`os.Getenv` 和 `os.Setenv`**：是读取和设置单个环境变量最直接的方法。
3. **空字符串表示不存在**：`os.Getenv` 在找不到指定变量时返回空字符串，这是一种需要注意的惯例。
4. **`os.Environ`**：用于获取当前进程可见的所有环境变量的快照。
5. **进程作用域**：通过 `os.Setenv` 设置的环境变量只对当前 Go 进程和它启动的子进程有效。

77. Logging (日志记录)

https://gobyexample.com/logging

77.1 核心概念

日志记录是应用程序开发中不可或缺的一部分，用于记录程序的运行状态、错误信息和调试线索。Go 语言提供了两个主要的日志包：

- **`log`**: 提供简单、传统的自由格式文本日志。

- **`log/slog`**: (自 Go 1.21 起) 提供功能更强大的结构化日志，支持 JSON 等格式和日志级别。

生活化类比：日志就像是飞机的“黑匣子”，记录下飞行过程中的关键信息。传统的 `log` 包就像是飞行员的语音记录，记录的是自然语言，方便人听。而结构化的 `slog` 包则更像是飞行数据记录器，它把信息按“高度”、“速度”、“时间”等标准字段存成键值对，这样机器就能轻松地对这些数据进行筛选、统计和分析。

77.2 使用 `log` 包 (传统日志)

`log` 包提供了一个默认的“标准记录器”，可以直接使用。

77.2.1 基本日志输出

`log.Println` 会将日志信息打印到标准错误（`os.Stderr`），并自动添加换行符。

```
log.Println("standard logger")
```

77.2.2 配置日志标志

`log.SetFlags` 可以用来配置日志输出的格式，例如添加时间戳、文件名和行号。

```
// 添加微秒级时间戳
log.SetFlags(log.LstdFlags | log.Lmicroseconds)
log.Println("with micro")

// 添加文件名和行号
log.SetFlags(log.LstdFlags | log.Lshortfile)
log.Println("with file/line")
```

77.2.3 创建自定义记录器

`log.New` 可以创建一个新的记录器实例，允许你指定输出目标（任何实现了 `io.Writer` 的对象）、日志前缀和标志。

```
// 创建一个输出到标准输出 (os.Stdout) 的记录器，并带上 "my:" 前缀
mylog := log.New(os.Stdout, "my:", log.LstdFlags)
mylog.Println("from mylog")

// 也可以在创建后修改前缀
mylog.SetPrefix("ohmy:")
mylog.Println("from mylog")
```

77.2.4 将日志写入缓冲区

你可以将日志输出重定向到一个内存中的缓冲区（如 `bytes.Buffer`），这在测试或需要捕获日志内容时非常有用。

```
var buf bytes.Buffer
buflog := log.New(&buf, "buf:", log.LstdFlags)
buflog.Println("hello")
fmt.Print("from buflog:", buf.String())
```

77.3 使用 `log/slog` 包 (结构化日志)

结构化日志以键值对的形式记录信息，更便于机器解析和查询。

77.3.1 创建 JSON 处理器和记录器

`slog.NewJSONHandler` 创建一个将日志格式化为 JSON 的处理器。然后用 `slog.New` 基于该处理器创建一个新的结构化记录器。

```
jsonHandler := slog.NewJSONHandler(os.Stderr, nil)
myslog := slog.New(jsonHandler)
```

77.3.2 记录结构化信息

`slog` 提供了 `Info`, `Debug`, `Warn`, `Error` 等方法来记录不同级别的日志。除了日志消息，你还可以附加任意数量的键值对。

```
myslog.Info("hi there")

myslog.Info("hello again", "key", "val", "age", 25)
```

77.4 完整代码示例

```
package main

import (
    "bytes"
    "fmt"
    "log"
    "log/slog"
    "os"
)

func main() {
    // --- 使用传统的 `log` 包 ---
    log.Println("standard logger")

    log.SetFlags(log.LstdFlags | log.Lmicroseconds)
    log.Println("with micro")

    log.SetFlags(log.LstdFlags | log.Lshortfile)
    log.Println("with file/line")

    mylog := log.New(os.Stdout, "my:", log.LstdFlags)
```

```
    mylog.Println("from mylog")

    mylog.SetPrefix("ohmy:")
    mylog.Println("from mylog")

    var buf bytes.Buffer
    buflog := log.New(&buf, "buf:", log.LstdFlags)
    buflog.Println("hello")
    fmt.Print("from buflog:", buf.String())

    // --- 使用 Go 1.21+ 的 `log/slog` 包 ---
    jsonHandler := slog.NewJSONHandler(os.Stderr, nil)
    myslog := slog.New(jsonHandler)
    myslog.Info("hi there")

    myslog.Info("hello again", "key", "val", "age", 25)
}
```

77.5 运行结果

```
$ go run logging.go
2023/10/27 10:00:00 standard logger
2023/10/27 10:00:00.123456 with micro
2023/10/27 10:00:00 logging.go:22: with file/line
my:2023/10/27 10:00:00 from mylog
ohmy:2023/10/27 10:00:00 from mylog
from buflog:buf:2023/10/27 10:00:00 hello
{"time":"2023-10-27T10:00:00.123456-04:00","level":"INFO","msg":"hi there"}
{"time":"2023-10-27T10:00:00.123456-04:00","level":"INFO","msg":"hello
again","key":"val","age":25}
```

(注：输出中的时间和行号会根据实际运行情况而变化)

77.6 关键要点

1. **两种日志风格**：Go 同时支持传统的自由文本日志 (`log`) 和现代的结构化日志 (`log/slog`)。
2. **`log` 包**：简单易用，适合快速开发或简单的应用。可以通过 `SetFlags` 定制输出格式，通过 `New` 创建自定义记录器。
3. **`log/slog` 包**：功能更强大，输出为键值对格式（如 JSON），非常适合在生产环境中进行日志的收集、查询和分析。
4. **日志级别**：`slog` 引入了日志级别的概念（`Info`, `Warn`, `Error` 等），可以根据严重性过滤和处理日志。
5. **可扩展性**：`log` 和 `slog` 都允许将日志输出到任何实现了 `io.Writer` 的地方，如文件、网络连接或内存缓冲区。

78. HTTP Client (HTTP 客户端)

https://gobyexample.com/http-client

78.1 核心概念

HTTP (Hypertext Transfer Protocol) 是现代网络通信的基石。Go 的 `net/http` 包为构建 HTTP 客户端和服务器提供了强大而易用的支持。本节将重点介绍如何作为客户端向其他服务器发送 HTTP 请求。

生活化类比：HTTP 客户端就像是你的程序派出去的一个“网络浏览器”。你给它一个网址（URL），它就会像浏览器一样去访问那个网站，然后把网站的页面内容（响应体）带回来给你。最重要的一点是，`defer resp.Body.Close()` 就像是你看完报纸后记得把它扔进回收箱，这是一个必须养成的“随手关门”的好习惯，防止资源浪费。

78.2 发送 HTTP 请求

78.2.1 简单的 GET 请求

`http.Get` 是发送一个 HTTP GET 请求最简单的方式。它接收一个 URL 字符串作为参数，并返回一个 `*http.Response` 和一个 `error`。

```
resp, err := http.Get("https://gobyexample.com")
```

78.2.2 错误处理和资源释放

与文件操作类似，HTTP 请求也可能失败，因此必须检查返回的 `error`。此外，`http.Response` 的 `Body` 是一个 `io.ReadCloser`，在使用完毕后必须被关闭以释放网络连接资源。使用 `defer` 是确保其被关闭的最佳实践。

```
if err != nil {
    panic(err)
}
// 在函数退出时，确保关闭响应体，防止资源泄漏。
defer resp.Body.Close()
```

78.2.3 读取响应状态和正文

- `resp.Status` 字段包含了响应的状态码和文本描述（例如 `"200 OK"`）。
- `resp.Body` 字段是一个 `io.Reader`，可以从中读取响应的正文内容。我们可以使用 `bufio.Scanner` 来方便地逐行读取。

```
fmt.Println("Response status:", resp.Status)

scanner := bufio.NewScanner(resp.Body)
// 循环读取响应体的前5行
for i := 0; scanner.Scan() && i < 5; i++ {
    fmt.Println(scanner.Text())
}
```

78.3 完整代码示例

```
package main
```

```go
import (
    "bufio"
    "fmt"
    "net/http"
)

func main() {

    // --- 步骤 1: 发送一个 HTTP GET 请求 ---
    // http.Get 是一个便捷的函数，用于发起 GET 请求。
    // 它返回一个响应对象 (*http.Response) 和一个错误。
    resp, err := http.Get("https://gobyexample.com")
    // 检查在请求过程中是否发生了错误（例如，DNS查询失败、网络连接问题）。
    if err != nil {
        panic(err)
    }
    // --- 步骤 2: 确保响应体被关闭 ---
    // resp.Body 是一个需要关闭的资源，以释放底层的网络连接。
    // 使用 defer 可以在函数结束时自动调用 resp.Body.Close()，这是 Go 中的一个重要实践。
    defer resp.Body.Close()

    // --- 步骤 3: 打印响应状态 ---
    // resp.Status 字段包含了 HTTP 状态码和状态文本，例如 "200 OK"。
    fmt.Println("Response status:", resp.Status)

    // --- 步骤 4: 读取响应正文 ---
    // resp.Body 是一个 io.Reader，我们可以用多种方式读取它。
    // 这里我们使用 bufio.Scanner 来方便地逐行读取。
    scanner := bufio.NewScanner(resp.Body)

    // 循环读取并打印响应体的前5行。
    // scanner.Scan() 在有下一行时返回 true。
    for i := 0; scanner.Scan() && i < 5; i++ {
        fmt.Println(scanner.Text())
    }

    // --- 步骤 5: 检查扫描过程中是否发生错误 ---
    // 在循环结束后，检查 scanner.Err() 是一个好习惯，以确保读取过程没有问题。
    if err := scanner.Err(); err != nil {
        panic(err)
    }
}
```

78.4 运行结果

```
$ go run http-client.go
Response status: 200 OK
<!DOCTYPE html>
<html>
  <head>
    <title>Go by Example</title>
    <link rel=stylesheet href="site.css">
```

(注：输出的 HTML 内容可能会因网站更新而略有不同)

78.5 关键要点

1. **`net/http` 包**：是 Go 中进行 HTTP 编程的标准库，功能全面。
2. **`http.Get`**：是发起简单 GET 请求的快捷方式。
3. **`defer resp.Body.Close()`**：这是一个至关重要的模式。忘记关闭响应体是常见的资源泄漏原因，必须牢记。
4. **`resp.Body` 是 `io.Reader`**：响应体是一个数据流，你可以像读取文件一样读取它，可以使用 `bufio.Scanner`、`io.ReadAll` 等多种工具。
5. **错误处理**：网络请求的每一步都可能出错（请求本身、读取响应体），因此需要仔细处理错误。

79. HTTP Server (HTTP 服务器)

https://gobyexample.com/http-server

79.1 核心概念

与 HTTP 客户端相对应，Go 的 `net/http` 包同样为构建 HTTP 服务器提供了简单而强大的功能。一个基本的 HTTP 服务器需要做两件事：注册请求处理器（handler）来响应特定的 URL 路径，以及监听一个端口来接收客户端的请求。

生活化类比：HTTP 服务器就像一家餐厅。`http.HandleFunc` 是你在制定菜单，比如规定“/hello”这道菜由 `hello` 这位厨师负责。`http.ListenAndServe(":8090", nil)` 则是餐厅在8090号门开门营业。当有客人点“/hello”这道菜时（客户端访问 `/hello` 路径），餐厅的领班（`DefaultServeMux`）就会把单子交给 `hello` 厨师去处理。

79.2 构建 HTTP 服务器

79.2.1 请求处理器 (Handler)

处理器是一个函数，它负责接收 HTTP 请求并构造 HTTP 响应。在 Go 中，一个处理器函数需要满足特定的签名：`func(w http.ResponseWriter, req *http.Request)`。

- `http.ResponseWriter w`: 这是一个接口，用于向客户端写入响应数据和头信息。
- `*http.Request req`: 这是一个结构体，包含了客户端发送的所有请求信息，如 URL、头信息和请求正文。

```go
// "hello" 处理器：向客户端写入一个简单的 "hello" 字符串。
func hello(w http.ResponseWriter, req *http.Request) {
    fmt.Fprintf(w, "hello\n")
}

// "headers" 处理器：遍历并打印出客户端请求的所有头信息。
func headers(w http.ResponseWriter, req *http.Request) {
    for name, headers := range req.Header {
        for _, h := range headers {
            fmt.Fprintf(w, "%v: %v\n", name, h)
        }
    }
}
```

79.2.2 注册处理器

`http.HandleFunc` 函数用于将一个处理器函数注册到一个特定的 URL 路径上。当服务器收到该路径的请求时，就会调用对应的处理器。

```go
http.HandleFunc("/hello", hello)
http.HandleFunc("/headers", headers)
```

79.2.3 启动服务器

`http.ListenAndServe` 用于启动服务器并开始在指定的端口上监听请求。这个函数会一直阻塞，直到服务器被关闭或发生致命错误。

- 第一个参数是监听的地址和端口（例如 `":8090"`）。
- 第二个参数通常是 `nil`，表示使用默认的请求多路复用器（`DefaultServeMux`），也就是我们刚刚通过 `http.HandleFunc` 注册处理器的那个复用器。

```go
http.ListenAndServe(":8090", nil)
```

79.3 完整代码示例

```go
package main

import (
    "fmt"
    "net/http"
)

// 处理器是 Go HTTP 服务器的核心。
// 它们是接收 http.ResponseWriter 和 http.Request 作为参数的函数。

// "hello" 处理器，一个简单的响应函数。
func hello(w http.ResponseWriter, req *http.Request) {

    // http.ResponseWriter 用于填充 HTTP 响应。
```

```
    // 这里我们简单地向其写入 "hello\n"。
    fmt.Fprintf(w, "hello\n")
}

// "headers" 处理器，它会读取并打印出 HTTP 请求中的所有头信息。
func headers(w http.ResponseWriter, req *http.Request) {

    // req.Header 是一个 map[string][]string，存储了请求的所有头信息。
    for name, headers := range req.Header {
        // 遍历每个头信息的值（一个头可能对应多个值）。
        for _, h := range headers {
            // 将头信息的键值对写入响应体。
            fmt.Fprintf(w, "%v: %v\n", name, h)
        }
    }
}

func main() {

    // --- 步骤 1: 注册处理器 ---
    // 使用 http.HandleFunc 函数将我们的处理器函数注册到服务器的路由上。
    // 这会将对 "/hello" 路径的请求交给 hello 函数处理。
    http.HandleFunc("/hello", hello)
    // 将对 "/headers" 路径的请求交给 headers 函数处理。
    http.HandleFunc("/headers", headers)

    // --- 步骤 2: 启动服务器 ---
    // 调用 ListenAndServe 来启动一个 HTTP 服务器。
    // 第一个参数是监听的地址和端口，":8090" 表示在所有网络接口的 8090 端口上监听。
    // 第二个参数是处理器，传 nil 表示使用我们刚刚注册的默认多路复用器。
    // 这个函数会阻塞主 goroutine，直到服务器被关闭。
    http.ListenAndServe(":8090", nil)
}
```

79.4 运行示例

1. 在一个终端中启动服务器

```
# 使用 go run 启动服务器。它会一直运行，等待请求。
$ go run http-server.go &
```

2. 在另一个终端中访问 `/hello` 路由

```
$ curl localhost:8090/hello
hello
```

3. 访问 `/headers` 路由

```
$ curl localhost:8090/headers
User-Agent: curl/7.77.0
Accept: */*
```

(注：`curl` 发送的头信息可能因版本和环境而异)

79.5 关键要点

1. **处理器函数 (Handler Func)**：是处理 HTTP 请求的业务逻辑核心，签名固定为 `func(http.ResponseWriter, *http.Request)`。
2. **`http.HandleFunc`**：用于将 URL 路径与处理器函数绑定起来，构建路由规则。
3. **`http.ListenAndServe`**：是启动并运行 HTTP 服务器的阻塞函数。
4. **`ResponseWriter` 和 `Request`**：`ResponseWriter` 用于构建返回给客户端的响应，而 `Request` 则包含了客户端发来的所有请求信息。
5. **默认多路复用器**：当 `ListenAndServe` 的第二个参数为 `nil` 时，Go 会使用一个全局的 `DefaultServeMux`，`http.HandleFunc` 就是在向它注册路由。

80. Context (上下文)

https://gobyexample.com/context

80.1 核心概念

在 Go 中，`context` 包提供了一种强大的机制，用于在 API 边界和多个 Goroutine 之间传递请求范围的值、取消信号和超时信息。`Context` 在处理网络请求或需要控制长时间运行任务的场景中至关重要，它能优雅地处理任务的取消和超时，避免资源浪费。

生活化类比：`Context` 就像是下发给一个任务小组的“行动指令书”。这份指令书不仅说明了任务目标，更重要的是包含了“中止条件”，比如“任务必须在午夜前完成”（超时），或者“一旦听到‘红色警报’的暗号就立刻撤退”（取消信号）。小组里的每个成员（每个 goroutine）都持有这份指令书，并且在执行任务的每个关键节点都会检查一下是否需要中止行动。

80.2 在 HTTP 服务器中使用 Context

一个常见的应用场景是在 HTTP 服务器中。服务器会为每个接收到的请求创建一个 `Context`。如果在请求处理完成前，客户端断开了连接，服务器可以通过这个 `Context` 得到通知，从而停止不必要的工作（如数据库查询）。

80.2.1 获取请求的 Context

在处理器函数中，可以通过 `req.Context()` 来获取当前请求关联的 `Context`。

```
ctx := req.Context()
```

80.2.2 监听 Context 的取消事件

`Context` 提供了一个 `Done()` 方法，它返回一个 channel。当这个 `Context` 被取消或超时时，这个 `Done()` channel 会被关闭。我们可以使用 `select` 语句来同时等待耗时操作和 `Context` 的取消事件。

```
select {
// 模拟一个耗时的操作，例如等待10秒
case <-time.After(10 * time.Second):
    // 操作完成，正常返回响应
    fmt.Fprintf(w, "hello\n")

// 如果 context 的 Done channel 被关闭了
case <-ctx.Done():
    // Context 被取消了（例如，客户端断开连接）
    err := ctx.Err() // 获取取消的原因
    fmt.Println("server:", err)
    // 停止工作，并向客户端返回一个错误
    http.Error(w, err.Error(), http.StatusInternalServerError)
}
```

80.3 完整代码示例

```
package main

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

// hello 是一个模拟耗时操作的 HTTP 处理器。
func hello(w http.ResponseWriter, req *http.Request) {

    // --- 步骤 1：获取请求的 Context ---
    // 对于每个 HTTP 请求，Go 服务器都会创建一个 Context。
    // 这个 Context 会在客户端断开连接时被取消。
    ctx := req.Context()
    fmt.Println("server: hello handler started")
    defer fmt.Println("server: hello handler ended")

    // --- 步骤 2：使用 select 等待耗时操作或 Context 取消 ---
    select {
    // 场景 A：耗时操作正常完成
    // time.After 返回一个 channel，它会在指定的时间后发送一个值。
    case <-time.After(10 * time.Second):
        // 10秒钟过去了，客户端没有断开连接。
        // 正常地向客户端写入响应。
        fmt.Fprintf(w, "hello\n")

    // 场景 B：Context 被取消
    // ctx.Done() 返回一个 channel。当 Context 被取消时，这个 channel 会被关闭。
    case <-ctx.Done():
        // 在10秒的等待结束前，Context 被取消了（例如，客户端按下了 Ctrl+C）。

        // ctx.Err() 返回导致 Context 被取消的原因。
        err := ctx.Err()
```

```
        fmt.Println("server:", err)

        // 向客户端报告错误。注意，此时客户端可能已经断开，
        // 所以这个错误可能不会被客户端接收到，但这是良好的服务器行为。
        internalError := http.StatusInternalServerError
        http.Error(w, err.Error(), internalError)
    }
}

func main() {
    // 将 /hello 路径的请求交给 hello 处理器。
    http.HandleFunc("/hello", hello)
    // 启动服务器，监听 8090 端口。
    http.ListenAndServe(":8090", nil)
}
```

80.4 运行示例

1. 在一个终端中启动服务器

```
$ go run context.go &
```

2. 在另一个终端中，使用 `curl` 访问服务器

```
$ curl localhost:8090/hello
```

3. 立即在 `curl` 终端中按下 `Ctrl+C`
`curl` 客户端会提前退出。

4. 观察服务器终端的输出
你会看到服务器端的日志显示 `Context` 被取消了，处理器也随之提前结束，而没有等到10秒结束。

```
server: hello handler started
server: context canceled
server: hello handler ended
```

如果你不按 `Ctrl+C`，那么大约10秒后，`curl` 会收到 `hello` 响应，服务器端则不会打印 `context canceled`。

80.5 关键要点

1. **`Context` 的核心作用**：在不同的 Goroutine 和函数调用链中传递“请求范围”的数据，特别是**取消信号**和**截止时间**。
2. **`req.Context()`**：在 HTTP 服务器中，每个请求的 `*http.Request` 对象都包含一个 `Context`，它与该请求的生命周期绑定。
3. **`ctx.Done()`**：这是 `Context` 的精髓。它返回一个 channel，通过 `select` 监听这个 channel，可以使你的函数优雅地响应取消事件。

4. **优雅地停止工作**：当 `ctx.Done()` 被触发时，意味着调用方（在这里是 HTTP 服务器）不再需要你的工作结果了。你的函数应该立即停止耗时操作（如数据库查询、外部 API 调用），并清理资源后返回。
5. **不仅仅用于 HTTP**：`Context` 是一种通用模式，广泛应用于任何需要控制并发操作生命周期的场景。

81. Spawning Processes (生成进程)

https://gobyexample.com/spawning-processes

81.1 核心概念

在 Go 中，`os/exec` 包提供了执行外部命令（即生成子进程）的功能。这使得 Go 程序可以与其他命令行工具进行交互，是编写系统级脚本和工具的必备技能。

生活化类比：生成进程就像是你的主程序（老板）请了一个外部专家（子进程，如 `grep` 或 `ls`）来完成一项特定任务。老板把任务要求和材料（参数和标准输入）交给专家，专家独立完成工作，然后把成果（标准输出）交回给老板。老板在专家工作时可以做别的事情，也可以选择等待专家完成。

81.2 执行外部命令

81.2.1 简单的命令执行

`exec.Command` 用于创建一个代表外部命令的 `Cmd` 对象。它接收命令名和任意数量的参数。

```
dateCmd := exec.Command("date")
```

`cmd.Output()` 方法会执行命令，并等待其完成，然后返回命令的标准输出。如果命令执行出错，它会返回一个错误。

```
dateOut, err := dateCmd.Output()
if err != nil {
    panic(err)
}
fmt.Println(string(dateOut))
```

81.2.2 通过管道与子进程交互

对于需要输入（stdin）或需要捕获输出（stdout）的更复杂的交互，可以创建管道（Pipe）。

1. **创建命令和管道**：

   ```
   grepCmd := exec.Command("grep", "hello")
   grepIn, _ := grepCmd.StdinPipe()    // 获取子进程的标准输入管道
   grepOut, _ := grepCmd.StdoutPipe()  // 获取子进程的标准输出管道
   ```

2. **启动命令**：
   `cmd.Start()` 异步地启动命令，不会等待它完成。

   ```
   grepCmd.Start()
   ```

3. **写入和读取**：
   通过输入管道向子进程写入数据，并通过输出管道读取其结果。写入完成后，必须关闭输入管道，这样子进程才能知道输入已经结束。

```
grepIn.Write([]byte("hello grep\ngoodbye grep"))
grepIn.Close()
grepBytes, _ := io.ReadAll(grepOut)
```

4. **等待命令完成**：
   `cmd.Wait()` 会等待命令执行完成，并释放相关资源。

```
grepCmd.Wait()
```

81.2.3 执行复杂的 Shell 命令

如果要执行一个包含 shell 特性（如管道 `|`、重定向 `>`）的完整命令字符串，一个常见的模式是调用 `bash` (或 `sh`) 并使用 `-c` 标志。

```
lsCmd := exec.Command("bash", "-c", "ls -a -l -h")
lsOut, err := lsCmd.Output()
```

81.3 完整代码示例

```
package main

import (
    "fmt"
    "io"
    "os/exec"
)

func main() {

    // --- 示例 1: 执行简单命令并捕获输出 ---
    // exec.Command 创建一个对象来表示外部进程。
    // 第一个参数是命令路径，后面是任意数量的参数。
    dateCmd := exec.Command("date")

    // .Output() 是另一个辅助函数，它会执行命令并收集其标准输出。
    // 如果发生错误，它会返回一个 *exec.Error。
    dateOut, err := dateCmd.Output()
    if err != nil {
        panic(err)
    }
    fmt.Println("> date")
    fmt.Println(string(dateOut))

    // --- 示例 2: 通过管道与子进程交互 ---
    // 创建一个 grep 命令。
```

```
    grepCmd := exec.Command("grep", "hello")

    // StdinPipe() 返回一个连接到子进程标准输入的管道 (io.WriteCloser)。
    grepIn, _ := grepCmd.StdinPipe()
    // StdoutPipe() 返回一个连接到子进程标准输出的管道 (io.ReadCloser)。
    grepOut, _ := grepCmd.StdoutPipe()

    // 使用 Start() 异步启动命令。
    grepCmd.Start()

    // 向子进程的 stdin 写入数据。
    grepIn.Write([]byte("hello grep\ngoodbye grep"))
    // 关闭输入管道。如果不关闭，grep 进程会一直等待更多输入而不会结束。
    grepIn.Close()

    // 使用 io.ReadAll 从子进程的 stdout 读取所有输出。
    grepBytes, _ := io.ReadAll(grepOut)

    // 使用 Wait() 等待子进程退出。
    grepCmd.Wait()

    fmt.Println("> grep hello")
    fmt.Println(string(grepBytes))

    // --- 示例 3: 执行包含 shell 功能的复杂命令 ---
    // 当需要执行一个完整的命令字符串时（包含管道、重定向等），
    // 一种常见的方法是调用 shell 的 -c 选项。
    lsCmd := exec.Command("bash", "-c", "ls -a -l -h")
    lsOut, err := lsCmd.Output()
    if err != nil {
        panic(err)
    }
    fmt.Println("> ls -a -l -h")
    fmt.Println(string(lsOut))
}
```

81.4 运行结果

```
$ go run spawning-processes.go
> date
Sat Oct 27 10:15:57 PDT 2023

> grep hello
hello grep

> ls -a -l -h
total 16
drwxr-xr-x  4 user group  128B Oct 27 10:15 .
drwxr-xr-x 39 user group  1.2K Oct 27 10:15 ..
-rw-r--r--  1 user group  533B Oct 27 10:15 spawning-processes.go
```

(注：`date` 和 `ls` 的输出会因系统和运行时间而异)

81.5 关键要点

1. **`os/exec` 包**：是 Go 与外部命令交互的标准库。
2. **`exec.Command`**：是创建命令的入口，它只定义命令和参数，并不立即执行。
3. **`Output` vs `Run` vs `Start`**：
   - `Output()`：执行命令，收集标准输出，并返回。适合简单的命令。
   - `Run()`：执行命令并等待其完成，但不收集输出。适合不需要输出的命令。
   - `Start()`：异步启动命令，不等待。需要与 `Wait()` 配对使用，通常用于复杂的 I/O 管道交互。
4. **管道（Pipes）**：通过 `StdinPipe` 和 `StdoutPipe` 可以建立与子进程标准输入/输出的数据流通道，实现复杂的交互。
5. **`bash -c`**：是执行需要 shell 解析的复杂命令字符串的实用技巧。

82. Exec'ing Processes (执行进程)

https://gobyexample.com/execing-processes

82.1 核心概念

在上一章中，我们学习了如何“生成”（spawn）一个子进程，这意味着 Go 程序会创建一个新的子进程，并继续运行自己的代码。本章将介绍一个完全不同的概念：“执行”（exec'ing）一个进程。**Exec'ing 会将当前的 Go 进程完全替换为一个全新的进程**。这意味着一旦 `exec` 调用成功，原来的 Go 程序将不复存在，它的所有内存和状态都会被新进程取代。

生活化类比：“执行”一个进程就像是科幻电影里的“灵魂互换”。你的程序（灵魂A）决定不再做自己，它调用 `exec` 后，它的身体（进程空间）被另一个全新的灵魂（新进程）完全占据。原来的灵魂A从此消失，世界上只留下了那个拥有旧身体的新灵魂。这与“生成”子进程完全不同，后者更像是生一个孩子，父子同时存在。

这种机制在需要将当前程序的执行权完全交给另一个程序时非常有用，例如在 shell 或进程管理器中。

82.2 使用 `syscall.Exec`

在 Go 中，`exec` 功能由 `syscall` 包提供。要执行一个新进程，你需要准备三样东西：

1. **要执行的程序的绝对路径**。
2. **传递给新程序的参数**（包括程序名本身）。
3. **新程序的环境变量**。

82.2.1 查找程序路径

`os/exec` 包的 `LookPath` 函数可以帮助我们在系统的 `PATH` 环境变量中查找一个可执行文件的绝对路径。

```
binary, lookErr := exec.LookPath("ls")
```

82.2.2 准备参数和环境

- **参数 (`args`)**: 必须是一个字符串切片。按照 Unix 的约定，切片的第一个元素应该是程序本身的名称。
- **环境 (`env`)**: `os.Environ()` 函数可以获取当前 Go 程序的所有环境变量，我们可以直接把它传递给新进程。

```
args := []string{"ls", "-a", "-l", "-h"}
env := os.Environ()
```

82.2.3 执行 `syscall.Exec`

`syscall.Exec` 会执行 `exec` 操作。如果这个调用成功，它将**永远不会返回**，因为 Go 程序已经被新进程替换了。如果它返回了，那一定是因为发生了错误。

```
execErr := syscall.Exec(binary, args, env)
if execErr != nil {
    panic(execErr)
}
```

82.3 完整代码示例

```
package main

import (
    "os"
    "os/exec"
    "syscall"
)

func main() {

    // --- 步骤 1: 查找要执行的命令的绝对路径 ---
    // 例如，我们要执行 `ls` 命令。`exec.LookPath` 会在系统的 PATH 中搜索 `ls`。
    binary, lookErr := exec.LookPath("ls")
    if lookErr != nil {
        // 如果找不到命令，则 panic。
        panic(lookErr)
    }

    // --- 步骤 2: 准备新进程的参数 ---
    // `Exec` 需要一个参数切片。按照约定，第一个参数应该是程序本身的名称。
    args := []string{"ls", "-a", "-l", "-h"}

    // --- 步骤 3: 准备新进程的环境变量 ---
    // `Exec` 还需要一个环境变量的切片。
    // 这里我们直接使用当前 Go 程序的环境变量。
    env := os.Environ()

    // --- 步骤 4: 执行 `exec` 调用 ---
    // `syscall.Exec` 是执行操作的核心。
    // 如果此调用成功，当前的 Go 进程将立即停止，并被 `ls -a -l -h` 进程完全取代。
    // Go 程序中此调用之后的任何代码都将不会被执行。
```

```
    execErr := syscall.Exec(binary, args, env)

    // --- 只有在 `Exec` 失败时，才会执行到这里 ---
    if execErr != nil {
        // 如果 `Exec` 调用返回一个错误，说明替换失败了。
        panic(execErr)
    }
}
```

82.4 运行结果

当你运行这个 Go 程序时，你不会看到任何 Go 程序的输出。相反，你的终端会直接显示 `ls -a -l -h` 命令的输出，因为 Go 程序已经被 `ls` 进程替换了。

```
$ go run execing-processes.go
total 24
drwxr-xr-x   5 user  group   160B Oct 27 10:20 .
drwxr-xr-x  39 user  group   1.2K Oct 27 10:15 ..
-rw-r--r--   1 user  group   533B Oct 27 10:15 spawning-processes.go
-rw-r--r--   1 user  group   433B Oct 27 10:20 execing-processes.go
```

(注：`ls` 的输出会因你的文件系统内容而异)

82.5 关键要点

1. **生成 (Spawning) vs 执行 (Exec'ing)**：这是两个完全不同的概念。**生成**是创建一个子进程，父进程继续存在；**执行**是用一个新进程完全替换当前进程。
2. **`syscall.Exec`**：是 Go 中实现 `exec` 操作的底层函数。
3. **永不返回**：成功的 `syscall.Exec` 调用不会返回到 Go 程序中。如果它返回了，就意味着发生了错误。
4. **参数和环境**：你需要为新进程明确提供程序路径、所有参数（包括程序名）和环境变量。
5. **与 `fork` 的关系**：Go 没有提供传统的 Unix `fork` 系统调用。在 Go 中，通常通过组合使用 goroutines、生成进程和执行进程来满足 `fork` 的各种使用场景。

83. Signals (信号处理)

https://gobyexample.com/signals

83.1 核心概念

信号（Signals）是 Unix-like 系统中进程间通信的一种方式，通常用于通知进程发生了某个事件。例如，当你按下 `Ctrl+C` 时，操作系统会向当前前台进程发送一个 `SIGINT`（中断）信号。在 Go 中，可以优雅地处理这些信号，例如在程序退出前执行清理操作。

生活化类比：信号处理就像是给你的程序装了一个“紧急呼叫”接收器。程序平时在正常工作，但它会一直听着一个特殊的频道（`sigs` channel）。当操作系统（比如你按下了 `Ctrl+C`）向这个频道发送一个紧急呼叫（`SIGINT` 信号）时，程序就会接收到，然后执行预设的应急预案（比如保存工作、清理现场），最后再安全地退出。

83.2 处理信号

Go 通过 `os/signal` 包将传入的信号转发到一个 channel 中，从而将信号处理整合到 Go 的并发模型里。

83.2.1 创建接收信号的 Channel

首先，需要创建一个 channel 来接收信号。这个 channel 的类型是 `os.Signal`。

```
sigs := make(chan os.Signal, 1)
```

注意：这个 channel 应该是一个缓冲 channel，因为信号通知的实现是非阻塞的。如果 channel 已满，信号可能会被丢弃。

83.2.2 注册要监听的信号

`signal.Notify` 函数用于注册给定的 channel，使其开始接收特定信号的通知。

```
// 让 sigs channel 开始接收 SIGINT 和 SIGTERM 信号。
// 如果不提供任何信号参数，它将监听所有传入的信号。
signal.Notify(sigs, syscall.SIGINT, syscall.SIGTERM)
```

83.2.3 等待信号

由于从 channel 接收信号是一个阻塞操作，所以通常会在一个单独的 goroutine 中执行，以避免阻塞主程序的执行。

```
done := make(chan bool, 1) // 创建一个用于同步的 channel

go func() {
    // 在这个 goroutine 中，程序会阻塞，直到从 sigs channel 中接收到一个信号。
    sig := <-sigs
    fmt.Println()
    fmt.Println(sig) // 打印接收到的信号
    done <- true     // 通知主 goroutine 可以退出了
}()

// 主 goroutine 在这里阻塞，等待信号处理 goroutine 发送完成信号。
fmt.Println("awaiting signal")
<-done
fmt.Println("exiting")
```

83.3 完整代码示例

```
package main

import (
    "fmt"
    "os"
    "os/signal"
    "syscall"
)
```

```go
func main() {

    // --- 步骤 1: 创建一个用于接收信号的 channel ---
    // Go 通过 channel 来传递信号通知。我们创建一个缓冲 channel，
    // 大小为1，以防止在还没准备好接收时错过信号。
    sigs := make(chan os.Signal, 1)

    // --- 步骤 2: 注册要监听的信号 ---
    // `signal.Notify` 将指定的 channel 注册为给定信号的接收者。
    // 这里我们希望捕获 `SIGINT` (Ctrl+C) 和 `SIGTERM` (由 `kill` 命令发送)。
    signal.Notify(sigs, syscall.SIGINT, syscall.SIGTERM)

    // --- 步骤 3: 创建一个用于程序同步的 channel ---
    // 这个 channel 将用于在信号处理完成后，通知主程序可以安全退出了。
    done := make(chan bool, 1)

    // --- 步骤 4: 启动一个 goroutine 来等待并处理信号 ---
    // 这是一个常见的模式，因为从 channel 接收信号是阻塞的，
    // 我们不希望它阻塞主程序的其他工作。
    go func() {
        // 程序会在这里阻塞，直到接收到一个在上面注册过的信号。
        sig := <-sigs
        fmt.Println() // 打印一个换行符，使输出更整洁
        fmt.Println(sig) // 打印接收到的信号，例如 "interrupt"

        // 当信号处理完成后，向 done channel 发送一个值。
        done <- true
    }()

    // --- 步骤 5: 主程序继续执行，并等待完成信号 ---
    fmt.Println("awaiting signal")
    // 主 goroutine 会在这里阻塞，直到从 done channel 接收到值为止。
    // 这确保了程序在信号处理逻辑执行完毕后才退出。
    <-done
    fmt.Println("exiting")
}
```

83.4 运行示例

1. 运行程序
程序会启动并打印 `awaiting signal`，然后等待。

```
$ go run signals.go
awaiting signal
```

2. 发送信号
在前台运行程序时，按下 `Ctrl+C`。这会向程序发送 `SIGINT` 信号。

3. 观察输出
程序捕获到信号，打印信号名称，然后优雅地退出。

```
$ go run signals.go
awaiting signal
^C
interrupt
exiting
```

83.5 关键要点

1. **`os/signal` 和 `syscall`**：是 Go 中处理系统信号的核心包。
2. **基于 Channel 的信号处理**：Go 将信号抽象为可以通过 channel 传递的值，这使得信号处理可以无缝地融入 Go 的并发模型。
3. **`signal.Notify`**：是连接系统信号和 Go channel 的桥梁。
4. **优雅关闭 (Graceful Shutdown)**：信号处理是实现服务优雅关闭的关键。当接收到 `SIGINT` 或 `SIGTERM` 时，程序可以在退出前完成当前正在处理的任务、关闭数据库连接、保存状态等。
5. **Goroutine 模式**：在单独的 goroutine 中阻塞等待信号是一种标准实践，它避免了阻塞主程序的正常逻辑。

84. Exit (退出)

https://gobyexample.com/exit

84.1 核心概念

在 Go 中，可以使用 `os.Exit` 函数来让程序以给定的状态码立即退出。一个重要的特性是，与正常的从 `main` 函数返回不同，**`os.Exit` 不会执行任何 `defer` 语句**。

生活化类比：`os.Exit` 就像是按下了电影院的紧急疏散按钮。一旦按下，所有电影（程序）会立即停止，灯光全亮，所有人（所有进程）必须立刻离场。它不会等待电影播放到结尾（函数正常返回），也不会执行映后打扫（`defer` 清理工作）。这是一种强制、立即终止的方式。

84.2 使用 `os.Exit`

`os.Exit` 接收一个整数作为退出状态码。按照惯例：

- 状态码 `0` 表示成功。
- 非零状态码（通常是 `1` 或更大）表示发生了错误。

```
// 使用状态码 3 立即退出程序
os.Exit(3)
```

`os.Exit` 与 `defer`

一个需要特别注意的关键点是，`os.Exit` 会立即终止程序，这意味着在调用它之前注册的 `defer` 函数将没有机会执行。

```
// 这条 defer 语句将不会被执行
defer fmt.Println("!")

os.Exit(3)
```

84.3 完整代码示例

```
package main

import (
    "fmt"
    "os"
)

func main() {

    // --- defer 语句的注册 ---
    // defer 会将一个函数调用推迟到其所在的函数即将返回之前执行。
    // 但是，如果程序通过 os.Exit 退出，defer 将不会被执行。
    defer fmt.Println("!")

    // --- 调用 os.Exit ---
    // os.Exit 会立即以给定的状态码终止当前程序。
    os.Exit(3)

    // --- 不会执行到的代码 ---
    // 由于 os.Exit 立即终止了程序，这行代码永远不会被执行。
    fmt.Println("This line will not be reached.")
}
```

84.4 运行示例

1. 使用 `go run` 运行
当你使用 `go run` 时，如果程序以非零状态码退出，Go 的运行工具会捕获并打印这个退出状态。

```
$ go run exit.go
exit status 3
```

注意，`!` 没有被打印出来。

2. 构建并手动运行
你可以先构建程序，然后运行它，并检查其退出状态码。在 Linux/macOS 中，可以使用 `echo $?` 来查看上一个命令的退出码。

```
# 构建可执行文件
$ go build exit.go

# 运行程序（它不会有任何输出）
$ ./exit

# 检查退出码
$ echo $?
3
```

84.5 关键要点

1. **`os.Exit` 用于立即终止**：它是一种强制、立即停止程序执行的方式。
2. **`defer` 不会被执行**：这是 `os.Exit` 和从 `main` 函数正常返回（`return`）之间最重要的区别。如果需要在程序退出前进行清理（如关闭文件、释放资源），应该避免使用 `os.Exit`，或者在调用它之前手动完成清理。
3. **状态码**：退出状态码是程序向其调用者（如 shell 脚本）传递成功或失败信息的一种标准方式。
4. **`main` 函数的返回**：Go 程序不像 C 语言那样通过 `main` 函数的返回值来传递退出码。在 Go 中，`main` 函数没有返回值，退出码必须通过 `os.Exit` 来指定。如果 `main` 函数正常结束而没有调用 `os.Exit`，程序会以状态码 `0` 退出。