Go学习笔记

2023.08.23

学习教程：尚硅谷 Golang入门到实战教程丨一套精通GO语言 共386集(117小时)

Go语言简介

诞生

Go语言（又称Golang）是由Google开发的一种编程语言。它的诞生背景可以追溯到2007年左右，以下是Go语言诞生的一些关键背景和动机：

1. **软件复杂性的挑战：** 在Google内部，软件系统变得越来越复杂，开发和维护变得困难。Google当时主要使用C++和Python等语言，但这些语言在大规模系统开发中可能存在一些问题，如编译速度慢、难以处理并发等。
2. **开发效率：** Google希望有一种可以提高开发效率、降低维护成本的新型编程语言。
3. **并发需求：** 随着多核处理器的普及，编写高效的并发代码变得越来越重要。Google内部许多任务都需要处理高并发，现有语言中的并发支持可能不够满足需求。
4. **编译速度：** 在Google这样的大型组织中，编译速度也变得至关重要。开发者希望有一种能够更快编译的语言，以加快迭代和测试。

基于以上的背景，Google决定开发一种新的编程语言，即Go语言。Go语言的开发始于2007年，其中主要的设计者包括Robert Griesemer、Rob Pike和Ken Thompson，他们都在Google工作。Go语言的设计注重简洁、可读性和高效性，同时提供了强大的并发支持，以满足Google内部开发的需求。

Go语言的公开发布是在2009年11月，这也是Go语言1.0版本的发布。随着时间的推移，Go语言受到了广泛的关注和采用，不仅在Google内部得到了应用，还在全球范围内的开发者社区中获得了越来越多的支持。至今，Go语言已经成为一种流行的编程语言，被用于开发各种类型的应用，从系统编程到网络服务和分布式系统等。

发展史

自Go语言于2009年发布1.0版本以来，Go团队不断进行了版本的更新和改进，每个版本都带来了新功能、改进和修复。以下是Go语言一些重要版本的简要概述：

1. **Go 1.0（2012年3月）：** Go 1.0标志着Go语言的正式发布。此版本的主要目标是稳定性和兼容性，确保代码在未来的版本中仍然有效。它引入了一些基本的标准库和语言特性。
2. **Go 1.1（2013年5月）：** 这个版本引入了一些小的语言和库的改进，以及对ARM架构的支持。
3. **Go 1.2（2013年12月）：** 引入了一些重要的改进，包括垃圾回收性能提升、工具链改进以及语言特性的增强。
4. **Go 1.3（2014年6月）：** 引入了新的Go工具、垃圾回收改进和一些语言特性的调整。
5. **Go 1.4（2014年12月）：** 该版本引入了新的编译器、新的垃圾回收器、并行测试等功能。
6. **Go 1.5（2015年8月）：** 引入了逐步采用的新的垃圾回收器、更好的并行编译器等。

> 本次更新中移除了"最后残余的C代码"。go1.5的发布被认为是历史性的。完全移除C语言部分，使用GO编译GO（ps：少量代码使用汇编实现），GO编译GO称之为Go的自举，是一门编程语言走向成熟的表现。

7. **Go 1.6（2016年2月）：** 引入了并行测试、内存性能优化、调度器等改进。
8. **Go 1.7（2016年8月）：** 引入了基于SSA的编译器、性能改进和一些库的更新。
9. **Go 1.8（2017年2月）：** 引入了HTTP/2支持、上下文包、性能改进等。
10. **Go 1.9（2017年8月）：** 引入了类型别名、Map类型的改进、性能优化等。
11. **Go 1.10（2018年2月）：** 引入了语法改进、并行编译性能优化等。
12. **Go 1.11（2018年8月）：** 引入了模块支持、WebAssembly支持、性能改进等。
13. **Go 1.12（2019年2月）：** 引入了模块支持的改进、更好的垃圾回收等。
14. **Go 1.13（2019年9月）：** 引入了模块支持的改进、更快的编译速度等。
15. **Go 1.14（2020年2月）：** 引入了模块支持的改进、新的内存分配器等。
16. **Go 1.15（2020年8月）：** 引入了链接器性能改进、错误处理的改进等。
17. **Go 1.16（2021年2月）：** 引入了嵌套的if/else、支持对错误的is操作等。
18. **Go 1.17（2021年8月）：** 引入了新的操作符、错误处理的改进、Fuzz测试支持等。
19. **Go 1.18（2022年3月）。**
20. **Go 1.19（2022年8月）。**
21. **Go 1.20（2023年2月）。**
22. **Go 1.21（2023年8月）。**

Go语言的版本更迭旨在持续改进语言和工具，以满足开发者的需求，提高性能和效率，并保持向后兼容性，以便现有的Go代码可以在新版本上继续运行。你可以在Go的官方网站上找到完整的版本发布历史和详细的变更说明。

参考：Go Release History(官方)

go语言图标的发展史是什么

特性

Go，通常称为Golang，是一种开源的编程语言，由Google于2007年启动并在2009年首次发布。Go的设计目标是提供一种简单、高效、可靠的编程语言，适用于大规模软件系统开发。它具有以下特点：

1. **并发支持：** Go内置了强大的并发支持，通过goroutine和channel来实现。Goroutine是一种轻量级的线程，可以在程序中同时运行成千上万个goroutine，而不会消耗大量的内存。Channel是一种用于在goroutine之间通信的机制，使并发编程更加简单和安全。
2. **快速编译：** Go编译器非常快速，从源代码到可执行文件的编译过程通常只需要几秒钟。这有助于开发者更迅速地迭代和测试代码。
3. **垃圾回收：** Go拥有自动垃圾回收（Garbage Collection）机制，开发者无需手动管理内存。这有助于减少内存泄漏和其他与内存管理相关的错误。
4. **静态类型：** Go是一种静态类型语言，这意味着变量的类型在编译时必须明确指定，这有助于提早捕获一些类型相关的错误。
5. **简洁的语法：** Go的语法设计简洁，去掉了一些冗余的特性，使代码更易于阅读和维护。

6. **面向接口：** Go鼓励使用接口（interfaces）进行抽象和多态编程。这使得代码更具灵活性，容易扩展和修改。
7. **内置工具：** Go附带了许多实用的工具，如格式化工具（`gofmt`）、文档生成工具（`godoc`）和测试工具（`go test`），这些工具有助于提高开发效率和代码质量。
8. **跨平台支持：** Go支持多种操作系统和硬件架构，可以在不同平台上进行开发和部署。
9. **开源：** Go是一个开源项目，其源代码可以在GitHub上找到，这意味着任何人都可以查看、使用和贡献代码。
10. **适用领域：** Go适用于各种类型的应用程序，从系统工具和网络服务器到分布式系统和云计算平台等。

官方文档

Golang 官方网站：https://go.dev/

Golang 官方标准库 API文档： https://pkg.go.dev/std 可以查看 Golang 所有包下的函数和使用。

Golang 中文网 在线标准库文档：https://studygolang.com/pkgdoc

准备工作

环境安装

安装Go语言的开发环境非常简单。以下是在常见操作系统上安装Go的步骤：

在 Windows 上安装 Go：

1. 前往 Go 官方下载页面，找到适用于 Windows 的安装包（`.msi` 文件）。
2. 下载并运行安装包。按照安装向导的指示进行安装。默认情况下，Go会安装到 `C:\Go` 目录。
3. 安装完成后，打开命令提示符（Command Prompt）或 PowerShell，并输入以下命令验证安装：

```
go version
```

在 macOS 上安装 Go：

1. 打开终端。
2. 使用以下命令安装 `brew`，如果你还没有安装它：

```
/bin/bash -c "$(curl -fsSL
https://raw.githubusercontent.com/Homebrew/install/HEAD/install.sh)"
```

3. 使用 `brew` 安装 Go：

```
brew install go
```

4. 安装完成后，在终端中输入以下命令验证安装：

```
go version
```

在 Linux 上安装 Go：

1. 打开终端。
2. 使用以下命令下载 Go 的安装包（根据你的系统架构选择合适的链接）：

```
wget https://golang.org/dl/goX.Y.Z.linux-amd64.tar.gz
```

3. 解压安装包并移动到适当的目录：

```
sudo tar -C /usr/local -xzf goX.Y.Z.linux-amd64.tar.gz
```

4. 配置环境变量。在终端中打开你的配置文件（如 `~/.bashrc`、`~/.zshrc` 等），添加以下行并保存：

```
export PATH=$PATH:/usr/local/go/bin
```

5. 在终端中输入以下命令刷新配置：

```
source ~/.bashrc
```

6. 验证安装：

```
go version
```

完成上述步骤后，你应该成功安装了 Go 的开发环境。现在你可以开始编写和运行 Go 程序了。

编程规范

Go语言有一套严格的编程规范，这些规范有助于保持代码的一致性、可读性和易于维护。以下是Go语言的一些主要编程规范：

1. **包名：** 包名应为小写字母，可以使用下划线分隔多个单词，遵循驼峰命名法。避免使用复数形式。
2. **导入语句：** 导入的包应使用全名，不要使用相对路径。导入语句按照标准库、第三方库、本地库的顺序分组。
3. **可见性：** 首字母大写的标识符是可导出的（public），可以在其他包中使用。首字母小写的标识符是私有的（private），只能在当前包内使用。
4. **注释：** 使用 `//` 进行单行注释，使用 `/* */` 进行多行注释。对于函数和变量，应编写清晰的注释来解释其功能。
5. **格式化：** 使用 `gofmt` 工具来格式化代码，保持一致的风格。Go语言官方推荐的格式化风格是没有缩进，只有一个Tab，且大括号与函数名同行。
6. **函数：** 函数名使用驼峰命名法，首字母大写表示导出，首字母小写表示私有。函数应当有明确的参数和返回值，避免过多的参数。
7. **错误处理：** 使用多值返回来返回函数的结果和错误。在函数中，如果出现错误，应将错误值返回给调用者。
8. **变量声明：** 使用短小的变量名，避免使用单个字符作为变量名，除非是临时的计数器或迭代变量。

9. **常量：** 使用全大写字母和下划线来表示常量，例如 `const MaxSize = 100`。
10. **枚举：** Go语言没有传统的枚举类型。可以使用常量来模拟枚举，也可以使用内置的 `iota` 常量生成器。
11. **结构体：** 结构体的字段名使用驼峰命名法，首字母大写表示导出，首字母小写表示私有。
12. **方法：** 方法的接收者类型应当尽量保持一致，要么是指针接收者，要么是值接收者，避免混用。
13. **错误类型：** 使用内置的 `error` 类型来表示错误，错误信息应当清晰、简洁并可读。
14. **空白导入：** 不应该使用空白导入，即 `import _ "package"`。
15. **包的划分：** 包应当有明确的功能，避免将太多不相关的功能放在同一个包中。
16. **测试：** 使用标准库的 `testing` 包来编写单元测试，并在测试文件中以 `_test.go` 结尾。

这只是Go语言编程规范的一部分。你可以在Go官方的 代码风格指南 中找到更详细的指导。遵循这些规范有助于写出清晰、一致和易于维护的Go代码。

注意事项

1. Go源文件以“go"为扩展名。
2. Go应用程序的执行入口是main()函数。
3. Go语言严格区分大小写。
4. Go方法由一条条语句构成，每个语句后不需要分号(Go语言会在每行后自动加分号)，这也体现出Golang的简洁性。
5. Go编译器是一行行进行编译的，因此我们一行就写一条语句，不能把多条语句写在同一个，否则报错。
6. **Go语言定义的变量或者import的包如果没有使用到，代码不能编译通过。**

   > 这个和其他编程语言是不一样的，未了防止多余变量或包的存在。

7. 大括号都是成对出现的，缺一不可。

简单案例

以下是一个简单的Go语言案例，展示了如何编写一个打印 "Hello, World!" 的程序：

```go
package main

import "fmt"

func main() {
    fmt.Println("Hello, World!")
}
```

让我们逐行解释一下这个程序：

- `package main`: 这是一个Go程序的起始点。每个Go程序都必须有一个`main`包，它包含了程序的入口函数。
- `import "fmt"`: 这里我们导入了标准库中的`fmt`包，它提供了格式化输出和输入的功能。

- `func main() { ... }`：这是程序的主函数。在Go中，每个可执行程序都必须有一个名为`main`的函数，它是程序的入口。代码块内的内容是主函数的实际逻辑。
- `fmt.Println("Hello, World!")`：这行代码使用了`fmt`包中的`Println`函数来打印文本。`Println`函数会在文本后添加一个换行符，所以输出会在终端上显示为一行。

要运行这个程序，你需要安装Go编译器，并将上述代码保存为`.go`文件，然后在终端中使用以下命令编译并运行程序：

```
go run filename.go
```

请将 "filename.go" 替换为你保存代码的文件名。

执行过程

Go 语言的代码执行过程涉及多个步骤，从代码编写到最终执行都包含在其中。以下是大致的执行过程：

1. **编写代码：** 首先，你需要编写 Go 代码。这可以在任何文本编辑器或集成开发环境（IDE）中完成。
2. **保存代码：** 将编写好的代码保存为以 `.go` 为后缀的文件，确保文件名符合Go的命名规范。
3. **编译：** 在终端中，使用 `go build` 命令编译你的代码。编译过程将把 Go 代码转换为可执行文件。如果你的代码有语法错误，编译器将会报告并指出错误所在。

   > 如：go build HelloWorld.go

4. **运行可执行文件：** 使用终端运行生成的可执行文件。例如，如果你的可执行文件名为 `myprogram`，那么在终端中运行：

```
./myprogram
```

5. **初始化：** Go 程序的执行从 `main` 包中的 `main` 函数开始。在 `main` 函数被调用之前，Go 运行时会进行一些初始化工作，例如设置运行时环境和加载包。
6. **执行 main 函数：** 一旦初始化完成，Go 运行时会调用 `main` 函数。这是程序的入口点，你可以在这里编写你的应用逻辑。
7. **并发执行：** 如果你在 `main` 函数中使用了 goroutine 和 channel，这些并发部分会在运行时创建并按需执行。
8. **结束程序：** `main` 函数执行完成后，程序将退出。如果你在代码中使用了类似 `os.Exit` 的方法，程序也可以在任何时候被显式地终止。

需要注意的是，Go 是一种编译型语言，但也有一些命令（例如 `go run`）可以一次性编译并运行代码。编译过程会将代码编译成机器码，这有助于提高执行效率。此外，Go 也支持交叉编译，允许你在一个操作系统上生成另一个操作系统的可执行文件。

语法基础

标识符的命名规则

在Go语言中，标识符是用来命名变量、函数、类型、常量等程序实体的名称。标识符的命名需要遵循一些规范和约定，以保持代码的可读性和一致性。以下是Go语言中标识符的命名规范：

1. **大小写敏感：** Go语言是大小写敏感的，因此 `myVar` 和 `MyVar` 是两个不同的标识符。
2. **只能以字母或下划线开头：** 标识符的第一个字符只能是字母（包括大小写）或下划线 `_`，不能以数字或其他特殊字符开头。
3. **后续字符可以是字母、数字或下划线：** 标识符的后续字符可以是字母、数字或下划线，但不能包含空格或其他特殊字符。
4. **不能使用关键字：** 不能将Go语言的关键字（如 `if`、`for`、`func` 等）作为标识符。
5. **约定使用驼峰命名法：** Go语言中的标识符通常使用驼峰命名法来命名，即将多个单词连接在一起，每个单词的首字母大写（除第一个单词外）。
6. **公开和私有标识符：** 标识符的首字母大写表示它是公开的，可以在不同包中访问。首字母小写的标识符是私有的，只能在同一个包中访问。

   > Go语言中没有`private`和`public`关键字，通过这种方式来表示权限。

7. **命名要有意义：** 标识符的命名应该具有描述性，能够清楚地表达其用途。避免使用过于简单或无意义的名称。
8. **遵循惯例：** 尽量遵循Go社区的命名惯例，例如使用 `camelCase` 来命名变量和函数，使用 `PascalCase` 来命名类型和导出的变量或函数。

   > 导出的变量或函数：权限为public，其他包也可以使用。

示例：

```go
package main

import "fmt"

func main() {
    var userName string // 符合驼峰命名法
    var user_name string // 不推荐的命名方式
    fmt.Println(userName)
}
```

遵循标识符的命名规范可以让你的代码更易于阅读、维护和协作。

运算符

Go语言支持多种运算符，用于执行各种数学和逻辑操作。以下是Go语言中常见的运算符分类：

1. **算术运算符：** 用于执行基本的数学操作。
   - `+`：加法
   - `-`：减法
   - `*`：乘法

- `/`：除法
- `%`：取模（求余数）

2. **赋值运算符：** 用于将值赋给变量。
   - `=`：赋值
   - `+=`：加法赋值
   - `-=`：减法赋值
   - `*=`：乘法赋值
   - `/=`：除法赋值
   - `%=`：取模赋值
3. **比较运算符：** 用于比较两个值之间的关系。
   - `==`：等于
   - `!=`：不等于
   - `<`：小于
   - `>`：大于
   - `<=`：小于等于
   - `>=`：大于等于
4. **逻辑运算符：** 用于执行逻辑操作。
   - `&&`：逻辑与
   - `||`：逻辑或
   - `!`：逻辑非
5. **位运算符：** 用于操作二进制位。
   - `&`：按位与
   - `|`：按位或
   - `^`：按位异或
   - `<<`：左移
   - `>>`：右移
6. **其他运算符：**
   - `&`：取地址运算符
   - `*`：指针运算符
   - `++`：自增
   - `--`：自减

需要注意的是，不同运算符之间的优先级是不同的，如果需要改变表达式的计算顺序，可以使用圆括号来明确优先级。

示例：

```go
package main

import "fmt"

func main() {
    a := 10
    b := 5

    fmt.Println(a + b) // 输出: 15
    fmt.Println(a - b) // 输出: 5
    fmt.Println(a * b) // 输出: 50
    fmt.Println(a / b) // 输出: 2
    fmt.Println(a % b) // 输出: 0

    x := 8
    y := 3
    fmt.Println(x == y) // 输出: false
    fmt.Println(x > y)  // 输出: true
    fmt.Println(x <= y) // 输出: false

    isTrue := true
    isFalse := false
    fmt.Println(isTrue && isFalse) // 输出: false
    fmt.Println(isTrue || isFalse) // 输出: true
    fmt.Println(!isTrue)           // 输出: false

    num := 8
    fmt.Println(num & 1) // 输出: 0（8的二进制是 1000，1的二进制是 0001，按位与后为 0000）
    fmt.Println(num | 1) // 输出: 9（按位或后为 1001）
    fmt.Println(num ^ 1) // 输出: 9（按位异或后为 1001）
    fmt.Println(num << 1) // 输出: 16（左移一位，相当于乘以2）
    fmt.Println(num >> 1) // 输出: 4（右移一位，相当于除以2）

    p := &num
    fmt.Println(p)  // 输出: 内存地址
    fmt.Println(*p) // 输出: 8（通过指针访问变量的值）
}
```

这些运算符是编写Go程序时常用的工具，能够处理各种数学计算、逻辑判断以及位操作等任务。

关于运算符的一道面试题

有两个变量，a 和 b，要求将其进行交换，但是不允许使用中间变量，最终打印结果。

```go
var a int = 10
var b int = 20
a = a + b
b = a - b           // b = (a+b)-b = a
a = a - b           // a = (a+b)-a = b
fmt.Println("a=", a) // 20
fmt.Println("b=", b) // 10
```

转义字符

Go语言中的转义字符用于表示一些特殊字符，例如换行符、制表符等。以下是一些常见的转义字符及其对应的含义：

- `\\`：反斜杠
- `\'`：单引号（使用\来转义，表示我们需要输出单引号，而不是字符串的结尾）
- `\"`：双引号
- `\n`：换行符
- `\r`：回车符
- `\t`：制表符（Tab）
- `\b`：退格符
- `\f`：换页符
- `\v`：垂直制表符
- `\0`：空字符
- `\xhh`：十六进制表示的字符，其中 `hh` 是两位十六进制数字
- `\uhhhh`：Unicode字符，其中 `hhhh` 是四位十六进制数字
- `\Uhhhhhhhh`：Unicode字符，其中 `hhhhhhhh` 是八位十六进制数字

这些转义字符可以在字符串字面值（用双引号或反引号括起来的内容）中使用，以便表示特殊字符。例如：

```
fmt.Println("Hello, \nWorld!")
```

这会打印出：

```
Hello,
World!
```

请注意，在原始字符串字面值（使用反引号括起来的内容）中，转义字符不会被解释，原样输出。例如：

```
fmt.Println(`Hello, \nWorld!`)
```

这会打印出：

```
Hello, \nWorld!
```

转义字符在Go语言中用于控制字符串的格式和特殊字符的表示，有助于编写更清晰和可读性更好的代码。

数据类型和变量

常见数据类型

Go语言中的数据类型用于定义变量、函数参数和返回值的类型。以下是Go语言中的一些常见数据类型：

1. **整数类型：**
   - `int`：根据系统架构，可以是32位或64位整数。
   - `int8`、`int16`、`int32`、`int64`：分别表示8位、16位、32位、64位有符号整数。
   - `uint`：根据系统架构，可以是32位或64位无符号整数。
   - `uint8`、`uint16`、`uint32`、`uint64`：分别表示8位、16位、32位、64位无符号整数。
2. **浮点数类型：**
   - `float32`：单精度浮点数。
   - `float64`：双精度浮点数。
3. **复数类型：**
   - `complex64`：由两个`float32`表示的复数。
   - `complex128`：由两个`float64`表示的复数。
4. **布尔类型：**
   - `bool`：表示真或假。
5. **字符串类型：**
   - `string`：表示文本字符串。
6. **字节类型：**
   - `byte`：`uint8`的别名，用于表示一个字节。
7. **符文类型：**
   - `rune`：`int32`的别名，用于表示一个Unicode字符。
8. **指针类型：**
   - `*T`：表示指向类型`T`的指针。
9. **数组类型：**
   - `[n]T`：表示由类型`T`组成的固定大小的数组，其中`n`是数组长度。
10. **切片类型：**
    - `[]T`：表示一个动态大小的切片，其中`T`是切片元素的类型。
11. **映射类型：**
    - `map[K]V`：表示一个映射，其中`K`是键的类型，`V`是值的类型。
12. **结构体类型：**
    - `struct`：自定义的结构类型，可以包含不同类型的字段。
13. **接口类型：**
    - `interface`：表示一组方法的集合，用于实现多态。

14. **函数类型：**
    - `func`：表示函数类型。

    > 在Go中，函数也是一种数据类型，可以赋值给一个变量，则该变量就是一个函数类型的变量了。
    > 通过该变量可以对函数调用。
    > 函数既然是一种数据类型，因此在Go中，函数可以作为形参，并且调用。

15. **通道类型：**
    - `chan T`：表示一个通道，其中 `T` 是通道元素的类型。
16. **自定义数据类型：**

为了简化数据类型定义，Go支持自定义数据类型。

基本语法: `type 自定义数据类型名 数据类型` // 理解: 相当于一个别名，但Go认为这是两种类型

> 案例1：`type mylnt int` // 这时mylnt 就等价int来使用了。（但不能将int类型的变量直接赋值给 mylnt 类型的变量，会报类型不相同的错误，需要显式转换）
>
> 案例2：`type mySum func(int,int) int` // 这时 mySum 就等价一个函数类型 func(int, int) int。

【注】前7种为基本数据类型。

> 和Java不同的是：Golang中没有专门的字符类型，而且string类型为基本数据类型。

这只是Go语言中一些常见的数据类型。通过使用这些数据类型，你可以声明并操作不同类型的变量、数据结构以及进行函数参数和返回值的定义。

各类型的默认值

在Go语言中，各个数据类型在未经初始化时都会有默认值。以下是一些常见数据类型的默认值：

1. **整数类型（包括有符号和无符号整数）：**
    - `int`、`int8`、`int16`、`int32`、`int64`：0
    - `uint`、`uint8`、`uint16`、`uint32`、`uint64`：0
2. **浮点数类型：**
    - `float32`：0.0
    - `float64`：0.0
3. **复数类型：**
    - `complex64`：0 + 0i
    - `complex128`：0 + 0i
4. **布尔类型：**
    - `bool`：false
5. **字符串类型：**
    - `string`：""
6. **字节类型：**
    - `byte`（实际上是 `uint8` 的别名）：0

7. **符文类型：**
    - `rune`（实际上是 `int32` 的别名）：0
8. **指针类型：**
    - 指针类型的默认值是 `nil`，表示指针未指向任何地址。
9. **切片类型：**
    - 切片类型的默认值是 `nil`，表示切片未指向任何底层数组。
10. **映射类型：**
    - 映射类型的默认值是 `nil`，表示映射未初始化。
11. **结构体类型：**
    - 结构体类型的默认值会根据字段的类型来确定，每个字段都会赋予其类型的默认值。
12. **接口类型：**
    - 接口类型的默认值是 `nil`，表示接口未指向任何值。

请注意，Go语言在变量声明时会自动为变量分配默认值，因此在未初始化之前，变量会具有其相应数据类型的默认值。

声明变量

在Go语言中，变量的声明使用关键字 `var` 或 `:=`（短变量声明）。以下是两种方式的示例：

使用 `var` 关键字声明变量

```go
var age int  // 声明一个名为 age 的整数变量
var name string  // 声明一个名为 name 的字符串变量

// 同时声明多个变量
var x, y int
var message string
```

使用 `:=` 短变量声明

短变量声明是Go语言中的一种便捷方式，用于声明并初始化变量。**它适用于局部变量，不能用于全局变量。**

```go
age := 25  // 声明并初始化一个名为 age 的整数变量
name := "Alice"  // 声明并初始化一个名为 name 的字符串变量

// 同时声明并初始化多个变量
x, y := 10, 20
message := "Hello, world!"
```

需要注意的是：

- 变量名在Go语言中遵循驼峰命名法，首字母小写表示私有变量，首字母大写表示公有变量（在包外可见）。
- 变量声明时必须指定变量的类型，除非使用短变量声明方式，Go会根据右侧的表达式推导出变量的类型。

- 使用 `:=` 短变量声明方式时，变量只能在函数内部使用，不能用于全局变量。

无论你使用 `var` 关键字还是 `:=` 短变量声明，Go语言都会自动推导变量的类型，这使得变量声明变得简洁和高效。

查看变量的类型

【方式1】格式化输出

使用Printf()函数的 %T，可以格式化输出变量的数据类型。

```
c = 12.2
fmt.Printf("变量c的类型：%T\n", c) // float64
```

> 注：这种方式底层也是使用的方式2中的reflect包的Typeof()函数。
>
> `p.fmt.fmtS(reflect.TypeOf(arg).String())`

【方式2】使用reflect包的Typeof()函数

在Go语言中，可以使用 `reflect` 包来查看变量的类型。`reflect` 包提供了一些函数和数据类型，用于在运行时获取变量的信息，包括其类型。以下是一个简单的示例：

```
package main

import (
    "fmt"
    "reflect"
)

func main() {
    var age int
    var name string

    age = 25
    name = "Alice"

    fmt.Println("age:", reflect.TypeOf(age))  // age: int
    fmt.Println("name:", reflect.TypeOf(name))  // name: string
}
```

在上面的示例中，我们导入了 `reflect` 包，并使用 `reflect.TypeOf()` 函数来获取变量的类型。这将打印出类似 `<type>` 的输出，表示变量的类型。注意，`reflect.TypeOf()` 返回的是 `reflect.Type` 类型，而不是字符串。

需要注意的是，Go语言是静态类型语言，变量的类型在编译时就已经确定了。`reflect` 包主要用于在运行时动态获取变量的类型和信息，通常用于编写通用函数或调试目的。不建议在正常的程序逻辑中频繁使用 `reflect` 包，因为它会引入一些性能开销。

类型转换

在Go语言中，严格来说是不能直接改变数据类型赋值的。Go是一门静态类型语言，变量在声明时就确定了其类型，之后无法直接改变变量的类型。例如，**你不能直接将一个整数类型赋值给一个字符串类型的变量。**

然而，你可以通过类型转换来实现从一个数据类型到另一个数据类型的转换。类型转换需要显式地指定目标类型，但在进行类型转换时需要注意以下几点：

1. **只能进行兼容的类型转换： 只能将兼容的类型进行转换**，例如数字类型之间的转换、字符串和字节切片之间的转换等。
2. **不支持隐式类型转换： Go语言不支持隐式的自动类型转换，因此你必须显式地执行类型转换。**

以下是一些示例：

```go
package main

import (
    "fmt"
)

func main() {
    var x int = 10
    var y float64 = float64(x) // 将整数类型转换为浮点数类型
    fmt.Println(y)

    var a float64 = 3.14
    var b int = int(a) // 将浮点数类型转换为整数类型（小数部分会被截断），b=3
    fmt.Println(b)

    var a int = 10
    //a = 10.2  // 注意，这样写会报错，因为a为int类型，不能保存float类型的数据
    fmt.Println("a=", a)

    var b float32 = 21.2
    b = 20 // 这里不会报错（因为20可以看做20.0），但b的类型还是float，而不是int
    fmt.Println("b=", b)
    fmt.Println("TypeOf(b)=", reflect.TypeOf(b)) // TypeOf(b)= float32

    c := 23
    d := 2
    e := c / d                                   // 这里e的类型，系统会推断为int而不是float，因为参与运算的c和d都是int类型
    fmt.Println("e=", e)                         // e= 11
    fmt.Println("TypeOf(e)=", reflect.TypeOf(e)) // TypeOf(e)= int

    // go语言中也不允许int类型和float类型相除
    //x := 100
    //y := 33.0
    //result := x / y // 报错：invalid operation: x / y (mismatched types float64 and int)
    //fmt.Println("result=", result)

```

```
    // 可以使用int()，float32()等函数，来进行强制类型转换
    // 但应该注意的是，由于Go语言中不存在隐式类型转换，所以强制类型转换不一定能得到我们想要的结果。
    var f_e float32 = float32(c / d) // 这里得到的f_e值为11，而不是11.5。
    // 因为这里是将c/d得到的11转换成float类型，而不是11.5。

    fmt.Println("f_e=", f_e)                            // 11
    fmt.Println("TypeOf(f_e)=", reflect.TypeOf(f_e)) // TypeOf(f_e)= float32
}
```

在这些示例中，我们使用类型转换将不同类型的数据进行了转换。请注意，类型转换可能会导致精度丢失或溢出，因此在进行类型转换时需要注意数据范围和精度。

细节说明：

1.**被转换的是变量存储的数据（即值），变量本身的数据类型并没有变化。**

```
var t1 int = 20
var t2 float32 = float32(t1)
fmt.Println("t1的类型：", reflect.TypeOf(t1)) // int
fmt.Println("t2的类型：", reflect.TypeOf(t2)) // float32
```

2.在转换中，比如将 int64 转成 int8 ，编译器不会报错，只是转换的结果是按溢出处理，和我们希望的结果不一样。

变量作用域

1. 函数内部声明/定义的变量叫局部变量，作用域仅限于函数内部。
2. 函数外部声明/定义的变量叫全局变量，作用域在整个包都有效，如果其首字母为大写，则作用域在整个程序有效。
3. 如果变量是在一个代码块，比如 for /if中，那么这个变量的的作用域就在该代码块。

格式化输出

在Go语言中，`fmt.Printf()` 函数用于格式化输出，它允许你将各种类型的值格式化成字符串，并在控制台打印出来。`Printf()` 函数使用格式字符串来指定输出的格式，格式字符串包含占位符，占位符会被实际的值替换。

以下是一些常见的格式化占位符及其用法：

1. **类型格式化：**
   - `%T`：变量的类型，例如 `int`、`string` 等。
2. **整数格式化：**
   - `%d`：有符号十进制整数。
   - `%b`：二进制表示。
   - `%o`：八进制表示。
   - `%x`、`%X`：十六进制表示（小写或大写）。
3. **浮点数格式化：**

- `%f`：浮点数（默认精度为6位）。
- `%.nf`：指定精度的浮点数（n代表要保留的小数位数）。

4. **布尔值格式化：**
   - `%t`：布尔值（true或false）。
5. **字符格式化：**
   - `%c`：字符（使用ASCII码值）。
6. **指针格式化：**
   - `%p`：指针的十六进制表示。
7. **宽度和对齐：**
   - `%5d`：至少占5个字符的整数，不足时用空格填充。
   - `%-5d`：至少占5个字符的整数，不足时用空格填充，左对齐。
   - `%05d`：至少占5个字符的整数，不足时用0填充。
8. **换行符和制表符：**
   - `\n`：换行。
   - `\t`：制表符。
9. **格式化输出结构体：**
   - `%v`：根据值的默认格式输出。
   - `%+v`：在每个字段上添加字段名。
   - `%#v`：输出值的 Go 语法表示。

示例：

```go
package main

import "fmt"

type Person struct {
    Name string
    Age  int
}

func main() {
    name := "Alice"
    age := 25
    height := 5.8
    person := Person{Name: "Bob", Age: 30}

    fmt.Printf("Name: %s, Age: %d, Height: %.2f\n", name, age, height)
    fmt.Printf("Name: %s, Age: %d\n", person.Name, person.Age)
    fmt.Printf("Person: %+v\n", person)
}
```

+号的使用

在Go语言中，`+` 号用于不同数据类型的加法操作。具体使用方式取决于操作数的类型。以下是 `+` 号的常见用法：

1. **整数和浮点数相加：**

```
a := 5
b := 3.14
result := a + b // result 的值为 8.14
```

2. **字符串相加（字符串拼接）：**

```
str1 := "Hello, "
str2 := "world!"
result := str1 + str2 // result 的值为 "Hello, world!"
```

3. **复数相加：**

```
c1 := complex(2, 3) // 2 + 3i
c2 := complex(1, 5) // 1 + 5i
result := c1 + c2   // result 的值为 3 + 8i
```

需要注意的是，`+` 号对于不同的数据类型有不同的行为。例如，对于整数和浮点数，`+` 号表示加法运算；对于字符串，`+` 号表示字符串的拼接；对于复数，`+` 号表示复数的加法运算。

如果操作数类型不匹配，将会导致编译错误。在字符串拼接时，Go语言会自动将其他数据类型转换为字符串，然后进行拼接。但是，对于其他类型的加法，数据类型必须匹配，否则会出现编译错误。

字符串底层相关

UTF-8编码

UTF-8 编码是一种变长的字符编码，它能够表示 Unicode 字符集中的所有字符，包括英文字母和汉字。在 UTF-8 编码下，不同的字符占用不同数量的字节，具体规则如下：

1. **ASCII 字符（0-127）：** ASCII 字符在 UTF-8 编码下使用一个字节表示，其编码值和 ASCII 码值相同。这包括英文字母、数字和常见符号。
2. **非 ASCII 字符：** 非 ASCII 字符（如汉字）在 UTF-8 编码下占用多个字节。UTF-8 使用多字节序列来表示这些字符，字节数的数量取决于字符的 Unicode 码点值。

对于英文字母（ASCII 范围内的字符），它们的 UTF-8 编码与其 ASCII 码值相同，因此在 UTF-8 编码下，英文字母仍然只占用一个字节。

对于汉字等非 ASCII 字符，UTF-8 编码使用多字节表示。常见的汉字通常使用 3 个字节来编码，但有些汉字会使用 4 个字节。具体的字节编码规则如下：

- 对于字符码点在 U+0000 到 U+007F 之间的字符，使用一个字节进行编码，与 ASCII 码兼容。
- 对于字符码点在 U+0080 到 U+07FF 之间的字符，使用两个字节进行编码。
- 对于字符码点在 U+0800 到 U+FFFF 之间的字符（包括常见的汉字），使用三个字节进行编码。

- 对于一些较大的字符码点，使用四个字节进行编码。

UTF-8 编码的设计目标是在保持与 ASCII 编码的兼容性的同时，能够表示更广泛的字符集，包括不同语言的字符和特殊符号。这使得 UTF-8 成为了一种通用的字符编码方式。

字符的底层存储

和其他编程语言不同的是，Golang底层存储字符串是用字节数组存的，而不是字符数组。

字符串使用UTF-8编码来存储Unicode字符。
在UTF-8编码中，每个字符由一个或多个字节组成。
ASCII字符（0-127）仍然使用单个字节表示，因此与ASCII兼容。
非ASCII字符（如汉字）使用多个字节表示，具体取决于字符的Unicode代码点。

对于单个字符，如果其码值小于等于255（在Ascii表里面），则可以直接保存到byte
如果其对应的码值大于255，则byte存放不下，可以考虑用int类型保存

```go
package main

import (
    "fmt"
    "reflect"
)

func main() {

    c1 := 'h'
    fmt.Println(c1)                             // 这里会输出h的ascii码104，而不是字符'h'
    fmt.Println("c1的类型：", reflect.TypeOf(c1)) // 系统推断的类型为：int32，也可以显式地指定类型为byte
    fmt.Printf("c1 = %c\n", c1)                 // 使用%c来格式化输出，c1=h

    var c2 byte = 'h'
    fmt.Println(c2)                             // 这里会输出h的ascii码104，而不是字符'h'
    fmt.Println("c2的类型：", reflect.TypeOf(c2)) // uint8（等价于byte）
    fmt.Printf("c2 = %c\n", c2)                 // h

    // 存储中文的情况
    c3 := '中'
    fmt.Println(c3)                             // 输出Unicode码值：20013
    fmt.Println("c3的类型：", reflect.TypeOf(c3)) // int32
    fmt.Printf("c3 = %c\n", c3)                 // 使用%c来格式化输出，c2=中

    // var c4 byte = '中'
    // 报错：cannot use '中' (untyped rune constant 20013) as byte value in variable declaration (overflows)
    // 因为byte类型存不下Unicode码值为20013的”中“字符，可以考虑用int来存。
}
```

字符串的底层存储

在Go语言中，字符串（`string`类型）是由一系列字节组成的，这些字节表示Unicode编码的字符。Go使用UTF-8编码来表示字符，UTF-8是一种可变长度的字符编码，它能够表示Unicode字符集中的所有字符。

在内存中，**Go语言的字符串是一个只读的字节数组，每个字节表示一个字符的UTF-8编码**。字符串的长度是字节数，而不是字符数。Go使用字节切片（`[]byte`）来表示字符串的底层数据，这是一种方便的方式来处理字符串的字节表示。

以下是一个示意图，表示字符串`"Hello"`的底层字节表示：

```
H   e   l   l   o
72  101 108 108 111
```

每个字节对应一个字符的UTF-8编码。字符串的索引是从0开始的，可以通过索引来访问字符串中的单个字节。这也意味着字符串的访问是O(1)时间复杂度。

需要注意的是，字符串是不可变的，一旦创建，就不能修改其内容。如果需要修改字符串，通常会将其转换为可变的字节切片，进行修改，然后再将字节切片转换回字符串。

总之，Go语言的字符串底层是由一系列字节组成的，使用UTF-8编码表示字符。字符串的索引访问和转换为字节切片是操作字符串的常见方式。

字符和字节的区别

字符和字节是计算机中的两个不同概念，虽然它们有些相似，但在某些情况下有着重要的区别。

字节（Byte）：

- 字节是计算机内存中存储数据的最小单位。
- 一个字节由8个比特（bits）组成。
- 字节是二进制数据的基本单位，在计算机中通常表示为整数值（0 到 255）。
- 字节用于存储各种数据，包括数字、字符、图像、音频等。

字符（Character）：

- 字符是人类语言的基本单位，表示一个字母、数字、标点符号或其他可见符号。
- 字符可以是单个字母（如英文字母）、数字、符号（如`!`、`#`）等。
- 不同的字符可能需要不同数量的字节来进行存储，这取决于字符编码方案（如ASCII、UTF-8、UTF-16等）。

区别总结：

- 字节是计算机内存存储的基本单位，通常由8个比特组成。
- 字符是人类语言的基本单位，通常由一个或多个字节组成，具体取决于字符的编码方式。
- 一个字节可以存储一个字符，也可以存储多个字符，具体取决于字符的编码方式。
- 字符和字节之间的关系取决于字符编码方案。在UTF-8编码中，一个字符可能由多个字节组成，而在ASCII编码中，一个字符只需要一个字节。

在处理文本数据时，了解字符和字节的区别非常重要，特别是在涉及不同字符编码的情况下。

存储英文字母

在Go语言中，英文字母是使用UTF-8编码来存储的，和其他Unicode字符一样。UTF-8是一种变长的字符编码方式，它将ASCII字符（0-127）表示为一个字节，这意味着英文字母在UTF-8编码下占用一个字节。

对于英文字母，其UTF-8编码与ASCII编码是兼容的。ASCII编码范围内的字符在UTF-8中保持不变，因此英文字母A到Z和a到z的UTF-8编码与其ASCII编码是一样的。

以下是一些英文字母的示例，展示了它们在UTF-8编码下的字节表示：

```
字符：  A    B    a    b
ASCII:  65   66   97   98
UTF-8:  41   42   61   62（16进制）
```

在这个示例中，字符A的ASCII编码是65，在UTF-8编码下也是65（16进制值为41），字符B的ASCII编码是66，在UTF-8编码下也是66（16进制值为42），以此类推。

在Go语言中，当你处理字符串时，无论是英文字母还是其他Unicode字符，Go会根据UTF-8编码自动进行字符解析。这使得在处理不同语言和字符集的文本数据时变得非常方便。

存储汉字

在Go语言中，字符串使用UTF-8编码来存储Unicode字符，包括汉字。UTF-8是一种变长的字符编码方式，它能够表示Unicode字符集中的所有字符，包括ASCII字符和非ASCII字符（如汉字）。

在UTF-8编码中，每个字符由一个或多个字节组成。ASCII字符（0-127）仍然使用单个字节表示，因此与ASCII兼容。非ASCII字符（如汉字）使用多个字节表示，具体取决于字符的Unicode代码点。

以下是一个示例，展示了UTF-8编码下汉字 "你好" 的字节表示：

```
字符：    你    好
Unicode: U+4F60 U+597D
UTF-8:   E4 BD A0 E5 A5 BD
```

在这个示例中，"你好" 这两个汉字的Unicode代码点分别为U+4F60和U+597D，它们的UTF-8编码分别为E4 BD A0和E5 A5 BD，这些编码表示了多个字节。

在Go语言中，当你创建一个字符串时，你实际上是在为每个字符使用UTF-8编码的字节序列分配内存。当你处理字符串时，Go会自动处理这些字节来表示正确的字符。这使得在Go中处理不同语言和字符集的文本数据变得非常方便。

目前学习到第41集。

2023.08.24

字符串的使用

在Go语言中，`string`是一种表示文本字符串的数据类型。字符串是不可变的，意味着一旦创建，就不能修改字符串的内容。字符串类型在Go中非常常见，用于存储和操作文本数据。

以下是有关Go语言中字符串类型的一些说明：

1. **字符串字面值：** 字符串字面值是用双引号 `""` 或反引号 ``` `` ``` 括起来的文本内容。双引号括起的字符串支持转义字符，而反引号括起的字符串是原始字符串，不会解释转义字符。
2. **字符串是不可变的：** 一旦创建，字符串的内容就不可更改。如果需要对字符串进行修改，通常需要将其转换为字节切片，然后操作字节切片，最后再转换回字符串。

```
var str1 string = "hello"
str1[0] = 'a'  // 报错：cannot assign to str1[0] (neither addressable nor a map index expression)
```

3. **字符串的长度：** 使用 `len()` 函数可以获取字符串的字节数（一个UTF-8编码字符可能占用多个字节）。

> 注意，这里返回的是字节数，而不是字符数。

```
var str1 string = "hello"
var str2 string = "中"
fmt.Printf("%d\n", len(str1))  // 5
fmt.Printf("%d\n", len(str2))  // 3，UTF-8编码中，一个汉字占三个字节
```

4. **字符串的索引和切片：** 可以通过索引获取字符串中的单个字符（字节），也可以使用切片操作来获取子字符串。Go中的索引是从0开始的。

```
var str1 string = "hello"

fmt.Printf("%s\n", str1)  // hello
fmt.Printf("%c\n", str1[0])  // h
fmt.Printf("%c\n", str1[1])  // e
```

5. **字符串的拼接：** 使用 `+` 号可以将两个字符串拼接在一起。但在大量拼接字符串时，最好使用 `strings.Join()` 函数或 `bytes.Buffer` 类型，以避免性能问题。

```
// 当字符串的拼接需要换行时，必须把“+”号写在上一行而不能写在下一行
// 因为编译器在每行的末尾自动加上“;”，而如果我们写了“+”号，就不会再添加“;”。
str2 := "hello" + "hello" +
 "hello" + "hello"
// str2 := "hello" + "hello"  // 这样写会报错
//   + "hello" + "hello"
fmt.Println(str2)
```

6. **字符串的迭代：** 可以使用 `range` 关键字迭代字符串，它会将字符串分解为单个字符（字节）。
7. **字符串的转换：** 使用 `strconv` 包可以进行字符串和其他数据类型（如整数、浮点数等）之间的转换。
8. **字符串的比较：** 使用 `==` 运算符可以比较两个字符串是否相等。Go语言中的字符串比较是基于字典序的。
9. **字符串处理包：** Go语言提供了 `strings` 包用于处理字符串，其中包含许多有用的函数，如查找、替换、分割等。
10. **Unicode和UTF-8：** Go语言的字符串使用UTF-8编码，因此它支持Unicode字符集的所有字符。

总之，字符串在Go语言中是非常重要的数据类型，用于处理文本数据。**要注意字符串是不可变的，对字符串的操作会生成新的字符串副本，而不是修改原始字符串。**

基本数据类型和string相互转换

在Go语言中，可以通过类型转换将不同类型的值相互转换。下面是基本数据类型和`string`类型之间相互转换的一些示例：

1. **基本数据类型转换为字符串：**
    - 1.使用`fmt.Sprintf()`函数。
    - 2.使用`strconv.Itoa()`函数（整数到字符串）

```go
package main

import (
    "fmt"
    "strconv"
)

func main() {
    // 使用 fmt.Sprintf()
    num := 42
    str1 := fmt.Sprintf("%d", num)
    fmt.Printf("String from int: %s\n", str1)

    // 使用 strconv.Itoa()（整数到字符串）
    // Itoa()等价于FormatInt(int64(i), 10)
    str2 := strconv.Itoa(num)
    fmt.Printf("String from int using strconv: %s\n", str2)

    // 使用 strconv.FormatFloat()（浮点数到字符串）
    f := 3.14159
    str3 := strconv.FormatFloat(f, 'f', 2, 64) // 'f': 格式，2: 小数位数，64: 位数
    fmt.Printf("String from float using strconv: %s\n", str3)
}
```

> 注意，不能使用string(num)，将数字转换为字符串。
>
> `string(g1)`这样的表达式将会把`g1`的整数值转换为对应的 Unicode 字符，然后将这个字符作为一个长度为 1 的字符串。因此，当你执行以下代码：
>
> ```go
> var g1 int = 65
> var g2 string = string(g1)
> ```
>
> 变量`g2`将会被赋值为一个包含一个 Unicode 字符的字符串，这个字符的 Unicode 码点是 65。在 Unicode 编码中，65 对应的字符是A。
>
> 要注意的是，这种整数到字符的转换通常用于处理字符编码和字符操作，而不是将整数转换为字符串的一种方式。

2. **字符串转换为基本数据类型：**
    - 使用 `strconv` 包中的函数可以将字符串转换为不同的基本数据类型。

```go
package main

import (
    "fmt"
    "strconv"
)

func main() {
    // 使用 strconv.Atoi()（字符串到整数）（等价于：ParseInt(s, 10, 0)）
    str1 := "42"
    // strconv.ParseXxx()函数有两个返回值：(i int64, err error)
    // 我们现在只需要第一个返回值，那么第二个返回值可以使用“_”接收，表示忽略该返回值。
    num1, _ := strconv.Atoi(str1)
    fmt.Printf("num2=%d\n", num1)  // num1= 42

    num2, _ := strconv.ParseInt(str1, 10, 64)
    fmt.Println("num2=", num2) // num2= 42

    // 使用 strconv.ParseFloat()（字符串到浮点数）
    str2 := "3.14"
    num2, _ := strconv.ParseFloat(str2, 64) // 64: 位数
    fmt.Printf("Float from string: %f\n", num2)

    // 使用 strconv.ParseBool()（字符串到布尔值）
    str3 := "true"
    boolVal, _ := strconv.ParseBool(str3)
    fmt.Printf("Bool from string: %t\n", boolVal)
}
```

> 注：如果string不能转换成指定的基本数据类型，则第一个参数得到该类型的默认值（int就是0，bool就是false），第二个参数的得到的是报错的信息。
>
> ```go
> str3 := "hello"
> str3Int, err := strconv.ParseInt(str3, 10, 64)
> fmt.Println("str3Int=", str3Int) // “hello”不能转换成Int，str3Int值为0
> fmt.Println("err=", err)         // err= strconv.ParseInt: parsing "hello": invalid syntax
> ```

这些示例演示了如何在Go语言中进行基本数据类型和 `string` 类型之间的转换。需要注意的是，类型转换可能会引发错误，你可以使用错误处理机制来处理转换可能的异常情况。

bool类型

在Go语言中，`bool` 类型是布尔类型，用于表示真（`true`）和假（`false`）两个值。布尔类型用于条件判断和逻辑操作，例如在条件语句和循环中。

以下是关于 Go 中 `bool` 类型的一些重要信息：

- `bool` 类型只有两个可能的值：`true` 和 `false`，它们都是预定义的标识符。
- 布尔类型在逻辑运算、条件语句和循环等地方非常常见。
- `if` 语句、`for` 循环、`while` 循环等都使用布尔表达式来判断条件是否成立。
- 布尔值可以用于与、或、非等逻辑运算，如 `&&`（与）、`||`（或）、`!`（非）。
- 在Go语言中，条件语句的判断部分必须是布尔表达式，不能使用非布尔类型进行隐式转换。

> 不可以使用0或非0的整数来替代false和true，这点和C语言不同。

示例：

```go
package main

import "fmt"

func main() {
    var isTrue bool = true
    var isFalse bool = false

    fmt.Println(isTrue)  // 输出: true
    fmt.Println(isFalse) // 输出: false

    age := 20
    isAdult := age >= 18
    fmt.Println("Is adult?", isAdult) // 输出: Is adult? true

    a := true
    b := false
    fmt.Println(a && b) // 逻辑与，输出: false
    fmt.Println(a || b) // 逻辑或，输出: true
    fmt.Println(!a)     // 逻辑非，输出: false
}
```

布尔类型在程序中用于控制逻辑流程和判断条件，是编程中非常重要的数据类型之一。

指针

在Go语言中，指针是一种特殊的数据类型，用于存储变量的内存地址。指针允许你直接访问内存中的数据，而不是通过变量名来访问。Go语言的指针提供了更直接的内存控制，同时也能减少内存和性能上的开销。

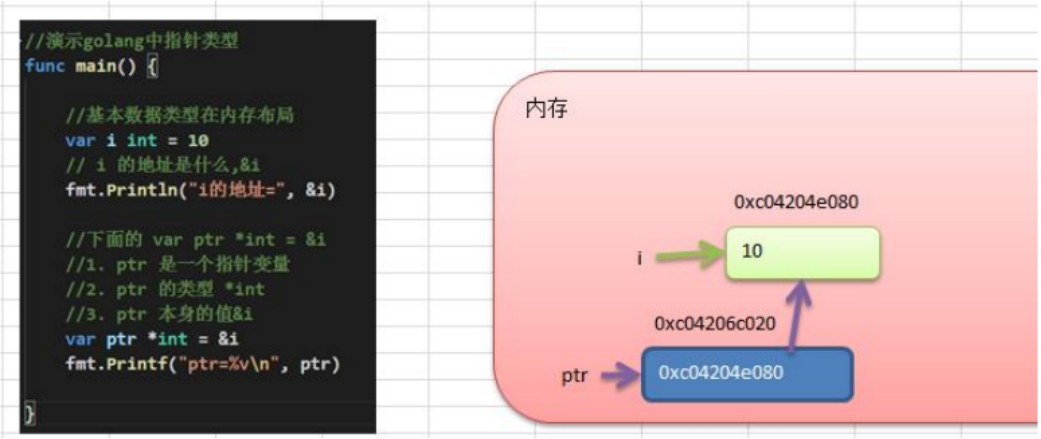
以下是关于Go语言中指针的一些重要概念和用法：

1. **创建指针：**
    - 使用 `&` 运算符可以获取变量的内存地址，从而创建一个指向该变量的指针。

```go
package main

import "fmt"

func main() {
```

```
    num := 42
    // var ptr *int = &num的解析：
    // 1.ptr 是一个指针变量
    // 2.ptr 的类型是 *int
    // 3.ptr 本身的值是 &num

    var ptr *int = &num
    fmt.Println("Value:", num)  // 输出: Value: 42
    fmt.Println("Pointer:", ptr) // 输出: Pointer: 0xc0000...（内存地址）
}
```

指针内存解析：



2. **解引用指针：**
    - 使用 `*` 运算符可以获取指针所指向的变量的值。

```
package main

import "fmt"

func main() {
    num := 42
    ptr := &num
    fmt.Println("Value:", *ptr) // 输出: Value: 42
}
```

3. **修改指针指向的值：**
    - 通过指针可以直接修改其指向的变量的值。

```go
package main

import "fmt"

func main() {
    num := 42
    var ptr := &num
    fmt.Println("Before:", *ptr) // 输出: Before: 42
    *ptr = 99
    fmt.Println("After:", *ptr)  // 输出: After: 99
}
```

4. **空指针：**
    - 指针的零值是 `nil`，表示空指针。在创建指针时没有显式赋值时，指针将自动初始化为 `nil`。

```go
package main

import "fmt"

func main() {
    var ptr *int
    if ptr == nil {
        fmt.Println("Pointer is nil")
    }
}
```

指针在Go语言中广泛用于多种场景，如在函数间传递大的数据结构、修改函数参数的值、在数据结构中创建链接等。然而，在使用指针时要小心，确保不会出现悬空指针或内存泄漏等问题。

指针使用细节说明

> 1.值类型，都有**对应的指针类型**， 形式为*数据类型，比如 int 的对应的指针就是 *int，float32对应的指针类型就是*float32、依次类推。
>
> 2.值类型包括：基本数据类型 int 系列，float 系列，bool，string，数组和结构体 struct。

值类型和引用数据类型

在Go语言中，数据类型可以分为值类型（Value Types）和引用类型（Reference Types）。这两种类型在内存中的存储方式和传递方式上有所不同。

值类型（Value Types）：
值类型直接存储数据的值，每个值都在内存中独立存在，不会因为复制而影响其他变量。当将值类型的数据传递给函数或者赋值给其他变量时，会发生数据的复制。常见的值类型有：基本数据类型（如整数、浮点数、布尔值、字符）、数组和结构体。

示例：

```go
package main

import "fmt"

func main() {
    a := 10
    b := a // 发生值的复制
    b = 20
    fmt.Println(a) // 输出: 10，a的值没有受到影响
}
```

引用类型（Reference Types）：
引用类型存储的是数据在内存中的地址，不直接存储数据的值。当将引用类型的数据传递给函数或者赋值给其他变量时，传递的是数据的引用（地址），而不是复制整个数据。常见的引用类型有：切片、映射（map）、通道（channel）、指针、接口等。

示例：

```go
package main

import "fmt"

func main() {
    slice1 := []int{1, 2, 3}
    slice2 := slice1 // 传递引用（切片的底层数组的引用）
    slice2[0] = 99
    fmt.Println(slice1) // 输出: [99 2 3]，slice1的值被影响
}
```

需要注意的是，虽然切片、映射和通道等在表现上更类似引用类型，但它们在内部实现上更复杂，使用了引用和底层数组的结合来提供更高效的内存管理和访问。因此，在Go语言中，不同于传统的引用类型，这些类型更常被称为“引用”或“引用型”。

获取键盘输入

`fmt.Scan()`、`fmt.Scanln()` 和 `fmt.Scanf()` 都是 Go 语言标准库 `fmt` 包中用于从标准输入（键盘）读取用户输入的函数，但它们之间有一些差异和用法上的不同。

1. `fmt.Scan()`:
    - `fmt.Scan()` 用于按空格分隔读取多个值，将用户输入的数据按空格分隔开，并依次存储到提供的变量中。用户输入的数据的数量必须与提供的变量数量相匹配。
    - 在遇到换行符（Enter键）之前，`fmt.Scan()` 会等待用户输入。
    - 通常用于读取多个值的情况，但输入时要注意空格的分隔。

```go
package main

import "fmt"

func main() {
    var name string
    var age int

    fmt.Print("Enter your name and age: ")
    fmt.Scan(&name, &age) // 读取用户输入并存储到 name 和 age 变量

    fmt.Printf("Name: %s, Age: %d\n", name, age)
}
```

2. `fmt.Scanln()`:
    - `fmt.Scanln()` 类似于 `fmt.Scan()`，但它会一直读取，直到遇到换行符（Enter键）为止，然后将输入的数据存储到提供的变量中。
    - 可以用于读取多个值，但不需要担心空格分隔的问题。

```go
package main

import "fmt"

func main() {
    var name string
    var age int

    fmt.Print("Enter your name: ")
    fmt.Scanln(&name) // 读取用户输入并存储到 name 变量

    fmt.Print("Enter your age: ")
    fmt.Scanln(&age) // 读取用户输入并存储到 age 变量

    fmt.Printf("Name: %s, Age: %d\n", name, age)
}
```

3. `fmt.Scanf()`:
    - `fmt.Scanf()` 是格式化输入函数，类似于 C 语言的 `scanf()`，它允许你指定输入的格式，然后根据格式读取用户输入的数据。
    - 在输入时，用户需要按照指定的格式输入数据，可以包含占位符。

```go
package main

import "fmt"

func main() {
    var name string
    var age int

    fmt.Print("Enter your name and age: ")
    fmt.Scanf("%s %d", &name, &age) // 读取用户输入并根据格式存储到 name 和 age 变量

    fmt.Printf("Name: %s, Age: %d\n", name, age)
}
```

总的来说，`fmt.Scan()`、`fmt.Scanln()` 和 `fmt.Scanf()` 都提供了不同的方式来读取用户输入。你可以根据具体的需求选择适合的函数。

进制和位移

进制的说明

Go语言支持多种进制表示，包括二进制、八进制、十进制和十六进制。这些不同的进制表示方式可以用来表示整数常量。以下是各种进制在Go语言中的说明：

1. **二进制表示（Binary）：**
   - 二进制数以 `0b` 或 `0B` 开头，后面跟着一串由 0 和 1 组成的数字。
   - 示例：`0b1101` 表示十进制的 13。
2. **八进制表示（Octal）：**
   - 八进制数以 `0` 开头，后面跟着一串由 0 到 7 的数字组成。
   - 示例：`0754` 表示十进制的 492。
3. **十进制表示（Decimal）：**
   - 十进制数是我们常用的十进制表示方式，不需要前缀，直接使用数字即可。
   - 示例：`123` 表示十进制的 123。
4. **十六进制表示（Hexadecimal）：**
   - 十六进制数以 `0x` 或 `0X` 开头，后面跟着一串由 0 到 9 和 A 到 F（大小写均可）的字母组成。
   - 示例：`0x1A` 表示十进制的 26。

示例代码：

```go
package main

import "fmt"

func main() {
    binaryValue := 0b1101   // 二进制
    octalValue := 0754      // 八进制
```

```
    decimalValue := 123    // 十进制
    hexadecimalValue := 0x1A // 十六进制

    fmt.Println("Binary:", binaryValue)         // 输出: Binary: 13
    fmt.Println("Octal:", octalValue)           // 输出: Octal: 492
    fmt.Println("Decimal:", decimalValue)       // 输出: Decimal: 123
    fmt.Println("Hexadecimal:", hexadecimalValue) // 输出: Hexadecimal: 26
}
```

在Go语言中，你可以根据需要使用不同的进制表示整数常量，这在处理各种问题时非常有用。

负数在底层表示

在Go语言中，负数是以二进制补码形式存储的。这种表示方式被广泛用于计算机中，它能够方便地处理整数的加减运算，而不需要额外的逻辑。以下是关于负数在Go语言底层的存放方式的一些解释：

1. **原码（Sign-Magnitude）：** 原码是一种最基本的整数表示方法。对于一个 N 位的有符号整数，它的最高位表示符号位，0 表示正数，1 表示负数。其余 N-1 位表示数值的绝对值。但原码的缺点是在加法和减法运算中需要额外的逻辑判断。
2. **反码（Ones' Complement）：** 反码是将原码中的除符号位外的每一位取反，0 变成 1，1 变成 0。这样可以方便实现负数的加减运算，但仍然需要处理有两个零的问题。
3. **补码（Two's Complement）：** 补码是目前最广泛使用的整数表示方法。负数的补码是正数的反码加 1。补码表示能够很好地处理加法和减法运算，而且没有两个零的问题。在补码表示中，负数的最高位为 1，其余位表示数值的绝对值。

Go语言使用补码来表示整数，包括负数。这种表示方式在计算机内部非常高效，并且允许简单而直接的加法和减法运算。因此，在Go语言中，负数以补码的形式存放在内存中。

位移操作

在Go语言中，负数的位移操作的结果是依赖于具体的机器架构和编译器实现的。在进行位移操作时，对于有符号整数，如果结果超出了位数，可能会导致未定义的行为。因此，应该避免对负数进行位移操作，以确保代码的可移植性和正确性。

对于负数的位移操作，可能的结果会因编译器和机器架构的不同而有所不同。一些常见的情况如下：

1. **右移操作：**
   - 对于有符号整数的右移操作（`>>`），**一般情况下，右移操作会将最高位的符号位进行扩展。这是为了保持负数的负号。**具体扩展的方式取决于机器的位数和编译器的实现。但在实践中，右移操作对于负数的结果是不确定的，应该避免使用。
2. **左移操作：**
   - 对于负数的左移操作（`<<`），同样存在不确定的情况，因为左移可能导致符号位溢出。

综上所述，负数的位移操作在Go语言中是不确定的行为，可能会因不同的编译器和机器架构而有所不同。因此，推荐避免对负数进行位移操作，以避免不确定性和错误。如果需要位移操作，建议使用无符号整数类型。

位移题目考察

位移运算后，a,b,c,d 结果是多少

```
var a int = 1 >> 2
var b int = -1 >> 2
var c int = 1 << 2
var d int = -1 << 2
```

解析：

```
a= 0
b= -1
c= 4
d= -4
```

这是因为这段代码中使用了位移运算符，而位移运算符会对整数在二进制位上进行移位操作。

1. `1 >> 2`：将二进制数 `0001` 向右移动两位，得到 `0000`，即十进制的 0。
2. `-1 >> 2`：在补码表示中，-1 的二进制是全1，向右移动两位得到 `1111`，再取补码得到 `-1`。
3. `1 << 2`：将二进制数 `0001` 向左移动两位，得到 `0100`，即十进制的 4。
4. `-1 << 2`：在补码表示中，-1 的二进制是全1，向左移动两位得到 `1100`，再取补码得到 `-4`。

包

在 Go 语言中，包（Package）是一种用于组织和管理代码的机制。一个包是一组相关的函数、类型、变量和常量的集合，可以将代码模块化、分层，并通过包名进行访问控制。包的使用有助于提高代码的可维护性、可读性和重用性。

以下是关于 Go 语言中包的一些重要概念：

1. **包的组织：**
   在 Go 中，每个源代码文件都属于一个包。包的名称通常与文件夹的名称相同。例如，一个名为 `example` 的包可以存储在名为 `example` 的文件夹中。

   > 同一个包不同文件中的代码是共享的。（等同于在同一个文件中）。所以，同一个包中不能有同名的全局变量、函数。
   >
   > 如果你要编译成一个可执行程序文件，就需要将这个包声明为main，即package main。这个就是一个语法规范，如果你是写一个库，包名可以自定义。

2. **包的导入：**
   在其他代码中使用一个包的功能时，需要使用 `import` 语句导入该包。导入的包名通常是包的路径，可以是标准库的包，也可以是自定义的包。

```
import "fmt"
import "math"
```

或者使用圆括号来导入多个包：

```
import (
    "fmt"
    "math"
)
```

> **如果包名较长，GO支持给包取别名**，注意细节：取别名后，原来的包名就不能使用了。
>
> ```
> package main
> import (
> "fmt"
> util "go_code/chapter06/fundemo01/utils"
> )
> ```
>
> 说明：如果给包取了别名，则需要使用别名来访问该包的函数和变量。

3. **可见性规则：**
   Go 语言中，标识符（函数、变量、类型等）的可见性由标识符的首字母大小写决定。以大写字母开头的标识符在包外部是可导出的，可以被其他包访问；以小写字母开头的标识符只在包内部可见。
4. **包的主要功能：**
   - 组织代码：将相关的功能组织在一个包中，使代码结构清晰。
   - 代码重用：通过导入其他包，可以在不同的项目中重用相同的功能。
   - 隔离作用域：不同的包具有独立的作用域，可以避免命名冲突。
   - 封装：通过限制外部访问可见的标识符，实现信息隐藏和封装性。
5. **自定义包：**
   开发者可以编写自己的包，将相关的代码放入其中，并按照可见性规则进行组织。自定义包可以被其他项目导入和使用。
6. **标准库包：**
   Go 语言标准库提供了大量的包，涵盖了各种基本功能，如输入输出、字符串处理、数学运算、网络通信等。这些标准库包在 Go 语言中非常常见，可以直接导入并使用。

分支循环结构

条件分支

if...else...

Go语言中的条件分支由 `if`、`else if` 和 `else` 关键字组成，用于根据条件来执行不同的代码块。以下是 Go 语言中条件分支的基本用法：

1. **if 语句：**
   `if` 语句用于执行一个代码块，当条件满足（为真）时。可以单独使用，也可以与 `else` 或 `else if` 配合使用。

```
if condition {
    // 如果条件为真，执行这里的代码
}
```

2. **if-else 语句：**
   `if-else` 语句用于在条件满足和条件不满足时执行不同的代码块。

```
if condition {
    // 如果条件为真，执行这里的代码
} else {
    // 如果条件不为真，执行这里的代码
}
```

3. **if-else if-else 语句：**
   `if-else if-else` 语句用于处理多个条件，根据不同的条件来执行不同的代码块。

```
if condition1 {
    // 如果条件1为真，执行这里的代码
} else if condition2 {
    // 如果条件2为真，执行这里的代码
} else {
    // 如果条件1和条件2都不为真，执行这里的代码
}
```

示例代码：

```
package main

import "fmt"

func main() {
    age := 18

    if age < 18 {
        fmt.Println("You are underage.")
    } else if age == 18 {
        fmt.Println("You just turned 18!")
    } else {
        fmt.Println("You are an adult.")
    }
}
```

需要注意的是，Go语言中的条件表达式不需要使用圆括号，但代码块（花括号包裹的部分）是必需的。

在使用多个条件分支时，注意条件的顺序，因为只会执行第一个满足条件的分支。

switch

Go语言中的 `switch` 语句用于根据一个表达式的值，在多个可能的情况中执行不同的代码块。`switch` 语句可以代替多个 `if-else` 分支，使代码更简洁和可读。以下是Go语言中 `switch` 语句的基本用法：

```
package main

import "fmt"
```

```
4
5  func main() {
6      day := 3
7
8      switch day {
9      case 1:
10         fmt.Println("Monday")
11     case 2:
12         fmt.Println("Tuesday")
13     case 3:
14         fmt.Println("Wednesday")
15     case 4:
16         fmt.Println("Thursday")
17     case 5:
18         fmt.Println("Friday")
19     default:
20         fmt.Println("Weekend")
21     }
22 }
```

在上面的示例中，`switch` 语句根据 `day` 的值执行不同的代码块。如果 `day` 的值是 3，则输出 "Wednesday"。**如果没有匹配的情况，会执行 `default` 分支。**

`switch` 语句还可以在 `case` 中使用多个值，或者不使用表达式，实现更灵活的匹配。此外，Go语言中的 `switch` 语句还可以不带表达式，用于处理布尔值的情况。

示例代码：

```
1  package main
2
3  import "fmt"
4
5  func main() {
6      fruit := "apple"
7
8      switch fruit {
9      case "apple", "pear":
10         fmt.Println("It's a fruit.")
11     case "carrot", "lettuce":
12         fmt.Println("It's a vegetable.")
13     default:
14         fmt.Println("Unknown.")
15     }
16 }
```

switch穿透——fallthrough

在 Go 语言中，`switch` 语句默认情况下是不会穿透（fallthrough）的，即当一个 `case` 匹配成功后，不会继续执行下一个 `case`。然而，你可以通过使用 `fallthrough` 关键字来实现 `switch` 语句的穿透效果，使其继续执行下一个 `case`。

下面是一个使用 `fallthrough` 的示例：

```go
package main

import "fmt"

func main() {
    num := 2

    switch num {
    case 1:
        fmt.Println("One")
        fallthrough
    case 2:
        fmt.Println("Two")
        fallthrough
    case 3:
        fmt.Println("Three")
    default:
        fmt.Println("Unknown")
    }
}
```

在上面的示例中，当 `num` 的值为 2 时，`case 2` 匹配成功后，`fallthrough` 使程序继续执行下一个 `case`，即 `case 3`，因此会输出：

```
Two
Three
```

需要注意的是，`fallthrough` **只会执行下一个** `case` **的代码块，不会检查下一个** `case` **的条件。**因此，在使用 `fallthrough` 时，要特别注意代码的逻辑和执行顺序，以免造成意料之外的结果。

在大多数情况下，`fallthrough` 的使用是可避免的，因为它可能会增加代码的复杂性和不可预测性。在实际编程中，要谨慎使用 `fallthrough`，并确保你了解它的影响。

嵌套条件分支

Go语言中的嵌套分支是指在一个条件分支的代码块内部再使用其他的条件分支结构。通过嵌套分支，可以更复杂地处理多个条件的情况。以下是Go语言中嵌套分支的示例：

```go
package main

import "fmt"

func main() {
    age := 18
    isStudent := true

    if age >= 18 {
        fmt.Println("You are an adult.")

        if isStudent {
```

```
            fmt.Println("You are a student.")
        } else {
            fmt.Println("You are not a student.")
        }
    } else {
        fmt.Println("You are underage.")
    }
}
```

在上面的示例中，首先检查了 `age` 是否大于等于 18，如果满足条件，则进入第一个嵌套的 `if` 代码块。在这个嵌套的 `if` 代码块中，又检查了 `isStudent` 的值，根据不同的情况输出相应的消息。

嵌套分支可以根据具体的需求进行多层嵌套，以处理更复杂的条件判断。但请注意，嵌套过深可能会导致代码可读性下降，应尽量保持代码的简洁和清晰。在需要使用多层嵌套时，考虑使用函数或提前进行条件判断，以避免过于复杂的代码结构。

循环结构

Go语言中的 `for` 循环是一种常用的循环结构，用于重复执行代码块。Go语言提供了多种形式的 `for` 循环，包括基本的 `for`、`for` + `range`、以及类似于其他编程语言的 `while` 形式的循环。以下是这些 `for` 循环的用法示例：

1.基本的 for循环

`for` 循环的基本形式包含初始化语句、循环条件和循环后操作，它们都使用分号分隔。

```
package main

import "fmt"

func main() {
    for i := 0; i < 5; i++ {
        fmt.Println(i)
    }
}
```

2.for+ range 循环

`for` 循环与 `range` 一起使用，用于遍历数组、切片、映射等集合类型。

```
package main

import "fmt"

func main() {
    numbers := []int{1, 2, 3, 4, 5}
    for index, value := range numbers {
        fmt.Printf("Index: %d, Value: %d\n", index, value)
    }
}
```

3.无限循环（类似于 while true）

Go语言的 `for` 循环可以省略初始化和后操作，实现类似于 `while true` 的无限循环。

```go
package main

import "fmt"

func main() {
    count := 0
    for {
        fmt.Println("This will run forever.")
        count++
        if count == 3 {
            break  // break：跳出循环
        }
    }
}
```

4.条件循环（类似于 while）

可以省略初始化和后操作，只保留循环条件，实现类似于 `while` 循环。

```go
package main

import "fmt"

func main() {
    num := 0
    for num < 5 {
        fmt.Println(num)
        num++
    }
}
```

Go语言中的 `for` 循环非常灵活，适用于多种不同的循环场景。根据具体的需求，你可以选择适合的循环形式。

> Go语言中没有while循环。

break和continue

在 Go 语言中，`break` 和 `continue` 都是用于控制循环流程的关键字。它们分别用于终止循环和跳过当前迭代，从而在循环中实现不同的行为。

break 语句

`break` 用于立即终止当前循环，不再执行循环中剩余的迭代。它通常在满足某个条件时使用，用于提前跳出循环。

```go
package main

import "fmt"

func main() {
    for i := 1; i <= 5; i++ {
        if i == 3 {
            break // 当 i 等于 3 时，终止循环
        }
        fmt.Println(i)
    }
}
```

在上面的示例中，当 `i` 等于 3 时，`break` 语句会终止循环，不再输出后续的数字。

continue 语句

continue` 用于跳过当前迭代，直接进入下一次迭代。它通常在满足某个条件时使用，用于跳过某些迭代。

```go
package main

import "fmt"

func main() {
    for i := 1; i <= 5; i++ {
        if i == 3 {
            continue // 当 i 等于 3 时，跳过这次迭代，进入下一次迭代
        }
        fmt.Println(i)
    }
}
```

在上面的示例中，当 `i` 等于 3 时，`continue` 语句会跳过这次迭代，直接进入下一次迭代，因此输出不包括数字 3。

带标签的 break 和 continue

在 Go 语言中，可以使用标签（label）来标记循环语句，然后在嵌套循环中使用带标签的 `break` 和 `continue` 来控制指定标签的循环。

```go
package main

import "fmt"

func main() {
OuterLoop:
    for i := 1; i <= 3; i++ {
        for j := 1; j <= 3; j++ {
            fmt.Printf("i: %d, j: %d\n", i, j)
            if i*j > 2 {
                break OuterLoop // 使用标签终止外层循环
```

```
            }
        }
    }
}
```

在上面的示例中，`break OuterLoop` 会终止外层循环。类似地，你也可以使用带标签的 `continue` 来跳过指定标签的迭代。

总之，`break` 和 `continue` 关键字可以在循环中控制代码的执行流程，使得循环更加灵活和可控。带标签的 `break` 和 `continue` 可以用于多重嵌套循环中，以实现精确的控制。但在使用标签时，要确保代码的可读性，避免过度复杂的嵌套结构。

goto语句

Go语言中提供了 `goto` 语句，它可以用来实现无条件的跳转到代码中的指定标签位置。然而，`goto` 语句容易导致代码的结构混乱，使得程序难以理解和维护，因此在实际编程中应该谨慎使用，尽量避免使用 `goto`。

以下是 `goto` 语句的基本语法：

```go
goto label
// ...
label:
// 这里是代码块
```

示例：

```go
package main

import "fmt"

func main() {
    i := 1

start:
    fmt.Println(i)
    i++
    if i <= 5 {
        goto start
    }
}
```

在上面的示例中，使用 `goto` 标签 `start` 实现了一个简单的循环。然而，这种使用方式容易导致代码变得混乱不堪，不利于代码的可读性和维护性。

在实际开发中，应该优先考虑使用结构化的控制流，如 `for`、`if`、`switch` 等，而不是使用 `goto`。结构化的控制流能够更清晰地表示代码逻辑，并且使得代码更易于理解和维护。只有在极少数情况下，当需要处理一些特殊情况时，才应该考虑使用 `goto`。

return语句

在 Go 语言中，`return` 语句用于从函数中返回一个值，并终止函数的执行。函数可以返回一个或多个值，也可以没有返回值。`return` 语句在函数内部使用，用于指定函数的返回值。

说明

> - 如果 return 是在普通的函数，则表示跳出该函数，即不再执行函数中 return 后面代码，也可以理解成终止函数。
> - 如果 return 是在 main 函数，表示终止 main 函数，也就是说终止程序。

以下是 `return` 语句的基本用法：

```go
func add(a, b int) int {
    result := a + b
    return result
}
```

在上面的示例中，函数 `add` 接受两个整数参数 `a` 和 `b`，然后计算它们的和，并通过 `return` 语句返回结果。

如果函数没有返回值，可以省略 `return` 语句中的表达式。例如：

```go
func greet(name string) {
    fmt.Println("Hello,", name)
    // 没有返回值的 return 语句
    return
}
```

如果函数声明了返回值类型，在函数体内就必须使用 `return` 语句来返回一个对应类型的值，否则会导致编译错误。

多个返回值的函数示例：

```go
func divide(a, b float64) (float64, error) {
    if b == 0 {
        return 0, errors.New("division by zero")
    }
    return a / b, nil
}
```

在上面的示例中，函数 `divide` 返回两个值，第一个值是 `a / b` 的结果，第二个值是可能的错误信息。如果除数 `b` 为零，函数返回一个错误信息。

总之，`return` 语句在 Go 语言中用于从函数中返回值，终止函数的执行。函数可以返回一个或多个值，也可以没有返回值。在编写函数时，根据函数的需求选择合适的返回值类型和返回值。

目前学习到第113集。

2023.08.25

函数

函数的定义

在 Go 语言中，函数是一种独立的代码块，用于执行特定的任务或操作。函数在程序中用于组织代码、实现代码的重用，以及将程序逻辑划分为更小的可管理单元。以下是关于 Go 语言中函数的一些重要特点和用法：

1. **函数的定义：**
   在 Go 语言中，函数的定义使用关键字 `func`，后面跟着函数的名称、参数列表、返回值类型以及函数体。

```go
func add(a, b int) int {
    result := a + b
    return result
}
```

2. **函数的参数：**
   函数可以接受零个或多个参数，参数列表位于函数名称后的括号内，多个参数使用逗号分隔。

```go
func greet(name string) {
    fmt.Println("Hello,", name)
}
```

3. **函数的返回值：**
   函数可以返回一个或多个值，也可以没有返回值。在函数签名中通过指定返回值的类型来定义函数的返回值。

```go
func divide(a, b float64) (float64, error) {
    if b == 0 {
        return 0, errors.New("division by zero")
    }
    return a / b, nil
}
```

> Go语言支持返回值命名：
>
> ```go
> func addSub(a, b int) (sum int, sub int) {
>     sum = a + b
>     sub = a - b
>     return
> }
> ```
>
> return后面不需要再写返回的值，系统会根据返回值的名称自动返回。

4. **多返回值：**
   Go 语言支持多个返回值，可以在函数签名中定义多个返回值的类型。

5. **匿名函数：**
   Go 语言支持匿名函数，也就是没有函数名的函数，可以在其他函数内部定义匿名函数。

```go
func main() {
    add := func(a, b int) int {
        return a + b
    }
    result := add(3, 5)
    fmt.Println(result) // 输出: 8
}
```

6. **可变参数函数：**
   Go 语言支持可变数量的参数，使用 `...` 表示，可变参数会被转化为一个切片。

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

> 可变参数只能写在参数列表的最后，否则会报错。

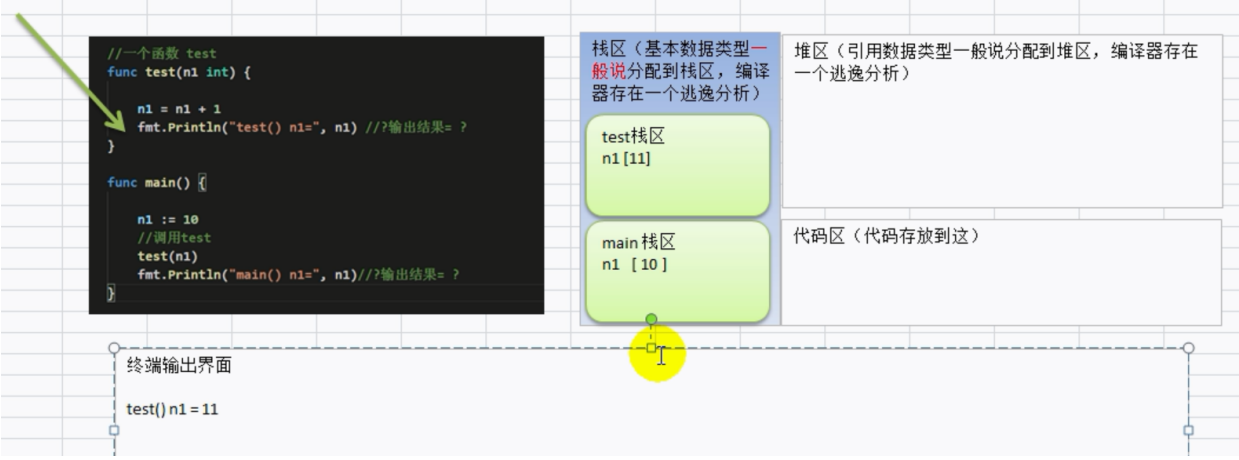
7. **函数作为参数和返回值：**
   在 Go 语言中，函数可以作为参数传递给其他函数，也可以作为返回值返回。

```go
func applyOperation(a, b int, operation func(int, int) int) int {
    return operation(a, b)
}

func multiply(x, y int) int {
    return x * y
}

func main() {
    result := applyOperation(3, 4, multiply)
    fmt.Println(result) // 输出: 12
}
```

8. **Go语言中不支持函数的重载**

总之，函数是 Go 语言中的重要组成部分，用于组织代码、实现代码重用和划分程序逻辑。了解函数的定义、参数、返回值、匿名函数、可变参数、函数作为参数和返回值等概念，有助于编写清晰、灵活和可维护的代码。

函数-调用过程



对上图说明
(1) 在调用一个函数时，会给该函数分配一个新的空间，编译器会通过自身的处理让这个新的空间和其它的栈的空间区分开来。
(2) 在每个函数对应的栈中，数据空间是独立的，不会混淆。
(3) 当一个函数调用完毕(执行完毕)后，程序销毁这个函数对应的栈空间。

递归调用

Go 语言支持函数的递归调用，就像许多其他编程语言一样。递归是一种编程技术，其中函数可以调用自身来解决问题。在使用递归时，关键是确保有基准情况（终止条件），以及每次递归都朝着基准情况逼近。

以下是一个简单的示例，演示了如何在 Go 中实现递归函数：

```go
package main

import "fmt"

func factorial(n int) int {
    // 基准情况：0的阶乘为1，1的阶乘也为1
    if n == 0 || n == 1 {
        return 1
    }

    // 递归调用：n的阶乘 = n * (n - 1)的阶乘
    return n * factorial(n-1)
}

func main() {
    num := 5
    result := factorial(num)
    fmt.Printf("%d的阶乘是%d\n", num, result)
}
```

在这个示例中，`factorial` 函数计算一个整数的阶乘。它在递归调用中不断减少输入的数字，直到达到基准情况（0或1），然后逐步返回计算结果。注意，递归函数必须在每个递归调用中趋近于基准情况，否则可能会导致无限递归。

尽管递归是一个有用的编程技术，但在实际使用时需要注意一些问题，比如递归深度过大可能会导致栈溢出。在某些情况下，迭代（循环）也可以用来替代递归，以避免这些问题。

函数递归需要遵守的重要原则：

1. 执行一个函数时，就创建一个新的受保护的独立空间(新函数栈)。
2. 函数的局部变量是独立的，不会相互影响。
3. 递归必须向退出递归的条件逼近，否则就是无限递归。
4. 当一个函数执行完毕，或者遇到return，就会返回，遵守谁调用，就将结果返回给谁。

一个帮助理解递归的案例

请说明下面这段代码的输出结果。

```
func main() {
    test(4)
}

func test(n int) {
    if n > 2 {
       n--
       test(n)
    }
    fmt.Println("n=", n)
}
```

解析：

1.第1次调用：test(4)

n=4，n>2，进入 if 语句块：

n--：n=3

test(3)：调用test(3)

2.第2次调用：test(3)

n=3，n>2，进入 if 语句块：

n--：n=2

test(2)：调用test(2)

3.第3次调用：test(2)

n=2，n>2，不满足 if 条件，往下执行输出语句：

fmt.Println("n=", n)：输出n=2

【注意】到这里，整个递归调用的过程并没有结束，而是回到前一个递归层次（这是很容易出错的地方）。

4.回到第2次调用，继续执行下面的输出语句：

fmt.Println("n=", n)：输出n=2（注意，这里的局部变量n值为2）

再回到前一个递归层次。

5.回到第1次调用，继续执行下面的输出语句：

fmt.Println("n=", n)：输出n=3

这已经是第1次调用了，整个递归调用结束。

综上，代码的输出结果为：

```
n= 2
n= 2
n= 4
```

注意，虽然 `test` 函数在递归时修改了 `n` 的值，但每个递归调用都有自己的局部变量副本，因此修改不会影响到其他调用。

值传递机制

在 Go 语言中，函数参数的传递方式是值传递（Pass by Value）。这意味着当你将一个参数传递给函数时，函数会接收该参数的一个副本，而不是原始数据本身。这样做的结果是，在函数内部对参数的修改不会影响到函数外部的原始数据。

以下是一个简单的例子，用于说明 Go 语言中的值传递机制：

```go
package main

import "fmt"

func modifyValue(x int) {
    x = 10
    fmt.Println("Inside function:", x)
}

func main() {
    num := 5
    modifyValue(num)
    fmt.Println("Outside function:", num)
}
```

在这个示例中，我们将一个整数 `num` 传递给了 `modifyValue` 函数。在函数内部，我们将参数 `x` 修改为 10，然后在函数外部打印 `num` 的值。

运行这段代码，你会看到输出如下：

```
Inside function: 10
Outside function: 5
```

从输出结果可以看出，在函数内部修改了 `x` 的值，但是函数外部的 `num` 的值并没有受到影响。这是因为函数参数的值传递方式导致函数内部操作的是参数的副本，而不是原始数据。

需要注意的是，**如果参数是指针类型，那么传递的是指针的副本，但仍然是值传递。如果在函数内部通过指针修改了指向的数据，那么函数外部的数据也会受到影响，因为它们指向同一块内存。**

> **如果希望函数内的变量能修改函数外的变量，可以传入变量的地址&，函数内以指针的方式操作变量。从效果上看类似引用。**

总结起来，Go 语言中的函数参数传递是值传递，这意味着函数内部操作的是参数的副本，而不会直接影响原始数据。

init()函数

在 Go 语言中，`init()` 函数是一个特殊的函数，它在程序运行时被自动调用。每个包可以包含一个或多个 `init()` 函数，它们用于执行一些初始化操作，如设置变量、执行计算等。这些初始化操作会在程序启动时自动执行，无需显式调用。

以下是关于 `init()` 函数的一些重要特点：

1. **自动调用：** `init()` 函数在程序启动时自动被调用，无需手动调用。每个导入的包中的 `init()` 函数都会被执行。
2. **执行顺序：** 如果一个包被导入多次，`init()` 函数只会执行一次。多个包的 `init()` 函数的执行顺序是不确定的，因此不应该依赖于其他包的 `init()` 函数的执行顺序。
3. **没有参数和返回值：** `init()` 函数没有参数和返回值。它的唯一目的是进行初始化工作。
4. **不可显式调用：** 你不能在代码中显式调用 `init()` 函数。它由 Go 编译器自动调用。
5. **在包级别：** `init()` 函数是在包级别上执行的，不会被外部调用，也不能在其他函数中被调用。

以下是一个示例，演示了如何在 Go 中使用 `init()` 函数：

```go
package main

import "fmt"

func init() {
    fmt.Println("Initialization from main package")
}

func main() {
    fmt.Println("Main function")
}
```

在这个示例中，`init()` 函数在 `main` 包中进行了初始化操作。当程序启动时，`init()` 函数会在 `main()` 函数之前被自动调用。

总之，`init()` 函数是一个在包初始化时自动执行的函数，用于执行一些初始化操作。在大多数情况下，我们会将初始化代码放在普通的函数中，但在需要在程序启动时立即执行的情况下，可以使用 `init()` 函数。

注：

> 如果一个文件同时包含全局变量定义，init() 函数和 main() 函数，则执行的流程：
> **全局变量定义 -> init() 函数 -> main() 函数**
>
> 详见案例：init_functuion

> 如果在程序中引入其他的包（包含init()函数），则先执行引入包的init()函数，然后在执行程序本身的init()函数。

匿名函数

Go 语言支持匿名函数。匿名函数是一种可以在代码中内联定义的函数，无需事先命名，通常用于需要定义一些临时逻辑的地方。它们可以作为参数传递给其他函数，或者直接被调用。

匿名函数使用方式1
　在定义匿名函数时就直接调用。
匿名函数使用方式2
　将匿名函数赋给一个变量(函数变量)，再通过该变量来调用匿名函数。
全局匿名函数
　如果将匿名函数赋给一个全局变量，那么这个匿名函数，就成为一个全局匿名函数，可以在程序有效。

以下是匿名函数的基本语法和用法示例：

```go
package main

import "fmt"

func main() {
    // 定义匿名函数并立即调用
    result := func(a, b int) int {
        return a + b
    }(3, 5)

    fmt.Println("Result:", result)

    // 将匿名函数赋值给变量，后续可通过变量调用
    multiply := func(x, y int) int {
        return x * y
    }

    product := multiply(2, 4)
    fmt.Println("Product:", product)
}
```

在上述示例中，我们定义了两个匿名函数。第一个匿名函数在定义时立即调用，计算了 3 和 5 的和。第二个匿名函数被赋值给了名为 `multiply` 的变量，然后通过该变量调用函数，计算了 2 和 4 的乘积。

总之，匿名函数是一种灵活的方式来定义临时的逻辑块，可以直接调用，也可以赋值给变量。它们在需要在某个作用域内定义短暂逻辑的情况下非常有用，尤其是与闭包结合使用时。

闭包

闭包（Closures）是指在**一个函数内部定义的函数，它可以访问其外部函数的变量**。换句话说，闭包包含了其声明时作用域内的变量，即使在其外部函数已经执行完毕后，闭包仍然可以访问和操作这些变量。

在 Go 语言中，匿名函数通常和闭包结合使用。闭包可以用来创建一些具有状态的函数，这些状态在函数调用之间保留。这使得闭包在并发编程中变得特别有用。

以下是一个示例，展示了如何在 Go 中使用闭包：

```go
package main

import "fmt"

func outerFunction() func() int {
    count := 0

    // 返回一个匿名函数
    return func() int {
        count++
        return count
    }
}

func main() {
    counter := outerFunction()

    fmt.Println(counter()) // 输出 1
    fmt.Println(counter()) // 输出 2
    fmt.Println(counter()) // 输出 3
}
```

在这个示例中，`outerFunction` 返回一个匿名函数。该匿名函数可以访问并修改 `outerFunction` 的局部变量 `count`。每次调用匿名函数，`count` 都会递增并返回新的值，这就形成了一个闭包，可以保留状态。

> 这个匿名函数和count这个变量共同构成了一个闭包。
>
> 大家可以这样理解: 闭包是类，函数是操作，n 是字段。函数和它使用到 n 构成闭包。
>
> 当我们反复的调用 counter 函数时，因为 count 是初始化一次，因此每调用一次就进行累计。
>
> 我们要搞清楚闭包的关键，就是要分析出返回的函数它使用(引用)到哪些变量，因为**函数和它引用到的变量共同构成闭包**。

闭包在一些需要保留状态的情况下非常有用，比如在创建计数器、实现延迟绑定（delayed binding）、生成不同的函数变体等场景中。

需要注意的是，闭包可能导致一些内存泄漏问题，因为闭包函数可以保留外部作用域的变量，从而延长了这些变量的生命周期。在使用闭包时，需要注意管理资源和变量的生命周期，以避免不必要的内存占用。

总之，闭包是 Go 语言中的一个重要概念，可以在函数内部创建保留状态的函数。它可以用于许多编程场景，但在使用时需要谨慎考虑资源管理和内存使用。

闭包的最佳实践

1. 编写一个函数 makeSuffix(suffix string) 可以接收一个文件后缀名(比如jpg)，并返回一个闭包。
2. 调用闭包，可以传入一个文件名，如果该文件名没有指定的后缀(比如.jpg)，则返回 文件名.jpg，如果已经有.jpg后缀，则返回原文件名。
3. 要求使用闭包的方式完成。
4. strings.HasSuffix：func HasSuffix(s, suffix string) bool，判断s是否有后缀字符串suffix。

```go
func main() {
    mySuffix := makeSuffix(".jpg") // 先获得一个闭包

    fmt.Println(mySuffix("hello.jpg")) // hello.jpg
    fmt.Println(mySuffix("winter"))    // winter.jpg
    fmt.Println(mySuffix("pic.png"))   // pic.png.jpg

}
func makeSuffix(suffix string) func(string) string {
    return func(name string) string {
       if strings.HasSuffix(name, suffix) {
          return name
       } else {
          return name + suffix
       }
    }
}
```

代码说明

1. 返回的匿名函数和 makeSuffix (suffix string)的 suffix 变量组合成个闭包，因为返回的函数用到suffix 这个变量。
2. 我们体会一下闭包的好处，如果使用传统的方法，也可以轻松实现这个功能，但是传统方法需要每次都传入 后缀名，比如 .jpg，而闭包因为可以保留上次引用的某个值，所以我们传入一次就可以反复使用。

defer

在 Go 语言中，`defer` 是一种用于延迟执行函数调用的机制。使用 `defer` 可以确保某个函数在当前函数执行完成后才被调用，无论函数是正常返回还是发生了错误。`defer` 通常用于在函数执行完毕后释放资源、关闭文件、解锁互斥锁等操作。

`defer` 语句将函数调用推迟到包含它的函数（或语句块）执行完毕后再执行。即使在函数内部多次使用 `defer`，它们的调用顺序仍然是后进先出（LIFO，Last-In-First-Out）的。

以下是一些示例，演示了如何使用 `defer` 语句：

示例1：

```go
func sum(n1 int, n2 int) int {
    //当执行到defer时，暂时不执行，会将defer后面的语句压入到独立的栈(defer栈)
    //当函数执行完毕后，再从defer栈，按照先入后出的方式出栈，执行
    defer fmt.Println("ok1 n1=", n1) //defer  3.ok1 n1= 10
    defer fmt.Println("ok2 n2=", n2) //defer  2.ok2 n2= 20
    res := n1 + n2                  // res = 30
    fmt.Println("ok3 res=", res)    // 1. ok3 res= 30
    return res
}
func main() {
    res := sum(10, 20)
    fmt.Println("res=", res) // 4.res= 30
}
```

在这个示例中，两个 `defer` 语句被使用，它们会在 `sum` 函数之后、回到 `main` 函数之前被调用。因此，输出顺序是：

```
ok3 res= 30
ok2 n2= 20
ok1 n1= 10
res= 30
```

示例2：

另一个示例，展示了 `defer` 如何用于资源管理：

```go
package main

import "fmt"

func main() {
    file := openFile("example.txt")
    defer closeFile(file)

    // 其他操作，如读取文件内容等
    fmt.Println("File operations here...")
}

func openFile(filename string) *File {
    fmt.Println("Opening file:", filename)
    // 模拟打开文件操作
    return &File{}
}

func closeFile(file *File) {
    fmt.Println("Closing file...")
    // 模拟关闭文件操作
}
```

在这个示例中，`openFile` 函数用于打开文件，`closeFile` 函数用于关闭文件。在 `main` 函数中，通过 `defer` 语句确保在执行完文件操作后关闭文件，即使出现错误也会执行关闭操作。

示例3

在defer 将语句放入到栈时，也会将相关的值拷贝同时入栈。请看一段代码：

```go
func sum(n1 int, n2 int) int {
    defer fmt.Println("ok1 n1=", n1) //defer  3.ok1 n1= 10（这里的n1是入栈时的值10而不是入栈后才增加的值11）
    defer fmt.Println("ok2 n2=", n2) //defer  2.ok2 n2= 20
    n1++                            // n1 = 11
    n2++                            // n2 = 21
    res := n1 + n2                  // res = 32
    fmt.Println("ok3 res=", res)    // 1. ok3 res= 32
    return res
}
func main() {
    res := sum(10, 20)
    fmt.Println("res=", res) // 4.res= 32
}
```

总之，`defer` 语句是 Go 语言中一种非常有用的机制，用于延迟执行函数调用，通常用于资源管理和清理操作。使用 `defer` 可以提高代码的可读性和维护性，确保重要的清理操作不会被遗漏。

常用字符串函数

在 Go 语言中，字符串是一个常见的数据类型，并且标准库提供了许多用于操作字符串的函数。以下是一些常用的字符串函数和方法：

1. **len(s string) int**： 返回字符串的字节数（不是字符数）。可以用来获取字符串的长度。
2. **str[i] byte**： 获取字符串的第 i 个字节（字节切片中的元素）。
3. **strings.HasPrefix(s, prefix string) bool**： 检查字符串是否以指定的前缀开头。
4. **strings.HasSuffix(s, suffix string) bool**： 检查字符串是否以指定的后缀结尾。
5. **strings.Contains(s, substr string) bool**： 检查字符串是否包含指定的子串。
6. **strings.Index(s, substr string) int**： 返回子串在字符串中第一次出现的索引，如果未找到则返回 -1。
7. **strings.LastIndex(s, substr string) int**： 返回子串在字符串中最后一次出现的索引，如果未找到则返回 -1。
8. **strings.Replace(s, old, new string, n int) string**： 将字符串中的旧子串替换为新子串，替换次数由 n 指定。
9. **strings.Count(s, substr string) int**： 统计子串在字符串中出现的次数。
10. **strings.ToUpper(s string) string**： 将字符串转换为大写。
11. **strings.ToLower(s string) string**： 将字符串转换为小写。
12. **strings.TrimSpace(s string) string**： 去除字符串两端的空白字符。
13. **strings.Trim(s string, cutset string) string**： 去除字符串两端指定的字符集合。
14. **strings.Split(s, sep string) []string**： 使用指定的分隔符将字符串拆分为子串切片。
15. **strings.Join(a []string, sep string) string**： 将字符串切片连接为一个字符串，使用指定的分隔符。
16. **strconv.Itoa(i int) string**： 将整数转换为字符串。
17. **strconv.Atoi(s string) (int, error)**： 将字符串转换为整数。

18. **fmt.Sprintf(format string, a ...interface{}) string：** 格式化输出，类似于 C 的 printf。

以上只是一些常用的字符串函数，Go 标准库还提供了更多的字符串处理函数。要了解更多细节，请查阅官方文档或其他资源。

日期时间函数

在 Go 语言中，时间和日期的处理涉及到 `time` 包。`time` 包提供了许多函数和类型，用于操作时间、日期以及时间间隔。以下是一些常用的日期时间函数和类型：

1. **time.Now() time.Time：** 获取当前的本地时间。
2. **time.Now().Unix() int64：** 获取自UTC 1970 年1月1日以来经过的秒数。

> 如果需要统计更加精确的时间，可以使用UnixMilli()(毫秒)/UnixMicro()(微秒)/UnixNano()(纳秒)函数，返回的就是对应粒度的时间。

3. **time.Date(year int, month Month, day, hour, min, sec, nsec int, loc *Location) time.Time：** 创建一个指定日期和时间的 `time.Time` 对象。
4. **time.Time.Format(layout string) string：** 将时间格式化为指定的字符串布局，例如："2006-01-02 15:04:05"。

```go
// 日期时间格式化
currentTime := time.Now()

// 格式化为 RFC3339 标准时间格式
fmt.Println("RFC3339:", currentTime.Format(time.RFC3339))

// 自定义格式化布局
fmt.Println("Custom format:", currentTime.Format("2006-01-02 15:04:05"))
// "2006-01-02 15:04:05"，这个字符串的各个数字是固定的，必须是这样写。
// 但是数字的组合格式可以自定义修改。
// 使用预定义的常用布局
fmt.Println("Standard layout:", currentTime.Format(time.RFC1123))

// 格式化为 Unix 时间戳
unixTimestamp := currentTime.Unix()
fmt.Println("Unix timestamp:", unixTimestamp)

// 解析 Unix 时间戳
unixTime := time.Unix(unixTimestamp, 0)
fmt.Println("Parsed Unix time:", unixTime.Format("2006-01-02 15:04:05"))
```

5. **time.Parse(layout, value string) (time.Time, error)：** 将字符串解析为时间对象，使用与 `Format` 函数相同的布局。
6. **time.Duration：** 表示时间间隔的类型，可以进行加减运算。
7. **time.Sleep(d Duration)：** 让程序休眠一段时间。

```go
// 休眠1秒
time.Sleep(time.Second)

// 休眠100毫秒
// 注意，休眠100毫秒只能写成time.Millisecond*100，不能写成time.Second*0.1
// 因为，Sleep函数能接受的参数类型为Duration类型：Sleep(d Duration)，其中type Duration int64
//time.Sleep(time.Second * 0.1)  // 报错：0.1 (untyped float constant) truncated to int64
time.Sleep(time.Millisecond * 100)
```

8. **time.After(d Duration) <-chan Time：** 返回一个通道，会在指定的时间间隔后发送当前时间。
9. **time.Tick(d Duration) <-chan Time：** 返回一个通道，会定期发送时间间隔。
10. **time.Since(t Time) Duration：** 返回当前时间与给定时间之间的时间间隔。
11. **time.Until(t Time) Duration：** 返回给定时间与当前时间之间的时间间隔。
12. **time.Time.Year() int：** 获取年份。
13. **time.Time.Month() Month：** 获取月份。
14. **time.Time.Day() int：** 获取日期。
15. **time.Time.Hour() int：** 获取小时。
16. **time.Time.Minute() int：** 获取分钟。
17. **time.Time.Second() int：** 获取秒数。
18. **time.Time.Location() *time.Location：** 获取时区信息。
19. **time.Time.Add(d Duration) Time：** 在当前时间上添加一个时间间隔。
20. **time.Time.Sub(t Time) Duration：** 计算两个时间之间的时间间隔。

这些函数和类型提供了对时间和日期的基本操作，可以用于计算时间差、格式化时间、解析时间字符串等操作。在处理时间和日期时，建议仔细查阅 `time` 包的官方文档，以了解更多详细信息和示例。

内置函数

常见内置函数

Go 语言内置了许多常用的函数，这些函数可以在任何地方直接使用，无需导入额外的包。以下是一些常见的内置函数：

1. **close(chan)：** 关闭通道，通知接收方不会再有更多的数据。
2. **len(v) int：** 返回字符串、数组、切片、映射、通道或数组的长度。
3. **cap(v) int：** 返回数组、切片或通道的容量。
4. **new(T) *T：** 创建一个新的 T 类型的零值，返回指向该值的指针。
5. **make(T, args) T：** 创建一个 T 类型的值，用于切片、映射和通道的初始化。
6. **append(s []T, x ...T) []T：** 向切片 s 追加元素 x，返回新的切片。
7. **copy(dst, src []T) int：** 将 src 中的元素复制到 dst，返回复制的元素个数。
8. **delete(m map[T]T, key T)：** 从映射 m 中删除指定的键值对。

9. **panic(v interface{})：** 引发 panic，通常用于异常情况。
10. **recover() interface{}：** 在 defer 函数中捕获 panic，通常用于恢复程序。
11. **print(args ...Type)：** 在标准输出上打印值，不添加换行符。
12. **println(args ...Type)：** 在标准输出上打印值，添加换行符。
13. **complex(r, i floatT) complexT：** 创建一个复数，其中 r 为实部，i 为虚部。
14. **real(c complexT) floatT：** 获取复数 c 的实部。
15. **imag(c complexT) floatT：** 获取复数 c 的虚部。
16. **panic和recover：** 用于处理 Go 语言的错误恢复机制。`panic` 用于引发异常，`recover` 用于在 `defer` 函数中捕获异常并进行处理。

这些是一些常见的内置函数，它们在 Go 语言的日常编程中非常有用。同时，Go 语言还提供了更多的内置函数，用于各种任务，包括类型转换、位操作、数学计算等。你可以查阅官方文档以获得完整的列表和详细信息。

new的使用

在 Go 语言中，`new` 是一个内置函数，用于创建一个新的值，并返回该值的指针。这个值的类型由传递给 `new` 的参数类型决定。在内存中，使用 `new` 创建的值会被初始化为该类型的零值。

`new` 函数的基本语法如下：

```
new(T)
```

其中，`T` 是一个类型，可以是基本数据类型、结构体、数组、指针等任何合法的类型。

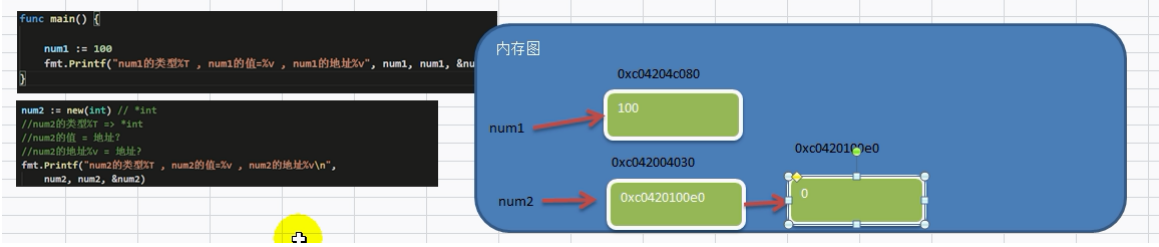
以下是一些示例，展示了如何使用 `new` 创建不同类型的值：

```
package main

import "fmt"

type Person struct {
    Name string
    Age  int
}

func main() {
    // 创建一个指向整数的指针
    intPointer := new(int)
    fmt.Println("Value of intPointer:", *intPointer) // 输出: 0

    // 创建一个指向 float64 的指针
    floatPointer := new(float64)
    fmt.Println("Value of floatPointer:", *floatPointer) // 输出: 0

    // 创建一个指向结构体的指针
    personPointer := new(Person)
    fmt.Println("Value of personPointer:", *personPointer) // 输出: { 0}
```

```
22
23      // 使用 new 创建的值的指针，可以像普通指针一样使用
24      *intPointer = 42
25      *floatPointer = 3.14
26      personPointer.Name = "Alice"
27      personPointer.Age = 30
28
29      fmt.Println("Value of intPointer:", *intPointer)
30      fmt.Println("Value of floatPointer:", *floatPointer)
31      fmt.Println("Value of personPointer:", *personPointer)
32  }
```

需要注意的是，`new` 创建的值是指向零值的指针。在使用 `new` 创建的指针后，你可以直接对指针所指向的值进行操作，就像对普通的指针所指向的变量进行操作一样。

参考示例图



总之，`new` 是 Go 语言中的一个内置函数，用于创建一个新的值，并返回该值的指针。这在创建结构体、基本数据类型和数组等类型时很有用。

make的使用

在 Go 语言中，`make` 是一个用于创建切片、映射和通道的内置函数。与 `new` 函数不同，`make` 函数返回的是特定类型（切片、映射或通道）的初始化后的值，而不是一个指针。因为切片、映射和通道都需要在底层进行一些初始化操作，所以需要使用 `make` 函数来创建它们。

`make` 函数的基本语法如下：

```
1  make(T, size)
```

其中，`T` 是切片、映射或通道的类型，`size` 是可选参数，表示创建的容量（对于切片和通道）或初始大小（对于映射）。

以下是一些示例，展示了如何使用 `make` 创建切片、映射和通道：

```
1  package main
2
3  import "fmt"
4
5  func main() {
6      // 创建一个切片，长度为 3，容量为 5
7      slice := make([]int, 3, 5)
8      fmt.Println("Slice:", slice)      // 输出：[0 0 0]
9      fmt.Println("Length:", len(slice)) // 输出：3
10     fmt.Println("Capacity:", cap(slice)) // 输出：5
```

```
11
12      // 创建一个映射
13      m := make(map[string]int)
14      m["one"] = 1
15      m["two"] = 2
16      fmt.Println("Map:", m) // 输出：map[one:1 two:2]
17
18      // 创建一个通道
19      ch := make(chan int)
20      go func() {
21          ch <- 42
22      }()
23      value := <-ch
24      fmt.Println("Channel value:", value) // 输出：42
25  }
```

需要注意的是，`make` 函数只适用于切片、映射和通道的创建。对于其他类型，如结构体、数组等，应使用 `new` 或直接声明变量来创建。

总之，`make` 是 Go 语言中的一个内置函数，用于创建切片、映射和通道，返回初始化后的值。使用 `make` 可以确保这些数据类型被正确初始化，以便能够正常使用。

make和new的区别

在 Go 语言中，`new` 和 `make` 是两个不同的内置函数，用于不同的目的，主要用于创建不同类型的数据结构。

`new`：

`new` 函数用于创建一个指向某个类型零值的指针。它接受一个参数，该参数是一个类型（不是值），并返回一个指向该类型零值的指针。`new` 创建的是一个值的指针，而不是一个初始化后的值。主要用于基本数据类型、结构体和数组等类型的初始化。

```
1  var x *int
2  x = new(int) // 创建一个 int 类型的零值的指针
```

`make`：

`make` 函数用于创建切片、映射和通道，返回一个已初始化的、可用于存储数据的值。`make` 接受一个类型参数，然后是一些用于创建指定类型的数据结构所需的参数。主要用于创建切片、映射和通道，因为它们需要底层的数据结构和内部状态。

```
1  slice := make([]int, 0, 10) // 创建一个切片，长度为 0，容量为 10
```

总结区别：

1. `new` 函数返回一个指向新分配的类型零值的指针，主要用于基本数据类型和结构体等的初始化。
2. `make` 函数返回已初始化的数据结构（切片、映射、通道），主要用于创建这些类型的数据结构。
3. `new` 创建的是值的指针，而 `make` 创建的是已初始化的值。
4. `new` 没有额外的参数，只需要传入类型。`make` 需要根据具体类型的需求传入相应的参数。

总之，`new` 和 `make` 在使用上有明显的区别，根据需要选择正确的函数来创建适当类型的数据结构。

目前学习到第135集。

异常处理

2023.08.26

简单处理

在 Go 语言中，异常处理的方式与许多其他编程语言不同。Go 语言鼓励使用返回值来处理错误情况，而不是使用异常。因此，Go 语言没有像其他语言那样的“try-catch”块来捕获异常。

错误处理的基本方法是通过函数返回值来传递错误信息。函数在遇到错误时，通常会返回一个非空的 `error` 值，如果执行成功，则返回 `nil`。调用者在调用函数后，可以检查返回的 `error` 值是否为 `nil` 来判断是否发生了错误。

以下是一个示例，展示了如何处理错误而不是使用异常：

```go
package main

import (
    "fmt"
    "errors"
)

func divide(a, b float64) (float64, error) {
    if b == 0 {
        return 0, errors.New("division by zero")
    }
    return a / b, nil
}

func main() {
    result, err := divide(10, 2)
    if err != nil {
        fmt.Println("Error:", err)
    } else {
        fmt.Println("Result:", result)
    }

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

如果你仍然想要实现类似于“try-catch”的错误处理，可以结合使用 `defer` 和 `recover`，但请注意，这不是 Go 语言中常见的错误处理方式，而且只能在处理 panic（Go 的异常机制）时使用。

```go
package main

import "fmt"

func main() {
    //test01()   // 报错后，程序中断，后面的代码不再执行
    test02() // 捕获异常之后，代码出错会提示，但不会中断程序，下面的代码继续执行
    fmt.Println("test01()下面的代码继续执行...")
}
func test01() {
    num1 := 10
    num2 := 0
    res := num1 / num2 // 报错：panic: runtime error: integer divide by zero
    fmt.Println("res=", res)
}

func test02() {
    // 使用defer + recover 来捕获异常
    defer func() {
        err := recover()
        if err != nil {
            fmt.Println("err:", err) // err: runtime error: integer divide by zero
        }
    }()

    num1 := 10
    num2 := 0
    res := num1 / num2 // 报错：panic: runtime error: integer divide by zero
    fmt.Println("res=", res)
}

```

上面的代码中，定义了一个匿名函数，加上了defer关键字来修饰，当本函数发生异常或结束时，自动调用，以此达到捕获异常的作用。

在匿名函数中，使用了recover()函数，来捕获当前协程中的panic。

然后检查返回的 `err` 值是否为 `nil` 来判断是否发生了错误。

> `recover` 是一个内置函数，用于捕获并恢复在当前协程中的 panic。`recover` 只能在 `defer` 函数中使用，用于在程序发生 panic 时进行恢复操作，以避免整个程序崩溃。
>
> 使用 `recover` 可以捕获 panic 的值，并进行相应的处理，比如记录日志、输出错误信息等。

手动抛出指定错误信息

Go程序中，也支持自定义错误， 使用 errors.New 和 panic 内置函数。

1. errors.New("错误说明")：返回一个 error 类型的值，表示一个错误。

   > 这里的错误说明是可以自定义的。

2. panic()内置函数：接收一个 interface{} 类型的值(也就是任何值了)作为参数。可以接收 error 类型的变量，输出错误信息，并退出程序。

```go
package main

import (
    "errors"
    "fmt"
)

func main() {
    testMyErr()
    fmt.Println("main()中代码继续执行...")
}

func readConf(name string) (err error) {
    if name != "config.ini" {
        return errors.New("读取配置文件出错！") // 当文件名不匹配时，返回一个 error 类型的值
    }
    return nil
}

func testMyErr() {
    err := readConf("config.ini")
    if err != nil {
        panic(err) // 使用panic()函数，手动抛出一个panic，会被系统捕获并中止程序
    } else {
        fmt.Println("配置文件读取成功！")
    }
    fmt.Println("testMyErr()继续执行...")
}
```

自定义错误类型

在 Go 语言中，你可以通过实现 `error` 接口来定义自己的错误类型。`error` 接口只包含一个方法 `Error()`，该方法返回一个描述错误的字符串。通过实现这个接口，你可以创建自定义的错误类型，以便在程序中使用。

以下是一个示例，展示了如何定义和使用自定义错误：

```go
package main

import (
    "fmt"
)

// 自定义错误类型
type MyError struct {
    Code    int
    Message string
```

```go
}

// 实现 error 接口的 Error 方法
func (e MyError) Error() string {
    return fmt.Sprintf("Error: %d - %s", e.Code, e.Message)
}

// 函数，可能返回自定义错误
func doSomething() error {
    return MyError{Code: 404, Message: "Page not found"}
}

func main() {
    err := doSomething()
    if err != nil {
        fmt.Println(err)
    }
}
```

在上述示例中，我们定义了一个自定义错误类型 `MyError`，实现了 `error` 接口的 `Error` 方法。然后，我们在 `doSomething` 函数中返回了一个 `MyError` 类型的错误。在 `main` 函数中，我们调用 `doSomething` 并检查返回的错误。

自定义错误类型的好处在于，你可以根据你的程序逻辑和需求，为不同的错误情况创建适当的错误类型，从而使错误处理更加明确和有意义。

需要注意的是，Go 语言中的错误处理是一种约定，通常函数在发生错误时会返回一个非空的 `error` 值，如果成功则返回 `nil`。因此，自定义错误类型应该按照这种约定来实现，以便在整个代码库中保持一致的错误处理方式。

数组与切片

数组的定义

在Go语言中，数组（Array）是一种固定长度且具有相同数据类型的数据结构。数组的长度在创建时确定，之后无法更改。每个数组的元素可以通过索引来访问，索引从0开始，逐个递增。以下是关于Go语言中数组的一些重要信息：

1. **数组的声明和初始化**：

在Go中，可以使用以下语法声明和初始化数组：

```go
var arr [5]int          // 声明一个包含5个整数的数组，初始值都为0
arr := [3]string{"a", "b", "c"}  // 声明并初始化一个包含3个字符串的数组

// 几种初始化数组的方式
// 方式1
var arr01 [5]int = [5]int{1, 2, 3, 4, 5}
// 前面的类型可以省略：var arr01 = [5]int{1, 2, 3, 4, 5}
fmt.Println("arr01：", arr01)

// 方式2：类型推断
```

```go
// 也可以写成：arr02 := [5]int{1, 2, 3, 4, 5}
arr02 := [5]int{1, 2, 3, 4, 5}
fmt.Println("arr02：", arr02)

// 方式3：[...]，动态获取初始化数组的长度
// 这里的[...]是固定写法，不能修改
arr03 := [...]int{1, 2, 3}   // 这样可以得到长度为3的数组，并给其赋值
fmt.Println("arr03：", arr03) // arr03：  [1 2 3]

// [...]的方式也可以，根据索引来赋值
arr04 := [...]int{1: 2, 0: 1, 2: 3}
fmt.Println("arr04：", arr04) // arr04：  [1 2 3]
```

2. **访问数组元素**：

通过索引访问数组元素：

```go
value := arr[2]  // 访问索引为2的元素
```

3. **数组长度**：

数组的长度是固定的，通过`len()`函数获取：

```go
length := len(arr)  // 获取数组arr的长度
```

4. **遍历数组**：

可以使用循环遍历数组中的元素：

```go
for i := 0; i < len(arr); i++ {
    fmt.Println(arr[i])
}

// 或者使用range关键字
for index, value := range arr {
    fmt.Printf("Index: %d, Value: %v\n", index, value)
}
```

5. **多维数组**：

Go语言支持多维数组，例如二维数组：

```go
var matrix [3][3]int  // 声明一个3x3的二维整数数组

// 初始化二维数组
matrix = [3][3]int{
    {1, 2, 3},
    {4, 5, 6},
    {7, 8, 9},
}
```

6. **数组是值类型**：

在Go中，数组是值类型，当数组赋值给另一个数组时，会发生一次完整的拷贝。这与切片（Slice）不同，切片是引用类型。

7. **注意事项**：

- 数组一旦声明，其长度不能更改。
- 长度是数组类型的一部分，在传递函数参数时，需要考虑数组的长度。
- 数组在函数间传递时会发生复制，可能引发性能问题。在需要在函数间传递大量数据时，通常使用切片。

总之，数组是Go语言中的一种基本数据结构，用于存储固定长度、相同类型的数据。如果你需要动态长度的容器，更常用的选择是切片（Slice）。

遍历数组

在Go语言中，有2种方式可以遍历数组：

1. **使用普通的for循环**：

```go
arr := [5]int{1, 2, 3, 4, 5}

for i := 0; i < len(arr); i++ {
    fmt.Println(arr[i])
}
```

2. **使用range关键字**：

```go
arr := [5]int{1, 2, 3, 4, 5}

for index, value := range arr {
    fmt.Printf("Index: %d, Value: %d\n", index, value)
}
```

`range`关键字用于迭代数组，每次迭代会返回当前元素的索引和值。如果你不需要使用索引或值，可以使用下划线`_`来忽略其中的一个。

数组的底层存储

在Go语言中，数组的底层存储是连续的内存块，其中每个元素占据固定大小的内存空间，这使得数组的访问和操作非常高效。

具体来说，Go语言的数组在内存中是按照顺序存储的，数组的第一个元素被存储在内存的起始位置，后续的元素依次存储在前一个元素之后的位置。这种连续存储的方式使得通过索引可以直接计算出元素在内存中的地址，从而实现了O(1)的时间复杂度来访问数组元素。

> 数组元素的地址，是整个数组元素的首地址，后面元素的地址根据首地址和偏移量来获得。
>
> 数组的各个元素的地址间隔（字节数）是依据数组的类型决定，比如 int64--->8 intt32--->4 ....。
>
> 如：

```go
var arr [5]int32
for i := 0; i < len(arr); i++ {
 fmt.Printf("arr[%d]的地址：%v\n", i, &arr[i])
 // arr[0]的地址：0xc00000e330，每次增加4个字节，因为每个元素占4个字节（32位）。
 // arr[1]的地址：0xc00000e334
 // arr[2]的地址：0xc00000e338
}
```

由于数组的长度在创建时就确定了，并且不可更改，所以内存分配是静态的，也就是说，一旦数组被创建，其所需的内存空间就会被分配好。这与切片（Slice）不同，切片是动态长度的，其底层数据结构包含指向连续内存块的指针、长度和容量字段。

另外，Go语言的数组在传递给函数时，会进行一次值传递。这意味着数组作为参数传递给函数时，会复制整个数组，而不仅仅是传递一个引用。这可能会导致一些性能开销，特别是在处理大型数组时。

如想在其它函数中，去修改原来的数组，可以使用引用传递（指针方式）。

```go
package main

import "fmt"

func main() {
 arr1 := [3]int{1, 2, 3}
 fmt.Println("arr1=", arr1) // arr1= [1 2 3]
 funcTest(&arr1)  // 传入数组的地址
 fmt.Println("arr1=", arr1) // arr1= [10 20 30]
}

func funcTest(arr *[3]int) {
 // 通过引用的方式传递，arr里面存放的是传递过来数组的地址
 // 通过arr加下标的方式可以访问到具体地址的值
 // 这样修改之后，原来数组的值也被修改了
 (*arr)[0] = 10 // (*arr)[0] 和 arr[0] 等价
 arr[1] = 20
 arr[2] = 30
}
```

总结起来，**Go语言中的数组底层存储是连续的内存块，元素按顺序存储，通过索引可以高效地访问。数组的长度在创建时确定，内存分配是静态的。数组在传递给函数时会进行值复制。**如果需要动态长度的容器，可以考虑使用切片。

目前学习到第148集。

二维数组

基本使用

```go
package main

import "fmt"

```

```go
func main() {
    var arr01 [3][4]int // 定义二维数组
    // 二维数组：相当于外层数组里面存的元素又是数组（两层数组）

    // 定义时赋值
    arr02 :=[2][3]{{1,2,3},{4,5,6}}

    arr01[1] = [4]int{1, 1, 1, 1} // arr01[1]的数据类型为[4]int
    fmt.Println("arr01:", arr01)
    fmt.Println(len(arr01)) // 二维数组的长度为外层数组的长度，这里为3

    for i := 0; i < len(arr01); i++ { // 输出外层数组
       fmt.Println(arr01[i])
    }

    for i := 0; i < len(arr01); i++ { // 输出二维数组中的每一个元素
       for j := 0; j < len(arr01[0]); j++ {
          fmt.Printf("%d ", arr01[i][j])
       }
       fmt.Println()
    }
}
```

底层存储

在 Go 语言中，二维数组的底层存储方式是连续的，每一行的整数元素依次存储在连续的内存地址中，而行与行之间也是连续存储的。这使得访问二维数组中的元素非常高效，因为计算元素的地址只需要一次内存寻址操作。

如图：


```go
var arr01 [3][4]int32 // 定义二维数组

// 二维数组的地址
fmt.Printf("arr01[0]的地址：%p\n", &arr01[0]) // 0xc00000e330
fmt.Printf("arr01[1]的地址：%p\n", &arr01[1]) // 0xc00000e340
// 0xc00000e330-0xc00000e340=0x10，转换为10进制就是16
// 由于内层数组的有4个32位（4 byte）的整数，所以arr01[0]和arr01[1]之间相差16 byte
```

切片

2023.08.28

在 Go 语言中，切片（Slice）是一种动态数组，它提供了对数组的部分或全部元素的引用。切片相对于数组更加灵活，因为切片的长度是可变的。切片是基于数组的抽象，它的底层实现是一个指向数组的指针，以及切片的长度和容量信息。

以下是一些关于 Go 语言中切片的基本概念：

1. **创建切片：** 可以使用以下方式来创建一个切片：

```go
var mySlice []int               // 声明一个切片
mySlice := make([]int, 0, 10)   // 使用 make() 函数创建一个切片，长度为 0，容量为 10
mySlice := []int{1, 2, 3, 4}    // 使用初始值创建切片
```

2. **切片表达式：** 切片可以从数组、切片或字符串中通过切片表达式创建。切片表达式的格式为 `[low:high]`，其中 low 是切片的起始索引（包含），high 是切片的结束索引（不包含）。

```go
myArray := [5]int{1, 2, 3, 4, 5}
mySlice := myArray[1:3] // 创建一个包含 myArray[1] 和 myArray[2] 的切片
```

> `var slice = arr[0:end]` 可以简写 `var slice = arr[:end]`
> `var slice = arr[start:len(arr)]` 可以简写: `var slice = arr[start;]`
> `var slice = arr[0:len(arr)]` 可以简写: `var slice = arr[:]`

3. **切片长度和容量：** 切片的长度是指切片中的元素个数，容量是指底层数组中从切片的起始索引到数组末尾的元素个数。可以使用 `len()` 函数获取切片长度，使用 `cap()` 函数获取切片容量。
4. **切片追加元素：** 使用内置的 `append()` 函数可以向切片追加一个或多个元素。如果切片容量不足以存储新元素，`append()` 函数会自动分配更大的底层数组，并复制原有数据到新数组中。

```go
mySlice := []int{1, 2, 3}
mySlice = append(mySlice, 4, 5) // 追加元素
```

> 切片append操作的底层原理分析:
>
> 1. 切片append操作的本质就是对数组扩容。
> 2. go底层会创建一下新的数组newArr(安装扩容后大小)。
> 3. 将slice原来包含的元素拷贝到新的数组new。
> 4. slice 重新引用到newArr。
> 5. 注意newArr是在底层来维护的，程序员不可见。

5. **切片复制：** 使用内置的 `copy()` 函数可以将一个切片的元素复制到另一个切片中。

```go
source := []int{1, 2, 3}
destination := make([]int, len(source))
copy(destination, source) // 复制 source 切片到 destination 切片
```

> 复制之后的切片和原来的切片是完全独立的，互不影响。

6. **切片作为函数参数：** 切片在函数之间传递时是按引用传递的，这意味着函数内部修改切片会影响到外部切片。

切片是 Go 语言中非常常用的数据结构，它的动态性和便利性使得它在处理变长数据集合时非常有用。在使用切片时，需要注意切片的底层数据共享，以及切片的长度和容量等特性。

字符串切片

```go
package main

import (
    "fmt"
    "reflect"
    "strings"
)

func main() {

    // 对string进行切片
    s1 := "hello,world!"
    slice01 := s1[0:5] // 这样得到的slice01还是string类型，而不是切片类型
    //必须使用[]byte()函数来对截取得到的字符串进行显式转换
    slice02 := []byte(s1[0:5])
    fmt.Println("slice01的类型:", reflect.TypeOf(slice01)) // string
    fmt.Println("slice02的类型:", reflect.TypeOf(slice02)) // []uint8

    // string是不可修改的
    //s1[0] = 'a'  // 报错: cannot assign to s1[0]

    // 如果需要修改字符串，可以先将string -> []byte /或者 rune -> 修改 -> 重写转成string。
    s2 := "hello,world!"
    slice03 := []byte(s2) // 这里需要将s2转换成切片类型
    // 注意，使用s2[:]得到的还是string类型，必须通过显示转换才可以
    slice03[0] = 'a'
    s2 = string(slice03)   // 再将切片类型转换回string类型
    fmt.Println("s2:", s2) // s2: aello,world!

    // 将含有存有多个字符串的切片按指定分隔符连接
    // 使用strings.Join()函数
    slice04 := []string{"hello", "world"}
    s3 := strings.Join(slice04, " ")
    fmt.Println("s3=", s3) // s3= hello world
}
```

排序和查找

排序

1.冒泡排序法 (Bubble sort2)

```go
// 冒泡排序：每轮使1个数在正确的位置上，需要双层循环
// 外层循环：控制循环的轮数，每轮排好1个数
// 内层循环：从外层循环的i值开始，表示第i个元素是即将需要参与排序的数
func bubbleSort(arr *[10]int) {
    for i := 0; i < len(arr)-1; i++ { // 控制循环的轮数
        for j := i; j > 0; j-- { // // 控制实际参与排序的数字
            if arr[j] < arr[j-1] {
                temp := arr[j]
                arr[j] = arr[j-1]
                arr[j-1] = temp
            }
        }
    }
}
```

2.快排序法(Quick sort)

目前还没学，后面学习数据结构与算法的时候再补上。

下面是Chatgpt给出的示例：

```go
package main

import "fmt"

func quickSort(arr []int) []int {
    if len(arr) <= 1 {
        return arr
    }

    pivot := arr[len(arr)-1]
    var left []int
    var right []int

    for _, num := range arr[:len(arr)-1] {
        if num <= pivot {
            left = append(left, num)
        } else {
            right = append(right, num)
        }
    }

    sortedLeft := quickSort(left)
    sortedRight := quickSort(right)

    return append(append(sortedLeft, pivot), sortedRight...)
}
```

```
27
28 func main() {
29     arr := []int{4, 2, 8, 5, 1, 9}
30     sortedArr := quickSort(arr)
31     fmt.Println(sortedArr) // 输出：[1 2 4 5 8 9]
32 }
```

在这个示例中，`quickSort` 函数接收一个整数切片并返回一个经过快速排序处理的新切片。它首先检查切片的长度，如果长度小于等于 1，则直接返回原始切片。否则，它选择最后一个元素作为“枢纽”（pivot），将比 pivot 小的元素放在 `left` 切片中，比 pivot 大的元素放在 `right` 切片中。

然后，递归地对 `left` 和 `right` 切片调用 `quickSort` 函数，最后将排序后的 `left`、pivot 和 `right` 拼接在一起返回。

请注意，这只是一个简单的示例，用于说明快速排序的基本思想。在实际应用中，可能需要考虑性能、边界情况以及更复杂的数据类型。通常情况下，使用标准库中的排序函数会更高效和可靠。

查找

Go 语言中有多种用于查找元素或数据的内置方法和数据结构。下面我将介绍一些常用的查找方法和数据结构。

1. **线性搜索：** 这是最简单的查找方法，从列表的开头开始逐个比较元素，直到找到目标元素或搜索完整个列表。
2. **二分搜索：** 适用于已排序的数据。它将目标元素与中间元素进行比较，然后根据比较结果决定在左半部分或右半部分继续搜索。

```
1  package main
2
3  import "fmt"
4
5  func main() {
6      arr := [...]int{1, 2, 3, 4, 5, 6, 7, 8}
7      binarySearch(&arr, 0, len(arr)-1, 8)
8  }
9  func binarySearch(arr *[8]int, leftIndex int, rightIndex int, findElem
   int) {
10
11     // 当leftIndex大于rightIndex时，表示未找到指定元素
12     if leftIndex > rightIndex {
13        fmt.Println("找不到")
14        return
15     }
16     middleIndex := (leftIndex + rightIndex) / 2
17     middleElem := arr[middleIndex]
18
19     if findElem < middleElem {
20        binarySearch(arr, leftIndex, middleIndex-1, findElem)
21     } else if findElem > middleElem {
22        binarySearch(arr, middleIndex+1, rightIndex, findElem)
23     } else {
```

```
        fmt.Println("找到了，下标为：", middleIndex)
    }
}
```

3. **map（映射）：** Go 的 map 是一种键值对的集合，你可以使用键来查找对应的值。map 的查找操作是常数时间复杂度，即使在大型数据集中也非常高效。

```
// 创建一个 map
m := map[string]int{"a": 1, "b": 2, "c": 3}
value, found := m["b"]
if found {
    fmt.Println("Found:", value) // 输出：Found: 2
}
```

4. **切片操作：** 使用切片的下标可以直接访问特定位置的元素。但是，这只适用于已知索引的情况。

```
slice := []int{1, 2, 3, 4, 5}
element := slice[2] // 获取索引为2的元素，即3
```

5. **排序和搜索算法库：** Go 标准库中提供了排序和搜索算法的实现，你可以使用这些库来执行高效的查找操作。例如，使用 `sort` 包进行排序，然后使用 `sort.Search` 函数进行二分搜索。
6. **自定义数据结构：** 根据需求，你可以构建自定义的数据结构来支持高效的查找。例如，你可以使用二叉搜索树（BST）来实现一种高效的查找机制。

总之，Go 语言提供了多种查找数据的方法，你可以根据具体的需求选择合适的方法。选择适当的数据结构和算法对于获得高性能和可维护的代码非常重要。

目前学习到第167集。