

A decorative graphic in the top-left corner consisting of a grid of small squares in red, orange, and yellow, arranged in a pattern that tapers to the right.

Онлайн образование

otus.ru



Проверить, идет ли запись

Меня хорошо видно
&& слышно?



Тема вебинара

Работа с вводом/выводом



Романовский Алексей

Преподаватель OTUS

Слак Otus: ищите по имени

Телеграм: alexus1024

GitHub: <https://github.com/alexus1024>

Преподаватель



Романовский Алексей

Сейчас - разработчик Golang

Более 5 последних лет - бекенд на Go
Ранее, 10 лет - фуллстек - MS .Net, C#

Часовой пояс: -4 GMT

Правила вебинара



Активно
участвуем



Off-topic обсуждаем
в Телеграме



Задаем вопрос
в чат или поднимаем руку



Вопросы вижу в чате,
могу ответить не сразу

Условные обозначения



Индивидуально



Время, необходимое
на активность



Пишем в чат



Говорим голосом



Документ



Ответьте себе или
задайте вопрос

Цели вебинара

После занятия вы сможете

1.	Использовать библиотеки ввода-вывода
2.	Не пугаться, если библиотека просит какой то Reader вместо string или []byte
3.	Форматировать строки
4.	Радоваться



Смысл

Зачем вам это уметь

1. Связать ваши решения с внешним миром
2. Свободно перемещать данные внутри программы и между ними

Маршрут вебинара

Введение в ввод-вывод в Go

Стандартные интерфейсы: Reader, Writer, Closer

Блочные устройства, Seeker

Буферизация ввода/вывода

Форматированный ввод и вывод: fmt

Работа с командной строкой

Рефлексия

Библиотеки, пакеты

- `io` - базовые функции и интерфейсы
- `bufio` - буферизованный ввод / вывод
- `fmt` - форматированный ввод / вывод
- `os` (точнее `os.Open` и `os.File`) - открытие файла

Также, для работы с файловой системой будут полезны:

- `path` и `path/filepath` - для работы с путями к файлам

`ioutil` - устарел в 1.16. -> `io`, `os`

Проблема унификации

Устройств ввода-вывода - много

Хотим писать код, не зависящий от конкретного устройства

Путь: **Dependency Injection (DI)**

Внезапно!

Вспоминаем: что такое

Dependency Injection?



Ответы пишем в чат, у нас на это 3 минуты



Абстракция устройства ввода-вывода

Устройств ввода-вывода - много. У каждого свои особенности.
У каждого устройства есть свой пакет с **имплементацией**:

память, файл, сеть, пользователь + любые ваши

Чтобы работать одинаково - го предоставляет **интерфейсы**;

Каждый **интерфейс** представляет одну “фичу” устройства;

Каждая **имплементация** реализует набор фич.



Пакет io

Содержит **интерфейсы** и инструменты для работы с ними.

Не привязан ни к каким **имплементациям**
(таким как os, файлы, сокеты и т.п.).

Обзор интерфейсов

Пакет io. Один интерфейс - одна фишка.

Основные,
чтение:

Основные,
запись:

Комментарий

`Reader`

`Writer`

массив байт

`ByteReader`

`ByteWriter`

байты по 1

`ReaderAt`

`WriterAt`

пропуски

`ReaderFrom`

`WriterTo`

в другой и-фейс

`RuneReader`

`StringWriter`

`ByteScanner`

`RuneScanner`

Составные:

- `ReadCloser`
- `ReadSeekCloser`
- `ReadSeeker`
- `ReadWriteCloser`
- `ReadWriteSeeker`
- `ReadWriter`
- `WriteCloser`
- `WriteSeeker`

Другое:

- **`Closer`**
- **`Seeker`**

Как это работает

```
func main() {  
    inFile, err := os.Open("filename.in"); check(err)  
    outFile, err := os.Create("filename.io"); check(err)  
    process(inFile, outFile)           // (1)  
  
    inMem := bytes.NewReader([]byte("some content"))  
    outMem := bytes.NewBuffer(nil)  
    process(inMem, outMem)             // (2)  
  
    process(os.Stdin, os.Stdout)       // (3)  
  
    process(inFile, outMem)           // (4)  
}
```

```
func process(  
    in io.Reader,  
    out io.Writer,  
) {  
    data := make([]byte, 1024)  
    _, err := in.Read(data)  
    check(err)  
  
    callLogic(data)  
  
    _, err = out.Write(data)  
    check(err)  
}
```

io.Reader

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

io.Reader - это нечто, ИЗ чего можно последовательно читать байты.

- Метод `Read` читает данные (из объекта) в буфер `p`,
не более чем `len(p)` байт.
- Метод `Read` возвращает количество байт `n`, которые были прочитаны и записаны в `p`,
причем `n` может быть меньше `len(p)`.
- Метод `Read` возвращает ошибку или `io.EOF` в случае конца файла,
при этом он так же может вернуть `n > 0`, если часть данных были прочитаны до ошибки.

`io.EOF` - специальная ошибка, означающая что мы достигли конца потока (файла)

io.Writer

```
type Writer interface {  
    Write(p []byte) (n int, err error)  
}
```

io.Writer - это нечто, ВО что можно последовательно записать байты.

- Метод Write записывает len(p) байт из p в объект (например файл или сокет).
- Метод Write реализует цикл до-записи внутри себя.
- Метод Write возвращает количество записанных байт n и ошибку, если n < len(p)

io.Closer

```
type Closer interface {  
    Close() error  
}
```

io.Closer - представляет ресурс, который следует вручную освободить после использования

- Если библиотека передала вам объект, поддерживающий Closer - значит библиотека ожидает что вы сами вызовете Close;
- Если вы передаёте свой объект в функцию, принимающую Closer ожидайте что его там закроют;
- Close следует вызывать как можно раньше;
- Close часто вызывают в блоке `defer f.Close()` (забывая при этом обработать ошибку);
- Забытый Close - причина утечки ресурсов (например, навечно открытый файл)

Остальные интерфейсы пакета io рассмотрим позже

Работа с файлами

... и последовательный доступ

Открываем файлы

Для открытия файла на чтение используем `os.OpenFile`

```
var file *os.File // файловый дескриптор в Go
file, err := os.OpenFile(path, os.O_RDWR, 0644)
if err != nil {
    if os.IsNotExist(err) {
        // файл не найден ...
    }
    // другие ошибки, например нет прав ...
}
defer file.Close()
```

Интерфейсы `os.File`:

`Reader`
`ReaderAt`
`ReaderFrom`

`Writer`
`WriterAt`
`StringWriter`

`Seeker`
`Closer (!)`

Так же есть специальные "сокращения":

- `os.Create(name)` = `OpenFile(name, os.O_RDWR|os.O_CREATE|os.O_TRUNC, 0666)`
- `os.Open(name)` = `OpenFile(name, os.O_RDONLY, 0)`

Читаем файл

Читаем первые N байт в buf, используя `io.Reader`

```
N := 1024 // мы заранее знаем сколько хотим прочитать
buf := make([]byte, N) // подготавливаем буфер нужного размера
file, _ := os.Open(path) // открываем файл (не забыть про err!)
offset := 0
for offset < N {
    read, err := file.Read(buf[offset:])
    offset += read
    if err == io.EOF {
        // что если не дочитали ?
        break
    }
    if err != nil {
        log.Panicf("failed to read: %v", err)
    }
}
```

Удобства для чтения

Гарантированно заполнить буфер

```
b := make([]byte, 1024*1024)
file, _ := os.Open(path)
read, err := io.ReadFull(file, b) // содержит цикл внутри
```

Прочитать все до конца файла

```
file, _ := os.Open(path)
b, err := io.ReadAll(file) // err настоящая ошибка, не EOF
```

Или еще короче (для скриптов)

```
b, err := os.ReadFile(path) // прочитать весь файл по имени
```

операции в пакете `io` применимы к любым `io.Reader`, а не только файлам

Пишем в файлы

Пишем содержимое `b` в файл используя `io.Writer`

```
b := make([]byte, 1024*1024) // заполнен нулями
file, _ := os.Create(path)
written, err := file.Write(b)
if err != nil {
    log.Panicf("failed to write: %v", err)
}
// мы записали 1М данных !
file.Close() // чтобы очистить буферы ОС
```

В отличие от операции чтения, тут цикл не нужен.

Удобства для записи

Целиком переписать файл:

```
b := make([]byte, 1024*1024)

err := os.WriteFile(path, b, 0644)
```

Скопировать данные из любого `io.Reader`:

```
f, _ := os.Create("tmp")

w, _ := io.Copy(f, os.Stdin) // в f из Stdin, свой цикл внутри и буффер

fmt.Printf("Written %v bytes", w)
```

`Copy` остановится на `io.EOF`,

- есть ещё `CopyN` - он остановится через N байт;
- а еще - `CopyBuffer` - он не создаёт свой буфер, а использует заданный.

операции в пакете `io` применимы к любым `io.Writer`, а не только файлам

Произвольный доступ

... и работа с файлами

Произвольный доступ

Устройства/технологии ввода/вывода данных можно условно разделить на

- поддерживающие произвольный доступ
жесткие диски память
- и поддерживающие последовательный доступ
терминал сетевое соединение pipe

Рассмотренные `io.Reader`, `io.Writer` - для последовательного доступа

а вот `io.ReaderAt`, `io.WriterAt`, `io.Seeker` - для произвольного доступа

io.Seeker

```
type Seeker interface {  
    Seek(offset int64, whence int) (int64, error)  
}
```

Позволяет передвинуть текущую "позицию" в файле вперед или назад на offset байт.

Аналог в линуксе - `man lseek`

Возможные значения whence:

- `io.SeekStart = 0` относительно начала файла;
- `io.SeekCurrent = 1` относительно текущего положения в файле;
- `io.SeekEnd = 2` относительно конца файла.

Тип `os.File` реализует интерфейс `io.Seeker`, а вот типа `net.TCPConn` - нет.

```
f.Seek(0, io.SeekStart) // начало файла  
f.Seek(1, io.SeekCurrent) // пропустить следующий байт  
f.Seek(-1, io.SeekEnd) // последний байт
```

io.ReaderAt и io.WriterAt

```
type ReaderAt interface {  
    ReadAt(p []byte, off int64) (n int, err error)  
}  
  
type WriterAt interface {  
    WriteAt(p []byte, off int64) (n int, err error)  
}
```

Позволяют прочитать/записать `len(p)` байт с указанным `off` смещением в файле, т.е. с произвольной позиции.

В отличие от `io.Reader`, реализации `io.ReaderAt` всегда читают ровно `len(p)` байт или возвращают ошибку.

Другие инструменты

io.WriterTo и io.ReaderFrom

```
type ReaderFrom interface {  
    ReadFrom(r Reader) (n int64, err error)  
}  
  
type WriterTo interface {  
    WriteTo(w Writer) (n int64, err error)  
}
```

При копировании с использованием `io.Reader` и `io.Writer` приходится выделять буфер в памяти, т.е. происходит двойное копирование данных.

Если же источник/получатель данных реализуют интерфейсы `io.WriterTo` / `io.ReaderFrom`, то копирование с помощью `io.Copy` может использовать оптимизацию и НЕ выделять промежуточный буфер.

Используются в `io.Copy`* автоматически

Например, в `linux` есть специальный системный вызов `sendfile` для быстрого копирования - и реализация этих интерфейсов позволяет использовать его вместо побайтового переноса данных.

Комбинированные интерфейсы

```
type ReadCloser interface {  
    Reader  
    Closer  
}  
  
func FooRead(src io.ReadCloser) {  
    src.Read(...)  
    src.Close()  
}
```

Если в вашем коде требуется объект с несколькими фичами -
вы можете использовать комбинированные интерфейсы.
Все практичные комбинации уже есть в пакете `io`.

Объединение потоков, chaining

io.MultiReader - позволяет последовательно читать из нескольких reader-ов.

По смыслу аналогично `cat file1 file2 file3`

```
func MultiReader(readers ...Reader) Reader
```

io.MultiWriter - позволяет записывать в несколько writer-ов.

Аналогично `tee file1 file2 file3`

```
func MultiWriter(writers ...Writer) Writer
```

io.LimitReader - создаёт новый ридер, который прочитает не более n байт, далее вернёт io.EOF

```
func LimitReader(r Reader, n int64) Reader
```


IO в память

bytes.Buffer - позволяет читать и писать в память в последовательном стиле

```
buf := bytes.NewBuffer(nil)
buf.Write([]byte{55,56,57})
buf.Read(...)
contents := buf.String()
```

bytes.Reader - позволяет только чтение, зато произвольный доступ

```
reader := bytes.NewReader([]byte("my data"))
reader.Seek(-1, io.SeekEnd)
reader.Read(...)
```

IO в память

Пакет `strings` также предлагает `Reader`, более адаптированный для строк
А `strings.Builder` реализует `io.Writer` и другие интерфейсы.

```
sr := strings.NewReader("my str")  
  
b := strings.Builder{}
```

Если какая то библиотека принимает интерфейсы пакета `io`,
а вам надо просто передать туда слайс или строку, используйте эти инструменты
Пример:

```
import "bytes"  
import "archive/zip"  
  
buf := bytes.NewBuffer([]byte{})  
zipper := zip.NewWriter(buf)  
_, err := zipper.Write(data)  
// в buf находится zip архив!
```

Буферизация

Пакет bufio

С помощью пакета bufio можно

- сократить число системных вызовов и
- улучшить производительность

в случае, если требуется читать/записывать данные небольшими кусками, например по строкам.

Буферизованная запись:

```
file, _ := os.Create(path)
bw := bufio.NewWriter(file)
written, err := bw.Write([]byte("some bytes"))
bw.WriteString("some string")
bw.WriteRune('±')
bw.WriteByte(42)
bw.Flush()    // очистить буфер, записать все в file
```

Пакет bufio

Чтение:

```
file, _ := os.Open(path)
br := bufio.NewReader(file)
line, err := br.ReadString(byte('\n'))
b, err := br.ReadByte()
br.UnreadByte() // иногда полезно при анализе строки
```

Форматированный ВВОД-ВЫВОД

Пакет fmt. Вывод

Пакет предоставляет возможности форматированного вывода.

Основные функции:

```
func Fprintf(w io.Writer, format string, a ...any) (n int, err error)
func Sprintf(format string, a ...any) string
func Printf(format string, a ...any) (n int, err error)
```

Пример:

```
m := map[string]int{"qwe": 1}
fmt.Printf("%s %x %#v", "string", 42, m)
```

<https://go.dev/play/p/zzPQla7yfVG>

Пакет fmt. Вывод. Основное

Общие:

`%v` представление по умолчанию для типа
 `%#v` вывести как Go код (удобно для структур)
`%T` вывести тип переменной
`%%` вывести символ %

Для целых:

`%b` base 2
`%d` base 10
`%o` base 8
`%x` base 16, with lowercase letters for af

Для строк:

`%s` the uninterpreted bytes of the string or slice
`%q` a doublequoted string safely escaped with Go syntax
`%x` base 16, lowercase, two characters per byte

Документация: <https://pkg.go.dev/fmt>

Пакет fmt. Вывод. Сложные типы

Для сложных типов (слайсы, словари, каналы) имеет смысл выводить

Адрес в памяти: %p

Представление по-умолчанию: %v

```
struct:           {field0 field1 ...}  
array, slice:     [elem0 elem1 ...]  
maps:            map[key1:value1 key2:value2 ...]  
pointer to above: &{}, &[], &map[]
```

Go представление: %#v

Попробуйте: <https://play.golang.org/p/Q2nl9ZnaF96>

Пакет fmt. Вывод. Пользовательские типы

Вы можете управлять строковым представлением (%s) вашего типа, реализовав интерфейс fmt.Stringer

```
type Stringer interface {  
    String() string  
}
```

Также можно управлять расширенным представлением (%#v), реализовав GoStringer

```
type GoStringer interface {  
    GoString() string  
}
```

<https://go.dev/play/p/AvlsmhiXtXv>

Пакет fmt. Ввод

Также с помощью fmt можно считывать данные в заранее известном формате Основные функции:

```
func Scanf(format string, a ...interface{}) (n int, err error)
func Fscanf(r io.Reader, format string, a ...interface{}) (n int, err error)
```

Например:

```
var s string
var d int64

fmt.Scanf("%s %d", &s, &d)
```

В функцию Scanf передаются указатели, а не сами переменные.

Scanf возвращает количество аргументов, которые удалось сканировать и ошибку, если удалось меньше ожидаемого.

Командная строка



Командная строка

Аргументы командной строки - просто слайс строк. В Go он доступен как `os.Args`
Например при вызове

```
$ myprog in=123 out 456 qwe
```

В слайсе `os.Args` будет:

```
["myprog", "in=123", "out", "456", "qwe"]
```

В `os.Args[0]` - всегда имя исполняемого файла

Командная строка

Для упрощения работы с командной строкой можно использовать пакет `flag`

```
import "flag"

var input string
var offset int

func main() {
    flag.StringVar(&input, "input", "", "file to read from")
    flag.IntVar(&offset, "offset", 0, "offset in input file")

    flag.Parse() // проанализировать аргументы

    // теперь в input и offset есть значения
}
```

Регулярные выражения

Regex

<https://pkg.go.dev/regexp>

<https://pkg.go.dev/regexp/syntax>

Использовать для гибкого сравнения строки с паттерном
и поиска значимых полей в строке

```
func main() {  
    re := regexp.MustCompile(`(gopher){2}`)  
    fmt.Println(re.MatchString("gopher"))  
    fmt.Println(re.MatchString("gophergopher"))  
    fmt.Println(re.MatchString("gophergophergopher"))  
}
```

```
false
```

```
true
```

```
true
```


Regex

- `mustCompile`, время жизни инстанции типа `Regex`
- `match/find`
- `Find(All)?(String)?(Submatch)?(Index)?`
- `replace`
- Не подходит для работы с рекурсивными данными (XML, JSON)

Regex (submatch)

Может не только искать строки, но и выделять в найденном заданные поля

https://go.dev/play/p/UEvbl2Yl_zY

```
func main() {  
    re := regexp.MustCompile(`a(x*)b`)  
    fmt.Printf("%q\n", re.FindAllStringSubmatch("-ab-", -1))  
    fmt.Printf("%q\n", re.FindAllStringSubmatch("-axxb-", -1))  
    fmt.Printf("%q\n", re.FindAllStringSubmatch("-ab-axb-", -1))  
    fmt.Printf("%q\n", re.FindAllStringSubmatch("-axxb-ab-", -1))  
}
```

```
[["ab" ""]]
```

```
[["axxb" "xx"]]
```

```
[["ab" ""] ["axb" "x"]]
```

```
[["axxb" "xx"] ["ab" ""]]
```

Вот и всё!

Теперь порефлекслируем

... и ещё вопросы готовьте пока

Big picture

- интерфейсы IO как “язык” описания устройств ввода-вывода
- утилиты для работы с этими интерфейсами (ReadAll, Copy, Chaining)
- пример на файлах
- последовательный и произвольный доступ
- IO в память
- буферизация
- форматирование (Printf, Scanf, F*, S*)
- командная строка
- Regex

**Заполните, пожалуйста,
опрос о занятии
по ссылке в чате**

Вопросы?



Ставим "+",
если вопросы есть



Ставим "-",
если вопросов нет



Домашнее задание

ДЗ №7 «Утилита для копирования файлов»



Следующий вебинар



15 июня 2023

Форматирование



Ссылка на вебинар
будет в ЛК за 15 минут



Материалы
к занятию в ЛК —
можно изучать



Обязательный материал
обозначен красной
лентой

