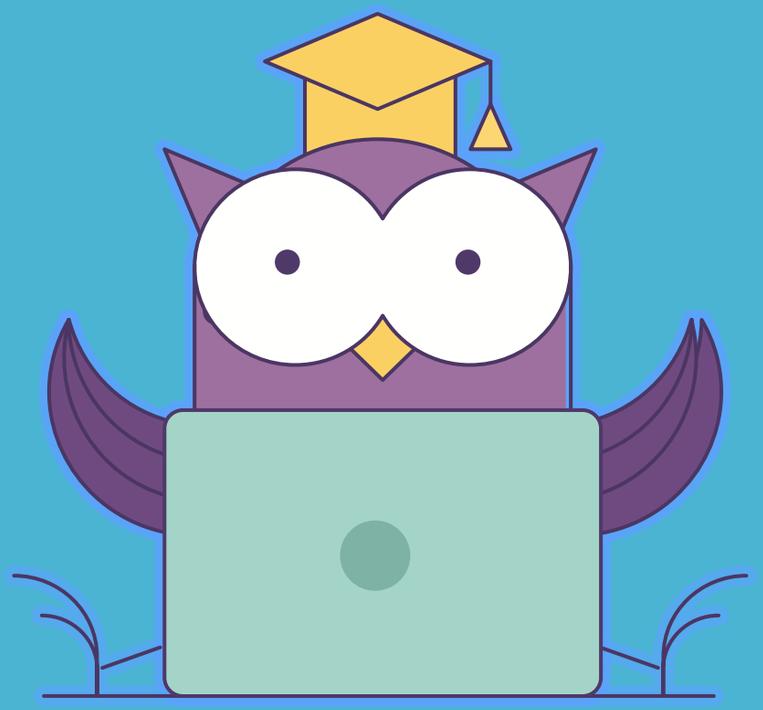
ОНЛАЙН-ОБРАЗОВАНИЕ

Работа с вводом/выводом

Дмитрий Смаль

Как меня слышно и видно?



> Напишите в чат

+ если все хорошо

- если есть проблемы со звуком или с видео

!проверить запись!

Небольшой тест

Пожалуйста, пройдите небольшой тест.

Возможно вы уже многое знаете про ввод/вывод в Go =)

https://forms.gle/1GzKZB1fhPmNdVDQ7

План занятия

- Стандартные интерфейсы: Reader, Writer, Closer
- Блочные устройства, Seeker
- Буферизация ввода/вывода
- Форматированный ввод и вывод: fmt
- Работа с командной строкой

Для работы с вводом / выводом в Go используются пакеты:

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

Так же для работы с файловой системой будут полезны:

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

Как открыть файл ?

Для открытия файла на чтение используем `os.OpenFile`

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

Так же есть специальные "сокращения":

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

Как прочитать данные из файла ?

OTUS

Сколько мы хотим прочитать ?

```
N := 1024 // мы заранее знаем сколько хотим прочитать

buf := make([]byte, N)  // подготавливаем буфер нужного размера

file, _ := os.Open(path) // открываем файл

offset := 0

for offset < N {
  read, err := file.Read(buffer[offset:])
  offset += read
  if err == io.EOF {
    // что если не дочитали ?
    break
  }
  if err != nil {
    log.Panicf("failed to read: %v", err)
  }
}

// мы прочитали N байт в buf !
```

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

Интерфейс io.Reader

Заметим, что тип `os.File` реализует интерфейс `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.

Если нужно просто прочитать все

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 := ioutil.ReadAll(file) // err - настоящая ошибка, не EOF
```

Или еще короче (для скриптов)

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

Как записать данные в файл ?

Сколько мы хотим записать ?

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

file, _ := os.Create(path)

wrote, err := file.Write(b[offset:])
if err != nil {
  log.Panicf("failed to write: %v", err)
}

// мы записали 1M данных !

file.Close() // что бы очистить буферы ОС
```

В отличие от операции чтения тут цикл не нужен.

Интерфейс io.Writer

Тип `os.File` реализует интерфейс `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)`

Если нужно просто записать все

Целиком перезаписать файл

```
b := make([]byte, 1024*1024)
err := ioutil.WriteFile(path, b, 0644)
```

Последовательный и 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

Как следствие есть два набора интерфейсов
- `io.Reader`, `io.Writer` - для последовательного доступа
- `io.ReaderAt`, `io.WriterAt`, `io.Seeker` - для произвольного доступа

Интерфейс io.Seeker

Интерфейс `io.Seeker` позволяет передвинуть текущую "позицию" в файле вперед или назад на `offset` байт (см man lseek)

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

возможные значения `whence`

- `io.SeekStart` - относительно начала файла, например `file.Seek(0, 0)` - установить позицию в начало файла.
- `io.SeekCurrent` - относительно текущего положения в файле.
- `io.SeekEnd` - относительно конца файла

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

Интерфейсы 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.

Используя методы `Read`, `Write` и промежуточный буфер не сложно сделать копирование между двумя файловыми (и не только).

А можно использовать и готовые реализации:

```
// копирует все вплоть до io.EOF
written, err := io.Copy(dst, src)

// копирует N байт или до io.EOF
written, err := io.Copy(dst, src, 42)

// копирует все вплоть до io.EOF, но использует заданный буфер
buffer := make([]byte, 1024*1024)
written, err := io.CopyBuffer(dst, src, buf)
```

Здесь `dst` должен реализовывать интерфейс `io.Writer`, а `src` - `io.Reader`

Оптимизация копирования

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.

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

NOTE: В linux есть специальный системный вызов `sendfile` который позволяет эту оптимизацию.

Другие стандартные интерфейсы

OTUS

В пакете `io` имеются так же интерфейсы

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

type ByteReader interface {
  ReadByte() (byte, error)
}

type ByteScanner interface {
  ByteReader
  UnreadByte() error
}
```

Комбинации интерфейсов

OTUS

А так же интерфейсы-комбинации

```
type ReadWriteCloser interface {
  Reader
  Writer
  Closer
}

type ReadWriteSeeker interface {
  Reader
  Writer
  Seeker
}
```

Вспомогательные типы

`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` - позволяет читать не более n байт, далее возвращает - `io.EOF`

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

Буферизация ввода/вывода

OTUS

С помощью пакета `bufio` можно сократить число системных вызовов и улучшить производительность в случае если требуется читать/записывать данные небольшими кусками, например по строкам.

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
```

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
```

Интерфейсы `io.Reader` и `io.Writer` могут быть реализованы различными структурами в памяти.

```
strings.Reader  //  реализует io.Reader
strings.Builder //  реализует io.Writer
bytes.Reader // реализует io.Reader
bytes.Buffer  // реализует io.Reader, io.Writer, io.ByteReader, io.ByteWriter, io.ByteScanner
```

Например можно

```
import "bytes"
import "archive/zip"

var buf bytes.Buffer
zipper := zip.NewWriter(buf)
_, err := zipper.Write(data)

// в buf находится zip архив!
```

Форматированный вывод

Пакет `fmt` предоставляет возможности форматированного вывода.
Основные функции:

```
func Printf(format string, a ...interface{}) (n int, err error)

func Fprintf(w io.Writer, format string, a ...interface{}) (n int, err error)
```

Например:

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

В отличие от языка C в Go можно определить тип аргументов с помощью `reflect` поэтому строка формата используется только для указания правил форматирования.

Основные флаги

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 lower-case letters for a-f
```

Для строк:

```
%s    the uninterpreted bytes of the string or slice
%q    a double-quoted string safely escaped with Go syntax
%x    base 16, lower-case, two characters per byte
```

Форматирование сложных типов

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

Форматирование для пользовательских типов

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

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

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

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

Форматированный 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)
```

ВНИМАНИЕ: В функцию `Scanf` передаются указатели, а не сами переменные.
`Scanf` возвращает количество аргументов, которые удалось сканировать и ошибку, если удалось меньше ожидаемого.

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"]
```

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

Использование flag

```
import "flag"

var input string
var offset int

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

func main() {
  flag.Parse() // проанализировать аргументы
  // теперь в input и offset есть значения
}
```

Небольшой тест

Проверим что мы узнали за этот урок

https://forms.gle/1GzKZB1fhPmNdVDQ7

Домашнее задание

Реализовать утилиту копирования файлов (см man dd). Выводить в консоль прогресс копирования. Программа должна корректно обрабатывать ситуацию, когда offset или offset+limit за пределами source файла.
Пример использования:

```
# копирует 2К из source в dest, пропуская 1К данных
$  gocopy -from /path/to/source -to /path/to/dest -offset 1024 -limit 2048
```

Настроить и запустить линтеры, создать Makefile для автоматизации тестирования и сборки. Должна быть возможность скачать протестировать и установить программу с помощью go get/test/install

Заполните пожалуйста опрос

https://otus.ru/polls/3865/

Спасибо за внимание!