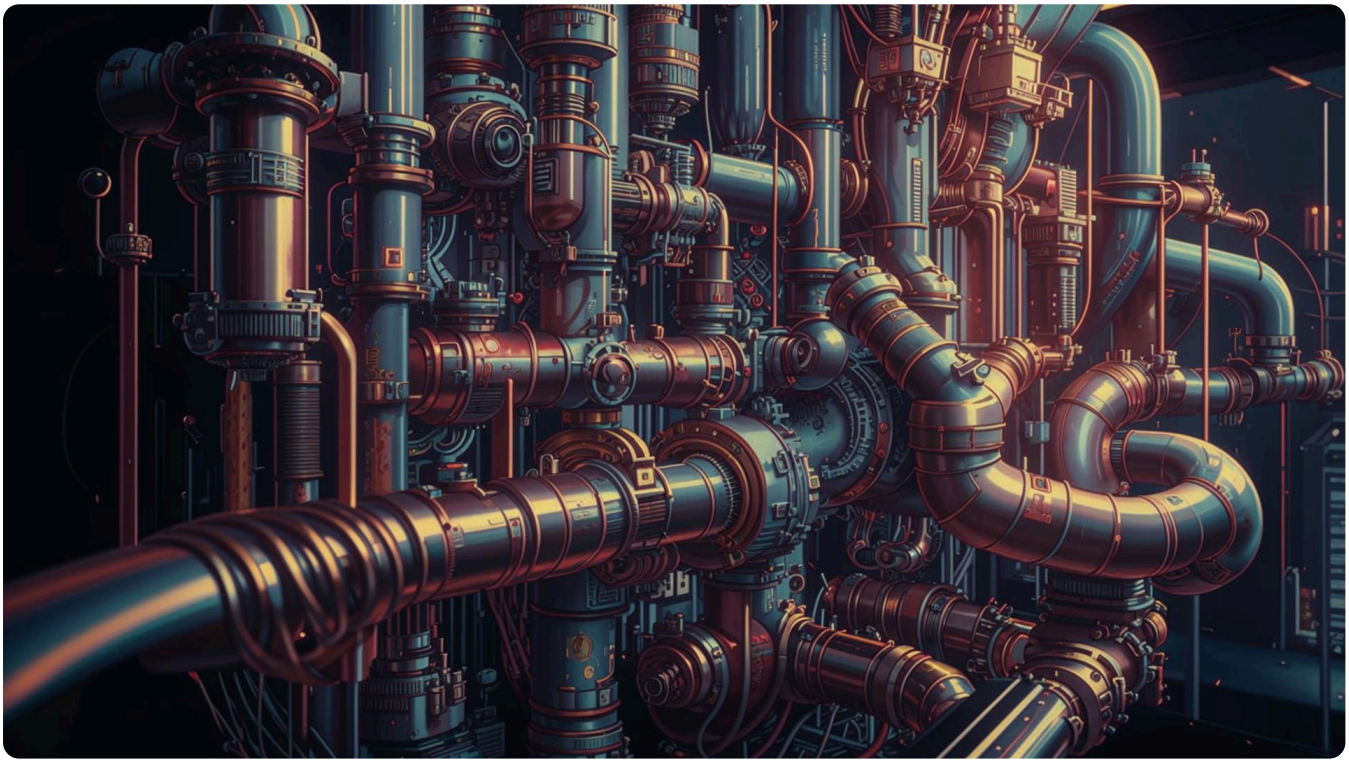
Go IO Closer, Seeker, WriterTo и ReaderFrom

июня 5, 2025 · 8 минут · German Gorelkin



Перевод/заметки Go I/O Closer, Seeker, WriterTo, and ReaderFrom

`io.Closer`

Интерфейс `Closer` разработан для работы с объектами, которые требуют освобождения ресурсов после завершения их использования.

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

Обычно вы не встретите объект `Closer`, который существует сам по себе. В стандартной библиотеке Go он входит в состав других интерфейсов, таких как `io.ReadCloser`, `io.WriteCloser` или `io.ReadWriteCloser`. Например, когда вы завершаете работу с файлом, сетевым соединением или с базой данных, вы вызываете метод `Close()`, чтобы привести все в порядок и освободить ресурсы.

Что произойдет, если вызвать `Close()` несколько раз?

Например, при попытке закрыть объект `os.File` во второй раз возникнет ошибка `os.ErrClosed`. Однако, когда дело доходит до `Response.Body` из пакета `net/http`, то, скорее всего, не возникнет никаких проблем.

> *Что произойдёт, если я забуду закрыть файл? Может ли это привести к утечке памяти?*

Когда вы открываете файл, операционная система выделяет специальное значение, называемое дескриптором файла. Если вы не закроете файл должным образом, дескриптор останется в системе. Если вы будете открывать файлы, не закрывая их, в конце концов в системе закончатся доступные дескрипторы, и вы получите ошибку «*too many open files*».

Однако не стоит беспокоиться, если вы забудете закрыть файл, сборщик мусора в Go автоматически очистит дескриптор, когда объект файла больше не будет использоваться.

Вот как это происходит:

```
func newFile(fd int, name string, kind newFileKind, nonBlocking bool) *File {
    f := &File{&file{
        pfd: poll.FD{
            Sysfd:         fd,
            IsStream:      true,
```

```
            ZeroReadIsEOF: true,
        },
        name:        name,
        stdoutOrErr: fd == 1 || fd == 2,
    }}
    ...

    runtime.SetFinalizer(f.file, (*file).close)
    return f
}
```

Видите эту строку `runtime.SetFinalizer(f.file, (*file).close)` ?

По сути, Go регистрирует функцию `(*file).close` , которая будет вызываться, когда на файл больше не будет ссылок. Таким образом, когда сборщик мусора начнёт свою работу, он автоматически закроет файл. Однако стоит отметить, что процесс сборки мусора не происходит мгновенно. Если вы оставите много открытых файлов, они могут накопиться до того, как сборщик мусора до них доберётся.

`io.Seeker`

Когда вы открываете файл или любой другой ресурс, например, буфер или сетевой поток, по умолчанию данные считываются или записываются последовательно, начиная с самого начала и продвигаясь вперёд. Однако иногда вам может потребоваться более гибкий контроль над процессом. Например, вы можете захотеть перейти к определённому месту в файле или вернуться назад и перечитать что-то.

Именно для этого предназначен интерфейс `io.Seeker` .

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

`Seeker` позволяет перемещать указатель файла в любую позицию внутри него.

Таким образом, вы можете начать чтение или запись файла с определённого места. Функция принимает два аргумента:

- **`offset`** : Это расстояние, на которое вы хотите переместить курсор.
- **`whence`** : Точка отсчёта, определяющая, откуда начинать движение курсора.

Есть три варианта:

1. **`os.SeekStart`** : Перемещение курсора относительно начала файла. Например, вызов `Seek(0, SeekStart)` возвращает вас в начало файла, а `Seek(100, SeekStart)` перемещает курсор на 100 байт от начала.
2. **`os.SeekCurrent`** : Перемещение курсора относительно текущего положения. Если вы вызовете `Seek(-10, SeekCurrent)` , то переместитесь на 10 байт назад.
3. **`os.SeekEnd`** : Перемещает курсор относительно конца файла. Обычно используется отрицательное смещение.

Давайте посмотрим на пример:

```
func main() {
    reader := strings.NewReader("Hello, World!")

    reader.Seek(7, io.SeekStart)
    readBytes(reader)
  // Output: Read 6 bytes: "World!"

    reader.Seek(-5, io.SeekCurrent)
    readBytes(reader)
  // Output: Read 5 bytes: "orld!"

    reader.Seek(-2, io.SeekEnd)
    readBytes(reader)
  // Output: Read 2 bytes: "d!"
}

// Helper
func readBytes(reader io.Reader) {
  buffer := make([]byte, 1024)
  n, _ := reader.Read(buffer)
  fmt.Printf("Read %d bytes: %q\n", n, buffer[:n])
}
```

В этом примере функция `readBytes` считывает строку «World!» из файла после того, как указатель сдвигается на 7 байт вперед от начала. Затем указатель перемещается в конец файла.

Важно отметить, что если вы открываете файл в режиме добавления (флаг `O_APPEND`), то поведение функции `Seek` становится несколько непредсказуемым. Это связано с тем, что при открытии файла с этим флагом указатель автоматически перемещается в конец перед каждой операцией записи, что может вызвать неожиданные результаты.

io.WriterTo

Обычно при работе с файлами или потоками данных мы следуем традиционному подходу: вызываем `Read()` на источнике и `Write()` на получателе, передавая данные небольшими фрагментами. Однако иногда такой метод кажется неэффективным, так как мы перемещаем данные туда и обратно в несколько этапов.

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

Метод `WriteTo(w Writer)` вызывается на объекте-источнике и напрямую записывает его данные в конечный объект `w`. В большинстве случаев `WriterTo` хорошо сочетается с `io.Reader`, позволяя читателю взять на себя управление и одним махом записать всё, что у него есть, в конечный объект.

Это упрощает весь процесс.

Если вы хотите сделать передачу данных более эффективной или внести некоторые изменения, рассмотрите возможность реализации этих интерфейсов. Они будут **иметь приоритет** над стандартными вызовами `Read()` и `Write()`, предоставляя вам больше контроля и потенциально улучшая производительность.

Давайте разберемся, что на самом деле означает «*иметь приоритет*» на практике, рассмотрев, как работает `io.Copy()`:

- **WriterTo**: Если источник (читатель) реализует интерфейс `WriterTo`, это означает, что он знает, как записать свои данные непосредственно в

получатель. В таком случае метод `io.Copy()` вызывает метод `WriteTo()`, избавляя от необходимости в дополнительном буфере.

- **ReaderFrom**: Если получатель (писатель) реализует интерфейс `io.ReaderFrom`, он умеет читать данные непосредственно из источника. Поэтому метод `io.Copy()` вызывает метод `ReadFrom()`.
- **32 КБ Buffer Fallback**: Если ни один из этих методов не реализован, то `io.Copy()` вернется к обычному методу: чтению данных из источника во внутренний буфер и записи их в получатель. Как вы помните из предыдущей статьи, размер буфера по умолчанию составляет 32КБ.

Ещё один важный момент: эти интерфейсы также являются приоритетными для буферизованных читателей и писателей `bufio.Reader` и `bufio.Writer`. Это означает, что эти буферизованные средства чтения и записи будут искать реализации `WriterTo` и `ReaderFrom` и использовать их, если они доступны.

> *Подождите, есть ли способ лучше, чем копирование фрагментами по 32 КБ? Я имею в виду, что поведение `io.Copy()` уже является достаточно хорошим, не так ли?*

Например, `os.File` реализует интерфейс `WriterTo`. Это означает, что если вы хотите скопировать данные из одного файла в другой, `os.File` может сразу приступить к работе и записать содержимое в целевой файл без необходимости разбивать его на стандартные фрагменты по 32КБ.

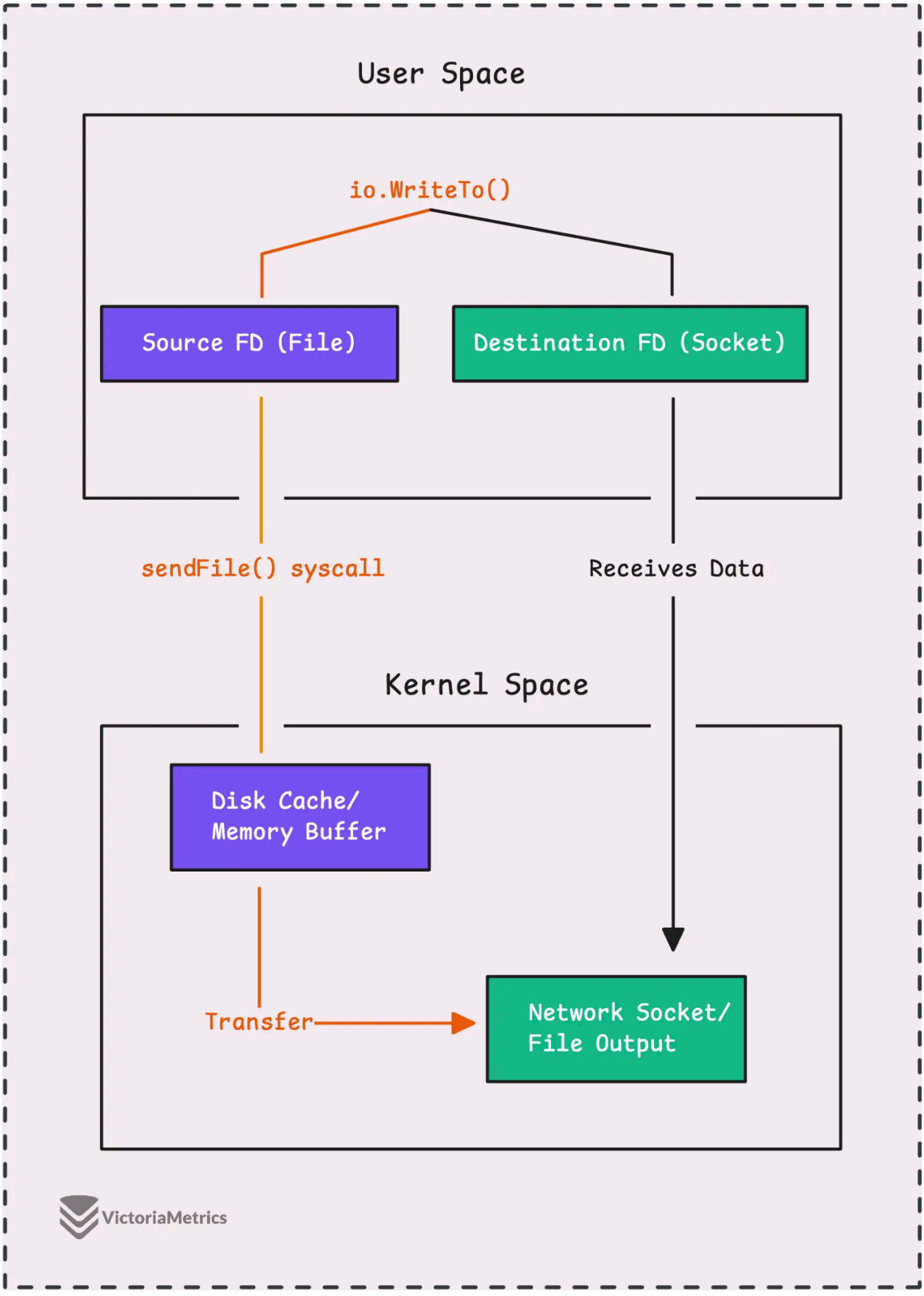
```
func main() {
    f, _ := os.Open("source.txt")
    defer f.Close()

    destFile, _ := os.Create("destination.txt")
    defer destFile.Close()

    n, _ := f.WriteTo(destFile)

    fmt.Printf("Wrote %d bytes\n", n)
}
```

В этом случае, вместо того чтобы выполнять цикл чтения и записи вручную, мы просто используем функцию `WriteTo()`, которая обрабатывает всё за один шаг.

> *Однако, почему это более быстрый метод, чем использование `io.Copy()` ?*

Допустим, вам нужно прочитать данные из файла и отправить их через сетевой сокет. Обычно данные считываются с диска в память вашего приложения, а затем отправляются обратно в пространство ядра через сокет. Это требует большого количества операций ввода-вывода.

Однако в Linux есть системный вызов, который позволяет избежать необходимости обращаться в пользовательское пространство. С его помощью можно передавать данные напрямую из одного файлового дескриптора (например, из файла) в другой (например, в сетевой сокет), что значительно ускоряет процесс.



Это гораздо более эффективный способ, поскольку он позволяет избежать копирования данных между пространством пользователя и пространством ядра.

Если вы работаете на другой платформе или такая прямая передача данных через файловый дескриптор невозможна, `io.Copy()` будет работать как обычно, читая и записывая данные кусками по 32КБ.

Вот ещё один пример: `bytes.Buffer`. Поскольку он предоставляет вам прямой доступ к своему внутреннему буферу, вы уже знаете, какого размера ваши данные, и они все находятся в памяти. Если бы вы использовали метод `io.Copy()`, то создали бы дополнительный буфер и копировали данные по частям, что было бы избыточно. Вместо этого `bytes.Buffer` может записать всё своё содержимое сразу в другой `io.Writer`, что позволяет избежать лишних действий.

`io.ReaderFrom`

Интерфейс `io.ReaderFrom` обычно реализуется типами, которые также являются `io.Writer`. Он создан для более эффективного чтения данных из источника в свой объект.

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

Хорошим примером является `os.File`, который поддерживает метод `ReadFrom()` для чтения данных непосредственно из любого устройства чтения в файл:

```
func main() {
  f, _ := os.Create("destination.txt")
  defer f.Close()

  r := strings.NewReader("This is some data to be read")

  n, _ := f.ReadFrom(r)
  fmt.Printf("Read %d bytes\n", n)
}
```

Как и в случае с `WriterTo`, если ваша ОС поддерживает более эффективный способ передачи данных, `ReadFrom()` будет использовать его. Если нет, то он

вернется к стандартному методу `io.Copy()`, передавая данные по 32КБ за раз.

`io.ByteReader/Writer` И `io.RuneReader/Writer`

Интерфейсы `ByteReader/Writer` и `RuneReader/Writer` удобны, когда вам нужно работать с данными по одному байту или символу за раз.

```
func main() {
    data := `{"name": "VictoriaMetrics", "age": 8}`
    reader := strings.NewReader(data)
    var b byte
    for {
        b, _ = reader.ReadByte()
        if b == '8' {
            b = '9'
        }
        fmt.Printf("%c", b)
    }
}

// Output:
// {"name": "VictoriaMetrics", "age": 9}
```

С другой стороны, у нас есть `RuneReader`, который становится особенно полезным, когда дело доходит до работы с текстом, особенно с Unicode.

Когда вы имеете дело с текстом, вас интересуют символы (или «руны» в Go), а не отдельные байты. Поскольку символы Unicode могут занимать от 1 до 4 байт, использование метода `Read([]byte)` вынудит вас самостоятельно декодировать эти байты в символы, что может быть довольно сложной задачей.

```
func main() {
    // Create a reader with a string that includes some emoji
    data := "Hello 🌍"

    // First loop using ReadByte
    bufReader := bufio.NewReader(strings.NewReader(data))
```

```
        for {
            part1, err := bufReader.ReadByte()
            ...

            fmt.Printf("ReadByte: %q (byte value: %d)\n", part1, part1)
        }

        // Second loop using ReadRune
        bufReader = bufio.NewReader(strings.NewReader(data))
        for {
            r, size, err := bufReader.ReadRune()
            ...

            fmt.Printf("ReadRune: %c (size: %d bytes)\n", r, size)
        }
    }

    // Output:
    // ReadByte: 'H' (byte value: 72)
    // ReadByte: 'e' (byte value: 101)
    // ReadByte: 'l' (byte value: 108)
    // ReadByte: 'l' (byte value: 108)
    // ReadByte: 'o' (byte value: 111)
    // ReadByte: ' ' (byte value: 32)
    // ReadByte: 'ð' (byte value: 240)
    // ReadByte: '\u009f' (byte value: 159)
    // ReadByte: '\u008c' (byte value: 140)
    // ReadByte: '\u008d' (byte value: 141)

    // ReadRune: H (size: 1 bytes)
    // ReadRune: e (size: 1 bytes)
    // ReadRune: l (size: 1 bytes)
    // ReadRune: l (size: 1 bytes)
    // ReadRune: o (size: 1 bytes)
    // ReadRune:   (size: 1 bytes)
    // ReadRune: 🌍 (size: 4 bytes)
```

Сначала мы читаем по одному байту с помощью `ReadByte()`. Это отлично работает для символов ASCII.

Но когда мы добираемся до эмодзи 🌍, все становится запутанным, потому что они состоят из 4 байт. `ReadRune` правильно считывает символы, независимо от того, состоят они из одного или нескольких байт.

Эти интерфейсы реализуются в большинстве стандартных типов, таких как `bufio.Reader/Writer` и `bytes.Buffer`. Обычно они должны иметь внутренний буфер, чтобы избежать многократного обращения к базовому ресурсу.

Комментарии в Telegram-группе!

go

« ПРЕДЫДУЩАЯ
Манифест пост-меритократии

СЛЕДУЮЩАЯ »
12 практических упражнений от Эпиктета