pkg/io

13 Mar 2019

Tomasz Grodzki
Co-Founder & Developer, AlphaSOC

Interfaces

Consist of one method:

```
Reader
Writer
Closer
Seeker

ReaderAt
WriterAt

ByteReader
RuneReader

StringWriter
ByteWriter

ReaderFrom
WriterTo
```

2

Interfaces

Composed of other interfaces:

```
ReadCloser
ReadSeeker
WriteCloser
WriteSeeker

ReadWriter
ReadWriteCloser
ReadWriteSeeker

ByteScanner
RuneScanner
```

3

Interfaces

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

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

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

4

Interfaces

```
type ReadCloser interface {
        Reader
        Closer
}

type WriteCloser interface {
        Writer
        Closer
}

type ReadWriteCloser interface {
        Reader
        Writer
        Closer
}
```

5

Interfaces

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

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

6

Interfaces

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

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

Interfaces

Interfaces have rich documentation specifing the expected behaviour.

A lot of corner cases.

See: io.Reader (https://golang.org/pkg/io/#Reader), io.Writer (https://golang.org/pkg/io/#Writer)

8

Wrappers

Reader

```
func LimitReader(r Reader, n int64) Reader
func MultiReader(readers ...Reader) Reader
func TeeReader(r Reader, w Writer) Reader
```

Writer

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

9

Section reader

SectionReader implements Read, Seek, and ReadAt on a section of an underlying ReaderAt.

```
type SectionReader
    func NewSectionReader(r ReaderAt, off int64, n int64) *SectionReader
    func (s *SectionReader) Read(p []byte) (n int, err error)
    func (s *SectionReader) ReadAt(p []byte, off int64) (n int, err error)
    func (s *SectionReader) Seek(offset int64, whence int) (int64, error)
    func (s *SectionReader) Size() int64
```

10

Functions

Copy from source (Reader) to destination (Writer):

```
func Copy(dst Writer, src Reader) (written int64, err error)
```

Same, but allocation can be avoided:

```
func CopyBuffer(dst Writer, src Reader, buf []byte) (written int64, err error)
```

Copy n bytes:

```
func CopyN(dst Writer, src Reader, n int64) (written int64, err error)
```

Uses `WriterTo` or `ReaderFrom` interfaces if possible.

Functions

Read exactly `len(buf)` bytes:

```
func ReadFull(r Reader, buf []byte) (n int, err error)
```

Read at least `min` (and at most `len(buf)`) bytes:

```
func ReadAtLeast(r Reader, buf []byte, min int) (n int, err error)
```

12

Functions

Write `string` to Writer (accepting `[]byte`):

```
func WriteString(w Writer, s string) (n int, err error)
```

Similar to executing:

```
w.Write([]bytes(s))
```

but `WriteString` method will be used if implemented:

```
func WriteString(w Writer, s string) (n int, err error) {
    if sw, ok := w.(StringWriter); ok {
        return sw.WriteString(s)
    }
    return w.Write([]byte(s))
}
```

13

Pipes

Pipe creates a synchronous in-memory pipe.

Can be used to connect code expecting io.Reader with code expecting io.Writer.

There is no internal buffering.

It is safe to call Read and Write in parallel with each other or with Close.

Examples

15

Copy

One-liner copying user input to standard output:

```
package main

import (
    "io"
    "os"
)

func main() {
    io.Copy(os.Stdout, os.Stdin)
}
```

16

Copy (http)

```
package main

import (
    "io"
    "log"
    "net/http"
)

func main() {
    handler := func(w http.ResponseWriter, r *http.Request) {
        io.Copy(w, r.Body)
    }

    log.Fatal(http.ListenAndServe(":8080", http.HandlerFunc(handler)))
}
```

Run

17

Copy and print

```
package main

import (
    "io"
    "log"
    "net/http"
    "os"
)

func main() {
    handler := func(w http.ResponseWriter, r *http.Request) {
        io.Copy(io.MultiWriter(w, os.Stdout), r.Body)
    }

    log.Fatal(http.ListenAndServe(":8080", http.HandlerFunc(handler)))
}
```

Run

Note: writing to os.Stdout from multiple routines is not thread-safe.

Copy and print

```
package main

import (
    "io"
    "log"
    "net/http"
    "os"
)

func main() {
    handler := func(w http.ResponseWriter, r *http.Request) {
        io.Copy(w, io.TeeReader(r.Body, os.Stdout))
    }

    log.Fatal(http.ListenAndServe(":8080", http.HandlerFunc(handler)))
}
```

Run

19

Count bytes

```
type Counter struct {
    Bytes int
    Lines int
}

func (r *Counter) Write(b []byte) (n int, err error) {
    r.Bytes += len(b)
    r.Lines += bytes.Count(b, []byte{'\n'})
    return len(b), nil
}

var handler = func(w http.ResponseWriter, r *http.Request) {
    cnt := new(Counter)
    io.Copy(io.MultiWriter(w, os.Stdout, cnt), r.Body)
    fmt.Printf("\n===\nBytes: %d\nLines: %d\n", cnt.Bytes, cnt.Lines)
}
```

Run

20

Pipe

```
pr, pw := io.Pipe()
defer pr.Close()

// Write stream of data into pipe in a separate routine
go func() {
    err := json.NewEncoder(pw).Encode(struct {
        Event   string
        Edition int
    }{
        "Golang Meetup Warsaw", 24,
    })

    pw.CloseWithError(err)
}()

// Read from pipe
_, err := http.Post("http://localhost:8080", "application/json", pr)
```

Run

21

Other packages

- io/ioutil (https://golang.org/pkg/io/ioutil/)
- testing/iotest (https://golang.org/pkg/testing/iotest/)

22

Summary

- Prefer pre-defined io interfaces
- Mix and match io types to achieve expected results
- Understand the contract for each interface
- Use `iotest` to test for some corner cases

Thank you

13 Mar 2019

Tomasz Grodzki
Co-Founder & Developer, AlphaSOC
tomasz@alphasoc.com (mailto:tomasz@alphasoc.com)