Learn the Go Programming Language

For experienced developers or
those of an adventurous nature

gotutorial.net
@GoTutorialNet

Matt Nunogawa
@amattn

Lesson 12

Readers, Writers, Buffers, IO!

`v0.1 draft`

Readers & Writers

- You should be familiar with interface by now.
- In CS you call it structural typing or duck typing. In ObjC or Java, these are called protocols. In python, you typically use the hasattr() to achieve the same effect.

Readers & Writers

```
// From the io pkg:

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

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

Conceptually

- Readers have data, and you read it out, and make use of that data.
- You have data and you want to shove it into a Writer where something happens to that data.

Reader Best Practices

```
Read(p []byte) (n int, err error)
```

- Read reads up to len(p) bytes into p. It returns the number of bytes read (0 <= n <= len(p)) and any error encountered.
- A successful read may return err == EOF or err == nil, depends on source and implementation.
- Even if Read returns n < len(p), it may use all of p as scratch space during the call.
- Implementations must not retain p

Reader Best Practices

```
Read(p []byte) (n int, err error)
```

- If some data is available but not len(p) bytes, Read conventionally returns what is available instead of waiting for more.
- Implementations of Read are discouraged from returning a zero byte count with a nil error, except when len(p) == 0. Callers should treat a return of 0 and nil as indicating that nothing happened; in particular it does not indicate EOF.
- Process the n > 0 bytes returned before considering the error err.

Reader Variants

```
type ReadWriter interface { Reader, Writer }
type ReadCloser interface { Reader, Closer }
type ReadSeeker interface { Reader, Seeker }
type ReadWriteCloser interface {Reader,Writer,Closer}
type ReadWriteSeeker interface {Reader,Writer,Seeker}
type ByteReader interface // single byte reader
type RuneReader interface // single rune reader

type LimitedReader struct // limited to N bytes
type PipeReader struct    // read half of a pipe
type SectionReader struct // read interior section
```

Reader Variants

```
// You can compose special readers…
func LimitReader(r Reader, n int64) Reader
func MultiReader(readers ...Reader) Reader
func TeeReader(r Reader, w Writer) Reader
```

Reader Implementations

```
Readers are all over the standard library:

bufio iotest bytes strings crypto debug packet
archive/…
image/…
compress/…
encoding/…
text/…
mime/multipart
net/textproto

// and many more…
```

Writer Best Practices

```
Write(p []byte) (n int, err error)
```

- Write must return a non-nil error if it returns n < len(p)
- Write must not modify the slice data, even temporarily
- Implementations must not retain p

Writer Variants

```
ReadWriter
StreamWriter
SegmentWriter
MultiWriter
ByteWriter
PipeWriter

// composition…
func MultiWriter(writers ...Writer) Writer
```

Writer Implementations

```
iotest/…
archive/…
compress/…
encoding/…
text/…

godoc/LinkWriter
net/http/ResponseWriter

// soooooo many more…
```

Using Readers & Writers

- Conceptually, like pipes
  - You can connect them serially
  - Or tee them off
  - Or route them to a file or stdout/err or into a buffer
    - From a buffer you can "covert" to []byte or string

OVERUSING READALL()

- io/ioutil has a convenience function to do Reader → []byte conversion. bytes.Buffer also has the similar in spirit Bytes() and String() methods.

- Do not overuse this. If your code flow is:

```
Reader.ReadAll() → []byte → Writer
```

- consider doing this instead:

```
io.Copy(dst Writer, src Reader)
```

Buffers

"In computer science, a data buffer (or just buffer) is a region of a physical memory storage used to temporarily store data while it is being moved from one place to another."

–Wikipedia, Data Buffer

Buffers in Go

- You will use these a lot. Usually you use the bytes.Buffer implementation
- Basically lets you turn a reader into []byte or string
- And lets you stuff a []byte or string into a writer
- bytes.Buffer is both a Reader and a Writer, but conceptually, figure out if you need a read buffer or a write buffer.

Golden Rule of Buffers

- Normally for a write buffer, use new(bytes.Buffer)
- Normally for a read buffer, use

```
bytes.NewBuffer(src)         // rw
bytes.NewBufferString(src) // rw
strings.NewReader(src)       // readonly
```

Read Buffer Example

```
// pkg bytes
func NewBuffer(buf []byte) *Buffer
func NewBufferString(s string) *Buffer
// pkg string
func NewReader(s string) *Reader // read-only!


buf    := bytes.NewBuffer(src_bytes)
sbuf   := bytes.NewBufferString(src_string)
rosbuf := strings.NewReader(src_string)

// now you can use any of the above anywhere a
io.Reader is required
```

Write Buffer Example

```
buf := new(bytes.Buffer)

// now you can use buf anywhere that requires
// an io.Writer.  Those functions usually will
// write data of some kind into buf.
```

Lots of Examples

FSCAN (READER)

```
func Fscan(r io.Reader, a ...interface{}) (n int, err error)
// also Fscanf and Fscanln
// Reads from the reader, separates by space or newline, stuffs
parsed values into argument pointers.

// http://play.golang.org/p/qO6Ig5ycE1

reader := strings.NewReader("source string")
var first_word string
var second_word string
var third_word string

n, err:=fmt.Fscan(reader, &first_word, &second_word, &third_word)
fmt.Println(err, n)

fmt.Println("the first word is:", first_word)
fmt.Println("the second word is:", second_word)
fmt.Println("the third word is:", third_word)
```

Fprint

```
// func Fprint(w io.Writer, a ...interface{}) (n int, err
error)

// http://play.golang.org/p/UQJxdbi7zI

write_buffer := new(bytes.Buffer)

first_word := "hi!"
second_word := 2
third_word := 3

n, err := fmt.Fprint(write_buffer, first_word,
second_word, third_word)

fmt.Println(n, err)
fmt.Println(write_buffer.String())
```

IOUTIL/READALL

```
// Read all data from r until err or EOF.
// func ReadAll(r io.Reader) ([]byte, error)

// http://play.golang.org/p/zrbEviE4fe

src := "source string"
reader := strings.NewReader(src)
b, _ := ioutil.ReadAll(reader)
read_string := string(b)

if src == read_string {
    fmt.Println("of course they are the same")
}
```

Compress/…

- In general, a compress/… package will read compressed data from a reader and decompress it
- And will write compressed data into an io.Writer

Compress/…

```
// http://play.golang.org/p/pylRKmOwLA

src := []byte("squish me")

// compress src
buf := new(bytes.Buffer)
w := gzip.NewWriter(buf)
w.Write(src)
w.Close()

// buf now contains our compressed data
gzip_reader, _ := gzip.NewReader(buf)
gzip_reader.Close()

// you can now read from gzip_reader to get the uncompressed data…

output, err := ioutil.ReadAll(gzip_reader)
fmt.Println(string(output), err)
```

COPY

```
src:= []byte(“squish me\n”)

// compress src

buf := new(bytes.Buffer)
w := gzip.NewWriter(buf)
w.Write(src)
w.Close()

// buf now contains our compressed data

gzip_reader, err := gzip.NewReader(buf)
gzip_reader.Close()

// you can now read or redirect from gzip_reader to get the
uncompressed data…
io.Copy(os.Stdout, r)
```

ENCODING/JSON

- You've no doubt seen/used Marshall/Unmarshall
- those operate on []byte slices
- There is an alternative Decoder/Encoder, which operates on io.Reader and io.Writer
  - Decoder is a struct that takes in an interface{} and writes out the encoded JSON representation
  - Encoder is a struct that takes in an interface{} and writes out the encoded JSON representation
  - See std library doc for example implentation

NET/HTTP HANDLER INTERFACE

Writer

```
type Handler interface {
        ServeHTTP(ResponseWriter, *Request)
}
```

Request.Body is a ReadCloser

NET/HTTP HANDLER INTERFACE

```
ServeHTTP(resp ResponseWriter, req *Request) {
    // get the body
    body_bytes, err := ioutil.Readall(req.Body)
    if err != nil {
        // something has gone terribly wrong.
        http.Error(resp, "shgtr", 500)
    }

    // write arbitrary data to resp:
    fmt.Fprintln(resp, "arbitrary data")

    // or pipe in data from some other reader:
    io.Copy(resp, json_buffer)

    // or use the built in convenience method:
    resp.Write([]byte(“Convenient!"))
}
```

Making a Custom Reader/Writer

Counting Buffer

- This is a working example of a custom buffer that implements writer and adds the ability to count how many bytes have been written
- The somewhat embarrassing real life code where I originally needed this: https://github.com/amattn/paral/blob/master/counting_buffer.go
- counting reader is left as an exercise to the audience

CountingBuffer

```
// Counts the total number of bytes written since
// creation or the most recent ClearCount()
// Reset() and Truncate() methods do NOT affect
// the running counts.

type CountingBuffer struct {
    totalIn  uint64
    totalOut uint64 // not yet implemented

    bytes.Buffer

    inmutex  *sync.Mutex
    outmutex *sync.Mutex
}
```

BYTES/BUFFER

```
// Creation / Mutation
func NewBuffer(buf []byte) *Buffer
func NewBufferString(s string) *Buffer
func (b *Buffer) Grow(n int)
func (b *Buffer) Reset()
func (b *Buffer) Truncate(n int)

// metadata
func (b *Buffer) Len() int

// getters
func (b *Buffer) Bytes() []byte
func (b *Buffer) Next(n int) []byte
func (b *Buffer) String() string
```

BYTES/BUFFER

```
// Readers
func (b *Buffer) Read(p []byte) (n int, err error)
func (b *Buffer) ReadByte() (c byte, err error)
func (b *Buffer) ReadBytes(delim byte) (line []byte, err error)
func (b *Buffer) ReadRune() (r rune, size int, err error)
func (b *Buffer) ReadString(delim byte) (line string, err error)
func (b *Buffer) UnreadByte() error
func (b *Buffer) UnreadRune() error
func (b *Buffer) WriteTo(w io.Writer) (n int64, err error)

// Writers
func (b *Buffer) Write(p []byte) (n int, err error)
func (b *Buffer) WriteByte(c byte) error
func (b *Buffer) WriteRune(r rune) (n int, err error)
func (b *Buffer) WriteString(s string) (n int, err error)
func (b *Buffer) ReadFrom(r io.Reader) (n int64, err error)
```

CountingBuffer

```
// There are a few methods to override/implement:

Write(p []byte) (n int, err error)
WriteString(s string) (n int, err error)
WriteByte(c byte) error
WriteRune(r rune) (n int, err error)
// don't be fooled! this writes to the buffer!
// This is the “continuous, real time, streaming”
// writer implementation…
ReadFrom(r io.Reader) (n int64, err error)
```

CountingBuffer

```
func (cb *CountingBuffer) Write(p []byte) (n int, err error) {
    n, err = cb.Buffer.Write(p)
    cb.totalInSafeInc(uint64(n))
    return
}
func (cb *CountingBuffer) WriteString(s string) (n int, err error) {
    n, err = cb.Buffer.WriteString(s)
    cb.totalInSafeInc(uint64(n))
    return
}
func (cb *CountingBuffer) WriteByte(c byte) error {
    err := cb.Buffer.WriteByte(c)
    if err == nil {
        cb.totalInSafeInc(1)
    }
    return err
}
func (cb *CountingBuffer) WriteRune(r rune) (n int, err error) {
    n, err = cb.Buffer.WriteRune(r)
    cb.totalInSafeInc(uint64(n))
    return
}
```

CountingBuffer

```
func (cb *CountingBuffer) ReadFrom(r io.Read
    buffer_space := 100 // we assume that mos
    buf := make([]byte, buffer_space)

    for {
        m, err := r.Read(buf)
        if m > 0 {
            n += int64(m)

            // copy whatever was written into
            mm, err2 := cb.Write(buf[0:m]
            if err2 != nil {
                return int64(mm), err2
            }
        }

        // now that our cb.Buffer h s everything, do some cleanup.
        if err == io.EOF {
            break
        }
        if err != nil {
            return n, err
        }
    // if m is largish, grow our buf
    if m > int(0.7*float64(buffer_sp
        buffer_space *= 2
        buf = make([]byte, buffer_space)
        }
    }
    return n, nil
}
```

We can't use the parent ReadFrom, That implementation blocks until EOF. instead we reimplement a simplified version of ReadFrom() here.

The actual Write (and count) happens here.

Misc

- RWMutex
  - Specially designed to lock a single writer or any number of readers

```
func (*RWMutex) Lock()      // Write lock
func (*RWMutex) Unlock()    // Write unlock
func (*RWMutex) RLock()     // Read lock
func (*RWMutex) RUnlock()   // Read unlock
```

Thank You, Credits & License

http://gotutorial.net
@GoTutorialNet

Matt Nunogawa
@amattn

- I owe many many, thanks to the many authors of Go.
- These slides are Copyright 2013-2015 Matthew Nunogawa