Beautiful IO

> A tour through standard library pkg/io and various implementations of its interfaces.

Golab 2019, 2019–10–21, Florence

Martin Czygan

About me

SWE @ubleipzig working mostly with Python and Go.

Taming data – open source – writing.

> Explore IO workshop at Golab 2017.

Background

- Go Proverbs (2015)

> The bigger the interface, the weaker the abstraction.

Prominent examples are `io.Reader` and `io.Writer`.

The IO package

- contains basic, widely used interfaces (within and outside standard library)
- utility functions

Why beautiful?

> La bellezza è negli occhi di chi guarda

- small, versatile interfaces
- composable

Praise and love

> This article aims to convince you to use io.Reader in your own code wherever you can. -- @matryer

> "Crossing Streams: a love letter to Go io.Reader" -- @jmoiron

> Which brings me to io.Reader, easily my favourite Go interface. -- @davecheney

What's in pkg/io?

- 25 types

- 21/25 are interfaces

- 12 functions, 3 constants, 6 errors

The concrete types are: `LimitedReader` , `PipeReader` , `PipeWriter` , `SectionReader` ; functions: `Copy` , `CopyN` , `CopyBuffer` , `Pipe` , `ReadAtLeast` , `ReadFull` , `WriteString` , `LimitReader` , `MultiReader` , `TeeReader` , `NewSectionReader` , `MultiWriter`

A few Interfaces

	R	W	C	S
io.Reader	x			
io.Writer		x		
io.Closer			x	
io.Seeker				x
io.ReadWriter	x	x		
io.ReadCloser	x		x	
io.ReadSeeker	x			x
io.WriteCloser		x	x	
io.WriteSeeker		x		x
io.ReadWriteCloser	x	x	x	
io.ReadWriteSeeker	x	x		x

Missing interfaces

You might find some missing pieces elsewhere.

```
https://github.com/go4org/go4/blob/94abd6928b1da39b1d757b60c93fb2419c409
33   // A ReadSeekCloser can Read, Seek, and Close.
34   type ReadSeekCloser interface {
35           io.Reader
36           io.Seeker
37           io.Closer
38   }
39
40   type ReaderAtCloser interface {
41           io.ReaderAt
42           io.Closer
43   }
```

How many readers, writers are there?

```
$ guru -json implements /usr/local/go/src/io/io.go:#3309
```

I counted over 280 implementations of io.Reader and X of io.Writer.

What is a Reader?

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

The reader implementation will populate a given byte slice.

- at most `len(p)` bytes are read

- to signal the end of a stream, return `io.EOF`

There is some flexibility around the end of a stream.

> Callers should always process the n > 0 bytes returned before considering the error err. Doing so correctly handles I/O errors that happen after reading some bytes and also both of the allowed EOF behaviors.

Notes

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

- The byte slice is under the control of the caller.

> Implementations must not retain p.

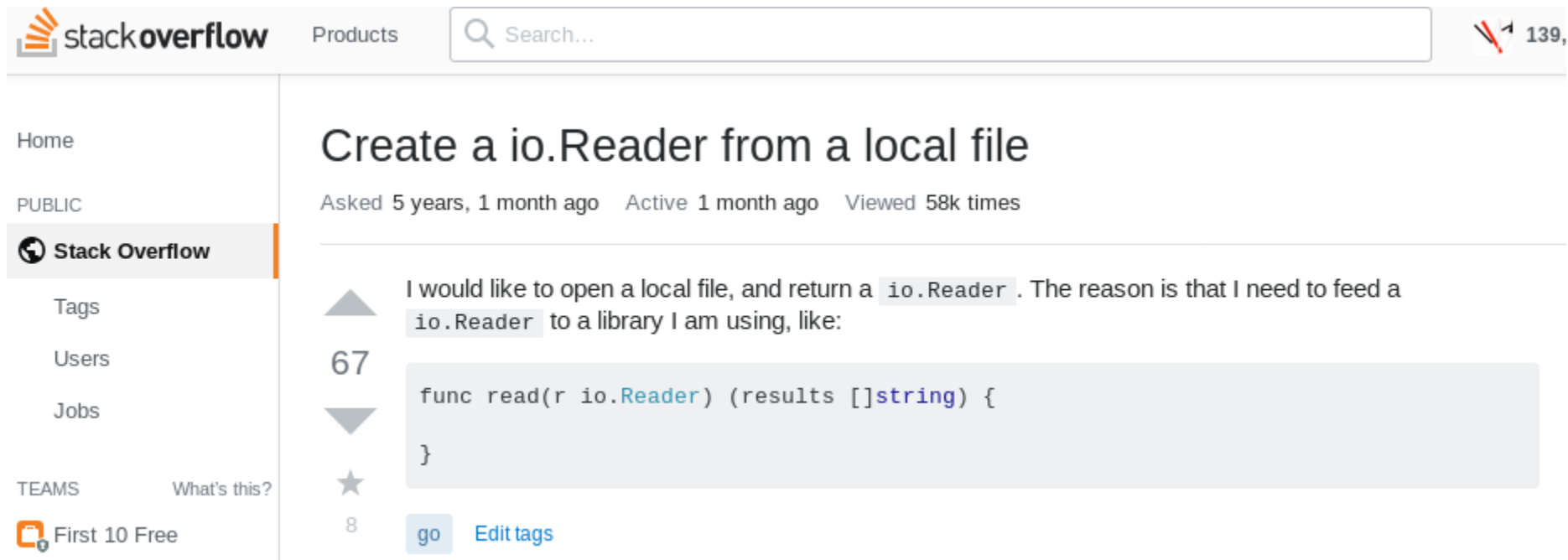This hints at the streaming nature of this interface.

Implementations

- files

- network connections

- HTTP response bodies

- standard input and output

- compression

- hashing

- encoding

- formatting

- ...

Many uses in testing as well.

Structural typing

- conversions are not required, a file implements `Read` and hence *is* a
  io.Reader

Streams

As layed out in the *love letter*, the use of `ioutil.ReadAll` is debatable.

It's in the standard library and useful, but not always necessary.

```
b, err := ioutil.ReadAll(r)
...
```

Streams

- you may lose the advantage to use the `Reader` in other places

- you may consume more memory

> Streams can trivially produce infinite output while using barely any memory at all - imagine an implementation behaving like /dev/zero or /dev/urandom.

- Memory control is an important advantage.

Follow the stream

Instead of writing:

```go
b, _ := ioutil.ReadAll(resp.Body) // Pressure on memory.
fmt.Println(string(b))
```

You may want to connect streams:

```go
_, _ = io.Copy(os.Stdout, resp.Body)
```

Stream advantages

- memory efficient

- can work with data, that does not fit in memory

- allows to work on different protocol parts differently (e.g. HTTP header vs HTTP body)

Another example

We often need to unmarshal JSON.

```
_ = json.Unmarshal(data, &v) // data might come from ioutil.ReadAll(resp.Body)
```

But we can decode it as well.

```
_ = json.NewDecoder(resp.Body).Decode(&v)
```

In this case, the JSON data must be fully read, so this is a weak example.

Glipse at composition

But what is we want need to preprocess the data, e.g. decompress it. Streams compose well.

```
zr, _ = gzip.NewReader(resp.Body)
_ json.NewDecoder(zr).Decode(&)
```

How do you implement one yourself?

You only need a `Read` method with the correct signature.

- Example: `/dev/zero`

```go
type devZero struct{}

func (r *devZero) Read(p []byte) (int, error) {
    for i := 0; i < len(p); i++ {
        p[i] = '\x00'
    }
    return len(p), nil
}
```

This is already an infinite stream.

Embed a reader

Often you want to transform a given data stream, so you embed it.

```go
type UpperReader struct {
        r io.Reader // Underlying stream
}

func (r *UpperReader) Read(p []byte) (int, error) {
        n, err := r.r.Read(p)
        copy(p, bytes.ToUpper(p))
        return n, err
}

func main() {
        if _, err := io.Copy(os.Stdout, &UpperReader{os.Stdin}); err != nil {
                log.Fatal(err)
        }
}
```

- Also try: https://tour.golang.org/methods/22