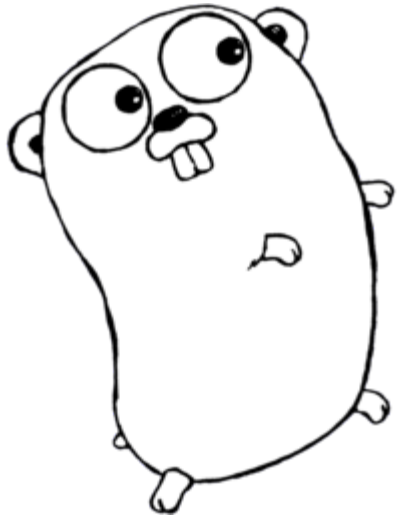

Exploring Readers and Writers

Golab.io, January 21, 2017

Martin Czygan, Leipzig University Library

Hello World

- Programmer at Leipzig University Library
- We develop search solutions for libraries
- We're using Go for about three years

Reading? Writing?

Motivation

Motivation. Go proverbs talk, Gopherfest, 2015.

> The bigger the interface, the weaker the abstraction.

Motivation

> ... satisfied implictly. But that's actually not the most important thing about Go's interfaces. The really most important thing is the culture around them that's captured by this proverb, which is that the smaller the interface is the more useful it is.

> io.Reader, io.Writer and the empty interface are the three most important interfaces in the entire ecosystem, and they have an average of 2/3 of a method.

[gentle laughter]

Motivation

> If you think about how a Java guy would build it you would have an interface like this [...] and it would only generalize a little bit. There might be two implementations of it. How many implementations of io.Reader are there? **I've written programs with twenty implementations of io.Reader inside.**

Motivation

And finally:

> This is really a Go specific idea here, that we want to make little interfaces, so that we can build components that share them.

This workshop should serve an illustration of the above. It's a rich topic, so we will only cover part of it.

Workshop goals

- See io.Reader and io.Writer in action. See implementations and contexts.
- Explore the interface idea as realized in Go.
- Stream composition.

Workshop goals

After this workshop, you:

- can implement a readers and writers in your own projects,
- recognize many implementations from the standard library,
- are aware when readers and writers could simplify an implementation.

Workshop roadmap

- Introduction and some background, then
- ~35 short quizzes and examples.
- Solutions.md

Workshop roadmap

All code is it at: https://github.com/miku/exploreio

Clone it or go get it.

- s00, s01, s02, ... s30, self contained examples with TODO
- try to resolve the TODO
- then: short recap, then: next

Workshop roadmap

- Everybody has the code?

Introduction

Do you know the signatures of io.Reader or io.Writer by heart?

Introduction

Defined in package io, along with a few others.

```
// Reader is the interface that wraps the basic Read method.
// ...
type Reader interface {
    func Read(p []byte) (n int, err error)
}
```

```
// Writer is the interface that wraps the basic Write method.
// ...
type Writer interface {
    func Write(p []byte) (n int, err error)
}
```

What is a Reader?

```
// Reader is the interface that wraps the basic Read method.
// ...
type Reader interface {
    func Read(p []byte) (n int, err error)
}
```

- What is it? Something that when given some space ([]byte) is able to populate
  it and eventually signal an end (io.EOF). Must not retain p.

What is a Writer?

```
// Writer is the interface that wraps the basic Write method.
// ...
type Writer interface {
    func Write(p []byte) (n int, err error)
}
```

- What is it? Something ([]byte) it is able to write it. Must not modify or retain p.

Intro

- dealing with byte streams (that can have an end)

> **A file is simply a sequence of bytes**. Its main attribute is its size. By contrast, on more conventional systems, a file has a dozen or so attributes. To specify and create a file it takes endless amount of chit-chat. If you are on a UNIX system you can simply ask for a file and use it interchangeble whereever you want a file. https://youtu.be/tc4ROCJYbm0?t=12m55s (Thompson, 1982)

Intro

- From Wikipedia: Everything is a file (descriptor)

> ... that a wide range of input/output resources such as documents, directories, hard-drives, modems, keyboards, printers and even some inter-process and network communications are simple streams of bytes exposed through the filesystem name space.

Or:

> The UNIX philosophy is often quoted as "everything is a file", but that really means "everything is a stream of bytes" (Torvalds, 2007).

Intro

> Historically, UNIX was the first operating system to abstract all I/O under such a unified concept and small set of primitives. At the time, most operating systems were providing a distinct API for each device or device family ...

Intro

> The unified API feature is extremely empowering and fundamental for UNIX programs: you can write a program processing a file while being unaware of whether the file is actually stored on a local disk, stored on a remote drive somewhere on the network, streamed over the Internet, typed interactively by the user or even generated in memory by another program.
> This dramatically **reduces the program complexity** and eases the developer's learning curve.

Intro

> Besides, this fundamental feature of the UNIX architecture also **makes composing programs together a snap** (you just pipe two special files: standard input and standard output).

(Hanrigou, 2012)

Intro

From: http://yarchive.net/comp/linux/everything_is_file.html

> In [...], you have 15 different versions
> of "read()" with sockets and files and pipes all
> having strange special cases and special system calls.
> That's not the [...] way. It should be just a "read()",
> and then people can use general libraries and treat all
> sources the same.

Intro: Filters

> The Pipes and Filters architectural pattern provides a structure for systems
> that process a stream of data. Each processing step is encapsulated in a filter
> component. Data [are] passed through pipes between adjacent filters.
> Recombining filters allows you to build families of related filters.

From: https://john.cs.olemiss.edu/~hcc/csci581oo/notes/pipes.html

Intro: Filters

> The filters are the processing units of the pipeline. A filter may enrich,
> refine, or transform its input data [Buschmann].

- It may **enrich** the data by computing new information from the input data
  and adding it to the output data stream.

Intro: Filters

- It may **refine** the data by concentrating or extracting information from the
  input data stream and passing only that information to the output stream.
- It may **transform** the input data to a new form before passing it to the output
  stream.
- It may, of course, do some **combination** of enrichment, refinement, and transformation.

Intro: Differences

- What is different in Go?
- Differences to java.lang.Readable or Python

Examples

- in each directory, there is a main.go
- expected output is in the comment
- solving TODOs

Examples

Starting with s00.

Examples

...

Wrap-up

- We say many examples, but there are much more.

```
$ godoc -http=":6060" -analysis="type"
```

- https://tour.golang.org/methods/23
- Write a "smallest buffer"

More implementations:

- composable buffers: https://github.com/djherbis/buffer
- circular buffer: https://github.com/ashishgandhi/buffer

Wrap-up

Thanks!