Go, The Standard Library

Real Code. Real Productivity. Master The Go Standard Library

Daniel Huckstep

Go, The Standard Library

Real Code. Real Productivity. Master The Go Standard Library

Daniel Huckstep

This book is for sale at http://leanpub.com/go-thestdlib

This version was published on 2013-03-10

This is a Leanpub book. Leanpub empowers authors and publishers with the Lean Publishing process. Lean Publishing is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

Tweet This Book!

Please help Daniel Huckstep by spreading the word about this book on Twitter!

The suggested hashtag for this book is #GoTheStdLib.

Find out what other people are saying about the book by clicking on this link to search for this hashtag on Twitter:

https://twitter.com/search/#GoTheStdLib

Contents

Introduction

When I sit down to build a new piece of software in my favorite programming language of the week, I open up my programmer's toolbox. I can pull out a number of things, like my knowledge of the language syntax and its quirks. It probably has some sort of library packaging system (rubygems[1] or python eggs[2]), and I have my list of libraries for doing certain jobs. The language also has a **standard library**. All of these tools combine to help solve difficult programming

Right now, my programming language of choice is Go[3] and it has a wonderful standard library. That standard library is what this book is about.

I wanted to take an in depth look at something which normally doesn't get a lot of press, and many developers overlook. The standard library usually has a number of great solutions to problems that you might be using some other dependency for, simply because you don't about them. *It makes no sense for my application depend on an external library or program if the standard distribution of the language has something built in.*[4]

Learning the ins and outs of your favorite programming language's standard library can help make you a better programmer, and streamline your applications by removing dependencies. If this sounds like something you're interested in, keep reading.

Target Audience

This book is for people that know how to program Go already. It's definitely not an intro. If you're completely new to Go, start with the documentation page[5] and the reference page[6]. The language specification is quite readable and if you're already familiar with other programming languages you can probably absorb the language from the spec.

If you know Go but want to step up your game and your usage of the standard library, this book is for you.

How To Read This Book

My goal for this book is a *readable reference*. I do want you to read it, but I also want you to be able to pull it off the electronic shelf and remind yourself of how to do something, like writing a zip file. It's not meant to be a replacement for the package reference[7] which is very useful to remember the

[1] http://rubygems.org/

[2] http://pypi.python.org/pypi/

[3] http://golang.org/

[4] Not to mention, the library you are using might only work on one operating system, while the standard library should work everywhere the language works.

[5] http://golang.org/doc/

[6] http://golang.org/ref/

[7] http://golang.org/pkg/

details about a specific method/function/type/interface.

So feel free to read from cover to cover, and in fact I recommend this approach. If you see something that does quite work with reading this way, let me know. Alternatively, try reading individual chapters when you start to deal with a given package to get a feel for it, and come back to skim to refresh your memory.

Code In The Book

All the code listed in the book is available on Github[8]

- Zip[9]
- Tar/Gzip[10]

Anything with a main package should be able to be executed with `go run` by Go Version 1. If it's not, please let me know, with as much error information as possible.

Some code may depend on output from previously shown code in the same chapter. For example, the tar archive reading code reads the tar created in the writing code.

Frequently I'll use other packages to make my life easier when write example code. Don't worry too much about it. If you're confused about some use of a package you're not familiar with yet, either try to ignore the details and trust that I'll explain it later, or jump ahead and choose your own adventure!

[8] https://github.com/darkhelmet/go-thestdlib
[9] https://github.com/darkhelmet/go-thestdlib/zipball/master
[10] https://github.com/darkhelmet/go-thestdlib/tarball/master

Thanks

Thanks for buying and checking out this book. As part of the lean publishing philosophy, you'll be able to interact with me as the book is completed. I'll be able to change things, reorganize parts, and generally make a better book. I hope you enjoy.

A big thanks goes out to all those who provided feedback during the writing process:

- Brad Fitzpatrick
- Mikhail Strebkov

archive

Meet The Archive Package

The `archive` package is used to read and write files in tar and zip format. Both formats pack multiple files into one big file, the main difference being that zip files support optional compression using the DEFLATE algorithm provided by the `compress/flate` package.

Writing tar Files

Writing a tar file starts with `NewWriter`. It takes an `io.Writer` type, which is just something that has a method that looks like `Write([]byte) (int, error)`. This is nice if you want to generate a tar file on the fly and write it out to an HTTP response, or feed it through another writer like a gzip writer. You'll see this *just give me an* `io.Writer` pattern a lot in the Go stdlib. In our case, I'm just going to write the archive out to a file.

Make sure to close the writer you pass in *after* you close the tar writer. It writes 2 zero blocks to finish up the file, but ignores any errors during this process. This *trailer* isn't strictly required, but it's good to have. If you use `defer` in the natural order, you should be okay.

To add files to the new tar writer, use `WriteHeader`. It needs a `Header` with all the information about this entry in the archive, including its name, size, permissions, user and group information, and all the other bits that get set when the tar file gets unpacked. Straight from the Go documentation, the `Header` type looks like this:

archive/tar_header.go

```
type Header struct {
    Name       string    // name of header file entry
    Mode       int64     // permission and mode bits
    Uid        int       // user id of owner
    Gid        int       // group id of owner
    Size       int64     // length in bytes
    ModTime    time.Time // modified time
    Typeflag   byte      // type of header entry
    Linkname   string    // target name of link
```

```
    Uname      string    // user name of owner
    Gname      string    // group name of owner
    Devmajor   int64     // major number of character or block device
    Devminor   int64     // minor number of character or block device
    AccessTime time.Time // access time
    ChangeTime time.Time // status change time
}
```

Some fields aren't really required if you're doing something quick and dirty, and some only apply to certain types of entries (controlled by the `Typeflag` field). For example, if you're packaging a regular file, you don't need to worry about `Devmajor` and `Devminor`.

I found that on top of the obvious `Name` and `Size` fields, I had to set the `ModTime` on the `Header`. GNU tar would unpack the file fine, but running the read script would throw the standard "archive/tar: invalid tar header" error back at me.

Let's see it all together:

archive/write_tar.go

```
package main

import (
    "archive/tar"
    "fmt"
    "io"
    "log"
    "os"
)

var files = []string{"write_tar.go", "read_tar.go"}

func addFile(filename string, tw *tar.Writer) error {
    file, err := os.Open(filename)
    if err != nil {
        return fmt.Errorf("failed opening %s: %s", filename, err)
    }
    defer file.Close()
```

```
    stat, err := file.Stat()
    if err != nil {
        return fmt.Errorf("failed file stat for %s: %s", filename, err)
    }

    hdr := &tar.Header{
        ModTime: stat.ModTime(),
        Name:    filename,
        Size:    stat.Size(),
        Mode:    int64(stat.Mode().Perm()),
    }

    if err := tw.WriteHeader(hdr); err != nil {
        return fmt.Errorf("failed writing tar header for %s: %s", filename,\
 err)
    }

    copied, err := io.Copy(tw, file)
    if err != nil {
        return fmt.Errorf("failed writing %s to tar: %s", filename, err)
    }

    // Check copied, since we have the file stat with its size
    if copied < stat.Size() {
        return fmt.Errorf("wrote %d bytes of %s, expected to write %d", cop\
ied, filename, stat.Size())
    }

    return nil
}

func main() {
    file, err := os.OpenFile("go.tar", os.O_WRONLY|os.O_CREATE|os.O_TRUNC, \
0644)
    if err != nil {
        log.Fatalf("failed opening tar for writing: %s", err)
    }
    defer file.Close()

    tw := tar.NewWriter(file)
    defer tw.Close()
```

```
    for _, filename := range files {
        if err := addFile(filename, tw); err != nil {
            log.Fatalf("failed adding file %s to tar: %s", filename, err)
        }
    }
}
```

Remember to `Close` the original `io.Writer` you passed in followed by the tar writer. If you use `defer` as you normally would (and as I have in the example), this is what will happen.

Writing zip Files

Writing a zip file is similar to writing a tar file. There's a `NewWriter` function that takes an `io.Writer`, so let's use that.

The zip package has a handy helper to let you quickly write a file to the archive without much cermony. We can use the `Create(name string)` method on the zip writer we got back from `NewWriter` to add an entry to the zip; no header information needed. There is a `Header` type, which looks like this:

archive/zip_header.go

```
type FileHeader struct {
    Name             string
    CreatorVersion   uint16
    ReaderVersion    uint16
    Flags            uint16
    Method           uint16
    ModifiedTime     uint16 // MS-DOS time
    ModifiedDate     uint16 // MS-DOS date
    CRC32            uint32
    CompressedSize   uint32
    UncompressedSize uint32
    Extra            []byte
    ExternalAttrs    uint32 // Meaning depends on CreatorVersion
    Comment          string
}
```

You *can* use `CreateHeader` if you need to do something special, but `Create` creates a basic header for us and gives us a writer back. We can now use this writer to write the file into the zip archive.

Make sure to write the entire file before calling any of `Create`, `CreateHeader`, or `Close`. You can only deal with one file at a time, and you certainly can't deal with the zip after you've closed it.

archive/write_zip.go

```
package main

import (
    "archive/zip"
    "fmt"
    "io"
    "log"
    "os"
)

var files = []string{"write_zip.go", "read_zip.go"}

func addFile(filename string, zw *zip.Writer) error {
    file, err := os.Open(filename)
    if err != nil {
        return fmt.Errorf("failed opening %s: %s", filename, err)
    }
    defer file.Close()

    wr, err := zw.Create(filename)
    if err != nil {
        return fmt.Errorf("failed creating entry for %s in zip file: %s", f\
ilename, err)
    }

    // Not checking how many bytes copied,
    // since we don't know the file size without doing more work
    if _, err := io.Copy(wr, file); err != nil {
        return fmt.Errorf("failed writing %s to zip: %s", filename, err)
    }

    return nil
}

func main() {
    file, err := os.OpenFile("go.zip", os.O_WRONLY|os.O_CREATE|os.O_TRUNC, \
0644)
    if err != nil {
        log.Fatalf("failed opening zip for writing: %s", err)
    }
```

```
    defer file.Close()

    zw := zip.NewWriter(file)
    defer zw.Close()

    for _, filename := range files {
        if err := addFile(filename, zw); err != nil {
            log.Fatalf("failed adding file %s to zip: %s", filename, err)
        }
    }
}
```

As with tar files, remember to `Close` the original `io.Writer` and the zip writer (in that order).

Reading tar Files

Reading tar files is pretty straight forward. You use `NewReader` to get a handle to a `Reader` type. Like `NewWriter` taking an `io.Writer` type, `NewReader` takes an `io.Reader` type, in order to plug into other streams for reading tar files on the fly.

Once you have your `Reader`, you can iterate over the entries in the archive with the `Next` method. It returns a `Header` and possibly and error. Remember to check the error since it's used to signal the end of the archive (with `io.EOF`) and other problems. **Always check those errors!**

You can read out an entry by calling `Read` on the reader you got back from `NewReader`, or pass it to a utility function to read out the full contents of the entry. In the example, I use `io.ReadFull` to read out the appropriate number of bytes into a slice, and can then print that to `stdout`.

archive/read_tar.go

```
package main

import (
    "archive/tar"
    "fmt"
    "io"
    "log"
    "os"
    "text/template"
)

var HeaderTemplate = `tar header
```

```
Name:       {{.Name}}
Mode:       {{.Mode | printf "%o" }}
UID:        {{.Uid}}
GID:        {{.Gid}}
Size:       {{.Size}}
ModTime:    {{.ModTime}}
Typeflag:   {{.Typeflag | printf "%q" }}
Linkname:   {{.Linkname}}
Uname:      {{.Uname}}
Gname:      {{.Gname}}
Devmajor:   {{.Devmajor}}
Devminor:   {{.Devminor}}
AccessTime: {{.AccessTime}}
ChangeTime: {{.ChangeTime}}
`
var CompiledHeaderTemplate *template.Template

func init() {
    t := template.New("header")
    CompiledHeaderTemplate = template.Must(t.Parse(HeaderTemplate))
}

func printHeader(hdr *tar.Header) {
    CompiledHeaderTemplate.Execute(os.Stdout, hdr)
}

func printContents(tr io.Reader, size int64) {
    contents := make([]byte, size)
    read, err := io.ReadFull(tr, contents)

    if err != nil {
        log.Fatalf("failed reading tar entry: %s", err)
    }

    if int64(read) != size {
        log.Fatalf("read %d bytes but expected to read %d", read, size)
    }

    fmt.Fprintf(os.Stdout, "Contents:\n\n%s", contents)
}

func main() {
```

```
    file, err := os.Open("go.tar")
    if err != nil {
        log.Fatalf("failed opening go.tar (did you run `go run write_tar.go\
` first?): %s", err)
    }

    defer file.Close()

    tr := tar.NewReader(file)
    for {
        hdr, err := tr.Next()
        if err == io.EOF {
            break
        }

        if err != nil {
            log.Fatalf("failed getting next tar entry: %s", err)
        }

        printHeader(hdr)
        printContents(tr, hdr.Size)
    }
}
```

Reading zip Files

Reading zip files is a walk in the park too. Start with `OpenReader` to get a `zip.ReadCloser`. It has a collection of `File` structs you can iterate through, each one with size and other information, and an `Open` method so you can get another `ReadCloser` to read out that individual file. Simple!

archive/read_zip.go

```
package main

import (
    "archive/zip"
    "fmt"
    "io"
    "log"
    "os"
)
```

```
func printFile(file *zip.File) error {
    frc, err := file.Open()
    if err != nil {
        return fmt.Errorf("failed opening zip entry %s for reading: %s", fi\
le.Name, err)
    }
    defer frc.Close()

    fmt.Fprintf(os.Stdout, "Contents of %s:\n", file.Name)

    copied, err := io.Copy(os.Stdout, frc)
    if err != nil {
        return fmt.Errorf("failed reading zip entry %s for reading: %s", fi\
le.Name, err)
    }

    if uint32(copied) != file.UncompressedSize {
        return fmt.Errorf("read %d bytes of %s but expected to read %d byte\
s", copied, file.UncompressedSize)
    }

    fmt.Println()

    return nil
}

func main() {
    rc, err := zip.OpenReader("go.zip")
    if err != nil {
        log.Fatalf("failed opening zip for reading (did you run `go run wri\
te_zip.go` first?): %s", err)
    }
    defer rc.Close()

    for _, file := range rc.File {
        if err := printFile(file); err != nil {
            log.Fatalf("failed reading %s from zip: %s", file.Name, err)
        }
    }
}
```

Remember to `Close` the first `ReadCloser` you get from `OpenReader`, as well as all the other ones you get while reading files.

Caveats

ZIP64

The `archive/zip` package doesn't support ZIP64. Wikipedia[11] has enough useful information for us:

> The original zip format had a 4 GiB limit on various things (uncompressed size of a file, compressed size of a file and total size of the archive), as well as a limit of 65535 entries in a zip archive.

This means you're limited to that 4GB and 65535 files when creating zip files.

[11] http://en.wikipedia.org/wiki/Zip_(file_format)#ZIP64

builtin

Batteries Included

The `builtin` package isn't a real package, it's just here to document the builtin functions that come with the language. Lower level than the standard library, these things are just…there. The builtins let you do things with maps, slices, channels, and imaginary numbers, cause and deal with panics, build objects, and get size information about certain things. Honestly, most of this can be learned from the spec, but I've included it for completeness.

Building Objects

make

`make` is used to build the builtin types like slices, channels and maps. The first argument is the type, and it can be one of those three types.

In the case of channels, there is an optional second integer parameter, the *capacity*. If it's zero (or not given), the channel is unbuffered. This means writes block until there is a reader ready to receive the data, and reads block until there is a write ready to give data. If the parameter is greater than zero, the channel is buffered with the capacity specified. On these channels, reads block only when the channel is empty, and writes block only when the channel is full.

In the case of maps, the second parameter is also optional, but is rarely used. It controls the initial allocation, so if you know exactly how big your map has to be, it can be helpful. `cap` (which we'll see later) doesn't work on maps though, so you can't really examine the effects of this second parameter easily.

In the case of slices, the second parameter is **not** optional, and specifies the starting length of the slice. Oh but the plot thickens! There is an optional third parameter, which controls the starting capacity, and it can't be smaller than the length.[12] This way, you can get really specific with your slice allocation and save subsequent reallocations if you know exactly how much space you need it to take up.

[12] If you specify a length greater than the capacity, you'll get a runtime panic.

builtin/make.go

```
package main

import "log"

func main() {
    unbuffered := make(chan int)
    log.Printf("unbuffered: %v, type: %T, len: %d, cap: %d", unbuffered, un\
buffered, len(unbuffered), cap(unbuffered))

    buffered := make(chan int, 10)
    log.Printf("buffered: %v, type: %T, len: %d, cap: %d", buffered, buffer\
ed, len(buffered), cap(buffered))

    m := make(map[string]int)
    log.Printf("m: %v, len: %d", m, len(m))

    // Would cause a compile error
    // slice := make([]byte)

    slice := make([]byte, 5)
    log.Printf("slice: %v, len: %d, cap: %d", slice, len(slice), cap(slice)\
)

    slice2 := make([]byte, 0, 10)
    log.Printf("slice: %v, len: %d, cap: %d", slice2, len(slice2), cap(slic\
e2))
}
```

new

The new function allocates a new object of the type provided, and returns a pointer to the new object. The object is allocated to be the zero value for the given type. It's not something you use terribly often, but it can be useful. If you're making a new struct, you probably want to use the composite literal syntax instead.

builtin/new.go

```
package main

import "log"

type Actor struct {
    Name string
}

type Movie struct {
    Title  string
    Actors []*Actor
}

func main() {
    ip := new(int)
    log.Printf("ip type: %T, ip: %v, *ip: %v", ip, ip, *ip)

    m := new(Movie)
    log.Printf("m type: %T, m: %v, *m: %v", m, m, *m)
}
```

Maps, Slices, And Channels

You've got slices, maps and channels as some of the fundamental types that Go provides. The functions `delete`, `close`, `append`, and `copy` all deal with these types to do basic operations.

delete

`delete` removes elements from a map. If the key doesn't exist in the map, nothing happens, nothing to worry about. If the map is `nil`, it panics, so make sure you're passing a valid map.

builtin/delete.go

```go
package main

import "log"

func main() {
    m := make(map[string]int)
    log.Println(m)

    m["one"] = 1
    log.Println(m)

    m["two"] = 2
    log.Println(m)

    delete(m, "one")
    log.Println(m)

    delete(m, "one")
    log.Println(m)

    m = nil
    delete(m, "two") // Will panic
}
```

close

`close` takes a writable channel and closes it. When I say writable, I mean either a *normal* channel like `var normal chan int` or a *write only* channel like `var writeOnly chan<- int`. You can still receive from a closed channel, but you'll get the *zero value* of whatever the type is. If you want to check that you actually got a value and not the zero value, use the *comma ok* pattern. Closing an already closed channel will panic, so watch those double closes.

builtin/close.go

```go
package main

import "log"

func main() {
    c := make(chan int, 1)
    c <- 1

    log.Println(<-c) // Prints 1

    c <- 2
    close(c)

    log.Println(<-c) // Prints 2
    log.Println(<-c) // Prints 0

    if i, ok := <-c; ok {
        log.Printf("Channel is open, got %d", i)
    } else {
        log.Printf("Channel is closed, got %d", i)
    }

    close(c) // Panics, channel is already closed
}
```

append

append tacks on elements to the end of a slice, exactly like it sounds. You need to keep the return value around, since it's the new slice with the extra data. It could return the same slice if it has space for the data, but it might return something new if it needed to allocate more memory. It takes a variable number of arguments, so if you want to append an existing array, use `...` to expand the array.

The idiomatic way to append to a slice is to assign the result to the same slice you're appending to. It's probably what you want.

builtin/append.go

```go
package main

import "log"

func main() {
    // Empty slice, with capacity of 10
    ints := make([]int, 0, 10)
    log.Printf("ints: %v", ints)

    ints2 := append(ints, 1, 2, 3)

    log.Printf("ints2: %v", ints2)
    log.Printf("Slice was at %p, it's probably still at %p", ints, ints2)

    moreInts := []int{4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14}
    ints3 := append(ints2, moreInts...)

    log.Printf("ints3: %v", ints3)
    log.Printf("Slice was at %p, and it moved to %p", ints2, ints3)

    ints4 := []int{1, 2, 3}
    log.Printf("ints4: %v", ints4)
    // The idiomatic way to append to a slice,
    // just assign to the same variable again
    ints4 = append(ints4, 4, 5, 6)
    log.Printf("ints4: %v", ints4)
}
```

copy

`copy` copies from one slice to another. It will also copy *from* a string, treating it as a slice of bytes. It returns the number of bytes copied, which is the shorter of the lengths of the two slices.

builtin/copy.go

```
package main

import "log"

func main() {
    ints := []int{1, 2, 3, 4, 5, 6}
    otherInts := []int{11, 12, 13, 14, 15, 16}

    log.Printf("ints: %v", ints)
    log.Printf("otherInts: %v", otherInts)

    copied := copy(ints[:3], otherInts)
    log.Printf("Copied %d ints from otherInts to ints", copied)

    log.Printf("ints: %v", ints)
    log.Printf("otherInts: %v", otherInts)

    hello := "Hello, World!"
    bytes := make([]byte, len(hello))

    copy(bytes, hello)

    log.Printf("bytes: %v", bytes)
    log.Printf("hello: %s", hello)
}
```

All The Sizes

A lot of things have lengths and capacities. With `len` and `cap`, you can find out about these values.

len

`len` tells you the actual *length* or size of something. In the case of slices, you get, well, the length. In the case of strings, you get the number of bytes. For maps, you get how many pairs are in the map. For channels, you get how many elements the channel has buffered (only relevant for buffered channels).

You can also call `len` with a pointer, but only a pointer to an array. It's the equivalent of calling it on the dereferenced pointer. But, since it still has a type, it's an *array* and not a *slice*, and the type

of an array includes the size, so it still works. The length is part of the type.

builtin/len.go

```
package main

import "log"

func main() {
    slice := make([]byte, 10)
    log.Printf("slice: %d", len(slice))

    str := "γειά σου κόσμε"
    log.Printf("string: %d", len(str))

    m := make(map[string]int)
    m["hello"] = 1
    log.Printf("map: %d", len(m))

    channel := make(chan int, 5)
    log.Printf("channel: %d", len(channel))
    channel <- 1
    log.Printf("channel: %d", len(channel))

    var pointer *[5]byte
    log.Printf("pointer: %d", len(pointer))
}
```

cap

`cap` tells you the capacity of something. It's similar to `len`, except it doesn't work on maps or strings. With arrays, it's the same as using `len`. With slices, it returns the max size the slice can grow to when you append to it. With channels, it returns the buffer capacity.

builtin/cap.go

```
package main

import "log"

func main() {
    slice := make([]byte, 0, 5)
    log.Printf("slice: %d", cap(slice))

    channel := make(chan int, 10)
    log.Printf("channel: %d", cap(channel))

    var pointer *[15]byte
    log.Printf("pointer: %d == %d", cap(pointer), len(pointer))
}
```

Causing And Handling Panics

`panic` and `recover` are typically used to deal with errors. These are errors where returning an error in the *comma err* style don't make sense. Things like programmer error or things that are seriously broken. *Usually.*

If bad things are afoot, you can use `panic` to throw an error. You can pass it pretty much any object, which gets carried up the stack. Deferred functions get executed, and up the error goes. It works sort of like `raise` or `throw` in other languages.

You can use `recover` to, as the name says, recover from a panic. `recover` must be excuted from *within* a deferred function, and not from within a function the deferred function calls. It returns whatever panic was called with, you check for `nil` and can then type cast it to something.

There are some creative uses[a] for `panic/recover` beyond error handling, but they should be confined to your own package. In Go, it's not nice to let a panic go outside your own little world. Better to handle the panic yourself in a way you know how, and return an appropriate error. In some cases, the panic makes sense. Err on the side of returning instead of panicking.

[a] See the code for the `encoding/json` package on one of them.

The example illustrates things much better.

builtin/panic_recover.go

```go
package main

import (
    "errors"
    "log"
)

func handlePanic(f func()) {
    defer func() {
        if r := recover(); r != nil {
            if str, ok := r.(string); ok {
                log.Printf("got a string error: %s", str)
                return
            }

            if err, ok := r.(error); ok {
                log.Printf("got an error error: %s", err.Error())
                return
            }

            log.Printf("got a different kind of error: %v", r)
        }
    }()
    f()
}

func main() {
    handlePanic(func() {
        panic("string error")
    })

    handlePanic(func() {
        panic(errors.New("error error"))
    })

    handlePanic(func() {
        panic(10)
    })
}
```

Complex Numbers

Go supports complex numbers as a builtin type. You can define them with literal syntax, or by using the builtin function `complex`. If you want to build a complex number from existing float values, you need to use the builtin function, and the two arguments have to be of the same type (`float32` or `float64`) and will produce a complex type double the size (`complex64` or `complex128`). Once you have a complex number, you can add, subtract, divide, and multiply values normally.

If you have a complex number and want to break it into the real and imaginary parts, use the functions `real` and `imag`.

builtin/complex.go

```
package main

import "log"

func main() {
    c1 := 1.5 + 0.5i
    c2 := complex(1.5, 0.5)
    log.Printf("c1: %v", c1)
    log.Printf("c2: %v", c2)
    log.Printf("c1 == c2: %v", c1 == c2)
    log.Printf("c1 real: %v", real(c1))
    log.Printf("c1 imag: %v", imag(c1))
    log.Printf("c1 + c2: %v", c1+c2)
    log.Printf("c1 - c2: %v", c1-c2)
    log.Printf("c1 * c2: %v", c1*c2)
    log.Printf("c1 / c2: %v", c1/c2)
    log.Printf("c1 type: %T", c1)

    c3 := complex(float32(1.5), float32(0.5))
    log.Printf("c3 type: %T", c3)
}
```

expvar

The `expvar` package is global variables done right.

It has helpers for `Float`, `Int`, `Map`, and `String` types, which are setup to be atomic. Things are registered by a string name, the `Key`, and they map to a corresponding `Var`, which is just an interface with a single method: `String() string`.

This simple interface allows you to use the more raw `Publish` method to register more custom handlers in the form of a `Func` type. These are just functions which take no arguments and return an empty interface (which, in implementation should probably be a string).

Examining the source for the package, you can see it uses this to register the memstats variable. When you iterate through the variables and you call the `String` method on the `Var`, the function runs to extract the memstats at that moment in time.

It's a pretty simple, but very powerful package. You can use it for metric type stuff, or you can use it as a more traditional global variable system. It can do it all.

expvar/expvar.go

```
package main

import (
    "expvar"
    "flag"
    "log"
    "time"
)

var (
    times      = flag.Int("times", 1, "times to say hello")
    name       = flag.String("name", "World", "thing to say hello to")
    helloTimes = expvar.NewInt("hello")
)

func init() {
    expvar.Publish("time", expvar.Func(now))
}

func now() interface{} {
    return time.Now().Format(time.RFC3339Nano)
}
```

```
func hello(times int, name string) {
    helloTimes.Add(int64(times))
    for i := 0; i < times; i++ {
        log.Printf("Hello, %s!", name)
    }
}

func printVars() {
    log.Println("expvars:")
    expvar.Do(func(kv expvar.KeyValue) {
        switch kv.Key {
        case "memstats":
            // Do nothing, this is a big output.
        default:
            log.Printf("\t%s -> %s", kv.Key, kv.Value)
        }
    })
}

func main() {
    flag.Parse()
    printVars()
    hello(*times, *name)
    printVars()
    hello(*times, *name)
    printVars()
}
```