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 2020-06-28

Leanpub

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:

#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 problems.

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 know about them. *It makes no sense for my application to 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.

[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/

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 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 doesn't quite work reading it 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 for download from Leanpub as an extra. Visit your dashboard[8] for access to the archives.

Anything with a main package should be able to be executed with `go run` by Go Version 1.2. 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 writing 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!

License

Code distributed as part of this book, either inline or with the above linked archive, is licensed under the MIT license:

[7] http://golang.org/pkg/
[8] https://leanpub.com/dashboard

LICENSE

```
Copyright (c) 2014 Daniel Huckstep

Permission is hereby granted, free of charge, to any person obtaining a copy of this\
 software and associated documentation files (the "Software"), to deal in the Softwa\
re without restriction, including without limitation the rights to use, copy, modify\
, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to\
 permit persons to whom the Software is furnished to do so, subject to the following\
 conditions:

The above copyright notice and this permission notice shall be included in all copie\
s or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, \
INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTIC\
ULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS\
 BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRA\
CT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR TH\
E USE OR OTHER DEALINGS IN THE SOFTWARE.
```

Some code is taken directly from the Go source distribution. This code is licensed under a BSD-style license by The Go Authors:

GOLICENSE

```
Copyright (c) 2012 The Go Authors. All rights reserved.

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions are
met:

   * Redistributions of source code must retain the above copyright
notice, this list of conditions and the following disclaimer.
   * Redistributions in binary form must reproduce the above
copyright notice, this list of conditions and the following disclaimer
in the documentation and/or other materials provided with the
distribution.
   * Neither the name of Google Inc. nor the names of its
contributors may be used to endorse or promote products derived from
this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
"AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
```

```
LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR
A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT
OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL,
SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT
LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE,
DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY
THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT
(INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE
OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
```

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
- Kim Shrier

Credits

[9] http://creativecommons.org/licenses/by-sa/2.0/deed.en

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

```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

```go
package main

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

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

```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 {
		msg := "failed writing tar header for %s: %s"
		return fmt.Errorf(msg, 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() {
		msg := "wrote %d bytes of %s, expected to write %d"
		return fmt.Errorf(msg, copied, filename, stat.Size())
	}

	return nil
}

func main() {
	flags := os.O_WRONLY | os.O_CREATE | os.O_TRUNC
	file, err := os.OpenFile("go.tar", flags, 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 tar writer first, followed by the original `io.Writer`. In the example, I `defer` the calls to `Close`. Because `defer` executes in a LIFO[a] order, this is exactly the order things get closed in. `defer` usually results in you not having to think too hard in these situations, just use `defer` the way it should be used, and everything should be fine.

[a]Last In First Out

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

```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 // deprecated; use CompressedSize64
	UncompressedSize   uint32 // deprecated; use UncompressedSize64
	CompressedSize64   uint64
	UncompressedSize64 uint64
	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

```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)
```

```go
	if err != nil {
		return fmt.Errorf("failed opening %s: %s", filename, err)
	}
	defer file.Close()

	wr, err := zw.Create(filename)
	if err != nil {
		msg := "failed creating entry for %s in zip file: %s"
		return fmt.Errorf(msg, filename, 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() {
	flags := os.O_WRONLY | os.O_CREATE | os.O_TRUNC
	file, err := os.OpenFile("go.zip", flags, 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 an `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

```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 {
		msg := "failed opening archive, run `go run write_tar.go` first: %s"
		log.Fatalf(msg, 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)
    }
}
```

Output:

```
tar header
Name:       write_tar.go
Mode:       644
UID:        0
GID:        0
Size:       1441
ModTime:    2014-03-07 23:02:17 -0700 MST
Typeflag:   '\x00'
Linkname:
Uname:
Gname:
Devmajor:   0
Devminor:   0
AccessTime: 0001-01-01 00:00:00 +0000 UTC
ChangeTime: 0001-01-01 00:00:00 +0000 UTC
Contents:

<snip contents of writer_tar.go>
tar header
Name:       read_tar.go
Mode:       644
UID:        0
GID:        0
Size:       1484
ModTime:    2014-03-07 23:00:03 -0700 MST
Typeflag:   '\x00'
Linkname:
Uname:
Gname:
Devmajor:   0
Devminor:   0
AccessTime: 0001-01-01 00:00:00 +0000 UTC
```

```
ChangeTime: 0001-01-01 00:00:00 +0000 UTC
Contents:

<snip contents of read_tar.go>
```

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

```go
package main

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

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

	if uint64(copied) != file.UncompressedSize64 {
```

```go
			msg := "read %d bytes of %s but expected to read %d bytes"
			return fmt.Errorf(msg, copied, file.UncompressedSize64)
		}

	fmt.Println()

	return nil
}

func main() {
	rc, err := zip.OpenReader("go.zip")
	if err != nil {
		msg := "failed opening archive, run `go run write_zip.go` first: %s"
		log.Fatalf(msg, 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)
		}
	}
}
```

Output:

```
Contents of write_zip.go:
<snip contents of write_zip.go>

Contents of read_zip.go:
<snip contents of read_zip.go>
```

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

Caveats

ZIP64

You may have noticed the `FileHeader` has two pairs of numbers for the size of a file in the archive. The `CompressedSize` and `UncompressedSize` are `uint32` values. These are deprecated, but in the interest of backwards compatibility will still work for regular zip files. If you're working with ZIP64 files, you need to use the newer `CompressedSize64` and `UncompressedSize64` `uint64` values. These will be correct for all files, so they are the preferred values to use.

bufio

Is That A Buffer In Your Pocket?

The `bufio` package pairs up with the `io.Reader` and `io.Writer` interfaces to make life a little faster by including a buffer. Buffered IO. The speed up comes from the fact that when you call `Write` on a buffered IO thing, it doesn't necessarily write the data. It might just store it in the buffer, and then when the buffer is full, it can write it out in one big chunk, reducing the number of **system calls**. System calls involve going from user space to kernel space, so they're kind of slow.

Buffered IO is preferable to *regular* IO for the increased speed, and the ability to peek at and push back (some) data, but it has drawbacks too. The bufer takes up memory (default of 4KB), which is the main kicker. Sometimes, you just can't afford that buffer size. The data is not always written right away either. Sometimes you need it to be written immediately, and in those cases, unbuffered is the way to go. In other situations, you could used buffered, but `Flush` on a regular basis.

With regards to speed, let's look at a little benchmark. Run this with `go test -test.bench '.*'`

bufio/bench/bufio_test.go

```
package main

import (
	"bufio"
	"io"
	"log"
	"os"
	"testing"
)

const str = "Go, The Standard Library"
const Times = 100

func openFile(name string) *os.File {
	file, err := os.OpenFile(name, os.O_WRONLY|os.O_CREATE|os.O_TRUNC, 0644)
	if err != nil {
		log.Fatalf("failed opening %s for writing: %s", name, err)
```

```go
	}
	return file
}

func BenchmarkBufio(b *testing.B) {
	file := openFile(os.DevNull)
	defer file.Close()

	bufferedFile := bufio.NewWriter(file)

	for i := 0; i < b.N; i++ {
		if _, err := bufferedFile.WriteString(str); err != nil {
			log.Fatalf("failed or short write: %s", err)
		}
	}

	bufferedFile.Flush()
}

func BenchmarkIO(b *testing.B) {
	file := openFile(os.DevNull)
	defer file.Close()

	for i := 0; i < b.N; i++ {
		if _, err := io.WriteString(file, str); err != nil {
			log.Fatalf("failed or short write: %s", err)
		}
	}
}
```

On my machine I was getting about 50 nanoseconds per operation for buffered and a whopping 1260 nanoseconds per operation for unbuffered. If you can spare the memory, you probably want buffered IO.

Reading

Using `bufio` to read and write things looks just like anything else from the outside, but the `Reader` and `Writer` types have some handy extra methods on them. When it comes to reading, you can read strings and runes. You can also *unread* individual bytes (only the last read byte) and individual runes (only after a call to `ReadRune`). You can read entire lines too. If you don't want to read just yet, you can `Peek`.

Use the `bufio.NewReader` function to wrap you existing `io.Reader` interface to get back your buffered io type.

bufio/reading.go

```go
package main

import (
	"bufio"
	"log"
	"os"
)

func init() {
	log.SetFlags(0)
	log.SetPrefix("» ")
}

func openFile(name string) *os.File {
	file, err := os.Open(name)
	if err != nil {
		log.Fatalf("failed opening %s for writing: %s", name, err)
	}
	return file
}

func doPeek(r *bufio.Reader) {
	normal := 4
	huge := 5000

	bytes, err := r.Peek(normal)
	if err != nil {
		log.Fatalf("Failed peeking: %s", err)
	}
	log.Printf("Peeked at the reader, saw: %s", bytes)

	_, err = r.Peek(huge)
	if err != nil {
		log.Printf("Failed peeking at %d bytes: %s", huge, err)
	}
}

func doStringRead(r *bufio.Reader) {
	word, err := r.ReadString(' ')
```

```
	if err != nil {
		log.Fatalf("failed reading string: %s", err)
	}
	log.Printf("Got first word: %s", word)
}

func doRuneRead(r *bufio.Reader) {
	ru, size, err := r.ReadRune()
	if err != nil {
		log.Fatalf("failed reading rune: %s", err)
	}
	log.Printf("Got rune %U of size %d (it looks like %q in Go)", ru, size, ru)

	log.Printf("Didn't mean to read that though, putting it back")
	err = r.UnreadRune()
	if err != nil {
		log.Fatalf("failed unreading a rune: %s", err)
	}
}

func doByteRead(r *bufio.Reader) {
	b, err := r.ReadByte()
	if err != nil {
		log.Fatalf("failed reading a byte: %s", err)
	}
	log.Printf("Read a byte: %x", b)

	log.Printf("Didn't mean to read that either, putting it back")
	err = r.UnreadByte()
	if err != nil {
		log.Fatalf("failed urneading a byte: %s", err)
	}
}

func doLineRead(r *bufio.Reader) {
	line, prefix, err := r.ReadLine()
	if err != nil {
		log.Fatalf("failed reading a line: %s", err)
	}
	log.Printf("Got the rest of the line: %s", line)

	if prefix {
		log.Printf("Line too big for buffer, only first %d bytes returned", len(line))
```

```
	} else {
		log.Printf("Line fit in buffer, full line returned")
	}

	log.Printf("After all that, %d bytes are buffered", r.Buffered())
}

func main() {
	file := openFile("reading.go")
	defer file.Close()

	br := bufio.NewReader(file)

	doPeek(br)
	doStringRead(br)
	doRuneRead(br)
	doByteRead(br)
	doLineRead(br)
}
```

Output:

```
» Peeked at the reader, saw: pack
» Failed peeking at 5000 bytes: bufio: buffer full
» Got first word: package
» Got rune U+006D of size 1 (it looks like 'm' in Go)
» Didn't mean to read that though, putting it back
» Read a byte: 6d
» Didn't mean to read that either, putting it back
» Got the rest of the line: main
» Line fit in buffer, full line returned
» After all that, 2023 bytes are buffered
```

Writing

On the writing side, you can write individual bytes, runes, and strings. Similar to reading, use `bufio.NewWriter` to wrap an `io.Writer` and go to town.

bufio/writing.go

```go
package main

import (
	"bufio"
	"log"
	"os"
)

func init() {
	log.SetFlags(0)
	log.SetPrefix("» ")
}

func openFile(name string) *os.File {
	file, err := os.OpenFile(name, os.O_WRONLY|os.O_CREATE|os.O_TRUNC, 0644)
	if err != nil {
		log.Fatalf("failed opening %s for writing: %s", name, err)
	}
	return file
}

func doWriteByte(w *bufio.Writer) {
	if err := w.WriteByte('G'); err != nil {
		log.Fatalf("failed writing a byte: %s", err)
	}
}

func doWriteRune(w *bufio.Writer) {
	if written, err := w.WriteRune(rune('o')); err != nil {
		log.Fatalf("failed writing a rune: %s", err)
	} else {
		log.Printf("Wrote rune in %d bytes", written)
	}
}

func doWriteString(w *bufio.Writer) {
	written, err := w.WriteString(", The Standard Library\n")
	if err != nil {
		log.Fatalf("failed writing string: %s", err)
	}
	log.Printf("Wrote string in %d bytes", written)
```

```go
}

func main() {
	file := openFile("bufio.out")
	defer file.Close()

	bw := bufio.NewWriter(file)

	// Remember to Flush!
	defer bw.Flush()

	doWriteByte(bw)
	doWriteRune(bw)
	doWriteString(bw)
}
```

Output:

```
» Wrote rune in 1 bytes
» Wrote string in 23 bytes
```

It's all pretty straight forward stuff. Wrap it, write it and read it!

Scanning

In Go 1.1, the `Scanner` type was added to the `bufio` package. It provides a simple interface to read chunks of things. By default it will read lines (excluding the terminator), but has support for custom split functions. It includes split functions to scan individual bytes, words (split on spaces), and runes. We'll look at the fun ones.

bufio/scanning.go

```go
package main

import (
	"bufio"
	"log"
	"os"
	"strings"
	"unicode/utf8"
)
```

```go

func init() {
	log.SetFlags(0)
	log.SetPrefix("» ")
}

func lines() {
	f, _ := os.Open("scanning.go")
	defer f.Close()
	s := bufio.NewScanner(f)
	for s.Scan() {
		log.Printf("line: %s", s.Text())
	}
}

func words() {
	r := strings.NewReader("I just wanna dance with somebody")
	s := bufio.NewScanner(r)
	s.Split(bufio.ScanWords)
	for s.Scan() {
		log.Printf("word: %s", s.Text())
	}
}

func runes() {
	r := strings.NewReader("I just wanna dance with somebody")
	s := bufio.NewScanner(r)
	s.Split(bufio.ScanRunes)
	for s.Scan() {
		log.Printf("rune: %s", s.Text())
	}
}

// Basically the `ScanWords` code, altered to split on periods.
func periods(data []byte, atEOF bool) (int, []byte, error) {
	start := 0
	for width := 0; start < len(data); start += width {
		var r rune
		r, width = utf8.DecodeRune(data[start:])
		if r != '.' {
			break
		}
	}
```

```go
	if atEOF && len(data) == 0 {
		return 0, nil, nil
	}
	for width, i := 0, start; i < len(data); i += width {
		var r rune
		r, width = utf8.DecodeRune(data[i:])
		if r == '.' {
			return i + width, data[start:i], nil
		}
	}
	return 0, nil, nil
}

func custom() {
	f, _ := os.Open("scanning.go")
	defer f.Close()
	s := bufio.NewScanner(f)
	s.Split(periods)
	for s.Scan() {
		log.Printf("between periods: %s", s.Text())
	}
}

func main() {
	lines()
	words()
	runes()
	custom()
}
```

Output:

```
» line: package main
» line: 
» line: import (
» line: 	"bufio"
» line: 	"log"
» line: 	"os"
» line: 	"strings"
» line: 	"unicode/utf8"
» line: )
» line: 
» line: func init() {
```

```
» line: 	log.SetFlags(0)
» line: 	log.SetPrefix("» ")
» line: }
» line:
» line: func lines() {
» line: 	f, _ := os.Open("scanning.go")
» line: 	defer f.Close()
» line: 	s := bufio.NewScanner(f)
» line: 	for s.Scan() {
» line: 		log.Printf("line: %s", s.Text())
» line: 	}
» line: }
» line:
» line: func words() {
» line: 	r := strings.NewReader("I just wanna dance with somebody")
» line: 	s := bufio.NewScanner(r)
» line: 	s.Split(bufio.ScanWords)
» line: 	for s.Scan() {
» line: 		log.Printf("word: %s", s.Text())
» line: 	}
» line: }
» line:
» line: func runes() {
» line: 	r := strings.NewReader("I just wanna dance with somebody")
» line: 	s := bufio.NewScanner(r)
» line: 	s.Split(bufio.ScanRunes)
» line: 	for s.Scan() {
» line: 		log.Printf("rune: %s", s.Text())
» line: 	}
» line: }
» line:
» line: // Basically the `ScanWords` code, altered to split on periods.
» line: func periods(data []byte, atEOF bool) (int, []byte, error) {
» line: 	start := 0
» line: 	for width := 0; start < len(data); start += width {
» line: 		var r rune
» line: 		r, width = utf8.DecodeRune(data[start:])
» line: 		if r != '.' {
» line: 			break
» line: 		}
» line: 	}
» line: 	if atEOF && len(data) == 0 {
» line: 		return 0, nil, nil
```

```
» line:         }
» line:         for width, i := 0, start; i < len(data); i += width {
» line:                 var r rune
» line:                 r, width = utf8.DecodeRune(data[i:])
» line:                 if r == '.' {
» line:                         return i + width, data[start:i], nil
» line:                 }
» line:         }
» line:         return 0, nil, nil
» line: }
» line:
» line: func custom() {
» line:         f, _ := os.Open("scanning.go")
» line:         defer f.Close()
» line:         s := bufio.NewScanner(f)
» line:         s.Split(periods)
» line:         for s.Scan() {
» line:                 log.Printf("between periods: %s", s.Text())
» line:         }
» line: }
» line:
» line: func main() {
» line:         lines()
» line:         words()
» line:         runes()
» line:         custom()
» line: }
» word: I
» word: just
» word: wanna
» word: dance
» word: with
» word: somebody
» rune: I
» rune:
» rune: j
» rune: u
» rune: s
» rune: t
» rune:
» rune: w
» rune: a
» rune: n
```

```
» rune: n
» rune: a
» rune:  
» rune: d
» rune: a
» rune: n
» rune: c
» rune: e
» rune:  
» rune: w
» rune: i
» rune: t
» rune: h
» rune:  
» rune: s
» rune: o
» rune: m
» rune: e
» rune: b
» rune: o
» rune: d
» rune: y
» between periods: package main

import (
        "bufio"
        "log"
        "os"
        "strings"
        "unicode/utf8"
)

func init() {
        log
» between periods: SetFlags(0)
        log
» between periods: SetPrefix("» ")
}

func lines() {
        f, _ := os
» between periods: Open("scanning
» between periods: go")
```

```
        defer f
» between periods: Close()
        s := bufio
» between periods: NewScanner(f)
        for s
» between periods: Scan() {
                log
» between periods: Printf("line: %s", s
» between periods: Text())
        }
}

func words() {
        r := strings
» between periods: NewReader("I just wanna dance with somebody")
        s := bufio
» between periods: NewScanner(r)
        s
» between periods: Split(bufio
» between periods: ScanWords)
        for s
» between periods: Scan() {
                log
» between periods: Printf("word: %s", s
» between periods: Text())
        }
}

func runes() {
        r := strings
» between periods: NewReader("I just wanna dance with somebody")
        s := bufio
» between periods: NewScanner(r)
        s
» between periods: Split(bufio
» between periods: ScanRunes)
        for s
» between periods: Scan() {
                log
» between periods: Printf("rune: %s", s
» between periods: Text())
        }
}
```

```

// Basically the `ScanWords` code, altered to split on periods
» between periods:
func periods(data []byte, atEOF bool) (int, []byte, error) {
	start := 0
	for width := 0; start < len(data); start += width {
		var r rune
		r, width = utf8
» between periods: DecodeRune(data[start:])
		if r != '
» between periods: ' {
			break
		}
	}
	if atEOF && len(data) == 0 {
		return 0, nil, nil
	}
	for width, i := 0, start; i < len(data); i += width {
		var r rune
		r, width = utf8
» between periods: DecodeRune(data[i:])
		if r == '
» between periods: ' {
			return i + width, data[start:i], nil
		}
	}
	return 0, nil, nil
}

func custom() {
	f, _ := os
» between periods: Open("scanning
» between periods: go")
	defer f
» between periods: Close()
	s := bufio
» between periods: NewScanner(f)
	s
» between periods: Split(periods)
	for s
» between periods: Scan() {
		log
» between periods: Printf("between periods: %s", s
```

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.[10] 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.

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

builtin/make.go

```go
package main

import "log"

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

        buffered := make(chan int, 10)
        log.Printf("buffered: %v, type: %T, len: %d, cap: %d", buffered, buffered, len(buff\
ered), 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(slice2))
}
```

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

```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 itself is `nil` it still works, just nothing happens.

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")
}
```

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

```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 before things are copied to a new backing array. This is why you have to save the return value of `append`. If `cap` returns 5 and you append 6 things to your slice, it's going to return you a slice backed by a new array.

With channels, it returns the buffer capacity.

builtin/cap.go

```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[11] 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.

The example illustrates things much better.

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

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

```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)
}
```

bytes

Bits and Bytes and Everything Nice

The `bytes` package deals with, you guessed it, bytes. More specifically byte slices, `[]byte`. You can do quite a bit with just a byte slice. You can compare and search them. If they aren't to your liking, you can change them. Splitting and joining them is simple stuff. You can change the case of the contents, making it upper or lowercase. Trimming contents from either end is also straightforward.

With the `Buffer` type, you can do some pretty sweet things too, like write anything to memory (and get a string out of it).

The `Reader` type lets you operate on a byte slice like various `io` package interfaces.

Comparison

Comparison of byte slices is pretty simple. `Compare` gives you the industry standard of -1/0/1 to denote less than/equal/greater than. `Equal` gives you a bool and checks for a simple byte for byte equality. `EqualFold` checks equality but ignores case. It's slightly more complicated than just *ignoring case* but that's the basic idea.

bytes/comparison.go

```go
package main

import (
	"bytes"
	"log"
)

func DemoCompare(a, b []byte) {
	if c := bytes.Compare(a, b); c == -1 {
		log.Printf("%s is less than %s", a, b)
	} else if c == 1 {
		log.Printf("%s is greater than %s", a, b)
	} else {
		log.Printf("%s and %s are equal", a, b)
```

```go
	}
}

func DemoEqual(a, b []byte) {
	if bytes.Equal(a, b) {
		log.Printf("%s and %s are equal", a, b)
	} else {
		log.Printf("%s and %s are NOT equal", a, b)
	}
}

func DemoEqualFold(a, b []byte) {
	if bytes.EqualFold(a, b) {
		log.Printf("%s and %s are equal", a, b)
	} else {
		log.Printf("%s and %s are NOT equal", a, b)
	}
}

func main() {
	golang := []byte("golang")
	gOlaNg := []byte("gOlaNg")
	haskell := []byte("haskell")

	DemoCompare(golang, golang)
	DemoCompare(golang, haskell)
	DemoCompare(haskell, golang)

	DemoEqual(golang, golang)
	DemoEqual(golang, haskell)

	DemoEqualFold(golang, gOlaNg)
	DemoEqualFold(golang, golang)
}
```

Output:

```
2014/08/21 18:02:13 golang and golang are equal
2014/08/21 18:02:13 golang is less than haskell
2014/08/21 18:02:13 haskell is greater than golang
2014/08/21 18:02:13 golang and golang are equal
2014/08/21 18:02:13 golang and haskell are NOT equal
2014/08/21 18:02:13 golang and gOlaNg are equal
2014/08/21 18:02:13 golang and golang are equal
```

Searching

If you're got a slice full of stuff, you probably want to search it. Luckily, the `bytes` package has everything you need. If you don't want to deal with raw bytes, there is probably some way of converting your slice of whatever to a slice of bytes. We'll see this a lot in the example below, in the form of the builtin type conversion going from a string to a slice of bytes.

bytes/searching.go

```go
package main

import (
	"bytes"
	"log"
)

func contains(s, sub []byte) {
	if bytes.Contains(s, sub) {
		log.Printf("%s contains %s", s, sub)
	} else {
		log.Printf("%s does NOT contain %s", s, sub)
	}
}

func count(s, sep []byte) {
	log.Printf("%s contains %d instance(s) of %s", s, bytes.Count(s, sep), sep)
}

func hasPrefix(s, prefix []byte) {
	if bytes.HasPrefix(s, prefix) {
```

```
			log.Printf("%s has the prefix %s", s, prefix)
		} else {
			log.Printf("%s does NOT have the prefix %s", s, prefix)
		}
}

func hasSuffix(s, suffix []byte) {
		if bytes.HasSuffix(s, suffix) {
			log.Printf("%s has the suffix %s", s, suffix)
		} else {
			log.Printf("%s does NOT have the suffix %s", s, suffix)
		}
}

func index(s, sep []byte) {
		if i := bytes.Index(s, sep); i == -1 {
			log.Printf("%s does NOT appear in %s", sep, s)
		} else {
			log.Printf("%s appears at index %d in %s", sep, i, s)
		}
}

func indexAny(s []byte, chars string) {
		if i := bytes.IndexAny(s, chars); i == -1 {
			log.Printf("No unicode characters in %q appear in %s", chars, s)
		} else {
			log.Printf("A unicode character in %q appears at index %d in %s", chars, i, s)
		}
}

func indexByte(s []byte, b byte) {
		if i := bytes.IndexByte(s, b); i == -1 {
			log.Printf("%q does NOT appear in %s", b, s)
		} else {
			log.Printf("%q appears at index %d in %s", b, i, s)
		}
}

func indexFunc(s []byte, f func(rune) bool) {
		if i := bytes.IndexFunc(s, f); i == -1 {
			log.Printf("Something controlled by %#v does NOT appear in %s", f, s)
		} else {
			log.Printf("Something controlled by %#v appears at index %d in %s", f, i, s)
```

```go
	}
}

func indexRune(s []byte, r rune) {
	if i := bytes.IndexRune(s, r); i == -1 {
		log.Printf("Rune %d does NOT appear in %s", r, s)
	} else {
		log.Printf("Rune %d appears at index %d in %s", r, i, s)
	}
}

func lastIndex(s, sep []byte) {
	if i := bytes.LastIndex(s, sep); i == -1 {
		log.Printf("%s does NOT appear in %s", sep, s)
	} else {
		log.Printf("%s appears last at index %d in %s", sep, i, s)
	}
}

func lastIndexAny(s []byte, chars string) {
	if i := bytes.LastIndexAny(s, chars); i == -1 {
		log.Printf("No unicode characters in %q appear in %s", chars, s)
	} else {
		log.Printf("A unicode character in %q appears last at index %d in %s", chars, i, s)
	}
}

func lastIndexFunc(s []byte, f func(rune) bool) {
	if i := bytes.LastIndexFunc(s, f); i == -1 {
		log.Printf("Something controlled by %#v does NOT appear in %s", f, s)
	} else {
		log.Printf("Something controlled by %#v appears at index %d in %s", f, i, s)
	}
}

func main() {
	golang := []byte("golang")
	haskell := []byte("haskell")
	lang := []byte("lang")
	gos := []byte("go")

	contains(golang, lang)
	contains(golang, haskell)
```

```

	count(golang, lang)
	count(haskell, []byte("l"))

	hasPrefix(golang, gos)
	hasPrefix(haskell, gos)

	hasSuffix(golang, lang)
	hasSuffix(haskell, lang)

	index(golang, lang)
	index(golang, gos)
	index(haskell, lang)

	indexAny(golang, "lang")
	indexAny(haskell, "lang")
	indexAny(haskell, "go")

	indexByte(golang, 'h')
	indexByte(golang, 'l')
	indexByte(haskell, 'l')

	g := rune('g')
	indexFunc(golang, func(r rune) bool { return r == g })
	indexFunc(haskell, func(r rune) bool { return r == g })

	indexRune(golang, rune('o'))
	indexRune(haskell, rune('l'))

	lastIndex(golang, []byte("g"))
	lastIndex(haskell, []byte("l"))

	lastIndexAny(golang, "abcdefg")
	lastIndexAny(haskell, "lmnop")

	lastIndexFunc(golang, func(r rune) bool { return r == g })
	lastIndexFunc(haskell, func(r rune) bool { return r == g })
}
```

Output:

```
2014/08/21 18:02:14 golang contains lang
2014/08/21 18:02:14 golang does NOT contain haskell
2014/08/21 18:02:14 golang contains 1 instance(s) of lang
2014/08/21 18:02:14 haskell contains 2 instance(s) of l
2014/08/21 18:02:14 golang has the prefix go
2014/08/21 18:02:14 haskell does NOT have the prefix go
2014/08/21 18:02:14 golang has the suffix lang
2014/08/21 18:02:14 haskell does NOT have the suffix lang
2014/08/21 18:02:14 lang appears at index 2 in golang
2014/08/21 18:02:14 go appears at index 0 in golang
2014/08/21 18:02:14 lang does NOT appear in haskell
2014/08/21 18:02:14 A unicode character in "lang" appears at index 0 in golang
2014/08/21 18:02:14 A unicode character in "lang" appears at index 1 in haskell
2014/08/21 18:02:14 No unicode characters in "go" appear in haskell
2014/08/21 18:02:14 'h' does NOT appear in golang
2014/08/21 18:02:14 'l' appears at index 2 in golang
2014/08/21 18:02:14 'l' appears at index 5 in haskell
2014/08/21 18:02:14 Something controlled by (func(int32) bool)(0x4680) appears at in\
dex 0 in golang
2014/08/21 18:02:14 Something controlled by (func(int32) bool)(0x46a0) does NOT appe\
ar in haskell
2014/08/21 18:02:14 Rune 111 appears at index 1 in golang
2014/08/21 18:02:14 Rune 108 appears at index 5 in haskell
2014/08/21 18:02:14 g appears last at index 5 in golang
2014/08/21 18:02:14 l appears last at index 6 in haskell
2014/08/21 18:02:14 A unicode character in "abcdefg" appears last at index 5 in gola\
ng
2014/08/21 18:02:14 A unicode character in "lmnop" appears last at index 6 in haskell
2014/08/21 18:02:14 Something controlled by (func(int32) bool)(0x46c0) appears at in\
dex 5 in golang
2014/08/21 18:02:14 Something controlled by (func(int32) bool)(0x46e0) does NOT appe\
ar in haskell
```

Manipulating

Manipulating a bunch of bytes is a common task too, and naturally, it's pretty easy too. `Map` allows you to change individual runes (it treats the byte slice as a bunch of bytes making up a "UTF-8-encoded Unicode code points"[12]). Replace works by

[12] From the `bytes` package documentation.

replacing chunks with the chunk you specify. `Runes` converts the byte slice to a rune slice, and `Repeat` gives you an easy way to build a byte slice prepopulate with default values.

bytes/manipulating.go

```go
package main

import (
	"bytes"
	"log"
)

func asciiAlphaUpcase(r rune) rune {
	return r - 32
}

func main() {
	golang := []byte("golang")

	// Map
	loudGolang := bytes.Map(asciiAlphaUpcase, golang)
	log.Printf("Turned %q into %q (ASCII alphabet upcase!)", golang, loudGolang)

	// Repalce
	original := []byte("go")
	replacement := []byte("Google Go")
	googleGolang := bytes.Replace(golang, original, replacement, -1)
	log.Printf("Replaced %q in %q with %q to get %q", original, golang, replacement, go\
ogleGolang)

	// Runes
	runes := bytes.Runes(golang)
	log.Printf("%q is made up of the following runes (in this case, ASCII codes): %v", \
golang, runes)

	// Repeat
	n := 8
	na := []byte("Na")
	batman := []byte(" Batman!")
	log.Printf("Made %d copies of %q and appended %q to get %q", n, na, batman, append(\
bytes.Repeat(na, n), batman...))
}
```

Output:

```
2014/08/21 18:02:13 Turned "golang" into "GOLANG" (ASCII alphabet upcase!)
2014/08/21 18:02:13 Replaced "go" in "golang" with "Google Go" to get "Google Golang"
2014/08/21 18:02:13 "golang" is made up of the following runes (in this case, ASCII \
codes): [103 111 108 97 110 103]
2014/08/21 18:02:13 Made 8 copies of "Na" and appended " Batman!" to get "NaNaNaNaNa\
NaNaNa Batman!"
```

Splitting and Joining

Splitting and joining strings and slices is a quick way to parse and build bits of information when a regex or a full lexer/parser would be overkill. The bytes package provides a host of functions for splitting byte slices, as well as the standard `Join` function.

bytes/splitjoin.go

```go
package main

import (
	"bytes"
	"log"
	"strings"
)

func main() {
	languages := []byte("golang haskell ruby python")

	individualLanguages := bytes.Fields(languages)
	log.Printf("Fields split %q on whitespace into %q", languages, individualLanguages)

	vowelsAndSpace := "aeiouy "
	split := bytes.FieldsFunc(languages, func(r rune) bool {
		return strings.ContainsRune(vowelsAndSpace, r)
	})
	log.Printf("FieldsFunc split %q on vowels and space into %q", languages, split)

	space := []byte{' '}
	splitLanguages := bytes.Split(languages, space)
	log.Printf("Split split %q on a single space into %q", languages, splitLanguages)
```

```

	numberOfSubslices := 2 // Not number of splits
	singleSplit := bytes.SplitN(languages, space, numberOfSubslices)
	log.Printf("SplitN split %q on a single space into %d subslices: %q", languages, nu\
mberOfSubslices, singleSplit)

	splitAfterLanguages := bytes.SplitAfter(languages, space)
	log.Printf("SplitAfter split %q AFTER a single space (keeping the space) into %q", \
languages, splitAfterLanguages)

	splitAfterNLanguages := bytes.SplitAfterN(languages, space, numberOfSubslices)
	log.Printf("SplitAfterN split %q AFTER a single space (keeping the space) into %d s\
ubslices: %q", languages, numberOfSubslices, splitAfterNLanguages)

	languagesBackTogether := bytes.Join(individualLanguages, space)
	log.Printf("Languages are back togeher again! %q == %q? %v", languagesBackTogether,\
 languages, bytes.Equal(languagesBackTogether, languages))
}
```

Output:

```
2014/08/21 18:02:14 Fields split "golang haskell ruby python" on whitespace into ["g\
olang" "haskell" "ruby" "python"]
2014/08/21 18:02:14 FieldsFunc split "golang haskell ruby python" on vowels and spac\
e into ["g" "l" "ng" "h" "sk" "ll" "r" "b" "p" "th" "n"]
2014/08/21 18:02:14 Split split "golang haskell ruby python" on a single space into \
["golang" "haskell" "ruby" "python"]
2014/08/21 18:02:14 SplitN split "golang haskell ruby python" on a single space into\
 2 subslices: ["golang" "haskell ruby python"]
2014/08/21 18:02:14 SplitAfter split "golang haskell ruby python" AFTER a single spa\
ce (keeping the space) into ["golang " "haskell " "ruby " "python"]
2014/08/21 18:02:14 SplitAfterN split "golang haskell ruby python" AFTER a single sp\
ace (keeping the space) into 2 subslices: ["golang " "haskell ruby python"]
2014/08/21 18:02:14 Languages are back togeher again! "golang haskell ruby python" =\
= "golang haskell ruby python"? true
```

Case

Frequently, you'll have a byte slice that's actually text. Maybe it's ASCII, maybe not. You might want to alter the slice with that in mind. We've already seen some functions that assume the data is really, and deal with runes. The bytes package also has 7 functions to deal with altering the case of the contained text. These include title casing, lower and upper casing.

bytes/case.go

```go
package main

import (
	"bytes"
	"log"
	"unicode"
)

func main() {
	quickBrownFox := []byte("The quick brown fox jumped over the lazy dog")

	title := bytes.Title(quickBrownFox)
	log.Printf("Title turned %q into %q", quickBrownFox, title)

	allTitle := bytes.ToTitle(quickBrownFox)
	log.Printf("ToTitle turned %q to %q", quickBrownFox, allTitle)

	allTitleTurkish := bytes.ToTitleSpecial(unicode.TurkishCase, quickBrownFox)
	log.Printf("ToTitleSpecial turned %q into %q using the Turkish case rules", quickBr\
ownFox, allTitleTurkish)

	lower := bytes.ToLower(title)
	log.Printf("ToLower turned %q into %q", title, lower)

	turkishCapitalI := []byte("İ")
	turkishLowerI := bytes.ToLowerSpecial(unicode.TurkishCase, turkishCapitalI)
	log.Printf("ToLowerSpecial turned %q into %q using the Turkish case rules", turkish\
CapitalI, turkishLowerI)

	upper := bytes.ToUpper(quickBrownFox)
	log.Printf("ToUpper turned %q to %q", quickBrownFox, upper)

	upperSpecial := bytes.ToUpperSpecial(unicode.TurkishCase, quickBrownFox)
	log.Printf("ToUpperSpecial turned %q into %q using the Turkish case rules", quickBr\
ownFox, upperSpecial)
}
```

Output:

```
2014/08/21 18:02:13 Title turned "The quick brown fox jumped over the lazy dog" into\
 "The Quick Brown Fox Jumped Over The Lazy Dog"
2014/08/21 18:02:13 ToTitle turned "The quick brown fox jumped over the lazy dog" to\
 "THE QUICK BROWN FOX JUMPED OVER THE LAZY DOG"
2014/08/21 18:02:13 ToTitleSpecial turned "The quick brown fox jumped over the lazy \
dog" into "THE QUİCK BROWN FOX JUMPED OVER THE LAZY DOG" using the Turkish case rule\
s
2014/08/21 18:02:13 ToLower turned "The Quick Brown Fox Jumped Over The Lazy Dog" in\
to "the quick brown fox jumped over the lazy dog"
2014/08/21 18:02:13 ToLowerSpecial turned "İ" into "i" using the Turkish case rules
2014/08/21 18:02:13 ToUpper turned "The quick brown fox jumped over the lazy dog" to\
 "THE QUICK BROWN FOX JUMPED OVER THE LAZY DOG"
2014/08/21 18:02:13 ToUpperSpecial turned "The quick brown fox jumped over the lazy \
dog" into "THE QUİCK BROWN FOX JUMPED OVER THE LAZY DOG" using the Turkish case rule\
s
```

Trimming

Lastly, trimming bytes from either end of a slice is a fairly common task. As is common in this chapter, the bytes package takes care of business.

Of special interest is the `TrimSpace` function. It's simple, but looking at the implementation gives you lots of other ideas. All it does it pass the `unicode.IsSpace` function to `TrimFunc`. All `TrimFunc` needs is a function that takes a rune and returns a bool, and the `unicode` package has plenty of those. You can trim digits, uppercase, lowercase, symbols, punctuation, and a whole mess of other things, by just combining the right `unicode` package function with `TrimFunc`.

bytes/trimming.go

```go
package main

import (
	"bytes"
	"log"
)

func trimOdd(r rune) bool {
	return r%2 == 1
}
```

```go

func main() {
	whitespace := " \t\r\n"

	padded := []byte("  \t\r\n\r\n\r\n  hello!!!    \t\t\t\t")
	trimmed := bytes.Trim(padded, whitespace)
	log.Printf("Trim removed runes in %q from the ends of %q to produce %q", whitespace\
, padded, trimmed)

	rhyme := []byte("aabbccddee")
	trimFunced := bytes.TrimFunc(rhyme, trimOdd)
	log.Printf("TrimFunc removed 'odd' runes from %q to produce %q", rhyme, trimFunced)

	leftTrimmed := bytes.TrimLeft(padded, whitespace)
	log.Printf("TrimLeft removed runes in %q from the left side of %q to produce %q", w\
hitespace, padded, leftTrimmed)

	leftTrimFunced := bytes.TrimLeftFunc(rhyme, trimOdd)
	log.Printf("TrimLeftFunc removed 'odd' runes from the left side of %q to produce %q\
", rhyme, leftTrimFunced)

	rightTrimmed := bytes.TrimRight(padded, whitespace)
	log.Printf("TrimRight removed runes in %q from the right side of %q to produce %q",\
 whitespace, padded, rightTrimmed)

	rightTrimFunced := bytes.TrimRightFunc(rhyme, trimOdd)
	log.Printf("TrimRightFunc removed 'odd' runes from the right side of %q to produce \
%q", rhyme, rightTrimFunced)

	spaceTrimmed := bytes.TrimSpace(padded)
	log.Printf("TrimSpace trimmed all whitespace from the ends of %q to produce %q", pa\
dded, spaceTrimmed)
}
```

Output:

```
2014/08/21 18:02:14 Trim removed runes in " \t\r\n" from the ends of "  \t\r\n\r\n\r\
\n  hello!!!    \t\t\t\t" to produce "hello!!!"
2014/08/21 18:02:14 TrimFunc removed 'odd' runes from "aabbccddee" to produce "bbccd\
d"
2014/08/21 18:02:14 TrimLeft removed runes in " \t\r\n" from the left side of "  \t\\
r\n\r\n\r\n  hello!!!    \t\t\t\t" to produce "hello!!!    \t\t\t\t"
2014/08/21 18:02:14 TrimLeftFunc removed 'odd' runes from the left side of "aabbccdd\
ee" to produce "bbccddee"
2014/08/21 18:02:14 TrimRight removed runes in " \t\r\n" from the right side of "  \\
t\r\n\r\n\r\n  hello!!!    \t\t\t\t" to produce "  \t\r\n\r\n\r\n  hello!!!"
2014/08/21 18:02:14 TrimRightFunc removed 'odd' runes from the right side of "aabbcc\
ddee" to produce "aabbccdd"
2014/08/21 18:02:14 TrimSpace trimmed all whitespace from the ends of "  \t\r\n\r\n\\
r\n  hello!!!    \t\t\t\t" to produce "hello!!!"
```

Buffer

The `Buffer` type is my favorite from the `bytes` package. It's your goto data structure for doing things in memory. It follows many of the interfaces in the `io` package, so it can be used any place that asks for those interfaces. Most importantly, it implements the `io.Reader` and `io.Writer` interfaces, which are used most when it comes to `io` operations. Run `buffer.go` will let you know all the interfaces it implements.

Now, it is a *buffer* so it doesn't implement the `io.ReaderAt` and `io.WriterAt` interfaces. You put stuff in and take stuff out, like a little black box. That being said, it's still a very useful data structure, especially for doing anything *in memory*.

bytes/buffer.go

```go
package main

import (
	"bytes"
	"io"
	"log"
	"os"
)

const interfaceFormat = "%T is an %s"

```

```go
func testInterfaces(buffer interface{}) {
	if _, ok := buffer.(io.ByteReader); ok {
		log.Printf(interfaceFormat, buffer, "io.ByteReader")
	}
	if _, ok := buffer.(io.ByteScanner); ok {
		log.Printf(interfaceFormat, buffer, "io.ByteScanner")
	}
	if _, ok := buffer.(io.Closer); ok {
		log.Printf(interfaceFormat, buffer, "io.Closer")
	}
	if _, ok := buffer.(io.LimitedReader); ok {
		log.Printf(interfaceFormat, buffer, "io.LimitedReader")
	}
	if _, ok := buffer.(io.ReadCloser); ok {
		log.Printf(interfaceFormat, buffer, "io.ReadCloser")
	}
	if _, ok := buffer.(io.ReadSeeker); ok {
		log.Printf(interfaceFormat, buffer, "io.ReadSeeker")
	}
	if _, ok := buffer.(io.ReadWriteCloser); ok {
		log.Printf(interfaceFormat, buffer, "io.ReadWriteCloser")
	}
	if _, ok := buffer.(io.ReadWriteSeeker); ok {
		log.Printf(interfaceFormat, buffer, "io.ReadWriteSeeker")
	}
	if _, ok := buffer.(io.ReadWriter); ok {
		log.Printf(interfaceFormat, buffer, "io.ReadWriter")
	}
	if _, ok := buffer.(io.Reader); ok {
		log.Printf(interfaceFormat, buffer, "io.Reader")
	}
	if _, ok := buffer.(io.ReaderAt); ok {
		log.Printf(interfaceFormat, buffer, "io.ReaderAt")
	}
	if _, ok := buffer.(io.ReaderFrom); ok {
		log.Printf(interfaceFormat, buffer, "io.ReaderFrom")
	}
	if _, ok := buffer.(io.RuneReader); ok {
		log.Printf(interfaceFormat, buffer, "io.RuneReader")
	}
	if _, ok := buffer.(io.RuneScanner); ok {
		log.Printf(interfaceFormat, buffer, "io.RuneScanner")
	}
```

```go
	if _, ok := buffer.(io.Seeker); ok {
		log.Printf(interfaceFormat, buffer, "io.Seeker")
	}
	if _, ok := buffer.(io.WriteCloser); ok {
		log.Printf(interfaceFormat, buffer, "io.WriteCloser")
	}
	if _, ok := buffer.(io.WriteSeeker); ok {
		log.Printf(interfaceFormat, buffer, "io.WriteSeeker")
	}
	if _, ok := buffer.(io.Writer); ok {
		log.Printf(interfaceFormat, buffer, "io.Writer")
	}
	if _, ok := buffer.(io.WriterAt); ok {
		log.Printf(interfaceFormat, buffer, "io.WriterAt")
	}
	if _, ok := buffer.(io.WriterTo); ok {
		log.Printf(interfaceFormat, buffer, "io.WriterTo")
	}
}

func fileExample(wr io.Writer) {
	log.Printf("wr is of type %T", wr)
	file, err := os.Open("buffer.go")
	if err != nil {
		log.Fatalf("Failed opening file: %s", err)
	}
	defer file.Close()
	io.Copy(wr, file)
}

func main() {
	var buffer bytes.Buffer
	testInterfaces(&buffer)
	fileExample(&buffer)
	log.Printf("Read %d byte file into buffer", buffer.Len())
	log.Println(buffer.String())
	buffer.Reset()
	log.Printf("After reset buffer is %d bytes long", buffer.Len())
}
```

Output:

```
2014/08/21 18:02:13 *bytes.Buffer is an io.ByteReader
2014/08/21 18:02:13 *bytes.Buffer is an io.ByteScanner
2014/08/21 18:02:13 *bytes.Buffer is an io.ReadWriter
2014/08/21 18:02:13 *bytes.Buffer is an io.Reader
2014/08/21 18:02:13 *bytes.Buffer is an io.ReaderFrom
2014/08/21 18:02:13 *bytes.Buffer is an io.RuneReader
2014/08/21 18:02:13 *bytes.Buffer is an io.RuneScanner
2014/08/21 18:02:13 *bytes.Buffer is an io.Writer
2014/08/21 18:02:13 *bytes.Buffer is an io.WriterTo
2014/08/21 18:02:13 wr is of type *bytes.Buffer
2014/08/21 18:02:13 Read 2597 byte file into buffer
2014/08/21 18:02:13 package main

import (
        "bytes"
        "io"
        "log"
        "os"
)

const interfaceFormat = "%T is an %s"

func testInterfaces(buffer interface{}) {
        if _, ok := buffer.(io.ByteReader); ok {
                log.Printf(interfaceFormat, buffer, "io.ByteReader")
        }
        if _, ok := buffer.(io.ByteScanner); ok {
                log.Printf(interfaceFormat, buffer, "io.ByteScanner")
        }
        if _, ok := buffer.(io.Closer); ok {
                log.Printf(interfaceFormat, buffer, "io.Closer")
        }
        if _, ok := buffer.(io.LimitedReader); ok {
                log.Printf(interfaceFormat, buffer, "io.LimitedReader")
        }
        if _, ok := buffer.(io.ReadCloser); ok {
                log.Printf(interfaceFormat, buffer, "io.ReadCloser")
        }
        if _, ok := buffer.(io.ReadSeeker); ok {
                log.Printf(interfaceFormat, buffer, "io.ReadSeeker")
        }
```

```
	if _, ok := buffer.(io.ReadWriteCloser); ok {
		log.Printf(interfaceFormat, buffer, "io.ReadWriteCloser")
	}
	if _, ok := buffer.(io.ReadWriteSeeker); ok {
		log.Printf(interfaceFormat, buffer, "io.ReadWriteSeeker")
	}
	if _, ok := buffer.(io.ReadWriter); ok {
		log.Printf(interfaceFormat, buffer, "io.ReadWriter")
	}
	if _, ok := buffer.(io.Reader); ok {
		log.Printf(interfaceFormat, buffer, "io.Reader")
	}
	if _, ok := buffer.(io.ReaderAt); ok {
		log.Printf(interfaceFormat, buffer, "io.ReaderAt")
	}
	if _, ok := buffer.(io.ReaderFrom); ok {
		log.Printf(interfaceFormat, buffer, "io.ReaderFrom")
	}
	if _, ok := buffer.(io.RuneReader); ok {
		log.Printf(interfaceFormat, buffer, "io.RuneReader")
	}
	if _, ok := buffer.(io.RuneScanner); ok {
		log.Printf(interfaceFormat, buffer, "io.RuneScanner")
	}
	if _, ok := buffer.(io.Seeker); ok {
		log.Printf(interfaceFormat, buffer, "io.Seeker")
	}
	if _, ok := buffer.(io.WriteCloser); ok {
		log.Printf(interfaceFormat, buffer, "io.WriteCloser")
	}
	if _, ok := buffer.(io.WriteSeeker); ok {
		log.Printf(interfaceFormat, buffer, "io.WriteSeeker")
	}
	if _, ok := buffer.(io.Writer); ok {
		log.Printf(interfaceFormat, buffer, "io.Writer")
	}
	if _, ok := buffer.(io.WriterAt); ok {
		log.Printf(interfaceFormat, buffer, "io.WriterAt")
	}
	if _, ok := buffer.(io.WriterTo); ok {
		log.Printf(interfaceFormat, buffer, "io.WriterTo")
	}
}
```

```

func fileExample(wr io.Writer) {
	log.Printf("wr is of type %T", wr)
	file, err := os.Open("buffer.go")
	if err != nil {
		log.Fatalf("Failed opening file: %s", err)
	}
	defer file.Close()
	io.Copy(wr, file)
}

func main() {
	var buffer bytes.Buffer
	testInterfaces(&buffer)
	fileExample(&buffer)
	log.Printf("Read %d byte file into buffer", buffer.Len())
	log.Println(buffer.String())
	buffer.Reset()
	log.Printf("After reset buffer is %d bytes long", buffer.Len())
}

2014/08/21 18:02:13 After reset buffer is 0 bytes long
```

Reader

The `bytes.Reader` gives you a way to wrap byte slices in a little structure implementing 8 interfaces from the `io` package. If you have a byte slice, and you need to *read* them, wrap it with the `bytes.NewReader` function and go to town. Running the `reader.go` file shows all the interfaces the `bytes.Reader` type implements.

bytes/reader.go

```
package main

import (
	"bytes"
	"io"
	"log"
)

const interfaceFormat = "%T is an %s"
```

```go

func testInterfaces(buffer interface{}) {
	if _, ok := buffer.(io.ByteReader); ok {
		log.Printf(interfaceFormat, buffer, "io.ByteReader")
	}
	if _, ok := buffer.(io.ByteScanner); ok {
		log.Printf(interfaceFormat, buffer, "io.ByteScanner")
	}
	if _, ok := buffer.(io.Closer); ok {
		log.Printf(interfaceFormat, buffer, "io.Closer")
	}
	if _, ok := buffer.(io.LimitedReader); ok {
		log.Printf(interfaceFormat, buffer, "io.LimitedReader")
	}
	if _, ok := buffer.(io.ReadCloser); ok {
		log.Printf(interfaceFormat, buffer, "io.ReadCloser")
	}
	if _, ok := buffer.(io.ReadSeeker); ok {
		log.Printf(interfaceFormat, buffer, "io.ReadSeeker")
	}
	if _, ok := buffer.(io.ReadWriteCloser); ok {
		log.Printf(interfaceFormat, buffer, "io.ReadWriteCloser")
	}
	if _, ok := buffer.(io.ReadWriteSeeker); ok {
		log.Printf(interfaceFormat, buffer, "io.ReadWriteSeeker")
	}
	if _, ok := buffer.(io.ReadWriter); ok {
		log.Printf(interfaceFormat, buffer, "io.ReadWriter")
	}
	if _, ok := buffer.(io.Reader); ok {
		log.Printf(interfaceFormat, buffer, "io.Reader")
	}
	if _, ok := buffer.(io.ReaderAt); ok {
		log.Printf(interfaceFormat, buffer, "io.ReaderAt")
	}
	if _, ok := buffer.(io.ReaderFrom); ok {
		log.Printf(interfaceFormat, buffer, "io.ReaderFrom")
	}
	if _, ok := buffer.(io.RuneReader); ok {
		log.Printf(interfaceFormat, buffer, "io.RuneReader")
	}
	if _, ok := buffer.(io.RuneScanner); ok {
		log.Printf(interfaceFormat, buffer, "io.RuneScanner")
```

```go
        }
        if _, ok := buffer.(io.Seeker); ok {
                log.Printf(interfaceFormat, buffer, "io.Seeker")
        }
        if _, ok := buffer.(io.WriteCloser); ok {
                log.Printf(interfaceFormat, buffer, "io.WriteCloser")
        }
        if _, ok := buffer.(io.WriteSeeker); ok {
                log.Printf(interfaceFormat, buffer, "io.WriteSeeker")
        }
        if _, ok := buffer.(io.Writer); ok {
                log.Printf(interfaceFormat, buffer, "io.Writer")
        }
        if _, ok := buffer.(io.WriterAt); ok {
                log.Printf(interfaceFormat, buffer, "io.WriterAt")
        }
        if _, ok := buffer.(io.WriterTo); ok {
                log.Printf(interfaceFormat, buffer, "io.WriterTo")
        }
}

func main() {
        golang := []byte("golang")
        reader := bytes.NewReader(golang)
        testInterfaces(reader)
}
```

Output:

```
2014/08/21 18:02:14 *bytes.Reader is an io.ByteReader
2014/08/21 18:02:14 *bytes.Reader is an io.ByteScanner
2014/08/21 18:02:14 *bytes.Reader is an io.ReadSeeker
2014/08/21 18:02:14 *bytes.Reader is an io.Reader
2014/08/21 18:02:14 *bytes.Reader is an io.ReaderAt
2014/08/21 18:02:14 *bytes.Reader is an io.RuneReader
2014/08/21 18:02:14 *bytes.Reader is an io.RuneScanner
2014/08/21 18:02:14 *bytes.Reader is an io.Seeker
2014/08/21 18:02:14 *bytes.Reader is an io.WriterTo
```

compress

Honey, I Shrunk The Kids

The `compress` package implements various compression algorithms. The `bzip2` subpackage is a bit of an odd child since it only implements a reader (decompression) and not a writer (compression).

Each package works pretty much the same. You create either a reader[13] or a writer,[14] maybe specifying some options like compression level, and use the object like any other reader or writer. Not much more complicated than that.

ALL THE CODE

Since the code is all very similar, we're just going to throw everything in one file, and use the `flag` package to control what we're doing.

compress/everything.go

```
package main

import (
	"compress/bzip2"
	"compress/flate"
	"compress/gzip"
	"compress/lzw"
	"compress/zlib"
	"flag"
	"fmt"
	"io"
	"log"
	"os"
)

var (
```

[13] Either an `io.Reader` or `io.ReadCloser`, or something that implements those interfaces.
[14] Either an `io.Writer` or `io.WriteCloser`, or something that implements those interfaces.

```
        compress   = flag.Bool("compress", false, "Perform compression")
        decompress = flag.Bool("decompress", false, "Perform decompression")
        algorithm  = flag.String("algorithm", "", "The algorithm to use (one of bzip2, flat\
e, gzip, lzw, zlib)")
        input      = flag.String("input", "", "The file to compress or decompress")
)

func filename() string {
        return fmt.Sprintf("%s.%s", *input, *algorithm)
}

func openOutputFile() *os.File {
        file, err := os.OpenFile(filename(), os.O_WRONLY|os.O_CREATE, 0644)
        if err != nil {
                log.Fatalf("failed opening output file: %s", err)
        }
        return file
}

func openInputFile() *os.File {
        file, err := os.Open(*input)
        if err != nil {
                log.Fatalf("failed opening input file: %s", err)
        }
        return file
}

func getCompressor(out io.Writer) io.WriteCloser {
        switch *algorithm {
        case "bzip2":
                log.Fatalf("no compressor for bzip2. Try `bzip2 -c everything.go > everything.go.b\
zip2`")
        case "flate":
                compressor, err := flate.NewWriter(out, flate.BestCompression)
                if err != nil {
                        log.Fatalf("failed making flate compressor: %s", err)
                }
                return compressor
        case "gzip":
                return gzip.NewWriter(out)
        case "lzw":
                // More specific uses of Order and litWidth are in the package docs
                return lzw.NewWriter(out, lzw.MSB, 8)
```

```
	case "zlib":
		return zlib.NewWriter(out)
	default:
		log.Fatalf("choose one of bzip2, flate, gzip, lzw, zlib with -algorithm")
	}
	panic("not reached")
}

func getDecompressor(in io.Reader) io.Reader {
	switch *algorithm {
	case "bzip2":
		return bzip2.NewReader(in)
	case "flate":
		return flate.NewReader(in)
	case "gzip":
		decompressor, err := gzip.NewReader(in)
		if err != nil {
			log.Fatalf("failed making gzip decompressor")
		}
		return decompressor
	case "lzw":
		return lzw.NewReader(in, lzw.MSB, 8)
	case "zlib":
		decompressor, err := zlib.NewReader(in)
		if err != nil {
			log.Fatalf("failed making zlib decompressor")
		}
		return decompressor
	}
	panic("not reached")
}

func compression() {
	output := openOutputFile()
	defer output.Close()
	compressor := getCompressor(output)
	defer compressor.Close()
	input := openInputFile()
	defer input.Close()
	io.Copy(compressor, input)
}

func decompression() {
```

```go
	input := openInputFile()
	defer input.Close()
	decompressor := getDecompressor(input)
	if c, ok := decompressor.(io.Closer); ok {
		defer c.Close()
	}
	io.Copy(os.Stdout, decompressor)
}

func main() {
	flag.Parse()
	if *input == "" {
		log.Fatalf("Please specify an input file with -input")
	}
	switch {
	case *compress:
		compression()
	case *decompress:
		decompression()
	default:
		log.Println("must specify one of -compress or -decompress")
	}
}
```

Accept-Encoding: gzip

In the real world, we can do some fun things. For requests, the `net/http` package handles compression for us. On the server side, you have to do things yourself.

You can decode a compressed body using a `gzip.Reader`, and you can send a compressed body using a `gzip.Writer`.

compress/http.go

```go
package main

import (
	"bytes"
	"compress/gzip"
	"flag"
	"fmt"
	"io"
	"log"
	"net/http"
	"os"
	"strings"
)

var (
	port     = flag.Int("port", 8888, "The port to listen on")
	compress = flag.Bool("compress", false, "Compress using gzip")
	input    = flag.String("input", "http.go", "The file to send to the echo")
)

func compressor(enc string, wr io.Writer) (io.Writer, string) {
	if strings.Contains(enc, "gzip") {
		return gzip.NewWriter(wr), "gzip"
	}
	return wr, ""
}

func decompressor(enc string, rd io.Reader) io.Reader {
	if strings.Contains(enc, "gzip") {
		gz, err := gzip.NewReader(rd)
		if err != nil {
			log.Fatalf("Failed creating gzip decompressor: %s", err)
		}
		return gz
	}
	return rd
}

func readBody(enc string, rc io.ReadCloser) *bytes.Buffer {
	var buffer bytes.Buffer
	rd := decompressor(enc, rc)
```

```go
	io.Copy(&buffer, rd)
	if c, ok := rd.(io.Closer); ok {
		c.Close()
	}
	rc.Close()
	return &buffer
}

func echo(w http.ResponseWriter, req *http.Request) {
	log.Printf("Request headers: %#v", req.Header)
	body := readBody(req.Header.Get("Content-Encoding"), req.Body)

	// Since we're echoing, just send the same Content-Type back
	w.Header().Set("Content-Type", req.Header.Get("Content-Type"))

	wr, enc := compressor(req.Header.Get("Accept-Encoding"), w)
	if enc != "" {
		w.Header().Set("Content-Encoding", enc)
	}
	if c, ok := wr.(io.Closer); ok {
		defer c.Close()
	}

	io.Copy(wr, body)
}

func server() {
	http.HandleFunc("/echo", echo)
	log.Fatal(http.ListenAndServe(fmt.Sprintf(":%d", *port), nil))
}

func encoding() string {
	if *compress {
		return "gzip"
	}
	return ""
}

func bufferFile(name string) (*bytes.Buffer, string) {
	var buffer bytes.Buffer
	file, err := os.Open(name)
	if err != nil {
		log.Fatalf("Failed opening file: %s", err)
```

```go
	}
	defer file.Close()
	wr, enc := compressor(encoding(), &buffer)
	if c, ok := wr.(io.Closer); ok {
		defer c.Close()
	}
	io.Copy(wr, file)
	return &buffer, enc
}

func httpClient() *http.Client {
	return &http.Client{
		Transport: &http.Transport{
			// The http client package handles gzip compression for us.
			DisableCompression: !*compress,
		},
	}
}

func client() {
	buffer, enc := bufferFile(*input)
	url := fmt.Sprintf("http://localhost:%d/echo", *port)
	req, err := http.NewRequest("POST", url, buffer)
	if err != nil {
		log.Fatalf("Failed creating request: %s", err)
	}
	req.Header.Set("Content-Type", "text/plain; charset=utf-8")

	if enc != "" {
		req.Header.Set("Content-Encoding", enc)
	}

	resp, err := httpClient().Do(req)
	if err != nil {
		log.Fatalf("Failed making HTTP request: %s", err)
	}
	defer resp.Body.Close()
	log.Printf("Response headers: %#v", resp.Header)
	io.Copy(os.Stdout, resp.Body)
}

func main() {
	flag.Parse()
```

```
        go server()
        client()
}
```

If the `Content-Encoding` is gzip, the `decompressor` function wraps the original reader in a `gzip.Reader`. Otherwise, it returns the original `io.Reader`. The `compressor` function does the same but with `Accept-Encoding` and `gzip.Writer`.

This simple wrapper let's the handler function optionally decode compressed request bodies, and optionally send compressed response bodies. The only thing to watch for is that there might be two readers or writers, which may or may not need closing.

container

The `container` package consists of 3 sub-packages to make you life a little easier when dealing with some basic container types.

The `list` and `ring` packages implement their own types, providing a `New()` function to create each structure. The `heap` package on the other hand, provides functions to operate on an interface. All you need to do is define the methods on your type, and away you go.

We'll look at the sub-packages in order, starting with the `heap` package.

heap

Unlike the other two types in the `container` package (which implement their own actual container type), the `heap` package is just a set of functions operating on an interface.

This means you get to deal with your own datatype. Just implement `sort.Interface` (three methods) and `heap.Interface` (two methods) and you can start dealing with your container as a `heap`.

Keep in mind that when you print out the raw heap (if you base the heap off a slice like I do) it won't be sorted. A heap basically stores a tree structure in a slice, so it's sorted in a way the heap package understands. When you `Pop` items from the heap, they come off in the correct order. The difference can be seen between the second and third lines of the output.

We'll see a cooler example using a `heap` later.

container/heap.go

```go
package main

// Interfaces
//
// type heap.Interface interface {
//     sort.Interface
//     // add x as element Len()
//     Push(x interface{})
```

```
//     // remove and return element Len() - 1.
//     Pop() interface{}
// }
//
// type sort.Interface interface {
//     // Len is the number of elements in the collection.
//     Len() int
//     // Less returns whether the element with index i should sort
//     // before the element with index j.
//     Less(i, j int) bool
//     // Swap swaps the elements with indexes i and j.
//     Swap(i, j int)
// }

import (
        "container/heap"
        "log"
        "math/rand"
)

type IntHeap []int

func (h IntHeap) Len() int {
        return len(h)
}

func (h IntHeap) Less(i, j int) bool {
        return h[i] < h[j]
}

func (h IntHeap) Swap(i, j int) {
        h[i], h[j] = h[j], h[i]
}

func (h *IntHeap) Push(v interface{}) {
        a := *h
        a = append(a, v.(int))
        *h = a
}

func (h *IntHeap) Pop() interface{} {
        a := *h
        n := len(a)
```

```
	v := a[n-1]
	*h = a[0 : n-1]
	return v
}

func main() {
	h := make(IntHeap, 0)
	log.Printf("%v", h)
	for i := 0; i < 10; i++ {
		heap.Push(&h, rand.Intn(25))
	}
	log.Printf("%v", h)

	l := h.Len()
	ints := make([]int, 0, l)
	for i := 0; i < l; i++ {
		ints = append(ints, heap.Pop(&h).(int))
	}
	log.Printf("%v", ints)
	log.Printf("%v", h)
}
```

list

The `list` package implements, as the overview says, a doubly linked list. You'll want to start with `list.New()` to get yourself a new list, and use `PushBack`, `PushBackList`, `PushFront`, and `PushFrontList` to add things to the list.

Once you have something built up, you can use `Front` and `Back` to get the beginning or end of the list (in the form of a pointer to a `list.Element` struct). Now you can use `Next` and `Prev` to advance through the list.

Once you have an `Element` you can use `MoveToBack` and `MoveToFront` to push the element around, or you can use `InsertAfter` and `InsertBefore` to insert a new element in a specific location. Removing an `Element` is easy once you have it as well, just use `Remove` on the list.

Unlike the `ring` package (which we'll see next), `list.List` doesn't have a `Do` method for iterating over all the elements, so I've implemented one. It's really simple, and in your normal day of coding the regular `for` loop would be preferred, but I'm doing it as an example.

container/list.go

```go
package main

import (
	"container/list"
	"log"
)

const size = 5

func Do(l *list.List, f func(interface{})) {
	// Standard list iterating straight from their example
	for e := l.Front(); e != nil; e = e.Next() {
		f(e.Value)
	}
}

func printList(l *list.List) {
	elements := make([]interface{}, 0, l.Len())
	Do(l, func(i interface{}) {
		elements = append(elements, i)
	})
	log.Printf("%v", elements)
}

func main() {
	l := list.New()
	printList(l) // []
	for i := 0; i < size; i++ {
		l.PushBack(i)
	}
	printList(l) // [0 1 2 3 4]

	l = l.Init()
	for i := 0; i < size; i++ {
		l.PushFront(i)
	}
	printList(l) // [4 3 2 1 0]

	f := l.Front()
	e := f.Next().Next()
	e = l.InsertAfter(10, e)
```

```
        printList(l)         // [4 3 2 10 1 0]
        log.Println(l.Len()) // 6

        l.Remove(e.Next())
        printList(l)         // [4 3 2 10 0]
        log.Println(l.Len()) // 5
}
```

ring

The `ring` container is interesting. Much like a tree, the `ring` type is both the top level container and an element in the container.[15] It's both the container, and what it contains. Woah.

Anyway, you can make a ring using `ring.New(n int)` or just by allocating yourself a new `ring.Ring` and going from there. After you've made a new `Ring`, you can add data to it simply by setting the `Value`.

To load up a ring, make a new one, and set it's `Value`. To add other values to the ring, advance the ring (remembering to save the return value, like with `append`) and set its `Value`. Rinse and repeat until the ring is full.

You could also use the `Link` method to add more nodes.

Once you have a ring, you can use the `Next`, `Prev`, `Move`, `Unlink`, and `Do` methods to manipulate the ring. `Next` and `Prev` are pretty straightforward

container/ring.go

```
package main

import (
        "container/ring"
        "log"
)

const size = 5

func printRing(r *ring.Ring) {
        elements := make([]interface{}, 0, r.Len())
        r.Do(func(i interface{}) {
```

[15] In a tree, a node would have pointers to the left and right subtrees (which are just nodes), and to the element the node holds. In the ring, it has the same pointers, except they are called `prev` and `next`.

```go
		elements = append(elements, i)
	})
	log.Printf("%v", elements)
}

func buildRingFirstMethod() *ring.Ring {
	r := ring.New(size)
	printRing(r) // [<nil> <nil> <nil> <nil> <nil>]
	for i := 0; i < size; i++ {
		r.Value = i
		r = r.Next()
	}
	return r
}

func buildRingSecondMethod() *ring.Ring {
	r := &ring.Ring{Value: 0}
	printRing(r) // [0]
	for i := 1; i < size; i++ {
		r.Prev().Link(&ring.Ring{Value: i})
	}
	return r
}

func main() {
	r := buildRingFirstMethod()
	printRing(r) // [0 1 2 3 4]

	r2 := buildRingSecondMethod()
	printRing(r2) // [0 1 2 3 4]
}
```

Thread Pool Example

In a language with raw threads (like Java or C#), you will typically see a `ThreadPool` type. You make one of a certain size, and submit jobs to it, and they get pulled off the queue in order. In Go, since goroutines aren't threads (but are managed by a thread pool which is in turn managed by the runtime), you typically don't have to do this, but we'll implement a `ThreadPool` using the `list` container anyway. You know, for fun.

It's not the best chunk of code (for example, it could be rewritten without the locks using channels, like the priority queue example we'll see later), but it illustrates that only up to 4 goroutines run at a time. You could do it with a simple slice too, instead of the `list` package. Again, in Go, you really don't need a `ThreadPool`.

container/thread_pool.go

```
package main

import (
	"container/list"
	"log"
	"sync"
	"time"
)

type ThreadPool struct {
	size, running int
	list          *list.List
	m             sync.Mutex
}

func NewThreadPool(size int) *ThreadPool {
	tp := &ThreadPool{
		size: size,
		list: list.New(),
	}
	return tp
}

func (tp *ThreadPool) onStop() {
	tp.m.Lock()
	tp.running--
	tp.m.Unlock()
	tp.run()
}

func (tp *ThreadPool) run() {
	tp.m.Lock()
	defer tp.m.Unlock()
	if tp.list.Len() > 0 && tp.running < tp.size {
		f := tp.list.Remove(tp.list.Front()).(func())
		tp.running++
		go func() {
```

```
                        f()
                        tp.onStop()
                }()
        }
}

func (tp *ThreadPool) Submit(f func()) {
        tp.list.PushBack(f)
        tp.run()
}

func main() {
        var wg sync.WaitGroup
        tp := NewThreadPool(4)
        for i := 0; i < 16; i++ {
                wg.Add(1)
                (func(id int) {
                        log.Printf("Subtmitted job %d", id)
                        tp.Submit(func() {
                                time.Sleep(3 * time.Second)
                                log.Printf("Hello from job %d", id)
                                wg.Done()
                        })
                })(i)
        }
        wg.Wait()
}
```

Round Robin Load Balancer Example

Sometimes you want to get a few goroutines running and then submit a bunch of jobs in a round robin fashion. Maybe start 4 goroutines, then submit a job which goes to the first. Submit another which goes to the second, then the third, then the fourth, and then the first again, back to the front.

We can use a ring for this. Again, probably not something you'd actually do in Go, and not the prettiest code I've ever written, but it's an example nonetheless.

container/round_robin.go

```go
package main

import (
	"container/ring"
	"log"
	"sync"
	"time"
)

type RoundRobin struct {
	ring *ring.Ring
	m    sync.Mutex
}

func process(id int, funcs chan func()) {
	for f := range funcs {
		f()
		log.Printf("Job finished in goroutine %d", id)
	}
}

func NewRoundRobinScheduler(ringSize, channelSize int) *RoundRobin {
	r := ring.New(ringSize)
	for i := 0; i < ringSize; i++ {
		c := make(chan func(), channelSize)
		go process(i, c)
		r.Value = c
		r = r.Next()
	}
	return &RoundRobin{ring: r}
}

func (rr *RoundRobin) Submit(f func()) {
	rr.m.Lock()
	defer rr.m.Unlock()
	c := rr.ring.Value.(chan func())
	c <- f
	rr.ring = rr.ring.Next()
}

func main() {
```

```go
	var wg sync.WaitGroup
	rr := NewRoundRobinScheduler(4, 4)
	for i := 0; i < 16; i++ {
		wg.Add(1)
		(func(id int) {
			log.Printf("Submitted job %d", id)
			rr.Submit(func() {
				time.Sleep(3 * time.Second)
				log.Printf("Hello from job %d", id)
				wg.Done()
			})
		})(i)
	}
	wg.Wait()
}
```

Priority Queue Load Balancer Example

Since the `heap` package works on an interface, you can bend it to your will. In this example, we'll implement a priority queue based load balancer.[16] You can submit jobs to it, and it submits the job to the worker with the shortest queue. The important parts are the methods implementing the `heap.Interface` interface.

Let it run for a bit and examine the log to see where requests are getting queued.

container/priority_queue.go

```go
// Original code from http://golang.org/doc/talks/io2010/balance.go
//
// Copyright (c) 2012 The Go Authors. All rights reserved.
//
// Redistribution and use in source and binary forms, with or without
// modification, are permitted provided that the following conditions are
// met:
//
//    * Redistributions of source code must retain the above copyright
// notice, this list of conditions and the following disclaimer.
//    * Redistributions in binary form must reproduce the above
// copyright notice, this list of conditions and the following disclaimer
```

[16] I've taken the code from a Google IO 2010 talk, and rewritten it a little bit for my purpose. The original code can be seen here (http://golang.org/doc/talks/io2010/balance.go) and is licensed by the Golang BSD license.

```go
// in the documentation and/or other materials provided with the
// distribution.
//    * Neither the name of Google Inc. nor the names of its
// contributors may be used to endorse or promote products derived from
// this software without specific prior written permission.
//
// THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
// "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
// LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR
// A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT
// OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL,
// SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT
// LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE,
// DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY
// THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT
// (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE
// OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

package main

import (
	"container/heap"
	"fmt"
	"log"
	"math/rand"
	"time"
)

const (
	MaxQueueLength = 100
	MaxRequesters  = 10
	Seconds        = 2e9
)

func requester(work chan Request) {
	for {
		time.Sleep(time.Duration(rand.Int63n(MaxRequesters * Seconds)))
		work <- func() {
			r := rand.Int63n(MaxRequesters*Seconds) + 10
			time.Sleep(time.Duration(r))
		}
	}
}
```

```go

type Request func()

type Worker struct {
	id       int
	pending  int
	requests chan Request
	index    int
}

func (w *Worker) work(done chan *Worker) {
	for {
		req := <-w.requests
		req()
		done <- w
	}
}

func (w *Worker) String() string {
	return fmt.Sprintf("W%d{pending: %d}", w.id, w.pending)
}

type Pool []*Worker

func (p Pool) Len() int {
	return len(p)
}

func (p Pool) Less(i, j int) bool {
	return p[i].pending < p[j].pending
}

func (p *Pool) Swap(i, j int) {
	a := *p
	a[i], a[j] = a[j], a[i]
	a[i].index = i
	a[j].index = j
}

func (p *Pool) Push(i interface{}) {
	w := i.(*Worker)
	a := *p
	n := len(a)
```

```
        w.index = n
        a = append(a, w)
        *p = a
}

func (p *Pool) Pop() interface{} {
        a := *p
        n := len(a)
        w := a[n-1]
        w.index = -1
        *p = a[0 : n-1]
        return w
}

type Balancer struct {
        pool Pool
        done chan *Worker
}

func NewBalancer(size int) *Balancer {
        done := make(chan *Worker, size)
        b := &Balancer{
                pool: make(Pool, 0, size),
                done: done,
        }
        for i := 0; i < size; i++ {
                w := &Worker{id: i, requests: make(chan Request, MaxQueueLength)}
                heap.Push(&b.pool, w)
                go w.work(done)
        }
        return b
}

func (b *Balancer) Balance(requests chan Request) {
        for {
                select {
                case req := <-requests:
                        b.dispatch(req)
                        log.Printf("New request, %s", b.pool)
                case w := <-b.done:
                        b.completed(w)
                        log.Printf("Request finished, %s", b.pool)
                }
```

```
        }
}

func (b *Balancer) dispatch(req Request) {
        w := heap.Pop(&b.pool).(*Worker)
        w.requests <- req
        w.pending++
        heap.Push(&b.pool, w)
}

func (b *Balancer) completed(w *Worker) {
        w.pending--
        heap.Remove(&b.pool, w.index)
        heap.Push(&b.pool, w)
}

func main() {
        requests := make(chan Request)
        for i := 0; i < MaxRequesters; i++ {
                go requester(requests)
        }
        NewBalancer(4).Balance(requests)
}
```

crypto

The `crypto` package is an umbrella for a wide variety of cryptographic related packages.

`crypto/aes` and `crypto/des` handle the popular block ciphers.

Digital signatures are handled by `crypto/ecdsa` and `crypto/dsa`.

The standard array of hashes are included in `cryto/{md5,sha1,sha256,sha512}`, along with `crypto/hmac`.

`crypto/rc4` is included for good measure. It's most likely for compatibility with other programs and languages, since in the documentation in the `Bugs` section discourages using it for new things.

Handling secure TCP connections with TLS is a breeze with the `crypto/tls` and `crypto/x509` packages.

Other entertaining packages include `crypto/rand` for handling random number generation in a cryptographically secure way, and the `crypto/subtle` package for doing constant time operations.

Disclaimer

Cryptography is a tricky subject and doing it wrong is pretty easy. **I am not a professional cryptographer**,[17] so be sure to do your own research and reading (from sources who are professional cryptographers) when doing anything cryptography related. I'm just doing my best to show you how to use the `crypto` packages in the Go Standard Library.

> ⚠ Do not copy any code from this book (or anywhere) and just paste it into your application without understanding what it does. As the code license says, I provide all the code without warranty of any kind.

[17] I don't even play one on TV.

Block Ciphers

AES[18] and DES[19] are symmetric block ciphers[20]. They are symmetric because the same key is used to both encrypt and decrypt. They are block ciphers because they operate on blocks of a fixed size.

In these examples I've used the `encoding/pem` package to serialize the keys. We'll look at the `encoding` package in more detail in a later chapter.

AES

The `crypto/aes` package implements the Advanced Encryption Standard[21] algorithm. Since it's a block cipher, you work with the `cipher.Block` interface from the `crypto/cipher` package.

To start, `aes.NewCipher` returns a `cipher.Block` which has the ability to `Encrypt` and `Decrypt` blocks of data. You probably don't want to use this type and these methods directly, since you have to work on individual blocks.

You might look at `cipher.NewCBCDecrypter` and `cipher.NewCBCEncrypter`, which allows you to deal with all your data at once, but your source data must be a multiple of the block size, 16 bytes.[22] Since your data probably won't be a nice multiple of 16, you'll have to do some **padding**. There are nice algorithms to do this, but they have their problems, and better methods have come along.

For a good look at a variety of block cipher modes, I'd recommended the Wikipedia page on the subject[23]. In Go, the fun parts are the **CFB**, **OFB**, and **CTR** modes. They give you a `cipher.Stream` type which lets you pump plaintext bytes through the stream and get ciphertext out the other side without worrying about padding. The counter (CTR) method seems to be the better mode, so I've used it in the example.

The other nice thing about these modes is that encryption and decryption work the same as far as the code is concerned. There is just the `XORKeyStream(dst, src []byte)` method, and if `dst` is your ciphertext and `src` is your plaintext, it encryptes. If you flip the two, your ciphertext gets decrypted. As the docs say, `dst` and `src` can also point to the same piece of memory, so the algorithm can work in essentially constant space.

All you need to build your `cipher.Stream` is an **initialization vector**, or IV. It has to be the same length as the block size, 16 in this case. The IV should be generated using a

[18] http://en.wikipedia.org/wiki/Advanced_Encryption_Standard
[19] http://en.wikipedia.org/wiki/Data_Encryption_Standard
[20] http://en.wikipedia.org/wiki/Block_cipher
[21] http://en.wikipedia.org/wiki/Advanced_Encryption_Standard
[22] See the `aes.BlockSize` constant.
[23] http://en.wikipedia.org/wiki/Block_cipher_modes_of_operation

secure random method (like `crypto/rand`) for each encryption, and you can then send the IV along with the encrypted data. Simply concatenating the IV and encrypted data is fine, as long as the IV is only ever used once (for that transmission) and is generated securely. I've just hardcoded the IV for the purpose of the example, but you could very easily (and should) use `crypto/rand` to generate one.

crypto/aes.go

```go
package main

import (
	"crypto/aes"
	"crypto/cipher"
	"crypto/rand"
	"encoding/pem"
	"flag"
	"fmt"
	"io/ioutil"
	"log"
)

const (
	KeyFile       = "aes.%d.key"
	EncryptedFile = "aes.%d.enc"
)

var (
	IV      = []byte("batman and robin") // 16 bytes
	message = flag.String("message", "Batman is Bruce Wayne", "The message to encrypt")
	keySize = flag.Int("keysize", 32, "The keysize in bytes to use: 16, 24, or 32 (defa\
ult)")
	do      = flag.String("do", "encrypt", "The operation to perform: decrypt or encryp\
t (default) ")
)

func MakeKey() []byte {
	key := make([]byte, *keySize)
	n, err := rand.Read(key)
	if err != nil {
		log.Fatalf("failed to read new random key: %s", err)
	}
	if n < *keySize {
		log.Fatalf("failed to read entire key, only read %d out of %d", n, *keySize)
	}
```

```
        return key
}

func SaveKey(filename string, key []byte) {
        block := &pem.Block{
                Type:  "AES KEY",
                Bytes: key,
        }
        err := ioutil.WriteFile(filename, pem.EncodeToMemory(block), 0644)
        if err != nil {
                log.Fatalf("failed saving key to %s: %s", filename, err)
        }
}

func ReadKey(filename string) ([]byte, error) {
        key, err := ioutil.ReadFile(filename)
        if err != nil {
                return key, err
        }
        block, _ := pem.Decode(key)
        return block.Bytes, nil
}

func Key() []byte {
        file := fmt.Sprintf(KeyFile, *keySize)
        key, err := ReadKey(file)
        if err != nil {
                log.Println("failed reading keyfile, making a new one...")
                key = MakeKey()
                SaveKey(file, key)
        }
        return key
}

func MakeCipher() cipher.Block {
        c, err := aes.NewCipher(Key())
        if err != nil {
                log.Fatalf("failed making the AES cipher: %s", err)
        }
        return c
}

func Crypt(bytes []byte) []byte {
```

```
        blockCipher := MakeCipher()
        stream := cipher.NewCTR(blockCipher, IV)
        stream.XORKeyStream(bytes, bytes)
        return bytes
}

func Encrypt() {
        encrypted := Crypt([]byte(*message))
        err := ioutil.WriteFile(fmt.Sprintf(EncryptedFile, *keySize), encrypted, 0644)
        if err != nil {
                log.Fatalf("failed writing encrypted file: %s", err)
        }
}

func Decrypt() {
        bytes, err := ioutil.ReadFile(fmt.Sprintf(EncryptedFile, *keySize))
        if err != nil {
                log.Fatalf("failed reading encrypted file: %s", err)
        }
        plaintext := Crypt(bytes)
        log.Printf("decrypted message: %s", plaintext)
}

func main() {
        flag.Parse()

        switch *keySize {
        case 16, 24, 32:
                // Keep calm and carry on...
        default:
                log.Fatalf("%d is not a valid keysize. Must be one of 16, 24, 32", *keySize)
        }

        switch *do {
        case "encrypt":
                Encrypt()
        case "decrypt":
                Decrypt()
        default:
                log.Fatalf("%s is not a valid operation. Must be one of encrypt or decrypt", *do)
        }
}
```

DES/TripleDES

DES, like AES is a symmetric block cipher. As far as code is concerned, it works exactly the same as AES. Generate a key, make the cipher, then use the `crypto/cipher` types to simplify things a bit.

You should prefer AES over DES for new applications, since the small 56-bit key size used by DES is just too small. A key can typically be cracked in a few days with good hardware (or even just a bunch of money thrown at Amazon EC2).

Since the flow is almost exactly the same as AES, this code is basically just the AES example with AES swapped out for DES.

Unlike the AES example, I don't change the key and encrypted file names if you use the `-3` flag to use 3DES. Try running encryption without the flag, and decryption with it.

crypto/des.go

```go
package main

import (
	"crypto/cipher"
	"crypto/des"
	"crypto/rand"
	"encoding/pem"
	"flag"
	"io/ioutil"
	"log"
)

const (
	KeyFile       = "des.key"
	EncryptedFile = "des.enc"
)

var (
	IV      = []byte("superman") // 8 bytes
	triple  = flag.Bool("3", false, "Use 3DES")
	message = flag.String("message", "Batman is Bruce Wayne", "The message to encrypt")
	do      = flag.String("do", "encrypt", "The operation to perform: decrypt or encryp\
t (default) ")
)

func MakeKey() []byte {
```

```
	size := 8
	if *triple {
		size *= 3
	}
	key := make([]byte, size)
	n, err := rand.Read(key)
	if err != nil {
		log.Fatalf("failed to read new random key: %s", err)
	}
	if n < size {
		log.Fatalf("failed to read entire key, only read %d out of %d", n, size)
	}
	return key
}

func SaveKey(filename string, key []byte) {
	block := &pem.Block{
		Type:  "DES KEY",
		Bytes: key,
	}
	err := ioutil.WriteFile(filename, pem.EncodeToMemory(block), 0644)
	if err != nil {
		log.Fatalf("failed saving key to %s: %s", filename, err)
	}
}

func ReadKey(filename string) ([]byte, error) {
	key, err := ioutil.ReadFile(filename)
	if err != nil {
		return key, err
	}
	block, _ := pem.Decode(key)
	return block.Bytes, nil
}

func Key() []byte {
	key, err := ReadKey(KeyFile)
	if err != nil {
		log.Println("failed reading keyfile, making a new one...")
		key = MakeKey()
		SaveKey(KeyFile, key)
	}
	return key
```

```
}

func MakeCipher() cipher.Block {
	var c cipher.Block
	var err error
	if *triple {
		c, err = des.NewTripleDESCipher(Key())
	} else {
		c, err = des.NewCipher(Key())
	}
	if err != nil {
		log.Fatalf("failed making the DES cipher: %s", err)
	}
	return c
}

func Crypt(bytes []byte) []byte {
	blockCipher := MakeCipher()
	stream := cipher.NewCTR(blockCipher, IV)
	stream.XORKeyStream(bytes, bytes)
	return bytes
}

func Encrypt() {
	encrypted := Crypt([]byte(*message))
	err := ioutil.WriteFile(EncryptedFile, encrypted, 0644)
	if err != nil {
		log.Fatalf("failed writing encrypted file: %s", err)
	}
}

func Decrypt() {
	bytes, err := ioutil.ReadFile(EncryptedFile)
	if err != nil {
		log.Fatalf("failed reading encrypted file: %s", err)
	}
	plaintext := Crypt(bytes)
	log.Printf("decrypted message: %s", plaintext)
}

func main() {
	flag.Parse()
	switch *do {
```

```
        case "encrypt":
                Encrypt()
        case "decrypt":
                Decrypt()
        default:
                log.Fatalf("%s is not a valid operation. Must be one of encrypt or decrypt", *do)
        }
}
```

Digital Signatures

Digital signature algorithms use asymmetric cryptography (with a public and private key pair) to *sign* messages. They can ensure a message came from a particular sender, and also ensure that a message was not tampered with. They also prevent somebody from later claiming they didn't sign a particular message.[24]

ECDSA

The `crypto/ecdsa` package handles the *elliptic curve* digital signature algorithm.

Cool story bro.

Anyway, it uses the `crypto/elliptic` package to do key generation. There's a whole whack of stuff behind it I'm not familiar with, so as with the other crypto things, do your own research and gain your own understanding (or just get a professional) before doing anything really interesting.

In a nutshell, you need to generate a key, which has both the public and private parts built in, hash your message, then sign it. Once you have the signature, you can verify a message using the public part of the key.

In this example, I've left out saving the key, because I'm unsure of the best way of doing it. You need to save some numbers, but also which curve was used. I could dump this out to JSON for all I care, but I'm sure there is a better, more standard way to do it.

[24] http://en.wikipedia.org/wiki/Non-repudiation

crypto/ecdsa.go

```go
package main

import (
        "crypto/ecdsa"
        "crypto/elliptic"
        "crypto/rand"
        "crypto/sha1"
        "flag"
        "io"
        "log"
)

var message = flag.String("message", "Nuke the site from orbit, it's the only way to\
 be sure.", "The message to sign")

func HashMessage() []byte {
        h := sha1.New()
        _, err := io.WriteString(h, *message)
        if err != nil {
                log.Fatalf("failed to hash message: %s", err)
        }
        return h.Sum(nil)
}

func Key() *ecdsa.PrivateKey {
        key, err := ecdsa.GenerateKey(elliptic.P521(), rand.Reader)
        if err != nil {
                log.Fatalf("failed to generate key: %s", err)
        }
        return key
}

func main() {
        flag.Parse()

        key := Key()
        hash := HashMessage()
        r, s, err := ecdsa.Sign(rand.Reader, key, hash)
        if err != nil {
                log.Fatalf("failed to sign message: %s", err)
        }
```

```
        log.Printf("r: %s", r)
        log.Printf("s: %s", s)

        if ecdsa.Verify(&key.PublicKey, hash, r, s) {
                log.Println("message is valid!")
        } else {
                log.Println("message invalid :(")
        }
}
```

DSA

`crypto/dsa` is very similar to `crypto/ecdsa` except that it has nothing to do with elliptic curves. You have to generate some parameters before generating a key, which can take a little while.[25]

The way I've done the key serialization (with `encoding/{asn1,pem}`) works with `ssh-keygen`. If you do `ssh-keygen -t dsa` and copy your `~/.ssh/id_dsa` file to dsa.key before you run the file, it will use that key and merrily carry on.

crypto/dsa.go

```
package main

import (
        "crypto/dsa"
        "crypto/rand"
        "crypto/sha1"
        "encoding/asn1"
        "encoding/pem"
        "flag"
        "io"
        "io/ioutil"
        "log"
        "math/big"
)

const (
        KeyFile = "dsa.key"
)

```

[25] According to the documentation, it can "[take] many seconds, even on fast machines."

```
var (
	message = flag.String("message", "Nuke the site from orbit, it's the only way to be\
 sure.", "The message to sign")
	do      = flag.String("do", "sign", "The operation to do, verify or sign (default)")
	rc      = flag.String("r", "", "The r to use when verifying")
	sc      = flag.String("s", "", "The s to use when verifying")
)

func HashMessage() []byte {
	h := sha1.New()
	_, err := io.WriteString(h, *message)
	if err != nil {
		log.Fatalf("failed to hash message: %s", err)
	}
	return h.Sum(nil)
}

type DsaKeyFormat struct {
	Version       int
	P, Q, G, Y, X *big.Int
}

func SaveKey(key *dsa.PrivateKey) {
	val := DsaKeyFormat{
		P: key.P, Q: key.Q, G: key.G,
		Y: key.Y, X: key.X,
	}
	bytes, err := asn1.Marshal(val)
	if err != nil {
		log.Fatalf("failed marshalling key to asn1: %s", err)
	}
	block := &pem.Block{
		Type:  "DSA PRIVATE KEY",
		Bytes: bytes,
	}
	err = ioutil.WriteFile(KeyFile, pem.EncodeToMemory(block), 0644)
	if err != nil {
		log.Fatalf("failed saving key to file %s: %s", KeyFile, err)
	}
}

func ReadKey() (*dsa.PrivateKey, error) {
	bytes, err := ioutil.ReadFile(KeyFile)
```

```
	if err != nil {
		return nil, err
	}
	block, _ := pem.Decode(bytes)
	val := new(DsaKeyFormat)
	_, err = asn1.Unmarshal(block.Bytes, val)
	if err != nil {
		return nil, err
	}
	key := &dsa.PrivateKey{
		PublicKey: dsa.PublicKey{
			Parameters: dsa.Parameters{
				P: val.P,
				Q: val.Q,
				G: val.G,
			},
			Y: val.Y,
		},
		X: val.X,
	}
	return key, nil
}

func MakeKey() *dsa.PrivateKey {
	key := new(dsa.PrivateKey)
	err := dsa.GenerateParameters(&key.Parameters, rand.Reader, dsa.L2048N256)
	if err != nil {
		log.Fatalf("failed to parameters: %s", err)
	}
	err = dsa.GenerateKey(key, rand.Reader)
	if err != nil {
		log.Fatalf("failed to generate key: %s", err)
	}
	return key
}

func Key() *dsa.PrivateKey {
	key, err := ReadKey()
	if err != nil {
		log.Printf("failed reading keyfile, making a new one: %s", err)
		key = MakeKey()
		SaveKey(key)
	}
```

```
        return key
}

func Sign() {
        key := Key()
        hash := HashMessage()
        r, s, err := dsa.Sign(rand.Reader, key, hash)
        if err != nil {
                log.Fatalf("failed to sign message: %s", err)
        }
        log.Printf("r: %v", r)
        log.Printf("s: %v", s)
}

func Verify() {
        r := big.NewInt(0)
        r.SetString(*rc, 10)

        s := big.NewInt(0)
        s.SetString(*sc, 10)

        hash := HashMessage()
        key := Key()
        if dsa.Verify(&key.PublicKey, hash, r, s) {
                log.Println("message is valid!")
        } else {
                log.Println("message is invalid :(")
                log.Println("did you use the -r and -s flags to pass the r and s values?")
        }
}

func main() {
        flag.Parse()
        switch *do {
        case "sign":
                Sign()
        case "verify":
                Verify()
        default:
                log.Fatalf("%s is not a valid operation, must be one of sign or verify", *do)
        }
}
```

Hashes

The hash functions provided by the `crypto` package are MD5, SHA1, SHA256, and SHA512. They all operate exactly the same, since they all deal with the `hash.Hash` interface. You create a new hash, write to it (`hash.Hash` implements `io.Writer`) and then get the `Sum` of it. You can `fmt.Sprintf` this to get your your standard hash-looking value. Pretty straightforward.

crypto/hash.go

```go
package main

import (
	"crypto/md5"
	"crypto/sha1"
	"crypto/sha256"
	"crypto/sha512"
	"flag"
	"hash"
	"io"
	"log"
)

var (
	algorithm = flag.String("algorithm", "md5", "The algorithm to use. Must be one of {\
md5,sha1,sha256,sha512}")
	message   = flag.String("message", "Go, The Standard Library", "The message to hash\
")
)

func GetHash() hash.Hash {
	switch *algorithm {
	case "md5":
		return md5.New()
	case "sha1":
		return sha1.New()
	case "sha256":
		return sha256.New()
	case "sha512":
		return sha512.New()
	default:
		log.Fatalf("No hash algorithm %s found", *algorithm)
	}
```

```
        panic("unreachable")
}

func main() {
        flag.Parse()
        hash := GetHash()
        io.WriteString(hash, *message)
        log.Printf("%x", hash.Sum(nil))
}
```

HMAC

HMAC isn't like the other hashes. You give it a hash function (a function that returns a `hash.Hash`) and a key in the form of a byte slice. You can then hash a message and send the result along with a message to somebody else. They can check that the message was received intact by hashing what they got and comparing that value with the one we sent.

The Crytpo Stack Overflow site has a good answer as to whether you do encrypt-then-mac or mac-then-encrypt: Should we MAC-then-encrypt or encrypt-then-MAC?[26] In my example, I use encrypt-then-mac as it's generally the better way to go.

Once you have your HMAC and encrypted data, get both of these pieces to the other party, and they can perform the same operation to verify the integrity of the encrypted data, and then decrypt the data.

I've seen other suggestions to not actually use the same key for the HMAC (just run your normal key through a hash function), and to run the HMAC on the encrypted data concatenated with the IV instead of the raw encrypted data. My gut tells me these things make sense, but I have no knowledge or math to back that intuition up. I haven't done either in the example.

[26] http://crypto.stackexchange.com/questions/202/should-we-mac-then-encrypt-or-encrypt-then-mac

crypto/hmac.go

```go
package main

import (
	"crypto/aes"
	"crypto/cipher"
	"crypto/hmac"
	"crypto/sha256"
	"flag"
	"log"
)

var (
	// 32 byte key for AES256, made from crypto/rand
	key     = []byte{0x98, 0x39, 0xea, 0x42, 0xd0, 0x3e, 0x36, 0x6b, 0xe3, 0x7b, 0x91, \
0x6, 0x50, 0x5b, 0x7f, 0xc9, 0x93, 0x56, 0xaa, 0xa8, 0x96, 0x33, 0x7, 0xd7, 0xf7, 0x\
50, 0xa5, 0x3a, 0xdc, 0x8e, 0xe2, 0x9f}
	iv      = []byte("batman and robin") // 16 bytes
	message = flag.String("message", "Batman and Robin are coming", "The message to use\
")
)

func main() {
	flag.Parse()
	block, err := aes.NewCipher(key)
	if err != nil {
		log.Fatalf("failed making AES block cipher: %s", err)
	}
	bytes := []byte(*message)
	stream := cipher.NewCTR(block, iv)
	stream.XORKeyStream(bytes, bytes)
	hash := hmac.New(sha256.New, key)
	hash.Write(bytes)
	log.Printf("message: %s", *message)
	log.Printf("encrypted message (raw bytes): %v", bytes)
	log.Printf("HMAC: %x", hash.Sum(nil))
}
```

RC4

RC4 is a widely used stream cipher algorithm. When I say widely used, I mean WEP, WPA, SSL, RDP, BitTorrent, etc. It's kind of a big deal. It does have problems though. For more specifics on the algorithm, other uses, and problems, I'd recommend starting with the Wikipedia page[27].

The Go documentation points out that it's a "poor choice to use for new protocols".

It's quite simple to use however. With your key of 10-256 bytes, simply make the cipher and use the `XORKeyStream` method to encrypt/decrypt data. The documentation suggests the `src` and `dst` shouldn't overlap, but I tried in the example and it worked pretty well.

crypto/rc4.go

```
package main

import (
	"crypto/rand"
	"crypto/rc4"
	"encoding/pem"
	"flag"
	"io/ioutil"
	"log"
)

const (
	EncryptedFile = "rc4.enc"
	KeyFile       = "rc4.key"
)

var (
	do      = flag.String("do", "encrypt", "The operation to perform, decrypt or encryp\
t (default)")
	message = flag.String("message", "Wolverines attack at dawn. Red Dawn.", "The messa\
ge to encrypt")
	keySize = flag.Int("keysize", 256, "Key size in bytes")
)

func MakeKey() []byte {
	key := make([]byte, *keySize)
```

[27] http://en.wikipedia.org/wiki/Rc4#RC4-based_cryptosystems

```
	n, err := rand.Read(key)
	if err != nil {
		log.Fatalf("failed to read new random key: %s", err)
	}
	if n < *keySize {
		log.Fatalf("failed to read entire key, only read %d out of %d", n, *keySize)
	}
	return key
}

func SaveKey(filename string, key []byte) {
	block := &pem.Block{
		Type:  "RC4 KEY",
		Bytes: key,
	}
	err := ioutil.WriteFile(filename, pem.EncodeToMemory(block), 0644)
	if err != nil {
		log.Fatalf("failed saving key to %s: %s", filename, err)
	}
}

func ReadKey(filename string) ([]byte, error) {
	key, err := ioutil.ReadFile(filename)
	if err != nil {
		return key, err
	}
	block, _ := pem.Decode(key)
	return block.Bytes, nil
}

func Key() []byte {
	key, err := ReadKey(KeyFile)
	if err != nil {
		log.Println("failed reading key, making a new one...")
		key = MakeKey()
		SaveKey(KeyFile, key)
	}
	return key
}

func Cipher() *rc4.Cipher {
	key := Key()
	cipher, err := rc4.NewCipher(key)
```

```
	if err != nil {
		log.Fatalf("failed to make RC4 cipher: %s", err)
	}
	return cipher
}

func Encrypt() {
	cipher := Cipher()
	text := []byte(*message)
	cipher.XORKeyStream(text, text)
	err := ioutil.WriteFile(EncryptedFile, text, 0644)
	if err != nil {
		log.Fatalf("failed to write encrypted file: %s", err)
	}
}

func Decrypt() {
	cipher := Cipher()
	bytes, err := ioutil.ReadFile(EncryptedFile)
	if err != nil {
		log.Fatalf("failed to read encrypted file. Did you encrypt first? %s", err)
	}
	cipher.XORKeyStream(bytes, bytes)
	log.Printf("decrypted message: %s", bytes)
}

func main() {
	flag.Parse()
	switch *do {
	case "encrypt":
		Encrypt()
	case "decrypt":
		Decrypt()
	default:
		log.Fatalf("%s not a valid operation. Must be one of encrypt or decrypt", *do)
	}
}
```

RSA

RSA is a public key encryption algorithm. It can be used to encrypt messages, where you can encrypt something with my public key, and then only I can read the message by decrypting it with the private half of the key. It can also be used to sign messages so that I can use your public key to be certain that the message did in fact come from you.

As the documentation states, you should be using OAEP instead of PKCS1v15 for new protocols.

As with the `crypto/dsa` example, you could use your existing RSA key from `~/.ssh/id_rsa`, just copy it to `rsa.key`.

crypto/rsa.go

```go
package main

import (
	"crypto"
	"crypto/md5"
	"crypto/rand"
	"crypto/rsa"
	"crypto/sha1"
	"crypto/sha256"
	"crypto/sha512"
	"crypto/x509"
	"encoding/pem"
	"flag"
	"hash"
	"io/ioutil"
	"log"
)

const (
	KeyFile       = "rsa.key"
	SignatureFile = "rsa.sig"
	EncryptedFile = "rsa.enc"
)

var (
	keySize       = flag.Int("keysize", 2048, "The size of the key in bits")
	do            = flag.String("do", "encrypt", "The operation to perform, decrypt or \
encrypt (default)")
```

```
        message       = flag.String("message", "The revolution has begun!", "The message to\
 encrypt")
        hashAlgorithm = flag.String("algorithm", "sha256", "The hash algorithm to use. Must\
 be one of md5, sha1, sha256 (default), sha512")
)

func MakeKey() *rsa.PrivateKey {
        key, err := rsa.GenerateKey(rand.Reader, *keySize)
        if err != nil {
                log.Fatalf("failed to create RSA key: %s", err)
        }
        return key
}

func SaveKey(filename string, key *rsa.PrivateKey) {
        block := &pem.Block{
                Type:  "RSA PRIVATE KEY",
                Bytes: x509.MarshalPKCS1PrivateKey(key),
        }
        err := ioutil.WriteFile(filename, pem.EncodeToMemory(block), 0644)
        if err != nil {
                log.Fatalf("failed saving key to %s: %s", filename, err)
        }
}

func ReadKey(filename string) (*rsa.PrivateKey, error) {
        bytes, err := ioutil.ReadFile(filename)
        if err != nil {
                return nil, err
        }
        block, _ := pem.Decode(bytes)
        key, err := x509.ParsePKCS1PrivateKey(block.Bytes)
        if err != nil {
                return nil, err
        }
        return key, nil
}

func Key() *rsa.PrivateKey {
        key, err := ReadKey(KeyFile)
        if err != nil {
                log.Printf("failed to read key, creating a new one: %s", err)
                key = MakeKey()
```

```
                SaveKey(KeyFile, key)
        }
        return key
}

func HashAlgorithm() (hash.Hash, crypto.Hash) {
        switch *hashAlgorithm {
        case "md5":
                return md5.New(), crypto.MD5
        case "sha1":
                return sha1.New(), crypto.SHA1
        case "sha256":
                return sha256.New(), crypto.SHA256
        case "sha512":
                return sha512.New(), crypto.SHA512
        default:
                log.Fatalf("%s is not a valid hash algorithm. Must be one of md5, sha1, sha256, sh\
a512")
        }
        panic("not reachable")
}

func HashMessage(data []byte) []byte {
        h, _ := HashAlgorithm()
        h.Write(data)
        return h.Sum(nil)
}

func Encrypt() {
        h, ha := HashAlgorithm()
        key := Key()
        encrypted, err := rsa.EncryptOAEP(h, rand.Reader, &key.PublicKey, []byte(*message),\
 nil)
        if err != nil {
                log.Fatalf("encryption failed: %s", err)
        }
        signature, err := rsa.SignPKCS1v15(rand.Reader, key, ha, HashMessage(encrypted))
        if err != nil {
                log.Fatalf("signing failed; %s", err)
        }
        err = ioutil.WriteFile(EncryptedFile, encrypted, 0644)
        if err != nil {
                log.Fatalf("failed saving encrypted data: %s", err)
```

```
	}
	err = ioutil.WriteFile(SignatureFile, signature, 0644)
	if err != nil {
		log.Fatalf("failed saving signature data: %s", err)
	}
}

func Decrypt() {
	key := Key()
	h, ha := HashAlgorithm()
	encrypted, err := ioutil.ReadFile(EncryptedFile)
	if err != nil {
		log.Fatalf("failed reading encrypted data: %s", err)
	}

	signature, err := ioutil.ReadFile(SignatureFile)
	if err != nil {
		log.Fatalf("failed saving signature data: %s", err)
	}

	if err = rsa.VerifyPKCS1v15(&key.PublicKey, ha, HashMessage(encrypted), signature);\
 err != nil {
		log.Fatalf("message not valid: %s", err)
	} else {
		log.Printf("message is valid!")
	}

	plaintext, err := rsa.DecryptOAEP(h, rand.Reader, key, encrypted, nil)
	if err != nil {
		log.Fatalf("failed decrypting: %s", err)
	}
	log.Printf("decrypted message: %s", plaintext)
}

func main() {
	flag.Parse()
	switch *do {
	case "encrypt":
		Encrypt()
	case "decrypt":
		Decrypt()
	default:
		log.Fatalf("%s is not a valid operation. Must be one of encrypt or decrypt")
```

```
        }
}
```

TLS/x509

The `crypto/tls` and `crypto/x509` packages provide a lot of functionality surrounding their respective topics. I'm not going to cover everything, but we'll look at a few basic things like generating, serializing and parsing certificates, and creating a simple echo server.

After the server starts, connect with the command it gives and type into the console and have it echoed back to you. Make sure to pass the `-tls1` flag to the `openssl s_client` command.

crypto/tls_x509.go

```go
package main

import (
        "crypto/rand"
        "crypto/rsa"
        "crypto/tls"
        "crypto/x509"
        "crypto/x509/pkix"
        "encoding/pem"
        "flag"
        "io"
        "io/ioutil"
        "log"
        "math/big"
        "net"
        "time"
)

const (
        CertFile = "tls.crt"
        KeyFile  = "tls.key"
)

var (
        do      = flag.String("do", "serve", "The operation to perform, key, cert, or serve\
```

```
 (default)")
        keySize = flag.Int("keysize", 2048, "The RSA keysize to use")
)

func MakeKey() *rsa.PrivateKey {
        key, err := rsa.GenerateKey(rand.Reader, *keySize)
        if err != nil {
                log.Fatalf("failed to create RSA key: %s", err)
        }
        return key
}

func PemEncodeKey(key *rsa.PrivateKey) []byte {
        block := &pem.Block{
                Type:  "RSA PRIVATE KEY",
                Bytes: x509.MarshalPKCS1PrivateKey(key),
        }
        return pem.EncodeToMemory(block)
}

func SaveKey(filename string, key *rsa.PrivateKey) {
        err := ioutil.WriteFile(filename, PemEncodeKey(key), 0644)
        if err != nil {
                log.Fatalf("failed saving key to %s: %s", filename, err)
        }
}

func ReadKey(filename string) (*rsa.PrivateKey, error) {
        bytes, err := ioutil.ReadFile(filename)
        if err != nil {
                return nil, err
        }
        block, _ := pem.Decode(bytes)
        key, err := x509.ParsePKCS1PrivateKey(block.Bytes)
        if err != nil {
                return nil, err
        }
        return key, nil
}

func Key() *rsa.PrivateKey {
        key, err := ReadKey(KeyFile)
        if err != nil {
```

```
			log.Printf("failed to read key, creating a new one: %s", err)
			key = MakeKey()
			SaveKey(KeyFile, key)
		}
		return key
}

func SaveCert(filename string, cert []byte) []byte {
		block := &pem.Block{
			Type:  "CERTIFICATE",
			Bytes: cert,
		}
		bytes := pem.EncodeToMemory(block)
		err := ioutil.WriteFile(filename, bytes, 0644)
		if err != nil {
			log.Fatalf("failed saving cert to %s: %s", filename, err)
		}
		return bytes
}

func MakeCert() tls.Certificate {
		key := Key()
		now := time.Now()
		template := &x509.Certificate{
			SerialNumber: big.NewInt(1),
			Subject: pkix.Name{
				Country:            []string{"CA"},
				Province:           []string{"Alberta"},
				Locality:           []string{"Edmonton"},
				Organization:       []string{"The Standard Library"},
				OrganizationalUnit: []string{"Go, The Standard Library"},
				CommonName:         "localhost",
			},
			NotBefore: now,
			NotAfter:  now.Add(24 * 365 * time.Hour), // 1 year
			KeyUsage:  0,
		}
		cert, err := x509.CreateCertificate(rand.Reader, template, template, &key.PublicKey\
, key)
		if err != nil {
			log.Fatalf("failed creating certificate: %s", err)
		}
		cert = SaveCert(CertFile, cert)
```

```
	c, err := tls.X509KeyPair(cert, PemEncodeKey(key))
	if err != nil {
		log.Fatalf("failed to load certificate: %s", err)
	}
	return c
}

func Cert() tls.Certificate {
	cert, err := tls.LoadX509KeyPair(CertFile, KeyFile)
	if err != nil {
		log.Printf("failed loading certificate, generating a new one: %s", err)
		cert = MakeCert()
	}
	return cert
}

func Config() *tls.Config {
	return &tls.Config{
		Certificates: []tls.Certificate{Cert()},
	}
}

func Serve() {
	addr := "localhost:4443"
	conn, err := net.Listen("tcp", addr)
	if err != nil {
		log.Fatalf("failed to listen on %s: %s", addr, err)
	}

	config := Config()
	listener := tls.NewListener(conn, config)
	log.Printf("listening on %s, connect with 'openssl s_client -tls1 -connect %s'", ad\
dr, addr)
	for {
		conn, err := listener.Accept()
		if err != nil {
			log.Fatalf("failed to accept: %s", err)
		}
		log.Printf("connection accepted from %s", conn.RemoteAddr())
		go func(c net.Conn) {
			_, err := io.Copy(c, c)
			if err != nil {
				log.Printf("error copying: %s", err)
```

```
                        }
                        log.Println("closing connection")
                        c.Close()
                }(conn)
        }
}

func main() {
        flag.Parse()
        switch *do {
        case "serve":
                Serve()
        case "cert":
                Cert()
        case "key":
                Key()
        default:
                log.Fatalf("%s is not a valid operation, must be one of serve, cert, or key", *do)
        }
}
```

Random Numbers

You've already seen the `crypto/rand` package used in all the examples in this chapter.

The package only has 3 methods and one variable. We've been using the `rand.Reader` for pretty much everything. It gives you an `io.Reader` that reads from `/dev/urandom` or the `CryptGenRandom` API depending on the platform.

The `rand.Read` function delegates to the `rand.Reader` variable.

`rand.Int` gives you a random int, in the form of a `big.Int`, and `rand.Prime` gives you a random prime number.[28]

[28] Well, as the documentation says, "[it] returns a number, p, of the given size, such that p is prime with high probability."

crypto/rand.go

```
package main

import (
	"crypto/rand"
	"flag"
	"log"
	"math/big"
)

var (
	iterations = flag.Int("iterations", 3, "The number of iterations to run on each thi\
ng")
	bits       = flag.Int("bits", 16, "The number of bits to use when generating a rand\
om prime")
	max        = flag.Int64("max", 256, "The max value to use when generating a random \
int")
)

func ShowInt() {
	for i := 0; i < *iterations; i++ {
		if n, err := rand.Int(rand.Reader, big.NewInt(*max)); err != nil {
			log.Fatalf("failed to read random int: %s", err)
		} else {
			log.Printf("got random int: %s", n)
		}
	}
}

func ShowPrime() {
	for i := 0; i < *iterations; i++ {
		if p, err := rand.Prime(rand.Reader, *bits); err != nil {
			log.Fatalf("failed to read random prime: %s", err)
		} else {
			log.Printf("got random prime: %s", p)
		}
	}
}

func ShowRead() {
	for i := 0; i < *iterations; i++ {
		bytes := make([]byte, 16)
```

```go
		if n, err := rand.Read(bytes); err != nil {
			log.Printf("failed reading random bytes: %s", err)
		} else {
			log.Printf("read %d bytes: %v", n, bytes[0:n])
		}
	}
}

func main() {
	flag.Parse()
	ShowInt()
	ShowPrime()
	ShowRead()
}
```

Constant Time Functions

The `crypto/subtle` package gives you a few function to do operations in constant time. Constant time comparisons are an important part of cryptography, as they help prevent timing attacks[29].

crypto/crypto.go

```go
package main

import (
	"crypto/subtle"
	"log"
)

func main() {
	log.Printf("%d", subtle.ConstantTimeByteEq(43, 65))
	log.Printf("%d", subtle.ConstantTimeCompare([]byte("batman"), []byte("robin ")))

	bytes := make([]byte, 6)
	subtle.ConstantTimeCopy(1, bytes, []byte("batman"))
	log.Printf("%s", bytes)

	log.Printf("%d", subtle.ConstantTimeEq(256, 255))
```

[29] http://en.wikipedia.org/wiki/Timing_attack

```
        log.Printf("%d", subtle.ConstantTimeSelect(1, 2, 3))
        log.Printf("%d", subtle.ConstantTimeSelect(0, 2, 3))
}
```

A Timing Attack In Action

This example shows how a timing attack could work. I'm just calling the function, but that could easily be replaced with making a web request or something else.

If you run the file without any arguments you can see it run through possible guesses for the password, where one letter takes a bit longer. Each letter that takes a little bit longer than the others is the best guess for that index. The last letter is trickier, but once you have the password solved except for that last letter, it's not a big deal to figure out that last letter. In the case of the example, it's downright obvious.

crypto/timing_attack.go

```
package main

import (
        "container/heap"
        "crypto/subtle"
        "flag"
        "log"
        T "testing"
        "time"
)

var (
        password   = flag.String("password", "secret", "The password to try and guess")
        characters = flag.String("characters", "abcdefghijklmnopqrstuvwxyz", "The set of ch\
aracters to use")
        compare    = flag.String("compare", "broken", "The comparison function to use. Must\
 be one of constant or broken (default)")
)

type TestRun struct {
        Time int64
        Byte byte
}
```

```

type Times []TestRun

func (t Times) Len() int           { return len(t) }
func (t Times) Less(i, j int) bool { return t[i].Time > t[j].Time }
func (t Times) Swap(i, j int)      { t[i], t[j] = t[j], t[i] }

func (t *Times) Push(v interface{}) {
	*t = append(*t, v.(TestRun))
}

func (t *Times) Pop() interface{} {
	a := *t
	n := len(a)
	v := a[n-1]
	*t = a[0 : n-1]
	return v
}

type Compare func(x, y []byte) int

func BrokenCompare(x, y []byte) int {
	for i := range x {
		if x[i] != y[i] {
			return 0
		}
	}
	return 1
}

func Crack(password []byte, comp Compare) []byte {
	n := len(password)
	guess := make([]byte, n)
	for index := range password {
		times := make(Times, 0)
		for _, letter := range []byte(*characters) {
			guess[index] = letter
			result := T.Benchmark(func(b *T.B) {
				for i := 0; i < b.N; i++ {
					comp(password, guess)
				}
			})
			heap.Push(&times, TestRun{
```

```
                                Time: result.NsPerOp(),
                                Byte: letter,
                        })
                        log.Printf("took %s (%d ns/op) to try %q for index %d", result.T, result.NsP
), letter, index)
                }
                tr := heap.Pop(&times).(TestRun)
                guess[index] = tr.Byte
                log.Printf("best guess is %q for index %d", tr.Byte, index)
                log.Printf("guess is now: %s", guess)
        }
        return guess
}

func ConstantTimeCrack(pw []byte) []byte {
        return Crack(pw, subtle.ConstantTimeCompare)
}

func BrokenCrack(pw []byte) []byte {
        return Crack(pw, BrokenCompare)
}

func main() {
        flag.Parse()
        var guess []byte
        pw := []byte(*password)
        start := time.Now()
        switch *compare {
        case "broken":
                log.Println("using broken compare function")
                guess = BrokenCrack(pw)
        case "constant":
                log.Println("using constant time compare function")
                guess = ConstantTimeCrack(pw)
        default:
                log.Fatalf("%s is not a valid compare function. Must be one of broken or constant")
        }
        end := time.Now()
        dur := end.Sub(start)
        log.Printf("password guess after %s is: %s", dur, guess)
}
```

go.crypto

The `go.crypto` package contains packages that will most likely be in the standard library at some point, but just aren't quite finalized yet

It is ready to use however, and has a number of great packages, a few of my favorites being `pbkdf2`, `bcrypt`, `blowfish`, `twofish`, and `openpgp`.

You can use it with `import "code.google.com/p/go.crypto"`‘ and the documentation can be found on GoPkgDoc[30].

Final Warning

As I said before, I'm not a cryptographer. Don't blame me if you copy and paste something out of here and a script kiddy steals all your stuff. I believe what I've said to be accurate in the usage of the APIs provided by the Go programming language. If you know better, please let me know so I can fix the contents of this book.

Writing this chapter has given me a renewed interest in cryptography, so I think I'll dust off my copy of Applied Cryptography[31].

[30] http://go.pkgdoc.org/code.google.com/p/go.crypto
[31] http://www.amazon.com/Applied-Cryptography-Protocols-Algorithms-ebook/dp/B000SEHPK6

database

The `database` package is for handling, well, database things. Right now, the only sub-package is `database/sql` which provides a nice interface for dealing with relational databases.

You can't do anything with it on its own though, you need a **driver**. On the *go-wiki*, they have a list of solid drivers for the `database/sql`.[32]

We'll be using the `sqlite3` driver at https://github.com/mattn/go-sqlite3[33] so before running the examples, install it with `go get github.com/mattn/go-sqlite3`.

Some of the specifics are different between databases and drivers, so the example might not work with another database or another driver.

Importing the driver is a little different than normal, since you just want to make sure the driver's `init` function is called to register the driver with the `database/sql` package. You import it with an underscore, which forces the `init` function to run, but doesn't actually import the package into the namespace.

Open

All your database interactions start with using `sql.Open` to get a handle to the database. As per the docs, you can share the handle between goroutines. Also as per the docs, if the specific driver supports it, the `database/sql` can manage connections and connection-state when it comes to transactions.

You may be thinking just open the database once and use that handle throughout the lifetime of your application. Not so fast, sport! The problem you might run into (again, depending on the driver) is that the connection is lost or times out, or something along those lines. The next time you try to do something with it you'll get back an error and the handle will effectively be dead. You'll have to get a new handle with `sql.Open` again. You can test this by opening a connection, making a query, then stopping and starting the database process, and trying another query. It will probably fail.

[32] http://code.google.com/p/go-wiki/wiki/SQLDrivers
[33] https://github.com/mattn/go-sqlite3

Try using postgres as your database in the example, but add a sleep between two of the main calls. It will fail, and the connection is effectively dead.

Keep this in mind. You may be better off opening and closing the handle to the database.

Exec

`Exec` is for doing things that don't really return anything, like inserting, deleting, and doing schema changes. It returns a `sql.Result` and an `error`. The `sql.Result` type can give you some basic information like `RowsAffected` and `LastInsertId` and the `error` gives you, well, error information.

Query

Using `DB.QueryRow` and `DB.Query` you can pull out a single row, or multiple rows. With a single row, you can scan directly into things (provided there was no error), but when querying multiple rows you have to iterate over the `sql.Rows` struct using `Rows.Next` to get everything.

Don't forget to check `Rows.Err` at the end to see if there were any problems iterating.

When querying multiple rows, you can also get the column names using `Rows.Columns` method.

Note that the exact query syntax depends on the driver and the database. In the sqlite example, I use `?` to denote where an argument would go, while in the postgres example I use `$1` and `$2` (and so on) to do the same thing. If you try to use question marks in the postgres example, you'll get an interesting error message.

Prepared Statements

Prepared statements allow you to make one statement and re-execute it with different arguments. For example, you can make an `INSERT` statement, and iterate over all the things you need to insert, just passing in the different values. You can usually realize some performance improvements doing this.

Use `DB.Stmt` to create a `sql.Stmt` struct and then the normal `QueryRow`, `Query`, and `Exec` methods to take care of business.

Transactions

You can start a transaction with `DB.Begin`. This gives you a `sql.Tx` struct, which has the usual array of methods: `Tx.Exec`, `Tx.QueryRow`, `Tx.Query`, `Tx.Prepare`. It also has three others.

`Tx.Commit` will cause the transaction to be committed. You might get an `error` back. `Tx.Rollback` will cause the transaction to be aborted, causing no changes to the database. It might also give you an `error` back.

`Stmt` takes a `sql.Stmt` and makes it specific to this transaction, giving you another `sql.Stmt`. No big deal.

Example

Now, as the saying goes, let's go for the gusto.

database/sql.go

```go
package main

import (
	"database/sql"
	"flag"

	_ "github.com/mattn/go-sqlite3"
	// An example if you want to use Postgres
	// _ "github.com/bmizerany/pq"
	"log"
)

var rollback = flag.Bool("rollback", false, "Rollback in the insert transaction")

func init() {
	log.SetFlags(0)
	log.SetPrefix("» ")
```

```
}

type Show struct {
	Name, Country string
}

func openSqlite() (*sql.DB, error) {
	return sql.Open("sqlite3", "go-thestdlib.db")
}

func openPostgres() (*sql.DB, error) {
	return sql.Open("postgres", "user=bob password=secret host=1.2.3.4 port=5432 dbname\
=mydb sslmode=verify-full")
}

func openDB() *sql.DB {
	db, err := openSqlite()
	// db, err := openPostgres()
	if err != nil {
		log.Fatalf("failed opening database: %s", err)
	}
	return db
}

func removeTable(db *sql.DB) {
	_, err := db.Exec("DROP TABLE IF EXISTS shows")
	if err != nil {
		log.Fatalf("failed dropping table: %s", err)
	} else {
		log.Println("dropped table (if it existed) shows")
	}
}

func createTable(db *sql.DB) {
	_, err := db.Exec("CREATE TABLE shows (name TEXT, country TEXT)")
	if err != nil {
		log.Fatalf("failed creating table: %s", err)
	} else {
		log.Println("created table shows")
	}
}

func insertRow(db *sql.DB) {
```

```
	// For postgres we use $1 and $2 instead of ?
	res, err := db.Exec("INSERT INTO shows (name, country) VALUES (?, ?)", "Nöjesmaskin\
en", "SE")
	if err != nil {
		log.Fatalf("failed inserting Swedish show: %s", err)
	} else {
		log.Println("inserted 1 Swedish TV show")
	}

	if id, err := res.LastInsertId(); err != nil {
		log.Printf("failed retrieving LastInsertId: %s", err)
	} else {
		log.Printf("LastInsertId: %d", id)
	}

	if n, err := res.RowsAffected(); err != nil {
		log.Printf("failed retrieving RowsAffected: %s", err)
	} else {
		log.Printf("RowsAffected: %d", n)
	}
}

func insertRows(db *sql.DB) {
	tx, err := db.Begin()
	if err != nil {
		log.Fatalf("failed starting transaction: %s", err)
	}

	shows := []Show{
		Show{"Top Gear", "UK"},
		Show{"Wilfred", "AU"},
		Show{"Top Gear", "US"},
		Show{"Arctic Air", "CA"},
	}

	stmt, err := tx.Prepare("INSERT INTO shows (name, country) VALUES (?, ?)")
	if err != nil {
		log.Fatalf("failed preparing statement: %s", err)
	}

	for _, show := range shows {
		_, err := stmt.Exec(show.Name, show.Country)
		if err != nil {
```

```
                        log.Fatalf("failed insert show %s (%s): %s", show.Name, show.Country, err)
                } else {
                        log.Printf("inserted show %#v for country %#v", show.Name, show.Country)
                }
        }

        if *rollback {
                if err := tx.Rollback(); err != nil {
                        log.Fatalf("failed rolling back transaction: %s", err)
                } else {
                        log.Println("rolled back transaction, nothing inserted")
                }
        } else {
                if err := tx.Commit(); err != nil {
                        log.Fatalf("failed committing transaction: %s", err)
                } else {
                        log.Println("committed transaction, 4 new shows added")
                }
        }
}

func queryCount(db *sql.DB) {
        row := db.QueryRow("SELECT COUNT(*) FROM shows")
        var count int
        if err := row.Scan(&count); err != nil {
                log.Fatalf("failed getting count: %s", err)
        }
        log.Printf("there are %d TV shows in the database", count)
}

func queryRow(db *sql.DB) {
        row := db.QueryRow("SELECT * FROM shows WHERE country = ? LIMIT 1", "CA")
        show := Show{}
        if err := row.Scan(&show.Name, &show.Country); err != nil {
                log.Printf("failed scanning single row: %s", err)
        } else {
                log.Printf("Found 1 %s TV show: %s", show.Country, show.Name)
        }
}

func queryRows(db *sql.DB) {
        name := "Top Gear"
        rows, err := db.Query("SELECT * FROM shows WHERE name = ?", name)
```

```
	if err != nil {
		log.Fatalf("failed querying multiple rows: %s", err)
	}
	shows := make([]Show, 0)
	for rows.Next() {
		show := Show{}
		if err := rows.Scan(&show.Name, &show.Country); err != nil {
			log.Fatalf("failed scanning row: %s", err)
		}
		shows = append(shows, show)
	}
	log.Printf("found %d shows named %#v", len(shows), name)
	for _, show := range shows {
		log.Printf("\t...in country %s", show.Country)
	}
	if err := rows.Err(); err != nil {
		log.Fatalf("got unexpected error during iteration: %s", err)
	}
}

func deleteRows(db *sql.DB) {
	_, err := db.Exec("DELETE FROM shows")
	if err != nil {
		log.Fatalf("failed deleting rows: %s", err)
	}
}

func main() {
	flag.Parse()
	db := openDB()
	defer db.Close()

	removeTable(db)
	createTable(db)
	insertRow(db)
	insertRows(db)
	queryCount(db)
	queryRow(db)
	// Sleep here...
	queryRows(db)
	deleteRows(db)
}
```

Output:

```
» dropped table (if it existed) shows
» created table shows
» inserted 1 Swedish TV show
» LastInsertId: 1
» RowsAffected: 1
» inserted show "Top Gear" for country "UK"
» inserted show "Wilfred" for country "AU"
» inserted show "Top Gear" for country "US"
» inserted show "Arctic Air" for country "CA"
» committed transaction, 4 new shows added
» there are 5 TV shows in the database
» Found 1 CA TV show: Arctic Air
» found 2 shows named "Top Gear"
»        ...in country UK
»        ...in country US
```

debug

The `debug` package is just a high level package holding other more useful subpackages. Inside it you'll find packages to deal with ELF[34], Mach-O files[35], and Windows PE[36] files.

On top of those 3 standards, you can of course look at Go files created by the standard gc compiler.

Finally, you can extract and investigate DWARF[37] debugging information.

elf

The `debug/elf` package lets you open up and play with ELF files. ELF, or the Executable and Linkable Format, is "a common standard file format for executables, object code, shared libraries, and core dumps."[38] The list of `Machine` constants in the package gives you an idea of how many actual machine types run this format.

The example is fairly simple, though it does touch most of the file so you can see what's there. This isn't a library that you'll use daily, but if you do, I'm sure you'll know more about the ELF format than I do already. If that's the case, you'll know what things to poke at.

debug/elf.go

```go
package main

import (
	"debug/elf"
	"log"
	"math/rand"
	"time"
)

func init() {
```

[34] http://en.wikipedia.org/wiki/Executable_and_Linkable_Format
[35] http://en.wikipedia.org/wiki/Mach-O
[36] http://en.wikipedia.org/wiki/Portable_Executable
[37] http://en.wikipedia.org/wiki/DWARF
[38] http://en.wikipedia.org/wiki/Executable_and_Linkable_Format

```
        rand.Seed(time.Now().UnixNano())
}

func printHeader(fh *elf.FileHeader) {
        log.Printf("fh.Class: %s", fh.Class)
        log.Printf("fh.Data: %s", fh.Data)
        log.Printf("fh.Version: %s", fh.Version)
        log.Printf("fh.OSABI: %s", fh.OSABI)
        log.Printf("fh.ABIVersion: %#x", fh.ABIVersion)
        log.Printf("fh.ByteOrder: %s", fh.ByteOrder)
        log.Printf("fh.Type: %s", fh.Type)
        log.Printf("fh.Machine: %s", fh.Machine)
}

func printSection(s *elf.Section) {
        log.Printf("section [Type: %s, Flags, %s, Addr: %#x, Offset: %#x, Size: %#x, Link: \
%#x, Info: %#x, Addralign: %#x, Entsize: %#x]", s.Type, s.Flags, s.Addr, s.Offset, s\
.Size, s.Link, s.Info, s.Addralign, s.Entsize)
}

func printProgramHeader(p *elf.Prog) {
        log.Printf("program header [Type: %s, Flags: %s, Off: %#x, Vaddr: %#x, Filesz: %#x,\
 Memsz: %#x, Align: %#x]", p.Type, p.Flags, p.Off, p.Vaddr, p.Filesz, p.Memsz, p.Ali\
gn)
}

func printSections(s []*elf.Section) {
        log.Printf("file has %d sections", len(s))
        for _, section := range s {
                printSection(section)
        }
}

func printProgs(p []*elf.Prog) {
        log.Printf("file has %d program headers", len(p))
        for _, prog := range p {
                printProgramHeader(prog)
        }
}

func printImportedLibraries(libs []string, err error) {
        if err != nil {
                log.Printf("failed getting imported libraries: %s", err)
```

```go
	} else {
		log.Printf("file imports %d libraries: %s", len(libs), libs)
	}
}

func printSymbols(symbols []elf.Symbol, err error) {
	if err != nil {
		log.Printf("no symbols: %s", err)
	} else {
		// Grab about 1% of the symbols
		symbolSelection := make([]string, 0, 20)
		for _, symbol := range symbols {
			if rand.Float32() <= 0.01 {
				symbolSelection = append(symbolSelection, symbol.Name)
			}
		}
		log.Printf("there are %d symbols, printing %d of them", len(symbols), len(symbolSe\
lection))
		log.Printf("a selection of symbols: %v", symbols)
	}
}

func printImportedSymbols(importedSymbols []elf.ImportedSymbol, err error) {
	if err != nil {
		log.Printf("no imported symbols: %s", err)
	} else {
		importedSymbolSelection := make([]string, 0, 20)
		for _, symbol := range importedSymbols {
			if rand.Float32() <= 0.1 {
				importedSymbolSelection = append(importedSymbolSelection, symbol.Na
ymbol.Library+",")
			}
		}
		log.Printf("there are %d imported symbols, printing %d of them", len(importedSymbo\
ls), len(importedSymbolSelection))
		log.Printf("a selection of imported symbols: %v", importedSymbolSelection)
	}
}

func printFileInformation(f *elf.File) {
	printHeader(&f.FileHeader)
	printSections(f.Sections)
	printProgs(f.Progs)
```

```go
        printImportedLibraries(f.ImportedLibraries())
        printSymbols(f.Symbols())
        printImportedSymbols(f.ImportedSymbols())
}

func main() {
        file, err := elf.Open("bash.elf")
        if err != nil {
                log.Fatalf("failed opening file: %s", err)
        }
        defer file.Close()
        printFileInformation(file)
}
```

macho

The `debug/macho` package is used for dealing with, you guessed it, Mach-O you'd find on your MacBook.

A limitation I found right away is that it doesn't load universal binaries[39] on its own. I tried to include and use the provided `bash` binary, but since it's universal it gave me errors right away. I had to use a single architecture binary for this to work. Universal binaries are basically just the separate binary blobs glued together in a special archive, so it shouldn't be terribly hard to read a file and pull out the individual parts.

In this package we start to see some discrepancies between the Go API and what the Mach-O file format on the Apple developer website[40]. For example, in the Go code, there are only two values for the `macho.Type` field in the `FileHeader`: executable and object. The Apple doc lists 8 different values. Okay that's fine, not a big deal, it just means you have to do a bit more work to check the type of your file once it's loaded instead of using the `macho.Type` constants. The file will load just fine, you'll just have to make your own constants. No big deal.

Another point, the `Flags` field in the `FileHeader` doesn't have any constants for it. If you want to check specific flags, you'll have to poke through `loader.h` in the macho source and the Apple docs to see what values are what to figure out what you want

[39] http://en.wikipedia.org/wiki/Universal_binary

[40] https://developer.apple.com/library/mac/#documentation/DeveloperTools/Conceptual/MachORuntime/Reference/reference.html

to check for. I've done exactly that in the example (I copy/pasted directly from the source, and modified slightly for Go).

Like the ELF example, this one isn't as fully featured as some examples in previous chapters, because you can do a lot with the information you get. You probably won't need to use this library in your day to day usage of Go either, but it should be enough to get you investigating the library if you have a specific use case.

debug/macho.go

```
package main

import (
	"debug/macho"
	"log"
	"math/rand"
)

const (
	MH_NOUNDEFS uint32 = 1 << iota /* the object file has no undefined
	   references */
	MH_INCRLINK /* the object file is the output of an
	   incremental link against a base file
	   and can't be link edited again */
	MH_DYLDLINK /* the object file is input for the
	   dynamic linker and can't be staticly
	   link edited again */
	MH_BINDATLOAD /* the object file's undefined
	   references are bound by the dynamic
	   linker when loaded. */
	MH_PREBOUND /* the file has its dynamic undefined
	   references prebound. */
	MH_SPLIT_SEGS /* the file has its read-only and
	   read-write segments split */
	MH_LAZY_INIT /* the shared library init routine is
	   to be run lazily via catching memory
	   faults to its writeable segments
	   (obsolete) */
	MH_TWOLEVEL /* the image is using two-level name
	   space bindings */
	MH_FORCE_FLAT /* the executable is forcing all images
	   to use flat name space bindings */
	MH_NOMULTIDEFS /* this umbrella guarantees no multiple
	   defintions of symbols in its
```

```
        sub-images so the two-level namespace
        hints can always be used. */
    MH_NOFIXPREBINDING /* do not have dyld notify the
        prebinding agent about this
        executable */
    MH_PREBINDABLE /* the binary is not prebound but can
        have its prebinding redone.
        only used when MH_PREBOUND is not set. */
    MH_ALLMODSBOUND /* indicates that this binary binds to
        all two-level namespace modules of
        its dependent libraries. only used
        when MH_PREBINDABLE and MH_TWOLEVEL
        are both set. */
    MH_SUBSECTIONS_VIA_SYMBOLS /* safe to divide up the sections into
        sub-sections via symbols for dead
        code stripping */
    MH_CANONICAL /* the binary has been canonicalized
        via the unprebind operation */
    MH_WEAK_DEFINES /* the final linked image contains
        external weak symbols */
    MH_BINDS_TO_WEAK /* the final linked image uses
        weak symbols */
    MH_ALLOW_STACK_EXECUTION /* When this bit is set, all stacks
        in the task will be given stack
        execution privilege.  Only used in
        MH_EXECUTE filetypes. */
    MH_ROOT_SAFE /* When this bit is set, the binary
        declares it is safe for use in
        processes with uid zero */
    MH_SETUID_SAFE /* When this bit is set, the binary
        declares it is safe for use in
        processes when issetugid() is true */
    MH_NO_REEXPORTED_DYLIBS /* When this bit is set on a dylib,
        the static linker does not need to
        examine dependent dylibs to see
        if any are re-exported */
    MH_PIE /* When this bit is set, the OS will
        load the main executable at a
        random address.  Only used in
        MH_EXECUTE filetypes. */
    MH_DEAD_STRIPPABLE_DYLIB /* Only for use on dylibs.  When
        linking against a dylib that
        has this bit set, the static linker
```

```
            will automatically not create a
            LC_LOAD_DYLIB load command to the
            dylib if no symbols are being
            referenced from the dylib. */
        MH_HAS_TLV_DESCRIPTORS /* Contains a section of type
            S_THREAD_LOCAL_VARIABLES */
        MH_NO_HEAP_EXECUTION /* When this bit is set, the OS will
            run the main executable with
            a non-executable heap even on
            platforms (e.g. i386) that don't
            require it. Only used in MH_EXECUTE
            filetypes. */
)

func printHeader(fh *macho.FileHeader) {
        log.Printf("fh.Magic: %#x", fh.Magic)
        log.Printf("fh.CPU: %s", fh.Cpu)
        log.Printf("fh.SubCPU: %#x", fh.SubCpu)

        log.Printf("fh.Type: %#x", fh.Type)
        switch fh.Type {
        case macho.TypeExec:
                log.Println("file is an executable")
        case macho.TypeObj:
                log.Println("file is an object")
        default:
                panic("not reachable")
        }

        log.Printf("fh.Ncmd: %d", fh.Ncmd)
        log.Printf("fh.Cmdsz: %d", fh.Cmdsz)
        log.Printf("fh.Flags: %#b", fh.Flags)

        switch fh.Flags & MH_NOUNDEFS {
        case 0:
                log.Println("MH_NOUNDEFS flag is not set")
        default:
                log.Println("object has no undefined references")
        }

        switch fh.Flags & MH_INCRLINK {
        case 0:
                log.Println("MH_INCRLINK flag is not set")
```

```
	default:
		log.Println("the object file is the output of an incremental link against a base f\
ile and can't be link edited again")
	}

	switch fh.Flags & MH_DYLDLINK {
	case 0:
		log.Println("MH_DYLDLINK flag is not set")
	default:
		log.Println("the object file is input for the dynamic linker and can't be staticly\
 link edited again")
	}

	switch fh.Flags & MH_SETUID_SAFE {
	case 0:
		log.Println("MH_SETUID_SAFE flag is not set")
	default:
		log.Println("executable is setuid safe")
	}
}

func printSection(s *macho.Section) {
	log.Printf("section %s", s.Name)
	log.Printf("\tSeg %s", s.Seg)
	log.Printf("\tAddr %#x", s.Addr)
	log.Printf("\tSize %d", s.Size)
	log.Printf("\tOffset %d", s.Offset)
	log.Printf("\tAlign %d", s.Align)
	log.Printf("\tReloff %s", s.Seg)
	log.Printf("\tNreloc %d", s.Nreloc)
	log.Printf("\tFlags %b", s.Flags)
}

func printSections(sections []*macho.Section) {
	for _, section := range sections {
		printSection(section)
	}
}

func printSymtab(symtab *macho.Symtab) {
	if symtab == nil {
		log.Println("no symbol table")
	}
```

```
        log.Printf("symtab.Cmd: %s", symtab.Cmd)
        log.Printf("symtab.Len: %d", symtab.Len)
        log.Printf("symtab.Symoff: %d", symtab.Symoff)
        log.Printf("symtab.Nsyms: %d", symtab.Nsyms)
        log.Printf("symtab.Stroff: %d", symtab.Stroff)
        log.Printf("symtab.Strsize: %d", symtab.Strsize)
        log.Printf("symtab has %d symbols", len(symtab.Syms))

        // Grab about 2.5% of the symbols
        symbols := make([]string, 0, len(symtab.Syms)/40)
        for _, symbol := range symtab.Syms {
                if rand.Float32() <= 0.025 {
                        symbols = append(symbols, symbol.Name)
                }
        }
        log.Printf("a selection of the symbols: %v", symbols)
}

func printDysymtab(dysymtab *macho.Dysymtab) {
        log.Printf("dysymtab.Cmd: %s", dysymtab.Cmd)
        log.Printf("dysymtab.Len: %d", dysymtab.Len)
        log.Printf("len(dysymtab.IndirectSyms): %d", len(dysymtab.IndirectSyms))
}

func printImportedLibraries(importedLibraries []string, err error) {
        if err != nil {
                log.Printf("failed getting imported libraries: %s", err)
                return
        }
        log.Printf("file imports %d libraries: %s", len(importedLibraries), importedLibrari\
es)
}

func printFileInformation(f *macho.File) {
        log.Printf("ByteOrder: %s", f.ByteOrder)
        printHeader(&f.FileHeader)

        // Also f.FileHeader.Ncmd
        log.Printf("file has %d load commands", len(f.Loads))
        log.Printf("file has %d sections", len(f.Sections))

        printSections(f.Sections)
```

```
        printSymtab(f.Symtab)
        printDysymtab(f.Dysymtab)
        printImportedLibraries(f.ImportedLibraries())
}

func main() {
        file, err := macho.Open("bash.macho")
        if err != nil {
                log.Fatalf("failed opening file: %s", err)
        }
        defer file.Close()
        printFileInformation(file)
}
```

pe

A Windows Portable Executable[41] file the format used on Windows. It fills the same gap as ELF and Mach-O, except it's for Windows.

I've made a simple Hello World application using C# and Mono[42] to use with the example. Who knows what the licensing problems would be distributing `cmd.exe`.

debug/pe.go

```
package main

import (
        "debug/pe"
        "log"
)

func printFileHeader(fh pe.FileHeader) {
        log.Printf("fh.Machine: %d", fh.Machine)
        log.Printf("fh.NumberOfSections: %d", fh.NumberOfSections)
        log.Printf("fh.TimeDateStamp: %d", fh.TimeDateStamp)
        log.Printf("fh.PointerToSymbolTable: %#x", fh.PointerToSymbolTable)
        log.Printf("fh.NumberOfSymbols: %d", fh.NumberOfSymbols)
        log.Printf("fh.SizeOfOptionalHeader: %d", fh.SizeOfOptionalHeader)
        log.Printf("fh.Characteristics: %#x", fh.Characteristics)
```

[41] http://en.wikipedia.org/wiki/Portable_Executable
[42] http://www.mono-project.com/

```go
}

func printSection(s *pe.Section) {
	log.Printf("section %s", s.Name)
	log.Printf("\tVirtualSize: %d", s.VirtualSize)
	log.Printf("\tVirtualAddress: %d", s.VirtualAddress)
	log.Printf("\tSize: %d", s.Size)
	log.Printf("\tOffset: %d", s.Offset)
	log.Printf("\tPointerToRelocations: %d", s.PointerToRelocations)
	log.Printf("\tPointerToLineNumbers: %d", s.PointerToLineNumbers)
	log.Printf("\tNumberOfRelocations: %d", s.NumberOfRelocations)
	log.Printf("\tNumberOfLineNumbers: %d", s.NumberOfLineNumbers)
	log.Printf("\tCharacteristics: %d", s.Characteristics)
}

func printSections(sections []*pe.Section) {
	for _, section := range sections {
		printSection(section)
	}
}

func printImportedLibraries(importedLibraries []string, err error) {
	if err != nil {
		log.Printf("failed getting imported libraries: %s", err)
		return
	}
	log.Printf("file imports %d libraries: %s", len(importedLibraries), importedLibrari\
es)
}

func printImportedSymbols(importedSymbols []string, err error) {
	if err != nil {
		log.Printf("failed getting imported symbols: %s", err)
		return
	}
	log.Printf("file imports %d symbols: %s", len(importedSymbols), importedSymbols)
}

func printFileInformation(f *pe.File) {
	printFileHeader(f.FileHeader)
	printSections(f.Sections)
	printImportedLibraries(f.ImportedLibraries())
	printImportedSymbols(f.ImportedSymbols())
```

```go
}

func main() {
	file, err := pe.Open("Hello.exe")
	if err != nil {
		log.Fatalf("failed opening file: %s", err)
	}
	defer file.Close()
	printFileInformation(file)
}
```

gosym

This package just wouldn't be complete without the ability to look at Go specifc information embedded by the gc family of compilers. The `debug/gosym` package lets you do that.

You start off by using one of the previous 3 packages, use the `debug/gosym` package to make a `LineTable` out of the `TEXT` segment. Then you can make a `Table` and start poking around.

I could only get this working on ELF files. I'm working on a MacBook and could not for the life of me get a Mach-O file to have the required sections. Not sure if this is a limitation of current implementation or if I'm just doing something wrong. That being said, when compiling an ELF file, you don't need to do anything special for the correct sections to be present. Since I didn't have Go setup on a Linux machine, I downloaded the `doozerd` binary from https://github.com/ha/doozerd[43]. It is licensed under the MIT license[44].

[43] https://github.com/ha/doozerd
[44] https://github.com/ha/doozerd/blob/master/LICENSE

debug/gosym.go

```go
package main

import (
	"debug/elf"
	"debug/gosym"
	"log"
	"math/rand"
	"time"
)

func init() {
	rand.Seed(time.Now().UnixNano())
}

func printSyms(syms []gosym.Sym) {
	selection := make([]string, 0, 24)
	for _, sym := range syms {
		if sym.Name != "" {
			if rand.Float32() <= 0.005 {
				selection = append(selection, sym.Name)
			}
		}
	}
	log.Printf("there are %d symbols, printing %d of them", len(syms), len(selection))
	log.Printf("a selection of symbols: %v", selection)
}

func printFuncs(funcs []gosym.Func) {
	selection := make([]string, 0, 24)
	for _, f := range funcs {
		if rand.Float32() <= 0.005 {
			selection = append(selection, f.Name)
		}
	}
	log.Printf("there are %d functions, printing %d of them", len(funcs), len(selection\
))
	log.Printf("a selection of functions: %v", selection)
}

func printFiles(files map[string]*gosym.Obj) {
	selection := make([]string, 0, 24)
```

```go
	for name := range files {
		if rand.Float32() <= 0.02 {
			selection = append(selection, name)
		}
	}
	log.Printf("there are %d files, printing %d of them", len(files), len(selection))
	log.Printf("a selection of files: %v", selection)
}

func getSectionData(f *elf.File, name string) []byte {
	section := f.Section(name)
	if section == nil {
		log.Fatalf("failed getting section %s", name)
	}
	data, err := section.Data()
	if err != nil {
		log.Fatalf("failed getting section %s data: %s", name, err)
	}
	return data
}

func processGoInformation(f *elf.File) {
	gosymtab := getSectionData(f, ".gosymtab")
	gopclntab := getSectionData(f, ".gopclntab")

	lineTable := gosym.NewLineTable(gopclntab, f.Section(".text").Addr)
	table, err := gosym.NewTable(gosymtab, lineTable)
	if err != nil {
		log.Fatalf("failed making table: %s", err)
	}

	printSyms(table.Syms)
	printFuncs(table.Funcs)
	printFiles(table.Files)
}

func main() {
	file, err := elf.Open("doozerd")
	if err != nil {
		log.Fatalf("failed opening file: %s", err)
	}
	defer file.Close()
	processGoInformation(file)
```

```
}
```

dwarf

DWARF[45] is a standardized file format for debugging information. You'll find it in Mach-O, ELF, and even Window Portable Executable files.

I've made a simple little program that prints `Hello, World` and also prints `ARGV` before and after sorting.

I've compiled it on a Ubuntu 10.04 64-bit box with `gcc -Wall -pedantic -O0 -g -ggdb -arch x86_64 -m64 -march=core2 -arch x86_64 -m64 -march=core2 hello.c -o hello`. We'll use this in the last example to look at the DWARF data inside the file.

As with the other examples, I'm only scratching the surface. If you're needing to play with DWARF info, you probably know more than I, and already have an idea as to what you're looking for.

debug/dwarf.go

```
package main

import (
	"debug/elf"
	"log"
)

func printDwarfInformation(f *elf.File) {
	dwarf, err := f.DWARF()
	if err != nil {
		log.Printf("failed getting DWARF info: %s", err)
		return
	}

	rd := dwarf.Reader()
	for {
		entry, err := rd.Next()
		if err != nil {
			log.Printf("failed getting next DWARF entry: %s", err)
			return
		}
```

[45] http://en.wikipedia.org/wiki/DWARF

```
		if entry == nil {
			// All done
			return
		}
		log.Printf("got entry with tag: %s, and offset %d", entry.Tag, entry.Offset)
		for _, field := range entry.Field {
			log.Printf("\t%s: %v", field.Attr, field.Val)
		}
	}
}

func main() {
	file, err := elf.Open("hello")
	if err != nil {
		log.Fatalf("failed opening file: %s", err)
	}
	defer file.Close()
	printDwarfInformation(file)
}
```

encoding

The `encoding` package, much like the `debug` package, is a high level package containing other packages where all the fun happens.

Can you guess what the `encoding` package does? I'll wait.

Encode things of course! Well, it'll decode too. This is where you get XML and JSON encoding, CSV encoding, base64, base32, and hex encoding.

Those all make perfect sense, and you'll probably use those regularly.

You also get ascii85[46] to play with Adobe file formats, asn1[47] and pem[48] to deal with their respective formats, and a `binary` package to deal with, well, binary data. You also get the `gob` package, which is a Go specific format.

We'll go through them in order.

ascii85

The `ascii85` example is quite terse, simply because there's not a whole lot to cover. There are two other package methods to `Encode` and `Decode` byte slices, but I've only covered the `Encoder` and `Decoder` which work on streams by way of `io.Writer` and `io.Reader`. If you have to choose between the two methods, you should probably opt for the stream based solution.

Don't forget to close the `ascii85.Encoder` when you are done writing to it!

[46] http://en.wikipedia.org/wiki/Ascii85
[47] http://en.wikipedia.org/wiki/Asn1
[48] http://en.wikipedia.org/wiki/Privacy_Enhanced_Mail

encoding/ascii85.go

```
package main

import (
	"bytes"
	"encoding/ascii85"
	"io"
	"io/ioutil"
	"log"
	"os"
)

func data() []byte {
	data, err := ioutil.ReadFile("ascii85.go")
	if err != nil {
		log.Fatalf("failed reading file: %s", err)
	}
	return data
}

func main() {
	var buffer bytes.Buffer
	enc := ascii85.NewEncoder(io.MultiWriter(os.Stdout, &buffer))
	log.Println("encoding to stdout")
	_, err := enc.Write(data())
	enc.Close()
	if err != nil {
		log.Fatalf("failed encoding: %s", err)
	}
	println()
	dec := ascii85.NewDecoder(&buffer)
	log.Println("decoding to stdout")
	io.Copy(os.Stdout, dec)
}
```

asn1

ASN.1[49] is a standard format for encoding and transmitting data. What kind of data? Well it doesn't really matter, but it should have some sort of defined structure. It's

[49] http://en.wikipedia.org/wiki/Asn1

more of a notation for describing the structure of the data. Even if you don't use this directly, you do indirectly: RSA keys are stored using ASN.1 (and then PEM encoded). We already saw ASN.1 used in the DSA example, and it's also used under the hood in the RSA example (by way of the `x509` package).

The format (notation) for the RSA private key can be seen in RFC 3447[50]

```
RSAPrivateKey ::= SEQUENCE {
    version           Version,
    modulus           INTEGER,  -- n
    publicExponent    INTEGER,  -- e
    privateExponent   INTEGER,  -- d
    prime1            INTEGER,  -- p
    prime2            INTEGER,  -- q
    exponent1         INTEGER,  -- d mod (p-1)
    exponent2         INTEGER,  -- d mod (q-1)
    coefficient       INTEGER,  -- (inverse of q) mod p
    otherPrimeInfos   OtherPrimeInfos OPTIONAL
}
```

RSA Private Key ASN.1 Notation

If you need to use this package, you'll probably have to refer back to the docs a bit more carefully, and possibly consult ASN.1 references somewhere online. It can get interesting. That being said, it's still a fairly straightforward encoding, so you can examine the byte slice and see how things are actually encoded.

For example, a `1` gets encoded as `[]byte{0x2, 0x1, 0x1}`. The `0x2` is a tag to say that it's an `INTEGER`, then `0x1` is the length (number of bytes), and finally the value.

`fizzbuzz` encodes as `[]byte{0x13, 0x8, 0x66, 0x69, 0x7a, 0x7a, 0x62, 0x75, 0x7a, 0x7a}`. It follows the same structure: tag, length, data. `0x13` (or 19 in decimal) is for a `PrintableString`, it's `0x8` bytes long, and then the actual data follows.

In the `IntRange` example, you can probably follow along[51]. I can't for the life of me figure out where the first `0x30` comes from, but everything after that makes sense.

[50] http://tools.ietf.org/html/rfc3447
[51] http://luca.ntop.org/Teaching/Appunti/asn1.html

encoding/asn1.go

```go
package main

import (
	"encoding/asn1"
	"log"
)

type IntRange struct {
	High, Low int
}

func encode(i interface{}) {
	data, err := asn1.Marshal(i)
	if err != nil {
		log.Printf("failed asn1 marshalling %#v: %s", i, err)
	} else {
		log.Printf("%#v marshals to %#v", i, data)
	}
}

func main() {
	encode(1)
	encode(1.5)
	encode('a')
	encode("fizzbuzz")
	encode(IntRange{10, 5})
}
```

base32

The `base32` package handles, of course, base32 based encoding. It actually does a couple different encodings that are standards. From the docs:

> StdEncoding is the standard base32 encoding, as defined in RFC 4648. HexEncoding is the “Extended Hex Alphabet” defined in RFC 4648. It is typically used in DNS.

When you make your `Decoder` or `Encoder`, you must pick one of these encodings to use, and naturally you have to use the same encoding when performing the opposite operation.

Run the example with and without the `-hex` flag to see the difference in the encodings.

encoding/base32.go

```go
package main

import (
	"bytes"
	"encoding/base32"
	"flag"
	"io"
	"io/ioutil"
	"log"
	"os"
)

var hex = flag.Bool("hex", false, "Use HexEncoding instead of StdEncoding")

func data() []byte {
	data, err := ioutil.ReadFile("base32.go")
	if err != nil {
		log.Fatalf("failed reading file: %s", err)
	}
	return data
}

func encoding() *base32.Encoding {
	if *hex {
		return base32.HexEncoding
	}
	return base32.StdEncoding
}

func main() {
	flag.Parse()
	var buffer bytes.Buffer
	enc := base32.NewEncoder(encoding(), io.MultiWriter(os.Stdout, &buffer))
	log.Println("encoding to stdout")
	_, err := enc.Write(data())
```

```
        enc.Close()
        if err != nil {
                log.Fatalf("failed encoding: %s", err)
        }
        println()
        dec := base32.NewDecoder(encoding(), &buffer)
        log.Println("decoding to stdout")
        io.Copy(os.Stdout, dec)
}
```

base64

Everybody knows base64! You've probably used it somewhere in your life.

The `base64` package works exactly like the `base32` package. You make an `Encoder` or `Decoder` from one of the two available encodings it has, and go to town.

Your two options for encodings are the `StdEncoding` which you're probably most familiar with. It came from RFC4648[52] and is seen in MIME and PEM. The other is `URLEncoding` which just replaces `+` and `/` with `-` and `_` so it can be used safely in URLs.

encoding/base64.go

```
package main

import (
        "bytes"
        "encoding/base64"
        "flag"
        "io"
        "io/ioutil"
        "log"
        "os"
)

var url = flag.Bool("url", false, "Use URLEncoding instead of StdEncoding")

func data() []byte {
        data, err := ioutil.ReadFile("base64.go")
```

[52] http://datatracker.ietf.org/doc/rfc4648/

```
	if err != nil {
		log.Fatalf("failed reading file: %s", err)
	}
	return data
}

func encoding() *base64.Encoding {
	if *url {
		return base64.URLEncoding
	}
	return base64.StdEncoding
}

func main() {
	flag.Parse()
	var buffer bytes.Buffer
	enc := base64.NewEncoder(encoding(), io.MultiWriter(os.Stdout, &buffer))
	log.Println("encoding to stdout")
	_, err := enc.Write(data())
	enc.Close()
	if err != nil {
		log.Fatalf("failed encoding: %s", err)
	}
	println()
	dec := base64.NewDecoder(encoding(), &buffer)
	log.Println("decoding to stdout")
	io.Copy(os.Stdout, dec)
}
```

binary

The `binary` package lets you deal with, holy popsicle sticks, deal with binary data. The raw functions only let you deal with basic bytes and int type stuff, which is pretty low level. The `Read` and `Write` functions give you a bit higher level wrapper around those, and let you deal with structs.

First, I show a basic encoding and decoding of `math.Pi`, then a broken version (encoding with one endianness[53] and decoding with the other), and then we look at the header for GIF files.

[53] http://en.wikipedia.org/wiki/Endianness

encoding/binary.go

```go
package main

import (
	"bytes"
	"encoding/binary"
	"log"
	"math"
)

func simple() {
	var buffer bytes.Buffer
	binary.Write(&buffer, binary.LittleEndian, math.Pi)
	log.Printf("encoded %#v, a %T, to %#v", math.Pi, math.Pi, buffer.Bytes())

	var pi float64
	binary.Read(&buffer, binary.LittleEndian, &pi)
	log.Printf("decoded %#v (is it equal?: %v)", pi, pi == math.Pi)
}

func broken() {
	var buffer bytes.Buffer
	binary.Write(&buffer, binary.BigEndian, math.Pi)
	log.Printf("encoded %#v, a %T, to %#v", math.Pi, math.Pi, buffer.Bytes())

	var pi float64
	binary.Read(&buffer, binary.LittleEndian, &pi)
	log.Printf("decoded %#v (is it equal?: %v)", pi, pi == math.Pi)
}

func main() {
	simple()
	broken()
}
```

GIF

A GIF file is stored using the Little Endian byte ordering, and the GIF header[54] looks like this:

[54] http://www.onicos.com/staff/iz/formats/gif.html

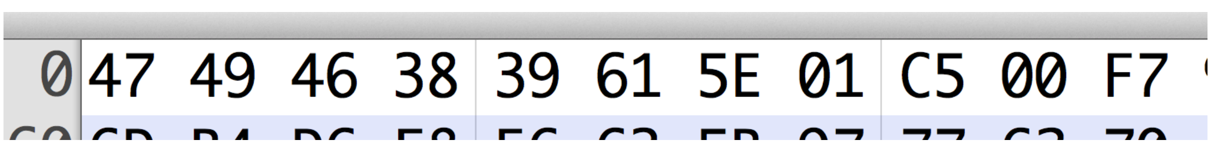
```
GIF Header

Offset   Length   Contents
  0      3 bytes  "GIF"
  3      3 bytes  "87a" or "89a"
  6      2 bytes  <Logical Screen Width>
  8      2 bytes  <Logical Screen Height>
 10      1 byte   bit 0:    Global Color Table Flag (GCTF)
                  bit 1..3: Color Resolution
                  bit 4:    Sort Flag to Global Color Table
                  bit 5..7: Size of Global Color Table: 2^(1+n)
 11      1 byte   <Background Color Index>
 12      1 byte   <Pixel Aspect Ratio>
 13      ? bytes  <Global Color Table(0..255 x 3 bytes) if GCTF is one>
         ? bytes  <Blocks>
         1 bytes  <Trailer> (0x3b)
```

GIF header structure

For this case, the raw functions are kind of gross honestly. I tried to make an example with them, but they are kind of unwieldy. Maybe if things were nice 32/64-bit values, but they aren't.

We can, however, use a struct and get the `binary` package to handle all the hard work. We'll just do the version and dimensions to keep the code short.

Before we begine, let's look at the GIF file in a hex editor to try and make some sense of it:



GIF header in HexFiend

Following the spec, the first three bytes `0x47 0x49 0x46` are the ASCII characters `GIF`. The next 3 bytes `0x38 0x39 0x61` are the ASCII characters `89a`. In order to decode into a struct, we must use fixed-sized values inside that struct. We create a `Version` type that is a 6 element byte array. We can tack a method onto it to make it a string, and bam, there's our version. The `binary` package will now decode the first 6 bytes into that array, and we can print it out as `GIF89a`.

For the dimensions, I used a single `uint32` since its size is 4 bytes, and the dimensions are 4 bytes (2 for width, 2 for height).

The `binary` package has no problem pulling the next 4 bytes out into the `Dimensions` value.

> The `0x5e 0x01` and `0xc5 0x00` in the hex editor would probably normally be written as `0x01 0x5e` and `0x00 0xc5` but remember the GIF is little endian. This means when it gets read out, things get flipped around. The width ends up in the lower half of the `uint32` value, even though it's *first* in the file as far as the raw bytes are concerned. This is because we read it out as part of a 32-bit value we are calling the dimensions. This is why we have to do the shift in the `Height` method instead of the `Width` method.

encoding/gif.go

```go
package main

import (
	"encoding/binary"
	"log"
	"os"
)

type Version [6]byte

func (v Version) String() string {
	return string(v[:])
}

type Dimensions uint32

func (d Dimensions) Width() int {
	return int(d) & 0xffff
}

func (d Dimensions) Height() int {
	return int(d>>16) & 0xffff
}

type GifHeader struct {
	Version    Version
	Dimensions Dimensions
}

func main() {
	file, err := os.Open("animated.gif")
	if err != nil {
```

```
                log.Fatalf("failed opening gif: %s", err)
        }
        defer file.Close()
        var header GifHeader
        binary.Read(file, binary.LittleEndian, &header)
        log.Printf("decoded a %s with width %dpx and height %dpx", header.Version, header.D\
imensions.Width(), header.Dimensions.Height())
}
```

CSV

Just like `base64`, you've probably seen CSV encoding before. Comma Separated Values is a nice way to encode tabular data, like say from a relational database.

The whole "comma separated" part is a bit of simplification, since you can separate the values with whatever works for your data.

The `csv.Reader` type has a few more configuration options than the `csv.Writer` type, which allows you to read a wider variety of files than you can write. Once you have a reader, before you start reading, you can configure a few things. The important ones are:

- Separator rune, defaults to comma.
- The comment rune. Lines starting with this rune will be ignored.
- Fields per record. Configures any checking/verification done on the number of fields in each record. If you don't change it, it ensures all the records have the same number of fields as the first row.

The writer only allows you to configure the separator (defaults to a comma) and whether to use `\r\n` instead of a plain `\n`.

The `csv` package doesn't have any helpers around structs, so you have to do it yourself (or write a `reflect`-based package and share it!)

encoding/csv.go

```go
package main

import (
	"bytes"
	"encoding/csv"
	"io"
	"log"
)

var records = [][]string{
	{"Show", "Seasons", "Year Began", "Year End"},
	{"The Simpsons", "24", "1989", ""},
	{"Star Trek: The Next Generation", "7", "1987", "1994"},
	{"Seinfeld", "9", "1989", "1998"},
	{"Go, Diego, Go!", "5", "2005", "2011"},
}

func write(w io.Writer, sep rune, recs [][]string) error {
	csvWriter := csv.NewWriter(w)
	csvWriter.Comma = sep
	return csvWriter.WriteAll(recs)
}

func read(r io.Reader, sep rune) ([][]string, error) {
	csvReader := csv.NewReader(r)
	csvReader.Comma = sep
	return csvReader.ReadAll()
}

func main() {
	var buffer bytes.Buffer
	err := write(&buffer, ',', records)
	if err != nil {
		log.Fatalf("failed writing: %s", err)
	}
	log.Printf("wrote: %s", &buffer)
	rs, err := read(&buffer, ',')
	if err != nil {
		log.Fatalf("failed reading: %s", err)
	}
	log.Printf("%v", rs)
```

```go

	buffer = bytes.Buffer{}
	err = write(&buffer, '|', records)
	if err != nil {
		log.Fatalf("failed writing: %s", err)
	}
	log.Printf("wrote: %s", &buffer)
	rs, err = read(&buffer, ',') // Will fail
	if err != nil {
		log.Fatalf("failed reading: %s", err)
	}
	panic("not reached")
}
```

gob

The `gob` package handles, you guessed it, gobs. Gobs are binary blobs that encode Go types, complete with a description of the type. This means you can send something across the wire to an application and when it decodes it it will just be the correct type. If you want to send a type as an interface implementation, you have to register the type, so what you'll frequently see is type definitions, and an `init` function to register those types with the `gob` package.

encoding/gob.go

```go
package main

import (
	"encoding/gob"
	"log"
	"net"
	"os"
)

var sock = "gob.sock"

type IntRange struct {
	High, Low int
}

func init() {
```

```
	gob.Register(IntRange{})
}

func handle(c net.Conn) {
	defer c.Close()
	decoder := gob.NewDecoder(c)
	var i interface{}
	for {
		err := decoder.Decode(&i)
		if err != nil {
			log.Printf("failed decoding value: %s", err)
			break
		}
		log.Printf("decoded: %#v", i)
	}
}

func server(sig chan bool) {
	addr, err := net.ResolveUnixAddr("unix", sock)
	if err != nil {
		log.Fatalf("failed to resolve addr: %s", err)
	}
	defer os.RemoveAll(sock)

	listener, err := net.ListenUnix("unix", addr)
	if err != nil {
		log.Fatalf("failed to listen: %s", err)
	}
	defer listener.Close()

	sig <- true
	conn, err := listener.Accept()
	if err != nil {
		log.Printf("failed accept: %s", err)
	}
	handle(conn)
	sig <- true
}

func client() {
	addr, err := net.ResolveUnixAddr("unix", sock)
	if err != nil {
		log.Fatalf("failed to resolve addr: %s", err)
```

```
	}

	conn, err := net.DialUnix("unix", nil, addr)
	if err != nil {
		log.Fatalf("failed dialing: %s", err)
	}
	defer conn.Close()

	encoder := gob.NewEncoder(conn)
	things := []interface{}{IntRange{5, 10}, 1, 1.5, "hello", 2 + 3i}
	for _, thing := range things {
		err = encoder.Encode(&thing)
		if err != nil {
			log.Printf("failed encoding: %s", err)
		} else {
			log.Printf("encoded: %#v", thing)
		}
	}
}

func main() {
	sig := make(chan bool)
	go server(sig)
	<-sig
	client()
	<-sig
}
```

hex

The `hex` package deals with hexadecimal encoded data. It can encode and decode byte slices, encode and decode to a string, and with a `hex.Dumper` it can also *dump* something to the same format as `hexdump -C`.

encoding/hex.go

```go
package main

import (
	"encoding/hex"
	"io/ioutil"
	"log"
	"os"
)

func dumpFile() {
	data, err := ioutil.ReadFile("hex.go")
	if err != nil {
		log.Fatalf("failed reading file: %s", err)
	}
	dumper := hex.Dumper(os.Stdout)
	defer dumper.Close()
	log.Println("dumping hex.go to stdout")
	dumper.Write(data)
}

func main() {
	hero := []byte("Batman and Robin")
	log.Printf("hero: %s", hero)
	encoded := hex.EncodeToString(hero)
	log.Printf("encoded: %s", encoded)
	decoded, _ := hex.DecodeString(encoded)
	log.Printf("decoded: %s", decoded)

	dumpFile()
}
```

json

Want to play with JSON? Use the `json` package. You can encode and decode simple types and structs, encode and decode other types that obey the relevant interfaces, and do all of that with readers and writers. You can also pretty print with `MarshalIndent`.

With the JSON package we also see use of field tags to control how the marshalling of struct fields happens. You can set the name of the name if you don't want it to get marshalled as the uppercase field name. You can also tell it to not marshal the field at all (even though it's an exported field) by setting the field name to -. You can also omit empty fields.

encoding/json.go

```go
package main

import (
	"bytes"
	"encoding/json"
	"fmt"
	"io"
	"log"
	"os"
)

type BlogPost struct {
	// Marshal as "writer" instead of Author
	Author string `json:"writer,omitempty"`
	// Will get marshalled as "Title"
	Title string
	Body  string `json:"body"`
	// Don't marshal this field at all
	Published bool `json:"-"`
}

// This would marshal just fine,
// but let's write out own marshaller.
type Pair struct {
	X, Y int
}

func (p Pair) MarshalJSON() ([]byte, error) {
	return []byte(fmt.Sprintf(`"%d|%d"`, p.X, p.Y)), nil
}

func (p *Pair) UnmarshalJSON(data []byte) error {
	_, err := fmt.Sscanf(string(data), `"%d|%d"`, &p.X, &p.Y)
	return err
}

func encodeTo(w io.Writer, i interface{}) {
	encoder := json.NewEncoder(w)
```

```
	if err := encoder.Encode(i); err != nil {
		log.Fatalf("failed encoding to writer: %s", err)
	}
}

func encode(i interface{}) []byte {
	data, err := json.Marshal(i)
	if err != nil {
		log.Fatalf("failed encoding: %s", data)
	}
	return data
}

func decode(data string) interface{} {
	var i interface{}
	err := json.Unmarshal([]byte(data), &i)
	if err != nil {
		log.Fatalf("failed decoding: %s", err)
	}
	return i
}

func simple() {
	log.Printf("encoded %d to %s", 1, encode(1))
	log.Printf("encoded %f to %s", 1.5, encode(1.5))
	log.Printf("encoded %s to %s", "Hello, World!", encode("Hello, World!"))

	log.Printf("decoded %f from %s", decode("1"), "1")
	log.Printf("decoded %v from %s", decode(`["foo","bar","baz"]`), `["foo","bar","baz"\
]`)
}

func custom() {
	pair := Pair{5, 10}
	encoded := encode(pair)
	log.Printf("encoded %v to %s", pair, encoded)

	var pair2 Pair
	if err := json.Unmarshal(encoded, &pair2); err != nil {
		log.Fatalf("failed decoding Pair: %s", err)
	}
	log.Printf("decoded %#v from %s", pair2, `"1|2"`)
}
```

```go

func structExample() {
	post := BlogPost{
		// Since Author is empty, it won't be written out
		Title:     "Being Awesome At Go",
		Body:      "Read this book!",
		Published: true,
	}
	encodeTo(os.Stdout, post)

	post = BlogPost{
		Author:    "Daniel Huckstep",
		Title:     "Being Awesome At Go",
		Body:      "Read this book!",
		Published: true,
	}
	encodeTo(os.Stdout, post)
}

func streamDecode() {
	var buffer bytes.Buffer
	post := BlogPost{
		Author:    "Daniel Huckstep",
		Title:     "Being Awesome At Go",
		Body:      "Read this book!",
		Published: true,
	}
	encodeTo(&buffer, post)

	decoder := json.NewDecoder(&buffer)
	var newPost BlogPost
	if err := decoder.Decode(&newPost); err != nil {
		log.Printf("decoding failed: %s", err)
	}
	log.Printf("decoded %#v", newPost)
}

func pretty() {
	post := BlogPost{
		Author:    "Daniel Huckstep",
		Title:     "Being Awesome At Go",
		Body:      "Read this book!",
		Published: true,
```

```go
	}
	data, err := json.MarshalIndent(post, "", "\t")
	if err != nil {
		log.Fatalf("failed marshal with indent: %s", err)
	}
	log.Printf("pretty print:\n%s", data)
}

func main() {
	simple()
	custom()
	structExample()
	streamDecode()
	pretty()
}
```

pem

PEM encoding from Privacy Enhanced Mail[55] is handled by the `pem` package. Where do you use this you might ask? RSA keys and SSL certificates, that's where!. Check your `~/.ssh` directory, and that `id_rsa` file is in PEM format.

We already saw `pem` in action in the RSA example using `x509.MarshalPKCS1PrivateKey` to get the `Bytes` for the `pem.Block`. This is a really simple example.

encoding/pem.go

```go
package main

import (
	"crypto/rand"
	"encoding/pem"
	"log"
	"os"
)

func main() {
	bytes := make([]byte, 1024)
	n, err := rand.Read(bytes)
	if err != nil {
```

[55] http://en.wikipedia.org/wiki/Privacy_Enhanced_Mail

```
		log.Fatalf("failed reading random data: %s", err)
	}
	if n != len(bytes) {
		log.Fatalf("failed reading correct amount of random data. only read %d bytes", n)
	}
	block := pem.Block{
		Type:  "Example Data",
		Bytes: bytes,
	}
	pem.Encode(os.Stdout, &block)
}
```

xml

The `xml` package handles going to and from XML. It's similar to the `json` package in that you can encode/decode to/from bytes, you can pretty print things, and you can do things with `io.Reader` and `io.Writer`. You can also control the output/parsing with tags.

Some extra things you can do when you're dealing with structs include serializing fields as attributes, include comments.

If you feel like it, you can even decode raw tokens.

In the example, pay attention to the tags in all the structs:

- And `XMLName` field with a tag to control the element name the struct gets encoded as.
- `xml:"id,attr"` on the Id field to make it an attribute instead of a nested element, and to change the attribute name to be lowercase instead of `Id`
- `xml:",omitempty"` on `Subtitle` to not include it if it's empty.
- `xml:"Tags>Tag"` on `Tags` to nest each tag as a `Tag` element inside a main `Tags` element.

encoding/xml.go

```go
package main

import (
	"bytes"
	"encoding/xml"
	"io"
	"log"
)

type Name struct {
	First, Last string `xml:",omitempty"`
}

type Author struct {
	Id   int `xml:"id,attr"`
	Name Name
}

type BlogPost struct {
	XMLName  xml.Name `xml:"Post"`
	Id       int      `xml:"id,attr"`
	Author   Author
	Title    string
	Subtitle string   `xml:",omitempty"`
	Tags     []string `xml:"Tags>Tag"`
	Body     string   `xml:"Content"`
	Notes    string   `xml:",comment"`
}

func encode(w io.Writer) {
	post := BlogPost{
		Id: 10,
		Author: Author{
			Id: 5,
			Name: Name{
				First: "Alan",
				Last:  "Kay",
			},
		},
		Title: "It's All About Messages",
		Tags:  []string{"object-oriented", "programming", "oop"},
```

```
		Body:  "It's not about objects, it's about messages",
		Notes: "He's the boss",
	}

	encoder := xml.NewEncoder(w)
	err := encoder.Encode(post)
	if err != nil {
		log.Fatalf("failed encoding to a stream: %s", err)
	}
}

func decode(r io.Reader) {
	var post BlogPost
	decoder := xml.NewDecoder(r)
	err := decoder.Decode(&post)
	if err != nil {
		log.Fatalf("failed decoding from stream: %s", err)
	}
	log.Printf("%#v", post)
}

func pretty() {
	post := BlogPost{
		Id: 5,
		Author: Author{
			Id: 2,
			Name: Name{
				First: "Daniel",
				Last:  "Huckstep",
			},
		},
		Title: "Go, The Standard Library",
		Tags:  []string{"golang", "programming", "reference"},
		Body:  "I <strong>like</strong> programming Go, it's so much fun!",
		Notes: "Need to write more often...",
	}
	data, err := xml.MarshalIndent(post, "", "\t")
	if err != nil {
		log.Fatalf("failed pretty printing: %s", err)
	}
	log.Printf("pretty print:%s", data)
}
```

```go
func tokens() {
	doc := []byte(`<post id="5"><title>Batman</title><author>Daniel Huckstep</author></\
post>`)
	decoder := xml.NewDecoder(bytes.NewReader(doc))
	for {
		token, err := decoder.Token()
		switch err {
		case nil:
			// Nothing to see here
		case io.EOF:
			log.Println("done parsing tokens")
			return
		default:
			log.Fatalf("got error getting token: %s", err)
		}

		switch tok := token.(type) {
		case xml.StartElement:
			log.Printf("found start element: %s", tok.Name)
		case xml.EndElement:
			log.Printf("found end element: %s", tok.Name)
		case xml.CharData:
			log.Printf("found chardata element: %s", tok)
		case xml.Comment:
			log.Printf("found comment element: %s", tok)
		case xml.ProcInst:
			log.Printf("found processing instruction: %s", tok.Target)
		case xml.Directive:
			log.Printf("found directive: %s", tok)
		default:
			panic("not reached")
		}
	}
}

func main() {
	pretty()
	var buffer bytes.Buffer
	encode(&buffer)
	log.Printf("encoded post to %s", buffer.String())
	decode(&buffer)
	tokens()
}
```

errors

The `errors` package let's you build an error. That's it. It has one function, and there is only one source file defining the entire package.

All you do is `errors.New("My error message")` and you've got yourself an error. More likely, you'll use the `fmt` package to build an error, but we'll look at it in a few chapters.

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

```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()
}
```

flag

The `flag` package is command line flag parsing in one tight package.

The basic usage consists of two APIs: the regular API, and the `*Var` API. The basic API returns a pointer to the thing it's handling, while the `*Var` API takes a pointer to an already existing thing that it should handle.

There is also a `FlagSet` so you can split up groups of flags, say if you're making something like the `go` program. Its first argument is the name of a tool, and each tool takes a different set of flags. You can organize these with a `FlagSet`.

You can also introspect the raw flags, see how many there are, and build your own custom types. It even builds in the `-h`/`-help`/`--help` flags and outputs appropriate help.

It supports both single and double dashes as the prefix, but if you want to support a short form (single letter) as well, you have to dance around a little, and it proves more work than it's worth.

The Basic Interface

flag/basic.go

```go
package main

import (
	"flag"
	"log"
)

var (
	count   = flag.Int("count", 1, "number of times to say hello")
	subject = flag.String("subject", "World", "subject to say hello to")
)

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

```go
}

func main() {
	flag.Parse()

	hello(*subject, *count)

	log.Printf("flag.NArg(): %d", flag.NArg())
	log.Printf("flag.Args(): %s", flag.Args())
}
```

The *Var Interface

flag/var.go

```go
package main

import (
	"flag"
	"log"
)

var (
	count   int
	subject string
)

func init() {
	flag.IntVar(&count, "count", 1, "number of times to say hello")
	flag.StringVar(&subject, "subject", "World", "subject to say hello to")

	flag.Parse()
}

func hello(s string, t int) {
	for i := 0; i < t; i++ {
		log.Printf("Hello, %s!", s)
	}
}

func main() {
```

```
        hello(subject, count)
}
```

FlagSet

flag/flagset.go

```go
package main

import (
        "flag"
        "log"
        "strings"
)

var (
        cmdFlags = map[string]*flag.FlagSet{
                "hello":   flag.NewFlagSet("hello", flag.ExitOnError),
                "goodbye": flag.NewFlagSet("goodbye", flag.ExitOnError),
        }
        subject = cmdFlags["hello"].String("subject", "World", "the subject to say hello to\
")
        dots    = cmdFlags["goodbye"].Int("dots", 3, "How many dots to print")
)

func hello(subject string) {
        log.Printf("Hello, %s!", subject)
}

func goodbye(dots int) {
        space := ", "
        if dots > 0 {
                space = strings.Repeat(".", dots)
        }
        log.Printf("Goodbye%scruel world!", space)
}

func main() {
        flag.Parse()
        for _, cmd := range flag.Args() {
                flags, ok := cmdFlags[cmd]
```

```
		if !ok {
			log.Fatalf("no command %q found", cmd)
		}
		flags.Parse(flag.Args()[1:])
		switch cmd {
		case "hello":
			hello(*subject)
		case "goodbye":
			goodbye(*dots)
		}
		break
	}
}
```

Custom

You can also implement an interface and parse custom types. Implement the two methods from `flag.Value`, and you're good to go.

flag/custom.go

```
package main

import (
	"flag"
	"fmt"
	"log"
)

type Point struct {
	X, Y int
}

func (p *Point) String() string {
	return fmt.Sprintf("%+d@%+d", p.X, p.Y)
}

func (p *Point) Set(s string) error {
	_, err := fmt.Sscanf(s, "%d@%d", &p.X, &p.Y)
	return err
}
```

```go

var point Point

func init() {
	flag.Var(&point, "point", "point as X@Y")
}

func main() {
	flag.Parse()
	log.Printf("%#v", point)
}
```

fmt

The `fmt` package takes care of formatting things. It will either return a string, or write to an `io.Writer` interface. There is also a convenience method to print to `stdout`. It can also *scan* things from a string or a `io.Reader` into various types.

I'm not going to cover the specific syntax for formatting certain values, since the regular docs cover that quite well.

Printing

Printing is straightforward. It's handled by all the functions with *print* in the name.

It's in the docs, but a quirk with the `Print` function is that it only puts a space between arguments when neither is a string. `Println` puts spaces between all arguments.

fmt/printing.go

```
package main

import (
	"fmt"
	"log"
	"os"
)

var (
	i   = 221
	b   = false
	f   = 5.1
	cn  = 3 + 1i
	s   = "batman"
	big = 13.8 * 100000
	c   = struct {
		Count int
		Debug bool
		Notes string
```

```
	}{8, true, "This is my boomstick!"}
)

func stdout() {
	fmt.Print("Print: ", c, i, b, f, cn, s, "\n")
	fmt.Println("Println:", c, i, b, f, cn, s)
	fmt.Printf("Printf: %#b %#x %t %v %T %e\n", i, i, true, c, c, big)

	// Padding strings
	fmt.Printf("%15s\n", "batman")
	fmt.Printf("%15s\n", "wat")
	fmt.Printf("%15s\n", "Bruce Wayne")
}

func writer() {
	file, err := os.OpenFile("output.txt", os.O_WRONLY|os.O_CREATE, 0644)
	if err != nil {
		panic(err)
	}
	defer file.Close()

	fmt.Fprint(file, "Fprint: ", c, i, b, f, cn, s)
	fmt.Fprintln(file, "Fprintln:", c, 1, false, f, cn, s)
	fmt.Fprintf(file, "Fprintf: %#b %#x %t %v %T %e\n", i, i, b, c, c, big)
}

func str() {
	out := fmt.Sprintln(c, i, b, f, cn, s)
	log.Printf("Sprintln: %s", out)

	out = fmt.Sprintf("%#b %#x %t %v %T %e", i, i, b, c, c, big)
	log.Printf("Sprintf: %s", out)
}

func main() {
	stdout()
	writer()
	str()
}
```

You'll notice I don't check the return value of any of these functions. While they do return the number of bytes written and a possible error, they are some of the functions that you probably don't need to bother checking the return value of. If you're writing to a file, the network, or something else important, you probably want to check, but if you're writing debug information to `stdout` you probably don't need to bother.

The example shows the use of the `#` flag, which prints things using an *alternate format*. In the example, this means printing binary with a leading `0b` and hexadecimal with a leading `0x`. The documentation covers the other situations.

Scanning

Scanning is also quite simple. It's handled by all the functions with *scan* in the name. Don't forget to pass things as pointers!

To simplify things, I won't bother with the functions that deal with `stdin`. Once you see the others working, it's pretty straight forward to use them. You could even used the `io.Reader` based ones and pass in `os.Stdin`.

fmt/scanning.go

```go
package main

import (
	"fmt"
	"log"
	"os"
)

func str() {
	var a int
	var b int

	log.Printf("a: %d, b: %d", a, b)
	fmt.Sscan("20\n20", &a, &b)
	log.Printf("a: %d, b: %d", a, b)

	fmt.Sscanf("(15, 30)", "(%d, %d)", &a, &b)
	log.Printf("a: %d, b: %d", a, b)

	// Will not go past the newline, only scans a
```

```go
	fmt.Sscanln("10\n10", &a, &b)
	log.Printf("a: %d, b: %d", a, b)
}

func reader() {
	file, err := os.Open("input.txt")
	if err != nil {
		panic(err)
	}
	defer file.Close()

	var scan struct {
		A, B float32
		C    bool
		D    string
	}

	log.Printf("scan: %v", scan)
	fmt.Fscan(file, &scan.A, &scan.B)
	log.Printf("scan: %v", scan)
	fmt.Fscan(file, &scan.C, &scan.D)
	log.Printf("scan: %v", scan)

	fmt.Fscanln(file, &scan.A, &scan.B, &scan.C, &scan.D)
	log.Printf("scan: %v", scan)

	fmt.Fscanf(file, "The Green %s %f %t %f", &scan.D, &scan.B, &scan.C, &scan.A)
	log.Printf("scan: %v", scan)
}

func main() {
	str()
	reader()
}
```

Printing Custom Types

Well that was fun! Actually not really. Formatting and scanning things? Yawn. It's all very straightforward and there's nothing missing from the standard documentation for everyday use of the `fmt` package.

But you don't have to live in the `fmt` walls, you can format your data anyway you want! There are 3 ways the `fmt` provides to let you customize formatting.

Stringer Interface

The `Stringer` interface you see a lot in Go. Define a method called `String` that takes no arguments and returns a `string`, and you're set. You can then pass your type to `fmt` and format it as a string with the `%s` verb and it will just work. Using the `%v` verb will also use the `Stringer` interface.

> If the thing implements the `Error` interface, it takes precedence over the `Stringer` interface.

While I won't repeat it here, make note of the recursion case in the documentation. You can shoot yourself in the foot, but you have tests right?

fmt/stringer.go

```go
package main

import (
	"fmt"
	"log"
)

type Tuple struct {
	Left, Right interface{}
}

func (t Tuple) String() string {
	log.Printf("in Stringer interface method for Tuple")
	return fmt.Sprintf("(%#v, %#v)", t.Left, t.Right)
}

type Tuple2 struct {
	Left, Right interface{}
}

func (t Tuple2) Error() string {
	log.Printf("in Error interface method for Tuple2")
	return "lol it's an error!"
}
```

```go

func (t Tuple2) String() string {
	log.Printf("in Stringer interface method for Tuple2")
	return fmt.Sprintf("(%#v, %#v)", t.Left, t.Right)
}

func main() {
	fmt.Printf("%s\n", Tuple{1, 2})
	fmt.Printf("%s\n", Tuple2{1.5, 2.1})
	fmt.Printf("%v\n", Tuple{"Bruce Wayne", "Batman"})
}
```

GoStringer Interface

The `GoStringer` interface operates like the `Stringer` interface in that you return a `string`, but is used with the `%#v` verb.

There's no example for this, since you can take the previous example, change `String()` to `GoString()` and `%s` to `%#v`, and you're basically done.

I'm also a little unsure why you'd want to override the default implementation of this, but you can. If you find a good example for this, please let me know!

Formatter Interface

For doing seriously custom formats, you can define `Format(f State, c rune)` on your type to implement the `Formatter` interface. You can inspect the `State` passed in to check for flags and other things. You can also see what the verb used is with the `c rune` argument. In the example, I use the `l`, `r`, and `P` verbs to format my `Tuple` type.

fmt/formatter.go

```go
package main

import (
	"fmt"
)

type Tuple struct {
	Left, Right int
}

```

```go
func (t Tuple) Format(f fmt.State, c rune) {
	switch c {
	case 'l':
		fmt.Fprintf(f, "%v", t.Left)
	case 'r':
		fmt.Fprintf(f, "%v", t.Right)
	case 'P', 's', 'v':
		fmt.Fprintf(f, "(%#v, %#v)", t.Left, t.Right)
	}
}

func main() {
	t := Tuple{1, 2}
	fmt.Printf("%l\n", t)
	fmt.Printf("%r\n", t)
	fmt.Printf("%P\n", t)
}
```

Scanning Custom Types

The `Scanner` interface lets you implement a custom scanner for your type. You get a `ScanState` which is similar to `State` from the formatting example, and the verb used as a `rune`. `ScanState` has the `Token` method, which is probably the most immediately useful method, except for the fact that that `ScanState` is an `io.Reader`. This means we can use other `fmt` functions like `fmt.Fscanf` to scan out a few things given a more specific format. This is how I've done things in the example.

fmt/scanner.go

```go
package main

import (
	"fmt"
)

type Tuple struct {
	Left, Right int
}

func (t Tuple) Format(f fmt.State, c rune) {
	switch c {
```

```
        case 'P':
                fmt.Fprintf(f, "(%#v, %#v)", t.Left, t.Right)
        }
}

func (t *Tuple) Scan(state fmt.ScanState, verb rune) error {
        switch verb {
        case 'P':
                n, err := fmt.Fscanf(state, "(%d, %d)", &t.Left, &t.Right)
                if err != nil {
                        return err
                }
                if n != 2 {
                        return fmt.Errorf("scanned %d things, expected 2", n)
                }
        }
        return nil
}

func main() {
        var i int
        var f float32
        var t Tuple

        fmt.Printf("%d %P %f\n", i, t, f)
        fmt.Sscanf("5 (1, 2) 2.5", "%d %P %f", &i, &t, &f)
        fmt.Printf("%d %P %f\n", i, t, f)
}
```

go

The `go` package, while not containing code itself and only other packages, is the place for all the code related to, well, the Go language itself.

There are packages to deal with lexing and parsing Go code into an AST[56], a package to deal with that AST, and a package to print the code from an AST.

There is also a package to look at Go documentation, which the `godoc` binary uses extensively.

The final package is the `build` package, which you probably don't have a use for normally, but the `go` tool builds your code given a few rules in the package.

Cross Platform Go Code

The `go/build` package is pretty simple, and most of it comes into play when you're trying to control what builds in what environment. Let's look at a simple example from the Go source code.

go/path_unix.go

```go
// Copyright 2011 The Go Authors. All rights reserved.
// Use of this source code is governed by a BSD-style
// license that can be found in the LICENSE file.

// +build darwin freebsd linux netbsd openbsd

package os

const (
	PathSeparator     = '/' // OS-specific path separator
	PathListSeparator = ':' // OS-specific path list separator
)

// IsPathSeparator returns true if c is a directory separator character.
func IsPathSeparator(c uint8) bool {
	return PathSeparator == c
}
```

[56] http://en.wikipedia.org/wiki/Abstract_syntax_tree

go/path_windows.go

```go
// Copyright 2011 The Go Authors. All rights reserved.
// Use of this source code is governed by a BSD-style
// license that can be found in the LICENSE file.

package os

const (
	PathSeparator     = '\\' // OS-specific path separator
	PathListSeparator = ';'  // OS-specific path list separator
)

// IsPathSeparator returns true if c is a directory separator character.
func IsPathSeparator(c uint8) bool {
	// NOTE: Windows accept / as path separator.
	return c == '\\' || c == '/'
}
```

Each of these files provides the `PathSeparator` and `PathListSeparator` constants, as well as the `IsPathSeparator` function in the `os` package. The key is in the naming.

Once is named `path_unix.go` and one is named `path_windows.go`. The former gets built when you're compiling for Linux, and the latter when compiling for Windows.

OS Specific

When specifying an operating system for the `build` package, it has to match something that `runtime.GOOS` likes. `darwin`, `freebsd`, `netbsd`, `bsd`, `plan9`, `windows`, `linux`, and `unix` are all valid values. There are others, but you might have to dig a little or run a simple `println(runtime.GOOS)` to see what the value should be for your specific situation.

Some, like `bsd`, mean that the file would get compiled on FreeBSD and NetBSD. If you specified `freebsd`, it would naturally only get compiled on FreeBSD.

Architecture Specific

You can also specify a CPU architecture: `386`, `amd64`, and `arm` are the possible values. Your files would look like `myfile_386.go` or `assembly_amd64.s`.

All Together

You can even combine the two, listing the OS first and architecture second: `myfile_-linux_amd64.go`. These conventions give you seriously easy ways to have all your code in one place and yet remain specific to different situations.

Build Constraints in Comments

If the file naming scheme doesn't feel right to you, or you need even more control, you can always use a comment. If you add a `// +build` comment at the top of your file (preceded by only blank lines or other **line comments**), you can put constraints in there. Simply specify all your conditions with spaces for `AND` and commas for `OR`. You can negate things with `!`, and you can also control whether something is built when `cgo` is used (or not) by using the `cgo` constraint.

Using the example from the documentation, `// +build linux,386 darwin,!cgo` would build Linux 386 or OSX without CGO. It would not get included on Windows, anything BSD, or Linux amd64.

Cool beans right? Check this out…

Custom Build Constraints

You can also use custom tags in your comments to control your build. If you pas `-tags foo` to `go build`, `go install`, or any other command that accepts `go build` flags, the `foo` build constraint is considered to be *met*. This means you can have `// +build foo` in your file and it will be built. If you have `// +build !foo` it will only be built if you *don't* specify the `foo` flag.

You could model `--with-feature` flags in your build this way. Say you have 4 files: `png.go`, `jpg.go`, `gif.go`, and `tiff.go`. Each file has `// +build <ext>` at the top, where `<ext>` is the file extension you're dealing with. Building with `-tags png,jpg,gif` would build with PNG, JPG, and GIF support, but skip TIFF.

Introspecting Packages

You can also use the `build` package to introspect things in your Go environment. The `Import` function gives you back a `build.Package`, which has a lot of information about said package, including the files that make it up, what imports it uses, and other fun things. Check out the full type description[57] for all the good things.

[57] http://golang.org/pkg/go/build/#Package

Here's some code to dump the imports and go files a given package uses.

go/package_info.go

```go
package main

import (
	"flag"
	"go/build"
	"log"
)

var importPath = flag.String("path", "net", "The import path")

func main() {
	flag.Parse()
	pkg, err := build.Import(*importPath, "", 0)
	if err != nil {
		log.Fatalf("failed getting package: %s", err)
	}
	fmt := "package %s imports %d packages, has %d go files in %s"
	log.Printf(fmt, pkg.Name, len(pkg.Imports), len(pkg.GoFiles), pkg.Dir)
	log.Println("imports")
	for _, imp := range pkg.Imports {
		log.Printf("\t%s", imp)
	}
	log.Println("go files")
	for _, file := range pkg.GoFiles {
		log.Printf("\t%s", file)
	}
}
```

Lexing Go Code

Lexing, or lexical analysis, is the process of turn a big blob of bytes (the file) into *tokens* which can be used by something else (usually the parser). The tokens are things like identifier, string, left curly brace, etc.

It's pretty straight forward to deal do this, so let's get right to it.

go/lexing.go

```go
package main

import (
	"go/scanner"
	"go/token"
	"io/ioutil"
	"log"
)

func main() {
	src, err := ioutil.ReadFile("lexing.go") // This file!
	if err != nil {
		log.Fatalf("failed reading source file: %s", err)
	}

	fset := token.NewFileSet()
	file := fset.AddFile("lexing.go", fset.Base(), len(src))
	var s scanner.Scanner
	format := "found a %s as %#v on line %d at column %d"
	s.Init(file, src, nil, 0)
	for {
		pos, tok, lit := s.Scan()
		if tok == token.EOF {
			break
		}
		position := fset.Position(pos)
		log.Printf(format, tok, lit, position.Line, position.Column)
	}
}
```

There's nothing too exciting going on, it's fairly standard code for setting something up and then grabbing piece after piece until it's done. You can see in some cases `lit` is an empty string because it wouldn't hold anything relevant anyway. If the `Token` is already identified as being `}`, we don't need `lit` to be the string `"}"` as well.

> It does let you see nice and clearly how semicolons work in Go. Not once in the file did I use a semicolon, but they are coming out of the lexer. Give the section in the

spec on semicolons[a] a read again to understand the specific rules behind this.

[a] http://golang.org/ref/spec#Semicolons

Parsing Go Code

Parsing is what happens after lexing. Parsing takes the tokens generated by the lexer and builds an *Abstract Syntax Tree*.[58]

The `go/parser` package part of the picture, since it gives you things from the `go/ast` package. You start with the `parser` package, but you'll probably spend most of your time dealing with things from the `ast` package.

You can parse a file, a directory of files, and even a simple expression. Once you have an AST you can print it to see what it's all about, or do other fun things, which we'll see later. Printing the tree is a good start, as it gives you a much better idea of how Go is representing itself.

go/parsing.go

```go
package main

import (
	"go/ast"
	"go/parser"
	"go/token"
	"log"
)

func main() {
	fset := token.NewFileSet()
	f, err := parser.ParseFile(fset, "parsing.go", nil, 0)
	if err != nil {
		log.Fatalf("failed parsing file: %s", err)
	}
	ast.Print(fset, f)

	expr, err := parser.ParseExpr(`foo.Bar(1, "argument", something())`)
	if err != nil {
		log.Fatal("failed parsing expression: %s", err)
```

[58] http://en.wikipedia.org/wiki/Abstract_syntax_tree

```
        }
        ast.Print(nil, expr)
}
```

Analyzing Go Code: Cyclomatic Complexity

Once you've parsed your code and get bored just printing things, you need to get to some analyzing. We're going to calculate the cyclomatic complexity of the functions and methods defined in a file.

Cyclomatic complexity is basically the number of decisions plus one. A decision is an `if` statement, a `case` in a `switch`, a condition in a loop (infinite loops don't count), and the binary `&&` and `||` ops. We'll want to walk down the AST for each function and method, and sum the number of these things we see.

Since you have a tree, there are many algorithms to walk down a tree and visit all the nodes. Looking at the `ast.File` type you get back from the `parser` package, there doesn't seem to be any easily useable structure on it to walk down. Oh wait, there's a `Walk` function in the `ast` package! Let's use that, to walk the tree and do something useful.

In our example, we use two different `Visitor` implementations. The first walks over the top level of a file, and finds all the function and method declarations. When it finds one, it walks the node with another `Visitor` to do the actual calculation. It's not terribly long, so give it a good read.

go/analyzing.go

```
package main

import (
        "bytes"
        "flag"
        "go/ast"
        "go/parser"
        "go/printer"
        "go/token"
        "log"
)

var path = flag.String("path", "analyzing.go", "The path to the file to parse and ex\
amine")
```

```go

func funcDeclToString(decl *ast.FuncDecl) string {
	var buffer bytes.Buffer
	var body *ast.BlockStmt
	body, decl.Body = decl.Body, nil
	printer.Fprint(&buffer, token.NewFileSet(), decl)
	decl.Body = body
	return buffer.String()
}

type ComplexityCalculator struct {
	Name       string
	Complexity int
}

func (cc *ComplexityCalculator) Visit(node ast.Node) ast.Visitor {
	switch exp := node.(type) {
	case *ast.IfStmt, *ast.CaseClause:
		cc.Complexity++
	case *ast.BinaryExpr:
		switch exp.Op {
		case token.LAND, token.LOR:
			cc.Complexity++
		}
	case *ast.ForStmt:
		if exp.Cond != nil {
			cc.Complexity++
		}
	}
	return cc
}

type FuncVisitor struct {
	FuncComplexities []*ComplexityCalculator
}

func (mv *FuncVisitor) Visit(node ast.Node) ast.Visitor {
	switch exp := node.(type) {
	case *ast.FuncDecl:
		cc := &ComplexityCalculator{
			Name:       funcDeclToString(exp),
			Complexity: 1,
		}
```

```
			mv.FuncComplexities = append(mv.FuncComplexities, cc)
			ast.Walk(cc, node)
			return nil // Return nil to stop this walk.
		}
		return mv
}

func main() {
		flag.Parse()
		fset := token.NewFileSet()
		f, err := parser.ParseFile(fset, *path, nil, 0)
		if err != nil {
			log.Fatalf("failed parsing file: %s", err)
		}
		var mv FuncVisitor
		ast.Walk(&mv, f)
		for _, mc := range mv.FuncComplexities {
			log.Printf("%s has complexity %d", mc.Name, mc.Complexity)
		}
}
```

Altering Go Code: Mutation Testing

As you walk down a tree, there is nothing stopping you from changing the nodes as you go, pun intended. This is exactly what `go fix` does.

We're going to look at using this to do mutation testing. Mutation testing is really testing your tests. You go through your source code, and alter things. Things like changing `==` to `!=`. You then run your tests, and something should fail. If nothing fails, you're missing some coverage with your tests.

> When I set out to write this chapter, I had this use case in mind. A quick Google lead me to Kamil Kisiel's mutatator[a] library he hacked up in response to a discussion on the golang-nuts mailing list[b]. While I'm not using all of his code directly, I am using it as a base for my example. I really like his use of an immediately executing function in the meat of the program to change the token in the AST but also ensure it gets set back. He gave me the go ahead to use his code as my inspiration, so thanks

to Kamil.

[a] https://github.com/kisielk/mutator
[b] https://groups.google.com/forum/?fromgroups#!forum/golang-nuts

So we're going to build a mutation testing executable. You give it a package and an operation to switch, it copies everything to a temporary directory, and runs through all the possible mutations, running tests for each, to see if the tests fail. All it has to do to mutate is change the `Op` field of the `ast.BinaryExpr` and write out the AST using the `go/printer` package. The `defer` inside the `RunMutation` function ensures the mutation gets reversed so as to not taint the run for subsequent mutations.

go/altering.go

```go
package main

import (
        "bytes"
        "flag"
        "fmt"
        "go/ast"
        "go/build"
        "go/parser"
        "go/printer"
        "go/token"
        "io"
        "io/ioutil"
        "log"
        "os"
        "os/exec"
        "path/filepath"
)

var (
        code     = 0
        name     = flag.String("pkg", "crypto/sha256", "The package to mutate")
        mutation = flag.String("mutation", "==", "The mutation")
        list     = flag.Bool("list", false, "Print available things to mutate")
)

var operators = map[string]token.Token{
        "==": token.EQL,
        "!=": token.NEQ,
```

```go
	">":  token.GTR,
	"<":  token.LSS,
	">=": token.GEQ,
	"<=": token.LEQ,
	"&&": token.LAND,
	"||": token.LOR,
	"&":  token.AND,
	"|":  token.OR,
}

var mutations = map[token.Token][]token.Token{
	token.EQL:  {token.NEQ},
	token.NEQ:  {token.EQL},
	token.GTR:  {token.LSS, token.GEQ, token.LEQ},
	token.LSS:  {token.GTR, token.LEQ, token.GEQ},
	token.GEQ:  {token.GTR, token.LEQ, token.LSS},
	token.LEQ:  {token.LSS, token.GEQ, token.GTR},
	token.LOR:  {token.LAND},
	token.LAND: {token.LOR},
	token.OR:   {token.AND},
	token.AND:  {token.OR},
}

type ExpressionFinder struct {
	Token token.Token
	Exps  []*ast.BinaryExpr
}

func (v *ExpressionFinder) Visit(node ast.Node) ast.Visitor {
	if exp, ok := node.(*ast.BinaryExpr); ok {
		if exp.Op == v.Token {
			v.Exps = append(v.Exps, exp)
		}
	}
	return v
}

func (v ExpressionFinder) Len() int {
	return len(v.Exps)
}

func copyFile(src, dir string) error {
	name := filepath.Base(src)
```

```
        srcFile, err := os.Open(src)
        if err != nil {
                return err
        }
        defer srcFile.Close()

        dstFile, err := os.Create(filepath.Join(dir, name))
        if err != nil {
                return err
        }
        defer dstFile.Close()

        _, err = io.Copy(dstFile, srcFile)
        return err
}

func copyFiles(src, dst string) {
        contents, err := ioutil.ReadDir(src)
        if err != nil {
                log.Fatalf("failed reading directory: %s", err)
        }
        for _, f := range contents {
                if f.Mode()&os.ModeType == 0 {
                        err := copyFile(filepath.Join(src, f.Name()), dst)
                        if err != nil {
                                log.Fatalf("failed copying %s: %s", f.Name(), err)
                        }
                }
        }
}

func RunMutation(index int, exp *ast.BinaryExpr, f, t token.Token, src string, fset \
*token.FileSet, file *ast.File) error {
        exp.Op = t
        defer func() {
                exp.Op = f
        }()

        err := printFile(src, fset, file)
        if err != nil {
                return err
        }

```

```go
	cmd := exec.Command("go", "test")
	cmd.Dir = filepath.Dir(src)
	output, err := cmd.CombinedOutput()
	if err == nil {
		code = 1
		log.Printf("mutation %d failed to break any tests", index)
	} else if _, ok := err.(*exec.ExitError); ok {
		lines := bytes.Split(output, []byte("\n"))
		lastLine := lines[len(lines)-2]
		if bytes.HasPrefix(lastLine, []byte("FAIL")) {
			log.Printf("mutation %d failed the tests properly", index)
		} else {
			log.Printf("mutation %d created an error: %s", index, lastLine)
		}
	} else {
		return fmt.Errorf("mutation %d failed to run: %s", index, err)
	}
	return nil
}

func MutateFile(src string, f, t token.Token) error {
	fset := token.NewFileSet()

	file, err := parser.ParseFile(fset, src, nil, 0)
	if err != nil {
		return fmt.Errorf("failed to parse %s: %s", src, err)
	}

	ef := ExpressionFinder{Token: f}
	ast.Walk(&ef, file)

	filename := filepath.Base(src)
	log.Printf("found %d occurrences of %s in %s", ef.Len(), f, filename)
	for index, exp := range ef.Exps {
		err := RunMutation(index, exp, f, t, src, fset, file)
		if err != nil {
			return err
		}
	}

	// Restore the original file
	err = printFile(src, fset, file)
	if err != nil {
```

```go
			return err
		}
		return nil
}

func printFile(path string, fset *token.FileSet, node interface{}) error {
		file, err := os.OpenFile(path, os.O_WRONLY|os.O_TRUNC, 0)
		if err != nil {
			return fmt.Errorf("failed to open output file: %s", err)
		}
		defer file.Close()

		err = printer.Fprint(file, fset, node)
		if err != nil {
			return fmt.Errorf("failed to write AST to file: %s", err)
		}
		return nil
}

func main() {
		flag.Parse()

		if *list {
			for thing, _ := range operators {
				fmt.Printf("%s\n", thing)
			}
			os.Exit(0)
		}

		from, ok := operators[*mutation]
		if !ok {
			log.Fatalf("%#v is not a valid mutation", *mutation)
		}

		pkg, err := build.Import(*name, "", 0)
		if err != nil {
			log.Fatalf("failed to import package: %s", err)
		}

		tmp, err := ioutil.TempDir("", "mutation")
		if err != nil {
			log.Fatalf("failed to create tmp directory: %s", err)
		}
```

```go

	log.Printf("mutating in %s", tmp)

	copyFiles(pkg.Dir, tmp)

	for _, f := range pkg.GoFiles {
		src := filepath.Join(tmp, f)
		for _, to := range mutations[from] {
			log.Printf("mutating %s to %s in %s", from, to, f)
			err := MutateFile(src, from, to)
			if err != nil {
				log.Fatalf("failed mutating file: %s", err)
			}
		}
	}
	os.Exit(code)
}
```

If we run this as `go run altering.go -pkg "crypto/sha256"` we see that it mutates the `==` operator to `!=`, and the tests break as they should. If we run it as `go run altering.go -pkg "crypto/sha256" -mutation "<"` there is a mutation that doesn't fail the tests. Specifically, it's the mutation of `<` to `<=` on line 147 of `sha256.go`: `for i := uint(0); i < 8; i++ {`. Looking at the code, we can see it's not a problem.

go/failed_mutation.txt

```
2013/03/09 22:28:23 mutating in /var/folders/t2/k4y07r396d5006j7y9w9zldc0000gn/T/mut\
ation867582255
2013/03/09 22:28:23 mutating < to > in sha256.go
2013/03/09 22:28:23 found 3 occurrences of < in sha256.go
2013/03/09 22:28:23 mutation 0 failed the tests, as it should
2013/03/09 22:28:24 mutation 1 failed the tests, as it should
2013/03/09 22:28:24 mutation 2 failed the tests, as it should
2013/03/09 22:28:24 mutating < to <= in sha256.go
2013/03/09 22:28:24 found 3 occurrences of < in sha256.go
2013/03/09 22:28:24 mutation 0 failed the tests, as it should
2013/03/09 22:28:24 mutation 1 failed the tests, as it should
2013/03/09 22:28:25 mutation 2 failed to break any tests
2013/03/09 22:28:25 mutating < to >= in sha256.go
2013/03/09 22:28:25 found 3 occurrences of < in sha256.go
2013/03/09 22:28:25 mutation 0 failed the tests, as it should
2013/03/09 22:28:25 mutation 1 failed the tests, as it should
2013/03/09 22:28:25 mutation 2 failed the tests, as it should
```

```
2013/03/09 22:28:25 mutating < to > in sha256block.go
2013/03/09 22:28:25 found 3 occurrences of < in sha256block.go
2013/03/09 22:28:26 mutation 0 failed the tests, as it should
2013/03/09 22:28:26 mutation 1 failed the tests, as it should
2013/03/09 22:28:26 mutation 2 failed the tests, as it should
2013/03/09 22:28:26 mutating < to <= in sha256block.go
2013/03/09 22:28:26 found 3 occurrences of < in sha256block.go
2013/03/09 22:28:27 mutation 0 failed the tests, as it should
2013/03/09 22:28:27 mutation 1 failed the tests, as it should
2013/03/09 22:28:27 mutation 2 failed the tests, as it should
2013/03/09 22:28:27 mutating < to >= in sha256block.go
2013/03/09 22:28:27 found 3 occurrences of < in sha256block.go
2013/03/09 22:28:27 mutation 0 failed the tests, as it should
2013/03/09 22:28:28 mutation 1 failed the tests, as it should
2013/03/09 22:28:28 mutation 2 failed the tests, as it should
exit status 1
```

This example has a small subset of the possible mutations you can do. The simple ones listed in the example include just change a basic binary operator. More advanced ones include changing constants in the code: 0 to a 1, changing strings to be nonsense, or just cut them in half. I'm sure your imagination can figure out a few more diabolical mutations.

hash

The `hash` package contains the interface for all things hash related. The main package provides the interface, including separate interfaces for 32 and 64-bit. The 4 sub-packages provide implementations for 3 different checksums (`adler` 32-bit and `crc` in both 32 and 64-bit), and the `fnv` non-cryptographic hash.

We've already seen the cryptographic hashes and other things implementing the `hash` interface in the `crypto` package: `sha1`, `sha256`, `sha512`, `md5`, and `hmac`. These operate the same way as the things in the `hash` package, because they follow the same interface.

While the algorithms in the `hash` package all follow the `hash.Hash` interface, they sometimes have different ways of building that interface, so we'll look at them separately.

adler32

The `adler32` package implements the Adler-32 checksum as defined in RFC-1950[59]. It provides the `New()` function to build a `hash.Hash`, and also a convenience `Checksum([]byte)` function if all you have is a simple `byte` slice.

hash/adler32.go

```
package main

import (
	"flag"
	"hash/adler32"
	"io"
	"io/ioutil"
	"log"
	"os"
)

var (
	filename  = flag.String("filename", "adler32.go", "The file to checksum")
	streaming = flag.Bool("streaming", false, "Whether to stream the file instead of re\
```

[59] http://www.ietf.org/rfc/rfc1950.txt

```go
ading it all into memory")
)

func stream(name string) uint32 {
	h := adler32.New()
	file, err := os.Open(name)
	if err != nil {
		log.Fatalf("failed opening %s: %s", name, err)
	}
	defer file.Close()
	io.Copy(h, file)
	return h.Sum32()
}

func simple(name string) uint32 {
	data, err := ioutil.ReadFile(name)
	if err != nil {
		log.Fatalf("failed reading %s: %s", name, err)
	}
	return adler32.Checksum(data)
}

func main() {
	flag.Parse()
	var checksum uint32

	if *streaming {
		checksum = stream(*filename)
	} else {
		checksum = simple(*filename)
	}

	log.Printf("the file %s has checksum %#x", *filename, checksum)
}
```

crc32

The `crc32` package implements 32-bit CRC. It supplies 3 different *polynomials*[60] for common use cases: the IEEE polynomial (most common) which is used in ethernet,

[60] http://en.wikipedia.org/wiki/Cyclic_redundancy_check#Designing_CRC_polynomials

gzip, etc, Castagnoli's which is used in iSCSI, and Koopman's polynomial. Being so common, there are convenience helpers for IEEE checksums.

hash/crc32.go

```go
package main

import (
	"flag"
	"fmt"
	"hash/crc32"
	"io"
	"io/ioutil"
	"log"
	"os"
	"strings"
)

type Polynomial struct {
	U uint32
}

var polynomials = map[string]uint32{
	"ieee":       crc32.IEEE,
	"castagnoli": crc32.Castagnoli,
	"koopman":    crc32.Koopman,
}

func (p *Polynomial) Set(s string) error {
	switch s {
	case "ieee", "castagnoli", "koopman":
		p.U = polynomials[s]
	default:
		var values []string
		for name, _ := range polynomials {
			values = append(values, name)
		}
		return fmt.Errorf("valid values are %s", strings.Join(values, ", "))
	}
	return nil
}

func (p *Polynomial) String() string {
	for name, value := range polynomials {
```

```
			if value == p.U {
				return fmt.Sprintf("%s", name)
			}
		}
	panic("not reached")
}

func (p *Polynomial) Table() *crc32.Table {
	return crc32.MakeTable(p.U)
}

var (
	filename   = flag.String("filename", "crc32.go", "The file to checksum")
	streaming  = flag.Bool("streaming", false, "Whether to stream the file instead of r\
eading it all into memory")
	polynomial = &Polynomial{crc32.IEEE}
)

func init() {
	flag.Var(polynomial, "polynomial", "The polynomial to use")
	flag.Parse()
}

func stream(name string) uint32 {
	h := crc32.New(polynomial.Table())
	file, err := os.Open(name)
	if err != nil {
		log.Fatalf("failed opening %s: %s", name, err)
	}
	defer file.Close()
	io.Copy(h, file)
	return h.Sum32()
}

func simple(name string) uint32 {
	data, err := ioutil.ReadFile(name)
	if err != nil {
		log.Fatalf("failed reading %s: %s", name, err)
	}
	return crc32.Checksum(data, polynomial.Table())
}

func main() {
```

```go
	var checksum uint32

	if *streaming {
		checksum = stream(*filename)
	} else {
		checksum = simple(*filename)
	}

	log.Printf("the file %s has checksum %#x", *filename, checksum)
}
```

crc64

As the named suggests, `crc64` implements the 64-bit CRC. Like `crc32` it provides some predefined polynomials, but neither of the two provided are exciting enough to warrant convenience helper functions. You have to build and use your own `crc64.Table` as I did in the previous example using the polynomial constants provided (or your own value if you know what you're doing). Luckily there's a function to make the table from a given polynomial.

The example is identical to the `crc32` example, except for the polynomial map, and `uint32` becomes `uint64`.

hash/crc64.go

```go
package main

import (
	"flag"
	"fmt"
	"hash/crc64"
	"io"
	"io/ioutil"
	"log"
	"os"
	"strings"
)

type Polynomial struct {
	U uint64
}
```

```

var polynomials = map[string]uint64{
	"iso":  crc64.ISO,
	"ecma": crc64.ECMA,
}

func (p *Polynomial) Set(s string) error {
	switch s {
	case "iso", "ecma":
		p.U = polynomials[s]
	default:
		var values []string
		for name, _ := range polynomials {
			values = append(values, name)
		}
		return fmt.Errorf("valid values are %s", strings.Join(values, ", "))
	}
	return nil
}

func (p *Polynomial) String() string {
	for name, value := range polynomials {
		if value == p.U {
			return fmt.Sprintf("%s", name)
		}
	}
	panic("not reached")
}

func (p *Polynomial) Table() *crc64.Table {
	return crc64.MakeTable(p.U)
}

var (
	filename   = flag.String("filename", "crc64.go", "The file to checksum")
	streaming  = flag.Bool("streaming", false, "Whether to stream the file instead of r\
eading it all into memory")
	polynomial = &Polynomial{crc64.ISO}
)

func init() {
	flag.Var(polynomial, "polynomial", "The polynomial to use")
	flag.Parse()
```

```
}

func stream(name string) uint64 {
	h := crc64.New(polynomial.Table())
	file, err := os.Open(name)
	if err != nil {
		log.Fatalf("failed opening %s: %s", name, err)
	}
	defer file.Close()
	io.Copy(h, file)
	return h.Sum64()
}

func simple(name string) uint64 {
	data, err := ioutil.ReadFile(name)
	if err != nil {
		log.Fatalf("failed reading %s: %s", name, err)
	}
	return crc64.Checksum(data, polynomial.Table())
}

func main() {
	var checksum uint64

	if *streaming {
		checksum = stream(*filename)
	} else {
		checksum = simple(*filename)
	}

	log.Printf("the file %s has checksum %#x", *filename, checksum)
}
```

fnv

The `fnv` package implements the fnv hash[61] and has no special convenience helper functions. It simply provides a 32-bit and 64-bit `hash.Hash` implementation. This is a hash algorithm as opposed to a checksum, so the `Checksum([]byte)` helpers functions we saw before don't make sense anyway.

[61] http://isthe.com/chongo/tech/comp/fnv/

hash/fnv.go

```go
package main

import (
	"flag"
	"hash/fnv"
	"io"
	"log"
	"os"
)

var (
	filename = flag.String("filename", "fnv.go", "The file to checksum")
	_64bit   = flag.Bool("64", false, "Use the 64-bit interface")
)

func runHash(name string, w io.Writer) {
	file, err := os.Open(name)
	if err != nil {
		log.Fatalf("failed opening %s: %s", name, err)
	}
	defer file.Close()
	io.Copy(w, file)
}

func hash64(name string) uint64 {
	h := fnv.New64()
	runHash(name, h)
	return h.Sum64()
}

func hash32(name string) uint32 {
	h := fnv.New32()
	runHash(name, h)
	return h.Sum32()
}

func main() {
	flag.Parse()
	if *_64bit {
		h := hash64(*filename)
		log.Printf("the file %s has hash %#x", *filename, h)
```

```
	} else {
		h := hash32(*filename)
		log.Printf("the file %s has hash %#x", *filename, h)
	}
}
```

html

The `html` package on its own isn't all that exciting. Two functions! Woohoo! Well, they might not be exciting, but they are useful.

The real meat of the `html` package is the `html/template` package inside it. While that package itself is really just an extension of the `text/template` package, it does some fancy things to make your life easier, and keep your app safer, when rendering HTML templates.

We'll start off with a single example of the `EscapeString` and `UnescapeString` functions from the base `html` package, but we'll spend most of our time building templates.

Escape Artist

The two escaping functions are very easy to use, and very self explanatory.

`EscapeString` takes a string and escapes it for use in HTML. It only deals with 5 characters though: angle brackets, quotes (single and double), and the ampersand. Really, these characters are the ones that will cause the most havoc.

`UnescapeString` takes an escaped string and reverses the process. It does a bit more though. It can handle HTML entities, like converting `á` to `á`. For this reason, the official package documentation provides the following caveat:

> `UnescapeString(EscapeString(s)) == s` always holds, but the converse isn't always true.

Let's see some code.

html/escaping.go

```go
package main

import (
	"html"
	"log"
)

func init() {
	log.SetFlags(0)
	log.SetPrefix("")
}

func main() {
	raw := []string{
		"hello",
		"<i>hello</i>",
		"alert('hello');",
		"foo & bar",
		`"how are you?" he asked.`,
	}

	log.Println("html.EscapeString")
	for _, s := range raw {
		log.Printf("\t%s -> %s", s, html.EscapeString(s))
	}

	log.Println("html.UnescapeString(html.EscapeString)")
	for _, s := range raw {
		flipped := html.UnescapeString(html.EscapeString(s))
		log.Printf("\t%s -> %s", s, flipped)
	}

	escaped := []string{
		"&#225;",
		"&raquo;",
		"&middot;",
		"&lt;i&gt;hello&lt;/i&gt;",
	}

	log.Println("html.UnescapeString")
	for _, s := range escaped {
```

```
                log.Printf("\t%s -> %s", s, html.UnescapeString(s))
        }
}
```

Output:

```
html.EscapeString
        hello -> hello
        <i>hello</i> -> &lt;i&gt;hello&lt;/i&gt;
        alert('hello'); -> alert(&#39;hello&#39;);
        foo & bar -> foo &amp; bar
        "how are you?" he asked. -> &#34;how are you?&#34; he asked.
html.UnescapeString(html.EscapeString)
        hello -> hello
        <i>hello</i> -> <i>hello</i>
        alert('hello'); -> alert('hello');
        foo & bar -> foo & bar
        "how are you?" he asked. -> "how are you?" he asked.
html.UnescapeString
        &#225; -> á
        &raquo; -> »
        &middot; -> ·
        &lt;i&gt;hello&lt;/i&gt; -> <i>hello</i>
```

Templating

When you first look at the package documentation for the `html/template` package, you might think, “cool story bro, but how do I actually use this?”

I know I did initially, but I just didn’t read hard enough.

Since the `html/template` package uses the same idea as the `text/template` package, you should go read the documentation for that package to get an idea of the basic usage. The `html/template` docs include things specific to it, like how things are escaped, extra/special functions or helpers, and stuff like that.

Naturally, in our examples, we’ll only look at things specific to the `html/template` package, and leave the basics for the `text/template` chapter.

Code time!

html/templating.go

```go
package main

import (
	T "html/template"
	"os"
)

const (
	template = `<html>
  <head>
    <link href="http://fonts.googleapis.com/css?family={{.FontName}}" rel="styleshee\
t" type="text/css">
  </head>
  <body>
    {{.Script}}
    {{.Safe}}
  </body>
</html>
`
)

func main() {
	context := struct {
		FontName string
		Script   string
		Safe     T.HTML
	}{
		"Pathway Gothic One",
		"<script>alert('i haz ur cookies');</script>",
		T.HTML("<script>console.log('generated by application')</script>"),
	}

	t := T.Must(T.New("thestdlib").Parse(template))
	t.Execute(os.Stdout, context)
}
```

Output:

```
<html>
  <head>
    <link href="http://fonts.googleapis.com/css?family=Pathway%20Gothic%20One" rel="\
stylesheet" type="text/css">
  </head>
  <body>
    &lt;script&gt;alert(&#39;i haz ur cookies&#39;);&lt;/script&gt;
    <script>console.log('generated by application')</script>
  </body>
</html>
```

There are a few important parts in this example. First is how I use the `FontName` attribute as a query value in a link tag. The `html/template` package knows the context, and properly escapes the `string` in a way suitable for the context. A more complete list of how things are escaped in various context is given in the package docs[62] so I won't repeat them here. The point is, the package is pretty smart about how things should be escaped, and they are escaped by default.

If you don't want things escaped, we can use the `template.HTML` type (which really just points at `string`). If you use something of this type in the appropriate context, it won't be escaped. There are similar types for JavaScript and CSS (`template.JS` and `templates.CSS`) and HTML attributes (`template.HTMLAttr`).

We can see the difference between using a regular string to put a JavaScript tag to the page vs using the `template.HTML` type. The former gets escaped, while the latter goes in untouched.

That's all there is to this templating system. It works just like `text/template` (which we'll see more completely later), but does some smart escaping.

If you're generating HTML, you should be using this package.

[62] http://golang.org/pkg/html/template/

image

The `image` package, as you might expect, deals with 2-D images. It can handle decoding `gif`, and can both encode and decode `jpg` and `png` images.

It also include a basic color library, as well as a library for compositing images.

No more installing PIL for you!

Typically, you'll work with the `image` package to decode an image, and use the `image.Image` interface. To enforce this, the only useful functions from the `image/jpg` and `image/png` packages are the ones to encode an `image.Image`.

You'll also work with the `color.Color` interface when dealing with pixels in the image, using the `At(x, y int) color.Color` method.

Converting images formats

Ensuring all your images are in a certain format might be something you want to do, so let's try that first.

It's pretty easy. Decode the image, then encode the image. Boom, done!

> Since the `image/gif` package doesn't have an `Encode` function, we just need to import with an underscore it to register the decoder. The `image/jpeg` and `image/png` packages can both encode, so we import them normally.

image/convert.go

```go
package main

import (
	"flag"
	"image"
	_ "image/gif"
	"image/jpeg"
	"image/png"
	"io"
```

```
	"log"
	"os"
)

var (
	jpgout = flag.String("jpg", "", "output to a jpg")
	pngout = flag.String("png", "", "output to a png")
	in     = flag.String("in", "", "input file")
)

type encf func(io.Writer, image.Image) error

func encode(encoder encf, img image.Image, filename string) {
	file, err := os.OpenFile(filename, os.O_WRONLY|os.O_CREATE, 0644)
	if err != nil {
		log.Printf("failed opening %s: %s", filename, err)
		return
	}
	defer file.Close()
	err = encoder(file, img)
	if err != nil {
		log.Printf("failed encoding to %s: %s", filename, err)
	}
}

func jpegEncode(w io.Writer, m image.Image) error {
	return jpeg.Encode(w, m, &jpeg.Options{Quality: 80})
}

func decode(filename string) image.Image {
	file, err := os.Open(filename)
	if err != nil {
		log.Fatalf("failed opening file: %s", err)
	}
	defer file.Close()

	img, _, err := image.Decode(file)
	if err != nil {
		log.Fatalf("failed decoding image: %s", err)
	}
	return img
}

```

```go
func main() {
	flag.Parse()

	img := decode(*in)

	if *pngout != "" {
		encode(png.Encode, img, *pngout)
	}

	if *jpgout != "" {
		encode(jpegEncode, img, *jpgout)
	}
}
```

Resizing

Resizing images and making thumbnails is a pretty common task too, so let's try that. I've used the simplest algorithm, nearest neighbour. You can replace the part commented as `// The important stuff` with some other algorithm, like bilinear interpolation.

image/resize.go

```go
package main

import (
	"flag"
	"image"
	"image/jpeg"
	"image/png"
	"io"
	"log"
	"os"
)

var (
	jpgout = flag.String("jpg", "", "output to a jpg")
	pngout = flag.String("png", "", "output to a png")
	in     = flag.String("in", "", "input file")
	size   = flag.Int("size", 0, "the new max dimension")
)
```

```go

type encf func(io.Writer, image.Image) error

func encode(encoder encf, img image.Image, filename string) {
	file, err := os.OpenFile(filename, os.O_WRONLY|os.O_CREATE, 0644)
	if err != nil {
		log.Printf("failed opening %s: %s", filename, err)
		return
	}
	defer file.Close()
	err = encoder(file, img)
	if err != nil {
		log.Printf("failed encoding to %s: %s", filename, err)
	}
}

func jpegEncode(w io.Writer, m image.Image) error {
	return jpeg.Encode(w, m, &jpeg.Options{Quality: 80})
}

func round(value float32) int {
	if value < 0.0 {
		value -= 0.5
	} else {
		value += 0.5
	}
	return int(value)
}

func scale(w, h, size int) (int, int, float32) {
	var factor float32
	width, height := float32(w), float32(h)
	if width > height {
		factor = float32(size) / width
	} else {
		factor = float32(size) / height
	}
	return round(factor * width), round(factor * height), factor
}

func resize(img image.Image, nsize int) image.Image {
	osize := img.Bounds().Size()
	nwidth, nheight, factor := scale(osize.X, osize.Y, nsize)
```

```go
	nimg := image.NewRGBA(image.Rect(0, 0, nwidth, nheight))
	for y := 0; y < nheight; y++ {
		for x := 0; x < nwidth; x++ {
			// The important stuff
			fx, fy := round(float32(x)/factor), round(float32(y)/factor)
			nimg.Set(x, y, img.At(fx, fy))
		}
	}
	return nimg
}

func decode(filename string) image.Image {
	file, err := os.Open(filename)
	if err != nil {
		log.Fatalf("failed opening file: %s", err)
	}
	defer file.Close()

	img, _, err := image.Decode(file)
	if err != nil {
		log.Fatalf("failed decoding image: %s", err)
	}
	return img
}

func main() {
	flag.Parse()
	if *size <= 0 {
		log.Fatalln("size must be greater than 0")
	}
	img := decode(*in)
	img = resize(img, *size)

	if *pngout != "" {
		encode(png.Encode, img, *pngout)
	}

	if *jpgout != "" {
		encode(jpegEncode, img, *jpgout)
	}
}
```

Cropping

Cropping an image, like grabbing the face from a larger image for thumbnail purposes, is another of the basic things everybody does with images. Let's see how we can do that. This is our first look at the `image/draw` package.

The `image/draw` package, while very basic, is very powerful. It's modeled after a paper by Thomas Porter and Tom Duff and gives you the basic primitives to do anything. Cropping is one of those basic primitives. It's actually pretty simple, but it took me a minute to sort it out.[63]

image/cropping.go

```go
package main

import (
	"flag"
	"fmt"
	"image"
	"image/draw"
	"image/jpeg"
	"image/png"
	"io"
	"log"
	"os"
)

type Cropping struct {
	Width, Height uint
	X, Y          int
}

func (c *Cropping) String() string {
	return fmt.Sprintf("%dx%d%+d%+d", c.Width, c.Height, c.X, c.Y)
}

func (c *Cropping) Set(s string) error {
	_, err := fmt.Sscanf(s, "%dx%d%d%d", &c.Width, &c.Height, &c.X, &c.Y)
	return err
}

```

[63] I'm not familiar with the Porter-Duff compositing paper, maybe I should read it. Sort of like learning about category theory so you can do IO in Haskell…

```go
var (
	jpgout   = flag.String("jpg", "", "output to a jpg")
	pngout   = flag.String("png", "", "output to a png")
	in       = flag.String("in", "", "input file")
	cropping = new(Cropping)
)

func init() {
	flag.Var(cropping, "crop", "crop to perform, like imagemagick WxH[-+]x[-+]y")
}

type encf func(io.Writer, image.Image) error

func encode(encoder encf, img image.Image, filename string) {
	file, err := os.OpenFile(filename, os.O_WRONLY|os.O_CREATE, 0644)
	if err != nil {
		log.Printf("failed opening %s: %s", filename, err)
		return
	}
	defer file.Close()
	err = encoder(file, img)
	if err != nil {
		log.Printf("failed encoding to %s: %s", filename, err)
	}
}

func jpegEncode(w io.Writer, m image.Image) error {
	return jpeg.Encode(w, m, &jpeg.Options{Quality: 80})
}

func decode(filename string) image.Image {
	file, err := os.Open(filename)
	if err != nil {
		log.Fatalf("failed opening file: %s", err)
	}
	defer file.Close()

	img, _, err := image.Decode(file)
	if err != nil {
		log.Fatalf("failed decoding image: %s", err)
	}
	return img
}
```

```go

func crop(img image.Image, c *Cropping) image.Image {
	r := image.Rect(0, 0, int(c.Width), int(c.Height))
	dst := image.NewRGBA(r)
	draw.Draw(dst, r, img, image.Pt(c.X, c.Y), draw.Src)
	return dst
}

func main() {
	flag.Parse()

	img := decode(*in)
	img = crop(img, cropping)

	if *pngout != "" {
		encode(png.Encode, img, *pngout)
	}

	if *jpgout != "" {
		encode(jpegEncode, img, *jpgout)
	}
}
```

Compositing: Building images from other images

Combining two images is the final basic building block that will let us do all sorts of fun things. We'll write a little program to add a border to an image, which is really just compositing one image on top of a slightly larger image of a solid color. This uses the same technique as the previous cropping example, but we do a little more work.

image/compositing.go

```go
package main

import (
	"flag"
	"fmt"
	"image"
	"image/color"
	"image/draw"
```

```
        "image/jpeg"
        "image/png"
        "io"
        "log"
        "os"
)

type Color struct {
        RGBA uint32
}

func (c *Color) String() string {
        return fmt.Sprintf("%#x", c.RGBA)
}

func (c *Color) Set(s string) error {
        _, err := fmt.Sscanf(s, "%x", &c.RGBA)
        return err
}

func (c *Color) ToRGBA() color.RGBA {
        var mask uint32 = 0xff
        return color.RGBA{
                R: uint8((c.RGBA >> 24) & mask),
                G: uint8((c.RGBA >> 16) & mask),
                B: uint8((c.RGBA >> 8) & mask),
                A: uint8(c.RGBA & mask),
        }
}

var (
        jpgout      = flag.String("jpg", "", "output to a jpg")
        pngout      = flag.String("png", "", "output to a png")
        in          = flag.String("in", "", "input file")
        width       = flag.Int("width", 25, "width of the border")
        borderColor = new(Color)
)

func init() {
        flag.Var(borderColor, "color", "the color of the border in RGBA")
}

type encf func(io.Writer, image.Image) error
```

```
func encode(encoder encf, img image.Image, filename string) {
        file, err := os.OpenFile(filename, os.O_WRONLY|os.O_CREATE, 0644)
        if err != nil {
                log.Printf("failed opening %s: %s", filename, err)
                return
        }
        defer file.Close()
        err = encoder(file, img)
        if err != nil {
                log.Printf("failed encoding to %s: %s", filename, err)
        }
}

func jpegEncode(w io.Writer, m image.Image) error {
        return jpeg.Encode(w, m, &jpeg.Options{Quality: 80})
}

func decode(filename string) image.Image {
        file, err := os.Open(filename)
        if err != nil {
                log.Fatalf("failed opening file: %s", err)
        }
        defer file.Close()

        img, _, err := image.Decode(file)
        if err != nil {
                log.Fatalf("failed decoding image: %s", err)
        }
        return img
}

func applyBorder(img image.Image, c *Color, w int) image.Image {
        // Make a new solid color image, slightly larger to form the border
        r := image.Rect(0, 0, img.Bounds().Dx()+(2*w), img.Bounds().Dy()+(2*w))
        dst := image.NewRGBA(r)
        draw.Draw(dst, r, image.NewUniform(c.ToRGBA()), image.ZP, draw.Src)

        // Draw the source image over the border image
        draw.Draw(dst, r, img, image.Pt(-w, -w), draw.Src)
        return dst
}
```

```
func main() {
	flag.Parse()

	img := decode(*in)
	img = applyBorder(img, borderColor, *width)

	if *pngout != "" {
		encode(png.Encode, img, *pngout)
	}

	if *jpgout != "" {
		encode(jpegEncode, img, *jpgout)
	}
}
```

gostagram

Instagram isn't a big deal right? I mean, you could totally build that in a weekend.[64]

Okay, well maybe not quite, but let's see what Go can do, and build some image filters. It'll just be a command line application, but you could imagine how you could use this and the rest of the code from this chapter to build a little web application Instagram clone.

All I have are:

- Black and white
- Sepia
- Blur (It's pretty slow for large radius values, and it doesn't handle edge cases properly)
- Borders (like in the previous example)

The black and white and sepia code is fine, though I wouldn't use my blur code in production.

[64] Or, if you're a Jeff Atwood fan, 6-8 weeks.

image/gostagram.go

```
package main

import (
	"flag"
	"fmt"
	"image"
	"image/color"
	"image/draw"
	"image/jpeg"
	"image/png"
	"io"
	"log"
	"math"
	"os"
)

var (
	jpgout      = flag.String("jpg", "", "output to a jpg")
	pngout      = flag.String("png", "", "output to a png")
	in          = flag.String("in", "", "input file")
	filter      = flag.String("filter", "", "filter to apply")
	borderWidth = flag.Int("border", 0, "border width, used in conjunction with -color")
	borderColor = new(Color)
	blur        = flag.Int("blur", 0, "blur the image with a Gaussian blur (slow!)")
)

func init() {
	flag.Var(borderColor, "color", "the color of the border in RGBA")
}

type Gaussian struct {
	kernel  []float32
	offsets []int
}

func (gaus *Gaussian) Blur(img image.Image, x, y int) color.Color {
	colors := make([]color.Color, 0, len(gaus.kernel))
	for _, yOffset := range gaus.offsets {
		for _, xOffset := range gaus.offsets {
			colors = append(colors, img.At(x+xOffset, y+yOffset))
		}
```

```
	}
	var rsum, gsum, bsum, asum float32
	for i, c := range colors {
		rgba := color.RGBAModel.Convert(c).(color.RGBA)
		factor := gaus.kernel[i]
		rsum += factor * float32(rgba.R)
		gsum += factor * float32(rgba.G)
		bsum += factor * float32(rgba.B)
		asum += factor * float32(rgba.A)
	}
	return color.RGBA{
		R: min(255, rsum),
		G: min(255, gsum),
		B: min(255, bsum),
		A: min(255, asum),
	}
}

func normalize(kernel []float32) {
	var sum float32
	for _, f := range kernel {
		sum += f
	}
	for i := range kernel {
		kernel[i] = kernel[i] / sum
	}
}

func spread(radius int) []int {
	s := make([]int, 0, 2*radius+1)
	low, high := -radius, radius
	for i := low; i <= high; i++ {
		s = append(s, i)
	}
	return s
}

func NewGaussian(radius int) *Gaussian {
	sigmaSquared := math.Pow(float64(radius)/2, 2)
	bottom := 2 * sigmaSquared
	G := func(x, y int) float32 {
		top := -(math.Pow(float64(x), 2) + math.Pow(float64(y), 2))
		exp := math.Exp(top / bottom)
```

```
			g := 1 / (2 * math.Pi * sigmaSquared) * exp
			return float32(g)
		}

		d := radius*2 + 1
		kernel := make([]float32, 0, d*d)
		rng := spread(radius)
		for _, y := range rng {
			for _, x := range rng {
				kernel = append(kernel, G(x, y))
			}
		}
		normalize(kernel)
		return &Gaussian{kernel, rng}
}

type Color struct {
		RGBA uint32
}

func (c *Color) String() string {
		return fmt.Sprintf("%#x", c.RGBA)
}

func (c *Color) Set(s string) error {
		_, err := fmt.Sscanf(s, "%x", &c.RGBA)
		return err
}

func (c *Color) ToRGBA() color.RGBA {
		var mask uint32 = 0xff
		return color.RGBA{
			R: uint8((c.RGBA >> 24) & mask),
			G: uint8((c.RGBA >> 16) & mask),
			B: uint8((c.RGBA >> 8) & mask),
			A: uint8(c.RGBA & mask),
		}
}

type Sepia struct {
		R, G, B float32
		A       uint8
}
```

```
func min(l, r float32) uint8 {
	if r > l {
		return uint8(l)
	}
	return uint8(r)
}

func (s *Sepia) RGBA() color.RGBA {
	r := min(255, s.R*0.393+s.G*0.769+s.B*0.189)
	g := min(255, s.R*0.349+s.G*0.686+s.B*0.168)
	b := min(255, s.R*0.272+s.G*0.534+s.B*0.131)
	return color.RGBA{r, g, b, s.A}
}

func NewSepia(c color.Color) *Sepia {
	rgba := color.RGBAModel.Convert(c).(color.RGBA)
	return &Sepia{float32(rgba.R), float32(rgba.G), float32(rgba.B), rgba.A}
}

type encf func(io.Writer, image.Image) error

func encode(encoder encf, img image.Image, filename string) {
	file, err := os.OpenFile(filename, os.O_WRONLY|os.O_CREATE, 0644)
	if err != nil {
		log.Printf("failed opening %s: %s", filename, err)
		return
	}
	defer file.Close()
	err = encoder(file, img)
	if err != nil {
		log.Printf("failed encoding to %s: %s", filename, err)
	}
}

func jpegEncode(w io.Writer, m image.Image) error {
	return jpeg.Encode(w, m, &jpeg.Options{Quality: 80})
}

func decode(filename string) image.Image {
	file, err := os.Open(filename)
	if err != nil {
		log.Fatalf("failed opening file: %s", err)
```

```
	}
	defer file.Close()

	img, _, err := image.Decode(file)
	if err != nil {
		log.Fatalf("failed decoding image: %s", err)
	}
	return img
}

func doBlackAndWhite(img image.Image) image.Image {
	r := img.Bounds()
	dst := image.NewGray(r)
	draw.Draw(dst, r, img, image.ZP, draw.Src)
	return dst
}

func doSepia(img image.Image) image.Image {
	r := img.Bounds()
	dst := image.NewRGBA(r)
	w, h := r.Dx(), r.Dy()
	for y := 0; y < h; y++ {
		for x := 0; x < w; x++ {
			sepia := NewSepia(img.At(x, y)).RGBA()
			dst.Set(x, y, sepia)
		}
	}
	return dst
}

func doBorder(img image.Image, c *Color, w int) image.Image {
	r := image.Rect(0, 0, img.Bounds().Dx()+(2*w), img.Bounds().Dy()+(2*w))
	dst := image.NewRGBA(r)
	draw.Draw(dst, r, image.NewUniform(c.ToRGBA()), image.ZP, draw.Src)
	draw.Draw(dst, r, img, image.Pt(-w, -w), draw.Src)
	return dst
}

func doBlur(img image.Image, radius int) image.Image {
	g := NewGaussian(radius)
	r := img.Bounds()
	dst := image.NewRGBA(r)
	w, h := r.Dx(), r.Dy()
```

```
	for y := 0; y < h; y++ {
		for x := 0; x < w; x++ {
			dst.Set(x, y, g.Blur(img, x, y))
		}
	}
	return dst
}

func main() {
	flag.Parse()

	img := decode(*in)

	switch *filter {
	case "bw":
		img = doBlackAndWhite(img)
	case "sepia":
		img = doSepia(img)
	}

	if *blur > 0 {
		img = doBlur(img, *blur)
	}

	if *borderWidth > 0 {
		img = doBorder(img, borderColor, *borderWidth)
	}

	if *pngout != "" {
		encode(png.Encode, img, *pngout)
	}

	if *jpgout != "" {
		encode(jpegEncode, img, *jpgout)
	}
}
```

index

The `index` package has nothing in it, except the `index/suffixarray` package.

This package lets you search for a sequence of bytes in a byte slice in logarithmic time. You have to build the index index first, so there's not only the time to build in the index, but also the memory cost associated with storing the index. It's not the answer to all substring search problems, but if your situation warrants it, you can get some nice speed improvements (at the cost of more memory).

suffixarray

We'll look for a known popular text in a large piece of text, and compare that to less fancy methods, like those in the `bytes` package.

index/suffixarray.go

```
package main

import (
	"bytes"
	"flag"
	"index/suffixarray"
	"io/ioutil"
	"log"
	T "testing"
)

type Searcher struct {
	index *suffixarray.Index
	n, h  []byte
}

func NewSearcher(n, h []byte) *Searcher {
	return &Searcher{
		n:     n,
		h:     h,
		index: suffixarray.New(h),
```

```go
	}
}

func (s *Searcher) SuffixarrayPrebuilt() int {
	results := s.index.Lookup(s.n, 1)
	if len(results) == 1 {
		return results[0]
	}
	return -1
}

func (s *Searcher) Suffixarray() int {
	index := suffixarray.New(s.h)
	results := index.Lookup(s.n, 1)
	if len(results) == 1 {
		return results[0]
	}
	return -1
}

func (s *Searcher) BytesIndex() int {
	return bytes.Index(s.h, s.n)
}

var (
	needle   = flag.String("needle", "O Romeo, Romeo! wherefore art thou Romeo?", "The \
string to search for")
	haystack = flag.String("haystack", "romeo-and-juliet.txt", "The file to search thro\
ugh")
)

func bench(name string, f func() int) {
	index := 0
	result := T.Benchmark(func(b *T.B) {
		for i := 0; i < b.N; i++ {
			index = f()
		}
	})
	log.Printf("%s took %d ns/op to find %#v at index %d", name, result.NsPerOp(), *nee\
dle, index)
}

func main() {
```

```
	flag.Parse()
	h, err := ioutil.ReadFile(*haystack)
	if err != nil {
		log.Fatalf("failed to read haystack: %s", err)
	}
	s := NewSearcher([]byte(*needle), h)
	bench("SuffixarrayPrebuilt", s.SuffixarrayPrebuilt)
	bench("Suffixarray", s.Suffixarray)
	bench("BytesIndex", s.BytesIndex)
}
```

Running the example, we get 3 lines from the `log` package. On my machine, using the prebuilt `suffixarray` index I get about **800 ns/op**. Building the index and doing the search (which is the basic usage example of the package) in the `Suffixarray` method, takes a very long time, on the order of **50 million ns/op**. Most of that time is building the index. The `bytes` package doing a simple `bytes.Index` takes about an order of magnitude longer than using the prebuilt `suffixarray`, coming in at around **7000 ns/op**.

As you can see, if you're doing a one-off search, the `bytes` package is fine. If you're searching the same body of text many times, and can afford to keep the index around in memory, the `suffixarray` package is your friend.

io

The `io` package is probably one of the most important packages in the Go standard library, but it's also one of the most basic.

There are only 4 types which are structs, 2 of which are related (`PipeReader` and `PipeWriter`). The other types are interfaces, and there's a lot of them. You'll see them all over the place, and it's usually just a matter of providing a type that matches that interface.

The top level functions in the package take care of abstracting a few things away from the lower level base interfaces. They also handle some basic common things, along with the `io/ioutil` package.

> If you're reading this section thinking this is where file IO happens, you're half right. While the `io` package has all the interfaces and some helpers, the real file IO happens in the `os` package package. That package allows you to open files and read and write them using the `os.File` type.

Reading

Computers are no fun if your programs can't talk to things outside themselves. Reading data in is half of the IO fun, and allows you program to get data from the outside world. The `io.Reader` type has a basic `Read` method to handle the most basic of read tasks. The `io` package has some helpers to move up a level of abstraction.

There are also a variety of interface types that provide other higher level methods, such as unreading data and reading runes.

With some of these IO things it's important to pay attention to the errors returned. With things like `ReadAtLeast`, it returns specific errors to signal specific cases when the minimum couldn't be read due to EOF.

`ReadAtLeast`

This function will read, big surprise, at least `min` bytes into the buffer, returning the standard bytes read and any error.

io/read_at_least.go

```
package main

import (
	"io"
	"log"
	"strings"
)

const (
	format = "len(buffer)=%d, min=%d, bytesRead=%d, err=%v, (%s)"
)

func init() {
	log.SetFlags(0)
	log.SetPrefix("» ")
}

type Example struct {
	BufferLength int
	MinimumRead  int
	Message      string
}

func ShowExample(ex Example) {
	rd := strings.NewReader(ex.Message)
	buffer := make([]byte, ex.BufferLength)
	bytesRead, err := io.ReadAtLeast(rd, buffer, ex.MinimumRead)
	log.Printf(format, ex.BufferLength, ex.MinimumRead, bytesRead, err, ex.Message)
}

func main() {
	examples := []Example{
		{10, 5, "OK; read less than buf can handle, plenty of data"},
		{100, 75, "Unexpected EOF; buf has space, but ran out of data"},
		{10, 15, "Short buffer; trying to read more than buf can handle"},
	}
	for _, ex := range examples {
		ShowExample(ex)
	}
}
```

Output:

```
» len(buffer)=10, min=5, bytesRead=10, err=<nil>, (OK; read less than buf can handle\
, plenty of data)
» len(buffer)=100, min=75, bytesRead=50, err=unexpected EOF, (Unexpected EOF; buf ha\
s space, but ran out of data)
» len(buffer)=10, min=15, bytesRead=0, err=short buffer, (Short buffer; trying to re\
ad more than buf can handle)
```

ReadFull

Use `ReadFull` if you want to read an exact number of bytes from something. It will return an error if it couldn't read the given number of bytes. In the output I'm showing the buffer as bytes to show that there are `NULL` bytes (the zeroes) in the buffer when you get EOF errors.

io/read_full.go

```go
package main

import (
	"io"
	"log"
	"strings"
)

const (
	format = "len(buffer)=%d, bytesRead=%d, err=%v, (%s)"
)

func init() {
	log.SetFlags(0)
	log.SetPrefix("» ")
}

type Example struct {
	BufferLength int
	Message      string
}

func ShowExample(ex Example) {
	rd := strings.NewReader(ex.Message)
```

```
        buffer := make([]byte, ex.BufferLength)
        bytesRead, err := io.ReadFull(rd, buffer)
        log.Printf("%v", buffer)
        log.Printf(format, ex.BufferLength, bytesRead, err, ex.Message)
}

func main() {
        examples := []Example{
                {10, "OK; filled up buf, plenty of data"},
                {55, "Unexpected EOF; buf has space, but ran out of data"},
        }
        for _, ex := range examples {
                ShowExample(ex)
        }
}
```

Output:

```
» [79 75 59 32 102 105 108 108 101 100]
» len(buffer)=10, bytesRead=10, err=<nil>, (OK; filled up buf, plenty of data)
» [85 110 101 120 112 101 99 116 101 100 32 69 79 70 59 32 98 117 102 32 104 97 115 \
32 115 112 97 99 101 44 32 98 117 116 32 114 97 110 32 111 117 116 32 111 102 32 100\
 97 116 97 0 0 0 0 0]
» len(buffer)=55, bytesRead=50, err=unexpected EOF, (Unexpected EOF; buf has space, \
but ran out of data)
```

`LimitedReader`

A `LimitedReader` is for when you want to make sure to never read more than a given amount from an `io.Reader`. If you've worked with `Enumerable` style methods in something like Ruby, this is basically implementing `take` on `io.Reader`.

io/limited_reader.go

```go
package main

import (
	"io"
	"log"
	"strings"
)

const (
	example = "The quick brown fox, he likes jumping, you know."
)

func init() {
	log.SetFlags(0)
	log.SetPrefix("» ")
}

func main() {
	lr := io.LimitedReader{strings.NewReader(example), 20}
	buffer := make([]byte, len(example))
	bytesRead, err := lr.Read(buffer)

	// Despite having space, only read 20 bytes
	log.Printf("%s", buffer)
	log.Printf("bytesRead=%d, err=%v", bytesRead, err)

	// Try reading more, won't read anything.
	bytesRead, err = lr.Read(buffer)
	log.Printf("bytesRead=%d, err=%v", bytesRead, err)
}
```

Output:

```
» The quick brown fox,\\\\\\\\\\\\\\\\\\\\\\\\\\\\
» bytesRead=20, err=<nil>
» bytesRead=0, err=EOF
```

`MultiReader`

A `MultiReader` lets you read from multiple readers, one after the other. If you open 3 files, make a `MultiReader` from them, and read until EOF, it would be the same as if you'd concatenated the files into a new file, and read that file instead. It just reads everything from everything in order.

An interesting point is that once a reader returns EOF, it will stop reading. You have to start it up again to read more. In our example we have to read 3 times to actually read all the data. We could also `ReadFull` which we used a few pages ago.

Once again, we'll look at the buffer as a byte slice to show the `NULL` values.

io/multi_reader.go

```go
package main

import (
	"io"
	"log"
	"strings"
)

func init() {
	log.SetFlags(0)
	log.SetPrefix("» ")
}

func main() {
	// Make our inputs
	a := strings.NewReader(strings.Repeat("A", 5))
	b := strings.NewReader(strings.Repeat("B", 5))
	c := strings.NewReader(strings.Repeat("C", 5))

	// Read ALL THE THINGS
	mr := io.MultiReader(a, b, c)

	// Read A
	buffer := make([]byte, 20)
	n1, err := mr.Read(buffer)
	log.Printf("%v", buffer)
	log.Printf("n1=%d, err=%v", n1, err)

	// Read B
	n2, err := mr.Read(buffer[n1:])
	log.Printf("%v", buffer)
	log.Printf("n2=%d, err=%v", n2, err)

	// Read C
```

```
        n3, err := mr.Read(buffer[(n1 + n2):])
        log.Printf("%v", buffer)
        log.Printf("n3=%d, err=%v", n3, err)

        // EOF
        n4, err := mr.Read(buffer[(n1 + n2 + n3):])
        log.Printf("%v", buffer)
        log.Printf("n4=%d, err=%v", n4, err)
}
```

Output:

```
» [65 65 65 65 65 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
» n1=5, err=<nil>
» [65 65 65 65 65 66 66 66 66 66 0 0 0 0 0 0 0 0 0 0]
» n2=5, err=<nil>
» [65 65 65 65 65 66 66 66 66 66 67 67 67 67 67 0 0 0 0 0]
» n3=5, err=<nil>
» [65 65 65 65 65 66 66 66 66 66 67 67 67 67 67 0 0 0 0 0]
» n4=0, err=EOF
```

TeeReader

A `TeeReader` works a lot like the `tee` unix program. With the `tee` program, you read some output, display it on STDOUT, and then also write it somewhere else. In the case of `TeeReader`, you read from it, but it also writes to an `io.Writer`. Pretty straightforward.

io/tee_reader.go

```
package main

import (
        "bytes"
        "io"
        "log"
        "strings"
)

func init() {
        log.SetFlags(0)
        log.SetPrefix("» ")
```

```
}

func main() {
	s := strings.NewReader("Get to the choppa!")
	var buf bytes.Buffer
	tr := io.TeeReader(s, &buf)
	b := make([]byte, s.Len())
	n, err := tr.Read(b)
	log.Printf("buf: %s", &buf)
	log.Printf("  b: %s", b)
	log.Printf("n=%d, err=%v", n, err)
}
```

Output:

```
» buf: Get to the choppa!
»   b: Get to the choppa!
» n=18, err=<nil>
```

SectionReader

The last thing we'll demo here is the `SectionReader`. It's sort of like `LimitedReader` but for specific sections of something. You need an `io.ReaderAt`, so you can't pass just anything into it. We're using a `bytes.Reader`, but `os.File` works as well.

io/section_reader.go

```
package main

import (
	"bytes"
	"io"
	"log"
)

var (
	s = "The quick brown fox, he likes jumping, you know."
)

func init() {
	log.SetFlags(0)
	log.SetPrefix("» ")
```

```go
}

func main() {
        // Build the block of data
        data := make([]byte, 0, 30)
        data = append(data, bytes.Repeat([]byte{'A'}, 10)...)
        data = append(data, bytes.Repeat([]byte{'B'}, 10)...)
        data = append(data, bytes.Repeat([]byte{'C'}, 10)...)

        // Create some SectionReaders to read the 3 sections
        r := bytes.NewReader(data)
        ar := io.NewSectionReader(r, 0, 10)
        br := io.NewSectionReader(r, 10, 10)
        cr := io.NewSectionReader(r, 20, 10)

        buf := make([]byte, 10)

        // Read the A section
        n, err := ar.Read(buf)
        log.Printf("buf: %s", buf)
        log.Printf("n=%d, err=%v", n, err)

        // Read the B section
        n, err = br.Read(buf)
        log.Printf("buf: %s", buf)
        log.Printf("n=%d, err=%v", n, err)

        // Read the C section
        n, err = cr.Read(buf)
        log.Printf("buf: %s", buf)
        log.Printf("n=%d, err=%v", n, err)
}
```

Output:

```
» buf: AAAAAAAAAA
» n=10, err=<nil>
» buf: BBBBBBBBBB
» n=10, err=<nil>
» buf: CCCCCCCCCC
» n=10, err=<nil>
```

Writing

Writing is the other half of the IO coin. Sending data outside your program, or within it, is essential to getting things done. Your basic `io.Writer` interface supports writing bytes, but we can also write strings, and write to multiple things at the same time.

io/writing.go

```go
package main

import (
	"io"
	"log"
	"os"
)

func init() {
	log.SetFlags(0)
	log.SetPrefix("» ")
}

func main() {
	w := io.MultiWriter(os.Stdout, os.Stderr)
	io.WriteString(w, "Hello, twice!!\n")
}
```

Output:

```
Hello, twice!!
Hello, twice!!
```

Copy

Copying data from one place to another is a pretty basic IO related task. Want to build a TCP load balancer? All you need is `io.Copy`[65].

The two copy methods block until they reach an error. If that error is EOF, it's silenced, and a `nil` error is returned. If you just want to copy and not worry about the errors, just run the call in a goroutine.

io/copy.go

```go
package main

import (
	"bytes"
	"io"
	"log"
	"os"
)

func init() {
	log.SetFlags(0)
	log.SetPrefix("» ")
}

func buffer() *bytes.Buffer {
	var buf bytes.Buffer
	buf.WriteString("I'm writing ")
	buf.WriteString("strings ")
	buf.WriteString("to this buffer ")
	buf.WriteString("and we'll copy it to os.Stdout.\n")
	return &buf
}

func DemoCopy() {
```

[65] https://github.com/darkhelmet/balance/blob/master/tcp.go

```
	buf := buffer()
	log.Printf("copying %d bytes to os.Stdout", buf.Len())
	io.Copy(os.Stdout, buf)
}

func DemoCopyN() {
	buf := buffer()
	n := int64(32)
	log.Printf("have %d bytes, only copying %d to os.Stdout", buf.Len(), n)
	io.CopyN(os.Stdout, buf, n)
	os.Stdout.Write([]byte{'\n'})
}

func BufferFun() {
	buf := buffer()
	n, _ := io.CopyN(os.Stdout, buf, 32)
	nn, _ := io.Copy(os.Stdout, buf)
	log.Printf("copied %d and then %d bytes to os.Stdout", n, nn)
}

func main() {
	DemoCopy()
	DemoCopyN()
	BufferFun()
}
```

Output:

```
» copying 67 bytes to os.Stdout
I'm writing strings to this buffer and we'll copy it to os.Stdout.
» have 67 bytes, only copying 32 to os.Stdout
I'm writing strings to this buff
I'm writing strings to this buffer and we'll copy it to os.Stdout.
» copied 32 and then 35 bytes to os.Stdout
```

Pipe

If you want to pipe data between two things, you have a couple options. You can use `bytes.Buffer`, which will buffer data, or you can use `io.Pipe`, which does no buffering and instead does synchronous piping of data.

In this example, note the partial reads, where it reads the remainder of a sentence. This is because the write on the other end isn't finished yet. It writes part of the sentence straight through, and then has to wait while the reading end writes out the log, and the loop starts over to read another chunk. Then the write call can continue and write the rest of the sentence, while the reading end reads it. The writing loop starts over and tries to write another sentence.

io/pipe.go

```
package main

import (
	"io"
	"log"
	"runtime"
)

func init() {
	log.SetFlags(0)
	log.SetPrefix("» ")
	runtime.GOMAXPROCS(8)
}

func Write(wr io.WriteCloser) {
	lyrics := []string{
		"I come home in the morning light",
		"My mother says when you gonna live your life right",
		"Oh mother dear we're not the fortunate ones",
		"And girls they want to have fun",
		"Oh girls just want to have fun",
	}

	for _, line := range lyrics {
		io.WriteString(wr, line)
	}
	wr.Close() // We're done, signal EOF
}

func main() {
	rd, wr := io.Pipe()
	go Write(wr)
	for {
		buf := make([]byte, 32)
		n, err := rd.Read(buf)
```

```
            log.Printf("buf: %s", buf)
            log.Printf("n=%d, err=%v", n, err)
            if err == io.EOF {
                break
            }
        }
}
```

Output:

```
» buf: I come home in the morning light
» n=32, err=<nil>
» buf: My mother says when you gonna li
» n=32, err=<nil>
» buf: ve your life right\\\\\\\\\\\\\\
» n=18, err=<nil>
» buf: Oh mother dear we're not the for
» n=32, err=<nil>
» buf: tunate ones\\\\\\\\\\\\\\\\\\\\\
» n=11, err=<nil>
» buf: And girls they want to have fun\
» n=31, err=<nil>
» buf: Oh girls just want to have fun\\
» n=30, err=<nil>
» buf: \\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
» n=0, err=EOF
```

`io/ioutil`

The `io/ioutil` package includes a selection of top level functions to assist in common IO tasks. They are all very simple and self explanatory, so we'll demo a few of the functions but not all.

io/ioutil.go

```go
package main

import (
	"io/ioutil"
	"log"
	"os"
)

func init() {
	log.SetFlags(0)
	log.SetPrefix("» ")
}

func DemoReadAll() {
	file, err := os.Open("ioutil.go")
	if err != nil {
		log.Panicf("failed opening file: %s", err)
	}
	defer file.Close()
	log.Println(`reading file "ioutil.go"`)
	data, err := ioutil.ReadAll(file)
	log.Printf("read %d bytes with err %v", len(data), err)
}

func DemoReadDir() {
	entries, err := ioutil.ReadDir(".")
	if err != nil {
		log.Panicf("failed reading directory: %s", err)
	}
	log.Printf("found %d files in the current directory", len(entries))
}

func DemoReadFile() {
	data, err := ioutil.ReadFile("ioutil.go")
	log.Printf("read %d bytes with err %v", len(data), err)
}

func main() {
	DemoReadAll()
	DemoReadDir()
	DemoReadFile()
```

```
}
```

Output:

```
» reading file "ioutil.go"
» read 772 bytes with err <nil>
» found 11 files in the current directory
» read 772 bytes with err <nil>
```

log

The log package is used to handle logging in your application. It has a basic logging package, and a syslog package.

Basic Logging

There are two ways to use the basic logging functionality.

1. Use the package level functions to access the global logger.
2. Build a new *log.Logger instance and use that.

Using the package level functions is easy and useful when you're dealing with a proper application. If you build a package to be used in an application, you probably shouldn't be using the package level functions and instead require the user of your package to pass in a *log.Logger.

In pretty much every example so far, I've used the package level functions to handle output. There are functions to print a string, print a formatted string, panic, and exit. You can also change where the output goes, change the format of messages (using the the flags), and change the prefix.

In the example, we won't use the package level functions, and just build our own instance.

log/log.go

```
package main

import (
	"log"
	"os"
	"time"
)

func main() {
	logger := log.New(os.Stdout, "", log.LstdFlags)

```

```
	defer func() {
		logger.SetFlags(log.LstdFlags)
		if err := recover(); err != nil {
			logger.Fatalf("recovered: %s", err)
		}
	}()

	logger.Println("just a string")
	logger.SetPrefix("[go-thestdlib] ")
	logger.Printf("the time is %s", time.Now())
	logger.SetFlags(log.Lshortfile)
	logger.Println("see, the format changed?")
	logger.Panicf("don't worry, we'll handle this")
}
```

Syslog

There are a couple ways to use the `log/syslog` package.

You can build a `*log.Logger` at a certain syslog priority. This has the advantage of using the same interface as the main package, but all your log messages are at the same priority level.

The other way is to create a `*syslog.Writer`. This has the advantage of being able to write log messages at different priority levels, but you lose the consistent interface. None of the `*syslog.Writer` methods support formatting things either, so you'll have to do that elsewhere. We'll look at both examples, and build a little struct to give you both the nice easy interface as well as logging multiple priorities.

> When running this example, make sure to `tail -f /var/log/system.log` if you're on a Mac or `tail -f /var/log/syslog` if you're on Linux in order to see the log messages.

log/syslog.go

```go
package main

import (
	"log"
	"log/syslog"
)

func MustSyslog(p syslog.Priority, flags int) *log.Logger {
	logger, err := syslog.NewLogger(p, flags)
	if err != nil {
		panic(err)
	}
	return logger
}

type Logger struct {
	Alert, Crit, Debug, Emerg, Err, Info, Notice, Warning *log.Logger
}

func NewLogger(flags int) *Logger {
	return &Logger{
		Alert:   MustSyslog(syslog.LOG_ALERT, flags),
		Crit:    MustSyslog(syslog.LOG_CRIT, flags),
		Debug:   MustSyslog(syslog.LOG_DEBUG, flags),
		Emerg:   MustSyslog(syslog.LOG_EMERG, flags),
		Err:     MustSyslog(syslog.LOG_ERR, flags),
		Info:    MustSyslog(syslog.LOG_INFO, flags),
		Notice:  MustSyslog(syslog.LOG_NOTICE, flags),
		Warning: MustSyslog(syslog.LOG_WARNING, flags),
	}
}

func basic() {
	logger, err := syslog.NewLogger(syslog.LOG_WARNING, log.Lshortfile)
	if err != nil {
		log.Fatalf("failed to make syslogger: %s", err)
	}
	logger.Println("hello, world")
}

func levels() {
```

```
        logger := NewLogger(log.Lshortfile)
        logger.Crit.Println("oh noes!")
        logger.Warning.Println("just a warning")
        logger.Alert.Println("alert time!")
}

func writer() {
        logger, err := syslog.New(syslog.LOG_WARNING, "go-thestdlib")
        if err != nil {
                log.Fatalf("failed to make a syslogger: %s", err)
        }
        logger.Warning("just a message...")
}

func main() {
        basic()
        levels()
        writer()
}
```

math

The `math` package, does all the math you could possibly want. The basics anyway. For the top level package, there is no value in an example file, because it's all very straightforward. You need `sin`? Call `math.Sin`. The functions are appropriately named so it should be easy to find what you're looking for. Typically everything deals with `float64` values.

The fun starts with the subpackages. There is the `math/big` package for dealing with big numbers, both rational and integers.

`math/cmplx` has functions similar to those in the toplevel `math` package but for complex numbers. I'll skip the example for that package because, like the `math` package, it's not very interesting. If you need to perform math on complex numbers, that's where you need to look.

If you need pseudo-random numbers, look at the `math/rand` package. It has functions to get all the different types of random data. You can also create a `rand.Rand` with a specific seed to get a reproducible sequence.

Big Numbers

The `math/big` package has two types: `big.Int` and `big.Rat`. `big.Int` deals with signed integers, and `big.Rat` deals with rational numbers. The APIs follow the same pattern where results get stored in the receiver and also returned.[66]

math/big.go

```
package main

import (
	"log"
	"math/big"
)

func bigPrime() {
	p := big.NewInt(2)
	p.Exp(p, big.NewInt(1398269), nil)
```

[66] Honestly, the API is slightly awkward, but I think it's in the interest in saving memory.

```
        p.Sub(p, big.NewInt(1))
        // Get ready to scroll
        log.Printf("a big prime number is %s", p)
        // Takes a while
        // log.Printf("2^1,398,269-1 is probably prime: %t", p.ProbablyPrime(1))
}

func ScientificNotation(coefficient, exponent int64) *big.Int {
        exp := big.NewInt(10)
        exp = exp.Exp(exp, big.NewInt(exponent), nil)
        coeff := big.NewInt(coefficient)
        return coeff.Mul(coeff, exp)
}

func astrophysics() {
        age := ScientificNotation(43, 16)
        log.Printf("the universe is about %s seconds old", age)
        size := ScientificNotation(88, 25)
        log.Printf("the universe is about %s light years across", size)
        stars := ScientificNotation(5, 22)
        log.Printf("the universe has about %s stars", stars)
        galaxies := ScientificNotation(125, 9)
        log.Printf("the universe has about %s galaxies", galaxies)
}

func primeList() {
        var primesFound int
        two := big.NewInt(2)
        p := big.NewInt(3)
        for {
                if p.ProbablyPrime(1) {
                        primesFound++
                        log.Printf("%s is a prime number", p)
                }

                if primesFound > 100 {
                        break
                }

                p.Add(p, two)
        }
}
```

```
func mul() {
	x, _ := new(big.Int).SetString("7612058254738945", 10)
	y, _ := new(big.Int).SetString("9263591128439081", 10)
	z := new(big.Int).Mul(x, y)
	log.Printf("%s x %s = %s", x, y, z)
}

func gcd() {
	a, _ := new(big.Int).SetString("7612058254738945", 10)
	b, _ := new(big.Int).SetString("9263591128439081", 10)
	z := new(big.Int).GCD(nil, nil, a, b)
	log.Printf("the GCD of %s and %s is %s", a, b, z)
}

var one = big.NewRat(1, 1)

func f(i *big.Rat, depth uint64) *big.Rat {
	if depth == 0 {
		return one
	}

	// Doing this is slightly faster
	// than the recursive version.
	c := make(chan *big.Rat, 1)
	go func() {
		n := new(big.Rat).Set(i)
		c <- f(n.Add(n, one), depth-1)
	}()

	num := new(big.Rat).Set(i)
	denom := big.NewRat(2, 1)
	denom = denom.Mul(denom, num)
	denom = denom.Add(denom, one)

	rest := new(big.Rat)
	rest = rest.Mul(num, denom.Inv(denom))
	rest = rest.Mul(rest, <-c)

	ret := big.NewRat(1, 1)
	return ret.Add(ret, rest)
}

func pi() {
```

```go
	value := f(big.NewRat(1, 1), 500)
	value.Mul(value, big.NewRat(2, 1))
	log.Printf("pi is %s", value.FloatString(100))
}

func main() {
	bigPrime()
	astrophysics()
	primeList()
	mul()
	gcd()
	pi()
}
```

There are a lot more operations available than in the example, but if you need to deal with big numbers, you get the idea.[67]

Random Numbers

The `math/rand` package generates pseudo-random numbers for you, but in a semi-predictable fashion. If you build a `math.Source` with a certain seed value, the `math.Rand` that you build from it will produce the same sequence of numbers every time. For more secure randomness, use `cryto/rand`. If you need predictable random numbers, use `math/rand`.

The `math/rand` package has top level functions that work with a "global" `math.Rand` instance. You can also build your own, and work with that. In the example we'll do that.

math/rand.go

```go
package main

import (
	"log"
	"math/rand"
)

func example(seed int64) {
	s := rand.NewSource(seed)
```

[67] That being said, if you have some nice ideas for other examples, please let me know.

```
	r := rand.New(s)
	log.Printf("ExpFloat64: %f", r.ExpFloat64())
	log.Printf("Float32: %f", r.Float32())
	log.Printf("Float64: %f", r.Float64())
	log.Printf("Int: %d", r.Int())
	log.Printf("Int31: %d", r.Int31())
	log.Printf("Int31n: %d", r.Int31n(10))
	log.Printf("Int63: %d", r.Int63())
	log.Printf("Int63n: %d", r.Int63n(15))
	log.Printf("Intn: %d", r.Intn(25))
	log.Printf("NormFloat64: %f", r.NormFloat64())
	log.Printf("Perm: %v", r.Perm(10))
	log.Printf("Uint32: %d", r.Uint32())
}

func main() {
	example(64)
	// Will print the same as above
	example(64)
	example(1)
}
```

mime

The toplevel `mime` package isn't that exciting.

Use `TypeByExtension` to turn a file extension, like `.html` or `.pdf` into `text/html; charset=utf-8` and `application/pdf`. Use `AddExtensionType` to add your own mapping.

With the functions `ParseMediaType` and `FormatMediaType` you can, as their names suggest, parse and format mime types. `ParseMediaType` can turn `text/html; charset=utf-8` into the string `text/html` and the map `map[string]string{"charset": "utf-8"}`. `FormatMediaType` does the inverse.

Exciting, amiright?

Let's look at the `mime/multipart` instead, shall we?

Multipart Parsing

The `mime/multipart` package is used for, you guessed it, parsing and generating multipart things.

I sent a test email and pulled out the body. Parsing the whole email can be done with the `net/mail` package, but we're only concerned with the multipart body.

> I use a hardcoded boundary value. If you parse an entire message, the `net/mail` package's `Message` type gives you the headers, which would included the content type indicating that it's multipart, and would also include the boundary value. This is where `ParseMediaType` would come in, and you can parse the content type to get the boundary value.

Each part of the body is separated by the boundary, and the `multipart.Reader` takes care of giving us each part. The first part is a `multipart/alternate` block, which has its own set of headers with a new boundary. This provides the body of the email in both `text/plain` and `text/html` content types. In the second part is a file attachment, Base64 encoded.

In this case, you'd parse the email, get the headers and body, see that it's a multipart message, parse the body as multipart, then see that one of the parse is again multipart, parse that, etc, etc. You end up with a bit of recursion, but at the same time, I can't see why you'd have to go much further than the two levels. That being said, I'm not aware of anything saying you couldn't "infinitely" nest multipart things, but I haven't read the whole spec.

mime/parse.go

```
package main

import (
	"bytes"
	"encoding/base64"
	"io"
	"io/ioutil"
	"log"
	"mime"
	"mime/multipart"
	"os"
)

type Part struct {
	*multipart.Part
	Body []byte
}

func (p *Part) Reader() io.Reader {
	return bytes.NewReader(p.Body)
}

func ReadMultipartFile(path, boundary string) (parts []*Part) {
	file, err := os.Open(path)
	if err != nil {
		log.Fatalf("failed opening %s: %s", path, err)
	}
	defer file.Close()
	return ReadMultipart(file, boundary)
}

func ReadMultipart(r io.Reader, boundary string) (parts []*Part) {
	mr := multipart.NewReader(r, boundary)
	for {
		part, err := mr.NextPart()
		if err != nil {
			if err == io.EOF {
```

```
				break
			}
			log.Fatalf("failed reading part: %s", err)
		}
		body, err := ioutil.ReadAll(part)
		if err != nil {
			log.Fatalf("failed reading part: %s", err)
		}
		parts = append(parts, &Part{part, body})
		part.Close()
	}
	return parts
}

func DecodeBody(r io.Reader, encoding string) []byte {
	switch encoding {
	case "base64":
		dec := base64.NewDecoder(base64.StdEncoding, r)
		data, err := ioutil.ReadAll(dec)
		if err != nil {
			log.Fatalf("failed decoding: %s", err)
		}
		return data
	default:
		log.Fatalf("can't decode %s", encoding)
	}
	panic("not reached")
}

func DumpParts(parts []*Part, prefix string) {
	log.Printf("found %d parts", len(parts))
	for i, part := range parts {
		ctype := part.Header.Get("Content-Type")
		log.Printf(prefix+"part %d has Content-Type: %s", i+1, ctype)
		mtype, params, err := mime.ParseMediaType(ctype)
		if err != nil {
			log.Fatalf("failed parsing media type %s: %s", ctype, err)
		}
		switch mtype {
		case "text/plain", "text/html":
			log.Printf(prefix+"content: %s", part.Body)
		case "application/octet-stream":
			body := DecodeBody(part.Reader(), part.Header.Get("Content-Transfer-Encodin
```

```go
                        log.Printf(prefix+"decoded attachment with contents: %s", body)
                case "multipart/alternative":
                        altParts := ReadMultipart(part.Reader(), params["boundary"])
                        DumpParts(altParts, prefix+"\t")
                }
        }
}

func main() {
        parts := ReadMultipartFile("body", "047d7bae420e40e18a04e7e1ead4")
        DumpParts(parts, "")
}
```

Multipart Generation

The other fun part of multipart things is generating them. When you upload a file in an HTML form, you have to mark the form as `multipart/form-data`. With the `mime/multipart` package you can therefore generate the body for an HTTP request where a file is sent along.

Using requestb.in[68] and jsfiddle[69], you can see that a form like this would produce a multipart body that looks like what the example produces.

mime/form.html

```html
<form method="POST" enctype="multipart/form-data" action="/server">
    <input type="text" name="book" />
    <input type="text" name="chapter" />
    <input type="text" name="examples" />
    <input type="file" name="uploaded" />
    <input type="submit" value="Submit" />
</form>
```

On with the example!

[68] http://requestb.in/
[69] http://jsfiddle.net/

mime/generate.go

```go
package main

import (
	"bytes"
	"io"
	"log"
	"mime/multipart"
	"os"
)

func Must(err error) {
	if err != nil {
		log.Fatalf("WriteField failed: %s", err)
	}
}

func WriteFile(w io.Writer, filename string) {
	file, err := os.Open(filename)
	if err != nil {
		log.Fatalf("failed opening file: %s", err)
	}
	defer file.Close()

	_, err = io.Copy(w, file)
	if err != nil {
		log.Fatalf("failed writing file: %s", err)
	}
}

func Generate(w io.Writer) string {
	wr := multipart.NewWriter(w)
	defer wr.Close()
	Must(wr.WriteField("book", "Go, The Standard Library"))
	Must(wr.WriteField("chapter", "mime"))
	Must(wr.WriteField("examples", "2"))
	ff, err := wr.CreateFormFile("uploaded", "generate.go")
	if err != nil {
		log.Fatalf("failed creating form file: %s", err)
	}
	WriteFile(ff, "generate.go")
	return wr.Boundary()
```

```go
}

func Parse(r io.Reader, boundary string) {
	rd := multipart.NewReader(r, boundary)
	form, err := rd.ReadForm(1024 * 1024 * 1024)
	if err != nil {
		log.Fatalf("failed reading form: %s", err)
	}

	for name, value := range form.Value {
		log.Printf("got form data %s: %s", name, value)
	}

	for name, fhs := range form.File {
		for _, fh := range fhs {
			log.Printf("got form file %s: %s", name, fh.Filename)
		}
	}
}

func main() {
	var buffer bytes.Buffer
	boundary := Generate(&buffer)
	log.Println(buffer.String())
	Parse(&buffer, boundary)
}
```

net (wip)

mail

os

The `os` package is a package you'll use fairly often, but probably just for the file IO and and maybe pulling things from the environment. If you're building some system utility, you'll be using a lot more of this package.

With the `os` package we have access to our 3 basic IO pipes, `stdin`, `stdout`, and `stderr`. We also get access to a null device, basically a place we can write data we don't care about.

We can change permissions on files, inspect the environment, and create and remove files. We can read and write files too, and inspect their metadata.

We can start other processes and provide input to them and read their output. We can also respond to incoming signals and make raw syscalls.

The `os` package is an important one, and provides the hooks into the operating system to get real work done. It should be every gopher's goal to learn it front and back. As a result, there are a lot of examples in this chapter.

`stdio` and `DevNull`

The 3 basic IO pipes every process has are `stdin`, `stdout`, and `stderr`. `stdin` allows the process to read data from outside itself. `stdout` is the main place the process can write output. `stderr` is where the process should write error information.

We also have a null device, which is essentially a black hole for writing data. It writes, but goes nowhere.

Run this program by piping in some data: `echo data | go run stdio.go`.

os/stdio.go

```go
package main

import (
	"io"
	"io/ioutil"
	"log"
	"os"
)

func init() {
	log.SetFlags(0)
	log.SetPrefix("» ")
}

func DemoStdin() {
	input, err := ioutil.ReadAll(os.Stdin)
	if err != nil {
		log.Fatalf("failed reading stdin: %s", err)
	}
	log.Printf("read %d from stdin: %s", len(input), input)
}

func DemoDevNull() {
	devNull, err := os.Open(os.DevNull)
	if err != nil {
		log.Fatalf("failed opening null device: %s", err)
	}
	defer devNull.Close()
	io.WriteString(devNull, "This is going nowhere\n")
}

func main() {
	io.WriteString(os.Stdout, "This is stdout\n")
	io.WriteString(os.Stderr, "This is stderr\n")
	DemoDevNull()
	DemoStdin()
}
```

Output:

```
This is stdout
This is stderr
» read 25 from stdin: This data going on stdin
```

Permissions

Sometimes, file permissions just aren't correct. Sometimes, you need to set specific permissions on files you create. You can change two kinds of permissions: the mode of the file, and the owner (user and group). You could do this with bash, or whatever your preferred shell is, but sometimes you need to manipulate permissions as part of a larger program where bash isn't appropriate.

You can also change the access and modified times of a file, if you want to get sneaky like that.

> You might be looking at this example and wondering where the `os.Chown` example is. Well, the annoying part is that function works with `uid` and `gid` and not names. There also aren't any places to convert names to ids. Yes, you could probably write some syscalls for it, or even parse `/etc/group`. I'll leave this as an exercise for the reader.

os/permissions.go

```go
package main

import (
	"flag"
	"log"
	"os"
	"strconv"
)

var (
	chmod = flag.String("chmod", "", "the file to chmod")
	mode  = flag.String("mode", "", "the mode to set")
)

func init() {
```

```
        log.SetFlags(0)
        log.SetPrefix("» ")
        flag.Parse()
}

func main() {
        fileMode, err := strconv.ParseUint(*mode, 8, 32)
        if err != nil {
                log.Fatalf("invalid mode: %s", err)
        }

        err = os.Chmod(*chmod, os.FileMode(fileMode))
        if err != nil {
                log.Fatalf("failed chmod: %s", err)
        }
}
```

String Expansion

A common task when writing system utilities is expanding string values using the environment variables. Luckily the `os` package has what we need in the form of the `ExpandEnv` function. It also has abstracted the pattern to `Expand` so you can provide your own function to retrieve values. As the documentation points out, `ExpandEnv(s string)` is just `Expand(s string, os.Getenv)`. We'll look at both of these functions.

os/expand.go

```
package main

import (
        "flag"
        "log"
        "os"
)

type expandable map[string]string

func (e expandable) Expand(s string) string {
        return e[s]
}

```

```
func init() {
	log.SetFlags(0)
	log.SetPrefix("» ")
	flag.Parse()
}

func DemoExpandEnv() {
	log.Println(os.ExpandEnv("$HOME"))
	log.Println(os.ExpandEnv("$PWD"))
}

func DemoExpand() {
	exp := expandable(map[string]string{
		"name":  "Superman",
		"alter": "Clark Kent",
	})
	log.Println(os.Expand("${name} is really ${alter}", exp.Expand))
}

func main() {
	DemoExpandEnv()
	DemoExpand()
}
```

Output:

```
» /Users/darkhelmet
» /Users/darkhelmet/dev/github/darkhelmet/go-thestdlib/manuscript/code/os
» Superman is really Clark Kent
```

Moving Around the Environment

Frequently you need to move around the filesystem to get your work done. Sometimes you need want to switch into a specific directory to simply make life easier. Luckily moving around is dead simple given two simple functions provided by the `os` package: `Chdir` and `Getwd`.

The important stuff happens in the `RunInDir` function. We use `Getwd` to figure out where we are, so we can return to it using `defer` (not strictly required, but it's good to "clean up"). We then use `Chdir` to do the actual moving around.

os/moving.go

```go
package main

import (
	"flag"
	"io/ioutil"
	"log"
	"os"
	"path/filepath"
)

func init() {
	log.SetFlags(0)
	log.SetPrefix("» ")
	flag.Parse()
}

func RunInDir(dir string, f func(string)) {
	dir, err := filepath.Abs(dir)
	if err != nil {
		log.Fatalf("failed getting absolute directory path: %s", err)
	}

	cwd, err := os.Getwd()
	if err != nil {
		log.Fatalf("failed to get working directory: %s", err)
	}

	os.Chdir(dir)
	defer os.Chdir(cwd)
	f(dir)
}

func Dir() []os.FileInfo {
	files, err := ioutil.ReadDir(".")
	if err != nil {
		log.Fatalf("failed reading directory: %s", err)
	}
	return files
}

func main() {
```

```go
        f := func(cwd string) {
                files, err := ioutil.ReadDir(".")
                if err != nil {
                        log.Fatalf("failed reading directory: %s", err)
                }
                log.Printf("found %d files in %s", len(files), cwd)
        }

        RunInDir(".", f)
        RunInDir("..", f)
        RunInDir("../..", f)
        RunInDir("../log", f)
        RunInDir("../../..", f)
}
```

Output:

```
» found 8 files in /Users/darkhelmet/dev/github/darkhelmet/go-thestdlib/manuscript/c\
ode/os
» found 38 files in /Users/darkhelmet/dev/github/darkhelmet/go-thestdlib/manuscript/\
code
» found 15 files in /Users/darkhelmet/dev/github/darkhelmet/go-thestdlib/manuscript
» found 2 files in /Users/darkhelmet/dev/github/darkhelmet/go-thestdlib/manuscript/c\
ode/log
» found 4 files in /Users/darkhelmet/dev/github/darkhelmet/go-thestdlib
```

Inspecting the Environment

Another important task is inspecting the environment a process is running in, and inspecting the process itself. Getting environment variables, the process ID, and information about the running user are all common tasks. Naturally, the `os` package provides some simple functions to take care of this business.

> This example is more interesting if you compile it and do some permission munging before hand. Compile with `go build inspecting.go`. Then `sudo chown root inspecting` and `sudo chmod u+s inspecting`. Now, when you run the program with `./inspecting`, you should see that the user id is your user's id, but the *effective* user id, is 0, that of the root user. This way, you can get the id of the user actually running the program, but

also inspect the effective id. The effective ids are used when it comes to checking permissions.

os/inspecting.go

```go
package main

import (
	"flag"
	"log"
	"os"
)

func init() {
	log.SetFlags(0)
	log.SetPrefix("» ")
	flag.Parse()
}

func DemoProcessIds() {
	log.Printf("process id: %d", os.Getpid())
	log.Printf("parent process id: %d", os.Getppid())
}

func DemoUserInfo() {
	// actually running the program
	log.Printf("user id: %d", os.Getuid())
	log.Printf("group id: %d", os.Getgid())

	// permissions
	log.Printf("effective user id: %d", os.Geteuid())
	log.Printf("effective group id: %d", os.Getegid())

	groups, err := os.Getgroups()
	if err != nil {
		log.Fatalf("failed getting groups: %s", err)
	}
	log.Printf("groups you belong to: %d", groups)
}

func DemoExtra() {
	log.Printf("$GOPATH: %s", os.Getenv("GOPATH"))
```

```go
	log.Printf("$TMPDIR: %s", os.Getenv("TMPDIR"))

	log.Printf("pagesize: %d bytes", os.Getpagesize())

	hostname, err := os.Hostname()
	if err != nil {
		log.Fatalf("failed getting hostname: %s", err)
	}
	log.Printf("hostname: %s", hostname)
}

func main() {
	DemoProcessIds()
	DemoUserInfo()
	DemoExtra()
}
```

Output:

```
» process id: 26500
» parent process id: 26499
» user id: 501
» group id: 20
» effective user id: 0
» effective group id: 20
» groups you belong to: [20 503 501 12 61 79 80 81 98 399 502 402 33 100 204 398]
» $GOPATH: /Users/darkhelmet/dev/go
» $TMPDIR: /var/folders/t2/k4y07r396d5006j7y9w9zldc0000gn/T/
» pagesize: 4096 bytes
» hostname: ada.local
```

Creating and Removing Files and Directories

Creating directories, removing them, renaming things, and managing links are all common tasks when dealing with an operating system. Luckily the `os` package has you covered. You can make & remove directories, remove files, rename things, manage links, and even change the size of files.

You can run this example, but it will cleanup everything it does. You might want to comment out a bunch of lines, mainly cleanup, and see how the filesystem changes. Clean things up yourself, then uncomment things, rinse and repeat until you understand all the changes.

os/managing_files.go

```go
package main

import (
	"flag"
	"log"
	"os"
)

func init() {
	log.SetFlags(0)
	log.SetPrefix("» ")
	flag.Parse()
}

func must(err error) {
	if err != nil {
		log.Fatalf("failed operation: %s", err)
	}
}

func DemoMkdir() {
	must(os.MkdirAll("foo/bar/baz", 0755))
	must(os.Mkdir("example", 0755))
}

func CleanupDir() {
	must(os.RemoveAll("foo"))
	must(os.Remove("example"))
}

func DemoLink() {
	must(os.Symlink("Makefile", "Makefile-symlink"))
	must(os.Link("Makefile", "Makefile-link"))
}
```

```go

func CleanupLink() {
	must(os.Remove("Makefile-symlink"))
	must(os.Remove("Makefile-link"))
}

func DemoRename() {
	must(os.Rename("Makefile", "makefile"))
}

func CleanupRename() {
	must(os.Rename("makefile", "Makefile"))
}

func DemoTruncate() {
	// Look at the size of Makefile after this
	// Content hasn't changed, but it's magically 1kb
	must(os.Truncate("Makefile", 1024))
}

func CleanupTruncate() {
	must(os.Truncate("Makefile", 315))
}

func main() {
	DemoMkdir()
	CleanupDir()
	DemoLink()
	CleanupLink()
	DemoRename()
	CleanupRename()
	DemoTruncate()
	CleanupTruncate()
}
```

File IO

File IO is of course another big important task when writing system software, or frankly any software. Need to read a configuration file? File IO. Want to talk to your database via a UNIX socket? That's file IO. Writing some fancy NoSQL database? You better believe that's file IO.

The `os` package has the `File` type, and functions like `Create`, `NewFile`, `Open`, and `OpenFile` to help do all the file IO things you could want. The actually `os.File` type is a struct that has a variety of basic methods, but it also abides by various interface types, such as `io.Reader`, `io.Writer`, and others. This means if the file doesn't have the method you want, you can probably dive into the `io` package for help.

os/file_io.go

```go
package main

import (
	"io/ioutil"
	"log"
	"os"
)

func init() {
	log.SetFlags(0)
	log.SetPrefix("» ")
}

func DemoCreate() {
	f, err := os.Create("demo.txt") // Truncates if file already exists, be careful!
	if err != nil {
		log.Fatalf("failed creating file: %s", err)
	}
	defer f.Close() // Make sure to close the file when you're done

	n, err := f.WriteString(`"And live from New York, it's Saturday Night!" - Cast of S\
NL`)

	if err != nil {
		log.Fatalf("failed writing to file: %s", err)
	}
	log.Printf("wrote %d bytes to %s", n, f.Name())
}

func DemoOpenFile() {
	// OpenFile lets you customize whether the file is truncated, must exist, or must n\
ot exist, etc
	// Open is your basic way to open a file for reading, but we need to write.
	f, err := os.OpenFile("demo.txt", os.O_WRONLY|os.O_APPEND, 0644)
```

```go
	if err != nil {
		log.Fatalf("failed opening file: %s", err)
	}
	defer f.Close()

	n, err := f.WriteString("\nSince 1985\n")
	if err != nil {
		log.Fatalf("failed writing to file: %s", err)
	}
	log.Printf("wrote another %d bytes to %s", n, f.Name())
}

func DemoWriteAt() {
	// In DemoOpenFile, we wrote the wrong date, let's fix that
	f, err := os.OpenFile("demo.txt", os.O_RDWR, 0644)
	if err != nil {
		log.Fatalf("failed opening file: %s", err)
	}
	defer f.Close()

	n, err := f.WriteAt([]byte{'7'}, 69)
	if err != nil {
		log.Fatalf("failed writing to file: %s", err)
	}
	log.Printf("wrote another %d bytes to %s", n, f.Name())
}

func DemoRead() {
	f, err := os.Open("demo.txt")
	if err != nil {
		log.Fatalf("failed opening file: %s", err)
	}
	defer f.Close()

	data, err := ioutil.ReadAll(f)
	if err != nil {
		log.Fatalf("failed reading %s: %s", f.Name(), err)
	}
	log.Printf("contents:\n%s", data)
}

func main() {
	DemoCreate()
```

```
        DemoRead()
        DemoOpenFile()
        DemoRead()
        DemoWriteAt()
        DemoRead()
}
```

Output:

```
» wrote 60 bytes to demo.txt
» contents:
"And live from New York, it's Saturday Night!" - Cast of SNL
» wrote another 12 bytes to demo.txt
» contents:
"And live from New York, it's Saturday Night!" - Cast of SNL
Since 1985
» wrote another 1 bytes to demo.txt
» contents:
"And live from New York, it's Saturday Night!" - Cast of SNL
Since 1975
```

FileInfo

Of course, since directories are just files, you can do fun things with directories as well, like read their contents. If you open a directory, you can use methods like `Readdir` and `Readdirnames` to read all the entries.

os/file_info.go

```
package main

import (
        "log"
        "os"
)

func init() {
        log.SetFlags(0)
        log.SetPrefix("» ")
}

```

```go
func DemoReaddir() {
	f, err := os.Open(".")
	if err != nil {
		log.Fatalf("failed opening directory: %s", err)
	}
	defer f.Close()

	fileInfos, err := f.Readdir(0)
	if err != nil {
		log.Fatalf("failed reading directory: %s", err)
	}

	for _, finfo := range fileInfos {
		log.Printf("Name: %s, Size: %db", finfo.Name(), finfo.Size())
	}
}

func DemoReaddirnames() {
	f, err := os.Open(".")
	if err != nil {
		log.Fatalf("failed opening directory: %s", err)
	}
	defer f.Close()

	names, err := f.Readdirnames(0)
	if err != nil {
		log.Fatalf("failed reading directory: %s", err)
	}

	for _, name := range names {
		log.Println(name)
	}
}

func main() {
	DemoReaddir()
	DemoReaddirnames()
}
```

Output:

```
» Name: .gitignore, Size: 20b
» Name: demo.txt, Size: 72b
» Name: expand.go, Size: 537b
» Name: expand.txt, Size: 129b
» Name: file_info.go, Size: 785b
» Name: file_info.txt, Size: 0b
» Name: file_io.go, Size: 1824b
» Name: file_io.txt, Size: 349b
» Name: inspecting, Size: 2081952b
» Name: inspecting.go, Size: 1018b
» Name: inspecting.txt, Size: 360b
» Name: Makefile, Size: 315b
» Name: managing_files.go, Size: 1076b
» Name: moving.go, Size: 920b
» Name: moving.txt, Size: 433b
» Name: permissions.go, Size: 474b
» Name: stdio.go, Size: 645b
» Name: stdio.txt, Size: 78b
» .gitignore
» demo.txt
» expand.go
» expand.txt
» file_info.go
» file_info.txt
» file_io.go
» file_io.txt
» inspecting
» inspecting.go
» inspecting.txt
» Makefile
» managing_files.go
» moving.go
» moving.txt
» permissions.go
» stdio.go
» stdio.txt
```

Process Creation, Management, and Signals

The `os` package has a few different ways to deal with processes. You can create new processes, provide them input and capture their output. You can manage existing processes, sending them signals, and you can wait on them to finish. All pretty standard stuff.

Let's look at `os/exec` first. This is your basic use case, running an external process, maybe providing some input, and reading the output.

os/exec.go

```go
package main

import (
        "io"
        "log"
        "os/exec"
)

func init() {
        log.SetFlags(0)
        log.SetPrefix("» ")
}

func DemoExec() {
        cmd := exec.Command("date", "-u")
        out, err := cmd.Output()
        if err != nil {
                log.Printf("failed running command: %s", err)
        }
        log.Printf("date -u: %s", out)
}

func DemoExecInput() {
        cmd := exec.Command("ruby", "-r", "active_support/all")

        wr, err := cmd.StdinPipe()
        if err != nil {
                log.Fatalf("failed getting stdin: %s", err)
        }

        rd, _ := cmd.StdoutPipe()
```

```
        if err != nil {
                log.Fatalf("failed getting stdout: %s", err)
        }

        err = cmd.Start()
        if err != nil {
                log.Fatalf("failed starting command: %s", err)
        }
        defer cmd.Wait()

        io.WriteString(wr, "puts 1.hour;")
        io.WriteString(wr, "puts 1.day;")
        wr.Close()

        hour := make([]byte, 5)
        rd.Read(hour)
        log.Printf("1.hour: %s", hour)

        day := make([]byte, 6)
        rd.Read(day)
        log.Printf("1.hour: %s", day)
}

func main() {
        DemoExec()
        DemoExecInput()
}
```

Output:

```
» date -u: Tue 24 Nov 2015 14:00:59 UTC
» 1.hour: 3600
» 1.hour: 86400
```

Other things can be done with the API found in the base `os` package. It's more tailored to dealing with existing processes versus starting new ones. For example, if you wanted to build your own `htop` clone, you'll want use the functions exposed in `os`.

We'll look at using signals to control the process as well. When handling signals, there are 3 steps:

1. Make a channel of `os.Signal`

2. Call `signal.Notify` with your channel the signals you care about
3. Start a goroutine which pulls from the channel and deals with the signals

Using a single channel and goroutine with a switch statement is a simple understandable way to process the signals.

os/processes.go

```
package main

import (
	"log"
	"os"
	"os/signal"
	"sync/atomic"
	"syscall"
	"time"
)

var (
	signals = make(chan os.Signal, 1)
	val     int32
)

func init() {
	log.SetFlags(0)
	log.SetPrefix("» ")

	signal.Notify(signals, syscall.SIGUSR1, syscall.SIGUSR2)
	go handleSignals()
}

func handleSignals() {
	for signal := range signals {
		switch signal {
		case syscall.SIGUSR1:
			log.Println("got USR1, adding 2")
			atomic.AddInt32(&val, 2)
		case syscall.SIGUSR2:
			log.Println("got USR2, subtracting 1")
			atomic.AddInt32(&val, -1)
		}
		log.Printf("val: %d", val)
```

```go
        }
}

func main() {
        os.Getpid()
        p, _ := os.FindProcess(os.Getpid())

        ticker := time.Tick(1 * time.Second)
        for now := range ticker {
                switch {
                case val > 5:
                        p.Kill()
                case now.Second()%2 == 0: // even
                        p.Signal(syscall.SIGUSR1)
                case now.Second()%2 == 1: // odd
                        p.Signal(syscall.SIGUSR2)
                }
        }
}
```

Output:

```
» got USR1, adding 2
» val: 2
» got USR2, subtracting 1
» val: 1
» got USR1, adding 2
» val: 3
» got USR2, subtracting 1
» val: 2
» got USR1, adding 2
» val: 4
» got USR2, subtracting 1
» val: 3
» got USR1, adding 2
» val: 5
» got USR2, subtracting 1
» val: 4
» got USR1, adding 2
» val: 6
signal: killed
```

Users

Finally, we'll look at the `os/user` package, which lets you query the system about the users on it. You can lookup users by name or id, or just get the current user. It's nothing to spectacular, but it's functionality you can use, so let's check it out.

os/user.go

```go
package main

import (
	"log"
	"os/user"
)

func init() {
	log.SetFlags(0)
	log.SetPrefix("» ")
}

func DemoCurrent() {
	u, _ := user.Current()
	log.Printf("%#v", u)
}

func DemoLookup() {
	u, _ := user.Lookup("nobody")
	log.Printf("%#v", u)
}

func DemoLookupId() {
	u, _ := user.LookupId("1")
	log.Printf("%#v", u)
}

func main() {
	DemoCurrent()
	DemoLookup()
	DemoLookupId()
}
```

Output:

```
» &user.User{Uid:"501", Gid:"20", Username:"darkhelmet", Name:"Daniel Huckstep", Hom\
eDir:"/Users/darkhelmet"}
» &user.User{Uid:"4294967294", Gid:"4294967294", Username:"nobody", Name:"Unprivileg\
ed User", HomeDir:"/var/empty"}
» &user.User{Uid:"1", Gid:"1", Username:"daemon", Name:"System Services", HomeDir:"/\
var/root"}
```

path

The `path` package is used for dealing with strings representing slash separated paths.

Okay, so why is there `path/filepath` as well? `path` assumes a forward slash (`/`) as the separator and doesn't make any other assumptions, like what is used to separate lists of paths. The `path/filepath` package deals with different separators for different operating systems. For example, Windows uses the backslash, where the rest of the world uses a forward slash.

It can also deal with lists of paths and their operating system specific separators, and has a way to recursively walk a directory structure.

The APIs are very straightforward, so let's dive right in.

path

path/path.go

```go
package main

import (
	"flag"
	"log"
	"path"
)

func main() {
	var p string
	flag.StringVar(&p, "path", "./foo/../baz.gif", "the path to examine")
	flag.Parse()

	log.Printf("p: %s", p)
	log.Printf("Base(p): %s", path.Base(p))
	log.Printf("Clean(p): %s", path.Clean(p))
	log.Printf("Dir(p): %s", path.Dir(p))
	log.Printf("Ext(p): %s", path.Ext(p))
	log.Printf("IsAbs(p): %t", path.IsAbs(p))
	log.Printf("Join(\"/fizz/bin\", p): %s", path.Join("/fizz/bin", p))
```

```go

	matched, err := path.Match("/*/bin/*.gif", p)
	log.Printf("Match(\"/*/bin/*.gif\", p): %t, %v", matched, err)

	matched, err = path.Match("/*/bin/*.gif", path.Join("/fizz/bin", p))
	log.Printf("Match(\"/*/bin/*.gif\", Join(\"/fizz/bin\", p)): %t, %v", matched, err)

	dir, file := path.Split(p)
	log.Printf("Split(p): %s, %s", dir, file)
}
```

Pretty easy right? Try running it with different arguments for `-path` to see what some of the results are.

`path/filepath`

How about `path/filepath`? Again, run this with different arguments for `-path`, as some operations need a file to actually exist, like `EvalSymlinks`.

path/filepath.go

```go
package main

import (
	"flag"
	"log"
	"os"
	"path/filepath"
)

var (
	p          string
	walk       string
	ignore     string
	ignoreList []string
)

type Walker struct {
	NumDirs  int
	NumFiles int
}
```

```go

func (w *Walker) Visit(path string, info os.FileInfo, err error) error {
	if info.IsDir() {
		base := filepath.Base(path)
		for _, dir := range ignoreList {
			if base == dir {
				return filepath.SkipDir
			}
		}
		w.NumDirs++
	} else {
		w.NumFiles++
	}
	return nil
}

func init() {
	flag.StringVar(&p, "path", "./foo/../baz.gif", "the path to examine")
	flag.StringVar(&walk, "walk", "..", "the path to walk")
	flag.StringVar(&ignore, "ignore", ".git:.hg", "directories to ignore")
	flag.Parse()

	ignoreList = filepath.SplitList(ignore)
}

func main() {
	log.Printf("p: %s", p)

	abs, err := filepath.Abs(p)
	log.Printf("Abs(p): %s, %v", abs, err)
	log.Printf("Base(p): %s", filepath.Base(p))
	log.Printf("Clean(p): %s", filepath.Clean(p))
	log.Printf("Dir(p): %s", filepath.Dir(p))

	sym, err := filepath.EvalSymlinks(p)
	log.Printf("EvalSymlinks(p): %s, %v", sym, err)
	log.Printf("Ext(p): %s", filepath.Ext(p))
	log.Printf("FromSlash(p): %s", filepath.FromSlash(p))

	glob, err := filepath.Glob("*.go")
	log.Printf("Glob(\"*.go\"): %s, %v", glob, err)
	log.Printf("IsAbs(p): %t", filepath.IsAbs(p))
	log.Printf("Join(\"/fizz/bin\", p): %s", filepath.Join("/fizz/bin", p))
```

```

        matched, err := filepath.Match("/*/bin/*.gif", p)
        log.Printf("Match(\"/*/bin/*.gif\", p): %t, %v", matched, err)

        matched, err = filepath.Match("/*/bin/*.gif", filepath.Join("/fizz/bin", p))
        log.Printf("Match(\"/*/bin/*.gif\", Join(\"/fizz/bin\", p)): %t, %v", matched, err)

        rel, err := filepath.Rel("/batman", "/path/file.go")
        log.Printf("Rel(\"/batman\", \"/path/file.go\"): %s, %v", rel, err)

        dir, file := filepath.Split(p)
        log.Printf("Split(p): %s, %s", dir, file)

        list := filepath.SplitList("/foo.go:/bar.go:/baz.go")
        log.Printf("SplitList(\"/foo.go:/bar.go:/baz.go\"): %s", list)

        var w Walker
        filepath.Walk("..", (&w).Visit)
        log.Printf("found %d directories and %d files", w.NumDirs, w.NumFiles)
}
```

find

Using `filepath.Walk`, and of course some other packages, we can replicate the functionality of the UNIX `find` utility. It's far from perfect, not exact, and obviously doesn't cover everything that `find` has to offer, but it's a start and you can see how you could implement the rest.

path/find.go

```
package main

import (
        "flag"
        "fmt"
        "os"
        "path/filepath"
)

type FilterFunc func(path string, info os.FileInfo, err error) bool
type FilterChain []FilterFunc
```

```

var (
        root                 string
        ftype, name          string
        printNewline, print0 bool
        filters              FilterChain

        output = func(s string) {}
)

func init() {
        flag.StringVar(&ftype, "type", "", "f for file, d for directory")
        flag.StringVar(&name, "name", "", "find files/directories that match")
        flag.BoolVar(&printNewline, "print", false, "print elements to stdout with newlines\
 separators")
        flag.BoolVar(&print0, "print0", false, "print elements to stdout with NULL separato\
rs")
        flag.Parse()
        root = flag.Arg(0)
        if root == "" {
                root = "."
        }
}

func setupPrinting() {
        if printNewline {
                output = func(s string) { fmt.Println(s) }
        } else if print0 {
                output = func(s string) { fmt.Printf("%s\x00", s) }
        } else {
                output = func(s string) { fmt.Println(s) }
        }
}

func nameFilter(path string, info os.FileInfo, err error) bool {
        matched, err := filepath.Match(name, filepath.Base(path))
        if err != nil {
                fmt.Printf("failed matching: %s", err)
                os.Exit(1)
        }
        return matched
}

```

```go
func fileFilter(path string, info os.FileInfo, err error) bool {
	return !info.IsDir()
}

func directoryFilter(path string, info os.FileInfo, err error) bool {
	return info.IsDir()
}

func ok(path string, info os.FileInfo, err error) bool {
	return true
}

func setupFilters() {
	switch ftype {
	case "f":
		filters = append(filters, fileFilter)
	case "d":
		filters = append(filters, directoryFilter)
	}

	if name != "" {
		filters = append(filters, nameFilter)
	}

	if len(filters) == 0 {
		filters = append(filters, ok)
	}
}

func walker(path string, info os.FileInfo, err error) error {
	for _, filter := range filters {
		if !filter(path, info, err) {
			return nil
		}
	}
	output(path)
	return nil
}

func main() {
	setupPrinting()
	setupFilters()
	filepath.Walk(root, walker)
```

```
}
```

reflect

The `reflect` package is the kind of package you probably shouldn't be messing around in too much. You can do some powerful things for sure, but the documentation page for the `reflect` package has 86 occurrences of the word `panic`. You can definitely shoot yourself in the foot.

That being said, you can do a lot of things you'd expect to be able to do.

Most operations revolve around working with a `reflect.Value`, but you can analyze and build types as well.

Select from an arbitrary number of channels

The `select` statement is incredibly useful in Go. It lets you deal with concurrent goroutines by selecting a channel to send on or receive from. A downside is that it's static, so you have to know all the channels you want to handle as you're writing code, at compile time.

This isn't any fun, so let's use the `reflect` package to select on an arbitrary number of channels.

reflect/select.go

```go
package main

import (
	"flag"
	"fmt"
	"log"
	"math/rand"
	"os"
	"reflect"
	"time"
)

var (
	count    = flag.Int("count", 10, "The number of channels to make")
	maxSleep = flag.Int("sleep", 15, "The maximum number of seconds a goroutine might s\
```

```
leep")
)

func init() {
	flag.Parse()
}

type P struct {
	Now   time.Time
	Index int
}

func (p P) String() string {
	return fmt.Sprintf("index=%d now=%s", p.Index, p.Now.Format(time.RFC3339))
}

func main() {
	// Build our channels and goroutines
	log.Printf("building %d channels, starting at %s", *count, time.Now())

	channels := make([]chan P, *count)
	for i := 0; i < *count; i++ {
		ch := make(chan P)
		channels[i] = ch
		go func(i int, ch chan P) {
			delay := time.Duration(rand.Intn(*maxSleep)) * time.Second
			log.Printf("channel %d sleeping for %s", i, delay)
			time.Sleep(delay)
			ch <- P{Now: time.Now(), Index: i}
			close(ch)
		}(i, ch)
	}

	// Build a slice of SelectCases
	cases := make([]reflect.SelectCase, 0, len(channels))
	for i, ch := range channels {
		value := reflect.ValueOf(ch)

		// Maybe this code is in a library, and your input is a []interface{},
		// it's good to ensure you were given channels and not something weird.
		// You would probably do something other that log.Fatalf though.
		if reflect.Chan != value.Kind() {
			log.Fatalf("%#v at index %d is not a channel,", value.Interface(), i)
```

```
		}

		// We can only handle receives, so make sure of that too.
		switch value.Type().ChanDir() {
		case reflect.SendDir:
			log.Fatalf("only recv channels allowed, channel %d is send-only", i)
		default:
			// All good
		}

		cases = append(cases, reflect.SelectCase{
			Dir:  reflect.SelectRecv,
			Chan: value,
		})
	}

	for {
		if len(cases) == 0 {
			log.Println("all done")
			os.Exit(0)
		}

		index, value, ok := reflect.Select(cases)

		if ok {
			log.Printf("got value %s", value)
		} else {
			cases = append(cases[:index], cases[index+1:]...)
		}
	}
}
```

reflect/select.txt

```
2020/06/28 14:44:46 building 10 channels, starting at 2020-06-28 14:44:46.990734 -03\
00 ADT m=+0.000143532
2020/06/28 14:44:46 channel 0 sleeping for 11s
2020/06/28 14:44:46 channel 8 sleeping for 1s
2020/06/28 14:44:46 channel 1 sleeping for 3s
2020/06/28 14:44:46 channel 2 sleeping for 10s
2020/06/28 14:44:46 channel 9 sleeping for 2s
2020/06/28 14:44:46 channel 5 sleeping for 1s
2020/06/28 14:44:46 channel 6 sleeping for 5s
```

```
2020/06/28 14:44:46 channel 7 sleeping for 14s
2020/06/28 14:44:46 channel 4 sleeping for 12s
2020/06/28 14:44:46 channel 3 sleeping for 0s
2020/06/28 14:44:46 got value index=3 now=2020-06-28T14:44:46-03:00
2020/06/28 14:44:47 got value index=8 now=2020-06-28T14:44:47-03:00
2020/06/28 14:44:47 got value index=5 now=2020-06-28T14:44:47-03:00
2020/06/28 14:44:48 got value index=9 now=2020-06-28T14:44:48-03:00
2020/06/28 14:44:49 got value index=1 now=2020-06-28T14:44:49-03:00
2020/06/28 14:44:51 got value index=6 now=2020-06-28T14:44:51-03:00
2020/06/28 14:44:56 got value index=2 now=2020-06-28T14:44:56-03:00
2020/06/28 14:44:57 got value index=0 now=2020-06-28T14:44:57-03:00
2020/06/28 14:44:58 got value index=4 now=2020-06-28T14:44:58-03:00
2020/06/28 14:45:00 got value index=7 now=2020-06-28T14:45:00-03:00
2020/06/28 14:45:00 all done
```

Write your own enumerable methods

Go has some easy constructs to iterate through collections like slices and maps, but maybe you just really like your Ruby enumerable methods. Don't worry, we can implement these, and the functions you pass in and values you get back can have real types.

I don't know if I'd use this in production. If you're writing Go, you probably care somewhat about performance, and reflection is added overhead to do simple things. I'm skipping a bunch of potential runtime error checking as well. You could ensure that you were passed a collection and a function, and that the function takes the appropriate number of arguments of the appropriate types, etc. For an implementation with all that error checking, see my enumerable library on GitHub[70].

With generics coming out in the future, there will no doubt be a better way to do this.

But enough with all that, now stop! Iterate and listen!

[70] https://github.com/darkhelmet/enumerable

reflect/enumerable.go

```go
package main

import (
	"log"
	"reflect"
)

func init() {
	log.SetFlags(0)
	log.SetPrefix("» ")
}

// Each iterates over a slice and calls a function for each element
func Each(col interface{}, fn interface{}) {
	cv := reflect.ValueOf(col)
	fnv := reflect.ValueOf(fn)
	length := cv.Len()

	for i := 0; i < length; i++ {
		input := cv.Index(i)
		fnv.Call([]reflect.Value{input})
	}
}

// EachWithIndex iterates over a slice and calls a function for each element also pa\
ssing the index
func EachWithIndex(col interface{}, fn interface{}) {
	cv := reflect.ValueOf(col)
	fnv := reflect.ValueOf(fn)
	length := cv.Len()

	for i := 0; i < length; i++ {
		input := cv.Index(i)
		fnv.Call([]reflect.Value{input, reflect.ValueOf(i)})
	}
}

// Map takes a collection, calls a mapper function for each element, and returns
// a new collection of the mapped values
func Map(col interface{}, fn interface{}) interface{} {
	cv := reflect.ValueOf(col)
```

```
        fnv := reflect.ValueOf(fn)

        // Make a slice to hold the mapped collection
        outputType := fnv.Type().Out(0)
        output := reflect.MakeSlice(reflect.SliceOf(outputType), cv.Len(), cv.Cap())

        EachWithIndex(col, func(v interface{}, idx int) {
                mapped := fnv.Call([]reflect.Value{reflect.ValueOf(v)})
                output.Index(idx).Set(mapped[0])
        })

        return output.Interface()
}

// Reduce takes a collection, a reducer function, and returns the collection reduced\
 to a single value.
func Reduce(col interface{}, fn interface{}, initial interface{}) interface{} {
        cv := reflect.ValueOf(col)
        fnv := reflect.ValueOf(fn)
        length := cv.Len()

        if length == 0 {
                return initial
        }

        start := 0
        var output reflect.Value
        if initial == nil {
                output = cv.Index(0)
                start = 1
        } else {
                output = reflect.ValueOf(initial)
        }

        for i := start; i < length; i++ {
                right := cv.Index(i)
                output = fnv.Call([]reflect.Value{output, right})[0]
        }

        return output.Interface()
}

func main() {
```

```
        n := []int{1, 2, 3}

        Each(n, func(i int) {
                log.Printf("got %d", i)
        })

        mapped := Map(n, func(i int) int {
                return i * i
        }).([]int)
        log.Printf("turned %#v into %#v", n, mapped)

        summed := Reduce(n, func(memo, i int) int {
                return memo + i
        }, nil).(int)
        log.Printf("got %d", summed)

        summed = Reduce(n, func(memo, i int) int {
                return memo + i
        }, 0).(int)
        log.Printf("with initial value: %d", summed)
}
```

reflect/enumerable.txt

```
» got 1
» got 2
» got 3
» turned []int{1, 2, 3} into []int{1, 4, 9}
» got 6
» with initial value: 6
```

Inspect struct tags

Another thing you can do with the `reflect` package is inspect any struct tags. Struct tags are used to add extra metadata onto the fields of a struct. The place you're most likely to see them used is with the `encoding/json` package or in an ORM type library talking to a database.

Let's look at retrieving data from a struct using the tags to find what we want.

reflect/tags.go

```go
package main

import (
	"log"
	"math"
	"reflect"
)

func init() {
	log.SetFlags(0)
	log.SetPrefix("» ")
}

type T1 struct {
	F1 string `kind:"key"`
	F2 int    `kind:"value"`
}

type T2 struct {
	F3 byte    `kind:"key"`
	F4 float64 `kind:"value"`
}

type T3 struct {
	F5 bool
	F6 int
}

func GetValueOfFieldOfKind(i interface{}, kind string) interface{} {
	st := reflect.TypeOf(i)
	// Iterate through all the fields
	for idx := 0; idx < st.NumField(); idx++ {
		field := st.Field(idx)
		// Lookup the tag of "kind"
		if k, ok := field.Tag.Lookup("kind"); ok {
			// If it's kind:"key"...
			if k == kind {
				value := reflect.ValueOf(i)
				// Return the value of rhe field
				return value.Field(idx).Interface()
			}
```

```
                }
        }
        return nil
}

// GetKey returns the key of a struct, identified by a struct tag kind
func GetKey(i interface{}) interface{} {
        return GetValueOfFieldOfKind(i, "key")
}

// GetValue returns the value of a struct, identified by a struct tag kind
func GetValue(i interface{}) interface{} {
        return GetValueOfFieldOfKind(i, "value")
}

func main() {
        t1 := T1{F1: "a key", F2: 5}
        t2 := T2{F3: 'k', F4: math.Pi}
        t3 := T3{F5: true, F6: 7}

        log.Printf("got %#v: %#v from t1", GetKey(t1), GetValue(t1))
        log.Printf("got %#v: %#v from t2", GetKey(t2), GetValue(t2))
        log.Printf("got %#v: %#v from t3", GetKey(t3), GetValue(t3))
}
```

reflect/tags.txt

```
» got "a key": 5 from t1
» got 0x6b: 3.141592653589793 from t2
» got <nil>: <nil> from t3
```

regexp

The `regexp` package deals with, you guessed it, regular expressions.

First, you must compile an expression. You'll probably want to compile a package level variable using `regexp.MustCompile`, which will `panic` immediately at runtime. This ensures that you're only compiling the regexp once, and you avoid checking the compile error every time.

Once you have your compiled expression, there are a whole slew of methods following a pattern. They all start with `Find`. `Find` on its own works on bytes and finds the first occurrence.

- Methods with `All` return all matching things instead of just the first.
- Methods with `String` work on `string` inputs and return `string` matches.
- Methods with `Index` returns indexes of matches.
- Methods with `Submatch` gives you information about capture groups in the regexp.

There are also methods to replace matches, work on `io.RuneReader`s, and split strings.

The syntax is generally compatible with other languages like Ruby, Python, and Perl, but not 100%. It gives up some things in favor of safe/predictable performance characteristics. The regexp engine is based around re2[71] and more information about the performance bits can be found here[72]. For full syntax, look at the `regexp/syntax` package. You can optionally work with the POSIX compatible syntax subset by compiling your regexp with `CompilePOSIX`.

The `regexp/syntax` package also provides ways to work with the regexp parse tree.

Matching

Matching is the basic thing everybody does with regular expressions, and it's super simple, so naturally the example is short.

[71] http://swtch.com/~rsc/regexp/regexp1.html
[72] http://swtch.com/~rsc/regexp/regexp1.html

regexp/matching.go

```go
package main

import (
	"bytes"
	"log"
	"regexp"
)

var (
	universes = regexp.MustCompile(`(batman and robin)|(thor and loki)`)
	heroes    = "batman and robin"
)

func main() {
	log.Println(universes.MatchString(heroes))
	log.Println(universes.Match([]byte(heroes)))
	rr := bytes.NewBufferString(heroes)
	log.Println(universes.MatchReader(rr))

	log.Println(universes.MatchString("batman and loki"))
}
```

Output:

```
2014/01/13 21:06:31 true
2014/01/13 21:06:31 true
2014/01/13 21:06:31 true
2014/01/13 21:06:31 false
```

Indexes

There's only so far you can get with knowing only whether the entire string matches the regexp. The next step is finding the indexes of matches.

I won't bother showing the non-string functions, since they operate the same as the ones using strings, they just use byte slices.

regexp/indexes.go

```
package main

import (
	"log"
	"regexp"
)

var (
	eqn = "3x * 2y - 9 / 3 / 5 * 5"
	mul = regexp.MustCompile(`\w+ \* \w+`)
	div = regexp.MustCompile(`\w+ / \w+`)
)

func main() {
	fmul := mul.FindStringIndex(eqn)
	log.Println(fmul, eqn[fmul[0]:fmul[1]])

	divs := div.FindAllStringIndex(eqn, -1)
	log.Println("divs", divs)
	for index, pair := range divs {
		log.Printf("match %d: %s", index, eqn[pair[0]:pair[1]])
	}
}
```

Output:

```
2014/01/13 21:06:17 [0 7] 3x * 2y
2014/01/13 21:06:17 divs [[10 15]]
2014/01/13 21:06:17 match 0: 9 / 3
```

In this example, note that `FindAllStringIndex` with the `div` regexp only matches 1 thing despite there being two division operations in the equation. This is because the two operations overlap with the `3`. It gets picked up as part of the first operation, but then the regexp starts reading at the space after `3` and can't match a full division, so you only get one match.

Capture Groups and Submatches

Submatches are the way to extract capture groups out of a string given a regexp. We'll parse an nginx log line.

regexp/submatches.go

```go
package main

import (
	"log"
	"regexp"
)

type Matcher struct {
	*regexp.Regexp
}

func (m *Matcher) FindAllStringSubmatchMap(s string) map[string]string {
	pairs := make(map[string]string)

	// Ignore the first one, it's the "whole" match
	subexpNames := m.SubexpNames()[1:]
	submatches := m.FindAllStringSubmatch(s, -1)
	if submatches == nil {
		return pairs
	}

	// Ignore the first one, it's the "whole" match
	for index, submatch := range submatches[0][1:] {
		name := subexpNames[index]
		pairs[name] = submatch
	}
	return pairs
}

var (
	nginxLogFormat = &Matcher{regexp.MustCompile(`(?P<RemoteAddr>\S+) (?P<Host>\S+) - \\
[(?P<Time>[^\]]+)\] "(?P<Method>\S+) (?P<Path>\S+) [^"]+" (?P<Status>\d+) (?P<Bytes>\
\d+) "(?P<UserAgent>[^"]+)" "(?P<Referer>[^"]+)" (?P<RequestTime>\d+\.\d+)`)}
	// log_format timed_combined '$remote_addr $host $remote_user [$time_local] "$reque\
st" $status $body_bytes_sent "$http_referer" "$http_user_agent" $request_time';
	logLine = `74.86.158.107 example.com - [01/Dec/2013:18:07:26 -0700] "GET /en/landin\
```

```
g HTTP/1.1" 302 108 "-" "Mozilla/5.0+(compatible; UptimeRobot/2.0; http://www.uptime\
robot.com/)" 0.087`
)

func main() {
	log.Printf("NumSubexp: %d", nginxLogFormat.NumSubexp())
	subexpNames := nginxLogFormat.SubexpNames()
	log.Printf("SubexpNames: %v", subexpNames)
	submatches := nginxLogFormat.FindAllStringSubmatch(logLine, -1)
	log.Println(submatches)
	log.Println(nginxLogFormat.FindAllStringSubmatchMap(logLine))
}
```

Output:

```
2014/01/13 21:06:02 NumSubexp: 10
2014/01/13 21:06:02 SubexpNames: [ RemoteAddr Host Time Method Path Status Bytes Use\
rAgent Referer RequestTime]
2014/01/13 21:06:02 [[74.86.158.107 example.com - [01/Dec/2013:18:07:26 -0700] "GET \
/en/landing HTTP/1.1" 302 108 "-" "Mozilla/5.0+(compatible; UptimeRobot/2.0; http://\
www.uptimerobot.com/)" 0.087 74.86.158.107 example.com 01/Dec/2013:18:07:26 -0700 GE\
T /en/landing 302 108 - Mozilla/5.0+(compatible; UptimeRobot/2.0; http://www.uptimer\
obot.com/) 0.087]]
2014/01/13 21:06:02 map[Host:example.com Path:/en/landing Referer:Mozilla/5.0+(compa\
tible; UptimeRobot/2.0; http://www.uptimerobot.com/) RemoteAddr:74.86.158.107 Time:0\
1/Dec/2013:18:07:26 -0700 Method:GET Status:302 Bytes:108 UserAgent:- RequestTime:0.\
087]
```

Replace

Replacing things in text is something everybody does with regular expressions, so let’s look at that.

regexp/replace.go

```go
package main

import (
	"log"
	"regexp"
	"strings"
)

var (
	redaction = regexp.MustCompile(`(password|token)=(\w+)`)
	pairs     = regexp.MustCompile(`(\w+)=`)
	logLine   = `2013-12-02T02:40:57.049407+00:00 app: at=info method=POST path=/login \
token=secret host=example.com password=sekrit connect=1ms service=82ms status=200 by\
tes=809`
)

func main() {
	log.Println(redaction.ReplaceAllString(logLine, "$1=[REDACTED]"))
	log.Println(pairs.ReplaceAllStringFunc(logLine, strings.ToUpper))
}
```

Output:

```
2014/01/13 21:05:08 2013-12-02T02:40:57.049407+00:00 app: at=info method=POST path=/\
login token=[REDACTED] host=example.com password=[REDACTED] connect=1ms service=82ms\
 status=200 bytes=809
2014/01/13 21:05:08 2013-12-02T02:40:57.049407+00:00 app: AT=info METHOD=POST PATH=/\
login TOKEN=secret HOST=example.com PASSWORD=sekrit CONNECT=1ms SERVICE=82ms STATUS=\
200 BYTES=809
```

io

The `regexp` package can also deal with `io` things directly, specifically the `io.RuneReader` interface. There are problems with this, in that it obviously has to read data, which changes the state of the reader. If that's not a problem for you and using the reader makes sense, continue on. There is a limited set of methods, but they can be useful.

regexp/reader.go

```go
package main

import (
	"bufio"
	"log"
	"os"
	"regexp"
)

var (
	function = regexp.MustCompile(`func (\w+)`)
)

func main() {
	file, err := os.Open("reader.go")
	if err != nil {
		log.Fatalf("failed opening file: %s", err)
	}
	defer file.Close()
	rr := bufio.NewReader(file)
	log.Println(function.MatchReader(rr))
}
```

Output:

```
2014/01/13 21:17:59 true
```

runtime

The `runtime` package is your window into the Go runtime. Yes, even though it's a compiled language, there's still a runtime under the covers.

The big thing the runtime controls is the goroutines, so a lot of the functions deal with that. It also keeps track of some metrics, can give you information about memory usage, and a few other little things.

> Some of the functions aren't really supposed to be used by you, the average non-Go runtime programmer, and are commented as such.

While not really specific to the `runtime` package, there are also a few environment variables that you can use to control the runtime itself. Some of them do have functions to set values while your program is running. The package docs do a good job of covering their use, and you probably won't need to use them unless you hit a specific situation. If you find yourself in one of those fun debugging scenarios, check the package docs.

Some of these are difficult to demo, but they are equally rarely used. You'll probably run into a problem and know the feature you need to fix it. In that case, check the `runtime` package.

There are a few sub-packages as well to solve more specific problems: `runtime/debug` and `runtime/pprof`.

Introspection

You can learn a few things about your program, like the compiler, language version, `GOOS`, `GOARCH`, and `GOROOT`.

runtime/introspection.go

```go
package main

import (
	"log"
	"runtime"
)

func init() {
	log.SetFlags(0)
	log.SetPrefix("» ")
}

func main() {
	log.Printf("GOOS:\t%s", runtime.GOOS)
	log.Printf("GOARCH:\t%s", runtime.GOARCH)
	log.Printf("GOROOT:\t%s", runtime.GOROOT())
	log.Printf("Compiler:\t%s", runtime.Compiler)
	log.Printf("Version:\t%s", runtime.Version())
}
```

Output:

```
» GOOS:         darwin
» GOARCH:         amd64
» GOROOT:         /Users/darkhelmet/local/go
» Compiler:         gc
» Version:         go1.3
```

Goroutines

Need a goroutine to stay on one CPU? Need to exit from a goroutine immediately? Need to see how many goroutines are running right now? The `runtime` package can do that.

`LockOSThread` is useful if you're interfacing with a C library that requires you stay on the same thread, like the VLC library. Keep in mind this isn't the same as CPU affinity.

runtime/goroutines.go

```go
package main

import (
	"log"
	"runtime"
)

func init() {
	log.SetFlags(0)
	log.SetPrefix("» ")
}

func main() {
	log.Printf("GOMAXPROCS: %d", runtime.GOMAXPROCS(0))
	runtime.GOMAXPROCS(runtime.NumCPU()) // Use the whole CPU
	log.Printf("GOMAXPROCS: %d", runtime.GOMAXPROCS(0))

	log.Printf("There are %d goroutines running", runtime.NumGoroutine())

	done := make(chan bool)
	go func() {
		log.Println("in the goroutine")

		runtime.LockOSThread()
		log.Println("locked to this OS thread")
		runtime.Gosched() // Let the CPU go

		runtime.UnlockOSThread()
		log.Println("unlocked")
		runtime.Gosched() // Let the CPU go

		// runtime.Goexit() // Will cause a deadlock

		done <- true
	}()

	log.Printf("There are %d goroutines running", runtime.NumGoroutine())
	<-done
}
```

Output:

```
» GOMAXPROCS: 1
» GOMAXPROCS: 8
» There are 4 goroutines running
» There are 5 goroutines running
» in the goroutine
» locked to this OS thread
» unlocked
```

Memory

If you need to see the current state of memory, Go lets you get at that. You can also force a GC run or set a finalizer on something.

> Run this example a number of times to watch the `runtime.MemStats` values change.

runtime/memory.go

```go
package main

import (
	"fmt"
	"log"
	"runtime"
)

func init() {
	log.SetFlags(0)
	log.SetPrefix("» ")
}

type movie struct {
	Title string
}

func (m *movie) String() string {
	return fmt.Sprintf("Movie{%s}", m.Title)
}
```

```go

func DemoFinalizers() {
	logging := make(chan string)

	rockOfAges := &movie{"Rock of Ages"}
	runtime.SetFinalizer(rockOfAges, func(m *movie) {
		logging <- fmt.Sprintf("%s is being cleaned up", m)
		close(logging)
	})

	rockOfAges = nil
	runtime.GC() // Force a GC so the finalizer runs

	for msg := range logging {
		log.Println(msg)
	}
}

func DemoMemstats() {
	var ms runtime.MemStats
	runtime.ReadMemStats(&ms)
	log.Printf("Alloc:\t%db", ms.Alloc)
	log.Printf("TotalAlloc:\t%db", ms.TotalAlloc)
	log.Printf("Mallocs:\t%d", ms.Mallocs)
	log.Printf("Frees:\t%d", ms.Frees)
	log.Printf("PauseTotalNs:\t%dns", ms.PauseTotalNs)
}

func main() {
	DemoFinalizers()
	DemoMemstats()
}
```

Output:

```
» Movie{Rock of Ages} is being cleaned up
» Alloc:        126920b
» TotalAlloc:        131240b
» Mallocs:        108
» Frees:        20
» PauseTotalNs:        159355ns
```

Callstack

If you want to inspect the callstack, `runtime` can make that happen.

runtime/callstack.go

```go
package main

import (
	"log"
	"runtime"
)

func init() {
	log.SetFlags(0)
	log.SetPrefix("» ")
}

func PrintStack() {
	stack := make([]byte, 1024)
	i := runtime.Stack(stack, false)
	log.Printf("%s", stack[0:i])
}

func C() {
	for i := 0; i < 6; i++ {
		log.Println(runtime.Caller(i))
	}
}

func B() {
	C()
```

```
}

func A() {
	B()
}

func main() {
	PrintStack()
	A()
}
```

Output:

```
» goroutine 16 [running]:
main.PrintStack()
	/Users/darkhelmet/dev/github/darkhelmet/go-thestdlib/manuscript/code/runtime/callst\
ack.go:15 +0x76
main.main()
	/Users/darkhelmet/dev/github/darkhelmet/go-thestdlib/manuscript/code/runtime/callst\
ack.go:34 +0x1a
» 8681 /Users/darkhelmet/dev/github/darkhelmet/go-thestdlib/manuscript/code/runtime/\
callstack.go 21 true
» 9034 /Users/darkhelmet/dev/github/darkhelmet/go-thestdlib/manuscript/code/runtime/\
callstack.go 26 true
» 9066 /Users/darkhelmet/dev/github/darkhelmet/go-thestdlib/manuscript/code/runtime/\
callstack.go 30 true
» 9103 /Users/darkhelmet/dev/github/darkhelmet/go-thestdlib/manuscript/code/runtime/\
callstack.go 35 true
» 73850 /Users/darkhelmet/local/go/src/pkg/runtime/proc.c 247 true
» 84032 /Users/darkhelmet/local/go/src/pkg/runtime/proc.c 1445 true
```

runtime/debug

This package has some utility functions to make your life easier when debugging interesting things. It also has some functions to change parts of the runtime that should probably only be used when you are debugging something or if you really know what you're doing. You can change when the GC runs for example, which isn't something you normally want to mess with.

As with the other examples, this is pretty simple, and there's other things you can do, but you really need a reason to be poking around in here. It's not a package you'll be in a lot.

runtime/debug.go

```
package main

import (
	"flag"
	"log"
	"runtime/debug"
)

var (
	gcPercent = flag.Int("gc", 100, "garbage collection target percentage")
)

func init() {
	log.SetFlags(0)
	log.SetPrefix("» ")
}

func C() {
	debug.PrintStack()
}

func B() {
	C()
}

func A() {
	B()
}

func DemoGCStats() {
	var gc debug.GCStats
	debug.ReadGCStats(&gc)
	log.Printf("LastGC:\t%s", gc.LastGC)
	log.Printf("PauseTotal:\t%s", gc.PauseTotal)
	log.Printf("NumGC:\t%d", gc.NumGC)
	log.Printf("Pause:\t%s", gc.Pause)
}

```

```go
func main() {
        flag.Parse()
        debug.SetGCPercent(*gcPercent)
        A()
        DemoGCStats()
}
```

Output:

```
/Users/darkhelmet/dev/github/darkhelmet/go-thestdlib/manuscript/code/runtime/debug.g\
o:19 (0x206a)
        C: debug.PrintStack()
/Users/darkhelmet/dev/github/darkhelmet/go-thestdlib/manuscript/code/runtime/debug.g\
o:23 (0x208a)
        B: C()
/Users/darkhelmet/dev/github/darkhelmet/go-thestdlib/manuscript/code/runtime/debug.g\
o:27 (0x20aa)
        A: B()
/Users/darkhelmet/dev/github/darkhelmet/go-thestdlib/manuscript/code/runtime/debug.g\
o:42 (0x2367)
        main: A()
/Users/darkhelmet/local/go/src/pkg/runtime/proc.c:247 (0x1209a)
        main: main·main();
/Users/darkhelmet/local/go/src/pkg/runtime/proc.c:1445 (0x14860)
        goexit: runtime·goexit(void)
» LastGC:        2014-07-01 23:39:07.601611649 -0600 MDT
» PauseTotal:        189.501us
» NumGC:        3
» Pause:        [51.426us 45.349us 92.726us]
```

runtime/pprof

This package can do performance tracing, and write it out so that the `pprof` tool can read it.

> As per the docs, this is pretty much useless on OSX, so run it on Linux if you can. It's also not a very exciting program, so the profiles are similarly unexciting.

runtime/pprof.go

```go
package main

import (
	"log"
	"os"
	"runtime/pprof"
)

const Flags = os.O_CREATE | os.O_TRUNC | os.O_WRONLY

func DumpHeap(name string) {
	file, err := os.OpenFile(name, Flags, 0644)
	if err != nil {
		log.Fatalln(err)
	}
	defer file.Close()
	pprof.Lookup("heap").WriteTo(file, 0)
}

func main() {
	file, err := os.OpenFile("cpu.prof", Flags, 0644)
	if err != nil {
		log.Fatalln(err)
	}
	defer file.Close()
	err = pprof.StartCPUProfile(file)
	if err != nil {
		log.Fatalln(err)
	}
	defer pprof.StopCPUProfile()

	DumpHeap("before.heap")

	fib := []int{0, 1}
	for i := 0; i < 1000000; i++ {
		fib = append(fib, fib[i]+fib[i+1])
	}

	DumpHeap("after.heap")
}
```

sort

The `sort` package handles, you guessed it, sorting things. It can sort anything that follows the interface it defines, which is a simple 3 method interface. All it needs are:

- `Len() int` to provide the number of elements in the collection
- `Less(i, j int) bool` to return `true` if you want the element at index `i` to appear before the element at index `j`.
- `Swap(i, j int)` naturally swaps the elements at the given indexes.

Normally you'd have to define these methods yourself on your own data structures, but the `sort` package provides some helpers to sort slices of `float64`, `int`, and `string` values.

It can also easily sort in reverse order, do a stable sort,[73] and also implements a generic binary search function, as well as binary sort functions for `float64`, `int`, and `string` slices.

One last important note, is that the sorting happens in place. Copy your data if you need to preserve the original order.

Basic Sorting

We'll first look at using the builtin helpers for the 3 simple types, and then building your own data structure which implements the interface.

[73] http://en.wikipedia.org/wiki/Sorting_algorithm#Stability

sort/basic_sorting.go

```go
package main

import (
	"log"
	"sort"
)

type Question struct {
	Q, A           string
	PositionOnExam int
}

type Exam []Question

func (e Exam) Len() int {
	return len(e)
}

func (e Exam) Less(i, j int) bool {
	return e[i].PositionOnExam < e[j].PositionOnExam
}

func (e Exam) Swap(i, j int) {
	e[i], e[j] = e[j], e[i]
}

func sortInts() {
	i := []int{5, 2, 9, 8, 7}
	log.Println(i, sort.IntsAreSorted(i))
	sort.Ints(i)
	log.Println(i, sort.IntsAreSorted(i))
}

func sortStrings() {
	s := []string{"robin", "batman", "thor", "loki", "captain america"}
	log.Println(s, sort.StringsAreSorted(s))
	sort.Strings(s)
	log.Println(s, sort.StringsAreSorted(s))
}

func sortFloats() {
```

```
        f := []float64{1.5, 2.3, 0.5, 0.4}
        log.Println(f, sort.Float64sAreSorted(f))
        sort.Float64s(f)
        log.Println(f, sort.Float64sAreSorted(f))
}

func sortCustomCollection() {
        exam := Exam{
                {Q: "How much wood...", A: "A lot", PositionOnExam: 4},
                {Q: "When did WWII start?", A: "1939", PositionOnExam: 5},
                {Q: "What color is the sky?", A: "Blue", PositionOnExam: 2},
                {Q: "Who builds the iPhone?", A: "Apple", PositionOnExam: 1},
                {Q: "Why is Go awesome?", A: "Lots of reasons", PositionOnExam: 3},
        }
        log.Println(exam, sort.IsSorted(exam))
        sort.Sort(exam)
        log.Println(exam, sort.IsSorted(exam))
}

func main() {
        sortInts()
        sortStrings()
        sortFloats()
        sortCustomCollection()
}
```

Output:

```
2014/01/13 23:36:09 [5 2 9 8 7] false
2014/01/13 23:36:09 [2 5 7 8 9] true
2014/01/13 23:36:09 [robin batman thor loki captain america] false
2014/01/13 23:36:09 [batman captain america loki robin thor] true
2014/01/13 23:36:09 [1.5 2.3 0.5 0.4] false
2014/01/13 23:36:09 [0.4 0.5 1.5 2.3] true
2014/01/13 23:36:09 [{How much wood... A lot 4} {When did WWII start? 1939 5} {What \
color is the sky? Blue 2} {Who builds the iPhone? Apple 1} {Why is Go awesome? Lots \
of reasons 3}] false
2014/01/13 23:36:09 [{Who builds the iPhone? Apple 1} {What color is the sky? Blue 2\
} {Why is Go awesome? Lots of reasons 3} {How much wood... A lot 4} {When did WWII s\
tart? 1939 5}] true
```

Advanced Sorting

When I sat down to write this example, I started to go ahead with an idea I had awhile ago, then quickly realized one of the examples that comes with the Go source distribution is exactly what I wanted. So let's just use that instead.

> As per the introduction, this code is licensed differently than the other code I've written myself. That being said, this code isn't exact, I've modified it to fit my `go run file.go` style of examples.

In their example, an API is designed which does two things:

- Makes reading the code incredibly easy. It's obvious what `By(name).Sort(planets)`
- Reduces the amount of code you have to write, by requiring only the comparison `Less(i, j int) bool` function to be implemented to change the sorting behaviour.

It's a pretty slick solution, and I've seen other things floating around the community as examples, and it's essentially what I wanted to showcase.

sort/advanced_sorting.go

```go
// Copyright 2013 The Go Authors. All rights reserved.
// Use of this source code is governed by a BSD-style
// license that can be found in the LICENSE file.

package main

import (
	"log"
	"sort"
)

type earthMass float64
type au float64

type Planet struct {
	name     string
	mass     earthMass
	distance au
```

```
}

type By func(p1, p2 *Planet) bool

func (by By) Sort(planets []Planet) {
        ps := &planetSorter{
                planets: planets,
                by:      by,
        }
        sort.Sort(ps)
}

type planetSorter struct {
        planets []Planet
        by      By
}

func (s *planetSorter) Len() int {
        return len(s.planets)
}

func (s *planetSorter) Swap(i, j int) {
        s.planets[i], s.planets[j] = s.planets[j], s.planets[i]
}

func (s *planetSorter) Less(i, j int) bool {
        return s.by(&s.planets[i], &s.planets[j])
}

var planets = []Planet{
        {"Mercury", 0.055, 0.4},
        {"Venus", 0.815, 0.7},
        {"Earth", 1.0, 1.0},
        {"Mars", 0.107, 1.5},
}

func main() {
        // Closures that order the Planet structure.
        name := func(p1, p2 *Planet) bool {
                return p1.name < p2.name
        }
        mass := func(p1, p2 *Planet) bool {
                return p1.mass < p2.mass
```

```
	}
	distance := func(p1, p2 *Planet) bool {
		return p1.distance < p2.distance
	}
	decreasingDistance := func(p1, p2 *Planet) bool {
		return !distance(p1, p2)
	}

	// Sort the planets by the various criteria.
	By(name).Sort(planets)
	log.Println("By name:", planets)

	By(mass).Sort(planets)
	log.Println("By mass:", planets)

	By(distance).Sort(planets)
	log.Println("By distance:", planets)

	By(decreasingDistance).Sort(planets)
	log.Println("By decreasing distance:", planets)
}
```

Output:

```
2014/01/13 23:36:09 By name: [{Earth 1 1} {Mars 0.107 1.5} {Mercury 0.055 0.4} {Venu\
s 0.815 0.7}]
2014/01/13 23:36:09 By mass: [{Mercury 0.055 0.4} {Mars 0.107 1.5} {Venus 0.815 0.7}\
 {Earth 1 1}]
2014/01/13 23:36:09 By distance: [{Mercury 0.055 0.4} {Venus 0.815 0.7} {Earth 1 1} \
{Mars 0.107 1.5}]
2014/01/13 23:36:09 By decreasing distance: [{Mars 0.107 1.5} {Earth 1 1} {Venus 0.8\
15 0.7} {Mercury 0.055 0.4}]
```

Searching

The search API is a little weird, since you call a function that seemingly has nothing to do with the data structure you're trying to search. The key is in the second argument to the search function, which you use to dig into your data structure. The structure must already be sorted, since it uses a binary search under the covers.

It's also more than just searching in the traditional sense of "find this thing in here". It gives you the first index at which the function returns true, or `n`, its first argument, if no index returns true. You also need to follow the contract that if `f(i)` is `true`, then `f(i + 1)` is `true`. This means you can't really use `==`. In the example, I have to use `>=` which means it *finds* 11 at index 5 even though it's not in the collection.

sort/searching.go

```go
package main

import (
	"log"
	"sort"
)

func searchInts(needle int) {
	haystack := []int{1, 4, 7, 9, 10, 66}
	n := len(haystack)
	index := sort.Search(n, func(i int) bool {
		return haystack[i] >= needle
	})
	if index == n {
		log.Printf("didn't find %d", needle)
	} else {
		log.Printf("maybe found %d at index %d", needle, index)
	}
}

func main() {
	searchInts(9)
	searchInts(11)
	searchInts(70)
}
```

Output:

```
2014/01/13 23:36:09 maybe found 9 at index 3
2014/01/13 23:36:09 maybe found 11 at index 5
2014/01/13 23:36:09 didn't find 70
```

strconv

The `strconv` package gives you all the tools you need to convert between strings, integers, floats, bools. Along with these, there is a set of functions to combine the conversions and `append` to build up byte slices.

It also gives you some functions to deal with quoting strings and handling escaping things.

We'll look at the basic conversion functions first, and then quote all the things.

Conversions

The conversion functions consist of those named `FormatThing` and `ParseThing`, where `Thing` is one of `Bool`, `Float`, `Int`, and `Uint`. There are also the two oddballs `Atoi` and `Itoa`.

> I won't actually use `Atoi` and `Itoa` since they are just wrappers around `ParseInt` and `FormatInt` with sane default values for `base` and `bitSize`. We'll also skip the `uint` functions since they're the same as the `int` ones.

Because things can fail, all the parsing functions return the value and an error. If you want to live on the edge it's trivial to write a small wrapper package to either `panic` on the errors with `MustParseBool` et al, or return a default on error with `ParseBoolWithDefault` and friends.

Let's get to the code.

strconv/conversion.go

```
package main

import (
	"log"
	"strconv"
)

func init() {
```

```go
	log.SetFlags(0)
	log.SetPrefix("» ")
}

func parseBools(strings ...string) {
	for _, s := range strings {
		b, err := strconv.ParseBool(s)
		log.Printf("%t, %s", b, err)
	}
}

func printBool(bools ...bool) {
	for _, b := range bools {
		log.Println(strconv.FormatBool(b))
	}
}

func parseFloats(bitSize int, strings ...string) {
	for _, s := range strings {
		f, err := strconv.ParseFloat(s, bitSize)
		log.Printf("bitSize: %d, %#v => %f, %s", bitSize, s, f, err)
	}
}

func printFloat(f float64, fmt byte, prec, bitSize int) {
	s := strconv.FormatFloat(f, fmt, prec, bitSize)
	lfmt := "fmt: %q, prec: %2d, bitSize: %d => %s"
	log.Printf(lfmt, fmt, prec, bitSize, s)
}

var bitSizes = []int{32, 64}
var formats = []byte("efg")
var precisions = []int{5, 10, 15}

func printFloats(fs ...float64) {
	for _, f := range fs {
		for _, fmt := range formats {
			for _, prec := range precisions {
				for _, bitSize := range bitSizes {
					printFloat(f, fmt, prec, bitSize)
				}
			}
		}
```

```go
	}
}

func parseInts(base, bitSize int, ss ...string) {
	for _, s := range ss {
		i, err := strconv.ParseInt(s, base, bitSize)
		fmt := "base: %2d, bitSize: %2d, %#v => %d, %s"
		log.Printf(fmt, base, bitSize, s, i, err)
	}
}

func printInts(base int, is ...int64) {
	for _, i := range is {
		s := strconv.FormatInt(i, base)
		log.Printf("base: %2d, %d => %#v", base, i, s)
	}
}

func DemoBool() {
	log.Println("DemoBool")

	parseBools("true", "1", "f", "wat")
	printBool(true, false)
}

func DemoFloat() {
	log.Println("DemoFloat")

	parseFloats(32, "1.0", "-1.5", "1e10", "wat", "4e38")
	parseFloats(64, "4e38")

	printFloats(1.1234567, 4e38)
}

func DemoInt() {
	log.Println("DemoInt")

	big := "1010101010101010101010101010101010101010"
	parseInts(2, 32, "101101010", "10", "8", big)
	parseInts(2, 64, big)
	parseInts(8, 8, "12345", "7")
	parseInts(10, 32, "12345", "7")
	parseInts(16, 32, "abcdef")
```

```go
	// Detect base based on prefix
	parseInts(0, 32, "0xff", "0644", "255")

	printInts(2, 100)
	printInts(3, 100)
	printInts(4, 100)
	printInts(5, -100)
	printInts(10, 100)
	printInts(16, 1250)
}

func main() {
	DemoBool()
	DemoFloat()
	DemoInt()
}
```

Output:

```
» DemoBool
» true, %!s(<nil>)
» true, %!s(<nil>)
» false, %!s(<nil>)
» false, strconv.ParseBool: parsing "wat": invalid syntax
» true
» false
» DemoFloat
» bitSize: 32, "1.0" => 1.000000, %!s(<nil>)
» bitSize: 32, "-1.5" => -1.500000, %!s(<nil>)
» bitSize: 32, "1e10" => 10000000000.000000, %!s(<nil>)
» bitSize: 32, "wat" => 0.000000, strconv.ParseFloat: parsing "wat": invalid syntax
» bitSize: 32, "4e38" => +Inf, strconv.ParseFloat: parsing "4e38": value out of range
» bitSize: 64, "4e38" => 399999999999999990995239293824136118272.000000, %!s(<nil>)
» fmt: 'e', prec:  5, bitSize: 32 => 1.12346e+00
» fmt: 'e', prec:  5, bitSize: 64 => 1.12346e+00
» fmt: 'e', prec: 10, bitSize: 32 => 1.1234567165e+00
» fmt: 'e', prec: 10, bitSize: 64 => 1.1234567000e+00
» fmt: 'e', prec: 15, bitSize: 32 => 1.123456716537476e+00
» fmt: 'e', prec: 15, bitSize: 64 => 1.123456700000000e+00
» fmt: 'f', prec:  5, bitSize: 32 => 1.12346
» fmt: 'f', prec:  5, bitSize: 64 => 1.12346
» fmt: 'f', prec: 10, bitSize: 32 => 1.1234567165
» fmt: 'f', prec: 10, bitSize: 64 => 1.1234567000
```

```
» fmt: 'f', prec: 15, bitSize: 32 => 1.123456716537476
» fmt: 'f', prec: 15, bitSize: 64 => 1.123456700000000
» fmt: 'g', prec:  5, bitSize: 32 => 1.1235
» fmt: 'g', prec:  5, bitSize: 64 => 1.1235
» fmt: 'g', prec: 10, bitSize: 32 => 1.123456717
» fmt: 'g', prec: 10, bitSize: 64 => 1.1234567
» fmt: 'g', prec: 15, bitSize: 32 => 1.12345671653748
» fmt: 'g', prec: 15, bitSize: 64 => 1.1234567
» fmt: 'e', prec:  5, bitSize: 32 => +Inf
» fmt: 'e', prec:  5, bitSize: 64 => 4.00000e+38
» fmt: 'e', prec: 10, bitSize: 32 => +Inf
» fmt: 'e', prec: 10, bitSize: 64 => 4.0000000000e+38
» fmt: 'e', prec: 15, bitSize: 32 => +Inf
» fmt: 'e', prec: 15, bitSize: 64 => 4.000000000000000e+38
» fmt: 'f', prec:  5, bitSize: 32 => +Inf
» fmt: 'f', prec:  5, bitSize: 64 => 399999999999999990995239293824136118272.00000
» fmt: 'f', prec: 10, bitSize: 32 => +Inf
» fmt: 'f', prec: 10, bitSize: 64 => 399999999999999990995239293824136118272.0000000\
000
» fmt: 'f', prec: 15, bitSize: 32 => +Inf
» fmt: 'f', prec: 15, bitSize: 64 => 399999999999999990995239293824136118272.0000000\
00000000
» fmt: 'g', prec:  5, bitSize: 32 => +Inf
» fmt: 'g', prec:  5, bitSize: 64 => 4e+38
» fmt: 'g', prec: 10, bitSize: 32 => +Inf
» fmt: 'g', prec: 10, bitSize: 64 => 4e+38
» fmt: 'g', prec: 15, bitSize: 32 => +Inf
» fmt: 'g', prec: 15, bitSize: 64 => 4e+38
» DemoInt
» base:  2, bitSize: 32, "101101010" => 362, %!s(<nil>)
» base:  2, bitSize: 32, "10" => 2, %!s(<nil>)
» base:  2, bitSize: 32, "8" => 0, strconv.ParseInt: parsing "8": invalid syntax
» base:  2, bitSize: 32, "1010101010101010101010101010101010101010" => 2147483647, s\
trconv.ParseInt: parsing "1010101010101010101010101010101010101010": value out of ra\
nge
» base:  2, bitSize: 64, "1010101010101010101010101010101010101010" => 733007751850,\
 %!s(<nil>)
» base:  8, bitSize:  8, "12345" => 127, strconv.ParseInt: parsing "12345": value ou\
t of range
» base:  8, bitSize:  8, "7" => 7, %!s(<nil>)
» base: 10, bitSize: 32, "12345" => 12345, %!s(<nil>)
» base: 10, bitSize: 32, "7" => 7, %!s(<nil>)
» base: 16, bitSize: 32, "abcdef" => 11259375, %!s(<nil>)
```

```
» base:  0, bitSize: 32, "0xff" => 255, %!s(<nil>)
» base:  0, bitSize: 32, "0644" => 420, %!s(<nil>)
» base:  0, bitSize: 32, "255" => 255, %!s(<nil>)
» base:  2, 100 => "1100100"
» base:  3, 100 => "10201"
» base:  4, 100 => "1210"
» base:  5, -100 => "-400"
» base: 10, 100 => "100"
» base: 16, 1250 => "4e2"
```

Appending

The append related functions do all the same things as the formatting functions, except they append the result to a byte slice. It's sort of like a string builder, except not.

strconv/appending.go

```go
package main

import (
	"log"
	"math"
	"strconv"
)

func init() {
	log.SetFlags(0)
	log.SetPrefix("» ")
}

func main() {
	var data []byte
	data = strconv.AppendBool(data, true)
	log.Printf("%s", data)

	data = append(data, ',', ' ')
	data = strconv.AppendFloat(data, math.Pi, 'e', 2, 32)
	log.Printf("%s", data)

	data = append(data, ',', ' ')
```

```go
        data = strconv.AppendInt(data, 42, 8)
        log.Printf("%s", data)

        data = append(data, ',', ' ')
        data = strconv.AppendQuote(data, `bat"man`)
        log.Printf("%s", data)

        data = append(data, ',', ' ')
        data = strconv.AppendQuoteRune(data, 0x30f0)
        log.Printf("%s", data)

        data = append(data, ',', ' ')
        data = strconv.AppendQuoteRuneToASCII(data, 0x30f0)
        log.Printf("%s", data)

        data = append(data, ',', ' ')
        data = strconv.AppendQuoteToASCII(data, "□□")
        log.Printf("%s", data)

        data = append(data, ',', ' ')
        data = strconv.AppendUint(data, 10, 2)
        log.Printf("%s", data)
}
```

Output:

```
» true
» true, 3.14e+00
» true, 3.14e+00, 52
» true, 3.14e+00, 52, "bat\"man"
» true, 3.14e+00, 52, "bat\"man", '□'
» true, 3.14e+00, 52, "bat\"man", '□', '\u30f0'
» true, 3.14e+00, 52, "bat\"man", '□', '\u30f0', "\u30f0\u30f1"
» true, 3.14e+00, 52, "bat\"man", '□', '\u30f0', "\u30f0\u30f1", 1010
```

Quoting

Quoting is taking those things that you normally can't represent in a string like newlines and double quotes, and escaping them so that they can be represented in a double quoted string. The most obvious example is the first one. There's a multiline string literal, and we end up with single line double quoted string with the whitespace escaped. Check it out.

strconv/quoting.go

```go
package main

import (
	"log"
	"strconv"
)

func init() {
	log.SetFlags(0)
	log.SetPrefix("» ")
}

func main() {
	str := `

    "wat"

`
	log.Println(strconv.Quote(str))
	log.Println(strconv.QuoteRune(7))             // ASCII bell
	log.Println(strconv.QuoteRuneToASCII(0x30f0)) // ヰ
	log.Println(strconv.QuoteToASCII("ヰ"))
	log.Println(strconv.Unquote(`\n\r\t`)) // invalid due to lack of quotes
	log.Println(strconv.Unquote(`"\n\r\t"`))
}
```

Output:

```
» "\n\n    \"wat\"\n\n"
» '\a'
» '\u30f0'
» "\u30f0"
»  invalid syntax
» 

         <nil>
```

strings

The `strings` package deals with, you guessed it, strings.

It has quite a few functions and a couple types, so we'll groups things so that they make sense.

This package is very similar to the [`bytes`]{#bytes}

Querying strings

This covers functions like `Contains` and `HasSuffix`. Query functions give you information about the string and its contents.

strings/querying.go

```go
package main

import (
	"log"
	"strings"
)

var s = "Go, The Standard Library"

func init() {
	log.SetFlags(0)
	log.SetPrefix("» ")
}

// Look for exact matches
func DemoContains() {
	needles := []string{"Library", "standard", "Standard"}
	for _, needle := range needles {
		found := strings.Contains(s, needle)
		log.Printf("Contains(%#v) %t", needle, found)
	}
}

```

```go
// Look for any of the unicode code points
func DemoContainsAny() {
	sets := []string{"aeiou", "zyx", "\t\r"}
	for _, set := range sets {
		found := strings.ContainsAny(s, set)
		log.Printf("ContainsAny(%#v) %t", set, found)
	}
}

func DemoContainsRune() {
	runes := []rune{'a', ' ', '.'}
	for _, rune := range runes {
		found := strings.ContainsRune(s, rune)
		log.Printf("ContainsRune(%q) %t", rune, found)
	}
}

// Count substrings
func DemoCount() {
	needles := []string{"", "a", ", "}
	for _, needle := range needles {
		count := strings.Count(s, needle)
		log.Printf("Count(%#v) %d", needle, count)
	}
}

// Is it equal ignoring unicode case
func DemoEqualFold() {
	ts := []string{s, strings.ToUpper(s), strings.ToLower(s)}
	for _, t := range ts {
		equal := strings.EqualFold(s, t)
		log.Printf("EqualFold(%#v) %t", t, equal)
	}
}

// Check for prefixes
func DemoHasPrefix() {
	prefixes := []string{"Go", "GO", "Go, "}
	for _, prefix := range prefixes {
		has := strings.HasPrefix(s, prefix)
		log.Printf("HasPrefix(%#v) %t", prefix, has)
	}
}
```

```go

// Check for suffixes
func DemoHasSuffix() {
	suffixes := []string{"Library", "", "Standard"}
	for _, suffix := range suffixes {
		has := strings.HasSuffix(s, suffix)
		log.Printf("HasSuffix(%#v) %t", suffix, has)
	}
}

func main() {
	log.Printf("haystack: %#v", s)

	DemoContains()
	DemoContainsAny()
	DemoContainsRune()
	DemoCount()
	DemoEqualFold()
	DemoHasPrefix()
	DemoHasSuffix()
}
```

Output:

```
» haystack: "Go, The Standard Library"
» Contains("Library") true
» Contains("standard") false
» Contains("Standard") true
» ContainsAny("aeiou") true
» ContainsAny("zyx") true
» ContainsAny("\t\r") false
» ContainsRune('a') true
» ContainsRune(' ') true
» ContainsRune('.') false
» Count("") 25
» Count("a") 3
» Count(", ") 1
» EqualFold("Go, The Standard Library") true
» EqualFold("GO, THE STANDARD LIBRARY") true
» EqualFold("go, the standard library") true
» HasPrefix("Go") true
» HasPrefix("GO") false
» HasPrefix("Go, ") true
```

```
» HasSuffix("Library") true
» HasSuffix("") true
» HasSuffix("Standard") false
```

Into the index

All the index related functions search the string for something and return the index into the string where that thing was found. This can be referred to as needle in the haystack. The needle might be another string, a `byte`, or a function that checks an individual `rune`.

strings/index.go

```go
package main

import (
	"log"
	"strings"
	"unicode"
)

func init() {
	log.SetFlags(0)
	log.SetPrefix("» ")
}

var s = "Go, The Standard Library"

// Find specific things
func DemoIndex() {
	needles := []string{",", "t", "The", "x"}
	for _, needle := range needles {
		index := strings.Index(s, needle)
		log.Printf("Index(%#v) %d", needle, index)
	}
}

// Search for any unicode code points
func DemoIndexAny() {
	needles := []string{",thx", "ray"}
	for _, needle := range needles {
```

```go
		index := strings.IndexAny(s, needle)
		log.Printf("IndexAny(%#v) %d", needle, index)
	}
}

// Search for a specific byte
func DemoIndexByte() {
	needles := []byte{',', 'y'}
	for _, needle := range needles {
		index := strings.IndexByte(s, needle)
		log.Printf("IndexByte(%q) %d", needle, index)
	}
}

func nonAlphaNumeric(r rune) bool {
	switch {
	case 48 <= r && r <= 57: // numbers
		return false
	case 97 <= r && r <= 122: // lowercase
		return false
	case 65 <= r && r <= 90: // uppercase
		return false
	}
	return true
}

// Use a function
func DemoIndexFunc() {
	funcs := []struct {
		name string
		f    func(rune) bool
	}{
		{"nonAlphaNumeric", nonAlphaNumeric},
		{"unicode.IsLower", unicode.IsDigit},
		{"unicode.IsLower", unicode.IsLower},
	}
	for _, f := range funcs {
		index := strings.IndexFunc(s, f.f)
		log.Printf("IndexFunc(%#v) %d", f.name, index)
	}
}

// Find a specific rune
```

```go
func DemoIndexRune() {
	runes := []rune{'a', ' ', '.'}
	for _, r := range runes {
		index := strings.IndexRune(s, r)
		log.Printf("IndexRune(%q) %d", r, index)
	}
}

// Find the last index of a substring
func DemoLastIndex() {
	needles := []string{"a", "r", "y", "\t"}
	for _, needle := range needles {
		index := strings.LastIndex(s, needle)
		log.Printf("LastIndex(%#v) %d", needle, index)
	}
}

// Find the last index of any of the given unicode code points
func DemoLastIndexAny() {
	needles := []string{",thx", "ray"}
	for _, needle := range needles {
		index := strings.LastIndexAny(s, needle)
		log.Printf("LastIndexAny(%#v) %d", needle, index)
	}
}

// Use a func to find the last index of something
func DemoLastIndexFunc() {
	funcs := []struct {
		name string
		f    func(rune) bool
	}{
		{"nonAlphaNumeric", nonAlphaNumeric},
		{"unicode.IsLower", unicode.IsDigit},
		{"unicode.IsLower", unicode.IsLower},
	}
	for _, f := range funcs {
		index := strings.LastIndexFunc(s, f.f)
		log.Printf("LastIndexFunc(%#v) %d", f.name, index)
	}
}

func main() {
```

```
        log.Printf("haystack: %#v", s)

        DemoIndex()
        DemoIndexAny()
        DemoIndexByte()
        DemoIndexFunc()
        DemoIndexRune()
        DemoLastIndex()
        DemoLastIndexAny()
        DemoLastIndexFunc()
}
```

Output:

```
» haystack: "Go, The Standard Library"
» Index(",") 2
» Index("t") 9
» Index("The") 4
» Index("x") -1
» IndexAny(",thx") 2
» IndexAny("ray") 10
» IndexByte(',') 2
» IndexByte('y') 23
» IndexFunc("nonAlphaNumeric") 2
» IndexFunc("unicode.IsLower") -1
» IndexFunc("unicode.IsLower") 1
» IndexRune('a') 10
» IndexRune(' ') 3
» IndexRune('.') -1
» LastIndex("a") 21
» LastIndex("r") 22
» LastIndex("y") 23
» LastIndex("\t") -1
» LastIndexAny(",thx") 9
» LastIndexAny("ray") 23
» LastIndexFunc("nonAlphaNumeric") 16
» LastIndexFunc("unicode.IsLower") -1
» LastIndexFunc("unicode.IsLower") 23
```

Hey, split it up!

Strings getting you down, fighting all the time? Split them up! With the `Split` functions and their friends the `Fields` functions, you can take a string and chop it up.

strings/split.go

```go
package main

import (
	"log"
	"strings"
	"unicode"
)

var s = "who,what,when,where,why"

func init() {
	log.SetFlags(0)
	log.SetPrefix("» ")
}

func dump(i interface{}) {
	log.Printf("%#v", i)
}

func DemoSplit() {
	dump(strings.Split(s, ","))
	dump(strings.SplitN(s, ",", 2))
}

func DemoSplitAfter() {
	dump(strings.SplitAfter(s, ","))
	dump(strings.SplitAfterN(s, ",", 3))
}

func DemoFields() {
	fox := "   The  quick brown Fox jumps    over the lazy Dog."
	dump(strings.Fields(fox))
	dump(strings.FieldsFunc(fox, unicode.IsUpper))
}

```

```go
func main() {
	DemoSplit()
	DemoSplitAfter()
	DemoFields()
}
```

Output:

```
» []string{"who", "what", "when", "where", "why"}
» []string{"who", "what,when,where,why"}
» []string{"who,", "what,", "when,", "where,", "why"}
» []string{"who,", "what,", "when,where,why"}
» []string{"The", "quick", "brown", "Fox", "jumps", "over", "the", "lazy", "Dog."}
» []string{"   ", "he  quick brown ", "ox jumps    over the lazy ", "og."}
```

Building and altering strings

Strings are fun and all, but sometimes you need to change them. These functions can build new strings, and change the contents of existing strings.

> Okay, you can't actually change the contents of a string since strings are immutable. The functions that *change* strings actually return a new version. This is important to know because if you don't care about the previous version, you're creating garbage. You might be better off dealing with a `[]byte`, which can be altered in place, but that might not be practical either. Measure your code, and if creating garbage strings is slowing things down, then worry about optimizing.

strings/altering.go

```go
package main

import (
	"log"
	"os"
	"strings"
	"unicode"
)

func init() {
```

```
	log.SetFlags(0)
	log.SetPrefix("» ")
}

var s = "red green blue"

func DemoJoin() {
	fields := strings.Fields(s)
	log.Println(strings.Join(fields, ","))
	log.Println(strings.Join(fields, ":"))
	log.Println(strings.Join(fields, ""))
}

func rot13(r rune) rune {
	switch {
	case 65 <= r && r <= 90:
		return 65 + ((r-65)+13)%26
	case 97 <= r && r <= 122:
		return 97 + ((r-97)+13)%26
	default:
		return r
	}
}

func DemoMap() {
	log.Println(strings.Map(unicode.ToUpper, s))
	mapped := strings.Map(func(r rune) rune {
		switch r {
		case 'e':
			return -1
		default:
			return r + 1
		}
	}, s)
	log.Println(mapped)
	log.Println(strings.Map(rot13, s))
}

func DemoRepeat() {
	log.Println(strings.Repeat("-", len(s)))
}

func DemoReplace() {
```

```go
        log.Println(strings.Replace(s, "e", "!", 1))
        log.Println(strings.Replace(s, "e", "!", -1))
}

func DemoReplacer() {
        r := strings.NewReplacer("e", "E")
        log.Println(r.Replace(s))
        r.WriteString(os.Stdout, s)
}

func main() {
        DemoJoin()
        DemoMap()
        DemoRepeat()
        DemoReplace()
        DemoReplacer()
}
```

Output:

```
» red,green,blue
» red:green:blue
» redgreenblue
» RED GREEN BLUE
» se!hso!cmv
» erq terra oyhr
» --------------
» r!d green blue
» r!d gr!!n blu!
» rEd grEEn bluE
rEd grEEn bluE
```

Upper and lower case

Sometimes you just need to convert a string to upper or lower case, or maybe even title case. These next functions do exactly that.

strings/case.go

```go
package main

import (
	"log"
	"strings"
)

func init() {
	log.SetFlags(0)
	log.SetPrefix("» ")
}

var s = "The quick brown Fox jumps over the lazy Dog."

func DemoTitle() {
	log.Println(strings.Title(s))
	log.Println(strings.ToTitle(s))
}

func DemoLower() {
	log.Println(strings.ToLower(s))
}

func DemoUpper() {
	log.Println(strings.ToUpper(s))
}

func main() {
	DemoTitle()
	DemoLower()
	DemoUpper()
}
```

Output:

```
» The Quick Brown Fox Jumps Over The Lazy Dog.
» THE QUICK BROWN FOX JUMPS OVER THE LAZY DOG.
» the quick brown fox jumps over the lazy dog.
» THE QUICK BROWN FOX JUMPS OVER THE LAZY DOG.
```

Trimming

Sometimes you want to make a specific change to a string, and a very common specific change is trimming from the right or left. It's also usually whitespace you're trimming. Luckily we have functions to do all this for us.

strings/trimming.go

```go
package main

import (
	"log"
	"strings"
)

func init() {
	log.SetFlags(0)
	log.SetPrefix("» ")
}

var s = "  \n all the spaces \t  "

func dump(i interface{}) {
	log.Printf("%#v", i)
}

func DemoTrim() {
	cutset := " \t\n"
	dump(strings.Trim(s, cutset))
	dump(strings.TrimLeft(s, cutset))
	dump(strings.TrimRight(s, cutset))
	dump(strings.TrimSpace(s))
}

func DemoPrefixSuffix() {
	s2 := "The Go Programming Language"
	dump(s2)
	s2 = strings.TrimPrefix(s2, "The Go ")
	dump(s2)
	s2 = strings.TrimSuffix(s2, " Language")
	dump(s2)
}
```

```go
func onlySpaces(r rune) bool {
	return r == ' '
}

func DemoTrimFunc() {
	dump(strings.TrimFunc(s, onlySpaces))
	dump(strings.TrimLeftFunc(s, onlySpaces))
	dump(strings.TrimRightFunc(s, onlySpaces))
}

func main() {
	dump(s)
	DemoTrim()
	DemoPrefixSuffix()
	DemoTrimFunc()
}
```

Output:

```
» "  \n all the spaces \t  "
» "all the spaces"
» "all the spaces \t  "
» "  \n all the spaces"
» "all the spaces"
» "The Go Programming Language"
» "Programming Language"
» "Programming"
» "\n all the spaces \t"
» "\n all the spaces \t  "
» "  \n all the spaces \t"
```

Reader

We can also treat a string as an `io.Reader` (and other `io` interfaces). It's really easy, just make a new `strings.Reader`!

strings/reader.go

```go
package main

import (
	"log"
	"os"
	"strings"
)

var s = "All your base are belong to us!"

func init() {
	log.SetFlags(0)
	log.SetPrefix("» ")
}

func main() {
	r := strings.NewReader(s)
	log.Println(r.Len())
	r.WriteTo(os.Stdout)
	log.Println(r.Len())
	r.WriteTo(os.Stdout) // It's empty, nothing prints
	r = strings.NewReader(s)

	chunk := make([]byte, 10)
	r.Read(chunk)
	log.Printf("%s", chunk)

	r = strings.NewReader(s)
	// Read a single byte
	b, err := r.ReadByte()
	log.Println(b, err)
	log.Println(r.Len())

	// Nevermind
	r.UnreadByte()
	log.Println(r.Len())
	b, err = r.ReadByte()
	log.Println(b, err)
}
```

Output:

```
» 31
All your base are belong to us!» 0
» All your b
» 65 <nil>
» 30
» 31
» 65 <nil>
```

sync

The `sync` package is to handle all those cases where you need thread safety. Sure, we have channels and the `select` statement to deal with the builtin features that make Go so nice to use, but sometimes that's not the best way to solve the problem. Sometimes, you need the old familiar constructs to ensure thread safety.

We'll look at the tools the `sync` package provides to help solve these problems. Hopefully you can avoid these things and use higher level features, but sometimes that's not the best way to solve the problem.

Once

`Once`, as the name suggests, lets you run a function once. This is useful for setup functions that should only be ran once, but that you want to try to run multiple times to keep the code pretty and coherent.

In the example, note that there is only one log message, despite there being two calls to `once.Do`.

sync/once.go

```
package main

import (
	"log"
	"runtime"
	"sync"
)

var once sync.Once

func init() {
	log.SetFlags(0)
	log.SetPrefix("» ")
	runtime.GOMAXPROCS(8)
}

func main() {
```

```
        f := func() {
                log.Println("Hello!")
        }
        once.Do(f) // Called
        once.Do(f) // Not called
}
```

Output:

```
» Hello!
```

Mutex

`Mutex` and `RWMutex` are your basic mutual exclusion locks. You can find them in pretty much every programmung language. When using mutexes it's very important to orchestrate your unlocking, so that you don't end up with deadlocks. `defer` is helpful in this situation, though it does have a performance overhead.

The `RWMutex` is special in that you can differentiate between reading and writing. Multiple things can read, but only 1 thing can write.

This example doesn't involve any fancy goroutines, it just shows the pattern for using the `Mutex`. The usage is the basically the same each time, so remember the pattern.

sync/mutex.go

```
package main

import (
        "log"
        "runtime"
        "sync"
)

// Regular Mutex
type Lockable struct {
        m sync.Mutex
        n int
}

func (l *Lockable) Set(i int) {
```

```
	l.m.Lock()
	defer l.m.Unlock()
	l.n = i
}

func (l *Lockable) Get() int {
	l.m.Lock()
	defer l.m.Unlock()
	return l.n
}

// RWMutex
type RWLockable struct {
	m sync.RWMutex
	n int
}

func (l *RWLockable) Set(i int) {
	l.m.Lock()
	defer l.m.Unlock()
	l.n = i
}

func (l *RWLockable) Get() int {
	l.m.RLock()
	defer l.m.RUnlock()
	return l.n
}

func init() {
	log.SetFlags(0)
	log.SetPrefix("» ")
	runtime.GOMAXPROCS(8)
}

func main() {
	l := &Lockable{}
	l.Set(10)
	log.Println(l.Get())

	rwl := &RWLockable{}
	rwl.Set(5)
	log.Println(rwl.Get())
```

```
}
```

Output:

```
» 10
» 5
```

Cond

The `Cond` struct implements a condition variable. I don't know about you, but I've never actually had to use one before as far as I can recall. I've done multithreaded programming before, in a variety of languages, but I always try to keep things as simple as possible, and look for other solutions when it starts to get out of hand. It's really easy to break things in a confusing manner when you have multiple threads and shared resources flying around, so the simpler the better.

Condition variables seem to fit a certain type of problem. I'm not sure I can accurately describe that problem in words, but I can provide a couple examples.

First, we have the `io.PipeReader` and `io.PipeWriter` structs returned from `io.Pipe()`. It has to coordinate between readers and writers within the `read()` and `write()` methods.

type pipe struct

This is the basic underlying struct:

sync/pipe_struct.go

```go
// Copyright 2009 The Go Authors. All rights reserved.
// Use of this source code is governed by a BSD-style
// license that can be found in the LICENSE file.

type pipe struct {
	rl    sync.Mutex // gates readers one at a time
	wl    sync.Mutex // gates writers one at a time
	l     sync.Mutex // protects remaining fields
	data  []byte     // data remaining in pending write
	rwait sync.Cond  // waiting reader
	wwait sync.Cond  // waiting writer
	rerr  error      // if reader closed, error to give writes
```

```
	werr  error      // if writer closed, error to give reads
}
```

There are 2 mutexes, `rl` and `wl`, to ensure there is only 1 reader and 1 writer at a time. They protect access to the struct itself. The other mutex, `l`, is used on the condition variables `rwait` and `wwait`, and protects access to the other internal struct fields so that either only the reader or writer is accessing them.

`PipeReader`

When we want to read, we lock `rl` and `l`. The pattern for using condition variables is to lock, and check the conditions in a loop where you wait at the end of the loop. We've already locked `l`, so in an infinite loop we check for read and write errors, and if there is any data. If there are errors, return those. If we don't have data yet, we `rwait.Wait()`. When we are woken up by a `rwait.Signal()`, we check everything again. If we have data, we can do the read. In this case it's just copying data from one slice to another, and shortening the buffer (that's not really a buffer) so reflect that we've read data from it. When we have read all the data, we clear out the internal data and call `wwait.Signal()` to tell the writer it can continue.

sync/pipe_reader.go

```go
// Copyright 2009 The Go Authors. All rights reserved.
// Use of this source code is governed by a BSD-style
// license that can be found in the LICENSE file.

func (p *pipe) read(b []byte) (n int, err error) {
	// One reader at a time.
	p.rl.Lock()
	defer p.rl.Unlock()

	p.l.Lock()
	defer p.l.Unlock()
	for {
		if p.rerr != nil {
			return 0, ErrClosedPipe
		}
		if p.data != nil {
			break
		}
		if p.werr != nil {
			return 0, p.werr
```

```
                }
                p.rwait.Wait()
        }
        n = copy(b, p.data)
        p.data = p.data[n:]
        if len(p.data) == 0 {
                p.data = nil
                p.wwait.Signal()
        }
        return
}
```

PipeWriter

When we want to write, we lock `wl` and `l`. If we have a `werr`, return that and do nothing. Otherwise, we save the data and `rwait.Signal()` to let the reader know they can wake up and read data. Now the writer can go into its loop to check and wait. If the data is `nil` (because the reader read everything and cleared it), everything is fine, and `break`. If we got a `rerr`, make sure to return that, and `break`. If we have a `werr`, make sure we return the `ErrClosedPipe`. If we've broken out of the loop, we make sure to return a sane `n` value, and clear out `data`.

sync/pipe_writer.go

```
// Copyright 2009 The Go Authors. All rights reserved.
// Use of this source code is governed by a BSD-style
// license that can be found in the LICENSE file.

var zero [0]byte

func (p *pipe) write(b []byte) (n int, err error) {
        // pipe uses nil to mean not available
        if b == nil {
                b = zero[:]
        }

        // One writer at a time.
        p.wl.Lock()
        defer p.wl.Unlock()

        p.l.Lock()
        defer p.l.Unlock()
```

```
        if p.werr != nil {
                err = ErrClosedPipe
                return
        }
        p.data = b
        p.rwait.Signal()
        for {
                if p.data == nil {
                        break
                }
                if p.rerr != nil {
                        err = p.rerr
                        break
                }
                if p.werr != nil {
                        err = ErrClosedPipe
                }
                p.wwait.Wait()
        }
        n = len(b) - len(p.data)
        p.data = nil // in case of rerr or werr
        return
}
```

If you follow the calls to `rwait.Signal()`, `rwait.Wait()`, `wwait.Signal()`, and `wwait.Wait()`, you can trace the program flow and see that it allows both a read and write to start, in either order, but the write obviously has to produce data before the read can read it, and the read returns before the write returns.

Cache

Another example which was given on Stack Overflow, was that of a cache: http://stackoverflow.co The example was psuedo code, and I've implemented it here as best I can. I think it's correct. At least the race detector doesn't complain complain.

I took it a little farther and used a `RWMutex` to protect the main cache, so you can have mulitple readers, and each key is protected individually as well, so getting one key doesn't block getting another.

[74] http://stackoverflow.com/a/2476820/4657

sync/cond.go

```go
package main

import (
	"fmt"
	"log"
	"runtime"
	"sync"
	"time"
)

func init() {
	runtime.GOMAXPROCS(8)
}

type Status int

const (
	Absent Status = iota
	InProgress
	Complete
)

func getData(key string) []byte {
	// Do some work
	time.Sleep(3 * time.Second)
	return []byte(fmt.Sprintf("getData: %v", key))
}

type CacheEntry struct {
	sync.Mutex
	C      *sync.Cond
	Status Status
	Data   []byte
}

func (ce *CacheEntry) SetComplete(data []byte) {
	ce.Lock()
	defer ce.Unlock()
	ce.Data = data
	ce.Status = Complete
}
```

```go

func (ce *CacheEntry) Wait() []byte {
	ce.Lock()
	defer ce.Unlock()
	for {
		if ce.Status == Complete {
			break
		}
		ce.C.Wait()
	}
	return ce.Data
}

type Cache struct {
	sync.RWMutex
	statuses map[string]Status
	data     map[string]*CacheEntry
}

func NewCache() *Cache {
	return &Cache{
		statuses: make(map[string]Status),
		data:     make(map[string]*CacheEntry),
	}
}

func (c *Cache) setComplete(key string) {
	c.Lock()
	defer c.Unlock()
	c.statuses[key] = Complete
}

func (c *Cache) setInProgress(key string) (*CacheEntry, bool) {
	c.Lock()
	defer c.Unlock()

	// Check again, maybe another thread go to this first
	// in between the c.RUnlock() and c.Lock()
	if c.statuses[key] != Absent {
		return c.data[key], false
	}

	c.statuses[key] = InProgress
```

```go
        entry := &CacheEntry{Status: InProgress}
        entry.C = sync.NewCond(entry)
        c.data[key] = entry
        return entry, true
}

func (c *Cache) Get(key string) []byte {
        c.RLock()

        status := c.statuses[key]
        switch status {
        case Absent:
                c.RUnlock() // We'll take a write lock right away.

                entry, ok := c.setInProgress(key)
                if !ok {
                        // Missed our chance, just wait.
                        return entry.Wait()
                }

                data := getData(key)
                entry.SetComplete(data)
                c.setComplete(key)

                // Wake up everybody, not just a single goroutine
                entry.C.Broadcast()

                return data
        case InProgress:
                entry := c.data[key]
                c.RUnlock()
                return entry.Wait()
        case Complete:
                entry := c.data[key]
                c.RUnlock()
                return entry.Data
        }
        panic("not reached")
}

func main() {
        log.Println("starting")

```

```go
	c := NewCache()
	var wg sync.WaitGroup
	wg.Add(5)
	for i := 0; i < 5; i++ {
		go func() {
			log.Printf("%s", c.Get("Batman"))
			log.Printf("%s", c.Get("Robin"))
			wg.Done()
		}()
	}
	wg.Wait()
	// These print right away, already in cache.
	log.Printf("%s", c.Get("Batman"))
	log.Printf("%s", c.Get("Robin"))

	wg.Add(5)
	for i := 0; i < 5; i++ {
		go func() {
			log.Printf("%s", c.Get("Captain America"))
			log.Printf("%s", c.Get("Thor"))
			wg.Done()
		}()
	}
	time.Sleep(time.Second)
	// These print right away, already in cache, not blocked by
	// other goroutines trying to read "captain america" and "thor"
	log.Printf("%s", c.Get("Batman"))
	log.Printf("%s", c.Get("Robin"))
	wg.Wait()
}
```

Output:

```
2014/09/13 21:54:32 starting
2014/09/13 21:54:35 getData: Batman
2014/09/13 21:54:35 getData: Batman
2014/09/13 21:54:35 getData: Batman
2014/09/13 21:54:35 getData: Batman
2014/09/13 21:54:35 getData: Batman
2014/09/13 21:54:38 getData: Robin
2014/09/13 21:54:38 getData: Robin
2014/09/13 21:54:38 getData: Robin
2014/09/13 21:54:38 getData: Robin
```

```
2014/09/13 21:54:38 getData: Robin
2014/09/13 21:54:38 getData: Batman
2014/09/13 21:54:38 getData: Robin
2014/09/13 21:54:39 getData: Batman
2014/09/13 21:54:39 getData: Robin
2014/09/13 21:54:41 getData: Captain America
2014/09/13 21:54:41 getData: Captain America
2014/09/13 21:54:41 getData: Captain America
2014/09/13 21:54:41 getData: Captain America
2014/09/13 21:54:41 getData: Captain America
2014/09/13 21:54:44 getData: Thor
2014/09/13 21:54:44 getData: Thor
2014/09/13 21:54:44 getData: Thor
2014/09/13 21:54:44 getData: Thor
2014/09/13 21:54:44 getData: Thor
```

What's important is the timing in the output. It starts, and after 3 seconds you get the 5 lines of Batman. The 3 seconds is from the `time.Sleep(3 * time.Second)` in the `getData(key)` function. They all print because when the first goroutine finishes, it broadcasts to the other goroutines, and they all wake up and can return the data. Then another 3 seconds pass and we get 5 lines of `Robin`. Then no seconds pass and we get another 2 lines of `Batman` and `Robin`. This is because they are already in the cache at that point, so there's no waiting.

Then we start another batch of 5 calls to `Get` with new keys that are not in the cache. We sleep for a second and again `Get Batman` and `Robin`, which happen immediately because they aren't blocked by the calls getting `Captain America` and `Thor`. Another 2 seconds pass, and we get our 5 lines of `Captain America` and 5 lines of `Thor`.

WaitGroup

A `WaitGroup` is used to wait until an expected number of things finish. This is useful when you aren't using channels and therefore don't have a channel to close.

The example seems trivial, but I find when I run into a problem where a `WaitGroup` would work well, it's fairly obvious.

sync/wait_group.go

```go
package main

import (
	"log"
	"runtime"
	"sync"
	"time"
)

var n = 5

func init() {
	log.SetFlags(0)
	log.SetPrefix("» ")
	runtime.GOMAXPROCS(8)
}

func Run(id int, wg *sync.WaitGroup) {
	for i := 0; i < n; i++ {
		time.Sleep(time.Second)
		wg.Done()
		log.Printf("%d is done", id)
	}
}

func main() {
	var wg sync.WaitGroup
	for i := 0; i < 3; i++ {
		wg.Add(n)
		go Run(i, &wg)
	}
	wg.Wait()
	log.Println("all done")
}
```

Output:

```
» 2 is done
» 0 is done
» 1 is done
» 2 is done
» 0 is done
» 1 is done
» 2 is done
» 0 is done
» 1 is done
» 2 is done
» 0 is done
» 1 is done
» 2 is done
» 0 is done
» 1 is done
» all done
```

Pool

A `Pool` is used to to prevent GC thrash by allowing you to reused allocated objects. The idea is that you have a bunch of goroutines that are all doing the same thing, and hence will all need the same type of data structure.

Maybe you run an animated gif website, and you want to know the dimensions of all the images you host. Since the size information is at the beginning of the file, you only need to read in a few bytes to find out what you want to know. Using a `Pool`, you can allocate and reused your `[]byte` instead of `make`ing a new one each time. If you're reading a few images, maybe this doesn't matter. If you're reading millions, it probably will.

In this example, the `New` function we create the `Pool` with just returns a struct that includes an `int` that increases on every call to `New`. Even though there are 20 calls to `pool.Get()`, the number only goes up to about 4 or 5. This is because at any given time there should only be a maximum of 4 `Thing`s out in the wild, and they get put back, and hence reused.

sync/pool.go

```go
package main

import (
	"log"
	"runtime"
	"sync"
	"time"
)

var n int

type Thing struct {
	N int
}

func init() {
	log.SetFlags(0)
	log.SetPrefix("» ")
	runtime.GOMAXPROCS(8)
}

func Run(pool *sync.Pool) {
	for i := 0; i < 5; i++ {
		thing := pool.Get().(*Thing)
		log.Println(thing.N)
		pool.Put(thing)
	}
}

func main() {
	pool := &sync.Pool{
		New: func() interface{} {
			n += 1
			return &Thing{n}
		},
	}

	go Run(pool)
	go Run(pool)
	go Run(pool)
	go Run(pool)
```

```go
	go Run(pool)

	time.Sleep(time.Second)
}
```

Output:

```
» 3
» 5
» 5
» 5
» 5
» 5
» 3
» 3
» 3
» 3
» 1
» 5
» 4
» 4
» 3
» 3
» 3
» 3
» 4
» 5
» 3
» 3
» 3
» 5
» 3
```

sync/atomic

The `sync/atomic` package contains a whole mess of functions to do atomic operations with integers. You can add a delta to them, swap them, compare and swap, load and store. These are mostly low level primitives, but sometimes they're just what the doctor ordered.

In the example, the code is the same, except for the single line that modifies `n`. Using `n++` doesn't result in the correct value because it reads old values. Using `atomic.AddInt32` gives the correct answer.

sync/atomic.go

```
package main

import (
	"log"
	"runtime"
	"sync"
	"sync/atomic"
)

var (
	expected int32 = 1000 * 1000
)

func init() {
	log.SetFlags(0)
	log.SetPrefix("» ")
	runtime.GOMAXPROCS(8)
}

func DemoBroken() {
	var n int32
	var wg sync.WaitGroup
	wg.Add(1000)
	for i := 0; i < 1000; i++ {
		go func() {
			for j := 0; j < 1000; j++ {
				n++
			}
			wg.Done()
		}()
	}
	wg.Wait()
	log.Printf("got %d, expected %d", n, expected)
}

func DemoAtomic() {
	var n int32
	var wg sync.WaitGroup
```

```go
	wg.Add(1000)
	for i := 0; i < 1000; i++ {
		go func() {
			for j := 0; j < 1000; j++ {
				atomic.AddInt32(&n, 1)
			}
			wg.Done()
		}()
	}
	wg.Wait()
	log.Printf("got %d, expected %d", n, expected)
}

func main() {
	DemoBroken()
	DemoAtomic()
}
```

Output:

```
» got 378537, expected 1000000
» got 1000000, expected 1000000
```

testing

The `testing` package contains functions and structures useful when testing Go applications and libraries. You don't normally need these things when writing your application or library (unless you're writing something to interact or help with testing), but they are your world when you're writing tests for your library or application.

The basic way to test your go code is to start out with a `*_test.go` file. Say you have a math library, `bookmath`, and you have `math_int.go` to handle doing math with `int`s. It has a function `SumInts`. You write tests for that file in `math_int_test.go`. Now you can run `go test` and go will run your tests. Great!

Now, write a function `TestSumInts`, or whatever, as long as it starts with `Test`. This function takes a `*testing.T` argument, and you're off to the races!

testing.T

The main thing you interact with is this `testing.T` type. There are a bunch of methods hanging off of it, but they all revolve around logging things, failing the current test, or skipping the current test. There is actually no built in `assert` like you'd see in many other testing libraries. If you want to assert something, you can use an `if` statement. The `testing` package is about having a toolbox of very basic tools, and building from those. This is one of the places in the standard library I like using an external library to layer onto the testing functionality to make things a bit smoother, but it's fine if you don't.

Let's look at an example.

testing/src/bookmath/math_int.go

```go
package bookmath

// SumInts adds up a bunch of ints
func SumInts(values ...int) (sum int64) {
	for _, value := range values {
		sum += int64(value)
	}
	return sum
}
```

testing/src/bookmath/math_int_test.go

```go
package bookmath_test

import (
	"bookmath"
	"testing"
)

func TestSumInts(t *testing.T) {
	tests := []struct {
		values   []int
		expected int64
	}{
		{[]int{1, 2, 3}, 6},
		{[]int{1, -1, 0}, 0},
	}

	for _, testCase := range tests {
		sum := bookmath.SumInts(testCase.values...)
		if sum != testCase.expected {
			t.Error("SumInts(%v), expected=%d, actual=%d", testCase.values, testCase.exp
, sum)
		}
	}
}
```

> You have to run these with a bit more finesse because we're outside `GOPATH`. When you're in the `testing` directory, run `GOPATH="$PWD":$GOPATH go test ./...` and it'll do the right thing.

Now we have some super simple output that looks like this:

```
ok bookmath 0.005s
```

That's basically all you need for testing go things. It's just simple programming. Nothing fancy to learn. There are more fun things we can do, so let's check them out.

Benchmarking

You'll maybe want to benchmark your code, so that when you run tests you can spot regressions in performance. The go team does this all the time, and naturally it's built into the `testing` package.

To write benchmark tests, you want to write functions prefixed with `Benchmark`. Then you can run `go test -bench .` and it'll go to town.

testing/src/bookmath/math_int_benchmark_test.go

```go
package bookmath_test

import (
	"bookmath"
	"testing"
)

func BenchmarkSumInts(b *testing.B) {
	for i := 0; i < b.N; i++ {
		bookmath.SumInts(1, 2, 3, 4, 5, 6, 7, 8, 9)
	}
}
```

These use a different struct, `testing.B`. There's an `N` attribute on it which has the number of times you should call your function, so naturally, we use a `for` loop.

Now we see an output like this:

```
BenchmarkSumInts-8 200000000 9.76 ns/op
```

So out `SumInts` function ran pretty fast.

Examples

You can also write examples in the tests. These will fail the tests if the output doesn't match what you said the output should be. They are kind of like unit tests that serve as, well, examples for other people when they need to figure out how to use your code. Sometimes looking at test code isn't very useful, and examples can help with that.

testing/src/bookmath/math_int_example_test.go

```go
package bookmath_test

import (
	"bookmath"
	"fmt"
)

func ExampleSumInts() {
	fmt.Println(bookmath.SumInts(1, 2, 3, 4, 5))
	// Output: 15
}
```

Just prefix a function with `Example`. These take no arguments, and should output to `STDOUT` using `fmt.Println`. Then, under your `fmt.Println(...)`, write a comment showing what the output should be: `// Output: <thing>`. In our case, it's `// Output: 55`

If the example fails, you'll see something like this:

```
--- FAIL: ExampleSumInts (0.00s) got: 15 want: 1 FAIL
```

text

The `text` package doesn't do anything other than hold other packages. There are things for scanning text, which is basically reading the "pieces" of it (so you can build compilers and stuff). There are things to write text, specifically tabbed column output, which is pretty cool. We also have a generic version of the `html/template` package, which lets us build and evaluate arbitrary templates.

So naturally, let's build a compiler first.

Let's build a calculator

Let's look at the `text/scanner` package first. We can use it to parse an expression for a Reverse Polish Notation calculator, and then evaluate the expression. We can build our own HP calculator.

With our RPN calculator, we'll have something like `1 1 +`. This basically says, put 1 on the stack, put 1 on the stack, pop 2 things off the stack and add them, and put that on the stack. Then we can print the last thing on the stack as our answer.

This code is actually mostly not related to `text/scanner`, but that's the beauty of it. You don't have to write a bunch of junk to handle reading and parsing the text, you can concentrate on the actual problem at hand. Try adding `sqrt` to this example, or adding custom values like `pi` and `e`.

text/calculator.go

```go
package main

import (
	"flag"
	"log"
	"regexp"
	"strconv"
	"strings"
	"text/scanner"
)

var (
```

```
	equation string
	numberRe = regexp.MustCompile(`-?[1-9][0-9]*(\.[0-9]+)?`)
)

func fn(num float64) string {
	return strconv.FormatFloat(num, 'f', -1, 64)
}

func show(l, r, value float64, operand string) {
	log.Printf("pushing %s %s %s => %s", fn(l), operand, fn(r), fn(value))
}

type Stack struct {
	data []string
}

func (s Stack) IsEmpty() bool {
	return len(s.data) == 0
}

func (s *Stack) Push(value string) {
	s.data = append(s.data, value)
}

func (s *Stack) PushNumber(num float64) {
	s.Push(fn(num))
}

func (s *Stack) Pop() string {
	if s.IsEmpty() {
		return ""
	}
	value, data := s.data[len(s.data)-1], s.data[:len(s.data)-1]
	s.data = data
	return value
}

func (s *Stack) PopNumber() float64 {
	value := s.Pop()
	num, err := strconv.ParseFloat(value, 64)
	if err != nil {
		log.Fatalf("failed parsing number: %s", err)
	}
```

```
        return num
}

func (s *Stack) PopOperands() (float64, float64) {
        r, l := s.PopNumber(), s.PopNumber()
        return l, r
}

func init() {
        log.SetFlags(0)
        log.SetPrefix("» ")

        flag.StringVar(&equation, "rpn", "1 2 + 3 * 2 / 10 -", "the equation to evaluate")
        flag.Parse()
}

func main() {
        var s scanner.Scanner
        s.Filename = "equation"
        s.Init(strings.NewReader(equation))

        stack := Stack{}
        for {
                // Using Scan() we skip whitespace
                tok := s.Scan()
                if tok == scanner.EOF {
                        break
                }
                text := s.TokenText()
                switch tok {
                case '+':
                        l, r := stack.PopOperands()
                        value := l + r
                        show(l, r, value, "+")
                        stack.PushNumber(value)
                case '-':
                        l, r := stack.PopOperands()
                        value := l - r
                        show(l, r, value, "-")
                        stack.PushNumber(value)
                case '*':
                        l, r := stack.PopOperands()
                        value := l * r
```

```
			show(l, r, value, "*")
			stack.PushNumber(value)
		case '/':
			l, r := stack.PopOperands()
			value := l / r
			show(l, r, value, "/")
			stack.PushNumber(value)
		default:
			switch {
			case numberRe.MatchString(text):
				log.Printf("pushing %s", text)
				stack.Push(text)
			}
		}
	}
	log.Printf("=> %s", stack.Pop())
}
```

Output:

```
» pushing 1
» pushing 2
» pushing 1 + 2 => 3
» pushing 3
» pushing 3 * 3 => 9
» pushing 2
» pushing 9 / 2 => 4.5
» pushing 10
» pushing 4.5 - 10 => -5.5
» => -5.5
```

Pretty console output

Something else the `text` package lets us do is write pretty tab separated columns so we can output data in tables. Like most great things in the Go standard library, this works with `io.Writer`, so we can basically write to anything. We'll be writing to `os.Stdout` in our example.

It's a pretty straightforward package, and the example is short, but it's useful.

text/tabwriter.go

```go
package main

import (
	"os"
	"strings"
	"text/tabwriter"
)

func main() {
	data := [][]string{
		{"Continent", "Country", "Nationality"},
		{"North America", "Canada", "Canadian"},
		{"Europe", "France", "French"},
	}

	writer := tabwriter.NewWriter(os.Stdout, 0, 8, 4, ' ', 0)
	defer writer.Flush() // Make sure to Flush the writer when you're done

	for _, tuple := range data {
		writer.Write([]byte(strings.Join(tuple, "\t")))
		writer.Write([]byte{'\n'})
	}
}
```

Output:

```
Continent        Country     Nationality
North America    Canada      Canadian
Europe           France      French
```

Templating

Finally, the `text` package lets us make arbitrary templates and evaluate those templates given a context. If you've used Ruby, this is basically like ERB. While you normally see ERB used to generate HTML, it just generates a text file given some other ruby code, and this is really no different.

text/template.go

```go
package main

import (
	"html/template"
	"os"
)

var (
	todoItems = []string{
		"cut the grass",
		"pick up milk",
		"feed the dog",
	}
)

func main() {
	t := template.Must(template.New("todos").Parse(`TODO:
{{ range $index, $item := . }}
{{ $index }}: {{ . }}{{ end }}
`))

	t.Execute(os.Stdout, todoItems)
}
```

Output:

```
TODO:

0: cut the grass
1: pick up milk
2: feed the dog
```

In that example, we iterate over our TODO items and make a list. When we have `{{ range $index, $item := . }}`, it's saying:

> Iterate over the current thing, and assign me an index and the item. Also, if you could start your range variables with a `$`, that'd be just great...

That `.` on the right side of `:=` is what you're iterating over, which is the current thing. Since at that point it's at the top level of the context, and we passed in our slice of TODO items, `.` is the slice of TODO items.

In the range body, we can output `{{ . }}`, and because we're in a range body, `.` is the element we're iterating over. The Go templates assume you want to iterate over the things in the slice. In normal go if you did `thing := range things`, `thing` would be the index, but in the templates so you simply `{{ range . }}` and `.` inside the range block would be the element, and not the index.

Functions in templates

The output is less than ideal, since it starts numbering at 0. Go doesn't allow completely arbitrary code in the template tags, so we can't just `$index + 1`. We need to write a function and add that to the template as a `FuncMap`.

text/template_funcs.go

```go
package main

import (
	"html/template"
	"os"
)

var (
	todoItems = []string{
		"cut the grass",
		"pick up milk",
		"feed the dog",
	}
)

func main() {
	tmpl := template.New("todos")
	tmpl.Funcs(map[string]interface{}{
		"inc": func(a, b int) int {
			return a + b
		},
	})
	t := template.Must(tmpl.Parse(`TODO:
{{ range $index, $item := . }}
{{ inc $index 1 }}: {{ $item }}{{ end }}
`))

	t.Execute(os.Stdout, todoItems)
}
```

Output:

```
TODO:

1: cut the grass
2: pick up milk
3: feed the dog
```

We define a function that take 2 integers and adds them together and returns the result. Now we can call the func in the template as `{{ inc $index 1 }}`. We don't have to use parens or commas, it works fine like that.

Notice we have to call the `Funcs` method before we parse the template. These templates give you all the glorious benefits of types Go has to offer, so if you try to use a function in your template you haven't defined, Go throws an error compiling the template.

Inline templates

Sometimes you want to define a quick template inline in the event you need to use it in multiple places. We don't really need to, but we can change our TODO example to use an inline template to render the TODO item.

text/template_inline.go

```go
package main

import (
	"html/template"
	"os"
)

var (
	todoItems = []string{
		"cut the grass",
		"pick up milk",
		"feed the dog",
	}
)

func main() {
	tmpl := template.New("todos")
	t := template.Must(tmpl.Parse(`{{ define "todo" }}- {{ . }}{{ end }}TODO:
```

```
{{ range $index, $item := . }}
{{ template "todo" $item }}{{ end }}
`))

        t.Execute(os.Stdout, todoItems)
}
```

Output:

```
TODO:

- cut the grass
- pick up milk
- feed the dog
```

Template files

In any normal application, you'll probably have the templates in separate files, and we can use those just fine. You can write out a number templates, load them all, and execute the one you want.

text/header.tmpl

```
You have {{ len . }} TODOs today:
```

<<text/todo.tmpl[75]

text/todos.tmpl

```
{{ template "header.tmpl" . }}
{{ range $index, $item := . }}
{{ template "item.tmpl" $item }}{{ end }}
```

[75] code/text/todo.tmpl

text/template_files.go

```go
package main

import (
	"html/template"
	"log"
	"os"
)

var (
	todoItems = []string{
		"cut the grass",
		"pick up milk",
		"feed the dog",
	}
)

func main() {
	t := template.Must(template.ParseGlob("*.tmpl"))
	err := t.ExecuteTemplate(os.Stdout, "todos.tmpl", todoItems)
	if err != nil {
		log.Fatalf("failed executing template: %s", err)
	}
}
```

Output:

```
You have 3 TODOs today:

- cut the grass
- pick up milk
- feed the dog
```

time

The `time` package, if you can believe it, deals with time. You can parse time, format a time to a string, compare times, and add and subtract times. It will also deal with timezone stuff.

You can also create timers and tickers for handling timeouts and sleeping.

There are a few main types in the `time` package: `time.Time` is the main type that represents a point in time. `time.Duration` represents a change in time, like 4 minutes. `time.Location` is where the timezone support comes from. `time.Ticker` will *tick* on a channel, and `time.Timer`, which sends the current time on a channel after the specified `Duration`.

Let's play around. I think I'll skip some of the really basic methods, like `func (t Time) Minute() int`. I think you can figure out what those do.

Parsing and Formatting

Parsing time and formatting it back to a `string` is one of the basic and most common tasks you'll do with time. Everything you need to know is listed in the package docs in the first constant section. It's a bit of a non-standard way to represent the string you are to parse in, but it's more readable than the `%` stuff you're used to dealing with.

While you can build your own layouts, if you're moving times between systems, you're probably better off using any of the preset constants that come in the time package. RFC822 and RFC3339 are a couple that come to mind. If you're displaying times, then you probably want to build your own.

Just remember to handle parsing errors, and not ignore them like I did.

time/parsing_formatting.go

```go
package main

import (
	"log"
	"time"
)

var (
	layouts = []string{
		time.RFC822,
		time.RFC3339,
		time.Kitchen,
		time.RubyDate,
		"2006-01-_2", // _ to not display leading zeroes
	}
	times = make(chan string, len(layouts))
)

func init() {
	log.SetFlags(0)
	log.SetPrefix("» ")
}

func DemoFormat() {
	now := time.Now()
	for _, layout := range layouts {
		formatted := now.Format(layout)
		times <- formatted
		log.Printf("%s + %#v = %#v", now, layout, formatted)
	}
	close(times)
}

func DemoParse() {
	for _, layout := range layouts {
		t := <-times
		parsed, _ := time.Parse(layout, t)
		log.Printf("%#v + %#v = %s", t, layout, parsed)
	}
}
```

```
func main() {
	DemoFormat()
	DemoParse()
}
```

Output:

```
» 2014-08-13 21:49:39.694096285 -0600 MDT + "02 Jan 06 15:04 MST" = "13 Aug 14 21:49\
 MDT"
» 2014-08-13 21:49:39.694096285 -0600 MDT + "2006-01-02T15:04:05Z07:00" = "2014-08-1\
3T21:49:39-06:00"
» 2014-08-13 21:49:39.694096285 -0600 MDT + "3:04PM" = "9:49PM"
» 2014-08-13 21:49:39.694096285 -0600 MDT + "Mon Jan 02 15:04:05 -0700 2006" = "Wed \
Aug 13 21:49:39 -0600 2014"
» 2014-08-13 21:49:39.694096285 -0600 MDT + "2006-01-_2" = "2014-08-13"
» "13 Aug 14 21:49 MDT" + "02 Jan 06 15:04 MST" = 2014-08-13 21:49:00 -0600 MDT
» "2014-08-13T21:49:39-06:00" + "2006-01-02T15:04:05Z07:00" = 2014-08-13 21:49:39 -0\
600 MDT
» "9:49PM" + "3:04PM" = 0000-01-01 21:49:00 +0000 UTC
» "Wed Aug 13 21:49:39 -0600 2014" + "Mon Jan 02 15:04:05 -0700 2006" = 2014-08-13 2\
1:49:39 -0600 MDT
» "2014-08-13" + "2006-01-_2" = 2014-08-13 00:00:00 +0000 UTC
```

Duration

Duration is something like `5h` or `2m30s`. You can add durations to a time to get a new time. You can also use durations to sleep or wait a certain amount of time. They are pretty straightforward to use, and the `flag` package can even parse them without any fuss. You can even round a time using the duration constants.

> Along with `Round` there is also `Truncate`. The former is for, uh, rounding, and the latter is like the mathematical `floor` function, forcing the time to round down.

The easiest way to get a duration from a constant is to multiply the constant by the unit you want from the `time` package.

It goes up to hours, because anything past that gets really scary when you have to deal with timezones, daylight savings time, and the fact that a day isn't really 24 hours. Okay, that last one probably isn't that big of a deal, but it's interesting to think about.

time/duration.go

```go
package main

import (
	"log"
	"time"
)

var (
	moon = time.Date(1969, time.July, 20, 20, 18, 4, 0, time.UTC)
)

func init() {
	log.SetFlags(0)
	log.SetPrefix("» ")
}

func DemoConstants() {
	log.Println("DemoConstants")
	log.Println(5 * time.Nanosecond)
	log.Println(5 * time.Microsecond)
	log.Println(5 * time.Millisecond)
	log.Println(5 * time.Second)
	log.Println(5 * time.Minute)
	log.Println(5 * time.Hour)
}

func DemoParsing() {
	log.Println("DemoParsing")
	d, _ := time.ParseDuration("5h2m55s10us5ns")
	log.Println(d)
	log.Printf("%fh == %fm == %fs", d.Hours(), d.Minutes(), d.Seconds())
}

func DemoRound() {
	log.Println("DemoRound")
	log.Println(moon)
	log.Println(moon.Round(time.Minute))
	log.Println(moon.Round(time.Hour))
}
```

```go

func DemoTruncate() {
	// Ignore this math until the next demo
	laterMoon := moon.Add(30 * time.Minute)
	log.Println("DemoTruncate")
	log.Println(laterMoon)
	log.Println(laterMoon.Truncate(time.Hour))
	// See how Round goes up and Truncate goes down?
	log.Println(laterMoon.Round(time.Hour))
}

func DemoSince() {
	log.Println("DemoSince")
	log.Printf("%s since %s", time.Since(moon), moon)
}

func main() {
	DemoConstants()
	DemoParsing()
	DemoRound()
	DemoTruncate()
	DemoSince()
}
```

Output:

```
» DemoConstants
» 5ns
» 5us
» 5ms
» 5s
» 5m0s
» 5h0m0s
» DemoParsing
» 5h2m55.000010005s
» 5.048611h == 302.916667m == 18175.000010s
» DemoRound
» 1969-07-20 20:18:04 +0000 UTC
» 1969-07-20 20:18:00 +0000 UTC
» 1969-07-20 20:00:00 +0000 UTC
» DemoTruncate
» 1969-07-20 20:48:04 +0000 UTC
» 1969-07-20 20:00:00 +0000 UTC
```

```
» 1969-07-20 21:00:00 +0000 UTC
» DemoSince
» 395189h47m26.171405848s since 1969-07-20 20:18:04 +0000 UTC
```

Math

Doing math on time is pretty straightforward. Sort of. You can:

- `Add` a `Duration` to get a new `Time`.
- `Sub` a `Time` to get a `Duration`.
- `AddDate(years, months, days)` to get a `Time`.

time/math.go

```go
package main

import (
	"log"
	"time"
)

var (
	moon = time.Date(1969, time.July, 20, 20, 18, 4, 0, time.UTC)
	now  = time.Now()
)

func init() {
	log.SetFlags(0)
	log.SetPrefix("» ")
}

func DemoAdd() {
	log.Println("DemoAdd")
	log.Println(moon.Add(4 * time.Hour))

	log.Println(now)
	// 24 hours from now
	log.Println(now.Add(24 * time.Hour))
	// 24 hours ago, you can add a negative duration
	log.Println(now.Add(-24 * time.Hour))
```

```
}

func DemoSub() {
        log.Println("DemoSub")
        log.Println(moon.Sub(time.Now()))
}

func DemoAddDate() {
        log.Println("DemoAddDate")
        log.Println(moon.AddDate(45, 0, 0))
}

func main() {
        log.Println(moon)
        DemoAdd()
        DemoSub()
        DemoAddDate()
}
```

Output:

```
» 1969-07-20 20:18:04 +0000 UTC
» DemoAdd
» 1969-07-21 00:18:04 +0000 UTC
» 2014-08-19 20:20:12.205032905 -0600 MDT
» 2014-08-20 20:20:12.205032905 -0600 MDT
» 2014-08-18 20:20:12.205032905 -0600 MDT
» DemoSub
» -395190h2m8.205453284s
» DemoAddDate
» 2014-07-20 20:18:04 +0000 UTC
```

Comparisons

Comparing time is pretty easy too. Like most other types in Go, you can't just throw ‹ and › around and have it work. You have `Before`, `After`, and `Eqaul`, and they all work as you'd expect.

time/comparisons.go

```go
package main

import (
	"log"
	"time"
)

var (
	utcPlusOne = time.FixedZone("UTC+1", 3600)
	moon       = time.Date(1969, time.July, 20, 20, 18, 4, 0, time.UTC)
	moonAlso   = time.Date(1969, time.July, 20, 21, 18, 4, 0, utcPlusOne)
	now        = time.Now()
)

func init() {
	log.SetFlags(0)
	log.SetPrefix("» ")
}

func DemoBefore() {
	log.Println("DemoBefore")
	log.Printf("moon before now? %t", moon.Before(now))
}

func DemoAfter() {
	log.Println("DemoAfter")
	log.Printf("moon after now? %t", moon.After(now))
}

func DemoEqual() {
	log.Println("DemoEqual")
	log.Printf("moon equal now? %t", moon.Equal(now))
	log.Printf("moon equal moon? %t", moon.Equal(moon))

	log.Printf("moon: %s", moon)
	log.Printf("moonAlso: %s", moonAlso)
	log.Printf("moon equal moonAlso? %t", moon.Equal(moonAlso))
}

func main() {
	DemoBefore()
```

```
	DemoAfter()
	DemoEqual()
}
```

Output:

```
» DemoBefore
» moon before now? true
» DemoAfter
» moon after now? false
» DemoEqual
» moon equal now? false
» moon equal moon? true
» moon: 1969-07-20 20:18:04 +0000 UTC
» moonAlso: 1969-07-20 21:18:04 +0100 UTC+1
» moon equal moonAlso? true
```

time.Timer

If you want to be notified after a certain amount of time, you want a `Timer`. You can work with timers in a few different ways. There are the package level `After` and `AfterFunc` functions. You'll commonly see `After` being used as the idiomatic way to timeout receiving from a channel. You can also build your own timer, stop it, and reset it.

This example includes `Sleep` because it doesn't really fit anywhere else.

time/timer.go

```go
package main

import (
	"log"
	"time"
)

var (
	fiveSeconds = 5 * time.Second
)

func init() {
```

```
	log.SetFlags(0)
	log.SetPrefix("» ")
}

func DemoTimer() {
	log.Printf("before NewTimer: %s", time.Now())
	t := time.NewTimer(fiveSeconds)
	time.Sleep(3 * time.Second)
	t.Reset(fiveSeconds)
	<-t.C
	// Should be at least 8 seconds later
	log.Printf(" after NewTimer: %s", time.Now())
}

func DemoSleep() {
	log.Printf("before Sleep: %s", time.Now())
	time.Sleep(fiveSeconds)
	// Five seconds later
	log.Printf(" after Sleep: %s", time.Now())
}

func DemoAfter() {
	log.Printf("before After: %s", time.Now())
	now := <-time.After(fiveSeconds)
	// Five seconds later
	log.Printf(" after After: %s", now)
}

func DemoAfterFunc() {
	c := make(chan time.Time)
	log.Printf("before AfterFunc: %s", time.Now())
	time.AfterFunc(fiveSeconds, func() {
		// Otherwise, the program would
		// end without this getting called
		c <- time.Now()
	})
	// Five seconds later
	log.Printf(" after AfterFunc: %s", <-c)
}

func main() {
	DemoTimer()
	DemoSleep()
```

```
        DemoAfter()
        DemoAfterFunc()
}
```

Output:

```
» before NewTimer: 2014-08-21 18:53:47.948734117 -0600 MDT
»  after NewTimer: 2014-08-21 18:53:55.950396133 -0600 MDT
» before Sleep: 2014-08-21 18:53:55.950453419 -0600 MDT
»  after Sleep: 2014-08-21 18:54:00.951471835 -0600 MDT
» before After: 2014-08-21 18:54:00.951513969 -0600 MDT
»  after After: 2014-08-21 18:54:05.952094974 -0600 MDT
» before AfterFunc: 2014-08-21 18:54:05.952147241 -0600 MDT
»  after AfterFunc: 2014-08-21 18:54:10.952484904 -0600 MDT
```

Frantic-tick-tick-tick-tick-tick-tick-tock: `time.Ticker`

That was a pretty terrible Metallica album…

A ticker is like a timer, except that it keeps happening. It ticks. You could use this to implement your own cron implementation, for example, since cron is basically "run X every Y duration".

It's very simple. You make a ticker, and receive on the channel in a loop. Boom.

Oh wait.

I'm not sure why they did this, but channel you have access to a receive only channel, and you can't close it. If you want to be stopping tickers, you probably want to hold on to a `stop` channel and `select` on it and the ticker channel.

time/ticker.go

```go
package main

import (
        "log"
        "time"
)

func init() {
        log.SetFlags(0)
        log.SetPrefix("» ")
```

```
}

func main() {
	stop := make(chan bool)
	ticker := time.NewTicker(time.Second)
	time.AfterFunc(5*time.Second, func() {
		ticker.Stop()
		stop <- true
	})

	for {
		select {
		case now := <-ticker.C:
			log.Println(now)
		case <-stop:
			log.Println("stopped")
			return
		}
	}
}
```

Output:

```
» 2014-08-21 19:41:57.521759536 -0600 MDT
» 2014-08-21 19:41:58.521761552 -0600 MDT
» 2014-08-21 19:41:59.521565763 -0600 MDT
» 2014-08-21 19:42:00.521819833 -0600 MDT
» 2014-08-21 19:42:01.520993048 -0600 MDT
» stopped
```

Timezones

Timezones are actually pretty easy in go. It knows about all the normal ones, or if you don't like those you can make your own. You can parse times in specific zones, or convert times to be in a zone.

time/timezones.go

```go
package main

import (
	"log"
	"time"
)

var (
	utcPlusOne = time.FixedZone("UTC+1", 3600)
	layout     = "Jan _2 15:04:05 2006"
	moon       = "Jul 20 20:18:04 1969"
)

func init() {
	log.SetFlags(0)
	log.SetPrefix("» ")
}

func main() {
	log.Println(time.LoadLocation("Canada/Mountain"))

	moonTime, err := time.Parse(layout, moon)
	// Defaults to UTC, kind of wish it defaulted to local
	log.Println(moonTime, err)

	// Same time, different timezone
	moonTime, err = time.ParseInLocation(layout, "Jul 20 21:18:04 1969", utcPlusOne)
	log.Println(moonTime, err)
	log.Println(moonTime.In(time.UTC))

	now := time.Now()
	log.Println(now)
	log.Println(now.In(utcPlusOne))
	log.Println(now.In(time.UTC))
}
```

Output:

```
» Canada/Mountain <nil>
» 1969-07-20 20:18:04 +0000 UTC <nil>
» 1969-07-20 21:18:04 +0100 UTC+1 <nil>
» 1969-07-20 20:18:04 +0000 UTC
» 2014-08-21 20:07:31.199496356 -0600 MDT
» 2014-08-22 03:07:31.199496356 +0100 UTC+1
» 2014-08-22 02:07:31.199496356 +0000 UTC
```

unicode

There are a lot of written languages out there, did you know that? Turns out, there are WAY too many characters to express using a single byte like with ASCII, so we have all these other encodings. Mostly what you'll today to do this is UTF-8, which uses anywhere from 1 to 4 bytes to encode stuff. You should be using UTF-8 for anything new you build.

The `unicode` package lets you query these unicode characters to find out what they are. Are they a number or a letter? A graphic? Lowercase or uppercase? It's not quite as simple as ASCII.

There are also functions to convert things to upper and lower case, again, because it's not trivial. For example, make Mýrdalsjökull uppercase, I dare you. Okay it's not that hard. In fact, we already did this in the `strings` package, but guess what functions that package uses to get the job done? The ones in the `unicode` package! These deal with individual runes, so they are less exciting, but are the necessary building blocks to handle all the world's languages.

Queries

First we'll look at figuring out what a specific character is. We can ask all sort of questions, like whether something is upper or lower case, if it's a symbol or punctuation, and a whole bunch more. We won't cover them all, because they all share the same function signature so they all operate the same way.

Unicode also has a bunch of categories so we can group like runes together. For example, *Number, Decimal Digit* has 550 characters, because beyond the regular ASCII 0-9, there are Arabic, Extended Arabic, Thai, Tibetan, and the list goes on!

And because various languages have their own characters that don't appear in other languages (or maybe they do), you can check if a rune appears in various ranges of runes.

unicode/queries.go

```go
package main

import (
	"fmt"
	"log"
	"unicode"
)

var (
	thai = "□□□□□/□□□□□□□□□□□□" // I will be right back. http://www.linguanaut.com/\
english_thai.htm
)

func init() {
	log.SetFlags(0)
	log.SetPrefix("» ")
}

func DemoThai() {
	for _, r := range thai {
		fmt.Printf("%c (%U): ", r, r)

		// Query individual runes
		if unicode.IsLetter(r) {
			fmt.Print("IsLetter")
		} else if unicode.IsPunct(r) {
			fmt.Print("IsPunct")
		} else if unicode.IsMark(r) {
			fmt.Print("IsMark")
		}

		// Check if a rune appears in a single range
		fmt.Printf(", Thai?: %t", unicode.Is(unicode.Thai, r))

		// Check if a run appears in multiple ranges (ANY)
		fmt.Printf(", Thai AND Tibetan?: %t", unicode.In(r, unicode.Thai, unicode.Tibetan))

		// Check multiple ranges again (ANY)
		// unicode.In is preferred, because, c'mon, look at that code vs this code.
		// thaiOrTibetan := []*unicode.RangeTable{unicode.Thai, unicode.Tibetan}
		// fmt.Printf(", Thai or Tibetan?: %t", unicode.IsOneOf(thaiOrTibetan, r))
```

```go

		fmt.Println()
	}
}

func main() {
	DemoThai()
}
```

Output:

```
□ (U+0E41): IsLetter, Thai?: true, Thai AND Tibetan?: true
□ (U+0E25): IsLetter, Thai?: true, Thai AND Tibetan?: true
□ (U+0E49): IsMark, Thai?: true, Thai AND Tibetan?: true
□ (U+0E27): IsLetter, Thai?: true, Thai AND Tibetan?: true
□ (U+0E09): IsLetter, Thai?: true, Thai AND Tibetan?: true
□ (U+0E31): IsMark, Thai?: true, Thai AND Tibetan?: true
□ (U+0E19): IsLetter, Thai?: true, Thai AND Tibetan?: true
/ (U+002F): IsPunct, Thai?: false, Thai AND Tibetan?: false
□ (U+0E1C): IsLetter, Thai?: true, Thai AND Tibetan?: true
□ (U+0E21): IsLetter, Thai?: true, Thai AND Tibetan?: true
□ (U+0E08): IsLetter, Thai?: true, Thai AND Tibetan?: true
□ (U+0E30): IsLetter, Thai?: true, Thai AND Tibetan?: true
□ (U+0E01): IsLetter, Thai?: true, Thai AND Tibetan?: true
□ (U+0E25): IsLetter, Thai?: true, Thai AND Tibetan?: true
□ (U+0E31): IsMark, Thai?: true, Thai AND Tibetan?: true
□ (U+0E1A): IsLetter, Thai?: true, Thai AND Tibetan?: true
□ (U+0E21): IsLetter, Thai?: true, Thai AND Tibetan?: true
□ (U+0E32): IsLetter, Thai?: true, Thai AND Tibetan?: true
□ (U+0E43): IsLetter, Thai?: true, Thai AND Tibetan?: true
□ (U+0E2B): IsLetter, Thai?: true, Thai AND Tibetan?: true
□ (U+0E21): IsLetter, Thai?: true, Thai AND Tibetan?: true
□ (U+0E48): IsMark, Thai?: true, Thai AND Tibetan?: true
```

Simple Conversion

With ASCII, it's really easy to tell what something is, lowercase letters are 97 to 122, uppercase are 65 to 90, and to convert between the two, you can just add or subtract 32. Super easy. Other languages, those expressed via the full power of unicode, aren't so easy. Luckily, the `unicode` package provides helpers to convert runes between upper and lower case, so you don't have to know the specifics about how to convert some interesting Swedish letters to the case you need.

unicode/conversion.go

```
package main

import (
	"flag"
	"fmt"
	"log"
	"unicode"
)

var (
	toggle string
)

func init() {
	log.SetFlags(0)
	log.SetPrefix("» ")

	flag.StringVar(&toggle, "toggle", "MýrDalSjökuLL", "toggle the case of each unicode\
 rune")
	flag.Parse()
}

func main() {
	toggled := make([]rune, len(toggle))
	for index, r := range toggle {
		if unicode.IsUpper(r) {
			toggled[index] = unicode.ToLower(r)
		} else {
			toggled[index] = unicode.ToUpper(r)
		}
	}
	fmt.Printf("original: %s\ntoggled: %s\n", toggle, string(toggled))
}
```

Output:

```
original: MýrDalSjökuLL
toggled: mÝ\RdALsJÖ\KUll
```

UTF-16

The `unicode/utf16` is mainly used for two things. First, you can convert something into UTF-16 if that's what it needs to be in. Maybe you're generating a CSV file for Excel so that it opens nicely. If you've ever done that, you have probably ended up with a TSV[76] file in UTF-16. Because that makes sense. Anyway…

Writing

Let's write out an Excel friendly TSV file.

unicode/utf16_writing.go

```go
package main

import (
	"bytes"
	"encoding/binary"
	"encoding/csv"
	"errors"
	"io"
	"log"
	"os"
	"unicode"
	"unicode/utf16"
)

var (
	proverb = "Alla är vi barn i början."
)

func init() {
	log.SetFlags(0)
	log.SetPrefix("» ")
```

[76]Tab Separated Values

```
}

type UTF16Writer struct {
	out     io.Writer
	bom     bool
	started bool
	buf     *bytes.Buffer
}

func NewUTF16Writer(out io.Writer, bom bool) *UTF16Writer {
	return &UTF16Writer{
		out: out,
		bom: bom,
		buf: new(bytes.Buffer),
	}
}

func (w *UTF16Writer) Write(p []byte) (int, error) {
	if !w.started {
		if w.bom {
			// We're assuming little endian, since that's what Excel wants,
			// but you could easily pass in a endianess.
			_, err := w.out.Write([]byte{'\xff', '\xfe'})
			if err != nil {
				return 0, err
			}
		}
		w.started = true
	}

	_, err := w.buf.Write(p)
	if err != nil {
		return 0, err
	}

	// omg such a hack
	for {
		r, s, err := w.buf.ReadRune()
		if err != nil {
			if err == io.EOF {
				return len(p), nil
			}
			return 0, err
```

```
		}

		// The lazy hack
		if r == unicode.ReplacementChar && s == 1 {
			return 0, errors.New("incomplete rune")
		}

		err = binary.Write(w.out, binary.LittleEndian, utf16.Encode([]rune{r}))
		if err != nil {
			return 0, err
		}
	}
}

func main() {
	proverbs := [][]string{
		{"Language", "Proverb"},
		{"sv", "Alla är vi barn i början."},
		{"zh", "□□□□□□□□□□□"},
	}
	csvWriter := csv.NewWriter(NewUTF16Writer(os.Stdout, true))
	csvWriter.Comma = '\t'
	err := csvWriter.WriteAll(proverbs)
	if err != nil {
		log.Fatalf("failed writing: %s", err)
	}
}
```

Now, this TSV example comes with quite the caveat. That caveat is you probably shouldn't use it. It works, but I would question its stability in a production system.

To take a byte slice (like the `io.Write` method expects) and convert it to UTF-16 properly is quite a process, mainly because you need to handle a rune being split between two calls to the method. If you could rely on complete runes being written, that would be nice. That's sort of what we're trying to do by using `bytes.Buffer` by writing in all the raw bytes and then trying to read runes. The "lazy hack" line deals with failing to read a rune, possibly because we've hit the middle of a one.

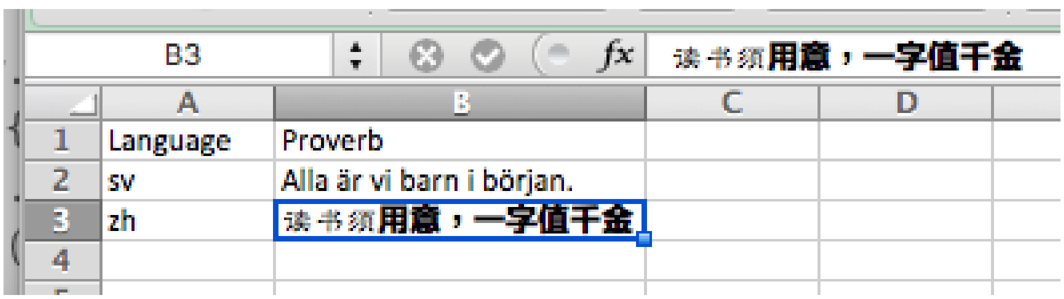
I basically searched through the standard library (a theme in this book…) to find something that would let me write bytes and read runes, and landed on the `bytes.Buffer` type. Now we can use the `unicode/utf16` package to encode the rune, and then use the `binary` package to write the encoded uint16 values to the actual output.

We had to do this exact thing in a ruby application, but the class is 34 only lines long. It was easy, but that's because we pulled in an external library to do the gnarly conversion.

Go doesn't have a one-line solution to this in the included standard library, but it does have https://godoc.org/golang.org/x/text/encoding, which is the next best thing. Those golang.org/x packages are sort of the standard library companion, a supplemental resource if you will. The developers can mess around a bit more and try things out before possibly adding them to the standard library in the case of golang.org/x/exp. Alternatively, they are just *Go Project* sanctioned libraries that the team doesn't feel the need to package with the standard distribution. This is also where some of the go tools are, like `go vet`, `godoc`, and others. Find all the golang.org/x packages on the wiki: https://github.com/golang/go/wiki/SubRepositories

So we're sort of breaking the rules by leaving the proper standard library, but we're not going too far to do it correctly. Encoding can be tricky, and in this case I'd recommend pulling in the golang.org/x package.

Anyway, what we came up with without leaving the standard library did an okay job:



Proverbs opened in Excel

Reading

The second thing you'll do is convert content from UTF-16, because let's face it, that's not a lot of fun to deal with.

Let's run this example by piping in the output from the previous writing example: `go run utf16_writing.go | go run utf16_reading.go`

unicode/utf16_reading.go

```go
package main

import (
	"bytes"
	"encoding/binary"
	"io"
	"io/ioutil"
	"log"
	"os"
	"unicode/utf16"
)

func init() {
	log.SetFlags(0)
	log.SetPrefix("» ")
}

type UTF16Reader struct {
	in      io.Reader
	bom     bool
	started bool
}

func NewUTF16Reader(in io.Reader, bom bool) *UTF16Reader {
	return &UTF16Reader{
		in:  in,
		bom: bom,
	}
}

func (r *UTF16Reader) Read(p []byte) (int, error) {
	if !r.started {
		if r.bom {
			// We're assuming little endian, since we used it in the previous example
			bom := make([]byte, 2)
			n, err := r.in.Read(bom)
			if err != nil || n != 2 {
				return n, err
			}
		}
		r.started = true
```

```go
	}

	// Read some data, deal with the ErrUnexpectedEOF here
	b1 := make([]byte, len(p)/4*4) // We have to read in multiples of 4 bytes
	n, err := io.ReadFull(r.in, b1)
	if err != nil && err != io.ErrUnexpectedEOF {
		return n, err
	}

	// binary.Read some data, make sure it doesn't return ErrUnexpectedEOF, because the\
n it just stops
	b2 := make([]uint16, n/2) // This always rounds down
	err = binary.Read(bytes.NewReader(b1), binary.LittleEndian, b2)
	if err != nil {
		return 0, err
	}

	runes := utf16.Decode(b2)
	bs := []byte(string(runes))
	n = copy(p, bs)
	return n, nil
}

func main() {
	r := NewUTF16Reader(os.Stdin, true)
	data, err := ioutil.ReadAll(r)
	if err != nil {
		log.Fatalf("failed reading: %s", err)
	}
	os.Stdout.Write(data)
}
```

Once again, this has some less than ideal code, and I wouldn't trust it in production. I'd use the `x/text/encoding` package we talked about earlier. This does get across the point that you can do some UTF16 magic with only the standard library. An annoying part is I had to call `io.ReadFull` myself and make sure `binary.Read` got exactly what it needed. This is because the `binary.Read` function just returns on any error from `io.ReadFull` before doing anything, so even if it's fine that we hit an "unexpected EOF", it doesn't know or care that everything is fine.