David Chisnall

ESSENTIAL GO CODE AND IDIOMS FOR ALL FACETS OF THE DEVELOPMENT PROCESS

The Go Programming Language

PHRASEBOOK

The Go Programming Language

PHRASEBOOK

David Chisnall

Addison-Wesley

DEVELOPER'S LIBRARY

Upper Saddle River, NJ · Boston · Indianapolis · San Francisco
New York · Toronto · Montreal · London · Munich · Paris · Madrid
Cape Town · Sydney · Tokyo · Singapore · Mexico City

Library of Congress Cataloging-in-Publication Data:

Chisnall, David.
The Go programming language phrasebook / David Chisnall.
p. cm.
Includes index.
ISBN 978-0-321-81714-3 (pbk. : alk. paper) — ISBN 0-321-81714-1 (pbk. : alk. paper)
1. Go (Computer program language) 2. Computer programming. 3. Open source software. I. Title.
QA76.73.G63C45 2012
005.3—dc23

2012000478

ISBN-13: 978- 0-321-81714-3
ISBN-10: 0-321-81714-1
This product is printed digitally on demand.

Second Printing: July 2014

Editor-in-Chief
Mark Taub

Acquisitions Editor
Debra Williams Cauley

Marketing Manager
Stephane Nakib

Managing Editor
Kristy Hart

Project Editor
Anne Goebel

Copy Editor
Gayle Johnson

Publishing Coordinator
Andrea Bledsoe

Cover Designer
Gary Adair

Senior Compositor
Gloria Schurick

Table of Contents

1 Introducing Go **1**

Go and C 1

Why Go? 4

Goroutines and Channels 7

Selecting a Compiler 10

Creating a Simple Go Program 13

The Go Type System 14

Understanding the Memory Model 16

2 A Go Primer **21**

The Structure of a Go Source File 23

Declaring Variables 26

Declaring Functions 29

Looping in Go 32

Creating Enumerations 35

Declaring Structures 37

Defining Methods 39

Implementing Interfaces 42

Casting Types 47

3 Numbers **51**

Converting Between Strings and Numbers 52

Using Large Integers 54

Converting Between Numbers and Pointers 56

4 Common Go Patterns **61**

Zero Initialization 62
Generic Data Structures 67
Specialized Generic Data Structures 69
Implementation Hiding 72
Type Embedding 75

5 Arrays and Slices 79
Creating Arrays 81
Slicing Arrays 83
Resizing Slices 85
Truncating Slices 87
Iterating Over Arrays 88

6 Manipulating Strings 91
Comparing Strings 92
Processing a String One Character at a Time 94
Processing a Partial String 96
Splitting and Trimming Strings 98
Copying Strings 102
Creating Strings from Patterns 102
Matching Patterns in Strings 104

7 Working with Collections 107
Creating a Map 108
Storing Unordered Groups of Objects 111
Using Lists 112
Defining New Collections 114

8 Handling Errors 117
Deferring Cleanup 118
Panicking and Recovering 121
Returning Error Values 125
Error Delegates 127

9 Goroutines 131
Creating Goroutines 131
Synchronizing Goroutines 134
Waiting for a Condition 137
Performing Thread-Safe Initialization 140
Performing Actions in the Background 142
Communicating Via Channels 144
Using Multiple Channels 148

10 Concurrency Design Patterns 151
Timing Out Connections 152
Aliased xor Mutable 154
Share Memory by Communicating 156
Transactions by Sharing Channels 159
Concurrent Objects 162
Implementing Futures in Go 164
Coalescing Events 166
Map Reduce, Go Style 168

11 Dates and Times 175
Finding the Current Date 176
Converting Dates for Display 177

Parsing Dates from Strings 179
Calculating Elapsed Time 180
Receiving Timer Events 181

12 Accessing Files and the Environment 183
Manipulating Paths 184
Reading a File 186
Reading One Line at a Time 188
Determining if a File or Directory Exists 190
Checking Environment Variables 192

13 Network Access 195
Connecting to Servers 196
Distributing Go 199
Serving Objects 204
Calling Remote Procedures 206

14 Web Applications 207
Integrating with a Web Server 208
Connecting to Web Servers 211
Parsing HTML 213
Generating HTML 216

15 Interacting with the Go Runtime 219
Finding the Type of a Variable 220
Finalizing Structures 223
Copying Arbitrary Types 226
Constructing Function Calls 228
Calling C Functions 230

16 Distributing Go Code **233**
Installing Third-Party Packages 234
Creating Packages 236
Documenting Your Code 240
Staying Up to Date 241

17 Debugging Go **243**
Using a Debugger 243
Misunderstanding Memory Ordering 247
Spotting Concurrency Bugs 249
Restricting Behavior 252
Building Unit Tests 257

Index **259**

This page intentionally left blank

About the Author

David Chisnall is a freelance writer and consultant. While studying for his PhD, he cofounded the Étoilé project, which aims to produce an open-source desktop environment on top of GNUstep, an open-source implementation of the OpenStep and Cocoa APIs. He is an active contributor to GNUstep and is the original author and maintainer of the GNUstep Objective-C 2 runtime library and the associated compiler support in the Clang compiler. He is also a FreeBSD committer working various aspects of the toolchain, including being responsible for the new C++ stack.

After completing his PhD, David hid in academia for a while, studying the history of programming languages. He finally escaped when he realized that there were places off campus with an equally good view of the sea and without the requirement to complete quite so much paperwork. He occasionally returns to collaborate on projects involving modeling the semantics of dynamic languages.

When not writing or programming, David enjoys dancing Argentine tango and Cuban salsa, playing badminton and ultimate frisbee, and cooking.

Acknowledgments

The first person I'd like to thank is Mark Summerfield, author of *Programming in Go: Creating Applications for the 21st Century*. If you finish this book and want to learn more, I'd recommend you pick up a copy. Mark was the person responsible for making me look at Go in the first place.

The next person I need to thank is Yoshiki Shibata. Yoshiki has been working on the Japanese translation of this book and, in doing so, has sent me countless emails highlighting areas that could be improved. If you enjoy reading this book then Yoshiki deserves a lot of the credit.

Finally, I need to thank everyone else who was involved in bringing this book from my text editor to your hands. A lot of people have earned some credit along the way. In particular, Debra Williams-Cauley, who masterminded the project, and Anne Goebel, who shepherded the book from a draft manuscript to the version you now hold.

1

Introducing Go

When learning a new language, there are three things that you need to understand. The first and most important is the abstract model that the language presents. The next is the concrete syntax. Finally, you need to learn your way around the standard libraries and the common idioms of the language.

This chapter will look at the abstract model that Go presents to programmers. If you want to dive straight into real examples, skip to the next chapter, which covers the concrete syntax. The rest of the book will cover highlights from the Go standard library and the various idioms that you will find common in Go code.

Go and C

In the late '60s, a small team at the Bell Telephone Laboratories wrote a simple operating system called UNICS, a very lightweight system inspired

by the MULTICS project, on the PDP-7 minicomputer that they had access to. When they wanted to port it to another system, they had to rewrite all of the code, which was written in PDP-7 assembly language.

To make the transition easier, they wanted to be able to share as much code as possible between different versions. They needed a language that was sufficiently low-level that a simple compiler (the only kind that existed in the '60s) could generate efficient machine code from it, yet which hid most of the irrelevant details of the target machine. BCPL was close, but it was too complex in some areas and lacked some required features in others.

Dennis Ritchie created the C programming language as a derivative of BCPL, and eventually most of the PDP-11 version of UNIX was rewritten in it. When UNIX was ported to the VAX, they just needed to retarget the compiler and write a small amount of very low-level assembly code. The majority of the system could be recompiled without modification.

Since its initial public release in 1978, C has become a very popular language. It is the de facto standard low-level language for programming these days, and it even finds use in a significant amount of application development.

The point of a low-level language is to provide an abstract machine model to the programmer that closely reflects the architecture of the

concrete machines that it will target. There is no such thing as a universal low-level language: a language that closely represents the architecture of a PDP-11 will not accurately reflect something like a modern GPU or even an old B5000 mainframe. The attraction of C has been that, in providing an abstract model similar to a PDP-11, it is similar to most cheap consumer CPUs.

Over the last decade, this abstraction has become less like the real hardware. The C abstract model represents a single processor and a single block of memory. These days, even mobile phones have multicore processors, and a programming language designed for single-processor systems requires significant effort to use effectively. It is increasingly hard for a compiler to generate machine code from C sources that efficiently uses the resources of the target system.

In 2007, Robert Griesemer, Pike, and Ken Thompson began work on a new language. Thompson had both been instrumental in the creation of C and Pike had worked on it later at Bell Labs, being members of the original UNIX team that drove the development of C. The aim of Go, their new language, was to fill the same niche today that C fit into in the '80s. It is a low-level language for multiprocessor development. Experience with C taught them that a successful systems programming language ends up being used for application development, so Go incorporates a number of high-level

features, allowing developers to use it for things like web services or desktop applications, as well as very low-level systems.

Both Pike and Thompson worked on Plan 9[1], a system designed to be a "better UNIX than UNIX." Plan 9 eventually gave birth to the Inferno distributed operating system. For Inferno, Pike created the Limbo programming language. If you've used Limbo, you will find a lot of ideas very similar. The module system, channel-based communication, garbage collection, much of the type system, and even a lot of the syntax in Go are inherited directly from Limbo. The reference implementation of Go is based on the Plan 9 compiler toolchain.

If you come from C, then many things in Go will seem familiar, but some will seem strange. As a trivial example, variable declarations in Go usually look like they are written back to front to C programmers, although if you come from other members of the Algol family, such as Pascal, then these may not seem so strange. Most of these changes come from decades of experience working with C, and seeing ways in which it can be improved.

Why Go?

In recent years, scalability has become a lot more important than raw speed. Moore's law tells us

[1] Named after the film *Plan 9 from Outer Space*.

that the number of transistors on a CPU can be expected to double roughly every 18 months. For a long time, this roughly corresponded to a doubling in performance for a single thread of execution. Now, it generally means that you get twice as many cores.

It used to be that you just had to wait six months, and your C code would run twice as fast on a new machine. This is no longer true. Now, if you want your code to be faster on new machines, then it must be parallel.

C is inherently a serial language. Various libraries, such as POSIX threads and OpenMP, make it possible to write multithreaded code in C, but it's very hard to write code that scales well. In creating DragonFly BSD, Matt Dillon observed that there was no point in creating an N:M threading model—where N userspace threads are multiplexed on top of M kernel threads—because C code that uses more than a handful of threads is very rare.

Go, in contrast, was designed with concurrency in mind. If you write idiomatic Go, then you will write code that, conceptually, does lots of things in parallel. The compiler and runtime environment can easily run this code on a single core by simply timeslicing between the various parts. They can also run it on a manycore machine by distributing the tasks across different threads.

This is a very important advantage. In the

past, I had to write some code that would work on my single-core laptop and yet scale up to a 64-processor SGI machine. Doing this in C was very hard, but doing the same thing in Erlang was trivial. In Erlang, I wrote code that used over a thousand Erlang processes, and the runtime automatically distributed them across the available cores.

The disadvantage of the Erlang version was that Erlang performs significantly worse than C in a single thread. Until you have a large number of available cores, the single-threaded C version will be faster than the concurrent Erlang version.

Go combines the best of both worlds. In single-threaded performance, it is close to C, yet it encourages a programming style that scales well to large numbers of cores. It's important to remember that the number of available cores is likely to follow a geometric growth pattern. Currently, two to eight cores is common[2] and machines with more than about 16 cores are expensive. In a few years, you will see mobile phones with 64 cores and laptops with even more. Writing C code that scales to two, or even eight cores is quite difficult but not insanely hard. Writing C code that scales to 64 or 256 cores is very challenging. With a language designed for concurrency, it is much easier.

Concurrency is the most obvious advantage

[2]If you are reading this book in a few years, this will probably seem laughably dated.

of Go, but it is not the only one. Go aims to provide a rich set of features without overcomplicating the language. Contrast this with C++, where even after having worked on a standard library implementation and a couple of compilers for the language, I still find myself having to refer to the spec periodically.

Go also includes a rich standard library, which makes developing complex web applications easy. It provides a number of mid-level abstractions, which provide high-level access to low-level features. We'll look at one of those in detail in Chapter 5, *Arrays and Slices*.

Goroutines and Channels

The fundamental concurrency primitive in Go is the *goroutine*. This is a pun on coroutine, a method of flow control popularized by Simula. A goroutine is a like function call that completes asynchronously. Conceptually, it runs in parallel, but the language does not define how this actually works in terms of real parallelism.

A Go compiler may spawn a new operating system thread for every goroutine, or it may use a single thread and use timer signals to switch between them. The exact implementation mechanism for goroutines is not specified by the language and may change over time.

By themselves, goroutines are not very useful. C lets you create threads almost as easily as

Go lets you create goroutines, yet that doesn't make it easy to write concurrent code in C. Creating concurrent subprograms (threads, child processes, or goroutines) is the easy part of the problem. The difficult part is communicating between them.

C does not provide any primitives for communicating between threads, because C does not recognize threads; they are implemented in libraries. Threads all share an address space, so it is possible to write your own code for communicating between them, and anyone who has written concurrent C code has probably done this at least once.

Go, in contrast, is designed for concurrency. It uses a form of C. A. R. Hoare's *Communicating Sequential Processes (CSP)* formalism to facilitate communication between goroutines. CSP defines communication channels that events can be sent down. Go programs can create channels and use them to communicate between threads.

A good rule of thumb for concurrent code is that the complexity of debugging it is proportional to the number of concurrent tasks multiplied by the number of possible ways in which they can interact. Because C threads use a shared-everything model, the number of possible ways in which they can interact is very large.

This is made worse by the fact that it is trivial for errors in code using pointers to mean that two C threads are sharing a data structure that

they shouldn't, for example via a buffer overrun or a dangling pointer. These problems do not manifest in Go because Go adds one feature to C and removes another. Go programs use garbage collection, making dangling pointers impossible, and disallows pointer arithmetic,[3] making most other categories of pointer-related errors impossible. We'll look at this later, in *Understanding the Memory Model*.

Creating a goroutine is intended to be much cheaper than creating a thread using a typical C threading library. The main reason for this is the use of *segmented stacks* in Go implementations.

The memory model used by early C implementations was very simple. Code was mapped (or copied) into the bottom of the address space. Heap (dynamic memory) space was put in just above the top of the program, and the stack grew down from the top of the address space. Low-level memory management worked using the `brk()` system call to add more pages at the top of the heap segment and the `sbrk()` call to add more pages at the bottom of the stack segment.

Threading complicated this. The traditional C stack was expected to be a contiguous block of memory. When you create a new thread, you need to allocate a chunk of memory big enough for the maximum stack size. Typically, that's about 1MB of RAM. This means that creating a thread requires allocating 1MB of RAM, even

[3]Except via the unsafe package.

if the thread is only ever going to use a few KB of stack space. This is required because compiled C code assumes that it can allocate more stack memory by moving the stack pointer. Operating systems usually mark the page below the bottom of the stack as no-access, so small stack overflows will cause a segmentation fault.

Go functions are more clever. They treat the stack as a linked list of memory allocations. If there is enough space in the current stack page for their use, then they work like C functions; otherwise they will request that the stack grows. A short-lived goroutine will not use more than the 4KB initial stack allocation, so you can create a lot of them without exhausting your address space, even on a 32-bit platform.

Goroutines are not intended to be implemented as kernel threads. The language does not make hard guarantees on their concurrency. Like Java threads or Erlang processes, a large number of goroutines can be multiplexed onto a small number of kernel threads. This means that context switches between goroutines is often cheaper than between POSIX threads.

Selecting a Compiler

At the time of writing, there are two stable Go compilers. The reference implementation is Gc, although it is commonly referred to as *6g*. This is based on the Plan 9 compiler toolchain.

The Plan 9 toolchain programs are named with a number indicating the architecture that they target, followed by a letter indicating their function. The three architectures supported by Go are ARM (5), x86-64 (6), and i386 (8). If you are using ARM, you would use the `5g` command instead of `6g` to compile Go programs, and `5l` instead of `6l` to link them.

The alternative is a front end for the *GNU Compiler Collection (GCC)*, called gccgo. This turns Go code into more or less the same intermediate representation that GCC uses for Fortran, C, and C++, and then subjects it to the same set of optimizations, again producing native code.

Currently, Gc is probably the better choice, although gccgo is starting to produce better code. It is the reference implementation of Go, and so is the subject of the most active development. There are several important differences between them, however.

The most obvious is that gccgo uses operating system threads to implement goroutines, and will not use segmented stacks in all configurations. This means that creating a goroutine is as expensive as creating a thread in C. If you are writing code with a high order of parallelism, then this will make gccgo much slower than 6g. If your code only uses a few goroutines, and doesn't create them very frequently, then the better optimization back end in GCC may make

it faster.

It's worth remembering that both compilers produce native executables. Go uses the same implementation model as Objective-C: native binaries and a small runtime library implementing the dynamic functionality. There is no virtual machine interpreting or JIT-compiling code. It would be possible to write a dynamic recompilation environment for Go, but the current implementations are static compilers. This means that distributing an application written in Go is as easy as distributing an application written in any other compiled language. You need to include any libraries that you use, but users don't need a large runtime environment, as they do with .NET, Java, or Python code, for example.

Since Go is a relatively new language, there will almost certainly be new implementations appearing over time. For example, it is currently possible to use the gcc front end with the LLVM code generator via the DragonEgg plugin, and a native Go front end for LLVM is likely to appear at some point.

Creating a Simple Go Program

```
$ 6g hello.go
$ 6l hello.6
$ ./6.out
Hello World!
$ go run hello.go
Hello World!
```

If you're using the Gc compiler, then you need to invoke the version of it specific to your architecture. If you're on an x86-64 system, then this will be `6g`. This takes a list of Go source files and produces object code. The object code must then be linked to produce the final binary.

At first glance, this is very similar to C, where you also first run the compiler and then the linker. There are a number of differences, which mostly make Go easier to compile.

When you run `6g`, it looks for **`import`** directives and inserts references to the relevant packages into the object code. This means that you usually don't need to specify any libraries to the linker: it will read the required packages from the object code file that you give it and link all of those into the resulting executable.

The linking step is needed to combine all of the Go packages that you use, along with any C libraries that you call via the foreign function interface, into a single executable. The compiler performs partial linking to produce packages.

The final linking step is only required when you want to import all of the separate bits of code and combine them with the system-specific preamble that all executables share.

The compiler and linker both generate default filenames from the target architecture. In the example at the start of this section, the 6g compiler generates a hello.6 object code file. If you used 8g instead, and generated 32-bit x86 code, then the resulting file would be hello.8 and the 8l linker would produce 8.out instead of 6.out. These are just the default output filenames. You can use -o with both tools to specify another filename.

As of Go 1.0, all of the details of this are typically hidden from you. The go command can compile and run programs for you with a single step. Simply type go run followed by the name of the source file and it will do all of this for you. If you specify the -x flag, then you can see exactly what this tool does as it runs.

The Go Type System

Go is a language with static typing and tight coupling between components. Go is also a language with dynamic typing and loose coupling between components. The language allows you to select which of these is more appropriate for each use case.

Go has a range of C-like primitive types and

structures that are similar to C structures, with the addition of methods (which are allowed on all Go types, not just structures) but without any form of inheritance. If you call a method on an expression with a static type directly, then the methods on it are just syntactic sugar on function calls. They are statically looked up and called.

The other side of the Go type system is visible via interfaces. Unlike Java interfaces or Objective-C protocols, they support *duck typing*[4] and don't have to be explicitly adopted. Any type that implements the methods that an interface lists implicitly implements that interface. If you've used languages in the Smalltalk family, including Python or Ruby, then you're probably familiar with duck typing.

Interface types can be used as variable types. When you call any method on an interface-typed variable, it uses dynamic dispatch to find the correct method implementation.

Go also supports *introspection* on types. You can query any variable to find out whether it is an instance of a specified type, or whether it implements a specified interface. This makes it easy to write generic data structures in Go. You can either define an interface specifying the methods that you require, or use the *empty interface*, which can be used to represent any

[4] If it walks like a duck and quacks like a duck, it's a duck.

type (including primitive types) if you are just storing values and don't need to call any methods.

One of the most useful features for a lazy programmer is the *type inference* that the Go compiler does. This allows you to avoid explicit type annotations on most variable declarations. If you combine initialization with declaration, then the compiler will infer the variable's type from the type of the expression assigned to it.

Understanding the Memory Model

Go uses *garbage collection (GC)*. Generally, people have one of two reactions to this. If you come from a high-level language, like Java, C#, Ruby, Python, or Smalltalk, then your reaction is likely to be "So what? It's a standard language feature these days." People coming from C or C++, in contrast, tend to regard GC as a decadent luxury and a sign of incompetence among programmers in general. Oh, and they also want you to get off their lawn.

Garbage collection means that you don't have to think about when to deallocate memory. In Go, you explicitly allocate values, but they are automatically reclaimed when they are no longer required. There is no equivalent of C's `free()` or C++'s `delete`. As with other garbage collected languages, it is still possible to leak objects if

you accidentally keep references to them after you stop using them.

When you're writing single-threaded code, garbage collection is a luxury. It's nice to have, but it's not a vital feature. This changes when you start writing multithreaded code. If you are sharing pointers to an object between multiple threads, then working out exactly when you can destroy the object is incredibly hard. Even implementing something like reference counting is hard. Acquiring a reference in a thread requires an atomic increment operation, and you have to be very careful that objects aren't prematurely deallocated by race conditions.

Like Java, and unlike C or C++, Go does not explicitly differentiate between stack and heap allocations. Memory is just memory. If you create an object with local scope, then current implementations will allocate it on the stack unless it has its address taken somewhere. Future implementations might always allocate it in a young GC generation and then move it to another generation if it has remaining references after a short amount of time. Alternatively, they may perform better escape analysis to allocate objects on the stack even if they have their address taken, as long as they are never referenced after the function in which they are allocated returns.

Go is designed to make garbage collection relatively easy to implement, although the