Go IN ACTION

SECOND EDITION

Joel Holmes
Andrew Walker
with William Kennedy

MANNING

praise for the first edition

A concise and comprehensive guide to exploring, learning, and using Go.
—Steven Francia, Creator of Hugo

This authoritative book is a one-stop shop for anyone just starting out with Go.
—Sam Zaydel, RackTop Systems

Brilliantly written comprehensive introduction to Go. Highly recommended.
—Adam McKay, SUEZ

This book makes the uncommon parts of Go understandable and digestible.
—Alex Vidal, Atlassian

Practical and in-depth introduction to Go.
—Thomas O'Rourke, Upstream Innovations Ltd.

Coming from a Java Enterprise world to the Go land has been a fun adventure mostly because of this book. It delivered pretty quickly everything I needed to start pushing beautiful code to production.
—Paulo Pires, littleBits Inc.

A great book, whether as a reference for an experienced Gopher, or a Baby Gopher, with a lot of explanation that will make you understand the how and why of writing Go code.
—Benoît Benedetti, University of Nice

One of the best ways right now to get Go-ing.
—Jeffrey ‘jf’ Lim, Wander

Go in Action, Second Edition

Joel Holmes, Andrew Walker with William Kennedy

To comment go to livebook.

Manning
Shelter Island

For more information on this and other Manning titles go to manning.com.

copyright

For online information and ordering of this and other Manning books, please visit www.manning.com. The publisher offers discounts on this book when ordered in quantity. For more information, please contact

Special Sales Department

Manning Publications Co.

20 Baldwin Road

PO Box 761

Shelter Island, NY 11964

Email: orders@manning.com

©2026 by Manning Publications Co. All rights reserved.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by means electronic, mechanical, photocopying, or otherwise, without prior written permission of the publisher.

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in the book, and Manning

Publications was aware of a trademark claim, the designations have been printed in initial caps or all caps.

♾Recognizing the importance of preserving what has been written, it is Manning's policy to have the books we publish printed on acid-free paper, and we exert our best efforts to that end. Recognizing also our responsibility to conserve the resources of our planet, Manning books are printed on paper that is at least 15 percent recycled and processed without the use of elemental chlorine.

The author and publisher have made every effort to ensure that the information in this book was correct at press time. The author and publisher do not assume and hereby disclaim any liability to any party for any loss, damage, or disruption caused by errors or omissions, whether such errors or omissions result from negligence, accident, or any other cause, or from any usage of the information herein.

Manning Publications Co.
20 Baldwin Road
PO Box 761
Shelter Island, NY 11964

Development editor: Doug Rudder
Technical editor: Pascal Bertrand
Review editor: Dunja Nikitovic
Production editor: Keri Hales
Copy editor: Andy Carroll
Proofreader: Jason Everett

Typesetter: Tamara Švelić Sabljić
Cover designer: Marija Tudor

ISBN 9781633439702

Printed in the United States of America

dedication

To my partner Chelsea, who has believed in me and helped me find my voice.

—Joel Holmes

contents

preface

Around a decade ago I was staring at my career wondering what direction I should be going. I was a Java developer at the time, and somewhere along the way it hit me: Java was already 20 years old. Did I want to do that for the rest of my career and end up like a COBOL developer? I started digging around and stumbled across Go. I was immediately struck by how simple it was, how everything you needed seemed to just be there. Looking at the books available at the time, I landed on the first edition of *Go in Action*, and that was the game changer.

Java had taught me a lot, though I didn't fully appreciate it until I left that world. Years of working in it gave me a real sense of what I cared about in code: Could I test it? Could I catch errors at compile time? Could I understand my code six months after writing it? Testability, type safety, legibility. Those principles stuck with me because Java punished you when you ignored them.

What Go did was let me hold onto all of that and throw out everything I hated. No application context. No Spring Boot annotations decorating every class with magical macros that required digging to fully understand. I could write a small, testable, type-safe service with the standard library and just run it.

I started writing Go professionally, and that day job turned into years of designing and shipping real production systems in Go. Eventually I had enough opinions to write *Shipping Go*, a book about applying those same principles (testability, type safety, legibility, and so on) to production applications. Writing that book forced me to articulate

things I'd been doing instinctively, and I came away from the experience with a much sharper sense of how well Go solves many software engineering problems.

So when Manning approached me about a second edition of *Go in Action*, I didn't hesitate. But I also didn't take it lightly. Go has changed dramatically in the decade since the first edition came out. Generics landed, being one of the most significant additions to the language in its history. Error handling has evolved to handle more complex scenarios. The testing toolchain now includes fuzzing and benchmarking as first-class features, not afterthoughts. I couldn't in good conscience hand readers a book that treated any of that as out of scope.

This book is for developers who already know how to program in other languages or want to upgrade their Go knowledge. You don't need me to explain what a loop is, but you may want to know about various looping patterns in Go. What you need is an honest, precise account of how Go works, how it thinks, and how to use it to build software you won't be embarrassed by in a year. That's the book I tried to write.

—Joel Holmes

acknowledgments

First, as in all things, I want to thank my wife and partner, Chelsea. She has always believed that the things I love to do are worth doing. She has encouraged me throughout each book I've written, helping me to find a voice, to write with confidence, and to believe in myself.

I'd also like to thank my two sons, Eli and Abel, who inspire me to teach, to write, and to listen and learn.

To Doc, who early in my career taught me that work is just work, no matter how much you love it, and that the life you build around it is what actually counts.

Thank you to William Kennedy for the initial edition of this book, which set me on my path to becoming a Go developer.

This book could not have been written without the support of Manning's publishing team. Thank you, Andy Waldron, for this opportunity to write the book that restarted my career. Doug Rudder had the unenviable job of keeping this book on track through every missed deadline and last-minute revision, and he did it with patience and grace. I'm deeply grateful for his guidance throughout the process. Thanks must go to my technical editor, Pascal Bertrand, who reviewed the manuscript for any technical issues, providing invaluable feedback that enhanced the accuracy and clarity of this book. Pascal is a senior software engineer with more than 15 years of experience as a developer and is the co-author of *Learn Go with Pocket-Sized Projects* (Manning, 2025). And last but not least, thanks also to the behind-

the-scenes team at Manning who helped coordinate the publishing of this book.

As a reviewer of many Manning books myself, I especially appreciate all those who provided feedback in the book reviews: Alexey Vyskubov, Bob Shock, Brandon Friar, Dmytro Lypai, Garry Offord, Ggicci T'ang, Gopal Venkatesan, Hyeonggeun Oh, Jens-Uwe Mager, Mark Harris, Martin Dehnert, Nicolas Fantoni, Nigel V Thomas, Peter Sellars, Raul Murciano, Rene Perrin, Richard Hilliar, Rishita Shaw, Rob Lacey, Victor Jones, and Yevhen Grinman. I appreciate everyone who purchased this book early via MEAP and provided feedback and support.

To my friend Cody, who I've known for 25 years. He has always been there to listen and to provide a sanity check for many critical parts of my life. Thanks for always being there.

To Frank, who has guided me over my career by providing support and advice for each decision I've made.

To EasyPost, a leading logistics platform that simplifies shipping for e-commerce and logistics companies, whose support made this book possible. Working alongside a team that genuinely values engineering craft inspired some of this writing.

And finally, thank you to my high school English teachers, who encouraged my writing and helped me establish my voice.

—Joel Holmes

about this book

Go in Action, Second Edition, tries to embody the spirit of the Go programming language by trying to keep everything as simple and clear as possible. Every chapter walks through an aspect of the language and the spirit and reason behind it. Rather than cataloging every language feature, this book focuses on understanding Go well enough to use it confidently in production.

This second edition is not a light refresh. Go has evolved substantially since the first edition, and this book treats those changes seriously. Generics, the updated testing toolchain with fuzzing and benchmarking, the `log/slog` structured logging package, modules and workspaces: these are not footnotes. They are first-class topics woven throughout the book.

Who should read this book

This book is written for developers who already know how to program. You don’t need prior Go experience, but you should be comfortable with concepts like variables, functions, loops, and data structures in at least one other language. If you’re coming from Python, Java, JavaScript, or C, you’ll find the comparisons throughout the book useful for grounding Go’s approach in something familiar.

This book is also valuable for developers who already know some Go but want to deepen their understanding of the type system, concurrency model, or testing practices. If you’re using Go 1.17 or earlier, the chapters on generics,

error handling patterns, and the standard library will cover ground that may be entirely new to you.

How this book is organized: A road map

The book is structured to move from language fundamentals to practical application. The early chapters build the vocabulary you'll need: types, collections, and the composition model that replaces inheritance. The later chapters apply that vocabulary to the problems you'll actually face: handling errors consistently, writing testable code, reasoning about concurrent execution, and organizing a multimodule codebase.

Chapter 1 introduces Go's origins and motivations, explaining why it was designed the way it was: fast compilation, strong typing, first-class concurrency, and what problems it was built to solve.

Chapter 2 gets you writing real Go immediately. You'll build a working command-line program from scratch, covering the toolchain, project structure, and the workflow you'll use throughout the rest of the book.

Chapter 3 covers Go's primitive types in depth: integers and their variants, Booleans, strings, runes, and pointers. Understanding these precisely matters because Go's type system does not coerce silently.

Chapter 4 introduces Go's built-in collection types: arrays, slices, and maps. These are foundational data structures that appear in nearly every Go program, and this chapter explains their behavior, their differences, and how to work with them idiomatically.

Chapter 5 explores Go's approach to modeling data and behavior. Named types, structs, embedding, interfaces, and methods are covered here alongside type assertions and

type switches. Go uses composition rather than inheritance, and this chapter shows how that model works in practice.

Chapter 6 covers generics, the most significant addition to the language in its history. You'll learn how type parameters and constraints work, how to write generic functions and data structures, and when generics are the right tool versus when simpler approaches will suffice.

Chapter 7 examines Go's error handling philosophy. Errors are values in Go, not exceptions, and this chapter explains how to work with that model effectively, including custom error types, wrapping, sentinel errors, and the controlled use of `panic` and `recover`.

Chapter 8 covers Go's testing ecosystem: unit tests, coverage analysis, benchmarks, fuzz tests, and examples. The `testing` package provides all of this out of the box, and this chapter treats testing as a design discipline, not just a verification step.

Chapter 9 introduces Go's concurrency model, covering goroutines, channels, and `select`, through the construction of a concurrent RSS reader. The chapter builds the application incrementally, introducing each concurrency primitive as it becomes necessary rather than all at once.

Chapter 10 surveys the standard library through a practical project: a command-line tool for parsing and converting between JSON and XML. It covers `fmt`, `os`, `io`, and the data parsing packages, demonstrating how much can be built without reaching for external dependencies.

Chapter 11 addresses code organization at scale: packages, modules, versioning, and workspaces. It takes the RSS reader from chapter 9 and refactors it into a multimodule

architecture, introducing `go.mod`, `go.sum`, and the patterns used in real-world Go projects.

Appendix A builds a vector search service with persistent storage. Vector search sits at the heart of AI-adjacent infrastructure (embedding lookups, semantic retrieval, recommendation systems) and it demands exactly the kind of high-throughput, low-latency performance that Go is built for. The appendix is a reference implementation you can adapt and extend, and it puts Go's benchmarking and performance tooling to work in a context that reflects where the industry is heading.

Appendix B builds a pipeline that integrates a large language model API into a Go application, demonstrating patterns for working with external services, handling streaming responses, and structuring AI-adjacent Go code.

The chapters are designed to be read sequentially, since later chapters build on concepts introduced earlier. That said, if you are already comfortable with Go's type system, you can skip chapters 3 through 5 and return to them as a reference. Chapters 6 through 11 are each relatively self-contained once you have the fundamentals.

About the code

All code examples in this book are written in Go. The examples are minimal by design: each one illustrates a specific concept without unnecessary scaffolding. Where a longer program is built incrementally across sections, that progression is intentional: you'll see the code grow as the ideas do.

The examples require Go 1.25 or later. No special hardware is needed. Each chapter is self-contained in its own

directory and can be run with `go run` or `go test` from the command line. Any external dependencies are declared in the chapter's `go.mod` file.

This book contains many examples of source code both in numbered listings and in line with normal text. In both cases, source code is formatted in a `fixed-width font like this` to separate it from ordinary text. Sometimes code is also **`in bold`** to highlight code that has changed from previous steps in the chapter, such as when a new feature adds to an existing line of code.

In many cases, the original source code has been reformatted; we've added line breaks and reworked indentation to accommodate the available page space in the book. In rare cases, even this was not enough, and listings include line-continuation markers (➥). Additionally, comments in the source code have often been removed from the listings when the code is described in the text. Code annotations accompany many of the listings, highlighting important concepts.

You can get executable snippets of code from the liveBook (online) version of this book at https://livebook.manning.com/book/go-in-action-second-edition. The complete code for the examples in the book is available for download from the Manning website at https://www.manning.com/books/go-in-action-second-edition, and from GitHub at https://github.com/holmes89/go-in-action-code.

liveBook discussion forum

Purchase of *Go in Action, Second Edition,* includes free access to a private web forum run by Manning Publications where you can make comments about the book, ask technical questions, and receive help from the authors and other users. To access the forum and to subscribe to it, go to https://livebook.manning.com/#!/book/go-in-action-second-edition/discussion.

Manning's commitment to our readers is to provide a venue where a meaningful dialogue between individual readers and between readers and the authors can take place. It is not a commitment to any specific amount of participation on the part of the authors, whose contribution to the forum remains voluntary (and unpaid). We suggest you try asking the authors some challenging questions lest their interest stray! The forum and the archives of previous discussions will be accessible from the publisher's website as long as the book is in print.

about the authors

Joel Holmes is a software developer who has been focused on building cloud native applications. He has worked at several start-ups helping architect, design, and develop new products and services to help those companies develop and grow. Along the way, he was able to help establish tools and processes that helped development and increased quality. He lives in Pittsburgh with his family and currently works building cloud applications at EasyPost.

Andy Walker is a software engineer living in Columbia, Maryland and working in threat intelligence. Originally from Charleston, West Virginia, he has been working with computers ever since he destroyed the family 286 at the age of 12 and had to restore it "from, like, 30 floppy disks." A consummate generalist, he is never so happy as when he is learning new skills and creating art, music, food, or useless widgets on a 3D printer.

William Kennedy is a managing partner at Ardan Labs in Miami, Florida. Ardan is a group of passionate engineers, artists, and business professionals focused on building and delivering reliable, secure, and scalable solutions. Bill is also the author of *Go in Action* and the *Ultimate Go Notebook*, plus the author of over 100 blog posts and articles about Go. Lastly, he is a founding member of GoBridge and the GDN, which are organizations working to increase Go adoption through diversity.

about the cover illustration

The figure on the cover of *Go in Action, Second Edition*, "Man from the East Indies," is taken from a book by Thomas Jefferys, published between 1757 and 1772.

In those days, it was easy to identify where people lived and what their trade or station in life was just by their dress. Manning celebrates the inventiveness and initiative of the computer business with book covers based on the rich diversity of regional culture centuries ago, brought back to life by pictures from collections such as this one.

1 Go is the solution

This chapter covers

- Go’s 2007 origin and its five design goals
- Fast compilation and type safety: how Go makes the compiler a productivity tool
- Efficient execution: native performance, managed memory, and self-contained deployment
- Built-in concurrency: goroutines, channels, and context as language primitives
- Simplicity by design: composition, interfaces, and a deliberately small language
- World-class tooling: the standardized toolchain that ships with every Go installation

Go is a language that rethinks several assumptions that have governed mainstream programming for decades. Where most languages treat concurrency as a library problem, Go makes it a language primitive. Where most languages force a choice between fast compilation and a strong type system, Go delivers both. Where most languages leave tooling to the ecosystem, Go ships a complete, standardized toolchain with every installation.

This chapter examines the decisions that shaped Go and the problems those decisions were designed to solve. You will come away with a clear mental model of why Go looks the way it does: why it has no classes, why its compiler is so fast, why goroutines are cheaper than threads, and why the standard toolchain exists. Understanding the *why* makes the *what* much easier to use well.

1.1 The problems Go was built to solve

In 2007, three engineers at Google had had enough. The codebases they worked in were enormous. Build times were punishing. Writing safe concurrent code in C++ was so difficult that most engineers avoided it entirely, accepting slower programs rather than fighting the language. The tools designed for an earlier era of software were straining under demands they had not been built to handle.

The three engineers were Ken Thompson, Rob Pike, and Robert Griesemer. They were not junior engineers looking for a new challenge. Between them, they had co-designed Unix, co-invented UTF-8, and built the compiler infrastructure behind both the Java HotSpot VM and Google's V8 JavaScript engine. Each of them had spent a career navigating the consequences of poor language design at scale, and each had clear opinions about what a better language would need to do.

So they sat down together and started designing one. They were not interested in academic novelty. They wanted a language they would actually want to use every day, one that would make a Google-scale engineering organization faster, safer, and less fragile. They left that conversation with five goals:

- *Fast compilation* to enable rapid development cycles and keep engineers in flow
- *Efficient execution* to handle the high-concurrency, high-throughput demands of modern networked services
- *Excellent support for concurrency* built in from the ground up, not bolted on afterward

- *Ease of programming* to reduce cognitive load, minimize boilerplate, and make the language learnable quickly
- *Robust tooling* for formatting, testing, and dependency management, shipped with the language itself

The language they built is called Go. It was open sourced in 2009, reached version 1.0 in 2012, and has since grown from an internal Google experiment into one of the most in-demand skills in the industry.

What makes Go unusual is not that it solves any one of these problems well. Python is easy to learn, C++ executes efficiently, and Java has a mature type system. What makes Go rare is that it addresses all five goals together, without significantly compromising any of them. That combination did not happen by accident. It happened because the people who designed Go had already seen firsthand what breaks down when you prioritize one of those goals at the expense of the others.

The sections that follow look at each of those goals in turn.

1.2 Fast compilation and type safety

When Thompson, Pike, and Griesemer set out to fix what was broken about their day jobs, two problems topped the list: build times that destroyed momentum, and runtime bugs that should have been caught before the code ever shipped. Go addresses both at the compiler level.

The Go compiler was built around a set of structural constraints that together make it fundamentally faster than its predecessors. Go does not allow circular imports: if package A imports package B, package B cannot import package A. This keeps the dependency graph directed and acyclic, so the compiler can always process it in a well-

defined order. Every import must also be used; if your code imports a package but does not reference anything from it, the compiler refuses to build. This is not just a stylistic rule. It is a structural guarantee that the compiler never processes dead code. Go's type system uses inference selectively, so the compiler knows the type of most variables from context, letting you write concise code without sacrificing the guarantees of a statically typed language.

The combined effect is a compiler that does less work per build than its predecessors. It runs fast enough that most Go developers trigger a build on every file save, using it as a near-instant feedback loop. The compiler becomes less like a gatekeeper you visit at the end of a session and more like a collaborator you work with continuously.

1.2.1 Development speed

Compiling a large application in C or C++ can take longer than getting a cup of coffee, and figure 1.1 has been a programmer's joke for decades precisely because it captures something real.

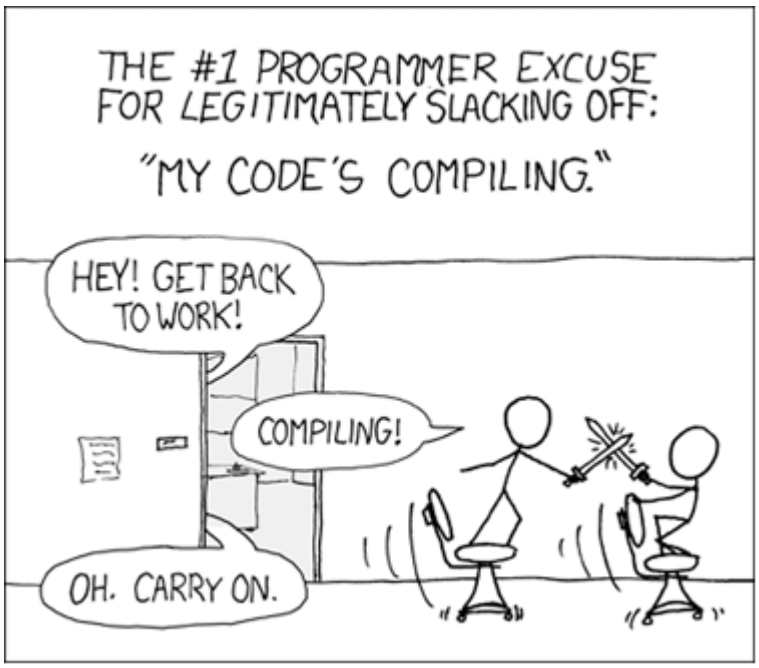


Figure 1.1 Working Hard? (via XKCD)

As a Go developer, you will need to find another excuse if you want to slack off. Go's compiler is fast by design, and the design traces directly back to Thompson's work on C.

Go requires every package to explicitly declare its imports, and the compiler examines only those directly imported packages. It does not traverse the full transitive dependency chain the way C, C++, and Java compilers do. This was a deliberate constraint placed on the language from the start: if your code does not import something, the compiler does not touch it. As a result, most Go programs compile in under a second, and the entire Go standard library compiles in seconds on modern hardware.

That speed closes the productivity gap with dynamic languages, which have traditionally justified their lack of static type-checking by pointing to fast iteration cycles. With Go, you get the fast edit-run loop *and* the safety net of a compiled, type-checked language. It is not a tradeoff.

1.2.2 Type safety

If you have ever worked in a dynamic language, you have probably seen something like the following error.

Listing 1.1 A common dynamic typing exception

```
AttributeError: 'NoneType' object has no attribute 'strip'
```

If you were lucky, you caught it during testing. But most developers have seen exactly this kind of error surface in production logs at the worst possible moment.

Type errors happen when a variable receives a value that is incompatible with how it is used elsewhere in the program. They sneak in as edge cases, code paths triggered only by a specific combination of user input, and by the time they surface, tracing the bad value back to its source can be genuinely hard work.

The industry has increasingly recognized this problem. TypeScript brings type safety to JavaScript, and Python's type hint system has grown rapidly in popularity. Both are meaningful improvements, but both involve tradeoffs. TypeScript adds a compilation step between the code you write and the code that runs. Python's hints remain advisory without additional tooling to enforce them.

Go takes a simpler position: every value has a type, the compiler tracks it, and type mismatches are build errors, not runtime surprises, as shown in the following listing.

Listing 1.2 Type mismatch caught at compile time

```
var number int
var str string

number = 1
str = "one"
number = str
// Error: cannot use str (variable of type string) as int in assign
ment
```

This code will not fail when it runs. It will never run at all. The compiler rejects it outright, and an entire category of bugs is eliminated before you ever deploy.

This is Griesemer's influence made visible. His work on HotSpot and V8 gave him a clear view of the costs that type ambiguity imposes at runtime, in error handling, in performance optimization, and in debugging. The Go compiler was designed to catch those problems where they are cheapest to fix: on your local machine, before anything ships.

1.3 Efficient execution

Efficient execution in Go means something broader than raw CPU throughput. It covers how the runtime manages memory, how the binary gets from your machine to the target, and how Go programs work alongside existing native code.

Go is garbage-collected: you do not manually allocate or free memory. Unlike the stop-the-world collectors you may have encountered in other languages, Go's garbage collector runs concurrently with your program, producing sub-millisecond pause times even under heavy load. For most workloads, you simply write your code, and the runtime keeps things running efficiently.

Thompson's background in C and systems programming meant the team understood exactly what it cost developers to manage memory manually, and what it cost users when that management went wrong. The Go garbage collector is the answer to both. You get the performance characteristics of a compiled, native language without the memory management responsibility that has historically come with it.

One of the quieter advantages of Go is that it does not require you to rewrite everything at once. Go's `cgo` facility lets you call existing C and C++ libraries directly from Go code, making it a practical choice for teams that want to migrate incrementally, adopting Go for new services while their existing native code continues to run.

This was a deliberate philosophical decision. The Go team recognized that the world runs on C and C++ code: operating system APIs, cryptographic libraries, database drivers, and graphics engines. Forcing developers to abandon that ecosystem to adopt Go would have been a significant barrier. By supporting C interop from the start, Go positioned itself as a practical upgrade path rather than a clean-break rewrite. A team can start by writing new services in Go, call into existing C libraries where needed, and migrate their most critical paths over time as confidence with the language grows.

Go's compilation model reinforces this pragmatism. Go compiles to a single, statically linked binary by default, with no runtime dependencies and no shared libraries to install on the target machine. You build your program, copy the binary, and run it. On systems where deployment complexity carries a real cost, that simplicity matters. It is also one of the reasons Go became the language of choice

for container tooling: Docker and Kubernetes both ship as self-contained binaries, and both are written in Go.

1.4 Concurrency built in

Chip manufacturers hit a wall around 2005. Transistors had gotten about as small as physics would allow, and the expected doubling of single-core performance every two years had slowed dramatically. The industry's answer was to go wide: more cores per chip, more chips per server. Servers today routinely ship with 64 or 128 cores. The mobile phone in your pocket almost certainly has more CPU cores than a laptop did ten years ago. The potential is enormous, but only if your software can actually use those cores.

Most languages treat concurrency as a library problem: you reach for a threading API, a thread pool, or an external framework. The result is code in which concurrency concerns are tangled up with business logic, and correctness depends on careful, manual synchronization of shared memory. Get that synchronization wrong, and you end up with race conditions and deadlocks: bugs that appear intermittently, under load, in ways that are extraordinarily hard to reproduce.

Go's approach to concurrency was not invented for Go. Rob Pike had been thinking about this problem for nearly two decades before Go existed. His language Newsqueak and later Limbo, used in the Inferno operating system, were both experiments in applying Tony Hoare's research paper on the communicating sequential processes (*CSP)* model to practical programming. CSP describes programs as independent processes that communicate by passing

messages rather than sharing memory. When Pike joined the Go project, that body of thinking came with him.

The result is a concurrency model that is part of the language itself: not a library you add, not a framework you configure, but syntax you use the same way you use a function call.

1.4.1 Goroutines

Go's concurrent execution primitive is called a *goroutine*: a function that runs concurrently with the rest of your program. To start one, you use the `go` keyword.

Listing 1.3 Starting a Goroutine

```
go myFunction()
```

That is the entire syntax. There's no thread pool to configure, no executor service to initialize, and no callback to register. The Go runtime manages goroutines for you, scheduling thousands of them across a small pool of OS threads, and starting, pausing, and resuming them as needed, as illustrated in figure 1.2.

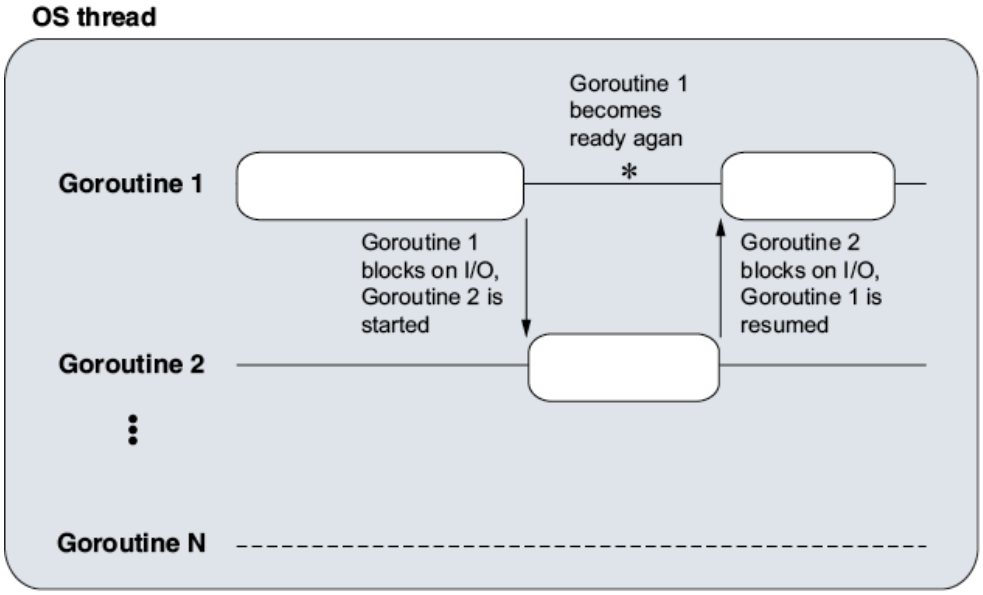


Figure 1.2 Many goroutines execute on a single OS thread

Goroutines are far cheaper to create than OS threads. Starting one costs a few kilobytes of stack space, compared with a megabyte or more for a thread. A single Go program can comfortably run thousands of goroutines at once, which is why Go's `net/http` package can handle enormous numbers of simultaneous HTTP connections behind a simple server written in a handful of lines. You will go deep on goroutines in chapter 9.

1.4.2 Channels and context

For goroutines to be useful, they need a way to communicate safely. Go provides *channels*: typed, synchronized conduits that let goroutines send data to each other without sharing memory, as illustrated in figure 1.3.

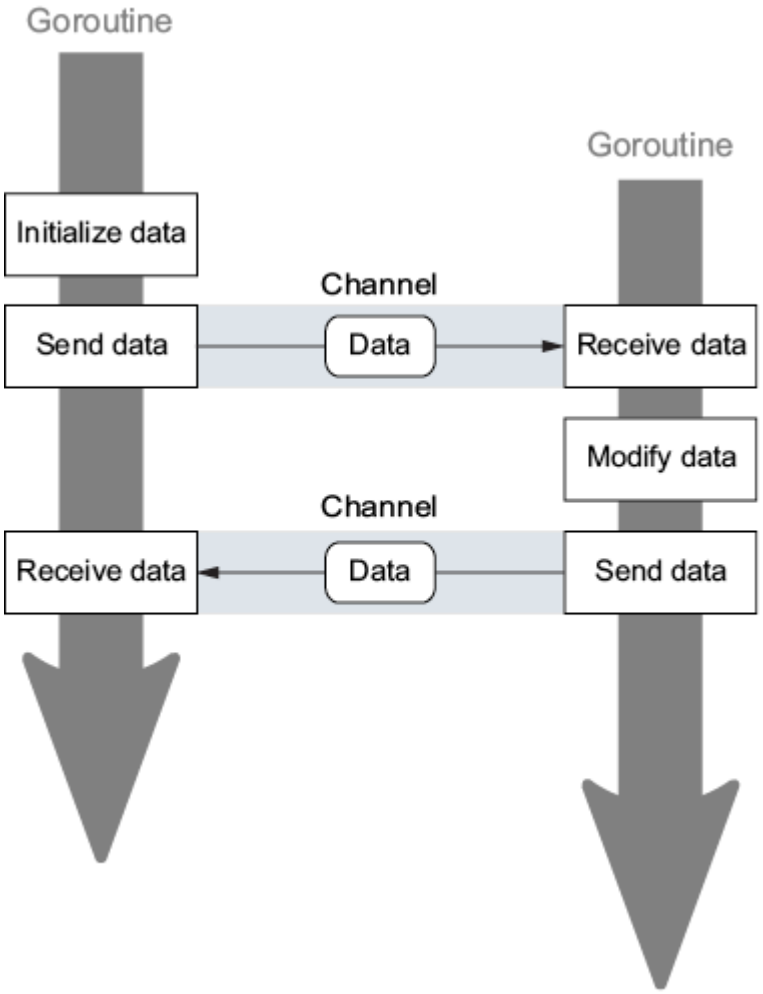


Figure 1.3 Channels pass data between goroutines

You can think of a channel as a synchronized mailbox. One goroutine places a value into it; another takes that value out. The exchange is atomic: both goroutines know it happened, and neither needs a mutex or a condition variable to make it safe. The channel itself handles the synchronization, so coordinating goroutines becomes a matter of passing values rather than guarding shared memory by hand. This is the idea behind a well-known Go proverb: don’t communicate by sharing memory; share memory by communicating.

Goroutines are cheap to start, but you also need a way to stop them. When a user disconnects, a deadline passes, or

a request is canceled, goroutines still running on behalf of that work need a clean exit. Go's `context` package provides exactly that: a cancellation signal and optional deadline that flows through an entire call chain, letting every goroutine in a pipeline shut down gracefully when the work is no longer needed.

Together, goroutines, channels, and context form a complete, coherent system for concurrent programming, one that was designed from the start rather than assembled from parts.

1.4.3 A request's life cycle

To see how these pieces fit together, picture a Go web service handling a single HTTP request, something a busy server does millions of times a day (figure 1.4). A request arrives, and the server starts a goroutine to handle it: a few kilobytes of stack with nothing to configure. That goroutine may start others—one to query a database and another to call a downstream API—and they coordinate by passing results over channels. A context travels with the request the whole way down: if the client disconnects or a deadline passes, every goroutine in the chain sees the cancellation and winds down. When the response goes out, the garbage collector reclaims whatever the request allocated, with no manual cleanup and no perceptible pause. The whole service is a single static binary, copied onto the machine and run with nothing to install.

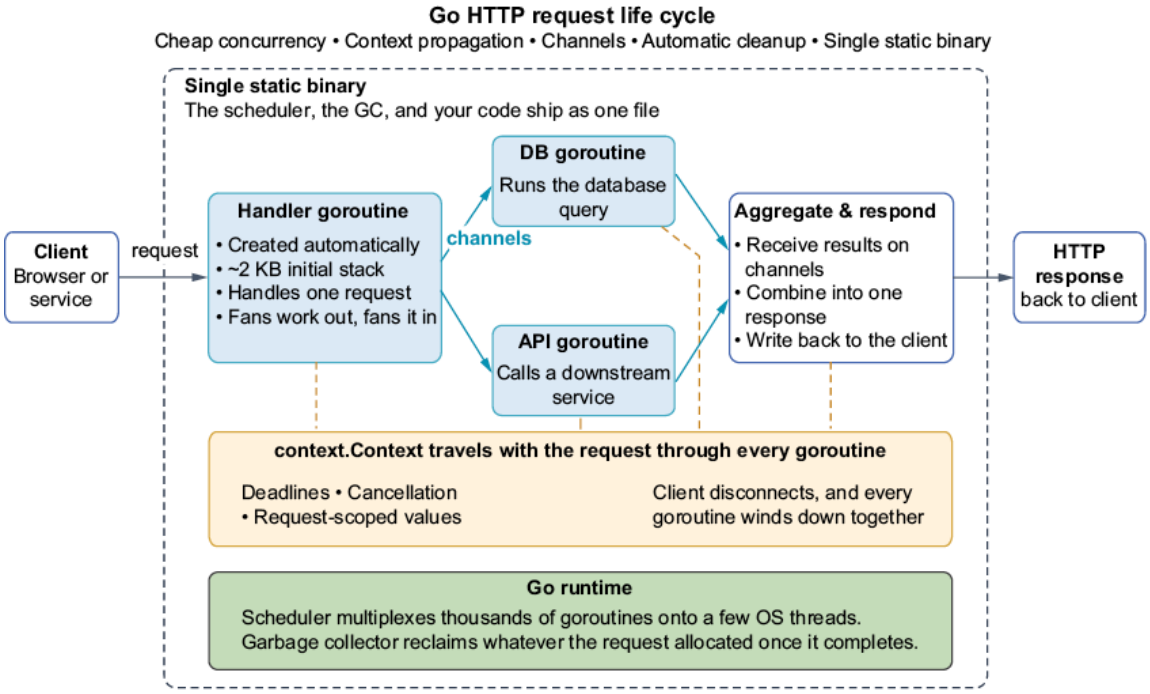


Figure 1.4 The life cycle of a single HTTP request

Building the same service the conventional way would carry a cost at every step, and Go's design choices answer each one directly. Handling each request on its own operating-system thread would cost about a megabyte of stack apiece and cap a server in the low thousands of concurrent requests; Go's goroutines start at a few kilobytes, so the same machine can carry tens of thousands. Coordinating those threads through shared memory would mean locks, along with the intermittent races that come from getting them wrong; a channel moves the data instead and leaves nothing to lock. Letting the work run with no cancellation signal would leave the child goroutines running long after the client has disconnected, piling up results no one will read; context is the off switch. Shipping a language runtime and its shared libraries alongside your program would create a version-matching problem on every machine you deploy to, whereas a single static binary is just a file you copy. The request itself is ordinary. What is rare is that Go makes all of it cheap at the same time.

1.5 Simplicity by design

Go is defined not only by what it includes but by what it deliberately leaves out. There are no classes, no class-based inheritance hierarchies, no operator overloading, and no manual memory management. Go does support *struct embedding*, which lets one type promote the fields and methods of another, but this is composition rather than inheritance: there is no type hierarchy, no virtual dispatch, and no implicit "is-a" relationship between the two types. What remains after all those subtractions is a small, orthogonal set of features that compose cleanly, along with a toolchain that makes working with them feel effortless.

One of the most striking things about Go's specification is how short it is. The language has a small number of keywords, no operator overloading, no implicit conversions, no class-based inheritance, and no exceptions. These are not oversights. They are decisions the team made deliberately, and decisions that Pike, Griesemer, and Thompson were qualified to make. Each of them had spent careers navigating the complexity that accumulates in large systems when a language gives you too many ways to do the same thing.

Instead of using classes and class hierarchies, you build types by composing simpler ones, a design the team chose in part because they had watched inheritance trees become a maintenance problem in large codebases at Google. Behavior accumulates in base classes that no single engineer fully understands, and changing anything near the root of the tree risks breaking code farther up the chain. This is visualized in figure 1.5.

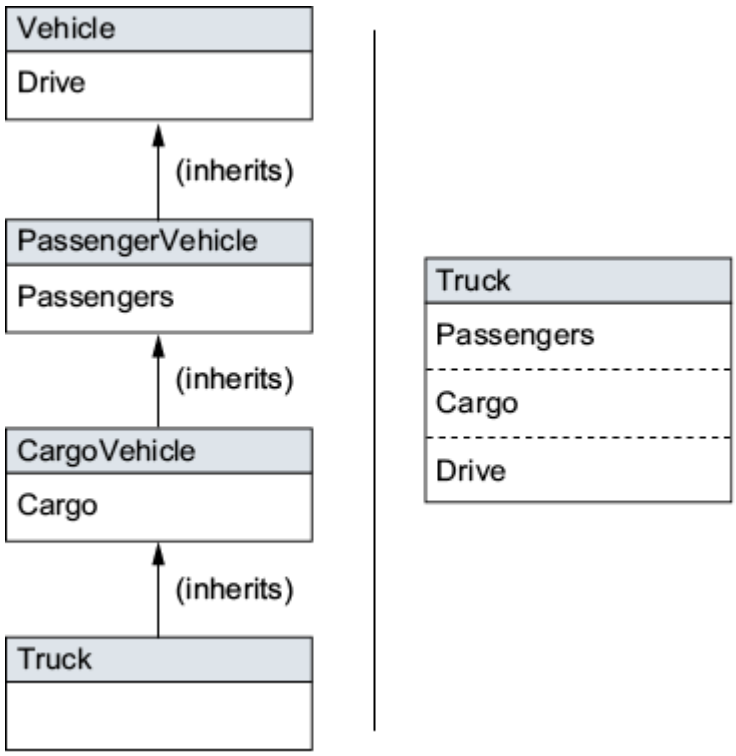


Figure 1.5 Inheritance vs. composition

The practical upside is that Go code tends to stay readable as it grows. You define a `struct` to hold data, attach methods to define its behavior, and embed structs together to model more complex concepts. There is no superclass to reason about and no method resolution order to untangle.

Go's interfaces complete this picture. An *interface* in Go is a set of method signatures, and any type that implements those methods satisfies the interface without explicitly declaring that it does. You can define an interface after the types it describes already exist, with no changes to those types required. The compiler figures it out automatically. This *implicit satisfaction* makes it easy to write flexible, testable code and tends to produce small, focused interfaces rather than the sprawling abstract base classes common in object-oriented languages.

Go 1.18 added *generics*, type parameters that let you write functions and data structures that work across a range of types with full compile-time type safety. When you need reusable code that works across multiple types, and interfaces are not the right fit, generics are the answer. Chapter 6 covers them in full.

1.6 World-class tooling

Here is something worth pausing on. You have probably worked with a language where tooling was a community problem. Someone had to choose a formatter, but which one? Someone had to pick a test framework, but there were six of them. The `package.json` for a new project had forty-two development dependencies before you had written a single line of application code.

Go ships with a built-in formatting tool, `gofmt`, which parses your source code and rewrites it into a single layout with consistent indentation, spacing, and alignment, regardless of who wrote it or how they normally format their own code. This isn't a linter offering style suggestions; it's an enforced standard with no configuration options, no style guide to debate, and no room for personal preference. That tradeoff is by design, and it's one of the things Go's creators are most emphatic about.

Rob Pike put it plainly: "Gofmt's style is no one's favorite, yet gofmt is everyone's favorite." No one gets exactly the style they would have chosen. Everyone gets to stop arguing about it.

That sentence captures something important about how the Go team thought about tooling. Most languages treat tooling as something the ecosystem will sort out over time.

The Go team had a different view, shaped by years of watching what happens when ecosystems do not.

At Google, software engineering happens at a scale where inconsistency compounds. Code formatted differently by different engineers slows down code review. Builds that are not reproducible create intermittent failures that are expensive to debug. Tests that require framework setup are tests that some engineers will skip. Pike and Griesemer had watched this play out for years, so they made a decision: Go's tooling would be standardized, consistent, and shipped with the language itself, not as an afterthought but as a first-class part of what Go *is*.

Every installation of Go includes the following tools:

- `go build` compiles your program. One command, no Makefile, no build configuration file.
- `go run` compiles and runs a program in a single step, making it as quick as a scripting language for fast iteration.
- `go fmt` formats your code according to the official Go style. Every Go codebase in the world is formatted the same way. Style debates end at the language level.
- `go test` runs your tests. There's no test framework to install, no configuration file to write. Table-driven tests, subtests, benchmarks, and fuzzing are all built-in and are covered in depth in chapter 8.
- `go vet` is a static analysis tool that catches common mistakes the compiler allows but that are almost certainly bugs.
- `go doc` generates and serves documentation directly from your source code. No separate documentation tool is required.
- `go mod` manages your dependencies. Every dependency is pinned to a version, and the `go.sum` file provides a cryptographic record of exactly what was downloaded, making your builds auditable and reproducible.

Consistent tooling matters more than it might seem at first. It means that any Go codebase, whether written by a team

of two or two hundred, is formatted, built, and tested the same way. That consistency allows engineers to move between Go projects without a lengthy onboarding period, and it allows large organizations to maintain Go codebases at scale without a dedicated build-systems team.

It also matters for performance-sensitive work. The `go test -bench` command gives you a professional benchmarking framework as part of the standard installation; no third-party profiling library is required. Appendix A shows you how to build a vector search service, the kind of component that is increasingly central to AI-adjacent applications. Measuring and optimizing the performance of that kind of code is something the Go toolchain handles out of the box.

1.7 What this book covers

The following chapters build on the foundation laid here, moving from language fundamentals (types, collections, and Go's composition model) to the problems that arise in real codebases: consistent error handling, testable concurrent code, and organizing a multimodule project. Two appendices round out the book with a vector search service and an LLM integration, both of which put the language's performance and tooling strengths to work in practical applications. For a complete chapter-by-chapter breakdown, see the "About this book" section at the beginning of the book.

Go is now nearly twenty years old, and the fact that it still looks and feels much like it did at its 1.0 release in 2012 is a testament to how well the original design held up. The language has grown, adding generics in 2022 and continuously improving its tooling, but its essential character has not changed. The simplicity that made it learnable in 2012 is still there. The concurrency model is still the one Pike derived from his decades of CSP research. The compiler is still fast. The toolchain still ships as a single download.

That stability is not conservatism for its own sake. It reflects how thoroughly the original design was thought through. Thompson, Pike, and Griesemer were not working from abstractions; they were working from experience. Every constraint in Go, every feature deliberately left out, and every tool included from day one was a response to a real problem they had personally encountered while building real systems for real organizations. This is not a language designed by committee or evolved through community

negotiation. It is a language designed by engineers who had already paid the cost of the problems it solves.

The results speak for themselves. Docker, the container runtime that transformed how software is deployed, is written in Go. Kubernetes, the platform that now orchestrates most of the world's cloud infrastructure, is written in Go. Terraform, Vault, CockroachDB, InfluxDB, and a long list of other foundational tools are all written in Go. These are not simple programs. They are large, concurrent, performance-sensitive systems that must be maintained by large teams over many years. They were built in Go because Go was built for exactly that kind of work.

Learning Go means you're learning a language designed from day one to solve the kind of problems that matter in professional software engineering. This book takes you through each of those solutions in turn, building your understanding of not just what Go does but why it was built that way. When you understand the *why*, the *what* becomes much easier to use well.

Summary

- Go was designed in 2007 by Ken Thompson, Rob Pike, and Robert Griesemer to solve real software engineering problems at scale.
- Fast compilation and strict type-checking close the gap with dynamic languages without sacrificing correctness.
- Go compiles to a single, statically linked binary with no runtime dependencies, making deployment as simple as copying a file.
- Concurrency is built into the language, not bolted on.
- Go's type system favors composition over inheritance.

2 Diving into Go

This chapter covers

- Building your first Go program
- Go source structure
- Go language fundamentals

You paid money for a book called *Go in Action*, so let's see some code already! Rather than just throw a bunch of reference material at you (there are plenty of other great books for that!), we thought we might try something a little more hands-on. This chapter will walk you through the creation of a single piece of software that mirrors a real-world utility, touching on various language features and fundamentals along the way.

Go is more than just a language and a compiler; it's an entire ecosystem of tools designed to help you format your code, manage dependencies, test code, and research documentation. Rather than gloss over these tools here, we believe you should get your hands dirty with them as soon as possible so that you can build a well-rounded foundation as a Go programmer.

Let's get to it!

2.1 Goal: A program to count words

If there's one thing computers are good at, it's counting things really, really quickly, which makes these tasks ripe for automation with the sort of elegant code programmers thirst for. A very common thing for programmers to count is

lines and words in text files, mainly because this is the quickest way to show off how many lines of code (or "loc" if you're cool) you've written, so that other programmers will be jealous of how productive you are.

This is so common, in fact, that there is already a ubiquitous command-line tool called `wc` (short for "word count") that does this and comes packaged with pretty much every Linux and macOS system. You could just use that, of course, but if we did, this chapter would be very short, you would leave very poor book reviews, and Manning would be very angry with us. So instead, let's say you've decided that `wc` is too old and crusty for your refined tastes and you're going to rewrite it in Go!

2.2 What you need first

We encourage you to follow along with the code in this book to get the most out of it and get used to writing Go in an actual development environment. You could do this with just a Go installation and your operating system's built-in terminal, but you'll have a better experience with an IDE or an alternative terminal emulator. We'll discuss these in detail below, but you can also visit the book's website at https://github.com/holmes89/go-in-action-code/blob/main/IDEs.md, where we keep an up-to-date list of recommended software and quick-start instructions for getting set up!

2.2.1 A code editor

You can use any editor you like to write Go, so use whatever is comfortable for you. If you're unsure where to start, the Go Wiki provides several popular suggestions. Check out our book's website, mentioned above, or head

over to https://go.dev/doc/editors and follow the link to commonly used editor plugins and IDEs listed on the Wiki.

The landscape of code editors is constantly shifting, so this book does not focus on any particular coding environment. Different editors have different levels of Go support, some of which might require extensions or plugins for features such as autocompletion and documentation. Check the documentation for your particular editor if you don't find it at the preceding link, and consider installing any support packages you need. Some editors might even install Go for you! If that's the case, just make sure you still have access to the Go tools from the command line, since this book will be using them pretty regularly.

If in doubt, you're probably safe to follow the steps in the next section to install Go on your operating system of choice. This will guarantee that you can access the tools you need.

2.2.2 Git

Git (https://git-scm.com/) is a free and open source revision control system, which is a piece of software used for managing software projects. It is one of the most popular tools for this purpose, and it is integrated into the Go toolchain as the default option for fetching dependencies from popular open source software repositories, such as GitHub. If it is not installed by your editor integration of choice, download it from https://git-scm.com/downloads for your system and install it so that you can easily fetch third-party libraries once you start to use dependencies beyond the Go standard library.

2.2.3 The Go toolchain

You'll also need Go itself. Go comes packaged for most major operating systems, so the best way to kick this process off is to visit https://go.dev/doc/install and follow the instructions there. This will install the latest Go distribution on your system, which includes the extensive standard library as well as the compiler and various command-line tools that you'll be using to build and test your software.

2.2.4 Terminal access

You will also want some interface to type these commands into, usually in the form of a *terminal emulator* application, sometimes also called a "command prompt" or simply a "terminal." If you are using an IDE, you may find that it has one built in, but every modern OS also has its own version of a built-in terminal. These can be a little bare-bones, though, so you may want to try out something else. Check out https://github.com/holmes89/go-in-action-code/blob/main/TERMINALS.md for some possibilities!

2.3 Creating your project root

Setting up a new Go project involves two steps: creating a directory to hold your code, and initializing that directory as a Go module. Let's walk through each in turn.

2.3.1 Creating a directory

The first step in every new Go project is creating a directory for your code to live in. Whether this code will be a reusable library or a runnable program, this directory, called the *project root*, is the home base for all of your code so that Go knows where to find everything and how to build it. Each

of the projects in this book will have its own root directory to keep it well contained.

For this project, you'll create a directory called `wordcount`. It might also help to create a single directory to hold all of the project root code for this book. Throughout this book, we'll locate projects in the `gobook` directory, but feel free to omit the containing directory if you like.

2.3.2 Initializing the project module

Go uses a system called *Go modules* to give names to individual software projects and manage their dependencies so that anyone who has access to the source code can be sure they are working on the same program with the same dependencies and versions as everyone else. Modules will be covered in greater detail in chapter 11, but for now, issue the following command in the `wordcount` directory to get your project ready for the code.

Listing 2.1 Initializing the wordcount module

```
$ go mod init gobook/wordcount
go: creating new go.mod: module gobook/wordcount
```

You can use the module name `gobook/wordcount` whether or not you put your new project in a `gobook` directory. For now, the only relevant portion of the name is the final path segment, which *must* match the directory name you run the command in. You could, of course, simply have named it `wordcount`, and this would work fine, but for convention's sake, you might want to get used to prefixing the module name with something, since eventually this prefix will contain repository information.

THE GO COMMAND

This is the first of many appearances of your new best friend and constant companion throughout your journey as a Go programmer: the `go` command. You will be using it *a lot*.

This little Swiss Army Knife does everything from building your code to testing it, installing its dependencies, and everything in between. It is broken down into a series of subcommands, such as `mod` (which handles dependencies). We'll touch on it only briefly in this chapter, but rest assured, there will be plenty to learn about this tool and more in chapter 3.

After this command is run, you will be left with a directory containing one file: `go.mod`. This directory is now the *main module* for your project, and the Go toolchain will reference this `go.mod` when building any code in this directory or in any directories beneath it. If you look inside this file, you will see something very similar to the following.

Listing 2.2 Your new go.mod file

```
module gobook/wordcount

go 1.25
```

There's not a lot to see here yet because this is a brand-new project with no external dependencies (or code). The first line is called the *module directive*, and it indicates that the name of this module is `gobook/wordcount`. The last line is called the *go directive*, and it says that this software was developed using Go version 1.25.

Now you have a shiny new module just brimming with possibilities for the future. Time to write some actual code!

GOROOT IN DAYS OF YORE

History lesson: In the early days of Go, you needed to have a `GOPATH` environment variable that pointed to a directory where all of your source and dependencies lived. This directory needed a `src` subdirectory for all of your source, a `bin` subdirectory for tools and such that you built, and a `pkg` subdirectory for cached library objects. This was fine, but all of your code ended up in one place, which was a little weird. Also, it didn't handle versioning very well, so we had to make sites like `gopkg.in` to keep versions separate; otherwise, you needed a bunch of separate `GOPATH`s, each with their own copies of dependencies. You also had to check dependencies in sometimes, to make sure they didn't get deleted from public repos or updated on you. So, yeah, it was a whole thing, and there were a bunch of third-party tools like `gb` and `glide` and `dep` that stepped up to fill in the gaps.

Eventually, though, we got Go Modules which does versioning and lets you build a project wherever, as long as you do a `go mod init` first. More importantly, we also got the Go package and checksum databases, which act as public mirrors for most Go code checked in to the major public repositories, making it very unlikely that your dependencies will just disappear on you.

2.4 The first iteration: Counting spaces

Spaces are the natural division between words, so as a first pass, you'll write a simple counter for spaces in a string. This counter will serve as the test case for the rest of the chapter, gaining improvements along the way.

2.4.1 Creating main.go

The first order of business is creating a file that contains some Go code to run. Open a file called `main.go` and put the following code in it.

Listing 2.3 Your first main.go file

```
package main

import "fmt"

func main() {
        // Use hard-coded text to start.
        text := "let's count some words!"

        var numSpaces int

        // Look at the text one byte at a time.
        for i := 0; i < len(text); i++ {
                if text[i] == ' ' {
                        numSpaces++
                }
        }

        fmt.Println("Found", numSpaces+1, "words")
}
```

Just like that, your first Go program is on the books! This code represents a complete Go program that counts the number of spaces in the `text` variable by looking at each character and checking whether it's a space. This implementation has a couple of rough edges, but you'll fix those soon enough. First, try it out!

2.4.2 Building the program with go build

To run your program for the first time, you can use the `go build` command from the directory where `main.go` resides. This command compiles all the .go files in that directory and, by default, creates an executable file with the same name as the directory itself.

If you're following along, this should produce a binary file called `wordcount` (or `wordcount.exe` on Windows) that is ready to run. You can then call it as a command, as follows.

Listing 2.4 Running the wordcount executable from the command line

```
$ ./wordcount
Found 4 words
```

Congratulations! You are now a Go programmer! Welcome, fellow Gopher.

2.4.3 Running the program with go run

Alternatively, you can type `go run` from inside the directory and run it without building first.

Listing 2.5 Running the program

```
$ go run main.go
Found 4 words
```

`go run` is another handy command provided by the `go` tool for quick tests like this (see chapter 3 for more), and it will be helpful as you add to the program later in the chapter.

Before you go running off to monetize word-counting-as-a-service (WCAAS is a thing, right?), let's take a minute to check your formatting.

2.4.4 Formatting your code with gofmt

If you're using a code editor or IDE, you may have noticed some small formatting changes to your code when you saved it. This is `gofmt` at work.

`gofmt` is another tool that comes bundled with the Go distribution, and its job is simple: make all Go code look the same everywhere.

Once upon a time, this might have seemed like a strange thing to add to a programming toolchain, but it is quickly becoming the norm, especially as public, open source communities have become more popular. There was a time not so long ago when you'd see lengthy newsgroup diatribes on syntactic style that could span weeks. It was all very fraught and dramatic, and a lot of good development time was burned debating the merits of variable grouping and cuddled else statements. Ain't no one got time for that.

With standard formatting, it doesn't matter what your preference might be. What style of bracketing should you use? Doesn't matter; it's `gofmt`'s way. This is actually rather freeing, since now you just write your code and don't worry about it. Moreover, standard formatting lowers the cognitive load of reading unfamiliar code, since at the very least, its shape will be the same. It's great.

If you're unsure about whether your editor is calling `gofmt` for you, just try the following command on your shiny new `main.go`.

Listing 2.6 Testing your formatting with gofmt

```
$ gofmt -d main.go
```

The `-d` (for "diff") flag tells `gofmt` that you want to see any differences between your code and the standard format. If you're lucky, it will output nothing, meaning your editor has probably already taken care of that for you. On the other hand, you might see something like this.

Listing 2.7 Output of gofmt -d when your formatting isn't standard

```
--- main.go.orig
+++ main.go
@@ -3,13 +3,13 @@
 import "fmt"

 func main() {
-  // Use hard-coded text to start.
-  text := "let's count some words!"
+       // Use hard-coded text to start.
+       text := "let's count some words!"

        var numSpaces int

        // Look at the text one byte at a time.
-       for i := 0; i < len(text); i++{
+       for i := 0; i < len(text); i++ {
                if text[i] == ' ' {
                        numSpaces++
                }
```

You can always use `gofmt -w` to modify your files in place, but who has time for that? The point of `gofmt` is not to have to care about it, and you shouldn't.

Now that that's out of the way, let's break this down and explore how the different pieces fit together.

2.4.5 Package declarations

Go programs are divided into "packages," which are the basic unit of code organization in Go. Packages define

boundaries between the different concerns of your code and determine which functions, types, and values are visible in other parts of the program. Each module can contain one or more packages, each of which can be imported separately if the programmer wishes.

Listing 2.8 Declaring package main

```
package main
```

The code line in listing 2.8 defines this file as belonging to the `main` package, which is special. It indicates to the Go compiler that you intend to produce an executable program that can be run all on its own. When Go sees a file or files in a directory that define the `main` package, it searches through them all, looking for a function called `main`, which is also special because it represents the starting point for the program.

If your program has both a `main` package and a `main` function, the compiler will output a single executable that can then be run from the command line.

2.4.6 Import declarations

Various pieces of code will live outside of the `main` package for better code organization. In order to use these packages, we will need to use what is known as an *import declaration*, which indicates that this package (`main`) is importing one or more external packages as dependencies. This helps the Go compiler know what additional software it needs to build to support your program.

Listing 2.9 Importing the fmt package

```
import "fmt"
```

In this case, there's only one dependency, but there can be any number of import declarations. They can appear on a single line (as in this example) or in blocks of multiple imports, which you'll see shortly. This part of the code must come directly after the package declaration so that the build toolchain can gather dependency information and build those dependencies, if needed, before moving on to your code.

In this case, the single dependency is the `fmt` package, which handles formatted output and provides some helpful functions for printing output to the screen.

2.4.7 Blocks and scope

The rest of the program (listing 2.7) is a single function, `main`, which begins on line #6 and ends with the closing brace on line #19. `main` takes no arguments and returns no values. It is the *entry point* into your program, meaning that when the program is run, code starts executing from this point. A `return` from `main` terminates the program.

Listing 2.10 The main function

```
func main() {
```

As you might have noticed, Go code uses the C-style approach of defining *blocks* of code, such as functions and loops, with curly braces `{}`, which mark the beginning and end of the block. Blocks determine the *scope*, or visibility, of identifiers such as function names and variables. Names declared inside a given block are visible only inside that block after they are declared, and they are visible in any inner block declared after them.

In addition to the normal brace-delimited blocks around functions and loops, there is also a single scope shared by

each package, called the "package block," which includes all of the functions defined in that package as well as identifiers declared outside of functions, such as variables. Anything declared at this level is visible throughout the package. Above this is the "universe block," which contains built-in functions such as `append`, as well as all of the built-in Go types and constants.

In figure 2.1, each of the identifiers has been marked, and you can see four different levels of scope at work. The eye icons indicate that each scope has visibility into the identifiers of any scopes that surround it. This means that at the innermost level, the code in the `for` loop has access to any variables declared above it in the `main` function, as well as to all the items in the package scope. It cannot see `newAmount`, because this was declared later, nor does it have access to the identifiers in the `addAmount` function, because they do not share a scope.

```
package main

import "fmt"

var amountToAdd int = 1

func addAmount(number int) int {
    newNumber := number + amountToAdd
    return newNumber
}

func main() {
    message := "The new number is:"
    for num := 0; num < 5; num++ {
        fmt.Println(message, addAmountTo(num))
    }
    newAmount := 10
    fmt.Println("new amount is", newAmount
}
```

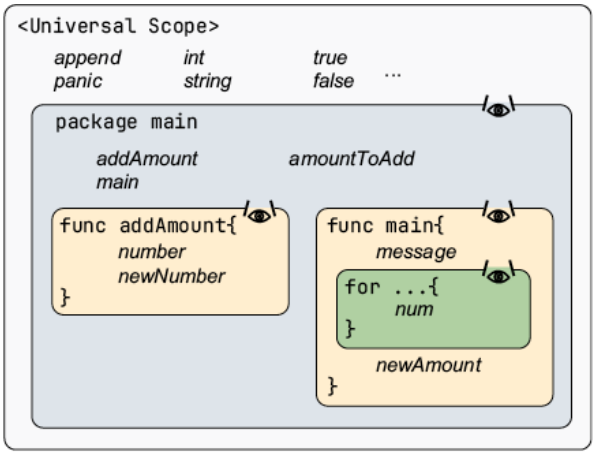


Figure 2.1 Scopes and visibility

2.4.8 Variable declaration

As you learned in chapter 1, all variables in Go have a *type*, which defines what kind of values they can hold. Types can be assigned to variables explicitly by using the `var` keyword, or you can use the short declaration operator (`:=`) to do it automatically.

Listing 2.11 Explicitly declaring the numSpaces counter variable with var

```
var numSpaces int
```

In the preceding listing, `numSpaces` is declared explicitly by using the `var` keyword, which defines the new variable and assigns it Go's integer type, `int`. Because it is a numeric value, it is automatically assigned a default value of 0, making it ready to use immediately. This is known as the *zero value* for that type. Declarations using `var` can also specify an initial value in the form `var name T = V`, where `T` is the type and `V` is the value, but the short variable form is usually more convenient.

> **NOTE** The zero values for Boolean values, numeric values, and strings are false, 0, and the empty string (""), respectively. All other types will get a value of nil. This is covered in greater detail in chapter 5.

Listing 2.12 Declaring and assigning text in a single step

```
text := "let's count some words!"
```

The preceding code line declares `text` with the short variable operator `:=`, which acts exactly like an assignment except that it also declares a new variable at the same time. The `text` variable automatically receives the type `string` through a mechanism called *type inference*, which allows Go to infer the type of the new variable from the value you are attempting to assign to it. This is a very useful feature

because it saves you from needing to perform separate declaration and assignment steps every time. Instead, you can create variables the first time you use them, and the type is taken care of for you.

SHORT DECLARATION CONSIDERATIONS

Short declarations are very useful, but they can have surprising behavior when changing scope. Consider the following example.

```
func main() {
    const defaultServer = "localhost"
        // getConfigValue returns false if the named value does n
ot exist
        serverAddress, found := getConfigValue("server")
        if !found {
                serverAddress := defaultServer
                fmt.Println("default server set:", serverAddress)
        }
        fmt.Println("connecting to server:", serverAddress)
}
// Output:
// default server set: localhost
// connecting to server:
```

Inside the `if` block, everything looks great, but once you exit, the value is empty again. What gives? This happens because the second assignment to `serverAddress` is not actually an assignment at all. Because it occurs within the `if` block, it happens inside a new scope and actually creates a *new* variable with the same name that exists only inside the `if` block, meaning the value of the outer `serverAddress` never changes. This is called *shadowing*, and it can be easy to miss. Fortunately, many editors will warn you about shadowing so that you can fix it.

2.4.9 Comments

Comments are one of the most important parts of any software, and Go uses C-style syntax: line comments start with `//`, and block comments are surrounded by `/*` and `*/`.

Listing 2.13 Go comment styles

```
// Use hard-coded text to start.
text := "let's count some words!"

// A single line comment lasts until the end of the current line.

/*
        A multi-line comment
        can span multiple lines for longer
        documentation.
*/
```

> **TIP** Go documentation is written in comments, so it helps to get in the habit of using complete sentences with proper punctuation.

2.4.10 Basic for loops

The next step is to look at each character in the string to see whether it's a space. For this, you need a loop. Although other languages provide multiple looping constructs, such as `while` and `do..while`, Go unifies them in the `for` statement.

Listing 2.14 Looping over string characters with a for loop

```
for i := 0; i < len(text); i++ {
        if text[i] == ' ' {
                numSpaces++
        }
}
```

The `for` loop in Go comes in three flavors, the first of which is the familiar three-part `for` loop. This will be familiar if you have worked with C-like languages such as JavaScript, C++, and Java.

It consists of three sections, followed by the block of code to execute, as illustrated in figure 2.2.

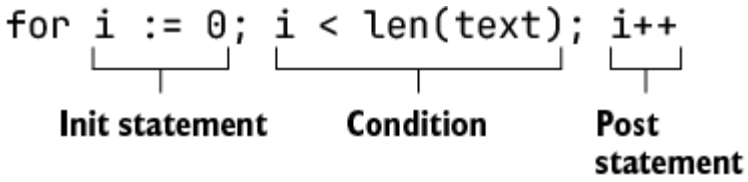


Figure 2.2 Anatomy of a for statement

The *init statement* runs once at the start of the loop and usually initializes an important variable. It is optional.

The *condition* is a Boolean expression evaluated before each loop iteration to see whether the loop should run. Unlike C and Java, Go requires a condition.

The *post statement* runs at the end of each loop iteration, including the last one. Usually, it increments some kind of counter. It is optional.

In this code, the init statement creates a variable `i` to track the character position in the string. This variable is then compared to a literal space character, `' '`. If it matches, `numSpaces++` increments the counter, and the loop continues to the end.

2.4.11 Calling functions in other packages

Finally, after all that counting and looping, it's time to return your results! This is a great job for the `fmt` package, which provides methods for printing and formatting messages.

Listing 2.15 Using `fmt.Println` to output the results

```
fmt.Println("Found", numSpaces+1, "words")
```

Go uses the familiar `<package>.<function>` syntax to call functions in other packages, so `fmt.Println` calls the `Println` function from the `fmt` package, which prints out its arguments in order, with spaces between them, to give you a nice tidy message.

> **NOTE** In Go, when an identifier such as a function name begins with an uppercase letter, it is *exported*, which means that it can be accessed by anyone who imports the package. This differs from other languages that require you to advertise the names you want to export and import using special keywords or lists. In Go, that first uppercase character is all you need. You'll see this throughout the rest of the chapter when calling functions from the standard library.

2.5 A better way to split the string

Counting spaces isn't the best algorithm for word counts, though. Sometimes people still use two spaces between words, and there's no guarantee that you won't have extra spaces elsewhere. Try adding spaces to the beginning and end of the `text` and see what happens.

Listing 2.16 Adding spaces to the text variable

```
text := " let's count some words! "
```

Suddenly, the output is wrong.

Listing 2.17 Incorrect output

```
Found 6 words
```

That's two words off! Not good! Instead of counting spaces, it would be better to split the string into different words. Turns out there's a function that can do exactly this: `strings.Fields`.

Listing 2.18 Altering main.go to use strings.Fields and len

```
package main

import (
        "fmt"
        "strings"
)

func main() {
        text := "let's count some words!"
        words := strings.Fields(text)
        fmt.Println("Found", len(words), "words")
}
```

That's much simpler! Three lines of code do all of that counting and looping.

2.5.1 Introducing package strings

The `strings` package is another standard library package that, as you might imagine, deals with processing and manipulating strings. Unlike the `fmt` package, which deals primarily with printing strings and placing variables in them, `strings` provides functions for doing all sorts of nifty stuff, such as converting strings to uppercase or lowercase, building them piece by piece, and splitting them up.

2.5.2 Using go doc to view package documentation

You can view the documentation for this package using another handy command, `go doc`, which parses the source

for documentation comments and prints them for you to read. By default, this command prints all package functions and types, but you can use the `-all` flag to make it print more detailed information, such as descriptions, methods, or basic usage.

Listing 2.19 Using go doc to view the documentation for the strings package

```
$ go doc -all strings
```

The preceding command will print a listing of all the relevant parts of the `strings` package, including the documentation for `strings.Fields`.

Listing 2.20 Documentation for strings.Fields

```
func Fields(s string) []string
   Fields splits the string s around each instance of one or more
   consecutive white space characters, as defined by unicode.IsSpac
e,
   returning a slice of substrings of s or an empty slice if s cont
ains only
   white space.
```

VIEWING DOCUMENTATION ONLINE

`go doc` is a great tool for viewing code documentation, but you can also view documentation online at the go.dev website. Maintained by the Go Project, this site lets you view documentation and find packages from across the Go ecosystem. Check out https://pkg.go.dev/strings to see the documentation online or to view the documentation for any of the packages you see in this book.

It is helpful to use standard library methods when you can, since many common use cases are already supported, which can save you time. In the case of the `strings` package, you no longer have to worry about extra spaces lurking around, since `strings.Fields` will simply ignore them for you.

2.5.3 Multiple imports in an import directive

To use `strings.Fields`, you need to import the `strings` package by adding it to the import directive. You can simply add another line (`import "strings"`), or you can import multiple packages in the same directive by using parentheses.

> **NOTE** Many code editors will take care of this for you and arrange your imports in a consistent way. If you see your imports changing slightly when you save, this is what's going on.

Listing 2.21 Multiple import directive for both fmt and strings

```
import (
        "fmt"
        "strings"
)
```

This is the way most Go code you run into will handle imports, though some authors will split them up. Usually, you will see imports grouped with the standard library imports first, followed by a blank line and then other dependencies.

2.5.4 Slices and len

Once the `strings` package is imported, you can call `Fields` using dot-notation.

Listing 2.22 Calling strings.Fields and getting the list of words

```
words := strings.Fields(text)
```

Looking at the documentation, you can see that `strings.Fields` takes a single string as an argument and returns a `[]string`, which is a *slice* of string values. We'll explore slices in more detail in the next chapter, but for now you can think of them as lists or arrays that can shrink and grow as you need them. Like arrays, they are *indexed* by number, and you can use the familiar bracket syntax to access their individual elements (see figure 2.3).

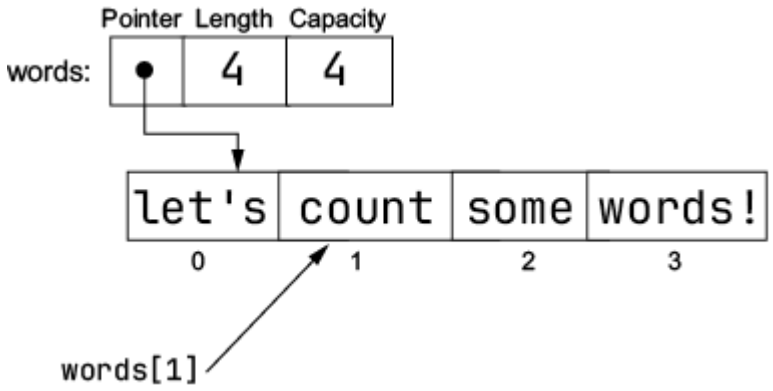


Figure 2.3 The slice of words

The built-in `len` function is provided so that you can get the length of things like strings, slices, and maps, and print it directly.

Listing 2.23 Printing the number of words using len

```
fmt.Println("Found", len(words), "words")
```

> **IMPORTANT** `len` will always return the number of elements in a collection, but indices always start at 0, so remember to stop at `len(yourVariable) - 1` if you're using a `for` loop.

2.6 Reading from a file

Now that you have a handy method for counting words, it's time to think bigger. You're never going to grab market share from `wc` in the word-counting space if you have to edit your source code every time you need to check a new string!

For that, you need to be able to load from a file, so it's time to check out the `os` package.

2.6.1 Using the os package to interact with the filesystem

The `os` package provides cross-platform access to things like environment variables, arguments, and the filesystem, which includes creating, removing, and opening files on disk.

The quickest way to get the contents of a file is to read it into memory. If you look through the `os` package documentation, you'll find a function called `os.ReadFile` that will do just that.

Listing 2.24 Altering main.go to read the contents of a file using os.ReadFile

```
package main

import (
        "fmt"
        "log"
        "os"
        "strings"
)

func main() {
        filename := "words.txt"

        fileContents, err := os.ReadFile(filename)
        if err != nil {
                log.Println(err)
                os.Exit(1)
        }

        words := strings.Fields(string(fileContents))

        fmt.Printf("found %d words", len(words))
}
```

Let’s break it down.

2.6.2 Multiple return values

Many functions in Go return more than one value, which lets a function hand back both a result and additional information, like an error.

```
fileContents, err := os.ReadFile(filename)
```

`os.ReadFile` is a bit different from the functions you’ve seen so far, most notably because it returns two values. If you look at the documentation, you can see what these values are.

Listing 2.25 Documentation for os.ReadFile

```
func ReadFile(name string) ([]byte, error)
    ReadFile reads the named file and returns the contents.
```

Looking at the function definition, you can see that it returns a slice of bytes representing the contents of the file and a value of type `error`, indicating whether there were any problems opening the file.

Functions in Go can return any number of values, and a common pattern is to return the important values from a function first, followed by an `error` value at the end.

2.6.3 Basic error handling and the log package

All error handling in Go eventually comes down to just two steps:

1. Check if an error occurred.
2. If so, handle it as soon as possible.

Error handling is important enough that we have dedicated chapter 7 to covering the topic in detail, but when you're developing small programs like this, it's often enough to log the error and exit. You can return to the error handling code later if you want to add special handling.

Listing 2.26 Logging an error and exiting

```
if err != nil {
        log.Println(err)
        os.Exit(1)
}
```

Error checking in Go is a simple matter of testing whether the error value is `nil`. If it is, you're good! If it's not, an error has happened, and you need to handle it.

To separate error output from normal program output, you can swap out `fmt` for `log`, which provides many of the same functions, including `Println`, but it prints to standard error and includes a helpful timestamp, making log messages easy to spot.

The `log` line is followed by a call to another OS function, `os.Exit(1)`, which causes the program to exit immediately with an *exit code* of 1, indicating that the program terminated in an unexpected way.

2.6.4 Counting the words in the loaded file

Now that you have the file's raw contents as a slice of bytes, you need to split that text into individual words so you can count them. The following listing uses `strings.Fields`, a function that splits a string on whitespace and returns each nonempty chunk as an element of a slice.

Listing 2.27 Using strings.Fields on the file contents

```
words := strings.Fields(string(fileContents))
```

After loading the file and checking for an error, the code is pretty similar to the last version of the program, except that `os.ReadFile` returns a slice of bytes instead of a string. This is because not all files are text, and Go doesn't want to assume anything about how you're going to handle the file. Instead, it returns the raw contents of the file, which you can use however you want.

Luckily, strings are also just bytes, so if you're loading a text file, you can use a *type conversion* to convert the bytes directly to a string, like this: `string(fileContents)`.

> **NOTE** Instead of `string`, you can also use the `bytes` package which provides many of the same methods as

`strings`, but for slices of bytes instead: `words := bytes.Fields(fileContents)`.

2.6.5 Running the program on a file

You can use `go run` to run this version of the program as before, but this time you'll need to create a `words.txt` file in the same directory as `main.go`. Earlier, in listing 2.26, we referenced this filename to be loaded by our application. In the future, you will learn how to add inputs into your applications so you don't need to hardcode the value to make the application more flexible. For now, we will just create the appropriate file we expect.

Listing 2.28 Running the application on words.txt

```
$ echo this is four words > words.txt
$ go run main.go
found 4 words
```

You can also test your error handling by removing the file and seeing what happens.

Listing 2.29 Running the application with no input file

```
$ rm words.txt
$ go run main.go
2022/05/05 20:33:39 open words.txt: no such file or directory
```

Try out different contents and see how well it works.

2.7 Specifying the file

Now that you can read the contents of a file, the next logical step is to make it work for *any* file, not just `words.txt`. To do this, you need to be able to process your program arguments. The `os` package has you covered here, too.

Listing 2.30 Altering main.go to take a filename as an argument

```
package main

import (
        "fmt"
        "log"
        "os"
        "strings"
)

func main() {
        if len(os.Args) < 2 {
                log.Println("need to provide filename!")
                os.Exit(1)
        }

        fileContents, err := os.ReadFile(os.Args[1])
        if err != nil {
                log.Println(err)
                os.Exit(1)
        }

        words := strings.Fields(string(fileContents))

        fmt.Println("Found", len(words), "words")
}
```

2.7.1 Getting program arguments from os.Args

Most languages give you access to the arguments your program was called with, and Go is no exception. When your program starts, the runtime takes stock of all the arguments passed to your program and stores them in memory. When you import the `os` package for the first time, it places them into an exported variable named `Args` (see figure 2.4). By convention, this list starts with the command itself, so it always has a length of at least 1.

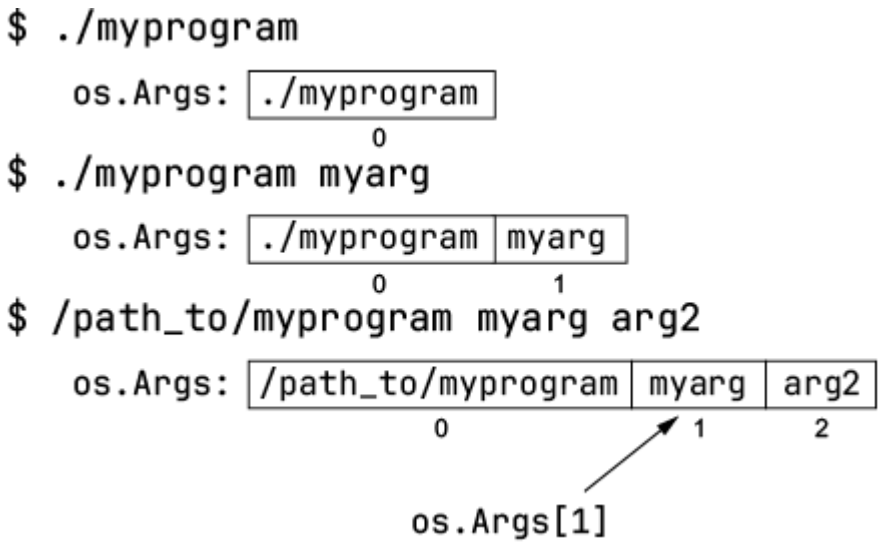


Figure 2.4 os.Args illustrated

This allows your program to access the first argument at `os.Args[1]` and treat it as the filename to load.

Listing 2.31 Opening the first argument as a file

```
fileContents, err := os.ReadFile(os.Args[1])
```

2.7.2 Avoiding a panic by using len

Because `os.Args` always contains the name of your program first, you need to look at `os.Args[1]` and above to get any of the arguments. But there's a catch: if your program isn't run with any arguments, you could end up trying to access a value that doesn't exist, as illustrated in figure 2.5.

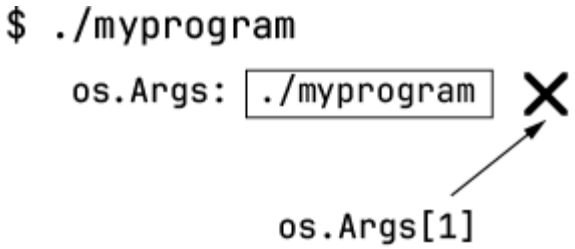


Figure 2.5 Looking where you shouldn't

If you don’t check the length of your arguments first, and you blindly try to access the first argument, this will result in a *panic*, where the Go runtime crashes your program rather than trying to read a value that doesn’t exist.

Listing 2.32 Panic output on missing arguments

```
panic: runtime error: index out of range [1] with length 1
```

To avoid this, your best bet is to check the number of arguments first with `len`.

Listing 2.33 Checking the argument length with len

```
if len(os.Args) < 2 {
        log.Println("need to provide filename!")
        os.Exit(1)
}
```

This way, your program does the right thing if it’s not given the name of a file to read from.

> **IMPORTANT** Always check the length of any slice with `len` before accessing an arbitrary element.

2.7.3 Running your program with arguments

Now that you have a more flexible program, you can try it out on any text file you like, including its own source.

Listing 2.34 Running wordcount with files as arguments

```
$ go build
$ ./wordcount words.txt
Found 4 words
$ ./wordcount main.go
Found 41 words
```

If you have `wc` installed on your system, and you're feeling cheeky, you can even test it out to see how it stacks up.

Listing 2.35 Testing against the competition

```
$ wc -w words.txt
       4 words.txt
$ wc -w main.go
      41 main.go
```

Hah, take *that* `wc`!

2.8 A more efficient method

`os.ReadFile` has been working pretty well so far, but it has one small problem: it reads the whole file into memory each time. This is fine for smaller files, but you can make it work for any number of files of any size with just a little more work.

Reading and writing data in manageable chunks is a common enough task that Go has an entire package dedicated to it. In this case, the package is called `bufio`, which buffers input and output so that your program doesn't hold on to more memory than it needs.

One of the types in this package that is particularly useful for reading bits of text is the `Scanner` type, which can split data in a variety of ways and give it to you a piece at a time. Using a package like this is handy because it allows you to focus on the important parts of the task at hand, without requiring you to handle all the implementation details yourself. In this case, it means you only have to worry about calling a couple of methods to get the latest bit of text and check for errors, without needing to write a bunch of extra code.

2.8.1 Using go doc to learn more about types

When you want to learn more about a package and its types, you can use `go doc` to get an overview by passing it the package and type names.

Listing 2.36 Output of go doc bufio Scanner

```
type Scanner struct {
        // Has unexported fields.
}
    Scanner provides a convenient interface for reading data such a
s a file
    of newline-delimited lines of text. Successive calls to the Sca
n method
    will step through the 'tokens' of a file, skipping the bytes be
tween the
    tokens. The specification of a token is defined by a split func
tion of
    type SplitFunc; the default split function breaks the input int
o lines
    with line termination stripped. Split functions are defined in
this
    package for scanning a file into lines, bytes, UTF-8-encoded ru
nes, and
    space-delimited words. The client may instead provide a custom
split
    function.

    Scanning stops unrecoverably at EOF, the first I/O error, or a
token too
    large to fit in the buffer. When a scan stops, the reader may h
ave
    advanced arbitrarily far past the last token. Programs that nee
d more
    control over error handling or large tokens, or must run sequen
tial
    scans on a reader, should use bufio.Reader instead.

func NewScanner(r io.Reader) *Scanner
func (s *Scanner) Buffer(buf []byte, max int)
func (s *Scanner) Bytes() []byte
func (s *Scanner) Err() error
func (s *Scanner) Scan() bool
func (s *Scanner) Split(split SplitFunc)
func (s *Scanner) Text() string
```

This is a typical document listing for a package type, and it's a good example of how `go doc` presents documentation at

a higher level. The first thing you'll see is the definition of the type itself, as well as any fields it might export inside the curly braces. In this case, `Scanner` doesn't expose any fields, so all you see is a comment that it contains unexported fields, just in case you care about that sort of thing.

After the type definition, `go doc` adds the type documentation, which is usually a description of the type and its purpose, often with information about how it works. The amount of detail varies by package, but standard library packages like this one tend to be fairly detailed, giving you as much information as possible so you don't miss anything that might be important (even if it might not concern your use case). In this example, the `Scanner` type documentation has a lot to say about how the type is used, as well as some more technical details about how it's implemented. There's a lot going on here, but it breaks down to these three points:

- `Scanner` splits data into smaller pieces called "tokens."
- The default mode is to split on newlines, meaning each token will be a single line of your file, with the trailing newline chopped off. You can also change this behavior if you want by using one of several predefined `SplitFunc` functions, or by writing your own.
- `Scanner` will keep returning tokens until you reach the end of the file (EOF) or an error occurs.

After the description, but before any methods, `go doc` also lists any functions in the same package that return a value (or pointer) of the type you are asking about. This is because some types are designed to be initialized before they are used; `go doc` lists anything that looks like it might be a constructor function for that type first, making you aware of it. In this case, there is a function called `NewScanner`

that plugs in the data source you'll be reading from and returns a new scanner for that data, ready to use.

CONSTRUCTORS IN GO

While many types in Go are designed to be useful without initialization, others require some configuration first and provide constructor functions that do the setup for you. The convention is to use a function of the form `NewT(<arguments>) *T`, where `T` is the type it returns and the arguments are any requirements or options passed to the function.

In this case, `Scanner` needs a source to read from, so `NewScanner` takes an `io.Reader` for the data source and initializes a new scanner to read from it. We'll get to `io.Reader` in just a moment, but first, you can examine the methods in greater detail to get an idea of how `bufio.Scanner` is supposed to be used. You can do this one method at a time by using `go doc <package name> <type name>.<method name>`, or you can use `-all` with the type to list them all at once.

There are three important methods for our word-counting program: `Scan`, `Text`, and `Err`.

Listing 2.37 go doc output for the Scan, Text, and Err methods

```
func (s *Scanner) Scan() bool
    Scan advances the Scanner to the next token, which will then be
    available through the Bytes or Text method. It returns false wh
en the
    scan stops, either by reaching the end of the input or an erro
r. After
    Scan returns false, the Err method will return any error that o
ccurred
    during scanning, except that if it was io.EOF, Err will return
nil. Scan
    panics if the split function returns too many empty tokens with
out
    advancing the input. This is a common error mode for scanners.
func (s *Scanner) Text() string
    Text returns the most recent token generated by a call to Scan
as a
    newly allocated string holding its bytes.
func (s *Scanner) Err() error
    Err returns the first non-EOF error that was encountered by the
Scanner.
```

It looks like the general flow is to call `Scan` repeatedly, get new tokens with `Text`, and finally call `Err` to see if anything went wrong. This is starting to make sense, but you'd be forgiven for thinking that the documentation alone doesn't give you the whole picture. Documentation is great for understanding the particulars, but sometimes it's far more helpful to see an example. To fill this gap, many Go packages include example snippets as part of their test suite, and you can view these using `go doc -ex`.

Listing 2.38 Example usage from go doc -ex bufio.NewScanner

```
scanner := bufio.NewScanner(os.Stdin)
for scanner.Scan() {
        fmt.Println(scanner.Text())
}
if err := scanner.Err(); err != nil {
        fmt.Fprintln(os.Stderr, "reading standard input:", err)
}
```

This makes the usage much clearer! Thanks to the example code, you can see that `Scan` is designed to be used with a `for` loop, which automatically takes care of getting new data on each iteration while making sure it's okay to proceed. Since `Scan` returns `false` when it's done, the loop will keep running as long as there is more data to get and nothing has gone wrong. This is a clever pattern because it avoids the need to check the `Scan` and `Err` methods repeatedly.

Examples like this can be very valuable when you are evaluating packages for use in your own code, and they can help jump-start development, since they provide a fully working example as a starting point.

> **TIP** Whenever you're in doubt about how you're supposed to use a package, try using `go doc -ex` to look for example code.

Adapting the example to the word-counting code you have so far, you can replace the code in the loop with the code you want to call for each line, and then check `Err` when you're done. This means you can use the same `strings.Fields` approach as before, but this time you're using it on only one line at a time, and you can keep a running total outside the loop.

Listing 2.39 Using strings.Fields inside the Scan loop

```
// Counter to track the running total.
var wordCount int

for scanner.Scan() {
        // This is done for each line of the file.
        words := strings.Fields(scanner.Text())
        wordCount += len(words)
}
```

This is clean and simple, and thanks to the buffering provided by the `Scanner` type, it is much more efficient. The only outstanding task is to open the file so that the scanner can read it as needed, instead of all at once with `os.ReadFile`.

2.8.2 Files and I/O interfaces

Earlier, you learned that `NewScanner` takes something called an `io.Reader` as its only input, and you need some way to connect a file to this type. If you check the documentation, you'll find that `Reader` is actually an interface with just a single method, `Read`.

Listing 2.40 Excerpt from go doc io.Reader

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

    Read reads up to len(p) bytes into p. It returns the number of
bytes
    read (0 <= n <= len(p)) and any error encountered. Even if Read
returns
    n < len(p), it may use all of p as scratch space during the cal
l. If
    some data is available but not len(p) bytes, Read conventionall
y returns
    what is available instead of waiting for more.
```

As we mentioned in chapter 1, interface values can accept any type, as long as that type has all the methods defined in the interface. With `Reader`, this requirement is easy to meet, since all you need is a type that has a `Read` method that takes a byte slice and returns an integer and an error. So what you really need is a type that represents a file and has a `Read` method.

As luck would have it, you don't have to look very far to find what you need, since right alongside `Readfile` in the `os` package is the `os.File` type, the standard Go type for representing any file on disk. A quick glance at the documentation shows that `os.File` does indeed have a `Read` method, and all you need to do is call `os.Open` with a filename, and you'll get a freshly opened `os.File` structure, ready to be accessed.

Listing 2.41 An excerpt from `go doc os.File`:

```
type File struct {
        // Has unexported fields.
}
    File represents an open file descriptor.
//...
func Open(name string) (*File, error)
//...
func (f *File) Read(b []byte) (n int, err error)
```

THE FUNCTION OF THE IO INTERFACES

One important function of the `io` package is that it defines a number of simple interface types that model important behavior for interacting with input/output sources in Go. In addition to `Reader`, you will find `Writer`, which defines sources that can be written to; `Seeker`, which defines sources that allow you to jump around in the data; and `Closer`, for types that have a `Close` method for indicating that the program is done with them. Some, such as `ReadCloser` or `ReadSeekCloser`, are combinations of two or more of the others.

By themselves, these interfaces don't perform any function, but they are important because they establish conventions for any Go code that needs to perform input and output. When you see a function that takes one of these interfaces as an argument, you will know right away how it intends to use the value and what it can and cannot do with the underlying type. Further, by implementing one of these interfaces in your own code, you allow any types you create to be used in any of the many different locations that follow these conventions.

The `io` interfaces are among the most common standard interfaces you will run across when looking at Go code in the standard library and elsewhere, and you are encouraged to read the documentation for the `io` package to learn more about them.

2.8.3 Integrating the scanner loop

With the final piece in place, all you have to do is replace the `os.Readfile` call with `os.Open`, pass the file to the scanner,

and start your loop.

Listing 2.42 Altering main.go to use bufio.Scanner

```
func main() {
        if len(os.Args) < 2 {
                log.Println("need to provide filename!")
                os.Exit(1)
        }

        // Open the first argument as a file.
        file, err := os.Open(os.Args[1])
        if err != nil {
                log.Println(err)
                os.Exit(1)
        }

        // Create a scanner to read from the file.
        scanner := bufio.NewScanner(file)

        // Counter to track the running total.
        var wordCount int

        // For each line of the file, get the number of words, and
add it to
        // the total.
        for scanner.Scan() {
                words := strings.Fields(scanner.Text())
                wordCount += len(words)
        }

        if scanner.Err() != nil {
                log.Println(scanner.Err())
                os.Exit(1)
        }

        fmt.Println("Found", wordCount, "words")
}
```

This functions exactly as before, but now it will work for any file, no matter how large it is. That’s pretty cool.

2.8.4 Function types for modifying behavior

There's one other interesting Go feature lurking in the type description for `bufio.Scanner` that can make things even easier. The docs mention that you can change the method used to split the input into tokens by using something called a `SplitFunc`. Looking at the documentation for this type reveals something interesting: the type is neither a struct nor an interface, but a *function*.

Listing 2.43 Excerpt from go doc bufio SplitFunc

```
type SplitFunc func(data []byte, atEOF bool)
        (advance int, token []byte, err error)

        SplitFunc is the signature of the split function used to to
kenize the
    input. The arguments are an initial substring of the remaining
    unprocessed data and a flag, atEOF, that reports whether the Re
ader has
    no more data to give. The return values are the number of bytes
to
    advance the input and the next token to return to the user, if
any, plus
    an error, if any.
```

This is a *function type*, which means that variables of this type can be called just like any other function. The interesting bit is that these variables can hold *any* function that has the same inputs and outputs. This is an extremely powerful language feature that we'll expand on in chapter 5, but the upshot here is that you can plug a variety of different functions into a scanner to get different behavior, while still getting all the benefits of the `Scanner` type.

The `bufio` package even provides a few of these for you, including one called `ScanWords`, which changes the tokens from individual lines to individual words instead.

Listing 2.44 go doc bufio ScanWords

```
func ScanWords(data []byte, atEOF bool)
        (advance int, token []byte, err error)

    ScanWords is a split function for a Scanner that returns each
    space-separated word of text, with surrounding spaces deleted.
It will
    never return an empty string. The definition of space is set by
    unicode.IsSpace.
```

Now the task becomes almost trivially easy, because all you really need to do is add 1 to the running total each time through the loop.

Listing 2.45 Using the ScanWords function instead of Strings.Fields

```
// Create a scanner to read from the file.
scanner := bufio.NewScanner(file)

// Change the split function to split on words instead of lines.
scanner.Split(bufio.ScanWords)

// Counter to track the running total.
var wordCount int

// Add 1 to the count for each word.
for scanner.Scan() {
        wordCount++
}
```

This is a great example of why it pays to check the documentation when working with new packages: often you will find something that can make the task at hand easier or more flexible. Sometimes you'll even learn a new trick, or an entirely different way to do things, such as the `Scanner` loop technique. Documentation is a programmer's best friend and, whether it is on your local machine with `go doc` or online at https://pkg.go.dev/, you should familiarize

yourself with the tools for reading Go documentation and use them often.

2.9 Reading multiple files

Now that you don’t have to worry about how much to read into memory, you can extend the program to read from any number of files in a single run. The basic operation remains the same; instead of reading only one program argument, you read one *or more* of them.

To accomplish this, you can loop over all the program arguments, starting from index 1.

Listing 2.46 Using a range loop to visit multiple items in a list

```
func main() {
        if len(os.Args) < 2 {
                log.Println("need to provide at least one filenam
e!")
                os.Exit(1)
        }

        // Loop over all arguments from index 1 onwards.
        for _, filename := range os.Args[1:] {
                file, err := os.Open(filename)

                // If there's an error, print it and skip to the ne
xt file.
                if err != nil {
                        log.Printf("%s: %v", filename, err)
                        continue
                }
                scanner := bufio.NewScanner(file)
                scanner.Split(bufio.ScanWords)

                var wordCount int

                for scanner.Scan() {
                        wordCount++
                }

                if scanner.Err() != nil {
                        log.Printf("scan error %s: %v", filename, s
canner.Err())
                        file.Close()
                        continue
                }

                file.Close()
                fmt.Printf("%s: %d\n", filename, wordCount)
        }
}
```

This combines a couple of useful features. `os.Args[1:]` is a *slice expression* that lets you start from a particular index,

while the `range` loop allows you to visit each of these items in turn, assigning the value to `filename` for each pass through the loop. We'll discuss slice expressions in more detail in chapter 4.

`range` is a flexible and convenient syntax that lets you avoid checking the limits of your collections, because it automatically terminates after viewing every item in the list. It is also one of the few places in the language where the behavior changes depending on how many values you assign.

Listing 2.47 The two different range loop forms for slices

```
stringList := []string{"A", "B", "C"}

// Single-value form gets just the index.
for index := range stringList {
        fmt.Println(index)
}
// Output:
// 0
// 1
// 2

// Two-value form gets the index and a copy of the value.
for index, value := range stringList {
        fmt.Println(index, value)
}
// Output:
// 0 A
// 1 B
// 2 C

// Discarding the index.
for _, value := range stringList {
        fmt.Println(value)
}
// Output:
// A
// B
// C
```

When you range over a collection, assigning just one variable gives you the index of each element in turn; assigning two variables gives you both the index and a copy of the element's value at that index. If you only want the value and not the index, you can discard the index by assigning it to the blank identifier: _. Range loops can be used on strings, slices and arrays, maps, and channels, and we will touch on these more in chapters 4 and 9.

In the meantime, let's examine the fruits of your labors. If everything has gone well, you should be able to create as many new files as you want and call the program on all of them at once. Try throwing in a nonexistent file for good measure to see how well it holds up.

Listing 2.48 Running the new code on multiple files

```
$ cat words2.txt
This file has a few more words in it, but it's still pretty tame co
mpared to
the collected works of Leo Tolstoy.
But, hey, now that you're scanning the file instead of reading it a
ll into
memory, you can scan whatever you want, without fear. wc is for chu
mps,
re-write everything in Go, this is the way.

$ go run main.go words.txt words2.txt not_there.txt
words.txt: 14
words2.txt: 57
not_there.txt: open not_there.txt: no such file or directory
```

Working great! Now that you have your first working program, try coming up with some other useful things you can do with it. Maybe you want to take more of the market share from `wc` and count lines in addition to words by combining the two scanning approaches. Or maybe you want to count word frequency or look for particular words. It's not too much of a stretch, as long as you handle case and punctuation (hint: check out the godoc for `strings ToLower` and `regexp ReplaceAllString`). Or just keep reading and come back to it later. The world of words is your oyster.

Summary

- Go is a compiled, statically typed language with a simple syntax and a rich feature set.

- Go has great documentation support, and once you know how to look things up, you can find information on types and methods as well as examples to help you get started.
- Go has an extensive standard library that provides a variety of useful features and helps you simplify your code.

3 Primitive types and operators

This chapter covers

- Numeric types and numeric operations
- The `bool` type
- Structs
- Pointers
- Strings, runes, and string manipulation

Go comes with a number of *types* built into the language for some of the most common computing tasks, such as working with numbers, strings, and Boolean types. In a statically typed language like Go, these built-in types are often called *primitive types* because they are the building blocks for all the other types in the language.

Primitive types are the most fundamental tools in your toolbelt when building software with Go, and you will use them in every program you write. If you think of writing software as building a house, the types in this chapter are like the nails and screws that hold everything together.

In this chapter, you'll learn more about these fundamental types, along with some tips for using them most effectively.

3.1 Integer types

Integer types store whole numbers within a certain range, depending on their size. In Go, we have two varieties of

integer—signed and unsigned, which are outlined in table 3.1—and there are several different size variations for each, as well as a couple of alias types that declare alternate names for certain circumstances. Integer types are very common in Go and are used for basic math, array and slice indices, and configuration arguments for many of the standard library functions.

Table 3.1 Predeclared integer types

Type name	Description	Value vange
`int`	Signed integers	32 or 64 bits, depending on your architecture
`int8`	Signed 8-bit integers	-128 to 127
`int16`	Signed 16-bit integers	-32,768 to 32,767
`int32`	Signed 32-bit integers	-2,147,483,648 to 2,147,483,647
`int64`	Signed 64-bit integers	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
`uint`	Unsigned integers	32 or 64 bits, depending on your architecture
`uint8`	Unsigned 8-bit integers	0 to 255
`uint16`	Unsigned 16-bit integers	0 to 65,535
`uint32`	Unsigned 32-bit integers	0 to 4,294,967,295
`uint64`	Unsigned 64-bit integers	0 to 18,446,744,073,709,551,615

Type name	Description	Value vange
`byte`	An alias for `uint8`	
`rune`	An alias for `int32`	

The types that start with `int` are the *signed integers*, which are stored in two's complement format, allowing them to represent positive and negative values. The types that start with `uint` are *unsigned integers*, and these types can store only positive values starting from 0. You might notice that the range of positive values for unsigned integers is larger than for the corresponding signed equivalent. This is because signed integers reserve one of their bits to indicate whether the value is negative. Unsigned integers have a few quirks that make them useful for a narrower range of purposes, which we will discuss later in the chapter.

Types with a numeric suffix, such as `int64` or `uint8`, are *constant-sized* or *architecture-independent* types, and the number indicates the size of each type in bits. These types will always consume the same amount of memory and have the same range of values whether they are run on a 32-bit or 64-bit system.

By contrast, the `int` and `uint` types are *architecture-dependent*, which means they will be either 32 or 64 bits in size, depending on the architecture of the system they are compiled for. These types are provided to give you general-purpose default integer types that will perform optimally, no matter which architecture they are compiled for.

Rounding out the list, `byte` and `rune` are aliases that give more meaningful names to integer values in certain contexts. `byte` is an alias for `uint8` because it is extremely common to refer to data in terms of 8-bit chunks, or "bytes." All of these types are represented in memory in similar ways, as outlined in figure 3.1.

The alias was created to distinguish between when code is dealing with data and when it is dealing with a numeric value. This convention is ubiquitous in the Go ecosystem, and you'll run across many functions in the standard library and elsewhere that either accept or return data as a sequence of bytes (`[]byte`) represented as slice of bytes. Similarly, `rune` is exactly the same as `int32`, except that it is used when talking about Unicode characters in a string. We'll cover this in greater detail when we explore the `string` type in section 3.8.

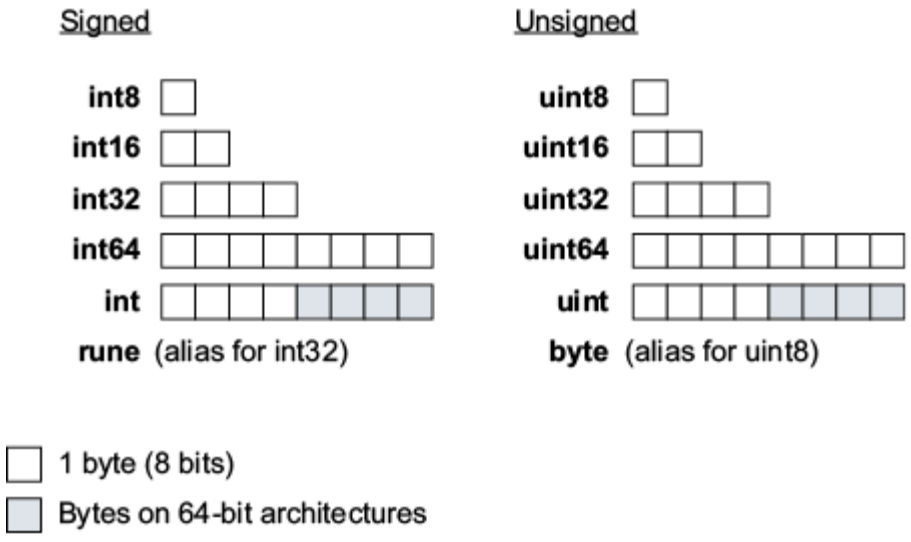


Figure 3.1 Integer type sizes in nemory

As you learned in chapter 2, variables can be declared using the `var` keyword with an optional value, or with the short declaration operator `:=`, which declares the variable and sets the value at the same time. Integers declared with `var` will

be of whatever type you choose and will receive the zero value of `0` if you don't assign one. Integers declared with a short declaration will receive the type `int` along with whatever value you assign.

Listing 3.1 Declaring and initializing integer values

```
// Declare a variable with a type of int (and a default value of
0).
var myInt int

// Declare variable with type int64
var largeInt int64

// Use short declaration to create variable with type int and a val
ue
// of 3.
i := 3

// Use a type conversion if you want to use short declaration with
other
// integer types.
u := uint64(4)
```

The values `3` and `4` in the preceding listing are known as *integer literals*, another term for an integer value appearing directly in the code. Integer literals can be positive or negative and can be written in base 10, hexadecimal, octal, or binary notation, depending on the prefix.

Listing 3.2 Integer declaration with various notations

```
// These are different ways of writing the same value.
decInt := 1000         // Base 10 Notation          (no prefix)
hexInt := 0x3E8        // Hexadecimal Notation      (Prefix: "0x")
octInt := 01750           // Octal Notation            (Prefix:
"0")
octInt = 0o1750        // Alternate Octal Notation  (Prefix: "0o")
binInt := 0b1111101000 // Binary Notation           (Prefix: "0b")

// Optional underscores can be used to separate digits for readabil
ity.
withSep := 1_000
hexWithSep := 0x3_E_8

// Notation is purely cosmetic, and does not affect value:
fmt.Println(decInt, hexInt, octInt, binInt, withSep, hexWithSep)
// Output: 1000 1000 1000 1000 1000 1000

// Negative integer literals:
negativeInt := -10
negativeHexInt := -0xa
```

WHAT ARE LITERALS?

A *literal* or *constant value* is an explicit, fixed value that appears in your source code. Literals indicate that the value being assigned is specified in the source, as opposed to coming from somewhere else, such as a variable or function call. When you make an assignment such as `myInt := 5` or `myString = "hello"`, the `5` and the `"hello"` are literals.

3.1.1 Choosing an integer type

With so many integer types to choose from, it helps to know which types you might want to use for what purposes. For most tasks, you will probably want to reach for `int` first,

since it can represent both positive and negative values and will be the most efficient integer type for your architecture. It will also mean less work for you in terms of conversions, because it is

- The default type for integer literals
- The most common integer type in standard library functions and most third-party packages
- The type returned by built-in functions such as `len()` and `cap()`
- The type used for indices in for-range loops

`int64` is the largest native integer type in the language, and you'll find it used in the standard library for values that need the greatest range, such as timestamps, file offsets, and size limits. If you need the full range of values provided by `int64`, or if you will be making many calls to functions that take an `int64` as an argument, it can sometimes be helpful to have a value of this type for compatibility. Otherwise, you can just convert from `int` as needed.

As for the rest of the signed types (`int32`, `int16`, and `int8`), consider them on an as-needed basis if they are required for compatibility or if there are measurable performance reasons for using them. If you don't *know* you need them, you probably don't. Try to avoid using another type just because it "takes less memory," since this is a form of premature optimization and will come with extra overhead in type conversions to interoperate with more common types. When in doubt, run benchmarks and profile your programs to ensure that your choice of types is really the problem (usually it's not).

Unsigned integers can be a bit trickier to use correctly because they can easily "wrap around," especially when values are small. You can see this in action by subtracting

from an unsigned integer to create a value that would otherwise be negative.

Listing 3.3 Subtracting from an unsigned integer

```
var u uint64 // 0
u = u - 1

fmt.Println(u) // output: 18446744073709551615
```

All of a sudden, the result is much larger! This happens because the value of `u` is already as low as it can go, so subtracting causes the bits to "wrap around" from 0 to the maximum value the type can hold. You can think of this as a bit like an odometer in reverse: once the value bottoms out at 0, the bits roll back around again. Figure 3.2 illustrates this as being similar to an odometer.

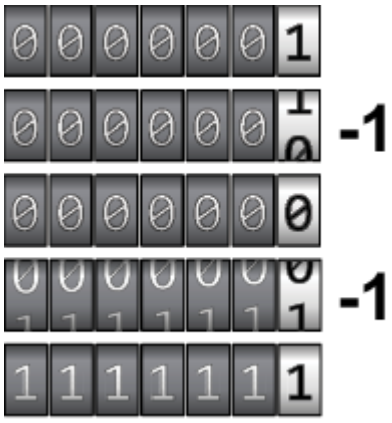


Figure 3.2 Unsigned integer wraparound

This behavior is common to all integer types when they reach the limits of the values they can contain, but it's much more likely with unsigned integers because they start at the edge of their range.

A similar condition can arise when exposing unsigned integers as function arguments. It can be tempting to declare functions that take unsigned integers to indicate

that only positive values are accepted, but doing so can have unintended consequences.

Consider the following function, which takes a `uint` to try to ensure that it accepts only positive values.

Listing 3.4 Function taking an unsigned int

```
// OnlyPositive only accepts positive values.
func OnlyPositive(u uint) {
        fmt.Println("positive value is", u)
}
```

If you call the function with a negative literal value, this will appear to work as intended because the compiler can recognize that a negative value cannot go into an unsigned variable.

Listing 3.5 Calling an unsigned int function with an explicitly negative value

```
func main() {
        OnlyPositive(-1)
}
// error: cannot use -1 (untyped int constant) as uint value in arg
ument to
// OnlyPositive (overflows)
```

But functions are not always called with literal values. You may have a situation where a user wishes to pass an `int` value to your function, which means they need to convert their `int` to a `uint` first. This is where it can go horribly wrong.

Listing 3.6 Calling a function that takes a uint with a converted negative value

```
func main() {
        // User has a signed integer value from somewhere else that
is
        // unintentionally negative.
        configVal := GetConfigValue() // returns int(-1)

        // They need to convert it to call your function, but wai
t...
        OnlyPositive(uint(configVal))
}
// output: positive value is 18446744073709551615
```

All of a sudden, it's the maximum value again! This time it's because the signed value passed is negative, and negative signed values contain a lot of 1s, thanks to two's complement, so you're back in a similar situation, only it's much harder to spot ahead of time. Bugs like this can be tough to find once they bite you.

> **NOTE** The full particulars of two's complement are beyond the scope of this book, but you're encouraged to look it up if you're curious. Go is far from the only language to use this technique, and this potential pitfall is shared by all languages that represent numbers in this way.

To avoid this pitfall, it's better to take an `int` and handle negative values yourself. The simplest option is to return an error if a negative value is unacceptable, along with helpful documentation on the function so that the user knows the score.

Listing 3.7 Checking for a negative value and returning an error

```
// BetterOnlyPositive only accepts positive values. An error will b
e
// returned if a negative value is passed.
func BetterOnlyPositive(i int) error {
        if i < 0 {
                return fmt.Errorf("value must be positive!")
        }
        fmt.Println("positive value is", i)
        return nil
}

func main() {
        configVal := GetConfigValue() // returns int(-1)

        if err := BetterOnlyPositive(configVal); err != nil {
                // Now you can detect the error!
                log.Fatal(err)
        }
        // Output (failure):
        // value must be positive!
}
```

Another technique is to give negative numbers a special meaning, if one applies to your use case. You can find this technique in the standard library for functions that perform an action some number of times, with negative numbers meaning "unlimited" or "as many times as necessary." An example is `strings.Replace`, which uses negative numbers to mean "replace every occurrence."

Listing 3.8 go doc for strings.Replace

```
func Replace(s, old, new string, n int) string
    Replace returns a copy of the string s with the first n non-ove
rlapping
    instances of old replaced by new. If old is empty, it matches a
t the
    beginning of the string and after each UTF-8 sequence, yielding
up to
        k+1 replacements for a k-rune string. If n < 0, there is no
limit on the
        number of replacements.
```

> **TIP** Do not use unsigned integers to mean "I don't accept positive values." Instead, use a signed integer and either return an error if it's negative or use negative numbers as a "special" value, if your use case supports it.

This doesn't mean you shouldn't use unsigned integers, though! One place unsigned integers are useful is in bitwise operations, where they are treated as large bit fields and you don't have to worry about the sign bit. Unsigned integers are much more predictable when shifting bits around, since they always shift in a `0` because we are not concerned about whether this is a positive or negative number, while signed integers have special behavior when shifting from the right and will use whatever their sign bit happens to be (this is another two's complement behavior). Figure 3.3 outlines how this works.

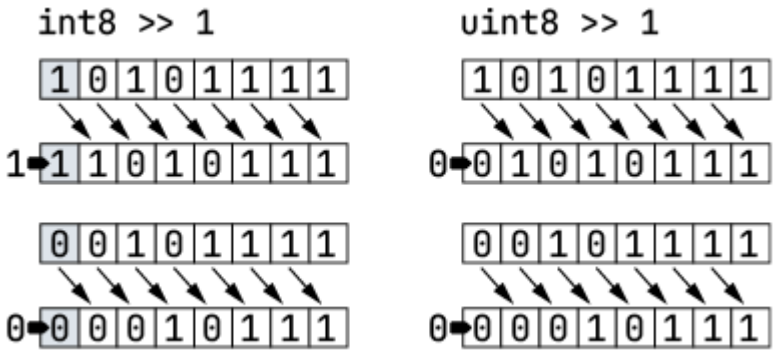


Figure 3.3 Right shifting in signed and unsigned integers

When doing bitwise operations on unsigned values, the opposite advice applies to picking a signed type: you will probably want to stick to sized variants for these kinds of operations. This is because they have exact sizes, which is almost always what you want when doing bit manipulation, since it models the data in memory more precisely. The `uint` type doesn't provide this, because its length changes depending on the architecture, so you won't have as much use for this type.

3.2 Floating-point number types

Floating-point types are used when you need a number with a fractional (or decimal) part, such as when calculating percentages or doing scientific calculations. These numbers can represent a greater range of values than integers, at the cost of some precision.

Go provides two types that implement the IEEE-754 standard for binary floating-point number representation outlined in table 3.2.

Table 3.2 Predeclared floating-point types

Type name	Description
`float32`	Single-precision floating-point numbers
`float64`	Double-precision floating-point numbers

`float32` takes up 32 bits of memory, while `float64` takes up 64 bits, giving it greater precision. Figure 3.4 illustrates the difference in memory allocation. If you're coming from C or Java, these are equivalent to the `float` and `double` types, respectively.

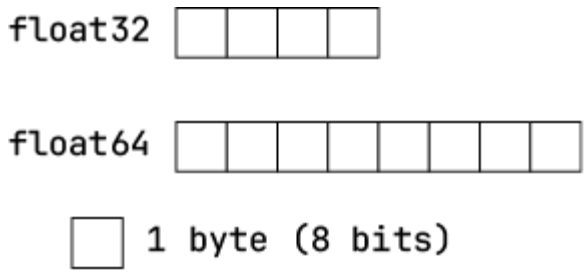


Figure 3.4 Floating-point type sizes in memory

As with integer declarations, floating-point variables declared with `var` get a value of `0` if no value is specified, while floating-point numbers declared with a short declaration get a type of `float64` by default.

Listing 3.9 Declaring and initializing floating-point types

```
// Declare a variable doubleFloat with type float64 (and a default
// value of 0).
var doubleFloat float64

// Declare a variable with type float32.
var singleFloat float32

// Use short declaration to create a variable f with type float64 a
nd a
// value of 12.1.
f := 12.1

// Use a type conversion if you want to use short declaration with
float32.
g := float32(12.1)
```

WHY ISN'T THERE A PLAIN FLOAT TYPE?

You may have noticed that, while Go provides architecture-dependent `int` and `uint` types, there is no corresponding architecture-dependent type for floating-point numbers. This is because the bit size of floating-point numbers is very important for precision, so early in the evolution of the language, it was decided to require an explicit choice. This is also why there is no `complex` type: complex numbers are made up of a combination of two floating-point values.

To assign floating-point values using literals, you can add a decimal point (`.`) to tell the compiler that you want a floating-point value instead of an integer. You can leave out either the integer or fractional part if you want, so `1.0`, `1.`, `0.1`, and `.1` all work. Another option is to use scientific notation by adding the `e` character to your number, followed by a positive or negative integer exponent.

Listing 3.10 Floating-point literal notations

```
// The following are different ways to write the same value.
floatVal = 12.          // fractional part is optional
floatVal = 12e0         // decimal point not required with exponent
floatVal = 012.         // leading zero is fine
floatVal = .12e+2       // integer part is optional
floatVal = 1_2.         // underscores allowed in floats, too
floatVal = float64(12) // explicit type for otherwise integer liter
al

fmt.Println(floatVal, myFloat)
// Output: 12 12

// Floats can be negative.
negativeFloat := -12.0

fmt.Println(negativeFloat)
// Output: -12
```

> **NOTE** The easiest way to use an integer value as a floating point is to add a decimal point at the end (`12.` instead of `12`), but you can also wrap the literal with a floating-point type if you want to be more explicit, like so: `x:=float64(12)`.

3.2.1 Special floating-point values

The standard Go compiler does not do any special checks for floating-point math in cases of overflow, division by zero, or operations that are not mathematically valid, such as taking the natural logarithm of a negative number. In these situations, the value will be set as shown in table 3.3 (these values are defined by the IEEE-754 standard).

Table 3.3 Special floating-point values

Value	Description
+Inf	Positive infinity
-Inf	Negative infinity
NaN	Not a number

These values can be checked with the `math.IsInf` and `math.IsNaN` functions, respectively.

Listing 3.11 Checking for special floating-point values

```
f := 2.0
// Make an "infinitely huge" number.
posInf := math.Pow(f, 10_000)
fmt.Println(posInf) // output: +Inf

// Make an "infinitely small" number.
negInf := f - math.Pow(f, 10_000)
fmt.Println(negInf) // output: -Inf

// The second argument to math.IsInf lets you check for a specific
infinity:
// <0: negative infinity
// >0: positive infinity
// 0: either infinity
fmt.Println(math.IsInf(posInf, 0))  // true
fmt.Println(math.IsInf(negInf, 1))  // false
fmt.Println(math.IsInf(negInf, -1)) // true

// Make an invalid number.
notANumber := math.Log(-f)
fmt.Println(notANumber) // output: NaN

fmt.Println(math.IsNaN(notANumber)) // true
```

> **TIP** Make sure you use the checks in the `math` package if there's a possibility that you could generate floating-point numbers that are invalid or out of range.

3.2.2 Choosing a floating-point type

For floating-point operations, you should pick `float64` unless you have a compelling performance reason to do otherwise. Most modern hardware, including 32-bit architectures, has special support for 64-bit floating-point numbers, so there is little reason to sacrifice the greater precision by using `float32`.

As with integer types, you should use `float32` only if you are trying to decrease memory consumption or improve performance and have determined that using `float64` is a significant bottleneck.

3.3 Complex number types

Complex numbers are a type of value that has both real and imaginary parts; they are useful in calculations for physics, signal analysis, and other fields. Many languages implement complex numbers in special libraries, but Go has them as native types. Complex types in Go are made up of two floating-point values representing the real and imaginary parts of the number, which makes them double the size of the float type used to implement them (see table 3.4). Figure 3.5 shows a size comparison in memory for these types.

Table 3.4 Predeclared Complex Types

Type Name	Usage
`complex64`	Complex numbers with `float32` real and imaginary parts
`complex128`	Complex numbers with `float64` real and imaginary parts

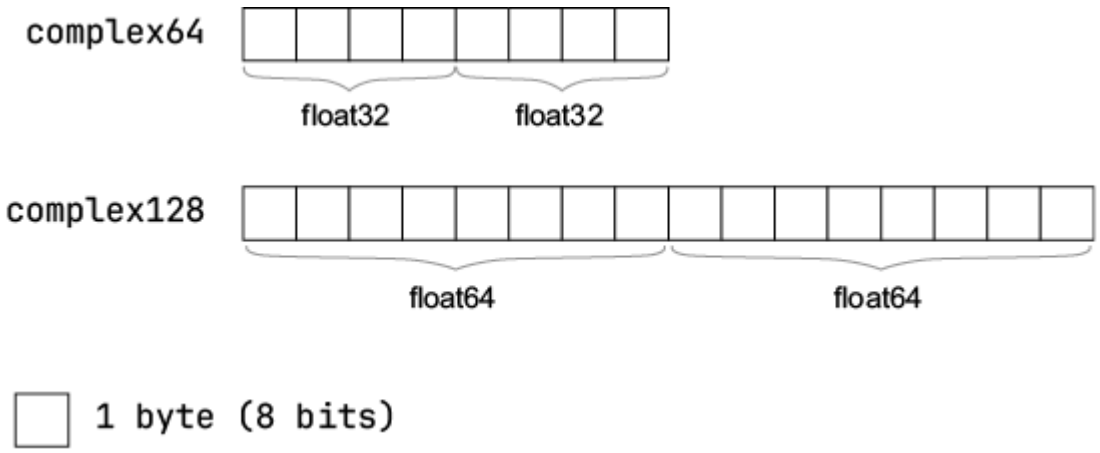


Figure 3.5 Complex type sizes in memory

Complex numbers are a bit different from the other primitive types because they need to be declared using either the special built-in `complex()` function or special complex expressions.

The `complex()` function takes two arguments, which are treated as float values and determine the type of the resulting complex value. If the values are `float64` or literals, the result has the type `complex128`. If the values are `float32`, the result is a `complex64`. Types cannot be mixed.

Listing 3.12 Creating complex values using the built-in complex() function

```
cmplxValue1 := complex(1.1, 2.2)                    // complex128
cmplxValue2 := complex(float32(1.1), float32(2.2)) // complex64

cmplxValue3 := complex(1, 2)       // complex128, equivalent to 1+2
i
cmplxValue4 := complex(0, -3)      // complex128, equivalent to 0-3
i
cmplxValue5 := complex(-3.3, -4.4) // complex128, equivalent to -3.
3-4.4i
```

Complex expression declarations take the form of an optional numeric value, which can be a float or an integer literal, followed by a `+` or `-`, and then an imaginary literal, which is a numeric value followed by the `i` character.

Listing 3.13 Creating complex values with complex expressions

```
cmplxValue1 = 1.1 + 2.2i  // equivalent to complex(1.1, 2.2)
cmplxValue3 = 1 + 2i      // equivalent to complex(1,2)
cmplxValue4 = -3i         // equivalent to complex(0, -3)
cmplxValue5 = -3.3 - 4.4i // equivalent to complex(-3.3,-4.4)
```

Go also has special built-in `real` and `imag` functions that can be used to extract the floating-point components that make up a complex number. In essence, these are the inverse of the `complex` function and return `float32` or `float64` values, depending on the type.

Listing 3.14 Using the real() and imag() functions

```
c1 := complex(1.1, 2.2)
fmt.Println(real(c1), imag(c1)) // "1.1 2.2"
```

Complex types in Go are rarely used on their own because most of the useful functions for manipulating them reside in the `math` and `math/cmplx` packages. Additionally, most uses of complex numbers are relatively niche and… well, honestly,

they’re pretty complex. Most programmers could go years without using them even once, but it’s important to know that they exist.

3.4 Mathematical operators

Numeric types are kind of boring if you can’t do math with them, so Go defines all the basic arithmetic operations you’d expect (listed in table 3.5), along with bitwise operations for manipulating the individual bits of integer values.

Table 3.5 Binary Arithmetic Operators

Operator	Description	Types
+	sum	integers, floating-point, complex, strings
-	difference	integers, floating-point, complex
*	product	integers, floating-point, complex
/	quotient	integers, floating-point, complex
%	remainder	integers

With the exception of the remainder (or modulo) operator %, binary arithmetic operators apply to all numeric types and return the corresponding types of their operands. Importantly, operations must be performed on values of the same type or on literals that can represent that type.

Listing 3.15 Examples of arithmetic in Go

```
        a := 7
        b := 3

        var i int
        var c int
        i = a + b // 10
        i = a - b // 4
        i = b - a // -4
        i = a * b // 21
        i = a % b // 1

        // Divide by zero can be caught by the compiler if the 0 is
a literal.
        //c = a / 0  // error: invalid operation: division by zero

        // Otherwise, divide by zero is a runtime panic.
        zero := 0
        c = a / zero // panic: runtime error: integer divide by zer
o

        // Float divide by zero does not panic and will result in a
n infinite
        // value.
        f := 1.0
        f = f / float64(zero) // +Inf

        u := uint64(1)
        u = u + i // error: mismatched types uint64 and int
        // Type conversion is needed to allow different types to wo
rk together.
        u = u + uint64(i) // 2
```

Bitwise operations operate only on integers and will be familiar if you have worked with them in other languages (see table 3.6). The exception is the bit clear operator `&^`, which is unique to Go and can be thought of as the opposite of bitwise OR (`|`): all of the `1` bits in the second operand are set to `0` in the result.

Table 3.6 Bitwise operators (integers only)

Operator	Description
`&`	AND
`|`	OR
`^`	XOR
`&^`	Bit clear
`<<`	Shift left
`>>`	Shift right

3.5 The bool type

Go has a dedicated `bool` type for representing Boolean (true/false) values. Values of this type can have one of two predeclared values, `true` or `false`, and have a zero value of `false`. `bool` is also the default type for Boolean expressions, so short variable declarations such as `value := x ⇐ y` create variables of this type.

Listing 3.16 Examples of Boolean values

```
var boolVar bool // Defaults to false

// Boolean variables can be used for conditionals, either
// on their own, or as part of a boolean expression. In this case,
// boolVar has not been explicitly set to anything yet, so its valu
e is
// false, and the if block will not run.
if boolVar {
        fmt.Println("boolVal is true.")
}

// Once boolVar is set to true, the same if condition will pass and
the
// block will run.
boolVar = true
if boolVar {
        fmt.Println("NOW boolVal is true!")
}

// Short declaration variables assigned to boolean expressions will
be
// given the bool type.
x := 0
y := 1
v := x < y // Type: bool, value: true.
```

3.5.1 Boolean functions

Booleans are used primarily in conditional logic, such as `if` statements, often as the return value for a helper function intended to give you a yes-or-no answer about a particular condition or a meaningfully named piece of data. For example, if you wanted to print all of the even items from a list of integers, you might write a helper function called `IsEven` that would take an integer value and return a Boolean indicating whether the number was evenly divisible by two.

Listing 3.17 A Boolean helper function for use in conditional logic

```
func IsEven(i int) bool {
        return i%2 == 0
}

func main() {
        numbers := []int{1, 2, 3, 4}
        for _, n := range numbers {
                if IsEven(n) {
                        fmt.Printf("%d is even.\n", n)
                }
        }
}
```

Although you could easily replace `IsEven` with a simple Boolean expression (`if n%2 == 0 {}`) and not lose much clarity, more complex tests, or tests that might depend on several pieces of information, will benefit greatly from having a dedicated function. A function like this allows you to keep all of the logic in one place without repeating it and to add logic to that function later if necessary without changing the calling code.

For example, you might want to write a function to tell you whether a particular time is during normal working hours. This would depend on a couple of pieces of information, such as the day of the week and the time of day. By putting this logic into a function called `IsWorkHours()`, which returns a `bool` value, you could then make determinations about things like sending a push notification versus an email.

Listing 3.18 Using a `bool` as a return value for a more complicated query

```
// IsWorkHours returns true if the given time falls within standard
working
// hours.
func IsWorkHours(t time.Time) bool {
        switch t.Weekday() {
        case time.Saturday, time.Sunday:
                return false
        }
        if t.Hour() < 9 || t.Hour() > 17 {
                return false
        }
        // TODO: Add holiday tracking
        return true
}

func main() {
        if IsWorkHours(time.Now()) {
                fmt.Println("During working hours - sending a push
notification.")
                // notification sending code
        } else {
                fmt.Println("Outside of working hours - sending an
email.")
                // email sending code
        }
}
```

By putting all of this logic in `IsWorkHours`, you can use the function in a conditional for any actions you want to take only during working hours. It could also be extended later to take holidays into account, as marked by the TODO.

3.6 Struct types

Struct types are a useful way to bundle several related values into a single unit. Each struct is defined as a set of named values, called *fields*, each of which can be accessed separately.

Structs are declared using the `struct` keyword, followed by a list of fields enclosed in brackets.

Listing 3.19 An example `struct` type

```
var person struct {
        name string
        age  int
}
```

Fields can then be assigned and retrieved by name using dot notation.

Listing 3.20 Setting and retrieving struct values with dot notation

```
andy.name = "Andy"
andy.age = 42
fmt.Println(andy.name, andy.age) // output: Andy 42
```

Struct variables can also be initialized with a short declaration using *struct literals*, which list the name and value for each field.

Listing 3.21 Declaring a `struct` with struct literals

```
james := struct {
        name string
        age  int
}{
        name: "James",
        age:  25,
}
fmt.Println(james.name, james.age) // output: James 26
```

There are a couple of things to note here. First, the trailing comma after the value for `age` is required. This is different from what you might expect if you have experience with JavaScript. In Go, it is always required, which makes it easier to rearrange and add lines without worrying about

removing or adding commas, and it has the nice side effect of reducing the noise in diffs!

The second thing to note is that this is *ugly* and tedious. While it might be useful for declaring one-off structs, no one wants to clutter their code with a bunch of random struct definitions if they can help it!

For this reason, structs often take the form of *types*, which allows them to be reused and passed around.

3.6.1 Structs as types

Using a structure type like this lets you give a definition a meaningful name and reuse it multiple times. With `person` declared as a type, declaring a variable is as simple as providing the type and the values for each field, which is much nicer.

Listing 3.22 Using a type in a declaration

```
type person struct {
        name string
        age  int
}

andy := person{
        name: "Andy",
        age:  42,
}
james := person{
        name: "James",
        age:  26,
}
fmt.Println(andy.name, andy.age)   // output: Andy 42
fmt.Println(james.name, james.age) // output: James 26
```

Structs and types form the basis of user-defined types in Go, and this example is just the very beginning of what's

possible. Check out chapter 5 for more information about declaring types in Go.

3.7 Pointer types

Pointers are special types that represent the *location* of data, rather than the data itself. They provide a way to share memory across different parts of a program without needing to copy the data. This can be very useful for passing around large or complex data structures, such as those representing a connection to a database or collections of data. Pointers play a large role in Go because they are the basis for the map and slice types and they're also a fundamental part of the type system.

To define a pointer variable with `var`, add a `*` before the type to tell the compiler which type your variable will point to.

Listing 3.23 Pointer variable declaration

```
var intPtr *int //intPtr is a pointer(*) to an integer value(int)
```

As with all declarations, pointer values start out with a zero value when they are initialized. The zero value for pointers is the special variable `nil`, indicating that the pointer does not yet point to a value, as illustrated in figure 3.6.

Figure 3.6 An uninitialized pointer variable

To get an address to store in a pointer, you need to use the *address operator*, `&`, which returns the memory address of the value it precedes, as shown in figure 3.7. This address

value will be of the type `*T`, where `T` is the type of the object.

```
intPtr := &intValue
```

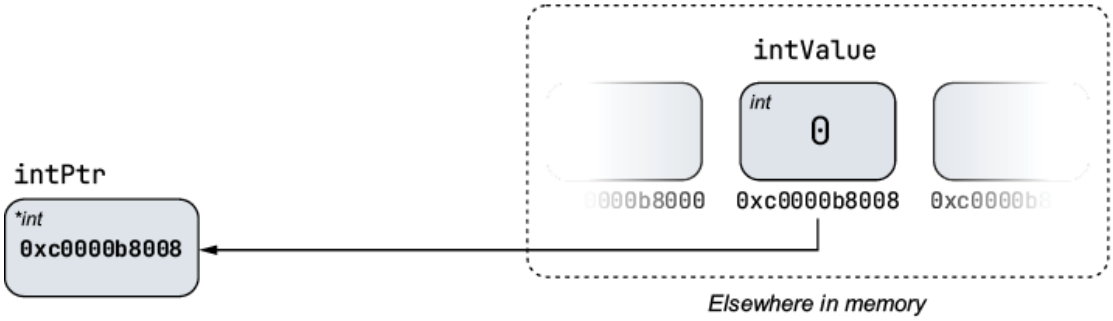


Figure 3.7 The address operator returns the memory address of a variable.

Listing 3.24 Assigning the address of a value to a pointer

```
// declare an integer variable
intValue := 0

var intPtr *int

// get the address (&) of the variable to store in the pointer
intPtr = &intValue

// when the address operator is used with short declaration,
// the new variable will automatically be the appropriate pointer t
ype
intPtr2 := &intValue //intPtr2 is type *int
```

By convention, we say that the pointer variable "points to" the variable it references, and we represent this with an arrow throughout in this book for simplicity (and to avoid drawing a bunch of variables with random addresses in them). This is shown in figure 3.8.

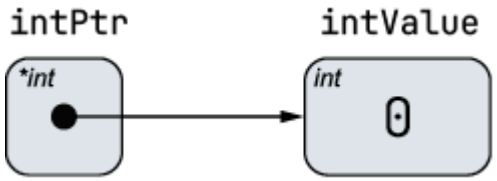


Figure 3.8 intPtr points to intVar

The address operator is used whenever you need to get the location of a value in memory, and, with a few exceptions, you can get a pointer to almost anything.

3.7.1 Dereferencing pointers

Once it's assigned, the value of a pointer is just the address it points to, which you can see if you try to print the value of `intPtr`.

Listing 3.25 A pointer variable just holds an address

```
fmt.Println(intPtr) //output: 0x14000122008#1
```

#1 The actual value will be different for every run of the program, but it will be something like this.

To access the actual value being pointed to, you need to prefix the pointer variable with a `*`, which is known as *dereferencing*. This allows you to access the value at that location in memory and even modify it if you wish.

Listing 3.26 Dereferencing a pointer to access or modify its contents

```
fmt.Println(*intPtr) //output: 0
// You can also use dereferencing to assign to the value.
*intPtr = 1
fmt.Println(intValue) //output: 1
```

Dereferencing, in effect, follows the reference to get the actual value behind it, as shown in figure 3.9.

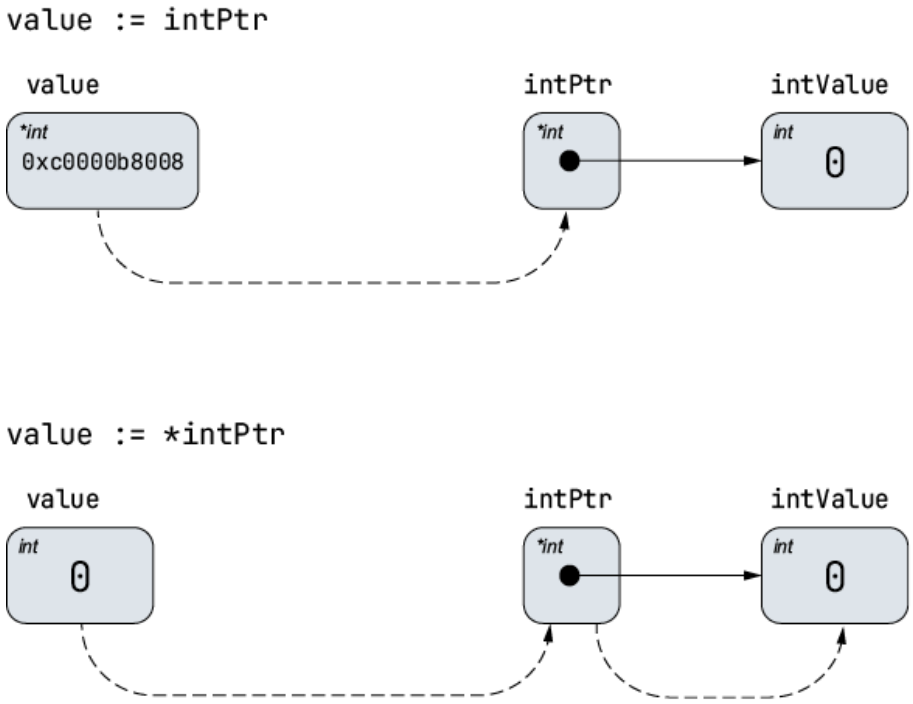


Figure 3.9 Pointer dereference vs. pointer value

> **IMPORTANT** To refer to the actual value being pointed to, use `*variableName`. To get the pointer address value, use `variableName`.

3.7.2 Creating pointers with new and literals

Pointers can also be created without an intermediate value by using the address operator directly with a literal or by using the built-in `new` function.

Listing 3.27 Creating pointers directly

```
type person struct {
        name string
        age  int
}

// These are the same.
newPerson := new(person)
newPerson = &person{}
```

The last two lines in the preceding listing are functionally identical, but the second form is preferred when working with struct types because it also allows you to initialize struct fields while creating the pointer.

Listing 3.28 Creating a pointer with struct initialization

```
newPerson = &person{
        name: "Andy",
        age: 42,
}
```

This is the more flexible style of pointer initialization for most types, and it's the one you'll use most of the time instead of `new`. The main reason to still use `new` is that it is the only way to initialize pointers to built-in types such as `int`.

Listing 3.29 Using new to create a reference to a fundamental type

```
newIntPtr := &int    // This is not allowed
newIntPtr := &int(1) // Neither is this

newIntPtr := new(int) // New is required for fundamental types.
```

3.8 The string type

Textual data is important for programming languages because it allows you to display and process messages in a variety of languages. Go supports this through a combination of robust Unicode support and a dedicated `string` type.

`string` is a relatively simple data type in Go. It's essentially a read-only array of bytes designed for holding text. What makes it different from the other types you've learned about in this chapter is how the bytes are interpreted for display and what happens when you modify the values.

3.8.1 Interpreted strings

Go has two types of string literals, both of which can be assigned to `string` variables. The first is the *interpreted string*, which uses the double-quote (`""`) style common to most other languages.

Listing 3.30 Declaring a string in Go

```
basicStr := "Hello, Gophers!"
```

Because Go source code supports UTF-8, any printable character is valid in a string.

Listing 3.31 Declaring a string with Unicode characters

```
unicodeStr := "你好，地鼠!"
```

Interpreted strings also support a variety of escape codes, allowing you to represent nonprintable characters such as tabs and newlines or to reference Unicode characters by their codepoint number.

Listing 3.32 Strings with escape sequences

```
strWithEscapes := "Hello\n\u5730\u9F20"
fmt.Println(strWithEscapes) // output: Hello<newline>地鼠
```

3.8.2 Raw string values

The second type of string is enclosed with backtick characters (``) and is known as a *raw string*. Raw strings may span multiple lines and are interpreted literally, without any escape codes whatsoever. They also support raw Unicode characters.

Listing 3.33 Declaring a raw string

```
rawString := `Hello\n\u5730\u9F20`
fmt.Println(rawString)

rawStrWithNewlines := `I can
span विभिन्न lines`
fmt.Println(rawStrWithNewlines)
```

Listing 3.34 Raw string output

```
Hello\n\u5730\u9F20
I can
span विभिन्न lines
```

3.8.3 String operations

Strings can be compared using the standard comparison operators, with "less than" and "greater than" calculated from the byte values of the strings.

Listing 3.35 String operations

```
fmt.Println("hello" == "hello") // true

fmt.Println("地鼠" == "\u5730\u9F20") // true, these escape codes ar
e the
// same characters

fmt.Println("地鼠" == `\u5730\u9F20`) // false, escape codes do not
 interpret
// in raw strings

fmt.Println("abc" < "cba")   // true
fmt.Println("andy" > "bill") // false
```

Strings can also be concatenated with the + operator.

Listing 3.36 Basic string concatenation

```
strHello := "hello"
strGophers := " Gophers!"
strHello += strGophers
fmt.Println(strHello) // output: "hello Gophers!"
```

3.8.4 len() and Unicode characters

As we touched on in the last chapter, a `string` variable is essentially just a length value coupled with a pointer to a backing array. You can use the built-in `len` function to get the length of the string in bytes, and if your string is entirely ASCII characters, this works exactly as you'd expect.

Listing 3.37 Using len on an ASCII string

```
asciiCharStr := "easy, right?"
fmt.Println(len(asciiCharStr)) // output: 12
```

Things get a little different once Unicode characters enter the mix, though.

Listing 3.38 Using len on a Unicode string

```
unicodeCharStr := "地鼠"
fmt.Println(len(unicodeCharStr)) // output: 6
```

All of a sudden, the length doesn't match up! What looks like two characters actually has a length three times that.

This is one of the most common head-scratchers people have when first experimenting with strings in Go, and the answer involves a surprising truth: strings are not actually arrays of *characters* but arrays of the bytes that *represent* those characters in UTF-8.

Without getting too far into the weeds on UTF-8 encoding in Go, the simple answer is that not all characters use the same number of bytes for representation. When you create a string in Go that contains characters beyond basic ASCII characters (that is, beyond the basic English letters and numbers), Go transparently encodes that string into UTF-8 and stuffs those bytes into the backing array for your string.

You can get a sense of the size of different characters by using `fmt.Printf` with the special `%x` (hex) verb to visualize the bytes that make up your string.

Listing 3.39 Print several single-character strings of different lengths

```
fmt.Printf("%x\n", "A") // output: 41
fmt.Printf("%x\n", "À") // output: c380
fmt.Printf("%x\n", "Ă") // output: e1beb8
fmt.Printf("%x\n", "🅰") // output: f09f85b0
```

The output of these lines shows that characters can be anywhere from 1 to 4 bytes in total length, depending on the character, as illustrated in figure 3.10. As you can see, `"A"` is just one byte long, exactly as it would be in ASCII.

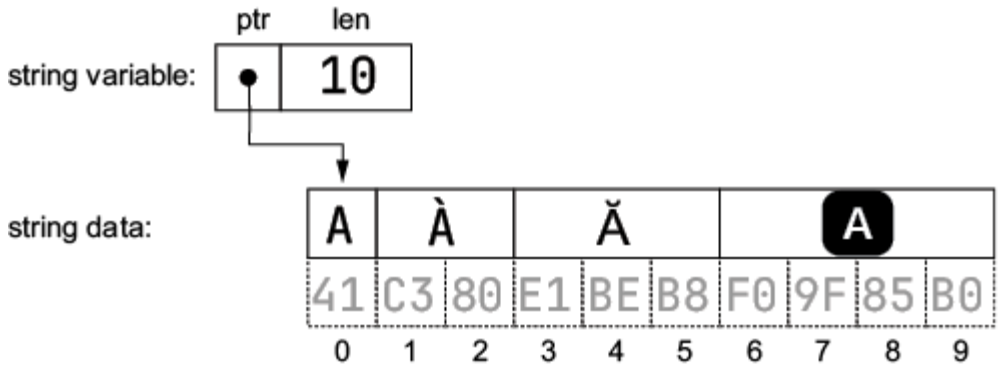


Figure 3.10 Visualizing character size in bytes

This is no coincidence, because UTF-8 is backward-compatible with ASCII, which is why you can iterate over a

basic ASCII string with a traditional `for` loop, and things work as you'd expect.

Listing 3.40 Getting ASCII bytes as strings with a for loop

```
asciiCharStr := "easy, right?"
for i := 0; i < len(asciiCharStr); i++ {
        fmt.Print(string(asciiCharStr[i]) + " ")
}
fmt.Println()
// output: e a s y ,   r i g h t ?
```

But try that with a string containing Unicode, and you're going to have a bad time.

Listing 3.41 Attempting to get Unicode bytes as strings with a for loop

```
unicodeCharStr := "地鼠"
for i := 0; i < len(unicodeCharStr); i++ {
        fmt.Print(string(unicodeCharStr[i]) + " ")
}
fmt.Println()
// output:  å  ° é ¼
```

To access the individual characters, another approach is needed.

3.8.5 Iterating Unicode strings using a range loop

As a special case for strings, you can use a `range` loop to get a list of characters one at a time.

Listing 3.42 range loop on a string

```
unicodeCharStr := "地鼠"
for i, r := range unicodeCharStr {
        fmt.Printf("%d:%s ", i, string(r))
}
fmt.Println()
//output: 0:地 3:鼠
```

This breaks the string up into its constituent characters, regardless of their size. You still get the index where each character starts, but instead of individual bytes, you get one character at a time as a `rune` value.

3.8.6 The rune type

Earlier in the chapter, we mentioned that the `rune` type is an alias for `int32` for string characters. This gives you a type that is large enough to represent any single character you might find in a string and a way to compare those characters with each other.

Runes also separate characters from their encoding because they represent the value of the Unicode character itself. Once you have a character's rune, you can get more information about it using the `unicode` and `unicode/utf8` packages.

Listing 3.43 Using the unicode and unicode/utf8 packages to inspect characters

```
interestingCharacters := "À🅰"
for _, r := range interestingCharacters {
        fmt.Printf("rune: %s byte length: %d\n", string(r), utf8.Ru
neLen(r))
        if unicode.IsLetter(r) && unicode.IsUpper(r) {
                fmt.Println("rune is an uppercase letter")
                fmt.Println("lowercase is:", string(unicode.ToLower
(r)))
        }
        if unicode.IsSymbol(r) {
                fmt.Println("rune is a symbolic character")
        }
}
```

This can provide some interesting insights into the characters in a string.

Listing 3.44 Output of Unicode character information

```
rune: À byte length: 2
rune is an uppercase letter
lowercase is: à
rune: 🅰 byte length: 4
rune is a symbolic character
```

Check out the documentation for these packages for more useful information.

3.8.7 Converting a string to a slice of runes

The `rune` type also gives you another way to get at the characters of a string: convert the string directly into a slice of runes. This will give you similar results, except that now the index represents the character position in the printed string. A string is just a collection of runes, so Go can easily convert between them, as demonstrated in the following listing.

Listing 3.45 Converting a string to a []rune

```
unicodeCharStr := "地鼠"
characters := []rune(unicodeCharStr)
for i, r := range characters {
        fmt.Printf("%d:%s ", i, string(r))
}
fmt.Println()
//output: 0:地 1:鼠
```

Summary

- Go provides a number of built-in primitive types for common computing tasks.
- There are several different integer types, but `int` is the most flexible and the one you should use for most tasks.
- `bool` types are useful for constructing conditional logic and can make complex checks easier when they're used as return values.
- Struct types group related values together and are used to define new types.
- Go supports pointer types, allowing you to pass references to data without copying it.
- Strings are read-only arrays of bytes that can contain UTF-8 encoded characters from any language.

4 Collection types

This chapter covers

- Ordered collections of data with arrays
- Dynamic ordered collections using slices
- Slice manipulation and slice expressions
- Key-value data with maps

Programming would be very difficult if we could only ever deal with one piece of data at a time. That's why we have collection types, which help us gather items together.

Go has three different collection types: arrays, slices, and maps. Arrays represent fixed-size lists like those you might find in C or C++, while slices take this a step further and provide an efficient dynamic array type. If you've ever wanted a flexible list of items that can expand as needed to suit dynamic data at runtime, slices have your back.

If you need more flexibility, the native map type allows you to store any type of data using a key of any type. If you want to store items by name, size, or any other index, maps are a great solution.

The best part is that these data structures are baked into the language, so they can be used anywhere in your program at any time, without needing to import a special package.

With slices and maps, you can solve a whole host of interesting problems, and once you learn how these data

structures work, programming in Go will become fun, fast, and flexible.

4.1 Arrays

An array in Go is a fixed-length collection that contains a contiguous block of elements of any type.

Listing 4.1 An example array

```
var intArray [5]int

intArray[0] = 10
intArray[1] = 20
intArray[2] = 30
intArray[3] = 40
intArray[4] = 50
```

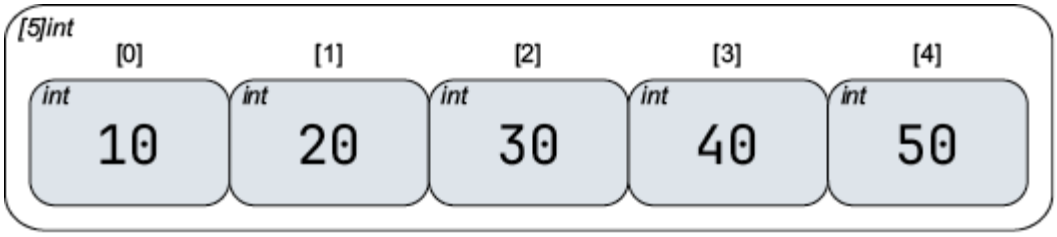


Figure 4.1 Illustration of intArray from listing 4.1 after declaration and assignment

Figure 4.1 illustrates the `intArray` data structure as it exists in memory after its declaration. The elements of the array are marked as gray boxes, and they're connected in series. Each element contains the same type of data and can be accessed through a unique index position.

NOTE Indexes start at `0`.

Arrays are valuable data structures because their memory is allocated sequentially. Keeping memory contiguous can help data stay loaded in CPU caches longer. Using index

arithmetic, you can quickly iterate through all the elements of an array.

> **NOTE** Index arithmetic works in Go, but pointer arithmetic is not allowed, so indices are the only way to access elements in an array or slice.

4.1.1 Declaring arrays

Arrays in Go require two things: a length, which defines how many elements the array holds, and an element type, which defines what kind of value each element stores. As you can see in table 4.1, an array might be declared to hold 5 integers; you specify the count (5) and the type (int), and Go determines the underlying byte size for you.

The length and the element type together form the array's type: two arrays with different lengths, or different element types, are considered different types. Array declaration can take several different forms, depending on the length of the array you want and whether or not you want to give a starting set of values.

Table 4.1 Different types of array declarations

Format	Description	Example	Result
`var A [SIZE]TYPE`	Array variable declaration	`var intArray[5]int`	`[0 0 0 0 0]`
`A := [SIZE]TYPE{VALUES}`	Array literal with optional starting values	`intArray := [5]int{10,20,30}`	`[10 20 30 0 0]`
`A := [SIZE]TYPE{INDEX: VALUE}`	Array literal with sparse values	`intArray := [5]int{1: 20, 3: 40}`	`[0 20 0 40 0]`
`A := […]TYPE{VALUES}`	Array literal with automatic size	`intArray := […]int{10,20,30}`	`[10 20 30]`

Arrays are initialized with zero values when they're created, based on the type of their elements. For example, arrays of numbers start out with `0` for all unspecified values, while arrays of strings have the empty string (`""`), and so on.

Once an array is declared, neither the type of data being stored nor its length can be changed. If you need more elements, you'll need to create a new array with the required length and then copy the values from one array to the other.

Declaring an array with `var` gives you an array of the specified size, with all values preset to the zero value, as you can see in figure 4.2.

Listing 4.2 Declaring an array with var

```
var intArray [5]int
```

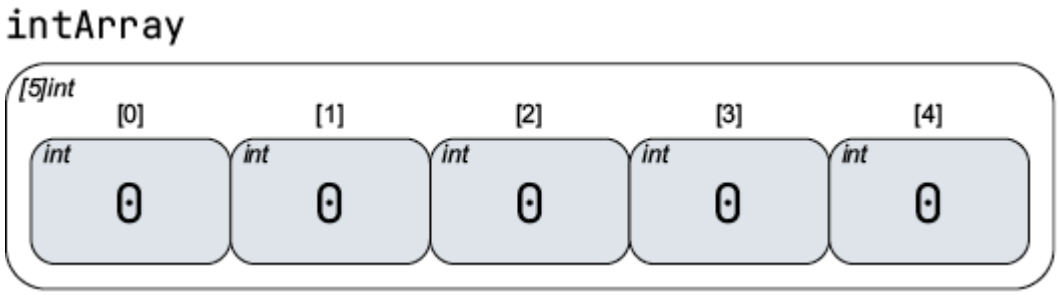


Figure 4.2 The intArray value after declaration

You can also use a short variable declaration with an *array literal* to declare and initialize an array at the same time. Array literals take the form of two curly brackets, `{}`, with a comma-separated list of values inside. You can see what an initalized array with values looks like in figure 4.3.

Listing 4.3 Initializing an array with values using short variable declaration

```
// Declare and initialize an array with values.
intArrayShort := [5]int{1, 2, 3, 4, 5}
```

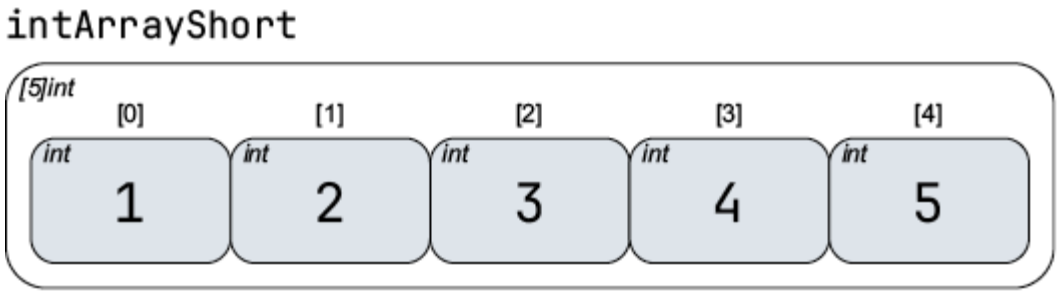


Figure 4.3 The intArrayShort value after declaration

Optionally, you can specify only certain indices, allowing you to set specific values or start from another location, as demonstrated in figure 4.4.

Listing 4.4 Initializing only some elements of an array

```
// Declare and initialize an array with only certain values.
intArraySparse := [5]int{1: 2, 3: 4}
// Declare and initialize an array starting from index 2.
intArrayLast3 := [5]int{2: 3, 4, 5}
```

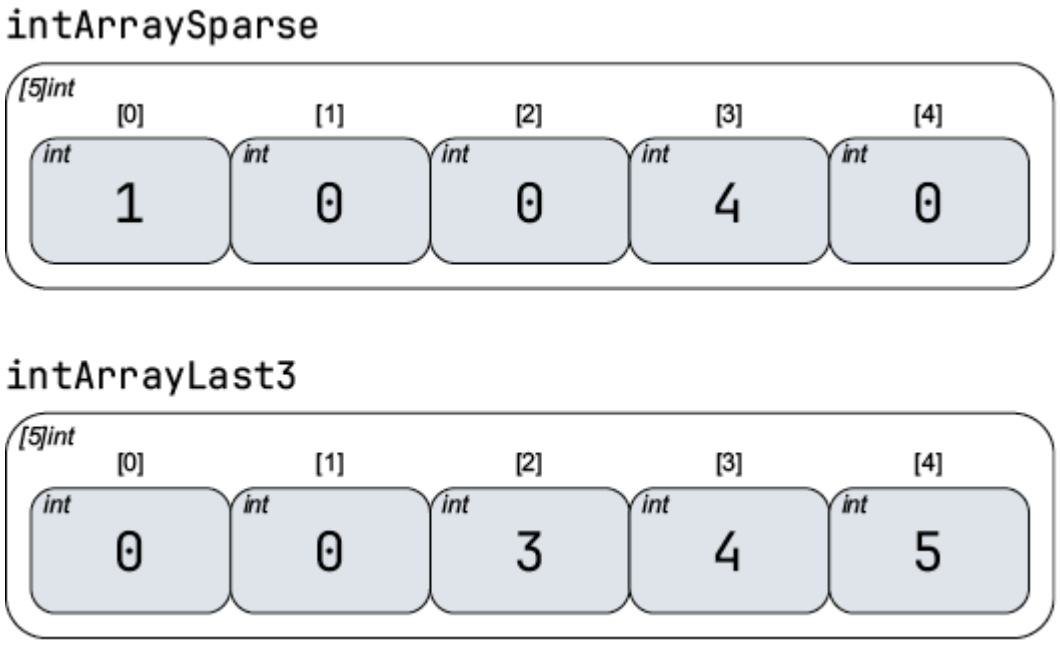


Figure 4.4 Array declared with only certain values

As a further convenience, you can also replace the length with …, and the compiler will automatically create an array type of the appropriate length. You can see this in figure 4.5.

Listing 4.5 Initializing an array with automatic length

```
// Declare and initialize an array with automatic length.
intArrayAuto := [...]int{5, 4, 3, 2, 1}

// If only using certain values, the largest value will be taken as
the
// size.
intArrayAutoSparse := [...]int{0: 5, 3: 2}
```

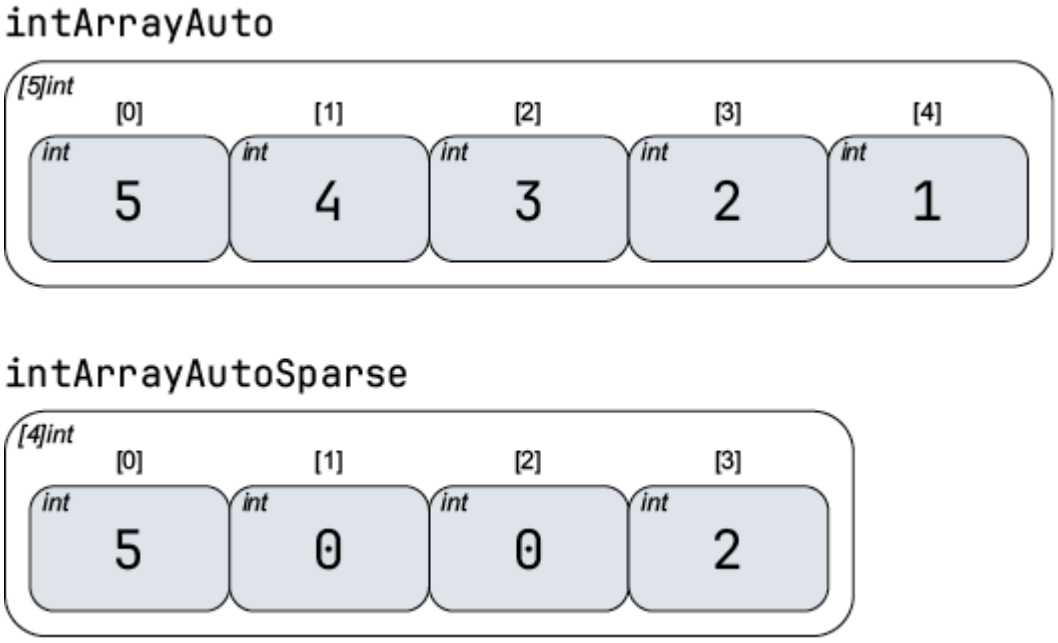


Figure 4.5 Arrays declared with lengths calculated automatically

> **IMPORTANT** Although automatic array length is a useful feature in fixtures such as tests, it should be used with care, because you may end up with lengths you did not intend. It is better to be explicit.

4.1.2 Array type and length

In Go, the length of an array is part of its type, which is why you cannot change an array's size—doing so would make it something else. This also means that the type `[5]int` is completely different from `[4]int`, so they are not interchangeable, even if you attempt to assign a smaller array to a larger one.

Listing 4.6 Attempting to assign array values of different lengths

```
var (
        fourArray [4]int
        fiveArray [5]int
)
fiveArray = fourArray
// error: cannot use fourArray (variable of type [4]int) as type
[5]int
// in assignment
```

To get the size of an array, you can use the built-in `len()` function, which returns an `int` value representing the number of elements the array can hold. This number is always the full size of the array, whether you've chosen to initialize any of its values or not.

Listing 4.7 Getting the size of an array with len()

```
var intArray [3]int
myArray := [3]int{1, 2, 3}

fmt.Println(len(intArray)) //output: 3
fmt.Println(len(myArray))  //output: 3
```

4.1.3 Working with array elements

Array elements can be set and retrieved using the familiar square bracket notation, where the index is an integer literal or a variable of any integer type.

Listing 4.8 Accessing array elements with bracket notation

```
intArray := [5]int{1, 2, 3, 4, 5}

firstValue := intArray[0] // 1

// Access the item using a variable value as the index.
index := 0
integerValue := intArray[index] // 1

// Other integer types work as well.
index32 := int32(1)
index64 := int64(2)
indexUint := uint64(3)

integerValue = intArray[index32] // 2
integerValue = intArray[index64] // 3
integerValue = intArray[indexUint] // 4
```

You can change the value of an array element in the same way, as is illustrated in figure 4.6.

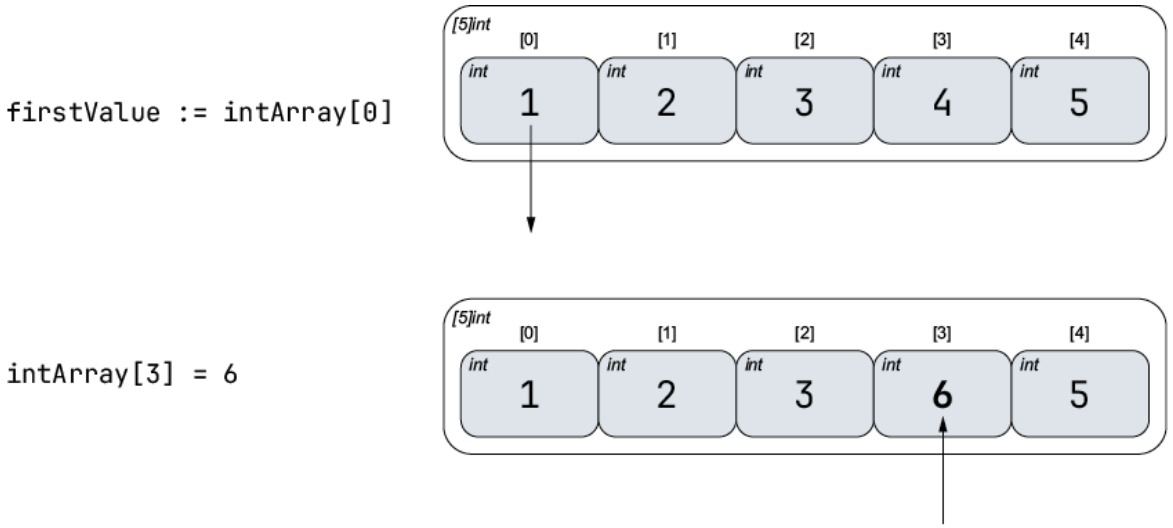


Figure 4.6 Retrieving and setting array values

Listing 4.9 Changing an array element

```
intArray := [5]int{1, 2, 3, 4, 5}

intArray[3] = 6
```

If the array value is a pointer, you can access the value of that pointer using the indirection operator, as you learned in chapter 3.

Listing 4.10 Accessing array pointer elements

```
// Declare an integer pointer array of five elements.
// Initialize index 0 and 1 of the array with integer pointers.
ptrArray := [5]*int{0: new(int), 1: new(int)}
// Assign values to index 0 and 1.
*ptrArray[0] = 10
*ptrArray[1] = 20

valOfPtr := *ptrArray[0] // 10
```

Figure 4.7 shows the result after the preceding array operations are complete.

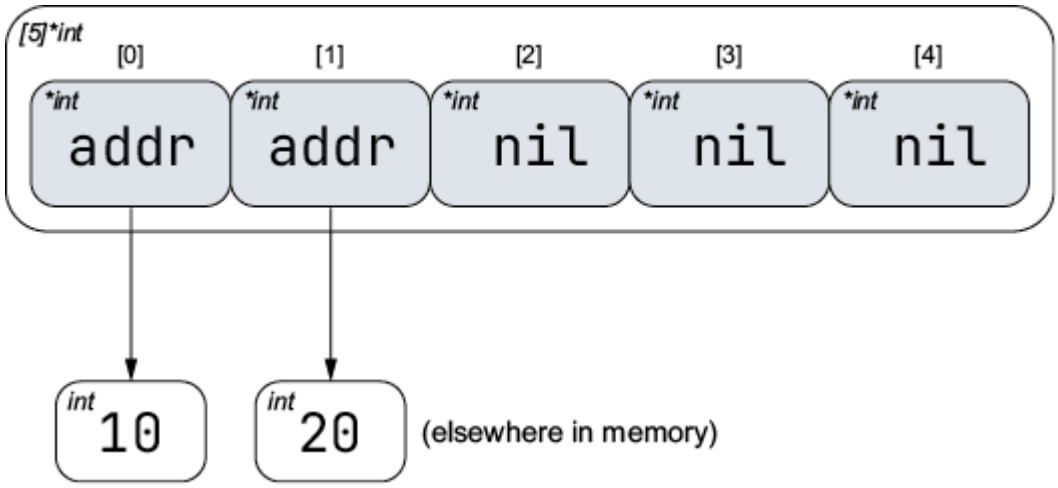


Figure 4.7 An array of pointers that point to integers

4.1.4 Iterating over arrays with for

You can iterating over arrays in Go with a `for` loop, using either a `range` clause or the more traditional approach with `len` and indices.

Listing 4.11 Example loops

```
for i := range items {
        // do something with items[i]
}

for i, item := range items {
        // do something with item or items[i]
}

for i := 0; i < len(items); i++{
        // do something with items[i]
}
```

`range` loops are the easiest and most flexible way to loop through items in Go because they take care of providing sequential indices and items for you, and they stop at the end of the list automatically. In the case of arrays (and slices, which we will explore shortly), `range` loops can be used with either one or two values. When used with a single value, a `range` loop returns an integer value starting at `0`, representing the starting index, and it increases this value by one for each iteration until it has returned the final index.

Listing 4.12 Iterating over an array using range to get indices

```
array := [...]int{1, 2, 3, 4}

// Use a range statement to provide indices in order.
fmt.Println("Starting range loop:")
for i := range array {
        fmt.Printf("index %d is: %d\n", i, array[i])
}
fmt.Println("Range loop complete!")
// output:
// Starting range loop:
// index 0 is: 1
// index 1 is: 2
// index 2 is: 3
// index 3 is: 4
// Range loop complete!
```

If you add a second value to the `range` expression, you will receive both the current index and a copy of the value at that index.

Listing 4.13 Iterating over an array using range to get indices and values

```
array := [...]int{1, 2, 3, 4}

// Use a range statement to provide indices and copies of each valu
e.
for i, item := range array {
        fmt.Printf("index %d is: %d\n", i, item)
}
// output:
// index 0 is: 1
// index 1 is: 2
// index 2 is: 3
// index 3 is: 4
```

You can also ignore the index by assigning it to the blank identifier (_), leaving you with just a copy of each value. This approach can be useful for performing actions on every item in a list without having to handle the index.

Listing 4.14 Iterating over an array using range to get indices and values

```
array := [...]int{1, 2, 3, 4}

// Use a range statement to operate on values only.
for _, item := range array {
        fmt.Println(item)
}

// output:
// 1
// 2
// 3
// 4
```

RANGE LOOPS AND VALUE COPIES

As mentioned earlier, the values you get from a two-value range loop are *copies* of the values in the array, which can result in unexpected behavior if you attempt to modify them.

```
strings := [...]string{"hello", "gophers"}
// Get each element of the array as variable 's'.
for _, s := range strings {
        // Add an exclamation point.
        s = s + "!"
}
fmt.Println(strings)
// output: [hello gophers]
```

This example does not do what you might want, since each value is a copy. To actually modify the elements, you can either make the array a list of pointers or use the index to change them directly.

```
for i := range strings {
        strings[i] = strings[i] + "!"
}
fmt.Println(strings)
// output: [hello! gophers!]
```

Iterating with a `for` loop and indices is fairly straightforward and will be familiar to you if you have used any C-like language before.

Listing 4.15 Iterating over an array using for and indices

```
array := [...]int{1, 2, 3, 4}

for i := 0; i < len(array); i++ {
        fmt.Printf("index %d is: %d\n", i, array[i])
}
```

This style of iteration is useful if you need to go through elements in another order, such as in reverse or by skipping every other element.

Listing 4.16 Iterating over an array using for to traverse the list in a different order

```
array := [...]int{1, 2, 3, 4}

// Iterate the array backwards.
for i := len(array) -1; i >=0 ; i-- {
        fmt.Printf("index %d is: %d\n", i, array[i])
}
// output:
// index 3 is: 4
// index 2 is: 3
// index 1 is: 2
// index 0 is: 1

// Skip every other element.
for i := 0; i<len(array); i+=2 {
        fmt.Printf("index %d is: %d\n", i, array[i])
}
// output:
// index 0 is: 1
// index 2 is: 3
```

4.1.5 Arrays as values

An array is a value in Go, which means you can assign array values of the same type to each other. The variable name denotes the entire array, so an array can be assigned to other arrays of the same type.

Listing 4.17 Assigning one array to another of the same type

```
// Declare a string array of length five.
var colors [5]string

// Initialize another five element array with values.
favoriteColors := [5]string{"Red", "Blue", "Green", "Yellow", "Pin
k"}

// Assignment will copy the contents of the array.
colors = favoriteColors
```

It's important to note that copying an array duplicates all of its contents and places them into the new array. After the `colors = favoriteColors` line, you will have two separate arrays with two sets of identical values, as illustrated in figure 4.8.

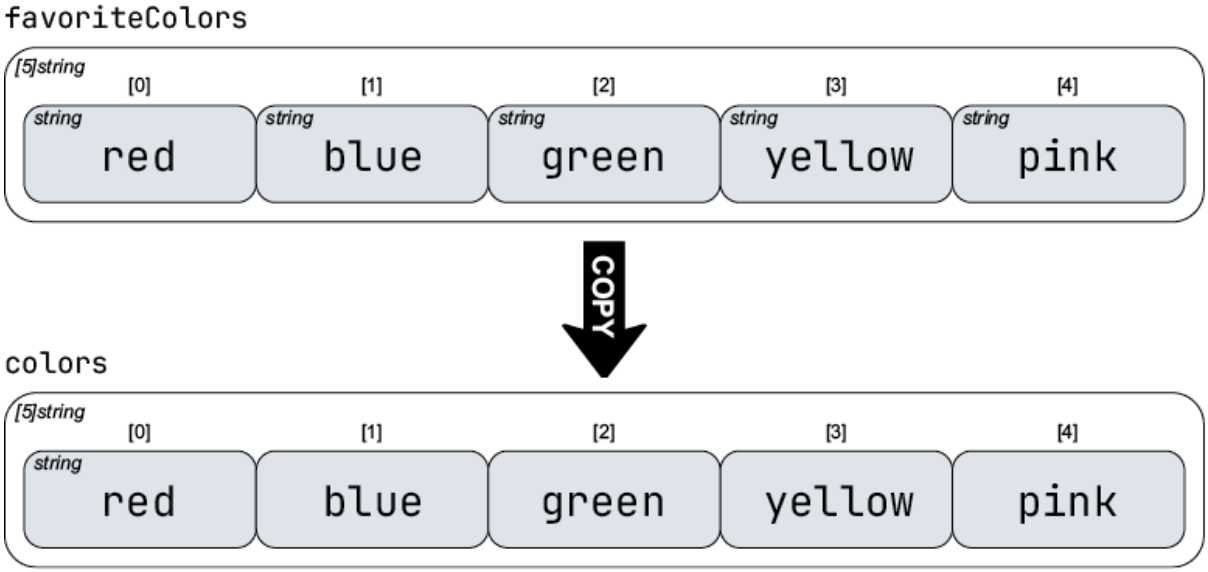


Figure 4.8 Array assignment

This applies to all arrays, regardless of type. Even if the array contains pointer types like the one you saw in listing 4.10, copies of those pointers will be created. In listing 4.18 you will see how two slices can point to the same underlying values. Try adding an additional assignment to that example.

Listing 4.18 Copying arrays with shared values

```
ptrArray := [5]*int{0: new(int), 1: new(int)}
*ptrArray[0] = 10
*ptrArray[1] = 20

ptrArray2 := ptrArray
```

What you'll get in `ptrArray2` is still a separate array with separate pointer values; they just happen to point to the same memory, as outlined in figure 4.9.

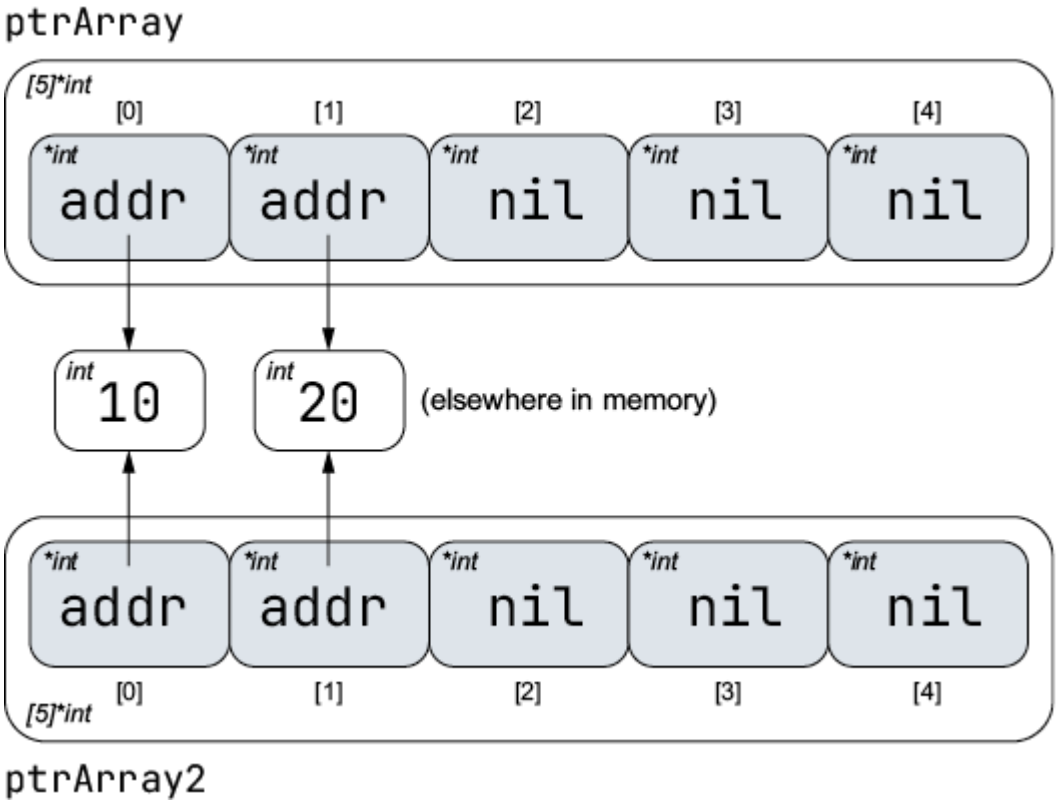


Figure 4.9 Two arrays of pointers referencing the same objects

4.1.6 Multidimensional arrays

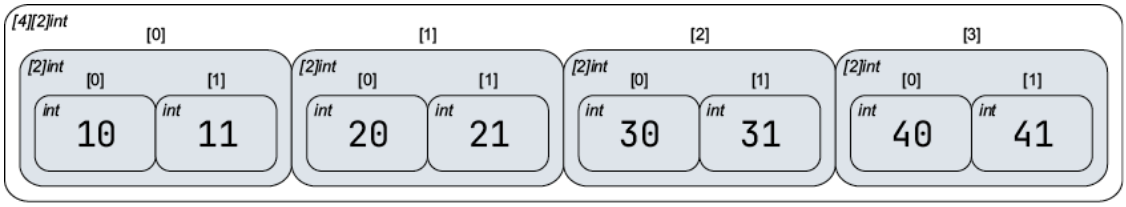
Although Go does not have true multidimensional array support in the form of concise declarations or matrix math, you can simulate the access semantics by composing arrays

together. This can be useful for representing tabular data or things like coordinate systems.

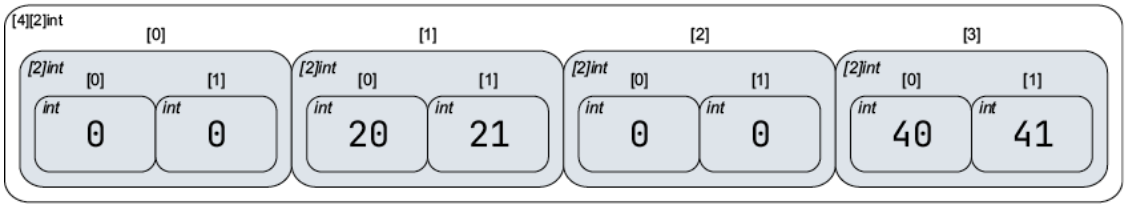
Listing 4.19 Declaring two-dimensional arrays

```
// Declare a two dimensional integer array of four elements by
//two elements.
var array [4][2]int

// Use an array literal to nitialize a two dimensional integer arra
y.
array = [4][2]int{{10, 11}, {20, 21}, {30, 31}, {40, 41}}

// Declare and initialize index 1 and 3 of the outer array.
array = [4][2]int{1: {20, 21}, 3: {40, 41}}

// Declare and initialize individual elements of the outer
// and inner array.
array = [4][2]int{1: {0: 20}, 3: {1: 41}}
```

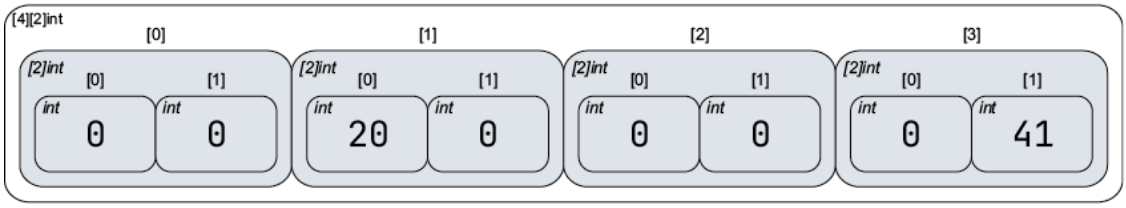
These declarations result in the data structures illustrated in figure 4.10.




```
array = [4][2]int{{10, 11}, {20, 21}, {30, 31}, {40, 41}}
```



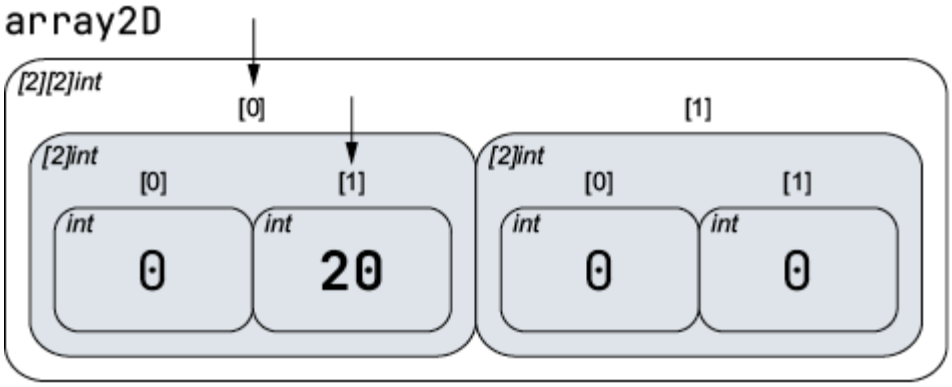

```
array = [4][2]int{1: {20, 21}, 3: {40, 41}}
```




```
array = [4][2]int{1: {0: 20}, 3: {1: 41}}
```

Figure 4.10 Two-dimensional arrays and their outer and inner values

Because these are just arrays of arrays, you can add another set of brackets to access the inner values, as shown in figure 4.11.




```
array2D[0][1] = 20
```

Figure 4.11 Assigning an element of a two-dimensional array

Listing 4.20 Accessing elements of a two-dimensional array

```
// Declare a two dimensional integer array of two elements.
var array2D [2][2]int
// Set integer values to each individual element.
array2D[0][0] = 10
array2D[0][1] = 20
array2D[1][0] = 30
array2D[1][1] = 40
```

As with other arrays, you can copy multidimensional arrays into each other, as long as they have the same element type and the lengths are the same all the way down.

Listing 4.21 Assigning multidimensional arrays of the same type

```
// Declare two different two dimensional integer arrays.
var array1 [2][2]int
var array2 [2][2]int
// Add integer values to each individual element.
array2[0][0] = 10
array2[0][1] = 20
array2[1][0] = 30
array2[1][1] = 40
// Copy the values from array2 into array1.
array1 = array2
```

Because each array is a value, you can extract individual subarrays or values, depending on how many levels you access, as shown in figure 4.12.

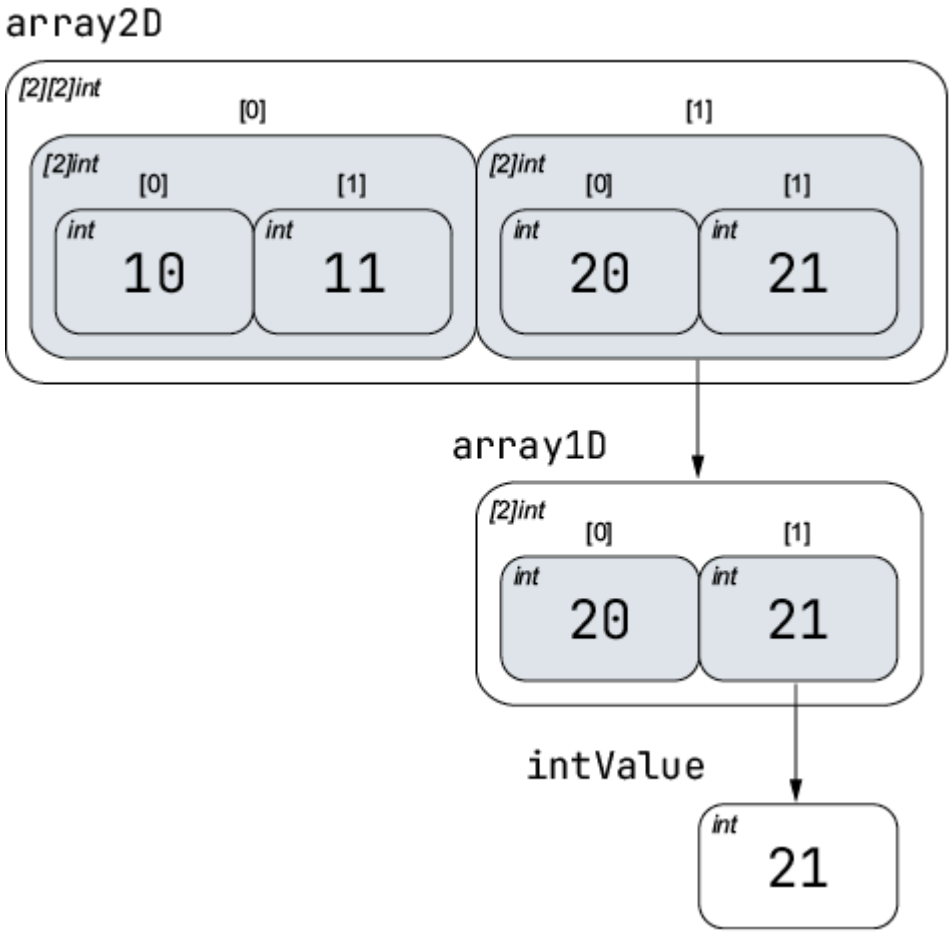


Figure 4.12 Values at different levels

Listing 4.22 Copying a subarray and value

```
array2D := [2][2]int{{10, 11}, {20, 21}}

array1D := array2D[1] // [20 21]

intValue := array1D[0] // 20
```

WHAT IS THE TYPE OF A COLLECTION?

Every variable in Go has one, and only one, type. This is easy to see with simple variables like `var count int`, because it reads naturally as "variable count is an integer," but it can be harder to intuit with collections like `var coordinates [4][2]int`, because it is tempting to think of collections only in terms of the items they contain. A helpful trick is to use the `%T` print verb with `fmt.Printf`. This tells you the Go type of a variable:

```
var array3D [4][3][2]string
fmt.Printf("array3D is a %T\n", array3D)
```

This will output `array3D is a [4][3][2]string`, which you can read as "array3D is a length-four array of length-three arrays of length-two arrays of strings."

4.1.7 Passing arrays to functions

Passing an array value to a function can be an expensive operation in terms of performance and memory usage if you're not careful. This is because all values in Go are passed by value, so every time you pass an array to a function, your program copies the entire array and passes it to the function, regardless of how large it is.

For example, let's say you have a large array representing a 4K image, and you want to pass it to a function for processing. That's 3840 × 2160, or around 8.2 million, pixels.

Listing 4.23 Passing a large array by value to a function

```
// image4K is a large array representing a 4K image #1
type image4K [829400]int // 3840 * 2160

func processPixels(pixels image4K) image4K {
        // Image Processing Code Here.
        return pixels
}

func main() {
        var pixels image4K
        pixels = processPixels(pixels)
}
```

#1 Types will be covered in greater detail in chapter 5.

Every time `processPixels` is called, 8.2 million new integer values will be allocated, and all the values in the `pixels` array will be copied into that allocation, as shown in figure 4.13. On return, the values will be copied again.

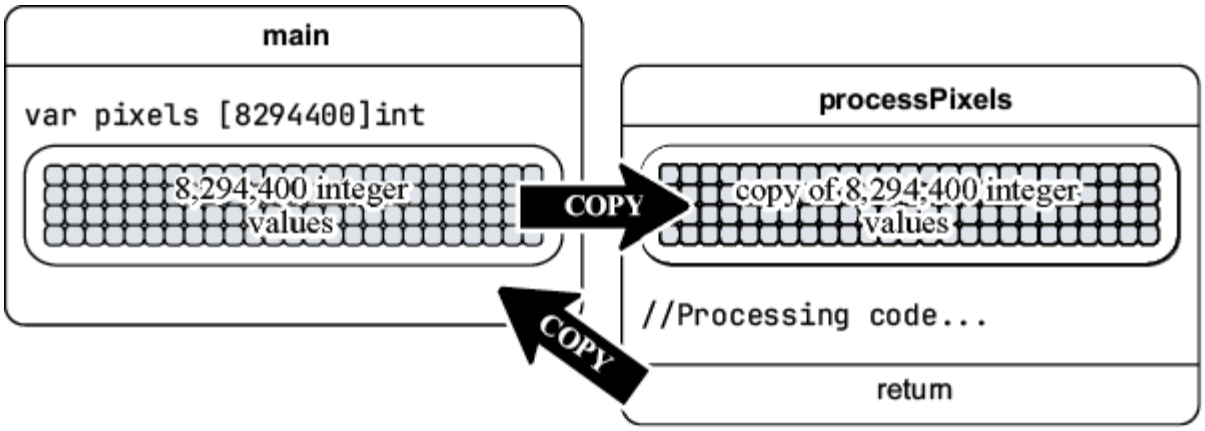


Figure 4.13 Passing a large array by value

Go can handle this copy operation for you, but a better and more efficient way to do this is to pass a pointer.

Listing 4.24 Passing a large array by pointer to a function

```
func processPixels(pixels *image4K) {
        // Processing code...
}

func main() {
        var pixels image4K

        processPixels(&pixels)
}
```

This time, `processPixels` takes a pointer to an array, which uses only about 8 bytes of memory, as opposed to the roughly 66 megabytes it would take to copy 8.2 million integers.

This operation is much more memory efficient and will yield better performance, though you need to keep in mind that any changes to the array will be reflected in the original value.

4.1.8 The problem with arrays

Although they are useful for certain applications that require a fixed number of items, arrays are generally a poor collection type.

Because their size is fixed, arrays cannot grow or shrink in response to different circumstances, which makes them difficult to use in real-world situations where the number of items in a list is not always known ahead of time. Arrays always take up the same amount of memory, which is determined when the program is compiled. This means that if you use less than the full amount, you will waste memory, and if you need more space, you will need to rebuild your program with a larger array size.

Arrays also require extra care when you share their contents because, as you've just learned, they are copied every time they pass from function to function.

Slices address both of these problems in an elegant way and allow you to work with lists much more effectively.

4.2 Slices

Slices are Go's primary ordered data type and can be used in all the same ways you would use an array, with additional dynamic features that make them much more flexible and memory efficient. At first glance, slices look a lot like arrays, except that they do not specify a size.

Listing 4.25 An example slice

```
slice := []int{1, 2, 3, 4}

value := slice[0] // 1

for i := range slice {
        fmt.Println(slice[i])
}
// output:
// 1
// 2
// 3
// 4
```

If you pull back the curtain and look at their representation in memory, illustrated in figure 4.14, slices start to get a bit more interesting. As you can see, slices comprise two parts: an array used for storage (called the *backing array*) and the slice data structure itself, also called the *slice header*. The slice header is the data that is actually stored in the slice variable, and it represents a dynamic view, or "slice," of an array, which is where the data structure gets its name.

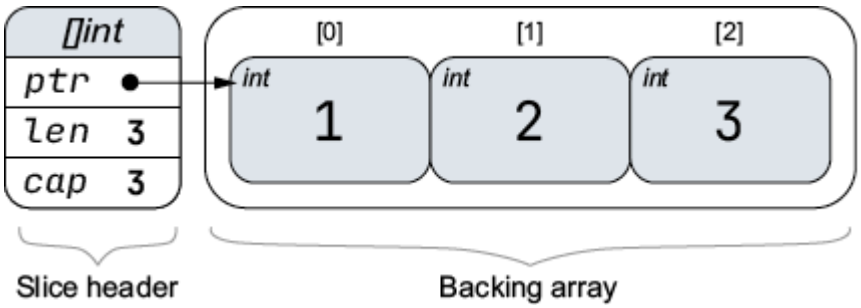


Figure 4.14 Slice internals

The slice header contains just three values: a pointer to the memory location where the slice begins, an `int` representing the number of elements in the slice (the length), and an `int` that specifies the total number of items available for storage or expansion (the capacity). This simple structure is what gives slices their flexibility and makes them so powerful. By separating an array's storage from its representation, slices provide a collection type that works like an array but can grow or shrink as needed and be copied efficiently without needing to copy all the values in the array. It's a really neat trick!

4.2.1 Declaring slices with slice literals

Using slice literals is the most straightforward way to create a new slice, and it looks exactly like using an array literal, as you can see in figure 4.15, without a size.

Listing 4.26 Declaring and initializing a slice with a literal

```
// Declare and initialize an integer slice with a literal.
sliceLit := []int{10, 20, 30}
```

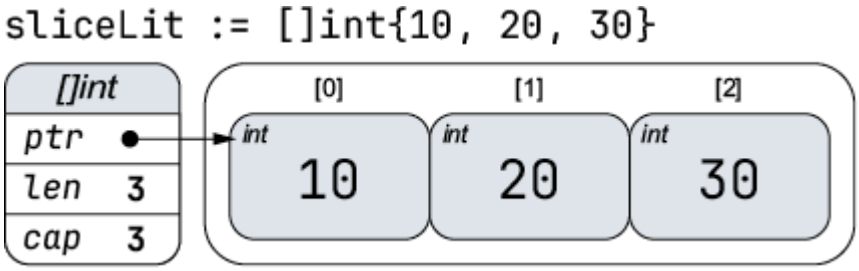


Figure 4.15 Slice variable declared and initialized with a literal

Slices declared in this way are allocated with a new backing array containing the provided items, and they point to the first element of that array. The length and capacity are also set to the number of values provided, which you can inspect using the built-in `len` and `cap` functions.

Listing 4.27 Getting the length and capacity of a slice

```
// Get the length and capacity of newly-created slice.
sliceLitLen := len(sliceLit) // int(3)
sliceLitCap := cap(sliceLit) // int(3)
```

4.2.2 Declaring slices with make

You can also create new slices using the built-in `make` function, which returns a new slice with all of the values set to the zero value for its element type. Calls to `make` take the form `make([]T, length)` or `make([]T, length, capacity)`, where `T` is the element type of the slice, and `length` and `capacity` are integers defining the number of values in your slice and the extra capacity for expansion. Specifying a capacity allows you to preallocate storage for a slice ahead of time, which allows it to grow without needing to create a new backing array. This is helpful in cases where you will be expanding your slice with the `append` call, which is how slices grow in Go.

If `make` is called with only a length, it returns a slice pointing to a backing array of exactly that size, as visualized in

figure 4.16, with all values zeroed out.

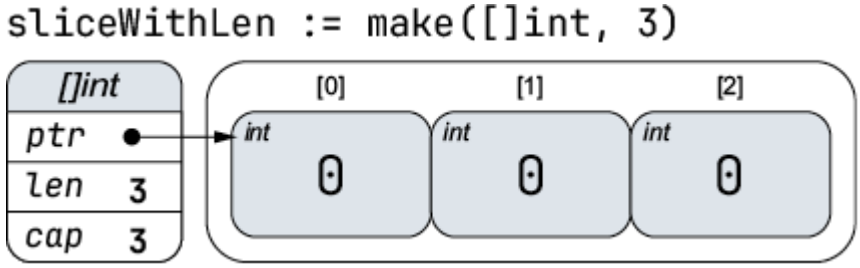


Figure 4.16 Slice Initialized with make and a length

Listing 4.28 Declaring a slice with make and a length

```
// Declare and initialize an integer slice with length 3
sliceWithLen := make([]int, 3) // identical to []int{0, 0, 0}
fmt.Println(len(sliceWithLen)) // 3
```

If `make` is given a capacity, it allocates extra storage beyond the length value for use in later expansion, as seen in figure 4.17.

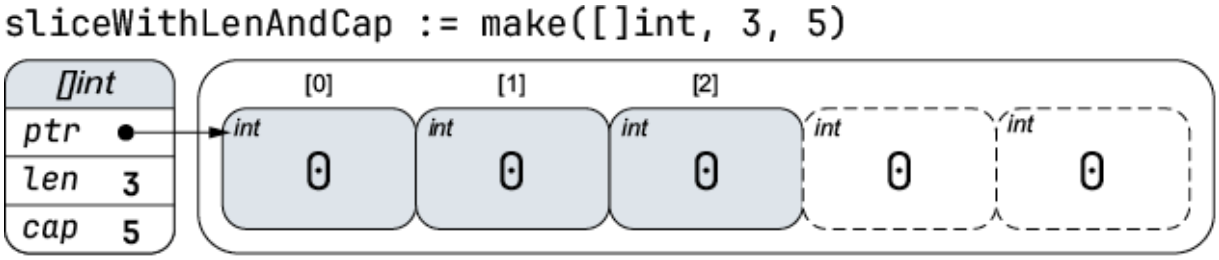


Figure 4.17 Slice initialized with a length and a larger capacity

Listing 4.29 Declaring a slice with make and both length and capacity

```
// Declare and initialize an integer slice with length 3, and capac
ity 5
sliceWithLenAndCap := make([]int, 3, 5)
fmt.Println(len(sliceWithLenAndCap)) // 3
```

Here, the dashed values in white represent items in the backing array that have been allocated but are currently inaccessible from within the slice because they are reserved for expansion.

IMPORTANT The capacity must be greater than or equal to the length. If the compiler can catch this for you, it will throw an error; otherwise, you will get a runtime panic.

If the slice is declared with a capacity greater than zero but a length of 0, the slice will appear to be empty, as shown in figure 4.18, but it will be backed by an array of the specified capacity.

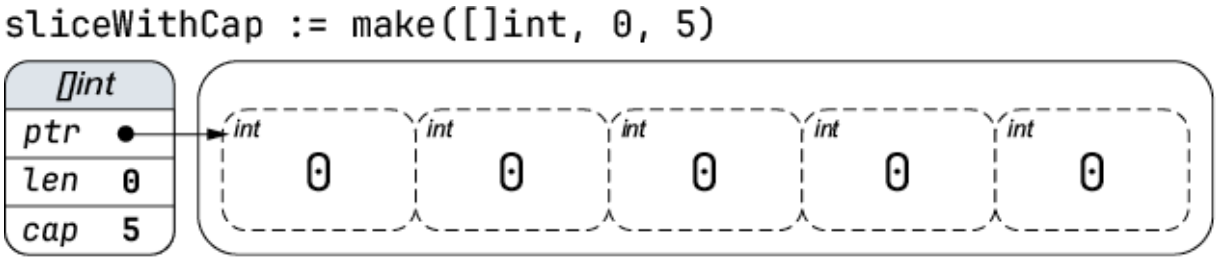


Figure 4.18 Slice Initialized with length 0 and a larger capacity

Listing 4.30 Declaring a slice with make and a capacity, but zero length

```
// Declare and initialize an integer slice with length 0, and capac
ity 5
sliceWithCap := make([]int, 0, 5)
fmt.Println(len(sliceWithCap)) // 0
```

4.2.3 Nil slices

Slices declared with `var` are uninitialized and are known as *nil slices* because they point to `nil` instead of to a backing array. They still return values for `len` and `cap` (both `0`), but they are considered equal to `nil`, and attempts to access their values will cause a runtime panic because they don't point to anything yet, as shown in figure 4.19.

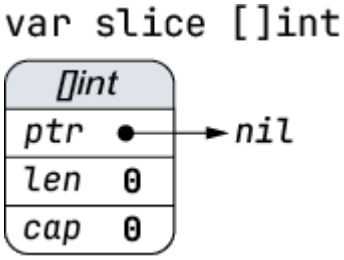


Figure 4.19 A nil slice

Listing 4.31 Slice variable declaration with var

```
// Declare an integer slice variable.
var sliceVar []int

fmt.Println(len(sliceVar), cap(sliceVar)) // 0 0
fmt.Println(sliceVar == nil) // output: true

value := sliceVar[0] // panic: index out of range [0] with length 0
```

Nil slices can be useful as placeholders if the slice is optional (for example, as part of a struct) or if you plan to assign values later through other means, but it's important to ensure that they are allocated before you attempt to access values from them. You can do this with literals or `make`, or with an assignment using `append` or slice expressions.

> **IMPORTANT** Slices declared with `var` are not usable until they have a value assigned.

4.2.4 Growing slices with append

One benefit of the way slices are implemented is that they can grow by increasing their length if capacity is available or by allocating new storage if it is not. This growth is handled by the built-in `append` function.

Listing 4.32 The append function

```
func append(slice []Type, elems ...Type) []Type
```

The `append` function works by taking an existing slice and one or more new values as arguments and returning a modified slice with the new values added. It examines the input slice and allocates storage as necessary to grow the slice. This makes it safe to call on any slice, even one that has not been initialized yet.

Listing 4.33 Using append to add an element to a slice

```
// Initialize a slice of integers with three values.
intSlice := []int{10, 20, 30}
// Assign the slice to the appended version of itself.
intSlice = append(intSlice, 40)

fmt.Println(intSlice)
// output: [10 20 30 40]

var sliceVar []string // nil slice
sliceVar := append(sliceVar, "This works!")
fmt.Println(sliceVar)
// output: ["This works!"]
```

Using `append` is easy and fairly straightforward, but it helps to know a bit more about how it works so you can understand how it can change the underlying array. Let's take a look at the `intSlice` example again, this time with some extra print statements to see what happens to the length and capacity.

Listing 4.34 The effect of append on length and capacity

```
intSlice := []int{10, 20, 30}
fmt.Printf("len: %d, cap: %d\n", len(intSlice), cap(intSlice))
// output: len 3, cap: 3

intSlice = append(intSlice, 40)
fmt.Printf("len: %d, cap: %d\n", len(intSlice), cap(intSlice))
// output: len: 4, cap: 6
```

From the output, you can see that the capacity of the slice starts out at `3`, which means that the underlying array has space to represent only three values and will need to grow to accommodate more. After the `append`, the length of the slice is `4`, which makes sense because one element has been added, but the capacity is now `6`, which is twice that of the original slice. This suggests that the underlying array has increased in size. To see how this works, let's take a look at what happens during the call to `append`.

First, a new, larger backing array is allocated to hold the new slice values (figure 4.20).

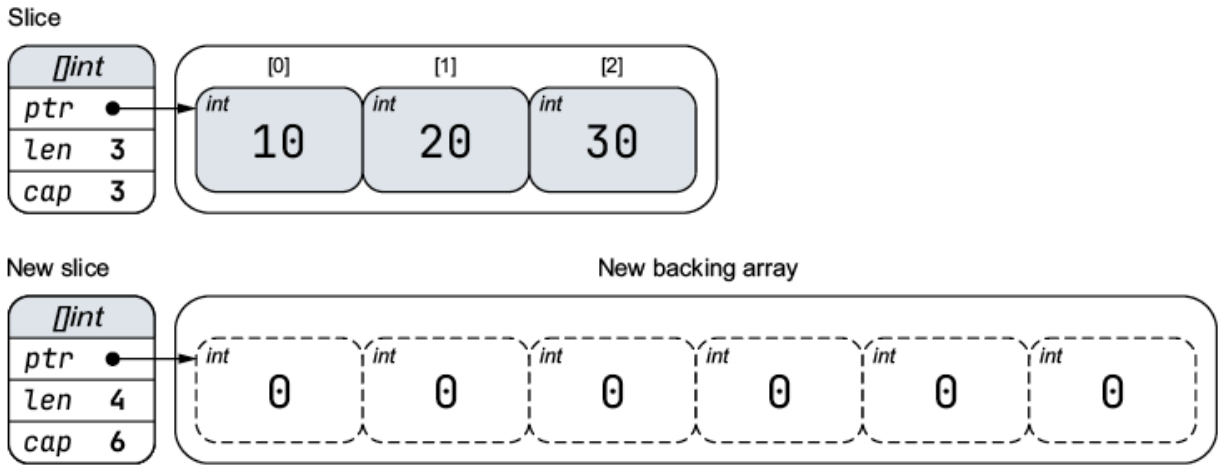


Figure 4.20 Append with allocation, step 1: allocate a new slice with storage

Next, the old values are copied into the new array, along with any new values to append (figure 4.21).

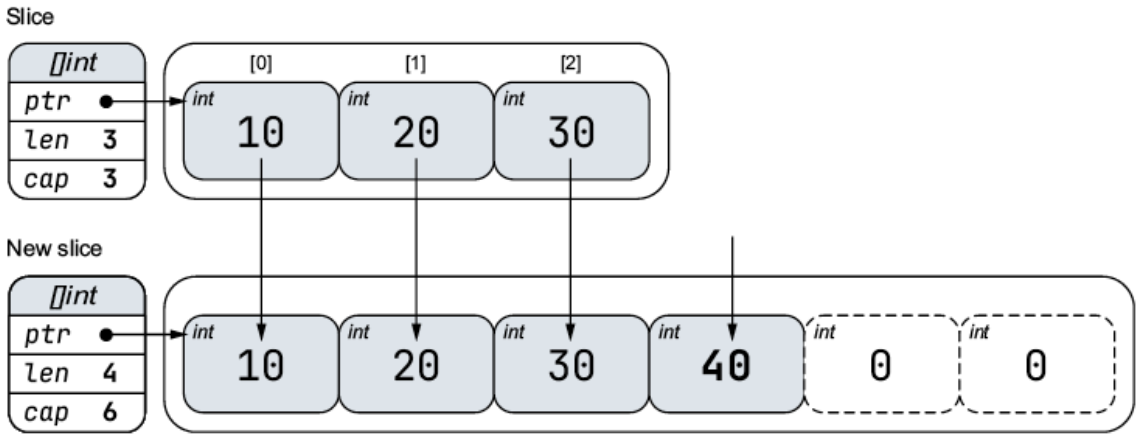


Figure 4.21 Append with allocation, step 2: copy the old values and assign new ones

Finally, the new slice header is returned, and the assignment overwrites the old one, releasing the memory held by the old array (figure 4.22).

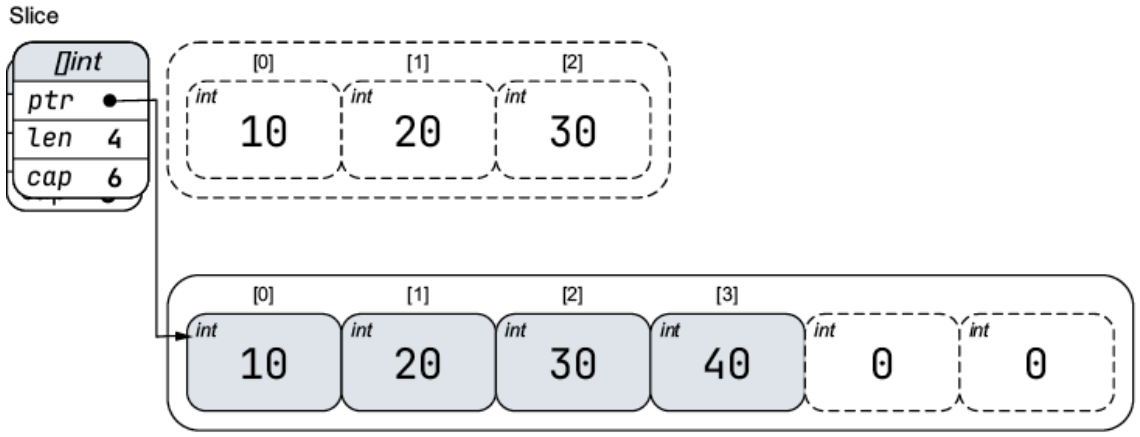


Figure 4.22 Append with allocation, step 3: return the new slice to overwrite the old one

The new slice has a larger capacity than it needs because copying values is an expensive operation, and doing it for every append would be wasteful. Go preallocates extra memory every time a slice needs to allocate new storage, reducing the cost of copying over time. This is sometimes called array doubling or amortized doubling.

Keeping this behavior in mind, what do you think will happen to the capacity if you append to this same slice again?

Listing 4.35 Appending to a slice that has capacity

```
slice = append(slice, 50)
fmt.Println(slice)
// output: [10 20 30 40 50]
fmt.Printf("len: %d, cap: %d\n", len(slice), cap(slice))
// output: len: 5, cap: 6
```

The length has now increased, but the capacity has stayed the same because the previous allocation left enough extra capacity to hold the new value without a reallocation, as seen in figure 4.23.

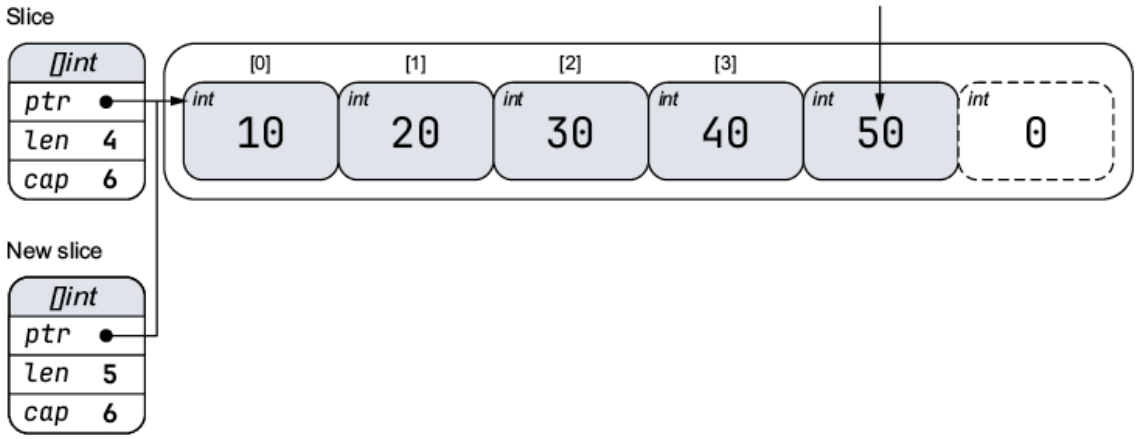


Figure 4.23 Append without allocation

This approach makes appending much less expensive over time because the extra capacity means that the slice won't need to be copied again until the new storage is used up.

HOW MUCH WILL APPEND GROW THE CAPACITY?

The `append` operation is clever when growing the capacity of the backing array. As of this printing, the algorithm will always double the existing capacity if the total number of elements is less than 1,024. Beyond this number, the capacity is grown by a factor of .25, or 25%, though this could change in future versions of the language as more optimizations are added.

Multiple values can be appended by simply adding more arguments. Or, if you are appending one slice to another, you can unpack the second slice with the special … operator, which transforms a slice into a list of arguments.

Listing 4.36 Using append to add multiple values to a slice

```
slice := []int{10, 20, 30}
slice := append(slice, 40, 50)
fmt.Println(slice)
// output: [10 20 30 40 50]

slice2 := []int{60, 70}
slice := append(slice, slice2...) // The same as append(slice, 60,
70)
fmt.Println(slice)
// output: [10 20 30 40 50 60 70]
```

WHY DOES APPEND RETURN A NEW SLICE?

It might look a little strange to append to slices with an assignment instead of a built-in method, but this approach is actually more flexible because it lets you choose whether to modify a slice by inserting new values or simply create a new slice with extra elements.

4.2.5 Avoiding append surprises

Because `append` sometimes allocates new storage, it can be a source of confusion for new Go programmers who expect to see changes to their slice after every append. Things can get a little tricky when slices cross function boundaries.

Normally, values in a slice can be modified by functions, as in the following example, which adds 1 to every slice element.

Listing 4.37 Modifying a slice with a function

```
// addOneToAll adds 1 to every slice value.
func addOneToAll(s []int) {
        for i := range s{
                s[i]+=1
        }
}

func main(){
        slice := []int{1,2,3}
        addOneToAll(slice)
        fmt.Println(slice)
}
```

Listing 4.38 Output after addOneToAll

```
[2 3 4]
```

This function works as intended, but adding an append changes the results.

Listing 4.39 Adding an append to addOneToAll

```
// addOneToAll grows the slice and adds 1 to every slice
// value.
func addOneToAll(s []int) {
        s = append(s, 0)
        for i := range s {
                s[i] += 1
        }
}

func main(){
        slice := []int{1,2,3}
        addOneToAll(slice)
        fmt.Println(slice)
}
```

Listing 4.40 Output after addOneToAll with an append

```
[1 2 3]
```

What happened to the changes? In fact, they were made, but to a completely different slice! In the original example, both slices pointed to the same memory, so after the function completed, the slice in `main` pointed to the new values (see figure 4.24).

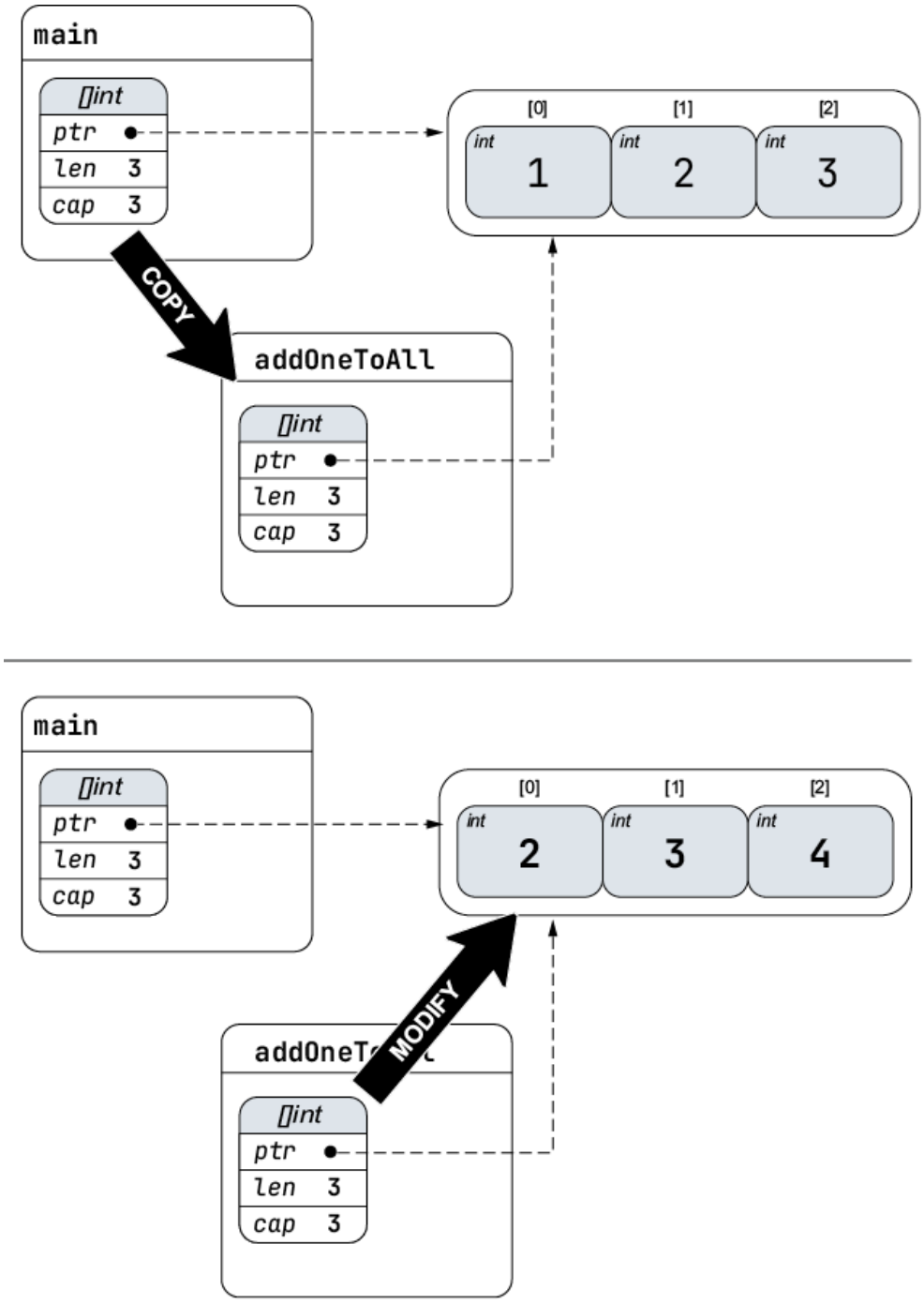


Figure 4.24 addOneToAll modifying the same values in memory

But once an `append` is added, a new backing array is allocated, and the two slice variables no longer point to the same memory. `addOneToAll` makes its changes to a slice that exists only within that function (see figure 4.25).

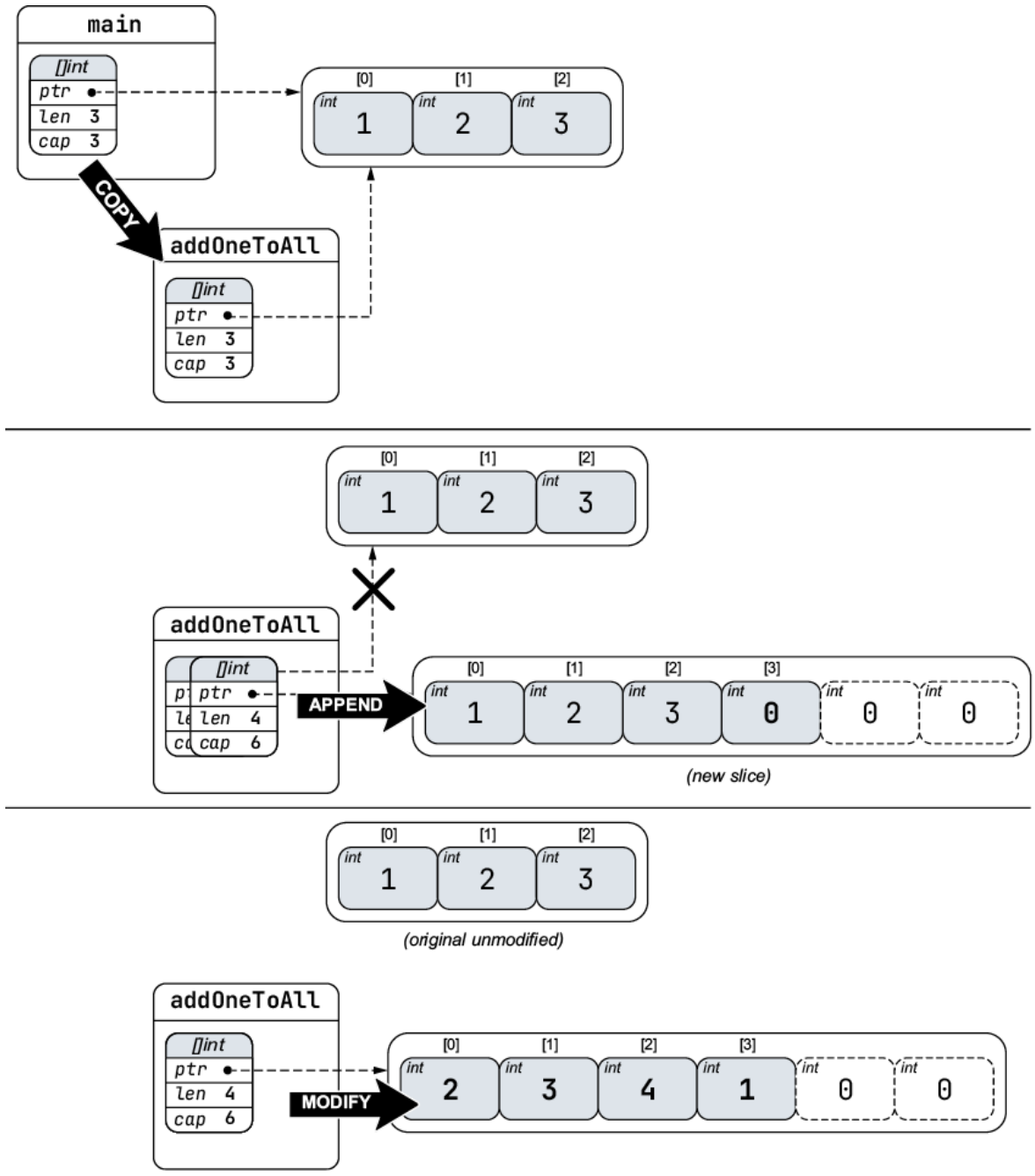


Figure 4.25 addOneToAll modifying a new slice after append

You can try to get around this by adding extra capacity in `main` before calling the function, but this doesn't quite work either.

Listing 4.41 Attempting to avoid the append allocation with extra capacity

```
func main(){
        // Create A Slice With Extra Capacity
        slice := make([]int, 0, 4)
        slice[0] = 1
        slice[1] = 2
        slice[2] = 3

        addOneToAll(slice)
        fmt.Println(slice)
}
```

Listing 4.42 Output after addOneToAll with extra capacity

```
[2 3 4]
```

The values were incremented, but the final value is missing. This is because the length of the original slice in `main` remains the same, so it cannot "see" the new value (see figure 4.26).

The right thing to do in this case is either to append to slices before passing them on or to return the appended slice from the function, similar to how `append` works.

Listing 4.43 Returning an appended slice from addOneToAll

```
func addOneToAll(s []int) []int {
        s = append(s, 0)
        for i := range s {
                s[i] += 1
        }
        return s
}

func main() {
        slice := []int{1, 2, 3}
        slice = addOneToAll(slice)
        fmt.Println(slice)
        // output: [2 3 4 1]
}
```

Listing 4.44 Output after addOneToAll with the returned slice

```
[2 3 4 1]
```

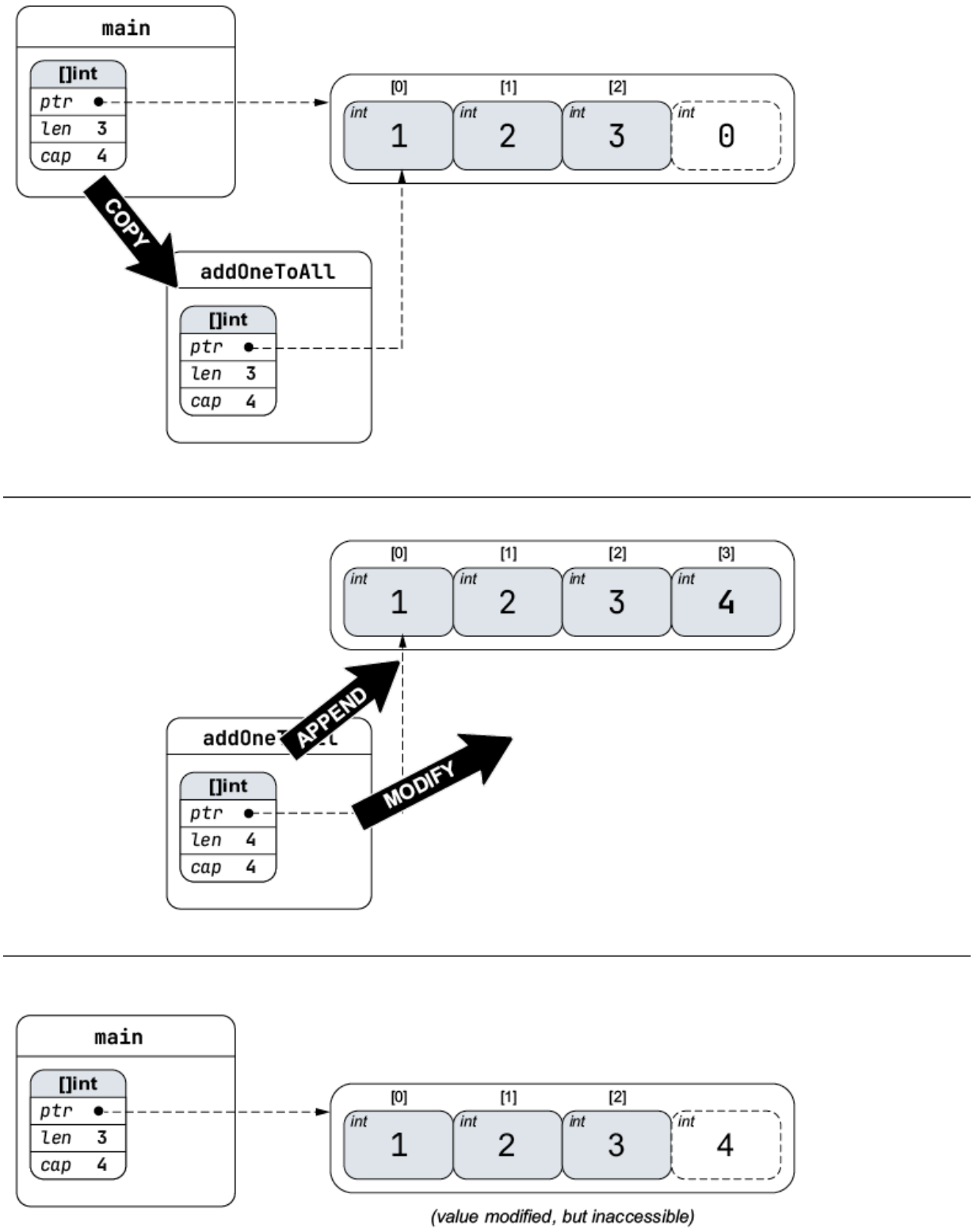


Figure 4.26 addOneToAll appending and modifying, but the new value is inaccessible

4.2.6 Slice expressions

One of the things that makes slices so flexible is that they can be created from sections of other slices or arrays while still sharing the same storage. This is done with *slice expressions*, which allow you to specify a range of values from a slice, creating a window into the existing slice.

Slice expressions take the form of `a[low : high]`, where `a` is an existing slice or array, and `low` and `high` are indices representing the range of values you want in the new slice. You can leave out one or both of these values; they default to `0` and `len(a)`, respectively, making it easy to specify ranges "starting from" or "up to" a particular index, as seen in figure 4.27.

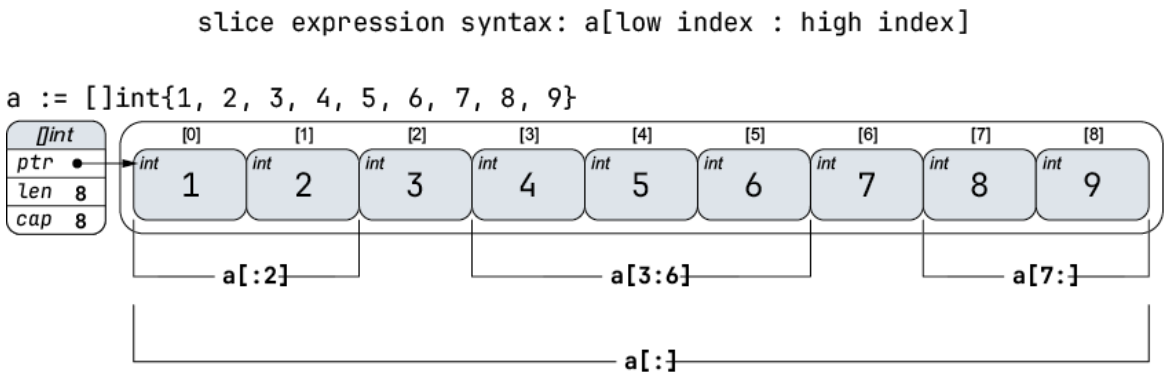


Figure 4.27 Slice expression syntax

When used with a slice, as in figure 4.28, a slice expression creates a new slice that points to the `low` index in the backing array, with its length being set based on the `high` value.

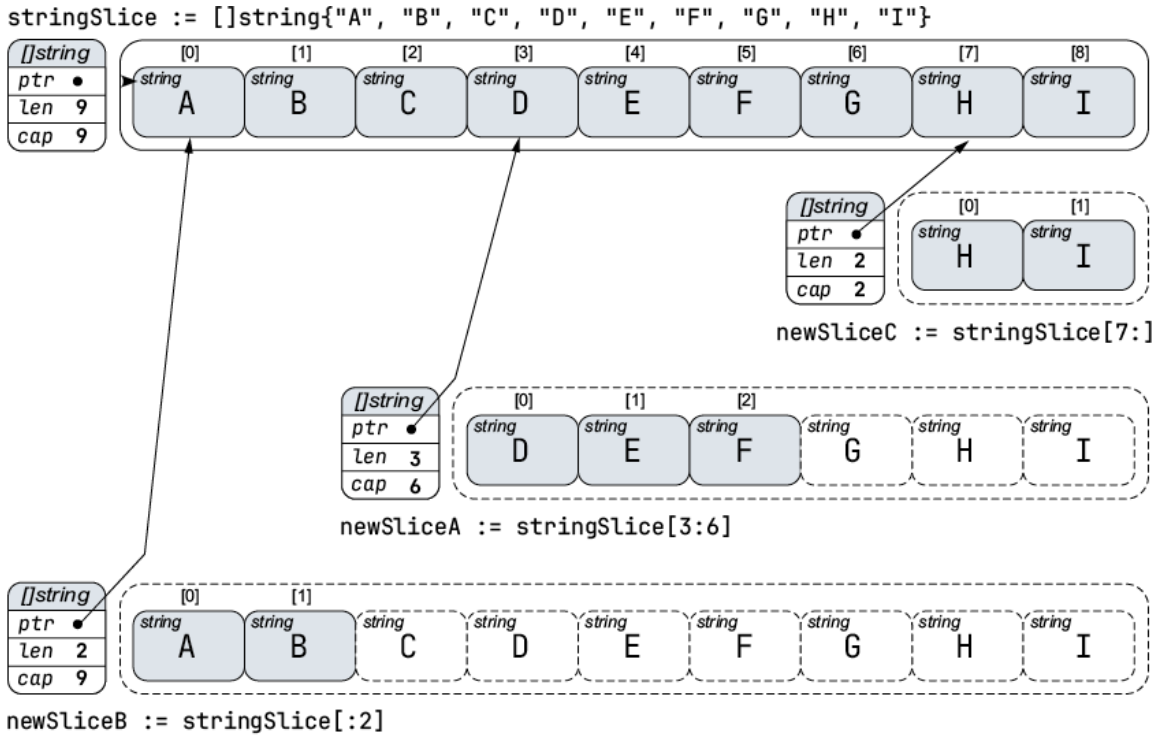


Figure 4.28 New slices created with slice expressions

Listing 4.45 Creating slices from an existing slice

```
stringSlice := []string{"A", "B", "C", "D", "E", "F", "G", "H",
"I"}

newSlice1 := stringSlice[3:6] // [D E F]
newSlice2 := stringSlice[:2]  // [A B] - "up to" index 2
newSlice3 := stringSlice[7:]  // [H I] - "starting from" index 7
```

When used with an array, a slice expression creates a new slice that uses the array for its storage, as shown in figure 4.29.

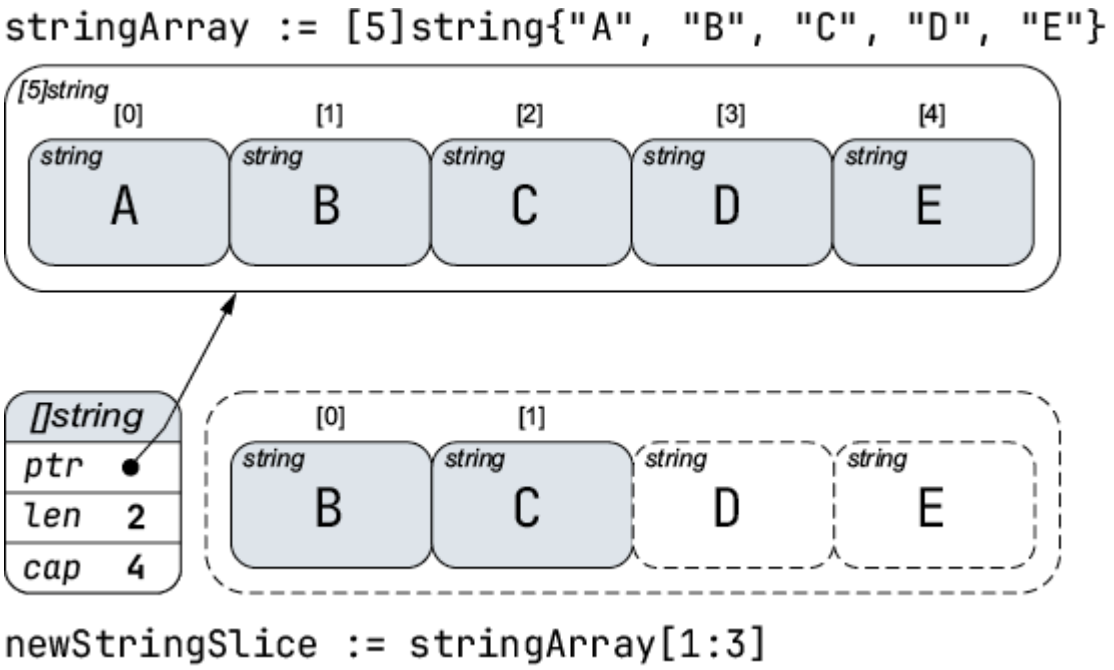


Figure 4.29 New slice created from an array

Listing 4.46 Creating a slice from an array

```
stringArray := [5]string{"A", "B", "C", "D", "E"}
newStringSlice := stringArray[1:3] // [B C]
```

SLICE EXPRESSIONS WITH CAPACITY

By default, slices created with slice expressions inherit the remaining portion of the slice or array they came from as capacity. Normally this is not a problem, but under certain circumstances, you might want to restrict the capacity of the new slice so that it cannot be appended past a certain point.

To do this, you can use a slice expression with three indices, provided in the form `a[low : high : max]`. In this form, the capacity of the new slice is cut off at the `max` index, preventing any modification of the original slice beyond this point, as seen in figure 4.30.

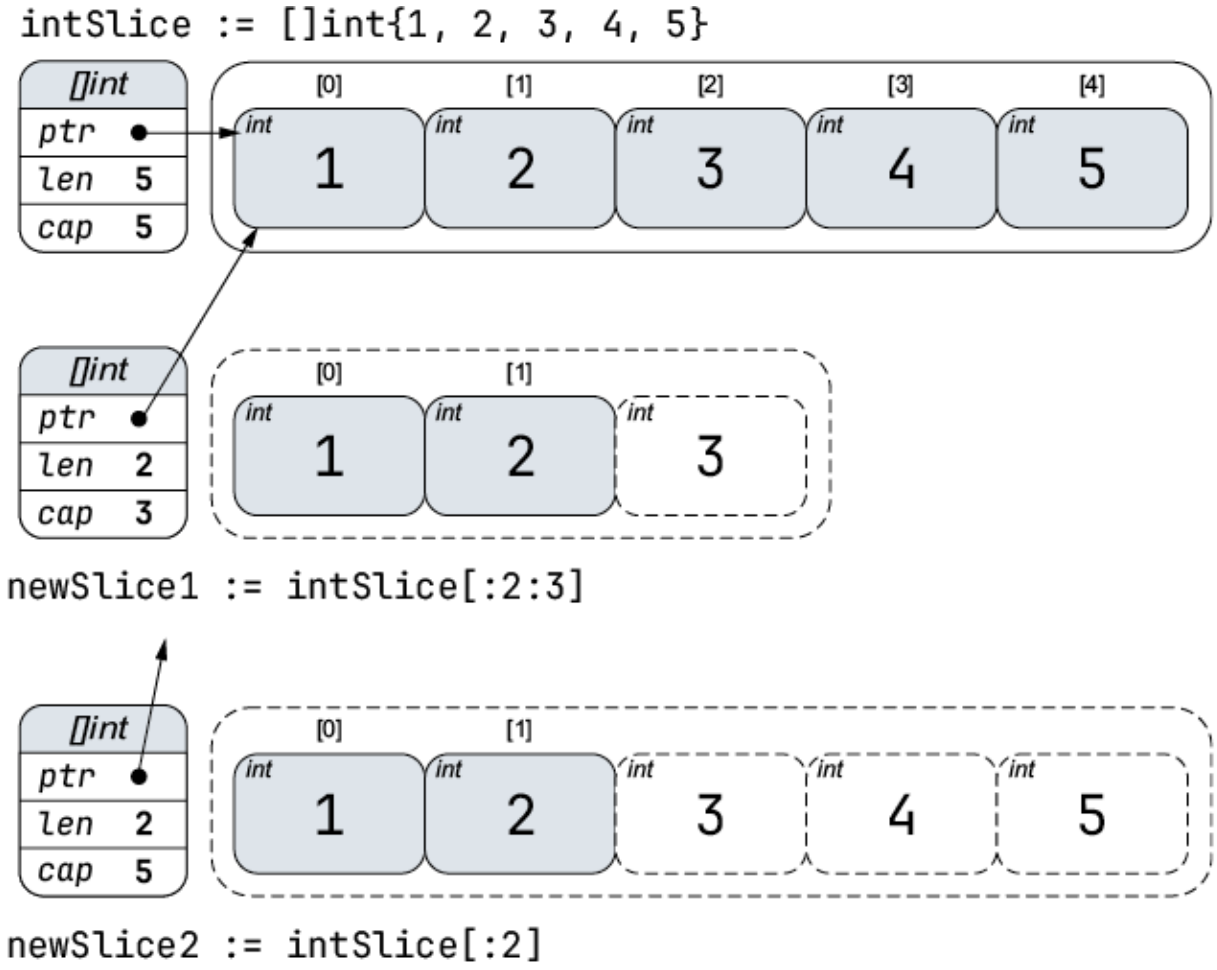


Figure 4.30 New slices created with and without a max value

Listing 4.47 Slice expressions with and without a max value

```
intSlice := []int{1, 2, 3, 4, 5}
newSlice1 := intSlice[:2]   // [1 2]
newSlice2 := intSlice[:2:3] // [1 2] (same values, less capacity)
```

SLICE EXPRESSIONS IN ACTION

Slice expressions can be useful for passing only a portion of a slice to a function that operates on slices. For example, suppose you want to write a function to average the values in a slice of `float64` values. If you write it using `range` and `len`, you can get meaningful values from slices of any size.

Listing 4.48 Function to return the average value of a slice of float64

```
// avg returns the average of a []float64.
func avg(fs []float64) float64 {
        var sum float64
        // Add each value to the sum.
        for i := range fs {
                sum += fs[i]
        }
        // Divide the sum by the length of the slice.
        return sum /float64(len(fs))
}
```

If you need to create an average from a large slice of values, such as metrics or sensor readings, you can choose to extract only a subset of values to pass to the function.

Listing 4.49 Averaging only the first 10 entries in a slice using a slice expression

```
func getMeasurements() []float64 {
        // Simulate a large slice with 10 values and a bunch of zer
os.
        return []float64{3.3, 1.5, 3.5, 2.9, 2.7, 3.0, 3.1, 4.5, 2.
9, 3.5, 99: 0}
}

func main(){
        measurements := getMeasurements()
        fmt.Printf("averaging 10 out of %d\n measurements", len(mea
surements))
        averageMeasurement := avg(measurements[:10])
        fmt.Printf("average measurement is: %f\n", averageMeasureme
nt)
}
```

Listing 4.50 Output from averaging a subset of a slice

```
averaging 10 out of 100 measurements
average measurement is: 3.090000
```

4.2.7 Copying slices with copy

Although `append` and slice expressions provide a fast, flexible way to create new slices and save on allocations, sometimes you will want to duplicate the contents of a slice into a new slice, and this is what the built-in `copy` function is for.

Listing 4.51 The copy function

```
func copy(dst, src []Type) int
```

The `copy` function takes two slices of the same type and copies the elements of `src` into `dst`, returning the number of elements copied. The one thing to keep in mind when using `copy` is that, when the lengths of the slices differ, it will use only the shorter of the two. If the source slice is shorter, `copy` will copy all the elements from the source slice, leaving the rest of the elements in the destination slice untouched. If the destination slice is shorter, `copy` will copy only enough values to fill the slice. This often trips people up because, unlike `append`, `copy` will not do any allocations for you; if either slice has a length of `0`, nothing will be copied, as shown in figure 4.31.

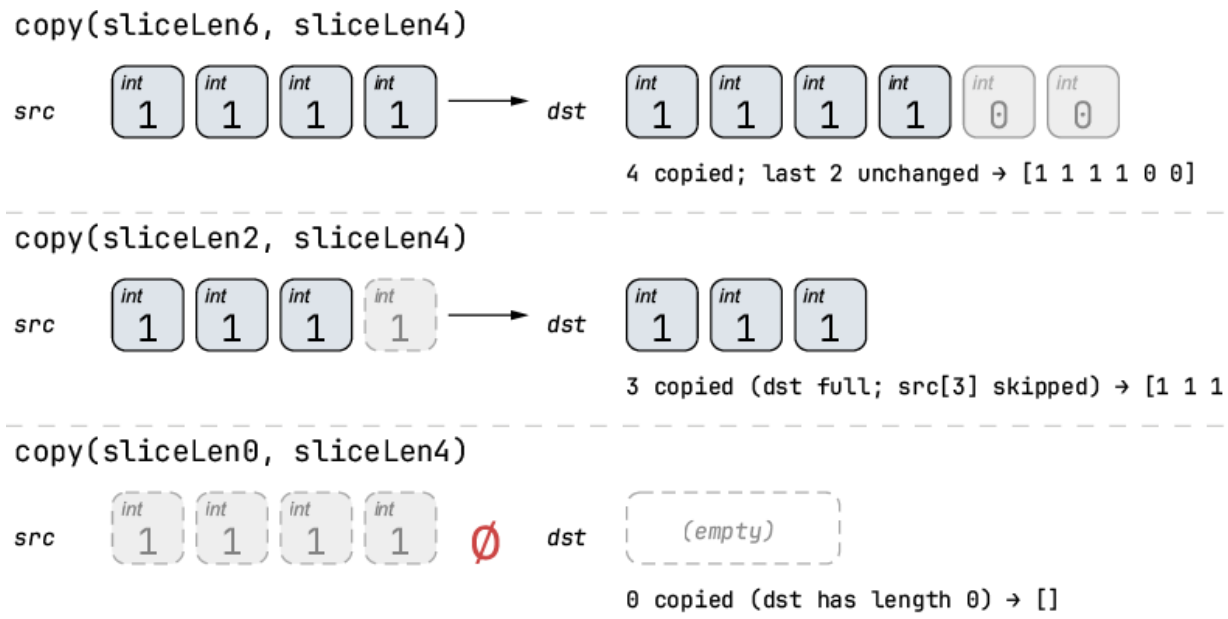


Figure 4.31 Copying a smaller slice into a larger slice will result in default values appended to the end.

Listing 4.52 Using copy on slices of different lengths

```
sliceLen6 := []int{0, 0, 0, 0, 0, 0}
sliceLen4 := []int{1, 1, 1, 1}
sliceLen2 := []int{2, 2, 2}
sliceLen0 := []int{}

copy(sliceLen6, sliceLen4)
fmt.Println(sliceLen6) // output: [1 1 1 1 0 0]

copy(sliceLen2, sliceLen4)
fmt.Println(sliceLen2) // output: [1 1 1]

copy(sliceLen0, sliceLen4)
fmt.Println(sliceLen0) // output: []
```

The `copy` function can be useful when you want to create a separate copy of a slice expression, or when you want to get a copy of a slice before or after calls to `append` that might modify it. You just need to ensure that you allocate a destination slice of the correct length to hold the copy. This is where `make` can be particularly helpful, since you can use

it to create a new slice of exactly the right length to hold the copied elements.

Listing 4.53 Using copy to create a separate copy of a slice expression

```
intSlice := []int{1, 2, 3, 4, 5, 6}
subSlice := intSlice[2:4] //[3 4]
subSliceCopy := make([]int, len(subSlice))
copy(subSliceCopy, subSlice)

// Change a value in the original slice.
intSlice[2] = 10
// Change is visible in subSlice, but not in subSliceCopy
fmt.Println(subSlice)     //output: [10 4]
fmt.Println(subSliceCopy) //output: [3 4]
```

4.2.8 Avoiding runtime panics when accessing slice values

Because the size of a slice isn't set until the program is running, you can sometimes run into a situation where a value you're expecting to find isn't there.

As an example, let's say you have a function that prints the top three finishers in a race.

Listing 4.54 Function to print the top three items

```
func printMedalists(raceName string, contestants []string) {
        fmt.Printf("Results for %s:\n", raceName)
        fmt.Printf("Gold Medal: %s\n", contestants[0])
        fmt.Printf("Silver Medal: %s\n", contestants[1])
        fmt.Printf("Bronze Medal: %s\n", contestants[2])
}
```

This function expects a sorted list with at least three entries, which it gets directly from the slice by index. If you pass a slice with more than three values, it works fine, but a slice with fewer than three values will cause a runtime panic.

Listing 4.55 Passing different lengths of slices to printMedalists

```
func main() {
        race1 := []string{"Sally", "Steve", "Jo", "Adam"}
        printMedalists("Race 1", race1)

        race2 := []string{"Jon", "Jen"}
        printMedalists("Race 2", race2)
}
```

Listing 4.56 Output from printMedalists with a runtime panic

```
Results for Race 1:
Gold Medal: Sally
Silver Medal: Steve
Bronze Medal: Jo
Results for Race 2:
Gold Medal: Jon
Silver Medal: Jen
panic: runtime error: index out of range [2] with length 2
```

The panic occurs because there's no graceful way for the Go runtime to handle a missing element that you explicitly asked for. The only thing it can do is stop the program immediately and tell you what went wrong.

Out-of-bounds errors on slices like this are among the most common panic conditions you'll encounter when starting out with Go, but fortunately, they are pretty easy to handle with a little foresight.

The first technique is to verify that you have at least the number of items you need by calling `len` on the incoming slice. In this case, you know that the function expects three entries, so you can just add a check for a shorter length and return an error otherwise.

Listing 4.57 Verifying slice size with len()

```
func printMedalists2(raceName string, contestants []string) error {
        if len(contestants) < 3 {
                return fmt.Errorf("not enough contestants for %s. W
ant: 3 Got: %d",
                        raceName, len(contestants))
        }
        fmt.Printf("Results for %s:\n", raceName)
        fmt.Printf("Gold Medal: %s\n", contestants[0])
        fmt.Printf("Silver Medal: %s\n", contestants[1])
        fmt.Printf("Bronze Medal: %s\n", contestants[2])
        return nil
}

func main() {
        race2 := []string{"Jon", "Jen"}

        // Test an invalid race with new error handling.
        err := printMedalists2("Race 2", race2)
        if err != nil {
                log.Println(err)
        }
}
```

Listing 4.58 Output from printMedalists2 with slice length checking

```
not enough contestants for Race 2. Want: 3 Got: 2
```

By adding a length check, you can ensure that accessing an invalid entry is impossible, since the function will just return an error otherwise. It also gives you the opportunity to log a helpful error message with the name of the invalid condition and the number of arguments you actually received. Of course, this means every call to your function needs to do error checking to ensure that nothing went wrong. This function is also pretty inflexible, since races with fewer than three people are invalid.

A much more flexible technique is to avoid direct index access altogether and write your code to handle the entire

slice with a `range` loop.

Listing 4.59 Using a range statement Instead of explicit indices

```
func printMedalists3(raceName string, contestants []string) {
        fmt.Printf("Results for %s:\n", raceName)
        for place, contestant := range contestants {
                switch place {
                case 0:
                        fmt.Printf("Gold Medal: %s\n", contestant)
                case 1:
                        fmt.Printf("Silver Medal: %s\n", contestan
t)
                case 2:
                        fmt.Printf("Bronze Medal: %s\n", contestan
t)
                default:
                        fmt.Printf("Finisher: %s\n", contestant)
                        // or just return
                }
        }
}

func main() {
        race1 := []string{"Sally", "Steve", "Jo", "Adam"}
        race2 := []string{"Jon", "Jen"}
        race3 := []string{}

        printMedalists3("Race 1", race1)
        // Race 2 only has two contestants, but we still need to li
st them!
        printMedalists3("Race 2", race2)
        // Throw in a race with no contestants at all.
        printMedalists3("Race 3", race3)
}
```

Listing 4.60 Output from printMedalists3 with a more flexible algorithm

```
Results for Race 1:
Gold Medal: Sally
Silver Medal: Steve
Bronze Medal: Jo
Finisher: Adam
Results for Race 2:
Gold Medal: Jon
Silver Medal: Jen
Results for Race 3:
```

By making the code more flexible, you can handle any number of contestants and even add new features, like printing fourth place and beyond. After all, sometimes just finishing is a victory! Because this code uses a `range` loop, it will repeat as few or as many times as it needs to or, in the case of no items at all, simply skip the loop altogether.

> **TIP** When working with slices, it is often more flexible to use a `range` loop than to attempt random index access.

Of course, another option is to do nothing at all and let the program panic if the truly unexpected occurs. If you always expect your slices to have a certain length, an exception to this might represent a significant enough bug that a panic will give a clear message that something is very wrong (panics have their purpose, after all). But if there's any valid possibility that you could get slices of an unexpected size, you might want to opt for one of the other techniques.

4.2.9 Nil slices versus empty slices

It might be tempting to use `if slice == nil` to check whether a slice is empty, but this can lead to surprising results. That is because slices are `nil` only if they have not been initialized or if they have been explicitly set to `nil` by the

programmer. A slice can have a `len` of `0` and still not be a nil slice.

Listing 4.61 Creating various slices of zero length

```
// An uninitialized slice is == nil.
var slice []int
fmt.Println(slice == nil) // true
fmt.Println(len(slice))   // 0

// A slice with an explicit len of zero is != nil.
sliceLenZero := make([]int, 0)
fmt.Println(sliceLenZero == nil) // false
fmt.Println(len(sliceLenZero))   // 0

// A slice made from an empty literal is != nil.
sliceEmptyLit := []int{}
fmt.Println(sliceEmptyLit == nil) // false
fmt.Println(len(sliceEmptyLit))   // 0

// A slice of a slice that has a length of zero is != nil.
sliceWithValues := []int{1, 2, 3}
slicedSlice := sliceWithValues[0:0]
fmt.Println(slicedSlice == nil) // false
fmt.Println(len(slicedSlice))   // 0

// The only way to get back to a nil slice is to set it to nil expl
icitly.
sliceLenZero = nil
fmt.Println(sliceLenZero == nil) // true
```

This code illustrates several different slices that are effectively empty but not all of which are equal to `nil` for different reasons as visualized in figure 4.32.

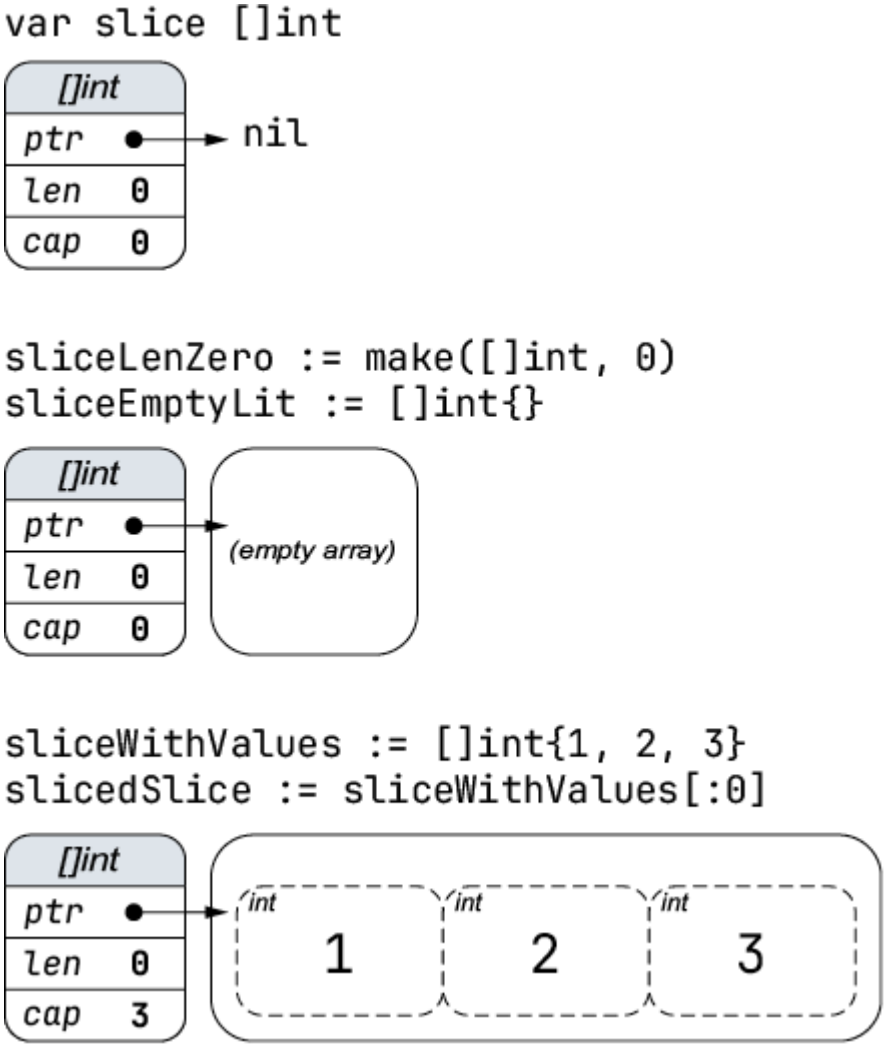


Figure 4.32 Different zero-length slices

`sliceLenZero` and `sliceEmptyLit` become initialized even if the backing array they point to is empty, because this can occur under valid conditions, such as explicit reallocation. In the case of `slicedSlice`, it still shares a backing array that has values, but the slice header itself has a length of `0`. This can happen when slices are programmatically resized and might not meet some threshold.

For this reason, you should always use a `len` check if you need to determine whether a slice is empty. Simply checking for `nil` equality will not always give you the answer you want.

Explicit `nil` checks should be reserved for cases when you want to tell the difference between a slice that somehow has a length of `0` and one that was never initialized in the first place.

4.3 Maps

Like arrays and slices, maps are collection types for storing multiple values of any type you need, but the way the values are indexed is much more flexible.

In arrays and slices, values are set and retrieved based on an integer index, which represents the value's position in the list. In maps, values are accessed using another value called the *key*, which can be almost any type you want. This lets you retrieve values quickly, using values that are more meaningful to your use case.

Looking at figure 4.33 as an example, let's say you need to write an inventory system for a grocery store, and you need to calculate the weights for shipping items in bulk. This would be kind of a pain with list types like arrays and slices because you'd have to keep two separate lists for both item names (strings) and weights (ints), but with maps, you can store them together.

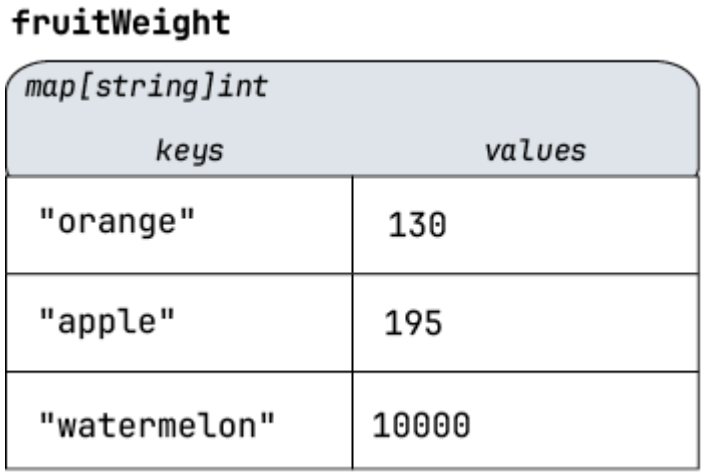


Figure 4.33 An example map

Listing 4.62 An example map

```
// fruitWeight stores the average weight of different kinds of frui
t
// in grams
fruitWeight := map[string]int {
        "orange"    : 130,
        "apple"     : 195,
        "watermelon": 10000,
}

// About how heavy are 10 apples and 30 oranges?
fmt.Println(10*fruitWeight["apple"] + 30*fruitWeight["orange"])
// output:
// 5850
```

In this example, the `fruitWeight` map stores the average weight of different kinds of fruit. The key type is `string`, while the value type is `int`, since weights are generally specified in numeric form.

You can think of a map as a bit like a dictionary, where definitions (values) are found by looking up the word (key) they are associated with.

Maps are valuable data structures because they provide a fast way to take a key, such as a string or a struct, and use

it as an index to store and retrieve values in the map. Figure 4.34 shows the relationship between keys and values in a map data structure.

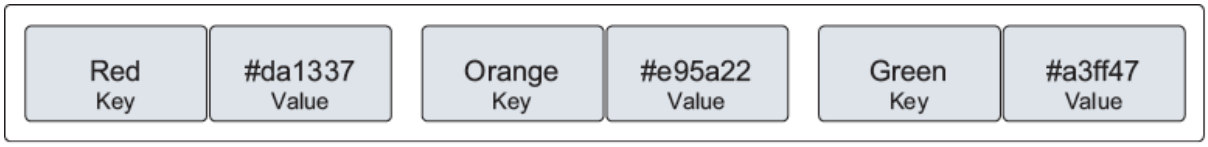
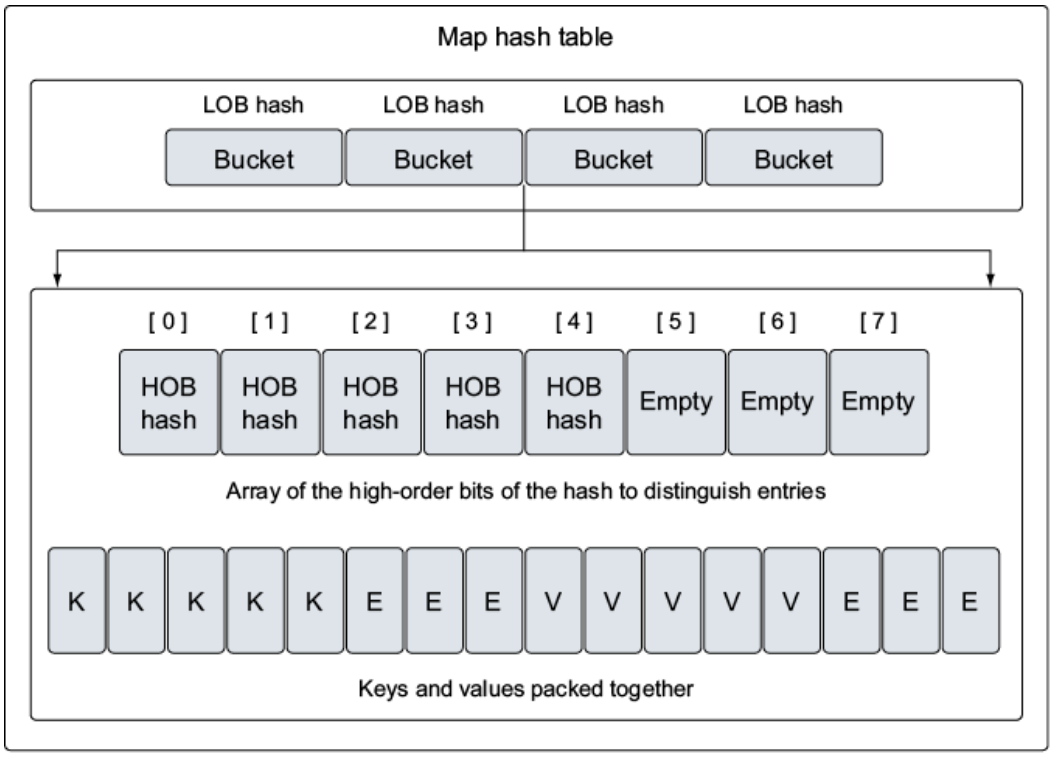


Figure 4.34 Relationship between keys and values

Internally, maps are implemented using hash tables, which contain a collection of buckets. You can see a visual representation of this in figure 4.35. When you're storing, removing, or looking up a key/value pair, everything starts with selecting a bucket. The map selects a bucket by passing the key specified in your map operation to the map's hash function.



K = Key V = Value E = Empty

Figure 4.35 Simple representation of the internal structure of a map

The purpose of the hash function is to generate an index that evenly distributes key/value pairs across all available buckets. You can see how this works in figure 4.36. The better the distribution, the more quickly you can find your key/value pairs as the map grows. If you store 10,000 items in your map, you don't want to ever look at 10,000 key/value pairs to find the one you want. You want to look at as few key/value pairs as possible. Looking at only 8 key/value pairs in a map of 10,000 items is a good, balanced map. A balanced list of key/value pairs across the right number of buckets makes this possible.

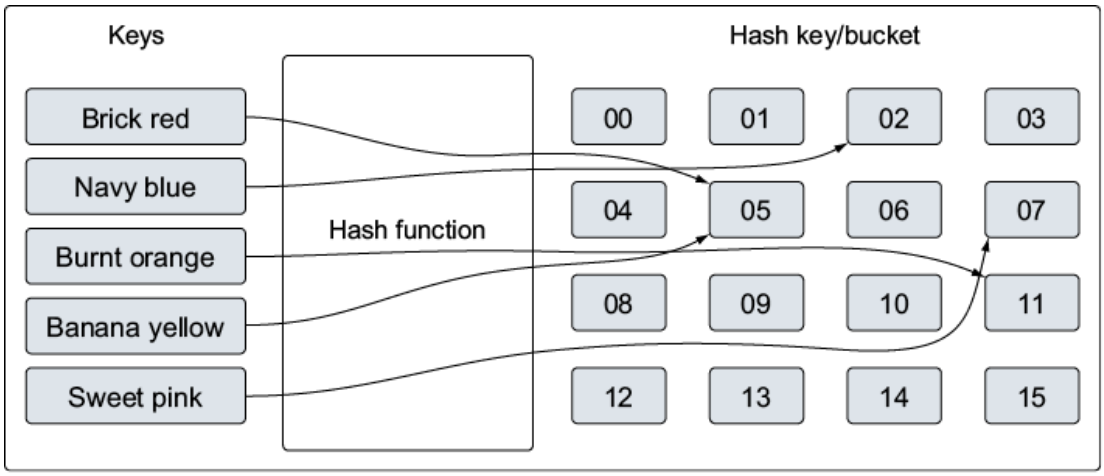


Figure 4.36 Simple representation of hash function placing keys in a map

The hash key that's generated for a Go map is a bit longer than it appears in figure 4.36, but it works the same way. In this example, the keys are strings that represent colors. Those strings are converted into numeric values within the scope of the number of buckets we have available for storage. The numeric value is then used to select a bucket for storing or finding the specific key/value pair. In the case of a Go map, a portion of the generated hash key, specifically the low-order bits (LOBs), is used to select the bucket.

Two data structures contain the data for the map. First, there's an array with the top eight high-order bits (HOBs) from the same hash key that was used to select the bucket. This array distinguishes each individual key/value pair stored in the respective bucket. Second, there's an array of bytes that stores the key/value pairs. The byte array packs all the keys and then all the values together for the respective bucket. This packing minimizes the memory required for each bucket.

Many other low-level implementation details about maps are outside the scope of this chapter. You don't need to understand all the internals to learn how to create and use maps. Just remember one thing: a map is an unordered collection of key/value pairs.

4.3.1 Declaring maps

Map declarations take the form of `map[K]E`, where `K` is the key type used for looking up values and `E` is the element type, which is the type of value stored in the map. Just like slices, maps can be created using `make`, or they can be created and initialized at the same time using map literals, which specify a starting list of key/value pairs, as shown in figure 4.37.

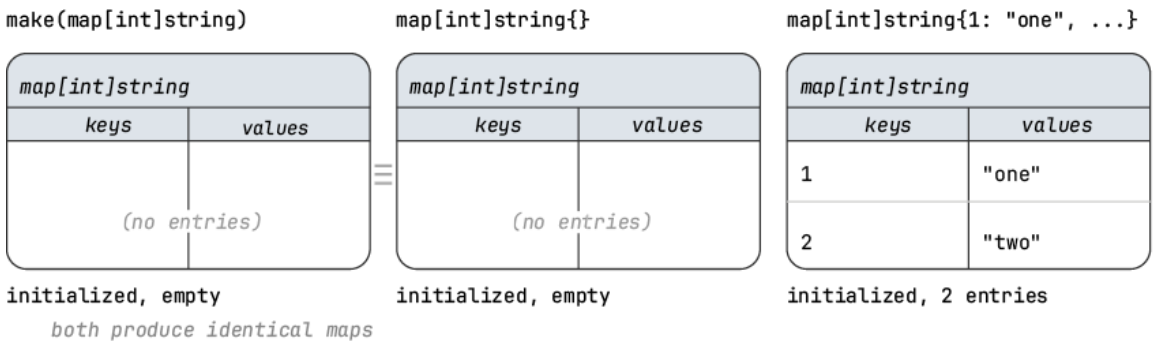


Figure 4.37 Map declarations are initialized in several different ways.

Listing 4.63 Declaring maps with make and literals

```
// Declare and initialize a map with make
intToString := make(map[int]string)

// Declare an empty map with a literal.
// (this is identical to using make)
intToString := map[int]string{}

// Declare and populate a map with a literal.
// Keys are separated from values with a colon, and commas separate
// key-value pairs.
intToString := map[int]string{
        1: "one",
        2: "two",
}
```

You can also declare map variables using the `var` keyword, but this comes with a caveat: values cannot be added to or retrieved from a map until it has been initialized, and attempts to do so will result in a runtime panic.

Listing 4.64 Declaring a map with var

```
// Declare a map variable using var
var intToString map[int]string

// Attempt to set a value in the map
intToString[1] = "one" // panic: assignment to entry in nil map
```

This is because maps declared without a literal are uninitialized, which means they do not have any memory allocated for their internal data structures, so they aren't set up to store values yet. This is known as a *nil map*. Like nil slices, nil maps are equal to `nil`, which you can check for. But unlike nil slices, there is nothing like `append` that can add values to a nil map, so it is easy to forget to initialize them and end up with a panic. For this reason, you should prefer using `make` or map literal initialization when you can. If a

map value is optional, you need to remember to check for `nil` first.

> **IMPORTANT** Nil maps *must* be initialized before they can be used. When in doubt, check to see if the map is `nil` first.

You can avoid nil map panics by checking whether a map value is `== nil` and initializing it if so.

Listing 4.65 Initializing a map if it Is a nil map

```
// Declare a map variable using var
var intToString map[int]string

if intToString == nil {
        intToString = map[int]string{}
}

// Assignment succeeds after the map has been initialized.
intToString[1] = "one"
```

Because map literals can create new maps and optionally populate them with values, they are generally the preferred method for allocating new map values. They are also somewhat shorter because you don't need two separate steps to create the map and populate it.

> **TIP** Map literals are the most flexible and idiomatic way to initialize maps in Go. Use them unless you have a good reason not to.

4.3.2 Setting and accessing map values

To set a map value, you need to provide a key and an element of the correct types. If a value already exists for the given key, it will be overwritten.

Listing 4.66 Assigning values for different map types

```
colors := map[string]string{}
// The key type of colors is string, so the indices are strings
// representing the name of the color.
colors["red"] = "#FF0000"
colors["green"] = "#00FF00"

// Assigning a value using an existing key will overwrite the old v
alue.
colors["red"] = "#DD2222" // (replaces "#FF0000")

userIDs := map[string]int{}
// The element type of userIDs is int, so the values are integers.
userIDs["Andy"] = 1
userIDs["Sarah"] = 2

// Variables can be used for both the key value and the element val
ue.
userName := "Brian"
userID := 3
userIDs[userName] = userID
```

When retrieving values from a map, you can access values by their key. If the key does not exist, Go returns the zero value for the element type instead. If you need to test whether a particular key exists, you can use the two-variable assignment form, which returns a second Boolean value representing whether the key was found in the map.

Listing 4.67 Retrieving values from a map

```
userIDs := map[string]int{
        "Andy": 1,
        "Sarah": 2,
}

AndyID := userIDs["Andy"] // 1
JenID := userIDs["Jen"] // 0 (key is not in the map)

JeffID, found := userIDs["Jeff"] // 0, false

// You can also omit the value variable if you are just checking fo
r
// membership. Using if with a simple-statement if is a common idio
m.
if _, found := userIDs["Carmen"]; !found {
        fmt.Println("Didn't find user Carmen")
}
```

It might seem odd at first that maps return values for keys that don’t exist, but this behavior enables some handy patterns when the existence of the key isn’t important.

One example might be a membership list, using `string` as the key type (representing a member’s name) and `bool` as the element type, representing whether that person is a member of the club.

Listing 4.68 Using zero values for membership queries

```
clubMembers := map[string]bool{
        "Susan": true,
        "Aditya": true,
}

for _, name := range []string{"Aditya", "Andy", "Susan"} {
        if clubMembers[name] {
                fmt.Printf("%s is a member.\n", name)
        } else {
                fmt.Printf("%s is not a member.\n", name)
        }
}

// output:
// Aditya is a member.
// Andy is not a member.
// Susan is a member.
```

Because the zero value of `bool` is `false`, you can get away with storing only members who are in the club, and all other queries will return `false`. This is a common way to create sets in Go because you can check for membership with a simple `if` statement. For larger sets, there is another technique using empty structs that is slightly less readable, but consumes less memory. We'll go over this in chapter 6.

Another case where this might be useful is looking up numeric data for an entity that might or might not be tracked yet.

Listing 4.69 Using the zero value with numeric data

```
wordsWritten := map[string]int {
        "Andy": 1000,
        "Jen": 500,
}

for _, name := range []string{"Jen", "Andy", "Eve"} {
        fmt.Printf("%s has written %d words\n", name, wordsWritten
[name])
}

// output:
// Jen has written 500 words
// Andy has written 1000 words
// Eve has written 0 words
```

In this example, `"Eve"` is not in the map, so you can assume they have written zero words and let the zero value speak for itself.

This attribute makes maps more consistent to work with and means that value access will never cause a panic. It also allows you to decide whether key existence is important. When you need to distinguish between "not in the map" and "in the map, but with a zero value," you can use the two-value assignment as needed.

4.3.3 Removing values from maps

The built-in `delete` function is used to remove items from a map by key. Removing keys that do not exist is safe and will not result in an error.

Listing 4.70 Removing a value from a map with delete

```
countryCodes := map[string]string{
        "FR": "France",
        "CN": "China",
        "ES": "Spain",
        "TX": "Texas",
}
fmt.Println(countryCodes)
// output: map[CN:China ES:Spain FR:France TX:Texas]

// Delete the value associated with key "TX" from the map
delete(countryCodes, "TX")
fmt.Println(countryCodes)
// output: map[CN:China ES:Spain FR:France]

// The "XX" key does not exist, so no changes will be made to the m
ap, and
// no error will result.
delete(countryCodes, "XX") // Key "XX" does not exist, so no action
will be
fmt.Println(countryCodes)
// output: map[CN:China ES:Spain FR:France]
```

4.3.4 Getting the number of values in a map

To get the number of values in a map, you can use the `len` built-in.

Listing 4.71 Using len to get the number of keys/values in a map

```
petWeightsKG := map[string]float64 {
        "Baxter": 8.6,
        "Barry": 3.17,
        "Bobby": 9.07,
        "Benny": 1.08,
}

numberOfPets := len(petWeightsKG) // 4
```

4.3.5 Iterating over maps

To iterate over a map, you can use a `range` loop, which returns a single map key for each iteration, along with an optional copy of the value.

Listing 4.72 Iterating over a map using for and range

```
groupNouns := map[string]string{
        "eagle":    "convocation",
        "cat":      "clowder",
        "kangaroo": "mob",
        "dog":      "pack",
}

// Iterate with keys only.
for animal := range groupNouns {
        fmt.Printf("A group of %ss is called a %s.\n", animal, grou
pNouns[animal])
}

// Iterate with keys and values.
for animal, group := range groupNouns {
        fmt.Printf("A group of %ss is called a %s.\n", animal, grou
p)
}
```

Because of the way maps are implemented, they have no specific order, and the order in which you receive keys from the `range` statement can differ each time. This is by design, since map order is unspecified and can differ from architecture to architecture. If you run the preceding code multiple times, you will likely see the order change.

Listing 4.73 Map iteration output from multiple runs

```
A group of cats is called a clowder.
A group of kangaroos is called a mob.
A group of dogs is called a pack.
A group of eagles is called a convocation.

A group of eagles is called a convocation.
A group of cats is called a clowder.
A group of kangaroos is called a mob.
A group of dogs is called a pack.
```

Map iteration can also be combined with the `delete` function, allowing you to examine each item in the map and remove it if necessary. This is useful for "filtering" a map by some criteria.

Listing 4.74 Filtering a map using a range loop

```
familyAges := map[string]int{
        "Grandpa": 83,
        "Homer":   36,
        "Marge":   34,
        "Bart":    10,
        "Lisa":    8,
        "Maggie":  1,
}

for person, age := range familyAges {
        if age < 18 {
                delete(familyAges, person)
        }
}
fmt.Println(familyAges)
// output: map[Grandpa:83 Homer:36 Marge:34]
```

In the preceding example, the list is filtered based on whether each person is over the age of 18. This is safe to do in Go, provided the map is not being accessed elsewhere at the same time. You will learn more about concurrent map access safety in chapter 9.

4.3.6 Passing maps to functions

Like slices, maps are fundamentally reference types because they point to an underlying data structure allocated by the runtime. This means that when you pass a map to a function, any modifications made to the map's contents inside that function will be visible to the caller.

4.3.7 Key and value types

You can use any type you like for map elements, but key types are slightly more restricted. The only requirement for key types is that they must be comparable using the `==` operator. Types that are not comparable in this way cannot be key types, so you cannot use functions, slices, or maps for your key type, or any structs that contain them. Everything else is fair game.

This leads to some interesting possibilities, such as maps that use a struct type as a key and slices as values.

Listing 4.75 Map with a struct type key and a slice value

```
favoriteFoods := map[person][]string{}

andy := person{
        firstName: "Andy",
        lastName:  "Walker",
        age:       42,
}

john := person{
        firstName: "John",
        lastName:  "Smith",
        age:       53,
}

favoriteFoods[andy] = []string{"jambalaya", "bulgogi"}
favoriteFoods[john] = []string{"crab cakes", "licorice"}

        johnFoods, found := favoriteFoods[person{
                firstName: "John",
                lastName:  "Smith",
                age:       29,
        }]
        // The "firstName" and "lastName" fields are the same, but
the "age"
        // field is different, so there is no match.
        fmt.Println(johnFoods, found) // output: [], false

        // All fields match a key value in the map, so the element
value is
        // returned.
        andyFoods, found := favoriteFoods[person{
                firstName: "Andy",
                lastName:  "Walker",
                age:       42,
        }]
        fmt.Println(andyFoods, found) // output: [jambalaya bulgog
i] true
```

Summary

- Arrays are constant-sized collections of data that form the basis of the slice and map types.
- Slices are powerful, flexible, ordered collection types, and they are the primary ordered collection type in Go.
- `append` works by returning a new slice header, though it does not always allocate new storage.
- Maps are Go's associative array, or "dictionary," type, and they allow you to store values of any type using a key type best suited for your use case.
- Slices and maps are fundamentally reference types that point to the data they contain, so it helps to be mindful of how they work and when you should modify them.

5 Working with types

This chapter covers

- How Go's type system lets you model real-world domains using named types, structs, and composition
- Defining and using custom types, struct embedding, and collections to organize and extend your data
- Attaching methods and interfaces to types to add behavior and enforce contracts
- Working with pointers, type assertions, and type switches to manage memory and check concrete types at runtime

In computer science, *types* define the properties and rules for representing data. We have already encountered a few built-in types, called *primitive types*, such as integers, floats, Booleans, and strings, which represent basic values like numbers, truth values, and text. A programming language's *type system* is the set of rules and features it provides for defining and working with types.

But real-world problems often require us to model more complex concepts than primitive types allow. By combining these types, we can create *composite* or *complex types* that better represent real-world entities and their relationships. This allows us to model domains more accurately, enforce rules in our code, and build safer, more expressive programs.

Go's type system is one of its most powerful features, allowing you to model real-world objects in a way that is both expressive and safe. Beyond the primitives we've

already covered, Go enables you to define your own types using structs, type aliases, and interfaces, giving you the flexibility to represent domain-specific concepts directly in your code.

One of the key benefits of Go's type system is its emphasis on type safety. By enforcing strict type checking at compile time, Go helps you catch many classes of bugs early in the development process, reducing runtime errors and making your programs more robust. The type system also supports composition over inheritance, encouraging you to build complex types by combining simpler ones, which leads to code that is easier to understand and maintain. *Type checking* is the process of verifying that the values assigned within a defined structure are correct, unlike unstructured data, where you can assign any number of properties and values without a compiler complaining.

In this chapter, we'll explore how you can use Go's type system to build a flexible recipe application, demonstrating how you can define new types, compose them, and use interfaces to add behavior. By using these features, you can create programs that are not only correct and efficient, but also clear and idiomatic.

5.1 Modeling a domain with types

Let's consider a recipe application example to help us explore the various concepts around types. We'll use Go's type system to represent the relationships and behaviors involved.

The set of types that describes a particular area of work is known as a *domain*. Domains are extremely helpful when thinking about your application and how data is represented. If we consider the domain of a recipe, certain

properties come to mind: ingredients, measurements, steps, and so on. Each property can be represented by numbers, strings, or many other types.

In Go, you define your own types using the `type` keyword, creating new named types that more clearly express your domain, either by aliasing existing primitive types or by composing more complex structures. This process involves specifying the `type` keyword, the name you choose, and the underlying Go type or type literal. By introducing your own types, you give the compiler more information about your intent, which helps catch errors and makes your code more readable. Think of types as containers that specify what kind of data can be stored and how it can be used, enabling you to model your domain more accurately and write code that is both expressive and safe.

For example, you might define a new type to represent a measurement, such as a weight or a volume, rather than just using a plain `float32` or `string`. You can also create structs to group related data together, such as an ingredient with its quantity and unit. By modeling your domain with types, you make your code safer, clearer, and easier to maintain.

Let's see how this works in practice as we build up a recipe application.

5.1.1 Named types

The simplest form of user-defined types in Go is called a *named type*, though since Go 1.9, the official language specification has used the term *defined type*—you may encounter both. When you write `type Magnitude float32`, you create a new, distinct type. Even though it shares the same underlying representation as `float32`, the compiler treats

them as different types and will refuse to mix them without an explicit conversion. This is different from a *type alias* (written as `type A = B`), which simply gives an existing type an alternative name without creating a new type; `A` and `B` remain completely interchangeable. Defined types can have their own methods, whereas type aliases cannot add methods.

A named (defined) type allows you to create a new type based on an existing primitive type, such as `int`, `float32`, or `string`. This gives you the flexibility to add meaning, constraints, or methods to otherwise generic data. By introducing named types, you make your code more expressive and self-documenting, because the type names communicate intent and domain concepts directly. Figure 5.1 shows the relationship between the source code and how the compiler translates the various types into a program.

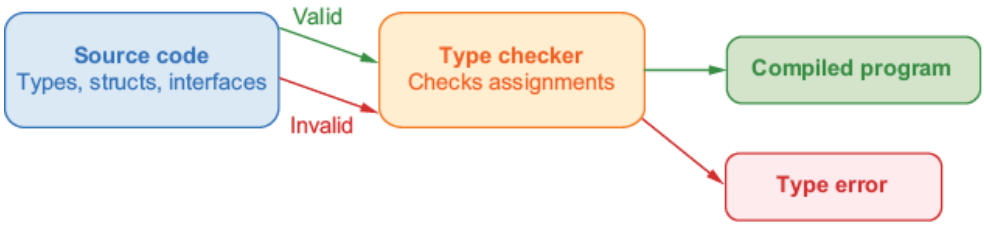


Figure 5.1 Type checking allows the compiler to determine whether the code is valid before the system runs.

For example, in our recipe application, we need to represent measurements, such as "200 grams" or "1 cup," which always consist of two pieces of information: the magnitude (the amount) and the unit (how it is measured). In addition, for the unit, we will want to know the system in which the unit exists, such as imperial or metric. Although this may be implied, we want to make some information explicit so that we can limit possible bugs and add additional features. Instead of using plain numbers and strings everywhere, we

can define named types to clarify their roles and prevent accidental misuse.

```
type MeasurementSystem string #1

// Magnitude represents the numeric value of a measurement.
// Unit represents the unit of measurement (e.g., grams, cups).
type Magnitude float32 #2
type Unit string #3
```

#1 A MeasurementSystem can represent a group of units based on a string name.
#2 Magnitude allows us to use a number with a decimal to represent the amount of a given unit that is present.
#3 Unit is again represented by a string for clarity.

By defining `Magnitude` and `Unit` as named types, we make it clear when a value is meant to be a measurement, rather than anything else a number or string might be used for. This helps catch errors at compile time, and it makes the code easier to read and maintain.

Recipes use many different units for measuring ingredients, in both the metric and imperial systems, but the set of possible units is finite and known ahead of time. In some programming languages, you might use an `enum` (enumeration) to represent this set of possibilities. Go does not have a built-in `enum` type, but you can achieve a similar effect by declaring a named type and then defining a set of constants for the allowed values.

For our recipe builder, we'll define common units for weight and volume as constants of the `Unit` type. This approach allows us to refer to units by name throughout our code, reducing the risk of typos and making it easy to update or extend the list of units in the future.

```
// Systems
const (
        Metric   MeasurementSystem = "metric" #1
        Imperial MeasurementSystem = "imperial"
)

// Units for weight and volume, defined as constants of type Unit.
const (
    Gram     Unit = "gram"
    KiloGram Unit = "kilogram"
    Ounce    Unit = "ounce"
    Pound    Unit = "pound"
    Liter    Unit = "liter"
    Cup      Unit = "cup"
    Pint     Unit = "pint"
    Quart    Unit = "quart"
    Gallon   Unit = "gallon"
)
```

#1 Using constants will allow you to predefine enumerated values for a type.

By combining named types and constants, we create a strong foundation for modeling measurements in our recipe application. This makes our code safer, more expressive, and easier to extend as we add new features, such as unit conversion or validation.

5.1.2 Structs

Structures are one of the most important composite types in Go. A *struct* groups zero or more values, potentially of different types, into a single entity. This makes structs essential for modeling real-world objects and concepts by bundling related data together. Previously, we created named types like `Magnitude` and `Unit` to clarify the roles of individual values. Now we can combine these into a struct to represent a complete measurement. Structs not only organize related data but also allow you to attach methods

and behaviors, making your types more expressive and useful.

Structs are especially useful for representing entities with multiple attributes or pieces of state. For example, a measurement in a recipe consists of both a magnitude (the amount) and a unit (such as grams or cups). By using a struct, you can combine these related values into a single, coherent type, making your code more organized and expressive, and less error-prone.

In Go, you define a struct type using the following syntax:

```
type TypeName struct {
    FieldName1 FieldType1
    FieldName2 FieldType2
    // ... more fields as needed
}
```

This definition creates a new type called `TypeName` with the specified fields. Each field has a name and a type, and you can access them using dot notation as in the following example.

```
var foo = TypeName{}
fmt.Println(foo.FieldName1)
```

Structs can contain any combination of built-in types, named types, or even other structs, making them highly flexible for modeling complex data.

In the following example, `Measurement` is defined as a struct type containing two fields (`Magnitude` and `Unit`), which together represent a single measurement:

```
type Measurement struct {
    Magnitude Magnitude #1
    Unit      Unit
}
```

#1 Properties can use built-in or user-defined types.

The `Magnitude` and `Unit` fields will be automatically defined for every value of the `Measurement` type. Once you have defined a struct type, you can use it just like any other built-in type: declare variables, assign values, and access its fields.

Let's see how we could create and use a `Measurement` value in practice:

```
import "fmt"

func main() {
    // Create a Measurement value with both magnitude and unit.
    m := Measurement{ #1
        Magnitude: 200,
        Unit:      Gram,
    }
    fmt.Printf("Measurement: %.2f %s\n", m.Magnitude, m.Unit) #2
}
```

#1 Create a new measurement variable.
#2 Access values through dot notation.

In this example, we create a `Measurement` representing 200 grams and print its fields. Notice that the field names (`Magnitude` and `Unit`) are capitalized. In Go, case determines visibility: capitalized fields and methods are *exported* (public), while lowercase ones are *unexported* (private to the package). This is Go's way of controlling access, similar to `public` and private in other languages.

But what happens if you omit a field when creating a struct? Go automatically assigns the *zero value* for that field's type.

For example, if you leave out `Magnitude`, it defaults to `0` (the zero value for numbers):

```
import "fmt"

func main() {
    // Create a Measurement value with only the unit specified.
    m := Measurement{
        Unit: Gram,
    }
    fmt.Printf("Measurement: %.2f %s\n", m.Magnitude, m.Unit)
}
```

Here, `m.Magnitude` prints as `0.00` because it was not set. Similarly, if you omit `Unit`, it defaults to the zero value for strings, which is an empty string. Understanding zero values is important when working with structs because it helps you anticipate default behavior and avoid subtle bugs.

Now that you've seen how to define and use structs, let's explore how Go enables you to extend types and model more complex relationships using type embedding.

5.2 Extending types

In object-oriented programming (OOP), inheritance allows one class (often called a "child" or "subclass") to inherit the properties and methods of another class (the "parent" or "superclass"). This enables code reuse and the creation of hierarchical relationships between types. Go does not support traditional inheritance as found in languages like Java or C++. Instead, Go uses a concept called "type embedding" to achieve similar behavior.

5.2.1 Type embedding

With type embedding, you can include one type within another. This means the new type automatically has access to all the fields and methods of the embedded type, almost as if it had inherited them. The new type can also define its own fields and methods and can even override methods from the embedded type by defining a method with the same name. This approach allows you to extend and customize types without the complexity of classical inheritance, as you can see in figure 5.2.

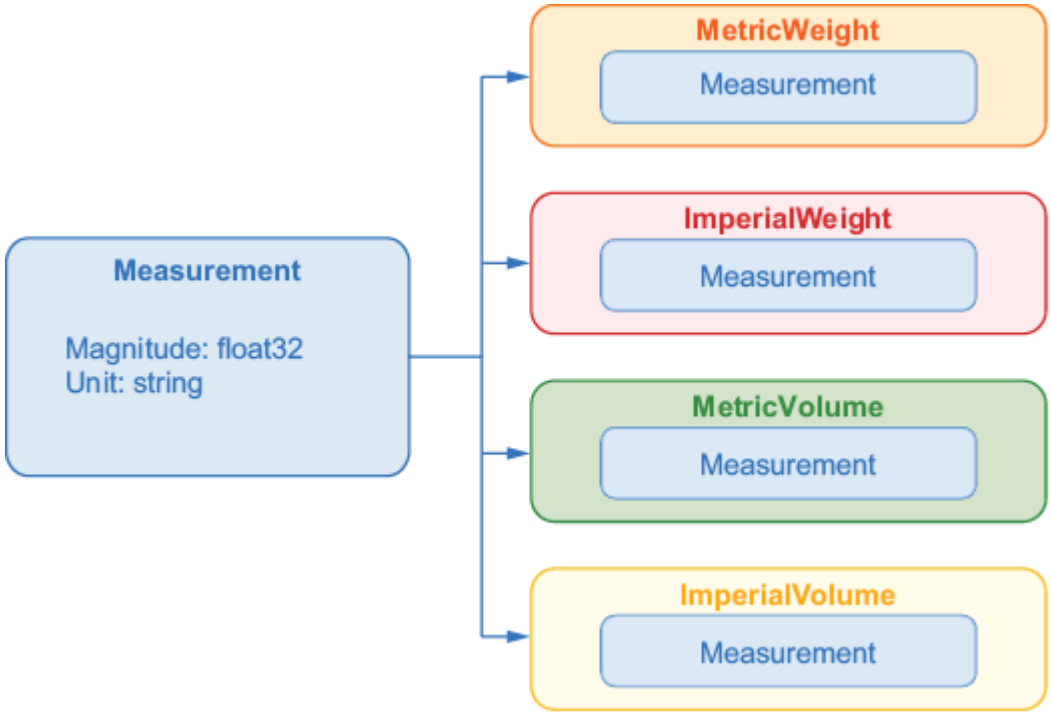


Figure 5.2 Type embedding allows you to extend structs by creating a new type from an existing type.

Let's demonstrate this by creating specific types for our measurements. We'll start with a general measurement type and then create more specialized types by embedding the general type within them.

```
type MetricWeight struct{ Measurement } #1
type MetricVolume struct{ Measurement }
type ImperialWeight struct{ Measurement }
type ImperialVolume struct{ Measurement }
```

#1 All that is needed to extend a type is to include it without a name as part of a structure's creation.

By embedding the `Measurement` type, each specialized type (such as `MetricWeight` or `ImperialVolume`) automatically gains access to all the fields and methods of `Measurement`. This approach lets you specify the measurement system and unit more explicitly, which is useful when adding logic or behavior specific to each type. When constructing these types, you provide the embedded `Measurement` as part of the initialization.

```
func main() {
    // Create a MetricWeight value with only the unit specified.
    mw := MetricWeight{
        Measurement: Measurement{
            Unit: Gram,
        },
    }
    fmt.Printf("MetricWeight: %.2f %s\n", mw.Magnitude, mw.Unit)

    // Create an ImperialVolume value with both magnitude and unit.
    iv := ImperialVolume{
        Measurement: Measurement{
            Magnitude: 2,
            Unit:      Quart,
        },
    }
    fmt.Printf("ImperialVolume: %.2f %s\n", iv.Magnitude, iv.Unit)
}
```

5.2.2 Complex types

So far, you've seen how to create new types from existing primitives and how to compose types using `structs`. But Go's

type system also provides powerful ways to work with *collections*, or homogeneous groups of values that you can treat as a single entity. These collection types, such as slices and maps, are essential for modeling real-world data when you need to manage sets or lists of items. Collections allow you to act on groups of data as a whole, providing abstractions that help you reason about your problem domain more effectively.

In Go, the most common collection types are *slices* (dynamic arrays) and *maps* (key/value stores). By defining your own named types based on these collections, you can add meaning and structure to your code, making it safer and more expressive.

Let's extend our recipe example by introducing several new types that use these collection concepts.

```
type Ingredient string

type IngredientMeasurement struct {
    Ingredient  Ingredient
    Measurement Convertible #1
}

type IngredientList []IngredientMeasurement #2

type Step struct {
    Description string
    Ingredients IngredientList
}
```

#1 Using the Convertible interface here (defined in section 5.4) allows any measurement type, metric or imperial, to be stored and converted.
#2 A type can be created from a slice or map.

Each of these types serves a specific purpose in modeling a recipe. The `Ingredient` type is a named type based on `string`, representing the name of an ingredient. Using a

named type instead of a plain string makes the code more self-documenting and helps prevent accidental misuse. The `IngredientMeasurement` struct pairs an `Ingredient` with a `Convertible` value. Later in this chapter, we will define methods on our measurement types to satisfy the `Convertible` interface, ensuring that any stored measurement can be converted between unit systems. The `IngredientList` type is a slice of `IngredientMeasurement` values, representing a list of measured ingredients, such as those needed for a particular step in a recipe. Finally, the `Step` struct represents a single step in a recipe, including a description and the list of ingredients used in that step. By combining these types, we can clearly express the structure of a recipe in our code.

Next, let's define the `Recipe` type, which brings everything together:

```
// Recipe type.
type Recipe struct {
    Title string
    Yield uint #1
    Steps []Step
}


type RecipeBox []Recipe
```

#1 uint (unsigned integer) is used because a serving count is always a positive whole number; note this also means the Scale method only accepts positive integer factors.

The `Recipe` type is a struct that represents a complete recipe, including its title, yield (how many servings it makes), and a slice of `Step` values that detail the instructions and ingredients for each part of the recipe. The `RecipeBox` type is defined as a slice of `Recipe` values, allowing you to group multiple recipes, such as in a cookbook or a user's personal collection.

Go’s collection types aren’t limited to slices. *Maps* are another powerful tool that allow you to associate values with unique keys. For example, you might want to keep track of the total amount of each ingredient needed across multiple recipes, which is useful for generating a grocery list.

Here’s how you can define a map-based collection type for this purpose:

```
type GroceryList map[Ingredient]Measurement
```

This type maps each `Ingredient` to a `Measurement`, representing the total quantity needed.

Let’s see how you might use this in practice. Suppose you want to generate a grocery list from a set of recipes. You can write a function that takes a `RecipeBox` and returns a `GroceryList` by iterating over all the ingredients in all the recipes, summing the quantities as you go.

```
func CreateGroceryList(recipes RecipeBox) GroceryList {
    shoppingList := make(GroceryList)  #1
    for _, recipe := range recipes {  #2
        for _, step := range recipe.Steps {  #3
            for _, im := range step.Ingredients {  #4
                metric := im.Measurement.ToMetric()  #5
                if existing, found := shoppingList[im.Ingredient];
found {  #6
                    shoppingList[im.Ingredient] = Measurement{
                        Magnitude: existing.Magnitude + metric.Magn
itude,  #7
                        Unit:      metric.Unit,  #8
                    }
                } else {
                    shoppingList[im.Ingredient] = metric  #9
                }
            }
        }
    }
    return shoppingList #10
}
```

#1 Create an empty GroceryList (map[Ingredient]Measurement).
#2 Iterate over each recipe in the RecipeBox.
#3 Iterate over each step in the recipe.
#4 Iterate over each IngredientMeasurement in the step.
#5 Convert measurement to metric system for consistency.
#6 If ingredient already in list, sum quantities.
#7 Add magnitudes together.
#8 Use metric unit for all entries.
#9 Add new ingredient to the list.
#10 Return the completed grocery list.

This function ensures that all ingredient quantities are summed in a consistent unit (metric, in this case), making it easy to generate a consolidated shopping list. This works because `IngredientMeasurement.Measurement` is typed as `Convertible`, which guarantees that the `ToMetric()` method is available on every stored measurement. We'll define the `Convertible` interface and its implementations in section 5.4.

By defining your own collection types, whether slices or maps, you can encapsulate logic and add methods that operate on these collections, further strengthening your code. The more you use Go's type system to model your domain, the more robust and maintainable your programs will be. Still, keep in mind that stronger typing often requires more up-front design and thought, so strive for a balance that fits your application's needs.

5.3 Pointer types

When you instantiate, or create, a type in Go, a portion of memory is allocated to store the value based on the type's definition. If you assign this value to a variable, that variable holds its own copy of the data. When you assign this variable to another variable, Go creates a new copy of the value in memory. This behavior is called *copy by value,* and it is the default in Go for most types, such as numbers, strings, structs, and arrays. Copy by value helps prevent many classes of bugs by ensuring that changes to one variable do not accidentally affect another.

Consider this example:

```
var i = "foo"
var j = i
fmt.Printf("i: %s, j: %s\n", i, j)
i = "bar"
fmt.Printf("i: %s, j: %s\n", i, j)
```

In this code, `j` gets its own copy of the value `"foo"`. When we later change `i` to `"bar"`, `j` remains unchanged, still holding `"foo"`. This is because the assignment `j = i` copies the value, not the memory location.

Sometimes you want two variables to refer to the same underlying value in memory, so that changes made through one variable are visible through the other. This is where *pointer types* come in. A pointer is a variable that stores the memory address of another value, rather than the value itself. In Go, you declare a pointer type using the `*` operator, and you obtain the address of a variable using the `&` operator.

Let's see how pointers work by modifying the previous example:

```
var i = "foo"
var j = &i // j is a pointer to i
fmt.Printf("i: %s, j: %s\n", i, *j)
i = "bar"
fmt.Printf("i: %s, j: %s\n", i, *j)
```

Here, `j` is a pointer to `i` (it holds the memory address of `i`). When we print `*j`, we are *dereferencing* the pointer, which means we are accessing the value stored at the memory address that `j` points to. After we change `i` to `"bar"`, dereferencing `j` also gives us `"bar"` because both `i` and `*j` refer to the same memory location.

Pointers are especially useful in Go because they let you efficiently pass large structs or arrays to functions without copying all their data, which can improve performance and reduce memory usage. Pointers also enable functions or methods to modify the original value passed in, rather than working with a copy. Additionally, pointers can represent optional or missing values, since the zero value of a pointer is `nil`.

Figure 5.3 illustrates how pointers work in Go at the memory level. When you create a pointer to a struct, the pointer variable holds the memory address of the struct's

data rather than the data itself. This means multiple variables can reference the same underlying memory, and changes made through one pointer are reflected everywhere that memory is accessed. In the figure, you can see a pointer variable (`ptr`) storing the address of a `Measurement` struct, with an arrow showing how the pointer "points to" the struct's location in memory. The struct's fields are stored in contiguous memory slots, and the pointer provides direct access to them. This visual representation helps clarify how pointers enable efficient sharing and manipulation of data in Go programs.

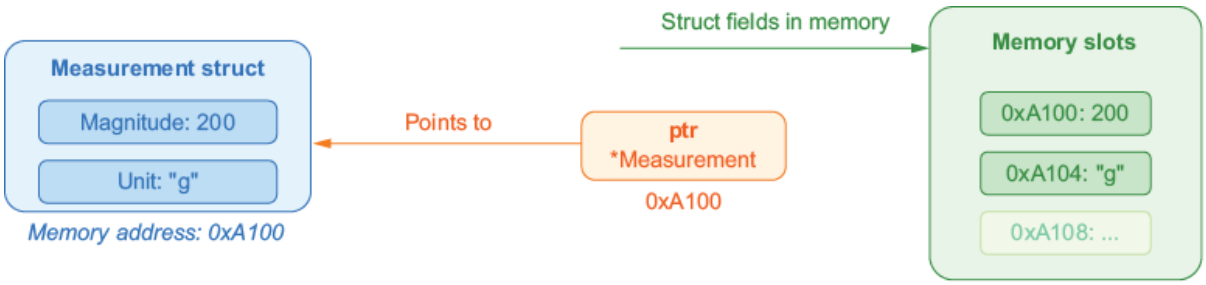


Figure 5.3 Pointers help programs manage shared sections of memory across an application.

You can also create pointers using the built-in `new` function, which allocates memory for a value of the given type and returns a pointer to it:

```
p := new(int) #1
*p = 42       #2
fmt.Println(*p) #3
```

#1 The new function allocates memory for an int and returns its address (pointer).
#2 Dereferencing the pointer allows you to set the value at the allocated memory location.
#3 Printing the dereferenced pointer displays the value stored at that memory address.

When working with structs, Go makes pointer use ergonomic. If you have a pointer to a struct, you can access

its fields directly using the dot (`.`) operator; Go automatically dereferences the pointer for you:

```
recipe := &Recipe{
    Title: "Pancakes",
    Yield: 2,
    Steps: []Step{
        {
            Description: "Mix dry ingredients",
            Ingredients: []IngredientMeasurement{
                {"Flour", MetricWeight{Measurement{Magnitude: 200,
Unit: Gram}}},
                {"Sugar", MetricWeight{Measurement{Magnitude: 50, U
nit: Gram}}},
            },
        },
        {
            Description: "Add wet ingredients",
            Ingredients: []IngredientMeasurement{
                {"Milk", MetricWeight{Measurement{Magnitude: 300, U
nit: Gram}}},
                {"Egg", MetricWeight{Measurement{Magnitude: 50, Uni
t: Gram}}},
            },
        },
    },
}
fmt.Println(recipe.Title) #1
```

#1 Prints "Pancakes" (Go automatically dereferences recipe)

There are a few important notes about pointers in Go. First, the zero value of a pointer is `nil`, which means that it does not point to any valid memory. Go does not support pointer arithmetic (unlike C or C++), which makes pointers safer and less error-prone. Additionally, slices, maps, and channels are reference types in Go, so assigning them to a new variable does not copy the underlying data; they already behave like pointers in many ways.

5.4 Methods and behavior

Go allows you to associate methods with types, enabling you to add behavior directly to your data structures. Methods are functions with a special receiver argument, which specifies the type the method is attached to. This makes your code more organized and expressive because you can group related data and behavior together. To define a method, you use the `func` keyword, followed by the receiver in parentheses, the method name, and its parameters. The receiver can be either a value or a pointer to the type. The choice between value and pointer receivers is important and affects how your methods interact with the underlying data.

A *value receiver* operates on a copy of the value. Any modifications made inside the method do not affect the original value. Use value receivers when your method does not need to modify the receiver or when you're working with small, immutable types.

A *pointer receiver*, on the other hand, operates on the original value via its memory address. Changes made inside the method are reflected in the original value. Use pointer receivers when your method needs to modify the receiver's state or when you're working with large structs to avoid unnecessary copying.

Let's look at some concrete examples to illustrate the difference. Suppose we want to add a method to our `ImperialWeight` type that converts it to a metric `Measurement`. Because this operation does not modify the original value, a value receiver is appropriate:

```
func (i ImperialWeight) ToMetric() Measurement {
        // 1 ounce ≈ 28.3495 grams
        return Measurement{
                Magnitude: i.Magnitude * 28.3495,
                Unit:      Gram,
        }
}
```

Here, `ToMetric` uses a value receiver (`i ImperialWeight`), so calling this method does not change the original `ImperialWeight` value. Instead, it returns a new `MetricWeight` value.

Now consider a method that modifies the receiver, such as scaling a recipe. In this case, we use a pointer receiver so that the changes are reflected in the original value:

```
// Scale multiplies the ingredient amounts by a factor.
func (r *Recipe) Scale(factor uint) {
    for i := range r.Steps {
        for j := range r.Steps[i].Ingredients { #1
            r.Steps[i].Ingredients[j].Measurement = scaleMeasuremen
t(r.Steps[i].Ingredients[j].Measurement, float32(factor)) #2
        }
    }
    r.Yield *= factor
}

func (r *Recipe) AddStep(step Step) {
    r.Steps = append(r.Steps, step)
}

func scaleMeasurement(m Convertible, factor float32) Convertible {
        switch v := m.(type) {
        case MetricWeight:
                return MetricWeight{Measurement{v.Magnitude * Magni
tude(factor), v.Unit}}
        case ImperialWeight:
                return ImperialWeight{Measurement{v.Magnitude * Mag
nitude(factor), v.Unit}}
        default:
                return m
        }
}
```

#1 We need to access array elements directly to modify their values; otherwise, Go will just copy them.
#2 Here we manipulate the internal value of the ingredient measurement.

Here, `Scale` uses a pointer receiver (`r *Recipe`), allowing the method to modify the original `Recipe` value in place. With this method, you can easily scale a recipe up or down by any factor, and the changes are reflected in the original `Recipe` value.

In summary, you should use value receivers when your method does not need to modify the receiver, such as for conversions or formatting, and use pointer receivers when

your method needs to modify the receiver's state or when working with large structs. Choosing between value and pointer receivers is an important design decision that affects how your methods interact with your data. Go's method system allows you to add behavior directly to your types, making your code more modular, expressive, and easier to maintain.

You've now seen how to attach methods to your own types, giving them behavior alongside their data. Methods alone will tell you what a specific type can do, but they don't give you a way to talk about a capability in the abstract, independent of which concrete type provides it. In many cases, it's just as important to categorize things by what they can do as by what they are. This is where *interfaces* come in.

5.5 Interfaces

An *interface* in Go defines a set of method signatures (behavior) that a type must implement, rather than specifying the data it contains. You can think of an interface as a contract: any type that provides the required methods "satisfies" the interface, regardless of its underlying structure. This enables powerful abstractions and flexible code because you can write functions and types that operate on any value that implements the desired behavior without knowing the details of how it works.

A helpful analogy is to think of a wall outlet: the outlet provides a standard way to access power, and any device with the right plug shape can use it, regardless of how the device is built internally. Similarly, an interface provides a standard way to interact with values, as long as they implement the required methods. The implementation

details are hidden, allowing you to focus on what the value can do, not how it does it.

Interfaces are used extensively throughout Go's standard library and are a key part of idiomatic Go code. In fact, you've already used interfaces, perhaps without realizing it. For example, the `fmt.Printf` function can use the `Stringer` interface, which is defined as follows:

```
type Stringer interface {
    String() string
}
```

If your type implements a `String()` method, `fmt.Printf` and similar functions will automatically use the value returned by that method when formatting your type as a string. This lets you control how your types are displayed, making your output more readable and meaningful.

Go has a naming convention for single-method interfaces: the interface name is formed by appending the suffix `-er` to the method name. An interface with a `String()` method is called `Stringer`; one with a `Read()` method is called `Reader`; one with a `Write()` method is called `Writer`. When you design your own single-method interfaces, following this convention makes your code more recognizable and idiomatic to other Go developers.

Go's interface system uses *structural typing*: a type is considered to implement an interface if it provides all the required methods; no explicit declaration is needed. This is sometimes informally called "duck typing," but there is an important distinction: unlike dynamic languages such as Python or Ruby, where missing methods are discovered only at runtime, Go verifies interface satisfaction at *compile*

time. If a type does not implement all of an interface's required methods, your program will not compile.

To implement an interface, a type must provide the methods defined by that interface. Let's see how this works in practice. Suppose we want to display units using their common abbreviations (such as "g" for grams or "oz" for ounces) instead of their full names. We can add a `String()` method to our `Unit` type:

```
func (u Unit) String() string { #1
    switch u {
    case Gram:
        return "g"
    case KiloGram:
        return "kg"
    case Ounce:
        return "oz"
    case Pound:
        return "lb"
    case Liter:
        return "L"
    case Cup:
        return "cup"
    case Pint:
        return "pt"
    case Quart:
        return "qt"
    case Gallon:
        return "gal"
    }
    return string(u)
}
```

#1 Implementation of the interface

Now, whenever we print a `Unit` value using `fmt.Printf`, Go automatically uses our custom string representation. This makes our output clearer and more user-friendly.

Beyond customizing output, interfaces allow you to define your own abstractions. For example, in our recipe application, we want to convert measurements between metric and imperial systems. We can define a `Convertible` interface that requires two methods: `ToImperial()` and `ToMetric()`. Any type that implements these methods can be used wherever a `Convertible` is expected.

```
// --- Convertible Interface ---

type Convertible interface {
    ToImperial() Measurement
    ToMetric() Measurement
}
```

By defining and using interfaces, you can write code that is more flexible, reusable, and easier to test. Interfaces enable polymorphism: the ability for different types to be treated the same way based on shared behavior, without requiring inheritance or complex hierarchies. This is a cornerstone of Go’s design philosophy and a key reason Go code is often simple, composable, and robust.

To implement these interfaces, we will create value receivers for the extended types we wrote in the previous section. This is because we are returning a new `Measurement` struct.

```
func (m MetricWeight) ToImperial() Measurement {
	// 1 gram is about 0.035274 ounces
	return Measurement{
		Magnitude: m.Magnitude * 0.035274,
		Unit:      Ounce,
	}
}
func (m MetricWeight) ToMetric() Measurement {
	return m.Measurement
}

func (i ImperialWeight) ToMetric() Measurement {
	// 1 ounce ≈ 28.3495 grams
	return Measurement{
		Magnitude: i.Magnitude * 28.3495,
		Unit:      Gram,
	}
}
func (i ImperialWeight) ToImperial() Measurement {
	return i.Measurement
}

func (m MetricVolume) ToImperial() Measurement {
	// 1 liter ≈ 2.11338 pints
	return Measurement{
		Magnitude: m.Magnitude * 2.11338,
		Unit:      Pint,
	}
}
func (m MetricVolume) ToMetric() Measurement {
	return m.Measurement
}
func (i ImperialVolume) ToMetric() Measurement {
	// 1 pint ≈ 0.473176 liters
	return Measurement{
		Magnitude: i.Magnitude * 0.473176,
		Unit:      Liter,
	}
}
func (i ImperialVolume) ToImperial() Measurement {
	return i.Measurement
}
```

The preceding code defines methods for converting between metric and imperial units for both weight and volume types. Each specialized type (such as `MetricWeight` or `ImperialVolume`) implements the `Convertible` interface by providing `ToImperial` and `ToMetric` methods. These methods use appropriate conversion factors to calculate the new magnitude and set the correct unit. When converting to the same system, the method simply returns the original `Measurement` value.

This approach demonstrates how Go's interfaces enable polymorphism: any type that implements the `Convertible` interface can be used wherever a `Convertible` is expected, regardless of its underlying implementation. This makes your code more flexible and extensible. For example, you could add new measurement types (such as temperature or length) by defining new structs and implementing the `Convertible` interface for them.

When designing interfaces, it's important to strike a balance between specificity and flexibility. A well-designed interface should capture the essential behavior needed by your code, without being overly restrictive or too generic. In this example, the `Convertible` interface is focused and clear, making it easy to extend your application with new types and conversion logic as needed.

With `Convertible` now fully defined and implemented, the `CreateGroceryList` function from the previous section will compile cleanly. Each `IngredientMeasurement.Measurement` value satisfies `Convertible`, so calling `.ToMetric()` normalizes every ingredient to a common unit before the quantities are summed.

5.6 Checking types

Now that we have an abstracted type in an interface, we will sometimes need to determine the concrete type that implements the interface at runtime. This is especially useful when you want to perform type-specific logic or extract additional information that is available only on certain types.

Go provides two main mechanisms for determining the underlying type of an interface value:

- *Type assertion*—This allows you to check whether an interface value holds a specific concrete type and to extract the value if it does. The syntax is `value, ok := iface.(ConcreteType)`. If the assertion is successful, `ok` is `true` and `value` holds the concrete value; otherwise, `ok` is `false`.
- *Type switch*—This special form of switch statement lets you compare the dynamic type of an interface value against multiple types in a concise way. This is particularly useful when you want to handle several possible types differently.

For example, suppose you have a `Convertible` interface, and you want to determine whether a given value is a metric or imperial measurement. You can use a type switch to inspect the underlying type and return the appropriate `MeasurementSystem`:

```
func MeasurementSystemFromConvertible(m Convertible) MeasurementSys
tem {
    switch m.(type) {#1
    case MetricWeight, MetricVolume:
        return Metric
    case ImperialWeight, ImperialVolume:
        return Imperial
    default:
        // Optionally handle unknown types
        return ""
    }
}
```

#1 Here we take the interface and determine the underlying type.

In this function, the `switch m.(type)` statement checks the dynamic type of `m`. If it matches `MetricWeight` or `MetricVolume`, we return `Metric`; if it matches `ImperialWeight` or `ImperialVolume`, we return `Imperial`. The `default` case can handle unexpected types or return a zero value.

You can also use a type assertion if you only care about one specific type:

```
if mw, ok := m.(MetricWeight); ok { #1
    // m is a MetricWeight, mw is the concrete value
    fmt.Println("MetricWeight magnitude:", mw.Magnitude)
}
```

#1 In line type checking allows you to verify the type before proceeding with logic.

Type switches and assertions are powerful tools for working with interfaces in Go. They let you write flexible, generic code while still being able to access type-specific behavior when needed. This pattern is common in Go programs that use interfaces for abstraction but occasionally need to “peek behind the curtain” to access concrete details.

In summary, to determine the underlying type of an interface value in Go, use type assertions for single-type checks or type switches for handling multiple types. This enables you to write code that is both abstract and type-safe, using Go's strong type system and interface-based design.

5.6.1 Type composition and wrapping

We can use composition to wrap measurements and provide additional behavior. In Go, composition is a core idiom that allows us to build complex types by embedding or including other types, rather than relying solely on inheritance. For example, when converting a recipe, we might define a new type that embeds an existing measurement type and adds methods for unit conversion or formatting. This approach uses Go's type system, enabling us to extend or modify behavior without altering the original type. By composing types in this way, we create flexible and reusable code that fits naturally with Go's emphasis on simplicity.

```
func wrapMeasurement(m Measurement, system MeasurementSystem) Conve
rtible { #1
    switch system {
    case Imperial:
        return ImperialWeight{m}
    case Metric:
        return MetricWeight{m}
    default:
        return MetricWeight{m}
    }
}
```

#1 Returns the correct Convertible type

5.6.2 Example: Putting it all together

Throughout this chapter, we've incrementally built a flexible and extensible recipe application using Go's type system. Starting with simple named types to represent measurements and units, we introduced structs to group related data, embedded types to model domain-specific concepts such as metric and imperial measurements, and collection types to manage lists of ingredients and steps. By defining interfaces and attaching methods, we enabled behaviors such as scaling recipes and converting between measurement systems.

Here's how you might use these types and methods to create, scale, convert, and summarize a recipe.

```
// Example usage in main.
func main() {
    recipe := Recipe{
        Title: "Pancakes",
        Yield: 2,
    }
    recipe.AddStep(Step{ #1
        Description: "Mix dry ingredients",
        Ingredients: []IngredientMeasurement{
            {"Flour", MetricWeight{Measurement{Magnitude: 200, Uni
t: Gram}}},
            {"Sugar", MetricWeight{Measurement{Magnitude: 50, Unit:
Gram}}},
        },
    })
    recipe.AddStep(Step{ #2
        Description: "Add wet ingredients",
        Ingredients: []IngredientMeasurement{
            {"Milk", MetricWeight{Measurement{Magnitude: 300, Unit:
Gram}}},
            {"Egg", MetricWeight{Measurement{Magnitude: 50, Unit: G
ram}}},
        },
    })

    fmt.Println("Original Recipe:", recipe) #3
    recipe.Scale(2) #4
    fmt.Println("Scaled Recipe:", recipe) #5

    list := CreateGroceryList(RecipeBox{recipe}) #6
    fmt.Println("Grocery List:") #7
    for ing, meas := range list {
        fmt.Printf("%s: %.2f %s\n", ing, meas.Magnitude, meas.Unit)
#8
    }
}
```

#1 Add the first step with dry ingredients.
#2 Add the second step with wet ingredients.
#3 Print the original recipe.
#4 Scale the recipe by a factor of 2.
#5 Print the scaled recipe.
#6 Generate a grocery list from the scaled recipe.

#7 Print the grocery list header.
#8 Print each ingredient and its total quantity.

This chapter has highlighted the benefits of Go's emphasis on type safety and composition over inheritance. By using named types and struct composition, you can create clear, idiomatic code that accurately models your problem domain. Interfaces provide powerful abstraction and polymorphism, allowing different types to share common behaviors. Altogether, Go's type system helps you write robust programs that are both correct and easy to extend.

Summary

- Go's type system lets you model real-world domains using named types, structs, and composition for safer, more expressive code.
- Named types and structs clarify intent, group related data, and enable attaching methods for behavior.
- Type embedding (composition) extends types and shares fields and methods without inheritance.
- Collection types (slices and maps) organize groups of values and support abstractions like ingredient lists and grocery lists.
- Pointers allow efficient sharing and modification of data, especially for methods that update receiver state.
- Methods and interfaces add behavior and define contracts, enabling polymorphism and flexible code.
- Type assertions and switches let you check and handle concrete types at runtime when working with interfaces.
- Composing and wrapping types enable you to extend functionality and add domain-specific logic.

6 Generics

This chapter covers

- How to use generics in Go
- Creating generics with interfaces and structs
- Defining type parameters for generics
- Creating functions with generics
- Using generics with the standard library

Computer science is built on various layers of abstraction. Electrical pulses lead to binary code, which leads to assembly code, and so on. In the last chapter, we looked at abstractions in the form of *types*, but in this section, we will explore another layer of abstraction on top of that, known as *generics*.

Generics allow you to define functions, types, and interfaces with type parameters, enabling the same code to operate on different data types while maintaining strict compile-time type safety. Introduced in Go 1.18, generics let you write reusable algorithms and data structures without duplicating code for each specific type or sacrificing compile-time safety.

In this chapter, we will explore the core concepts and practical applications of generics in Go by examining a set of generic collection types and utility functions (lists, sets, and maps), starting with how type parameters work and working up to increasingly powerful patterns.

6.1 Typed parameters in generics

To see why generics matter, consider the recipe application we built in the previous chapter. Suppose you want a function that filters a slice down to only the elements that match some condition. Without generics, you would need separate functions for every type, as illustrated in the following code:

```
// Without generics -- the logic is identical, only the type differ
s.
func FilterRecipes(items []Recipe, keep func(Recipe) bool) []Recipe
{
    var result []Recipe
    for _, item := range items {
        if keep(item) {
            result = append(result, item)
        }
    }
    return result
}

func FilterSteps(items []Step, keep func(Step) bool) []Step {
    var result []Step
    for _, item := range items {
        if keep(item) {
            result = append(result, item)
        }
    }
    return result
}
```

The body of the two preceding functions is identical. The only difference is their types. Before generics, this kind of duplication was unavoidable in Go. Generics in Go are represented by square brackets with a variable (in most cases `T`) with an assigned type (`any`, `int`, etc).

With generics, both of the preceding functions can collapse into one:

```
func Filter[T any](items []T, keep func(T) bool) []T {
        var result []T
    for _, item := range items {
        if keep(item) {
            result = append(result, item)
        }
    }
    return result
}
```

This single `Filter` function works for recipes, steps, ingredients, or any other type. The type parameter `T` is inferred by the compiler from the arguments you pass, so callers can write natural-looking code:

```
metricRecipes := Filter(recipes, func(r Recipe) bool {
    return MeasurementSystemFromConvertible(r) == Metric
})
```

Generics allow developers to write functions, types, and data structures that can operate on different types while maintaining type safety. Before generics, Go developers often used the `interface{}` type to achieve flexibility, but this approach sacrificed compile-time safety and required type assertions. With the introduction of generics, the `any` type can be used as a type constraint, making it easier and safer to write reusable code.


```
metricRecipes := Filter(recipes, func(r Recipe) bool {
    return MeasurementSystemFromConvertible(r) == Metric
})
```


For example, consider a generic `List`. Although Go’s built-in slices are flexible and efficient for storing sequential data, a custom generic `List` can provide additional behaviors, such as efficient insertion or removal at arbitrary positions, or the enforcement of specific type constraints. Such generic data structures are especially useful when you need more

advanced operations or want to ensure type safety across your codebase. Figure 6.1 demonstrates how generics act as containers that provide functionality without requiring knowledge of the underlying type.

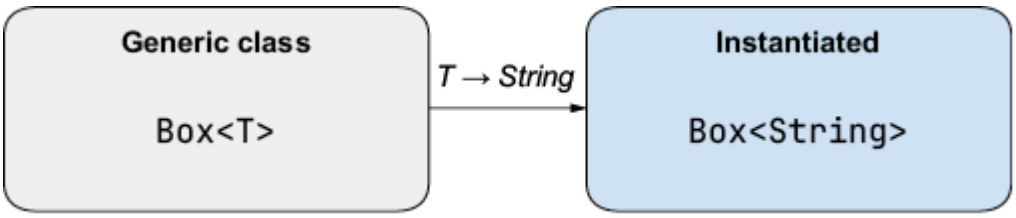


Figure 6.1 Generic structure that eventually holds a string type

With generics, you can write code that is both reusable and type-safe. In the following sections, we will build up a linked list to demonstrate generic syntax in depth. A linked list is a data structure in which each element (called a node) stores a value and a pointer to the next element in the chain. Unlike a Go slice, elements are not stored contiguously in memory, which makes insertion and removal at arbitrary positions efficient because other elements do not need to be shifted.

You would rarely implement your own linked list in production Go code, since slices handle most real use cases more simply, but the linked list is an ideal teaching vehicle for generics because it requires type parameters at multiple levels: the node, the list struct, and every method on the list. When building data structures such as a list, the individual elements are typically represented as nodes.

In Go, you can define a generic node by specifying a type parameter using square brackets after the type name, such as `node[T any]`. Here, `T` is the type parameter, and `any` is the constraint that allows any type. Type parameters can also use interfaces or other types as constraints, which lets you restrict the types that can be used. This approach lets you

use all the types and interfaces discussed in the previous chapter as part of your generic definitions. Let's see how this works in practice.

Listing 6.1 A generic node implementation

```
type node[T any] struct { #1
        value T #2
        next  *node[T] #3
}
```

#1 Notating the node struct as a generic
#2 To use the type parameter just use T within the struct or function.
#3 The type parameter can be used to reference other generic types.

In addition to defining generic types, you can define interfaces that use generics. This powerful feature allows you to specify contracts for collections or other abstractions that work with any type while still maintaining type safety. Throughout this chapter, we will build various collections to demonstrate the power and flexibility of generics in Go. Let's start by defining a generic `Collection` interface.

Listing 6.2 A generic collection interface

```
type Collection[T any] interface { #1
    Add(item T) #2
    Remove(item T) bool
    Contains(item T) bool
    Size() int #3
    Clear()
}
```

#1 The interface uses a type parameter T to make it generic.
#2 Methods use T so the collection works with any type.
#3 Methods like Size() return a fixed type, not using T.

This interface provides a blueprint for any collection type, such as a list, set, or map, that can store elements of any type. Using a type parameter ensures that all operations are

type-safe and consistent for the specific type chosen when the collection is instantiated.

Figure 6.2 illustrates how interfaces interact with generics in Go, showing the flow of type parameters from interface definitions to concrete implementations.

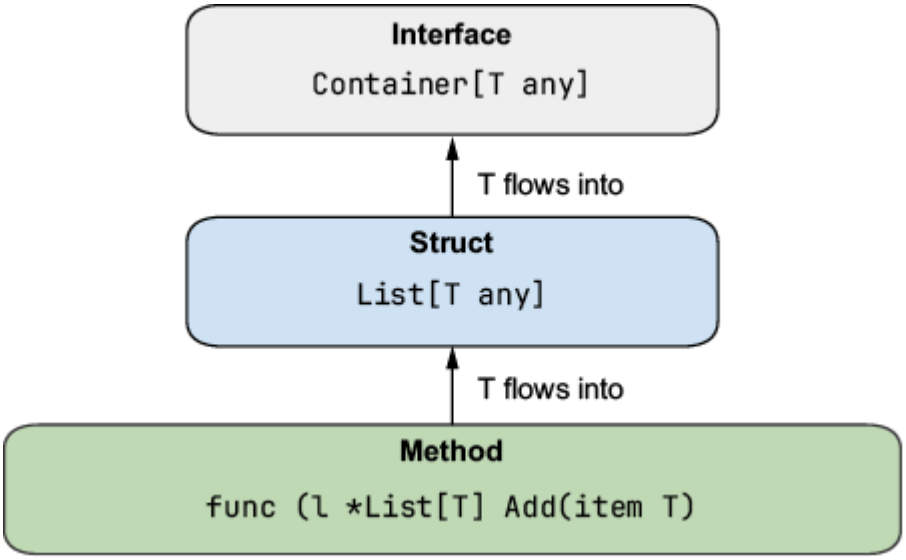


Figure 6.2 Type parameters in an interface are passed to implementing types and methods, ensuring that generic constraints and behaviors are enforced in the implementation.

Go's generics system also allows you to use interfaces as type constraints for type parameters. These constraints can be simple, like `any` (which is equivalent to the empty interface `interface{}`), or more restrictive, such as `comparable`. The `comparable` constraint is particularly useful for types that need to support equality operations, such as when you're implementing sets or maps. It ensures that the type supports the `==` and `!=` operators, which are required for many collection operations.

For example, if you want to create a linked list that only accepts types that can be compared for equality, you can use the `comparable` constraint.

Listing 6.3 A comparable `LinkedList`

```
type LinkedList[T comparable] struct { #1
    head *node[T]
    size int
}
```

#1 The type parameter T is constrained to types that support comparison operations.

This approach allows you to use Go's type system to enforce correctness at compile time, reducing the risk of runtime errors and making our code more robust. As you can see, generics in Go are not limited to types; they extend to interfaces and constraints, enabling you to build highly reusable, type-safe abstractions. This flexibility allows you to tailor your data structures and algorithms to a wide range of requirements while maintaining the safety and clarity that Go is known for.

Let's look at how you can use different kinds of constraints to further refine your generic types.

6.1.1 Using interface constraints

Suppose you want to create a linked list that only accepts elements that implement the `fmt.Stringer` interface. This is useful when you want to ensure that every element in your list can be converted to a string, perhaps for logging or display purposes. You can express this requirement directly in your type definition.

Listing 6.4 A Stringer interface `LinkedList`

```
type StringLinkedList[T fmt.Stringer] struct { #1
    head *node[T]
    size int
}
```

#1 The type parameter T is constrained to types that implement the fmt.Stringer interface.

With this definition, any attempt to instantiate `StringLinkedList` with a type that does not implement `fmt.Stringer` results in a compile-time error. This ensures that your collection is always used correctly and that you can confidently call the `String()` method on any element.

6.1.2 Defining custom constraints

Go generics also allow you to define constraints by creating custom interfaces. For example, you might want to restrict a collection to numeric types, such as `int` or `float32`. You can define a `Numeric` interface that lists the allowed types.

listing 6.5 A Numeric `LinkedList`

```
type Numeric interface {
    int | float32 #1
}

type NumericLinkedList[T Numeric] struct { #2
    head *node[T]
    size int
}
```

#1 The Numeric interface is a type set that includes int and float32.
#2 The NumericLinkedList type is constrained to types that satisfy the Numeric interface.

This approach is powerful, but it has a limitation: it only matches the exact types listed. If you define a new type,

such as `type MyInt int`, it will not satisfy the `Numeric` constraint, even though its underlying type is `int`.

6.1.3 Using underlying types with tilde (~)

To make your constraints more flexible, Go allows you to use the tilde (`~`) operator in type sets. The tilde tells the compiler to accept any type whose underlying type matches the specified type, not just the exact type itself. This is especially useful when you want your generic code to work with both built-in types and user-defined types that have the same underlying representation.

For example, consider the following custom type.

Listing 6.6 Example of a custom type

```
type MyInt int
```

If you define a constraint without the tilde, as follows, the `Numeric` interface will match only values of type `int` or `float32`, not user-defined types like `MyInt` whose underlying type is `int`.

Listing 6.7 An interface with strict type parameters

```
type Numeric interface {
    int | float32
}
```

To address this limitation, you can use the tilde (`~`) operator to specify that any type whose underlying type matches the specified type should satisfy the constraint.

In the following listing, `int`, `float32`, and any user-defined types whose underlying type is `int` or `float32` (such as `MyInt`) will satisfy the `Numeric` constraint.

Listing 6.8 An interface with flexible type parameters

```
type Numeric interface {
    ~int | ~float32
}
```

This makes your generic types more broadly applicable and reusable.

The tilde operator is particularly powerful when building libraries or frameworks because it allows your generic functions and types to work seamlessly with custom types defined by users, as long as they share the same underlying type. This approach enhances code flexibility without sacrificing type safety.

Figure 6.3 shows how we can extend types by first establishing `Numeric`, which encapsulates floats and integers, and then using that type within a larger `Ordered` type, which adds strings to the type. You could imagine generics using this type to perform both numeric and alphabetic ordering.

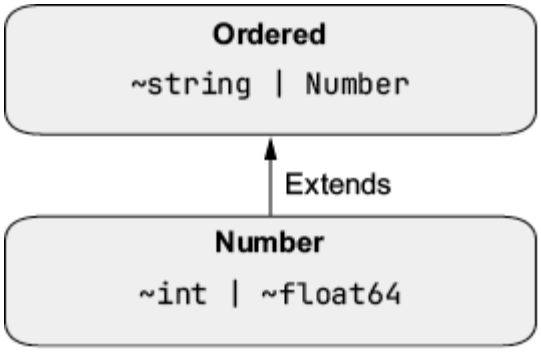


Figure 6.3 Constraint hierarchy for type parameters

The tilde operator in type sets enables you to accept both built-in and user-defined types with the same underlying type, write more generic and reusable code, and maintain strict compile-time type safety. Understanding and using the tilde operator is an important part of mastering Go's

generics system, especially when designing APIs or libraries intended for broad use.

If you would rather be more explicit with your types, you can expand them as in the following example.

Listing 6.9 A numeric `LinkedList` with inline constraint

```
type NumericLinkedList[T ~int | ~float32] struct { #1
    head *node[T]
    size int
}
```

#1 The type parameter T is constrained inline to any type whose underlying type is int or float32.

6.1.4 Summary of type constraints

These examples illustrate how Go's generics system allows you to precisely control which types are permitted in your generic data structures. By using built-in interfaces, custom type sets, and the tilde operator, you can create collections and algorithms that are both flexible and safe.

Understanding and using type parameters and constraints is foundational to effective generic programming in Go. Before we move on to full collection implementations, let's see how these constraints look when applied to a simple, practical function. The `Filter` function from the beginning of section 6.1 uses the `any` constraint because it only needs to store and return elements, not compare them.

```
func Filter[T any](items []T, keep func(T) bool) []T {
    var result []T
    for _, item := range items {
        if keep(item) {
            result = append(result, item)
        }
    }
    return result
}
```

You could use this immediately with the recipe types from chapter 5:

```
// Keep only recipes that serve more than four people.
largeBatch := Filter(recipeBox, func(r Recipe) bool {
    return r.Yield > 4
})
```

The compiler infers `T` as `Recipe` from the slice argument, so no explicit type annotation is needed at the call site. In the following sections, we will build on these concepts to implement practical and efficient generic collections and demonstrate how type constraints can help you enforce invariants and express intent in your code.

6.2 Implementing generic functions

Now that we've established how to define generic structures and interfaces using type parameters, the next step is to put these abstractions into practice by implementing functions that use generics. This allows you to create reusable, type-safe operations that work seamlessly with your generic types.

By using generic functions, you can write code that adapts to a wide variety of types without sacrificing safety or clarity. These functions can accept parameters or return values that are themselves generic, making it possible to

build flexible APIs and utilities that work across different data structures. This approach not only reduces code duplication but also helps ensure that errors are caught at compile time rather than at runtime.

With generic data structures like `LinkedList[T]` or interfaces such as `Collection[T]`, you can write functions that operate on any instance of these types, regardless of the element type. This enables you to handle common tasks, such as creating new instances, adding or removing elements, searching, or transforming data, while maintaining type safety and eliminating code duplication. For example, after defining a generic `LinkedList`, you might implement a generic constructor function that accepts a type parameter and returns a pointer to a new `LinkedList` of that type.

Listing 6.10 A function to create a new generic type

```
func NewLinkedList[T comparable]() *LinkedList[T] {
    return &LinkedList[T]{} #1
}
```

#1 Instantiation of a struct using a generic defined in the function

Here, the `NewLinkedList` function is parameterized with `[T comparable]`, meaning it can create a linked list for any type `T` that supports comparison operations. This pattern is a natural extension of our earlier generic type definitions, allowing us to instantiate and work with collections in a type-safe manner.

Figure 6.4 shows how we move from using a type parameter to create an interface, then proceed through the generic structure to an instantiated value.

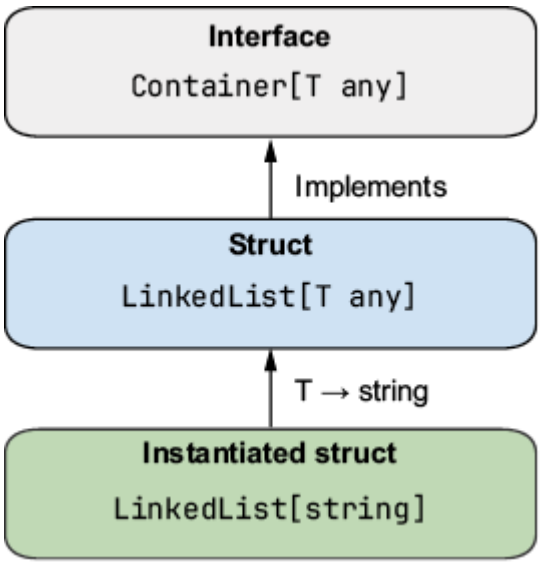


Figure 6.4 A single type parameter flowing from the List interface to an instantiated LinkedList

Generics also allow us to write functions that operate on interfaces with type parameters. Suppose we have a generic `Collection` interface, as defined in listing 6.2. We can now write utility functions that accept any type that implements this interface, regardless of the underlying element type. For instance, consider a function that prints the contents of any collection.

Listing 6.11 `PrintCollection` takes a generic interface that prints contents.

```
// Sample function using Collection
func PrintCollection[T any](c Collection[T]) {                  #1
        fmt.Printf("Collection has %d items:\n", c.Size())        #
2
        if list, ok := c.(List[T]); ok {
#3
                // If the collection is a List, we can access eleme
nts by index
                for i := 0; i < c.Size(); i++ {
#4
                        val, _ := list.Get(i)
#5
                        fmt.Println(val)
                }
        }
}
```

#1 The function is generic, parameterized by type T, and accepts any Collection of T.
#2 Prints the number of items in the collection using the Size() method from the Collection interface
#3 Uses a type assertion to check if the collection also implements the List[T] interface, which provides indexed access
#4 Iterates over the collection by index, assuming the collection supports indexing
#5 Retrieves the element at index i using the List's Get method and prints its value

In this example, `PrintCollection` is a generic function that takes a `Collection[T]` as its argument. The type parameter `T` ensures that the function is type-safe and works with a collection of any element type. Inside the function, we use a type assertion to check whether the collection also implements a `List[T]` interface, allowing us to access elements by index. This demonstrates how generics enable flexible, reusable code that can adapt to different collection types while maintaining strict type safety.

Another common pattern is transforming a collection from one type to another. Suppose you want to extract every

recipe title from a `RecipeBox` to populate a menu. Without generics, you would write a specific conversion function for every pair of types. With two type parameters, you can express the general pattern once.

```
func Transform[T any, U any](items []T, fn func(T) U) []U {
    result := make([]U, len(items))
    for i, item := range items {
        result[i] = fn(item)
    }
    return result
}
```

You can then use it with the recipe domain.

```
titles := Transform(recipeBox, func(r Recipe) string {
    return r.Title
})
```

The compiler infers `T` as `Recipe` and `U` as `string` from the arguments. Two type parameters let you describe a relationship between two distinct types in a single, reusable function, which was not possible without either generics or runtime `interface{}` casts.

6.3 Implementing generic interfaces

In the previous sections, we've explored how to define generic interfaces and types in Go, laying the groundwork for building flexible and reusable abstractions. But defining an interface is only the first step; it's equally important to understand how to implement these generic interfaces in practice. By implementing generic interfaces, we can create concrete data structures that use the power of type parameters, ensuring both type safety and code reusability.

Our next step will be implementing a generic interface. We'll start by defining a `List` interface that uses type parameters and then provide a concrete implementation in the form of a `LinkedList`. This example will demonstrate how to attach type parameters to both interfaces and their implementing structs, and how to fulfill the contract specified by a generic interface.

Listing 6.12 Defining a generic `List` interface

```
type List[T any] interface {
        Collection[T]
        Get(index int) (T, error)
        Set(index int, item T) error
        Insert(index int, item T) error
        RemoveAt(index int) (T, error)
}
```

Notice that our generic `List[T]` interface allows us to work with any type, such as `List[int]` or `List[User]`, and it provides methods like `Insert`, `Get`, and `RemoveAt` that operate on those types. This abstraction lets us manipulate and organize data at a higher level, reducing duplication and increasing flexibility.

To implement this interface, we use the `node[T any]` type introduced earlier, but our concrete implementation, `LinkedList[T comparable]`, uses the `comparable` constraint for its type parameter. Because `comparable` is a subset of `any`, this still satisfies the `List[T any]` interface. This approach ensures that our linked list can store only types that support equality comparison, which is important for operations like `Remove` or `Contains`.

We defined the struct and the function in the previous section. Now we'll add the functions necessary to fulfill the interface.

Listing 6.13 The `Add` function that appends an item to the list

```
func (l *LinkedList[T]) Add(item T) {           #1
        n := &node[T]{value: item}                  #2
        if l.head == nil {                          #3
                l.head = n
        } else {
                curr := l.head
                for curr.next != nil {                  #4
                        curr = curr.next
                }
                curr.next = n
        }
        l.size++                                    #5
}
```

#1 Method to add an item to the generic linked list
#2 Creates a new node with the provided value of type T
#3 If the list is empty, set the new node as the head.
#4 Otherwise, traverse to the end of the list to append the new node
#5 Increment the size of the list after adding the new item.

Here we are creating nodes of unknown types and then using them to store any entity supported by the type parameter. Think of examples from your codebase: patterns or data structures you frequently replicate. With generics, you can abstract these away and avoid duplication.

Let’s finish implementing the `Collection` interface by adding the final methods to our struct.

Listing 6.14 Implementing part of the interface

```
func (l *LinkedList[T]) Remove(item T) bool {          #1
        var prev *node[T]
        curr := l.head
        for curr != nil {
                if curr.value == item {                        #2
                        if prev == nil {
                                l.head = curr.next
#3
                        } else {
                                prev.next = curr.next
#4
                        }
                        l.size--                                  #
5
                        return true
                }
                prev = curr
                curr = curr.next
        }
        return false
}

func (l *LinkedList[T]) Contains(item T) bool {        #6
        curr := l.head
        for curr != nil {
                if curr.value == item {                        #7
                        return true
                }
                curr = curr.next
        }
        return false
}

func (l *LinkedList[T]) Size() int {                   #8
        return l.size
}

func (l *LinkedList[T]) Clear() {                      #9
        l.head = nil
        l.size = 0
}
```

#1 Implements the Remove method to delete the first occurrence of item from the list
#2 Checks if the current node's value matches the item to remove
#3 If the item is at the head, update the head pointer.
#4 If the item is in the middle or end, bypass the current node.
#5 Decrement the size after removal
#6 Implements the Contains method to check if an item exists in the list
#7 Returns true if the item is found in any node
#8 Returns the current number of elements in the list
#9 Clears the list by resetting the head and size

What makes lists special is that we can pull items out at a given location because everything is ordered. This random-access capability is a key distinction between lists and other collection types, such as sets or maps, which do not guarantee order or positional access.

We will implement a `Get` function that retrieves a value at a given location in the list. In the following listing, you will notice that we define a `zero` value early in the function. This is because we cannot fully instantiate the type because we do not know what the type is; we can only declare the variable and rely on the compiler to know the proper empty value of that type.

Listing 6.15 Implementing a `Get` call that relies on an index to retrieve an item

```
func (l *LinkedList[T]) Get(index int) (T, error) {
        var zero T                                          #1
        if index < 0 || index >= l.size {                   #2
                return zero, errors.New("index out of bounds")
        }
        curr := l.head
        for i := 0; i < index; i++ {                        #3
                curr = curr.next
        }
        return curr.value, nil                              #4
}
```

#1 Declare a zero value of type T to return in case of an error.
#2 Check if the index is out of bounds; return an error if so.
#3 Traverse the list to the node at the specified index.
#4 Return the value at the given index and nil error.

Additionally, we can override the value at a particular location by using the `Set` method, which updates the node at the specified index. The `Insert` method allows us to add a new element at any position in the list, shifting subsequent elements forward. The `RemoveAt` method removes the element at a given index, reconnecting the surrounding nodes to maintain the list's integrity. These operations demonstrate how a linked list provides flexible manipulation of elements by position, enabling efficient insertion, removal, and updating of values at arbitrary locations within the collection.

Listing 6.16 Finishing the interface implementation

```
func (l *LinkedList[T]) Set(index int, item T) error {
        if index < 0 || index >= l.size {                #1
                return errors.New("index out of bounds")
        }
        curr := l.head
        for i := 0; i < index; i++ {                     #2
                curr = curr.next
        }
        curr.value = item                                #3
        return nil
}

func (l *LinkedList[T]) Insert(index int, item T) error {
        if index < 0 || index > l.size {                 #4
                return errors.New("index out of bounds")
        }
        n := &node[T]{value: item}                       #5
        if index == 0 {                                  #6
                n.next = l.head
                l.head = n
        } else {
                curr := l.head
                for i := 0; i < index-1; i++ {               #7
                        curr = curr.next
                }
                n.next = curr.next                           #8
                curr.next = n
        }
        l.size++                                         #9
        return nil
}

func (l *LinkedList[T]) RemoveAt(index int) (T, error) {
        var zero T
        if index < 0 || index >= l.size {                #10
                return zero, errors.New("index out of bounds")
        }
        var prev *node[T]
        curr := l.head
        for i := 0; i < index; i++ {                     #11
                prev = curr
```

```
                curr = curr.next
        }
        if prev == nil {                                    #12
                l.head = curr.next
        } else {
                prev.next = curr.next                           #13
        }
        l.size--                                            #14
        return curr.value, nil
}
```

#1 Checks if the index is out of bounds for setting a value
#2 Traverses the list to the node at the specified index
#3 Updates the value at the given index with the new item
#4 Checks if the index is valid for insertion (can insert at size for append)
#5 Creates a new node with the provided value of type T
#6 Handles insertion at the head of the list
#7 Traverses to the node just before the insertion point
#8 Inserts the new node at the specified index
#9 Increments the size of the list after insertion
#10 Checks if the index is out of bounds for removal
#11 Traverses to the node at the specified index, keeping track of the previous node
#12 Handles removal at the head of the list
#13 Bypasses the node to be removed by updating the previous node's next pointer
#14 Decrements the size of the list after removal

Now that we've implemented a complete generic structure with its associated methods and utility functions, let's see how it works in practice. The following `main` function demonstrates how to use our new `LinkedList` type.

Listing 6.17 Using our `LinkedList` type

```
func main() {

        ll := NewLinkedList[int]()      #1
        ll.Add(10)                      #2
        ll.Add(20)
        ll.Add(30)
        ll.Insert(1, 15)                #3
        PrintCollection(ll)             #4
        val, _ := ll.Get(2)             #5
        fmt.Println(val)
        ll.Set(2, 25)                   #6
        PrintCollection(ll)
        removed, _ := ll.RemoveAt(1)    #7
        fmt.Println(removed)
        PrintCollection(ll)

}
```

#1 Create a new generic linked list for integers.
#2 Add elements to the list using the Add method.
#3 Insert the value 15 at index 1.
#4 Print the contents of the list.
#5 Retrieve the value at index 2.
#6 Set the value at index 2 to 25.
#7 Remove the element at index 1 and store the removed value.

Now let's write a function that operates on this generic type by using the interface methods to reverse the list. By using the generic `List[T]` interface, we can create a reusable function that works with any list implementation, regardless of the underlying element type. This approach demonstrates the power and flexibility of generics in Go, allowing us to write algorithms that are both type-safe and broadly applicable.

The following listing shows a `ReverseList` function that takes a generic list and reverses its elements in place.

Listing 6.18 Function for reversing a generic list using type parameters

```
// Sample function using List
func ReverseList[T comparable](l List[T]) {           #1
        n := l.Size()                                     #2
        for i := 0; i < n/2; i++ {                        #3
                a, _ := l.Get(i)                              #4
                b, _ := l.Get(n - 1 - i)                      #5
                l.Set(i, b)                                   #6
                l.Set(n-1-i, a)                               #7
        }
}
```

#1 The function is generic, parameterized by type T constrained to comparable, and accepts any List of T.
#2 Retrieves the size of the list to determine the number of elements
#3 Iterates over the first half of the list to swap elements from both ends
#4 Gets the element at the current index i
#5 Gets the element at the mirrored index from the end (n-1-i)
#6 Sets the element at index i to the value from the mirrored index
#7 Sets the element at the mirrored index to the value from index i, effectively swapping the two

This `ReverseList` function iterates over the first half of the list, swapping each element with its counterpart from the other end. The use of type parameters ensures that the function is type-safe and that it works with any type that satisfies the `comparable` constraint.

Now we can use `ReverseList` in our `main` function.

Listing 6.19 Example of using `ReverseList`

```
func main() {
    ...
    ReverseList(ll)
    PrintCollection(ll)
}
```

This example highlights how generics enable you to write flexible, reusable code that can operate on a wide variety of

types and data structures while maintaining strict compile time type safety. The same `LinkedList` type works equally well with domain types. For example, `NewLinkedList[Recipe]()` produces a list that holds only `Recipe` values, with every method fully type-checked at compile time.

Throughout this section, we've seen how a single type parameter allows us to build generic collections and utility functions that are both expressive and robust. By using type parameters and constraints, we've eliminated much of the boilerplate and risk of runtime errors that previously accompanied type-agnostic code in Go.

Many real-world scenarios require working with more than one type at a time. For example, data structures like maps or dictionaries associate keys with values, each of which may have a different type. To address these use cases, Go's generics system supports multiple type parameters, allowing you to define abstractions that relate several types in a type-safe manner. Let's explore how we can use multiple type parameters to build even more powerful and flexible abstractions.

6.4 Multiple type parameters

Go's generics system allows you to define types, interfaces, and functions that operate on one or more type parameters. This feature is especially powerful when you need to relate multiple types in a single abstraction. For example, a map data structure associates keys of one type with values of another type. In Go, you can express this relationship by specifying multiple type parameters, separated by commas, in your generic type or interface definitions.

Multiple type parameters are common in generic programming, especially for data structures such as maps,

dictionaries, or pairs, where two or more types are inherently linked. A hash map (or dictionary) stores values indexed by unique keys, and looking up a value by key is very fast regardless of the map size. We will be building a generic `HashMap` type that wraps Go's built-in map type to demonstrate how multiple type parameters work in practice. The same pattern appears in real codebases when you need a map with additional behavior, such as ordered iteration or concurrency safety.

In Go, you declare multiple type parameters in the same square brackets, such as `[K comparable, V any]`, where `K` is the type of the key (constrained to types that can be compared for equality) and `V` is the type of the value (which can be any type). Figure 6.5 shows how we can define an interface, create a generic struct, and instantiate values for multiple type parameters.

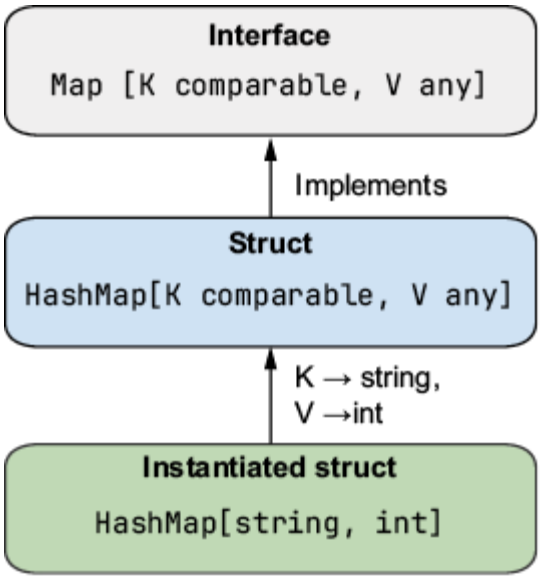


Figure 6.5 Two type parameters flowing from the Map interface to an instantiated HashMap

Let's see how this works in practice by defining a generic `Map` interface that uses two type parameters: one for the key and one for the value. This approach enables you to create

reusable, type-safe abstractions for mapping relationships, similar to Go's built-in `map` type, but with the flexibility to define your own behaviors and constraints. It will also require our keys to be comparable.

```
type Map[K comparable, V any] interface {           .#1
        Put(key K, value V)                             #2
        Get(key K) (V, bool)                            #3
        Remove(key K) bool                              #4
        ContainsKey(key K) bool                         #5
        Size() int                                      #6
        Clear()                                         #7
        Keys() []K                                      #8
        Values() []V                                    #9
}
```

#1 The interface is parameterized with two type parameters: K (the key, must be comparable) and V (the value, can be any type).
#2 Put inserts or updates a value for the given key.
#3 Get retrieves the value for a key, returning the value and a boolean indicating if the key exists
#4 Remove deletes the entry for the given key, returning true if the key was present.
#5 ContainsKey checks if the map contains the specified key.
#6 Size returns the number of key-value pairs in the map.
#7 Clear removes all entries from the map.
#8 Keys returns a slice of all keys in the map.
#9 Values returns a slice of all values in the map.

As you can see, the keys and values are separate types. Let's implement this by creating a `HashMap` struct. The `HashMap` will be a concrete implementation of the `Map[K, V]` interface, using Go's built-in `map` type under the hood. By parameterizing both the key and value types, we ensure that our map is type-safe and flexible, supporting any combination of comparable keys and value types.

It is important to note that the values returned by the map will not be deterministic because we are using the underlying map type for our implementation. Still, you could

attempt to create more advanced structures, such as an `OrderedMap`, using the same `Map` interface we defined,

Here's how you can define the `HashMap` struct and its constructor, which is similar to the previous `LinkedList` example.

```
type HashMap[K comparable, V any] struct {      #1
        m map[K]V                                  #2
}

func NewHashMap[K comparable, V any]() *HashMap[K, V] {   #3
        return &HashMap[K, V]{m: make(map[K]V)}               #4
}
```

#1 The struct is parameterized with two type parameters: K (the key, must be comparable) and V (the value, can be any type).
#2 The underlying storage uses Go's built-in map type with key type K and value type V.
#3 The constructor function is generic, creating a new HashMap for any key and value types.
#4 Initializes the internal map with the specified key and value types

Now we will implement the methods required for our `HashMap` type to fulfill the `Map[K, V]` interface. These methods add a key/value pair (`Put`), retrieve a value by key (`Get`), remove a key/value pair (`Remove`), and check for the existence of a key (`ContainsKey`). We'll also provide utility methods to get the number of entries (`Size`), clear the map (`Clear`), and retrieve all keys or values as slices (`Keys` and `Values`). Each method uses Go's built-in map type for efficient storage and lookup while maintaining type safety through generics. The following code shows these implementations in detail.

```
func (h *HashMap[K, V]) Put(key K, value V) {
        h.m[key] = value                                    #1
}

func (h *HashMap[K, V]) Get(key K) (V, bool) {
        val, ok := h.m[key]                                 #2
        return val, ok
}

func (h *HashMap[K, V]) Remove(key K) bool {
        if _, ok := h.m[key]; ok {                          #3
                delete(h.m, key)                                #4
                return true
        }
        return false
}

func (h *HashMap[K, V]) ContainsKey(key K) bool {
        _, ok := h.m[key]                                   #5
        return ok
}

func (h *HashMap[K, V]) Size() int {
        return len(h.m)                                     #6
}

func (h *HashMap[K, V]) Clear() {
        h.m = make(map[K]V)                                 #7
}

func (h *HashMap[K, V]) Keys() []K {
        keys := make([]K, 0, len(h.m))                      #8
        for k := range h.m {
                keys = append(keys, k)                          #9
        }
        return keys
}

func (h *HashMap[K, V]) Values() []V {
        values := make([]V, 0, len(h.m))                    #10
        for _, v := range h.m {
                values = append(values, v)                      #11
        }
```

```
        return values
}
```

#1 Inserts or updates the value for the given key in the map
#2 Retrieves the value for the given key and a boolean indicating if the key exists
#3 Checks if the key exists in the map before attempting to delete.
#4 Deletes the key-value pair from the map if present.
#5 Checks if the key exists in the map and returns the result
#6 Returns the number of key-value pairs in the map
#7 Clears the map by reinitializing the underlying map storage
#8 Creates a slice to hold all keys with an initial capacity equal to the map size
#9 Appends each key in the map to the keys slice
#10 Creates a slice to hold all values with an initial capacity equal to the map size
#11 Appends each value in the map to the values slice

Let’s play around with this map now in our main function. In the following code, we’ll create a map that associates specific keys with corresponding values.

```
func main() {
        ...
        mapA := NewHashMap[string, int]()         #1
        mapA.Put("one", 1)                        #2
        mapA.Put("two", 2)
        for _, k := range mapA.Keys() {           #3
                        fmt.Printf("%v", k)                #4
        }
        for _, v := range mapA.Values() {         #5
                        fmt.Printf("%v", v)                #6
        }
}
```

#1 Create a new generic hash map with string keys and int values.
#2 Add key-value pairs to the map using the Put method.
#3 Iterate over all keys in the map using the Keys method.
#4 Print each key.
#5 Iterate over all values in the map using the Values method.
#6 Print each value.

Notice how the generic HashMap type defined here uses multiple type parameters—one for the key and one for the

value—allowing us to specify the types for both independently. This demonstrates the power of generics with multiple type parameters in Go: you can create flexible, type-safe data structures that relate different types, such as mapping strings to integers or any other combination you need. This approach enables efficient storage, retrieval, and updating of data based on unique identifiers, all while maintaining strict type safety.

Now that you've seen how to define and use generic map types, let's take it a step further by writing a function that combines two maps into one. This is a common operation when you want to aggregate data from multiple sources or merge configurations. By using generics, we can create a reusable `MergeMaps` function that works with any map implementation conforming to our `Map[K, V]` interface, regardless of the key or value types.

The following code demonstrates how to implement such a function in a type-safe and flexible way by requiring the key type to be comparable so we can identify conflicting keys while allowing the value type to be anything we want.

```
// Sample function using Map
func MergeMaps[K comparable, V any](a, b Map[K, V]) Map[K, V] {
        result := NewHashMap[K, V]()
        for _, m := range []Map[K, V]{a, b} {
                if hm, ok := m.(*HashMap[K, V]); ok {
                        for k, v := range hm.m {
                                result.Put(k, v)
                        }
                }
        }
        return result
}
```

As you introduce more type parameters or constraints, generic code can appear complex or verbose. But generics

are designed to enable powerful abstractions that promote reusable, type-safe code. Although the syntax may become more intricate, the resulting flexibility and maintainability are often worth the tradeoff. When designing abstractions, it’s helpful to start with concrete implementations and refactor toward generic solutions as patterns emerge. This approach ensures that abstractions remain practical and aligned with real use cases, avoiding unnecessary complexity.

When you’re working with multiple type parameters, it’s important to document your code and use descriptive type parameter names where appropriate. Doing so helps keep your abstractions understandable and maintainable, even as they grow in complexity. Remember, the goal of generics is not to make your code more complicated but to enable you to express general-purpose logic safely and efficiently.

The following code demonstrates how you can create two maps that match a given function with the same type parameters. Notice that you do not need to include the type parameters on the function because the compiler knows the types match from the structs.

```
func main() {
        ...
        mapB := NewHashMap[string, int]()           #1
        mapB.Put("three", 3)                        #2
        mapB.Put("two", 22)                         #3
        merged := MergeMaps(mapA, mapB)             #4
        for _, k := range merged.Keys() {           #5
                        fmt.Printf("%v", k)                 #6
        }
        for _, v := range merged.Values() {         #7
                        fmt.Printf("%v", v)                 #8
        }
}
```

#1 Create a new generic hash map with string keys and int values.
#2 Add the key "three" with value 3 to mapB.
#3 Add the key "two" with value 22 to mapB (overrides "two" if present in mapA).
#4 Merge mapA and mapB into a new merged map.
#5 Iterate over all keys in the merged map.
#6 Print each key (should be fmt.Printf("%v", k)).
#7 Iterate over all values in the merged map.
#8 Print each value (should be fmt.Printf("%v", k), but likely intended to print v).

Working with multiple type parameters lets you express relationships between types that a single parameter cannot capture. The `MergeMaps` function is a clear example: `K` must be `comparable` because map keys require equality checking, while `V` can remain unconstrained. Each parameter carries its own constraint independently, and the compiler verifies all of them at the call site.

Now that we've covered generic types, functions, and multiparameter abstractions, the next step is understanding where generics have limits and when simpler alternatives are the better choice.

6.5 Limitations and considerations

Go went a long time without generics, and as a result, many Go developers are accustomed to writing concrete, type-specific code. Although generics provide powerful abstractions and enable greater code reuse, they also introduce new considerations and tradeoffs that are important to understand before adopting them widely in your projects.

In this section, we'll explore some of the key limitations and considerations when working with generics in Go. We'll discuss how generics can affect performance, highlight certain language restrictions, and point out common pitfalls that developers may encounter. We'll also cover how generics were introduced in a way that preserves Go's strong commitment to backward compatibility and examine the current state of tooling and best practices for using generics effectively.

By understanding these aspects, you'll be better equipped to decide when and how to use generics in your code and how to balance the benefits of abstraction with the simplicity and clarity that Go is known for.

6.5.1 Performance considerations

Generics in Go provide powerful abstractions and code reuse, but they also come with performance characteristics worth understanding before you adopt them in hot paths.

Go's compiler uses a strategy for garbage collection (GC) called *GC-shape stenciling* (sometimes called partial monomorphization). For each distinct "GC shape" of a type argument (broadly, types that share the same memory layout and pointer structure), the compiler generates one specialized version of the generic function. Primitive types like `int`, `int64`, and `float64` each get their own stencil,

whereas pointer types share a single stencil because all pointers have the same memory layout. This is meaningfully different from the "one implementation for all types via interfaces" approach: most numeric generics perform comparably to handwritten concrete code, while pointer-heavy code may incur a small overhead from the shared stencil's use of an internal dictionary to dispatch method calls.

In practice, the performance difference is small for the vast majority of Go code. Consider this generic sum function:

```
func Sum[T int | int64 | float64](s []T) T {
    var total T
    for _, v := range s {
        total += v
    }
    return total
}
```

For `int` and `float64`, the compiler generates specialized code with no interface overhead. A concrete `SumInt(s []int) int` may still be marginally faster in tight benchmarks because the compiler has more context for inlining decisions, but the gap is rarely significant enough to guide design choices. Always profile before optimizing.

Memory allocations are a more common concern than raw CPU speed. If a generic function causes values to escape to the heap by storing them in an interface, garbage collection pressure increases. Keep type constraints as specific as the problem allows: `comparable` or a narrow type set gives the compiler more information than `any`.

When writing efficient generic code, prefer narrow constraints over `any` when operations on the type are needed, avoid unnecessary interface conversions inside hot

loops, and benchmark before and after introducing a generic abstraction to verify that it meets your performance requirements.

6.5.2 Limitations of Go generics

Another limitation is that type parameters cannot be used for constants or struct field names. For example, you cannot declare a constant of a generic type, nor can you use a type parameter to dynamically name a struct field. This restricts some patterns that are possible in other languages with more flexible generics.

Go also restricts the use of operators with generic types. You cannot use arithmetic or comparison operators on a type parameter unless you explicitly constrain it to types that support those operations. For example, to use the `+` operator in a generic function, you must constrain the type parameter to a set of numeric types.

```
type Numeric interface {
    ~int | ~float64 | ~int64
}

func AddAll[T Numeric](a, b T) T {
    return a + b
}
```

Without such a constraint, attempting to use `+` on a generic type parameter results in a compile-time error. This ensures type safety, but it also means you need to be explicit about which operations are allowed on your type parameters.

6.5.3 Common gotchas

When working with generics in Go, developers may encounter several common pitfalls and subtle problems.

INTERFACE CONSTRAINTS AND METHOD SETS

When you use an interface as a type constraint, only the methods in the interface's method set are available on the type parameter. This can be surprising if you expect additional methods to be accessible. For example, if you constrain a type parameter with `fmt.Stringer`, you can only call the `String()` method, even if the underlying type has more methods.

TYPE ASSERTION PITFALLS WITH GENERIC INTERFACES

Type assertions involving generic interfaces can be tricky because the type parameters must match exactly. Asserting a `Collection[int]` to a matching concrete instantiation, such as `*LinkedList[int]`, is checked at runtime and succeeds when the dynamic type matches. Asserting it to an instantiation that could never implement the interface, such as `*LinkedList[string]`, is different: the compiler rejects it outright before the program runs, rather than returning `false` at runtime.

Listing 6.20 Type assertion with generics

```
var c Collection[int] = NewLinkedList[int]()

ll, ok := c.(*LinkedList[int])         #1

// ll2, ok := c.(*LinkedList[string])  #2
```

#1 ok is true here: the dynamic type is *LinkedList[int], so the runtime assertion matches.
#2 This line is commented out because it does not compile. *LinkedList[string] cannot implement Collection[int], so the compiler rejects the assertion instead of deferring to a runtime false.

TYPE PARAMETER SHADOWING AND NAMING CONFUSION

It's easy to accidentally reuse type parameter names in nested scopes, leading to confusion or subtle bugs. Always use clear and distinct names for type parameters, especially when working with multiple generic types or functions. In the following case, because our type parameter is defined as `T`, we would not want to create a variable `T` within the body of the function.

Listing 6.21 Shadowing parameters

```
func Foo[T any](x T) {

        var T int #1
}
```

#1 The T variable would override or hide the outer T type parameter.

UNEXPECTED TYPE INFERENCE OR CONSTRAINT ERRORS

Go's type inference for generics is powerful, but sometimes it can infer types you didn't expect or fail to infer a type at all. This can lead to confusing error messages or require you to specify type parameters explicitly. Because types can be inferred, it is sometimes important to be explicit about how you are using the generic, as in the following examples.

Listing 6.22 Explicit identities

```
func Identity[T any](v T) T { return v }
x := Identity(42) #1
y := Identity[string]("hello") #2
```

#1 Integer type is inferred.
#2 String type is explicit and clear to the reader.

INTERACTIONS WITH REFLECTION AND LOSS OF TYPE INFORMATION

When using reflection with generic types, you may lose information about the specific type parameters at runtime. Go's type system erases type parameters after compilation, so you can't always recover the original type arguments using reflection. This can make certain advanced patterns more difficult or impossible with generics. In the following example, if `v` is an interface value, `%T` prints the dynamic type stored in the interface, not the static type `T`. This behavior can be confusing if you expect the generic type parameter to always match the runtime type.

Listing 6.23 Reflection issues

```
func PrintType[T any](v T) {
        fmt.Printf("Type: %T\n", v) #1
}
```

#1 This formatting option will print the concrete type.

By being aware of these common gotchas, you can avoid subtle bugs and write more robust generic code in Go. When adopting generics, start with small, focused abstractions and gradually expand their use as you gain confidence. Always test your generic code thoroughly, and use Go's type inference and explicit type parameter syntax to clarify intent when needed. As the ecosystem matures, more patterns and best practices will emerge, so stay engaged with the Go community and keep learning from real-world examples.

6.5.4 When to use generics

Generics are a powerful tool, but they are not always the right one. Go's philosophy favors simplicity and readability,

and generics add syntactic complexity that is only justified when the abstraction genuinely earns its keep.

Reach for generics in cases like these:

- You are writing a utility function whose logic is identical across multiple types (`Filter`, `Map`, `Reduce`, `Contains`, and similar collection helpers are the canonical examples).
- You are building a library or shared package that needs to work across codebases with different domain types.
- You need compile-time guarantees about type relationships, such as a cache or result-wrapper that must preserve the exact type of its contents.

Avoid generics in cases like these:

- A plain interface already expresses the abstraction cleanly, especially when behavior matters more than type identity.
- The function or type is called with only one concrete type; the indirection adds complexity without benefit.
- The logic differs meaningfully between types, which signals that separate concrete functions are clearer.
- You are early in a design, and the right abstraction has not emerged yet: concrete code first, generic later.

Here's a useful rule of thumb from the Go team: if you cannot write at least two independent call sites that exercise a generic function with different type arguments, the function probably does not need to be generic.

6.5.5 Backward compatibility

One of the most important aspects of Go's introduction of generics was maintaining backward compatibility. The Go team has always emphasized the Go 1 compatibility promise, which ensures that code written for one version of

Go will continue to work in future versions. Generics were designed and implemented in a way that did not break any existing code, allowing developers to upgrade to Go 1.18 and beyond without fear of regressions.

Generics are fully interoperable with existing nongeneric code. You can use generic types and functions alongside traditional concrete types, and you can gradually introduce generics into your codebase as needed. For example, you might start by refactoring a utility function to use type parameters, while leaving the rest of your code unchanged. This incremental approach makes it easy to adopt generics without requiring a complete rewrite.

6.6 Standard library support

With the introduction of generics, the Go standard library has evolved to include new packages and features that make it easier to write reusable, type-safe code. These additions provide generic utilities for common data structures and algorithms, allowing developers to use generics without reinventing the wheel. In this section, we'll explore how the standard library supports generics, including the new `constraints`, `slices`, `maps`, and `cmp` packages, and you'll see how they enable concise and expressive programming patterns in modern Go.

6.6.1 The constraints package

When generics shipped in Go 1.18, the `golang.org/x/exp/constraints` package provided standard type constraints such as `constraints.Ordered` for use in generic code. That package is still available, but as of Go 1.21, the constraint you'll need most often, the set of all types that support `<`, `⇐`, `>`, and `>=`, was promoted into the standard

library as `cmp.Ordered`. New code targeting Go 1.21 and later should prefer `cmp.Ordered` and can drop the `golang.org/x/exp` dependency entirely.

The `cmp.Ordered` constraint matches all built-in integer, float, and string types, enabling you to write generic functions that only accept types where ordering makes sense.

Listing 6.24 Example: Using cmp.Ordered (Go 1.21+)

```
import "cmp"

// Maximum returns the greater of two ordered values.
func Maximum[T cmp.Ordered](a, b T) T {
    if a > b {          #1
        return a        #2
    }
    return b            #3
}

func main() {
    fmt.Println(Maximum(3, 7))          // int      #4
    fmt.Println(Maximum(2.5, 1.8))      // float64  #5
    fmt.Println(Maximum("go", "java")) // string   #6
}
```

#1 Compares the two values using the > operator, which is allowed by the cmp.Ordered constraint
#2 Returns a if it is greater than b.
#3 Returns b if a is not greater than b.
#4 Calls Maximum with int arguments
#5 Calls Maximum with float64 arguments
#6 Calls Maximum with string arguments

If you are targeting Go 1.18 or 1.19, use `constraints.Ordered` from `golang.org/x/exp/constraints` instead. The function signature and behavior are identical; only the import path differs.

6.6.2 The slices and maps packages

With the addition of generics in Go 1.18, the standard library later added the `slices` and `maps` packages, which provide generic utility functions for working with slices and maps of any element type. These packages make it easy to perform common operations such as sorting, searching, copying, and comparing collections in a type-safe, reusable way, without writing boilerplate code for each specific type.

For example, `slices.Sort` can sort a slice of any ordered type, and `maps.Equal` can compare two maps with the same key and value types for equality. These functions are generic and use type parameters to ensure that only compatible types are used.

Listing 6.25 Example: Generic operations with slices and maps

```
import (
        "slices"
        "maps"
        "fmt"
)

func main() {
        nums := []int{3, 1, 2}
        slices.Sort(nums) // sorts the slice in place          #1
        fmt.Println(nums) // Output: [1 2 3]                  #2

        words := []string{"go", "rust", "java"}
        slices.Sort(words)                                    #3
        fmt.Println(words) // Output: [go java rust]          #4

        m1 := map[string]int{"a": 1, "b": 2}
        m2 := map[string]int{"a": 1, "b": 2}
        equal := maps.Equal(m1, m2) // true                   #5
        fmt.Println(equal)                                    #6
}
```

#1 Sorts the nums slice of integers in place using slices.Sort
#2 Prints the sorted nums slice
#3 Sorts the words slice of strings in place using slices.Sort
#4 Prints the sorted words slice
#5 Compares two maps for equality using maps.Equal
#6 Prints the result of the map equality comparison

These functions become especially expressive with domain types. For example, sorting a `RecipeBox` by serving size uses `slices.SortFunc`, which accepts a generic comparison function instead of requiring the element type to implement `cmp.Ordered`.

Listing 6.26 Example: Sorting a `RecipeBox` with `slices.SortFunc`

```
import "slices"

// Sort recipes from fewest to most servings.
slices.SortFunc(recipeBox, func(a, b Recipe) int {
    return cmp.Compare(a.Yield, b.Yield) #1
})
```

**#1 cmp.Compare returns -1, 0,
or 1 and satisfies the comparison function signature slices.SortFunc expects.**

The same `maps.Keys` function works directly on the `GroceryList` type from chapter 5 (`map[Ingredient]Measurement`) to retrieve a sorted list of ingredient names:

```
ingredients := slices.Sorted(maps.Keys(groceryList))    #1
```

#1 maps.Keys returns an iterator over the keys, and slices.Sorted collects it into an alphabetically sorted []Ingredient. Sorting works because Ingredient is a named string type, which satisfies cmp.Ordered.

6.6.3 The cmp package

The `cmp` package offers generic comparison functions for values of any type that support ordering. This is particularly useful when you need to compare values in a generic context, such as when implementing custom sorting logic or writing generic algorithms that depend on value comparison. The `cmp.Compare` function returns `-1`, `0`, or `1`, depending on whether the first argument is less than, equal to, or greater than the second.

Listing 6.27 Example: Generic comparison with `cmp.Compare`

```
import (
        "cmp"
        "fmt"
)

func MaxOfSlice[T cmp.Ordered](s []T) (T, bool) {
        if len(s) == 0 {                                    #1
                var zero T                                      #2
                return zero, false
        }
        max := s[0]                                         #3
        for _, v := range s[1:] {                           #4
                if cmp.Compare(v, max) > 0 {                    #5
                        max = v
                }
        }
        return max, true                                    #6
}

func main() {
        nums := []int{5, 2, 9, 1}
        max, ok := MaxOfSlice(nums)                         #7
        if ok {
                fmt.Println("Max:", max) // Output: Max: 9   #8
        }
}
```

#1 Checks if the input slice is empty; returns zero value and false if so
#2 Declares a zero value of type T to return when the slice is empty
#3 Initializes max with the first element of the slice
#4 Iterates over the remaining elements of the slice
#5 Uses cmp.Compare to determine if the current value is greater than max
#6 Returns the maximum value found and true to indicate success
#7 Calls MaxOfSlice with a slice of integers
#8 Prints the maximum value if found

By using these standard library packages, you can take full advantage of generics in Go and write concise, reusable, type-safe code for a wide range of data structures and algorithms.

6.7 Just getting started

In this chapter, we've explored the powerful new world of generics in Go. From their historical context and the motivations behind their introduction to the practical details of defining and using type parameters and constraints, including multiple type parameters, you've seen how generics enable you to write more flexible, reusable, and type-safe code. By building generic collections such as lists and maps and implementing utility functions that operate on these abstractions, you've learned how generics can eliminate boilerplate, reduce duplication, and catch errors at compile time.

We've also discussed important considerations and limitations, including performance tradeoffs, language restrictions, and common pitfalls to watch out for. Although generics add expressive power to Go, they do so in a way that preserves the language's core values of simplicity and clarity.

As you continue your journey with Go, generics will become an essential tool in your programming toolkit. They allow you to create robust libraries, reusable components, and maintainable codebases that scale with your needs. By understanding and applying the concepts covered in this chapter, you'll be well-equipped to use generics effectively in your projects, writing Go code that is both elegant and efficient.

Summary

- Generics allow you to write reusable, type-safe code by defining functions, types, and interfaces with type parameters.

- Type parameters can be constrained using built-in interfaces, such as `any` and `comparable`, standard library constraints, or custom type sets.
- The tilde (`~`) operator in constraints enables generic code to work with user-defined types that share the same underlying type as built-in types.
- Generic interfaces and structs enable the creation of flexible data structures such as lists, sets, and maps that work with any compatible type.
- Multiple type parameters allow you to relate different types, as in generic map implementations.
- The Go standard library provides generic packages (`constraints`, `slices`, `maps`, `cmp`) to support common operations on collections and values.
- Generics improve code reuse and maintainability but may introduce performance considerations and new language restrictions.

7 Errors in Go

This chapter covers

- The `error` type
- Best practices for error handling in Go
- Identifying different types of errors
- Creating and using custom error types
- Exceptional situations with `panic` and `recover`

In an ideal world, software would always function as intended, and programs left to their own devices would do their jobs flawlessly forever. This is a nice dream, but even if your code is perfect in every way (which of course it is!), nothing runs in a vacuum. Even the simplest code can encounter unexpected conditions outside its control, so it's important to make sure your software can handle them when they happen.

Go aims to make error handling as straightforward and unremarkable as possible, favoring directness and explicitness over abstractions such as exception handling. It's a prosaic (some might even say boring!) approach, but it also has surprising flexibility, allowing programmers to handle errors as simply or as deeply as they choose.

In this chapter, we will explore how error handling is done the Go way, allowing you to write more resilient, robust software. To demonstrate the various ways you can use errors, we will write a small key/value database.

7.1 Go errors are values

Error handling in Go is nothing fancy from a technical perspective. It's probably one of the simplest error-handling paradigms in modern programming languages. Errors are just values of the `error` type returned from a function call alongside other values. Because `error` is just a type, we can return it like any other value in Go.

To begin our application, let's create a new `main.go` file and define an interface for our key/value database.

Listing 7.1 main.go: Defining functions with error return values

```
package main

type Store interface {
        Put(key string, val string) error #1
        Get(key string) (string, error) #2
        Delete(key string) error
}
```

#1 Function returning only an error
#2 Function returning two values with the second being an error

By convention, a function's error values are returned as the final (or only) value in the return from any function that can fail. While this is not strictly required by the language, it is the standard way of reporting errors, and you will see it in virtually all Go code you encounter, so you should follow suit. This approach means that you will have access to the error as soon as possible after it occurs, so you can figure out what to do with it. It also makes it easy to spot which functions can fail, because they are the ones that list an `error` value in their return type.

Because errors are available as part of the response, you can verify whether the function worked as intended by testing the value to see whether it is equal to `nil`. Let's see

how we can handle errors in our `main` function by using our interface.

Listing 7.2 main.go: A common error-handling flow in Go

```
package main

type Store interface {
        ...
}

func main() {

        var store Store
        err := store.Put("foo", "bar") #1
        if err != nil { #2
                // Handle Error
        }

        if err := store.Put("fizz", "buzz"); err != nil { #3
                // Handle Error
        }
}
```

#1 Capture the error as a new variable from the returning function.
#2 Test to see if the error is nil to handle it.
#3 Simple statement format for error handling

This pattern of calling a function and immediately following that with an `if err != nil` check is one of the most common in the language, and you will end up typing many of these checks in your Go programs. If you are used to a language that uses exceptions for error handling, this approach will likely seem very different to you. It may even feel a tad repetitive, but this repetition is by design. Instead of allowing errors to surface in a separate exception-handling block, which can often be far away from the code that caused them, Go prefers errors to be handled early, at the location where they are first encountered, keeping the

program flow consistent and predictable, even if it is a little repetitive.

Because this is such a common pattern, it is often simplified for functions that return only an error value. In the preceding example, the return values are captured only for the lifetime of the `if` statement, and they can then be used in the body to handle the error.

Reporting errors with return values like this isn't exactly a new approach; in fact, C has been doing this for decades. But Go makes this more flexible by allowing errors to exist alongside the other values returned by a function, rather than using a special "magic value" (such as a negative number) to indicate failure. Go also aims for greater flexibility in what you can do with your error values through the use of the dedicated `error` type.

7.2 The Go error type

So what is this special `error` type that all the fuss is about? It turns out it is a surprisingly lightweight abstraction.

Listing 7.3 The error interface

```
type error interface {
        Error() string
}
```

Yes, that's it! The base type for all errors in Go is nothing more than an interface with a single method, `Error()`, which returns a string representing the message itself. This may look very basic (and it is), but this is again by design. By keeping the interface simple, Go lets any type represent an error while ensuring that the programmer always knows at least two important things about every error:

- It happened (the type is `error`, and the value is not `nil`)
- What is reported (the string returned from the `Error()` method)

Beyond this, errors are free to have any extra information or behavior they need, and the programmer can choose to dig deeper if they wish. Still, knowing nothing more than these two things, you can already do a lot of work with errors in Go:

- Immediately return the error upstream for handling elsewhere.
- Add your additional context to the error by annotating it.
- Log the error and exit.
- Retry the failed operation.
- Ignore the error explicitly, if you want to.

> **IMPORTANT** Few things are more common in Go programming than an error check, and you will find yourself needing to handle errors almost every time you touch existing code. The cardinal rule of good error handling in Go is to handle your results as soon as you get them, whether you simply return them yourself or inspect them more deeply and act on them. Keep this in mind as you proceed through the rest of this chapter.

7.3 Generating new errors

As you've seen, the most important part of an error is what it says about what went wrong, and most of the time, that's all you will need when reporting errors yourself. The most basic error-creation method, `errors.New`, does nothing more than create an error from a string value.

Listing 7.4 Go doc for errors.New()

```
func New(text string) error
        New returns an error that formats as the given text. Each c
all to New
        returns a distinct error value even if the text is identica
l.
```

This can be useful for returning quick error messages when the error is simple, but its use is somewhat limited for error reporting because it only ever returns a static string error message. More often than not, you will reach for `fmt.Errorf` instead, which, as its name and package suggest, allows you to create formatted errors, similar to other `fmt` functions, such as `Printf`.

Let's create an instance of our `Store` interface and put errors in as placeholders until we are ready to start the implementation. This is a common pattern in software development as you start to assemble various interfaces and definitions before you are ready to implement them.

Listing 7.5 main.go: New errors with errors.New() and fmt.Errorf

```
package main

...

func main() {

        var store Store
        store = NewStore()
        err := store.Put("foo", "bar")
        ...
}

type KVStore struct {} #1

func NewStore() Store { #2
        return KVStore{}
}

func (store *KVStore) Put(key string, val string) error {
        return errors.New("not implemented") #3
}

func (store *KVStore) Get(key string) (string, error) {
        return "", fmt.Errorf("no value for key %s", key) #4
}

func (store *KVStore) Delete(key string) error {
        return errors.New("not implemented")
}
```

#1 Create an empty struct which we will use to build our example.
#2 Return a new error using the error package.
#3 Return a formatted error with additional context for the developer.
#4 Return error along with an empty value for multiple returns.

As you can see, `fmt.Errorf` can convey far more information, and many Go programmers use it exclusively, even when the error is just a static string. It is also useful when dealing with other errors, as you will soon see, because it can help you deliver more accurate error reporting, which is one of

the most important things you can do when writing good Go code.

To see these concepts in action, we'll explore a more realistic case study of reporting and refactoring errors in an existing codebase.

7.4 Handling errors

Errors are just pieces of information, and they are not necessarily binary. Instead, they function on a spectrum and require logic to handle them. This is why Go treats errors as values. With values, you have another branch that you are required to think about within your code.

You could easily ignore the value by deciding not to capture it, as in the following listing.

Listing 7.6 An example of not capturing an error

```
val, _ := store.Get("fizz")
```

This is normally an antipattern because we typically want to do something about an error.

Languages that use a `try/catch` methodology attempt to sift through errors using an inheritance structure. This often leads to generalized logic and complicated branches in code to handle errors. With Go's value system, we can also check and handle errors by type, but it requires us to handle them in the moment.

When you decide to handle an error, a common first step is to output information to a file known as a log, where someone can use it to reason about what went wrong in the system. Often, this error entry includes information about

the time of the event and whatever text you wrote to help locate the relevant code for further debugging.

After it's logged, the error can then be passed up to the calling function. This is done by returning a new error message, as we demonstrated before; by wrapping an error, which we'll cover later; or by passing the existing error up. The calling function then repeats the pattern until, ultimately, one function exits the application.

This can be done gracefully by the system by calling `os.Exit(int)`. An `int` value of `0` means that the application exited normally, while any other number indicates that something went wrong. If, instead, we want to exit the application from whatever function we are in, we can call `panic(any)`, which will output whatever string or error you want and provide a code path to help you find your way to the panicked function. Go will perform panics on its own if something serious occurs, and handling them can be difficult, but not impossible. You will learn more about how to manage panics at the end of the chapter.

> **NOTE** Using `panic(nil)` results in a special type of exit that nothing can handle. We will explore this more in the `recovery` section.

Errors are often an afterthought. Yet most code within a system involves some sort of error handling. This is because many things can go wrong, and typically only one thing can be right in a software application. Properly handling errors can help you in the long run by providing the right information in the right places. Let's see how we can do that through logs.

7.5 Logging errors

One of the easiest ways to report an error is to print it to the screen, which you can do with either the `fmt` or `log` packages provided by the standard library.

> **NOTE** We'll explore more advanced logging options in future chapters.

Listing 7.7 Basic error reporting with the fmt package

```
import (
        "errors"
        "fmt"
)

func main() {

        store := NewKVStore() #1
        err := store.Put("foo", "bar")
        if err != nil { #2
                fmt.Print(err) #3
        }

}
```

#1 Create a new store.
#2 Check the value of the error to see if it is set.
#3 If there is an error we can print it on our screen.

We will see the results if we run our application:

```
not implemented
unable to place value: not implemented
```

You may also wish to use the standard `log` package when printing errors and other messages. It works much the same way as `fmt`, but it takes care of adding newlines and supports flags that add extra information to every line.

Listing 7.8 Basic error reporting with the log package

```
import (
        "errors"
        "fmt"
        "log"
)

func main() {

        store := NewKVStore()
        err := store.Put("foo", "bar")
        if err != nil {
                log.Print(err) #1
        }

}
2025/02/21 11:24:55 not implemented
```

#1 Log is a more detailed output that is often used by developers to provide context.

By default, the `log` package adds a timestamp to every message, but it also supports other flags that add info such as the filename and line where the message was logged. You can also disable extra output altogether by calling `log.SetFlags(0)`. We will cover the different logging packages in the standard library in more detail in chapter 10, but you can also check out the `log` in Go's documentation for more info; it can do a lot!

7.6 Annotating error logs with formatting

Unfortunately, errors sometimes need additional context as they move through an application. Lack of error specificity is common, especially when you're dealing with multiple errors in one place that come from code in another package. Error messages can have more or less detail

depending on how they were implemented, and you can't always ensure that the message will be unambiguous enough to show you where to start looking.

Fortunately, the fix is pretty simple: just add a little extra text when you log the message using formatted text. By convention, error messages do not start with a capital letter.

Listing 7.9 main.go: Examples of annotated error logs

```
import (
        "errors"
        "fmt"
        "log"
)

func main() {

        store := NewKVStore()
        err := store.Put("foo", "bar")
        if err != nil {
                log.Print(err)
        }

        if err := store.Put("fizz", "buzz"); err != nil {
                log.Printf("unable to place value: %v", err) #1
        }

}
```

#1 Log's Printf is similar to fmt but with the additional context mentioned before.

Errors can be printed directly using either the `%s` (string) or `%v` (value) format verb, and because of the way Go handles these two internally, there isn't much difference between them. `%v` is slightly better if you plan on printing an error that might be `nil`, but in practice, it's generally fine to use

`%s`. Either one will call the `Error()` method behind the scenes to get the text of the error message for you.

Adding context to errors by annotating them is one of the most important parts of working with errors in Go, whether you are logging them, passing them on, or generating them yourself. Annotation ensures the particular operation that failed is clear, even if the actual failure message isn't.

> **NOTE** Another great technique specific to logging errors is *structured logging*, where your log message can have multiple different fields, representing different things that were going on at the time of the error. This is provided by the `slog` package, which we will cover in chapter 11.

7.7 Sentinel errors

One special type of error value you will find in Go is what's known as a *sentinel error*. The primary distinguishing feature of a sentinel error is that it is a specific, unique value (or *identity*) that represents a particular condition. This means that sentinel errors tend to be less flexible than other errors, which may contain specific data about the condition. But because they are consistent, we can use them as part of our error-handling logic.

Sentinel errors tend to communicate common errors at the package level and are defined in blocks that follow a particular convention: starting the error variable with `Err`. Let's define a few sentinels for our example.

Listing 7.10 main.go: Sentinel errors

```
type KVStore struct {
        m    map[string]string
}
func NewKVStore() *KVStore {
        m := make(map[string]string)
        return &KVStore{
                m: m,
        }
}

var ( #1
        ErrConflict error = errors.New("key already exists") #2
        ErrNotExist error = errors.New("no value exists for key")
)
```

#1 It is convention to create a variable block of errors for others to easily find when searching your source code.
#2 Most sentinel errors start with Err and are common cases in which a developer would want to handle.

We can then replace our existing errors with these new sentinel errors to simplify our logic moving forward.

Listing 7.11 main.go: Using Sentinel errors

```
var (
        ErrConflict error = errors.New("key already exists")
        ErrNotExist error = errors.New("no value exists for key")
)

func (store *KVStore) Put(key string, val string) error {
        if _, ok := store.m[key]; ok {
                return ErrConflict #1
        }
        return nil
}

func (store *KVStore) Get(key string) (string, error) {
        v, ok := store.m[key]
        if !ok {
                return "", ErrNotExist #2
        }
        return v, nil
}

func (store *KVStore) Delete(key string) error {
        _, ok := store.m[key]
        if !ok {
                return ErrNotExist #3
        }
        delete(store.m, key)
        return nil
}
```

#1 In the case of the key already existing we will return our ErrConflict sentinel value.
#2 In the case where the value is not in the map we will want to send the ErrNotExist sentinel.
#3 Sentinel values can be used throughout the package to handle common cases.

We have now created some of our own sentinel errors. As you write more Go code, you will encounter some that are part of the standard library, such as `io.EOF`, which is a common way to indicate that you cannot read any more data. Using these errors will allow you to incorporate your

code into other libraries seamlessly and handle common cases.

Now that we have specific errors to deal with, we should consider how we will handle them as part of our code.

7.8 Identifying different types of errors

So far, our error examples have dealt only with the behavior you can get from the error interface itself, but that's just the tip of the iceberg! Because `error` is an interface, any number of different types could be contained within it. As the old Go proverb goes, "errors are values," and these values can come with all sorts of extra information and logic that can help you learn more about the circumstances in which they occurred.

Go's standard `errors` package provides two primary functions you can use to start exploring this extra information: `errors.Is()` and `errors.As()`. We explored sentinel values in a previous section. Now let's use the sentinel values as the basis for testing these two functions from the error package to help with our business logic.

In our existing logic, our `Put` function does not allow us to insert a value if the key conflicts with an existing one. This means that if a value already exists, we choose not to overwrite it. Still, it would be convenient to force the data into the store so we can create a function that captures the conflict and then handles it.

Listing 7.12 main.go: Using errors.Is to check for sentinel errors

```
type Store interface {
...
        ForcePut(key string, val string) error
}
...

func (store *KVStore) ForcePut(key string, val string) error {
        err := store.Put(key, val)
        if err == nil {
                return nil
        }
        if errors.Is(err, ErrConflict) { #1
                if err := store.Delete(key); err != nil {
                        return errors.New("unable to delete as part
of force put") #2
                }
                if err := store.Put(key, val); err != nil {
                        return errors.New("unable to delete as part
of force put)
                }
                return nil
        }
        return errors.New("unable to force put, unknown error")
}
```

#1 Here we can verify that the error returned was a conflict error and handle it by using the Is function.
#2 If the subsequent functions fail we want to wrap our error into another error to provide more context to the callers.

In many cases, the error message itself isn't particularly important, and a `true` result from `errors.Is()` is enough for you to decide what to do. In the preceding example, the error reports important additional information indicating whether an error occurred during the various steps the function took.

`errors.Is()` exists to help you make these kinds of decisions. Although the error message strings for sentinel errors are generally somewhat helpful, simply returning them as-is

with no additional handling negates the point of using `errors.Is()` in the first place. (The actual error message at the time of this writing would be open missingconfig.cfg: no such file or directory.) The `errors.Is()` function will also search within a stack of errors to determine whether this error occurred.

To wrap errors, we can use a special value in our format function. Let’s update our code to wrap some of these errors using `%w`.

Listing 7.13 main.go: Using errors.Is to handle specific cases for sentinel errors

```
type Store interface {
...
        ForcePut(key string, val string) error
}
...

func (store *KVStore) ForcePut(key string, val string) error {
        err := store.Put(key, val)
        if err == nil {
                return nil
        }
        if errors.Is(err, ErrConflict) {
                if err := store.Delete(key); err != nil {
                        return fmt.Errorf("unable to delete as part
of force put: %w", err) #1
                }
                if err := store.Put(key, val); err != nil {
                        return fmt.Errorf("unable to delete as part
of force put: %w", err)

                }
                return nil
        }
        return fmt.Errorf("unknown error: %w", err)
}
```

#1 We can use the %w value within the format package to wrap errors that can then be grabbed and used by errors.Is and errors.As.

There are many sentinel errors throughout the standard library, most of which start with `Err` for easy identification, such as `flag/ErrHelp` or `io/ErrUnexpectedEOF`.

This section showed you the simplest way to create a custom error that can be used programmatically and handled properly. Let's now see how we can create more complex error types.

7.9 Creating a custom error type

Sometimes we want to include additional information in our errors. This occurs when we are performing a complex set of operations and want to capture some of the details or handle logic based on the error's contents. Remember from earlier that `error` is a type in Go that requires a simple interface to be fulfilled. This allows you to turn any type or struct into an error.

To demonstrate this, we will create a function that allows us to grab multiple values by passing several keys at once. In our error, we want to capture the missing keys so we can display them in our error message. Let's add the method to our interface.

Listing 7.14 main.go: Adding a new method to our interface

```
type Store interface {
...
        BulkGet(keys []string) ([]string, error)
}
```

Now we need to create a new error type. To keep things simple, we will create a new type based on a slice of strings. We will have two methods on the type: the first adds the key that was missing, and the second implements the `Error` interface.

Listing 7.15 main.go: Defining a new error type

```
type bulkGetError []string #1

func (e bulkGetError) addKey(key string) bulkGetError { #2
        return append(e, key)
}

func (e bulkGetError) Error() string { #3
        if len(e) == 0 {
                return ""
        }
        return fmt.Sprintf("missing keys: %s", strings.Join(e,
",","))
}
```

#1 We create our error type which is a string slice to capture missing keys.
#2 The AddKey function is a helper to allow us to extend our slice with new values.
#3 By creating an Error function that returns a string we can fulfil the error type interface.

As you can see, all we need to do for this to become an error is to write a new error message as a string. Here, we just concatenate the missing keys into a comma-separated list.

The following listing shows how a method can use this error.

Listing 7.16 main.go: Using our new error type

```
func (store *KVStore) BulkGet(keys ...string) ([]string, error) {
        var res []string
        var errb bulkGetError #1
        for _, key := range keys {
                v, err := store.Get(key)
                if errors.Is(ErrNotExist) { #2
                        errb = errb.AddKey(key) #3
                        continue #4
                }
                if err != nil {
                        return nil, fmt.Errorf("unexpected error oc
curred when bulk fetching: %w", err)
                }
                res = append(res, v)
        }
        if errb != nil { #5
                return res, errb #6
        }
        return res, nil
}
```

#1 Initialize a new bulk error message.
#2 Check to see if we error on a missing key by checking the sentinel value.
#3 Add the key using the helper function.
#4 Don’t proceed and start the loop over.
#5 Check to see bulkGetError has been set.
#6 If it has we will return our new error.

As we process through the keys, we check the error against our sentinel error for missing keys. If the key doesn’t exist, we can add it to our error type. When all keys have been checked, the error can be returned. In our example, if we don’t encounter an error, the type is never initialized and will be set to nil.

Let’s test this in our main method.

Listing 7.17 main.go: Verifying whether an error is returned in bulk get methods

```
func main() {
        ...

        if _, err := store.BulkGet("foo"); err != nil {
                log.Println(err)
        } else {
                log.Println("no error on bulk get")
        }

        if _, err := store.BulkGet("foo", "fizz"); err != nil {
                log.Println(err) #1
        }
}
```

#1 Our main application has no clue about the complex error we created and uses it just like any other error.

Now we have a custom error that captures some of the data we need to debug. This information is then displayed as part of the error string. This pattern is not new. Many other languages create special types to capture error information. In the Java world, this is done by creating custom exceptions and then handling them accordingly. Go relies on this one interface to pass errors throughout the system.

But what if we want to do more with this error's data? We somehow need to get it back from the `error` type into our custom type. To do that, we will use another special function from the `errors` library.

7.10 Getting the underlying error with errors.As

There are other types of errors that provide more detailed, task-specific information about what went wrong, but to access this information, you first need to peek behind the

`error` interface to get at the actual type. This is where `errors.As()` comes in.

`errors.As()` allows you to specify a particular error type you might expect to see and to create an instance of it for examination if you find it. This is where the function's name comes from: you are asking Go to give you the error *as* it was before it was wrapped up in the `error` value you received.

As an example, let's see if we can handle our error from `BulkGet` in our main function.

Listing 7.18 main.go: Getting the underlying bulk error keys

```
func main() {
        ...
        if _, err := store.BulkGet("foo"); err != nil {
                log.Println(err)
        } else {
                log.Println("no error on bulk get")
        }

        if _, err := store.BulkGet("foo", "fizz"); err != nil {
                log.Println(err)
        }

        if _, err := store.BulkGet("foo", "fizz"); err != nil {
                var berr bulkGetError #1
                if errors.As(err, &berr) { #2
                        for _, key := range berr { #3
                                _ = store.Put(key, "temp") #4
                        }
                }
        }

        if _, err := store.BulkGet("foo"); err != nil {
                log.Println(err)
        } else {
                log.Println("no error on bulk get")
        }
}
```

#1 To analyze the underlying error we will need to grab the underlying type and initialize it.
#2 Using errors.As we can force the error data into the underlying type if possible.
#3 Once cast into its original type we can use it as a slice.
#4 We will handle this by then initializing each value that was missing.

In this example, we handle the error by using `errors.As` to get the underlying list of keys, which we then add to our store as placeholders. Once we're in that block, we have access to all the underlying data we captured earlier. This process allows the errors to flow through the system until they are ready to be handled, if they need to be.

`errors.As()` is a great source of information, provided you know what you're looking for, but it can be a bit tricky to use. It requires you to initialize a value of the type you're interested in first (either a concrete type or an interface), and it requires you to pass a pointer to that value. This allows it to take care of all the messy behind-the-scenes work of making sure the error is of that type or implements that interface. If it returns `true`, you can then access the instance. This can be confusing at first because most error types are implemented with pointers, so you need a new *pointer* to the error type, and you have to pass a pointer to that pointer, as in listing 7.18.

To get an idea of where you might want to employ `errors.As`, consult the `godoc` for the packages you're interested in. Any exported error type is a candidate for unwrapping, and good documentation should tell you what to expect from each call. A common convention in the Go standard library is to suffix such types with `Error`, such as `json.MarshalerError`, and many third-party packages follow this convention as well.

7.11 An unexpected error journey

Sometimes an application has such a severe error that it is unable to proceed and quits instead. This often happens because the application is in a state where it can no longer function or a language rule has been violated. The most common of these errors is the *null or nil reference*, meaning that data that was expected to be there isn't, and the application can't process something because it isn't there. So we are met with a giant error message and an application that has exited.

These little traps happen all the time and cause many bugs within systems. But not all emergency exits occur at

runtime. In some instances, an application is encouraged to just quit and restart. In designing resilient web applications, the application often should be easy to stop and restart without a major disruption. This form of robustness happens by giving developers the power to exit and restart.

A good example is an application's inability to connect to an external service. Let's say our application relies on a database connection to run. Instead of continuing to try to reconnect, the application might restart and begin the process over again. In clustered systems, this can be helpful because it might not be a software problem at all but a hardware problem. By stopping the application, the cluster may then choose to move it to a different node that can connect, making the system resilient through self-healing.

In Go, we can trigger this emergency exit through the `panic` function. You may have already seen a panic message. The Go runtime will spew out a large error message, with the reason for the panic at the top. Often, it will be due to an uninitialized variable causing a nil panic. The `panic` function accepts any value as its parameter to be displayed at the top of the printed messages.

Our storage system relies on an underlying `map` to function properly. If it is uninitialized because the struct was created directly, a nil panic will occur. We don't want our users to have to decipher why they are getting a nil error; instead, we want to provide a better error message. To demonstrate this, let's create a new method on our store to list the available keys.

Listing 7.19 main.go: The Keys method

```
type Store interface {
...
        Keys() []string
}

func (store *KVStore) Keys() []string {
        if store.m == nil { #1
                panic("uninitialized kv store") #2
        }
        out := []string{}
        for k := range store.m {
                out = append(out, k)
        }
        return out
}
```

#1 Accessing a value in an uninitialized map will result in a panic from the underlying Go runtime.
#2 We can create our panic to provide a better context.

Now let's see if we can invoke this panic. To do so, we will edit our `main` function and create the `KVStore` struct directly instead of using the `NewKVStore` method.

Listing 7.20 main.go: Calling the Keys method from an empty store

```
func main() {

        panicStore := &KVStore{} #1
        panicStore.Keys() #2
}

panic: uninitialized kv store

goroutine 1 [running]:
main.(*KVStore).Keys(...)
```

#1 Creating a struct directly will not provide an initialized map.
#2 Calling the Keys function then will result in our custom panic message.

If you were to run this, you would see a `panic: uninitialized kv store` message appear! This is exactly what we wanted: a better description for the user and a way to avoid our `nil` error. But what if we wanted to continue through our application? Is it possible for our application to recover and proceed?

7.12 Don’t panic

If you decide to panic in your application, it is because you know why and have a good reason for it. But sometimes, events occur that you did not plan for, and you may want to attempt to handle the error instead. In chapter 2, you saw the `defer` statement, which is used to do something before a function returns. In the same way, `defer` can be used to handle an unplanned exit from the function you are in, but it will not be enough to handle the panicked routine.

Instead, you need to combine `defer` with a `recover` statement. The `recover` statement is another built-in tool that you can use to handle panics. It must be wrapped in a `defer` function and, in the event of a panic, it will set a value to be returned from `recover`. If nothing happens, you will receive a `nil` value and can proceed as normal. There is a special case when you use `panic(nil)` where Go will not allow you to recover at all, causing the program to exit immediately.

When would you need something like this? Well, in the last section, we discussed how some applications benefit from a quick exit and restart. But some applications may be critical and cannot have any downtime. In this scenario, you would want to do all that you could at various levels to help with recovery so you do not lose data, uptime, or connections

with other systems. This can be true for applications like databases, which require this level of durability.

That being said, let's demonstrate the ability of our `KVStore` to safely handle a `panic` from the `Keys()` function. To do that, we will wrap it in another function for safety. Most of the time, recovery calls occur outside the panicking function, as demonstrated in the following listing.

Listing 7.21 main.go: Recovering from a panic

```
func main() {

        pstore := &KVStore{}
        // pstore.Keys()

        pstore.SafeKeys() #1

        ...
}

func (store *KVStore) SafeKeys() []string {
        defer func() { #2
                if r := recover(); r != nil { #3
                        fmt.Println("recovered! ", r)
                }
        }()
        return store.Keys() #4
}

recovered!  uninitialized kv store
```

#1 We create a new SafeKeys function to prevent a panic and exit of our application.
#2 We create a function that is deferred to recover in case there is an underlying panic.
#3 If there is a panic recover will return a value for us to use.
#4 Calling the Keys function now will either return the values or if panicked will print the recovery message set by the panicked function.

In this function, we use the `defer` statement with `recover()` to print out our underlying panic message. By commenting out

the old function and replacing it with `SafeKeys`, we will still be able to proceed with our application.

7.12.1 Using Panic and Recover in your code

This treatment of `panic` and `recover` has been deliberately brief because, in general, you should not have to use them or worry about them beyond knowing they exist. There is almost always a better solution than calling `panic` in your code, and if you are considering doing so, try to bear the following questions in mind:

- *Is there any chance my panic could affect someone using my code?*—If the answer is "yes," don't panic. Find a way to return an error instead.
- *Is this a truly exceptional situation that absolutely must terminate the entire program immediately?*—If the answer is "no," don't panic. Find a way to return an error instead.
- *Is this a situation in which there is no reasonable way to recover the process, leading to an inconsistent state?*—If the answer is "yes," this is where you might want to consider using `panic`, though finding yourself in this sort of situation often means you need to reconsider how your program is written. The only sorts of places you might leave a `panic` lying around indefinitely are those where it is remotely possible that some external factor beyond your control might hamper your code in a way that is difficult to predict, such as a very rare edge case causing you to run out of memory unexpectedly. It should be noted that panics are hidden within the code, unlike other languages where error types are part of the function signature. Therefore, if your function will panic, it should be documented as a comment for the generated `godoc`.

`recover` can sometimes be a stopgap solution if you are encountering sporadic panics in code you are using from elsewhere, but the right approach is almost always to track down the error and fix it (if it's your own code) or to try to identify the source of the problem and get the author of the

code to address it. Failing that, work around it by changing your usage or finding a replacement.

Summary

- Errors in Go are values returned from a function, typically as the last return value.
- Values of any type can be errors, as long as that type implements the `error` interface by defining an `Error() string` method.
- Errors should be handled as soon as possible where they are returned, typically by returning them to the calling function with a meaningful annotation, if necessary.
- When writing your packages, try to make the errors you return relevant to the function of each package. Using your own error types can be helpful here. Try not to return errors without this context, such as errors from other packages or errors that wrap them. Correspondingly, make sure you wrap errors from within your package that might otherwise be obscured.
- If you need to test for a particular error, use `errors.Is()` to check for it. If you need to test for a particular error type, use `errors.As()`.
- Panics indicate a condition that is truly unexpected, and they should be treated with care because they break the normal control flow of your program. `panic` is not intended as an error-handling mechanism, and if the structure of your program necessitates a panic, you need to document this possibility. Otherwise, use ordinary errors.

8 Testing and tooling

This chapter covers

- Unit testing with the `testing` package
- Test-driven development
- Code coverage
- Test fixtures and helpers
- Benchmarking
- Fuzzing

Testing sometimes gets a bad rap as a chore or an afterthought, as something to be added later after the difficult work of solving problems is done. Often, you will hear programmers talk about testing in terms of "peace of mind," meaning that tests can help verify that your software continues to work as expected, even as new features are added or parts of the underlying logic are changed. If the tests keep passing, the reasoning goes, your code is still functioning as intended, or at least in all the ways that matter.

Peace of mind is certainly an important benefit of testing, but if all you are concerned with is verifying your code after the fact, then you're missing out on much of what testing can do for you, especially in a language with a solid set of tooling like Go.

Rather than being defensive, tests can be proactive: you can wield them like a scalpel instead of a shield to identify and target areas for improvement in existing code.

The extended tooling around profiling and testing remains one of Go's enduring strengths and, when combined with benchmarking, it enables you to identify and target areas for performance improvement. By creating a benchmark and adding a couple of flags, you can get targeted statistics on CPU usage and memory allocation, sometimes down to the individual line.

Using the built-in fuzzer, you can test input-driven tests more thoroughly, and you can proactively search for breaking bugs by testing your invariants against a randomly generated corpus of data.

Tests can even help drive code and enforce contracts through test-driven development, a workflow in which your tests drive your code, not vice versa.

There's a great deal you can do with testing and tooling in Go, and while this chapter can only scratch the surface, we hope it gives you the motivation to explore this ecosystem.

8.1 Test files and functions

Tests for Go projects are defined in files with the naming pattern `*_test.go`. Any file matching this pattern is ignored during the normal build process but is compiled into the package during testing as you can see in figure 8.1 below.

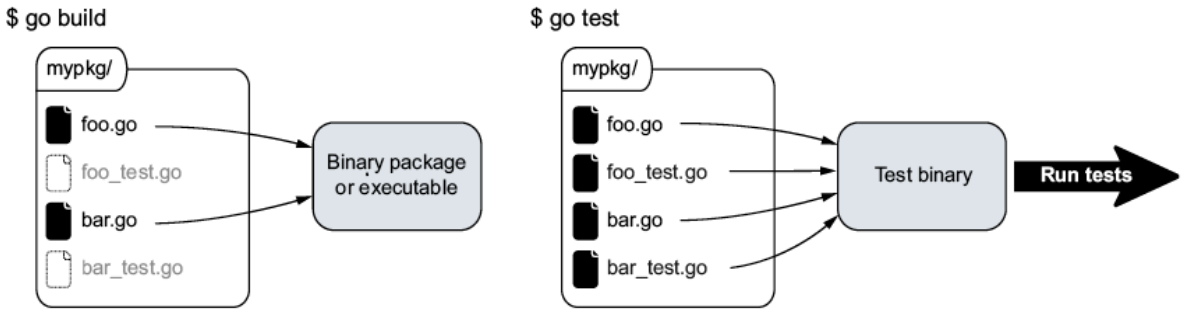


Figure 8.1 Files used in build vs. testing

For a function to be run by the `go test` tool, it needs to have a name and an argument list that follow a particular format. For example, tests use the `func TestXxx(t *testing.T)` format, where `Xxx` is a unique name for the test and starts with an uppercase letter.

In addition to test functions, we will cover the three other function types listed in table 8.1, each of which has its own naming and argument conventions, depending on its role.

Table 8.1 Types of test functions

Function name	Argument type	Type	Purpose	Example
`TestXxx`	`*testing.T`	Test	Test functionality with pass/fail status	`func TestMyFeature(t *testing.T){}`
`BenchmarkXxx`	`*testing.B`	Performance benchmark	Report performance data and enable profiling	`func BenchmarkMethod(b *testing.B){}`
`FuzzXxx`	`*testing.F`	Input fuzzing	Test inputs and detect edge cases	`func FuzzMyInputs(f *testing.F){}`
`ExampleXxx`	(none)	Testable example	Document usage and provide additional testing	`func ExampleUsage(){}`

For each of these types, the function needs a name that follows the correct format and, with the exception of examples, must take a single argument of the proper type, which it will use to set test parameters and report results.

> **IMPORTANT** If your tests don't seem to be running or if you see a "no tests to run" warning, make sure your tests are correctly defined in a `_test.go` file and follow the proper naming and argument convention. See `go help testfunc` for more details.

8.2 Running tests with go test

You run tests by invoking the test tool with `go test`, which compiles all `_test.go` files in a package into a special test executable that runs behind the scenes so it can track test progress, enforce timeouts, and safely capture unexpected problems, such as panics. Using command flags, you can control various aspects of test running, such as setting the timeout duration or selecting which tests you want to run. See `go help test` and `go help testflag` for a detailed rundown.

By default, running `go test` runs a test build in the current directory and prints a summary of the results. To see this in action, create a `testexample` directory and initialize it as a Go module by running `go mod init testexample` from within the directory. Then fire up your text editor and add the following contents to a file called `example_test.go`.

Listing 8.1 Example test function

```
// example_test.go
package example

import (
        "testing"
)

func TestPass(t *testing.T) {
        t.Log("This test will pass.")
}
```

This file doesn’t do much beyond printing a single log message, but because it follows the correct format for a testing function, it will be run if you execute `go test`, which defaults to running all the tests it finds in the current directory.

Listing 8.2 Running the example test

```
$ go test
PASS
ok      example    0.104s
```

The `PASS` line in the output represents the combined result of all tests in this run. Here, all the tests have passed. The `ok` message is the package summary, which is always displayed, and it similarly indicates that there were no problems compiling the package and that everything went as expected.

This test output might seem a little underwhelming, but don’t worry; that’s a good thing when it comes to tests. By default, `go test` prints information only about tests that fail, so that problems are easier to find. No news is good news!

It might seem odd to see a passing result from a test that doesn’t test anything, but this is by design. Rather than requiring you to manually mark tests as passed or failed, `go test` assumes that, as long as a test doesn’t time out, doesn’t panic, and doesn’t explicitly fail, it has passed. This helps simplify test writing because you need to concern yourself only with whether a failure has occurred.

To verify that the new test was run, you can add the `-v` flag to turn on verbose mode, which shows output for all tests, regardless of their result.

Listing 8.3 go test verbose output

```
$ go test -v
=== RUN   TestPass
    example_test.go:5: This test will pass.
--- PASS: TestPass (0.00s)
PASS
ok      example    0.287s
```

In verbose mode, you get a series of blocks for each test. `=== RUN <TestName>` indicates the start of a particular test, and the `---` line indicates the result along with the elapsed time. Logs appear between these two lines, along with the filename and line number where the log occurred.

8.3 Test functions

Tests are just functions that run against a piece of code. When a test is isolated, meaning it doesn't have any external dependencies, it is known as a *unit test* because you are testing a single unit of software. These tests often make up the bulk of the tests you will write.

Go handles testing by convention, as it does everything else. Test functions must begin with the word `Test` and have a `t *testing.T` parameter in their signature without returning a value, as you can see in figure 8.2. The `t` parameter contains all the tooling you need to either pass or fail a test, along with some other functions that we will explore later.

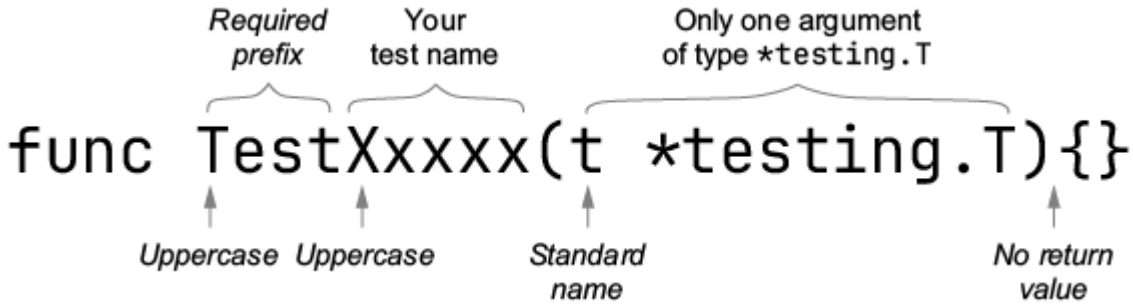


Figure 8.2 Signature of a test function

We will not discuss testing theory in depth here, but tests often follow an *arrange*, *act*, *assert* flow. You first set up the parameters of your test, execute the test, and then verify the results. You also need to consider what you are testing. Some developers like to test the internal pieces of the package, while others prefer to approach it as consumers of a library. Go allows you to do this by separating your test code into its own package with the suffix `_test`.

> **NOTE** If you're looking to dive deeper into testing theory, grab a copy of *Test-Driven Development* by Kent Beck (Addison-Wesley Professional, 2002), a foundational text in modern test-driven development. In figure 8.3, you can see the steps for best testing practices.

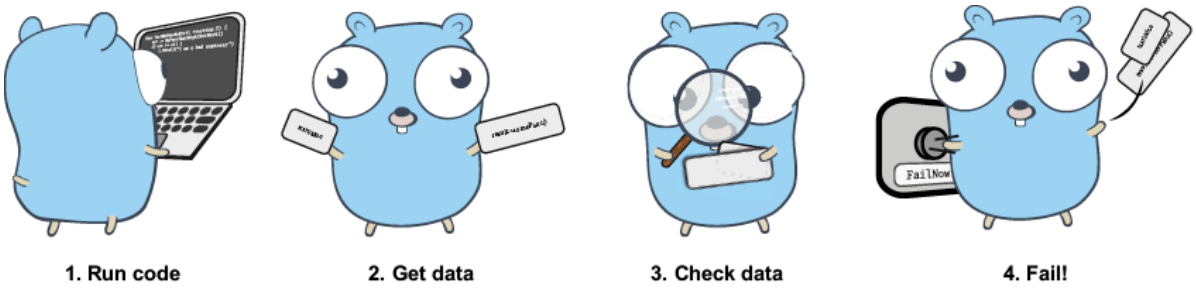


Figure 8.3 Testing steps

No matter your approach to testing, always remember that you are trying to verify that the code works as you expect it to; you are not writing the code just to pass the test. If your test does not work as expected, return a value that

helps explain what the expectation was and what the result ended up being.

Listing 8.4 An example test

```
func TestMyFeature(t *testing.T) {
        // 1. Define some test data.
        arg := "my data"

        // 2. Define an expected result.
        want := "expected value"

        // 3. Interact with your code and get some data.
        got := FunctionToTest(arg)

        // 4. Inspect the results.
        if got != want {
                // 5. Mark the test as failed, and log the reason w
hy.
                t.Errorf("bad result - want: %v, got %v", want, go
t)
        }
}
```

Tests may be more or less complex and may involve one or more cycles of these steps to validate the code they are testing, but the basic workflow is the same everywhere.

8.4 Test life cycle

Instead of returning a value to indicate the test results, Go test functions follow an interactive workflow centered on the `*testing.T` value passed into every function as its sole argument. By convention, this parameter is called `t` for brevity.

The `t` parameter is initialized for each function by `go test`, and it is the point of contact between your code and the test runner that executes it. By calling methods on `t`, you

can log messages, skip tests, and mark tests as failed. You can think of `t` as your test control panel, with all the buttons and levers necessary to control your test run, including a big red button labeled "REJECT". See `go doc -all testing.T` for everything this type can do.

To see what happens when a test fails, you can add a function, `TestFail`, to `example_test.go` and call one of the provided failure methods, `t.Error`. This informs `go test` that the function should be marked as failed, which causes it to be reported at the end of the test run.

Listing 8.5 Example of a test failure

```
// example_test.go
package example

import (
        "testing"
)

func TestPass(t *testing.T) {
        t.Log("This test will pass.")
}

func TestFail(t *testing.T) {
        t.Error("This test will fail.")
        if 0 != 1 {
                t.Error("0 is not equal to 1")
        }
}
```

Running `go test` again reveals a failing test.

Listing 8.6 go test failure output

```
$ go test
--- FAIL: TestFail (0.00s)
    example_test.go:16: This test will fail.
    example_test.go:18: 0 is not equal to 1
FAIL
exit status 1
FAIL    example    0.193s
```

When one or more tests fail, `go test` reports the results with a heading for each failed test, including the log messages printed, along with a helpful annotation of the file and line number where they were logged, which will help you locate failures.

WHY DON'T TESTS RETURN A RESULT?

Using `t` to report errors may seem unusual, but it works well for testing because it allows for multiple failures and gives you better control over what to do with them.

In a normal Go function, you are encouraged to use `error` values to indicate failure (see chapter 7), but tests may encounter multiple errors throughout their run, and returning only one error at a time would be inefficient. Tests are all about the big picture, so using a reporting type like `t` lets you log as many failures as needed while still being able to exit the test immediately if necessary.

Altogether, `testing.T` provides six methods for indicating test failure. These can be divided into two categories: fatal errors, which exit the test immediately, and non-fatal failures, which continue to run the test after marking it as failed.

1. Non-fatal failure
   - `Fail` - Marks the test as failed but continues execution
   - `Error`: Logs a `println`-style message and calls `t.Fail`
   - `Errorf` logs a formatted message and calls `t.Fail`
2. Fatal failure
   - `FailNow` - Marks the test as failed and exits the test immediately
   - `Fatal`: Logs a println-style message and calls `t.FailNow`
   - `Fatalf` logs a formatted message and calls `t.FailNow`

To illustrate the difference between these two types of test failure, you can add one last function to `example_test.go` that swaps `Error` with `Fatal`.

Listing 8.7 Examples of fatal and non-fatal methods

```
// example_test.go
func TestFail(t *testing.T) {
        t.Error("This test will fail.")
        if 0 != 1 {
                t.Error("0 is not equal to 1")
        }
}

func TestFatal(t *testing.T) {
        t.Fatal("This test will also fail.")
        if 1 != 2 {
                t.Error("2 is not equal to 2")
        }
}
```

These are the test results.

Listing 8.8 `go test` output for fatal and nonfatal failures

```
--- FAIL: TestFail (0.00s)
    example_test.go:16: This test will fail.
    example_test.go:18: 0 is not equal to 1
--- FAIL: TestFatal (0.00s)
    example_test.go:23: This test will also fail.
FAIL
exit status 1
FAIL    example    0.193s
```

Both tests fail and log two failures, but for `TestFatal`, only the first message is logged because the fatal method exits the function early.

Fatal methods are typically used when a failure makes it impossible to continue testing. Often, this is the result of a test dependency or other invariant that unexpectedly breaks.

Listing 8.9 Example of fatal test conditions

```
func TestMyType(t *testing.T) {
        m, err := NewMyType()
        if err != nil {
                // r is nil, we can't test a type when we don't hav
e a value.
                t.Fatalf("NewReplacer() - %v", err)
        }
        // otherwise, test methods as normal
        err = m.SomeMethod()
        if err != nil {
                t.Errorf("SomeMethod() - %v", err)
        }
}

func TestTempFile(t *testing.T) {
        // Get a temp file for testing.
        f, err := os.CreateTemp("", "example")
        if err != nil {
                // Shouldn't happen -- can't continue.
                t.Fatalf("os.CreateTemp - unexpected error: %v", er
r)
        }
}
```

A good rule of thumb when choosing your error methods is to stick with non-fatal failure methods unless you are sure the test cannot continue otherwise. You will be thankful for this when debugging, since it’s not uncommon for a breaking change to introduce failures that surface in multiple places, and an early exit risks turning debugging into a protracted game of whack-a-mole, where you fix one failure only to have a new one pop up on the very next run.

8.5 Test-driven design: A calculator with a twist

Now that you understand the basics, let's take a look at how you might apply testing strategies to a real-life project.

One of the first choices you can make when starting a new project or adding features to an existing project is whether to write your code first or start with tests. Starting with tests is known as *test-driven development* (TDD), and it has the advantage of giving you clear, measurable targets ahead of time (your tests) that your code can then fulfill. This gives you the opportunity to start working with your API immediately, so you can see if it works for you.

Let's design a simple `Calculator` type. It should support the basic operations of addition, multiplication, division, and subtraction, but with a few extra features to make it interesting. First, it should support storing the last calculation in memory and retrieving it later, just like the pocket calculators from days of yore. Second, instead of using separate methods for each operation, it should have a single method that takes a string for the operation and two operand values to run the operation on. Finally, it should return errors if the operation is unknown or in the case of division by zero, which would otherwise cause a panic.

With the basic idea in place, it's time to design the type. Based on the requirements, the type should define the following methods:

1. `Calculate(operation string, a int, b int) (int, error)`
   - Description: Run a calculation using the "add", "sub", "mul", and "div" operations.
   - Returns: An `int` result, and an `error` if `b` is zero or the operation is invalid.
2. `Store()`
   - Description: Store the result of the last calculation in memory. If the last calculation resulted in an error, do nothing.

    - Returns: none.
3. `Recall() int`
    - Description: Retrieves the value stored in memory.
    - Returns: The `int` value set by the last successful call to `Store`.

The first order of business in a TDD workflow is to create a skeleton for the code being tested so that the tests can run and you can start writing them. Create a new module directory called `basicmath` and initialize the module by running `go mod init basicmath` inside the directory. Then add the following code to `calculator.go`.

Listing 8.10 Calculator skeleton

```
package basicmath

import "errors"

var (
        ErrDivByZero        = errors.New("divide by zero")
        ErrUnknownOperation = errors.New("unknown operation")
)

type Calculator struct{}

func NewCalculator() *Calculator {
        return &Calculator{}
}

func (c *Calculator) Calculate(operation string, a, b int) (int, er
ror) {
        // not implemented
        return 0, errors.New("unimplemented")
}

func (c *Calculator) Store() {
        // not implemented
}

func (c *Calculator) Recall() int {
        // not implemented
        return 0
}
```

This code implements all the necessary methods but doesn't implement any functionality yet, aside from returning zero values where necessary. With this foundation in place, you can start writing tests.

8.5.1 Black-box testing

To start your tests for the `Calculator` type, you need to create a testing file for the tests to live in. This file can be

anything you like, as long as it has the format `*_test.go`, but the usual convention is to append `_test` to the name of the file containing the code to be tested so that it's easy to find. In this case, that would be `calculator_test.go`, so create that file inside the `basicmath` directory with the following contents.

Listing 8.11 The testing file skeleton

```
package basicmath_test

import (
        "errors"
        "testing"

        "basicmath"
)
```

Normally, Go requires all files in the same directory to be in the same package, but the exception for test files is that they can also use the package `<package name>_test`, which allows them to treat the target package as a dependency and test it from an outside perspective. This is known as *black box testing*, because there is no visibility inside the package being tested. Correspondingly, test files that use the same package can perform *open box testing*, also known as *white box* or *clear box* testing, where they have full access to the target.

Open box tests are powerful because they can access unexported code, but they can also create conditions that would be impossible to encounter in normal operation, producing misleading results or a false sense of security. By contrast, black box tests can use your code only as a user would, making their results more realistic and the tests safer to write. Black box testing also works well with TDD, which is focused on implementing features.

Even if you are not following a TDD approach, consider using black-box tests wherever possible. There may be occasions, such as regression testing, when it is much more efficient to access package internals, but it's better to consider these the exception rather than the rule.

8.5.2 Writing tests for error conditions

One thing to keep your eye out for when testing a package is any exported error types or sentinel errors, since these represent specific error conditions that you want to make sure work as intended. Try to find clear paths for reproducing them and make sure that your tests can exercise them. That way, you can ensure that the failures are well-defined and predictable. For this package, there are two named sentinel errors, `ErrDivByZero` and `ErrUnknownOperation`, both of which have a clear reproduction case, so it makes sense to write a test for each of them.

Depending on how well-defined the conditions for an error message are, you can perform an action that should produce the error and verify that the returned error is the same one. Since both errors are sentinel errors, `errors.Is` is the tool for the job.

The design states that `ErrDivByZero` should be returned if the second argument in a division is `0`, so the test should perform that action and check the result. If the error is not the correct value, the test should fail.

Listing 8.12 Testing divide by zero

```
func TestErrDivideByZero(t *testing.T) {
        c := basicmath.NewCalculator()
        got, err := c.Calculate("div", 1, 0)
        if !errors.Is(err, basicmath.ErrDivByZero) {
                t.Errorf("expected ErrDivByZero - got: %v", err)
        }
        if got != 0 {
                t.Errorf("error result should be 0 - got: %d", got)
        }
}
```

The test for `ErrUknownOperation` is similar; the only failing condition is any operator that isn't one of the four allowed. Since the operation is based on a string, you can also test a few likely scenarios, such as the empty string and a couple of misspellings. For this, you can loop over a hardcoded string slice and add a log message for each one using `t.Log` or `t.Logf` to give context for the error.

Listing 8.13 Testing unknown operation

```
func TestErrUnknownOperation(t *testing.T) {
        c := basicmath.NewCalculator()
        for _, op := range []string{"", "mult", "asd"} {
                t.Logf("test invalid operation %q", op)

                got, err := c.Calculate(op, 1, 1)
                if !errors.Is(err, basicmath.ErrUnknownOperation) {
                        t.Errorf("expected ErrUnknownOperation - go
t: %v", err)
                }
                if got != 0 {
                        t.Errorf("error result should be 0 - got: %
d", got)
                }
        }
}
```

If you run `go test` now, you will see a litany of failures, which is fine because `Calculate` is still unimplemented. Using a test-driven workflow means starting with many failures that you winnow down as you complete the implementation.

Listing 8.14 go test output for the error tests

```
go test
--- FAIL: TestErrDivideByZero (0.00s)
    calculator_test.go:14: expected ErrDivByZero - got: unimplement
ed
--- FAIL: TestErrUnknownOperation (0.00s)
    calculator_test.go:24: test invalid operation ""
    calculator_test.go:28: expected ErrUnknownOperation - got: unim
plemented
    calculator_test.go:24: test invalid operation "mult"
    calculator_test.go:28: expected ErrUnknownOperation - got: unim
plemented
    calculator_test.go:24: test invalid operation "asd"
    calculator_test.go:28: expected ErrUnknownOperation - got: unim
plemented
FAIL
FAIL    command-line-arguments  0.235s
FAIL
```

8.5.3 Writing good test logs

Good logging is crucial to effective testing in Go because the output from `go test` is often the only information you have to go on. If your messages are too vague, debugging will be more difficult, but too many log messages, or messages that are overly long, can crowd the output.

As with logs, be specific but brief. Remember that `go test` already adds the name of the test and attaches the file location to every log message, so there's no need to get more specific about *where* the failure occurred. Instead, concentrate on *why* the error occurred.

Lead with what you expected or what was unexpected about the failure, followed by the specifics. If you are expecting a certain value, log that value as `want` or `wanted` and the value you received as `got`. If you are handling an unexpected condition, such as an error, mention that it was unexpected and log the value. If you are making the same check repeatedly with different arguments, you can either add these to your formatted error or, if that gets too busy, log the values on every iteration using `t.Log` or `t.Logf`.

Here are some examples of effective failure messages:

- `funcName - want: a, got: b`
- `expected ErrSomeError - got: <nil>`
- `unexpected error: <error value>`

Finally, avoid using `t.Fail` and `t.FailNow` unless there are log messages immediately nearby to help locate the failure. These methods do not log anything, so they provide no indication of where a failure occurred.

8.5.4 Choosing testing targets

You can organize your tests in several different ways, depending on what is being tested. For a type like `Calculator`, you can unit test all the methods individually, or you can test related methods as a single feature if it makes sense for your use case. For example, the `Store()` and `Recall()` methods are both related to the calculator memory feature, and verifying one naturally involves the other, so it is easier to test them together in a function called `TestMemory`.

For this test, you want to verify all the functionality mentioned in the specification: that a value can be stored,

that this value is the result of the previous calculation, and that the value can be retrieved again.

Listing 8.15 Testing the memory feature

```
func TestMemory(t *testing.T) {
        c := basicmath.NewCalculator()
        // memory should hold 0 for a new value
        got := c.Recall()
        if got != 0 {
                t.Errorf("initial memory value should be 0 - got: %
d", got)
        }
        // run a calculation so that we have a new value to store i
n memory
        t.Logf("Calculate - div, 12, 3")
        got, err := c.Calculate("div", 12, 3)
        if err != nil {
                t.Fatalf("unexpected error: %v", err)
        }
        if got != 4 {
                t.Errorf("incorrect result - want: 4, got: %d", go
t)
        }
        // Ensure the value is stored and correctly retrieved again
        c.Store()
        got = c.Recall()
        if got != 4 {
                t.Errorf("recall - want: 4, got: %d", got)
        }
}
```

On lines 4-6, the test checks that the initial value stored in memory for a new `Calculator` is `0`. This is intended to help verify that the value is updated later, but in a TDD workflow, it is also a new design constraint that the code will need to implement. This is one way testing can help drive development in a TDD workflow. (In a real-world situation, you might update the design document to reflect the new invariant or file a ticket so the decision can be discussed.)

Lines 9-16 run a calculation using the `Calculate` method and verify the result. Because this section involves two separate checks against the result from `Calculate`, the arguments are logged using `t.Logf` on line 9. This ensures that they appear in the logs above either of the two failures. An error from the `Calculate` call is also defined as fatal because the design of `Store` requires a valid calculation, and the test cannot continue without one.

Although the verification on line 14 isn't strictly necessary for completeness (because `Calculate` will be tested on its own), a little overlap doesn't hurt, and it gives `TestMemory` another place for `Calculate` errors to surface.

Having set up the initial conditions for testing, lines 18-22 call `Store` followed by `Recall` and test to make sure that the resulting value is the same as in the prior step.

At this point, you could run `go test` again to see all the failures, but that would be pretty noisy. Instead, you can zero in on just `TestMemory` by using the `-run` flag, which takes a regular expression for matching test names.

Listing 8.16 Using `go test -run` to run a single test

```
go test -run TestMemory
--- FAIL: TestMemory (0.00s)
    calculator_test.go:98: Calculate - div, 12, 3
    calculator_test.go:101: unexpected error: unimplemented
FAIL
exit status 1
FAIL    basicmath  0.185s
```

8.5.5 Table-driven testing

The final test you need to write for the `Calculator` type is for the `Calculate` method, which performs all of the

mathematical operations. You'll want to test each operation with a variety of arguments for completeness' sake, but repeating the same steps multiple times would be extremely time-consuming because you would need to write them over and over again. This is no fun, and it's ugly and hard to follow.

Listing 8.17 Testing `Calculate` permutations the hard way

```
func TestCalculatePainful(t *testing.T) {
        c := basicmath.NewCalculator()

        want := 2
        got, err := c.Calculate("div", 10, 5)
        if err != nil {
                t.Errorf("unexpected error - %v", err)
        }
        if got != want {
                t.Errorf("incorrect result - want: %d, got: %d", wa
nt, got)
        }

        want = -8
        got, err = c.Calculate("add", -12, 4)
        if err != nil {
                t.Errorf("unexpected error - %v", err)
        }
        if got != want {
                t.Errorf("incorrect result - want: %d, got: %d", wa
nt, got)
        }

        // and so on...
}
```

Fortunately, you can turn the repetition in these sorts of tests into an advantage by extracting the testing logic and putting it in a loop. This approach is known as *table-driven testing*.

The first step in converting a test case to be table-driven is to create a `struct` to hold all the values necessary for each test case. For example, if your test calls a function `Foo(strVal string, intVal int) int` and tests the results, you will need three values in your struct: two for the arguments and one for the result you are going to test. This is outlined in figure 8.4.

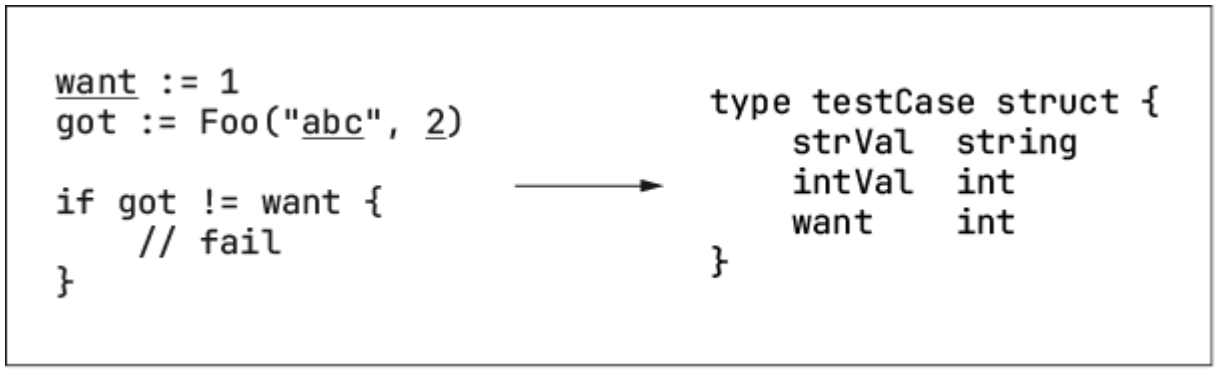


Figure 8.4 Creating a struct to hold test case values

Then you can create a slice of this type, with each value representing a single iteration of your test. Figure 8.5 shows how to structure the validations so you have repeatable validation steps per testCase.

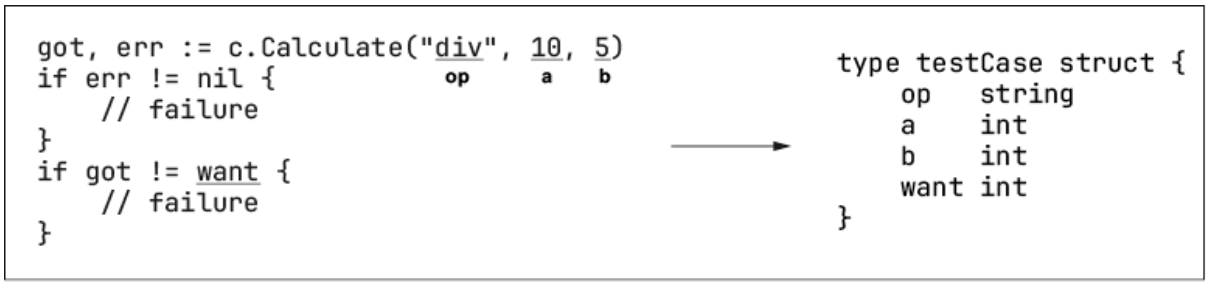


Figure 8.5 Testing a slice of test cases in a loop

Applying this to the `Calculate` method, this means you'll need a `string` for the operation, two `int` values for the operands, and a third `int` for the expected result. This gives you your `testCase` struct, which can be declared locally in the test function itself.

Listing 8.18 Testing `Calculate` permutations with table-driven testing

```
func TestCalculate(t *testing.T) {
        type testCase struct {
                op   string
                a    int
                b    int
                want int
        }

        c := basicmath.NewCalculator()

        for _, tc := range []testCase{
                {op: "div", a: 10, b: 5, want: 2},
                {op: "div", a: -12, b: 4, want: -3},
                {op: "add", a: 1, b: 0, want: 1},
                {op: "add", a: 2, b: -4, want: -2},
                {op: "mul", a: 100, b: 0, want: 0},
                {op: "mul", a: 2, b: -4, want: -8},
                {op: "sub", a: 20, b: 7, want: 13},
                {op: "sub", a: 5, b: -4, want: 9},
        } {
                t.Log("Calculate - ", tc.op, tc.a, tc.b)
                got, err := c.Calculate(tc.op, tc.a, tc.b)
                if err != nil {
                        t.Errorf("unexpected error: %v", err)
                        continue
                }
                if got != tc.want {
                        t.Errorf("incorrect result - want: %d got:
%d", tc.want, got)
                }
        }
}
```

This technique makes tests much more readable because all the test cases are grouped at the start of the test. It also makes the test easier to maintain because the testing code is written only once, and adding or removing test cases is as easy as adding or removing entries from the slice.

Another benefit is that a single log line can print a header for each case simply by printing the values in the struct.

```
$ go test -run TestCalculate
--- FAIL: TestCalculate (0.00s)
    calculator_test.go:82: Calculate - div 10 5
    calculator_test.go:85: unexpected error: unimplemented
    calculator_test.go:82: Calculate - div -12 4
        #1
FAIL
exit status 1
FAIL    basicmath  0.185s
```

#1 And so on…

8.5.6 Implementation and initial testing

With all the tests in place, you can finally get down to the business of filling in the calculator type itself. Guided by the design and the tests you've come up with, you know that it needs two `int`s to hold the memory value and the last calculation, as well as an `error` that `Store` can check to avoid storing an invalid value in memory.

Listing 8.19 The `Calculator` type

```
// Calculator is a calculator type with memory.#1
// The zero value is a calculator ready to use.
type Calculator struct {
        mem    int
        result int
        err    error
}

// NewCalculator returns a basic mathematical calculator.
func NewCalculator() *Calculator {
        return &Calculator{}
}
```

#1 Can't forget the godoc!

Next come the methods. `Calculate` can be implemented using a type switch and a single error check for a zero operand to division, while `Store` and `Recall` are even more straightforward, only requiring a simple check on the part of `Store` for an invalid result of the last calculation.

Listing 8.20 `Calculator` methods

```
// Calculate performs a calculation using the supplied operator and
integer
// operands.
// Valid operators are: add, sub, mul, div.
func (c *Calculator) Calculate(operation string, a, b int) (int, er
ror) {
        if operation == "div" && b == 0 {#1
                c.err = ErrDivByZero#2
                c.result = 0
                return 0, ErrDivByZero
        }
        switch operation {
        case "add":
                c.result = a + b #3
        case "sub":
                c.result = a - b
        case "mul":
                c.result = a * b
        case "div":
                c.result = a / b
        default:
                c.err = ErrUnknownOperation
                return 0, ErrUnknownOperation
        }
        return c.result, nil
}

// Store, stores the current result in memory, if it's valid. Other
wise, it
// does nothing.
func (c *Calculator) Store() {
        if c.err != nil {
                return
        }
        c.mem = c.result
}

// Recall returns the current value stored in memory.
func (c *Calculator) Recall() int {
        return c.mem
}
```

#1 Check the unhappy path first.
#2 Place the result where Store can see it.
#3 Setting the result instead of returning resets the result.

And what does `go test` have to say?Nice! Behold the power of test-driven development!

Listing 8.21 Output of `go test`

```
go test
PASS
ok      basicmath  0.200s
```

8.5.7 Code coverage

Ideally, tests should be able to run as much of the target package as possible. This is typically measured in terms of *code coverage*, or the percentage of the total statements in the target package that your tests executed.

In practice, getting 100% coverage is not always a realistic goal, and you are likely to fall short in certain areas, especially those involving more esoteric or exceptional circumstances, such as failed file access or problems with network activity.

Still, it’s always a good idea to check your code coverage periodically while you are developing tests, just to make sure you haven’t missed anything you wanted to target.

To view the code coverage for a particular package, you can run `go test -cover`, which runs the tests as normal and reports the percentage of statements that your tests managed to run.

Listing 8.22 Output of `go test -cover` for the `basicmath` package

```
$ go test -cover
PASS
coverage: 93.8% of statements
ok      basicmath  0.195s
```

93.8% isn't too bad under most circumstances, but there isn't a lot of code in this package, so it's worth looking into a little more deeply.

If you are using an IDE or editor that supports it, you can probably issue a command that shows your code coverage with fancy highlighting (and you should familiarize yourself with this functionality if you have it), but you can also view code coverage in a normal web browser using the built-in tools. To do this, you first need to generate a coverage report: a special text file created by `go test` that contains annotated listings of lines in your source along with their coverage status.

To generate this file, run `go test -coverprofile=<file name>`. `go test` will once again run all the tests, output the coverage, and write the specified file. For this example, we used `coverage.out`, but any name will do.

Listing 8.23 Using `go test -coverprofile` to generate a coverage profile file

```
$ go test -coverprofile=coverage.out
PASS
coverage: 93.8% of statements
ok      basicmath  0.195s

$ tail -n 3 coverage.out  #1
goinaction.book/ch09/basicmath/calculator.go:50.18,52.3 1 0
goinaction.book/ch09/basicmath/calculator.go:53.2,53.18 1 1
goinaction.book/ch09/basicmath/calculator.go:57.35,59.2 1 1
```

#1 tail is a standard POSIX command to print the last few lines of a file.

By itself, this output isn't very useful because it is intended to be read by other software, but you can open it in a browser using `go tool cover`, which loads the files referenced in the coverage profile into a web application.

Listing 8.24 Opening the code coverage in the browser

```
$ go tool cover -html=coverage.out
```

This command will parse the coverage output file and load any files that reference your source code. It will then apply the annotations and render everything to a temporary HTML file before opening the file in your default browser.

The page you see should look something like figure 8.6, with color highlighting indicating the coverage status of each statement in the file (green for covered, red for uncovered). Files are listed alongside their coverage percentages in the dropdown menu at the top, and you can select a new file to change the view.

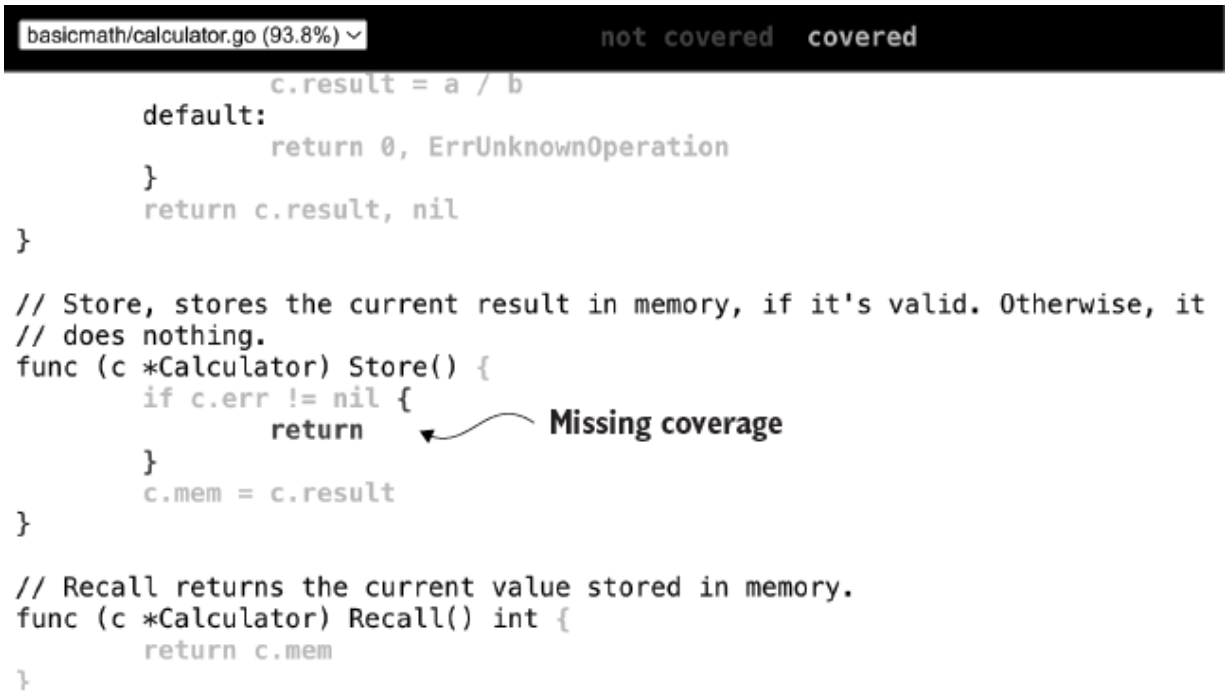


Figure 8.6 Screenshot of the coverage viewer web application

In this case, you can see some missing coverage in the `Store` method for the `Calculator` type, indicating that the error condition (trying to call `Store` after an invalid calculation) was never run during testing. To fix this, you need a test that demonstrates that `Store` does *not* update the value in calculator memory after an invalid result but continues to work afterward.

Listing 8.25 Updating `TestMemory` to test storage error and recovery

```
func TestMemory(t *testing.T) {
        c := basicmath.NewCalculator()

        /*
                OLD TESTS
        */

        // Ensure that the value is not updated when calling store
after an
        // invalid calculation
        _, _ = c.Calculate("div", 0, 0)
        c.Store()
        got = c.Recall()
        if got != 4 {
                t.Errorf("storage should not update value after inv
alid result")
        }

        // Ensure that storage works again after error.
        _, err = c.Calculate("mul", 4, 3) // 12
        if err != nil {
                t.Fatalf("unexpected error: %v", err)
        }
        c.Store()
        got = c.Recall()
        if got != 12 {
                t.Errorf("recall after storage recover - want: 12,
got: %d", got)
        }
}
```

To perform this test, a call to `Calculate` is made on line 10 that is guaranteed to be invalid: divide by zero. This call sets up the failure condition for `Store`, so testing the error or return value is unnecessary, since these are already tested elsewhere. Then `Store` and `Recall` are run again. At this point in the test, the last successful `Store` was a value of `4` (see listing 8.5), so line 13 checks to make sure that this hasn't changed. If it has, this would be a test failure, since it would mean that `Store` is ignoring the error.

Finally, lines 18-26 run a final calculation to make sure that `Store` continues to work correctly.

This brings the coverage up to 100%, but the results from `go test` reveal that something more is going on:

```
$ go test -cover
--- FAIL: TestMemory (0.00s)
    calculator_test.go:102: Calculate - div, 12, 3
    calculator_test.go:137: recall after storage err - want: 12, go
t: 4
FAIL
coverage: 100.0% of statements
exit status 1
FAIL    basicmath  0.240s
```

It looks as if the very last test has failed. It turns out that `Store` does not start working again after an error. The quest for coverage, it seems, has helped uncover a bug!

Inspection of the `Calculate` method reveals the problem: the internal error value is never reset to `nil` after a successful calculation.

Listing 8.26 Fixing `Calculate` to reset `Calculator.err`

```
func (c *Calculator) Calculate(operation string, a, b int) (int, er
ror) {
        if operation == "div" && b == 0 {
                c.err = ErrDivByZero
                c.result = 0
                return 0, ErrDivByZero
        }
        c.err = nil // ensure the internal error value is reset
        switch operation {
        case "add":
                c.result = a + b
        case "sub":
                c.result = a - b
        case "mul":
                c.result = a * b
        case "div":
                c.result = a / b
        default:#1
                c.err = ErrUnknownOperation
                return 0, ErrUnknownOperation
        }
        return c.result, nil
}
```

#1 The default case will ensure the error is set again, if necessary.

A simple change on line 7 is all it takes, and coverage is at 100%, with all tests passing.

```
$ go test -cover
PASS
coverage: 100.0% of statements
ok      goinaction.book/ch09/basicmath  0.210s
```

8.6 Simulating dependencies with an HTTP server

The standard library provides additional testing packages for more advanced use cases. One commonly used library is

the `httptest` package, which, as the name suggests, provides tooling to test HTTP endpoints. HTTP requests and responses can be complicated, and capturing result data can be tricky. This package provides tools to isolate specific methods and also allows us to create a temporary test server. Let's create an example endpoint that we can then test with this package.

When you open a web page, your browser sends an HTTP request. This request has a few different elements. First, the URL directs the message to the proper server. The URL is a path that a server or service uses to route the message to a specific location (for a file) or a function (for a service). Along with the URL is a request type, which is a standard string such as `GET`, `POST`, or `PUT`. These request types are interpreted by the consuming service to determine what the response should be. The request can optionally send data in the *body* of the request. This data is also interpreted by the service. In some web applications, this data is encoded as JSON (JavaScript Object Notation) or XML (Extensible Markup Language). Finally, supplemental information is added to the request in the form of a *header*, which is a map of string values that the service can use to provide additional context for the request.

That's a lot of information, but it is helpful in our example. In Go, requests are sent to *handlers*, which are special Go types that are functions with two parameters: `http.ResponseWriter` and `*http.Request`. A server mounts a handler, and the request is passed into that function, where logic can be implemented to process the data and write the results to the response. A status code is set to let the consuming service know exactly what happened. You may be familiar with the values `404` or `500` from browsing the internet; they indicate a missing web page (not found) or an error, respectively.

To better understand some of these concepts, let's create a handler. First, create a file called `handler.go`. In it, we will create a simple `GET` handler that writes the value "hello" back to the caller.

Listing 8.27 handler.go: Basic handlers to test

```
func HelloGet(w http.ResponseWriter, req *http.Request) { #1
        w.Write([]byte("hello")) #2
}
```

#1 Handlers are considered a type in Go's http package that are a function that accepts a ResponseWriter and Request as parameters.
#2 The writer allows for byte data to be written as part of the response.

If you were to run this on a server and navigate to a web page, you would see the clear text "hello". This is because, by default, a web browser request uses `GET`. To verify that this works, we will write a test using Go's special `httptest` library, which is included as part of the standard library. As mentioned earlier, it provides several tools to request and capture data. In this instance, we will be able to create a special recorder to help us verify the response.

Listing 8.28 handler_test.go: Basic handlers test

```
func TestGet(t *testing.T) {
        // Arrange
        handler := http.HandlerFunc(hello.HelloGet)  #1
        rr := httptest.NewRecorder() #2
        req, err := http.NewRequest("GET", "", nil) #3
        if err != nil {
                t.Errorf("unable to create request %s", err)
                t.FailNow()
        }

        // Act
        handler.ServeHTTP(rr, req) #4

        // Assert
        if rr.Code != http.StatusOK { #5
                t.Errorf(`expected status %d but received %d`,
                        http.StatusOK, rr.Code)
        }

        respBody, err := io.ReadAll(rr.Body) #6
        if err != nil {
                t.Errorf("unable to read response %s", err)
                t.FailNow()
        }
        if string(respBody) != "hello" { #7
                t.Errorf("expected 'hello' received '%s'", string(r
espBody))
                t.Fail()
        }
}
```

#1 Create a function to serve the HTTP request to your handler.
#2 Create a special ResponseWriter to capture your data from your handler as part of your test.
#3 Create a request to send to your handler.
#4 Pass your data through the serve function to simulate a request to your handler and record the results.
#5 Verify the status code of your request.
#6 Read the contents of the body of the response.
#7 Verify the results of the body content.

In this example, we create a handler based on our function and then pass a sample request and a recorder to capture the resulting data. We can then compare the results and confirm that we receive the "hello" message as well as an OK status code.

Since we've established the first pattern of fetching data from a handler, we should test that we are handling data as part of a request. To do that, we need to read the body of the request message as part of a `POST` request.

Listing 8.29 handler.go: `POST` handler to test

```
func HelloPost(w http.ResponseWriter, req *http.Request) {
        body, err := io.ReadAll(req.Body) #1
        if err != nil {
                w.WriteHeader(http.StatusInternalServerError) #2
                return
        }
        hello := append([]byte("hello "), body...)
        w.WriteHeader(http.StatusCreated)
        w.Write(hello)
}
```

#1 Read the content of the request.
#2 Write special header codes depending on if the handler errors.

Here, you can see that we read in all the data from the body of the request. If we experience an error, we can respond with the proper error code; otherwise, we will append whatever the user sends us as part of a "hello" text. To verify this, we will capture the results once again but send a different request message. This test will be similar to our other test, so it's a perfect opportunity to set up a table test, where we can pass in multiple handlers and verify their responses.

Listing 8.30 handler_test.go: Table test for handlers

```
func TestHandler(t *testing.T) {
        // Arrange
        tt := []struct { #1
                Handler          http.HandlerFunc
                Action           string
                Body             []byte
                StatusCode       int
                ExpectedResponse string
        }{
                {
                        Handler:          hello.HelloGet,
                        Action:           "GET",
                        StatusCode:       http.StatusOK,
                        Body:             nil,
                        ExpectedResponse: "hello",
                },
                {
                        Handler:          hello.HelloPost,
                        Action:           "POST",
                        StatusCode:       http.StatusCreated,
                        Body:             []byte("Go in Action"),
                        ExpectedResponse: "hello Go in Action",
                },
        }

        for _, test := range tt { #2
                handler := http.HandlerFunc(test.Handler)
                rr := httptest.NewRecorder()

                req, err := http.NewRequest(
                test.Action, "", bytes.NewBuffer(test.Body))
                if err != nil {
                        t.Errorf("unable to create request %s", er
r)
                        t.FailNow()
                }

                // Act
                handler.ServeHTTP(rr, req)

                // Assert
```

```
                if rr.Code != test.StatusCode {
                        t.Errorf(`expected status %d but received %
d`,
                                test.StatusCode, rr.Code)
                }

                respBody, err := io.ReadAll(rr.Body)
                if err != nil {
                        t.Errorf("unable to read response %s", err)
                        t.FailNow()
                }
                if string(respBody) != test.ExpectedResponse {
                        t.Errorf("expected '%s' received '%s'",
                        test.ExpectedResponse, string(respBody))
                        t.Fail()
                }
        }

}
```

#1 Create a table test to go through our handlers to verify results.
#2 The logic of testing the results is similar to our previous test.

When you navigate to a website, you use a URL. After the URL is a *path*, which is separated by slashes. Within a web server, this is known as a *route*. A handler function is attached to a route when it is mounted within a server. When a server needs multiple routes, it creates a *mux*, which keeps track of all paths and routes requests to them accordingly. Let’s create a handler like this that can be attached to a running server.

Listing 8.31 handler.go: Handler with a route

```
func HelloHandler(mux *http.ServeMux) { #1
        mux.HandleFunc("/hello", func(w http.ResponseWriter, req *h
ttp.Request) {
                switch req.Method { #2
                case http.MethodGet:
                        HelloGet(w, req)
                case http.MethodPost:
                        HelloPost(w, req)
                default: #3
                        w.WriteHeader(http.StatusNotFound)
                        w.Write([]byte("invalid hello method"))
                }
        })
}
```

#1 A mux allows us to mount a particular path to our handlers.
#2 We go through the methods to determine which handler to call.
#3 If we don't support a method we should return a 404 or Not found error code.

Here, we mount our other handlers for the `GET` and `POST` methods. If we don't support the path, we return a not found error with a custom message. To verify that it works, we can use the `httptest` package to start a test server on a random port and then make calls against it.

Listing 8.32 handler_test.go: Test with a route

```
func TestServer(t *testing.T) {
        // Arrange
        tt := []struct { #1
                Path             string
                Action           string
                Body             []byte
                StatusCode       int
                ExpectedResponse string
        }{
                {
                        Path:             "/hello",
                        Action:           "GET",
                        StatusCode:       http.StatusOK,
                        Body:             nil,
                        ExpectedResponse: "hello",
                },
                {
                        Path:             "/hello",
                        Action:           "POST",
                        StatusCode:       http.StatusCreated,
                        Body:             []byte("Go in Action"),
                        ExpectedResponse: "hello Go in Action",
                },
                {
                        Path:             "/hello",
                        Action:           "PUT",
                        StatusCode:       http.StatusNotFound,
                        Body:             []byte("bad request"),
                        ExpectedResponse: "invalid hello method",
                },
        }
        mux := http.NewServeMux() #2
        hello.HelloHandler(mux) #3

        srvr := httptest.NewServer(mux) #4

        defer srvr.Close()
        for _, test := range tt {

                path, err := url.JoinPath(srvr.URL, test.Path) #5
                if err != nil {
```

```
                        t.Errorf("unable to create request %s", er
r)
                        t.FailNow()
                }
                req, err := http.NewRequest(test.Action, path,
                bytes.NewBuffer(test.Body))
                if err != nil {
                        t.Errorf("unable to create request %s", er
r)
                        t.FailNow()
                }
                // Act
                resp, err := srvr.Client().Do(req) #6
                if err != nil {
                        t.Errorf("unable to create request %s", er
r)
                        t.FailNow()
                }

                // Assert
                if resp.StatusCode != test.StatusCode {
                        t.Errorf(`expected status %d but received %
d`,
                                test.StatusCode, resp.StatusCode)
                }

                respBody, err := io.ReadAll(resp.Body)
                if err != nil {
                        t.Errorf("unable to read response %s", err)
                        t.FailNow()
                }
                if string(respBody) != test.ExpectedResponse {
                        t.Errorf("expected '%s' received '%s'", tes
t.ExpectedResponse,
                        string(respBody))
                        t.Fail()
                }
        }

}
```

#1 Again create a table test, this time with URLs instead of handlers.
#2 Create a new mux to mount your handlers.

#3 Add your paths to the mux.
#4 Use the mux to create a test server.
#5 Create the request URL based on the test server URL and the path you want to test.
#6 Call the endpoint and verify similar to the tests above.

8.7 Beyond unit tests

Testing applications goes beyond just verifying that a given function returns the results you expect. The unit tests we have been writing are the most common form of tests included in a codebase. But as your application grows, you may want to add more checks. You may wonder whether one algorithm performs better than another. Does the function you wrote cover cases that you may not have thought of but that could pose a serious problem if a bug is discovered? Go's built-in testing library can help you with both types of verification.

Let's look at how we can use benchmark and fuzz testing to add these types of checks.

8.7.1 Benchmark testing

Measuring performance is extremely important as your application becomes more mature. Poor performance can lead to slower response times, increased processing power, and possible system crashes. Yet, benchmark testing is often left out as part of a testing suite because it can sometimes be difficult to set up. However, Go's existence is partially due to the need to have a performant language for modern application development, so the language has a built-in benchmark testing tool we can use.

Benchmark testing puts an application under extreme load to see how it performs. It is common to write tests that have consistent outcomes and expectations, much like unit

tests. The difference with benchmark testing is that we put the function we are testing under extreme load, making many calls with many values to determine an average processing time. This is often done as a comparison with another operation, so the parameters should be the same while the underlying function may be different.

Let's demonstrate with a simple example in which a function creates a random string of a given length and returns it.

Listing 8.33 xstrings.go: Simple string generation method

```
import (
        "math/rand"
        "strings"
)

var alphabetArray = []byte("abcdefghijklmnopqrstuvwxyz") #1

func RandomString(n int) string {
        b := make([]byte, 0, n) #2
        for range n {
                idx := rand.Int31() % 26 #3
                b = append(b, alphabetArray[idx]) #4
        }
        return string(b) #5
}
```

#1 Create an alphabet array to draw random characters.
#2 Initialize an array for the result.
#3 Get the index of a random character.
#4 Add character to response array.
#5 Cast array as a string and return.

As you can imagine, there are several ways to improve this code, but we will start with this naive implementation to get a baseline for our benchmark tests. To test its performance, we will create a test, but instead of using `testing.T` as a parameter, we will use `testing.B`, where the B stands for benchmark.

Listing 8.34 xstrings_test.go: Simple string-generation method

```
import (
        "strings"
        "testing"

        xstring "github.com/go-in-action/chpt_8/benchmark/xstring"
)

func BenchmarkRandomString(b *testing.B) { #1
        for i := range b.N { #2
                res := xstring.RandomString(i) #3
                if len(res) != i {
                        b.Error("result has the wrong length")
                }
        }
}
```

#1 Benchmark tests should begin with the word Benchmark and accept the testing.B parameter.
#2 Use the parameter to create a number to use for the test, in this case, it will be the length of the string.
#3 Test the underlying method.

Here, we verify that we get a valid string back each time. Additionally, the benchmark test keeps track of the runtime and efficiency of each iteration. Over time, the values sent to the service will be collated and averaged as part of a result that you can compare against another service.

To do this, we will run `go test -bench=.`, which runs our benchmark tests. The results will look like this:

```
go test -bench=.
goos: darwin
goarch: arm64
pkg: github.com/holmes89/go-in-action/chpt_8/benchmark
cpu: Apple M3 Pro
BenchmarkRandomString-11                   30609            119318
ns/op
```

This gives us some additional information, such as the processor and operating system type, all of which affect the results. From the name of the test, you can see how many cores were used; in this example, there were 11 cores. The `30609` is the number of iterations used to create the statistics, while the last number shows the nanoseconds per operation it took to run.

Now let's create a separate method to compare the results against. This time, we will optimize by using the `strings.Builder` object. This is part of the standard library, so we can see if it works better than our implementation using an array.

Listing 8.35 xstrings.go: String builder generation method

```
import (
        "math/rand"
        "strings"
)

...

func RandomStringBuilder(n int) string {
        builder := strings.Builder{} #1
        for range n {
                idx := rand.Int31() % 26
                builder.WriteRune(rune(alphabetArray[idx]))
        }
        return builder.String() #2
}
```

#1 The strings builder is a struct that allows strings to be built using other strings, runes, or bytes.
#2 Once all parts are written to the builder we can return the string.

We can then add a test to generate the results, which look similar to our previous test.

Listing 8.36 xstrings_test.go: Simple string test method

```
func BenchmarkRandomStringBuilder(b *testing.B) {
        for i := range b.N {
                res := xstring.RandomStringBuilder(i) #1
                if len(res) != i {
                        b.Error("result is empty")
                }
        }
}
```

#1 New string method to use

Now we can compare the results from the two methods:

```
go test -bench=.
goos: darwin
goarch: arm64
pkg: github.com/holmes89/go-in-action/chpt_8/benchmark
cpu: Apple M3 Pro
BenchmarkRandomString-11                    30013            116375
ns/op
BenchmarkRandomStringBuilder-11             24721            120046
ns/op
```

We can see that our implementation was faster overall in terms of operational costs. We can continue to work with these methods to find other ways to improve them, now with measurable data to show which direction we are moving in. But benchmarking only allows you to see the load on the system given a set of parameters. Bugs often lie at the edges of our code, outside our normal set of parameters and in areas we haven't explored yet. To find those sorts of bugs, we will need to write fuzz tests.

8.7.2 Fuzz testing

"You can't think of everything!" This is a common saying that people use to help each other feel better about mistakes caused by overlooking or missing a minor detail. It

also applies to development. We can't always think of every scenario that might be put into our method or application. Some of the best testers can inject random characters, empty spaces, or large numbers to try to break our systems, but something will eventually be missed. But we're programmers, so it may be easier to have our computers create the data and then validate the results. Given a set of valid test parameters, imagine an application that uses that set, expands upon it in different ways, and then runs the test. This is exactly the purpose of fuzz testing.

Go valued this type of testing enough to add it to the standard library, just like benchmark testing. To write a fuzz test, we use a different parameter for our test—we will use `testing.F`.

Before we write our test, we first need to create a method. Continuing with our `xstring` example, we will make our own `Contains` function, which will check whether a string contains a given rune.

Listing 8.37 xstrings.go: Contains functions for strings

```
func Contains(text string, char rune) bool {
        for _, c := range []byte(text) { #1
                if rune(c) == char { #2
                        return true #3
                }
        }

        return false
}
```

#1 Loop through each character in the string.
#2 Verify the characters match.
#3 Exit loop early and return true.

If we were going to write a unit test, we would consider various cases, such as empty strings, numbers, or even

UTF-8 characters, but we wouldn’t think of every edge case. With a fuzz test, we should be able to get these extra cases for free. To write our fuzz test, we will prefix the test name with `Fuzz` and use the `testing.F` library as a parameter. Fuzzing works with a *corpus*, a collection of examples, to base additional tests on. Let’s create our `FuzzContains` function in our test file.

Listing 8.38 xstrings_test.go: Fuzz contains function

```
func FuzzFindInString(f *testing.F) {
        cases := []struct { #1
                word string
                char rune
        }{
                {
                        word: "hello world",
                        char: 'e',
                },
                {
                        word: "goodbye",
                        char: 'e',
                },
                {
                        word: "",
                        char: 'o',
                },
                {
                        word: "a longer example",
                        char: 'z',
                },
        }
        for _, ca := range cases { #2
                f.Add(ca.word, ca.char)
        }
        f.Fuzz(func(t *testing.T, in string, r rune) { #3
                contains := xstring.Contains(in, r)
                if strings.Contains(in, string(r)) != contains { #4
                        t.Errorf("failed to find '%b' in %s", r, i
n)
                }
        })
}
```

#1 Create a list of test cases.
#2 Add each case to the Fuzz corpus.
#3 The Fuzz function provides an input string and rune for us to use.
#4 Verify results against another method.

To run this test, we will use a flag similar to the one we used for benchmarks. This time, we will run the following

test command and watch the output:

```
go test -fuzz=. -fuzztime=30s
fuzz: elapsed: 0s, gathering baseline coverage: 0/4 completed
fuzz: elapsed: 0s, gathering baseline coverage: 4/4 completed,
 now fuzzing with 11 workers
fuzz: minimizing 467-byte failing input file
fuzz: elapsed: 0s, minimizing
--- FAIL: FuzzFindInString (0.13s)
    --- FAIL: FuzzFindInString (0.00s)
        methods_test.go:56: failed to find '10000110' in ◆

    Failing input written to testdata/fuzz/FuzzFindInString/465a20c
97b1ec3b2
    To re-run:
    go test -run=FuzzFindInString/465a20c97b1ec3b2
FAIL
exit status 1
```

It looks like we found a bug! And it's the result of a value we would not have originally tested. What did we do wrong in the previous test? Looking at the code, we shouldn't have converted the string into a byte array. Instead, we should have just iterated over the string and grabbed the runes to compare the values.

Listing 8.39 xstrings.go: Fixing the bug fuzzing found

```
func Contains(text string, char rune) bool {
        for _, c := range text { #1
                if c == char {
                        return true
                }
        }

        return false
}
```

#1 Ranging over a string results in a list of runes so no need to cast.

Rerunning our tests, we can see that things are now working as expected. Fuzzing can go on forever if you don't set a `-fuzztime` flag, just in case you want to test forever.

```
go test -fuzz=. -fuzztime=30s
fuzz: elapsed: 0s, gathering baseline coverage: 0/5 completed
fuzz: elapsed: 0s, gathering baseline coverage: 5/5 completed,
now fuzzing with 11 workers
fuzz: elapsed: 3s, execs: 1148614 (382751/sec), new interesting: 0
(total: 5)
fuzz: elapsed: 6s, execs: 2380058 (410504/sec), new interesting: 0
(total: 5)
fuzz: elapsed: 9s, execs: 3590979 (403737/sec), new interesting: 0
(total: 5)
fuzz: elapsed: 12s, execs: 4744436 (384406/sec), new interesting: 0
(total: 5)
fuzz: elapsed: 15s, execs: 5931985 (395846/sec), new interesting: 0
(total: 5)
fuzz: elapsed: 18s, execs: 7015194 (361116/sec), new interesting: 0
(total: 5)
fuzz: elapsed: 21s, execs: 8117228 (367257/sec), new interesting: 0
(total: 5)
fuzz: elapsed: 24s, execs: 9320427 (401176/sec), new interesting: 0
(total: 5)
fuzz: elapsed: 27s, execs: 10571569 (417032/sec), new interesting:
0 (total: 5)
fuzz: elapsed: 30s, execs: 11827267 (418555/sec), new interesting:
0 (total: 5)
fuzz: elapsed: 30s, execs: 11827267 (0/sec), new interesting: 0 (to
tal: 5)
PASS
ok      github.com/holmes89/go-in-action/chpt_8/benchmark       30.
745s
```

This is just a taste of what you can do with Go's testing packages. Go is part of the next generation of languages that ship with testing in mind, and it provides many tools right out of the box. Others developers have extended these libraries to handle many more scenarios and create more

testing tools. With all of this at your disposal, there's no excuse not to write tests for your code!

Summary

- Testing in Go uses functions and test files so your code can test itself.
- Go comes packaged with the powerful `go test` test runner, which performs many useful functions in addition to running your tests.
- If you choose to follow a test-driven development (TDD) workflow, tests can act as your design specification and can guide development instead of merely checking existing code.
- Techniques such as table-driven tests can simplify repetitive tasks.
- Test coverage is an important tool that helps you check for gaps in your testing and find new bugs.
- In addition to unit testing, the Go test tooling can run benchmark and fuzz testing to test the performance and edge cases of your function.

9 Concurrency

This chapter covers

- Go’s concurrency model
- Using internal libraries to manage concurrent tasks
- Creating channels for asynchronous communication
- Applying concurrent patterns

Imagine you have a list of chores: bake a chicken, do the laundry, and send an email. You might work on all three at once. While the chicken bakes and the laundry runs, you can send your email. In this scenario, multiple tasks are happening *concurrently*. When two or more tasks are being executed at the same time, such as the oven and washing machine running simultaneously, that’s called *parallelism*.

It’s common to confuse concurrency and parallelism. *Concurrency* is about managing multiple tasks at once, allowing progress on several activities by switching between them. *Parallelism* is about actually performing multiple tasks at the same time, typically on separate processors or cores.

Modern applications rely on concurrency to remain responsive, such as when you receive a call while browsing the web or sending an email. Without concurrency, programs would have to finish one task before starting another, making them slow and unresponsive.

What makes concurrency special in Go is its simplicity and power. Go was designed with concurrency as a core feature. Instead of requiring developers to manage threads directly,

Go provides lightweight tools: *goroutines* for running functions concurrently and *channels* for safe communication between them. The Go runtime handles the complex details of scheduling and execution, making it much easier to write efficient, concurrent programs.

In this chapter, we'll explore Go's approach to concurrency and look at some common patterns and best practices for using concurrency to improve performance and scalability.

9.1 When to use concurrency

Knowing when to use something is almost as important as knowing how to use it. When you should use concurrency is an important question because concurrency adds complexity to your code. It is best to have a simple solution before moving on to something more complex. As you write simple programs, you will inevitably encounter bugs. Imagine trying to investigate bugs concurrently!

You should use concurrency in Go when your program needs to perform multiple independent tasks. Handling multiple network requests, processing files, and performing background computations are all scenarios where concurrency can help your program do more at once. It is also useful when you want to improve responsiveness. If your application has a user interface, you can keep it active and responsive to user input while performing long-running operations in the background. Finally, if you need to use multiple CPU cores for CPU-bound tasks, concurrency allows you to achieve better performance through parallelism. By running tasks in parallel, you can make full use of the available hardware.

Concurrency is beneficial when your application waits for I/O operations, such as disk or network access. While

waiting for these operations to complete, your program can make progress on other work, improving overall efficiency. To demonstrate this, let's create a small application that fetches data from a news feed.

> **NOTE** RSS (Really Simple Syndication) is a standardized web feed format used to publish frequently updated information, such as news headlines, blog posts, and podcasts. An RSS feed provides a structured XML document that contains summaries or the full content of recent updates from a website. Users and applications can subscribe to RSS feeds to automatically receive the latest content without manually visiting each site. This capability makes RSS a popular choice for aggregating news, tracking updates, and building custom readers or notification systems.

We are going to build a simple RSS reader application that fetches news articles from sources like the New York Times and BBC. The goal is to demonstrate Go's concurrency features by retrieving multiple RSS feeds, parsing their articles, and saving them in memory for later access. This scenario is a practical example of handling blocking I/O operations, such as network requests, while also managing shared data safely. Throughout the chapter, we'll incrementally develop this RSS reader, showing how you can to fetch feeds concurrently, store articles, and expose them through an API. Let's start by initializing our project and adding the `gofeed` library, which will help us fetch and parse RSS feeds.

Listing 9.1 bash: Initialize the project

```
go mod init rssreader
go get github.com/holmes89/gofeed
```

Now we can create our application. Create a `main.go` file and open it in your editor.

Listing 9.2 main.go: Initial application creation using synchronous fetching

```
package main

import (
 "fmt"

 "github.com/holmes89/gofeed"
)

func main() {
 fp := gofeed.NewParser() #1
 feed, _ := fp.ParseURL("https://rss.nytimes.com/
 services/xml/rss/nyt/World.xml") #2
 fmt.Println(feed.Title) #3
}
```

#1 Create new parser.
#2 Fetch and parse an RSS feed.
#3 Print the title of the feed.

When the application starts, it makes a request to fetch the data from the specified URL and then waits for the response. During this time, the program is blocked; nothing else can happen until the network call completes. Once the response is received, the data is parsed and the feed title is printed. This is an example of a *synchronous* operation: the steps are performed one after the other, and the program cannot proceed until the current step finishes.

While this approach is simple, it has limitations. If the network is slow or the server is unresponsive, your entire application will be stuck waiting. In real-world applications, especially those that need to remain responsive or handle multiple tasks at once, this blocking behavior is undesirable.

To address this, Go provides a simple way to run code concurrently using the `go` keyword, which creates a *goroutine*, or separate process. By launching a function as a goroutine, you allow it to execute independently of the main program flow. One common pattern is to wrap the synchronous code in an anonymous function (closure) and start it as a goroutine. This enables your application to perform other work while waiting for the network response, improving responsiveness and efficiency.

Listing 9.3 main.go: Fetching data via a goroutine

```
package main

...

func main() {
 fp := gofeed.NewParser()
 go func(url string) { #1
  feed, _ := fp.ParseURL(url) #2
  fmt.Println(feed.Title)
 }("https://rss.nytimes.com/services/xml/rss/nyt/World.xml") #3

}
```

#1 We create a closure and use the go keyword to have the function run in the background. The function wants us to provide a URL string to fetch the data.
#2 We still fetch and parse the URL that is sent.
#3 Pass in our RSS Feed into the closure.

If you run this application, you will find that nothing is printed. The application exits before the goroutine has a chance to fetch and print the feed title. This happens because the `main` function completes immediately after launching the goroutine, causing the program to terminate before the background work finishes. In Go, the `main` function controls the lifetime of the program. When `main()` returns, the program exits, and any running goroutines are abruptly stopped. This means that simply launching a

goroutine with the `go` keyword is not enough; you need to ensure the `main` function waits for the goroutine to complete its work.

To make this work, we need to wait for the routine to run. We can add a short sleep after launching the goroutine to give it time to finish:

```
import (
        "time"
        "fmt"

    "github.com/holmes89/gofeed"
)

func main() {
 fp := gofeed.NewParser()
 go func(url string) {
  feed, _ := fp.ParseURL(url)
  fmt.Println(feed.Title)
 }("https://rss.nytimes.com/services/xml/rss/nyt/World.xml")

 time.Sleep(15 * time.Second) #1
 fmt.Println("stopped.")
}
```

#1 Wait 15 seconds until the main function exits.

But using `time.Sleep` is not a robust solution because it relies on guessing how long the work will take. In production code, you should use a more reliable way to synchronize goroutines and ensure all work is completed before exiting. We will explore those options in future sections.

You don’t always need to use a closure or anonymous function to launch a goroutine. If you already have a function that takes the required arguments, you can simply use the `go` keyword before the function call. Let’s refactor our code by extracting the logic into a new function called

`printFeedTitle(url string)` and then launch it directly as a goroutine.

Listing 9.4 main.go: Concurrency with the go keyword

```
import (
 "fmt"
 "time"

 "github.com/holmes89/gofeed"
)

func printFeedTitle(url string) { #1
 fmt.Println("fetching feed: ", url)
 fp := gofeed.NewParser()
 feed, _ := fp.ParseURL(url)
 fmt.Println(feed.Title)
 fmt.Println("fetched: ", url)
}

func main() {
 go printFeedTitle("https://rss.nytimes.com/services/
 xml/rss/nyt/World.xml") #2
 fmt.Println("waiting....")
 time.Sleep(15 * time.Second)
 fmt.Println("stopped.")
}
```

#1 Create a new function to output the feed title.
#2 Use the new function asynchronously by using the go keyword.

The `go` command tells our process to run in the background. Now we add the timer to *block*, or stop processing for a period of time, so we can see the results come in.

Now suppose we want to fetch another feed. We can do that just by adding another `go` command. By launching multiple goroutines, each responsible for fetching a different RSS feed, we can perform several network requests concurrently. This means our application can start fetching

the next feed without waiting for the previous one to finish, making much better use of time and system resources.

In the following listing, we simply add another `go printFeedTitle(…)` line for each feed URL we want to process. Each goroutine runs independently, and the main function uses `time.Sleep` to give all the goroutines enough time to complete their work.

Listing 9.5 main.go: A second goroutine

```
func main() {
 go printFeedTitle("https://rss.nytimes.com/services/xml/rss/nyt/Wo
rld.xml")
 go printFeedTitle("https://feeds.bbci.co.uk/news/
 rss.xml") #1
 fmt.Println("waiting....")
 time.Sleep(15 * time.Second)
 fmt.Println("stopped.")
}
```

#1 Add a second call that runs concurrently.

If you run this, you should see two sets of messages come through, one for each feed being fetched and printed. This demonstrates the simplest way in Go to perform asynchronous tasks: launching functions as goroutines using the `go` keyword. Each goroutine runs independently, allowing multiple operations to proceed concurrently without blocking the main program.

But just launching goroutines isn’t enough to build robust concurrent programs. There are several important considerations to keep in mind. Synchronization is crucial. As mentioned earlier, using `time.Sleep` is a crude way to wait for goroutines to finish. In real applications, you need reliable synchronization to ensure all work is completed before the program exits.

Although Go makes it easy to start concurrent tasks, building reliable and maintainable concurrent programs requires careful coordination, error handling, and communication. Go's synchronization package provides primitives and patterns that help you manage concurrency effectively.

9.2 Sync package

Go makes it so easy to write concurrent code that you will often find yourself running multiple goroutines that may need some level of coordination or additional handling. The standard library provides the `sync` package to help manage this type of programming by allowing you to synchronize access to shared resources or to coordinate the completion of tasks.

9.2.1 Wait groups

Let's start by looking at `WaitGroup`, which does exactly what it sounds like: it waits for a group of concurrent functions to run to completion. It works by keeping count of the number of processes it's managing, waiting for the counter to become 0, and then moving on. Let's see this in action.

Listing 9.6 main.go: Concurrency with the sync.WaitGroup.Go method

```
import (
 ...
 "sync"
 ...
)

func main() {
 var wg sync.WaitGroup #1
 wg.Go(func() { #2
  printFeedTitle("https://rss.nytimes.com/services/xml/rss/nyt/Worl
d.xml")
 })
 wg.Go(func() {
  printFeedTitle("https://feeds.bbci.co.uk/news/rss.xml")
 })
 fmt.Println("waiting....")
 wg.Wait() #3
 fmt.Println("stopped.")
}
```

#1 Create a WaitGroup to coordinate routines.
#2 Execute a closure within the WaitGroup.
#3 Wait for all routines to finish before proceeding.

Now we don’t need the timer! By using a `WaitGroup`, we can reliably wait for all our goroutines to finish before the main function exits, ensuring that each feed is fetched and printed as expected. The `WaitGroup` acts as a synchronization primitive, tracking the number of active goroutines and blocking until all have completed. This approach is more robust and idiomatic because it doesn’t rely on guessing how long the work will take, and it guarantees that all concurrent tasks are finished before proceeding.

To further improve our code structure and make it reusable, let’s refactor this logic into its own function, which can handle a list of feed URLs concurrently.

Listing 9.7 main.go: Concurrency with the go keyword with a WaitGroup

```
import (
 ...
 "sync"
 ...
)

...

func fetchFeeds(urls []string) {
 var wg sync.WaitGroup
 for _, url := range urls {
  wg.Add(1) #1
  go func(url string) {
   defer wg.Done() #2
   printFeedTitle(url)
  }(url)
 }
 fmt.Println("waiting....")
 wg.Wait() #3
}

func main() {
 fetchFeeds([]string{
  "https://rss.nytimes.com/services/xml/rss/nyt/World.xml",
  "https://feeds.bbci.co.uk/news/rss.xml",
 })
 fmt.Println("stopped.")
}
```

#1 WaitGroups can have a count associated with the amount of work they will need to do.
#2 A Done method allows the WaitGroup to decrement its counter to know when to stop waiting.
#3 Block until everything has resolved.

In listing 9.7, we use a `WaitGroup` with anonymous functions to coordinate our goroutines. For each feed, we increment the `WaitGroup` counter before launching a goroutine, and inside each goroutine, we use `defer wg.Done()` to ensure the counter is decremented when the goroutine finishes. This

pattern guarantees that the `main` function waits for all concurrent operations to complete before proceeding. The `Go` method on the `WaitGroup` is a newer addition to the structure, so you will often see the `Add` pattern in legacy codebases, which is why we demonstrated both approaches here.

So far, we've ignored error handling for simplicity. In concurrent programs, tracking and handling errors from multiple goroutines can be tricky. If one goroutine fails, how do you capture and report that error while still waiting for all the other goroutines to finish? The `WaitGroup` itself doesn't provide a way to collect errors.

9.2.2 Error Groups

This is where the `errgroup` package comes in. An `errgroup.Group` works like a `WaitGroup`, but it also collects errors from goroutines. If any goroutine returns an error, the group captures it, and you can check the result after all goroutines have finished. This is especially useful when you want to stop processing or handle failures gracefully if something goes wrong in any concurrent task.

To use `errgroup`, replace your `WaitGroup` with an `errgroup.Group`. Instead of launching goroutines manually, use the `Go` method provided by the group, as we did in the first `WaitGroup` example. Each function passed to `Go` should return an error. After launching all the goroutines, call `Wait()` on the group, which blocks until all functions have returned. If any function returns a non-nil error, `Wait()` returns the first error encountered.

Let's update our code to use `errgroup` for better error handling and synchronization.

Listing 9.8 main.go: Concurrency with the go keyword and errorgroup

```
import (
 ...
 "sync/errgroup"
 ...
)

...

func fetchFeed(url string) (gofeed.Feed, error) { #1
 fmt.Println("fetching feed: ", url)
 fp := gofeed.NewParser()
 feed, err := fp.ParseURL(url)
 fmt.Println("fetched: ", url)
 return feed, err
}


func fetchFeeds(urls []string) {
 eg := new(errgroup.Group) #2
 for _, url := range urls {
  eg.Go(func() error { #3
   feed, err := fetchFeed(url)
   if err != nil {
    return err
   }
   fmt.Println(feed.Title)
   return nil
  })
 }
 fmt.Println("waiting....")
 err := eg.Wait() #4
 if err != nil {
  fmt.Println("not all urls were fetched", err)
 }
}

func main() {
 ...
}
```

#1 Change the function to return errors.
#2 Use errgroup to manage goroutines and error handling.

#3 Closure must return an error.
#4 Wait returns an error if any goroutine failed.

By using `errgroup`, we can efficiently wait for all feed-fetching goroutines to complete and handle any errors that occur. This approach ensures that our program doesn't exit prematurely and that errors are properly reported, all while using Go's concurrency model for efficient parallel processing. As you build more complex concurrent applications, tools like `errgroup` become invaluable for managing synchronization and error handling in a clean, idiomatic way.

9.2.3 Mutexes

When multiple goroutines access shared data without proper coordination, you can encounter a *race condition*, a situation where the outcome depends on the unpredictable timing of goroutine execution. This can lead to data corruption, inconsistent results, or hard-to-find bugs. To prevent these problems, Go's `sync` package provides synchronization primitives, including the `Mutex`, which acts as a lock to ensure only one goroutine can access a critical section of code at a time. A *mutex*, which is short for mutual exclusion, is a mechanism that allows only one process to use a resource or object.

Let's see how a mutex can be used by building a simple data store for our articles. We'll design it to store each article only once and to maintain a date-sorted list. We'll start by defining the structure and then add the logic for safe concurrent access.

Listing 9.9 main.go: Creating a data store

```
type ArticleStore struct {
    mu       sync.RWMutex #1
    cache   map[string]any #2
    sorted   []*gofeed.Item
}

func NewArticleStore() *ArticleStore{
 return &ArticleStore{
  cache: make(map[string]any),
 }
}

func (s *ArticleStore) AddArticles(items []*gofeed.Item) {
    s.mu.Lock() #3
    defer s.mu.Unlock() #4
}

func (s *ArticleStore) GetRecent(n int) []*gofeed.Item {
    s.mu.RLock() #5
    defer s.mu.RUnlock() #6
}
```

#1 Use a regular mutex (mu.Lock/Unlock) to protect writes and modifications to the store.
#2 This cache will be used to check if an article has already been added.
#3 Lock the store before adding articles to ensure exclusive access.
#4 Unlock the store after adding articles to allow other goroutines to proceed.
#5 Use a read lock (mu.RLock/RUnlock) to allow concurrent reads while preventing writes.
#6 Unlock the store after reading articles to release the read lock.

Notice that there are two types of lock methods: `Lock`/`Unlock` and `RLock`/`RUnlock`. A `Mutex` (`Lock`/`Unlock`) ensures exclusive access for one goroutine and is typically used when you're modifying shared data. An `RWMutex` (`RLock`/`RUnlock`) allows multiple goroutines to read the data concurrently but still ensures exclusive access when writing. Use `RLock`/`RUnlock` for read-only operations to improve performance, and use `Lock`/`Unlock` when making changes to shared resources.

Let’s now implement the logic for storing data to and retrieving it from the internal map.

Listing 9.10 main.go: Creating a data store with write locks

```
func (s *ArticleStore) AddArticle(item *gofeed.Item) {
    s.mu.Lock() #1
    defer s.mu.Unlock() #2
 if _, ok := s.cache[item.Link] { #3
  return
 }

 s.cache[item.Link] = nil #4
 s.sorted = append(s.sorted, item)
 slices.Sort(s.sorted) #5
}

func (s *ArticleStore) AddArticles(items []*gofeed.Item) {
    s.mu.Lock() #6
    defer s.mu.Unlock() #7
 var hasNewArticles bool #8
 for _, item := range items { #9
  if _, ok := s.cache[item.Link] { #10
   continue
  }
  hasNewArticles = true
  s.cache[item.Link] = nil
  s.sorted = append(s.sorted, item)
 }
 if hasNewArticles { #11
  slices.Sort(s.sorted)
 }
}
```

#1 Lock the mutex before modifying it.
#2 Ensure the lock is released after the function returns.
#3 Check if the article already exists in the cache.
#4 Add the article to the cache to prevent duplicates.
#5 Sort the articles after adding a new one.
#6 Lock the mutex before batch modification.
#7 Ensure the lock is released after the function returns.
#8 Track if any new articles were added.
#9 Iterate over the incoming articles.
#10 Skip articles already in the cache.
#11 Only sort if new articles were added.

The preceding listing demonstrates the use of a write lock. By acquiring a write lock, we ensure exclusive access to the underlying data, preventing any other goroutine from reading or writing while modifications are in progress. This guarantees data consistency during updates.

Next, we'll see how to use a read lock, which allows multiple goroutines to read the data concurrently, as long as no write lock is held.

Listing 9.11 main.go: Using read locks

```
func (s *ArticleStore) GetRecent(n int) []*gofeed.Item {
    s.mu.RLock() #1
    defer s.mu.RUnlock() #2

 if n > len(s.sorted) { #3
  return s.sorted
 }
 return s.sorted[:n] #4
}
```

#1 Create a read lock.
#2 Ensure the lock is released.
#3 If the slice is smaller than the total requested, return the whole list.
#4 Truncate to the requested size.

To wrap up, let's demonstrate how the `ArticleStore` can be used in your application to safely store and retrieve articles concurrently. Each goroutine can call `AddArticles` to store the fetched articles, and you can use `GetRecent` to retrieve the most recent articles for display or further processing.

Here's a simple example of how you might use the store in your main application logic:

```
func main() {
 store := NewArticleStore()
 var wg sync.WaitGroup
 feeds := []string{
  "https://rss.nytimes.com/services/xml/rss/nyt/World.xml",
  "https://feeds.bbci.co.uk/news/rss.xml",
 }

 for _, url := range feeds {
  wg.Add(1)
  go func(feedURL string) {
   defer wg.Done()
   fp := gofeed.NewParser()
   feed, err := fp.ParseURL(feedURL)
   if err != nil {
    fmt.Println("error fetching:", feedURL, err)
    return
   }
   store.AddArticles(feed.Items)
  }(url)
 }

 wg.Wait()

 // Retrieve and print the 5 most recent articles
 recent := store.GetRecent(5)
 for _, article := range recent {
  fmt.Println(article.Title, "-", article.Link)
 }
}
```

This example launches a goroutine for each feed, fetches the articles, and adds them to the shared `ArticleStore`. The store uses mutexes to ensure thread-safe access. After all the goroutines finish, the `main` function retrieves and prints the most recent articles.

By encapsulating your shared data and protecting it with synchronization primitives, you can safely use the store in concurrent scenarios, avoiding race conditions and ensuring data consistency. Although we have been able to manage

the flow of data asynchronously by using various locks, we still want control over other aspects of our functions by providing additional context to our data.

9.3 Adding some context

One final way we can ensure we have control over our structures before we send them out to the concurrency sea is by providing them with additional information for a downstream function. We may want to pass information like a timeout or metadata to the process so it knows how to act. Although we haven't explicitly linked concurrent processes together yet, we are relying on our HTTP client to act on our behalf, and it may need that information. This is why the standard library has the `context` package and `Context` interface.

As our implementation stands, if we can connect to the RSS server but no data comes through, we could end up waiting forever. We will change our implementation for fetching a feed to eliminate this potential problem.

Listing 9.12 main.go: Adding contexts

```
import (
 ...
 "sync/errgroup"
 ...
)

...

func fetchFeed(ctx context.Context, url string) (
   gofeed.Feed, error) { #1
 fmt.Println("fetching feed: ", url)
 fp := gofeed.NewParser()
 feed, err := fp.ParseURLWithContext(ctx, url) #2
 fmt.Println("fetched: ", url)
 return feed, err
}


func fetchFeeds(ctx context.Context, urls []string) { #3
 eg, ectx := errgroup.WithContext(ctx) #4
 for _, url := range urls {
  eg.Go(func() error {
   feed, err := fetchFeed(ectx, url) #5
...
  })
 }
 eg.Wait()
...
}

func main() {
 ctx, cancel := context.WithTimeout(
   context.Background(), 60*time.Second) #6
 defer cancel() #7
 fetchFeeds(ctx, []string{
  "https://rss.nytimes.com/services/xml/rss/nyt/World.xml",
  "https://feeds.bbci.co.uk/news/rss.xml",
 })
 fmt.Println("stopped.")

}
```

#1 Pass context into the fetch function.
#2 Use the context-based call to leverage cancellation.
#3 Modify the function to pass the context.
#4 Create a new context for errgroup that appends its own cancellation policy.
#5 Pass in new context.
#6 Create initial context with a timeout.
#7 Call cancellation function before returning.

Now that we've added contexts, let's clarify how they flow through our system. When `main` starts, it creates a context with a 60-second timeout. This means we expect all downstream operations to complete within 60 seconds; otherwise, the provided `cancel` function is called. The `cancel` function acts as an escape switch, allowing us to return early or update the `Context` state to wrap things up. Database and HTTP servers use this mechanism to remove dead connections or cancel long-running operations. Contexts are temporary values and should not be embedded in structs. Instead, they should be passed as values throughout the system as much as possible.

We can use this functionality when making the feed call. We can also derive new contexts from existing ones as you pass them along. In our case, we create a new context when initializing our error group. This allows us to cancel all ongoing operations if any error is returned, cutting the other calls short. This is especially useful when you want an all-or-nothing approach for a series of fetches.

9.4 Channels

Up until this point, we've basically allowed our function to run to its own conclusion asynchronously. We've used tools to help us know when it has completed, but we haven't fully used the benefits of concurrency yet. Imagine a modern factory with various stations for assembly. Some places may

have one large machine that does things in batches, while others have a series of machines feeding into and out of each other, possibly working on one piece or multiple pieces. But at no point will one item go through the entire pipeline before the next item starts.

To take this analogy even further, each station processes data differently and may have data flowing in and out of it in different ways. In manufacturing, this is a very complicated process that logistics and industrial engineering attempt to solve. In computer science, we face similar challenges with concurrency and routing. This has been a long-studied problem, and multiple languages attempt to solve communication in concurrent systems, each with its own tradeoffs.

Go and other languages use the *communicating sequential processes* (CSP) model, which acts somewhat like a queue for messages from one function to another. These queue-like structures in Go are known as *channels*. Channels in Go provide a safe and efficient way for goroutines to communicate and synchronize by passing values. A channel is a typed conduit through which you can send and receive values with the <- operator. Channels help coordinate work between concurrent goroutines, making it easier to build pipelines, manage shared state, and avoid race conditions without explicit locking. You can see how this is visualized in figure 9.1. They are a core part of Go’s approach to concurrency, enabling patterns like fan-in, fan-out, and worker pools, which we will explore in this section.

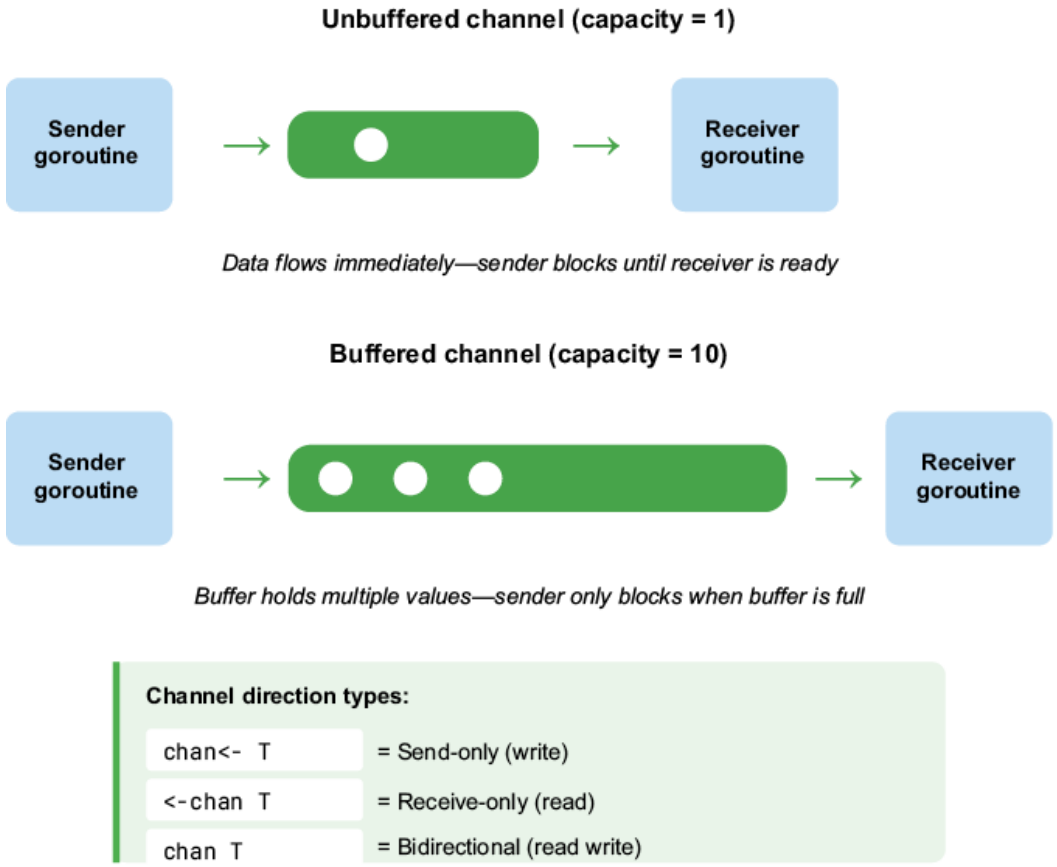


Figure 9.1 Channels in Go are like queues with various capacities

Because we will need some coordination, let's group some of our functions into a struct called `App`. We can then create the memory store and integrate it with our fetch feeds methods.

Listing 9.13 main.go: Application structure

```
type App struct { #1
 store *ArticleStore
}

func (a *App) fetchFeed(ctx context.Context, url string)
(gofeed.Feed, error) { #2
...
}


func (a *App) fetchFeeds(ctx context.Context, urls []string) {
...
}

func main() {
 ctx, cancel := context.WithTimeout(context.Background(), 60*time.S
econd)
 defer cancel()
 store := NewArticleStore() #3
 app := &App{ #4
  Store: store,
 }
 app.fetchFeeds(ctx, []string{
  "https://rss.nytimes.com/services/xml/rss/nyt/World.xml",
  "https://feeds.bbci.co.uk/news/rss.xml",
 })
 fmt.Println("stopped.")

}
```

#1 Create an app struct to hold our store.
#2 Change functions to methods on the new struct.
#3 Create a store to put in the struct.
#4 Initialize new structure.

Before we move forward, let’s enhance our application by adding an HTTP handler to our `App` struct. This handler will allow us to expose the articles we’ve saved so far through a simple API endpoint. By doing this, we can easily verify that our concurrent fetching and storage logic is working as

expected and provide a foundation for further experimentation with concurrency patterns in a real-world context.

We’ll implement a handler that responds with the most recent articles in JSON format. This will not only help us test our code but will also demonstrate how concurrency in Go can be integrated into practical applications, such as building a web service that aggregates and serves data from multiple sources concurrently.

Listing 9.14 main.go: Article handler

```
import (
 ...
 "encoding/json"
 "net/http"
 "strconv"
)

func (a *App) articlesHandler(w http.ResponseWriter,
r *http.Request) { #1
 numberOfArticles := 10
 if sizeStr := r.URL.Query().Get("count");
 sizeStr != "" { #2
  if size, err := strconv.Atoi(sizeStr); err == nil && size > 0 {
   numberOfArticles = size
  }
 }
 articles := a.store.GetRecent(numberOfArticles)
 w.Header().Set("Content-Type", "application/json")
 json.NewEncoder(w).Encode(articles) #3
}

func main(){
 ctx, cancel := context.WithTimeout(context.Background(), 60*time.S
econd)
 defer cancel()
 store := NewArticleStore()
 app := &App{
  Store: store,
 }
 app.fetchFeeds(ctx, []string{
  "https://rss.nytimes.com/services/xml/rss/nyt/World.xml",
  "https://feeds.bbci.co.uk/news/rss.xml",
 })
 http.HandleFunc("/articles", app.articlesHandler) #4
 fmt.Println("HTTP server listening on :8080")
 if err := http.ListenAndServe(":8080", nil);
 err != nil { #5
  fmt.Println("server error:", err)
 }
}
```

#1 Create a new handler based on the Handler type definition.
#2 Find a query parameter for the length of the list of articles to return.
#3 Encode article values.
#4 Handle calls to the /articles endpoint.
#5 Run the HTTP server until an error occurs.

In Go, channels are first-class values, meaning they can be created, passed to functions, and returned from functions just like any other type. This flexibility lets you design modular, concurrent programs in which communication between goroutines is abstracted and managed through function calls. By treating channels as values, you can encapsulate channel-creation logic inside functions, pass channels to worker goroutines, or return channels from factory functions, making your code more reusable and expressive.

Go channels are a powerful tool for coordinating work between goroutines. You can create channels using the `make` function and use them to send or receive values between concurrent functions. Channels can be returned from functions, passed as arguments, or even stored as fields in structs, enabling flexible and modular concurrent designs.

Let’s look at the different ways channels are used in the code. One common pattern is to return a channel from a function. This allows the caller to receive results or errors from multiple concurrent goroutines. For example, you might launch a goroutine for each URL you want to fetch and send any errors encountered to a shared error channel. The caller can then listen on this channel to handle errors as they occur.

Listing 9.15 main.go: Returning and reading channels from a function

```
func fetchFeeds(ctx context.Context, urls []string)
chan error { #1
 errChan := make(chan error) #2
 for _, url := range urls {
  go func() {
   feed, err := fetchFeed(ctx, url)
   if err != nil {
     errChan <- err #3
   }
  }
 }
 return errChan #4
}
```

#1 You can return a channel just like any other type.
#2 Use the make keyword to create a new unbuffered channel.
#3 Add items to a channel via the <- operator.
#4 Return the channel at the end.

Another approach is to pass a channel as an argument to a function. This decouples the producer, which generates values, from the consumer, which processes them. For example, you can create a channel for articles, pass it to a saving function running in its own goroutine, and then send articles to the channel as they are fetched. The saving function processes each article as it arrives. When the channel is closed and all values have been received, the loop in the consumer function exits.

```
func saveFeed(articles chan *gofeed.Article) { #1
 fmt.Println("adding articles...")
 for article := range articles { #2
  fmt.Println("adding article: ", article.Link)
  fmt.Println("article added: ", article.Link)
 }
 fmt.Println("all articles added")
}
```

#1 Pass in channels like any other parameter.
#2 Read data from channel using range.

Receiving from a channel in a goroutine is also a common pattern. The `saveFeed` function receives articles from the channel and processes them as they arrive. This function runs in its own goroutine and continues processing until the channel is closed. By using channels in this way, you can process data concurrently as it becomes available without having to manage explicit locking or shared state.

Listing 9.16 main.go: Passing a channel to another function

```
func fetchFeed(ctx context.Context, url string) (gofeed.Feed, erro
r) {
 fmt.Println("fetching feed: ", url)
 fp := gofeed.NewParser()
 feed, err := fp.ParseURLWithContext(ctx, url)
 if err != nil {
    return gofeed.Feed{}, err
 }
 fmt.Println("fetched: ", url)
 out := make(chan *gofeed.Article) #1
 go saveFeed(out) #2
 defer close(out) #3
 for _, article := range feed.Article{
  fmt.Println("ingesting article: ", article.Link)
  out <- article #4
 }
 fmt.Println("processed: ", url)
 return feed, nil
}
```

#1 Create a channel to pass to the save function.
#2 Pass as part of a goroutine.
#3 Close the channel when done.
#4 Write values to channel.

Channels can also be stored as fields in a struct, such as an `App` struct, to coordinate asynchronous operations across your application. By embedding channels in your struct, you can manage communication between different components, launch background goroutines, and handle results or errors in a centralized way. This approach is especially useful for

building scalable, maintainable concurrent systems in which different parts of your application need to interact safely and efficiently.

Listing 9.17 main.go: Adding channels to a struct

```
func (a *App) fetchFeed(ctx context.Context, url string)
(gofeed.Feed, error) {
...
 go a.saveFeed(out) #1
 defer close(out)
 for _, article := range feed.Article{
  fmt.Println("ingesting article: ", article.Link)
  out <- article
 }
 fmt.Println("processed: ", url)
 return feed, nil
}

func (a *App) saveFeed(articles chan *gofeed.Article) {
 fmt.Println("adding articles...")
 for article := range articles {
  fmt.Println("adding article: ", article.Link)
  a.store.AddArticle(article) #2
  fmt.Println("article added: ", article.Link)
 }
 fmt.Println("all articles added")
}
```

#1 Add struct receiver value to leverage the save method.
#2 Add struct receiver to access the store.

Here the `save` method creates a channel and starts a goroutine to consume it. This will run in the background until it is able to complete. Looking at how we range over the channel will show you that we are in a loop, blocking until that channel is explicitly closed. Like water from a tap, we will consume until it is shut off.

We've included some logging statements because we want to demonstrate the impacts of some of these different

channel patterns. Here we are making the saving of the articles concurrent, which means that at any point, another process can come along and start doing the same thing. The outcome of this run will be fairly predictable because of our configuration. Right now, our channel is of a default size of one, so this service will almost be like a pub/sub system: each time an article is published on the channel, it is immediately consumed, and another item cannot be added to the channel until the previous one has been grabbed.

We can define whether a channel can be read from, written to, or both. This adds a level of type safety that can help when writing your applications. For example, you may want one process to read values while another writes them. They share the same channel, but the types limit what each process can do. In our case, we will only ever read when we save an article. Let’s modify our code to demonstrate this.

Listing 9.18 main.go: Adding functions to fetch and save articles

```
func (a *App) fetchFeed(ctx context.Context, url string) {
 ...
 fmt.Println("fetched: ", url)
 var writeChan chan<- *gofeed.Article #1
 out := make(chan *gofeed.Article)
 writeChan = out #2
 go a.saveFeed(out)
 defer close(out)
 for _, article := range feed.Article{
  fmt.Println("ingesting article: ", article.Link)
  writeChan <- article
 }
 fmt.Println("processed: ", url)
}

func (a *App) saveFeed(articles
<-chan *gofeed.Article) { #3
 fmt.Println("adding articles...")
 for article := range articles {
  fmt.Println("adding article: ", article.Link)
  a.store.AddArticle(article)
  fmt.Println("article added: ", article.Link)
 }
 fmt.Println("all articles added")
}
```

#1 Create a channel to write only.
#2 Assign the channel to the write-only type.
#3 Receive only read only channel.

By specifying whether a channel is read-only, write-only, or bidirectional, Go enables developers to enforce stricter control over how data flows between different parts of an application. This directional type safety helps prevent accidental misuse of channels, making concurrent code easier to reason about and less prone to bugs.

Now that you've seen how channel directionality can clarify communication patterns, let's explore another important aspect of channels: buffering. Buffered channels allow us to

control how many values can be queued in a channel before send operations block, giving us even more flexibility in managing concurrent workflows.

If you watch, you'll notice that each time something is added to the channel, it is immediately processed, but what you may not be seeing is that nothing else can be added to the channel until the value is processed. This seems a little counterintuitive from a concurrency perspective, since we are blocking until a value is processed. This is because our channel has a capacity of one: it can hold a single value at a time, and any attempt to send a second value will block until the first has been received.

We can instead change the size to be of whatever capacity we want. This gives us the ability to accumulate values and process them as we have capacity, much like queues and workers. Let's modify our channel to have a larger capacity. With this change, our channel can now hold more items, allowing the producer to continue sending values without waiting for each one to be processed immediately. This buffered approach improves throughput and decouples the timing between producers and consumers.

```
func (a *App) fetchFeed(ctx context.Context, url string) {
...
 fmt.Println("fetched: ", url)
 out := make(chan *gofeed.Article, 10) #1
 go a.saveFeed(out)
 ...
}
```

#1 Add a buffer of 10 to the channel.

So far, we've demonstrated how to pass channels as values, which is a common pattern in Go. But channels are even more powerful when used as fields within a struct, enabling you to coordinate asynchronous operations across your

application. By embedding channels in your struct, you can manage communication between different components, launch background goroutines, and handle results or errors in a centralized way. Let’s see how this works by modifying our `App` struct to include channels as properties.

Listing 9.19 main.go: Using channels within a struct

```
type App struct {
 store *ArticleStore
 feedRead chan string
}

func (a *App) run() {
 for feed := a.feedRead { #1
  ctx := context.WithTimeout(context.Background(), time.Second * 3
0)
  go fetchFeed(ctx, feed)
 }
}

func (a *App) fetchFeeds(ctx context.Context,
urls []string) { #2
 for _, feed := range urls {
  a.feedRead <- feed  #3
 }
}

func (a *App) Close() {
 close(a.feedRead) #4
}

func NewApp() *App {
 feedReader := make(chan string, 5) #5
 store := NewArticleStore()
 app := &App{
  Store: store,
  feedRead: feedReader,
 }
 go app.run() #6
 return app
}
```

#1 Consume any messages from the internal channel just like a queue.
#2 Fetch feeds will add messages to the channel.
#3 Add a new URL to the channel.
#4 Close channel when app is done.
#5 Create a buffered channel for internal messaging.
#6 Start consumer in the background.

We’ve created a channel and a background function to fetch messages as items are added, allowing automatic ingestion.

As our application grows, we may want to handle multiple streams of data flowing through the system, such as feed URLs, articles, and errors, all at the same time. Managing these different types of information concurrently is a common requirement in real-world applications, where you need to coordinate work, handle results, and process errors independently.

To further improve our application’s robustness, we can incorporate error handling by introducing a separate channel dedicated to processing error messages. This approach keeps the flow of data and error reporting decoupled, making the system more flexible and easier to maintain.

```
type App struct {
 store *ArticleStore
 feedRead chan string
 errChan chan error #1
}

...

func (a *App) fetchFeed(ctx context.Context, url string) {
 fmt.Println("fetching feed: ", url)
 fp := gofeed.NewParser()
 feed, err := fp.ParseURLWithContext(ctx, url)
 if err != nil {
  fmt.Println("errored: ", url)
  errChan <- err #2
  return
 }
 fmt.Println("fetched: ", url)
...
 fmt.Println("processed: ", url)
}

func (a *App) Close() {
 close(a.feedRead)
 close(a.errChan) #3
}

func NewApp() *App {
 feedReader := make(chan string, 5)
 errChan := make(chan error, 5)
 store := NewArticleStore()
 app := &App{
  Store: store,
  feedRead: feedReader,
  errChan: errChan, #4
 }
 go app.run()
 return app
}
```

#1 Structs can hold multiple channels for coordination.
#2 If an error occurs, publish a message on the error channel.

#3 Close channels as part of a proper shutdown.
#4 Add a new channel to the struct.

Here we publish the error to the error channel if one occurs during ingestion. This means that if an error happens while fetching or processing a feed, we immediately send the error to the error channel (`errChan`) and exit the current goroutine, allowing the rest of the application to continue processing other feeds or tasks. This approach ensures that errors are not silently ignored and can be handled or logged elsewhere in your application.

Go's `select` statement is a powerful feature for handling multiple channels concurrently. It allows a goroutine to wait on multiple communication operations and proceed with whichever channel is ready first. This is especially useful when you need to coordinate work across several channels, such as receiving data, handling errors, or responding to cancellation signals. With `select`, you can implement timeouts, multiplex input from different sources, or perform nonblocking operations by including a `default` case. This pattern enables responsive, robust concurrent programs by letting your code react dynamically to whichever event occurs first among several possible channel operations.

```
func (a *App) run() {
 for {
  select { #1
  case feed := <-a.feedRead: #2
   ctx, cancel := context.WithTimeout(context.Background(), 30*tim
e.Second)
   go a.fetchFeed(ctx, feed)
   cancel()
  case err := <-a.errChan: #3
   fmt.Println("error:", err)
  }
 }
}
```

#1 Create a select statement to handle multiple channels.
#2 Read from feed channel and handle the messages.
#3 Read errors and print the results.

By using `select`, your main event loop can efficiently coordinate work, such as fetching feeds or handling errors, without blocking on any single channel. This pattern is fundamental in Go for building responsive, concurrent systems that need to manage multiple streams of communication. With this understanding, you're now equipped to structure your application's core event loop for robust, real-world concurrency.

Next, we'll see how these concepts come together in practical concurrency patterns and how to expose your concurrent logic through an API.

9.5 Concurrency patterns

Once you understand the various use cases and setup of concurrency, it's helpful to learn some common patterns that will improve your concurrent programs. In this section, we'll cover two essential patterns—worker pools and fan-in/fan-out—both of which are useful for increasing your application's throughput.

> **NOTE** For a deeper exploration of concurrency patterns and best practices, see James Cutajar's book *Learn Concurrent Programming with Go* (Manning, 2023).

A worker pool allows you to process many tasks concurrently while controlling the level of parallelism. By creating a fixed number of worker goroutines that pull tasks from a shared queue (channel), you can efficiently distribute work, avoid resource exhaustion, and handle bursts of load gracefully. This pattern is especially useful when you have a large number of independent jobs to process, such as fetching multiple feeds or handling incoming requests.

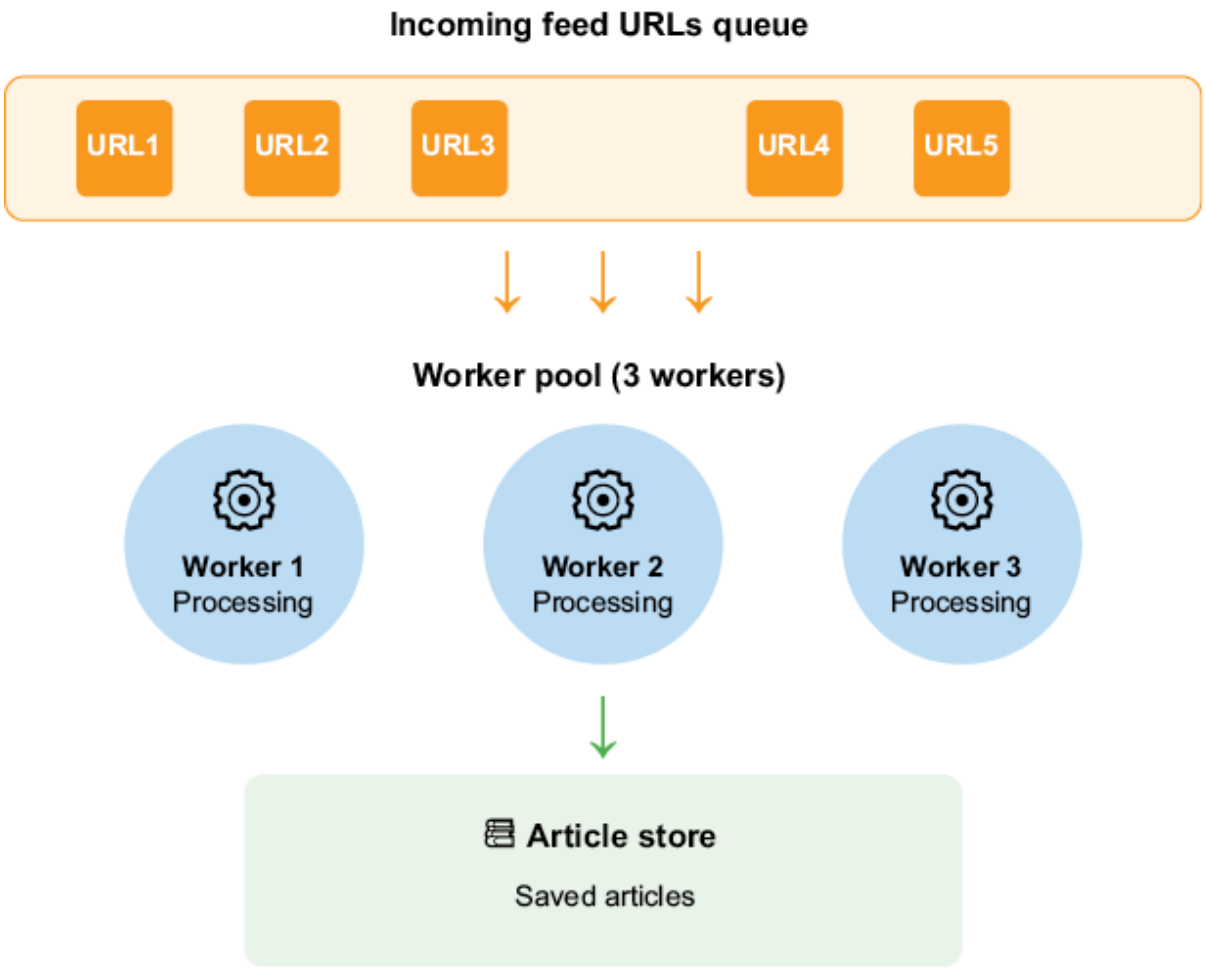
The fan-out/fan-in pattern is another powerful concurrency technique. *Fan-out* means launching multiple goroutines to perform work in parallel, such as fetching data from several sources at once. *Fan-in* is the process of collecting results from these goroutines, aggregating their outputs, and handling errors or combining data as needed. This pattern is ideal for scenarios where you need to maximize throughput and responsiveness by using parallelism while still coordinating and merging the results in a controlled way.

Together, these patterns help you design robust, high-performance applications that can handle concurrent workloads efficiently, scale with demand, and maintain a clear, maintainable code structure.

9.5.1 Workers

We have compared our concurrency approach to services that sit at the end of a queue. In systems architecture, this service is often referred to as a *worker*, and it can scale depending on the load at any given time. In figure 9.2, you can see how workers handle workloads. We can build a

similar structure in our application by treating our channel as a queue and creating a pool of workers to ingest our feeds.



Multiple workers process jobs from a shared queue concurrently

Figure 9.2 Worker pools ingest incoming messages across multiple processes.

Let's begin by extending our `App` struct into a worker that also has a channel for incoming requests.

Listing 9.20 main.go: Creating workers

```
type Worker struct {
 *App #1
 incomingFeeds <-chan string #2
}

func (w *Worker) run() {
 for url := range w.incomingFeeds { #3
  ctx, _ := context.WithTimeout(context.Background(), 30*time.Secon
d)
  w.App.fetchFeed(ctx, url)
 }
}

func NewWorker(app *App, incomingFeeds <-chan string) *Worker { #4
 w := &Worker{
  App: app,
  incomingFeeds: incomingFeeds,
 }
 go w.run()
 return w
}
```

#1 Embed the App struct to reuse its methods and data to access underlying functions.
#2 Add a read-only channel for incoming feed URLs to go to the worker to process.
#3 Continuously read URLs from the channel and process them as they come in.
#4 This is a factory function to create and start a new Worker and start it.

That's it! Now we have a service that can run in the background and ingest our feeds.

Having a single worker is helpful, but using multiple workers allows you to process more feeds concurrently. To do this, we'll create a `WorkerPool` that manages several worker services, all sharing a single queue for incoming work. The pool size, determined by the number of workers, can be set as a parameter during creation.

Listing 9.21 main.go: Worker pools

```
type WorkerPool struct {
 workers []*Worker #1
 incomingFeeds chan string #2
}

func NewWorkerPool(app *App, count int) *WorkerPool {
 wp := &WorkerPool{}
 wp.incomingFeeds := make(chan string, 100) #3
 for i := range count {
  w := NewWorker(app, incomingFeeds) #4
  append(wp.workers, w)
 }
 return wp
}

func (wp *WorkerPool) Submit(url string) {
 wp.incomingFeeds <- url #5
}

func (wp *WorkerPool) Stop() {
 close(wp.incomingFeeds)
}
```

#1 List of workers managed by the pool
#2 Channel for distributing feed URLs to workers
#3 Initialize the shared channel with a buffer.
#4 Start each worker using the shared channel.
#5 Submit a feed URL to the pool for processing.

A slice holds all worker instances in the pool, while a channel acts as a shared queue for incoming feed URLs. By creating a buffered channel, feed URLs can be queued for workers, ensuring that all workers share access to the same queue. Each worker is created with a reference to this shared channel, allowing it to pull feed URLs from the common queue. To submit a new feed URL to the pool, the URL is simply sent to the shared channel, where it will be picked up by the next available worker.

Finally, let's incorporate the worker pool into our `App` struct so it runs in the background and handles feed ingestion for our API. By embedding `WorkerPool` as a field in the `App` struct, we can delegate the processing of feed URLs to a pool of worker goroutines, improving throughput and responsiveness. The `App`'s `run` loop submits incoming feed URLs to the pool, and the pool distributes them among its workers for concurrent processing. This design decouples the submission of work from its execution, allowing the application to scale efficiently as load increases.

Listing 9.22 main.go: Adding a worker pool to app creation

```
type App struct {
...
 workerPool *WorkerPool #1
}

func (a *App) Close() {
...
 a.workerPool.Stop() #2
}

func NewApp() *App {
 ...
 app := &App{
  ...
 }
 app.workerPool = NewWorkerPool(app, 3) #3
 go app.run()
 return app
}
```

#1 Add a workerPool field to the App struct to manage background workers.
#2 Stop the worker pool when closing the app to clean up resources.
#3 Initialize the worker pool with a specified number of workers.

9.5.2 Fan-out and fan-in

There will be times when you want to use concurrency but still notify the caller. We explored some of those situations earlier when we talked about wait groups. Then we looked at establishing async flows through the system using channels. If you combine the two, a powerful pattern emerges: fan-out/fan-in. You can see the "fans" in figure 9.3. This pattern requires some bookkeeping at the end of the function, but the structure of the Go concurrency model makes it easy to implement.

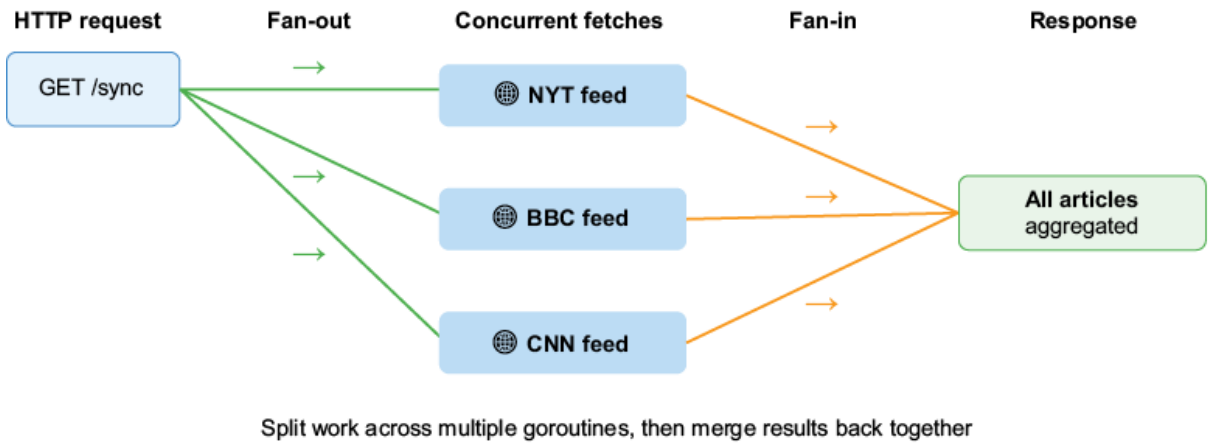


Figure 9.3 Fan-out/fan-in

We will demonstrate how this works by creating an endpoint that forces the fetching of new results, saves them, and then responds to the caller with the list of articles. To do this, we need to make a slight change to our `fetchFeeds` method by adding a `return` statement and publishing messages on the channel. This `return` statement will help us merge the data. It will not affect performance because we will already have all of the articles.

```
func (a *App) fetchFeed(ctx context.Context, url string)
([]*gofeed.Item, error) {
 fmt.Println("fetching feed: ", url)
 fp := gofeed.NewParser()
 feed, err := fp.ParseURLWithContext(ctx, url)
 if err != nil {
  fmt.Println("errored: ", url)
  a.errChan <- err
  return nil, err #1
 }
 fmt.Println("fetched: ", url)

 // Channel with capacity
 out := make(chan *gofeed.Item, 10)
 go a.saveFeed(out)
 defer close(out)

 for _, article := range feed.Items {
  fmt.Println("ingesting article: ", article.Link)
  out <- article
 }
 fmt.Println("processed: ", url)
 return feed.Items, nil #2
}
```

#1 We want to return an error if one occurs to handle it synchronously.
#2 We want to ensure we return items for our fan-in process.

Now we can create our handler to fetch the same endpoints as before, save the articles to our store, and return the aggregated results to the client, all using the fan-out/fan-in concurrency pattern. In this approach, we “fan-out” by launching a separate goroutine for each feed URL, allowing all feeds to be fetched in parallel. Each goroutine independently fetches its assigned feed, processes the articles, and sends the results (or any errors) back through a shared results channel.

We then “fan-in” by collecting results from all goroutines via the results channel, using a `WaitGroup` to track completion, and closing the channel when all fetches are done. As

results arrive, we aggregate the articles and collect any errors. This pattern ensures our handler is responsive, uses system resources efficiently, and provides a complete response, including both successfully fetched articles and any errors encountered, making it robust and scalable for concurrent operations.

Listing 9.23 main.go: New handler for fan-out/fan-in call

```
func (a *App) syncHandler(w http.ResponseWriter, r *http.Request) {

 // Fan-out: launch goroutines for each feed
 type result struct {
    articles []*gofeed.Item
    err      error
 }
 resultsChan := make(chan result, len(feeds)) #1
 var wg sync.WaitGroup #2

 for _, feedURL := range feeds {
    wg.Add(1)
    go func(url string) {
     defer wg.Done()
     ctx, cancel := context.WithTimeout(context.Background(), 30*ti
me.Second)
     defer cancel()

     fetchedArticles, err := a.fetchFeed(ctx, url)
     resultsChan <- result{articles: fetchedArticles, err: err} #3
    }(feedURL)
 }

 // Close results channel after all goroutines complete
 go func() {
    wg.Wait()
    close(resultsChan)
 }() #4
```

#1 Create a buffered channel for results.
#2 WaitGroup to track goroutines for fetching results.
#3 Send result to channel.
#4 Close channel after all goroutines finish.

This pattern should look familiar: we create a channel and a `WaitGroup` and then iterate over the feeds, launching a goroutine for each fetch operation. As each result becomes available, it is sent on the results channel for aggregation.

Listing 9.24 main.go: Waiting for the results

```
func (a *App) syncHandler(w http.ResponseWriter, r *http.Request) {
 ...

 // Close results channel after all goroutines complete
 go func() {
  wg.Wait() #1
  close(resultsChan) #2
 }()

 // Fan-in: collect results from all goroutines
 var errors []string
 var articles []*gofeed.Item
 for res := range resultsChan { #3
  if res.err != nil {
   errors = append(errors, res.err.Error())
  } else {
   articles = append(articles, res.articles...)
  }
 }
```

#1 Launch a goroutine to close the results channel after all fetches complete.
#2 Wait for all goroutines to finish using the WaitGroup.
#3 Collect articles and errors from each fetch operation.

As the results come in, we aggregate the data for later use. The `for-range` loop over the results channel ensures that we process each result as soon as it arrives, regardless of the order in which the goroutines complete. This approach lets us efficiently collect articles and errors from all concurrent fetch operations without blocking on any single request. The loop continues to receive and aggregate results until the results channel is closed, which happens only after all goroutines have finished their work and the `WaitGroup` signals

completion. This guarantees that we don’t miss any results and that our aggregation is complete before we respond to the client.

Finally, we can return the aggregated values to the client.

Listing 9.25 main.go: Create the response

```
func (a *App) syncHandler(w http.ResponseWriter, r *http.Request) {
 ...
// Return errors if any occurred
 if len(errors) > 0 { #1
  w.WriteHeader(http.StatusPartialContent)
  response := map[string]interface{}{
   "articles": articles,
   "errors":   errors,
  }
  json.NewEncoder(w).Encode(response) #2
  return
 }

 // Return aggregated articles
 w.Header().Set("Content-Type", "application/json")
 json.NewEncoder(w).Encode(articles) #3
}
```

#1 If any errors occurred, return a partial response with both articles and errors.
#2 Encode the response as JSON, including both articles and errors.
#3 Otherwise, encode and return the aggregated articles as JSON.

This approach ensures that all feed fetches are attempted in parallel and that the results, both successful articles and any errors, are collected and returned in a single response.

To complete this step, simply add the new handler to your API, as shown in the following listing. With this endpoint in place, clients can trigger a synchronous refresh of all feeds and receive the latest articles and any errors in a single, consistent response. This demonstrates the power of Go’s

concurrency model and provides a practical, scalable solution for real-world data aggregation tasks.

Listing 9.26 main.go: Add the handler

```
var feeds = []string{
 "https://rss.nytimes.com/services/xml/rss/nyt/World.xml",
 "https://feeds.bbci.co.uk/news/rss.xml",
}

func main() {
 app := NewApp()
 defer app.Close()

 // Submit initial feeds
 for _, feed := range feeds {
  app.feedRead <- feed
 }

 // Set up HTTP handler
 http.HandleFunc("/articles", app.articlesHandler)
 http.HandleFunc("/sync", app.syncHandler) #1

 fmt.Println("HTTP server listening on :8080")
 if err := http.ListenAndServe(":8080", nil); err != nil {
  fmt.Println("server error:", err)
 }
}
```

#1 Add new sync handler.

In this small project, you can see how quickly concurrent code can become complex. As we introduced new modules and experimented with different concurrency patterns, the codebase grew more intertwined and harder to reason about. This is a common code smell in concurrent programming, often resulting from poor planning, unclear responsibilities, or frequent changes without refactoring. The key to managing this complexity is to keep your modules focused, design clear communication patterns, and

manage your channels and synchronization primitives carefully.

By structuring your code thoughtfully and following established concurrency patterns, you can build scalable and maintainable concurrent applications in Go. But it’s important to recognize that concurrency introduces new challenges, such as race conditions, deadlocks, and resource leaks, that require discipline and best practices to avoid.

Let’s now review some best practices for writing safe, efficient, and maintainable concurrent code in Go.

9.6 Helpful concurrency tips

Concurrency in Go is a powerful tool that can greatly improve the performance and responsiveness of your applications, but it also introduces new challenges and potential pitfalls. Although Go makes it easy to launch goroutines and coordinate work with channels, it’s important to follow best practices to avoid common problems such as race conditions, deadlocks, and resource leaks. By understanding both the strengths and limitations of Go’s concurrency model and applying proven patterns and techniques, you can write safe, efficient, and maintainable concurrent code. The following sections outline key best practices, good use cases, and situations where concurrency may not be the right choice.

9.6.1 Best practices for Go concurrency

Writing concurrent code in Go can be powerful, but it’s important to keep your approach clear and maintainable. Here are some key guidelines:

- *Keep it simple*—Favor straightforward, readable code. Avoid unnecessary complexity; simple concurrency is easier to reason about and debug.
- *Prefer channels over shared memory*—Use goroutines and channels for communication and synchronization. Passing data through channels helps prevent race conditions and makes your code safer.
- *Protect shared data*—If you must share data between goroutines, always use synchronization primitives like `sync.Mutex` or `sync.RWMutex` to avoid race conditions and ensure data consistency.
- *Use context for cancellation and timeouts*—Use `context.Context` to manage cancellation, deadlines, and request-scoped values. This makes your concurrent code more robust and responsive.
- *Limit goroutine count*—Don't launch more goroutines than your system can handle. Use worker pools to control concurrency and manage workloads efficiently, or use a `SetLimit` function on structures like `errgroup`.
- *Close channels properly*—Always close channels when they're no longer needed to prevent resource leaks and allow goroutines to exit cleanly.
- *Handle errors explicitly*—Make sure to capture and propagate errors from concurrent operations so they can be logged or handled appropriately.
- *Test for race conditions*—Use `go test -race` during development to catch race conditions early and ensure your code is safe for production.
- *Keep track of goroutines*—Use a `WaitGroup` or `errgroup` rather than a plain `go` keyword.

9.6.2 When to use concurrency in Go

Concurrency is especially useful in Go in a variety of scenarios where tasks can run independently or in parallel. Here are some common situations where concurrency can improve your application's performance and responsiveness:

- *Handling multiple network requests*—Web servers, API gateways, and microservices often need to process many requests at the same time. Using concurrency allows each request to be handled in its own goroutine, improving throughput and responsiveness.
- *Performing I/O-bound operations in parallel*—When your application needs to read or write files, make HTTP requests, or query databases, concurrency lets you perform these operations simultaneously, reducing wait times and making better use of system resources.
- *Building data pipelines*—In scenarios like ETL jobs or streaming data processing, different stages of the pipeline can run concurrently, passing data between stages using channels for efficient and safe communication.
- *Running background tasks*—Tasks such as sending emails, notifications, or scheduled jobs can run in the background using goroutines so they don't block the main application flow.
- *Implementing worker pools*—When you have a queue of jobs to process, worker pools let you control the level of parallelism and efficiently distribute work among multiple goroutines.
- *Managing real-time data feeds or event streams*—Applications like chat servers or financial tickers benefit from concurrency by handling multiple streams of data in real time.
- *Parallelizing CPU-bound computations*—For tasks such as image processing or scientific simulations, concurrency enables you to use multiple CPU cores and speed up computation.
- *Coordinating multiple sensors or devices*—In embedded or IoT applications, concurrency helps manage communication and data collection from multiple devices at once.
- *Building responsive user interfaces*—In GUI applications, concurrency allows background work to happen without freezing the user interface, keeping the application interactive.

9.6.3 When not to use concurrency in Go

Although concurrency is a powerful feature, there are situations where it may not be appropriate or beneficial:

- *Simple, single-task programs*—If your application is straightforward, short-lived, or only performs one task, concurrency can add unnecessary complexity without any real advantage. In these cases, a sequential approach is easier to write, understand, and maintain.
- *Tasks that must run in strict order*—When operations are tightly coupled and must be performed in a specific sequence, introducing concurrency can make the code harder to reason about and can introduce subtle bugs.
- *Lack of experience with concurrency*—If you are not comfortable with synchronization, race conditions, or debugging concurrent code, it's best to avoid concurrency until you have more experience. Improper use can lead to hard-to-find bugs and unpredictable behavior.
- *Overhead outweighs benefits*—For very small or performance-critical sections, the overhead of managing goroutines and channels may outweigh any potential gains. In these cases, a simple, sequential solution is often more efficient.

You've seen that Go's concurrency model is both simple and powerful, making it easy to add parallelism and responsiveness to your applications. With goroutines and channels as first-class features, you can quickly build scalable systems that handle multiple tasks efficiently. Still, concurrency introduces new challenges, such as race conditions and increased complexity, that require careful design and discipline. Always start with a straightforward, sequential solution, and introduce concurrency only when it's truly needed. By following this approach, you'll keep your code maintainable while still taking full advantage of Go's elegant concurrency tools.

Summary

- Go's concurrency model is built around goroutines and channels, making concurrent programming simple and efficient.

- Use goroutines to run functions concurrently without managing threads directly.
- Synchronize goroutines using tools like `WaitGroup`, mutexes, and the `errgroup` package for error handling.
- Protect shared data with mutexes to avoid race conditions and ensure data consistency.
- Use `context.Context` to manage cancellation, timeouts, and request-scoped values in concurrent operations.
- Channels provide safe communication between goroutines and enable patterns like pipelines, worker pools, and fan-in/fan-out.
- Buffered and directional channels help control data flow and improve throughput.
- Worker pools and fan-in/fan-out are common concurrency patterns for scalable, efficient processing.
- The `select` statement allows a goroutine to wait on multiple channels, enabling responsive and robust event loops.
- Always close channels when you're done and handle errors explicitly to avoid resource leaks and silent failures.
- Use concurrency when tasks can run independently, especially for I/O-bound, network, or parallelizable workloads.
- Avoid unnecessary concurrency in simple, strictly ordered, or single-task programs to keep code maintainable.

10 The standard library

This chapter covers

- Command-line tool-creation basics
- Discovering libraries within the standard library
- Using the standard library for common use cases

Programming languages provide us with a basic syntax and grammar to use when writing our applications. Each language uses different primitives and patterns that make the language unique, much like our own languages. We have words, ways to pronounce them, and ways to string them together into a standard grammar, and we compile them into pages, books, and libraries. With programming languages, we end up making our own types of libraries, or collections of commonly used code, that speed up development.

When Go is installed, it includes its own library, known simply as the *standard library*, as opposed to community libraries, which are installed afterward. We have already used some standard library functions and structures in our small programs. In fact, it is fairly difficult to write anything without a standard library. The fact that so much can be done with the standard library makes Go a great language for modern application development.

To demonstrate the power of the standard library, this chapter will show you how to build an application that uses many of its most commonly used packages. The application will be a JSON and XML validation and conversion utility that takes advantage of some of Go’s common abstractions,

such as `Reader` and `Writer`, that explores operating system methods and functions, that shows how to parse and manipulate data, and that pipes the data through different packages to determine its form and structure. Each section of the chapter will add new functionality to this application, demonstrating how the different standard library packages can work together in a real application. We will not be able to touch on all of the standard library packages, but you will learn enough to write a powerful program.

10.1 The fmt package

"Hello World!" is a common first application for people who are learning how to program or learning a new programming language. Output is an essential part of many applications, and it can be handled in a variety of ways. One of the most straightforward ways is to use the `fmt` package. You have already seen this package, but it deserves its own section because it is often the first package people encounter when creating an application.

The `fmt` package in Go provides formatted I/O functions similar to those found in C's `printf` and `scanf`. It is widely used for printing output to the console, formatting strings, and reading input. The package supports a variety of formatting verbs that allow you to control how values are displayed or parsed, making it a fundamental tool for both debugging and user-facing output.

Let's start building our JSON/XML validation and conversion CLI tool by creating a new `main.go` file and outputting the name of our application. We'll expand this file as we progress through the chapter, adding flags, file I/O, data conversion, and validation. For now, choose a name for your tool, such as `myapp`, and let's get started.

Listing 10.1 main.go: Simple output using Print

```
package main

import (
    "fmt" #1
)

func main() {
    fmt.Print("myapp") #2
}
```

#1 Importing the standard library package
#2 Print will display text on a screen.

This is the most common and direct way to output to a screen with no additional formatting. If we were to add a second line as a description, we would do something like the following.

Listing 10.2 main.go: Additional output without formatting

```
import (
    "fmt"
)

func main() {
    fmt.Print("myapp")
    fmt.Print("tools to convert between json and xml") #1
}
```

#1 The second line will run into the first without the newline.

If you were to run this, you would see that the text runs together. Our assumption was that two separate `Print` functions would print on two separate lines, but they don't. To get the expected format, we need to use `Println`, or print line, which automatically puts a return at the end of our strings.

```
import (
    "fmt"
)

func main() {
    fmt.Println("myapp") #1
    fmt.Println("tools to convert between json and xml")
}
```

#1 Each line will be printed with a return at the end.

Now we will see the output on two separate lines. But maybe we would like to have a little more control over the format of our output. Let's say we want to set the name as a variable and use it throughout the app, but also output it. To do that, we can use `Printf`.

```
import (
    "fmt"
)

func main() {
    app := "myapp"
    version := 1
    fmt.Printf("%s - version %d - tools to convert between json and
xml",
     app, version) #1
}
```

#1 Formatting text with escape characters converts values to text for display.

Here, we are outputting our variables with a preformatted string. This can be very helpful as we build string values, because we won't need to worry about conversion and we gain some type safety. The "%" values are known as *escape codes*, and they can output a variety of values, as listed in table 10.1.

Beyond printing to the console, the `fmt` package provides flexible functions for creating formatted strings that can be

assigned to variables, passed to other functions, or used to construct custom error messages. This is particularly useful when you need to build complex output or error details before deciding how to use them.

Table 10.1 Common fmt escape codes

Code	Type	Description	Example output
`%v`	`any`	Default format	`{Name:myapp}`
`%+v`	`struct`	Includes field names	`{Name:myapp Version:1}`
`%#v`	any	Go syntax representation	`main.Config{Name:"myapp"}`
`%T`	any	Type of value	`main.Config`
`%t`	`bool`	Boolean	`true`
`%d`	`int`	Decimal integer	`42`
`%b`	`int`	Binary	`101010`
`%x`	`int`	Hexadecimal (lowercase)	`2a`
`%X`	`int`	Hexadecimal (uppercase)	`2A`
`%f`	`float`	Decimal point (always six digits)	`3.141593`
`%.2f`	`float`	Decimal with precision	`3.14`
`%e`	`float`	Scientific notation	`3.141593e+00`
`%s`	`string`	String	`myapp`

Code	Type	Description	Example output
`%q`	`string`	Quoted string	`"myapp"`
`%p`	pointer	Pointer address	`0xc000010200`
`%%`	literal	Percent sign	`%`

The `Sprint` family of functions works identically to `Print` but returns the formatted output as a string instead of writing it to standard output. This string is stored in a variable that you define, where it remains in memory until your program uses it, the program reassigns the variable, or the variable goes out of scope. You might use this captured string immediately in a conditional check, pass it to another function, append it to a log file, or save it for later processing; the timing and use depend entirely on your application's needs.

Listing 10.3 Building strings with Sprint functions

```
import "fmt"

func main() {
    app := "myapp"
    version := 1

    // Sprint concatenates arguments
    message := fmt.Sprint(app, " v", version) #1

    // Sprintf formats with escape codes
    formatted := fmt.Sprintf("%s - version %d", app, version) #2

    // Use the strings however needed
    fmt.Println(message)
    fmt.Println(formatted)
}
```

#1 Returns "myapp" as a string
#2 Returns a formatted string with type conversion

Finally, we can use the `fmt` package not only to print output but also to read input. By using `fmt.Scan`, `fmt.Scanf`, and `fmt.Scanln`, we can display output and then wait for input.

The `fmt` package also provides functions for reading user input directly from the terminal. The `Scan` family of functions (`Scan`, `Scanf`, `Scanln`) reads from standard input and parses values according to the format you specify. Although our CLI tool will primarily read from files or piped input, understanding these functions is useful for building interactive tools.

The following listing shows how you can read input and create formatted error messages.

Listing 10.4 main.go: Reading input and creating formatted errors

```
func main() {
    ...

    var input string
    fmt.Print("Input json: ") #1
    fmt.Scanln(&input) #2

    if input == "" {
        err := fmt.Errorf("invalid json: empty input") #3
        fmt.Println(err) #4
        os.Exit(1)
    }

    fmt.Printf("Received: %s\n", input)
}
```

#1 Display a prompt to the user.
#2 Read a line of input into the variable (note the pointer).
#3 Create a formatted error using Errorf.
#4 Write the error to stderr instead of stdout.

10.2 The flag package

Applications like CLI tools and other services often need configuration without changes to the underlying code. One common way to provide this flexibility is to let users pass options or settings directly when running the program from the terminal. In Go, the standard library provides the `flag` package to make this process straightforward. A *flag* is a command-line option that lets users specify values or toggle features, such as setting an input file, enabling verbose output, or providing a configuration path. The `flag` package offers a simple API to define, parse, and retrieve these values, making it easy to build flexible, user-friendly CLI tools.

Flags can be defined with both short (single hyphen, e.g., `-h`) and long (double hyphen, e.g., `--help`) forms, though the

standard library's `flag` package does not distinguish between the two formats. Flags can accept different types of input: Boolean flags, which are `true` if present, such as `-verbose`; string flags, such as `-config=config.json`; integer flags; and more. To extract these values, you define variables using functions like `flag.Bool`, `flag.String`, or `flag.Int`, and then call `flag.Parse()` to process the command-line arguments. After parsing, you can access the values through the variables you defined. This approach allows you to easily add configurable options to your CLI tool, making it more flexible and user-friendly.

Let's first define a `help` flag that we can update as we add new information, and use some other tools to help maintain it.

Listing 10.5 main.go: Adding a help flag

```
import (
    "flag"
    "fmt"
)

func main() {
    app := "myapp"
    version := 1

    help := flag.Bool("help", false, "Show help message") #1
    flag.Parse() #2

    if *help { #3
        fmt.Printf("Usage: %s [options]\n", app)
        fmt.Println("  -help   Show help message")
        return
    }

    fmt.Printf("%s - version %d - tools to convert between json and
xml\n", app, version)
}
```

#1 Set a new Boolean flag as a possible input.
#2 Parses input to see if flags have been set
#3 All values are pointers and need to be dereferenced to get the value.

We can also add optional flags to display something like a banner.

Listing 10.6 main.go: Optional string flag for banner

```
import (
    "flag"
    "fmt"
    "os"
)

func main() {
...
    banner := flag.String("banner", "", "Display a custom banner me
ssage")
    flag.Parse()

    if *help {
...
        fmt.Println("  -banner   Display a custom banner message")
        return
    }

    if *banner != "" {
        fmt.Println(*banner)
    }

...
}
```

For more advanced aliasing, consider using third-party packages like `pflag` or `cobra`. These packages aren't part of the standard library, but they provide additional tooling that helps you build more sophisticated CLI applications. They offer features like flag aliasing, nested subcommands, and automatic help generation, which go well beyond what the standard library's `flag` package supports on its own.

The final way you can use the `flags` package is to capture arguments passed to the application. Arguments are typically the final values passed into the application and should be differentiated from the flags themselves. While flags are optional parameters that modify behavior (like `-`

`help` or `-banner="Welcome"`), *arguments* are mandatory positional values that the application requires to function, such as input and output file paths. The `flag.Args()` function returns any additional command-line arguments that are not parsed as flags. This is useful for handling positional arguments, such as input or output filenames, after all defined flags have been processed. For example, if your program is run as `myapp -banner="Welcome" input.json output.xml`, you can call `flag.Parse()`, and `flag.Args()` will return a slice containing `["input.json", "output.xml"]`. Your application can then validate that the required number of arguments was provided and handle the errors appropriately if mandatory arguments are missing.

Listing 10.7 main.go: Checking for input and output files

```
func main() {
...
    flag.Parse()
    args := flag.Args() #1
    if len(args) < 2 {
        fmt.Println("Error: Please provide input and output file
s.")
        fmt.Println("Tip: Use -help for usage information.")
        return
    }
...
    inputFile := args[0] #2
    outputFile := args[1]

    fmt.Printf("Input file: %s\n", inputFile)
    fmt.Printf("Output file: %s\n", outputFile)
}
```

#1 Grab arguments passed into the flags as a slice.
#2 The argument slice returns values passed into the application.

The `flag` package is a powerful, easy-to-use tool for handling command-line options in Go applications. It lets you define and parse a variety of flag types, manage help

messages, and handle positional arguments, all with minimal code. Although it covers most basic needs for CLI tools, you can always extend its functionality or use third-party libraries for more advanced scenarios. Mastering the `flag` package will help you build flexible, user-friendly command-line applications that are easy to configure and maintain.

10.3 The io package

The `io` package defines fundamental interfaces for I/O operations. Understanding these interfaces is key to writing flexible, composable code that can work with files, network connections, memory buffers, and any other data source. Most other I/O-related packages in the standard library build on these core interfaces.

The most important of the `io` interfaces are `Reader`, `Writer`, and `Closer`, which are then combined into other interfaces, such as `ReaderWriter` and `ReadCloser`. In figure 10.1, you can see the relationship between the `io` interfaces and the implementations.

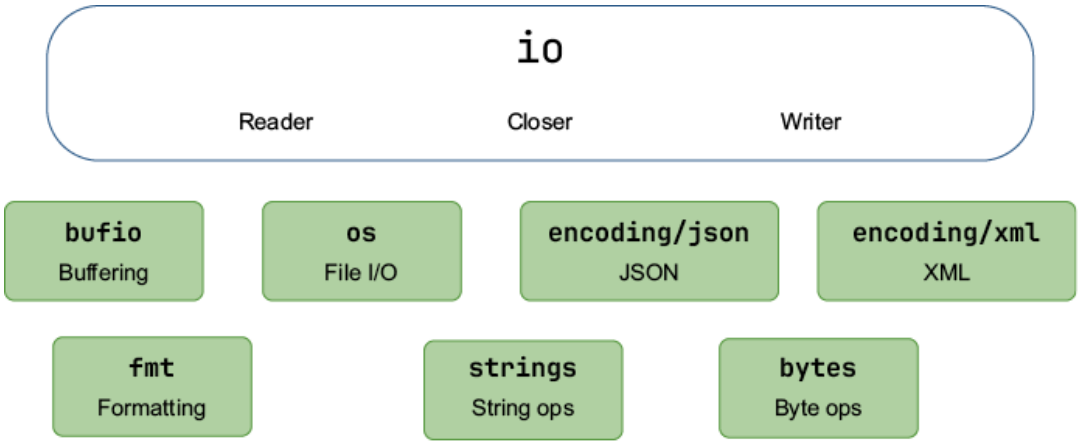


Figure 10.1 io.Reader and io.Writer interface composition with multiple package implementations

Each interface provides the abstraction required to handle many different inputs in the same way. To demonstrate this, we will start writing some of our functionality using the `Reader` and `Writer` interfaces, which will allow us to handle different inputs. We will do this in two ways: by reading a file and by reading from the operating system's input. We can then provide options to output to a file or to the screen, both of which implement the `Writer` interface. In doing this, we can write a flexible method for processing our input.

Listing 10.8 Core io interfaces

```
func process(w io.Writer, r io.Reader) error {
    return nil
}
```

Using these two interfaces, we could have this `process` function work as part of a server (reading and writing HTTP requests) or interact with cloud blob storage.

One of the most useful functions in the `fmt` package for outputting formatted text to different destinations is `fmt.Fprintf`. Unlike `fmt.Printf`, which writes to standard output, `fmt.Fprintf` lets you specify any `io.Writer` as the target, making it ideal for writing formatted strings to files, buffers, or even network connections. This flexibility is especially valuable when building CLI tools or logging systems.

Listing 10.9 Using fmt.Fprintf to write error and status messages

```
var writer io.Writer #1
writer := os.Stdout #2
fmt.Fprintf(writer, "Output written to: %s\n", outputFile) #3
```

#1 Empty writer, which can be a file or another output
#2 os.Stdout is a standard writer that will write to the screen.
#3 Fprintf will then write to that writer rather than the terminal.

Go's interfaces compose naturally, allowing you to build more complex functionality from simple building blocks. This means we can allow the process function to safely close the files it reads from or writes to. These interfaces are `ReadCloser` and `WriteCloser`.

The package also supports advanced composition patterns. For example, `io.MultiReader` and `io.MultiWriter` allow you to combine multiple readers or writers into a single stream, enabling sequential reading or simultaneous writing to multiple destinations. `io.TeeReader` is handy for duplicating data as it passes through a stream, such as logging input while processing it. To work with file segments or random access, `io.SectionReader` provides a way to read a specific section of a larger reader.

The `io` package provides several utility functions that work with any types implementing the core interfaces. Go used to have an `ioutil` package that is no longer used; instead, a number of methods, such as `ReadAll`, have been moved to the `io` package.

Listing 10.10 Using io.ReadAll() for simple data reading

```
func process(w io.Writer, r io.Reader) error {
    _, err := r.ReadAll(r) #1
    return err
}
```

#1 Reads all content from the reader

Beyond the core interfaces, the `io` package provides a variety of utility functions that make stream manipulation and composition straightforward. For example, `io.Copy` efficiently transfers data from any `Reader` to any `Writer`, making it ideal for copying files or streaming data between sources and sinks. Its variant, `io.CopyN`, lets you copy a

specific number of bytes, which is useful when you need to process only part of a stream.

Beyond the core interfaces, the `io` package provides a variety of utility functions that make stream manipulation and composition straightforward. For example, `io.Copy` efficiently transfers data from any `Reader` to any `Writer`, making it ideal for copying files or streaming data between sources and sinks. Its variant, `io.CopyN`, lets you copy a specific number of bytes, which is useful when you need to process only part of a stream.

Other utilities help with precise reading and writing. `io.ReadFull` ensures that a buffer is filled with exactly the number of bytes requested, returning an error if the stream ends prematurely. `io.LimitReader` creates a reader that stops after a set number of bytes, which is useful for processing fixed-length records or preventing resource exhaustion.

The `io.Reader` and `io.Writer` interfaces are arguably the most widely used interfaces in the Go ecosystem. You'll find them throughout the standard library and in virtually every well-designed third-party package. Their simplicity and composability make them the foundation of Go's I/O philosophy. When designing your own packages, you can follow this pattern by accepting `io.Reader` for input and `io.Writer` for output, making your code maximally flexible and reusable. Even better, when your functions need to read from sources that should be closed afterward, returning an `io.ReadCloser` (which combines `Reader` and `Closer`) is a recommended practice. This signals to callers that they're responsible for closing the resource, preventing resource leaks while maintaining the composability that makes Go's I/O system so powerful. By using these patterns, you can compose robust I/O pipelines and handle a wide variety of data sources and sinks with minimal effort.

10.4 The os package

The `os` package provides a platform-independent interface to operating system functionality. For CLI tools like our JSON/XML converter, it's essential for file operations, process management, and system interaction. This package serves as the bridge between your Go program and the underlying operating system, offering everything from file manipulation to environment variable access. This means that the application we are writing should function the same way on a Windows, Mac, or Linux machine, despite slight implementation differences between operating systems, such as file paths and data representation. Let's extend our CLI to use some operating system methods to enhance our application.

The first part of the application we should update is its use of *exit codes*. Proper exit codes are crucial for CLI tools because they communicate success or failure to the shell, scripts, and other programs that might invoke your application. Exit codes follow Unix conventions, where 0 indicates success and nonzero values indicate various types of errors.

Let's update our example to use proper exit status codes.

Listing 10.11 main.go: Adding a proper exit status

```
import (
    "flag"
    "fmt"
    "os"
)

func main() {
  ...

    if *help {
        fmt.Printf("Usage: %s [options] input_file output_file\n",
app)
        fmt.Println("  -help     Show help message")
        fmt.Println("  -banner   Display a custom banner message")
        os.Exit(0) #1
    }

    args := flag.Args()
    if len(args) < 2 {
        fmt.Println("Error: Please provide input and output file
s.\n")
        os.Exit(1) #2
    }

   ...

    // Process files...
    if err := processFiles(inputFile, outputFile); err != nil {
        fmt.Fprintf(os.Stderr, "Error: %v\n", err)
        os.Exit(1)
    }

    fmt.Println("Conversion completed successfully!")
}
```

#1 Valid exit asking for the help prompt
#2 An error occurs, so we set a nonzero value on exit.

These are some common exit codes:

- 0—Success

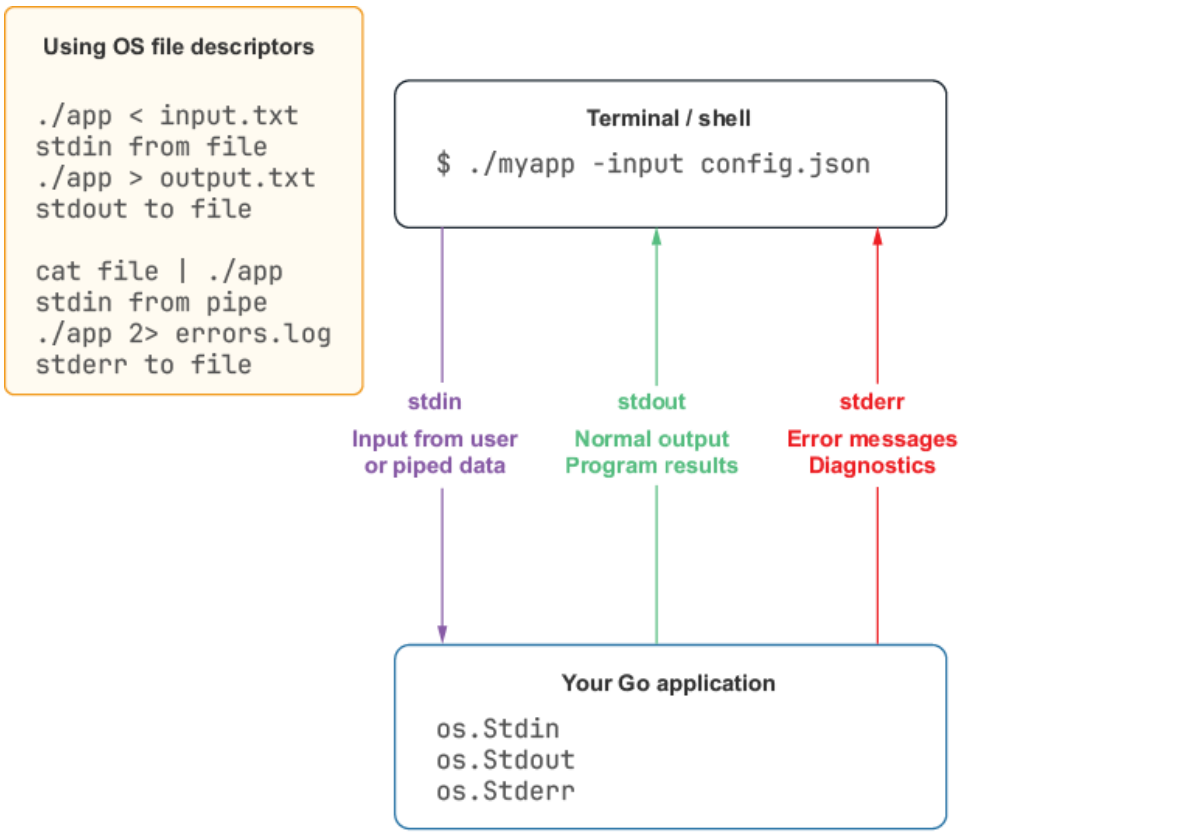
- `1`—General error
- `2`—Misuse of shell command
- `126`—Command invoked cannot execute (permission problem)
- `127`—Command not found

The key difference between `os.Exit()` and `return` is that `os.Exit()` terminates the program immediately, bypassing any deferred function calls, whereas `return` allows the normal function cleanup to occur.

Go provides access to the three standard file descriptors, which are how applications can interact with the system. This is done through `os.Stdin`, `os.Stdout`, and `os.Stderr`, which are technially files but are interactions with various parts of the system. Understanding when to use each is crucial for building well-behaved CLI tools:

- `os.Stdout`—Normal program output
- `os.Stderr`—Error messages and diagnostics
- `os.Stdin`—Input from the user or piped data

These three streams work together to create a clear separation between normal output, error reporting, and user input. Although most terminals display both `Stdout` and `Stderr` messages the same way on the screen, this separation is crucial for proper Unix-style command composition. It allows scripts and other programs to capture or redirect error messages separately from normal output, enabling robust error handling and logging in automation pipelines. Figure 10.2 illustrates how these standard streams connect your application to the terminal, showing the flow of data between your program and the user's environment.




```
fmt.Println("result") // Writes to stdout
fmt.Fprintln(os.Stderr, "err") // Writes to stderr
```

Figure 10.2 Standard input, output, and error streams and their relationship to the terminal

Notice that we refer to these as *file descriptors*. In Unix-based systems, everything is treated as a file, so reading input or writing output is similar to reading from or writing to a file. `os.Stdin`, `os.Stdout`, and `os.Stderr` provide this abstraction, making it easy to read from or write to what the operating system has determined to be standard. Even better, these descriptors implement the `io.Reader` and `io.Writer` interfaces, allowing us to use our `process` function.

The following listing shows how we can handle input from either a file or stdin.

Listing 10.12 Reading from stdin when no input file is provided

```
func main() {
    ...

    var input io.Reader
    var output io.Writer

    input = os.Stdin #1
    output = os.Stdout #2

    ...

    if err := process(input, output); err != nil {
        fmt.Println("Error: %v\n", err)
        os.Exit(1)
    }

}
```

#1 Stdin implements the io.Reader interface.
#2 Stdout implements the io.Writer interface.

In our application, we use the generic `io.Reader` and `io.Writer` interfaces to abstract input and output sources. This lets us seamlessly handle both standard input/output (such as the terminal) and file-based input/output. By assigning `os.Stdin` and `os.Stdout` to variables of type `io.Reader` and `io.Writer`, we can process data from the console. When working with files, we simply assign file handles (from `os.Open` or `os.Create`) to these same variables. This approach makes our processing functions flexible and reusable, enabling them to work with any data source or destination that implements the `io.Reader` and `io.Writer` interfaces. `os.File` implements both the reader and writer interfaces as well, so the two can be interchanged.

The `os` package provides several functions for opening files with different access patterns:

- `os.Open()`—Opens a file for reading only
- `os.Create()`—Creates or truncates a file for writing
- `os.OpenFile()`—Opens a file with custom flags and permissions

Listing 10.13 Using flags as guards on our application

```
func main() {

    inputFlag := flag.String("f", "", "Input file path")
    outputFlag := flag.String("o", "", "Output file path")

    ...

    var input io.Reader = os.Stdin
    var output io.Writer = os.Stdout

    if inputFlag != nil && *inputFlag != "" {
        file, err := os.Open(*inputFlag) #1
        if err != nil {
            fmt.Fprintf(os.Stderr, "Error opening input file: %v
\n", err)
            os.Exit(1)
        }
        defer file.Close()
        input = file
    }

    if outputFlag != nil && *outputFlag != "" {
        outFile, err := os.Create(*outputFlag) #2
        if err != nil {
            fmt.Fprintf(os.Stderr, "Error creating output file: %v
\n", err)
            os.Exit(1)
        }
        defer outFile.Close()
        output = outFile
    }

    // Pass to process function
    if err := process(input, output); err != nil {
        fmt.Fprintf(os.Stderr, "Error: %v\n", err)
        os.Exit(1)
    }
}
```

#1 Open will attempt to read the contents of a file at that location.
#2 Create will attempt to create an empty file at the location passed.

Before we process a file, it’s important to inspect its metadata to ensure that it’s suitable for our application. The `os.Stat()` function returns a `FileInfo` object containing details such as file size, modification time, permissions, and type. We can use this information to validate that the input is a regular file, not a directory or symlink, and that it meets any size or permission requirements.

Let’s update our main file-handling logic to include file metadata validation and safe output writing. This will ensure our CLI tool is robust and avoids common pitfalls, such as overwriting files or processing empty or invalid inputs.

Listing 10.14 main.go: Validate file metadata and use atomic writes

```
func openValidatedInput(path string) (io.ReadCloser, error) {
    info, err := os.Stat(path) #1
    if err != nil {
        if os.IsNotExist(err) { #2
            return nil, fmt.Errorf("input file %q does not exist",
path)
        }
        return nil, fmt.Errorf("error accessing input file: %w", er
r)
    }
    if !info.Mode().IsRegular() {
        return nil, fmt.Errorf("input file %q is not a regular fil
e", path)
    }
    if info.Size() == 0 {
        return nil, fmt.Errorf("input file %q is empty", path)
    }
    file, err := os.Open(path)
    if err != nil {
        return nil, fmt.Errorf("error opening input file: %w", err)
    }
    return file, nil
}
```

#1 Stat will try to grab information about a file at that given location.
#2 There are several distinct error types around OS operations.

Now we can integrate these helpers into our `main` function for a clean, consistent pattern.

```
func main() {
    inputFlag := flag.String("f", "", "Input file path")
    outputFlag := flag.String("o", "", "Output file path")
    flag.Parse()

    var input io.Reader = os.Stdin
    var output io.Writer = os.Stdout
    var tempFile *os.File #1

    if inputFlag != nil && *inputFlag != "" {
        file, err := openValidatedInput(*inputFlag)
        if err != nil {
            fmt.Fprintf(os.Stderr, "Error: %v\n", err)
            os.Exit(1)
        }
        defer file.Close() #2
        input = file #3
    }

    if outputFlag != nil && *outputFlag != "" {
        dir := filepath.Dir(*outputFlag)
        tf, err := os.CreateTemp(dir, "myapp-*.tmp") #4
        if err != nil {
            fmt.Fprintf(os.Stderr, "Error creating temp file: %v
\n", err)
            os.Exit(1)
        }
        tempFile = tf #5
        defer tempFile.Close() #6
        output = tempFile #7
    }

    // ... process(input, output) ...

    // If writing to a file, use atomic move
    if outputFlag != nil && *outputFlag != "" {
        if err := safeWriteFile(*outputFlag, tempFile); err != nil
{
            fmt.Fprintf(os.Stderr, "Error writing output: %v\n", er
r)
            os.Exit(1)
        }
```

```
    }
}
```

#1 Create a temporary file variable to hold our output data.
#2 Files should be closed; otherwise they can be left in a bad state.
#3 Files implement the io.Reader interface, so we set the opened file to our input.
#4 Create the temporary file for output.
#5 Set it to the tempFile created earlier for capturing output.
#6 Ensure the file is closed before exiting.
#7 Output will be written to the temporary file.

Temporary files and directories are essential for safe, atomic operations and intermediate processing. They can help you avoid data corruption and simplify cleanup. The `os` package provides several functions for managing temporary files and directories:

- `os.CreateTemp()`—Creates a new temporary file in a specified directory
- `os.MkdirTemp()`—Creates a new temporary directory
- `os.Remove()`—Deletes a file or an empty directory
- `os.RemoveAll()`—Recursively deletes a directory and all its contents

For cleanup, use `os.Remove()` to delete individual files or empty directories, and use `os.RemoveAll()` for recursive removal of directories and their contents:

```
err := os.Remove("oldfile.txt") #1
err := os.RemoveAll("tempdir") #2
```

#1 Remove a file.
#2 Remove a directory and all contents.

By following these patterns, CLI tools can safely manage temporary resources and ensure proper cleanup, reducing the risk of leaving behind unwanted files or corrupting existing data.

Flags allow you to customize an application's behavior by passing specific options or arguments at runtime. Although flags are useful for dynamic configuration, you may not want to pass the same information every time you run the application. For these cases, environment variables provide a convenient alternative.

Environment variables are key/value pairs defined in the operating system that applications can access at runtime. They let you configure applications without modifying the code or relying solely on command-line flags. This approach is particularly useful for setting defaults that flags can override, managing sensitive information such as API keys, or configuring behavior across different environments (e.g., development, staging, production).

In Go, you can access environment variables using the `os` package. The `os.Getenv` function retrieves the value of an environment variable, while `os.Setenv` lets you set one programmatically. You can also use `os.LookupEnv` to check whether a variable is set without defaulting to an empty string.

Our CLI application currently uses a flag to set an application name; let's instead use an environment variable.

Listing 10.15 main.go: Adding environment variable support

```
import (
    "flag"
    "fmt"
    "os"
)

func main() {
    app := "myapp"
    if envAppName, exists := os.LookupEnv("APP_NAME"); exists { #1
        app = envAppName
    }

    banner := os.Getenv("BANNER") #2
    if banner != "" {
        fmt.Println(banner)
    }

    help := flag.Bool("help", false, "Show help message")
    flag.Parse()

...
    fmt.Printf("%s - tools to convert between json and xml\n", app)
}
```

#1 LookupEnv will give an indication if an environment variable is set.
#2 GetEnv will return an empty string if the value is not set.

Here, we can manipulate the application without changing the code because the values are pulled from the hosted environment. This approach lets us add another level of input to the system beyond flags.

Although we've explored many aspects of the `os` package, some additional features are worth mentioning. The package provides tools for process control, such as `os.StartProcess` and `os.FindProcess`, which are useful for spawning and interacting with child processes. The `os/signal` package, often used alongside `os`, lets you handle operating

system signals like `SIGINT` and `SIGTERM`, enabling graceful application shutdowns.

The `os` package is an essential component of Go's standard library, offering a comprehensive set of tools for interacting with the operating system. It provides platform--independent abstractions for file operations, process management, and environment handling, enabling developers to build reliable and portable applications. By using the `os` package, you can create cross-platform tools that integrate smoothly with the underlying system, ensuring both flexibility and robustness.

10.5 The encoding package

Working with structured data is fundamental for many modern applications. Whether you're building APIs, processing configuration files, or exchanging data between systems, you need reliable ways to convert between raw bytes and meaningful data structures. Go's `encoding` package and its subpackages provide robust, efficient tools for handling these transformations across a wide variety of formats.

The `encoding` package serves as the foundation for data serialization and deserialization in Go. It defines core interfaces, such as `BinaryMarshaler` and `TextMarshaler`, that establish patterns for converting Go values to and from different representations. Built on this foundation, specialized subpackages handle specific formats: `encoding/json` for JSON data, `encoding/xml` for XML documents, `encoding/csv` for comma-separated values, `encoding/base64` for base64 encoding, and several others for binary formats and specialized encodings.

These encoding packages integrate seamlessly with Go's I/O system. Because they work directly with byte slices and implement the `io.Reader` and `io.Writer` interfaces, you can easily compose encoding operations with file I/O, network communication, or in-memory buffers. This composability makes it straightforward to build flexible data processing pipelines that read from various sources, transform data between formats, and write results to different destinations.

For our JSON and XML validation and conversion tool, the encoding packages are essential. They will allow us to parse input data, validate its structure against Go types, transform it between formats, and output properly formatted results, all with minimal code and excellent performance.

10.5.1 JSON processing

JSON (JavaScript Object Notation) has become the de facto standard for data exchange in modern applications, particularly for web APIs and configuration files. Go's `encoding/json` package provides comprehensive support for working with JSON data, from simple parsing to complex custom serialization logic. Understanding how to use this package effectively is crucial for building tools that interact with JSON data.

To demonstrate JSON processing in practice, let's work with a realistic example: a server configuration file. This common use case requires validating that configuration data matches expected formats and contains sensible values. The following listing shows a typical configuration expressed as JSON.

Listing 10.16 config.json: Configuration defined in JSON format

```
{
    "name": "myapp",
    "version": "1.0.0",
    "debug": false,
    "port": 8080,
    "host": "localhost"
}
```

This configuration includes various data types, including strings, Booleans, and integers, that represent common server settings. To work with this data in Go, we need to define a corresponding struct that mirrors this structure. The `encoding/json` package uses struct field tags to map JSON keys to Go struct fields, allowing for flexible naming and transformation during marshaling and unmarshaling.

Struct tags are specially formatted strings attached to struct fields that provide metadata about how those fields should be processed. For JSON, tags specify the JSON key name that corresponds to each field. Importantly, struct fields must be exported (capitalized) to be accessible to the `encoding/json` package for marshaling and unmarshaling. Here's how we define a struct to represent our configuration.

Listing 10.17 Defining a configuration structure with JSON tags

```
// Config represents the structure of a JSON configuration file
type Config struct {
    Name      string `json:"name"` #1
    Version   string `json:"version"`
    Debug     bool   `json:"debug"` #2
    Port      int    `json:"port"` #3
    Host      string `json:"host,omitempty"` #4
    LogLevel  string `json:"-"` #5
}
```

#1 The struct tag matching the input data will be mapped to the field.
#2 The parser will attempt to parse the raw value into the given type.
#3 This includes integers as well as other types.
#4 Do not output the content if the value is empty.
#5 Have JSON parser skip adding the value.

With our struct defined, we can now parse JSON data into Go values using the `json.Unmarshal` function. This function takes a byte slice containing JSON data and a pointer to a struct (or other Go value) where the parsed data should be stored. If the JSON structure doesn't match the target type–for example, if a field has the wrong type or required fields are missing–`Unmarshal` returns an error, making it easy to detect malformed input.

Let's create a validation function that demonstrates this unmarshaling process. This function will accept a template type identifier and raw JSON bytes, and it then will attempt to parse the data into the appropriate struct. This pattern is useful when you need to validate different types of configurations or support multiple data schemas.

Listing 10.18 main.go: Template validation function

```
func validate(template string, data []bytes) error {
    var st any
    switch template {
    case "config", "configuration":
        st := &Config{}
    default:
      return fmt.Errorf("unknown template: %s", template)
    }
    return json.Unmarshal(st, data)
}
```

The `Unmarshal` function performs several important operations beyond simple parsing. It validates that JSON types match the expected Go types, it handles nested structures recursively, and it applies any custom unmarshaling logic defined on your types. If the data successfully unmarshals, you have a validated, type-safe Go value ready to use in your application. If not, the returned error provides information about what went wrong during unmarshaling.

The counterpart to `Unmarshal` is `Marshal`, which converts Go values into JSON-formatted byte slices. This is essential when you need to output configuration data, serialize application state, or generate JSON responses. The `Marshal` function traverses your Go data structures and produces well-formed JSON, respecting struct tags and applying proper formatting.

Beyond basic marshalling and unmarshalling, the `encoding/json` package supports custom serialization logic through the `Marshaler` and `Unmarshaler` interfaces. By implementing `MarshalJSON` or `UnmarshalJSON` methods on your types, you can control exactly how values are converted to and from JSON. This is particularly valuable for validation,

data transformation, or handling complex types that don't have a straightforward JSON representation.

For example, we might want to validate that the `Host` field contains a properly formatted URL during unmarshaling. Custom unmarshaling allows us to add this validation logic directly in our type definition, ensuring that any time we unmarshal a `Config`, the host value has already been validated.

```
// UnmarshalJSON validates the Host field to ensure it is a valid U
RL
func (c *Config) UnmarshalJSON(b []byte) error {
    temp := struct { #1
        Name    string `json:"name"`
        Version string `json:"version"`
        Debug   bool   `json:"debug"`
        Port    int    `json:"port"`
        Host    string `json:"host"`
    }{}

    if err := json.Unmarshal(b, &temp); err != nil { #2
        return err
    }

    // Validate the Host field
    if _, err := url.ParseRequestURI(temp.Host); err != nil { #3
        return fmt.Errorf("invalid Host URL: %v", err)
    }

    *c = Config(temp) #4
    return nil
}
```

#1 To prevent a loop, we need to create a temporary struct that matches a similar signature.
#2 Read the data into the temporary structure.
#3 Pull the host value out to validate.
#4 If everything is okay, set the values for configuration from the temporary struct.

Notice the pattern used here: we create a temporary anonymous struct to avoid recursion (calling `json.Unmarshal` on a type from within its own `UnmarshalJSON` method would create an infinite loop because it would call its own function), we perform our validation or transformation, and then we assign the result back to the receiver. This technique gives us complete control over the unmarshaling process while still using the standard JSON parsing machinery.

Similarly, custom marshalling with `MarshalJSON` allows you to transform data during output. A common use case is redacting sensitive information, such as passwords or API keys, before serializing data for logging or external transmission. Here's an example that redacts the host value when marshalling a configuration:

```
func (c Config) MarshalJSON() ([]byte, error) {
    // Use an anonymous struct to avoid recursion
    temp := &struct {
        Name    string `json:"name"`
        Version string `json:"version"`
        Debug   bool   `json:"debug"`
        Port    int    `json:"port"`
        Host    string `json:"host"`
    }{
        Name:    c.Name,
        Version: c.Version,
        Debug:   c.Debug,
        Port:    c.Port,
        Host:    "[REDACTED]", // Redact the host value
    }

    return json.Marshal(temp)
}
```

These custom marshalling and unmarshalling methods demonstrate the flexibility of Go's encoding system. By implementing these interfaces, you can add validation,

transformation, sanitization, or any other logic you need while maintaining compatibility with all the standard JSON processing functions.

10.5.2 XML processing

Although JSON has become predominant for many use cases, XML remains important for legacy systems, enterprise integrations, and certain domains, such as configuration management and document processing. Go's `encoding/xml` package provides comprehensive XML support with patterns very similar to those used for JSON processing. Understanding both formats and how to convert between them is valuable for building flexible data processing tools.

XML processing in Go follows the same fundamental pattern as for JSON: you define Go structs with field tags and then use `Marshal` and `Unmarshal` functions to convert between XML bytes and Go values. But XML has additional complexities compared to JSON because it supports attributes, namespaces, mixed content, and more elaborate document structures. The `encoding/xml` package handles these features through extended struct tag syntax and specialized field types.

For our configuration structure, we can add XML support simply by including additional struct tags. This allows the same Go type to work with both JSON and XML representations, which is exactly what our conversion tool needs:

```
.Defining a configuration structure with XML tags
[source,go,linenums]
----
type Config struct {
    XMLName   xml.Name `xml:"configuration"` // Root element name
    Name      string   `json:"name" xml:"name,attr"` // XML attribu
te
    Version   string   `json:"version" xml:"version,attr"` // XML a
ttribute
    Namespace string   `xml:"xmlns,attr,omitempty"` // XML namespac
e
    Debug     bool     `json:"debug" xml:"debug"`
    Port      int      `json:"port" xml:"port"`
    Host      string   `json:"host" xml:"host,omitempty"`
    Comment   string   `xml:",comment"` // XML comment
}
----
```

The dual tags in our struct enable seamless conversion between formats. Figure 10.3 illustrates how the struct tags map Go fields to both JSON keys and XML elements, allowing a single type definition to support multiple serialization formats.

Go struct definition

```
type Config struct {
    Name string          `json:"name" xml:"name"`
    Version string       `json:"version" xml:"version"`
    Debug bool           `json:"debug" xml:"debug"`
    Port int             `json:"port" xml:"port"`
    Host string          `json:"host" xml:"host"`
}
```

JSON representation

```
{
    "name": "myapp",
    "version": "1.0.0",
    "debug": false,
    "port": 8080,
    "host": "localhost"
}
```

XML representation

```
<config>
    <name>myapp</name>
    <version>1.0.0</version>
    <debug>false</debug>
    <port>8080</port>
    <host>localhost</host>
</config>
```

Figure 10.3 Struct tags enable marshalling to both JSON and XML formats.

With both JSON and XML tags in place, our `Config` struct can be marshalled to or unmarshaled from either format. This dual-format support is the foundation of our conversion tool, allowing us to read data in one format and output it in another by simply choosing which marshalling function we use.

The conversion process involves parsing the input format into our Go struct and then serializing that struct in the output format. This approach ensures that the data structure is validated and normalized during conversion. The function in the following listing demonstrates bidirectional conversion.

Listing 10.19 Converting between JSON and XML formats

```
func convertConfig(data []byte, outputFormat string) ([]byte, erro
r) {
    // First, parse as JSON
    var cfg Config #1
    err := json.Unmarshal(data, &cfg) #2
    if err != nil {
        return nil, fmt.Errorf("error parsing JSON: %w", err)
    }

    // Convert to the requested output format
    var output []byte
    switch outputFormat {
    case "xml":
        output, err = xml.MarshalIndent(cfg, "", "  ")
        if err != nil {
            return nil, fmt.Errorf("error formatting XML: %w", err)
        }
    default:
        output, err = json.MarshalIndent(cfg, "", "  ")
        if err != nil {
            return nil, fmt.Errorf("error formatting JSON: %w", er
r)
        }
    }

    return output, nil
}
```

#1 Initialize an empty variable for config.
#2 Verify that the incoming JSON is valid.

This conversion function demonstrates several important techniques. First, it uses the intermediate Go struct as a validated data representation, ensuring that only properly structured data is converted. Second, it uses `MarshalIndent` for both formats, which produces nicely formatted, human-readable output with consistent indentation, which is much better than the compact output from `Marshal` for configuration files. Finally, it provides clear error messages

that distinguish between parsing and formatting failures, making debugging easier.

10.6 The bufio package

While the `io` package provides the fundamental interfaces for reading and writing data, the `bufio` package builds on these interfaces to add buffering capabilities. Buffering improves performance by reducing the number of system calls when reading or writing data. Instead of making a system call for every single read or write operation, buffered I/O accumulates data in memory and processes it in larger chunks. This is particularly important when working with files, network connections, or any I/O operation in which the overhead of individual reads and writes can affect performance.

The performance difference between buffered and unbuffered I/O can be dramatic, especially when processing large files or streams. Figure 10.4 illustrates this difference, showing how buffered I/O batches multiple operations into fewer system calls, significantly reducing overhead and improving throughput.

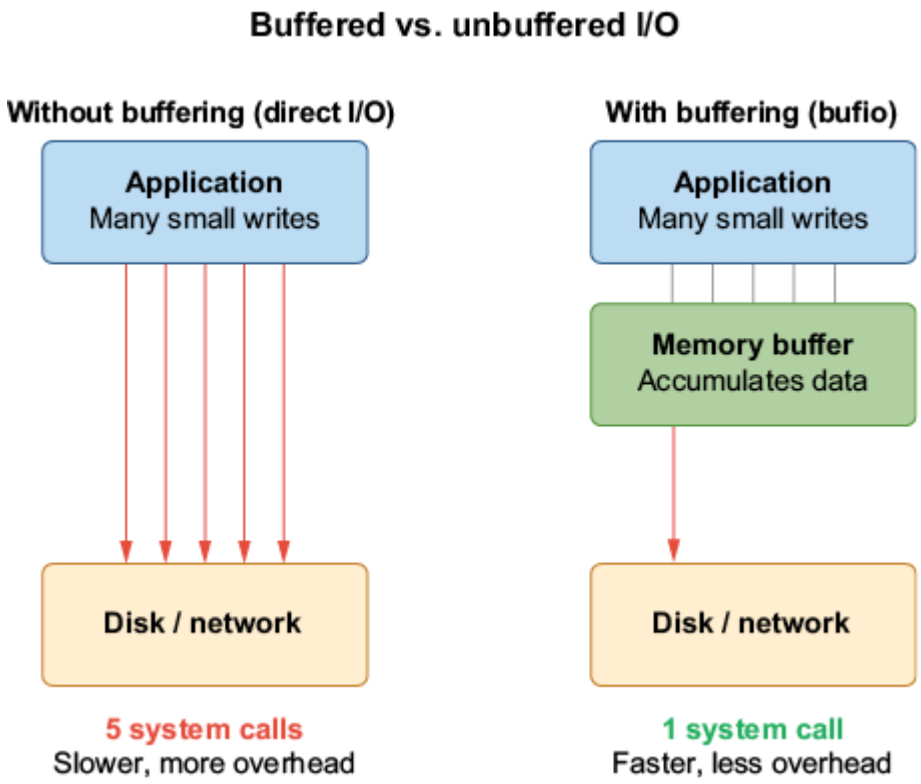


Figure 10.4 Comparison of unbuffered and buffered I/O, showing performance benefits

The `bufio` package is especially useful in our JSON/XML converter tool when we need to read input line by line, scan for specific patterns, or write output efficiently. It provides several types that wrap existing `io.Reader` and `io.Writer` implementations to add buffering functionality: `bufio.Reader` for buffered reading, `bufio.Writer` for buffered writing, and `bufio.Scanner` for convenient token-based reading.

The `bufio.Reader` type wraps any `io.Reader` and adds a buffer layer. This is particularly useful when you need to read data in small chunks or perform look-ahead operations without the performance penalty of multiple system calls. A common use case for `bufio.Reader` is reading data line by line. This is especially relevant for our CLI tool if we want to process configuration files or validate JSON/XML that might

contain multiple records. The `ReadString` and `ReadLine` methods make this straightforward.

Listing 10.20 Reading input line by line

```
func processLineByLine(r io.Reader) error {
    scanner := bufio.NewReader(r) #1

    for {
        line, err := scanner.ReadString('\n')
        if err != nil {
            if err == io.EOF {
                // Process the last line if it doesn't end with a n
ewline
                if len(line) > 0 {
                    fmt.Printf("Processing: %s\n", line)
                }
                break
            }
            return fmt.Errorf("error reading input: %w", err)
        }

        // Process each line
        fmt.Printf("Processing: %s", line)
    }

    return nil
}
```

#1 Data can gradually be read from the source, allowing for more efficient processing.

The `bufio.Reader` also provides a method for peeking at data without consuming it (`Peek`), which can be useful for detecting file formats or validating input before processing. For example, you might peek at the first few bytes to determine whether the input is JSON (starts with `{` or `[`) or XML (starts with `<`).

Listing 10.21 Detecting the input format by peeking

```
func detectFormat(r io.Reader) (string, error) {
    br := bufio.NewReader(r)

    firstByte, err := br.Peek(1) #1
    if err != nil {
        return "", fmt.Errorf("error peeking at input: %w", err)
    }

    switch firstByte[0] {
    case '{', '[':
        return "json", nil
    case '<':
        return "xml", nil
    default:
        return "", fmt.Errorf("unknown format")
    }
}
```

#1 Peek at the first byte without consuming it.

We can use the detect format function in the preceding code to determine the flow of data through the detection and processing workflow instead of depending on a parsing error.

Just as `bufio.Reader` improves read performance, `bufio.Writer` enhances write performance by buffering output before writing it to the underlying `io.Writer`. This is especially important when writing many small pieces of data, as it reduces the number of system calls and improves throughput.

Creating a buffered writer is similar to creating a buffered reader, as shown in the following listing.

Listing 10.22 Using buffered writing for efficient output

```
func writeOutput(w io.Writer, data []string) error {
    bw := bufio.NewWriter(w) #1

    for _, line := range data {
        _, err := bw.WriteString(line) #2
        if err != nil {
            return fmt.Errorf("error writing output: %w", err)
        }
        bw.WriteString("\n")
    }

    // Important: Flush remaining buffered data
    if err := bw.Flush(); err != nil { #3
        return fmt.Errorf("error flushing output: %w", err)
    }

    return nil
}
```

#1 Create a buffered writer from an existing writer.
#2 Write a string to the buffer.
#3 Flush anything in the buffer to the reader.

The critical detail when using `bufio.Writer` is the `Flush()` call. The buffered writer holds data in memory until the buffer is full or until `Flush()` is called explicitly. If you forget to flush, your data might not be written to the underlying writer, leading to incomplete output. This is particularly important when writing to files or network connections, where you need to ensure that all data is transmitted.

For our CLI tool, we could use buffered writing when outputting converted JSON or XML to improve performance, especially when generating large output files.

Listing 10.23 Buffered output in the conversion process

```
func process(r io.Reader, w io.Writer) error {
    bw := bufio.NewWriter(w) #1
    defer bw.Flush() #2

    // Read all input data
    data, err := io.ReadAll(r) #3
    if err != nil {
        return fmt.Errorf("error reading input: %w", err)
    }

    // Detect format by peeking at the data
    br := bytes.NewReader(data) #4
    format, err := detectFormat(br) #5
    if err != nil {
        return fmt.Errorf("error detecting format: %w", err)
    }

    // Convert based on detected format
    var convertedData []byte
    if format == "json" {
        convertedData, err = convertConfig(data, "xml")
    } else {
        convertedData, err = convertConfig(data, "json")
    }

    if err != nil {
        return fmt.Errorf("error converting data: %w", err)
    }

    _, err = bw.Write(convertedData) #6
    if err != nil {
        return fmt.Errorf("error writing output: %w", err)
    }

    return nil
}
```

#1 Create buffered writer.
#2 Ensure flush even if errors occur.
#3 Read all data from input into memory.
#4 Create a bytes.Reader to enable format detection without consuming the data.

#5 Pass the reader to detectFormat as defined earlier.
#6 Write converted data efficiently.

The `bufio` package demonstrates Go's philosophy of building higher-level abstractions on top of simple, composable interfaces. By wrapping `io.Reader` and `io.Writer` with buffering capabilities, it provides significant performance improvements while maintaining full compatibility with the standard I/O interfaces. For CLI tools and applications that process text or structured data, understanding and using `bufio` effectively can make a substantial difference in code clarity and performance.

10.7 The regexp package

Regular expressions provide powerful pattern-matching capabilities for text processing and validation. The `regexp` package offers compiled regular expression support with good performance, which is essential for validating configuration fields, sanitizing input, and extracting structured data from text.

Regular expressions (often abbreviated as "regex" or "regexp") are patterns that describe sets of strings. They use special syntax to define matching rules: literal characters match themselves, while metacharacters like `.` (any character), `*` (zero or more), `+` (one or more), and `?` (zero or one) provide flexible matching. These operators are known as *greedy* selectors because they match as much text as possible while still satisfying the pattern. Alternatively, you can append `?` to these operators (for example, `*?` or `+?`) to make them *lazy*, which causes them to match as little text as possible while still satisfying the pattern. Character classes like `[a-z]` match any character within the specified range, and anchors like `^` (start of string) and `$` (end of string) help enforce exact patterns.

Table 10.2 outlines some basic regex expressions. Go's `regexp` package implements the RE2 syntax, which guarantees linear-time performance and avoids some pitfalls of other regex implementations.

Table 10.2 Common regular expression metacharacters

Metacharacter	Description	Example
`.`	Any character	`a.c` matches "abc", "a1c"
`*`	Zero or more	`ab*c` matches "ac", "abc", "abbc"
`+`	One or more	`ab+c` matches "abc", "abbc" (not "ac")
`?`	Zero or one	`ab?c` matches "ac", "abc"
`[a-z]`	Character class	`[a-z]` matches any lowercase letter
`^`	Start of string	`^abc` matches "abc" at start only
`$`	End of string	`abc$` matches "abc" at end only

For those new to regular expressions or needing a refresher, https://regex101.com/ provides an interactive playground for testing patterns and learning the syntax. The Go regexp package documentation at https://pkg.go.dev/regexp also includes comprehensive examples and syntax references.

In the CLI tool we have been building for configuration validation, we'll use regular expressions to ensure that field

values conform to expected formats, such as alphanumeric names, semantic versioning strings, and valid hostnames.

Regular expressions in Go are compiled into `*regexp.Regexp` objects that can be reused efficiently. Our configuration validator demonstrates several common validation patterns. For example, we will want to verify that the name follows a certain format.

Listing 10.24 Validating configuration fields with regex

```
import "regexp"

// ValidationError represents a validation error with context
type ValidationError struct {
    Field  string
    Reason string
}

func validateConfig(cfg *Config) []ValidationError {
    var validationErrors []ValidationError

    // Validate name: must be non-empty and alphanumeric with hyphe
ns/underscores
    namePattern := regexp.MustCompile(`^[a-zA-Z0-9_-]+$`) #1
    if cfg.Name == "" {
        validationErrors = append(validationErrors, ValidationError
{
            Field:  "name",
            Reason: "cannot be empty",
        })
    }
    if !namePattern.MatchString(cfg.Name) { #2
        validationErrors = append(validationErrors, ValidationError
{
            Field:  "name",
            Reason: "must contain only alphanumeric characters, hyp
hens, or underscores",
        })
    }

    return validationErrors
}
```

#1 Compile the regex expression into a struct to match potential strings.
#2 Use the pattern to run against a string to match.

The `regexp` package is an essential tool for text validation and pattern matching in Go. Although they're powerful, regular expressions can become complex and difficult to maintain. Use them when sophisticated pattern matching is

genuinely required, but prefer simpler string operations from the `strings` package for basic tasks such as prefix or suffix checks and substring searches. For performance, always precompile patterns using `regexp.MustCompile()` when they will be used repeatedly rather than compiling them on each use. As a best practice, document each regex pattern with a comment explaining what it matches, and include an example input. This will help both other developers and your future self understand and maintain the code.

10.8 The strings package

The `strings` package provides functions for manipulating UTF-8 encoded strings. It's one of the most frequently used packages in Go, offering utilities for searching, replacing, splitting, joining, and transforming strings. Unlike languages where strings have built-in methods, Go provides these operations as package functions, consistent with the language's design philosophy.

The `strings` package provides functions for examining and comparing strings. Common operations include checking if a string contains a substring, identifying prefixes or suffixes, and performing case-insensitive comparisons. These functions are fundamental for input validation and format detection in CLI tools.

Listing 10.25 String operations for normalizing names for files

```
import "strings"

func normalizeConfigName(name string) string {
    name = strings.TrimSpace(name) #1

    name = strings.ToLower(name) #2

    name = strings.ReplaceAll(name, " ", "-") #3

    return name
}
```

#1 Remove leading and trailing whitespace.
#2 Convert to lowercase for consistency.
#3 Replace spaces with hyphens.

When processing user input or file paths, it’s often necessary to detect the format or type based on string patterns. The `strings` package makes this straightforward by providing functions to check prefixes, suffixes, and the presence of substrings, allowing you to build intelligent format-detection logic.

Listing 10.26 Additional string operations for validation in CLI tools

```
func validateInputFormat(input string) (string, error) {
    input = strings.TrimSpace(input)

    if strings.HasSuffix(input, ".json") { #1
        return "json", nil
    }

    if strings.HasSuffix(input, ".xml") {
        return "xml", nil
    }

    // Check content for format detection - XML first since it's mo
re specific
    if strings.Contains(input, "<?xml") || strings.HasPrefix(input,
"<") { #2
        return "xml", nil
    }

    // Check for JSON last to avoid false positives from XML contai
ning braces
    if strings.HasPrefix(input, "{") || strings.HasPrefix(input, "
[") { #3
        return "json", nil
    }

    return "", fmt.Errorf("unknown format")
}
```

#1 HasSuffix will check to match the end part of the string.
#2 Contains checks for XML declaration; HasPrefix checks for opening tag.
#3 HasPrefix ensures content starts with JSON syntax, avoiding false matches.

The `strings` package offers essential functions for splitting and joining strings, making it invaluable for input and formatting output. Its simple, well-named functions are highly composable, allowing you to build sophisticated text manipulation logic from basic operations.

10.9 The strconv package

The `strconv` package handles conversions between strings and other basic data types, such as integers, floats, and Booleans. This capability is crucial for CLI tools that receive text input from command-line arguments or files and need to convert it to typed values for processing. Although we've been using `fmt.Sprintf` to convert values to strings, `strconv` provides more control and better performance for type conversions. It also offers comprehensive error handling, which is important when parsing user input that might be malformed.

The most common use case for `strconv` is parsing numeric strings into typed values, with proper error handling for invalid input. We'll add parsing and validation for our input using a custom parser we will create.

Listing 10.27 Parsing numeric configuration values

```
import "strconv"

func parsePortNumber(portStr string) (int, error) {
    port, err := strconv.ParseInt(portStr, 10, 32) #1
    if err != nil {
        return 0, fmt.Errorf("invalid port number %q: %w", portStr,
err)
    }

    // Validate port range
    if port < 1 || port > 65535 {
        return 0, fmt.Errorf("port must be between 1 and 65535, got
%d", port)
    }

    return int(port), nil
}
```

#1 ParseInt parameters: string, base (0 for auto-detect), bit size

The following listing demonstrates how to parse string values into Boolean types using the `strconv` package. The `ParseBool` function is particularly useful when reading configuration values from environment variables, configuration files, or command-line arguments, where values are initially represented as strings.

Note that `ParseBool` is flexible in what it accepts as valid Boolean values. It recognizes "1", "t", "T", "true", "TRUE", and "True" as true, and "0", "f", "F", "false", "FALSE", and "False" as false. Any other value returns an error.

Listing 10.28 Parsing bool configuration values

```
func parseDebugFlag(debugStr string) (bool, error) {
    debug, err := strconv.ParseBool(debugStr) #1
    if err != nil {
        return false, fmt.Errorf("invalid boolean value %q: %w",
            debugStr, err)
    }
    return debug, nil
}
```

#1 ParseBool accepts: "1", "t", "T", "true", "TRUE", and "True".

We've parsed strings into typed values, but the reverse operation, converting typed values back into strings, is equally important for generating output, creating log messages, and building user-facing displays. The `strconv` package provides dedicated formatting functions for each type that offer better performance and more control than generic string conversion.

The key advantage of using `strconv` for formatting is that you can specify exactly how values should be represented. For integers, you can choose the numeric base (decimal, hexadecimal, binary). For floating-point numbers, you can control the format style and precision. This level of control

is essential when you need consistent, predictable string output.

Listing 10.29 Formatting values for display

```
func formatConfigValue(key string, value any) string {
    switch v := value.(type) {
    case int:
        return key + ": " + strconv.Itoa(v) #1
    case int64:
        return key + ": " + strconv.FormatInt(v, 10) #2
    case bool:
        return key + ": " + strconv.FormatBool(v) #3
    case float64:
        return key + ": " + strconv.FormatFloat(v, 'f', -1, 64) #4
    default:
        return key + ": " + fmt.Sprintf("%v", v)
    }
}
```

#1 Itoa is the most common integer-to-string conversion (base 10).
#2 FormatInt allows specifying the base: 2 for binary, 16 for hex, etc.
#3 FormatBool converts boolean values to their string representation.
#4 FormatFloat offers fine control over format style (‘f’, ‘e’, ‘g’) and precision.

The `FormatFloat` function deserves special attention because it provides the most options. The format parameter controls the notation style: `'f'` for decimal notation (e.g., “123.456”), `'e'` for scientific notation (e.g., “1.23456e+02”), and `'g'` for automatic selection of the most compact representation. The precision parameter specifies how many digits to include after the decimal point, or -1 to use the smallest number of digits needed to represent the value exactly.

The `strconv` package is essential for robust CLI tools. Unlike `fmt.Sprintf`, it provides dedicated functions for each conversion type, with better performance and more detailed error information. When parsing user input, always use

`strconv` functions and handle errors explicitly so you can provide helpful feedback when input is malformed.

10.10 Scratching the surface

Throughout this chapter, we've built a complete JSON/XML validation and conversion CLI tool by progressively integrating packages from Go's standard library. Each section added new capabilities, from parsing command-line arguments with `flag`, to reading and writing files with `os` and `io`, to converting between structured data formats with `encoding/json` and `encoding/xml`, to validating input with `regexp` and `strings`. Figure 10.5 shows how all these components fit together in our final application architecture, illustrating the flow of data from command-line input through validation and conversion to final output.

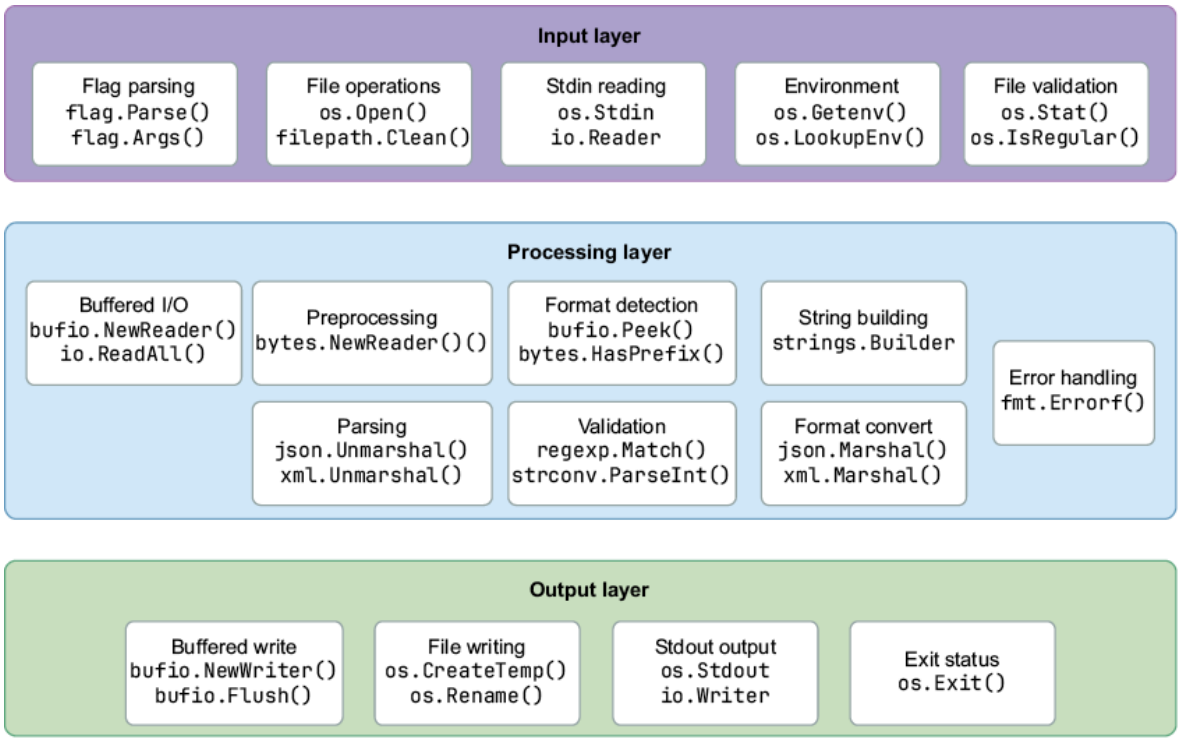


Figure 10.5 The complete CLI tool architecture, showing all layers and components

We've completed our CLI tool, but this is just the beginning. Beyond the packages we've covered, the Go standard library contains many more specialized tools for various programming tasks. You can explore the complete standard library at https://pkg.go.dev/std, which provides comprehensive documentation for every package along with examples and usage patterns. Each package page includes detailed descriptions, function signatures, and often runnable examples that you can modify and test directly in your browser. As you encounter new programming challenges, always check to see whether the standard library has a solution. Otherwise, you'll need to use a community package, as we'll demonstrate in the next chapter.

The Go standard library embodies the language's philosophy: simplicity, clarity, and practicality. It provides everything you need to build production-quality applications, from simple scripts to complex distributed systems. Master these packages, and you'll be well-equipped to tackle any Go project that comes your way.

Summary

- The `fmt` package provides formatted I/O functions for printing, formatting strings, and reading input with type conversion.
- The `flag` package simplifies command-line argument parsing and supports Boolean, string, and integer flags, as well as positional arguments.
- The `io` package defines fundamental interfaces (`Reader`, `Writer`, `Closer`) that enable flexible, composable I/O operations across files, networks, and memory.
- The `os` package provides platform-independent file operations, environment variables, and process management for cross-platform applications.

- The `encoding/json` and `encoding/xml` packages enable marshaling and unmarshaling structured data with struct tags for field mapping.
- Custom `MarshalJSON` and `UnmarshalJSON` methods provide control over serialization for validation, transformation, and sanitization.
- The `bufio` package adds buffering to I/O operations, significantly improving performance by reducing system calls.
- The `regexp` package offers compiled regular expression support for pattern matching, validation, and text extraction.
- The `strings` package provides essential string-manipulation functions, including searching, trimming, case conversion, and splitting.
- The `strconv` package handles type-safe conversions between strings and basic types with detailed error handling.

11 Working with larger projects

This chapter covers

- Organizing code with packages and modules
- Designing a shared domain model using interfaces
- Coordinating multiple modules during development with workspaces
- Building a multimodule application

Throughout this book, we have built several small applications to demonstrate various parts and patterns of the Go programming language. For the most part, these applications have been self-contained, relying on the standard library packages we explored in previous chapters and custom packages we wrote ourselves. In everyday development, though, this is rarely the case. Developers routinely use shared libraries to support their projects. Unlike the standard library, these external libraries are managed by individuals or small teams rather than undergoing a standardization process that requires community consensus.

Beyond external libraries, applications interact with external services through clients and APIs. Documenting and managing these dependencies can be challenging but is essential in most workflows.

In chapter 9, we started an application that used an external dependency to parse RSS feeds. Now we will build on that foundation by creating a multimodule architecture

that separates concerns and allows different components to evolve independently. We'll explore packages, modules, and workspaces to keep our code organized and reusable. Specifically, we'll separate our RSS reader into four modules: a shared domain module (domain model), a storage module (persistent implementation using bbolt), a REST API service, and a background worker that fetches feeds asynchronously.

Software is complicated because it rarely exists in a vacuum, but Go keeps things as simple as possible by providing guidance and tools for developing organized, clean code. As developers, it's our responsibility to keep our projects maintainable.

Go has several layers for organizing code. At the most basic level are *packages*. Collections of `packages` can be grouped into *modules*, either as libraries for others to use or as installable applications that can be downloaded and installed directly on your local system. These `modules` can then have various *versions* as the module changes, allowing for continued compatibility. Finally, multiple modules can be managed within a single parent module known as a *workspace*. This structure supports more complex organization, helping keep similar pieces of code together while allowing each submodule to change over time.

Figure 11.1 illustrates how Go's organizational layers build upon each other. At the foundation are packages, which are individual units of code within a module. Modules group related packages and track their dependencies through `go.mod` files. Each module can have multiple versions, allowing consumers to choose which release to use. At the top level, workspaces coordinate multiple modules during development, enabling you to work on related modules simultaneously while maintaining their independence.

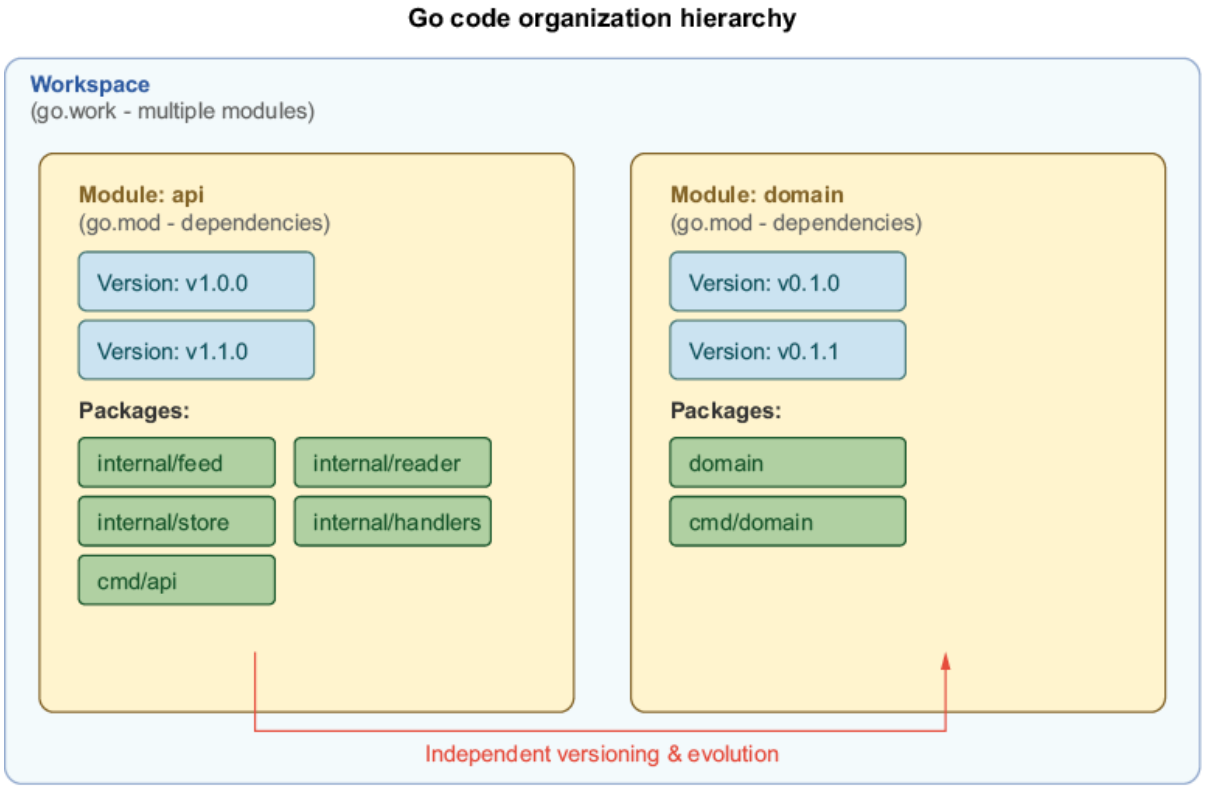


Figure 11.1 Go's code organization hierarchy, from packages to workspaces

11.1 Modules

To begin organizing our code using these concepts, we will migrate the RSS reader application we built in chapter 9, moving some of its existing concurrency source code into this new structure and establishing some best practices along the way. We will drop some of the concurrency to keep things simple, but you can use the concepts learned in this chapter to adapt the migrated application into a more robust one.

Before we copy over the code, though, we will set up our new modules. Modules can be hosted locally or on the web. Many modules that you import will be backed by some Git storage provider; we will be using Git throughout the chapter.

> **NOTE** *Git* is a distributed version control system that tracks changes to your code over time. It's the de facto standard for source code management in modern software development. Although a full Git tutorial is beyond the scope of this book, you'll need a few basic commands to follow along: `git clone` (copy a repository), `git add` (stage changes), `git commit` (save changes), `git tag` (mark versions), and `git push` (upload to a remote server). If you're new to Git, GitHub provides an excellent quick start guide that covers everything you'll need for this chapter: https://docs.github.com/en/get-started/quickstart. The key concepts we'll use are repositories (projects), commits (saved changes), and tags (version markers).

We will use GitHub as our storage provider, but you can easily use other services, such as GitLab. Set up your account and create a new repository called `go-news`. Copy the URL; it will look something like `github.com/your_user_name/go-news`. Once that's done, switch to a terminal and clone the repository so you can work in it.

Now we will create four directories: `domain` for our shared business logic, `storage` for our in-memory storage implementation, `api` for our REST service, and `worker` for our background feed fetcher. We'll start by setting up the domain module with its interfaces, then create the storage module that implements those interfaces, and finally build the API and worker modules that depend on both.

We are doing this because we want to manage these components separately, which will allow us to update one without affecting the others. The `domain` will act as a stable set of definitions for our product, `storage` will provide a shared implementation, and the services will use them both. This is a common pattern in microservice

architectures where services need to share both interfaces and utility implementations without tight coupling.

We are being this modular on purpose, so we can demonstrate a few concepts. You may find that you or your team want less granularity for your own projects.

Before we start building our modules, let's look at how Go manages dependencies through the `go.mod` and `go.sum` files. These two files work together to provide reproducible builds and dependency management.

Every Go module starts with a `go.mod` file created by the `go mod init` command:

```
cd domain
go mod init github.com/YOUR_USERNAME/go-news/domain
```

This creates a `go.mod` file in the current directory with minimal content:

```
module github.com/YOUR_USERNAME/go-news/domain

go 1.25
```

The *module path* serves as both a unique identifier and the import path prefix for packages within the module. For modules that are hosted on GitHub or GitLab, the path should match the repository location (such as `github.com/YOUR_USERNAME/go-news/domain`) so the `go` command can automatically fetch it. Multimodule repositories use subdirectories in the path, and major versions append `/v2`, `/v3`, etc.

As you add dependencies, your `go.mod` file will grow to include more information:

```
module github.com/YOUR_USERNAME/go-news/api #1

go 1.25 #2

require ( #3
    github.com/YOUR_USERNAME/go-news/domain v0.0.1
    github.com/gorilla/mux v1.8.0
)

require (
    github.com/other/transitive v0.5.2 // indirect #4
)
```

#1 Module directive declares this module's path/identifier.
#2 Go directive specifies the minimum Go version required (affects language features available).
#3 Require blocks list direct dependencies with their versions.
#4 Indirect comment marks transitive dependencies (dependencies of your dependencies) that aren't directly imported in your code.

The `go.mod` file is both human-readable and machine-editable, though most changes happen automatically through commands like `go get` or `go mod tidy`. To add a dependency, use `go get` with the package path:

```
go get github.com/holmes89/gofeed           # Latest version
go get github.com/holmes89/gofeed@v1.2.1    # Specific version
```

Alternatively, simply add the import to your code and run `go mod tidy` to fetch and add it automatically.

Alongside `go.mod`, Go maintains a `go.sum` file containing cryptographic checksums:

```
github.com/gorilla/mux v1.8.0 h1:i40aqfkR1h2SlN9hojwV5ZA91wcXFOvkdN
IeFDP5koI=
github.com/gorilla/mux v1.8.0/go.mod h1:DVbg23sWSpFRCP0SfiEN6jmj59U
nW/n46BH5rLB71So=
```

Each dependency typically has two entries: one for the module's source code (the `h1:…` hash) and one for its `go.mod`

file. These checksums serve critical security and reproducibility functions by ensuring that the code hasn't been tampered with. If someone modifies a published version, even maliciously, the checksum won't match and Go will refuse to use it. This security mechanism ensures that all developers and build systems get identical code for a given version, eliminating "works on my machine" problems caused by different dependency versions. Go also checks downloaded modules against both `go.sum` and a global checksum database (sum.golang.org by default), providing defense in depth.

The `go.sum` file should always be committed to version control alongside `go.mod`. Although `go.mod` declares what your dependencies are, `go.sum` verifies that you're getting the exact code you expect.

As your project evolves, you may add or remove dependencies. The `go mod tidy` command keeps everything synchronized:

```
go mod tidy
```

This command performs several cleanup operations:

- Removes dependencies that are no longer imported in your code
- Adds missing dependencies that are imported but not listed
- Updates `go.sum` with checksums for all dependencies
- Removes unused checksums from `go.sum`

Run `go mod tidy` regularly, especially before committing changes, to keep your module files accurate and minimal.

11.1.1 Packages

Packages are Go's fundamental unit of code organization. When creating an executable application, we use `main` as the package name to indicate where the application starts: the `main()` function in the `main` package serves as the entry point. All other packages can be named according to their purpose and the code they contain.

Although Go provides significant flexibility in how you structure your packages, there is one critical rule: packages cannot contain circular dependencies. That is, package A cannot import package B if package B directly or indirectly imports package A. The Go compiler rejects such arrangements. This constraint encourages thoughtful design and a clear separation of concerns.

Beyond this requirement, Go offers few prescriptive rules about package organization. This freedom is powerful, but it can lead to confusion if not approached thoughtfully. A well-organized codebase makes it easier to locate code, understand dependencies, and maintain the project over time. Poorly organized code, by contrast, can become difficult to navigate and test.

There is no official way to lay out your application or packages, but conventions and patterns can help keep your code organized. Some codebases are small enough to have a very flat structure with few or no packages other than `main`. In more complicated applications or libraries, developers add some structure:

- `cmd` is a directory often created to contain all executables for a given package. It might contain APIs, workers, command-line applications, or various binaries that share a common set of libraries. Each executable should have its own subdirectory (e.g., `cmd/myapp/main.go`) with a `main` package and `main()` function. This

pattern allows a single repository to provide multiple related command-line tools or services while sharing common code.

- `internal` is a special directory because it is not visible to external applications that wish to import a library. This is great for logic that should not be changed or used externally. The Go compiler enforces this visibility rule: code in `internal` can be imported only by code in the directory tree rooted at the parent of `internal`. For example, code in `github.com/user/repo/internal/foo` can be imported by code in `github.com/user/repo` but not by `github.com/other/repo`.
- `pkg` is a directory containing library code that is safe for external applications to use. It explicitly indicates which packages are intended for public consumption. Although this is a common convention, it's not strictly necessary; many projects simply place their public packages at the root level.
- `api` is used to store API definition files such as OpenAPI/Swagger specs, protocol buffer definitions, JSON schema files, or other API contracts. This structure helps keep your interface definitions organized and easily accessible.
- `web` contains web application assets such as static files, templates, and single-page application (SPA) builds. This directory is particularly useful if your Go application serves web content.
- `configs` holds configuration file templates or default configurations. Keep in mind that actual deployment configurations with secrets should not be committed to version control for security reasons. Secrets should be loaded via environment variables or through a secrets manager.
- `scripts` is for build, installation, analysis, and other operational scripts. These scripts keep the root Makefile small and simple.
- `docs` contains design and user documents in addition to godoc-generated documentation. This is where you might place architecture decision records, user guides, or other project documentation.

ANTIPATTERNS

Avoid the src directory. Unlike Java or other languages that encourage a src directory, Go projects should not use this pattern. In Go, the repository root is already the source root. Adding a src directory creates unnecessary nesting and goes against Go community conventions. If you're coming from languages like Java where src/main/java is standard, resist the urge to replicate this structure in Go. Your Go packages should live at the repository root or in meaningfully named directories like those described above.

Our multimodule RSS reader project demonstrates this standard layout in practice. Figure 11.2 shows the complete directory structure, with the workspace root containing four independent modules, each with its own `go.mod` file and organized according to these conventions.

```
go-news/
├── go.work                                  // Workspace coordination
├── domain/                                  // Domain module
│   ├── go.mod
│   ├── article.go
│   ├── feed.go
│   ├── fetcher.go                           // Core entities and interfaces
│   └── storage.go
├── storage/                                 // Storage module
│   ├── go.mod                               // Bbolt persistence adapter
│   └── store.go                             // API module
├── api/                                     // REST service entry point
│   ├── go.mod                               // Worker module
│   └── cmd/api/main.go                      // Feed fetcher entry point
└── worker/
    ├── go.mod
    └── cmd/worker/main.go
```

Figure 11.2 Complete directory structure for go-news with domain, storage, API, and worker modules

11.1.2 Domain module: Shared domain model

The domain module will contain our core entities and interfaces, which both the API and worker modules will depend on. Our *domain model* contains core concepts like articles and feeds, which define what our system does, independent of how it does it. This approach comes from domain-driven design (DDD), which emphasizes modeling software around real-world business concepts rather than technical implementation details. By extracting the domain into its own module, we create a shared vocabulary that multiple services can use without tight coupling.

Figure 11.3 illustrates this architecture, with the domain sitting at the center, defining entities and interfaces. Both the API and worker depend on the domain, but the domain depends on nothing. In the figure, the protocol (HTTP) uses

the generic definitions created by the domain. This inward dependency flow keeps our core business logic stable and independent of infrastructure concerns like databases or HTTP frameworks.

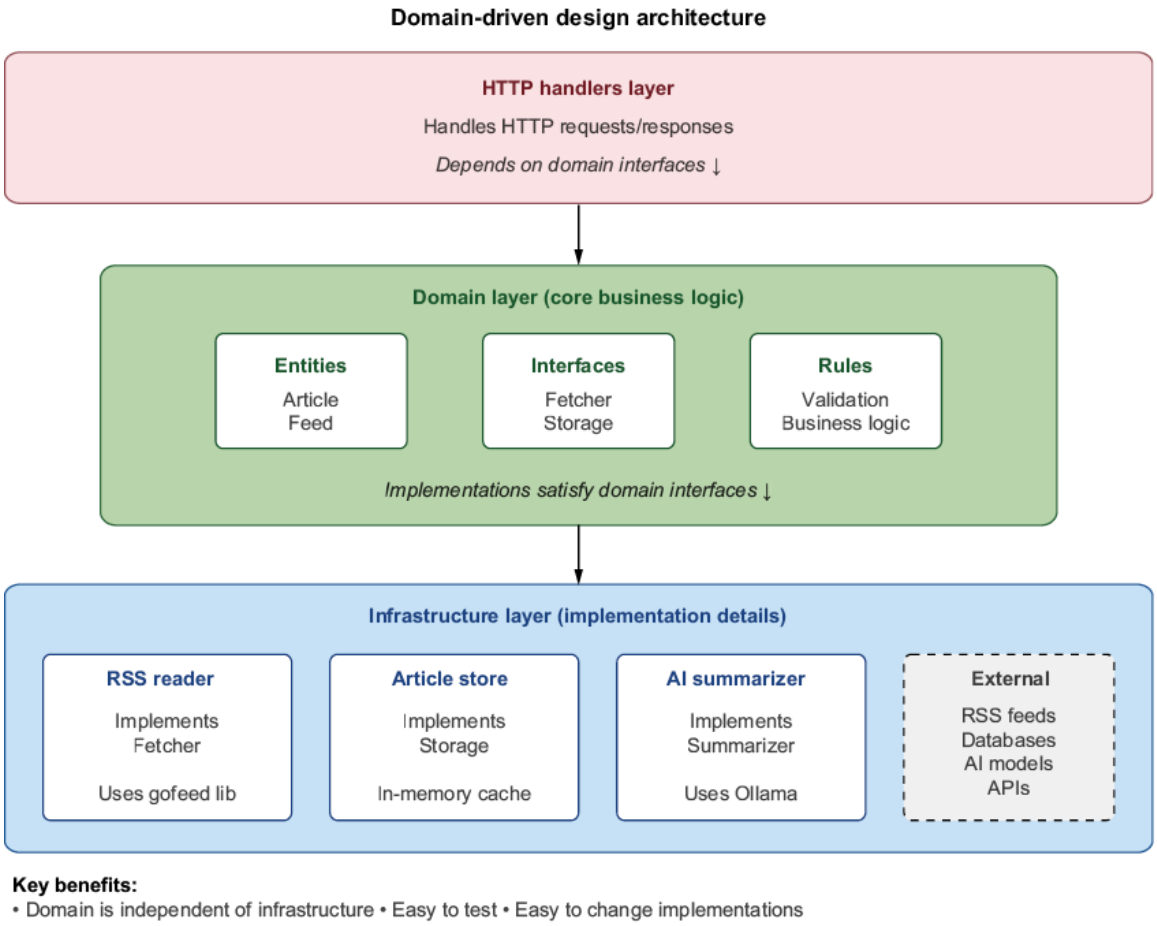


Figure 11.3 Domain-driven design separates core logic from infrastructure.

Let's start by creating `domain/article.go` to define our core types. First, we'll model what an article looks like in our system:

```
package domain #1

import (
    "time"
)

// Article represents a news article from an RSS feed
type Article struct {  #2
    ID          string
    Title       string
    Description string
    Link        string
    Published   *time.Time
    FeedTitle   string
}
```

#1 Package name is domain for our shared module.
#2 Article struct represents our core entity with all necessary fields.

The `Article` type contains only the fields we care about for our application. Notice we're using standard library types like `time.Time` rather than types from external libraries. This keeps our domain model clean and portable.

Now let's define a `Feed` type to represent an entire RSS/Atom feed. Create `domain/feed.go`:

```
package domain

// Feed represents an RSS/Atom feed
type Feed struct {
    Title       string
    Description string
    Link        string
    Articles    []*Article
}
```

DEFINING BEHAVIOR THROUGH INTERFACES

In domain-driven design, we establish our entities as the *nouns* of our system, such as `Article` and `Feed`. Now we need to define the *verbs*: the actions that can be performed on these entities. This is where interfaces become powerful. Rather than tightly coupling our code to specific implementations, we define interfaces that describe the behaviors we need.

Let's define an interface for fetching feeds. Create `domain/fetcher.go`:

```
package domain

import "context"

// Fetcher retrieves and parses RSS feeds
type Fetcher interface {
    FetchFeed(ctx context.Context, url string) (*Feed, error)
}
```

This interface describes a single behavior: given a URL, it fetches and returns a `Feed`. Notice that we're specifying *what* needs to happen, not *how* it happens. This separation provides flexibility, which means we can change the underlying implementation without affecting any code that uses the interface.

Think of it like electrical outlets in your home. If everything were directly hardwired, moving a lamp to a different room would require rewiring the house (and risking a nasty shock!). Instead, outlets provide a standard interface: plug things in wherever you need them. The same principle applies here: our `Fetcher` interface is the "outlet" that any feed-fetching implementation can "plug into."

We also need to define how articles will be stored. Let's add a `Storage` interface to our domain. Create `domain/storage.go`:

```
package domain

// ArticleReader provides read-only access to articles
type ArticleReader interface {
    GetRecent(n int) []*Article
    GetByID(id string) (*Article, error)
}

// Storage persists and retrieves articles
type Storage interface {
    ArticleReader #1
    AddArticles(articles []*Article) error
}
```

#1 This interface extends the read only interface so they stay in sync.

This interface describes the storage operations our application needs without specifying whether articles are stored in memory, a database, or elsewhere.

THE LISKOV SUBSTITUTION PRINCIPLE

These interfaces embody a fundamental principle in software design called the *Liskov substitution principle* (LSP), named after computer scientist Barbara Liskov. The LSP states that any implementation of an interface should be usable wherever that interface is expected without breaking the application. In simpler terms, if your code expects a `Fetcher`, it shouldn't matter which specific `Fetcher` implementation you give it; they should all work interchangeably.

This principle gives us tremendous flexibility. Our worker can use an `RSSReader` that fetches real RSS feeds, while our tests can use a `MockFetcher` that returns predetermined data. A future implementation could use an entirely different RSS library without changing a single line of code in the packages that *use* the interface.

This architectural pattern is known as *hexagonal architecture*, or ports and adapters. Figure 11.4 shows how the domain sits at the center as a hexagon, with ports (interfaces) on each side. Adapters plug into these ports to connect the domain to external systems like HTTP servers, databases, or RSS feeds. The key insight is that dependencies always connect to the domain. Adapters depend on domain interfaces, never the reverse.

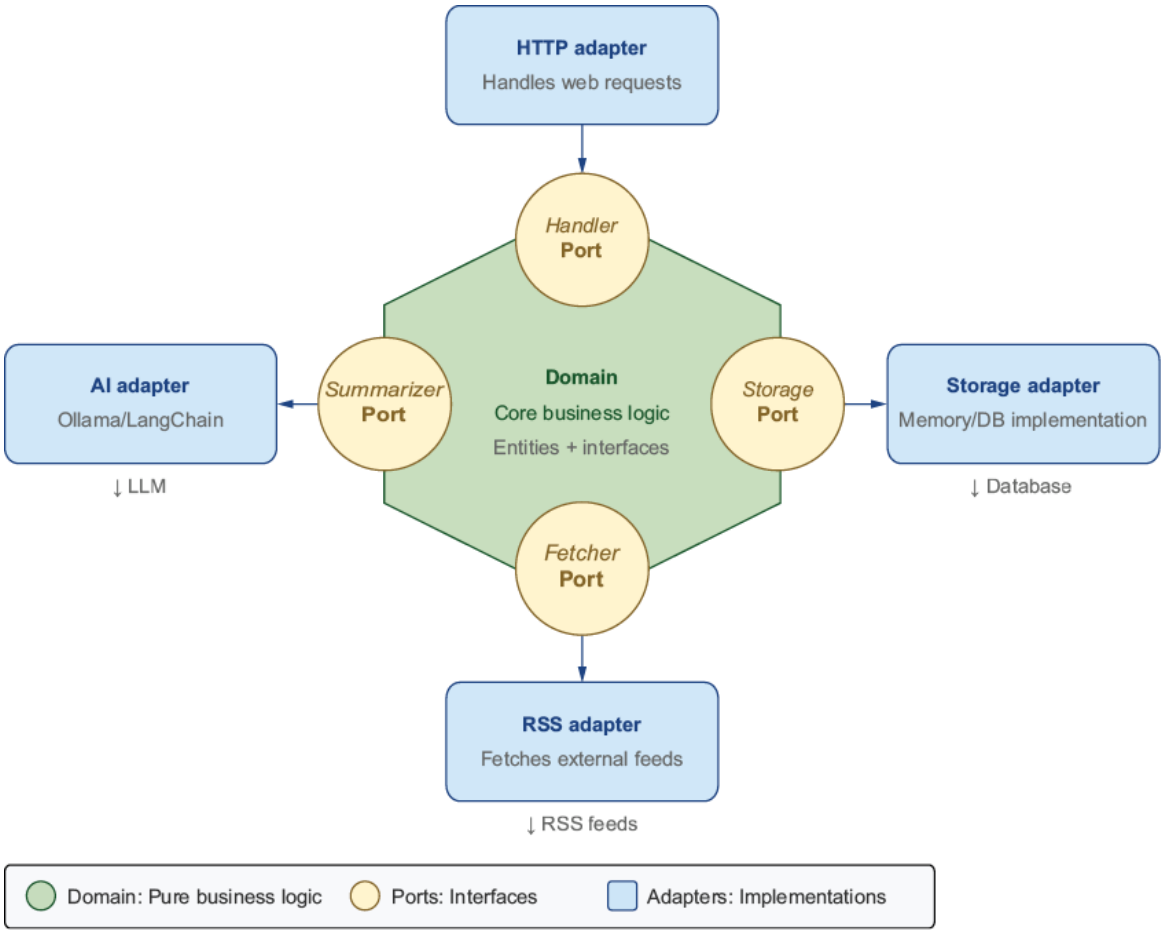


Figure 11.4 Hexagonal architecture: Domain at the center with ports and adapters

By defining our domain as a separate module, we achieve several architectural benefits. First, dependencies connect to the domain, where consumer modules depend on stable domain abstractions rather than volatile implementation

details. Second, the domain layer remains completely independent of infrastructure concerns, such as which RSS library we use or which HTTP framework serves our API. Third, we gain the flexibility to swap implementations without affecting any consuming code. Finally, our interfaces expose focused behaviors that reveal only the operations consumers actually need, hiding unnecessary complexity from the underlying libraries.

This arrangement also prevents circular dependencies. The domain module, which contains interfaces and entities, has no dependencies on anything else in our application. Implementation modules like `api` and `worker` can import the domain to satisfy its interfaces, but the domain never imports implementation modules. This unidirectional dependency flow keeps our architecture clean and maintainable.

11.2 Versioning modules

Now that we've defined our domain module with our core entities and interfaces, it's time to version and publish it. Versioning allows us to track changes over time, and it gives consumers of our module clear expectations about compatibility. Go follows semantic versioning (SemVer), where versions have the format `vMAJOR.MINOR.PATCH`:

- *Major* version increments indicate incompatible API changes (breaking changes)
- *Minor* version increments add functionality in a backward-compatible manner
- *Patch* version increments indicate backward-compatible bug fixes

For our domain module, this is our initial release, so we'll tag it as v0.0.1, which signals a stable API that other

modules can depend on. Let's commit and tag our domain module:

```
cd domain
git add .
git commit -m "Initial domain module with core entities and interfa
ces"
git tag domain/v0.0.1
git push origin domain/v0.0.1
```

Notice the tag format: `domain/v0.0.1`. When you have multiple modules in a repository, you prefix tags with the module path. This tells Go which module the version applies to.

Now let's create the storage module that implements the `domain.Storage` interface. This module will provide persistent storage that both the API and the worker can use simultaneously. By extracting this functionality into its own module, we avoid code duplication and demonstrate another benefit of workspaces: shared utility modules.

This storage module exemplifies the *adapter* pattern in *hexagonal architecture* (also called *ports and adapters*). In this architectural style, the domain defines ports, which are interfaces that describe required behaviors without specifying how they're implemented. The storage module is an adapter that plugs into the `domain.Storage` port, providing a concrete implementation. The domain doesn't know or care whether articles are stored in memory, PostgreSQL, MongoDB, or even a filesystem. It simply defines what storage operations are needed, and adapters like this one fulfill those requirements.

This separation is powerful because it allows us to swap implementations without touching the domain or the services that depend on it. If we later needed to replace our

storage with a Redis cache or a PostgreSQL database, we would create a new adapter module (e.g., `storage-postgres`) that implements the same `domain.Storage` interface. The API and worker modules wouldn't need to change at all because they depend on the port (interface), not the adapter (implementation). This is the essence of dependency inversion: high-level modules (such as API and worker) depend on abstractions (domain interfaces), not on low-level details (storage mechanisms).

WHY PERSISTENT STORAGE?

Our architecture presents an interesting challenge: both the worker and API services need to access the same article data, but they run as independent processes. An in-memory solution would give each process its own isolated storage, meaning articles fetched by the worker wouldn't be visible to the API. We need a persistence mechanism that both processes can connect to.

For this demonstration, we'll use *bbolt* (formerly BoltDB), an embedded key/value database written in pure Go. Bbolt is an excellent choice for several reasons. First, it's embedded directly into your application, so there's no separate database server to install or manage. The entire database is a single file on disk that both processes can access. Second, bbolt handles concurrent access elegantly: it supports multiple readers simultaneously while allowing only one writer at a time, which perfectly matches our pattern, where the worker writes periodically and the API reads continuously. Third, the API is beautifully simple. You store and retrieve byte slices organized into named "buckets" (similar to tables in SQL databases). Finally, bbolt is used in production by major systems like etcd and Consul, which demonstrates its reliability.

The choice of bbolt demonstrates an important architectural principle: start simple and proven. Rather than introducing the complexity of PostgreSQL or Redis for our demonstration, bbolt provides just enough persistence to make our multiservice architecture work while keeping the code focused on module organization.

Create the storage module and add the bbolt dependency:

```
cd storage
go mod init github.com/YOUR_USERNAME/go-news/storage
go get github.com/YOUR_USERNAME/go-news/domain@v0.0.1
go get go.etcd.io/bbolt@v1.3.11
```

DATABASE INITIALIZATION AND STRUCTURE

Before we can store articles, we need to open a connection to the database file and ensure our storage structure exists. Bbolt organizes data into *buckets*, which are like namespaces or tables that hold related key/value pairs. We'll create an "articles" bucket to store all our article data.

Create `store.go` in the storage module with the following initialization code so we can set up a database if it doesn't already exist.

Listing 11.1 Initializing the bbolt database and articles bucket

```
package storage

import (
    "encoding/json"
    "fmt"
    "slices"
    "time"

    "github.com/YOUR_USERNAME/go-news/domain"
    "go.etcd.io/bbolt"
)

const articlesBucketName = "articles" #1

var _ domain.Storage = (*BoltStore)(nil) #2

type BoltStore struct {
    db *bbolt.DB
}

func NewBoltStore(dbPath string, readOnly bool) (*BoltStore, error)
{ #3
    db, err := bbolt.Open(dbPath, 0600, &bbolt.Options{ #4
        Timeout:  1 * time.Second,
        ReadOnly: readOnly,
    })
    if err != nil {
        return nil, fmt.Errorf("failed to open database: %w", err)
    }

    if !readOnly { #5
        err = db.Update(func(tx *bbolt.Tx) error {
            _, err := tx.CreateBucketIfNotExists([]byte(articlesBuc
ketName))
            return err
        })
        if err != nil {
            db.Close()
            return nil, fmt.Errorf("failed to create bucket: %w", e
rr)
        }
```

```
    }

    return &BoltStore{db: db}, nil
}

func (s *BoltStore) Close() error { #6
    return s.db.Close()
}
```

#1 Constant for our bucket name keeps it consistent across operations.
#2 Compile-time verification that BoltStore satisfies domain.Storage interface
#3 Constructor accepts readOnly parameter to control database access mode.
#4 Open database with ReadOnly option for safe concurrent access patterns
#5 Only create bucket if opened in write mode (read-only mode can't modify schema).
#6 Close method properly releases database resources.

The `NewBoltStore` constructor handles several important initialization steps. First, it accepts a `readOnly` parameter that controls the database access mode, which is critical for enforcing the separation between read-only services (API) and read-write services (worker) at the database level. Second, it opens or creates the database file at the specified path with `0600` permissions, meaning only the file's owner can read and write it. The timeout option prevents the application from hanging indefinitely if the database file is locked by another process.

When opened in write mode (`readOnly: false`), the constructor uses `db.Update()` to create the articles bucket if it doesn't already exist. Bbolt uses transactions for all operations: `Update` for read-write transactions and `View` for read-only transactions. When opened in read-only mode, we skip bucket creation entirely because read-only databases cannot modify the schema. This means you must run the worker first to create the database structure before starting the API.

STORING ARTICLES WITH SERIALIZATION

Now let's implement `AddArticles`, which the worker will use to persist fetched articles. Since bbolt stores byte slices, we need to serialize our `domain.Article` structs into a format suitable for storage. JSON is an excellent choice: it's human-readable for debugging, it's widely understood, and Go's standard library provides robust serialization support.

Add the `AddArticles` method to `store.go`.

Listing 11.2 Storing articles in a write transaction

```
func (s *BoltStore) AddArticles(articles []*domain.Article) error {
    return s.db.Update(func(tx *bbolt.Tx) error { #1
        bucket := tx.Bucket([]byte(articlesBucketName)) #2
        if bucket == nil {
            return fmt.Errorf("articles bucket not found")
        }

        for _, article := range articles { #3
            data, err := json.Marshal(article) #4
            if err != nil {
                return fmt.Errorf("failed to marshal article %s: %
w",
                    article.ID, err)
            }

            if err := bucket.Put([]byte(article.ID), data); err !=
nil { #5
                return fmt.Errorf("failed to store article %s: %w",
                    article.ID, err)
            }
        }

        return nil
    })
}
```

#1 Use Update transaction for write operations.
#2 Get the articles bucket from the transaction.
#3 Iterate through all articles to store.
#4 Marshal each article to JSON bytes.
#5 Store using article ID as key and JSON data as value.

The `AddArticles` implementation demonstrates bbolt's transactional nature. Everything happens within a single `Update` transaction, meaning either all articles are stored successfully or none are. There's no partial state. This atomicity is crucial for data integrity.

We use each article's `ID` field, converted to bytes, as the key. This gives us natural deduplication: if the worker

fetches the same article multiple times, which happens when RSS feeds republish content, the latest version simply overwrites the previous one. The article data itself is JSON-encoded, making it easy to inspect the database file with bbolt's command-line tools during development.

Error handling is explicit at each step. If JSON marshaling fails or the database write fails, we return a descriptive error that includes context about which article caused the problem. The transaction automatically rolls back on error, ensuring the database remains consistent.

RETRIEVING RECENT ARTICLES

The API needs to retrieve the most recently published articles. This requires iterating through all stored articles, sorting them by publication date, and returning the top *N* results. Add the `GetRecent` method.

Listing 11.3 Retrieving the most recent articles

```
func (s *BoltStore) GetRecent(n int) []*domain.Article {
    var articles []*domain.Article

    s.db.View(func(tx *bbolt.Tx) error { #1
        bucket := tx.Bucket([]byte(articlesBucketName))
        if bucket == nil {
            return nil
        }

        bucket.ForEach(func(k, v []byte) error { #2
            var article domain.Article
            if err := json.Unmarshal(v, &article); err != nil { #3
                return fmt.Errorf("failed to unmarshal article: %
w", err)
            }
            articles = append(articles, &article)
            return nil
        })

        return nil
    })

    // Sort by publication date, newest first #4
    slices.SortFunc(articles, func(a, b *domain.Article) int {
        if a.Published == nil && b.Published == nil {
            return 0
        }
        if a.Published == nil {
            return 1  // nil published dates sort to the end
        }
        if b.Published == nil {
            return -1 // non-nil comes before nil
        }
        // Compare b to a (reverse order) for newest first
        return b.Published.Compare(*a.Published)
    })

    if n > len(articles) { #5
        n = len(articles)
    }
```

```
    return articles[:n]
}
```

#1 Use View transaction for read-only operations (allows concurrent readers).
#2 ForEach iterates over all key-value pairs in the bucket.
#3 Unmarshal JSON bytes back into domain.Article struct.
#4 Use slices.SortFunc with custom comparison for newest-first ordering.
#5 Handle case where fewer articles exist than requested

The `GetRecent` method uses a `View` transaction rather than `Update`. This is important for performance: bbolt allows multiple concurrent readers, so the API can serve many requests simultaneously while the worker writes new articles. The read-only nature of `View` transactions makes this safe because readers never see partial writes.

The `ForEach` method iterates through every article in the bucket. For each key/value pair, we unmarshal the JSON bytes back into a `domain.Article` struct and collect the articles in a slice. Once we have all articles in memory, we sort them by publication date, newest first, using `slices.SortFunc` from Go's standard library.

The sort comparison function handles nil publication dates gracefully by placing them at the end of the results. For articles with valid publication dates, we use the `Compare` method (introduced in Go 1.20), which returns -1, 0, or 1 depending on whether the time is before, equal to, or after another time. By comparing `b.Published.Compare(*a.Published)` (reversed order), we achieve a newest-first sort. The `slices.SortFunc` function uses an efficient sorting algorithm (typically quicksort or introsort) that performs well even with thousands of articles.

LOOKING UP INDIVIDUAL ARTICLES

Finally, let's implement `GetByID` to retrieve specific articles by their identifier. This operation is much simpler than `GetRecent` because bbolt's key/value structure makes lookups extremely efficient:

```
func (s *BoltStore) GetByID(id string) (*domain.Article, error) {
    var article *domain.Article

    err := s.db.View(func(tx *bbolt.Tx) error { #1
        bucket := tx.Bucket([]byte(articlesBucketName))
        if bucket == nil {
            return fmt.Errorf("articles bucket not found")
        }

        data := bucket.Get([]byte(id)) #2
        if data == nil {
            return nil // Article not found #3
        }

        article = &domain.Article{}
        if err := json.Unmarshal(data, article); err != nil { #4
            return fmt.Errorf("failed to unmarshal article: %w", er
r)
        }

        return nil
    })

    return article, err
}
```

#1 Use View transaction for consistent read.
#2 Direct key lookup by article ID
#3 Return nil for article if key doesn't exist (not treated as error).
#4 Unmarshal JSON bytes into domain.Article struct

The `GetByID` implementation showcases bbolt's efficiency as a key/value store. The `Get` operation is an O(1) lookup that uses a B+tree internally, making it extremely fast even with thousands of articles. This is one of the key advantages of using article IDs as keys: direct access without iteration.

The method distinguishes between “article not found” (returns a nil article with no error) and “database error” (returns a nil article with an error). This distinction is important for API handlers: a missing article should return HTTP 404 Not Found, while a database error should return HTTP 500 Internal Server Error.

COMPLETE STORAGE ADAPTER

Our complete storage adapter now implements all three methods required by `domain.Storage`: `AddArticles` for the worker to persist fetched content, `GetRecent` for the API to serve recent articles, and `GetByID` for individual article lookups. The adapter demonstrates several software engineering best practices worth highlighting.

First, the adapter completely isolates bbolt’s implementation details from the rest of our application. The domain module knows nothing about buckets, transactions, or JSON serialization. It simply defines storage operations, and this adapter fulfills them. If we later decide to migrate to PostgreSQL or MongoDB, we can create a new adapter implementing the same interface and swap it in `main.go`. The API and worker code never change.

Second, the implementation handles concurrent access correctly. Both services can open `BoltStore` instances pointing to the same database file. The worker writes during its periodic fetch cycles using `Update` transactions, while the API continuously reads using `View` transactions. Bbolt manages the synchronization, ensuring that readers never see inconsistent data and the single writer never corrupts the database.

Third, error handling follows Go conventions. Operations that might fail return errors with context, making debugging straightforward. The constructor's signature, `NewBoltStore(dbPath string, readOnly bool) (*BoltStore, error)`, signals to callers that initialization can fail, prompting them to handle the error case properly. The explicit `readOnly` parameter makes the access pattern visible at the call site.

With our storage implementation complete, we now have a concrete adapter that plugs into the domain's storage port. Both services can share data through this persistent layer while remaining independent processes. Let's version this module so other services can depend on it:

```
cd storage
git add .
git commit -m "Add bbolt persistent storage implementation"
git tag storage/v0.0.1
git push origin storage/v0.0.1
```

Now both the API and worker modules can depend on specific versions:

```
require (
    github.com/YOUR_USERNAME/go-news/domain v0.0.1
    github.com/YOUR_USERNAME/go-news/storage v0.0.1
)
```

This versioning strategy establishes a stable domain model that other modules can depend on, allows modules to evolve independently, and communicates compatibility through semantic versioning.

The storage module demonstrates the power of the adapter pattern. Implementing `domain.Storage` with bbolt gives us persistence, concurrent access, and simplicity. The domain will be the most stable module, storage might change for

optimizations, and the API and worker will evolve most rapidly.

11.3 Workspaces

Now that we have a versioned domain module, we're ready to build services that depend on it. Go workspaces allow us to develop multiple interdependent modules together while preserving their independence. We can make changes across module boundaries without constantly publishing versions. Each module keeps its own `go.mod` file and releases independently, but during development, they reference each other's latest code directly.

11.3.1 Understanding workspaces

A workspace is defined by a `go.work` file at your repository root. During development, Go uses the local module versions listed in `go.work` rather than fetching them from remote repositories. Normally, when module A depends on module B, Go downloads B from its published location at a specific version. Workspaces create a development mode where Go automatically uses your local modules instead.

Each module maintains its own `go.mod` file and can be versioned independently. The API module might be at v1.5.2 while the worker is at v1.2.0. Workspaces are purely for development; when users import your modules, they get published versions from the module proxy, not your `go.work` file.

Figure 11.5 shows how the workspace coordinates development across modules. The `go.work` file at the repository root lists all modules. During development, Go uses the local versions specified in `go.work` rather than

downloading published versions, enabling seamless cross-module development while each module retains its independence.

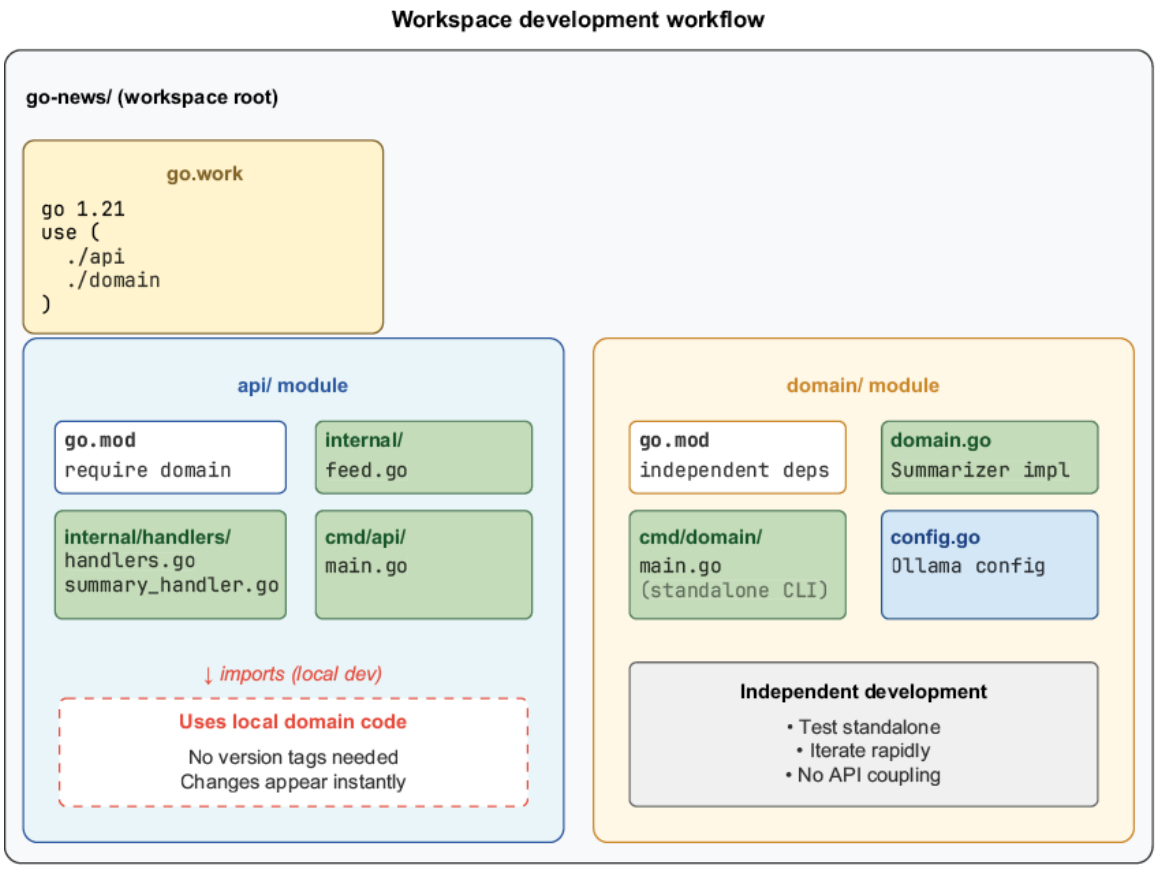


Figure 11.5 Workspace development workflow with local module resolution

Let's create a workspace. At the root of your `go-news` repository (above the `domain`, `storage`, `api`, and `worker` directories), run this command:

```
go work init ./domain ./storage
```

This creates a `go.work` file that includes both the domain and storage modules. We'll add the API and worker modules as we create them. Your repository structure now looks like this:

```
go-news/
├── go.work
├── domain/
│   ├── go.mod
│   ├── article.go
│   ├── feed.go
│   ├── fetcher.go
│   └── storage.go
└── storage/
    ├── go.mod
    └── store.go
```

11.3.2 API module: REST service

Now let's create our API module, which will provide HTTP endpoints for accessing articles. This module will depend on both the domain module (for types and interfaces) and storage module (for the implementation), but it will focus on HTTP concerns and presenting data to clients.

This is our first executable component in the workspace, so we'll introduce the `cmd` directory pattern. This convention allows a single module to contain multiple executable programs, each in its own subdirectory. For our API module, we'll create `cmd/api/main.go`, which will contain the `main` package and `main()` function that starts our HTTP server.

Create the API module:

```
cd api
go mod init github.com/YOUR_USERNAME/go-news/api
go get github.com/YOUR_USERNAME/go-news/domain@v0.0.1
go get github.com/YOUR_USERNAME/go-news/storage@v0.0.1
```

Add it to the workspace:

```
# From repository root
go work use ./api
```

The API module structure will be broken into a command directory, and the handlers are placed in an internal directory since they are only used in this module:

```
api/
├── cmd/
│   └── api/
│       └── main.go         #1
├── internal/
│   └── handlers/
│       ├── handlers.go      #2
│       └── handlers_test.go
└── go.mod
```

#1 Executable entry point with main() function
#2 HTTP handlers with dependency injection

Note that the API module doesn't have its own `store` package; it imports the shared `storage` module instead. This demonstrates a key benefit of workspaces: shared implementations can be extracted into reusable modules that multiple services can depend on.

HTTP HANDLERS WITH DEPENDENCY INJECTION

Now we can build our HTTP layer, where we'll see the benefit of the domain's interface design. Our handlers are read-only: they serve data but don't modify it. Rather than depending on the full `domain.Storage` interface, which includes write operations, we can depend on just the `domain.ArticleReader` interface that the domain module already separated out for exactly this purpose.

This approach provides several benefits. First, it makes constraints explicit: the type system ensures that handlers can't accidentally modify data. Second, it simplifies testing because we only need to mock the operations we actually use. Third, it improves code comprehension because anyone

reading the handler constructor immediately understands that it’s read-only without examining the implementation.

With the interface narrowed to reads, the handler type becomes small: it holds a single dependency, registers its own routes, and does nothing but translate between HTTP and the domain. Create `internal/handlers/handlers.go` as follows.

Listing 11.4 HTTP handlers with an injected ArticleReader

```
package handlers

import (
    "encoding/json"
    "net/http"
    "strconv"

    "github.com/YOUR_USERNAME/go-news/domain"
)

type Handlers struct { #1
    articles domain.ArticleReader
}

func New(articles domain.ArticleReader) *Handlers { #2
    return &Handlers{
        articles: articles,
    }
}

func (h *Handlers) RegisterRoutes(mux *http.ServeMux) { #3
    mux.HandleFunc("/articles", h.articlesHandler)
}

func (h *Handlers) articlesHandler(w http.ResponseWriter, r *http.R
equest) { #4
    if r.Method != http.MethodGet {
        http.Error(w, "method not allowed", http.StatusMethodNotAll
owed)
        return
    }

    n := 10
    if countStr := r.URL.Query().Get("count"); countStr != "" { #5
        if count, err := strconv.Atoi(countStr); err == nil && coun
t > 0 {
            n = count
        }
    }

    articles := h.articles.GetRecent(n)
```

```
    w.Header().Set("Content-Type", "application/json") #6
    if err := json.NewEncoder(w).Encode(articles); err != nil {
        http.Error(w, "failed to encode response", http.StatusInter
nalServerError)
        return
    }
}
```

#1 Handlers manages HTTP request handlers with their dependencies
#2 New creates handlers accepting domain.ArticleReader for read-only access.
#3 RegisterRoutes mounts all handler routes on the provided mux.
#4 articlesHandler returns recent articles as JSON.
#5 Parse optional count query parameter.
#6 Return JSON response.

Notice how `Handlers` depends on `domain.ArticleReader`, not `domain.Storage`. Our storage module's `BoltStore` implements the full `domain.Storage` interface, which embeds `ArticleReader`, so it automatically satisfies the read-only interface requirement. This is the Liskov substitution principle in action: we can pass our full storage implementation to code that expects a read-only view, and everything works seamlessly.

The `RegisterRoutes` method mounts all handler routes on the provided mux. This pattern colocates route definitions with handlers, enables multiple handler groups to be composed, and dramatically improves testability by allowing tests to create isolated muxes without application-wide setup.

TESTING HANDLERS WITH TEST DOUBLES

Our interface-based architecture enables easy testing through *test doubles*, which are simplified implementations used exclusively for testing. Like a stand-in actor in rehearsal, test doubles let us test code without real databases or external services. We test that handlers parse

query parameters, handle HTTP methods, and return JSON correctly, without testing storage itself. By using a test double that implements `domain.ArticleReader`, we test handlers in complete isolation.

By creating a simple struct that implements `domain.ArticleReader`, we can provide canned test data to our handlers without involving the actual storage layer. This approach again demonstrates the Liskov substitution principle: any implementation of an interface can be substituted for another, whether it's the real `BoltStore` in production or a test stub in your tests. The handlers work identically with both because they depend only on the interface contract, not the concrete implementation.

> **NOTE** Test doubles come in different varieties. A *stub* returns predetermined data, while a *mock* verifies that specific methods were called with the expected arguments. Stubs are simpler and sufficient when you only need to provide data. For more complex scenarios where you need to assert interaction patterns, consider mocking libraries like the one in `github.com/stretchr/testify`.

For comprehensive examples of testing HTTP handlers using Go's `httptest` package, including how to create test stubs, test requests with different HTTP methods, and verify responses, see chapter 8. That chapter provides usage of the `httptest.NewRecorder()` and `httptest.NewRequest()` functions used for testing handlers, as well as table-driven test patterns for exercising multiple scenarios.

WIRING EVERYTHING TOGETHER

Now let’s create the main entry point that wires all these pieces together. Create `cmd/api/main.go` as follows:

```
package main

import (
    "fmt"
    "log"
    "net/http"

    "github.com/YOUR_USERNAME/go-news/api/internal/handlers"
    "github.com/YOUR_USERNAME/go-news/storage"
)

func main() {
    store, err := storage.NewBoltStore("./articles.db", true) #1
    if err != nil {
        log.Fatalf("failed to initialize storage: %v", err)
    }
    defer store.Close() #2

    h := handlers.New(store) #3

    mux := http.NewServeMux() #4
    h.RegisterRoutes(mux)

    fmt.Println("Starting API server on :8080")
    if err := http.ListenAndServe(":8080", mux); err != nil { #5
        log.Fatal(err)
    }
}
```

#1 Open database in read-only mode for API (prevents accidental writes).
#2 Ensure database is properly closed when application exits.
#3 Inject storage into handlers (satisfies ArticleReader interface).
#4 Create HTTP router and register handler routes.
#5 Start HTTP server on port 8080.

This `main.go` demonstrates the power of dependency injection and interface-based design. We create a concrete `BoltStore` from the storage module, which implements

`domain.Storage`. We pass it to the `handlers` constructor, which expects an `ArticleReader`. Although `BoltStore` was designed to satisfy `domain.Storage` (with both read and write operations), it automatically satisfies `ArticleReader` (with only read operations) because Go's interface satisfaction is implicit and structural.

Notice how we handle storage initialization. The constructor returns both a store and an error, requiring us to check whether initialization succeeded. The `defer store.Close()` statement ensures that even if the HTTP server fails to start or crashes later, we properly close the database connection. This cleanup pattern is essential for resources such as database connections, file handles, and network connections.

The database file path `"./articles.db"` is relative to where the application runs. Both the API and worker services will use the same path, so they'll share the same database file. This is precisely what we need: the worker writes articles to the database during its periodic fetches, and the API reads them to serve HTTP requests. Both processes access the same persistent data store.

The storage module provides the adapter, handlers define their needs, and `main` wires them together. Each layer depends only on what it needs and nothing more.

11.3.3 Worker module: Background feed fetcher

The worker module has a single responsibility: periodically fetching RSS feeds and storing new articles. This represents the classic producer-consumer pattern, where the worker produces data and the API consumes it, with each operating independently. Users querying `/articles` get instant responses from data already fetched, never waiting for slow

network requests to RSS providers. Like Unix tools that compose through pipes, these focused services can be restarted, deployed, and scaled independently while communicating through shared domain interfaces.

Create the worker module:

```
cd worker
go mod init github.com/YOUR_USERNAME/go-news/worker
go get github.com/YOUR_USERNAME/go-news/domain@v0.0.1
go get github.com/YOUR_USERNAME/go-news/storage@v0.0.1
go get github.com/holmes89/gofeed
```

Add it to the workspace:

```
# From repository root
go work use ./worker
```

Now we will define the worker module's application structure:

```
worker/
├── cmd/
│   └── worker/
│       └── main.go         #1
├── internal/
│   └── reader/
│       └── reader.go        #2
└── go.mod
```

#1 Executable entry point that orchestrates periodic fetching
#2 RSS fetcher adapter implementing domain.Fetcher interface

Like the API module, the worker doesn't have its own store package. It imports the shared `storage` module. This demonstrates a key benefit of workspaces: multiple services can share utility implementations without duplication.

The domain module defined a `Fetcher` interface describing how to retrieve feeds. Now we'll create an adapter that

plugs into this port using the `gofeed` library. Create `internal/reader/reader.go` as follows.

Listing 11.5 RSS adapter implementing domain.Fetcher

```
package reader

import (
    "context"

    "github.com/YOUR_USERNAME/go-news/domain"
    "github.com/holmes89/gofeed"
)

var _ domain.Fetcher = (*RSSReader)(nil) #1

type RSSReader struct {
    parser *gofeed.Parser
}

func NewRSSReader() *RSSReader {
    return &RSSReader{
        parser: gofeed.NewParser(),
    }
}

func (r *RSSReader) FetchFeed(ctx context.Context, url string) (*do
main.Feed, error) {
    feed, err := r.parser.ParseURLWithContext(url, ctx) #2
    if err != nil {
        return nil, err
    }

    domainFeed := &domain.Feed{ #3
        Title:       feed.Title,
        Description: feed.Description,
        Link:        feed.Link,
        Articles:    make([]*domain.Article, 0, len(feed.Items)),
    }

    for _, item := range feed.Items {
        article := &domain.Article{ #4
            ID:          item.GUID,
            Title:       item.Title,
            Description: item.Description,
            Link:        item.Link,
```

```
            Published:   item.PublishedParsed,
            FeedTitle:   feed.Title,
        }
        domainFeed.Articles = append(domainFeed.Articles, article)
    }

    return domainFeed, nil
}
```

#1 Compile-time verification that RSSReader satisfies domain.Fetcher
#2 Use context-aware parsing to respect timeouts and cancellation.
#3 Transform external library types to domain types.
#4 Map each RSS item to our domain.Article structure.

This adapter demonstrates the port-adapter boundary. The `gofeed` library has its own types (`Feed`, `Item`), with many fields we don't need. Rather than exposing these external types throughout our application, we transform them at the boundary into our domain types. This transformation isolates our domain from the external library. If we later switch to a different RSS library, only this adapter changes.

The compile-time interface check (`var _ domain.Fetcher = (*RSSReader)(nil)`) is a valuable pattern. If `RSSReader` doesn't correctly implement all methods of `domain.Fetcher`, the code won't compile. This catches interface mismatches immediately rather than discovering them at runtime.

Now let's create the worker executable that periodically fetches feeds and stores articles. This demonstrates concurrency patterns from chapter 9 applied in a real service. Create `cmd/worker/main.go`, starting with the imports and core business logic.

Listing 11.6 Feed list and core fetch-and-store logic

```
package main

import (
    "context"
    "fmt"
    "log"
    "os"
    "os/signal"
    "syscall"
    "time"

    "github.com/YOUR_USERNAME/go-news/domain"
    "github.com/YOUR_USERNAME/go-news/storage"
    "github.com/YOUR_USERNAME/go-news/worker/internal/reader"
)

var feeds = []string{ #1
    "https://rss.nytimes.com/services/xml/rss/nyt/World.xml",
    "https://feeds.bbci.co.uk/news/rss.xml",
}

func fetchAndStore(ctx context.Context, fetcher domain.Fetcher,
                   store domain.Storage, urls []string) { #2
    for _, url := range urls { #3
        feed, err := fetcher.FetchFeed(ctx, url) #4
        if err != nil {
            log.Printf("Error fetching %s: %v", url, err)
            continue #5
        }

        if err := store.AddArticles(feed.Articles); err != nil { #6
            log.Printf("Error storing articles from %s: %v", url, e
rr)
            continue
        }

        fmt.Printf("Fetched and stored %d articles from %s\n",
                   len(feed.Articles), feed.Title)
    }
}
```

#1 Configure feed URLs (could be loaded from config file in production)
#2 Core business logic extracted into separate function for clarity and testability
#3 Process each feed URL sequentially.
#4 Use domain.Fetcher interface (any implementation works here).
#5 Continue processing remaining feeds even if one fails.
#6 Store articles using domain.Storage interface.

The `fetchAndStore` function encapsulates the worker's core responsibility: fetching RSS feeds and persisting their articles. By extracting this logic into a separate function, we make it testable in isolation and keep the `main` function focused on orchestration concerns like signal handling and scheduling. The error handling continues processing the remaining feeds when one fails, preventing a single misbehaving RSS source from stopping all work.

With the fetching logic isolated, `main` is left with the operational concerns: wiring up dependencies, listening for shutdown signals, and running the fetch on a schedule.

Listing 11.7 Worker startup, scheduling, and graceful shutdown

```
func main() {
    store, err := storage.NewBoltStore("./articles.db", false) #1
    if err != nil {
        log.Fatalf("failed to initialize storage: %v", err)
    }
    defer store.Close() #2

    fetcher := reader.NewRSSReader() #3

    ctx, cancel := context.WithCancel(context.Background()) #4
    defer cancel()

    sigChan := make(chan os.Signal, 1) #5
    signal.Notify(sigChan, syscall.SIGINT, syscall.SIGTERM)

    go func() { #6
        <-sigChan
        fmt.Println("\nShutdown signal received, stopping worke
r...")
        cancel()
    }()

    fmt.Println("Worker started, fetching feeds every 5 minute
s...")
    ticker := time.NewTicker(5 * time.Minute) #7
    defer ticker.Stop()

    fetchAndStore(ctx, fetcher, store, feeds) #8

    for { #9
        select {
        case <-ticker.C:
            fetchAndStore(ctx, fetcher, store, feeds)
        case <-ctx.Done():
            fmt.Println("Worker stopped gracefully")
            return
        }
    }
}
```

#1 Open database in write mode (worker needs to persist articles).
#2 Ensure database is properly closed on shutdown.

#3 Create RSS fetcher adapter implementing domain.Fetcher.
#4 Context enables graceful shutdown propagation throughout the application.
#5 Listen for OS signals (Ctrl+C, container stop commands).
#6 Separate goroutine monitors for shutdown signals and cancels context.
#7 Ticker triggers fetch cycles every 5 minutes.
#8 Fetch immediately on startup before entering ticker loop.
#9 Select statement coordinates ticker events and cancellation.

Notice the critical difference in database initialization between the two services: the API opens the database with `readOnly: true`, while the worker uses `readOnly: false`. This architectural decision enforces the principle of least privilege at the database level. The API physically cannot write to the database, even if there were a bug in the handler code. Bbolt prevents all write operations when opened in read-only mode. This provides defense in depth, complementing the type-level constraints from our interface design.

Read-only mode also improves concurrency. When bbolt opens a database in read-only mode, it acquires a shared file lock rather than an exclusive one. This allows multiple API instances to run simultaneously against the same database file, each serving requests concurrently. The worker opens the database in write mode, which takes an exclusive lock during write transactions but releases it immediately after each commit. This means the API continues serving requests with minimal blocking, pausing only briefly during the worker’s periodic writes.

This implementation handles production concerns gracefully. Signal handling allows a clean shutdown, which is essential in containerized environments where orchestrators like Kubernetes send `SIGTERM` before forcibly killing processes. The error handling continues processing other feeds even when one fails, so a single misbehaving RSS source doesn’t stop the entire worker. The interface-based dependencies

make the function testable and implementation-agnostic. We could swap bbolt for PostgreSQL by changing just the storage initialization in `main()`.

> **NOTE** For production systems with many feeds, you would want a more sophisticated worker pool pattern, as demonstrated in chapter 9. The simple sequential fetching shown here processes one feed at a time. A worker pool with controlled parallelism would dramatically improve throughput while maintaining the same architectural principles.

Your complete `go.work` file now includes all four modules:

```
go 1.25

use (
    ./domain
    ./storage
    ./api
    ./worker
)
```

We've established a complete multimodule architecture built on several key principles. The domain module defines a stable domain model through interfaces, creating clear service boundaries that allow the API and worker to depend on abstractions rather than concrete implementations. Each service has a single, focused responsibility: the API serves data, the worker fetches it, and they communicate through shared domain types without direct coupling. This separation isn't merely organizational; it enables independent scaling, deployment, and testing of each component. The storage module demonstrates another benefit: shared implementations can be extracted and versioned independently, avoiding code duplication while maintaining flexibility. These boundaries make the system

resilient: a worker failure doesn't affect API availability, and each module can evolve at its own pace. With our architecture established, let's verify that these separate modules work together as intended.

11.3.4 Testing the complete system

With all four modules implemented, let's verify that the complete system works as designed. We'll run the worker and API as separate processes, demonstrating how they communicate through the shared storage implementation while remaining independent services.

The workspace makes this integration testing straightforward. Even though we haven't published any module versions yet, both the API and worker can use the local domain and storage modules directly. Go's workspace resolution automatically uses `./domain` and `./storage` from your `go.work` file instead of trying to download published versions from the internet. This means you can iterate rapidly on all modules simultaneously without the ceremony of tagging releases.

Open three terminal windows and run the following commands. We'll start the worker first so it can begin fetching articles, then start the API server, and finally query the API to see the results:

```
# Terminal 1: Run worker to fetch feeds
cd worker
go run ./cmd/worker

# Terminal 2: Run API to serve articles
cd api
go run ./cmd/api

# Terminal 3: Query the API
curl http://localhost:8080/articles?count=5
```

When you start the worker, you'll see output like this:

```
Worker started, fetching feeds every 5 minutes...
Fetched and stored 15 articles from NYT > World
Fetched and stored 23 articles from BBC News
```

The worker immediately fetches feeds on startup, then continues fetching every 5 minutes. Each feed is processed sequentially, and if any feed fails (because of a network timeout or malformed XML, for example), the worker logs the error and continues with the remaining feeds. This graceful degradation is exactly what you want in production: one misbehaving feed shouldn't stop all processing.

In the second terminal, the API server starts and waits for requests:

```
Starting API server on :8080
```

The API doesn't fetch any feeds itself. It simply serves whatever articles the worker has already stored. This separation means API response times are predictable and fast and are never blocked by slow external RSS providers.

In the third terminal, query the API to see the aggregated articles:

```
curl http://localhost:8080/articles?count=5 | jq
```

You'll get a JSON response containing recent articles from all configured feeds:

```
[
  {
    "ID": "https://www.nytimes.com/2026/02/10/world/...",
    "Title": "International Summit Addresses Climate Action",
    "Description": "World leaders gathered to discuss...",
    "Link": "https://www.nytimes.com/2026/02/10/world/...",
    "Published": "2026-02-10T14:30:00Z",
    "FeedTitle": "NYT > World"
  },
  {
    "ID": "https://www.bbc.co.uk/news/world-12345678",
    "Title": "Breaking: Technology Breakthrough Announced",
    "Description": "Scientists have discovered...",
    "Link": "https://www.bbc.co.uk/news/world-12345678",
    "Published": "2026-02-10T13:45:00Z",
    "FeedTitle": "BBC News"
  }
]
```

Notice that articles from the New York Times and BBC feeds appear together, sorted by publication date. The API has aggregated content from multiple sources into a unified view.

Behind the scenes, both services create their own `BoltStore` instance pointing to the same `articles.db` file. Bbolt handles synchronization automatically: the worker's write transactions briefly lock the database exclusively during writes, while the API's read-only view transactions run concurrently without contention. This multiversion concurrency control (MVCC) design means the API serves many users simultaneously while the worker writes periodically.

The workspace architecture shines here too. Both services compile using the local domain and storage modules from your `go.work` file, rather than from the internet. Modify `domain/article.go` to add a field, and both services immediately pick it up on the next `go run`, with no version bumps or dependency updates needed during development.

Try experimenting: query the API at startup (empty results), wait for the worker's first fetch (articles appear), and then restart the worker (no duplicates because article IDs serve as keys). The persistence, concurrent access, and interface-based design work together to demonstrate essential distributed systems concepts in a simple, tangible way.

11.3.5 Versioning independent modules

With our multimodule system built and tested, each component is ready to be versioned and released. Versioning serves as a communication mechanism: when you publish `api/v1.0.0`, you promise that future v1.x releases will remain compatible. This predictability lets teams upgrade dependencies confidently. Version numbers also appear in logs and monitoring, making it easy to identify which version introduced a problem and to roll back if needed.

The real power of independent module versioning becomes clear as your project grows. When you're ready to release, you can version each module independently based on its own change history and stability:

```
# Release updated API with new endpoint
cd api
git add .
git commit -m "Add filtering options to articles endpoint"
git tag api/v1.1.0
git push origin api/v1.1.0

# Release optimized worker
cd worker
git add .
git commit -m "Optimize feed fetching with connection pooling"
git tag worker/v1.1.0
git push origin worker/v1.1.0

# Domain remains at v0.0.1 (no changes needed)#1
```

#1 Marshal each article to JSON bytes.

Each Git tag marks a release point for a specific module. Figure 11.6 illustrates how these tags create independent version histories. The domain module might have tags at domain/v0.0.1, domain/v1.0.0, and domain/v1.1.0, while the API module follows its own timeline with api/v1.0.0, api/v1.1.0, api/v1.2.0, and so on. This independence means teams can release updates at their own pace without coordinating across all modules simultaneously.

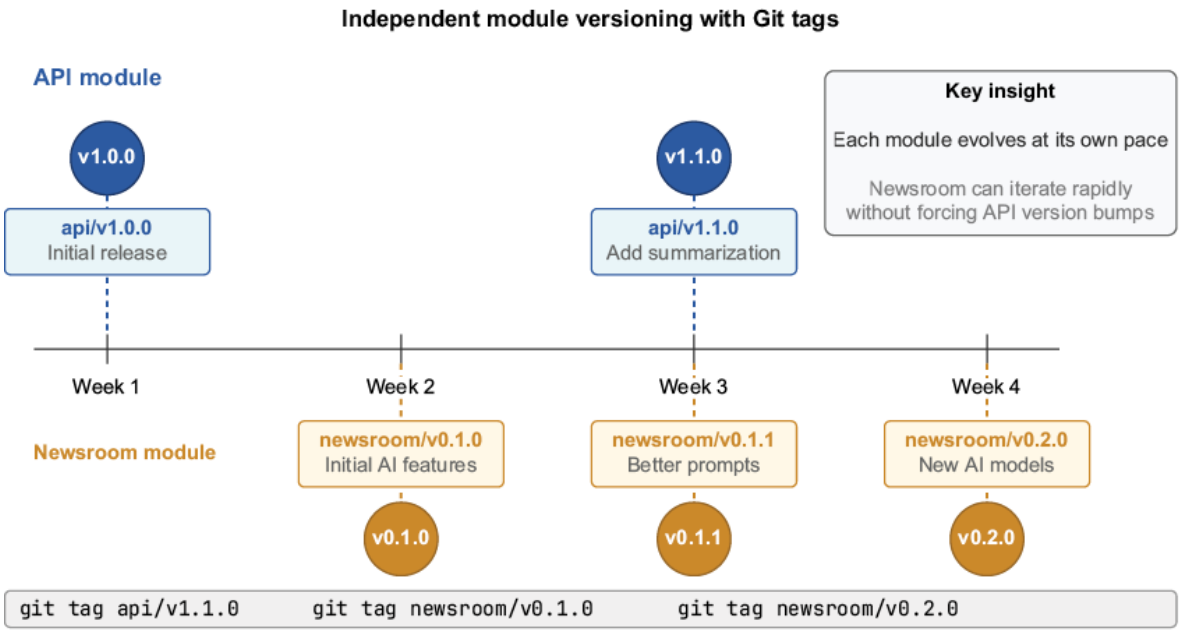


Figure 11.6 Independent module version timelines marked by Git tags

Each module's version communicates its stability and maturity independently. A breaking change to the domain would bump it to v2.0.0, and both API and worker would need updates to their `go.mod` files to adopt the new version. But changes within the API or worker don't affect the other as long as they both satisfy the domain model.

11.3.6 Workspace benefits demonstrated

The workspace we've built demonstrates sustainable software evolution. API and worker modules evolve independently. Each maintains its own `go.mod` file and version while sharing a stable domain model. During development, changes to any module are instantly available to others without publishing tags or updating versions, which dramatically accelerates iteration compared to constantly publishing and updating module versions.

This architectural foundation makes extending the system natural. Appendix A discusses vector search functionality (Qdrant and Ollama), and appendix B introduces LLM-

powered summarization (LangChain), both as independent modules that plug into the same domain interfaces. The clean boundaries between modules (domain defines the model, storage provides persistence, API handles HTTP, worker manages background tasks) follow the dependency inversion principle, where services depend on abstractions rather than implementations.

Workspaces shine when you're building multiple related modules that evolve together, testing changes across module boundaries before publishing, or managing monorepos with independently versioned components. For single-module projects, simpler package organization is enough. For stable external dependencies, just reference the versions in `go.mod`. The key is to recognize workspaces as a development-time tool for coordinating related modules, not as a requirement for every Go project.

Summary

- Packages are Go's fundamental unit of code organization, with strict rules against circular dependencies, organized through directories such as `cmd`, `internal`, and `pkg`.
- Modules group packages that use semantic versioning (vMAJOR.MINOR.PATCH) tracked in `go.mod` files and that work with `go.sum` for cryptographic dependency verification.
- Domain-driven design separates core business logic (entities and interfaces) from infrastructure concerns, creating stable foundations that services depend on.
- Interface-based architecture and the Liskov substitution principle enable flexibility. Any implementation that satisfies an interface works interchangeably, which is critical for testing and swapping implementations.
- Hexagonal architecture uses ports (domain interfaces) and adapters (concrete implementations) to cleanly separate core logic from

infrastructure.

- Workspaces coordinate multiple modules during development through `go.work` files, enabling local cross-module changes without publishing versions while maintaining independent versioning.
- Test doubles enable isolated unit testing by implementing domain interfaces with predetermined data, allowing you to test handlers without databases or external services.
- Dependency inversion structures applications so that high-level modules depend on domain abstractions rather than low-level details, enabling components to evolve independently.
- Multimodule patterns, demonstrated through our RSS reader, show how clean boundaries enable natural system extension with new capabilities.

appendix A Vector search with storage

A growing trend in the software industry is adding deep search and AI capabilities to products. Building on the RSS reader system from chapter 11, this appendix demonstrates how you can add semantic search capabilities by extending the existing storage module. We'll add vector database support directly to the storage implementation, keeping the architecture simple and focused.

To understand why vector search matters, consider how traditional keyword search works. When a user searches for "kubernetes security," a traditional system looks for articles containing those exact words. If an article discusses "pod security policies" or "container hardening" without using the phrase "kubernetes security," it won't be found, even though these topics are highly relevant. Traditional search fails when users phrase queries differently from article text, when synonyms or related terms are used, or when context matters more than exact word matches.

Vector search solves this problem by understanding semantic meaning rather than matching keywords. A *vector* is a numerical representation that captures meaning, most commonly by converting words and sentences into arrays of floating-point numbers. Similar concepts produce similar vectors, even when they use different words. For example (simplified to four dimensions; real embeddings use hundreds of dimensions), "Kubernetes security" might become [0.234, -0.156, 0.891, 0.423], while "Container hardening" becomes [0.241, -0.149, 0.885, 0.437]. These vectors are mathematically close because the concepts are

related. Meanwhile, “Banana recipes” produces a vector like [-0.621, 0.732, -0.123, -0.891], which is far away in vector space, reflecting its completely different meaning.

These vectors are created by embedding models, which are neural networks trained to encode semantic meaning as numerical representations. When you convert both your stored articles and user queries into vectors, you can find semantically similar content by measuring the distance between vectors in this high-dimensional space. The process works in two phases: at index time, you convert each article to a vector embedding and store it; at query time, you convert the search query to a vector, find the stored vectors closest to it, and return the corresponding articles ranked by similarity.

While traditional databases like bbolt excel at exact lookups and CRUD operations, they aren’t designed for this type of similarity search. *Vector databases* are specialized systems optimized for storing and searching high-dimensional vectors. They use specialized algorithms like HNSW (hierarchical navigable small world) or IVF (inverted file index) to quickly find nearest neighbors in vector space, operations that would be prohibitively slow with standard databases, especially as your dataset grows.

For this implementation, we’ll use Qdrant as our vector database and Ollama with the *nomic-embed-text* model for embedding generation. Qdrant is an open source vector database that provides fast similarity search with a simple API. Ollama allows you to run embedding models locally; the nomic-embed-text model generates 768-dimensional vectors and runs entirely on your machine, avoiding cloud dependencies and API costs. The storage module will manage both bbolt and Qdrant internally, providing a

unified interface that writes to both storage backends simultaneously and enables semantic search when needed.

A.1 Docker Compose setup

Vector search requires Qdrant (vector database) and Ollama (embedding generation). Docker Compose provides the cleanest local setup.

A.1.1 Installing Docker

To install Docker on macOS or Windows, download Docker Desktop from https://docs.docker.com/desktop/.

To install Docker on Linux, install it via your package manager as follows:

```
sudo apt-get install docker.io docker-compose-plugin
sudo systemctl start docker
sudo usermod -aG docker $USER  # Log out and back in after this
```

Then verify the installation:

```
docker --version
docker compose version
```

A.1.2 Docker Compose configuration

Create `docker-compose.yml` in your existing repository root for the project from chapter 11:

```
version: '3.8'

services:
  ollama:  #1
    image: ollama/ollama:latest
    ports:
      - "11434:11434"  #2
    volumes:
      - ollama-data:/root/.ollama  #3
    restart: unless-stopped

  qdrant:  #4
    image: qdrant/qdrant:latest
    ports:
      - "6333:6333"  #5
      - "6334:6334"  #6
    volumes:
      - qdrant-data:/qdrant/storage  #7
    restart: unless-stopped

volumes:  #8
  ollama-data:
  qdrant-data:
```

#1 Ollama service runs the embedding model locally (nomic-embed-text).
#2 Port 11434 exposes Ollama's HTTP API for generating embeddings.
#3 Volume persists downloaded models (avoid re-downloading the ~274 MB model).
#4 Qdrant service provides the vector database for similarity search.
#5 Port 6333 is Qdrant's HTTP API (REST interface).
#6 Port 6334 is Qdrant's gRPC API (used by Go client for better performance).
#7 Volume persists vector data and collection configuration.
#8 Named volumes ensure data survives container restarts.

To start our dependencies, we will run the following Docker command. This allows the services we need to run in the background.

```
# Start all services
docker compose up -d

# Download the embedding model (one-time, ~274MB)
docker compose exec ollama ollama pull nomic-embed-text

# Verify Qdrant is running
curl http://localhost:6333/
# Should return: {"title":"qdrant - vector search engine","versio
n":"..."}

#Verify Ollama is running
curl http://localhost:11434/api/tags
```

The services are now ready. Qdrant runs on port 6333 (HTTP), and Ollama runs on port 11434.

A.2 Extending the storage module

Now we'll extend the existing `storage` module from chapter 11 to support vector operations alongside bbolt persistence. The RSS reader system from chapter 11 follows a clean architecture pattern in which the domain package defines interfaces that describe capabilities without specifying implementations. This separation allows us to add vector search functionality while maintaining the existing code structure and keeping concerns properly separated.

We'll use a *decorator pattern* to add search capabilities. Rather than modifying the existing `storage` implementation, we'll create a wrapper that adds vector search on top of any `storage` backend. This keeps `BoltStore` focused solely on persistence, while a separate `SearchStore` adds semantic search capabilities by wrapping it. This approach provides several key benefits: `storage` handles CRUD operations while search handles vectors, and neither needs to know about the other's internals. Any `storage` implementation, whether

bbolt, PostgreSQL, or S3, can gain search capabilities simply by being wrapped. Systems can use plain storage or storage with search without code changes, making search an optional enhancement. Each component can be tested independently, improving testability across the system.

The implementation requires three new domain types that work together to enable semantic search. The `Embedder` interface abstracts text-to-vector conversion, allowing us to switch between embedding services such as Ollama, OpenAI, or others without changing dependent code. The `Searchable` interface defines semantic search operations and returns articles ranked by similarity scores. Finally, the `SearchableStorage` interface combines both `Storage` and `Searchable` through interface composition, allowing code to depend on the interface it needs: a handler that only needs storage can accept `Storage`, while a search handler can require `Searchable`.

Let's start by defining the `Embedder` interface, which provides the contract for converting text into vector embeddings. This interface takes plain text and returns an array of `float32` values representing the semantic meaning of that text.

Create `domain/embedder.go`:

```
package domain

import "context"

// Embedder generates vector embeddings from text
type Embedder interface {
    Embed(ctx context.Context, text string) ([]float32, error)
}
```

Next, we need to define what semantic search looks like in our domain. The `Searchable` interface provides a `Search` method that takes a query string and returns results with similarity scores. Each `SearchResult` pairs an article with a score indicating how closely it matches the query; higher scores mean better matches.

Create `domain/searchable.go`:

```
package domain

import "context"

// SearchResult represents a semantic search match
type SearchResult struct {
    Article *Article
    Score   float32  // Similarity score (0-1, higher = more simila
r)
}

// Searchable enables semantic search on articles
type Searchable interface {
    Search(ctx context.Context, query string, limit int) ([]SearchR
esult, error)
}
```

Finally, we need to update the existing `Storage` interface and create a combined interface that offers both capabilities. The original `Storage` interface focused on CRUD operations but didn't accept a `context.Context` for `AddArticles`. We'll update it to support context-based cancellation and timeouts, which is important when generating embeddings that might take time. The `SearchableStorage` interface then combines both `Storage` and `Searchable` through interface embedding, demonstrating Go's interface composition pattern.

Update `domain/storage.go` to add context support and create the combined interface:

```
package domain

import "context"

type Storage interface {  #1
    AddArticles(ctx context.Context, articles []*Article) error  #2
    GetRecent(n int) []*Article
    GetByID(id string) (*Article, error)
    Close() error
}

search.

type SearchableStorage interface {  #3
    Storage   #4
    Searchable  #5
}
```

#1 Storage interface handles only CRUD operations—no search concerns.
#2 Now accepts context.Context for cancellation and timeout support.
#3 SearchableStorage combines both capabilities through interface composition.
#4 Embeds Storage—provides all CRUD methods
#5 Embeds Searchable—adds semantic search capability

By keeping `Storage` and `Searchable` separate, we can compose them as needed. The `BoltStore` implementation will implement only `Storage`, focusing on what it does best: persisting data to disk. The `SearchStore` wrapper will add `Searchable` by decorating any `Storage` implementation, keeping concerns cleanly separated. This design means existing code that depends on `Storage` continues to work unchanged, while new code that needs search can depend on `SearchableStorage` or just `Searchable`, depending on its requirements.

A.2.1 Installing dependencies

Before implementing the embedding and vector search components, we need to add the necessary dependencies to our storage module.

The Qdrant Go client provides the interface for interacting with the vector database over gRPC:

```
cd storage
go get github.com/qdrant/go-client@latest
go mod tidy
```

Now we'll implement the `Embedder` interface using Ollama's HTTP API. This implementation sends text to the locally running Ollama service and receives a 768-dimensional vector that captures the semantic meaning of that text. The embedder handles connection details, request formatting, and error handling, providing a clean interface for the rest of our code.

Create `storage/ollama_embedder.go` with the structure and constructor:

```
package storage

import (
    "bytes"
    "context"
    "encoding/json"
    "fmt"
    "io"
    "net/http"

    "github.com/YOUR_USERNAME/go-news/domain"
)

type OllamaEmbedder struct {
    baseURL string  #1
    model   string  #2
    client  *http.Client
}

func NewOllamaEmbedder(baseURL, model string) *OllamaEmbedder {
    if baseURL == "" {
        baseURL = "http://localhost:11434"  #3
    }
    if model == "" {
        model = "nomic-embed-text"  #4
    }

    return &OllamaEmbedder{
        baseURL: baseURL,
        model:   model,
        client:  &http.Client{},
    }
}

type embedRequest struct {  #5
    Model  string `json:"model"`
    Prompt string `json:"prompt"`
}

type embedResponse struct {  #6
    Embedding []float32 `json:"embedding"`
}
```

#1 Ollama service URL
#2 The embedding model to use (nomic-embed-text produces 768-dimensional vectors)
#3 Default to localhost on Ollama's standard port.
#4 Default model matches what we downloaded via Docker Compose.
#5 Request structure matches Ollama's API expectations.
#6 Response contains the vector embedding as an array of floats.

The structure defines the embedder type and its constructor, along with the request and response types that match Ollama's API. The constructor provides sensible defaults while allowing configuration for different environments.

Now we'll implement the `Embed` method, which performs the actual conversion from text to vector. This method constructs an HTTP request to Ollama, sends the text for embedding, and decodes the resulting vector.

Add the `Embed` method to `storage/ollama_embedder.go`:

```
func (e *OllamaEmbedder) Embed(ctx context.Context, text string)
([]float32, error) {
    reqBody := embedRequest{
        Model:  e.model,
        Prompt: text,
    }

    jsonData, err := json.Marshal(reqBody)  #1
    if err != nil {
        return nil, fmt.Errorf("marshal request: %w", err)
    }

    req, err := http.NewRequestWithContext(  #2
        ctx,
        "POST",
        e.baseURL+"/api/embeddings",
        bytes.NewBuffer(jsonData),
    )
    if err != nil {
        return nil, err
    }

    req.Header.Set("Content-Type", "application/json")

    resp, err := e.client.Do(req)  #3
    if err != nil {
        return nil, fmt.Errorf("ollama request: %w", err)
    }
    defer resp.Body.Close()

    if resp.StatusCode != http.StatusOK {  #4
        body, _ := io.ReadAll(resp.Body)
        return nil, fmt.Errorf("ollama error %d: %s", resp.StatusCo
de, body)
    }

    var result embedResponse
    if err := json.NewDecoder(resp.Body).Decode(&result); err != ni
l {  #5
        return nil, fmt.Errorf("decode response: %w", err)
    }
```

```
    return result.Embedding, nil  #6
}
```

#1 Convert request to JSON for HTTP transport.
#2 Create HTTP request with context for cancellation support.
#3 Send request to Ollama service.
#4 Check for HTTP errors and return meaningful error messages.
#5 Decode the JSON response containing the embedding vector.
#6 Return the float32 array representing the text's semantic meaning.

This implementation follows standard HTTP client patterns in Go: construct a request, send it, check for errors, and decode the response. The use of `context` allows callers to cancel long-running embedding operations, which is important when processing large batches of articles.

A.2.2 Implementing the Qdrant client

With the embeddings handled, we need a client for interacting with Qdrant's vector database. This client wraps Qdrant's Go SDK to provide three core operations: ensuring that a collection exists with the correct configuration, storing vectors with their metadata, and searching for similar vectors. The client uses Qdrant's gRPC interface for better performance than REST.

Create `storage/qdrant_client.go` with the client structure and connection setup:

```
package storage

import (
    "context"
    "fmt"

    "github.com/google/uuid"
    "github.com/qdrant/go-client/qdrant"
)

const vectorDimension = 768

type QdrantClient struct {
    collectionsClient qdrant.CollectionsClient  #1
    pointsClient      qdrant.PointsClient       #2
    collectionName    string                   #3
}

func NewQdrantClient(ctx context.Context, address, collectionName s
tring) (*QdrantClient, error) {
    if address == "" {
        address = "localhost:6334"  #4
    }
    if collectionName == "" {
        collectionName = "articles"
    }

    conn, err := qdrant.NewClient(&qdrant.Config{  #5
        Host: address,
    })
    if err != nil {
        return nil, fmt.Errorf("connect to qdrant: %w", err)
    }

    client := &QdrantClient{
        collectionsClient: conn,
        pointsClient:      conn,
        collectionName:    collectionName,
    }

    if err := client.ensureCollection(ctx); err != nil {  #6
        return nil, err
    }
```

```
    return client, nil
}

func (qc *QdrantClient) ensureCollection(ctx context.Context) error
{
    exists, err := qc.collectionsClient.Exists(ctx, &qdrant.Collect
ionExistsRequest{  #7
        CollectionName: qc.collectionName,
    })
    if err != nil {
        return fmt.Errorf("unable to check if collection %q exists:
%w", qc.collectionName, err)
    }

    if exists.Result {
        return nil
    }

    _, err = qc.collectionsClient.Create(ctx, &qdrant.CreateCollect
ion{  #8
        CollectionName: qc.collectionName,
        VectorsConfig: qdrant.NewVectorsConfig(&qdrant.VectorParams
{
            Size:     vectorDimension,  #9
            Distance: qdrant.Distance_Cosine,  #10
        }),
    })
    if err != nil {
        return fmt.Errorf("unable to create collection %q: %w", qc.
collectionName, err)
    }
    return nil
}
```

#1 Client for managing collections (create, delete, configure)
#2 Client for managing points (vectors) within collections
#3 Name of the collection storing article vectors
#4 Default to gRPC port (6334) rather than HTTP port (6333).
#5 Connect to Qdrant via gRPC for better performance.
#6 Pass caller’s context so collection setup can be cancelled or timed out.
#7 Check if our articles collection already exists.
#8 Create collection with vector configuration.

#9 Named constant makes the dimension self-documenting and easy to update if the model changes.
#10 Cosine distance measures similarity between vectors (ranges -1 to 1).

The constructor handles the connection setup and ensures the collection is ready to use. By creating the collection automatically, we avoid requiring manual database setup, making the system self-initializing. The collection stores 768-dimensional vectors using cosine similarity, which is ideal for semantic search because it measures the angle between vectors rather than their absolute distance.

With the client initialized and collection configured, we can now implement the operations for storing vectors and searching them. The `Store` method converts articles into Qdrant's point format, while `Search` performs similarity searches.

Add these methods to `storage/qdrant_client.go`:

```
func (qc *QdrantClient) Store(ctx context.Context, id string, vecto
r []float32, metadata map[string]any) error {
    payload := make(map[string]*qdrant.Value)  #1
    for k, v := range metadata {
        payload[k] = valueToQdrant(v)
    }

    point := &qdrant.PointStruct{  #2
        Id:      qdrant.NewIDString(id),  #3
        Vectors: qdrant.NewVectors(vector...),  #4
        Payload: payload,  #5
    }

    _, err := qc.pointsClient.Upsert(ctx, &qdrant.UpsertPoints{  #6
        CollectionName: qc.collectionName,
        Points:         []*qdrant.PointStruct{point},
    })

    return err
}

func (qc *QdrantClient) Search(ctx context.Context, vector []float3
2, limit int) ([]SearchMatch, error) {
    response, err := qc.pointsClient.Search(ctx, &qdrant.SearchPoin
ts{  #7
        CollectionName: qc.collectionName,
        Vector:         vector,  #8
        Limit:          uint64(limit),  #9
        WithPayload:    qdrant.NewWithPayload(true),  #10
    })
    if err != nil {
        return nil, err
    }

    matches := make([]SearchMatch, len(response.Result))
    for i, hit := range response.Result {
        matches[i] = SearchMatch{
            ID:    hit.Id.GetString(),  #11
            Score: hit.Score,  #12
        }
    }

    return matches, nil
```

```
}

type SearchMatch struct {
    ID    string   #13
    Score float32  #14
}

func valueToQdrant(v any) *qdrant.Value {  #15
    switch val := v.(type) {
    case string:
        return qdrant.NewValueString(val)
    case int64:
        return qdrant.NewValueInt(val)
    case float64:
        return qdrant.NewValueDouble(val)
    case bool:
        return qdrant.NewValueBool(val)
    default:
        return qdrant.NewValueString(fmt.Sprintf("%v", v))
    }
}
```

#1 Convert metadata to Qdrant's value format.
#2 A point represents one vector with its metadata in Qdrant.
#3 Use article ID as the point ID for easy retrieval.
#4 The embedding vector (768 floats)
#5 Metadata like title and feed for filtering/debugging
#6 Upsert inserts or updates—idempotent operation
#7 Search finds vectors closest to the query vector.
#8 The query vector to find matches for
#9 Maximum number of results to return
#10 Include metadata in results for debugging.
#11 Article ID for fetching full article from storage
#12 Similarity score (higher is better match)
#13 Links search result back to the article in primary storage
#14 Cosine similarity score indicating relevance
#15 Helper converts Go types to Qdrant's type system.

The `Store` method converts articles into Qdrant's point format, which pairs a vector with metadata and an ID. Using upsert makes the operation idempotent: running it multiple times with the same ID simply updates the vector. The `Search` method takes a query vector and returns the

most similar vectors, ranked by cosine similarity score. This scoring allows us to return results in order of relevance rather than as unordered matches.

A.2.3 Search wrapper Implementation

Now comes the key piece that demonstrates the decorator pattern: wrapping `BoltStore` with vector search capabilities without modifying it. The `SearchStore` type embeds a `Storage` interface, automatically delegating all storage operations to the wrapped implementation. It then overrides `AddArticles` to add embedding generation after storage and implements `Search` to provide semantic search. This design means `SearchStore` knows about storage, but `BoltStore` knows nothing about search: a perfect separation of concerns.

Create `storage/search_store.go` with the structure and constructors:

```
package storage

import (
    "context"
    "fmt"
    "log"

    "github.com/YOUR_USERNAME/go-news/domain"
)

var _ domain.SearchableStorage = (*SearchStore)(nil)  #1

type SearchStore struct {
    domain.Storage                    #2
    embedder       domain.Embedder    #3
    qdrant         *QdrantClient      #4
}

func NewSearchStore(storage domain.Storage, embedder domain.Embedde
r, qdrant *QdrantClient) *SearchStore {
    return &SearchStore{  #5
        Storage:  storage,
        embedder: embedder,
        qdrant:   qdrant,
    }
}


func NewSearchStoreWithDefaults(ctx context.Context, storage domai
n.Storage) (*SearchStore, error) {  #6
    embedder := NewOllamaEmbedder("", "")

    qdrant, err := NewQdrantClient(ctx, "", "")
    if err != nil {
        return nil, fmt.Errorf("connect to qdrant: %w", err)
    }

    return NewSearchStore(storage, embedder, qdrant), nil
}
```

#1 Verify SearchStore implements SearchableStorage (Storage + Searchable).
#2 Embedding Storage interface means all methods auto-delegate to

wrapped storage.
#3 Embedder converts article text into vector representations.
#4 QdrantClient handles all vector database operations.
#5 Constructor allows injecting any storage, embedder, and vector DB.
#6 Convenience constructor with sensible defaults for local development

The embedded `Storage` interface is the key to the decorator pattern. Methods like `GetRecent`, `GetByID`, and `Close` automatically delegate to the wrapped storage without any additional code. This means we only need to implement the methods we want to enhance or add.

A.2.4 Overriding AddArticles for embedding

Now we'll override `AddArticles` to enhance the storage operation with vector embedding generation. The pattern is to first delegate to the wrapped storage to persist the article, then generate and store its embedding.

Add this method to `storage/search_store.go`:

```
func (ss *SearchStore) AddArticles(ctx context.Context, articles []
*domain.Article) error {

    if err := ss.Storage.AddArticles(ctx, articles); err != nil {
#1
        return err
    }


    for _, article := range articles {
        if err := ss.embedAndStore(ctx, article); err != nil {  #2
            log.Printf("Warning: Failed to embed article %s: %v", a
rticle.ID, err)
           // Continue - embedding failures don't break article sto
rage  #3
        }
    }

    return nil
}

func (ss *SearchStore) embedAndStore(ctx context.Context, article *
domain.Article) error {
    text := article.Title
    if article.Description != "" {
        text += "\n\n" + article.Description  #4
    }

    vector, err := ss.embedder.Embed(ctx, text)  #5
    if err != nil {
        return fmt.Errorf("embed text: %w", err)
    }

    metadata := map[string]any{  #6
        "title":      article.Title,
        "feed_title": article.FeedTitle,
    }

    return ss.qdrant.Store(ctx, article.ID, vector, metadata)  #7
}
```

#1 Delegate storage operation to wrapped implementation first.
#2 Then enhance by adding vector embeddings

#3 Embedding failures are logged but don’t fail the write—graceful degradation.
#4 Concatenate title and description for richer semantic meaning.
#5 Generate vector using embedder (calls Ollama).
#6 Include metadata for debugging and potential filtering.
#7 Store vector in Qdrant with article ID as key.

This override demonstrates the decorator’s power: it calls the wrapped storage first to persist the article (delegation) and then adds the enhancement (embedding generation). If embedding fails, we log it but we don’t fail the entire operation; this graceful degradation ensures the system remains functional even if the vector database is temporarily unavailable.

A.2.5 Implementing semantic search

Finally, we’ll implement the `Search` method, which provides the semantic search capability. This method converts the query to a vector, searches Qdrant for similar vectors, and fetches the full articles from storage.

Add this method to `storage/search_store.go`:

```
func (ss *SearchStore) Search(ctx context.Context, query string, li
mit int) ([]domain.SearchResult, error) {
    queryVector, err := ss.embedder.Embed(ctx, query)  #1
    if err != nil {
        return nil, fmt.Errorf("embed query: %w", err)
    }

    matches, err := ss.qdrant.Search(ctx, queryVector, limit)  #2
    if err != nil {
        return nil, fmt.Errorf("vector search: %w", err)
    }

    results := make([]domain.SearchResult, 0, len(matches))
    for _, match := range matches {
        article, err := ss.Storage.GetByID(match.ID)  #3
        if err != nil {
            log.Printf("Warning: Could not fetch article %s: %v", m
atch.ID, err)
            continue
        }

        results = append(results, domain.SearchResult{
            Article: article,
            Score:   match.Score,  #4
        })
    }

    return results, nil
}
```

#1 Convert query text to a vector using the same embedding model used when storing articles.
#2 Find vectors in Qdrant closest to query vector.
#3 Fetch full article data from underlying storage using matched IDs.
#4 Return articles with similarity scores for ranking.

The decorator pattern shines here: `SearchStore` wraps any `Storage` without changing it. Methods like `GetRecent`, `GetByID`, and `Close` automatically delegate to the wrapped storage through the embedded interface. Only `AddArticles` is overridden to add embedding generation, and `Search` is newly implemented. This means `BoltStore` remains simple

and focused, while `SearchStore` adds sophisticated vector search capabilities through composition.

A.3 Using vector search

Now let's see how we can use the enhanced storage from the worker and API. The beauty of the decorator pattern is that both services can continue using the familiar `Storage` interface while transparently gaining search capabilities.

A.3.1 Worker integration

The worker from chapter 11 needs minimal changes: just wrap the base storage with search capabilities and add context to the `AddArticles` call. The wrapping happens at initialization, and from that point forward, every article stored automatically gets embedded and indexed in Qdrant.

Update `worker/cmd/worker/main.go`:

```
package main

import (
    "context"
    ...
)

func main() {
    baseStore, err := storage.NewBoltStore("./articles.db")
    if err != nil {
        log.Fatal(err)
    }
    defer baseStore.Close()

    store, err := storage.NewSearchStoreWithDefaults(ctx, baseStor
e)  #1
    if err != nil {
        log.Printf("Warning: Could not enable search: %v", err)
        log.Println("Continuing with basic storage only")
    }

    ctx := context.Background()
    for _, feedURL := range feeds {
        ...
        if err := store.AddArticles(ctx, feed.Articles); err != nil
{  #2
            log.Printf("Error storing articles: %v", err)
            continue
        }
        ...
    }
}
```

#1 Wrap base storage with search decorator—if this fails, fall back to baseStore.
#2 Pass context to AddArticles for embedding timeout/cancellation support.

The key architectural insight here is *graceful degradation*. If Qdrant or Ollama are unavailable when `NewSearchStoreWithDefaults` is called, the worker logs a warning but continues operating with only the base storage.

This fallback behavior ensures the system remains operational even when search infrastructure fails. Articles are still stored in bbolt; they simply won't be searchable until the vector services are restored.

When `store.AddArticles(ctx, feed.Articles)` is called, the decorator pattern executes a two-phase process:

1. *Persistence*—The call delegates to the wrapped `BoltStore`, which serializes articles to JSON and stores them in bbolt. If this fails, the entire operation fails; article persistence is mandatory.
2. *Embedding*—For each successfully stored article, the decorator concatenates the title and description, sends the text to Ollama for embedding (generating a 768-dimensional vector), and stores that vector in Qdrant with the article ID as the key. If embedding fails for an article, it's logged, but execution continues because search enhancement is optional.

This approach provides *eventual consistency* for search: articles appear immediately in storage but might lag slightly in search results if embedding is slow or temporarily unavailable.

A.3.2 API usage

The API demonstrates interface segregation by creating handlers that depend only on the interfaces they need. The search handler depends on `Searchable`, while article handlers depend on `Storage`. Both are satisfied by the same `SearchStore` instance.

Create `api/internal/handlers/search_handler.go`:

```
package handlers

import (...)

type SearchHandlers struct {
    searchable domain.Searchable  #1
}

func NewSearchHandlers(searchable domain.Searchable) *SearchHandler
s {
    return &SearchHandlers{searchable: searchable}
}

func (sh *SearchHandlers) HandleSearch(w http.ResponseWriter, r *ht
tp.Request) {
    query := r.URL.Query().Get("q")
    if query == "" {
        http.Error(w, "Missing query parameter 'q'", http.StatusBad
Request)
        return
    }

    limitStr := r.URL.Query().Get("limit")
    limit := 10
    if limitStr != "" {
        if l, err := strconv.Atoi(limitStr); err == nil {
            limit = l
        }
    }

    results, err := sh.searchable.Search(r.Context(), query, limit)
#2
    if err != nil {
        http.Error(w, "Search failed", http.StatusInternalServerErr
or)
        return
    }

    w.Header().Set("Content-Type", "application/json")
    json.NewEncoder(w).Encode(map[string]any{
        "query":   query,
        "results": results,
```

```
    })
}
```

#1 Only depends on Searchable interface—doesn't need full Storage
#2 Perform semantic search using the query text.

The `SearchHandlers` struct demonstrates *interface segregation*: it depends only on the `Searchable` interface because that's all it needs. This handler doesn't store articles, query by ID, or list recent articles; it only performs semantic searches. By depending on the minimal interface, we make the handler easier to test (mock implementations need fewer methods) and more flexible (any type implementing `Searchable` works, not just `SearchStore`).

The search process flows as follows:

1. Extract the query parameter `q` from the URL (e.g., "kubernetes security").
2. Call `searchable.Search(r.Context(), query, limit)`, which does the following:
   - Sends the query text to Ollama to generate a query vector.
   - Searches Qdrant for vectors similar to the query vector
   - Returns article IDs ranked by similarity score
3. For each ID, fetch the full article from the underlying storage.
4. Return articles with their similarity scores in the response.

This handler accepts only the `Searchable` interface, showing that it doesn't need full storage access; it only performs searches. This keeps dependencies minimal and makes testing easier.

Now wire everything together in the API's main function. The key change from chapter 11 is wrapping the base storage with search capabilities and adding the search handler.

Update `api/cmd/api/main.go`:

```
package main

...

func main() {
    baseStore, err := storage.NewBoltStore("./articles.db")
    if err != nil {
        log.Fatal(err)
    }
    defer baseStore.Close()

    store, err := storage.NewSearchStoreWithDefaults(context.Backgr
ound(), baseStore)  #1
    if err != nil {
        log.Fatal(err)
    }

    articleHandlers := handlers.NewArticleHandlers(store)  #2
    http.HandleFunc("/articles", articleHandlers.HandleArticles)
    http.HandleFunc("/articles/recent", articleHandlers.HandleRecen
t)

    searchHandlers := handlers.NewSearchHandlers(store)  #3
    http.HandleFunc("/search", searchHandlers.HandleSearch)

    log.Println("API server starting on :8080")
    log.Fatal(http.ListenAndServe(":8080", nil))
}
```

#1 Wrap base storage with search decorator.
#2 Existing handlers from chapter 11 work unchanged with wrapped storage.
#3 New search handler uses only the Searchable interface.

> **NOTE** This is interface segregation in action. The same `store` instance satisfies different interface requirements for different handlers. `ArticleHandlers` receives it as `Storage`, giving access to `GetRecent`, `GetByID`, `AddArticles`, and `Close`, while `SearchHandlers` receives it as `Searchable`,

giving access only to `Search`. This is possible because `SearchStore` implements `SearchableStorage`, which combines both interfaces through embedding.

The API architecture ensures the following:

- Traditional REST endpoints (`/articles/recent`) continue working exactly as they did in chapter 11.
- The new search endpoint (`/search`) provides semantic search capabilities.
- Both use the same underlying storage instance, ensuring consistency.
- If vector search fails, traditional endpoints remain unaffected.

This demonstrates a key benefit of the decorator pattern: *backward compatibility*. Existing code that depends on `Storage` works unchanged, while new code can opt into enhanced capabilities by depending on `Searchable` or `SearchableStorage`.

A.3.3 Running the complete system

With all the components in place, let's walk through the full life cycle of running the system and understanding what happens at each step.

STEP 1: START DOCKER

Run the following command:

```
# Start Docker services
docker compose up -d
```

This command starts both Qdrant (a vector database) and Ollama (an embedding service) in detached mode as background processes. Docker Compose creates the following:

- *Qdrant container*—Listens on ports 6333 (HTTP API) and 6334 (gRPC API). The gRPC port is what our Go client uses for better performance. Data persists in a Docker volume named `qdrant-data`.
- *Ollama container*—Listens on port 11434 (HTTP API). This service manages embedding models and provides the API to generate vectors. Downloaded models persist in the `ollama-data` volume.

Verify the services are running:

```
# Check Qdrant health
curl http://localhost:6333/
# Returns: {"title":"qdrant - vector search engine","versio
n":"..."}

# Check Ollama status
curl http://localhost:11434/api/tags
# Returns: {"models":[]} initially (no models downloaded yet)
```

STEP 2: DOWNLOAD THE EMBEDDING MODEL

Now we will leverage the internal commands built into the Ollama container to download the models we need. The `nomic-embed-text` model is used to create the vectors necessary for us to search in the future.

```
# Download nomic-embed-text model (~274MB)
docker compose exec ollama ollama pull nomic-embed-text
```

The size of the nomic-embed-text embedding model is about 274 MB so it is fairly light-weight compared to other models. While we are using this model for this example take some time to explore other options to see what will work best for your project. The model only needs to be downloaded once; it persists in the Docker volume, so subsequent restarts of the Ollama container will reuse it.

STEP 3: RUN THE WORKER TO POPULATE STORAGE

With the environment in place, we are now able to start our background consumer, which will index articles as they come in, using the embedding model we downloaded earlier.

```
cd worker
go run ./cmd/worker
```

The worker executes the following workflow:

1. *Initialize storage*—Creates `articles.db` (a bbolt database file) if it doesn't exist
2. *Wrap with search*—Connects to Qdrant (port 6334) and Ollama (port 11434) and creates the search decorator
3. *Fetch feeds*—Downloads RSS/Atom feeds from configured URLs
4. *Store articles*—For each article, it does the following:
   - Stores article JSON in bbolt under the "articles" bucket
   - Concatenates title + description into text
   - Sends text to Ollama and receives a 768-float vector in return
   - Stores the vector in Qdrant's "articles" collection with the article ID
5. *Log progress*—Outputs counts such as "Stored 45 articles from Go Blog"

After completing this section, you should have

- An `articles.db` file containing article data in bbolt format
- A Qdrant collection, "articles," containing embeddings for semantic search
- Both data stores synchronized with matching article IDs

STEP 4: START THE API SERVER

With the background worker downloading and indexing articles, we can now run the API to perform searches.

```
cd ../api
go run ./cmd/api
```

The API server starts on port 8080 and provides three HTTP endpoints:

- `GET /articles/recent?n=10`—Returns recent articles (traditional storage query)
- `GET /articles/:id`—Retrieves a specific article by ID (traditional storage query)
- `GET /search?q=kubernetes&limit=5`—Semantic search (vector search query)

When we start the server, it needs to do a few things to get started before a call can be made:

1. Opens the `articles.db` file (read-only mode)
2. Wraps storage with search capabilities (connects to Qdrant/Ollama)
3. Creates article handlers (depending on the `Storage` interface)
4. Creates search handlers (depending on the `Searchable` interface)
5. Registers HTTP routes and starts the server

STEP 5: TEST TRADITIONAL AND SEMANTIC SEARCH

First, test the traditional endpoint that queries bbolt directly:

```
# Get 5 most recent articles
curl "http://localhost:8080/articles/recent?n=5"
```

This queries bbolt, deserializes all the articles, sorts them by publication date, and returns the top five. The response includes article titles, descriptions, links, and publication dates.

Now test the semantic search:

```
# Search for articles about Kubernetes security
curl "http://localhost:8080/search?q=kubernetes%20security&limit=5"
```

This executes a more complex workflow:

1. *Query embedding*—Sends "kubernetes security" to Ollama and receives a 768-dimensional vector representing the semantic meaning of this phrase
2. *Vector search*—Queries Qdrant using cosine similarity to find the five vectors closest to the query vector
3. *Score ranking*—Qdrant returns article IDs sorted by similarity score (higher = more similar)
4. *Article retrieval*—For each ID, fetches the full article from bbolt
5. *Response*—Returns articles with their similarity scores

The semantic search finds articles about "pod security policies," "container isolation," "cluster hardening," and other related topics even if they don't contain the exact words "kubernetes" and "security" together. This demonstrates the power of vector search: it captures semantic similarity between concepts, surfacing related content even when the exact keywords don't match.

UNDERSTANDING THE DATA FLOW

The complete system demonstrates several architectural patterns working together. Each component maintains *separation of concerns* through a focused responsibility: Docker Compose manages infrastructure (Qdrant and Ollama), the worker handles data ingestion (fetching feeds, storing articles, and generating embeddings), the API handles queries (both traditional REST endpoints and semantic search), the storage module abstracts persistence (bbolt) and search (Qdrant), and the domain module defines interfaces that serve as contracts for all components.

The *decorator pattern* is central to the architecture. The `SearchStore` wraps any `Storage` implementation, adding search capabilities without modifying the underlying storage. This design means BoltStore can be tested independently without Qdrant, `SearchStore` can wrap different storage backends (PostgreSQL, S3, etc.), systems can opt into search by wrapping or use plain storage, and code that depends on `Storage` works unchanged with or without the decorator.

Interface segregation ensures that different handlers depend only on the interfaces they need. Article handlers depend on `Storage` for CRUD operations, while search handlers depend on `Searchable` for search operations only. Both interfaces are satisfied by the same `SearchStore` instance, with each handler seeing only the methods it requires. This minimal-dependency approach makes testing easier and components more flexible.

The system implements *graceful degradation* to handle infrastructure failures. If Qdrant or Ollama fails, the worker logs warnings but continues storing articles in bbolt, the API continues serving traditional endpoints, the search endpoint returns errors while other endpoints work, and the system remains operational with reduced functionality. This resilience ensures core operations continue even when enhanced features are unavailable.

Finally, the *dual write pattern* ensures consistency across storage systems. Articles are written to two systems with a clear priority: the primary write goes to bbolt (the source of truth for article data), while the secondary write goes to Qdrant (the index for semantic search). Secondary failures are logged but don’t fail the operation, providing eventual consistency for search. This approach guarantees that

article data is always persisted while search capabilities catch up when available.

The search endpoint finds articles by semantic meaning, not just keyword matches!

Summary

- Vector search finds content by semantic meaning rather than exact keyword matches, using embedding models that convert text into high-dimensional numerical vectors that represent meaning.
- The decorator pattern allows you to add search capabilities to any existing `Storage` implementation without modifying it. `SearchStore` wraps the base storage, delegating CRUD operations while adding embedding generation and search.
- Interface segregation keeps dependencies minimal: handlers depend only on `Storage` (for CRUD) or `Searchable` (for search), not both. The `SearchableStorage` interface combines them through composition when needed.
- The dual write pattern maintains consistency across storage systems: bbolt serves as the source of truth for article persistence, while Qdrant provides the search index. Secondary write failures don't fail the operation, preserving eventual consistency.
- Graceful degradation ensures system resilience: if the vector search infrastructure (Qdrant or Ollama) fails, the system continues operating with traditional storage, logging warnings while maintaining core functionality.
- Qdrant (vector database) and Ollama (embedding service) enable local semantic search capabilities without cloud dependencies, using Docker Compose for straightforward local development and deployment.

appendix B LLM-powered summarization

In appendix A, we added `/search`, giving the newsreader the ability to find articles by using fuzzy search with vectors instead of relying solely on keywords. The key infrastructure for that was Qdrant, a vector database that stores high-dimensional representations of each article, and Ollama, a local model server running `nomic-embed-text` to convert article text into those vectors. But returning a ranked list of article titles still puts the burden on the reader to figure out what matters. This appendix takes the next step: using those search results to generate a human-readable summary with a large language model (LLM).

Qdrant will continue to provide retrieval, finding articles related to the topic we want to summarize. Ollama will gains second responsibility: alongside running `nomic-embed-text` for embedding, it will also serve `llama3`, a text generation model that synthesizes retrieved articles into prose. The `domain.Storage` and `domain.Searchable` interfaces from appendix A require no changes; the new summary handler consumes them directly. Our focus is to add a new `summarizer` package and a new HTTP handler that uses LLMs while keeping our design flexible for the future.

By the end of this appendix, we'll have a `/summary` endpoint that works in two modes. Called without parameters, it will produce a brief digest of our most recent articles. Called with a topic, such as `?prompt=technology`, it will use the vector search pipeline from appendix A to find relevant articles and ask the LLM to synthesize them into a focused narrative.

B.1 LLM technologies

A large language model (LLM) is a neural network trained on enormous amounts of text. It learns the statistical relationships between words well enough to generate coherent, contextually relevant prose. For our purposes, the key property is *abstractive summarization*: an LLM doesn't just extract existing sentences; it generates new text that synthesizes meaning from multiple sources.

Consider three articles: one covering a major AI framework release, one on new data privacy legislation, and one on an industry acquisition. A keyword extractor pulls sentences from each independently. An LLM reads all three, recognizes that the framework release and privacy legislation both point to a shift in how data is processed and governed, and writes a paragraph connecting them, then addresses the acquisition as market context. That's what makes an LLM worth the extra complexity.

One of the most useful patterns in applied AI is *retrieval-augmented generation* (RAG): using a search system to find relevant documents, then feeding those documents to an LLM to generate a response grounded in real data. That's exactly what our `/summary?prompt=X` endpoint will implement.

A query like "What's new in AI research?" enters the system and is converted to a vector by `nomic-embed-text`. Qdrant finds the ten most semantically similar articles stored from the worker's last run. Those articles are assembled into a prompt and sent to `llama3`, which reads them and generates a coherent paragraph. The user receives a focused summary rather than a ranked list of links.

This matters because it grounds the LLM's output in our actual data. Without retrieval, an LLM would draw on its

training knowledge, which may be stale or irrelevant to our feed. With retrieval, every claim in the summary comes from an article our worker fetched. Figure B.1 shows the flow through the system.

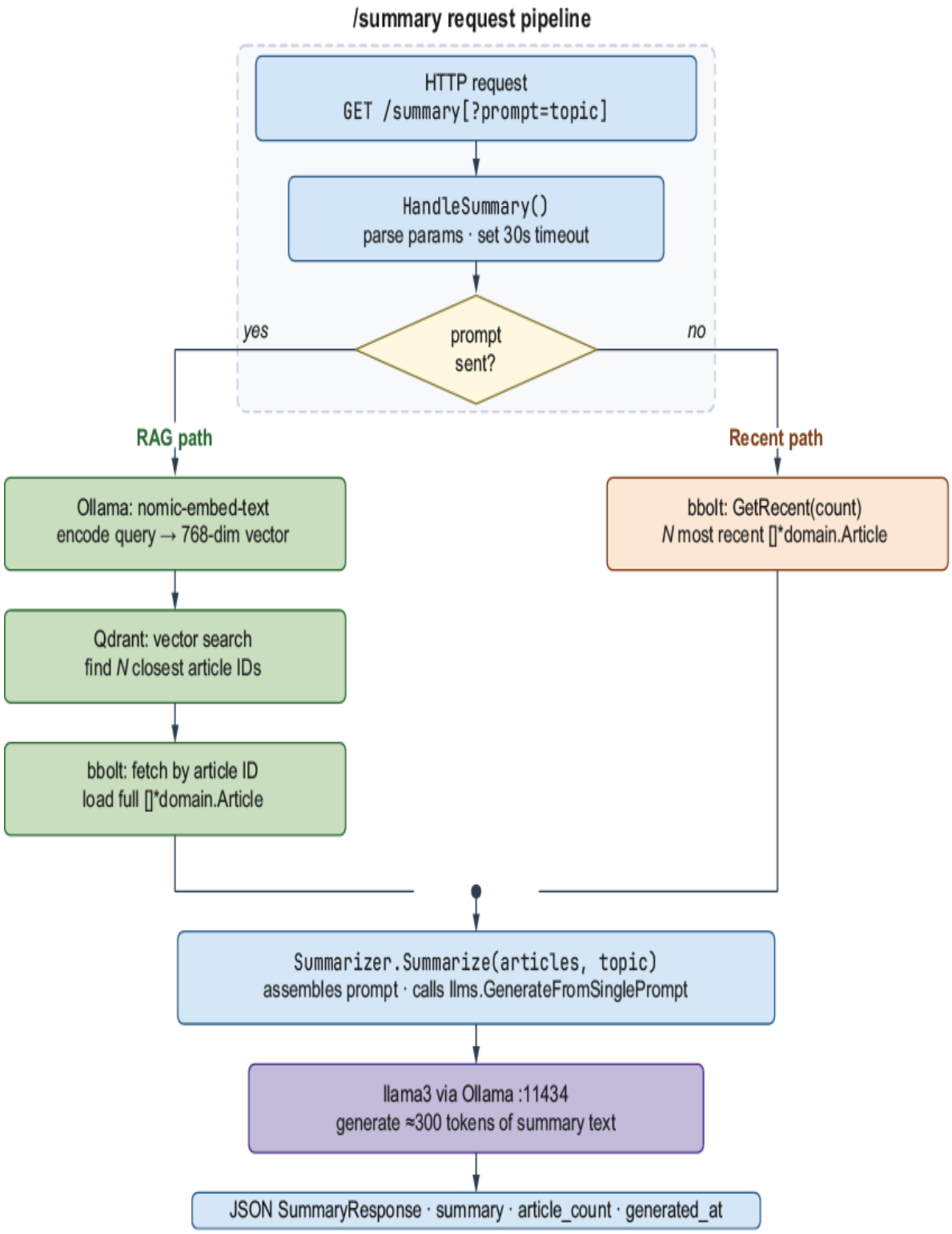


Figure B.1 RAG request pipeline: the two code paths through the system. When a topic is provided, the query is embedded by nomic-embed-text, and Qdrant retrieves the most semantically similar articles; those articles are then sent to llama3 for synthesis into prose. Without a topic,

the most recent articles are fetched directly from bbolt and passed to llama3, skipping the vector search step entirely.

B.1.1 Ollama: Running models locally

Ollama is a local model runtime: a server that manages downloading, loading, and serving AI models on your own hardware. When we run `docker compose up`, Ollama starts as a background service on port `11434` and exposes an HTTP API that lets you send input to a loaded model and get its generated output back. This process of running a model to produce a result is called inference. We interact with it by sending requests to that API, either directly or through a library.

The practical advantage of running models locally is that nothing leaves your machine. Cloud LLM providers (OpenAI, Anthropic, Google) require API keys, they charge per token, and they send your data to external servers. Ollama gives you the same text generation capabilities for free, with complete data privacy. The tradeoff is hardware: a GPU improves throughput dramatically, but Ollama can run on a CPU, just more slowly.

In appendix A, Ollama was already serving the `nomic-embed-text` model inside Docker for embedding, converting article text into the 768-dimensional vectors stored in Qdrant. In this appendix, we'll download a second model, `llama3`, for text generation. Both models run on the same Ollama server concurrently with no changes to Docker Compose. Ollama automatically schedules inference requests between models.

We'll use Llama 3 8B (`llama3`) because it sits at a practical sweet spot for local development: strong enough to produce coherent summaries and small enough to run on most developer machines without a GPU.

> **NOTE** If your machine has less than 8 GB of RAM, use `llama3.2:3b` instead. It's smaller and faster, though with some quality cost. On a machine with a GPU, generation time drops from 2–10 seconds to under 1 second.

B.1.2 LangChain: A consistent interface for LLMs

`langchaingo` (`github.com/tmc/langchaingo`) is the Go port of LangChain, a library built to simplify writing applications that use LLMs. It gives us two specific things. First, it provides a uniform `llms.Model` interface across providers: whether the backend is Ollama, OpenAI, or Anthropic, the call to generate text looks identical in our code. If we later move to a cloud model, only the initialization in `New()` changes; `Summarize()` and `buildPrompt()` stay exactly the same. Second, it provides generation functions like `llms.GenerateFromSinglePrompt` that handle the HTTP request lifecycle, response streaming, and parameter encoding, so our code stays focused on prompts and article assembly rather than HTTP boilerplate.

A lighter alternative would be to call Ollama's HTTP API directly with the standard library. We'll use `langchaingo` because provider portability is a realistic goal: prompt--tuning and quality work done against a local Ollama model will transfer to OpenAI with a one-line change, which matters as projects move from development toward production.

B.2 Infrastructure setup

Our Docker services from appendix A are unchanged. The only new step for this appendix is downloading the Llama 3 model. `nomic-embed-text` should already be present from

appendix A; the `pull` command will skip it if it is already cached in the `ollama-data` volume.

```
docker compose exec ollama ollama pull llama3  #1
docker compose exec ollama ollama list         #2
```

#1 Downloads llama3 (~4.7GB), cached in the ollama-data volume after the first pull
#2 Confirms both models are available. Expected output shows llama3:latest and nomic-embed-text:latest.

With both models downloaded, run a quick sanity check before writing any code. This will confirm that Ollama has loaded the model and can respond to inference requests. Expect a coherent single-sentence response about collecting and organizing articles from multiple sources. If Ollama returns an error, check that the container is running with `docker compose ps`.

```
docker compose exec ollama ollama run llama3 "Summarize what a news
aggregator does in one sentence."
```

B.3 Creating the summary module

The `summarizer` package is kept as a separate module rather than added directly to the API package for the same reason we isolated the storage layer in chapter 11: it has its own dependency (`langchaingo`) and its own configuration surface, and it could be reused by a future worker or CLI tool without pulling in the entire HTTP layer. It communicates through `domain.Article` values, so the LLM integration stays behind a clear boundary that the rest of the project never needs to cross.

From the projects repository root we will need to add an additional module:

```
mkdir -p summarizer/cmd/summarizer
cd summarizer
go mod init github.com/YOUR_USERNAME/go-news/summarizer
cd ..
go work use ./summarizer

cd summarizer
go get github.com/tmc/langchaingo@latest
go get github.com/YOUR_USERNAME/go-news/domain@latest
go mod tidy
```

B.3.1 Configuration and initialization

Create `summarizer/summarizer.go`. The package has two types: `Config` carries the Ollama connection settings, and `Summarizer` holds the live LLM client. We'll start with the types and default configuration:

```
package summarizer

import (
    ...
    "github.com/tmc/langchaingo/llms"
    "github.com/tmc/langchaingo/llms/ollama"
)

type Config struct {
    OllamaURL string // e.g. "http://localhost:11434"
    Model     string // e.g. "llama3"
}

func DefaultConfig() Config {
    return Config{
        OllamaURL: "http://localhost:11434",
        Model:     "llama3",
    }
}

type Summarizer struct {
    llm llms.Model  #1
}
```

#1 llms.Model is langchaingo's provider-agnostic interface; only New() changes when switching backends.

In the following code, `New` creates the Ollama client and returns a ready-to-use `Summarizer`. The model itself is not loaded into memory until the first inference call, so startup stays fast even though `llama3` is a 4.7 GB model:

```
func New(config Config) (*Summarizer, error) {
    ...
    llm, err := ollama.New(
        ollama.WithServerURL(config.OllamaURL),
        ollama.WithModel(config.Model),
    )
    if err != nil {
        return nil, fmt.Errorf("initialize Ollama client: %w", err)
    }
    return &Summarizer{llm: llm}, nil
}
```

B.3.2 The Summarize method

The `Summarize` method is where the LLM interaction happens. Two parameters worth understanding before reading the code are `WithTemperature` and `WithMaxTokens`.

Temperature controls how deterministic or predictable the model's output will be. At 0.0, the model always selects the statistically most likely next word, producing consistent but sometimes repetitive text. At 1.0, it samples more broadly, which increases creativity but can also lead to incoherent output. A value of 0.7 is a practical middle ground for prose summarization: the output reads naturally without wandering off topic.

Tokens are the units the model processes internally, each roughly three-quarters of an English word. Setting `WithMaxTokens(300)` limits the response to approximately 200 to 250 words, and generation time scales roughly linearly with this value, so increasing it for longer summaries means increasing the request context deadline proportionally.

```
// Summarize generates a natural language summary of the provided ar
ticles.
func (s *Summarizer) Summarize(ctx context.Context, articles []*doma
in.Article, focusTopic string) (string, error) {
    if len(articles) == 0 {
        return "", fmt.Errorf("no articles to summarize")
    }

    result, err := llms.GenerateFromSinglePrompt(
        ctx,
        s.llm,
        s.buildPrompt(articles, focusTopic),
        llms.WithTemperature(0.7), #1
        llms.WithMaxTokens(300),   #2
    )
    if err != nil {
        return "", fmt.Errorf("generate summary: %w", err)
    }

    return strings.TrimSpace(result), nil
}
```

#1 Controls output randomness; 0.7 balances readability with natural variation.
#2 Limits the response to roughly 200-250 words; increase alongside the context timeout for longer summaries

B.3.3 Building the prompt

The prompt is the primary control surface for LLM behavior. The same model, articles, and temperature setting can produce substantially different output depending on the instructions in the prompt. Extracting the templates as package-level constants keeps them easy to find and tune without touching the assembly logic:

```
const topicPrompt = `You are summarizing news articles related to: %
[1]s

Focus on information relevant to "%[1]s". Create a cohesive summary
that answers
what's new or important about this topic. If articles cover differen
t aspects,
organize your summary logically.`

const generalPrompt = `You are summarizing recent news articles.

Create a brief news digest covering the key stories. If articles are
related,
connect them. If they cover different topics, mention each briefly.
Keep it concise and informative.`

func (s *Summarizer) buildPrompt(articles []*domain.Article, focusTo
pic string) string {
    var sb strings.Builder

    if focusTopic != "" {
        fmt.Fprintf(&sb, topicPrompt, focusTopic) #1
    } else {
        sb.WriteString(generalPrompt) #2
    }

    sb.WriteString("\n\nArticles to summarize:\n\n")
    for i, article := range articles {
        fmt.Fprintf(&sb, "%d. Title: %s\n", i+1, article.Title)
        if article.Description != "" {
            fmt.Fprintf(&sb, "   Summary: %s\n", article.Descriptio
n)
        }
        sb.WriteString("\n")
    }

    sb.WriteString("Your summary (2-3 paragraphs):\n") #3
    return sb.String()
}
```

#1 System instruction that steers the model’s attention to the topic before it reads the articles
#2 Neutral framing produces an even-handed digest; connecting related articles is part of the instruction

#3 Completion cue: the trailing newline signals where the model's response should begin.

B.4 Wiring it into the API

With the `summarizer` package complete, we need to connect it to the HTTP layer. This section creates the summary handler, which sits between the HTTP server and the `summarizer` and registers it on the mux alongside the handlers from chapter 11 and appendix A. The handler is the only part of the API that knows about the `summarizer` package; the rest of the server is unchanged.

B.4.1 What the handler needs

The summary handler depends on two interfaces: `domain.Storage` for `GetRecent(n int)` and `domain.Searchable` for `Search(ctx, query, limit)`. Our `BoltStore` already satisfies both. The handler accepts them as separate parameters, even though the same concrete value satisfies both, making each dependency explicit and allowing either to be swapped independently in the future.

Critically, the handler never imports anything related to Qdrant or embeddings; it passes a plain string to `Search` and receives `[]domain.SearchResult` back. The vector machinery stays inside the storage layer, where it belongs, as outlined in figure B.2.

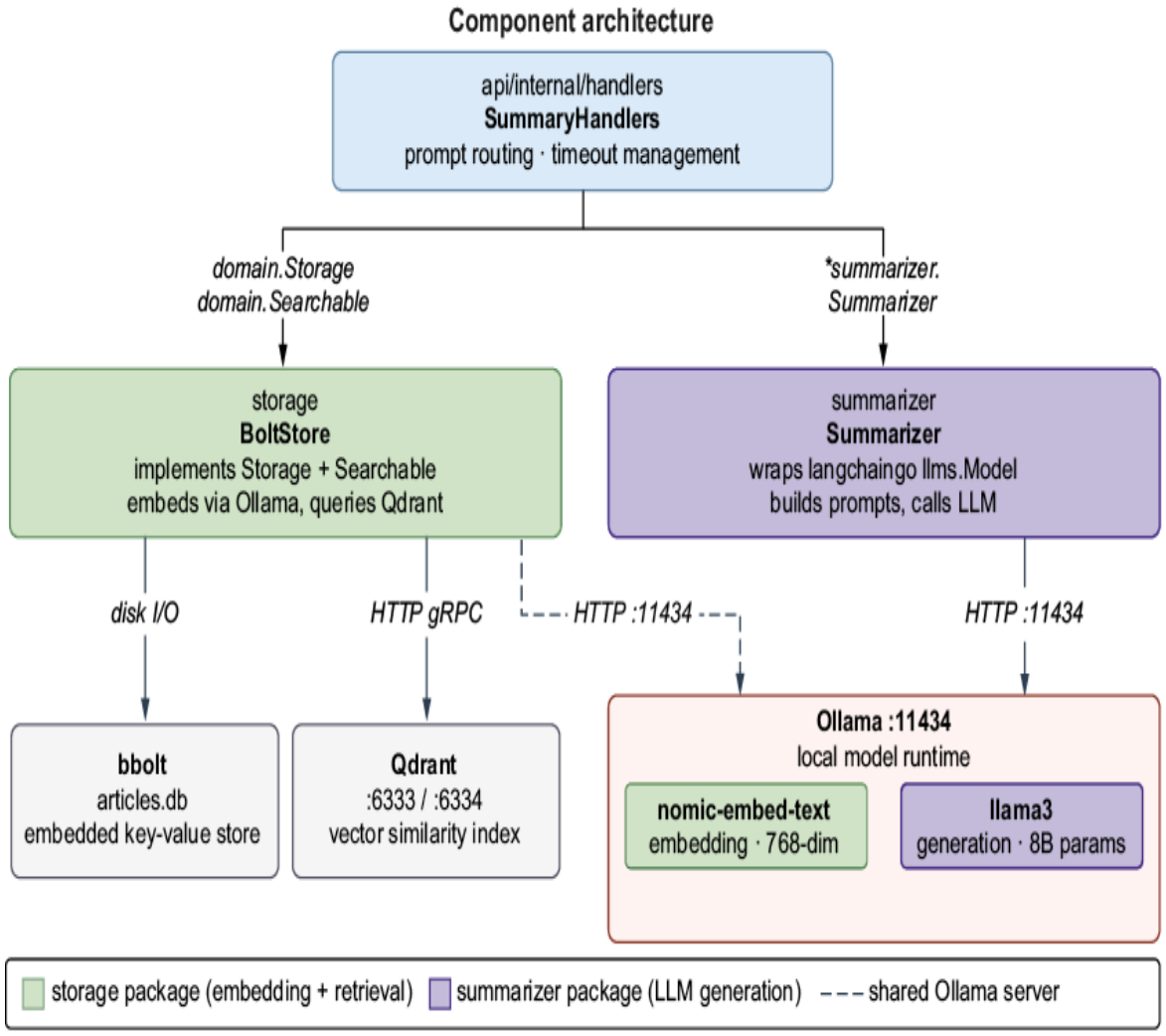


Figure B.2 Component architecture showing how the packages depend on one another and the external services they each talk to

B.4.2 The summary handler

Create `api/internal/handlers/summary_handler.go`:

```
package handlers

import (
    "context"
    "encoding/json"
    "log"
    "net/http"
    "strconv"
    "time"

    "github.com/YOUR_USERNAME/go-news/domain"
    "github.com/YOUR_USERNAME/go-news/summarizer"
)

type SummaryHandlers struct {
    storage    domain.Storage
    searcher   domain.Searchable
    summarizer *summarizer.Summarizer
}

// NewSummaryHandlers creates handlers with AI capabilities.
// In practice, storage and searcher will be the same *BoltStore val
ue.
func NewSummaryHandlers(
    storage domain.Storage,
    searcher domain.Searchable,
    sum *summarizer.Summarizer,
) *SummaryHandlers {
    return &SummaryHandlers{storage: storage, searcher: searcher, su
mmarizer: sum}
}

type SummaryResponse struct {
    Summary      string    `json:"summary"`
    ArticleCount int       `json:"article_count"`
    FocusTopic   string    `json:"focus_topic,omitempty"`
    GeneratedAt  time.Time `json:"generated_at"`
}
```

`SummaryHandlers` holds the three dependencies the handler needs: `domain.Storage` for fetching recent articles by recency, `domain.Searchable` for vector search, and

`*summarizer.Summarizer` for LLM generation. `NewSummaryHandlers` accepts them as separate typed parameters so that each dependency is named and verified by the compiler at the call site rather than discovered at runtime. `SummaryResponse` is the JSON shape returned by every call to the endpoint; the `focus_topic` field uses `omitempty` so it is omitted entirely from the response when no topic was provided, keeping the general-digest response clean.

`HandleSummary` is where those dependencies are put to use. The method reads the optional `prompt` and `count` query parameters, then routes the request down one of two code paths.

```
// HandleSummary handles GET /summary.
//
// Query parameters:
//   prompt=<topic>  - find articles about this topic via vector sea
rch, then summarize
//   count=<n>       - number of articles to use (default 10, max 2
0)
func (sh *SummaryHandlers) HandleSummary(w http.ResponseWriter, r *h
ttp.Request) {
    if r.Method != http.MethodGet {
        http.Error(w, "Method not allowed", http.StatusMethodNotAllo
wed)
        return
    }

    prompt := r.URL.Query().Get("prompt")
    count  := 10
    if s := r.URL.Query().Get("count"); s != "" {
        if n, err := strconv.Atoi(s); err == nil && n > 0 && n <= 20
{
            count = n
        }
    }

    // 30 seconds gives the LLM enough time on CPU hardware.
    ctx, cancel := context.WithTimeout(r.Context(), 30*time.Second)
    defer cancel()

    var articles []*domain.Article

    if prompt != "" {
        // Query-focused path: delegate to Searchable, which interna
lly embeds the
        // query string, queries Qdrant, and fetches full articles f
rom bbolt.
        results, err := sh.searcher.Search(ctx, prompt, count)
        if err != nil {
            log.Printf("search error: %v", err)
            http.Error(w, "Failed to search articles", http.StatusIn
ternalServerError)
            return
        }
        articles = make([]*domain.Article, 0, len(results))
```

```
        for _, result := range results {
            articles = append(articles, result.Article)
        }
    } else {
        // General path: most recent articles from bbolt.
        articles = sh.storage.GetRecent(count)
    }

    if len(articles) == 0 {
        http.Error(w, "No articles available to summarize", http.Sta
tusNotFound)
        return
    }

    summary, err := sh.summarizer.Summarize(ctx, articles, prompt)
    if err != nil {
        log.Printf("summarize error: %v", err)
        http.Error(w, "Failed to generate summary", http.StatusInter
nalServerError)
        return
    }

    log.Printf("summary: %d articles, topic=%q", len(articles), prom
pt)

    w.Header().Set("Content-Type", "application/json")
    if err := json.NewEncoder(w).Encode(SummaryResponse{
        Summary:      summary,
        ArticleCount: len(articles),
        FocusTopic:   prompt,
        GeneratedAt:  time.Now().UTC(),
    }); err != nil {
        log.Printf("encode response: %v", err)
    }
}
```

The handler routes every request through one of two distinct code paths based on the presence of `prompt`. When `prompt` is empty, it calls `sh.storage.GetRecent(count)`, a direct bbolt lookup that returns in milliseconds. When `prompt` is provided, it delegates to `sh.searcher.Search(ctx, prompt, count)`, which quietly runs the full RAG pipeline: it embeds the query with

Ollama (`nomic-embed-text`), searches Qdrant for similar vectors, and fetches each matched article from bbolt. From `HandleSummary`'s perspective, all of this is a single function call returning `[]domain.SearchResult`. The complexity lives where it belongs, inside the storage layer.

The 30-second context timeout in `HandleSummary` acts as the hard ceiling on total request time. LLM generation on CPU hardware typically takes 2 to 10 seconds for a 300-token response; the embedding and vector search path adds another 100 to 300 ms. If we increase `MaxTokens` in `buildPrompt`, the context timeout should increase proportionally. The relationship between this value and the server's `WriteTimeout` is covered next.

B.4.3 Updating the API

The `summarizer` package and handler are complete. The last step is registering the new handler in `main.go`. The changes are entirely additive: two new environment variables for the LLM configuration, a `summarizer.New` call that uses the same Ollama URL already read for the storage layer, and a fourth route on the mux. Everything from chapter 11 and appendix a remains untouched.

Start with configuration and storage setup, which is unchanged from appendix A:

```
package main

import (
    "fmt"
    "log"
    "net/http"
    "os"
    "time"

    "github.com/YOUR_USERNAME/go-news/api/internal/handlers"
    "github.com/YOUR_USERNAME/go-news/storage"
    "github.com/YOUR_USERNAME/go-news/summarizer"
)

func main() {
    dbPath           := getEnv("DB_PATH", "./data/news.db")  #1
    qdrantHost       := getEnv("QDRANT_HOST", "localhost")
    qdrantCollection := getEnv("QDRANT_COLLECTION", "articles")
    ollamaURL        := getEnv("OLLAMA_URL", "http://localhost:1143
4")
    llmModel         := getEnv("LLM_MODEL", "llama3")

    articleStore, err := storage.NewBoltStore(dbPath, storage.Config
{
        QdrantHost:       qdrantHost,
        QdrantCollection: qdrantCollection,
        OllamaBaseURL:    ollamaURL,
    })
    if err != nil {
        log.Fatalf("storage: %v", err)
    }
    defer articleStore.Close()
    log.Printf("Storage: %s", dbPath)
    ...
```

#1 All configuration is read from environment variables with localhost defaults for local development; set these in the environment or a .env file for production.

With the store open, we initialize the summarizer. Notice that `summarizer.New` receives the same `ollamaURL` value already used by `storage.NewBoltStore`. Both packages talk to the same Ollama server, one for embedding and one for generation,

but neither package imports the other. The shared URL flows in as plain configuration, keeping the two packages fully independent. The handler wiring that follows uses the same pattern from chapter 11: construct each handler group with its dependencies, register routes, and start the server.

```
sum, err := summarizer.New(summarizer.Config{  #1
        OllamaURL: ollamaURL,
        Model:     llmModel,
    })
    if err != nil {
        log.Fatalf("summarizer: %v", err)
    }
    log.Printf("Summarizer: model=%s", llmModel)

    articleHandlers := handlers.NewArticleHandlers(articleStore)
    searchHandlers  := handlers.NewSearchHandlers(articleStore)
    summaryHandlers := handlers.NewSummaryHandlers(articleStore, art
icleStore, sum)  #2

    mux := http.NewServeMux()
    mux.HandleFunc("/articles",        articleHandlers.HandleArticle
s)
    mux.HandleFunc("/articles/recent", articleHandlers.HandleRecent)
    mux.HandleFunc("/search",          searchHandlers.HandleSearch)
    mux.HandleFunc("/summary",         summaryHandlers.HandleSummar
y)

    server := &http.Server{
        Addr:         fmt.Sprintf(":%s", getEnv("PORT", "8080")),
        Handler:      mux,
        ReadTimeout:  30 * time.Second,
        WriteTimeout: 60 * time.Second,  #3
    }

    log.Printf("Listening on %s", server.Addr)
    if err := server.ListenAndServe(); err != nil {
        log.Fatalf("server: %v", err)
    }
}

func getEnv(key, defaultValue string) string {
    if v := os.Getenv(key); v != "" {
        return v
    }
    return defaultValue
}
```

#1 Same Ollama server as appendix A; no new infrastructure required
#2 articleStore satisfies both interfaces; the compiler verifies this at the

call site.
#3 Must exceed the handler's 30-second context deadline to allow full response delivery

Notice that `summarizer.New` receives the same `ollamaURL` already used by the storage layer. Both packages talk to the same Ollama server, one for embedding and one for generation, but neither imports the other; the URL is plain configuration. Passing `articleStore` twice to `NewSummaryHandlers` works because it satisfies both `domain.Storage` and `domain.Searchable`; the compiler verifies this at the call site with no type assertions needed.

The `WriteTimeout` on the server and the context deadline in the handler are related but distinct. `WriteTimeout` is enforced by the HTTP server and caps the time allowed to write the complete response, protecting against slow or stalled clients. The context deadline inside the handler caps application-level work: if the LLM or Qdrant takes too long, the handler surfaces a clean error rather than blocking indefinitely. Both values need to remain larger than the expected LLM generation time, so if we switch to a larger model, we increase both.

Now add the dependency:

```
cd api
go get github.com/YOUR_USERNAME/go-news/summarizer@latest
go mod tidy
```

B.5 End-to-end test

With all components in place, you can start the infrastructure and confirm that both models are available:

```
docker compose up -d
docker compose exec ollama ollama list
```

When running the preceding `docker` command, the output should be a list of available models for Ollama to use. Both `llama3:latest` and `nomic-embed-text:latest` must appear in the list before continuing in order for the application to work. `nomic-embed-text` handles embedding, converting text to vectors for storage and search; `llama3` handles generation, synthesizing article text into a summary. Both run on the same Ollama server on port 11434. If either model is missing, pull it with `docker compose exec ollama ollama pull <model-name>`.

Next, populate the article store by running the worker:

```
cd worker && go run ./cmd/worker
```

The worker runs the same dual-write process from appendix A: fetching RSS feeds, storing articles in bbolt, and generating vector embeddings in Qdrant. The summarizer does not change this step; it consumes articles the worker has already collected. After the worker completes, `articles.db` should be on disk, and the `articles` collection in Qdrant should be populated.

With the data in place, you can start the API server:

```
cd ../api && go run ./cmd/api
```

The server starts on port 8080 and logs its configuration:

```
Storage: ./data/news.db
Summarizer: model=llama3
Listening on :8080
```

The server now exposes four endpoints. The first three are from chapter 11 and appendix A; the fourth is new:

- `GET /articles/recent?n=10`—Recent articles from bbolt

- `GET /articles/:id`—Single article by ID
- `GET /search?q=<query>&limit=5`—Semantic search via Qdrant
- `GET /summary[?prompt=<topic>&count=<n>]`—LLM-generated summary

Test the general digest first:

```
curl "http://localhost:8080/summary"
```

This exercises the no-prompt code path: the handler calls `storage.GetRecent(10)`, retrieves the 10 most recent articles directly from bbolt without involving Qdrant, and passes them to the summarizer. Expect a 2- to 5-second response time on CPU hardware; bbolt reads are instant, but LLM generation dominates. The response includes the summary text, article count, and a `generated_at` timestamp.

To see the full RAG pipeline in action, run both the topic-focused summary and the raw search for the same query back-to-back:

```
curl "http://localhost:8080/summary?prompt=technology%20news"
curl "http://localhost:8080/search?q=technology%20news&limit=5"
```

The `/search` response shows the raw ranked list: article titles with similarity scores. The `/summary` response takes those same articles and synthesizes them into prose. Comparing the two makes the value of the summarization step concrete: the same articles are presented in two very different ways.

When `prompt=technology news` is set, the handler executes the full RAG pipeline: it sends the query to Ollama for embedding, searches Qdrant for the nearest vectors, fetches the matched articles from bbolt, and then sends those articles to `llama3` for summarization. Expect the process to take 3 to 7 seconds total. The extra time compared to the

general digest comes from the embedding and vector search overhead, which is typically 100 to 300 ms.

Here is a successful summary response:

```
{
  "summary": "The technology sector saw a flurry of activity this week. Several major AI research labs published findings on improved reasoning in smaller models, suggesting that raw parameter count matters less than training methodology. On the open-source front, a new framework for building LLM-powered applications gained significant community traction, with contributors citing its straightforward Go API as a key differentiator.",
  "article_count": 6,
  "focus_topic": "technology news",
  "generated_at": "2024-02-17T10:27:33Z"
}
```

B.6 What to try next

The most instructive next step is to experiment with model quality directly. Set `LLM_MODEL=llama3.2:3b` and rerun the same queries you used during testing. The smaller model is noticeably faster on CPU hardware, but it produces shallower summaries, and seeing that tradeoff firsthand will give you an intuition for model selection that no benchmark can replicate. From there, try tuning the prompt in `buildPrompt`: add a "respond in bullet points" instruction, specify a reading level, or tell the model to lead with the most important story. Prompt changes have an outsized effect on output quality and cost nothing to iterate on.

For more substantial extensions, `langchaingo` supports streaming responses. Instead of waiting for the full summary before writing the HTTP response, you can write tokens as they arrive and dramatically improve perceived latency. Caching is another high-value addition: if `/summary?prompt=kubernetes` is called repeatedly, storing the result for an

hour saves several seconds of LLM time per request with no loss in quality for most use cases. Finally, try stopping the Qdrant container while the API is running. `/articles` and `/articles/recent` should keep working, `/summary?prompt=X` should return a clean error, and `/summary` with no prompt should still succeed. Verifying graceful degradation across the system is the best way to build confidence in the layering established across both appendices.

Summary

- Retrieval-augmented generation (RAG) grounds LLM output in real data: the `/summary?prompt=X` endpoint uses vector search to find relevant articles first and then synthesizes them with the LLM, ensuring that every claim in the summary comes from your actual feed.
- The `Summarizer` package wraps Ollama's `llms.Model` interface through `langchaingo`. Swapping to a different LLM provider (OpenAI or Anthropic, for example) requires changing only the initialization in `New()`; `Summarize()` and `buildPrompt()` are unchanged.
- Interface segregation carries forward from appendix A: `SummaryHandlers` depends on `domain.Storage` for article retrieval and `domain.Searchable` for vector search, accepting them as separate parameters even though one concrete value satisfies both.
- Prompt engineering is the primary control surface for LLM behavior. The same model, same articles, and same code produce substantially different output depending on whether `focusTopic` is set. Adjusting the prompt is the first tool to reach for when output quality isn't where you want it.
- Temperature and token limits are the primary dials for controlling LLM output. Temperature governs randomness (0.0 = deterministic, 1.0 = highly creative), while token limits govern response length and generation time. Tuning both is a normal part of integrating an LLM, not a one-time setup step.
- Ollama serves both the embedding model (`nomic-embed-text`) and the generation model (`llama3`) from the same port. No additional

infrastructure is required beyond what appendix A already established.

index

SYMBOLS

A

C

O