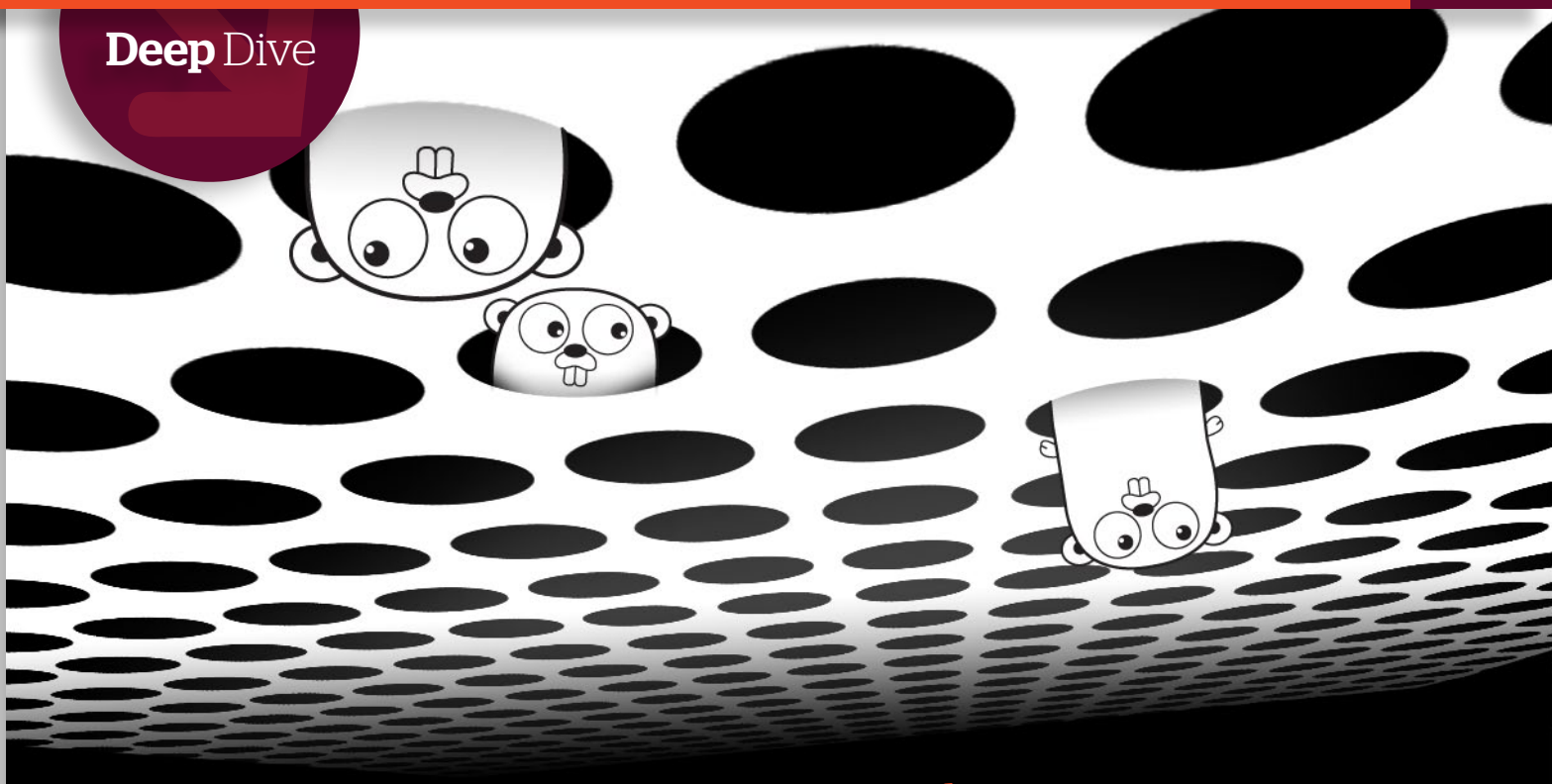
InfoWorld DeepDive

What you need to know about Google Go

GO GOPHER CHARACTER: RNEE FRENCH

Deep Dive



Get concurrent with the Go language

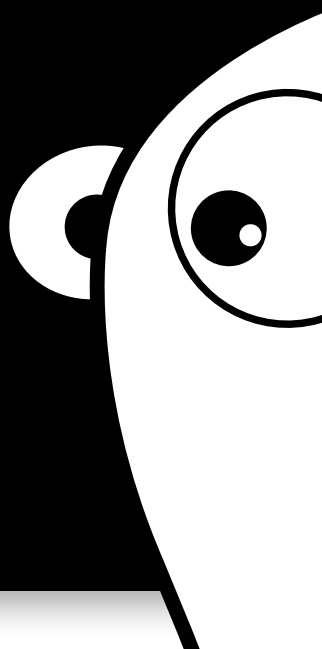
Google's Go language is perfect for today's distributed application development, enabling complex operations across multiple services to run concurrently.

BY SIMON BISSON

What happens when the people behind one of the most influential operating systems of the last 50 years or so decide they don't like what's happened to the language they wrote their OS in?

It turns out they decide to rebuild it their own way. That's what happened when Unix creators Rob Pike and Ken Thompson (who wrote the original B language that formed the foundation of the entire family of C-derived languages we use) felt it was time to develop a new systems-level programming language, with a focus on speeding up software development.

Working with Robert Griesmer at Google, they created Go, a C-derived language with static typing (using type inference where necessary to ensure type-safe code), along with such features as garbage collection that are usually associated with dynamic interpreted languages such as JavaScript or byte-code languages such as Java or C#.



Deep Dive

In the five or so years since Go was announced, it's become the basis of many production systems at Google, because it contains constructs required for concurrent programming — making it a useful tool for cloud-scale distributed application development.

Go from the ground up

Like C, Go programs are made up of packages, with program execution handled by the main package. Imported packages and libraries can be called from here. A simple naming convention handles exported names — names in a package are only exported if they start with a capital letter.

Typing is handled by a mix of static and dynamic techniques, so your code is type-safe, but it's easy to write and understand. For example, if two or more consecutive parameters in a function have the same type, you need to declare the type only once.

One interesting feature is the ability to define your own types, allowing you to create specific types for recurring variables — for example, declaring a type that's an ITU-standard telephone number. Declaring a number that's not the right format as a telephone number would return an error, allowing you to write code that handles data checks at a very low level.

If you've programmed in C or C++, it's not difficult to get started with Go. There are a lot of similarities, and where differences emerge, it's not hard to pick up alternate ways of working — for example, the way in which Go uses interfaces instead of inheritances.

Perhaps the biggest difference between Go and other modern languages is how it handles errors. There's no try-catch construct, because the designers felt that would make the code too complex. However, Go's multi-value returns mean there's an error return type that functions and methods can use, which makes it possible to write your own error-handling code.

Working together

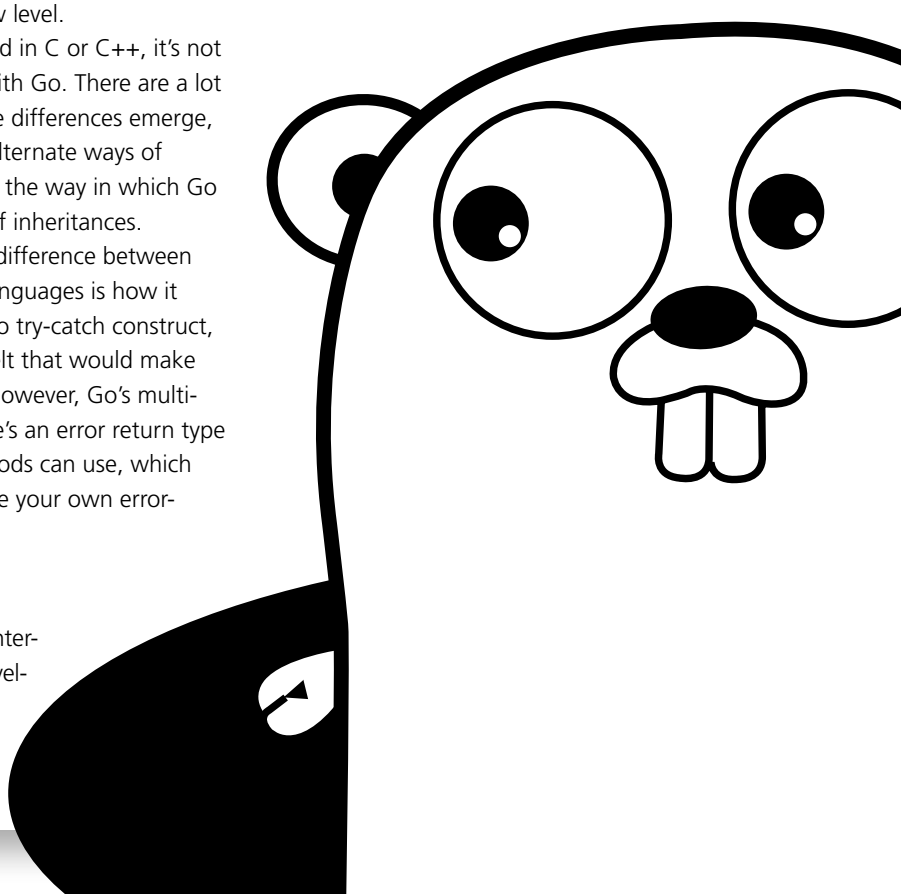
Where things get really interesting for cloud-scale development is in Go's explicit design for concurrent programming.

Based around the concept of Communicating Sequential Processes, Go uses a model similar to that of such functional languages as Erlang or Occam (the language developed for INMOS's innovative multiprocessing Transputer processor). The result is a series of separate blocking functions implemented as lightweight processes, known as goroutines.

Goroutines can be moved to nonblocked threads to allow a program to continue to run. Keeping goroutines separate from threads allows a Go program to use much less memory, minimizing the use of system resources. It's also an automatic process, so there's no need for developers to write code to handle blocking events.

Goroutines can be connected by channels, simple structures that let you send values between concurrent processes. This is where Go's process communication comes into play; it gives you the tools you need to pass messages between processes to trigger appropriate actions. Channels are synchronized, type safe, and work with send and receive statements. Data is sent and received over defined channels, with channels blocked until the communication operation has completed.

If you've programmed in C or C++, **it's not difficult to get started with Go.**



Deep Dive

What Go is good for

The result is a set of language tools that can be used to build distributed applications, using channels to link goroutines and to handle how data is passed between them.

There's the option of making channels buffered, for sending larger amounts of data — letting you work with streaming data from IoT devices, for example. A buffered channel will send data until it fills, at which point it blocks until emptied by the receiving process. Similarly, a receiver will block until it receives data, giving you a mechanism for basic flow control for streamed data. Once data has been transferred a sender can close a channel, though in practice, you'll keep a channel open for additional data transfers.

Using concurrent techniques like these means that code can scale from one CPU to many to a cloud-scale cluster of machines.

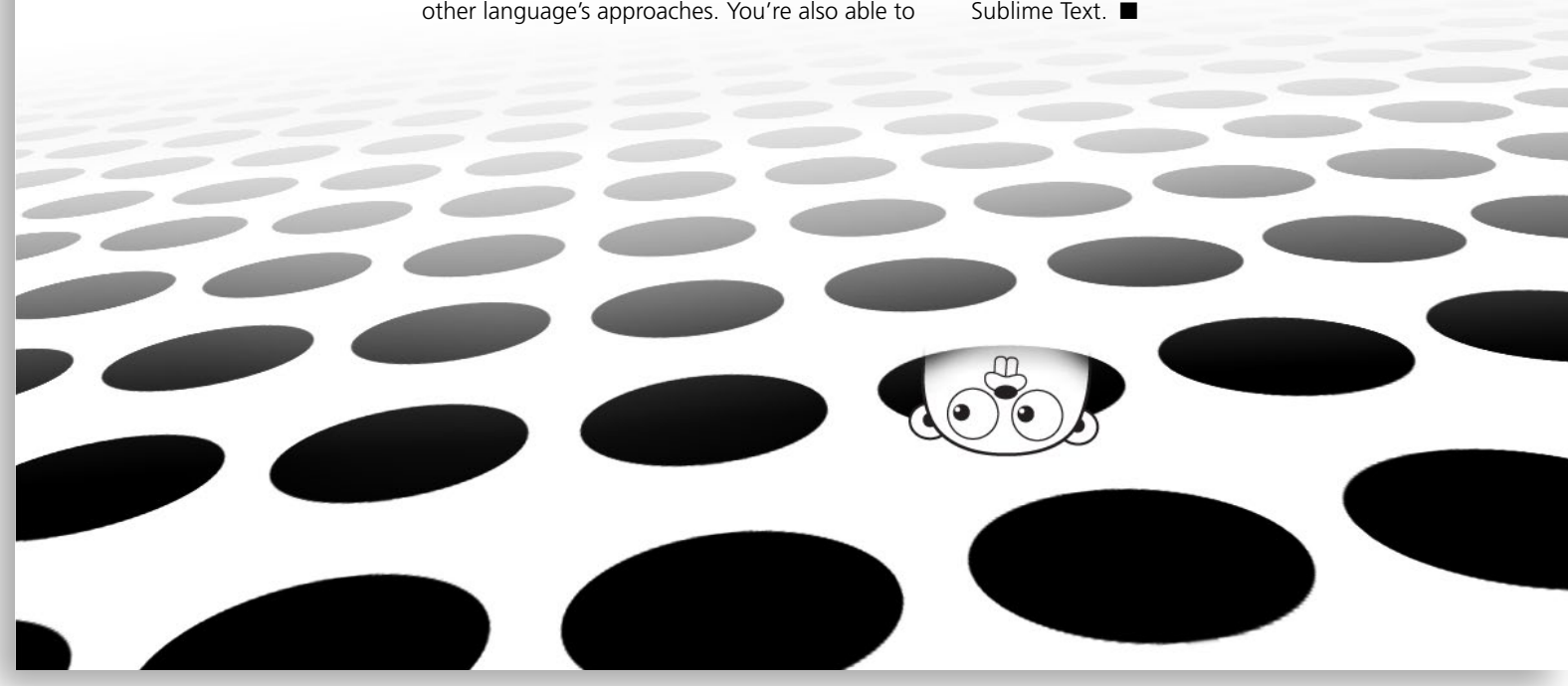
Tools like this make Go a useful tool for building servers — and microservices. You can use Go's select statement to manage communication across channels, with goroutines that open new channels as they unblock and picking channels at random if several are needed at the same time. It's an interesting approach to building parallel code and compares well with other language's approaches. You're also able to build timeouts into channels, so they don't halt execution waiting to send data.

One of Go's creators, Rob Pike, compares the concurrency model in Go to the world outside the computer — where independent pieces interact. Using concurrent techniques like these means that code can scale from one CPU to many to a cloud-scale cluster of machines. It's a philosophy Pike encapsulates as "don't communicate by sharing memory, share memory by communicating."

You can see why Google uses Go. It allows complex operations across multiple services to run concurrently, so a search can be working with Web, image, and video goroutines, bringing in results from them all and delivering them in one function return. The resulting code is not only faster (because it's not sequential), it's also more robust: If one search channel doesn't respond, you can open another.

Go is now one of the standard languages for Google's App Engine PaaS, with compilers for most common OSes, including Android. There's also a live playground for trying out Go code on the Go website (it's also built into the language tutorial so that you can try out code snippets as you go).

If you want to work with Go, then you'll find plenty of development tooling support, with cross-platform IDEs widely available, as well as support in modern code editors like Sublime Text. ■

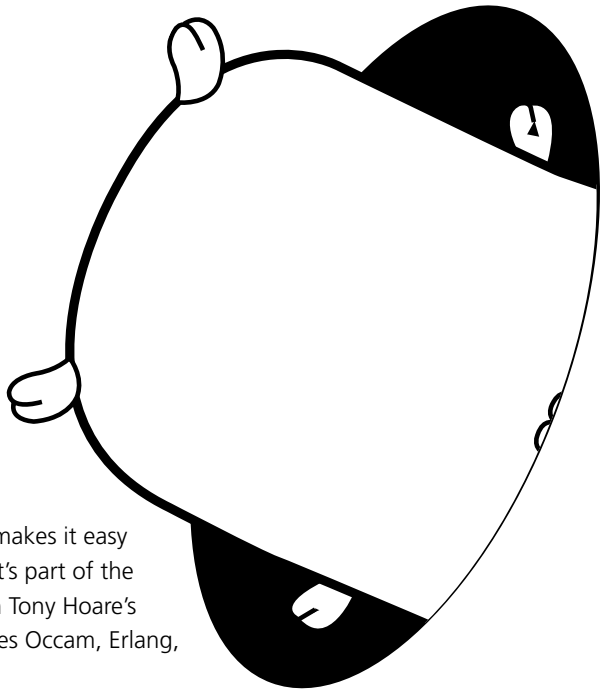


Deep Dive

A quick guide to Google Go programming

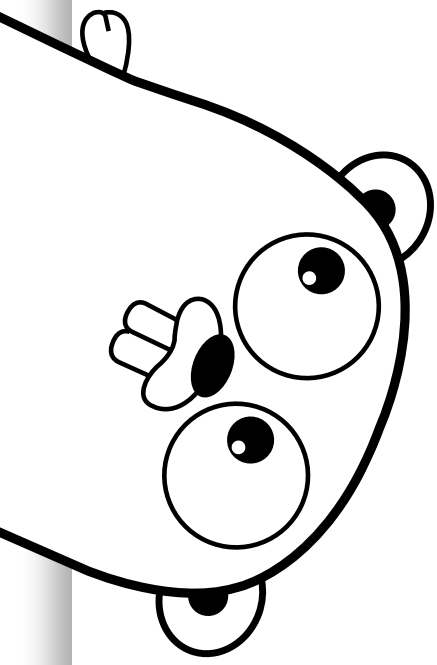
Need a concise, simple, safe, and fast compiled language with wonderful concurrency features? Go with Google

BY MARTIN HELLER



The open source Go programming language makes it easy to build simple, reliable, and efficient software. It's part of the programming language lineage that started with Tony Hoare's Communicating Sequential Processes and includes Occam, Erlang, Newsqueak, and Limbo.

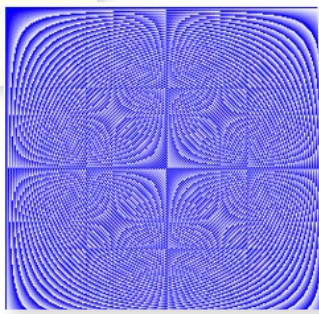
Here, we'll demonstrate some of the differentiating features of the language, including its extremely lightweight concurrency. The Go project currently has more than 500 contributors, led by Rob Pike, a distinguished engineer at Google, who worked at Bell Labs as a member of the Unix team and co-created Plan 9 and Inferno.



Beyond arrays: Slices

The Go language extends the idea of arrays with slices. A slice points to an array of values and includes a length. `[]T` is a slice with elements of type T. In the pictured exercise, we use slices of slices of unsigned bytes to hold the pixels of an image we generate. With `package main`, programs start running. The `import` statement is an extended version of C and C++'s `include` statement; here we are getting the `pic`file from a Mercurial repository. The `:=` syntax declares and initializes a variable, and the compiler infers a type whenever it can. Also, `make` is used to create slices and some other types. A `for..range` loop is the equivalent of C#'s `for..in` loop.

```
1  package main
2
3  import "code.google.com/p/go-tour/pic"
4
5  func Pic(dx, dy int) [][]uint8 {
6      slice := make([][]uint8,dy)
7      for i:=range slice {
8          slice[i] = make([]uint8,dx)
9          for j:=range slice[i] {
10             slice[i][j] = uint8(i*j)
11         }
12     }
13     return slice
14 }
15
16 func main() {
17     pic.Show(Pic)
18 }
```



Deep Dive

```go
package main

import "fmt"

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

    m["Answer"] = 42
    fmt.Println("The value:", m["Answer"])

    m["Answer"] = 48
    fmt.Println("The value:", m["Answer"])

    delete(m, "Answer")
    fmt.Println("The value:", m["Answer"])

    v, ok := m["Answer"]
    fmt.Println("The value:", v, "Present?", ok)
}
```

Map statements

The Go `map` statement maps keys to values. As with `slice`, you create a `map` with `make`, not `new`. In the example above, we are mapping string keys to integer values. Here we demonstrate inserting, updating, deleting, and testing for `map` elements.

The pictured program prints:

```
The value: 42
The value: 48 The value: 0
The value: 0
Present? False
```

```go
package main

import (
    "fmt"
    "math"
)

type Vertex struct {
    X, Y float64
}

func (v *Vertex) Abs() float64 {
    return math.Sqrt(v.X*v.X + v.Y*v.Y)
}

func main() {
    v := &Vertex{3, 4}
    fmt.Println(v.Abs())
}
```

Structs and methods

The Go language lacks classes but has a `struct`, which is a sequence of named elements, called fields, each with a name and a type. A `method` is a function with a receiver. A method declaration binds an identifier (the method name) to a method and associates the method with the receiver's base type. In this example, we declare a `Vertex struct` to contain two floating point fields, `X` and `Y`, and a method `Abs`. Fields that begin with uppercase letters are public; fields that begin with lowercase letters are private. Fields and methods are addressable through the dot notation; `*` and `&` signify pointers, as in C. This program prints `5`.

Deep Dive

```
type Abser interface {
    Abs() float64
}

func main() {
    var a Abser
    f := MyFloat(-math.Sqrt2)
    v := Vertex{3, 4}

    a = f  // a MyFloat implements Abser
    a = &v // a *Vertex implements Abser

    // In the following line, v is a Vertex (not *Vertex)
    // and does NOT implement Abser.
    a = v

    fmt.Println(a.Abs())
}

type MyFloat float64

func (f MyFloat) Abs() float64 {
    if f < 0 {
        return float64(-f)
    }
    return float64(f)
}

type Vertex struct {
    X, Y float64
```

Interfaces

An interface type is defined by a set of methods. A value of interface type can hold any value that implements those methods. In this example, we define an **`interface Abser`** and a variable a of type **`Abser`**. Note that the assignments in lines 17 and 18 work, but the assignment in line 22 does not even compile. The **`Abs`** method of **`Vertex`**, which we saw in the previous slide, has a pointer to **`Vertex`** type for its receiver, so a **`*Vertex`** implements **`Abser`**, but a **`Vertex`** does not.

```
package main

import (
    "fmt"
    "runtime"
)

func main() {
    fmt.Print("Go runs on ")
    switch os := runtime.GOOS; os {
    case "darwin":
        fmt.Println("OS X.")
    case "linux":
        fmt.Println("Linux.")
    default:
        // freebsd, openbsd,
        // plan9, windows...
        fmt.Printf("%s.", os)
    }
}
```

Switch statements

The **`switch`** statement in Go is similar to the **`switch`** statement in other C-like languages, except that the **`case`** statements can be types or expressions in addition to simple values, and the cases automatically break unless they end with **`fallthrough`** statements. The cases are evaluated in the order they are defined.

```
package main

import (
    "fmt"
    "time"
)

func say(s string) {
    for i := 0; i < 5; i++ {
        time.Sleep(100 * time.Millisecond)
        fmt.Println(s)
    }
}

func main() {
    go say("world")
    say("hello")
}
```

Goroutines

Goroutines are, to a rough approximation, extremely lightweight threads, in the spirit of Tony Hoare's Communicating Sequential Processes. Line 16 in the sample above calls the **`say`** function asynchronously, while line 17 calls the **`say`** function synchronously. Goroutines, channels, and **`select`** statements form the core of Go's highly scalable concurrency, one of the strongest selling points of the language. The language also has conventional synchronization objects, but they are rarely needed. The program above outputs:

```
hello
world
hello
world
hello
world
hello
world
hello
```

Deep Dive

```
1  package main
2
3  import "fmt"
4
5  func sum(a []int, c chan int) {
6      sum := 0
7      for _, v := range a {
8          sum += v
9      }
10     c <- sum // send sum to c
11 }
12
13 func main() {
14     a := []int{7, 2, 8, -9, 4, 0}
15
16     c := make(chan int)
17     go sum(a[:len(a)/2], c)
18     go sum(a[len(a)/2:], c)
19     x, y := <-c, <-c // receive from c
20
21     fmt.Println(x, y, x+y)
22 }
```

Channels

Channels in Go provide a mechanism for concurrently executing functions to communicate by sending and receiving values of a specified element type. The value of an uninitialized channel is nil. In line 16, we create a bidirectional channel of integers. We could also make unidirectional sending **`<-c`** and receiving **`c<-`**channels. In lines 17 and 18, we call **`sum`** asynchronously with slices of the first and second half of **`a`**. In line 19, the integer variables **`x`** and **`y`** receive the two sums from the channel. In line 7, the underscore _, the blank identifier, means to ignore the first result value from the **`for.. range`** loop, which is the index. The program output is **`17 -5 12.`**

```
1  package main
2
3  import (
4      "fmt"
5  )
6
7  func fibonacci(n int, c chan int) {
8      x, y := 0, 1
9      for i := 0; i < n; i++ {
10         c <- x
11         x, y = y, x+y
12     }
13     close(c)
14 }
15
16 func main() {
17     c := make(chan int, 10)
18     go fibonacci(cap(c), c)
19     for i := range c {
20         fmt.Println(i)
21     }
22 }
```

Range and close

A sender can **`close`** a channel to indicate that no more values will be sent. Receivers can test whether a channel has been closed by assigning a second parameter to the receive expression. A **`loop`** for **`i := range c`** receives values from the channel repeatedly until it is closed. The **`cap`** of the channel is the capacity, which is the size of the buffer in the channel, set as the optional second argument when you make a channel, as in line 17. Note the compact form of the assignment statements in the **`Fibonacci`** function. The program output is the first 10 values of the Fibonacci series, 0 through 34.

Deep Dive

Select statements

A `select` statement chooses which of a set of possible `send` or `receive` operations will proceed. It looks similar to a `switch` statement but with all the cases referring to communication operations. A `select` blocks until one of its cases can run, then it executes that case. It chooses one at random if multiple are ready.

Here the `main` function calls the `Fibonacci` function with two unbuffered channels, one for results and one for a `quit` signal. The `Fibonacci` function uses a `select` statement to wait on both channels. The anonymous, asynchronous `go` function that starts at line 21 waits to receive values at line 23, then prints them. After 10 values, it sets the `quit` channel, so the `Fibonacci` function knows to stop.

```
1  package main
2
3  import "fmt"
4
5  func fibonacci(c, quit chan int) {
6      x, y := 0, 1
7      for {
8          select {
9          case c <- x:
10             x, y = y, x+y
11         case <-quit:
12             fmt.Println("quit")
13             return
14         }
15     }
16 }
17
18 func main() {
19     c := make(chan int)
20     quit := make(chan int)
21     go func() {
22         for i := 0; i < 10; i++ {
23             fmt.Println(<-c)
24         }
25         quit <- 0
26     }()
27     fibonacci(c, quit)
28 }
```

```
func fanIn(input1, input2 <-chan string) <-chan string {
    c := make(chan string)
    go func() {
        for {
            select {
            case s := <-input1: c <- s
            case s := <-input1: c <- s
            }
        }
    }()
    return c
}
```

Concurrency patterns, example 1

In this example we are using `select` to create a fan-in goroutine that combines two input channels of string, `input1` and `input2`, into one unbuffered output channel, `c`. The `select` statement allows `fanIn` to listen to both input channels simultaneously and relay whichever is ready to the output channel. It doesn't matter that both cases are using the same temporary variable name to hold the string from its respective input channel. The example is from Rob Pike's 2012 talk on Concurrency Patterns in Go.

Concurrency patterns, example 2

This sample implements a parallel search of the Internet, sort of like what Google actually does. To begin with, `replicas ...Search` is a variadic parameter to the function; both Search and Result are types defined elsewhere.

The caller passes N search server functions to the `First` function, which creates a channel `c` for results and defines a function to query the `i`th server and saves it `insearchReplica`. Then First calls searchReplica asynchronously for all N servers, always returning the answer on channel c, and returns the first result to come back from the N servers. The example is from Rob Pike's 2012 talk on Concurrency Patterns in Go.

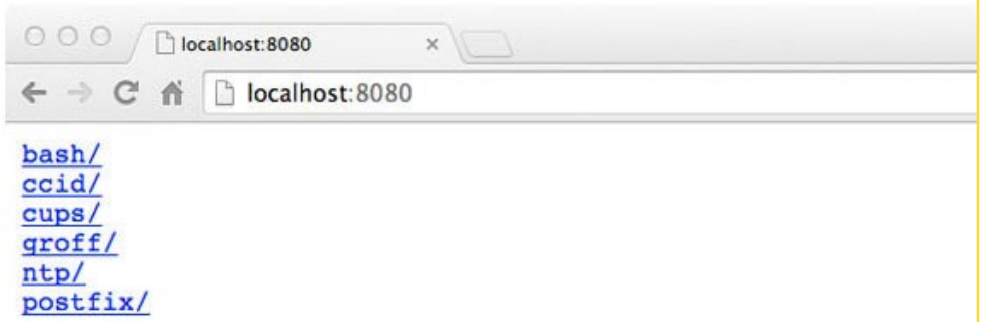
```
func First(query string, replicas ...Search) Result {
        c := make(chan Result)
        searchReplica := func(i int) { c <- replicas[i]
(query) }
        for i := range replicas {
                go searchReplica(i)
        }
        return <-c
}
```

Deep Dive

```
package main

import (
    "log"
    "net/http"
)

func main() {
    // Simple static webserver:
    log.Fatal(http.ListenAndServe(":8080", http.FileServer(http.Dir("/usr/share/doc"))))
}
```



Package http

The Go `net/http` package provides HTTP client and server implementations. This example implements a simple Web server that returns the contents of the directory `/usr/share/doc` to a Web client. The example does not work properly in the Go Playground online environment, but run on a Mac command line, it returns the following to a Web browser asking for http://localhost:8080/:

```
bash/
ccid/
cups/
groff/
ntp/
postfix/
```

Package template

The Go `html/template` package implements data-driven templates for generating HTML output that is safe against code injection. Without all of the escaping added by the `html/template` package, the example could have produced a runnable JavaScript string, `Hello, <script>alert('you have been pwned')</script>!`.

Example

```
import "text/template"
...
t, err := template.New("foo").Parse(`{{define "T"}}Hello, {{.}}!{{end}}`)
err = t.ExecuteTemplate(out, "T", "<script>alert('you have been pwned')</script>")
```

produces

```
Hello, <script>alert('you have been pwned')</script>!
```

but the contextual autoescaping in html/template

```
import "html/template"
...
t, err := template.New("foo").Parse(`{{define "T"}}Hello, {{.}}!{{end}}`)
err = t.ExecuteTemplate(out, "T", "<script>alert('you have been pwned')</script>")
```

produces safe, escaped HTML output

```
Hello, &lt;script&gt;alert(&#39;you have been pwned&#39;)&lt;/script&gt;!
```

Deep Dive

What's Google Go good for?

Google's snappy young language shines for many applications and falls short in others. Here's a quick snapshot, from best to worst

BY SERDAR YEGULALP

After more than five years in the wild, Google's Go language has gone from a curiosity to a wise choice for fast-moving new projects.

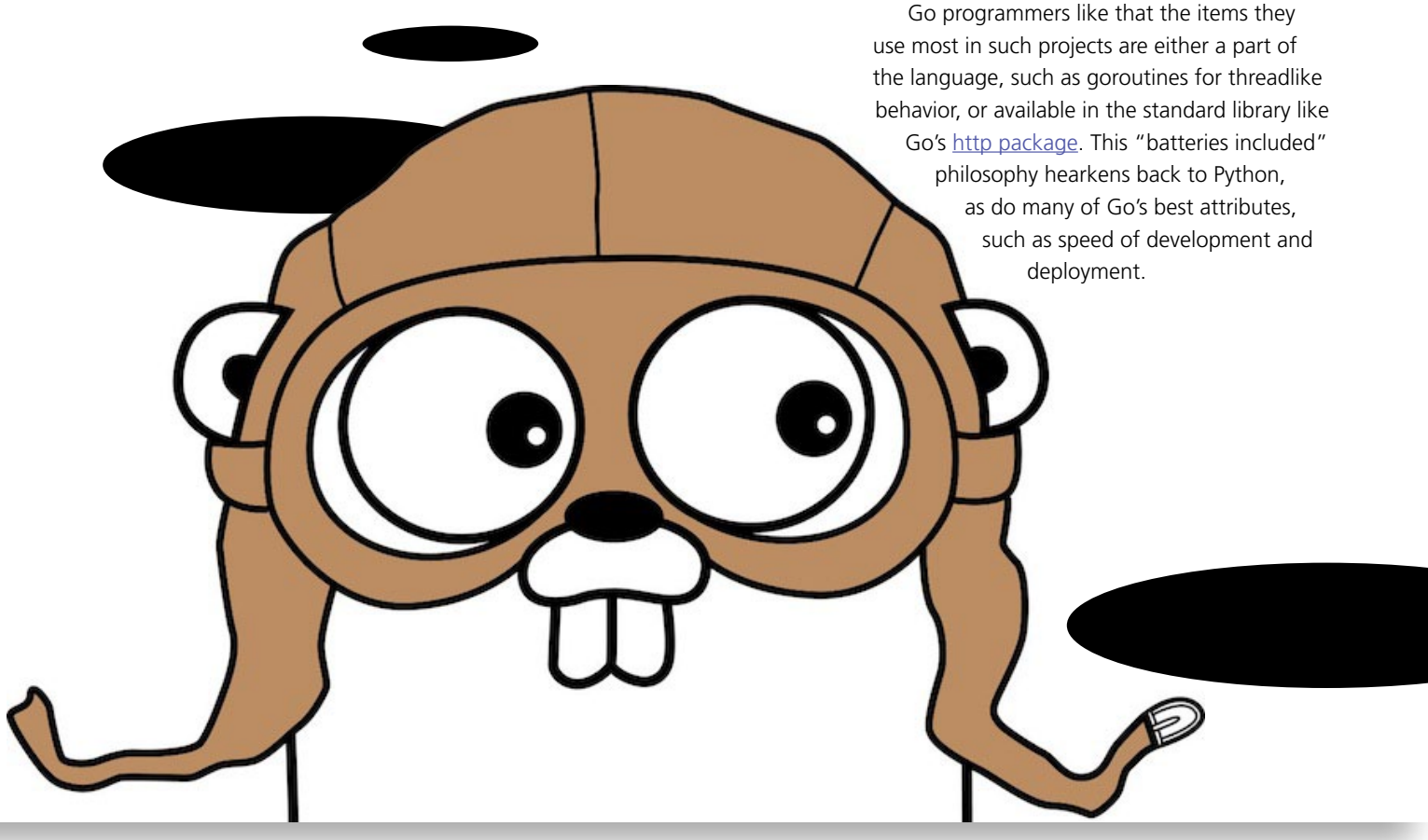
But what kinds of projects are Go best for building — and how is that likely to change as the language evolves through new versions and grows in popularity? Here are the types of applications where Go really excels, where it works well, and where you should opt for another language.

THE REALLY GOOD

Network and Web servers

Network applications live and die by concurrency, and Go's native concurrency features — goroutines and channels, mainly — are well suited for such work. Consequently, many Go projects are for networking, distributed functions, or services: APIs, Web servers, minimal frameworks for Web applications, and the rest.

Go programmers like that the items they use most in such projects are either a part of the language, such as goroutines for threadlike behavior, or available in the standard library like Go's http package. This "batteries included" philosophy hearkens back to Python, as do many of Go's best attributes, such as speed of development and deployment.



Deep Dive

Due to Go's consistent behavior across platforms, it's easy to put out simple command-line apps that run most anywhere.

THE ALSO REALLY GOOD

Stand-alone command-line apps or scripts

Due to Go's consistent behavior across platforms, it's easy to put out simple command-line apps that run most anywhere. It's another echo of Go's similarities to Python, and here Go has a few advantages.

For one, the executables created by Go are precisely that: Stand-alone executables, with no external dependencies unless you specify them. With Python, you must have a copy of the interpreter on the target machine or an interpreter of a particular revision of Python (in the case of some Python scripts).

Another advantage Go has here is speed. The resulting executables run far faster than vanilla Python, or for that matter most any other dynamically executed language, with the possible exception of JavaScript.

Finally, none of the above comes at the cost of being able to talk to the underlying system. Go programs can talk to external C libraries or make native system calls. Docker, for instance, works this way. It interfaces with low-level Linux functions, cgroups, and namespaces, to work its magic.

THE NOT SO GOOD

Desktop or GUI-based apps

Here's where the going gets a little grimmer. Right now, the software for building rich GUIs for Go applications, such as those in desktop applications, is still scattered.

That said, various projects exist — there are bindings for the GTK and GTK3 frameworks, and another intended to provide platform-native UIs, although the latter relies on C bindings and is not written in pure Go. Windows users can try out walk, and some folks at Google are in the process of building a cross-platform GUI library.

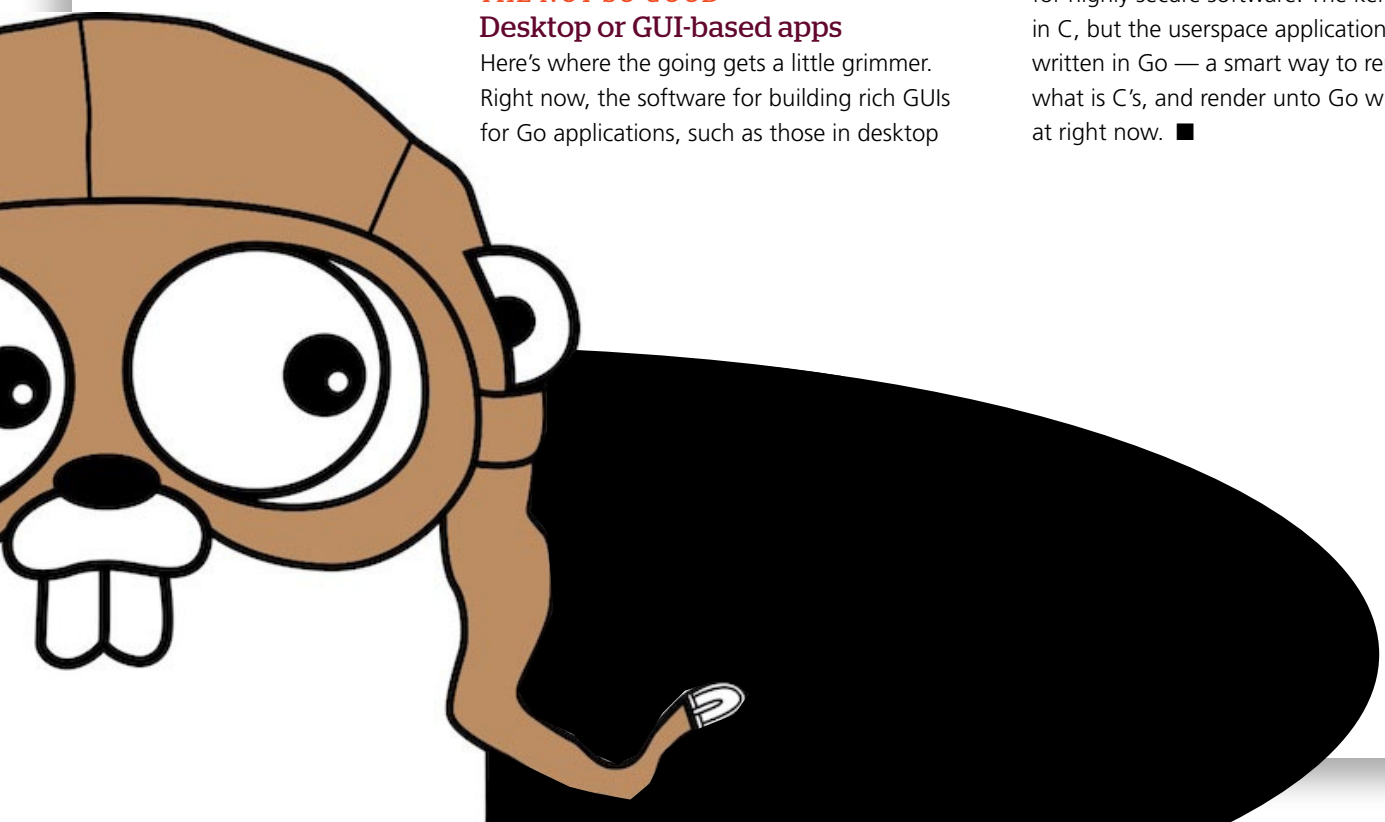
Lacking right now is a sense of any of these being a clear winner or a safe long-term bet. Also, because Go is platform-independent by design, it's unlikely any of these will become a part of the standard package set.

THE LESS GOOD

System-level programming

While Go can talk to native system functions, it's not as good a fit for creating extremely low-level system components, like embedded systems design, kernels, or device drivers. Some of this is a by-product of the language's intentions, since the runtime and the garbage collector for Go applications are dependent on the underlying OS. (Those interested in a cutting-edge language for that kind of work should look into Mozilla's Rust.)

One project currently in the works that partially leverages Go for systems programming is Ethos, an OS intended to serve as a platform for highly secure software. The kernel is written in C, but the userspace applications will be written in Go — a smart way to render unto C what is C's, and render unto Go what Go's best at right now. ■



Deep Dive

10 open source projects prove the power of Go

From application virtualization to self-hosted Git services, Google Go is becoming the tool of choice for forward-thinking projects

BY SERDAR YEGULALP

A mere six years in the wild, Google's Go is a language on the cusp. Lightweight and quick to compile, Go has stirred significant interest from early adopters due to its generous libraries and abstractions that make it easier to program concurrent and distributed (read: cloud) applications.

But the true measure of any language's success is in the projects developers spawn with it. True, Go hasn't made the dent in the developer world that Java, Python, or C/C++ have, but it's starting to accrue a base of meaningful projects highlighting its strengths and benefits.

Here are 10 notable projects written in Go, each of which has already made a splash or is primed to do so. Many of them interrelate, either by including or building on top of each other, each leveraging Go as a language and development system in similar ways.

All the projects featured here are hosted on GitHub, so it's easy for the Go-curious to take a peak at Go code in action.

1 Docker

You'd have a hard time finding a better success story for Go than Docker, at least for now. In a little more than a year, this software-containerization technology has become the poster child for Go's suitability to large-scale, distributed software projects. The Docker team liked Go because it offered a slew of benefits: static compilation with no dependencies, a strong standard library, a full development environment, and the ability to build for multiple architectures with minimal hassle.

Project: Docker
GitHub

2 Kubernetes

If Docker is written in Go, it stands to reason that some of the most significant projects expanding on it are also written in Go. Kubernetes, Google's orchestration project for Docker, is a Go project, along with the Docker networking project Weave and the Shipyard cluster-management system. Since building on top of Docker means facing the same technical challenges as Docker, it only makes sense to continue that project's work using its own idioms and choice of language.

Project: Kubernetes
GitHub

3 Etcd and Fleet

CoreOS uses Docker to turn Linux into a herd of loosely coupled containers, a possible path away from the tangled skein of dependencies that has become the sine qua non of Linux package management. It's no surprise, then, that CoreOS leverages Docker to accomplish this magic — and that two of CoreOS's fundamental services, etcd

Deep Dive

and fleet, are both written in Go. Fleet lets you "treat your CoreOS cluster as if it shared a single init system"; etcd, a distributed key-value store, handles the synchronization of settings between Docker applications and CoreOS instances. Both were written in Go because of Go's "excellent cross-platform support, small binaries and a great community behind it."

Project: Etcd and Fleet

GitHub

4 Deis

If Docker constitutes the bottom strata of a pyramid, with CoreOS as the middle level, Deis could be the capstone. It gives developers a Heroku-esque PaaS built entirely out of open source components and with a lightweight overall construction. Anything that can be put into a Docker container can be deployed, along with existing Heroku build packs if you still happen to have those lying around. The whole package is assembled out of the aforementioned pieces: Docker for the container structure and CoreOS for the OS layer, with Deis itself also written in Go (with some Python for the API server).

Project: Deis

GitHub

5 Flynn

A Y Combinator-supported project, Flynn has many of the same concepts and conceits as Deis, but with subtle differences. Like Deis, Flynn can also work with Heroku buildpacks and Dockerfiles, and it's built using elements of CoreOS, although not directly on top of it. Unlike Deis, though, Flynn is built around a two-tier system architecture, with the lower tier inspired by the Google Omega project, a scalable scheduling system for compute clusters, and the upper tier designed to handle the actual deployment and maintenance. Both layers are written in Go, of course.

Project: Flynn

GitHub

6 Lime

Many of Go's features are a strong complement for distributed server applications, so it's little surprise the major Go projects are server- and cloud-oriented. But desktop applications written in Go are starting to turn up, too, and Lime is a good one. It's a re-creation of the widely used and loved Sublime Text editor — which, for all of its positive attributes, is not open source, to the chagrin of many who work with and depend on it. Lime is still heavily prototypical, so it shouldn't be used for production work, but with almost 10,000 stars on GitHub as of this writing and almost 800 forks, it's making fast progress.

Project: Lime

GitHub

7 Revel

One sign of real-world language traction is when developers build commonly used tools with it, such as a Web framework. Enter Revel, a "high-productivity, full-stack Web framework" written in Go. It's outfitted with all the features you'd expect: routing, caching, parameter parsing, templating. It also sports a few features you might not expect, such as internationalization, a testing framework, and a modular design that lets developers add their own request processing systems (aka filters) to Revel without having to rip everything apart. While Revel is technically production-ready, the Revel team is still holding off on applying the all-important 1.0 moniker before a few more pieces are snapped into place.

Project: Revel

GitHub

Deep Dive

8 InfluxDB

InfluxDB is "distributed time series database with no external dependencies." The term "time series" means InfluxDB is mainly concerned with taking in metrics or events, and allowing them to be analyzed in real time. "No external dependencies" means you need no other software to use InfluxDB; it's entirely self-contained. Data can be written to or read from the database by way of REST calls that submit JSON, and queries can be made via a simple SQL language that even allows regexes. InfluxDB is highly elastic and horizontally scalable, and it's likely Go was chosen as the language to make those features possible and easy.

Project: InfluxDB
GitHub

9 Syncthing

Syncthing could be either a desktop or server project, given that it's theoretically useful in both contexts, but it's tempting to call it the former for the sake of noting that Go is also useful for desktop applications.

Syncthing implements a protocol for synchronizing folders between multiple devices, à la the BitTorrent Sync project. The developers' rationale for using Go revolves around many of the language's most praised features: simplicity, speed, concurrency, ease of deployment across architectures, and so on. Although nothing is stopping anyone else from building Syncthing clients in Python or C, the Go version is here and is clearly aimed at cross-platform use.

Project: Syncthing
GitHub

10 Gogs (Go Git Service)

Ever wanted to host your own Git service, like GitHub but on your own hardware and under your own control? Gogs (Go Git Service) was written to fill that need and can do it across all major OS platforms (Linux, Windows, and Mac OS X). All that's needed is a copy of Git, a database of some kind — MySQL, PostgreSQL, and SQLite3 (included) are all supported — and the binaries.

Project: Gogs (Go Git Service)
GitHub