THE PAST, PRESENT, AND FUTURE
OF GO 2
QCON SHANGHAI 2018
Today

Where Go came from?

How Go has evolved since it was launched?

What’s happening in Go 2?
THE PAST

2007–2009
Why Go?

Why is there a language called Go?

We have C++, Java, C#, Python, Ruby, PHP, and Javascript

Why did Rob Pike, Ken Thompson, and Robert Griesemer decide to write a new language?
The Go programming language, 2009
Go at Google: Language Design in the Service of Software Engineering

1. Abstract

(This is a modified version of the keynote talk given by Rob Pike at the SPLASH 2012 conference in Tucson, Arizona, on October 25, 2012.)

The Go programming language was conceived in late 2007 as an answer to some of the problems we were seeing developing software infrastructure at Google. The computing landscape today is almost unrelated to the environment in which the languages being used, mostly C++, Java, and Python, had been created. The problems introduced by multicore processors, networked systems, massive computation clusters, and the web programming model were being worked around rather than addressed head-on. Moreover, the scale has changed: today’s server programs comprise tens of millions of lines of code, are worked on by hundreds or even thousands of programmers, and are updated literally every day. To make matters worse, build times, even on large compilation clusters, have stretched to many minutes, even hours.

Go was designed and developed to make working in this environment more productive. Besides its better-known aspects such as built-in concurrency and garbage collection, Go’s design considerations include rigorous dependency management, the adaptability of software architecture as systems grow, and robustness across the boundaries between components.

This article explains how these issues were addressed while building an efficient, compiled programming language that was easy to read, write, and maintain.
A LANGUAGE FOR DEVELOPER PRODUCTIVITY

Together these presentations provide a rationale for a new language, originally designed for Google's software development needs.

As it turns out—because we all need software—Go has become a pretty good fit for anyone writing large scale server software.

Because, at its core, the goal of Go is to improve developer productivity.
THE DIFFERENCE BETWEEN PROGRAMMING AND SOFTWARE ENGINEERING

“Software engineering is what happens to programming when you add time and other programmers.”

—Russ Cox
The difference between software programming and software engineering is not the size of the program, but how long the program will live for.

Sitting down and writing a script or a throw away program for a single computation is software programming.

That’s totally fine, sometimes that is all the problem calls for.
THE DIFFERENCE BETWEEN PROGRAMMING AND SOFTWARE ENGINEERING

On the other hand, Software engineering is a more deliberate, considered, act.

It requires a broader view of the software development lifecycle than just focusing on lines of code, syntax, and algorithms.

When Go launched it was with the explicit intent to improve the life of the software engineer.
The PRESENT

2009–2018
THE PLATFORMS

When Go was open sourced on the 11th of November 2009 it supported Linux, Mac OS X, on 386, amd64, and if you were running Linux, ARMv5 and v6.

By the time Go 1.0 was launched in March of 2012 we added support for Windows, FreeBSD, OpenBSD
The Platforms

In Go 1.3 we added support for FreeBSD, DragonflyBSD, OpenBSD, and NetBSD, Plan 9 on 386 and Native Client (NaCl), and Solaris on amd64

Go 1.4 added support for cross compiling to Android, NaCl on ARM, and Plan 9 amd64

Go 1.5 added support for arm64 on Linux and OS X.

Go 1.6 added support for 64bit MIPS on Linux, and Android on 386

Go 1.7 added support for IBM System/z and 64 bit PowerPC

Go 1.8 added support for 32 bit MIPS

Go 1.11 added support for web assembly and plans are in the works for a RISC-V port
The performance

Go 1.0 vs Go 1.11

`linux/amd64 (Lenovo x220 Core i5-2520M)`
The companies

Atlassian, Heptio, Digital Ocean, Netflix, Pulimi, Twitch, Google, Microsoft, Reddit, Cloudflare, MongoDB, InfluxDB, Datadog, bookings.com, Rakuten, GitHub, GitLab, Freelancer, Fastly, Netlify, Pivotal, Couchbase, Lyft, Monzo, Uber, Source{d}, srcgraph, …
THE COMPANIES

QINIU
Baidu
PingCAP
Alibaba Cloud
Grab
Bilibili

Didi
Tantan
The projects

Vitess, Docker, Traefic, Kubernetes, Istio, GitLab, Vault, Consol, Terraform, CockroachDB, CloudFoundry, Gobot, Beego, …
The books

HTTPS://GITHUB.COM/GOLANG/GO/WIKI/BOOKS
The books

https://github.com/golang/go/wiki/books
The community
Go meetups in China
GODOC.ORG, 800,000+ PACKAGES

Search for Go Packages

Search for package by import path or keyword.

GoDoc hosts documentation for Go packages on Bitbucket, GitHub, Google Project Hosting and Launchpad. Read the About Page for information about adding packages to GoDoc and more.

Popular Packages
- github.com/Shopify/sarama
- github.com/aws/aws-sdk-go/aws
- github.com/dgrijalva/jwt-go
- github.com/gin-gonic/gin
- github.com/go-redis/redis
- github.com/golang/protobuf/protobuf

More Packages
- Go Standard Packages
- Go Sub-repository Packages
- Projects @ go-wiki
- Most stars, most forks, recently updated on GitHub
The conferences

GopherCon Denver

GopherCon EU

DOTGO Paris

GopherCon Brasil

GopherCon Singapore

GopherCon UK
Go conferences in China

GopherChina 2015
第一届 GopherChina 大会

GopherChina 2016
第二届 GopherChina 大会

GopherChina 2017
第三届 GopherChina 大会

GopherChina 2018
第四届 GopherChina 大会
Next year, GopherChina Beijing
THE GOPHER

The Go gopher was designed by Renee French.

The design is licensed under the Creative Commons 3.0 Attributions license.
THE GOPHER

The Go gopher was designed by Renee French.

The design is licensed under the Creative Commons 3.0 Attributions license.
GO TEAM.
This is fine
The growth of the language

“My best estimate is now between 0.8 and 1.6 million. It seems to me likely that we've crossed a million Go developers.”
—Russ Cox, July 2018
The future
2018–
Gophercon 2017, Russ Cox announced it was time to start talking about Go 2

https://blog.golang.org/toward-go2

Go 2 would not be an opportunity to redesign the language from scratch.

Instead, Go 2 would be an evolution of Go 1, designed to address pain points Go developers worldwide have felt for a decade.
“Our goal for Go 2 is to fix the most significant ways Go fails to scale.”

—Russ Cox, GopherCon 2017
How shall we discover where Go failed to scale?

In his presentation at Gophercon 2017 Russ described the methodology for how the large issues which caused Go to fail at scale will be identified.

Specifically Russ called on the users of Go to write experience reports; gists, blog posts, issues, that demonstrated clearly the issues that developers were having using Go for increasingly larger and larger projects.

https://github.com/golang/go/wiki/ExperienceReports

Now it’s a year later, what did the Go team discover?
Towards Go 2
Towards Go 2

Top three pain points for Go developers:
Towards Go 2

Top three pain points for Go developers:

• Dependency management – modules
Towards Go 2

Top three pain points for Go developers:

• Dependency management – modules
• Error handling – check, handle, and error values
Towards Go 2

Top three pain points for Go developers:

• Dependency management – modules
• Error handling – check, handle, and error values
• Generics
DEPENDENCY MANAGEMENT

Go modules
The first improvement is the addition of a new concept to the Go tool, a module.

A module is a collection of packages.

Just as we have .go source files grouped into a package, so too can a collection of packages with shared prefix be considered a module.

Now, this probably looks pretty close to a concept that you already know, a git repository. But there is an important difference, modules have an explicit understanding of versions.
Why do we need Go modules?

Prior to Go modules, `go get` only knew how to fetch whatever revision happened to be current in your repository at the time.

If you already had a copy of a package in your `$GOPATH` then `go get` would skip it, so you might end up building against a really old version.

If you used the `go get -u` flag to force it to download a fresh copy, you might find that you now had a much newer version of a package than the author.
Go get does not provide reproducible builds

Put simply, go get doesn’t guarantee reproducible builds. We’ve had many people propose solutions, tools like:

• godep
• gopkg.in
• govendor
• gb

Promoted the idea of a vendor/ directory, a self-contained gopath that could be checked in with the code so that your program had a copy of each of the dependencies it needed.
In 2016 Peter Bourgon formed a working group to focus on solving the dependency management problem and called on the go team to join him in this effort.

From that working group grew a tool we know as **dep**.

dep drew much of its inspiration from the authors experience with their previous tools **glide**.

dep encouraged the use of **semver**, **semantic versioning**, using tags on your git repos, to provide tools like dep with a way of managing the contents of your vendor/ directory.
The Go team introduce modules

In early 2018 the Go team proposed their own tool, at the time given the working title vgo, now known as go modules.

Go modules are integrated into the Go tool. The notion of modules is baked in as a first class citizen.

This makes it possible for Go developers to build their code anywhere they want.

Go modules don’t require a vendor/directory, and if you use modules you no longer need to use GOPATH to hold all your Go source code.
Go modules live demo
You can use Go modules today

Go 1.11, which shipped in August, includes full support for modules.

It's opt-in at the moment, because we realise there is a large change, not just for package authors but for the ecosystem of tool authors.

Experiment with converting your projects to use `go.mod` and please give the Go team feedback via Github.
ERROR HANDLING

CHECK, HANDLE, AND ERROR VALUES
**Error Handling in Go**

Unlike Java, Ruby, Python, or C#, Go does not use exceptions for control flow.

Instead Go's error handling takes advantage of the language’s native support for multiple return values.

```go
cfunc Open(path string) (*File, error)
```

By convention, if a function returns an error value, then the caller should check if that error value to see if the operation succeeded or failed.
Error Handling in Go

By convention, if a function returns an error value, then the caller should check the error value to see if the operation succeeded or failed.

```go
f, err := os.Open("/etc/passwd")
if err != nil {
    return err
}
```

Go developers believe that by being forced to think about failure case before the success case, leads to more robust programs.

However, this form of error checking means it can feel repetitive to write this error checking code by hand.
func CopyFile(src, dst string) error {
    r, err := os.Open(src)
    if err != nil {
        return err
    }
    defer r.Close()

    w, err := os.Create(dst)
    if err != nil {
        return err
    }
    defer w.Close()

    if _, err := io.Copy(w, r); err != nil {
        return err
    }
    if err := w.Close(); err != nil {
        return err
    }
    return nil
}
func CopyFile(src, dst string) error {
    r, err := os.Open(src)
    if err != nil {
        return err
    }
    defer r.Close()

    w, err := os.Create(dst)
    if err != nil {
        return err
    }
    defer w.Close()

    if _, err := io.Copy(w, r); err != nil {
        return err
    }
    if err := w.Close(); err != nil {
        return err
    }
    return nil
}
func CopyFile(src, dst string) error {
    r, err := os.Open(src)
    if err != nil {
        return err
    }
    defer r.Close()

    w, err := os.Create(dst)
    if err != nil {
        return err
    }
    defer w.Close()

    if _, err := io.Copy(w, r); err != nil {
        return err
    }
    if err := w.Close(); err != nil {
        return err
    }
    return nil
}
func CopyFile(src, dst string) error {
    r, err := os.Open(src)
    if err != nil {
        return err
    }
    defer r.Close()
    ...
}
func CopyFile(src, dst string) error {
    r, err := os.Open(src)
    if err != nil {
        return err
    }
    defer r.Close()
    ...
}

func check(rc io.ReadCloser, err error) io.ReadCloser {
    if err == nil {
        if err != nil {
            return rc
        }
        panic(err)
    }
    return rc
}
func CopyFile(src, dst string) error {
    r := check(os.Open(src))
    defer r.Close()
    ...
}

func check(rc io.ReadCloser, err error) io.ReadCloser {
    if err == nil {
        return rc
    }
    panic(err)
}
func CopyFile(src, dst string) error {
    r := check(os.Open(src))
    defer r.Close()
    ...
}

func check(rc io.ReadCloser, err error) io.ReadCloser {
    if err == nil {
        return rc
    }
    panic(err)
}

Two values go into check
func CopyFile(src, dst string) error {
    r := check(os.Open(src))
    defer r.Close()
    ...
}

func check(rc io.ReadCloser, err error) io.ReadCloser {
    if err == nil {
        return rc
    }
    return rc
    panic(err)
}

Two values go into check
One value comes out
func CopyFile(src, dst string) error {
    r := check(os.Open(src))
    defer r.Close()
    ...
}

func check(rc io.ReadCloser, err error) io.ReadCloser {
    if err == nil {
        return rc
    }
    panic(err)
}

check must return a value
func CopyFile(src, dst string) error {
    r := check(os.Open(src))
    defer r.Close()
    ...
}

func check(rc io.ReadCloser, err error) io.ReadCloser {
    if err == nil {
        return rc
    }
    panic(err)
}

check must return a value

Crashes the whole program 😞
```go
func CopyFile(src, dst string) error {
    r := check(os.Open(src))
    defer r.Close()
    ...
}

func check(rc io.ReadCloser, err error) io.ReadCloser {
    if err == nil {
        return rc
    }
    return err
}
```
func CopyFile(src, dst string) error {
    r := check(os.Open(src))
    defer r.Close()
    ...
}

func check(rc io.ReadCloser, err error) io.ReadCloser {
    if err == nil {
        return rc
    }
    return err
}

We want to return the error here
Go programmers cannot write the own check functions today as we cannot return to the caller of the caller of check.

So the Go team are adding a new `check` keyword to the language which does exactly this.
func CopyFile(src, dst string) error {
    r, err := os.Open(src)
    if err != nil {
        return err
    }
    defer r.Close()

    w, err := os.Create(dst)
    if err != nil {
        return err
    }
    defer w.Close()

    if _, err := io.Copy(w, r); err != nil {
        return err
    }
    if err := w.Close(); err != nil {
        return err
    }
    return nil
}
func CopyFile(src, dst string) error {
    r := check os.Open(src)
    defer r.Close()

    w := check os.Create(dst)
    defer w.Close()

    check io.Copy(w, r)
    check w.Close()
    return nil
}
func CopyFile(src, dst string) error {
    r, err := os.Open(src)
    if err != nil {
        return err
    }
    defer r.Close()

    w, err := os.Create(dst)
    if err != nil {
        return err
    }
    defer w.Close()

    if _, err := io.Copy(w, r); err != nil {
        return err
    }
    if err := w.Close(); err != nil {
        return err
    }
    return nil
}
func CopyFile(src, dst string) error {
    r, err := os.Open(src)
    if err != nil {
        return err
    }
    defer r.Close()
    
    w, err := os.Create(dst)
    if err != nil {
        return err
    }
    defer w.Close()
    
    if _, err := io.Copy(w, r); err != nil {
        return err
    }
    if err := w.Close(); err != nil {
        return err
    }
    return nil
}
func CopyFile(src, dst string) error {
    r, err := os.Open(src)
    if err != nil {
        return err
    }
    defer r.Close()

    w, err := os.Create(dst)
    if err != nil {
        return err
    }
    defer w.Close()

    if _, err := io.Copy(w, r); err != nil {
        return err
    }
    if err := w.Close(); err != nil {
        return err
    }
    return nil
}
func CopyFile(src, dst string) error {
    r, err := os.Open(src)
    if err != nil {
        return err
    }
    defer r.Close()

    w, err := os.Create(dst)
    if err != nil {
        return err
    }
    defer w.Close()

    if _, err := io.Copy(w, r); err != nil {
        return err
    }
    if err := w.Close(); err != nil {
        return err
    }
    return nil
}
func CopyFile(src, dst string) error {
    r, err := os.Open(src)
    if err != nil {
        return fmt.Errorf("copy %s %s: %v", src, dst, err)
    }
    defer r.Close()

    w, err := os.Create(dst)
    if err != nil {
        return fmt.Errorf("copy %s %s: %v", src, dst, err)
    }

    if _, err := io.Copy(w, r); err != nil {
        w.Close()
        os.Remove(dst)
        return fmt.Errorf("copy %s %s: %v", src, dst, err)
    }

    if err := w.Close(); err != nil {
        os.Remove(dst)
        return fmt.Errorf("copy %s %s: %v", src, dst, err)
    }
    return nil
}
func CopyFile(src, dst string) error {
    r, err := os.Open(src)
    if err != nil {
        return fmt.Errorf("copy %s %s: %v", src, dst, err)
    }
    defer r.Close()

    w, err := os.Create(dst)
    if err != nil {
        return fmt.Errorf("copy %s %s: %v", src, dst, err)
    }
    if _, err := io.Copy(w, r); err != nil {
        w.Close()
        os.Remove(dst)
        return fmt.Errorf("copy %s %s: %v", src, dst, err)
    }
    if err := w.Close(); err != nil {
        os.Remove(dst)
        return fmt.Errorf("copy %s %s: %v", src, dst, err)
    }
    return nil
}
func CopyFile(src, dst string) error {
    r, err := os.Open(src)
    if err != nil {
        return fmt.Errorf("copy %s %s: %v", src, dst, err)
    }
    defer r.Close()

    w, err := os.Create(dst)
    if err != nil {
        return fmt.Errorf("copy %s %s: %v", src, dst, err)
    }

    if _, err := io.Copy(w, r); err != nil {
        w.Close()
        os.Remove(dst)
        return fmt.Errorf("copy %s %s: %v", src, dst, err)
    }

    if err := w.Close(); err != nil {
        os.Remove(dst)
        return fmt.Errorf("copy %s %s: %v", src, dst, err)
    }
    return nil
}
func CopyFile(src, dst string) error {
    r, err := os.Open(src)
    if err != nil {
        return fmt.Errorf("copy %s %s: %v", src, dst, err)
    }
    defer r.Close()

    w, err := os.Create(dst)
    if err != nil {
        return fmt.Errorf("copy %s %s: %v", src, dst, err)
    }
    if _, err := io.Copy(w, r); err != nil {
        w.Close()
        os.Remove(dst)
        return fmt.Errorf("copy %s %s: %v", src, dst, err)
    }
    if err := w.Close(); err != nil {
        os.Remove(dst)
        return fmt.Errorf("copy %s %s: %v", src, dst, err)
    }
    return nil
}
The solution the Go team are proposing a new statement called `handle`.

You can think of `handle` as being similar to `defer`. Control will transfer to the handle block if `err != nil`.

Just like `defer`, handle functions can appear anywhere during the function. If a `check` fails, it transfers control to the innermost handler, which transfers control to the next handler above it, and so on, until a handler executes a return statement.
func CopyFile(src, dst string) error {
    handle err {
        return fmt.Errorf("copy %s %s: %v", src, dst, err)
    }

    r := check os.Open(src)
    defer r.Close()

    w := check os.Create(dst)
    handle err {
        w.Close()
        os.Remove(dst) // (only if a check fails)
    }

    check io.Copy(w, r)
    check w.Close()
    return nil
}
func CopyFile(src, dst string) error {
    handle err {
        return fmt.Errorf("copy %s %s: %v", src, dst, err)
    }

    r := check os.Open(src)
    defer r.Close()

    w := check os.Create(dst)
    handle err {
        w.Close()
        os.Remove(dst) // (only if a check fails)
    }

    check io.Copy(w, r)
    check w.Close()
    return nil
}
func CopyFile(src, dst string) error {
    handle err {
        return fmt.Errorf("copy %s %s: %v", src, dst, err)
    }

    r := check os.Open(src)
    defer r.Close()

    w := check os.Create(dst)
    handle err {
        w.Close()
        os.Remove(dst) // (only if a check fails)
    }

    check io.Copy(w, r)
    check w.Close()
    return nil
}
func CopyFile(src, dst string) error {
    handle err {
        return fmt.Errorf("copy %s %s: %v", src, dst, err)
    }

    r := check os.Open(src)
    defer r.Close()

    w := check os.Create(dst)
    handle err {
        w.Close()
        os.Remove(dst) // (only if a check fails)
    }

    check io.Copy(w, r)
    check w.Close()
    return nil
}
func CopyFile(src, dst string) error {
    handle err {
        return fmt.Errorf("copy %s %s: %v", src, dst, err)
    }

    r := check os.Open(src)
    defer r.Close()

    w := check os.Create(dst)
    handle err {
        w.Close()
        os.Remove(dst) // (only if a check fails)
    }

    check io.Copy(w, r)
    check w.Close()

    return nil
}
func CopyFile(src, dst string) error {
    handle err {
        return fmt.Errorf("copy %s %s: %v", src, dst, err)
    }
    r := check os.Open(src)
    defer r.Close()
    w := check os.Create(dst)
    handle err {
        w.Close()
        os.Remove(dst) // (only if a check fails)
    }
    check io.Copy(w, r)
    check w.Close()
    return nil
}
func CopyFile(src, dst string) error {
    handle err {
        return fmt.Errorf("copy %s %s: %v", src, dst, err)
    }
    r := check os.Open(src)
    defer r.Close()
    w := check os.Create(dst)
    handle err {
        w.Close()
        os.Remove(dst) // (only if a check fails)
    }
    check io.Copy(w, r)
    check w.Close()
    return nil
}
INSPECTING ERRORS

Go programmers have two main techniques for providing information in errors. If the intent is only to describe a unique condition with no additional data, a variable of type error suffices

```go
var ErrUnexpectedEOF = errors.New("unexpected EOF")
```

Programs can act on such sentinel errors by a simple comparison:

```go
if err == io.ErrUnexpectedEOF { ... }
```
INSPECTING ERRORS

To provide more information, the programmer can define a new type that implements the error interface. For example, `os.PathError` is a struct that includes a pathname.

Programs can extract information from these errors by using type assertions:

```go
if pe, ok := err.(*os.PathError); ok { ... pe.Path ... }
```
INSPECTING ERRORS

We could instead create a new type to hold additional details along with the underlying error

```go
if err != nil {
    return &WriteError{Database: "users", Err: err}
}
```

However, this could break the caller as type of the error has changed to our WriteError type.

Either way, wrapping breaks both equality checks and type assertions looking for the original error. This discourages wrapping, leading to less useful errors.
The first part of the design is to add a standard, optional interface implemented by errors that wrap other errors:

```go
package errors

// A Wrapper is an error implementation
// wrapping context around another error.
type Wrapper interface {
    // Unwrap returns the next error in the error chain.
    // If there is no next error, Unwrap returns nil.
    Unwrap() error
}
```

Programs can inspect the chain of wrapped errors by using a type assertion to check for the Unwrap method and then calling it.
Wrapping errors breaks the two common patterns for acting on errors, equality comparison and type assertion.

To reestablish those operations, the second part of the design adds two new functions: `errors.Is`, which searches the error chain for a specific error value.

```go
// instead of err == io.ErrUnexpectedEOF
if errors.Is(err, io.ErrUnexpectedEOF) { ... }
```

The `errors.Is` function is used instead of a direct equality check.
The second helper is `errors.As`, which searches the chain for a specific type of error.

The `errors.As` function is used instead of a type assertion:

```go
// instead of pe, ok := err.(*os.PathError)
if pe, ok := errors.As(*os.PathError)(err); ok {
    ... pe.Path ...
}
```
Error handling

check and handle for cleaning up error handling boilerplate

ersors.Is and errors.As for error inspection
GENERICS
Why do Go programmers want generics?
Why do Go programmers want generics?

```go
func Max(a, b int) int {
    if a > b {
        return a
    }
    return b
}
```
Why do Go programmers want generics?

```go
func Max(a, b int) int {
    if a > b {
        return a
    }
    return b
}

Only works with ints
```
package math

import "github.com/pkg/math"

Package math provides helper functions for mathematical operations over all integer Go types.

Almost all files in this package are automatically generated.

To regenerate this package

make -B

This package relies on github.com/davecheney/godoc2md.

Index

func EqualBigInt(a, b *big.Int) bool
func Eqv (T, U Op, V bool) bool
func MinInt32

func MinInt32(a, b int32) int32

MinInt32 returns the smaller of two int32s.

func MinInt32N

func MinInt32N(v ...int32) int32

MinInt32N returns the smallest int32 in the set provided. If no values are provided, MinInt32 returns 0.

func MinInt64

func MinInt64(a, b int64) int64

MinInt64 returns the smaller of two int64s.

func MinInt64N

func MinInt64N(v ...int64) int64
GENERIC MAX IMPLEMENTATION
func Max(a, b T) T {
    if a > b {
        return a
    }
    return b
}
func Max(a, b T) T {
    if a > b {
        return a
    }
    return b
}

func main() {
    var A, B uint8 = 50, 90
    result := Max(A, B)
    fmt.Println(result)
}
func Max(a, b uint8) uint8 {
    if a > b {
        return a
    }
    return b
}

func main() {
    var A, B uint8 = 50, 90
    result := Max(A, B)
    fmt.Println(result) // 90
THE PROBLEM WITH TEMPLATE SUBSTITUTION

func Max(a, b T) T {
    if a > b {
        return a
    }
    return b
}

func main() {
    var A, B = "Hello", "QCon"
    result := Max(A, B)
    fmt.Println(result)
}
The problem with template substitution

```go
func Max(a, b string) string {
    if a > b {
        return a
    }
    return b
}

func main() {
    var A, B = "Hello", "QCon"
    result := Max(A, B)
    fmt.Println(result) // "QCon"
}
```
The problem with template substitution

```go
func Max(a, b T) T {
    if a > b {
        return a
    }
    return b
}

func main() {
    var A, B = []byte("Hello"), []byte("QCon")
    result := Max(A, B)
}
```
func Max(a, b []byte) []byte {
    if a > b {
        return a
    }
    return b
}

func main() {
    var A, B = []byte("Hello"), []byte("QCon")
    result := Max(A, B)
}
func Max(a, b []byte) []byte {
    if a > b {
        return a
    }
    return b
}

func main() {
    var A, B = []byte("Hello"), []byte("QCon")
    result := Max(A, B)
}
The problem with template substitution

```go
func Max(a, b []byte) []byte {
    if a > b {
        return a
    }
    return b
}

func main() {
    var A, B = []byte("Hello"), []byte("QCon")
    result := Max(A, B)
}
```

Compiler complains here

But the bug is actually here
The problem with template substitution

```go
func Max(a, b T) T {
    if a > b {
        return a
    }
    return b
}

func main() {
    var A, B float64 = 3.1417, math.NaN()
    result := Max(A, B)
    fmt.Println(result)
}
```
THE PROBLEM WITH TEMPLATE SUBSTITUTION

```go
func Max(a, b float64) float64 {
    if a > b {
        return a
    }
    return b
}

func main() {
    var A, B float64 = 3.1417, math.NaN()
    result := Max(A, B)
    fmt.Println(result) // ???
}
```
The problem with template substitution

We need a way of applying a constraint on which types can be substituted for T.

In other languages, like Java, this is called a type bound.

```java
public static <T extends Number> T max(T a, T b) { ... }
```

Go doesn't have type inheritance, and we don't want to add it, we see not having inheritance as a feature, not a bug.
The suggestion the Go team have come up with is called a contract.

A contract is a way to write down a list of requirements for a type implementing T

```go
contract comparable(t T) {
    t > t
    t << 1
}
```
COMPARABLE CONTRACT

contract comparable(t T) {
    t > t
    t <<= 1
}

**COMPARABLE CONTRACT**

```go
c contract comparable(t T) {
    t > t  // T must be a type with a greater than operator. This excludes slices, maps, channels, or structs.
    t << 1
}
```
**Comparable Contract**

```go
contract comparable(t T) {
    t > t
    t <<= 1
}
```

- **T** must be a type with a greater-than operator. This excludes slices, maps, channels, or structs.
- **T** must be a type that can be shifted, this excludes float64.
HOW DO WE USE A CONTRACT?

contract comparable(t T) {
    t > t
    t <<= 1
}

func Max(type T comparable)(a, b T) T {
    if a > b {
        return a
    }
    return b
}
HOW DO WE USE A CONTRACT?

contract comparable(t T) {
    t > t
    t <<= 1
}

func Max(type T comparable)(a, b T) T {
    if a > b {
        return a
    }
    return b
}
HOW DO WE USE A CONTRACT?

contract comparable(t T) {
    t > t
    t <<= 1
}

func Max(type T comparable)(a, b T) T {
    if a > b {
        return a
    }
    return b
}
How do we use a contract?

```go
ccontract comparable(t T) {
    t > t
    t <<= 1
}

cfunc Max(type T comparable)(a, b T) T {
    if a > b {
        return a
    }
    return b
}
```

- **Type parameter**
- **Formal parameters**
- **Return values**
THE GENERIC DILEMMA

The generics debate in Go is not new. Years ago Russ Cox wrote a short post called the Generics Dilemma, on the three approaches to adding generics to any language

1. Don't do it. This is the approach C tool, and, until now, the approach Go chose.

2. Compile-time specialisation or template expansion. This is the C++ approach. It generates a lot of code, much of it redundant, and needs a good linker to eliminate duplicate copies. This slows down compilation

3. Box everything and insert casts at runtime. This is the Java approach.
The important thing to recognise in this proposal is the syntax I shown in the previous slides does not dictate how the feature will be implemented.

Unlike the C++ implementation which is explicitly defined to rely on template text substitution, or the java solution which requires boxing every parameter into an object, this proposal does not specify how the compiler should implement this feature.

The Go compiler may choose to specialise a generic function at compile-time or use run-time boxing and casting. The decision becomes purely a compiler optimization, not one of semantic significance.
Would you like to know more?

If you'd like to know more, read the design documents, and importantly contribute your feedback on these proposals at this page

https://blog.golang.org/go2draft

Go modules implementation is much further along, as I mentioned, its available to try in Go 1.11 today, so feedback and experience reports are best directed to the issue tracker.
There will be no Go 2

And that’s OK
Go developers recognise that over the last 9 years the value Go has bought to you is not what has been added to go, but what has not changed.

The value in Go is the huge base of software written in the language that was defined in 2012, and which we’ve been using productively since then.
The value of Go is in the commitment to backwards compatibility that the Go 1 contract bought us for the last nine years.
Adoption

The value in language is all of you in this room today. Because ultimately a programming language is only successful if it has a large user base of people who are happy to continue to use it.
Towards Go 2

In a few months it will be December, then January 2019.

2019 is a whole new year, distinct and separate from the previous 365 days of 2018.

Yet, except for changing the year, January 1st 2019 will be in every other respect just a continuation of December 31st, 2018.
Towards Go 2

For all of the Go users today, Go 2 is not a single release we're working towards.

Just like one day following the next, the progress of small, frequent, releases will continue, adding these features that I discussed today,—and maybe a few other small tweaks—until one day we decide to call it Go 2.
THANK YOU!

ENJOY QCON SHANGHAI 2018