Go Tutorial

Ian Lance Taylor

GCC Summit, October 27, 2010

Go

- Go is a new experimental general purpose programming language. Main developers are:
  - Russ Cox
  - Robert Griesemer
  - Rob Pike
  - Ian Lance Taylor
  - Ken Thompson
- It was released as free software in November 2009.
- `http://golang.org/`
- This talk is about the language.

Talk

I'm not going to cover the basics of the language. The language is not large and you can all learn it easily.

1. Why introduce a new language?
2. Discuss some of the more unusual aspects of the language.

What problems does Go solve?

- Types get in the way too much.
- Multi-core is an opportunity.
- Concurrency is very hard to get right.
- Computers are fast but building programs is slow.
- Programming is less fun.

What problems does Go solve?

- Types get in the way too much.
- Multi-core is an opportunity.
- Concurrency is very hard to get right.
- Computers are fast but building programs is slow.
- Programming is less fun.

Go is an attempt to solve these problems.

What problems does Go solve?

- Types get in the way too much.
  - Go has a lightweight typing system, and many types need not be written explicitly.
- Multi-core is an opportunity.
- Concurrency is very hard to get right.
- Computers are fast but building programs is slow.
- Programming is less fun.

Go is an attempt to solve these problems.

What problems does Go solve?

- Types get in the way too much.
  - Go has a lightweight typing system, and many types need not be written explicitly.
- Multi-core is an opportunity.
  - Go provides concurrent threads of execution as part of the language.
- Concurrency is very hard to get right.
- Computers are fast but building programs is slow.
- Programming is less fun.

Go is an attempt to solve these problems.

What problems does Go solve?

- Types get in the way too much.
  - Go has a lightweight typing system, and many types need not be written explicitly.
- Multi-core is an opportunity.
  - Go provides concurrent threads of execution as part of the language.
- Concurrency is very hard to get right.
  - Go provides channels for reliable communication: a CSP-like model.
- Computers are fast but building programs is slow.
- Programming is less fun.

Go is an attempt to solve these problems.

What problems does Go solve?

- Types get in the way too much.
  - Go has a lightweight typing system, and many types need not be written explicitly.
- Multi-core is an opportunity.
  - Go provides concurrent threads of execution as part of the language.
- Concurrency is very hard to get right.
  - Go provides channels for reliable communication: a CSP-like model.
- Computers are fast but building programs is slow.
  - Go's package system minimizes the effect of dependencies.
- Programming is less fun.

Go is an attempt to solve these problems.

What problems does Go solve?

- Types get in the way too much.
  - Go has a lightweight typing system, and many types need not be written explicitly.
- Multi-core is an opportunity.
  - Go provides concurrent threads of execution as part of the language.
- Concurrency is very hard to get right.
  - Go provides channels for reliable communication: a CSP-like model.
- Computers are fast but building programs is slow.
  - Go's package system minimizes the effect of dependencies.
- Programming is less fun.
  - The language is small and does not get in your way.

Go is an attempt to solve these problems.

Why a new language?

The problems are inherent to existing languages.

New libraries means moving in the wrong direction. Adding to something complex can only make it more complex. Go aims for simplicity.

Go is an attempt to start over. Nothing has been added to Go unless there was a clear advantage for it discovered while writing real programs.

Performance

The performance of the best Go code is unlikely to match the performance of the best C++ code.

- Go is safe, with array bounds checking and no pointer arithmetic.
- Go is garbage collected.
- Go does not have templates and thus does not have inline template specializations.
  - Some form of generics are a possible future language extension.

Gccgo does beat gcc on a few single-threaded benchmarks, and is generally competitive on most. Gccgo is significantly faster on some benchmarks where Go makes it easy to parallelize the problem.

Hello, Go

```
package main

import "fmt"

func main() {
        fmt.Print("Hello, 世界\n")
}
```

Slices

```
// A slice type has no length.
type ArrayOfInt [10]int
type SliceOfInt []int

// A slice expression makes a slice.
var v = ArrayOfInt{1, 2, 3, 4, 5,
        6, 7, 8, 9, 10}
var s = v[2:5]
// Index like an array.
var x = s[0] // x == 3

// Slices have a length and a capacity.
// len(s) == 5; cap(s) == 8
// Slice index ranges from 0 to len(s).

// Reslice a slice to make it larger.
var s2 = s[0:8]
```

Methods

Any type can have methods.

```
// Uppercase names are public.
type Point struct {
    X, Y float
}
func (p *Point) Abs() float {
    return math.Sqrt(p.X*p.X + p.Y*p.Y)
}

type Polar struct {
    R, Theta float
}
func (p Polar) Abs() float {
    return p.R
}
```

Interfaces 1

An interface type is a list of methods. An interface value may hold any value whose type implements those methods.

```
type Abser interface {
    Abs() float
}
func f() {
    point := &Point{1, 2}
    var a Abser = point
    // Static type of a is (always) Abser.
    // Dynamic type of a is (now) Point.
    fmt.Print(a.Abs())
    polar := Polar{3, 4}
    a = polar
    // Dynamic type of a is Polar.
    fmt.Print(a.Abs())
}
```

No need to explicitly declare that `Point` satisfies `Abser`.

Interfaces 2

The `io` package defines the `io.Writer` interface.

```
type Writer interface {
    Write(p []byte) (n int, err os.Error)
}
```

Any type with a `Write` method with that signature satisfies the `io.Writer` interface. Any function that needs to write something can take a `io.Writer` as a parameter.

E.g., `fmt.Fprintf` takes an `io.Writer` parameter.

E.g., `bufio.NewWriter` takes an `io.Writer` and returns a buffered type that satisfies `io.Writer`.

Interfaces 3

- A value of one interface type may be converted to another interface type.
- The conversion succeeds if the dynamic type—the type of the value stored in the interface—supports all the methods of the destination interface.
- The conversion may fail at runtime with a `panic`.
- To dynamically test without panicking, use the "comma ok" form.

```
a, ok := Abser(p)
if ok {
    // a is not nil.
} else {
    // a is nil.
}
```

Goroutines

- Goroutines are concurrent execution threads.
- In gccgo there is one goroutine per pthread; in gc goroutines are multiplexed onto operating system threads.
- The go statement starts a goroutine.

```
func f() {
    go expensiveComputation()
    anotherExpensiveComputation()
}
```

Channels 1

To communicate with a goroutine, use a channel.

```
func computeAndSend(ch chan int) {
    ch <- expensiveComputation()
}

func f() {
    ch := make(chan int)
    go computeAndSend(ch)
    v2 := anotherExpensiveComputation()
    v1 := <-ch
    fmt.Println(v1, v2)
}
```

Channels 2

- Channels are both a communication mechanism and a synchronization mechanism.
- A channel write is a release operation on all memory stores before the write.
- A channel read is an acquire operation for all memory reads following the read.
- Go slogan: *Do not communicate by sharing memory; instead, share memory by communicating.*

A multiplexed server

```
type Request struct {
    a, b   int
    // reply channel
    replyc chan int
}

type binOp func(a, b int) int

func run(op binOp, req *Request) {
    req.replyc <- op(req.a, req.b)
}

func server(op binOp, service chan *Request) {
    for {
        req := <-service // requests arrive here
        go run(op, req)  // don't wait for op
    }
}

func StartServer(op binOp) chan *Request {
    reqChan := make(chan *Request)
    go server(op, reqChan)
    return reqChan
}
```

The client

```
// Start server; receive a channel on which
// to send requests.
server := StartServer(
          func(a, b int) int {return a+b})

// Create requests
req1 := &Request{23,45, make(chan int)}
req2 := &Request{-17,1<<4, make(chan int)}

// Send them in arbitrary order
server <- req
server <- req2

// Wait for the answers in arbitrary order
fmt.Printf("Answer2: %d\n", <-req2.replyc)
fmt.Printf("Answer1: %d\n", <-req1.replyc)
```

Defer

- The `defer` statement executes a call when the function returns.
- Deferred calls are executed in LIFO order.
- The `defer` statement implements a `finally` clause, but it is dynamic rather than syntactic.

```
func printInt(i int) {
    for i != 0 {
        defer fmt.Print(i % 10)
        i /= 10
    }
}
```

Panic and recover

- The `panic` function throws an exception.
- Runtime errors such as interface conversion failures call `panic` with a value that satisfies the `runtime.Error` interface.
- A `panic` unwinds the stack, executing deferred functions.
- The `recover` function stops the stack unwind and returns the value passed to `panic`.
- If `recover` is called when no `panic` is in progress, `recover` returns `nil`.
- If a `panic` reaches the top of a goroutine stack without being recovered, it aborts the program.
- The `recover` function implements a `catch` clause, but it is dynamic rather than syntactic.

Panic and recover example

```
func SafeDivide(x, y int)
                (ret int, error string) {
    defer func() {
        if e := recover(); e != nil {
            if y == 0 {
                error = "division by zero"
            } else if y == -1 {
                error = "division overflow"
            } else {
                panic(e)
            }
        }
    }()
    ret = x / y
}
```