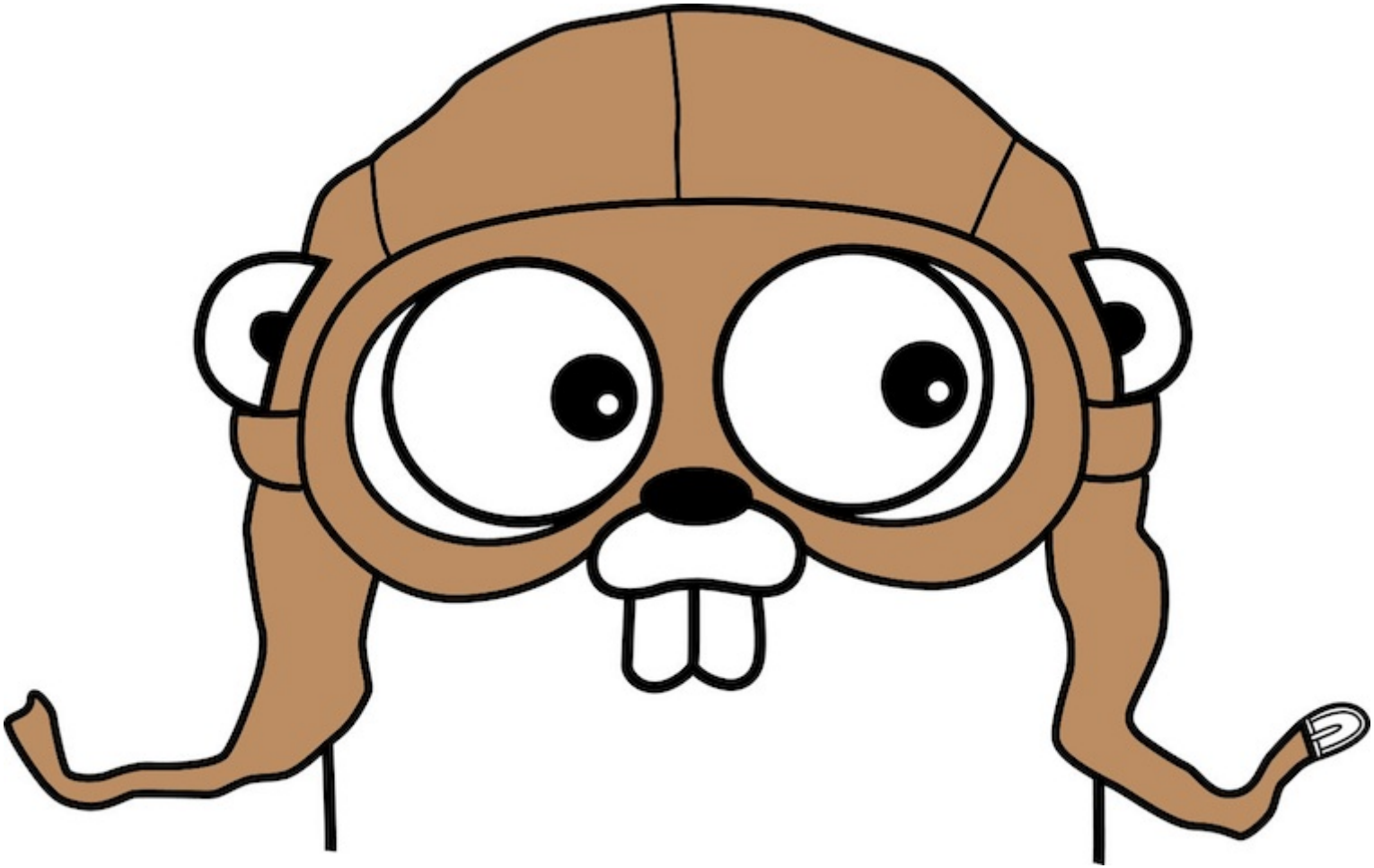
PostgreSQL and Google Go Language

April 14, 2015, Austin, Texas, USA

John Scott
Consultant, American Messaging, 2006-now
Founder, SetSpace, Inc, 1998-now

Glenda the Gopher



About the Talk

- History, Motivation and Highlights of Go Language
- How to Query PostgreSQL in Go
- Benchmarks Comparing Go Query to C Code

About Go

Go is an open source programming language that makes it easy to build simple, reliable, and efficient software.

Design began in late 2007.

- Robert Griesemer, Rob Pike, Ken Thompson
- Russ Cox, Ian Lance Taylor

Became open source in November 2009.

Developed entirely in the open; very active community.
Language stable as of Go 1, early 2012.
Work continues.

Go

A deliberately simple but powerful and fun language.

- start with C, remove complex parts
- add interfaces, concurrency
- also: garbage collection, closures, reflection, slices ...
- strings encoded in UTF-8 only

For more background on design:

- Less is exponentially more (http://commandcenter.blogspot.com/2012/06/less-is-exponentially-more.html)
- Go at Google: Language Design in the Service of Software Engineering (http://talks.golang.org/2012/splash.article)

Building Go Executables

- Statically compiled and linked executables
- No Runtime Libraries - Not Even libc!
- Size of Smallest Executable is Small (400KB)
- Process Startup about 50-100x Quicker Java
- Range Checking on Arrays/Slice
- Null Pointer Deferences Caught at Runtime
- No Magic

Production Go

- Docker blog.golang.org/docker (http://blog.golang.org/docker)
- Dropbox github.com/dropbox/godropbox (https://github.com/dropbox/godropbox)
- YouTube, Twitch
- GitHub, Tumblr (gocircuit!), Heroku
- Canonical/Ubuntu, Mozilla, SoundCloud
- Digital Ocean, Getty Images, DataDog
- Apple, Walmart, Intel are Recently Advertising Go Jobs
- Honest Dollar in Austin

For Jobs See golangprojects.com (http://www.golangprojects.com/)

Go Syntax Easy to Carry in Your Head

- for {} is only loop construct
- if/else, switch{}, goto, labeled break
- common unary/binary operators: */+- == != > >= < <= ++ -- << >>
- address/content prefix operators: & * (with garbage collection!)
- json style struct/interface {} declarations
- arrays and slices (python)
- functions with closures (javascript)
- interface just a set of methods
- channels multiplex with select{} statement

Unified Declaration and Assignment

```
//  create variable initialized to utf-8 string
//  no need for 'var' statement

s := "hello, world"

i := 0

//  explicit creation of 8 bit unsigned variable
ui := uint8(0)
```

New Types - Unified Declaration and Assignment

```
type bean_count uint16

//  cast 0 to bean_count and create a variable
bc := bean_count(0)

Println(bc)
Println(uint16(bc))
Println(bean_count(0))
```

- Cast must be explicit but compiler is smart about type lineage

Methods - Unified Declaration and Assignment

```
type bean_count uint16

//  bind private print() to any bean_count type

func (bc bean_count) print() {
        Println(bc)
}

...

    bc := bean_count(0)
    bc.print()
```

- A function bound to a type is called a "Method"

Structures - Unified Declaration and Assignment

```
type wine struct {
    vitner      string
    name        string
    year        uint16
}

//  json style initialization of new struct value

w := wine{
    vitner:    "Conchay Toro",
      name:    "Casillero del Diablo",
      year:    2008,
}
w.year = 2013

//  take address of wine structure value.  '&' operator from C language

wp := &w

//  dereference pointer using '.' and not the C style '->' operator

wp.year = 2012
```

Function Pointers - Unified Declaration and Assignment

```
//  variable info is a pointer to a function - named closure

info := func(msg string) {
    Println(msg)
}

info("hello, world")

...

info("good bye, cruel world")
```

Memory Reference - Unified Declaration and Assignment

```
//  create variable initialized to utf-8 string

s := "hello, world"

//  take memory address of variable s

sp := &s

//  change contents of what new variable s references

*sp = "good bye, cruel world"
```

Run Memory Reference - Unified Declaration and Assignment

```
package main

import "fmt"

func main() {

    //  create variable initialized to utf-8 string

    s := "hello, world"
    fmt.Println(s)

    //  take memory address of variable s

    sp := &s

    //  change what s references

    *sp = "good bye, cruel world"

    fmt.Println(s)        //  prove it
}
```

Run

Arrays and Slices - Unified Declaration and Assignment

- Arrays are static with immutable sizes
- Slices (from python) are windows into a static array
- Slices reference from 0
- Slices can grow and shink but not beyond the capacity of array

Channels - Unified Declaration and Assignment

- Think of full duplex pipes for any data type ... including channels

```
done := make(chan int)

...

//  send the boolean value 'true' down the channel.
//  block until other ends reads

done <- true

...

//  the other end waits for done

<- done
Print("finished")
```

Go is a Blend of Sequential and Concurrent Coding

Typical Sequential Coding

Fibonacci series: f(n) = f(n-1) + f(n-2)

```
package main
import . "fmt"

func fib(i int) int {

    if i < 2 {
        return 1
    }

    return fib(i - 1) + fib(i - 2)
}

func main() {

    Println(fib(40))

}
```

Run

Slower Sequential Coding

Fibonacci series: f(n) = f(n-1) + f(n-2)

```
package main
import . "fmt"

func fib(i int) int {

    if i < 2 {
        return 1
    }

    return fib(i - 1) + fib(i - 2)
}

func main() {

    Println(fib(40), fib(40), fib(40), fib(40))
}
```

Run

Easy Concurrent Coding

```
package main
import . "fmt"

func fib(i int) int {
    if i < 2 {
        return 1
    }
    return fib(i - 1) + fib(i - 2)
}
func main() {

    answer := make(chan int)

    fib_worker := func(i int) {
        answer <- fib(i)
    }

    go fib_worker(40)        // Just add "go" in front of function call
    go fib_worker(40)
    go fib_worker(40)
    go fib_worker(40)

    Println(<- answer, <- answer, <- answer, <- answer)
}
```

Run

Concurrency with Channels

Share memory by communicating rather than communicating by sharing memory

- Based on Communicating Sequential Processes by Sir Tony Hoare
- Channels are Typed, Two Way Pipes
- Writer Blocks until Reader Completes
- Read on Closed Channel Returns Nil - A Cheap Broadcast
- Write on Closed Channel is Runtime Panic
- Multiple Channels Read/Written with select{} statement
- Channels can be Buffered

Any Datatype Sent Over a Channel

Channels can be sent over channels

```
type session struct {
    name    string
}

c := make(chan *session)

// channel of channels of pointers to sessions
load_balancer := make(chan chan *session)
```

"Hello, World" is Load Balancer

Select Reads/Write Many Channels at Once

```
func flow (north, south, east, west chan int) {

    var message int

    for {
        select {

        case message <- west:
            ..
        case message <- south:
            ..
        case north <- message:
            ..
        case east <- message:
            ..
        }
    }
}
```

Go Memory Management

- Garbage Collection of Unreferenced Data
- True Pointers and Writing Directly into Memory
- Programmer Controls Memory Layout
- Possible to Write a Typed malloc(), Relieving Pressure on Collector

Database/Sql Core Package and PostgreSQL pq

GoLang Package database/sql

Abstracts SQL Databases

- Similar to famous Perl package named DBI
- Written by Brad Fitzpatrick (Live Journal)
- Two PostgreSQL Drivers: native or static link against C library libpq
- Other Drivers: MS Server, MySQL, SQLite, DB2, ODBC, Oracle8, Sybase, Firebird, YQL

Builtin Connection Pooling

- All databases drivers can pool
- Single DB Object transparently pools across connections
- Transaction binds to single pool connection
- SetMin/MaxIdleConnections()

database/sql Data Types

- Database
- Driver (referenced via URI)
- Statement
- Transaction
- Row, Rows,
- Result
- Bool, Float64, Int64, String

database/sql Setup Methods

- Open()
- Prepare()
- Close()
- Begin()
- Commit()
- Rollback()

Query Methods - Package database/sql

- Exec()
- Query()
- QueryRow()
- Next()
- Scan()

Example - Query Multiple Rows

```
rows, err := db.Query("SELECT ...")

...

defer rows.Close()

for rows.Next() {
    var id int
    var name string

    err = rows.Scan(&id, &name)

    ...
}

err = rows.Err() // get any error encountered during iteration
...
```

PostgreSQL Driver database/sql/pq

- 100% go language - no linking of C library libpq
- Hidden driver underneath generic core library
- Not distributed with golang core - available on github
- Very, very fast

```
$ go get github.com/lib/pq
```

sql-bench.go - import statements

```
package main

import (
    "bufio"
    "database/sql"
    "io"
    "os"
    "strings"
    "syscall"

    . "fmt"
    _ "github.com/lib/pq"
)
```

sql-bench.go - constants & declarations

```
const (
    select_stmt = `
     select
         exists (
          select
              cookie
            from
                login_session
            where
                cookie = $1
        )`
    QUERY_COUNT = 9
    COOKIE_CHANNEL_SIZE = 96
)

var (
    Stdin = os.NewFile(uintptr(syscall.Stdin), "/dev/stdin")
)

type stat struct {      //  track stats for select queries for each go routine
    true_count      uint64
    false_count     uint64
}
```

sql-bench.go - open database and prepare statement

```
func open_db() (
    db *sql.DB,
    stmt *sql.Stmt,
) {
    var err error

    //  open postgresql database, inheriting PG params from environment
    db, err = sql.Open("postgres", "sslmode=disable")
    if err != nil {
        panic(err)
    }

    db.SetMaxIdleConns(8)       //  keep 8 idle connections
    db.SetMaxOpenConns(128)     //  no more 128 simultaneous db connection

    stmt, err = db.Prepare(select_stmt)
    if err != nil {
        panic(err)
    }
    return db, stmt
}
```

sql-bench.go - query worker

```
func select_cookies(
    db *sql.DB,                 //  database connection
    stmt *sql.Stmt,             //  prepared statment
    cookies chan string,        //  read cookies
    done chan *stat,            //  answer with stats
) {
    stat, exists := &stat{}, false

    //  read cookies on string channel
    for cookie := range cookies {
        err := stmt.QueryRow(cookie).Scan(&exists)
        if err != nil {
            panic(err)
        }
        if exists {
            stat.true_count++
        } else {
            stat.false_count++
        }
    }
    done <- stat
}
```

sql-bench.go - send cookies to competing workers

```
func send_cookies (cookies chan string) {

    //  for each cookie read on standard input,
    //  send that cookie to some query go routines
    in := bufio.NewReader(Stdin)
    for {
        cookie, err := in.ReadString('\n')
        if err != nil {
            if err == io.EOF {
                break
            }
            panic(err)
        }

        //  send to competing pool of selects
        cookies <- strings.TrimRight(cookie, "\n")
    }
    close(cookies)
}
```

sql-bench.go - wait for queries to finish, print stats

```
func wait(done chan *stat) {

    true_count := uint64(0)
    false_count := uint64(0)

    //  wait for select()s to finish in the
    for i := 0;  i < QUERY_COUNT;  i++ {

        // read stat from each terminating query worker
        s := <- done

        //  accumulate total stats as workers exit
        true_count += s.true_count
        false_count += s.false_count
    }
    Printf(" True/False Count: %d/%d\n", true_count, false_count)
}
```

sql-bench.go - run the benchmark in main()

```
func main() {

    //  open postgresql database, inheriting PG params from environment
    db, stmt := open_db()

    // works indicate finished by sending stats on 'done' channel
    done := make(chan *stat)

    // workers compete by reading string cookies from this channel
    cookies := make(chan string, COOKIE_CHANNEL_SIZE)

    //  spawn competing go routines
    for i := 0;  i < QUERY_COUNT;  i++ {
        go select_cookies(db, stmt, cookies, done)
    }
    //  boot up cookie switch in background
    go send_cookies(cookies)

    // synchronously wait for results
    wait(done)

    os.Exit(0)     //  good bye, cruel world
}
```

Benchmark Comparing Queries in Go/pq to C/libpq

Source Code for C and Go Benchmarks

Go Benchmark in `sql-bench.go`

http:pgdfw-gopq/sql-bench.go (http:pgdfw-gopq/sql-bench.go)

Shell Script to Run `sql-bench-go.sh`

http:pgdfw-gopq/sql-bench-go-sh.txt (http:pgdfw-gopq/sql-bench-go-sh.txt)

C Benchmark in `sql-bench.pgc`

http:pgdfw-gopq/sql-bench-pgc.txt (http:pgdfw-gopq/sql-bench-pgc.txt)

Shell Script to Run `sql-bench.pgc`

http:pgdfw-gopq/sql-bench-pgc-sh.txt (http:pgdfw-gopq/sql-bench-pgc-sh.txt)

Apolgies for txt links for source code

.txt file cause problems for browser - try hard refresh in browser

Software & Hardware Environment for Benchmark

PostgreSQL 9.4.1

```
4GB Shared Buffers
Compiled From Source
```

My Desktop iMac OSX 10.9.5

```
Intel Core i7 @ 3.4GHz, 4 Cores, 8 Threads
32G RAM, DDR3 @ 1600 MHz Bus, 8Mb L2 Cache
786Gb Apple SSD SM768E, Intel Controller
```

Query a Snapshot of Production HTTP Session Cookie Table

Tuple Width is 41 bytes

```
create table public.login_session
(
    cookie          udig primary key,    --  is just a sha1

    create_time     timestamp,
    active_time     timestamp
);
```

1,525,746 Tuples in Table with 160 bit Primary Key (SHA)

```
SELECT
    EXISTS (
      SELECT
          cookie
        FROM
            login_session
        WHERE
            cookie = $1
    )
;
```

Who Won the Shootout?

Go *pq* Package

```
16.50s for 1525861 queries = 92,476 lookup/sec
```

C *libpq* Library

```
15.37s for 1525861 queries = 99,275 lookup/sec
```

Perhaps pgbench in Go Language?

Links

Interactive Tour of Go (http://tour.golang.org/)

An Introduction to Programming in Go - Online Book (http://www.golang-book.com/)

Meet the Go Team - Q/A at Google I/O 2012 (https://www.youtube.com/watch?v=sln-gJaURzk&list=PLoJWLKOp927tFsMbO2onhp26NnOAKAtO#t=921)

Concurrency is Not Parallelism by Rob Pike @ Vimeo (http://vimeo.com/49718712)

Communication Sequential Processes - The Theory That Inspired Go @ Wikipedia (http://en.wikipedia.org/wiki/Communicating_sequential_processes)

Communicating Sequential Processes - Readable Intro to CSP Algebra (PDF) (http://www.usingcsp.com/cspbook.pdf)

Searchable Documentation on Popular Go Packages (http://www.godoc.org)

PostgreSQL Package PQ Driver @ GoDoc (http://godoc.org/github.com/lib/pq)

Thank you

April 14, 2015, Austin, Texas, USA

John Scott
Consultant, American Messaging, 2006-now
Founder, SetSpace, Inc, 1998-now

jmscott@setspace.com (mailto:jmscott@setspace.com)
john.scott@americanmessaging.net (mailto:john.scott@americanmessaging.net)