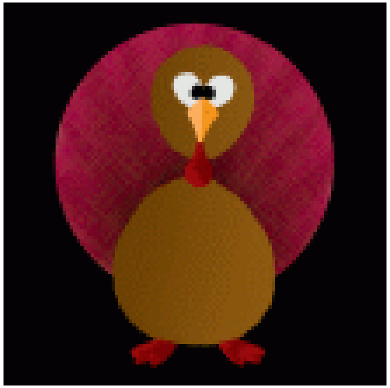
See Python, See Python Go, Go Python Go

Presented by **Andrey Petrov** at **PyCon 2016**

Andrey Petrov

@shazow



urllib3, ssh-chat, briefmetrics, colorblendy, tweepsect, funny tweets, and more

Do you Go?

Do you Go?

- Simple syntax that is easy to learn

Do you Go?

- Simple syntax that is easy to learn
- Compiles superfast

Do you Go?

- Simple syntax that is easy to learn
- Compiles superfast
- Statically typed

Do you Go?

- Simple syntax that is easy to learn
- Compiles superfast
- Statically typed
- Statically linked binaries

Do you Go?

- Simple syntax that is easy to learn
- Compiles superfast
- Statically typed
- Statically linked binaries
- Cross-compiles to ~every platform

Do you Go?

- Simple syntax that is easy to learn
- Compiles superfast
- Statically typed
- Statically linked binaries
- Cross-compiles to ~every platform
- Easy concurrency

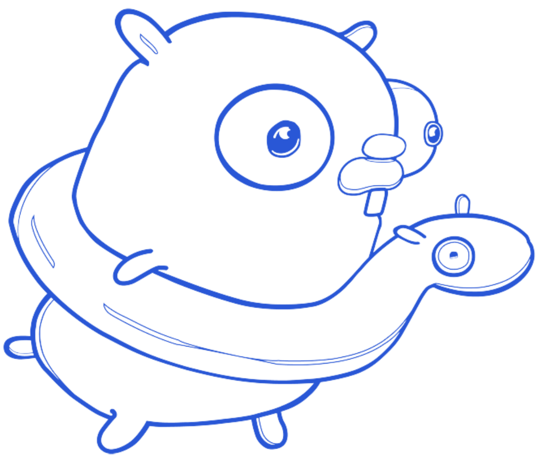
Do you Go?

- Simple syntax that is easy to learn
- Compiles superfast
- Statically typed
- Statically linked binaries
- Cross-compiles to ~every platform
- Easy concurrency
- Great standard library

Do you Go?

- Simple syntax that is easy to learn
- Compiles superfast
- Statically typed
- Statically linked binaries
- Cross-compiles to ~every platform
- Easy concurrency
- Great standard library
- Gophers!

Python Go



Credit: Renee French & Alice Tribuleva

Go in Python

```python
import os

os.system("go run main.go")
```

Go in Python

```python
import os

os.system("go run main.go")
```

lol just kidding

Fun Facts

- Python speaks with C

Fun Facts

- Python speaks with C
- Go speaks with C

Fun Facts

- Python speaks with C
- Go speaks with C
- Therefore, Python speaks with Go?

Challenges

1. Runtime barriers

Challenges

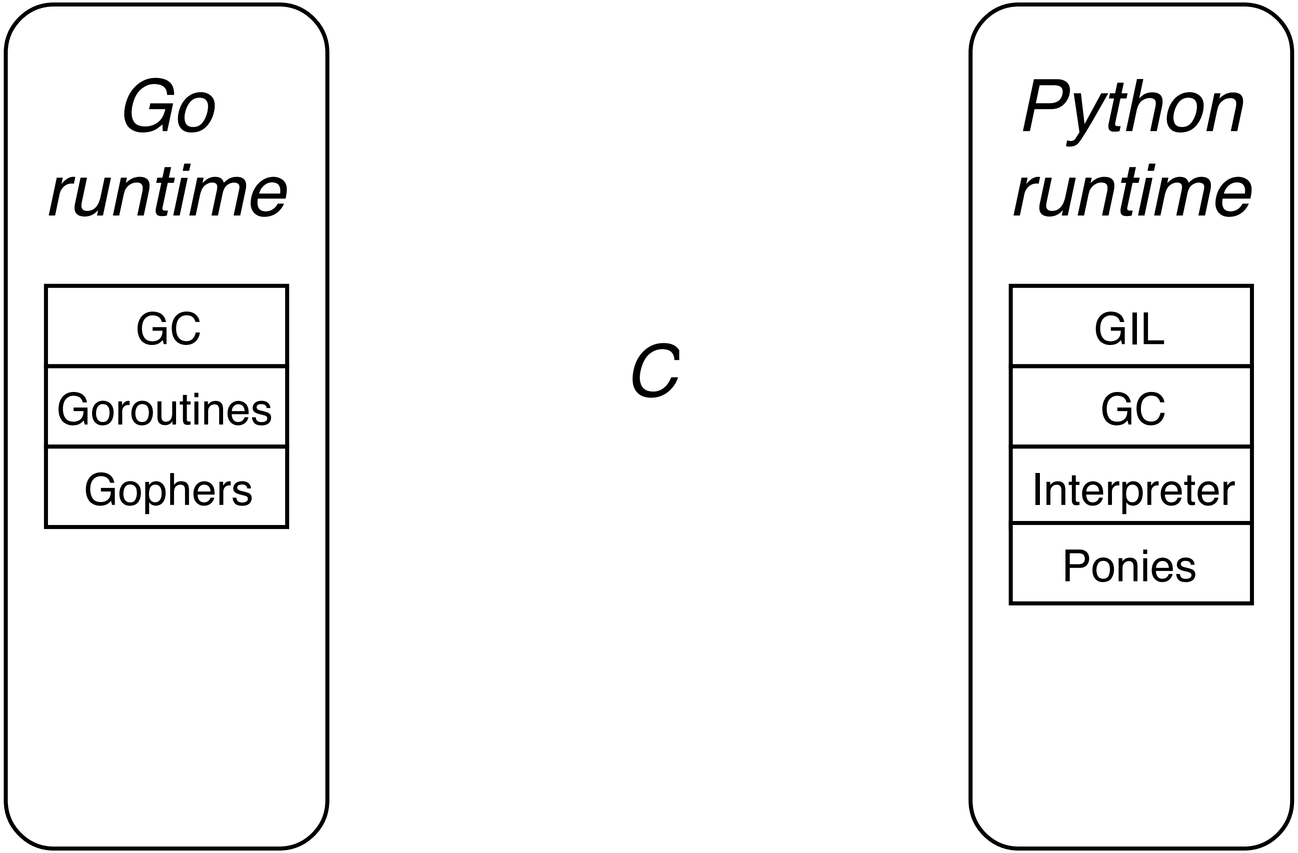
1. Runtime barriers
    - Garbage Collectors (GC)
    - Global Interpreter Lock (GIL)
    - Just In Time compiling (JIT)
    - Resource Pools (Threads)

Challenges

1. Runtime barriers
    - Garbage Collectors (GC)
    - Global Interpreter Lock (GIL)
    - Just In Time compiling (JIT)
    - Resource Pools (Threads)
2. Syntax and feature barriers

Challenges

1. Runtime barriers
    - Garbage Collectors (GC)
    - Global Interpreter Lock (GIL)
    - Just In Time compiling (JIT)
    - Resource Pools (Threads)
2. Syntax and feature barriers
    - Go: Interface, Goroutines, etc.
    - Python: Classes, Generators, etc.
    - Oodles of other language-specific constructs


Go runtime
GC
Goroutines
Gophers
C
Python runtime
GIL
GC
Interpreter
Ponies

Let's take a moment and imagine...

Running a webserver in Go

```go
package main

import (
    "fmt"
    "net/http"
)

func index(w http.ResponseWriter, req *http.Request) {
    fmt.Fprintf(w, "Hello, world.\n")
}

func main() {
    http.HandleFunc("/", index)
    http.ListenAndServe("127.0.0.1:5000", nil)
}
```

Running a webserver in Python

```python
from flask import Flask
app = Flask(__name__)

@app.route('/')
def index():
    return 'Hello, world!\n'

if __name__ == '__main__':
    app.run(host='127.0.0.1', port=5000)
```

Running a Go webserver in Python??

```
from gohttp import route, run

@route('/')
def index(w, req):
    w.write("Hello, world.\n")

if __name__ == '__main__':
    run(host='127.0.0.1', port=5000)
```

Running a Go webserver in Python??

```
from gohttp import route, run

@route('/')
def index(w, req):
    w.write("Hello, world.\n")

if __name__ == '__main__':
    run(host='127.0.0.1', port=5000)
```

Compare

```
from flask import Flask
app = Flask(__name__)

@app.route('/')
def index():
    return 'Hello, world!\n'

if __name__ == '__main__':
    app.run(host='127.0.0.1', port=5000)
```

Comparing handlers

Go (net/http)

```go
func index(w http.ResponseWriter, req *http.Request) {
    fmt.Fprintf(w, "Hello, world.\n")
}
```

Go in Python (gohttplib)

```python
def index(w, req):
    w.write("Hello, world.\n")
```

Python (flask)

```python
def index():
    return 'Hello, world!\n'
```

gohttplib

This is an actual functioning thing.

github.com/shazow/gohttplib

gohttplib

This is an actual functioning thing.

github.com/shazow/gohttplib

Sponsored by [Glider Labs](#)

Here's how it works...

The Plan

1. Go: Export Go functions to a C shared library

2. C:

3. Python: Call C and wrap it in a Python-shaped bow

4. Make it actually work ¯_(ツ)_/¯

WORLD OF GO

Andrey Petrov · @shazow

Calling C from Go

```
package main

/*
int the_answer() {
    return 42;
}
*/
import "C"
import "fmt"

func main() {
    r := C.the_answer()
    fmt.Println(r)
}
```

Calling C from Go

```
package main

/*
int the_answer() {
    return 42;
}
*/
import "C"
import "fmt"

func main() {
    r := C.the_answer()
    fmt.Println(r)
}
```

```
$ go build -o answer
$ ./answer
42
```

Calling Go from C

```
package main

import "C"

//export TheAnswer
func TheAnswer() C.int {
    return C.int(42)
}

func main() {}
```

Calling Go from C

```go
package main

import "C"

//export TheAnswer
func TheAnswer() C.int {
    return C.int(42)
}

func main() {}
```

```
$ go build -buildmode=c-shared -o libanswer.so
```

Calling Go from C

```go
package main

import "C"

//export TheAnswer
func TheAnswer() C.int {
    return C.int(42)
}

func main() {}
```

```
$ go build -buildmode=c-shared -o libanswer.so
```

```c
#include <stdio.h>
#include "libanswer.h"

int main() {
    int r = TheAnswer();
    printf("%d\n", r);
    return 0;
}
```

WoRLD OF

Andrey Petrov · @shazow

See CPython C

- [CPython Extension Interface]: no dependencies, but lots of boilerplate
- [CFFI]: a little more magic but does more work for us and more portable

Calling C from Python

```python
# answer_build.py:
from cffi import FFI
ffi = FFI()

ffi.cdef("int the_answer();")

ffi.set_source("_answer", """
    int the_answer() {
        return 42;
    }
""")

if __name__ == "__main__":
    ffi.compile()
```

Calling C from Python

```
# answer_build.py:
from cffi import FFI
ffi = FFI()

ffi.cdef("int the_answer();")

ffi.set_source("_answer", """
    int the_answer() {
        return 42;
    }
""")

if __name__ == "__main__":
    ffi.compile()
```

```
$ python answer_build.py
$ ls
_answer.c       _answer.o       _answer.so      answer_build.py
```

Calling C from Python

```python
# answer_build.py:
from cffi import FFI
ffi = FFI()

ffi.cdef("int the_answer();")

ffi.set_source("_answer", """
    int the_answer() {
        return 42;
    }
""")

if __name__ == "__main__":
    ffi.compile()
```

```
$ python answer_build.py
$ ls
_answer.c       _answer.o       _answer.so      answer_build.py
```

```python
# answer.py:
from _answer import lib

print(lib.the_answer())
```

Calling Python from C

Simple function pointer that can be used in C:

```
@ffi.callback("int(int, int)")
def add(x, y):
    return x + y
```

Calling Python from C

Simple function pointer that can be used in C:

```
@ffi.callback("int(int, int)")
def add(x, y):
    return x + y
```

~handwaving~

Calling Python from C

Simple function pointer that can be used in C:

```
@ffi.callback("int(int, int)")
def add(x, y):
    return x + y
```

~handwaving~

```
static int (*add)(int x, int b);
```

That's all we need for now. More on CFFI embedding here.

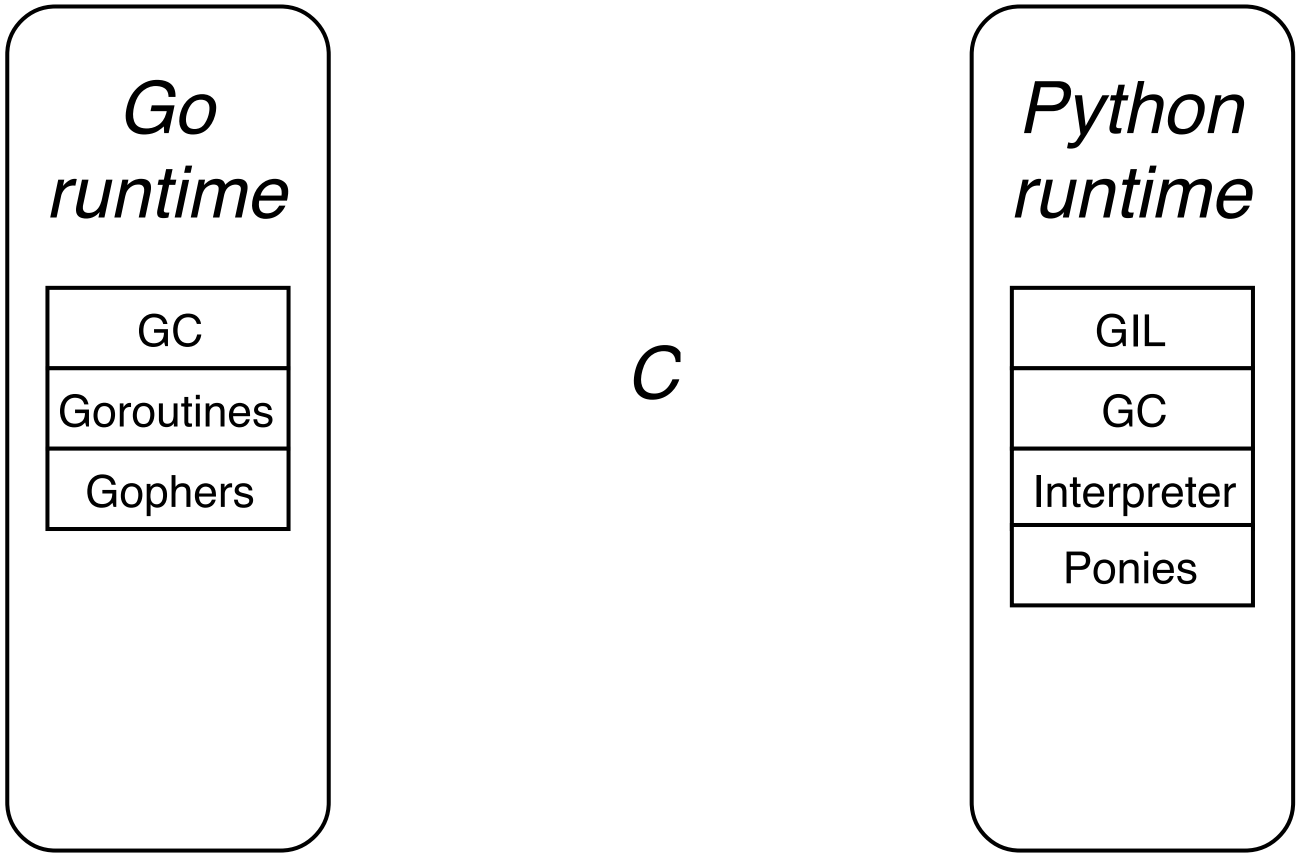
Andrey Petrov · @shazow

Challenges

1. Runtime barriers
    - Garbage Collectors (GC)
    - Global Interpreter Lock (GIL)
    - Just In Time compiling in PyPy (JIT)
    - Thread Pools
2. Syntax and feature barriers
    - Go: Interface, Goroutines, etc.
    - Python: Classes, Generators, etc.
    - Oodles of other language-specific constructs

Overcoming Challenges

1. **Don't share memory**


Go runtime
GC
Goroutines
Gophers
C
Python runtime
GIL
GC
Interpreter
Ponies

Challenge 1: Don't share memory

```
http.HandleFunc(pattern, func(w http.ResponseWriter, req *http.Request) {
    ...
})
```

Our Python code will need to provide this callback, but we can't pass `*http.Request` to Python.

Challenge 1: Don't share memory

```
http.HandleFunc(pattern, func(w http.ResponseWriter, req *http.Request) {
    ...
})
```

Our Python code will need to provide this callback, but we can't pass `*http.Request` to Python.

```
typedef struct Request_
{
    const char *Method;
    const char *Host;
    ...
} Request;
```

Challenge 1: Don't share memory

```
http.HandleFunc(pattern, func(w http.ResponseWriter, req *http.Request) {
    ...
})
```

Our Python code will need to provide this callback, but we can't pass `*http.Request` to Python.

```
typedef struct Request_
{
    const char *Method;
    const char *Host;
    ...
} Request;
```

```
http.HandleFunc(pattern, func(w http.ResponseWriter, req *http.Request) {
    // Wrap relevant request fields in a C-friendly datastructure.
    creq := C.Request{
        Method: C.CString(req.Method),
        Host:   C.CString(req.Host),
        ...
    }
    ...
```

Overcoming Challenges

1. Don't share memory

2. **Translation layer**

Challenge 2: Translation layer

```
http.HandleFunc(pattern, func(w http.ResponseWriter, req *http.Request) {
    ...
})
```

Our Python code will need to use the `http.ResponseWriter` interface.

Challenge 2: Translation layer

```
http.HandleFunc(pattern, func(w http.ResponseWriter, req *http.Request) {
    ...
})
```

Our Python code will need to use the `http.ResponseWriter` interface.

```
ResponseWriter.Write([]byte) (int, error)
ResponseWriter.WriteHeader(int)
```

Challenge 2: Translation layer

```
http.HandleFunc(pattern, func(w http.ResponseWriter, req *http.Request) {
    ...
})
```

Our Python code will need to use the `http.ResponseWriter` interface.

```
ResponseWriter.Write([]byte) (int, error)
ResponseWriter.WriteHeader(int)
```

```
//export ResponseWriter_Write
func ResponseWriter_Write(wPtr C.uint, cbuf *C.char, length C.int) C.int {
    buf := C.GoBytes(unsafe.Pointer(cbuf), length)
    ...
    n, _ := (*(*http.ResponseWriter)(w)).Write(buf)
    return C.int(n)
}

//export ResponseWriter_WriteHeader
func ResponseWriter_WriteHeader(wPtr C.uint, header C.int) {
    ...
    (*(*http.ResponseWriter)(w)).WriteHeader(int(header))
}
```

Challenge 2: Translation layer (Cont.)

We exported these functions in Go

```
ResponseWriter_Write(wPtr C.uint, cbuf *C.char, length C.int) C.int
ResponseWriter_WriteHeader(wPtr C.uint, header C.int)
```

Now we can wrap them in Python

```
lib = ffi.dlopen(...)

class ResponseWriter:
    def __init__(self, w):
        self._w = w

    def write(self, body):
        n = lib.ResponseWriter_Write(self._w, body, len(body))
        if n != len(body):
            raise IOError("Failed to write to ResponseWriter.")

    def set_status(self, code):
        lib.ResponseWriter_WriteHeader(self._w, code)
```

Challenge 2: Translation layer (Cont.)

Original interface in Go:

```
type http.ResponseWriter interface { WriteHeader(int) }
```

Challenge 2: Translation layer (Cont.)

Original interface in Go:

```
type http.ResponseWriter interface { WriteHeader(int) }
```

Exported interface from Go:

```
func ResponseWriter_WriteHeader(wPtr C.uint, header C.int)
```

Challenge 2: Translation layer (Cont.)

Original interface in Go:

```
type http.ResponseWriter interface { WriteHeader(int) }
```

Exported interface from Go:

```
func ResponseWriter_WriteHeader(wPtr C.uint, header C.int)
```

C header:

```
void ResponseWriter_WriteHeader(unsigned int p0, int p1);
```

Challenge 2: Translation layer (Cont.)

Original interface in Go:

```
type http.ResponseWriter interface { WriteHeader(int) }
```

Exported interface from Go:

```
func ResponseWriter_WriteHeader(wPtr C.uint, header C.int)
```

C header:

```
void ResponseWriter_WriteHeader(unsigned int p0, int p1);
```

Accessing C from Python:

```
ResponseWriter_WriteHeader(w, header)
```

Challenge 2: Translation layer (Cont.)

Original interface in Go:

```
type http.ResponseWriter interface { WriteHeader(int) }
```

Exported interface from Go:

```
func ResponseWriter_WriteHeader(wPtr C.uint, header C.int)
```

C header:

```
void ResponseWriter_WriteHeader(unsigned int p0, int p1);
```

Accessing C from Python:

```
ResponseWriter_WriteHeader(w, header)
```

Python wrapper:

```
ResponseWriter.set_status(self, code)
```

Overcoming Challenges

1. Don't share memory

2. Translation layer

3. **Seriously, don't share memory**

Challenge 3: Seriously, don't share memory

- Wait, what's a Go interface?

Challenge 3: Seriously, don't share memory

- Wait, what's a Go interface?
- To use it, we need to pass a *pointer* through

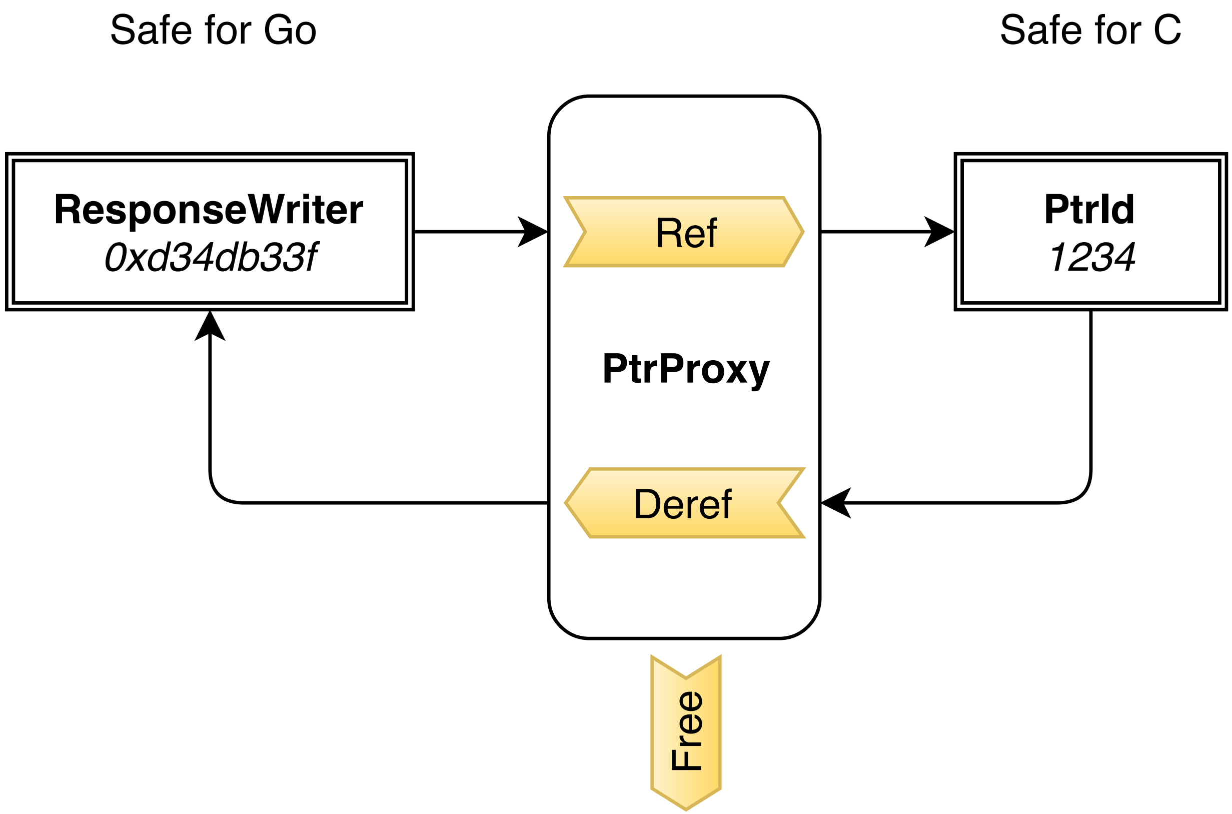
  ☞ Go ⇢ C ⇢ Python ⇢ C ⇢ Go ☞

Challenge 3: Seriously, don't share memory

- Wait, what's a Go interface?
- To use it, we need to pass a *pointer* through

  ☞ Go ⇢ C ⇢ Python ⇢ C ⇢ Go ☞
- Solution: Pointer Proxy!

Pointer Proxy



Andrey Petrov · @shazow

Pointer Proxy

```
type ptrProxy struct {
    sync.Mutex
    count  uint
    lookup map[uint]unsafe.Pointer
}

func (p *ptrProxy) Ref(ptr unsafe.Pointer) C.uint { ... }

func (p *ptrProxy) Deref(id C.uint) (unsafe.Pointer, bool) { ... }

func (p *ptrProxy) Free(id C.uint) { ... }
```

Now we can pass an opaque uint across enemy lines and it's perfectly safe.

Quick flashback: Translation layer

One pointer proxy to rule them all.

```
var cpointers = PtrProxy()
```

Quick flashback: Translation layer

One pointer proxy to rule them all.

```
var cpointers = PtrProxy()
```

In the callback, reference to bind them.

```
http.HandleFunc(pattern, func(w http.ResponseWriter, req *http.Request) {
    // Wrap relevant request fields in a C-friendly datastructure.
    creq := C.Request{ ... }

    wPtr := cpointers.Ref(unsafe.Pointer(&w))
    ...
    cpointers.Free(wPtr)
})
```

Quick flashback: Translation layer

One pointer proxy to rule them all.

```
var cpointers = PtrProxy()
```

In the callback, reference to bind them.

```
http.HandleFunc(pattern, func(w http.ResponseWriter, req *http.Request) {
    // Wrap relevant request fields in a C-friendly datastructure.
    creq := C.Request{ ... }

    wPtr := cpointers.Ref(unsafe.Pointer(&w))
    ...
    cpointers.Free(wPtr)
})
```

With pointer proxy, dereference to find them.

```
func ResponseWriter_WriteHeader(wPtr C.uint, header C.int) {
    w, _ := cpointers.Deref(wPtr)
    (*(*http.ResponseWriter)(w)).WriteHeader(int(header))
}
```

Recap

- If our languages can speak with C, they can speak with each other.

Recap

- If our languages can speak with C, they can speak with each other.
- Be careful going in and out of runtimes.

Recap

- If our languages can speak with C, they can speak with each other.
- Be careful going in and out of runtimes.
- Be super-careful with sharing memory.

Recap

- If our languages can speak with C, they can speak with each other.
- Be careful going in and out of runtimes.
- Be super-careful with sharing memory.
- We'll need a translation layer to use non-trivial language constructs.

Other Considerations

Other Considerations

- Memory leaks

Other Considerations

- Memory leaks
- Race conditions

Other Considerations

- Memory leaks
- Race conditions
- Context switching overhead

Other Considerations

- Memory leaks
- Race conditions
- Context switching overhead
- Probably security issues because C is hard

Other Considerations

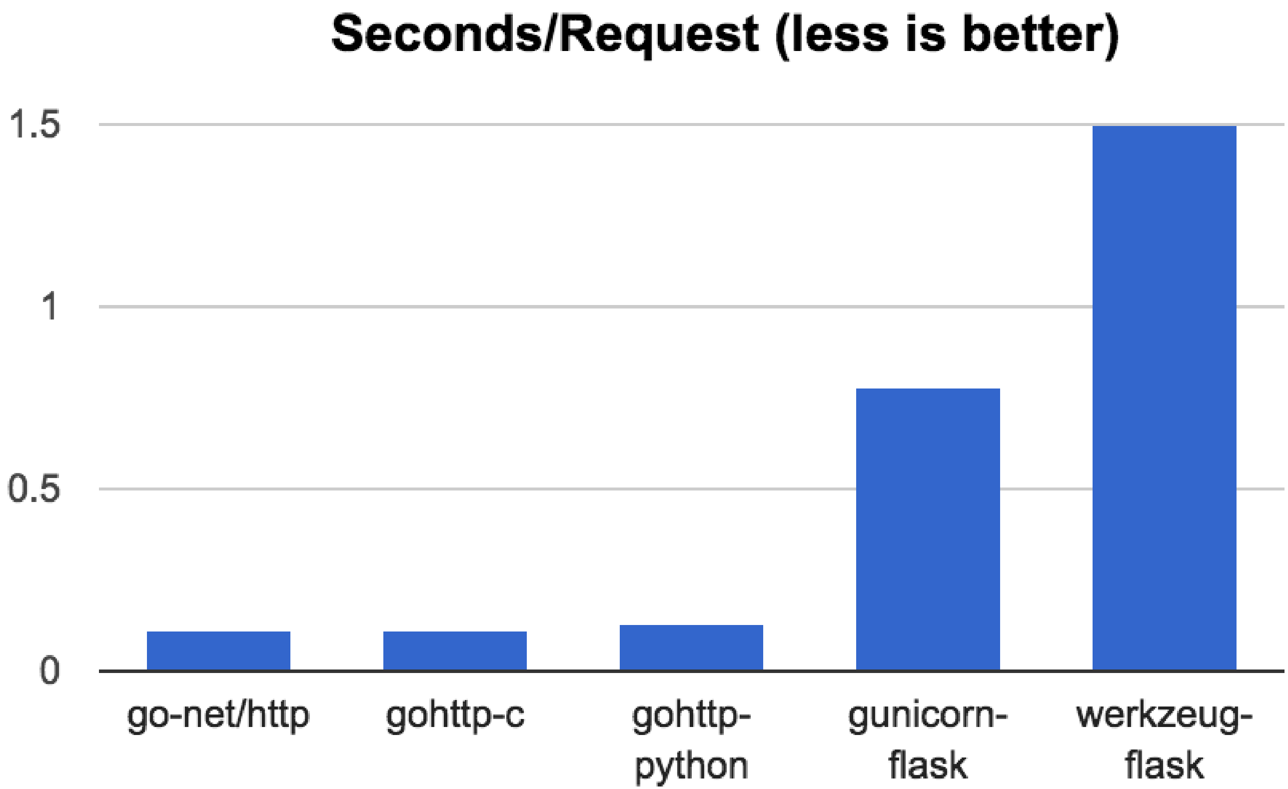
- Memory leaks
- Race conditions
- Context switching overhead
- Probably security issues because C is hard
- Architecture campanelle

A Palate Cleanser

A Palate Cleanser

LOLBENCHMARKS!

LOLBENCHMARKS



LOLBENCHMARKS

Name	Total	Req/Sec	Time/Req
go-net/http	1.115	8969.89	0.111
gohttp-c	1.181	8470.97	0.118
gohttp-python	1.285	7779.87	0.129
gunicorn-flask	7.826	1277.73	0.783
werkzeug-flask	15.029	665.37	1.503

Conditions: `ab` doing 10,000 requests with 10 concurrency on my .

Thank you

- github.com/shazow/gohttplib
- hrku.co/gopythongo
- twitter.com/shazow
- shazow.net