Cgo
Go under the hood

Rajesh Ramachandran
Qube Cinema

Why Cgo There?

- Interface with existing C libraries
- Operating Systems' interfaces
- High performance apps like signal processing
  - Vectorization
  - GPU programming

hello.c

```c
#include <stdio.h>

int main(void)
{
  printf("hello, world\n");
  return 0;
}
```

0: hello.c -> hello.go

```c
#include <stdio.h>

int main(void)
{
  printf("hello, world\n");
  return 0;
}
```

1: hello.c -> hello.go

```
package main


#include <stdio.h>

int main(void)
{
  printf("hello, world\n");
  return 0;
}
```

2: hello.c -> hello.go

```
package main

/*
#include <stdio.h>

int _main(void)
{
  printf("hello, world\n");
  return 0;
}
*/
```

3: hello.c -> hello.go

```
package main

/*
#include <stdio.h>

int _main(void)
{
  printf(“hello, world\n”);
  return 0;
}
*/
import “C”
```

4: hello.c -> hello.go

```
package main

/*
#include <stdio.h>

int _main(void)
{
  printf(“hello, world\n”);
  return 0;
}
*/
import “C”

func main() {
    C._main ()
}
```

go build

- `import` "C" triggers Cgo
  - generates clean .go files for 6g
  - generates .c/.h files
    - some are handled by gcc/clang
    - others are for 6c
  - any non-Go files in the directory are compiled
    - .c, .s or .S by the C compiler
    - .cc, .cpp, .cxx by the C++ compiler
- #cgo pseudo directives and environment variables to flag compiler and linker

Cgo generated Go wrapper

```
//hello.cgo1.go
package main

func main() {
    _Cfunc__main()
}


//_cgo_gotypes.go
func _Cfunc__main() (r1 _Ctype_int) {
    _cgo_runtime_cgocall_errno(
        _cgo_2b504f279e52_Cfunc__main,
        uintptr(unsafe.Pointer(&r1)))
    return
}
```

Cgo generated C wrapper

```
//hello.cgo2.c
#include <stdio.h>

static int _main(void)
{
  printf("%d: hello, world\n");
}

void
_cgo_2b504f279e52_Cfunc__main(void *v)
{
        struct {
                int r;
                char __pad4[4];
        } __attribute__((__packed__)) *a = v;
        char *stktop = _cgo_topofstack();
        __typeof__(a->r) r = _main();
        a = (void*)((char*)a + (_cgo_topofstack() - stktop));
        a->r = r;
}
```

Calling back into C

```
package main

/*
extern void progress(int);
static void fill(int *x, int len)
{
  int interval = len / 100;
  for(int i = 0; i < len; i++) {
    if (i % interval == 0)
      progress(i / interval);
    x[i] = i;
  }
}
*/
import "C"
import ("unsafe”; ”fmt)

func main() {
    var nums []C.int = make([]C.int, 1e9)
    C.fill((*C.int)(unsafe.Pointer(&nums[0])), (C.int)(len(nums)))
}

//export progress
func progress(percent C.int) {
    if percent%10 == 0 {
        fmt.Printf("%d%%", percent)
    } else {
        fmt.Print(".")
    }
}
```

Cgo generated C wrappers

```
// _cgo_export.c
void progress(int p0)
{
        struct {
                int p0;
                char __pad0[4];
        } __attribute__((__packed__)) a;
        a.p0 = p0;
        crosscall2(_cgoexp_6784b7ee9109_progress, &a, 8);
}

// _cgo_defun.c
void
_cgoexp_6784b7ee9109_progress(void *a, int32 n)
{
                runtime·cgocallback(·progress, a, n);
}
```

Callbacks: In an ideal world

```
package main

/*
static void fill(int *x, int len, void (*prog)(int))
{
  int interval = len / 100;
  for(int i = 0; i < len; i++) {
    if (i % interval == 0)
      prog(i / interval);
    x[i] = i;
  }
}
*/
import "C"
import ("fmt”; "unsafe”)

func main() {
    var nums []C.int = make([]C.int, 1e9)
    C.fill((*C.int)(unsafe.Pointer(&nums[0])), (C.int)(len(nums)),
         progress)
}

//export progress
func progress(percent C.int) {
    if percent%10 == 0 {
        fmt.Printf("%d%%", percent)
    } else {
        fmt.Print(".")
    }
}
```

Callbacks: With Cgo

```
package main

/*
static void fill(int *x, int len, void (*prog)(int))
{
  int interval = len / 100;
  for(int i = 0; i < len; i++) {
    if (i % interval == 0)
      prog(i / interval);
    x[i] = i;
  }
}

extern void progress(int);
static void fill_wrap(int *x, int len)
{
  fill(x, len, progress);
}
*/
import "C"
import ("fmt"; "unsafe")

func main() {
    var nums []C.int = make([]C.int, 1e9)
    C.fill_wrap((*C.int)(unsafe.Pointer(&nums[0])), (C.int)(len(nums)))
}

//export progress
func progress(percent C.int) {
    if percent%10 == 0 {
        fmt.Printf("%d%%", percent)
    } else {
        fmt.Print(".")
    }
}
```

Crossing the Chasm

- Go to C with runtime.cgocall
  - Will not block other goroutines and GC
  - Runs on OS allocated stack
  - Outside of $GOMAXPROCS accounting
- C to Go with runtime.cgocallback
  - Runs on original goroutine's stack
  - $GOMAXPROCS accounting enforced
- Recursion allowed across the chasm
- Implemented in Go, C and Assembly

Cgo

Relationship Status: It's Complicated

- Smoother start than JNI, Extension Modules
- Callbacks can be cumbersome
- Cross Platform Builds?
- Slower compile times
  - 10x on hello, world!
- GC
  - `go C.fill((*C.int)(unsafe.Pointer(&nums[0])), …`
- Changes in 1.5

Thank You!

- http://golang.org/src/runtime/cgocall.go
- http://golang.org/src/cmd/cgo/
- http://golang.org/misc/cgo/
- https://golang.org/cmd/cgo/
- http://akrennmair.github.io/golang-cgo-slides
- http://morsmachine.dk/go-scheduler