Interactions C/C++ et Go

Auteurs : Yi Wang, Gustavo Niemeyer, Andrew Gerrand, adg, Xavier Méhaut

V1.0

Sommaire

Préambule 3
Appel d'une DLL Windows à partir de Go (adg) 3
Bases de l'encapsulation du C en Go (Gustavo Niemeyer) 4
La commande cgo (golang.org) 6
C? Go? Cgo! (Andrew Gerrand) 7
Callbacks avec cgo (Ostsol) 10
Linker statiquement du code C++ avec du code en Go (YI Wang) 14
Plus de ressources concernant SWIG et GO ? 16

Préambule

Go permet d'appeler du code C voire C++ , soit en chargeant une bibliothèque partagée (.so, .dll), soit en intégrant le code Go directement avec du C (cgo, swig).

Appel d'une DLL Windows à partir de Go (adg)

Un simple exemple d'appel de DLL windows en Go:

```
package main

import (
	"syscall"
	"unsafe"
	"fmt"
)


func abort(funcname string, err int) {
	panic(funcname + " failed: " + syscall.Errstr(err))
}

var (
	kernel32, _          = syscall.LoadLibrary("kernel32.dll")
	getModuleHandle, _   = syscall.GetProcAddress(kernel32,
"GetModuleHandleW")

	user32, _            = syscall.LoadLibrary("user32.dll")
	messageBox, _        = syscall.GetProcAddress(user32,
"MessageBoxW")
)


const (
	MB_OK                      = 0x00000000
	MB_OKCANCEL                = 0x00000001
	MB_ABORTRETRYIGNORE        = 0x00000002
	MB_YESNOCANCEL             = 0x00000003
	MB_YESNO                   = 0x00000004
	MB_RETRYCANCEL             = 0x00000005
	MB_CANCELTRYCONTINUE       = 0x00000006
	MB_ICONHAND                = 0x00000010
	MB_ICONQUESTION            = 0x00000020
	MB_ICONEXCLAMATION         = 0x00000030
	MB_ICONASTERISK            = 0x00000040
	MB_USERICON                = 0x00000080
	MB_ICONWARNING             = MB_ICONEXCLAMATION
	MB_ICONERROR               = MB_ICONHAND
	MB_ICONINFORMATION         = MB_ICONASTERISK
	MB_ICONSTOP                = MB_ICONHAND

	MB_DEFBUTTON1              = 0x00000000
```

```
	MB_DEFBUTTON2			= 0x00000100
	MB_DEFBUTTON3			= 0x00000200
	MB_DEFBUTTON4			= 0x00000300
)

func MessageBox(caption, text string, style uintptr) (result int) {
	ret, _, callErr := syscall.Syscall9(uintptr(messageBox),
		0,

 uintptr(unsafe.Pointer(syscall.StringToUTF16Ptr(text))),

 uintptr(unsafe.Pointer(syscall.StringToUTF16Ptr(caption))),
		style,
		0,
		0,
		0,
		0,
		0)
	if callErr != 0 {
		abort("Call MessageBox", int(callErr))
	}
	result = int(ret)
	return
}

func GetModuleHandle() (handle uintptr) {
	if ret, _, callErr :=
syscall.Syscall(uintptr(getModuleHandle), 0, 0, 0); callErr != 0 {
		abort("Call GetModuleHandle", int(callErr))
	} else {
		handle = ret
	}
	return
}

func main() {
	defer syscall.FreeLibrary(kernel32)
	defer syscall.FreeLibrary(user32)

	fmt.Printf("Retern: %d\n", MessageBox("Done Title", "This test
is Done.", MB_YESNOCANCEL))
}

func init() {
	fmt.Print("Starting Up\n")
}
```

Bases de l'encapsulation du C en Go (Gustavo Niemeyer)

Avec CGo, le nom donné au mécanisme d'intégration d C avec Go, on declare simplement une instruction d'importation special qui indique au pré-processeur de regarder dans le commentaire le precedent pour y trouver les informations nécessaire à l'intégration voulue.

Quelque chose comme ceci :

```
// #include <zookeeper.h>
import "C"
```

Le commentaire n'a pas à se restreindre à une seule ligne, ou même à des instructions de type *#include*. Le code C contenu dans le commentaire sera inséré de manière transparente à l'intérieur du fichier C qui est compilé et *linké* avec le fichier *obj* final. Du côté Go, l'import "C" est simulé comme si c'était un import normal de telle manière que les fonctions C, les valeurs, et les types sont directement accessibles.

Voici comme exemple une fonction C avec son prototype :

```
int zoo_wexists(zhandle_t *zh, const char *path, watcher_fn watcher,
                void *context, struct Stat *stat);
```

qui peut être utilisée en Go de la façon suivante :

```
cstat := C.struct_Stat{}
rc, cerr := C.zoo_wexists(zk.handle, cpath, nil, nil, &cstat)
```

Quand la fonction C est utilisée dans un contexte où deux valeurs sont requises, comme c'est le cas ci-dessus, Cgo conservera au chaud la variable bien connue *errno* après la fin d'exécution de la fonction et la renverra encapsulée (wrapped) dans la valeur *os.Errno*.

Notez également comment la structure C est définie de telle façon qu'elle peut être passée directement à la fonction C. Fait intéressant, l'allocation de la mémoire pour la structure est réalisée et suivie par le *runtime* Go et sera récupérée de manière appropriée une fois qu'aucune référence n'existera plus à *l'intérieur du runtime Go*. Il faut garder cela à l'esprit puisque l'application crashera si une valeur allouée normalement à l'intérieur du programme Go est sauvée dans une fonction étrangère (*foreign function*) C et maintenue malgré tout après que toutes les références Go aient disparues.

L'alternative est dans ce cas d'appeler les fonctions habituelles C pour récupérer la mémoire de ces valeurs impliquées. Cette mémoire ne sera pas affectée par le ramasse-miettes, et, bien entendu, devra être explicitement libérée quand elles ne seront plus nécessaires. Ci-dessous un exemple simple montrant une allocation explicite :

```
cbuffer := (*C.char)(C.malloc(bufferSize))
defer C.free(unsafe.Pointer(cbuffer))
```

Notez l'utilisation de l'instruction **defer** ci-dessus. Même lorsqu'il s'agit de fonctionnalités étrangères (foreign), c'est très pratique. L'appel ci-dessus va s'assurer que le *buffer* est désalloué juste avant que la fonction ne se termine.

En terme de typage, Go est encore plus stricte que le C et Cgo s'assurera que les types retournés et passé à la fonction étrangère seront correctement typées. Notez également ci-dessus comment l'appel à la fonction *free()* a besoin de convertir explicitement la valeur en *unsafe.Pointer*, même si en C il n'y a pas besoin de *cast* pour transformer un pointeur en un paramètre *void **. Pour tout autre conversion de type, Go s'assurera à la compilation que la conversion est une opération sûre.

La commande cgo (golang.org)

Cgo permet la création de packages Go qui appelant du code C.

Usage:

```
cgo [compiler options] file.go
```

Les options de compilation sont passées sans interprétation à l'invocation de *gcc* pour compiler les parties C du package.

Le fichier d'entrée *file.go* est un fichier Go syntactiquement valide qui importe le pseudo-package "C" et qui référence des types comme *C.size_t*, des variables comme *C.stdout*, ou des fonctions telles *C.putchar*.

Si l'import de "C" est immédiatement précédé par un commentaire, ce commentaire est utilisé comme entête à la compilation des parties en C du package. Par exemple :

```
// #include <stdio.h>
// #include <errno.h>
import "C"
```

CFLAGS et LDFLAGS peuvent être définies avec la pseudo directive #cgo à l'intérieur du commentaire. Pour piloter le comportement de gcc. Les valeurs définies dans de multiples directives sont concaténées ensemble. Les options préfixées par $GOOS, $GOARCH, ou $GOOS/$GOARCH sont seulement définies dans les systèmes correspondants. Par exemple :

```
// #cgo CFLAGS: -DPNG_DEBUG=1
// #cgo linux CFLAGS: -DLINUX=1
// #cgo LDFLAGS: -lpng
// #include <png.h>
import "C"
```

Alternatively, CFLAGS and LDFLAGS may be obtained via the pkg-config tool using a '#cgo pkg-config:' directive followed by the package names. For example:

```
// #cgo pkg-config: png cairo
// #include <png.h>
import "C"
```

A l'intérieur d'un fichier Go, les identifieurs C ou les noms de champs qui sont des mots clefs en Go peuvent être accédés en les préfixant avec un _ : si x pointe vers une structure en C avec un champ nommé « type, `c._type` permet d'accéder à ce champ.

Les types standards numériques du C sont disponibles sous les noms suivants : `C.char`, `C.schar (signed char)`, `C.uchar (unsigned char)`, `C.short`, `C.ushort (unsigned short)`, `C.int`, `C.uint (unsigned int)`, `C.long`, `C.ulong (unsigned long)`, `C.longlong (long long)`, `C.ulonglong (unsigned long long)`, `C.float`, `C.double`.

Le type `void*` est représenté par le pointeur non sûr unsafe.Pointer en Go.

Pour accéder à un `struct`, une `union`, ou un `enum` directement, les préfixer respectivement par `struct_`, `union_`, ou `enum_`, comme dans `C.struct_stat`.

N'importe quelle fonction qui renvoie une valeur peut être appelée dans de multiples contextes d'affectation pour récupérer et la valeur de retour, et la variable C *errno* comme une *os.Error*. Par exemple:

```
n, err := C.atoi("abc")
```

En C, un argument de fonction écrit comme un tableau de valeurs fixes requiert en fait un pointeur sur le premier élément du tableau. Les compilateurs C sont conscients de la convention d'appel et ajustent l'appel en conséquence, mais Go ne le peut pas. En Go, vous devez passer explicitement le pointeur au premier élément: `C.f(&x[0])`.

Quelques rares fonctions convertissent des types Go en C er réciproquement en effectuant une copie des données. Dans des definitions en pseudo-Go:

```
// Go string to C string
func C.CString(string) *C.char

// C string to Go string
func C.GoString(*C.char) string

// C string, length to Go string
func C.GoStringN(*C.char, C.int) string

// C pointer, length to Go []byte
func C.GoBytes(unsafe.Pointer, C.int) []byte
```

Cgo transforme le fichier d'entrée en quatre fichiers de sortie:

- Deux fichiers source en Go
- Un fichier C pour 6c (or 8c or 5c),
- Et un fichier pour gcc

Les règles du makefile dans le package standard dans Make.pkg automatise l'usage de cgo. Voir $GOROOT/misc/cgo/stdio et $GOROOT/misc/cgo/gmp comme exemples.

Cgo ne fonctionne pas encore avec gccgo.

C? Go? Cgo! (Andrew Gerrand)

Cgo laisse les packages Go appeler le code. Etant donné un fichier source en Go écrit avec des caractéristiques spéciales, cgo génère des fichiers Go et C qui peuvent être combinées en un seul package Go.

Pour avancer avec un exemple, ci-dessous voici un package qui fournit deux fonctions - `Random` and `Seed` – qui encapsulent les fonctions C `random` et `srandom`.

```
package rand
```

```
/*
#include <stdlib.h>
*/
import "C"

func Random() int {
   return int(C.random())
}

func Seed(i int) {
   C.srandom(C.uint(i))
}
```

Regardons ce qu'il advient ici, en commençant par l'instruction import.

La package *rand* importe "C", mais vous ne trouverez aucun package dans la bibliothèque standard de Go. Et ceci parce qu'il s'agit d'un pseudo-package., un nom spécial interprété par cgo comme une référence à un espace de noms C.

La package *rand* contient 4 références au package C : les appels à `C.random` et `C.srandom`, la conversion `C.uint(i)`, et l'instruction d'import.

La fonction *Random* appelle la fonction *random libc* et renvoie le résultat. En C, *random* retourne une valeur du type C `long`, qui Cgo représente comme `C.long`. Il doit être converti en un type Go avant d'être utilisé par le code Go en dehors de ce package, en utilisant une conversion de type ordinaire Go :

```
func Random() int {
   return int(C.random())
}
```

Nous avons ici une fonction équivalente qui utilise une variable temporaire pour illustrer la conversion de type de manière plus explicite :

```
func Random() int {
   var r C.long = C.random()
   return int(r)
}
```

La fonction `Seed` fait l'inverse. Elle prend un *int* Go habituel, le convertit en un type C `unsigned int`, et le passe à al fonction C `srandom`.

```
func Seed(i int) {
   C.srandom(C.uint(i))
}
```

Notez que cgo connait le `unsigned int` en tant que `C.uint`; voir cgo documentation pour une liste complete des noms de type numériques reconnus.

Un détail que nous n'avons pas encore examiné est le commentaire au dessus de l'instruction import :

```
/*
#include <stdlib.h>
```

```
*/
import "C"
```

Cgo reconnait ce commentaire et l'utilise comme une entête lorsqu'il compile les parties C. Dans ce cas, il s'agit juste d'une simple instruction, mais cela peut être n'importe quel code en C. ! Le commentaire doit être immédiatement mis avant la ligne qui importe « C », sans ligne vide, ou autre commentaire.

Strings et choses

Contrairement à Go, C ne possède pas de type chaine explicite. Les strings en C sont représentées par un tableau de caractères terminé par zéro.

La conversion entre des chaînes Go et C est effectué avec les fonctions `C.CString`, `C.GoString`, et `C.GoStringN`. Ces conversions créent une copie de la chaîne de caractères.

L'exemple suivant implémente une fonction `Print` qui écrit sur la sortie standard en utilisant la fonction C `fputs` de la bibliothèque `stdio`:

```
package print

// #include <stdio.h>
// #include <stdlib.h>
import "C"
import "unsafe"

func Print(s string) {
   cs := C.CString(s)
   C.fputs(cs, (*C.FILE)(C.stdout))
   C.free(unsafe.Pointer(cs))
}
```

Les allocations mémoire faites par le C ne sont pas connues par le gestionnaire de mémoire Go. Quand vous créez une *string* C avec `C.CString` (ou n'importe quelle fonction d'allocation mémoire en C), vous devez vous souvenir de libérer la mémoire quand vous en avez terminé avec les chaines de caractères utilisées en appelant `C.free`.

L'appel à `C.CString` renvoie un pointeur vers le début du tableau, ainsi avant que la fonction ne sorte, nous la convertissons en un `unsafe.Pointer` (équivalent en Go du `void* C`) et nous libérons la mémoire allouée avec `C.free`. Un idiome courant en Go est d'utuliser l'instruction **defer** :

```
func Print(s string) {
   cs := C.CString(s)
   defer C.free(unsafe.Pointer(cs))
   C.fputs(cs, (*C.FILE)(C.stdout))
}
```

Construire des packages cgo

Si vous utilisez goinstall, la construction des packages cgo est triviale. Il suffit juste

d'invoquer *goinstall* comme à l'accoutumée et l'import `"C"` sera automatiquement reconnu et cgo sera utilisé pour les fichiers concernés.
Si vous désirez construire l'exécutable avec des *Makefiles* Go, la variable `CGOFILES` liste les fichiers devant être pris en charge par cgo, juste comme la variable `GOFILES` contient la liste des fichiers sources Go.

UN `Makefile` pour le package `rand` reseemblerait à cela :

```
include $(GOROOT)/src/Make.inc

TARG=goblog/rand
CGOFILES=\
   rand.go\

include $(GOROOT)/src/Make.pkg
```

Pour builder, taper "`gomake`" comme d'habitude.

Plus de ressources sur Cgo?

Pour des exemples nombreux voir :

http://golang.org/misc/cgo/

Pour un exemple simple et idiomatique , voir :

http://code.google.com/p/gosqlite/source/browse/sqlite/sqlite.go

Ainsi que :

http://godashboard.appspot.com/project?tag=cgo

et finalement :

http://code.google.com/p/go/source/browse/src/pkg/runtime/cgocall.c

Callbacks avec cgo (**Ostsol**)

Une limitation majeure de cgo est le manque de support pour les callbacks ou, en des termes plus généraux, la possibilité pour le C d'appeler une fonction Go. C'est un impératif fréquent quand on désire construire un système base sur les évènements. A part l'utilisation de SWIG, il existe tout de même une solution.

Dans un des fichiers cgo, exposer rune fonction se fait très simplement : on fait précéder l'entête de la fonction par un commentaire avec le mot export et suivi par le nom de la fonction. Par exemple :

```
//export Foo

func Foo() {

   // ...

}
```

Quand cgo voit ceci, il génère deux nouveaux fichiers C :

- _cgo_export.c
- _cgo_export.h.

Ces fichiers fournissent les wrappers qui permettent à un programme C d'appeler des fonctions en Go.

Fourni avec la distribution courante de Go, il existe un exemple de programme montrant ce fonctionnement. Il peut être trouvé ici :

```
$GOBIN/misc/cgo/life
```

En regardant cet exemple, vous devriez noter un problème : ce n'est pas exemple classique de callback. Go ne passé pas de fonction au C. A la place, les fonctions sont simplement exposées de telle façon que le C peut les appeler. Ainsi le problème demeure dans le cas de systèmes guidés par les évènements tels que GLUT. Les callbacks sont toutefois rendues possibles en utilisant un brin d'indirection.

Commençons par créer une bibliothèque simple qui ajoute un nombre via une callback. La fonction *call_add* prend un pointeur sur une fonction et l'appelle pour ajouter deux entiers.

```
// calladd.h
typedef int(*adder)(int, int);

extern void call_add(adder);

//callback.c

#include <stdio.h>

#include "calladd.h"

void
```

```
call_add(adder func) {
    printf("Calling adder\n");
    int i = func(3, 4);
    printf("Adder returned %d\n", i);
}
```

Rien de plus. Compilez-la et transformez la en objet partagé (shared object).

```
gcc -g -c -fPIC calladd.c
gcc -shared -o calladd.so calladd.o
```

Maintenant la partie intéressante : dans le but de passer une fonction Go au C, nous devons créer un wrapper qui passé le wrapper à la fonction Go. C'est pas beau ça? Deux niveaux d'indirection !!!

```
// funcwrap.h
extern void pass_GoAdd(void);

// funcwrap.c

#include "_cgo_export.h"

#include "calladd.h"

void

pass_GoAdd(void) {
    call_add(&GoAdd);
}
```

Notre fichier cgo est plutôt simple:

```
//callback.go
package callback

// #include "callgo.h"
// #include "funcwrap.h"

import "C"

func Run() {
    // call the wrapper
```

```
    C.pass_GoAdd()
}
//export GoAdd
func GoAdd(a, b int) int {
    return a + b
}
```

Ceci n'est pas réellement une callback, puisque nous ne pouvons pas passer une fonction arbitraire. Ceci requerrait un niveau supplémentaire d'indirection.

Pour conclure, vous pouvez trouver ici le *Makefile* de ce projet :

```
CGO_LDFLAGS=_cgo_export.o calladd.so funcwrap.so $(LDPATH_linux)
CGO_DEPS=_cgo_export.o calladd.so funcwrap.so

CLEANFILES+=main

include $(GOROOT)/src/Make.pkg

funcwrap.o: funcwrap.c _cgo_export.h
    gcc $(_CGO_CFLAGS_$(GOARCH)) -g -c -fPIC $(CFLAGS) funcwrap.c

funcwrap.so: funcwrap.o
    gcc $(_CGO_CFLAGS_$(GOARCH)) -o $@ funcwrap.o $(_CGO_LDFLAGS_$(GOOS))

main: calladd.so install main.go
    $(GC) main.go
    $(LD) -o $@ main.$O
```

Linker statiquement du code C++ avec du code en Go (YI Wang)

Depuis la version 2.0, SWIG peut supporter le langage Go. Cela rend possible l'écriture de programmes appelant du C ou du C++. Habituellement, le code C/C++ et les wrappers générés par SWIG sont générés dans une *shared library* ; le code invoquant Go et le code d'encapsulation Go généré par SWIG se trouvent une fois compilé dans le même exécutable, qui dynamiquement lie (link) à la bibliothèque partagée.

Cette solution est suffisante à moins que vous n'écriviez un programme de calcul distribué, qui doit être déployé sur de nombreux ordinateurs avec différentes version de la bibliothèque installées. Dans ce contexte, dans le but d'éviter une trop grande complexité de déploiement, nous voulons habituellement linker statiquement les bibliothèques dynamiques en un seul exécutable. De toute façon, je n'ai pas trouvé de façon de liner du code C/C++ statiquement dans du code Go quand j'utilise le GC (5g, 6g, 8g etc), parce que le code d'encapsulation C/C++ généré par SWIG est pour le Plan 9 C compiler (5c, 6c, 8c etc), qui ne peut pas être linké avec du code venant de GCC.

Néanmoins, merci grandement à Ian Taylor et son GCCGO, le *frontend* Go pour GCC, qui me permet maintenant de lier du code go statiquement avec du code C/C++ (code venant du document SWIG):

```
01 /* File : example.c */

02

03 double My_variable  = 3.0;

04

05 /* Compute factorial of n */

06 int fact(int n) {

07     if (n <= 1) return 1;

08     else return n*fact(n-1);

09 }

10

11 /* Compute n mod m */

12 int my_mod(int n, int m) {
```

```
13     return(n % m);
14 }
```

Et le fichier d'interface correspondant SWIG

```
01 /* File : example.i */
02 %module example
03 %{
04 /* Put headers and other declarations here */
05 extern double My_variable;
06 extern int  fact(int);
07 extern int  my_mod(int n, int m);
08 %}
09
10 extern double My_variable;
11 extern int  fact(int);
12 extern int  my_mod(int n, int m);
```

Nous générons le code d'encapsulation C/C++ (*example_wrap.c*) et le code d'encapsulation Go (*example.go*) utilisant SWIG:

```
swig -go -gccgo example.i
```

En lisant le code Go généré, vous pouvez savoir comment écrire le programme appelant. Regardons le fichier `main.go` comme exemple :

```
1 package main
2
3 import "fmt"
4 import "example"
5
```

```
6 func main() {

7   fmt.Printf("Hello World:%d\n", example.Fact(3))

8 }
```

Compilons tout ce code C/C++ et Go:

gcc -c example.c -o example_impl.o
gcc -c example_wrap.c -o example_wrap.o
gccgo -c example.go -o example.o
gccgo -c main.go -o main.o

et linkons les ensemble (statiquement):

```
gccgo main.o example.o example_wrap.o example_impl.o -o hello
```

Et voila! Maintenant, nous pouvons exécuter le binaire généré `hello` et obtenir :

```
Hello World!:6
```

Plus de ressources concernant SWIG et GO ?

http://www.swig.org/Doc2.0/Go.html