The Go Blog

C? Go? Cgo!

17 March 2011

Introduction

Cgo lets Go packages call C code. Given a Go source file written with some special features, cgo outputs Go and C files that can be combined into a single Go package.

To lead with an example, here's a Go package that provides two functions - `Random` and `Seed` - that wrap C's `random` and `srandom` functions.

```
package rand

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

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

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

Let's look at what's happening here, starting with the import statement.

The `rand` package imports `"C"`, but you'll find there's no such package in the standard Go library. That's because `C` is a "pseudo-package", a special name interpreted by cgo as a reference to C's name space.

The `rand` package contains four references to the `C` package: the calls to `C.random` and `C.srandom`, the conversion `C.uint(i)`, and the `import` statement.

The `Random` function calls the standard C library's `random` function and returns the result. In C, `random` returns a value of the C type `long`, which cgo represents as the type `C.long`. It must be converted to a Go type before it can be used by Go code outside this package, using an ordinary Go type conversion:

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

Next article

Gobs of data

Previous article

Go becomes more stable

Links

golang.org
Install Go
A Tour of Go
Go Documentation
Go Mailing List
Go+ Community
Go on Twitter

Blog index

Here's an equivalent function that uses a temporary variable to illustrate the type conversion more explicitly:

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

The `Seed` function does the reverse, in a way. It takes a regular Go `int`, converts it to the C `unsigned int` type, and passes it to the C function `srandom`.

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

Note that cgo knows the `unsigned int` type as `C.uint`; see the cgo documentation for a complete list of these numeric type names.

The one detail of this example we haven't examined yet is the comment above the `import` statement.

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

Cgo recognizes this comment. Any lines starting with `#cgo` followed by a space character are removed; these become directives for cgo. The remaining lines are used as a header when compiling the C parts of the package. In this case those lines are just a single `#include` statement, but they can be almost any C code. The `#cgo` directives are used to provide flags for the compiler and linker when building the C parts of the package.

There is a limitation: if your program uses any `//export` directives, then the C code in the comment may only include declarations (`extern int f();`), not definitions (`int f() { return 1; }`). You can use `//export` directives to make Go functions accessible to C code.

The `#cgo` and `//export` directives are documented in the cgo documentation.

Strings and things

Unlike Go, C doesn't have an explicit string type. Strings in C are represented by a zero-terminated array of chars.

Conversion between Go and C strings is done with the `C.CString`, `C.GoString`, and `C.GoStringN` functions. These conversions make a copy of the string data.

This next example implements a `Print` function that writes a string to standard output using C's `fputs` function from the `stdio` library:

```
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))
}
```

Memory allocations made by C code are not known to Go's memory manager. When you create a C string with `C.CString` (or any C memory allocation) you must remember to free the memory when you're done with it by calling `C.free`.

The call to `C.CString` returns a pointer to the start of the char array, so before the function exits we convert it to an `unsafe.Pointer` and release the memory allocation with `C.free`. A common idiom in cgo programs is to `defer` the free immediately after allocating (especially when the code that follows is more complex than a single function call), as in this rewrite of `Print`:

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

Building cgo packages

To build cgo packages, just use `go build` or `go install` as usual. The go tool recognizes the special `"C"` import and automatically uses cgo for those files.

More cgo resources

The cgo command documentation has more detail about the C pseudo-package and the build process. The cgo examples in the Go tree demonstrate more advanced concepts.

Finally, if you're curious as to how all this works internally, take a look at the introductory comment of the runtime package's cgocall.go.

By Andrew Gerrand

Related articles

- HTTP/2 Server Push
- Introducing HTTP Tracing

- Generating code
- Introducing the Go Race Detector
- Go maps in action
- go fmt your code
- Organizing Go code
- Debugging Go programs with the GNU Debugger
- The Go image/draw package
- The Go image package
- The Laws of Reflection
- Error handling and Go
- "First Class Functions in Go"
- Profiling Go Programs
- A GIF decoder: an exercise in Go interfaces
- Introducing Gofix
- Godoc: documenting Go code
- Gobs of data
- JSON and Go
- Go Slices: usage and internals
- Go Concurrency Patterns: Timing out, moving on
- Defer, Panic, and Recover
- Share Memory By Communicating
- JSON-RPC: a tale of interfaces