Data Science & Tech Blog

Communicating between Go and Python or R

Matthew Mahowald on *Thu, Jun 13, 2019*

Data science and engineering teams at Open Data Group are polyglot by design: we like to choose the best tool for the task at hand. Most of the time, this means our services and components communicate through things like client libraries and RESTful APIs. But sometimes, we need code from one language to call code written in another language directly. In this post, we'll take a short look at how to do that using C foreign function interfaces (FFI) as a way to call functions written in Go using Python.

Getting started with FFIs

A foreign function interface is a mechanism by which a program written in one language can call routines or make use of services written in another. (Thanks, Wikipedia!) FFIs are how many popular data science and machine learning packages function. For example, NumPy, Pandas, and Tensorflow all have much of their core numeric operations implemented in C, and expose bindings to Python (and,

Recent Posts

- Machine Learning Model Interpretation
  15 Aug 2019
- Matching for Non-Random Studies
  25 Jul 2019
- Distances and Data Science
  27 Jun 2019
- Communicating between Go and Python or R
  13 Jun 2019
- An Introduction to Hierarchical Models
  23 May 2019

for Tensorflow, other languages) to call these functions. This pattern allows users to keep the flexibility and ease of writing code in their favorite language (e.g. Python), while also taking advantage of the performance gains available when using native C or Fortran code.

Python has a number of packages that enable wrapping C functions with Python, the most popular of which is likely the built-in `ctypes` package in the standard library. R also has good native C (and C++) support, as well as convenient "higher level" support through the `Rcpp` package. Hadley Wickham has a great introduction to the Rcpp package on his website. Since we'll be looking at calling Go code from Python or R via a C FFI, the Go C FFI documentation is also a helpful resource.

The setup

Our goal is to call a function written in Go using Python or R. To do that, we need some functions in Go. Here are two of them:

```
package readwrite

import (
  "fmt"
)

func Read(path string) []byte {
  result := []byte("read from " + path)
    return result
}

func Write(data []byte, path string) {
  fmt.Println("Go: wrote to", path)
    return
}
```

These functions are largely vacuous, but will demonstrate passing binary data between Go and other languages.

From Go to C

These two functions are pure Go, but there's no native Go-Python FFI (yet), so we should also wrap them with functions that can be

exported to C using `cgo`. This is done by including the `"C"` package in your Go program's import statement, adding `//export` comments to the functions you want to export, and then building a C shared object with the result:

```
package main

import (
  "C"
  "path/to/readwrite"
  "fmt"
  "unsafe"
  "encoding/binary"
)

func main() {
  fmt.Println("All main'd up!")
  return
}

//export Read
func Read(path *C.char) unsafe.Pointer {
  s := C.GoString(path)
  read := readwrite.Read(s)
  length := make([]byte, 8)
  binary.LittleEndian.PutUint64(length, uint64(len(read)))
  return C.CBytes(append(length, read...))
}

//export Write
func Write(data *C.char, path *C.char, size C.int) {
  d := C.GoBytes(unsafe.Pointer(data), size)
  s := C.GoString(path)
  readwrite.Write([]byte(d), s)
}
```

These functions look a little different than what we wrote in the `readwrite` package! A couple of notes about what's going on here:

1. For string data, we have to convert between C strings (pointers to `char` arrays) and Go-native strings. Fortunately, there's a library function that does this (`C.GoString`).
2. For the C-exportable `Read` and `Write` functions, we also need to include information about how many bytes our byte arrays will be. This is because any code that calls these functions will need to know how many bytes to expect from C. We could accomplish this by including a stop character (e.g. a null byte) to terminate

the byte array, but since we want to support passing arbitrary binary data back and forth, I've instead chosen to make the first 8 bytes the *length* of the rest of the byte array. Then, the calling code can read the first 8 bytes to know how many more bytes to expect. (This limits us to a maximum of about 18,446,744 terabytes per message, but hopefully that's enough!)

To build the C shared object, just use the `go` build tool with the `c-shared` buildmode:

```
go build -o libreadwrite.so -buildmode=c-shared
```

Calling our functions from Python

Using `ctypes`, it's relatively straightforward to call our functions in Python. First, import `ctypes`, load the shared object, and bind the right argument types to each function:

```
import ctypes

lib = ctypes.cdll.LoadLibrary("./libreadwrite.so")
lib.Read.argtypes = [ctypes.c_char_p]
lib.Read.restype = ctypes.POINTER(ctypes.c_ubyte*8)
lib.Write.argtypes = [ctypes.c_char_p, ctypes.c_char_p, ctypes.c_int]
```

Note that we're setting the initial response for `Read` to just 8 bytes—recall that the first 8 bytes are the length of the remaining binary data we want to read in.

Now let's define our wrapper functions:

```
def read(path):
    ptr = lib.Read(path.encode("utf-8"))
    length = int.from_bytes(ptr.contents, byteorder="little")
    data = bytes(ctypes.cast(ptr,
            ctypes.POINTER(ctypes.c_ubyte*(8 + length))
            ).contents[8:])
    return data

def write(data, path):
    lib.Write(data, path, len(data))
```

The `write` function maps very cleanly to the C function call: `ctypes` takes care of automatically handling conversion to C bytes for us, so it's easy to call that function. Note that the `data` argument must be a `bytes` object (not `str`).

The `read` function requires a bit more work: we're getting a pointer returned from the function, so we have to first access its contents to get the full length of the byte array, and then recast that pointer to a pointer to a byte array of the appropriate size. Finally we extract the byte content of that array and return those bytes (dropping the first 8 bytes specifying the length). Fortunately, these operations are transparent to the end user—the `read` function also has the same semantics as the function defined in the Go package.

Here's how to call these functions in Python:

```
x = read("abc")
print(x) # prints "b'read from abc'"

write(b"abc", "123") # prints "Go: wrote to 123"
```

Calling our functions from R

R includes packages like `Rcpp` to make it easy to write functions in C/C++ that can manipulate R structures and make it possible to write highly performant loops. Since we're really interested in passing data back and forth between R and Go, we'll just be using the built-in C FFI (but send us a note if there's a slick way to do this with `Rcpp`!) The way we'll do this is by creating *another* shared object which wraps the Go functions with code that can manipulate R structures directly. This means we'll have two shared object wrappers: the Go-to-C shared object we've already built, and a new C-to-R shared object that links to the first one.

This second shared object we have to implement directly in C. Here's the code:

```
#include <R.h>
#include <Rinternals.h>
#include "libreadwrite.h"
```

```
SEXP Writeit(SEXP data, SEXP path, SEXP length) {
  char *arg1 = (char *) 0;
  char *arg2 = (char *) 0;
  int arg3 = 0;
  arg1 = (char *)(strdup(CHAR(STRING_ELT(data, 0))));
  arg2 = (char *)(strdup(CHAR(STRING_ELT(path, 0))));
  arg3 = INTEGER(length)[0];
  Write(arg1, arg2, arg3);
  free(arg1);
  free(arg2);

  return R_NilValue;
}

SEXP Readit(SEXP path){
  SEXP r_ans = R_NilValue ;
  char *arg1 = (char *) 0;
  char *result = 0 ;
  arg1 = (char *)(strdup(CHAR(STRING_ELT(path, 0))));
  result = (char *)Read(arg1) + 8;
  free(arg1);

  r_ans = result ? Rf_mkString((char *)(result)) : R_NilValue;

  return r_ans;
}
```

Note that we're being a little more careful about explicit memory management for the arguments in each case, and that there are R structures like `SEXP` (meaning "S expression") and `R_NilValue` floating around—these are the structs that R actually can use directly. Also note that in contrast to the slicing we have to do in Python, since we're writing native `C` code here, we can just offset the pointer returned by `Read` by 8 instead of constructing a subslice of the returned byte array.

To build a shared object that R can link with, run the following command:

```
R CMD SHLIB -L. -lreadwrite readwriteR.c
```

This will produce a new shared object, `readwriteR.so`, which we can then load directly in R. Now, to write some R code that calls this, first make sure that the original `libreadwrite.so` produced by Go is in the

library path (e.g. by running `export LD_LIBRARY_PATH=/path/to/...`) Next, we can just use R's built-in `dyn.load` and `.Call` functions:

```
dyn.load("goreadwrite.so")

writeit <- function(data, path, length){
  .Call("Writeit", data, path, length)
}

readit <- function(path){
  return(.Call("Readit", path))
}
```

We can now pass objects directly from R to these functions:

```
writeit("abc", "123", 3L) # prints "Go wrote to 123"
readit("abc") # returns "read from abc"
```

And that's it! While this is something of a toy example, it's a valuable one: with the explosion in data science and machine learning tools and languages, using thin wrappers built around FFIs saves development time and allows re-use of code between languages.

Technology

ModelOp Center

Solutions

What is Model Operations?
For Data Scientists
For IT Professionals
For Business Teams

Contact Us

Contact Us

learn@modelop.com

333 West Wacker Drive
1st Floor
Chicago, IL 60606

Resources

Solutions
Resources
White Papers
Webinars

Company

Our Company

Documentation

Getting Started Guide
CLI