Python from go

Sevki

Sun Feb 9 19:09:28 CES 2014

Abstract

This article is briefly describes how to do cutesy stuff with python over go.

go-python

There is a somewhat in-complete yet absolutely capable go package that lets you do python from go, if you are so inclined to look at indentations … not that there's anything wrong with that.

go-python package is go bindings of `libpython` over `cgo`. The reason I mention cgo is that it means there is a fair bit of c involved in making that package therefore GAE won't like it.

How well does it work?

Color highlighting on this blog is done by pygements, with the go to pygments bindings I wrote called sandman. So if you know what you are doing you could potentially get stuff done.

Downside

It's a lot of work getting something to work, specially if you are not a python expert like me, however it is fairly easier than doing the same thing in c (and if you are doing that you might as well join the python development team) thanks to the work that has been done in the `go-python`.

`libpython` library forces you to implement most of the compiler work, you have to check everything your self and the crashes are anything but graceful because c doesn't fuck around.

For instance if you are the king of person that likes his code to look like psuedo code … not that there's anything wrong with that, you would import

`HtmlFormatter` from the `pygments.formatters` library, and and call it to get the lexer object.

```
from pygments.formatters import HtmlFormatter
```

In it's go counter part (remember this is even more simplified then c) we would use some thing like to do the same thing.

```
GetFormatterByName := getFunction("pygments.formatters", "HtmlFormatter")
```

However that would be cheating since there is a `getFunction` method that is not shown,

```
func getFunction(module_name string, function_name string) *python.PyObject {

    Module := python.PyImport_ImportModule(module_name)
    if Module == nil {
        log.Fatal("Failed to load the "+ module_name+" module")
    }

    var MethodDesired *python.PyObject
    if Module.HasAttrString(function_name) == 1 {
        MethodDesired = Module.GetAttrString(function_name)
    }
    if !MethodDesired.Check_Callable() {
        log.Fatal(module_name+" is not callable")
    }
    return MethodDesired

}
```

You might think that's not bad, `python` compiler does that too, which is exactly the point, you are assuming the responsibilities of the compiler as a deverloper. In the `getFunction` method, you import a module, check if it's loaded, create a `PyObject`, check if the function is present, load the function, check if the function is actually a function (because functions and attributes seem to be PyObjects of module) and finally if nothing catches fire you return the PyObject that is your desired function. That's the ammount of work it takes to do that one simple thing, but you only do it once.

Now that there is something that can be called lets look at how to call it, if you are the kinda person that thinks `egg`, `spam` and `parrot` are clever inside jokes, not that there's anything wrong with that, you might write something like this:

```
formatter = HtmlFormatter(linenos=True, encoding="utf-8")
```

And it's go counterpart would be, actually I couldn't come up with a counterpart for that, `linenos` and `encoding` are members of that class. I'm sure there is a way to call a function like that, but I just ignored those two, since linenos defults to false anyways and came up with something like this

```
FormatterArgs := python.PyTuple_New(0)
Formatter:= GetFormatterByName.CallObject(FormatterArgs)
```

Which actually started causing problems, because pygments does only ascii formatting if you don't actually let it know you want utf-8 like a civilized human being. So whenever I put in something that contained non-ascii it would crash like hindenburg. So I figured out how to set the attributes of an object after init which worked just fine.

```
GetFormatterByName := getFunction("pygments.formatters", "HtmlFormatter")
FormatterArgs := python.PyTuple_New(0)
Formatter:= GetFormatterByName.CallObject(FormatterArgs)

if Formatter == nil {
    log.Fatal("Couldn't get formatter")
}
if Formatter.HasAttrString("encoding") == 0 {
    log.Fatal("Wrong formatter")
}
if Formatter.HasAttrString("linenos") == 0 {
    log.Fatal("Wrong formatter")
}

Formatter.SetAttrString("encoding", python.PyString_FromString("utf-8"))
Formatter.SetAttrString("linenos", python.PyBool_FromLong(lnos))
```

How about calling methods with python objects, well; if you are the kind of person that likes being as far away from the metal as polka ... not that there's anything wrong with that, you might write something like:

```
result = highlight(code, lexer, formatter)
```

And it's go counterpart would be a little different

```
HighlighterArgs := python.PyTuple_New(3)
python.PyTuple_SetItem(HighlighterArgs, 0, python.PyString_FromString(code))
python.PyTuple_SetItem(HighlighterArgs, 1, Lexer)
python.PyTuple_SetItem(HighlighterArgs, 2, Formatter)

highlighted := Highlighter.CallObject(HighlighterArgs)
if highlighted == nil {
    log.Fatal("Couldn't highlight")
}
return python.PyString_AsString(highlighted)
```

We would start of by making 3 tuples, adding the args to the tuple set and calling the function with the tuple set.

Advice... yeah why not?

Python is a very well established language, and there are some amazing libraries written in it that is really not that feasable to port at the moment due to the ammount of time it would take. So something like go-python comes in handy, but you when you are doing similar work besure you st. nick it all the way, check everything twice, makesure things are loaded and available.

For instance, my pygments bindings package is designed to be used along side goeylinguine which relies on github's linguist data, so if you had updated the linguist data after wwdc but you haven't updated pygments and you think I should go and write a blog post explaining how helloworld works in swift, sandman (go-pygments thing) will assume pygments has an object which that doesn't, now unlike django, which will handle crashes gracefully, your go app will crash like the greek economy. This is all because we are actually doing c work, and even though we haven't done any memory management, it doesn't mean it's not there. There are some a bunch of `defer`'ed that handle `free` and `alloc` calls in the go-python library.

libpython makes a lof of assumptions, cheif among them is that you have some idea what you are doing, and you know a little about memory management, go-pyhton abstracts some of this, and does most of the heavylifting for you and it's a great excersize if you are willing to learn about python's internals, how it works.