Testing native binaries using CFFI

Noufal Ibrahim

September 24, 2016

About me

- Founder of Hamon Technologies - Automation/Infrastructure/IoT
- Co-Founder of Pipal academy - Niche trainings.
- Also mentor students via. The Lycæum.
- Long time Python user and been involved with PyCon India since the first conference.
- `@noufalibrahim`

Outline

Coupling between languages

- Web based APIs.

Coupling between languages

- Web based APIs.
- IPC

Coupling between languages

- Web based APIs.
- IPC
- Common runtime

Coupling between languages

- Web based APIs.
- IPC
- Common runtime
- True FFI

An aside - language wars

- Language warriors vs Languages themselves

An aside - language wars

- Language warriors vs Languages themselves
- The real barrier for cooperation is stuck up programmers. Not languages or technologies.

About the talk

- Using Python testing tools to test libraries in lower level languages.
- The cpslib library (C) and its tests (python).
- About ways of interfacing C and Python.

Cpslib

- cpslib is a port of psutil to C.
- Allows you to query process and system information in a cross platform fashion (e.g. number of CPUs etc.)

```
/* cpu_count.c */
#include <inttypes.h>
#include <stdio.h>
#include <stdbool.h>

uint32_t cpu_count(bool);

void main()
{
  uint32_t physical;
  physical = cpu_count(false);
  printf("Physical : %" PRIu32 "\n", physical);
}
```

```
$ gcc -std=gnu11 -L. cpu_count.c -o cpu_count -lpslib
$ ./cpu_count
Physical : 1
```

- The original idea was to make it compatible with psutils.
- So we have something to test against.

Ctypes

- The Python stdlib ffi module.
- Example wrapping

```
cdll = ctypes.CDLL("./libpslib.so")
cdll.cpu_count(True) # 4
cdll.cpu_count(False) # 1
```

- A little more complex with compound types

Leftpad

```
char * left_pad_string(char *ip, size_t ip_count, size_t pad_count);
```

```
$ gcc -fPIC -shared -std=gnu11 -Wall -Wextra -Wunused -o libleftpad.so leftpad.c
```

```
from ctypes import *
leftpad = CDLL("./libleftpad.so")

leftpad.left_pad_string.argtypes = [c_char_p, c_size_t, c_size_t]
leftpad.left_pad_string.restype = c_char_p

def leftpad(ip):
    ip = ip.encode('ascii')
    ret = c_leftpad.left_pad_string(ip, len(ip), 20)
    return ret.decode('ascii')

print ("'%s'"%(leftpad("python"))) # '          python'
```

Ctypes

- Works on pypy, cpython and even jython.
- This is great. Why cffi then?

CFFI

- CFFI has some advantanges

CFFI

- CFFI has some advantanges
- No DSL (`c_size_t` etc.)

CFFI

- CFFI has some advantanges
- No DSL (`c_size_t` etc.)
- Can work at `API` or `ABI` level. The former is more portable.

CFFI

- CFFI has some advantanges
- No DSL (`c_size_t` etc.)
- Can work at `API` or `ABI` level. The former is more portable.
- Apparently faster because of custom code.

CFFI using ABI

- Similar to ctypes

```
# leftpad_cffi_abi.py
import cffi

ffi = cffi.FFI()
ffi.cdef("char *left_pad_string(char *ip, size_t ip_count, size_t pad_count);")

c_leftpad = ffi.dlopen("./libleftpad.so")

def leftpad(ip):
    ip = ip.encode('ascii')
    op = c_leftpad.left_pad_string(ip, len(ip), 20)
    return ffi.string(op).decode('ascii')
```

- We have to guess memory layout and calling conventions here.
- This is hard to get right
- The compiler is what usually does this for us

CFFI Using API

- We need a build script for this

```
# leftpad_cffi_build.py
from cffi import FFI

ffi = FFI()
ffi.set_source('pyleftpad', '',
               libraries=["leftpad"],
               library_dirs=['.'])

ffi.cdef("char *left_pad_string(char *ip, size_t ip_count, size_t pad_count);")

if __name__ == '__main__':
    ffi.compile()
```

- Run this to get a `pyleftpad.so`. A native C extension.

CFFI Using API

- We need a build script for this

```
# leftpad_cffi_build.py
from cffi import FFI

ffi = FFI()
ffi.set_source('pyleftpad', '',
               libraries=["leftpad"],
               library_dirs=['.'])

ffi.cdef("char *left_pad_string(char *ip, size_t ip_count, size_t pad_count);")

if __name__ == '__main__':
    ffi.compile()
```

- Run this to get a `pyleftpad.so`. A native C extension.
- Yup. No more manual C extensions and

CFFI Using API

- We need a build script for this

```
# leftpad_cffi_build.py
from cffi import FFI

ffi = FFI()
ffi.set_source('pyleftpad', '',
               libraries=["leftpad"],
               library_dirs=['.'])

ffi.cdef("char *left_pad_string(char *ip, size_t ip_count, size_t pad_count);")

if __name__ == '__main__':
    ffi.compile()
```

- Run this to get a `pyleftpad.so`. A native C extension.
- Yup. No more manual C extensions and
- The build scripts can generate C extensions compatible with PyPy too.

CFFI Using API

- We need a build script for this

```
# leftpad_cffi_build.py
from cffi import FFI

ffi = FFI()
ffi.set_source('pyleftpad', '',
               libraries=["leftpad"],
               library_dirs=['.'])

ffi.cdef("char *left_pad_string(char *ip, size_t ip_count, size_t pad_count);")

if __name__ == '__main__':
    ffi.compile()
```

- Run this to get a `pyleftpad.so`. A native C extension.
- Yup. No more manual C extensions and
- The build scripts can generate C extensions compatible with PyPy too.
- Though higher level abstractions are usually a good idea.

Using the generated C extension

```
# leftpad_cffi_api.py

import pyleftpad # Loads a native C extension

def leftpad(ip):
    ip = ip.encode('ascii')
    # Don't forget the .lib.
    op = pyleftpad.lib.left_pad_string(ip, len(ip), 20)
    return pyleftpad.ffi.string(op).decode('ascii')
```

Some quick performance numbers

```
# perf.py
import timeit

from leftpad_ctypes import leftpad as ctypes_leftpad
from leftpad_cffi_abi import leftpad as cffi_abi_leftpad
from leftpad_cffi_api import leftpad as cffi_api_leftpad

print ("CFFI API", timeit.timeit(lambda : cffi_api_leftpad("python")))
print ("CFFI ABI", timeit.timeit(lambda : cffi_abi_leftpad("python")))
print ("Ctypes ", timeit.timeit(lambda : ctypes_leftpad("python")))
```

```
CFFI API 2.1375274590009212
CFFI ABI 2.7309077310001157
Ctypes   2.928419708998262 6
```

The general approach

- `set_source` for headers and `cdef` for all declarations
- Build native extension.
- Load it up and use it inside python
- An example test for `left_pad_string` would be

```
# test_leftpad.py
from leftpad_cffi_api import leftpad

def test_leftpad():
    ip = "python"
    assert leftpad(ip) == ip.rjust(20)
```

- Can be run using `py.test`

Wrapping cpslib

```
ffi.set_source("pycpslib",
               """#include <stdio.h>
               #include <stdlib.h>
               #include <sys/types.h>
               #include <unistd.h>
               #include "pslib.h"
               """,
               libraries = ["pslib"],
               library_dirs = [project_root],
               include_dirs = [project_root])

ffi.cdef('''
typedef int32_t pid_t;
typedef int32_t bool;
''')

with lines = open("../pslib.h").readlines()
altered_lines = ['' if line.startswith('#include ') else line for line in lines]
ffi.cdef(''.join(altered_lines))

if __name__ == '__main__':
    ffi.compile()
```

Testing cpslib

```
import psutil
from pycpslib import lib as P

def test_logical_cpu_count(flush):
    assert P.cpu_count(1) == psutil.cpu_count(True)

def test_physical_cpu_count(flush):
    assert P.cpu_count(0) == psutil.cpu_count(False)
```

- Useful to prevent regressions.
- For feature parity.
- To verify functionality on new kernels/platforms.

Test coverage

- `gcov` allows us to measure coverage of C files.
- You compile with a few extra flags

```
gcc -fprofile-arcs -ftest-coverage -o leftpad leftpad.c
```

- Compiling it will produce a `.gcno` file (the call graph)

Test coverage

- `gcov` allows us to measure coverage of C files.
- You compile with a few extra flags

```
gcc -fprofile-arcs -ftest-coverage -o leftpad leftpad.c
```

- Compiling it will produce a `.gcno` file (the call graph)
- Then run it `./leftpad`
- You'll get a `.gcda` file (the actual data)

Test coverage

- `gcov` allows us to measure coverage of C files.
- You compile with a few extra flags

```
gcc -fprofile-arcs -ftest-coverage -o leftpad leftpad.c
```

- Compiling it will produce a `.gcno` file (the call graph)
- Then run it `./leftpad`
- You'll get a `.gcda` file (the actual data)
- Then run `gcov leftpad` (human readable output)

Test coverage

- `gcov` allows us to measure coverage of C files.
- You compile with a few extra flags

```
gcc -fprofile-arcs -ftest-coverage -o leftpad leftpad.c
```

- Compiling it will produce a `.gcno` file (the call graph)
- Then run it `./leftpad`
- You'll get a `.gcda` file (the actual data)
- Then run `gcov leftpad` (human readable output)
- And you'll finally get coverage data in `leftpad.c.gcov`

pytest-gcov

- This is a simple `py.test` plugin.
- It will automatically do all this for you and print coverage statistics at the end.
- Lots of limitations but "works for me".

Thanks

- noufal@nibrahim.net.in
- @noufalibrahim
- github.com/nibrahim