Programming language design and analysis

Interfacing languages

Marius Minea

9 January 2017

Foreign Function Interface

take advantage of different language features

code reuse (libraries)

efficiency (of libraries)

increase acceptance by providing *bindings* to other languages

Issues to consider

function call mechanism (parameter passing)

storage layout of objects

naming conventions for external function symbols

memory management (garbage collection)

exception handling

API vs ABI

Application Binary Interface
= machine-level interface between program modules

Covers:

- size and alignment of data types
- calling convention
- how system calls are made
- function name mangling (for overloading, e.g. C++)

Calling conventions

`cdecl`:

caller cleans up stack
args passed right to left
regs eax, ecx, edx are caller-saved, rest: callee-saved
result returned in eax

typical for Linux/GCC

`stdcall`:

callee cleans up stack (must know arg count)

typical for MS Win32 API

Calling C from C++

simplest: just declare function as `extern "C"` ...

ensures function name is not mangled as in C++
(symbol name is just function name)

Calling C from Python: ctypes

Many Python libraries are written in C, so interfacing is natural

Python's `ctypes` module can:
  load C functions on the fly from shared libraries (DLLs)
  translate simple data types between C and Python

```
import ctypes
libc = ctypes.CDLL( '/lib/libc.so.6' )
t = libc.time(None)   # call C function, None = NULL
print t               # use result in Python
```

code: ctypes tutorial + Wikipedia

types corresponding to C: `c_int`, `c_char_p`, etc
  and corresponding values (`None` for `NULL`)

access to representation: `.raw` vs `.value` for strings

Python bytes objects are immutable ⇒ `create_string_buffer()` to call C functions which expect mutable memory

Calling C from C#: P/Invoke

Platform Invocation Services

Two options, depending on availability of library source code (and need to marshal function arguments)

Implicit PInvoke (C++ Interop)
usable if parameter types have same representation in managed and unmanaged memory — no conversion required
better efficiency and type safety

Explicit PInvoke
`DllImportAttribute` placed before function decl
can specify type of marshaling needed
creates managed entry point with needed *thunk* (transition code) and simple data conversions

One more option: IJW (It Just Works)
no `DLLImport` declarations but explicit marshalling code

Java to native: three options

JNI: Java Native Interface
historically first

JNA: Java Native Access
community-developed, simpler, no boilerplate/glue code

JNR: Java Native Runtime
current JEP (Enhancement Proposal), good performance

JNI: Java Native Interface

Native function is written with two extra arguments:

a `JNIEnv` pointer for interface to the JVM
  with lots of functions to interact with the JVM
  e.g. convert arrays and strings a `jobject` reference to the current

object (of the class where the native method is declared)

JNI Pitfalls

Triggering array copies: arrays are passed as opaque handles;
should use callbacks into JVM to get/set elements

Reaching back instead of passing arguments
  usual style: pass object, access fields
  here: each object access must reach (crosss) back into JVM

Native code must check for exceptions on JNI calls

Local references created have lifetime until native code completion

Memory leaks: global references created and not garbage collected

JNA: Java Native Access

simplified, no generated headers or wrappers for native code
pure Java implementation, based on `libffi` library
(library to interface with various calling conventions, calling any function based on a call interface description)

but: does not support C++
slower (data accesses in Java; copies b/w C and Java; cost of calls since type information determined at runtime, not statically)
Java code following C data may be layout-dependent and ugly

JNR :Java Native Runtime

aims to overcome the cumbersome parts and portability issues of JNI, and the performance problems of JNA

also based on libffi, with several levels in between

wide coverage of native functions (POSIX, etc.)

proposed basis for a standard Java FFI