Interpreted languages - leveraging the compiled language behind the interpreter

Asked 16 years, 4 months ago Modified 6 years, 6 months ago

Viewed 1k times

5

If there are any language designers out there (or people simply in the know), I'm curious about the methodology behind creating standard libraries for interpreted languages. Specifically, what seems to be the best approach? Defining standard functions/methods in the interpreted language, or performing the processing of those calls in the compiled language in which the interpreter is written?

What got me to thinking about this was the SO question about a stripslashes()-like function in Python. My first thought was "why not define your own and just call it when you need it", but it raised the question: is it preferable, for such a function, to let the interpreted language handle that overhead, or would it be better to write an extension and leverage the compiled language behind the interpreter?

performance language-agnostic language-features interpreted-language

Share

Improve this question

Follow

edited Jun 19, 2018 at 16:23

rene
42.3k ● 78 ● 119 ● 163

asked Aug 17, 2008 at 11:12

Brian Warshaw
23k ● 9 ● 54 ● 72

3 Rather than "leverage" the compiled language, why not "use" it or "take advantage" of it? Let's not increase the number of language keywords in English unless we have to :) – MarkJ Jan 28, 2009 at 12:57

4 Answers

Sorted by: Highest score (default)

6

The line between "interpreted" and "compiled" languages is really fuzzy these days. For example, the first thing Python does when it sees source code is compile it into a bytecode representation, essentially the same as what Java does when compiling class files. This is what *.pyc files contain. Then, the python runtime executes the bytecode without referring to the original source. Traditionally, a purely interpreted language would refer to the source code continuously when executing the program.

When building a language, it is a good approach to build a solid foundation on which you can implement the higher level functions. If you've got a solid, fast string handling system, then the language designer can (and should) implement something like stripslashes() outside the base runtime. This is done for at least a few reasons:

- The language designer can show that the language is flexible enough to handle that kind of task.
- The language designer actually writes real code in the language, which has tests and therefore shows that the foundation is solid.
- Other people can more easily read, borrow, and even change the higher level function without having to be able to build or even understand the language core.

Just because a language like Python compiles to bytecode and executes that doesn't mean it is slow. There's no reason why somebody couldn't write a Just-In-Time (JIT) compiler for Python, along the lines of what Java and .NET already do, to further increase the performance. In fact, IronPython compiles Python directly to .NET bytecode, which is then run using the .NET system including the JIT.

To answer your question directly, the only time a language designer would implement a function in the language behind the runtime (eg. C in the case of Python) would be to maximise the performance of that function. This is why modules such as the regular expression parser are written in C rather than native Python. On the other hand, a module like getopt.py is implemented in pure Python because it can all be done there and there's no benefit to using the corresponding C library.

Share Improve this answer Follow

answered Aug 17, 2008 at 12:39

Greg Hewgill
990k ● 191 ● 1.2k ● 1.3k

3

There's also an increasing trend of reimplementing languages that are traditionally considered "interpreted" onto a platform like the JVM or CLR -- and then allowing easy access to "native" code for interoperability. So from Jython and JRuby, you can easily access Java code, and from IronPython and IronRuby, you can easily access .NET code.

In cases like these, the ability to "leverage the compiled language behind the interpreter" could be described as the primary motivator for the new implementation.

Share Improve this answer Follow

answered Aug 17, 2008 at 13:33

Curt Hagenlocher
20.9k ● 8 ● 62 ● 50

2

See the 'Papers' section at www.lua.org.

Especially [The Implementation of Lua 5.0]()

Lua defines all standard functions in the underlying (ANSI C) code. I believe this is mostly for performance reasons. Recently, i.e. the 'string.*' functions got an alternative implementation in pure Lua, which may prove vital for subprojects where Lua is run on top of .NET or Java runtime (where C code cannot be used).

Share Improve this answer

Follow

answered Sep 17, 2008 at 19:44

akauppi

18k ● 15 ● 102 ● 124

1

As long as you are using a portable API for the compiled code base like the ANSI C standard library or STL in C++, then taking advantage of those functions would keep you from reinventing the wheel and likely provide a smaller, faster interpreter. Lua takes this approach and it is definitely small and fast as compared to many others.

Share Improve this answer

Follow

answered Aug 17, 2008 at 13:41

Judge Maygarden

27.5k ● 9 ● 83 ● 100