Extending Ruby with C

by Garrett Rooney
11/18/2004

Ruby, if you've never heard of it, is an object-oriented scripting language, simil to Perl and Python. It originates from Japan and is young, as far as programming There are many really good reasons you might want to use the Ruby language; I go into all of them here, but the one at the core of this article is the ease with wl write Ruby extensions in C.

I'm a big fan of the so-called agile programming languages. I think they have a huge advantage over more traditional languages like C and C++. They also have some drawbacks, among the largest being that there's an awful lot of existing code written in C and C++. It's hard to sell people on moving to something new if they have to leave all their old toys behind.

The standard response to these sort of arguments is that you can easily write an extension that bridges the gap between your old C code and your new Perl or Python, or whatever agile language is hot this week, code. Unfortunately, I've generally found that the APIs for bridging t Perl and C are either cryptic (XS) or fragile (Inline::C). While Python is better i still find its C API rather difficult to read. Tools such as SWIG can help alleviat but you still need to write a bunch of glue code to bridge the gap between the hi languages and the low-level C code.

When I first looked at doing the same kind of thing for Ruby, a whole new worl APIs are simple, to the point where I was up and running in minutes rather than need to know to start is in the `README.EXT` file in the top level of the Ruby so need help with something that isn't documented there, you can't ask for a clearer the Ruby source code itself. In short, I was just aching for a test case, some C c up in a Ruby extension to prove how simple it is to make something that's easy test case, I chose the GenX library.

GenX

GenX is a simple C library for generating correct, canonical XML. It verifies that its data is valid UTF-8 and the structure of the XML document being generated is valid, and it forces you to use canonical XML--that may not mean much to you right now, but it can be significant if you need to compare two XML documents to determine their equivalence. Tim Bray wrote GenX and hosts it at GenxStatus. GenX originally attracted me because it provides a way to avoid problems related to invalid XML, which I've encountered in my own work. In addition to its usefulness, it's also perfect for an example of how to embed a C library in Ruby because it's very small, self contained, and has a well-defined API we can wrap up in Ruby without too much trouble.

Justification

At this point it's worth asking whether a Ruby extension is really the best way to make this kind of functionality available.

Using an extension means that users need to install a binary distribution precom versions of Ruby and their operating systems, or to build it themselves, which m access to a C compiler. Additionally, extending Ruby via C has its own set of d screw something up in a standard Ruby module, pretty much the worst you can exception to be thrown. It's possible for users to recover from this if they're para about catching exceptions and structure their code correctly. In a C-based exten corrupt memory or cause a segmentation fault or any number of other problems recovery is difficult, and all of which have the chance to crash the underlying R

That said, in this particular case I think providing direct access to the underlyin via a C extension is the way to go. The GenX library is available right now, it w its job in a very efficient manner. There's no reason to duplicate functionality un if I did rewrite this in pure Ruby, all I am likely to accomplish is slowing things GenX is exceptionally self contained; while using the library does require that u precompiled extension or possess a C compiler, it at least doesn't bring in any ot requirements. Finally, the GenX API is quite straightforward. It's reasonable to be able to implement this extension without undue risk of crashing our Ruby int bugs in our code.

Some Basic Functionality

The first step in writing a Ruby extension is to create something that compiles a means writing an *extconf.rb* file that tells Ruby how to compile and link your ex writing the bare-bones C file that makes up the extension. With these two steps have a Ruby module that you can `require` and a new class you can instantiate useful one because it won't actually have any methods.

The *extconf.rb* file is a short Ruby program that makes use of the `mkmf` module makefile, which you use to build your extension. There's a fair amount of specia

functionality you can put in your *extconf.rb* file, but for our purposes the bare m
Here's the entirety of my *extconf.rb* file:

```
require 'mkmf'

dir_config("genx4r")

create_makefile("genx4r")
```

This tells Ruby to use all of the *.c* files in the current working directory to build
named *genx4r*, and that it should write out a makefile to compile and link it. If y
and *.h* files from the GenX tarball into the current directory, you can run `ruby
&& make` and then have a Ruby extension sitting there just waiting for you to r
our script. Here's the process:

```
$ ruby extconf.rb
creating Makefile
$ make
gcc -fno-common   -g -Os -pipe -no-cpp-precomp -fno-common -DHAVI
-pipe -pipe  -I. -I/usr/lib/ruby/1.6/powerpc-darwin7.0 -I.    -c
charProps.c
gcc -fno-common   -g -Os -pipe -no-cpp-precomp -fno-common -DHAVI
-pipe -pipe  -I. -I/usr/lib/ruby/1.6/powerpc-darwin7.0 -I.    -c
genx.c
cc -fno-common   -g -Os -pipe -no-cpp-precomp -fno-common -DHAVE_
-pipe -pipe  -dynamic -bundle -undefined suppress -flat_namespace
-L/usr/lib/ruby/1.6/powerpc-darwin7.0 -L/usr/lib  -o genx4r.bundl
genx.o   -ldl -lobjc
$ ls
Makefile        charProps.o     genx.c          genx.o
charProps.c     extconf.rb      genx.h          genx4r.bundle*
$ irb
irb(main):001:0> require 'genx4r'
LoadError: Failed to lookup Init function ./genx4r.bundle
        from (irb):1:in `require'
        from (irb):1
irb(main):002:0>
```

OK, so that sort of works.... There's a Ruby extension, but trying to `require` it
Ruby only produces an error. That's because none of the *.c* files defined an `Ini`
Ruby tries to load an extension, the first thing it does is look for a function nam
`Init_extname`, where `extname` is the name of the extension. Because that
exist, Ruby obviously can't find it and throws a `LoadError` exception.

The next step is to implement `Init_genx4r` to allow the extension to load su
bare minimum necessary is simply an empty function named `Init_genx4r` t
arguments and returns nothing. I like that. Here are the current contents of the *g*

```
#include "ruby.h"

void
Init_genx4r()
{
  /* nothing here yet */
}
```

Rerun `extconf.rb` and `make`. When you try to load the `genx4r` module wi
should have better results:

```
$ irb
irb(main):001:0> require 'genx4r'
=> true
irb(main):002:0>
```

Creating a Class

The extension loads, but it still doesn't actually do anything. It needs definitions that make up the interface to the GenX library. For now, I'll define one top-leve named `GenX` and a single class, `Writer`, that lives in it. That class is simply a around the C-level `genxWriter` type. Here's the next iteration of *genx4r.c*:

```
#include "ruby.h"

#include "genx.h"

static VALUE rb_mGenX;
static VALUE rb_cGenXWriter;

static void
writer_mark (genxWriter w)
{}

static void
writer_free (genxWriter w)
{
  genxDispose (w);
}

static VALUE
writer_allocate (VALUE klass)
{
  genxWriter writer = genxNew (NULL, NULL, NULL);

  return Data_Wrap_Struct (klass, writer_mark, writer_free, write
}

void
Init_genx4r ()
{
  rb_mGenX = rb_define_module ("GenX");

  rb_cGenXWriter = rb_define_class_under (rb_mGenX, "Writer", rb_

  /* NOTE: this only works in ruby 1.8.x.  for ruby 1.6.x you ins
   *       a 'new' method, which does much the same thing as this
  rb_define_alloc_func (rb_cGenXWriter, writer_allocate);
}
```

That's a lot of new code. First comes the `#include` of `genx.h`, because it ne functions defined by GenX. The two `VALUE` variables represent the module an object in Ruby (and remember, *everything* in Ruby is an object) has a `VALUE`; t reference to the object. The beginning of `Init_genx4r` initializes these varia `rb_define_module` to create the `GenX` module and `rb_define_class_` the `Writer` class.

Next, Ruby needs to know how to allocate the guts of the `GenX::Writer` obj `allocate` comes in, using `rb_define_alloc_func` to associate the `writer_allocate` function with the `allocate` method. `writer_alloc`

genxWriter object with genxNew and turns it into a Ruby object via the Data_Wrap_Struct macro. Data_Wrap_Struct simply takes a VALUE class of the new object (passed as an argument to writer_allocate), two f used for Ruby's mark-and-sweep garbage collection, and a pointer to the underl structure--in this case the genxWriter itself--and returns a new VALUE that r object, which is simply a thin wrapper around the C-level pointer. Finally, the c function for the object, writer_mark, which actually does nothing, and the d writer_free, which calls genxDispose to clean up the genxWriter al writer_allocate. When wrapping a more complicated structure that inclu other Ruby-level objects, the mark function must call rb_gc_mark on each o Ruby when nothing references them any longer and they are ready for garbage

One thing to note about *genx4r.c* is that the only function accessible outside tha Init_genx4r. Everything else is static, which means that it won't leak ou namespace and cause linkage errors if some other part of the program happens t function or variable name.

It's time to take a quick jaunt through irb to confirm that it's possible to create new class:

```
$ irb
irb(main):001:0> require 'genx4r'
=> true
irb(main):002:0> w = GenX::Writer.new
=> #<GenX::Writer:0x321f84>
irb(main):003:0>
```

Sure enough, it's an instance of our new GenX::Writer class. Adding a few make it actually useful!

Pages: **1**, 2, 3

size and speed, and making the most of C++ without a performance penalty.