Table of Contents

Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly Media, Inc.
Print Publication Date: 2006/07/19

22. Extending Ruby with Other Languages

When you decide to use an interpreted language such as Ruby, you're trading raw speed for ease of use. It's far easier to develop a program in a higher-level language, and you get a working program faster, but you sacrifice some of the speed you might get by writing the program in a lower-level language like C and C++.

That's the simplified view. Anyone who's spent any serious amount of time working with higher-level languages knows that the truth is usually more complex. In many situations, the tradeoff doesn't really matter: if the program is only going to be run once, who cares if it takes twice as long to do its job? If a program is complex enough, it might be prohibitively hard to implement in a low-level language: you might never actually get it working right without using a language like Ruby.

But even Ruby zealots must admit that there are still situations where it's useful to be able to call code written in another language. Maybe you need a particular part of your program to run blazingly fast, or maybe you want to use a particular library that's implemented in C or Java. When that happens you'll be grateful for Ruby's extension mechanism, which lets you call C code from a regular Ruby program; and the JRuby interpreter, which runs atop the Java Virtual Machine and uses Java classes as though they were Ruby classes.

Compared to other dynamic languages, it's pretty easy to write C extensions in Ruby. The interfaces you need to understand are easy to use and clearly defined in just a few header files, there are numerous examples available in the Ruby standard library itself, and there are even tools that can help you access C libraries without writing any C code at all.

So let's break out that trusty C compiler and learn how to drop down under the hood of the Ruby interpreter, because you just never know when your next program will to turn into one of those situations where a little bit of C code is the only solution to the problem.

—Garrett Rooney

Recipe 22.1. Writing a C Extension for Ruby

Credit: Garrett Rooney

Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly Media, Inc.
Print Publication Date: 2006/07/19

Problem

You want to implement part of your Ruby program in C. This might be the part of your program that needs to run really fast, it might contain some very platformspecific code, or you might just have a C implementation already, and you don't want to also write one in Ruby.

Solution

Write a C extension that implements that portion of your program. Compile it with `extconf.rb` and `require` it in your Ruby program as though it were a Ruby library. You'll need to have the Ruby header files installed on your system.

Here's a simple Ruby program that requires a library called `example`. It instantiates an instance of `Example::Class` from that library, and calls a method on that library:

```
require 'example'
e = Example::Class.new
e.print_string("Hello World\n")
# Hello World
```

What would the `example` library look like if it were written in Ruby? Something like this:

```
# example.rb
module Example
  class Class
    def print_string(s)
      print s
    end
  end
end
```

Let's implement that same functionality in C code. This small C library, `example.c`, defines a Ruby module, class, and method using the functions made available by `ruby.h`:

```
#include <ruby.h>
#include <stdio.h>

static VALUE rb_mExample;
static VALUE rb_cClass;

static VALUE
print_string(VALUE class, VALUE arg)
{
  printf("%s", RSTRING(arg)->ptr);
  return Qnil;
}
void
Init_example()
{
  rb_mExample = rb_define_module("Example");

  rb_cClass = rb_define_class_under(rb_mExample, "Class", rb_cObject);
```


Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly Media, Inc.
Print Publication Date: 2006/07/19

```
  rb_define_method(rb_cClass, "print_string", print_string, 1);
}
```

To build the extension, you also need to create an `extconf.rb` file:

```
# extconf.rb
require 'mkmf'

dir_config('example')
create_makefile('example')
```

Then you can build your library by running `extconf.rb`, then make:

```
$ ls
example.c extconf.rb

$ ruby extconf.rb
creating Makefile

$ make
gcc -fPIC -Wall -g -O2 -fPIC -I. -I/usr/lib/ruby/1.8/i486-linux
  -I/usr/lib/ruby/1.8/i486-linux -I. -c example

gcc -shared -L"/usr/lib" -o example.so example.o -lruby1.8
  -lpthread -ldl -lcrypt -lm -lc

$ ls
Makefile example.c example.o example.so extconf.rb
```

The `example.so` file contains your extension. As long as it's in your Ruby include path (and there's no `example.rb` that might mask it), you can use it like any other Ruby library:

```
require 'example'
e = Example::Class.new
e.print_string("Hello World\n")
# Hello World
```

Discussion

Most programs can be implemented using plain old Ruby code, but occasionally it turns out that it's better to implement part of the program in C. The example library above simply provides an interface to C's `printf` function, and Ruby already has a perfectly good `IO#printf` method.

Perhaps you need to perform a calculation hundreds of thousands of times, and implementing it in Ruby would be too slow (the `Example::Class#print_string` method is faster than `IO#printf`). Or maybe you need to interact with some platformspecific API that's not exposed by the Ruby standard library. There are a number of reasons you might want to fall back to C code, so Ruby provides you with a reasonably simple way of doing it.

Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly Media, Inc.
Print Publication Date: 2006/07/19

Unfortunately, the fact that it's easy doesn't always mean it's a good idea. You must remember that when writing C-level code, you're playing with fire. The Ruby interpreter does its best to limit the damage you can do if you write bad Ruby code. About the worst you can do is cause an exception: another part of your program can catch the exception, handle it, and carry on. But C code runs outside the Ruby interpreter, and an error in C code can crash the Ruby interpreter.

With that in mind, let's go over some of the details you need to know to write a C extension.

A Ruby extension is just a small, dynamically loadable library, which the Ruby interpreter loads via `dlopen` or something similar. The entry point to your extension is via its Init function. For our `example` module, we defined an `Init_example` function to set everything up. `Init_example` is the first function to be called by the Ruby interpreter when it loads our extension.

The `Init_example` function uses a number of functions provided by the Ruby interpreter to declare modules, classes, and methods, just as you might in Ruby code. The difference, of course, is that here the methods are implemented in C. In this example, we used `rb_define_module` to create the `Example` module, then `rb_define_class_under` to define the `Example::Class` class (which inherits from `Object`), and `finally rb_define_ method` to give `Example::Class a print_string` method.

The first thing to notice in the C code is all the `VALUE` variables lying around. A `VALUE` is the C equivalent of a Ruby reference, and it can point to any Ruby object. Ruby provides you with a number of functions and macros for manipulating `VALUE`s.

The `rb_cObject` variable is a `VALUE`, a reference to Ruby's `Object` class. When we pass it into `rb_define_class_under`, we're telling the Ruby interpreter to define a new subclass of `Object`. The `ruby.h` header file defines similar variables for many other Rubylevel modules (named using the `rb_mFoo` convention) and classes (the convention is `rb_cFoo`).

To manipulate a `VALUE`, you need to know something about it. It makes no more sense in C code than in Ruby code to call a method of `File` on a value that refers to a string. The simplest way to check a Ruby object's type is to use the `Check_Type` macro, which lets you see whether or not a `VALUE` points to an instance of a particular Ruby class. For convenience, the `ruby.h` file defines constants `T_STRING, T_ARRAY`, and so on, to denote built-in Ruby classes.

But that's not what we'd do in Ruby code. Ruby enforces duck typing, in which objects are judged on the methods they respond to, rather than the class they instantiate. C code can operate on Ruby objects the same way. To check whether an object responds to a particular

Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly Media, Inc.
Print Publication Date: 2006/07/19

message, use the function `rb_respond_to`. To send the message, use `rb_funcall`. It looks like this:

```
static VALUE
write_string(VALUE object, VALUE str)
{
  if (rb_respond_to(object, rb_intern("<<")))
  {
     rb_funcall(object, rb_intern("<<"), 1, str);
  }
  return Qnil;
}
```

That's the C-level equivalent of the following Ruby code:

```
def write_string(object, str)
  object << str if object.respond_to?('<<')
  return nil
end
```

A few more miscellaneous tips: the `rb_intern` function takes a symbol name as a C string and returns the corresponding Ruby symbol ID. You use this with functions like `rb_respond_to` and `rb_funcall` to refer to a Ruby method. `Qnil` is just the C-level name for Ruby's special `nil` object. There are a few similar constants, like `Qfalse` and `Qtrue`, which do just about what you'd think they'd do.

There are a number of other C level functions that let you create and manipulate strings (look in for functions that start with `rb_str`), arrays (`rb_ary`), and hashes (`rb_ hash`). These APIs are pretty self-explanatory, so we won't go into them in depth here, but you can find them in the Ruby header files, specifically `ruby.h` and `intern.h`.

Ruby also defines some macros to do convenient things with common data types. For example, the `StringValuePtr` macro takes a VALUE that refers to a ruby `String` and returns a C-style char pointer. This can be useful for interacting with C-level APIs. You can find this and other similar helpers in the `ruby.h` header.

See Also

- The file README.EXT file in the Ruby source tree
- Recipe 22.2, "Using a C Library from Ruby"

Recipe 22.2. Using a C Library from Ruby

Credit: Garrett Rooney

Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly Media, Inc.
Print Publication Date: 2006/07/19

Problem

You'd like to use a library in your Ruby program, but the library's implemented in C and there are no bindings.

Solution

Write a Ruby extension that wraps the C library with Ruby classes and methods.

Let's say we want to give a Ruby interface to C's file methods (yes, the `File` class already does this, but this makes a good example). We want to make it possible to open a disk file and read from it a byte at a time.

Just as in Recipe 22.1, you'll need a C file that implements the actual extension. This one is called `stdio.c`. It's got an `Init_stdio` function that defines a Ruby module (`Stdio`), a Ruby class (`Stdio::File`), and some methods for that class.

The `file_allocate` function corresponds to the `Stdio::File` constructor. Because it's a constructor, we must also define some hook functions to create and destroy the underlying resources (in this case, a filehandle and the memory it uses):

```
#include "stdio.h"
#include "ruby.h"

static VALUE rb_mStdio;
static VALUE rb_cStdioFile;

struct file
{
  FILE *fhandle;
};

static VALUE
file_allocate(VALUE klass)
{
  struct file *f = malloc(sizeof(*f));
  f->fhandle = NULL;
  return Data_Wrap_Struct(klass, file_mark, file_free, f);
}

static void
file_mark(struct file *f)
{
}

static void
file_free(struct file *f)
{
  fclose(f->fhandle);
  free(f);
}
```

The `file_open` function implements the `Stdio::File#open` method:

```
static VALUE
file_open(VALUE object, VALUE fname)
```


Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly Media, Inc.
Print Publication Date: 2006/07/19

```
{
  struct file *f;
  Data_Get_Struct(object, struct file, f);
  f->fhandle = fopen(RSTRING(fname)->ptr, "r");
  return Qnil;
}
```

`file_readbyte` implements the `Stdio::File#readbyte` method:

```
static VALUE
file_readbyte(VALUE object)
{
  char buffer[2] = { 0, 0 };
  struct file *f;

  Data_Get_Struct(object, struct file, f);

  if (! f->fhandle)
    rb_raise(rb_eRuntimeError, "Attempt to read from closed file");

  fread(buffer, 1, 1, f->fhandle);

  return rb_str_new2(buffer);
}
```

Finally, our `Init_` method defines the `Stdio` module, the `File` class, and the three methods defined for the `File` class:

```
void
Init_stdio()
{
  rb_mStdio = rb_define_module("Stdio");
  rb_cStdioFile = rb_define_class_under(rb_mStdio, "File", rb_cObject);

  rb_define_alloc_func(rb_cStdioFile, file_allocate);
  rb_define_method(rb_cStdioFile, "open", file_open, 1);
  rb_define_method(rb_cStdioFile, "readbyte", file_readbyte, 0);
}
```

As before, you'll need an `extconf.rb` file that knows how to compile your C library:

```
# extconf.rb
require 'mkmf'
dir_config("stdio")
create_makefile("stdio")
```

Once the C library is compiled, you can use it from Ruby as though it were a Ruby library:

```
open('foo.txt', 'w') { |f| f << 'foo' }

require 'stdio'
f = Stdio::File.new
f.open('foo.txt')
f.readbyte                                          # => "f"
f.readbyte                                          # => "o"
f.readbyte                                          # => "o"
```


Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly Media, Inc.
Print Publication Date: 2006/07/19

Discussion

The basic idea when writing a Ruby extension is to create a C data structure and wrap it in a Ruby object. The C data structure gives you someplace to store whatever data you need, so you can access it in your C methods. You're creating a primitive form of object-oriented programming in C.

Ruby provides some macros to help with this. `Data_Wrap_Struct` wraps a C data structure in a Ruby object. It takes a pointer to your data structure, along with a few pointers to callback functions, and returns a VALUE. The `Data_Get_Struct` macro takes that VALUE and gives you back a pointer to your C data structure.

You usually use `Data_Wrap_Struct` inside your class's `allocate` function (called by the constructor), and `Data_Get_Struct` inside its instance methods. In the example above, the `file_allocate` function creates a C struct (containing a variable of type `FILE`) and passes it into `Data_Wrap_Struct` to get a VALUE. The functions for the instance methods, `file_open` and `file_readbyte`, both take a VALUE as an argument, and pass it into `Data_Get_Struct` to get a C struct.

So what about those callback functions? There are three of them: an "allocate" function, a "mark" function, and a "free" function. The "allocate" function is called whenever an object is created. The other two have to do with garbage collection.

Ruby's garbage collector uses a mark-and-sweep algorithm: it runs through all the "live" objects in the system, marking them to note that it was able to reach them. Then it destroys every object that it couldn't reach: by definition, those objects are no longer in use, and don't need to be kept around in memory. To make this work, you need to provide two callbacks: one that marks an object as reachable, and one that frees the underlying resources for all unreachable objects.

In this case, both functions are simple. The "free" callback simply closes the filehandle and calls the `C free` function. The "mark" callback doesn't need to do anything, since this object doesn't refer to any other Ruby objects.

If your object does contain references to other Ruby objects, all you need to do is explicitly mark them (by calling the `rb_gc_mark` function) in your "mark" callback. This example goes a bit further than it needs to by defining an empty mark callback; it could accomplish the same thing by passing in a NULL function pointer.

To summarize: if your library doesn't define its own data structures, define your own C struct. Implement methods that translate Ruby arguments into their C equivalents, call

Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly Media, Inc.
Print Publication Date: 2006/07/19

the library functions you're interested in, then translate the return values back into Ruby data structures, so that the rest of the Ruby program can use it.

See Also

- The README.EXT file in the Ruby source tree
- Recipe 22.1, "Writing a C Extension for Ruby"
- Recipe 22.3, "Calling a C Library Through SWIG," might do what you want with less complication

Recipe 22.3. Calling a C Library Through SWIG

Credit: Garrett Rooney

Problem

You want to use a C library in your Ruby code, but you don't want to have to write any C code to do it.

Solution

Use SWIG to generate the C extension for you. SWIG is a programming tool that takes as its input a file containing the information about C functions. It produces source code that lets you access those C functions from a variety of programming languages, including Ruby.

All you you need to write is an interface file, containing the prototypes for the C functions you want to call. The interface file also contains a few directives to control things like the name of the resulting module. Process that file with the `swig` command-line tool, build your extension, and you're up and running.

Let's build a SWIG extension that lets Ruby access functions from the standard C library. It'll provide access to enough functionality that you can read data from one file and write it to another. In Recipe 22.1, we wrote the C code for a similar extension ourselves, but here we'll let SWIG do it.

First we'll need a SWIG interface file, `libc.i`:

```
%module libc

FILE *fopen(const char *, const char *);

int fread(void *, size_t, size_t, FILE *);
int fwrite(void *, size_t, size_t, FILE *);
int fclose(FILE *);

void *malloc(size_t);
```


Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly Media, Inc.
Print Publication Date: 2006/07/19

This file specifies the name of our extension as "libc". For SWIG Ruby extensions, this means the extension will be named "libc", and the code will be contained in a Ruby module claled `Libc`. This file also provides the prototypes for the functions we're going to want to call.

You'll also need an `extconf.rb` program, similar to the one we used in the previous two recipes:

```
# extconf.rb
require 'mkmf'
dir_config('tcl')
dir_config('libc')
create_makefile('libc')
```

To generate the C extension, we process the header file with the `swig` command-line tool. We then run Ruby's `extconf.rb` program to generate a makefile, and run `make` to compile the extension:

```
$ swig -ruby libc.i
$ ls
extconf.rb libc.i libc_wrap.c

$ ruby extconf.rb --with-tcl-include=/usr/include/tcl8.4
creating Makefile

$ make
…

$ ls
Makefile extconf.rb libc.i libc.so libc_wrap.c libc_wrap.o
```

Once the module is compiled, we can use it just like any other Ruby extension. This code uses a Ruby interface to prepopulate a file with random data, then uses the C interface to copy the contents of that file to another file:

```
random_data = ""
10000.times { random_data << rand(255) }
open('source.txt', 'w') { |f| f << random_data }

require 'libc'
f1 = Libc.fopen('source.txt', 'r')
f2 = Libc.fopen('dest.txt', 'w+')

buffer = Libc.malloc(1024)

nread = Libc.fread(buffer, 1, 1024, f1)

while nread > 0
 Libc.fwrite(buffer, 1, nread, f2)
 nread = Libc.fread(buffer, 1, 1024, f1)
end
Libc.fclose(f1)
Libc.fclose(f2)

# dest.txt now contains the same random data as source.txt.
random_data == open('dest.txt') { |f| f.read }
# => true
```


Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly Media, Inc.
Print Publication Date: 2006/07/19

There you have it: without writing a line of C code, we've been able to call into a C library from Ruby.

Discussion

The great advantage of SWIG over writing your own interface to a C library is that you don't have to write your own interface to a C library. The disadvantage is that you get the exact same interface (or a subset) as the C library. The `Libc` module exposes a Ruby module that's nothing more than a collection of C functions. If you want a friendlier interface, you need to write it yourself on top of the SWIGgenerated module.

In addition to the actual function prototypes, the interface file needs to have a little metadata about your extension. At the minimum, you'll need a `%module` line that tells SWIG what to call the extension it generates. Depending on your C code, you might also need to tell SWIG how to handle C constructs that don't map directly to Ruby; see the SWIG documentation on `%typemap` for details.

There are two main ways to create an interface file. The simplest way is simply to copy the prototypes for your C functions right from your header file into your SWIG interface file. Alternatively, you can use the `%import` filename directive to include a C header file in a SWIG interface file.

One more thing: note the references to `tcl` in the `extconf.rb` file and in the commandline invocation of `extconf.rb`. Our `Libc` module has nothing to do with Tcl, but SWIG's Ruby bindings always generate code that relies on the Tcl libraries. Unless your Tcl header files live in one of your system's standard include directory, you need to tell `extconf.rb` where to find them.

See Also

- http://www.swig.org/
- On Debian GNU/Linux systems, you can install SWIG as the `swig` package

Recipe 22.4. Writing Inline C in Your Ruby Code

Credit: Garrett Rooney

Problem

You want to implement small portions of your program in C without going to the trouble of creating a C extension to Ruby.

Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly Media, Inc.
Print Publication Date: 2006/07/19

Solution

Embed C code right in your Ruby program, and let RubyInline (available as the `rubyinline` gem) create an extension automatically.

For example, if you want to use C's stdio functions to copy a file, you can write RubyInline code like this:[1]

[1] RubyInline won't work from within `irb`, so this is a standalone program.

```
#!/usr/bin/ruby -w
# copy.rb
require 'rubygems'
require 'inline'

class Copier
inline do |builder|
  builder.c <<END
void copy_file(const char *source, const char *dest)
{
  FILE *source_f = fopen(source, "r");
  if (!source_f)
  {
    rb_raise(rb_eIOError, "Could not open source : '%s'", source);
  }

  FILE *dest_f = fopen(dest, "w+");
  if (!dest_f)
  {
    rb_raise(rb_eIOError, "Could not open destination : '%s'", dest);
  }

  char buffer[1024];

  int nread = fread(buffer, 1, 1024, source_f);
  while (nread > 0)
  {
    fwrite(buffer, 1, nread, dest_f);
    nread = fread(buffer, 1, 1024, source_f);
  }
}
END
 end
end
```

The C function `copy_file` now exists as an instance method of `Copier`:

```
open('source.txt', 'w') { |f| f << 'Some text.' }
Copier.new.copy_file('source.txt', 'dest.txt')
puts open('dest.txt') { |f| f.read }
```

Run this Ruby script, and you'll see it copy the string "Some text." from `source.txt` to `dest.txt`.

Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly Media, Inc.
Print Publication Date: 2006/07/19

Discussion

RubyInline is a framework that lets you embed other languages inside your Ruby code. It defines the `Module#inline` method, which returns a builder object. You pass the builder a string containing code written in a language other than Ruby, and the builder transforms it into something that you can call from Ruby.

When given C or C++ code (the two languages supported in the default RubyInline install), the builder objects writes a small extension to disk, compiles it, and loads it. You don't have to deal with the compilation yourself, but you can see the generated code and compiled extensions in the `.ruby_inline` subdirectory of your home directory.

There are some limitations you should be aware of, though.

First, RubyInline only understands a limited subset of C and C++. The functions you embed can only accept and return arguments of the types `char`, `unsigned`, `unsigned int`, `char *`, `int`, `long`, and `unsigned long`.

If you need to use other types, RubyInline won't be able to automatically generate the wrapper functions. You'll have to work around the problem using the `inline.c_raw` function to embed code that conforms to the Ruby C API, just like any other extension.

Second, if you're going to just run a script that uses RubyInline, you'll need to have the Ruby development libraries and headers installed, along with a C/C++ compiler to actually build the extension.

There's a way around this, though: RubyInline lets you generate a RubyGem package with a precompiled extension. See the RubyInline docs on the `inline_package` script for details.

As always, be careful to make sure that it's actually worth the trouble to write C code. You should only rewrite part of a Ruby program in C if you've actually determined that Ruby spends a lot of time there. You should benchmark before and after your change, to make sure that you're making things better rather than worse. Writing C code within your Ruby code is much easier than writing a separate extension, but writing Ruby code is easier still.

See Also

- http://www.zenspider.com/ZSS/Products/RubyInline/
- http://rubyforge.org/projects/rubyinline/
- Recipe 17.12, "Profiling Your Application"
- Recipe 17.13, "Benchmarking Competing Solutions"

Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly Media, Inc.
Print Publication Date: 2006/07/19

Recipe 22.5. Using Java Libraries with JRuby

Credit: Thomas Enebo

Problem

Java offers many class libraries that would be useful to a Ruby programmer; you'd like to use one of those libraries from within Ruby. A Java JDBC database may allow you to connect to a database for which Ruby has no connector. Or perhaps you need to use an obscure Java library that has no Ruby counterpart.

Solution

JRuby provides an alternate implementation of the Ruby programming language that runs atop the Java Virtual Machine. When you interpret a Ruby program with JRuby instead of using the default Ruby interpreter, you can load and use Java classes from within the Ruby code.

The first step to using JRuby is to install it:

1. Download the latest copy of JRuby (see below for the address).
2. Unzip the JRuby package into the directory where you'd like to install it.
3. Add to your PATH environment variable the `bin/` subdirectory of your JRuby installation.
4. Unless you've already installed it, download the Java Runtime Environment from Sun's Java web site and install it. You'll need the JRE version 1.4.x or higher.

Now you can invoke the JRuby interpreter with the `jruby` command and use it to run Ruby code. Here's a simple example that imports and uses Java's built-in `Random` class:

```
#!/usr/bin/env jruby
# random.jrb
require 'java'
include_class 'java.util.Random'

r = Random.new(123)
puts "Some random number #{r.nextInt % 10}"
r.seed = 456
puts "Another random number #{r.nextInt % 10}"
```

Heres a run of this program:

```
$ jruby random.jrb
Some random number 9
Another random number 0
```


Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly Media, Inc.
Print Publication Date: 2006/07/19

Discussion

JRuby generally behaves like Ruby. The `jruby` interpreter supports a common subset of Ruby's command-line options, and includes a subset of common core libraries. As JRuby is developed, it will eventually end up with all of Ruby's options and libraries.

The first step in a JRuby program is to load the Java support classes. If you don't do this, you can still use the JRuby interpreter, but you'll be limited to a subset of the Ruby core libraries: you might as well just use the C implementation.

The statement `require 'java'` updates Ruby's `Object` class with an `include_class` method, which you can use to import Java classes. When we call `include_class` to include a class like `java.util.Random`, Ruby inserts a class called `Random` into the current namespace. This class is really a Ruby class that proxies method calls to the underlying Java class.

The `Random` class proxies a constructor call to the `java.util.Random` constructor. `Random#nextInt` becomes a call to `java.util.Random#nextInt`. `Random#seed=` becomes a call to `java.util.Random#setSeed`; JRuby creates `seed=` as a Ruby convenience method, to make the Java classes feel more like Ruby.

If you're including a Java class whose name conflicts with an existing constant in your namespace, then `include_class` will throw a `ConstantAlreadyExistsError`. This is problematic if you want to use Java classes like `java.lang.String`, whose names conflict with the names of built-in Ruby classes. Fortunately, you can customize the name of the proxy class created by `include_class`. This piece of code loads '`java.lang.String`' as the class `JString` instead of `String`:

```
include_class('java.lang.String') { |package,name| "J" + name }
```

It's worth noting that JRuby implicitly translates primitive types between Ruby and Java. In the `Random` constructor, the `Fixnum` argument 123 gets implicitly converted to a Java primitive long, since that's what the `java.util.Random` constructor takes.

Table 22-1.

Ruby type	Java type
String	char, String
Fixnum	long, int, java.lang.Long, java.lang.Integer
Float	float, double, Java.lang.Float, java.lang.Double
Boolean	java.lang.Boolean, boolean

Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly Media, Inc.
Print Publication Date: 2006/07/19

This automatic conversion creates some amount of ambiguity, because Java supports method overloading and Ruby doesn't. Suppose you have a Java class which defines two methods with the same name:

```
class Foo
{
  public void bar(int arg) {…}
  public void bar(long arg) {…}
}
```

If you import that class into JRuby and call `Foo#bar`, to which method should the proxy class dispatch your call?

```
Foo.new.bar(5)
```

In JRuby, the exact heuristic is undefined. In practice, this is not a huge problem, since methods that define same-named methods are semantically equivalent. If you do encounter an ambiguous case, you can work around ambiguity using Java's reflection APIs.

Convenience methods

JRuby tries to make Java classes and objects seem as unobtrusive to Ruby as it can. In our earlier example, we saw how a setter:

```
setSeed(value);
```

Can be called from Ruby as:

```
seed = value
```

JRuby supports the following additional Ruby method name shortcuts:

Table 22-2.

Java	Ruby
obj.getFoo( )	obj.foo
obj.setFoo(value)	obj.foo = value
obj.isFoo(value)	obj.foo? value

The original name still exists, so if you like you can use `getFoo` and `setFoo` from Ruby. Of course, if Java already has a method by the same shorthand name (e.g., `obj.foo`), Ruby won't create the shorthand name.

JRuby also provides some Ruby methods that make Java classes seem more like Ruby classes. Here is a list as of Ruby 0.8.3:

Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly Media, Inc.
Print Publication Date: 2006/07/19

- All of Java's `Map`, `Set`, and `List` types define `each`
- `java.lang.Comparable` defines `<=>`
- List defines `<<`, `sort`, and `sort!`

JRuby is still a project under development, so expect to see more added as developers discover more candidates.

See Also

- JRuby is available from http://jruby.sourceforge.net/
- You can download the JRE from Sun's Java site at http://java.sun.com/

Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly Media, Inc.
Print Publication Date: 2006/07/19