Chapter 21

Ruby and Microsoft Windows

Ruby runs in a number of environments. Some of these are Unix-based, and others are based on the various flavors of Microsoft Windows. Ruby came from people who were Unix-centric, but over the years it has developed a whole lot of useful features in the Windows world, too. In this chapter, we'll look at these features and share some secrets that let you use Ruby effectively under Windows.

Getting Ruby for Windows

Although you could build Ruby for Windows from source, most people simply download the prebuilt binaries from the main Ruby FTP site.[1] Create a directory for your Ruby installation, and download the latest zip file into it. Unzip the file, and you'll end up with a complete, standard Ruby directory tree (\bin, \doc, \lib and so on). Add the bin directory to your path, and Ruby should be available to you. For example, I downloaded the `.zip` file into the directory C:\ruby19:

```
C:\> mkdir \ruby19
C:\ruby19> cd \ruby19
C:\ruby19> ftp ftp.ruby-lang.org
Connected to carbon.ruby-lang.org.
User (carbon.ruby-lang.org:(none)): ftp
331 Please specify the password.
Password: your email address
230 Login successful.
ftp> cd pub/ruby/binaries/mswin32/unstable
250 Directory successfully changed.
ftp> dir
-rw-r--r-- 1 ... Jul 08  2007 ruby-1.9.0-20070709-i386-mswin32.zip
-rw-r--r-- 1 ... Jul 08  2007 ruby-1.9.0-20070709-x64-mswin64_80.zip
-rw-r--r-- 1 ... Oct 28 15:31 ruby-1.9.1-preview1-i386-mswin32.zip
-rw-r--r-- 1 ... Oct 28 15:31 ruby-1.9.1-preview1-x64-mswin64_80.zip
```

1. ftp://ftp.ruby-lang.org/pub/ruby/binaries/mswin32/unstable/

```
ftp> bin
200 Switching to Binary mode.
ftp> get ruby-1.9.1-preview1-i386-mswin32.zip
200 PORT command successful. Consider using PASV.
150 Opening BINARY mode data connection
    for ruby-1.9.1-preview1-i386-mswin32.zip (13535099 bytes).
226 File send OK.
ftp: 13535099 bytes received in 48.06Seconds 280.21Kbytes/sec.
ftp> by
C:\ruby19> unzip.exe ruby-1.9.0-0-i386-mswin32.zip
   :  :
C:\ruby19> PATH=\ruby19\bin;%PATH%
C:\ruby19> ruby -v
ruby 1.9.1 (2008-10-28 revision 19983) [i386-mswin32]
```

Running Ruby Under Windows

You'll find two executables in the Ruby Windows distribution.

ruby.exe is meant to be used at a command prompt (a DOS shell), just as in the Unix version. For applications that read and write to the standard input and output, this is fine. But this also means that any time you run ruby.exe, you'll get a DOS shell even if you don't want one—Windows will create a new command prompt window and display it while Ruby is running. This may not be appropriate behavior if, for example, you double-click a Ruby script that uses a graphical interface (such as Tk) or if you are running a Ruby script as a background task or from inside another program.

In these cases, you will want to use rubyw.exe. It is the same as ruby.exe except that it does not provide standard in, standard out, or standard error and does not launch a DOS shell when run.

You can set up file associations using the assoc and ftype commands so that Ruby will automatically run Ruby when you double-click the name of a Ruby script:

```
C:\> assoc .rb=RubyScript
C:\> ftype RubyScript="C:\ruby1.9\bin\ruby.exe %1 %*
```

Win32API

If you plan on doing Ruby programming that needs to access some Windows 32 API functions directly or that needs to use the entry points in some other DLLs, we've got good news for you—the Win32API library.

As an example, here's some code that's part of a larger Windows application used by our book fulfillment system to download and print invoices and receipts. A web application generates a PDF file, which the Ruby script running on Windows downloads into a local file. The script then uses the print shell command under Windows to print this file.

What About the One-Click Installer?

Ruby 1.8 had a no-assembly-required package called the One-Click Ruby Installer (1CRI). Download it, and it will install Ruby, a bunch of gems, and even a version of the original PickAxe.

However, because this installer packages so many gems and because many of these gems haven't been updated for Ruby 1.9, the team has not released a Ruby 1.9 version of 1CRI at the time of this writing. Check http://rubyinstaller.rubyforge.org for the current status.

```
arg   = "ids=#{resp.intl_orders.join(",")}"
fname = "/temp/invoices.pdf"
site = Net::HTTP.new(HOST, PORT)
site.use_ssl = true
http_resp, = site.get2("/ship/receipt?" + arg,
                       'Authorization' => 'Basic ' +
                       ["name:passwd"].pack('m').strip )
File.open(fname, "wb") {|f| f.puts(http_resp.body) }
shell = Win32API.new("shell32","ShellExecute",
                       ['L','P','P','P','P','L'], 'L' )
shell.Call(0, "print", fname, 0,0, SW_SHOWNORMAL)
```

You create a Win32API object that represents a call to a particular DLL entry point by specifying the name of the function, the name of the DLL that contains the function, and the function signature (argument types and return type). In the previous example, the variable shell wraps the Windows function ShellExecute in the shell32 DLL. The second parameter is an array of characters describing the types of the parameters the method takes: 'n' and 'l' represent numbers, 'i' represent integers, 'p' represents pointers to data stored in a string, and 'v' a void type (used for export parameters only). These strings are case-insensitive. So our method takes a number, four string pointers, and a number. The last parameter says that the method returns a number. The resulting object is a proxy to the underlying ShellExecute function, and can be used to make the call to print the file that we downloaded.

Many of the arguments to DLL functions are binary structures of some form. Win32API handles this by using Ruby String objects to pass the binary data back and forth. You will need to pack and unpack these strings as necessary.

Windows Automation

If groveling around in the low-level Windows API doesn't interest you, Windows Automation may—you can use Ruby as a client for Windows Automation thanks to a Ruby extension called WIN32OLE, written by Masaki Suketa. Win32OLE is part of the standard Ruby distribution.

Windows Automation allows an automation controller (a client) to issue commands and queries against an automation server, such as Microsoft Excel, Word, and so on.

You can execute an automation server's method by calling a method of the same name from a WIN32OLE object. For instance, you can create a new WIN32OLE client that launches a fresh copy of Internet Explorer and commands it to visit its home page:

```
require 'win32ole'
ie = WIN32OLE.new('InternetExplorer.Application')
ie.visible = true
ie.gohome
```

You could also make it navigate to a particular page:

```
require 'win32ole'
ie = WIN32OLE.new('InternetExplorer.Application')
ie.visible = true
ie.navigate("http://www.pragprog.com")
```

Methods that aren't known to WIN32OLE (such as visible, gohome, or navigate) are passed on to the WIN32OLE#invoke method, which sends the proper commands to the server.

Getting and Setting Properties

You can set and get *properties* from the server using normal Ruby hash notation. For example, to set the Rotation property in an Excel chart, you could write this:

```
excel = WIN32OLE.new("excel.application")
excelchart = excel.Charts.Add()
...
excelchart['Rotation'] = 45
puts excelchart['Rotation']
```

An OLE object's properties are automatically set up as attributes of the WIN32OLE object. This means you can set a property by assigning to an object attribute:

```
excelchart.rotation = 45
r = excelchart.rotation
```

The following example is a modified version of the sample file excel2.rb (found in the ext/win32/samples directory). It starts Excel, creates a chart, and then rotates it on the screen. Watch out, Pixar!

```
require 'win32ole'
#   -4100 is the value for the Excel constant xl3DColumn.
ChartTypeVal = -4100;
excel = WIN32OLE.new("excel.application")
# Create and rotate the chart
excel['Visible'] = TRUE
```

```
excel.Workbooks.Add()
excel.Range("a1")['Value'] = 3
excel.Range("a2")['Value'] = 2
excel.Range("a3")['Value'] = 1
excel.Range("a1:a3").Select()
excelchart = excel.Charts.Add()
excelchart['Type'] = ChartTypeVal
30.step(180, 5) do |rot|
  excelchart.rotation = rot
  sleep(0.1)
end
excel.ActiveWorkbook.Close(0)
excel.Quit()
```

Named Arguments

Other automation client languages such as Visual Basic have the concept of *named arguments*. Suppose you had a Visual Basic routine with the following signature:

```
Song(artist, title, length):    rem Visual Basic
```

Instead of calling it with all three arguments in the order specified, you could use named arguments:

```
Song title := 'Get It On':      rem Visual Basic
```

This is equivalent to the call Song(nil, 'Get It On', nil).

In Ruby, you can use this feature by passing a hash with the named arguments:

```
Song.new('title' => 'Get It On')
```

for each

Where Visual Basic has a for each statement to iterate over a collection of items in a server, a WIN32OLE object has an each method (which takes a block) to accomplish the same thing:

```
require 'win32ole'
excel = WIN32OLE.new("excel.application")
excel.Workbooks.Add
excel.Range("a1").Value = 10
excel.Range("a2").Value = 20
excel.Range("a3").Value = "=a1+a2"
excel.Range("a1:a3").each do |cell|
  p cell.Value
end
```

Events

Your automation client written in Ruby can register itself to receive events from other programs. This is done using the WIN32OLE_EVENT class.

This example (based on code from the Win32OLE 0.1.1 distribution) shows the use of an event sink that logs the URLs that a user browses to when using Internet Explorer:

```
require 'win32ole'
$urls = []
def navigate(url)
  $urls << url
end
def stop_msg_loop
  puts "IE has exited..."
  throw :done
end
def default_handler(event, *args)
  case event
  when "BeforeNavigate"
    puts "Now Navigating to #{args[0]}..."
  end
end
ie = WIN32OLE.new('InternetExplorer.Application')
ie.visible = TRUE
ie.gohome
ev = WIN32OLE_EVENT.new(ie, 'DWebBrowserEvents')
ev.on_event {|*args| default_handler(*args)}
ev.on_event("NavigateComplete") {|url| navigate(url)}
ev.on_event("Quit") {|*args| stop_msg_loop}
catch(:done) do
  loop do
    WIN32OLE_EVENT.message_loop
  end
end
puts "You Navigated to the following URLs: "
$urls.each_with_index do |url, i|
  puts "(#{i+1}) #{url}"
end
```

Optimizing

As with most (if not all) high-level languages, it can be all too easy to churn out code that is unbearably slow, but that can be easily fixed with a little thought.

With WIN32OLE, you need to be careful with unnecessary dynamic lookups. Where possible, it is better to assign a WIN32OLE object to a variable and then reference elements from it, rather than creating a long chain of "." expressions.

For example, instead of writing this:

```
workbook.Worksheets(1).Range("A1").value = 1
workbook.Worksheets(1).Range("A2").value = 2
workbook.Worksheets(1).Range("A3").value = 4
workbook.Worksheets(1).Range("A4").value = 8
```

we can eliminate the common subexpressions by saving the first part of the expression to a temporary variable and then make calls from that variable:

```
worksheet = workbook.Worksheets(1)
worksheet.Range("A1").value = 1
worksheet.Range("A2").value = 2
worksheet.Range("A3").value = 4
worksheet.Range("A4").value = 8
```

You can also create Ruby stubs for a particular Windows type library. These stubs wrap the OLE object in a Ruby class with one method per entry point. Internally, the stub uses the entry point's number, not name, which speeds access.

Generate the wrapper class using the olegen.rb script in the ext\win32ole\samples directory, giving it the name of the type library to reflect on:

```
C:\> ruby olegen.rb 'NetMeeting 1.1 Type Library' >netmeeting.rb
```

The external methods and events of the type library are written as Ruby methods to the given file. You can then include it in your programs and call the methods directly. Let's try some timings:

```
require 'netmeeting'
require 'benchmark'
include Benchmark
bmbm(10) do |test|
  test.report("Dynamic") do
    nm = WIN32OLE.new('NetMeeting.App.1')
    10000.times { nm.Version }
  end
  test.report("Via proxy") do
    nm = NetMeeting_App_1.new
    10000.times { nm.Version }
  end
end
```

produces:

```
Rehearsal ---------------------------------------------
Dynamic     0.600000   0.200000   0.800000 (  1.623000)
Via proxy   0.361000   0.140000   0.501000 (  0.961000)
------------------------------------ total: 1.301000sec

                user     system      total        real
Dynamic     0.471000   0.110000   0.581000 (  1.522000)
Via proxy   0.470000   0.130000   0.600000 (  0.952000)
```

The proxy version is more than 40 percent faster than the code that does the dynamic lookup.

More Help

If you need to interface Ruby to Windows NT, 2000, or XP, you may want to take a look at Daniel Berger's Win32Utils project (http://rubyforge.org/projects/win32utils/). There you'll find modules for interfacing to the Windows clipboard, event log, scheduler, and so on.

Also, the DL library (described briefly on page 746) allows Ruby programs to invoke methods in dynamically loaded shared objects. On Windows, this means that your Ruby code can load and invoke entry points in a Windows DLL. For example, the following code, taken from the DL source code in the standard Ruby distribution, pops up a message box on a Windows machine and determines which button the user clicked:

Download **samples/win32_15.rb**

```
require 'dl'
User32 = DL.dlopen("user32")
MB_OKCANCEL = 1
message_box = User32['MessageBoxA', 'ILSSI']
r, rs = message_box.call(0, 'OK?', 'Please Confirm', MB_OKCANCEL)
case r
when 1
  print("OK!\n")
when 2
  print("Cancel!\n")
end
```

This code opens the User32 DLL. It then creates a Ruby object, message_box, that wraps the MessageBoxA entry point. The second paramater, "ILSSI", declares that the method returns an Integer and takes a Long, two Strings, and an Integer as parameters.

The wrapper object is then used to call the message box entry point in the DLL. The return values are the result (in this case, the identifier of the button pressed by the user) and an array of the parameters passed in (which we ignore).