Chapter 25

Reflection, ObjectSpace, and Distributed Ruby

One of the advantages of dynamic languages such as Ruby is the ability to *introspect*—to examine aspects of a program from within the program itself. This process is also called *reflection*.

When you introspect, you think about your thoughts and feelings. This is interesting, because you're using thought to analyze thought. It's the same when programs use introspection—a program can discover the following information about itself:

- What objects it contains
- Its class hierarchy
- The attributes and methods of objects
- Information on methods

Armed with this information, we can look at particular objects and decide which of their methods to call at runtime—even if the class of the object didn't exist when we first wrote the code. We can also start doing clever things, perhaps modifying the program while it's running. Later in this chapter we'll look at distributed Ruby and marshaling, two reflection-based technologies that let us send objects around the world and through time.

Looking at Objects

Have you ever craved the ability to traverse *all* the living objects in your program? We have! Ruby lets you perform this trick with ObjectSpace.each_object. We can use it to do all sorts of neat tricks.

For example, to iterate over all objects of type Float, you'd write the following:

```
a = 102.7
b = 95.1
ObjectSpace.each_object(Float) {|x| p x }
```

produces:

```
95.1
102.7
2.71828182845905
3.14159265358979
2.22044604925031e-16
1.79769313486232e+308
2.2250738585072e-308
```

Hey, where did all those extra numbers come from? We didn't define them in our program. Well, the Math module defines constants for e and π, and if you look on pages 528 and 588, you'll see that the Float class defines constants for the maximum and minimum float, as well as epsilon, the smallest distinguishable difference between two floats. Since we are examining *all* living objects in the system, these turn up as well.

Let's try the same example with different values. This time, they're objects of type Fixnum:

```
a = 102
b = 95
ObjectSpace.each_object(Fixnum) {|x| p x }
```

(Produces no output.)

Neither of the Fixnum objects we created showed up. That's because ObjectSpace doesn't know about objects with immediate values: Fixnum, Symbol, true, false, and nil.

Looking Inside Objects

Once you've found an interesting object, you may be tempted to find out just what it can do. Unlike static languages, where a variable's type determines its class, and hence the methods it supports, Ruby supports liberated objects. You really cannot tell exactly what an object can do until you look under its hood.[1] We talk about this in the *Duck Typing* chapter starting on page 370.

For instance, we can get a list of all the methods to which an object will respond (these include methods in an object's class and that class's ancestors):

```
r = 1..10 # Create a Range object
list = r.methods
list.length   # =>  101
list[0..3]    # =>  [:==, :===, :eql?, :hash]
```

We can check to see whether an object responds to a particular method:

```
r.respond_to?("frozen?")   # =>  true
r.respond_to?(:has_key?)   # =>  false
"me".respond_to?("==")     # =>  true
```

1. Or under its bonnet, for objects created to the east of the Atlantic.

We can determine our object's class and its unique object ID and test its relationship to other classes:

```
num = 1
num.object_id              # =>  3
num.class                  # =>  Fixnum
num.kind_of? Fixnum        # =>  true
num.kind_of? Numeric       # =>  true
num.instance_of? Fixnum    # =>  true
num.instance_of? Numeric   # =>  false
```

Looking at Classes

Knowing about objects is one part of reflection, but to get the whole picture, you also need to be able to look at classes—the methods and constants that they contain.

Looking at the class hierarchy is easy. You can get the parent of any particular class using Class#superclass. For classes *and* modules, Module#ancestors lists both superclasses and mixed-in modules:

```
klass = Fixnum
begin
  print klass
  klass = klass.superclass
  print " < " if klass
end while klass
puts
p Fixnum.ancestors
```

produces:

```
Fixnum < Integer < Numeric < Object < BasicObject
[Fixnum, Integer, Numeric, Comparable, Object, Kernel, BasicObject]
```

If you want to build a complete class hierarchy, just run that code for every class in the system. We can use ObjectSpace to iterate over all Class objects:

```
ObjectSpace.each_object(Class) do |klass|
   # ...
end
```

Looking Inside Classes

We can find out a bit more about the methods and constants in a particular object. Instead of just checking to see whether the object responds to a given message, we can ask for methods by access level, and we can ask for just singleton methods.

1.9 We can also take a look at the object's constants, local, and instance variables:

```
class Demo
  @@var = 99
  CONST = 1.23

  private
    def private_method
    end
  protected
    def protected_method
    end
  public
    def public_method
      @inst = 1
      i = 1
      j = 2
      local_variables
    end

  def Demo.class_method
  end
end

Demo.private_instance_methods(false)     # =>   [:private_method]
Demo.protected_instance_methods(false)   # =>   [:protected_method]
Demo.public_instance_methods(false)      # =>   [:public_method]
Demo.singleton_methods(false)            # =>   [:class_method]
Demo.class_variables                     # =>   [:@@var]
Demo.constants(false)                    # =>   [:CONST]

demo = Demo.new
demo.instance_variables                  # =>   []
# Get 'public_method' to return its local variables
# and set an instance variable
demo.public_method                       # =>   [:i, :j]
demo.instance_variables                  # =>   [:@inst]
```

You may be wondering what all the false parameters were in the previous code. As of Ruby 1.8, these reflection methods will by default recurse into parent classes, their parents, and so on, up the ancestor chain. Passing in false stops this kind of prying.

Given a list of method names, we may now be tempted to try calling them. Fortunately, that's easy with Ruby.

Calling Methods Dynamically

The Object#send method lets you tell an object to invoke a method by name. It works on any object.

```
"John Coltrane".send(:length)           # =>  13
"Miles Davis".send("sub", /iles/, '.')  # =>  "M. Davis"
```

Another way of invoking methods dynamically uses Method objects. A Method object is like a Proc object: it represents a chunk of code and a context in which it executes. In this case, the code is the body of the method, and the context is the object that created the method. Once we have our Method object, we can execute it sometime later by sending it the message call:

```
trane = "John Coltrane".method(:length)
miles = "Miles Davis".method("sub")

trane.call               # =>  13
miles.call(/iles/, '.')  # =>  "M. Davis"
```

You can pass the Method object around as you would any other object, and when you invoke Method#call, the method is run just as if you had invoked it on the original object. It's like having a C-style function pointer but in a fully object-oriented style.

You can use Method objects where you could use proc objects. For example, they work with iterators:

```
def double(a)
  2*a
end

method_object = method(:double)

[ 1, 3, 5, 7 ].map(&method_object)  # =>  [2, 6, 10, 14]
```

Method objects are bound to one particular object. You can create *unbound* methods (of class UnboundMethod) and then subsequently bind them to one or more objects. The binding creates a new Method object. As with aliases, unbound methods are references to the definition of the method at the time they are created:

```
unbound_length = String.instance_method(:length)
class String
  def length
    99
  end
end
str = "cat"
str.length          # =>  99
bound_length = unbound_length.bind(str)
bound_length.call   # =>  3
```

As good things come in threes, here's yet another way to invoke methods dynamically. The eval method (and its variations such as class_eval, module_eval, and instance_eval) will parse and execute an arbitrary string of legal Ruby source code.

```
trane = %q{"John Coltrane".length}
miles = %q{"Miles Davis".sub(/iles/, '.')}

eval trane   # =>  13
eval miles   # =>  "M. Davis"
```

When using eval, it can be helpful to state explicitly the context in which the expression should be evaluated, rather than using the current context. You can obtain a context by calling Kernel#binding at the desired point:

```
def get_a_binding
  val = 123
  binding
end

val = "cat"

the_binding = get_a_binding
eval("val", the_binding)   # =>  123
eval("val")                # =>  "cat"
```

The first eval evaluates val in the context of the binding *as it was* when the method get_a_binding was executing. In this binding, the variable val had a value of 123. The second eval evaluates val in the top-level binding, where it has the value "cat".

Performance Considerations

As we've seen in this section, Ruby gives us several ways to invoke an arbitrary method of some object: Object#send, Method#call, and the various flavors of eval.

You may prefer to use any one of these techniques depending on your needs, but be aware that eval is significantly slower than the others (or, for optimistic readers, send and call are significantly faster than eval):

Download **samples/ospace_15.rb**

```
require 'benchmark'
include Benchmark
test = "Stormy Weather"
m = test.method(:length)
n = 100000
bm(12) {|x|
  x.report("call") { n.times { m.call } }
  x.report("send") { n.times { test.send(:length) } }
  x.report("eval") { n.times { eval "test.length" } }
}
```

produces:

```
                 user     system      total        real
call         0.020000   0.000000   0.020000 (  0.022663)
send         0.010000   0.000000   0.010000 (  0.016671)
eval         0.780000   0.000000   0.780000 (  0.776943)
```

System Hooks

A *hook* is a technique that lets you trap some Ruby event, such as object creation. Let's take a look at some common Ruby hook techniques.

Hooking Method Calls

The simplest hook technique in Ruby is to intercept calls to methods in system classes. Perhaps you want to log all the operating system commands your program executes. Simply rename the method Kernel.system and substitute it with one of your own that both logs the command and calls the original Kernel method:

```
module Kernel
  alias_method :old_system, :system
  def system(*args)
    result = old_system(*args)
    puts "system(#{args.join(', ')}) returned #{result.inspect}"
    result
  end
end
system("date")
system("kangaroo", "-hop 10", "skippy")
```

produces:

```
Mon Apr 13 13:26:15 CDT 2009
system(date) returned true
system(kangaroo, -hop 10, skippy) returned nil
```

The problem with this technique is that you're relying on there not being an existing method called old_system. A better alternative is to make use of method objects, which are effectively anonymous:

```
module Kernel
  old_system_method = instance_method(:system)
  define_method(:system) do |*args|
    result = old_system_method.bind(self).call(*args)
    puts "system(#{args.join(', ')}) returned #{result.inspect}"
    result
  end
end
system("date")
system("kangaroo", "-hop 10", "skippy")
```

produces:

```
Mon Apr 13 13:26:15 CDT 2009
system(date) returned true
system(kangaroo, -hop 10, skippy) returned nil
```

Object Creation Hooks

Ruby lets you get involved when objects are created. If you can be present when every object is born, you can do all sorts of interesting things: you can wrap them, add methods to them, remove methods from them, and add them to containers to implement persistence—you name it. We'll show a simple example here. We'll add a timestamp to every object as it's created. First, we'll add a timestamp attribute to every object in the system. We can do this by hacking class Object itself:

```
class Object
  attr_accessor :timestamp
end
```

Then, we need to hook object creation to add this timestamp. One way to do this is to do our method renaming trick on Class#new, the method that's called to allocate space for a new object. The technique isn't perfect—some built-in objects, such as literal strings, are constructed without calling new—but it'll work just fine for objects we write.

```
class Class
  old_new = instance_method :new
  define_method :new do |*args, &block|
    result = old_new.bind(self).call(*args, &block)
    result.timestamp = Time.now
    result
  end
end
```

Finally, we can run a test. We'll create a couple of objects a few milliseconds apart and check their timestamps:

```
class Test
end

obj1 = Test.new
sleep(0.002)
obj2 = Test.new

obj1.timestamp.to_f   # =>   1239647175.73519
obj2.timestamp.to_f   # =>   1239647175.73724
```

Tracing Your Program's Execution

While we're having fun reflecting on all the objects and classes in our programs, let's not forget about the humble statements that make our code actually do things. It turns out that Ruby lets us look at these statements, too.

First, you can watch the interpreter as it executes code. set_trace_func executes a proc with all sorts of juicy debugging information whenever a new source line is executed, methods are called, objects are created, and so on.

You'll find a full description on page 576, but here's a taste:

Download **samples/ospace_21.rb**

```
class Test
  def test
    a = 1
    b = 2
  end
end
set_trace_func lambda {|event, file, line, id, binding, classname|
  printf "%8s %s:%-2d %-15s %-15s\n", event, file, line, classname, id
}
t = Test.new
t.test
```

produces:

```
c-return prog.rb:10 Kernel          set_trace_func
    line prog.rb:11
  c-call prog.rb:11 Class           new
  c-call prog.rb:11 BasicObject     initialize
c-return prog.rb:11 BasicObject     initialize
c-return prog.rb:11 Class           new
    line prog.rb:12
    call prog.rb:2  Test            test
    line prog.rb:3  Test            test
    line prog.rb:4  Test            test
  return prog.rb:2  Test            test
```

The method trace_var (described on page 579) lets you add a hook to a global variable; whenever an assignment is made to the global, your proc is invoked.

How Did We Get Here?

That's a fair question...one we ask ourselves regularly. Mental lapses aside, in Ruby at least you can find out exactly "how you got there" by using the method caller, which returns an Array of String objects representing the current call stack:

```
def cat_a
  puts caller
end
def cat_b
  cat_a
end
def cat_c
  cat_b
end
cat_c
```

produces:

```
/tmp/prog.rb:5:in `cat_b'
/tmp/prog.rb:8:in `cat_c'
/tmp/prog.rb:10:in `<main>'
```

Once you've figured out how you got there, where you go next is up to you.

Source Code

Ruby executes programs from plain old files. You can look at these files to examine the source code that makes up your program using one of a number of techniques.

The special variable __FILE__ contains the name of the current source file. This leads to a fairly short (if cheating) Quine—a program that outputs its own source code:

```
print File.read(__FILE__)
```

produces:

```
print File.read(__FILE__)
```

As we saw in the previous section, the method Kernel.caller returns the call stack as a list. Each entry in this list starts off with a filename, a colon, and a line number in that file. You can parse this information to display source. In the following example, we have a main program, main.rb, that calls a method in a separate file, sub.rb. That method in turns invokes a block, where we traverse the call stack and write out the source lines involved. Notice the use of a hash of file contents, indexed by the filename.

Here's the code that dumps out the call stack, including source information:

```
def dump_call_stack
  file_contents = {}
  puts "File            Line  Source Line"
  puts "----------------+----+------------"
  caller.each do |position|
    next unless position =~ /\A(.*?):(\d+)/
    file = $1
    line = Integer($2)
    file_contents[file] ||= File.readlines(file)
    printf("%-15s:%3d - %s", File.basename(file), line,
           file_contents[file][line-1].lstrip)
  end
end
```

The (trivial) file sub.rb contains a single method:

```
def sub_method(v1, v2)
  main_method(v1*3, v2*6)
end
```

And here's the main program, which invokes the stack dumper after being called back by the submethod:

```
require 'sub'
require 'stack_dumper'
def main_method(arg1, arg2)
  dump_call_stack
end
sub_method(123, "cat")
```

produces:

```
File              Line  Source Line
----------------+----+------------
main.rb         :  5 - dump_call_stack
sub.rb          :  2 - main_method(v1*3, v2*6)
main.rb         :  8 - sub_method(123, "cat")
```

The SCRIPT_LINES__ constant is closely related to this technique. If a program initializes a constant called SCRIPT_LINES__ with a hash, that hash will receive a new entry for every file subsequently loaded into the interpreter using require or load. The entry's key is the name of the file, and the value is the source of the file as an array of strings.

Behind the Curtain: The Ruby VM

1.9

Ruby 1.9 comes with a new virtual machine, called YARV. As well as being faster than the old interpreter, YARV exposes some of its state via Ruby classes.

If you'd like to know what Ruby is doing with all that code you're writing, you can ask YARV to show you the intermediate code that it is executing. You can ask it to compile the Ruby code in a string or in a file and then disassemble it and even run it.[2] Here's a trivial example:

```
code = RubyVM::InstructionSequence.compile('a = 1; puts 1 + a')
puts code.disassemble
```

produces:

```
== disasm: <RubyVM::InstructionSequence:<compiled>@<compiled>>==========
local table (size: 2, argc: 0 [opts: 0, rest: -1, post: 0, block: -1] s1)
[ 2] a
0000 trace            1                                               (   1)
0002 putobject        1
0004 setlocal         a
0006 trace            1
0008 putnil
0009 putobject        1
0011 getlocal         a
0013 opt_plus
0014 send             :puts, 1, nil, 8, <ic>
0020 leave
```

Maybe you want to know how Ruby handles #{ } substitutions in strings:

```
code = RubyVM::InstructionSequence.compile('a = 1; puts "a = #{a}."')
puts code.disassemble
```

2. People often ask if they can dump the opcodes out and later reload them. The answer is no—the interpreter has the code to do this, but it is disabled because there is not yet an intermediate code verifier for YARV.

produces:

```
== disasm: <RubyVM::InstructionSequence:<compiled>@<compiled>>==========
local table (size: 2, argc: 0 [opts: 0, rest: -1, post: 0, block: -1] s1)
[ 2] a
0000 trace            1                                               (   1)
0002 putobject        1
0004 setlocal         a
0006 trace            1
0008 putnil
0009 putobject        "a = "
0011 getlocal         a
0013 tostring
0014 putstring        "."
0016 concatstrings    3
0018 send             :puts, 1, nil, 8, <ic>
0024 leave
```

For a full list of the opcodes, print out RubyVM::INSTRUCTION_NAMES.

Marshaling and Distributed Ruby

Ruby features the ability to *serialize* objects, letting you store them somewhere and reconstitute them when needed. You can use this facility, for instance, to save a tree of objects that represent some portion of application state—a document, a CAD drawing, a piece of music, and so on.

Ruby calls this kind of serialization *marshaling* (think of railroad marshaling yards where individual cars are assembled in sequence into a complete train, which is then dispatched somewhere). Saving an object and some or all of its components is done using the method Marshal.dump. Typically, you will dump an entire object tree starting with some given object. Later, you can reconstitute the object using Marshal.load.

Here's a short example. We have a class Chord that holds a collection of musical notes. We'd like to save away a particularly wonderful chord so we can e-mail it to a couple of hundred of our closest friends. They can then load it into their copy of Ruby and savor it too. Let's start with the classes for Note and Chord:

```
class Note < Struct.new(:value)
  def to_s
    value.to_s
  end
end
class Chord
  def initialize(arr)
    @arr = arr
  end
  def play
    @arr.join('-')
  end
end
```

Now we'll create our masterpiece and use Marshal.dump to save a serialized version to disk:

```
c = Chord.new( [ Note.new("G"),
                 Note.new("Bb"),
                 Note.new("Db"),
                 Note.new("E") ] )
File.open("posterity", "w+") do |f|
  Marshal.dump(c, f)
end
```

Finally, our grandchildren read it in and are transported by our creation's beauty:

```
chord =   Marshal.load(File.open("posterity"))

chord.play   # =>   "G-Bb-Db-E"
```

Custom Serialization Strategy

Not all objects can be dumped: bindings, procedure objects, instances of class IO, and singleton objects cannot be saved outside the running Ruby environment (a TypeError will be raised if you try). Even if your object doesn't contain one of these problematic objects, you may want to take control of object serialization yourself.

Marshal provides the hooks you need. In the objects that require custom serialization, simply implement two instance methods: one called marshal_dump, which writes the object out to a string, and one called marshal_load, which reads a string that you had previously created and uses it to initialize a newly allocated object. (In earlier Ruby versions you'd use methods called _dump and _load, but the new versions play better with Ruby 1.8's new allocation scheme.) The instance method marshal_dump should return an object representing the state to be dumped. When the object is subsequently reconstituted using Marshal.load, the method marshal_load will be called with this object and will use it to set the state of its receiver—it will be run in the context of an allocated but not initialized object of the class being loaded.

For instance, here is a sample class that defines its own serialization. For whatever reasons, Special doesn't want to save one of its internal data members, @volatile. The author has decided to serialize the two other instance variables in an array.

```
class Special
  def initialize(valuable, volatile, precious)
    @valuable = valuable
    @volatile = volatile
    @precious = precious
  end
  def marshal_dump
    [ @valuable, @precious ]
  end
  def marshal_load(variables)
    @valuable = variables[0]
    @precious = variables[1]
    @volatile = "unknown"
  end
```

```
  def to_s
    "#@valuable #@volatile #@precious"
  end
end
obj = Special.new("Hello", "there", "World")
puts "Before: obj = #{obj}"
data = Marshal.dump(obj)
obj = Marshal.load(data)
puts "After: obj = #{obj}"
```

produces:

```
Before: obj = Hello there World
After: obj = Hello unknown World
```

For more details, see the reference section on Marshal beginning on page 583.

YAML for Marshaling

The Marshal module is built into the interpreter and uses a binary format to store objects externally. Although fast, this binary format has one major disadvantage: if the interpreter changes significantly, the marshal binary format may also change, and old dumped files may no longer be loadable.

An alternative is to use a less fussy external format, preferably one using text rather than binary files. One option, supplied as a standard library, is YAML.[3]

We can adapt our previous marshal example to use YAML. Rather than implement specific loading and dumping methods to control the marshal process, we simply define the method to_yaml_properties, which returns a list of instance variables to be saved:

```
require 'yaml'
class Special
  def initialize(valuable, volatile, precious)
    @valuable = valuable
    @volatile = volatile
    @precious = precious
  end
  def to_yaml_properties
    %w{ @precious @valuable }
  end
  def to_s
    "#@valuable #@volatile #@precious"
  end
end
```

3. http://www.yaml.org. YAML stands for YAML Ain't Markup Language, but that hardly seems important.

```
obj = Special.new("Hello", "there", "World")
puts "Before: obj = #{obj}"
data = YAML.dump(obj)
obj = YAML.load(data)
puts "After: obj = #{obj}"
```

produces:

```
Before: obj = Hello there World
After: obj = Hello  World
```

We can have a look at what YAML creates as the serialized form of the object—it's pretty simple:

```
obj = Special.new("Hello", "there", "World")
puts YAML.dump(obj)
```

produces:

```
--- !ruby/object:Special
precious: World
valuable: Hello
```

Distributed Ruby

Since we can serialize an object or a set of objects into a form suitable for out-of-process storage, we can use this capability for the *transmission* of objects from one process to another. Couple this capability with the power of networking, and *voilà*—you have a distributed object system. To save you the trouble of having to write the code, we suggest using Masatoshi Seki's Distributed Ruby library (drb), which is now available as a standard Ruby library.

Using drb, a Ruby process may act as a server, as a client, or as both. A drb server acts as a source of objects, while a client is a user of those objects. To the client, it appears that the objects are local, but in reality the code is still being executed remotely.

A server starts a service by associating an object with a given port. Threads are created internally to handle incoming requests on that port, so remember to join the drb thread before exiting your program:

```
require 'drb'
class TestServer
  def add(*args)
    args.inject {|n,v| n + v}
  end
end
server = TestServer.new
DRb.start_service('druby://localhost:9000', server)
DRb.thread.join   # Don't exit just yet!
```

A simple drb client simply creates a local drb object and associates it with the object on the remote server; the local object is a proxy:

```
require 'drb'
DRb.start_service()
obj = DRbObject.new(nil, 'druby://localhost:9000')
# Now use obj
puts "Sum is: #{obj.add(1, 2, 3)}"
```

The client connects to the server and calls the method add, which uses the magic of inject to sum its arguments. It returns the result, which the client prints out:

```
Sum is: 6
```

The initial nil argument to DRbObject indicates that we want to attach to a new distributed object. We could also use an existing object.

Ho hum, you say. This sounds like Java's RMI, or CORBA, or whatever. Yes, it is a functional distributed object mechanism—but it is written in just a few hundred lines of Ruby code. No C, nothing fancy, just plain old Ruby code. Of course, it has no naming service or trader service, or anything like you'd see in CORBA, but it is simple and reasonably fast. On my 2.5GHz Power Mac system, this sample code runs at about 1,300 remote message calls per second. And if you do need naming services, DRb has a ring server that might fit the bill.

And, if you like the look of Sun's JavaSpaces, the basis of the JINI architecture, you'll be interested to know that drb is distributed with a short module that does the same kind of thing. JavaSpaces is based on a technology called Linda. To prove that its Japanese author has a sense of humor, Ruby's version of Linda is known as *Rinda*.

Compile Time? Runtime? Anytime!

The important thing to remember about Ruby is that there isn't a big difference between "compile time" and "runtime." It's all the same. You can add code to a running process. You can redefine methods on the fly, change their scope from public to private, and so on. You can even alter basic types, such as Class and Object.

Once you get used to this flexibility, it is hard to go back to a static language such as C++ or even to a half-static language such as Java.

But then, why would you want to do that?