CHAPTER 30
Know Your Ruby Implementation

Back in the days when I programmed in C, every development team seemed to contain one engineer who knew something about assembly language. Since I did little or no coding in assembly language, this appeared to me to be a fairly useless skill—except that those assembly language-enabled engineers always seemed to have a unique perspective on how everything worked. Eventually I figured out that these two things were related: Having a basic understanding of the lower levels of your programming language helps you be a better programmer, even if you never actually work in those lower levels.

So, in this chapter we are going to look at the gears and pulleys of the major Ruby implementations. As we go along we will see there is something in this chapter for Rubyists who know something about C and for those who are bilingual in Java. Even if you are one of those people who think the word *Java* actually translates to "Run away! Run away!" or that the C programming language looks like a unfortunate editor incident, keep reading. You will be surprised at the insight you'll gain from even a naive inspection of a Ruby implementation. After all, there is no substitute for knowing your tool.

A Fistful of Rubies

Of course, to really know your tool you need to know *which* tool you are talking about. As I write this the Ruby community is going through a version transition: While much of our code still runs on Ruby 1.8, version 1.9 is out and stable, and there is a general upgrade movement in progress. To make things even more interesting, there are also three widely used implementations of Ruby, and not all of them support all versions of the language:

- First, there is the original Ruby implementation written by Yukihiro Matsumoto, the beloved father of Ruby and widely known as Matz. Rubyists call this founding implementation Matz's Ruby Interpreter, or MRI. MRI is written in C and supports Ruby 1.8.7.
- Next we have an implementation known as Yet Another Ruby VM, or YARV, which runs version 1.9.X of the language. YARV is slated to take over as the Ruby implementation once the 1.8 to 1.9 transition is complete. Like MRI, YARV is written in C.
- Finally, there is JRuby, an implementation of Ruby for the Java VM. JRuby supports version 1.8 and is rapidly closing in on complete support for Ruby 1.9.

Along with the "big three" implementations, there are a number of less well-known or less complete Ruby implementations. These include:

- Rubinius, which aspires to be a self-hosting implementation of Ruby. In simple English, the folks behind Rubinius hope to produce an implementation of Ruby that is completely written in Ruby.
- IronRuby, an implementation of Ruby for the .Net platform. IronRuby is to Microsoft's CLR what JRuby is to the Java virtual machine.
- Cardinal, a Ruby implementation that runs on Parrot, a VM that aspires to host a number of dynamic languages.

In this chapter I will focus primarily on the more popular Ruby implementations, MRI, YARV, and JRuby, although we will take one quick detour into the land of Rubinius.

MRI: An Enlightening Experience for the C Programmer

It all started with a man, a dream, and lots of C code. The man was, of course, Matz. The dream was of a simple, flexible, powerful programming language. The code was

MRI, the original Ruby implementation. You can get the MRI source code from www.ruby-lang.org. Unpack the archive into some suitable directory and you can see how it all began.

What you will see is a top-level directory containing about 55 C source files and headers, along with a bunch of subdirectories. Remarkably, the subdirectories are just the supporting code, things like the SSL library and the TK GUI interface. Those 55 top-level source files are the heart of the Matz Ruby interpreter.

If you can speak any C at all, looking through the Ruby 1.8 implementation can be an enlightening experience. As language interpreters go, MRI has a kind of plain-spoken eloquence. Obviously, the main job of any Ruby implementation is to turn your code into a living, breathing program. The first step of that process is to take the program text, break it up into the individual words and numbers and other assorted bits, and then reassemble those bits into a tree structure, the **abstract syntax tree**, or **AST**.

For example, if you fed MRI this trivial bit of Ruby code:

```
if denominator != 0
  quotient = numerator / denominator
end
```

You would end up with an AST that looks something like Figure 30-1.[1] The task of turning your Ruby code into the AST falls mainly on the code in `lex.c` and `parse.y`.[2]

Once MRI has the AST it executes it, and therefore your program, using the simplest technique imaginable. Starting at the root of the abstract syntax tree, MRI works its way down, recursively doing what the tree tells it to do. So in the previous example, MRI would start at the top of the tree, see that it had an `if` statement, know that it needed to evaluate the condition, realize that it needed to evaluate the variable `denominator`—and the constant 0—and . . . well, you get the picture. You can find the bulk of the code to do the evaluation in the aptly named `eval.c` file.

All of this basic parsing and executing code only occupies a handful of the core MRI source files.[3] Most of the rest of the 50 or so top-level files follow a very consistent

1. To keep things simple, Figure 30-1 is very schematic. For various practical reasons, an actual MRI AST tree is a bit more complex than I'm showing here—a bit, but not much.

2. The funny `.y` name comes from the fact that `parse.y` is actually the half-code/half-grammar input to a parser-generating tool called YACC.

3. To be fair, each of those files is pretty lengthy and full of some fairly sophisticated C.

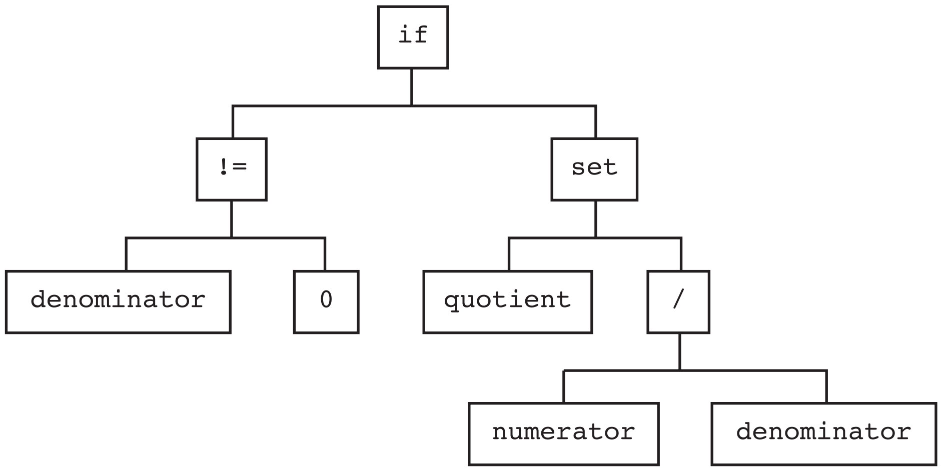

Figure 30-1 AST for a simple Ruby `if` statement

pattern: Each implements one or a few closely related Ruby classes. At the bottom of each file is an initialization function, with a name like `Init_Array` or `Init_Object` that defines each Ruby class and associates the class with its methods. So, if you ever wondered how the `Object` class gets defined, you can satisfy your curiosity by peeking into `object.c`. Toward the bottom of the file you will find `Init_Object`, and in there you will find the following three lines:

```
    rb_cObject = boot_defclass("Object", 0);
    rb_cModule = boot_defclass("Module", rb_cObject);
    rb_cClass =  boot_defclass("Class",  rb_cModule);
```

This is the ruby interpreter creating the `Object`, `Module`, and `Class` classes. The second parameter of `boot_defclass` method is the superclass of the new class; thus, we can see that the superclass of `Class` is `Module` and the superclass of `Module` is `Object`, and that `Object` doesn't have a superclass—at least not in version 1.8. We already knew this, but it's good to see it in black and white.

The `object.c` file is just full of other fascinating stuff. Here, for example, is the default implementation of the == method:

```
static VALUE
rb_obj_equal(obj1, obj2)
    VALUE obj1, obj2;
```

```
{
    if (obj1 == obj2) return Qtrue;
    return Qfalse;
}
```

Read the code carefully and you will discover a whole range of interesting tidbits about MRI. Notice how the two parameters of `rb_obj_equal` are both declared in the C code to be of type `VALUE`: Inside of MRI, when you have a reference to an object, you have a `VALUE`. It's also easy to deduce that `Qtrue` and `Qfalse` are the C versions of the Ruby `true` and `false` objects.

Looking a little further, you can find the code behind the `Array map!` method (also known as `collect!`), code that replaces the contents of an array with the values returned from invoking a block on each original array element:

```
static VALUE
rb_ary_collect_bang(ary)
    VALUE ary;
{
    long i;

    rb_ary_modify(ary);
    for (i = 0; i < RARRAY(ary)->len; i++) {
        rb_ary_store(ary, i, rb_yield(RARRAY(ary)->ptr[i]));
    }

    return ary;
}
```

Again, even without being a C hacker it is easy enough to follow the flow: Run through the array with a `for` loop, calling the code block (with `rb_yield`) with the value of each element and replacing each element (via `rb_ary_store`) with the result of the code block as you go.

YARV: MRI with a Byte Code Turbocharger

If you pull down and unpack YARV,[4] you will discover that YARV is the next generation of MRI. If you did take the time to look at MRI, you will also have no problem

4. Also to be found at www.ruby-lang.org.

finding your way around the YARV source code. For example, here is the YARV implementation of the `map!/collect!` method:

```
static VALUE
rb_ary_collect_bang(VALUE ary)
{
    long i;

    RETURN_ENUMERATOR(ary, 0, 0);
    rb_ary_modify(ary);
    for (i = 0; i < RARRAY_LEN(ary); i++) {
        rb_ary_store(ary, i, rb_yield(RARRAY_PTR(ary)[i]));
    }
    return ary;
}
```

The family relationship between this code and the MRI version we saw earlier is very striking.

Still, YARV is a real advance over MRI. One big difference is that MRI supports Ruby 1.8 while YARV has moved on to version 1.9. So if you look at the YARV code that creates the `Object` class, you find that it now has a Ruby 1.9-style `BasicObject` as a superclass:

```
    rb_cBasicObject = boot_defclass("BasicObject", 0);
    rb_cObject = boot_defclass("Object", rb_cBasicObject);
    rb_cModule = boot_defclass("Module", rb_cObject);
    rb_cClass =  boot_defclass("Class",  rb_cModule);
```

A more subtle difference between the two Ruby implementations is that when running Ruby code, YARV adds an extra step between the parse tree and execution. After parsing the Ruby source, YARV turns the resulting tree into a more or less flat list of byte codes. It is these byte codes that YARV actually executes. Thus, in YARV, our little Ruby fragment:

```
if denominator != 0
  quotient = numerator / denominator
end
```

Turns into a list of byte codes that looks something like this:

```
    0014 trace              1    (   4)
    0016 getdynamic         denominator, 0
    0019 putobject          0
    0021 opt_neq            <ic>, <ic>
    0024 branchunless       41
    0026 trace              1   (   5)
    0028 getdynamic         numerator, 0
    0031 getdynamic         denominator, 0
    0034 opt_div
    0035 dup
    0036 setdynamic         quotient, 0
    0039 leave              (   4)
    0040 pop
    0041 putnil             (   5)
```

Although the difference between executing byte codes and the original tree may seem esoteric, the effect is there for all the world to see: YARV and its byte codes are dramatically faster than MRI.

JRuby: Bending the "J" in the JVM

Unlike YARV, which comes with its own homegrown virtual machine, the challenge that faced the authors of JRuby was to adopt Ruby to the existing Java virtual machine, the JVM. Their answer was a Ruby that is completely implemented in, and integrated with, Java. Since running in the Java world presents different challenges from building a C application, the JRuby source code is something of a departure from MRI and YARV—a departure, but still recognizable.

For example, if you download the JRuby source archive[5] and unpack it, you will find what looks like a very conventional Java development project. Under the top-level directory, there is an ant `build.xml`[6] file and `bin` and `lib` and `src` directories. Go looking for the JRuby equivalent of `array.c` and you will find it in the most obvious (at least to a Java programmer) place: `src/org/jruby/RubyArray.java`. And in `RubyArray.java`

5. You can find it at http://jruby.org. The code here is from Jruby 1.4 RC1.

6. Ant is the traditional Java build tool, along the lines of rake.

you will find the JRuby implementation of the `map!/collect!` method, complete with a helpful comment pointing us back to the original C function:

```
/** rb_ary_collect_bang
 *
 */
public RubyArray collectBang(
    ThreadContext context, Block block) {

    if (!block.isGiven())
        throw context.getRuntime().
            newLocalJumpErrorNoBlock();

    modify();
    for (int i = 0, len = realLength; i < len; i++) {
        store(i, block.yield(context, values[begin + i]));
    }
    return this;
}
```

In C or Java,[7] `collect!` is pretty much the same: Run through the array, replacing values as you go.

Rubinius

If your head is swimming from all of this C and Java in a Ruby book but you are still interested in Ruby implementations, let me invite you to have a look at Rubinius. Rubinius (http://rubini.us) is a Ruby implementation whose ambition is to be self-hosting: The goal of Rubinius is to implement Ruby *in Ruby*. Although Rubinius is not really finished, it is a very informative source of Ruby implementation knowledge. Here, for example, is the Rubinius version of our favorite array rewriting method, this time called `map!`:

```
# Replaces each element in self with the return value
# of passing that element to the supplied block.
def map!
  Ruby.check_frozen
```

7. I did edit the code a bit to make it fit on the page.

```
    return to_enum(:map!) unless block_given?

    i = -1
    each { |x| self[i+=1] = yield(x) }

    self
  end
```

Life is a bit easier if you get to use Ruby as your implementation language.

In the Wild

One of the best ways to gain some insight into how your Ruby implementation works is to look into how to extend it. Every Ruby implementation allows you to add features to the native implementation. Since you need some understanding of the implementation in order to extend it, the "how to extend Ruby" documentation is a great source of insight into how that Ruby works. Both the MRI and YARV source code come with a `README.EXT` file[8] that does a good job of explaining the basics.

The JRuby project has an entire section of its website devoted to explaining how JRuby works. As I write this the URL for this documentation is `www.kenai.com/projects/jruby/pages/Internals`. It is well worth a look.

Staying Out of Trouble

There is really only one danger with knowing your language implementation, and it's a very easily avoidable risk. The problem arises when you think too hard about what's going on under the covers: "Gee, an awful lot of C . . . or Java . . . or whatever gets fired for every Ruby class I write and every method I call. That has got to be slow. Maybe I should start writing fewer classes or longer methods or. . . ."

Don't go there. A half century of software engineering says that you should write the code first and worry about making it faster only if it is too slow. Donald Knuth is right: Premature optimization is the root of all evil. Don't let a little bit of insight into your Ruby implementation blind you to this fundamental truth. Your Ruby was fast enough yesterday, before you started poking around in its innards. It is still fast enough.

8. Yes, that suffix is `.EXT`, as in "extension," not `.TXT`.

Wrapping Up

In this chapter we took a very rapid "If it's Tuesday this must be YARV"-style tour of Ruby implementations. We saw how MRI was the original, C-based implementation, now being eclipsed by YARV. We also saw that JRuby is bringing Ruby to the JVM. We even visited briefly with Rubinius. Mostly, though, we spent our time trying to get a feel for how these implementations actually work, how MRI and YARV turn C into Ruby while JRuby performs the same miracle for Java.

The ironic thing about this chapter is that many, perhaps most, of us got into Ruby so we would never have to write C or Java ever again. And yet here we are, back again. Ironic or not, the simple fact is that the better you understand Ruby, the less it seems like magic, and the better Ruby programmer you will be.