4

CONTROL STRUCTURES AND METHOD DISPATCH

In Chapter 3 I explained how YARV uses a stack while executing its instruction set and how it can access variables locally or dynamically. Controlling the flow of execution is another fundamental requirement for any programming language, and Ruby has a rich set of control structures, too. How does YARV implement control structures?

Like Ruby, YARV has its own control structures, albeit at a much lower level. Instead of `if` or `unless` statements, YARV uses two low-level instructions called `branchif` and `branchunless`. Instead of using control structures such as `while...end` or `until...end` loops, YARV has a single low-level function called `jump` that allows it to change the program counter and move through your compiled program. By combining the `branchif` or `branchunless` instruction with the `jump` instruction, YARV can execute most of Ruby's simple control structures.

When your code calls a method, YARV uses the `send` instruction. This process is known as *method dispatch*. You can consider `send` to be another one of Ruby's control structures—the most complex and sophisticated one of all.

In this chapter we'll learn more about YARV by exploring how it controls execution flow in your program. We'll also look at the method dispatch process as we learn how Ruby categorizes methods into types, calling each method type differently.

ROADMAP

How Ruby Executes an if Statement

In order to understand how YARV controls execution flow, let's see how the `if...else` statement works. The left side of Figure 4-1 shows a simple Ruby script that uses both `if` and `else`. On the right side of the figure, you can see the corresponding snippet of compiled YARV instructions. Reading the YARV instructions, you see that Ruby follows a pattern for implementing the `if...else` statement. It goes like this:

1. Evaluate condition
2. Jump to false code if condition is false
3. True code; jump past false code
4. False code

```
i = 0
if i < 10
  puts "small"
else
  puts "large"
end
puts "done"
```

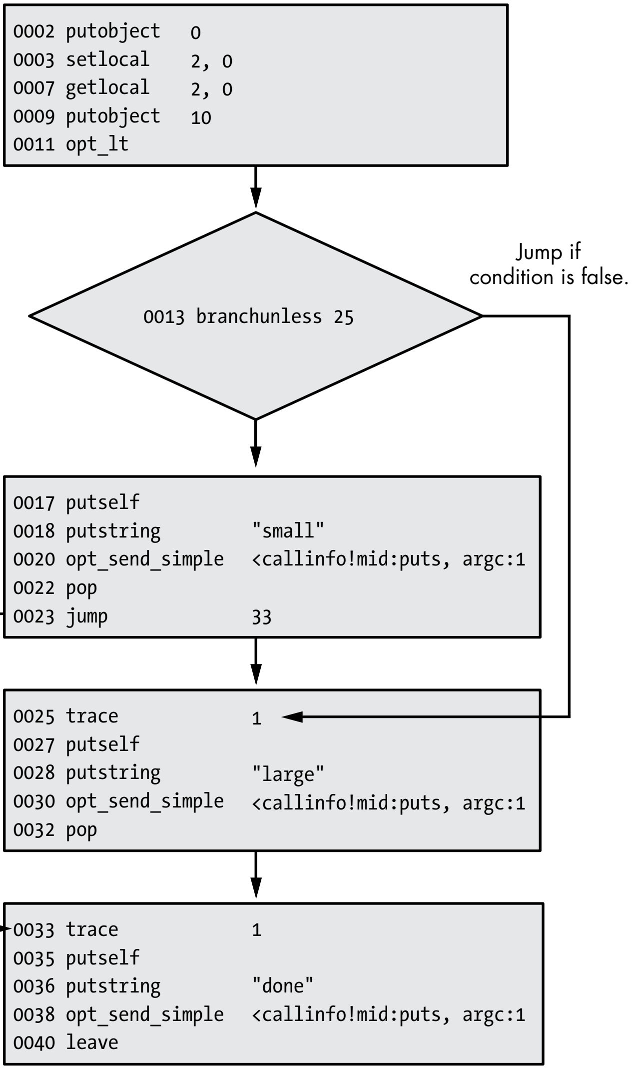
```
0000 trace            1
0002 putobject        0
0003 setlocal         2, 0
0005 trace            1
0007 getlocal         2, 0
0009 putobject        10
0011 opt_lt           <callinfo!mid:<, argc:1
0013 branchunless     25
0015 trace            1
0017 putself
0018 putstring        "small"
0020 opt_send_simple  <callinfo!mid:puts, argc:1
0022 pop
0023 jump             33
0025 trace            1
0027 putself
0028 putstring        "large"
0030 opt_send_simple  <callinfo!mid:puts, argc:1
0032 pop
0033 trace            1
0035 putself
0036 putstring        "done"
0038 opt_send_simple  <callinfo!mid:puts, argc:1
0040 leave
```

Figure 4-1: How Ruby compiles an `if...else` statement

This pattern should be a bit easier to follow in the flowchart shown in Figure 4-2 on the next page. The `branchunless` instruction in the center of the figure is the key to how Ruby implements `if` statements. It works as follows:

1. Ruby evaluates the condition of the `if` statement, `i < 10`, using the `opt_lt` (optimized less-than) instruction. This evaluation leaves either a true or false value on the stack.
2. `branchunless` jumps down to the `else` code if the condition is false. That is, it "branches unless" the condition is true. Ruby uses `branchunless`, not `branchif`, for `if...else` conditions because the positive case is compiled to appear right after the condition code. Therefore, YARV needs to jump if the condition is false.
3. If the condition is true, Ruby does not branch and just continues to execute the positive case code. Once it's finished, it jumps down to the instructions following the `if...else` statement using the `jump` instruction.
4. Whether or not it branches, Ruby continues to execute the subsequent code.

YARV implements the `unless` statement similarly to how it implements `if`, except that the positive and negative code snippets are in reverse order. For looping control structures like `while...end` and `until...end`, YARV uses the `branchif` instruction instead, but the idea is the same: Calculate the loop condition, execute `branchif` to jump as necessary, and then use `jump` statements to implement the loop.



```
0002 putobject   0
0003 setlocal    2, 0
0007 getlocal    2, 0
0009 putobject   10
0011 opt_lt
```

```
0013 branchunless 25
```

```
0017 putself
0018 putstring        "small"
0020 opt_send_simple  <callinfo!mid:puts, argc:1
0022 pop
0023 jump             33
```

```
0025 trace            1
0027 putself
0028 putstring        "large"
0030 opt_send_simple  <callinfo!mid:puts, argc:1
0032 pop
```

```
0033 trace            1
0035 putself
0036 putstring        "done"
0038 opt_send_simple  <callinfo!mid:puts, argc:1
0040 leave
```

Figure 4-2: This flowchart shows the pattern Ruby uses to compile `if...else` *statements.*

Jumping from One Scope to Another

One of the challenges YARV has in implementing some control structures is that, as with dynamic variable access, Ruby can jump from one scope to another. For example, `break` can be used to exit a simple loop like the one in Listing 4-1.

```
i = 0
while i<10
  puts i
  i += 1
  break
end
```

Listing 4-1: `break` *used to exit a simple loop*

And it can also be used to exit a block iteration, like the one in Listing 4-2.

```
10.times do |n|
  puts n
  break
end
puts "continue from here"
```

Listing 4-2: `break` *used to exit a block*

In the first listing, YARV can exit the `while` loop using simple `jump` instructions. But exiting a block like the one in the second listing is not so simple: In this case, YARV needs to jump to the parent scope and continue execution after the call to `10.times`. How does YARV know where to jump to? And how does it adjust both its internal stack and your Ruby call stack in order to continue execution properly in the parent scope?

To implement jumping from one place to another in the Ruby call stack (that is, outside the current scope), Ruby uses the `throw` YARV instruction. This instruction resembles the Ruby `throw` method: It sends, or throws, the execution path back up to a higher scope. For example, Figure 4-3 shows how Ruby compiles Listing 4-2, with the block containing a `break` statement. The Ruby code is on the left, and the compiled version is on the right.

```
10.times do |n|
  puts n
  break
end
puts "continue from here"
```

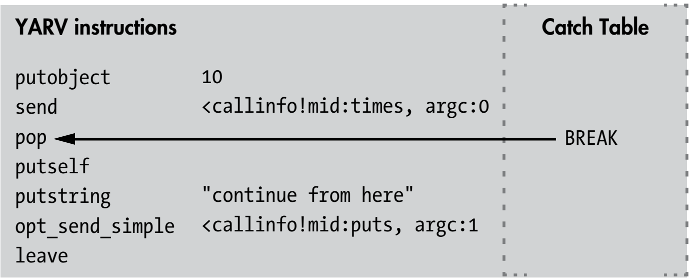
```
putself
getlocal        2, 0
opt_send_simple <callinfo!mid:puts, argc:1
pop
putnil
throw           2
leave
```

```
putobject       10
send            <callinfo!mid:times, argc:0
pop
putself
putstring       "continue from here"
opt_send_simple <callinfo!mid:puts, argc:1
leave
```

Figure 4-3: How Ruby compiles a `break` *statement used inside a block*

Catch Tables

At the top right of Figure 4-3, the `throw 2` in the compiled code for the block throws an exception at the YARV instruction level using a *catch table*, or a table of pointers that may be attached to any YARV code snippet. Conceptually, a catch table might look like Figure 4-4.



```
putobject        10
send             <callinfo!mid:times, argc:0
pop
putself
putstring        "continue from here"
opt_send_simple  <callinfo!mid:puts, argc:1
leave
```

Figure 4-4: Each snippet of YARV code can contain a catch table.

This catch table contains just a single pointer to the `pop` statement, where execution would continue after an exception. Whenever you use a `break` statement in a block, Ruby compiles the `throw` instruction into the block's code. And whenever you call a block or write a loop using `while`, `for`, and so on, Ruby adds the `BREAK` entry into the parent scope's catch table. If you wrote a nested loop, Ruby would add the `BREAK` entry to the outer loop scope's catch table.

Later, when YARV executes the `throw` instruction, it checks to see whether there's a catch table containing a break pointer for the current YARV instruction sequence, as shown in Figure 4-5.

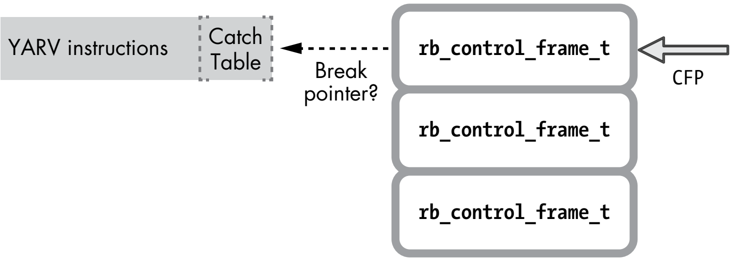


Figure 4-5: While executing a `throw` instruction, YARV starts iterating down the Ruby call stack.

If it doesn't find a catch table, Ruby starts to iterate down through the stack of `rb_control_frame_t` structures in search of a catch table containing a break pointer, as shown in Figure 4-6.

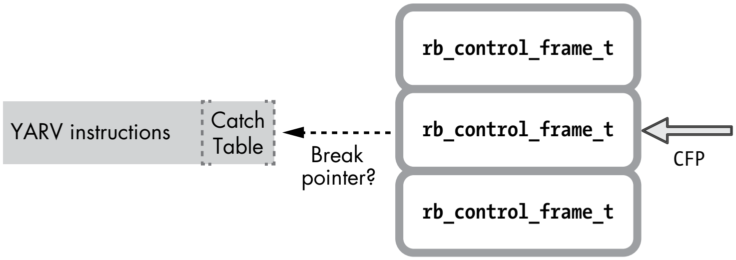


Figure 4-6: Ruby continues to iterate down the call stack looking for a catch table with a break pointer.

As you can see in Figure 4-7, Ruby continues to iterate until it finds a catch table with a break pointer.

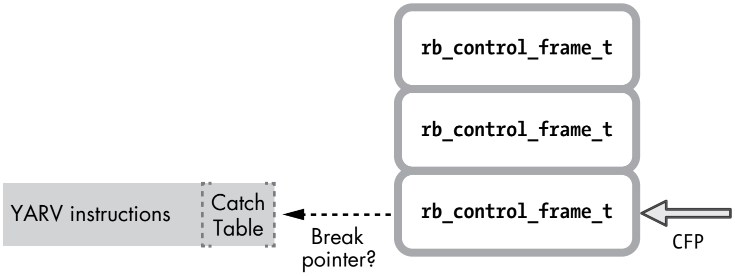
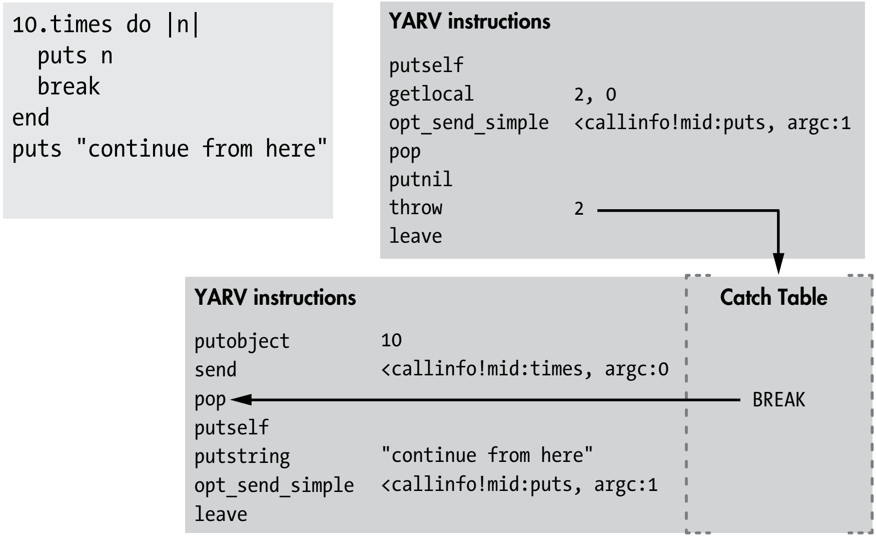


Figure 4-7: Ruby keeps iterating until it finds a catch table with a break pointer or reaches the end of the call stack.

In this simple example, there is only one level of block nesting, so Ruby finds the catch table and break pointer after just one iteration, as shown in Figure 4-8.



```
10.times do |n|
  puts n
  break
end
puts "continue from here"
```

YARV instructions

```
putself
getlocal         2, 0
opt_send_simple  <callinfo!mid:puts, argc:1
pop
putnil
throw            2
leave
```

YARV instructions

```
putobject        10
send             <callinfo!mid:times, argc:0
pop
putself
putstring        "continue from here"
opt_send_simple  <callinfo!mid:puts, argc:1
leave
```

Figure 4-8: Ruby finds a catch table with a break pointer.

Once Ruby finds the catch table pointer, it resets both the Ruby call stack (the `CFP` pointer) and the internal YARV stack to reflect the new program execution point. YARV continues to execute your code from there—that is, it resets the internal `PC` and `SP` pointers as needed.

NOTE *Ruby uses a process similar to raising and rescuing an exception internally in order to implement a very commonly used control structure: the* `break` *keyword. In other words, what in more verbose languages is an exceptional occurrence becomes in Ruby a common, everyday action. Ruby has wrapped up a confusing, unusual syntax—the raising/rescuing of exceptions—into a simple keyword,* `break`*, and made it very easy to understand and use. (Of course, Ruby needs to use exceptions because of the way blocks work. On the one hand, they're like separate functions or subroutines, but on the other, they're just part of the surrounding code.)*

Other Uses for Catch Tables

The `return` keyword is another ordinary Ruby control structure that also uses catch tables. Whenever you call `return` from inside a block, Ruby raises an internal exception that it rescues with a catch table pointer in the same way it does when you call `break`. In fact, `break` and `return` are implemented with the same YARV instructions with one exception: For `return`, Ruby passes a 1 to the `throw` instruction (for example, `throw 1`), while for `break`, it passes a 2 (`throw 2`). The `return` and `break` keywords are really two sides of the same coin.

Besides `break`, Ruby uses the catch table to implement the control structures `rescue`, `ensure`, `retry`, `redo`, and `next`. For example, when you explicitly raise an exception in your Ruby code using the `raise` keyword, Ruby implements the `rescue` block using the catch table, but with a `rescue` pointer. The catch table is simply a list of event types that can be caught and handled by that sequence of YARV instructions, just as you would use a `rescue` block in your Ruby code.

Experiment 4-1: Testing How Ruby Implements for Loops Internally

I had always known that Ruby's `for` loop control structure worked essentially the same way as a block with the `each` method of the `Enumerable` module. That is, I knew that this code:

```
for i in 0..5
  puts i
end
```

worked like this code:

```
(0..5).each do |i|
  puts i
end
```

But I never suspected that internally Ruby actually implements `for` loops using `each`! In other words, Ruby has no `for` loop control structure. Instead, the `for` keyword is really just syntactical sugar for calling `each` with a range.

To prove this, simply inspect the YARV instructions produced by Ruby when you compile a `for` loop. In Listing 4-3, let's use the same `RubyVM::InstructionSequence.compile` method to display the YARV instructions.

```
code = <<END
for i in 0..5
  puts i
end
END
puts RubyVM::InstructionSequence.compile(code).disasm
```

Listing 4-3: This code will display how Ruby compiles a `for` *loop.*

Running this code gives the output shown in Listing 4-4.

```
== disasm: <RubyVM::InstructionSequence:<compiled>@<compiled>>==========
== catch table
| catch type: break  st: 0002 ed: 0006 sp: 0000 cont: 0006
|------------------------------------------------------------------------
local table (size: 2, argc: 0 [opts: 0, rest: -1, post: 0, block: -1] s1)
[ 2] i
0000 trace            1                                               (   1)
0002 putobject        0..5
0004 send             <callinfo!mid:each, argc:0, block:block in <compiled>>
0006 leave
== disasm: <RubyVM::InstructionSequence:block in <compiled>@<compiled>>=
== catch table
| catch type: redo   st: 0004 ed: 0015 sp: 0000 cont: 0004
| catch type: next   st: 0004 ed: 0015 sp: 0000 cont: 0015
|------------------------------------------------------------------------
local table (size: 2, argc: 1 [opts: 0, rest: -1, post: 0, block: -1] s3)
[ 2] ?<Arg>
0000 getlocal_OP__WC__0 2                                             (   3)
0002 setlocal_OP__WC__1 2                                             (   1)
0004 trace            256
0006 trace            1                                               (   2)
0008 putself
0009 getlocal_OP__WC__1 2
0011 opt_send_simple  <callinfo!mid:puts, argc:1, FCALL|ARGS_SKIP>
0013 trace            512                                             (   3)
0015 leave
```

Listing 4-4: The output generated by Listing 4-3

Figure 4-9 shows the Ruby code on the left and YARV instructions on the right. (I've removed some of the technical details, like the `trace` statements, in order to simplify things a bit.)

```
for i in 0..5
  puts i
end
```

```
putobject   0..5
send        <callinfo!mid:each, argc:0
leave
```

```
getlocal         2, 0
setlocal         2, 1
putself
getlocal         2, 1
opt_send_simple  <callinfo!mid:puts, argc:1
leave
```

Figure 4-9: A simplified display of the YARV instructions in Listing 4-4

Notice that there are two separate YARV code blocks: The outer scope calls `each` on the range `0..5`, and an inner block makes the `puts i` call. The `getlocal 2, 0` instruction in the inner block loads the implied block parameter value, and the `setlocal` instruction that follows saves it into the local variable `i`, located back in the parent scope using dynamic variable access.

In effect, Ruby has automatically done the following:

- Converted the `for i in 0..5` code into `(0..5).each do`
- Created a block parameter to hold each value in the range
- Copied the block parameter, or the iteration counter, back into the local variable `i` each time around the loop

The send Instruction: Ruby's Most Complex Control Structure

We've seen how YARV controls the execution flow of our Ruby program using low-level instructions such as `branchunless` and `jump`. However, the most commonly used and important YARV instruction for controlling Ruby program execution flow is the `send` instruction. The `send` instruction tells YARV to jump to another method and start executing it.

Method Lookup and Method Dispatch

How does `send` work? How does YARV know which method to call, and how does it actually call the method? Figure 4-10 shows a high-level overview of the process.

This seems very simple, but the algorithm Ruby uses to find and call the target method is actually very complex. First, in *method lookup*, Ruby searches for the method your code actually should call. This involves looping through the classes and modules that make up the receiver object.

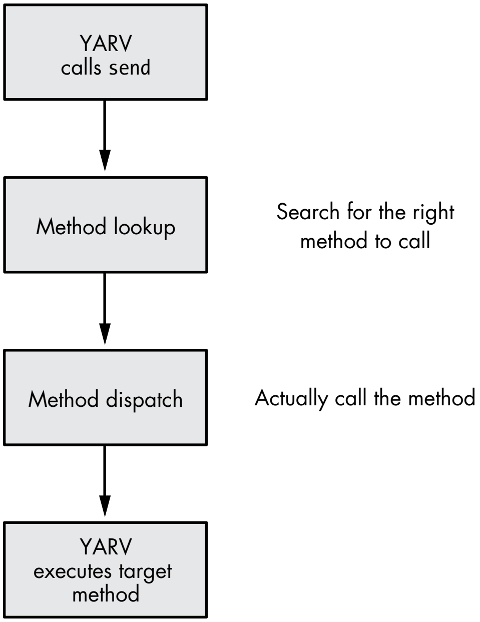


Figure 4-10: Ruby uses method lookup to find which method to call, and uses method dispatch to call it.

Once Ruby finds the method your code is trying to call, it uses *method dispatch* to actually execute the method call. This involves preparing the arguments to the method, pushing a new frame onto YARV's internal stack, and changing YARV's internal registers in order to actually start executing the target method. Like method lookup, method dispatch is a complex process because of the way Ruby categorizes your methods.

During the rest of this chapter I'll discuss the method dispatch process. We'll see how method lookup works in Chapter 6, once we have learned more about how Ruby implements objects, classes, and modules.

Eleven Types of Ruby Methods

Internally, Ruby categorizes your methods into 11 different types! During the method dispatch process, Ruby determines which type of method your code is trying to call. It then calls each type of method differently depending on its type, as shown in Figure 4-11.

Most methods—including all methods you write with Ruby code in your program—are referred to as ISEQ, or *instruction sequence* methods, by YARV's internal source code because Ruby compiles your code into a series of YARV bytecode instructions. But internally, YARV uses 10 other method types as well. These other method types are required because Ruby needs to call certain methods in a special way in order to speed up method dispatch, because these methods are implemented with C code or for various internal, technical reasons.

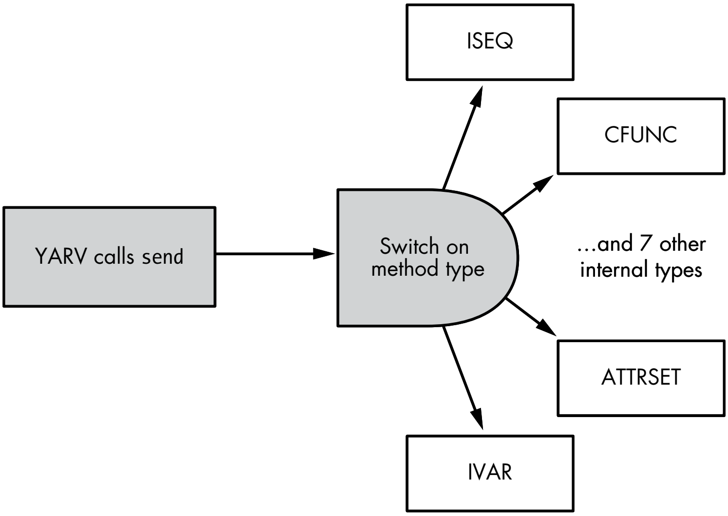


Figure 4-11: While executing send, YARV switches on the type of the target method.

Here's a quick description of all 11 method types. We'll explore some of these in more detail in the following sections.

ISEQ A normal method that you write using Ruby code, this is the most common method type. ISEQ stands for *instruction sequence.*

CFUNC Using C code included directly inside the Ruby executable, these are the methods that Ruby implements rather than you. CFUNC stands for *C function.*

ATTRSET A method of this type is created by the `attr_writer` method. ATTRSET stands for *attribute set.*

IVAR Ruby uses this method type when you call `attr_reader`. IVAR stands for *instance variable.*

BMETHOD Ruby uses this method type when you call `define_method` and pass in a proc object. Because the method is represented internally by a proc, Ruby needs to handle this method type in a special way.

ZSUPER Ruby uses this method type when you set a method to be public or private in a particular class or module when it was actually defined in some superclass. This method is not commonly used.

UNDEF Ruby uses this method type internally when it needs to remove a method from a class. Also, if you remove a method using `undef_method`, Ruby creates a new method of the same name using the UNDEF method type.

NOTIMPLEMENTED Like UNDEF, Ruby uses this method type to mark certain methods as not implemented. This is necessary, for example, when you run Ruby on a platform that doesn't support a particular operating system call.

OPTIMIZED Ruby speeds up some important methods using this type, like the `Kernel#send` method.

MISSING Ruby uses this method type if you ask for a method object from a module or class using `Kernel#method` and the method is missing.

REFINED Ruby uses this method type in its implementation of refinements, a new feature introduced in version 2.0.

Now let's focus on the most important and frequently used method types: ISEQ, CFUNC, ATTRSET, and IVAR.

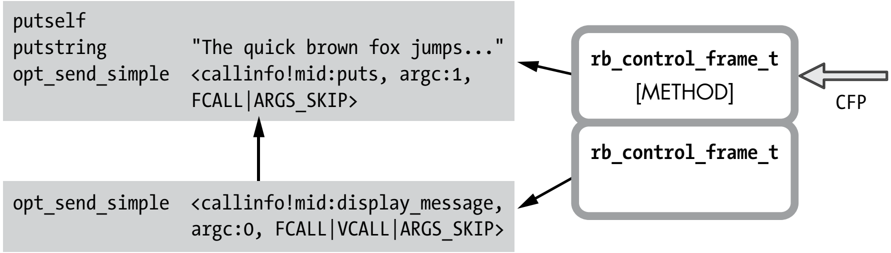
Calling Normal Ruby Methods

Most methods in your Ruby code are identified by the constant `VM_METHOD_TYPE_ISEQ` inside Ruby's source code. This means that they consist of a sequence of YARV instructions.

You define standard Ruby methods in your code with the `def` keyword, as shown here.

```
def display_message
  puts "The quick brown fox jumps over the lazy dog."
end
display_message
```

`display_message` is a standard method because it's created using the `def` keyword followed by normal Ruby code. Figure 4-12 shows how Ruby calls the `display_message` method.



```
putself
putstring        "The quick brown fox jumps..."
opt_send_simple  <callinfo!mid:puts, argc:1,
                 FCALL|ARGS_SKIP>
```

```
opt_send_simple  <callinfo!mid:display_message,
                 argc:0, FCALL|VCALL|ARGS_SKIP>
```

Figure 4-12: A normal method is comprised of YARV instructions.

On the left are two snippets of YARV code: the calling code at the bottom and the target method at the top. On the right you can see that Ruby created a new stack frame using a new `rb_control_frame_t` structure, set to type METHOD.

The key idea in Figure 4-12 is that both the calling code and the target method are comprised of YARV instructions. When you call a standard method, YARV creates a new stack frame and then starts executing the instructions in the target method.

Preparing Arguments for Normal Ruby Methods

When Ruby compiles your code, it creates a table of local variables and arguments for each method. Each argument listed in the local table is labeled as standard (`<Arg>`) or as one of a few different special types, such as block, optional, and so on. Ruby records the type of each method's arguments in this way so it can tell whether any additional work is required when your code calls the method. Listing 4-5 shows a single Ruby method that uses each type of argument.

```
def five_argument_types(a, b = 1, *args, c, &d)
  puts "Standard argument #{a.inspect}"
  puts "Optional argument #{b.inspect}"
  puts "Splat argument array #{args.inspect}"
  puts "Post argument #{c.inspect}"
  puts "Block argument #{d.inspect}"
end

five_argument_types(1, 2, 3, 4, 5, 6) do
  puts "block"
end
```

Listing 4-5: Ruby's argument types (argument_types.rb)

Listing 4-6 shows the result when we call the example method with the numbers 1 through 6 and a block.

```
$ ruby argument_types.rb
Standard argument 1
Optional argument 2
Splat argument array [3, 4, 5]
Post argument 6
Block argument #<Proc:0x007ff4b2045ac0@argument_types.rb:9>
```

Listing 4-6: The output generated by Listing 4-5

To make this behavior possible, YARV does some additional processing on each type of argument when you call a method:

Block arguments When you use the `&` operator in an argument list, Ruby needs to convert the provided block into a proc object.

Optional arguments Ruby adds additional code to the target method when you use an optional argument with a default value. This code sets the default value into the argument. When you later call a method with an optional argument, YARV resets the program counter or PC register to skip this added code when a value is provided.

Splat argument array For these, YARV creates a new array object and collects the provided argument values into it. (See the array `[3, 4, 5]` in Listing 4-6.)

Standard and post arguments Because these need no special treatment, YARV has no additional work to do.

Then there are keyword arguments. Whenever Ruby calls a method that uses keyword arguments, YARV has even more work to do. ("Experiment 4-2: Exploring How Ruby Implements Keyword Arguments" on page 99 explores this in more detail.)

Calling Built-In Ruby Methods

Many of the methods built into the Ruby language are CFUNC methods (`VM_METHOD_TYPE_CFUNC` in Ruby's C source code). Ruby implements these using C code rather than Ruby code. For example, consider the `Integer#times` method from "Executing a Call to a Block" on page 61. The `Integer` class is included in the Ruby interpreter, and the `times` method is implemented by C code in the file *numeric.c*.

The classes you use every day have many examples of CFUNC methods, such as `String`, `Array`, `Object`, `Kernel`, and so on. For example, the `String#upcase` method is implemented by C code in *string.c*, and `Struct#each` is implemented by C code in *struct.c*.

When Ruby calls a built-in CFUNC method, it doesn't need to prepare the method arguments in the same way it does with normal ISEQ methods; it simply creates a new stack frame and calls the target method, as shown in Figure 4-13.

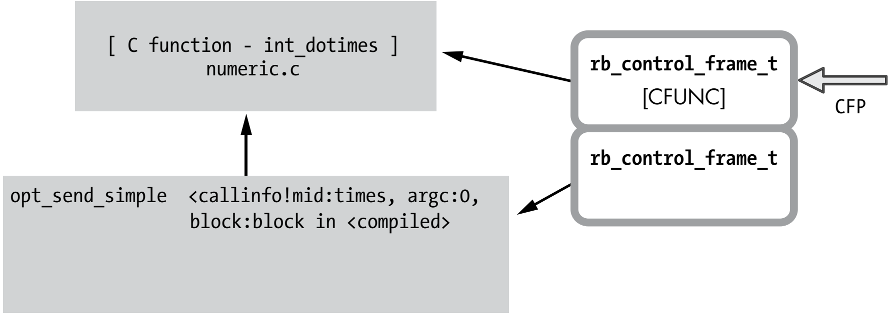


Figure 4-13: Ruby implements CFUNC methods using C code in one of Ruby's C source files.

As we saw with ISEQ methods in Figure 4-12, calling a CFUNC method involves creating a new stack frame. This time, however, Ruby uses a `rb_control_frame_t` structure with type CFUNC instead.

Calling attr_reader and attr_writer

Ruby uses two special method types, IVAR and ATTRSET, to speed up the process of accessing and setting instance variables in your code. Before I explain what these method types mean and how method dispatch works with them, have a look at Listing 4-7, which retrieves and sets the value of an instance variable.

```
  class InstanceVariableTest
❶   def var
      @var
    end
❷   def var=(val)
      @var = val
    end
  end
```

Listing 4-7: A Ruby class with an instance variable and accessor methods

In this listing, the class `InstanceVariableTest` contains an instance variable, `@var`, and two methods, `var` ❶ and `var=` ❷. Because I wrote these methods using Ruby code, both will be standard Ruby methods with the type set to `VM_METHOD_TYPE_ISEQ`. As you can see, they allow you to get or set the value of `@var`.

Ruby actually provides a shortcut for creating these methods: `attr_reader` and `attr_writer`. The following code shows a shorter way of writing the same class, using these shortcuts.

```
class InstanceVariableTest
  attr_reader :var
  attr_writer :var
end
```

Here, `attr_reader` automatically defines the same `var` method, and `attr_writer` automatically defines the `var=` method, both from Listing 4-7.

And here's an even simpler, more concise way of defining the same two methods using `attr_accessor`.

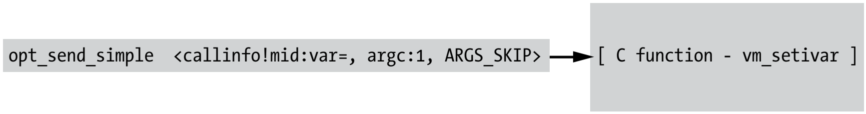
```
class InstanceVariableTest
  attr_accessor :var
end
```

As you can see, `attr_accessor` is shorthand for calling `attr_reader` and `attr_writer` together for the same instance variable.

Method Dispatch Optimizes attr_reader and attr_writer

Since Ruby developers use `attr_reader` and `attr_writer` so often, YARV uses two special method types, IVAR and ATTRSET, to speed up method dispatch and make your program run faster.

Let's begin with the ATTRSET method type. Whenever you define a method using `attr_writer` or `attr_accessor`, Ruby marks the generated method with the `VM_METHOD_TYPE_ATTRSET` method type internally. When Ruby executes the code and calls the method, it uses a C function, `vm_setivar`, to set the instance variable in a fast, optimized manner. Figure 4-14 shows how YARV calls the generated `var=` method to set `var`.

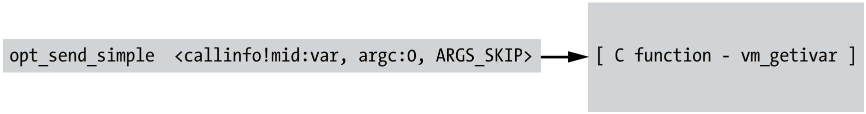


```
opt_send_simple  <callinfo!mid:var=, argc:1, ARGS_SKIP> ──▶ [ C function - vm_setivar ]
```

Figure 4-14: `VM_METHOD_TYPE_ATTRSET` *methods call* `vm_setivar` *directly.*

Notice that this figure is similar to Figure 4-13. In both cases, Ruby calls an internal C function when executing our code. But notice in Figure 4-14 that when executing an ATTRSET method, Ruby doesn't even create a new stack frame. It doesn't need to because the method is so short and simple. Also, because the generated `var=` method will never raise an exception, Ruby doesn't need a new stack frame to display in error messages. The `vm_setivar` C function can very quickly set the value and return.

The IVAR method type works similarly. When you define a method using `attr_reader` or `attr_accessor`, Ruby marks the generated method with the `VM_METHOD_TYPE_IVAR` method type internally. When it executes IVAR methods, Ruby calls an internal C function called `vm_getivar` to get and return the instance variable's value quickly, as shown in Figure 4-15.



```
opt_send_simple  <callinfo!mid:var, argc:0, ARGS_SKIP> ──▶ [ C function - vm_getivar ]
```

Figure 4-15: `VM_METHOD_TYPE_IVAR` *methods call* `vm_getivar` *directly.*

Here, the `opt_send_simple` YARV instruction on the left calls the `vm_getivar` C function on the right. As in Figure 4-14, when calling `vm_setivar`, Ruby doesn't need to create a new stack frame or execute YARV instructions. It simply returns the value of `var` immediately.

Experiment 4-2: Exploring How Ruby Implements Keyword Arguments

Beginning with Ruby 2.0, you can specify labels for method arguments. Listing 4-8 shows a simple example.

```
❶ def add_two(a: 2, b: 3)
    a+b
  end

❷ puts add_two(a: 1, b: 1)
   => 2
```

Listing 4-8: A simple example of using keyword arguments

We use the labels `a` and `b` for the keyword arguments to `add_two` ❶. When we call the function ❷, we get the result 2. I hinted in Chapter 2 that Ruby uses a hash to implement keyword arguments. Let's prove this is the case using Listing 4-9.

```
  class Hash
❶   def key?(val)
❷     puts "Looking for key #{val}"
      false
    end
  end

  def add_two(a: 2, b: 3)
    a+b
  end

  puts add_two (a: 1, b: 1)
```

Listing 4-9: Demonstrating that Ruby uses a hash to implement keyword arguments

We override the `key?` method ❶ of the `Hash` class, which displays a message ❷ and then returns `false`. Here's the output we get when we run Listing 4-9.

```
Looking for key a
Looking for key b
5
```

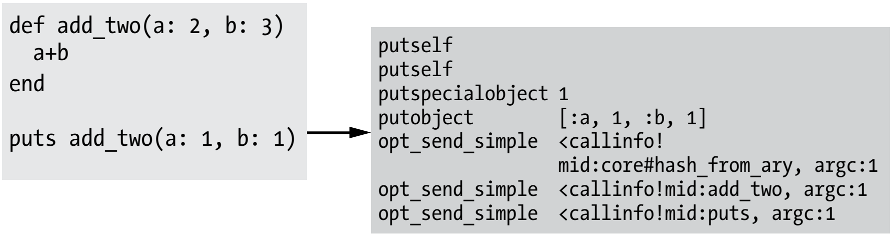
As you can see, Ruby is calling `Hash#key?` twice: once to find the key `a` and a second time to find the key `b`. For some reason, Ruby has created a hash even though we never used a hash in the code. Also, Ruby is now ignoring the values we pass into `add_two`. Instead of 2, we get 5. It looks like Ruby is using the default values for `a` and `b`, not the values we provided. Why did Ruby create a hash, and what does it contain? And why did Ruby ignore my parameter values when I overrode `Hash#key?`?

To learn how Ruby implements keyword arguments and to explain the results we see running Listing 4-9, we can examine the YARV instructions generated by Ruby's compiler for `add_two`. Running Listing 4-10 displays the YARV instructions that correspond to Listing 4-9.

```
code = <<END
def add_two(a: 2, b: 3)
  a+b
end

puts add_two(a: 1, b: 1)
END

puts RubyVM::InstructionSequence.compile(code).disasm
```

Listing 4-10: Displaying the YARV instructions for the code in Listing 4-9

Figure 4-16 shows a simplified version of the output generated by Listing 4-10.



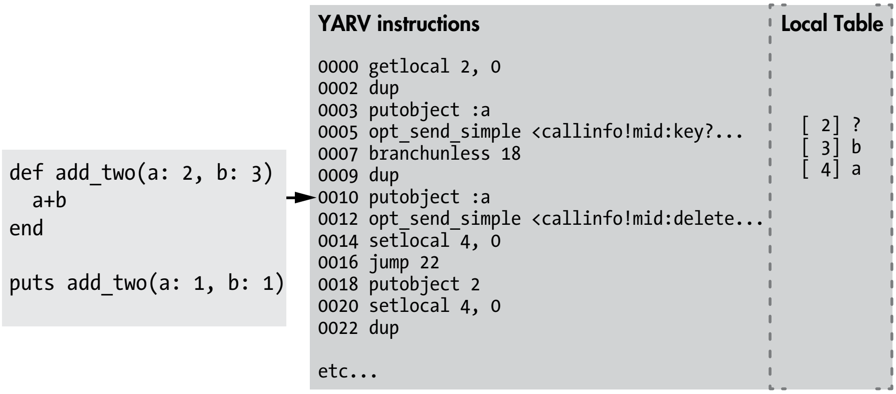
```
def add_two(a: 2, b: 3)
  a+b
end

puts add_two(a: 1, b: 1)
```

```
putself
putself
putspecialobject 1
putobject        [:a, 1, :b, 1]
opt_send_simple  <callinfo!
                 mid:core#hash_from_ary, argc:1
opt_send_simple  <callinfo!mid:add_two, argc:1
opt_send_simple  <callinfo!mid:puts, argc:1
```

Figure 4-16: Part of the output generated by Listing 4-10

On the right of Figure 4-16, you can see that Ruby first pushes an array onto the stack: `[:a, 1, :b, 1]`. Next, it calls the internal C function `hash_from_ary`, which we can guess will convert the `[:a, 1, :b, 1]` array into a hash. Finally, Ruby calls the `add_two` method to add the numbers and the `puts` method to display the result.

Now let's look at the YARV instructions for the `add_two` method itself, shown in Figure 4-17.



```
def add_two(a: 2, b: 3)
  a+b
end

puts add_two(a: 1, b: 1)
```

YARV instructions

```
0000 getlocal 2, 0
0002 dup
0003 putobject :a
0005 opt_send_simple <callinfo!mid:key?...
0007 branchunless 18
0009 dup
0010 putobject :a
0012 opt_send_simple <callinfo!mid:delete...
0014 setlocal 4, 0
0016 jump 22
0018 putobject 2
0020 setlocal 4, 0
0022 dup

etc...
```

Local Table

```
[ 2] ?
[ 3] b
[ 4] a
```

Figure 4-17: The YARV instructions compiled from the beginning of the add_two method

What are these YARV instructions doing? The Ruby method `add_two` didn't contain any code similar to this! (All `add_two` does is add `a` and `b` together and return the sum.)

To find out, let's walk through Figure 4-17. On the left side, we see the Ruby `add_two` method, and on the right, the YARV instructions for `add_two`. On the far right, you see the local table for `add_two`. Notice that there are three values listed there: `[ 2] ?`, `[ 3] b`, and `[ 4] a`. It should be clear that `a` and `b` correspond to the two arguments to `add_two`, but what does `[ 2] ?` mean? This appears to be some sort of mystery value.

The mystery value is the hash we saw created in Figure 4-16! In order to implement keyword arguments, Ruby has created this third, hidden argument to `add_two`.

The YARV instructions in Figure 4-17 show that `getlocal 2, 0` followed by `dup` places this hash onto the stack as a receiver. Next, `putobject :a` puts the symbol `:a` onto the stack as a method parameter, and `opt_send_simple <callinfo!mid:key?` calls the `key?` method on the receiver, which is the hash.

These YARV instructions are equivalent to the following line of Ruby code. Ruby is querying the hidden hash object to see whether it contains the key `:a`.

```
hidden_hash.key?(:a)
```

Reading the rest of the YARV instructions from Figure 4-17, we see that if the hash contains the key, Ruby calls the `delete` method, which removes the key from the hash and returns the corresponding value. Next, `setlocal 4, 0` saves this value into the `a` argument. If the hash didn't contain the key `:a`, Ruby would call `putobject 2` and `setlocal 4, 0` to save the default value 2 into the argument.

To summarize, all of the YARV instructions shown in Figure 4-17 implement the snippet of Ruby code shown in Listing 4-11.

```
if hidden_hash.key?(:a)
  a = hidden_hash.delete(:a)
else
  a = 2
end
```

Listing 4-11: The YARV instructions shown in Figure 4-17 are equivalent to this Ruby code.

Now we can see that Ruby stores the keyword arguments and their values in the hidden hash argument. When the method starts, it first loads each argument's value from the hash or uses the default value if there is none. The behavior indicated by the Ruby code in Figure 4-14 explains the results we saw when running Listing 4-9. Remember that we changed the `Hash#key?` method to always return `false`. If `hidden_hash.key?` always returns `false`, Ruby will ignore the value of each argument and use the default value instead, even if a value was provided.

One last detail about keyword arguments: Whenever you call any method and use keyword arguments, YARV checks to see whether the keyword arguments you provide are expected by the target method. Ruby raises an exception if there is an unexpected argument, as shown in Listing 4-12.

```
def add_two(a: 2, b: 3)
  a+b
end

puts add_two(c: 9)
 => unknown keyword: c (ArgumentError)
```

Listing 4-12: Ruby throws an exception if you pass an unexpected keyword argument.

Because the argument list for `add_two` didn't include the letter `c`, Ruby throws an exception when we try to call the method with `c`. This special check happens during the method dispatch process.

Summary

This chapter began with a look at how YARV controls the execution flow of your Ruby program using a series of low-level control structures. By displaying the YARV instructions produced by Ruby's compiler, we saw some of YARV's control structures and learned how they work. In Experiment 4-1, we discovered that Ruby implements `for` loops internally using the `each` method with a block.

We also learned that internally Ruby categorizes methods into 11 types. We saw that Ruby creates a standard ISEQ method when you write a method using the `def` keyword and that Ruby labels its own built-in methods as CFUNC methods because they are implemented using C code. We learned about the ATTRSET and IVAR method types and saw how Ruby switches on the type of the target method during the method dispatch process.

Finally, in Experiment 4-2, we looked at how Ruby implements keyword arguments, and we discovered along the way that Ruby uses a hash to track the argument labels and default values.

In Chapter 5 we'll switch gears and explore objects and classes. We'll return to YARV internals again in Chapter 6 when we look at how the method lookup process works and discuss the concept of lexical scope.