Extracted from:

Text Processing with Ruby

Extract Value from the Data That Surrounds You

The Pragmatic Bookshelf

Dallas, Texas • Raleigh, North Carolina

The Pragmatic Programmers

Text Processing with Ruby

Extract Value from the Data That Surrounds You

Rob Miller

edited by Jacquelyn Carter

Text Processing with Ruby

Extract Value from the Data That Surrounds You

Rob Miller

The Pragmatic Bookshelf

Dallas, Texas • Raleigh, North Carolina

Pragmatic Bookshelf

The team that produced this book includes:

Jacquelyn Carter (editor)
Potomac Indexing, LLC (index)
Cathleen Small; Liz Welch (copyedit)
Dave Thomas (layout)
Janet Furlow (producer)
Ellie Callahan (support)

Printed in the United States of America.
ISBN-13: 978-1-68050-070-7
Encoded using the finest acid-free high-entropy binary digits.
Book version: P1.0—September 2015

CHAPTER 13

Writing to Other Processes and to Files

We've looked at writing to standard output, a useful technique that allows our programs' output to either be read by people or be redirected into another process as part of a pipeline chain. Sometimes, though, we don't want to leave the choice of redirecting output up to the user of our program; we want to write directly to a file or to another process ourselves.

This might be so that we can persist our output between invocations of our program, storing up more data over time, in which case it would be useful to be able to write data to a file. Or it might be to harness the power of other programs, so that we can avoid reimplementing logic ourselves. We could achieve this by constructing our own pipeline chains within our program, sending input in one and capturing the transformed output.

Let's take a look at this latter case first. It's a natural extension of writing to standard output but will give us great power and flexibility.

Writing to Other Processes

In Chapter 3, *Shell One-Liners*, on page ?, we looked at how powerful chaining processes together in pipelines could be. The chapter culminated in our mixing Ruby with other Unix commands to process text in an ad hoc fashion, with multiple processes linked together in a *pipeline chain*. Each took some text as input and then passed it along, in some transformed form, as output to the next process.

Here, for example, we use Ruby to extract only those rows of a file that we're interested in, and then use sort, uniq, and head to further process those lines:

```
$ cat error_log \
  | ruby -ne '$_ =~ /^\[.+\] \[error\] (.*)$/ and puts $1' \
  | sort | uniq -c | sort -rn | head -n 10
```

By using a pipeline chain like this, we speed up our development by having to solve only the novel parts of the problem. We're not forced to reimplement solutions to long-solved problems such as sorting, counting unique lines, and so on.

But in this case, the decision to use those further commands, to perform this extra logic, was taken by the user. Ruby had no knowledge of them. It just wrote its text to standard output, and that was that.

There's a certain usefulness and generality to that approach, but it doesn't always fit. Sometimes we know in advance that we want to harness the power of other processes and that we don't want to have to reinvent the wheel. In these cases, we'll need to be able to use these external utilities from within our scripts, not just on the command line.

Simple Writing

To write to another process, we need to do two things. First, we have to spawn the process that we want to communicate with. Second, we need to hook its standard input stream up to something in our script, which we can then write to, so that our data will be passed into the script in the same way as it would be on the command line.

Ruby offers us a short and simple way to do both of those things, and it's with our trusty friend `open`. (We've previously used `open` to read from files, in Chapter 1, *Reading from Files*, on page ?, and to fetch URLs, in Chapter 6, *Scraping HTML*, on page ?.)

As well as allowing us to read from files and URLs, `open` has a third function: it can read to and write from other processes. We achieve this by passing `open` a pipe symbol followed by a command name:

open-pipe.rb
```ruby
open("| sort | uniq -c | sort -rn | head -n 10", "w") do |sort|
  open("data/error_log") do |log|
    log.each_line do |line|
      if line =~ /^\[.+\] \[error\] (.*)$/
        sort.puts $1
      end
    end
  end
end
```

Here we open the same pipeline chain as we saw earlier on the command line. By passing `w` as the second argument to `open`, we tell Ruby that we want to write to the process. Then we open up the log file—the step that was performed by cat in our manual pipeline chain—and loop over the lines within it. If a line matches our conditions, we write the error message to the pipeline chain, causing it to be passed to the standard input stream of the first process.

When we execute this script, Ruby will dutifully start all of the processes we requested, hooking each one's standard output stream to the standard input stream of the next. It then creates a pipe, the output end of which is attached to the first process in the chain and the input end of which is passed to our block.

This works in exactly the same way as the command-line version did and produces the same output. Just like in the command-line version, we haven't had to reimplement any logic that isn't part of our core problem.

This technique is even more useful when used to perform tasks that would be impossible—or at least difficult—to perform in Ruby alone, but for which there exists ready-made command-line utilities. Let's imagine that we wanted to compress some text generated by our script and write that compressed data to a file.

There's a command called gzip that will do this for us. Like other filter commands, it takes standard input, compresses it, and outputs that compressed data to standard output. That means we can use it exactly like we did our previous pipeline, writing data to gzip's standard input stream and getting compressed data back into our program from gzip's standard output stream.

Unlike our first example, though, we don't want to have the output from the final process printed to our screen. We want to write it to a file. open has us covered here, too:

open-gzip.rb
```ruby
open("|gzip", "r+") do |gzip|
  gzip.puts "foo"
  gzip.close_write
  File.open("foo.gz", "w") do |file|
    file.write gzip.read
  end
end
```

In this case, we specify a mode of r+ to the pipe, to specify read and write mode. We then write to the pipe as we did in the first example, and then close it for writing. This sends the "End of File" notice to the gzip command, signaling that the input is complete. We then use read to read gzip's entire output, and we write that to a new file.

By doing this, we've been able to harness the power of an existing command-line tool, rather than needing to either implement gzip encoding ourselves (which would be infeasible) or use a Ruby library that does it for us (which might be impractical or undesirable).

A Practical Example: Paging Output

If you've ever used the version-control software Git, you might have noticed a particular behavior it has. If you use, for example, the `git log` command in a project with a lot of history, Git won't dump thousands of lines on your screen in one go, swamping you and making it difficult to see what was at the top of the output. Instead, it uses `less` to allow you to scroll through the output—a nice touch that makes using Git much easier.

If we're writing a script that outputs a large amount of text, we can achieve this same effect to make things easier for the people running it. By combining what we learned about redirecting standard output in the previous chapter with our newfound ability to write to other processes, we should be able to do just that.

Here's the code:

paging-output.rb

```
def page_output
  stdout = STDOUT.clone
  less = open("|less", "w")
  STDOUT.reopen(less)
  at_exit do
    STDOUT.reopen(stdout)
    less.close
  end
end
```

Just like in the previous chapter, we first clone `STDOUT` so that we can restore it later. Next we use `open` to spawn the `less` process for us. This, as we saw earlier in the chapter, will give us an `IO` object that will allow us to write to the `less` process's standard input stream. Then we use `reopen` to point the standard output stream of our own process to the standard input stream of `less`, ensuring that everything we write from this point on will be paged correctly—even output from subshells.

The final part of the method uses an exit handler to tell Ruby that once our script finishes, we should restore the original standard output stream and, critically, close the `IO` object that we have pointing to `less`. This allows `less` to function properly, since it allows it to know how many lines it's been given.

Everything output after we call the page_output method will be passed into less. Here's an example that checks for the presence of a terminal and, if it finds one, chooses to page the output. It then outputs a thousand lines of text:

paging-output.rb
```
page_output if STDOUT.tty?
500.times do |n|
  puts "#{n + 1}: Hello from Ruby"
  system "echo '#{n + 1}: Hello from a sub-shell'"
end
```

If we run this example, we should see the top of the output first—however many lines will fit in our terminal. We can then use the arrow keys to scroll up and down the text (and horizontally, if the lines were very long or if our terminal was very narrow). Compared to the experience of dumping a thousand lines of text into the terminal normally, this is a huge improvement.

There's an added bonus, though. Because we're checking for the presence of a terminal before redirecting our output to less, we've not broken the usual Unix model, so we can still use our script in a pipeline. For example, we could count the number of lines being output:

```
$ ruby paging-output.rb | wc -l
1000
```

This is the same behavior as Git, incidentally. It will page output when a terminal is present but do nothing when one isn't.

If we think about it, the implementation cost of all this is basically nothing. We don't have to reimplement paging ourselves; we just need to point a single stream to a different place. Our users also get the same behavior that they're familiar with from elsewhere. They don't get some poorly implemented subset of less; we give them the real deal. This is a great thing. Thanks to fundamental design decisions made decades ago, the hard work is done for us.