CHAPTER 5

Building Files the Smart Way

One of the most brilliant design choices found in UNIX-style operating systems is the notion that "everything is just a file." In other words, if your programming language can do very basic reading/writing of files and a little string handling, you can drive the configuration and environment of every bit of software you care about.

In this chapter, we will explore the most common file-building methodologies and find out how to apply them the Ruby way. In particular, before getting down to writing a single byte, we'll discuss and create some safe file operations. Such operations are imperative in system maintenance and administration scripts. Thus we will deal with the safety issues before proceeding to discuss both program- and template-driven file creation.

Safety First

I remember one morning my coworkers and I were sitting around in the systems support room chatting about something (snack foods as the key to happiness if memory serves) when someone uttered those portentous words: "That's odd." Now, an experienced administrator will tell you that these two little gems, when spoken together, are the verbal equivalent of a small highland terrier yapping inexplicably 30 minutes before your city is leveled by an earthquake.

In this case, the doomsayer in question had noticed that the performance of his web browser had become a little erratic, seeming to operate in fits and starts. No sooner had he explained this than the guy behind me complained that his mail client had started timing out. It's at this point that we received our first phone call requesting support.

Two or three hours later, when the apocalypse had spent itself, we were finally in a position to understand what had gone wrong. You see, our DNS server was driven dynamically out of a database. Every few minutes, a little script would run to read the configuration and the zone files, compare them with what it thought they should say according to the contents of the database, and update them as necessary (refer to Figure 5-1). If an update did take place, the script would nudge the name-server daemon to get it to notice the new configuration.

In theory, a single run of the script was supposed to take far less time than the interval between such runs. In theory, it was unthinkable that more than one instance of the script would be running at once. In theory, tomatoes are fruit. So much for theory.

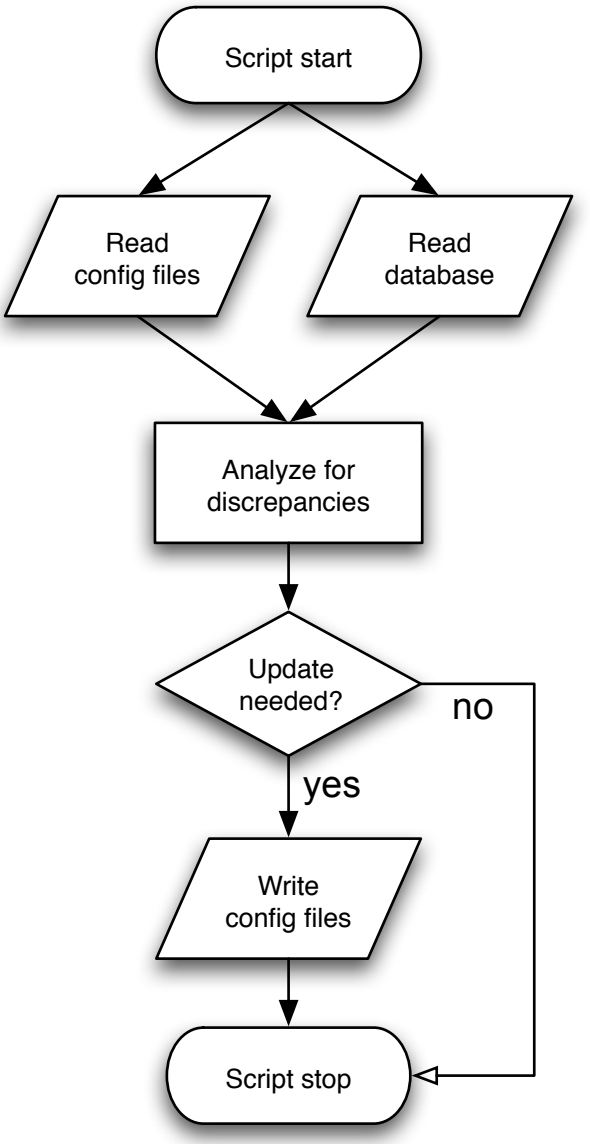

Figure 5-1. *The deceptively simple flow of the update script*

The first instance of the script (we'll call it A) started running and, probably due to abnormal load on the server, got about halfway through writing an updated configuration file when a second instance (B) was invited to the party. It read the half-written file and naturally concluded that there was a vast swath of zones missing from the server. Hence, B leapt into action and

started writing a new configuration file on top of the now mostly complete one A had been committing to disk. As A finished, it nudged the name-server daemon to let it know it should reread its configuration files, which at this point (of course) were only halfway through being written by B. As if by magic, a good chunk of zones suddenly went AWOL.

Now people were having problems with web access, e-mail, and even network volumes, which all relied heavily on DNS resolution. This meant more hits of Refresh buttons and sides of computers. The abnormally large number of demands for name resolution in turn put increased load on the DNS server, which meant that within about 20 minutes there were four instances of this script competing with each other, completely obliterating the zone information on the server and pushing it to sustain maximum load. By the time one of our guys realized it was a DNS issue and could get a remote session on the machine, an entire Whitehall farce was going on in memory, all because of one little false assumption made by the scriptwriter.

The moral I took from this episode was an important one. When writing scripts that mess about with files on disk, concurrency issues must be considered from the very beginning. In other words, it is important to take explicit steps to enforce the kinds of mutually exclusive behavior required by the script in this story. Fortunately, Ruby makes it easy to do the right thing in this respect.

File Locking

All the mainstream operating systems implement some form of file locking and vend a pretty simple API for using it. In the context of a file, a *lock* is a flag that can be requested by a process that, in general terms, means "This lavatory is currently occupied—wait your turn." More specifically, there are two styles of locks supported by the UNIX locking mechanism.

A *shared* lock can be held by many processes at the same time. An *exclusive* lock can be held by only one process at a time. You are most likely to use exclusive locking day to day, as you generally want to just stake a claim to a file for the whole duration of the read/write portion of your script. This is the so-called coarse locking approach and, while lacking elegance, it is the most difficult to get wrong.

You might be wondering under what circumstances you'd use shared locking. The classic scenario employing this approach involves a file that lots of processes want to read from and occasionally write to. Each process acquires a shared lock while reading and then continues on its merry way until it has finished reading, releasing the lock when done. Of course, if the file isn't changing, it doesn't matter how many processes read from it simultaneously—it will always say the same thing. However, some process now decides to modify the file. This writing process should ask for an exclusive lock on the file and, since only one process can hold an exclusive lock on a particular file at any one time, it will have to wait until all of the shared locks have gone away. Thus shared locks are overwhelmingly a mechanism for ensuring read integrity, whereas exclusive locks ensure both read *and* write integrity.

Blocking Locks

The simplest kind of locking is that which blocks your program until the lock may be granted. Have a look at the code in Listing 5-1, which demonstrates a simple lock-write-unlock procedure.

Listing 5-1. *Safe File Writing Using a Lock*

```
File.open("/tmp/foo", "w") do |f|
  f.flock(File::LOCK_EX)
  f.puts "Locking is the key to ... pun interrupted"
  f.flock(File::LOCK_UN)
end
```

This code starts by attempting to open a file for writing and names the object representing access to that file `f` (a file handle). Personally, I always use this block-style invocation of `File.open`, as the block doesn't execute unless the file was successfully opened with the required access rights. Also, it makes something we're about to do much easier.

Once we have our file handle, we call the `flock` method with the argument `File::LOCK_EX`, requesting an exclusive lock on it (`File::LOCK_SH` would request a shared lock). At this point, if any other process has any kind of lock on the file, our script will block and wait for the other locks to be released. Once the file is clear of locks, our script can then write its little message into the file. As a final act of tidiness, we explicitly unlock the file again, indicating that we've finished with it.

Nonblocking Locks

Now what if we merely want to check whether there's a lock and do something different if we can't get immediate access to the file? The `flock` call allows us to request this behavior with an extra flag (`File::LOCK_NB`), as shown in Listing 5-2.

Listing 5-2. *Locking but Not Blocking*

```
File.open("/tmp/foo", "w") do |f|
  if f.flock(File::LOCK_EX | File::LOCK_NB)
    f.puts "I want to lock it all up in my pocket"
    f.flock(File::LOCK_UN)
  else warn "Couldn't get a lock - better luck next time"
  end
end
```

In this listing, we once again use the block form of `File.open`, but instead of the lock-write-unlock approach we used last time we do the following:

1. Try to acquire a lock without blocking
2. If successful, write and then unlock
3. If unsuccessful, throw up a warning

Note, incidentally, the use of the bitwise OR operator (`|`) when selecting the behavior of `flock`. This should be familiar to anyone who's done this sort of thing in C.

Abstracted Locking

The fact that, in general terms, we always seem to end up with a lock call at the beginning of the block and an unlock call at the end points to a possibility for abstraction. There are many ways to do what I'm about to (see the sidebar "Open Abuse" for an alternative approach). The following is probably the easiest to follow and provides a nice opportunity to show off the ability to add methods to built-in classes in Ruby.

What we want to do is define a method that acts as much like `open` as possible, but will implicitly give us an exclusively locked file to play with and automatically unlock it for us. Take a look at Listing 5-3.

Listing 5-3. *Extending the File Class to Include a Convenient Locked Writing Method*

```
class File
  def File.open_locked(*args)
    File.open(args) do |f|
      begin
        f.flock(File::LOCK_EX)
        result = yield f
      ensure
        f.flock(File::LOCK_UN)
        return result
      end
    end
  end
end
```

As previously explored, we can extend Ruby's built-in classes like any other class with a simple `class` declaration, inside which all definitions are in the context of that class. The method signature for `open_locked` uses the * construction. This simply means "gather up any arguments passed to this method and dump them in an array." In this case, the array is called `args`, but there is nothing special about that name.

The usefulness of the `args` variable we define becomes obvious in the next line, where we invoke `File.open` in its by now familiar block form. Essentially, we are ensuring that our function has an identical signature to the function we're mimicking by not giving the slightest hoot what that signature is. It also means that any future changes to the argument structure of `File.open` won't break this code.

The actual meat of the code contains the requisite lock-operate-unlock logic. Just as with `File.open`, any code using our method should present `open_locked` with a block of code that accepts a file handle. The difference, of course, is that this handle will be exclusively locked when passed by our method.

We expressly want any errors that would normally raise an exception to still do so unhindered, in keeping with our mimicry of `File.open`. The only minor difference is that, whatever may go wrong, if we've locked the file we must unlock it again. Hence the use of a `begin...ensure...end` block. As you might surmise, everything within the `ensure` part is run irrespective of whether an exception is in the process of being raised.

Our striving to impersonate `File.open` as much as possible is why we're also careful to preserve the `result` of the passed code block. Once again, this is how `File.open` behaves and it allows us to do this kind of thing:

```
raw_data = File.open_locked("/tmp/foo") { |f| f.read }
```

OPEN ABUSE

Instead of creating an `open`-like method we could specifically override `open`. This would mean that `File.open` itself would automatically lock and unlock files for us (just like our `open_locked` method). Playing with such an important method is probably something we want to do in a fairly careful manner, as it will affect a lot of subsequent code. On this basis, I want to emphasize that this example should be taken more as an academic "what if" than something you would choose to do in production code. `File.open` is just too fundamental to mess with in large projects without inviting trouble.

Grabbing an exclusive lock regardless of whether a read-only or read-write file handle has been requested would needlessly degrade performance. Thus we need to achieve two things that `open_locked` didn't. The first is to override `File.open` correctly so that we don't start causing warnings to appear and so that it is still available to us internally. After all, neither you nor I have the desire to completely reimplement file operations from scratch in any language if we don't have to. The second goal is to be a little smarter with the type of lock we ask for than our first attempt in Listing 5-3 (which you should read before this sidebar). Here's a solution:

```
class File
  class << self
    alias open_old open
  end

  def File.open(path, mode = "r", perm = 0644)
    File.open_old(path, mode, perm) do |f|
      begin
        f.flock(mode == "r" ? LOCK_SH : LOCK_EX)
        result = yield f
      ensure
        f.flock(File::LOCK_UN)
        return result
      end
    end
  end
end
```

Examining the features that make this different from the `open_locked` solution, we first have to discuss the aliasing process. Within the context of a class definition, a new alias to an existing instance method can be created with the `alias` keyword. The problem is that the method we are trying to overwrite is a class method, and so we have to use a somewhat advanced bit of syntax. The double chevron (`<<`) construction means that we wish to extend the class in the context of its class-level namespace (the double chevron actually means "extend the current implicit object"). Thus the aliasing is done at this level.

The rest of the code should be much easier to follow. This time we have to care about the parameters to `open`, as the type of lock will depend on the `mode` variable. Also, for quickness, we take certain liberties with the permissions setting just to give us something to pass to the original method without having to worry too much.

Process Locking

One cute trick I've seen employed quite a lot ensures that there is never more than one instance of a script running. First, pick a location on the disk that is appropriate for the storing of process running data. If your script is designed to run as a UNIX daemon, it will usually have a PID file associated with it that you create in `/var/run/myscript.pid`, so that is an obvious choice for this next step. Otherwise something in `/tmp` will do.

As early as possible in your script, get a handle to your chosen file and attempt to gain an exclusive lock on it in the manner discussed throughout this section. If you can't get the lock, you know another instance of your script is already lurking somewhere in memory and the new instance can bow out gracefully.

What if you forget to release the lock before your script exits, or the script crashes before it has an opportunity to unlock the file? Won't that leave you unable to run the script ever again? The beauty of this approach is that file locks are a property of the process requesting them (at least for our purposes). If the process goes away for whatever reason, the OS guarantees that the lock will be released.

Locking Considerations

Whether protecting files or processes, there are a couple of points to be aware of when employing file locks:

- UNIX-style systems treat file locking as advisory by default. The fact that you've been good and wrapped your file system calls in `flock`s in no way implies that another application will be blocked if it attempts to do an ordinary write to the file of interest without any `flock`ing.
- Some file systems have inconsistent/nonexistent support for the `flock` call (NFS prior to Linux 2.6.12 in particular springs to mind), so you need to be conscious of where your code will be deployed.

Safe File Operations

This concern over regimented access to files is all very well, but it doesn't protect you from one other important class of file-integrity issue: *partial writes*. If your script crashes halfway through its operation, the OS will tidy up the locks and another process will be free to access the file you were writing. Unfortunately, if you didn't finish writing data to disk, you now have a serious problem.

You simply cannot guarantee that your script won't crash or run out of disk space or some such thing. We have to find a way to make sure that a configuration file, for example, only gets overwritten with a complete replacement or not at all. Also, it would be nice to achieve the kind of file-locking safety we had in the previous subsection so that when we write files we can guarantee that no other process is reading from them.

Abstracting Safe File Operations

As before, our approach will be to add a method to the built-in `File` class that mimics the behavior of `File.open` but that copes with our file-safety concerns transparently. The strategy for performing a safe write will be to write all data to a temporary file and then move it on top of the original only upon its closure.

This approach prevents partial writes, as a given file updated this way can be overwritten only when there is a complete replacement ready. In addition, the only file system operation that is as close to being atomic (indivisible—it either completes or it doesn't) as we can hope for is a move. Moving a file on disk actually does nothing of the sort; it simply changes the path associated with the file. This means that the code shown in Listing 5-4 will provide the safe writes we're looking for.

Listing 5-4. *Extending the File Class to Include an Anti–Partial Writes Method*

```
require "fileutils"
require "tempfile"

class File
  def File.open_safely(path)
    result, temp_path = nil, nil
    Tempfile.open("#{$0}-#{path.hash}") do |f|
      result = yield f
      temp_path = f.path
    end
    FileUtils.move(temp_path, path)
    result
  end
end
```

Let's unpick this carefully. The first thing to note is that we've included the functionality provided by the `FileUtils` module—this is necessary for the file move we'll be doing near the end. Additionally, we'll call on the services of the `Tempfile` class for creating the temporary file.

You should immediately notice a similarity of construction when comparing Listing 5-4 to Listings 5-2 and 5-3. We are essentially wrapping a block operation (albeit with a single postoperation step) in another block-accepting method (ours).

The definition of `open_safely` is simpler than that of `File.open` because it is a specifically write-only method. As such, it takes only one argument, namely the `path` of the file to be written. We use a numeric `hash` of this `path` together with the name of the invoking application (`$0`) to construct a unique path for the temporary file. `Tempfile.open` takes this base name and adds the application's PID and a unique index to it, and this forms the complete temporary filename. An example of a path produced via this method would be `/tmp/filesift-736474449.5189.1`.

`Tempfile.open` creates this file read-writeable by the user running the script only (mode `600`). Thus we `yield` the handle to this file to the block our method was passed. The calling script can write data to its heart's content and then, once it returns, we note the `result` (for the same reasons discussed previously in the context of Listing 5-3).

We also make a note of the path allocated by `Tempfile`. We have to get this from within the temporary file block—we don't necessarily know this in advance, as the index on the end of the filename can be anything from `0` up to `9`.

With the file written and the result secure, all that remains is to move the temporary file over the top of the original. This is accomplished through one of the nice convenience methods of the `FileUtils` module—in this case, `FileUtils.move`. You should take a look at `ri FileUtils` to see a complete list of the rather handy set of methods provided by this module.

The more astute among you may be wondering what happened to the locking code. Didn't we still want that kind of protection? Remember that we are creating a brand-new file—nothing else will be reading from or writing to it because no other process should have the vaguest clue that it exists. Once the move has happened, the original file will actually hang around if any other processes have file handles to it—implying read safety on an albeit out-of-date copy of the file. In other words, since the moving of the file is atomic and it doesn't destroy any currently occurring access to the old file, we get concurrency protections automatically.

Using the Abstraction

Given the functionality provided by the code in Listing 5-4, it would be useful to consider an example of its use. In particular, because this file-safety approach compensates for the broadest spectrum of potential mishaps, it is the one I use almost all the time.

A recent example of this was the building of `/etc/passwd` on a Linux box. As you will doubtless be aware, this file contains the definition of the set of users on such a machine together with pertinent information such as their UID, home directory, and default shell.

In this case, I was presented with a challenge to build this file based on the contents of a CSV export from a central management system. I will not show the steps taken to parse to CSV file, as this subject is covered extensively in Chapter 7. Instead, we enter the fray in Listing 5-5 with an array called `users` that is populated with one hash per user. In turn, each hash has all the data needed to construct dynamic bits of the appropriate line in `/etc/passwd`.

Listing 5-5. *Safely Writing /etc/passwd*

```
require "safe_file"  # this will be the code from Listing 5-4

FIELDS = [:login, :password, :uid, :gid, :name, :homedir, :shell]

File.open_safely("/etc/passwd") do |passwd|
  users.each do |user|
    passwd.puts user.values_at(*FIELDS).join(":")
  end
end
```

As you can see in this listing, there is not a lot of work involved in actually creating the file. For each user, the defined set of field names (`FIELDS`) is used to pick out the relevant fields from the hash containing the user's information. Note the use of * to unpack the array object into a list of arguments to be passed to `Hash.values_at`. The array thus generated is joined together with semicolons in accordance with the format used by `/etc/passwd`.

As we would demand from any decent abstraction, all the hard work of ensuring the file is written safely and completely (or not at all) is handled by the `open_safely` method we wrote in Listing 5-4. This leaves us to concentrate on getting the format of the file right.

The Pen Is Mightier Than the Words

Now that a snug feeling of safety permeates the way we interact with files on disk, it's time to create some. Conceivably, there are thousands of different formats of file you might wish to create in myriad bizarre ways. You'll understand then that it is a continuing surprise to me that there is basically only one big choice when setting out to design your file-writing scripts. You're either a builder or a templater (which is almost certainly not a real word).

This section is intended to provide a flavor of the sorts of file creation approaches available in Ruby. It does not go into detail about any particular technology; rather, it presents a conceptual overview of the kinds of subtasks system administrators often find themselves doing when generating files.

Mob the Builder: Program-Driven File Creation

The first approach is usually typified by writing a lot more code than data. Every time you write a script that does something like `puts "fish-sticks: 8"`, you are engaging in an excruciatingly simple form of the builder pattern. You are telling your script not only what to write, but also exactly when to write it.

When Ruby-ists talk of builders, however, they rarely include such trivialities. The accolade of "proper builder library" tends to be reserved for honking great kitchen-sinks of modules bristling with classes that allow you to use tiny drops of code to create vast oceans of data.

Builders tend to exist for document formats designed to hold highly structured data. The API usually gives you a means to construct a version of the document in memory that uses formal object relationships and other niceties of the language to track the structure. Once the data has been built up in this manner, there is often one big, thrilling write command that will give you a text/binary dump of the file in the appropriate format to reproduce the given data structure.

Building XML

Probably the best-known builder tools are the various XML building libraries. The need to write XML is becoming more and more essential all the time, as everyone and his monkey decide that it is the one true way to exchange data (more on this in Chapter 7). Hence, making it easy for poor system administrators is essential. Listing 5-6 is a quick example using the well-known Builder library (`http://builder.rubyforge.org`) to produce a tiny, headless XHTML document.

Listing 5-6. *Building an XHTML Document*

```
require "builder"

builder = Builder::XmlMarkup.new
page = builder.html do |html|
```

```
  html.head { |head| head.title("Users") }
  html.body { |body| body.a("bob", "href" => "b1") }
end
```

This listing demonstrates the use of Ruby structural formalisms (mostly blocks) to specify the construction of an XML document. We start by initializing a builder object that is responsible for doing the actual building. Note that the initialization step can be used to do things like specify an output file handle and default indentation policy. See the documentation for more detail.

Thus initialized, the `page` is constructed as the output of a set of nestled Builder commands. Note the ability to refer to the names of elements (`head`, `body`) without special quoting. Builder is using the `method_missing` approach discussed in Chapter 4 to provide an on-demand domain-specific language whereby any unknown method name is treated as an instruction to open a new child XML element with that name.

Simple string content is added by providing a single argument to one of these method calls, as seen in `head.title("Users")`. Any properties to go inside the tag are specified using a hash after this initial string, as demonstrated in the next line where an `href` is specified for the link. This is akin to the flexible method signature approach also discussed in Chapter 4.

Given the code as written in Listing 5-6, the output will look like this:

```
<html><head><title>Users</title></head><body><a href="b1">bob</a></body></html>
```

Building Images

Perhaps a better example of the way in which a builder can be helpful in abstracting away the complexity of a genuinely inscrutable file format is seen with an early Ruby interface to the popular GD (`www.boutell.com/gd/`) library. This library provides a mechanism for translating individual drawing commands into a rasterized image, and an example of just this is shown in Listing 5-7.

Listing 5-7. *Using GD to Build a GIF*

```
require "GD"

image = GD::Image.new(100, 100)      # create an empty canvas, 100 pixels square
red = image.colorAllocate(255, 0, 0) # define the color red as RGB(255, 0, 0)
image.rectangle(25, 25, 75, 75, red) # draw a red square in a particular place
image.gif STDOUT                     # dump the GIF of this drawing to standard out
```

What I love about this approach to drawing is that I don't have to give a flying walnut about how the GIF format works in order to produce one. It's like playing with Logo (`http://el.media.mit.edu/Logo-foundation/`) all over again. This kind of abstraction is often one of the biggest advantages of Builder libraries. Much more detail on this subject can be found in Chapter 9.

Building Repetitive Text Files

The other place you find a little code to generate a lot of data is when generating files with a lot of repetition. For example, I recently had to build a file that mapped every IP address in a certain subnet to a particular time of day. I leave it as an exercise for the reader's paranoia to decide why one might require the script shown in Listing 5-8.

Listing 5-8. *Building a File with Repetitive, Programmatically Defined Contents*

```
seconds_per_ip = 60 * 60 * 24 / 254.0

File.open("config.txt", "w") do |f|
  1.upto(254) do |d|
    seconds = (d - 1) * seconds_per_ip
    hrs = seconds / (60 * 60)
    mins = (seconds / 60) % 60
    secs = seconds % 60
    f.puts "1.2.3.#{d} " + sprintf("%.2i:%.2i:%.2i", hrs, mins, secs)
  end
end
```

This listing shouldn't be too difficult to follow. It opens a file, and then iterates through the set of numbers from 1 to 254, using those numbers to determine both the IP address and an appropriate time of day. Note again that, because of the block structure of the `File.open` invocation, the script will do no work if the file fails to open for writing—exactly as we'd want.

Building/Changing Text Files Using a Policy

The final pertinent example is where a script essentially acts as a conversion tool, taking a set of structured data and wrangling it into a different format or updating some of the values as they go past. This sort of script usually does a few things a lot of times, implying repetition of operations rather than data. Take as an example the code in Listing 5-9 that I use to scan all of my C source files in a given directory and check that the copyright notices are up to date.

Listing 5-9. *Using Ruby As a Line Editor to Update Copyright Notices*

```
Dir["*.[c|h]"].each do |path|
    lines = IO.readlines(path)

    line_number, first_year = nil, nil
    lines.each_with_index do |line, i|
        next unless line =~ /Copyright (\d+)/
        line_number, first_year = i, $1.to_i
        break
    end

    this_year = Time.now.year
    expected_notice = if first_year and first_year < this_year
        "// Copyright #{first_year}-#{this_year} Andre Ben Hamou"
```

```
    else "// Copyright #{this_year} Andre Ben Hamou"
    end

    if line_number
        next if lines[line_number].chomp == expected_notice
        lines[line_number] = expected_notice
    else lines.unshift(expected_notice)
    end

    puts "Updating #{path.inspect}"
    File.open(path, "w") { |f| f.puts lines }
end
```

This is not the most elegant of scripts, but it took about two minutes to write and has proven invaluable ever since. Because it demonstrates a number of useful Ruby-isms, I'm going to take a few moments to explain it, beginning with the very useful `Dir` class.

Anyone familiar with the command line will know about *globbing* patterns. These are the standard means for selecting files on the basis of patterns within file paths. So the invocation on the first line retrieves every path matching `anything.c` *or* `anything.h` as an array we can iterate through.

■ **Tip** Going deeper into the `Dir` invocation at the top of Listing 5-9, you might be tempted to think that the use of square brackets in this manner is hard-coded into the language somehow. In fact, this operator and many others that have to be special in other languages can be overridden in Ruby. Simply use a method definition of the form `def [](key)` or `def []=(key, value)`.

For each of the paths generated, we read every line of the corresponding file into an array. For such trivial access to files, particularly where you will be fiddling with the contents on a line-by-line basis, you'll find yourself enjoying the convenience of `IO.readlines` quite a bit.

The next task of this script is to scan through the source file looking for the copyright notice. Note the use of `each_with_index`, which is available to any class that includes the behavior from the rather useful `Enumerable` module. We use this method because we're going to need to remember what line the copyright notice was on for later.

Within the line-iterating block, we use the facility of being able to state conditionals after the block of code they refer to (go back and look if you don't believe me). This convention makes that line very easy to read—an inherent good. It also means less nesting of code, as we only do the rest of the block if the line matches the regular expression.

If a match is found, we assume that it will be the only one, note its position and first stated year, and then break out of our iteration. After all, there is no more scanning to be done. Why waste time doing so?

A rather useful property of the Ruby way of doing things is that blocks, procedures, and other logical groupings of code almost always return the value of the last expression evaluated within them. In the previous section, the methods had an explicit `return` at the end of them to

make it obvious to the beginner what was being returned. Had I removed this keyword and simply left the variable in question by itself, the code would have been semantically identical.

It turns out that we can treat an `if-elsif-else-end` block like this as well. So rather than waste code by having to explicitly assign the correct string to `expected_notice` in both branches of the conditional, we can think about it more abstractly. Specifically, we can assign the value of the last evaluated expression in the conditional to the variable. Pretty neat.

Following on from this, the script now tests to see whether any copyright notice was found in the source file. If so, and it was correct, we jump to the next file and start again. Note that the `next` in this case refers to the most local iteration we are doing, which is the walk through the path names. If the line needs updating, we overwrite it in memory. If no copyright notice was found, we shove the correct notice at the top of the set of lines.

If we're still executing the block at this point, then we've done something that requires updating the file on disk. We do so and dump a quick note to that effect for the operator. Mission accomplished.

Some of you may be familiar with source control systems that use placeholders in the comments so that scripts can insert copyright and other information. In other words, the source file is basically still a source file, but it contains little pockets of metainformation designed to be substituted for actual information. Such schemes are the very essence of the second file-creation concept: *templates*.

ThundERbolts and Lightning: Template-Driven File Creation

Already in this book, we have gained much utility from the Ruby feature for inserting dynamic values into strings. This pattern of having a moderate amount of static data with a few dynamic bits is most conducive to the template approach. Think about this code snippet:

```
nouns = ["mind", "body", "soul", "independence", "shock", "fishcakes"]
puts "Randomness is a state of #{nouns[rand(nouns.size)]}."
```

Here we define a list of (in)appropriate nouns for our sentence and then, just before printing it out, we insert a random noun as the last word. For simple substitutions, this is a great way to build what is after all a dynamic string (since at least one character is built programmatically). This is the template pattern in action.

Web developers who have been held hostage by PHP will recognize one of its few genuine strengths is that it allows you to do something like this:

```
<p>Today we celebrate the first, glorious anniversary of <? amusing_event ?>.</p>
```

The ability to in-line small chunks of code into other data is what made PHP so popular for rapid web development (and rightly so). It should come as no surprise that very early on the Ruby community decided it needed an equivalent system. Thus *Embedded Ruby* (ERb for short) was born.

Anyone who has constructed a site in Rails will be eminently familiar with the use of ERb. Ruby code gets wrapped in either a `<% code_here %>`-style block if you need to perform some logical operation that has no direct output for the page or a `<%= code_here %>` to specifically insert the string value of the last evaluation in the code block. As an example, I might want the title of my web page to include the time it was rendered. The following snippet achieves this:

```
<head>
  <title>15 Ways to Slice a Mango - <%= Time.now %></title>
</head>
```

It occurred to me almost as soon as I saw this style of file creation that it could be enormously useful for configuration files. Take my message-of-the-day file sitting at `/etc/motd`. I would quite like this file to be updated periodically with some dynamic information, but I have an overall layout that I like and don't want to have to re-create in code. So I have an ERb version that looks like this:

```
*** Welcome to <%= `/bin/hostname -s` %> ***

There are <%= `/usr/bin/who`.split("\n").size %> active sessions as of➥
  <%= Time.now %>.

Remember - nothing unreal exists so if you see Santa in here, give him a wave.
```

ERb can be invoked within a script as part of a more complex procedure (try unpicking the Rails view system if you have a spare week), but in this case we just want to parse the entire file and do the ERb substitution. If your system has a build of Ruby, it will almost certainly have the `erb` script installed. Mine does and so, every quarter of an hour, it runs `erb /etc/motd.erb > /etc/motd`.

You might well imagine a huge configuration file like the one for the SSH daemon whereby we want to set up defaults for the vast majority of settings but need a couple to be dynamically generated. Or how about belly-dancer files, which have static tops and tails but dynamic middles? They always bring a smile to my face.

Note The main reason I've ended up disliking PHP has very little to do with fundamental problems in the language and far more to do with emotional baggage. It's the prospect of having to debug yet another 650KB monolithic dino dung-ball of PHP that has been driving most of a web site for a few years that brings me out in a cold sweat. The reason such carbuncles exist is that PHP put an emphasis on in-lining, and then lots of happy-go-lucky web designers got so used to thinking in these terms they didn't stop to consider the point at which the use of templates by themselves becomes hopelessly unwieldy. Let their mistake be our education.

When Flat Files Fall Flat

In this chapter, we took a detailed look at how to deal with files safely, based on the argument that proficiency in handling files defensively is essential due to the pervasive nature of file manipulation within a system administrator's daily work. Such file operation safety has been achieved both via canonical file locking and the use of temporary files to prevent partial writes. We also saw how to add methods providing such functionality to the core Ruby `File` class as part of a convenient abstraction.

We engaged in a high-level discussion of the most common approaches to creating files via building and templating. The Builder library example hopefully provided a sense of how complex XML might be built with simple Ruby. In addition, we saw that templating can make

for an extremely easy-to-maintain mechanism when dealing with documents whose contents have extensive static sections.

The remarkably powerful convention of file-based configuration allows you to place all kinds of weird and wonderful abstractions on top of the basic business of configuring your applications and daemons. However, there rapidly comes a point where data needs to exist independently of a single computer. We have concentrated on writing files, but where is the data coming from?

More and more, new versions of software seem to be moving away from basic text files to database-held configuration. Conceivably, this could be a problem as no interface can be as commonly accessible as a simple file. However, as long as the interfaces are still well understood and supported, this design choice can offer some real advantages in terms of speed and functionality. The next chapter will shine a spotlight on such storage choices as we move from flat files to a cloud of data.