Table of Contents

Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

19. Automating Tasks with Rake

Even when your software is written, tested, and packaged, you're still not done. You've got to start working on the next version, and the next... Every release you do, in some cases every change you make to your code, will send you running through a maze of repetitive tasks that have nothing to do with programming.

Fortunately, there's a way to automate these tasks, and the best part is that you can do it by writing more Ruby code. The answer is Rake.

Rake is a build language, Ruby's answer to Unix `make` and Java's Ant. It lets you define *tasks*: named code bocks that carry out specific actions, like building a gem or running a set of unit tests. Invoke Rake, and your predefined tasks will happily do the work you once did: compiling C extensions, splicing files together, running unit tests, or packaging a new release of your software. If you can define it, Rake can run it.

Rake is available as the `rake` gem; if you've installed Rails, you already have it. Unlike most gems, it doesn't just install libraries: it installs a command-line program called `rake`, which contains the logic for actually performing Rake tasks. For ease of use, you may need to add to your PATH environment variable the directory containing the `rake` script: something like `/usr/lib/ruby/gems/1.8/gems/rake-0.6.2/bin/`. That way you can just run `rake` from the command line.

A Rakefile is just a Ruby source file that has access to some special methods: `task`, `file`, `directory`, and a few others. Calling one of these methods defines a task, which can be run by the command-line `rake` program, or called as a dependency by other tasks.

The most commonly used method is the generic one: `task`. This method takes the name of the task to define, and a code block that implements the task. Here's a simple Rakefile that defines two tasks, `cross_bridge` and `build_bridge`, one of which depends on the other. It designates `cross_bridge` as the default task by defining a third task called `default` which does nothing except depend on `cross_bridge`.

```
# Rakefile
desc "Cross the bridge."
task :cross_bridge => [:build_bridge] do
  puts "I'm crossing the bridge."
end

desc "Build the bridge"
task :build_bridge do
```


Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

```
  puts 'Bridge construction is complete.'
end

task :default => [:cross_bridge]
```

Call this file `Rakefile`, and it'll be automatically picked up by the `rake` command when you run the command in its directory. Here are some sample runs:

```
$ rake
Bridge construction is complete.
I'm crossing the bridge.

$ rake build_bridge
Bridge construction is complete.
```

Note all the stuff I didn't have to do. I didn't have to write code to process command-line options and run the appropriate tasks: the `rake` command does that. The `rake` command also takes care of loading the Rake libraries, so I didn't have to recite `require` statements at the beginning of my Rakefile. I certainly didn't have to learn a whole new programming language or a new file format: just one new Ruby method and its arguments.

Adapt the recipes in this chapter to your project's Rakefile, and a lot of the auxilliary work that surrounds a software project will simply disappear. You won't have to remember to run unit tests or generate documentation after every change, because it will happen as a side effect of things you do anyway. If your unit tests fail, so will your attempt to release your project, and you won't be embarrassed by bugs.

Whenever you ask yourself: "What was the command to ...?", just invoke rake with the `-T` option. It will print a list of available tasks and a description of each:

```
$ rake -T
(in /home/leonardr/my_project/)
rake build_bridge  # Build the bridge.
rake cross_bridge  # Cross the bridge.
```

Nothing says you can only use Rake in Ruby projects. Most Rake tasks simply run external programs and move disk files around: the same things tasks do in other build languages. You can use Rake as a replacement for `make`, build static web sites with it, or automate any other repetitive action made up of smaller, interlocking actions.

Here are some more resources for automating tasks with Ruby:

- The site http://docs.rubyrake.org/ provides a tutorial, a user guide, and examples for Rake.
- The generated RDoc for Rake has a good overview of the special methods available to Rakefiles (http://rake.rubyforge.org/files/doc/rakefile_rdoc.html)

Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

Recipe 19.1. Automatically Running Unit Tests

Credit: Pat Eyler

Problem

You want to make it easy to run your project's unit test suite. You also want the tests to run automatically before you do a new release of your project.

Solution

Require the `rake/testtask` library and create a new `Rake::TestTask`. Save the following code in a file called `Rakefile` in the project's top-level directory (or add it to your existing Rakefile).

```
require 'rake/testtask'

Rake::TestTask.new('test') do |t|
  t.pattern = 'test/**/tc_*.rb'
  t.warning = true
end
```

This Rakefile makes two assumptions:

1. The `Test::Unit` test cases live in files under the `test` directory (and its subdirectories). The names of these files start with `tc_` and end in `.rb`.
2. The Ruby libraries to be tested live under the `lib` directory. Rake automatically appends this directoy to Ruby's load path, the list of directories that Ruby searches when you try to `require` a library.

To execute your test cases, run the command `rake test` in the project's top-level directory. The tests are loaded by a new Ruby interpreter with warnings enabled. The output is the same as you'd see from `Test::Unit`'s console runner.

Discussion

If it's easy to trigger the test process, you'll run your tests more often, and you'll detect problems sooner. Rake makes it really convenient to run your tests.

We can make the test command even shorter by defining a *default* task. Just add the following line to the Rakefile. The position within the file doesn't matter, but to keep things clear, you should put it before other task definitions:

```
task "default" => ["test"]
```


Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

Now, whenever we run `rake` without an argument, it will invoke the `test` task. If your Rakefile already has a default task, you should be able to just add the `test` task to its list of prerequisites. Similarly, if you have a task that packages a new release of your software (like the one defined in Recipe 19.4), you can make the `test` task a prerequisite. If your tests fail, your package won't be built and you won't release a buggy piece of software.

The `Rake::TestTask` has a special attribute, `libs`; the entries in this array are added to Ruby's load path. As mentioned above, the default value is ["`lib`"], making it possible for your tests to require files in your project's `lib/` subdirectory. Sometimes this default is not enough. Your Ruby code might not be in the `lib/` subdirectory. Or worse, your test code might change the current working directory. Since `lib/` is a relative path, the default value of `libs` would start out as a valid source for library files, and then stop being valid when the test code changed the working directory.

We can solve this problem by specifying the absolute path to the project's lib directory in the `Rakefile`. Using an absolute path is generally more stable. In this sample `Rakefile`, we give the load path the absolute path to the `lib` and `test` subdirectories. Adding the `test` directory to the load path is useful if you need to `require` a library full of test utility methods:

```
require 'rake/testtask'

lib_dir = File.expand_path('lib')
test_dir = File.expand_path('test')

Rake::TestTask.new("test") do |t|
    t.libs = [lib_dir, test_dir]
    t.pattern = "test/**/tc_*.rb"
    t.warning = true
end
```

Test suites

As a project grows, it takes longer and longer to run all the test cases. This is bad for the habit we're trying to inculcate, where you run the tests whenever you make a change. To solve this problem, group the test cases into *test suites*. Depending on the project, you might have a test suite of all test cases concerning file I/O, another suite for the console interface, and so on.

Let's say that when you're working on the `DataFile` class, you can get away with only running the file I/O test suite. But before releasing a new version of the software, you need to run all the test cases.

Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

To create a Rake test suite, instantiate a `Rake::TestTask` instance, and set the `test_files` attribute to something other than the complete list of test files. This sample Rakefile splits up the test files into two suites.

```
require 'rake/testtask'

Rake::TestTask.new('test-file') do |t|
    t.test_files = ['test/tc_datafile.rb',
                    'test/tc_datafilewriter.rb',
                    'test/tc_datafilereader.rb']
    t.warning = true
end

Rake::TestTask.new('test-console') do |t|
    t.test_files = ['test/tc_console.rb',
                    'test/tc_prettyprinter.rb']
    t.warning = true
end
```

Invoking `rake test-file` runs the tests related to file I/O, and invoking `rake test-console` tests the console interface. The only thing missing is a task that runs all tests. You can either use the all-inclusive task from the Rakefile given in the Solution, or you can create a task that has all the test suites as prerequisites:

```
task 'test' => ['test-file', 'test-console']
```

When this `test` task is invoked, Rake runs the `test-file` suite and then the `test-console` suite. Each suite is run in its own Ruby interpreter.

See Also

- Recipe 17.8, "Running Unit Tests"
- For a guide to the options available to the `TestTask` class, consult its RDoc; it's available at, for instance, http://rake.rubyforge.org/classes/Rake/TestTask.html

Recipe 19.2. Automatically Generating Documentation

Credit: Stefan Lang

Problem

You want to automatically create HTML pages from the RDoc formatted comments in your code, and from other RDoc formatted files.

Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

Solution

Within your Rakefile, require the `rake/rdoctask` library and create a new `Rake::RDocTask`. Here's a typical example:

```
require 'rake/rdoctask'

Rake::RDocTask.new('rdoc') do |t|
  t.rdoc_files.include('README', 'lib/**/*.rb')
  t.main = 'README'
  t.title = "MyLib API documentation"
end
```

Now you can run the command `rake rdoc` from a shell in your project's top-level directory. This particular Rake task creates API documentation for all files under the *lib* directory (and its subdirectories) whose names end in `.rb`. Additionally, the RDoc-formatted contents of the top-level `README` file will appear on the front page of the documentation.

The HTML output files are written under your project's *%(filename)html%* directory. To read the documentation, point your browser to *%(filename)html/index.html%*. The browser will show "MyLib API documentation" (that is, the value of the task's `title`) as the page title.

Discussion

It is common practice among authors of Ruby libraries to document a library's API with RDoc-formatted text. Since Ruby 1.8.1, a standard Ruby installation contains the `rdoc` tool, which extracts the RDoc comments from source code and creates nicely formatted HTML pages.

Unlike the tasks you define from scratch with the `task` method, but like the `TestTask` covered in Recipe 19.1, `Rake::RDocTask.new` takes a code block, which is executed immediately at task definition time. The code block lets you customize how your RDoc documentation should look. After running your code block, the `Rake::RDocTask` object defines *three* new Rake tasks:

`rdoc`

: Updates the HTML documentation by running RDoc.

`clobber_rdoc`

: Removes the directory and its contents created by the `rdoc` task.

Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

`rerdoc`

Force a rebuild of the HTML-documentation. Has the same effect as running `clobber_rdoc` followed by `rdoc`.

Now we know enough to integrate the `Rake::RDocTask` into a more useful Rakefile. Suppose we want a task that uploads the documentation to RubyForge (or another site), and a general cleanup task that removes the generated HTML-documentation as well as all backup files in the project directory. To keep the example simple, I've inserted comments instead of the actual commands for uploading and removing the files; see Recipes 19.3 and 19.8 for more realistic examples.

```
require 'rake/rdoctask'

Rake::RDocTask.new('rdoc') do |t|
  t.rdoc_files.include('README', 'lib/**/*.rb')
  t.main = 'README'
  t.title = "MyLib API documentation"
end

desc 'Upload documentation to RubyForge.'
task 'upload' => 'rdoc' do
   # command(s) to upload html/ and contents to RubyForge
end
desc 'Remove generated and backup files.'
task 'clobber' => 'clobber_rdoc' do
    # command(s) to remove all files ending in ~ or .bak
end
```

Finally, we make the default task dependent on the `rdoc` task, so that RDoc gets built automatically when you invoke `rake` with no task. If there already is a default task, this code will simply add another dependency to the existing task:

```
task :default => ['rdoc']
```

Available attributes

Here's a list of attributes that can be set in the block given to `Rake::RDocTask.new`.

`rdoc_dir`

Name of the directory where the produced HTML files go. Defaults to *html*.

`title`

A title for the produced HTML pages.

Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

`main`

Name of the input file whose contents should appear at the initial page of the HTML output.

`template`

Name of the template to be used by RDoc.

`rdoc_files`

Initialized to an empty filelist. Just call the `include` method with the names of files to be documented, or glob patterns matching multiple files.

`options`

An array of arguments to be passed directly to `rdoc`. Use this if none of the other attributes fits your needs. Run `rdoc --help` for a list of available options.

See Also

- Recipe 19.3, "Cleaning Up Generated Files"
- Recipe 19.8, "A Generic Project Rakefile"
- The RDoc documentation for the `Rake::RDocTask` class (http://rake.rubyforge.org/classes/Rake/RDocTask.html)

Recipe 19.3. Cleaning Up Generated Files

Credit: Stefan Lang

Problem

You want to clean up files that aren't actually part of your project: generated files, backup files, and so on.

Solution

Within your Rakefile, require the `rake/clean` library to get access to the `clean` and `clobber` tasks. Put glob patterns for all your generated files in the `CLOBBER FileList`. Put glob patterns for all other scratch files in the `CLEAN FileList`.

Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

By default, CLEAN also includes the patterns **/*~, **/*.bak, and **/core. Here's a typical set of CLOBBER and CLEAN files:

```
require 'rake/clean'

# Include the "pkg" and "doc" directories and their contents.
# Include all files ending in ".o" in the current directory
# and its subdirectories (recursively).
CLOBBER.include('pkg', 'doc', '**/*.o')

# Include InstalledFiles and .config: files created by setup.rb.
# Include temporary files created during test run.
CLEAN.include('InstalledFiles', '.config', 'test/**/*.tmp')
```

Run rake clean to remove all files specified by the CLEAN filelist, and rake clobber to remove the files specified by *both* file lists.

Discussion

The rake/clean library initializes the constants CLEAN and CLOBBER to new Rake::FileList instances. It also defines the tasks clean and clobber, making clean a prerequisite of clobber. The idea is that rake clean removes any files that might need to be recreated once your program changes, while rake clobber returns your source tree to a completely pristine state.

Other Rake libraries define cleanup tasks that remove certain products of their main tasks. An example: the packaging libraries create a task called clobber_package, and make it a prerequisite of clobber. Running rake clobber on such a project removes the package files: you don't have to explicitly include them in your CLOBBER list.

You can do the same thing for your own tasks: rather than manipulate CLEAN and CLOBBER, you can create a custom cleanup task and make it a prerequisite of clean or clobber. The following code is a different way of making sure that rake clobber removes any precompiled object files:

```
desc 'Remove all object files.'
task 'clobber_objects' do
  rm_f FileList['**/*.o']
end

# Make clobber_objects a prerequisite of the preexisting clobber task
task 'clobber' => 'clobber_objects'
```

Now you can run rake clobber_objects to remove all object files, and rake clobber to remove all other unwanted files as well.

Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

See Also

- The documentation for the `Dir.glob` method describes the format for the patterns accepted by `FileList#include`; it's accessible via `ri Dir.glob`
- Online documentation for the `rake/clean` library (http://rake.rubyforge.org/files/lib/rake/clean_rb.html)

Recipe 19.4. Automatically Building a Gem

Credit: Stefan Lang

Problem

You want to automatically build a gem package for your application or library whenever you do a release.

Solution

Require the `rake/gempackagetask` library within your Rakefile, and create a `Gem::Specification` instance that describes your project. Feed it to the `Rake::GemPackageTask` constructor, which automatically defines a number of gem-related tasks:

```
require 'rake/gempackagetask'

# Create a gem specification
gem_spec = Gem::Specification.new do |s|
  s.name = 'docbook'
  s.version = '1.0.0'
  s.summary = 'DocBook formatting program and library.'

  # Files containing Test::Unit test cases.
  s.test_files = FileList['tests/**/*']

  # Executable scripts under the "bin" directory.
  s.executables = ['voc']

  # List of other files to be included.
  s.files = FileList['README', 'ChangeLog', 'lib/**/*.rb']
end

Rake::GemPackageTask.new(gem_spec) do |pkg|
  pkg.need_zip = false
  pkg.need_tar = false
end
```

Run the command `rake package`, and (assuming those files actually exist), Rake will build a gem file `docbook-1.0.0.gem` under the `pkg/` directory.

Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

Discussion

The RubyGems library provides the `Gem::Specification` class, and Rake provides the `Rake::GemPackageTask` class that uses it. Creating a new `Rake::GemPackageTask` object automatically defines the three tasks: `package`, `clobber_package`, and `repackage`.

The `package` task builds a gem inside the project's *pkg/* directory. The `clobber_package` task removes the *pkg/* directory and its contents. The `repackage` task just invokes `clobber_package` to remove any old package file, and then invokes `package` to rebuild them from scratch.

The example above sets to false the attributes `need_zip` and `need_tar` of the `Rake::GemPackageTask`. If you set them to true, then in addition to a gem you'll get a ZIP file and a gzipped tar archive containing the same files as the gem. Note that Rake uses the `zip` and `tar` command-line tools, so if your system doesn't provide them (the way a standard Windows installation doesn't), the `package` task won't be able to create these ZIP or tar archives.

The `package` task recreates a package file only if it doesn't already exist, or if you've updated one of your input files since you last built the package. The most common problem you'll run into here is that you'll decide to stop packaging a certain file. Rake won't recognize the change (since the file is gone), and running `rake package` won't do anything. To force a rebuild of your package file(s), run `rake repackage`.

See Also

- Recipe 18.6, "Packaging Your Code as a Gem"
- The `Gem::Specification` reference describes everything you can do when creating a gem (http://docs.rubygems.org/read/chapter/20)
- The Rake alternative Rant can build gems, ZIP files, and tarballs without calling out to external tools; point your browser to http://make.ruby-co.de

Recipe 19.5. Gathering Statistics About Your Code

Credit: Stefan Lang

Problem

You want to gather statistics about your Ruby project, like the total number of lines of code.

Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

Solution

Here's a class that parses Ruby source files and gathers statistics. Put this in `scriptlines.rb` in your project's top-level directory.

```
# scriptlines.rb
# A ScriptLines instance analyses a Ruby script and maintains
# counters for the total number of lines, lines of code, etc.
class ScriptLines

  attr_reader :name
  attr_accessor :bytes, :lines, :lines_of_code, :comment_lines

  LINE_FORMAT = '%8s %8s %8s %8s %s'

  def self.headline
    sprintf LINE_FORMAT, "BYTES", "LINES", "LOC", "COMMENT", "FILE"
  end

  # The 'name' argument is usually a filename
  def initialize(name)
    @name = name
    @bytes = 0
    @lines = 0    # total number of lines
    @lines_of_code = 0
    @comment_lines = 0
  end

  # Iterates over all the lines in io (io might be a file or a
  # string), analyses them and appropriately increases the counter
  # attributes.
  def read(io)
    in_multiline_comment = false
    io.each { |line|
      @lines += 1
      @bytes += line.size
      case line
      when /^=begin(\s|$)/
        in_multiline_comment = true
        @comment_lines += 1
      when /^=end(\s|$)/:
        @comment_lines += 1
        in_multiline_comment = false
      when /^\s*#/
        @comment_lines += 1
      when /^\s*$/
        # empty/whitespace only line
      else
        if in_multiline_comment
          @comment_lines += 1
        else
          @lines_of_code += 1
        end
      end
    }
  end

  # Get a new ScriptLines instance whose counters hold the
  # sum of self and other.
  def +(other)
    sum = self.dup
    sum.bytes += other.bytes
    sum.lines += other.lines
    sum.lines_of_code += other.lines_of_code
    sum.comment_lines += other.comment_lines
    sum
  end
```


Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

```
  # Get a formatted string containing all counter numbers and the
  # name of this instance.
  def to_s
    sprintf LINE_FORMAT,
      @bytes, @lines, @lines_of_code, @comment_lines, @name
  end
end
```

To tie the class into your build system, give your Rakefile a `stats` task like the following. This task assumes that the Rakefile and `scriptlines.rb` are in the same directory:

```
task 'stats' do
  require 'scriptlines'

  files = FileList['lib/**/*.rb']

  puts ScriptLines.headline
  sum = ScriptLines.new("TOTAL (#{files.size} file(s))")

  # Print stats for each file.
  files.each do |fn|
    File.open(fn) do |file|
      script_lines = ScriptLines.new(fn)
      script_lines.read(file)
      sum += script_lines
      puts script_lines
    end
  end

  # Print total stats.
  puts sum
end
```

Discussion

`ScriptLines` performs a very basic parsing of Ruby code: it divides a source file into blank lines, comment lines, and lines containing Ruby code. If you want more detailed information, you can `include` each file and get more information about the defined classes and methods with reflection or an extension like Parse Tree.

Invoke the `stats` task to run all the Ruby scripts beneath your `lib/` directory through `ScriptLines`. The following example output is for the highline library:

```
$ rake stats
(in /usr/local/lib/ruby/gems/1.8/gems/highline-1.0.1)
BYTES  LINES    LOC  COMMENT   FILE
18626    617    360    196   lib/highline.rb
12745    375    168    181   lib/highline/menu.rb
15760    430    181    227   lib/highline/question.rb
801       25      7     14   lib/highline/import.rb
47932   1447    716    618   TOTAL (4 scripts)
```

`BYTES` is the file size in bytes, `LINES` the number of total lines in each file, `LOC` stands for "Lines Of Code," and `COMMENT` is the number of comment-only lines.

Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

These simple metrics are good for gauging the complexity of a project, but don't use them as a measure of day-to-day progress. Complexity is not the same as progress, and a good day's work might consist of replacing a hundred lines of code with ten.

See Also

- `ri Kernel#sprintf`
- The RDoc documentation for Rake's `FileList` class (http://rake.rubyforge.org/classes/Rake/FileList.html)
- The `ParseTree` extension (http://rubyforge.org/projects/parsetree/)

Recipe 19.6. Publishing Your Documentation

Credit: Stefan Lang

Problem

You want to automatically update your project's web site on RubyForge (or some other site) with generated documentation or custom pages.

Solution

As seen in Recipe 19.2, Rake provides a `RDocTask` for generating RDoc documentation:

```
require 'rake/rdoctask'

html_dir = 'doc/html'
library = 'MyLib'
Rake::RDocTask.new('rdoc') do |t|
  t.rdoc_files.include('README', 'lib/**/*.rb')
  t.main = 'README'
  t.title = "#{library} API documentation"
  t.rdoc_dir = html_dir
end
```

To upload your generated documentation to RubyForge, use this task along with the `upload-docs` task defined below. The Unix `scp` command-line tool does the actual work of uploading:

```
# Define your RubyForge username and your project's Unix name here:
rubyforge_user = 'user'
rubyforge_project = 'project'
rubyforge_path = "/var/www/gforge-projects/#{rubyforge_project}/"
desc 'Upload documentation to RubyForge.'
task 'upload-docs' => ['rdoc'] do
     sh "scp -r #{html_dir}/* " +
         "#{rubyforge_user}@rubyforge.org:#{rubyforge_path}"
end
```


Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

Discussion

Set off the publishing process by invoking `rake upload-docs`. The `upload-docs` task has the `rdoc` task as a prerequisite, so the HTML pages under `doc/html/` will be created if necessary.

Then `scp` prompts for your RubyForge account password. Enter it, and all files under `doc/html/` and its subdirectories will be uploaded to RubyForge. The docs will become available under http://project.rubyforge.org/, where "project" is the Unix name of your project. Now your users can read your RDoc online without having to generate it themselves. Your documentation will also show up in web search results.

Rake's `sh` method starts an instance of the OS's standard shell. This feature is used to run the `scp` command-line tool. This means that this recipe will only work if `scp` is installed on your system.

The `scp` command copies all the files that the RDoc placed under `doc/html/`, to the root of your project's web site on the RubyForge server. In effect, the main page of the API documentation will appear as your project's homepage. Some RubyForge projects don't have a custom homepage, so this is a good place to put the RDoc. If you want a custom homepage, just copy the RDoc into a different directory by changing `rubyforge_path`:

```
rubyforge_path =  "/var/www/gforge-projects/#{rubyforge_project}/rdoc/"
```

You'll have to manually create the `rdoc` directory before you can use the `scp` shortcut. After that, the generated RDoc will show up at http://project.rubyforge.org/rdoc/, and you can link to it from your custom homepage with a relative link to `rdoc/`.

You can make Rake upload your custom homepage as well, of course. Just add an `upload-site` task that uploads your custom homepage and other web content. Make `upload-site` and `upload-docs` prerequisites of an overarching `publish` task:

```
website_dir = 'site'
desc 'Update project website to RubyForge.'
task 'upload-site' do
  sh "scp -r #{website_dir}/* " +
    "#{rubyforge_user}@rubyforge.org:/var/www/gforge-projects/project/"
end

desc 'Update API docs and project website to RubyForge.'
task 'publish' => ['upload-docs', 'upload-site']
```

Now you can run `rake publish` to update the generated API documentation, and upload it together with the rest of the web site to RubyForge. The `publish` task can be just one more prerequisite for an overarching `release` task.

Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

Of course, you can use this same technique if you're using a web host other than RubyForge: just change the destination host of the `scp` command.

See Also

- Recipe 17.11, "Documenting Your Application," covers writing RDoc documentation
- Recipe 19.2, "Automatically Generating Documentation"

Recipe 19.7. Running Multiple Tasks in Parallel

Problem

Your build process takes too long to run. Rake finishes copying one set of files only to start copying another set. You could save time by running these tasks in parallel, instead of stringing them one after another.

Solution

Define a task using the `multitask` function instead of `task`. Each of that task's prerequisites will be run in a separate thread.

In this code, I'll define two long-running tasks:

```
task 'copy_docs' do
  # Simulate a large disk copy.
  sleep 5
end

task 'compile_extensions' do
  # Simulate a C compiler compiling a bunch of files.
  sleep 10
end

task 'build_serial' => ['copy_docs', 'compile_extensions']
multitask 'build_parallel' => ['copy_docs', 'compile_extensions']
```

The `build_serial` task runs in about 15 seconds, but the `build_parallel` task does the same thing in about 10 seconds.

Discussion

A `multitask` runs just like a normal `task`, except that each of its dependencies runs in a separate thread. When running the dependencies of a `multitask`, Rake first finds any common secondary dependencies of these dependencies, and runs them first. It then spawns a separate thread for each dependency, so that they can run simultaneously.

Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

Consider three tasks, `ice_cream`, `cheese`, and `yogurt`, all of which have a dependency on `buy_milk`. You can run the first three tasks in separate threads with a `multitask`, but Rake will run `buy_milk` before creating the threads. Otherwise, `ice_cream`, `cheese`, and `yogurt` would *all* trigger `buy_milk`, wasting time.

When your tasks spend a lot of time blocking on I/O operations (as many Rake tasks do), using a `multitask` can speed up your builds. Unfortunately, it can also cause the same problems you'll see with any multithreaded code. If you've got a fancy Rakefile, in which the tasks keep state inside Ruby data structures, you'll need to synchronize access to those data structures to prevent multithreading problems.

You may also have problems converting a `task` to a `multitask` if your dependencies are set up incorrectly. Take the following example:

```
task 'build' => ['compile_extensions', 'run_tests', 'generate_rdoc']
```

The unit tests can't run if the compiled extensions aren't available, so `:compile_extensions` shouldn't be in this list at all: it should be a dependency of `:run_tests`. You might not notice this problem as long as you're using `task` (because `:compile_extensions` runs before `:run_tests` anyway), but if you switch to a `multitask` your tests will start failing. Fixing your dependencies will solve the problem.

The `multitask` method is available only in Rake 0.7.0 and higher.

See Also

- Chapter 20

Recipe 19.8. A Generic Project Rakefile

Credit: Stefan Lang

Every project's Rakefile is different, but most Ruby projects can be handled by very similar Rakefiles. To close out the chapter, we present a generic Rakefile that includes most of the tasks covered in this chapter, and a few (such as compilation of C extensions) that we only hinted at.

This Rakefile will work for pure Ruby projects, Ruby projects with C extensions, and projects that are *only* C extensions. It defines an overarching task called `publish` that builds the project, runs tests, generates RDoc, and releases the whole thing on Ruby-Forge.

Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

It's a big file, but you don't have to use all of it. The `publish` task is made entirely of smaller tasks, and you can pick and choose from those smaller tasks to build your own Rakefile. For a simple project, you can just customize the settings at the beginning of the file, and ignore the rest. Of course, you can also extend this Rakefile with other tasks, like the `stats` task presented in Recipe 19.5.

This Rakefile assumes that you follow the directory layout conventions laid down by the `setup.rb` script, even if you don't actually use `setup.rb` to install your project. For instance, it assumes you put your Ruby files in `lib/` and your unit tests in `test/`.

First, we include Rake libraries that make it easy to define certain kinds of tasks:

```
# Rakefile
require "rake/testtask"
require "rake/clean"
require "rake/rdoctask"
require "rake/gempackagetask"
```

You'll need to configure these variables:

```
# The name of your project
PROJECT = "MyProject"

# Your name, used in packaging.
MY_NAME = "Frodo Beutlin"

# Your email address, used in packaging.
MY_EMAIL = "frodo.beutlin@my.al"

# Short summary of your project, used in packaging.
PROJECT_SUMMARY = "Commandline program and library for …"

# The project's package name (as opposed to its display name). Used for
# RubyForge connectivity and packaging.
UNIX_NAME = "my_project"

# Your RubyForge user name.
RUBYFORGE_USER = ENV["RUBYFORGE_USER"] || "frodo"

# Directory on RubyForge where your website's files should be uploaded.
WEBSITE_DIR = "website"

# Output directory for the rdoc html files.
# If you don't have a custom homepage, and want to use the RDoc
# index.html as homepage, just set it to WEBSITE_DIR.
RDOC_HTML_DIR = "#{WEBSITE_DIR}/rdoc"
```

Now we start defining the variables you probably won't have to configure. The first set is for your project includes C extensions, to be compiled with `extconf.rb`, these variables let Rake know where to find the source and header files, as well as `extconf.rb` itself:

```
# Variable settings for extension support.
EXT_DIR = "ext"
HAVE_EXT = File.directory?(EXT_DIR)
EXTCONF_FILES = FileList["#{EXT_DIR}/**/extconf.rb"]
EXT_SOURCES = FileList["#{EXT_DIR}/**/*.{c,h}"]
```


Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

```
# Eventually add other files from EXT_DIR, like "MANIFEST"
EXT_DIST_FILES = EXT_SOURCES + EXTCONF_FILES
```

This next piece of code automatically finds the current version of your project, so long as you define a file `my_project.rb`, which defines a module `MyProject` containing a constant `VERSION`. This is convenient because you don't have to change the version number in your gemspec whenever you change it in the main program.

```
REQUIRE_PATHS = ["lib"]
REQUIRE_PATHS << EXT_DIR if HAVE_EXT
$LOAD_PATH.concat(REQUIRE_PATHS)
# This library file defines the MyProject::VERSION constant.
require "#{UNIX_NAME}"
PROJECT_VERSION = eval("#{PROJECT}::VERSION") # e.g., "1.0.2"
```

If you don't want to set it up this way, you can:

- Have the Rakefile scan a source file for the current version.
- Use an environment variable.

Hardcode `PROJECT_VERSION` here, and change it whenever you do a new version.

These variables here are for the `rake clobber` tasks: they tell Rake to clobber files generated when you run `setup.rb` or build your C extensions.

```
# Clobber object files and Makefiles generated by extconf.rb.
CLOBBER.include("#{EXT_DIR}/**/*.{so,dll,o}", "#{EXT_DIR}/**/Makefile")
# Clobber .config generated by setup.rb.
CLOBBER.include(".config")
```

Now we start defining file lists and options for the various tasks. If you have a non-standard file layout, you can change these variables to reflect it.

```
# Options common to RDocTask AND Gem::Specification.
#   The --main argument specifies which file appears on the index.html page
GENERAL_RDOC_OPTS = {
  "--title" => "#{PROJECT} API documentation",
  "--main" => "README.rdoc"
}

# Additional RDoc formatted files, besides the Ruby source files.
RDOC_FILES = FileList["README.rdoc", "Changes.rdoc"]
# Remove the following line if you don't want to extract RDoc from
# the extension C sources.
RDOC_FILES.include(EXT_SOURCES)

# Ruby library code.
LIB_FILES = FileList["lib/**/*.rb"]

# Filelist with Test::Unit test cases.
TEST_FILES = FileList["test/**/tc_*.rb"]

# Executable scripts, all non-garbage files under bin/.
BIN_FILES = FileList["bin/*"]

# This filelist is used to create source packages.
```


Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

```
# Include all Ruby and RDoc files.
DIST_FILES = FileList["**/*.rb", "**/*.rdoc"]
DIST_FILES.include("Rakefile", "COPYING")
DIST_FILES.include(BIN_FILES)
DIST_FILES.include("data/**/*", "test/data/**/*")
DIST_FILES.include("#{WEBSITE_DIR}/**/*.{html,css}", "man/*.[0-9]")
# Don't package files which are autogenerated by RDocTask
DIST_FILES.exclude(/^(\.\/)?#{RDOC_HTML_DIR}(\/|$)/)
# Include extension source files.
DIST_FILES.include(EXT_DIST_FILES)
# Don't package temporary files, perhaps created by tests.
DIST_FILES.exclude("**/temp_*", "**/*.tmp")
# Don't get into recursion…
DIST_FILES.exclude(/^(\.\/)?pkg(\/|$)/)
```

Now we can start defining the actual tasks. First, a task for running unit tests:

```
# Run the tests if rake is invoked without arguments.
task "default" => ["test"]

test_task_name = HAVE_EXT ? "run-tests" : "test"
Rake::TestTask.new(test_task_name) do |t|
  t.test_files = TEST_FILES
  t.libs = REQUIRE_PATHS
end
```

Next a task for building C extensions:

```
# Set an environment variable with any configuration options you want to
# be passed through to "setup.rb config".
CONFIG_OPTS = ENV["CONFIG"]
if HAVE_EXT
  file_create ".config" do
    ruby "setup.rb config #{CONFIG_OPTS}"
  end

  desc "Configure and make extension. " +
    "The CONFIG variable is passed to `setup.rb config'"
  task "make-ext" => ".config" do
    # The -q option suppresses messages from setup.rb.
    ruby "setup.rb -q setup"
  end

  desc "Run tests after making the extension."
  task "test" do
    Rake::Task["make-ext"].invoke
    Rake::Task["run-tests"].invoke
  end
end
```

A task for generating RDoc:

```
# The "rdoc" task generates API documentation.
Rake::RDocTask.new("rdoc") do |t|
  t.rdoc_files = RDOC_FILES + LIB_FILES
  t.title = GENERAL_RDOC_OPTS["--title"]
  t.main = GENERAL_RDOC_OPTS["--main"]
  t.rdoc_dir = RDOC_HTML_DIR
end
```


Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

Now we define a gemspec for the project, using the customized variables from the beginning of the file. We use this to define a task that builds a gem.

```
GEM_SPEC = Gem::Specification.new do |s|
  s.name = UNIX_NAME
  s.version = PROJECT_VERSION
  s.summary = PROJECT_SUMMARY
  s.rubyforge_project = UNIX_NAME
  s.homepage = "http://#{UNIX_NAME}.rubyforge.org/"
  s.author = MY_NAME
  s.email = MY_EMAIL
  s.files = DIST_FILES
  s.test_files = TEST_FILES
  s.executables = BIN_FILES.map { |fn| File.basename(fn) }
  s.has_rdoc = true
  s.extra_rdoc_files = RDOC_FILES
  s.rdoc_options = GENERAL_RDOC_OPTS.to_a.flatten
  if HAVE_EXT
    s.extensions = EXTCONF_FILES
    s.require_paths >> EXT_DIR
  end
end

# Now we can generate the package-related tasks.
Rake::GemPackageTask.new(GEM_SPEC) do |pkg|
  pkg.need_zip = true
  pkg.need_tar = true
end
```

Here's a task to publish RDoc and static HTML content to RubyForge:

```
desc "Upload website to RubyForge. " +
  "scp will prompt for your RubyForge password."
task "publish-website" => ["rdoc"] do
  rubyforge_path = "/var/www/gforge-projects/#{UNIX_NAME}/"
  sh "scp -r #{WEBSITE_DIR}/* " +
    "#{RUBYFORGE_USER}@rubyforge.org:#{rubyforge_path}",
    :verbose => true
end
```

Here's a task that uses the `rubyforge` command to log in to RubyForge and publish the packaged software as a release of the project:

```
task "rubyforge-setup" do
  unless File.exist?(File.join(ENV["HOME"], ".rubyforge"))
    puts "rubyforge will ask you to edit its config.yml now."
    puts "Please set the `username' and `password' entries"
    puts "to your RubyForge username and RubyForge password!"
    puts "Press ENTER to continue."
    $stdin.gets
    sh "rubyforge setup", :verbose => true
  end
end

task "rubyforge-login" => ["rubyforge-setup"] do
  # Note: We assume that username and password were set in
  # rubyforge's config.yml.
  sh "rubyforge login", :verbose => true
end

task "publish-packages" => ["package", "rubyforge-login"] do
  # Upload packages under pkg/ to RubyForge
  # This task makes some assumptions:
  # * You have already created a package on the "Files" tab on the
```


Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

```
  #   RubyForge project page. See pkg_name variable below.
  # * You made entries under package_ids and group_ids for this
  #   project in rubyforge's config.yml. If not, eventually read
  #   "rubyforge --help" and then run "rubyforge setup".
  pkg_name = ENV["PKG_NAME"] || UNIX_NAME
  cmd = "rubyforge add_release #{UNIX_NAME} #{pkg_name} " +
        "#{PROJECT_VERSION} #{UNIX_NAME}-#{PROJECT_VERSION}"
  cd "pkg" do
  sh(cmd + ".gem", :verbose => true)
    sh(cmd + ".tgz", :verbose => true)
    sh(cmd + ".zip", :verbose => true)
  end
end
```

Now we're in good shape to define some overarching tasks. The `prepare-release` task makes sure the code works, and creates a package. The top-level `publish` task does all that and also performs the actual release to RubyForge:

```
# The "prepare-release" task makes sure your tests run, and then generates
# files for a new release.
desc "Run tests, generate RDoc and create packages."
task "prepare-release" => ["clobber"] do
  puts "Preparing release of #{PROJECT} version #{VERSION}"
  Rake::Task["test"].invoke
  Rake::Task["rdoc"].invoke
  Rake::Task["package"].invoke
end

# The "publish" task is the overarching task for the whole project. It
# builds a release and then publishes it to RubyForge.
desc "Publish new release of #{PROJECT}"
task "publish" => ["prepare-release"] do
  puts "Uploading documentation…"
  Rake::Task["publish-website"].invoke
  puts "Checking for rubyforge command…"
  `rubyforge --help`
  if $? == 0
    puts "Uploading packages…"
    Rake::Task["publish-packages"].invoke
    puts "Release done!"
  else
    puts "Can't invoke rubyforge command."
    puts "Either install rubyforge with 'gem install rubyforge'"
    puts "and retry or upload the package files manually!"
  end
end
```

To get an overview of this extensive Rakefile, run `rake -T`:

```
$ rake -T
rake clean            # Remove any temporary products.
rake clobber          # Remove any generated file.
rake clobber_package  # Remove package products
rake clobber_rdoc     # Remove rdoc products
rake package          # Build all the packages
rake prepare-release  # Run tests, generate RDoc and create packages.
rake publish          # Publish new release of MyProject
rake publish-website  # Upload website to RubyForge. scp will prompt for your
                      # RubyForge password.
rake rdoc             # Build the rdoc HTML Files
rake repackage        # Force a rebuild of the package files
rake rerdoc           # Force a rebuild of the RDOC files
rake test             # Run tests for test
```


Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

Here's the idea behind `prepare-release` and `publish`: suppose you get a bug report and you need to do a new release. You fix the bug and add a test case to make sure it stays fixed. You check your fix by running the tests with `rake` (or `rake test`). Then you edit a library file and bump up the project's version number.

Now that you're confident the bug is fixed, you can run `rake publish`. This task builds your package, tests it, packages it, and uploads it to RubyForge. You didn't have to do any work besides fix the bug and increment the version number.

The `rubyforge` script is a command-line tool that performs common interactions with RubyForge, like the creation of new releases. To use the `publish` task, you need to install the `rubyforge` script and do some basic setup for it. The alternative is to use the `prepare-release` task instead of `publish`, and upload all your new packages manually.

Note that Rake uses the `zip` and `tar` command-line tools to create the ZIP file and tarball packages. These tools are not available on most Windows installations. If you're on windows, set the attributes `need_tar` and `need_zip` of the `Rake::GemPackageTask` to `false`. With these attributes, the `package` task only creates a gem package.

See Also

- Recipe 19.4, "Automatically Building a Gem"
- You can download the `rubyforge` script from http://rubyforge.org/projects/codeforpeople/

Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01