Community Experience Distilled

Rake Task Management Essentials

Deploy, test, and build software to solve real-world automation challenges using Rake

Andrey Koleshko

[PACKT] open source*
PUBLISHING
community experience distilled

Rake Task Management Essentials

Deploy, test, and build software to solve real-world automation challenges using Rake

Andrey Koleshko

[PACKT] PUBLISHING open source community experience distilled

BIRMINGHAM - MUMBAI

Rake Task Management Essentials

First published: April 2014

Production Reference: 1140414

Published by Packt Publishing Ltd.
Livery Place
35 Livery Street
Birmingham B3 2PB, UK.

ISBN 978-1-78328-077-3

www.packtpub.com

Cover Image by Kim Nousaine (kimberlyjg@icloud.com)

Credits

Author
Andrey Koleshko

Reviewers
Mario Miguel Agüero Obando
Stuart Ellis
Avinasha Sastry

Commissioning Editor
Grant Mizen

Acquisition Editor
Neha Nagwekar

Content Development Editor
Priya Singh

Technical Editor
Dennis John

Copy Editor
Stuti Srivastava

Project Coordinator
Harshal Ved

Proofreaders
Simran Bhogal
Ameesha Green

Indexer
Mehreen Deshmukh

Graphics
Sheetal Aute

Production Coordinator
Arvindkumar Gupta

Cover Work
Arvindkumar Gupta

About the Author

Andrey Koleshko had his first touch with programming while at school, when he worked on Pascal. He had been solving basic algorithmic tasks. The first programming language he used at the beginning of his career was Java. He worked with this language for a year and then migrated to the Ruby language, which he worked with for the next four years. Of these four years, he worked with Altoros for three. He had an amazing time there, learning the language and technologies deeply.

Currently, he works at a local cloud hosting company. The company change provided him with the opportunity to deal with a lot of challenges concerning application architecture, code testing, debugging, and deployment processes. As a result, he has been able to contribute to some famous Ruby libraries. More detailed information about his contributions can be found on GitHub at `http://github.com/ka8725`.

He mostly works with the Rails framework. He openly shares all of his thoughts and his most interesting experiences through his blog at `http://railsguides.net`. He has recently started to learn the Python programming language.

He lives in Minsk, Belarus, and likes to watch and play sports such as soccer, ping-pong, and volleyball. He also likes travelling to tropical countries with his wife. Teaching people gives him immense pleasure.

Acknowledgements

I'm thankful to many people who've helped me write this book. But firstly, I would like to thank the publishers who offered me an opportunity to write this book. If it wasn't for them, who knows what would have happened with this book. Thankfully, the Packt Publishing team was very supportive and helped me deliver this useful book with high quality. A big thanks goes to Sergey Avseyev, who always supported me in the technical and difficult parts of the book. He also made me believe that I could write this book from scratch. I would also like to thank Lee Hambley, who shared his experience with me, and as a result, the last chapter of the book is more hands-on. I have no doubt now that Rake has a very successful future despite other competitive tools.

I'm grateful to my wife for allowing me to allocate enough time to write the chapters. Lastly, I would like to acknowledge the creator of Rake, Jim Weirich. He created a really great and powerful tool. Unfortunately, he won't be with us to see this book published. He passed away recently. But anyway, I believe that he would be happy that his creation helped develop this book.

About the Reviewers

Mario Miguel Agüero Obando is a software engineer with experience in both frontend and backend sides of software development. He has worked in intensive data processing applications and also in new UX designs.

He is also an experienced programming trainer and has reviewed several technical books.

Stuart Ellis works for a Ruby on Rails and mobile software development company, where he wears many hats. He has also worked as a .NET and Ruby programmer, tamed various brands of databases, managed different combinations of Windows and Linux, and studied history. He has always been a Yorkshireman.

Avinasha Sastry has been involved in technology and startups right from his college days. He has never worked in big companies because he loves the business challenges in startups as much as he loves technology. He has been working with SupportBee for the last three years. He is an avid reader, a Harry Potter fan, and a globetrotter.

Table of Contents

Preface

Maybe every Ruby developer who is familiar with Rails knows what Rake is. However, many of them are unaware of the complete power of this tool and its real aim. The goal of this book is to improve this situation.

Have you ever had to perform boring, repetitive tasks while deploying your project? I assume here that a project is not only something written in Ruby or another programming language, but it can also consist of operations with files. For example, it might be a book or the documentation of a project that you are writing in Markdown and then compiling into HTML. Or it can be compiling a lot of files to one package. Have you ever wished to build a project or run tests on a project whenever it undergoes a change? All this stuff is easily made possible by programs called software management tools. Rake is one such program.

Rake was initially implemented as a Ruby version of Make—a commonly used build utility. However, calling Rake a build utility undermines its true power. Rake is actually an automation tool—it's a way to put all those tasks that you perform under the project into one neat and tidy place.

Basically, build automation includes the following processes:

- Compiling the computer source code into binary code
- Packaging the binary code
- Running tests
- Deployment to production systems
- Creating documentation and/or release notes

Rake can be used in all these situations, and this book shows you how Rake performs all the steps. After reading this book, you will know Rake better and be able to write more clear and robust Rake code.

What this book covers

Chapter 1, *The Software Task Management Tool – Rake*, introduces you to the basic usage of Rake and its command-line utilities. You will learn what a rake task is and how to set dependencies between rake tasks, what a default rake task is, `Rakefile`, and the global `Rakefile`. This chapter also contains information about the Rake project structure and how to organize the code.

Chapter 2, *Working with Files*, explains the foundational features of Rake that help us work with files. This is mandatory information because of Rake's orientation—it is built to be an automation tool. You will see that there is a special rake task for file processing called `file`. The main part of the chapter contains the explanation of utilities that are offered by Rake to work with the files: `FileList` and `FileUtils`. At the end, you will be given a real-world example on how to apply the acquired knowledge.

Chapter 3, *Working with Rules*, will show how knowing a `rule` task may allow you to write more robust and precise code.

Chapter 4, *Cleaning Up a Build*, describes one of the useful features of Rake—the capability to clean the build of your project with the `clean` standard task.

Chapter 5, *Running Tasks in Parallel*, helps us figure out how to speed up the resulting task execution with `multitask`. We will learn which basic problems may arise while implementing parallelism and how to avoid them.

Chapter 6, *Debugging Rake Tasks*, provides the basic knowledge to debug Rake projects. You will be provided with an example on how to debug rake tasks inherent to Rake techniques and also to Ruby projects in general.

Chapter 7, *Integration with Rails*, provides an overview of how Rake is integrated into the famous Ruby web framework, Rails. The chapter shows how to write custom rake tasks in a Rails project and run them manually or automatically on schedule.

Chapter 8, *Testing Rake Tasks*, details the reasons we should test rake tasks. Also, you will see an example of how to write the tests with MiniTest—a built-in Ruby test framework.

Chapter 9, *Continuous Integration*, briefly introduces you to Jenkins—a continuous integration software. You will see how to configure it and run rake tasks with its help.

Chapter 10, *Relentless Automation*, doesn't introduce any new Rake terms, but you will find useful examples of the Rake appliance by popular programs. You will be introduced to the Thor utility, which can replace Rake in some circumstances. Then we will compare both of these frameworks. Finally, we will briefly gather all the information that was provided throughout the book.

What you need for this book

To run the examples in this book, you must have Ruby installed. The examples can be run in all operation systems where Ruby can be installed. However, a few chapters provide examples that may be run only on Unix-based systems such as Linux and OS X. The command-line examples are written in a Unix-like style, but Windows users will also be able to run them.

Who this book is for

This book requires basic knowledge of Ruby because Rake is written in this programming language. But it doesn't mean that gurus of other languages will not be able to understand the examples. If you are working with a build automation tool that doesn't fit your requirements or seems too complicated, this book is what you need.

Conventions

In this book, you will find a number of styles of text that distinguish between different kinds of information. Here are some examples of these styles, and an explanation of their meaning.

Code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles are shown as follows: "We can include other contexts through the use of the `include` directive."

A block of code is set as follows:

```
task :hello do
  puts 'Hello, Rake!'
end
```

Any command-line input is written as follows:

```
$ rake task2
```

All command-line outputs have been highlighted and will appear as follows:

```
rake aborted!
this is an error
```

New terms and **important words** are shown in bold. Words that you see on the screen, in menus or dialog boxes for example, appear in the text like this: "You will be redirected to the configuration page of the created project. There you will find the **Build** section with the **Add build step** dropdown."

[Warnings or important notes appear in a box like this.]

[Tips and tricks appear like this.]

Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this book—what you liked or may have disliked. Reader feedback is important for us to develop titles that you really get the most out of.

To send us general feedback, simply send an e-mail to `feedback@packtpub.com`, and mention the book title via the subject of your message.

If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, see our author guide on `www.packtpub.com/authors`.

Customer support

Now that you are the proud owner of a Packt book, we have a number of things to help you to get the most from your purchase.

Downloading the example code

You can download the example code files for all Packt books you have purchased from your account at `http://www.packtpub.com`. If you purchased this book elsewhere, you can visit `http://www.packtpub.com/support` and register to have the files e-mailed directly to you.

Errata

Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you find a mistake in one of our books—maybe a mistake in the text or the code—we would be grateful if you would report this to us. By doing so, you can save other readers from frustration and help us improve subsequent versions of this book. If you find any errata, please report them by visiting `http://www.packtpub.com/submit-errata`, selecting your book, clicking on the **errata submission form** link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata will be uploaded on our website, or added to any list of existing errata, under the Errata section of that title. Any existing errata can be viewed by selecting your title from `http://www.packtpub.com/support`.

Piracy

Piracy of copyright material on the Internet is an ongoing problem across all media. At Packt, we take the protection of our copyright and licenses very seriously. If you come across any illegal copies of our works, in any form, on the Internet, please provide us with the location address or website name immediately so that we can pursue a remedy.

Please contact us at `copyright@packtpub.com` with a link to the suspected pirated material.

We appreciate your help in protecting our authors, and our ability to bring you valuable content.

Questions

You can contact us at `questions@packtpub.com` if you are having a problem with any aspect of the book, and we will do our best to address it.

1
The Software Task Management Tool – Rake

In this chapter, we will cover the installation of Rake, the definition of basic terms such as **rake task** and **Rakefile**, and how to use them for easy programming issues. The introduction will be given using straightforward examples to explain the terms as clearly as possible. You will see that Rake is a tool that is written in the Ruby programming language, and that's why any Ruby code can be written in a Rake application. Also, you have the choice of using any available Ruby library in a Rake project. This feature makes Rake the winner compared to many other build tools, which use their own limited languages. The chapter will serve as a base for introducing Rake's **Domain Specific Language** (**DSL**) and project file structuring.

In this chapter, we will cover the following topics:

- Installing Rake
- Introducing rake tasks
- The command-line arguments
- Using global Rakefiles to run tasks anywhere
- Defining custom rake tasks
- The structure of a Rake project
- The code conventions of Rake

Installing Rake

As Rake is a Ruby library, you should first install Ruby on the system if you don't have it installed already. The installation process is different for each operating system. However, we will see the installation example only for the Debian operating system family.

Just open the terminal and write the following installation command:

```
$ sudo apt-get install ruby
```

> If you have an operating system that doesn't contain the `apt-get` utility and if you have problems with the Ruby installation, please refer to the official instructions at `https://www.ruby-lang.org/en/installation`. There are a lot of ways to install Ruby, so please choose your operating system from the list on this page and select your desired installation method.

Rake is included in the Ruby core as Ruby 1.9, so you don't have to install it as a separate gem. However, if you still use Ruby 1.8 or an older version, you will have to install Rake as a gem. Use the following command to install the gem:

```
$ gem install rake
```

> The Ruby release cycle is slower than that of Rake and sometimes, you need to install it as a gem to work around some special issues. So you can still install Rake as a gem and in some cases, this is a requirement even for Ruby Version 1.9 and higher.

To check if you have installed it correctly, open your terminal and type the following command:

```
$ rake --version
```

This should return the installed Rake version.

The next sign that Rake is installed and is working correctly is an error that you see after typing the `rake` command in the terminal:

```
$ mkdir ~/test-rake
$ cd ~/test-rake
$ rake
rake aborted!
```

```
No Rakefile found (looking for: rakefile, Rakefile, rakefile.rb,
Rakefile.rb)
(See full trace by running task with --trace)
```

Downloading the example code
You can download the example code files for all Packt books you have purchased from your account at `http://www.packtpub.com`. If you purchased this book elsewhere, you can visit `http://www.packtpub.com/support` and register to have the files e-mailed directly to you.

Introducing rake tasks

From the previous error message, it's clear that first you need to have `Rakefile`. As you can see, there are four variants of its name: `rakefile`, `Rakefile`, `rakefile.rb`, and `Rakefile.rb`. The most popularly used variant is `Rakefile`. Rails also uses it. However, you can choose any variant for your project. There is no convention that prohibits the user from using any of the four suggested variants.

`Rakefile` is a file that is required for any Rake-based project. Apart from the fact that its content usually contains DSL, it's also a general Ruby file. Also, you can write any Ruby code in it. Perform the following steps to get started:

1. Let's create a `Rakefile` in the current folder, which will just say `Hello Rake`, using the following commands:

   ```
   $ echo "puts 'Hello Rake'" > Rakefile
   $ cat Rakefile
   puts 'Hello Rake'
   ```

   Here, the first line creates a `Rakefile` with the content, `puts 'Hello Rake'`, and the second line just shows us its content to make sure that we've done everything correctly.

2. Now, run `rake` as we tried it before, using the following command:

   ```
   $ rake
   Hello Rake
   rake aborted!
   Don't know how to build task 'default'
   (See full trace by running task with --trace)
   ```

   The message has changed and it says `Hello Rake`. Then, it gets aborted because of another error message. At this moment, we have made the first step in learning Rake.

3. Now, we have to define a default rake task that will be executed when you try to start Rake without any arguments. To do so, open your editor and change the created Rakefile with the following content:

   ```
   task :default do
     puts 'Hello Rake'
   end
   ```

4. Now, run `rake` again:

   ```
   $ rake
   ```

   ```
   Hello, Rake
   ```

The output that says `Hello, Rake` demonstrates that the task works correctly.

The command-line arguments

The most commonly used rake command-line argument is `-T`. It shows us a list of available rake tasks that you have already defined.

We have defined the `default` rake task, and if we try to show the list of all rake tasks, it should be there. However, take a look at what happens in real life using the following command:

```
$ rake -T
```

The list is empty. Why? The answer lies within Rake. Run the `rake` command with the `-h` option to get the whole list of arguments. Pay attention to the description of the `-T` option, as shown in the following command-line output:

```
-T, --tasks [PATTERN] Display the tasks (matching optional PATTERN) with
descriptions, then exit.
```

[You can get more information on Rake in the repository at the following GitHub link at `https://github.com/jimweirich/rake`.]

The word `description` is the cornerstone here. It's a new term that we should know. Additionally, there is also an optional description to name a rake task. However, it's recommended that you define it because you won't see the list of all the defined rake tasks that we've already seen. It will be inconvenient for you to read your `Rakefile` every time you try to run some rake task. Just accept it as a rule: always leave a description for the defined rake tasks.

Now, add a description to your rake tasks with the `desc` method call, as shown in the following lines of code:

```
desc "Says 'Hello, Rake'"
task :default do
  puts 'Hello, Rake.'
end
```

As you see, it's rather easy. Run the `rake -T` command again and you will see an output as shown:

```
$ rake -T
rake default  # Says 'Hello, Rake'
```

If you want to list all the tasks even if they don't have descriptions, you can pass an `-A` option with the `-T` option to the `rake` command. The resulting command will look like this: `rake -T -A`.

Using global Rakefiles to run tasks anywhere

By default, Rake is looking for tasks that are placed in the current folder (that is, the folder where you run the `rake` command) in the `Rakefile`. Assume that we need to have a rake task that can be executed in any folder. For example, say that we have a rake task that cleans the Linux files ending with `~`. The following Rakefile defines the rake task to remove them:

```
desc 'Cleans backup files *~'
task :default do
  files = Dir['*~']
  rm(files)
end
```

Here, we get temporary files in the current folder and remove them with the `rm` method. This method is defined in the `FileUtils` module, which is included in Rake as well. So, we will discuss it in the next chapters.

When you are in the current folder, check this rake task using the `Rakefile`:

```
$ rake
rm
```

Here, we see that the `rm` command was executed and Rake explicitly said this in the second line. If you don't want to see this verbose message, pass the `-q` option to the command.

However, what would happen if we go to the folder one level up? When you try to type the `rake` command, you will have an error message that says that no `Rakefile` was found. We can get rid of this problem by passing the `-f` option with the path to the `Rakefile` as shown in the following lines of code:

```
$ rake -f ~/my-rake-task/Rakefile
rm
```

This works well, but you may agree with me that it's too unhandy. Rake produces one useful feature to make this situation work the way we want. It's based on the method of finding the `Rakefile`. First, Rake tries to find the `Rakefile` in the current folder. If Rake can't find it there, the search continues till it reaches the user's home folder. If there is no `Rakefile` there, it finally raises an exception saying that the `Rakefile` was not found. We can apply this behavior to our issue. Just move the `Rakefile` to your home folder and mark the rake tasks defined in it as available for the current user everywhere. Open the terminal and type the following commands to achieve the expected output:

```
$ mv ~/my-rake-task/Rakefile ~/
$ cd ~/my-rake-task
$ rake
(in /Users/andrey)
rm
```

As you can see, this works as expected, and there is one more new line, as follows:

```
(in /Users/andrey)
```

This command says that the `Rakefile` was found at the user home folder. You can disable showing this information by passing the `-s` option.

There is another way to define global `Rakefiles`. You have an option to define them in the `~/.rake` folder, and they can be executed from any folder with the help of the `-g` option. The following is the Rake output of the `help` command:

```
-g, --system Using system wide (global) rakefiles (usually '~/.rake/*.
rake').
```

So, let's define a global `Rakefile` in this way and check it in action. The following is an example of how to do it through the terminal:

```
$ mkdir ~/.rake
$ touch ~/.rake/hello.rake
$ echo -e 'task "hello" do\n  puts "Hello, Rake"\nend' > ~/.rake/hello.
rake
$ rake -g hello
Hello, Rake
```

Defining custom rake tasks

So far, we defined only one task named `default`. Rake allows you to define your custom tasks with any name. The common form of the custom rake task definition is passing a task name to the `task` method and a block as a second argument. The block defines some action and usually contains some Ruby code. The rake task might have an optional description, which is defined with the `desc` method. This method accepts a text for the description of the task. The following code snippet is an example of defining a custom rake task:

```
desc 'Restart web server'
task :restart do
  touch '~/restart.txt'
end
```

This is an example of a possible rake task to restart **Passenger** (this is a module for the Nginx web server, which works with the Rails applications). We name the task `restart`. To run this task, just pass its name as the second argument to the `rake` command as shown in the following line of code:

```
$ rake restart
```

If you have a lot of tasks, it's handy to enclose them to the named spaces, as shown in the following code snippet:

```
namespace :server do
  desc 'Restart web server'
  task :restart do
    touch './tmp/restart.txt'
  end
end
```

You can also run the task in the command line using the following command:

```
$ rake server:restart
```

Actually, the task method accepts more arguments. However, they are related to other topics that are explained further along in the book in *Chapter 2*, *Working with Files*, and *Chapter 3*, *Working with Rules*.

Task dependencies – prerequisites

Sometimes, you have to write tasks that depend on other tasks. For example, when I'm going to seed data in my project, I want to clean all the persisting data that can break my code. In this case, we can say that our seed data task depends on the clean seed data task. The following code example shows us a `Rakefile` for this case:

```
task :clean do
  puts 'Cleaning data...'
end

task :seed => :clean do
  puts 'Seeding data...'
end
```

The preceding code executes the `clean do` task before running the `seed` task. The result of the execution of this task is shown below the following line of code:

```
$ rake seed
Cleaning data...
Seeding data...
```

It works as expected.

If you have to run the task from another namespace, pass its whole name as a string, as shown in the following code snippet:

```
namespace :db do
  task :clean do
    puts 'Cleaning data...'
  end
end

task :seed => 'db:clean' do
  puts 'Seeding data...'
end
```

However, if the dependent task is in the same namespace, you don't have to pass it as a string, as shown in the following code snippet:

```
namespace :db do
  task :clean do
    puts 'Cleaning data...'
  end

  task :seed => :clean do
    puts 'Seeding data...'
  end
end
```

Earlier in this chapter, we defined the `default` rake task. To be honest, we did it just to understand what happens on running `rake` without arguments and to introduce Rake in a few steps giving as less information as possible in an interactive way. However, in the practical word, nobody defines the `default` rake task with an action. Setting dependencies is a convenient feature. It allows the `default` task to refer to some other task as many times as you want without regression. For example, today, the `default` task runs a `doc:generate` task but tomorrow, we decide to run a `test:run` task instead. In such a situation, we can just change the prerequisite and that's it. So, always define your `default` rake task with the following template:

```
task :default => :some_task
```

It's also possible to pass many prerequisites for a task. The following line of code is an example of how to do this:

```
task :task1 => [:task2, :task3]
```

Multiple tasks definitions

A task might be specified more than once. Each specification adds its dependencies and implementation to the existing definition. This allows one part of a `Rakefile` to specify the actions and a different `Rakefile` (perhaps a separately generated one) to specify the dependencies.

For example, take a look a `Rakefile` that contains the following code:

```
task :name => [:prereq1, :prereq2] do
  # action
end
```

It can be rewritten as the following code:

```
task :name
task :name => [:prereq1]
task :name => [:prereq2]
task :name do
  # action
end
```

Passing arguments to the tasks

Assume that you have a rake task that sets the title for our blog and you want to pass it from the command line; this should be optional. If you don't pass the title of the blog, the default title should be set.

We have two solutions to solve this problem. The first solution is to pass parameters through the environment variable that is passed into the `ENV` variable in Ruby code (`ENV` is a hash-like accessor for environment variables, and it is available in any Ruby program). The second solution is using the built-in Rake syntax—you just pass variables to each task through square braces. The first use case doesn't allow you to pass variables for each task in isolation. The variables are shared among all the tasks in the `Rakefile`. So, the preferable style is the second choice. However, we are shown two alternatives, which will be discussed in the next sections.

The first alternative

The first alternative is a case where we pass variables using environment variables. The following code represents a `Rakefile`:

```
task :set_title do
  title = ENV['TITLE'] || 'Blog'
  puts "Setting the title: #{title}"
end
```

The following code is a usage example:

```
$ rake set_title TITLE='My Blog'
Setting the title: My Blog
$ rake set_title # default title should be set in this case
Setting the title: Blog
```

In the preceding example, the `ENV` variable approach can be used without any caution. The following code snippet represents the collision in sharing the variable between the tasks. Check the following `Rakefile`:

```
task :task1 do
```

```
  puts "#{ENV['TITLE']} in task1"
end

task :task2 do
  puts "#{ENV['TITLE']} in task2"
end
```

The following code is an example of usage:

```
$ rake task1 task2 TITLE='test'
test in task1
test in task2
```

You can see that the `TITLE` variable is accessible in both the tasks and is the same. Sometimes, you don't want to get this behavior and you need to pass the variables to each task individually.

A variable declared within a rake command will not persist in the environment. The following terminal output will confirm this statement:

```
$ export TITLE='Default Title'
$ rake set_title TITLE='My Blog'
Setting the title: My Blog
$ echo $TITLE
Default Title
```

The second variant

The second variant has a built-in Rake feature. The following is the `Rakefile` code:

```
task :set_title, [:title] do |t, args|
  args.with_defaults(:title => 'Blog')
  puts "Setting title: #{args.title}"
end
```

Please ignore the `t` variable at this moment; you will see what it means and what its usages are in *Chapter 2, Working with Files*.

Look at `args`, which is a hash-like object of the `Rake::TasksArguments` class. It has a useful method that is used here, named `with_defaults`, to merge the given arguments from the command line and the default values. If you don't pass the variables through the command line, the default variable for the title will be set.

The following code depicts how it may be used:

```
$ rake "set_title[My Blog]"
Setting title: My Blog
$ rake set_title
Setting title: Blog
```

Here, to pass the argument as a string with space (`My Blog`), I have enclosed the rake task with the argument within quotes. It's not the only case where I have to enclose the task name within double quotes. There are some terminals that don't understand the squared parentheses in the command line and should escape them with \ at the end of the code line of the rake task that is enclosed within the double quotes.

You are also able to pass multiple arguments to the rake task by separating them with a comma, as shown in the following line of command:

```
$ rake "name[Andrey,Koleshko]"
```

The task declaration for the preceding task is as follows:

```
task :name, [:first_name, :last_name] do |t, args|
  puts "First name is #{args.first_name}"
  puts "Last  name is #{args.last_name}"
end
```

Finally, you are able to pass variable-length parameters to the task with a comma, as we did in the previous example. In this case, you may use the `extras` method on the given `args` variable:

```
task :email, [:message] do |t, args|
  puts "Message: #{args.message}"
  puts "Recipients: #{args.extras}"
  puts "All variables: #{args.to_a}"
end
```

In the following example, the first argument will be assigned to the `message` variable on the `args` variable and the remaining arguments will go to the `extras` method. If you want to have an array that passes all the variables including the one associated with the `message` variable, you can call the `to_a` method on the `args` variable, as demonstrated in the preceding `Rakefile`.

```
$ rake "email[Hello Rake, ka8725@gmail.com, test@example.com]"
Message: Hello Rake
Recipients: ["ka8725@gmail.com", "test@example.com"]
All variables: ["Hello Rake", "ka8725@gmail.com", "test@example.com"]
```

The structure of a Rake project

Apart from the necessary Rakefile, there is a technique that allows us to form a good structure of a Rake project. Say that you have a very complicated Rake project with a lot of tasks. It's a good idea to split them into separate files and include them in the Rakefile. Fortunately, Rake already has this feature and you shouldn't care about implementing this feature from the scratch. Just place your separated files to the `rakelib` folder (it can be changed to custom by passing the `-R` option), give these files a `.rake` extension, and that's it. You don't have to do anything additional. Files with the `*.rake` extensions are included in the `Rakefile` automatically for you. Nonstandard extension such as `.rake` for the files should not scare you. These are the usual Ruby files. There you can write any Ruby code, define their rake tasks, include the related libraries, and so on. So, take this feature as a good thing to refactor a Rake project.

To approve the things said in this section, please open the terminal and check the following example:

```
$ mkdir rakelib
$ cat > rakelib/clean.rake
task :clean do
  puts 'Cleaning...'
end
^D
$ cat > Rakefile
task :default => :clean
^D
$ rake
Cleaning...
```

In this example, `^D` is a keyboard shortcut: *Ctrl* + *D*. The `cat` utility writes the standard output to the files here.

Using the import method to load other Rakefiles

It's possible to include other Ruby files or `Rakefiles` to describe a current `Rakefile`. It can be achieved by a standard `require` Ruby statement. However, what do we do when the including files depend on some method or variable defined in the describing `Rakefile`? To demonstrate the situation, create the following two files in a folder:

- `rakefile`
- `dep.rb`

The `rakefile` has some tasks definition, a method definition, and a `require` statement, as shown in the following code snippet:

```
require './dep.rb'

def method_from_rakefile
  puts 'it is a rakefile method'
end

task :a do
  puts 'task a'
end

task :b do
  puts 'task b'
end
```

The `dep.rb` file just defines a new task that has both the prerequisites tasks, `a` and `b`. Also, it calls the defined method, `method_from_rakefile()`, for some reason, as shown in the following code snippet:

```
method_from_rakefile()

task :c => [:a, :b] do
  puts 'task c'
end
```

Trying to run a rake task defined in `Rakefile` will cause an exception that says that there is no defined `method_from_rakefile` while the `dep.rb` file is loading:

```
$ rake c
rake aborted!
undefined method `method_from_rakefile' for main:Object
~/dep.rb:1:in `<top (required)>'
~/rakefile:1:in `<top (required)>'
(See full trace by running task with --trace)
```

The exception occurs when the `dep.rb` file is required by the `Rakefile`. The problem here is caused because the required file loaded even before the `Rakefile` could load. One of the possible solutions here is just to move the `require` statement to the last line of the `Rakefile`. As a result, the method and tasks required for the `dep.rb` file will be defined at the time of the `dep.rb` file being included in the `Rakefile`. To be honest, the solution seems like a hack; this is the Rake way.

Fortunately, Rake provides us with a tool to resolve this issue—the `import` method. It does what we really want here; the `import` statement may be used in any line of the `Rakefile`, and this doesn't apply to the loading process at all. The imported files will be loaded after the whole `Rakefile` is loaded. Its usage looks similar to the `require` statement and is shown in the following line of code:

```
import(filenames)
```

Here, you are able to pass more than one file.

There is one more feature of the `import` method. If you pass the filenames to the `import` task, they are evaluated first, and this allows us to generate the dependent files on the fly. Look at the following `Rakefile`:

```
task 'dep.rb' do
  sh %Q{echo "puts 'Hello, from the dep.rb'" > dep.rb}
end

task :hello => 'dep.rb'

import 'dep.rb'
```

This example generates the `dep.rb` file on the `file` due to the `import 'dep.rb'` call that evaluates the `'dep.rb'` task. The result of the `hello` task execution is shown as follows:

```
$ rake hello
echo "puts 'Hello, from the dep.rb'" > dep.rb
Hello, from the dep.rb
```

It is a really helpful feature that can not only help you in writing the Rake project, but also in a simple Ruby project.

Running rake tasks from other tasks

Sometimes, you will have to execute some defined task from your task manually. For this purpose, you have two methods of the `Rake::Task` class: `execute` and `invoke`. The difference between the two methods is that `execute` doesn't call dependent tasks, but the `invoke` method does. Both of these methods also accept arguments that can be passed to the tasks if you need them. Their usage is the same and is shown as follows. The following is the first code:

```
Rake::Task['hello'].invoke
```

The following is the second code:

```
Rake::Task['hello'].execute
```

With the `Rake::Task['hello']` code, we got the `hello` rake task. It returns an instance of the `Rake::Task` class and then, we are able to run any method on this. In the preceding examples, we called `invoke` and `execute`.

To get the namespaced task by name, like in the previous example, use a syntax or similar to the following line of code:

```
Rake::Task['my:hello']
```

One more difference between these methods is that the `invoke` method can't be executed twice without some trick. If you need to run the task more than once with the `invoke` method, use the `reenable` method as shown in the following code snippet:

```
Rake::Task['hello'].invoke
Rake::Task['hello'].reenable
Rake::Task['hello'].invoke
```

These capabilities can be used when you need to run some other rake task after a current task has been executed. Look at the following example that depicts how to use it in task actions. It demonstrates the usage of the `invoke` and `reenable` methods:

```
task :clean do
  puts 'cleaning data...'
end

task :process do
  puts 'processing some data...'
  Rake::Task['clean'].invoke
end

task :process_with_double_clean do
  puts 'processing some data...'
  Rake::Task['clean'].invoke
  Rake::Task['clean'].invoke
end

task :process_with_double_clean_and_reenable do
  puts 'processing some data...'

  Rake::Task['clean'].invoke
  Rake::Task['clean'].reenable
  Rake::Task['clean'].invoke
end
```

Try to paste this code in a `Rakefile` and run the `process`, `process_with_double_clean`, and `process_with_double_clean_and_reenable` tasks to find the difference between them. The following code is the output of the executions:

```
$  rake -f rakefile22 process
processing some data...
cleaning data...
$  rake -f rakefile22 process_with_double_clean
processing some data...
cleaning data...
$  rake -f rakefile22 process_with_double_clean_and_reenable
processing some data...
cleaning data...
cleaning data...
```

The code conventions of Rake

The words `namespace`, `desc`, `task`, `touch`, and so on in the `Rakefile` are general methods and, of course, you are able to pass parentheses when you pass the parameters there, as shown in the following code snippet:

```
namespace(:server) do
  desc('Restart web server')
  task(:restart) do
    touch('./tmp/restart.txt')
  end
end
```

However, the code looks quite ugly now, so it's recommended that you avoid using styles such as the one used here. Rake has its own DSL, and if you follow it, the code will be more readable.

The `namespace` and `task` methods are the basic methods that accept blocks that make the Rake code very expressive. For the `task` method, the block in the task definitions is optional, similar to what we saw in the *Task dependencies – prerequisites* section.

The blocks can be specified with either a `do/end` pair or with curly braces in Ruby. To specify a `Rakefile`, it's strongly recommended that you define rake tasks only with `do/end`. Because the `Rakefile` idiom tends to leave off parentheses on the tasks definitions, unusual ambiguities can arise when using curly braces. Take a look at the following proposed `Rakefile`:

```
def dependent_tasks
  [:task2, :task3]
```

```
end

task :task2 do
  puts 'In task2...'
end

task :task3 do
  puts 'In task3...'
end

task :task1 => dependent_tasks {
  puts 'In task1...' # We are expecting this code to be run but it's
not
}
```

The following is the result of the execution of `task1`:

```
$ rake task1
In task2...
In task3...
```

The defined *action* in `task1` is not evaluated. It leads to unexpected behavior. Because curly braces have a higher precedence than `do/end`, the block is associated with the `dependent_tasks` method rather than the `task` method.

A variant of passing the block after the dependent task name is not valid Ruby code at all, as shown:

```
require 'rake'
task :task1 => :task2 { }
```

It might seem strange but unfortunately, this code doesn't work and gives a syntax error as shown:

```
# => SyntaxError: syntax error, unexpected '{', expecting end-of-input
```

The conclusion of this is that if you just follow the `Rakefile` convention, you won't have problems with Rake's unexpected behavior.

Finally, the last tip for `Rakefiles` description: don't use the new style of a hash definition in the task prerequisites (in other words, don't describe tasks dependencies like this: `task1: :task2`). Often, only one prerequisite, defined at the first instance, transforms to the list of prerequisites and then you will have to translate the hash definition to the old style (in other words, the `task1: :task2` code transforms to `:task1 => [:task2, task3]`). Usually, all the task definitions contain the hash rocket instead of the colon notation. The conclusion here is simple: use the old style of the creation of Ruby hashes in the rake tasks definitions.

Summary

In this chapter, you learned what Rake is, what you have to do to start using Rake, how to use the `rake` command-line tool, how to write and run rake tasks, how to set their dependencies, and how to structure the code. There was some advice about Rake's DSL and an explanation on why should you follow it. This chapter doesn't demonstrate any real examples of how to use Rake because this knowledge is not enough to work with the files, but please be patient because you will see them in the upcoming chapters.

The next chapter will explain the basics to work with files using Rake. Also, you will see the first real example of put the knowledge of both these chapters to practice.

2
Working with Files

Rake is a tool that is intended to work primarily with files, and it actually has the best instruments to do this. In this chapter, we will see what Rake provides us with so we can work with files. We will look at utilities in order to work with them and at the end of the chapter, you will see how this information can be applied in practice.

In this chapter, we will discuss the following topics:

- Using file tasks to work with files
- The characteristics of the file task dependencies
- Creating a folder with the directory method
- Using Rake's file utilities
- A practical example of automatically generating a config file

Using file tasks to work with files

Often, you have to transform files from one type to another using a utility. For example, compiling source code to byte code in a language such as C or Java, or converting PNG images to JPG, and so on. For these challenges, Rake has many useful arms in its arsenal.

Assume that we have a Ruby project and it has a YAML-generated config file ending with `.yaml` and, for some reason, we have decided to rename it so that it ends with `.yml`. This process might have to be repeated very often as the file is generated by a third-party tool. Hence, we have to automate this process. We could do it manually with the following command:

```
$ mv settings.yaml settings.yml
```

The Rake produces a special type of task for cases like this. This is the **file task**. To define a file task, use a `file` method. The usage of this task is similar to general tasks. Honestly, it inherits all the general task behaviors. In a file task, we can set the prerequisites and write a task action or set description and so on. See a possible solution to the task of moving the `settings.yaml` file using the file task in the following example. This is the `Rakefile` with the file task:

```
file 'settings.yml' => 'settings.yaml' do
  mv 'settings.yaml', 'settings.yml'
end
```

With this code, we have defined the `settings.yml` file task that depends on the `settings.yaml` file. This means that if there is no `settings.yaml` file, then the file task will fail with the corresponding message. Try this file task in action using the terminal as shown in the following lines:

```
$ echo '' > settings.yml
$ rake settings.yml
mv settings.yaml settings.yml

$ rake settings.yml
rake aborted!
Don't know how to build task 'settings.yml'

Tasks: TOP => settings.yml
(See full trace by running task with --trace)
```

As you can see, the generated `settings.yaml` file was moved to `settings.yml` and the second attempt to run the command was not successful. The `settings.yml` file task depends on the `settings.yaml` file, but we don't have it at the moment. The resolution is exactly the same as that of general rake tasks.

In addition to general task behavior, the file task has a very useful feature. If source files (prerequisites) are not changed, the second attempt won't execute the file task action at all. The file task handles timestamp changes for source files, and if they are not touched, Rake decides to not run this file task. Try this with the example of copying files rather than moving them in the previous example. Create this `Rakefile` to check the feature out:

```
file 'settings.yml' => 'settings.yaml' do
  cp 'settings.yaml', 'settings.yml'
end
```

Now, create an empty `settings.yaml` file and make sure that there is no `settings.yml` file in the folder with `Rakefile`. Try to run it a few times and see that Rake hasn't changed the timestamps of the new `settings.yml` file. The following is the output of the terminal for the investigation:

```
$ rake -f rakefile01 settings.yml
cp settings.yaml settings.yml
$ stat -f "%m%t%Sm %N" settings.yml
1395252779   Mar 19 21:12:59 2014 settings.yml
$ rake -f rakefile01 settings.yml
$ stat -f "%m%t%Sm %N" settings.yml
1395252779   Mar 19 21:12:59 2014 settings.yml
```

The `mv` method of the previous example and the `cp` method are methods of the `FileUtils` module that will be introduced later in this chapter. For now, just know that they can be called in the rake tasks as we saw in the examples and they do what the corresponding commands in the terminal do.

The characteristics of the file task dependencies

In real life, we encounter more complicated examples like we have just seen. Often, a project build contains a lot of transformations of many files.

Consider that you have a blog and it contains a lot of articles. To build the site, you have to translate each chapter from the Markdown format to HTML. This will allow you to publish the blog on the Internet.

Markdown is a plain text formatting syntax designed so that it can be optionally converted to HTML using a tool by the same name. More about this can be found at `http://en.wikipedia.org/wiki/Markdown`.

To convert articles from Markdown to HTML, we will use the `pandoc` (`http://johnmacfarlane.net/pandoc/`) command-line utility. If you use Linux, your system's package manager may already contain it, and the installation is very easy. Use the following command to install `pandoc`:

```
$ apt-get install pandoc
```

If you want to know more about the utility, or if you have some other operating system and have installation troubles, please refer to the official website. It's located at `http://johnmacfarlane.net/pandoc`.

The following is a `Rakefile` with some simplification:

```
task :default => 'blog.html'

file 'article1.html' => 'article1.md' do
  sh 'pandoc -s article1.md -o article1.html'
end

file 'article2.html' => 'article2.md' do
  sh 'pandoc -s article2.md -o article2.html'
end

file 'blog.html' => ['article1.html', 'article2.html'] do
  File.open('blog.html', 'w') do |f|
    html = <<-EOS
                <!DOCTYPE html>
                <html>
                <head>
                  <title>Rake essential</title>
                </head>
                <body>
                  <a href='article1.html'>Article 1</a> <br />
                  <a href='article2.html'>Article 2</a>
                </body>
                </html>
              EOS
    f.write(html)
  end
end
```

This example contains code that might be unfamiliar for you. This is the creation of a multiline string to build the HTML: `<< -EOS … EOS`. It's called **heredoc**. More about this can be found at `http://en.wikipedia.org/wiki/Here_document`.

On the first line, we defined the `default` task and it is linked to the task that will generate the blog. It's already familiar to you. The following two file tasks define tasks to generate HTML articles for each one. The last file task, `book.html`, looks messy for now. We said that it depends on two file tasks: `article1.html` and `article2.html`. This means that when Rake tries to process the `blog.html` task, it will try to run both these tasks sequentially, and if they are done, the code in the blog will be executed. There is nothing difficult here; we just created the `book.html` page with links to the articles mentioned.

> To run this example, you have to have the `pandoc` utility installed. Create the proposed `Rakefile` in the folder with these two files: `article1.md` and `article2.md`. These files might contain any text or might just be blank. It doesn't matter. The command to run is `rake` or `rake blog.html`.

This `Rakefile` has some disadvantages and we think that you can see them. Firstly, if we add new posts in our blog, we will have to change this code. It would be better to not change `Rakefile` and have Rake find all chapters itself. The second thing that might be refactored is inserting links to the resulting `blog.html`.

The following is an example to simplify this:

```
task :default => 'blog.html'

articles = ['article1', 'article2']

articles.each do |article|
  file "#{article}.html" => "#{article}.md" do
    sh "pandoc -s #{article}.md -o #{article}.html"
  end
end

file 'blog.html' => articles.map { |a| "#{a}.html" } do
  File.open('blog.html', 'w') do |f|
    article_links = articles.map do |article|
                      <<-EOS
                        <a href='#{article}.html'>
                          Article #{article.match(/\d+$/)}
                        </a>
                      EOS
                    end
```

```
      html = <<-EOS
                  <!DOCTYPE html>
                  <html>
                  <head>
                    <title>Rake essential</title>
                  </head>
                  <body>
                    #{article_links.join('<br />')}
                  </body>
                  </html>
                EOS
      f.write(html)
    end
  end
```

This code is more universal now. If you write a new chapter, you won't have to change a lot of code. All you will have to do is add the name of the file without the extension to the `articles` array, and it will add it to the generated book. However, there is still a problem with generating the `blog.html` file: even if you add a new filename article to the array, it won't generate a new file. Here, Rake is powerless in detecting when to generate a new file (remember that Rake only takes care of the file and won't run it if the related files' timestamps are not changed). The only solution here is deleting the `blog.html` file every time you run Rake. The `FileUtils.rm` method will help us with this problem. There is one more useful feature of the `file` method that might be useful in refactoring the `blog.html` task. It yields one optional argument—the task object. It contains information about the task name, related tasks, and so on. Look at this `Rakefile` that demonstrates this with the following code:

```
require_relative 'blog_generator'

articles = ['article1', 'article2']

task :default => 'blog.html'

articles.each do |article|
  file "#{article}.html" => "#{article}.md" do
    sh "pandoc -s #{article}.md -o #{article}.html"
  end
end

FileUtils.rm('blog.html', force: true)

file 'blog.html' => articles.map { |a| "#{a}.html" } do |t|
  BlogGenerator.new(t).perform
end
```

Now look at the new code. The first change is the new line, `FileUtils.rm(blog.html, force: true)` that removes the `blog.html` file each time Rake is run. The `:force` option tells the `rm` method to not raise an exception if there is no such file. We may have this situation, for example, on the first run or if we have deleted the `blog.html` file ourselves. The next change consists of getting the `t` parameter in the blog to define the `blog.html` file task. It is then used to get the task name for the generating file and to get dependent tasks with the `prerequisites` method. It returns a string array with names of the dependent tasks.

Dive into the content of the file task, `blog.html`—it doesn't require any external information. This allows us to easily move the code to a separate file. Let's do this in the next step of refactoring. Create the `BlogGenerator` class; its `initializer` accepts the task and has one method, `perform`, which generates the site. Move it to a separate file and insert it into `Rakefile` with the `require_relative` method.

So, the `blog_generator.rb` looks like the following code:

```
class BlogGenerator
  def initialize(task)
    @task = task
  end

  def perform
    article_links = @task.prerequisites.map do |article|
                      <<-EOS
                        <a href='#{article}'>
                          Article #{article.match(/\d+/)}
                        </a>
                      EOS
                    end
    html = <<-EOS
             <!DOCTYPE html>
             <html>
             <head>
               <title>Rake essential</title>
             </head>
             <body>
               #{article_links.join('<br />')}
             </body>
             </html>
           EOS

    File.write(@task.name, html)
  end
end
```

The given `Rakefile` looks like the following code:

```
require_relative 'blog_generator'

articles = ['article1', 'article2']

task :default => 'blog.html'

articles.each do |article|
  file "#{article}.html" => "#{article}.md" do
    sh "pandoc -s #{article}.md -o #{article}.html"
  end
end

FileUtils.rm('blog.html', force: true)

file 'blog.html' => articles.map { |a| "#{a}.html" } do |t|
  BlogGenerator.new(t).perform
end
```

This code is much more flexible now and looks better. However, we still have to change the `Rakefile` code to inform the **blog generator** to generate a new article if we add one. In *Chapter 3*, *Working with Rules*, you will learn how to get rid of this issue.

Creating a folder with the directory method

Sometimes, you will have to create folders with nesting. You can create files and folders with file tasks. If you need to create a folder tree, you can achieve this with the file tasks' definitions and their dependencies. The following is an example of this usage:

```
file 'my_gem' do |t| mkdir t.name end
file 'my_gem/tests" => ['my_gem'] do |t| mkdir t.name end
file 'my_gem/tests/fixtures" => ['my_gem/tests/fixtures'] do |t|
  mkdir t.name
end
```

When you try to execute the `my_gem/tests/fixtures` task, it will first call the dependent `my_gem/tests` task and then the call will be passed to the `my_gem` task. The tasks create folders with their name. Finally, we will have created a ready-to-use folder path, `my_gem/tests/fixtures`.

Another way is to use the `FileUtils#mkdir_p` method, which might be used in the task action or just in the `Rakefile` context. However, this is not the Rake way. There is a special way to define folder tasks in the Rake language: using the `directory` method. The following is an example of how we could use it in the previous code snippet:

```
directory 'my_gem/tests/fixtures'
```

That's it. This is a like a synonym for `FileUtils#mkdir_p`, but it also defines the Rake task that could be used in the prerequisites of other tasks:

```
directory 'my_gem/tests/fixtures'

file 'README.md' => 'my_gem/tests/fixtures' do
  sh 'echo test > my_gem/tests/fixtures README.md'
end
```

Now, when you call the `README.md` task, it will try to call the `my_gem/tests/fixtures` task to create the required folder for the file.

The `directory` method doesn't accept any arguments except the name of the folder to be created. However, if you need to add prerequisites or actions for the directory task, you can achieve this with the `file` method, as shown in the following code snippet:

```
directory 'my_gem'
file 'my_gem' => ['otherdata']
file 'my_gem' do
  cp Dir['gem_template/**/*'], 'my_gem'
end
```

Using Rake's file utilities

Rake provides us with a few helpful modules and methods. They may be very useful in some cases, especially while working with files. Knowing them might help you keep `Rakefiles` clean and precise. The following is a list of these features:

- The `FileList` module
- The `FileUtils` module
- The `pathmap` method

The next three sections will explore them in detail.

Using the FileList module functionality to collect the files

There is only one thing remaining in our `Rakefile` that would be great to get rid of. Currently, there is a need to change the list of articles manually. Luckily, Rake provides us with a tool to solve this problem—`Rake::FileList`. It provides agile instruments to tune your own list of files to be generated. It is flexible enough to filter the list of files by the category of your choice. It also enables you to filter out temp files that are generated by your editor, folders that have to be ignored, and files with some features that can be detected dynamically (timestamps are included files to control the version system for example). Now, take a look at how it can be used for our `Rakefile`:

```
require_relative 'blog_generator'

articles = Rake::FileList.new('**/*.md', '**/*.markdown') do |files|
             files.exclude('~*')
             files.exclude(/^temp.+\//)
             files.exclude do |file|
               File.zero?(file)
             end
           end

task :default => 'blog.html'

articles.ext.each do |article|
  file "#{article}.html" => "#{article}.md" do
    sh "pandoc -s #{article}.md -o #{article}.html"
  end

  file "#{article}.html" => "#{article}.markdown" do
    sh "pandoc -s #{article}.markdown -o #{article}.html"
  end
end

FileUtils.rm('blog.html', force: true)

file 'blog.html' => articles.ext('.html') do |t|
  BlogGenerator.new(t).perform
end
```

Take a look at how we defined the `articles` list. It was created by the `Rake::FileList` class. Its initializer accepted the list of file masks; we got all the files with the `.md` and `.markdown` extensions in any folder; then, the list got filtered out by the `~*` pattern. (This is a list of temp files that are generated by the **Emacs** editor.) Then, we ignored the files in the folders that start with the word `temp` (see the `/^temp.+\//` regular expression). The last filter demonstrates how to filter out files with zero size that should be ignored by us too. Pay attention to the useful method named `.ext`, which is used here to construct the list of filenames. We will demonstrate it separately:

```
$ irb -r rake --prompt=simple
>> articles = Rake::FileList.new('**/*.md', '**/*.markdown')
=> ["article1.md", "article2.md", "article3.md"]
>> articles.ext
=> ["article1", "article2", "article3"]
>> articles.ext('.html')
=> ["article1.html", "article2.html", "article3.html"]
```

Look at how the `irb` tool is run. We passed the argument `-r rake` to automatically require the `rake` library to the Ruby shell and `--prompt=simple` to get a simplified output of evaluating the code. Without this option, the Ruby shell will output unnecessary information such as the command number, the current context, and line numbers of the typed code. Because you are using this option the verbosity won't distract you from the important text. For a detailed explanation of the IRB utility and its command-line arguments, refer to the official documentation at `http://ruby-doc.org/stdlib-2.1.0/libdoc/irb/rdoc/IRB.html`.

Using pathmap to transform file lists

Working with list files and folders, we often have to transform a batch of files from one type to another. We have already seen the `.ext` method in action. However, in some cases, this won't be enough. Rake provides one more interesting method for this purpose—`pathmap`. It might be called on the `FileList` object or on any string because Rake extends the `String` class. It accepts two arguments: the required specification and an optional block. The `#pathmap` method collects the path according to the given specification. The specification controls the details of the mapping. The following special patterns are recognized:

- `%p`: This is the complete path.
- `%f`: This is the base filename of the path with its file extension, but without any directories.
- `%n`: This is the filename of the path without its file extension.

- `%d`: This is the directory list of the path.
- `%x`: This is the file extension of the path. It is an empty string if there is no extension.
- `%X`: This includes everything but the file extension.
- `%s`: This is the alternate file separator if defined; otherwise, use the standard file separator.
- `%%`: This is a percent sign.

Assume that we have the following list of files and directories: `file1.txt`, `file2.pdf`, `sources/file3.txt`, and `bin/file4`. To demonstrate how all these specifications work, we prefer to provide the results in a table. This table will contain the results of calling `#pathmap` on a given file list for each specification. The testing code is as follows:

```
require 'rake'
list = FileList['file1.txt', 'file2.pdf', 'sources/file3.txt', 'bin/
file4']
list.pathmap('%p')
list.pathmap('%f')
list.pathmap('%n')
list.pathmap('%d')
list.pathmap('%x')
list.pathmap('%X')
list.pathmap('%s')
list.pathmap('%%')
```

You can construct the `FileList` object not only with the `.new` method that we used before, but also with the `.[]` method that is used in the previous code.

The result of the execution of each call to `#pathmap` is provided in the following table:

Specification	**Result**
`%p`	`["file1.txt", "file2.pdf", "sources/file3.txt", "bin/file4"]`
`%f`	`["file1.txt", "file2.pdf", "file3.txt", "file4"]`
`%n`	`["file1", "file2", "file3", "file4"]`
`%d`	`[".", ".", "sources", "bin"]`
`%x`	`[".txt", ".pdf", ".txt", ""]`

Specification	Result
%X	["file1", "file2", "sources/file3", "bin/file4"]
%s	["/", "/", "/", "/"]
%%	["%", "%", "%", "%"]

It is good to be aware of what each specification produces, but let's see how we could use it in practice. Say, we are going to transform all the given files in the list to the HTML format. Using the `#pathmap` method, we can achieve this quite easily with `list.pathmap('%X.html')`, and the resulting list will be `["file1.html", "file2.html", "sources/file3.html", "bin/file4.html"]`.

In the next example, you are going to move all the files from the list to a folder named `output`. So, it will be fine to get the list of all files to the folder using the following command:

```
>> list.pathmap('output/%f')
=> ["output/file1.txt", "output/file2.pdf", "output/file3.txt",
"output/file4"]
```

We can use the following command if the resulting file list needs to have the same extension:

```
>> list.pathmap('output/%X.html')
=> ["output/file1.html", "output/file2.html", "output/sources/file3.
html", "output/bin/file4.html"]
```

You also can combine the specifications using the following code:

```
>> list.pathmap('output%s%n%s%f')
=> ["output/file1/file1.txt", "output/file2/file2.pdf", "output/file3/
file3.txt", "output/file4/file4"]
```

The `%d` specifier can also have a numeric prefix (for example, `%2d`). If the number is positive, it only returns (up to) `n` directories in the path, starting from the left-hand side. If the value of `n` is negative, it returns (up to) `|n|` directories from the right-hand side of the path, as follows:

```
>> 'a/b/c/d/file.txt'.pathmap("%2d")
=> 'a/b'
>> 'a/b/c/d/file.txt'.pathmap("%-2d")
=> 'c/d'
```

You have to generate a list of arguments for the command line from the given file list once. For example, it's no secret that to run Ruby scripts, you should include some libraries before running it, as shown in the following line of code. In particular, if you are running a unit test, you have to do it. In other cases, the test will fail:

```
$ ruby -Ilib/my_class -Ilib/common test/my_class_test.rb
```

With `#pathmap`, you are ready to construct this very easily:

```
require 'rake'
list = FileList['lib/my_class', 'lib/common']
ruby "#{list.pathmap('-I%p')} test/my_class_test.rb"
```

We are using a new method here, named `ruby`. It's defined in the `FileUtils` module. It takes arguments to pass to the command line and executes the `ruby` command with them. `FileList` defines the `to_s` method that is used in the string interpolation. As you can see, its behavior is different from the implementation of `Array`. It just joins all the items in the path with spaces between them. If you want to get the string that is returned by `Array#to_s` from `FileList`, you have to explicitly convert `FileList` to `Array`:

```
require 'rake'
list = FileList['lib/my_class', 'lib/common']

list.to_s      # => "lib/my_class lib/common"
list.to_a.to_s # => "[\"lib/my_class\", \"lib/common\"]"

list.pathmap('-I%p').to_s      # => "-Ilib/my_class -Ilib/common"
list.pathmap('-I%p').to_a.to_s # => "[\"-Ilib/my_class\", \"-Ilib/
common\"]"
```

There is one more useful feature of the specifications to translate the `sources` folder to the `output` folder. The `%d`, `%p`, `%f`, `%n`, `%x`, and `%X` operators can take a pattern/replacement argument to perform simple string substitutions on a particular part of the path. The pattern and replacement should be separated by a comma and should be enclosed within curly braces. The replacement should be after the `%` character but before the operator letter, for example, `%{in,out}d`. Multiple replacements can also be defined by separating them with semicolons, for example, `%{in,out;old,new}d`. Assimilate the information in the following example, which could be applied in real life:

```
>> "app/assets/js/app.coffee".pathmap("%{^app/assets/js,public}X.js")
=> "public/app.js"
```

Remember that `%X` gives us the file path without an extension and it won't be difficult to understand substituting the leading `app/assets/js` folder in the `public` folder, and the ending extension `.coffee` to the `.js` with this specification: `%{^app/assets/js,public}X.js`.

As you can see, the **regular expressions** may be used in patterns of the specifications (^ is an element of the regular expression, which means a beginning of the line). The replacement text might contain backreferences of the pattern:

```
>> "app/assets/js/app.coffee".pathmap("%{^app/assets/(js),public/new-\\1}
X.js")
=> "public/new-js/app.js"
```

[You can find full information about backreferences and usage examples at `http://www.regular-expressions.info/backref.html`.]

However, pay attention to the fact that the power of regular expressions is restricted here. Curly braces, commas, and semicolons are excluded from both the pattern and replacement text.

If you find `#pathname` and `#ext` methods useful and would like to use them in your project without using all the Rake features, you can do so with the following single line of code:

```
require 'rake/ext/string'
```

Introducing the FileUtils module

We've already mentioned this module in the book. It contains a lot of useful methods to work with files. So, if you have some task that requires you to work with files, just remember that there is a library like this one, and if you look through this module, you might find a suitable method for your issue. You have a good chance of avoiding reinventing the wheel. All operations with files and directories (copying, removing, moving, creating, changing permissions, linking, and so on) exist in this module.

Check out the entire documentation of each method of this module at `http://goo.gl/ec4arH`.

A practical example of automatically generating a config file

Now that you have some knowledge, we would like to show you how Rake can be used in practice. Every Rails developer knows that the first step they have to take when they have a new project is to create the `config/database.yml` file. It's a rather boring process and includes a lot of manual processes. If there is a `config/database.yml.template` file, you are lucky and you have to just copy it to `config/database.yml`. However, if this template file is not in the project, you will have to copy it from another project or find it on the Internet, in the documentation, or somewhere else. The next step consists of changing the configuration itself. As usual, it includes changing the username, password, adapter, and database. Also, you have to change these variables for each environment. It's a boring process, isn't it? So, we decided that if we create a task to automate this process, it will be useful. We hope your Rake arsenal is complete and prepared to attack this code:

```
require 'yaml'

desc 'Generates database.yml, optional arguments: [adapter, user,
password]'
task :dbconfig => 'database.yml'

file 'database.yml', [:adapter, :username, :password] do |t, args|
  Dir.chdir('config')
  args.with_defaults(:project_path => Dir.pwd)
  DBConfigGenerator.new(t, args).generate
end

class DBConfigGenerator
  ENVIRONMENTS = %w(production development test)
  DEFAULTS = {
    'adapter' => 'postgresql',
    'encoding' => 'unicode',
    'username' => Etc.getlogin,
    'pool' => 5,
    'password' => nil
  }

  def initialize(task, options = {})
    @database_pattern = "#{options[:project_path].pathmap('%-1d')}_%s"
    @template = {}
    @output_file = task.name
    @defaults = DEFAULTS.tap do |defaults|
                  defaults.each_key do |k|
                    defaults[k] = options[k] if options[k]
                  end
                end
  end

  def generate
    ENVIRONMENTS.each do |env|
      @template[env] = @defaults.merge('database' => @database_pattern
% env)
    end

    File.write(@output_file, @template.to_yaml)
  end
end
```

Put this in the `~/.rake/copy_db_config.rb` file and you will be able to execute this task everywhere with this command: `rake -g dbconfig`. As you have already guessed, we created the **global rake task** here.

Summary

This chapter covered the basic features of Rake for working with files. Now, you should be able to use Rake to solve any file problems that you might encounter in your daily work. At the end of the chapter, we saw an example that used this knowledge.

However, one more problem that may seem inconvenient to you is its code duplication in the file tasks' definitions. In the next chapter, you will see how to improve the code of this chapter with rules. You will be introduced to rules and see how to use them in practice.

3
Working with Rules

In the previous chapter, we saw how to work with files. This knowledge will be enough for you to manage any task with files that you might have. However, sometimes, the code will look messy and you will find it inconvenient. So, the goal of this chapter is to improve the code and get rid of duplication in a task's actions. The chapter covers a technique to get rid of duplication of code in a task's body with the help of rules. The **rules** allow us to specify the templates with which we can catch a lot of task names and evaluate corresponding actions. This feature might help in many situations while writing a Rake project.

In this chapter, we will cover the following topics:

- Understanding the duplication of the file tasks
- Using a rule to get rid of the duplicated file tasks
- Detecting a source for the rule dynamically
- Using a regular expression to match more tasks

Understanding the duplication of the file tasks

In the previous chapter, we successfully created the blog builder. Let's revise the `Rakefile` that we've written to do it:

```
require_relative 'blog_generator'

articles = Rake::FileList.new('**/*.md',
                   '**/*.markdown') do |files|
              files.exclude('~*')
              files.exclude(/^temp.+\//)
              files.exclude do |file|
              File.zero?(file)
            end
          end

task :default => 'blog.html'

articles.ext.each do |article|
  file "#{article}.html" => "#{article}.md" do
    sh "pandoc -s #{article}.md -o #{article}.html"
  end

  file "#{article}.html" => "#{article}.markdown" do
    sh "pandoc -s #{article}.markdown -o #{article}.html"
  end
end

FileUtils.rm('blog.html', force: true)

file 'blog.html' => articles.ext('.html') do |t|
  BlogGenerator.new(t).perform
end
```

The following problems arise with the highlighted chunk of code:

- First of all, it contains the duplication of the task definitions.
- Secondly, we should iterate through all the articles to define all of these tasks that lead to a lot of task definitions. This is not a good thing.

In object-oriented programming, we always try to keep a class' interface narrow, because a bloated interface leads to lot of mess. When you define a new rake task, you define new functions, and that's what extends the interface of `Rakefile`. When you have a large number of tasks, the complexity of managing them will bother you. Moreover, when you define a rake task, a new instance of the `Rake::Task` class occupies the memory (for the file task, the instance of `Rake::FileTask` will be allocated). It would be great to have an opportunity to define patterns for both of these tasks and combine them in to one. In this chapter, we will try to refactor and improve this code to solve the problems stated previously with one more useful feature of Rake called **rule**.

Using a rule to get rid of the duplicated file tasks

To get rid of duplicated file tasks, there is a special rake task called `rule`. This is a general rake task, but it has one peculiarity. It allows us to define a mask for a task rather than the exact name. We will postpone the whole explanation of this, and for now, we will only see how to do this with the `rule` method. The following is our fixed `Rakefile` with a `rule` method:

```
require_relative 'blog_generator'

articles = Rake::FileList.new('**/*.md',
                 '**/*.markdown') do |files|
             files.exclude('~*')
             files.exclude(/^temp.+\//)
             files.exclude do |file|
               File.zero?(file)
             end
           end
task :default => 'blog.html'

rule '.html' => '.md' do |t|
  sh "pandoc -s #{t.source} -o #{t.name}"
end

rule '.html' => '.markdown' do |t|
  sh "pandoc -s #{t.source} -o #{t.name}"
end

FileUtils.rm('blog.html', force: true)

file 'blog.html' => articles.ext('.html') do |t|
  BlogGenerator.new(t).perform
end
```

There is a `default` rake task that depends on the `blog.html` task. Also, `blog.html` depends on the list of `*.html` file tasks. This list is an extension replacement of all the markdown files to the `.html` extension.

In the previous example, we achieved some results: we removed the iteration from all the tasks manually to define the two tasks for each article to translate the Markdown file to the HTML file. As a result, we won't have a lot of tasks in the feature. As you might have guessed, the signature of the `rule` method is the same for a task or file. However, there is one big difference: you pass a *pattern* to define dependencies and a task name rather than *string values*. The rules from these rake tasks will catch all the tasks that end with `.html`. The first rule defines that it depends on files ending with `.md`, while the second rule matches files that end with `.markdown`. For example, if we try to run the `task.html` task, the rules will be associated with it and they will be dependent on `task.md` (first rule) or `task.markdown` (second rule).

There is still one more improvement that we need to add in the code: we can replace the two task rules with one without loosing all the features. The content of both rules is the same, so it would be great to use only one rule for this purpose. Rake's documentation says that we can pass `proc` in place of a dependency definition.

In Ruby, `proc` is a type of closure or anonymous function. If you are not familiar with this, please refer to the Ruby documentation to learn more about it: `http://ruby-doc.org/core-2.1.0/Proc.html`.

The `proc` object yields one option, `task_name`, and then we are able to do everything we want with this to define the dynamic **prerequisites**. In other words, we should translate `task_name` to a demanding task that really exists. If a dependent task doesn't exist, an exception will be raised by Rake that there is no such task. For our case, possible values for `task_name` are the `articles` list with extensions replaced from `.md` or `.markdown` to `.html`. Remember how `rule` works to understand this: `blog.html` depends on the list of `*.html` tasks that are taken from the list of articles, which are in the Markdown format (`.md` or `.markdown`).

Detecting a source for the rule dynamically

Say, we have this list of articles in our blog: `article1.md`, `article2.md`, and `article3.markdown`. When we get the value of `task_name`, which is one among `article1.html`, `article2.html`, or `article3.html`, there is no information on what the source is. If the value of `task_name` is `article1.html`, what is the source? Is it `article1.md` or `article1.mardown`? To understand this, we should define it dynamically. The possible solution is to go through `articles` and see if there is a file named `article1.md` or `article1.mardown` there. This is a possibility with `proc` too. Describing this is rather complicated.

The following example might give you some clarity:

```
require_relative 'blog_generator'

articles = Rake::FileList.new('**/*.md',
                  '**/*.markdown') do |files|
             files.exclude('~*')
             files.exclude(/^temp.+\//)
             files.exclude do |file|
               File.zero?(file)
             end
           end

task :default => 'blog.html'

detect_file = proc do |task_name|
  articles.detect { |article| article.ext == task_name.ext }
end

rule '.html' => detect_file do |t|
  sh "pandoc -s #{t.source} -o #{t.name}"
end

FileUtils.rm('blog.html', force: true)

file 'blog.html' => articles.ext('.html') do |t|
  BlogGenerator.new(t).perform
end
```

Take a look at `detect_file`—it's the `proc` object that I just explained. It just iterates through all articles in the Markdown format and tries to detect the filenames without extensions. If it's detected, the real filename of the Markdown file will be returned. Finally, figuratively speaking, a rake task such as `article.html ⇒ article1.md` will be defined. (Actually, this task will not be defined. This is just an example of how the dependency is achieved and how the Rake will work when the `article1.md` file is persisted in the `blog` folder.)

Using a regular expression to match more tasks

It's possible to pass a regular expression as a rule pattern in the rule definition. The following example shows you how to do it:

```
rule /\.html$/ => '.md' do |t|
  sh "pandoc -s #{t.source} -o #{t.name}"
end
```

We used the new method in the preceding example, `source`, which is called on the task. Its name explicitly says what it does—it returns the name of the source. In our case, this is the Markdown file.

Summary

This chapter covered one of the painful themes in the programming world: how to refactor the repeated code. The chapter also explained how to achieve this with rules.

In the next chapter, you will see how to remove the files generated by the rake tasks using the standard features of Rake.

4
Cleaning Up a Build

Sometimes, you will have to clean the generated files with rake tasks to get into the initial state of the build for some reason. Maybe you would like to delete intermediate files while building or removing produced files to make sure that the build is clean and generates new files with confidence. In this situation, the built-in mechanisms of Rake could help you, and this chapter is about these mechanisms.

The solution is based on a simple idea, so the chapter includes the following topics:

- Setting up a project
- The cleaning tasks

Setting up a project

To write this book, I had been using the AsciiDoc format and the `git-scribe` tool to generate output formats such as PDF, MOBI, HTML, and so on. However, these tools bring some inconvenience with them. For example, to generate a PDF format, many temporary files are generated and left in the `output` folder. In this case, Rake brings special instruments out of the box, and this chapter is about their usage.

The AsciiDoc file format was designed especially to write books. You can find more information about this on the official page at `http://www.methods.co.nz/asciidoc`.

More information about the `git-scribe` tool can be found at `https://github.com/schacon/git-scribe`.

Firstly, it's recommended that you to try out this tool in action to understand what this chapter will explain further. So, please go to the main page of `git-scribe` and install this tool following the installation instructions in the `README` file. Then, generate the skeleton of the book with the following command:

```
$ git scribe init <directory name>
```

This is enough to generate the book into the proposed formats from the generated template. Currently, the tool may generate the HTML, PDF, EPUB, or MOBI formats. It also generates a site for the book from the template that can be changed manually for your purposes. The `git-scribe` tool is a kind of software that can be excellently solved with Rake. Unfortunately, it was written from scratch with pure Ruby, and that's why we can't extend this program with rake tasks by defining the task prerequisites or other Rake features. However, this would be the best choice to solve the problem.

So let's write a wrapper for the tool with rake tasks. The first change is to create a task to generate any format from the proposed formats using the `git-scribe gen` command (the `git-scribe gen pdf` command looks too long, so we will cut it down to two words: `rake pdf`). You can already imagine how to do this because this operation is rather simple. The second change is to create a task to clean the temporary files from the `output` folder.

Start and try the `git-scribe` tool in action. Firstly, let's bootstrap the application on which we have to perform the experiment:

1. Generate the book project with the following command:
   ```
   $ git scribe init the-book
   ```
2. Go to the generated template of the project. Use the following command to do this:
   ```
   $ cd the-book
   ```
3. Make sure that there is no `output` folder there yet.
4. Run the following command to generate the book in the PDF format:
   ```
   $ git scribe gen pdf
   ```

These commands should create the `output` folder with a lot of files with different types. The following is an output from my terminal (you should be getting this list as well):

```
$ cd ~/projects
$ git-scribe init the-book
initializing the-book
```

```
$ cd the-book/
$ ls -ln
total 16
-rw-r--r--  1 501  20  303 Dec 17 01:07 LICENSE
-rw-r--r--  1 501  20  300 Dec 17 01:07 README.asciidoc
drwxr-xr-x  6 501  20  204 Dec 17 01:07 book
$ git-scribe gen pdf
GENERATING PDF
GENERATING DOCBOOK

Making portrait pages on A4 paper (210mmx297mm)
Dec 17, 2013 1:16:59 AM org.apache.fop.events.LoggingEventListener
processEvent
INFO: Rendered page #1.
Dec 17, 2013 1:17:00 AM org.apache.fop.events.LoggingEventListener
processEvent
INFO: Rendered page #2.
$ ls -ln output
total 168
-rw-r--r--  1 501  20  48465 Dec 17 01:16 book.fo
-rw-r--r--  1 501  20  27736 Dec 17 01:17 book.pdf
-rw-r--r--  1 501  20   3016 Dec 17 01:16 book.xml
-rw-r--r--  1 501  20    153 Dec 17 01:16 chapter2.asc
drwxr-xr-x  4 501  20    136 Dec 17 01:16 image
drwxr-xr-x  3 501  20    102 Dec 17 01:16 include
drwxr-xr-x  4 501  20    136 Dec 17 01:16 stylesheets
```

Look at the result of the last command. The `output` folder contains a huge list of files that are not interesting to us. They are temporary, so we have to delete them somehow. In this case, the standard rake task that is named `clean` could help us. Further, we will write `Rakefile` to solve our tasks and you will see how to use it.

Start with the task wrapper to generate the book to these possible formats: HTML, PDF, EPUB, or MOBI. The following is an instance of `Rakefile` that is saved in the generated book project:

```
FORMATS = [:pdf, :html, :mobi, :epub]

FORMATS.each do |f|
  desc "Generate the book in '#{f}'"
```

```
    task f do |t|
      sh "git-scribe gen #{t.name}"
    end
  end
```

There is no new information for you here. Notice that we haven't used flexible tasks such as `file` and `rule`, which are more appropriate for tasks like this one because we can't completely manipulate the whole processing files including filenames. As the filename of the book is hardcoded in the `git-scribe` tool, we wouldn't benefit from their usage. Now we have crept closer to the clean tasks.

The cleaning tasks

At this stage, we need to be introduced to some terms that we should know in order to write clean tasks. To be able to define clean tasks, we have to include the cleaner with the following line of code:

```
require 'rake/clean'
```

This defines two constants, `CLEAN` and `CLOBBER`, and two tasks, `clean` and `clobber`.

- `CLEAN`: This is a list of files to be cleaned. The `clean` task goes though this list and removes them.
- `CLOBBER`: This is a list of generated files. These are files that are produced by rake tasks and they are usually the last files in the chain. The `clobber` task goes through this list and removes these files. This task has one prerequisite: it has to be a `clean` task.

The idea of using clean tasks is very simple. Just add the necessary files or folders to both these lists and run the appropriate tasks, as shown in the following code snippet (it's up to you to separate the files to `clean` or `clobber`). That's it, let's do this:

```
require 'rake/clean'

FORMATS = [:pdf, :html, :mobi, :epub]

FORMATS.each do |f|
  desc "Generate the book in '#{f}'"
  task f do |t|
    sh "git-scribe gen #{t.name}"
  end

  CLOBBER.include("output/*.#{f}")
```

```
    end

    CLEAN.include('output/*.asc')
    CLEAN.include('output/*.fo')
    CLEAN.include('output/*.xml')
    CLEAN.include('output/stylesheets/')
    CLEAN.include('output/include/')

    CLOBBER.include('output/image/')
```

Now it's time to check out the PDF task:

```
$ rake pdf
git-scribe gen pdf
GENERATING PDF
GENERATING DOCBOOK


Making portrait pages on A4 paper (210mmx297mm)
Dec 17, 2013 2:54:59 AM org.apache.fop.events.LoggingEventListener
processEvent
INFO: Rendered page #1.
Dec 17, 2013 2:55:00 AM org.apache.fop.events.LoggingEventListener
processEvent
INFO: Rendered page #2.
$ ls -ln output
total 168
-rw-r--r--  1 501  20  48465 Dec 17 02:54 book.fo
-rw-r--r--  1 501  20  27736 Dec 17 02:55 book.pdf
-rw-r--r--  1 501  20   3016 Dec 17 02:54 book.xml
-rw-r--r--  1 501  20    153 Dec 17 02:54 chapter2.asc
drwxr-xr-x  4 501  20    136 Dec 17 02:54 image
drwxr-xr-x  3 501  20    102 Dec 17 02:54 include
drwxr-xr-x  4 501  20    136 Dec 17 02:54 stylesheets
```

As you saw, the command generated not only the demanding PDF file, but also a lot of unnecessary files. Remove them with the `clobber` task, as shown in the following lines of code:

```
$ rake clobber
$ ls -ln output
```

The `clobber` task depends on the `clean` task that is called at first and deletes all files and folders except the `image` folder (it is required by the `html` format, and we should delete it in the `clobber` task). Then, the `clobber` task completely removes the entire content of the `output` folder. The last command approves that the `output` folder is clear after the `clobber` task is invoked.

Summary

This chapter described one more useful feature of Rake: the capability to clean the build of your project with the standard `clean` task that goes out of the box.

In the next chapter, we will see how Rake is good at multithreading and how to run rake tasks in parallel.

5 Running Tasks in Parallel

To speed up rake tasks, you could do a lot of things, starting with simple refactoring and ending with algorithm improvements. However, the easiest and the most efficient way is **parallelism**. It means that you invoke your chunks of code simultaneously in their own threads, if that's possible, instead of executing them consequently. This is why the finish time will be shorter in theory.

In this chapter, we will see how to use this feature and how Rake uses multithreading to run tasks in parallel. The chapter includes the following topics:

- Defining tasks with parallel prerequisites
- Thread safety of multitasks
- Multiple task definitions with a common prerequisite
- Applying multitasks in practice

Defining tasks with parallel prerequisites

Describing rake tasks that depend on other tasks that should be executed in parallel is actually a straightforward, solvable problem. Just define your tasks with a `multitask` method instead of the `task` method, as shown in the following code snippet, and that's it:

```
multitask :setup => [:install_ruby, :install_nginx, :install_rails] do
  puts "The build is completed"
end
```

In this example, the tasks `install_ruby`, `install_nginx`, and `install_rails` will be executed in parallel before the action of the `setup` task. This means that for each dependent task, a Ruby thread will be created and they will be run at the same time. The `setup` task will wait for the threads until they are finished.

Check out the following `Rakefile` that will be used to verify the previously mentioned statements:

```
task :task1 do
  puts 'Action of task 1'
end

task :task2 do
  puts 'Action of task 2'
end

multitask :task3 => [:task1, :task2] do
  puts 'Action of task 3'
end
```

Now, open the terminal and try to run the `task3` task a few times to see that the tasks are running in a random order, as shown in the following lines of command:

```
$ rake task3
Action of task 1
Action of task 2
Action of task 3
$ rake task3
Action of task 2
Action of task 1
Action of task 3
```

As you can see, the order of the execution of `task1` and `task2` is not predictable because they are run in the threads. The first time, `task1` is executed before `task2` and the second time, the order is the opposite. Pay attention to the fact that you may have other results, and to see different orders, you will have to run the `task3` task more times than we did here. When you have to deal with a thread, the order of their running is always unpredictable. This is why you mustn't rely on the order of the task execution in your rake tasks if you are going to run them in parallel.

For example, the following `Rakefile` is sensitive to the order of the execution of the dependent tasks:

```
task :set_a do
  @a = 2
end

task :set_b do
```

```
    @b = 3 + @a
  end

  multitask :sum => [:set_a, :set_b] do
    puts "@b = #{@b}"
  end
```

When you run the `sum` task, you get an exception and the task gets interrupted. For example, if the `set_b` task is run before the `set_a` task, the `@a` variable will not be initialized; it will contain `nil`, so the expression in the `set_b` task `@b = 3 + @a` will fail: `3 + nil` gives an exception in Ruby. You can observe it in the interactive Ruby shell; try this in the command line:

```
$ irb --simple-prompt
>> 3 + nil
TypeError: nil can't be coerced into Fixnum
```

So sometimes, everything will work (when the `set_a` task is run before `set_b`) without errors as we expect, but sometimes you will have an exception (when the `set_b` task is run before `set_a`). Try the following by yourself in the terminal on the `sum` rake task:

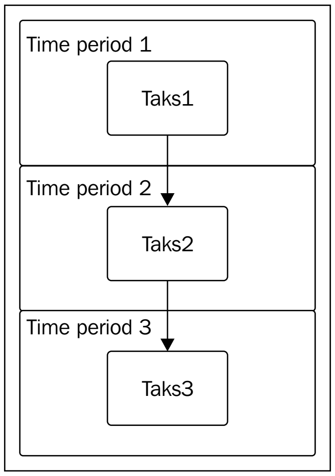
```
$ rake sum
rake aborted!
nil can't be coerced into Fixnum
...
$ rake sum
@b = 5
```

When we ran the task for the first time, it failed because the `set_b` task was executed before `task_a`. However, the second execution was successful because of the correct order. Note that to get both the results (fail and pass), you may need to run the task more than twice because of the unpredictability of the execution of the tasks.

In the following diagram, you may see how the tasks would be executed when defined with the `task` method. In this case, the working flow is sequential and the tasks are run one by one. The resulting time will be the sum of the time periods of the execution of each task. In this example, it will be the following:

Time period 1 + Time period 2 + Time period 3

The following is a figure that shows us the sequential task execution:

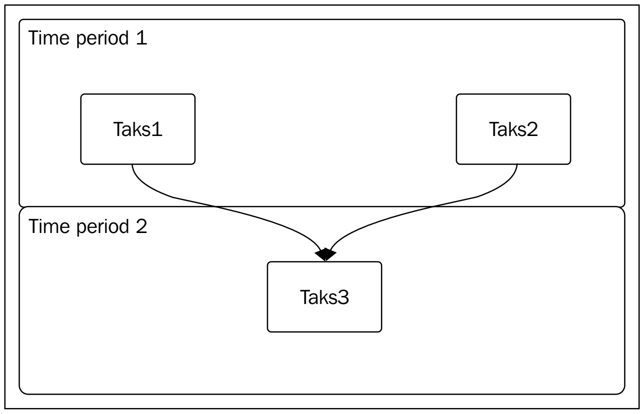


Now, in the next figure, you can see what is going on when we define the task dependencies with the `multitask` method. The `task1` and `task2` tasks are executed at the same time, so the resulting time will be lesser:

Time period 1 + Time period 2

Here, *Time period 1* is a period of time to execute both the tasks (`task1` and `task2`) in parallel.

The following is a figure that shows us the parallel task execution:



Thread safety of multitasks

Rake's internal data structures are thread-safe, so we don't have to do extra synchronization for the benefit of Rake. However, if we have shared variables or resources (for example, the database or files) and the parallel tasks are simultaneously performing operations under them, we must prevent **race conditions** with additional effort. Basically, this requires using additional tools to synchronize the data.

You've already seen the problem with the `@a` variable in the previous example. To get rid of the problem, we have to ask the `set_b` task to wait for the `set_a` task. However, as we don't have public access to their threads, we can't to do this, so they can't be executed in parallel. In the example, multitasking is redundant and a sequential execution will be more appropriate there; be careful when using multitasking because of this particular reason.

Unfortunately, this book is not intended to explain the multithreading theme and how it works in Ruby. To get more information, please refer to `http://goo.gl/GaeHgd`.

Multiple task definitions with a common prerequisite

Assume that we have rake tasks that have a common prerequisite and at the same time, these tasks are prerequisites for a multitask. Which order will we get for the task execution when we run the multitask? You may think that the multitask's prerequisites will be run in parallel and the common prerequisite will run as many times as the number of the multitask's prerequisites. However, this is not true; actually, the multitask's prerequisites will wait until the common prerequisite gets completed. As a result, the common prerequisite will be run only once.

The following example will demonstrate the idea clearly:

```
task :copy_src => [:prepare_for_copy] do
  puts 'In the #copy_src'
end

task :copy_bin => [:prepare_for_copy] do
  puts 'In the #copy_bin'
end

task :prepare_for_copy do
  puts 'In the #prepare_for_copy'
```

```
end

multitask :all_copy => [:copy_src, :copy_bin] do
  puts 'In the #all_copy'
end
```

We get the following output as a result of the task execution:

```
$ rake all_copy
In the #prepare_for_copy
In the #copy_src
In the #copy_bin
In the #all_copy
```

The `copy_src` and `copy_bin` tasks have a common prerequisite—the `prepare_for_copy` task. So, when the `all_copy` method of `multitask` tries to execute its prerequisites in parallel, the secondary common prerequisite `prepare_for_copy` will be run first. Then, the `copy_src` and `copy_bin` prerequisites will be run. Please note that despite the fact that the `prepare_for_copy` is occurring in two places as a prerequisite, it is run only once.

Applying multitasks in practice

Every Ruby developer knows about the bundler gem: the tool used to manage the gem dependencies in the projects. It is based on Rake's ideas and its usage is rather simple. You write to `Gemfile` the list of the gems that are required by the application and then just inform the bundler to install them with command prompt.

In the old versions, the bundler would download the gems from the Internet and sequentially install them one by one to a system. The main disadvantage of this process was that it was too long and would take a lot of time. The resulting time was dependent on the connection speed of the Internet, the number of dependent gems, and many other factors.

However, life goes ahead and the current bundler's release (Version 1.5.0) is able to install the gems in parallel. The resulting time of the installation process is decreased by a huge amount. This is a real-life example from where multitasks can be applied. With the new version of bundler, you can install gems in parallel:

```
$ bundle install -j 4
```

In this example, the number `4` is a number of threads to be spawned to install the gems.

If you are not familiar with the bundler and are ready to learn more about this tool, refer to the information provided at `http://bundler.io/`.

Summary

In this chapter, we figured out how to speed up the resulting time of the task execution with Rake multitasks. We learned about the basic problems that might arise when we use parallelism and how to avoid them. Whether it would be reasonable to use multitasking or not is up to you. It depends on the issue that is being resolved. Sometimes, it may speed up the execution process, but sometimes it may lead to unexpected behavior. However, if you choose to use multitasks, just be attentive to them.

The next chapter will share ideas on how to debug a Rake project.

6
Debugging Rake Tasks

Debugging is an unavoidable process when building applications, and this concerns Rake applications too. Rake provides a lot of techniques that will be helpful in many situations not only while writing, but also while using a Rake application. In this chapter, we will look into a number of debugging tools that are available out of the box. There will an example on using Ruby's `debugger` to debug a Rake code.

In this chapter, we will cover the following topics:

- Using command-line arguments for debugging
- Getting a dependency's resolution with `--prereqs`
- Using the `--rules` option to trace the rule resolution
- Using the Ruby approach to debug a Rake project

Using command-line arguments for debugging

The main information in the development process is a **backtrace**—a report of a certain point in time during the execution of a program. When a rake task fails, you won't need the whole backtrace; Rake narrows it in the default behavior. To explain the idea, see the following `Rakefile` with failed `task1`:

```
task :task1 do
  raise 'this is an error'
end

task :task2 => :task1 do
  puts 'task 2'
end
```

The following is a result of the `task2` execution:

```
$ rake task2
rake aborted!
this is an error
~/rakefile:2:in `block in <top (required)>'
Tasks: TOP => task2 => task1
(See full trace by running task with --trace)
```

Notice that the backtrace contains only one line of code (`~/rakefile:2:in 'block in <top (required)>'`). To see the full trace of the code execution, use the `--backtrace` option:

```
$ rake --backtrace task2
rake aborted!
this is an error
~/rakefile1:2:in `block in <top (required)>'
.../ruby/gems/2.1.0/gems/rake-10.1.1/lib/rake/task.rb:236:in `call'
.../ruby/gems/2.1.0/gems/rake-10.1.1/lib/rake/task.rb:236:in `block in
execute'
… # the following lines will contain a lot of lines, so they are omitted
here
Tasks: TOP => task2 => task1
```

If the execution doesn't fail, the result won't contain the backtrace. Note the presence of leading … in each line at the beginning of the backtrace. We cut them intentionally for better readability. However, you will have full paths for Ruby libraries if you run this example by yourself. So, just be aware that the destinations for the libraries (that is, the backtrace lines) differ and depend on how you have installed Ruby, the gems that are used in your project, and other files that are required to run the task.

If you want to see the order of the execution of the tasks, you can use a `--trace` option (also, there is a short form of this option, `-t`). By the way, this suggestion is from Rake for when the rake task fails (we mean the text `(See full trace by running task with –trace)`).

An example of its usage is as follows:

```
$ rake --trace task2
** Invoke task2 (first_time)
** Invoke task1 (first_time)
** Execute task1
```

```
task 1
** Execute task2
task 2
```

As you can see, the output contains rather useful information such as the order of the execution of tasks including prerequisites and the detailed outcome of each one. If the task raises an exception, the whole backtrace will be showed too, when executed with the `--backtrace` option.

The backtrace may contain a lot of lines that describe dependent libraries, including a Ruby core. Sometimes it may be rather difficult to find a bug in this enormous text. In this situation, a `--suppress-backtrace` option could help you filter out the undesired lines, provided there's a pattern by you. The following is a usage example:

```
$ rake --suppress-backtrace /ruby/2.1.0/ task2
```

Now the backtrace will not contain Ruby's internal calls, which are not of interest to us in general. The option is ignored with a `--trace` option passed at the same time.

Getting a dependency's resolution with --prereqs

Another useful option that only shows us the task dependency resolution and nothing more is `-P` or `--prereqs`. Unlike the `--trace` option, it doesn't execute the tasks, as shown in the following lines of command:

```
$ rake -P
rake task1
rake task2
    task1
```

The list contains the defined tasks and their dependencies with indentation. Take a look at the `task1` text that is shifted in the last line. It is written after `rake task2` and it means that `task2` is dependent on `task1`.

Using the --rules option to trace the rule resolution

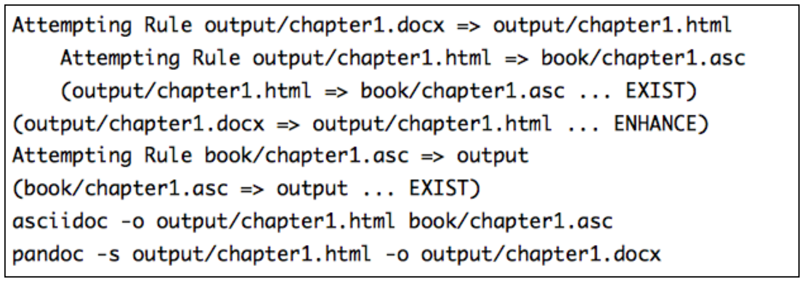
The relationship between tasks becomes the most difficult when the tasks are rules. In this case, a `rule` task may be suited for many task names. For example, when the rule specifies a regular expression instead of a name. In this case, the rules that were described previously won't help us anyway.

When you run a task that was accepted by a rule, it's very useful to know which task will be executed and in which order. Rake provides us with a `--rules` option to show us a rule's resolution. Take a look at the following `Rakefile` from *Chapter 3, Working with Rules*:

```
require 'rake/clean'

BOOK = 'book/book.asc'
CHAPTERS = FileList['book/*.asc'].exclude(BOOK)
DOCX_OUTPUT = 'output/%n.docx'
ODT_OUTPUT = 'output/%n.odt'

CLEAN.include('output/*.html')
CLOBBER.include('output')

namespace :generate do
  directory 'output'

  desc 'Generate only one article with given number'
  task :article, [:number] do |t, args|
    num = args.number
    article = CHAPTERS.detect { |ch| ch =~ /#{num}.asc$/ }
    Rake::Task[article.pathmap(DOCX_OUTPUT)].invoke
    Rake::Task[:clean].invoke
  end

  desc 'Generate articles one by one'
  task :articles => CHAPTERS.pathmap(DOCX_OUTPUT)
  task :articles => CHAPTERS.pathmap(ODT_OUTPUT)
  task :articles => [:clean]

  desc 'Generate the entire book'
  task :book => BOOK.pathmap(DOCX_OUTPUT)
  task :book => [:clean]

  file 'output/book.html' => ['book/book.asc'] do |t|
    sh "asciidoc -d book -o #{t.name} book/book.asc"
  end

  rule /\.docx|\.odt$/ => '.html' do |t|
    sh "pandoc -s #{t.source} -o #{t.name}"
  end

  rule '.html' => proc { |name| name.pathmap('book/%n.asc') } do |t|
    sh "asciidoc -o #{t.name} #{t.source}"
  end

  rule '.asc' => 'output'
end
```

It's complicated to demonstrate the output of the rule's resolution on the `generate:article[number]` task:

The following is the output of the `$ rake --rules generate:article[1]` command:



```
Attempting Rule output/chapter1.docx => output/chapter1.html
    Attempting Rule output/chapter1.html => book/chapter1.asc
    (output/chapter1.html => book/chapter1.asc ... EXIST)
(output/chapter1.docx => output/chapter1.html ... ENHANCE)
Attempting Rule book/chapter1.asc => output
(book/chapter1.asc => output ... EXIST)
asciidoc -o output/chapter1.html book/chapter1.asc
pandoc -s output/chapter1.html -o output/chapter1.docx
```

As you can see, the information contains a chain of rule executions. This information may be very useful when you have an unexpected behavior in the `rule` task's execution.

Using the Ruby approach to debug a Rake project

As you already know that a Rake project is a Ruby project; therefore, all manipulations that you may use with a Ruby project can be applied to Rake project. So, you are able to use general debuggers from the Ruby world.

Currently, there are two tools to debug the Ruby code: `debugger` and `pry`. While these tools may be used for the debugging process, they are slightly different type of tools: the `debugger` tool is specifically a debugger in many programming languages, but the `pry` tool is an improved console that is designed to work in multiple contexts, including the applications running in the background. We can use either as per our requirement. The idea is the same for both of these tools, and the following steps will be performed for each:

1. Install the tool.
2. Add a breakpoint.
3. Run a rake task. The code will be paused on the breakpoint.
4. Investigate the environment.

`debugger` and `pry` are third-party tools. You can find more information about them at `https://github.com/cldwalker/debugger` and `https://github.com/pry/pry`, respectively.

Before using a debugger, you need to install it. Here, you have two choices: using `bundler` or installing `gem` to the system or `gemset`. It depends on how you organize a project code, but if you are going to maintain the project, you will have to eventually use `bundler`.

We will demonstrate the usage of `debugger`. Say, we have the following `Rakefile`:

```
require 'debugger'

task :test do
  puts 'starting the test task'
  debugger
  puts 'ending the test task'
end
```

When you run the `test` task, the code will be stopped on the `debugger` line, shown as follows:

```
$ rake test
starting the test task
~/rakefile:6
puts 'ending the test task'


[1, 10] in ~/rakefile
   1  require 'debugger'
   2
   3  task :test do
   4    puts 'starting the test task'
   5    debugger
=> 6    puts 'ending the test task'
   7  end
(rdb:1)
```

Now you will be able to execute chunks of code here from the environment and investigate a problem, if you have one.

This code is tested by Ruby Version 2.1.0 or lower and it works correctly, but according to the `README` file for `debugger`, the tool does not fully support the current releases of Ruby. So, if you have troubles with debugging this code with the `debugger` tool, you can try to debug it using the `pry` utility.

There are a lot of available commands for the `debugger` tool. To get their full list, use the `help` command. Just type it in the current mode of the `debugger` tool and press the *Enter* key. If you want to know what a command does, type `help <command>`. For example, to see what the `continue` command does, type `help continue`. The most frequently used commands in the debug process are `next` or `continue`. The `next` command executes the current line and goes to the next one; the `continue` command runs the code until it meets the next breakpoint or runs the code until the end if there are no more breakpoints.

To describe how to use the full power of the debug tools, we will need to dive into a large amount of information. This topic is worth a separate book, but here we are limited to a brief introduction. If you want to learn more about the tool, please follow the official documentation at `https://github.com/cldwalker/debugger`.

Summary

This chapter provided us with a basic knowledge to debug Rake projects. We learned how to figure out the order of the tasks that will be executed. We briefly saw how to use the Ruby debug tool in the example of the `debugger` gem.

In the next chapter, you will be introduced to the theme of how Rake is integrated with the most famous Ruby framework, Rails.

7
Integration with Rails

This chapter explains how Rake is used by Rails. This is a framework that is used by a major bulk of Ruby developers to develop web applications.

You may ask why we are going to explain the Rails framework here. Well, this is a reasonable question. Firstly, because it will clarify one of Rake's primary activities—deploying a system to production; secondly, because of its popularity; and finally, because you don't need to have any special knowledge to understand the ideas that will be given further.

In this chapter, we will cover the following topics:

- Introducing Rake's integration with Rails
- Custom rake tasks in a Rails project
- Recurrent running of tasks

Introducing Rake's integration with Rails

Before starting the theme analysis, install the `Rails` gem and generate a Rails application, or you can just open a Rails project that is ready if you have it. The installation is a rather easy process:

```
$ gem install rails
$ rails g test_app
```

This command will generate a skeleton of the Rails application.

> If you face any issues during this process, please follow the official site at `http://rubyonrails.org`.

Now, go to the `test_app` folder; see if there is a `Rakefile` there. As you may have already guessed, this is the first sign that the application uses Rake. Although Rails is not a full Rake application, it uses it as an auxiliary tool. If you open `Rakefile`, you probably won't understand what is going on there. To be honest, you should not care about its content at all because this is an example of how a Rake project could be structured. The file contains only two lines of code! However, you may see that after using the `rake -T` command, the Rails application has a lot of tasks. This means that all the tasks are defined separately in the core of Rails. The authors of Rails have adopted Rake to their needs and made a lot of changes and customized their Rake project structure.

If you want to know how this works under the hood, you can open the Rails sources and go through the code and try to figure it out, but we omit this journey in this book because it's out of the scope of this book.

Another interesting thing apart from the Rake code structure is the feature that defines custom rake tasks. To define your own rake task, you can just place a `.rake` file in the `lib/tasks` folder. Remember that Rake loads the `.rake` files from the `rakelib` folder with default settings.

Rails comes with a helpful utility that may be used to generate may be used to generate a Rails component. The component may be migration, model, controller, or other class for a Rails application. It's interesting that there is a generator for rake tasks too. It means that you are able to create rake tasks for Rake manually or with the generator. Its usage is very easy and is shown as follows:

```
$ rails g task namespace task1 task2
```

The command will generate a file with the following content in the path `lib/tasks/namespace.rake`:

```
namespace :namespace do
  desc "TODO"
  task :task1 => :environment do
  end

  desc "TODO"
  task :task2 => :environment do
  end
end
```

More information about the Rails task generator can be found on the blog post at `http://goo.gl/l1XSqm`.

Note that all the generated tasks have one prerequisite—`environment`. This is a task that loads the Rails environment for the application and allows you to use defined classes in the task's actions. For example, if you omit the prerequisite and try to use the `User` action model that is defined at `app/models/user.rb` (`app/models` is the default folder of Rails where all the models for the application are defined), you will get an exception saying that no `User` class has been defined. So, just be attentive and don't panic when you have an error like this and stay confident that the code is written correctly.

Apart from writing custom rake tasks, Rails allows us to use core tasks. They are divided by the appliance; some of them are used to work with the database, the assets, the routes, and so on. We are not going to study them here because of their huge number. All this information can be found in the official documentation at `http://guides.rubyonrails.org/command_line.html#rake`.

This sums up the integration of the Rake in Rails.

Custom rake tasks in a Rails project

Now, let's figure out why the custom rake tasks are used in a Rails project. First of all, they are aimed to support tasks that make something recur with the project. For example, this can be generating a `sitemap.xml` file for the site, cleaning up old data, backing up the database, sending e-mails in the background, and similar tasks.

In the demonstration, the custom rake tasks in a Rake project have no special steps. Assume that we have `orders` in our system and we have to delete them if their status is `deleted`. There aren't any special steps to write a custom rake task to solve this problem. Just open the terminal and generate the following rake task:

```
$ rails g task orders cleanup
      create  lib/tasks/orders.rake
```

Now open the generated file and write the following code there:

```
namespace :orders do
  desc 'Remove old orders with the deleted status'
  task :cleanup => :environment do
    Order.where(:status => 'deleted').find_each(&:destroy)
  end
end
```

To run this rake task, just use the `rake` command as usual, shown as follows:

```
$ rake orders:cleanup
```

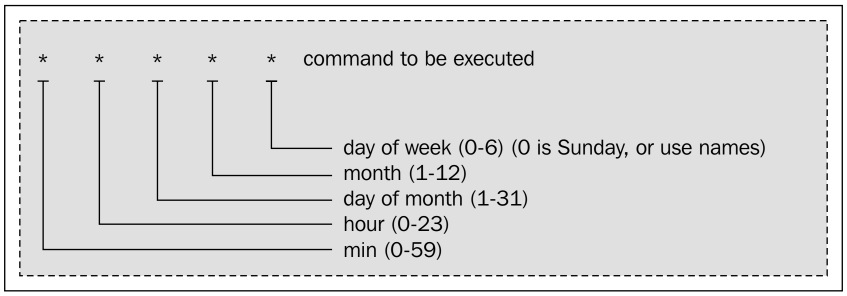
As you can see, it's that simple for a plain Rake project. However, there are some additional conveniences with the generator and the loading environment; you don't have to worry about the classes that are required to run the action, and you are not obligated to require them one by one.

Recurrent running of tasks

By now, everything is fine and there is nothing to disturb us; we have solved our issue, it's ready to use, and we can run it manually tomorrow, after tomorrow, and on any day in the future, or we can forgot about it completely. It would be great to run this task automatically each day or every week or even hourly. The main problem is how to run this *task on the schedule*. In Unix-like systems, there is a utility that enables these type of tasks—`cron`.

> To learn more about `cron`, use the `man cron` command on the Unix-like operating system; there is an excellent blog post about it at `http://goo.gl/XZ4GrC`. For people who use Windows, there is an analogue program at `http://cronw.sourceforge.net`. There is also a program included in Windows and you don't even have to install third-party tools for this at all. This tool is called Task Scheduler (`http://en.wikipedia.org/wiki/Task_Scheduler`).

We are not going to dive into the depth of `cron` now, but you need to know some basic information about how the utility works. There is a `config` file for `cron` that describes what to do and when and follows the following format:


* * * * * command to be executed
day of week (0-6) (0 is Sunday, or use names)
month (1-12)
day of month (1-31)
hour (0-23)
min (0-59)

For instance, assume that we have to write the current date to a file at 12:00 every day. To solve this issue we could use the `cron` tool and its config would look like the following:

```
0   12   *   *   *   /bin/date > ~/tmpfile
```

You don't have to change the configuration manually; you have to do it with the `crontab` utility that goes with `cron` out of the box on Unix-like systems. So, to configure and test the previous example, run the `crontab -e` command. An editor will open for you to write this line of code down there; the number line in which you will insert it is irrelevant. Save the editor and that's it; the job will be done at 12:00, so now you have to wait for this time or make some changes to your current time.

Okay, now you know how to do this on the server—just go to the server, run the `crontab -e` command, and paste the code to run the rake task. Brilliant, but it's not the end of the story; to write the line of code properly, you have to worry about where Rake is installed because `cron` requires the entire path to the set of programs where the project is installed as well as the task name. You will have to change these lines when some of this information is changed. There is a ready solution for these issues—the `whenever` gem.

[Check out the documentation on how to use it at `https://github.com/javan/whenever`.]

In our case, we have to install it according to the documentation and change the configuration file. The installation is very simple:

1. Just add a line of code to the Gemfile that is situated in the root folder of the project:

   ```
   gem 'whenever', :require => false
   ```

2. Run the `bundle install` command.
3. Then, run the `wheneverize` command. This command will generate a `config/schedule.rb` file.
4. Paste the following lines of code there:

   ```
   every :day, :at => '12:20pm' do
     rake 'orders:cleanup'
   end
   ```

5. Now, run the `whenever -w` command. It will set the properly generated `config` file to the `cron` config. Pay attention to the fact that it won't delete other `cron` jobs if you have some in `config`; it's very handy because the config change won't affect other systems that use the cron config either.

Summary

In this chapter, we gave you a brief overview on how the Rake is integrated to Rails and also on how to write the custom Rake tasks and run them manually or automatically on schedule.

The next chapter will provide you with ideas on how to test the rake tasks. There will be explanations on why we have to test the rake tasks at all. Finally, you will see an example of how to employ them using Ruby's internal test framework.

8
Testing Rake Tasks

In this chapter, you will find information on how to test the rake tasks. It is important to understand that if you don't perform the tests, your rake tasks may fail just like a usual program. This is why the chapter starts by providing cases when the tasks fail, along with the reasons for their failure. Here, we will provide you with an example on how to test rake tasks using Ruby's embedded unit test framework—**MiniTest**.

In this chapter, we will cover the following topics:

- The need for tests
- Writing tests for rake tasks

The need for tests

The rake tasks are not run as often as regular code from the application. Consider a basic situation when you have a web application with rake tasks that have not been tested. At first glance, after deploying the code to a server and manually testing the application through the web interface, you can be confident that the application works. However, because the rake tasks are usually run by a scheduler, it may be a time bomb. Finally, when the time comes to execute a rake task, it doesn't work because it wasn't tested! Such issues might often occur at the start of your career as a Ruby programmer.

For example, each Rails application has a `db:seed` rake task. This task is often used to initialize the application using some essential data. For example, it could contain the code to create an administrator in our application. The code is defined in the `db/seeds.rb` folder that is related to the root of the application. The following is a basic Ruby code to create an administrator in our application:

```
User.create!({
   :admin => 'example@email.com',
   :password => 'password'
})
```

Suppose one particular day we created this code and manually ran the rake task with the `rake db:seed` command. Several days passed and we decided to add validation to the user model for the *name presence*. The previous code won't work in this case, but as we don't test the `db:seed` tasks, all other tests are passed and we are ready to deploy the application because everybody has already forgotten about this piece of code and nobody knows that it doesn't work at all. The issue will be detected when the application is initially deployed to a production server. At that moment, we may have a lot of outdated tasks. The problem keeps cropping up again and again until we finally decide to write the tests for rake tasks. So, in order to avoid mistakes such as these, writing tests may provide us with an airbag.

Writing tests for rake tasks

Now, when you are prepared to write the tests, it's time to figure out how to do it. To demonstrate this, assume that we have to write a rake task named `send_email`, which has these optional arguments: `subject` and `body`. The task should write the given strings to a `sent_email.txt` file (this is for the purpose of simplicity to demonstrate the tests; in reality, you may want to use a real mailer).

Start to solve the problem statement. The following is a basic class `Mailer` that will be used in the rake task (place this code in the `mailer.rb` file):

```
class Mailer
  DESTINATION = 'sent_email.txt'.freeze
  DEFAULT_SUBJECT = 'Greeting!'
  DEFAULT_BODY = 'Hello!'

  def initialize(options = {})
    @subject = options[:subject] || DEFAULT_SUBJECT
    @body = options[:body] || DEFAULT_BODY
  end

  def send
```

```
      puts 'Sending email...'
      File.open(DESTINATION, 'w') do |f|
        f << "Subject: #{@subject}\n"
        f << @body
      end
      puts 'Done!'
    end
  end
```

The interface of the class is very simple. Its initializer accepts `subject` and `body` as parameters of `hash`, and then we are ready to use the `send` method on the object of this class to create the `sent_email.txt` file. This is a simulation to send e-mails for easier the demonstration of the tests.

The following is `Rakefile` (it's assumed that this file is in the folder where you created `mailer.rb`):

```
require 'rake/clean'
require_relative './mailer'

CLOBBER.include(Mailer::DESTINATION)

desc "Sending email. The email is saved to the file
#{Mailer::DESTINATION}"
task :send_email, :subject, :body do |t, args|
  Mailer.new({
    :subject => args.subject,
    :body => args.body
  }).send
end
```

As you can see, there are no complications here. It's quite a simple task. Besides the `send_email` task, it also defines the `clobber` task to remove the generated file `sent_email.txt`. The `send_email` task has arguments that may be passed through the command line. At this point, we are able to check the rake task using the following commands:

```
$ rake send_email
Sending email...
Done!
$ cat sent_email.txt
Subject: Greeting!
Hello!
$ rake "send_email[Test, Hi]"
```

```
Sending email...
Done!
$ cat sent_email.txt
Subject: Test
Hi
$ rake clobber
```

Now we will talk about the final file that you will see in this chapter. There are many test frameworks for Ruby; here, we will take a look at the tests within the MiniTest framework that is in-built in Ruby since Version 1.9. So, you don't have to install any additional software except Ruby 1.9 to run these Ruby tests.

Check out the online documentation for the MiniTest framework at `http://goo.gl/elz2hH`.

Now let's try to test the tasks from the previous `Rakefile`. These are tests for `send_email` and the `clobber` rake tasks (place this code in the file with the `send_mail_test.rb` name in the `test` folder, which should be created in the folder where you have the previous two files):

```
require 'minitest/autorun'
require 'rake'

class TestSendEmail < MiniTest::Unit::TestCase
  def setup
    Rake.application.init
    Rake.application.load_rakefile

    @task = Rake::Task[:send_email]
    @task.reenable
  end

  def teardown
    if File.exists?(Mailer::DESTINATION)
      File.delete(Mailer::DESTINATION)
    end
  end

  def test_sending_email_with_default_params
    @task.invoke
    assert_equal email, "Subject: Greeting!\nHello!"
  end

  def test_sending_email_with_custom_subject_and_body
    @task.invoke('Test', 'Hi!')
    assert_equal email, "Subject: Test\nHi!"
  end

  def test_clobber_task_deletes_email
    @task.invoke
    Rake::Task[:clobber].invoke
    refute File.exists?(Mailer::DESTINATION)
  end

  private

  def email
    File.readlines(Mailer::DESTINATION).join
  end
end
```

Let's figure out what is going on here. In the beginning, we check the `setup` method. This is a cornerstone to test the rake tasks. To get ready to test a task, we have to initialize the Rake application first. Kindly note that we don't use the `rake` utility in these tasks because the tests would be rather inefficient if we test them by invoking commands with the code. For example, in this case, we won't have access to the rake task's internals and so we won't be able to make **stubs**, but sometimes, it's reasonable to replace real classes with their fake replacements in the tests.

The technique that is explained here is called **Test-driven Development** (**TDD**). If you are new to this, please follow the wiki page at `http://en.wikipedia.org/wiki/Test-driven_development` for more details.

After initializing the Rake application, we get the rake task and save it to the `@task` variable. This variable is then made accessible in each test. The last line of the `setup` method allows us to run the rake task several times. By default, the Rake counts the task starts and doesn't allow us to invoke the tasks again. This line of code resets the counter and provides us with an opportunity to run a task with the `invoke` method as many times as we want per one test. To avoid this behavior, a method named `reenable` is defined in `Rake::Task`. The next method, `teardown`, will be executed after each test run.

The lines after the `teardown` method define the tests themselves. They are the same as the ones for general Ruby unit tests. There are two tests for the `send_email` task and one for the `clobber` task:

- The first one tests the execution of the rake task without parameters
- The second one tests the rake task for the acceptance of the command-line arguments, `subject` and `body`
- The third test tests the `clobber` task that should delete the generated file

Note that the test file is placed to the separated subdirectory—`test`. We have done this to separate the tests from the main code. In real life, we have to split the code into many files because of its complexity. Following the known programming paradigm *divide and conquer*, we get huge benefits from this improvement.

Having the main logic in classes but not in the rake task actions is a good practice and allows us to test the classes in isolation. This is much easier than testing the rake tasks themselves.

Now, let's stop the theory for a while and just try to run the tests that are written. To run the tests, use the following command:

```
$ ruby test/send_mail_test.rb
```

The output of the command should look like the following:

```
MiniTest::Unit::TestCase is now Minitest::Test. From test/send_email_
test.rb:4:in `<main>'
Run options: --seed 19500


# Running:


Sending email...
Done!
.Sending email...
Done!
Sending email...
Done!
.Sending email...
Done!
Sending email...
Done!
```

```
Sending email...
Done!
.


Finished in 0.013278s, 225.9376 runs/s, 225.9376 assertions/s.


3 runs, 3 assertions, 0 failures, 0 errors, 0 skips
```

We can see that there are three tests and all of them have passed. Also, the output includes the outgoing messages from the `send_mail` task.

If you are interested in working with Rails, there is a useful article about how to test rake tasks in Rails with RSpec at `http://goo.gl/baLy0R`. **RSpec** is another test framework for Ruby (`http://rspec.info`).

Summary

In this chapter, we explained to you why we should test the rake tasks. Then, you saw an example of how to write tests for the rake tasks with the MiniTest testing framework.

In the next chapter, you will learn how Rake can be useful for continuous integration with an example of the Jenkins tool. You will also see how to install the Rake plugin for Jenkins.

9
Continuous Integration

Running tests is one of the goals of a build automation system. **Continuous integration** is a practice in software engineering that involves running the test process. A continuous integration system can run any kind of tasks, jobs, or their bundles, but it's often used to run tests. The goal of this chapter is to demonstrate how to use Rake in bundle with continuous integration with the help of a continuous integration tool called **Jenkins**.

In this chapter, we will cover the following topics:

- Introducing Jenkins
- Setting up Jenkins
- Configuring Jenkins to run rake tasks

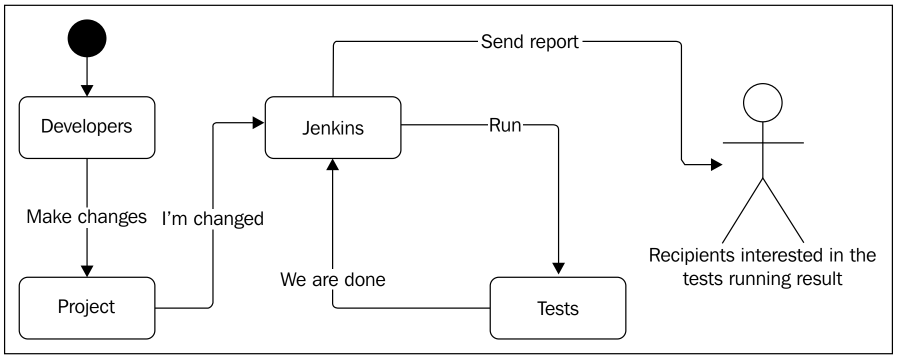
Introducing Jenkins

Jenkins is a tool that provides continuous integration services written in Java. Simply speaking, Jenkins' aim is to track changes performed by a software. For example, we have a Rails application that is under a **version control system** (the Ruby community currently prefers using **Git**), and we want to run the tests on each commit (or a change) to the project. When the tests fail, we would like to be informed about it via an e-mail. Jenkins could help us in the automation of this process. It may track the changes in the project and send e-mails accordingly.

[**Git** is a famous version control system. More information about Git can be found at `http://git-scm.com`.]

Note that you can set up Jenkins the way you want: for many types of tasks and to handle many events. However, the examples used in this chapter are intended to show you how to use Rake tasks. If you want to know more, please refer to the official documentation at `http://jenkins-ci.org`.

Now let's understand Jenkins and its workings through the following diagram:



The diagram explains the process of tracking the changes in the project, running the tests, and sending the notification to the developers. The following is the sequence of the processes:

1. Developers make changes in the project.
2. Jenkins receives a notification about the changes.
3. Jenkins runs the tests.
4. Jenkins sends e-mails to the developers after running the tests.

Steps 2 to 4 are configured in Jenkins, and in this chapter, you will see how to do this in detail.

Setting up Jenkins

There are many ways to install Jenkins. You can download the compiled version and install it within your operating system in a few seconds! However, Jenkins is usually run on a separate machine that is accessed 24 hours a day. However, it will be costly to buy or rent a server just for demonstration; this is why the chapter will demonstrate this on a virtual machine. For this purpose, we will use the **VirtualBox** and **Vagrant** command-line tools. We have to go through the following steps to set up Jenkins:

1. Firstly, install VirtualBox. Download it from the official site at `https://www.virtualbox.org` and follow the instructions given.

2. We should have a command-line tool for the easy configuration of virtual machines. For this purpose, install Vagrant. Go to the `http://www.vagrantup.com/downloads` page and download the application for your operating system.
3. Then, install it as a basic application though the installer of your operating system. To make sure that you have installed Vagrant correctly, type the following command in the terminal:

   ```
   $ vagrant -v
   Vagrant 1.4.3
   ```

   The command should output the version of the installed Vagrant application.
4. Now, it's time to get a virtual machine. For this purpose, get ready to use a box that I've created especially for your usage at `http://goo.gl/OCBgKj`. This is a Debian operating system based on Linux. Run the following command to generate the configuration file for the virtual machine:

   ```
   $ vagrant init jenkins http://goo.gl/OCBgKj
   ```
5. This command will generate `Vagrantfile`. This is a configuration file for the virtual server. Jenkins provides us with a web interface for its configuration, and we are going to use it. For this, the server should have opened HTTP access. In other words, it should have an IP address that we could type in the browser to get access to the web interface.
6. To achieve this, we should configure the virtual machine. Open the generated `Vagrantfile` and uncomment (remove the starting `#` symbol in a line) the following line of code:

   ```
   config.vm.network :hostonly, "192.168.33.10"
   ```
7. The following command will download the box created by me. Configure and start the virtual server (the size of the box is about 300 MB, so be ready to wait for some time):

   ```
   $ vagrant up
   ```
8. If you didn't have any errors during the installation of the server, you are ready to open the server's terminal with the following command:

   ```
   $ vagrant ssh
   ```
9. Now you are on the server and the command line is ready to install Jenkins. According to the official documentation (`http://goo.gl/OzEX4p`), use the following commands:

   ```
   $ wget -q -O - http://pkg.jenkins-ci.org/debian/jenkins-ci.org.key
   | sudo apt-key add -
   ```

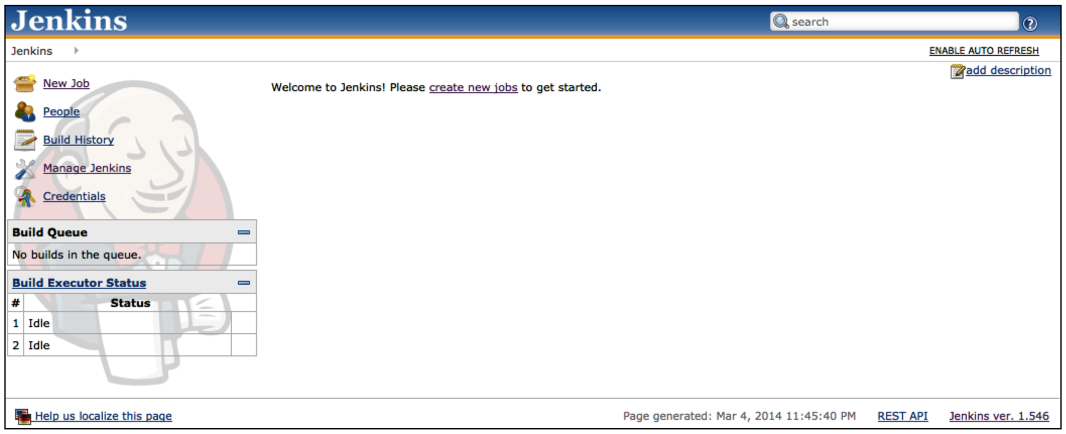
```
$ sudo sh -c 'echo deb http://pkg.jenkins-ci.org/debian binary/ >
/etc/apt/sources.list.d/jenkins.list'
$ sudo apt-get update
$ sudo apt-get install -y jenkins
```

> Note that you have to run these commands on the server. If you have some problems installing Jenkins with the preceding commands, try to follow these instructions given at `http://pkg.jenkins-ci.org/debian-stable`. They are written especially for the Debian operating system.

That's it. We have installed Jenkins.

10. Open your favorite browser and paste the following link at `http://192.168.33.10:8080`.

You should see the web interface of Jenkins. The following is a screenshot of the screen that you should have:



Jenkins

search

Jenkins

ENABLE AUTO REFRESH

add description

New Job

People

Build History

Manage Jenkins

Credentials

Welcome to Jenkins! Please create new jobs to get started.

Build Queue

No builds in the queue.

Build Executor Status

#	Status
1	Idle
2	Idle

Help us localize this page

Page generated: Mar 4, 2014 11:45:40 PM REST API Jenkins ver. 1.546

Until now, Ruby and Rake have not been installed to the server. So, install them separately using the following command:

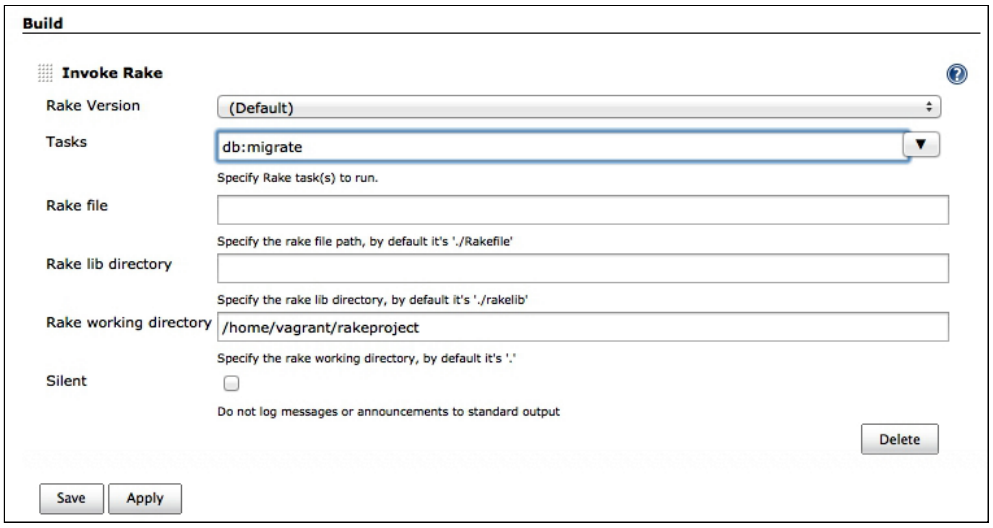
```
$ sudo apt-get install -y ruby rake
```

Configuring Jenkins to run rake tasks

Now it is time to demonstrate Jenkins in action. As we are going to run the rake tasks in our project build, we have to install the Rake plugin for Jenkins. Perform the following steps:

1. Go to `http://192.168.33.10:8080/pluginManager`, find the Rake plugin there, and install it.
2. In the next step, create an application in Jenkins. To do so, go to `http://192.168.33.10:8080/view/All/newJob`. Fill in the job name and the project name (let it be `Test rake task`), select **Build a free-style software** project option from the proposed, and click on the **OK** button at the bottom of the page.
3. You will be redirected to the configuration page of the created project. There, you will find the **Build** section with the **Add build step** dropdown. Choose the **Invoke Rake** option from the dropdown.
4. You will see the **Tasks** input. Considering that we are going to run the Rake task `db:migrate`, fill in the field with this task.
5. Now, we should configure the path to the folder of our Rake project for Jenkins. Click on the **Advanced** button in the section and additional fields will appear.
6. Find the **Rake working directory** input field. This field needs the path where our `Rakefile` will be present. Type the path `/home/vagrant/rakeproject` into it.

   The following screenshot shows us what the final section will look like:


Build

Invoke Rake

Rake Version: (Default)

Tasks: db:migrate

Specify Rake task(s) to run.

Rake file:

Specify the rake file path, by default it's './Rakefile'

Rake lib directory:

Specify the rake lib directory, by default it's './rakelib'

Rake working directory: /home/vagrant/rakeproject

Specify the rake working directory, by default it's '.'

Silent: ☐

Do not log messages or announcements to standard output

7. To test the rake task manually, find the **Build now** link in the sidebar situated on the left-hand side when you are on the project page. Click on this link and the process will start.
8. After running each build, a report will be generated with the logs, and you will be able to see the console output. The first result will fail because `Rakefile` is not present there yet, so you have to see the fails in the console output.
9. Now, create a valid `Rakefile` in the `/home/vagrant/rakeproject` path on the server:

   ```
   task 'db:migrate' do
     puts 'Hello for Rake!'
   end
   ```

10. Run the build again and it will be fixed.

It is possible to configure Jenkins to run builds automatically on each change of the Rakefile, but the investigation of this is left to you.

Summary

In this chapter, we were briefly introduced to Jenkins, the continuous integration software. We saw how to configure it using an example that showed us how to use rake tasks.

The next chapter is the last chapter of the book and our goal will be to summarize the knowledge you have gained from all the chapters. Also, you will find some information about the tools inspired by Rake, which are very successful in the Ruby world.

10
Relentless Automation

In this chapter, you will find real examples on how Rake is used by famous applications. You will also get to learn about the tools that are inspired by Rake and have familiar DSL. Also, you will be introduced to a similar Rake tool that has some advantages over Rake.

In this chapter, we will cover the following topics:

- Examples of Rake being used by famous gems
- Other examples of Sinatra using Rake
- Thor—the next generation of Rake

Examples of Rake being used by famous gems

Currently, Rake is used by the Ruby community in general. The most frequent application of Rake is to automate running tests and generating the project documentation. There are tons of gems that use Rake for these purposes. You may choose to trust me or check it out yourself by visiting `http://github.com`. In this chapter, we will examine a **Sinatra** gem that is more suitable for us. It uses Rake for both these tasks: running tests and generating documentation.

[Sinatra is a lightweight framework that allows us to create web applications with minimum effort. To read more about this, check out the official documentation at `http://www.sinatrarb.com`.]

There is one more interesting application of Rake. Some tools inherit the basic classes of Rake and in this way, extend the functionality and the DSL for their needs. One of the most famous tools that use Rake like this is **Capistrano**. It's a deployment tool that has been rewritten recently from pure Ruby implementations to inheritance from Rake. Further, you will see what the benefits of this change were.

[Capistrano is a remote server automation and deployment tool written in Ruby (official site `http://capistranorb.com`).]

The pain of task execution

To understand why a lot of gems use Rake as a task executor, we should figure out what the problem with running the tests without Rake is. In *Chapter 8*, *Testing Rake Tasks*, we have already seen how to write and run the tests that are placed in one file. This is a rather simple process, and the following command-line example serves as a reminder for you:

```
$ ruby test/some_feature_test.rb
```

However, the most frequently used form looks like the following command line:

```
$ ruby -Ilib -Itest test/some_feature_test.rb
```

This command additionally loads the required libraries to run the tests. So, in this example, the `lib` and `test` folders will be loaded before the execution of the code in the `some_feature_test.rb` file. You should not make a big effort to run the tests that are situated in one file.

Imagine that we need to run all projects' tests and they are placed in many files. How do we run them by a single command? Well, it may be not so straightforward if you don't know the bash or some Ruby tricks. According to `http://stackoverflow.com`, people run the tests by one command in many ways. The following are examples some of them:

```
$ ruby -e 'ARGV.each { |path| require path }' test/test_*.rb
$ for file in spec/*.rb; do ruby $file; done
```

[Stack Overflow is a popular site where users ask questions and get answers from more experienced users.]

You may argue with the conclusion that these commands are not suitable at all because you have to remember these complicated commands and need to spend a lot of time to achieve the simple result.

Sinatra using Rake to run tests

Sinatra has a lot of tasks that are separated by many files. Let's figure out how the issue is solved in this gem. There is only one case that explains us how to do it—open its sources and look into `Rakefile`. Sources of Sinatra are placed on GitHub at `https://github.com/sinatra/sinatra`. Open the `Readme` file in the browser after following the link `http://goo.gl/M6gq79` and peek into the online file, or clone the Git repository to your local machine and open the file using your favorite editor. We are interested in the following code in `Rakefile`:

```
...
require 'rake/testtask'
...
Rake::TestTask.new(:test) do |t|
  t.test_files = FileList['test/*_test.rb']
  t.ruby_opts = ['-rubygems'] if defined? Gem
  t.ruby_opts << '-I.'
  t.warning = true
end
...
```

These lines of code do some magic. This is another feature of Rake that hasn't been described yet in the book. So, this is the right time to do so. As you might have guessed, this piece of code creates the `test` task and the lines in the block configure the task. At first, we set the list of test files for the task and then we set the command-line options: `-rubygems` if there is a defined `Gem` constant (There are operating systems which don't have installed `rubygems` as a gem. The `rubygems` gem can be installed as a package (or just program) to the system. In this way the `Gem` variable won't be defined and we should use the `-rubygems` option to run a Ruby script/test. It tells to load path with installed rubygems in `$LOAD_PATH`—this is a variable that contains a list of paths to find libraries in Ruby scripts.) and `-I.` to include the current folder in the runtime execution. The `warning` option speaks for itself.

Pay attention to the fact that you have to require the defined class `Rake::TestTask` to use this with the `require 'rake/testtask'` statement in the beginning of `Rakefile`. With the lines of code from the previous snippet, we are able to run the tests placed in the `test` folder. It's assumed that the test files end with `_test.rb`. Now, the command to run the tests transforms to the following:

```
$ rake test
```

Often, maintainers of Ruby gems set the `default` task to `test`, and in this case, the command is cut to one word:

```
$ rake
```

Go back to *Chapter 1, The Software Task Management Tool – Rake* to revise how to set the `default` task to `test` in `Rakefile`:

```
rake :default => :test
```

There are many other useful options of `Rake::TestTask`. For example, the `verbose` option expects a Boolean value and shows you the constructed command that runs the tests, the `libs` option includes folders to the command line, and `pattern` allows you to set patterns for the test files (the default is `test/test*.rb`). To find more options, look at the sources of `Rake::TestTask` at `http://goo.gl/SjwSsz`.

Gems that use the RSpec tool for tests usually don't need to have some external helper execute the tests because RSpec has its own convention to name the tests files and decide where to place them. If you follow the RSpec convention to run all tests, you have to use the following simple command:

```
$ rspec
```

Sinatra using Rake to generate documentation

The task to generate documentation in Sinatra is rather simple; it just runs the external `yardoc` utility to generate them. To conclude this, you may look into the `Rakefile` of Sinatra:

```
    desc 'Generate RDoc under doc/api'
    task 'doc'     => ['doc:api']
    task('doc:api') { sh "yardoc -o doc/api" }
    CLEAN.include 'doc/api'
```

There is nothing new for you here. Two tasks are defined here: `doc` and `doc:api`. They are just aliases as you see—the `doc` task depends on the `doc:api` task. However, the `doc` task's action is absent, so it is created just to give the `doc:api` task pseudonym. The last line of the code example is familiar for you from *Chapter 4, Cleaning Up a Build*. It just adds the generated docs in the list of tasks to be removed by the `clean` task.

[**Yardoc** is a Ruby documentation tool. To get to know more about it, please follow the official site at `http://yardoc.org`.]

If Sinatra uses such processes to generate the documentation, it doesn't mean that other gems do the same or that you should follow this approach in your project too. There are many other techniques to generate the documentation. For instance, you can use the `rdoc` gem or even write your generator from the scratch using Rake.

Capistrano extending the Rake implementation

As Ruby is an object-oriented language and Rake is written in this language, you are able to inherit the basic classes of Rake and change its standard behavior. This allows you to create power tools with DSL. In fact, the newest version of Capistrano v3 uses Rake in this way. The following is a good example to see this appliance in action.

First of all, take a look at the `capistrano/application.rb` file (short link: `http://goo.gl/iJv7xq`):

```
module Capistrano
  class Application < Rake::Application

    def initialize
      super
      @rakefiles = %w{capfile Capfile capfile.rb Capfile.rb} <<
capfile
    end

    def name
      "cap"
    end

    def run
      Rake.application = self
      super
    end

    ...

  end

end
```

The idea here is simple—just inherit the base class of Rake `Capistrano::Application` and change the parent methods. This chunk of code changes the possible names of `Rakefile` and changes the name of the `rake` command to `cap`. Also, the `run` method was slightly modified, as it changes the `application` attribute of Rake to `self`, so now, Rake will use the `Capistrano::Application` class instead of the `Rake::Application` class in its core. In the `bin` folder, you can see this class in action in the `cap` file (short link: `http://goo.gl/59LPs7`):

```
#!/usr/bin/env ruby
require 'capistrano/all'
Capistrano::Application.new.run
```

This code allows us to use the `cap` command, but in reality, it just tunes Rake. This means that you are able to use this command as the `rake` utility. The `cap` command accepts all the `rake` options.

You can see the trick when extending DSL in the `dsl/task_enhancements.rb` file found at `http://goo.gl/OzABkA`. It adds such useful DSL methods as `before`, `after`, and `remote_file`.

Capistrano v2 was built in pure Ruby and recently it was rewritten to Rake.
The following are the benefits from the architecture changes:

- V3 is faster.
- It is easier to work with.
- It has better modularization. The tool can now be used to easily write extensions for other languages . Currently, there are extensions for PHP's **Symphony** and **WordPress**.
- It has a better DSL.
- It has the ability to integrate Capistrano with other deployment tools such as Chef and Puppet.

If you have a project with similar problems, inheriting the functionality and extending the DSL may be a better choice for you, rather than writing the application from scratch.

Other examples of Sinatra using Rake

If you look at the `Rakefile` of Sinatra, you may see that there a lot of examples of the Rake appliance. For example, you could find a task to package the Sinatra into a gem:

```
file package('.gem') =>
   %w[pkg/ sinatra.gemspec] + spec.files do |f|
  sh "gem build sinatra.gemspec"
  mv File.basename(f.name), f.name
end
```

You will also find how the developers configure the list for the `clobber` task (go back to *Chapter 4*, *Cleaning Up a Build*), installation tasks, and so on. Reviewing a real example is a good practice to consolidate knowledge, but here, we have limited the scope to be able to fully investigate `Rakefile`, so this exercise is delegated to you.

Thor – the next generation of Rake

There is a tool that can be used as an alternative to Rake and promises to get rid of Rake's disadvantage—parsing command-line arguments (as you can remember from *Chapter 1*, *The Software Task Management Tool – Rake*, it was a little bit inconvenient). The tool is **Thor** (official page: `http://whatisthor.com`). Another advantage of this tool is a great feature that is completely absent in Rake—**generators**. For example, Rails uses Thor's generators to generate a skeleton for an application.

Now, let's create some test task of Thor. Say, the task should just get an optional argument, an e-mail, and just write this to the standard output. Just create this file with the following code and the name `test.thor`:

```
class Test < Thor
  desc 'send_email', 'Send mail'
  method_option :email,
    :aliases => '-e',
    :desc => 'email of a recipient'
  def send_email
    puts "Recipient: #{options[:email]}"
  end
end
```

Now if you have installed Thor, you will be able to run the `test:send_email` task from the command line and pass the `-e` (or `--email`) argument, as shown:

```
$  thor test:send_email -e 'ka8725@gmail.com'
Recipient: ka8725@gmail.com
$  thor test:send_email --email 'ka8725@gmail.com'
Recipient: ka8725@gmail.com
```

To install Thor, use the `gem` command as shown:

```
$ gem install thor
```

Researching on the generator feature of Thor has been proposed for you. In fact, Thor has some features that are more convenient than Rake, for example, installing a task to the system or uninstalling it, creating executable scripts, and so on. Unfortunately, this will require an entire new book. If you are really interested in this tool, please follow the official site at `http://whatisthor.com` and its GitHub wiki page at `http://goo.gl/q9aONf`.

Despite the fact that Thor has some advantages over Rake, it can't replace it completely, because it only solves some specific tasks. Thor doesn't have as powerful instruments to work with files as Rake. Also, remember that Rake is embedded in Ruby. This is the biggest advantage because you don't have to install any third-party tools. However, simultaneous usage of these tools may bring you the real power that exists in Rails.

Summary

We have come to the end of this book, so let's revise what we learned. At the start, there was an explanation of Rake's foundation and its DSL. We saw what a `Rakefile` is, how to define custom tasks and their prerequisites, what the default rake task is, how to define global rake tasks, and how to use the `rake` command-line utility and its common options.

Next, you were taught how to work with files and what arsenal Rake has for operations with the files. How to refactor and get rid of task duplication with `rule` was explained in next chapter. After that, we saw how to clean a build with the standard features of Rake, such as `CLOBBER` and `CLEAN` lists. Software productivity is the main problem of Ruby, and sometimes, parallelism might speed up a Rake application with `multitask`. *Chapter 5, Running Tasks in Parallel*, explained when it's rational to use multitasking and how to use it.

Then, we discussed another existing problem in the programming world—debugging. You were shown how to debug rake tasks with help of the `rake` command-line tool's arguments and with the `debugger` gem.

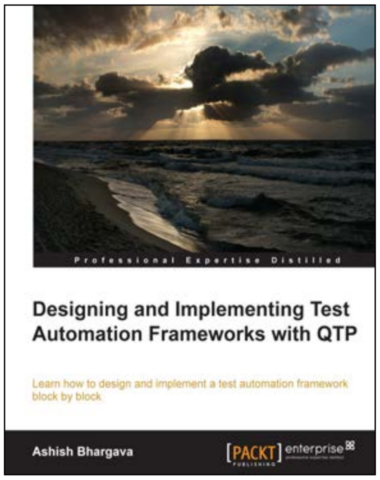
Any Rails application contains Rake and a Rake application is always ready to use them in a convenient way. This was presented in *Chapter 7, Integration with Rails*. This chapter also showed you how to run the recurrent rake tasks. You can run them on any Rake or Ruby project as well, but as it's used often with the Rails bundle, this information was provided in this chapter.

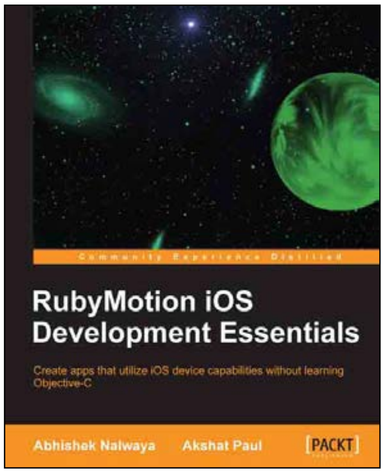
Why you should test rake tasks and what is possible if you skip this was discussed in the next chapter. After this, you walked through the Jenkins installation and saw how it may use the Rake as an installed plugin. Jenkins is a continuous integration tool, so this knowledge may be applied to any other similar software.

Thor was discussed at the end of the book, and you could put it there because the emphasis was on the real examples of the Rake usage.

This book doesn't contain exhaustive information about Rake. The examples given in the book are simplified. But now you've read it, you are ready to start your journey in to the Rake world. I hope this information will help you at this point. However, if you have a rather complicated project, you will have to perform tasks such as maintaining, refactoring, and changing the architecture. These are unavoidable processes of any project. The situations discussed in this book may not help you in such case. Reading the source code of Rake (`https://github.com/jimweirich/rake`) and other great tools such as **Capistrano** (`https://github.com/capistrano/capistrano`) is the single best way to do this.

Index


Professional Expertise Distilled
Designing and Implementing Test Automation Frameworks with QTP
Learn how to design and implement a test automation framework block by block
Ashish Bhargava
[PACKT] enterprise


Community Experience Distilled
RubyMotion iOS Development Essentials
Create apps that utilize iOS device capabilities without learning Objective-C
Abhishek Nalwaya
Akshat Paul
[PACKT]