**Rakefile Format**

First of all, there is no special format for a Rakefile. A Rakefile contains executable Ruby code. Anything legal in a ruby script is allowed in a Rakefile.

Now that we understand there is no special syntax in a Rakefile, there are some conventions that are used in a Rakefile that are a little unusual in a typical Ruby program. Since a Rakefile is tailored to specifying tasks and actions, the idioms used in a Rakefile are designed to support that.

So, what goes into a Rakefile?

**Tasks**

Tasks are the main unit of work in a Rakefile. Tasks have a name (usually given as a symbol or a string), a list of prerequisites (more symbols or strings) and a list of actions (given as a block).

**Simple Tasks**

A task is declared by using the task method. task takes a single parameter that is the name of the task.

task :name

**Tasks with Prerequisites**

Any prerequisites are given as a list (enclosed in square brackets) following the name and an arrow (=>).

task name: [:prereq1, :prereq2]

**NOTE:** Although this syntax looks a little funky, it is legal Ruby. We are constructing a hash where the key is :name and the value for that key is the list of prerequisites. It is equivalent to the following …

hash = Hash.new

hash[:name] = [:prereq1, :prereq2]

task(hash)

You can also use strings for task names and prerequisites, rake doesn't care. This is the same task definition:

task 'name' => %w[prereq1 prereq2]

As is this:

task name: %w[prereq1 prereq2]

We'll prefer this style for regular tasks with prerequisites throughout the rest of the document. Using an array of strings for the prerequisites means you will need to make fewer changes if you need to move tasks into namespaces or perform other refactorings.

**Tasks with Actions**

Actions are defined by passing a block to the task method. Any Ruby code can be placed in the block. The block may reference the task object via the block parameter.

task name: [:prereq1, :prereq2] do |t|

# actions (may reference t)

end

**Multiple Definitions**

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

For example, the following is equivalent to the single task specification given above.

task :name

task name: :prereq1

task name: %w[prereq2]

task :name do |t|

# actions

end

**File Tasks**

Some tasks are designed to create a file from one or more other files. Tasks that generate these files may be skipped if the file already exists. File tasks are used to specify file creation tasks.

File tasks are declared using the file method (instead of the task method). In addition, file tasks are usually named with a string rather than a symbol.

The following file task creates a executable program (named prog) given two object files named a.o and b.o. The tasks for creating a.o and b.o are not shown.

file "prog" => ["a.o", "b.o"] do |t|

sh "cc -o #{t.name} #{t.prerequisites.join(' ')}"

end

**Directory Tasks**

It is common to need to create directories upon demand. The directory convenience method is a short-hand for creating a FileTask that creates the directory. For example, the following declaration …

directory "testdata/examples/doc"

is equivalent to …

file "testdata" do |t| mkdir t.name end

file "testdata/examples" => ["testdata"] do |t| mkdir t.name end

file "testdata/examples/doc" => ["testdata/examples"] do |t| mkdir t.name end

The directory method does not accept prerequisites or actions, but both prerequisites and actions can be added later. For example …

directory "testdata"

file "testdata" => ["otherdata"]

file "testdata" do

cp Dir["standard\_data/\*.data"], "testdata"

end

**Tasks with Parallel Prerequisites**

Rake allows parallel execution of prerequisites using the following syntax:

multitask copy\_files: %w[copy\_src copy\_doc copy\_bin] do

puts "All Copies Complete"

end

In this example, copy\_files is a normal rake task. Its actions are executed whenever all of its prerequisites are done. The big difference is that the prerequisites (copy\_src, copy\_bin and copy\_doc) are executed in parallel. Each of the prerequisites are run in their own Ruby thread, possibly allowing faster overall runtime.

**Secondary Prerequisites**

If any of the primary prerequisites of a multitask have common secondary prerequisites, all of the primary/parallel prerequisites will wait until the common prerequisites have been run.

For example, if the copy\_*xxx* tasks have the following prerequisites:

task copy\_src: :prep\_for\_copy

task copy\_bin: :prep\_for\_copy

task copy\_doc: :prep\_for\_copy

Then the prep\_for\_copy task is run before starting all the copies in parallel. Once prep\_for\_copy is complete, copy\_src, copy\_bin, and copy\_doc are all run in parallel. Note that prep\_for\_copy is run only once, even though it is referenced in multiple threads.

**Thread Safety**

The Rake internal data structures are thread-safe with respect to the multitask parallel execution, so there is no need for the user to do extra synchronization for Rake's benefit. However, if there are user data structures shared between the parallel prerequisites, the user must do whatever is necessary to prevent race conditions.

**Tasks with Arguments**

Prior to version 0.8.0, rake was only able to handle command line arguments of the form NAME=VALUE that were passed into Rake via the ENV hash. Many folks had asked for some kind of simple command line arguments, perhaps using “–” to separate regular task names from argument values on the command line. The problem is that there was no easy way to associate positional arguments on the command line with different tasks. Suppose both tasks :a and :b expect a command line argument: does the first value go with :a? What if :b is run first? Should it then get the first command line argument.

Rake 0.8.0 solves this problem by explicitly passing values directly to the tasks that need them. For example, if I had a release task that required a version number, I could say:

rake release[0.8.2]

And the string “0.8.2” will be passed to the :release task. Multiple arguments can be passed by separating them with a comma, for example:

rake name[john,doe]

Just a few words of caution. The rake task name and its arguments need to be a single command line argument to rake. This generally means no spaces. If spaces are needed, then the entire name + argument string should be quoted. Something like this:

rake "name[billy bob, smith]"

(Quoting rules vary between operating systems and shells, so make sure you consult the proper docs for your OS/shell).

**Tasks that Expect Parameters**

Parameters are only given to tasks that are setup to expect them. In order to handle named parameters, the task declaration syntax for tasks has been extended slightly.

For example, a task that needs a first name and last name might be declared as:

task :name, [:first\_name, :last\_name]

The first argument is still the name of the task (:name in this case). The next two arguments are the names of the parameters expected by :name in an array (:first\_name and :last\_name in the example).

To access the values of the parameters, the block defining the task behaviour can now accept a second parameter:

task :name, [:first\_name, :last\_name] do |t, args|

puts "First name is #{args.first\_name}"

puts "Last name is #{args.last\_name}"

end

The first argument of the block “t” is always bound to the current task object. The second argument “args” is an open-struct like object that allows access to the task arguments. Extra command line arguments to a task are ignored.

If you wish to specify default values for the arguments, you can use the with\_defaults method in the task body. Here is the above example where we specify default values for the first and last names:

task :name, [:first\_name, :last\_name] do |t, args|

args.with\_defaults(:first\_name => "John", :last\_name => "Dough")

puts "First name is #{args.first\_name}"

puts "Last name is #{args.last\_name}"

end

**Tasks that Expect Parameters and Have Prerequisites**

Tasks that use parameters have a slightly different format for prerequisites. Use the arrow notation to indicate the prerequisites for tasks with arguments. For example:

task :name, [:first\_name, :last\_name] => [:pre\_name] do |t, args|

args.with\_defaults(:first\_name => "John", :last\_name => "Dough")

puts "First name is #{args.first\_name}"

puts "Last name is #{args.last\_name}"

end

**Tasks that take Variable-length Parameters**

Tasks that need to handle a list of values as a parameter can use the extras method of the args variable. This allows for tasks that can loop over a variable number of values, and its compatible with named parameters as well:

task :email, [:message] do |t, args|

mail = Mail.new(args.message)

recipients = args.extras

recipients.each do |target|

mail.send\_to(target)

end

end

There is also the convenience method to\_a that returns all parameters in the sequential order they were given, including those associated with named parameters.

**Deprecated Task Parameters Format**

There is an older format for declaring task parameters that omitted the task argument array and used the :needs keyword to introduce the dependencies. That format is still supported for compatibility, but is not recommended for use. The older format may be dropped in future versions of rake.

**Accessing Task Programmatically**

Sometimes it is useful to manipulate tasks programmatically in a Rakefile. To find a task object use [Rake::Task.[]](https://ruby.github.io/rake/Rake/Task.html#method-c-5B-5D).

**Programmatic Task Example**

For example, the following Rakefile defines two tasks. The :doit task simply prints a simple “DONE” message. The :dont class will lookup the doit class and remove (clear) all of its prerequisites and actions.

task :doit do

puts "DONE"

end

task :dont do

Rake::Task[:doit].clear

end

Running this example:

$ rake doit

(in /Users/jim/working/git/rake/x)

DONE

$ rake dont doit

(in /Users/jim/working/git/rake/x)

$

The ability to programmatically manipulate tasks gives rake very powerful meta-programming capabilities w.r.t. task execution, but should be used with caution.

**Rules**

When a file is named as a prerequisite, but does not have a file task defined for it, Rake will attempt to synthesize a task by looking at a list of rules supplied in the Rakefile.

Suppose we were trying to invoke task “mycode.o”, but no task is defined for it. But the rakefile has a rule that look like this …

rule '.o' => ['.c'] do |t|

sh "cc #{t.source} -c -o #{t.name}"

end

This rule will synthesize any task that ends in “.o”. It has a prerequisite a source file with an extension of “.c” must exist. If Rake is able to find a file named “mycode.c”, it will automatically create a task that builds “mycode.o” from “mycode.c”.

If the file “mycode.c” does not exist, rake will attempt to recursively synthesize a rule for it.

When a task is synthesized from a rule, the source attribute of the task is set to the matching source file. This allows us to write rules with actions that reference the source file.

**Advanced Rules**

Any regular expression may be used as the rule pattern. Additionally, a proc may be used to calculate the name of the source file. This allows for complex patterns and sources.

The following rule is equivalent to the example above.

rule( /\.o$/ => [

proc {|task\_name| task\_name.sub(/\.[^.]+$/, '.c') }

]) do |t|

sh "cc #{t.source} -c -o #{t.name}"

end

**NOTE:** Because of a *quirk* in Ruby syntax, parenthesis are required on **rule** when the first argument is a regular expression.

The following rule might be used for Java files …

rule '.class' => [

proc { |tn| tn.sub(/\.class$/, '.java').sub(/^classes\//, 'src/') }

] do |t|

java\_compile(t.source, t.name)

end

**NOTE:** java\_compile is a hypothetical method that invokes the java compiler.

**Importing Dependencies**

Any ruby file (including other rakefiles) can be included with a standard Ruby require command. The rules and declarations in the required file are just added to the definitions already accumulated.

Because the files are loaded *before* the rake targets are evaluated, the loaded files must be “ready to go” when the rake command is invoked. This makes generated dependency files difficult to use. By the time rake gets around to updating the dependencies file, it is too late to load it.

The import command addresses this by specifying a file to be loaded *after* the main rakefile is loaded, but *before* any targets on the command line are invoked. In addition, if the file name matches an explicit task, that task is invoked before loading the file. This allows dependency files to be generated and used in a single rake command invocation.

Example:

require 'rake/loaders/makefile'

file ".depends.mf" => [SRC\_LIST] do |t|

sh "makedepend -f- -- #{CFLAGS} -- #{t.prerequisites} > #{t.name}"

end

import ".depends.mf"

If “.depends” does not exist, or is out of date w.r.t. the source files, a new “.depends” file is generated using makedepend before loading.

**Comments**

Standard Ruby comments (beginning with “#”) can be used anywhere it is legal in Ruby source code, including comments for tasks and rules. However, if you wish a task to be described using the “-T” switch, then you need to use the desc command to describe the task.

Example:

desc "Create a distribution package"

task package: %w[ ... ] do ... end

The “-T” switch (or “–tasks” if you like to spell things out) will display a list of tasks that have a description. If you use desc to describe your major tasks, you have a semi-automatic way of generating a summary of your Rake file.

traken$ rake -T

(in /home/.../rake)

rake clean # Remove any temporary products.

rake clobber # Remove any generated file.

rake clobber\_rdoc # Remove rdoc products

rake contrib\_test # Run tests for contrib\_test

rake default # Default Task

rake install # Install the application

rake lines # Count lines in the main rake file

rake rdoc # Build the rdoc HTML Files

rake rerdoc # Force a rebuild of the RDOC files

rake test # Run tests

rake testall # Run all test targets

Only tasks with descriptions will be displayed with the “-T” switch. Use “-P” (or “–prereqs”) to get a list of all tasks and their prerequisites.

**Namespaces**

As projects grow (and along with it, the number of tasks), it is common for task names to begin to clash. For example, if you might have a main program and a set of sample programs built by a single Rakefile. By placing the tasks related to the main program in one namespace, and the tasks for building the sample programs in a different namespace, the task names will not interfere with each other.

For example:

namespace "main" do

task :build do

# Build the main program

end

end

namespace "samples" do

task :build do

# Build the sample programs

end

end

task build: %w[main:build samples:build]

Referencing a task in a separate namespace can be achieved by prefixing the task name with the namespace and a colon (e.g. “main:build” refers to the :build task in the main namespace). Nested namespaces are supported.

Note that the name given in the task command is always the unadorned task name without any namespace prefixes. The task command always defines a task in the current namespace.

**FileTasks**

File task names are not scoped by the namespace command. Since the name of a file task is the name of an actual file in the file system, it makes little sense to include file task names in name space. Directory tasks (created by the directory command) are a type of file task and are also not affected by namespaces.

**Name Resolution**

When looking up a task name, rake will start with the current namespace and attempt to find the name there. If it fails to find a name in the current namespace, it will search the parent namespaces until a match is found (or an error occurs if there is no match).

The “rake” namespace is a special implicit namespace that refers to the toplevel names.

If a task name begins with a “^” character, the name resolution will start in the parent namespace. Multiple “^” characters are allowed.

Here is an example file with multiple :run tasks and how various names resolve in different locations.

task :run

namespace "one" do

task :run

namespace "two" do

task :run

# :run => "one:two:run"

# "two:run" => "one:two:run"

# "one:two:run" => "one:two:run"

# "one:run" => "one:run"

# "^run" => "one:run"

# "^^run" => "rake:run" (the top level task)

# "rake:run" => "rake:run" (the top level task)

end

# :run => "one:run"

# "two:run" => "one:two:run"

# "^run" => "rake:run"

end

# :run => "rake:run"

# "one:run" => "one:run"

# "one:two:run" => "one:two:run"

**FileLists**

FileLists are the way Rake manages lists of files. You can treat a FileList as an array of strings for the most part, but FileLists support some additional operations.

**Creating a FileList**

Creating a file list is easy. Just give it the list of file names:

fl = FileList['file1.rb', file2.rb']

Or give it a glob pattern:

fl = FileList['\*.rb']

**Odds and Ends**

**do/end versus { }**

Blocks may be specified with either a do/end pair, or with curly braces in Ruby. We *strongly* recommend using do/end to specify the actions for tasks and rules. Because the rakefile idiom tends to leave off parentheses on the task/file/rule methods, unusual ambiguities can arise when using curly braces.

For example, suppose that the method object\_files returns a list of object files in a project. Now we use object\_files as the prerequisites in a rule specified with actions in curly braces.

# DON'T DO THIS!

file "prog" => object\_files {

# Actions are expected here (but it doesn't work)!

}

Because curly braces have a higher precedence than do/end, the block is associated with the object\_files method rather than the file method.

This is the proper way to specify the task …

# THIS IS FINE

file "prog" => object\_files do

# Actions go here

end

**Rakefile Path**

When issuing the rake command in a terminal, Rake will look for a Rakefile in the current directory. If a Rakefile is not found, it will search parent directories until one is found.

For example, if a Rakefile resides in the project/ directory, moving deeper into the project's directory tree will not have an adverse effect on rake tasks:

$ pwd

/home/user/project

$ cd lib/foo/bar

$ pwd

/home/user/project/lib/foo/bar

$ rake run\_pwd

/home/user/project

As far as rake is concerned, all tasks are run from the directory in which the Rakefile resides.

**Multiple Rake Files**

Not all tasks need to be included in a single Rakefile. Additional rake files (with the file extension “.rake”) may be placed in rakelib directory located at the top level of a project (i.e. the same directory that contains the main Rakefile).

Also, rails projects may include additional rake files in the lib/tasks directory.

**Clean and Clobber Tasks**

Through require 'rake/clean' Rake provides clean and clobber tasks:

**clean**

Clean up the project by deleting scratch files and backup files. Add files to the CLEAN FileList to have the clean target handle them.

**clobber**

Clobber all generated and non-source files in a project. The task depends on clean, so all the CLEAN files will be deleted as well as files in the CLOBBER FileList. The intent of this task is to return a project to its pristine, just unpacked state.

You can add file names or glob patterns to both the CLEAN and CLOBBER lists.

**Phony Task**

The phony task can be used as a dependency to allow file-based tasks to use non-file-based-tasks as prerequisites without forcing them to rebuild. You can require 'rake/phony' to add the phony task.