Table of Contents

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

17. Testing, Debugging, Optimizing, and Documenting

The recipes in previous chapters focus on writing code to do what you want. This chapter focuses on verifying that your code really works, and on fixing it when it breaks. We start off simple and move to more advanced debugging techniques.

What happens when your program has a bug? The best-case scenario is that you discover the bug before it affects anyone, including other developers. That's the goal of unit tests (Recipe 17.7). Ruby and the Ruby community promote a philosophy of writing automated tests as (or even before) you write the corresponding functionality. At every stage of development, you know that your program works, and if you make a change that breaks something, you know about it immediately. These tests can replace much boring manual testing and bug hunting.

Suppose a bug slips past your tests, and you only discover it in production. How's it going to manifest itself? If you're lucky, you'll see an exception: a notification from some piece of Ruby code that something is wrong.

Exceptions interrupt the normal flow of execution, and, if not handled, will crash the program. The good news is that they give you a place in the code to start debugging. It's worse if a bug *doesn't* cause an exception, because you'll only notice its byproducts: corrupt data or even security violations. We show code for handling exceptions (Recipes 17.3 and 17.4) and for creating your own (Recipe 17.2).

Successful debugging means reproducing the bug in an environment where you can poke at it. This may mean dropping from a running program into an `irb` session (Recipe 17.10), or it may be as simple as adding diagnostic messages that make the program show its work (Recipe 17.1).

Even a program that has no noticeable bugs may run too slowly or use too many resources. Ruby provides two tools for doing performance optimization: a profiler (Recipe 17.12) and a benchmarking suite (Recipe 17.13). It's easy to create your own analysis tools by writing a trace function that hooks into the Ruby interpreter as it runs. The call graph tracker presented at chapter's end (Recipe 17.15) exploits this feature.

Recipe 17.1. Running Code Only in Debug Mode

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

Problem

You want to print out debugging messages or run some sanity-checking code, but only while you're developing your application;, not when you're running it in production.

Solution

Run the code only if the global variable `$DEBUG` is true. You can trigger debug mode by passing in the `--debug` switch to the Ruby interpreter, or you can set the variable `$DEBUG` to true within your code.

Here's a Ruby program to divide two random numbers. It contains a trivial bug. It usually runs to completion, but sometimes it crashes. A line of debug code has been added to give some more visibility into the internal workings of the program:

```
#!/usr/bin/env ruby
# divide.rb
numerator = rand(100)
denominator = rand(10)
$stderr.puts "Dividing #{numerator} by #{denominator}" if $DEBUG
puts numerator / denominator
```

When run with the `--debug` flag, the debug message is printed to standard error:

```
$ ./divide.rb --debug
Dividing 64 by 9
7

$ ./divide.rb --debug
Dividing 93 by 2
46

$ ./divide.rb --debug
Dividing 54 by 0
Exception `ZeroDivisionError' at divide_buggy.rb:6 - divided by 0
divide_buggy.rb:6:in `/': divided by 0 (ZeroDivisionError)
        from divide_buggy.rb:6
```

Once the bug is fixed, you can go back to running the script normally, and the debug message won't show up:

```
$ ./divide.rb
24
```

Discussion

This is a common technique when a "real" debugger is too much trouble. It's usually used to send debug messages to standard error, but you can put any code at all within a `$DEBUG` conditional. For instance, many Ruby libraries have their own "verbose", "debug level", or "debug mode" settings: you can choose to set these other variables appropriately only when `$DEBUG` is true.

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

```
require 'fileutils'
FileUtils.cp('source', 'destination', $DEBUG)
```

If your code is running deep within a framework, you may not have immediate access to the standard error stream of the process. You can always have your debug code write to a temporary logfile, and monitor the file.

Use of `$DEBUG` costs a little speed, but except in tight loops it's not noticeable. At the cost of a little more speed, you can save yourself some typing by defining convenience methods like this one:

```
def pdebug(str)
  $stderr.puts('DEBUG: ' + str) if $DEBUG
end

pdebug "Dividing #{numerator} by #{denominator}"
```

Once you've fixed the bug and you no longer need the debugging code, it's better to put it into a conditional than to simply remove it. If the problem recurs later, you'll find yourself adding the debugging code right back in.

Sometimes commenting out the debugging code is better than putting it into a conditional. It's more difficult to hunt down all the commented-out code, but you can pick and choose which pieces of code to uncomment. With the `$DEBUG` technique, it's all or nothing.

It doesn't *have* to be all or nothing, though. `$DEBUG` starts out a boolean but it doesn't have to stay that way: you can make it a numeric "debug level". Instead of doing something if $DEBUG, you can check whether `$DEBUG` is greater than a certain number. A very important piece of debug code might be associated with a debug level of 1; a relatively unused piece might have a debug level of 5. Setting `$DEBUG` to zero would turn off debugging altogether.

Here are some convenience methods that make it easy to use `$DEBUG` as either a boolean or a numeric value:

```
def debug(if_level)
  yield if ($DEBUG == true) || ($DEBUG && $DEBUG >= if_level)
end

def pdebug(str, if_level=1)
  debug(if_level) { $stderr.puts "DEBUG: " + str }
end
```

One final note: make sure that you put the `--debug` switch on the command line *before* the name of your Ruby script. It's an argument to the Ruby interpreter, nottoyour script.

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

See Also

- Recipe 17.5, "Adding Logging to Your Application," demonstrates a named system of debug levels; in fact, if your debug messages are mainly diagnostic, you might want to implement them as log messages

Recipe 17.2. Raising an Exception

Credit: Steve Arneil

Problem

An error has occurred and your code can't keep running. You want to indicate the error and let some other piece of code handle it.

Solution

Raise an exception by calling the `Kernel#raise` method with a description of the error. Calling the `raise` method interrupts the flow of execution.

The following method raises an exception whenever it's called. Its second message will never be printed:

```
def raise_exception
  puts 'I am before the raise.'
  raise 'An error has occurred.'
  puts 'I am after the raise.'
end

raise_exception
# I am before the raise.
# RuntimeError: An error has occurred
```

Discussion

Here's a method, `inverse`, that returns the inverse of a number `x`. It does some basic error checking by raising an exception unless `x` is a number:

```
def inverse(x)
  raise "Argument is not numeric" unless x.is_a? Numeric
  1.0 / x
end
```

When you pass in a reasonable value of `x`, all is well:

```
inverse(2)                                  # => 0.5
```


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

When `x` is not a number, the method raises an exception:

```
inverse('not a number')
# RuntimeError: Argument is not numeric
```

An exception is an object, and the `Kernel#raise` method creates an instance of an exception class. By default, `Kernel#raise` creates an exception of `RuntimeError` class, which is a subclass of `StandardError`. This in turn is a subclass of `Exception`, the superclass of all exception classes. You can list all the standard exception classes by starting a Ruby session and executing code like this:

```
ObjectSpace.each_object(Class) do |x|
  puts x if x.ancestors.member? Exception
end
```

This variant lists only the better-known exception classes:

```
ObjectSpace.each_object(Class) { |x| puts x if x.name =~ /Error$/ }
# SystemStackError
# LocalJumpError
# EOFError
# IOError
# RegexpError
# …
```

To raise an exception of a specific class, you can pass in the class name as an argument to `raise`. `RuntimeError` is kind of generic for the `inverse` method's check against `x`. The problem is there is actually a problem with one of the arguments passed into the method. A more aptly named exception class for that check would be `ArgumentError`:

```
def inverse(x)
  raise ArgumentError, 'Argument is not numeric' unless x.is_a? Numeric
  1.0 / x
end
```

To be even more specific about an error, you can define your own `Exception` subclass:

```
class NotInvertibleError < StandardError
end
```

The implementation of `inverse` method would then become:

```
def inverse(x)
  raise NotInvertibleError, 'Argument is not numeric' unless x.is_a? Numeric
  1.0 / x
end

inverse('not a number')
# NotInvertibleError: Argument is not numeric
```


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

In some other programming languages, exceptions are "thrown." In Ruby, they are not thrown but "raised." Ruby does have a `Kernel#throw` method, but it has nothing to do with exceptions. See Recipe 7.8 for an example of `throw`, as opposed to `raise`.

See Also

- Recipe 7.8, "Stopping an Iteration"
- Recipe 17.2, "Raising an Exception"
- Recipe 17.3, "Handling an Exception"

Recipe 17.3. Handling an Exception

Credit: Steve Arneil

Problem

You want to handle or recover from a raised exception.

Solution

Rescue the exception with a `begin/rescue` block. The code you put into the `rescue` clause should handle the exception and allow the program to continue executing.

This code demonstrates the `rescue` clause:

```
def raise_and_rescue
  begin
    puts 'I am before the raise.'
    raise 'An error has occurred.'
    puts 'I am after the raise.'
  rescue
    puts 'I am rescued!'
  end
  puts 'I am after the begin block.'
end

raise_and_rescue
# I am before the raise.
# I am rescued!
# I am after the begin block.
```

The exception doesn't stop the program from running to completion, but the code that was interrupted by the exception never gets run. Once the exception is handled, execution continues immediately after the `begin` block that spawned it.

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

Discussion

You can handle an exception with a `rescue` block if you know how to recover from the exception, if you want to display it in a nonstandard way, or if you know that the exception is not really a problem. You can solve the problem, present it to the end user, or just ignore it and forge ahead.

By default, a `rescue` clause rescues exceptions of class `StandardError` or its subclasses. Mentioning a specific class in a `rescue` statement will make it rescue exceptions of that class and its subclasses.

Here's a method, `do_it`, that calls the `Kernel#eval` method to run some Ruby code passed to it. If the code cannot be run (because it's not valid Ruby), `eval` raises an exception—a `SyntaxError`. This exception is not a subclass of `StandardError`; it's a subclass of `ScriptError`, which is a subclass of `Exception`.

```
def do_it(code)
  eval(code)
rescue
  puts "Cannot do it!"
end

do_it('puts 1 + 1')
# 2

do_it('puts 1 +')
# SyntaxError: (eval):1:in `do_it': compile error
```

That `rescue` block never gets called because `SyntaxError` is not a subclass of `StandardError`. We need to tell our `rescue` block to rescue us from `SyntaxError`, or else from one of its superclasses, `ScriptError` and `Exception`:

```
def do_it(code)
  eval(code)
rescue SyntaxError
  puts "Cannot do it!"
end

do_it('puts 1 +')
# Cannot do it!
```

You can stack `rescue` clauses in a `begin/rescue` block. Exceptions not handled by one `rescue` clause will trickle down to the next:

```
begin
 # …
rescue OneTypeOfException
 # …
rescue AnotherTypeOfException
 # …
end
```


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

If you want to interrogate a rescued exception, you can map the `Exception` object to a variable within the `rescue` clause. `Exception` objects have useful methods like `message` and `backtrace`:

```
begin
  raise 'A test exception.'
rescue Exception => e
  puts e.message
  puts e.backtrace.inspect
end
# ["(irb):33:in `irb_binding'",
# "/usr/lib/ruby/1.8/irb/workspace.rb:52:in `irb_binding'",
# ":0"]
```

You can also use the special variable $! within a `rescue` block to refer to the most recently raised `Exception`. If you do a `require 'English'`, you can use the `$ERROR_INFO` variable, which is easier to remember.

```
require 'English'
begin
  raise 'Another test exception.'
rescue Exception
  puts $!.message
  puts $ERROR_INFO.message
end
# Another test exception.
# Another test exception.
```

Since $! is a global variable, and might be changed at any time by another thread, it's safer to map each `Exception` object you rescue to an object.

See Also

- Recipe 17.2, "Raising an Exception"
- Recipe 17.4, "Rerunning After an Exception"

Recipe 17.4. Rerunning After an Exception

Credit: Steve Arneil

Problem

You want to rerun some code that raised an exception, having (hopefully) fixed the problem that caused it in the first place.

Solution

Retry the code that failed by executing a `retry` statement within a `rescue` clause of a code block. `retry` reruns the block from the beginning.

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

Here's a demonstration of the `retry` statement. The first time the code block runs, it raises an exception. The exception is `rescued`, the problem is "fixed," and the code runs to completion the second time:

```
def rescue_and_retry
  error_fixed = false
  begin
    puts 'I am before the raise in the begin block.'
    raise 'An error has occurred!' unless error_fixed
    puts 'I am after the raise in the begin block.'
  rescue
    puts 'An exception was thrown! Retrying…'
    error_fixed = true
    retry
  end
  puts 'I am after the begin block.'
end
rescue_and_retry
# I am before the raise in the begin block.
# An exception was thrown! Retrying…
# I am before the raise in the begin block.
# I am after the raise in the begin block.
# I am after the begin block.
```

Discussion

Here's a method, `check_connection`, that checks if you are connected to the Internet. It will try to connect to a `url` up to `max_tries` times. This method uses a `retry` clause to retry connecting until it successfully completes a connection, or until it runs out of tries:

```
require 'open-uri'

def check_connection(max_tries=2, url='http://www.ruby-lang.org/')
  tries = 0
  begin
    tries += 1
    puts 'Checking connection…'
    open(url) { puts 'Connection OK.' }
  rescue Exception
    puts 'Connection not OK!'
    retry unless tries >= max_tries
  end
end

check_connection
# Checking connection…
# Connection OK.

check_connection(2, 'http://this.is.a.fake.url/')
# Checking connection…
# Connection not OK!
# Checking connection…
# Connection not OK!
```

See Also

- Recipe 17.2, "Raising an Exception"
- Recipe 17.3, "Handling an Exception"

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

Recipe 17.5. Adding Logging to Your Application

Problem

You want to make your application log events or diagnostic data to a file or stream. You want verbose logging when your application is in development, and more taciturn logging when in production.

Solution

Use the `logger` library in the Ruby standard library. Use its `Logger` class to send logging data to a file or other output stream.

In most cases, you'll share a single `Logger` object throughout your application, as a global variable or module constant:

```
require 'logger'
$LOG = Logger.new($stderr)
```

You can then call the instance methods of `Logger` to send messages to the log at various levels of severity. From least to most severe, the instance methods are `Logger#debug`, `Logger#info`, `Logger#warn`, `Logger#error`, and `Logger#fatal`.

This code uses the application's logger to print a debugging message, and (at a higher severity) as part of error-handling code.

```
def divide(numerator, denominator)
  $LOG.debug("Numerator: #{numerator}, denominator #{denominator}")
  begin
    result = numerator / denominator
  rescue Exception => e
    $LOG.error "Error in division!: #{e}"
    result = nil
  end
  return result
end

divide(10, 2)
# D, [2006-03-31T19:35:01.043938 #18088] DEBUG -- : Numerator: 10, denominator 2
# => 5

divide(10, 0)
# D, [2006-03-31T19:35:01.045230 #18088] DEBUG -- : Numerator: 10, denominator 0
# E, [2006-03-31T19:35:01.045495 #18088] ERROR -- : Error in division!: divided by 0
# => nil
```

To change the log level, simply assign the appropriate constant to `level`:

```
$LOG.level = Logger::ERROR
```


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

Now our logger will ignore all log messages except those with severity `ERROR` or `FATAL`:

```
divide(10, 2)
# => 5

divide(10, 0)
# E, [2006-03-31T19:35:01.047861 #18088] ERROR -- : Error in division!: divided by 0
# => nil
```

Discussion

Ruby's standard logging system works like Java's oft-imitated Log4J. The `Logger` object centralizes all the decisions about whether a particular message is important enough to be written to the log. When you write code, you simply assume that all the messages will be logged. At runtime, you can get a more or a less verbose log by changing the log level. A production application usually has a log level of `Logger::INFO` or `Logger::WARN`.

The `DEBUG` log level is useful for step-by-step diagnostics of a complex task. The `ERROR` level is often used when handling exceptions: if the program can't solve a problem, it logs the exception rather than crash and expects a human administrator to deal with it. The `FATAL` level should only be used when the program cannot recover from a problem, and is about to crash or exit.

If your log is being stored in a file, you can have `Logger` rotate or replace the log file when it get too big, or once a certain amount of time has elapsed:

```
# Keep data for the current month only
Logger.new('this_month.log', 'monthly')

# Keep data for today and the past 20 days.
Logger.new('application.log', 20, 'daily')

# Start the log over whenever the log exceeds 100 megabytes in size.
Logger.new('application.log', 0, 100 * 1024 * 1024)
```

If the default log entries are too verbose for you, you have a couple of options. The simplest is to set `datetime_format` to a more concise date format. This code gets rid of the milliseconds:

```
$LOG.datetime_format = '%Y-%m-%d %H:%M:%S'
$LOG.error('This is a little shorter.')
# E, [2006-03-31T19:35:01#17339] ERROR -- : This is a little shorter.
```

If that's not enough for you, you can replace the `call` method that formats a message for the log:

```
class Logger
  class Formatter
    Format = "%s [%s] %s %s\n"
    def call(severity, time, progname, msg)
```


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

```
      Format % [severity, format_datetime(time), progname, msg]
    end
  end
end

$LOG.error('This is much shorter.')
# ERROR [2006-03-31T19:35:01.058646 ] This is much shorter.
```

See Also

- The standard library documentation for the `logger` library

Recipe 17.6. Creating and Understanding Tracebacks

Problem

You are debugging a program, and need to understand the stack traces that come with Ruby exceptions. Or you need to see which path the Ruby interpreter took to get to a certain line of code.

Solution

You can call the `Kernel#caller` method at any time to look at the Ruby interpreter's current call stack. The call stack is represented as a list of strings.

This Ruby program simulates a company with a top-down management style: one method delegates to another, which calls yet another. The method at the bottom can use `caller` to look upwards and see the methods that called it:

```
#!/usr/bin/ruby -w
# delegation.rb
class CEO
  def CEO.new_vision
    Manager.implement_vision
  end
end

class Manager
  def Manager.implement_vision
    Engineer.do_work
  end
end

class Engineer
  def Engineer.do_work
    puts 'How did I get here?'
    first = true
    caller.each do |c|
      puts %{#{(first ? 'I' : ' which')} was called by "#{c}"}
      first = false
    end
  end
end

CEO.new_vision
```


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

Running this program illustrates the path the interpreter takes to `Engineer.do_work`:

```
$ ./delegation.rb
How did I get here?
I was called by "delegation.rb:11:in `implement_vision'"
 which was called by "delegation.rb:5:in `new_vision'"
 which was called by "delegation.rb:26"
```

Discussion

Each string in a traceback shows which line of Ruby code made some method call. The first bit of the traceback given above shows that `Engineer.do_work` was called by `Manager.implement_vision` on line 11 of the program. The second line shows how `Manager.implement_vision` was called, and so on.

Remember the stack trace displayed when a Ruby script raises an exception? It's the same one you can get any time by calling `Kernel#caller`. In fact, if you `rescue` an exception and assign it to a variable, you can get its traceback as an array of strings— the equivalent of calling `caller` on the line that triggered the exception:

```
def raise_exception
  raise Exception, 'You wanted me to raise an exception, so…'
end

begin
  raise_exception
rescue Exception => e
  puts "Backtrace of the exception:\n #{e.backtrace.join("\n ")}"
end
# Backtrace of the exception:
# (irb):2:in `raise_exception'
# (irb):5:in `irb_binding'
# /usr/lib/ruby/1.8/irb/workspace.rb:52:in `irb_binding'
# :0
```

Note the slight differences between a backtrace generated from a Ruby script and one generated during an `irb` session.

If you've used languages like Python, you might long for "real" backtrace objects. About the best you can do is to parse the strings of a Ruby backtrace with a regular expression. The `parse_caller` method below extracts the files, lines, and method names from a Ruby backtrace. It works in both Ruby programs and `irb` sessions.

```
CALLER_RE = /(.*):([0-9]+)(:in \`(.*)')?/
def parse_caller(l)
  l.collect do |c|
    captures = CALLER_RE.match(c)
    [captures[1], captures[2], captures[4]]
  end
end

begin
  raise_exception
```


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

```
rescue Exception => e
  puts "Exception history:"
  first = true
  parse_caller(e.backtrace).each do |file, line, method|
    puts %{ #{first ? "L" : "because l"}ine #{line} in "#{file}"} +
         %{ called "#{method}" }
    first = false
   end
end
# Exception history:
#  Line 2 in "(irb)" called "raise_exception"
#  because line 24 in "(irb)" called "irb_binding"
#  because line 52 in "/usr/lib/ruby/1.8/irb/workspace.rb" called "irb_binding"
#  because line 0 in "" called ""
```

See Also

- Recipe 17.3, "Handling an Exception"

Recipe 17.7. Writing Unit Tests

Credit: Steve Arneil

Problem

You want to write some unit tests for your software, to guarantee its correctness now and in the future.

Solution

Use `Test::Unit`, the Ruby unit testing framework, from the Ruby standard library.

Consider a simple class for storing the name of a person. The `Person` class shown below stores a first name, a last name, and an age: a person's full name is available as a computed value. This code might go into a Ruby script called `app/person.rb`:

```
# app/person.rb
class Person
  attr_accessor :first_name, :last_name, :age

  def initialize(first_name, last_name, age)
    raise ArgumentError, "Invalid age: #{age}" unless age > 0
    @first_name, @last_name, @age = first_name, last_name, age
  end

  def full_name
    first_name + ' ' + last_name
  end
end
```

Now, let's write some unit tests for this class. By convention, these would go into the file `test/person_test.rb`.

First, require the `Person` class itself and the `Test::Unit` framework:

```
# test/person_test.rb
require File.join(File.dirname(__FILE__), '..', 'app', 'person')
require 'test/unit'
```

Next, extend the framework class `Test::Unit::TestCase` with a class to contain the actual tests. Each test should be written as a method of the test class, and each test method should begin with the prefix `test`. Each test should make one or more *assertions*: statements about the code which must be true for the code to be correct. Below are three test methods, each making one assertion:

```
class PersonTest < Test::Unit::TestCase
  def test_first_name
    person = Person.new('Nathaniel', 'Talbott', 25)
    assert_equal 'Nathaniel', person.first_name
  end

  def test_last_name
    person = Person.new('Nathaniel', 'Talbott', 25)
    assert_equal 'Talbott', person.last_name
  end

  def test_full_name
    person = Person.new('Nathaniel', 'Talbott', 25)
    assert_equal 'Nathaniel Talbott', person.full_name
  end

  def test_age person =
    Person.new('Nathaniel', 'Talbott', 25)
    assert_equal 25, person.age
    assert_raise(ArgumentError) { Person.new('Nathaniel', 'Talbott', -4) }
    assert_raise(ArgumentError) { Person.new('Nathaniel', 'Talbott', 'four') }
  end
end
```

This code is somewhat redundant; see below for a way to fix that issue. For now, let's run our four tests, by running `person_test.rb` as a script:

```
$ ruby test/person_test.rb
Loaded suite test/person_test
Started
….
Finished in 0.008837 seconds.

4 tests, 6 assertions, 0 failures, 0 errors
```

Great! All the tests passed.

Discussion

The `PersonTest` class defined above works, but it's got some redundant and inefficient code. Each of the four tests starts by creating a `Person` object, but they could all share the same `Person` object. The `test_age` method needs to create some additional, invalid `Person` objects to verify the error checking, but there's no reason why it can't share the same "normal" `Person` object as the other three test methods.

`Test::Unit` makes it possible to refactor shareable code into a method named `setup`. If a test class has a `setup` method, it will be called before any of the assertion methods. Conversely, any clean-up code that is required *after* each test method runs can be placed in a method named `teardown`.

Here's a new implementation of `PersonTest` that uses `setup` and class constants to remove the duplicate code:

```
# person2.rb
require File.join(File.dirname(__FILE__), '..', 'app', 'person')
require 'test/unit'

class PersonTest < Test::Unit::TestCase
  FIRST_NAME, LAST_NAME, AGE = 'Nathaniel', 'Talbott', 25

  def setup
    @person = Person.new(FIRST_NAME, LAST_NAME, AGE)
  end

  def test_first_name
    assert_equal FIRST_NAME, @person.first_name
  end

  def test_last_name
    assert_equal LAST_NAME, @person.last_name
  end

  def test_full_name
    assert_equal FIRST_NAME + ' ' + LAST_NAME, @person.full_name
  end

  def test_age
    assert_equal 25, @person.age
    assert_raise(ArgumentError) { Person.new(FIRST_NAME, LAST_NAME, -4) }
    assert_raise(ArgumentError) { Person.new(FIRST_NAME, LAST_NAME, 'four') }
  end
end
```

There are lots of assertion methods besides the `assert_equal` and `assert_raise` method used in the test classes above: `assert_not_equal, assert_nil`, and more exotic methods like `assert_respond_to`. All the assertion methods are defined in the `Test::Unit::Assertions` module, which is mixed into the `Test::Unit::TestCase` class.

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

The simplest assertion method is just plain `assert`. It causes the test method to fail unless it's passed a value other than `false` or `nil`:

```
def test_first_name
  assert(FIRST_NAME == @person.first_name)
end
```

`assert` is the most basic assertion method. All the other assertion methods can be defined in terms of it:

```
def assert_equal(expected, actual)
  assert(expected == actual)
end
```

So, if you can't decide (or remember) which particular assertion method to use, you can always use `assert`.

See Also

- `ri Test::Unit`
- The documentation for the `Test::Unit` library is also online at http://www.ruby-doc.org/stdlib/libdoc/test/unit/rdoc/index.html
- Recipe 15.22, "Unit Testing Your Web Site"
- Recipe 17.8, "Running Unit Tests"
- Recipe 19.1, "Automatically Running Unit Tests"

Recipe 17.8. Running Unit Tests

Credit: Steve Arneil

Problem

You want to run some or all of the unit tests you've written.

Solution

This solution uses the example test class `PersonTest` from the previous recipe, Recipe 17.7. In that scenario, this code lives in a file `test/person_test.rb`, and the code to be tested lives in `app/person.rb`. Here's `test/person_test.rb` again:

```
# person_test.rb
require File.join(File.dirname(__FILE__), '..', 'app', 'person')
require 'test/unit'

class PersonTest < Test::Unit::TestCase
  FIRST_NAME, LAST_NAME, AGE = 'Nathaniel', 'Talbott', 25
```

```
  def setup
    @person = Person.new(FIRST_NAME, LAST_NAME, AGE)
  end

  def test_first_name
    assert_equal FIRST_NAME, @person.first_name
  end

  def test_last_name
    assert_equal LAST_NAME, @person.last_name
  end

  def test_full_name
    assert_equal FIRST_NAME + ' ' + LAST_NAME, @person.full_name
  end
  def test_age
    assert_equal 25, @person.age
    assert_raise(ArgumentError) { Person.new(FIRST_NAME, LAST_NAME, -4) }
    assert_raise(ArgumentError) { Person.new(FIRST_NAME, LAST_NAME, 'four') }
  end
end
```

As seen in the previous recipe, the simplest solution is to run the script that contains the tests as a Ruby script:

```
$ ruby test/person_test.rb
Loaded suite test/person_test
Started
….
Finished in 0.008955 seconds.

4 tests, 6 assertions, 0 failures, 0 errors
```

But the `person_test.rb` script also accepts command-line arguments. You can use the `--name` option to choose which test methods to run, and the `--verbose` option to print each test method as it's run:

```
$ ruby test/person_test.rb --verbose --name test_first_name \
  --name test_last_name
Loaded suite test/person_test
Started
test_first_name(PersonTest): .
test_last_name(PersonTest): .

Finished in 0.012567 seconds.

2 tests, 2 assertions, 0 failures, 0 errors
```

Discussion

How do the tests run when `person_test.rb` doesn't appear to do anything but define a class? How can `person_test.rb` accept command-line arguments? We wrote that file, and we didn't put in any command-line parsing code.

It all happens behind the scenes. When we required the `Test::Unit` framework, it passed a block into the method method `Kernel#at_exit`. This block is guaranteed to be called before the Ruby interpreter exits. It looks like this:

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

```
$ tail -5 /usr/local/lib/ruby/1.8/test/unit.rb
at_exit do
  unless $! || Test::Unit.run?
    exit Test::Unit::AutoRunner.run
  end
end
```

Once the code in `person_test.rb` defines its test class, the Ruby interpreter exits: but first, it runs that block, which triggers the `AutoRunner` test runner. This does the command-line parsing, the execution of the tests in `PersonTest`, and all the rest of it.

Here are a few more helpful options to a unit test script.

The `--name` option can be used with a regular expression to choose the test methods to run.

```
$ ruby test/person_test.rb --verbose --name '/test_f/'
Loaded suite test/person_test
Started
test_first_name(PersonTest): .
test_full_name(PersonTest): .

Finished in 0.014891 seconds.

2 tests, 2 assertions, 0 failures, 0 errors
```

The `Test::Unit` framework can be also be loaded alone to run tests in the current directory and its subdirectories. Use the `--pattern` option with a regular expression to select the test files to run:

```
$ ruby -rtest/unit -e0 -- --pattern '/_test/'
Loaded suite .
Started
…
Finished in 0.009329 seconds.

4 tests, 6 assertions, 0 failures, 0 errors
```

To list all the available `Test::Unit` options, use the `--help` option:

```
$ ruby test/person_test.rb --help
```

Additional options are available when the `Test::Unit` framework is run standalone. Again, use the `--help` option:

```
$ ruby -rtest/unit -e0 -- --help
```

See Also

- `ri Test::Unit`
- Recipe 15.22, "Unit Testing Your Web Site"
- Recipe 17.7, "Writing Unit Tests"
- Recipe 19.1, "Automatically Running Unit Tests"

Recipe 17.9. Testing Code That Uses External Resources

Credit: John-Mason Shackelford

Problem

You want to test code without triggering its real-world side effects. For instance, you want to test a piece of code that makes an expensive network connection, or irreversibly modifies a file.

Solution

Sometimes you can set up an alternate data source to use for testing (Rails does this for the application database), but doing that makes your tests slower and imposes a setup burden on other developers. Instead, you can use Jim Weirich's FlexMock library, available as the `flexmock` gem.

Here's some code that performs a destructive operation on a live data source:

```
class VersionControlMaintenance

  DAY_SECONDS = 60 * 60 * 24

  def initialize(vcs)
    @vcs = vcs
  end

  def purge_old_labels(age_in_days)
    @vcs.connect
    old_labels = @vcs.label_list.select do |label|
      label['date'] <= Time.now - age_in_days * DAY_SECONDS
    end
    @vcs.label_delete(*old_labels.collect{|label| label['name']})
    @vcs.disconnect
  end
end
```

This code would be difficult to test by conventional means, with the `vcs` variable pointing to a live version control repository. But with FlexMock, it's simple to define a mock `vcs` object that can impersonate a real one.

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

Here's a unit test for `VersionControlMaintenance#purge_old_labels` that uses Flex-Mock, instead of modifying a real version control repository. First, we set up some dummy labels:

```
require 'rubygems'
require 'flexmock'
require 'test/unit'

class VersionControlMaintenanceTest < Test::Unit::TestCase

  DAY_SECONDS = 60 * 60 * 24
  LONG_AGO = Time.now - DAY_SECONDS * 3
  RECENT = Time.now - DAY_SECONDS * 1
  LABEL_LIST = [
               { 'name' => 'L1', 'date' => LONG_AGO },
               { 'name' => 'L2', 'date' => RECENT }
              ]
```

We use FlexMock to define an object that expects a certain series of method calls:

```
def test_purge
  FlexMock.use("vcs") do |vcs|
    vcs.should_receive(:connect).with_no_args.once.ordered
    vcs.should_receive(:label_list).with_no_args.
          and_return(LABEL_LIST).once.ordered

    vcs.should_receive(:label_delete).
          with('L1').once.ordered

    vcs.should_receive(:disconnect).with_no_args.once.ordered
```

Then we pass our mock object into the class we want to test, and call `purge_old_labels` normally:

```
      v = VersionControlMaintenance.new(vcs)
      v.purge_old_labels(2)

      # The mock calls will be automatically varified as we exit the
      # @FlexMock.use@ block.
    end
  end
end
```

Discussion

FlexMock lets you script the behavior of an object so that it acts like the object you don't want to actually call. To set up a mock object, call `FlexMock.use`, passing in a textual label for the mock object, and a code block. Within the code block, call `should_receive` to tell the mock object to expect a call to a certain method.

You can then call `with` to specify the arguments the mock object should expect on that method call, and call `and_returns` to specify the return value. A call to `#once` indicates that the tested code should call the method only one time, and `#ordered` indicates that the tested code must call these mock methods in the order in which they are defined.

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

After the code block is executed, FlexMock verifies that the mock object's expectations were met. If they weren't (the methods weren't called in the right order, or they were called with the wrong arguments), it raises a `TestFailedError` as any `Test::Unit` assertion would.

The example above tells Ruby how we expect `purge_old_labels` to work. It should call the version control system's `connect` method, and then `label_list`. When this happens, the mock object returns some dummy labels. The code being tested is then expected to call `label_delete` with "L1" as the sole parameter.

This is the crucial point of this test. If `purge_old_labels` is broken, it might decide to pass both "L1" and "L2" into `label_delete` (even though "L2" is too recent a label to be deleted). Or it might decide not to call `label_delete` at all (even though "L1" is an old label that ought to be deleted). Either way, FlexMock will notice that `purge_old_labels` did not behave as expected, and the test will fail. This works without you having to write any explicit `Test::Unit` assertions.

`FlexMock` lives up to its name. Not only can you tell a mock object to expect a given method call is expected once and only once, you have a number of other options, summarized in Tables 17-1 and 17-2.

Table 17-1. From the RDoc

Specifier	Meaning	Modifiers allowed?
`zero_or_more_times`	Declares that the message may be sent zero or more times (default, equivalent to `at_least.never`)	No
`once`	Declares that the message is only sent once	Yes
`twice`	Declares that the message is only sent twice	Yes
`never`	Declares that the message is never sent	Yes
`times(n)`	Declares that the message is sent n times	Yes

Table 17-2. From the RDoc

Modifier	Meaning
`at_least`	Modifies the immediately following message count declarator to mean that the message must be sent at least that number of times; for instance, `at_least.once` means that the message is expected at least once but may be sent more than once
`at_most`	Similar to `at_least`, but puts an upper limit on the number of messages

Both the `at_least` and `at_most` modifiers may be specified on the same expectation.

Besides listing a mock method's expected parameters using `with(arglist)`, you can also use `with_any_args` (the default) and `with_no_args`. With

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

`should_ignore_missing`, you can indicate that it's okay for the tested code to call methods that you didn't explicitly define on the mock object. The mock object will respond to the undefnied method, and return `nil`.

Especially handy is `FlexMock's` support for specifying return values as a block. This allows us to simulate an exception, or complex behavior on repeated invocations.

```
# Simulate an exception in the mocked object.
mock.should_receive(:connect).and_return{ raise ConnectionFailed.new }

# Simulate a spotty connection: the first attempt fails
# but when the exception handler retries, we connect.
i = 0
mock.should_receive(:connect).twice.
  and_return{ i += 1; raise ConnectionFailed.new unless i > 1 }
end
```

Test-driven development usually produces a design that makes it easy to substitute mock objects for external dependencies. But occasionally, circumstances call for special magic. In such cases Jim Weirich's `class_intercepter.rb` is a welcome ally.

The class below instantiates an object which connects to an external data source. We can't touch this data source when we're testing the code.

```
class ChangeHistoryReport
  def date_range(label1, label2)
    vc = VersionControl.new
    vc.connect
    dates = [label1, label2].collect do |label|
      vc.fetch_label(label).files.sort_by{|f|f['date']}.last['date']
    end
    vc.disconnect
    return dates
  end
end
```

How can we test this code? We could refactor it—introduce a factory or a dependency injection scheme. Then we could substitute in a mock object (although in this case, we'd simply move the complex operations to another method). But if we are sure we "aren't going to need it" (as the saying goes) and since we are programming in Ruby and not a less flexible language, we can test the code as is.

As before, we call `FlexMock.use` to define a mock object:

```
require 'class_intercepter'
require 'test/unit'
class ChangeHistoryReportTest < Test::Unit::TestCase
  def test_date_range
      FlexMock.use('vc') do |vc|
      # initialize the mock
      vc.should_receive(:connect).once.ordered
      vc.should_receive(:fetch_label).with(LABEL1).once.ordered
      vc.should_receive(:fetch_label).with(LABEL2).once.ordered
```


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

```
        vc.should_receive(:disconnect).once.ordered
        vc.should_receive(:new).and_return(vc)
```

Here's the twist: we reach into the `ChangeHistoryReport` class and tell it to use our mock class whenever it wants to use the `VersionControl` class:

```
ChangeHistoryReport.use_class(:VersionControl, vc) do
```

Now we can use a `ChangeHistoryReport` object without worrying that it will operate against any real version control repository. As before, the FlexMock framework takes care of making the actual assertions.

```
        c = ChangeHistoryReport.new
        c.date_range(LABEL1, LABEL2)
        end
      end
    end
end
```

See Also

- The `FlexMock` generated RDoc (http://onestepback.org/software/flexmock/)
- `class_intercepter.rb` (http://onestepback.org/articles/depinj/ci/class_intercepter_rb.html)
- Alternatives to FlexMock include RSpec (http://rspec.rubyforge.org/) and Test::Unit::Mock (http://www.deveiate.org/projects/Test-Unit-Mock/)
- Jim Weirich's presentation on Dependency Injection is closely related to testing with mock objects (http://onestepback.org/articles/depinj/)
- Kent Beck's classic *Test Driven Development: By Example* (Addison-Wesley) is a must read; even the seasoned TD developer will benefit from Kent's helpful patterns section at the back of the book

Recipe 17.10. Using breakpoint to Inspect and Change the State of Your Application

Problem

You're debugging an application, and would like to be able to stop the program at any point and inspect the application's state (variables, data structures, etc.). You'd also like to be able to modify the application's state before restarting it.

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

Solution

Use the breakpoint library, available as the `ruby-breakpoint` gem.

Once you `require 'breakpoint'`, you can call the `breakpoint` method from anywhere in your application. When the execution hits the `breakpoint` call, the application turns into an interactive Ruby session.

Here's a short Ruby program:

```
#!/usr/bin/ruby -w
# breakpoint_test.rb
require 'rubygems'
require 'breakpoint'

class Foo
  def initialize(init_value)
    @instance_var = init_value
  end

  def bar
    test_var = @instance_var
    puts 'About to hit the breakpoint!'
    breakpoint
    puts 'HERE ARE SOME VARIABLES:'
    puts "test_var: #{test_var}, @instance_var: #{@instance_var}"
  end
end

f = Foo.new('When in the course')
f.bar
```

When you run the application, you quickly hit the call to `breakpoint` in `Foo#bar`. This drops you into an `irb` session:

```
$ ruby breakpoint_test.rb
About to hit the breakpoint!
Executing break point at breakpoint_test.rb:14 in `bar'
irb(#<Foo:0xb7452464>):001:0>
```

Once you quit the `irb` session, the program continues on its way:

```
irb(#<Foo:0xb7452a18>):001:0> quit
HERE ARE SOME VARIABLES:
test_var: When in the course, @instance_var: When in the course
```

But there's a lot you can do within that `irb` session before you quit. You can look at the array `local_variables`, which enumerates all variables local to the current method. You can also look at and modify any of the variables that are currently in scope, including instance variables, class variables, and globals:

```
$ ruby breakpoint_test.rb
About to hit the breakpoint!
Executing break point at breakpoint_test.rb:14 in `bar'
```


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

```
irb(#<Foo:0xb7452464>):001:0> local_variables
=> ["test_var", "_"]
irb(#<Foo:0xb7452428>):002:0> test_var
=> "When in the course"
irb(#<Foo:0xb7452428>):003:0> @instance_var
=> "When in the course"
irb(#<Foo:0xb7452428>):004:0> @instance_var = 'of human events'
=> "of human events"
```

As before, once you quit the `irb` session, the program continues running:

```
irb(#<Foo:0xb7452428>):005:0> quit
HERE ARE SOME VARIABLES:
test_var: When in the course, @instance_var: of human events
```

Because we changed the variable `@instance_variable` within our breakpoint, the `puts` in the program reports the new value after we leave the breakpoint session.

Discussion

There is another way to access a breakpoint. Instead of calling breakpoint directly, you can pass a code block into `assert`. If the block evaluates to false, `assert` executes a breakpoint. Let's say you want to execute a breakpoint only if the instance variable `@instance_variable` has a certain value. Here's how:

```
#!/usr/bin/ruby -w
# breakpoint_test_2.rb
require 'rubygems'
require 'breakpoint'

class Foo
  def initialize(init_value)
    @instance_var = init_value
  end

  def bar
    test_var = @instance_var
    puts 'About to hit the breakpoint! (maybe)'
    assert { @instance_var == 'This is another fine mess' }
    puts 'HERE ARE SOME VARIABLES:'
    puts "test_var: #{test_var}, @instance_var: #{@instance_var}"
  end
end

Foo.new('When in the course').bar             # This will NOT cause a breakpoint
Foo.new('This is another fine mess').bar      # This will NOT cause a breakpoint

$ ruby breakpoint_test_2.rb
About to hit the breakpoint! (maybe)
HERE ARE SOME VARIABLES:
test_var: When in the course, @instance_var: When in the course
About to hit the breakpoint! (maybe)
Assert failed at breakpoint_test_2.rb:14 in `bar'. Executing implicit breakpoint.
irb(#<Foo:0xb7452450>):001:0> @instance_var
=> "This is another fine mess"
irb(#<Foo:0xb7452450>):002:0> quit
HERE ARE SOME VARIABLES:
test_var: This is another fine mess, @instance_var: This is another fine mess
```


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

By using `assert`, you can enter an interactive `irb` session only when the state of your application is worth inspecting.

Recipe 17.11. Documenting Your Application

Problem

You want to create a set of API documentation for your application. You might want to go so far as to keep all your documentation in the same files as your source code.

Solution

It's good programming practice to preface each of your methods, classes, and modules with a comment that lets the reader know what's going on. Ruby rewards this behavior by making it easy to transform those comments into a set of HTML pages that document your code. This is similar to Java's JavaDoc, Python's PyDoc, and Perl's Pod.

Here's a simple example. Suppose your application contains only one file, `sum.rb`, which defines only one method:

```
def sum(*terms)
  terms.inject(0) { |sum, term| sum + term}
end
```

To document this application, use Ruby comments to document the method, and also to document the file as a whole:

```
# Just a simple file that defines a sum method.

# Takes any number of numeric terms and returns the sum.
#   sum(1, 2, 3)                            # => 6
#   sum(1, -1, 10)                          # => 10
#   sum(1.5, 0.2, 0.3, 1)                   # => 3.0
def sum(*terms)
  terms.inject(0) { |sum, term| sum + term}
end
```

Change into the directory containing the `sum.rb` file, and run the `rdoc` command.

```
$ rdoc
sum.rb: .
Generating HTML…

Files: 1
Classes: 0
Modules: 0
Methods: 1
Elapsed: 0.101s
```

The `rdoc` command creates a `doc/` subdirectory beneath the current directory. It parses every Ruby file it can find in or below the current directory, and generates HTML files from the Ruby code and the comments that document it.

The `index.html` file in the `doc/` subdirectory is a frameset that lets users navigate the files of your application. Since the example only uses one file (`sum.rb`), the most interesting thing about its generated documentation is what RDoc has done with the comments (Figure 17-1).

Discussion

RDoc parses a set of Ruby files, cross-references them, and generates a web site that captures the class and module structure, and the comments you wrote while you were coding.

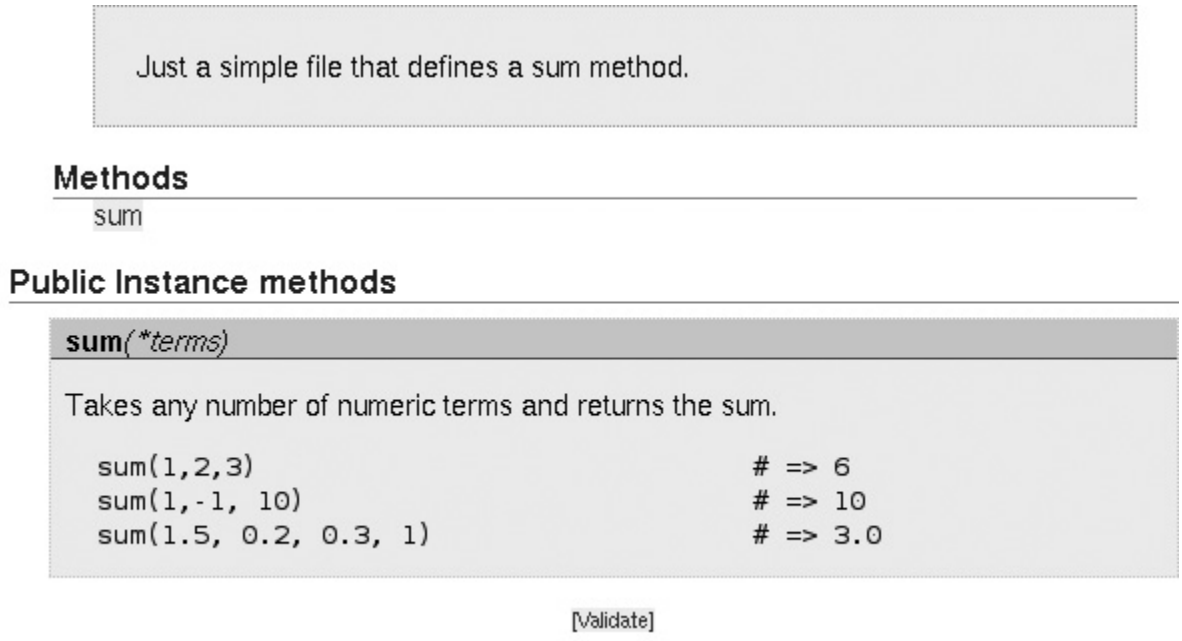
Generated RDoc makes for a useful reference to your classes and methods, but it's not a substitute for handwritten examples or tutorials. Of course, RDoc comments can *contain* handwritten examples or tutorials. This will help your users and also help you keep your documentation together with your code.

Notice that when I wrote examples for the `sum` method, I indented them a little from the text above them:

```
# Takes any number of numeric terms and returns the sum.
#   sum(1, 2, 3)                          # => 6
```


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

Figure 17-1. RDoc comments

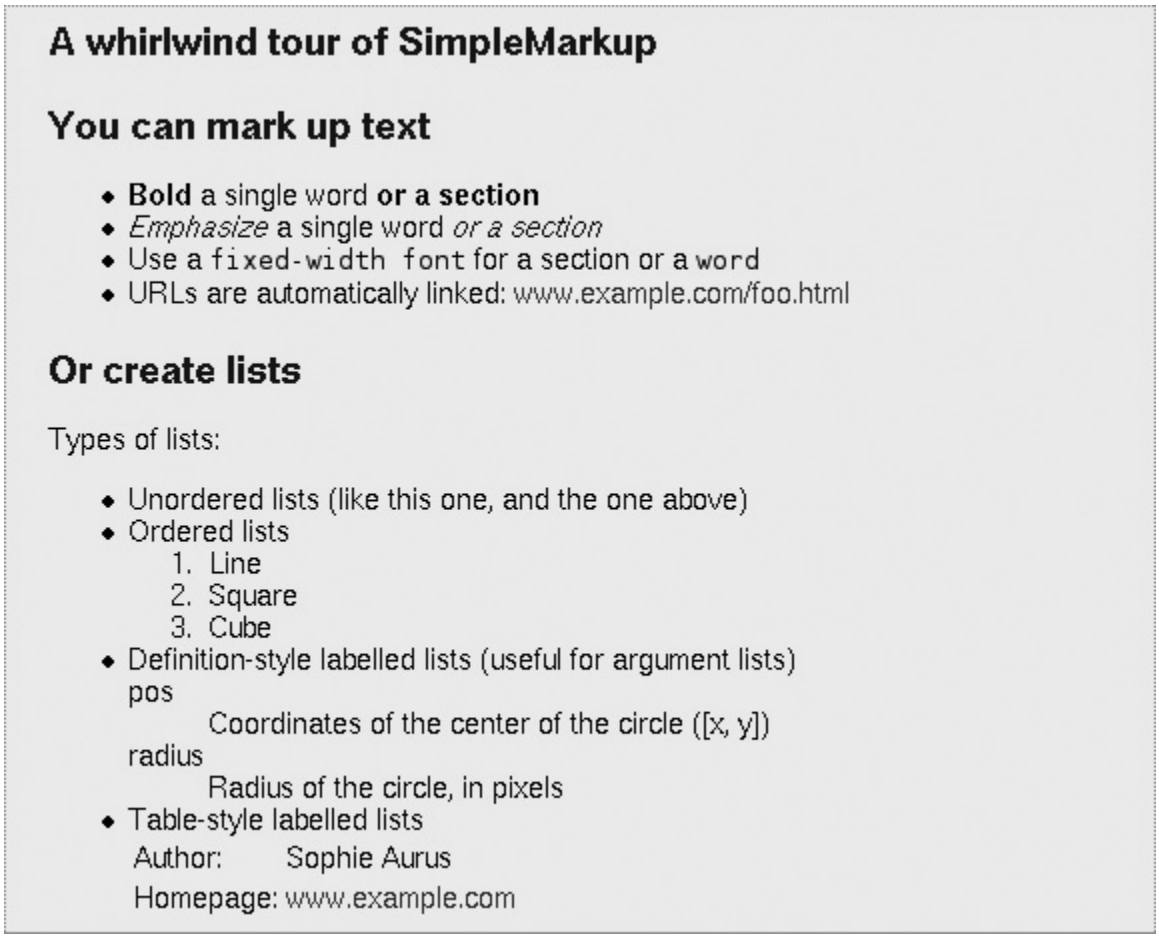


RDoc picked up on this extra indentation and displayed my examples as Ruby code, in a fixed-width font. This is one of many RDoc conventions for improving the looks of the rendered HTML. As with wiki markup, the goal of the RDoc conventions is to allow text to render nicely as HTML while being easy to read and edit as plain text (Figure 17-2).

```
# =A whirlwind tour of SimpleMarkup
#
# ==You can mark up text
#
# * *Bold* a single word <b>or a section</b>
# * _Emphasize_ a single word <i>or a section</i>
# * Use a <tt>fixed-width font</tt> for a section or a +word+
# * URLs are automatically linked: https://www.example.com/foo.html
#
# ==Or create lists
#
# Types of lists:
# * Unordered lists (like this one, and the one above)
# * Ordered lists
#   1. Line
#   2. Square
#   3. Cube
# * Definition-style labelled lists (useful for argument lists)
#   [pos] Coordinates of the center of the circle ([x, y])
#   [radius] Radius of the circle, in pixels
# * Table-style labelled lists
#   Author:: Sophie Aurus
#   Homepage:: http://www.example.com
```


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

Figure 17-2. Plain text



There are also several special RDoc directives that go into comments on the same line as a method, class, or module definition. The most common is `:nodoc:`, which is used if you want to hide something from RDoc. You can and should put an RDoc-style comment even on a `:nodoc:` method or class, so that people reading your Ruby code will know what it does.

```
# This class and its contents are hidden from RDoc; here's what it does:
# …
#
class HiddenClass # :nodoc:
  # …
end
```

Private methods don't show up in RDoc generated documentation—that would usually just mean clutter. If you want one particular private method to show up in the documentation (probably for the benefit of people subclassing your class), use the `:doc:` directive; it's the opposite of the `:nodoc:` directive: [1]

[1] If you want all private methods to show up in the documentation, pass the `--all` argument to the `rdoc` command. The `rdoc` command supports many command-line arguments, giving you control over the rules for generating the documentation and the layout of the results.

```
class MyClass
  private

  def hidden_method
  end

  def visible_method # :doc:
  end
end
```

If a comment mentions another class, method, or source file, RDoc will try to locate and turn it into a hyperlinked cross-reference. To indicate that a method name is a method name and not just a random word, prefix it with a hash symbol or use its fully qualified name (`MyClass.class_method` or `MyClass#instance_method`:

```
# The SimplePolynomial class represents polynomials in one variable
# and can perform most common operations on them.
#
# See especially #solve and #derivative. For multivariate polynomials,
# see MultivariatePolynomial (especially
# MultivariatePolynomial#simplify, which may return a
# SimplePolynomial), and much of calculus.rb.
```

Other ways of creating RDoc

The Ruby gem installation process generates a set of RDoc files for every gem it installs. If you package your software as a gem, anyone who installs it will automatically get the RDoc files as well.

You can also create RDoc files programatically from a Ruby program, by creating and scripting `RDoc` objects. The `rdoc` command itself is nothing more than Ruby code such as the following, along with some error handling:

```
#!/usr/bin/ruby
# rdoc.rb
require 'rdoc/rdoc'
RDoc::RDoc.new.document(ARGV)
```

See Also

- Recipe 18.5, "Reading Documentation for Installed Gems"
- The RDoc documentation covers all the markup conventions and directives in detail (http://rdoc.sourceforge.net/doc/)
- http://rdoc.sourceforge.net/doc/files/markup/simple_markup_rb.html

Recipe 17.12. Profiling Your Application

Problem

You want to find the slowest parts of your application, and speed them up.

Solution

Include the Ruby profiler in your application with `include 'profile'` and the profiler will start tracking and timing every subsequent method call. When the application exits, the profiler will print a report to your program's standard error stream.

Here's a program that contains a performance flaw:

```
#!/usr/bin/env ruby
# sequence_counter.rb
require 'profile'

total = 0
# Count the letter sequences containing an a, b, or c.
('a'..'zz').each do |seq|
  ['a', 'b', 'c'].each do |i|
    if seq.index(i)
      total += 1
      break
    end
  end
end
puts "Total: #{total}"
```

When the program is run, the profiler shows the parts of the program that are most important to optimize:

```
$ ruby sequence_counter.rb
Total: 150
  %    cumulative   self              self      total
 time    seconds   seconds    calls   ms/call   ms/call  name
 54.55      0.30      0.30      702      0.43      0.50  Array#each
 32.73      0.48      0.18        1    180.00    550.00  Range#each
  7.27      0.52      0.04     1952      0.02      0.02  String#index
  3.64      0.54      0.02      702      0.03      0.03  String#succ
  1.82      0.55      0.01      150      0.07      0.07  Fixnum#+
…
```

The program takes about 0.3 seconds to run, and most of that is spent in `Array#each`. What if we replaced that code with an equivalent regular expression?

```
#!/usr/bin/env ruby
# sequence_counter2.rb
require 'profile'

total = 0
# Count the letter sequences containing an a, b, or c.
('a'..'zz').each {|seq| total +=1 if seq =~ /[abc]/ }
puts "Total: #{total}"
```


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

Running this program yields a much better result:

```
$ ruby sequence_counter2.rb
Total: 150
  %    cumulative       self              self      total
 time    seconds       seconds   calls    ms/call   ms/call   name
 83.33      0.05          0.05       1      50.00     60.00   Range#each
 16.67      0.06          0.01     150       0.07      0.07   Fixnum#+
  0.00      0.06          0.00       1       0.00      0.00   Fixnum#to_s
…
```

The new version takes only 0.05 seconds to run, and as near as the profiler can measure, it's running nearly as fast as an empty iterator over the range `'a'..'zz'`.

Discussion

You might think that `regex_counter2.rb` has a performance problem of its own. After all, it initializes the regular expression `/[abc]/` within a loop, which seems to indicate that it's being initialized multiple times. The natural instinct of the optimizing programmer is to move that definition outside the loop; surely that would be more efficient.

```
re = /[abc]/
('a'..'zz').each {|seq| total +=1 if seq =~ re }
```

But it's not (try it!). The profiler actually shows a *decrease* in performance when the regular expression is assigned to a variable outside the loop. The Ruby interpreter is doing some optimization behind the scenes, and the code with an "obvious" performance problem beats the more complex "optimized" version.[2] There is a general lesson here: the problem is often not where you think it is, and empirical data always beats guesswork.

[2] Of course, a regular expression is a pretty simple object. If you've got a loop that builds a million-element data structure, or reads the same file over and over, the Ruby interpreter can't help you. Move that sucker out of the loop. If you make this kind of mistake, it'll show up in the profiler.

Ruby's profiler is a fairly blunt tool (it's written in only about 60 lines of Ruby), and to instrument it for anything but a simple command-line application, you'll need to do some work. It helps if your code has unit tests, because profiler tests require a lot of the same scaffolding as unit tests. You can even build up a library of profiler test scripts to go with your unit tests, although the profiler output is difficult to analyze automatically.

If you know that some particular operation is slow, you can write code that stresstests that operation (the way you might write a unit test), and run only that code with the profiler. To stress-test `sequence_counter2.rb`, you might change it to operate on a larger range like (`'a'..'zzzz'`). Big datasets make performance problems more visible.

If you don't know which operations are slow, pick the most common operations and instrument them on large datasets. If you're writing an XML library, write a profiler script that loads and parses an enormous file, and one that turns an enormous data structure

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

into XML. If you've got no ideas at all, run the profiler on your unit test suite and look for problems. The tests that run slowly may be exercising problematic parts of your program.

The profiler results are ordered with the most time-consuming method calls first. To optimize your code, go from the top of the profiler results and address each call in turn. See why your script led to so many calls of that method, and what you can do about it. Either change the underlying code path so it doesn't call that method so many times, or optimize the method itself. If the method is one you wrote, you can optimize it by profiling it in isolation.

The timing data given by the profiler isn't terribly accurate,[3] but it should be good enough to find problem areas. If you want a more reliable estimate of how long some code takes to run, try the `benchmark` library, or run your script using the Unix `time` command.

[3] Note the timing inconsistencies in the examples above. Somehow the entire original `sequence_counter.rb` runs in 0.30 seconds, but when you ignore all the `Array#each` calls, the cumulative time jumps up to 0.48 seconds.

The Ruby profiler sets the interpreter's trace function (by passing a code block into `Kernel#set_trace_func`), so if your program uses a trace function of its own, using the profiler will overwrite the old function. This probably won't affect you, because the trace function is mainly used by profilers and other analysis tools.

See Also

If the profiler says your problem is in a commonly-called method like `Array#each`, you need to somehow figure out which calls to the method are the problematic ones; see Recipe 17.15, "Who's Calling That Method? A Call Graph Analyzer"

Recipe 17.13. Benchmarking Competing Solutions

Problem

You want to see which of two solutions to a problem is faster. You might want to compare two different algorithms, or two libraries that do the same thing.

Solution

Use the `benchmark` library to time the tasks you want to run. The `Benchmark.bm` method gives you an object that can report on how long it takes for code blocks to run.

Let's explore whether the `member?` method is faster on arrays or hashes. First, we create a large array and a large hash with the same data, and define a method that exercises the `member?` method:

```
RANGE = (0..1000)
```


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

```
array = RANGE.to_a
hash = RANGE.inject({}) { |h,i| h[i] = true; h }

def test_member?(data)
  RANGE.each { |i| data.member? i }
end
```

Next, we call `Benchmark.bm` to set up a series of timing tests. The first test calls `test_member?` on the array; the second one calls it on the hash. The results are printed in a tabular form to standard error:

```
require 'benchmark'

Benchmark.bm(5) do |timer|
  timer.report('Array') { test_member?(array) }
  timer.report('Hash') { test_member?(hash) }
end
#        user      system       total        real
# Array  0.260000    0.060000    0.320000 (  0.332583)
# Hash   0.010000    0.000000    0.010000 (  0.001242)
```

As you'd expect, `member?` is much faster on a hash.

Discussion

What do the different times mean? The `real` time is "wall clock" time: the number of seconds that passed in the real world between the start of the test and its completion. This time is actually not very useful, because it includes time during which the CPU was running some other process. If your system is operating under a heavy load, the Ruby interpreter will get less of the CPU's attention and the `real` times won't reflect the actual performance of your benchmarks. You only need `real` times when you're measuring user-visible performance on a running system.

The `user` time is time actually spent running the Ruby interpreter, and the `system` time is time spent in system calls spawned by the interpreter. If your test does a lot of I/O, its `system` time will tend to be large; if it does a lot of processing, its `user` time will tend to be large. The most useful time is probably `total`, the sum of the `user` and `system` times.

When two operations take almost exactly the same time, you can make the difference more visible by putting a `times` loop within the code block passed to `report`. For instance, array lookup and hash lookup are both very fast operations that take too little time to measure. But by timing thousands of lookup operations instead of just one, we can see that hash lookups are a tiny bit slower than array lookups:

```
Benchmark.bm(5) do |timer|
  timer.report('Array') { 1000.times { RANGE.each { |i| array[i] } } }
  timer.report('Hash') { 1000.times { RANGE.each { |i| hash[i] } } }
end
#         user      system       total        real
# Array   0.950000    0.210000    1.160000 (  1.175042)
# Hash    1.010000    0.210000    1.220000 (  1.221090)
```


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

If you want to measure one operation instead of comparing several operations to each other, use `Benchmark#measure`. It returns an object that you can interrogate to get the times, or print out to get a listing in the same format as `Benchmark.bm`. This code demonstrates that I/O-bound code has a larger `system` time:

```
def write_to_file
  File.open('out', 'w') { |f| f.write('a') }
end

puts Benchmark.measure { 10000.times { write_to_file } }
# 0.120000 0.360000 0.480000 ( 0.500653)
```

Recall that the `real` time can be distorted by the CPU doing things other than running your Ruby process. The `user` and `system` times can also be distorted by the Ruby interpreter doing things besides running your program. For instance, time spent doing garbage collection is counted by `benchmark` as time spent running Ruby code.

To get around these problems, use the `Benchmark.bmbm` method. It runs each of your timing tests twice. The first time is just a rehearsal to get the interpreter into a stable state. Nothing can completely isolate the time spent running benchmarks from other tasks of the Ruby interpreter, but `bmbm` should be good enough for most purposes.

See Also

- The standard library documentation for the `benchmark` library has lots of information about varying the format of benchmark reports

Recipe 17.14. Running Multiple Analysis Tools at Once

Problem

You want to combine two analysis tools, like the Ruby profiler and the Ruby tracer. But when one tool calls `set_trace_func`, it overwrites the trace function left by the other.

Solution

Change `set_trace_func` so that it keeps an array of trace functions instead of just one. Here's a library called `multitrace.rb` that makes it possible:

```
# multitrace.rb
$TRACE_FUNCS = []

alias :set_single_trace_func :set_trace_func
def set_trace_func(proc)
  if (proc == nil)
```


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

```
    $TRACE_FUNCS.clear
  else
    $TRACE_FUNCS << proc
  end
end

trace_all = Proc.new do |event, file, line, symbol, binding, klass|
  $TRACE_FUNCS.each { |p| p.call(event, file, line, symbol, binding, klass)}
end
set_single_trace_func trace_all

def unset_trace_func(proc)
  $TRACE_FUNCS.delete(proc)
end
```

Now you can run any number of analysis tools simultaneously. However, when one of the tools stops, they will all stop:

```
#!/usr/bin/ruby -w
# paranoia.rb
require 'multitrace'
require 'profile'
require 'tracer'

Tracer.on
puts "I feel like I'm being watched."
```

This program's nervousness is well-justified, since its every move is being tracked by the Ruby tracer *and* timed by the Ruby profiler:

```
$ ruby paranoia.rb
#0:./multitrace.rb:9:Array:<:     $TRACE_FUNCS << proc
#0:./multitrace.rb:11:Object:<: end
#0:paranoia.rb:9::-: puts "I feel like I'm being watched."
#0:paranoia.rb:9:Kernel:>: puts "I feel like I'm being watched."
…
  %       cumulative     self                self       total
 time       seconds     seconds     calls     ms/call    ms/call    name
  0.00          0.00        0.00         1        0.00       0.00    Kernel.require
  0.00          0.00        0.00         1        0.00       0.00    Fixnum#==
  0.00          0.00        0.00         1        0.00       0.00    String#scan
…
```

Without the `include 'multitrace'` at the beginning, only the profiler will run: its trace function will override the tracer's.

Discussion

This example illustrates yet again how you can benefit by replacing some built-in part of Ruby. The `multitrace` library creates a drop-in replacement for `set_trace_func` that lets you run multiple analyzers at once. You probably don't really want to run the tracer and the analyzer simultaneously, since they're both monolithic tools. But if you've written some smaller, more modular analysis tools, you're more likely to want to run more than one during a single run of a program.

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

The standard way of stopping a tracer is to pass `nil` into `set_trace_func`. Our new `set_trace_func` will accept `nil`, but it has no way of knowing which trace function you want to stop.[4] It has no choice but to remove all of them. Of course, if you're writing your own trace functions, and you know `multitrace` will be in place, you don't need to pass nil into `set_trace_func`. You can call `unset_trace_func` to remove one particular trace function, without stopping the rest.

[4] Well, you could do this by taking a snapshot of the call stack every time `set_trace_func` was called with a `Proc` object. When `set_trace_func` was called with `nil`, you could look at the call stack at that point (see Recipe 17.6), and only remove the `Proc` object(s) inserted by the same file. For instance, if a `nil` call comes in from `profiler.rb`, you could remove only the `Proc` object(s) inserted by calls coming from `profiler.rb`. This is probably not worth the trouble.

See Also

- The tracer function created in Recipe 17.15, "Who's Calling That Method? A Call Graph Analyzer," is the kind of lightweight analysis tool I'd like to see more of: one that it makes sense to run in conjunction with others

Recipe 17.15. Who's Calling That Method? A Call Graph Analyzer

Suppose you're profiling a program such as the one in Recipe 17.12, and the profiler says that the top culprit is `Array#each`. That is, your program spends more time iterating over arrays than doing any one other thing:

```
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 12.19      2.74     2.74     4930     0.56     0.77  Array#each
```

This points you in the right direction, but where do you go from here? Most programs are full of calls to `Array#each`. To optimize your program, you need to know which lines of code are responsible for most of the `Array#each` calls. Ruby's profiler can't give tell you which line of code called a problem method, but it's easy to write a different profiler that can.

The heart of any Ruby profiler is a `Proc` object passed into the `Kernel#set_trace_func` method. This is a hook into the Ruby interpreter itself: if you set a trace function, it's called every time the Ruby interpreter does something interesting like call a method.

Here's the start of a `CallTracker` class. It initializes a hash-based data structure that tracks "interesting" classes and methods. It assumes that we pass a method `tally_calls` into `set_trace_func`; we'll define `tally_calls` a little later.

```
class CallTracker
```


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

```
  # Initialize and start the trace.
  def initialize(show_stack_depth=1)
    @show_stack_depth = show_stack_depth
    @to_trace = Hash.new { |h,k| h[k] = {} }
    start
    at_exit { stop }
end

# Register a class/method combination as being interesting. Subsequent calls
# to the method will be tallied by tally_calls.
def register(klass, method_symbol)
  @to_trace[klass][method_symbol] = {}
end

# Tells the Ruby interpreter to call tally_calls whenever it's about to
# do anything interesting.
def start
  set_trace_func method(:tally_calls).to_proc
end

# Stops the profiler, and prints a report of the interesting calls made
# while it was running.
def stop(out=$stderr)
  set_trace_func nil
  report(out)
end
```

Now let's define the missing methods `tally_calls` and report. The `Proc` object passed into `set_trace_func` needs to take six arguments, but this analyzer only cares about three of them:

`event`

Lets us know what the interpreter is doing. We only care about "`call`" and "`c-call`" events, which let us know that the interpreter is calling a Ruby method or a C method.

`klass`

The `Class` object that defines the method being called.

`symbol`

The name of the method as a `Symbol`.

The `tally_calls` method looks up the class and name of the method being called to see if it's one of the methods being tracked. If so, it grabs the current call stack with `Kernel#caller`, and notes where in the execution path the method was called:

```
# If the interpreter is about to call a method we find interesting,
# increment the count for that method.
def tally_calls(event, file, line, symbol, binding, klass)
  if @to_trace[klass] and @to_trace[klass][symbol] and
  (event == 'call' or event =='c-call')
    stack = caller
```


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

```
    stack = stack[1..(@show_stack_depth ? @show_stack_depth : stack.size)]
    @to_trace[klass][symbol][stack] ||= 0
    @to_trace[klass][symbol][stack] += 1
  end
end
```

All that's left is the method that prints the report. It sorts the results by execution path (as indicated by the stack traces), so the more often a method is called from a certain line of code, the higher in the report that line of code will show up:

```
  # Prints a report of the lines of code that called interesting
  # methods, sorted so that the the most active lines of code show up
  # first.
  def report(out=$stderr)
    first = true
    @to_trace.each do |klass, symbols|
      symbols.each do |symbol, calls|
        total = calls.inject(0) { |sum, ct| sum + ct[1] }
        padding = total.to_s.size
        separator = (klass.is_a? Class) ? '#' : '.'
        plural = (total == 1) ? '' : 's'
        stack_join = "\n" + (' ' * (padding+2))
        first ? first = false : out.puts
        out.puts "#{total} call#{plural} to #{klass}#{separator}#{symbol}"
        (calls.sort_by { |caller, times| -times }).each do |caller, times|
          out.puts " %#{padding}.d #{caller.join(stack_join)}" % times
        end
      end
    end
  end
end
```

Here's the analyzer in action. It analyses my use of the Rubyful Soup HTML parser (which I was working on optimizing) to see which lines of code are responsible for calling `Array#each`. It shows three main places to look for optimizations:

```
require 'rubygems'
require 'rubyful_soup'
tracker = CallTracker.new
tracker.register(Array, :each)

BeautifulSoup.new(open('test.html') { |f| f.read })
tracker.stop($stdout)
# 4930 calls to Array#each
# 1671 ./rubyful_soup.rb:715:in `pop_to_tag'
# 1631 ./rubyful_soup.rb:567:in `unknown_starttag'
# 1627 ./rubyful_soup.rb:751:in `smart_pop'
#    1 ./rubyful_soup.rb:510:in `feed'
```

By default, the `CallTracker` shows only the single line of code that called the "interesting" method. You can get more of the call stack by passing a larger `show_stack_depth` into the `CallTracker` initializer.

See Also

- Recipe 17.6, "Creating and Understanding Tracebacks"
- Recipe 17.12, "Profiling Your Application"

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