RUBY MTN: Spring 2016, LECTURE#6

Занятие: 6

Класс Dir

Основная направленность класса `Dir` - предоставить пользователю возможность запрашивания, обхода и фильтрации по директориям файловой системы. Данный класс предоставляет несколько базовых методов для создания и удаления директорий.

Создание директорий

Вы можете создать новую директорию в вашей файловой системе просто передав желаемый путь к ней (полный или относительный) в качестве аргумента в метод `Dir#mkdir`. Этот метод вызовет ошибку `Errno::EEXIST`, если такая директория уже существует, или `SystemCallError`, если директория не может быть создана в следствии ограниченых прав доступа.
Все относительные пути (включая и те, что вы передаете в методы класса `File`) происходят из пути к текущей рабочей директории, доступ к которой можно получить через метод `Dir#pwd`.
Вы можете изменить рабочую директорию через метод `Dir#chdir`. Для упрощения переходов между рабочими дииректориями вы можете передать блок кода в метод `#chdir` и рабочая директория будет изменена после его выполнения:

```
Dir.pwd
# => "/home/k3/code"
Dir.mkdir("test")
Dir.chdir("test") do
  Dir.pwd
  # "/home/k3/code/test"
  File.new("file.rb", "w")
end

Dir.pwd
# => "/home/k3/code"
```

Обход содержимого директории Класс `Dir` предоставляет два метода для обхода содержимого директорий, это методы `Dir#entries` и `Dir#glob`.
`Dir#entries` возвращает массив с именами всего содержимого директории, включая текущий путь (`.`), родительскую директорию (`..`) и все скрытые файлы (в `linux` это все те, в начале имен которых стоит точка). Те же правила относятся и к методу `#each`, который возвращает все содержимое и действует на основе методов из модуля `Enumerable`.
Метод `#entries` может вызывать раздражение, если вы желаете работать только с файлами, как это продемонстрировано в следующем примере, где мы пытаемся удалить все файлы из определенной директории:

```
d = Dir.new("/home/k3/code")
d.entries.each do |e|
  next if e =~ /^\./
  file = File.join(d.path, e)
  File.delete(file) if File.file?(file)
end
```

Гораздо более удобный способ обхода файлов в директории — использовать метод `#glob`, или очень похожий метод `#[]`. Эти методы принимают шаблон и возвращают массив с путями всех видимых файлов, которые соответствуют шаблону. Шаблоны основаны на специальном кратком синтаксисе, где `*` представляет любое число групповых символов, `**` представляет все дочерние директории (их проверка производится рекурсивно) и `?` представляет один групповой символ.

- `*` - все файлы текущей директории
- `help.*` - все файлы с именем `help` и любым расширением

- `*/**/*.rb` - все файлы с расширением `.rb`, из текущей и вложенных директорий

Предыдущий пример с удалением, обновлённый:

```
Dir["/home/k3/code/*"].each { |f| File.delete(f) if File.file?(f) }
```

Модуль `FileUtils`

Модуль `FileUtils` предоставляет интересный подход к манипуляции файлами путем эмуляции множества команд `Unix` для работы с файлами и большинства их опций (флагов). То, что вы можете сделать командами типа `rm -rf` и `ln -s` может быть выполнено соответственно методами `FileUtils#rm_rf` и `FileUtils#ln_s`.
Так как они придерживаются синтаксиса знакомого большинству `Ruby` программистов они являются очень простыми для понимания, делая список вызовов методов `FileUtils` очень похожим на сеанс `bash`. Пример ниже демонстрирует эту схожесть:

```
require "fileutils"
FileUtils.touch(["some_file.rb", "another_file.rb"])
FileUtils.mkdir("code")
FileUtils.mv(["another_file.rb", "../other_file.rb"], "code")
Dir["code/*"]
# => ["code/some_file.rb", "code/another_file.rb"]

FileUtils.cp_r("code", "bkp")
FileUtils.rm_r("code") Dir["code/*"]
# => []

Dir["bkp/*"]
# => ["bkp/some_file.rb", "bkp/another_file.rb"]
```

Как и многие команды `Unix`, многие методы `FileUtils` знают как работать с множеством файлов которое передано параметром в виде массива, например, метод `FileUtils#cp`.
Они также могут принимать флаги для изменения своего поведения:

```
FileUtils.rm("a_file.rb") # removes this file
FileUtils.rm(Dir["bkp_*"]) # remove all files that start with bkp
FileUtils.rm(Dir["bkp_*"], verbose: true) # print the equivalent stmt and remove the bkp files
```

Класс `Pathname`

Класс `Pathname` занимается представлением пути, размещения файла в файловой системе и предоставляет возможности для запрашивания и манипулирования данными пути.
Хотя класс `Pathname` не так универсален в использовании как модуль `FileUtils`, он может принести больше точности когда вам необходимы сложные обходы файловой системы. `Pathname` также используется в таких `gem`'ах как:
`Sprockets` (компилятор и сборщик вебстатики: `js` / `css` и тд) и `Carrierwave` (загрузчик картинок в вебпроектах).

Работа с коммандной строкой

Ввод и обработка пользовательских данных

- `$stdin.gets` - считать с `stdin`
- `#chomp` - обрезать последний управляющий символ символ

```
print "Hello: "
answer = gets
p answer
# => 'Hello:'
```

`$?` - объект статуса последней выполненной команды

```
puts $?.success?
```

`ARGV` - массив опций, с какими был запущен данный скрипт:

```
# 1.rb
p ARGV
```
```bash
$ ruby 1.rb c 1 B 1
=> ["c", "1", "B", "1"]
```

Обратная кавычка `

- возвращает стандартный вывод команды (`STDOUT`)
- блокирующая (пока не выполниться - ждём)
- ошибка вызывает исключение в мастерпроцессе
- без перенаправления нет возможности отловить `STDERR`

```
ls = `ls a`
puts ls
```

```
output = `xxxx`
puts "output is #{output}"
# Exception возникнет раньше выполненения #puts
```

Другая возможная запись: `%x` , позволяет использовать другие разделители, нежели `

```
output = %x[ ls ]
output = %x{ ls }
```

Метод `system`

Очень похожий на `, но разница в том, что `system` возвращает `true` , если команда была выполнена успешно (статус 0), `false` - если статус не нулевой и `nil` - если команда упала. Вторым важным отличием является то, что `system` игнорирует исключения, так что ваш процесс не упадёт, если команда упадёт.

```
output = system('xxxxxxx')
puts "output is #{output}" # => output is

print "Hello: "
answer = gets.chomp
system(answer)

output=`ls no_existing_file`
result=$?.success?
puts result
```

`exec`

- заменяет ваш текущий процесс на переданную команду

```
exec 'echo'
:~/tmp$ # я больше не в `irb`!
```

На самом же деле, под копотом, оба `system` и обратная кавычка делают форк текущего процесса и на нём уже выполняют команду через `exec` 😄

Так как `exec` меняет текущий процесс - он не возвращает ничего, если операция успешна. Если же неуспешна - вернётся `SystemCallError`.

`popen3`

Если вам всё же нужно отдельно отлавливать разные потоки - то это идеальный вариант, так как он разделяет `stdin`, `stdout` и `stderr` автоматически.

```
require 'open3'

cmd = 'ping www.google.com'

Open3.popen3(cmd) do |stdin, stdout, stderr, wait_thr|
  exit_status = wait_thr.value
  unless exit_status.success?
    abort "FAILED !!! #{cmd}"
  end
end
```

`Process#spawn`

`Kernel#spawn` выполняет заданную команду в `subshell`, отдельным процессом и возвращает `id` этого процесса:

```
pid = Process.spawn("ls al")
# => 81001
```

Безопасность запуска команд

```
untrusted = "ls -a; rm -rf /tmp/*"

out = `echo #{untrusted}` # BAD
puts out
out = `echo "#{untrusted}"` # BAD (vulnerable for "" inside untrusted)
puts out

ret = system "echo #{untrusted}" # BAD
ret = system 'echo', untrusted  # GOOD!

require 'open3'
out, err, st = Open3.capture3("echo #{untrusted}") # BAD
puts out
puts err

p st.exitstatus

out, err, st = Open3.capture3('echo', untrusted) # GOOD!
puts err
puts out
p st.exitstatus
```

Bonus

Добавляем возможности итерации для своего класса

Мы уже знакомы с модулем `Enumerable` в `Ruby` и помним, что он добавляет некоторые методы для коллекций, например `#map`, `#inject`, `#select` и т.д.

Рассмотрим пример, в котором у нас имеется класс `Team`, который управляет командами и членами этих комманд.

```
class Team
  include Enumerable
```

```
  attr_accessor :members

  def initialize(members = [])
    @members = members
  end

  def each(&block)
    @members.each { |member| block.call(member) }
  end
end
```

`Enumerable` требует использования в контексте вашего класса метода `#each`, который передает элементы в некую коллекцию (у нас в `@members`). Все методы из модуля `Enumerable` полагаются на это.
Для примера давайте воспользуемся методом `#map`:

```
team = Team.new(['joshua', 'gabriel', 'jacob'])
# => #<Team:0x00000002541968 @members=["joshua", "gabriel", "jacob"]>

team.map { |member| member.capitalize }
# => ["Joshua", "Gabriel", "Jacob"]
```

Теперь мы можем вызывать любой метод из модуля `Enumerable` как метод экземпляра класса `Team` и этот метод будет знать, что нам требуется работать внутри с массивом `@members`. `Enumerable` может быть мощной примесью к вашим собственным классам.

`Module#prepend` и `#ancestors`

```
module M
  def test
    p "module test"
  end
end

class Test1
  include M

  def test
    p "class test"
  end
end

Test1.new.test # => "class test"

class Test2
  prepend M

  def test
    p "class test"
  end
end

Test2.new.test # => "module test"
```

Домашнее задание: 6

Теория:

- прочесть заметки лекции ещё раз, два, три...
- изучить следующие ссылки:
  - http://mentalized.net/journal/2010/03/08/5-ways-to-run-commands-from-ruby/
  - http://lonelyelk.ru/posts/26
- осмотреться в документации:
  - http://www.ruby-doc.org/core-2.2.0/IO.html
  - http://www.ruby-doc.org/core-1.9.3/Kernel.html#method-i-gets
  - http://www.ruby-doc.org/core-2.2.0/File.html
  - http://ruby-doc.org/stdlib-1.9.3/libdoc/yaml/rdoc/YAML.html

    - http://www.ruby-doc.org/core-2.2.0/Dir.html
    - http://www.ruby-doc.org/stdlib-2.2.0/libdoc/fileutils/rdoc/FileUtils.html
- прочитать главы о работе с файлами и коммандной строкой в любой из книг по `Ruby`

Практика: Нет, пятница же 😄

Всем удачи!