RubyCodeSnippets - A Selection of Ruby Code Snippets

Table of Contents

RubyCodeSnippets

A Selection of Ruby Code Snippets

Sat Aug 11 11:56:18 UTC 2007

Erik Veenstra <erikveen@dds.nl>

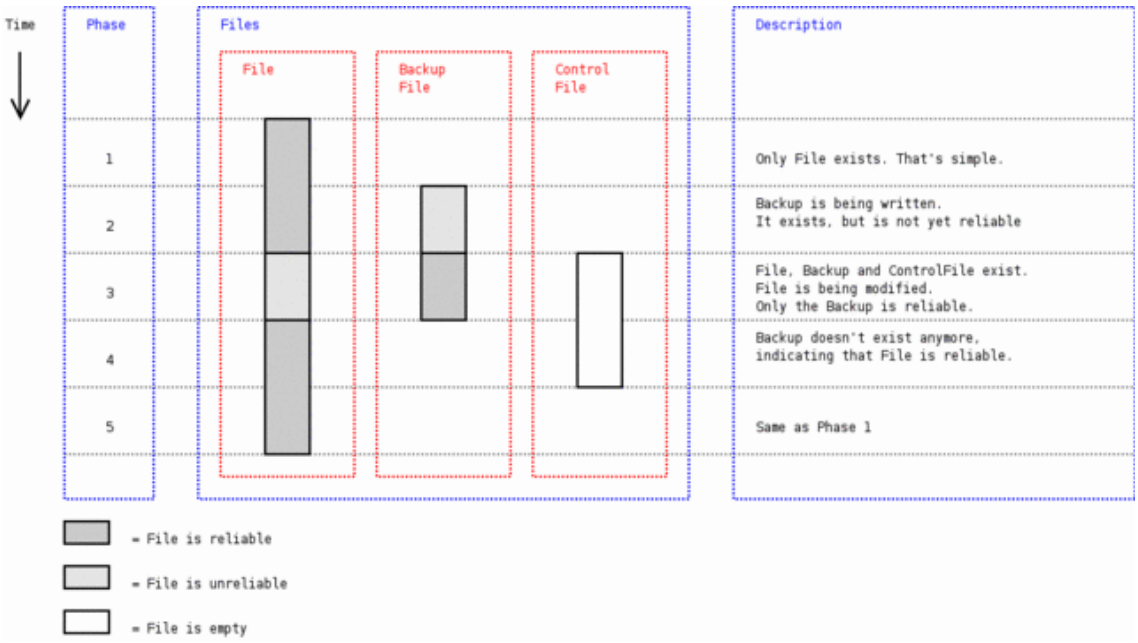
1. File.rollbackup

It's a combination of *Rollback* and *Backup*... It's kind of file handling in a transaction: It either succeeds, or it fails. After restarting the program, you've got either the old contents or the new contents. No half-written files, no corrupt files...

`File.rollbackup` expects two parameters and a block: `filename` and `mode` (like in `File.new`) and the block which is called within the transaction. If the block receives a parameter (`arity==1`), the file is opened with `File.open(filename,mode)` and passed to the block. If the block doesn't receive a parameter (`arity==0`), the opening of the file has to be done by hand, in the block (or not at all...).

You might do a `File.rollbackup(filename)`, without the mode and without the block: The rollback is done, if necessary, and nothing else.

Here's a schematic overview of the internals of `File.rollbackup`:



1.1. Examples

a) Manually Opening the File

```
file = "hash.txt"

File.rollbackup(file) do
  hash = Hash.file(file)
  # ... manipulate the hash ...
  hash.save(file)
end
```

Because the arity of the block is 0, the file isn't opened.

b) Automatically Opening the File

```
file = "data.txt"

File.rollbackup(file, "w") do |f|
  # ... write the file ...
end
```

Because the arity of the block is 1, the file is opened and the handler is passed to the block.

This is basically the same as `File.open(file, "w"){|f| ...}`, except for the backup and the rollback.

1.2. Code

```
require "ftools"

class File
  def self.rollbackup(file, mode="r", &block)
    backupfile  = file + ".RB.BACKUP"
    controlfile = file + ".RB.CONTROL"
    res         = nil
    is_new      = (not File.file?(file))

    File.open(file, "a"){|f|}    unless File.file?(file)

    backup =
    lambda do
      File.copy(file, backupfile)
      File.open(controlfile, "a"){|f|}
    end

    rollback =
    lambda do
      if File.file?(backupfile) and File.file?(controlfile)
        File.copy(backupfile, file)
        File.delete(file)       if is_new
      end
    end

    cleanup =
    lambda do
      File.delete(backupfile)   if File.file?(backupfile)
      File.delete(controlfile)  if File.file?(controlfile)
    end

    rollback.call
    cleanup.call

    if block_given?
      begin
        backup.call

        if block.arity == 0
          res   = block.call
        else
          File.open(file, mode) do |f|
```

```
            res = block.call(f)
          end
        end

        cleanup.call
      ensure
        rollback.call
        cleanup.call
      end
    end

    res
  end
end
```

1.3. Unit Tests

```
require "ev/rollbackup" # That's where I store File#rollbackup.
require "test/unit"

class TestRollBackup < Test::Unit::TestCase
  TEMP_DIR      = ENV["TEMP"] || "/tmp"
  DATA_FILE     = File.expand_path("ut_rollbackup_test_file", TEMP_DIR)
  BACKUP_FILE   = DATA_FILE + ".RB.BACKUP"
  CONTROL_FILE  = DATA_FILE + ".RB.CONTROL"
  DATA_1        = "something one"
  DATA_2        = "something two"

  class BOOM < RuntimeError
  end

  def setup
    File.delete(DATA_FILE)      if File.file?(DATA_FILE)
    File.delete(BACKUP_FILE)    if File.file?(BACKUP_FILE)
    File.delete(CONTROL_FILE)   if File.file?(CONTROL_FILE)
  end

  def teardown
    assert(! File.file?(BACKUP_FILE))
    assert(! File.file?(CONTROL_FILE))

    File.delete(DATA_FILE)      if File.file?(DATA_FILE)
    File.delete(BACKUP_FILE)    if File.file?(BACKUP_FILE)
    File.delete(CONTROL_FILE)   if File.file?(CONTROL_FILE)
  end

  def test_backup_and_control_files_exist
    File.rollbackup(DATA_FILE) do
      assert(File.file?(DATA_FILE))
      assert(File.file?(BACKUP_FILE))
      assert(File.file?(CONTROL_FILE))
    end
  end

  def test_without_mode_and_new_file
    File.rollbackup(DATA_FILE) do
      File.open(DATA_FILE, "w") do |f|
        f.write(DATA_2)
      end
```

```
    end

    assert_equal(DATA_2, get_data)
  end

  def test_with_mode_and_new_file
    File.rollbackup(DATA_FILE, "w") do |f|
      f.write(DATA_2)
    end

    assert_equal(DATA_2, get_data)
  end

  def test_without_mode_and_existing_file
    File.open(DATA_FILE, "w")           {|f| f.write(DATA_1)}

    File.rollbackup(DATA_FILE) do
      File.open(DATA_FILE, "w") do |f|
        f.write(DATA_2)
      end
    end

    assert_equal(DATA_2, get_data)
  end

  def test_with_mode_and_existing_file
    File.open(DATA_FILE, "w")           {|f| f.write(DATA_1)}

    File.rollbackup(DATA_FILE, "w") do |f|
      f.write(DATA_2)
    end

    assert_equal(DATA_2, get_data)
  end

  def test_with_mode_and_raise
    File.open(DATA_FILE, "w")           {|f| f.write(DATA_1)}

    assert_raise(BOOM) do
      File.rollbackup(DATA_FILE, "w") do |f|
        f.write(DATA_2)

        raise BOOM, "Oops!"
      end
    end

    assert_equal(DATA_1, get_data)
  end

  def test_with_mode_and_throw
    File.open(DATA_FILE, "w")           {|f| f.write(DATA_1)}

    assert_throws(:boom) do
      File.rollbackup(DATA_FILE, "w") do |f|
        f.write(DATA_2)

        throw :boom
      end
    end
```

```
    assert_equal(DATA_1, get_data)
  end

  def test_recovery_from_phase_2
    File.open(DATA_FILE, "w")           {|f| f.write(DATA_1)}
    File.open(BACKUP_FILE, "w")         {|f| f.write(DATA_2)}

    File.rollbackup(DATA_FILE)

    assert_equal(DATA_1, get_data)
  end

  def test_recovery_from_phase_3
    File.open(DATA_FILE, "w")           {|f| f.write(DATA_1)}
    File.open(BACKUP_FILE, "w")         {|f| f.write(DATA_2)}
    File.open(CONTROL_FILE, "a")        {|f|}

    File.rollbackup(DATA_FILE)

    assert_equal(DATA_2, get_data)
  end

  def test_recovery_from_phase_4
    File.open(DATA_FILE, "w")           {|f| f.write(DATA_1)}
    File.open(CONTROL_FILE, "a")        {|f|}

    File.rollbackup(DATA_FILE)

    assert_equal(DATA_1, get_data)
  end

  def get_data
    File.open(DATA_FILE){|f| f.read}
  end
end
```

2. SparseFile

I had to send huge files over a network to another machine. Most of these files were image files for QEMU: typically 4 GB, of which only a small portion (~ 400 MB) was used. Both client and server were Ruby programs on Linux boxes, communicating via FTP.

I thought it was a good idea to use sparse files to save disk space (not bandwidth...), so I searched for a SparseFile class, couldn't find one [1] and wrote one myself.

It seems to work pretty well on both Linux and Cygwin. On Windows, the resulting file isn't sparse, although the checksum is correct. This means that you can safely use SparseFile in your platform-agnostic application.

(What about OS/X? Somebody willing to give it a try?)

2.1. Example

A kind of sparsifier...

```
inputfile       = ARGV.shift
outputfile      = ARGV.shift
blocksize       = ARGV.shift || 4096

File.open(inputfile , "rb") do |f1|
  EV::SparseFile.open(outputfile, "wb") do |f2|
    while (block = f1.read(blocksize))
      f2.write block
    end
  end
end
```

2.2. Code

```
module EV
  class SparseFile
    # You should always call close. Really...
    # Use SparsFile::open with a block instead of EV::SparseFile::new.

    attr_reader :pos
    attr_writer :pos

    def self.open(*args)
      sparse_file       = new(*args)

      if block_given?
        begin
          res   = yield(sparse_file)
        ensure
          sparse_file.close
        end
      else
        res     = sparse_file
      end
```

```
    res
  end

  def initialize(file_name, mode="wb")
    @file             = File.new(file_name, mode)
    @pos              = 0
    @last_byte_pos    = 0
    @last_byte        = nil
  end

  def read(length=nil)
    @file.pos = @pos
    @file.read(length)
  end

  def write(data)
    length    = data.length

    unless data.count("\000") == length
      @file.pos       = @pos
      @file.write(data)
    end

    @pos += length

    if @pos > @last_byte_pos
      @last_byte_pos  = @pos
      @last_byte      = data[-1..-1]
    end

    length
  end

  def truncate(length)
    write_last_byte

    @pos      = length

    @file.truncate(length)
  end

  def close
    write_last_byte

    @file.close
  end

  def size
    [@last_byte_pos, @pos].max
  end

  private

  def write_last_byte
    if @last_byte_pos > 0
      @file.pos       = @last_byte_pos-1
      @file.write(@last_byte)

      @pos            = @last_byte_pos
      @last_byte_pos  = 0
```

```
      end
    end
  end
end
```

2.3. Unit Tests

```
require "ev/sparsefile" # That's where I store EV::SparseFile.
require "md5"
require "test/unit"

class TestSparseFile < Test::Unit::TestCase
  TEMP_DIR      = ENV["TEMP"] || "/tmp"
  FILE_1        = File.expand_path("ut_sparsefile_test_file_1", TEMP_DIR)
  FILE_2        = File.expand_path("ut_sparsefile_test_file_2", TEMP_DIR)

  def setup
    File.delete(FILE_1) if File.file?(FILE_1)
    File.delete(FILE_2) if File.file?(FILE_2)
  end

  def teardown
    File.delete(FILE_1) if File.file?(FILE_1)
    File.delete(FILE_2) if File.file?(FILE_2)
  end

  def test_size
    EV::SparseFile.open(FILE_2) do |f|
      f.write("\000"*10)

      size1     = 10
      size2     = f.size

      assert_equal(size1, size2)
    end

    size1       = 10
    size2       = File.size(FILE_2)

    assert_equal(size1, size2)
  end

  def test_position
    EV::SparseFile.open(FILE_2) do |f|
      f.write("\000"*10)

      pos1      = 10
      pos2      = f.pos

      assert_equal(pos1, pos2)
    end
  end

  def test_reposition
    EV::SparseFile.open(FILE_2) do |f|
      f.write("\000"*10)

      f.pos     = 1
      f.write("1")
```

```
      f.pos     = 5
      f.write("5")
    end

    size1       = 10
    size2       = File.size(FILE_2)
    data1       = "\000"*10
    data1[1]    = "1"
    data1[5]    = "5"
    data2       = File.open(FILE_2, "rb"){|f| f.read}

    assert_equal(size1, size2)
    assert_equal(data1, data2)
  end

  def test_write_past_end
    EV::SparseFile.open(FILE_2) do |f|
      f.pos     = 9
      f.write("9")
    end

    size1       = 10
    size2       = File.size(FILE_2)
    data1       = "\000"*10
    data1[9]    = "9"
    data2       = File.open(FILE_2, "rb"){|f| f.read}

    assert_equal(size1, size2)
    assert_equal(data1, data2)
  end

  def test_sparsify_big_file
    File.open(FILE_1, "wb") do |f|
      f.write("\000" * 1E6)

      f.pos     = 111_111
      f.write("X")

      f.pos     = 777_777
      f.write("Y")
    end

    File.open(FILE_1) do |f1|
      EV::SparseFile.open(FILE_2) do |f2|
        while (block = f1.read(4096))
          f2.write(block)
        end
      end
    end

    size1       = File.size(FILE_1)
    size2       = File.size(FILE_2)

    assert_equal(size1, size2)

    md5sum1     = File.open(FILE_1, "rb"){|f| MD5.new(f.read)}
    md5sum2     = File.open(FILE_2, "rb"){|f| MD5.new(f.read)}

    assert_equal(md5sum1, md5sum2)
```

```
  end

  def test_just_low_values
    EV::SparseFile.open(FILE_2) do |f|
      f.write("\000"*100_000)
    end

    data1       = "\000"*100_000
    data2       = File.open(FILE_2, "rb"){|f| f.read}

    assert_equal(data1, data2)
  end

  def test_read_with_close
    EV::SparseFile.open(FILE_2) do |f|
      f.write("\000"*10)

      f.pos     = 5
      f.write("5")
    end

    data1       = "\000"*10
    data1[5]    = "5"
    data2       = EV::SparseFile.open(FILE_2, "rb"){|f| f.read}

    assert_equal(data1, data2)

    data1       = "5"
    data2       = EV::SparseFile.open(FILE_2, "rb"){|f| f.pos=5 ; f.read(1)}

    assert_equal(data1, data2)
  end

  def test_read_without_close
    EV::SparseFile.open(FILE_2, "w+b") do |f|
      f.write("\000"*10)

      f.pos     = 5
      f.write("5")

      f.pos     = 5

      data1     = "5"
      data2     = f.read(1)

      assert_equal(data1, data2)

      size1     = 10
      size2     = f.size

      assert_equal(size1, size2)
    end
  end

  def test_truncate
    EV::SparseFile.open(FILE_2, "w+b") do |f|
      f.write("\000"*10)

      f.pos     = 5
      f.write("5")
```

```
      f.truncate(7)

      size1     = 7
      size2     = f.size

      assert_equal(size1, size2)

      pos1      = 7
      pos2      = f.pos

      assert_equal(pos1, pos2)
    end

    data1       = "\000"*7
    data1[5]    = "5"
    data2       = EV::SparseFile.open(FILE_2, "rb"){|f| f.read}

    assert_equal(data1, data2)

    size1       = 7
    size2       = File.size(FILE_2)

    assert_equal(size1, size2)
  end
end
```

3. Generic Delegator

A simple Delegator class: "Delegation is a way of extending and reusing a class by writing another class with additional functionality that uses instances of the original class to provide the original functionality."

3.1. Examples

a) Plain Delegation

Foo is a simple ordinary class:

```
class Foo
  def one
    1
  end

  def two
    2
  end
end
```

Bar is a simple delegator class:

```
class Bar < EV::Delegator
  def two
    22
  end
end
```

As you see, although Bar is going to be used as a delegator of Foo, we can't see this relation in the class definitions. Can we say that delegator *classes* don't exist and only delegator *objects* exist?

This is how `Foo` and `Bar` are used and how they interact:

```
o1  = Foo.new
o2  = Bar.delegate(o1)

p o1.one       # ==> 1
p o1.two       # ==> 2

p o2.one       # ==> 1
p o2.two       # ==> 22
```

b) Hash with Accessors

```
class HashWithAccessors < EV::Delegator
  def method_missing(method_name, value=nil, &block)
    method_name = method_name.to_s
    key         = method_name.gsub(/=$/, "").intern

    if method_name =~ /=$/
      @real_object[key] = value
```

```
    else
      @real_object[key]
    end
  end
end

class Hash
  def with_accessors(&block)
    HashWithAccessors.delegate(self, &block)

    self
  end
end
```

Using with_accessors to easily access the pairs in a hash:

```
hash    = {:a=>111, :b=>222}

hash.with_accessors do |h|
  h.c = h.a + h.b
end

p hash  # ==> {:a=>111, :b=>222, :c=>333}
```

3.2. Code

```
module EV
  class Delegator
    def initialize(real_object, &block)
      @real_object      = real_object

      block.call(self)  if block
    end

    def method_missing(method_name, *args, &block)
      @real_object.__send__(method_name, *args, &block)
    end

    def self.delegate(*args, &block)
      res       = self.new(*args)
      res       = block.call(res)       if block
      res
    end

    def self.un_delegate(delegator_object, &block)
      res       = delegator_object.instance_variable_get("@real_object")
      res       = block.call(res)       if block
      res
    end
  end
end
```

3.3. Unit Tests

```
require "ev/delegator"  # That's where I store EV::Delegator.
require "test/unit"
```

3. Generic Delegator

```
class TextDelegator < Test::Unit::TestCase
  class Foo
    def one
      1
    end

    def two
      2
    end
  end

  class Bar < EV::Delegator
    def two
      22
    end

    def three
      33
    end
  end

  def test_delegate
    foo = Foo.new
    bar = Bar.delegate(foo)

    assert(Foo, foo.class)
    assert(Bar, bar.class)

    assert_not_same(foo, bar)

    assert_equal(1, foo.one)
    assert_equal(2, foo.two)
    assert_raise(NoMethodError) {foo.three}

    assert_equal(1, bar.one)
    assert_equal(22, bar.two)
    assert_equal(33, bar.three)
  end

  def test_undelegate
    foo1        = Foo.new
    bar         = Bar.delegate(foo1)
    foo2        = Bar.un_delegate(bar)

    assert(Foo, foo1.class)
    assert(Foo, foo2.class)
    assert(Bar, bar.class)

    assert_not_same(foo1, bar)
    assert_same(foo1, foo2)
  end

  def test_block
    foo = Foo.new

    result =
    Bar.delegate(foo) do |delegated_foo|
      assert(delegated_foo.kind_of?(Bar))

      :result
```

```
    end

    assert_equal(:result, result)
  end
end
```

Notes

[1] *The library "win32/file" seems to be able to handle sparse files on WIN32 systems.*