Ruby 初級者向けレッスン第 32 回

okkez @ Ruby 関西

2009 年 10 月 31 日

今回の内容

- Test::Unit の使い方
- テスト駆動開発

今回のゴール

- Test::Unit の使い方を知る
- テスト駆動開発を体験する

Ruby でのテスト

- テストとは？
- Test::Unit の基礎
- いろんなテストメソッド
- ランナー
- Rake で実行

テストとは？

- 起こるべき動作を記述する
- テストを見れば仕様がわかる
- コードの変更を安心して行える

Test::Unit

- 助田さんによる RubyUnit が始まり
- Ruby1.8 では test/unit として標準添付
- Ruby1.9 では test/unit とそこそこ互換性のある miniunit が標準添付されている
  - test/unit は Gem としてより新しいバージョンがあります

Test::Unit の基礎

「1 + 1 は 2 であるべき」のテスト

例:

```
01: # test_sample.rb
02: require 'test/unit'
03: class TestSample < Test::Unit::TestCase
04:   def test_tashizan
05:     assert_equal(2, 1 + 1, '1 + 1 equals 2.')
06:   end
07: end
```

- class TestSample < Test::Unit::TestCase
  - テスト用のクラスは Test::Unit::TestCase を継承する
- def test_tashizan
  - テストを実行するメソッドは 'test' で始める
- assert_equal(expected, actual, msg = nil)
  - expected == actual ならば成功で、失敗なら msg を出力する

実行結果

例:

```
$ ruby samples/test_sample.rb
Loaded suite samples/test_sample
Started
.
Finished in 0.00051 seconds.

1 tests, 1 assertions, 0 failures, 0 errors
```

もし失敗すると...

例:

```
$ ruby samples/test_sample.rb
Loaded suite samples/test_sample
Started
F
Finished in 0.097217 seconds.

  1) Failure:
test_tashizan(TestSample) [samples/test_sample.rb:5]:
1 + 1 equals 2.
<2> expected but was
<3>.

1 tests, 1 assertions, 1 failures, 0 errors
```

前準備と後処理

- setup：各テストメソッドが呼ばれる前に呼ばれる
- teardown：各テストメソッドが呼ばれた後に呼ばれる

例:

```
require 'test/unit'
class TestArray < Test::Unit::TestCase
  def setup
    @obj = Array.new
  end

  def test_size
    assert_equal(0, @obj.size)
  end
end
```

いろんなテストメソッド

- assert(boolean, msg = nil)
  - boolean が真なら成功
- assert_match(pattern, string, msg = '')
  - string =~ pattern が真なら成功

- assert_raise(expected_exception_klass, msg = ”) {...}
  - ブロックを実行して expected_exception_klass クラスの例外が発生すれば成功
- などなど（ri Test::Unit::Assertions を参照）
- http://doc.okkez.net/187/view/class/Test=Unit=Assertions

例:

```
class TestHoge < Test::Unit::TestCase
  def setup
    @obj = Hoge.new
  end
  def test_hoge
    assert @obj.hoge?
    assert_match /\Afoo\z/, @obj.foo
    assert_raise(StandardError) do
      @obj.baz # このメソッドの呼び出しで StandardError が発生するはず
    end
  end
end
```

Test::Unit の小技

–help が効く

例:

```
$ ruby test_stack.rb --help
Test::Unit automatic runner.
Usage: test_stack.rb [options] [-- untouched arguments]

    -r, --runner=RUNNER
    -n, --name=NAME
    -t, --testcase=TESTCASE
    -v, --verbose=[LEVEL]
```

- テストメソッドの指定

```
$ ruby test_stack.rb -n test_empty?
```

- テストクラスの指定

```
$ ruby test_stack.rb -t TestStack
```

- ランナー (テストの UI) の指定

```
$ ruby test_stack.rb -r tk (Ruby/Tk)
$ ruby test_stack.rb -r gtk2 (Ruby/GTK2)
```

- テストの詳細の表示

```
$ ruby test_stack.rb -v
```

miniunit では出来ません。

複数のテストファイルの連続実行

```
# runner.rb
require 'test/unit'
Test::Unit::AutoRunner.run(true)
```

- ruby runner.rb とやると、カレントディレクトリの test_*.rb をすべて実行する
- AutoRunner にも--help が効く

```
$ ruby runner.rb --help
Test::Unit automatic runner.
Usage: runner.rb [options] [-- untouched arguments]

    -a, --add=TORUN
    -p, --pattern=PATTERN
    -x, --exclude=PATTERN
```

- 実は testrb というコマンドを使えば上記のファイルを作る必要は無い
  - 1.8.1 からあるけど意外と知られていない？

Rake を使ってテストを実行する

例:

```
01: # -*- coding: utf-8 -*-
02: require 'rake/testtask'
03:
04: Rake::TestTask.new('test') do |t| # タスク名を定義
05:   t.libs << 'samples'             # 追加の $LOAD_PATH を指定する
06:   t.pattern << 'test/test_*.rb'   # テストファイル名のパターン
07:   t.verbose = true
08: end
```

上記のように必要なファイルを require してタスクを定義すれば良い。詳細はマニュアルを参照してください。

- gem server を実行してローカルにインストールされている RDoc を見る
- Ruby リファレンスマニュアル刷新計画で作成中のマニュアルを参照する

その他のテスティングフレームワーク

RSpec
: http://rspec.info/

expectations
: http://expectations.rubyforge.org/

shoulda
: http://www.thoughtbot.com/projects/shoulda

miniunit
: http://rubyforge.org/projects/bfts

test/unit
: http://test-unit.rubyforge.org/index.html.ja

テスト駆動開発

- まずテストを書く -> 失敗
- 次にコードを書く -> 成功
- リファクタリング -> 成功

リファクタリングとは

- **仕様を保持したまま**コードを変更する
- だから、仕様が決まっていないとリファクタリングしようがない
- だから、テストがないと「仕様の保持」が保証できない
- 「リファクタリング中に 2、3 日システムが動かなくなっちゃってーなどと言ってる奴がいたら、

んなもんリファクタリングじゃあなーいと言ってやれ」by Martin Fowler

演習

スタッククラスを作ろう！

- テスト駆動開発で「スタック」のクラスを作る。
- push で順に要素を追加していって、pop で最後から順に要素を取り出す。
- 皿を縦に積んで上から取る感じ。
- このような動作を LIFO (last in, first out) という。

ワンポイントアドバイス

今、自分が「実装コードを書いている」のか「テストコードを書いているのか」を常に意識するようにしてください。

定義するメソッド

全て Stack クラスのインスタンスメソッドです。

- empty?
  - スタックが空なら true、そうでなければ false を返す。
- size
  - スタックのサイズを返す。
- push(val)
  - 引数の値をスタックの一番上に積む。
- pop
  - スタックの一番上の値を取り除いて返す。スタックが空の場合は Stack::EmptyError が発生する。

Step 1

- Stack#empty? のテスト
- 「新しいスタックは空なので empty? は真」

```
# test_stack.rb
require 'stack'
require 'test/unit'
class TestStack < Test::Unit::TestCase
  def setup
    @stack = Stack.new
  end
  def test_empty?
    ... # ここに記述する内容を考える
  end
end
```

- Stack#empty? を作る (1) - 失敗

```
# stack.rb
class Stack
  def empty?
    return false
  end
end
```

- 「真偽を返す」形でメソッドを定義
- テストの実行

```
1) Failure:

test_empty?(TestStack) [test_stack.rb:6]:
a new stack is empty.
<false> is not true.
```

- Stack#empty? で true を返して、とりあえず「テストが通る」コードにする
- これをフェイクする (Fake it!) と言う

Step 2

- Stack#push と Stack#pop のテスト
- 「新しいスタックに 3 を push して pop すると 3 が返る」

```
# test_stack.rb (一部)
class TestStack < Test::Unit::TestCase
  def test_push_and_pop
    ... # ここに記述する内容を考える
  end
end
```

- Stack#push と Stack#pop を作る (1) - 失敗

```
# stack.rb（一部）
class Stack
  def push(val)
  end
  def pop
    return 0
  end
end
```

- Stack#push は「引数を一つ取る」形で定義
- Stack#pop は「何かを返す」形で定義
- テストの実行

```
1) Failure:

test_push_and_pop(TestStack) [test_stack.rb:12]:
pop returns the last value.
<3> expected but was
<0>.
```

- Stack#push は何もせず Stack#pop で 3 を返す
- またもや Fake it!

Step 3

- Stack#size のテスト
- 「新しいスタックに 3 を push すると size は 1」

```
# test_stack.rb（一部）
class TestStack < Test::Unit::TestCase
  def test_push_and_size
    ... # ここに記述する内容を考える
  end
end
```

- Stack#size を作る (1) - 失敗

```
# stack.rb（一部）
class Stack
  def size
```

```
    return 0
  end
end
```

- 「整数を返す」形で定義
- テストの実行

```
1) Failure:

test_push_and_size(TestStack) [test_stack.rb:19]:
push increments the size.
<1> expected but was
<0>.
```

- Stack#size で 1 を返す
- しつこく Fake it!

Step 4

- Stack#size のテスト (2)
- 「新しいスタックに 3 を push すると size は 1」
- 「さらに 5 を push すると size は 2」

```
# test_stack.rb（一部）
class TestStack < Test::Unit::TestCase
  def test_push_and_size
    ... # ここに記述する内容を考える
  end
end
```

- テストの実行

```
1) Failure:

test_push_and_empty?(TestStack) [test_stack.rb:26]:
a stack with data is not empty.
<2> expected but was
<1>.
```

- Stack#initialize でサイズを格納するインスタンス変数 @size を作る
- Stack#push で @size を増やす
- Stack#size で @size を返す
- このように、テストを追加して Fake it! が通らなくすることをトライアンギュレーション (Triangulation) と言う

Step 5

- Stack#empty? のテスト (2)
- 「新しいスタックに 3 を push すると empty? は偽」

```
# test_stack.rb (一部)
class TestStack < Test::Unit::TestCase
  def test_push_and_empty?
    ... # ここに記述する内容を考える
  end
end
```

- テストの実行

```
1) Failure:

test_push_and_empty?(TestStack) [test_stack.rb:26]:
a stack with data is not empty.
<false> expected but was
<true>.
```

- Stack#empty? で @size が 0 なら true を返す

Step 6

- Stack#pop のテスト (2)
- 「新しいスタックを pop すると Stack::EmptyStackError が発生する」

```
# test_stack.rb (一部)
class TestStack < Test::Unit::TestCase
  def test_empty_pop
    ...
  end
end
```

- テストの実行

```
1) Failure:

test_empty_pop(TestStack) [test_stack.rb:30]:
to pop a empty stack raise an error.
<Stack::EmptyStackError> exception expected but none was thrown.
```

- Stack#pop で、Stack#empty? が true なら Stack::EmptyError を発生させる

Step 7

- Stack#pop のテスト (3)
- 「新しいスタックに 3 を push して 5 を push して pop すると size は 1」

```
# test_stack.rb（一部）
class TestStack < Test::Unit::TestCase
  def test_push_push_pop_and_size
    ... # ここに記述する内容を考える
  end
end
```

- テストの実行

```
1) Failure:

test_push_push_pop_and_size(TestStack) [test_stack.rb:37]:
pop decrements the size.
<1> expected but was
<2>.
```

- Stack#pop で @size を減らす
- Stack::EmptyError を定義する

Step 8

- Stack#pop のテスト (4)
- 「新しいスタックに 3 を push して 5 を push して pop すると 5 が返る」

```
# test_stack.rb（一部）
class TestStack < Test::Unit::TestCase
  def test_push_push_and_pop
    ... # ここに記述する内容を考える
  end
end
```

- テストの実行

```
1) Failure:

test_push_push_and_pop(TestStack) [test_stack.rb:43]:
pop returns the last value.
<5> expected but was
<3>.
```

- Stack#initialize で値を格納する配列を作る
- Stack#push でサイズを増やして、配列の最後の場所に値を格納する
- Stack#pop で配列の最後の場所の値を取得してサイズを減らし、値を返す

スタッククラスの実装に連結リストを使用してみよう

先の演習で作ったスタッククラスの実装を連結リストを用いたものに変更してみよう。スタッククラスに定義するメソッドは同じです。テストーコードは先の演習で書いたものをそのまま使用できると思います。

逆ポーランド記法で使える電卓を作ろう

```
$ rpn.rb 1 2 + 3 4 + \*
21
# * はシェルのワイルドカードなのでエスケープしている
```

のような引数を受け取って計算結果を返すプログラムを書いてください。逆ポーランド記法の詳しい説明は Wikipedia などを参照してください。

- 先ほど作ったスタッククラスを使用する
- テストを先に記述する
- 四則演算ができればよい
- 演算子が受け取る引数は二つまで。

以上に注意して進めてください。

ヒント

雛形:

```
# rpn.rb
def rpn(expr)
end

def main
  puts rpn(...)
end

if __FILE__ == $0
  main
end
```

```
# test_rpn.rb
require 'test/unit'
require 'rpn'

class TestRpn < Test::Unit::TestCase
  def test_1_plus_1
    assert_equal(2, rpn(...))
  end
end
```

手順は以下のような感じです。

1. 簡単な式を Fake it して結果を返す。
2. 簡単な式を少し変更して 1. で作成したプログラムが失敗するようにテストケースを追加する
3. 2. で失敗したテストケースが通るようにプログラムを修正する
4. 3. で作成したプログラムが失敗するようなテストケースを追加する
5. 4. で追加したテストケースが通るようにプログラムを修正する
6. 必要であればリファクタリングを行う
7. 以下繰り返し。満足したら終了。

今後の情報源

公式 Web サイト
http://www.ruby-lang.org/

Ruby リファレンスマニュアル刷新計画
http://doc.loveruby.net/

日本 Ruby の会
http://jp.rubyist.net/

Rubyist Magazine
http://jp.rubyist.net/magazine/

okkez weblog
http://typo.okkez.net/