

## Ruby 初級者向けレッスン第 3 回

かずひこ@株式会社 ネットワーク応用通信研究所

2005 年 10 月 8 日

### Ruby で学ぼうテスト駆動開発

- テストとは？
- テスト駆動開発の流れ
- Test::Unit の基礎
- いろんなテストメソッド
- ランナー

#### 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

#### テスト駆動開発の流れ

- 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くて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

### Ruby のテスト・フレームワーク

- 助田さんによる RubyUnit が始まり
- 今は Test::Unit として標準添付

#### Test::Unit の基礎

- 「1+1 は 2 であるべき」のテスト

```
# test_sample.rb
require 'test/unit'
class TestSample < Test::Unit::TestCase
  def test_tashizan
    assert_equal(2, 1+1, '1+1 equals 2.')
  end
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 を出す

- 実行結果

```
$ ruby test_sample.rb
Loaded suite test_sample
Started
.
Finished in 0.002019 seconds.

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

- もし失敗すると...

```
$ ruby test_sample.rb
Loaded suite test_sample
Started
F
```

```
Finished in 0.029859 seconds.
```

```
1) Failure:
test_tashizan(TestSample) [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

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

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

- 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

## 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た、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。

## 定義するメソッド

- 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::EmptyStackError が発生する。

## 演習：スタッククラスを作ろう Step 1

- empty? のテスト
- 「新しいスタックの 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?
    ...
  end
end
```

- 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.
```

- 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!) という

## 演習：スタッククラスを作ろう Step 2

- push と 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
    ...
  end
end
```

- push と pop を作る (1) - 失敗

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

- push は「引数を取って」形で定義
- 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>.
```

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

## Test::Unit の小技

- --help が効く

```
$ 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
```

## 演習：スタッククラスを作ろう Step 3

- 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
    ...
  end
end
```

- 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>.
```

- size で 1 を返す
- しつこく Fake it!

## 演習：スタッククラスを作ろう Step 4

- 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
    ...
  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>.
```

- initialize でサイズを格納するインスタンス変数 @size を作る
- push で @size を増やす
- 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グレーション (Triangulation) と言う

## 演習：スタッククラスを作ろう Step 5

- 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?
    ...
  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>.
```

- empty? で @size が 0 なら true を返す

## 演習：スタッククラスを作ろう Step 6

- 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.
```

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

## 演習：スタッククラスを作ろう Step 7

- 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
    ...
  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>.
```

- pop で@size を減らす

## 演習：スタッククラスを作ろう Step 8

- 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
    ...
  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>.
```

- initialize で値を格納する配列を作る
- 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
- 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

## Test::Unit の小技

- 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
```

## テスト駆動開発をすると...

- 全てのコードがテストを通った状態で完成する
- テストしやすい設計をするようになる
- よりよいモジュール分割をするようになる
- よりよいインタフェースを作るようになる
- 開発にリズムができて楽しくなる

## テスト駆動開発の主なアプローチ

- テストを書く (失敗)    フェイクする (成功)    リファクタリング (成功)
- テストを書く (失敗)    フェイクする (成功)    テストを足す (失敗)    リファクタリング (成功)
- テストを書く (失敗)    明白実装 (成功)

## まとめ

- テスト駆動であなたも上品プログラマ
- いろんなテストメソッドを使いこなそう
- 各種オプションと AutoRunner も便利

## 参考文献

『Test::Unit 入門』

<http://www.programmers-paradise.com/ruby/testunit.html>

『Ruby リファレンスマニュアル - Test::Unit』

<http://www.ruby-lang.org/ja/man/?cmd=view;name=Test%3A%3AUnit>

『車窓からの TDD』

<http://www.objectclub.jp/technicaldoc/testing/stack.tdd.pdf>

## 今後の情報源

公式 Web サイト

<http://www.ruby-lang.org/>

リファレンスマニュアル

<http://www.ruby-lang.org/ja/man/>

日本 Ruby の会

<http://jp.rubyist.net/>

Rubyist Magazine

<http://jp.rubyist.net/magazine/>

ふえみにん日記

<http://kazuhiko.tdiary.net/>