Perlを使ったテストの書き方

何故テストを書くのか？

コードを書いたら、テストを書きましょう。それは何故ですか？コードの正しさを保証するためです。未来の自分、もくは他人がコードを修正しても、テストが通っていればアプリケーションは正常に動く、そういうことを (ある程度) 保証するのがテストです。

テストの書き方

テストの基本は Test::More です。 perldoc を読むのがいいですが、以下で順を追って解説していきます。

基本は ok()

とりあえず、解説のためにくだらないモジュールを追加します。

lib/Calc/Simple.pm

```
 package Calc::Simple;
use strict;
use warnings;
use base qw(Exporter::Lite);

our @EXPORT = qw(add);

sub add {
    $_[0] + $_[1];
}

1;
```

新しいロジックを書いたからには、このコードの正しさを保証するテストを書かなきゃいけない。以下でさっそくテス

トスクリプトを書いていきます。 テストスクリプトは、普通

- ディレクトリ t/ 以下に置き、
- 拡張子は .t

というルールで書く。

なので、t/calc-simple.t とでも名前をつけて、以下のようなファイルを作る。実体はただの perl スクリプトです。

```
 use strict;
use warnings;
use Test::More tests => 2;
use Calc::Simple;

ok add(2, 3) == 5, '2 + 3 should be 5';
ok add(100, 0) == 100, '100 + 0 should be 100';
```

簡単ですね。`use Test::More` の行でテストの総数を宣言し、真でなければならない式を `ok()` に渡すだけです。最後の引数は各テストの説明です。これで一応、テストは完成しました。

テストを実行するには、単に Perl スクリプトとして評価してやるだけ。

```
 % perl -Ilib t/calc-simple.t
1..2
ok 1
ok 2
```

テストを実行する — prove

`prove` というコマンド (Test::Harness を入れるとついてくるらしい) を使うと、先の出力をよしなに整形してテストの結果を報告してもらうことができます。

"perl" を "prove" で置き換えるだけ。

```
 % prove -Ilib t/calc-simple.t
t/calc-simple....ok
All tests successful.
Files=1, Tests=2,  0 wallclock secs ( 0.03 usr  0.05 sys +  0.01 cusr  0.05 cs
ys =  0.14 CPU)
Result: PASS
```

テストが通過していることを確認して、このロジックが正しいと確信を得ます。

テストが失敗していたら

例えば誰かが先の Calc/Simple.pm に手を入れて

```
 package Calc::Simple;
use strict;
use warnings;
use base qw(Exporter::Lite);
```

```
our @EXPORT = qw(add);

sub add {
    $_[0] - $_[1]; # おいおい
}

1;
```

などと修正してしまった場合、prove の結果は

```
 % prove -Ilib t/calc-simple.t
t/calc-simple....1/2
#   Failed test at t/calc-simple.t line 6.
# Looks like you failed 1 test of 2.
t/calc-simple.... Dubious, test returned 1 (wstat 256, 0x100)
 Failed 1/2 subtests

Test Summary Report
                                    *
t/calc-simple (Wstat: 256 Tests: 2 Failed: 1)
  Failed test:  1
  Non-zero exit status: 1
Files=1, Tests=2,  0 wallclock secs ( 0.04 usr  0.04 sys +  0.03 cusr  0.03 cs
ys =  0.14 CPU)
Result: FAIL
```

となり、テストスクリプトは 6 行目

```
ok add(2, 3) == 5;
```

で失敗していることが分かります。こうやて、Calc/Simple.pm に手を入れた人は自分の修正が間違っていることを知ることができます。

その他のテスト関数 — is, like, is*deeply, isa*ok, diag

基本の ok() はある式が真か偽かをチェックするだけだけど、テストを実行してるともうちょっと説明がほしいときもある。のでいろいろ関数が用意されてます。

is()

```
ok $got == $expected;
```

の代わりに、

```
is $got, $expected;
```

と書ける。このテストが失敗したら

```
 #   Failed test at t/calc-simple.t line 6.
#          got: '-1'
#     expected: '5'
# Looks like you failed 1 test of 2.
```

という感じにメッセージが出てうれしいです。

like

正規表現でチェック。正規表現は qr// で渡す。("m//" と書くとその場で $_ とのマッチングが行われるのでおそらく期待と違う動作をします: See `perldoc perlop')

```
like $_->res->content, qr/^[[:ascii:]]*$/, 'Content should be ASCII';
```

is_deeply

ハッシュリファレンスとか配列リファレンス同士を比較する。

```
is_deeply $app->where, ['( foo = ? ) and ( bar != ? )', 1, 2];
```

isa_ok

あるクラスのインスタンスかどうかをチェックする。

```
isa_ok $movies, 'DBIx::MoCo::List';
```

diag

これはテスト関数じゃないけど紹介。warn 的なもの。warn よりはこちらを使おう。

ok() 系の関数は失敗すると偽を返す (たぶん) ので、こういう使い方ができます:

```
ok ref $encode_guess && $encode_guess eq 'UTF-16LE', 'encoding is utf-16le'
    or diag YAML::Dump $encode_guess;
```

Test::Class

で、Test::More を使ってきたけど、Test::Class を使うともっと便利にテストできる。これを使ったテストは

```
 package ThisTest;
use strict;
use warnings;
use base qw(Test::Class);
use Test::More;

sub sample_test : Test(2) {
    is add(2, 3), 5;
    is add(100, 0), 100;
}
```

```
__PACKAGE__->runtests; # これ忘れずに

1;
```

て感じに、Test::Class を継承したクラスを作って、Test(n) とアトリビュートをつけたメソッドの中にテストを入れていく。もちろん n はテストの数。で最後に runtests する。

Test(setup) アトリビュート

"Test(setup)" というアトリビュートのついたメソッドは、各テストメソッドが呼ばれる前に必ず実行されます。なので fixture (後述) のロードなどはここでやる。

TEST_METHOD 環境変数

Test::Class の便利なところは、TEST_METHOD 環境変数にメソッド名を入れて走らせると、そのメソッドだけをテストするとこ。テスト全体が長いけど、一部のロジックだけテストしたい、というとき重宝する。

```
% TEST_METHOD="program" prove -lv -It/lib t/app/ds-hotmovies.t
```

prove のオプション

詳しくは perldoc prove を読みましょう。

-l

-llib と一緒

-v

verbose になる

--state={save,failed}

--state=save としておくと、テストの結果を ./.prove に保存してくれる。

この後で --state=failed を指定してもう一度テストを走らせると、前回失敗したテストだけを実行してくれるので便利。

.proverc

上で書いたオプションは、~/.proverc に書いておくことで prove 時に自動的に付与してくれる。motemen の ~/.proverc は

```
--state=save -l -It/lib
```

です。基本は prove で、テストが失敗したら、成功するまで prove --state=failed を繰り返すかんじ。

テストを書くときの注意点

　ここまで簡単なテストの仕方を書いてきましたが、ここでちょっと話は変わり、テストを書くときの注意点について説明しようと思います。

　テストというものは、安全である必要があります((ただし「安全」については場合によって変わる場合もあります))。また出来る限り環境が変わっても同じような結果になるべきです。

例えば安全という観点から言えば次のようなものがあります。

- 外部にアクセスしない、害を与えない

また、結果が環境に依存しないという部分で言えば

- リソースを前提としない
- 実行中にユーザの操作が発生しない

それ以外にも

- 実行に時間がかかり過ぎない

などといったことが挙げられます。

外部にアクセスしない、害を与えない

外部にアクセスするようなテストを書くとどうなるでしょう。例えば外部のWeb APIをテストから叩くようになっていたとすると、テストの投稿がそのサービスに流れてしまい、そのサービスに迷惑をかけてしまいます。またテストで本番のデータベースにアクセスしていて、害を与えているというのも問題があります。他にも外部に依存しているとそのサービスが落ちていた時に、同時にテストも落ちてしまうということが起こります。

このように外部にアクセスするようなテストを書くと、「外部に害を与える」「外部にテスト結果が引きずられる」などといった問題が起こりえます。そのためできる限りテストでは外部へアクセスしないようにしましょう。　外部というものは色々ありますが、例えば以下のようなものがあります。

- localhost以外のmysql, memcached, MogileFS, etc...
- Web API
- メール

このあたりにアクセスせずにテストをするにはいくつかの方法がありますが、例えば

- テストの時だけDBのdsnを書き換えておきたい
    - dsnの設定をテスト用にする
    - DBIx::RewriteDSNを使って書き換える
- 外部APIにアクセスするメソッドを書き換えたい
    - 型グロブを書き換える
    - Test::Mock::Guardをつかう

のような方法があります。

しかし、あまりいろいろとテスト用に変更を加えてしまうとテストの意味がなくなってしまうので、出来る限り書き換えなどは最低限になるようにしましょう。

リソースを前提としない

テスト時にテストデータがDBに入っている前提でテストをすると、ある人はテストが通るのに、別の人は通らないといったようなことが起こります。そのため出来る限りリソースがあることを前提とせず、テスト時にデータを入れるなどのようにすると良いと思います。

実行中にユーザの操作が発生しない

実行中にユーザの操作が発生するようなテストを書いてしまった場合、テスト実行を自動化することが困難になってしまいます。そのためテスト中にはユーザの操作を必要としないようにテストを書いたほうが良いです。inputが必要なモジュールをテストする場合はその部分だけ置き換えるみたいなことをするとよい((例えばSTDINを置き換えるなど))かもしれません。

実行に時間がかかり過ぎない

実行中に時間がかかりすぎるようなテストの場合、テスト実行のために開発がストップするといったような問題が発生してしまいます。もちろん自分のPCだけでなく、CIツールなどを用いて継続的にテストを実行するべきですが、そ

れにしても1~2時間もかかるようなテストを作ってしまった場合、テスト実行が難しくなってしまいます。そのため、出来る限り実行に時間がかかりすぎないようにテストを書くよう心がけましょう。

テスト実行に時間がかかる理由とその解決法は様々ありますが、例えば

- 内部でsleepをたくさん実行していた
  - Test::Timeなどを用いて、sleep自体を置き換える
- 必要以上にテストデータを入れていた
  - ストレージデータが本当に必要なテストだけデータを入れる
  - ストレージに入っているデータに出来る限り紐づかないクラスを作る(実装的に)

などがあります。