第6章

テストの薦め

6.1 Perlとテスト

Perlといえば手軽にプログラムを書く時の代表的な言語というイメージが強いので、プログラムのテストに関して特に強いという印象は持っていないかもしれません。しかもPerl言語の信条は「怠惰」「短気」「傲慢」ですから、そもそもテストに力を入れる言語ではないように思えます。

ですが実際には、Perlほどテストとテストインフラに力を入れている言語・コミュニティは他には存在しません。それはなぜか。テストを書くことこそが前述の3条件を極めるための最も確実な方法だからです。

正しいナマケモノのススメ

「テスト」と聞くと逃げ出したくなる方も多いと思います。ひとまずそんな衝動を抑えて「なぜテストを書くのか」という点について考えてみてください。

「テストは開発の時間を取るばかりで意味がない」という意見をたまに聞きます。また、「テストを書くのは面倒だ」というぼやきもよく聞きます。もしあなたもそのように思っているのであれば、一度視点を変えてテストがもたらす効果について考えてみてください。

テストを書くという行為は、放っておけば露呈されるであろう問題を先につぶしたり、実際に問題が起こってしまった際のデバッグ時間を大幅に短縮したりすることを目的とする先行投資ですが、これほどリターンが確実に見込める投資はそうそうありません。テストは書いたら書いた分だけ、将来怠けることができるのです。

プログラム作成依頼を受ける際に、仕様書を渡されなかった経験はないでしょうか？「だいたいこんな感じで」とニュアンスを伝えられて、そのままプログラムを書ききってしまったことはありませんか？参加しているメンバーの数が多い、比較的大規模なプロジェクトでは、よりスムーズな意思疎通を目的として、もしくは問題が起こってしまった場合に備えて仕様書を用意します。しかし、完成イメージを口頭で伝えられたり、おおまかな仕様を図や表でまとめられる程度にとどまっているプロジェクトも多々あります。そんなプロジェクトに限って現場のプログラマも適当に仕様を決めて、適当にコードを書いてしまうものです。そしていざ問題が起こった時には何の対処もできない。よく聞くホラーストーリーです。

ちゃんと仕様書をもらっていても、バグを追っていたら実際のコードと違った、ということもよくあります。そんなことがあった場合、最初から仕様書と実際のコードは違っていたのでしょうか？ それとも、開発途中の何らかの仕様変更に伴って違いが生じてしまったのでしょうか？ そもそもこのような事態が起こらないようにもっと前の段階で止めることができていれば今夜は家に帰れたのに、とつぶやいたことはありませんか？

テストとは、これらのような事態に備えるために書いておく物です。もちろん、自分が本来開発するべき対象以外のコードを書くのですから、時間は余分にかかります。しかし毎日起こる日常のメンテナンスやデバッグに割かざるを得ない時間を考えた時に、それを短縮できるとすれば長い目で見たときにどれだけの時間を節約できるでしょう。今テストを書くのに使う 1 時間は、将来の 1 週間、1 か月の間ずっと行わなければいけないかもしれない業務を未然に防ぐことになるのではないでしょうか。

もしあなたが研究開発を行っているのであれば、このような問題とは関係なく毎日の業務をこなせるかもしれません。しかし、大多数のプログラマは新規開発とデバッグとメンテナンスを同時に行っています。これを読んでいるあなたもそのようなプログラマの一人ではないでしょうか。どうせプログラムを書く仕事をしているのならデバッグして残業するより、新しい何かの開発に時間を費やしたくありませんか？

自動テストの重要性

テストを書くと言っても、エクセルにテスト手順を並べるようなテストでは意味がありません。それではテストを人間が手動で行うことになり、テスト実行に時間と労力が掛かる上に毎回同じテスト結果を期待できません。一番効率の良いテストは**自動化**されているテストです。自動化さえしてあればテストを実行する際のノウハウを担当者間で伝えあったり、そもそも人手によりテストを実行するという手間をかけずにすみます。

テストを人間にまかせた場合は担当者がどんなに優秀な人でも手順書にある指令を愚直に指示通り間違いなく繰り返すようなことはできません。繰り返していくうちに手を抜くこともあるでしょうし、単純に手順を忘れたり間違ったりすることもあるでしょう。その一方、一旦作成された自動テストは毎回必ず指示された通りの動作を愚直に実行し、テストを実行します。きちんとしたテストを書けば毎回寸分変わらぬ環境と引数パターンを使い、人間が絶対にできない頻度で指示された通りの確認が行えます。また担当者が代わった場合でもテストスクリプトの実行の仕方さえ分かればあとの反復作業はテスト本体に任せることができます。

筆者が勤務していた経験のある企業では時間ごとに自動的にテストが走り、問題があった場合はそれが誰がいつ行った変更のせいなのかを自動的につきとめてメールで連絡するというシステムがありました。この存在により開発者は自分の間違いを遅くとも 1 時間強で知り得ましたし、なにより怖じけずにコードを変更することができたのです。いざバグが混入してしまった場合でも、開発者達はそれまでに蓄積されたテストの種類と範囲から、バグが発現する可能性のある状態を特定するのに、ある程度探索範囲を狭めることが容易にできました。

これと対照的にテストがない企業では、問題が起こるとバグの位置特定は完全にそれを担当した人の勘と経験に頼ることになりがちです。どこまでが「安全」なのか事前に教えてくれるテストがなけ

れば、あとはしらみつぶしに対応していくしかありません。バグが直せたとしても、簡単に自動テストにテストを追加できる仕組みがすでになければ、そのノウハウも開発者個人の知識として蓄積されるのみで、次回から同様の問題を事前に察知できるかどうかは運任せです。

このように、自動テストのあり／なしは開発効率に大きく影響します。バグや仕様変更は必ず起きる事象なので防ぎようがなく、テストを全く書かないと後々自分を含めた誰かが必ず貧乏くじを引かされることになります。逆にテストがあれば相当な量の労力を節約することができます。テストを書かないナマケモノは実際には後で働かされるので真のナマケモノとは言えません。本当のナマケモノは将来自分が働かなくて済むように先に準備をしておくのです。

Perl コミュニティは、他言語と比べてもかなりテストを重要視しています。それは、Perl の基本思想の一つが「(可能な限り) 怠けること」だからです。テストを書くのはつらい業務だと思われがちですが、優れたプログラマは、自分の書いたプログラムのデバッグ作業に過剰に時間を費やすことこそが自分の能率を落とす原因だと知っています。彼らは、不毛な作業をなるたけ行わないためにテストを書くのです。

かといって怠けるために苦労してテストフレームワークを導入するのでは意味がありません。Perl ならテストを走らせるところまでの準備が簡単にできるのです。

はじめの一歩

具体的なテストを書き始める前に、テスト対象を CPAN 形式に整えましょう（P.288 参照）。CPAN の章でも触れていますが、CPAN 形式で開発を進めていれば、テストを走らせるための基本機能もすでに組み込まれているので、特に新しくテスト用のツールを実装する必要がなく、とても簡単に自動テストを実行するまでの準備を行えます。

ここでは Example::Software というモジュールをテストすると仮定します。CPAN の章で紹介したように、基本構成は以下のようになります：

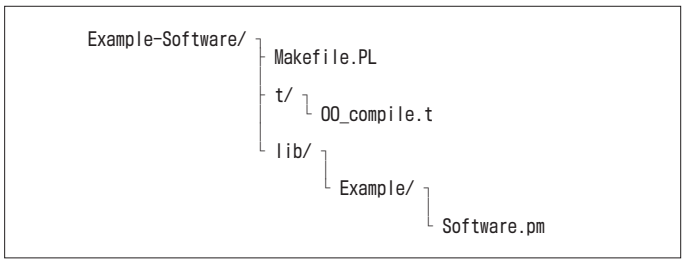

図 6.1 ディレクトリ構成の例

lib/Example/Software.pm

```
package Example::Software;
use strict;
our $VERSION = '0.00001';
# ... Example::Software の実装 ....
1;
```

Makefile.PL

```
use strict;
use inc::Module::Install;

name('Example-Software');
all_from('lib/Example/Software.pm');

WriteAll;
```

t/00_compile.t

```
# t/00_compile.t
use strict;
use Test::More;
plan( tests => 1 );

use_ok("Example::Software");
```

以上の 3 ファイルを用意すると、Example::Software モジュールに構文エラーがないか確認するテストを走らせる準備が整います。この構成を揃えた後、以下のコマンドを実行するだけで lib ディレクトリのモジュールが処理され、t/ ディレクトリ以下のテストが実行されます。難解な XML 定義ファイルや特殊なツールは一切不要です：

```
> perl Makefile.PL
> make
> make test
```

Column テストファイルの指定

Makefile.PLで明示的にテストを指定しない場合は、（それぞれのファイルが存在すれば）Makefile.PLと同じディレクトリ内のtest.plおよびt/ディレクトリ直下の.tで終わる全てのファイルを、テストファイルとして認識してくれます。ただし、t/ディレクトリ以下のサブディレクトリについては、再帰的にテストファイルを探してくれませんので明示的に指定する必要があります。Module::Installを使用している場合はtest()関数をMakefile.PLで使用することで実行するべきテストファイルを指定できます：

```
use strict;
use inc::Module::Install;

name('Example-Software');
all_from('lib/Example/Software.pm');

# t/*.t は「tディレクトリ内の .t で終わるファイル」
# t/*/*.t は「t/ディレクトリ内の全てのディレクトリ内 .t で終わるファイル」
# をそれぞれ指定
test('t/*.t t/*/*.t');

WriteAll;
```

make testを実行した結果は以下のようになります：

```
> make test
cp lib/Example/Module.pm blib/lib/Example/Module.pm
PERL_DL_NONLAZY=1 /usr/local/bin/perl "-MExtUtils::Command::MM" "-e"
    "test_harness(0, 'blib/lib', 'blib/arch')" t/*.t
t/00_compile.....ok
Files=1, Tests=1, 0 wallclock secs
    ( 0.03 usr  0.00 sys +  0.36 cusr  0.04 csys =  0.43 CPU)
Result: PASS
```

t/00_compileの行に「ok」とありますので、t/00_compile.tというテストファイルで定義されたテスト内容についてはパスしました。テストファイルは一つしかなかったので、全体としてもパスとなります。よって、全体の結果として最後に「Result: PASS」と出力されました。

これ以降はExample::Softwareにどのような変更を加えたとしても、その後にmake testと打ち込む

だけで 構文エラーがないか即座に判断することができます。このテスト一個だけではそれほど強力とは言い難いですが、このようにすでに Perl が動作する環境であれば、ただファイル構成を揃えるだけで容易に自動テストを実行する環境が作れるのが、他の言語と比べて圧倒的に Perl でのテストが楽な所以と言えます。

新規テストを追加する場合は単純にテストファイルを追加していきます。テストファイルは Test::More、もしくはそれと同様の形式の出力を行うスクリプトであれば内容はどのようなものでも問題はありませんが、慣例としてファイル名の命名規則があります：

```
t/00_compile.t
t/01_basic.t
t/02_exception.t
...
```

上記のように先頭に番号をつけ、.t を拡張子として使うのが一般的です。これは CPAN 形式のテストは多様な環境下で動作することを前提としているので、t/ ディレクトリ内のファイル名を読み込む際に、読み込まれるファイルの順番が必ずしも全ての環境で一致するとは限らないため、一番処理が影響しない（はずの）数字を先頭に組み込むことによって実行順序が一緒になるからです。

なお、テストファイルのリストは Makefile 作成時に読み込まれるので、新規テストを Makefile に認識してもらうには一旦 Makefile を作り直す必要があります：

```
> make clean
> perl Makefile.PL
> make
> make test
```

新規ファイルの追加、ファイルの削除等のファイルレイアウト構成の変更があった場合は make clean を実行する癖をつけておくとよいでしょう。

Column prove

make test と打てば簡単にテストを走らせることができるのが CPAN 形式の強みの一つですが、make test はどの環境でも動作するように必要最低限の機能しか実装していません。もし、もう少し高機能なテストツールが欲しい場合は prove コマンドの導入を検討してみてください。

prove コマンドは Test::Harness というディストリビューションに梱包されているツールです。現時点では prove コマンド自体は最初から Perl 本体と同梱されているわけではありませんので、CPAN から最新版をインストールしてください（Perl 5.8.9 RC1 には Test::Harness の少し古いバージョンが同梱されています）：

```
> cpan Test::Harness
```

prove に実装されている機能のうち、筆者が重宝しているのはテスト結果のカラー出力とテスト実行時間の計測です。ちょうどこの原稿を執筆中に拙作の Google::Chart に関してバグ報告があったので、テストを追加し実行してみた結果が以下です（紙面上モノクロなので、ぜひ実際に動作させてカラー出力を確認してみてください）：

```
daisuke@beefcake-7 trunk$ prove  --timer
[10:24:59] t/00_compile.........ok      637 ms
[10:25:00] t/01_size............ok      307 ms
[10:25:00] t/02_marker..........ok      353 ms
[10:25:01] t/10_text............ok      320 ms
[10:25:01] t/11_extended........ok      310 ms
[10:25:01] t/20_color...........ok      461 ms
[10:25:02] t/21_legend..........ok      460 ms
[10:25:02] t/22_grid............ok      461 ms
[10:25:03] t/50_line............ok      473 ms
[10:25:03] t/51_bar.............ok      481 ms
[10:25:04] t/52_pie.............ok      474 ms
[10:25:04] t/53_qrcode..........ok      452 ms
[10:25:05] t/60_axis............ok      464 ms
[10:25:05] t/61_fill............ok      481 ms
[10:25:06] t/90_env_proxy.......1/4
#   Failed test 'http proxy should be set'
#   at t/90_env_proxy.t line 24.
#          got: undef
#     expected: 'http://localhost:3128'
# Looks like you failed 1 test of 4.
[10:25:06] t/90_env_proxy....... Dubious, test returned 1
    (wstat 256, 0x100) Failed 1/4 subtests
```

```
[10:25:06] t/99-pod-coverage....skipped: Enable TEST_POD
    environment variable to test POD
[10:25:06] t/99-pod.............skipped: Enable TEST_POD
    environment variable to test POD
[10:25:06]

Test Summary Report
-------------------
t/90_env_proxy   (Wstat: 256 Tests: 4 Failed: 1)
  Failed test:  3
  Non-zero exit status: 1
Files=17, Tests=186,  7 wallclock secs
    ( 0.12 usr  0.07 sys +  6.12 cusr  0.45 csys =  6.76 CPU)
Result: FAIL
```

上記のうち "Failed test..." で始まる部分が赤く表示されるはずです。実行スピードの表示はパフォーマンスチューニングの必要性の判断に役立ちますし、エラーは赤で表示されるので、どこでエラーが発生したか非常に分かりやすくなります。

本書では make test しか使用しませんが、他にもテストを並列で走らせることができる等、使いようによってはテストの幅が広がるかもしれません。

6.2 テストの書き方

テストを実行する環境は整いましたが、テストの書き方が分からなければどうしようもありません。まずは前節で紹介した簡単なスクリプトの内容から、Perl でのテストの基本を紹介してみましょう：

```
# t/00_compile.t
use strict;
use Test::More;
plan( tests => 1 );

use_ok("Example::Software");
```

このテストでは Test::More モジュールを使用してテストを書いています。Perl でテストを書くためのモジュールは Test.pm を皮切りに Test::Simple, Test::Class 等、様々な形態のものが存在しますが、現在 Perl に標準で付属されており、なおかつ一般的に使用頻度の高いテスト用ユーティリティが揃っているのは Test::More モジュールです（標準モジュール化は perl 5.6.2 より）。use Test::More とすると、

plan(), ok(), use_ok() 等の関数が使用できるようになりますので、先に読み込んでおきます。

Perl テストではテストを開始する前に、まずそのテストスクリプト内で何個のテストを実行するのか宣言する必要があります。これはテストの途中でなんらかのバグにより中断、もしくは exit(0) を呼ばれてしまった場合など、明示的なエラー以外の理由でテストが途中で終了してしまったことを判別できるようにするためです。この宣言を plan() 関数で行っています。ここではテストを 1 個だけ実行する予定なので tests => 1 と宣言しています。

そして実際のテスト内容は use_ok() 関数を使用し、Example::Software モジュールを正しく読み込めるかどうかテストしています。読み込みに失敗する（構文エラーなどがある）と、テストは失敗したと見なされます。

Example::Software にはモジュールが一つしか存在しませんでしたので、このテストはこれで完成ですが、Test::More にはこの他にも様々な関数が用意されており、簡単にアプリケーションのテストを追加することができます。比較的頻繁に使う Test::More 関数には以下のようなものがあります：

```
ok( $true_or_false, '値が真か確認' );

# $class が use できるかどうか確認
use_ok( $class, @args );

# $class_or_file が require() できるか確認
require_ok( $class_or_file );

# $object がクラス名のインスタンスか確認
isa_ok( $object, 'クラス名' );

# $class が @methods に定義された関数を実装しているか確認
can_ok( $class, @methods );

is( $value, '期待値', '$value が期待値と一致するか確認' );
isnt( $value, '期待しない値', '$value が期待しない値と一致しないことを確認');

like( $value, qr/正規表現/, '$value が正規表現と一致するか確認' );
unlike( $value, qr/正規表現/, '$value が正規表現と一致しないことを確認' );

is_deeply( $value, [ '構造体' ], '$value が構造体と一致するか確認' );
is_deeply( $value, { key => '構造体' }, '$value が構造体と一致するか確認' );
```

これらの関数は実際にテストを行う関数で、確認すべき値と期待値を与えてそれぞれが一致するかどうか確認するために使います。テストしたい内容と照らし合わせて最適なものを選ぶようにしてください。