use Perl::Object 0;

文 編集部 mitty

お詫びと訂正[*1]

前回の記事(16 号)の「目次」では、『次は第 1 章の「use による既存モジュールの活用と package 宣言」から始める』としていましたが、多角的に検討した結果、まずは初歩的な導入部分を用意して、第 1 章は年度を改めてから始めた方が良かろう……という結論に至りました。従いまして今回は第 0 章と称して、「Perl のデータ構造および CPAN からのモジュール導入」を扱います。

なお、掲載されているソースコードは、以下の処理系で確認しております。

- ActivePerl 5.12.2 build 1203 x86 および x64 on Windows 7 x64
- perl 5.10.1-8ubuntu2 x64 on Ubuntu 10.04.1 LTS（Lucid Lynx）x64

予定

第 0 章 Perl のデータ構造および CPAN からのモジュール導入（今回）
第 1 章 use による既存モジュールの活用と package 宣言
第 2 章 Perl でのクラス・インスタンス・メソッド
第 3 章 既存モジュールの拡張・継承
第 4 章 POE: Perl Object Environment の紹介
第 5 章 POE を使ったマルチタスキング
第 6 章 POE によるイベントドリブン生活
第 7 章 並列クローラの作成

Perl における変数の型

Perl で書かれたコードに頻繁に出てくる「$」「@」「%」などですが、これは ***Sigil***[*2] と呼ばれていて、変数の「型」を大まかに表しています。

「$」は「スカラー変数」、「@」は「配列変数」、「%」は「連想配列変数(ハッシュ)」と呼ばれ、右の「ソースコード 1」を見て貰うと分かるかと思いますが、スカラー変数は値を一つ、配列変数は値を複数持ちます。これ以外にもファイルやディレクトリを扱う際に用いる「ハンドル」や「サブルーチン」であることを表す「&」、Sigil の親玉である「型グロブ」を示す「*」などがありますが、追々触れていくことにしましょう。

ソースコード 1

use0-01.pl

```
1:  #! /usr/bin/perl -w
2:
3:  use strict;
4:  use warnings;
5:  use feature qw(say);
6:
7:  my $hoge = 0;
8:  my @hoge = (0, 1, 2, 3, 4, 5, 6, 7);
9:  my %hoge = (
10:     zero => 0,
11:     one  => 1,
12:     two  => 2,
13: );
14: say $hoge;
15: say join(", ", @hoge);
16: say "0 != 1" if ($hoge != $hoge[1]);
17: foreach my $key (keys %hoge) {
18:     say "$key => $hoge{$key}";
19: }
```

*1 お詫びと訂正：某ふぁい氏の記事とはたぶんきっとおそらく関係がありません。
*2 Sigil：元々は魔術などの分野で使われる「紋章」といった意味を持つようです。カタカナだと「シジル」になるらしい。なお、「プログラミング Perl vol.1」では"funny character"と書かれているが、そう呼ばれているのを今知ったくらいに使われているところを見ない……。

「ソースコード 1 の実行結果」で示されるように、「$hoge」と「$hoge[1]」は違う変数です。そして「$hoge{one}」もまた違う変数(の要素)を指しています。

なお、ここまでで疑問に思われた方も居るかと思いますが、Perl5 までの Perl にはいわゆる「int」「string」のような、データそのものの種類を表す型はありません。

ソースコード 1 の実行結果

```
$ ./use0-01.pl
0
0, 1, 2, 3, 4, 5, 6, 7
0 != 1
one => 1
zero => 0
two => 2
```

『**例1)「==」と「eq」の違い**』を見て下さい。***[Ctrl+D]***までの太字の部分が入力したコード、***[Ctrl+D]***より下が出力結果です。出力結果から分かるように、「==」演算子はオペランドを数値として比較し、「eq」演算子は文字列として比較します。文字列を数値として評価すると「0」として扱われるため、「**例1**」のような結果となります。

例 1)「==」と「eq」の違い

```
$ perl -w
use feature qw(say);
say (("0" == 0) ? "true" : "false");
say (("0" == 0.0) ? "true" : "false");
say (("0" == "0.0") ? "true" : "false");
say (("0" eq 0) ? "true" : "false");
say (("0" eq "0.0") ? "true" : "false");
[Ctrl+D]
true
true
true
true
false
```

変数に型が無い、ということはその分記述に曖昧さが生じますが、むしろ慣れるとこの自由度の高さからストレス無く書けるようになるでしょう。

なお、以降のソースコード中に出てくる「$_」や「@_」などの、Perl 特有の特殊変数については、前回の記事[*3] で一部触れているので、説明を省略している箇所があります。

スカラーコンテキストとリストコンテキスト

Perl における変数の型として、「スカラー変数」と「配列変数[*4]」があることは既に述べましたが、Perl において「スカラー」および「リスト」の 2 単語は変数の型にとどまらず、「コンテキスト」と言われる文字通りコード中で式がどのように評価されるか、という意味でも使われます。

---

*3 前回の記事：http://lab.mitty.jp/word/use_Perl_Object/use_Perl_Object.pdf　あるいは、WORD16 号「侵略されたいでゲソ号」をご覧下さい。

*4 配列変数：より正確には、値が複数代入できる変数を、単純な配列変数として扱うか連想配列変数として扱うかによって、取り出される値が違うだけ過ぎません。

```
%hoge = (zero => 0, one => 1, two => 2); for (%hoge) { print $_, ", " };
# => one, 1, zero, 0, two, 2,
```

「%hoge」も上記で示されるように単純な配列として扱えます。ただし、「@hoge」とは異なり(定義時や要素追加などの)順番は保存されません。

「ソースコード 2」の 9 行目を見て下さい。配列変数である「@hoge」を、スカラー変数である「$fuga」へ代入する場合、「@hoge」は「スカラーコンテキスト」で評価されます。配列変数をスカラーコンテキストで評価した場合、得られる値はリストの「要素数」となり、この場合「$fuga」には「8」が代入されます。

一方、10 行目では「@hoge」は「リストコンテキスト」で評価され、リストが返されます。

ソースコード 2

```
use0-02.pl
1:   #! /usr/bin/perl -w
2:
3:   use strict;
4:   use warnings;
5:   use feature qw(say);
6:
7:   my @hoge = (0, 1, 2, 3, 4, 5, 6, 7);
8:
9:   my $fuga = @hoge;
10:  my @fuga = @hoge;
11:  say join(", ", $fuga);
12:  say join(", ", @fuga);
```

ソースコード 2 の実行結果

```
$ ./use0-02.pl
8
0, 1, 2, 3, 4, 5, 6, 7
```

では、次の「例 2）スカラー変数とリストコンテキスト」はどうでしょうか。

「$hoge」自身はスカラー変数ですが、「()」で囲むことによってリストコンテキストとして扱われ、「@hoge」の要素の一番目「0」が代入されます。

例 2）スカラー変数とリストコンテキスト

```
$ perl -w
($hoge) = @hoge = (0, 1, 2, 3, 4, 5, 6, 7); print $hoge;
[Ctrl+D]
0
```

「コンテキスト」には他に「ブール値コンテキスト」などがあり、これは真偽値として評価される場合が該当します。「ブール値コンテキスト」は「スカラーコンテキスト」の一種なので、変数に以下で述べる特定の値以外が含まれている限り、真となります。

「ソースコード 3」において「@hoge」から「pop」関数によって一つずつ要素が取り出されますが、要素がある限り while ループからは抜け出さないことになります。

ソースコード 3

```
use0-03.pl
1:  #! /usr/bin/perl -w
2:
3:  use strict;
4:  use warnings;
5:  use feature qw(say);
6:
7:  my @hoge = (0, 1, 2, 3, 4, 5, 6, 7);
8:
9:  while (@hoge) {
10:     say pop @hoge;
11: }
```

実行結果

```
$ ./use0-03.pl
7
6
5
4
3
2
1
0
```

Perl5 において「真」とならない、つまり「偽」として扱われるデータは以下の通りです。

- 0
- 空文字列
- 空リスト
- undef（未定義値）

空ではない任意の文字列は、数値として評価した場合には「0」と評価されるため、「偽」となります。

注意して欲しいのですが、「例 3）スカラー変数とリスト変数の真偽値」で示されるとおり、ブール値コンテキストで評価すると「$hoge = 0」は「偽」となりますが、「@hoge = (0)」は要素が一つあるため「真」となります。

例3）スカラー変数とリスト変数の真偽値

```
$ perl -w
@hoge = ($hoge = 0); print "true" if @hoge;
[Ctrl+D]
true
```

リファレンス

変数を受け取り値を 2 倍にして返す関数を考えます。受け取る変数がスカラー変数か配列変数か分からないとすると、以下の「ソースコード 4」および「ソースコード 5」のように分けて書くことがまず考えられます。

ソースコード4

```
use0-04.pl
1:  #! /usr/bin/perl -w
2:
3:  use strict;
4:  use warnings;
5:  use feature qw(say);
6:
7:  my $hoge = 1;
8:
9:  say scalardouble($hoge);
10:
11: sub scalardouble {
12:     my $scalar = shift;
13:
14:     $scalar *= 2;
15:     return $scalar;
16: }
```

ソースコード4の実行結果

```
$ ./use0-04.pl
2
```

ソースコード5

```
use0-05.pl
1:  #! /usr/bin/perl -w
2:
3:  use strict;
4:  use warnings;
5:  use feature qw(say);
6:
7:  my @hoge = (0, 1, 2, 3);
8:
9:  say join(
10:     ", ",
11:     listdouble(@hoge)
12: );
13:
14: sub listdouble {
15:     my @list = @_;
16:
17:     return map {
18:         $_ * 2
19:     } @_;
20: }
```

ソースコード5の実行結果

```
$ ./use0-05.pl
0, 2, 4, 6
```

しかし、扱いたいデータが常に、スカラー変数あるいは配列変数のどちらかであるとは限りません。

ここで、「ソースコード 6」で示されるように「¥(円記号あるいはバックスラッシュ)」演算子を用いて、引数となる変数の「リファレンス」を渡すようにすると、スカラー変数か配列変数かで関数を分ける必要が無くなります。

例4) 2次元配列(失敗)

```
$ perl -w
use feature qw(say);
@hoge = ( (1, 2), (3, 4) );
say "no exist" unless exists $hoge[1][1];
say $hoge[3];
[Ctrl+D]
no exist
4
```

例5) 2次元配列

```
$ perl -w
@hoge = ( [1, 2], [3, 4] );
print $hoge[1][1];
[Ctrl+D]
4
```

また、いわゆる「n 次元配列」のような変数を扱いたい場合を考えてみましょう。単純に考えると「例 4) 2 次元配列(失敗)」のようになりますが、これは単純な一次元配列と同じになってしまいます。

実際に「$hoge[1][1]」に値を代入するには「例 5) 2 次元配列」のように書く必要があります。

「()」だった部分が「[]」に変わっていますが、これにより「『配列のリファレンス』を要素に持つ配列」が「@hoge」に代入されているのです。

さて、ここで「『ソースコード 6』では『${...}』とか『@{...}』、あるいは『$$ret』といった変な Sigil があるのに、リファレンスが含まれるはずの『例 5)』ではそれが見あたらない」と気づいた方は鋭いです。

実は、「$hoge[1][1]」は「$hoge[1]->[1]」の省略形で、リファレンスの中身を取り出す「デリファレンス」演算子である「->」が隠れているのです。

ソースコード6

```
use0-06.pl
1:  #! /usr/bin/perl -w
2:
3:  use strict;
4:  use warnings;
5:  use feature qw(say);
6:
7:  my $hoge = 1;
8:  my @hoge = (0, 1, 2, 3);
9:
10: my $ret = double(¥$hoge);
11: say $$ret;
12: my @ret = double(¥@hoge);
13: say join(
14:     ", ",
15:     @{ $ret[0] },
16: );
17:
18: sub double {
19:     my $reference = shift;
20:
21:     my $type = ref $reference;
22:     if ($type eq "SCALAR") {
23:         ${$reference} *= 2;
24:     }
25:     elsif ($type eq "ARRAY") {
26:         @{$reference} =
27:             map { $_ * 2 }
28:                 @{$reference};
29:     }
30:     else {
31:         warn "something wrong";
32:     }
33:     return $reference;
34: }
```

ソースコード6の実行結果

```
$ ./use0-06.pl
2
0, 2, 4, 6
```

「例 6) 2 次元配列の構造」を見て頂くと、「@hoge」の構造が理解頂けるかと思います。

つまり、「$hoge[1][1]」というコードは、ARRAY のリファレンスである「$hoge[1]」を「->」演算子でデリファレンスし、得られた配列の 2 番目の要素「$hoge[1]->[1]」にアクセスしているのです。

「ソースコード 6」では「->」演算子ではなく、「${*スカラーリファレンス*}」や「@{*配列リファレンス*}」といった形でアクセスしているわけです。

なお、文法的に曖昧さが無い場合は「${...}」や「@{...}」の「{」と「}」は省略することが可能で、「$$ret」のように書けます。

例6) 2次元配列の構造

```
$ perl -de 0
@hoge = ( [1, 2], [3, 4] );
print join(", ", @hoge);
[Ctrl+D]
ARRAY(0x11efdf0), ARRAY(0x129dce0)
```

リファレンスによる複雑なデータ構造

「[]」による配列のリファレンス以外に、「{}」によるハッシュのリファレンス、「sub { ... }」によるコードリファレンスなどがあります。これによって階層化された複雑なデータ構造が表現できます。

例7) Twitter用botの設定ファイルをロードした後、Data::Dumperを通して出力したもの

```
{
'hashtag' => [
    '#perlbot',
    '#bot'
    ],
'allow' => {
    'screen_name' => [
        'twitter'
        ]
    },
'mail' => {
    'pickup' => [
        'search',
        'mention'
        ],
    'to' => [
        'info@example.com',
        ],
    'subject' => 'new tweets for retweetbot are found',
    'from' => 'retweetbot@twit.example.jp (Cron Daemon)',
    'server' => 'localhost',
    'contenttype' => 'text/plain; charset="ISO-2022-JP"'
    },
}
```

「例 7)で示されるデータ構造が「$conf」に代入されている場合、「@{ $conf->{mail}{to} }」にアクセスすると、「(info@example.com)」という配列が得られます。

CPAN

さて、ここまでで Perl のデータ構造に関してはある程度理解して頂けたかと思います。文法なども掘り下げると色々と楽しいのですが、きりがないのでこの辺にして、Perl でコーディングするときに大変便利な CPAN について解説しようと思います。

CPAN とは「Comprehensive Perl Archive Network」の略語で、Ruby における RubyGems や PHP における PEAR に近いでしょうか。とはいうものの、あまりにも規模が大きい[*5]ため、大抵のライブラリやちょっとしたツールであれば車輪の再発明をするまでもなく既に CPAN に登録されていることが多いです。

CPAN に登録されているモジュールは、 http://search.cpan.org/ から検索することが出来ます。たとえば Twitter に関する（と思われる）モジュールは 118 件（2011/01/21）あるようです。

CPAN
Home · Authors · Recent · News · Mirrors · FAQ · Feedback
twitter
in All CPAN Search
Results 1 - 10 of 118 Found
1 · 2 · 3 · 4 · 5 · 6 · Next >> Page Size: 10 20 50 100

Net::Twitter
A perl interface to the Twitter API
Net-Twitter-3.14002 (6 Reviews) - 02 Nov 2010 - Marc Mims

Net::Twitter::Lite
A perl interface to the Twitter API
Net-Twitter-Lite-0.10003 (3 Reviews) - 27 May 2010 - Marc Mims

Twitter::Badge
Perl module that displays the current Twitter information of a user
Twitter-Badge-0.02 - 09 May 2008 - Arul John

Dancer::Plugin::Auth::Twitter
Authenticate with Twitter
Dancer-Plugin-Auth-Twitter-0.02 - 17 Jan 2011 - Alexis Sukrieh

Net::Twitter::Role::API::REST
A definition of the Twitter REST API as a Moose role
Net-Twitter-3.14002 (6 Reviews) - 02 Nov 2010 - Marc Mims

Twitter::Shell
Twitter From Your Shell!
Twitter-Shell-0.03 - 08 May 2007 - Daisuke Maki

Catalyst::Authentication::Credential::Twitter
Twitter authentication for Catalyst
Catalyst-Authentication-Credential-Twitter-0.01001 - 06 Dec 2009 - Jesse Stay

Net::Twitter::Antispam
Making Twitter usable
Net-Twitter-Antispam-0.02 - 27 Jun 2009 - James Laver

CPAN モジュールの使い方

詳しくは今後の記事で再度触れますが、簡単に説明すると、「`use Module::Name;`」でモジュールをロードし、「`$instance = Module::Name->new( aruguments );`」で「インスタンス変数」を新しく用意して、そのインスタンス変数を通じてメソッド呼び出し、というのが基本的な使い方になります。モジュールによっては「クラスメソッド」にあたる関数が用意されていることもあります。

独断と偏見で、よく使われるモジュールを紹介してみます。

・LWP::UserAgent

LibWWWPerl つまり WWW 関係の Perl ライブラリのうち、Perl をウェブクライアントとして使う際に必要となる機能をまとめたモジュールです。wget コマンドをイメージするとどう使うか把握しやすいかも知れません。

http://search.cpan.org/ に接続を試行し、成功すれば得られたコンテンツを、失敗した場合はステータスを表示するコード「lwp-ua.pl[*6]」を次に示します。

---

*5 規模が大きい：「89356 Modules, 8717 Uploaders」だそうです。(2011/01/21 08:55 JST)

*6 lwp-ua.pl：http://search.cpan.org/perldoc?LWP::UserAgent の「SYNOPSIS」ほぼそのままです。

```
lwp-ua.pl
1:  #! /usr/bin/perl -w
2:
3:  use LWP::UserAgent;
4:
5:  my $ua = LWP::UserAgent->new;
6:  $ua->timeout(10);
7:  $ua->env_proxy;
8:
9:  my $response = $ua->get('http://search.cpan.org/');
10:
11: if ($response->is_success) {
12:     print $response->decoded_content;  # or whatever
13: }
14: else {
15:     die $response->status_line;
16: }
```

・Data::Dumper

「Data::Dumper 先生」と敬称で呼びたくなるほど便利なモジュールです。これを use しておくと、既に出てきたように「Dumper」というクラスメソッドに引数を渡すことで、データ構造を再現した形で中身を出力してくれます。print 文などに渡せば OK。

ウェブページをスクレイピングしたりする際に、メソッドから返ってくるオブジェクトの構造がよく分からなかったりする場合は先生の出番です。Twitter 関連のモジュールである「Net::Twitter::Lite」を使って RT ボットを作る過程で、返値を実際に出力したものが http://lab.mitty.jp/trac/lab/wiki/Dev/Twitter/Perl/Dumper にありますので参考まで。

・Web::Scraper

最近その存在を知ったのでまだあまり使いこなせていませんが、非常に強力なスクレイピングモジュールです。scraper というコマンドラインツールも付いてきます。同じような目的では、「HTML::TreeBuilder」や「WWW::Mechanize」などといったモジュールもありますが、「Web::Scraper」はそれらに比べ少ないコード量で済む設計になっているようです。

この他にも、本当に多種多様なモジュールがあります。是非自分で探してみて下さい。

CPAN モジュールの導入

「LWP::UserAgent」「Data::Dumper」は CPAN からダウンロードしなくても、perl 本体の配布パッケージと一緒になって導入されているはずです。それ以外の「Web::Scraper」や「Net::Twitter」などは自分で導入する必要があります。

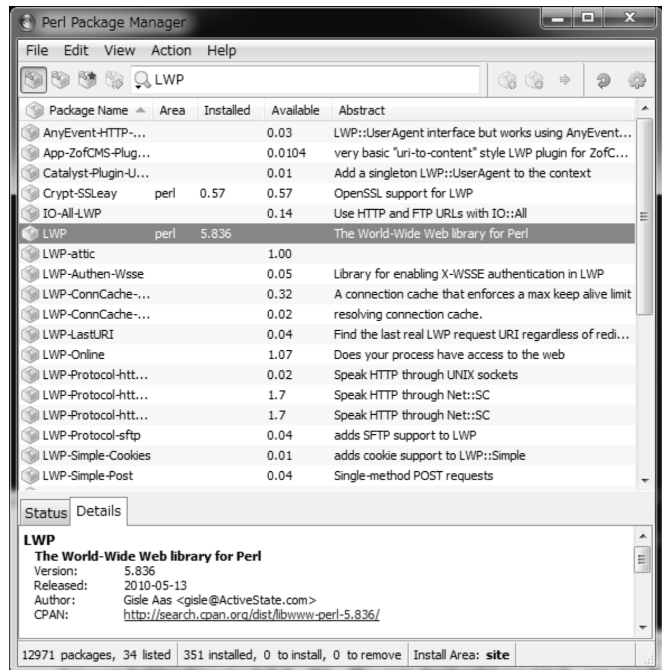
このとき、導入方法には大まかに次の三つがあります。

- CPAN 上のモジュールのページから tarball をダウンロードし、make する
- cpan コマンドにモジュール名を渡し、インストールする
- その他、OS 付属のパッケージ管理ツールを用いる

後に挙げた方法ほど容易になります。cpan コマンドに関してはウェブ上に文献がいくらでもありますし、筆者自身普段はパッケージ管理ツールを用いているので、ここではパッケージ管理ツールを用いた導入方法を簡単に説明しておきます。

・Windows

Windows 用のメジャーな Perl のバイナリは「ActivePerl」という名前で配布されています。これには「ppm」という、GUI ツールが付属しています。デフォルトのインストール先は「C:¥Perl¥bin」です。



cpan コマンドなどもそうですが、「ppm」もモジュール間の依存関係を自動で解決して必要なモジュールを追加で導入してくれます。また、既に build 済みの形でダウンロードされるため、cpan とは違い make を行わず、(Perl プロジェクトではない)外部のライブラリを必要とすることがありません。

その代わり、CPAN に置かれているパッケージの全てが網羅されているわけではなく、特に GNU のライブラリなどに依存しているモジュールは無いことが多いです。

・Ubuntu

aptitude または apt-get で導入することが出来ます。パッケージ名にちょっと癖があり、たとえば「WWW::Google::Calculator」であれば「aptitude install libwww-google-calculator-perl」で導入できます。こちらは ppm よりも網羅されておらず、2000 パッケージ強しかないようです。

Perl のモジュールは依存関係が入れ子になっていることが多く、あるモジュールを導入する際に、それが依存しているモジュールを導入しようとするとさらに依存が発生して……、と予想以上にモジュールの導入に時間が掛かることが良くあります。このため、導入対象のモジュールそのものは ppm や aptitude などのパッケージ管理ツールには載っていない場合でも、あらかじめ依存するモジュールを導入してから対象モジュールを cpan コマンドで導入すると、比較的楽に入れることが出来るようです。

最後に

如何でしたでしょうか。かなり端折った部分も多く、まだまだ Perler として力不足を感じていますが、この記事が楽しい Perl 生活のきっかけにでもなれば、幸いです。

ハイパー添削タイム(あるいは蛇足)

前回の記事に掲載しました mycat に対して、ありがたいご指摘と、添削があります。

- http://twitter.com/uasi/status/4133434205143040
  今回の WORD に print for <>; って Perl のワンライナーが載ってたけど print <>; でもいいんじゃないか。どっちもリストコンテキストだから print する前にファイル全体を読み込むはず。それとも for <> は最適化されんのかな。
- http://twitter.com/uasi/status/4133886183350272
  1 行ずつ読むなら print while <>; すべき。

while の場合山括弧演算子によって、ファイルから一行ずつ読み込まれ画面に出力されますが、for/foreach ではリストコンテキストで評価されるため、ファイルから一度に全て読み込んでしまいます。前回の例では問題にはなりませんでしたが、読み込み対象が巨大なファイルの場合は問題が生じます。

右端のグラフは上段が CPU 使用率、下段がメモリ使用量です。というか Windows のタスクマネージャーです。

mycat.pl
```
1:  #! /usr/bin/perl -w
2:
3:  use strict;
4:  use warnings;
5:
6:  print for <>;
```

mycat4.pl
```
1:  #! /usr/bin/perl -w
2:
3:  use strict;
4:  use warnings;
5:
6:  print while <>;
```

①「mycat.pl」、②「mycat4.pl」の順番に、300MB のファイルを対象に読み込みを実行しましたが、見ての通り for 文ではメモリが大量に確保されています。

ありがたいご指摘をして頂いた uasi 氏に感謝致します。

次に添削についてですが、「perl」のコマンドラインオプションに興味深い記述があります。

```
$ perl --help
  -n                assume "while (<>) { ... }" loop around program
  -p                assume loop like -n but print line also, like sed
```

どうやら、ループを書かなくてもループを追加してくれるオプションがあるらしい……。

結果、mycat はさらに短くなりました。

ここまで来ると、何がしたいのかコードからは完全に不明[*7] ですが、何事もなかったかのように動作します。さすが Perl だ、なんともないぜ。

mycat5.pl
```
1:  #! /usr/bin/perl -wp
```

*7 完全に不明：コード量ゼロですし。

このコマンドラインオプションを用いることで、今回ソースコードに行番号を付けて出力するために用いた「add-line-number.pl」は、以下のように書くことが出来ます。

```
add-line-number.pl
1:  #! /usr/bin/perl -wp
2:
3:  print "$.:¥t";
```

Perl で書かれたコードは大変シンプルになることがお分かり頂けたかと思います。

参考文献

- 「プログラミング Perl 第 3 版 VOLUME 1,2」(オライリー・ジャパン)
- 「PERL HACKS プロが教えるテクニック＆ツール 101 選」(オライリー・ジャパン)
- http://www.perl.org/
- http://www.cpan.org/
- perldoc コマンド
- Google 先生

免責

本記事の内容は、引用部分や素材など、著作権その他の権利が筆者に帰属しない物、あるいは個別に但書きされている物を除き、Perl そのものと同じ「Artistic License[*8]」か、あるいは「2-clause BSD license[*9]」に従って再利用できます。

また、記事タイトルの「Perl::Object」というモジュールは実際に存在しているわけではありませんのでご注意下さい。

本記事の内容は、以下の URL でも公開しております。

http://lab.mitty.jp/word/

*内容に間違い・質問などありましたら、*twitter:@mittyorz *まで連絡して頂ければ幸いです。*

*8 Artistic License：http://dev.perl.org/licenses/artistic.html

*9 2-clause BSD license：http://www.freebsd.org/copyright/freebsd-license.html