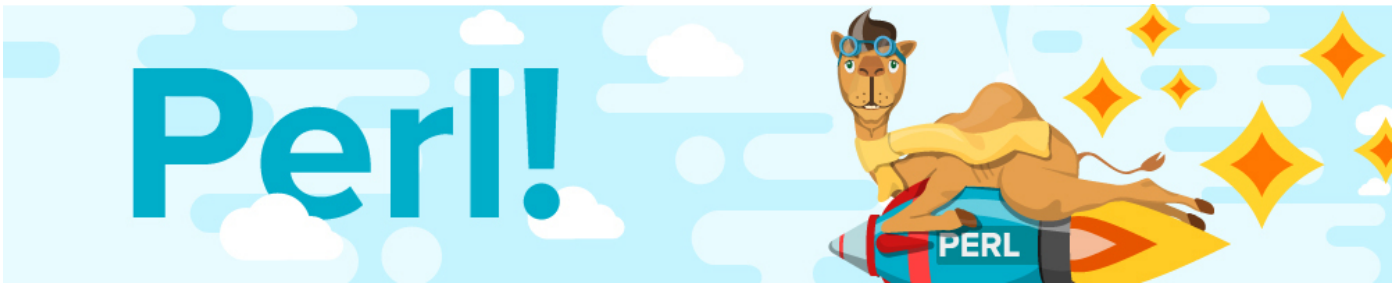
REG.RU 62,68
Каждый день мы делаем Рунет лучше! Вы с нами?

12 августа 2014 в 15:56

Функции в Perl

tutorial

Блог компании REG.RU, Программирование*, Perl*



В Perl заложено огромное количество возможностей, которые, на первый взгляд, выглядят лишними, а в неопытных руках могут вообще приводить к появлению багов. Доходит до того, что многие программисты, регулярно пишущие на Perl, даже не подозревают о полном функционале этого языка! Причина этого, как нам кажется, заключается в низком качестве и сомнительном содержании литературы для быстрого старта в области программирования на Perl. Это не касается только книг с Ламой, Альпакой и Верблюдом («Learning Perl», «Intermediate Perl» и «Programming Perl») — мы настоятельно рекомендуем их прочитать.

В этой статье мы хотим подробно рассказать о маленьких хитростях работы с Perl, касающихся необычного использования функций, которые могут пригодится всем, кто интересуется этим языком.

Как работают функции Perl?

В большинстве языков программирования описание функции выглядит примерно так:

```
function myFunction (a, b) {
        return a + b;
}
```

А вызывается функция так:

```
myFunction(1, 2);
```

На первый взгляд всё просто и понятно. Однако вызов этой функции в следующем виде:

```
myFunction(1, 2, 3);
```

… в большинстве случаев приведёт к ошибкам, связанным с тем, что в функцию передано неверное количество аргументов.

Функция в Perl может быть записана так:

```
sub my_sub($$;$) : MyAttribute {
        my ($param) = @_;
}
```

Где $$;$ — это прототип, а MyAttribute — это атрибут. Прототипы и атрибуты будут рассмотрены далее в статье. А мы пока рассмотрим более простой вариант записи функции:

```
sub my_sub {
        return 1;
}
```

Здесь мы написали функцию, которая возвращает единицу.

Но в этой записи не указано, сколько аргументов принимает функция. Именно поэтому ничего не мешает вызвать её вот так:

```
my_sub('Туземец', 'Бусы', 'Колбаса', 42);
```

И всё прекрасно выполняется! Это происходит потому, что в Perl передача параметров в функцию сделана хитро. Perl славится тем, что у него много так называемых «специальных» переменных. В каждой функции доступна специальная переменная @_, которая является массивом входящих параметров.

Поэтому внутри функции мы можем поместить входные параметры в переменные так:

```
my ($param) = @_;
```

Это работает и в случае нескольких параметров:

```
my ($param1, $param2, $param3) = @_;
```

Очень часто в функциях пишут следующее:

```
sub my_sub {
        my $param = shift;
        ...
}
```

Дело в том, что в Perl многие функции при вызове без аргументов используют переменные по умолчанию. shift же по умолчанию достаёт данные из массива @_. Поэтому записи:

```
my $param = shift;
```

… и

```
my $param = shift @_;
```

… совершенно эквивалентны, но первая запись короче и очевидна для Perl-программистов, поэтому используется именно она.

shift можно использовать и для получения нескольких параметров, в том числе комбинируя в одно списочное присваивание:

```
my ($one, $two, $three) = (shift, shift, shift);
```

Другая запись:

```
my ($one, $two, $three) = @_;
```

… работает точно так же.

А теперь внимание! Грабли, на которые рано или поздно наступает каждый Perl-программист:

```
sub my_sub {
        my $var = @_;
        print $var;
}
```

Если вызвать данную функцию как my_sub(1, 2, 3) в $var мы внезапно получим не 1, а 3. Это происходит потому, что в данном случае контекст переменной определяется как скалярный, а в Perl массив в скалярном контексте возвращает свой размер, а не первый элемент. Чтобы исправить ошибку, достаточно взять $var в скобки, чтобы контекст стал списочным. Вот так:

```
sub my_sub {
        my ($var) = @_
}
```

И теперь, как и ожидалось, при вызове my_sub(1, 2, 3) в $var будет 1.

Кроме того, в Perl параметры передаются по ссылке. Это значит, что мы можем внутри функции модифицировать параметры, которые в неё переданы.

Например:

```
my $var = 5;
my_sub($var);
print $var;

sub my_sub {
        # вспоминаем, что доступ к элементам массива выполняется в скалярном контекс
те
        # т. е. доступ к нулевому элементу массива @arr будет выглядеть как $arr[0],
 то же самое и с
        # @_.
        $_[0]++; # $_[0] — первый элемент массива @_.
}
```

Результат будет 6. Однако в Perl можно сделать в каком-то роде «передачу по значению» вот так:

```
my $var = 5;
my_sub($var);
print $var;

sub my_sub {
        my ($param) = @_;

        $param++;
}
```

А вот теперь результат будет 5.

И последние два нюанса, которые очень важны. Perl автоматически возвращает из функции результат последнего выражения.

Возьмём код из предыдущего примера и немного его модифицируем:

```
my $var = 5;
my $result = my_sub($var);
print $result;

sub my_sub {
        my ($param) = @_;

        ++$param;
}
```

Это будет работать точно так же, как если бы в последней строке функции был явный возврат значения:

```
return ++$param;
```

Функция вернёт 6.

И ещё одна особенность: если в теле функции вызывается другая функция с использованием амперсанда и без скобок, то внутренняя функция получает на вход параметры той функции, в теле которой она вызывается. Т. е. массив @_ будет автоматически передан из внешней функции во внутреннюю. Это может привести к неочевидным багам.

```
use strict;
use Data::Dumper;
```

```
my_sub(1, 2, 3);
sub my_sub {
        &inner;
}

sub inner {
        print Dumper \@_;
}
```

Результат:

```
$VAR1 = [
  1,
  2,
  3
];
```

Однако, если явно указать (с помощью пустых скобок), что функция вызывается без параметров, то всё в порядке:

```
sub my_sub {
        &inner();
}
```

И вывод будет выглядеть вот так:

```
$VAR1 = [];
```

Впрочем, вызовы функций с использованием амперсанда используются очень редко и в коде почти не встречаются.

Анонимные функции

Анонимные функции объявляются в месте использования и не получают уникального идентификатора для доступа к ним. При создании они либо вызываются напрямую, либо ссылка на функцию присваивается переменной, с помощью которой затем можно косвенно вызывать эту функцию.

Элементарное объявление анонимной функции в Perl:

```
my $subroutine = sub {
        my $msg = shift;
        printf "I am called with message: %s\n", $msg;
        return 42;
};
# $subroutine теперь ссылается на анонимную функцию
$subroutine->("Oh, my message!");
```

Анонимные функции можно и нужно использовать как для создания блоков кода, так и для замыканий, о которых речь дальше.

Замыкания

Замыкание — это особый вид функции, в теле которой используются переменные, объявленные вне тела этой функции.

В записи это выглядит как, например, функция, находящаяся целиком в теле другой функции.

```
# возвращает ссылку на анонимную функцию
sub adder($) {
        my $x = shift;     # в котором x — свободная переменная,
        return sub ($) {
        my $y = shift;     # а y — связанная переменная
        return $x + $y;
        };
}

$add1 = adder(1);   # делаем процедуру для прибавления 1
```

```
print $add1->(10);  # печатает 11

$sub1 = adder(-1);  # делаем процедуру для вычитания 1
print $sub1->(10);  # печатает 9
```

Замыкания использовать полезно, например, когда необходимо получить функцию с уже готовыми параметрами, которые будут в ней сохранены. Или же для генерации функции-парсера, колбеков.

Бесскобочные функции

На наш взгляд, это самый подходящий перевод термина parenthesis-less.

Например, print часто пишется и вызывается без скобок. Возникает вопрос, а можем ли мы тоже создавать такие функции?

Безусловно. Для этого у Perl есть даже специальная прагма — subs. Предположим, нам нужна функция, проверяющая значение переменной на истинность.

```
use strict;
use subs qw/checkflag/;
my $flag = 1;
print "OK" if checkflag;

sub checkflag {
        return $flag;
}
```

Эта программа напечатает OK.

Но это не единственный способ. Perl хорошо продуман, поэтому, если мы реструктуризируем нашу программу и приведём её к такому виду:

```
use strict;

my $flag = 1;
sub checkflag {
        return $flag;
}

print "OK" if checkflag;
```

…то результат будет тот же.

Закономерность здесь следующая — мы можем вызывать функцию без скобок в нескольких случаях:

- используя прагму subs;
- написав функцию ПЕРЕД её вызовом;
- используя прототипы функций.

Обратимся к последнему варианту.

Прототипы функций

Зачастую разное понимание цели этого механизма приводит к холиварам с адептами других языков, утверждающих, что «у перла плохие прототипы». Так вот, прототипы в Perl не для жёсткого ограничения типов параметров, передаваемых функциям. Это подсказка для языка: как разбирать то, что передаётся в функцию.

Авторы из PerlMonks объясняли это как «parameter context templates» — шаблоны контекста параметров. Детали на примерах ниже.

Есть, к примеру, абстрактная функция, которая называется my_sub:

```
sub my_sub {
        print join ', ', @_;
}
```

Мы её вызываем следующим образом:

```
my_sub(1, 2, 3, 4, 5);
```

Функция напечатает следующее:

```
1, 2, 3, 4, 5,
```

Получается, что в любую функцию Perl можно передать любое количество аргументов. И пусть сама функция разбирается, что мы от неё хотели.

Предполагается, что должен быть механизм контроля переданных в функцию аргументов. Эту роль и выполняют прототипы.

Функция Perl с прототипами будет выглядеть так:

```
sub my_sub($$;$) {
        my ($v1, $v2, $v3) = @_;
        $v3 ||= 'empty';
    printf("v1: %s, v2: %s, v3: %s\n", $v1, $v2, $v3);
}
```

Прототипы функций записываются после имени функции в круглых скобках. Прототип $$;$ означает, что в качестве параметров необходимо присутствие двух скаляров и третьего по желанию, «;» отделяет обязательные параметры от возможных.

Если же мы попробуем вызвать её вот так:

```
my_sub();
```

…то получим ошибку вида:

```
Not enough arguments for main::my_sub at pragmaticperl.pl line 7, near "()"
Execution of pragmaticperl.pl aborted due to compilation errors.
```

А если так:

```
&my_sub();
```

…то проверка прототипов не будет происходить.

Резюмируем. Прототипы будут работать в следующих случаях:

- Если функция вызывается без знака амперсанда (&). Perlcritic (средство статического анализа Perl кода), кстати говоря, ругается на запись вызова функции через амперсанд, то есть такой вариант вызова не рекомендуется.
- Если функция написана перед вызовом. Если мы сначала вызовем функцию, а потом её напишем, при включённых warnings получим следующее предупреждение:
  `main::my_sub() called too early to check prototype at pragmaticperl.pl line 4`

Ниже пример правильной программы с прототипами Perl:

```
use strict;
use warnings;
use subs qw/my_sub/;

sub my_sub($$;$) {
        my ($v1, $v2, $v3) = @_;
        $v3 ||= 'empty';
    printf("v1: %s, v2: %s, v3: %s\n", $v1, $v2, $v3);
}
my_sub();
```

В Perl существует возможность узнать, какой у функции прототип. Например:

```
perl -e 'print prototype("CORE::read")'
```

Выдаст:

```
*\$$;$
```

Оверрайд методов

Оверрайд — часто довольно полезная штука. Например, у нас есть модуль, который писал некий N. И всё в нём хорошо, а вот один метод, допустим, call_me, должен всегда возвращать 1, иначе беда, а метод из базовой поставки модуля возвращает всегда 0. Код модуля трогать нельзя.

Пусть программа выглядит следующим образом:

```
use strict;
use Data::Dumper;

my $obj = Top->new();
if ($obj->call_me()) {
        print "Purrrrfect\n";
}
else {
        print "OKAY :(\n";
}

package Top;
use strict;
sub new {
        my $class = shift;
        my $self = {};
        bless $self, $class;
        return $self;
}

sub call_me {
        print "call_me from TOP called!\n";
        return 0;
}

1;
```

Она выведет:

```
call_me from TOP called!
OKAY :(
```

И снова у нас есть решение.

Допишем перед вызовом $obj->call_me() следующую вещь:

```
*Top::call_me = sub {
        print "Overrided subroutine called!\n";
        return 1;
};
```

А ещё лучше, для временного оверрайда используем ключевое слово local:

```
local *Top::call_me = sub {
        print "Overrided subroutine called!\n";
```

```
        return 1;
};
```

Это заменит функцию call_me пакета Top в лексической области видимости (в текущем блоке).
Теперь наш вывод будет выглядеть так:

```
Overrided subroutine called!
Purrrrfect
```

Код модуля не меняли, функция теперь делает то, что нам надо.

На заметку: если приходится часто использовать данный приём в работе — налицо архитектурный косяк. Хороший пример использования — добавление вывода отладочной информации в функции.

Wantarray

В Perl есть такая полезная штука, которая позволяет определить, в каком контексте вызывается функция. Например, мы хотим, чтобы функция вела себя следующим образом: когда надо возвращала массив, а иначе — ссылку на массив. Это можно реализовать, и к тому же очень просто, с помощью wantarray. Напишем простую программу для демонстрации:

```
#!/usr/bin/env perl

use strict;
use Data::Dumper;

my @result = my_cool_sub();
print Dumper @result;

my $result = my_cool_sub();
print Dumper $result;

sub my_cool_sub {
    my @array = (1, 2, 3);

    if (wantarray) {
        print "ARRAY!\n";
        return @array;
    }
    else {
        print "REFERENCE!\n";
        return \@array;
    }
}
```

Что выведет:

```
ARRAY!
$VAR1 = 1;
$VAR2 = 2;
$VAR3 = 3;
REFERENCE!
$VAR1 = [
  1,
  2,
  3
];
```

Также хотелось бы напомнить про интересную особенность Perl. %hash = @array; В этом случае Perl построит хэш вида ($array[0] => $array[1], $array[2] => $array[3]);

Посему, если применять my %hash = my_cool_sub(), будет использована ветка логики wantarray. И именно по этой причине

wanthash нет.

AUTOLOAD

В Perl одна из лучших систем управления модулями. Мало того что программист может контролировать *все* стадии исполнения модуля, так ещё существуют интересные особенности, которые делают жизнь проще. Например, AUTOLOAD.

Суть AUTOLOAD в том, что когда вызываемой функции в модуле не существует, Perl ищет функцию AUTOLOAD в этом модуле, и только затем, если не находит, выбрасывает исключение о вызове несуществующей функции. Это значит, что мы можем описать обработчик ситуаций, когда вызывается несуществующая функция.

Например:

```
#!/usr/bin/env perl
use strict;

Autoload::Demo::hello();
Autoload::Demo::asdfgh(1, 2, 3);
Autoload::Demo::qwerty();

package Autoload::Demo;
use strict;
use warnings;

our $AUTOLOAD;

sub AUTOLOAD {
    print $AUTOLOAD, " called with params: ", join (', ', @_), "\n";

}

sub hello {
    print "Hello!\n";
}

1;
```

Очевидно, что функций qwerty и asdfgh не существует в пакете Autoload::Demo. В функции AUTOLOAD специальная глобальная переменная $AUTOLOAD устанавливается равной функции, которая не была найдена.

Вывод этой программы:

```
Hello!
Autoload::Demo::asdfgh called with params: 1, 2, 3
Autoload::Demo::qwerty called with params:
```

Генерация функций на лету

Допустим, нам нужно написать множество функций, выполняющих примерно одинаковые действия. Например, набор аксессоров у объекта. Написание подобного кода вряд ли кому-то доставит удовольствие:


```
sub getName {
    my $self = shift;
    return $self->{name};
}

sub getAge {
    my $self = shift;
    return $self->{age};
}
```

```
sub getOther {
    my $self = shift;
    return $self->{other};
}
```

Это Perl. «Лень, нетерпение, надменность» (Л. Уолл).

Функции можно генерировать. В Perl есть такая штука как тип данных typeglob. Наиболее точный перевод названия — таблица имён. Typeglob имеет свой сигил — «*».

Для начала посмотрим код:

```
#!/usr/bin/env perl
use strict;
use warnings;

package MyCoolPackage;

sub getName {
    my $self = shift;
    return $self->{name};
}

sub getAge {
    my $self = shift;
    return $self->{age};
}

sub getOther {
    my $self = shift;
    return $self->{other};
}


foreach (keys %{*MyCoolPackage::}) {
        print $_." => ".$MyCoolPackage::{$_}."\n";
}
```

Вывод:

```
getOther => *MyCoolPackage::getOther
getName => *MyCoolPackage::getName
getAge => *MyCoolPackage::getAge
```

В принципе, глоб — это хэш с именем пакета, в котором он определен. Он содержит в качестве ключей элементы модуля + глобальные переменные (our). Логично предположить, что если мы добавим в хэш свой ключ, то этот ключ станет доступен как обычная сущность. Воспользуемся генерацией функций для генерации данных геттеров.

И вот что у нас получилось:

```
#!/usr/bin/env perl
use strict;
use warnings;

$\ = "\n";
my $person = Person->new(
    name    =>  'justnoxx',
    age     =>  '25',
    other   =>  'perl programmer',
);

print "Name: ", $person->get_name();
```

```
print "Age: ", $person->get_age();
print "Other: ", $person->get_other();

package Person;
use strict;
use warnings;

sub new {
    my ($class, %params) = @_;

    my $self = {};

    no strict 'refs';
    for my $key (keys %params) {
        # __PACKAGE__ равен текущему модулю, это встроенная
        # волшебная строка
        # следующая строка превращается в, например:
        # Person::get_name = sub {...};
        *{__PACKAGE__ . '::' . "get_$key"} = sub {
            my $self = shift;
            return $self->{$key};
        };
        $self->{$key} = $params{$key};
    }

    bless $self, $class;
    return $self;
}

1;
```

Эта программа напечатает:

```
Name: justnoxx
Age: 25
Other: perl programmer
```

Атрибуты функций

В Python есть такое понятие как декоратор. Это такая штуковина, которая позволяет «добавить объекту дополнительное поведение».

Да, в Perl декораторов нет, зато есть атрибуты функций. Если мы откроем perldoc perlsub и посмотрим на описание функции, то увидим любопытную запись:

```
sub NAME(PROTO) : ATTRS BLOCK
```

Таким образом, функция с атрибутами может выглядеть так:

```
sub my_sub($$;$) : MyAttr {
        print "Hello, I am sub with attributes and prototypes!";
}
```

Работа с атрибутами в Perl — дело нетривиальное, потому уже довольно давно в стандартную поставку Perl входит модуль Attribute::Handlers.

Дело в том, что атрибуты из коробки имеют довольно много ограничений и нюансов работы, так что, если кому-то интересно, можно обсудить в комментариях.

Допустим, у нас есть функция, которая может быть вызвана только в том случае, если пользователь авторизован. За то, что

пользователь авторизован отвечает переменная $auth, которая равна 1, если пользователь авторизован, и 0, если нет. Мы можем сделать следующим образом:

```
my $auth = 1;

sub my_sub {
        if ($auth) {
        print "Okay!\n";
        return 1;
        }
        print "YOU SHALL NOT PASS!!!1111";
        return 0;
}
```

И это приемлемое решение.

Но может возникнуть такая ситуация, что функций будет становиться больше и больше. А в каждой делать проверку будет всё накладнее. Проблему можно решить с помощью атрибутов.

```
use strict;
use warnings;
use Attribute::Handlers;
use Data::Dumper;

my_sub();

sub new {
        return bless {}, shift;
}

sub isAuth : ATTR(CODE) {
        my ($package, $symbol, $referent, $attr, $data, $phase, $filename, $linenum) = @_;
        no warnings 'redefine';
        unless (is_auth()) {
        *{$symbol} = sub {
            require Carp;
            Carp::croak "YOU SHALL NOT PASS\n";
        };
        }
}

sub my_sub : isAuth {
        print "I am called only for auth users!\n";
}

sub is_auth {
        return 0;
}
```

В данном примере вывод программы будет выглядеть так:

```
YOU SHALL NOT PASS at myattr.pl line 18. main::__ANON__() called at myattr.pl line 6
```

А если мы заменим return 0 на return 1 в is_auth, то:

```
I am called only for auth users!
```

Не зря атрибуты представлены в конце статьи. Для того чтобы написать этот пример, мы воспользовались:

- анонимными функциями
- оверрайдом функций
- специальной формой оператора goto

Несмотря на довольно громоздкий синтаксис, атрибуты успешно и активно применяются, например, в веб-фреймворке Catalyst.

Однако, не стоит забывать, что атрибуты, всё-таки, являются экспериментальной фичей Perl, а потому их синтаксис может меняться в следующих версиях языка.

Статья написана в соавторстве и по техническому материалу от Дмитрия Шаматрина (@justnoxx) и при содействии программистов REG.RU: Тимура Нозадзе (@TimurN), Виктора Ефимова (@vsespb), Полины Шубиной (@imagostorm), Andrew Nugged (@nugged)

Perl, REG.RU, рег.ру, программирование

+94 36,3k 309

Автор: @Alessandra

рейтинг
REG.RU 62,68
Компания прекратила активность на сайте

ПОХОЖИЕ ПУБЛИКАЦИИ

29 сентября 2015 в 14:50
OTRS на прокачку в стиле REG.RU
+9 16,1k 51 28

20 марта 2014 в 17:35
Хостинг-провайдер vs. DDoS-атака. Решение REG.RU – профессиональный аутсорс
+1 20,4k 19 23

10 сентября 2013 в 15:34
Perl Golf от REG.RU на YAPC::Europe 2013
+21 3,8k 12 22

Комментарии (33)

dmn42 12 августа 2014 в 16:35 +17

Оу. Пост про перл на хабре. Спасибо, почитал с удовольствием.

Про замыкания и аттрибуты очень полезно, пригодится.

nfubh 12 августа 2014 в 16:42 +3

Мне Perl и, в частности, книжка Higher-Order Perl вернули интерес к программированию, утраченный за время использования C++.

youlose 12 августа 2014 в 23:06 +1

Поддерживаю, отличная книга про функциональное программирование.

JIghtuse 12 августа 2014 в 16:45 *(комментарий был изменён)* +3

Спасибо, познавательная статья. Всё-таки неспроста Perl широко распространён.

> Это не касается только книг с Ламой, Альпакой и Верблюдом (“Learning Perl”, “Intermediate Perl” и “Programming Perl”) — мы настоятельно рекомендуем их прочитать.

Что скажете о Modern Perl?

TimurN 12 августа 2014 в 23:30 +3

«Modern Perl» появилась в тот момент, когда все классические книги по Perl давно не переиздавались. А при этом как раз возникал это самый обновленческий Modern Perl, появлялись новые техники и практики, которых в старых книгах не было, менялись взгляды на то, что раньше считалось нормой в написании кода. Так что на тот момент это была, можно сказать, единственная актуальная книга по Perl.

С появлением свежих версий классики актуальность её существенно уменьшилась, да и уровень изложения там не так высок. Но, тем не менее, внимания заслуживает, особенно для тех, кто не готов с ходу осилить что-то более объёмное. ;)

Я её переводил, кстати, если кому интересно на русском.
Исходники на Гитхабе: github.com/timurn/modern_perl_book/tree/russian_translation.
Сайт: modernperlbook.ru (до конца так и не докрутили, но читать можно).

evnp 20 августа 2014 в 10:24 0

А готового pdf/fb2/html для чтения в оффлайне вы не делали?

TimurN 20 августа 2014 в 11:43 0

Нет, так пока руки и не дошли допилить. Принимаем патчи. ;)

Nyaon 12 августа 2014 в 16:50 +4

Круто, хорошо. Можно ещё добавить про __SUB__ и сигнатуры из 5.20.

Boomburum 12 августа 2014 в 17:31 +8

Эталонно оформленная статья, добавил в избранное для семинаров :)

UncleAndy 12 августа 2014 в 17:51 +3

Работаю с perl более 10 лет. Статью почитал с большим интересом. Спасибо!

kloppspb 12 августа 2014 в 18:02 +2

в Perl параметры передаются по ссылке. Это значит, что мы можем из функции модифицировать параметры, которые в неё переданы

Вот, кстати, один из «жизненных» примеров: perltrap.com/ru/totalnaya-numerifikacia/

Да и вообще, на PerlTrap много чего интересного, и про функции, и про не-функции :)

SLY_G 12 августа 2014 в 18:07 0

Реквестирую такую же подробную статью про plack

@/ liruoko 18 августа 2014 в 20:41 0

Кстати, про Plack есть цикл статей в PragmaticPerl:
статья 1,
статья 2,
статья 3,
статья 4,
статья про разворачивание Plack-приложения.

Есть совсем краткое введение-инструкция здесь:
про Plack вообще,
про Plack + SOAP.

TimurN 20 августа 2014 в 11:54 0

Кстати, цикл вводных статей по Plack в PragmaticPerl тоже написан основным автором этой статьи (justnoxx). Не думаю, что есть смысл дублировать это на Хабре. Если какие-то более конкретные вопросы интересуют — пишите, может, что и придумаем.

bes_internal 12 августа 2014 в 19:11 +6

Первый пример не очень получился. Лучше так:

```
sub mySub {
      my $var = @_;
      print $var;
  }
```

Если вызвать данную функцию как mySub('a', 'b', 'c') в $var мы внезапно получим не 'a', а 3.

Вообще написание хороших примеров — это особое исскусство.

Статья получилась отличная. Только недавно просил на ru.pm чтобы кто-нибудь написал сводную статью про функции. Не хватает только сигнатур из 5.20

Вот хороший однострочник для затравки:

```
perl -E 'say "kaboom" if bomb; $because=bareword; say $because'
```

jrip 12 августа 2014 в 20:09 *(комментарий был изменён)* +8

Всегда любил перл.
Но когда в коде используешь все его специфические «фишки», не покидает очень своеобразное ощущение, что рассказываешь очень специфический анекдот, которого не поймет никто.

portsigar 12 августа 2014 в 23:28 0

Отличная статья. Освежил память и узнал про атрибуты:)
Хочу поинтересоваться, а как Вы относитесь к использованию модулей? Например, мой типичный набор для функций и методов включает в себя Params::Validate и аналогичные (например MooseX::Params::Validate), что позволяет более точно контроллировать данные на входе.

OlegTar 13 августа 2014 в 01:06 +5

Перл Хабр!

ksurent 13 августа 2014 в 03:06 0

Последний пример сломан же, не? Скорее всего имелось в виду что–то типа:

```
*{$symbol} = sub {
    if(is_auth()) {
        goto &$referent;
    }
    else {
        require Carp;
        Carp::croak "nope";
    }
};
```

justnoxx 13 августа 2014 в 10:50 *(комментарий был изменён)* 0

Спасибо, goto там было лишним, исправлено, однако пример не сломан. Дело в том, что если проверка атрибутов проходит успешно далее вызывается эта самая функция и goto не нужен. В данном случае проверка атрибутов выполняется успешно, если is_auth возвращает 1.
Я писал большую статью в pragmaticperl по атрибутам и более подробно рассматривал подробные примеры.

ignat99 13 августа 2014 в 04:47 *(комментарий был изменён)* –1

В Софии в Болгарии 22 августа откроется конференция по Perl: YAPC::Europe 2014
Booking (крупный Perl сервис), видимо будет нанимать Perl специалистов в свой офис в Амстердаме.
Было бы приятно увидеть там коллег по Хабру и Perl.

Alessandra 13 августа 2014 в 10:39 *(комментарий был изменён)* 0

Двое из имеющих отношение к этой статье авторов (@TimurN, @nugged) в Софии будут.

ignat99 15 августа 2014 в 01:42 –1

Делаю маленький доклад. Вот в расписании act yapc eu / ye2014 / schedule? day = 2014-08-23
23 августа в 10 часов утра. «Using Perl for autogeneration physical formulas».

Постараюсь сделать сообщение по интересному с конференции по Perl: YAPC::Europe 2014.

nfubh 13 августа 2014 в 13:02 0

Они уже ищут желающих пожить в Амстердаме и не только. Дерзайте :)

akzhan 13 августа 2014 в 04:56 *(комментарий был изменён)* 0

Многие давно используют нормальные функции и методы (например, MooseX::Method::Signatures):

```
method hello (Str :$who, Int :$age where { $_ > 0 }) {
    $self->say("Hello ${who}, I am ${age} years old!");
}
```

Ну и напомню — habrahabr.ru/post/52532/

unel 13 августа 2014 в 12:08 0

Приятно удивлён, что статья про perl на хабре встречена позитивно)

Вопрос авторам статьи: каково ваше мнение про perl6? Взлетит или нет? Как там вообще прогресс, есть?

TimurN 13 августа 2014 в 15:14 +1

На этот вопрос вряд ли могут достоверно ответить даже авторы Perl 6, что уж там говорить про авторов статьи.

Пока Perl 6 — это академический проект. Взлетит — хорошо, не взлетит — для пользователей Perl 5 всё равно вряд ли что-то кардинально от этого изменится. Строить же свои проекты или свою карьеру с расчётом на Perl 6 пока однозначно не стоит.

Pilat 13 августа 2014 в 13:47 0

Интересная статья.
Мне кажется, что ещё более интересной и полезной была бы статья о стиле программирования на перле, который не приводит к проблемам поддержки и отладки программ. Что я имею ввиду:
Например, генерация методов на лету, — приводит к тому, что функция то есть, то её нет в зависимости от переданных параметров. Когда-то это полезно, конечно, но если этим увлекаться, то поиск места объявления такой функции превращается в квест. AUTOLOAD сам по себе приносит больше проблем чем пользы, мне кажется. Да, это дополнительные возможности, но и дополнительные проблемы, иногда через несколько лет после написания программы. Хотя многие синтаксические сахары именно на определении на лету и AUTOLOAD построены. Но отлаживать их просто ад.

Приведённый пример с isAuth выглядит привлекательным, но всё хорошо пока есть привычка именно к такому стилю программирования. Не говоря о экспериментальном характере атрибутов. Тот же каталист по неизвестным причинам проигрывает неатрибутному Rose в скорости на порядок. И вообще если для использования какой-то части языка нужен специальный модуль — это уже повод задуматься.
MooseX::Method::Signatures, MooseX::Declare выглядят хорошо, но с ними не дружат некоторые перловые IDE. Не считая что это ещё и Moose.

Прототипы теоретически хорошо, но практически мало где используется. Проблем от них больше чем пользы, Params::Validate удобнее. Этот момент давно надо было бы улучшить в перле.

Насчёт бесскобочных функций как-то невнятно сказано, что все функции можно вызывать без скобок. Главное, чтобы это была именно функция, что достигается либо её объявлением в use subs, либо обычным определением. При чём тут третий путь — прототипы — непонятно.

justnoxx 13 августа 2014 в 14:03 0

Насчет прототипов. Например, константы в Perl это функции с пустым прототипом ().

vsespb 13 августа 2014 в 14:10 −1

> Например, генерация методов на лету, — приводит к тому, что функция то есть, то её нет в зависимости от переданных параметров. Когда-то это полезно, конечно, но если этим увлекаться, то поиск места объявления такой функции превращается в квест.

Вот у нас есть Redis, у него несколько БД, у каждой БД свой индекс (для команды select), свои таймауты (в зависимости от паттерна использования), список БД redis хранится в конфиге.
Сделали автогенерации функций на стадии компиляции модуля: `r_somedb`, `r_anotherdb` — возвращает нужный коннект. Т.е. `r_somedb->somecommand` выполняет нужную команду. Всё документировали. Написали тесты.
Без автогенерации пришлось бы делать что-то вроде `redis("somedb")->somecommand`, при этом в `"somedb"` не должно быть опечатки. Или сделать константу `SOMEDB => "somedb"`, что опять, является дубликатом конфига, и использовать `redis(SOMEDB)->somecommand`. В итоге

это было бы неудобно, т.к. такое обращение используется в сотнях строчек кода.

> AUTOLOAD сам по себе приносит больше проблем чем пользы, мне кажется.

Опять про Redis: Есть CPAN модули Redis и Redis::Fast. В обоих модуль не знает полный список команд сервера Redis. И выполняет любую команду вызванную как функцию. Т.е.
`$redis->mget $redis->get` — всё это обрабатывается AUTOLOAD.

> И вообще если для использования какой-то части языка нужен специальный модуль — это уже повод задуматься.

Например, для использования наследования нужен специальный модуль. `use parent` или `use base`. Напрямую `@ISA` не принято писать. Ещё для экспортирования функций используют `use Exporeter`. Эти модули как и `Attribute::Handlers` — модули ядра.

 **TimurN** 13 августа 2014 в 15:09 0

Написать проблемный, потенциально бажный, неотлаживаемый, неподдерживаемый и т. п. код можно и с использованием самых простых конструкций языка. Причём, наверное, любого языка.

Так что нужно просто включать мозг в каждом конкретном случае. И иметь какие-то правила и отстроенное взаимодействие внутри проекта. А какой-то такой универсальный стиль программирования, позволяющий писать любые программы так, чтобы всё было шоколадно, вряд ли существует.

 **mialinx** 14 августа 2014 в 00:35 *(комментарий был изменён)* +4

Еще по поводу прототипов:
Обычно все говорят **как** и никто не говорит **зачем**.

В Perl есть хорошо известное соглашение про передачу параметров в пользовательские функций.
Но некоторые **встроенные** функции работают немного не по этому соглашению.

```
push @arr, $elem; # @arr не сливается в один список с $elem.
```

Так вот: прототипы в Perl нужны с единственной целью — позволить программистам делать так же.
Как правило это удобно для очень низкоуровневых функций для работы с данными и блоками.
В прикладной же разработке — почти нафиг не нужно.

Пожалуйста, подумайте дважды прежде чем писать прототип: вы точно делаете новый map?
Пока видел одно достойное применение прототипов — Try::Tiny и аналоги.

 **Foxcool** 14 августа 2014 в 16:31 *(комментарий был изменён)* 0

Пользуясь случаем предлагаю перловикам присоединиться к поддержке любимого языка. secure.donor.com/pf012/give

donate.perlfoundation.org/donate.html

Только зарегистрированные пользователи могут оставлять комментарии. Войдите, пожалуйста.

Зачем в команде мобильной разработки появился сейф под управлением Windows 10

+18 4,5k 17 21

Как развивать продукт, если в команде один разработчик и два заказчика?

+6 3,4k 16 4

Делаем более-менее универсальный калькулятор услуг для сайта

+13 704 20 6

Предсказываем будущее с помощью библиотеки Facebook Prophet

+20 1,6k 28 3

ИНТЕРЕСНЫЕ ПУБЛИКАЦИИ

Делаем более-менее универсальный калькулятор услуг для сайта

658 19 6

Предсказываем будущее с помощью библиотеки Facebook Prophet

1,5k 27 3

Обзор JBL Trip: автомобильная гарнитура не для уха GT

1,6k 3 14

Замена Oracle на PostgreSQL и возможности работы с секционированием внутри DLP-системы

703 4 0

Эксперты ставят под сомнение работу новой твердотельной батареи Гуденафа GT

7k 11 22