PRAGMATIC PERL

25

03/2015

pragmaticperl.com

Pragmatic Perl 25

pragmaticperl.com

Выпуск 25. Март 2015

Другие выпуски и форматы журнала всегда можно загрузить с pragmaticperl.com. С вопросами и предложениями пишите на почту editor@pragmaticperl.com.

Комментарии к каждой статье есть в html-версии. Подписаться на новые выпуски можно по ссылке pragmaticperl.com/subscribe.

Авторы статей: Андрей Шитов, Владимир Леттиев, Александр Ружников

Обложка: Марко Иваныk

Корректор: Андрей Шитов

Выпускающий редактор: Вячеслав Тихановский

Ревизия: 2015-03-02 09:40

Оглавление

1 От редактора. Два года журналу

Друзья! Нашему журналу исполняется два года с первого выпуска. Поздравляем вас и нас с этим событием. Подведем итоги двух лет.

За прошлый год мы выпустили без перерывов и почти вовремя 12 номеров, взяли интервью у известных в Perl-сообществе программистов, авторов книг, организаторов конференций. В создании журнала поучаствовали 29 авторов. Совместными усилиями авторов, корректоров и редакторов подготовили 1479 страниц текста!

Мы провели опрос, по результатам которого были составлены рекомендации для авторов. Читатели в целом оценили журнал хорошо. Подписалось на рассылку по email

— 1141 человек, по rss — 194. Присоединяйтесь!

У сайта журнала было около 105 000 посещений, из которых 45 000 были уникальными. Журнал был скачан 60 000 раз.

География (top 10):

- Россия 64 171
- Украина 21 273
- Беларусь 2 965
- США 2 081
- Нидерланды 1 408
- Казахстан 1 286
- Германия 1 218
- Велокобритания 1 021
- Израиль 907
- Литва 493

Друзья, журнал ищет новых авторов. Не упускайте такой возможности! Если у вас есть идеи или желание помочь, пожалуйста, свяжитесь с нами.

Приятного чтения.

■ *Вячеслав Тихановский*

2 Подключение в Mojolicious модели для бизнес-логики

Рассмотрен вариант автоматического подключения классов моделей из указанной директории

Mojolicious — фреймворк для написания веб-приложений, имеющий в своём арсенале много полезного функционала как поначалу кажется, на все случаи жизни. Но когда начинаешь с ним плотно работать и пытаться написать большое приложение, можно наткнуться на ситуацию, когда нужные решения отсутствуют в базовом наборе. Лично для меня недостатком моджо явилось отсутствие в нём моделей. Притом, что странно, данный функционал не реализован ни в самом моджо, ни в плагинах к нему, поиск по CPAN и в Гугле

не выявил соответствующих плагинов.

За время работы с фреймворком Catalyst я привык к MVC, и не хотелось отказываться от данного принципа разработки в моджо.

Вкратце напомню, в чём суть MVC. Приложение явно разделяется на три части: контроллер, модель и представление данных. В контроллер поступают запросы от пользователя, и в них же ведётся предварительная обработка данных (выведение переданных параметров, их валидация, проверка авторизации и пр.). Бизнес-логика и работа с базой данных вынесена в модель. Например, получение суммы балланса пользователя из базы данных, услуги, подключенные у пользователя, и т.п. Полученные данные возвращаются обратно в контроллер и далее отображаются пользователю через представление

(view). Как правило, для представления используются шаблонизаторы наподобие Template::Toolkit. Контроллер в данной схеме выступает тонкой прослойкой между пользовательским запросом, моделью и представлением. В самом контроллере, как правило, бизнес-логику не размещают, иначе получаются «толстые уродливые контроллеры», в которых функционал размазан и плохо поддаётся тестированию.

Итак, чего хочется? Хочется моделей как в Catalyst, чтобы из контроллера можно было делать вызовы вида:

```
1 sub my_controller {
2     my $self = shift;
3
4     # ...
5     $self->model('ModelName')->
          model_method;
6
```

```
7       # ...
8 }
```

и чтобы фреймворк сам подгружал все модели из соответствующей директории.

В моджо программисту самому предлагается дописывать необходимый функционал (например, вынести в модули отдельной директорией), либо писать обработку данных и бизнес-логику прямо в контроллерах. Подобный подход применяется в веб-фреймворках на других языках программирования, например, laravel на php, flask на python и revel на Go.

Мне же, напомню, хотелось явного определения моделей в коде. После вдумчивого гугления, просмотра кода других разработчиков и длительного курения мануалов я выработал для себя реализацию моделей

в моджо, которой хочу поделиться с вами ниже.

Реализация

Итак, для создания модели делаются следующие шаги (покажу на примере нового приложения):

- Генерим mojo-приложение: `mojo generate app MyApp && cd my_app/lib`.
- Создаём необходимые файлы и директории: `touch MyApp/Model.pm && mkdir MyApp/Model && touch MyApp/Model/Base.pm`.
- Правим `MyApp.pm`:

```
1  package MyApp;
2
3  use Mojo::Base 'Mojolicious';
4  use MyApp::Model;    # <--
       подключаем модуль с моделью
5
6  sub startup {
7      my $self = shift;
8
9      #
           ##############################;

10     # подключаем модель
11     my $model = MyApp::Model->new
           (app => $self);
12
13     # создадим соответствующий
           хелпер для вызова модели
14     # из контроллеров
15     $self->helper(
16         model => sub {
17             my ($self,
                   $model_name) = @_;
```

```
18                 return $model->
                       get_model(
                       $model_name);
19             }
20     );
21     #
           ##############################;


22
23     my $r = $self->routes;
24
25     $r->get('/')->to('root#index'
           );
26 }
27
28 1;
```

Правим MyApp/Model.pm:

```
1 package MyApp::Model;
2
3 use Mojo::Loader;
4 use Mojo::Base -base;
```

```
5
6  use Carp qw/ croak /;
7
8  has modules => sub { {} };
9
10 sub new {
11     my ($class, %args) = @_;
12     my $self = $class->SUPER::new
           (%args);
13
14     my $model_packages = Mojo::
           Loader->search('MyApp::
           Model');
15     for my $pm (grep { $_ ne '
           MyApp::Model::Base' } @{
           $model_packages}) {
16         my $e = Mojo::Loader->
               load($pm);
17         croak "Loading '$pm'
               failed: $e" if ref $e;
18         my ($basename) = $pm =~ /
               MyApp::Model::(.*)/;
```

```
19          $self->modules->{
                $basename} = $pm->new
                (%args);
20      }
21  }
22
23  sub get_model {
24      my ($self, $model) = @_;
25      return $self->modules->{
            $model} || croak "Unknown
            model '$model'";
26  }
27
28  1;
```

Правим MyApp/Model/Base.pm:

```
1  package MyApp::Model::Base;
2
3  use Mojo::Base -base;
4
5  has 'app';
6
```

```
7 1;
```

Теперь подробнее о том, что было сделано.

В файле `MyApp.pm` мы создали объект класса `MyApp::Model`, которому в конструктор передали текущий объект. Передача в конструктор текущего объекта необходима для того, чтобы потом из модели можно было обращаться ко всем методам текущего класса.

В `MyApp/Model.pm` в конструкторе мы находим все файлы в директории `MyApp/Model/` используя класс `Mojo::Loader` и метод `search`. Тут необходимо уточнить, что `Mojo::Loader` не умеет рекурсивно обходить каталог, т.е. если директория `MyApp/Model` имеет вид:

```
1  MyApp/Model
2      |_ModelFile.pm
```

```
    |_ModelFile2.pm
    |_MoreModel
        |_MoreModel1.pm
        |_MoreModel2.pm
```

то `Mojo::Loader->search('MyApp::Model')` проигнорирует директорию `MoreModel` и загрузит модули только из корневой директории. Как вариант, можно дополнительно в цикле перебрать все поддиректории в корневой директории и скормить их также `Mojo::Loader` для загрузки модулей.

Далее по коду. В конструкторе модели мы подгружаем каждый найденный модуль с моделью (за исключением `MyApp::Model::Base`, о нём дальше) и загружаем его в хеш `modules`. Метод `get_model` возвращает класс, соответствующий запрашиваемому, либо падает с ошибкой.

В файле MyApp/Model/Base.pm мы указываем все методы, которые будут наследоваться остальными моделями. Исходя из названия становится понятно, что это родительский класс для остальных модулей с моделями. Строка

```
has 'app';
```

говорит о том, что необходимо представить элемент объекта

```
$self->{app}
```

как метод класса. Т.е. вызов $self->app->config равносилен $self->{app}->config. Это синтаксический сахар от создателя моджо.

Что такое app можно понять, взглянув на код модуля MyApp.pm. Это объект самого верхнего класса MyApp.pm, который мы

передали в конструктор.

Теперь приведём пример простейшего модуля с моделью, например, `MyApp/Model/MyModel.pm`:

```
package MyApp::Model::MyModel;

use Mojo::Base 'MyApp::Model::Base';

# получить все данные из конфига
sub get_config_data {
    my $self = shift;

    return $self->app->config;
}

1;
```

Вызвать данный метод из контроллера легче лёгкого:

```
my $config = $self->model('
    MyModel')->get_config_data;
```

В данном случае будет вызван хелпер `model`, определёный в методе `MyApp::startup`. Данный хелпер вернёт вызов метода `get_model` объекта класса `MyApp::Model`.

Ну вот и всё. Как видите, ничего особо сложного нет. Я надеюсь, что через некторое время у меня дойдут руки оформить это всё в виде плагина для моджо, так как с ним в последнее время я работаю довольно активно, и писать один и тот же код постоянно утомляет. По крайней мере, проект на гитхабе под это дело уже создал :)

У кого есть вопросы, задавайте их в комментариях.

Дополнительные материалы:

- Mojo documentation
- Что такое helper в моджо
- Mojo example apps
- Пара хороших статей о моджо на хабре

■ *Александр Ружников*

3 Мутационное тестирование

Еще один способ сделать Perl-код качественнее — мутировать тесты для нахождения непротестированного кода

Мутационное тестирование — способ проверки качества тестовых наборов путем внесения случайных изменений в тестируемый код. Если после таких изменений тесты все еще проходят, значит они недостаточно качественны, т.е. не полны.

Чтобы понять, зачем это нужно и как это проверять, рассмотрим следующий пример:

```
1 package Temp;
2
3 use strict;
4 use warnings;
```

```
5
6  sub new {
7      my $class = shift;
8      my (%params) = @_;
9
10     my $self = {};
11     bless $self, $class;
12
13     $self->{critical} = $params{
           critical};
14
15     return $self;
16 }
17
18 sub is_critical {
19     my $self = shift;
20     my ($temp) = @_;
21
22     return $temp >= $self->{
           critical};
23 }
24
25 1;
```

Данный класс возвращает флаг, сигнализирующий о критической температуре. Тест для него был написан не очень внимательным программистом и выглядит следующим образом:

```
use strict;
use warnings;

use Test::More;
use Temp;

my $temp = Temp->new(critical =>
    32);

ok $temp->is_critical(32);

done_testing;
```

Тест явно недостаточен. Проверяется только лишь равенство критической температуре. Как же выявить, что тест неполный? По-

пробуем посмотреть на покрытие кода тестами, используя Devel::Cover:

```
$ PERL5OPT=—MDevel::Cover prove t
    && cover
lib/Temp.pm
                                  100.0
      n/a    n/a  100.0    0.0
   100.0
```

Покрытие 100%. Таким способом мы ничего не выявили. Неужели нельзя каким-то образом автоматически выявлять подобные проблемы, без участия человека? Можно! Здесь нам и пригодится мутационное тестирование.

Задача мутационного тестирования на основе исходного класса, скрипта, да вообще любого кода, — сгенерировать несколько вариантов, где случайным образом изменить операторы, ключевые слова, удалить

или добавить переменные и так далее. Затем мутированный код запускается через исходный набор тестов, и если тесты проходят, значит они недостаточно качественны и полны.

В описанном примере при мутации оператор >= будет заменен, например, на <=, и тесты все также будут проходить, таким образом выявляя свои недостатки.

На сегодняшний день для многих языков программирования существуют подсобные утилиты для проведения мутационного тестирования. Здесь же рассмотрим, как это можно сделать в Perl.

Как отпарсить Perl

Как можно проводить мутации? Можно, конечно, использовать регулярные выражения, однако на практике разбирать Perl-код с помощью регулярных выражений невозможно без высокой доли ложных срабатываний, ошибок и прочего. Гораздо более надежный способ — это использовать PPI.

Несмотря на известное выражение “Only perl can parse Perl” (“Только perl-интерпретатор может отпарсить Perl-язык”), PPI работает довольно сносно для большинства примеров кода. Вот как, например, заменить в описанном выше классе оператор >= на <=:

```
my $ppi = PPI::Document->new('
    Temp.pm');

if (my $operators = $ppi->find('
```

```
    PPI::Token::Operator')) {
4     foreach my $operator (
          @$operators) {
5         if ($operator->content eq
              '>=') {
6             $operator->
                  set_content('<=');
7
8             $ppi->save('Temp.pm.
                  mutant');
9         }
10    }
11 }
```

Т.е. находим все операторы (токен `PPI::Token::Operator`), затем нужный заменяем на `<=` и сохраняем мутанта под новым именем. Можно проводить и несколько мутаций, таким образом выявляя практически непроявляющиеся ошибки.

Для мутационного тестирования на CPAN есть модуль `Devel::Mutator`. Для генерации мутантов необходимо указать, какие файлы мутировать:

```
$ mutator mutate -r lib/*
```

По умолчанию все мутанты складываются в директорию `mutants`. Ключ `-r` для рекурсивного нахождения модулей.

Для запуска же самих тестов выполняется команда `test`:

```
$ mutator test
```

Это команда для каждого мутанта из директории `mutants` запускает тесты из текущей директории. По умолчанию, запускается `prove -l t`, но это можно переопределить с помощью опции `--command`. Успешность или неуспешность

тестов проверяется по значению $?. Т.е. если вы используете нестандартные тестовые модули, достаточно делать `exit(0)` в случае успеха или `exit(255)` или любое другое значение при неудаче.

Во время выполнения тестов на экран выводится текущий мутант и статус выполнения тестирования, где `ok` означает, что тесты не проходят, а `not ok` — наоборот. Например:

```
$ mutator test
(1/10) ./mutants/49606d6...009
   cd7a2/MyClass.pm ... ok
(2/10) ./mutants/3f9577f...3
   fd5f5b9/MyClass.pm ... not ok
(3/10) ./mutants/a7c40ad...
   f05df2e4/MyClass.pm ... not ok
(4/10) ./mutants/514abf0...8401
   c2f3/MyClass.pm ... not ok
...
```

Если мутант был запущен неуспешно, можно просто посмотреть, что было изменено и как это повлияло на тесты:

```
diff lib/MyClass.pm mutants/49606
    d6bcaaf550b6cf76abf009cd7a2/
    MyClass.pm
```

Из `diff` можно понять правильно ли была выполнена мутация, стоит ли обращать на это внимание. `diff` может показываться и автоматически, если указать опции `-v`, так можно сразу видеть, где проблема:

```
$ mutator test -v
(1/10) ./mutants/3
    f9577f3d44ac20732b6340d3fd5f5b9
    /MyClass.pm ... not ok
628c628
<     my @related = @_ == 1 ? ref
     $_[0] eq 'ARRAY' ? @{$_[0]} :
     ($_[0]) : ({@_});
---
```

```
6 >       my @related = @_ != 1 ? ref
        $_[0] eq 'ARRAY' ? @{$_[0]} :
        ($_[0]) : ({@_});
```

После выполнения всех тестов выводится общий результат. Например:

```
1 Result: FAIL (31/38)
```

или

```
1 Result: PASS
```

Таким образом с помощью модуля `Devel::Mutator` можно протестировать сами тесты и понять, как их можно улучшить, тем самым повысив качество программы в целом.

Недостатки

Существенным недостатком мутационного тестирования является возможность после мутации получить эквивалентную программу. В этом случае тесты будут проходить, и произойдет ложное срабатывание. Нахождение эквивалентных программ чрезвычайно сложно и на данный момент нерешаемо. Есть несколько стратегий борьбы с ложными срабатываниями, написаны несколько десятков технических статей, но универсальное решение так и не предложено.

Еще одним недостатком является вероятность создания бесконечных циклов. Для борьбы с этим в `Devel::Mutator` тесты запускаются с timeout, который по умолчанию равен 10 секундам.

■ *Вячеслав Тихановский*

4 Про переменные и сигнатуры в Perl 6

В этой статье описаны интересные синтаксические возможности Perl 6, о которых не было упомянуто в прошлый раз

Обновитесь

21 февраля появился релиз Rakudo Star 2015.02. Одновременно объявлено, что это последний релиз с поддержкой Parrot. Дальнейшая разработка будет вестись исключительно под виртуальную машину MoarVM.

Лексические переменные

Лексические переменные в Perl 6 ведут себя так, как и должны :-) Если попытаться использовать пременную вне блока, в котором она определена, возникнет ошибка. В Perl 5 ошибка возникнет лишь при наличии `use strict` или при указании версии, в которой `strict` заработает сам, например: `use v5.12`. В Perl 6 все работает сразу, и при попытке обратиться к переменной вне области ее видимости появится ошибка `Variable '$x' is not declared`.

```
{
    my $x = 42;
    say $x; # OK
}

#say $x; # Не OK
```

Лексические переменные, тем не менее, возможно удачно использовать в замыканиях. В следующем примере функция `seq()` возвращает блок, в котором используется переменная, определенная внутри функции:

```
sub seq($init) {
    my $c = $init;

    return {$c++};
}
```

После вызова функции эта переменная не только не исчезнет, но и сохранит свое значение, что хорошо видно при последующих вызовах:

```
my $a = seq(1);

say $a(); # 1
say $a(); # 2
say $a(); # 3
```

Возможно создать две независимые копии локальной переменной:

```
my $a = seq(1);
my $b = seq(42);

say $a(); # 1
say $a(); # 2
say $b(); # 42
say $a(); # 3
say $b(); # 43
```

state-переменные

Отдельно нужно упомянуть `state`-переменные. Они появились в Perl 5.10 и работают так же, как и в Perl 6. Переменная, объявленная с ключевым словом `state` внутри функции, инициализируется при первом вызове и сохраняет значение при

повторных обращениях к функции.

При этом нужно понимать, что создается действительно один-единственный экземпляр переменной. Если вернуться к примеру со счетчиком и использовать там `state` вместо `my`, то возвращаемое замыкание будет обращаться к одной и той же переменной.

```
sub seq($init) {
    state $c = $init;

    return {$c++};
}
```

Сколько бы не создавалось счетчиков:

```
my $a = seq(1);
my $b = seq(42);
```

Все они будут являться одним и тем же:

```
say $a(); # 1
say $a(); # 2
say $b(); # 3
say $a(); # 4
say $b(); # 5
```

Динамические переменные

Область видимости динамических переменных, в отличие от лексических, вычисляется в тот момент, когда к переменной происходит обращение. Поэтому разные вызовы одного и того же кода могут дать разные результаты.

Динамические переменные помечаются вторым сигилом (твигилом) * (с намеком на wildcard).

В следующем примере функция `echo()` печатает динамическую переменную `$*var`, которая не только не определена в самой функции `echo()`, но и не является глобальной переменной в программе. Тем не менее, имя резолвится при вызове из других функций, в каждой из которых есть своя переменная с таким именем:

```
sub alpha {
    my $*var = 'Alpha';
    echo();
}

sub beta {
    my $*var = 'Beta';
    echo();
}

sub echo() {
    say $*var;
}

```

```
15 alpha(); # Alpha
16 beta();  # Beta
```

Анонимные блоки

В Perl 6 есть понятие arrow blocks (или pointy blocks) — это такие анонимные блоки-замыкания, которые возвращают ссылку на функцию и могут принимать аргументы.

Синтаксически они снабжаются стрелкой `->`, за которой следует список аргументов и блок кода:

```
1 my $cube = -> $x {$x ** 3};
2 say $cube(3); # 27
```

Здесь сначала создается блок `{$x ** 3}`, принимающий один аргумент `$x`, а за-

тем делается вызов, аналогичный вызову функции через указатель на нее: `$cube(3)`.

Такие блоки со стрелкой удобно использовать в циклах:

```
for 1..10 -> $c {
    say $c;
}
```

Фактически, `for` здесь принимает список `1..10` и блок кода с аргументом `$c`, хотя сначала может показаться, что эта конструкция — синтаксис для циклов. В следующем разделе мы вернемся к этому примеру, но уже без использования явной переменной (и поэтому без стрелки).

Список аргументов вполне может состоять и более чем из одного элемента:

```
my $pow = -> $x, $p {$x ** $p};
say $pow(2, 15); # 32768
```

То же самое применимо и к спискам:

```
for 0..9 -> $i, $j {
    say $i + $j;
}
```

В этом случае за один проход цикл будет поглощать сразу по два значения. Тело цикла отработает пять раз, печатая попарно сумму соседних цифр: 1, 5, 9 и т.д.

Плейсхолдеры

При создании анонимного блока кода, даже такого, который будет принимать аргументы, объявлять их необязательно. Perl 6 разрешает сразу использовать их подобно предопределенным переменным $a и $b в Perl 5.

В Perl 6 такие переменные должны быть снабжены твигилом ^, а порядок формальных аргументов будет соответствовать алфавитному порядку.

```
my $pow = {$^x ** $^y};
say $pow(3, 4); # 81
```

Фактические значения 3 и 4 окажутся, соответственно, в переменных $^x и $^y.

А теперь вернемся к циклу и перепишем его тело без использования явных аргументов:

```
for 0..9 {
    say $^n2, $^n1;
}
```

Обратите внимание, что, во-первых, блок кода начинается сразу без предваряющей его стрелки, а переменные $^n1 и $^n2

упоминаются в коде не в алфавитном порядке, но при этом получают правильные значения, как если бы они были указаны в сигнатуре функции в виде `($n1, $n2)`.

Наконец, плейсхолдеры могут быть именованными, вся разница в использовании внутри блока кода заключается в твигиле — теперь это двоеточие `:`:

```
my $pow = {$:base ** $:exp};
say $pow(:base(25), :exp(2)); # 625
```

Порядок значений при вызове функции теперь не имеет значения. Следующий вызов вернет тот же результат:

```
say $pow(:exp(2), :base(25)); # 625
```

Переопределение функций

Ключевое слово `multi` позволяет определить несколько функций с одним именем, которые различаются списком своих аргументов (сигнатурой). В Perl 6 аргументы функций указывают сразу в заголовке, причем, как и обычные переменные, аргументы могут быть типизированы (собственно, без указания типа мультифункции теряют смысл).

```
multi sub twice(Int $x) {
    return $x * 2;
}

multi sub twice(Str $s) {
    return "$s, $s";
}

say twice(42); # 84
say twice("hi"); # hi, hi
```

Работа этого примера довольно очевидна: когда передано целочисленное значение, вызывается `twice(Int)`, а когда строка — `twice(Str)`.

Переопределение с подтипами

Еще более интересное переопределение можно сделать, использовав подтипы. Подтипы в Perl 6 создаются с помощью ключевого слова `subset`. На основе одного из существующих типов возможно создать более узкий тип, допустимые значения которого будут проверяться условием, описанным в определении подтипа.

В следующем примере созданы два подтипа: для четных и нечетных целых чисел. Условие отбора явно определено в блоке после ключевого слова `where`.

Далее объявлены и определены две функции с одним именем `testnum`, но с агрументами разных типов.

```
subset Odd of Int where {$^n % 2
    == 1};
subset Even of Int where {$^n % 2
    == 0};

multi sub testnum(Odd $x) {
    say "$x is odd";
}

multi sub testnum(Even $x) {
    say "$x is even";
}
```

Теперь при вызове функции с именем `testnum` будет выбрана одна из двух: `testnum(Even)` для четных чисел и `testnum(Odd)` для нечетных:

```
for 1..4 -> $x {
```

```
    testnum($x);
}
```

Программа напечатает ожидаемую последовательность строк, что говорит о том, что Perl 6 выбрал правильный вариант функции, используя правила, заданные для подтипов `Odd` и `Even`.

```
1 is odd
2 is even
3 is odd
4 is even
```

Продолжение следует

В этом номере журнала вас еще ждет статья о модулях в Perl 6, а в следующем выпуске читайте о промисах (promises) и подробный рассаказ о том, как работать с грамма-

тиками (grammar).

■ *Андрей Шитов*

5 Модули в Perl 6

Краткий обзор основных моментов, которые полезно знать при работе с модулями в Perl 6

Тем, кто знаком с модулями в Perl 5, без труда разберутся с тем, как использовать их в Perl 6. Тем не менее, есть несколько важных отличий, которые необходимо знать перед началом работы.

Модули хранятся в файлах с тем же расширением `.pm`. Точно так же организуется иерархия: модуль `X::Y` компилятор будет искать в файле `X/Y.pm` в одном из предопределенных каталогов или в каталоге, указанном в опции `-I` при запуске из командной строки.

В теории, имена модулей не обязаны быть привязаны к файлам (а на практике сейчас имя модуля, указанное в самом модуле, похоже, вообще не используется). Однако пока про это, видимо, лучше не думать. Еще одна интересная особенность, а именно, возможность хранить и подключать один и тот же модуль, но разных версий, пока в Rakudo не реализована.

module

Модуль объявляется ключевым словом `module`, за которым следует название. Возможны два варианта. Объявление может быть в виде директивы в начале файла, и весь остаток файла будет телом модуля:

```
1 module X;
2
```

```
sub x() {
    say "X::x()";
}
```

Второй вариант — поместить тело модуля в блок:

```
module X {
    sub x() {
        say "X::x()";
    }
}
```

export

Переменные (`my`, `our`) и функции (`sub`), определяемые внутри модуля, по умолчанию не видны за его пределами. Для того, чтобы экспортировать имя, необходимо указать свойство (trait) `is export`:

```
module X;

sub x() is export {
    say "X::x()";
}
```

Это все, что требуется для того, чтобы функцией `x()` удалось воспользоваться в программе, которая будет использовать модуль. Никаких многословных конструкций и манипулирования массивами `@EXPORT` и `@EXPORT_OK` больше не требуется.

use

Подключение модуля — простая операция с помощью ключевого слова `use`.

Сначала создаем файл `Greet.pm`:

```
module Greet;

sub hey($name) is export {
    say "Hey, $name!";
}
```

А затем используем его:

```
use Greet;

hey("you"); # Hey, you!
```

Точно так же все работает, если имя модуля более сложное. В этом случае все импортируемые имена оказываются в текущей области видимости.

Файл `Greet/Polite.pm` с модулем `Greet::Polite`:

```
module Greet::Polite {
    sub hello($name) is export {
        say "Hello, $name!";
```

```
    }
}
```

И программа, его использующая:

```
use Greet;
use Greet::Polite;

hey("you"); # функция из модуля Greet
hello("Mr. X"); # из Greet::Polite
```

import

Ключевое слово `use` автоматически выполняет импорт имен из подключаемого модуля. Однако, если модуль определен в текущем файле в лексической области видимости (обратите внимание, что мож-

но указать модуль локальным, написав `my module`), автоматический импорт не произойдет, и придется сделать его явно:

```
my module M {
    sub f($x) is export {
        return $x;
    }
}

import M;

say f(42);
```

Без принудительного импорта (`import M;`) обращение к функции `f()` из модуля приведет к ошибке. Причем импорту подвергнутся только имена, помеченные на экспорт конструкцией `is export`.

Обратите внимание, что импорт происходит при компиляции, поэтому имена

станут доступны в программе даже выше строки про импорт:

```
my module M {
    . . .
}

say f(1);
import M;
say f(2);
```

need

Если же возникнет необходимость просто загрузить модуль, но ничего из него не импортировать, следует воспользоваться ключевым словом `need`.

Создаем модуль `N` с методом `n()` (метод создан как `our` — это важно, но без `is export`

— а это не так важно):

```
module N;

our sub n() {
    say "N::n()";
}
```

И теперь указываем, что нужен этот модуль, а потом напрямую обращаемся к его методу:

```
need N;

N::n();
```

Последовательность `use M; import M;` (именно в таком порядке) аналогична одному `use M;`.

require

Ключевое слово `require` позволяет загрузить модуль не во время компиляции (как это делает `use`), а во время исполнения.

Для примера — модуль с единственной функцией, возвращающей сумму переданных ей аргументов:

```
1 module Math;
2
3 our sub sum(*@a) {
4     return [+] @a;
5 }
```

(Звездочка в `*@a` нужна, чтобы Perl свернул все аргументы в один массив, и можно было бы написать `sum(1, 2, 3)`. Без звездочки это станет синтаксической ошибкой, поскольку метод будет ожидать массив, а не

три скаляра.)

Теперь подключаем модуль с помощью `require` и вызываем функцию:

```
1 require Math;
2
3 say Math::sum(24..42); # 627
```

Если попытаться вызвать `Math::sum()` до `require`, программа не сможет найти нужное имя. При этом импортировать метод, написав `import Math;`, тоже не получится, поскольку импорт происходит на этапе компиляции, то есть до того, как `require` загрузит модуль.

Заключение

Для удобства, вот список ключевых слов Perl 6, которые потребуются при работе с модулями:

- `use` — загрузка и импорт на этапе компиляции;
- `need` — загрузка без импорта на этапе компиляции;
- `import` — импорт имен из загруженного модуля на этапе компиляции;
- `require` — загрузка без импорта во время исполнения.

■ *Андрей Шитов*

6 Обзор CPAN за февраль 2015 г.

Рубрика с обзором интересных новинок CPAN за прошедший месяц.

Статистика

- Новых дистрибутивов — 202
- Новых выпусков — 815

Новые модули

Crypt::U2F::Server

Модуль `Crypt::U2F::Server` является обёрткой к C-библиотеке libu2f-server. Мо-

дуль реализует серверную часть протокола U2F, предназначенного для универсальной реализации двухфакторной аутентификации, когда пользователь помимо ввода пароля в браузере использует ещё физическое устройство (например, USB-донгл), которое доступно браузеру через специальное Javascript API.

Plack::Middleware::Antibot

`Antibot` — это набор фильтров для PSGI-приложений, которые позволяют бороться с отправкой форм роботами:

- FakeField — проверка, что было отправлено скрытое поле;
- Static — проверка, что был загружен статический файл (например,

css-стили) до отправки формы;
- TextCapcha — проверка, что была правильно решена текстовая капча;
- TooFast — проверка, что форма была отправлена очень быстро;
- TooSlow — проверка, что форма была отправлена через большой интервал времени.

Inline::Perl6

`Inline::Perl6`, как следует из названия, позволяет выполнять из Perl 5 код на языке Perl 6, а также загружать модули и выполнять методы объектов Perl 6. Для работы модуля требуется установить Rakudo с включённым бекендом MoarVM.

Promises6

`Promises6` — довольно любопытная реализация спецификации Promises/A+, предоставляющая синтаксис обещаний в соответствии со спецификацией ECMAScript 6. Модуль требует для работы Perl не ниже 5.20.0.

AnyEvent::TLS::SNI

`AnyEvent::TLS::SNI` — это модуль, который добавляет поддержку TLS-расширения SNI в `AnyEvent::TLS`. Указав параметр `host_name`, можно задать имя сервера, к которому будет производиться подключение. Поддержка SNI работает только для стороны клиента.

```
use AnyEvent::HTTP;
```

```
use AnyEvent::TLS::SNI;

AnyEvent::HTTP::http_get
    'https://www.goggle.com/',
    tls_ctx => {
        verify => 1,
        verify_peername => 'https
            ',
        host_name => 'www.google.
            com'
    },
    ...
```

Обновлённые модули

Gtk2 1.2495

Обновление Perl-интерфейса к библиотеке `Gtk2` вышло в конце января 2015,

но только в феврале вендоры обратили внимание, что исправленная ошибка с некорректным управлением памятью в `Gtk2::Gdk::Display::list_devices` может потенциально быть использована для выполнения произвольного кода. Всем пользователям рекомендуется обновиться.

perl 5.20.2

Вышел корректирующий релиз стабильной версии Perl 5.20.2.

- Обновлён модуль `Data::Dumper`, в котором исправлена проблема CVE-2014-4330, приводящая к рекурсии при дампе глубоко вложенных структур данных.

- Исправлен крах в `PerlIO::Sacalar` при задании файловой позиции за пределами скаляра.
- Появился новый документ perlunicook, который содержит исчерпывающую информацию о работе с Юникодом в Perl.
- Восстановлена работоспособность сборки на платформах IRIX и Tru64.
- Исправлены несколько ошибок assert в отладочных сборках Perl.
- Исправлен крах при вызове gmtime() с параметром NaN.
- Исправлено переполнение буфера и крах при компиляции определённых шаблонов в регулярных выражениях.
- Устранена утечка памяти в некоторых регулярных выражениях, появившаяся в Perl 5.20.1.

Role::Tiny 2.000000

Вышел второй мажорный релиз модуля для создания классов-ролей `Role::Tiny`. Основное несовместимое изменение — теперь `Role::Tiny` не делает предупреждения фатальными для классов-ролей, созданных с его помощью. Прагмы `strict` и `warnings` по-прежнему включены.

strictures 2.000000

Вслед за `Roly::Tiny` новая версия прагмы `strictures` отказывается от полной фатализации предупреждений. Следующие виды предупреждений больше не вызывают аварийное завершение работы:

- exec

- recursion
- internal
- malloc
- newline
- experimental
- deprecated
- portable

Mojolicious 6.0

Выпущен шестой мажорный релиз веб-фреймворка Mojolicious с кодовым названием «Clinking Beer Mugs» (звенящие пивные кружки). Вопреки традиции, разработчики не стали ждать мая (последние мажорные релизы выпускались с промежутком около одного года). Удалён внушительный список устаревших методов, переименовано множество других методов. Будьте

внимательны при обновлении.

Plack 1.0034

В новом релизе «суперклея» для веб-фреймворков `Plack` исправлена проблема в безопасности при использовании `Plack::App::File` на платформе Win32. Дело в том, что в некоторых версиях Windows возможно использование больше двух точек в фрагменте пути для перехода в вышестоящие каталоги.

Compress::Bzip2 2.22

Вышел новый релиз обёртки к библиотеке сжатия Bzip2. Новая версия содержит исправление крупной утечки памяти,

возникавшей при декомпрессии.

Perl::Critic 1.124

Новый релиз модуля для проверки исходного Perl-кода на соответствие принятым стандартам `Perl::Critic` содержит обновление для политики `ProhibitUnusedPrivateSubroutines` (запрет на неиспользуемые приватные подпрограммы), в которой появилась возможность задать исключение для некоторых файлов с помощью опции `skip_when_using`, если они используют определённый модуль. Это может быть полезно для классов-ролей, где определённые приватные методы могут использоваться в классах, использующих эту роль.

Кроме того, политики `RequireUseStrict`

и `RequireUseWarnings` теперь учитывают, что `Moose`, `Moo`, `Mouse`, `Dancer`, `Mojolicious` и некоторые другие модули автоматически включают прагмы strict и warnings.

■ *Владимир Леттиев*

7 Интервью с Владимиром Леттиевым

Владимир Леттиев (crux) — постоянный автор журнала, модулей на CPAN, основатель perlnews.ru

Как и когда научился программировать?

Все проходили основы программирования на уроках информатики в школе. Когда я учился в школе, у нас стоял класс Корветов. Нас учили рисовать на бумажке блок-схемы алгоритмов, а потом реализовывать их на Бейсике. Для меня компьютеры всегда воспринимались как чудо техники и прогресса, о которых я ещё вчера читал в фантастических рассказах, и вот они уже есть в школах и даже в продаже в магазинах электроники.

Классе в 7-м у меня появился мой первый компьютер Дельта-С (клон ZX Spectrum). Там загружался сразу REPL с Бейсиком, поэтому поневоле любой пользователь компьютера начинал программировать. Сначала я только играл в игры. Подавляющее число игр имело графическую заставку, на которой красовалась подпись «Cracked by Bill Gilbert». Мне тоже захотелось крякать игры. Где-то за 4 месяца я успел купить и зачитать до дыр книги по бейсику и ассемблеру Z80. Первой взломанной игрой оказалась игрушка «River Rescue», где я, исследуя дизассемблированный код, нашёл инструкцию по декременту счётчика жизней и заменил его пустой операцией. Потом по всем правилам оформил бейсик-загручик, поместив код на строку 0 и закрыв его от взгляда трюком с цветом фона. Поправил в графическом редакторе заставку на

«Cracked by Vladimir Lettiev» и записал на кассету. Вскоре компьютер сгорел от разряда статического электричества, и на этом карьера юного хакера оборвалась…

Какой редактор используешь?

Сейчас я использую Vim. Как и многие я был в шоке от первого знакомства с vi. Какая-то команда, запущенная в рутовом шелле открыла мне конфигурационный файл в этом редакторе. Оно пищало, мигало и запортило мне конфиг, прежде чем я смог убить его kill'ом в соседнем терминале.

Сначала редактировал в mcedit, так как он был похож на Far, который я использовал под Windows. Пробовал nano, joe, но потом всё-таки сел и изучил vim. И вот лет 10 использую только vim. Из плагинов исполь-

зую только Nerdtree, из тем предпочитаю desert или wombat256mod.

Доводилось также сделать целый проект в Padre IDE. Адаптировал своё поведение так, чтобы он не падал и, в принципе, был доволен им. Но, в любом случае, GUI-инструменты тяжело использовать через ssh-соединение, поэтому удобнее vim вряд ли что-то можно придумать.

Как и когда познакомился с Perl?

Кажется это был 2002 год. После окончания университета я остался там работать сисадмином и заодно учился в аспирантуре. Одной из первых задач, которая возникла передо мной, стало создание учёта интернет-трафика пользователей. Мы использовали прокси-сервер Squid, и нужно было распарсить лог и выполнить расчёты.

Я скопировал файл в несколько гигабайт на свой компьютер. В университете мы изучали Паскаль, поэтому я написал на Паскале программу, проверил её на паре строчек лога, а потом запустил обработку всего файла. После 15 минут ожидания я подумал, что что-то тут не так. Мой коллега, который в то время писал cgi-скрипты для веб-сайта университета, посоветовал попробовать распарсить лог Перлом, т.к. он здорово подходит для работы с текстовыми файлами.

Я взглянул на пример кода и был неприятно удивлён, что для переменных использовалась закорючка $, почти как строковые переменные в древнем Бейсике ZX Spectrum. Но, тем не менее, я изучил синтаксис по образцу какой-то готовой программы, слепил простой цикл, который построчно читал файл, сплитил строку

и проводил суммирование. Запустил программу и подумал, чем бы заняться, пока она работает, но каково было моё удивление, когда программа завершилась в считанные секунды. Это был шок. С тех пор использую Perl для любых задач с неизменным успехом.

С какими другими языками интересно работать?

Javascript. Это язык фронтенда, и его придётся изучить в любом случае, если вы пишете для веба. Jquery даже делает его немного похожим на Perl (что не удивительно, т.к. создатель фреймворка был перловиком).

Язык C прекрасен. Он позволяет писать быстрые XS-подпрограммы для Perl.

Что, по-твоему, является самым большим преимуществом Perl?

Perl позволяет очень быстро прототипировать программы. Можно быстро начать писать код, потом постепенно трансформировать программу, разнося код по модулям/-методам, когда кодовая база начнёт расти.

Как-то я дорабатывал реализацию одного сетевого протокола на C, и это заняло уйму времени, особенно на отладке. Потом передо мной встала задача обрабатывать получаемые данные прикладного уровня и записывать их в СУБД. Это уже явно требовало языка более высокого уровня. Я за пару недель переписал реализацию протокола на чистом Перле в отдельный модуль и продолжил реализовывать логику приложения. Уровень сложности и время разработки при переходе снизились

на порядок.

Ещё Perl универсален. На нём можно писать короткие однострочники, скрипты, модули и огромные проекты. Функциональный стиль, ООП, DSL — Perl можно адаптировать под любой стиль программирования. Perl позволяет менять свой синтаксис, что позволяет на базе Perl строить надстройки. Посмотрите на Rex — это отличный пример как можно писать сжатые сценарии на языке более высокого уровня на основе Perl.

Что, по-твоему, является самой важной особенностью языков будущего?

Период полураспада. Мне кажется, что все языки будущего будут появляться и исчезать. Вечным будет только C, так как на нём и будут писать новые языки.

Что думаешь о будущем Perl?

Я смотрю на индекс TIOBE и особо не волнуюсь за будущее Perl. Perl есть везде. На любой платформе, в дистрибутиве любой юниксподобной системы его всегда можно будет найти.

Perl по-настоящему свободный проект и принадлежит сообществу. Нет одной компании, узурпирующей направление развития. Сообщество очень внимательно и дружелюбно к новичкам. Любой человек может влиться в разработку и получить максимальную поддержку. Perl имеет чёткий план регулярных релизов и здоровую ротацию ответственных за релизы. Такая организация даёт +100500 к Viability.

Мне бы очень хотелось увидеть в будущем улучшенную организацию CPAN.

Мне кажется, что очень не хватает своего GPG-сервера ключей для CPAN-авторов, которыми бы можно было подписывать релизы.

Просто напрашивается наличие аналога сервиса travis-ci, заточенного под Perl. Как обходной вариант есть специальный хелпер, но если бы удалось создать специализированную систему, которая объединит разрозненные системы Perl QA cpantesters, coverage и другие метрики исходного кода, то это было бы просто фантастикой.

Почему пишешь статьи для Pragmatic Perl? Откуда берешь идеи?

Однажды я написал статью по настройке Postfix и LDAP и опубликовал её на одном ресурсе. Через полгода я по этой статье как в первый раз настраивал новый сервер.

Тогда же я и осознал, что любые действия необходимо записывать, т.к. память это самая ненадёжная штука.

Когда начал писать статьи для Pragmatic Perl я понял, что можно изучать новые вещи, до которых никак не доходили руки, а статья — это отличный повод, а заодно и шпаргалка на будущее. Так, например, я разбирался с DBIx::Class, который бы ни в жизни по другому бы не изучил. Поэтому я всем советую, если есть желание что-то выучить, то учите и параллельно пишите конспект. Из конспекта получится хорошая статья, которую можно опубликовать и получить плюс в карму. Все ваши ошибки подметят и поправят — это уже избавит вас от прохода по граблям. Win-Win.

Что касается источников идей, то их два. Первый — это публикации в блогах, но-

востях, рассылках. Если проскакивает какая-то интересная тема, я могу сделать пометку и развить её в статью. Второй источник — CPAN. Готовя обзор CPAN, я просматриваю список из порядка 1000 модулей с их описанием и изменениями. Какой-то модуль или что-то, связанное с ним может стать предметом для статьи.

Чем так заинтересовал HTTP2, что даже написал реализацию Protocol::HTTP2? На какой стадии находится этот модуль? Что такое Shuvgey?

В начале 2014 года я прочитал книжку Ilya Grigorik «High-Performance Browser Networking». Там в частности рассказывали о протоколе HTTP2, который был создан на основе протокола SPDY от Google. Я и раньше слышал о нём, но теперь прочитал о том, как он устроен. Это показалось мне

интересным и я подписался на рассылку ietf-http-wg, в которой обсуждался черновик стандарта нового протокола HTTP2. Довольно известные в мире веба люди предлагали новые фичи, делились своим мнением, звучали и критические отзывы, но в целом дискуссия была продуктивной.

Вскоре я уже знал про все существующие реализации и видел, что одна из существующих реализаций на Perl'е уже безнадёжно устарела (по факту она толком и не работала). Взглянув на код я понял, что там идёт привязка к `IO::Async`, а мне было интересно получить что-то работающее без привязки к конкретным событийным фреймворкам. Так и началась работа над `Protocol::HTTP2`.

Мне очень понравился интерфейс `Protocol::WebSocket`, я взял его как основу, но

потом даже ещё больше упростил. Первый работающий релиз `Protocol::HTTP2` вышел уже в мае 2014 года, где был реализован 12-й черновик спецификации протокола. После этого спецификация менялась уже только в сторону упрощения: с каждым новым черновиком я просто удалял куски кода.

Сейчас работа над спецификацией HTTP2 официально завершена. Вот уже на днях ожидается официальный номер RFC, после чего начнётся, нет, не правильно, **продолжится** его экспансия в веб. Вот вам простой факт: Firefox 35 по умолчанию поддерживает http2 (черновик 14), за месяц его функционирования обнаружили, что 9% соединений к сайтам происходило по http2. Это говорит о том, что в отличие от ipv6, http2 довольно быстро станет доминирующим протоколом веба.

`Protocol::HTTP2` в целом работоспособен, там есть несколько некритичных недоделок (например, приоритеты потоков), которые можно будет закрыть, как только на них появится минимальный спрос.

В качестве примера реализации веб-сервера на основе `Protocol::HTTP2` я создал Shuvgey — асинхронный однопоточный PSGI веб-сервер на основе AnyEvent с поддержкой протокола http2. Он похож на Twiggy, только работает с http2 и может использовать TLS. Кстати, сайт проекта крутится именно на шувгее, что также даёт вам возможность оценить работоспособность серверной реализации http2. Чтобы его увидеть вам потребуется Firefox 35 или Chrome 40.

Вообще, Шувгей — это собирательное на-

звание нечистой силы у коми-зырян и коми-пермяков. Материализуется в виде сильного ветра, вихря. Это название мне показалось интересным, своеобразный ответ питоновскому Торнадо, а также дань карго-культу странных названий из национального фольклора.

Как возникла идея perlnews.ru? Как дела с посещаемостью, какие дальнейшие планы?

«Pragmatic Perl» — это отличное место, чтобы опубликовать уникальную статью с примерами кода и какими-то важными результатами. Но переводные материалы, короткие новости уже выходят за рамки формата. Персональный блог тоже плохое место для подобных вещей, так как если пост о том, как вчера стоял в очереди в регистратуру поликлиники, соседствует с

новостью о том, что вышел новый Perl, и там такие-то улучшения, то это выглядит дико. Соответственно, требуется отдельный обезличенный тематический ресурс, специализирующийся на новостях.

Идея возникла в конце января, сайт заработал 2 февраля, 3 февраля его здорово пропиарили на форуме и в рассылке Pragmatic Perl, что дало 159 посещений 122 пользователей. Сейчас в среднем около 45 сеансов в день. Примерно 40 подписчиков по RSS. Треть посетителей приходит по ссылкам из твиттера, ещё треть, по всей видимости, добавила ресурс в закладки.

По статистике около 62% посещений из России, 13% Украина, 3,5% Беларусь, также немного Нидерланды, США, Германия, Таиланд. После того, как был опубликован пересказ интервью Рикардо Сигнеса под-

касту rebuild.fm, пост заметил Миягава и написал в твиттере, что ему понравилось, прочитал его через Google Translate. Это вылилось в 23 хита из Японии, так что о русскоязычном новостном ресурсе о Perl узнали и в Японии.

В планах пока ничего экстраординарного. Думаю сделать опрос, чтобы выяснить, какие темы более интересны читателям, чтобы скорректировать приоритеты. Отвечать можно в комментариях к этому интервью :)

Как можно написать новость на perlnews.ru?

Думаю, что ресурс здорово выиграет, если на нём появится больше новостей от других авторов. Для этого потребуется лишь немного навыков работы с github.

Исходный код сайта доступен на github –

perlnews.ru. Из данного репозитория с помощью Statocles генерируется статический сайт.

Чтобы добавить свою новость, исправить опечатку или внести новый ресурс в перечень полезных русскоязычных ресурсов, нужно форкнуть указанный выше репозиторий, внести изменения и отправить запрос на слияние (pull request). Более подробная информация есть в README.md.

Есть ли какие-то другие проекты, о которых хотел бы рассказать?

Есть проект реализации протокола TLS на чистом Perl Protocol::TLS, который на данный момент уже поддерживает версию TLS 1.2. Смысл проекта состоит в том, чтобы уйти от зависимости на OpenSSL, который развивается очень уж неспешно и

консервативно. Попробуйте обновить libssl в каком-нибудь CentOS до свежих версий с поддержкой всех новых фич безопасности, шифров и последних версий протокола TLS. Это практически нереальный квест, так как практически всё современное программное обеспечение линкуется с libssl.

К сожалению, нехватка свободного времени сейчас сказывается на дальнейшем развитии проекта.

Где сейчас работаешь? Сколько времени проводишь за написанием Perl-кода?

Я работаю в крупной энергетической компании. Занимаюсь телекоммуникациями, поддержкой средств мониторинга и промышленных систем учёта.

Программирование не входит в круг моих рабочих обязанностей, но это единственный способ облегчить свою жизнь. Регулярно приходится писать клей между различными системами, вносить улучшения в систему мониторинга, создавать и развивать разноплановые внутренние веб-сервисы.

Вне работы программирую на Perl в основном для своих открытых проектов на github. Но это происходит импульсивно, т.е. возникает вдохновение, пишешь код недели на пролёт, а потом выгораешь и ждёшь очередного озарения.

Стоит ли советовать молодым программистам учить сейчас Perl?

В первую очередь я бы посоветовал молодым ребятам обратить внимание на

свободное программное обеспечение. Постарайтесь выбрать такие программы и окружение, которые являются свободными. Это может быть Linux или одна из свободных BSD-систем. Выбирайте любой свободный язык: Perl, PHP, Python, JavaScript и прочие. Свободный редактор Vim/Emacs/Eclipse. Свободный браузер: Firefox, Chromium и другие. Свободную СУБД: PostgreSQL/MySQL и подобные. Но без фанатизма (привет, Nvidia).

Это простое правило позволит вам не разочароваться в жизни и не попасть в кабалу к проприетарному вендору, от прихотей которого будет зависеть ваше профессиональное будущее.

Perl стоит выучить, даже если вы не программист. Он позволит вам запросто решать различные задачи.

Вопросы от читателей

Надолго ли хватит сил на perlnews.ru?

Надеюсь, что сил хватит. На самом деле, писать короткие новости и переводы из блогов на порядок проще, чем придумывать темы и писать большие статьи. Выделив в дневном расписании какое-то небольшое время на выпуск новостей можно успешно поддерживать ресурс жизнеспособным.

Но было бы ещё интереснее, если бы появились другие авторы, тогда каждая новость могла бы прорабатываться более тщательно, что повысило бы качество материала для посетителей.

Что значит crux?

В студенчестве я играл в одной блэк-метал-банде местной андерграунд-сцены, мы даже записали один альбом, который назвали «crux anasta» (да, с опечаткой, правильно пишется «crux ansata»), это один из главных символов древних египтян. Позже, когда мне нужно было придумать свой первый рабочий логин для linux, я взял этот прикольный короткий *крукс*.

■ *Вячеслав Тихановский*