Pragmatic Perl 2

pragmaticperl.com

Выпуск 2. Апрель 2013

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

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

Авторы статей: Денис Федосеев (alpha6), Евгений Ардаров (spleenjack), Владимир Леттиев (crux), Дмитрий Шаматрин (justnoxx).

Выпускающий редактор: Вячеслав Тихановский (vti)

Ревизия: 2014-12-02 16:45

Оглавление

1. От редактора

Когда мне пришла в голову идея сделать журнал про Perl на русском языке, я и не думал, что его первый выпуск вызовет такую бурную дискуссию. Несмотря на то, что выпуск был пилотным, кое-где встречались недочеты и неточности, его встретили хорошо.

Большое спасибо нашим новыми читателям за комментарии и отзывы. Особенно спасибо за конструктивную критику и советы. Мы постарались учесть все, о чем вы написали. Например, уже после выпуска мы добавили возможность комментировать каждую статью отдельно и выпуск в целом. Больше внимания стали уделять качеству статей и достоверности представленной в них информации.

Мы продолжаем искать авторов. В этом номере к нам присоединились Денис Федосеев, Евгений Ардаров и Дмитрий Шаматрин. Как вы, возможно, уже заметили на данном этапе развития журнала статьи не объединяются по какой-то отдельной тематике. Принимаются любые интересные и полезные материалы. Если у вас есть желание поучаствовать в журнале, пожалуйста, пишите нам на editor@pragmaticperl.com.

С момента выпуска первого номера появилась полноценная подписка. Рассылаться будут свежие выпуски, а также некоторые важные новости журнала. Если вы еще не подписались, самое время сделать это на странице рассылки. Количество подписчиков является неким индикатором интереса к нашей работе. Если вам нравится то, что мы делаем, советуйте друзьям, знакомым и коллегам. И это, в свою очередь, будет мотивировать нас на интересные и регулярные выпуски.

Приятного чтения.

■ *Вячеслав Тихановский*

2. Преобразование XML в Perl-структуры с помощью XML::Simple

XML::Simple не рекомендуется к использованию для работы с XML в современном Perl, однако бывают ситуации, когда выбор уже сделан и необходимо поддерживать имеющийся код. Для таких случаев и была написана данная статья.

Преобразование XML в Perl-структуры и обратно довольно часто встречающаяся задача. Конечно, в большинстве таких случаев XML-файлы имеют простую структуру. Однако, решение совсем не тривиально.

`XML::Simple` — очень популярный модуль времен, когда `XML::LibXML` был большим и слишком сложным. С тех пор утекло много воды, появился Modern Perl, интерфейс `XML::LibXML` существенно упростился, а `XML::Simple`, наоборот, перестал рекомендоваться к использованию, причем самим автором модуля.

Не рекомендуется также использовать библиотеку для работы с большими или сложными XML. Модуль предоставляет простой интерфейс и достаточное кол-во настроек обработки — но платой за это является не совсем очевидная работа в сложных случаях. Проще говоря, приходится изрядно потрудиться, чтобы заставить его выдавать нужный результат на сложных структурах данных.

Создание XML

Мы будем рассматривать задачу создания XML-документа, потому что с разбором заданной структуры никаких проблем нет, и результат там практически всегда логичен и понятен.

Итак, на сервер приходит запрос списка объектов и их значений. На выходе мы хотим получить следующий результат:

```
1 <?xml version="1.0" encoding="UTF8"?>
2 <cats>
3   <cat name='Daisy'>4</cat>
```

```
  <cat name='Abby'>5</cat>
</cats>
```

Попробуем построить документ, воспользуясь XML::Simple:

```
use XML::Simple;

my $xs      = XML::Simple->new();
my $hashref = {
    cats => {
        cat => [
            {
                name  => 'Daisy',
                value => 4
            },
            {
                name  => 'Abby',
                value => 5
            },
        ]
    }
};

say $xs->XMLout($hashref);
```

... на выходе получилось совсем не то, что нужно. Декларации XML-документа нет, все обернуто в `<opt>`:

```
<opt>
  <cats>
    <cat name="Daisy" value="4" />
    <cat name="Abby" value="5" />
  </cats>
</opt>
```

После более детального ознакомления с документацией, напишем следующий код:

```
my $hashref = {
    cat => [
        {
            name    => 'Daisy',
            content => 4
        },
        {
            name    => 'Abby',
            content => 5
```

```
        },
    ]
};

say $xs->XMLout(
    $hashref,
    XMLDecl  => '<?xml version="1.0" encoding="UTF8"?>',
    RootName => 'cats',
);
```

То, что нужно!

Итак, мы принудительно указали декларацию XML-документа, задали корневой элемент, а так же изменили `value` на `content` — это ключевое слово в имени ключа заставило `XML::Simple` перенести значения объектов в содержимое тегов, а не в атрибуты. Как видно, довольно много опцией для такого простого файла.

Изменив `value` на `content`, мы использовали возможности преобразования данных которые нам предоставляет модуль. Однако, это не всегда удобно, хотя и, зачастую, повышает читаемость исходных данных. В большинстве же случаев можно обойтись стандартным поведением: по умолчанию модуль преобразовывает структуру вида `{ key => 'value' }` в xml вида `<key attr="value" />`, а структуру `{key => ['value']}` в `<key>value</key>`. Т.е. каждая простая строка будет отображена в виде атрибута, а массив будет превращен во вложенные элементы.

Как же нужно изменить код, чтобы добавить новые значения?

```
<?xml version="1.0" encoding="UTF8"?>
<cats>
  <cat name='Daisy'>
    <age>4</age>
    <weight>3.5</weight>
  </cat>
  <cat name='Abby'>
    <age>5</age>
    <weight>6</weight>
  </cat>
</cats>
```

Просто добавив поля:

```
{
    name => 'Daisy',
    weight => 4,
    age => 3.5,
},
```

Но мы получим не совсем то, что ожидалось:

```
<cat name="Daisy" age="3.5" weight="4" />
```

Более логичным было бы сделать следующим образом:

```
{
    name => 'Daisy',
    content => {
        weight => 4,
        age => 3.5,
    },
},
```

Но на самом деле правильный вариант:

```
name => 'Daisy',
weight => {
    content => 4,
},
age => {
    content => 3.5,
},
```

Что в целом объяснимо — мы превращаем age в структуру с ключевым словом content, которое подставляет свое значение в содержимое тегов. Однако, из примера выше видно, что если в content мы подставим структуру — то все особенные свойства сразу исчезнут и это будет обычный тег. Но что делать, если в исходных данных есть такое название столбца и вы хотите чтобы он был атрибутом, а не проявлял свои особенные свойства в самый неподходящий момент? Делаем так:

```
$xs—>XMLout($hashref ,
    XMLDecl => '<?xml version="1.0" encoding="UTF8"?>',
    RootName => 'cats',
    ContentKey => ''
);
```

И теперь ни один из тегов не будет рассмативаться как содержимое, а только как атрибуты. Если нужно, наоборот, добавить свои теги — то просто указываем их в качестве списка. Еще один важный момент — указанные элементы затрут список по умолчанию, так что в него нужно включать все теги, которые необходимы.

Также можно вообще запретить использование тегов в качестве атрибутов:

```
$xs->XMLout($hashref , NoAttr => 1);>

<cat>
    <name>Abby</name>
    <age>5</age>
    <weight>6</weight>
</cat>
```

Разбор XML

После генерации XML-документа рассмотрим особенности разбора. В этом случае все несколько проще:

```
my $ref = $xs->XMLin($xml_file);
```

Если мы возьмем сгенерированный выше XML и передадим его парсеру — то получим не ту структуру из которой XML создавался. Так как тег `name` заставит парсер создать структуру вида:

```
'cat' => {
   'Daisy' => {
               'weight' => '4',
               'age' => '3.5'
             },
   'Abby' => {
              'weight' => '6',
              'age' => '5'
            }
}
```

Чтоб этого избежать задаем опцию `KeyAttr => ''`:

```
my $ref = $xs->XMLin($xml_file, KeyAttr => '');

```

```
'cat' => [
   {
        'weight' => '4',
        'name' => 'Daisy',
        'age' => '3.5'
   },
   {
        'weight' => '6',
        'name' => 'Abby',
        'age' => '5'
    }
]
```

Следующий нюанс — как видно выше, список объектов представляет собой массив хешей с атрибутами. Но что будет, если объект только один? Список превратится в хеш и нам придется отслеживать этот момент для правильной обработки. К сожалению, опции для нормального решения при разборе нет. Использование `ForceArray => qw/cat/` приводит к вот таким последствиям:

```
'cat' => [
   {
        'weight' => [
                  '4'
            ],
        'name' => [
                 'Daisy'
            ],
        'age' => [
              '3.5'
            ]
    }
]
```

Опция `GroupTags => { cats => 'cat' }` тоже не приводит к нужному эффекту. Причем ни в случае одного элемента, ни в случае нескольких.

Выводы

В целом, `XML::Simple` реализует простой интерфейс к XML (что логично следует из названия модуля). Проблема в том, что этот интер-

фейс реализует большое кол-во опций и это приводит к нелогичному поведению и запутанности при работе с XML со сложной структурой. Так что, если нужно быстро создать или разобрать простой XML небольшого объема, то можно смело использовать этот модуль. Он достаточно быстр и стабилен. Но если необходимо работать со сложными структурами и большими объемами — то лучше использовать `XML::LibXML` и аналоги. При более высоком пороге вхождения они дадут более предсказуемое поведение и высокую скорость обработки данных.

■ *Денис Федосеев*

3. Удобное логирование с Log::Any

Log::Any — исключительно полезная библиотека, которая скрывает внутренности механизма логирования от кода, которому это логирование нужно.

Какую проблему решает?

Существует огромное количество разных библиотек на CPAN, помогающих организовать логирование в своём проекте. Выбор есть на любой вкус: простые и сложные, быстрые и не очень. К сожалению, такое разнообразие и вездесущий TIMTOWTDI имеет и негативную сторону.

Допустим, вам понадобилось добавить логирование к вашему модулю управления кофеваркой. Используется он в веб-интерфейсе удалённого управления домашней кофеваркой, а также в скрипте для варки кофе не вылезая из консоли. Представим, что первый написан на Mojolicious, второй — собран по-быстрому на коленке на чистом Perl, без всяких фреймворков. Ах да: а ещё есть неограниченное количество благодарных программистов, которые интегрировали кофеварочный модуль в свои проекты.

При этом у всех потребителей вашего модуля есть собственные соображения по поводу логирования в их проекте: один пишет в `STDERR`, второй ограничивается Mojo::Log, третьего устраивает Log::Dispatch, а четвёртый прикрутил Log::Log4perl с развесистой конфигурацией.

Какой механизм логирования выбрать в сложившейся ситуации? Так, чтобы логирование всё-же появилось, и не представляло собой нестандартную реализацию в виде накопления лога в какой-нибудь `last_log_messages()`?

`Log::Any` решает именно эту проблему: унификация интерфейса логирующих вызовов. Идейно он напоминает известные AnyEvent, CHI и DBI.

Как работает?

Довольно просто, на самом деле. Необходимо только разделить логирующий механизм на две составляющие:

- *создание данных* (*log production*); включает в себя весь внешний интерфейс: методы передачи данных (`debug()`, `warning()` и прочие) и флаги проверки включенного уровня логирования (`is_debug()` и так далее); в каком-либо виде это присутствует во всех логирующих библиотеках;
- *потребление переданных данных и их обработка* (*log consumption*); фактически, это конфигурирование (куда направлять вывод, как его форматировать, фильтровать и так далее) и непосредственный код реализации; выбор реализации — это зона ответственности приложения и её разработчик волен включать любой подходящий механизм.

Внешний интерфейс и предоставляет `Log::Any`. Он достаточно простой, обобщённый и легковесный. А связывание интерфейса с логгером выполняется через базовый модуль Log::Any::Adapter.

Внешний интерфейс

Доступные методы упрощены до предела и делятся на два типа:

- *logging*methods_ и их псевдонимы: `trace`, `debug`, `notice`, `warning`, `error`, `alert` и прочие;
- *detection*methods_ и их псевдонимы: `is_trace`, `is_debug` и так далее
- `printf`-версии логирующих методов, умеющие показывать сложные структуры: `debugf`, `errorf` и остальные;
- установка нужного адаптера через `set_adapter()`; по умолчанию, включается `Null`-адаптер, который просто игнорирует все получаемые данные.

Именно на этот унифицированный интерфейс и нужно закладываться в своём модуле. Теперь эта задача делегируется разработчику приложения, использующего модуль. Вот всё, что нужно сделать:

```
use Log::Any '$log';
$log->debug("что-то произошло");
```

То есть, импортировать логгер как объект `$log` для вызова его интерфейсных методов. И не забыть прописать `Log::Any` в зависимостях модуля, проекта или приложения.

Следует сделать ремарку о глобальных переменных, которые, как известно, ни к чему хорошему не приводят. Здесь как раз тот случай, где использование глобальной переменной на уровне класса оправдано. Особенно это чувствуется в большом проекте, с большим количеством своего кода, модулей, классов, каждый из которых что-то логирует в процессе своей работы. Передавать объект логгера через accessor'ы и поля классов при их инициализации — неудобно: нужен он абсолютно всем и каждому. Замучаешься. Кстати, в том же Python такой подход с его `import logging` — повсеместная практика.

Адаптер

Весь задекларированный интерфейс должен реализовываться через специальный адаптер. На CPAN уже имеется множество их вариантов под самые известные логгеры, например, Log::Any::Adapter::Log4perl или Log::Any::Adapter::Mojo.

Для подключения нужного адаптера в приложении достаточно примерно такого кода в инициализирующей части:

```
use Log::Any::Adapter;
Log::Any::Adapter->set('Log4perl');
```

И, опять же, не забыть прописать выбранный адаптер в зависимостях.

В создании своего адаптера нет ничего сложного, достаточно написать реализацию тех самых методов, соответсвующих интерфейсу

`Log::Any`. Распространённая практика — создавать методы с помощью шаблонов кода, так как чаще всего они сильно похожи друг на друга и отличаются только именами вызываемых методов логгера. Здесь пригодятся списки `Log::Any->logging_methods` и `Log::Any->detection_methods`, а также вспомогательная функция `Log::Any::Adapter::Util::make_method()`. Подробности можно найти в Log::Any::Adapter::Development. А пример написания своего собственного простого адаптера будет дальше.

Пример использования

Возьмём всё тот же пример с кофеварочным модулем и одного из его потребителей — консольный скрипт. Скрипт совсем простой и, по сути, является обвязкой над кофеварочным модулем, который очень хочется выложить на CPAN. Только проблема в том, что он использует корпоративный модуль для логирования, который нельзя публиковать.

Вот здесь нам и пригодится Log::Any.

Допустим, так выглядит наш корпоративный логгер (пример схематичный — настоящий модуль может быть намного сложнее):

```
1  package Logger;
2
3  use strict;
4  use warnings;
5  use v5.10;
6
7  use Term::ANSIColor ':constants';
8
9  sub Debug { say STDERR         shift        }
10 sub Info  { say STDERR GREEN,  shift, RESET }
11 sub Warn  { say STDERR YELLOW, shift, RESET }
12 sub Error { say STDERR RED,    shift, RESET }
13
14 1;
```

От него нам предстоит отвязаться.

Так выглядит консольный скрипт-обвязка:

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

use strict;
use warnings;

use CoffeeMaker;

CoffeeMaker->make(80);
```

А вот и сам кофеварочный модуль. Для наглядности и краткости с одной единственной функцией, которая не делает ничего полезного, кроме гипотетического логирования:

```
package CoffeeMaker;

use strict;
use warnings;
use utf8;

use Logger;

sub make {
    my $self = shift;
    my ($temperature) = @_;

    Logger::Debug("лоток загружен, вода есть");
    Logger::Info("варю кофе, температура $temperature");
    Logger::Warn("осторожно, горячо!")
      if $temperature > 70;
    Logger::Error("непредвиденная проблема");
    Logger::Info("готово");

    return;
}

1;
```

В нём во весь рост видна проблема сильной связности и зависимости от реализации механизма логирования.

Устраняется она легко:

```
package CoffeeMaker;

use strict;
use warnings;
use utf8;
```

```
6
7  use Log::Any '$log';
8
9  sub make {
10     my $self = shift;
11     my ($temperature) = @_;
12
13     $log->debug("лоток загружен, вода есть");
14     $log->info("варю кофе, температура $temperature");
15     $log->warn("осторожно, горячо!")
16       if $temperature > 70;
17     $log->error("непредвиденная проблема");
18     $log->info("готово");
19
20     return;
21 }
22
23 1;
```

Что мы сделали:

- заменили `use Logger` на `use Log::Any '$log'`, то есть импортировали объект `$log`, который предоставляет унифицированный интерфейс логирования и скрывает выбранный потребителем модуля механизм логирования;
- заменили все вызовы `Logger::*` на `$log->*`.

После того, как `CoffeeMaker` перешел на `Log::Any`, при вызове `make()` из скрипта никаких логов мы не увидим. Это сделано специально, так как на данный момент `Log::Any` не знает, какую реализацию логирования мы хотим использовать, поэтому по умолчанию включает `Null`-адаптер. Теперь у нас развязаны руки и мы можем использовать любой понравившийся логгер (всё с того же CPAN). Или, наоборот, ничего не добавлять и не зависеть от дополнительного модуля, если логи нам не интересны.

Свой адаптер

А теперь попробуем написать свой адаптер для `Log::Any`, который будет проксировать логирующие методы в наш корпоративный логгер.

```
1  package Adapter;
2
3  use strict;
4  use warnings;
5
6  use Log::Any::Adapter::Util 'make_method';
7
8  use Logger;
9
10 use base 'Log::Any::Adapter::Base';
11
12 my %pairs = (
13     Debug => [qw/debug/],
14     Info  => [qw/info inform/],
15     Warn  => [qw/notice warn warning/],
16     Error => [qw/err error fatal crit critical alert
          emergency/],
17 );
18
19 while (my ($function, $methods) = each %pairs) {
20     my $code = <<EOC;
21 sub {
22     shift;
23     \@_ = (join '', \@_);
24     \&Logger:\:$function;
25 }
26 EOC
27
28     my $sub = eval $code;
29
30     for my $method (@$methods) {
31         make_method($method, $sub);
32     }
33 }
34
35 for my $method (Log::Any->detection_methods) {
36     make_method($method, sub { 1 });
37 }
38
39 1;
```

Разберём по-порядку. Есть инициализация:

- импортируем `make_method` из утилит базового адаптера;
- делаем класс зависимым от базового адаптера, именно его интерфейс необходимо реализовать;
- задаём хеш с маппингом «функция `Loggera`» <-> «интерфейсные функции адаптера»; интерфейс у адаптера содержит самые распространённые названия логирующих функций, которые могут обозначать одно и то же: в нашем случае, например, реализация `warn()` и `warning()` будет использовать одну и ту же функцию `Logger::Warn()`.

И генерация нужных функций:

- для каждой функции `Logger` генерируем код с помощью шаблона;
- создаём через `make_method()` все связанные с этой функцией методы (по списку);
- а также реализуем все `detection_methods()`, которые всегда возвращают `true`: делаем вид, что логируется любой уровень важности.

Реальный адаптер, конечно, может быть намного сложнее. Для более глубокого погружения имеет смысл изучить реализации уже существующих адаптеров. А при написании тестов воспользоваться удобным Log::Any::Test.

Теперь подключаем адаптер к скрипту и возвращаем назад «раскрашенные» логи от `Logger`:

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

use strict;
use warnings;

use Log::Any::Adapter;
Log::Any::Adapter->set('+Adapter');

use CoffeeMaker;
```

```
10
11 CoffeeMaker->make(80);
```

Чуть больше сахара

Для ещё большего упрощения логирования в повседневных скриптах, есть интересный модуль Log::Any::App. Фактически, это заранее сконфигурированный адаптер к `Log4perl`, который умеет адаптироваться под разные контексты (one-liner выводит лог на экран, скрипт — в файл, демон — в `syslog`). Дефолтная конфигурация продумана заранее за вас и подходит в большинстве ситуаций. Есть и возможность точечной настройки.

То есть, вместо:

```
1 use Log::Any '$log';
2 use Log::Any::Adapter;
3 use Log::Log4perl;
4 my $log4perl_config = '
5   some
6   long
7   multiline
8   config...';
9 Log::Log4perl->init(\$log4perl_config);
10 Log::Any::Adapter->set('Log4perl');
```

делаем так:

```
1 use Log::Any::App '$log';
```

и логгер готов к работе.

Заключение

Разделение интерфейса логгера и его реализации помогает упростить зависимости модуля и отдать право выбора логгера непосредственному потребителю. `Log::Any` удобен в использовании и для авторов, выкладывающих свои работы на CPAN, и для интеграции

в корпоративные проекты с большим количеством собственных модулей и классов. Также упрощается тестирование отдельно взятых модулей, так как устраняется необходимость инициализировать и поднимать всю существующую экосистему логгера только для того, чтобы модуль вообще мог работать — это достигается включенным по умолчанию `Null`-адаптером.

■ *Евгений Ардаров*

4. Debug-fu в стиле Perl

Чаку Норрису не нужен отладчик, он просто пристально смотрит на код пока тот сам не сознается, где в нём баги. Если вы не Чак Норрис, то вам придётся изучить кунг-фу отладки кода, чтобы одержать победу в поединке с собственными (или чужими) ошибками.

Немного философии о программировании и ошибках

Когда программа начинает выполняться не так, как ожидается, это значит, что где-то в ней содержится ошибка, а скорее всего даже несколько. Ошибки присутствуют всегда, даже когда кажется, что их нет. В этом утверждении нет противоречия, вспомните, как мы приходим в изумление, если наш код выполнился с первого раза и мы смотрим исходники снова, чтобы убедиться, что там точно всё в порядке. При этом вы обязательно что-нибудь найдёте, хотя бы грамматическую ошибку в комментарии.

Если задуматься, то практически все современные языки программирования, все рекомендуемые практики программирования ориентированны на то, чтобы мы совершали меньше ошибок и быстрее их находили, например:

- парадигма ООП — разложите код в маленькие чёрные ящички, чтобы я легко мог найти именно тот, в котором есть ошибки и мог его независимо подправить;
- TDD — напишите тест, чтобы проверить, что в коде нет ошибок, до того как написан сам код.

Кстати, почему программистам важна читаемость кода? Они спорят о пробелах и табуляциях в тексте, используют моноширинный шрифт и подсветку синтаксиса в редакторе. Думаете это нужно, чтобы понять код? Да, конечно, но если ваш код работает — зачем его

читать и понимать? Читаемость кода, комментарии, тесты, документация — всё это служит одной задаче — найти ошибки в программе. Только когда что-то ломается, программисты сломя голову бросаются читать код и расстраиваются, если видят там нечитаемую кашу.

Таким образом, мы только что познакомились с самым первым отладчиком, который используют программисты — собственный мозг. Они смотрят на код, запускают его в виртуальной машине внутри черепной коробки и пытаются понять, что не так. Возможности этого отладчика очень индивидуальны, и, как вы поняли, сильно зависят от таких характеристик кода как читаемость, наличие комментариев и документации. В общем все те вещи, на которые компилятор не обращает внимания. По этим причинам в данной статье мы его не рассматриваем.

Начало программы — самый важный участок кода

Самый первый этап проверки на правильность кода начинается с валидации его синтаксиса. Perl даёт большую свободу программисту при написании кода, что позволяет писать как короткие и эффективные однострочники, так и гигантские модульные комплексы. Если вы *не* пишете однострочник или обфусцированный код, то рекомендуется начинать программу с использования прагм `strict` и `warnings`.

```
1 use strict;
2 use warnings;
```

В этом случае включается режим строгой проверки на отсутствие небезопасных конструкций, на наличие объявления переменных, подпрограмм и ссылок. Кроме того, интерпретатор может выдавать предупреждения в процессе выполнения кода. Всё это очень важно и позволяет подсказать вам, где находятся потенциальные источники проблем в вашем коде. Если вы новичок и вам трудно разобраться, что означают все эти непонятные предупреждения, можно использовать прагму:

```
1 use diagnostics;
```

Которая будет развёрнуто объяснять вам, что означает то или иное предупреждение.

Кроме того, пользуясь случаем, хочу порекомендовать использовать CPAN-модуль `strictures`:

```
use strictures 1;
```

Эта прагма эквивалентна такому коду:

```
use strict;
use warnings FATAL => 'all';
```

Т.е. теперь любые предупреждения будут трактоваться как ошибки. Кроме того, в случае, если `strictures` выявит, что запуск происходит в условиях разработки (запуск скрипта из каталогов `t`, `xt`, `lib`, `blib` и наличие в текущем каталоге `.git` или `.svn`), это расширяется в конструкцию:

```
use strict;
use warnings FATAL => 'all';
no indirect 'fatal';
no multidimensional;
no bareword::filehandles;
```

Три дополнительные прагмы являются соответствующими модулями на CPAN и если их нет у вас в системе, то использоваться они не будут. Но они, несомненно, полезны, так как позволяют обнаруживать потенциальные ошибки в вашем коде на этапе разработки, отловить которые бывает очень сложно.

`no indirect 'fatal'` — запрещает использовать непрямые вызовы методов, например:

```
my $obj = new Module::Name;
```

вместо записи:

```
my $obj = Module::Name->new()
```

Почему первый вариант не стоит использовать? Посмотрите код:

```
sub mess($) {
    shift->{message}
}
```

```
4
5 my $y = mess { message => "hello world!" };
```

Вроде всё отлично, но попробуйте передвинуть определение функции `mess`, после её первого использования. Это, по идее, должно быть вполне законно, но в итоге приводит к странной ошибке:

```
Can't locate object method "mess" via package "message" (
perhaps you forgot to load "message"?)
```

Ссылку на хеш Perl воспринял за класс и попробовал вызвать метод `mess`. Вот и получили настоящую неразбериху...

`no multidimensional` — спасает нас в ситуации, когда мы хотим получить срез значений хеша:

```
1 %hash = (
2     "a" => 1,
3     "b" => 2
4 );
5 print @hash{"a","b"};
```

Как и ожидается, это выведет значения ключей `a` и `b`, но если мы сделали опечатку и, вместо символа `@`, ввели `$`, то получим пустой результат, а Perl даже не будет ругаться, так как это вполне допустимая конструкция. Прагма `no multidimensional` прервёт такую программу с ошибкой.

Третья прагма `bareword::filehandles` отучит вас от привычки засорять глобальное пространство имён, давая названия файловым дескрипторам в виде «голых» слов (*bareword*):

```
1 my $content;
2 open(FH, "</some/file") or die $!;
3 while(<FH>) {
4     $content .= $_;
5 }
6 close(FH);
```

Этот код будет выдавать ошибку с данной прагмой. Разумеется, использовать стандартные `STDOUT`, `STDERR` и прочие встроенные «голые» слова не запрещается.

От простого к сложному

Итак, мы пристегнули ремень безопасности и теперь смело можем ехать дальше. Очень трудно сосчитать сколько приёмов отладки придумано на сегодняшний день. Способы могут быть простыми или сложными, универсальными или специфическими. Попробуем рассмотреть доступные варианты, взвесим их плюсы и минусы, область применения и решаемые проблемы.

Оператор print

Самый популярный, на сегодняшний день отладчик кода — это оператор `print`. Расставляя его в различных местах программы, можно проверять значения переменных, видеть в каком направлении исполняется программа. Думаю, что использовать его совершенно не зазорно, т.к. это самый простой и очевидный способ поиска проблемы. Вероятно, он даже сразу подтвердит вашу догадку об ошибке без приложения больших усилий и затрат времени. Ещё лучше использовать его вместе с модулем `Data::Dumper`, который позволяет выводить человеко-читаемый вывод для сложных структур данных:

```
1 use Data::Dumper;
2 print Dumper $some_complex_ref;
```

Плохая сторона использования `print` — этот кусок кода нельзя оставлять в программе. При таком способе отладки в нашем коде могут появляться десятки вызовов `print`, которые потом надо или удалять, или комментировать. А если ошибка снова всплывёт — снова восстанавливать. Кроме того, `print`, вставленный в неудачном месте, может повлиять на логику программы (например, если окажется последним оператором в коде функции).

print if $DEBUG

Использование переменной или константы, которая регулирует включён режим отладки или нет, позволяет не убирать отладочный код из программы.

```
my $DEBUG = 1;
print "debugging message" if $DEBUG;
```

или даже так:

```
use constant DEBUG => 0;
print "debugging message" if DEBUG;
```

Второй вариант даёт небольшой выигрыш в скорости, так как на этапе компиляции Perl будет знать значение константы и просто удалит все куски отладочного кода, если отладка отключена.

Есть только один недостаток у подобного метода — необходимо модифицировать программу, чтобы произвести отладку. Также некоторые неудобства с тем, что нельзя регулировать уровень информативности.

Smart::Comments

Вероятно, после трудоемкого пути с комментированием/раскомментированием строк с отладочным `print` кому-то пришла в голову идея вставлять команды отладки через комментарии. Идея была воплощена в виде модуля `Smart::Comments`. В код вставляются комментарии, состоящие из трёх или более подряд идущих символов #:

```
my $var = 10;
### $var
### Doubled $var: $var*2
```

Такие комментарии заставят программу вывести значения указанных выражений:

```
$ perl –MSmart::Comments application.pl

```

```
### $var: 10
### Doubled $var: 20
```

Для вывода простого текстового сообщения нужно добавить троеточие в конце комментария:

```
### Do some work...
```

Также можно вставлять в такие текстовые сообщения метку времени и номер строки программы:

```
### <now> Do some work at <here>...
```

выведет:

```
### Sat Mar 16 15:03:09 2013 Do some work at "application
    .pl", line 12...
```

Существует возможность задать разный уровень у отладочных сообщений, варьируя количество символов комментария #. Тогда при запуске с такими параметрами:

```
$ perl -MSmart::Comments=###,#### application.pl
```

будут выводиться только комментарии уровня 3 и 4, но не 5 и более.

Есть также и `assert`, который позволяет прервать работу программы, в случае если условие не выполнено, например:

```
my $x = 10
### assert: $x < 10
```

остановит программу и выведет:

```
### $x < 10 was not true at application.pl line 12.
###     $x was: 10
```

Если прерывать программу не требуется, но нужно выдать предупреждение, если условие не выполнилось, можно использовать `check`:

```
my $x = 10
### check: $x < 10
```

С помощью умных комментариев можно даже выводить индикатор прогресса, например, для итераций циклов:

```
for my $var ( @list ) { ### Progressing...   done

Progressing...                 done
Progressing..........          done
Progressing................    done
```

Если кому-то нравится индикатор прогресса с процентами, то это тоже возможно:

```
for my $var ( @list ) { ### Evaluating [===|    ] % done

Evaluating [|             ] 0% done
Evaluating [===|          ] 33% done
Evaluating [======|       ] 55% done
```

Кроме того, в случае цикла `for`, при продолжительном времени работы цикла, начинает выводиться также и время оставшееся до завершения цикла:

```
Evaluating [|              ] 9% done  (about 1 minute
    remaining)
```

`Smart::Comments` позволяют достаточно просто отлаживать программы. Комментарии не оказывают никаких побочных эффектов на программу, поэтому их можно безболезненно оставлять в коде.

Из недостатков можно назвать использование модулем фильтра исходного кода, что может оказывать негативные эффекты на код всего приложения при отладке. Есть также несколько неприятных ошибок, например, модуль может портить содержимое переменной `$@` при выводе сообщений и не может вывести значение выражения в простом примере:

```
### $x*2
```

Devel::Comments

`Devel::Comments` — это форк `Smart::Comments`, который обладает некоторым новым функционалом, например, есть возможность вы-

водить сообщения не на экран, а в файл:

```
use Devel::Comments ({—file => 'log'});
```

Но при этом модуль имеет все те же проблемы, что и `Smart::Comments`.

Devel::Trace

Чтобы проследить как выполняется программа, можно использовать `Devel::Trace`, который отображает каждую строку исходного кода на экран перед её выполнением:

```
$ perl —d:Trace test1.pl
>> test1.pl:2: print "hello,world";
>> test1.pl:3: exit 0;
hello,world
```

Основное достоинство модуля в его простоте. Но никакой другой важной информации вы от него не получите. Существуют и альтернативные модули, которые позволяют фильтровать выводимую информацию в зависимости от модулей и/или функций в которых в текущий момент находится интерпретатор.

Carp::REPL

Модуль `Carp::REPL` также может использоваться при отладке:

```
$ perl —MCarp::REPL=warn my_code.pl
```

В случае, если в коде возникает ошибка или даже предупреждение, происходит останов программы и вы попадаете в интерактивную оболочку `Devel::REPL`, в которой можно попробовать вывести значения переменных, стек вызовов и т.д. Модуль может быть полезен для того, чтобы добраться до проблемной точки в программе и попытаться выяснить ошибку на месте в многофункциональной интерактивной командной оболочке. Модуль требует установленного `Devel::REPL`, который имеет зависимость от `Moose` (для кого-то это может быть недостатком).

Встроенный отладчик Perl

Ранее были рассмотрены некоторые способы отладки, которые требовали модификации исходного кода. Вмешательство в код программы может само стать источником проблем. Часто проблема может быть скрыта где-то глубоко, причиной может стать и ошибка в стороннем модуле. Везде расставить `print` или комментарии не всегда возможно и правильно. В этом случае никак не обойтись без использования отладчика.

Дистрибутив Perl уже включает в себя отладчик. Для того, чтобы запустить отладку программы, достаточно использовать ключ `-d` интерпретатора:

```
$ perl -d -e 1

Loading DB routines from perl5db.pl version 1.33
Editor support available.

Enter h or `h h' for help, or `man perldebug' for more
    help.

main::(-e:1): 1
DB<1>
```

После чего мы попадаем в командную оболочку программы, которая позволяет проводить отладку приложения. Отлаживаемая программа загружается, но останавливается перед самым первым оператором (секции `BEGIN` и `CHECK` при этом выполняются). Важно, чтобы программа компилировалась без ошибок, в противном случае отладка будет невозможна. Если отладчик не может распознать команду, то он попытается выполнить её через `eval` как обычный Perl-код. С этой точки зрения отладчик можно рассматривать как простую *REPL* (*Read Eval Print Loop*) оболочку для Perl.

Следует только учитывать, что область видимости лексических переменных ограничена внутри `eval` и они не будут доступны для использования в дальнейшем.

```
DB<2> my $x=1
DB<3> print $x

DB<4> $x =1
```

```
DB<5> print $x
1
```

Команды отладчика

Вся работа по отладке приложения выполняется с помощью команд. Можно устанавливать точки останова, запускать выполнение программы по одному оператору, выводить значения переменных. Рассмотрим наиболее часто используемые команды, которые помогут выполнить отладку кода.

- `h [command]` — без параметров выведет помощь по всем командам отладчика; если команда указана, то по ней будет выведена короткая документация; указание параметра `h` приведёт к подробному выводу помощи по всем командам и вывод будет достаточно длинным; для чтения длинного вывода удобно использовать пейджер, задаваемый символом вертикальной черты:

```
| h h
```

- `p expr` — выведет значение выражения `expr`, аналогичен оператору `print`;
- `x [maxdepth] expr` — вычисляет значение выражения `expr` в списочном контексте и выводит результат, только, в отличие от `print`, может в читаемом виде показывать сложные вложенные структуры;
- `V [[pkg] [vars]]` — отобразит все или только указанные переменные пакета (по умолчанию `main`); имя переменных указывается без символа `$`:

```
DB<2> V main ]
$] = 5.016003
```

для фильтрации списка удобно использовать регулярное выражение, указав перед шаблоном символ `~` (или `!` — для исключения):

```
DB<2> V main ~\d
$2 = '~\\d'
$1 = 'main'
$0 = '—e'
```

- `X [vars]` — то же самое, что и `V currentpackage [vars]`;
- `y [level [vars]]` — показывает все или выбранные лексические переменные;
- `T` — вывод трассировки;
- `s` — выполнение одного оператора программы со входом в функции;
- `n` — выполнение одного оператора программы без входа в функции;
- `r` — продолжить выполнение до конца текущей функции;
- `c` — продолжить выполнение до следующей точки останова;
- `l` — показать несколько строк исходного кода;
- `t` — переключает режим трассировки;
- `b` — устанавливает точку останова на указанной строке;
- `B` — удаляет точку останова;
- `enable/disable` — включает/выключает точку останова;
- `w expr` — задаёт наблюдение за указанной переменной;
- `W expr` — удаляет наблюдение за переменной;
- `q` — выход из программы;
- `R` — перезапуск программы;
- `m` — список доступных для запуска методов/функций;
- `M` — список загруженных модулей.

Опции отладчика

Опции отладчику можно задавать в переменной окружения `PERLDB_OPTS`, в файле `~/.perldb` или в опциях командной строки. Они регулируют поведение отладчика, вот некоторые из них:

- `AutoTrace` — задаёт режим трассировки;
- `NonStop` — отключает интерактивный режим, происходит запуск программы, пока не будет прервана сигналом или каким-

то другим способом;
- `ReadLine` — задаёт использование библиотеки `ReadLine`; иногда требуется её отключать, если отлаживаемая программа сама использует `ReadLine`;
- `LineInfo` — задаёт файл для записи информации о строках кода `frame` — задаёт уровень информативности вывода при входе/выходе в/из процедур;
- `RemotePort` — `host:port` для удалённой отладки; ввод и вывод перенаправляются на сетевой сокет.

Пример неинтерактивного запуска отладчика с выводом информации о вызове функций в файл `listing`:

```
1 $ PERLDB_OPTS="NonStop LineInfo=listing frame=2" perl -d
    /usr/bin/cpan
```

Дополнительная информация

Подробную информацию об отладчике можно почитать в man-страницах perldebug и perldebtut

Другие интерфейсы к отладчику Perl

Консольный интерфейс может показаться неудобным в использовании и тяжёлым для освоения, поэтому существует несколько проектов, дающих возможность использовать более дружественные для пользователя интерфейсы.

- `Devel::ptkdb` — это графический интерфейс к отладчику с использованием тулкита `Tk`; он достаточно лёгкий, но рекомендовать его можно только, если вам не интересны другие альтернативы; всё-таки используемый тулкит староват и выглядит не очень привычно для современного десктопа;
- `Padre` — имеет в своём составе графический интерфейс к отладчику; доступны большинство функций отладчика, выглядит

вполне современно и удобно в использовании;

- `Devel::hdb` – недавно появившийся довольно перспективный модуль, который позволяет отлаживать ваши скрипты, создавая web-интерфейс к отладчику:

```
$ perl -d:hdb /usr/bin/cpan
Debugger listening on http://127.0.0.1:8080/
```

- `Vim::Debug` — модуль позволяет производить отладку внутри редактора vim.

Новое поколение отладчиков для Perl

Встроенный отладчик Perl имеет долгую историю. Сам по себе он представляет собой скрипт `perl5db.pl` в 10 тыс. строк и несёт в себе тяжёлое наследие продолжительного развития вместе с Perl. Разобраться в его коде не так просто, исправлять проблемы тяжело, а расширять функционал крайне затруднительно. По этой причине появились проекты, поставившие задачу создать новый отладчик для Perl, который можно было бы легко развивать и дополнять функционал расширениями.

Devel::ebug

`Devel::ebug` — это простой и расширяемый отладчик для Perl, имеющий понятное API. Он позиционируется как замена встроенному отладчику, хорошо протестированная основа для создания других отладчиков с консольным, графическим или веб-интерфейсом.

Для `Devel::ebug` на данный момент есть два интерфейса: `ebug` — консольный клиент (в составе самого пакета) и `ebug_http` — веб-интерфейс (в составе `Devel::ebug::HTTP`). Кроме того, в состав пакета входят клиент `ebug-client` и сервер `ebug-server` позволяющий производить удалённую отладку приложения.

Devel::Trepan

`Devel::Trepan` — это довольно серьёзная альтернатива стандартному отладчику Perl. Автор модуля *Rocky Bernstein* имеет большой опыт в области создания отладчиков, он приложил руку к созданию отладчиков для языков программирования Python, Ruby и Shell. Руководствуясь всё теми же мотивами (устаревший и тяжелый в поддержке и развитии код встроенного отладчика), создан новый отладчик, который также выдвигает новое важное требование — совместимость с набором команд `gdb`. Не секрет, что отладчик `gdb` знаком многим программистам независимо от того, на каком языке они программируют — отладка C/C++-программ и библиотек может потребоваться в любой момент. Если отладчик Perl будет иметь сходный набор команд, он будет значительно проще в освоении.

После установки доступен консольный отладчик `trepan.pl`. Запуск отладки происходит привычным способом:

```
$ trepan.pl application.pl
-- main::(application.pl:5)
print "Hello, World!\n";
(trepanpl):
```

Можно обратить внимание на использование удобной цветной подсветки синтаксиса. Кроме того, доступны некоторые полезные опции командной строки:

- `-c` — подключить файл с командами отладчика (по аналогии с ключом `-x` gdb);
- `-x` — трассировка выполняемых команд (по аналогии запуска `set -x` в shell);
- `--client` или `--server` — удалённая отладка кода (клиент или сервер).

Команды trepan.pl

- `help` — выведет подсказку по существующим командам; вывод очень похож на аналогичный вывод gdb; помощь разделе-

на на несколько секций-классов:

- `breakpoints` — остановка программ в нужном месте;
- `data` — изучение данных;
- `files` — изучение файлов;
- `running` — управление запущенной программой;
- `stack` — изучение стека;
- `status` — статусная информация о программе;
- `support` — утилиты отладчика;
- `syntax` — синтакс команд.

Наиболее часто используемые команды:

- `step` — выполнить следующий оператор (со входом в функции);
- `next` — выполнить следующий оператор (без входа в функции);
- `finish` — выполнить все операторы до конца данной функции;
- `continue` — продолжить выполнение до точки остонова;
- `quit` — выйти из программы;
- `list` — показать исходный код;
- `info var (l|m|o)` — показать определённый класс переменных в данной области видимости;
- `backtrace` — информацию о стеке фреймов;
- `break` — установка точки останова;
- `watch` — отслеживание изменения переменной.

На большинство самых востребованных команд присутствует короткие псевдонимы: `s` — step, `n` — next, `l` — list и т.д. Весь список можно увидеть по команде `alias`.

Любопытно, что можно рекурсивно запускать отладку кода внутри отладчика, с помощью команды `debug`. Например, отладка функции Фибоначчи:

```
1 debug fibonacci(5)
```

Расширения Devel::Trepan Для отладчика `Devel::Trepan` уже существуют расширения, улучшающие его возможности:

- `Devel::Trepan::Shell` — интерактивная командная строка Perl внутри `Devel::Trepan` (на основе `Devel::REPL`). Интерактивная командная строка в отладчике присутствует, но всё же имеет ограничения. С данным расширением вы получаете полноценный *REPL*;
- `Devel::Trepan::Disassemble` — поддержка команды `dissassemble` в отладчике. Служит для вывода *дизасемблированного* кода операций пакета или подпрограммы. Для вывода используется модуль `B::Concise`.

Заключение

Несмотря на все старания, обзор получился скорее всего не полный. Не была рассмотрена отладка XS-приложений, регулярных выражений, ничего не было сказано про профайлеры, такие как `Devel::NYTProf`, а также не упомянуты модули `B::Concise` и `B::Deparse`, позволяющие рассмотреть, во что транслируется Perl-код для глубокого погружения в пучины отладки. Но думаю, что этим и другим более продвинутым методам можно посвятить отдельную статью.

■ *Владимир Леттиев*

5. Введение в разработку web-приложений на PSGI/Plack

PSGI/Plack — современный способ написания web-приложений на Perl. Практически каждый фреймворк так или иначе поддерживает или использует эту технологию. В статье представлено краткое введение, которое поможет быстро сориентироваться и двигаться дальше.

Мы живем в такое время, когда технологии и подходы в области web-разработки меняются очень быстро. Сначала был CGI, потом, когда его стало недостаточно, появился FastCGI. FastCGI решал главную проблему CGI. В CGI при каждом обращении было необходимо перезапускать серверную программу, обмен данными происходил при помощи `STDIN` и `STDOUT`. В FastCGI взаимодействие с сервером происходит через TCP/IP или Unix Domain Socket. Теперь у нас есть `PSGI`.

Что это такое?

`PSGI`, как говорит его разработчик Tatsuhiko Miyagawa, это «Перловый суперклей для веб-фреймворков и веб-серверов». Ближайшие родственники — `WSGI` (Python) и `Rack` (Ruby). Идея тут вот в чем. Разработчик очень часто тратит довольно много времени, чтобы адаптировать свое приложение под как можно большее количество движков, а `PSGI` предоставляет единый интерфейс для работы с различными серверами, что сильно упрощает жизнь.

Особенности

Безусловно, формат статьи не позволяет описать полностью все нюансы, поэтому здесь и далее будут только ключевые моменты.

- для обмена информацией между клиентом и сервером используется `$env` (представляет из себя ссылку на хеш);

- PSGI приложение — ссылка на Perl-функцию, которая принимает в качестве параметра $env;
- функция возвращает ссылку на массив, который состоит из 3 элементов: HTTP статус, [HTTP заголовки], [Тело ответа];
- функция может вернуть и ссылку на другую функцию, но это будет рассмотрено в других более углубленных статьях;
- расширение файла, содержащего код запуска приложения, должно быть .psgi.

На данном этапе это все, что нужно для того, чтобы начать разбираться с кодом непосредственно.

PSGI-приложение

Ниже приведен код простейшего PSGI-приложения.

```
1 my $app = sub {
2     my $env = shift;
3
4 # Производим необходимые манипуляции с $env
5     return [200, ['Content-Type' => 'text/plain'], ["
          hello, world\n"]];
6 };
```

Сохраняем это приложение в файле app.psgi, или любом другом с расширением psgi. Смотрим на особенности. Потом на код. Потом опять на особенности. Все сходится. Запускаем.

При запуске perl app.psgi он «молча» отрабатывает, но приложение не запущено.

Основные PSGI-серверы

Для того, чтобы запускать PSGI-приложения нам необходим PSGI-сервер. На данный момент серверов несколько.

- Starman

- Twiggy
- Feersum
- Corona

Кратко о PSGI-серверах

- `Starman` — pre-forking сервер; работает довольно быстро, многое умеет из коробки, поддержку unix domain sockets, например;
- `Twiggy` — асинхронный сервер, базируется на `AnyEvent`;
- `Feersum` — субъективно, самый быстрый из этого всего списка; основная часть реализована в виде `XS`-модулей. Базируется на `EV`;
- `Corona` — асинхронный сервер, базируется на `Coro`.

Все эти сервера доступны на CPAN. В дальнейшем мы будем использовать `Starman`, затем сменим его на `Twiggy`, а затем на `Feersum`. Каждой задаче свой сервер.

Запуск приложения

Приложение абсолютно одинаково запустится на любом из этих серверов, может быть, под `Corona` его придется чуть видоизменить. После установки сервера, а в нашем случае это `Starman`, в `/usr/bin` или `/usr/local/bin` должен появиться исполняемый файл `starman`. Запуск производится следующей командой:

```
/usr/local/bin/starman app.psgi
```

По умолчанию `PSGI`-серверы используют 5000 порт. Мы можем его изменить, запустив приложение с ключом `--port 8080`, например. Напомним, что `PSGI` — спецификация. В данном случае мы использовали эту спецификацию для написания простейшего web-приложения. Очевидно, что для нормальной разработки нам необходимо реализовать и множество вспомогательных функций,

от получения GET-параметров до получения данных cookie. Этого всего не было бы без необходимого функционала.

Plack

`Plack` — это реализация `PSGI` (в Perl есть стандартный модуль `Pack`, потому реализация получила имя `Plack`). `Plack` существенно облегчает нам жизнь, как разработчикам. Он содержит в себе огромное количество функций для работы с `$env`.

В базовой комплектации `Plack` состоит из довольно большого количества модулей. На данном этапе нас интересуют только эти:

- Plack
- Plack::Request
- Plack::Response
- Plack::Builder
- Plack::Middleware

`Plack::Request` и `Plack::Response` возвращают различные значения типа `Hash::MultiValue`, на которые стоит обратить внимание.

Hash::MultiValue

Модуль, автором которого тоже является Tatsuhiko Miyagawa, представляет собой хеш, но с одним нюансом. Он может хранить несколько значений по одному ключу. Например: `$hash->get('key')` вернет value, если же значений по ключу несколько, то оно вернет последнее, а если нужны все значения, то можно воспользоваться функцией `$hash->get_all('key')`, тогда результат будет `('value1','value2')`. `Hash::MultiValue` также учитывает контекст вызова, так что будьте внимательны.

Plack::Request

Модуль, который содержит функции для работы с запросами клиента. Методов содержит много, всегда можно ознакомиться на CPAN. В рамках этой статьи, дальше, мы будем использовать следующие методы:

- `env` — возвращает `$env`;
- `method` — возвращает метод запроса: GET, POST, OPTIONS, HEAD, и т.д.;
- `path_info` — важный метод; возвращает локальный путь к текущему скрипту;
- `parameters` — возвращает параметры (*x-www-form-url-encoded*, параметры адресной строки) в виде `Hash::MultiValue`;
- `uploads` — возвращает параметры (переданные при помощи *multipart-form-data*) тоже в виде `Hash::MultiValue`.

Plack::Response

- `status` — устанавливает статус (код ответа HTTP), будучи вызванным без параметров, возвращает ранее установленный статус;
- `headers` — устанавливает заголовки ответа;
- `finalize` — точка выхода, последняя функция приложения; возвращает `PSGI`-ответ согласно спецификации.

Plack::Builder

Рассматривать методы не будем, отметим только, что это весьма гибкий маршрутизатор. Например, он позволяет устанавливать обработчик (`PSGI`- приложение) на локальный адрес:

```
1 my $app = builder {
2     mount "/" => builder { $my_cool_app; };
3 };
```

Результат — обращения по адресу `/` будут перенаправлены в соответствующее `PSGI`-приложение. В данном случае это `$my_cool_app`.

Маршруты могут быть вложенными, например:

```
my $app = builder {
    mount "/" => builder {
        mount "/another" => builder {
           $my_another_cool_app; };
        mount "/"        => builder { $my_cool_app; };
    };
};
```

И эти маршруты могут быть вложенными. В этом примере, все, что не попадает в `/another` отправляется в `/`.

Plack::Middleware

Базовый класс для создания middleware-приложений. Middleware это «промежуточное программное обеспечение». Используется тогда, когда нужно модифицировать `PSGI`-запрос или готовый `PSGI`-ответ, а также предоставить специфические условия для запуска определенной части приложения.

Перепишем приложение на Plack

```
use strict;
use Plack;
use Plack::Request;

my $app = sub {
    my $env = shift;
    my $req = Plack::Request->new($env);
    my $res = $req->new_response(200);

    $res->body('Hello World!');

    return $res->finalize();
};
```

Это простейшее приложение, использующее Plack. Оно совершенно наглядно демонстрирует принцип его работы.

На что надо обратить внимание. $app — ссылка на функцию. Очень часто, когда идет быстрое написание нечто подобного, забывается символ ; после окончания ссылки на функцию или создание Plack ::Request без передачи $env. Стоит быть внимательным.

Для проверки синтаксиса можно использовать perl −c app.psgi.

Вот еще один важный момент касательно написания PSGI-приложений: при формировании тела ответа стоит убедиться, что там находятся байты, а не символы (например, UTF-8). Обнаруживается такая ошибка весьма сложно. Ее наличие приводит к пустому ответу сервера с ошибкой в psgi.error:

```
"Wide character at syswrite"
```

Запускается наше приложение аналогично предыдущему.

- $req — это объект типа Plack::Request; $req содержит в себе данные запроса клиента; он получает их из хеша $env, который передается в функцию;
- $res — Plack::Response, это ответ клиенту; строится по запросу при помощи метода new_response, в качестве параметра принимает код ответа (200 в нашем случае);
- body — устанавливает тело ответа;
- finalize — преобразование объекта ответа в ссылку на массив PSGI-ответа (который, как было описано выше, состоит из статуса, заголовков и тела ответа).

Да, Hello world это конечно неплохо, но мало функционально. Сейчас, используя весь инструментарий, попробуем написать простейшее приложение (но оно будет гораздо полезнее, правда).

Напишем API, реализующее три функции:

- первая будет принимать строку в качестве входяшего параметра и говорить о том, что строка успешно принята; адрес для

обращения — `localhost:8080/`;
- вторая функция будет принимать строку в качестве параметра и возвращать, например, является ли эта строка палиндромом (слово или фраза, которая одинаково выглядит с обеих сторон, например — «Аргентина манит негра»); располагаться будет по адресу `localhost:8080/palindrome`;
- третья функция будет принимать в качестве параметра ту же строку и возвращать ее перевернутой; располагаться будет по адресу `localhost:8080/reverse`.

В результате написания кода у нас должно получиться нечто, умеющее следующие вещи:

- при обращении на `/` отвечать что все ок, если передан параметр `string`;
- при обращении на `/palindrome` проверять наличие параметра `string`, отвечать, является оно палиндромом или нет;
- при обращении на `/reverse` отдавать перевернутую строку.

Для переворачивания строки будем использовать следующую конструкцию:

```
$string = scalar reverse $string;
```

Для определения, является ли строка палиндромом, будем использовать следующую функцию:

```
sub palindrome {
    my $string = shift;

    $string = lc $string;
    $string =~ s/\s//gs;

    if ($string eq scalar reverse $string) {
        return 1;
    }
    else {
        return 0;
    }
}
```

Приложение

`Plack::Request` позволяет получать параметры при помощи метода `parameters`.

```
my $params = $req->parameters();
```

Доработаем приложение и приведем его к виду:

```
use strict;
use Plack;
use Plack::Request;

my $app = sub {
    my $env    = shift;

    my $req    = Plack::Request->new($env);
    my $res    = $req->new_response(200);
    my $params = $req->parameters();

    my $body;
    if ($params->{string}) {
        $body = 'string exists';
    }
    else {
        $body = 'empty string';
    }

    $res->body($body);

    return $res->finalize();
};
```

Запускаем. Первая часть готова.

Перейдя по адресу `http://localhost:8080/?string=1` мы увидим ответ, который скажет нам о том, что строка есть. Переход же по адресу `http://localhost:8080/` вернет нам ошибку.

Остальную логику можно реализовать прямо в этом же приложении, разделяя логику по `path_info`, которая будет содержать текущий путь. Для справки, разбор `path_info` может быть реализован следующим образом:

```
my @path = split '\/', $req->path_info();
```

```
2 shift @path;
```

И теперь в `$path[0]` находится необходимый нам путь.

Важно: после внесения изменений в код, сервер необходимо перезапускать!

Plack::Builder

А вот теперь стоит повнимательнее посмотреть на маршрутизатор.

Он дает возможность использовать другие `PSGI`-приложения в качестве компонентов. Еще очень полезной будет возможность подключать middleware.

Переделаем первое приложение так, чтобы оно использовало маршрутизатор.

```
1  use strict;
2  use Plack;
3  use Plack::Request;
4  use Plack::Builder;
5
6  my $app = sub {
7      my $env = shift;
8
9      my $req = Plack::Request->new($env);
10     my $res = $req->new_response(200);
11     $res->header('Content-Type' => 'text/html', charset
          => 'Utf-8');
12
13     my $params = $req->parameters();
14     my $body;
15     if ($params->{string}) {
16         $body = 'string exists';
17     }
18     else {
19         $body = 'empty string';
20     }
21
22     $res->body($body);
23
24     return $res->finalize();
```

```
};

my $main_app = builder {
    mount "/" => builder { $app; };
};
```

Теперь `$main_app` это основное `PSGI`-приложение. `$app` присоединяется к нему по адресу `/`. Кроме того, была добавлена функция для установки заголовков в ответ (через метод `header`). Стоит сделать важное замечание: в данном приложении для упрощения все функции помещены в один файл. Для более сложных приложений так делать, конечно, не рекомендуется.

Теперь подключим компонент для переворачивания строки в виде приложения, которое будет находиться по адресу `http://localhost:8080/reverse`.

```
use strict;
use Plack;
use Plack::Request;
use Plack::Builder;

my $app = sub {
    my $env = shift;

    my $req = Plack::Request->new($env);
    my $res = $req->new_response(200);
    $res->header('Content-Type' => 'text/html', charset
       => 'Utf-8');

    my $params = $req->parameters();
    my $body;
    if ($params->{string}) {
        $body = 'string exists';
    }
    else {
        $body = 'empty string';
    }

    $res->body($body);

    return $res->finalize();
};

my $reverse_app = sub {
    my $env    = shift;
```

```
29
30      my $req    = Plack::Request->new($env);
31      my $res    = $req->new_response(200);
32
33      my $params = $req->parameters();
34      my $body;
35      if ($params->{string}) {
36          $body = scalar reverse $params->{string};
37      }
38      else {
39          $body = 'empty string';
40      }
41
42      $res->body($body);
43
44      return $res->finalize();
45  };
46
47  my $main_app = builder {
48      mount "/reverse" => builder { $reverse_app };
49      mount "/"        => builder { $app; };
50  };
```

Адрес для проверки — `http://localhost:8080/reverse?string=test%20string`.

2/3 задачи выполнено. Однако, в данном случае уж очень похожие получились `$app` и `$reverse_app`. Проведем небольшой рефакторинг. Сделаем функцию, которая будет возвращать другую функцию (иначе, функцию высшего порядка).

Теперь приложение выглядит так:

```
1  use strict;
2  use Plack;
3  use Plack::Request;
4  use Plack::Builder;
5
6  sub build_app {
7      my $param = shift;
8
9      return sub {
10         my $env = shift;
11
12         my $req = Plack::Request->new($env);
13         my $res = $req->new_response(200);
```

```
14          $res->header('Content-Type' => 'text/html',
               charset => 'Utf-8');
15
16          my $params = $req->parameters();
17          my $body;
18          if ($params->{string}) {
19              if ($param eq 'reverse') {
20                  $body = scalar reverse $params->{string};
21              }
22              else {
23                  $body = 'string exists';
24              }
25          }
26          else {
27              $body = 'empty string';
28          }
29
30          $res->body($body);
31
32          return $res->finalize();
33      };
34  }
35
36  my $main_app = builder {
37      mount "/reverse" => builder { build_app('reverse') };
38      mount "/"        => builder { build_app() };
39  };
```

Так гораздо лучше. Теперь добавим третью и последнюю функцию в наше API и закончим, наконец, приложение. В результате всех доработок получилось приложение вида:

```
1  use strict;
2  use Plack;
3  use Plack::Request;
4  use Plack::Builder;
5
6  sub build_app {
7      my $param = shift;
8
9      return sub {
10         my $env = shift;
11
12         my $req = Plack::Request->new($env);
13         my $res = $req->new_response(200);
14         $res->header('Content-Type' => 'text/html',
               charset => 'Utf-8');
15
```

```
16          my $params = $req->parameters();
17          my $body;
18          if ($params->{string}) {
19              if ($param eq 'reverse') {
20                  $body = scalar reverse $params->{string};
21              }
22              elsif ($param eq 'palindrome') {
23                  $body =
24                    palindrome($params->{string})
25                    ? 'Palindrome'
26                    : 'Not a palindrome';
27              }
28              else {
29                  $body = 'string exists';
30              }
31          }
32          else {
33              $body = 'empty string';
34          }
35
36          $res->body($body);
37
38          return $res->finalize();
39      };
40  }
41
42  sub palindrome {
43      my $string = shift;
44
45      $string = lc $string;
46      $string =~ s/\s//gs;
47
48      if ($string eq scalar reverse $string) {
49          return 1;
50      }
51      else {
52          return 0;
53      }
54  }
55
56  my $main_app = builder {
57      mount "/reverse"    => builder { build_app('reverse')
            };
58      mount "/palindrome" => builder { build_app('
           palindrome') };
59      mount "/"           => builder { build_app() };
60  };
```

Ссылка для проверки:

```
http://localhost:8080/palindrome?string=argentina%20Manit
%20negra
```

В дальнейших статьях будут рассмотрены более углубленные темы: middleware, сессии, cookie, обзор серверов, с примерами для каждого конкретного + небольшие бенчмарки, особенности и тонкости `PSGI/Plack`, `PSGI` под нагрузкой, обзор способов разворачивания `PSGI`-приложений, `PSGI`-фреймворки, профилирование, Starman + Nginx, запуск CGI-скриптов в `PSGI`-режиме или «У меня CGI приложение, но я хочу `PSGI`» и так далее.

■ *Дмитрий Шаматрин*

6. Обзор CPAN за март 2013 г.

Рубрика с обзором интересных новинок CPAN за прошедший месяц.

Статистика

- Новых дистрибутивов — 280
- Новых выпусков — 915

Новые модули

- App::jt Утилита для форматирования JSON-документов. Кроме форматирования присутствуют функции сжатия, фильтрации, поиска.
- Data::SimplePath Обращение и установка элементов глубоко вложенных структур посредством простого описания. Например: `->set('hash_ref1/hash_ref2/array_ref/0', 'new value')`
- App::GitHubPullRequest Утилита для управления `pull requests` на GitHub. Позволяет просматривать, открывать, закрывать или коментировать.
- DBIx::Iterator Создание итераторов налету при запросах к базе данных, которые возвращают несколько строк. Способ похожий на `fetchrow_hashref`, но позволяющий существенно упростить код.
- Proc::FastSpawn Единый интерфейс для быстрого создания новых процессов, который выполняет `vfork+exec`, `spawn` или `fork+exec` в зависимости от системы на которой выполняется.

Обновлённые модули

- DBIx::AssertIndex 0.02 MySQL-специфичный модуль позволяющий определить какие SQL-запросы не используют индексы.
- Future 0.12 Реализация Future паттерна для событийно-ориентированного программирования.
- HTTP::Tiny 0.028 Легковесный модуль как альтернатива `LWP::UserAgent`. В новом выпуске добавлена поддержка `cookies`.
- Inline 0.52 Модуль позволяющий писать вставки на других языках программирования прямо в коде Perl.
- Plack 1.0018 Реализация PSGI. В новом выпуске существенно ускорено быстродействие метода `query_parameters` в `Plack::Request`.
- App::cpanminus 1.6103 Простой в использовании установщик CPAN-модулей. В новом выпуске исправлена ошибка проявляющаюся на perl 5.8.
- IO::Socket::IP 0.19 Модуль претендующий на замену `IO::Socket::INET`. В новом выпуске исправлены некоторые методы для совместимости с `IO::Socket::INET`.
- AnyEvent::Task 0.720 Асинхронная клиент-серверная реализация для пула задач. В новом выпуске улучшена документация и тесты.
- Pinto 0.067 Система развертывания, контроля и фиксирования версий модулей локального CPAN. В новом выпуске существенно переработаны внутренности библиотеки, включая переход на `HTTP::Tiny`, систему для миграции между версиями и ускорение работы приложения вообще.
- Starman 0.3007 Популярный PSGI-сервер. В новом выпуске приведено в соответствие с HTTP-спецификацией обработка заголовков запроса.

■ *Вячеслав Тихановский*

7. Интервью с Alexis Sukrieh

Alexis Sukrieh — французский Perl-программист. Наиболее известен как автор популярного веб-фреймворка Dancer.

Как и когда начал изучать программирование?

Я начал изучать программирование когда еще был в средней школе, у меня был калькулятор, кажется, Casio, в систему была встроена минимальная реализация языка BASIC, и я начал экспериментировать. Помню, что моей первой программой была текстовая игровая лотерея. Вот так вот я и познакомился с `if–then–else` операторами и `while`-циклами.

Какой редактор используешь?

Vim! Не могу использовать ничего другого для написания кода. Каждый раз когда я пробую другой редактор, возвращаюсь к vim через 10 минут. Думаю, что мой мозг, буквально, сросся с vim.

Как и когда познакомился с Perl?

Это было во время моей первой работы (почти 13 лет назад). У меня было задание написать небольшую статистическую утилиту, которая бы доставала из MySQL большое количество данных и строило бы удобные PDF-отчеты через LaTeX. Меня познакомили с Perl, дав книги издательства O'Reilly «Введение в Perl» и, конечно же, «Camel Book». Вот так все и было. Поигравшись с этим несколько дней, я полюбил этот язык. Настолько просто было учить, настолько интуитивно, я никогда не прекращал писать на Perl с того времени.

С какими другими языками программирования приятно работать?

Некоторое время я программировал на Ruby, когда работал в Yoolink (компания, которую я соосновал и где был техническим директором). Наше веб-приложение было написано на Ruby on Rails. Это было очень модно в то время (2008 г.) и мы хотели поэкспериментировать с этой новой штуковиной. Мне очень понравилось работать

с Rails тогда, но я никогда не терял удовольствия при работе с Perl.

Наверное, я никогда не мог найти той же свободы, которая есть в Perl. Я, как будто, знаю, что могу написать практически что угодно на Perl, даже расширить его синтаксис с помощью… DSL (*Domain Specific Language – Предметно-ориентированный язык* – прим. перев.). Понятно к чему я веду, правда?

Какое, по-твоему, самое большое преимущество Perl?

Хм, извини, но кажется, я рассказал про это в предыдущем ответе! Да, для меня самым большим плюсом Perl есть свобода, которую он дает. Можно расширять его покуда есть желание. У этого есть и свои недостатки, потому что свобода достается по цене очень… «толерантного» синтаксиса, а многие программисты хотят строгости (которую дает Python, например). И это можно понять, мне кажется, что обе философии имею право на жизнь, и очень хорошо, что у нас есть эти два стиля. Perl-путь наиболее удовлетворяет моим потребностям.

И, конечно, я не могу закончить здесь, если мы говорим о достоинствах Perl… без упоминания CPAN, разумеется. Даже самый плохой язык на земле стал бы замечательным, будь у него CPAN. Поэтому люди и приходят в Perl, мне кажется. Потому что любую возникшую проблему можно решить используя несколько хороших CPAN-модулей.

И если мы говорим про CPAN, то мы говорим и о Perl-сообществе. Множество умнейших людей ежедневно привносят что-то в язык, появляются очень умные идеи (Moose, к примеру, как проект наиболее впечатляющий и изменяющий мир Perl), а также мощные утилиты для поддержки, сопровождения и тестирования наших дистрибутивов. Автоматическое тестирование, MetaCPAN, cpanminus — все это делает работу с Perl удобной.

Какая, по-твоему, характеристика наиболее важна для языков будущего?

Вот это да, каверзный вопрос! Дай-ка подумать… Если бы я мог щелкнуть пальцами и волшебным образом получить универсаль-

ный язык будущего, каким бы он был? Хм, я думаю у него был бы синтаксис сильно похожий на натуральный язык. Синтаксис, который бы позволял программисту «описать» (в отличие от запрограммировать) данные с которыми он работает и поведение этих данных (если так можно выразиться).

Возможно, мы бы могли написать спецификацию нашей программы на обычном английском (конечно, придерживаясь некоторых соглашений) и затем направить это описание нашей магической программе. Все это бы компилировало документ в программу и тест. Мы могли бы затем проверить тест, и двигаться дальше. Думаю, что-то вроде этого.

Мне кажется, что когда мы говорим о будущем компьютерных технологий, мы говорим о мире, где машины все лучше и лучше начинают понимать нас, людей. Так что языки будущего будут несомненно похожи на то, что я описал.

Что мотивирует тебя на такую активность в Perl-сообществе и open source в целом?

Для начала, если быть честным, я не настолько активен в Perl и open source сообществе как был когда-то. Во время моей наибольшей «активности», я разрабатывал и поддерживал Backup Manager, писал для Debian (в течение трех лет был разработчиком Debian) и, конечно же, выкладывал модули на CPAN. К сожалению, моя личная и профессиональная жизнь больше не позволяют мне быть таким же активным. Поэтому я сфокусировал свое время для «свободного ПО» на Dancer.

Что меня мотивирует? Очень хороший вопрос. Думаю, что удовольствие писать код с единственной целью… писать код. Свободное ПО это единственная область где можно найти такое: когда замешаны деньги, красота кода меняется в силу внешних приоритетов. А здесь время не имеет значения. Все что имеет значение это то, что получается на выходе. С интеллектуальной точки зрения это очень удовлетворяет и ободряет.

Написание, выкладывание и сопровождение свободного ПО позволяет улучшать навыки, которые далеки от чистого программирова-

ния. Приходится рекламировать продукт, предоставлять поддержку пользователям, документировать код, исправлять ошибки… это действительно переводит на другой уровень как программиста, это уж точно, и это очень ценно. Это расширяет ваши горизонты как разработчика, вы начинаете видеть всю картину в целом.

К тому же, в конце концов, у вас могут взять дружественное интервью, и это может быть настоящей причиной почему вы занимаетесь свободным ПО! Нарциссизм!

Dancer — это, несомнено, один из самых популярных веб-фреймворков на Perl. Когда и почему начал этот проект? Почему, по-твоему, он стал настолько популярным и привлек такое большое количество разработчиков?

Очень сложно ответить на этот вопрос. Я могу объяснить свое видение, что послужило причиной такой популярности, но здесь много неизвестного.

Во-первых, Dancer заполнил нишу. В то лето 2009 г. не было ничего похожего на Sinatra на CPAN: готовый микро-фреймворк, который предоставляет богатый DSL для написания веб-приложений. Dancer представил новый способ для решения проблемы «веб-разработки» на Perl.

Во-вторых, конечно же, его дух: быть интуитивным насколько возможно. Поэтому у вас нет `$self` в контроллерах, это преднамеренное решение. Все, что может быть удалено из синтаксиса, должно быть удалено. Как можно меньше лишнего. Это то, что в конечном итоге приносит чувство простоты, легкости и развлечения при разработке веб-приложений. Большинство пользователей говорят, что любят Dancer за то, что он не принуждает делать что-то, чего требует фреймворк, он не мешает им. И все это из-за небольшой и интуитивной прослойки в качестве DSL.

И, напоследок, думаю, тот факт, что я всегда пытался поощрять усилия сообщества в разработке, таким образом привлекая все больше и больше программистов.

Очень скоро я выдал права доступа к репозиторию другим разра-

ботчикам, это быстро привело к идее «ядра разработчиков», очень помогло в техническом обслуживании и создало много энергии вокруг проекта.

Посмотрите на список изменений Dancer, и посчитайте имена… Это, наверное, одно из самых приятных чувств: видеть такое большое количество людей присоединяющихся к тому, что ты когда-то начал. Так что изначальная идея была не такая уж и плохая!

Почему, по-твоему, параллельные выпуски и поддержка Dancer и Dancer2 это хорошая идея?

Потому что это фактически два разных способа реализации концепции Dancer, и изменения в дизайне настолько существенны, что невозможно полностью сохранить обратную совместимость с уже существующей экосистемой Dancer1. Я объяснил свою позицию подробнее в своем блоге. В заключении той записи обозначены причины такого решения:

> Есть множество реальных приложений на D1. Большинство из них работают на движках, которые не работают с D2. К тому же, инкапсуляция некоторых приложений может быть нарушена, если его разбить на мелкие пакеты.
>
> По этой причине я выпущу Dancer 2 под новым именем, и это привнесет новые силы в оба проекта:
>
> Dancer 1 в то же время размораживается, продолжается разработка (с тем условием, что изменения не приведут к новому Dancer 2!);
>
> Dancer 2 выпускается на CPAN для развития собственной экосистемы;
>
> Пользователи Dancer 1 счастливы;

Пользователи Dancer 2 счастливы;

Миграция приложений становится контроллируемой.

В свое время было много попыток очернить проект. Что помогло тебе справиться с этим? Почему, по-твоему, такое происходит в open source мире и как с этим бороться?

Ха-ха, эти времена давно позади. Но действительно, в первые дни или месяцы Dancer был окружен страстями и войнами! Вначале было это недоразумение кто же был первым Dancer или Mojolicious::Lite (*не стоит путать с Mojolicious* — прим. перев.). Так как оба проекта были выпущены примерно в одно и то же время это создавало путаницу. Эта же путаница привела к созданию «сторон» и представило Dancer как конкурента Mojolicious (или наоборот), что фактически неправильно: Mojolicious отличный проект со множеством интересных идей. Это полноценное решение для веб-разработки с другой философией (что хорошо). Dancer это другая область, у него другое видение: он то же для веб-разработки, что Perl для программирования: минимальный и расширяемый набор ключевых слов, позволяющих описать решение проблемы. Вначале никто из нас (по обе стороны) не понимал, что мы играем на разных полях (включая меня, должен сознаться). Поэтому мы вели себя по-детски и вдоволь кормили троллей.

Был даже один индивидуум, который тратил время на плохие отзывы в CPAN рейтинге или флудил в наших объявлениях на Hacker News, даже пытался распространять слухи, что сообщество фреймворка Sinatra ненавидит Dancer! Что привело к официальному сообщению в блоге Sinatra: Sinatra любит Dancer!

В конечном счете ты понимаешь, что чем больше ты пытаешься навредить проекту, тем больше ты даешь ему сил, все возвращается как позитивная энергия. Даже плохой пиар все равно пиар.

Когда все это происходило я находил силы для продолжения своей работы по простой причине: позитивные отзывы от наших пользователей, взгляните на нашу страницу с отзывами, она побуждает действовать. Я думал: если я сделал что-то не так с Dancer, почему

столько разработчиков интересуются им? И некоторые из них известные и заслуженные Perl-хакеры. Я должен был что-то сделать хорошо, не может же быть все насколько плохо!

Почему такое происходит в open source мире? Потому что когда мы общаемся online наше общение… бесчувственно. Нет чувств в email, нет возможности понять иронию или что ты кого-то обидел или рассмешил. Поэтому расстраиваться из-за email это плохая идея. Можно по-разному интерпретировать слова. Поэтому, мне кажется, хакеры из-за разногласий так часто воюют online!

Да и в конце концов, ты понимаешь, что это все о строчках кода! Это не имеет большого значения!

Где сейчас работаешь? Использует ли твоя компания Perl? Важно ли, по-твоему, поддерживать язык Perl и его сообщество на бизнес уровне?

Я работаю в Weborama, одной из лидирующих в Европе (у нас, кстати, с прошлого лета есть представительство и в России) компаний в индустрии онлайн рекламы. Я заведую R&D (*научно-исследовательские и опытно-конструкторские работы* — прим. перев.) отделом и работаю с очень талантливыми людьми, чрезвычайно рад работать в этой команде. Мы очень старались сделать нашу платформу используя Perl в полную силу. И не только язык, но и «культуру»: мы используем `CPAN::Mini` для создания приватных зеркал для наших внутренних модулей (в Weborama окружении). Все, что мы пишем, это CPAN дистрибутивы, написанные на современном Perl (Moose/Moo/Dancer…), а разворачиваем все с помощью `perlbrew`. Мы повторили настоящий CPAN, чтобы пользоваться всей инфраструктурой: юнит-тестами, автоматическими тестами, `cpanminus` как системой для развертывания… Мы используем `Perl::Critic` подключенный в `git hooks`, для проверки, что все придерживаются нашего стиля кодирования. Думаю, понятно, что все здесь базируется на Perl. Так работать доставляет большое удовольствие, и помогает нам предоставлять программное обеспечение высокого качества для компании.

Также мы стараемся привлекать как можно больше талантливых Perl-хакеров, недавно мы добавили вакансии на LinkedIn и

jobs.perl.org и получили много ответов от людей со всего мира, желающих переехать в Париж и присоединиться к нашей команде. Работать в таком окружении захватывающе! Да, я не объективен! Все-таки я отвечаю за эту команду и ничего не могу сказать о ней плохого! Но я думаю, что если спросить моих коллег, они со мной согласяться (я надеюсь!).

Думаю, очень важно поддерживать язык Perl на бизнес уровне, как мы делаем в Weborama. Мы уже некоторое время спонсируем Perl-мероприятия, франзцузские Perl-воркшопы, YAPC::EU и другие. Мы также помогаем нашим сотрудникам посещать такие события, спонсируя билеты и отели или разрешаем им не брать отпускные дни.

Содействие Perl-сообществу очень ценно для компании, которая работает с этим языком, это здравый смысл.

Стоит ли сейчас советовать молодым программистам учить Perl?

Думаю, да. Perl — это язык, который легко выучить, у него большой потенциал благодаря своей экосистеме и сообществу. Работа с Perl может быть очень эффективной, поэтому чем больше у нас молодых разработчиков — тем лучше. Стоит направлять любые усилия для привлечения новых людей в сообщество.

Вопросы от читателей

Используешь Mojolicious?

Конечно! Все мои веб-приложения написаны на `Mojolicious`. Не могу и думать о других способах.

...

Помните, я говорил раньше о том, что тяжело понять иронию из email? Если серьезно, я не пользовался `Mojolicious`, но читал код, когда начал разрабатывать Dancer 2. Я хотел посмотреть как сдела-

ны некоторые вещи, и тогда я понял, что Dancer это совсем не то же самое, что Mojolicious::Lite. Lite это небольшая DSL-обертка над большим фреймворком. Dancer это полноценный DSL. Это не совсем одно и то же. Опять же, я думаю, что оба подхода имеют право на существование.

Так что нет, я не использую Mojolicious, но считаю проект хорошим.

Думаешь, что умеешь танцевать?

(*Dancer с англ. танцор* — прим. перев.)

Думаю, что моя лунная походка совсем неплоха! И нет, я не пришлю видео, придется поверить мне на слово.

Известны ли тебе большие реально работающие проекты, написанные на Dancer?

Во-первых, у нас в Weborama работает рекламный сервер (Управление кампаниями), предоставляющий API для веб-интерфейса и клиентов. Этот веб-сервис написан на Dancer (первой версии) и отлично работает. Недавно мы выпустили новый продукт на Dancer 2. Вне Weborama есть множество «реальных» компаний, которые используют Dancer. Например, Moonfruit.com с гордостью работает на Dancer. Также я слышал, что у Novell в недавно выпущенном приложении для развертывания серверов Baracus используется встроенный REST-сервис, который написан на Dancer. Другие проекты можно найти у нас на сайте perldancer.org/dancefloor.

Приедешь ли на YAPC::EU в Киев?

Пока не знаю, все зависит от моего расписания, но кто знает!

Оставил ли ты Dancer 1 сообществу или все еще участвуешь в развитии?

Dancer 1 сейчас поддерживает Yanick Champoux, но это не значит, что я не участвую в дискуссиях. Я недалеко, даже если все силы направляю на Dancer2.

Спасибо за интервью, используйте Perl и танцуйте!

■ *Вячеслав Тихановский*

8. Perl Quiz

Perl Quiz — уже ставшая традиционной на многих Perl-конференциях викторина на «знание» Perl. Почему в кавычках? Это вы поймете из самих вопросов. Ответы на викторину в текущем выпуске будут опубликованы в следующем. Итак, поехали!

Ответы из предыдущего выпуска: 1) RUZ (Руслан Закиров, 39 модулей), 2) Florian Ragwitz на YAPC::Brazil 2012, 3) Новосибирск, 4) 4 (ActiveState Perl, Strawberry Perl, DWIM Perl, Citrus Perl), 5) Скипетр, 6) tuit, от фразы "When I get around *to it*", 7) Это название лука-порея, которое перекликается с изначальным названием языка Perl (pearl onion), 8) Плохие программы пишет программист, хорошие — язык, 9) Германия, 10) 2.

1. Какой любимый цвет Ларри Уолла?

    1. Циан
    2. Лазурный
    3. Бордо
    4. Хаки
    5. Шартрёз

2. Кто первым обозначил различия между Perl, perl и PERL?

    1. Ларри Уолл
    2. Билл Гейтс
    3. Рэндел Шварц
    4. Тим О'Райли
    5. Дэмиан Конвей

3. Кто не любит, чтобы его выступления на Perl-конференциях записывались на видео?

    1. Рэндел Шварц
    2. Абигэйл
    3. Ларри Уолл
    4. Барби

5. Одри Танг

4. За какие заслуги Perl-программисты получают награду White Camel?

    1. Участие в разработке Perl
    2. Выдающийся модуль на CPAN
    3. Не связанное с программированием участие в Perl-сообществе
    4. Авторство популярной статьи о Perl
    5. Организацию бизнеса построенного на Perl-инфраструктуре

5. В каком году была организована comp.lang.perl?

    1. 1987
    2. 1988
    3. 1989
    4. 1990
    5. 1991

6. Как называется популярный немецкий Perl-журнал?

    1. $foo
    2. $bar
    3. undef
    4. Perl
    5. dePerl

7. Официальным логотипом Perl 6 является:

    1. Верблюд
    2. Медведь
    3. Динозавр
    4. Насекомое
    5. Помесь всего вышеперечисленного

8. Какая из этих платформ уже не поддерживается в Perl 5.14?

    1. Windows ME
    2. Windows XP
    3. NetBSD

4. MirOS BSD
5. AIX

9. В Perl 5.10 элементы регулярных выражений какого языка поддерживаются наравне со встроенными?

    1. Ruby
    2. JavaScript
    3. Python
    4. Lua
    5. Ada

10. Где была проведена первая Perl-конференция?

    1. Нью-Йорк
    2. Москва
    3. Сан-Хосе
    4. Сан-Франциско
    5. Чикаго

■ *Вячеслав Тихановский*