# Pragmatic Perl 3

pragmaticperl.com

Выпуск 3. Май 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: Виктор Турский, Jeffrey Thalhammer (thaljef), Владимир Леттиев (crux), Дмитрий Шаматрин

Выпускающий редактор: Вячеслав Тихановский (vti)

Ревизия: 2014-12-04 14:54

# Оглавление

# 1. От редактора

Новый выпуск журнала мы встречаем вместе с пятьюстами подписчиками!

В этом номере к нам присоединился Jeffrey Thalhammer, автор известного и весьма полезного модуля по статистическому анализу Perl-кода `Perl::Critic`. В своей статье он описывает новое приложение `Pinto`, позволяющее с легкостью создавать собственный CPAN-репозиторий. Кроме того, Jeff начал кампанию по сбору средств на развитие проекта. Если `Pinto` покажется вам интересным, подумайте над пожертвованием!

Также в журнале появился раздел с вакансиями. Надеемся, что он будет полезным.

Конференция YAPC::Europe 2013 в Киеве уже не за горами. Недавно на сайте конференции зарегистрировался Ларри Уолл. Присоединяйтесь и вы!

Мы продолжаем искать авторов для следующих номеров. Если у вас есть идеи или желание помочь, пожалуйста, свяжитесь с нами.

Приятного чтения.

■ *Вячеслав Тихановский*

# 2. Три правила тестирования кода, написанного с использованием ORM-фреймворка

*Рассмотрены частые проблемы при тестировании ORM-классов, приведены несколько способов их решения.*

Тестирование является важной частью разработки ПО, но на практике многие разработчики часто не пишут тестов. Почему так? Казалось бы, сегодня можно найти достаточное количество документации на эту тему, в которой на пальцах объясняется, как написать тест (например, Test::Tutorial). В реальности же код, который нужно тестировать, намного сложнее обучающих примеров, особенно это касается кода, который работает с базой данной. Еще более усугубляется ситуация, когда мы пытаемся написать тесты к коду, написанному с использованием ORM-фреймворка. Не будем пытаться категоризировать тесты и делить их на разные виды, а попытаемся предложить решение лишь некоторых проблем, с которыми вам придется столкнуться при написании тестов. Примеры будут для `Rose::DB::Object`.

## Плохие новости

*Код, написанный с использованием ORM-фреймворка, — это код, который зависит от базы данных*

Этот факт создает дополнительные трудности во время тестирования. Естественно, зависимость от базы данных логична, поскольку ORM-фреймворк и предназначен как раз для того, чтобы отобразить объектную модель на реляционную базу данных. Особенно это касается шаблона проектирования Active Record (`Rose::DB::Object` и `DBIx::Class` используют данный шаблон), который делает нашу модель и хранилище еще более зависимыми и связанными.

*ORM-фреймворк — это черный ящик*

ORM фреймворк — это черный ящик и мы не знаем, что происходит внутри. Мы используем его API, загружаем объекты, вызываем разные методы и для нас происходит некая магия, фреймворк за нас формирует запросы к базе, загружает данные, создает объекты, отслеживает отношения между ними. С одной стороны это, конечно, положительный момент, но с другой стороны — негативный. Мы одновременно и зависим от базы данных, и еще больше пытаемся от нее абстрагироваться. Это заставляется нас отказаться от Mock-объектов.

А имеются ли хорошие новости? Хорошие новости имеются и их тоже две:

## Хорошие новости

*Мы работаем с объектами и классами*

То есть, мы можем абстрагироваться от базы данных и просто считать, что мы имеем некие персистентные объекты и тестировать их как обычно.

*Для тестирования многих объектов необязательна персистентность*

Необязательно для тестирования классов сохранять что-то в базу данных. Часто бывает достаточно создать экземпляр объекта в памяти.

Держа в уме все эти новости, в нашей компании были выработаны три правила, которые позволили облегчить процесс тестирования ORM-зависимого кода. Предлагаю рассмотреть их:

### Правило №1 — никаких тестов для простых классов

Не пишите тесты для примитивных классов, в которых нет никакой логики, а только схема данных. Вот пример такого класса:

```
1 package Language;
```

```
2
3 use base qw(Rose::DB::Object);
4
5 __PACKAGE__->meta->setup(
6     table   => languages,
7     columns => [
8         language_id => { type => varchar, length => 3,
              not_null => 1 },
9         name        => { type => varchar, length => 64,
              not_null => 1 },
10    ],
11    primary_key_columns => [ language_id ],
12 );
```

Если мы посмотрим на этот код, то увидим, что он не содержит никакой логики, а только метаданные для отображения класса на реляционную базу данных. Естественно, все зависит от проекта и требований, но мы на своей практике избегаем тестирования такого кода по нескольким причинами:

1. Тесты — это код, который необходимо написать, он может содержать ошибки, и его нужно постоянно поддерживать.
2. Заниматься тестированием таких классов — это, по сути, заниматься тестированием самого ORM-фреймворка, у которого и так имеются свои тесты.

Соглашусь, что в таких классах тоже могут быть ошибки, поскольку человеку свойственно ошибаться. Хорошим решением будет уменьшить участие человека в написании такого кода. Достаточно создать генератор кода, написать к нему тесты, и автоматически сгенерировать простые классы. Для `Rose::DB::Object` существует `Rose::DB::Object::Loader`, для `DBIx::Class` — `DBIx::Class::Schema::Loader`.

### Правило №2 — передавайте зависимости снаружи

Если вы знакомы с термином внедрение зависимостей (Dependency Injection), то для вас это будет очевидным. Давайте рассмотрим

несколько примеров. Примеры реальные, но значительно упрощенные.

*Пример №1 — «передача данных снаружи»*

Часто возможно избежать работы с базой данной за счет передачи всех необходимых для тестирования данных снаружи. Допустим, у нас есть класс для расчета налогов, налоги и их ставки хранятся в базе данных. Схема класса выглядит так:

```
1  package TAX;
2
3  use base qw(Rose::DB::Object);
4
5  __PACKAGE__->meta->setup(
6      table   => taxes,
7      columns => [
8          tax_id => { type => varchar, length   => 16,
               not_null => 1 },
9          rate   => { type => integer, not_null => 1 }
10     ],
11     primary_key_columns => [ tax_id ],
12 );
13
14 sub calculate_tax_amount  { ... }
```

Есть два атрибута — код налога и ставка налога. И есть метод для расчета суммы налога — `calculate_tax_amount`.

В реальной жизни мы пишем так:

```
1 my $tax        = Tax->new( tax_id => 'VAT_20' )->load();
2 my $tax_amount = $tax->calculate_tax_amount( 102.51 );
```

То есть:

1. Создаем объект.
2. Загружаем его из базы данных по его идентификатору.
3. Вызываем метод `calculate_tax_amount` и рассчитываем сумму налога.

В тестах же мы можем передавать ставку налога снаружи в конструктор и затем вызывать пересчет.

```
my $tax_obj = Tax->new( rate => 20 );
is( $tax_obj->calculate_tax_amount( 123.01 ), 20.50 );
is( $tax_obj->calculate_tax_amount( 456.00 ), 76.00 );
```

Этот пример наглядно показывает как протестировать простой объект, который не зависит от других объектов.

*Пример №2 — внедрение зависимостей*

Это более сложный пример и он требует от нас того, чтобы мы во время написания кода не забывали, что нам придется его тестировать. Создаем некий финансовый документ с двумя записями и хотим посчитать налог на весь документ.

В реальной жизни пишем так:

```
my $doc = FinancialDoc->new(
    lines => [
        {amount => 102.22},
        {amount => 103.27}
    ],
    date => "2010-10-10"
);

$doc->build();
my $amount = $doc->tax_amount();
```

`$doc->build` — загружает налог, который был актуален в 2010 году, подсчитывает сумму налога и сохраняет его в атрибуте документа `tax_amount`. Пример немного искусственный и предназначен исключительно для демонстрации подхода.

В тестах же нам необходимо избежать обращения к базе данных, мы можем это сделать за счет внедрения зависимостей в объект `FinancialDoc` через конструктор. Добавим дополнительный неперсистентный атрибут `tax_object` в наш класс (многие ORM позволяют такое сделать) и передадим в конструктор объект налога.

```
my $doc = FinancialDoc->new(
    lines => [
        {amount => 102.22},
        {amount => 103.27}
    ],
    date => "2010-10-10",
```

```
    tax_object => Tax->new( rate => 20 )
);
$doc->build();

is( $doc->tax_amount(), 41.01 );
```

Как мы видим, внедрение зависимостей часто позволяет облегчить тестирование и подменить объект загружаемый из базы другим объектом, но, к сожалению, это не всегда так просто сделать. Встречаются ситуации, когда мы имеем сложные операции с данными и имеем сложные спрятанные зависимости и нам приходится использовать базу данных в тестах.

Усложняют ситуацию два дополнительных факта:

1. Мы не можем использовать DBI Mock, поскольку мы не должны зависеть от реализации фреймворка.
2. Мы имеем одну базу данных для всей модели.

Что делать в такой ситуации? Первое решение, которое приходит на ум — это использовать заданный набор тестовых данных, который хранится в тестовой базе данных. Например, это могут быть страны, пользователи, компании, материалы, клиенты, контрагенты, склады, курсы валют, запасы... и множество других.

Общий набор тестовых данных будет работать, но это будет ровно до тех пор, пока мы не начнем тестировать объекты, которые агрегируют данные или работают со списками. Например, отчеты, которые будут зависеть не просто он наличия необходимых ему данных, а от наличия всех данных. К примеру, отчет по количеству материала на складе. Для добавления нового теста для такого отчета потребуется расширять набор данных, но при расширении у нас сломаются тесты для существующих отчетов.

Таким образом мы приходим к третьем правилу:

### Правило №3 — индивидуальное тестовое окружение для каждого набора тестов

Звучит вполне логично и на практике работает. Остается вопрос, как можно такого добиться? Есть несколько вариантов решения:

1. Пересоздавать базу данных для каждого набора тестов. Такой подход будет гарантированно работать, но на пересоздание базы данных требуется слишком много времени. Кроме того, скорее всего буду данные, которые никогда не меняются и практически не влияют на тесты.
2. Использовать встраиваемую базу данных, например, SQLite. Таким образом можно просто копировать файл эталонный базы перед каждым набором тестов. Будет работать достаточно быстро, но тут возникает другая проблема — необходимость поддерживать два описания базы данных. Кроме того, нужно учитывать, что это все таки разные базы данных и результаты в тестах и реальной жизни могу отличаться.
3. Вручную удалять все данные после каждого теста. Такой подход часто встречается, но приходится писать дополнительные процедуры очистки данных и поддерживать их в актуальном состоянии.

В нашей компании мы пришли к следующему решению — по завершению тестов мы просто откатываем транзакцию:

```
1 use Test::More;
2
3 my $db = Rose::DB->new_or_cached();
4 $db->begin_work();
5
6 init_test_data();
7
8 do_testing();
9
10 $db->rollback();
```

В данном примере происходит следующее:

1. Мы стартуем новую транзакцию.

2. Инициализируем индивидуальное тестовое окружение. Наполняем базу необходимыми для данного конкретного тестового набора данными.
3. Запускаем тесты, которые могу делать любые изменения в базе данных.
4. Запускаем откат транзакции и база данных возвращается в исходное состояние.

Единственный момент, про который стоит помнить — это, чтобы база данных поддерживала транзакции. В MySQL (InnoDB) такой подход отлично работает. В результате мы получаем для каждого теста свой собственный набор данных, не прилагая особых усилий для очистки базы после каждого теста.

Какой можно сделать вывод из всего вышесказанного? Иногда простые правила, позволяют серьезно облегчить работу программиста. Еще раз три правила:

1. Не пишите тесты для простых классов.
2. Используйте внедрение зависимостей.
3. Используйте индивидуальное тестовое окружение для каждого набора тестов.

В завершение статьи приведу пример кода, который позволит упростить старт и откат транзакции. Пример очень близкий к реальному:

```
1  use Test::More;
2  use Test::Project;
3
4  my $t = Test::Project->new();
5
6  $t->object_factory->buy_material_from_partner({price =>
       1200});
7  $t->object_factory->buy_service_from_partner({price =>
       1200});
8
9  $t->iterate_test_data(
10     "report_a",
11     sub {
12         my ($report_params, $expected_output) = @_;
```

```
13          my $report = Project::Report::A->new(%
               $report_params);
14
15          $t->test_report($report, $expected_output);
16      }
17  );
```

Когда у нас создается объект класса `Test::Project`, то автоматически стартует транзакция. Когда объект уничтожается, то транзакция откатывается.

И, соответственно, код класса `Test::Project`:

```
1  package Test::Project;
2
3  use Moo;
4
5  has 'object_factory' => (
6      is      => 'lazy',
7      default => sub {
8          return Test::ProjectObjectFactory->new();
9      }
10 );
11
12 has 'db' => (
13     is      => 'lazy',
14     default => sub {
15
16         # Switch database
17         Project::DB->default_domain('test');
18         return Project::DB->new_or_cached();
19     }
20 );
21
22 sub BUILD {
23     my $self = shift;
24
25     die "ALREADY IN TRANSACTION" if $self->db->
          in_transaction();
26     $self->db->begin_work();
27
28     return $self;
29 }
30
31 sub DEMOLISH {
32     my $self = shift;
33     $self->db->rollback();
```

```
34 }
35
36 1;
```

■ *Виктор Турский, технический директор компании WebbyLab*

# 3. Pinto — собственный CPAN из коробки

*Pinto — многообещающий инструмент для построения собственного CPAN-репозитория. На данный момент автор собирает средства (собрано уже более 80%) на реализацию новых интересных задач. Поучаствуйте и вы! http://tinyurl.com/gopinto*

Одной из лучших вещей в мире Perl являются открытые модули, которые доступны на CPAN. Но следить за ними сложно. Каждую неделю появляются сотни новых релизов, и никогда не знаешь когда новая версия модуля привнесет ошибку, которая сломает твое приложение.

Одна из стратегий решения этой проблемы — сделать свой CPAN-репозиторий, который содержит только те версии модулей, которые нужны. Затем можно использовать CPAN-инструментарий для сборки своего приложения из модулей своего репозитория, неподверженного вихрю изменений публичного CPAN.

За прошедшие годы я построил несколько CPAN-репозиториев, используя `CPAN::Mini` и `CPAN::Site`. Но они всегда казались мне неуклюжими и никогда не удовлетворяли моим потребностям. Пару лет назад один клиент нанял меня для построения очередного CPAN. Но в этот раз у меня была возможность начать с нуля. `Pinto` — результат этой работы.

`Pinto` — это надежный инструмент для создания и управления своим CPAN-репозиторием. У него несколько мощных свойств, которые помогут безопасно управлять всеми нужными для приложения Perl-модулями. Эта статья расскажет как создать свой CPAN с помощью `Pinto` и продемонстрирует некоторые особенности.

## Установка Pinto

`Pinto` доступен из CPAN и может быть уставлен как любой другой модуль, используя cpan или cpanm утилиты. Но `Pinto` это больше приложение, чем библиотека. Это инструмент, которые использу-

ется для управления кодом приложения, не будучи частью самого приложения. Поэтому я рекомендую устанавливать Pinto как самодостаточную программу следующими двумя командами:

```
curl -L http://getpinto.stratopan.com | bash
source ~/opt/local/pinto/etc/bashrc
```

Это установит Pinto в ~/opt/local/pinto, а также добавит необходимые директории в PATH и MANPATH. Все автономно, поэтому установка Pinto не меняет окружение для разработки, а также окружение никак не может повлиять на Pinto.

## Изучаем Pinto

Первое, что нужно знать при работе с новым инструментом, это как получить помощь:

```
pinto commands             # Show a list of available commands
pinto help <COMMAND>       # Show a summary of options and arguments for <COMMAND>
pinto manual <COMMAND>     # Show the complete manual for <COMMAND>
```

Pinto также поставляется с другой документацией, включая урок и короткий справочник команд. Можно всегда посмотреть эту документацию одной из следующих команд:

```
man Pinto::Manual::Introduction  # Explains basic Pinto conecpts
man Pinto::Manual::Installing    # Suggestions for installing Pinto
man Pinto::Manual::Tutorial      # A narrative guide to Pinto
man Pinto::Manual::QuickStart    # A summary of common commands
```

## Создание репозитория

Первым шагом в использовании `Pinto` является создание репозитория с использованием команды `init`:

```
1 pinto -r ~/repo init
```

Что создаст новый репозиторий в директории `~/repo`. Если директория не существует, она будет создана. Если она уже существует, то должна быть пустой.

Опция `-r` (или `--root`) указывает где находится репозиторий. Она обязательна для каждой команды `pinto`. Если это достаточно утомительно, можно установить переменную окружения `PINTO_REPOSITORY_ROOT` и опустить опцию `-r`.

## Инспектирование репозитория

Теперь, когда репозиторий создан, посмотрим что в нем находится. Для того, чтобы узнать что в репозитории, воспользуемся командой `list`:

```
1 pinto -r ~/repo list
```

На данном этапе команда ничего не распечатает, потому что ничего нет в репозитории. Но в этой статье команда `list` будет использоваться довольно часто.

## Добавление CPAN-модулей

Предположим идет работа над приложением `My-App`, которое содержит модуль `My::App` и он зависит от модуля `URI`. Добавим модуль `URI` в репозиторий используя команду `pull`:

```
1 pinto -r ~/repo pull URI
```

Будет предложено ввести сообщение для журнала, описывающего для чего вносится данное изменение. Вверху сообщения будет сге-

нерированное сообщение, которое можно редактировать. Внизу сообщения находятся добавленные модули. После внесения нужных изменений, редактор необходимо закрыть.

Теперь в `Pinto`-репозитории находится модуль `URI`. Посмотрим, что же действительно получилось. Воспользуемся командой `list` для просмотра содержимого репозитория:

```
pinto -r ~/repo list
```

На это раз, список будет выглядеть похожим образом:

```
rf  URI              1.60  GAAS/URI-1.60.tar.gz
rf  URI::Escape      3.31  GAAS/URI-1.60.tar.gz
rf  URI::Heuristic   4.20  GAAS/URI-1.60.tar.gz
...
```

Видно, что модуль `URI` был добавлен в репозиторий, а также все его зависимости, зависимости зависимостей и так далее.

## Добавление приватных модулей

Предположим работа над `My-App` завершена, осталось выпустить первый релиз. Используя предпочитаемую систему сборки (например, `ExtUtils::MakeMaker`, `Module::Build`, `Module::Install` и т.п.) собирается релиз `My-App-1.0.tar.gz`. Его можно добавить в репозиторий с помощью команды `add`:

```
$> pinto -r ~/repo add path/to/My-App-1.0.tar.gz
```

На этот раз снова будет предложено ввести сообщение, описывающее изменение. Сейчас, при просмотре содержимого репозитория, будет показан модуль `My::App`, а в качестве автора будет указан пользователь, добавивший модуль.

```
rl  My::App           1.0  JEFF/My-App-1.0.tar.gz
rf  URI              1.60  GAAS/URI-1.60.tar.gz
rf  URI::Escape      3.31  GAAS/URI-1.60.tar.gz
rf  URI::Heuristic   4.20  GAAS/URI-1.60.tar.gz
...
```

## Установка модулей

Теперь когда в Pinto-репозитории есть модули, следующим шагом является сборка и установка. Под капотом Pinto-репозиторий организован практически как и CPAN-репозиторий, поэтому он полностью совместим с cpanm или любым другим инсталлятором Perl-модулей. Все, что нужно сделать, это указать инсталлятору на Pinto -репозиторий:

```
1 cpanm --mirror file://$HOME/repo --mirror-only My::App
```

Это команда соберет и установит My::App, используя *только* модули из Pinto-репозитория. Поэтому на выходе будут те же версии модулей, даже если они будут удалены или обновлены на публичном CPAN.

С cpanm опция --mirror-only крайне важна, потому что это предотвращает cpanm от обращения к публичному CPAN в случае, когда он не может найти модуль в репозитории. Когда такое происходит, обычно это говорит о том, что в каком-то дистрибутиве неправильно указаны зависимости в файле META. Чтобы починить, достаточно воспользоваться командой pull и добавить все нужные модули.

## Обновление модулей

Предположим прошло несколько недель после первого релиза My -App и теперь у модуля URI появилась на CPAN новая версия 1.62. В ней появились критические исправления, которые тоже хотелось бы получить. Опять же, для обновления воспользуемся командой pull. Но в силу того, что в репозитории уже существует версия URI следует точно указать, что необходимо установить *новую*, указав минимальную версию:

```
1 pinto -r ~/repo pull URI~1.62
```

При просмотре содержимого репозитория в этот раз, видна новая версия URI (и, возможно, также других модулей):

```
rl  My::App           1.0  JEFF/My-App-1.0.tar.gz
rf  URI              1.62  GAAS/URI-1.62.tar.gz
rf  URI::Escape      3.38  GAAS/URI-1.62.tar.gz
rf  URI::Heuristic   4.20  GAAS/URI-1.62.tar.gz
...
```

Если новая версия `URI` требует каких-нибудь обновлений или дополнительных зависимостей, они тоже попадут в репозиторий. И при установке `My::App` будет использована версия 1.62.

## Работа со стеками

До сих пор над репозиторием выполнялись команды как над единственным ресурсом. Поэтому обновление `URI` в предыдущем разделе затронет каждого человека и каждое приложение, которые используют этот репозиторий. Такое широкое воздействие крайне нежелательно. Предпочтительно попробовать изменения изолированно и протестировать их перед тем, как заставлять всех обновляться. Для этого и существуют стеки.

Все CPAN-репозитории имеют индекс, который связывает последнюю версию каждого модуля с архивом, в котором он содержится. Обычно есть только один индекс в репозитории. Но в `Pinto` -репозитории может быть несколько индексов. Каждый такой индекс называется *стек*. Это позволяет создавать разные стеки зависимостей в одном репозитории. Поэтому можно иметь стек для разработки и стек для боевой машины, или стек `perl-5.8` и стек `perl-5.16`. Добавление или обновление модуля всегда затрагивает только один стек.

Перед тем как идти дальше, стоит узнать про стек по умолчанию. Для большинства команд имя стека это опциональный параметр. Поэтому если стек не указан, используется указанный по умолчанию.

В репозитории никогда нет больше одного стека по умолчанию. Когда был создан репозиторий, был создан стек `master` и он был назначен стеком по умолчанию. Можно изменить стек по умолчанию

или изменить его название, но пока не будем в это вдаваться. Стоит запомнить, что `master` — это название стека по умолчанию при создании репозитория.

### Создание стека

Предположим, в репозитории находится версия 1.60 модуля `URI`, но недавно вышла версия 1.62, как и в предыдущих примерах. Необходимо обновиться, но в этот раз воспользуемся отдельным стеком.

До сих пор все, что добавлялось в репозиторий, добавлялось в стек `master`. Поэтому создадим копию этого стека, используя команду `copy`:

```
pinto -r ~/repo copy master uri_upgrade
```

Это создаст новый стек с названием `uri_upgrade`. Чтобы посмотреть содержимое этого стека, воспользуемся командой `list` с опцией `--stack`:

```
pinto -r ~/repo list --stack uri_upgrade
```

Будет выдан список идентичный стеку `master`:

```
rl  My::App           1.0  JEFF/My-App-1.0.tar.gz
rf  URI              1.60  GAAS/URI-1.60.tar.gz
...
```

### Обновление стека

Теперь когда есть отдельный стек, попробуем обновить `URI`. Как и раньше, воспользуемся командой `pull`. Но в этот раз скажем `Pinto` добавить модули в стек `uri_upgrade`:

```
pinto -r ~/repo pull --stack uri_upgrade URI~1.62
```

Можно сравнить стеки `master` и `uri_upgrade`, используя команду `diff`:

```
pinto -r ~/repo diff master uri_upgrade

+rf URI              1.62 GAAS/URI-1.62.tar.gz
+rf URI::Escape      3.31 GAAS/URI-1.62.tar.gz
+rf URI::Heuristic   4.20 GAAS/URI-1.62.tar.gz
...
-rf URI              1.60 GAAS/URI-1.60.tar.gz
-rf URI::Escape      3.31 GAAS/URI-1.60.tar.gz
-rf URI::Heuristic   4.20 GAAS/URI-1.60.tar.gz
```

Вывод похож на вывод команды `diff(1)`. Записи, начинающиеся с `+`, были добавлены, а начинающиеся с `-`, были удалены. Можно увидеть, что модули из дистрибутива `URI-1.60` были заменены модулями из дистрибутива `URI-1.62`.

#### Установка из стека

Как только новые модули появились в стеке `uri_upgrade`, можно попробовать собрать приложение, указав `cpanm` на этот стек. У каждого стека есть поддиректория внутри репозитория, поэтому достаточно просто добавить название к адресу:

```
cpanm --mirror file://$HOME/repo/stacks/uri_upgrade --
    mirror-only My::App
```

Если все тесты проходят, можно уверенно обновить `URI` до версии 1.62 также в стеке `master`, воспользовавшись командой `pull`. Так как `master` это стек по умолчанию, его можно не указывать в качестве параметра:

```
pinto -r ~/repo pull URI~1.62
```

## Замораживание версий

Стеки это отличный способ для тестирования эффекта изменения зависимостей на приложении. Но что, если тесты не проходят? Если проблема в `My-App`, то можно ее быстро исправить, изменив код, выпустив версию 2.0 и затем обновить `URI` на стеке `master`.

Но что, если ошибка в модуле URI или исправление в My-App займет много времени, тут возникает проблема. Не хочется, чтобы кто-то обновил URI, также не хочется, чтобы модуль обновился у других зависимостей My-App. Пока неизвестно, что проблема решена, необходимо предотвратить обновление URI. Для этого есть замораживание версий.

### Замораживание модуля

Когда модуль замораживается, версия этого модуля фиксируется в стеке. Любая попытка обновить его (напрямую или через другую зависимость) будет неудачной. Для замораживания модуля используется команда pin (от англ. закрепить — прим. перев.):

```
pinto -r ~/repo pin URI
```

Если снова просмотреть стек master, будет примерно такой вывод:

```
...
rl  My::App           1.0  JEFF/My-App-1.0.tar.gz
rf! URI              1.60  GAAS/URI-1.60.tar.gz
rf! URI::Escape      3.31  GAAS/URI-1.60.tar.gz
...
```

Символ ! в начале записи означает, что модуль заморожен. Если кто-нибудь попытается обновить URI или добавить дистрибутив, который требует новую версию URI, Pinto выдаст предупреждение и откажется принимать новые дистрибутивы. Стоит отметить, что заморожены все модули внутри дистрибутива URI-1.60, поэтому невозможно частично обновить дистрибутив (такая ситуация возможна, когда модуль перемещается в другой дистрибутив).

### Размораживание модуля

Через некоторое время, допустим, решается проблема в My-App или выходит новая версия URI с исправлением ошибки. Когда такое происходит, можно разморозить URI в стеке, воспользовавшись командой unpin:

```
pinto -r ~/repo unpin URI
```

С этого времени можно без проблем обновить `URI` до последней версии при необходимости. Также как и с заморозкой, разморозка освобождает все модули внутри дистрибутива.

## Использование замораживания и стеков одновременно

Замораживание и стеки часто используются сообща для упрощения цикла разработки. Например, можно создать стек под названием `prod`, который содержит все известные надежные зависимости. В то же самое время можно создать стек `dev`, который содержит экспериментальные зависимости для следующего релиза. Изначально, стек `dev` это просто копия стека `prod`.

В ходе разработки возможно обновление или добавление нескольких модулей в стек `dev`. Если обновленный модуль ломает приложение, этот модуль замораживается в стеке `prod`, сигнализируя о том, что он не должен быть обновлен.

### Замораживание и патчи

Иногда, когда у модуля в CPAN-дистрибутиве есть ошибка, автор не может или не хочет ее исправлять (во всяком случае до релиза вашего приложения). В этой ситуации можно сделать локальный патч CPAN-дистрибутива.

Предположим создается копия модуля `URI` с локальной версий дистрибутива с названием `URI-1.60_PATCHED.tar.gz`. Его можно добавить с репозиторий командой `add`:

```
pinto -r ~/repo add path/to/URI-1.60_PATCHED.tar.gz
```

В этой ситуации не лишним будет и заморозить модуль, так как его обновление нежелательно до тех пор, пока версия на CPAN не содержит нужный патч, или автор не исправляет ошибки каким либо

другим образом.

```
1 pinto -r ~/repo pin URI
```

Когда автор URI выпускает версию 1.62, стоит протестировать ее перед тем, как размораживать локальную исправленную версию. Как и прежде, это может быть достигнуто клонированием стека с помощью команды copy. На этот раз назовем стек trial.

```
1 pinto -r ~/repo copy master trial
```

Но перед обновлением URI в стеке trial необходимо там его разморозить:

```
1 pinto -r ~/repo unpin --stack trial URI
```

Теперь можно попробовать обновить URI в стеке и собрать My::App следующим образом:

```
1 pinto -r ~/repo pull --stack trial URI~1.62
2 cpanm --mirror file://$HOME/repo/stacks/trial --mirror-
    only My::App
```

Если все прошло успешно, размораживаем в стеке master и обновляем версию URI.

```
1 pinto -r ~/repo unpin URI
2 pinto -r ~/repo pull URI~1.62
```

## Обзор изменений

Как, наверное, уже можно было заметить, каждая команда, которая изменяет состояние стека, требует сообщение для описания действия. Эти сообщения могут быть просмотрены командой log:

```
1 pinto -r ~/repo log
```

На выходе должно получиться примерно следующее:

```
1 revision 4a62d7ce-245c-45d4-89f8-987080a90112
2 Date: Mar 15, 2013 1:58:05 PM
3 User: jeff
4
```

```
5       Pin GAAS/URI-1.59.tar.gz
6
7       Pinning URI because it is not causes our foo.t
           script to fail
8
9  revision 4a62d7ce-245c-45d4-89f8-987080a90112
10 Date: Mar 15, 2013 1:58:05 PM
11 User: jeff
12
13      Pull GAAS/URI-1.59.tar.gz
14
15      URI is required for HTTP support in our application
16
17 ...
```

Заголовок каждого сообщения показывает кто и когда сделал изменение. Также у сообщений есть уникальный идентификатор похожий на SHA-1 слепок в `git`. Можно использовать эти идентификаторы для просмотра изменений между разными версиями или же для сброса состояния стека до предыдущей версии [NB: правда, это еще не до конца реализовано].

## Завершение

В этой статей были приведены основные команды для создания `Pinto`-репозитория и наполнения его модулями. А также как использовать стеки и замораживание зависимостей при различных препятствиях во время разработки.

У каждой команды есть несколько опций, который не были рассмотрены, и есть команды, которые вообще не были упомянуты. Поэтому я призываю вас изучить документацию к каждой команде и узнать больше.

■ *Jeffrey Thalhammer* (перев. Вячеслав Тихановский)

# 4. Введение в Perl XS

*Вероятно, многим Perl-программистам никогда не приходилось применять язык XS в разработке. Поэтому, если для вас акроним XS ассоциируется больше с размером одежды, то это нормально и, кстати, удачно описывает узость ниши его применения, но, ни в коем случае, не объём предоставляемых возможностей. Знание о том, какие возможности открывает создание расширений для Perl с использованием XS и какие требования налагает его использование в программах, безусловно будет полезно для всех.*

## Что такое XS?

**XS** — акроним от *eXternal Subroutine* (*внешняя подпрограмма*), представляет собой макроязык, предназначенный для стыковки кода функций, написанных на языке C (или C++) для использования в Perl-программах. Макроязык XS описывает интерфейс функций и служит для согласования модели вызова Perl-функций с моделью вызова C-функций, что включает в себя преобразование типов и манипуляции с размещением аргументов функций и возвращаемых значений. Каждую отдельно описанную функцию в интерфейсе принято называть *XSUB*.

XS используется в тех случаях, когда требуется сделать обвязки (*bindings*) или интерфейс к существующим C-библиотекам для использования в Perl. Например, модуль `Gtk3` — это интерфейс к C-библиотеке `libgtk3`.

XS может использоваться для написания части функций критичных к скорости выполнения или объёму потребления памяти, реализация которых на языке Perl может быть значительно медленнее или требовать больше ресурсов, чем написанная на языке C. Примером могут служить различные вычислительные задачи с большим объёмом вычислений и количеством операций с памятью, как например, `Math::FFT` — модуль с реализацией алгоритмов для выполнения быстрого преобразования Фурье.

XS может потребоваться в системном программировании для низкоуровневого взаимодействия с системой. Например, модуль `Socket::Netlink` — это интерфейс для работы с сокетом семейства `PF_NETLINK` в Linux.

Часто под XS также понимают вообще весь код модуля написанный не на Perl (XS-часть модуля) или в целом аппаратно-зависимые модули, которые требуют для сборки наличие компилятора языка C/C++ (XS-модули). Хотя это и вполне допустимо, но следует знать, что написать модуль для языка Perl на языке программирования C/C++ можно не только с помощью XS, но и, например, с помощью проекта Swig или вообще без использования XS, а только используя Perl API.

Как было сказано, XS — это макроязык и представляет собой набор макросов. Существует компилятор для языка XS, называемый `xsubpp`, который раскрывает код макросов в конструкции C-кода, использующих Perl API. Компилятор использует карту типов (*typemaps*) для преобразования типов аргументов и возвращаемых значений функций к типам используемым в Perl. Таким образом, на выходе компилятора XS мы получаем обычный C-код, который затем компилируется C-компилятором и линкуется в бинарный модуль.

Задача языка XS — упростить написание модулей, заменяя типовой код обвязки короткими макросами. Но сколько бы XS не упрощал жизнь разработчика, он не отменяет необходимости изучения внутреннего строения Perl и API Perl, без чего попытаться объяснить, как писать XS-расширения для Perl невозможно.

## Краткий экскурс во внутренний мир Perl

Сам Perl написан на языке C, поэтому работа с API Perl достаточна естественна при программировании расширений с использованием языка C. В первую очередь, необходимо узнать какие основные типы данных определены в Perl, поскольку многие API-функции оперируют со специфическими типами данных, присутствующих только в Perl.

`SV` — это название типа данных, описывающий скаляр в Perl. Он представляет собой структуру, состоящую из заголовка (с полями: флаг обозначающий тип, количество ссылок на скаляр, ссылка на тело) и объединения, которое собственно и содержит данные, соответствующие типу скаляра. Из структуры видно, что для переменных ведётся подсчёт ссылок и когда он становится равным нулю, это означает, что переменная будет уничтожена (структура освобождена).

Все прочие типы данных `AV` — массив, `HV` — хеш, `CV` — код, `GV` — глоб, по сути являются той же структурой `SV`, выбирается только соответствующий тип в объединении. Такая организация данных позволяет свободно конвертировать один тип в другой. Кроме того, в отличии от простых типов C, это позволяет переменным в Perl самим себя описывать — какой тип данных содержится в них.

Непосредственно значением скаляра могут являться типы: `IV` — `long`, `UV` — `unsigned long`, `NV` — `double`, `PV` — `char *` и некоторые другие типы. Здесь важно лишь то, что нам никогда не потребуется использовать внутреннее представление скаляра, для получения соответствующих полей в его структуре, поскольку для этих целей в API предусмотрены соответствующие макросы/функции.

## Perl API

Документация perlapi содержит список и описание всех публичных (доступных для использования в расширениях) функций. Perl API не является каким-то фиксированным набором функций, как и сам язык Perl, его C API претерпевает изменение: появляются новые функции, исчезают устаревшие. С каждым новым мажорным релизом Perl происходит закрепление нового API и в минорных выпусках стабильных версий API не изменяется. Именно по этой причине при установке новой мажорной версии Perl всегда требуется пересборка всех аппаратно-зависимых модулей. Как правило, поддерживается обратная совместимость, поэтому кроме пересборки никаких других манипуляций может не потребоваться. Но для надёжности существуют механизмы, о которых будет сказано чуть позже, которые позволяют без особых усилий поддерживать воз-

можность сборки и работы ваших расширений на старых версиях Perl.

Перечислять все функции не имеет смысла, но важно обратить внимание на то, какие принципы используются при именовании функций, которые позволяют понять назначение функции и те типы данных, над которыми функция оперирует и какие возвращает.

Например, создание новых переменных:

- `SV* newSV(const STRLEN len)` — создаёт новый скаляр и резервирует под него `len+1` байт (дополнительный 1 байт зарезервирован под нуль-символ);
- `AV* newAV()` — создание нового массива;
- `HV* newHV()` — создание нового хеша.

Как видно, функции создания имеют общий вид: суффикс `new` + возвращаемый тип данных (в верхнем регистре). Тип данных, который получит создаваемый скаляр может конкретизироваться суффиксом (в нижнем регистре), например функции:

```
1 SV* newSViv(IV);
2 SV* newSVuv(UV);
3 SV* newSVnv(double);
```

Как видно, создавая скаляр, можно сразу уточнить, какой тип данных там будет содержаться изначально: `iv` — целое число, `uv` — неотрицательное целое, `nv` — число с плавающей запятой и т.п.

Для доступа к значению (с выполнением преобразования), содержащимся в переменной, используются функции вида:

```
1 IV    SvIV(SV* sv)
2 UV    SvUV(SV* sv)
3 NV    SvNV(SV* sv)
4 char* SvPV(SV* sv, STRLEN len)
5 char* SvPV_nolen(SV* sv)
```

Первая часть названия указывает на тип входных данных (`Sv`, `Av`...) далее может быть суффикс описывающий тип возвращаемых дан-

ных (IV, NV...) и завершать название функции может имя, поясняющая смысл функции. Ещё примеры функций, которые подходят под данное правило:

- `int AvFILL(AV* av)` — получить длину массива;
- `HV* CvSTASH(CV* cv)` — получить стеш (хеш-таблица символов переменных пакета).

Также присутствует набор макросов, название которых пишется полностью в нижнем регистре:

- `void av_clear(AV *av)` — очистить массив;
- `AV* get_av(const char *name, I32 flags)` — получить массив по имени глобальной переменной;
- `CV* get_cvn_flags(const char* name, STRLEN len, I32 flags)` — получить заданную Perl-подпрограмму.

Принцип именования также интуитивно понятен: `av`, `cv` и подобное — указывают на тип данных, а дополнительные суффиксы и префиксы раскрывают суть выполняемой операции.

Также можно обратить внимание, что в названиях разных функций в названии типа можно увидеть дополняющие символ, например, cv**n**, pv**f**, pv**s**. В этом случае, под `n` — понимается длина (т.е. в функции есть параметр, задающий длину `STRLEN`), `f` — использование в функции параметра формата (`sprintf`), `s` — статичной строки (`const char*`). Например:

- `void sv_catpv(SV *const sv, const char* ptr)` — копирует одну строку в конец строки в `sv`;
- `void sv_catpvn(SV *dsv, const char *sstr, STRLEN len)` — копирует указанное число байтов из одной строки в конец другой;
- `void sv_catpvs(SV* sv, const char* s)` — тоже самое, что и `sv_catpvn` только копирует литеральную строку (нультерминированную);

- `void sv_catpvf(SV *const sv, const char *const pat, ...)` — обрабатывает аргумента также как `sprintf()`, добавляя отформатированный вывод в `sv`.

Глобальные переменные Perl API начинаются с префикса `PL_`, например: `PL_sv_undef` — это `undef`, `PL_na` — длина строки, которая используется по умолчанию в операциях без указания длины строки.

## Создание XS-модуля

В дополнение к привычной структуре файлов в составе модуля создаётся файл с названием модуля и с расширением `xs`, который помещается в корневом каталоге проекта. Простейший `xs`-файл выглядит следующим образом:

```
#include "EXTERN.h"
#include "perl.h"
#include "XSUB.h"

MODULE = MyModule    PACKAGE = MyModule
```

По структуре файл состоит из двух частей. Первая часть до строки, начинающейся с `MODULE =`, рассматривается как C-код. Вторая часть, начиная с `MODULE =`, рассматривается как, собственно, `xs`-код. Парсер `xsubpp` также способен распознать POD-документацию, как в 1-ой, так и 2-ой части файла и при компиляции удаляет её, что позволяет безопасно снабжать модуль документацией, не опасаясь ошибок компиляции.

Первые три строки — это стандартные подключаемые заголовочные файлы Perl, минимально необходимые для создания расширения. Далее могут располагаться функции, написанные на языке C, реализующие функционал вашего модуля. Директива `MODULE` во второй части файла задаёт имя файла, в котором содержится Perl-код модуля, из которого будет происходить начальная загрузка (*bootstrap*) двоичного кода, а директива `PACKAGE` определяет пространство имён, в котором будут определены последующие функции. В теле `xs`-директивы `MODULE` и `PACKAGE` могут встречаться несколько

раз, соответственно меняя имя модуля и пространство имён для следующих после них функций.

Данный пример — пустой, в нём не содержится никаких функций. В соответствии с традицией, создадим `hello,world!` с использованием XS. Для удобства, воспользуемся утилитой `h2xs`, которая позволяет автоматически создавать первоначальные шаблоны всех необходимых файлов.

```
$ h2xs -b 5.8.9 -A -n Hello::World --use-xsloader --skip-
    ppport
Writing Hello-World/lib/Hello/World.pm
Writing Hello-World/World.xs
Writing Hello-World/Makefile.PL
Writing Hello-World/README
Writing Hello-World/t/Hello-World.t
Writing Hello-World/Changes
Writing Hello-World/MANIFEST
```

Пояснение по опциям:

- `-b` задаёт минимальную версию Perl, на которой планируется работа модуля;
- `-A` — в модуле не требуется использование AUTOLOAD;
- `--use-xsloader` — вместо `DynaLoader` используем упрощённый `XSLoader`;
- `--skip-ppport` — пока нам не требуется совместимость со старыми версиями Perl API.

Заглянем в `lib/Hello/World.pm`:

```
package Hello::World;

use 5.008009;
use strict;
use warnings;

require Exporter;

our @ISA = qw(Exporter);

our %EXPORT_TAGS = ( 'all' => [ qw(
) ] );
```

```
13
14 our @EXPORT_OK = ( @{ $EXPORT_TAGS{'all'} } );
15
16 our @EXPORT = qw(
17 );
18
19 our $VERSION = '0.01';
20
21 require XSLoader;
22 XSLoader::load('Hello::World', $VERSION);
23
24 1;
```

Так выглядит модуль `Hello::World`. Условно тут можно выделить две секции: секция `Exporter`, в которой задаются какие функции будет экспортировать модуль, и секция `XSLoader`. При добавлении названий функций в массив `@EXPORT` будет обеспечиваться экспорт функций в адресное пространство программы, которая будет загружать данный модуль. При добавлении в массив `@EXPORT_OK` функции не будут экспортироваться автоматически, но их можно будет экспортировать, указав их в параметре к загрузке модуля. Во второй секции, состоящей всего из двух строк и происходит вся магия загрузки XS-составляющей модуля. Далее в этом модуле можно создавать методы написанные на Perl, которые могут совместно работать с методами и функциями определёнными в XS.

Рассмотрим `Makefile.PL`:

```
1 use 5.008009;
2 use ExtUtils::MakeMaker;
3 WriteMakefile(
4     NAME              => 'Hello::World',
5     VERSION_FROM      => 'lib/Hello/World.pm',
6     PREREQ_PM         => {},
7     ($] >= 5.005 ?     ## Add these new keywords
        supported since 5.005
8       (ABSTRACT_FROM  => 'lib/Hello/World.pm', # retrieve
           abstract from module
9        AUTHOR         => 'Vladimir Lettiev <crux@cpan.org
           >') : ()),
10    LIBS              => [''], # e.g., '-lm'
11    DEFINE            => '', # e.g., '-DHAVE_SOMETHING'
12    # Insert -I. if you add *.h files later:
13    INC               => '', # e.g., '-I/usr/include/
        other'
```

```
14      # Un-comment this if you add C files to link with
            later:
15      # OBJECT              => '$(O_FILES)', # link all the C
            files too
16  );
```

Никаких специфических описаний для XS-файла не требуется, если он расположен в стандартном месте для поиска и имеет расширение `xs`. Параметр `INC`, позволяет задавать каталоги с заголовочными файлами, `LIBS` может позволить нам указать, какие библиотеки требуется подключать при линковке, `OBJECT` позволяет задать какие дополнительно требуется собрать C-файлы в проекте, с которыми будет линковаться код модуля.

В случае использования системы сборки `Module::Install`, `Makefile.PL` выглядит ещё проще:

```
1  use inc::Module::Install;
2
3  perl_version '5.008009';
4  name         'Hello-World';
5  all_from     'lib/Hello/World.pm';
6
7  WriteAll;
```

Для задания опций сборки XS-модуля в данной системе сборки, может быть удобен модуль `Module::Install::XSUtil`.

Рассмотрим файл `World.xs`. Он содержит минимальное содержимое, дополним его функцией печати сообщения.

```
1  #include "EXTERN.h"
2  #include "perl.h"
3  #include "XSUB.h"
4
5  MODULE = Hello::World       PACKAGE = Hello::World
6
7  void
8  greeting()
9      CODE:
10         printf("Hello World!\n");
```

Мы описываем функцию `greeting()`, которая печатает сообщение `Hello World!` на экран. Описание XSUB производится в следующем

порядке: первая строка содержит тип возвращаемый функцией — void. Вторая строка соответствует прототипу функции (имя и, возможно, аргументы). В последующих строках описываются директивы XS. В данном случае директива CODE (для лучшей читаемости делается отступ) — указывает, что далее идёт C-код функции.

Для сборки выполним команды:

```
$ perl Makefile.PL
...
$ make
cp lib/Hello/World.pm blib/lib/Hello/World.pm
/usr/bin/perl5.16.3 "-Iinc" /usr/share/perl5/ExtUtils/xsubpp  -typemap
/usr/share/perl5/ExtUtils/typemap  World.xs > World.xsc && mv World.xsc
World.c
Please specify prototyping behavior for World.xs (see perlxs manual)
gcc -c   -D_REENTRANT -D_GNU_SOURCE -fno-strict-aliasing -pipe
-fstack-protector -I/usr/local/include -D_LARGEFILE_SOURCE
-D_FILE_OFFSET_BITS=64 -pipe -Wall -g -O2   -DVERSION=\"0.01\"
-DXS_VERSION=\"0.01\" -fPIC "-I/usr/lib64/perl5/CORE" World.c
Running Mkbootstrap for Hello::World ()
chmod 644 World.bs
rm -f blib/arch/auto/Hello/World/World.so
gcc  -shared -pipe -Wall -g -O2 -L/usr/local/lib -fstack-protector World.o
-o blib/arch/auto/Hello/World/World.so  \
    \
-L/usr/lib64/perl5/CORE -lperl -lpthread
chmod 755 blib/arch/auto/Hello/World/World.so
cp World.bs blib/arch/auto/Hello/World/World.bs
chmod 644 blib/arch/auto/Hello/World/World.bs
```

Рассмотрим вывод make. Первая строка копирует Perl модуль в каталог blib. Затем запускается xsubpp, который компилирует World.xs в файл World.xsc и затем переименовывает его в World.c. Строку с предупреждением о прототипах можно проигнорировать, поскольку xsubpp по умолчанию ожидает, что мы опишем прототип функции (в терминах Perl), но это необязательно. Далее идёт компиляция полученного C-файла с помощью gcc, а затем и линковка. По умол-

чанию используются те опции, с которыми собирался и линковался сам perl (посмотреть можно по команде в `perl -V`). Собранная разделяемая библиотека `World.so` и пустой бутстрап-файл `World.bs` помещается каталог `blib/arch/auto/Hello/World`. Бустрап-файл по сути — это Perl-программа, которая выполняется перед загрузкой разделяемого модуля. В обычных ситуациях он не нужен и, как правило, пустой, но на определённых платформах может использоваться для корректной загрузки разделяемой библиотеки.

Проверим как работает модуль:

```
$ perl -Iblib/lib -Iblib/arch -MHello::World -e 'Hello::
    World::greeting()'
Hello World!
```

Параметр `-I` указывает, где perl будет искать модуль, `-M` загружает модуль, `-e` выполняет код, в данном случае вызов `greeting()`. Модуль работает как и ожидается.

Описанный выше код имеет один недостаток: весь C-код функции находится непосредственно в описании интерфейса функции. Взглянем на альтернативный вариант записи кода:

```
#include "EXTERN.h"
#include "perl.h"
#include "XSUB.h"

void greeting() {
    printf("Hello World!\n");
}

MODULE = Hello::World       PACKAGE = Hello::World

void
greeting()
```

Здесь функция `greeting()` описывается в 1-ой секции XS-файла, как обычная C-функция. А во-второй секции файла описание интерфейса радикально сократилось! Указывается только возвращаемый тип данных, прототип функции, а секция `CODE` опущена. `xsubpp` автоматически привяжет к функции модуля одноимённую C-функцию, определённую выше.

Этот пример даёт понять, что, насколько это возможно, необходимо разделять код функций и код-интерфейса для удобства сопровождения такого кода в дальнейшем. Например, код функции `greeting()` можно вынести и вне файла xs и подключить его код с помощью дополнительного `#include` заголовочного файла. В дальнейшем можно менять код функций, при этом никак не затрагивая кода интерфейса.

## Примеры XSUB

Список и документацию всех директив XS можно прочитать в perlxs. Изучать же применение директив удобнее на реальных примерах XSUB-функций.

Рассмотрим пример XS-кода модуля `Readonly::XS` (выборочный фрагмент):

```
#include "EXTERN.h"
#include "perl.h"
#include "XSUB.h"

#include "ppport.h"

MODULE = Readonly::XS        PACKAGE = Readonly::XS

int
is_sv_readonly(sv)
    SV *sv
PROTOTYPE: $
CODE:
    RETVAL = SvREADONLY(sv);
OUTPUT:
    RETVAL
```

Что нового видно в этом примере? Во-первых, объявляемая функция `is_sv_readonly()` возвращает результат типа `int`. После названия функции следует описание аргумента функции, в данном случае `sv*`, имеющий тип `SV`. Каждый аргумент функции описывается на отдельной строке. Далее следует директива `PROTOTYPE`, которая указывает Perl-прототип функции — в данном случае `$` — функция принимает скаляр в качестве аргумента. В секции `CODE` обнаружива-

ем, что переменной RETVAL присваивается значение одного из макросов Perl API — SvREADONLY(), который возвращает значение флага SVf_READONLY скаляра sv, т.е. происходит проверка является ли скаляр доступным только для чтения.

Переменная RETVAL — это специальная переменная, которая декларируется автоматически и её тип соответствует типу, который должна вернуть эта функция. В Perl-функциях все аргументы и возвращаемые значения помещаются в стек. В XS существует специальный макрос ST(x), где x — номер позиции в стеке, который адресует каждый элемент помещённый в него. Позиция 0, соответствует ST(0). В простых случаях xsubpp помещает значение RETVAL в ST(0), избавляя от необходимости вручную манипулировать со стеком.

Директива OUTPUT подсказывает xsubpp, какие параметры функции должны быть обновлены, когда XSUB будет завершена и определённые значения будут возвращены в вызывающую Perl-функцию. Без этой директивы xsubpp не поймёт, что надо возвращать именно переменную RETVAL.

В том случае, если требуется самостоятельно управлять стеком, можно переписать интерфейс функции следующим образом:

```
1 void
2 is_sv_readonly(sv)
3     SV *sv
4 PROTOTYPE: $
5 PPCODE:
6     PUSHs(sv_2mortal(newSViv(SvREADONLY(sv))));
```

Вместо секции CODE используется директива PPCODE, что указывает xsubpp, что все операции со стеком выполняются вручную. Развернём выражение в секции PPCODE в порядке выполнения операций:

- макрос SvREADONLY(sv) возвращает значение соответствующего флага скаляра sv;
- макрос newSViv создаёт новый скаляр для целочисленного типа, куда помещает значение флага;
- макрос sv_2mortal помечает созданный скаляр *«смертным»*, т.е. после завершения XSUB, счётчик ссылок скаляра будет уменьшен, что приведёт к освобождению занимаемой им

памяти;
- PUSHs помещает полученный скаляр в стек (который должен иметь достаточно места).

Ручное управление стеком достаточно сложно и в большинстве случае его можно благополучно избегать. Но это единственный способ возвращать больше одного значения из подпрограммы.

### Typemap

Возможно, вы обратили внимание в последнем примере, что описание возвращаемого значения функции s_sv_readonly() изменилось с int на void. Помимо управления стеком xsubpp автоматически выполняет конвертацию типов из Perl в C и обратно. В Perl нет типа int, поэтому существует соответствие, по которому тип int преобразуется в скаляр, содержащий значение целого числа. И, соответственно, аргументы C-функции преобразуются из Perl-типа в соответствующий C-тип. Информация о том, как выполнять преобразования, помещается в файл typemap. По умолчанию xsubpp имеет базовый необходимый набор для преобразования всех простых C-типов в Perl и обратно.

Файл typemap состоит из трёх секций: TYPEMAP, INPUT и OUTPUT. Первая секция начинается сразу с начала файла, в ней в каждой строчке перечисляется какой-либо C-тип и его уникальный идентификатор, называемый XS-типом. С ключевого слова INPUT начинается следующая секция, которая описывает как преобразовывать Perl-типы к C-типам. С ключевого слова OUTPUT следует секция, в которой описывается как выполнить обратное преобразование из C-типов к Perl типам.

Например, в стандартном typemap можно увидеть такие преобразования для типа int:

```
1 int          T_IV
2 INPUT
3 T_IV
4     $var = ($type)SvIV($arg)
5 OUTPUT
```

```
6 T_IV
7     sv_setiv($arg, (IV)$var);
```

Как видно, в первой строке для типа `int` задаётся XS-тип `T_IV`. В секции `INPUT` для XS-типа `T_IV` указывается преобразование. `xsubpp` выполняет подстановку переменных в данном выражении: вместо `$var` подставляется имя переменной, вместо `$type` подставляется C-тип, `$arg` заменяется на имя аргумента. В секции `OUTPUT` описывается обратное преобразование из C-типа в Perl-скаляр.

Вообще стоит отметить, что каждая строка с выражением преобразования в секциях `INPUT` и `OUTPUT` — это Perl-строка, которая будет помещена в двойные кавычки, а затем выполнена через `eval`, для того, чтобы произошла подстановка переменных. По этой причине любые двойные кавычки должны быть экранированы обратным слешем. С другой стороны, это также позволяет производить внедрение исполняемого Perl-кода. Например, реальный пример из файла `typemap` модуля `GTK`:

```
1 T_GtkPTROBJOrNULL
2     $var = SvTRUE($arg)
3         ? Cast$type(SvGtkObjectRef($arg, \"" . ($foo=
             $ntype,$foo=~s/_OrNULL//,$foo). "\"))
4         : 0
```

В данной строке видно, что после неэкранированной двойной кавычки идёт обычный Perl-код, который возвращает название первоначального типа с удалённым фрагментом `_OrNULL` в конце. Чтобы не возникло синтаксической ошибки, после кода добавляется ещё одна неэкранированная двойная кавычка. Это даёт большие возможности для манёвра, но с другой стороны усложняет понимание, поэтому используйте эту силу с умом и осторожностью.

Возьмём следующий пример XSUB, который возвращает среднее значение двух целых чисел:

```
1 double
2 avg(x,y)
3     int x
4     int y
5 CODE:
6     RETVAL = (double) (x+y)/2;
7 OUTPUT:
```

```
8       RETVAL
```

Если выполнить `make` и взглянуть на сгенерированный `xsubpp` C-код, можно увидеть такой результирующий код функции:

```
1  {
2      dVAR; dXSARGS;
3      if (items != 2)
4         croak_xs_usage(cv,  "x, y");
5      {
6      int x = (int)SvIV(ST(0));
7      int y = (int)SvIV(ST(1));
8      double  RETVAL;
9      dXSTARG;
10     RETVAL = (double) (x+y)/2;
11     XSprePUSH; PUSHn((double)RETVAL);
12     }
13     XSRETURN(1);
14 }
```

В 6-ой и 7-ой строках переданные через стек скаляры преобразуется к типу `int` именно так, как и описано в файле `typemap` в секции `INPUT`. Преобразование возвращаемого значения `RETVAL` из типа `double` в скаляр, описано в секции `OUTPUT` `typemap` так:

```
1  T_DOUBLE
2      sv_setnv($arg, (double)$var);
```

Но `xsubpp` в последствии оптимизирует эту запись, заменяя её на макрос `PUSHn`.

Если в функциях модуля вам часто требуется выполнять преобразования к какому-то специфическому типу, вероятно удобнее вынести такую операцию в `typemap`. По умолчанию `xsubpp` ищет файл с названием `typemap` в текущем каталоге или выше по иерархии директорий проекта.

### Файл ppport.h

Отдельно стоит рассказать о файле `ppport.h`. В примере модуля `Readonly::XS` можно увидеть включение заголовочного файла `ppport.h`:

```
#include "ppport.h"
```

Этот файл формируется автоматически с помощью модуля `Devel::PPPort` для того, чтобы дать создаваемому модулю доступ к некоторым API-функциям новых версий Perl для возможности работы с ними на старых версиях Perl. Тем самым улучшая переносимость программ на старые версии интерпретатора. Поддерживаются версии Perl от 5.003 до 5.11.5.

Файл является не простым заголовочным файлом для C. В тоже время это полноценный Perl-скрипт, со встроенной POD-документацией. Скрипт может проанализировать исходные XS-файлы проекта и сообщить какие проблемы в них могут присутствовать в плане совместимости со старыми версиями интерпретатора Perl и даже подготовить патч для исходного кода с необходимыми изменениями. Для этого достаточно запустить команду в корневом каталоге проекта:

```
$ perl ppport.h
Scanning XS.xs...
Doesn't seem to need ppport.h.
--- XS.xs       2012-08-31 22:24:26.000000000 +0400
+++ XS.xs.patched       2013-04-16 00:01:57.386431309
    +0400
@@ -2,7 +2,6 @@
 #include "perl.h"
 #include "XSUB.h"

-#include "ppport.h"


 MODULE = Readonly::XS           PACKAGE = Readonly::XS
```

Прочитать все возможные опции можно во встроенной POD-документации файла:

```
$ perldoc ppport.h
```

Сформировать файл можно командой:

```
$ perl -MDevel::PPPort -e 'Devel::PPPort::WriteFile();'
```

Также он может создаваться автоматически утилитой `h2xs`, если не указывать опцию `--skip-ppport`.

### Переменное число аргументов функции

На примере функции `minstr()` модуля `List::Util` можно рассмотреть каким образом в XS-функции можно организовать обработку переменного числа аргументов:

```
1  #define SLU_CMP_LARGER   1
2  #define SLU_CMP_SMALLER -1
3
4  void
5  minstr(...)
6  PROTOTYPE: @
7  ALIAS:
8      minstr = SLU_CMP_LARGER
9      maxstr = SLU_CMP_SMALLER
10 CODE:
11 {
12     SV *left;
13     int index;
14     if(!items) {
15         XSRETURN_UNDEF;
16     }
17     left = ST(0);
18     for(index = 1 ; index < items ; index++) {
19         SV *right = ST(index);
20         if(sv_cmp(left, right) == ix)
21         left = right;
22     }
23     ST(0) = left;
24     XSRETURN(1);
25 }
```

Функция `minstr()` ищет наименьшую строку в списке. Как видно в 5-ой строке, в прототипе функции указывается троеточие, что даёт понять `xsubpp`, что функция имеет переменное число аргументов. Для обработки аргументов автоматически создаётся переменная `items`, которая содержит количество аргументов, переданных функции. В строке 14 происходит проверка, переданы ли параметры или нет. Если параметров нет, то применяется специальный макрос `XSRETURN_UNDEF`, который приводит к завершению функции и возврату значения константы `&PL_sv_undef` (`undef`). Далее в цикле происходит поиск наименьшего значения среди аргументов, обращаясь к каждому из них с помощью макроса `ST(index)`. Результат помещается в начало стека уже известным макросом `ST(0)`. Макрос

`XSRETURN(1)` производит возврат из XSUB и также сообщает сколько возвращаемых значений находится в стеке. Такой способ возврата значений также может использоваться и является альтернативой уже известного `RETVAL`.

Также, в данной функции используется директива `ALIAS`. Данная директива позволяет создать несколько альтернативных имён для функции, которые будут видны в Perl: `minstr()` и `maxstr()`. В зависимости от того под каким именем вызвана функция в коде инициализируется специальная переменная `ix`, которой присваивается значение, заданное в секции `ALIAS`. Т.е. в случае вызова `minstr()` в `ix` будет содержаться значение константы `SLU_CMP_LARGER`, а в случае `maxstr()` в `ix` буде уже `SLU_CMP_SMALLER`. Это позволяет соответствующим образом менять логику функции в зависимости от использованного имени функции.

### Создание обвязок (bindings) к C-библиотекам

Обвязки (*bindings*) к библиотекам позволяют использовать функции реализованные в этих библиотеках внутри Perl-программ. На сегодняшний день существует огромное множество модулей на CPAN, которые реализуют интерфейс к различным библиотекам. Например, `XML::LibXML`, `Gtk3`, `SDL` и т.д.

Рассмотрим создание такой обвязки на примере создания модуля `My::Libm`, который предоставляет интерфейс к некоторым функциям стандартной математической библиотеки C — `libm`.

Автоматически создадим проект:

```
$ h2xs -b 5.8.9 -A -n My::Libm --use-xsloader --skip-pppport
```

Дополним файл `Libm.xs` интерфейсами нескольких выбранных нами функций: `ceil()`, `floor()` и `pow()`. Кроме того, подключим заголовочный файл `<math.h>` откуда будут извлечены реальные прототипы этих функций. В итоге получится такой код:

```
#include "EXTERN.h"
#include "perl.h"
```

```
#include "XSUB.h"

#include <math.h>

MODULE = My::Libm       PACKAGE = My::Libm

double
ceil(x)
    double  x
    PROTOTYPE: $

double
floor(x)
    double  x
    PROTOTYPE: $

double
pow(x, y)
    double  x
    double  y
    PROTOTYPE: $$
```

Чтобы собираемый модуль слинковался с библиотекой `libm`, необходимо в `Makefile.PL` указать в параметре `LIBS` ключ для линковки с библиотекой `-lm`:

```
use 5.008009;
use ExtUtils::MakeMaker;
WriteMakefile(
    NAME              => 'My::Libm',
    VERSION_FROM      => 'lib/My/Libm.pm',
    PREREQ_PM         => {},
    ($] >= 5.005 ?     ## Add these new keywords
       supported since 5.00
      (ABSTRACT_FROM  => 'lib/My/Libm.pm',
       AUTHOR         => 'Vladimir Lettiev <crux@cpan.org
          >') : ()),
    LIBS              => ['-lm'],
    INC               => '',
);
```

Поскольку заголовочный файл `math.h` находится в стандартном каталоге (`/usr/include`) включать этот каталог в `INC` не нужно.

В файле `lib/My/Libm.pm` указывается список функций, которые мы собираемся импортировать:

```
...
our @EXPORT = qw(
    ceil floor pow
);
...
```

Необходимо убедиться, что есть заголовочный файл `math.h` (идёт в составе glibc) и попробовать собрать проект:

```
$ perl Makefile.PL
$ make
```

Теперь можно проверить (без установки), что модуль и функции работают правильно с помощью простого теста в `t/My-Libm.t`:

```
use strict;
use warnings;

use Test::More;
BEGIN { use_ok('My::Libm') };

is floor(1.9), 1, "floor()";
is ceil(1.9), 2, "ceil()";
is pow(2,3), 8, "pow()";

done_testing();
```

Запустим `prove`:

```
$ prove -b -v
t/My-Libm.t ..
ok 1 - use My::Libm;
ok 2 - floor()
ok 3 - ceil()
ok 4 - pow()
1..4
ok
All tests successful.
Files=1, Tests=4,  0 wallclock secs ( 0.05 usr  0.00 sys
    +  0.05 cusr  0.00 csys =  0.10 CPU)
Result: PASS
```

Модуль работает. Теперь можно выполнить команду `make dist` и смело отправлять это творение на CPAN. Конечно, данный пример очень простой, но он демонстрирует общий подход к задаче.

## Справочные материалы

Для дальнейшего изучения темы рекомендуется чтение следующей документации:

- perlguts — введение во внутренности Perl;
- perlapi — документация по API функциям Perl;
- perlxs — документация по XS;
- perlxstut — вводное руководство с хорошими примерами по созданию XS;
- perlapio — документация по работе с интерфейсом PerlIO в XS.

Google также выдаёт много интересных ссылок, но надо быть внимательным, поскольку очень много материалов уже достаточно сильно устарели. Лучше ориентироваться на последнюю актуальную документацию в составе дистрибутива Perl. Успехов в изучении!

■ *Владимир Леттиев*

# 5. Введение в разработку web-приложений на PSGI/Plack. Часть 2.

*Продолжение статьи о PSGI/Plack. Рассмотрены более подробно Plack::Builder, а также Plack::Middleware.*

В прошлой статье мы рассмотрели спецификацию `PSGI`, как она появилась, почему ей стоит пользоваться. Рассмотрели `Plack` — реализацию `PSGI`, основные его компоненты и написали простейшее API, которое выполняло поставленные перед ним задачи, вскользь рассмотрели основные `PSGI` сервера.

Во второй части статьи мы рассмотрим следующие моменты:

- `Plack::Builder` — мощный маршрутизатор и не только.
- `Plack::Middleware` — расширяем наши возможности при помощи «прослоек».

Мы по-прежнему используем `Starman`, который является preforking сервером (использует модель предварительно запущенных процессов).

## Более пристальный взгляд на Plack::Builder

В предыдущей статье мы вкратце рассмотрели `Plack::Builder`. Теперь пришло время рассмотреть его более подробно. Решение рассматривать `Plack::Builder` вместе с `Plack::Middleware` весьма логично, потому что они очень тесно взаимосвязаны. Рассматривая в разных статьях эти два компонента, обе статьи содержали бы перекрестные ссылки друг на друга, что не очень удобно в формате журнала.

Базовая конструкция `Plack::Builder` выглядит так:

```
1 builder {
2     mount '/foo' => builder { $bar };
```

```
3 }
```

Эта конструкция указывает нам на то, что по адресу `/foo` будет располагаться `PSGI`-приложение (`$bar`). То, что мы обернули в `builder` должно быть обязательно ссылкой на функцию, иначе можем получить ошибку следующего вида:

```
Can't use string ("stupid string")as a subroutine ref while
 "strict refs" in use at /usr/local/share/perl/5.14.2/Plack
/App/URLMap.pm line 71.
```

Маршруты могут быть вложенными, например:

```
1 builder {
2     mount '/foo' => builder {
3         mount '/bar' => builder { $bar; };
4         mount '/baz' => builder { $baz; };
5         mount '/'    => builder { $foo; };
6     };
7 };
```

Эта запись означает, что по адресу `/foo` будет располагаться приложение `$foo`, по адресу `/foo/bar` приложение `$bar`, а по адресу `/foo/baz` приложение `$baz` соответственно.

Однако, никто не мешает записать предыдущую запись в следующем виде:

```
1 builder {
2     mount '/foo/bar' => builder { $bar };
3     mount '/foo/baz' => builder { $baz };
4     mount '/foo/'    => builder { $foo };
5 };
```

Обе записи эквивалентны и выполняют одну и ту же задачу, но первая выглядит проще и понятнее. `Plack::Builder` можно использовать в объектно-ориентированном стиле, но лично мне удобнее использовать его в процедурном виде. Применение `Plack::Builder` в объектно-ориентированном виде выглядит так:

```
1 my $builder = Plack::Builder->new;
2 $builder->mount('/foo' => $foo_app);
3 $builder->mount('/'    => $root_app);
4 $builder->to_app;
```

Эта запись эквивалентна:

```
builder {
    mount '/foo' => builder { $app; };
    mount '/'    => builder { $app2; };
};
```

Какой из способов использовать — дело сугубо индивидуальное. Мы вернемся к рассмотрению `Plack::Builder` после ознакомления с `Plack::Middleware`.

## Plack::Middleware

`Plack::Middleware` — это базовый класс для написания, как говорит нам CPAN, «простых в использовании `PSGI`-прослоек». Для чего это нужно? Рассмотрим на примере реализации некоего API.

Представим, что код нашего приложения выглядит так:

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

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

    my $params = $req->parameters();
    if ($params->{string} && $params->{string} eq 'data')
        {
        $res->body('ok');
    }
    else {
        $res->body('not ok');
    }

    return $res->finalize();
};

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

Это приложение отлично работает, но теперь представим, что вдруг понадобилось принимать данные только в том случае, если они пе-

редаются методом POST.

Тривиальное решение — привести наше приложение к следующему виду:

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

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

    my $params = $req->parameters();
    if ($req->method() ne 'POST') {
        $res->status(403);
        $res->body('Method not allowed');
        return $res->finalize();
    }

    if ($params->{string} && $params->{string} eq 'data')
        {
        $res->body('ok');
    }
    else {
        $res->body('not ok');
    }

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

Понадобилось всего лишь 4 строки, чтобы решить проблему. А теперь представим, что понадобилось сделать еще одно приложение, которое тоже должно принимать данные отправленные только методом POST. Что будем делать? Писать в каждом это условие? Это не вариант по нескольким причинам:

- Увеличивается объем кода, а как следствие его энтропия (простое лучше, чем сложное).
- Больше вероятность сделать ошибку (человеческий фактор).
- Если мы передадим проект другому программисту, он может забыть и сделать что-то не так (человеческий фактор).

Итак, сформулируем проблему. Мы не можем сделать так, чтобы все наши приложения одновременно приобрели определенные свой-

ства не изменяя их код. Или можем?

Механизм `Middleware` отлично подходит для предоставления сквозного функционала всему приложению. Стоит, конечно, чувствовать меру и добавлять только действительно необходимый всей программе код.

Для того, чтобы построить свое `Middleware` (свою прослойку, другими словами), необходимо добиться исполнения следующих условий:

- Находиться в пакете `Plack::Middleware::MYCOOLMIDDLEWARE`, где `MYCOOLMIDDLEWARE` название вашего `Middleware`.
- Расширять базовый класс `Plack::Middleware` (`use parent qw/Plack::Middleware/;`).
- Реализовывать метод (функцию) `call`.

Итак, реализуем простейшее `Middleware` учитывая все вышеперечисленное:

```
1  package Plack::Middleware::PostOnly;
2  use strict;
3  use warnings;
4  use parent qw/Plack::Middleware/;
5  use Plack;
6  use Plack::Response;
7  use Plack::Request;
8
9  sub call {
10     my ($self, $env) = @_;
11
12     my $req = Plack::Request->new($env);
13
14     if ($req->method() ne 'POST') {
15         my $new_res = $req->new_response(405);
16         $new_res->body('Method not allowed');
17         return $new_res->finalize();
18     }
19
20     return $self->app->($env);
21 }
```

Рассмотрим подробнее то, что получилось. Есть код, который находится в пакете `Plack::Middleware` (1 пункт), который наследует базовый класс `Plack::Middleware` (2 пункт), реализовывает метод `call` (3 пункт).

Представленная реализация `call` делает следующее:

- Принимает в качестве параметров экземпляр `Plack::Middleware` и `env` (`my ($self, $env)= @_;`).
- Создает запрос, который принимает приложение (создание аналогично тому, которое использовалась в предыдущих примерах).
- Проверяет, не является ли метод запроса POST, если является, то `Middleware` пропускает обработку запроса дальше.

Рассмотрим, что происходит, если метод запроса не является POST.

Если метод не является POST создается новый объект `Plack::Response` и сразу же возвращается, *не вызывая* приложение.

Вообще функция `call` в `Middleware` может делать ровно 2 действия. Это:

- Обработка `env` ПЕРЕД выполнением приложения.
- Обработка результата ПОСЛЕ выполнения приложения.

Это будет проиллюстрировано в конце статьи, когда мы будем подводить итоги и разбираться в нюансах.

## Совместное использование Plack::Middleware и Plack::Builder

Есть готовая прослойка `Plack::Middleware::PostOnly`, у нас есть PSGI прилжение, у нас есть проблема. Напоминаю, проблема выгля-

дит так: “На данный момент мы не можем глобально влиять на поведение приложений”. Теперь можем. Рассмотрим самый важный момент Plack::Builder — ключевое слово enable.

Ключевое слово `enable` позволяет подключать `Plack::Middleware` к приложению. Делается это следующим образом:

```
my $main_app = builder {
    enable "PostOnly";
    mount "/" => builder { $api_app; };
}
```

Это очень простой и очень мощный механизм одновременно. Объединим весь код в одном месте и посмотрим на результат.

PSGI приложение:

```
use strict;
use warnings;

use Plack;
use Plack::Builder;
use Plack::Request;
use Plack::Middleware::PostOnly;

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

    warn 'WORKS';

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

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

    if ($params->{string} && $params->{string} eq 'data')
        {
        $res->body('ok');
    }
    else {
        $res->body('not ok');
    }
    return $res->finalize();
};

my $main_app = builder {
    enable "PostOnly";
```

```
30     mount "/" => builder { $api_app };
31 }
```

Middleware:

```
1  package Plack::Middleware::PostOnly;
2  use strict;
3  use warnings;
4  use parent qw/Plack::Middleware/;
5  use Plack;
6  use Plack::Response;
7  use Plack::Request;
8
9  sub call {
10     my ($self, $env) = @_;
11
12     my $req = Plack::Request->new($env);
13
14     if ($req->method() ne 'POST') {
15         my $new_res = $req->new_response(405);
16         $new_res->body('Method not allowed');
17         return $new_res->finalize();
18     }
19
20     return $self->app->($env);
21 }
```

Приложение запускается следующей командой:

```
1 /usr/bin/starman --port 8080 app.psgi
```

В коде использовалось `enable "PostOnly"` потому, что `Plack::Builder` автоматически подставляет к имени пакета `Plack::Middleware`. Пишется `enable "PostOnly"`, имеется в виду `enable "Plack::Middleware::PostOnly"` (указать полный путь к своему классу можно используя в качестве префикса `+`, например, `enable "+MyApp::Middleware::PostOnly";` – прим. редактора).

Теперь, если обратиться по адресу `http://localhost:8080/` при помощи метода GET, то получим сообщение о том, что Method not allowed с кодом ответа 405, тогда как при обращении методом POST все будет нормально.

Не зря в коде приложения присутствует строка `warn "WORKS"`. Она подтверждает отсутствие выполнения приложения, если метод не

является POST. Попробуйте отправить GET, вы не увидите этого сообщения в STDERR `starman`.

У `PSGI`-серверов есть еще довольно много интересных особенностей поведения, они обязательно будут рассмотрены в следующих статьях.

Рассмотрим еще несколько полезных моментов `Plack::Middleware`, а именно:

- Обработка результатов ПОСЛЕ выполнения приложения.
- Передача параметров в `Middleware`.

Допустим, есть два `PSGI`-приложения и необходимо сделать так, чтобы одно работало через POST, а другое только через GET. Можно решить проблему в лоб, написав еще одно `Middleware`, которое будет отвечать только на метод GET, например, так:

```
1  package Plack::Middleware::GetOnly;
2  use strict;
3  use warnings;
4  use parent qw/Plack::Middleware/;
5  use Plack;
6  use Plack::Response;
7  use Plack::Request;
8
9  sub call {
10     my ($self, $env) = @_;
11
12     my $req = Plack::Request->new($env);
13
14     if ($req->method() ne 'GET') {
15         my $new_res = $req->new_response(405);
16         $new_res->body('Method not allowed');
17         return $new_res->finalize();
18     }
19
20     return $self->app->($env);
21 }
```

Задача решена, однако остается много дублирования.

Решение этой проблемы как нельзя лучше поможет разобраться со следующими вещами:

- Механизмы передачи переменных в `Middleware`.
- Подключение `Middleware` для приложений индивидуально.

Решение проблемы следующее: передавать желаемый метод в качестве переменной. Вернемся к рассмотрению `enable` из `Plack::Builder`. Оказывается, `enable` умеет принимать переменные. Выглядит это следующим образом:

```
my $main_app = builder {
    enable "Foo", one => 'two', three => 'four';
    mount "/" => builder { $api_app };
}
```

В самом `Middleware` к этим переменным можно добраться напрямую через `$self`. Например, для того чтобы получить значение, переданное переменной one, необходимо обратиться к `$self->{one}` в коде `Middleware`. Продолжаем изменять `PostOnly`.

Пример:

```
package Plack::Middleware::GetOnly;
use strict;
use warnings;
use parent qw/Plack::Middleware/;
use Plack;
use Plack::Response;
use Plack::Request;

sub call {
    my ($self, $env) = @_;

    my $req = Plack::Request->new($env);

    warn $self->{one} if $self->{one};

    if ($req->method() ne 'GET') {
        my $new_res = $req->new_response(405);
        $new_res->body('Method not allowed');
        return $new_res->finalize();
    }

```

```
22      return $self->app->($env);
23  }
```

Перезапускаем starman, делаем запрос на `localhost:8080`, в STDERR видим следующее:

```
two at /home/noxx/perl/lib/Plack/Middleware/PostOnly.pm
line 12.
```

Так передаются переменные в `Plack::Middleware`.

Используя данный механизм, напишем `Middleware`, которое теперь будет называться `Only`.

```
1  package Plack::Middleware::Only;
2  use strict;
3  use warnings;
4  use parent qw/Plack::Middleware/;
5  use Plack;
6  use Plack::Response;
7  use Plack::Request;
8
9  sub call {
10     my ($self, $env) = @_;
11
12     my $req = Plack::Request->new($env);
13
14     my $method = $self->{method};
15
16     $method ||= 'ANY';
17
18     if ($method ne 'ANY' && $req->method() ne $method) {
19         my $new_res = $req->new_response(405);
20         $new_res->body('Method not allowed');
21         return $new_res->finalize();
22     }
23
24     return $self->app->($env);
25 }
26
27 1;
```

Теперь `Middleware` умеет отвечать только на переданный в параметрах метод запроса. Немного изменившееся подключение выглядит так:

```
my $main_app = builder {
    enable "Only", method => 'POST';
    mount "/" => builder { $api_app };
};
```

В данном случае приложение будет исполняться только в том случае, если метод запроса был POST.

Рассмотрим обработку результатов ПОСЛЕ выполнения приложения. Допустим, необходимо чтобы в случае, если метод разрешен, к телу ответа добавлялось слово “ALLOWED”.

То есть, если приложение должно отдавать ok, оно отдаст ok ALLOWED, если, конечно, запрос будет выполнен с допустимым методом.

Приведем `Only.pm` к следующему виду:

```
package Plack::Middleware::Only;
use strict;
use warnings;
use parent qw/Plack::Middleware/;
use Plack;
use Plack::Response;
use Plack::Request;

sub call {
    my ($self, $env) = @_;

    my $req = Plack::Request->new($env);

    my $method = $self->{method};

    $method ||= 'ANY';

    if ($method ne 'ANY' && $req->method() ne $method) {
        my $new_res = $req->new_response(405);
        $new_res->body('Method not allowed');
        return $new_res->finalize();
    }

    my $plack_res = $self->app->($env);
    $plack_res->[2]->[0] .= 'ALLOWED';
    return $plack_res;
}

```

```
29 1;
```

`$self->app->($env)` возвращает ссылку на массив из трех элементов (`PSGI`-спецификация), тело которого модифицируется и отдается в качестве ответа.

Убедится, что это все работает и работает так как надо, можно передав разрешенным методом переменную `string=data` и `string=data1`. В первом случае, если метод разрешен, ответ будет выглядеть “okALLOWED”, во втором “not okALLOWED”.

И в заключение рассмотрим, как именно можно объединить все вышеперечисленное в одно `Plack`-приложение. Возвращаемся к первоначальной задаче. Необходимо разработать простейшее API, которое принимает переменную `string` и если `string=data` ответить ok, иначе not ok, а также соблюдать следующие правила:

- При обращении по адресу `/` отвечать на любой метод.
- При обращении по адресу `/post` отвечать только на метод POST.
- При обращении по адресу `/get` отвечать только на метод GET.

На самом деле, понадобится ровно одно приложение, которое написано — `$api_app` и немного модифицированный `builder`.

В результате, используя все вышеописанное, должно получиться приложение следующего вида:

```
1 use strict;
2 use warnings;
3
4 use Plack;
5 use Plack::Builder;
6 use Plack::Request;
7 use Plack::Middleware::PostOnly;
8 use Plack::Middleware::Only;
9
10 my $api_app = sub {
11     my $env = shift;
12
13     my $req = Plack::Request->new($env);
```

```
14      my $res = $req->new_response(200);
15
16      my $params = $req->parameters();
17
18      warn 'RUN!';
19
20      if ($params->{string} && $params->{string} eq 'data')
            {
21          $res->body('ok');
22      }
23      else {
24          $res->body('not ok');
25      }
26
27      return $res->finalize();
28  };
29
30  my $main_app = builder {
31      mount "/" => builder {
32          mount "/post" => builder {
33              enable "Only", method => 'POST';
34              $api_app;
35          };
36          mount "/get" => builder {
37              enable "Only", method => 'GET';
38              $api_app;
39          };
40          mount "/" => builder {
41              $api_app;
42          };
43      };
44  };
```

Таким образом, подключение `Middleware` работает во вложенных маршрутах `Plack::Builder`. Стоит обратить внимание на простоту и логичность кода.

Отложенный ответ будет рассмотрен в одной из статей посвященных асинхронным серверам (`Twiggy`, `Corona`, `Feersum`).

■ *Дмитрий Шаматрин*

# 6. Обзор CPAN за апрель 2013 г.

*Рубрика с обзором интересных новинок CPAN за прошедший месяц.*

Похоже астрологи объявили апрель месяцем веб-фреймворков. В этом месяце вышло целых два новых веб-фреймворка Cot и Framework::Core, и вышли обновления для ещё десятка (`Dancer`, `Mojolicious`, `Catalyst`, `Kossy`, `Kelp`, `Amon2`, `Drogo`, `Nephia`, `Dancer2`, ...). Если так пойдёт и дальше, то придётся выделять отдельный раздел под веб-фреймворки.

## Статистика

- Новых дистрибутивов — 247
- Новых выпусков — 849

## Новые модули

- AnyEvent::Fork Модуль для корректного создания новых процессов в AnyEvent-приложениях
- SockJS Реализация SockJS для Perl. Модуль позволяет состыковать фронтенд, использующий javascript библиотеку SockJS, и бэкенд, реализуемый на Perl.
- Text::YAWikiFormater Ещё один новый модуль для преобразования wiki-текста в html
- Digest::SpookyHash Реализация алгоритма хэширования SpookyHash
- IO::FDPass Модуль для передачи файловых дескрипторов через сокет. Уникален тем, что работает даже на win32
- Crypt::SRP Реализация протокола SRP для Perl, позволяющего прозводить аутентификацию, устойчивую к прослушиванию и MITM-атакам и не требующий третьей доверенной стороны.

- Language::SNUSP Для ценителей экзотических языков выполнена реализация языка SNUSP, представляющий из себя помесь языка Brainfuck и игры Lemmings.
- HTML::Detergent Модуль для извлечения контента вебстраницы с зачисткой от лишних элементов дизайна, навигации и т.п. Нет, искусственного интеллекта ещё нет, вырезка производится по заданному XPath выражению.
- File::Gettext Модуль для чтения/записи GNU gettext po/mo файлов (без использования `libintl-perl`)
- App::cpanminus::reporter Утилита для разбора логов `cpanm` и генерации отчёта для базы тестеров CPAN.
- String::Print Более удобные в использовании и сопровождении альтернативы старому доброму `printf`.
- Spellunker Проверка орфографии на чистом Perl
- Test::Is и Test::DescribeMe Выполняет тест при определённых условиях. Модули являются реализацией так называемого Ланкастерского Консенсуса (соглашения о тестировании принятые на Perl QA хакатоне 2013 года).
- Hash::Spy Модуль позволяет отслеживать изменения хэша, запуская заданные процедуры при изменениях.
- CljPerl Реализация Lisp на языке Perl. Позволяет вам программировать на Lisp и использовать при этом всю мощь CPAN.
- Monoceros Новый PSGI/Plack сервер, поддерживающий HTTP/1.0, совмещающий в себе две модели параллельной обработки подключений: асинхронный управляющий процесс и предварительно запущенные рабочие процессы.

## Обновлённые модули

- Perl 5.17.11 Новый минорный выпуск версии Perl для разработчиков. Этот релиз является финальным в ветке 5.17, а значит

следующим будет выпущен 5.18.0 . Основное заметное изменение в этой версии — группа возможностей `switch`, такие как оператор `~~`, `given` и `when` помечаются экспериментальными и их использование теперь даёт соответствующее предупреждение.

- Gitalist 0.004002 Современный веб-фронтенд для git-репозиториев на основе `Catalyst`. Этот релиз вышел почти после годового перерыва с исправлением ошибок.

- Cache::Memcached::Fast 0.21 Perl клиент к Memcashed, написанный на C. Исправлена ошибка при работе со связанными (`tie`) скалярами.

- AnyEvent::DBI 2.3 Асинхронный DBI. В этом выпуске исправлены ошибки и улучшены тесты.

- Web::Query 0.16 Ещё одна реализация библиотеки для разбора веб-страниц с интерфейсом похожим на JQuery. Добавлены множество новых методов (`append`, `prepend`, `after`, `before`...) и исправлены ошибки.

- App::FatPacker 0.009016 Утилита `fatpack` позволяет запаковать вашу программу вместе со всеми зависимыми модулями в один “жирный” perl-скрипт без зависимостей. В новом выпуске появилась одна команда `pack`, которая выполняет все операции по запаковке разом.

- DBIx::Class 0.08250 Реализация ORM для выполнения SQL запросов с использованием объектно-ориентированного интерфейса. В новой версии сделано множество улучшений и оптимизаций, утверждается, что текущий код способен работать быстрее, чем `DBIx::DataModel` и `Rose::DB::Object` в некоторых конфигурациях. Оптимизирован метод cursor(), дающий почти 10-кратное увеличение скорости работы при итерациях. Исправлен падающий тест для новейших версий SQLite, реализовано обходное решение для критической ошибки в `DBD::SQLite` и другие исправления.

- Devel::NYTProf 5.02 Новый мажорный релиз лучшего профайлера для Perl. Основной фичей нового релиза стал “Пламенный График” — наглядная визуализация глубины стэка вызо-

вов и загруженности каждого участка кода. Также была увеличена точность вычисления времени выполняемых операций.

- Imager 0.95 Новый стабильный выпуск расширения для Perl для создания изображений. Выпуск содержит исправление ошибок.

- App::ForkProve v0.4.7 Это альтернативная реализация prove, которая предварительно подгружает модули, а затем запускает каждый тест через `fork()` в отдельном процессе, позволяя значительно повысить скорость работы тестов, особенно для тяжёлых приложений. В новой версии произошёл переход на систему сборку `Module::Build`.

- App::Ack 2.04 Это альтернатива `grep` для программистов. Выпущен долгожданный новый мажорный релиз утилиты ack.

- ExtUtils::ParseXS 3.18 Компилятор XS-кода в C-код. В новой версии восстановлена совместимость с Perl 5.6.x.

- ExtUtils::MakeMaker 6.66 Модуль для формирования Makefile. Если у вас нет религиозных предубеждений, то версия из трёх шестёрок и с цитатой из песни *“The Number Of The Beast”* группы *Iron Maiden* в анонсе вас не смутит. Вероятно именно эта версия попадёт в Perl 5.18.

- CPAN 2.00 На QA хакатоне в Ланкастере был сделан релиз новой мажорной версии старейшего модуля CPAN. В новом релизе исправления ошибок и интеграция модуля `App::Cpan`.

- Padre 0.98 Через год и месяц после последнего релиза была представлена новая версия IDE для Perl — Padre. В этом выпуске было исправлено огромное число багов, обновлены переводы интерфейса. Большую задержку в выходе новой версии за последние месяцы вызывали проблемы с тестами модуля `Socket` на платформе Win32, что в конечном счёте привело к переходу на `IO::Socket::IP`.

■ *Владимир Леттиев*

# 7. Интервью с Sawyer X

*Sawyer X — израильский Perl-программист, один из разработчиков Dancer, активный участник Perl-сообщества и TelAviv.pm. Известен своими харизматичными докладами на технических конференциях.*

**Как и когда начал изучать программирование?**

Ближе к старшим классам у нас был внеклассный кружок программирования. Я присоединился и выучил некоторые основы. В действительности я не очень учился вместе с кружком, просто развлекался. Затем в 10 классе я пошел на компьютерные уроки в школе. Мы изучали ассемблер, C и Pascal. На самом деле мне не нравилось учиться и я отставал. Однажды я взял и купил "A Book on C" и вместо класса сам взялся изучать. Когда все начинали школьные проекты, свой я уже закончил. Пока все писали "змейку" (нам предоставили базовую библиотеку для работы с DOS-графикой), я имплементировал алгоритм шифрования, который был кроссплатформенным для Windows, и GNU/Linux с командным интерфейсом (именно для GNU/Linux пришлось использовать ncurses).

**Какой редактор используешь?**

На текущий момент я использую Vim без плагинов или специальных настроек, кроме отличной подсветки синтаксиса и нескольких маленьких удобств. Я попробовал несколько плагинов, но в конце концов все их удалил. Для меня конфигурация IDE очень запутанна и хрупка. Когда мне приходится писать на Perl для Windows, я использую Padre, Perl IDE. Мне в действительности плевать на войны редакторов. Используйте что для вас комфортно, и, желательно, время от времени пробовать новые вещи.

**Как и когда познакомился с Perl?**

Еще будучи в школе на хардкор концерте я встретил хакера, который программировал на Perl. Чувак был действительно крут и посоветовал дать шанс этому языку, я попробовал и взорвался. С тех пор и использую!

**С какими другими языками программирования приятно работать?**

В течение лет я пользовался многими языками (особенно ассемблер, C, C++, Ruby и Python) но, если честно, единственным языком с которым было приятно работать был Perl. С C было тоже весело, но по другим причинам, наверное. Perl несомненно отхватил большую части пирога.. э-э-э.. или лука.

**Какое, по-твоему, самое большое преимущество Perl?**

Технически? Эластичность. Perl не просто гибок, он эластичен. Можно двигать стены, гнуть их, это как чертова матрица! `Devel::Declare` всего лишь один пример того, что с Perl можно делать буквально все. Лимитом может быть только ваша фантазия.

Думаю, что меня и других разработчиков Perl привлекает простотой решения задач, так как он просто искривляет пространство вокруг вашей точки зрения, какой бы она ни была. Если вы можете произнести это, это можно написать на Perl!

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

Параллелизм и сообщество, и точка. Во всяком случае для меня.

Скорость выполнения, конечно, важна, но правильные параллельные вычисления, как по мне, становятся все более и более важными. Я считаю, что поэтому Perl 6 и поставил на это.

Вместе с тем, что параллельные вычисления и скорость важны, есть лимит вовлеченности, если все, что у вас есть, это синтаксис. Я имею в виду, что хороший язык это… хорошо, но сообщество вокруг него дает чувство принадлежности, как одному из основных инстинктов и желаний, который у нас есть как у животных. Мы хотим принадлежности, быть частью чего-нибудь, создавать для себя, для других, с другими, и видеть как это расцветает и принимает форму. Мы хотим существовать и создавать существование. Именно это и есть сообщество. Если мы хотим язык, который устойчив за пределами своих технических возможностей, у нас должно быть хорошее сооб-

щество, чтобы это было нечто большее, чем упорядоченный ассортимент людей.

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

Несколько вещей:

1. Я люблю создавать. Когда я первый раз взял в руки гитару, я не начал учить песню, я попытался написать свою. Я хотел создать что-то новое. Открытый код и свободное ПО продвигают эту идею. Они говорят "давайте что-то сделаем!" — Perl как бы центр этого. Perl всегда говорит "не бойся пробовать, не бойся ошибиться".
2. Я люблю жаловаться на вещи и чинить их. У меня уже хорошо получается показывать и говорить "это фигня", и я стараюсь научиться чинить проблемы. Открытый и свободной код позволяют делать именно это. Он подталкивает к участию. Perl потрясающий пример вовлечения людей. Сильно впечатляет, когда вы наблюдаете насколько много Perl-программистов работают над одним, двумя или десятью проектами одновременно. Можно вовлечь людей в десятки проектов в течение одного года. Это сумасшествие какое-то. Посмотрите на rafl (Florian Ragwitz — прим. перев.)! Он везде!
3. Я люблю взаимодействовать с людьми. Если и есть что либо более мотивирующее, то это возможность работать с другими. Мне удается работать с самыми умными людьми, которых я когда либо встречал. У Perl есть много ярких умов. Большая их часть это охрененно крутые люди и я чрезвычайно рад, что встретил их и могу считать себя другом хотя бы некоторых из них. Есть нечто волнительное в работе с другими над одной целью. Это заполняет тебя значимостью и удовлетворением.

**Кажется ты пишешь много асинхронного кода. Почему выбрал AnyEvent, в то время как многие наговаривают на этот модуль?**

Вообще я начал с POE. Давным-давно, когда я пытался понять что к чему, я зашел на IRC канал и задал вопрос. Мне советовали написать свой мультиплексор, чтобы лучше понять как они работают.

Один собеседник после моих попыток что-то собрать в течение часа, спросил “что конкретно тебе непонятно?”. Затем в течение получаса он объяснил мне POE и помог лучше понять этот модуль. Позже я узнал, что это был Rocco Caputo, автор POE.

Я решил попробовать AnyEvent из-за более удобного интерфейса и лучшей скорости. Я до сих пор люблю POE (и Reflex выглядит впечатляюще), но он слишком многословен для меня. Мне скорее всего придется работать с Reflex в будущем, и я очень рад этому.

Я не сильно замечал, что люди наговаривают на AnyEvent, это больше как высказывание пренебрежения к тому как автор (Marc) иногда выражает свои мысли и как минимум к одному случаю, когда он сломал модуль, который неправильно использовать API у AnyEvent. Я бы предпочел не углубляться в это, более того я согласен с некоторыми вещами, и не согласен с другими. Кроме того, есть много того, чего я не знаю.

Единственная вещь, которая меня беспокоит, это что у AnyEvent нет сообщества. Надеюсь, что это изменится в будущем.

**Ты присоединился к проекту Dancer довольно рано. Почему думаешь он привлек тебя?**

Несколько вещей: он был небольшим, был легкий интерфейс с DSL, и была теплая атмосфера на IRC-канале. Когда я начал участвовать было три или четыре человека на канале, если я правильно помню.

Когда я написал свою первую программу на Dancer, я думал, что он не поддерживает CGI (я еще тогда не знал про PSGI) и написал пост в свой блог что-то вроде “плохо, что нет такой поддержки, фреймворк выглядит круто”. В тот вечер я пошел есть хумус со своей девушкой и на пару минут решил выйти онлайн и проверить, пробовал ли кто-то Dancer и CGI. Я увидел чей-то пост “Sawyer написал про Dancer, и что он не поддерживает CGI, но это не так. Вот такие настройки нужно использовать!” — это был Alexis Sukrieh, автор Dancer. Я был сильно поражен тем, что он увидел мой пост и в ответ написал свой. Это было круто. Поэтому я и зашел на IRC-канал.

Очень манила простота помощи в Dancer и работа с другими. Ты

становился частью только лишь переступив порог, и я чувствовал себя прекрасно, работая над фреймворком. Люди в сообществе приятные и вдумчивые, ты чувствуешь, что вложение своего времени в Dancer полезно для пользователей и Perl-сообщества вообще.

**Расскажи о своей роли в TelAviv.pm и группе вообще. Важны ли встречи сообщества?**

В Israel.pm было несколько организаторов, в основном Shlomi Fish и Gabor Szabo. После того как Shlomi перестал заниматься встречами, я решил попробовать что-нибудь организовать и увеличить посещаемость. Я назвал это TelAviv.pm, потому что мы встречались в университете Тель-Авива. Хотя довольно скоро мы переместились в Рамат-Ган, который находится в 20 минутах от Тель-Авива, но там также свой муниципалитет. Я все равно оставил имя, потому что это все еще часть большого района Тель-Авива (что само по себе странно, потому что Тель-Авив очень небольшой). Я сделал плохенький вебсайт (позже обновил на хороший) и связался с людьми, чтобы они подготовили доклады, и подготовил несколько своих. В течение нескольких последних лет я организовывал встречи, иногда с помощью Shlomi или Gabor, которые все еще активные участники сообщества. В последние месяцы мы решили, чтобы каждую встречу организовывал кто-то другой. Это было довольно успешно. Мы также успешно организовали два израильских воркшопа.

Я думаю, что встречи сообщество чрезвычайно важны! Посетив встречу вы можете:

- Улучшить свои знания языка и сторонних модулей
- Научиться разным трюкам
- Встретить потенциальных коллег и работодателей
- Получить практику выступлений
- Завести друзей и возможных соавторов
- Получить помощь с работой или идеей
- Хорошо провести время.

Иногда кажется, что это бессмысленно, но когда вы начинаете посещать такие мероприятия, открываетесь и встречаете новых людей,

вы начинаете понимать как это потрясающе. Это как бесплатный наркотик, который дарит хорошее настроение, делает вас умнее и лучше на своей работе без побочных эффектов кроме траты некоторого времени :)

**Ты часто пишешь в твиттер об опыте работы с git. Чем особенная эта система контроля версий?**

Git один из лучших инструментов для любого автора, будь-то писателя, программиста или графика. Конечно, есть и другие инструменты, но все они фиговые при сравнении. Git небольшой, быстрый и мощный. В своем роде это Perl систем контроля версий.

**Недавно ты был в Румынии. Что делал?**

Для меня было честью посетить юбилейную встречу Cluj.pm. Я сделал несколько докладов и провел время с отличными людьми. Я написал детальный отчет у себя в блоге. Покороче? Съездите туда! Повстречайте этих людей! Посетите их конференции! Мы так многому можем от них научиться!

**Твои выступления на технических мероприятиях очень позитивны и энергичны. В чем твой секрет?**

Спасибо :)

Я просто думаю, что то, что есть в Perl-сообществе, техническом и социальном, это так круто, что я не могу сдержаться.

Есть много захватывающих и забавных вещей в Perl. Когда вы смотрите на другие сообщества, вокруг других языков, не складывается впечатление, что у них есть яркие преимущества (у многих вообще никаких преимуществ нет), но они знают как увлечься тем, что у них есть. Они представляют какой-то модуль, который вы знаете есть уже лет 7 в Perl, но это их сильно волнует! В Perl вы можете написать что-нибудь крутое и люди просто ответят холодным “круто” и все. Мы должны увлекаться, возбуждаться и должны понять насколько фантастически то, что у нас есть и восторгаться этим!

Также я учитываю когда выступаю, что если доклад не веселый, то

тяжело чему-то научиться. Тяжело сосредотачиваться когда материал сухой. Он может быть интересен с технической точки зрения, но нужно представить его в интересной и увлекательной манере. Посмотрите на Paul Fenwick и его выступления. Можно ли представить кого-нибудь кто бы его не слушал и не научился чему-нибудь? Конечно, не все могут быть как Paul (даже Paul трудно быть собой), но мы может сделать гораздо больше, чем показать кусок кода. Это наши конференции и встречи, не так ли? Пусть же нам будет весело!

**Где ты сейчас работаешь? Сколько твоего времени проводить за программирование на Perl? Ты где-то упоминал, что занимаешься также администрированием, это до сих пор так?**

Сейчас я работаю в двух компаниях: электронная коммерция и VoIP стартап. Мне повезло с моим начальником, который позволяет мне работать с Perl. Я начал с 50% времени программирования на Perl и 50% администрирования. Когда я стабилизировал инфраструктуру, я смог запрограммировать некоторые части на Perl, вместо бегания за своим хвостом (что администраторы часто делают). Таким образом у меня была стабильная основа и я начал писать большие компоненты инфраструктуры на Perl. Сейчас около 80% моего времени я пишу на современном Perl, как части инфраструктуры, так и сервисов, окружающих ее.

Таким образом я занимался администрированием до недавнего времени, потому что в июне я переезжаю в Европу на другую работу. Возможно, я скоро напишу об этом в своем блоге.

**Стоит ли сейчас советовать молодым программистам учить Perl?**

Определенно! Нам только нужно научиться радоваться этому. Нужно время, чтобы понять, что решение элегантное, почему кусок кода не сильно отличается от произведения искусства. Черт возьми, нам всем требуется время, чтобы это понять. Молодые программисты, у которых еще меньше опыта, не всегда просто понимают почему Moose захватывающий. Мы должны иметь возможность объяснить это. Мы должны научиться привлекать их не в силу нашего опыта, а в силу их неопытности. Мы должны говорить на их языке и подключать их. Шаг за шагом они научатся ценить вещи с более изыс-

канного аспекта, поэтому не стоит об этом волноваться. Сейчас мы должны их заинтриговать и заинтересовать.

## Вопросы от читателей

**Saywer это твое настоящее имя?**

Вообще-то нет. Это прозвище, которое очень давно мне дали мои близкие друзья. Оно происходит от художественного героя Тома Сойера. Есть история почему так произошло, но я сохраню ее для следующего интервью :)

Если вы не уверены стоит ли меня называть Sawyer, подумайте над следующим: я люблю заводить друзей, поэтому без проблем зовите меня как это делают мои друзья!

**Почему ты нецензурно выражаешься как пятилетний?**

Это фраза соскочила у меня с языка (на одной из Perl-конференций Sawyer рассказывал как он ходил с товарищем по городу и матерился как пятилетний – прим. перев.). Я хотел сказать “тринадцатилетний”. Наверное ругань для меня один из инструментов акцентирования, когда я говорю. В смысле, это сразу заметно! Также я вырос, окруженный американскими экшн-фильмами и ругательства в Израиле очень распространены, так что моя речь подстроилась под это. Ужасно, не правда ли?

**Удалось выпить с Sebastian Riedel?**

Еще нет. Но я на это надеюсь!

Мне удалось встретиться с Glen Hintle (активный участник Mojolicious-сообщества — прим. перев.), очень обаятельный человек. Надеюсь, что в следующий раз когда его встречу будет больше времени поболтать.

**Приедешь в Киев на YAPC::Europe в этом году?**

Я должен был ехать, потом не смог, и теперь есть вероятность, что у меня будет такая возможность. Я не могу обещать, потому что люди будут злиться на меня, если я не приеду, но в конце концов очень может быть я смогу принять участие. Вопрос в том, насколько поздно можно подать заявку на доклад :)

■ *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) Шартрёз, 2) Рэндел Шварц, 3) Абигэйл, 4) Не связанное с программированием участие в Perl-сообществе, 5) 1989, 6) $foo, 7) Насекомое (Camelia), 8) Windows ME, 9) Python, 10) Сан-Хосе

1. В какой версии Perl были добавлены регулярные выражения?

    1. 0
    2. 1
    3. 2
    4. 3
    5. 4

2. Как называется протокол вывода процесса тестирования в Perl?

    1. TOP
    2. TAP
    3. TEP
    4. TIP
    5. TDD

3. На сколько килобайт отличаются версии 5.003_97g и 5.003_97h?

    1. 125
    2. 16
    3. 42
    4. 3
    5. 2014

4. Что выведется в результате `say say say say 'hi'`?

1. `"hi"`
2. `"hi\n1\n1\n1\n"`
3. ничего
4. будет ошибка
5. segmentation fault

5. Когда выйдет Perl 6?

1. Через 20 лет
2. На Рождество
3. Никогда
4. Уже вышел
5. Никто не знает

6. Кто скрывается за ником **kraih**?

1. Larry Wall
2. Анатолий Шарифулин
3. Sebastian Riedel
4. Peter Rabbitson
5. brian d foy

7. Сколько людей сделали хотя бы один коммит в код MetaCPAN?

1. 14
2. 23
3. 54
4. 78
5. 79

8. Как расшифровывается аббревиатура PM, обозначающая локальную Perl-группу?

1. Perl Monks
2. Perl Monsters
3. Perl Mongers
4. Perl Maniacs
5. Perl Machos

9. Где была организована первая русскоязычная Perl-конференция?

1. Киев
2. Санкт-Петербург
3. Одесса
4. Владивосток
5. Москва

10. Как называется еженедельная рассылка Perl-новостей?

    1. Perl News
    2. Perl Weekly
    3. Perl Every Week
    4. Perl Buzz
    5. Perl Pulse

■ *Вячеслав Тихановский*

# 9. Perl-вакансии

*Рубрика с интересными вакансиями.*

В компании WebbyLab на постоянной основе открыта вакансия Perl-разработчика.

Необходимые навыки:

- Опыт работы с Perl более 1 года.
- Опыт работы с базами данных.
- Владение HTML, CSS, JavaScript, jQuery.
- Понимание и умение использовать ООП в Perl.
- Опыт и желание работать в команде.
- Писать код только с «use strict» и «use warnings»!

Будет плюсом:

- Знание ОС Linux
- Умение работать с системами контроля версиями git/svn.
- Опыт проектирования и разработки высоконагруженных приложений.
- Огромным плюсом будет, если в Вашем резюме есть ссылки на реальные проекты и/или на исходные кода расположенные на одном из хостингов проектов (A la GitHub, Bitbucket, SourceForge и т.п.).

Мы предлагаем:

- Свободный график.
- Испытательный срок 1 мес.
- Радушный коллектив.
- Оплачиваемый отпуск 4 недели.

■