Pragmatic Perl 5

pragmaticperl.com

Выпуск 5. Июль 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:

- Андрей Шитов (ash)
- Дмитрий Шаматрин (justnoxx)
- Виктор Турский (koorchik)
- Денис Федосеев (alpha6)

Корректор: Андрей Шитов (ash)

Выпускающий редактор: Вячеслав Тихановский (vti)

Ревизия: 2014-11-29 23:29

Оглавление

1 От редактора 1

2 LIVR (Language Independent Validation Rules) — независимые от языка правила валидации 2

3 Введение в разработку web-приложений на PSGI/Plack. Часть 4. Асинхронность 14

4 Многопроцессовый сервер на AnyEvent 25

5 Обзор CPAN за июнь 2013 г. 31

6 Интервью с брайаном ди фоем про будущее. Часть 1 . 33

7 Perl Quiz 40

1. От редактора

Конференция YAPC::Europe 2013 в Киеве уже совсем скоро. Отличная возможность познакомиться и пообщаться с известными Perl-программистами, в том числе и с автором языка — Ларри Уоллом.

Напоминаем, что журнал проводит розыгрыш одного билета на YAPC::Europe! Для участия вам необходимо всего лишь подписаться на журнал и зарегистрироваться на сайте конференции. Победитель будет выбран случайным образом и объявлен в августовском выпуске.

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

Приятного чтения.

■ *Вячеслав Тихановский*

2. LIVR (Language Independent Validation Rules) — независимые от языка правила валидации

Предложен универсальный формат записи правил валидации с любым уровнем вложенности. Представлено несколько реализаций для разных языков программирования.

Каждый программист регулярно сталкивается с необходимостью проверки данных на их валидность. Для решения этой задачи существует множество различных библиотек и способов описания валидации. Одни предоставляют только валидацию, другие предоставляют весь функционал по получению данных от пользователя, начиная от генерации формы для ввода данных. Некоторые библиотеки предоставляют возможность расширения, некоторые — нет.

Также, если вы занимаетесь веб-разработкой, то часто хочется иметь возможность описывать валидацию на сервере и на клиенте одинаковым способом.

В нашей компании мы в основном используем 3 языка для разработки веб-приложений: Perl, Javascript, PHP. И хотелось бы при переключении между проектами иметь возможность использовать одинаковый подход к валидации.

В связи с этим мы решили написать универсальный валидатор.

Требования к валидатору

В первую очередь мы определили основные требования к валидатору:

1. Правила валидации должны быть декларативными и не зависеть от языка програмирования, который используется. Под

этим подразумевается, что правила не должны содержать код, ссылки на функции, вызовы методов. Правила должны быть просто структурой данных, которую можно передать по сети или сохранить на диске в виде файла.

2. Для любого значения должна быть возможность описать любое количество правил валидации.
3. При валидации данных должна быть возможность получить ошибки для всех невалидных значений. Это означает, что валидация не останавливается на первом невалидном значении, а всегда продолжает проверять оставшиеся значения. Например, если есть форма регистрации, то это позволит подсветить все поля с ошибками, а не только первое.
4. Все данные, для которых не описаны правила валидации, должны быть исключены. Это вопрос безопасности. То есть, валидатор после валидации должен возвращать очищенную структуру.
5. Должна быть возможность описывать правила валидации не только для простых структур данных, но и для сложных иерархических.
6. Описание правил должно быть понятным. Смотря на правила валидации, должно быть ясно, какая структура будет им соответствовать. То есть, правила могут выступать неким описанием формата/спецификацией данных.
7. Правила должны возвращать понятные коды ошибок. Было решено не использовать сообщение об ошибках, а использовать строковые коды ошибок. Строковые коды предназначены для обработки кодом и понятны человеку более, чем числовые. Примеры, кодов ошибок: “REQUIRED”, “NOT_POSITIVE_INTGER”, “WRONG_EMAIL”.
8. Расширяемость. Стандартного набора правил часто бывает недостаточно, и должна быть возможность создавать свои правила валидации. Для валидатора все правила должны быть равнозначными. Например, валидатор не делает различия между “required” (праверяет, что значение присутствует) и “nested_object”(описание валидации вложенного объекта).
9. Универсальность. Валидатор должен быть универсальным и не быть завязан на валидациию только пользовательских форм. Должна быть возможность использовать его для валидации пользовательского ввода, для валидации конфигураци-

онных файлов, для контрактного программирования. По сути, валидатор должен отвечать исключительно за валидацию данных.

10. Поддержка Unicode.

Спецификация

Поскольку изначально валидатор задумывался как многоязычный инструмент (Perl, PHP, JS), то было решено начать со спецификации.

Спецификация должна:

1. Описывать формат правил валидации и коды ошибок.
2. Описывать набор правил, которые должны поддерживаться каждой реализацией.
3. Иметь набор тест-кейсов для проверки реализации на соответствие спецификации.

Формат правил валидации

Мы какое-то время использовали `Validate::Tiny`. Это очень неплохой модуль и многие идеи по описанию мы взяли с него.

Вот так может выглядеть валидация регистрационной формы:

```
{
    name      => 'required',
    email     => ['required', 'email'],
    gender    => {one_of => [['male', 'female']]},
    phone     => {max_length => 10},
    password  => ['required',    {min_length => 10}],
    password2 => {equal_to_field => 'password'},
    address   => {
        nested_object => {
            city => 'required',
            zip  => ['required', 'positive_integer']
        }
    }
}
```

Как это интерпретировать? Описываются пары ключ-значение для каждого поля. Ключ — это имя поля, значения — это правила валидации: ПОЛЕ => ПРАВИЛО_ВАЛИДАЦИИ.

Правило валидации — это просто имя функции со списком аргументов. Общая структура правила такая: { ФУНКЦИЯ => [АРГУМЕНТ1, АРГУМЕНТ2, ...] }.

Если только один аргумент, то можно использовать сокращенную запись — { ФУНКЦИЯ => АРГУМЕНТ1 }, если вообще нет аргументов, то можно просто написать имя функции — “ФУНКЦИЯ”.

Например,

```
{ required => [] }                  # required()
{ max_length => ['5'] }             # max_lenght(5)
{ max_length => '5' }               # max_lenght(5)
{ one_of => [['male', 'female']] } # one_of(['male', '
    female'])
```

Также было решено, что только “required” должен проверять наличие значения. Остальные правила валидируют значение только если оно присутствует и не отвечает за проверку его присутствия.

Правила могут не только проверять значение, но и изменять его.

Возврат ошибок

Одним из требований к валидатору является возможность получать ошибки для всех значений. Также нужно учитывать, что валидатор может валидировать иерархические структуры. Было решено, что в случае возникновения ошибок мы получим структуру, аналогичную входящей, но вместо значений в ней будут находится коды ошибок (только для полей с ошибками).

Например для вышеописанной структуры при передаче

```
{
    name      => '',
    gender    => 'male',
    phone     => '1234567890123',
```

```
    password  => 'password12345',
    password2 => 'password12345',
    address   => {
        city => 'Kiev',
        zip  => 'FK12321'
    }
}
```

Мы получим следующую структуру, описывающую ошибку:

```
{
    name    => 'REQUIRED',
    email   => 'REQUIRED',
    phone   => 'TOO_LONG',
    address => {
        zip => 'NOT_POSITIVE_INTEGER'
    }
}
```

Стандартный набор правил

Правильный выбор правил валидации, которые должны поддерживаться каждой реализацией, достаточно важная задача. Чем больше правил в спецификации, тем больше времени требуется на реализацию валидатора. С другой стороны, чем меньше правил, тем менее он удобен и тем менее переносимы правила, поскольку программисту необходимо будет реализовать свои правила сразу для нескольких языков.

На данный момент стандартный (должны поддерживаться каждой реализацией) набор правил включает следующие:

- Базовые правила
  - required
  - not_empty
- Правила для валидации строк
  - one_of
  - max_length

  - min_length
  - length_between
  - length_equal
  - like

- Правила для валидации чисел
  - integer
  - positive_integer
  - decimal
  - positive_decimal
  - max_number
  - min_number
  - number_between

- Специальные правила
  - email
  - equal_to_field

- Метаправила (для создания правил на базе других правил)
  - nested_object
  - list_of
  - list_of_objects
  - list_of_different_objects

Примеры и коды ошибок для всех правил описаны в LIVR-спецификации, остановимся только на метаправилах.

Метаправила (для создания правил на базе других правил) Метаправила — это правила, которые позволяет скомбинировать простые правила в более сложные для валидации сложных иерархических структур данных. Важно понимать, что валидатор не делает различия между правилами и метаправилами. Метаправила ничем не отличаются от того же “required”.

nested_object Позволяет описывать правила валидации для вложенных объектов.

Код ошибки зависит от вложенных правил. Если вложенный объект не является хешом, то поле будет содержать ошибку: “FORMAT_ERROR”.

Пример использования:

```
address: { 'nested_object': [{
    city: 'required',
    zip: ['required', 'positive_integer']
}]}
```

list_of Позволяет описать правила валидации для списка значений. Каждое правило будет применяться для каждого элемента списка.

Код ошибки зависит от вложенных правил.

Пример использования:

```
{ product_ids: { 'list_of': [[ 'required',  '
    positive_integer' ]] }}
```

list_of_objects Позволяет описать правила валидации для массива хешей. Правила применяются для каждого элемента в массиве.

Код ошибки зависит от вложенных правил. В случае если значение не является массивом, для поля будет возращен код “FORMAT_ERROR”.

Пример использования:

```
products: ['required', { 'list_of_objects': [{
    product_id: ['required','positive_integer'],
    quantity: ['required', 'positive_integer']
}]}]
```

list_of_different_objects Позволяет описать правила для списка разного вида объектов. Правила валидации будут применяться к каждому объекту.

Код ошибки зависит от вложенных правил валидации. Если вложенных объект не является хешом, то поле будет содержать ошибку “FORMAT_ERROR”.

Пример использования:

```
"products": ["required", { "list_of_different_objects": [
    "product_type", {
        "material": {
            "product_type": "required",
            "material_id": ["required", "positive_integer
               "],
            "quantity": ["required", {"min_number": 1} ],
            "warehouse_id": "positive_integer"
        },
        "service": {
            "product_type": "required",
            "name": ["required", {"max_length": 10} ]
        }
    }
]}]
```

В этом примере валидатор будут смотреть на “product_type” в каждом хеше и, в завимости от значения этого поля, будет использовать соответствующие правила валидации.

Стандартный набор правил планируется расширить. Одним из кандидатов является “not_empty_list” - аналог “not_empty”, но для массива. И “required_if”.

Набор тест-кейсов

Для тестирования реализации был создан набор тест-кейсов. Это набор файлов в формате JSON, который позволяет протестировать реализацию на соответствие требованиям спецификации. Каждый позитивный тест, это 3 файла:

- rules.json — описание правил валидации;
- input.json — структура, которая передается валидатору на проверку;

- output.json — очищенная структура, которая получается после валидации.

Каждый негативный тест вместо `output.json` содержит `errors.json` с описанием ошибки, которая должна возникнуть в результате валидации.

Реализация

Perl-реализация находится на CPAN — это модуль Validator::LIVR. Также имеется LIVR::Contract, который позволяет использовать LIVR в контрактном программировании.

Validator::LIVR

Использовать валидатор достаточно просто, а как это делать, должно быть понятно из примера:

```
use Validator::LIVR;
Validator::LIVR->default_auto_trim(1);

my $validator = Validator::LIVR->new(
    {
        name   => 'required',
        email  => ['required', 'email'],
        gender => {one_of => [['male', 'female']]},
        phone => {max_length => 10},
        password  => ['required',    {min_length => 10}],
        password2 => {equal_to_field => 'password'}
    }
);

if (my $valid_data = $validator->validate($user_data)) {
    save_user($valid_data);
}
else {
    my $errors = $validator->get_errors();
    ...;
}
```

Регистрация правил Для валидации используются функции обратного вызова, которые осуществляют проверку значений, это очень похоже на формат Validate::Tiny. Попробуем описать новое правило под названием “strong_password”. Будем проверять, что значение больше 8 символов и содержит цифры и буквы в верхнем и нижнем регистрах.

```
my $livr = {password => ['required', 'strong_password']};

my $validator = Validator::LIVR->new($livr);

$validator->register_rule(
    strong_password => sub {
        return sub {
            my $val = shift;

            return if !defined($value) || $value eq '';

            return "WEAK_PASSWORD"
              if length($value) < 8
              || $value !~ m/[0-9]/
              || $value !~ m/[a-z]/
              || $value !~ m/[A-Z]/;

            return;
          }
    }
);
```

Теперь добавим возможность задавать минимальное количество символов в пароле и зарегистрируем это правило как глобальное (доступное во всех экземплярах валидатора).

```
my $livr = {password => ['required', {'strong_password'
   => 10}]};

sub strong_password {
    my ($min_length) = @_;

    return sub {
        my $val = shift;

        return if !defined($value) || $value eq '';

        return "WEAK_PASSWORD"
          if length($value) < $min_length
          || $value !~ m/[0-9]/
```

```
14            || $value !~ m/[a-z]/
15            || $value !~ m/[A-Z]/;
16
17          return;
18      };
19  }
20
21  Validator::LIVR->register_default_rule(
22      'strong_password' => \&strong_password);
```

Вот так, достаточно просто, происходит регистрация новых правил. Если необходимо описать более сложные правила, то лучшим вариантом будет посмотреть список стандартных правил, реализованных в валидаторе:

- Validator::LIVR::Rules::Common
- Validator::LIVR::Rules::Numeric
- Validator::LIVR::Rules::String
- Validator::LIVR::Rules::Special
- Validator::LIVR::Rules::Helpers

Есть возможность регистрации правил, которые будут не только валидировать значение, но и изменять его. Например, приводить к верхнему регистру или удалять лишние пробелы.

LIVR::Contract

LIVR::Contract — это экспериментальная реализация контрактного програмирования при помощи LIVR. Модуль позволяет описать контракты для классов при помощи LIVR. На данный момент контракты можно описывать только в классе, в котором находится реализация, но в планах добавить возможность описывать контракты в отдельных подключаемых файлах, используя для этого механизм “ролей”. Этот модуль пока еще не готов к “продакшн”-применению.

Планы на будущее

В ходе использования LIVR в реальных проектах возник ряд идей по улучшению:

1. Планируется добавить новые правила в спецификацию. NOT_EMPTY_LIST — будет проверять, что список не является пустым.
2. Планируется добавить дополнительную логику для обработки null-значений.
3. Будет описан механизм добавления правил-фильтров (поддержка уже есть). Это позволит делать следующие записи: `[ email => ['required', 'email', 'to\_lower\_case' ]`.
4. Планируется сделать полноценную реализацию валидатора на JavaScript.

■ *Виктор Турский, технический директор компании WebbyLab*

3. Введение в разработку web-приложений на PSGI/Plack. Часть 4. Асинхронность

Продолжение цикла статей посвященных разработке PSGI/Plack. Разбираемся с асинхронностью.

В предыдущий статьях мы рассмотрели основные аспекты разработки под PSGI/Plack, которых, в принципе, достаточно для разработки приложений практически любой сложности.

Мы разобрались, что такое `PSGI`, разобрались как устроен `Plack`, затем мы разобрались, как устроены основные компоненты `Plack` (`Plack::Builder`, `Plack::Request`, `Plack::Middleware`). Затем мы подробно рассмотрели `Starman`, который является хорошим `PSGI`-сервером, готовым для использования в production.

Нюанс

Все, что было рассматрено ранее, касалось разработки под модель выполнения, которая называется **синхронной**. Сейчас рассмотрим асинхронную модель.

Синхронность и асинхронность

Синхронная модель это просто и понятно. Все происходит друг за другом в определенном порядке. Это называется процессом выполнения. Рассмотрим один процесс интерпретатора, который, скажем, выполняет цикл, один из элементов которого — ввод пользовательской информации. Следующая итерация цикла не будет выполнена, пока не будет окончена предыдущая, которая включает в себя ожидание ввода пользователем данных. Это — синхронная модель.

Пока пользователь ничего не вводит, программа ожидает ввода и **ничего полезного** не делает. Эта ситуация называеся блокировкой процесса исполнения. В этом случае простой программы просто

утилизирует процессорное время. А вот если в процессе ожидания пользователя программа делает что-то другое, **ожидая** ввода, то процесс становится асинхронным, а ситуация, соответственно, — неблокирующей.

Идем в бар

Рассмотрим в качестве примера бар. Простой бар или паб, в котором клиенты сидят и пьют пиво. Клиентов много. В баре работают два официанта — Боб и Джо. Они работают по двум разным схемам. Боб подходит к клиентам, принимает заказ, идет к барной стойке, заказывает бармену бокал пива, ждет, пока бармен нальет бокал, относит его клиенту, ситуация повторяется. Боб работает синхронно. Джо же поступает совсем по другому. Он принимает заказ у клиента, идет к бармену, говорит ему: “Эй ты, налей-ка бокал %beername%”, затем идет принимать заказ у следующего клиента. Как только бармен наливает бокал, он зовет Джо, который забирает бокал и относит его клиенту.

В этом случае Боб работает синхронно, а Джо, соотственно, асинхронно. Модель работы Джо — событийно-ориентированная. Это наиболее популярная модель работы асинхронных систем. В нашем случае ожидание ввода — время, необходимое на заполнения бокала пивом, менеджер событий — бармен, а событие — это крик бармена “%beername% налито”.

Проблема

Вот теперь у читателей, которые никогда не работали с асинхронными системами, должен возникнуть вопрос. “А зачем, собственно, делать синхронные вещи, если асинхронность быстрее и удобнее?”.

Это очень популярное заблуждение, но это не так. Асинхронные решения тоже имеют ряд проблем и недостатков. Очень много где можно прочитать, что асинхронные решения более производительные, чем синхронные. И да, и нет.

Вернемся к официантам. Боб работает неторопливо, рассказывает анекдоты бармену, размеренно разносит бокалы, а Джо постоянно мотается как угорелый. Нагрузка на Джо, естественно, выше, т.к. он делает гораздо больше всего одновременно. Нагрузка же на Боба минимальна, пока нет клиентов. Как только клиентов становится много, они начинают громко требовать свое пиво и торопить Боба. Нагрузка **со стороны клиента** на него возрастает, но Боб продолжает работать в том же темпе, ему плевать, от своей схемы работы он отказываться не собирается, и пусть хоть небо рухнет.

Итак, отсюда можно сделать вывод, что асинхронность это неплохо, но следует понимать, что асинхронная система будет постоянно находиться под нагрузкой. Нагрузка, в принципе, будет такая же, как и на синхронную систему, но с одним отличием. Синхронная система подвержена **пиковым** нагрузкам, а асинхронная эти нагрузки "размазывает" по времени исполнения.

Ну и самое главное, нельзя забывать о том, что любая система может выполнять одновременно столько задач, сколько ядер процессора доступно процессу.

Асинхронный PSGI/Plack

Классическое `Plack`-приложение (пропустим секцию `builder`):

```
1 my $app = sub {
2     my $env = shift;
3     my $req = Plack::Request->new($env);
4     my $res = $req->new_response(200);
5     $res->body('body');
6     return $res->finalize();
7 };
```

Из кода видно, что скаляр `$app` содержит в себе ссылку на функцию, которая возвращает валидный `PSGI`-ответ (ссылку на массив). Таким образом — это ссылка на функцию, которая возвращает ссылку на массив. Здесь можно добавить асинхронность, но дела из этого не выйдет, ведь исполняемый процесс будет блокироваться.

PSGI-приложение, которое является ссылкой на функцию, которая возвращает ссылку на массив, должно выполниться до конца, а только затем освободить поток исполнения.

Естественно, этот код будет работать правильно на любом PSGI -сервере, т.к. он синхронный. Любой асинхронный сервер умеет выполнять синхронный код, но синхронный сервер асинхронный код исполнять не может. Код, приведенный выше, является синхронным. В прошлой статье мы немного касались такого PSGI-сервера, как Twiggy. Рекомендую установить его, если его у вас еще нет. Это можно сделать несколькими способами. При помощи cpan (`cpan install Twiggy`), при помощи cpanm (`cpanm Twiggy`), или же взять на github.

Twiggy

Twiggy — асинхронный сервер. Автор у Twiggy и Starman один и тот же — @miyagawa.

Про Twiggy @miyagawa говорит следующее:

> PSGI/Plack HTTP-сервер, базирующийся на AnyEvent.

Twiggy — супермодель из 60-х, которая, как многие считают, положила начало моде на “худышек”, а т.к. сервер очень “легкий”, “тонкий”, “маленький”, то название было выбрано не случайно.

Отложенный ответ

PSGI-приложение с отложенным ответом представлено в документации следующим образом:

```
1 my $app = sub {
2     my $env = shift;
3     return sub {
4         my $responder = shift;
```

```
5
6           fetch_content_from_server(sub {
7               my $content = shift;
8               $responder->([ 200, $headers, [ $content ] ])
                    ;
9           });
10      };
11  };
```

Разберемся, как это работает, чтобы понять, как это использовать дальше и написать свое приложение, работающее с отложенным ответом.

Приложение является ссылкой на функцию, которая возвращает функцию, которая будет выполнена после выполнения некоторых условий (callback). В результате приложение является ссылкой на функцию, которая возвращает **ссылку на функцию**. Вот и все, что надо понимать. Сервер, если установлена переменная окружения `PSGI` "psgi.streaming", будет пытаться выполнить эту операцию в неблокирующем режиме, т.е. асинхронно.

Так как же это работает?

Если выполнять подобное приложение на `Starman`, то разницы не будет, но если мы будем использовать отложенный ответ на асинхронном сервере, то процесс исполнения будет выглядеть следующим образом.

- Сервер получает запрос.
- Сервер запрашивает данные откуда-нибудь, откуда они идут длительное время (функция `fetch_content_from_server`).
- Затем, пока ожидает ответа, он **может** принимать еще запросы.

Если бы модель была синхронной, то сервер бы не смог принять ни единого запроса, пока не отработал бы предыдущий.

Напишем приложение, используя механизм отложенного ответа. Приложение будет выглядеть следующим образом:

```
use strict;
use Plack;
my $app = sub {
    my $env = shift;
    return sub {
        my $responder = shift;
        my $body = "ok\n";
        $responder->([ 200, [], [ $body ] ]);
    }
}
```

А теперь запустим приложение как при помощи `Starman`, так и при помощи `Twiggy`.

Команда на запуск при помощи `Starman` у нас не меняется и выглядит следующим образом:

```
starman --port 8080 app.psgi
```

Для запуска при помощи `Twiggy`:

```
twiggy --port 8081 app.psgi
```

Теперь сделаем запрос сначала к одному серверу, затем к другому.

Запрос к Starman:

```
curl localhost:8080/
ok
```

Запрос к Twiggy:

```
curl localhost:8081/
ok
```

Пока-что отличий никаких, и сервера отрабатывают одинаково.

А теперь проведем простой эксперимент с `Twiggy` и `Starman`. Представим, что нам надо написать приложение, которое будет что-то выполнять по запросу клиента, а после завершения операции отчитываться о выполненной работе. Но, т.к. клиента нам держать не нужно, воспользуемся для имитации выполнения чего-либо `AnyEvent->timer()` для `Twiggy`, `sleep 5` для `Starman`. Вообще, `sleep` здесь не самый лучший вариант, но другого у нас нет, т.к.

код с `AnyEvent` в `Starman` работать не будет.

Итак, реализуем два варианта.

Блокирующий:

```
use strict;
sub {
    my $env = shift;
    return sub {
        my $responder = shift;
        sleep 5;
        warn 'Hi';
        $responder->([ 200, [ 'Content-Type' => 'text/
           json'], [ 'Hi' ] ]);
    }
}
```

Как бы мы его не запускали, хоть при помощи `Starman`, хоть при помощи `Twiggy`, результат будет всегда один. Запустим его, для начала, при помощи `Starman` следующей командой:

```
starman --port 8080 --workers=1 app.psgi
```

Внимание: для чистоты эксперимента надо использовать Starman с одним рабочим процессом.

Обращаясь к серверу из разных терминалов одновременно, мы можем видеть, как это приложение исполняется. Сначала worker возьмет первый запрос и начнет его исполнять. В этот момент второй запрос будет стоять в очереди. Как только первый запрос полностью выполнится, сервер начнет обрабатывать следующий запрос.

Суммарно два запроса будут выполняться приблизительно 10 секунд (второй запускается на обработку только после первого). Если запроса будет 3, то примерное время выполнения будет 18 секунд. Именно эта ситуация называется блокировкой.

Асинхронный код

Если запустить предыдущий пример на исполнение при помощи `Twiggy`, результат будет такой же точно. Сейчас может возникнуть вопрос, зачем нужен асинхронный сервер, если он блокируется и `Starman` работает точно также.

Дело в том, что для того, чтобы что-то работало асинхронно, необходим механизм, который будет обеспечивать асинхронность, цикл событий (event loop), например.

`Twiggy` построена вокруг `AnyEvent`-механизма, который запускается при старте сервера. Мы можем им пользоваться сразу же после старта сервера. Возможно использовать и `Coro`, статья по которому тоже обязательно будет.

Теперь напишем код, который **не будет** работать со `Starman`, и получим готовое асинхронное приложение.

Приведем в порядок код и сделаем приложение асинхронным. В результате у нас должно получиться нечто следующего вида:

```
sub {
    my $env = shift;
    return sub {
        my $respond = shift;
        $env->{timer} = AnyEvent->timer(
            after => 5,
            cb    => sub {
                warn 'Hi' . time() . "\n";
                $respond->([200, [], ['Hi' . time() . "\n
                   "]]);
            }
        );
    }
}
```

Стоит напомнить, что блокировки будут **всегда**, от написания кода зависит то, где они будут. Чем меньше времени сервер будет заблокирован, тем лучше.

Как это работает?

В первую очередь запускается таймер. Основной момент заключается в том, что в `return sub {...}` необходимо присваивать объект-наблюдатель (`AnyEvent->timer(...)`) переменной, которая была объявлена **до** `return sub {...}`, либо же использовать `condvar`. Иначе таймер никогда не будет выполнен, т.к. `AnyEvent` посчитает, что функция выполнена и ничего делать не надо. По истечению таймера возникает событие, функция выполняется, и сервер возвращает результат. Если сделать из разных терминалов, например, три запроса, то они будут все выполняться асинхронно, а по срабатыванию события таймера будет возвращен ответ. Но здесь самое главное то, что блокировки не происходит. Об этом свидетельствует результат трех запросов, выполненных с разных терминалов, вывод STDERR:

```
twiggy --port 8080 app.psgi
Hi1372613810
Hi1372613811
Hi1372613812
```

Запуск сервера был осуществлен следующей командой:

```
twiggy --port 8080 app.psgi
```

А запросы выполнялись при помощи curl:

```
curl localhost:8080
```

Напомним, что preforking-сервер в классическом виде синхронен. Одновременность запросов обрабатывается при помощи определенного количества worker'ов. Т.е. если запустить предыдущий синхронный код:

```
use strict;
sub {
    my $env = shift;
    return sub {
        my $responder = shift;
        sleep 5;
        warn 'Hi';
        $responder->([ 200, [ 'Content-Type' => 'text/
           json'], [ 'Hi' ] ]);
    }
}
```

с несколькими worker, то получится, что два запроса будут выполняться одновременно. Но тут дело не в асинхронности, а в том, что каждый запрос обрабатывается своим рабочим процессом. Так работает Starman, preforking PSGI server.

Возьмем асинхронный пример:

```
sub {
    my $env = shift;
    return sub {
        my $respond = shift;
        $env->{timer} = AnyEvent->timer(
            after => 5,
            cb    => sub {
                warn 'Hi' . time() . "\n";
                $respond->([200, [], ['Hi' . time() . "\n
                   "]]);
            }
        );
    }
}
```

Запуск произведем следующей командой:

```
twiggy --port 8080 app.psgi
```

и повторим эксперимент с двумя одновременными запросами.

Действительно, Twiggy работает одним процессом, однако ничто не мешает ей выполнять в процессе ожидания другие полезные действия. Это и есть асинхронность.

Данный пример был использован исключительно ради демонстрации того, как можно использовать отложенный ответ. Для лучшего понимания принципов работы Twiggy рекомендуется ознакомиться со статьями, посвященными AnyEvent в предыдущих номерах журнала (“Все, что вы хотели знать про AnyEvent, но боялись спросить” и “AnyEvent и fork”).

На данный момент существует довольно большое количество PSGI -серверов, которые поддерживают циклы событий. А именно:

```
* `Feersum` --- асинхронный XS-сервер с нереальной
   производительностью,
  базируется на `EV`.
```

```
* `Twiggy` ——— асинхронный сервер, базируется на `AnyEvent`.
* `Twiggy::TLS` ——— та же самая `Twiggy`, но с поддержкой ssl.
* `Twiggy::Prefork` ——— та же самая `Twiggy`, но с workers.
* `Monoceros` ——— молодой сервер, гибридный, имеет в себе как синхронную,
  так и асинхронную части.
* `Corona` ——— асинхронный сервер, базируется на `Coro`.
```

Выводы

У любой технологии есть свои нюансы. Решать, какой подход использовать, нужно, опираясь на данные по каждой конкретной задаче, но не использовать везде асинхронный подход, потому что модно.

В следующей статье мы подробнее остановимся на `Feersum` и начнем разбираться с `PSGI` deployment.

■ *Дмитрий Шаматрин*

4. Многопроцессовый сервер на AnyEvent

Рассмотрена универсальная параллельная обработка данных путем запуска нескольких процессов под управлением AnyEvent.

Очень часто приходится обрабатывать большое количество каких-то задач или данных. И зачастую данные эти удобно обрабатывать в несколько потоков, дабы полностью утилизировать всю мощь современных многопроцессорных компьютеров. Самый простой путь для осуществления этой цели — многопоточное приложение. Но у него есть ограничения, например — оно тяжело параллелится на несколько процессорных ядер (хотя в целом это решаемый вопрос). В данной статье рассмотрим решение такой задачи с помощью событийно-ориентированного приложения.

Итак, как будет выглядеть приложение? Напишем сервер, который будет работать с пулом процессов. Он будет создавать процессы (не обязательно перловые, это могут быть и программы на C/Java/Python/Ruby), перехватывать их STDOUT/STDERR и заниматься прочими мелкими задачами, которыми и положено заниматься такому серверу.

Создаем каркас приложения:

```
#!/usr/bin/env perl
use v5.12;
use AnyEvent;
use AnyEvent::Handle;
use IPC::Open3;

my $cv = AE::cv;

my $alive_timer = AnyEvent->timer(
    after    => 60,
    interval => 60,
    cb       => sub {
        say "Server alive";
    }
);

$cv->recv;
```

Получилось приложение, которое раз в минуту пишет в консоль о том, что оно все еще работает. Пример довольно простой, перейдем к более сложным вещам.

Итак, первым делом понадобится пул, где хранится вся информация о запущенных процессах. Это вынесем в отдельный пакет, дабы не загромождать код самого событийного механизма и улучшить читаемость.

```
1  package My::Pool;
2
3  my %commands_pool = ();
4  my %process_pool  = ();
5
6  sub new {
7      my $self = {};
8      bless $self;
9  }
10
11 sub update_pool {
12     my $self = shift;
13
14     return {
15         command_id => int(rand(10000)),
16         command    => 'test.pl',
17         options    => int(rand(100))
18     };
19 }
20
21 sub store_process {
22     my $self = shift;
23     my $proc = shift;
24
25     $process_pool{$proc->{'pid'}} = $proc;
26 }
27
28 sub process_done {
29     my $self      = shift;
30     my $pid       = shift;
31     my $exit_code = shift;
32
33     my $comm_id = $process_pool{$pid}->{'command_id'};
34
35     $command_pool{$comm_id}->{'exit_code'} = $exit_code;
36     delete($process_pool{$pid});
37 }
38
```

```
39 sub save_command {
40     my $self    = shift;
41     my $comm_id = shift;
42 
43     ...
44 
45     delete($command_pool{$comm_id});
46 
47     return 1;
48 }
49 
50 1;
```

Получилась простейшая обвязка для обработки пула, теперь настало время запускать комманды. Модифицируем каркас следующим образом:

```
1 my $Pool = My::Pool->new();
2 
3 my $cv = AE::cv;
4 
5 my $alive_timer = AnyEvent->timer(
6     after    => 60,
7     interval => 60,
8     cb       => sub {
9         say "Server alive";
10    }
11 );
```

Создаем новый таймер для переодического опроса новых данных:

```
1 my $process_time = AnyEvent->timer(
2     after    => 1,
3     interval => 5,
4     cb       => sub {
```

Запрашиваем новую комманду для выполнения:

```
1         my $command = $Poll->update_pool();
```

С помощью `IPC::Open3` создаем фоновый процесс. 0 на третьем месте в списке аргументов говорит, что `STDERR` надо перенаправить в `STDOUT`. Все дальнейшие опции аналогичны `system`:

```
1         my ($chld_out, $chld_in);
2         my $pid = open3($chld_in, $chld_out, 0, $command
             ->{'command'},
```

```
3             'options=' . $command->{'options'});
```

Создаем объект `AnyEvent::Handler` для обработки информации, поступающей от процесса:

```
1         my $hdl;
2         $hdl = AnyEvent::Handle->new(
3             fh      => \*$chld_out,
4             pid     => $pid,
5             oper_id => $oper_id,
6             on_eof  => sub {
7                 my ($hdl) = @_;
```

Тут можно что-то сделать при закрытии канала. Например — уничтожить хэндлер `$hdl->destroy`.

```
1             },
```

В данном случае каждая строка от дочернего процесса будет выводиться на экран:

```
1             on_read => sub {
2                 my ($hdl) = @_;
3                 $hdl->push_read(
4                     line => sub {
5                         my ($hdl, $line) = @_;
6                         say "process [$pid] got line <
                              $line>"
7                     );
8                 }
9             );
10        }
11    );
```

Создаем объект, следящий за процессом и получающий его код завершения:

```
1     my $w = AnyEvent->child(
2         pid => $pid,
3         cb  => sub {
4             my ($pid, $status) = @_;
5             $status = $status >> 8;
```

Выводим на экран информацию о завершении процесса и вызываем его обработчик в пуле:

```
            say "process [$pid] done with code [$status]"
               ;
            $Pool->process_done($pid, $status);
        }
    );
```

теперь сохраним все в пуле процессов:

```
    $Pool->store_process(
        {pid => $pid, handler => $hdl, watcher => $w}
    );
});

$cv->recv;
```

На данный момент этот сервер успешно запускает комманды, которые получает из пула и выводит логи их работы на экран. Теперь можно наращивать функционал, например, добавить уничтожение зависших процессов и прочую логику. При тестировании на релаьных примерах эта конструкция (обернутая примерно в 10 КБ кода с логикой) прекрасно держит 50 одновременно запущенных процессов + еще десяток удаленных клиентов, обменивающихся с ними информацией через сокеты. Больше процессов не держит — у сервера ресурсов не хватает, процессы довольно тяжелые.

Несколько недостатков представленного решения:

- Во первых, нужно внимательно следить за блокировками в коде.
- Во вторых — из четырех Windows-машин заработало только на одной. Может это ошибка `IPC::OpenX`. Т.е. запускаться и рождать процессы модуль будет, но данных из `STDOUT` не будет возвращать. Но под Windows можно использовать следующий прием — заменяем `my $pid = open3(...);` на `my $pid = system(1, $command, '> $log_file 2>$1');` и дальше навешиваем `AE::Handler` на этот `$log_file`.
- Буферизация `STDOUT`. Т.е. лог работы процесса возвращается практически одновременно с его завершением. Но лог `STDERR` — появится мгновенно т.к. на нем нет буферизации. Стоит обратить на это внимание.

■ *Денис Федосеев*

5. Обзор CPAN за июнь 2013 г.

Рубрика с обзором интересных новинок CPAN за прошедший месяц.

Статистика

- Новых дистрибутивов — 226
- Новых выпусков — 876

Новые модули

- App::aki Консольная утилита для обработки веб-контента. Например, поиск значения в JSON-документе.
- Net::WebSocket::Server Вебсокет-сервер с минимум зависимостей.
- Docopt Perl-порт docopt.org, позволяющий простым документированием опций автоматически генерировать парсер для их обработки.
- Carp::Reply Во время использования этого модуля при возникновении исключения управление передается оболочке Reply.
- Thrall Plack-сервер, использующий `threads`, правильно работает на Windows.
- App::pmdeps Утилита для получения зависимостей модуля на CPAN. Сообщает также зависимости от core-модулей.

Обновлённые модули

- Plack 1.0028 Удалены все XS-зависимости, что существенно упрощает развертывание приложения.

- Rex 0.42.3 Утилита для удобного развертывания приложений и настройки серверов. Исправлено большое количество ошибок.
- Pinto 0.42.3 Собственный CPAN из коробки. Исправления мелких ошибок. Добавление документации с перечислением всех тех, кто помог с финансированием (об этом мы писали в прошлом выпуске журнала), а также спрятанное пасхальное яйцо!
- lib::remote 0.11 Модуль позволяет подключать библиотеки с удаленных серверов без их локальной установки. Особенное внимание привлекает FAQ модуля.
- App::Cronjob 1.2000001 Утилита для обертки скриптов для их безопасного использования в cron. Позволяет управлять отправкой почты, блокированием параллельного запуска. В этом релизе больших изменений нет.
- Routes::Tiny 0.11 Добавлена возможность вложенных маршрутов.
- HTML::Copy 1.31 Модуль для копирования HTML с сохранением ссылок. После пятилетнего перерыва исправлены ошибки, связанные с битыми ссылками, а также исправлены тесты для perl-5.18.
- HTML::Tree 5.03 Использование “слабых” ссылок для избежания утечек памяти.
- App::perlbrew 0.64 Новая команда `install-multiple` для одновременной установки нескольких версий perl.
- App::cpanminus 1.6922 Довольно много исправлений ошибок. Файл build.log теперь создается в директории дистрибутива (с символической ссылкой на предыдущее местоположение).
- IO::Socket::SSL 1.94 Исправление сообщения об ошибке версии `Net::SSLeay` при установке, когда этот модуль вообще не установлен.

■ *Вячеслав Тихановский*

6. Интервью с брайаном ди фоем про будущее. Часть 1

25 и 26 мая в Варшаве прошел первый польский перл-воркшоп. На него приехал и брайан ди фой (brian d foy), который, по его словам, всегда старается побывать на мероприятии в городе, где он еще не был. Брайан уделил нашему журналу час и ответил на вопросы о том, каким он видит будущее: будущее книг, перла, его синтаксиса и сообщества (публикуется с сокращениями).

Будущее книг

— Несколько дней назад в своем блоге вы упомянули о том, что издательство O'Reilly переходит на новую издательскую платформу. Что это такое?

— Да. Я пользуюсь этой новой платформой для книги Mastering Perl. Я не все еще там понимаю, я только учусь. Они отказались от Subversion в пользу Git, что очень хорошо, потому что я все равно пользуюсь гитом и мне надо было брать оттуда свои тексты и коммитить к ним в Subversion. Сейчас этот шаг уже не нужен. А еще теперь они могут без труда взять мои файлы и опубликовать их на сайте, чтобы люди видели, что сейчас происходит. Поскольку я делаю это уже довольно давно, вы прямо сейчас можете следить за развитием Mastering Perl. В случае с Mastering Perl нас спонсирует OSCON, поэтому ее можно читать бесплатно. Мне кажется, что возможность посмотреть книгу до публикации очень важна.

С книгой Learning Perl все просто, мы все более или менее знаем, что там должно быть. Никто не скажет мне: «О, а вот есть еще такая штука, вы знаете о ней?», то есть существуют массивы, скаляры и хеши, все просто. А вот для Mastering Perl кто-то может спросить, знаком ли я с каким-нибудь странным эзотерическим случаем в какой-то из тем. Например, что есть сериализатор Sereal, который может заменить Storable. Это действительно сложный момент, потому что поскольку я уже пользуюсь перлом так давно и привык делать что-то определенным способом, то мне не всегда интересно, что появи-

лось нового: старое вроде работает и способно решить мои задачи.

Так вот, теперь, когда содержимое Mastering Perl находится в репозитории, то как только я в него коммичу, автоматически генерируются HTML-страницы, которые выглядят в стиле O'Reilly; люди могут их читать и комментировать, а я могу учесть замечания в книге, и уже на следующий день это будет доступно.

— Вы ожидаете, что люди сами захотят коммитить в ваш репозиторий с книгой?

— Да нет, наверное и нет такого доступа к гиту. Но даже если бы он был, я могу представить, что найдутся люди, да я и знаю нескольких таких, кто захочет сделать pull request, но это будет весьма забавно. Проблема с коммитами или попытками пропатчить книгу в том, что какие-то изменения просто не смогут сработать. Патчить код очень легко. А с книгой я могу решить, что мне не нравится что-то в таком-то разделе или я захочу полностью сместить фокус одной из глав и полностью изменю порядок изложения и допишу новый контент.

Хороший пример такого взаимодействия — спелчекеры. Я довольно неплохо пишу, но хуже набираю, так что могу что-то пропустить, а компьютер начинает предполагать, какое слово я имел в виду, и это может оказаться совсем не тем, что я хотел написать. Например, я набираю CPAN на маке, а автокоррекция хочет сделать замену CLAN, поскольку не знает, что такое CPAN, но знает, что такое к-л-а-н. И если я не посмотрю в этот момент на экран, то вместо CPAN останется CLAN. Кто-то это заметит и захочет прислать мне патч, это хорошо. Но до этого я могу полностью изменить тот абзац, и патч попросту не подойдет.

Я получаю большой объем обратной связи и могу сказать: лучше просто напишите мне письмо. Даже если и с патчем в итоге получится разобраться, то вряд ли O'Reilly дадут доступ к гиту.

— Недавно вы сообщили, что впервые подписали электронную книгу. Какие чувства это вызывает?

— Да. Это интересно, на прошлой неделе я сделал такое впервые.

Кто-то попросил меня подписать PDF, и я ответил: «OK, подпишу». Приложение Preview на маке может сделать так, что можно оставить подпись на обычной бумаге, показать ее в камеру и разместить ее в нужном месте в PDF. Это замечательно и в итоге позволяет отказаться от бумаги. Знаете, банки, например, требуют все распечатать, потом подписать, отсканировать и отправить обратно по электронной почте или по факсу.

В общем, я теперь так могу подписывать книги и добавлять небольшой текст. Хотелось бы, чтобы это можно было делать и в электронных книгах. С электронным контентом можно делать так много всего, что мы еще не делаем, причем намного лучше и проще. Надеюсь, мы еще застанем это.

— А вообще, вы считаете продажу электронных книг вместо бумажных приемлемой ситуацией?

— Это интересный вопрос, «электронные книги против бумажных». Я не знаю, принадлежу ли я к тем, кто любит бумажные книги. У таких книг лучше разрешение. Их можно сложить стопкой или разложить на столе.

Самое интересное, если предлагать электронную и бумажную книги вместе, то это увеличивает продажи. То есть, если сказать, что можно купить, например, бумажный Mastering Perl — и O'Reilly это уже делало в качестве эксперимента, — и за дополнительный доллар получить и электронную версию, то мы получаем намного больше продаж. То есть люди не хотели конкретно электронную или бумажную версию. Им нужна была просто книга.

Надеюсь, что бумажные книги не исчезнут, я не так сильно люблю электронные книги. Мне сложно запомнить, где в них что находится. В бумажной книге я знаю, например, что видел что-то на левой или на правой станице ближе к середине. Я могу быстро пролистать книгу, потому что примерно помню, как выглядела страница. Я обычно не помню все, что читаю. В смысле, конкретные факты или точное содержание. Помню, что автор что-то про это сказал и это было где-то здесь в книге. Таким способом я много чего могу найти. Хотя, если человек привык к электронной книге, то, возможно, и не развил в себе такую способность.

Вот что меня беспокоит: чем больше у нас технологий, тем хуже мы представляем контент. Возьмем фильм на пленке с хорошим разрешением. Из этого мы делаем несколько вариантов. Сейчас есть потоковое видео, но оно может идти с паузами, и удовольствие от просмотра теряется. А потом мы можем попытаться посмотреть то же на айфоне или еще где. Надеюсь, что с книгами этого не произойдет. Мы начали с того, что можно напечатать и с мелких-мелких точек на странице, 1200 точек на дюйм или даже больше. А сейчас мы читаем на устройствах с 72 точками на дюйм. Ну, на ретине побольше, но все равно это далеко от разрешения, доступного на бумаге.

Так что я не знаю. Я не книжный коллекционер и не любитель книжного запаха или золотых переплетов, но, в тоже время, не хочу полностью переходить на электронные книги.

— Вы стараетесь обновлять свои книги с выходом новых версий перла. Это для вас хобби, развлечение или тяжелая работа?

— Каждый раз, когда выходит новая версия перла, есть опасение, что предыдущие издания Learning Perl, Intermediate Perl и Programming Perl, которые мы написали, устареют. Сейчас, в компьютерном мире, книга, которая старше трех лет, совершенно неактуальна. Она больше ни для чего не годна.

С Learning Perl все хорошо, потому что по сути там в основном Perl 4. Ну то есть все эти скаляры, массивы и хеши и большая часть стандартной библиотеки. Но ведь добавляются и новые фичи. Последнее издание описывает Perl 5.14, а в Perl 5.16 или 5.18 на самом деле не так много того, без чего нельзя обойтись в этой книге. А вот с Intermediate Perl немного иначе, потому что там многое изменилось. Теперь, например, функции push, pop, shift, и unshift понимают ссылки на массивы и хеши. Операторы могут получить ссылку и выяснить сами: «О, а это ведь хеш, а с ним я умею работать». То же самое произошло с each, который теперь работает с массивом. Так что приходится править такие вещи.

Но большинство обновляют перл очень, очень медленно. Так что если написать книгу про Perl 5.14, то и шесть лет спустя книга окажется полезной, поскольку кто-то только что перешел на Perl 5.14.

Я сделал небольшой опрос среди какого-то числа пользователей. Большинство никогда не апгрейдятся. Они пользуются тем, что есть на системе, и их это устраивает. Многие из моих клиентов до сих пор на 5.8. Мне не особо это нравится, то такова правда.

Поэтому я думаю, что большинство из тех, кто покупает книги, не обращают внимания на конкретную версию перла. Они не смотрят на обложку и не думают, что книгу надо купить, потому что она про новую версию. До сих пор покупают третье, четвертое и пятое издания Learning Perl. Я не знаю, где люди берут эти книги, но они их покупают: я получаю с них пару долларов в квартал.

Книга Programming Perl — совершенно другая история. Эта книга последний раз издалась, вроде, в 2000 году. Десять лет назад, да больше десяти. Переписать последнее четвертое издание заняло около двух лет. Том (Tom Christiansen) проделал значительную работу по обновлению всего, что связано с юникодом. На обложке есть и мое имя, но я делал очень скучную несложную часть в конце книги. Том занимался первой большей частью, которая намного сложнее. И, разумеется, Ларри наблюдал за процессом и говорил, что хочет то-то так-то и так-то.

Было бы неплохо, чтобы эта книга была для каждой большой версии, но постойте, столько там страниц, около 1600? Мы не будем ее обновлять каждый год с каждой новой версией, иначе это превратится в непрекращающуюся работу над книгой, и никого не сделает богатым. При этом людям-то все равно. Я надеюсь, что каждый перлпрограммист купил Programming Perl. Просто, чтобы книга была на полке. Но, честно говоря, не будь я сам одним из авторов, я бы ее не купил. Я знаю перл, я могу читать документацию, могу посмотреть в онлайне, могу задавать вопросы. Мне не нужна эта огромная книга, разве что подпереть ею дверь или уронить кому-то на ногу.

Я стараюсь не отрываться от реальности: очень трудно что-то продать, если существенная часть работы тех же авторов попадает в документацию. Но книга эта замечательная. То, что сделал Том, просто удивительно, и просто великолепно, что тот же человек разместит это бесплатно в документации.

Mastering Perl не обновлялся уже долгое время, но эта книга на са-

мом деле не зависит от версии. Она больше про то, как думать о том, что возможно сделать на перле вообще, а не с конкретной его версией. Думаю, что и книга Effective Perl Programming такая же. Это отличная книга, которая привлекает заслуженное внимание. Когда Джозеф Хол (Joseph Hall) опубликовал эту книгу, мне она казалась лучшей, когда либо написанной про перл. И если сегодня мне потребуется составить список книг, я думаю, что она заняла бы одну из верхних позиций.

Другая книга, которую стоит обновлять, это Perl Cookbook. Было бы очень интересно увидеть, как можно применить к конкретным задачам все новые фичи, которые появились в 5.10.

Будущее smart match

— В 5.10 появился оператор smart match (~~), у которого непростая судьба. Что вы о нем думаете, и должен ли он вообще остаться в языке — как есть или измененный?

— Оператор smart match, появившийся в 5.10, — действительно интересный случай. Я думаю, надеюсь, что мы усвоили урок о том, что такие экспериментальные возможности надо помечать экспериментальными. Это и случилось в 5.18.

Мне хотелось использовать smart match, но я подумал, что он может довольно скоро исчезнуть из перла, так что лучше я лучше запишу это иначе, хоть и более сложным образом. Мне надо сделать простую вещь, просто сопоставить с регулярным выражением набор ключей хеша, и smart match сможет это сделать: с одной стороны оператора регулярное выражение, с другой — хеш, и все.

Я всегда задаюсь вопросом, облегчает ли это жизнь перл-программистам. Следует ли это принципу о том, что простые вещи должны быть простыми и обыденными, а сложные по крайней мере возможными. Лично мне кажется, что я бы хотел, чтобы так было со smart match, но не думаю, что это получится, потому что у нас нет консенсуса относительно того, что там должно остаться.

Это означает, что мне придется вернуться и обновить Learning Perl, удалив оттуда главу про smart matching. Она появилась с 5.10.1 и, вроде, до сих пор актуальна, но не думаю, что мы можем учить людей тому, что исчезнет через год в 5.20.

— Но в 15-й главе рассказывается еще и про `given` и `when`. Их тоже придется исключить?

— Да. (Я удивлен тому, что вы знаете номер главы.) Да, в 15-й главе книги Learning Perl (по крайней мере в 6-м издании, не знаю, поменяются ли главы в будущем) есть и smart match, и `given`, и `when`. Работа `given` была несколько проблематичной, потому что происходили странные вещи с лексической переменной `$_`, что вызывало много проблем, например, внутри `map` и `grep` или внутри `given`. Но мне кажется это уже исправлено в 5.18, думаю, что `given` отказался от этой магии с `$_` и теперь делает то же, что делают `for` и `foreach`.

Так что в этой главе много всего, что придется убрать, если отказаться от оператора smart match. Но `given` и `when` без smart match окажутся не намного более мощными, чем просто набор нескольких `if`.

Во второй части интервью мы поговорим про PM-группы (а брайан в свое время создал первую такую группу) и о том, что он думает про Perl 7.

■ *Андрей Шитов*

7. 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) 2, 2) 1, 3) 3, 4) 2, 5) 2, 6) 4, 7) 2, 8) 3, 9) 4, 10) 2.

1. В каком городе пройдет конференция YAPC::Europe 2014?

    1. Гранада (Испания)
    2. Клуж-Напока (Румыния)
    3. Перл (Германия)
    4. София (Болгария)

2. Что думает brian d foy про Perl 7?

    1. Срочно переименовать
    2. Перлу отказать вообще сразу
    3. Подождать еще пару лет и переименовать
    4. Прекратить об этом говорить

3. Кто автор выражения «Перлу отказать вообще сразу»?

    1. Ларри Уолл
    2. Ларри Кинг
    3. Алена Владимирская
    4. Алена Свиридова

4. Сколько книг про Perl в коллекции Wendy van Dijk?

    1. 84
    2. 184
    3. 284
    4. 384

5. Какой юникодный символ Ларри Уолл предлагал использовать в сообщениях об ошибке?

    1. \x{23CF} EJECT SYMBOL
    2. \x{26A0} WARNING SIGN
    3. \x{1F4A9} PILE OF POO
    4. \x{1F42A} DROMEDARY CAMEL

6. Какого модуля еще нет на CPAN?

    1. Plack::Middleware::iPhone
    2. Plack::Middleware::JSON
    3. Plack::Middleware::QRCode
    4. Plack::Middleware::XSLT

7. В какой гостинице остановится больше всего участников конференции YAPC::Europe 2013?

    1. «Днипро»
    2. «Крещатик City Center»
    3. «Лыбидь»
    4. «Украина»

8. Какой редактор кода был разработан в расчете на начинающих?

    1. Eclipse
    2. Kephra
    3. Komodo Edit
    4. Padre

9. Какая конструкция является экспериментальной?

    1. ==
    2. . .
    3. ! !
    4. ~~

10. Чем определяется стандарт языка Perl 5?

    1. Спецификацией в формате EBNF

2. Набором тестов из дистрибутива
3. Реализацией компилятора
4. Описанием в книге Programming Perl

■ *Андрей Шитов*