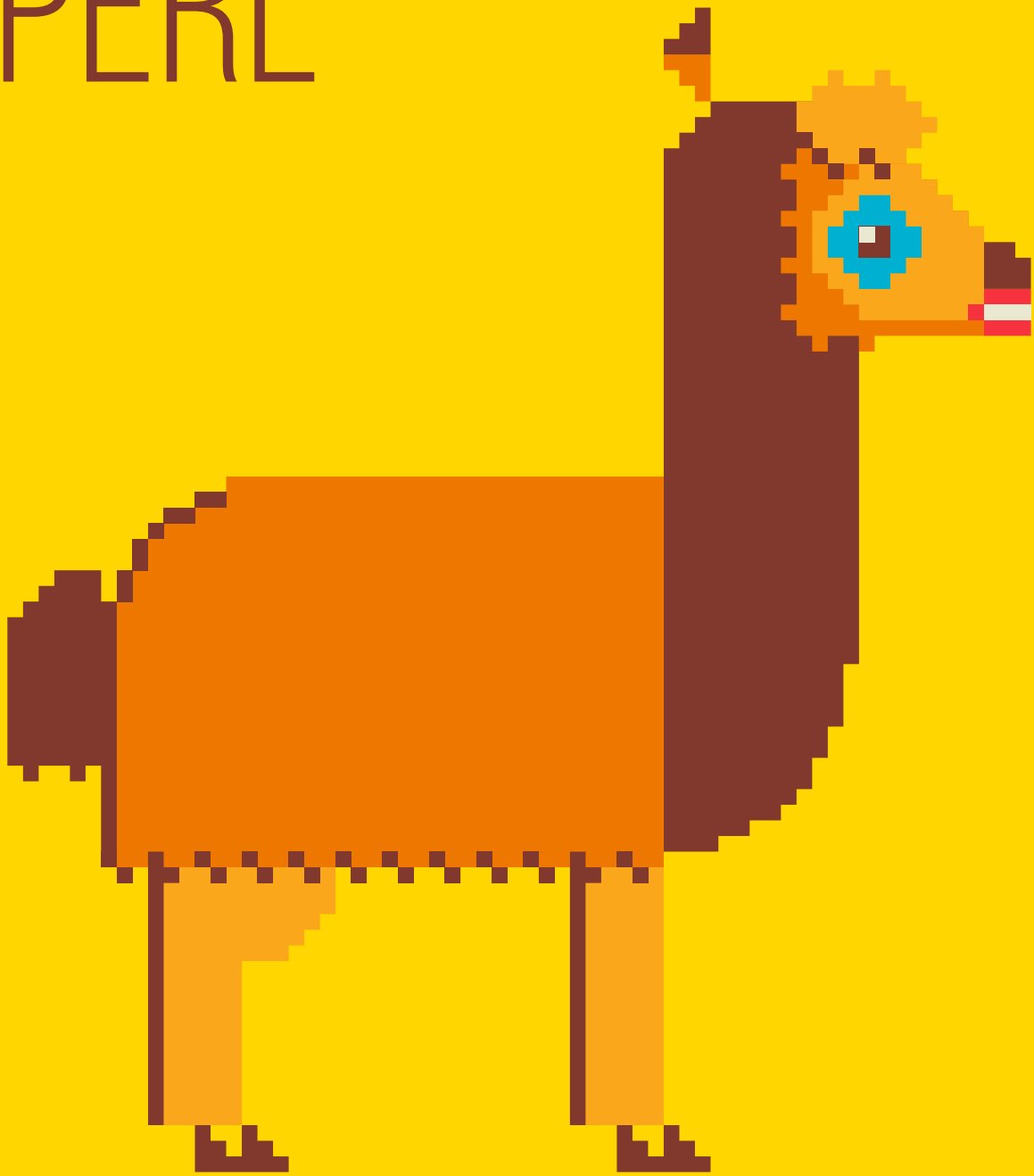
PRAGMATIC PERL

24



02/2015

pragmaticperl.com

Pragmatic Perl 24

pragmaticperl.com

Выпуск 24. Февраль 2015

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

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

Авторы статей: Олеся Кузьмина, Владимир Леттиев, Андрей Шитов

Обложка: Марко Иванык

Корректор: Андрей Шитов

Выпускающий редактор: Вячеслав Тихановский

Ревизия: 2015-10-09 11:33

Оглавление

1. От редактора. Опрос и Форум

У нас появился форум! Исходный код доступен на GitHub.

Закончился опрос «Как сделать журнал лучше?». Спасибо всем, кто поучаствовал. Это нам очень пригодится. На основе ваших пожеланий был составлен список интересных тем. На этой же странице можно узнать, как оформлять статьи для публикации.

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

Приятного чтения.

■ *Вячеслав Тихановский*

2. Тестирование черного ящика

Рассмотрены особенности тестирования приложений в целом как черного ящика

Продолжаем цикл статей про тестирование: Тестирование в Perl. Лучшие практики, Тестирование в Perl. Практика, Тестирование с помощью Mock-объектов.

Юнит-тестирование не дает ответа на вопрос «Работает ли система целиком?», так как каждый тест пишется как можно независимее от других подсистем, широко используются Mock-объекты и подстановки. Тестирование производится тем, кто реализует требуемый функционал и подразумевает, что тесты некоторым образом все-таки завязаны на реализацию.

Как же проверить, что система работает? Обычно система запускается в тестовом режиме, где некоторые подсистемы представляют собой тестовые реализации, например, подключение к банку, обращение к другому сайту и прочее. Далее на запущенную систему запускаются тесты, которые никоим образом не знают, как внутренне устроена система и тестируют только то, что доступно извне.

Для примера рассмотрим, как тестируется подписка на наш журнал. Подписка это важная подсистема, и тестировать ее вручную довольно утомительно, так как там несколько шагов, которые еще и разветвляются. Подписка должна обязательно работать, поэтому написание автоматических тестов и их выполнение перед каждым запуском в работу просто необходимы.

Настройка окружения

При запуске приложения указывается, что оно исполняется в тестовом режиме. В примере с журналом читается тестовый конфигурационный файл, где указаны путь к тестовой базе данных, настройки тестового почтового сервера и локальный URL.

С тестовой базой и локальным URL все понятно. Как же устроен тестовый почтовый сервер? Обычно используются `sendmail` или аналоги, в тестовом

же режиме отправляемое сообщение просто пишется во временный файл, который создается и проверятся во время теста. Это позволяет проверить не только факт отправления сообщений, но и правильность их формата, текста и т.д.

Так, например, выглядит вспомогательная функция для чтения почты:

```
sub _read_email {
    my $self = shift;

    local $/;
    open my $fh, '<', '/tmp/mailer.log' or die $!;
    my $email = <$fh>;
    close $fh;

    my ($body) = $email =~ m/\r?\n\r?\n(.*)/ms;
    $body = MIME::Base64::decode_base64($body);
    return Encode::decode('UTF-8', $body);
}
```

Запуск приложения

Запускать приложение вручную не всегда удобно, это можно автоматизировать. Так как журнал совместим с PSGI-интерфейсом, приложение можно запускать прямо из теста. Тестирование осуществляется с помощью `Test::WWW::Mechanize::PSGI`, поэтому создать агента довольно просто:

```
sub _build_ua {
    my $self = shift;

    my $app = PP->new;
    return Test::WWW::Mechanize::PSGI->new(app => $app->to_app);
}
```

Таким образом с помощью созданного объекта можно выполнять различные HTTP-запросы и проверять их результаты.

Тестирование

Вначале убедимся, что при заходе на путь `/newsletter` пользователь получает правильную страницу:

```
subtest 'shows subscribe page' => sub {
    my $ua = $self->_build_ua;

    $ua->get_ok('/newsletter');
    $ua->content_contains('Подписка');
};
```

В данном случае мы проверям то, что нам нужно. Не стоит проверять, что есть навигационная панель, правильные пути к стилям или картинкам. Дизайн часто меняется и тесты будут ломаться, хотя сама функциональность будет сохранена. Чтобы этого избежать, тестируем только то, что необходимо для правильной работы системы.

Обычно при заполнении форм стоит проверить, что срабатывает валидация. Не стоит проверять каждое поле, это уже задача юнит-тестирования. Главное проверить, что все работает целиком. Например, проверим, что при незаполнении полей получаем ошибку:

```
subtest 'shows validation errors' => sub {
    my $ua = $self->_build_ua;

    $ua->get('/subscribe');
    $ua->submit_form(fields => {});
    $ua->content_contains(q{Обязательно к заполнению});
};
```

Далее проверяем сценарии подписки. Чтобы протестировать подписку, необходимо проверить, что пользователь получает правильное сообщение после нажатия кнопки и ему отправляется правильный email с правильной ссылкой на подтвержение подписки. Далее необходимо проверить, что при клике на ссылку подтверждения пользователь получает правильный email.

Здесь покажем как проверить, что email был отправлен:

```
subtest 'sends confirmation email' => sub {
    my $ua = $self->_build_ua;

    $ua->get('/subscribe');
    $ua->submit_form(fields => {email => 'foo@bar.com'});
```

```
6
7       my $email = $self->_read_email;
8       like($email, qr{subscribe/[a-z0-9]+});
9       like($email, qr{unsubscribe/[a-z0-9]+});
10  };
```

Здесь используется вспомогательная функция для чтения отправленного email. Также проверяется наличие двух ссылок: одна для подтверждения, вторая для отписки, если письмо пришло по ошибке. Содержимое письма проверяется довольно общим образом, так как сообщения могут меняться. Главное, что есть правильные ссылки.

Для того, чтобы проверить отписывание от рассылки, необходимо каждый раз подписываться. Чтобы это автоматизировать, выделим отдельный метод для создания новой подписки:

```
1  sub _build_subscribed_ua {
2       my $ua = $self->_build_ua;
3
4       $ua->get('/subscribe');
5       $ua->submit_form(fields => {email => 'foo@bar.com'});
6
7       my $email = $self->_read_email;
8       my ($token) = $email =~ m{subscribe/([a-z0-9]+)};
9
10      $ua->get('/subscribe/' . $token);
11
12      unlink '/tmp/mailer.log';
13
14      return $ua;
15  }
```

Таким образом, мы получаем уже подписанный email и можем дальше проверять правильность удаления подписки без дублирования кода:

```
1  subtest 'sends confirmation email on unsubscribe' => sub {
2       my $ua = $self->_build_subscribed_ua;
3
4       $ua->get('/unsubscribe');
5       $ua->submit_form(fields => {email => 'foo@bar.com'});
6
7       my $email = $self->_read_email;
8       like($email, qr{unsubscribe/[a-z0-9]+});
9   };
```

Заключение

Таким образом осуществляется тестирование системы целиком.

Здесь мы не проверяем, что email добавляется в базу данных, ведь она недоступна извне, это все проверяется в юнит-тестах на соответствующие контроллеры. Иногда функциональные тесты могут пересекаться с юнит-тестами, например, в юнит-тестах проверяется, что при неправильном вводе данных срабатывает валидация. Однако при работающей системе перед контроллером и после него могут быть различные подсистемы, и их слаженную работу как раз и стоит проверить.

■ *Вячеслав Тихановский*

3. Реализация удаленного вызова процедур (RPC) в Perl с помощью Thrift

Рассмотрены основы работы со Thrift в Perl

Thrift — разработка Facebook, которая была открыта и передана организации Apache. Особенность данной реализации RPC — в бинарном протоколе и автоматической генерации кода для обработки сообщений для разных языков программирования. В качестве конфигурации RPC служит файл спецификации, где на C++-подобном синтаксисе описываются типы данных и сервисы с методами.

Где это может пригодиться? Thrift может быть полезен в проектах, где применяются разные языки программирования, когда необходимо наладить обмен данными и удаленно вызывать процедуры в отдельно запущенных сервисах. RPC также подойдет тогда, когда написание XS-кода усложняется, например, наличием сложной логики инициализации или использованием тредов.

Установка Thrift

На момент написания статьи Thrift присутствовал в Debian-дистрибутиве только в виде компилятора, без заголовков и библиотек для разработки. Поэтому собираем из исходников. Установка в систему не требуется, достаточно при генерации кода правильно указать пути к файлам.

Скачивание исходников

Исходники находятся на сайте Apache http://thrift.apache.org/download.

Компилирование

Вначале устанавливаем необходимые библиотеки для компилирования:

```
sudo apt-get install libboost-dev libboost-test-dev \
    libboost-program-options-dev libboost-system-dev \
    libboost-filesystem-dev libevent-dev automake libtool \
    flex bison pkg-config g++ libssl-dev
```

Для поддержки Perl необходимо установить модули `Bit::Vector` и `Class::Accessor` и при конфигурации указать опцию `--with-perl`.

```
./configure --with-perl
```

Чтобы собралась библиотека для C++, необходимо зайти в `lib/cpp` и запустить `make`:

```
cd lib/cpp
make
```

Если при компилировании возникает подобная ошибка:

```
lib/cpp/.libs/libthrift.so: undefined reference to `TLSv1_method'
lib/cpp/.libs/libthrift.so: undefined reference to `SSL_connect'
lib/cpp/.libs/libthrift.so: undefined reference to `sk_pop_free'
lib/cpp/.libs/libthrift.so: undefined reference to `BIO_ctrl'
...
```

Заходим в `lib/cpp/test` и вручную добавляем `-lssl`:

```
/bin/bash ../../../libtool  --tag=CXX   --mode=link g++ -Wall -
    Wextra -pedantic -g \
     -O2 -std=c++11 -L/lib64  -o Benchmark Benchmark.o
        libtestgencpp.la -lrt \
     -lpthread -lssl
```

Теперь в `lib/cpp/.libs` лежат библиотеки `libthrift.a` и `libthrift.so`.

Установка Perl-модулей:

Заходим в директорию `lib/perl` и устанавливаем модуль как обычно через `cpanm`:

```
cpanm .
```

На данный момент есть скомпилированный Thrift и установлены необходимые Perl-модули. Рассмотрим несколько примеров.

Примеры использования

Простейший ping-pong

Вначале для ознакомления с принципами написания Thrift-обвязок реализуем два Perl-сервиса (клиент и сервер), при котором клиент будет отправлять на сервер метод `ping` и ожидать ответа `pong`.

Thrift-спецификация может выглядеть следующим образом (файл `example1.thrift`):

```
namespace perl Example1

service Service {
   string ping()
}
```

`namespace` необходим для того, чтобы генерировать Perl-пакеты с префиксом `Example1`. Далее определяется сервис с методом `ping()`, возвращающим строку.

Теперь генерируем Perl-код с помощью Thrift-компилятора:

```
mkdir -p example1/lib
thrift-0.9.2/compiler/cpp/thrift --gen perl -out example1/lib
   example1.thrift
```

Теперь в директории `example1/lib` будет примерно следующая структура:

```
example1/
    lib/
        Example1/
            Constants.pm
            Service.pm
            Types.pm
```

Сгенерированный код нас не сильно интересует, в нем находится обработка Thrift-сообщений: упаковка, отправка, прием и т.п. Этот код не меняется и не требует доработки. Нам же необходимо реализовать клиент и сервер.

Пример реализации клиента:

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

```
2
3 use strict;
4 use warnings;
5
6 use Thrift;
7 use Thrift::BinaryProtocol;
8 use Thrift::Socket;
9 use Thrift::BufferedTransport;
10
11 use Example1::Service;
12
13 my $socket    = Thrift::Socket->new('localhost', 9090);
14 my $transport = Thrift::BufferedTransport->new($socket, 1024,
     1024);
15 my $protocol  = Thrift::BinaryProtocol->new($transport);
16 my $client    = Example1::ServiceClient->new($protocol);
17
18 $transport->open();
19
20 eval { print $client->ping(), "\n"; } or do {
21     my $e = $@;
22
23     print "Error: $e->{message}\n";
24 };
25
26 $transport->close();
```

Модули `Thrift::` это внутренние модули Thrift, а `Example1::` это модули, которые были сгенерированны ранее. В данном примере для транспорта используется сокет. Thrift поддерживает и другие транспорты, выбор которых зависит от задачи.

В данном коде вызываем метод `ping` и печатаем ответ от сервер. В случае ошибки печатаем и ее.

Пример реализации сервера:

```
1 #!/usr/bin/perl
2
3 use strict;
4 use warnings;
5
6 use Thrift::Socket;
7 use Thrift::Server;
8
9 use Example1::Service;
```

```
10
11 package ServiceHandler;
12 use base 'Example1::ServiceIf';
13
14 sub new {
15     my $class = shift;
16
17     my $self = {};
18     bless $self, $class;
19
20     return $self;
21 }
22
23 sub ping { 'pong' }
24
25 package main;
26
27 my $handler       = ServiceHandler->new;
28 my $processor     = Example1::ServiceProcessor->new($handler);
29 my $serversocket  = Thrift::ServerSocket->new(9090);
30 my $forkingserver = Thrift::ForkingServer->new($processor,
     $serversocket);
31
32 print "Starting the server...\n";
33 $forkingserver->serve();
```

Код сервера заключен в модуле `ServiceHandler`, который наследует сгенерированный интерфейс `Example::ServiceIf`. Этот интерфейс необходим для проверки соответствия реализации Thrift-спецификации. Если в `ServiceHandler` не будет реализован метод `ping`, то при вызове получим подобную ошибку:

```
1 implement interface at lib/Example1/Service.pm line 133.
```

Далее запускаем `./server.pl` и `./client.pl`. Клиент отправит `ping`, напечатает ответ и завершится:

```
1 pong
```

Обмен сложными структурами

Thrift поддерживает обмен и сложными структурами. Например, пусть клиент передает хеш, сервер его модифицирует и возвращает. Thrift-файл будет выглядить следующим образом:

```
namespace perl Example2

struct Hash {
    1: string foo = "bar"
}

service Service {
   Hash ping(1:Hash arg)
}
```

В реализации метода ping в качестве первого аргумента будет объект класса Example2::Hash. Меняем значение поля foo и возвращаем клиенту:

```
#!/usr/bin/perl

use strict;
use warnings;

use Thrift::Socket;
use Thrift::Server;
use Example2::Service;

package ServiceHandler;
use base 'Example2::ServiceIf';

sub new {
    my $class = shift;

    my $self = {};
    bless $self, $class;

    return $self;
}

sub ping {
    my $self = shift;
    my ($arg) = @_;

    $arg->foo('baz');

    return $arg;
}

package main;

my $handler       = ServiceHandler->new;
my $processor     = Example2::ServiceProcessor->new($handler);
my $serversocket  = Thrift::ServerSocket->new(9090);
```

```
36 my $forkingserver = Thrift::ForkingServer->new($processor,
       $serversocket);
37
38 print "Starting the server...\n";
39 $forkingserver->serve();
```

На стороне клиента, во-первых, подключаем модуль `Example2::Types`, где определен тип `Example2::Hash`, создаем экземпляр структуры и отправляем серверу:

```
1 #!/usr/bin/env perl
2
3 use strict;
4 use warnings;
5
6 use Thrift;
7 use Thrift::BinaryProtocol;
8 use Thrift::Socket;
9 use Thrift::BufferedTransport;
10
11 use Example2::Service;
12 use Example2::Types;
13
14 my $socket = Thrift::Socket->new('localhost', 9090);
15 my $transport = Thrift::BufferedTransport->new($socket, 1024,
       1024);
16 my $protocol  = Thrift::BinaryProtocol->new($transport);
17 my $client    = Example2::ServiceClient->new($protocol);
18
19 $transport->open();
20
21 eval {
22     my $in = Example2::Hash->new;
23     my $out = $client->ping($in);
24     print $out->foo, "\n";
25 } or do {
26     my $e = $@;
27
28     print "Error: $e->{message}\n";
29 };
30
31 $transport->close();
```

Таких аргументов и типов может быть сколь угодно. Подробнее о типах можно почитать в исчерпывающей документации Thrift.

Исключения

Часто сервисы могут бросать исключения при ошибках в методах. Thrift поддерживает обмен исключениями, а также позволяет указывать, какой метод какие исключения может бросить.

Например, в следующем примере метод `ping` может бросить системное исключение:

```
namespace perl Example3

exception SystemError {
    1: i32 code,
    2: string message
}

service Service {
   string ping() throws (1:SystemError err)
}
```

а реализация метода `ping` на серверной стороне может выглядеть следующим образом:

```
sub ping {
    die Example3::SystemError->new(
        {
            code    => 500,
            message => 'System error'
        }
    );
}
```

А на стороне клиента:

```
eval { print $client->ping(), "\n"; } or do {
        my $e = $@;

            print "Error: ", $e->message, "\n";
};
```

Void и неблокирующие вызовы

Если метод ничего не возвращает, достаточно указать тип возвращаемого значения `void`:

```
service Service {
   void restart()
}
```

Однако, клиент будет ожидать завершения операции `restart`. Чтобы вызов был неблокирующим, т.е. сразу же получить ответ и отключиться, указывается ключевое слово `oneway`:

```
service Service {
   oneway void restart()
}
```

Дуплексный RPC

Недостатком клиент-серверной архитектуры является невозможность сервера связаться с клиентом по своей инициативе. Этот недостаток мешает реализации, например, асинхронных уведомлений от сервера. Одним из решений данной проблемы является реализация второго реверсного RPC-канала на стороне сервера. Сервер таким образом сможет отправлять клиенту уведомления в виде неблокирующих сообщений.

В качестве примера рассмотрим простейшую подписку клиентом на уведомления, а сервер при запросе подписки запускает клиента и отправляет периодические уведомления. Здесь мы не будет углубляться в реализацию с помощью forkов, главное показать принцип.

Спецификация:

```
namespace perl Example4

service Service {
   oneway void subscribe()
}
```

Спецификация сервера уведомлений:

```
namespace perl Example4Publisher

service Service {
   oneway void notify(1:string notification)
}
```

Реализация клиента:

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

use strict;
use warnings;

use Thrift;
use Thrift::BinaryProtocol;
use Thrift::Socket;
use Thrift::BufferedTransport;
use Thrift::Server;

use Example4::Service;
use Example4Publisher::Service;

package ServiceHandler;
use base 'Example4Publisher::ServiceIf';

sub new {
    my $class = shift;

    my $self = {};
    bless $self, $class;

    return $self;
}

sub notify {
    my $self = shift;
    my ($notification) = @_;

    print $notification, "\n";
}

package main;

my $socket = Thrift::Socket->new('localhost', 9090);
my $transport = Thrift::BufferedTransport->new($socket, 1024,
   1024);
my $protocol  = Thrift::BinaryProtocol->new($transport);
my $client    = Example4::ServiceClient->new($protocol);
```

```
40
41 $transport->open();
42 $client->subscribe();
43 $transport->close();
44
45 my $handler       = ServiceHandler->new;
46 my $processor     = Example4Publisher::ServiceProcessor->new(
   $handler);
47 my $serversocket  = Thrift::ServerSocket->new(9091);
48 my $forkingserver = Thrift::ForkingServer->new($processor,
   $serversocket);
49
50 print "Starting the notification server...\n";
51 $forkingserver->serve();
```

Реализация сервера:

```
1 #!/usr/bin/perl
2
3 use strict;
4 use warnings;
5
6 use Thrift::Socket;
7 use Thrift::Server;
8 use Example4::Service;
9 use Example4Publisher::Service;
10
11 package ServiceHandler;
12 use base 'Example4::ServiceIf';
13
14 sub new {
15     my $class = shift;
16
17     my $self = {};
18     bless $self, $class;
19
20     return $self;
21 }
22
23 sub subscribe {
24     my $socket = Thrift::Socket->new('localhost', 9091);
25     my $transport = Thrift::BufferedTransport->new($socket, 1024,
          1024);
26     my $protocol  = Thrift::BinaryProtocol->new($transport);
27     my $client    = Example4Publisher::ServiceClient->new(
          $protocol);
28
29     $transport->open();
```

```
30
31      while (1) {
32          $client->notify(time);
33
34          sleep 1;
35      }
36
37      $transport->close();
38  }
39
40  package main;
41
42  my $handler       = ServiceHandler->new;
43  my $processor     = Example4::ServiceProcessor->new($handler);
44  my $serversocket  = Thrift::ServerSocket->new(9090);
45  my $forkingserver = Thrift::ForkingServer->new($processor,
        $serversocket);
46
47  print "Starting the server...\n";
48  $forkingserver->serve();
```

В данном примере используются два порта: 9090 и 9091. По первому осуществляется подписка на уведомления, а по второму собственно они и рассылаются. Для более полной реализации необходимы простейшие менеджер процессов и обзервер, что остается в качестве упражнения читателю :)

■ *Олеся Кузьмина*

4. Fuzzing-тестирование perl-интерпретатора с помощью afl

Закончились новогодние каникулы. Кто-то ездил отдыхать в жаркие страны, кто-то смотрел телевизор и не вылезал из-за(под) стола. Но были и те, кому было интересно провести бесчеловечные эксперименты с Perl. Об одном таком эксперименте и пойдёт речь.

American Fuzzy Lop

Существует довольно известный метод тестирования — фаззинг-тестирование, когда тестируемой программе передаются на ввод некоторые некорректные данные и затем наблюдают за её реакцией. Если происходит зависание, крах, утечка памяти или другое ненормальное поведение, то можно говорить об обнаружении проблемы, и высока вероятность, что это проблема в безопасности. Программа, осуществляющая фаззинг, называется фаззером. Фаззер может использовать грубую силу для перебора всевозможных входных данных или использовать какие-то шаблоны, заточенные под определённые форматы данных. В основном их разрабатывают и используют исследователи в области безопасности для поиска уязвимостей в программных продуктах.

Не так давно появился очень перспективный фаззер American Fuzzy Lop (afl), который разрабатывает известный эксперт в области безопасности Michal Zalewski. Название фаззера происходит от породы кроликов «Американский пушистый вислоухий». Как поясняет автор, фаззер создавался под впечатлением от утилиты *bunny-the-fuzzer* (*кролик-фаззер*) от Tavis Ormandi, что как-то и объясняет такое странное название.

afl внедряет в код специальные ассемблерные инструкции в точки ветвления для отслеживания поведения подопытной программы, позволяя определять, когда происходит переключение на новую ветвь исполнения, и какие входные данные повлияли на это. Для этих целей требуется собрать исходный код программы с помощью компиляторов-обёрток afl-gcc, afl-gcc++ или afl-clang в зависимости от того, какой компилятор используется.

После того как программа собрана, она может быть подвергнута тестированию с помощью фаззера afl-fuzz. Фаззер создаёт форк-сервер, который начинает запускать экземпляр программы и передавать входные данные. Отслеживает инструментальный вывод, который был зашит при компиляция программы, для того чтобы определять выбранную ветвь исполняемого кода. Автоматически вычисляются лимиты для времени выполнения кода, чтобы выявлять зависания. Как только произойдёт крах или зависание, то экземпляр входных данных, который их вызвал, помещается в выделенный каталог для последующего исследования.

Утилите afl-fuzz передаются следующие обязательные параметры:

- `-i` — каталог с начальными образцами входных данных, которые будет передаваться тестируемой программе;
- `-o` — каталог, где будут сохраняться результаты: образцы ввода, которые приводят к краху/зависанию, а также текущая очередь мутировавших входных данных.

Есть два способа передачи данных подопытной программе: на стандартный ввод или указав путь к файлу. В первом случае не требуется никаких опций:

```
1 $ afl-fuzz -i input -o output testing_program
```

Во втором случае необходимо указать шаблон `@@` в качестве параметра для испытуемой программы, который будет меняться на актуальное имя тестового файла. Это может быть полезно, если программа умеет работать только с файлами:

```
1 $ afl-fuzz -i input -o output testing_program @@
```

Иногда программа проверяет расширение файла или имеет проблемы с теми именами, которые придумывает afl-fuzz, для это есть удобная опция `-f`, которая передаёт одно и тоже имя файла:

```
1 $ afl-fuzz -i input -o output -f input_file.ext testing_program
    @@
```

После запуска фаззера мы получаем подобную картинку:

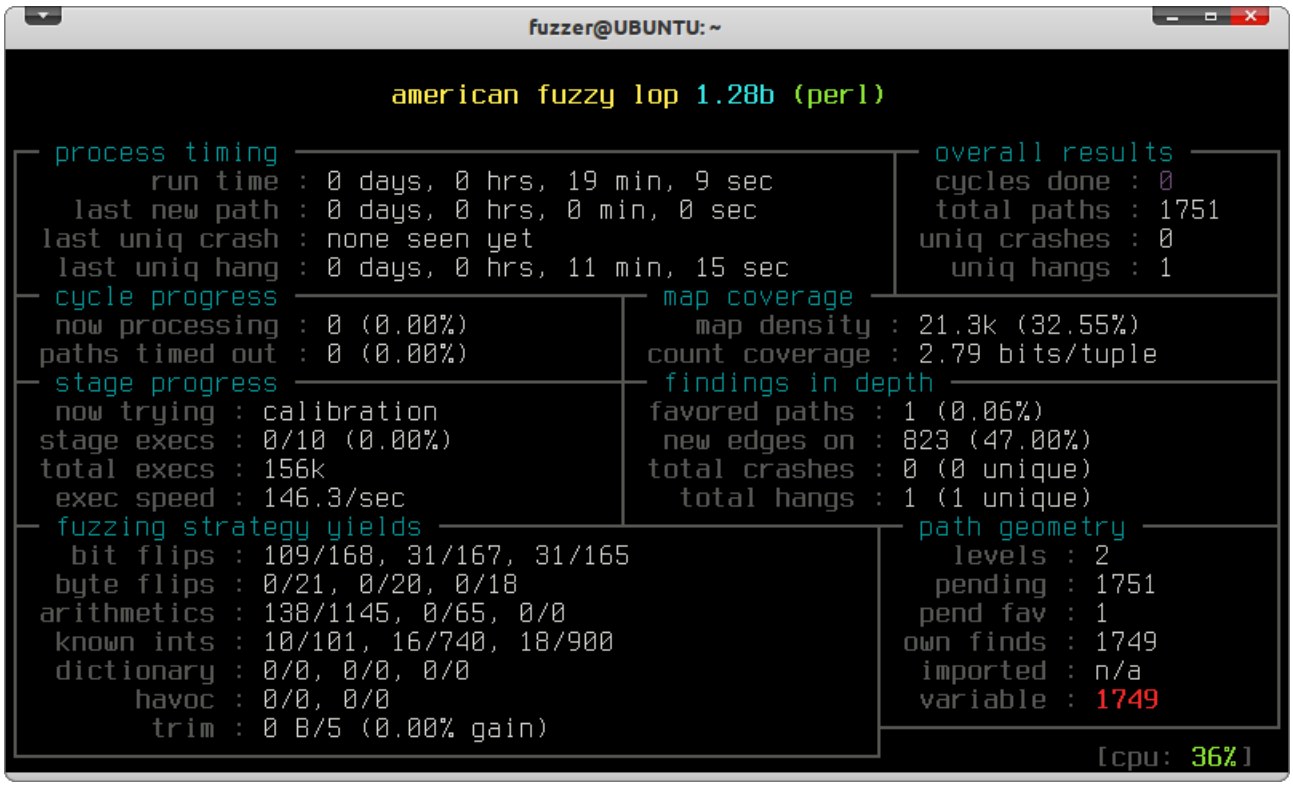


Рис. 1: afl

Здесь отображаются текущие результаты поиска. Самый важный параметр это `uniq crashes`, который определяет количество найденных крахов программы.

Тестирование Perl

Вернёмся к новогоднему эксперименту. Некто Brian Carpenter основательно взялся за проверку Perl с помощью фаззера afl. Запустив afl на простом примере CGI-скрипта, он за несколько праздничных дней умудрился собрать богатый урожай багов:

Баг #123539 (исправлен)

Простейший экземпляр кода, приводящий к чтению неинициализированной памяти (`...` — 80 символов b):

```
$perl -e '/bbbbbbbbb...bbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbAAbbb/
    il'
```

```
panic: reg_node overrun trying to emit 0, f60370>=f60370 at –e
    line 1.
```

Ошибка была выявлена в компиляторе регулярных выражений, который делает два прохода, сначала оценивая объём необходимого буфера, а затем заполняющего его. Такой дизайн зачастую приводит к рассогласованию в некоторых граничных условиях и как следствие подобным ошибкам.

Баги #123551, 123554 (исправлены)

Следующий пример:

```
$ perl –e '33x~3'
```

Приводил к панике для perl < 5.20 и ошибке сегментирования в старших версиях. Использование символа ~ для числа повторов приводило к целочисленному переполнению.

Баг #123617

Пример кода, вызывающего ошибку сегментирования:

```
$ perl –e '"$a{m/""$b
 / m ss
";@c = split /x/'
```

Проблема пока не исправлена.

Баг #123542 (исправлен)

Пример кода, приводящего к краху:

```
$ perl –e 's/${<>{})//'
```

При построении дерева операций происходит разыменование нулевого указателя и последующий крах программы.

Баг #123677

Пример кода, приводящего к краху:

```
$ perl –e 's)$0{0h());qx(@0);qx(@0);qx(@0)'
```

Проблема актуальна для версии Perl ≥ 5.21.7.

Как присоединиться к тестированию

Вероятно, у вас появится желание самим заняться тестированием, поэтому можно составить короткую инструкцию о том, как собрать Perl для подобных целей.

Прежде всего необходимо создать пользователя в системе для проведения тестов (в идеале нужна выделенная виртуальная машина или чрут), так как в процессе тестирования могут выполняться произвольный код, создаваться или удаляться файлы. Не стоит рисковать рабочей системой.

```
$ sudo useradd –m fuzzer
$ sudo su – fuzzer
```

Загрузим и соберём последнюю версию фаззера:

```
$ wget http://lcamtuf.coredump.cx/afl/releases/afl–latest.tgz
$ tar xf alf–latest.tgz
$ cd afl–1*
$ make PREFIX=$HOME/afl
$ make PREFIX=$HOME/afl install
```

Чтобы собрать perl с использованием afl, удобнее всего воспользоваться perlbrew. Например, чтобы собрать последний версию Perl из blead, нужно выполнить команду:

```
$ perlbrew install –n ––as perl–blead–afl –Dusedevel \
    –Dcc=$HOME/afl/bin/afl–gcc \
    –DAFL_PATH=$HOME/afl/lib/afl \
    perl–blead
```

Здесь мы указываем путь к компилятору afl-gcc, которым будет собран perl. Переключимся на новый Perl

```
$ perlbrew use perl-blead-afl
```

Создадим произвольный пример кода

```
$ mkdir perl-input
$ echo '$x = eval { die $! }' > perl-input/test.pl
```

Запустим фаззер:

```
$ ~/afl/bin/afl-fuzz -i perl-input -o perl-output perl
```

Ждем сутки, двое, трое... и возможно фаззер что-нибудь найдёт.

Что ещё удалось найти

Воспользовавшись указанной выше инструкцией, мне удалось найти четыре(!) уникальных примера, которые приводили к краху Perl. Опишу только те, которые успел проанализировать.

Баг #123652

Пример кода:

```
$ perl -e '$1=eval{a:}'
zsh: segmentation fault perl -e '$1=eval{a:}'
```

Данная проблема появилась, начиная с Perl 5.13.6. Присвоение переменной только для чтения `$1` здесь для отвлечения глаз, а основная проблема в `eval`, который содержит лишь одну метку. Код оптимизатора содержал ошибку, обращаясь к элементу структуры, которая не была создана. Детальное пояснение сделал Father Chrysostomos, который и исправил этот баг.

Баг #123712

Ещё один простой пример некорректного кода, который вызывает крах парсера:

```
$ echo –n '/$a[/<<' | perl
```

В данном примере используется `echo –n`, чтобы подчеркнуть, что код не содержит переноса строки (при использовании `–e` это уже не сработает). В конце примера располагается `<<`, что указывает на начало встроенного документа, но он пустой. Фрагмент кода функции сканирующая встроенный документ `S_scan_heredoc`:

```
while (s < bufend – len + 1 &&
    memNE(s,PL_tokenbuf,len) ) {
    if (*s++ == '\n')
        ++shared–>herelines;
}
```

В данном коде указатель на строку `s`, в которой ищется перенос строки, оказывается равным `NULL`, и при выполнении функции `memNE` (`memcmp`) происходит ошибка сегментирования.

Как получить минимальный пример кода, который приводит к краху?

Предположим, вы получили образец кода, который приводит к краху, но как вычистить мусор, который не влияет на результат и мешает поиску ошибки? Для этих целей фаззер имеет опцию `–C`, которая позволяет проводить исследование краха. Например:

```
$ ~/afl/bin/afl–fuzz –C –i dir–with–crash–sample –o other–crash–
    samples perl
```

Если в каталоге dir-with-crash-sample есть образец скрипта, который приводит к краху Perl, то в этом режиме фаззер начнёт модифицировать пример и ожидать краха. Если крах не происходит — пример отбрасывается, если происходит, то образец сохраняется. Среди них можно обнаружить наиболее короткие и возможно наиболее чётко выявляющие проблему образцы.

Альтернативный и более быстрый вариант получения минимального образца — это использование утилиты afl-tmin. В этом случае происходит быстрый перебор отсечения блоков различного размера для получения минимального размера файла, который приводит к краху. В этом случае никакой интеллектуальной эвристики не применяется.

```
$ ~/afl/bin/afl-tmin -i crash_sample.pl -o min_crash_sample.pl
    perl
```

Другие варианты исследований

Помимо самого Perl можно производить исследование различных модулей со CPAN и прежде всего XS-модулей. Можно попробовать фаззить сериализаторы `Storable`, `Sereal`, `JSON::XS` и другие. Можно также подвергнуть тестированию шаблонизаторы, например, `Text::Xslate`. Можно исследовать и модули, работающие с изображениями.

Для примера, возьмём `Imager`.

```
$ perlbrew use perl-blead-afl
$ cpanm Imager
```

В данном случае cpanm автоматически соберёт модуль `Imager` с использованием компилятора, которым собирался Perl, т.е. afl-gcc.

```
$ ~/afl/bin/afl-fuzz -i dir-with-image -o imager-output \
    -f image_file \
    perl -MImager -e 'Imager->new(file=>$ARGV[0])' @@
```

Данная команда будет загружать `Imager` и подгружать файл изображения, которые генерирует afl-fuzz. Т.о. можно искать ошибки в `Imager`.

К сожалению, `Imager` работает достаточно медленно, и пока ничего интересного выявить не удалось.

Заключение

В статье рассмотрен инструмент для фаззинг-тестирования afl, приведены примеры найденных с его помощью ошибок в Perl. Если вас заинтересовал этот инструмент, то вот несколько ссылок на статьи о других найденных ошибках с помощью afl:

- Список трофеев afl

- Ошибки termcap в OpenBSD
- Как файл со словом hello мутирует в JPEG-изображение

■ *Владимир Леттиев*

5. Каналы в Perl 6

Первая часть обзора возможностей Perl 6 для параллельных и конкуррентных вычислений

В Perl 6 входят многие интересные решения, предназначенные для параллельных вычислений. Причем это встроено прямо в синтаксис языка, поэтому для работы со всем этим не придется подключать какие-либо библиотеки. Несмотря на серьезную задержку с разработкой языка, сегодняшнее распространение многоядерных процессоров делает Perl 6 хорошим кандидатом для того, чтобы вновь завоевать сердца программистов.

В этой серии статей я рассмотрю те механизмы, которые доступны сегодня, и с которыми можно экспериментировать, используя компилятор Rakudo Star с виртуальной машиной MoarVM (о его установке читайте в предыдущем номере журнала). Сегодня речь пойдет о каналах.

Rakudo 2014.12

После выхода предыдущей статьи появилось обновление Rakudo Star, поэтому прежде чем продолжить, есть смысл обновиться. Процедура установки из нового дистрибутива (rakudo-star-2014.12.1.tar.gz) осталась неизменной. О том, что изменилось в самом Rakudo, можно почитать на сайте проекта.

```
$ perl6 –v
This is perl6 version 2014.12 built on MoarVM version 2014.12
```

Каналы

Идея очень простая (и уже реализованная, например, в Go). Создается канал, в который можно писать, и из которого можно читать. Эдакий пайп, но канал.

Запись и чтение

В Perl 6 существует класс `Channel`, где определены, в частности, методы `.send` и `.receive`. Вот простейший пример, в котором в канал `$c` пишется целое число, которое тут же читается и выводится на экран:

```
my $c = Channel.new;
$c.send(42);
say $c.receive; # 42
```

Канал, как и любую переменную, можно передать в функцию, и тогда внутри нее удастся читать из этого канала:

```
my $ch = Channel.new;
$ch.send(2015);
func($ch);

sub func($ch) {
    say $ch.receive; # 2015
}
```

В канал можно отправить несколько значений. А затем прочитать их одно за другим (на выходе из канала данные появляются в том же порядке, в котором они были добавлены):

```
my $channel = Channel.new;

# В канал уходят несколько нечетных чисел:
for <1 3 5 7 9> {
    $channel.send($_);
}

# А теперь они читаются до тех пор, пока в канале есть данные.
# while @a -> $x эквивалентно for my $x (@a) в Perl 5.
while $channel.poll -> $x {
    say $x;
}

# После того, как все прочитано, возвращается Nil.
$channel.poll.say; # Nil
```

В последнем примере вместо метода `.receive` был применен `.poll`. Их различие проявляется, когда в канале больше нет данных для чтения. В этом случае первый метод блокирует выполнение программы до поступления новых данных, а второй сразу возвращает `Nil`.

Если же использовать в цикле метод `.receive`, но закрыть перед этим канал:

```
$channel.close;
while $channel.receive –> $x {
    say $x;
}
```

то уже находящиеся в канале данные возможно прочесть, но после того, как они закончатся, произойдет исключение: `Cannot receive a message on a closed channel`. Код, разумеется, можно поместить внутрь блока `try`, но куда проще использовать `.poll`.

```
$channel.close;
try {
    while $channel.receive –> $x {
        say $x;
    }
}
```

В этом примере закрытие канала — обязательное условие, без которого программа будет бесконечно ожидать в канале новых данных.

Метод list

Помимо методов для записи и получения одиночных значений существует метод `.list`, возвращающий все непрочитанное, что осталось в канале:

```
my $c = Channel.new;

$c.send(5);
$c.send(6);

$c.close;
say $c.list; # 5 6
```

Метод блокирует программу до тех пор, пока канал не иссякнет, поэтому перед вызовом полезно закрыть канал, вызвав `.close`.

За пределами скаляров

Здесь уместно удивиться, почему для чтения списков создан отдельный метод вместо того, чтобы изменить работу метода `.receive` в списочном контексте. А все потому, что в Perl 6 списки и хеши вполне могут быть использованы точно так же, как скаляры: и там, где в Perl 5 список развернулся бы в набор отдельных значений, в Perl 6 он передается как единая переменная. Поэтому возможны такие трюки:

```
my $c = Channel.new;
my @a = (2, 4, 6, 8);
$c.send(@a);

say $c.receive; # 2 4 6 8
```

Массив `@a` передается в поток как единое целое, и точно так же извлекается весь целиком за один вызов `.receive`;

Более того, если присвоить результат скалярной переменной, то в этом контейнере окажется массив:

```
my $x = $c.receive;
say $x.WHAT; # (Array)
```

То же самое будет работать, например, и с хешами:

```
my $c = Channel.new;
my %h = (alpha => 1, beta => 2);
$c.send(%h);

say $c.receive; # "alpha" => 1, "beta" => 2
```

Вместо метода `.list` возможно использовать сам канал в списочном контексте (предварительно закрыв канал):

```
$c.close;
my @v = @$c;
say @v;
```

Или так:

```
$c.close;
for @$c -> $x {
    say $x;
}
```

Метод .closed

В классе Channel определен еще один полезный метод .closed, позволяющий проверить, открыт канал или нет:

```
my $c = Channel.new;
say "open" if !$c.closed; # открыт

$c.close;
say "closed" if $c.closed; # закрыт
```

Несмотря на простоту использования метода, на самом деле он возвращает не булево значение, а объект-промис (переменную типа Promise, о них в следующий раз). В первом случае обещание (промис) того, что канал закрыт, еще только дано:

```
Promise.new(scheduler => ThreadPoolScheduler.new(initial_threads
    => 0,
max_threads => 16, uncaught_handler => Callable), status =>
    PromiseStatus::Planned)
```

А во втором оно уже сдержано: канал к этому времени действительно закрыт.

```
Promise.new(scheduler => ThreadPoolScheduler.new(initial_threads
    => 0,
max_threads => 16, uncaught_handler => Callable), status =>
    PromiseStatus::Kept)
```

Состояние промиса указано в поле status.

Заключение

В этой статье рассказано, как использовать каналы в обычной программе. Но тем не менее каналы считаются thread-safe, поэтому замечательно подходят для многопоточных приложений. Я постараюсь вернуться к этому вопросу в будущем, после рассмотрения промисов и саплаеров.

■ *Андрей Шитов*

6. Perl 6, или Get ready to party

Заметки с выступления Ларри Уолла на Фосдеме

Пару месяцев назад появилось сообщение о том, что на конференции FOSDEM Ларри Уолл объявит, что Perl 6 будет готов для продакшна в 2015 году. Я не мог пропустить такое событие и съездил на час в Брюссель ради того, чтобы услышать это из первых уст.

Фосдем (Free and Open source Software Developers' European Meeting) — двухдневная бесплатная конференция, которая проходит ежегодно в конце января — начале февраля и собирает несколько тысяч разработчиков. Отдельным языкам программирования выделены так называемые деврумы, где в течение дня звучат выступления про этот язык.

В этому году в расписании субботнего Perl-деврума было аж пять докладов про Perl 6:

- Штефан Сайферт. Leapfrogging the bootstrap Bringing whole module ecosystems to Perl 6
- Патрик Мишо. Bringing whole module ecosystems to Perl 6. Lessons learned
- Куртис Поэ. Perl 6. A Dynamic Language for Mere Mortals
- Тадеуш Сошнеш. "Fast enough" Perl 6. How I learned to stop worrying and love games
- Джонатан Вортингтон. Perl 6: beyond dynamic vs. static

А в воскресенье состоялось выступление Ларри Уолла Get ready to party!. Это было уже вне формата деврума: доклад размещен в секции «Языки» и проходил в главном зале Janson, вмещающим 1500 слушателей. Свободных мест почти не было.

На сайте Фосдема опубликовано небольшое интервью, сделанное перед конференциией. Видеозапись выступления должна появиться на странице live.fosdem.org.

Ларри начал с цитаты из «Хоббита»:

Когда мастер Бильбо Бэггинс из Бэг-Энда объявил, что намерен в самом скором времени отпраздновать свое одиннадцатьдесят однолетие особенно великолепным приемом, весь Хоббитон загудел и заволновался.

111 лет это довольно много, и ему самому в этом году будет 61. С того времени, как начали заниматься Perl 6 в 2000 году, пару раз он боролся с раком, перенес две операции по удалению катаракты, и дважды случился экономический кризис.

Весь доклад был построен на параллелях с произведениями Толкиена. Например, Perl 5 и Perl 6 соотносятся так же, как две книги: «Хоббит» и «Властелин колец».

Толкиен тоже долго писал продолжение. Но для тех, кто только сейчас начнет его читать, ждать 15 лет уже не нужно. Общий принцип: если что-то делалось долго, но сейчас уже завершено или почти готово, то нет никакого повода переживать, особенно тем, кто не участвовал в процессе ожидания. Те, кто родился позже, сразу получат готовыми обе книги.

За последние годы было предпринято много попыток форкнуть пятый перл, чтобы создать Perl-подобный язык, но все это не было особо успешно. Perl 6 не обязательно должен стать для кого-то первым языком, неплохо, если он станет для них последним и лучшим.

Ларри немного рассказал о задачах, которые стояли перед дизайном Perl 6, и о том, насколько они были амбициозны и невыполнимы, и показал несколько примеров, где Perl 6 значительно превосходит Perl 5 и другие языки.

Программа на Perl 5 компилируется в несколько проходов, на каждом шаге исходный код постепенно изменяется, и поэтому компилятор не всегда точно знает, с каким языком имеет дело в данный момент. В Perl 6 этот аспект более логичен.

Например, шаблоны (бывшие регулярные выражения) теперь не просто строки. Они парсятся наравне с основным языком как его подъязык.

В Perl 5 в двух частях оператора `s///` одна и та же подстрока именуется поразному: то `\1`, то `$1`:

```
1 s/ \b (\w+) \s+ \1 \b /$1/x
```

В Perl 6 все едино:

```
s/ « (\w+) \s+ $0 » /$0/
```

Важной особенностью Perl 6 является возможность изменять грамматику языка на лету, подмешивая к ней новые правила.

Например, определив постфикс ! для вычисления факториала:

```
sub postfix:<!>($n) {
    [*] 2..$n;
}

say 42!
# 1405006117752879898543142606244511569936384000000000
```

Причем модифицированная грамматика начинает действовать даже пока она еще не полностью описана, и это, по мнению Ларри Уолла, является киллер-фичей Perl 6. Классическое рекурсивное определение факториала содержит вызов ! внутри себя:

```
sub postfix:<!>($n) {
    $n < 2
        ?? 1
        !! $n * ($n−1)!
}
```

То есть новое правило используется прямо в теле своего определения.

Важная особенность, которая вошла в язык, — ленивые списки, контексты и даже исключения.

Пример ленивого списка: определение (причем как константы) бесконечного списка

```
constant fib = 0, 1, * + *, * ... *;
```

Правила определены в момент компиляции, а реальные вычисления происходят по мере необходимости:

```
say fib[^20];
# 0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597 2584 4181
```

Ленивые контексты, соотвтетственно, тратят ресурсы только в момент вызова функций. А ленивые исключения могут ничего не сообщить об исключи-

тельной ситуации до тех пор, пока ее результат реально не позволит продолжать выполнение программы (в этот момент, тем не менее, сообщение об ошибке будет содержать информацию о причинах исключения).

Наконец, на 41-й минуте (из 50) выступления, было сделано долгожданное объявление. В сентябре (на день рождения Ларри) будет выпущена официальная бета, а на Рождество 2015-го — релиз 6.0.0.0.0.0.

Тут же последовал мелкий шрифт с перечнем того, что может этому помешать, либо чего не следует ожидать сразу.

1. Автобус.
2. Все участники процесса — волонтеры, поэтому могут вноситься коррективы в приоритетах и пр.
3. Из набора тестов будет исключено все, что не реализовано на момент релиза.
4. На Рождество достаточно одной реализации, и это будет Rakudo на MoarVM.
5. Производительность будет выше, чем сейчас, но все равно не стоит ожидать, что она превзойдет Perl 5 (стабильность — более важная задача).
6. Версия 6.0 это все еще *.0*, поэтому она не обязательно будет сразу совершенной.

Сейчас процесс разработки старается завершить три основные задачи: GLR, NSA, NFG. Это, соответственно, Grand List Refactoring, Natively Shaped Arrays и Normal Form Grapheme. За подробностями я отсылаю читателя к недавнему посту из адвент-календаря The State of Perl 6 in 2014.

Наконец, Ларри добавил, что когда-нибудь может быть начнет и Perl 7.

■ *Андрей Шитов*

7. Обзор CPAN за январь 2015 г.

Рубрика с обзором интересных новинок CPAN за прошедший месяц.

В этом месяце заметно выросло число обновлённых модулей, что отчасти является следствием проводимого конкурса 2015 CPAN Pull Request Challenge.

Статистика

- Новых дистрибутивов — 252
- Новых выпусков — 1222

Новые модули

Shell::Tools

`Shell::Tools` отличное подспорье администраторам при написании небольших скриптов на Perl. При включении модуля загружается множество полезных функций для работы с файлами, каталогами, например, функции `cwd`, `abs_path`, `basename`, `copy`, `make_path`, `tempfile` и множество других. Кроме того, модуль автоматически добавляет функции `VERSION_STRING` и `HELP_MESSAGE`, которые соответственно вызываются, если указываются параметры скрипта `--version` и `--help`. Версия берётся из переменной `$VERSION`, а текст помощи из встроенной POD-документации скрипта.

Mojo::Reactor::POE

`Mojo::Reactor::POE` позволяет использовать модуль POE в качестве бекенда для обработки событий в `Mojo::IOLoop`. Необходимо лишь задать переменную окружения `MOJO_REACTOR` до загрузки `Mojo::IOLoop`:

```
1 BEGIN { $ENV{MOJO_REACTOR} ||= 'Mojo::Reactor::POE' }
2 use Mojo::IOLoop;
```

FFI::Me

Модуль `FFI::Me` предоставляет удобный способ вызывать функции из C-библиотек, не применяя XS-магии. Модуль базируется на `FFI::Raw` и просто добавляет немного синтаксического сахара для удобства использования:

```
package My::Trig;
use FFI::Me;
...
ffi cosine => (
    lib => 'libm.so'
    rv  => ffi::double,
    arg => [ffi::double],
    sym => 'cos',
    method => 1,
);

package main;

my $trig = My::Trig->new;
$trig->cosine(2.0);
```

HTML::Differences

Модуль `HTML::Differences` даёт вменяемый способ найти различия между двумя HTML-файлами или фрагментами HTML-документов. Под капотом используется `HTML::Parser`, таким образом применяется разбор тегов и строится модель HTML-документа. Для построения различий игнорируются лишние пробелы (за исключением текста в `<pre>`), начальные теги нормализуются, сортируются их атрибуты. Отсутствующие закрывающие теги не добавляются автоматически, поэтому они будут видны в выводе отличий двух документов. Отличия в блоках комментариев также будут видны, но можно задать флаг для игнорирования различий в них.

Plack::App::Hostname

`Plack::App::Hostname` позволяет организовать функцию виртуального хостинга в PSGI-приложении: вызов приложения, в зависимости от имени, указанного в запросе заголовка Host.

DTL::Fast

Модуль `DTL::Fast` — это реализация языка шаблонов Django для Perl. Реализована поддержка большинства тегов, но есть определённые несовместимости с оригинальной спецификацией. Приведены результаты бенчмарков, по которым `DLT::Fast` работает на 80% быстрые, чем оригинальный питоновский шаблонизатор Django.

Crypt::Spritz

Марк Леманн представил реализацию нового поточного шифра Spritz для Perl. Алгоритм Spritz уникален тем, что имеет многостороннее применение: и как потоковый шифр, и как генератор случайных чисел, хеш и аутентифицированное шифрование.

Data::Fake

Модуль `Data::Fake` позволяет генерировать случайные данные разных типов, использую декларативный синтаксис, например:

```
fake_name() ––– произвольное имя
fake_sentences(1) ––– произвольное предложение
fake_past_date("%Y–%m–%d") ––– произвольная дата в прошлом
```

Модуль позволяет создавать не только простые структуры данных, но также хеши и массивы. Основное применение — данные для тестов.

FFI::Platypus

Platypus (с англ. *утконос*) — это ещё один модуль для динамического вызова функций библиотек на C/C++ в Perl без использования XS. Данная реализация использует библиотеку libffi. Platypus имеет хорошо продуманную систему типов, которая позволяет работать не только с простыми типами `int`, `float` и прочими, но и со сложными структурами данных.

Plack::I18N

Модуль `Plack::I18N` позволяет добавить поддержку интернационализации в ваше веб-приложение. Поддерживаются как традиционные po-файлы переводов gettext, так и pm-файлы модуля локализации `Locale::Maketext`.

Модуль включается в приложение как прослойка `Plack::Middleware::I18N`, которая позволяет определять предпочитаемый язык веб-клиента (по `PATH_INFO`, сессии, http-заголовка `Accept`) и соответствующим образом формировать переменную окружения `plack.i18n`.

Обновлённые модули

Dancer2 0.158000

Новая версия Dancer2 вышла в первый день нового года. В новом релизе включены несколько важных исправлений, в том числе правильный импорт прагмы utf8 в код приложения, что само по себе очень поучительно и полезно знать каждому Perl-программисту.

DateTime 1.18

В новом релизе добавлены данные о новой секунде координации, которая будет добавлена 30 июня 2015 г. Ждём новых зависаний серверов этим летом.

Twiggy 0.1025

Новый релиз асинхронного веб-сервера `Twiggy` исправляет ошибку, когда во время потоковой передачи клиент неожиданно завершает соединение, что приводит к остановке всего процесса веб-сервера.

perlsecret 1.012

Новый релиз секретных операторов Perl содержит описание нового секрета — Змей-искуситель (*Serpent of truth*).

```
my $true  = ~~!! 'a string';    # 1
my $false = ~~!! undef;         # 0
```

Он составлен из двух операторов: `~~` (червяк) и `!!` (пиф-паф) и аналогичен по действию оператору `0+!!` (ключ к правде). Позволяет превратить логическое значение в цифровое.

Cpanel::JSON::XS 3.0115

Обновлённый `Cpanel::JSON::XS` содержит исправление ошибки при кодировании вложенных объектов методом FREEZE, которая могла приводить к повреждению стека и аварийному завершению процесса. Скорее всего проблема присутствует и в оригинальном `JSON::XS`, но о его исправлении ничего неизвестно.

Markdent 0.25

Парсер Markdown-разметки `Markdent` теперь генерирует HTML-документ в соответствии с спецификацией HTML 5. Также исправлена ошибка с использованием закрывающего тега `</th>` для всех ячеек при генерации таблиц.

DBD::Pg 3.5.0

`DBD::Pg` — драйвер базы данных PostgreSQL для DBI `DBD::Pg` теперь поддерживает экранирование символа подстановки `?`. Например, следующий запрос содержит JSONB-оператор `?` и один символ, который будут заменён значением:

```
SELECT '{"a":1, "b":2}'::jsonb \? ?
```

IO::Socket::SSL 2.010

В новом релизе `IO::Socket::SSL` появилась поддержка расширения ALPN, которое используется в протоколе HTTP2 (требуется openssl ≥ 1.02, Net::SSLeay ≥ 1.56). Удалено анонсирование слабого шифра RC4.

perl 5.21.8

Выпущен девятый релиз Perl для разработчиков 5.21.8. Среди новшеств стоит отметить новый флаг регулярных выражений `n`, который отключает захват и заполнение переменных `$1`, `$2` и т.д. внутри групп:

```
"hello" =~ /(hi|hello)/n; # $1 не устанавливается
```

Появился экспериментальный атрибут `const` для анонимных функций, который тут же запускает функцию, фиксирует возвращаемое значение и возвращает функцию-константу:

```
*INLINED = sub : const { $x }
```

Таким образом, даже если переменная `$x` будет изменена в будущем, функция `INLINED` всегда будет возвращать только то значение, которое было в момент присвоения `INLINED`.

Появилось несовместимое изменение для экспериментальных сигнатур, теперь прототип необходимо указывать не до, а после сигнатуры, что является более естественным:

```
sub sum ($left, $right) : prototype($$) {
    return $left + $right;
}
```

Несколько обескураживающим выглядит изменение в прагме warnings. Теперь появились предупреждения вне категории `all`. Дерево категорий теперь выглядит так:

```
everything -+
            |
            +- all -+
            |
            .........
```

```
6               |
7               +– extra –+
```

Теперь, чтобы охватить все-все-все возможные предупреждения придётся писать

```
1 use warnings 'everything';
```

Font::FreeType

Описание прислал basiliscos, который теперь сопровождает модуль `Font::FreeType`

После 10 лет стагнации обновился модуль `Font::FreeType`, который представляет собой Perl-обвязку для библиотеки FreeType2. FreeType2 является кроссплатформенным высококачественным движком для работы с векторными и растровыми шрифтами разных типов (ttd, odf, bdf и др.). Например, есть возможность рендеренга глифов шрифтов.

В новой версии `Font::FreeType`:

- исправлена сборка и компиляция модуля для более современных версий библиотеки FreeType2;
- демонстрационный скрипт `examples/font–info.pl` выводит больше информации, напоминая вывод ftdump из набора демонстрационных утилит FreeType;
- возможность получения дополнительных метрик шрифтов, таких как высота шрифта (`text_height`), размер глифов, находящихся над и под базовой линией (ascender и descender), ограничивающего минимального прямоугольника (`bounding box`);
- возможность доступа к таблицам кодировок шрифтов (`CharMap`);
- возможность получения мета-информации о шрифте (`NamedInfo`), такой как авторские права, лицензия, сайт автора шрифтов и т.п.

Modern::Perl 1.20150127

Модуль `Modern::Perl` традиционно выходит в начале года. Теперь запись

```
1 use Modern::Perl '2015';
```

включает новые возможности релиза perl 5.20.

■ *Владимир Леттиев*

8. Интервью с Нилом Бауэрсом (Neil Bowers)

Нил Бауэрс (Neil Bowers) — британский Perl-программист, инициатор соревнования CPAN Pull Request Challenge

Как и когда научился программировать?

В 1980-м мой учитель физики купил в класс ZX-80 (компьютерный комплект, доступный только в Британии). Через некоторое время в школе появился Commodore Pet, и я начал с ним играться. Тот уже учитель физики повез нескольких из нас в Лондон, где я купил свою первую книгу о программировании (в основном про игры, похожие на Hunt the Wunpus (*«Охота на Вампуса»* — прим. ред.)) на BASIC.

Через некоторое время у меня появился первый компьютер, Vic-20. Это было началом моей зависимости.

Назвать точную дату, когда именно я научился программировать, конечно, невозможно. Мы постоянно учимся, но все-таки же есть некоторые вехи и ключевой опыт, на который я могу сослаться. Например: на кафедре компьютерных наук я провел целое лето, портируя программу по проектированию интегральных схем с Unix на VMS, чтобы работать на имеющихся у нас графических терминалах. Мне очень помогал доктор с кафедры (спасибо, Нил!). Я выучил C, make, вообще, что есть софт, графические терминалы, проявил настойчивость и научился задавать вопросы.

Какой редактор используешь?

Я пользуюсь vim.

Моим первым настоящим текстовым редактором был SOS, которым я пользовался на первом году обучения в университете на компьютере DEC-10. В конце года я получил аккаунт на системе BSD Unix и познакомился с vi. После SOS это было потрясающе. Я также пользовался Eve и EDT, но по возможности хотелось использовать vi.

Время от времени я пробую новые текстовые редакторы, был у меня продолжительный опыт с emacs в 90-е, но когда мне требуется *серьезное* реадактирование, всегда использую vi.

Я очень смеялся, когда читал интервью с RJBS, где он сказал, что он «не Vim-фанатик и не эксперт». На QA-хакатоне в прошлом году я сидел рядом с ним, работая над тестами PAUSE. Ушел я с мыслью: «ничего себе, мне надо подтянуть свои навыки vim» и затем даже купил книгу. Лучше мне ее прочесть до следующего хакатона :)

Когда и как познакомился с Perl?

В конце 80-х — начале 90-х я много занимался обработкой текста, писал скрипты на awk и sed, также как и мой товарищ в отделе. Он наткнулся на Perl и сказал, что нам стоит попробовать. О, эта была жесть.

Я переехал в Нью-Мексико и работал над системой визуализации и обработки графических данных, написанной на C, она должна была работать на как можно многих вариантах Unix. У нас было множество машин, в том числе и арендаторов. Когда я начал там работать, у них была система сборки, написанная на шелл-скриптах. Я переписал ее на Perl 4, и мне это очень понравилось. С того момента я стабильно все больше и больше писал на Perl.

С какими другими языками нравится работать?

Много времени я провел за программированием на Javascript в то время. Иногда мне нравилось. В разное время мне нравилось писать на C, Postscript, Inform и ассемблере, не сколько из-за языка, а из-за задачи, которую я решал.

Perl всегда казался тем языком, который соответствовал моему образу мышления. Я почуствовал то же самое, когда познакомился со Scala, но писал на нем не много, и, возможно, это просто увлечение.

Многие Perl-программисты участвовали в онлайн-курсах по программированию, и видя их твиты, мне и самому захотелось поучаствовать. Я игрался с Haskell, Clojure, Rust и Go. Кто-то должен запустить соревнование по языкам: ежемесячная случайная задача на случайном языке!

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

Наверное, ответ, которого все ждут, это CPAN и сообщество вокруг него.

Лично мне с помощью Perl удается наиболее эффективно выражать себя, хотя это может быть просто опыт.

Но что действительно делает меня эффективным в Perl это CPAN. Причем это может быть на разных уровнях абстракции от раздутого фреймворка до отдельной функции в библиотеке. Когда на выходных я писал скрипт для соревнования Pull Request, мне нужно было рандомизировать массив. Я легко мог бы написать это и сам, но подумал «наверное, что-то должно быть в List::Util», и действительно, там была функция `shuffle()`.

Среди многих рейтингов/графиков на твоем сайте, о каком из них стоит людям узнать побольше?

Это не совсем рейтинг или график, но я бы сказал, что о списке модулей, которым требуются сопровождающие.

Открытый код написан и сопровождается в основном в свободное время. Интересы людей, их мотивация и время угасают по естественным причинам. Многие в порыве для удовлетворения какой-то потребности пишут модуль, загружают его на CPAN, затем занимаются чем-то другим. Позже у других людей появляется аналогичная потребность, но теперь над модулем нужно поработать, прежде чем он начнет функционировать.

Если вы хотите познакомиться с CPAN, с системой и сообществом, но у вас нет идей для написания собственного модуля, поищите дистрибутив, которому нужно внимание и который вам нравится, и принимайтесь за работу.

Как только вы закончите, подозреваю, что у вас уже появятся идеи для релиза собственного модуля.

Думаешь, CPAN нуждается в модерации?

Ха! Да, нуждается. Вообще, мне кажется, что модерация может принимать несколько форм:

- Полное удаление дистрибутивов с CPAN. Тут нужно быть осторожным, потому что нет способа узнать, используется ли кем-то этот модуль, кроме ситуации, когда он используется другим модулем. Но есть очень старые дистрибутивы, которые годами не используются. И с достаточно длинным циклом депрекации, есть множество новых дистрибутивов, которые также можно безопасно удалить.
- Депрекация модулей, в особенности, когда есть дистрибутивы, выпол-

няющие ту же задачу, но лучше. Депрекация модуля это хорошая штука, когда дистрибутив больше не сопровождается, но его нельзя удалить (пока что), потому как есть другие дистрибутивы, зависящие от него (и не забывайте про DarkPAN). Хороший пример — Any::Moose; он депрецирован, но есть множество дистрибутивов, которые его используют. Надеюсь, что они постепенно перейдут на Moo или Moose.

- Улучшение документации модулей. Очевидно, что третьи лица могут улучшать документацию, присылая патчи или pull-запросы. AnnoCPAN был попыткой решить эту проблему, но пока он не встроен в такие сервисы как MetaCPAN, он никогда не взлетит. С GitHub у меня появилась лучшая возможность предложить улучшение документации.

- Качественное содержание секции SEE ALSO в документации вашего модуля может быть одним из лучших способов помочь людям найти нужное, потому как это, например, улучшает поиск на MetaCPAN.

- Сгенерированные сообществом метаданные или аннотации. На данный момент это возможность добавить модули в Избранные, но можно также добавить теги и другие метаданные из самих дистрибутивов. Хорошая идея Рейтинг CPAN, но даже если вы улучшите однажды низко оцененный дистрибутив, очень сложно изменить его рейтинг (например, добавляя негативные отзывы, относящиеся к предудыщему релизу). Такого вида аннотации могут категоризировать «лучшее из CPAN» без написания еще одного CPAN.

- Сходимость. В то время как есть множество модулей с одинаковым функционалом, нет очевидного «лучшего» модуля: у каждого свои сильные стороны, например. Иногда, но не всегда, будет полезнее взять лучшие части разных модулей для написания одного «лучшего» модуля (с надеждой, что он будет основан на существующем модуле, а не представлять собой совершенно новый). Не совсем понятно, в чем тут мотивация для авторов, так как отдельные модули были написаны для удовлетворения каких-то текущих потребностей. Показательный пример: я пытался сделать что-то подобное вместе с Олафом Алдерсом (интервью, — прим. ред.) основываясь на одном из моих сравнений модулей, но мы оба постоянно отвлекались на другие проекты.

- Инфраструктура и документация также требует модерации. Сейчас стало ясно, что людям не так-то просто начать писать модули для CPAN.

Легко написать код, но превратить его в хороший CPAN-модуль совсем не тривиально.

- Заимствование идей из других языков и утилит. В глобальном смысле мы этого не умеем.

Что такое соревнование CPAN Pull Request? Как можно присоединиться?

Это соревнование 2015 года для мотивации людей на участие в CPAN и еще большее вовлечение в «сообщество». Если вы зарегистрируетесь, то первого числа каждого месяца я отправляю вам email со случайно выбранным CPAN-дистрибутивом (у которого также есть репозиторий на GitHub).

У вас есть один месяц для написания хотя бы одного pull-запроса. Если захотите, можно сделать и больше. Идея в том, чтобы сделать что-то полезное для дистрибутива, будь то исправление ошибки, улучшение документации, соответствие современным CPAN-соглашениям или улучшие покрытия тестами, например.

Все это основывается на следующих идеях, которые у меня появились после участия в соревновании 24pullrequests.com:

> Каждый CPAN-дистрибутив можно улучшить с пользой для CPAN. Следуя этой мысли, это может быть и депрекация модуля. Выбор репозиториев в 24pullrequests был сложным. Поэтому я просто случайным образом буду выбирать вам модули. Один модуль в день это слишком много работы. Один в месяц звучит более реально. Я выставляю оценки CPAN-дистрибутивам в соответствии с их запущенностью, и выбираю те, у которых эта оценка наивысшая.

Я надеялся, что людям понравится одновременно знакомится с инфраструктурой и улучшать качество CPAN.

Чтобы присоединиться, отправьте мне свой GitHub-аккаунт: neil at bowers dot com. Также сообщите свой PAUSE-аккаунт, если он имеется, чтобы я не назначал вам ваши же модули.

Ожидал ли ты, что CPAN PRP станет таким популярным? Какие планы?

Вообще не ожидал. В прошлом я пытался проводить несколько CPAN-соревнований, и основываясь на том опыте, ожидал около дюжины регистраций. На данный момент зарегистрировано 372 участника! Было три волны регистраций: первая, когда в ноябре я написал в своем блоге, что привлекло 20 участников. Затем в сочельник я написал пост на blogs.perl.org и получил еще 50 регистрация. Затем перед Новым годом я запостил на Hacker News, и регистрации поперли!

Теперь есть IRC-канал и список рассылки, где участники помогают друг другу, например, с идеями, что делать, а также как пользоваться GitHub, Travis и т.д. Это очень позитивный канал, так как каждый пытается делать то же самое. К тому же есть естесственный самоотбор. Возможно, мы все психи?

Я ожидаю около сотни тех, кто отвалится в течение месяца, но на сегодняшний момент энергии очень много, и многое делается. Мне это доставляет удовольствие и немного пугает.

Что дальше? Я работаю над усовершенствованием схемы ранжирования дистрибутивов. Когда участники получают свое задание, их email содержит список предлагаемых идей, с которых можно начать. Чем больше вариантов мы предлагаем людям, тем больше вероятность того, что они за что-то возьмутся и сделают pull-запрос.

Вначале у меня были планы начать второе соревнование, но текущее отнимает много времени. Второе соревнование будет побуждать людей выкладывать CPAN-дистрибутивы на GitHub. Если вы это сделаете (и в метаданных дистрибутива будет присутствовать репозиторий), то дистрибутив будет участвовать в первом соревновании в следующем месяце.

Где сейчас работаешь, сколько времени проводишь за написанием Perl-кода?

Я работаю в компании, которая называется Cogendo, мы ее основали с моим другом, когда нам нечем было заняться.

Я много пишу на Perl (и Javascript), но, как и в других небольших компаниях, также занимаюсь множеством дополнительных вещей. Я программирую практически каждый день, также занимаюсь администрированием, проекти-

рованием, поддержкой и т.д.

Стоит ли молодым людям советовать сейчас учить Perl?

Зависит от того, что понимать под «молодым».

Моему сыну семь лет и он немного увлечен Logo. Пока что я не буду пытаться учить его Perl, но, скорее всего, попытаюсь с другими языками. Мне кажется, что детям лучше знакомиться с менее сложными языками, что проще и легче в обучении.

Пока что он думает, что компьютеры предназначены для Minecraft и flash-игр.

Вопросы от читателей

Когда будет CPAN-отчет за 2014 год?

Ха! Надеюсь, что скоро. Я планировал написать его где-то в начале года, но в то время началось PR-соревнование. Теперь, когда у каждого есть задание на февраль, надеюсь, что у меня появится время для отчета.

Когда я только получил вопросы на это интервью, я подумал: «О, все будет готово прежде, чем я отвечу на эти вопросы!». Хм. Похоже, что я не сильно продвинулся.

Будет также небольшое изменение в отчете этого года. Мне кажется, что стоит быть осторожным в статистике, которую предоставляешь в качестве топ-списка, так как она может побуждать людей на нежелательное поведение.

Да и неплохо, сохраняя новизну, каждый год делать что-то новое.

Насколько длинный список модулей для обзора, и когда появятся собственно обзоры?

Прежде чем я назову число, должен объяснить: когда я понимаю, что есть несколько модулей, который делают почти одно и тоже, я думаю «это потенциальный обзор», и добавляю в свой список на trello (*веб-сервис*, — прим. ред.).

У меня есть несколько обзоров в процессе написания и несколько в виде простого списка модулей, другие же просто на trello.

Общий список содержит 22 обзора. Следующий обзор будет посвящен моулям типа Exporter. Я описал почти 40 модулей, даже делал доклад на лондонском Perl-воркшопе в прошлом году. Я постоянно отвлекаюсь. Смотрите ответ на предыдущий вопрос :-)

Какой, по-твоему, самый лучший способ внести свой вклад в Perl-сообщество?

Поучаствовать в соревновании Pull Request и сделать как минимум 11 pull-запросов до конца года! :-)

Лучшим способом внести свой вклад это помочь другим людям конкретными вещами, особенно новичкам. Когда вы помогаете другим, сами многому учитесь. Я многое узнал занимаясь PRC: люди задают вопросы, и я часто думаю: «О, этого я не знаю. Но было бы полезно узнать».

Если вы хотите сделать нечто большее, займитесь сопровождением CPAN-дистрибутива. Отполируйте его и, возможно, передайте дальше.

Если вы не хотите сопровождать дистрибутив, поработайте над менее «веселыми» частями разработки и подержки CPAN: улучшите документацию, почините вредный баг, или увеличьте покрытие тестами. Получить такой pull-запрос будет приятным сюрпризом для автора.

И напишите об этом. Поделитесь своим опытом и воодушевите других.

Одной из вещей, которые радуют меня в PR-соревновании, это атмосфера дружественности и готовности помочь друг другу.

■ *Вячеслав Тихановский*