Pragmatic Perl 11

pragmaticperl.com

Выпуск 11. Январь 2014

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, Владимир Леттиев, Илья Чесноков, Роман Чепляка

Корректоры: Андрей Шитов, Георгий Бажуков

Выпускающий редактор: Вячеслав Тихановский

Ревизия: 2014-11-29 16:16

Оглавление

1 От редактора: взгляд на 2013 г.

С новым 2014 годом, Perl-программисты, менеджеры Perl-проектов, и все, кому интересны Perl и Perl-сообщество!

В англоязычном интернете вышла замечательная статья о том, что же произошло в 2013 году с Perl http://www.perl.com/pub/2014/01/the-year-in-perl-2013-retrospective.html.

Попробуем вспомнить, что же произошло в русскоязычной Perl-среде.

Март

4 марта вышел первый номер нашего журнала. Идея возникла еще в феврале, но, как уже стало традицией, выпуск был назначен

на начало нового месяца. Журнал сразу же органично заполнил пустующую нишу, и на сегодня мы имеем 11 номеров и более 800 подписчиков!

Апрель

С 4 апреля в офисе компании mail.ru начали проходить технические встречи сообщества Moscow.pm. Видео и слайды докладов можно найти на http://corp.mail.ru/Moscow.pm.

20 апреля в Одессе компания ProvectusIT провела встречу «Expert Day», посвященную отладке Perl-приложений и архитектуре web-приложения Perl/XSLT.

Август

12–14 августа прошла конференция

YAPC::Europe 2013 в Киеве. Ей предшествовало более года подготовки. Посетили конференцию более 300 человек из более чем 20 стран мира. Участники высоко оценили качество мероприятия. Многие узнали, что и на пост-советском пространстве есть большое количество Perl-программистов. А наши соотечественники имели возможность познакомиться с известными Perl-хакерами, включая автора языка Ларри Уолла.

Сентябрь

28 сентября был объявлен новый лидер Moscow.pm — Павел Щербинин.

Декабрь

21 декабря прошел воркшоп Saint Perl 2013, собравший более 70 программистов

в Санкт-Петербурге. Подробно о мероприятии читайте в этом выпуске. Также было объявлено о проведении конференции «Perl Mova» YAPC::Russia 14 июня в Киеве. Уже можно регистрироваться.

Надеемся, что и новый 2014 год будет насыщен Perl-мероприятиями!

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

Приятного чтения.

■ *Вячеслав Тихановский*

2 Впечатления от Saint Perl 5

О пятом воркшопе в Санкт-Петербурге

21 декабря в Санкт-Петербурге состоялся одиннадцатый российский Perl-воркшоп «Saint Perl». Это уже пятое мероприятие в этом городе, приуроченное ко дню рождения языка. Приятно, что с годами число участников держится на хорошем высоком уровне, в этот раз зарегистрировалось 76 человек из четырех стран.

Помещение любезно предоставила компания JetBrains. Кстати, это тоже интересное достижение: все пять лет организаторам удается найти бесплатное помещение, и каждый раз оно новое, что косвенно говорит о том, что интерес к перлу среди IT-компаний не угасает. В этом году

хозяева помещения устроили небольшую экскурсию на крышу здания, что было отличным дополнением к и без того прекрасному виду из окон.

В программе было одиннадцать полноформатных докладов и несколько блицев. Презентации всех выступлений доступны на сайте конференции.

Отдельно хочу выделить доклад Ильи Чесноков «Разработка на Perl под Raspberry PI». Илья рассказал об оптимизации небольшой программы для работы на Raspberry PI. Несмотря на то, что в реальной практике такое вряд ли востребовано большим числом разработчиков, а саму программу для такого устройства можно было бы написать на C, рассказ был крайне интересен именно тем, что в нем было показано, что можно сделать на перле и

как выжать из него максимум скорости без увеличения затрат на память. Из исходной программы, работающей 30 секунд, получилось сделать полный аналог, который выполняется менее чем за секунду.

В процессе работы над кодом удалось оптимизировать загрузку одних модулей и полностью избавиться от других, устранить повторные вычисления, и, наконец, пересмотреть сам алгоритм задачи (кстати, интересующимся подобными темами рекомендую читать отчеты о работе над задачами Perl Golf начиная с восьмого номера Pragmatic Perl). Приятно, что все это сделано на современной версии перла 5.14.2. Рассказ Ильи опубликован и в этом номере журнала.

Надеюсь, что в следующем году получится провести как минимум такую же успеш-

ную конференцию «Saint Perl — 6» в этом прекрасном городе.

■ *Андрей Шитов*

3 Про опыт разработки на Perl под Raspberry PI

В статьей представлен расширенный вариант доклада на воркшопе Saint Perl 2013

Задача

Как-то раз на одном из фрилансерских сайтов я нашел такой проект: разработать на Perl программу для поиска объявлений, релевантных для соответствующих пользователей. Данные объявлений и профилей пользователей хранятся в XML-файлах и имеют примерно следующий вид:

Профили (`profiles.xml`):

```
1 <?xml version="1.0" encoding="utf
```

```
    –8"?>
<xml>
  <profiles>
    . . .
    <profile>
      <name>David Brown</name>
      <gender>Male</gender>
      <age>34</age>
      <likes>
        <like>Music</like>
        <like>Sports</like>
        <like>Movies</like>
      </likes>
      <timestamp>2013–10–25
         13:34:45</timestamp>
    </profile>
    . . .
  </profiles>
</xml>
```

Объявления (`media.xml`):

```
<?xml version="1.0" encoding="utf
```

```
    –8"?>
<xml>
  <Playlist PlayerId="1000001">
    <Content id="23" type="image"
        location="/Content/a2/1
       w3ewsed.jpg" ageMin="23"
       ageMax="35" gender="male"
       likes="music;movies" />
    <Content id="237" type="image
       " location="/Content/0f/
       gtfrddsed.jpg" ageMax="35"
        gender="all" likes="
       sports" />
    <Content id="21" type="image"
        location="/Content/bf/1
       w3ewsed.jpg" ageMin="40"
       gender="female" />
    <Content id="33" type="video"
        location="/Content/9b/
       jiuhnnj.mp4" gender="male"
        likes="music;movies" />
  </Playlist>
</xml>
```

Для определения релевантности объявления используется следующий алгоритм: каждое объявление сравнивается с каждым профилем; если у объявления и профиля совпали такие параметры как возраст, пол, каждая из пар предпочтений (поле `likes`), то объявлению в каждом случае добавляется одно очко. Затем выбирается объявление с максимальным числом очков или, если таких объявлений несколько, случайным образом выбираем одно из них.

Основным условием задачи было то, что программа должна работать на Raspbery Pi — время поиска наиболее релевантного объявления для файлов из 100 профилей и 100 объявлений не должно превышать двух секунд. Как оказалось, последнее условие и стало самым сложным для выполнения. Также необходимо было записывать

устаревшие профили в отдельный файл `profiles_outdated.xml`, а в оригинальном файле оставлять только те, у которых `timestamp` имеет значение не раньше, чем «10 секунд назад».

Raspberry Pi

Это компьютер размером с кредитную карту на платформе ARM11 с процессором частотой 700 МГц (я говорю о Raspberry Pi model B) и графическим ядром, которое поддерживает OpenGL ES 2.0 и может декодировать Full HD видео, 512 Мб ОЗУ, парой разъемов USB 2.0, разъемом Ethernet, HDMI, RCA Video, Audio, GPIO и разъемом для SD-карты, с которой и загружается ОС.

Питать его можно от зарядного устройства

смартфона, способного выдавать 5В x 750 мА (ток лучше больше) через разъем Micro USB, ну или от других аналогичных девайсов (коллега запитывал от USB-разъема Mac Mini через переходник).

Для него существует несколько «официальных» дистрибутивов на базе Linux, доступных на странице http://www.raspberrypi.org/downloads (в их числе и нашумевшая в рунетах Pidora) и много неофициальных. Как правило, систему можно установить простым копированием образа на карту памяти формата SD. Лично мне для первоначальной настройки не понадобились даже клавиатура или монитор — девайс после загрузки сразу стал доступен по сети по доменному имени `raspberrypi`.

Стоимость такого устройства официально $35 — но это в Англии, а в российской роз-

нице мне пришлось купить его за 2200 руб (примерно $66).

Для работы мы выбрали Raspbian — это дистрибутив, основанный на Debian, который из коробки содержит Perl v5.14.2, много модулей в пакетах системы, а также на него можно установить все необходимые дополнения типа `perlbrew`, `cpanminus`, `local::lib` и т.д.

Написание программы

Первоначальный вариант программы был написан довольно быстро — в нем использовались модули, которые я привык использовать при работе на быстрых серверах при низкой нагрузке и в предварительно загружаемой среде (т.е. время

загрузки модулей не имело для меня значения). Для парсинга XML я использовал потоковый парсер `XML::Rules` в предположении, что мне будет удобно асинхронно обрабатывать устаревшие профили. Также я активно использовал отладку, а для отображения структур данных использовал функции модуля `Data::Dump`. Начало программы выглядело примерно так:

```
use common::sense;
use local::lib;
use Const::Fast;
use DateTime;
use Data::Dump qw(pp);
use File::Temp ();
use File::Copy qw(mv);
use List::UtilsBy qw(max_by);
use Time::HiRes qw(gettimeofday
    tv_interval);
use XML::Rules;
```

Используемый алгоритм был довольно прост: сначала файл профилей целиком считывался в память, значение атрибута `likes` приводилось к нижнему регистру и преобразовывалось в массив предпочтений. Затем в процессе парсинга файла профилей для каждого профиля код обходил все объявления и искал совпадающие атрибуты — пол, возраст, предпочтения. Для каждого совпадения объявлению добавлялось одно очко. Разумеется, это выполнялось только для достаточно свежих профилей (не старше 10 секунд), устаревшие профили отсеивались.

Правило для `XML::Rules` выглядело таким образом:

```
# Parse profiles file
my $profile_parser = XML::Rules->
    new(
style          => ''filter,
```

```
rules           => {
    _default  => 'raw',
    profile   => sub {
        my ($tag_name, $attr,
           undef, undef, $parser)
            = @_;

        my $profile =
           extract_tags({ %{
           $attr } }->{_content})
           ;
        $profile->{likes}
            = [map { lc $_
               ->[1]->{_content}
               } @{ $profile->{
               likes} }];

        if (is_profile_outdated(
           $profile->{timestamp})
           ) {
            mark_as_outdated(
               $profile);
            return ();
```

```
16          }
17
18          for my $ad (@{ $parser->{
               parameters}->{ads} })
               {
19              my $score =
                   calc_score($ad,
                   $profile);
20              $ad->{scores}->{
                   $profile->{name} }
                    = $score;
21              $ad->{scores}->{total
                   } += $score;
22              $parser->{parameters
                   }->{total_scores}
                   += $score;
23          }
24          return $tag_name => $attr
               ;
25      },
26  });
27  $profile_parser->filterfile(
       $profile_file, $tmp_fh->
```

```
    filename, $playlist);
```

Здесь можно увидеть, что `XML::Rules` используется в режиме фильтра, а не парсера и сохраняет результат (только свежие профили) во временный файл `$tmp_fh`.

Все работало, но время выполнения этого варианта программы составляло 30 секунд, что было для заказчика совершенно неприемлемо. Поэтому я начал оптимизировать.

Оптимизация

Шаг 1. Используем быстрые модули

Первые мои действия по оптимизации были немного наивными и основыва-

лись не на точном расчете, а, скорее, на интуиции. Для определения возраста профилей я использовал модуль `DateTime`. Я знал, что модуль `DateTime::TimeZone` довольно долго определяет локальную временную зону при первом запуске (это написано и в его документации), и поэтому решил заменить `DateTime` чем-то более быстрым. После некоторого поиска решил использовать `Date::Calc`, который имеет не слишком-то красивый интерфейс, безо всякого ООП, зато очень сильно оптимизирован при помощи XS. Замена `DateTime` на `Date::Calc` сократило время выполнения программы примерно на 4 секунды.

Помимо этого, так как мне не нужна была потоковая загрузка объявлений, для загрузки файла объявлений я решил использовать модуль `XML::Fast`, который

также написан на Си и очень быстро работает на небольших файлах.

Шаг 2. Ненужная отладка

Для отладки изначально использовался следующий код:

```
sub debug {
    warn @_ if $DEBUG;
}

# Где-то в …коде
debug('ads with scores: ' . pp(
    $playlist));
```

Может это и незаметно на первый взгляд, но этот код работает даже тогда, когда нам это не нужно — т.е. когда флаг `$DEBUG` не установлен. В том случае, когда `$playlist` содержит много данных, преобразование

содержащейся в этой переменной структуры в читабельный вид занимает довольно много времени, поэтому, когда я изменил код для отладки на приведенный ниже, время работы программы сократилось на 19 (!) секунд.

```
if ($DEBUG) {
    require Data::Dump;
}

sub debug {
    return if !$DEBUG;

    warn map { ref $_ ? Data::
       Dump::pp($_) : $_ } @_;
}

# Где-то в …коде
debug('ads with scores: ',
    $playlist); # Здесь запятая!
```

Шаг 3. Замеряем время работы

На данный момент время работы программы составляло уже около 7 секунд, но все «интуитивные» возможности оптимизации были уже исчерпаны, поэтому я решил наконец-то взяться за точные измерения и вооружился модулем `Devel::Timer` от Gábor Szabó. Этот модуль позволяет сделать то же, что бы вы сделали при помощи функций `Time::HiRes` — измерить время выполнения каждой отдельной секции кода, но предоставляет для этого удобный интерфейс и формирует отчет на основании измеренных значений.

Понаставив везде «отметок», я получил следующие результаты для хода выполнения программы:

```
Devel::Timer Report -- Total time
    : 7.3259 secs
```

```
Count     Time    Percent
-------------------------------------------------------------------------------

1  3.6184  49.39%  created temp file -> profiles processed   # подсчет очков
1  1.6455  22.46%  BEGIN -> loaded              # загрузка модулей
1  1.1747  16.03%  loaded -> media file read     # чтение файла объявлений
1  0.8662  11.82%  file moved -> ad shown        # get_ad_to_show() && show_ad()
1  0.0079   0.11%  ad shown -> END
1  0.0060   0.08%  set up profile rules -> created temp file
1  0.0036   0.05%  profiles processed -> file moved
1  0.0032   0.04%  media file read -> set up profile rules
```

```
12 1  0.0004   0.00%  INIT -> BEGIN
```

Таким образом, видно, что одна только загрузка и компиляция программы и используемых модулей занимает 1,6 секунды — почти столько, сколько должна выполняться вся программа. Поэтому я решил проверить, сколько времени загружаются модули, и получил следующий результат:

```
1 Devel::Timer Report -- Total time
    : 1.6027 secs
2 Interval  Time    Percent
3 ----------------------------------------------------------------

4 09 -> 10  0.3867  24.13%  loaded
    List::UtilsBy -> loaded XML::
    Rules;
5 06 -> 07  0.3401  21.22%  loaded
    Data::Dump -> loaded File::
    Temp
6 02 -> 03  0.3279  20.46%  loaded
    common::sense; -> loaded local
```

```
    ::lib;
10 -> 11  0.1328   8.28%  loaded
    XML::Rules; -> loaded XML::
    Fast
04 -> 05  0.1080   6.74%  loaded
    Const::Fast; -> loaded Date::
    Calc
05 -> 06  0.0851   5.31%  loaded
    Date::Calc -> loaded Data::
    Dump
11 -> 12  0.0772   4.81%  loaded
    XML::Fast -> loaded File::Map
03 -> 04  0.0539   3.36%  loaded
    local::lib; -> loaded Const::
    Fast;
07 -> 08  0.0521   3.25%  loaded
    File::Temp -> loaded File::
    Copy
08 -> 09  0.0265   1.65%  loaded
    File::Copy -> loaded List::
    UtilsBy
01 -> 02  0.0095   0.59%  BEGIN
    -> loaded common::sense;
```

```
15 13 -> 14  0.0024   0.15%  modules loaded -> END
16 00 -> 01  0.0004   0.02%  INIT -> BEGIN
17 12 -> 13  0.0003   0.02%  loaded File::Map -> modules loaded
```

Из этого отчета видно, что дольше всего загружаются модули `XML::Rules`, `File::Temp` и `local::lib`. От `local::lib` и `File::Temp` можно было избавиться довольно легко — прописав пути к библиотекам в переменной среды `PERL5LIB` и используя стандартное имя для временного файла — типа `profiles.xml.tmp`. Также я решил избавиться и от `XML::Rules`, заменив его полностью на `XML::Fast`. Асинхронная обработка, которую я надеялся использовать для ускорения программы, мне оказалась не нужна, и выигрыш от ее использования перекрывался необходимостью загрузки

«тяжелого» модуля.

Также я избавился от модулей `File::Copy`, `File::Map` и `Const::Fast` — работать с файлами оказалось быстрее стандартными средствами Perl, а пару используемых констант можно было заменить переменными — программа получилась не настолько большой, чтобы использование констант было принципиально.

Некоторый прирост по скорости принесло то, что я решил сразу приводить к нижнему регистру при помощи функции `lc()` все содержимое файлов перед парсингом, а не отдельно предпочтения для каждого профиля или объявления.

После выполнения всех этих шагов я получил такой результат работы программы:

```
time ./parser.pl samples/profile.
```

```
    xml samples/media.xml
2
3 Devel::Timer Report ~~ Total time
    : 1.8889 secs
4 Count     Time    Percent
5 ---------------------------------------------------------------

6 1  1.3133  69.53%  calculating
    scores: start -> calculating
    scores: end
7 1  0.4887  25.87%  program: start
     -> modules loaded
8 1  0.0329   1.74%  read profile
    file: start -> read profile
    file: end
9 1  0.0157   0.83%  searching for
    ad to show: start -> searching
     for ad to show: end
10 1  0.0143   0.76%  write profile
    file: start -> write profile
    file: end
11 1  0.0126   0.67%  read media
    file: start -> read media file
```

```
    : end
1  0.0046   0.24%  convert media file: start -> convert media file: end
1  0.0027   0.14%  searching for ad to show: end -> program: end
1  0.0023   0.12%  write profile file: end -> searching for ad to show: start

real    0m2.142s
user    0m2.040s
sys 0m0.070s
```

То есть можно было бы на этом и остановиться, но тут в голову пришла интересная мысль, которую я и решил воплотить в жизнь.

Шаг 4. Оптимизация алгоритма

Из замеров выше можно увидеть, что наибольшее время теперь занимает подсчет очков для каждого объявления — то есть собственно ядро программы — ее основная функция, вычисление очков для каждого объявления.

На втором месте по «тормознутости» — загрузка и компиляция программы. Этот этап можно было бы оптимизировать при помощи демонизации — т.е. предварительно загружать программу и по таймеру или через `Inotify` проверять изменения в файлах, и заново подсчитывать очки. Но этот вариант я оставил в качестве резервного, решив поближе рассмотреть алгоритм основной функции программы.

Как мы видим, ядром этой функции явля-

ется вложенный цикл, который выглядит примерно так:

```
for my $profile (@profiles) {
    for my $ad (@ads) {
        $ad->{score} +=
            calc_score($ad,
            $profile);
    }
}
```

Если оценить сложность этого алгоритма, то она составляет `O(MxN)`, где `M` — количество объявлений, а `N` — количество профилей. Таким образом, для 100 объявлений и 100 профилей мы получим 10000 итераций. Можно ли уменьшить сложность алгоритма и количество итераций? В программировании одной из частых практик для достижения подобной цели является обмен процессора на память или наоборот. То есть мы должны

пожертвовать неким количеством памяти, чтобы уменьшить загрузку процессора. В данном случае мы можем сделать это, проиндексировав объявления — то есть сохранив для каждого возможного значения каждого параметра (пол, возраст, предпочтения) список `Id` объявлений, которым соответствует значение этого параметра. И использовать впоследствии этот список при подсчете очков. Вот код, который выполняет эту работу:

```
my %age;    # Ads valid for given age
my %gender; # Ads scored for given gender
my %like;   # Ads scored for given preference
for my $ad (@{ $playlist->{content} }) {
    # Fill ages
    my $age_min = int($ad->{agemin}) || 0;
```

```
    my $age_max = int($ad->{agemax}) || 100;
    for my $current_age ($age_min .. $age_max) {
        push @{ $age{$current_age} }, $ad->{id};
    }
    # Fill genders
    given ($ad->{gender}) {
        when ('male') {
            push @{ $gender{male} }, $ad->{id};
        }
        when ('female') {
            push @{ $gender{female} }, $ad->{id};
        }
        when ('all') {
            push @{ $gender{male} }, $ad->{id};
            push @{ $gender{female} }, $ad->{
```

```
                id};
22          }
23      }
24      # Fill likes
25      for my $current_like (split
           /;/, $ad->{likes}) {
26          push @{ $like{
               $current_like} }, $ad
               ->{id};
27      }
28  }
```

Объявления проиндексированы, теперь мы должны подсчитать очки для каждого объявления. Теперь это сделать намного проще, так как нам не нужно для каждого профиля проходить по всем объявлениям и проверять, совпадают ли его параметры с параметрами профиля — мы можем сразу получить список всех объявлений, значение соответствующего параметра которых совпадает со значением параметра профи-

ля, так как эти объявления у нас хранятся в индексе. И, соответственно, добавлять очки только этим объявлениям. Код для подсчета очков выглядит таким образом:

```
# Score of each ad
my %score;
sub calc_score {
    my ($profile) = @_;
    # Gender
    if (exists $profile->{gender
       }) {
        for my $ad_id (@{ $gender
           {$profile->{gender}}
           || [] }) {
            $score{$ad_id}++;
       }
    }
    # Age
    if (exists $profile->{age}) {
        for my $ad_id (@{ $age{
           $profile->{age}} }) {
            $score{$ad_id}++;
```

```
15          }
16      }
17      # Likes
18      if (ref $profile->{likes} &&
            ref $profile->{likes}->{
            like}) {
19          for my $profile_like (@{
                $profile->{likes}->{
                like} }) {
20              for my $ad_id (@{
                    $like{
                    $profile_like } ||
                     [] }) {
21                   $score{$ad_id}++;
22              }
23          }
24      }
25  }
```

После этой оптимизации тайминги работы программы на имеющихся тестовых данных выглядели так:

```
time ./parser.pl samples/profile.xml samples/media.xml

Devel::Timer Report -- Total time : 0.8667 secs
Count     Time    Percent
-------------------------------------------------------------

1  0.5066  58.45%  program: start -> modules loaded
1  0.1656  19.11%  calculating scores: start -> calculating scores: end
1  0.1177  13.58%  index media file: start -> index media file: end
1  0.0339   3.91%  read profile file: start -> read profile file: end
1  0.0152   1.75%  write profile file: start -> write profile file: end
1  0.0130   1.50%  read media
```

```
    file: start -> read media file
    : end

real    0m1.119s
user    0m1.070s
sys 0m0.020s
```

Таким образом, за счет индексации объявлений мы сэкономили около одной секунды — то есть почти вдвое уменьшили время работы программы.

Шаг 5 и последний. С ног на голову

В последний момент я решил попробовать ускорить работу программы, не трогая саму программу. Как это так? Очень просто — дело в том, что Raspberry Pi поддерживает оверклокинг, причем Raspbian из коробки предлагает инструменты для него.

Вводим команду `sudo raspi-config`, входим в меню Overclocking и выбираем нужную частоту из доступных: 700 (по умолчанию), 800, 900, 950, 1000 МГц. При выборе последней опции нас предупреждают, что в этом режиме может испортиться карта памяти, но нас это не страшит... В итоге при частоте 1000 МГц время выполнения программы сократилось еще на 0,3 секунды и составило около 0,8 секунд. Не то чтобы этот выигрыш нам был так уж нужен, но мы можем просто иметь его в виду на случай, если нам вдруг понадобится прибавка к скорости.

Заключение

На конференции Saint Perl 2013, где я впервые делал этот доклад, мне немного не

хватило времени, и поэтому залу разрешили задать только один вопрос. Этот вопрос был: «Зачем нужна вся эта оптимизация, если можно просто переписать всё на Си?». Ответил я на него в том ключе, что изначально задача состояла в написании программы на Perl, и я действовал в рамках задания. Однако, подумав над этим вопросом хорошенько после конференции, я пришел и к другому ответу.

Во-первых, мне нравится Perl и я знаю его намного лучше, чем Си. Я бы просто не стал браться за задачу, если бы ее нужно было написать на Си. Видимо, какие-то подобные размышления руководили и заказчиком этой программы — наверное, он знал немного Perl и чувствовал себя в силах изменить или исправить как-то программу на Perl, если в ней что-то пойдет не так. А может им и экономические мотивы

руководили, не знаю. В любом случае это как раз тот случай, когда не технология руководит людьми, а люди меняют технологию под себя, чтобы она служила их интересам. Кроме того, основные используемые модули, собственно, написаны на Си и вызываются из Perl при помощи XS.

Во-вторых, как бы ни был быстр язык сам по себе, плохой алгоритм рано или поздно даст о себе знать. И лучше заранее включить мозг и найти реальные проблемы в своей программе, чем пытаться использовать другую технологию, которая по интуитивной оценке должна работать быстрее, чем используемая в данный момент. Интуитивные оценки мы перестали использовать на шаге 2 оптимизации, когда включили логику и начали искать реальные проблемы в коде, а полностью искоренили их на шаге 3, когда замерили

время выполнения, условно говоря, каждой строчки кода и начали делать из этого правильные выводы :-)

■ *Илья Чесноков*

4 Асинхронное программирование с IO::Async

Наряду с популярным AnyEvent для Perl существуют другие модули для событийно-ориентированного программирования (СОП), и, в том числе, модуль `IO::Async`, автором которого является Paul Evans. Рассмотрим чем же интересен данный швейцарский нож и какими уникальными лезвиями он обладает.

`IO::Async`

IO::Async содержит обширный набор модулей, которые позволяют решать разные небольшие задачи, такие как отслеживание состояния дескрипторов ввода/вывода (возможность записать или прочесть дан-

ные), запуск таймеров, перехват сигналов, управление процессами-потомками, наблюдение за изменениями в файловой системе и даже управление потоками выполнения (сопрограммы).

Важная особенность `IO::Async` — это активное использование так называемых обещаний (*Promises*), которые в терминологии `IO::Async` именуются объектам *Future* из одноимённого модуля. Данный объект представляет собой некоторую выполняемую задачу, которая ещё не завершилась. Future появились в `IO::Async` не так давно и ещё активно прорабатывается, но тенденция к повсеместному их использованию чётко прослеживается.

`IO::Async` может оказаться особенно удобным при использовании ООП подхода при разработке приложений, поскольку позво-

ляет наследовать свои базовые классы и задавать обработку событий в методах ваших модулей.

IO::Async::Loop

IO::Async::Loop — это базовый модуль, который задаёт цикл обработки событий в программе. Типичная создаваемая программа создаёт один цикл и добавляет в него обработчики различных событий, после чего запускает цикл в работу.

```
use IO::Async::Loop;

# Создание цикла
my $loop = IO::Async::Loop->new;

# Добавляем какой-либо обработчик
$loop->add(...);

```

```
# Запускаем главный цикл
    обработки событий
$loop->run
```

Класс `IO::Async::Loop` является абстрактным и под его капотом может работать какая-либо из существующих реализаций мультиплексора событий, например, идущие в поставке `IO::Async::Loop::Select` и `IO::Async::Loop::Poll`, соответственно использующие системные вызовы `select` и `poll`. На CPAN также могут быть найдены реализации с использованием специфический системных вызовов, например, kqueue и epoll, или библиотек EV и UV.

Приоритетом выбора того или иного бэкенда можно управлять, например, через переменную окружения `IO_ASYNC_LOOP` или переменную `$IO::Async::Loop::`

LOOP, в которых задаётся разделённая запятой последовательность субклассов:

```
$IO::Async::Loop::LOOP = 'EV,
    Epoll,Poll'
```

Или непосредственно создав цикл из нужного субкласса:

```
use IO::Async::Loop::Poll;

my $loop = IO::Async::Loop::Poll
    ->new;
```

Уведомители `IO::Async::Notifier`

Для каждого наблюдаемого события создаётся свой так называемый уведомитель — объект базового класса `IO::Async::Notifier`, который и выполняет все

низкоуровневые операции по наблюдению.

- `IO::Async::Handle` — наблюдение за событиями ввода/вывода файлового дескриптора
- `IO::Async::Stream` — подкласс для работы с потоковыми протоколами обмена
- `IO::Async::Socket` — подкласс для работы с датаграммами или сокетом напрямую
- `IO::Async::Timer` — базовый класс для таймеров
- `IO::Async::Timer::Absolute` — вызов процедуры в заданное время
- `IO::Async::Timer::Countdown` — вызов процедуры через заданный промежуток времени
- `IO::Async::Timer::Countdown` —

периодический вызов процедуры

- `IO::Async::Signal` — наблюдение и обработка получаемых процессом сигналов
- `IO::Async::Process` — запуск и наблюдение за дочерними процессами
- `IO::Async::PID` — отслеживание завершения дочернего процесса
- `IO::Async::File` — отслеживание изменений файла (поля в выводе `stat()`)
- `IO::Async::Function` и `IO::Async::Routine` — асинхронный вызов функции (вызов кода в дочернем процессе/треде)

Операции с вводом/выводом

На примере создания сетевого сервера, рассмотрим использование `IO::Async::Handle` и `IO::Async::Stream`. Сервер будет принимать соединения на порт 1234, читать ввод построчно и возвращать перевёрнутую строку в ответ:

```
use IO::Socket::INET;
use IO::Async::Handle;
use IO::Async::Stream;
use IO::Async::Loop;

my $loop = IO::Async::Loop->new;

# Создание сокета, ожидающего соединения на порт 1234
my $socket = IO::Socket::INET->new( LocalPort => 1234, Listen => 1, ReuseAddr=>1 );

```

```
# Создание хендла для наблюдения за сокетом
my $handle = IO::Async::Handle->new(
 handle => $socket,
 on_read_ready  => sub {

  # Новое подключение. Вызов accept() на сокете
  my $client = $socket->accept or die $!;

  # Создание потокового сервера
  my $stream; $stream = IO::Async::Stream->new(
   handle => $client,
   close_on_read_eof => 0,

   # Читаем полученные данные
```

```
25              on_read => sub {
26                my ( $self,
                      $buffref, $eof )
                       = @_;
27
28                # Читаем данные из
                      буфера построчно
29                # Пишем данные
                      ответа в
                      выходной буфер
                      потока
30                while( $$buffref =~
                       s/^(.*)\n// ) {
31                   $stream->write(
                        scalar
                        reverse "\n".
                        $1);
32                }
33
34                # Если получен EOF,
                       то закрываем
                      соединение,
```

```
35                  # как только все
                       данные будут
                       отправлены
36                  $stream->
                       close_when_empty
                       () if $eof;
37
38                  # не вызывать
                       повторно эту
                       функцию после
                       EOF
39                  return 0;
40              },
41          );
42
43          # Добавляем поток в цикл
44          $loop->add( $stream );
45      },
46
47  );
48
49  # Добавление хендла в цикл
50  $loop->add( $handle );
```

```
51
52 # Запуск цикла
53 $loop->run
```

Каждое новое соединение обрабатывается с помощью класса `IO::Async::Stream`, который оптимизирован для работы с потоковыми данными и автоматически формирует буфер для чтения и записи, которые передаются в виде ссылки в соответствующую функцию-колбек.

Операции ввода/вывода и объекты Future

Создание сетевого сервера и клиента достаточно распространённая задача, поэтому данный функционал был добавлен в `IO::Async::Loop`. Предыдущий пример

можно переписать следующим образом:

```
use IO::Async::Loop;

my $loop = IO::Async::Loop->new;

# Создание сокета,
    прослушивающего порт 1234
$loop->listen(
    family   => "inet",
    socktype => "stream",
    service  => 1234,
    on_resolve_error => sub { die
        "Cannot resolve - $_[0]\n
       "; },
    on_listen_error  => sub { die
        "Cannot listen\n"; },

    # Обработка подключений
    on_stream => sub {
        my $stream = shift;

        # Конфигурируем поток
```

```
18          $stream->configure(
               on_read => sub { 0 },
               close_on_read_eof => 0
                );
19
20          # Добавляем поток в цикл
21          $loop->add( $stream );
22
23          # Возвращается Future-
               объект для операции
               чтения потока до EOF
24          my $f = $stream->
               read_until_eof;
25
26          # Функция-колбек при
               завершении операции
27          $f->on_done(sub {
28                  my ($buf, $eof) =
                       @_;
29
30                  while( $buf =~ s
                       /^(.*)\n// ) {
```

```
31                          $stream->
                               write(
                               scalar
                               reverse "\
                               n".$1);
32                      }
33
34                      $stream->
                           close_when_empty
                           () if $eof;
35                      return 0;
36                  });
37      }
38  );
39
40  $loop->run
```

В данном примере при обработке клиентских подключений автоматически выполняется `accept()` и в функцию-колбэк `on_stream` передаётся готовый объект `IO::Async::Stream`. С помощью мето-

да `configure` существует возможность сконфигурировать поток точно также, как это было сделано в первом примере. Но в данном случае продемонстрирован новый подход по работе с асинхронными функциями: использование Future-объектов.

Метод `read_until_eof()` возвращает объект `Future` — *обещание*, т.е. некоторое задание, которое находится в процессе выполнения. Само задание — вычитать поток до `EOF`. С помощью метода `on_done` мы устанавливаем функцию, которая будет вызвана, когда задание будет успешно выполнено.

С первого взгляда не видно никаких преимуществ использования Future-объектов. Попробуем усложнить задачу сервера. Теперь на каждую строку, начинающуюся с `http://` выполнять загрузку указанного

url и возвращать клиенту его содержимое.

Без использования Future-объектов это можно реализовать так:

```
1
2  sub on_read {
3      my ( $self, $buffref, $eof )
           = @_;
4
5      # ёУчт количества запросов
6      my $req = 0;
7
8      while( $$buffref =~ s/^(.*)\n
           // ) {
9          my $url = $1;
10         if ($url =~ m{^http\://}
               ) {
11             $req++;
12
13             # http-запрос
14             my $http = Net::Async
                   ::HTTP->new();
```

```
            $loop->add( $http );
            $http->do_request(
                uri => $url,
                on_response =>
                   sub {
                    my ($response
                       ) = @_;
                    $stream->
                       write(
                       $response
                       ->content)
                       ;
                    $req--;
                },
                on_error => sub {
                    $req--;
                }
            )
        }
    }

    if ($eof and $req == 0) {
```

```
31          $stream->close_when_empty
               ();
32      }
33      return 0;
34  }
```

Как видно мы получаем вложенную функцию-колбек `on_response`, которая будет собирать результаты http-запросов и отправлять клиенту их результат. Если бы нам потребовалось выполнить ещё несколько асинхронных действий после успешного http-запроса, это привело бы к дополнительным вложенным колбекам, что усложняло бы восприятие кода.

Рассмотрим вариант с Future-объектами:

```
1  sub on_stream {
2      my $stream = shift;
3      $stream->configure( on_read
          => sub { 0 },
```

```
      close_on_read_eof => 0 );
$loop->add( $stream );

my $f = $stream
    ->read_until_eof
    ->then(sub {
        my ($buf, $eof) = @_;

        my @req = ();
        while( $buf =~ s
            /^(.*)\n// ) {
            my $url = $1;
            next if $url !~ m
                {http://};
            my $http = Net::
                Async::HTTP->
                new();
            $loop->add( $http
                );

            # do_request()
                возвращает
                Future-объект
```

```
19                  push @req, $http
                        ->do_request(
                        uri => $url );
20              }
21
22              return Future->
                    wait_all( @req );
23          })
24          ->on_done( sub {
25              $stream->write($_->
                    get->content) for
                    @_;
26              $stream->
                    close_when_empty()
                    ;
27          })
28  }
```

В данном примере выполнение операций выстраивается в последовательную цепочку:

- Вычитывается поток.
- Выполняется создание http-запросов по заданным url и возвращается Future-объект, который выполнится при условии завершения всех http-запросов.
- Финальный результат отправляется клиенту, и соединение завершается.

Если бы потребовалось добавить какие-то дополнительные операции после выполнения http-запроса, то они бы последовательно добавлялись в данную цепочку, ясно и чётко демонстрируя логику программы.

Датаграммы и прямая работа с сокетом

В отличии от `IO::Async::Stream`, `IO::Async::Socket` предназначен для работы

с датаграммами, т.е. дискретными сообщениями.

```
use IO::Async::Socket;

use IO::Async::Loop;
my $loop = IO::Async::Loop->new;

# Подключение по протоколу UDP на порт 1234
$loop->connect(
    host     => "127.0.0.1",
    service  => "1234",
    socktype => 'dgram',

    # Успешное подключение
    on_connected => sub {
        my ( $sock ) = @_;

        # Непосредственная работа с сокетом
        my $socket = IO::Async::Socket->new(
```

```
18              handle => $sock,
19
20              # получен пакет
21              on_recv => sub {
22                  my ( $self,
                        $dgram, $addr
                        ) = @_;
23
24                  print "Получена
                        датаграмма:
                        $dgram\n",
25                  $loop->stop;
26              },
27
28              # ошибка при отправке
                    в сокет
29              on_recv_error => sub
                    {
30                  my ( $self,
                        $errno ) = @_;
31                  die "Ошибка -
                        $errno\n";
32              },
```

```
33          );
34
35          $loop->add( $socket );
36
37          # Отправка сообщения в
               сокет
38          $socket->send( "Привет,
               мир!" );
39      },
40
41      on_resolve_error => sub { die
            "Cannot resolve - $_[0]\n
           "; },
42      on_connect_error => sub { die
            "Cannot connect\n"; },
43  );
44
45  $loop->run;
```

В указанном примере создаётся объект класса `IO::Async::Socket`, который становится наблюдателем за сокетом

соединения. Функция-колбек `on_recv` вызывается каждый раз при получение порции данных (как правило с приходом каждого пакета). `IO::Async::Socket` можно использовать и для работы с сокетом типа `SOCK_STREAM`, в этом случае поток данных обрабатывается дискретно, т.е. по мере поступления порций данных в сокет.

Таймеры

Таймеры позволяют запустить некоторую операцию в заданный момент времени. В примере сетевого сервера таймер мог быть полезен для отслеживания висящих без дела клиентов.

```
1 sub on_stream {
2     my $stream = shift;
3
```

```
    # Запускаем сторожевой таймер
      на 5 секунд
    my $watchdog = IO::Async::
      Timer::Countdown->new(
        delay => 5,

        # Закрываем поток по
          истечении времени
        on_expire => sub {
            $stream->close;
        }
    );
    $watchdog->start();
    $loop->add( $watchdog );

    $stream->configure(
        on_read => sub {
            my ( $self, $buffref,
                $eof ) = @_;

            # Сброс таймера, если
              данные появились
            $watchdog->reset;
```

```
22              ...
23          },
24          on_closed => sub {
25
26              # Остановить таймер,
                  если поток
                  закрылся
27              $watchdog->stop;
28          }
29      );
30      $loop->add( $stream );
31      ...
32  }
```

Помимо `IO::Async::Timer::Countdown` существуют таймеры `IO::Async::Timer::Absolute` (для запуска задачи в указанное время) и `IO::Async::Timer::Periodic` (для периодического запуска задачи).

Сигналы

С помощью `IO::Async::Signal` существует возможность асинхронно обрабатывать поступающие сигналы, выполняя заданный код обработчика сигнала. При обычном способе задания обработки сигнала, через установку `$SIG{NAME}`, может произойти ситуация, когда сигнал придёт в середине выполнения какой-либо важной транзакции, и код обработчика потенциально способен нарушить её. Реализация `IO::Async::Signal` гарантирует выполнение кода обработчика после завершения текущего такта цикла, т.о. происходит синхронизация относительно основного цикла событий.

Допускается установка множества обработчиков для одного сигнала. В этом случае каждый из них будет выполнен, но без

какого-либо фиксированного порядка.

Пример обработки сигнала SIGHUP:

```
use IO::Async::Signal;
use IO::Async::Loop;

my $loop = IO::Async::Loop->new;

my $signal = IO::Async::Signal->
    new(
    name => "HUP",

    # Завершить цикл при получении
        сигнала SIGHUP
    on_receipt => sub {
        print "Stop loop after
            SIGHUP\n";
        $loop->stop;
    },
);

$loop->add( $signal );
```

```
17
18 $loop->run;
```

Возможен также альтернативный вариант создания обработчиков сигналов, через метод `attach_signal` в `IO::Async::Loop`:

```
1 use IO::Async::Loop;
2
3 my $loop = IO::Async::Loop->new;
4 my $sig_id = $loop->attach_signal
    ( "SIGHUP", sub {
5     print "Stop loop after SIGHUP
        \n";
6     $loop->stop;
7 });
8
9 $loop->run;
```

Метод `detach_signal` позволяет удалить обработчик:

```
1 $loop->detach_signal($sig_id);
```

Управление процессами

Одно из важных условий корректной работы цикла обработки событий — неблокируемый код и отсутствие процессороёмких вычислений. В тех случаях, когда избежать этого невозможно, можно прибегнуть к выполнению таких операций в отдельном процессе или треде.

В `IO::Async` существуют несколько способов для запуска и управления дочерними процессами.

IO::Async::Process `IO::Async::Process` позволяет запустить внешнюю программу или код. Модуль выполняет последовательные вызовы `fork()` и `exec()`, если это внешняя команда, или `fork()` и `eval()` в скалярном контексте, если вызывается код.

В случае выполнения внешней программы, существует множество способов для получения stdout, stderr запущенного процесса или передачи данных на stdin дочернего процесса.

Например, простой пример для захвата stdout дочернего процесса:

```
use IO::Async::Loop;
use IO::Async::Process;

my $loop = IO::Async::Loop->new;

my $stdout;
my $process = IO::Async::Process
   ->new(
   command => [ "writing-program"
      , "arguments" ],
   stdout => { into => \$stdout
      },
   on_finish => sub {
      print "The process has
```

```
            finished, and wrote:\n";
        print $stdout;
    }
);

$loop->add( $process );
$loop->run
```

Пример посложнее, для вычисления с помощью консольного калькулятора bc. Выражение передаётся на stdin процесса, а результаты забираются из stdout:

```

use IO::Async::Loop;
use IO::Async::Process;

my $loop = IO::Async::Loop->new;

my $process = IO::Async::Process
    ->new(
    command => [ "bc", "-q" ],

```

```
10      # Формируется pipe, дочерний
            процесс читает stdin и
            пишет на stdout
11      stdio => {
12          via => "pipe_rdwr"
13      },
14
15      on_finish => sub {
16          $loop->stop;
17      },
18  );
19
20  # Чтение данных в родительском
        процессе
21  $process->stdio->configure(
22      on_read => sub {
23          my ( $stream, $buffref )
                = @_;
24          while( $$buffref =~ s
                /^(.*\n)// ) {
25              print "Answer is $1";
26          }
27          return 0;
```

```
28     },
29 );
30
31 $loop->add( $process );
32
33 # Отправка выражений для
       вычислений в bc
34 $process->stdio->write("$_\n")
       for ("2+2", "2+2*2", "(2+2)*2"
        );
35
36 $loop->run
```

На выходе программы получим такой результат:

```
1 Answer is 4
2 Answer is 6
3 Answer is 8
```

Выполнение кода позволяет получить только код завершения процесса:

```
my $process = IO::Async::Process
    ->new(
     code => sub {
         return 42
     },
     on_finish => sub {
         my ($self, $code) = @_;
         printf "Exit code: %d\n",
             $code>>8;
     },
);
...
```

Методы IO::Async::Loop Как и во многих других случаях, создание дочерних процессов продублировано в реализации `IO::Async::Loop`. Например, метод `open_child()` является обёрткой вокруг `IO::Async::Process`:

```
$pid = $loop->open_child(
```

```
    command => [...],
    on_finish => sub { ... }
);
```

Кроме того, присутствует метод `run_child`, который является упрощённым способом вызова внешних процессов, если требуется получить только их вывод:

```
$loop->run_child(
   command => [ "ls", "-1" ]
   on_finish => sub {
      my ( $pid, $exitcode,
         $stdout, $stderr ) = @_;
      my $status = ( $exitcode >>
          8 );
      ...
   },
);
```

IO::Async::PID `IO::Async::PID` позволяет отслеживать завершение работы дочернего процесса. Может быть полезен, если по каким-то причинам вас не устроили все предыдущие методы запуска внешних процессов.

```
use IO::Async::PID;
use IO::Async::Loop;
my $loop = IO::Async::Loop->new;

# Запуск дочернего процесса
my $kid = $loop->fork(
   code => sub {
      print "Сон..\n";
      sleep 10;
      print "Выход\n";
      return 20;
   },
);

print "Запущен дочерний процесс
   $kid\n";
```

```
16
17 my $pid = IO::Async::PID->new(
18    pid => $kid,
19
20    on_exit => sub {
21       my ( $self, $exitcode ) =
            @_;
22       printf "Дочерний процесс %d
             завершился с кодом %d\n
            ",
23          $self->pid, $exitcode
               >>8;
24    },
25 );
26
27 $loop->add( $pid );
28
29 $loop->run;
```

Асинхронный запуск функций с помощью IO::Async::Function и IO::Async::Routine

В разделе, посвящённом `IO::Async::Process` упоминалось, что дочерний процесс может выполнять и фрагмент кода, но функционал этот достаточно бедный, т.к. позволяет вернуть только код завершения процесса из процесса-потомка.

Для асинхронного выполнения функций с возможностью получения результатов выполнения в `IO::Async` созданы специальные классы: `IO::Async::Function` и `IO::Async::Routine`.

IO::Async::Function Объект `IO::Async::Function` представляет собой функцию, которая будет выполняться параллельно ос-

новному процессу программы. Существует возможность выбора модели запуска — это либо fork нового процесса, либо создание нового треда. При первом старте или вызове такой функции происходит запуск одного или нескольких (в зависимости от настроек) процессов/нитей — *рабочих*. Эти рабочие содержат код выполняемой функции и могут многократно вызывать её и возвращать результат обратно в основной процесс, поэтому важно, чтобы функция обладала свойством реентерабельности, т.е. не изменяла своих внутренних структур, что могло бы изменить её поведение при повторных вызовах. Помимо количества рабочих можно задавать и таймаут, после которого неиспользуемый рабочий будет остановлен для экономии ресурсов.

Внутренне `IO::Async::Function` построен на базе `IO::Async::Routine`, о

котором будет рассказано ниже. Основной процесс запускает процесс/нить рабочего и формирует каналы для ввода/вывода. Через входной канал рабочего передаются аргументы функции, а через выходной — возвращаемые данные. Для сериализации/десериализации данных используются функции модуля `Storable`.

Пример функции:

```
use IO::Async::Function;
use IO::Async::Loop;
use Crypt::CBC;

my $loop = IO::Async::Loop->new;

my $cipher = Crypt::CBC->new(-key
    => 'secret', -cipher => '
   Blowfish');

my $function = IO::Async::
   Function->new(
```

```
10
11      # Код шифровщика
12      code => sub {
13          $cipher->encrypt(shift);
14      },
15      model => 'fork',     # форк
16      max_workers => 10,  #
           максимальное число рабочих
17      min_workers => 2,   #
           минимальное число рабочих
18      idle_timeout => 10, #
           выключение неиспользуемых
           рабочих
19                              # по
                                   таймауту
20  );
21
22  $loop->add( $function );
23
24  # Вызов функции
25  $function->call(
26
```

```
27      # Аргументы
28      args => [ "Plain text" ],
29      on_return => sub {
30          my $ciphertext = shift;
31          print "'Plain text'
                encrypted to '".
                $ciphertext ."'\n";
32      },
33      on_error => sub {
34          warn "Все CPU перегреты\n";
35      },
36  );
37
38  $loop->run;
```

Основное назначение таких функций: выполнение каких-либо интенсивных вычислений или блокирующихся операций, которым не требуется хранить информацию о состоянии между вызовами. Т.е. в идеале функции, результат которых зависит только от входных аргументов.

IO::Async::Routine IO::Async::Routine также как и IO::Async::Function позволяет выполнять код во внешнем процессе/нити. Отличие состоит в том, что создаваемый рабочий запускается один раз и производит обмен данными с основным процессом через специальные каналы `IO::Async::Channel`. Такой рабочий может иметь сложную логику, хранить информацию о состоянии в процессе обмена с основным процессом и хорошо подходит для организации потоковой обработки данных.

Рассмотрим пример программы:

```
1 use IO::Async::Routine;
2 use IO::Async::Channel;
3 use IO::Async::Loop;
4
5 my $loop = IO::Async::Loop->new;
6
7 # Каналы для обмена данными
```

```
8  my $input  = IO::Async::Channel->
       new;
9  my $output = IO::Async::Channel->
       new;
10
11 # Функция рабочего процесса
12 my $routine = IO::Async::Routine
       ->new(
13      channels_in  => [ $input  ],
14      channels_out => [ $output ],
15
16      code => sub {
17          my @nums = @{ $input->
                recv };
18          my $ret = 0; $ret += $_
                for @nums;
19          $output->send( \$ret );
20      },
21
22      on_finish => sub {
23          print "Рабочий завершил
                работу - $_[-1]\n";
24          $loop->stop;
```

```
    },
);

$loop->add( $routine );

# Отправка данных рабочему
$input->send( [ 10, 20, 30 ] );

# Функция-колбек для ёприма
    данных
$output->recv(
    on_recv => sub {
        my ( $ch, $totalref ) =
            @_;
        print "10 + 20 + 30 =
            $$totalref\n";
        $loop->stop;
    }
);

$loop->run;
```

В примере создаются два канала `IO::Async::Channel` для обмена данными с рабочим. Данные объекты ориентированы на использование исключительно совместно с `IO::Async::Routine`, поэтому их не требуется вручную добавлять в цикл обработки событий, их управлением займётся сам `IO::Async::Routine`. Для этого требуется указать их в списках каналов `channels_in` и `channels_out`, т.е. определить направление записи в канал: входной и выходной каналы для рабочего соответственно.

Канал позволяет передавать только ссылки. Все такие ссылки в момент передачи сериализируются с помощью `Storable`, поэтому после того как объект сериализован, любые изменения в нём уже не будут видны тому процессу, в который он был передан.

Канал имеет два метода: `send()` и `recv()` для передачи и получения данных в/из канала соответственно. Выполнение `send()` происходит без блокировки. `recv()` блокируется в том случае, если вызван без указания колбека `on_recv` и возвращает полученные данные.

Изменения в файловой системе

Модуль `IO::Async::File` позволяет создавать уведомители об изменениях в файлах и каталогах. Наблюдать можно за изменениями любой характеристики файла, доступной в вызове `stat()`: время создания/доступа/модификации, размер, права доступа, владелец и другие.

```
1 use IO::Async::File;
2 use IO::Async::Loop;
3
```

```
my $loop = IO::Async::Loop->new;

# Наблюдаем за изменением времени
    модификации
# файла исходного кода программы
my $file = IO::Async::File->new(
   filename => __FILE__,
   on_mtime_changed => sub {
      my ( $self, $new_mtime,
         $old_mtime ) = @_;
      print "Время модификации
         файла изменилось",
            "с mtime $old_mtime
               на $new_mtime\n";
   }
);

$loop->add( $file );

$loop->run;
```

Так же есть возможность наблюдать за всеми характеристиками одновременно, указав функцию-колбек `on_stat_changed`, которая вызывается при изменении любой из характеристик. В качестве параметров передаются объекты класса `File::stat` с текущим и предыдущим состояниями файлового дескриптора.

Внутри модуль реализован на простом периодическом вызове функции stat() на файле каждые 2 секунды. Регулировать интервал опроса можно с помощью параметра `interval` при создании объекта.

ООП подход при использовании IO::Async

Пример сетевого сервера, рассмотренный в начале статьи, можно переделать с применением ООП подхода. Метод `listen` `IO::Async::Loop` возвращает объект `IO::Async::Listener`. Мы можем создать свой собственный класс `MyListener`, который будет наследовать методы `IO::Async::Listener` и будет иметь свой обработчик события `on_stream`:

```
use IO::Async::Loop;

my $loop = IO::Async::Loop->new;

# Создание объекта уведомителя MyListener,
# производного от IO::Async::Listener
my $myl = MyListener->new;
```

```
8
9  $loop->add($myl);
10
11 $myl->listen(
12     family   => "inet",
13     socktype => "stream",
14     service  => 1234,
15     on_resolve_error => sub {
          print STDERR "Cannot
          resolve - $_[0]\n"; },
16     on_listen_error  => sub {
          print STDERR "Cannot
          listen\n"; },
17 );
18
19 $loop->run;
20
21 # Собственный модуль обработки
      потока
22 package MyListener;
23
24 # Наследуем от класса IO::Async::
      Listener
```

```
25 use base qw( IO::Async::Listener );
26
27 # Реализуем метод on_stream, вызываемый при подключении клиента
28 sub on_stream {
29     my ($self, $stream) = @_;
30
31     $stream->configure(
32         on_read => sub { 0 },
33         close_on_read_eof => 0
34     );
35
36     # Добавляем поток в цикл
37     $self->loop->add( $stream );
38
39     # Возвращается Future-объект для операции чтения потока до EOF
40     my $f = $stream->read_until_eof;
41
```

```
42      # Функция-колбек при
            завершении операции
43      $f->on_done(sub {
44          my ($buf, $eof) = @_;
45
46          while( $buf =~ s/^(.*)\n
                // ) {
47              $stream->write(scalar
                    reverse "\n".$1);
48          }
49
50          $stream->close_when_empty
                () if $eof;
51          return 0;
52      });
53  }
54
55  1;
```

Такой подход позволяет получить логичный набор классов для каждой решаемой задачи и избежать появления

огромных простыней кода с вложенными функциями-колбеками.

Заключение

Хочется отметить, что несмотря на то, что это *ещё один фреймворк для СОП*, он, тем не менее, заслуживает серьёзного внимания уже благодаря полноте реализуемых задач, а также уникальной интеграции событийно-ориентированного программирования с обещаниями и объектно-ориентированным подходом к компоновке программы.

Как уже было отмечено, цикл `IO::Async` может использовать множество реализаций мультиплексоров под капотом, что делает его похожим на `AnyEvent`. В экоси-

стеме модулей `IO::Async` есть также модули для асинхронного разрешения имён с использованием системного резолвера, модуль http-клиента, прозрачная поддержка SSL и т.д. `IO::Async` можно использовать в веб-приложениях `Mojolicious`. Всё это позволяет строить широкий класс приложений на его основе.

■ *Владимир Леттиев*

5 Обход дерева директорий на Perl и Haskell (часть 1)

Perl, как известно, не единственный язык программирования, однако он сочетает в себе множество подходов и стилей. В данной статьей рассматривается сравнение реализаций обхода дерева директорий на Perl и Haskell

Функциональное программирование неизбежно проникает в массы. Даже такие безнадежные языки, как Java и C++ больше не могут сопротивляться здравому смыслу и обзаводятся анонимными функциями (или «лямбда-выражениями»).

Что уж говорить о Perl, в котором анонимные функции (`$subref = sub { ... }`) и классические комбинаторы вроде `map` и

`grep` уже давно не новость.

С другой стороны, наличие в языке функциональных элементов побуждает программистов на этом языке считать «Зачем мне функциональные языки, я и на своем языке могу писать в функциональном стиле».

И действительно, Perl достаточно выразителен, чтобы на нем можно было реализовать множество функциональных идиом, что блестяще демонстрирует книга «Higher-order Perl» авторства Mark Jason Dominus.

Все же, есть ли какая-то выгода от написания кода на Haskell по сравнению с Perl? Чтобы это выяснить, я решил взять пример из Higher-Order Perl и переписать его на Haskell.

Сразу хочу предупредить, что целью этой статьи не является обучить читателя языку Haskell или даже добиться полного понимания приведенных здесь примеров. Моя задача — продемонстрировать (довольно продвинутые) возможности и идиомы Haskell и подтолкнуть читателя к изучению этого языка.

Не стоит бояться, если вы не понимаете, что за синтаксис используется в каком-то конкретном примере. Главное видеть общую картину и интуитивно понимать, что делает код.

Пример, который мы будем рассматривать, описывается в главе 1.6 и представляет собой рекурсивный обход дерева директорий. Задача здесь — не просто обойти дерево с конкретной целью, а написать «обобщенный» обход, который можно

использовать, например, для:

- получения дерева в виде структуры данных;
- поиска файла в дереве;
- печати списка директорий;
- поиска поломанных символических ссылок.

и т.д.

Чтобы наша функция могла совершать разные действия, мы параметризуем ее двумя коллбеками — для файлов и директорий. Вот функция `dir_walk`, написанная Марком на Perl:

```
# From Higher-Order Perl by Mark Dominus, published by Morgan Kaufmann Publishers
# Copyright 2005 by Elsevier Inc
```

```
# LICENSE: http://hop.perl.plover.com/LICENSE.txt

sub dir_walk {
  my ($top, $filefunc, $dirfunc) = @_;
  my $DIR;

  if (-d $top) {
    my $file;
    unless (opendir $DIR, $top) {
      warn "Couldn't open directory $code: $!; skipping.\n";
      return;
    }

    my @results;
    while ($file = readdir $DIR) {
      next if $file eq '.' || $file eq '..';
      push @results, dir_walk("
```

```
            $top/$file", $filefunc,
            $dirfunc);
20      }
21      return $dirfunc->($top,
           @results);
22    } else {
23      return $filefunc->($top);
24    }
25  }
```

Вот довольно близкая трансляция этого кода на Haskell:

```
1  import System.FilePath
2  import System.Directory
3
4  dir_walk :: FilePath -> (FilePath
       -> IO a) -> (FilePath -> [a]
     -> IO a) -> IO a
5  dir_walk top filefunc dirfunc =
     do
6    isDirectory <-
        doesDirectoryExist top
```

```
7
8     if isDirectory
9       then do
10        files <-
             getDirectoryContents top
11        let nonDotFiles = filter (
             not . (`elem` [".", ".."
             ])) files
12        results <- mapM (\file ->
             dir_walk (top </> file)
             filefunc dirfunc)
             nonDotFiles
13        dirfunc top results
14      else
15        filefunc top
```

Поскольку это может быть вашим первым знакомством с Haskell, давайте потратим немного времени и сравним версии на Perl и на Haskell.

Первое, что бросается в глаза в коде на

Haskell — сигнатура типа (строка 4). Сигнатуры типов в Haskell в большинстве случаев, включая этот, опциональны — компилятор достаточно умен, чтобы самостоятельно вывести тип. Тем не менее, обычно программисты выписывают сигнатуры, потому что это дополняет документацию и упрощает чтение и понимание кода. Никто не возбраняет сначала написать тело функции, а затем спросить у компилятора сигнатуру и скопировать ее в код, хотя при написании сколь-нибудь сложных функций сигнатуру обычно пишут еще до кода самой функции — это позволяет сразу ответить себе на вопрос: «Что эта функция должна делать?».

Давайте попробуем расшифровать эту сигнатуру. Когда мы видим последовательность типов, разделенных стрелочками, мы должны интерпретировать это как

функцию, у которой последний тип является типом возвращаемого значения, а все предыдущие — типами аргументов. Как вы догадываетесь, такая нотация возникла не просто так — на то есть глубокие причины, но мы сейчас на них останавливаться не будем.

Итак, `dir_walk` — функция, возвращающая `IO a` и принимающая три аргумента, типы которых `FilePath`, `FilePath -> IO a` и `FilePath -> [a] -> IO a` соответственно.

Здесь `a` — переменная типа, обозначающая тип результата, который мы хотим получить в результате обхода. Например, если мы хотим узнать суммарный размер директории, то при вызове функции вместо `a` будет подставлен тип `Integer`.

Чем отличается `IO a` от `a`? `IO a` — тип *действий*, которые, будучи исполненными, возвращают значение типа `a`. Действие можно исполнить (и получить его результат) только в контексте другого действия, используя следующий синтаксис:

```
action_b :: IO b
action_b = do
  result_of_a <- action_a :: IO a
  -- отсюда и до конца do-блока
     result_of_a имеет тип a
  ...
```

Функцию, у которой тип возвращаемого значения имеет вид `IO a`, можно упрощенно понимать как функцию, возвращающую `a` и при этом совершающую какие-либо побочные эффекты (ввод-вывод, работа с файловой системой и сетью, и т.д.). Разница в том, что, в отличие от Perl, соответствующие действия не выполняются каждый раз,

когда вызов функции встретился в коде. Например, мы можем написать

```
action_b :: IO b
action_b = do
  let x = action_a :: IO a
  -- отсюда и до конца do-блока x
       имеет тип IO a
  ...
```

Здесь мы дали действию `action_a` альтернативное имя `x`, но от этого само действие не выполнилось, и его результата мы не получили. Чтобы выполнить действие как часть другого действия, мы должны использовать синтаксис `<-`, как показано выше.

В скомпилированной программе действие выполняется только если оно (прямо или косвенно) является частью главного действия программы — действия под именем

`main` (по аналогии с языком C).

Возвращаясь к нашей сигнатуре, мы видим, что оба коллбека — для файлов и для директорий — возвращают IO-действия, и ожидается, что эти действия будут выполнены в процессе обхода дерева.

Перейдем теперь к собственно коду функции `dir_walk`. На строке 6 мы видим пример работы с действиями с использованием уже знакомого нам синтаксиса `<-`. Дело в том, что функция `doesDirectoryExist` (аналог оператора `-d` из Perl) имеет тип `FilePath -> IO Bool`, поскольку для получения ответа требуется доступ к файловой системе. Строчка `isDirectory <- doesDirectoryExist top` является аналогом `$isDirectory = -d $top;` — она выполняет действие `doesDirectoryExist top :: IO Bool` и записывает результат

в переменную `isDirectory`. Однако, если бы мы наивно написали

```
1 if doesDirectoryExist top
2   then ...
3   else ...
```

то получили бы ошибку типов, потому что конструкция `if` ожидает выражение типа `Bool`, а не `IO Bool`.

Аналогично, в строке 10 выполняется действие `getDirectoryContents top` типа `IO [FilePath]`, и результат записывается в переменную `files`.

В строке 11 мы отбрасываем «виртуальные» директории `.` и `..` (по аналогии со строкой 18 из кода на Perl). Обратите внимание, что функция `filter` не требует выполнения каких-либо побочных эффектов, поэтому мы используем синтаксис

`let`, а не `<-`. В общем случае, чтобы выяснить, требует ли функция выполнения действий, надо посмотреть на ее тип. Так, функция `filter` имеет тип

```
1 filter :: (a -> Bool) -> [a] -> [
      a]
```

В строке 12 мы рекурсивно вызываем функцию `dir_walk`, по аналогии со строкой 19 кода на Perl, для каждого файла или поддиректории в нашей директории. Для этого используется функция

```
1 mapM :: (a -> IO b) -> [a] -> IO
      [b]
```

которая применяет функцию к каждому элементу списка `nonDotFiles`, выполняет получившиеся действия и объединяет результаты в новый список.

Наконец, в строках 13 и 15 мы вызываем коллбеки, подобно строкам 21 и 23 кода на Perl. Здесь мы не использовали синтаксис `<-`, чтобы записать результат в переменную — вместо этого, результат действия становится возвращаемым значением нашей функции, прямо как в Perl. Альтернативно, блок вида

```
1 do
2   ...
3   dirfunc top results
```

можно заменить на

```
1 do
2   ...
3   result <- dirfunc top results
4   return result
```

Здесь `result` имеет тип `a` (поскольку это результат действия типа `IO a`), а функция `return` (тип которой `a -> IO a`) делает

из значения «тривиальное» действие, которое возвращает результат без каких-либо побочных эффектов.

Внимательный читатель мог заметить, что в приведенном коде на Haskell отсутствует обработка ошибок. Например, если по какой-то причине `getDirectoryContents` не сможет выполниться успешно, будет выброшено исключение, которое либо будет перехвачено выше по стеку вызовов, либо приведет к завершению программы. В таком случае поведение наших реализаций на Haskell и на Perl будет разным.

Как мы видим, код на Haskell лишь слегка отличается от функционального кода на Perl. Различия обсусловлены тем, что в Haskell функции осуществляют доступ к файловой системе не напрямую, а через *действия*, а также мелкими различиями в

стандартных библиотеках наших языков.

На первый взгляд, морока с «действиями» не оправдывает себя и лишь запутывает. Однако, в следующем выпуске журнала мы познакомимся с типами действий помимо `IO`, которые позволят решить слегка модифицированную задачу по обходу дерева гораздо проще, чем без них.

Чтобы узнать больше о Haskell и попробовать его в действии, рекомендую слайды со встречи одесской группы пользователей Haskell. Там вы найдете ссылки на учебные материалы, инструкции по установке и другую полезную информацию.

Если вам кажется, что вы не поняли что-то важное в этой статье, пожалуйста, напишите мне, и я постараюсь прояснить это в следующем номере. Дальше будет сложней и

интересней!

■ *Роман Чепляка*

6 Обзор CPAN за декабрь 2013 г.

Рубрика с обзором интересных новинок CPAN за прошедший месяц.

Статистика

- Новых дистрибутивов — 193
- Новых выпусков — 795

Новые модули

- SQL::Composer

Модуль для генерации SQL-запросов и преобразования полученных массивов

данных в структуру, удобную для использования в шаблонизаторах. Поддерживается стандартный набор CRUD-запросов и вложенные JOIN. Похож на модуль `SQL::Abstract`, но проще и имеет бонус в виде парсера возвращаемых данных.

- Log::Declare

Модуль для ведения логов приложения, позволяет задавать уровень информативности (ошибки, предупреждения, отладка и т.д), категорию сообщений и удобный синтаксис для вывода ссылок и структур. Особенность модуля в том, что он не даёт никаких накладных расходов в случае, если все или какие-то определённые уровни информативности отключены. Типичная запись `$log->debug()`, в случае отключенного режима отладки тем

не менее приводит к вызову метода. В `Log::Declare` запись

```
1     debug "debug message" [
          category];
```

разворачивается в

```
1     debug && $Log::Declare::
          logger->log('debug', ['
          category'], "debug message
          ");
```

Таким образом, выражение после && никогда не вычисляется.

- SQ

Простенький модуль, полезный, когда вам нужно использовать символ одиночной кавычки в однострочнике. Обычно

для это требуется применять страшное экранирование '\'':

```
$ perl -e 'print "'\'' - single quote"'
' - single quote
```

Модуль SQ предоставляет переменные $s, $q и $sq, которые содержат одиночную кавычку:

```
$ perl -MSQ -e 'print "$q - single quote"'
' - single quote
```

- Sub::Deprecated

Sub::Deprecated позволяет использовать атрибут :Deprecated для методов, которые устарели. В случае использования таких методов будет генерироваться

предупреждение.

Обновлённые модули

- YAML 0.88

В новой версии YAML возвращена совместимость с Perl 5.8.

- ack 2.12

Обновлённая версия утилиты для поиска ack исправляет проблему с безопасностью. При использовании опций `--pager`, `--regex` и `--output` ack больше не загружает ackrc-файлы, которые находятся в каталогах проектов, где происходит поиск.

- Data::Validate::IP 0.22

Новый релиз модуля для проверки валидности IP-адресов содержит обновлённую документацию. Также обновлена информация о тестовых сетях и зарезервированных сетях IPv6.

- Socket6 0.25

После пятилетнего перерыва обновился модуль `Socket6`, который добавляет константы и функции для работы с IPv6. Исправлены множество багов, долгое время висевших без какой-либо реакции со стороны автора. Хотя на данный момент IPv6 отлично работает и в модуле `Socket`, а также в `IO::Socket::IP`, данное обновление очень важно хотя бы потому,

что модуль с таким именем должен, по меньшей мере, быть рабочим, чтобы избежать истерик, что в Perl не работает IPv6, а значит он мёртв

- Thrall 0.0200

Обновился PSGI-веб-сервер Thrall, использующий нити для обработки входящих запросов. В новом релизе появилась поддержка HTTP 1.1, IPv6, SSL.

- Markdent 0.23

Модуль `Markdent`, предназначенный для обработки markdown-текстов парсером, генерирующим события. Помимо привычного преобразования в HTML-формат

возможно назначение обработчиков на какие-либо события (найдена ссылка, выделение, заголовок и т.д.) для специфической обработки текста. В новой версии добавлена поддержка «ленивых» и «пьяных» списков.

- XML::SAX::Writer 0.54

В новом релизе `XML::SAX::Writer` исправлена ошибка при экранизации символов создаваемого XML-документа. В случае если использовалось несколько объектов `XML::SAX::Writer` с разным регулярным выражением для экранирования, самое первое из них кешировалось и использовалось для всех объектов.

- ObjectDB 3.03

В декабре на CPAN появилась новая мажорная версия легковесного ORM `ObjectDB`. Многие идеи взяты из проекта `Rose::DB::Object`, но основной упор сделан на простоту и прямолинейность.

- Dancer2 0.11

Новый релиз веб-фреймворка `Dancer2` включает несколько важных исправлений в коде и документации. Переработана реализация ключевых слов `halt`, `redirect` и `forward` для немедленного возврата их роута или хука. Чтоб не загрязнять `@INC`, теперь запускаемый скрипт ответственен за подключение каталога `./lib`, если в этом есть необходимость.

- HTML::FormFu 1.00

Вышел мажорный релиз популярного фреймворка для создания и проверки HTML-форм с внушительным списком изменений и исправлений.

- Test::Continuous 0.74

Обновлён модуль `Test::Continuos` для выполнения тестирования в процессе разработки. В новой версии исправлена проблема с Perl 5.18.

- Sereal 2.010

Вышел новый мажорный релиз модуля для быстрой (де-)сериализации данных. В новой версии введена поддержка v2-протокола, исправлены редкие случаи утечек и краха приложения.

■ *Владимир Леттиев*

7 Интервью с Tokuhiro Matsuno

Tokuhiro Matsuno — автор plenv, Amon2, Furl, Minilla и многих других популярных модулей на CPAN.

Когда и как начал программировать?

Мой отец дал мне первый компьютер, когда мне было десять лет. Это было в 1994 году и он назывался карманный компьютер Sharp PC (http://fr.wikipedia.org/wiki/Sharp_PC-1262). Можно было использовать 24 символа для отображения и он понимал язык BASIC. Это был для меня первый опыт программирования.

Когда я перешел в старшую школу, то начал посещать клуб по персональным компьютерам. Тогда я изучал язык HSP (извест-

ный только в Японии). Он был эволюционным продолжением BASIC. Это был язык, специально разработанный для написания игр. Однако, я так и не написал ни одной игры, я разработал библиотеку.

Когда я поступил в технический колледж, мой отец дал мне Visual Basic. И я продолжил изучать программирование. На Visual Basic можно было писать для Windows. Было очень просто писать GUI-приложения. Однако, я разработал библиотеку и так и не написал ни одного GUI-приложения.

В то время в колледже было много книг по программированию, я много почерпнул из них. Тогда я использовал Ruby и Python и думал, что Perl устарел. Это было около 2000–2005 (в 2005 году в Японии много книг было опубликовано по Ruby).

Однако, в то время Perl-хакеры, включая miyagawa, играли большую роль в Японии. И я захотел написать что-нибудь и на Perl.

Теперь я пишу модули на Perl.

Какой редактор используешь?

В основном я использую vim. Есть три условия, которые я выдвигаю редактору:

1. Он работает в терминале.
2. Его можно расширять.
3. Он хорошо работает с японским.

Когда я пытался найти подходящий редактор, были только vim и emacs.

Но когда я использую emacs, у меня постоянно болит мизинец. Поэтому я и не

использую emacs.

Когда и как познакомился с Perl?

Меня взяли на работу, когда мне было двадцать лет. В компании использовали Perl, и я его начал учить. В то время я уже знал Python и Ruby. Тогда я думал: «Perl очень странный язык». Однако, с течением времени, я все больше знакомился с Perl и в конце концов решил, что это неплохой язык.

С какими другими языками интересно работать?

Я использую разные языки. Для iPhone-приложений использую Objective-C. Когда нужен очень быстрый модуль, использую C или C++. В довесок к этому могу писать на Python, Ruby, PHP и т.д. Думаю, что у каждого языка есть своя область приме-

нения. Но в основном я пишу на Perl и C.

Какое, по-твоему, самое большое преимущество Perl?

Perl замечательный язык.

Мне очень нравится сообщество, которое борется за обратную совместимость. Программа, написанная для Perl 5.8, работает практически без изменений и на Perl 5.18. Это удивительно. Я не хочу переписывать свои программы после каждого обновления.

В Perl 5 мне нравится управление памятью с помощью подсчета ссылок. В языках, которые используют «алгоритм пометок», достаточно сложно добиться оптимизаций. Но было бы хорошо иметь в Perl 5 этот вид

сборки мусора как опцию, по аналогии с Python.

Также мне нравится лексическая область видимости, управляемая `my`. Явное объявление переменной часто снижает вероятность ошибок.

Удивительна и CPAN-культура. Всегда есть документация, и большинство модулей имеют историю изменений.

Какой, по-твоему, важной особенностью должны обладать языки будущего?

Это сложный вопрос. У меня нет на него хорошего ответа.

Что думаешь о будущем Perl?

Думаю, что разработка Perl 5 продолжится.

Perl 6 наконец можно будет использовать на практике. И оба языка будут дополнять и развивать друг друга.

Откуда столько энергии для написания и поддержки такого количества CPAN-модулей?

Я занимаюсь разработкой программного обеспечения. Загружаю многие связанные с работой модули на CPAN. Некоторые из них разрабатываю как хобби. Мне нравится писать программы, которые можно повторно использовать.

Расскажи об Amon2. Чем он отличается от других Perl-фреймворков для веб?

Amon2 очень простой, надежный веб-фреймворк общего назначения.

Отличия от Mojolicious

Mojolicious неплох, и мне нравится сам подход. К сожалению, там не сохраняется обратная совместимость. В Amon2 же наоборот. Мне кажется, что ломать обратную совместимость можно только с изменением названия. Когда я решу что-либо серьезно изменить, я выпущу Amon3.

Отличия от Catalyst

Catalyst зависит от Moose, а Amon2 — нет. Это потому, что я хочу, чтобы мои приложения загружались быстро.

Отличия от Dancer

Практически нет никаких отличий между Amon2 и Dancer, включая Dancer2.

Насколько стабильны биндинги для UnQLite? Используешь ли в продакшене? Что думаешь о самой базе данных?

UnQLite очень практична. UnQLite.pm уже можно пользоваться (мне пока не довелось использовать в продакшене).

Объясню, для чего я написал эту обертку.

Мне нужно было практичное хранилище с доступом по ключам. В ядре Perl есть GDBM_File, но к сожалению он устанавливается не для всех операционных систем. Его нельзя поставить с CPAN, что делает его неудобным в использовании.

В то же время UnQLite.pm обладает следующими преимуществами:

1. Можно установить с CPAN.

2. Можно использовать на любой ОС.
3. Обладает большой скоростью.

Складывается впечатление, что ты придерживаешься подхода, когда используется как можно меньше зависимостей. Почему это важно?

Мне не нравится скрипты, которые медленно стартуют. Если брать в общем, то чем меньше зависимостей, тем скорость запуска скрипта увеличивается. Поэтому мне кажется, что это правильное направление.

Есть также и другой ответ.

Программное обеспечение это как мультипликация. Вся программа превратится в мусор, если один из ее компонентов станет мусором. Чем меньше зависимостей, тем меньше вероятность, что поддержка

приложения усложнится в будущем.

Почему написал Minilla?

Меня не устраивала скорость запуска dzil. Мне не нравится, когда что-то долго запускается. Я бы не написал Minilla, если бы dzil использовал Moo.

И вообще «я просто хотел это написать».

Почему стоит использовать `plenv` вместо `perlbrew`?

Miyagawa очень хорошо описал причины:

`plenv` целиком написан на bash (кроме perl-build, который загружает дистрибутивы perl с PAUSE и запускает patchperl), он кладет в переменную `PATH` небольшие скрипты, которые находят нужный perl и

инсталированные программы, и запускает их.

И это все, нет никакой магии с функциями шелла, которые меняют `PATH` перед запуском программ, нет необходимости переключать perl, исключена возможность запуска другой версии perl по ошибке.

`plenv` позволяет переключать perl тремя способами: через переменную окружения `PLENV_VERSION`, файлом `.perl-version` в текущей директории и глобальной настройкой в `~/.plenv/version`.

Больше текста по ссылке (на английском): http://weblog.bulknews.net/post/58079418600/plenv-alternative-for-perlbrew.

Есть планы по развитию `pvip` и `seis`? Можешь объяснить, что это вообще такое?

PVIP это парсер для Perl 6. Он написан на C и поддерживает около 50% спецификации Perl 6. SEIS это транслятор из Perl 6 на Perl 5. В данный момент это всего лишь игрушка.

PVIP и SEIS довольно интересные проекты. Однако, мне нужно было время на другие вещи и разработка приостановилась.

Есть ли у тебя какие-нибудь не очень известные модули, которые были бы полезны многим программистам?

`Test::Power` очень интересен.

Когда пишете подобный код на Perl:

```
use Test::Power;

sub foo { 4 }
expect { foo() == 3 };
expect { foo() == 4 };
```

То при запуске получаете:

```
not ok 1 – L12 : expect { foo()
    == 3 };
#   Failed test 'L12 : ok { foo()
    == 3 };'
#   at foo.pl line 12.
# foo()
#    => 4
ok 2 – L13 : expect { foo() == 4
    };
1..2
# Looks like you failed 1 test of
    2.
```

`Test::Power` печатает процесс выполнения. Больше не нужно вручную вставлять бесполезные конструкции типа `diag()`.

Участвуешь в организации YAPC::Asia?

Нет, я только докладчик.

Где сейчас работаешь? Требуют ли проекты высокой скорости выполнения?

Я работаю в компании, которая разрабатывает приложения для смартфонов. Я пишу серверную часть на Perl. Некоторыми приложениями нашей компании пользуются более миллиона человек. Все это работает на Amon2.

Стоит ли сейчас советовать молодым людям изучать Perl?

Это сложный вопрос. У меня нет на него хорошего ответа.

Вопросы от читателей

Docopt вполне себе хорошая имплементация. Но почему же все в одном файле?

Потому что в версии на Python так и было. `Docopt.pm` был транслирован с исходника на Python. При транслировании кода, очень важно делать это построчно. Поэтому, я так и сделал. Сейчас, когда перенос кода завершен, его можно разделить. Но у меня нет мотивации это делать. Принимаю патчи :)

Ищешь ли на CPAN прежде чем писать новый модуль?

Я всегда захожу на http://search.cpan.org прежде чем писать новый модуль. Глупо переписывать что-либо. Однако, я уже несколько раз загружал модули с пересекающимся функционалом. Это потому, что оригинальные модули не удовлетворяли мои нужды.

Обычно я загружаю альтернативную реализацию по следующим причинам:

1. Качество довольно низкое.
2. Слишком много зависимостей.
3. Мне не нравится дизайн.

Почему начал писать `tora` и почему бросил?

Я хотел написать новый современный язык программирования, очень похожий на Perl 5.

`tora` это язык обладающий следующими свойствами:

1. В отличие от Perl, есть сигнатуры функций.
2. Все — объект.
3. Видимость имен, аналогичная Perl 5.

Разработка TORA была прервана из-за нескольких причин, таких как Amon2 и других. У меня были другие интересные проекты. Более того, мне нужно было больше времени для изучения Perl 6, и я разрабатывал SEIS и PVIP как побочные вещи.

В прошлом месяце я возобновил разработку TORA и надеюсь поработать над ним следущей весной.

■ *Вячеслав Тихановский*

8 Perl Golf

Perl Golf — это соревнование по поиску Perl-кода наименьшего размера (меньше всего символов), который решает заданную задачу. В этом выпуске будет сделан обзор решений предыдущего задания — «Искусственный интеллект», торжественно оглашено имя победителя и предложена новая головоломка.

«Искусственный интеллект»

Создать более или менее приличного робота на основе информации о четырёх соседних клетках оказалось нереально. Любым алгоритмам поиска пути, как например, A* требуется информация обо всех клетках до цели. Поэтому ни один робот

не справлялся с задачей, если на пути встречалось препятствие хитрой формы.

С другой стороны, задача по оптимизации размера кода робота становилась гораздо важнее и интереснее. В этом отношении те, кто следил за отправленными решениями, наблюдали захватывающее сражение между роботами Сергея Можайского technix и Василия Короля vakorol.

Рассмотрим окончательные варианты решения. Решение, которое сделал Василий Король, имеет длину 90 байт. Отформатированный вариант:

```
#!perl -apl
for $j ( 0 .. 3 ) {
    $_ = $j % 2 ? "0 $x" : "$x 0"
      if $F[$j] ne '#'
      && $F[ 4 + $j % 2 ] * ( $x
          = $j % 3 > 0 || -1 ) > 0
```

```
                | /#/;
}
$| = 1
```

Опция компилятора `-p` задаёт режим, в котором тело скрипта оборачивается в цикл while и происходит построчное считывание с `STDIN` и вывод значения переменной `$_` на `STDOUT`. Опция `-a` совместно с `-p` выполняет команду split на каждую полученную строку и записывает результат в массив `@F`. Опция `-l` производит автоматическую обработку конца строк, выполняя `chomp` у входных строк и добавляя перенос строки при выводе `$_`.

Выражение `$| = 1` в конце скрипта включает autoflush для `STDOUT`, заставляя perl делать вывод при каждой записи на вывод, отключая буферизацию.

Цикл `for` обходит четыре значения массива `@F`, в которых сохранены значения окружающих голову змеи клеток. На каждой итерации происходит попытка присвоения окончательного варианта смещения, при условии выполнения сложного условия. Если индекс массива чётный, то перемещение будет по вертикали, если нет — по горизонтали. Направление определяется из условия `$x = $j % 3 > 0 || -1`, то есть в сторону роста координат смещение положительное, а в сторону уменьшения координат отрицательное. Смещение будет зафиксировано при условии, что в текущей клетке нет препятствия, а также если смещение уменьшит расстояние до яблока.

Решение Сергея Можайского имеет длину 103 байта. Отформатированный вариант:

```
#!perl -alp
```

```
2 for $i ( 0 .. 3 ) {
3     $| = @p = ( $i < 3 ? $i — 1 :
          0, $i ? 2 — $i : 0 );
4     $_ = $F[$i] eq '#' ? next : "
         @p";
5     last if grep $_ * pop @p > 0,
          @F[ 5, 4 ];
6 }
```

Данный алгоритм схож с описанным выше. В строке 3 в массиве @p задаётся смещение в зависимости от направления текущей просматриваемой клетки. В следующей строке происходит присвоение $_ строчного значения массива, если в текущей клетке нет препятствия, иначе происходит следующая итерация. Затем следует проверка: если смещение уменьшит расстояние до яблока, то дальнейшие итерации не требуются.

Как обычно, за несколько минут до конца конкурса, то есть в тот момент, когда другие люди открывали шампанское, перед новогодним боем курантов, Иван Крылов aitap присылает своё решение длиной в 163 байта. Отформатированный вариант:

```
#!/usr/bin/perl -pal
$| = @_ = map $F[ 4 + $_ % 2 ] > 0 ? $_ : ( $_ + 2 ) % 4,
  ( abs $F[5] > abs $F[4] ) ? ( 1, 2 ) : ( 2, 1 );
@_[ 2, 3 ] = map { ( $_ + 2 ) % 4 } @_[ 1, 0 ];
($a) = grep $F[$_] ne '#', @_;
$_ = "-" x !( $a % 3 ) . 1;
$_ = $a % 2 ? "0 $_" : "$_ 0";
```

Если кто-то поймёт, как это работает, прошу в комментариях к статье рассказать — это будет отличной головоломкой. Особенность данной змейки в том, что она при

перемещении к яблоку может двигаться по диагонали (точнее, по ломаной линии). За это как раз отвечает условное выражение `( abs $F[5] > abs $F[4] )?`.

Итоги

Проведя тестирование, были получены следующие результаты:

```
example.pl: length=541, path= 60, fights= 86
  aitap.pl: length=163, path= 60, fights= 42
technix.pl: length=103, path= 60, fights= 76
vakorol.pl: length= 90, path= 60, fights= 78

And the oscar goes to vakorol.pl
```

В тестировании была выполнена проверка на длину пути, который проходит змейка. Все роботы следуют по кратчайшей дистанции при отсутствии препятствий. Проверка на набранные очки при многократном запуске поединков роботов показала, что каких-то явных отличий между ними нет, поэтому очки даны лишь для информации, без влияния на результат.

Таким образом, в этом конкурсе у нас побеждает Василий Король, с решением всего в 90 байт. Поздравляем Василия с победой! Огромная благодарность Сергею и Ивану за участие. Хочется также особо отметить работу Ивана за запутанность алгоритма и внешне отличающееся от остальных поведение змейки.

■ *Владимир Леттиев*