Pragmatic Perl 4

pragmaticperl.com

Выпуск 4. Июнь 2013

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

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

Авторы статей:

- Олег Алистратов (ali)
- Владимир Леттиев (crux)
- Олег Фиксель
- Дмитрий Шаматрин (justnoxx)

Корректор: Андрей Шитов (ash)

Выпускающий редактор: Вячеслав Тихановский (vti)

Ревизия: 2014-11-29 23:28

Оглавление

1 От редактора

Встречаем лето очередным номером журнала!

В прошлом выпуске была анонсирована кампания по сбору средств для разработки нового функционала для `Pinto`. Кампания была завершена успешно, и средств собрано даже немного больше, чем необходимо. Спасибо всем, кто откликнулся! Jeffrey Thalhammer написал по этому поводу пост.

Конференция YAPC::Europe 2013 в Киеве все ближе и ближе. Отличная возможность познакомиться и пообщаться с известными Perl-программистами, в том числе и с автором языка — Ларри Уоллом. В этом номере читайте интервью с организатором конференции Андреем Шитовым.

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

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

Приятного чтения.

■ *Вячеслав Тихановский*

2 Сортировка в Perl

Если для сравнения элементов массива используется нетривиальная функция, сортировку массива можно ускорить, кэшируя ключи сортировки или подобрав функцию попроще.

Основы

Каждый программист сталкивается с задачей упорядочения массивов данных. Иногда по пять раз еще до завтрака. Хорошо, если данные сразу приходят упорядоченными из внешнего источника (базы данных, файловой системы и т.п.), если нет — тоже не страшно, можно в самой программе отсортировать.

Существует несколько десятков алгоритмов сортировки. Почтенный дедушка Дональд Кнут написал о них эпохальный труд сорок лет назад. Самые удачные алгоритмы всем известны и «прошиты» в языки программирования и стандартные библиотеки.

В Perl тоже есть встроенная функция сортировки. Называется `sort`. Работает в списковом контексте, принимает неупорядоченный список, возвращает упорядоченный.

```
1 my @sorted = sort @unsorted;
```

> На этом, кажется, можно было закончить. Но рассмотрим случаи посложнее.

Программисту не обязательно знать, какой

именно алгоритм сортировки у Perl под капотом (хотя способ узнать и даже изменить есть).

В свою очередь, универсальные алгоритмы сортировки не нуждаются ни в каком особом знании о предметах, которые они сортируют. Им достаточно двух функций: *сравнить* два элемента и *переставить* их местами. Как переставлять элементы, интерпретатор Perl отлично знает, а вот насчет сравнения может и вас послушать.

Функция сравнения

Если для `sort` не указана функция сравнения, будет использована операция `cmp` — лексикографическое сравнение строк. Четыре наиболее популярных функции:

```
{ $a <=> $b }
{ $b <=> $a }
{ $a cmp $b }
{ $b cmp $a }
```

заменяются на встроенный код, выполняющийся очень быстро.

> Иногда можно встретить замечание, что
>
> ```
> reverse sort { $a <=> $b
>     } @list
> ```
>
> выполняется быстрее, чем
>
> ```
> sort { $b <=> $a } @list
> ```
>
> В современных версиях Perl это не так. Не используйте `reverse` для изменения порядка сортировки.

Функция сравнения должна возвращать число, меньшее, большее или равное нулю, в зависимости от того, как переменные `$a` и `$b` связаны отношением порядка. Но даже встроенные функции не всегда соблюдают это требование: операция численного сравнения `<=>` возвращает `undef`, если в операндах есть *NaN*, «не-число». Поэтому избегайте списков, содержащих *NaN*, результат их сортировки будет неопределенным. Очистить список можно так:

```
1 @clean = grep { $_ == $_ } @nans;
```

Сортировка по нескольким полям

Одной операцией сравнения не обойтись если нужна комбинированная сортировка. Очень удобно объединять несколько сравнений с помощью `||` — логического

«или».

Упорядочим строки сначала по длине, затем (строки одной длины) лексикографически:

```
my @sorted = sort {
    length $a <=> length $b
    ||
    $a cmp $b
} @strings;
```

Для первой операции *ключом сортировки* является длина строки, для второй — сама строка.

Теперь упорядочим список сотрудников по фамилии, имени и размеру тапочек (тапочки сортируем от больших к меньшим):

```
my @sorted = sort {
    $a->{LastName}  cmp $b->{
        LastName}  ||
```

```
3       $a->{FirstName} cmp $b->{
            FirstName} ||
4       $b->{ShoeSize}  <=> $a->{
            ShoeSize}
5   } @employees;
```

Такая функция сравнения уже достаточно сложна, чтобы занимать заметное процессорное время, учитывая, что для списка из тысячи элементов она будет вызвана десяток тысяч раз. Запомните: всё, что происходит в функции сравнения, происходит `O(n log n)` раз.

Маневр орков (Orcish Maneuver, OM)

Во многих задачах ключ сортировки вычисляется сложной или недетерминированной

функцией, или даже внешней функцией (*XSUB*). Популярный пример — сортировка списка файлов по времени модификации. Очевидная реализация:

```
sort { -M $a <=> -M $b } @files;
```

имеет два недостатка: во-первых, функция `stat` будет вызвана столько раз, сколько потребуется сравнений, то есть `O(n log n)`. Во-вторых, между вызовами функции файл может быть изменен, получив новое значение в качестве ключа.

Закэшируем вычисленное значение времени модификации. Тогда для каждого файла функция `stat` будет вызвана единожды:

```
my %mod_times;  # cache
my @sorted = sort {
    ( $mod_times{$a} //= -M $a )
    <=>
    ( $mod_times{$b} //= -M $b )
```

```
6 } @files;
```

Оба недостатка ликвидированы.

> Название идиомы происходит от созвучия слов «Or» и «Cache» со словом «Orcish»; пока в Perl не появился оператор *defined-or*, приходилось использовать обычное *or*, с тем неудобством, что если значение ключа интерпретировалось как *false*, оно вычислялось повторно.

Преобразование Шварца (Schwartzian transform, ST)

Добиться вычисления ключа только один раз для каждого элемента можно иным способом:

1. Дополним элементы исходного списка вычисленным ключом: построим новый список с элементами-кортежами вида `[item, sortkey]`, которые содержат собственно элемент и ключ сортировки.
2. Упорядочим новый список «по второй колонке».
3. Извлечем из упорядоченного списка исходные элементы.

Perl позволяет лаконично записать такое преобразование с использованием анонимных списков:

```
my @sorted = map  { $_->[0] }
             sort { $a->[1] <=>
                    $b->[1] }
             map  { [$_, -M] }
                  @files;
```

Эта идиома программирования названа в честь Рэндала Л. Шварца, соавтора многих книг о Perl (в частности, легендарной «Llama book»). Рэндал Шварц продемонстрировал её в 1994 году, вскоре после выхода Perl 5.

> Справедливости ради заметим, что метод был известен и ранее, в других языках, как идиома «*decorate-sort-undecorate*», а на-

звание «*Schwartzian transform*» применяется в основном в Perl-практике.

Ключ будет вычислен ровно N раз для списка из N элементов. Поскольку нет расходов на поиск в хеш-таблице, ST, как правило, выполняется быстрее, чем OM. Но если сортируемый список содержит значительное число одинаковых элементов, то метод OM может оказаться эффективнее: попадание в кэш будет происходить чаще (на всякий случай напомню, что если множество значений элементов мало, то можно упорядочить список вообще за линейное время — смотрите *сортировку подсчетом*).

Преобразование Гаттмана-Рослера (Guttman-Rosler Transform, GRT)

Вызов пользовательской функции сравнения достаточно дорог, а обычное численное или лексикографическое сравнение выполняется быстро. Метод оптимизации, предложенный Perl-хакерами Uri Guttman и Larry Rosler, заключается в преобразовании элементов сортируемого списка таким образом, чтобы их сравнение выполнялось встроенными функциями.

Допустим, исходный список представляет собой набор кортежей-троек целых чисел в диапазоне от 0 до 99 включительно, и упорядочить его нужно сначала по первым числам, затем по вторым и т.д. Сортировка с пользовательской функцией сравнения:

```
my @sorted = sort {
    $a->[0] <=> $b->[0] ||
    $a->[1] <=> $b->[1] ||
    $a->[2] <=> $b->[2]
} @triplets;
```

Воспользуемся тем фактом, что числа укладываются в два десятичных разряда и составим из троек целые числа в диапазоне `[ 0, 999999 ]` (они даже помещаются в машинное целое). Затем отсортируем полученные элементы обычным численным сравнением `$a <=> $b` (несмотря на то, что код тоже выглядит как пользовательская функция, Perl будет использовать встроенный метод). Наконец, разобьем числа на тройки по двум цифрам, вернув исходное представление:

```
my @sorted = map  {
                        $x = int($_
                             /
```

```
                    100**2);
3                  $y = int($_
                        / 100)
                       - $x *
                       100;
4                  $z = $_ %
                       100;
5                  [ $x, $y,
                       $z ];
6              }
7          sort { $a <=> $b }
8          map  { $_->[0] *
               100**2 + $_->[1]
               * 100 + $_->[2] }
9                @triplets;
```

На массивах из ста тысяч элементов можно получить выигрыш в скорости выполнения более чем в два раза.

Второй вариант — преобразовать тройки чисел в строки длиной три байта и

упорядочить их лексикографически:

```
my @sorted = map  { [ unpack "C3"
    , $_ ] }
               sort
               map  { pack "C3",
                  @$_ }
                    @triplets;
```

Для упаковки сортируемых объектов в числа или строки есть множество способов:

- запись в битовые поля;
- арифметическое кодирование;
- формирование строк фиксированной длины (*padding*);
- формирование строк с разделителями, например, нулевыми байтами.

Здесь будут полезны функции `sprintf`, `pack`, `join`, регулярные выражения. При

сериализации чисел в строку не забывайте выравнивать их по правому краю, дополняя нулями или пробелами, и использовать фиксированное число знаков после точки. Побитовое отрицание над строками удобно, чтобы изменить порядок сортировки строк на обратный. Не забудьте, что на лексикографическое сравнение действует включенная локаль (*locale collation*).

Сериализация и десериализация данных — также довольно дорогие действия. Насколько эффективным будет GRT, зависит от способа сериализации, от сложности функции сравнения и от отношения количества сравнений к количеству преобразований данных. Можно ускорить сортировку на порядок, можно и наоборот, потерять в скорости. Обязательно протестируйте несколько вариантов упаковки объектов на данных, максимально приближенных к

реальным.

Выбор алгоритма сортировки

Начиная с версии 5.8, Perl использует два алгоритма сортировки: mergesort — сортировку слиянием, и quicksort — быструю сортировку.

В большинстве случаев выгоднее алгоритм mergesort: он имеет гарантированную сложность `O(n log n)`, в то время как quicksort деградирует до квадратичной сложности в худших случаях, и он *устойчив*, то есть сохраняет исходный порядок элементов, равных по ключу.

Однако иногда может быть полезнее quicksort, который потребляет меньше

памяти и быстрее сортирует списки, содержащие небольшое количество уникальных элементов (то есть списки, имеющие высокую, плохую *selectivity* в терминах реляционных баз данных).

Вы можете указать предпочитаемый алгоритм директивой (прагмой) `sort`, например, потребовать, чтобы обязательно использовался устойчивый алгоритм:

```
1 use sort qw(stable);
```

Директива действует в лексической области видимости.

Модули CPAN

Sort::Maker

Очень мощный модуль с единственной экспортируемой функцией `make_sorter`, которая построит для вас функцию сравнения с любым упомянутым преобразованием.

Sort::MultipleFields

Не содержит никаких оптимизаций, зато позволяет записать сортировку по нескольким полям компактнее и понятнее.

Sort::Key

Самая лучшая оптимизация — написать функцию сравнения на C. Модуль предоставляет целую пачку XSUB для сравнения ключей и ссылается на не меньшую пачку модулей Sort::Key::*.

Sort::Fields

Весьма старый, но вполне работающий модуль для сортировки текстовых таблиц (наподобие CSV).

Sort::XS

Еще один XS-модуль, предоставляющий уже не функции сравнения, а собственно алгоритмы сортировки.

Sort::External

Реализация внешней сортировки, то есть сортировки списков, не помещающихся в память. Подмодуль Sort::External::Cookbook содержит прекрасное описание метода GRT.

Заключение

- Функция сравнения вызывается `O(n log n)` раз.
- Кэшируйте ключи сортировки.
- Упакуйте элементы списка в числа или строки и сравнивайте их одной операцией.
- Как водится, на CPAN есть много вкусного.

Исходные тексты с тестами производительности

1-basic.pl

Тестирование тривиальных операций сравнения. Для сортировки в обратном порядке

функция `reverse` не дает преимуществ. Если же в результате операции изменить знак, функция будет воспринята уже как пользовательская, а не встроенная — с соответствующим падением производительности.

2-om-st.pl

Упорядочим набор точек на плоскости по расстоянию до центра (или, если угодно, упорядочим векторы по длине). Расчет длины вектора даже в 2-мерном пространстве достаточно трудоемок, чтобы кэширование принесло значительную пользу. ST справляется быстрее, чем OM.

3-om-st-cardinality.pl

Изменим предыдущую задачу так, чтобы набор содержал большое число одинако-

вых точек (в примере генерируются точки с целыми координатами в диапазоне [0, 10], то есть всего сто уникальных значений). На этот раз OM быстрее, чем ST.

4-grt.pl

Сортировка троек целых чисел методом GRT. Три различных способа сериализации; упаковка в битовые поля наиболее эффективна.

■ *Олег Алистратов*

3 Создание RSS из списка файлов

В позапрошлом номере было упомянуто о преобразовании XML в Perl-структуры. В этой статье будет рассказано об обратном пути на примере генерирования RSS из списка файлов в каталоге.

В качестве задачи рассмотрим следующее. В каталоге собираются медиафайлы, например, подкасты, которые будет раздавать веб-сервер. Напишем скрипт, который бы создавал RSS на основе этих файлов.

Создание RSS можно было бы описать в несколько предложений, но, как мы увидим позже, есть несколько подводных камней.

Реверс-инжиниринг

Постараемся исходить из того, что нужно на выходе, а потом зададимся тем, как это реализовать. Итак, RSS должен выглядеть следующим образом:

```
<?xml version="1.0" encoding="UTF-8"?>
<rss version="2.0" xmlns:blogChannel="http://backend.userland.com/blogChannelModule">
<channel>
    <title>Very Cool Podcast</title>
    <link>http://www.example.com</link>
    <description>Very Cool Podcast Description</description>
    <image>
```

```
        <title>Very Cool Podcast
           </title>
        <url>http://www.example.
           com/images/logo.png</
           url>
        <link>http://www.example.
           com</link>
    </image>
    <item>
        <title>Very Cool Podcast
           - 2013-02-18T19-00-01.
           mp3</title>
        <link>http://www.example.
           com/very_cool_podacst
           /2013-02-18T19-00-01.
           mp3</link>
        <description>2013-02-18
           T19-00-01.mp3</
           description>
        <enclosure url="http://
           www.example.com/
           very_cool_podacst
           /2013-02-18T19-00-01.
```

```
                mp3" type="audio/mpeg"
                 />
    </item>
</channel>
</rss>
```

Как мы видим, для минимального RSS нужно довольно много информации. Проще всего получить имя файла и дату его создания/изменения. Остальную информацию, общую для всех файлов, будем хранить в отдельном `ini`-файле в том же каталоге. Для чтения будем использовать `Config::Simple`.

Вот как будет выглядеть `ini`-файл для подкаста:

```
title="Very Cool Podcast"
link="http://www.example.com"
image="http://www.example.com/
    images/logo.png"
```

```
4 description="Very Cool Podcast
    Description"
5 extensions="ogg", "mp3"
6 httpbase="http://www.example.com/
    very_cool_podacst"
```

Осталось ещё одна мелочь. Так как мы хотим, чтобы подкаст смогли скачивать RSS-агрегаторы и мобильные устройства, нужен тег `enclosure` с правильным `mime` -типом. Чтобы определить правильный `mime`-тип воспользуемся модулем `MIME::Types`.

Сценарий

Скрипт можно разбить на несколько частей.

- Разбор командной строки с помощью `Getopt::Long`.
- Загрузка `ini`-файла с помощью `Config::Simple`.
- Создание `XML::RSS` объекта.
- Проход файлов в каталоге.
- Добавление элементов в `XML::RSS`-объект.
- Запись RSS-файла.

Разбор командной строки с помощью `Getopt::Long`

Это самая тривиальная часть. Минимальный джентльменский набор:

```
#!/usr/bin/perl

use strict;
use warnings;
```

readdir ведёт себя по-другому в 5.11.2:

```
use 5.012; # so readdir assigns
    to $_ in a lone while test
```

Загружаем все нужные модули:

```
use Getopt::Long;
use Config::Simple;
use XML::RSS;
use MIME::Types;
```

Декларируем глобальные переменные:

```
my $projPath;
my $debug = 0;
my $test  = 0;
my $config;
my $sampleConfig = <<EOF;
title="Podcast Title"
link="http://blah.com"
description="Podcast Description"
extensions = "flv", "mp4", "mp3"
```

```
10 httpbase="http://server/test-
       podcast"
11 EOF
```

`$projPath` будет указывать на каталог с медиафайлами, а `$config` на объект конфигурации. Также декларируем пример конфигурационного файла для удобства. Кстати, изображение не обязательно. К этому вернёмся ещё раз при создании `XML::RSS`-объекта.

Начнем с функции вывода помощи. `$0` будет знаменен на полный путь и имя вызванного скрипта.

```
1 sub usage {
2     die <<EOF;
3 Usage:
4     $0 -path /path/to/my/
          videos_or_audio_podcasts
          [-verbose] [-debug] [-test
```

```
        ]

—debug    enable debugging output

EOF
}
```

И сам разбор командной строки:

```
GetOptions(
    "path=s" => \$projPath,
    "debug"  => \$debug,
    "test"   => \$test
);
usage() if !defined $projPath;
```

Загрузка ini-файла с помощью Config::Simple

Файл конфигурации должен находиться в каталоге проекта.

```
my $configFile = $projPath . '/
    config.ini';
```

Выдаём пример конфигурации, если файл конфигурации не найден, или не читаем.

```
print "Checking if $configFile is
     readable.$/" if $debug;
die "Can't read config file
    $configFile!"
   . "$/ Here is a sample config:$
     /$sampleConfig"
  if !-r $configFile;
```

Загружаем файл конфигурации и проверяем, все ли переменные декларированы. Ес-

ли нет, выдаём также пример конфигурации. В конце сохраняем ссылку на хеш объекта конфигурации в глобальную переменную для последующего использования.

```
my %tempConfig;
print "Loading config file $configFile.$/" if $debug;
Config::Simple->import_from( $configFile, \%tempConfig);

print "Checking if needed variables exist in the config file.$/" if $debug;
if (   defined $tempConfig{'default.title'}
    && defined $tempConfig{'default.link'}
    && defined $tempConfig{'default.description'}
    && defined $tempConfig{'default.extensions'}
```

```
10      && defined $tempConfig{'
           default.httpbase'})
11  {
12      print "All config variables
           exist.$/" if $debug;
13  }
14  else {
15      die <<EOF;
16  Can't find all of the needed
       variables in the config file.
17  The config file should look like
       this:
18  --cut--
19  $sampleConfig
20  --cut--
21
22  EOF
23  }
24  $config = \%tempConfig;
```

На этом этапе файл конфигурации загружен, и ссылка на его объект находится в

глобальной переменной `$config`.

Создание `XML::RSS`-объекта

Предупреждаем, если скрипт работает в тестовом режиме.

```
print "Running in test mode, no
    xml file will be written.$/"
  if $debug && $test;
```

Генерируем `XML::RSS`-объект. Здесь важны версия 2.0 и ключ `encode_output => 0`. Версия нужна для тега `enclosure`, а ключ `encode_output` важен для русского. Без него все русские буквы (и не только) будут кодироваться в HTML-коды.

```
print "Generating RSS.$/" if
    $debug;
```

```
2 my $rss = XML::RSS->new(version
    => '2.0', encode_output => 0);
```

Заполняем поля канала из объекта конфигурации.

```
1 $rss->channel(
2     title       => $config->{'
        default.title'},
3     link        => $config->{'
        default.link'},
4     description => $config->{'
        default.description'},
5 );
```

Если найдено изображение, то добавляем и его.

```
1 if (defined $config->{'default.
    image'}) {
2     print "Found image variable
        in the config files, "
```

```
3         . "adding image tag to rss.
              $/"
4         if $debug;
5     $rss->image(
6         title => $config->{'
              default.title'},
7         url   => $config->{'
              default.image'},
8         link  => $config->{'
              default.link'},
9     );
10 }
```

На этом этапе мы создали базовый `XML::RSS`-объект.

Проход файлов в каталоге

Необходимо получить список всех файлов в каталоге проекта `$projPath` и отсортиро-

вать их по дате создания. Для получения списка файлов используем readdir из стандартного комплекта Perl. Но перед этим стоит узнать, какие типы файлов нас интересуют. Для этого и был объявлен массив `extensions` в файле конфигурации.

Сообщаем, что будем искать.

```
print "Searching for "
  . join(', ', @{$config->{'
    default.extensions'}})
  . " in $projPath.$/"
  if $debug;
```

Генерируем регулярное выражение для скорости. Регулярное выражение может выглядеть, например, так: `/(ogg|mp3)$/i`

```
my $extensionsMatch = join('|', @
    {$config->{'default.extensions
    '}});
$extensionsMatch = qr/(
```

```
    $extensionsMatch)$/i;
```

Теперь приступим к самим файлам.

Создаём хеш для последующего добавления файлов и сортировки по дате.

```
my %items;

opendir(my $dh, $projPath) || die
    ;
while (readdir $dh) {
    my $fullPath = $projPath . '/
        ' . $_;
```

Игнорируем каталоги и пустые файлы.

```
    next if -d $fullPath;
    next if -z $fullPath;
```

Если имя файла (оно будет в $_) совпадает с регулярным выражением, которое бы-

ло создано до этого, то добавляем файл в хеш, используя как ключ дату создания и inode, чтобы избежать коллизий на тот случай, если несколько файлов были созданы одновременно.

```
1       if (/$extensionsMatch/i) {
2           my $timeStamp = (stat(
               $fullPath))[9];
3           my $inode     = (stat(
               $fullPath))[1];
4           $items{$timeStamp .
               $inode} = $_;
5       }
6   }
7   closedir $dh;
```

Добавление элементов в `XML::RSS`-объект

На этом этапе есть хеш с ключами, которые нужно отсортировать по убыванию. Потом добавить элементы хеша (имена файлов) в `XML::RSS`-объект предварительно определив `mime`-тип.

Создаём объект для определения `mime`-типа.

```
1 my $mimetypes = MIME::Types->new;
```

Проходим по всем ключам, отсортированным по убыванию.

```
1 foreach my $key ( sort {$b <=> $a
    } keys(%items) ){
2     my $filename = $items{$key};
```

Берём только расширение файла и по нему определяем mime-тип.

```
1       my ($fileExtension) = (
           $filename =~ /\.([\w\d]+)$
           /);
2       my $fileMimeType = $mimetypes
           –>mimeTypeOf($filename)–>
           type();
```

Добавляем файл в XML::RSS-объект как элемент.

```
1       print "Adding $filename.$/"
           if $debug;
2       $rss–>add_item(
3           title     => $config–>{'
              default.title'} . ' –
              ' . $filename,
4           link      => $config–>{'
              default.httpbase'} . '
              /' . $filename,
5           enclosure => {
```

```
            url  => $config->{'
                default.httpbase'}
                 . '/' . $filename
                ,
            type => $fileMimeType
        },
        description => $filename,
    );
}
```

Запись RSS-файла

Осталось теперь только записать RSS-файл.

Если указан флаг `--test`, то только выдаём результат на `STDOUT` и завершаем выполнение.

```
if ($test) {
    print $rss->as_string;
```

```
    exit 0;
}
```

Если работаем не в тестовом режиме, то записываем файл в каталог проекта `$projPath` стандартными средствами Perl. Мы не используем функцию `XML::RSS save` (`$rss->save('filename.xml')`), потому что она записывает результат не в UTF-8, что делает его нечитаемым.

```
my $outFile = $projPath . '/rss.xml';
print "Saving xml to $outFile$/" if $debug;
open(my $fh, ">", $outFile);
print $fh $rss->as_string;
close($fh);
```

Заключение

Красота Perl — в простоте использования различных модулей: соединяя их своим кодом, получаем комплексное решение. Надеюсь, мне удалось это продемонстрировать в этой короткой статье. Полный текст программы доступен в репозитории.

■ *Олег Фиксель*

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

Продолжение цикла статей о PSGI/Plack. Рассмотрен более подробно preforking PSGI-сервер Starman.

Starman?

Автор данного сервера (Tatsuhiko Miyagawa) говорит про него следующее:

> Название Starman взято из из песни Star H. A. Otoko японской рок-группы Unicorn (Да, Unicorn). У David Bowie тоже есть одноименная песня, Starman — имя персонажа

культовой японской игры Earthbound, название музыкальной темы из Super Mario Brothers.

Я устал от именования Perl-модулей наподобие HTTP::Server::PSGI::H а в результате люди называют это HSPHIWWWM в IRC. Это плохо произносится и создает проблемы новичкам. Да, может быть я упорот. Время покажет.

С названием разобрались. Теперь будем разбираться с самим сервером.

Preforking?

Preforking-модель у Starman подобна наиболее высокопроизводительным Unix-

серверам. Он использует модель предварительно запущенных процессов. Также он автоматически рестартует пул воркеров и убирает свои зомби-процессы.

Plack-приложение

В этот раз Plack-приложение будет совсем элементарным:

```
use strict;
use warnings;

use Plack;
use Plack::Builder;
use Plack::Request;

sub body {
    return 'body';
}

```

```
12 sub body2 {
13     return shift;
14 }
15
16 my $app = sub {
17     my $env = shift;
18     my $req = Plack::Request->new
           ($env);
19     my $res = $req->new_response
           (200);
20     $res->body(body());
21     return $res->finalize();
22 };
23
24 my $main_app = builder {
25     mount "/" => builder { $app
           };
26 };
```

При разработке под Starman необходимо понимать один очень важный момент его работы. Рассмотрим, например, соедине-

ние с базой данных. Очень часто для того, чтобы сэкономить время и строки кода, инициализацию соединения выносят в самое начало скрипта. Это касается CGI и иногда FastCGI. В случае с PSGI так делать нельзя. И вот почему. При старте сервера этот код будет выполнен ровно **один** раз для каждого воркера. А опасность ситуации заключается в том, что поначалу, пока соединение не вылетит либо по таймауту, либо по каким-то еще причинам, приложение будет работать в штатном режиме. В случае с асинхронными серверами в начале кода приложения можно инициализировать пул соединений (соединение != пул соединений).

Для того, чтобы это подтвердить или опровергнуть, внесем изменения в код приложения. Добавим в начало кода, после импортов, следующую строчку:

```
warn 'AFTER IMPORT';
```

Теперь приложение должно иметь вид:

```
use strict;
use warnings;

use Plack;
use Plack::Builder;
use Plack::Request;

warn 'AFTER IMPORT';

sub body {
    return 'body';
}

sub body2 {
    return shift;
}

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

```
20      my $req = Plack::Request->new
            ($env);
21      my $res = $req->new_response
            (200);
22      $res->body(body());
23      return $res->finalize();
24  };
25
26  my $main_app = builder {
27      mount "/" => builder { $app
            };
28  };
```

Для чистоты эксперимента будем проводить запуск starman с одним воркером следующей командой:

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

Где app.psgi — приложение.

Незамедлительно после выполнения запуска видим следующую картину в STDERR:

```
noxx@noxx-inferno ~/perl/psgi $ starman --port 8080 app.psgi --workers 1
2013/06/02-15:05:31 Starman::Server (type Net::Server::PreFork) starting! pid(4204)
Resolved [*]:8080 to [::]:8080, IPv6
Not including resolved host [0.0.0.0] IPv4 because it will be handled by [::] IPv6
Binding to TCP port 8080 on host :: with IPv6
Setting gid to "1000 1000 4 24 27 30 46 107 125 1000 1001"
AFTER IMPORT at /home/noxx/perl/psgi/app.psgi line 7.
```

Если отправить запрос на localhost:8080/, можно убедиться, что ничего нового в

STDERR не появилось, а сервер нормально отвечает.

Для того, чтобы убедиться, что worker действительно один, выполним следующую команду:

```
ps uax | grep starman
```

Результат:

```
noxx       4204  0.6  0.1  57836
    11264 pts/3    S+   15:05
    0:00 starman master --port
    8080 app.psgi --workers 1
noxx       4205  0.2  0.1  64708
    13164 pts/3    S+   15:05
    0:00 starman worker --port
    8080 app.psgi --workers 1
noxx       4213  0.0  0.0  13580
      940 pts/4    S+   15:05
    0:00 grep --colour=auto
    starman
```

Процесса два. Но на самом деле worker из них только один. Проведем еще один эксперимент. Запустим starman с тремя воркерами.

```
starman --port 8080 --workers 3 app.psgi
```

Результат:

```
2013/06/02-15:11:08 Starman::Server (type Net::Server::PreFork) starting! pid(4219)
Resolved [*]:8080 to [::]:8080, IPv6
Not including resolved host [0.0.0.0] IPv4 because it will be handled by [::] IPv6
Binding to TCP port 8080 on host :: with IPv6
Setting gid to "1000 1000 4 24 27 30 46 107 125 1000 1001"
```

```
6 AFTER IMPORT at /home/noxx/perl/
    psgi/app.psgi line 7.
7 AFTER IMPORT at /home/noxx/perl/
    psgi/app.psgi line 7.
8 AFTER IMPORT at /home/noxx/perl/
    psgi/app.psgi line 7.
```

Все верно. Теперь посмотрим на список процессов. У меня он выглядит так:

```
1 noxx      4219  0.1  0.1  57836
    11264 pts/3    S+   15:11
    0:00 starman master --port
    8080 app.psgi --workers 3
2 noxx      4220  0.0  0.1  64460
    12756 pts/3    S+   15:11
    0:00 starman worker --port
    8080 app.psgi --workers 3
3 noxx      4221  0.0  0.1  64460
    12920 pts/3    S+   15:11
    0:00 starman worker --port
    8080 app.psgi --workers 3
```

```
4 noxx       4222  0.0  0.1  64460
    12756 pts/3    S+   15:11
    0:00 starman worker --port
    8080 app.psgi --workers 3
5 noxx       4224  0.0  0.0  13580
      936 pts/4    S+   15:12
    0:00 grep --colour=auto
    starman
```

Один мастер, три воркера.

С порядком выполнения разобрались. Теперь добавим еще один warning.

```
1 warn 'IN BUILDER'
```

Приложение выглядит следующим образом:

```
1 use strict;
2 use warnings;
3
4 use Plack;
```

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

warn 'AFTER IMPORT';

sub body {
    return 'body';
}
sub body2 {
    return shift;
}
my $app = sub {
    my $env = shift;
    my $req = Plack::Request->new
        ($env);
    my $res = $req->new_response
        (200);
    $res->body(body());
    return $res->finalize();
};
my $main_app = builder {
    warn 'IN BUILDER';
```

```
25      mount "/" => builder { $app
            };
26  };
```

Для одного worker-процесса вывод выглядит так (команда запуска: `starman --port 8080 --workers 1 app.psgi`):

```
1  2013/06/02-17:33:27 Starman::
       Server (type Net::Server::
       PreFork) starting! pid(4430)
2  Resolved [*]:8080 to [::]:8080,
       IPv6
3  Not including resolved host
       [0.0.0.0] IPv4 because it will
        be handled by [::] IPv6
4  Binding to TCP port 8080 on host
       :: with IPv6
5  Setting gid to "1000 1000 4 24 27
        30 46 107 125 1000 1001"
6  AFTER IMPORT at /home/noxx/perl/
       psgi/app.psgi line 7.
```

```
7 IN BUILDER at /home/noxx/perl/
    psgi/app.psgi line 23.
```

Если же мы запустим приложение с тремя воркерами, то увидим следующую картину в STDERR:

```
1 AFTER IMPORT at /home/noxx/perl/
    psgi/app.psgi line 7.
2 IN BUILDER at /home/noxx/perl/
    psgi/app.psgi line 23.
3 AFTER IMPORT at /home/noxx/perl/
    psgi/app.psgi line 7.
4 IN BUILDER at /home/noxx/perl/
    psgi/app.psgi line 23.
5 AFTER IMPORT at /home/noxx/perl/
    psgi/app.psgi line 7.
6 IN BUILDER at /home/noxx/perl/
    psgi/app.psgi line 23.
```

Сделав запрос на localhost:8080/, легко можно убедиться в том, что ничего нового

в STDERR не появилось.

Можно сделать следующие выводы:

- Данное действие **будет** выполняться при старте приложение. Это справедливо как для начала скрипта, так и для builder-секции, если она есть.
- Данное действие **не будет** выполняться при запросах на сервер.
- Рабочие процессы Starman стартуют последовательно.

Это дает возможность конструировать тяжелые объекты как при старте скрипта, так и в builder-части.

А вот теперь добавим в код еще один warning следующего вида:

```
1 warn 'REQUEST';
```

И приведем приложение к следующему виду:

```
use strict;
use warnings;

use Plack;
use Plack::Builder;
use Plack::Request;

warn 'AFTER IMPORT';

sub body {
    return 'body';
}
sub body2 {
    return shift;
}
my $app = sub {
    warn 'REQUEST';
    my $env = shift;
    my $req = Plack::Request->new
        ($env);
```

```
20      my $res = $req->new_response
            (200);
21      $res->body(body());
22      return $res->finalize();
23  };
24  my $main_app = builder {
25      warn 'IN BUILDER';
26      mount "/" => builder { $app
            };
27  };
```

Теперь запустим приложение с одним рабочим процессом (`starman --port 8080 --workers 1 app.psgi`). Пока что ничего не изменилось:

```
1  AFTER IMPORT at /home/noxx/perl/
       psgi/app.psgi line 7.
2  IN BUILDER at /home/noxx/perl/
       psgi/app.psgi line 24.
```

Но стоит сделать запрос, как в STDERR появится новая запись.

```
REQUEST at /home/noxx/perl/psgi/
    app.psgi line 16.
```

Подведем итог. При каждом запросе к starman будет выполняться только код непосредственно приложения (стоит вспомнить `return sub ...`), но при старте этот код выполняться не будет.

А теперь, допустим, один процесс упал. Добавим следующую строчку в `return sub ...`:

```
die("DIED");
```

В результате должны получить приложение следующего вида:

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

```
3
4  use Plack;
5  use Plack::Builder;
6  use Plack::Request;
7
8  warn 'AFTER IMPORT';
9
10 sub body {
11     return 'body';
12 }
13 sub body2 {
14     return shift;
15 }
16 my $app = sub {
17     warn 'REQUEST';
18     my $env = shift;
19     my $req = Plack::Request->new
           ($env);
20     my $res = $req->new_response
           (200);
21     $res->body(body());
22     die("DIED");
23     return $res->finalize();
```

```
24 };
25 my $main_app = builder {
26     warn 'IN BUILDER';
27     mount "/" => builder { $app
          };
28 };
```

Запускаем приложение с одним рабочим процессом, делаем запрос. Приложение, естественно, падает. Но результат любопытен, хотя и закономерен. Приложение не упало, в STDERR появилось только два уведомления:

```
1 REQUEST at /home/noxx/perl/psgi/
    app.psgi line 16.
2 DIED at /home/noxx/perl/psgi/app.
    psgi line 21.
```

А теперь заменим `die('DIED');` на `exit 1;`. Запустим Starman, сделаем запрос на localhost:8080/. Вот теперь рабочий процесс

упал. Это видно по STDERR, который будет выглядеть теперь так:

```
1 REQUEST at /home/noxx/perl/psgi/
    app.psgi line 16, <$read> line
     7.
2 AFTER IMPORT at /home/noxx/perl/
    psgi/app.psgi line 7, <$read>
    line 8.
3 IN BUILDER at /home/noxx/perl/
    psgi/app.psgi line 26, <$read>
     line 8.
```

После каждого запроса рабочий процесс будет падать, но master-процесс будет его поднимать.

Оставим `Starman` ненадолго. Попробуем запустить данное приложение, например, под `Twiggy`. Если данный сервер не установлен, то самое время его установить. Пакет называется `Twiggy`.

После установки Twiggy запустим наше приложение следующей командой:

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

И сделаем запрос. Все как у Starman, за исключением одной особенности. Сервер свалился.

```
1 noxx@noxx-inferno ~/perl/psgi $
    twiggy --port 8080 app.psgi
2 AFTER IMPORT at /home/noxx/perl/
    psgi/app.psgi line 7.
3 IN BUILDER at /home/noxx/perl/
    psgi/app.psgi line 26.
4 REQUEST at /home/noxx/perl/psgi/
    app.psgi line 16, <> line 5.
5 noxx@noxx-inferno ~/perl/psgi $
```

Разумеется, это потому, что у Twiggy отсутствует мастер-процесс и поднять упавшего рабочего некому. А теперь отсюда следует очень важный момент, который

надо учитывать. **Перед рестартом сервера необходимо убедиться в том, что его код корректен и не содержит синтаксических ошибок**. Если попробовать запустить приложение, которое содержит ошибку, при помощи Starman, произойдут несколько событий в следующем порядке:

- Starman запустит master-процесс, проверит, может ли он запустить рабочие процессы.
- Starman запустит рабочие процессы и передаст на исполнение код приложения.
- Рабочие процессы начнуть падать, а мастер начнет их поднимать.

Нагрузка увеличивается невероятно и за очень короткий промежуток времени.

Ошибки во время исполнения не настолько критичны. Уберем падения из приложения, приведя его практически к начальному виду:

```
use strict;
use warnings;

use Plack;
use Plack::Builder;
use Plack::Request;

warn 'AFTER IMPORT';

sub body {
    return 'body';
}
sub body2 {
    return shift;
}
my $app = sub {
    warn 'REQUEST';
    my $env = shift;
```

```
19      my $req = Plack::Request->new
            ($env);
20      my $res = $req->new_response
            (200);
21      $res->body(body());
22      return $res->finalize();
23  };
24  my $main_app = builder {
25      warn 'IN BUILDER';
26      mount "/" => builder { $app
            };
27  };
```

И попробуем сделать следующее именно в таком порядке:

- Приводим приложение к начальному виду.
- Запустим его при помощи `Starman`.
- Сделаем запрос.

- Изменим код приложения и сохраним его.
- **Не рестартуя приложение** сделаем запрос на него еще раз.

Результат:

```
curl localhost:8080/
body
```

Сохраняем приложение, меняем функцию body. Пусть теперь, например, она возвращает nobody. Делаем запрос — результат, если мы не рестартовали сервер, следующий:

```
curl localhost:8080/
body
```

Но стоит сделать рестарт, как все меняется:

```
curl localhost:8080/
nobody
```

Еще один важный вывод. Для того, чтобы приложение обновилось, изменения файлов недостаточно. Необходимо перезапустить сервер. Или же отправить специальный сигнал мастер-процессу.

Starman и сигналы

Представим, что у нас большое PSGI-приложение, останавливать которое нельзя, т.к. у нас довольно тяжелые библиотеки, которые загружаются в память, скажем, десять секунд.

Повторим предыдущую цепочку действий, но с одним изменением. Добавим отправку сигналов.

Сигнал, который указывает `Starman`, что

надо бы перечитать — SIGHUP.

Команда на отправку данного сигнала выглядит так:

```
kill -s SIGHUP [pid]
```

Получить значение pid можно следующей командой:

```
ps uax | grep starman | grep master
```

Пример вывода команды:

```
noxx       6214  0.8  0.1  54852 10288 pts/3    S+   19:17   0:00 starman master --port 8080 --workers 1 app.psgi
```

pid = 6214.

Проверяем запрос-ответ. Заменяем nobody обратно на body и запускаем приложение.

Результат:

```
curl localhost:8080
body
kill -s SIGHUP 6214
curl localhost:8080
nobody
```

А тем временем в STDERR `Starman` мы можем видеть следующее:

```
AFTER IMPORT at /home/noxx/perl/psgi/app.psgi line 7.
IN BUILDER at /home/noxx/perl/psgi/app.psgi line 24.
REQUEST at /home/noxx/perl/psgi/app.psgi line 16.
Sending children hup signal
AFTER IMPORT at /home/noxx/perl/psgi/app.psgi line 7, <$read>
```

```
    line 2.
IN BUILDER at /home/noxx/perl/
    psgi/app.psgi line 24, <$read>
     line 2.
REQUEST at /home/noxx/perl/psgi/
    app.psgi line 16, <$read> line
     2.
```

Таким образом, есть два способа обновления PSGI-приложения. Какой выбирать — зависит от задачи.

Допустим, понадобился еще один рабочий процесс. Его можно добавить двумя способами. Рестартовать сервер с необходимым параметром (`--workers`) или же отправить сигнал. Сигнал на добавление одного рабочего процесса — `TTIN`, на удаление — `TTOU`. Если же мы хотим полностью безопасно остановить сервер, мы можем воспользоваться сигналом `QUIT`.

Итак. Запустим наше приложение с одним рабочим процессом:

```
starman --port 8080 --workers 1
```

Затем добавим два процесса, выполнив следующую команду дважды:

```
kill -s TTIN 6214
```

Список процессов Starman:

```
noxx      6214  0.0  0.1  54852 10304 pts/3    S+   19:17   0:00 starman master --port 8080 --workers 1 app.psgi
noxx      6221  0.0  0.1  64724 13188 pts/3    S+   19:19   0:00 starman worker --port 8080 --workers 1 app.psgi
noxx      6233  0.0  0.1  64476 12872 pts/3    S+   19:26   0:00 starman worker --port 8080 --workers 1 app.psgi
```

```
4 noxx       6239  2.0  0.1  64480
     12872 pts/3    S+   19:29
     0:00 starman worker --port
     8080 --workers 1 app.psgi
```

В STDERR уже привычное:

```
1 BAFTER IMPORT at /home/noxx/perl/
     psgi/app.psgi line 7, <$read>
     line 4.
2 IN BUILDER at /home/noxx/perl/
     psgi/app.psgi line 24, <$read>
      line 4.
3 AFTER IMPORT at /home/noxx/perl/
     psgi/app.psgi line 7, <$read>
     line 4.
4 IN BUILDER at /home/noxx/perl/
     psgi/app.psgi line 24, <$read>
      line 4.
```

Затем уберем один процесс:

```
1 kill -s TTOU 6214
```

Можем видеть, что команда возымела эффект, посмотрев на список процессов:

```
noxx        6214  0.0  0.1  54852
    10304 pts/3    S+   19:17
    0:00 starman master --port
    8080 --workers 1 app.psgi
noxx        6221  0.0  0.1  64724
    13188 pts/3    S+   19:19
    0:00 starman worker --port
    8080 --workers 1 app.psgi
noxx        6233  0.0  0.1  64476
    12872 pts/3    S+   19:26
    0:00 starman worker --port
    8080 --workers 1 app.psgi
noxx        6238  0.0  0.0  13584
      936 pts/4    S+   19:29
    0:00 grep --colour=auto
    starman
```

Но в STDERR это не отобразится.

А теперь завершим работу нашего прило-

жения, отправив ему сигнал QUIT.

```
kill -s QUIT 6214
```

Сервер пишет в STDERR:

```
2013/06/02-19:32:15 Received QUIT
    . Running a graceful shutdown
Sending children hup signal
2013/06/02-19:32:15 Worker
    processes cleaned up
2013/06/02-19:32:15 Server
    closing!
```

И завершает работу.

Это все, что необходимо знать о Starman для того, чтобы начать с ним работать.

Осталась еще одна важная деталь. При запуске Starman можно указать через ключ -M необходимый модуль для загрузки **через** master-процесс. Но тогда начинает

работать следующее ограничение. Модули, загруженные через `-M` (`-MDBI -MDBIx::Class`), при `SIGHUP` перечитываться **не будут**.

Еще одна полезная опция сервера — `-I`. Она позволяет указать путь Perl-модулям перед стартом master-процесса. Starman умеет также работать с Unix-сокетами, но эта возможность будет рассмотрена подробнее в следующих статьях, начиная со статьи по разворачиванию и администрированию `Plack`.

Ну и напоследок — флаг `-E`, который устанавливает переменную окружения (`PLACK_ENV`) в переданное состояние.

Следующая статья будет посвящена асинхронному PSGI-серверу — `Twiggy`.

■ *Дмитрий Шаматрин*

5 AnyEvent и fork

Довольно часто возникает задача выполнения некоторых действий в отдельном процессе, например, для исполнения блокирующихся операций или запуска внешних программ. fork отлично выполняет свою задачу, но если ваше приложение построено на базе AnyEvent, необходимо знать о некоторых нюансах, прежде чем начинать создавать новые процессы.

Fork

Классическим способом создания новых процессов в UNIX-подобных системах является выполнение системного вызова `fork`, который создаёт копию выполняемого процесса и начинает выполнять

оба процесса, начиная с последующей инструкции. Интерпретатор Perl просто выполняет системный вызов, если он доступен, а в случае его отсутствия — выполняет эмуляцию, которая в общем случае абсолютно прозрачна для программиста (с небольшими оговорками, смотрите perlfork). В современных операционных системах `fork` оптимизирован и не выполняет копирования всего адресного пространства процесса, выполняя копирование лишь при изменении данных в процессе-потомке (механизм *COW* — копирование-при-записи). Так или иначе, оба процесса имеют в своём распоряжении все данные, располагавшиеся в родительском процессе, что в том числе касается и открытых файловых дескрипторов и директорий. Если выполняемый родительский процесс являлся многопоточной программой и в ней работали несколько нитей, то

копируется только нить, которая запустила системный вызов, остальные нити в дочернем процессе бесследно исчезают.

Для запуска нового приложения применяется системный вызов из семейства `exec`, который замещает образ текущего процесса новым. По этой причине для параллельного запуска новой программы совместно с работающей используют последовательность вызовов `fork` и затем `exec` в процессе-потомке. При выполнении `exec` все используемые данные старого процесса освобождаются, но новый процесс наследует те же самые pid и ppid старого процесса, и, кроме того, в новый процесс копируются все открытые файловые дескрипторы (если они не были открыты с флагом `O_CLOEXEC`).

Оба варианта запуска новых процессов мо-

гут быть использованы в Perl для выполнения параллельных задач.

Проблемы использования fork

Описанное поведение `fork` может оказывать нежелательные побочные эффекты на работу приложения. Рассмотрим проблемы, которые могут возникнуть в общем случае, и проблемы, которые специфичны для `AnyEvent`-приложений.

1. Медленное создание копии крупного процесса

Если объём памяти, занимаемой приложением, исчисляется несколькими гигабайтами, то создание копии процесса может

занять ощутимое время. Несмотря на наличие механизма COW, создание процесса по-прежнему требует инициации структуры нового процесса в ядре, копирования таблиц страниц памяти родительского процесса, которые могут занимать существенный объём для огромного процесса. Это также может усугубляться тем, что единственной требуемой операцией в дочернем процессе будет создание нового процесса через вызов `exec`. Это приводит к напрасной трате ресурсов и большим накладным расходам для создания новых процессов.

2. Копирование бесполезных для дочернего процесса данных

Данные, копируемые в дочерний процесс, могут быть абсолютно бесполезны для

дочернего процесса. Благодаря механизму COW они не занимают дополнительного объёма памяти, но это так, пока родительский процесс тоже их не изменяет. В тот самый момент, когда родительский процесс освободит память под уже ненужные данные или обновит их, старая версия этих данных перетечёт в дочерний процесс. Получим парадоксальную ситуацию, когда освобождённая память не освобождается, а просто утекает в другой процесс.

3. Fork может сделать невозможным работу дочернего процесса

Программа может использовать нити через модуль `threads` или загружать модули, которые используют POSIX threads неявно, например модуль `IO::AIO`. Специфика клонирования многопоточного процесса

может привести к тому, что полученный дочерний процесс окажется в неконсистентном состоянии, препятствующем его дальнейшей корректной работе. Такая ситуация возможна, даже если вы не используете нити, а вызываете `fork` внутри обработчика сигнала, который может выполниться в произвольной точке программы, разорвав, например, критичную транзакцию.

4. Обработка событий в дочернем процессе

При создании дочернего процесса в него копируются все действующие объекты, наблюдающие за событиями. Не все событийные библиотеки, работающие в бэкенде `AnyEvent`, способны корректно начать работать в дочернем процессе.

Поэтому обработка событий в дочернем процессе может оказаться невозможной. Даже если библиотека обработки событий гарантирует работоспособность после `fork`, в дочернем процессе могут начать запускаться таймеры или обработчики сигналов, которые имели смысл только в родительском процессе.

5. `Fork+exec` нового процесса интерпретатора Perl может быть не так прост

Не всегда легко обнаружить правильный путь к интерпретатору Perl, в переменной `^X` может содержаться совсем не интерпретатор.

6. `Fork+exec` нового процесса может оказаться неудачным

Часто процесс может выполняться очень длительное время, за которое в окружении могут произойти перемены, например, обновиться модули или даже сам интерпретатор. Попытка `fork+exec` нового процесса может завершиться ошибкой, например, из-за проблемы в обновлённом стороннем модуле.

AnyEvent::Fork

Недавно вышедший модуль `AnyEvent::Fork` Марка Леменна (Marc Lehmann) пытается решить все указанные выше проблемы использования `fork` в `AnyEvent`-приложениях, предоставляя безопасный и гибкий инструмент создания и управления процессами.

Особенность модуля в том, что он создаёт новые процессы путём запуска нового интерпретатора perl или посредством клонирования существующего “шаблонного” процесса. Такой подход позволяет решить половину указанных проблем, связанных с разделением общих данных в процессах (2, 3 и 4).

Для решения проблемы со скоростью работы `fork+exec` в нём используется модуль `Proc::FastSpawn`, который при наличии поддержки на платформе применяет системные вызовы `vfork+exec`. Вызов `vfork` аналогичен `fork` с той лишь разницей, что не происходит копирования структур данных родительского процесса в процесс-потомок. При этом родительский процесс останавливается до тех пор, пока дочерний процесс не завершится или не выполнит вызов `exec`. Это значитель-

но ускоряет запуск нового процесса. На платформе `win32` и других, где нет вызова `vfork`, по возможности используется `spawn` или происходит откат на обычный `fork+exec`.

Модуль также пытается обнаружить нужный интерпретатор perl, анализируя глобальную переменную `^X`, и в случае проблем откатывается на использование `$Config::Config{perlpath}`. Кроме того, существует возможность переопределить путь к интерпретатору через глобальную переменную `$AnyEvent::Fork::PERL`.

Чтобы избежать проблем с обновлённым окружением для долгоработающих процессов, `AnyEvent::Fork` предлагает использование шаблонного процесса, который запускается в самом начале работы

программы и позже при необходимости используется для создания новых процессов через `fork`. Кроме того, это решает и проблему, появившуюся с использованием `exec`, когда требуется каждый раз загружать модули, используемые программой, — использование шаблонного процесса позволяет загрузить необходимые модули один раз.

Использование AnyEvent::Fork

Рассмотрим типичный пример использования:

```
1 use AnyEvent::Fork;
2
3 AnyEvent::Fork
4     ->new
5     ->require ("MyModule")
```

```
6       ->run ("MyModule::server", my
            $cv = AE::cv);
7
8 my $fh = $cv->recv;
```

Объект AnyEvent::Fork создаётся вызовом метода new(). Метод new() помимо создания объекта производит запуск нового процесса интерпретатора perl, который становится “шаблонным” процессом. Из этого шаблона в дальнейшем и формируются все новые процессы посредством вызова fork.

Вызов метода require() позволяет выполнить загрузку модуля в процессе потомка. Метод run() выполняет подпрограмму в процессе-потомке. Первым параметром указывается имя подпрограммы, вторым параметром передаётся колбэк-функция, которая выполняется после того, как будет

запущен процесс потомок.

Первым параметром, который получают запускаемая в новом процессе подпрограмма и колбэк-функция в родительском процессе, — это дескриптор сокета, открытого между процессами. Если использование сокета не требуется, его можно закрыть в этих подпрограммах.

Передача параметров и файловых дескрипторов в дочерний процесс

Для того, чтобы передать дополнительные параметры в подпрограмму, вызываемую в дочернем процессе, используется метод `send_arg()`:

```
1 use AnyEvent::Fork;
2
3 AnyEvent::Fork
```

```
4       ->new
5       ->eval('
6           sub runner {
7               my ($fh, @params) =
                    @_;
8               ...
9           }
10      ')
11      ->send_arg(@params)
12      ->run("runner", my $cv = AE::
            cv);
13
14  my $fh = $cv->recv;
```

Метод передаёт только строковые параметры, а для передачи сложных структур данных требуется использовать сериализацию.

Также существует возможность передавать файловые дескрипторы с помощью метода `send_fh()`:

```
open my $log, '>', '/var/log/file
    ' or die $!;

AnyEvent::Fork
    ->new
    ->eval('
        sub runner {
            my ($fh, $log,
               @params) = @_;
            ...
        }
    ')
    ->send_fh($log)
    ->send_arg(@params)
    ->run("runner", my $cv = AE::
       cv);

my $fh = $cv->recv;
```

Это открывает широкие возможности для коммуникации между процессами родителя и потомка.

Создание и управление пулом процессов

Метод `fork()` используется для клонирования нужного шаблонного процесса:

```
my $pool =
  AnyEvent::Fork
    ->new
    ->require ("MyModule");

my @pool;

for my $id (1..10) {
    push @pool,
      $pool
        ->fork
        ->send_arg($id)
        ->run("MyModule::foo",
           sub {

             my ($fh) = @_;
```

```
16                 print "Process id $id
                       \n";
17                 ...
18             });
19 }
```

Программа создаст пул из десяти процессов из заданного шаблонного процесса.

Работа модуля при отсутствии возможности запуска нового интерпретатора

В некоторых ситуациях бывает невозможно запустить новый интерпретатор perl. Например, он может быть встроенным в другое приложение, которое сейчас выполняет этот код, или выполнение `fork` может испортить работу какого-либо уже загруженного модуля.

Для этих случаев в состав дистрибутива AnyEvent::Fork включены два модуля: AnyEvent::Fork::Early и AnyEvent::Fork::Template.

AnyEvent::Fork::Early должен быть загружен в самом начале вашего приложения до загрузки каких-либо других библиотек. В этом случае происходит форк текущего процесса, который затем и используется для создания новых процессов:

```
#!/usr/bin/perl
use AnyEvent::Fork::Early;

# some other code
```

Действие, которое выполняет AnyEvent::Fork::Template, — практически то же самое, только перед созданием шаблонного процесса есть возможность загрузить некоторые дополнительные модули, ко-

торые станут доступны во всех вновь создаваемых процессах:

```
use Some::Harmless::Module;
use My::Worker::Module;

use AnyEvent::Fork::Template;

use Gtk2 -init;

$AnyEvent::Fork::Template->fork->
    run ("My::Worker::Module::
    run_worker", sub { ... });
```

В данном примере после создания шаблона загружается модуль `Gtk2`, в котором могут возникнуть проблемы при использовании `fork`. В последней строке происходит создание нового процесса из шаблонного процесса, в котором не загружен “опасный” модуль `Gtk2`.

AnyEvent::Fork::RPC

В состав модуля `AnyEvent::Fork` не встроено никаких средств для создания очереди заданий или протокола двустороннего обмена между процессом-потомком и родительским процессом в виде запросов/ответов. Поэтому был создан модуль AnyEvent::Fork::RPC, который предназначен именно для подобных целей. На текущий момент его API имеет статус экспериментального, поэтому вполне вероятны изменения.

Пример использования:

```
use AnyEvent;
use AnyEvent::Fork::RPC;

my $done = AE::cv;

my $rpc = AnyEvent::Fork
```

```
7       ->new
8       ->eval(do { local $/; <DATA>
           })
9       ->AnyEvent::Fork::RPC::run ("
           worker",
10           on_error   => sub { warn
                "FATAL: $_[0]"; exit 1
                 },
11           on_event   => sub { warn
                "$_[0] requests
                handled\n" },
12           on_destroy => $done,
13        );
14
15  for my $id (1..100) {
16      $rpc->( "foo" => "bar", sub {
17          $_[0] or warn "$id: $_
                [1]\n";
18      });
19  }
20
21  undef $rpc;
22
```

```
23 $done->recv;
24
25 __DATA__
26
27 my $count;
28
29 sub worker {
30     my ($command, $data) = @_;
31
32     AnyEvent::Fork::RPC::event (
          $count) unless ++$count %
          10;
33
34     my $status = ...
35
36     $status or (0, "$!")
37 }
```

После обычного создания объекта нового процесса выполняется метод `AnyEvent::Fork::RPC::run`, возвращающий объект, который можно будет использовать

для многократного выполнения вызовов подпрограммы в дочернем процессе. Существует возможность регистрации функций-колбэков на различные события, которые будут происходить при запуске подпрограммы: ошибки, уничтожение объекта и т.п.

В данном примере происходит последовательное заполнение очереди команд на запуск подпрограммы и регистрации функций-колбеков для получения результатов выполнения команды в родительском процессе. После формирования очереди объект `$rpc` освобождается присвоением переменной значения `undef`. Это приведёт к тому, что после выполнения всей очереди заданий процесс потомок будет завершён. Это в свою очередь приведёт к вызову колбэка уничтожения объекта (`on_destroy`).

AnyEvent::Fork::Pool

Если в вашем приложении требуется более тонкое управление пулом процессов, то для этих целей можно использовать модуль AnyEvent::Fork::Pool. Так же как и предыдущий модуль, API данного модуля сейчас находится в альфа-стадии и может активно изменяться.

Пример использования:

```
use AnyEvent;
use AnyEvent::Fork::Pool;

my $pool = AnyEvent::Fork
    ->new
    ->require ("MyWorker")
    ->AnyEvent::Fork::Pool::run (
       "MyWorker::run", # the
          worker function

```

```
10          # pool management
11          max         => 4,   # absolute maximum # of processes
12          idle        => 0,   # minimum # of idle processes
13          load        => 2,   # queue at most this number of jobs per process
14          start       => 0.1, # wait this many seconds before starting a new process
15          stop        => 10,  # wait this many seconds before stopping an idle process
16          on_destroy => (my $finish = AE::cv), # called when object is destroyed
```

```
17
18          # parameters passed to
               AnyEvent::Fork::RPC
19          async       => 0,
20          on_error    => sub { die "
               FATAL: $_[0]\n" },
21          on_event    => sub { my
               @ev = @_ },
22          init        => "MyWorker::
               init",
23          serialiser => $AnyEvent::
               Fork::RPC::
               STRING_SERIALISER,
24        );
25
26  for (1..10) {
27      $pool->(doit => $_, sub {
28          print "MyWorker::run
               returned @_\n";
29      });
30  }
31
32  undef $pool;
```

```
33
34 $finish->recv;
```

В данном примере создаётся пул процессов, которые выполняют функцию `MyWorker::run`. В соответствии с заданными настройками `AnyEvent::Fork::Pool` автоматически регулирует количество необходимых для выполнения заданий процессов, время их работы и необходимость уничтожения.

Параметры для управления пулом:

- idle — указывает минимальное число простаивающих процессов; если количество свободных процессов снижается меньше этого уровня, происходит запуск новых процессов;
- max — максимальное число процес-

сов в пуле;
- load — максимальное количество задач, отправляемых на один рабочий процесс;
- start — задержка запуска нового процесса, в случае, если в соответствии с настройками требуется запуск нового процесса;
- stop — время, через которое рабочий процесс будет остановлен, если он простаивает.

Заключение

Модуль `AnyEvent::Fork` и вспомогательные модули на его основе позволяют безопасно и гибко управлять созданием новых процессов для решения задач по параллельной обработке данных. Успешно

решены проблемы с побочными эффектами `fork` для асинхронных приложений и реализовано универсальное и кроссплатформенное решение.

Возможно, ещё преждевременно рекомендовать модули для использования в промышленной эксплуатации, поскольку API модулей ещё не стабилизированы, но определённо есть смысл начать изучать и экспериментировать. Удачных опытов!

■ *Владимир Леттиев*

6 Что нового в Perl 5.18.0

18 мая 2013 была выпущена новая стабильная версия языка программирования Perl 5.18.0. Разработка велась примерно 12 месяцев, начиная с Perl 5.16.0, и содержит примерно 400 000 изменённых строк среди 2100 файлов от 113 авторов.

Ключевые изменения

Среди большого списка изменений можно выделить ключевые:

- Пересмотрена реализация хешей. По умолчанию два различных хеша с идентичными ключами и значениями теперь могут возвращать их

содержимое в различном порядке, тогда как раньше порядок всегда был одинаковым. Данное изменение затронуло множество модулей на CPAN, которые в некоторых случаях полагались на фиксированный порядок следования ключей хеша. Perl никогда не гарантировал какой-либо упорядоченности ключей в хеше, теперь же он гарантирует их абсолютную беспорядочность.

- Новая хеш-функция по умолчанию — ONE_AT_A_TIME_HARD. Замена хеш-функции — это одно из последствий устранения уязвимости CVE-2013-1667, позволявшей проводить атаки на алгоритмическую сложность против хеш-функции, что могло приводить к DoS.

- Появился новый тип предупре-

ждений — «экспериментальные» (experimental). Такие предупреждения выводятся, когда в коде задействуются экспериментальные возможности. Также появился и способ подавления подобных предупреждений.

- Возможности из семейства умного сравнения теперь являются экспериментальными, а код, который их использует, теперь выводит соответствующие предупреждения. Операторы `~~`, `given` и `when` могут быть полностью переработаны или вообще убраны из языка в будущих версиях Perl. Основная причина этого изменения — чрезвычайно сложная и запутанная таблица действий в зависимости от типа аргументов этих операторов. Умное сравнение требует радикального пересмотра в сторону

упрощения.

- Поддержка стандарта Юникод 6.2. А также возможность скомпилировать Perl с любой старой версией Юникода по выбору.

- Несколько интересных экспериментальных возможностей: лексические подпрограммы, видные только в области видимости, в которой они определены, и возможность использовать операции со множествами в регулярных выражениях для кодов символов.

- Внушительное число исправленных ошибок.

perldelta

Полный список изменений описан в perldelta.pod. Далее представлен перевод этого документа на русский язык. Оригинальный файл перевода в формате POD доступен на github.

Имя

perl5180delta — что нового в perl v5.18.0

Описание

Этот документ описывает различия между релизом v5.16.0 и релизом v5.18.0.

Если вы обновляетесь с более раннего релиза, такого как v5.14.0, сначала прочтите perl5160delta, который описывает различия между v5.14.0 и v5.16.0.

Улучшения ядра

Новый механизм экспериментальных возможностей

Впервые добавляемые экспериментальные возможности теперь требуют подобного вызова:

```
no warnings "experimental::
    feature_name";
use feature "feature_name";
```

Появилась новая категория предупреждений, называемых «экспериментальными», содержащие предупреждения, которые выдаёт прагма feature, когда подключаются экспериментальные возможности.

Впервые добавляемые экспериментальные возможности также будут содержать специальные идентификаторы предупреждения, которые состоят из `experimental::` с последующим названием возможности. (Есть план для расширения действия этого механизма в конечном счёте на все предупреждения, позволяя включать или отключать их персонально, а не только по категориям.)

Указывая

```
no warnings "experimental::
    feature_name";
```

вы берёте ответственность за любую поломку, которая может возникнуть в связи с изменением или удалением соответствующей возможности.

Так как некоторые возможности (такие как `~~` или `my $_`) теперь выводят предупреждение об их экспериментальном характере и вам может потребоваться отключить их в коде, который может также быть запущен под perl, который не распознаёт этих категорий предупреждений, рассмотрите возможность использования прагмы `if` таким образом:

```
no if $] >= 5.018, 'warnings', "
    experimental::feature_name";
```

Существующие экспериментальные возможности также могут начать выводить эти предупреждения. Пожалуйста, обратитесь к perlexperiment за информацией о тех

возможностях, которые рассматриваются как экспериментальные.

Пересмотр реализации хеша

Изменения в реализации хешей в perl v5.18.0 станут наиболее заметными изменениями в поведении существующего кода.

По умолчанию два различных хеша с идентичными ключами и значениями могут возвращать их содержимое в различном порядке, тогда как раньше порядок был одинаковым.

При столкновении с этими изменениями ключевым моментом для исправления последствий станет принятие правила, что **хеши — это беспорядочные коллекции** и

действие в соответствии с ним.

Рандомизация хеша Ключ (seed), используемый хеш-функцией Perl, теперь является случайным. Это означает, что порядок, в котором ключи/значения будут возвращены из функций `keys()`, `values()` и `each()`, будет различаться от запуска к запуску.

Это изменение было сделано, чтобы сделать хеши Perl более устойчивыми к атакам на алгоритмическую сложность, а также мы обнаружили, что это выявляет ошибки зависимости от порядка хеша, делая их простыми для обнаружения.

Для разработчиков утилит сборки есть повод вложить силы в дополнительную инфраструктуру для тестирования подобных вещей. Запуская тесты несколько раз

подряд и сравнивая результаты, можно легко выявить зависимости от порядка хеша в коде. Авторам настойчиво рекомендуется не демонстрировать ключ рандомизации хешей Perl для небезопасной публики.

Кроме того, каждый хеш имеет свой собственный порядок итерации, что делает значительно более сложным определение того, какой ключ рандомизации хеша используется в данный момент.

Новые хеш-функции Perl v5.18 включает поддержку множества хеш-функций и меняет хеш-функцию по умолчанию (на ONE_AT_A_TIME_HARD), вы можете выбрать другой алгоритм, указывая параметр при компиляции. Для просмотра текущего списка обратитесь к документу `INSTALL`. Обратите внимание, что начиная с Perl

v5.18 мы можем рекомендовать использовать только хеш-функцию по умолчанию или SIPHASH. Все другие имеют известные проблемы с безопасностью и могут использоваться только для исследовательских целей.

Переменная окружения PERL_HASH_SEED теперь принимает шестнадцатиричное значение PERL_HASH_SEED больше не принимает целочисленное значение как параметр; вместо этого ожидается бинарное значение, закодированное в шестнадцатиричную строку, как например, “0xf5867c55039dc724”. Это сделано для того, чтобы поддерживать ключи рандомизации хеша произвольной длины, которые могут превышать лимит целого числа (int). (SipHash использует ключ в 16 байт).

Добавлена переменная окружения PERL_PERTURB_KEYS Переменная окружения PERL_PERTURB_KEYS позволяет контролировать уровень случайности, применяемый к keys и подобным.

Когда PERL_PERTURB_KEYS установлен в 0, perl не будет рандомизировать порядок ключей. Шанс, что изменения в keys из-за вставки будут такими же как в предыдущих perl, в основном из-за изменения размера ячейки.

Когда PERL_PERTURB_KEYS установлен в 1, perl будет рандомизировать ключи в неповторяющемся порядке. Шанс, что вывод keys изменится в результате вставки, крайне высок. Это наиболее безопасно и является режимом по умолчанию.

Когда PERL_PERTURB_KEYS установлен

в 2, perl будет рандомизировать ключи в воспроизводимом порядке. Повторные запуски одной и той же программы каждый раз будут выводить один и тот же результат.

PERL_HASH_SEED подразумевает установку нестандартного PERL_PERTURB_KEYS. Устанавливая PERL_HASH_SEED=0 (именно один 0) устанавливает PERL_PERTURB_KEYS=0 (рандомизация хеша отключена); установка PERL_HASH_SEED в любое другое значение подразумевает PERL_PERTURB_KEYS=2 (детерминированную и повторяемую рандомизацию хеша). Явное указание PERL_PERTURB_KEYS в другое значение переопределяет это поведение.

Hash::Util::hash_seed() теперь возвращает строку Hash::Util::hash_seed() теперь воз-

вращает строку, вместо целого числа. Это необходимо, чтобы сделать инфраструктурную поддержку для ключей рандомизации хеша произвольной длины, превышающей размер целого числа. (SipHash использует ключ размером 16 байт).

Изменился вывод PERL_HASH_SEED_DEBUG
Переменная окружения PERL_HASH_SEED_DE теперь заставляет perl показывать оба параметра: хеш-функцию, с которой был собран perl, *и* ключ рандомизации в шестнадцатиричном виде, используемый в текущем процессе. Существующий код, обрабатывающий этот вывод, должен быть изменён под новый формат. Пример нового формата:

```
$ PERL_HASH_SEED_DEBUG=1 ./perl -
    e1
```

```
2 HASH_FUNCTION = MURMUR3 HASH_SEED
      = 0x1476bb9f
```

Обновление до Юникода 6.2

Perl теперь поддерживает стандарт Юникода версии 6.2. Список отличий от Юникода 6.1 доступен на http://www.unicode.org/versions/Unicode6.2.0.

Псевдоним символа теперь может включать символы не из диапазона Latin1

Теперь возможно определить ваши собственные имена для использования в `\N{...}`, `charnames::vianame()` и т.п. Эти имена теперь могут включать сим-

волы из всего диапазона Юникода. Это позволяет использовать имена на вашем родном языке, а не только на английском. Некоторые ограничения накладываются на символы, которые могут быть использованы (вы не можете использовать имена со знаками пунктуации внутри, например). Смотрите “CUSTOM ALIASES” in charnames.

Новые проверки DTrace

Следующие новые датчики (probes) были добавлены в DTrace:

- `op-entry`
- `loading-file`
- `loaded-file`

${^LAST_FH}

Эта новая переменная даёт доступ к последнему файловому дескриптору, из которого происходило чтение. Это дескриптор, который используется в `$.`, `tell` и `eof` без аргументов.

Операции со множествами в регулярных выражениях

Это **экспериментальная** возможность, которая позволяет проверять совпадения в объединении, пересечении и другими операциями со множествами кодов символов, подобно модулю Unicode::Regex::Set. Это также может использоваться для расширения возможностей обработки в `/x` [заключённых в скобки] классов симво-

лов и как замена свойств, определённых пользователем, позволяя создавать более сложные выражения, чем это было возможно до этого. Смотрите “Extended Bracketed Character Classes” in perlrecharclass.

Лексические подпрограммы

Эта возможность по-прежнему рассматривается **экспериментальной**. Для её включения необходимо:

```
use 5.018;
no warnings "experimental::
    lexical_subs";
use feature "lexical_subs";
```

Теперь вы можете декларировать подпрограммы с помощью `state sub foo`, `my sub foo` и `our sub foo`. (`state sub`

также требует, чтобы возможность “state” была включена, если только вы не записываете её в виде `CORE::state sub foo`.)

`state sub` создаёт подпрограмму, видимую внутри лексической области видимости, в которой она определена. Подпрограмма является разделяемой между вызовами внешней подпрограммы.

`my sub` декларирует лексическую подпрограмму, видимую внутри лексической области видимости, в которой она определена. `state sub` в целом работает немного быстрее, чем `my sub`.

`our sub` декларирует лексический псевдоним для подпрограммы пакета с тем же именем.

Для более подробной информации обратитесь к “Lexical Subroutines” in perlsub.

Вычисляемые метки

Для управления циклом `next`, `last` и `redo`, а также специального оператора `dump`, теперь можно использовать произвольное выражение для вычисления меток во время работы программы. Ранее любые аргументы, которые не являлись константами, рассматривались как пустая строка.

Больше CORE::-подпрограмм

Ещё несколько встроенных функций было добавлено как подпрограммы в

пространство имён CORE::, т.е. те непереопределяемые ключевые слова, которые могут быть определены без специальных парсеров: `defined`, `delete`, `exists`, `glob`, `pos`, `protoytpe`, `scalar`, `split`, `study` и `undef`.

Так как некоторые из них имеют прототипы, `prototype('CORE::...')` был изменён таким образом, чтобы не делать различий между переопределяемыми и непереопределяемыми ключевыми словами. Это было необходимо, чтобы согласовать `prototype('CORE::pos')` с `prototype(&CORE::pos)`.

`kill` с отрицательным именем сигнала

`kill` всегда допускал использование отрицательного номера сигнала, который

убивал группу процессов, вместо одного процесса. Также он разрешал использовать и имена сигналов. Но вёл себя непоследовательно, поскольку отрицательные имена сигналов рассматривались как 0. Теперь отрицательные имена сигналов, как например, `-INT` поддерживаются и рассматриваются по аналогии, что и -2 [perl #112990].

Безопасность

Смотрите также: пересмотр реализации хеша

Некоторые изменения в пересмотре реализации хеша были сделаны для улучшения безопасности. Пожалуйста, прочтите эту секцию.

Предупреждение о безопасности в документации `Storable`

Документация `Storable` теперь включает секцию, которая предупреждает читателя об опасности принятия данных формата Storable от источников, к которым нет доверия. В кратком изложении, десереализация некоторых типов данных может привести к загрузке модулей и другим вариантам выполнения кода. Это задокументированное и ожидаемое поведение, но, с другой стороны, открывает вектор атаки для злоумышленников.

`Locale::Maketext` позволял включение произвольного кода через специальный темплейт

Если пользователи имеют возможность предоставлять строки переводов для `Locale::Maketext`, это может быть использовано для вызова произвольной Perl-подпрограммы, доступной в текущем процессе.

Это было исправлено, но по-прежнему доступна возможность вызова любого метода, предоставляемого самим `Locale::Maketext` или подклассом, который вы используете. Один из таких методов, в свою очередь, выполняет встроенную Perl подпрограмму `sprintf`.

В целом, разрешать пользователям делать перевод без аудита их результата является

плохой идеей.

Эта уязвимость задокументирована в CVE-2012-6329.

Избегайте вызова memset с отрицательным числом

Плохо написанный Perl-код, который даёт возможность атакующему указывать число повторов строки в операторе x, уже подвержен атакам на отказ в обслуживании при исчерпании памяти. Уязвимость в версиях perl до v5.15.5 позволяет, сверх того, привести к переполнению кучи, что в совокупности с использованием glibc версии до 2.16 позволяет привести к выполнению произвольного кода.

Эта уязвимость получила идентификатор

CVE-2012-5195, и была обнаружена Тимом Брауном (Tim Brown).

Несовместимые изменения

Смотрите также: пересмотр реализации хеша

Некоторые изменения в пересмотре реализации хеша несовместимы с предыдущими версиями perl. Пожалуйста, прочтите эту секцию.

Неизвестное имя символа в `\N{...}` теперь является синтаксической ошибкой

Ранее выводилось предупреждение и производилось замещение Юникод-символом замещения REPLACEMENT CHARACTER. Стандарт Юникод теперь рекомендует обрабатывать данную ситуацию как ошибку. Кроме того, предыдущее поведение приводило к некоторым сбивающим с толку предупреждениями и действиям, и, так как символ REPLACEMENT CHARACTER не имеет других применений, кроме как замены для неизвестного символа, любой код, который имеет такую проблему, — кривой.

Ранее устаревший символ в `\N{}`-псевдониме символа теперь является ошибкой

Начиная с v5.12.0 было помечено устаревшим использование некоторых символов в определённых пользователем `\N{...}` -именах символов. Теперь это вызывает синтаксическую ошибку. Например, теперь считается ошибкой начинать имена с цифр, как в случае

```
my $undraftable = "\N{4F}";     # Syntax error!
```

или иметь запятые где угодно в имени. Смотрите “CUSTOM ALIASES” in charnames.

`\N{BELL}` теперь соответствует коду U+1F514 вместо U+0007

Unicode 6.0 использовал имя “BELL” для другого кода символа вместо традиционного значения. Начиная с Perl 5.14 использование этого имени по-прежнему ссылалось на код U+0007, но выводило предупреждение об устаревшей конструкции. Теперь “BELL” ссылается на код U+1F514, а именем для U+0007 стало “ALERT”. Все соответствующие функции в charnames были обновлены.

Новые ограничения в многосимвольном нечувствительном к регистру сравнении в регулярных выражениях для заключённых в скобки классах символов

Юникод теперь убрал свои ранние рекомендации для регулярных выражений для автоматической обработки случаев, когда один символ может совпадать с несколькими символами в нечувствительном к регистру сравнении, например, буква LATIN SMALL LETTER SHARP S и последовательность `ss`. Поскольку оказалось, что это не может быть выполнено корректно во всех случаях. Так как Perl пытается сделать сделать всё от него возможное, он продолжит делать так, как раньше. (Мы рассматриваем вариант опции для отключения этого поведения). Однако, новые ограничения были добавлены на подобные сравнения, когда они проис-

ходят в [заключённых в скобки] классах символов. Люди, указывающие вещи наподобии `/[\0–\xff]/i`, бывают удивлены, когда это совпадает с последовательностью из двух символов `ss` (поскольку LATIN SMALL LETTER SHARP S входит в этот диапазон). Это поведение также рассогласованно с использованием свойств вместо диапазонов: `\p{Block=Latin1}` также включает LATIN SMALL LETTER SHARP S, но `/[\p{Block=Latin1}]/i` не совпадает с `ss`.

По новым правилам, чтобы происходило совпадение при регистронезависимом сравнении с несколькими символами внутри заключённого в скобки класса символов, требуется, чтобы символ был указан там явно и не являлся конечным символом диапазона. Это хорошо соответствует принципу наименьшего удивления.

Смотрите “Bracketed Character Classes” in perlrecharclass. Обратите внимание, что ошибка [perl #89774] была исправлена как часть этого изменения, препятствовавшая работе сравнения в полной мере.

Ясные правила для имён переменных и идентификаторов

В результате недосмотра односимвольные имена переменных в v5.16 были полностью неограниченны. Это открывало двери для разного рода безумств. Начиная с v5.18 они теперь следуют правилам других идентификаторов и в дополнении допускающих использование символов, соответствующих свойству `\p{POSIX_Punct}`.

Теперь нет никакой разницы в разборе идентификаторов, указанных с исполь-

зованием фигурных скобок и без них. Например, perl допускал использование `${foo:bar}` (с одиночным двоеточием), но не допускал `$foo:bar`. Теперь оба случая обрабатываются одним участком кода и оба обрабатываются одинаково: оба запрещены. Обратите внимание, что это изменение касается границ допустимого в задании буквенных идентификаторов, а не других выражений.

Вертикальная табуляция теперь пробельный символ

Никто не может вспомнить, почему `\s` не совпадает с `\cK` — вертикальной табуляцией. Теперь совпадает. Учитывая экстремальный раритет данного символа, ожидается очень небольшое число поломок. Как было сказано, теперь это означает, что:

`\s` в регулярных выражениях всегда совпадает с вертикальной табуляцией при любых условиях.

Литеральные вертикальные табуляции в регулярных выражениях теперь игнорируются, когда используется модификатор `/x`.

Лидирующие символы вертикальной табуляции, поодиночке или вперемешку с другими пробельными символами, теперь игнорируются при интерпретации строки как числа. Например:

```
$dec = " \cK \t 123";
$hex = " \cK \t 0xF";

say 0 + $dec;   # было 0 с предупреждением, теперь 123
say int $dec;   # было 0, стало 123
say oct $hex;   # было 0, стало 15
```

`/(?{})/` и `/(??{})/` были сильно переработаны

Реализация этой возможности была практически полностью переписана. Хотя основной целью было исправление ошибок, некоторое поведение, особенно связанное с областью видимости лексических переменных, было изменено. Это описано более полно в секции «Некоторые исправленные ошибки».

Более строгий синтаксический разбор подстановки замены

Теперь больше невозможно злоупотреблять особенностью разбора синтаксиса `s///e`, как в данном случае:

```
%_=(_,"Just another ");
```

```
$_="Perl hacker,\n";
s//_}->{_/e;print
```

`given` теперь ссылается на глобальную `$_`

Вместо присвоения явной лексической переменной `$_`, `given` теперь делает глобальную `$_` псевдонимом для своего аргумента, так же как и `foreach`. Однако он по-прежнему использует лексическую `$_`, если присутствует лексическая `$_` в данной области видимости (и снова, также как `foreach`) [perl #114020].

Возможности из семейства умного сравнения теперь являются экспериментальными

Умное сравнение, которое было добавлено в v5.10.0 и значительно пересмотрено в v5.10.1, постоянно было источником недовольства. Несмотря на некоторое число полезных возможностей, оно всё же признавалось проблемным и сбивающим с толку как для пользователей, так и разработчиков Perl. Было сделано несколько предложений по исправлению проблемы. Стало ясно, что умное сравнение практически наверняка будет либо изменено или исчезнет в будущем. Полагаться на его текущее поведение не рекомендуется.

Теперь будут выдаваться предупреждения, когда парсер будет видеть `~~`, `given` или `when`. Для отключения этих предупрежде-

ний, вы можете добавить данную строку в соответствующую область видимости:

```
no if $] >= 5.018, 'warnings', "
    experimental::feature_name";
```

Тем не менее, рассматривайте замену использования этих возможностей, так как они могут изменить своё поведение снова перед тем как вновь станут стабильными.

Лексическая $_ теперь является экспериментальной

Начиная со своего появления в Perl v5.10, она вызывала много путаницы без ясного решения:

- Различные модули (например, List::Util) ожидают, что колбэк-

функции используют глобальную `$_`. `use List::Util 'first'; my $_; first { $_ == 1 } @list` не будет работать так, как ожидается.

- Объявление `my $_` ранее в том же файле может вызывать вывод сбивающего с толку предупреждения замыкания.
- Символ _ в прототипе подпрограммы позволяет вызываемой подпрограмме иметь доступ к вашей лексической `$_`, так что она в итоге не является по-настоящему приватной.
- Несмотря на это, подпрограммы с прототипом (`@`) или методы не могут получить доступ к лексической переменной `$_` вызывающего кода, если только они не написаны с использованием XS.
- Но даже XS-подпрограммы не могут получить доступ к лексической

переменной $_, объявленной не в вызывающей подпрограмме, а во внешней области видимости, если эта подпрограмма не упоминает $_ или не использует операции, которые по умолчанию взаимодействуют с $_.

Мы надеемся, что лексическая $_ может быть реабилитирована, но это может вызвать изменения в её поведении. Пожалуйста, используйте её с осторожностью, пока она не станет стабильной.

readline() в `$/ = \N` теперь читает N символов, а не N байтов

Ранее при чтении из потока со слоями I/O, такими как `encoding`, функция readline(),

иначе известная как оператор <>, читала *N* байт с верхнего слоя. [perl #79960]

Теперь вместо этого читается *N* символов.

Никаких изменений в поведении при чтении из потоков без дополнительных слоёв нет, так как байты точно соответствуют символам.

Переопределённый `glob` теперь передаёт один аргумент

Переопределение `glob` раньше передавало магический недокументированный второй аргумент, который идентифицировал вызывающий его код. Ничего на CPAN не использовало это, поэтому изменение было внесено как исправление бага и аргумент был удалён. Если вам действи-

тельно необходимо идентифицировать вызывающую подпрограмму, смотрите Devel::Callsite на CPAN.

Синтаксический разбор встроенной документации (here doc)

Тело встроенного документа внутри оператора цитирования теперь всегда начинается на строке после маркера `"<<foo"`. Ранее было задокументировано, что оно начинается на следующей строке, содержащей оператор цитирования, но это было лишь изредка так [perl #114040].

Цифробуквенные операторы должны быть теперь отделены от закрывающего ограничителя регулярного выражения

Теперь вы не можете записывать так:

```
m/a/and 1
```

Вместо этого вы должны записать:

```
m/a/ and 1
```

с пробелом, отделяющим оператор от закрывающего ограничителя регулярного выражения. Отсутствие пробела приведёт к выводу предупреждения об устаревшем функционале начиная с Perl v5.14.0.

qw(...) теперь не может быть использован в качестве скобок

Списки `qw` раньше заставляли парсер думать, что они всегда заключены в скобки. Это позволяло использовать удивительные конструкции, такие как `foreach $x qw (a b c){...}`, которые на самом деле должны были быть записаны `foreach $x (qw(a b c)){...}`. Это иногда приводило лексический анализатор в неверное состояние, поэтому они не работали в полной мере, и схожая запись `foreach qw(a b c){...}`, которая ожидалась дозволительной, вообще никогда не работала.

Эти побочные эффекты `qw` теперь удалены. Они были помечены устаревшими начиная с Perl v5.13.11. Теперь обязательно требуется использовать скобки везде, где

грамматика требует их.

Взаимодействие лексических и обычных предупреждений

Включение любых лексических предупреждений использовалось раньше для отключения всех предупреждений по умолчанию, если лексические предупреждения не были ещё подключены:

```
$*; # deprecation warning
use warnings "void";
$#; # void warning; no
    deprecation warning
```

Теперь категории `debugging`, `deprecated`, `glob`, `inplace` и `malloc` предупреждений остаются включёнными при включении лексических предупреждений (разумеется,

если они не отключены через `no warnings`).

Это может привести к появлению предупреждений об устаревших конструкциях в коде, который раньше не выдавал предупреждений.

Это единственные подобные категории, состоящие только из предупреждений по умолчанию. Предупреждения по умолчанию в других категориях по-прежнему отключены при использовании `use warnings "category"`, так как у нас пока нет инфраструктуры за контролем индивидуальных предупреждений.

`state sub` и `our sub`

По случайности, `state sub` и `our sub` были эквивалентны обычному `sub`, так что была возможность даже создать анонимную подпрограмму с помощью `our sub { ... }`. Теперь это запрещено вне действия возможности “lexical_subs”. При включении возможности “lexical_subs” они получает новый смысл, описанный в “Lexical Subroutines” in perlsub.

Значения, хранимые в переменных окружения, принудительно переводятся в байтовые строки

Значение, хранимое в переменной окружения, всегда приводилось к строке. В данном выпуске оно конвертируется в

байтовую строку. Раньше она приводилась только к строковому виду. Если строка utf8 и аналог `utf8::downgrade()` срабатывает, то используется полученный результат, в противном случае используется эквивалент `utf8::encode()` и выводится предупреждение о широком символе (“Diagnostics”).

`require` умирает на нечитаемых файлах

Когда `require` обнаруживает нечитаемый файл, он теперь умирает. Раньше файл игнорировался и продолжался поиск в директориях в `@INC` [perl #113422].

`gv_fetchmeth_*` и SUPER

Различные XS-функции `gv_fetchmeth_*` раньше рассматривали пакет, чьё имя заканчивается на `::SUPER`, по-особому. Поиск метода в пакете `Foo::SUPER` рассматривался как поиск `SUPER` метода в пакете `Foo`. Больше это не так. Для поиска метода `SUPER` передавайте стэш `Foo` и флаг `GV_SUPER`.

Первый аргумент `split` теперь интерпретируется более согласовано

После некоторых изменений ранее в v5.17, поведение `split` было упрощено: если аргумент шаблона вычисляется в строку, содержащую один пробел, то оно обрабатывается также как и *литеральная* строка, содер-

жащая один пробел как было раньше.

Устаревшие конструкции

Удаление модулей

Следующие модули будут удалены из базового дистрибутива в будущих релизах и потребуют в будущем установки со CPAN. В дистрибутивах на CPAN, требующие эти модули, потребуется указывать как зависимости.

Версии модулей в базовом дистрибутиве теперь будут выдавать предупреждения из категории *устаревшее* для предупреждения вас об этом факте. Для отключения этих предупреждений об устаревании, установите эти модули с CPAN.

Обратите внимание, что (за редким исключением) это прекрасные модули, которые мы призываем вас продолжать использовать. Их исключение из базового состава — в основном следствие необходимости предварительной подготовки (bootstrapping) полнофункциональной, CPAN-совместимой инсталляции Perl, а не обеспокоенность их дизайном.

- encoding

  Использование этой прагмы теперь крайне не рекомендуется. Она объединяет кодирование исходного текста и кодирование данных ввода/вывода, повторно интерпретирует управляющие последовательности в исходном тексте (сомнительный выбор) и приводит к UTF-8-багам во всех приложениях, обрабатываю-

щим строки символов. Она сломана как и ожидалось и не может быть исправлена.

Для использования не-ASCII литеральных символов в исходном коде, пожалуйста смотрите utf8. Для работы с текстовым данными ввода/-вывода, пожалуйста смотрите Encode и open.

- Archive::Extract
- B::Lint
- B::Lint::Debug
- CPANPLUS и все включенные в `CPANPLUS::*` модули
- Devel::InnerPackage
- Log::Message
- Log::Message::Config

- Log::Message::Handlers
- Log::Message::Item
- Log::Message::Simple
- Module::Pluggable
- Module::Pluggable::Object
- Object::Accessor
- Pod::LaTeX
- Term::UI
- Term::UI::History

Устаревшие утилиты

Следующие утилиты будут удалены из базового дистрибутива в будущих релизах,

так как связанные с ними модули устарели. Они по-прежнему будут доступны в соответствующем CPAN-дистрибутиве.

- cpanp
- `cpanp-run-perl`
- cpan2dist

  Это утилиты — часть дистрибутива `CPANPLUS`.
- pod2latex

  Эта утилита — часть дистрибутива `Pod::LaTeX`.

PL_sv_objcount

Эта глобальная в интерпретаторе переменная, раньше учитывала общее число

Perl-объектов в интерпретаторе. Она более не сопровождается и будет удалена в Perl v5.20.

Пять дополнительных символов должны быть экранированы в шаблонах с /x

В шаблонах регулярных выражений, компилируемых с ключом /x, Perl игнорирует 6 символов пробельного типа, такие как пробел и табуляция. Однако, в стандарте Юникод рекомендуется рассматривать как пробельные 11 символов. Мы будем соответствовать этому стандарту в будущей версии Perl. В данный момент, использование этих недостающих символов в неэкранированном виде будет приводить к выводу предупреждений, если предупреждения не отключены. Вот эти 5 символов:

```
U+0085 NEXT LINE
U+200E LEFT-TO-RIGHT MARK
U+200F RIGHT-TO-LEFT MARK
U+2028 LINE SEPARATOR
U+2029 PARAGRAPH SEPARATOR
```

Определённые пользователем имена символов с неожиданными пробелами

Определённые пользователем имена символов с завершающими или несколькими подряд идущими пробелами с большой вероятностью являются опечатками. Теперь такая запись выдаёт предупреждение на основании того, что его использование вряд ли требует избыток пробелов.

Различные функции, вызываемые в XS-программах, теперь устарели

Все функции, использовавшиеся для классификации символов, будут удалены из будущей версии Perl и не должны использоваться. При помощи C-компилятора (например, gcc) сборка любого файла, которая использует любую из этих функций, будут выдавать предупреждение. Они не были предназначены для публичного использования; существуют эквивалентные более быстрые макросы для большинства из них.

Смотрите “Character classes” in perlapi. Это полный список:

```
is_uni_alnum, is_uni_alnumc, is_uni_alnu
,    is_uni_alnum_lc,    is_uni_alpha
,    is_uni_alpha_lc,    is_uni_ascii
```

, is_uni_ascii_lc, is_uni_blank
, is_uni_blank_lc, is_uni_cntrl
, is_uni_cntrl_lc, is_uni_digit
, is_uni_digit_lc, is_uni_graph,
is_uni_graph_lc, is_uni_idfirst,
is_uni_idfirst_lc, is_uni_lower
, is_uni_lower_lc, is_uni_print
, is_uni_print_lc, is_uni_punct
, is_uni_punct_lc, is_uni_space
, is_uni_space_lc, is_uni_upper,
is_uni_upper_lc, is_uni_xdigit,
is_uni_xdigit_lc, is_utf8_alnum
, is_utf8_alnumc, is_utf8_alpha,
is_utf8_ascii,is_utf8_blank,is_utf8_ch
,is_utf8_cntrl,is_utf8_digit,is_utf8_g
, is_utf8_idcont, is_utf8_idfirst,
is_utf8_lower,is_utf8_mark,is_utf8_per
,is_utf8_perl_word,is_utf8_posix_digit
,is_utf8_print,is_utf8_punct,is_utf8_s
, is_utf8_upper, is_utf8_xdigit,
is_utf8_xidcont, is_utf8_xidfirst

.

В дополнении к этому три функции, которые никогда не работали правильно, теперь объявлены устаревшими: `to_uni_lower_lc`, `to_uni_title_lc` и `to_uni_upper_lc`.

Определённые редкие случаи использования обратной косой черты внутри регулярных выражений теперь являются устаревшими

Существуют три пары символов, которые Perl распознаёт как метасимволы в шаблонах регулярных выражений: `{}`, `[]` и `()`. Они могут использоваться для разделения шаблонов, как например:

```
m{foo}
```

```
s(foo)(bar)
```

Поскольку они являются метасимволами, они имеют специальное значение в шаблонах регулярных выражений, и оказывается, что вы не можете отключить это специальное значение при обычном использовании предшествующего символа обратной косой, если вы используете их сдвоенными внутри шаблона, ограниченного ими же. Например:

```
m{foo\{1,3\}}
```

обратная косая черта не изменяет поведение и это совпадает с `"fo"` с последующим одним или тремя включениями `"o"`.

Подобное использование, где они интерпретируются как метасимволы, крайне редко; мы полагаем, что такого нет, на-

пример, на всём CPAN. Следовательно, эта устаревшая конструкция затронет очень немного кода. Однако, будет выдано уведомление, что любой подобный код должен быть изменён, что в свою очередь даст нам возможность изменить поведение будущих версий Perl так, что обратная косая черта будет действовать без страха, что мы молча сломаем чей-то существующий код.

Разделение символов `(?` и `(*` в регулярных выражениях

Теперь выводится предупреждение об устаревшей конструкции, если `(` и `?` разделены пробелом или комментарием в шаблонах регулярного выражения `(?...)`. То же относится к случаю, если `(` и `*` разделены в конструкции `(*VERB...)`.

Пред-PerlIO реализации ввода/вывода

В теории вы можете собрать perl без PerlIO. Вместо него вам придётся использовать обёртку вокруг stdio и sfio. На практике это не очень полезно. Это не так хорошо протестировано, и без какой-либо поддержки для слоёв ввода/вывода или (таким образом) Юникода это лишь небольшой кусок perl. Возможность сборки без PerlIO скорее всего будет удалена в следующей версии perl.

PerlIO поддерживает слой `stdio`, если желательно использование stdio. Точно так же слой sfio может быть создан в будущем, если будет нужен.

Устаревающий функционал

- Платформы без инфраструктурной поддержки

  Обе платформы Windows CE и z/OS исторически поддерживались и на данный момент не собираются успешно и нерегулярно тестируются. Прилагаются усилия для изменения ситуации, но не гарантируется, что платформы безопасны и поддерживаемы. Если они не начнут собираться и регулярно тестироваться, их поддержка может быть удалена в будущих релизах. Если у вас есть интерес к этим платформам, и вы можете потратить своё время, опыт или аппаратное обеспечение для помощи в поддержке этих платформ, пожалуйста, дайте знать

разработчикам, написав письмо на perl5-porters@perl.org.

Некоторые платформы, которые, похоже, окончательно мертвы, также в списке на удаление к v5.20.0:

- DG/UX
- NeXT

Также мы думаем, что, вероятно, текущие версии Perl больше не собираются на AmigaOS, DJGPP, NetWare (нативно), OS/2 и Plan 9. Если вы используете Perl на этих платформах и имеете интерес к обеспечению будущего Perl на них, пожалуйста свяжитесь с нами.

Мы уверенны, что Perl уже долго не может быть собран на смешанных endian архитектурах (такие как PDP-11), и намереваемся удалить весь

оставшийся код их поддержки. Также код поддержки давно неподдерживаемого GNU dld будет скоро удалён, если никто не заявит о себе как о заинтересованном пользователе.

- Обмен < >

  Perl поддерживал идиому обмена < > (а также ()) для временного сброса привилегий, начиная с 5.0, как например:

  ```
  1 ($<, $>) = ($>, $<);
  ```

  Однако такая идиома, изменяющая реальный идентификатор пользователя/группы и которая может иметь нежелательные побочные эффекты, больше не является полезной на любых платформах, поддерживаемых perl и усложняет реализацию этих переменных и присвоения списком в целом.

Как альтернатива, рекомендуется присваивать только $>:

```
local $> = $<;
```

Смотрите также: Setuid Demystified.

- `microperl` давно сломан и неясно его нынешнее предназначение, будет удалён.

- Поправлена семантика `"\Q"` в строках с двойными кавычками при комбинации с другими экранирующими последовательностями.

  Присутствовало несколько ошибок и противоречий, затрагивающих комбинацию `\Q` и экранированных записей `\x`, `\L` и т.п. внутри пары `\Q...\E`. Это требует исправления, что это обязательно приведёт к изменению текущего поведения. Эти изменения пока ещё не устоялись.

- Использование записи `$x`, где `x` представляет собой любой (непечатный) контрольный символ, будет запрещено в будущей версии Perl. Используйте вместо этого `${x}` (где снова `x` означает управляющий символ), а ещё лучше, `$^A`, где `^` — это знак вставки (CIRCUMFLEX ACCENT), а `A` означает любой символ, указанный в списке в конце “OPERATOR DIFFERENCES” in perlebcdic.

Увеличение производительности

- Списочное определение лексических переменных (`my($x, $y)`) теперь оптимизировано к одной операции и, следовательно, работает быстрее, чем раньше.

- Была добавлена новая константа для C-препроцессора `NO_TAINT_SUPPORT`, которая, если установлена, полностью отключает поддержку taint-режима. Использование флагов -T или -t в командной строке приведёт к фатальной ошибке. Будьте осторожны, поскольку и базовые тесты, и многие тесты CPAN-дистрибутивов будут провалены с этим изменением. С другой стороны, предоставляется небольшое повышение производительности, связанное с уменьшением ветвления кода.

  Не устанавливайте эту константу, если вы точно не понимаете, к чему это вас приведёт.

- `pack` с аргументом-константой теперь вычисляется в константу в большинстве случаев. [perl #113470].

- Ускорение регулярных выражений при поиске совпадений с Юникод-свойствами. Наиболее заметное ускорение получено для `\X`, «расширенный кластер графем» Юникода. Ускорение для него составляет где-то 35-40%. Заключённые в скобки классы символов, например, `[0–9\x{100}]`, содержащие свыше 255 кодовых точек, также стали быстрее.

- На платформах, поддерживающих их, некоторые бывшие макросы теперь были реализованы как статические встроенные (inline) функции. Это должно ускорить их значительно на не-GCC платформах.

- Оптимизация хешей в логическом контексте было расширено и на случаи `scalar(%hash)`, `%hash ? ... : ...` и `sub { %hash || ... }`.

- Операторы файловых тестов управляют стеком немного более эффективным способом.

- Глобы, используемые в числовом контексте, теперь приводятся к числу в большинстве случаев непосредственно, а не путём предварительного приведения к строковому виду.

- Оператор повторения x теперь приводится к простой константе на этапе компиляции, если вызывается в скалярном контексте с постоянным операндом и без скобок вокруг левого операнда.

Модули и прагмы

Новые модули и прагмы

- Был добавлен модуль Config::Perl::V версии 0.16 как “живущий в двух местах” (dual-lifed) модуль. Он предоставляет структурированную выборку данных из вывода `perl -V`, включая информацию, известную только двоичному файлу perl, и недоступную через Config.

Обновлённые модули и прагмы

Для полного списка обновлений запустите:

```
$ corelist --diff 5.16.0 5.18.0
```

Вы также можете заменить нужную вам версию взамен `5.16.0`.

- Archive::Extract был обновлён до 0.68.

  Обход проблемы в некоторых случаях на Linux с unzip из Busybox.

- Archive::Tar был обновлён до 1.90.

  `ptar` теперь поддерживает опцию `-T` также, как и опцию без дефисов [rt.cpan.org #75473], [rt.cpan.org #75475].

  Авто-кодирование имён файлов, помеченных как UTF-8 [rt.cpan.org #75474].

  Не используется `tell` на дескрипторах IO::Zlib [rt.cpan.org #64339].

  Не пытается вызывать `chown` на символических ссылках.

- autodie был обновлён до 2.13.

  `autodie` теперь играет по правилам с прагмой ‘open’.

- B был обновлён до 1.42.

  Был добавлен метод `stashoff` для COPs. Это даёт доступ к внутренним свойствам, добавленным в perl 5.16 в сборках с поддержкой нитей [perl #113034].

  `B::COP::stashpv` теперь поддерживает UTF-8 имена пакетов и встроенные NUL’ы.

  Все `CVf_*` и `GVf_*`, и другие относящиеся к SV флаги теперь доступны как константы в пространстве имён `B::` и доступны для экспортирования. Список экспортируемых символов не изменился.

Теперь модуль может работать с новым pad API.

- B::Concise был обновлён до 0.95.

  Была исправлена опция `-nobanner`, а `format` теперь может быть выведен. Когда указывается имя подпрограммы, для вывода делается также проверка, является ли это именем формата. Если подпрограмма и формат имеют одно и тоже имя, то они оба будут выведены.

  Также была добавлена поддержка новых флагов `OpMAYBE_TRUEBOOL` и `OPpTRUEBOOL`.

- B::Debug был обновлён до 1.18.

  Была добавлена поддержка (экспериментальная) для `B::PADLIST`, которая была добавлена в Perl 5.17.4.

- B::Deparse был обновлён до 1.20.

Убраны предупреждения, если запускается под `perl -w`.

Теперь он разбирает управляющие конструкции циклов в правильном порядке и множественные операции в `format`-строке теперь также разбираются правильно.

В этом выпуске подавляются точки с запятой в конце записи формата.

В этом выпуске добавлена заглушка для разбора лексических подпрограмм.

Он больше не падает при разборе `sort` без аргументов. Теперь он корректно опускает запятые в `system $prog @args` и `exec $prog @args`.

- bignum, bigint и bigrat были обновлены до 0.33.

Переопределение для `hex` и `oct` было переписано, решая несколько проблем и внося одно несовместимое изменение:

- Прежде, какой бы из модулей `use bigint` или `use bigrat` не был скомпилирован позже, он брал приоритет над другими, приводя к тому, что `hex` и `oct` не учитывали действие других прагм в данной области видимости.
- Использование любых из этих трёх прагм приводило к тому, что `hex` и `oct` во всех других местах программы обрабатывали свои аргументы в списочном контексте и препятствовали работе с $_, когда вызывались без аргументов.

- Использование любой из этих трёх прагм заставляло `oct ("1234")` возвращать 1234 (для любого числа, не начинающегося с 0) в любом месте программы. Теперь “1234” транслируются из восьмеричного к десятичному, независимо от того, находится ли оно в области видимости прагмы или нет.
- Глобальное переопределение, которое устанавливает лексическое использование `hex` и `oct`, теперь учитывает любые существующие переопределения, которые действовали до включения нового переопределения, возвращая их вне действия области действия `use bignum`.
- `use bignum "hex"`, `use bignum "oct"` и подобные

вызовы для bigint и bigrat теперь экспортируют функции `hex` или `oct` вместо создания глобального переопределения.

- Carp был обновлён до 1.29.

  Carp больше не приходит в замешательство, если `caller` возвращает undef для пакета, который был удалён.

  Функции `longmess()` и `shortmess()` теперь задокументированы.

- CGI был обновлён до 3.63.

  Нераспознаваемые экранирующие последовательности HTML теперь обрабатываются лучше, проблемные завершающие переводы строк больше не вставляются после тегов `<form>` функциями `startform()`

или `start_form()`, и ложное предупреждение “Insecure Dependency” (небезопасная зависимость) на некоторых версиях perl теперь обходится.

- Class::Struct был обновлён до 0.64.

  Конструктор теперь учитывает переопределённый метод доступа [perl #29230].

- Compress::Raw::Bzip2 был обновлён до 2.060.

  Неправильное использование «магического» API Perl было исправлено.

- Compress::Raw::Zlib был обновлён до 2.060.

  Была обновлена поставляемая с модулем библиотека zlib до версии 1.2.7

  Были исправлены ошибки сборки на Irix, Solaris и Win32, а также сборка с использованием C++

[rt.cpan.org #69985], [rt.cpan.org #77030], [rt.cpan.org #75222].

Неправильное использование «магического» API Perl было исправлено.

`compress()`, `uncompress()`, `memGzip()` и `memGunzip()` были ускорены за счёт более эффективной проверки параметров.

- CPAN::Meta::Requirements был обновлён до 2.122.

  Обрабатываются `undef`-зависимости в `from_string_hash` как 0 (с предупреждением).

  Добавлен метод `requirements_for_mo`

  .

- CPANPLUS был обновлён до 0.9135.

  Разрешено добавлять `blib/script` к PATH.

Сохраняется история между запусками командной строки.

Лучше поддерживаются множественные аргументы для `makemakerargs` и `makeflags`.

Исправлены проблемы с SQLite-движком.

- Data::Dumper был обновлён до 2.145.

  Было оптимизировано создание хеша уже обработанных скаляров только при необходимости, тем самым существенно повышена скорость сериализации.

  Были добавлены дополнительные тесты для улучшения покрытия операторов, веток, условий и подпрограмм. По данным анализа покрытия, некоторые внутренности Dumper.pm были переработаны. Почти все методы теперь задокументированы.

- DB_File был обновлён до 1.827.

  Основной Perl-модуль больше не использует конструкцию "@_".

- Devel::Peek был обновлён до 1.11.

  Были исправлены возможность сборки компилятором C++ и работы с новым pad API.

- Digest::MD5 был обновлён до 2.52.

  Исправлен откат на `Digest::Perl::MD5` в OO-интерфейсе [rt.cpan.org #66634].

- Digest::SHA был обновлён до 5.84.

  Исправлена ошибка с двойным освобождением памяти, которая могла приводить к уязвимости в некоторых случаях.

- DynaLoader был обновлён до 1.18.

Сделано небольшое исправление в XS для VMS-реализации.

Исправлено предупреждение об использовании секции `CODE` без секции `OUTPUT`.

- Encode был обновлён до 2.49.

  Был добавлен алиас x-mac-ce для Mac и исправлены ошибки в `Encode::Unicode`, `Encode::UTF7` и `Encode::GSM0338`.

- Env был обновлён до 1.04.

  Его реализация SPLICE больше не ведёт себя неправильно в списочном контексте.

- ExtUtils::CBuilder был обновлён до 0.280210.

  Файлы манифестов теперь корректно встраиваются для тех версий VC++,

которые используют их. [perl #111782, #111798].

Список экспортируемых символов может быть передан в `link()` на Windows, так же как и на других ОС [perl #115100].

- ExtUtils::ParseXS был обновлён до 3.18.

  Генерируемый C-код теперь избегает ненужного увеличения `PL_amagic_generation` на версиях Perl, где это делается автоматически (или на текущем Perl, где переменная больше не существует).

  Это устраняет ложное предупреждение об инициализированном XSUB без параметров [perl #112776].

- File::Copy был обновлён до 2.26.

  `copy()` больше не обнуляет файлы при копировании в тот же каталог и

теперь завершается с ошибкой (как это было давно задокументировано), когда пытается копировать файл в самого себя.

- File::DosGlob был обновлён до 1.10.

  Внутренний кэш имён файлов, который он поддерживает для каждого вызывающего, теперь освобождается, когда освобождается вызывающий код. Это означает, что в `use File::DosGlob 'glob'; eval 'scalar <*>'` больше не утекает память.

- File::Fetch был обновлён до 0.38.

  Добавлена опция ‘file_default’ для URL, которые не имеют файловой составляющей.

  Используйте `File::HomeDir`, если доступно, и выставляйте `PERL5_CPANPLU` для замены автоопределения.

Всегда запрашивает `CHECKSUMS`, если установлен `fetchdir`.

- File::Find был обновлён до 1.23.

  Была исправлена несовместимая обработка unix-путей на VMS.

  Выборочные файлы теперь могут указываться в списках директорий для поиска [perl #59750].

- File::Glob был обновлён до 1.20.

  В File::Glob сделано то же исправление, что и в File::DosGlob. Поскольку именно его использует собственный оператор `glob` (кроме VMS), это означает, что в `eval 'scalar <*>'` больше нет утечки.

  Список разделённых пробелом шаблонов, возвращающий длинный список результатов, больше не приводит к повреждению памяти или

аварийному завершению. Эта проблема появилась в Perl 5.16.0. [perl #114984]

- File::Spec::Unix был обновлён до 3.40.

  `abs2rel` может вывести некорректный результат, если получает два относительных пути или две корневых директории [perl #111510].

- File::stat был обновлён до 1.07.

  `File::stat` игнорирует прагму filetest и предупреждает, если используется в комбинации с ней. Но предупреждения не выдавалась при использовании `-r`. Это было исправлено [perl #111640].

  `-p` теперь работает и не возвращает ложь для каналов [perl #111638].

  Ранее перегруженные операторы `-x` и `-X` из `File::stat` не давали

правильных результатов для директорий или исполнимых файлов, если были запущены с правами root. Они рассматривали право исполнения для root также как и для всех других пользователей, выполняя проверку вхождения в группу для файлов, не принадлежащих root. Теперь они следуют правильному поведению в Unix — для директорий они всегда истинны, а для файла, если любой из трёх битов разрешения запуска установлен, они сообщают, что root может запустить файл. Встроенные операторы –x и –X всегда работали правильно.

- File::Temp был обновлён до 0.23

  Исправлены различные ошибки, связанные с удалением директорий. Откладывается удаление временных

файлов, если первоначальное удаление завершаются неудачей, это исправляет проблемы на NFS.

- GDBM_File был обновлён до 1.15.

  Недокументированный необязательный пятый параметр к `TIEHASH` был удалён. Он задумывался для предоставления контроля над колбэком, используемым функциями `gdbm*` в случае фатальных ошибок (такие как проблемы файловой системы), но не работал (и никогда не работал). Ни один модуль на CPAN даже не предпринимал попыток его использования. Колбэк теперь всегда по умолчанию `croak`. Были решены проблемы с тем, как на некоторых платформах вызывается C-функция `croak`.

- Hash::Util был обновлён до 0.15.

`hash_unlocked` и `hashref_unlocked` теперь возвращают истину, если хеш разблокирован, вместо того, чтобы постоянно возвращать ложь [perl #112126].

`hash_unlocked`, `hashref_unlocked`, `lock_hash_recurse` и `unlock_hash_re` теперь доступны для экспорта [perl #112126].

Были добавлены две новые функции: `hash_locked` и `hashref_locked`. Как это ни странно, обе этих функции уже экспортировались несмотря на то, что они не существовали [perl #112126].

- HTTP::Tiny был обновлён до 0.025.

  Добавлены возможности проверки SSL [github #6], [github #9].

  Включается финальный URL в хеш-ссылку ответа.

Добавлена опция `local_address`.

Улучшена поддержка SSL.

- IO был обновлён до 1.28.

  `sync()` теперь можно вызывать на доступных только на чтение файловых дескрипторах [perl #64772].

  IO::Socket пытается лучше кэшировать или в противном случае получить информацию из сокета.

- IPC::Cmd был обновлён до 0.80.

  Используется `POSIX::_exit` вместо `exit` в `run_forked` [rt.cpan.org #76901].

- IPC::Open3 был обновлён до 1.13.

  Функция `open3()` больше не использует `POSIX::close()` для закрытия файлового дескриптора, так как это ломает подсчёт ссылок файловых

дескрипторов, который ведёт PerlIO в случаях, если файловые дескрипторы разделены потоками PerlIO, что приводит к попыткам закрытия файлового дескриптора во второй раз, когда любой из подобных PerlIO потоков закрывается позже.

- Locale::Codes был обновлён до 3.25.

  Он включает некоторые новые коды.

- Memoize был обновлён до 1.03.

  Исправлена опция кэша `MERGE`.

- Module::Build был обновлён до 0.4003.

  Исправлена проблема, когда модули без `$VERSION` могли иметь версию ‘0’, указанную в метаданных ‘provides’, которая будет отклоняться в PAUSE.

  Исправлена проблема в PodParser для разрешения цифр в именах модулей.

Исправлена проблема, когда повторное указание аргументов приводит к тому, что они становятся массивами, приводящее к тому, что пути установки превращались в подобное `ARRAY(0xdeadbeef)/lib/Foo.pm`.

Небольшая проблема была исправлена, позволяя использовать разметку вокруг начального “Name” в строке POD “abstract”, и было сделано несколько улучшений в документации.

- Module::CoreList был обновлён до 2.90

  Информация о версии теперь сохраняются в виде дельт, что значительно уменьшает размер файла `CoreList.pm`.

  Восстановлена совместимость со старыми версиями perl и подчищены данные списка базовых пакетов для

различных модулей.

- Module::Load::Conditional был обновлён до 0.54.

  Исправлено использование `requires` для perl, установленного в пути, содержащем пробелы.

  Различные улучшения, включая использование нового модуля Module::Metadata.

- Module::Metadata был обновлён до 1.000011.

  Создание объекта Module::Metadata для типичного файла модуля ускорено примерно на 40%, и некоторые ложные предупреждения о `$VERSION` были подавлены.

- Module::Pluggable был обновлён до 4.7.

  В ряду других изменений теперь разрешено использование триггеров на

событиях, которое даёт большие возможности по изменению поведения.

- Net::Ping был обновлён до 2.41.

  Исправлено несколько падающих тестов на Windows.

- Opcode был обновлён до 1.25.

  Отражено удаление опкода boolkeys и добавление опкодов clonecv, introcv и padcv.

- overload был обновлён до 1.22.

  `no overload` теперь предупреждает об ошибочных аргументах, как и `use overload`.

- PerlIO::encoding был обновлён до 0.16.

  Это модуль, который реализует слой ":encoding(...)" ввода/вывода. Он больше не повреждает память и не падает,

если кодирующий код повторно выделяет буфер или возвращает typeglob или разделяемый скалярный ключ хеша.

- PerlIO::scalar был обновлён до 0.16.

  Передаваемый скаляр буфера теперь может содержать только кодовые точки 0xFF или ниже. [perl #109828]

- Perl::OSType был обновлён до 1.003.

  Исправлена ошибка определения операционной системы VOS.

- Pod::Html был обновлён до 1.18.

  Была восстановлена опция `--libpods`. Она устарела, и её использование ничего не делает, кроме выдачи предупреждения, что она не поддерживается.

  Так как HTML-файлы генерируются с помощью pod2html, который уверяет,

что использует кодировку UTF-8, запись файлов идёт с использованием UTF-8 [perl #111446].

- Pod::Simple был обновлён до 3.28.

  Несколько улучшений было сделано в основном в Pod::Simple::XHTML, который также имеет совместимое изменение: опция `codes_in_verbatim` отключена по умолчанию. Смотрите все детали в `cpan/Pod-Simple/ChangeLog`.

- re был обновлён до 0.23

  Односимвольные классы символов, как `/[s]/` или `/[s]/i` теперь оптимизируются, как если бы они и не имели скобок, т.е. `/s/` или `/s/i`.

  Смотрите замечания о `op_comp` в секции “Внутренние изменения” ниже.

- Safe был обновлён до 2.35.

Исправлено взаимодействие с `Devel::Cover`.

Не выполняется eval кода при действии `no strict`.

- Scalar::Util был обновлён до 1.27.

  Исправлена проблема с `sum`.

  `first` и `reduce` теперь проверяет сначала колбэк (таким образом `&first(1)` теперь не позволяется).

  Исправлен `tainted` на магических значениях [rt.cpan.org #55763].

  Исправлен `sum` на ранее магических значениях [rt.cpan.org #61118].

  Исправлено чтение за пределами фиксированного буфера [rt.cpan.org #72700].

- Search::Dict был обновлён до 1.07.

Больше не требуется `stat` на файловых дескрипторах.

Используйте `fc` для приведения к одному регистру.

- Socket был обновлён до 2.009.

  Были добавлены константы и функции, требуемые для членства в исходных группах IP мультискаста.

  `unpack_sockaddr_in()` и `unpack_sock()` теперь возвращает только IP-адрес в скалярном контексте, и `inet_ntop()` теперь защищён от некорректной длины скаляра, передаваемого ему.

  Исправлено чтение неинициализированной памяти.

- Storable был обновлён до 2.41.

  Изменение `$_[0]` внутри `STORABLE_fre` больше не приводит к краху [perl #112358].

Объект, чей класс реализует STORABLE_a
, теперь размораживается только один раз, когда присутствуют несколько ссылок на него в структуре, которая размораживается [perl #111918].

Ограниченные хеши не всегда размораживались корректно [perl #73972].

Storable будет выдавать ошибку при замораживании благословенной ссылки объекта с методом STORABLE_freeze() [perl #113880].

Теперь он может замораживать и размораживать корректно vstrings. Это приводит к незначительному несовместимому изменению в формате хранения, поэтому версия формата была увеличена до 2.9.

Также содержатся различные исправления, включающие исправления сов-

местимости с ранними версиями Perl и обработки vstring.

- Sys::Syslog был обновлён до 0.32.

  Содержится несколько исправлений ошибок, связанных с `getservbyname()`, `setlogsock()` и уровнями логов в `syslog()`, совместно с исправлениями для Windows, Haiku-OS и GNU/kFreeBSD. Смотрите все детали в `cpan/Sys-Syslog/Changes`

- Term::ANSIColor был обновлён до 4.02.

  Добавлена поддержка для курсива.

  Улучшена обработка ошибок.

- Term::ReadLine был обновлён до 1.10.

  Было исправлено использование оболочек **cpan** и **cpanp** на Windows в случае, если текущий диск содержит файл `\\dev\\tty` .

- Test::Harness был обновлён до 3.26.

  Исправлена семантика glob на Win32 [rt.cpan.org #49732].

  Не используется `Win32::GetShortPath` при вызове perl [rt.cpan.org #47890].

  Игнорируется -T при чтении шебанга [rt.cpan.org #64404].

  Более элегантно обрабатывается случай, когда мы не знаем статуса wait теста.

  Реализована возможность переопределения строчки отчёта теста ‘ok’, и изменение её в плагине, чтобы сделать вывод prove неизменным.

  Не запускаются файлы, доступные для записи для всех.

- Text::Tabs и Text::Wrap были обновлены до 2012.0818.

В оба была добавлена поддержка комбинированных символов Юникода.

- threads::shared был обновлён до 1.31.

  Добавлена опция, которая предупреждает или игнорирует попытки клонировать структуры, которые не могут быть клонированы, вместо безусловного падения в таких случаях.

  Добавлена поддержка переменных с двойным значением, создаваемые с помощью Scalar::Util::dualvar.

- Tie::StdHandle был обновлён до 4.3.

  `READ` теперь учитывает аргумент отступа в `read` [perl #112826].

- Time::Local был обновлён до 1.2300.

  Значения секунд больше 59, но меньше 60 больше не приводят к ошибкам в `timegm()` и `timelocal()`.

- Unicode::UCD был обновлён до 0.53.

  Была добавлена функция all_casefolds(), которая возвращает все случаи регистрового изменения.

- Win32 был обновлён до 0.47.

  Новые API были добавлены для получения и установки текущей кодовой страницы.

Удалённые модули и прагмы

- Version::Requirements был удалён из базового дистрибутива. Теперь он доступен под другим именем: CPAN::Meta::Requirements.

Документация

Изменения в существующей документации

perlcheat

- perlcheat был реорганизован и было добавлено несколько новых секций.

perldata

- Теперь явно документировано поведение инициализации хеша списком, который содержит дублирующие ключи.

perldiag

- Объяснение, почему символические ссылки запрещены при “strict refs”, теперь не предполагает, что читатель знаком с тем, что представляют собой символические ссылки.

perlfaq

- perlfaq был синхронизирован с версией 5.0150040 на CPAN.

perlfunc

- Возвращаемое значение `pipe` теперь задокументировано.

- Прояснена документация `our`.

perlop

- Команды управления циклом (`dump`, `goto`, `next`, `last` и `redo`) всегда имели тот же приоритет, что и операторы присвоения, но это не было задокументировано до этого времени.

Диагностика Следующие дополнения или изменения были сделаны в диагностическом выводе, включая предупреждения или сообщения о фатальных ошибках. Для полного списка диагностических изменений смотрите в perldiag.

Новая диагностика

Новые ошибки

- Unterminated delimiter for here document

  Это сообщение теперь появляется, когда метка встроенного документа имеет начальный символ цитирования, но отсутствует завершающий.

  Это замещает некорректное сообщение об ошибке, что не найдена сама метка [perl #114104].

- panic: child pseudo-process was never scheduled

  Эта ошибка выводится, когда дочерний псевдо-процесс в ithreads реализации на Windows не был запланирован за допустимый период времени и

таким образом не смог быть инициализирован корректно [perl #88840].

- Group name must start with a non-digit word character in regex; marked by <-- HERE in m/%s/

  Эта ошибка была добавлена для `(?&0)`, которая некорректна. Раньше выводилось непонятное сообщение об ошибке [perl #101666].

- [Can't use an undefined value as a subroutine reference](http://search.cpan.org use an undefined value as %s reference)

  Вызов неопределённого значения как подпрограммы теперь выводит данное сообщение об ошибке. Раньше оно присутствовало, но случайно было отключено сначала в Perl 5.004 для немагических переменных, затем в Perl v5.14 для магических (например, связанных) переменных. Теперь оно

было восстановлено. На протяжении этого времени undef рассматривался как пустая строка [perl #113576].

- [Experimental “%s” subs not enabled](http://
“%s” subs not enabled)

  Для использования лексических подпрограмм вы должны сначала включить их:

```
no warnings 'experimental::
    lexical_subs';
use feature 'lexical_subs';
my sub foo { ... }
```

Новые предупреждения

- ‘Strings with code points over 0xFF may not be mapped into in-memory file handles’

- [‘%s’ resolved to ‘\o{%s}%d’](http://search.cp
  resolved to ‘\o{%s}%d’)
- ‘Trailing white-space in a charnames alias definition is deprecated’
- ‘A sequence of multiple spaces in a charnames alias definition is deprecated’
- [‘Passing malformed UTF-8 to “%s” is deprecated’](http://search.cpan.org/perldoc
  malformed UTF-8 to “%s” is deprecated)
- [Subroutine “&%s” is not available](http://se
  “&%s” is not available)

  (Категория closure) Во время компиляции внутренняя именованная подпрограмма или eval пытаются использовать внешнюю лексическую подпрограмму, которая недоступна в данный момент. Это может происходить по двум причинам. Во-первых, лексическая подпрограмма могла

быть объявлена во внешней анонимной подпрограмме, которая ещё не была создана. (Помните, что именованные подпрограммы создаются на этапе компиляции, в то время как анонимные подпрограммы создаются во время выполнения). Например:

```
sub { my sub a {...} sub f { \&a } }
```

Ко времени, когда создаётся f, она не может получить доступ к текущей подпрограмме “a”, так как анонимная подпрограмма ещё не была создана. И наоборот, следующее выражение не выдаст предупреждение, так как анонимная подпрограмма уже была создана и доступна:

```
sub { my sub a {...} eval 'sub f { \&a }' }->();
```

Вторая ситуация вызвана функцией eval, которая обращается к переменной, которая уже ушла из области видимости, например:

```
sub f {
    my sub a {...}
    sub { eval '\&a' }
}
f()->();
```

Здесь, когда `\\&a` внутри `eval` компилируется, `f()` в данный момент не выполняется и поэтому его `&a` не доступна.

- ["%s" subroutine &%s masks earlier declaration in same %s](http://search.cpan.o subroutine &%s masks earlier declaration in same %s)

  (Категория misc) Подпрограмма "my" или "state" была повторно объявлена в текущей области видимости

или операторе, уничтожая доступ к предыдущему экземпляру. Это практически всегда опечатка. Обратите внимание, что ранняя подпрограмма по-прежнему будет существовать до завершения текущей области видимости или пока все замыкания, ссылающиеся на неё, не будут уничтожены.

- The %s feature is experimental

  (Категория experimental) Это предупреждение выводится, когда вы подключаете экспериментальные возможности с помощью `use feature`. Просто подавляйте предупреждение, если вы хотите использовать эту возможность, но делая так, вы берёте на себя риски использования экспериментальной возможности, которая может измениться или будет удалена

в будущей версии Perl:

```
no warnings "experimental::
    lexical_subs";
use feature "lexical_subs";
```

- sleep(%u) too large

  (Категория overflow) Вы вызвали `sleep` с числом, которое больше, чем он может надёжно принять, и `sleep`, вероятно, будет спать меньшее время, чем было запрошено.

- Wide character in setenv

  Попытки поместить широкие символы в переменные окружения через `%ENV` теперь выдают подобное предупреждение.

- “Invalid negative number (%s) in chr”

  `chr()` теперь предупреждает, когда ему передаётся отрицательное значение [perl #83048].

- “Integer overflow in srand”

  `srand()` теперь предупреждает, если переданное значение не помещается в `UV` (так как значение будет обрезано и не произойдёт переполнения) [perl #40605].

- “-i used with no filenames on the command line, reading from STDIN”

  Запуск perl с ключом `-i` теперь выводит предупреждение, если не было указано ни одного входного файла в командной строке [perl #113410].

Изменения в существующией диагностике

- $* is no longer supported

Предупреждение о том, что использование $* и $# больше не поддерживается, теперь выводится для каждого участка кода, который ссылается на них. Раньше оно не выводилось, если другая переменная, использовавшая такой же typeglob, обнаруживалась раньше (например, @* перед $*), а также не выводилось второй и последующих записей. (Достаточно трудно исправить ошибку вывода предупреждения вообще, без того, чтобы не выводить это предупреждение каждый раз, и предупреждение каждый раз согласуется с предупреждением, которое выводится для $[).

- Были добавлены предупреждения для \b{ и \B{. Это предупреждения об устаревших конструкциях, которые должны отключаться по выключению данной категории. Теперь

не требуется также выключать предупреждения регулярных выражений, чтобы добиться их отключения.

- Constant(%s): Call to &{$^H{%s}} did not return a defined value

  Перегрузка константы, которая возвращает `undef` приводит к подобному сообщению об ошибке. Для числовых констант раньше выводилось “Constant(undef)”. “undef” был заменён самим числом.

- Ошибка, которая выводится, когда модуль не может быть загружен, теперь включает подсказку, какой модуль требуется к установке: “Can’t locate hopping.pm in @INC (you may need to install the hopping module) (@INC contains: ...)”

- vector argument not supported with

alpha versions

Это предупреждение не подавлялось даже при помощи `no warnings`. Теперь его можно подавить, и оно было перемещено из категории “internal” в категорию “printf”.

- `Can't do {n,m} with n > m in regex; marked by <-- HERE in m/%s/`

  Эта фатальная ошибка была превращена в предупреждение, которое читается так:

  [Quantifier {n,m} with n > m can’t match in regex](http://search.cpan.org/perldoc?pe {n,m} with n > m can’t match in regex)

  (Категория regexp) Минимальное значение должно быть меньше или равно максимальному. Если вы действительно хотите получить совпадение 0 раз, просто задайте {0}.

- Предупреждение о “Runaway prototype”, которое возникает в редких ситуациях, было удалено из-за бесполезности и противоречивости.
- Ошибка “Not a format reference” была удалена так как единственный случай, в котором она выводилось, являлось внутренней ошибкой.
- Ошибка “Unable to create sub named %s” была удалена по той же причине.
- Ошибка ‘Can’t use “my %s” in sort comparison’ была понижена до предупреждения ‘ “my %s” used in sort comparison’ (со ‘state’ вместо ‘my’ для случая переменных состояния). В дополнение, эвристика для угадывания, какая из лексических ab были неправильно использованы, была улучшена для снижения количества ложных срабатываний. Лексические

ab больше не запрещены, если они вне блока sort. Также именованный унарный или списочный оператор внутри блока sort больше не приводит к игнорированию ab [perl #86136].

Изменения в утилитах

h2xs

- `h2xs` больше не выдаёт неправильный код для пустых определений [perl #20636]

Конфигурация и компиляция

- Добавлена опция `useversionedarchnam` в Configure

  Если она установлена, то в ‘archname’ включается значение ‘api_versionstring’. Например, x86_64-linux-5.13.6-thread-multi. Она выключена по умолчанию.

  Эта возможность была запрошена Тимом Бансом (Tim Bunce), который заметил, что `INSTALL_BASE` создаёт структуру библиотеки, которая не зависит от версии perl. Вместо этого, он помещает архитектурно-зависимые файлы в “*install_base/lib/perl5/*archname”.
  Это затрудняет использование общего `INSTALL_BASE` пути библиотеки с разными версиями perl.

  Устанавливая `–Duseversionedarchnam`

, $archname будет отличаться по архитектуре *and* по версии API, позволяя смешанное использование INSTALL_BASE.

- Добавлена опция PERL_NO_INLINE_FUNC
.

  Если определена PERL_NO_INLINE_FUNC
, то не включается заголовок “inline.h”.

  Это позволяет создавать код, включающий perl-заголовки для определений, без создания зависимости на библиотеку perl при компоновке (которая ещё может не существовать).

- Configure учитывает внешнюю переменную окружения MAILDOMAIN, если она установлена.

- installman больше не игнорирует опцию тишины.

- Оба файла `META.yml` и `META.json` теперь включаются в дистрибутив.

- `Configure` теперь корректно определяет `isblank()` при сборке с помощью компилятора C++.

- Обнаружение утилиты постраничного вывода в `Configure` было улучшено и позволяет указывать опции после имени программы, например, **/usr/bin/less -R**, если пользователь принимает значение по умолчанию. Это позволяет **perldoc** при обработке ANSI-экранирования [perl #72156].

Тестирование

- Тестовый набор теперь содержит секцию тестов, которые требуют

очень большого количества памяти. Эти тесты не запускаются по умолчанию; они могут быть включены установкой переменной окружения `PERL_TEST_MEMORY` со значением количества гибибайт памяти, которое может быть безопасно использовано.

Поддержка платформ

Более неподдерживаемые платформы

- BeOS

  BeOS была операционной системой для персональных компьютеров, разрабатываемой Be Inc, первоначально для их аппаратной платформы BeBox. ОС Haiku была создана как заме-

на с открытыми исходниками для продолжения развития BeOS и его perl-порт на данный момент активно поддерживается.

- UTS Global

  Поддерживающий код, относящийся к UTS global, был удалён. UTS являлся версией System V для мейнфреймов, созданной Amdahl, а затем проданной UTS Global. К порту не притрагивались, начиная с Perl v5.8.0, и UTS Global на текущий момент не функционирует.

- VM/ESA

  Поддержка для VM/ESA была удалена. Порт был протестирован на 2.3.0, который перестал поддерживаться IBM с марта 2002. Для 2.4.0 завершилось обслуживание в июне 2003 и было заменено на Z/VM. Текущая версия Z/VM

V6.2.0 и запланирована к окончанию поддержки к 2015/04/30.

- MPE/IX

  Поддержка для MPE/IX была удалена.

- EPOC

  Поддерживающий код для EPOC был удалён. EPOC являлся семейством операционных систем, разрабатываемых в Psion для мобильных устройств. Он являлся предшественником Symbian. Порт последний раз был обновлён в апреле 2002.

- Rhapsody

  Поддержка для Rhapsody была удалена.

Платформо-специфичные замечания

AIX Configure теперь всегда добавляет `-qlanglvl=extc99` в CC флаги на AIX, если используется xlC. Это позволяет упростить сборку XS модулей, которые предполагают C99 [perl #113778].

clang++ Теперь присутствует обходное решение для ошибки в компиляторе, которая препятствовала сборке с использованием clang++ начиная с Perl v5.15.7 [perl #112786].

C++ При сборке ядра Perl как C++ (что лишь частично поддерживается), математические функции компилируются как `extern "C"`, для гарантированной бинарной совместимости. (Однако, бинарная совместимость в общем случае не гарантируется в любом случае в ситуациях, где бы это имело значение).

Darwin Убрано жёстко заданное выравнивание к 8-байтовым границам для исправления сборки с использованием `-Dusemorebits`.

Haiku Perl теперь должен работать из коробки на Haiku R1 Alpha 4.

MidnightBSD `libc_r` был удалён в последних версиях MidnightBSD, а поздние версии лучше работают с `pthread`. Многопоточность теперь включается с использованием `pthread`, что исправляет сборочные ошибки при сборке с включённой многопоточностью на 0.4-CURRENT.

Solaris Configure избегает запуска команды sed с флагами, которые не поддержива-

ются на Solaris.

VMS

- Где возможно, регистр имён файлов и аргументов командной строки теперь сохраняется за счёт включения возможностей CRTL DECC$EFS_CASE_PRESERVE и DECC$ARGV
  при запуске. Последний действует только если расширенный разбор включён в процессе, из которого запускается Perl.
- Набор символов для расширенного файлового синтаксиса (EFS) теперь включается по умолчанию на VMS. Среди прочего это даёт лучшую обработку точек в именах директорий, нескольких точек в именах файлов и пробелах в файловых

именах. Чтобы получить старое поведение, установите логическое имя DECC$EFS_CHARSET в значение DISABLE.

- Исправлена компоновка на сборках, сконфигурированных с –Dusemymalloc=y.
- Экспериментальная поддержка сборки Perl с помощью компилятора HP C++ доступна при конфигурации с –Dusecxx.
- Все заголовочные файлы C из верхней директории дистрибутива теперь устанавливаются на VMS, согласованно с устоявшейся практикой на других платформах. Ранее устанавливалась только часть, что ломало сборку внешних расширений для расширений, которые зависят от отсутствующих заголовочных файлов.
- Цитирование удалено из команд-

ных слов (но не из параметров) для команд, вызываемых через `system`, обратные кавычки или `open` с перенаправлением. Раньше кавычки на командах пропускались через DCL, который мог не узнать команду. Также, если команда это в действительности путь к образу или командной процедуре на томе ODS-5, цитирование теперь позволяет использовать пути, содержащие пробелы.
- Сборка **a2p** была исправлена для компилятора HP C++ на OpenVMS.

Win32

- Perl теперь может быть собран с использованием компилятора Visual C++ 2012 Microsoft при указании CCTYPE=MSVC110 (или

MSVC110FREE, если вы используете бесплатную Express-версию для десктопа Windows) в `win32/Makefile`.

- Опция для сборки без `USE_SOCKETS_AS_` была удалена.

- Исправлена проблема, когда perl мог упасть при очистке нитей (включая главную нить) в сборках с многопоточностью и отладочными символами на Win32 и, возможно, на других платформах [perl #114496].

- Редкая ситуация гонки, которая приводит к тому, что sleep занимает больше времени, чем запрошено и возможно даже зависает, теперь была исправлена [perl #33096].

- `link` на Win32 теперь пытается установить `$!` в более уместное значение, основанное на коде ошибки Win32

API [perl #112272].

Perl больше не портит блок окружения, например при запуске нового подпроцесса, когда окружение содержит не-ASCII символы. Однако, известные проблемы по-прежнему остаются, если окружение содержит символы вне текущей кодовой таблицы ANSI (например, смотрите блок по Юникоду в `%ENV` в http://perl5.git.perl.org/perl.git/blob/HEAD:/Porting/todo.pod). [perl #113536]

- Сборка perl некоторыми Windows-компиляторами раньше завершалась ошибкой из-за проблем с оператором `glob` в miniperl (который использует программу `perlglob`), удалявшими переменную окружения PATH [perl #113798].

- Была добавлена новая опция для

Windows makefile'ов `USE_64_BIT_INT`. Установите её в значение "define" при сборке 32-битного perl, если вы хотите, чтобы он использовал 64-битные целые.

Уменьшение размера машинного кода, которое уже было выполнено для DLL XS-модулей в Perl v5.17.2, теперь было расширенно и на DLL самого perl.

Сборка с помощью VC++ 6.0 была нечаянно сломана в Perl v5.17.2, но теперь была восстановлена.

WinCE Сборка на WinCE теперь снова стала возможна, однако больше работы требуется для полного восстановления чистой сборки.

Внутренние изменения

- Были созданы синонимы для вводящего в заблуждения имени `av_len()` : `av_top_index()` и `av_tindex`. Все три возвращают самый старший индекс в массиве, а не число элементов, который он содержит.

- `SvUPGRADE()` больше не является выражением. Изначально этот макрос (и используемая в нём функция `sv_upgrade()`) был задокументирован как логический, хотя в реальности они всегда падали на ошибках и никогда не возвращали ложное значение. В 2005 документация была обновлена с указанием, что возвращается пустое значение, но для обратной совместимости `SvUPGRADE()` всегда возвращал 1. Теперь это

было удалено, и `SvUPGRADE()` теперь является оператором, который не возвращает никакого значения.

Таким образом, это теперь синтаксическая ошибка:

```
if (!SvUPGRADE(sv)) { croak(...); }
```

Если у вас есть подобный код, просто замените его на

```
SvUPGRADE(sv);
```

или для избежания предупреждения компилятора со старыми версиями perl возможно

```
(void)SvUPGRADE(sv);
```

- Perl имеет новый механизм копирования при-записи, который позволяет любым SvPOK-скалярам быть улучшенным до скаляра, копируемого-при-записи. Число ссылок на буфер

строки сохраняется в самом буфере строки. Эта возможность **не включена по умолчанию**.

Он может быть включён в сборке perl при помощи запуска `Configure` с флагом **-Accflags=-DPERL_NEW_COPY_**
и мы призываем XS-авторов попробовать свой код на perl, собранным подобном образом, и отправить свои отзывы. К сожалению, пока нет хорошего руководства для обновления XS-кода для работы с COW. Пока такой документ недоступен, обращайтесь за консультациями в почтовый список рассылки perl5-porters.

Он ломает некоторые XS-модули, позволяя копируемым-при-записи скалярам появляться в коде, где никто раньше не ожидал их видеть.

- Копирование-при-записи боль-

ше не использует флаги SvFAKE и SvREADONLY. Следовательно, SvREADONLY теперь показывает по-настоящему доступные только на чтение SV.

Используйте SvIsCOW макрос (как и раньше) для идентификации скаляров, копируемых-при-записи.

- `PL_glob_index` исчез.

- Приватный Perl_croak_no_modify лишился параметра своего контекста. Теперь у него пустой прототип. Пользователи публичного API croak_no_modify остаются незатронутыми.

- Копируемые-при-записи (разделяемый ключ хеша) скаляры больше не помечаются как доступные только для чтения. `SvREADONLY` возвращает

ложь на подобных SV, но `SvIsCOW` по-прежнему возвращает истину.

- Появился новый тип операций `OP_PADRANGE`. Оптимизатор perl, где возможно, заменит на одиночную операцию padrange операцию pushmark с последующей одной или несколькими pad-операциями, и, возможно, пропуская операции list и nextstate. В дополнение, операция может взять задачу, связанную с RHS присвоения `my(...)= @_`, чтобы эти операции также могли быть оптимизированы.

- Регистро-независимое сравнение внутри заключённых в скобки символьных классов с многосимвольной формой больше не исключает одну из возможностей в случаях, когда оно используется именно для этого [perl

#89774].

- `PL_formfeed` был удалён.

- Движок регулярных выражений больше не читает один байт после конца целевой строки. Для всех внутренне корректно сформированных скаляров это никогда не было проблемой, это изменение упрощает применение хитрых трюков с буферами строк в CPAN-модулях [perl #73542].

- Внутри блока BEGIN `PL_compcv` теперь указывает на компилируемую в данный момент подпрограмму, а не на сам блок BEGIN.

- `mg_length` устарела.

- `sv_len` теперь всегда возвращает число байт, а `sv_len_utf8` число символов. Ранее `sv_len` и `sv_len_utf8`

обе были сломанными и иногда возвращали число байт, а иногда число символов. `sv_len_utf8` больше не считает, что её аргумент в кодировке UTF-8. Никто из них больше не создаёт UTF-8-кэши для связанных или перегруженных значений или для не-PV.

- `sv_mortalcopy` теперь копирует строковые буферы скаляров разделяемых ключей хеша, когда вызывается из XS модулей [perl #79824].

- `RXf_SPLIT` и `RXf_SKIPWHITE` больше не используются. Сейчас они определены как 0.

- Новый флаг `RXf_MODIFIES_VARS` может быть установлен в произвольном движке регулярных выражений для указания, что запуск регулярного

выражения может привести у изменению переменных. Это позволяет `s///` знать, когда нужно пропускать определённые оптимизации. Собственный движок регулярных выражений устанавливает этот флаг для специальных команд прохода с возвратом, которые устанавливают $REGMARK$REGERROR.

- API для доступа к лексическим pad'ам было существенно изменено.

  `PADLIST` больше не является `AV`, а имеет свой собственный тип. `PADLIST` теперь содержит `PAD`, а `PADNAMELIST` состоит из `PADNAME`, а не `AV` для pad и списка pad-имён. `PAD`, `PADNAMELIST` и `PADNAME` доступны, соответственно, через появившееся новое pad API, вместо прямого `AV` и `SV` API. Смотрите perlapi для

подробностей.

- В API регулярных выражений функции-колбэки пронумерованных захватов передают индекс, указывающий, к какой переменной осуществлён доступ. Существуют специальные индексные значения для переменных $, $&, $&. Раньше такие же три значения использовались также для получения `${^PREMATCH}`, `${^MATCH}`, `${^POSTMATCH}`, но они теперь получили три отдельных значения. Смотрите “Numbered capture callbacks” in perlreapi.

- `PL_sawampersand` раньше был логическим значением, которое указывало, что любые из $, $&, $&‘ были видимы; теперь он содержит три однобитных флага, указывающих на наличие любой из переменных индивидуаль-

но.

- `CV *` typemap теперь поддерживает `&{}` перегрузку и typeglob'ы, также как и `&{...}` [perl #96872].

- Флаг `SVf_AMAGIC`, указывающий на перегрузку теперь в стэше, а не в объекте. Теперь он устанавливается автоматически при изменении метода или @ISA, так что его смысл теперь изменился также. Теперь он означает "потенциально перегружен". Когда рассчитывается таблица перегрузок, флаг автоматически выключается, если отсутствует перегрузка, поэтому заметного замедления нет.

  Устаревание таблицы перегрузок теперь проверяется, когда вызывается перегруженный метод, а не во время `bless`.

Магический “A” исчез. Изменения в обработке флага `SVf_AMAGIC` устранили необходимость в нём.

`PL_amagic_generation` был удалён и больше не нужен. Для XS-модулей это теперь макрос-псевдоним к `PL_na`.

Установка откат-перегрузки теперь сохраняется в записи стэша отдельно от самой перегруженности.

- Код обработки символов был местами подчищен. Изменения должны быть невидимы на практике.

- Функция `study` была сделана бездейственной в v5.16. Она была попросту выключена вставкой оператора `return`; код же оставался на месте. Теперь поддерживающий код, который использовался в `study`, полностью был удалён.

- В perl с поддержкой нитей больше нет отдельного PV, выделенного для каждого COP для хранения имени его пакета (`cop->stashpv`). Вместо этого есть смещение (`cop->stashoff`) в новый массив `PL_stashpad`, который хранит указатели стэша.

- В расширяемом API регулярных выражений структура `regexp_engine` получила новое поле `op_comp`, которое в данный момент используется только для внутренних целей perl и должно быть инициализировано в NULL другими модулями расширения регулярных выражений.

- Новая функция `alloccopstash` была добавлена в API, но она рассматривается как экспериментальная. Смотрите perlapi.

- Раньше в Perl было реализовано получение магии способом, иногда скрывавшим ошибки в коде, который мог вызывать mg_get() слишком много раз на магические значения. Такого сокрытия ошибок больше не происходит, так что давние ошибки теперь могут стать заметными. Если вы видите относящиеся к магии ошибки в XS-коде, проверьте, чтобы быть уверенными, вместе с другими Perl API-функциями, которые его используют, что вызовы mg_get() происходят только один раз на SvGMAGICAL() значения.

- Выделение OP для CV теперь использует slab-выделение. Это упрощает управление памятью для OP, выделенных для CV, поэтому очистка после ошибок компиляции проще и более безопасна [perl #111462][perl

#112312].

- PERL_DEBUG_READONLY_OPS был переписан для работы с новым slab-распределителем, позволяя захватывать больше нарушений, чем раньше.
- Старый slab-распределитель для OP, который включался только для PERL_IMPLICIT_SYS и PERL_DEBUG_REA, был удалён.

Выборочные исправления ошибок

- Ограничители встроенной документации больше не требуют завершающего символа новой строки, если находятся в конце файла. Это уже бы-

ло так в конце строки, выполняемой через eval [perl #65838].

- Сборка с `-DPERL_GLOBAL_STRUCT` теперь освобождает глобальную структуру **после** того, как закончит использовать её.

- Завершающий '/' в путях в @INC больше не дополняется ещё одним '/'.

- Слой `:crlf` теперь работает, когда возвращаемые данные не вмещаются в свой собственный буфер [perl #112244].

- `ungetc()` теперь поддерживает данные, кодированные в UTF-8 [perl #116322].

- Ошибка в базовых typemap привела к тому, что любые C-типы, которые

соответствуют базовому T_BOOL typemap-элементу не устанавливаются, не обновляются и не модифицируются, когда T_BOOL-переменная используется в OUTPUT: секции с исключением для RETVAL. T_BOOL в INPUT: секции не затронуты. Использование возвращаемого типа T_BOOL в XSUB (RETVAL) не было затронуто. Побочным эффектом исправления этой ошибки стало то, что если T_BOOL указан в секции OUTPUT: (что раньше ничего не меняло для SV) и передаётся доступный только для чтения SV (литерал) в XSUB, происходит ошибка вида Modification of a read-only value attempted" [perl #115796].

- На многих платформах указание имени директории как имени скрипта приводило к тому, что perl не де-

лал ничего и сообщал об успешном запуске. Сейчас он всегда должен сообщать об ошибке и завершаться с ненулевым кодом возврата [perl #61362].

- `sort {undef} ...` при действии фатальных предупреждений больше не падает. Он начал падать в Perl v5.16.

- Стэши, благословлённые друг в друге (`bless \%Foo::, 'Bar'; bless \%Bar::, 'Foo'`), больше не приводят к двойному освобождению. Эта проблема стала возникать в Perl v5.16.

- Несколько утечек памяти было исправлено, большая часть затрагивает фатальные предупреждения и синтаксические ошибки.

- Некоторые несовпавшие регулярные выражения, такие как `'f' =~ /../g`, не сбрасывали `pos`. Кроме того, “совпавшие-лишь-раз” шаблоны (`m?...?g`) не сбрасывали его также, когда вызывались второй раз [perl #23180].

- Несколько ошибок, включающие `local *ISA` и `local *Foo::`, приводившие к устареванию MRO кэшей, были исправлены.

- Определение подпрограммы, когда для её typeglob был задан псевдоним, больше не приводит к устареванию кэшей методов. Эта ошибка появилась в Perl v5.10.

- Локализация typeglob’а, содержащего подпрограмму, когда пакет typeglob’а был удалён из своего родительского

стэша, больше не приводит к ошибке. Эта ошибка появилась в Perl v5.14.

- В некоторых ситуациях `local *method=...` мог не сбросить кэши методов при выходе из области видимости.

- `/[.foo.]/` больше не является ошибкой, но выводит предупреждение (как и раньше) и рассматривается как `/[.fo]/` [perl #115818].

- `goto $tied_var` теперь вызывает FETCH перед тем как решит какого типа этот goto (подпрограмма или метка).

- Переименование пакетов через присвоение glob (`*Foo:: = *Bar::; *Bar:: = *Baz::`) в комбинации с `m?...?` и `reset` больше не приводит к падению сборки с поддержкой

нитей.

- Некоторое число ошибок, относящиеся к присвоению списка хешу, было исправлено. Многие их них включают списки из повторяющихся ключей вида `(1, 1, 1, 1)`.

  - Выражение `scalar(%h = (1, 1, 1, 1))` теперь возвращает 4, а не 2.
  - Возвращаемое значение `%h = (1, 1, 1)` в списочном контексте было неправильным. Раньше возвращало `(1, undef, 1)`, теперь возвращает `(1, undef)`.
  - Теперь Perl выдаёт такое же предупреждение для `($s, %h) = (1, {})`, какое он выдаёт для `(%h)= ({})`, “Reference found where even-sized list expected” (“Найдена ссылка там, где ожи-

дался список с чётным числом элементов”).
  - Несколько дополнительных краевых случаев в присвоении списков хешу было исправлено. Для дополнительных деталей смотрите коммит 23b7025ebc.

- С атрибутами, присвоенными лексическим переменным, больше не утекает память [perl #114764].

- `dump`, `goto`, `last`, `next`, `redo` или `require` с последующим тривиальным словом (bareword) или версией и далее идущим инфиксным оператором больше не является синтаксической ошибкой. Раньше так было для некоторых инфиксных операторов (вроде +), которые имеют другое понимание того, где ожидается элемент [perl #105924].

- `require a::b . 1` и `require a::b + 1` больше не выдаёт ошибочное предупреждение о двусмысленности [perl #107002].

- Вызов метода класса теперь разрешается для любой строки, а не только строки, начинающейся с буквенно-цифрового символа [perl #105922].

- Пустой шаблон, созданный с помощью `qr//`, используемый в `m///`, больше не вызывает поведения “пустой шаблон повторно использует последний шаблон” [perl #96230].

- Связывая хеш во время итерации больше не приводит к утечкам памяти.

- Освобождение связанного хеша во время итерации больше не приводит к утечкам памяти.

- Присвоение списка связанному массиву или хешу, который умирает при операции STORE, больше не приводит к утечке памяти.

- Если хинт-хеш (%^H) является связанным, элемент области видимости во время компиляции (который копирует хинт-хеш) больше не теряет память, если умирает вызов FETCH [perl #107000].

- Вычисление констант больше не вызывает неуместное специальное поведение `split " "` [perl #94490].

- `defined scalar(@array)`, `defined do { &foo }` и схожие конструкции теперь рассматривают аргумент к `defined` как простой скаляр [perl #97466].

- Запуск отладчика, который не определяет глоб `*DB::DB` или не предоставляет подпрограмму заглушку для `&DB::DB`, больше не приводит к краху, а выдаёт сообщение об ошибке [perl #114990].

- `reset ""` теперь соответствует своей документации. `reset` сбрасывает только шаблон `m?...?` при запуске без аргумента. Пустая строка в аргументе теперь не делает ничего. (Раньше это рассматривалось как отсутствие аргумента) [perl #97958].

- `printf` с аргументом, возвращающим пустой список, больше не читает дальше конца стека, приводившего к ошибочному поведению [perl #77094].

- `--subname` больше не выдаёт ошибочных неясных предупреждений

[perl #77240].

- `v10` теперь доступна как метка или имя пакета. Это было нечаянно сломано, когда v-строки были добавлены в Perl v5.6 [perl #56880].

- `length`, `pos`, `substr` и `sprintf` могут быть сбиты с толку связываниями, ссылками и typeglob'ами, если привидение их к строковому виду меняет внутреннее представление в или из UTF-8 [perl #114410].

- `utf8::encode` теперь вызывает FETCH и STORE на связанных переменных. `utf8::decode` теперь вызывает STORE (FETCH уже был вызван).

- `$tied =~ s/$non_utf8/$utf8/` больше не зацикливается в бесконечный цикл, если связанная пере-

менная возвращает строку в Latin-1, разделяемый скаляр ключа хеша, или ссылку, или typeglob, который приводится к строковому виду как ASCII или Latin-1. Это была регрессия, начиная с v5.12.

- `s///` без /e теперь лучше определяет, когда необходимо отказаться от определённых оптимизаций, исправляя некоторые ошибочные ситуации:

  - Совпадение с переменными в определённых конструкциях (&&, ||, .. и другие) в части замены; например, `s/(.)/${1}{$a||$1}/g`. [perl #26986]
  - Псевдонимы при поиске соответствия переменным в части замещения.
  - $REGERROR или $REGMARK в замещении [perl #49190].

  - Пустой шаблон (`s//$foo/`), который приводит к тому, что используется последний успешно совпавший шаблон, в случае если шаблон содержит кодовый блок, который изменяет переменные в замещении.

- Заражённость замещаемой строки больше не влияет на заражённость возвращаемого значения `s///e`.

- Переменная автоматического сброса буферов `$|` создаётся на лету при необходимости. Если это происходило (например, если она была упомянута в модуле или в `eval`), когда текущий выбранный файловый дескриптор был `typeglob` с пустым IO слотом, раньше это приводило к краху [perl #115206].

- Номера строк в конце eval строки больше не удлиняются на единицу [perl #114658].

- Фильтры @INC (подпрограммы возвращаемые подпрограммами в @INC), которые устанавливают $_ в копируемый-при-записи скаляр, больше не заставляют парсер модифицировать этот строковой буфер на месте.

- `length($object)` больше не возвращает неопределённое значение, если объект имеет перегрузку строки, которая возвращает undef [perl #115260].

- Было восстановлено использование `PL_stashcache`, кэша стэша используемого для поиска имён для вызова методов.

Коммит da6b625f78f5f133 в августе 2011 нечаянно сломал код, который получал значения в `PL_stashcache`. Так как это только кэш, всё корректно работало и без него.

- Ошибка “Can’t localize through a reference” (“Не могу локализовать через ссылку”) исчезла в v5.16.0, когда `local %$ref` появлялось на последней строке lvalue подпрограммы. Эта ошибка исчезла для `\local % $ref` в perl v5.8.1. Сейчас она была восстановлена,

- Разбор встроенной документации был существенно улучшен, исправляя несколько ошибок разбора, падений и одной утечки памяти, а также корректируя неверную последовательность номеров строк при определённых условиях.

- Внутри eval сообщение об ошибке для нетерминированных встроенных документах больше не имеет переноса строка в середине [perl #70836].

- Замещение внутри замещающего шаблона (`s/${s|||}//`) больше не смущает парсер.

- Возможно не очень правильное место для разрешения комментариев, но `s//"" # hello/e` всегда работало, *только* если не было null символа перед первым #. Теперь это работает даже при наличии null.

- Неправильный диапазон в `tr///` или `y///` больше не приводит к утечке памяти.

- Строковой eval больше не рассматривает оператор цитирования с точкой

запятой в качестве ограничителя в конце строки (`eval 'q;;'`) как синтаксическую ошибку.

- `warn {$_ => 1} + 1` больше не является синтаксической ошибкой. Раньше парсер сбивался при определённых списочных операторах с последующим анонимным хешем и далее идущим инфиксным оператором, который разделяет свою форму с унарным оператором.

- `(caller $n)[6]` (который возвращает текст из eval) раньше возвращал актуальный буфер парсера. Изменение которого могло привести к краху. Теперь он всегда возвращает копию. Возвращаемая строка больше не содержит “\n;” прикреплённый к концу. Возвращаемый текст также включает встроенную документацию,

которая раньше пропускалась.

- Кэш позиции UTF-8 теперь сбрасывается, при доступе к магическим переменным для избежания рассинхронизации строкового буфера и кэша позиции UTF-8 [perl #114410].

- Были исправлены различные случаи двухкратного получения магии для магических строк UTF-8.

- Этот код (когда отсутствует $&)

```
$_ = 'x' x 1_000_000;
1 while /(.)/;
```

  раньше пропускал копирование буфера по причине производительности, но страдал этим при изменении $1, если оригинальная строка менялась. Теперь это было исправлено.

- Perl теперь больше не использует PerlIO для сообщения об отсутствии памяти, так как PerlIO может попытаться выделить ещё памяти.

- В регулярных выражениях, если что-либо подсчитывается с помощью `{n,m}`, где `n > m`, не может привести к совпадению. Раньше это приводило к фатальной ошибке, но сейчас это просто предупреждение (что нечто не сможет совпасть) [perl #82954].

- Раньше была возможность для форматов, определённых в подпрограммах, которые затем становились неопределёнными и снова определялись, закрываться в переменных в неправильном pad (вновь определённая окружающая подпрограмма), что приводило к краху или ошибкам

“Странная копия”.

- Переопределение XSUB во время работы могло выводить предупреждение с неправильным номером строки.

- Формат %vd sprintf не поддерживает объект версии для альфа-версий. Раньше он выводил сам формат (%vd), когда принимал альфа-версию и также выдавал предупреждение “Invalid conversion in printf” (“Неверный перевод в printf”). Больше этого не происходит, но выдаёт пустую строку на выходе. Также он больше не теряет память в этом случае.

- Вызов `$obj->SUPER::method` в главном пакете может не удаться, если SUPER-пакет уже был вызван другими средствами.

- Создание псевдонима стэша (`*foo:: = *bar::`) больше не приводит к тому, что вызов SUPER игнорирует изменения в методах, или @ISA, или в использует неверный пакет.

- Вызов метода в пакетах, имена которых заканчиваются на ::SUPER, больше не рассматриваются как вызов метода SUPER, что приводило к ошибкам поиска метода. Кроме того, определение подпрограмм в подобных пакетах больше не приводит к их поиску вызовом метода SUPER на содержащий их пакет [perl #114924].

- `\w` теперь совпадает с кодами U+200C (ZERO WIDTH NON-JOINER) и U+200D (ZERO WIDTH JOINER). `\W` больше не совпадает с ними. Это изменение связано с коррекцией в определении Юникода, с чем должен совпадать `\w`.

- `dump LABEL` больше не приводит к утечке этой метки.

- Вычисление констант больше не меняет поведение функций `stat()` и `truncate()`, которые могут работать как с именами файлов, так из дескрипторами файлов. `stat 1 ? foo : bar` теперь рассматривает свой аргумент как имя файла (так как это произвольное выражение), а не как дескриптор “foo”.

- `truncate FOO, $len` больше не откатывается к рассмотрению “FOO” как имя файла, если файловый дескриптор был удалён. Это было сломано в Perl v5.16.0.

- Переопределение подпрограммы после присвоения подпрограммы в глоб и глоб в глоб больше не приводит к

двойному освобождению памяти или паническим сообщениям.

- `s///` теперь превращает v-строки в обычные строки, когда делает подстановку, даже если результирующая строка та же самая (`s/a/a/`).

- Предупреждение о несовпадающем прототипе больше не рассматривает подпрограммы-константы как не имеющие прототипа, тогда как они на самом деле имеют “”.

- Подпрограммы-константы и упреждающая декларация больше не препятствует предупреждениям о несовпадающем прототипе опускать имя подпрограммы.

- `undef` на подпрограмму теперь очищает проверку вызова.

- Оператор `ref` стал терять память на благословенных объектах в Perl v5.16.0. Это было исправлено [perl #114340].

- `use` больше не пытается разбирать свои аргументы как оператор, делая `use constant { ()};` синтаксической ошибкой [perl #114222].

- В сборках с отладочными символами, предупреждения “неинициализированности” внутри форматов больше не приводят к ошибкам утверждения.

- В сборках с отладочными символами, подпрограммы вложенные внутри форматов больше не приводят к ошибкам утверждения [perl #78550].

- Форматы и операторы `use` теперь разрешены внутри форматов.

- `print $x` и `sub { print $x }->()` теперь всегда выдают одинаковый вывод. Для последнего было возможен отказ в замыкании $x, если переменная была неактивна; т.е. если она была определена вне текущей запущенной именованной подпрограммы.

- Точно так же `print $x` и `print eval '$x'` теперь выводят одинаковый вывод. Это также позволяет видеть “my $x if 0” переменные в отладчике [perl #114018].

- Форматы, вызываемые рекурсивно, не затрагивают своих лексических переменных, а каждый рекурсивный вызов имеет свой собственный набор лексических переменных.

- Попытка освободить текущий формат или дескриптор, связанный с

ним, больше не приводит к краху.

- Разбор формата больше не сбивается на фигурных скобках, точках с запятой и низкоприоритетных операторах. Раньше было возможно использовать скобки как разделители формата (вместо `=` и `.`), но только иногда. Точка с запятой и низкоприоритетные операторы в аргументах строк формата больше не заставляют парсер игнорировать возвращаемое значение строки. В строках аргумента формата фигурные скобки могут быть использованы для анонимных хешей, вместо того, чтобы всегда восприниматься как `do` блоки.

- Форматы могут быть теперь вложенными внутри кодовых блоков регулярных выражений и других цитирующих конструкций (`/(?{...})`

`/` и `qq/${...}/`) [perl #114040].

- Форматы больше не создаются после ошибок компиляции.

- В сборках с отладочными символами опция командной строки -DA приводила к краху Perl, начиная с v5.16.0. Это было исправлено [perl #114368].

- Потенциальный сценарий взаимной блокировки, включающий преждевременное уничтожение дочернего процесса, запущенного через псевдо-fork в сборках на Windows с поддержкой нитей, был исправлен. Это решает общую проблему зависания теста `t/op/fork.t` на Windows [perl #88840].

- Код, который генерирует ошибки от `require()`, потенциально мог читать один или два байта до старта имени

файла для файловых имён длиной менее трёх байт и завершающихся на `/\.p?\z/`. Теперь это было исправлено. Обратите внимание, что это в любом случае никогда не происходило с именами модулей, переданных в `use()` или `require()`.

- Обработка путей модулей, переданных в `require()`, стала безопасна в многопоточных приложениях на VMS.

- Неблокирующиеся сокеты исправлены на VMS.

- Документация POD теперь может находиться внутри кода, расположенного в цитированной конструкции вне строки eval. Это работает только внутри eval-строк [perl #114040].

- `goto ''` теперь проверяет на пустые метки, выдавая сообщение об ошибке “goto must have label” (“goto должен иметь метку”) вместо выхода из программы [perl #111794].

- `goto "\0"` теперь умирает с сообщением “Can’t find label” (“Не могу найти метку”), вместо “goto must have label” (“goto должен иметь метку”).

- C-функция `hv_store` раньше приводила к краху, когда использовалась на `%^H` [perl #111000].

- Проверяющий код вызова, привязанный к прототипу замыкания через `cv_set_call_checker`, теперь копируется в замыкание, склонированное из него. Таким образом, `cv_set_call_checker` теперь работает внутри обработчика атрибута замыкания.

- Запись в `$^N` раньше не имела никакого эффекта. Теперь по умолчанию происходит ошибка “Modification of a read-only value” (“Изменение значения доступного только для чтения”), но это может быть переопределено в специальном движке регулярных выражений, как и с `$1` [perl #112184].

- `undef` на глоб управляющего символа (`undef *^H`) больше не выдаёт ошибочного предупреждения о неопределённости [perl #112456].

- Ради эффективности многие операторы и встроенные функции возвращают один и тот же скаляр каждый раз. Lvalue-подпрограммы и подпрограммы в пространстве имён CORE:: в подобной детали реализации допускали сквозную утечку. Подобное происходило с

lvalue-подпрограммами, возвращающими значение `uc`. Теперь значение копируется в подобных ситуациях.

- Синтаксис `method {}` с пустым блоком или блоком, возвращающим пустой список, раньше приводил к краху или использовал случайное значение в стеке вызывавшего. Теперь он выдаёт ошибку.

- `vec` теперь работает с экстремальными отступами (>2 GB) [perl #111730].

- Изменения для перегрузки настроек теперь начинают действовать немедленно, как и начинают действовать изменения в наследовании, которые затрагивают перегрузку. Раньше они начинали действовать только после `bless`.

Объекты, которые были созданы до того как класс имел какую-либо перегрузку, оставались раньше неперегруженными даже если классы получали перегрузку через `use overload` или изменение в @ISA, и даже после `bless`. Это было исправлено [perl #112708].

- Классы с перегрузкой могут теперь наследовать значения отката.

- Перегрузка не учитывала значение отката 0, если перегруженные объекты были на обоих сторонах оператора присвоения, как например += [perl #111856].

- `pos` теперь завершается с ошибкой с хешем или массивом в качестве аргумента, вместо вывода ошибочного предупреждения.

- `while(each %h)` теперь подразумевает `while(defined($_ = each %h))`, как `readline` и `readdir`.

- Подпрограммы в пространстве имён CORE:: больше не падают после `undef *_`, когда вызываются без списка аргументов (`&CORE::time` без скобок).

- `unpack` больше не выдаёт ошибку “‘/’ must follow a numeric type in unpack” (“‘/’ должен следовать за числовым типом в unpack”), когда причина в данных, которые некорректны [perl #60204].

- `join` и `"@array"` теперь вызывают FETCH только однажды при связанном `$"` [perl #8931].

- Некоторые вызовы подпрограмм, сгенерированные компиляцией

базовых операций, затронутые переопределением `CORE::GLOBAL`, делают проверку операций дважды. Проверка всегда неизменна для чистого Perl-кода, но двойная проверка может иметь значение, когда затронуты определённые пользователем проверки вызовов.

- Условие гонки присутствовало раньше в fork, которое могло привести к отправке сигнала в родитель и обработано и в родителе, и в дочернем процессе. Сигналы теперь кратковременно блокируются вокруг fork, чтобы предотвратить это [perl #82580].

- Реализация блоков кода в регулярных выражениях, таких как `(?{})` и `(??{})`, была сильно переработана для удаления целого болота ошибок.

Основные заметные пользователю изменения:

– Блоки кода внутри шаблонов теперь разбираются в тот же проход, что и окружающий код; особо надо отметить, что больше не требуется балансировать скобки — теперь это работает:

```
/(?{  $x='{'  })/
```

Это означает, что это сообщение об ошибке больше не выводится:

```
Sequence (?{...}) not
    terminated or not {}—
    balanced in regex
(Последовательность
    (?{...})
    нетерминированна или
    несбалансированы {} в
регулярном выражении)
```

но можно увидеть новую ошибку:

```
Sequence (?{...}) not terminated with ')'
(Последовательность (?{...}) нетерминированна символом ')' )
```

Также литеральные блоки кода внутри шаблонов во время выполнения компилируются только один раз при компиляции perl-кода:

```
for my $p (...) {
    # этот блок кода 'FOO' компилируется только один раз,
    # в то же время, что и окружающий цикл 'for'
```

```
    /$p{(?{FOO;})/;
}
```

– Лексические переменные теперь нормальные в плане поведения в области видимости, рекурсии и замыканиях. В частности, `/A(?{B})C/` ведёт себя (с точки зрения замыкания) в точности как `/A/ && do { B } && /C/`, в то время как `qr/A(?{B})C/` такой же как `sub {/A/ && do { B } && /C/}`. Таким образом, данный код работает так как вы ожидаете, создавая три регулярных выражения, которые совпадают с 0, 1 и 2:

```
for my $i (0..2) {
    push @r, qr/^(??{$i})$/;
}
```

```
"1" =~ $r[1]; # совпадает
```

– Прагма `use re 'eval'` теперь требуется только для блоков кода, определённых во время исполнения; в частности в следующем, текст шаблона `$r` по прежнему интерпретируется в новом шаблоне и перекомпилируется, но индивидуальные скомпилированные блоки кода внутри `$r` используются повторно вместо повторной компиляции, и `use re 'eval'` больше не требуется:

```
my $r = qr/abc(?{...})def/;
/xyz$r/;
```

– Операторы управления хода выполнения кода больше не

приводят к краху. Каждый блок кода запускается в новой динамической области видимости, поэтому `next` и т.д. не видят любой внешний цикл. `return` возвращает значение из блока кода, а не окружающей подпрограммы.

– Perl обычно кэширует компиляцию шаблонов времени выполнения и не перекомпилирует, если шаблон не меняется, но это теперь отключается, если это требуется для корректного поведения в замыканиях. Например:

```
1 my $code = '(??{$x})';
2 for my $x (1..3) {
3 # перекомпиляция, чтобы
      видеть свежее
      значение $x каждый
```

```
    раз
    $x =~ /$code/;
}
```

- `/msix` и `(?msix)` и прочие флаги теперь применяются в возвращаемое значение из `(??{})`; теперь это работает:

```
"AB" =~ /a(??{'b'})/i;
```

- Предупреждения и ошибки будут появляться из окружающего кода (или, для блоков кода времени выполнения, из eval), а не из `re_eval`:

```
use re 'eval'; $c = '(?{
    warn "foo" })'; /$c
   /;
/(?{ warn "foo" })/;
```

раньше выдавало:

```
foo at (re_eval 1) line
   1.
```

```
2 foo at (re_eval 2) line
      1.
```

а теперь выдаёт:

```
1 foo at (eval 1) line 1.
2 foo at /some/prog line
      2.
```

- Perl теперь может быть собран с использованием любой версии Юникода. В v5.16, это работало для Юникода 6.0 и 6.1, но существовали различные ошибки, если использовались более ранние выпуски; и чем старше выпуск тем больше было проблем.

- `vec` больше не выводит предупреждений о “неинициализированности” в lvalue-контексте [perl #9423].

- Оптимизация, включающая фиксированные строки в регулярных

выражениях, могла вызывать некоторое ухудшение производительности в краевых условиях. Это было исправлено [perl #76546].

- В некоторых случаях включение пустых подшаблонов внутри регулярного выражения (такие как `(?:)` или `(?:|)` могут отключить определённые оптимизации. Это было исправлено.

- Сообщение “Can’t find an opnumber” (“Не могу найти номер операции”), которое выдаёт `prototype`, когда передана строка вида “CORE::nonexistent_key
теперь пропускает UTF-8 и встроенные NUL без изменений [perl #97478].

- `prototype` теперь обрабатывает магические переменные, такие как `$1` так же, как и немагические переменные при проверке префикса CORE::,

а не рассматривает их как имена подпрограмм.

- В сборках perl с поддержкой нитей блок кода, исполняющийся в регулярном выражении, мог портить имя пакета, сохранённое в дереве операций, приводя к ошибочному чтению в `caller` и возможно к краху [perl #113060].

- Ссылка на прототип замыкания (`\&{ $_[1]}` в обработчике атрибута для замыкания) больше не приводит к копированию подпрограммы (или ошибкам утверждения на сборках с отладочной информацией).

- `eval '__PACKAGE__'` теперь возвращает правильный ответ на сборках с поддержкой нитей, если текущий пакет был переименован (как в

`*ThisPackage:: = *ThatPackage ::`) [perl #78742].

- Если пакет удалён кодом, который он вызвал, возможно, что `caller` увидит стековый фрейм, принадлежащий удалённому пакету. `caller` может упасть, если адрес памяти стэша стал использоваться скаляром, а замена происходила в том же скаляре [perl #113486].

- `UNIVERSAL::can` больше не рассматривает свой первый аргумент по-разному в зависимости от того, строка или номер он внутренне.

- `open` с `<&` используется для проверки режима, является ли третий аргумент числом, чтобы решить, рассматривать ли его как файловый дескриптор или как имя дескриптора. Магические переменные, такие как

$1, всегда проваливали проверку на число и рассматривались как имена дескрипторов.

- Обработка в `warn` магических переменных ($1, связывания) претерпела несколько исправлений. `FETCH` вызывается только один раз сейчас для связанного аргумента или связанной $@ [perl #97480]. Связанные переменные, возвращающие объекты, которые имеют строковое представление “”, больше не игнорируются. Связанная $@, которая возвращала ссылку при предыдущем использовании, больше не игнорируется.

- `warn ""` теперь рассматривает $@ с числом внутри также, независимо от того как оно задано `$@=3` или `$@="3"`. Раньше игнорировался первый вариант. Теперь он добавляет “\t...caught”,

как это всегда было с $@="3".

- Числовые операторы на магических переменных (например, $1 + 1) раньше использовали операции с плавающей запятой, даже если целочисленные операции были более подходящими, приводя к потери точности на 64-битных платформах [perl #109542].

- Унарное отрицание больше не рассматривает строку как число, если строка используется как число в определённый момент. Поэтому, если $x содержит строку “dogs”, то −$x вернёт “-dogs”, даже если будет записано как $y=0+$x.

- В Perl v5.14, −'−10' было исправлено возвращать “10”, а не “+10”. Но магические переменные ($1, связанные)

не были исправлены до текущего момента [perl #57706].

- Унарное отрицание теперь рассматривает строки согласованно, независимо от внутреннего флага `UTF8`.

- Была исправлена регрессия, появившаяся в Perl v5.16.0, затрагивающая `tr/_СПИСОКПОИСКА_/_СПИСОКЗАМЕНЫ_/`. Только первый экземпляр имел смысл, если символ появлялся более одного раза в `_СПИСОКПОИСКА_`. В некоторых случаях, последний экземпляр переопределял все предыдущие [perl #113584].

- Регулярные выражения, такие как `qr/\87/`, раньше молча вставляли NUL-символ, так как если бы оно было записано как `qr/\00087/`. Теперь оно будет выполнять поиск как если бы

было записано как `qr/87/`, с сообщением о том, что последовательность `"\8"` не распознаётся.

- `__SUB__` теперь работает в специальных блоках (`BEGIN`, `END` и т.д.).

- Создание нитей на Windows могло теоретически приводить к краху, если выполнялось внутри блока `BEGIN`. Оно по прежнему не работает правильно, но по крайне мере не падает [perl #111610].

- `\&{''}` (с пустой строкой) теперь автоматически создаёт заглушку как и для любого другого имени подпрограммы и больше не выводит ошибку “Unable to create sub” (“Не могу создать подпрограмму”) [perl #94476].

- В v5.14.0 появилась регрессия, которая была сейчас исправлена, в которой некоторые вызовы модуля `re` приводили к порче содержимого `$_` [perl #113750].

- `do FILE` теперь всегда устанавливает или очищает `$@`, даже если файл не может быть прочитан. Это гарантирует, что первоначальное тестирование `$@` (как и рекомендуется в документации) всегда возвращает корректные результаты.

- Итерация по массиву, используемая для конструкции `each @array` теперь корректно сбрасывается, когда `@array` очищается [perl #75596]. Это происходит, например, когда массив глобально присваивается, как в `@array = (...)`, а не когда его **значениям** происходит присваивание.

В определениях XS API это означает, что `av_clear()` теперь сбрасывает итератор.

Это копирует поведение итератора хеша, когда хеш очищается.

- `$class->can`, `$class->isa` и `$class->DOES` теперь возвращают корректные результаты независимо от того, существует ли пакет, на который ссылается `$class` [perl #47113].

- Приходящие сигналы больше не очищают $@ [perl #45173].

- Разрешено определение `my ()` с пустым списком переменных [perl #113554].

- При разборе синтаксиса подпрограммы, определённые после ошибок,

больше не оставляют заглушек [perl #113712].

- Замыкания, не содержащие строковых eval, больше не зависают в содержащихся в них подпрограммах, позволяя переменным, замкнутых в них из внешних подпрограмм, быть освобождёнными, когда внешняя подпрограмма освобождается, даже если внутренняя подпрограмма по-прежнему существует [perl #89544].

- Удвоение дескрипторов файлов, находящихся в оперативной памяти, с помошью режимов открытия “<&=” или “>&=” перестало работать правильно в v5.16.0. Оно приводило к созданию нового дескриптора, ссылающегося на другую скалярную переменную. Это было исправлено [perl #113764].

- Выражения `qr//` больше не падают с теми движками регулярных выражений, которые не устанавливают `offs` во время компиляции регулярного выражения [perl #112962].

- `delete local` больше не приводит к краху с определёнными магическими массивами и хешами [perl #112966].

- `local` на элементы определённых магических массивов и хешей раньше не приводил в порядок удалённые элементы при выходе из области видимости, даже если элементы не существовали до `local`.

- `scalar(write)` больше не возвращает несколько элементов [perl #73690].

- Конвертация строк в значения с плавающей запятой больше не разбирает неправильно некоторые строки при

действии `use locale` [perl #109318].

- Фильтры `@INC`, которые умирают, больше не приводят к утечкам памяти [perl #92252].

- Реализация перегруженных операций теперь вызывается в правильном контексте. Это, помимо прочего, позволяет корректно переопределять `<>` [perl #47119].

- Указание только `fallback`-ключа при вызове `use overload` теперь ведёт себя правильно [perl #113010].

- `sub foo { my $a = 0; while ( $a){ ... } }` и `sub foo { while (0){ ... } }` теперь возвращают одно и тоже [perl #73618].

- Отрицание строки ведёт себя так же при действии `use integer;`, как и без неё [perl #113012].

- `chr` теперь возвращает символ Юникода “Замена символа” (U+FFFD) для -1 независимо от внутреннего представления. -1 раньше скрывался, если аргумент являлся связанным или был представлен внутренне строкой.

- Использование `format` после того, как включающая его подпрограмма была освобождена, приводило к краху, начиная с perl v5.12.0, если на формат ссылалась лексическая переменная из внешней подпрограммы.

- Использование `format` после того, как включающей его подпрограмме было присвоено неопределённое значение, приводило к краху, начиная с perl v5.10.0, если на формат ссылалась лексическая переменная из внешней подпрограммы.

- Использование `format`, определённого внутри замыкания, на который ссылаются лексические переменные из вне, никогда реально не работало, если только вызов `write` не находился внутри замыкания. В 5.10.0 это даже начало приводить к краху. Теперь копия этого замыкания рядом с самым верхом стека вызовов теперь используется для поиска этих переменных.

- Форматы, которые замыкают переменные в специальных блоках, больше не приводят к краху, если существует заглушка с тем же именем, что и у специального блока до того, как специальный блок скомпилирован.

- Парсер больше не путается, рассматривая `eval foo ()` как синтаксиче-

скую ошибку, если ей предшествовала запись `print;` [perl #16249].

- Возвращаемое `syscall` значение больше не обрезается на 64-битных платформах [perl #113980].

- Вычисление констант больше не приводит к тому, что `print 1 ? FOO : BAR` печатает в файловый дескриптор FOO [perl #78064].

- `do subname` теперь вызывает именованную подпрограмму и использует имя файла, которое она возвращает, вместо открытия файла с именем “subname”.

- Поиск подпрограмм с помощью проверочных хуков rv2cv (зарегистрированных XS-модулем) теперь учитываются при определении

является ли `foo bar` вызовом подпрограммы `foo(bar)` или вызовом метода `"bar"->foo`.

- `CORE::foo::bar` больше не обрабатывается специально, допуская прямой вызов глобального переопределения через `CORE::GLOBAL::uc(...)` [perl #113016].

- Вызов неопределённой подпрограммы, чей typeglob был установлен в неопределённое значение, теперь выдаёт привычную ошибку “Undefined subroutine called” (“Вызвана неопределённая подпрограмма”) вместо “Not a CODE reference” (“Не кодовая ссылка”).

- Две ошибки, затрагивающие @ISA, были исправлены. `*ISA = *glob_with` и `undef *ISA; @{*ISA}` предотвратят будущие изменения в @ISA от

обновления внутреннего кэша, используемого для поиска методов. Случай с *glob_without_array являлся регрессией с Perl v5.12.

- Оптимизация регулярных выражений иногда приводит к тому, что `$` с `/m` не совпадает или даёт неверные совпадения [perl #114068].

- `__SUB__` теперь работает в блоке `sort`, когда включающая его подпрограмма предекларирована с помощью синтаксиса `sub foo;` [perl #113710].

- Свойства Юникода применяются только к кодам Юникода, что приводит к некоторым тонкостям, когда регулярное выражение совпадает с такими кодовыми точками. Генерируется предупреждение, чтобы привлечь ваше внимание к этому факту. Однако, это предупреждение

иногда оказывается неуместным, как при синтаксическом анализе программы. Не-юникодные сравнения, такие как `\w` и `[:word:]`, не должны выводить предупреждения, так как их определение не ограничивает их применение только к кодам Юникода. Теперь сообщение генерируется только когда происходит сравнение с `\p{}` и `\P{}`. Пока остаётся проблема [perl #114148], для очень небольшого числа свойств Юникода, которые совпадают только с одной кодовой точкой. Предупреждение не генерируется, если они совпадают с такой кодовой точкой Юникода.

- Предупреждения о неинициализированности, упоминающие элементы хеша, будут упоминать только имя элемента, если он находится не в первой цепочке хеша, из-за ошибки

превышения/занижения на единицу (off-by-one).

- Ошибка в оптимизаторе регулярного выражения может приводить к тому, что многострочный “^” ведет себя некорректно в присутствии переносов строк, например, в таком выражении `"/\n\n" =~ m#\A(?:^/$)#im`, которое не совпадёт [perl #115242].

- Неудавшийся `fork` в списочном контексте больше не приводит к повреждению стека. `@a = (1, 2, fork, 3)` раньше поглощал 2 и присваивал `(1, undef, 3)`, если вызов `fork` завершался неудачно.

- Различные утечки памяти были исправлены, в основном затрагивающие связанные переменные, которые

умирают, классы символов регулярных выражений и блоков кода, а также синтаксические ошибки.

- Присвоение регулярного выражения (`${qr//}`) переменной, которая содержит число с плавающей точкой, больше не вызывает ошибок утверждения на сборках с отладочной информацией.

- Присвоение регулярного выражения скаляру, содержащему число, больше не вызывает последующего приведения к числовому значению, создающему произвольное число.

- Присвоение регулярного выражения магической переменной больше не стирает магию. Это являлось регрессией, начиная с v5.10.

- Присвоение регулярного выражения благословлённому скаляру больше не приводит к краху. Это также являлось регрессией с v5.10.

- Регулярное выражение может теперь быть присвоено связанному хешу и элементу массив с привидением к строковому виду.

- Приведение регулярного выражения к числу больше не приводит к выводу предупреждения о неинициализированном значении.

- Отрицательные индексы массива больше не приводят к тому, что методы EXISTS связанной переменной игнорировались. Это являлось регрессией с v5.12.

- Отрицательные индексы массива больше не приводят к краху на мас-

сивах, связанными с не-объектами.

- `$byte_overload .= $utf8` больше не приводит к двойному кодированию UTF-8, если скаляр с левой стороны создал UTF-8 строку при последнем вызове операции перегрузки.

- `goto &sub` теперь использует текущее значение @_ вместо использования массива, с которым первоначально была вызвана подпрограмма. Это означает, что `local @_ = (...); goto &sub` теперь работает [perl #43077].

- Если отладчик вызывается рекурсивно, то он больше не портит свои собственные лексические переменные. Раньше при рекурсии все вызовы разделяли одинаковый

набор лексических переменных [perl #115742].

- `*_{ARRAY}`, возвращённый из подпрограммы, больше не становится самопроизвольно пустым.

Известные проблемы

- Строки с UTF8-флагом в `%ENV` на HP-UX 11.00 глючат.

  Взаимодействие строк с UTF8-флагом и `%ENV` на HP-UX 11.00 на данный момент очень хитрое в некоторых не до конца диагностированных случаях. Ожидаемы ошибки в тесте `t/op/magic.t`, с непредсказуемым поведением при сохранении широких символов в переменных окружения.

Некролог

Ходжанг Юн (Hojung Yoon) (AMORETTE), 24 года, Сеул, Южная Корея, отправился в свой последний путь 8 мая 2013 со статуэткой ламы и карточкой, подписанной TIMTOADY. Это был прекрасный молодой хакер Perl 5 и 6 и преданный участник Seoul.pm. Он программировал на Perl, говорил о Perl, ел Perl и любил Perl. Мы верим, что он по-прежнему где-то программирует на Perl на своём сломанном ноутбуке IBM. Его будет не хватать нам.

Acknowledgements

Perl v5.18.0 представляет собой примерно 12 месяцев разработки, начиная с Perl v5.16.0, и содержит примерно 400,000 изменённых

строк среди 2,100 файлов от 113 авторов.

Perl продолжает бурно развиваться в своей третьей декаде благодаря активному сообществу пользователей и разработчиков. Известно, что следующие люди содействовали в улучшении того, что стало Perl 5.18.0:

Aaron Crane, Aaron Trevena, Abhijit Menon-Sen, Adrian M. Enache, Alan Haggai Alavi, Alexandr Ciornii, Andrew Tam, Andy Dougherty, Anton Nikishaev, Aristotle Pagaltzis, Augustina Blair, Bob Ernst, Brad Gilbert, Breno G. de Oliveira, Brian Carlson, Brian Fraser, Charlie Gonzalez, Chip Salzenberg, Chris 'BinGOs' Williams, Christian Hansen, Colin Kuskie, Craig A. Berry, Dagfinn Ilmari Mannsåker, Daniel Dragan, Daniel Perrett, Darin McBride, Dave Rolsky, David Golden, David Leadbeater, David Mitchell, David Nicol, Dominic Hargreaves, E. Choroba, Eric Brine,

Evan Miller, Father Chrysostomos, Florian Ragwitz, François Perrad, George Greer, Goro Fuji, H.Merijn Brand, Herbert Breunung, Hugo van der Sanden, Igor Zaytsev, James E Keenan, Jan Dubois, Jasmine Ahuja, Jerry D. Hedden, Jess Robinson, Jesse Luehrs, Joaquin Ferrero, Joel Berger, John Goodyear, John Peacock, Karen Etheridge, Karl Williamson, Karthik Rajagopalan, Kent Fredric, Leon Timmermans, Lucas Holt, Lukas Mai, Marcus Holland-Moritz, Markus Jansen, Martin Hasch, Matthew Horsfall, Max Maischein, Michael G Schwern, Michael Schroeder, Moritz Lenz, Nicholas Clark, Niko Tyni, Oleg Nesterov, Patrik Hägglund, Paul Green, Paul Johnson, Paul Marquess, Peter Martini, Rafael Garcia-Suarez, Reini Urban, Renee Baecker, Rhesa Rozendaal, Ricardo Signes, Robin Barker, Ronald J. Kimball, Ruslan Zakirov, Salvador Fandiño, Sawyer X, Scott Lanning, Sergey Alekseev, Shawn M Moore, Shirakata

Kentaro, Shlomi Fish, Sisyphus, Smylers, Steffen Müller, Steve Hay, Steve Peters, Steven Schubiger, Sullivan Beck, Sven Strickroth, Sébastien Aperghis-Tramoni, Thomas Sibley, Tobias Leich, Tom Wyant, Tony Cook, Vadim Konovalov, Vincent Pit, Volker Schatz, Walt Mankowski, Yves Orton, Zefram.

Список выше, конечно, неполон, так как был автоматически сгенерирован из истории системы контроля версий. В частности, он не включает имена (очень высоко ценимых) помощников, которые сообщали о проблемах в Perl баг-трекер.

Множество изменений, включённых в эту версию, идут от CPAN-модулей, включённых в ядро Perl. Мы благодарны всему CPAN-сообществу за помощь в развитии Perl.

Полный список всех принимавших участие в разработке в истории Perl смотрите пожалуйста в файле AUTHORS в дистрибутиве исходного кода Perl.

Сообщения об ошибках

Если вы найдёте то, что, как вы считаете, является ошибкой, вы можете проверить ранее опубликованные статьи в новостной группе comp.lang.perl.misc и базе ошибок perl на http://rt.perl.org/perlbug/. Также может быть информация на http://www.perl.org/, домашней странице Perl.

Если вы уверены, что у вас ещё никем не сообщённая ошибка, пожалуйста запустите программу perlbug, включённую в ваш

релиз. Убедитесь, что вы привели максимально краткий, но достаточный пример, для воспроизведения проблемы. Ваш отчёт по ошибке, вместе с выводом `perl -V`, будет отправлен на адрес perlbug@perl.org для анализа командой портирования Perl.

Если ошибка, о которой вы сообщаете, имеет отношение к безопасности, что делает ее неуместным для отправки в публичную архивируемую почтовую рассылку, пожалуйста отправьте его на `perl5-security-report@perl.org`. Это неархивируемая почтовая рассылка с закрытой подпиской, которая включает всех главных коммитеров и позволит скоординировать выпуск патча для смягчения или исправления проблемы на всех платформах, на которых поддерживается Perl. Пожалуйста используйте этот адрес только для проблем безопасности в базовом Perl, а

не для модулей, которые распространяются на CPAN.

Смотрите также

Файл `Changes` для просмотра исчерпывающей информации о том, что изменилось.

Файл `INSTALL` о том, как собирать Perl.

Файл `README` для общей информации.

Файлы `Artistic` и `Copying` для информации по правам.

■ *Владимир Леттиев (перев.)*

7 Обзор CPAN за май 2013 г.

Рубрика с обзором интересных новинок CPAN за прошедший месяц.

После успеха кампании по сбору средств для проекта `Pinto`, в этом месяце появилось целых два новых проекта по созданию собственного CPAN-архива: Escort и OrePAN2, а также вышел новый релиз WorePAN.

В связи с выходом нового стабильного релиза Perl 5.18.0, в котором серьёзно изменилась реализация функции хеширования и произошла рандомизация порядка следования ключей хеша, некоторые модули CPAN оказались сломаны из-за этого изменения. Поэтому обновления части модулей в этом месяце содержат упоминание об

исправлениях совместимости с новой версией Perl.

Статистика

- Новых дистрибутивов — 267
- Новых выпусков — 986

Новые модули

- Web-Dash Модуль, который предоставляет веб-интерфейс к Unity Dash, позволяя использовать различные его возможности: локальный поиск (приложения, документы, музыка, видео и т.д.), поиск внешних ресурсов, использование различных «линз»

Unity Dash и т.д.

- Geo::IP2Location Модуль для быстрого получения различной геоинформации по ip-адресу. Используются файлы баз данных `ip2location.com`, имеющих определённые ограничения для бесплатного использования.

- GeoIP2 Perl API для доступа к новому веб-сервису геоинформации `GeoIP2` компании MaxMind.

- Parse::PMFile Модуль позволяет обработать .pm-файлы модулей так же, как это делает PAUSE. Модуль может быть полезен для выполнения проверки вашего дистрибутива перед отправкой в PAUSE, чтобы заранее увидеть возможные проблемы.

- Magpie Фреймворк для быстрой разработки веб-приложений, реализован-

ный в виде слоя `Plack::Middleware`. Основной парадигмой разработки является ресурсно-ориентированное программирование, подразумевающее создание RESTful веб-сервисов.

- Mail::DMARC Perl-реализация спецификации `DMARC` для идентификации почтовых сообщений принимающими серверами с использованием `SPF` и `DKIM`. Основное назначение модуля — фильтрация фишинговых и спам-сообщений.

- Escape::Houdini Perl API к библиотеке `Houdini` для экранирования символов в HTML/XML/JS/URI-документах и строках.

- UAV::Pilot Модуль и утилиты для управления беспилотными воздушными моделями дронов. Модуль

позволяет передавать по Wi-Fi подключению команды для дрона, обеспечивая управление полётом.

- match::smart С выходом Perl 5.18 оператор умного сравнения ~~ был объявлен экспериментальным. В данном модуле создана альтернативная реализация умного сравнения в виде инфиксного оператора `|M|`.

- Image::WebP Интерфейс к библиотеке `libwebp` Google. Позволяет получать информацию об изображении, а также конвертировать изображение из формата webp в набор данных RGB и обратно.

- Linux::ACL Perl-расширение для чтения и записи ACL (списки контроля доступа) файлов с помощью библиотеки `libacl`.

- JSON-MaybeXS Модуль включает использование `Cpanel::JSON::XS` или откатывается, в случае его отсутствия, на `JSON::PP`. Таким образом, он является альтернативой модуля JSON с той лишь разницей, что вместо основного бэкенда `JSON::XS` используется его форк — `Cpanel::JSON::XS`. Форк отличается наличием метода `binary`, позволяющего отключать автоматическое детектирование кодировки UTF-8 в строках и рассматривать их как последовательности байт, а также наличием поддержки Perl 5.6.2.

- App::cpanchanges Утилита, которая позволяет быстро получить список последних изменений (ChangeLog) любого модуля или дистрибутива CPAN.

- experimental Модуль, позволяю-

щий легко задействовать новые экспериментальные возможности (и подавить соответствующие предупреждения).

Обновлённые модули

- Tickit 0.33 Обновился модуль для создания интерактивных консольных интерфейсов.
- HTML::TreeBuilder::LibXML 0.23 Вышло обновление модуля, являющегося совместимой с `HTML::TreeBuilder` и `XPath` реализацией на основе библиотеки `libxml`. Добавлены несколько методов, необходимых для работы модуля `Web::Query::LibXML`.

- XML::RSS 1.52 Модуль для создания RSS-лент. В новой версии исправлены ошибки модуля при работе на новой версии Perl 5.18.0.

- Sys::Virt 1.0.5 Модуль для управления виртуальными машинами через библиотеку `libvirt`. В новой версии добавлена поддержка `libvirt` API 1.0.5.

- Smart::Match 0.007 Модуль со множеством полезных функций для умного сравнения. В новой версии отключаются предупреждения об экспериментальном характере оператора умного сравнения на версиях Perl >= 5.17.11.

- Mojolicious 4.09 Вышел новый мажорный релиз веб-фреймворка `Mojolicious`. В данной версии сделано огромное число изменений, включая несовместимые, и добавлены новые возможности. Исправлена

совместимость с версиями Perl >= 5.17.11.

- JSON::XS 2.34 Обновлён модуль `JSON::XS`. Исправлен тест для работы на Perl 5.18.0, а также изменена логика выделения памяти при сортировке ключей на использование кучи вместо стека для больших хешей.

- GraphViz2 2.09 Модуль-обёртка для Graphviz теперь требует для работы версию Perl не ниже 5.14.

■ *Владимир Леттиев*

8 Интервью с Андреем Шитовым

Андрей Шитов — организатор большинства Perl-мероприятий на территории СНГ и бывшего СССР. За его плечами воркшопы Perl Today, May Perl, Saint Perl, BY Perl, BG Perl, Perl Mova, конференции YAPC::Russia и YAPC::Europe 2011. А впереди киевская YAPC::Europe 2013.

Как и когда начал изучать программирование?

Когда я начинал изучать программирование, компьютеров еще не было :-) Я где-то прочитал про школьный алгоритмический язык (и 1С тогда не было) и пытался что-то нарисовать, перемещая курсор. Потом появился курс информатики в школе (но не в нашей, там компьютеров тоже не было,

а в соседней). Там стояли Синклеры с Бейсиком. Через год компьютеры завезли и в нашу школу, но это были такие дивайсы под названием «Корвет», выпускаемые центром ЭЛЕКС ГКВТИ. Интерфейс был русский, поэтому при загрузке на мониторе (монохромном, кстати) было написано: «Загрузка ОС». Мне эти осы не понравились, и мы с одноклассником попросились ходить на уроки в соседнюю школу.

В университете я впитывал всю литературу, которую мог найти: как минимум, помню про MS-DOS, паскаль, ассемблер, C++, пролог, фортран и Java. Это самообразование в чистом виде, потому что курс программирования был весьма скромным, и один из преподавателей ходил ко мне консультироваться по C++. А с Java меня познакомил мой научный руководитель: он принес одну из первых книг (если вообще не первую)

на русском языке об этом языке.

Кстати, я учился не только программированию, но и тому, как об этом пишут. Как, например, легко получается писать про ассемблер у Питера Нортона (и почему у самого ничего не получается с первого раза). Или о том, как А. Архангельский может писать тысячестраничные книги про любой новый продукт Борланда. Лучшей книгой о языке программирования я до сих пор считаю «Turbo C++: язык и его применение» Цимбала, Майорова и Козодаева.

Какой редактор используешь?

Долгое время пользовался Komodo Edit, но они не сумели вовремя подстроиться под ретину, поэтому пришлось перейти на Sublime Text 2. (Одновременно я открыл

программу ExpanDrive, после чего выбор редактора не упирался в умение работать с файлами по SSH.) На втором месте идут vim и редактор, встроенный в mc. Редактор мне нужен только как программа, в которой можно писать; я никогда не пользуюсь ни плагинами, ни возможностями для просмотра структуры кода или рефакторинга.

Как и когда познакомился с Perl?

Где-то в 2000-м. Во-первых, Perl был на обложках книг в магазине, которые я еще не читал. Во-вторых, один товарищ рассказал мне, что он по работе познакомился с перлом, и там можно писать и так, и сяк, и задом наперед, и все будет работать. Я решил не отставать от моды.

С какими другими языками программи-

рования приятно работать?

Нежно люблю C++. Заинтересовался Go. Хотя ни на том, ни на другом не пишу много. Go, между прочим, очень легко дается тем, кто знаком с перлом.

Когда-то посмотрел видеозаписи выступлений Дугласа Крокфорда и сильно удивился, насколько интересно устройство JavaScript.

Думаю, было бы приятно работать с Perl 6, но.

Какое, по-твоему, самое большое преимущество Perl?

У перла было невероятное преимущество: он шел в комплекте с любым юниксом. Важная черта языка — лаконичность и относительная синтаксическая свобода. В интерне-

те есть замечательный майндмап на эту тему, составленный три года назад Валерием Студенниковым для доклада на конференции May Perl в Москве.

Я не думаю, что новые языки (в частности, динамические) сильно уступают перлу, но перл как-то ближе. Настолько близок, что мне проще говорить о самых больших недостатках перла, чем о преимуществах :-) Немного про это я рассказывал в недавнем подкасте YAPP № 17.

На перле удобно быстро писать прототипы любых сетевых приложений, и время от времени разрабочики это успешно демонстрируют.

И, без сомнения, среди важных достоинств — регулярные выражения и хорошая поддержка юникода (хотя иногда и приходится

танцевать с перекодировками и попытками угадать, выставлен ли у строки utf8-флаг).

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

Возможность решать задачи, которые возникнут в будущем.

Скорее всего, потребуется удобно и просто описывать в программе работу с распределенными (или хотя бы удаленными) ресурсами и параллельность.

Важно, что такие возможности должны быть заложены не только в синтаксисе языка, но и в самом компиляторе. Программа не должна превращаться в код на ассемблере, детально перечисляя все шаги по работе с распределенными данными или кодом. Существенную роль должен

выполнять компилятор.

В большинстве случаев компьютер может распределить ресурсы намного эффективнее, чем это сделал бы обычный программист. Пример этого эффекта — управление памятью. На перле намного проще создавать объекты, не задумываясь о выделении памяти, чем это нужно делать, например, на C. Если язык требует вручную освобождать память, это часто приводит лишь к необходимости быть более внимательным, хотя задача при этом решается с ошибками. Точно так же языки будущего должны самостоятельно заботиться обо всем, что связано с распределенной работой, будь это многопроцессорный компьютер или несколько компьютеров, подключенных к интернету.

Как было организовано сообщество

Moscow.pm? Соответствует ли текущее положение тому, что задумывалось?

Moscow.pm создал не я, а Руслан Закиров. Он, например, первый встречался в Москве с Джонатаном Вортингтоном. Я лишь увеличил численность подписчиков Moscow.pm на 15 000% и начал устраивать воркшопы и конференции.

Кстати, уже пару лет труъ-домен сайта Moscow.pm — moscow.pm.

По поводу соответствия задумке: скорее да, чем нет. В начале у меня было две задачи. Во-первых, познакомить людей друг с другом, во-вторых, организовать парочку мероприятий. Обе эти задачи выполнены вполне успешно. Помимо этого заработала рассылка Mocsow.pm, появились несколько групп в других городах и проводятся хоть

и не регулярные, но и не слишком редкие офлайновые встречи.

Детального плана с датами по развитию группы, разумеется, нет, поэтому все опирается на энтузиазм людей, которые сами хотят что-то сделать. Наличие Moscow.pm значительно облегчает жизнь таким людям, потому что это неплохая точка входа, чтобы оповестить коллег о своих намерениях. Например, в этом году Павел Щербинин предложил собирать офлайновые технические встречи в офисе Mail.ru и хочет сделать их ежемесячными, да еще и с видеотрансляцией.

Что мотивирует тебя на такую активность в организации Perl-мероприятий? Кому и чем они могут быть полезны?

Мне интересно послушать других разра-

ботчиков, поэтому конференции я делал именно так, чтобы мне было интересно туда придти. К тому же я люблю путешествовать, поэтому появились конференции за пределами Москвы.

Организация конференций — весьма интересное дело само по себе. Когда я посещаю мероприятия, я всегда обращаю внимание на организационные детали и стараюсь перенять что-то интересное и избежать чужих ошибок. Ну и вообще интересно было побить рекорды не только по числу организованных мероприятий, но и по числу охваченных стран.

Два года я ждал мероприятия во Владивостоке. Но потом все срослось само собой, и мы провели там дальневосточный Perl-воркшоп, да еще и с зарубежным гостем, Джонатаном Вортингтоном (он сам очень

любит путешествия и не боится ездить в плацкарте, поэтому его совсем не надо было упрашивать).

Примерно тогда же я ходил вокруг карты и очень хотел сделать что-нибудь в Польше. Но тогда это не получилось, а в мае 2013 ребята из Варшавы наконец-то провели первый польский воркшоп.

Перловые мероприятия — это не только сидеть и слушать доклады. Кому-то доклады на конференции, возможно, помогают разобраться в теме, над которой они тоже работают. Но лично мне, например, более интересно послушать о том, чем вообще занимаются люди. Иногда очень полезно бывает узнать о какой-то технологии, готовом решении или просто сайте, о котором ты никогда не задумывался. То есть одна-единственная ссылка или название

продукта, оставшаяся в заметках после получасового выступления, часто оказываются намного полезнее любого рассказа о том, как это работает. Конференция позволяет заглянуть за горизонт своих знаний. Именно заглянуть, а не рассмотреть все в деталях.

Вторая составляющая наших мероприятий — это просто общение, знакомства, обмен идеями и совместный прием пищи.

Почему локальные конференции всегда бесплатные?

Первый воркшоп в Москве был условно-платным. Участникам предлагалось либо прийти бесплатно, либо заплатить столько, сколько они хотят. Этот эксперимент хотя и не был полностью провальным, показал, что в сборе денег за участие нет никакого

смысла.

Чтобы огранизация мероприятия окупила себя только со входных билетов, во многих случаях (прежде всего, когда не удается найти бесплатное помещение), цену надо было бы сделать достаточно высокой. Достаточно настолько, чтобы оттолкнуть людей от участия. Единственный практический смысл от продажи билетов — более точно спрогнозировать численность участников. Билет даже по символической цене в 100-200 рублей (*примерно 2,5-5 евро* — прим. ред.) купит скорее всего именно тот, кто действительно планирует посетить конференцию, а не просто зарегистрироваться на сайте на всякий случай.

Левых людей на бесплатных Perl-мероприятия обычно не бывает, так что отсутствие билета как входного порога ничего не меняет.

А деньги на организацию намного проще найти где-нибудь в другом месте. Один-два спонсора могут покрыть все расходы, и это позволяет избавиться от сложной работы по сбору денег с десятков участников.

С большими конференциями типа YAPC::Europ этот принцип уже не работает. Входные билеты составляют примерно треть от бюджета, а это уже довольно существенная часть. Хотя при большом желании можно было бы сделать бесплатной и YAPC::Europe, но этим мы поломаем традицию и испортим жизнь следующим организаторам, которые будут вынуждены искать деньги вместо того, чтобы работать с докладчиками и делать конференцию максимально полезной.

Несколько раз YAPC::Europe получали на все три дня бесплатное помещение, и

тогда конференция становилась прибыльной, а деньги шли либо на гранты для работы над перлом, либо на спонсирование конференции следующего года. С конференцией в Амстердаме в 2001 году случилась курьезная история, когда ни помещение, ни кейтеринговая компания после неоднократных напоминаний в течение нескольких лет так и не выставили счет организаторам. На этом бюджете возник фонд YAPC::Europe Foundation, где организаторы могут попросить так называемые kick-start donations для своих мероприятий.

Как продвигается организация конференции YAPC::Europe в Киеве? Какие сложности приходится преодолевать при организации мероприятия такого уровня?

Мы начали готовить эту конференцию за

год до ее начала (даже раньше). Причем не просто фантазировать, а именно готовить: уже заключен договор с помещением, арендован корабль, есть договоренность с организацией экскурсий для партнеров. Согласован приезд Ларри Уолла с супругой. Сейчас идет работа над меню для ежедневных кофе-брейков, обедов и фуршета на корабле. Проделана (но еще не закончена) сложная работа с привлечением спонсоров. Этот паззл мы уже собрали настолько, что картинка стала понятной и дальше можно не сверяться с подсказкой. Я очень доволен работой с Вячеславом Тихановским, потому что все, что нужно решать на месте в Киеве, делает именно он.

Поэтому удается расслабиться и делать то, что обычно никто не делает. Например, писать еженедельные посты о подготовке конференции, пиарить нашу конферен-

цию на других мероприятиях, делать для сайта фичи, которых ни у кого нет, думать о том, что нового изобрести для процесса регистрации и о том, какие напечатать футболки.

Заодно удается подумать над тем, чтобы сделать содержание конференции максимально точно соответствующим названию (Future Perl). Программный коммитет работает с авторами докладов, которые уже начали поступать.

Так что трудности приходится не преодолевать, а придумывать :-) Но это только подталкивает на новые выдумки. Например, после того, как я опубликовал вариант футболки с большой надписью Perl 7 и появились громкие негативные отзывы (абсолютно необоснованные на мой взгяд, но все-таки), потребовалось быстро при-

думать что-то, чтобы это преодолеть. Так появился конструктор футболок, где каждый участник может выбрать себе вариант надписи, который ему больше нравится и не противоречит его взглядам. Хотелось бы, чтобы эта идея смогла воплотиться в жизнь.

Что дальше после конференции в Киеве?

Хороший вопрос. Я занимаюсь конференциями с 2007 года, и за это время у меня накопились идеи за пределами и конференций, и перла. Если когда-нибудь отменят визы в Россию, хочется сразу подать заявку на YAPC::Europe в Москве. Я уже знаю, чем займусь после конференции в Киеве, но рассказывать об этом пока не буду.

Организация конференций показала, что

там, где есть активные люди, мероприятия хорошо получаются и без моего участия. Лучший пример здесь — Болгария. Мариан Маринов успешно делает болгарский Perl-воркшоп уже в пятый раз, хотя вместе мы с ним делали только первый из них в 2009 году.

С перлом у меня связаны еще пара задумок (даже три).

Во-первых, хочется заново запустить сайт yapc.tv и превратить его в более полезный и обновляемый ресурс не только с видео, но и с отсортированными по темам презентациями. Кстати, там появилось два новых видео с польского воркшопа.

Во-вторых, моя давняя мечта — сделать онлайн-коллекцию всех книг про Perl, с перекрестными ссылками на разные издания

и переводы. Эдакий старый books.perl.org, только лучше. Проект задуман исключительно как исторический сайт-справка, возможность скачать текст не предусмотрена. Мы уже договорились с Венди ван Дайк, что через месяц-другой попробуем отсканировать обложки книг из ее огромной коллекции. Я, в свою очередь, собрал почти все книги о перле на русском языке.

Еще один онлайновый проект, связанный с конференциями по перлу, я собираюсь показать в Киеве.

Можешь дать несколько советов тем, кто хочет организовать Perl-хакатон или воркшоп у себя в городе?

Прежде всего надо посетить несколько других подобных мероприятий и смотреть на то, как все организовано. Ближайшая

хорошая возможность, разумеется, это конференция YAPC::Europe в Киеве. А дальше нужно найти помещение, открыть сайт и собрать участников и докладчиков и сделать мероприятие интересным в первую очередь для себя. Если у вас найдутся единомышленники, которым тоже будет интересно, то они обязательно придут на ваше мероприятие.

Где сейчас работаешь? Сколько времени уделяешь программированию? Сколько на Perl?

Сейчас я работаю в компании по онлайновому бронированию гостиниц «Островок.ру» и занимаюсь там автоматизацией контекстной рекламы, работой с аффилиатами (партнерами, которые продают наши номера у себя на сайтах) и парой внутренних сервисов по мониторингу. С

разработкой собственно сайта, которым пользуются клиенты, я почти не связан.

Честно говоря, никогда не считал, сколько времени на что уходит. Обычно мне интересно самостоятельно запрограммировать прототип проекта, после чего передать его в руки более опытных в программировании молодых коллег и направлять весь проект в нужное русло. Придумывать задачи и проекты, интересные себе и при этом полезные для компании, — это очень увлекательно.

Большинство кода, к которому я сам приложил руку, написано на перле; есть несколько критичных по времени программ на C++ и парочка на Go. И, разумеется, бесконечное число XSLT-файлов.

Стоит ли сейчас советовать молодым

программистам учить Perl?

Молодым программистам я бы посоветовал знакомиться со всем, на что хватает времени. Не обязательно сразу профессионально изучать все тонкости языков и технологий, потому что первая задача — сформировать кругозор, который позволит выбирать нужное направление по мере поступления реальных задач.

Познакомиться с перлом обязательно нужно, даже если этот язык покажется недостаточно строгим, чтобы на нем серьезно программировать. Молодой программист должен пользоваться своим главным преимуществом — смотреть на мир незашоренными глазами. Будет достаточно, если в один прекрасный день, увидев очередную задачу, он скажет: «О, а это же лучше сделать на перле!»

Вопросы от читателей

Как работалось в студии Лебедева?

Работалось просто замечательно. Это одно из лучших мест в России, а может и в мире, хотя оставаться там на всю жизнь, наверное, не нужно. Сейчас Студия совсем другая (да и до меня она уже успела сильно измениться), но зато работа с Темой оставляет след, и я могу сказать, что уже появилось второе поколение его школы (это когда бывшие работники заражают хорошим людей, которые в Студии никогда не работали).

А еще очень полезно было поработать с другим отцом руского интернета, Антоном Носиком (посмотрите, хотя бы, сколько я здесь по тексту наставил ссылок).

Стоит ли подписываться на рассылку Moscow.pm?

Рассылка Mocsow.pm является, фактически, ExUSSR.pm. Сейчас на нее подписаны 335 человек, и не все из них живут в Москве. У нас есть подписчики из других городов и России, и мира. Интересно, что те, кто подписаны на рассылки других PM-групп, часто читают и Moscow.pm. Есть по крайней мере один подписчик из Австрии, который хотя и не говорит по-русски, но в какой-то степени понимает письменную речь и иногда даже отвечает.

Удивительно, что такой олдскульный формат вполне заменяет общение в других местах: сейчас, например, нет более или менее живых форумов про перл на русском языке (не говоря уже про G+), поэтому письменное общение на Moscow.pm относительно

оживленное, и я бы сказал, что это единственное место, где сейчас собраны активные люди.

Как дополнительный бонус, рассылка иногда становится онлайновой встречей Moscow.pm. Например, почти любой пост с вакансией вызывает поток ответов с другими вакансиями или троллингом по поводу условий работы и зарплаты.

Кстати, работодатели часто сами не подписываются на рассылку, но пытаются отправить в нее сообщение. Как они читают ответы, если люди отвечают не личным письмом, а в общий лист, — загадка.

Еще одно полезное применение рассылки — узнать об офлайновой встрече (или предложить собрать ее). Так что есть смысл подписаться уже для того, чтобы знать, когда и

где участников можно увидеть в реале.

Когда появится украинская версия shitov.ru?

У меня и русская-то версия давно не обновлялась. Версии сайта на других языках — одностраничные страницы, где висит какой-то текст типа «Привет, меня зовут так-то», я обновляю их, когда могу на этом языке написать что-то осмысленное без онлайн-переводчика. Но с украинским языком я познакомился еще до интернета, поэтому нет и украинской версии.

Как тебе Ларри Уолл в общении?

Приятный, скромный человек, но при этом всегда чувствуется, что перл ему действительно очень интересен. В том числе он очень трепетно относится к работе над Perl

6 и хочет, чтобы он получился.

Приезжайте в Киев и пообщайтесь лично: это просто. Минимальный вариант — попросите с ним сфотографироваться.

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

После своего первого визита на YAPC::Europe в Вене в 2007 году я пропустил только одну из этих конференций, прошлогоднюю во Франкфурте. Так что в этом году обязательно наверстаю упущенное.

Моя компания хочет спонсировать конференцию. Что мне делать?

Написать о желании на mail@yapcrussia.org. Причем тянуть с этим не надо, конференция уже на носу.

Явное спонсорство — не единственная возможность. Если вы забронируете гостиницу, используя сервис на сайте конференции, вы чуть-чуть, но увеличите наш бюджет, при том, что для вас гостиница будет стоить меньше, чем на любом другом сайте.

Но даже простое участие — уже само по себе помощь для развития и перла, и сообщества.

■ *Вячеслав Тихановский*

9 Perl Quiz

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

Ответы из предыдущего выпуска: 1) 3, 2) 2, 3) 3, 4) 2, 5) 2, 6) 3, 7) 5 (на момент текущего выпуска уже 80!), 8) 3, 9) 5, 10) 2.

1. Почему в 5.18 порядок хешей стал случайным при каждом вызове?

    1. Вызвано ошибкой
    2. Сделано для улучшения безопасности
    3. Просто так

    4. Никто не знает почему

2. Какую переменную нужно установить, чтобы отключить случайный порядок ключей в хешах?

    1. `PERL_PERLTURB_KEYS`
    2. `PERL_NO_RANDOM_KEYS`
    3. `PERL_SWITCH_OFF_RANDOMNESS`
    4. `PERL_MAKE_ME_GOOD`

3. Поддержка какой версии Unicode реализована в 5.18?

    1. 6.0
    2. 6.1
    3. 6.2
    4. 8

4. Что содержит новая переменная `\$\{LAST\_FH\}`?

    1. Последний файловый дескриптор на запись
    2. Последний файловый дескриптор на чтение
    3. Последнее значение переменной FH
    4. Нет такой переменной

5. Что происходило при переходе на метку, содержащуюся в переменной, до 5.18?

    1. Все работало
    2. Она рассматривалась как пустая строка
    3. Синтаксическая ошибка
    4. Segmentation fault

6. Какой символ в 5.18 стал пробельным?

    1. Точка

2. Табуляция
3. Тире
4. Вертикальная табуляция

7. Что происходит с перменными окружения в 5.18?

    1. Они не работают
    2. Их значения переводятся в байты
    3. Их значения переводятся в символы
    4. Ничего не поменялось

8. Почему из 5.20 будут удалены многие модули?

    1. Они всем надоели
    2. Они плохого качества
    3. Они создают проблемы при сборке Perl
    4. Их никто не хочет поддерживать

9. Отношения Perl и Plan 9.

    1. Слишком плохой фильм
    2. Слишком плохой язык
    3. Слишком старая платформа
    4. Слишком ненадежная поддержка

10. Должен ли Perl-программист знать, что такое символические ссылки в соответствии с документаций 5.18?

    1. Да
    2. Нет
    3. Должен не только знать, но и использовать!
    4. `use strict` выкинут из ядра

■ *Вячеслав Тихановский*